Lasse Bergroth

Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen

Turku Centre *for* Computer Science

TUCS Dissertations
No 146, June 2012

Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen

Lasse Bergroth

Esitetään Turun yliopiston matemaattis-luonnontieteellisen tiedekunnan luvalla julkisesti tarkastettavaksi Turun yliopiston Salon toimipisteen pienessä auditoriossa kesäkuun 20. päivänä 2012 kello 12

Turun yliopisto
Informaatioteknologian laitos
Joukahaisenkatu 3 – 5, IV kerros
20014 Turun yliopisto

2012

Ohjaajat

Professori Olli Nevalainen
Turun yliopisto
Informaatioteknologian laitos
Joukahaisenkatu 3 – 5 B, huone 4088
20014 Turun yliopisto

Lehtori Jukka Teuhola
Turun yliopisto
Informaatioteknologian laitos
Joukahaisenkatu 3 – 5 B, huone 4087
20014 Turun yliopisto

Tarkastajat

Tutkijatohtori Leena Salmela
Helsingin yliopisto
Tietojenkäsittelytieteen laitos
Gustaf Hällströmin katu 2b, huone B230
PL 68, 00014 Helsingin yliopisto

Yliassistentti Kimmo Fredriksson
Itä-Suomen yliopisto, Kuopion kampus
Tietojenkäsittelytieteen laitos, huone D2026
Technopolis, Microkatu 1, D-siipi
PL 1627, 70211 Kuopio

Vastaväittäjä

Yliopistotutkija Juha Kärkkäinen
Helsingin yliopisto
Tietojenkäsittelytieteen laitos
Gustaf Hällströmin katu 2b, huone B214
PL 68, 00014 Helsingin yliopisto

ISBN 978-952-12-2756-1

Tiivistelmä

Tämä tutkielma kuuluu merkkijonoalgoritmiikan piiriin. Merkkijono *S* on merkkijonojen *X*[1..*m*] ja *Y*[1..*n*] yhteinen alijono, mikäli se voidaan muodostaa poistamalla *X*:stä 0..*m* ja *Y*:stä 0..*n* kappaletta merkkejä mielivaltaisista paikoista. Jos yksikään *X*:n ja *Y*:n yhteinen alijono ei ole *S:*ää pidempi, sanotaan, että *S* on *X*:n ja *Y*:n pisin yhteinen alijono (lyh. PYA). Tässä työssä keskitytään kahden merkkijonon PYAn ratkaisemiseen, mutta ongelma on yleistettävissä myös useammalle jonolle.

PYA-ongelmalle on sovelluskohteita – paitsi tietojenkäsittelytieteen niin myös bioinformatiikan osa-alueilla. Tunnetuimpia niistä ovat tekstin ja kuvien tiivistäminen, tiedostojen versionhallinta, hahmontunnistus sekä DNA- ja proteiiniketjujen rakennetta vertaileva tutkimus. Ongelman ratkaisemisen tekee hankalaksi ratkaisualgoritmien riippuvuus syötejonojen useista eri parametreista. Näitä ovat syötejonojen pituuden lisäksi mm. syöttöaakkoston koko, syötteiden merkkijakauma, PYAn suhteellinen osuus lyhyemmän syötejonon pituudesta ja täsmäävien merkkiparien lukumäärä. Täten on vaikeaa kehittää algoritmia, joka toimisi tehokkaasti kaikille ongelman esiintymille.

Tutkielman on määrä toimia yhtäältä käsikirjana, jossa esitellään ongelman peruskäsitteiden kuvauksen jälkeen jo aikaisemmin kehitettyjä tarkkoja PYA-algoritmeja. Niiden tarkastelu on ryhmitelty algoritmin toimintamallin mukaan joko rivi, korkeuskäyrä tai diagonaali kerrallaan sekä monisuuntaisesti prosessoiviin. Tarkkojen menetelmien lisäksi esitellään PYAn pituuden ylä- tai alarajan laskevia heuristisia menetelmiä, joiden laskemia tuloksia voidaan hyödyntää joko sellaisinaan tai ohjaamaan tarkan algoritmin suoritusta. Tämä osuus perustuu tutkimusryhmämme julkaisemiin artikkeleihin. Niissä käsitellään ensimmäistä kertaa heuristiikoilla tehostettuja tarkkoja menetelmiä.

Toisaalta työ sisältää laajahkon empiirisen tutkimusosuuden, jonka tavoitteena on ollut tehostaa olemassa olevien tarkkojen algoritmien ajoaikaa ja muistinkäyttöä. Kyseiseen tavoitteeseen on pyritty ohjelmointiteknisesti esittelemällä algoritmien toimintamallia hyvin tukevia tietorakenteita ja rajoittamalla algoritmien suorittamaa tuloksetonta laskentaa parantamalla niiden kykyä havainnoida suorituksen aikana saavutettuja välituloksia ja hyödyntää niitä.

Tutkielman johtopäätöksinä voidaan yleisesti todeta tarkkojen PYA-algoritmien heuristisen esiprosessoinnin lähes systemaattisesti pienentävän niiden suoritusaikaa ja erityisesti muistintarvetta. Lisäksi algoritmin käyttämällä tietorakenteella on ratkaiseva vaikutus laskennan tehokkuuteen: mitä paikallisempia haku- ja päivitysoperaatiot ovat, sitä tehokkaampaa algoritmin suorittama laskenta on.

Asiasanat: merkkijonoalgoritmit, pisin yhteinen alijono, PYA-heuristiikat

Sammandrag

Denna avhandling tillhör området strängalgoritmik. En sträng S är strängarnas $X[1..m]$ och $Y[1..n]$ gemensam delsekvens, om den kan bildas genom att avlägsna från X $0..m$ och från Y $0..n$ stycken symboler från godtyckliga ställen. Om ingendera delsekvensen av X och Y är längre än S, kallas S den längsta gemensamma delsekvensen av X och Y (förk. LGD). Inom detta arbete koncentreras i lösande av LGD-problemet för två strängar, med problemet kan generaliseras även för flera strängar.

LGD-problemet tillämpas inte enbart inom informationsbehandling utan även inom bioinformatik. Problemets mest kända tillämpningar är text- och bildkompression, filernas versionshantering, mönsteridentifiering samt DNA- och proteinkedjornas struktur jämförande forskning. Problemets lösande försvårar LGD-algoritmernas beroende av inmatade strängars flera olika parametrar. Till dessa hör utöver strängarnas längd bl. a. den använda alfabetens storlek, de inmatade strängarnas teckenfördelning, LGDs relativa andel av den kortare strängens längd samt antalet matchande teckenpar. Därigenom är det svårt att utveckla en algoritm, som skulle fungera effektivt för alla problemets förekomster.

Avhandlingen skall å ena sidan fungera som handbok, där det efter förklaringen av problemets grundbegrepp förevisas tidigare utvecklade exakta LGD-algoritmer. Deras betraktande har grupperats enligt algoritmens funktionssätt till antingen en rad, en kontur eller en diagonal på gång eller mångriktat behandlande. Därtill presenteras en övre eller nedre gräns för LGDs längd räknande heuristiska metoder, vilkas resultat kan utnyttjas antingen som sådana eller för styrning av exakta algoritmers utförande. Detta avsnitt baserar sig på artiklar utgivna av vår forskningsgrupp. I dessa publikationer behandlas för första gång exakta metoder förstärkta med en heuristik.

Å andra sidan innehåller arbetet en tämligen omfattande empirisk del, vars målsättning har varit att effektivera existerande exakta algoritmers exekveringstid och minnesförbrukning. Till ifrågavarande mål har strävats programmeringstekniskt genom att introducera väl understödande datastrukturer för algoritmernas funktionsmodell och genom att begränsa algoritmernas resultatlösa räkning genom att förbättra deras förmåga att iaktta under räkneproceduren nådda mellanresultat och utnyttja dem.

Som avhandlingens slutsatser kan i allmänhet konstateras att de exakta metodernas heuristiska förprocessering nästan systematiskt förminskar deras körningstid och särskilt minnesbehov. Därtill har datastrukturen tillämpad i algoritmen en avgörande inverkan på räkningens effektivitet: ju lokalare sök- och uppdateringsoperationerna är, desto effektivare är algoritmens prestation.

Ämnesord: strängalgoritmer, längsta gemensamma delsekvens, LGD-heuristiker

Abstract

The topic of this thesis belongs to string algorithmics. A string denoted by S is a common subsequence of strings $X[1..m]$ and $Y[1..n]$, if it can be extracted by deleting arbitrarily $0..m$ symbols from X and $0..n$ symbols from Y. If no common subsequence of X and Y is longer than S, it is determined that S is the longest common subsequence (abbr. LCS) of X and Y. In this work, solving the LCS problem for two strings will be concerned, but the problem can also be generalized for several strings.

LCS problem has applications not only in computer science but also in research areas of bioinformatics. To the best known applications belong text and image compression, version maintenance of files, pattern recognition and comparison research of DNA and protein structures. Solving the problem is difficult, because the algorithms depend on several parameters of the input strings. To those belong, for instance, length of the input strings, size of the input alphabet, character distribution of the inputs, proportion of LCS in the shorter input string and amount of matching symbol pairs. Thus it is difficult to develop an algorithm which could run effectively for all problem instances.

The thesis should on the one hand be regarded as a handbook, where, after the description of basic concepts, already earlier developed exact LCS algorithms will be introduced. They are considered in groups which are classified according to the processing model of the algorithm: one row, contour or one diagonal at a time or multi-directedly. In addition, heuristic methods for calculating an upper or a lower bound for LCS will be represented. The results calculated by those methods can be utilized either as such or in order to steer the processing of an exact algorithm. This section is based on articles published by our research group. Exact LCS algorithms enforced with heuristic preprocessing are introduced for the first time in those articles.

On the other hand, the work contains quite a comprehensive section of empirical research, which aims at intensifying the running time and reduction of the memory consumption of existing exact algorithms. This target was tried to be reached in terms of programming techniques by introducing well-supporting data structures for the processing model of the algorithms, and by restricting fruitless calculation performed by the algorithms by improving their capability to detect intermediate results obtained once during the running time and to utilize them.

As conclusions of the thesis, it can be generally mentioned that the heuristic preprocessing almost systematically reduces the running time and especially the memory need of exact LCS algorithms. Furthermore, the data structure of the algorithms has a crucial influence on the efficiency of the calculation: the more local the search and update operations are, the more efficient is the calculation of the algorithm.

Key words: string algorithms, longest common subsequence, LCS heuristics

Anotacija

Šis darbas priklauso sekų algoritmikos sričiai. Seka *S* yra sekų *X*[1..*m*] ir *Y*[1..*n*] bendras posekis, jeigu ją galima sudaryti panaikinant iš laisvai pasirinktų vietų 0..*m* ženklus iš *X* ir 0..*n* ženklus iš *Y*. Jeigu joks sekų *X* ir *Y* bendras posekis nėra ilgesnis už *S*, jis yra sekų *X* ir *Y* bendras ilgiausias posekis (sutr. BIP). Šiame darbe dėmesys sutelkiamas į dviejų sekų BIP-o sprendimą, bet panašus problemos sprendimas gali būti pritaikomas ir tada, kai sekų yra daugiau negu dvi.

BIP-o problema yra pritaikoma ne vien tik kompiuterių mokslui, bet ir bioinformatikos sričiai. Geriausiai žinomas jos pritaikymas teksto ir vaizdų kompresijai, failų versijų valdymui, šabloniniam atpažinimui bei DNR-ų ir proteinų grandinių struktūrų palyginamajam tyrimui. Problemos sprendimą komplikuoja BIP-o algoritmų priklausomybė nuo daugybės įvestų sekų įvairiausių parametrų. Šiems parametrams priklauso, pavyzdžiui, įvestų sekų ilgiai, taikomos abėcėlės dydis, sekų simbolių skirstinys, BIP-o procentinė dalis trumpesnėje sekoje ir sutampančių simbolių porų skaičius. Todėl yra sudėtinga išvystyti algoritmą, kuris veiktų efektyviai bet kuriems problemos atvejams.

Viena vertus, darbas gali būti panaudotas kaip vadovėlis, kur po problemos pagrindinių sąvokų pristatymo supažindinama su jau anksčiau išvystytais tiksliais BIP-o algoritmais. Jų svarstymas yra sugrupuotas pagal algoritmo veiksmo būdą: tvarkoje į vienos eilės, sutapimų klasės arba įstrižainės sutapimų per kartą ieškančius, bei įvairiomis kryptimis apdorojančius algoritmus. Be to, darbe supažindinama su BIP-o ilgio viršutinę arba apatinę ribą skaičiuojančiais euristiniais metodais, kurių skaičiavimo rezultatai gali būti panaudoti ar iškart, arba, alternatyviai, skirti tikslaus algoritmo veiksmo reguliavimui. Ši darbo dalis remiasi mūsų tyrimo grupės narių straipsniais. Juose pirmą kartą nagrinėjami euristiniais metodais pagerinti tikslūs BIP-o algoritmai.

Kita vertus, darbo sudėtyje yra ir gana plati empirinių tyrimų dalis. Tyrimų tikslas buvo sutrumpinti dabartinių tiksliųjų algoritmų vykdymo laiką ir sumažinti jų atminties poreikį. Paminėtas tikslas buvo siekiamas programavimo technikos metodais, atrenkant į algoritmų veiksmo būdą gerai tinkamas duomenų struktūras, ir apribojant algoritmų daromą nerezultatyvų skaičiavimą pagerinant jų sugebėjimą pastebėti vykdymo metu jau suskaičiuotus preliminarius rezultatus bei jais naudotis.

Darbo išvadose apskritai teigiama, kad tiksliųjų BIP-o algoritmų išankstinis euristinis skaičiavimas beveik sistemingai sutrumpina jų vykdymo laiką ir ypač jų atminties poreikį. Toliau, algoritme vartojama duomenų struktūra turi lemiamą poveikį skaičiavimo efektyvumui: kuo lokalesnės paieškos ir aktualizavimo operacijos, tuo efektyvesnis yra algoritmo skaičiavimas.

Pagrindiniai žodžiai: sekų algoritmai, bendras ilgiausias posekis, BIP-o euristikos

Esipuhe

Käsillä olevaan väitöskirjaan johtaneen työn voidaan katsoa alkaneen jo vuonna 1994, jolloin ensimmäisen kerran tutustuin lähemmin merkkijonoalgoritmeihin ja erityisesti kahden merkkijonon pisimmän yhteisen alijonon ongelmaan. Tämä tapahtui lehtori *Timo Raidan* vetämässä maisteriopintojen tutkimusseminaarissa Turun yliopistossa. Aihe tuntui alusta lähtien mielenkiintoiselta ja hedelmälliseltä myöhempiäkin opinnäytetöitäni ajatellen. Mielenkiintoni kohdistumista merkkijonojen tutkimiseen edesauttoi paljolti Timo Raidan asiantunteva ja innostava ote aiheeseen. Myöhemmin, tultuani valituksi tietojenkäsittelyopin henkilökuntaan, yhteistyöni hänen kanssaan jatkui, ja muodostimme tuolloin kolmen hengen tutkimusryhmän yhdessä assistentti *Harri Hakosen* kanssa. Erityisesti monet heuristisista PYA-menetelmistä on kehitetty ryhmän yhteistyön tuloksina. Haluankin kiittää sittemmin vuonna 2002 jo edesmennyttä Timo Raitaa sekä tästä eteenpäin yhä kolleganani jatkanutta Harri Hakosta rakentavasta yhteistyöstä PYA-ongelman tutkimisessa. He olivat kumpikin taustavoiminani myös osallistuessani ensimmäisiin tieteellisiin konferensseihin esitelmöitsijänä.

Lisäksi haluan kiittää väitöskirjani kahta ohjaajaa, professori *Olli Nevalaista* ja lehtori *Jukka Teuholaa* sekä työn valvojaa, professori *Aulis Tuomista*. Työn edetessä Olli Nevalainen käytti runsaan työpanoksen työn rakenteen ohjaamiseen sekä sisällön oikolukuun ja tarkastamiseen, ja Jukka Teuholalta sain asiantuntijan kommentteja työn sisältöön ennen tutkielman antamista tarkastettavaksi. Näiden kahden ohjaajan johdolla työ saatiin etenemään johdonmukaisesti ja ilman suuria viiveitä. Aulis Tuomisen ansiota on pitkälti se, että olen pystynyt suorittamaan tutkimuksiani kotipaikkakunnallani: Turun yliopiston *Salon toimipisteessä*.

Lämpimät kiitokseni kuuluvat lisäksi ystävälleni *Birutė Savickienėlle*, joka oikoluki työn liettuankielisen tiivistelmän. Lopuksi haluan mainita työn tärkeinä taustatukina vanhempani *Eila* ja *Aimo Bergrothin*, jotka ovat aina olleet tarmolla ja kannustavin ajatuksin tukemassa tieteellisiä pyrkimyksiäni, sekä tyttäreni *Claudian* ja *Kristinan*, joilta tähän työhön käyttämäni aika on valitettavan usein jäänyt paitsi.

Salossa 2012-05-23

Lasse Bergroth

Sisällysluettelo

1 Johdanto

Merkkijonoilla on keskeinen asema päivittäisessä käytännön tietojenkäsittelyssä. Etsiessämme tietoa verkkosivuilta on tarpeen käyttää joukkoa avainsanoja haun täsmentämistä varten. Vastaavasti, kirjoittaessamme pitkiä tekstidokumentteja, saatamme haluta etsiä yhden tai useampia kohtia, joissa esiintyy jokin tietty kirjoittamamme sana, jonka mahdollisesti haluaisimme korvata jollakin toisella. Kummassakin tilanteessa haettavista sanoista koostuva *malli* (engl. *pattern*) tulkitaan merkkijonoksi, jolle yritetään löytää täsmäävää vastinetta tekstidokumentista.

Suoritettava merkkijonohaku voi olla joko *tarkkaa*, jolloin kaikkien etsittävien merkkien pitää esiintyä tarkasteltavassa dokumentissa *peräkkäin ja vieläpä täsmälleen samassa järjestyksessä*. Tällöin etsittäessä vaikkapa mallia ”HAUKI” pitää sanan löytyä dokumentista juuri tässä muodossa[1]. Haku voi olla tarkan sijasta toisinaan *likimääräistä*, jolloin sallitaan pieniä eroavaisuuksia haettavan mallin ja sen vastineen välillä. Jos äskeisessä esimerkkitilanteessa sallittaisiin yhden merkin virhe, myös dokumentissa mahdollisesti esiintyvät merkkijonot ”HALKI” ja ”HAULI” olisivat riittävän lähellä etsittyä. Kummassakin tapauksessa tarvitaan haun toteuttamiseksi tehokkaasti toimivia *merkkijonoalgoritmeja* (engl. *string algorithms*). Näitä käsittelevä tutkimus – *merkkijonoalgoritmiikka* (engl. *stringology*/*string algorithmics*) – on yksi tietojenkäsittelytieteen tutkimusalueista.

Merkkijonoalgoritmit voidaan edelleen jakaa aliluokkiin käyttötarkoituksensa mukaisesti. Edellä kuvattua mallin tarkkaa etsintää dokumentista suorittavat *osajononhakualgoritmit*. Halutessamme puolestaan selvittää, mikä on minimaalinen määrä merkkien lisäyksiä, poistoja ja vaihtoja, jotka pitää tehdä ensimmäisenä parametrina annetun tiedoston muuntamiseksi jälkimmäisen kaltaiseksi (tai päinvastoin), käytetään merkkijonojen *lyhimmän editointietäisyyden* määräävää algoritmia. Vaihto-operaatio määritellään kustannukseltaan usein kaksinkertaiseksi lisäykseen ja poistoon verrattuna, sillä sen voi ajatella muodostuvan yhdestä poisto-operaatiota, jota seuraa yksi lisäysoperaatio (tai päinvastoin). Siten esimerkiksi merkkijonojen ”HAUKI” ja ”HAULI” välinen lyhin editointietäisyys olisi kahden mittainen: ensin mainitusta merkkijonosta pitäisi vaihtaa merkin ”K” paikalle ”L”, jotta siitä tulisi jälkimmäisen kaltainen. Tätä muistuttavalla periaatteella toimii mm. UNIX-käyttöjärjestelmän komento *diff*, joka listaa kuvaruudulle näkyviin rivit, joilla eroja tekstitiedostojen välillä esiintyy, kunhan niitä ei ole kohtuuttoman suurta määrää.

Kahden merkkijonon välisen samankaltaisuuden astetta voidaan mitata myös sillä, mikä on maksimaalisen pituinen kokoelma merkkejä, jotka esiintyvät samassa järjestyksessä kummassakin tarkasteltavassa merkkijonossa. Kerättäessä nämä merkit peräkkäin alkuperäisessä järjestyksessään muodostuu kahden merkkijonon *pisin yhteinen alijono* (lyh. PYA). Esimerkiksi merkkijonojen ”H**AU**K**I**” ja ”P**AU**L**I**” PYA

[1] Usein kuitenkin saman aakkossymbolin isot ja pienet kirjainmerkit samaistetaan yhdeksi merkiksi.

olisi ”**AUI**”, ja sen pituus on kolme. Pisimmän yhteisen alijonon ja lyhimmän editointietäisyyden ongelmat ovat läheistä sukua toisilleen, sillä yhtään syötejonojen PYAan kuuluvaa merkkiä ei tarvitse muuttaa toiseksi muunnettaessa jompaakumpaa jonoista toisen kaltaiseksi. Tässä työssä tarkastelun kohteina ovat juuri *kahden merkkijonon PYAn määräävät tarkat ja heuristiset algoritmit*, joille sovelluskohteita löytyy niin kuvien ja tekstien tiivistämisestä, tiedostojen versionhallinnasta, hahmontunnistuksesta kuin bioinformatiikastakin, jossa merkkijonoja edustavat DNA- tai proteiiniketjut, joiden samankaltaisuutta halutaan tutkia.

Koska merkkijonoja tarvitaan tietokoneohjelmissa usein, niiden esittämiseksi on monissa ohjelmointikielissä sekä tietokantajärjestelmissä varattu oma erillinen tyyppi *string* sekä sitä tukevat operaatiot. Tyypillisiä merkkijono-operaatioita ovat esimerkiksi kahden merkkijonon vertailu keskenään, merkkijonon pituuden laskeminen, merkkijonojen yhdistäminen peräkkäin eli *katenaatio*, merkkien järjestyksen kääntäminen päinvastaiseksi sekä tietyn osan valitseminen merkkijonosta.

Käsillä olevan työn tarkoituksena on valaista kahden merkkijonon pisimmän yhteisen alijonon ongelmaa useilta eri tahoilta. PYA-ongelma on siinä mielessä erityisen haasteellinen, että sen ratkaisemiseksi tarvittavien resurssien määrään vaikuttavat monet eri tekijät, kuten vertailun kohteena olevien merkkijonojen pituus, niissä esiintyvien erilaisten symbolien kokonaismäärä, PYAn pituuden prosentuaalinen osuus lyhemmän syötejonon pituudesta, aputietorakenteiden koko sekä merkkijakauma syötejonojen sisällä – pelkästään muutamia mainitaksemme. Lisäksi ongelman ratkaisemiselle on tyypillistä *kompromissien tekeminen* odotettavissa olevan suoritusajan sekä muistintarpeen välillä.

Koska monet eri parametrit vaikuttavat PYA-algoritmien suoritusaikaan ja muistinkäyttöön, on käytännössä mahdotonta kehittää PYA-ongelman ratkaisevaa algoritmia, joka toimisi tehokkaasti kaikille ajateltavissa oleville syötteille. Algoritmin valitseminen ilman ennakkotietoja tarkasteltavien syötejonojen ominaisuuksista johtaa ennemmin tai myöhemmin tehottomaan suoritukseen. Tämän työn tarkoituksena onkin tarkastella PYA-ongelmaa eri näkökulmilta ja sen myötä avustaa lukijaa sopivan algoritmin löytämisessä eri käyttökohteisiin.

Rakenteellisesti työ on jaettu kymmeneen eri *päälukuun* sekä viimeisen pääluvun perään sijoitettuun *liitteeseen* (luku 11). Ykkösluvun *johdantoa* seuraavassa toisessa luvussa esitellään tarkasteltavan ongelman kannalta tarpeelliset *määritelmät ja terminologia* sekä kuvataan algoritmeissa yleisimmin käytettävät *tietorakenteet*. Luvun päätteeksi luodaan katsaus aiemmissa tutkimuksissa todistettuihin PYA-ongelman vaativuuden *teoreettisiin rajoihin*, satunnaisten jonojen *PYAn pituutta koskeviin tuloksiin* sekä läheisiin *sukulaisongelmiin*. Kolmannessa luvussa pyritään antamaan *katsaus* ryhmittäin eri toimintaperiaatteilla laskentaa suorittaviin *PYAn ratkaiseviin menetelmiin*. Ryhmien sisällä pysyttäydytään kronologisessa järjestyksessä, jotta algoritmien kehittyminen ajan myötä olisi selkeästi hahmotettavissa. Työn neljännessä luvussa hypätään hetkeksi tarkastelemaan, miten kahden merkkijonon *lyhimmän*

editointietäisyyden laskevat algoritmit voidaan muuntaa PYA-ongelman ratkaiseviksi. Jo ennestään tunnettujen algoritmien kuvaus näissä kahdessa luvussa ennen työn varsinaisten tutkimustulosten esittelyä on tarpeen, sillä aikaisemmat algoritmit toimivat lähtökohtana uusien menetelmien kehitystyölle. Lisäksi niiden esittely antaa lukijalle selkeän ja yhtenäisen kokonaiskuvan eri menettelytavoista, miten PYA-ongelma voidaan ratkaista. Samalla uusien menetelmien kilpailuttaminen testiajoissa vanhojen kanssa on mielekkäämpää, kun kaikkien vertailtavien algoritmien tärkeimmät ominaisuudet ovat riittävän hyvin lukijan tiedossa. Kiteytetysti voidaan todeta, että aikaisemmin kehitetyt menetelmät muodostavat kattavan johdannon työssä esiteltäville uusille PYAn ratkaisumenetelmille. Mikäli lukijalla on jo ennestään vankat perustiedot kahden merkkijonon PYAn ja/tai lyhimmän editointietäisyyden määräävien algoritmien peruskäsitteistä ja niiden laskentatavoista, hän voi selailla työn lukuja 2 – 4 melko lailla valikoiden. Sen sijaan työn aiheeseen ennestään niukaltikin perehtynyt lukija pystyy kyseiset luvut läpi käymällä hankkimaan itselleen riittävät pohjatiedot työn kysymyksenasettelun ja tutkimustulosten ymmärtämiseksi.

Siinä missä luvuissa 1 – 4 pitäydytään jo aiemmin tutkitussa, työn loppuosa painottuu uusimpien tutkimustulosten havainnollistamiseen. Siinä esitellään tutkimusryhmämme ja tekijän tätä väitöskirjaa varten suorittamia tutkimuksia ja niiden tuloksia. Viidennessä luvussa esitellään erilaisia *heuristisia menetelmiä* PYA-ongelman ratkaisemiseksi. Näiden menetelmien tarkoituksena on tuottaa alkuperäiselle ongelmalle *likimääräisratkaisu* huomattavasti tarkan menetelmän vaatimia vähäisemmin resurssein. Kuudennessa luvussa tarkastellaan sekä *staattista että dynaamista heuristista esiprosessointia* tarkan algoritmin toiminnan tehostajana, minkä jälkeen seitsemännessä luvussa esitetään tekniikka, jota käyttämällä pyritään *parantamaan heuristisen esiprosessoinnin luotettavuutta*. Kahdeksannessa luvussa tarkastellaan erilaisten *hakutietorakenteiden* vaikutusta tarkkojen PYA-algoritmien suorittaman laskennan tehokkuuteen. Lisäksi siinä kiinnitetään huomiota, miten laskennan aikana havaittujen optimaalisten välitulosten muistiin kirjaaminen voisi vähentää algoritmien suorittamaa turhaa työtä. Yhdeksäs luku on varattu *empiiristen tutkimusten* tulosten esittelylle, ja kymmenennen luvun *yhteenveto* koostaa ja tiivistää työn tärkeimmät havainnot. Liitteen asemassa oleva yhdestoista luku sisältää lähes kaikkien työssä käsiteltävien PYA-algoritmien ja heuristiikkojen *pseudokoodit*.

Tätä työtä kirjoitettaessa turvaudutaan monin paikoin kuuteen joko yksinäni tai työryhmän jäsenenä julkaisemaani *konferenssiartikkeliin* [BHR98][BHR00][BHV03] [Ber05][Ber06][Ber07]. Empiirisessä tarkastelussa on pyritty pitäytymään pitkälti näiden artikkelien mukaisessa testausasetelmassa. Siten testiajoihin on otettu mukaan välillä pelkästään tarkkoja PYA-menetelmiä, toisinaan taas yksistään heuristiikkoja. Muutamissa testeissä on puolestaan kilpailutettu yhden menetelmän eri versioita, jotka eroavat toisistaan ainoastaan siinä käytetyn aputietorakenteen osalta. Lisäksi tarkastellaan tuloksia testiajoista, joissa heuristisella esiprosessoinnilla vahvistettujen

tarkkojen menetelmien suorituskykyä vertaillaan alkuperäismenetelmille mitattujen vastaavien tulosten kanssa.

Tutkielman lähestymistapa on yhtäältä *oppikirjamaisen esittelevä*, eli työssä lähdetään liikkeelle tarkasteltavan ongelman perusteista tuomalla esiin asian ymmärtämiseksi vaadittavat käsitteet ja teoreemat todistuksineen, minkä jälkeen selvitetään yksityiskohtaisin kuvauksin jo aikaisemmin kehitettyjen varsinaisten PYA-algoritmien sekä myös lyhimmän editointietäisyyden laskevien algoritmien toimintaperiaatteet. Algoritmien kuvaus on tehty *sanallisesti* asianomaista menetelmää käsittelevässä aliluvussa, ja niiden toimintaa on lisäksi valaistu monin itse kehitetyin ja paikoitellen myös kirjallisuudesta lainatuin *esimerkein*.

Lähes kaikki menetelmät on esitelty erikseen vielä *pseudokoodimuodossa* siltä varalta, että lukija haluaa perehtyä menetelmiin niiden pienimpiä yksityiskohtia myöten. Varhaisinta ja samalla yksinkertaisinta PYAn ratkaisumenetelmää eli *Wagnerin ja Fischerin* algoritmia lukuun ottamatta pseudokoodit löytyvät työn *liitesivuilta*. Lisäksi *Goemanin ja Clausenin* algoritmista on otettu mukaan erillinen lyhennetty pseudokoodi varsinaiseen tekstiosuuteen. Näin on toimittu, sillä se selkeyttää lukijalle toiminnaltaan muutoin varsin monimutkaista menetelmää. Pseudokoodeissa on *kontrollirakenteiden* eli silmukoiden, ehto- ja valintalauseiden vaikutusalueet esitetty *porrastuksella*. Kuitenkin, jos ne ovat päällekkäisiä tai niiden vaikutusalue on kovin pitkä hahmotettavaksi, niihin on lisätty myös *lopettava käskysulku*. Jokainen algoritmeista on kuitenkin pyritty esittelemään siten, ettei pseudokoodiin perehtyminen ole välttämätöntä menetelmän tärkeimpien toiminnallisten piirteiden ymmärtämiseksi. Pseudokoodien järjestys liitesivuilla on yhdenmukainen menetelmien esittelyjärjestyksen kanssa työn varsinaisissa lukukappaleissa.

Esimerkin 3.1 merkkijonoja on annettu syötteiksi useille eri algoritmeille, mikä edesauttaa algoritmien toiminnan vertailua keskenään. Muutamien menetelmien pseudokoodeissa esiintyneet, tutkimusryhmämme havaitsemat virheet on tässä työssä korjattu. Joidenkin algoritmien pseudokoodeja on lisäksi täydennetty testiajoihin implementointia varten siten, että ne pystyvät palauttamaan PYAn pituuden lisäksi myös jonkin tarkasteltavan ongelman ratkaisuksi kelpaavista jonoista. Muutamat algoritmeista ovat kokonaisuutena verraten monimutkaisia. Työssä esitettyjä pseudokoodeja lukija voi vapaasti käyttää ei-kaupallisiin tarkoituksiin. Algoritmien esittelyn yhteydessä on lisäksi analysoitu niiden *suoritusaika ja tilantarve*. Mikäli analyysiä ei ole esitetty algoritmin julkaisuartikkelissa, tai siinä esitetyn analyysin sisältämät termit eivät ole tämän työn kannalta tarkoituksenmukaisia, algoritmi on analysoitu erikseen tätä työtä varten.

Vasta tunnettujen perusalgoritmien esittelyn jälkeen siirrytään laajentamaan tarkastelua uudemmilla tutkimuksilla saavutettuihin tuloksiin kuten PYAn heuristisiin likimääräisalgoritmeihin sekä heuristiikoilla, uudistetuilla tietorakenteilla tai ohjelmointiteknisesti parannettuihin tarkkoihin PYA-algoritmeihin. Lisäksi työssä puidaan keskenään vaihtoehtoisten hakutietorakenteiden etuja ja nurjia puolia. Työn

toisaalta *vertaileva lähestymistapa* tulee parhaiten ilmi luvussa 9, jossa esitellään sekä aikaisemmin kehitetyille että uusille menetelmille tehtyjen testiajojen tuloksia ja analysoidaan eri menetelmien, laskentamallien ja käytettävien hakutietorakenteiden välisiä paremmuuseroja erityyppisille syötteille. Tutkimusryhmämme on vastannut kaikkien vertailtujen algoritmi- ja heuristiikkaversioiden ohjelmoinnista työn empiiristä osuutta varten.

Työn tavoitteena on antaa laaja ja analyyttinen katsaus PYAn muodostaviin algoritmeihin ja heuristisiin menetelmiin, niiden kehitykseen, toimintaperiaatteisiin, menetelmissä hyödynnettäviin tietorakenteisiin sekä – ennen kaikkea – niiden suorituskykyyn ja menetelmien välisiin paremmuuseroihin erityyppisille syötteille. Jottei työ kuitenkaan pursuaisi kohtuuttoman mittavaksi, on selvää, ettei kaikkia tunnettuja PYAn ratkaisumenetelmiä ole voitu mahduttaa työhön mukaan. Esimerkkeinä enempien tarkastelujen ulkopuolelle jätetyistä algoritmeista mainittakoon *Apostolicon*, *Brownen* ja *Guerran* vuonna 1992 [Apo92] sekä *Rickin* vuonna 2000 [Ric00_2] esittelemät lineaaritilaiset menetelmät. Näistä ensin mainitussa algoritmin pseudokoodi on vaillinainen ja jälkimmäisestä[2] se puuttuu kokonaan. Myös erittäin raskasta esiprosessointia vaativa mutta asymptoottisesti tehokas, varsinaisesti syötejonojen lyhimmän editointietäisyyden laskentaan suunniteltu *Masekin ja Patersonin algoritmi* [Mas80], on tässä työssä sivuutettu. Syynä tähän on algoritmin kelvottomuus PYA-ongelman ratkaisemiseen, elleivät syötejonot ole hyvin pitkiä [All86], minkä johdosta menetelmä olisi joka tapauksessa pitänyt jättää testiajopaketin ulkopuolelle[3]. Lisäksi syötejonojen *kaikki PYAn esiintymät etsivä Rickin algoritmi* [Ric00_1] sekä uusimmat *bittivektoreiden käsittelyyn* perustuvat algoritmit kuten *Crochemoren*, *Iliopoulosin*, *Pinzonin* ja *Reidin* algoritmi [Cro01] sekä *Iliopoulosin* ja *Pinzonin* algoritmi [Ili02] on jätetty työssä käsittelemättä. Bittivektorien käsittelyyn perustuvia algoritmeja on tarkoitus tarkastella myöhemmissä vertailevissa julkaisuissani.

Johdannon päätteeksi esitellään seuraavassa vielä työn tärkeimmät saavutukset pähkinänkuoressa. Ensiksi on haluttu muodostaa PYA-ongelman sekä tarkasti että heuristisesti ratkaisevista menetelmistä yhtenäinen *karttamainen kokonaiskuva*, jotta lukija voisi hahmottaa, mitä vaihtoehtoisia tapoja ongelman ratkaisemiseksi on olemassa. Mitään vastaavan laajuista PYA-algoritmien kokoomateosta, joka pitäisi sisällään sekä ongelman peruskäsitteet, menetelmien yksityiskohtaiset toiminnan kuvaukset, niihin liittyvät keskeisimmät teoreemat todistuksineen, pseudokoodin että analyysin — sekä tarkoille että heuristisille menetelmille — ei tietojeni mukaan ole julkaistu. Toiseksi, lähestulkoon kaikki tarkasteltavat algoritmit on työtä varten *ohjelmoitu* noudattamalla yhtenäistä ohjelmointityyliä ja muistinvaraustekniikkaa, jotta

[2] Rickin lineaaritilaiseen algoritmiin palataan tosin lyhyesti saman tekijän ei-lineaaritilaisen, mutta laskentatavaltaan samankaltaisen algoritmin esittelyn yhteydessä aliluvussa 3.4.1.3.

[3] *Allisonin* ja *Dixin* mukaan Masekin ja Patersonin algoritmi on kilpailukykyinen vasta silloin, kun syötejonojen pituus on vähintään 200 000 merkkiä. Tässä työssä syötejonojen enimmäispituus on 10 000 merkkiä (käytetyn testausjärjestelmän asettama yläraja).

eri menetelmille mitatut ajoajat ja muistinkulutus olisivat keskenään vertailukelpoisia. Myös alkuperäisartikkelien pseudokoodeihin jääneitä painovirheitä on pyritty tässä oikaisemaan. Kolmanneksi, työssä otetaan kantaa kysymykseen, voidaanko huolellisella *tietorakenteiden valinnalla* sekä *heuristisella esiprosessoinnilla* saada lisää tehoa alkuperäisiin PYA-menetelmiin. Neljäntenä, ehkäpä tärkeimpänä sanomana lukijalle, työssä esitellään laaja *empiirinen tulospaketti*, jossa ongelman kannalta oleellisten parametrien kuten syöttöaakkoston koon, merkkijakauman ja PYA-osuuden annetaan vaihdella. Työn tulosluku on siten omiaan helpottamaan lukijaa valitsemaan tehokkaan algoritmin tarkastelemansa ongelman instanssiin, kunhan syötejonojen ominaisuuksia voidaan ainakin jossain määrin ennakoida. Silloin, kun mitään ennakkotietoja ei ole suoraan käytettävissä, voidaan tietyt algoritmien toiminnan kannalta kriittiset parametrit – kuten syöttöaakkoston koko ja merkkijakauma – selvittää nopeasti ennen PYAn ratkaisevan algoritmin kiinnittämistä.

2 Pisimmän yhteisen alijonon ongelman perusteet

Pisimmän yhteisen alijonon ongelmassa syötejonoja pitää olla vähintään kaksi, mutta niiden lukumäärää ei periaatteessa tarvitsisi rajoittaa ylhäältä. Monissa käytännön sovellutuksissa riittää tarkastella kuitenkin vain kahta syötejonoa, ja kaiken lisäksi kolmen tai useamman jonon samanaikainen tarkasteleminen tekee ongelmasta oleellisesti vaikeamman. Kolmelle jonolle tarkka ratkaisu löydetään naiivilla, *dynaamiseen ohjelmointiin* perustuvalla algoritmilla hitaasti ajassa $O(n^3)$, missä n on pisimmän syötejonon pituus [Alg10]. Tästä syystä kolmen tai useamman jonon eksaktin PYAn ratkaisevia menetelmiä ei juurikaan ole kehitetty[4], vaan usealle jonolle sovelletaan tavallisimmin – erityisesti bioinformatiikassa – ns. *heuristisia rinnakkainasettamismenetelmiä* (engl. *multiple sequence alignment methods*) [PBL08]. Siten käsillä olevassa työssä rajoitutaan tarkastelemaan ainoastaan *kahta syötemerkkijonoa*.

Tässä luvussa annetaan aluksi PYA-ongelmaan liittyvät *peruskäsitteet ja määritelmät*, minkä jälkeen esitellään dynaamiseen ohjelmointiin perustuva, Wagnerin ja Fischerin kehittämä PYAn *naiivi ratkaisualgoritmi*. Tämä yksinkertainen algoritmi on muodostanut perustan ja lähtökohdan usean tehokkaamman PYA-algoritmin kehitystyölle. Naiivin ratkaisualgoritmin esittelyn jälkeen keskitytään ensiksi pidemmälle jalostettuihin PYA-menetelmiin liittyvien *lisähuomioiden* ja *terminologian* esittelyyn sekä PYA-ongelmalle tyypillisten piirteiden syvempään tarkasteluun. Myöhemmin luodaan katsaus algoritmeissa yleisimmin apuna käytettyihin *hakutietorakenteisiin*. Luvun loppuosassa luodaan lyhyt katsaus PYA-ongelman ratkeamisen vaativuudelle laskettuihin *teoreettisiin rajoihin* sekä PYAn *keskimääräisen pituuden* arvioimiseen satunnaisille syötemerkkijonoille. Lisäksi esitellään lyhyesti muutamia PYA-ongelmalle *läheistä sukua olevia ongelmia*. Aivan luvun päätteeksi esitetään vielä luettelo työssä vakiintuneista *lyhenteistä* ja *merkinnöistä* selityksineen.

2.1 Peruskäsitteitä ja määritelmiä

2.1.1 Alijonon käsite ja pisin yhteinen alijono

PYA-ongelmassa tarkasteltavat *syötejonot* on tallennettu vektoreihin $X[1..m]$ ja $Y[1..n]$, missä m, $n \geq 0$ ja kuvaavat niiden pituutta. Syötejonojen mihin tahansa merkkiin päästään siten viittaamaan suoraan sijaintipaikkansa perusteella. Vektoreiden indeksointi alkaa aina ykkösestä, ja jos jälkimmäinen indeksi on numeroltaan ensimmäistä pienempi, kyseessä on tyhjä vektori. Rajoittamatta mitenkään

[4] Kaksi usean merkkijonon PYAn ratkaisevaa menetelmää on julkaistu teoksessa [Fra92].

tarkasteltavien algoritmien yleisyyttä[5] voidaan olettaa, että jono Y on aina vähintään yhtä pitkä kuin X, eli $m \leq n$. Mielivaltaisen merkkijonon Z pituus voidaan lausua käyttämällä merkintää *pituus*(Z). Erityisesti on voimassa *pituus*(X) = m ja *pituus*(Y) = n.

Jono $Z[1..t] = z_1z_2z_3\ldots z_t$ ($0 \leq t \leq m$), on jonon $X[1..m]$ *alijono*, mikäli se saadaan poistamalla jonosta X mielivaltaisen monta ($0..m$) merkkiä vapaasti valittavista indeksipaikoista[6]. Esimerkiksi Z = ”**ALI**” olisi yksi mahdollinen jonon X = ”P**A**U**LI**” alijono, joka on saatu poistamalla jonon X ensimmäinen ja kolmas merkki.

Mikäli jono Z täyttää edellä esitetyn alijonon määritelmän samalla sekä merkkijonolle X että Y, on Z tällöin kyseisten kahden merkkijonon *yhteinen alijono*. Esimerkiksi pituudeltaan kahden mittainen merkkijono Z_1 = ”**UI**” olisi jonojen X = ”HA**U**K**I**” ja Y = ”PA**U**L**I**” yhteinen alijono, sillä merkit ”U” ja ”I” löytyvät tässä järjestyksessä sekä X:stä että Y:stä. Yhteisen alijonon määritelmästä seuraa, että sen pituus on enintään lyhemmän syötemerkkijonon mittainen. Jos syötejonoilla X ja Y ei ole yhtään yhteistä symbolia, niiden ainoa yhteinen alijono on tyhjä jono.

Syötejonojen X ja Y *pisin yhteinen alijono* voitaisiin määritellä formaalisti seuraavalla tavalla. Olkoon Γ syötejonojen X ja Y kaikkien yhteisten alijonojen joukko. Tällöin alijono $Z_k \in \Gamma$, jolle on voimassa $(\forall i \mid Z_i \in \Gamma$: *pituus*($Z_k$) $\geq$ *pituus*(Z_i)), on jonojen X ja Y PYA. Edellisessä esimerkissä Z_1 ei selvästikään ole syötejonojensa PYA, sillä myös sitä pidempi, kolmen merkin mittainen jono Z_2 = ”**AUI**”, olisi esimerkin syötejonojen X ja Y yhteinen alijono. Jono Z_2 puolestaan olisi nyt samalla myös esimerkkisyötteiden PYA, sillä syötejonon X symboleita ”H” ja ”K” ei esiinny lainkaan toisessa syötejonossa Y. PYAn pituudesta käytetään vastedes lyhennysmerkintää p. Äskeisessä esimerkissämme siis PYA = ”**AUI**” ja p = 3. Haluttaessa täsmentää, minkä kahden syötejonon pisimmästä yhteisestä alijonosta on kyse, voidaan merkintää PYA selventää argumentein muotoon *PYA*(X, Y).

PYA ei välttämättä ole aina yksikäsitteinen. Esimerkiksi syötemerkkijonojen ollessa X = ”H**E**L**S**INKI” ja Y = ”AM**S**T**E**RDAM” on PYA vain yhden mittainen, sillä X:n alijonolla ”HLINKI” ei ole yhtään yhteistä merkkiä jonon Y kanssa. Merkeille ”E” ja ”S” puolestaan löytyy vastine Y:stä, mutta koska ne esiintyvät Y:ssä vain päinvastaisessa järjestyksessä, ainoastaan toinen niistä – joko ”E” tai ”S” – voi kerrallaan kuulua jonojen PYAan. Siten esimerkissämme p = 1, ja PYAksi kelpaavat sekä ”**E**” että ”**S**”. Lisäksi on mahdollista, että samansisältöinen PYA voidaan toisinaan kerätä useilla eri tavoilla. Esimerkiksi merkkijonojen X = ”H**EL**S**IN**K**I**” ja Y = ”B**E**R**L**I**I**N**I**” PYA ”**ELINI**” voidaan valita jonosta Y kahdella eri tavalla: PYAn kolmas merkki ”I” voidaan ottaa jonosta Y joko indeksipaikasta 5 tai 6.

[5] Voidaan aina sijoittaa lyhyempi syötemerkkijonoista vektoriin X ja pidempi vektoriin Y. Näin on mahdollista menetellä, sillä PYA-ongelma on *kommutatiivinen*: PYA(X, Y) = PYA(Y, X).
[6] Jos Z saadaan poistamalla X:stä kaikki merkit, Z on X:n *tyhjä* ja samalla *triviaali alijono*.

2.1.2 Syöttöaakkosto

Useissa PYA-ongelman ratkaisevissa algoritmeissa *syöttöaakkoston koolla* eli syötemerkkijonoissa esiintyvien erilaisten merkkien lukumäärällä on voimakas vaikutus käytettävän algoritmin tehokkuuteen. Syöttöaakkostoon kuuluvien symbolien joukosta käytetään lyhennysmerkintää Σ ja siihen kuuluvien merkkien lukumäärästä lyhennettä σ. Työssä oletetaan, että syöttöaakkoston yksittäisiä symboleja merkitään tunnuksilla s_i, missä i on indeksoitu välille $[1..\sigma]$. Tämä onnistuu helposti lineaarikuvauksella miltä tahansa alkuperäiseltä symbolijoukolta[7], jonka koko on σ. Edelleen oletetaan, että tarkasteltavan ongelman syöttöaakkoston koko selvitetään ennen algoritmin suorittamista, mikä voidaan tehdä ajassa $O(n \log \sigma)$ selaamalla tarvittaessa kumpikin syötevektoreista kertaalleen läpi ja kirjaamalla erillisten symbolien lukumäärä etsimällä kutakin symbolia tarkasteluhetkeen asti löydettyjen merkkien järjestetystä vektorista. Mikäli kuitenkin tiedetään syöttöaakkoston koon olevan suuruusluokkaa $O(n)$, saadaan aakkoston koko myös selvitettyä samaisessa ajassa hyödyntämällä lineaarisessa tilassa toimivaa lajittelumenetelmää.

2.1.3 Osajono sekä alku- ja loppuliitteet

Mielivaltaisen ei-tyhjän merkkijonon $Z[1..u]$ yhtenäisen indeksialueen $i..j$ rajoittamaa jonoa $Z'[i..j]$, missä $i \leq j$ ja $i, j \in [1, u]$, kutsutaan Z:n *osajonoksi*. Erityisesti, jos $Z' = Z[1..k]$ $(1 \leq k \leq u)$ – eli Z' sisältää Z:n k ensimmäistä merkkiä, kutsutaan Z':a merkkijonon Z k:n mittaiseksi *alkuliitteeksi* eli *prefiksiksi*. Vastaavasti Z:n loppupään merkeistä koostuva osajono $Z''[t..u]$ $(1 \leq t \leq u)$ on Z:n *loppuliite* eli *suffiksi*, jonka pituus on $u–t+1$. Mikäli puolestaan edellä määritellyissä jonon Z alku- ja loppuliitteissä $k = 0$ tai $t > u$, on kyseessä Z:n tyhjä alku- vs. loppuliite.

Tähtisymboli (*) merkkijonon tunnuksen jälkeen ilmaisee, että merkkijonoa tarkastellaan käännetyssä järjestyksessä. Esimerkiksi merkintä Z^* tarkoittaisi merkkijonoa Z käännettynä päinvastaiseen järjestykseen. Vastaavasti $Z^*[i..j]$ $(i \geq j)$ tarkoittaisi Z:n käännettyä osajonoa $z_i z_{i-1} \ldots z_j$.

2.1.4 Täsmäys ja sen luokka

Mikäli merkkijonon X indeksipaikassa i esiintyy sama symboli kuin merkkijonon Y paikassa j, käytetään indeksiparista (i, j) nimitystä *täsmäys* (engl. *match*). Kohdassa (i, j) sijaitsevan täsmäyksen *luokka* k $(1 \leq k \leq p)$ kuvaa syötejonojen alkuliitteiden $X[1..i]$ ja

[7] Työn esimerkeissä merkitään syöttöaakkoston symboleita kuitenkin suomen kielen aakkosmerkeillä, jotta esimerkit olisivat lukijalle selkeämpiä.

$Y[1..j]$ välisen PYAn pituutta. Kaikkien luokkaan k kuuluvien täsmäysten joukkoa esittää merkintä C_k. Lisäksi määritellään pelkästä pseudotäsmäyksestä (0, 0) koostuva luokka C_0, jota tarvitaan avuksi muutamissa PYA-algoritmeissa alustustoimenpiteitä varten. Luokkaan C_k voi kuulua korkeintaan $m+n-2k+1$ täsmäystä. Kyseinen enimmäismäärä samanaikaisesti kaikille täsmäysluokille C_k, samoin kuin kaikkien luokkien täsmäysten teoreettinen yhteenlaskettu maksimimäärä mn, saavutetaan tarkalleen silloin, kun syöttöaakkosto — ja samalla tietysti myös syötemerkkijonot — koostuvat ainoastaan yhdestä symbolista. Viimeksi mainitut täsmäysten ominaisuudet ovat helposti todennettavissa. Jos kummassakin syötejonossa esiintyy vain yhtä ja samaa symbolia, niin tällöin jokainen X:n merkki täsmää jokaisen Y:n merkin kanssa, mistä saadaan täsmäysten kokonaismäärä mn. Lisäksi voidaan havaita, että tuolloin erityisesti $X[k]$ muodostaa määritelmän mukaisesti luokan k täsmäyksen merkkien $Y[k..n]$ ja vastaavasti $Y[k]$ merkkien $X[k..m]$ kanssa. Näiltä alueilta löytyy täsmäyksen (k, k) lisäksi muita luokan k täsmäyksiä yhteensä $m-k+n-k$ kappaletta, joten C_k:hon kuuluvien täsmäysten enimmäismäärä on siten $m+n-2k+1$ kappaletta. Täsmäysten kokonaismäärää tarkasteltavassa ongelmassa edustaa algoritmien analyysissä termi r.

Koska PYAan kuuluvien merkkien pitää löytyä samassa järjestyksessä molemmista syötejonoista, PYAn ja sen pituuden määräämiseksi voitaisiinkin siten rajoittua tarkastelemaan pelkkiä täsmäyksiä. Useimmat tehokkaimmista PYA-algoritmeista rajoittuvatkin laskennassa pelkästään täsmäyskohtien tarkasteluun. Seuraavassa aliluvussa esiteltävä ensimmäinen, *dynaamiseen ohjelmointiin* perustuva PYA-algoritmi, laskee PYAn pituuden kuitenkin syötejonojen kaikille mahdollisille alkuliitepareille $X[1..i]$, $Y[1..j]$, $0 \leq i \leq m$, $0 \leq j \leq n$.

2.2 Dynaamiseen ohjelmointiin perustuva PYAn naiivi ratkaisualgoritmi (WFI)

Ensimmäinen PYA-ongelman ratkaiseva algoritmi esiteltiin vuonna 1974, ja sen kehittivät *Wagner* ja *Fischer* (*WFI*) [Wag74]. Menetelmän suoritusaika ja muistintarve ovat aina suuruusluokkaa $O(mn)$, ja ne riippuvat siten yksinomaan syötevektoreiden pituuksista: jonojen muilla ominaisuuksilla ei ole merkitystä suoritusajan kannalta.

Algoritmi varaa käyttöönsä 2-ulotteisen matriisin M, jossa on $m+1$ kappaletta rivejä ja $n+1$ saraketta. Sekä rivien että sarakkeiden indeksointi alkaa nollasta. Matriisin yläreunan *nollas rivi* edustaa vektorin X ja vasemman reunan *nollas sarake* vektorin Y *tyhjää alkuliitettä*. Algoritmi alustaa nollannen rivin ja sarakkeen solut nolliksi, sillä kahden merkkijonon PYA on määritelmän mukaisesti väistämättä aina nollan mittainen, jos jompikumpi syötteistä on tyhjä.

Alustuksen päätyttyä algoritmi käy läpi syötevektorin X merkit järjestyksessä alusta loppuun ja vertaa kulloinkin tarkastelun kohteena olevaa X:n merkkiä vuoron perään

jokaisen *Y*:n merkin kanssa alusta loppuun. Matriisissa edetään siten rivi kerrallaan vasemmalta oikealle ja ylhäältä alas. Siirryttäessä matriisissa *M* riviä *i* ($0 \leq i \leq m$) pitkin oikealle sarakkeelta *j* sarakkeelle *j*+1 ($0 \leq j \leq n$–1) pysyy *X*:n alkuliitteen pituus ennallaan, mutta *Y*:n alkuliite pitenee yhdellä merkillä. Vastaavasti siirtyminen alaspäin riviltä *i* ($0 \leq i \leq m$–1) riville *i*+1 saraketta *j* pitkin ($0 \leq j \leq n$) kasvattaa *X*:n alkuliitteen pituutta yhdellä, kun taas *Y*:n alkuliitteen pituus pysyy muuttumattomana. Matriisiin jokaiseen soluun $M[i, j]$ ($1 \leq i \leq m$, $1 \leq j \leq n$) algoritmin suorituksen aikana sijoitettava arvo määräytyy seuraavien laskentasääntöjen perusteella:

$$1° \quad (x_i = y_j) \Rightarrow M[i, j] = M[i–1, j–1] + 1$$
$$2° \quad (x_i \neq y_j) \Rightarrow M[i, j] = Max\{M[i, j–1], M[i–1, j]\}$$

Laskentasäännöistä on helposti havaittavissa, että matriisin soluun $M[i, j]$ tallentuva arvo määräytyy joko sen vasemmassa ylänurkassa sijaitsevaan soluun (1. sääntö) tai sitten sitä lähinnä vasemmalla ja suoraan yläpuolella sijaitseviin soluihin (2. sääntö) tallennettujen arvojen perusteella. Alustustoimenpiteenä tehty matriisin nollannen rivin ja sarakkeen täyttäminen takaa sen, ettei laskentasäännöissä milloinkaan viitata määrittelemättömiin arvoihin. Arvo paikassa $M[i, j]$ kuvaa syötejonon *X* alkuliitteen $X[1..i]$ ja syötejonon *Y*$[1..j]$ PYAn pituutta. Lisäksi huomataan, että soluun $M[i, j]$ voi tallentua edellä mainittuihin kolmeen naapurisoluun tallennettuja arvoja suurempi arvo ainoastaan sellaisissa paikoissa, joissa $X[i] = Y[j]$, eli *X*:n *i*. ja *Y*:n *j*. merkki ovat *keskenään samat*, eli *täsmäyskohdissa*.

Kun laskenta on edennyt soluun $M[m, n]$ asti, on koko syötejonojen PYAn pituus *p* selvillä, ja se tallentuu kyseiseen soluun. Haluttaessa löytää myös jokin PYA-jonoista lähdetään perääntymään solusta $M[m, n]$ ensiksi niin pitkään vasemmalle, kunnes löydetään rivin *m vasemmanpuoleisin solu*, johon on tallennettuna arvo *p*. Olkoon tämän solun sarakeindeksi *j*. Tämän jälkeen jatketaan puolestaan kyseisestä solusta saraketta *j* pitkin ylöspäin niin kauan, kunnes löydetään *ylin rivi*, jossa esiintyy arvo *p*. Olkoon tämän rivin indeksi *i*. WFI-algoritmin laskentasääntöjen perusteella solun $M[i, j]$ arvo on nyt saatu soveltamalla sääntöä 1, koska sekä solun vasemmalta että yläpuolelta löytyy vain *p*:tä pienempiä arvoja. Täsmäyksen $M[i, j]$ muodostava merkki x_i (samalla y_j) on nyt siten PYAn viimeinen symboli. Seuraavaksi pienennetään *p*:n arvoa yhdellä, siirrytään matriisissa soluun $M[i–1, j–1]$ ja jatketaan samaan tapaan edellisten PYAan kuuluvien merkkien etsintää, kunnes lopulta *p* pienenee nollaksi. Tällöin yksi ratkaisu PYAlle on löydetty. Seuraavassa on esitettynä WFI-algoritmin pseudokoodi.

```
ALGORITMI WFI (X, m, Y, n):
FOR i := 0, 1, … → m
    M[i, 0] := 0;  /* Alustetaan matriisin M vasen pystyrivi. */
FOR j := 1, 2, … → n
    M[0, j] := 0;  /* Alustetaan matriisin M ylin vaakarivi. */
```

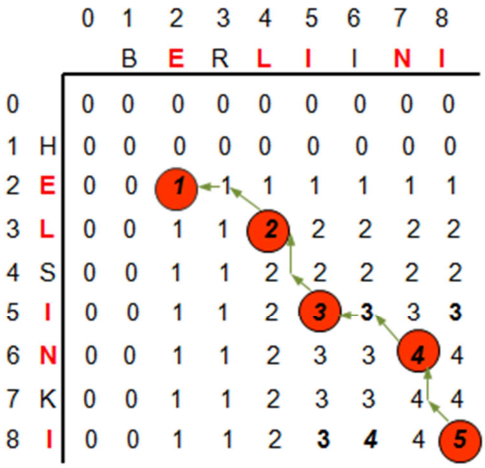
```
S1: FOR i := 1, 2, … → m /* Tarkastellaan rivi kerrallaan ylhäältä alaspäin …*/
   S2: FOR j := 1, 2, … → n /* … ja sarake kerrallaan vasemmalta oikealle. */
        IF (x_i = y_j)  /* Ovatko syötevektorien merkit x_i ja y_j samat? */
            M[i, j] := M[i–1, j–1] + 1  /* Kyllä: alkuliitteiden PYAn pituus kasvaa yhdellä. */
        ELSE
            M[i, j] := Max{M[i, j–1], M[i–1, j]}  /* Eivät: alkuliitteiden PYA ei pidentynyt. */
   ENDFOR (S2)
ENDFOR (S1)
pituus := M[m, n];  /* Syötteiden PYAn pituus tallentuu soluun M[m, n]. */
Tulosta(pituus): /* Tulostetaan PYAn pituus. */
i := m;  /* Alustetaan rivinumero m:ksi ... */
j := n;  /* … ja sarakenumero n:ksi. */
PYAn tulostaminen(M, i, j, pituus);
END (ALGORITMI WFI).

PROSEDUURI PYAn tulostaminen (M, i, j, pituus):
WHILE (pituus ≠ 0)  /* Kerätään PYA-jono lopusta alkuun päin. */
    IF (M[i, j–1] = pituus)  /* Onko solussa M[i, j] sama arvo kuin solussa M[i, j–1]? */
        j := j – 1  /* On: siirrytään vasemmanpuoleiseen soluun. */
    ELSE  /* Ei: M[i, j] > M[i, j-1]. */
        IF (M[i–1, j] = pituus)  /* Onko solussa M[i, j] sama arvo kuin solussa M[i–1, j]? */
            i := i – 1  /* On: siirrytään yläpuolella olevaan soluun. */
        ELSE  /* Ei: M[i, j] > Max{M[i, j–1], M[i–1, j]} */
            PYA[pituus] := x_i;  /* Täytetään PYA-jonoa lopusta alkuun päin. */
            pituus := pituus – 1;  /* PYAn merkeistä on nyt yksi vähemmän löytymättä. */
            i := i – 1;  /* Siirretään X-kohdistinta kohti alkua. */
            j := j – 1;  /* Tehdään samoin Y-kursorille. */
Tulosta PYA-jonoon kuuluvat merkit vektorista PYA.
END (PROSEDUURI PYAn tulostaminen)
```

Seuraavassa esitetään vielä esimerkki 2.1, joka havainnollistaa, miten WFI-algoritmi ratkaisee PYA-ongelman syötejonoille X = ”HELSINKI” ja Y = ”BERLIINI”. Jos ongelman PYA voidaan, kuten esimerkissämme, muodostaa useammalla kuin yhdellä tavalla, edellä kuvatun etsintäperiaatteensa ansiosta algoritmi löytää PYAlle aina sen esiintymän, jota edustava polku kulkee matriisissa lähellä vasenta alakulmaa[8].

[8] Tarkemmin ilmaistuna algoritmin löytämä ratkaisu muodostuu jokaisen luokan *k alimmista* ns. *dominanteista täsmäyksistä*, joista PYA on vielä muodostettavissa. Nämä esitellään aliluvussa 2.3.1.

Esimerkki 2.1: X = ”HELSINKI”, Y = ”BERLIINI”, $m = n = 8$, $p = 5$, PYA = ”ELINI”.



Oheisessa kuvassa ovat nähtävissä matriisiin M tallentuvat arvot ratkaistaessa PYA dynaamisella ohjelmoinnilla syötejonoille X = ”HELSINKI” ja Y = ”BERLIINI”. Matriisiin merkityt punaiset pallukat kuvaavat indeksipareja (i, j), joissa alkuliitteiden X[1..i] ja Y[1..j] PYA saavuttaa ensimmäisen kerran pituuden k ($1 \leq k \leq 5$). PYAn pituus on selvillä tallennettaessa arvo 5 soluun (m, n) = (8, 8). **Oliivinvihreät** nuolet kuvaavat PYA-jonoon ”**ELINI**” tallennettavien merkkien löytymisjärjestystä. PYAan kuuluvat merkit on maalattu syötejonoissa **punaisiksi**.

2.3 Lisää määritelmiä

2.3.1 Dominantti täsmäys

Tutkimalla täsmäysten ominaisuuksia syvällisemmin havaitaan, että välttämättä kaikki täsmäykset eivät ole tarpeellisia ratkaistaessa PYA-ongelmaa. Tarkastellaan uudelleen esimerkkiä 2.1. Matriisin M riviltä 5 löytyy useita luokkaan 3 kuuluvia täsmäyksiä: paitsi sarakkeesta 5 niin myös sarakkeista 6 ja 8, koska kaikista näistä Y:n indeksipaikoista löytyy X:n viides merkki ”I”. Täsmäyskohdat on merkitty kuvaan **lihavoiduin** numeroin. Koska jo alkuliitteiden X[1..5] = ”HELSI” ja Y[1..5] = ”BERLI” välisen PYAn pituus on 3, olisi turhaa sitoa Y:stä kuuden tai kahdeksan pituista alkuliitettä kolmen mittaisen PYAn muodostamiseksi alkuliitteen X[1..5] kanssa. Valitsemalla täsmäyksen (5, 6) täsmäyksen (5, 5) asemesta ei esimerkissämme tosin vielä aiheutuisi mitään vahinkoa, mutta valitsemalla täsmäyksen (5, 8) menettäisimme mahdollisuuden muodostaa kolmea merkkiä pidempää PYAa, koska valinta (5, 8) sitoisi tarpeettomasti jo kokonaisuudessaan jonon Y = ”BERLIINI” jonon X alkuliitteen ”HELSI” kanssa. Sen sijaan valinnan (5, 5) jälkeen voi PYA edelleen pidentyä loppuliitteiden X[6..8] ja Y[6..8] täsmäyksillä, koska tällöin syötejonosta Y kiinnitetään ainoastaan alkuliite ”BERLI”. Esimerkissä täsmäys (5, 5) on siten Y:n alkuliitteen pituuden rivillä 5 minimoiva eli *dominantti* luokan 3 täsmäys. Muita synonyymejä dominanteille täsmäyksille ovat *minimaalinen* ja *peittävä täsmäys* sekä *minimaalinen ehdokas* (engl. *dominant/minimal/covering match*, *minimal candidate*). Dominantit täsmäykset erottuvat esimerkin muista täsmäyksistä *kursivoinnilla*.

Voidaan määritellä, että luokkaan k kuuluva täsmäys (i, j) on *k-dominantti* eli kuuluu luokan k *dominanttien täsmäysten joukkoon* D_k ($D_k \subseteq C_k$) tarkalleen silloin, kun ei ole olemassa toista luokan k täsmäystä (u, v), joka toteuttaisi ehdon $((u = i) \wedge (v < j)) \vee ((u < i) \wedge (v = j))$. Tästä seuraa, että kullakin rivillä ja sarakkeella voi sijaita

korkeintaan yksi samaan luokkaan kuuluva dominantti täsmäys. Lisäksi on helppo päätellä, että luokan k dominanttien täsmäysten sarakenumerot pienenevät sitä mukaa kun niiden rivinumerot kasvavat. Tämä havainto perustuu sekä yleisen että dominantin täsmäyksen määritelmään. Mikäli on olemassa (dominantti) k-täsmäys paikassa (i, j) sekä toinen (dominantti) täsmäys paikassa (u, v), missä $i < u$ ja $j < v$, ei (u, v) voi selvästikään kuulua enää samaan luokkaan k, koska nyt oletuksen perusteella osajonojen $X[i+1..u]$ ja $Y[j+1..v]$ PYA on ainakin yhden mittainen, sillä se sisältää täsmäyksen (u, v). Siten alkuliitteiden $X[1..u]$ ja $Y[1..v]$ PYAn on oltava pidempi kuin lyhyempien alkuliitteiden $X[1..i]$ ja $Y[1..j]$ välisen, minkä perusteella (dominantti) täsmäys paikassa (u, v) kuuluu johonkin muuhun luokkaan l ($l > k$). Pseudotäsmäys (0, 0) muodostaa joukon D_0 ainoan alkion. Kaikkien dominanttien täsmäysten unioniin $U_{1 \le k \le p} D_k$ kuuluvien täsmäysten lukumäärää kuvaa merkintä d.

Jokaiseen luokkaan C_k kuuluu selvästikin vähintään yksi täsmäys, sillä PYAn muodostavassa merkkijonossa $z_1 z_2 \ldots z_p$ jokainen z_k ($1 \le k \le p$) kuuluu indeksinsä mukaiseen luokkaan C_k PYAn määritelmän mukaisesti. Lisäksi on dominantin täsmäyksen määritelmän perusteella ilmeistä, että luokan C_k täsmäyksistä ainakin yksi toteuttaa dominantin täsmäyksen kriteerin, eli tällöin myös jokainen luokka D_k on ei-tyhjä. Dominanttien k-täsmäysten lukumäärä ei kuitenkaan yleisesti ole mitenkään sidoksissa luokan k kaikkien täsmäysten lukumäärään. Ajatellaan jälleen tilannetta, jossa syöttöaakkosto koostuu vain yhdestä symbolista, jolloin luokkaan k kuuluu kappaleen 2.1.4 analyysin mukaisesti $m+n-2k+1$ täsmäystä. Sen sijaan dominantteja näistä on ainoastaan yksi — indeksiparissa (k, k) sijaitseva. Vastaavasti unioniin $U_{1 \le k \le p} D_k$ kuuluisi tuolloin ainoastaan m kappaletta koko ongelman yhteensä mn täsmäyksestä.

Analysoidaan seuraavaksi mielenkiinnosta vielä dominanttien täsmäysten teoreettista enimmäismäärää. Määritelmänsä perusteella kuhunkin luokkaan D_k voi kuulua enintään $m-k+1$ täsmäystä: yksi jokaista matriisin M riviä $k..m$ kohti[9]. Tämä on mahdollista ainoastaan silloin, kun kaikki X:n merkit ovat keskenään erillisiä. Jos nimittäin edes jotain merkkiä esiintyisi X:ssä useammin kuin kertaalleen, enää korkeintaan ensimmäinen niistä voisi muodostaa dominantin 1-täsmäyksen. Dominanttien 1-täsmäysten lukumäärän maksimoimiseksi pitää lisäksi kaikkien X:n merkkien esiintyä Y:n alkuosassa tarkalleen käänteisessä järjestyksessä[10] — muutoin kaikki X:n merkit eivät muodostaisi lainkaan luokan 1 täsmäystä.

Oletetaan nyt, että dominantteja 1-täsmäyksiä on maksimaaliset m kappaletta. Nyt puolestaan jokaiselle riveistä $i \in 2..m$ voisi muodostua dominantti 2-täsmäys ainoastaan sillä ehdolla, että rivin i dominantti 2-täsmäys sijaitsee vähintään m sarakkeen päässä samaisella rivillä sijaitsevasta dominantista 1-täsmäyksestä. Kyseisen vaatimuksen pitää täyttyä, sillä alimman rivin dominantti 2-täsmäys voi sijaita aikaisintaan indeksissä

[9] Luokkaan k kuuluvia täsmäyksiä ei voi esiintyä riveillä $1..k-1$, sillä tällöin X:n alkuliitteen pituus olisi riittämätön k:n mittaisen PYAn muodostamiseksi syötejonon Y kanssa.

[10] Vektorissa Y voi lisäksi esiintyä mielivaltaisen monta sellaista merkkiä, joita ei esiinny X:ssä.

m+1. Muutoin syntyisi ristiriita aikaisemmin tehdyn oletuksen kanssa, jonka mukaan Y:n alussa kaikki X:n merkit esiintyvät päinvastaisessa järjestyksessä, ja tämän indeksialueen pituus on m. Koska siirryttäessä riviä ylemmäs dominanttien 2-täsmäysten sarakeindeksit kasvavat aina vähintään yhdellä, sijaitsee nyt rivin 2 eli samalla ylin 2-täsmäys aikaisintaan sarakkeella 2m–1. Vaaditun kaltaiset dominantit 2-täsmäykset muodostuvat, mikäli Y:ssä X[1]:n ensimmäisen esiintymän jälkeen sijaitsevat nyt kaikki X:n m–1 viimeistä merkkiä käännetyssä järjestyksessä. Luokkaan 3 mahtuu samalla periaatteella m–2 dominanttia täsmäystä, jos X[2]:n toista esiintymää Y:ssä seuraavat X:n m–2 viimeistä merkkiä käänteisessä järjestyksessä. Yleisesti, dominanttien täsmäysten teoreettista kokonaismäärää kuvaa siten aritmeettinen summa $m + (m–1) + (m–2) + \ldots + 2 + 1 = ½m(m+1)$. Sama summalauseke kuvaa myös vektorin Y vähimmäispituutta n silloin, kun dominanttien täsmäysten lukumäärä saavuttaa teoreettisen maksimiarvonsa. Seuraava esimerkki valaissee dominanttien täsmäysten analyysiä.

Esimerkki 2.2: X = ”ABCD”, Y = ”DCBADCBDCD”, p = 4, PYA = ”ABCD”.

			∅	D	C	B	A	D	C	B	D	C	D
			0	1	2	3	4	5	6	7	8	9	10
∅	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	**1**	1	1	1	1	1	1	1
B	2	0	0	0	**1**	1	1	1	**2**	2	2	2	2
C	3	0	0	**1**	1	1	1	**2**	2	2	**3**	3	3
D	4	0	**1**	1	1	1	**2**	2	2	**3**	3	**4**	

Esimerkki syötejonoparista, jolla saavutetaan dominanttien täsmäysten teoreettinen maksimimäärä. Tällöin välttämättä aina $n > m$, kun $m > 1$.

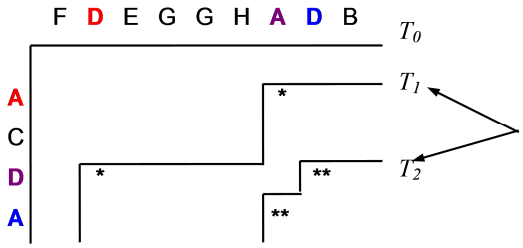
2.3.2 Korkeuskäyrät ja kynnysarvot

Korkeuskäyräksi eli *kontuuriksi* kutsutaan *murtoviivaa*, joka kulkee matriisissa M olevien dominanttien k-täsmäysten kautta. Kullekin joukolle D_k, $0 \leq k \leq p$, muodostuu oma korkeuskäyränsä T_k, joka ei milloinkaan leikkaa korkeuskäyrää $T_{k'}$, $0 \leq k' \leq p$, kun $k \neq k'$. Leikkaamattomuus on ilmeistä, koska dominantteja k–1 -täsmäyksiä ei voi niiden määritelmän mukaisesti esiintyä matriisissa M alueella, jossa syötevektorien alkuosien X[1..u] ja Y[1..v] PYA on jo k:n mittainen. Dominantin pseudotäsmäyksen (0, 0) määräämän luokan D_0 korkeuskäyrä kulkee matriisin nollatta riviä ja saraketta pitkin.

Oletetaan, että matriisin M ylin dominantti k-täsmäys sijaitsee rivillä i sarakkeessa j ($k \leq i \leq m$, $k \leq j \leq n$). Tällöin ei korkeuskäyrä T_k kulje laisinkaan rivien 1, … i–1 kautta. Sen sijaan riviltä i lähtien T_k kulkee kaikkien matriisin loppujen rivien kautta alareunaan asti. T_k rajoittaa vasemmalta ja ylhäältä päin alueen, jonka kaikissa

soluissa X:n ja Y:n alkuliitteiden PYA on k:n mittainen. Aluetta rajoittavat oikealta ja alhaalta päin puolestaan mahdollinen seuraava korkeuskäyrä eli T_{k+1} sekä matriisin M oikea ja alareuna. T_k etenee riviltä i *pystysuoraan alaspäin* niiden matriisin rivien ylitse, joilla ei esiinny kyseisen luokan dominanttia täsmäystä. Jos myöhemmin riviltä $u > i$ löytyy uusi, (i, j):stä katsottuna lähinnä seuraava dominantti k-täsmäys sarakkeesta $v < j$, etenee korkeuskäyrä T_k tällöin *vaakasuoraan riviä u pitkin vasemmalle* sarakkeelta j sarakkeelle v. Seuraavassa on nähtävissä pieni esimerkki korkeuskäyristä.

Esimerkki 2.3: X = ”ACDA”, Y = ”FDEGGHADB”, p = 2, PYA = ”AD”/ ”DA”.



Korkeuskäyrien T_1 ja T_2 rajaaman alueen sisäpuolella sijaitsevat tarkalleen ne solut, joissa X:n ja Y:n alkuliitteiden PYA on pituudeltaan 1.

Matriisissa esiintyvät dominantit 1-täsmäykset on merkitty tähdellä (*****) ja dominantit *2*-täsmäykset kahdella tähdellä (******). Pelkän dominantin 1-täsmäyksen muodostavat merkit on maalattu syötejonoissa **punaisiksi** ja ainoastaan dominantin 2-täsmäyksen muodostavat **sinisiksi**. **Violetin** väriset merkit muodostavat eri PYA-jonoissa ”AD” ja ”DA” eri luokkiin (D_1 tai D_2) kuuluvan dominantin täsmäyksen.

Rivi- ja korkeuskäyräkohtaisesti määriteltäviä lukuja $Kynnys_{i,k}$ kutsutaan riviin i liittyviksi *kynnysarvoiksi*. Yksittäiselle kynnysarvolle $Kynnys_{i,k}$ on olemassa matemaattisesti lausuttuna seuraavanlainen tulkinta:

$$Kynnys_{i,k} = Min\{\, j \mid pituus(PYA(X[1..i], Y[1..j])) = k\}.$$

Toisin sanoen arvo $Kynnys_{i,k}$ tarkoittaa vasemmanpuoleisinta[11] saraketta $j \geq k$, jota pitkin T_k kulkee rivillä i. Ellei tällaista j:tä ole olemassa, $Kynnys_{i,k} = n+1$ eli arvoltaan määrittelemätön. Muussa tapauksessa on voimassa $x_t = y_j$, missä $k \leq t \leq i$. Lisäksi $Kynnys_{i,0} = 0$ $(\forall i \mid 0 \leq i \leq m)$. Kynnysarvot toteuttavat seuraavat kolme lemmaa, jotka *Hunt* ja *Szymanski* ovat artikkelissaan [Hun77] esittäneet ja oikeiksi todistaneet.

Lemmassa 2.1 tarkastellaan rivillä i esiintyviä kynnysarvoja luokasta 1 alkaen. Siinä väitetään, että jos rivillä i kynnysarvot luokille 1, 2, …, k ovat määriteltyjä, niiden sarakenumerot ovat luokittain aidosti kasvavia.

Lemma 2.1: $Kynnys_{i,1}, Kynnys_{i,2}, \ldots, Kynnys_{i,k} < n + 1$
$\Rightarrow Kynnys_{i,1} < Kynnys_{i,2} < \ldots < Kynnys_{i,k}$.

[11] Korkeuskäyrä T_k kulkee *riviä i pitkin* eli mahdollisesti useiden sen sarakkeiden kautta, mikäli rivillä sijaitsee dominantti k-täsmäys.

Todistus: Lemman todistus on verrattain suoraviivainen. Koska määritelmänsä mukaisesti $Y[1..Kynnys_{i,k}]$ on lyhin Y:n alkuliite, jonka kanssa $X[1..i]$ muodostaa k:n mittaisen PYAn, niin silloin Y:stä pitää väistämättä valita ainakin yksi merkki lisää, jotta $X[1..i]$:n ja Y:n alkuliitteen PYAn pituus voisi kasvaa arvoon k+1. Tämä pätee selvästikin mille tahansa k:lle, jolle on voimassa $Kynnys_{i,k} < n + 1$. □

Lemma 2.2: $Kynnys_{i,k-1} < n + 1 \Rightarrow Kynnys_{i,k-1} < Kynnys_{i+1,k} \leq Kynnys_{i,k}$ $(i < m)$.

Todistus: Lemman 2.2 vasemman puolen mukaisesti luokan k–1 kynnysarvo rivillä i on aidosti pienempi kuin seuraavan luokan k kynnysarvo seuraavalla rivillä i+1. Määritelmänsä mukaisesti sarakkeesta $Kynnys_{i,k-1}$ löytyy rivien 1..i *vasemmanpuoleisin luokan* k–1 *täsmäys*, eli syötevektorin Y indeksipaikassa $Kynnys_{i,k-1}$ sijaitseva merkki tarvitaan muodostamaan alkuliitteiden $X[1..i]$ ja $Y[1..Kynnys_{i,k-1}]$ k–1:n mittainen PYA. Jotta PYAn pituus voisi kasvaa pidennettäessä X:n alkuliitettä yhdellä merkillä, täytyy merkin $X[i+1]$ muodostaa nyt dominantti k-täsmäys sellaisen Y:n symbolin kanssa, joka sijaitsee indeksipaikan $Kynnys_{i,k-1}$ oikealla puolella. Siten voidaan todeta lemman vasemman puolen epäyhtälön $Kynnys_{i,k-1} < Kynnys_{i+1,k}$ pitävän paikkansa.

Tarkastellaan seuraavaksi lemman oikeanpuoleista epäyhtälöä. Sen mukaan luokan k kynnysarvot joko pienenevät tai pysyvät ennallaan, kun X:n alkuliitteen pituutta kasvatetaan yhdellä i:stä i+1:een. On ilmeistä, että luokan k kynnysarvot eivät voi milloinkaan kasvaa pidennettäessä X:n alkuliitettä yhdellä, sillä $X[1..i+1]$ sisältää osajononaan jonon $X[1..i]$. Siten myös väitteen oikea puoli pitää paikkansa. Koska relaatiot < ja ≤ ovat transitiivisia, voidaan lemman 2.2 todeta kokonaisuudessaan pitävän paikkansa. □

Lemma 2.3: $Kynnys_{i+1,k} = Min\{j \mid x_{i+1} = y_j$ ja $Kynnys_{i,k-1} < j < Kynnys_{i,k}\}$. *Ellei tällaista* j*:tä ole olemassa,* $Kynnys_{i+1,k} = Kynnys_{i,k}$.

Todistus: Jos merkki $X[i+1]$ ei täsmää yhdenkään Y:n merkin kanssa vaaditulla indeksialueella, ei myöskään k:n mittaisen PYAn muodostamiseksi tarvittavien merkkien lukumäärä vektorista Y vähene X:n alkuliitteen yhdellä pidentymisen johdosta. Jos sen sijaan $X[i+1]$ kuitenkin löytyy Y:stä ehdossa rajatulta alueelta, on ensimmäinen eli vasemmanpuoleisin alueen täsmäyksistä nyt uusi dominantti k-täsmäys, jonka indeksi j on arvoa $Kynnys_{i,k}$ pienempi. Tällöin X:n alkuliitteen pidentäminen i:stä i+1 merkin mittaiseksi lyhentää vaadittavan Y:n alkuliitteen pituutta aikaisemmasta arvosta $Kynnys_{i,k}$ arvoon j. Täten myös lemma 2.3 pitää paikkansa. □

Lemman 2.3 nojalla $Kynnys_{i+1,k}$ – eli luokan k kynnysarvo rivillä i+1 – on sama kuin vastaavan luokan kynnysarvo yhtä riviä ylempänä, jollei indeksialueelta $f = Y[Kynnys_{i,k-1}+1..Kynnys_{i,k}-1]$ löydy yhtään symbolin $X[i+1]$ esiintymää. Muussa

tapauksessa $Kynnys_{i+1,k}$ on vasemmanpuoleisimman indeksialueella *f* sijaitsevan merkin $X[i+1]$ indeksi *j*. Lemmasta seuraa, että jos kahden perättäisen täsmäysluokan *k* ja *k*+1 arvot rivillä *i* poikkeavat toisistaan vain *ykkösellä*, ei luokan *k*+1 kynnysarvo voi pienentyä rivillä *i*+1. Samoin siitä on pääteltävissä, että kun $Kynnys_{i,k}$ saavuttaa ensi kerran arvon *k*, se *ei enää voi pienentyä* *X*:n alkuliitettä pidennettäessä, vaan se on saavuttanut teoreettisen minimiarvonsa.

2.4 Algoritmien toimintaa tehostavat tietorakenteet

Lähes kaikki[12] Wagnerin ja Fischerin algoritmia kehittyneemmät PYA-algoritmit pyrkivät – toimintaperiaatteestaan riippumatta – etsimään syötejonoista pelkät täsmäyskohdat ja jättämään indeksiparit, joissa $X[i] \neq Y[j]$, tarkastelun ulkopuolelle. Tämä on mahdollista algoritmin laskentaa tukevan *esiprosessoinnin*[13] avulla. Luvun 3 PYA-algoritmien esittely paljastaa, että käytännössä yleisin algoritmien suorittama alitehtävä on etsiä tietyn merkin $X[i]$ lähin sijaintipaikka syötevektorista *Y* indeksistä *j* lähtien. Jotta tuohon kysymykseen voitaisiin vastata selaamatta kaikkia *Y*:n merkkejä *lineaarihaulla* [Mau74] aloittamalla paikasta *j*, pitäisi olla ennalta tiedossa, missä *Y*:n indeksipaikoissa merkkiä $X[i]$ esiintyy. Tiedot *X*:n eri merkkien sijaintipaikoista *Y*:ssä voidaan haun nopeuttamiseksi säilöä esiprosessoinnin yhteydessä johonkin hakua tukevaan aputietorakenteeseen, joista yleisimmät ovat *esiintymälista* ja *lähiesiintymätaulukko*, joka voidaan muistitilan säästämiseksi puristaa tarvittaessa kokoon 1-ulotteiseksi *lähiesiintymävektoriksi*. Vastaavasti kynnysarvojen kirjaaminen muistiin erityiseen *kynnysarvovektoriin* nopeuttaa päätöksentekoa siitä, kannattaako laskennan aikana löydetty yksittäinen täsmäys huomioida PYAa määrättäessä. Seuraavissa neljässä aliluvussa tutustutaan tarkemmin näihin tietorakenteisiin.

2.4.1 Esiintymälista

Esiintymä- eli *täsmäyslista* (engl. *matchlist*) on tavallisesti ainoastaan *yhteen suuntaan linkitetty lista*, joka sisältää kutakin aakkoston merkkiä s_g ($g \in [1..\sigma]$) kohti nousevaan (tai laskevaan) suuruusjärjestykseen lajitellun luettelon niistä syötevektorin *Y* indeksipaikoista, joissa merkki s_g esiintyy. Listan loppuun lisätään yleensä pysäytyssolmu, johon tallennetaan joko arvo *n*+1 tai ∞ (tai 0) tiedoksi siitä, ettei käsiteltävää merkkiä enää löydy vektorin *Y* loppuosasta (alkuosasta). Täsmäyslistan avulla löydetään tietyn merkin nykyistä esiintymäkohtaa seuraavan (edeltävän)

[12] Poikkeuksena voisi mainita vaikkapa aliluvussa 3.3.1 esiteltävän *Nakatsun*, *Kambayashin* ja *Yajiman* algoritmin.

[13] Esiprosessoinnilla tarkoitetaan vaihetta, jossa PYAn ratkaisevan algoritmin tarvitsemat aputietorakenteet konstruoidaan ja täytetään syötevektoreiden sisällön ohjaamina.

esiintymän sijaintipaikka vektorissa Y vakioajassa ($O(1)$), mikäli meillä on käytettävissä osoitin, joka osoittaa listassa nykyiseen esiintymäkohtaan. Jos kuitenkin pitää etsiä merkin s_g ensimmäinen esiintymä Y:n mielivaltaisen indeksipaikan j jälkeen (ennen indeksiä j), joudutaan turvautumaan lineaarihakuun, jonka aikakompleksisuus on menetelmän nimen mukaisesti suuruusluokkaa $O(n)$. Esiintymälista voitaisiin rakentaa myös *Skip-listaksi* [Rai96], jolloin päästäisiin hyödyntämään puolitushakua[14] [Mau74], jonka vaatima suoritusaika on $O(\log n)$. Myös esiintymälistan rakentamiskustannus on lineaarinen, koska listan konstruoimiseksi on vektori Y käytävä kertaalleen läpi eri merkkien sijaintipaikkojen kirjaamiseksi muistiin.

Esimerkki 2.4: Σ={“A”, “B”,“C”,“D”}, σ = 4, Y = “ABDBABBCDA”, *pituus*(Y) = 10. Merkkien A, B, C ja D esiintymälistat näyttäisivät seuraavanlaisilta (arvo 11 kunkin listan lopussa on ko. listan *pysäytysalkio*):

A: 1 → 5 → 10 → 11
B: 2 → 4 → 6 → 7 → 11
C: 8 → 11
D: 3 → 9 → 11

Algoritmien kuvausten yhteydessä esiintymälistoihin viitataan seuraavassa luettelossa esitellyin merkinnöin. Merkinnät ovat viimeistä lukuun ottamatta voimassa riippumatta esiintymälistan järjestyksestä (nouseva vai laskeva suuruusjärjestys) ja toteutustavasta (dynaaminen vai vektorimuotoinen).

- Merkintä *EsLista*[s_g] tarkoittaa yleisesti merkin s_g koko esiintymälistaa.
- Funktio *EsLista*[s_g, k] palauttaa merkin s_g k. esiintymän sijaintipaikan vektorissa Y.
- Funktio *SeurEs*[s_g, k] palauttaa puolitushakua käyttäen merkin s_g ensimmäisen esiintymän paikan k jälkeen. Tällöin esiintymälistalle on käytettävä vektoritoteutusta.

2.4.2 Lähiesiintymätaulukko

Kuten edellä todettiin, esiintymälistojen rakentaminen on nopeaa, mutta niiden käyttöä monimutkaistaa toisinaan *suorasaannin* puuttuminen. Haluttaessa löytää syötevektorista vaikkapa merkin ”A” ensimmäinen esiintymä indeksipaikan u jälkeen, jouduttaisiin ottamaan jokin täsmäyslistan toteutuksen tukema hakumenetelmä avuksi. Jos esiintymälista on toteutettu yksinkertaisena linkitettynä listana, ainoa yleispätevä

[14] Puolitushakua päästäisiin hyödyntämään myös silloin, kun listarakenteen sijaan käytettäisiin *vektoria* kunkin merkin esiintymäkohtien kirjaamiseen. Tällöin olisi kuitenkin tarpeen tietää (tai selvittää) syöttöaakkoston merkkien frekvenssit syötteiden ollessa pitkiä, jotta muistinvaraus voitaisiin toteuttaa järkevästi.

ratkaisu on selata merkin ”A” listaa alusta alkaen lineaarihaulla niin pitkälle, kunnes kohdataan ensimmäinen *u*:ta suurempi indeksiarvo.

Lähiesiintymätaulukko (engl. *CLOSEST-matrix*/*-table*) sen sijaan tukee suorasaantia, ja sen rakentaminen mahdollistaa minkä tahansa syöttöaakkostoon Σ kuuluvan merkin s_g lähimmän esiintymän löytämisen vakioajassa syötevektorista *Y* tietystä indeksipaikasta *j* alkaen. Lähiesiintymätaulukko on *2-ulotteinen*: rivi-indekseinä ovat syöttöaakkoston merkit s_1, s_2, ..., s_σ ja sarakeindekseinä taulukon kohdevektorin (oletus: *Y*-vektorin) indeksit laajennettuina indeksipaikalla 0. Yksittäinen lähiesiintymätaulukon soluun (s_g, *j*) tallennettu arvo sisältää normaalisti[15] tiedon siitä, *mistä löytyy merkin* s_g *ensimmäinen esiintymä Y-vektorista indeksipaikan j jälkeen*[16]. Jos kyseisen indeksipaikan jälkeen ei enää löydy tarkastelun kohteena olevaa merkkiä *Y*:stä, soluun on tallennettu arvo *n*+1. *X*-vektorista rakennetusta lähiesiintymätaulukosta käytetään jatkossa lyhennettä *LähiEsTauluX* ja *Y*-vektorista rakennetusta lyhennettä *LähiEsTauluY*. Algoritmien kuvauksissa käytettävä funktio *LähiEsTauluY*[s_g, *k*] palauttaa merkin s_g lähimmän esiintymäkohdan *Y*-vektorissa indeksin *k* jälkeen.

Esimerkki 2.5: Σ = {”A”,”B”,”C”,”D”}, σ = 4, *Y* = “ABDBABBCDA”, *pituus*(*Y*) = 10.
Vektorista Y muodostettu lähiesiintymätaulukko näyttää seuraavalta:

	0	1	2	3	4	5	6	7	8	9	10
A	1	5	5	5	5	10	10	10	10	10	11
B	2	2	4	4	6	6	7	11	11	11	11
C	8	8	8	8	8	8	8	8	11	11	11
D	3	3	3	9	9	9	9	9	9	11	11

Lähiesiintymätaulukon konstruoimiskustannusta kuvaa termi $O(\sigma z)$, missä σ on syöttöaakkoston koko ja *z* on taulukon kohdevektorin pituus[17]. Kustannus ei siis ole lineaarinen suhteessa syötevektorin pituuteen kuten täsmäyslistalla, vaan lähiesiintymätaulukon sekä aika- että tilakompleksisuus riippuvat — paitsi syötteen *Y* pituudesta — myös *syöttöaakkoston koosta*. Tästä voidaan päätellä, että taulukon nopeista hakuoperaatioista saatava hyöty etenkin lyhyillä syötteillä heikkenee aakkoston koon kasvaessa, koska tuolloin esiprosessoinnin kustannukset kasvavat voimakkaasti suhteessa näitä huomattavasti yksinkertaisempien esiintymälistojen rakentamis-kustannuksiin. Toisaalta pitkien syötteiden käsittelyssä, jossa on paljon tarvetta suorasaannille, lähiesiintymätaulukon perustaminen alkaa maksaa itseään takaisin lyhentämällä tuntuvasti ohjelman esiprosessoinnin jälkeistä suoritusaikaa hakuaikojen minimoituessa kaikkiin kilpaileviin aputietorakenteisiin verrattuna.

---

[15] Lähiesiintymätaulukon merkitys voidaan määritellä tarpeen mukaan myös toisin kuin tässä kappaleessa on esitelty. Tällöin taulukko nimetään *symbolijärjestystaulukoksi*. Vaihtoehtoinen määrittelytapa on nähtävissä aliluvussa 3.1.6.

[16] Muutamissa algoritmeissa myös indeksipaikka *j* itse kelpaa seuraavaksi haettavaksi esiintymäkohdaksi.

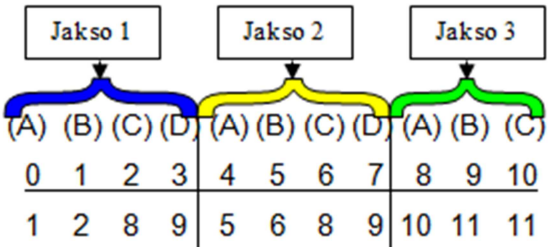
[17] Useimmiten lähiesiintymätaulukkojen muodostamisalgoritmien aika- ja tilakompleksisuus on suuruus-luokkaa $O(n\sigma)$.

2.4.3 Lähiesiintymävektori

Lähiesiintymävektoria (engl. *CLOSE-array*) voidaan luonnehtia ominaisuuksiensa perusteella eräänlaiseksi esiintymälistan ja lähiesiintymätaulukon risteytykseksi. Tässä tietorakenteessa on kombinoitu täsmäyslistojen nopea konstruoitavuus sekä lähiesiintymätaulukon staattinen muistinvaraus, joka helpottaa hakuoperaatioiden suorittamista. Lähiesiintymävektorilla on – toisin kuin lähiesiintymätaulukolla – vain yksi dimensio, jonka pituus on sen kohdevektorin pituus lisättynä indeksipaikalla 0. Algoritmien pseudokoodilistauksissa lähiesiintymävektorista käytetään lyhennettä *LähiEsVekt*.

Lähiesiintymävektori on jaettu syöttöaakkoston koon mukaisiin jaksoihin. Kunkin jakson ensimmäisessä positiossa i oleva arvo ($i \in \{0, \sigma, 2\sigma, ..., \sigma(n \text{ DIV } \sigma)\}$) kertoo merkin s_1 lähimmän sijaintipaikan Y-vektorissa indeksipaikan i jälkeen. Vastaavasti jakson toisessa positiossa i+1 oleva arvo ilmoittaa merkin s_2 lähimmän sijaintipaikan Y:ssä indeksipaikan i+1 jälkeen. Jakson muiden positioiden merkitys on analoginen kahteen ensimmäiseen positioon nähden, joten kunkin täyden[18] jakson viimeisessä positiossa $i+\sigma-1$ on tallennettuna merkin s_σ lähin esiintymä Y:ssä paikan $i+\sigma-1$ jälkeen. Seuraavassa pieni esimerkki lähiesiintymävektorista:

Esimerkki 2.6: Σ = {"A","B","C","D"}, σ = 4, Y = "ABDBABBCDA", *pituus*(Y) = 10.
Y-vektoriin perustuva lähiesiintymävektori näyttää seuraavanlaiselta:



Esimerkin lähiesiintymävektorin yhden jakson oletuspituus on aakkoston koko eli 4. Viimeinen jakso jää nyt kuitenkin vain kolmen mittaiseksi, sillä (n + 1) MOD σ = 11 MOD 4 = 3 ≠ 0.

Jos halutaan verrata esimerkissä 2.6 esitetyn lähiesiintymävektorin sisältämiä arvoja samasta syötevektorista muodostettuun lähiesiintymätaulukkoon esimerkissä 2.5, havaitaan suora yhteys taulukoiden sisältämien arvojen välillä. Vektorin ensimmäiseen jaksoon *LähiEsVekt*[0..3] kuuluvat arvot 1, 2, 8 ja 9 löytyvät lähiesiintymätaulukosta paikoista (s_1, 0), (s_2, 1), (s_3, 2) ja (s_4, 3). Vektorin seuraavan jakson arvot löytyvät puolestaan paikoista (s_1, 4), (s_2, 5), (s_3, 6) ja (s_4, 7). Yleisesti, lähiesiintymävektorin k:nnen jakson arvot löytyvät vastaavasta taulukosta indekseistä (s_1, σ*(k–1)), (s_2, σ*(k–1)+1), … (s_σ, σ*(k–1)+σ–1), kunhan vain vektorin k. jakso on täysi. Lähiesiintymävektoriin tallentuvat toisin sanoen lähiesiintymätaulukon joka σ. diagonaalin sisältämät arvot päädiagonaalilta (solusta (s_1, 0) alkavalta) lähtien.

[18] Lähiesiintymävektorin viimeinen jakso on täysi eli pituudeltaan s:n mittainen vain silloin, kun (n + 1) MOD σ = 0. Muussa tapauksessa viimeisessä jaksossa on vain (n + 1) MOD σ paikkaa.

Kun on tarpeen tietää tietyn merkin s_g lähin esiintymä vektorissa Y position j jälkeen, on aluksi selvitettävä, minkä merkin indeksiä j seuraava esiintymäkohta Y:ssä on tallennettuna *LähiEsVekt*[j]:hin. Tämä saadaan selville laskemalla indeksin j ja aakkoston koon σ välinen jakojäännös ja suorittamalla tämän jälkeen laskutoimitus $j' = j - j \text{ MOD } \sigma + g - 1$. Jos kyseisen laskutoimituksen tulos $j' = j$ eli $j \text{ MOD } \sigma = g - 1$, saadaan haluttu vastaus vakioajassa, koska tuolloin *LähiEsVekt*[j]:hin on tallennettuna juuri merkin s_g seuraava esiintymäkohta Y:ssä. Mikäli laskutoimituksessa ei käynyt näin onnellisesti[19], riippuu jatkokäsittely siitä, onko

1) $j' < j$ eli $g - 1 < j \text{ MOD } \sigma$
   vai
2) $j' > j$ eli $g - 1 > j \text{ MOD } \sigma$.

Tapauksessa 1) *LähiEsVekt*[j]:ssä sijaitsee *järjestysnumeroltaan g:tä suuremman merkin* seuraava sijaintipaikka Y:ssä. Tällöin on siirryttävä tutkimaan paikassa *LähiEsVekt*[j'] olevaa arvoa. Olkoon kyseinen arvo k. Jos nyt $k > j$, arvo k kelpaa suoraan haetuksi esiintymäksi. Muussa tapauksessa pitää etsiä merkin s_g esiintymäkohdista vektorissa Y indeksiarvoltaan pienin j:tä suurempi alueelta (k+1)..*LähiEsVekt*[$j'+\sigma$]. Etsintä voidaan toteuttaa puolitushakua käyttäen edellyttäen, että lähiesiintymävektorin ohella pidetään yllä vektorimuotoista täsmäyslistaa s_g:n sijaintipaikoista Y:ssä. Tämäkään ylimääräinen tietorakenne ei vielä yksinään riitä, vaan lisäksi tarvitaan esiintymälistan käänteiskuvausta, jonka avulla selvitetään, kuinka monensina arvot k ja *LähiEsVekt*[$j'+\sigma$] sijaitsevat merkin s_g esiintymälistassa puolitushaun indeksirajojen määräämiseksi! Koska etsittävällä alueella voi huonoimmassa tapauksessa sijaita korkeintaan σ kappaletta s_g:tä, lähiesiintymävektorista hakemisen kustannus on suuruusluokkaa $O(\log \sigma)$.

Tapauksessa 2) puolestaan paikassa $LähiEsVekt_j$ sijaitsee järjestysnumeroltaan *g:tä pienemmän merkin* seuraavan esiintymäkohdan indeksi. Tällöin siirrytään tutkimaan paikassa *LähiEsVekt*[$j'-\sigma$] esiintyvää arvoa h. Jos nyt $h > j$, vastaus saadaan jälleen suoraan. Muulloin joudutaan suorittamaan puolitushaku samoin reunaehdoin kuin tapauksessa 1), mutta nyt hakualueen rajat esiintymälistassa ovat j+1..*LähiEsVekt*[j'][20]. Kannattaa huomioida, että puolitushaku päättyy molemmissa tapauksissa aina onnistuneesti, jos esiintymälistan loppuun on asetettu pysäytysalkio n+1. Seuraavassa näytetään vielä esimerkki, miten haku lähiesiintymävektorista suoritetaan kaikissa kolmessa edellä analysoidussa tapauksessa.

---

[19] Lähiesiintymävektoria rakennettaessa on huomioitava erikseen tilanteet, joissa laskutoimituksen $j - j \text{ MOD } s + g - 1$ tuottamaa indeksiä j' ei ole lainkaan olemassa, vaan tapahtuu viittaus negatiiviseen indeksipaikkaan. Yksi mahdollisuus ratkaista tästä aiheutuva pulmatilanne on ottaa käyttöön ei-positiivinen indeksialue, josta löytyy kunkin merkin ensimmäinen esiintymäkohta Y:ssä. Ratkaisun tarkempi esittely kuitenkin sivuutetaan tässä.

[20] Hakualueen oikeana rajana on pysäytysalkio, mikäli indeksi $j' > n$ eli viittaa vektorin oikean reunan ulkopuolelle.

Esimerkki 2.7: *Haku lähiesiintymävektorista, kun* Σ = { "A", "B", "C", "D" }, σ = 4, *Y* = "ABDBABBCDA", *pituus*(*Y*) = *10, ja vektorin sisältö on:*

(A)	(B)	(C)	(D)	(A)	(B)	(C)	(D)	(A)	(B)	(C)
0	1	2	3	4	5	6	7	8	9	10
1	2	8	9	5	6	8	9	10	11	11

1° Etsitään ensimmäinen "D" indeksin 3 jälkeen:

$j' = j - j \text{ MOD } \sigma + g - 1 = 3 - 3 \text{ MOD } 4 + 4 - 1 = 3$

$j' = 3 = j \Rightarrow Vastaus = LähiEsVekt[3] = 9$

2° Etsitään ensimmäinen "C" indeksin 3 jälkeen:

$j' = j - j \text{ MOD } \sigma + g - 1 = 3 - 3 \text{ MOD } 4 + 3 - 1 = 2$

$j' = 2 < j = 3, LähiEsVekt[j'] = LähiEsVekt[2] = 8 > j$

$\Rightarrow Vastaus = LähiEsVekt[2] = 8$

3° Etsitään ensimmäinen "B" indeksin 4 jälkeen:

$j' = j - j \text{ MOD } \sigma + g - 1 = 4 - 4 \text{ MOD } 4 + 2 - 1 = 5$

$j' = 5 > j = 4, LähiEsVekt[j'-\sigma] = LähiEsVekt[1] = 2 < j$ (*ei kelpaa*)

$\Rightarrow$ *Etsitään puolitushaulla pienin j:tä suurempi arvo joukosta {4, 6}, joka sisältää kaikki B:n esiintymät indekseissä 3..6*

$\Rightarrow Vastaus = 6$

Lähiesiintymävektorin selkeänä etuna esiintymälistaan verrattuna on itse vektoritoteutus, jossa tarvittavien hakuoperaatioiden tekeminen on huomattavasti yksinkertaisempaa kuin listassa. Lisäksi on oleellista, että hakualueen koko on maksimissaan σ, mikä on eduksi syötteiden ollessa pitkiä. Pienillä syöttöaakkostoilla voitaisiin käyttää jopa lineaarihakua puolitushaun sijaan ja välttyä siten esiintymälistojen ja niiden käänteiskuvauksen kirjaamiselta aputietorakenteiksi. Vertailtaessa puolestaan lähiesiintymävektorin ja -taulukon perustamiskustannuksia voidaan todeta, että kustannussäästö vektorin eduksi lähestyy tuloa $z(\sigma-3)$, missä z on tietorakenteen kohdevektorin pituus[21]. Suurella syöttöaakkostolla ja pitkillä syötteillä suorasaannin mahdollistavan lähiesiintymätaulukon perustaminen vie siten huomattavasti enemmän aikaa kuin tiivistetyssä muodossa olevan vastaavan vektorin rakentaminen. Prosessoinnin aikana tarvittavien hakujen määrä, merkkien jakauma sekä syöttöaakkoston koko ratkaisevat lopulta, millainen tietorakenne soveltuu parhaiten käytettäväksi syöttöaakkoston merkkiesiintymien kuvaamiseen PYA-ongelman eri instansseille.

[21] Lähiesiintymävektori vaatii myös esiintymälistojen ja niiden käänteiskuvauksen muodostamista, ja ne vievät yhteensä tilaa termin 2z verran. Lähiesiintymätaulukkoa käytettäessä näitä ei tarvita.

2.4.4 Kynnysarvovektori

Pituudeltaan m+1 -paikkaista *kynnysarvovektoria* (engl. *threshold array*) *Kynnysarvo*[0..*m*] käytetään aputietorakenteena lähinnä menetelmissä, jotka käsittelevät syötevektoreista konstruoitavissa olevaa matriisia *M*[0..*m*][0..*n*] rivi kerrallaan. Kynnysarvovektorin indeksipaikasta u ($0 \leq u \leq m$) löytyvä arvo antaa tiedon siitä, *mikä on pienin Y:n indeksi – eli vasemmanpuoleisin M:n sarake – jota pitkin u. korkeuskäyrä kulkee oltaessa prosessoimassa riviä i*. Kynnysarvovektorin nollas positio alustetaan nollaksi, kun taas kaikkiin muihin positioihin sijoitetaan alkuarvoksi n+1. Mikäli vektorin jossain positiossa u esiintyy arvo n+1, kyseistä arvoa käsitellään *määrittelemättömänä*, eli korkeuskäyrä T_u ei kulje matriisin *M* paraikaa tarkasteltavan rivin kautta. Koska kynnysarvovektorin arvot ovat sidoksissa eri korkeuskäyrien sijainteihin yksittäisellä rivillä, ovat vektorin arvot aidosti kasvavia ensimmäisen määrittelemättömän arvon sisältävään indeksiin asti, eli toisin sanoen on voimassa lemman 2.1 perusteella määräytyvä ehto

$$(\forall i \mid 0 \leq i, j \leq m: ((i < j) \wedge (Kynnysarvo[i] \leq n)) \Rightarrow Kynnysarvo[i] < Kynnysarvo[j]).$$

Seuraavassa vielä näytetään, miltä kynnysarvovektorin sisältö näyttäisi esimerkin 2.2 mukaisille syötejonoille matriisin kunkin rivin tultua käsitellyksi.

Esimerkki 2.8: *X* = ”ABCD”, *Y* = ”DCBADCBDCD”, *p* = 4, *PYA* = ”ABCD”. *Kynnysarvovektorin sisällön kehittyminen riveittäin: rivin käsittelyn aikana muuttuneet arvot on* ***lihavoitu****.*

	0	1	2	3	4
Alkutilanne	0	11	11	11	11
Rivin 1 jälkeen	0	**4**	11	11	11
Rivin 2 jälkeen	0	**3**	**7**	11	11
Rivin 3 jälkeen	0	**2**	**6**	**9**	11
Lopputilanne	0	**1**	**5**	**8**	**10**

Riveiltä löytyneet täsmäykset:
1-täsmäys (1, 4)
1-täsmäys (2, 3), 2-täsmäys (2, 7)
1-t (3,2), 2-t (3,6), 3-t (3,9)
1-t (4,1), 2-t (4,5), 3-t (4,8), 4-t (4,10)

2.5 Teoreettisia rajoja PYA-ongelman vaativuudelle

PYA-ongelman ratkaisemisen aikavaativuutta on tarkasteltu – ilman varsinaisten ratkaisualgoritmien esittelyä – muutamissa artikkeleissa 1970-luvulta lähtien [Aho76] [Hir78][Won76]. Niissä huomio kohdistuu erityisesti *syöttöaakkoston koon* σ asemaan keskeisenä tekijänä ongelman ratkaisemisessa. Analyysin kannalta on merkityksellistä, voidaanko σ olettaa tunnetuksi – tai ainakin ylhäältä rajoitetuksi, vai pidetäänkö aakkostoa rajoittamattomana. Lisäksi tarkastellaan erikseen tapauksia sen mukaan, tapahtuvatko merkkien väliset vertailut ainoastaan kahden eri syötejonon positioiden välillä, vai sallitaanko lisäksi vertailut yhden ja saman merkkijonon sisällä. Edelleen,

merkitystä analyysin kannalta on myös sillä, voidaanko merkkipareille esitettävät kyselyt esittää pelkästään siten, että vastaus on tyyppiä ”*kyllä/ei*”, vaan ovatko myös *suuruusvertailut* ”<”, ”=” ja ”>” mahdollisia [Hir78].

Aho, Hirschberg ja *Ullman* toteavat [Aho76], että PYAn määrääminen yleisessä tapauksessa ajassa, joka on kertaluokaltaan pienempi kuin $O(mn)$, osoittautuu hyvin vaikeaksi. Kyseinen aikavaativuusluokka kelpaa selvästi kuitenkin kahden jonon PYAn ratkaisemiseen kuluvan ajan *ylärajaksi*, kuten edellä todettiin tarkasteltaessa *Wagnerin* ja *Fischerin* algoritmin toimintaa. Mainittu algoritmihan käsitteli kaikkia mahdollisia syötejonopareja täysin monotonisesti pyrkimättä keräämään tietoja niiden ominaisuuksista.

Kahden jonon PYAn ratkaisemiseksi suoritettava tehtävä saattaa helpottua oleellisesti, mikäli syöttöaakkosto voidaan kiinnittää ennalta: mitä pienempi aakkoston koko, sitä vähemmän joudutaan tekemään merkkien välisiä vertailuja. Erikoisasemassa on pelkästään kahdesta syötemerkistä koostuva *binääriaakkosto*. Sille pystytään kahden yhtä pitkän syötejonon PYA ratkaisemaan tekemällä $2n - 1$ jonojen X ja Y välistä merkkiparivertailua[22]. Riittää, että kummankin jonon ensimmäistä syötemerkkiä verrataan toisen jonon kaikkien merkkien kanssa, eli etsitään täsmäyskohdat matriisin M ylimmältä riviltä ja vasemmanpuoleisimmasta sarakkeesta. Kaikkien muiden rivien i ($1 < i \leq m$) täsmäykset voidaan helposti määrätä vertaamalla rivin merkkiä x_i ensimmäisellä sarakkeella sijaitsevaan merkkiin x_1. Jos ne ovat keskenään samat, ovat rivien sarakekohtaiset täsmäyskohdat keskenään identtiset, ja muussa tapauksessa päinvastaiset binääriaakkoston ollessa kyseessä. Kannattaa huomioida, että asianomainen vertailumäärä on minimaalinen binääriaakkostolle riippumatta siitä, ovatko jonon sisäiset vertailut sallittuja vai eivät [Aho76].

Jos syöttöaakkoston koon annetaan kasvaa kolmeen tai tätä isommaksi, saavutetaan merkittäviä hyötyjä sillä, että myös *jonon sisäiset vertailut* hyväksytään, eli voidaan verrata paitsi jonojen X ja Y välisiä merkkipareja, niin myös kahta saman jonon merkkiä keskenään. Aho, Hirschberg ja Ullman toteavat, että silloin, kun $\sigma \geq 3$, ei PYAa voida enää ratkaista nopeammin kuin ajassa $O(n^2)$ kahdelle yhtä pitkälle syötejonolle olettaen, että ainoastaan X:n ja Y:n merkkien väliset vertailut ovat sallittuja. Tämä voitaisiin todeta vaikkapa seuraavasti: tarkastellaan aakkoston kokoa 3 ja tehdään vastaoletus, että mielivaltaista merkkiparia (x_i, y_j) $(1 \leq x, y \leq n)$ ei olisi ratkaisun löytämiseksi vertailtu laisinkaan. Valitaan ensimmäiseksi syötejonojen täyttövaihtoehdoksi sellainen, että jono X sisältää ainoastaan merkkiä ”A” ja jono Y merkkiä ”B”. Kolmatta syötemerkkiä ”C” ei esiinny kummassakaan jonossa. Tällöin PYAn pituudeksi tulee selvästikin 0. Täytetään toisessa vaihtoehdossa syötejonot siten, että tarkalleen yksi jonon X merkeistä, x_i, asetetaan ”C”:ksi, kun taas muut ovat edelleen ”A”:ta. Vastaavasti jonon Y kaikki muut merkit paitsi y_j ovat edelleen ”B”:tä, mutta y_j vaihdetaan ”C”:ksi. Nyt PYAn pituus muuttuu ykköseksi, sillä pari (x_i, y_j) muodostaa merkkien ”C” välisen täsmäyksen. Tämä

[22] Merkkiparia (x_1, y_1) verrataan selvästikin ainoastaan kertaalleen.

tarkoittaisi nyt sitä, että päätöspuu, jonka mukaan merkkiparin (x_i, y_j) vertailu voitaisiin jättää tekemättä, johtaa kahteen erisuureen PYAn pituuteen. Tästä seuraa, ettei mainitulla päätöspuulla, josta indeksiparin (x_i, y_j) merkkien vertailu puuttui, voitu ratkaista asetettua ongelmaa. Nyt voidaan päätellä, että vastaoletus oli virheellinen ja että kaikkien jonojen *X* ja *Y* indeksiparien vertailujen pitää sisältyä ongelman päätöspuun eri haaroihin, ja indeksipareja on n^2 kappaletta. Sama tarkastelu pätee yhtäläisesti, jos $\sigma > 3$ [Aho76].

Mikäli syötejonojen sisäiset merkkivertailut hyväksytään ja syöttöaakkoston koko σ on rajoitettu, on mahdollista saavuttaa kertaluokkaa $O(n^2)$ tiukempia rajoja PYA-ongelman ratkaisemiselle myös muille kuin yksistään binääriaakkostoa noudattaville syötteille [Aho76]. Tämä perustuu siihen, että voidaan selata syötejonojen merkkejä yksi kerrallaan ja pitää kirjaa tiettyyn hetkeen mennessä kohdatuista syöttöaakkoston merkeistä. Oletetaan seuraavaksi, että $\sigma \leq n$, ollaan selaamassa syötejonoa *Y*, tarkastellaan sen merkkiä y_i ($1 \leq i \leq n$) ja että ainoastaan ”kyllä/ei”-vastauksen antavat kyselyt ovat mahdollisia. Jos nyt $i \leq \sigma$, pystytään enintään $i - 1$ vertailua tekemällä saamaan selville, onko merkki y_i kohdattu jo aikaisemmin. Jos taas $i > \sigma$, selviää viimeistään $\sigma - 1$. vertailun jälkeen, onko merkki y_i esiintynyt jo aiemmin. Vertailujen *enimmäismääräksi* tulee siten

$$\sum_{i=1}^{\sigma-1} i + (n - \sigma)(\sigma - 1) = (\sigma - 1)(n - \sigma/2).$$

Tällä tavoin voitaisiin siten koota merkkikohtaiset esiintymälistat syötevektorista *Y*, ja molemmista syötevektoreista ne pystytään konstruoimaan tekemällä enintään kaksinkertainen määrä vertailuja eli $(\sigma - 1)(2n - \sigma/2)$ kappaletta. Koska naiivi WFI-algoritmi pystyy ratkaisemaan PYAn kahdelle mielivaltaiselle syötejonolle tekemällä tarkalleen n^2 vertailua, saadaan enintään *n*:n kokoiselle aakkostolle vertailujen kokonaismäärän ylärajaksi lauseke $Min\{n^2, (\sigma - 1)(2n - \sigma/2)\}$ [Aho76][23]. Ongelman instansseille, joissa syöttöaakkoston koko on rajoitettu, mutta se ylittää pidemmän syötejonon pituuden *n*, ”kyllä/ei”-tyyppisten merkkivertailujen ylärajaksi määräytyy WFI:n vaatima määrä n^2 [Aho76][Won76].

Käytännön sovelluksissa esiintyy erittäin usein tilanne, että syötejonot ovat selkeästi pidempiä kuin syöttöaakkoston koko. Tämä pätee esimerkiksi DNA- ja proteiiniketjujen kuten myös tekstidokumenttien tarkasteluun. Olettaen esimerkiksi, että kumpikin syötejono on 1 000 merkin mittainen ja syöttöaakkoston koon ylärajaksi määritellään 256, saataisiin edellä esitetylle lausekkeelle ylärajaksi 255 * (2 000 – 128) = 477 360. Tällöin jäädään alle puoleen Wagnerin ja Fischerin algoritmin tekemästä vertailujen määrästä 1 000 000. Mitä pienemmäksi aakkoston koon ja syötejonojen

[23] Jos ei oletettaisi – toisin kuin artikelissa – syötejonojen olevan yhtä pitkiä, voidaan yläraja kirjoittaa muotoon $Min\{mn, (\sigma - 1)(m + n - \sigma/2)\}$.

pituuksien välinen suhde tulee, sitä suuremmaksi suhteellisesti kasvaa naiivin menetelmän tekemien tarpeettomien merkkivertailujen lukumäärä.

Aho, Hirschberg ja Ullman ovat tarkastelleet artikkelissaan [Aho76] myös tarvittavien vertailujen *vähimmäismääriä* ja osoittaneet, että vähintään $\sigma/2(n + \sigma/2)$ vertailua pitää suorittaa, mikäli tarkasteltavassa PYA-ongelmassa syöttöaakkostoon kuuluu korkeintaan yhtä monta symbolia kuin pidempään syötejonoon *Y*. Tätä laajemmille sallituille syöttöaakkostoille merkkiparien vertailujen alarajaksi saadaan $3n\sigma/4$ silloin, kun $n \leq \sigma \leq 4n/3$. Mikäli aakkosto on tätäkin laajempi, tarvitaan yhteensä n^2 vertailua.

Tässä aliluvussa on toistaiseksi oletettu, että vain ”kyllä/ei”-tyyppisiin vastauksiin johtavat kyselyt merkkiparien välillä olisivat sallittuja. Hirschberg analysoi [Hir78] tarpeellisten *merkkiparien vertailujen alarajaa* silloin, kun sekä pienemmyys-, yhtäsuuruus- että suuremmuusvertailut ovat mahdollisia, ja lisäksi syöttöaakkoston koolle ei aseteta ennalta rajoituksia. Hän osoittaa, että tarpeellisten vertailujen minimimäärä on tällöin suuruusluokkaa $n \log n$. Tulos on ymmärrettävissä siten, että kyseinen määrä vertailuja vaaditaan syöttöaakkoston merkkien järjestämiseen silloin, kun aakkostossa esiintyvistä symboleista ei ole saatavilla mitään ennakkotietoa. Siten pituudeltaan $n \log n$ mittainen polku esiintyy jokaisessa päätöspuussa, joka ratkaisee PYA-ongelman, kun pidemmän syötejonon pituus on *n*. Vastaavanlaiseen tulokseen ovat päätyneet myös *Wong* ja *Chandra* [Won76].

2.6 PYAn keskimääräinen pituus

Keskimääräisiä rajoja kahden satunnaisen, tietyn mittaisen ja kiinnitettyä aakkoston kokoa noudattavan merkkijonon PYAn pituudelle on tarkasteltu kirjallisuudessa jonkin verran [Chv75][Dek78][Sim89][Bae96][Kiw05][Lue09]. PYAn odotettavissa oleva pituus voidaan lyhyille syötejonoille ja pienelle syöttöaakkoston koolle laskea tarkasti luettelemalla kaikki mahdollisesti esiintyvät syötejonoparien yhdistelmät, laskemalla kunkin yhdistelmän PYAn pituus ja jakamalla se yhdistelmien lukumäärällä. Otetaan seuraavassa pieni esimerkki *binääriaakkoston* tapauksesta olettaen, että kummankin syötejonon pituus on 2, ja käytössä olevat aakkoston symbolit ovat ”A” ja ”B”.

Esimerkki 2.9: *Mahdolliset syötejonojen X ja Y yhdistelmät, kun m = n = 2, σ = 2 ja Σ =* {”A”, ”B”}.

Rivi	***X***	***Y***	***PYAn pituus***
1	”AA”	”AA”	2
2	”AA”	”AB”	1
3	”AA”	”BA”	1
4	”AA”	”BB”	0
5	”AB”	”AA”	1
6	”AB”	”AB”	2
7	”AB”	”BA”	1
8	”AB”	”BB”	1
9	”BA”	”AA”	1
10	”BA”	”AB”	1
11	”BA”	”BA”	2
12	”BA”	”BB”	1
13	”BB”	”AA”	0
14	”BB”	”AB”	1
15	”BB”	”BA”	1
16	”BB”	”BB”	2
		Rivit yhteensä	**18**

Esimerkkitapauksessa voi esiintyä yhteensä 16 erilaista syötejonoparia. Näistä kymmenessä PYA muodostuu yhdestä merkistä (riveillä 2, 3, 5 ja 9 merkistä ”A”, riveillä 8, 12, 14 ja 15 merkistä ”B” ja riveillä 7 ja 10 kummasta tahansa), neljässä tapauksessa (rivit 1, 6, 11 ja 16) PYA on kahden mittainen, ja kahdella syötteiden kombinaatiolla (rivit 4 ja 13) PYA on tyhjä jono. Tässä tapauksessa keskimääräiseksi PYAn pituudeksi saadaan 1.125 olettaen, että mikä tahansa mahdollisista syötejonopareista on yhtä todennäköinen. Jakamalla kyseinen tulos syötteen pituudella päästään arvoon 0.5625. Kyseistä arvoa voidaan pitää eräänlaisena indikaattorina täsmäyksen esiintymisen todennäköisyydelle syötejonon yhtä merkkiä kohti, kun syötteet ovat tietyn mittaiset mutta satunnaiset, ja aakkoston koko σ on kiinnitetty.

Jos syötejonojen annetaan pidentyä kolmen mittaisiksi pitämällä aakkoston koko ennallaan, tyhjän jonon esiintymisen todennäköisyys syötteiden PYAna pienenee, sillä tällainen tilanne syntyisi silloinkin ainoastaan kahdesti kuten edellä – nyt tyhjään PYAan johtavat syötteet *X* = ”AAA” ja *Y* = ”BBB” tai päinvastoin, mutta mahdollisten syöteparien yhdistelmien määrä nelinkertaistuu 16:sta 64:ään. Sen sijaan ilmestyy 8 uutta tapausta, joissa PYAn pituudeksi tulee 3. Samalla tavalla kuin edellä saadaan keskipituudeksi nyt 1.8125, josta jakamalla kolmella saadaan täsmäyksen esiintymisen todennäköisyydeksi yhtä merkkiä kohti 0.604167 [Chv75]. Vertailemalla tätä lukua

vastaavaan arvoon yhtä lyhyemmille jonoille havaitaan, että todennäköisyys täsmäyksen esiintymiselle kasvoi.

Laskettaessa tarkkoja arvoja pienille syötteiden ja syöttöaakkoston koille voidaan havaita, että kiinnitettäessä syöttöaakkoston koko ja pidennettäessä syötejonoja todennäköisyys täsmäyksen esiintymiselle yksittäistä merkkiä kohti kasvaa vähitellen. Toisaalta, jos syötteiden pituudet kiinnitetään mutta syöttöaakkoston koon annetaan kasvaa, todennäköisyys täsmäykselle pienenee. Kumpikin on ymmärrettävissä intuitiivisesti: mitä pidemmät syötejonot, sitä useampia tapoja muodostuu syötejonojen merkkien kohdistamiselle pareittain [Bae96]. Esimerkiksi binääriaakkostolla täsmäyksen todennäköisyyttä kuvaava luku kasvaa 0.6492188:aan, kun $m = n = 5$. Vastaavasti, sitä mukaa kun aakkosto laajenee, kunkin yksittäisen merkin esiintymisen todennäköisyys selvästikin pienenee satunnaisesti valituilla syötejonoilla. Jos vaikkapa $m = n = 2$, esiintyy 15-merkkisen syöttöaakkoston ollessa käytössä täsmäys yksittäisen merkin kohdalla alhaisella todennäköisyydellä 0.122665 binääriaakkoston vastaavaan lukemaan (0.5625) verrattuna [Chv75].

Tarkasta laskennasta tulee nopeasti vaivalloista, kun syötteiden pituudet ja/tai syöttöaakkoston koko lähtevät kasvamaan. Jos edellisessä esimerkissä mentäisiin vaikkapa pelkkää syöttöaakkoston kokoa kasvattamaan kahdesta kolmeen, tulisi taulukkoon rivejä jo $(\sigma^2)^2 = 3^4 = 81$ riviä. Yleisesti, kun kummankin syötejonon pituus on n ja aakkoston koko on σ, mahdollisia syötejonojen yhdistelmiä esiintyy σ^{2n} kappaletta. Siten satunnaisten, tiettyä aakkoston kokoa noudattavien syötejonojen PYAlle on pyritty laskemaan *ylä- ja alarajoja*, kun syötteen koon annetaan kasvaa rajatta [Bae96][Dan95]. Uusimpien, *Luekerin* esittämien tulosten mukaan syötejonon yksittäisen merkin muodostaman täsmäyksen todennäköisyydelle, josta käytetään lyhennettä γ_2, saadaan binääriaakkostolla vaihteluväliksi (0.788071, 0.826280) [Lue09]. Tekijän mukaan sen määräämiseksi käytettävää tekniikkaa voitaisiin soveltaa vaihteluvälien määräämiseen myös binääriaakkostoa laajemmille syöttöaakkostoille, eli termille γ_k myös silloin, kun $k > 2$. Tällöin kuitenkin rajojen laskenta-aika kasvaa nopeasti parametrin k arvon kasvaessa. Aikaisemmin on saavutettu esimerkiksi kahdeksan kokoiselle syöttöaakkostolle vaihteluväliksi (0.42237, 0.57541) ja 15:n kokoiselle (0.32732, 0.46462) [Dan94].

Binääriaakkostolle saaduista rajoista voi päätellä, että generoimalla pitkiä satunnaisia syötejonoja, jotka noudattavat tasaista merkkijakaumaa, ei pystytä saamaan aikaan syötejonoparia, jossa PYAn osuus lyhyemmän syötejonon pituudesta jäisi vaikkapa 70 %:iin. Täten on selitettävissä, miten pienimmillä aakkostoilla ei työn tulososassa (luvussa 9) ole testattu kovin matalia PYA-osuuksia. Jos haluttaisiin tuottaa alarajaa pienempiä PYA-osuuksia, jouduttaisiin poikkeamaan tasaisen merkkijakauman taustaoletuksesta.

2.7 PYA-ongelman sukulaisongelmia

Paitsi että kahden merkkijonon pisimmän yhteisen alijonon ongelma on jo sinällään mielenkiintoinen ja sovellettavissa usein käytäntöön, sillä on läheinen yhteys myös muutamaan muuhun ongelmaan, joita voidaan tästä syystä pitää PYA-ongelman *sukulaisongelmina*. Näistä erityisesti kahden merkkijonon välisen *lyhimmän editointietäisyyden* määrääminen nousee esiin tässä työssä, sillä alun perin kyseiseen tarkoitukseen kehitetyt algoritmit voidaan verraten vähällä työllä muuntaa PYAn ratkaiseviksi. Tällaisia algoritmeja tarkastellaan lähemmin pääluvussa 4. Seuraavaksi esitellään lyhyesti kolme muutakin ongelmaa, jotka muistuttavat kahden merkkijonon PYA-ongelmaa.

2.7.1 Pisin kasvava alijono

Pisimmän kasvavan alijonon (lyh. PKA, engl. LIS sanoista *longest increasing subsequence*) ongelmaa on tarkasteltu kirjallisuudessa jo aikaisemmin kuin ensimmäisiä kahden merkkijonon PYAn ratkaisevia menetelmiä [Sch61]. Suunnilleen samoihin aikoihin Wagnerin ja Fischerin PYA-algoritmin ilmestymisen kanssa *Fredman* osoitti [Fre75], että yksittäisen jonon, jonka alkiot ovat peräisin lineaarisesti järjestetystä joukosta (esimerkiksi kokonaislukuja), PKA pystytään ratkaisemaan tekemällä enintään $n \log n - n \log \log n + O(n)$ vertailua pahimmassa tapauksessa. Syötejonon $L[1..n]$ pisin kasvava alijono määritellään seuraavasti:

$$PKA(L) = MAX\ z\text{: } \{1 \leq i_1 < i_2 < i_3 < \ldots < i_z, \text{ kun } L[i_1] < L[i_2] < L[i_3] < \ldots < L[i_z]\}$$

Kahden merkkijonon PYA-ongelma pystytään muuntamaan yhden jonon PKA-ongelmaksi tarkastelemalla aluksi lyhemmän syötejonon X täsmäyksiä pidemmässä syötejonossa Y [BHR00]. Täsmäykset kirjataan rivikohtaisesti listoiksi vähenevässä suuruusjärjestyksessä, ja listat liitetään peräkkäin rivi kerrallaan. Vaiheen päättyessä on muodostettuna yhtenäinen lista L kaikista syötejonojen X ja Y välisistä täsmäyksistä. Tällöin L sisältää m kappaletta väheneviä jaksoja, jotka voivat olla myös tyhjiä. Alkuperäisen ongelman PYAn pituus saadaan nyt selville määräämällä listan L PKA:n pituus. Esitetään seuraavassa esimerkki muunnoksen havainnollistamiseksi:

Esimerkki 2.10: *X* = ”HELSINKI”, *Y* = ”BERLIINI”, *m* = *n* = 8, *p* = 5, *PYA* = ”ELINI”
X:n merkkien täsmäyskohdat Y:ssä käännetyssä järjestyksessä (*merkitty aaltosulkeiden sisään*) *sekä listan L sisällön kehittyminen: PKA:han kuuluvat merkit on värjätty* (*ensimmäisen ”I”:n valinnalle on kaksi vaihtoehtoa, eli Y-indekseistä 5 ja 6 valitaan jompikumpi*).

X[1] = ”H”: → ∅ → *L* = []
X[2] = ”E”: → {2} → *L* = [2]
X[3] = ”L”: → {4} → *L* = [2, 4]
X[4] = ”S”: → ∅ → *L* = [2, 4]
X[5] = ”I”: → {8, 6, 5} → *L* = [2, 4, 8, 6, 5]
X[6] = ”N”: → {7} → *L* = [2, 4, 8, 6, 5, 7]
X[7] = ”K”: → ∅ → *L* = [2, 4, 8, 6, 5, 7]
X[8] = ”I”: → {8, 6, 5} → *L* = [2, 4, 8, 6, 5, 7, 8, 6, 5]

Koska PYA-ongelmaa PKA-ongelmaksi muunnettaessa listaan *L* sijoitetaan kaikki alkuperäisen ongelman täsmäykset, on aavistettavissa. että PKA-ongelma ratkeaa nopeasti, kun täsmäysten määrä on vähäinen eli ≤ *n*. Tällöinhän listasta *L* tulee lyhyt, ja PKA saadaan ratkaistua ajassa ≤ *n* log *n*. Jos puolestaan täsmäyksiä on runsaasti, tulee *L*:n pituudeksi pahimmillaan n^2, eikä alkuperäisen ongelman muunto PKA-ongelmaksi selvästikään kannata.

2.7.2 Lyhin yhteinen ylijono

Kahden merkkijonon *lyhimmän yhteisen ylijonon* (lyh. LYY, engl. SCS sanoista *shortest common supersequence*) ongelma on läheistä sukua kahden merkkijonon *X* ja *Y* PYA-ongelmalle. Lyhimmällä yhteisellä ylijonolla tarkoitetaan lyhintä sellaista jonoa *L*, joka sisältää alijonoinaan sekä *X*:n että *Y*:n. Se saadaan muodostettua lisäämällä jonojen PYAan siihen kuulumattomat *X*:n ja *Y*:n merkit samassa järjestyksessä kuin ne ovat näissä syötejonoissa oikeille paikoille PYAan kuuluvien merkkien väliin. Siten PYAn ja LYY:n pituuden välillä vallitsee seuraavanlainen yhteys:

$$pituus(LYY(X, Y)) = m + n - pituus(\mathrm{PYA}(X, Y)).$$

Lyhintä yhteistä ylijonoa on tarkasteltu kirjallisuudessa melko lailla vähemmän kuin PYA-ongelmaa. Ongelmaa ovat tutkineet muun muassa *Maier* [Mai78] sekä *Jiang* ja *Li* [Jia95]. Tässä yhteydessä ei LYY-ongelmaa kuitenkaan tarkastella teoreettisesti lähemmin, vaan tyydytään esittämään esimerkki ongelmien välisestä yhteydestä.

Esimerkki 2.11: *X* = ”HELSINKI”, *Y* = ”BERLIINI”, *m* = *n* = *8*, *p* = *5*, *PYA* = ”ELINI”

1) Kiinnitetään aluksi PYA-jono ”ELINI” osaksi LYY(*X*, *Y*):tä:
   → *L* = *LYY*(*X*, *Y*) = ”ELINI”
2) Sijoitetaan PYAn ulkopuolelle jääneet *X*:n merkit alkuperäisessä järjestyksessään jonoon *L* siten, että *X* esiintyy *L*:n alijonona (tällöin hetkellisesti *L* = *X*).
   → *L* = ”**HELSINKI**”
3) Lisätään PYAn ulkopuolelle jääneet *Y*:n merkit alkuperäisessä järjestyksessään jonoon *L* siten, että ne sijoittuvat oikeille paikoilleen PYAan nähden.
   → *L* = ”**HBERLSIINKI**”

Muodostunut jono *L*[1..11] = ”HBERLSIINKI” sisältää selvästikin sekä jonon *X*[1..8] = ”**HELSINKI**” että *Y*[1..8] = ”**BERLIINI**” alijonoinaan. Esimerkin syötejonojen LYY:n vaatimukset täyttävän lyhyemmän jonon *L’* muodostaminen ei selvästikään onnistu. Sen sijaan lyhimmän yhteisen ylijonon valinta ei ole välttämättä yksikäsitteinen: myös *L’* = ”**BHERLSIINKI**” kelpaisi ratkaisuksi.

2.7.3 Useamman kuin kahden syötejonon PYA

PYA-ongelman ratkaisemiseen tarvittava laskentatyö lisääntyy tuntuvasti, mikäli syötejonojen lukumäärän *k* annetaan kasvaa tässä työssä tarkasteltavista kahdesta. Sen sijaan tehtävän ratkaisevan *perusalgoritmin* toimintalogiikka ei muutu oleellisesti: pitää vain huomioida, että kukin täsmäys koostuu nyt kaikkien *k* syötevektorin pisteiden yhdistelmästä. Esimerkiksi Wagnerin ja Fischerin algoritmi [Wag74] voitaisiin yleistää toimimaan kolmelle syötevektorille *X*, *Y* ja *Z* laajentamalla välitulosten tallentamista varten muodostettava matriisi *M kolmiulotteiseksi* [Irv92]. Matriisin jokaisen dimension nollas indeksi alustettaisiin nollilla seuraavasti samaan tapaan kuin kahden jonon tapauksessa:

$$M[i, j, 0] = M[i, 0, k] = M[0, j, k] := 0.$$

Vastaavasti matriisin muihin soluihin tapahtuisi arvojen tallettaminen seuraavien laskentasääntöjen mukaisesti, jotka muistuttavat niin ikään kahden syötejonon mukaisia sääntöjä:

$1°$ $(x_i = y_j = z_k) \Rightarrow M[i, j, k] = M[i–1, j–1, k–1] + 1$

$2°$ $M[i, j] = Max\{M[i–1, j, k], M[i, j–1, k], M[i, j, k–1]\}$ muulloin

Esimerkiksi syötejonoille *X* = ”A**BB**B”, *Y* = ”A**B**A**B**” ja *Z* = ”**BB**AA” muodostuisi kahden mittainen PYA, joka koostuu merkeistä ”BB”. Laajennetun, *k* syötejonolle toimivan Wagner – Fischer -algoritmin version suoritusaika ja muistintarve ovat suuruusluokkaa $O(n^k)$, mikäli syötejonot ovat keskenään yhtä pitkät. Erityisesti silloin, kun $k = 3$, tulee suoritusajasta *kuutiollinen*. Muun muassa *Irving* ja *Fraser* esittävät artikkelissaan kaksi naiivia algoritmia kehittyneempää menetelmää [Irv92], joilla päästään yleensä $O(n^k)$:ta nopeampaan suoritusaikaan riippuen PYAn pituudesta. PYA-ongelman enempi tarkastelu useammalle kuin kahdelle syötejonolle kuitenkin sivuutetaan tässä työssä.

2.8 Käytettävät lyhenteet ja merkinnät

Seuraavalla sivulla olevaan taulukkoon 2.1 on kerätty tässä työssä esiintyvät usein käytetyt lyhenteet ja niiden merkitykset. Niiden lyhenteiden merkitys, joita ei tarvita työssä tämän tästä, on pyritty kertaamaan ennen niiden käyttöä lukijan työn keventämiseksi.

Taulukko 2.1: *Työssä esiintyvät tärkeimmät lyhenteet merkityksineen.*

d	*dominanttien täsmäysten lukumäärä tarkasteltavassa ongelmassa*
ε	*suurin sallittu maksimi erotukselle m – p*
LE tai LE(X, Y)	*tarkasteltavien merkkijonojen välinen lyhin editointietäisyys*
$L_i(k)$	*Viimeinen Y:n indeksi, josta alkava loppuliite muodostaa k:n mittaisen yhteisen alijonon* $X[i..m]$*:n kanssa. Ellei tällaista ole olemassa,* $L_i(k) = 0$.
m	*syötevektorin X pituus*
M	*2-ulotteinen matriisi, jota tarvitaan algoritmeissa aputietorakenteena*
Min$\{X, Y\}$	*funktio, joka palauttaa argumentteina annettujen lukujen minimin*
Max$\{X, Y\}$	*funktio, joka palauttaa argumentteina annettujen lukujen maksimin*
n	*syötevektorin Y pituus*
p tai p(X, Y)	*pisimmän yhteisen alijonon pituus*
p_{alar}	*pisimmän yhteisen alijonon pituuden heuristinen alaraja*
$p_{ylär}$	*pisimmän yhteisen alijonon pituuden heuristinen yläraja*
P tai P(X, Y)	*tarkasteltavien merkkijonojen välinen puristettu etäisyys*
pituus(X)	*funktio, joka palauttaa argumenttina annetun merkkijonon pituuden*
PYA tai PYA(X, Y)	*tarkasteltavien merkkijonojen pisin yhteinen alijono*
r	*täsmäysten lukumäärä tarkasteltavassa ongelmassa*
σ	*syöttöaakkoston koko*
Σ	*syöttöaakkostoon kuuluvien merkkien joukko*
X	*lyhyempi syötemerkkijono*
Y	*pidempi syötemerkkijono*

3 Tarkat PYA-algoritmit

Käsillä olevassa luvussa esitellään joukko tunnetuimpia, varta vasten PYA-ongelman ratkaisemiseksi kehitettyjä tarkkoja algoritmeja. Tarkastelu on jaettu alilukuihin menetelmien toimintaperiaatteen mukaisesti. Ensimmäisessä aliluvussa 3.1 tarkastellaan *korkeuskäyrä kerrallaan* laskentaa suorittavia algoritmeja, kun taas toinen aliluku 3.2 keskittyy *rivi kerrallaan* prosessoivien PYA-algoritmien esittelyyn. Kummankin ryhmän sisällä pitäydytään algoritmien kronologisessa julkaisemisjärjestyksessä, jotta voitaisiin havaita, miten aikaisemmin kehitettyjä algoritmeja on pyritty tehostamaan.

Korkeuskäyrä kerrallaan prosessoivista algoritmeista esitellään *Hirschbergin I* ja *II* [Hir77], *Hsun ja Dun I* [Hsu84], *Apostolicon ja Guerran I* [Apo87] ja *lineaaritilainen* [Apo85] sekä *Chinin ja Poonin algoritmi* [Chi90]. **Riveittäisistä** menetelmistä esitellään puolestaan *Hirschbergin lineaaritilainen* [Hir75], *Huntin ja Szymanskin* [Hun77], *Mukhopadhyayn* [Muk80], *Hsun ja Dun II* [Hsu84], *Allisonin ja Dixin* [All86], *Apostolicon ja Guerran II* [Apo87], *Kuon ja Crossin* [Kuo89] sekä *Rickin II algoritmi* [Ric94].

Näiden kahden aliluvun ulkopuolella kuvaillaan erikseen vielä aliluvussa 3.3 *Nakatsun, Kambayashin ja Yajiman* [NKY82] kehittämää, edellisten alilukujen teemaan huonosti sopivaa **syötejonojen loppuliitteitä laajentavaa PYA-algoritmia** ja sen toimintaa, sekä kyseisestä menetelmästä johdettua lineaaritilaista *Kumarin ja Ranganin algoritmia* [Kum87]. Tämän lisäksi vielä erillinen aliluku 3.4 on omistettu kahdelle PYA-menetelmälle, jotka suorittavat prosessointia **sekä riveittäin että sarakkeittain**. Näihin menetelmiin kuuluvat *Rickin I* [Ric94] sekä *Goemanin ja Clausenin algoritmi* [Goe99], joista jälkimmäinen toimii lineaarisessa muistitilassa.

Yhteenvetotaulukko kaikista luvussa 3 esiteltävistä algoritmeista löytyy työn tulososasta aliluvusta 9.2. Kyseinen taulukko on jaettu värikoodein menetelmän toimintaperiaatteen (**korkeuskäyrittäin**, **riveittäin**, **loppuliitteitä laajentamalla** tai **kaksisuuntaisesti prosessoivat**) mukaisesti. Jako on täysin yhdenmukainen käsillä olevan luvun alilukuihin jaottelun kanssa.

Tässä luvussa esitellään siten *sekä ei-lineaarisessa että lineaarisessa muistitilassa* suhteessa syötejonojen pituuteen toimivia algoritmeja. Muutamasta alun perin ei-lineaarisesta menetelmästä on myöhemmin esitelty myös lineaaritilainen versio joko säilyttämällä alkuperäismenetelmän toimintaperiaate sellaisenaan [Hir75][Apo85] [Apo92] tai sitä voimakkaasti jäljittelemällä [Kum87]. Lisäksi myös suoraan lineaaritilassa toimiviksi suunniteltuja PYA-algoritmeja on kehitetty [Goe99][Ric00_2] [Cro01][Ili02]. Kaikille lineaaritilaisille menetelmille yhteinen piirre on, ettei niissä ylläpidetä laskennan aikana tietoa kaikista mahdollisista eri ratkaisupoluista PYA-ongelmalle, vaan säilytetään yksinomaan välttämättömät tiedot PYAn pituuden laskemiseksi. Tällä tavoin säästetään (dominanttien) täsmäysten kirjaamiseen tarvittava muistitila, joka on enimmillään suuruusluokkaa $O(mn)$ [Ric94].

3.1 Korkeuskäyrä kerrallaan tapahtuva ratkaiseminen

Tässä aliluvussa luodaan johdannonomainen katsaus tärkeimpiin korkeuskäyrä kerrallaan laskentaa suorittavaan PYA-algoritmeihin. Tarkoituksena on selvittää ja havainnollistaa, miten kyseistä laskentatapaa soveltavat menetelmät ovat kehittyneet vuosien mittaan. Lisäksi analysoidaan jokaisen algoritmin vaatima suoritusaika ja muistintarve sekä esitetään parannusehdotuksia muutamiin menetelmiin.

3.1.1 Hirschbergin I algoritmi (HI1)

3.1.1.1 Yleistä

Hirschberg esitteli vuonna 1977 [Hir77] kaksi korkeuskäyrä kerrallaan laskentaa suorittavaa PYA-algoritmia. Näistä ensimmäinen eli *HI1* (alkuperäisartikkelissa *ALGD*) on aika- ja tilakompleksisuudeltaan suuruusluokkaa $O(pn)$. Pahimmassa tapauksessa, eli silloin kun PYA muodostuu koko lyhyemmästä syötejonosta X, $p = m$. Tällöin algoritmi ei asymptoottisesti suoriudu tehtävästä Wagnerin ja Fischerin naiivia algoritmia vähemmin resurssein. Jos $p << m$, resursseja säästetään kuitenkin huomattavasti. Algoritmin toimintaperiaatteen havainnollistamiseksi voidaan siinä mieltää WFI:n tapaan muodostettavan 2-ulotteinen matriisi, jonka rivit edustavat X:n ja sarakkeet Y:n symboleita, vaikkei tätä HI1-algoritmissa eksplisiittisesti perustetakaan.

Hirschbergin I algoritmissa perustetaan *esiprosessoinnin* yhteydessä jokaiselle syöttöaakkostoon Σ kuuluvalle symbolille *esiintymälistat*. Samassa yhteydessä lasketaan kaikkien aakkoston symbolien esiintymäkertojen lukumäärät eli frekvenssit vektorissa Y vektoriin *Frekvenssit*$[1..\sigma]$. Jokaisen merkin esiintymälistan alkuun sijoitetaan pysäytysalkioksi indeksiarvo 0. Jos tiettyä Σ:aan kuuluvaa merkkiä ei esiinny lainkaan Y:ssä, sen esiintymälistaan sijoitetaan pelkästään kaksi peräkkäistä nollaa indeksiylivuodon välttämiseksi algoritmin suorituksen aikana. Kunkin Y:ssä esiintyvän merkin esiintymälistat kootaan kasvavaan indeksijärjestykseen. Algoritmi käyttää kaksiulotteista taulukkoa D, joka on indeksoitu alueelle $[0..m, 0..m]$. Siihen kerätään algoritmin laskemat kynnysarvot kutakin riviä kohti. Alustuksena taulukon nollas rivi täytetään nollilla. Lisäksi alustetaan muuttuja *YlinRivi* $\leftarrow 0$. Kyseinen muuttuja ilmaisee, miltä riviltä on löydetty ylin dominantti k–1 -täsmäys etsittäessä algoritmin uloimmassa silmukassa luokan k dominantteja täsmäyksiä. Luokan k täsmäyksiähän voi esiintyä aikaisintaan yhtä riviä myöhemmin kuin luokan k–1 täsmäyksiä.

3.1.1.2 PYAn etsintävaihe

Algoritmin laskenta tapahtuu kahdessa sisäkkäisessä silmukassa. *Ulomman silmukan* laskurina toimii muuttuja *k*, joka saa alkuarvokseen 1 ja ilmaisee, monennenko korkeuskäyrän dominantteja täsmäyksiä ollaan etsimässä. Silmukan alussa kunkin *Y*:n merkin esiintymälistan osoittimet siirretään osoittamaan listan viimeistä alkiota[24]. Samassa yhteydessä kopioidaan vektorista *Frekvenssit* vektoriin *MoneskoEsiintymä*[1..σ] tieto kunkin symbolin esiintymiskertojen lukumäärästä. Loogista tyyppiä olevassa apumuuttujassa *löytyi* pidetään kirjaa siitä, onko vielä ehditty löytää yhtään dominanttia *k*-täsmäystä, ja se alustetaan silmukan alussa epätodeksi. Muuttujan *alaraja* arvoksi asetetaan edellisen korkeuskäyrän sijainti yhtä riviä ylempänä. Lemman 2.3 mukaisestihan *k*-täsmäysten tulee sijaita *k*–1 -täsmäysten oikealla alapuolella. Tultaessa silmukkaan ensi kertaa tulee alarajan arvoksi nolla, sillä nollas korkeuskäyrä leikkaa kaikki rivit sarakkeessa 0. Muuttuja *yläraja* puolestaan kuvaa pieninumeroisinta saraketta, josta on jo löytynyt *k*-täsmäys. Silmukan kierroksen aluksi ylärajan arvo on *n*+1, koska yhtään *k*-täsmäystä ei vielä ole ehditty löytää.

Algoritmin *sisäsilmukan* laskurina toimii *i*, joka saa kasvavia arvoja väliltä *YlinRivi..m*. Näiltä riveiltä on nyt tarkoitus lähteä etsimään *k*-dominantteja täsmäyksiä. Kullakin rivillä *i* rullataan merkin *X*[*i*] esiintymälistaa lopusta alkua kohti niin pitkään, kunnes esiintymälistasta löydetään pienin sellainen indeksi, joka on vielä alarajaa suurempi eli on sarakeindeksiltään *j* edeltävän *k*–1 -täsmäyksen indeksiarvoa suurempi. Jos tällainen *j* on olemassa ja se on samalla lisäksi aidosti ylärajaa pienempi, lemman 2.3 vaatimukset täyttyvät. Täten on löydetty luokan *k* dominantti täsmäys paikasta (*i*, *j*), ja sen sarakeindeksi *j* kirjataan muistiin kynnysarvotaulukkoon paikkaan *D*[*k*, *i*]. Samalla uudeksi ylärajaksi päivitetään *j* merkiksi siitä, että seuraavilla riveillä mahdollisesti esiintyvien dominanttien *k*-täsmäysten pitää sijaita nyt sarakkeella, jonka indeksi on *j*:tä pienempi: toisin sanoen täsmäyksen (*i*, *j*) vasemmalla alapuolella. Kun ensimmäinen dominantti *k*-täsmäys löydetään, asetetaan vielä muuttujan *löytyi* arvoksi *tosi* tiedoksi siitä, että korkeuskäyrä *k* on olemassa, eli PYA on vähintään *k*:n mittainen. Jos taasen edellä tarkasteltu *j* ei täyttänytkään lemman 2.3 kriteereitä, asetetaan *D*[*k*, *i*]:n arvoksi nolla, mikä osoittaa, ettei riviltä *i* löytynyt dominanttia *k*-täsmäystä. Silmukan lopussa tutkitaan vielä, esiintyikö tutkitulla rivillä luokan *k*–1 täsmäystä. Jos tällainen löytyi, päivitetään uudeksi *k*-täsmäysten alarajaksi kyseisen täsmäyksen sarake, joka saadaan selville paikasta *D*[*k*–1, *i*]. Tämä mahdollistaa *k*. korkeuskäyrän siirtymisen nykyriviä kauemmas vasemmalle riviltä *i*+1 lähtien. Sisemmän silmukan suoritus päättyy, kun *X*:n viimeinenkin eli *m*. merkki on tutkittu ja selvitetty, muodostaako se dominantin *k*-täsmäyksen.

Ulomman silmukan jokaisen kierroksen lopussa sen laskurin *k* arvoa kasvatetaan yhdellä. Suoritusta silmukassa jatketaan niin pitkään, kunnes kohdataan ensimmäinen

[24] Niiden Σ:n merkkien, joita ei esiinny *Y*:ssä vaan yksistään *X*:ssä, esiintymälistan loppuosoitin osoittaa indeksiarvoon 0. Näiden merkkien esiintymälistassa esiintyy arvo 0 kahdesti.

sellainen *k*, jolle ei enää löydetä yhtään korkeuskäyrän pistettä. PYAn pituudeksi *p* saadaan tällöin *k*–1. Ongelman ratkaisuksi kelpaava jono löydetään nyt käyttämällä avuksi taulukkoa *D*. Jonon etsintä tapahtuu yksinkertaisessa silmukassa, jonka laskurimuuttuja *i* kiertää laskevasti rivi-indeksit *m*:stä 1:een. Lisäksi tarvitaan apumuuttujaa *k*, jonka alkuarvoksi asetetaan löydetty PYAn pituus *p*. Jos kynnysarvotaulukon solusta *D*[*p*, *i*] löytyy nollaa suurempi arvo, on rivillä *i* tällöin luokan *p* dominantti täsmäys, joka asetetaan PYA-jonon *k*. alkioksi. Tämän jälkeen pienennetään *k*:n arvoa ykkösen verran ja jatketaan silmukassa seuraavalle kierrokselle. Koska algoritmi valitsee PYA-jonoon aina sellaiset dominantit *k*-täsmäykset, joiden rivinumero on mahdollisimman suuri, muodostuu Hirschbergin I algoritmin löytämä PYA-jono niistä dominanteista *k*-täsmäyksistä, jotka sijaitsevat matriisiesityksessä mahdollisimman alhaalla vasemmalla siten, että PYA mahtuu kulkemaan niiden kautta. Mikäli lukija haluaa perehtyä algoritmiin syvemmin tarkastelemalla sen pseudokoodia, se löytyy liitesivujen kohdasta 11.1.1.

3.1.1.3 Aika- ja tilavaativuus

Alustustoimenpiteet vievät suoritusajan $O(n+\sigma)$, sillä sekä koko syöttöaakkosto että pidempi syötevektoreista pitää tutkia, ja silmukoiden sisällä olevat lauseet voidaan suorittaa vakioajassa. Alustuksen jälkeistä toiminnallista silmukkaa, joka pseudokoodissa on nimetty tunnisteella *S1*[25], suoritetaan tarkalleen $p+1$ kertaa. Siinä etsitään edustajia kulloinkin tarkasteltavaan täsmäysluokkaan. Yhtä ulkosilmukan kierrosta kohti voidaan sisäsilmukassa joutua käymään koko pidempi syötevektori läpi kertaalleen[26], koska kunkin *Y*:ssä esiintyvän *X*:n merkin esiintymälista voidaan joutua kelaamaan lopusta alkuun yhden ulomman silmukan kierroksen aikana, ja listojen yhteispituus on $O(n)$. Koska silmukoiden käynnistysrivejä lukuun ottamatta kaikki muut lauseet ovat vakioaikaisia, on pahimman tapauksen suoritusaika $O(pn)$.

Muistitilaa joudutaan varaamaan paitsi syötevektoreille, myös esiintymälistoille, frekvenssivektoreille ja taulukolle *D*. Koska syöttöaakkoston koko $\sigma \leq n$ ja esiintymälistojen yhteispituus $O(n)$, ei syöttöaakkoston koko dominoi muistintarvetta. Sen sijaan taulukolle *D* varataan aina $m+1$ riviä ja saraketta, joten algoritmin kokonaistilantarve on $O(Max\{n, m^2\})$. Suuri osa matriisin *D* soluista jää tosin aina käyttämättä.

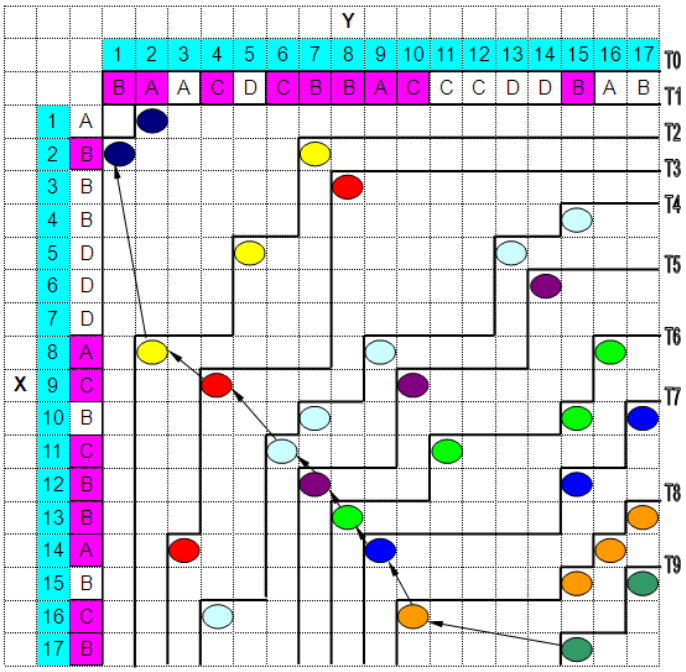
Lopuksi esitetään vielä esimerkki HI1-algoritmin toiminnasta. Samoja esimerkkisyötteitä käytetään myös myöhemmin esiteltävien algoritmien toimintaa

[25] Vastedes merkinnät *S#*, *E#* ja *V#* viittaavat kulloinkin tarkasteltavasta menetelmästä liitteessä esitettyihin pseudokoodilistauksiin: *S* = silmukka, *E* = ehtolause, *V* = valintalause ja *#* on asianomaisen tyypin mukaisen kontrollirakenteen järjestysnumero menetelmässä (esimerkiksi *S1* olisi järjestyksessä menetelmän ensimmäinen pseudokoodissa alkava nimetty toistorakenne).

[26] Vähemmällä selvitään, mikäli tarkasteltavilla matriisin loppuosan riveillä ei esiinny kaikkia *Y*:n sisältämiä merkkejä.

kuvattaessa. Esimerkissä Σ = { A, B, C, D }, σ = 4, X = ”ABBBDDDACBCBBABCB”, Y = ”BAACDCBBACCCDDBAB”, PYA = ”BACCBBACB” / ”ABBACCBAB” ja p = 9. Algoritmi löytää PYAlle ratkaisuksi ensin mainitun esiintymän, sillä se pyrkii valitsemaan mahdollisimman alhaalla vasemmalla sijaitsevan, dominanteista täsmäyksistä koostuvan PYAn.

Esimerkki 3.1: *PYAn ratkaiseminen HI1-algoritmilla, kun m = n = 17, p = 9.*



3.1.2 Hirschbergin II algoritmi (HI2)

3.1.2.1 Menetelmän erityispiirteet ja tietorakenteet

Hirschberg julkaisi edellisessä aliluvussa mainitussa artikkelissaan myös toisen korkeuskäyrittäin etenevän PYA-algoritmin [Hir77]. Tämän Hirschbergin II algoritmin (*HI2*), josta tekijä käyttää nimeä *ALGE*, aikakompleksisuus on $O(n + pe \log n)$, missä $e = \varepsilon + 1$, ja ε tarkoittaa suurinta sallittua arvoa lyhemmän syötemerkkijonon ja PYAn pituuden erotukselle $m - p$, eli toisin sanoen niiden X:n merkkien maksimimäärää, jotka eivät kuulu syötejonojen PYAan. Parametrin ε arvo asetetaan tarkoituksenmukaiseksi ennen algoritmin suoritusta. Aikakompleksisuudesta voi päätellä, että HI2-algoritmia ei ole kehitetty HI1:n kilpailijaksi, vaan pikemminkin täydentämään sitä. HI2 nimittäin toimii parhaiten juuri silloin, kun PYA on pitkä suhteessa X:n pituuteen m, kun taas HI1 soveltuu parhaiten syötteille, joiden PYA on tuntuvasti m:ää lyhyempi. Kannattaa huomioida, ettei HI2 löydä lainkaan ratkaisua, jos $p < m - \varepsilon$.[27] Algoritmin

[27] Ratkaisua ei löydy, ellei algoritmia tuolloin suoriteta toistuvasti esimerkiksi kahdentamalla ε:n arvo epäonnistuneen yrityksen jälkeen.

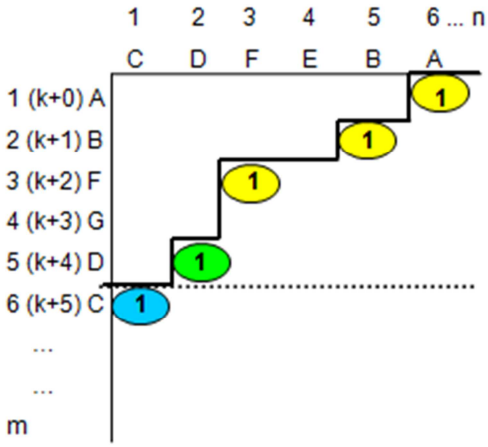
tilakompleksisuus on $O(n+e^2)$. Jos nyt $e \leq n^{½}$, algoritmi toimii lineaarisessa tilassa suhteessa syötemerkkijonojen pituuteen.

Hirschbergin II algoritmia voidaan pitää ensimmäisenä PYA-algoritmina, joka käyttää laskennassaan hyväksi ennakkotietoa PYAn pituudesta. Tosin HI2-algoritmin kohdalla voidaan osuvammin puhua arvauksesta tiedon sijaan: parametri ε asetetaan tekemällä sopiva arvaus, jonka taustalla on aavistus PYAn vähimmäispituudesta tarkasteltavassa ongelmassa. Myöhemmin luvussa 6 tarkastellaan PYA-algoritmeja, joissa *ennakkotietoa PYAn pituudesta hankitaan heuristisin menetelmin*. Kyseiset algoritmit tuottavat aina tarkan ratkaisun riippumatta siitä, miten hyvän arvion PYAn pituudesta heuristiikka pystyy ennalta laskemaan. Parametrin ε valinta vaikuttaa ratkaisevassa määrin sekä algoritmin tuloksellisuuteen — s. o. saadaanko ylipäätään vastausta vai ei — että vastauksen löytyessä laskennan tehokkuuteen. Parametri ε osoittaa, montako X:n merkkiä voidaan enintään ohittaa PYAa muodostettaessa. Käytännössä tämä tarkoittaa sitä, että luokan k dominantteja täsmäyksiä etsitään ainoastaan riveiltä $k..k+\varepsilon$. Näillä riveillä sijaitsevista dominanteista k-täsmäyksistä käytetään HI2-algoritmin yhteydessä nimitystä *kelvollinen* (engl. *feasible*) *dominantti k-täsmäys*. Mitä suurempi arvo ε:lle asetetaan, sitä enemmän X:n merkkejä joudutaan tutkimaan kunkin luokan D_k edustajien etsimiseksi.

Algoritmissa käytetään aputietorakenteita laskennassa kerättyjen tietojen säilömiseen. Vektoriin $F[0..\varepsilon]$ sijoitetaan tieto siitä, miltä riviltä on löydetty viimeksi luokan k täsmäys hyppäämällä vektorista X alusta lukien yhteensä tarkalleen h merkin ylitse, kun $h \in [0..\varepsilon]$. Ellei tällaista k-täsmäystä ole olemassa jollain h:n arvolla, $F[h]$ asetetaan arvoon $F[h–1]$. Rekursiivisen määritelmän terminointia varten määritellään $F[-1] = 0$. Vastaavasti vektorissa $G[0..\varepsilon]$ ylläpidetään tietoa siitä, missä Y-indeksissä sijaitsee vasemmanpuoleisin luokan k täsmäys rivillä $k+h$, missä jälleen $h \in [0..\varepsilon]$, ja ellei tällaista löydy, $G[h]$:lle kopioidaan arvo $G[h–1]$. Rekursiivisen säännön palautumista varten asetetaan $G[-1] = n+1$. Esimerkit 3.2 ja 3.3 kuvaavat, miten dominanttien 1-täsmäysten etsintä HI2-algoritmissa etenee, kun parametrille ε annetaan arvot 4 tai 3, ja miten vektorien F ja G sisältö kehittyy kyseisten täsmäysten etsinnän aikana.

Jotta algoritmissa pystyttäisiin pitämään kirjaa siitä, minkä kaikkien X:n merkkien ylitse pitää hypätä vähintään $m–\varepsilon$:n pituisen PYAn määräämiseksi, tarvitaan opastetietuevektoreita $P[0..\varepsilon]$ ja $UusiP[0..\varepsilon]$ sekä taulukkoa *Opaste*, jonne kootaan tiedot kaikista ohitettavista riveistä kullakin mahdollisella PYAn ratkaisupolulla parametrin ε valinnan mukaisesti. Vektori P on luonteeltaan dynaaminen ja alustetaan algoritmin suorituksen käynnistyessä *nollaosoittimilla*. Ne tulkitaan vektorin P paikoissa $h \in [0..\varepsilon]$ loppuosoittimiksi merkityksellä, ettei ole tarpeen hypätä yhdenkään riveistä $1..k+h$ ylitse vektorissa X syötejonojen k:n mittaisen PYAn muodostamiseksi.

Esimerkki 3.2: *Kelvollisen dominantin k-täsmäyksen määrittely.*



Kuvan esimerkissä parametrilla ε on arvo 4. Tämä tarkoittaa sitä, että kelvollisia dominantteja 1-täsmäyksiä voi esiintyä ainoastaan riveillä 1-5. Pisteviiva kuvaa siten kelvollisten 1-täsmäysten sijaintialueen alarajaa. Keltaisella maalatut täsmäykset paikoissa (1,6), (2,5) ja (3,3) ovat kaikki kelvollisia, samoin vihreä paikassa (5,2). Sen sijaan paikassa (6,1) oleva dominantti 1-täsmäys ei enää sijaitse kelvollisella alueella ε:n arvolla 4. Mikäli ε:n arvoa pienennettäisiin 3:een, myös täsmäys (5,2) rajautuisi pois tarkastelusta.

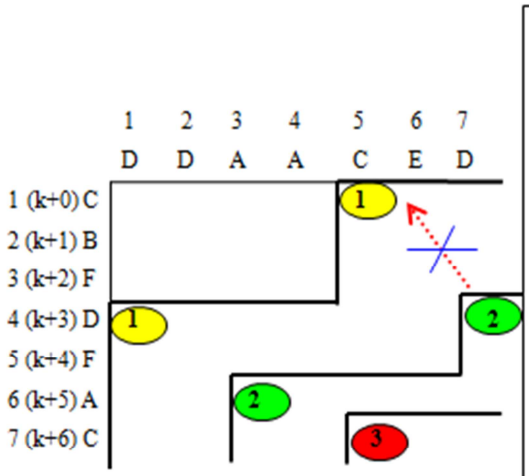
Esimerkki 3.3: *Vektorien F ja G merkitys Hirschbergin II algoritmissa*:
X = "ABFGDC", Y = "CDFEBA", $m = n = 6$, $\varepsilon = 4$, $k = 1$.

$F[0..\varepsilon] = F[0..4] =$ 1 *2 3* **3** *5*
$G[0..\varepsilon] = G[0..4] =$ 6 *5 3* **3** *2*

Vektorit F ja G sisältävät yllä olevat arvot, kun kaikki kelvolliset D_1-täsmäykset on etsitty. Vektorin indeksi ilmaisee, monenko rivin eli X:n merkin ylitse on jouduttu hyppäämään. Paikassa (1, 6) sijaitseva D_1-täsmäys löytyy hyppäämättä yhdenkään X:n rivin yli. Sen sijaan kursiivilla vektoreihin merkittyjen täsmäysten (2, 5), (3, 3) ja (5, 2) saavuttamiseksi on jouduttu hyppäämään yhden tai vastaavasti kahden tai neljän rivin yli mainitussa järjestyksessä. Koska lihavoidut arvot paikoissa F_3 = **3** ja G_3 = **3** ovat samat kuin F_2 ja G_2, ei löytynyt D_1-täsmäystä, johon päästäisiin hyppäämällä kolmen rivin ylitse. Tämä selittyy sillä, että merkki $x_{3+k} = x_4$ = "G" ei muodosta dominanttia 1-täsmäystä.

Jos suorituksen aikana osoittautuu mahdottomaksi kerätä k:n pituinen PYA hyppäämällä tarkalleen h rivin ylitse X:n alkuosassa, asetetaan *UusiP*[h]:n arvoksi *roskaosoitin* -1. Tällaista polkua ei selvästikään ole enää tarpeen ylläpitää. Muussa tapauksessa apuvektorin *UusiP* indeksipaikkaan h tallennetaan osoitin tietueeseen, joka sisältää kolmikon <i–1, *lkm*, P[h–*lkm*]>. Tietueen eri kentät kuvaavat rivillä i oltaessa, että merkkiä X[i] edeltävät *lkm* merkkiä pitää ohittaa, ja polun alkua kohti seuraava opastetietue löytyy paikasta P[h–*lkm*]. Seuraavassa esitetään pieni esimerkki asian valaisemiseksi. Kannattaa huomioida, että etsittäessä dominantteja k-täsmäyksiä riveiltä $h \in [k..k+\varepsilon]$ päivitetään kullakin h:n arvolla vektorin *UusiP* eikä suinkaan vektorin P sisältöä. Vektorin P sisältö päivitetään kopioimalla siihen *UusiP*:n sisältö vasta kaikkien kelvollisten dominanttien k-täsmäysten löydyttyä. Näin joudutaan toimimaan, ettei löydetyn k-täsmäyksen generoinutta luokan k–1 edeltäjäsolmua mahdollisesti menetettäisi.

Esimerkki 3.4: *Opastetaulukon ja -tietueiden kehittyminen, kun $\varepsilon = 4$.*



Opastetietue P:n sisältö luokan *k* valmistuttua:

Alustus: $P[0] = P[1] = P[2] = P[3] = P[4] = 0$.

$k = 1$: $P[0] = 0$, $P[1] = -1$, $P[2] = -1$,
$P[3] = a \rightarrow < 3, 3, P[0] >$,
$P[4] = -1$

$k = 2$: $\mathbf{P[0] = P[1] = -1}$
$\mathbf{P[2] = b \rightarrow < 3, 2, P[0] >}$, $P[3] = -1$,
$P[4] = c \rightarrow < 5, 1, P[3] >$

$k = 3$: $P[0] = P[1] = P[2] = P[3] = -1$
$P[4] = c$ [täsmäys (6,3) generoi (7,5):n]

Opastetaulun sisältö:

$Opaste[P[0]] = 0$
$Opaste[P[1]] = -1$
$Opaste[P[2]] = b \rightarrow < 3, 2, P[0] >$
$Opaste[P[3]] = a \rightarrow < 3, 3, P[0] >$
$Opaste[P[4]] = c \rightarrow < 5, 1, P[3] >$

Esimerkistä havaitaan, että luokan 3 ainoa täsmäys saavutetaan hyppäämällä neljän *X*:n merkin ylitse. Nämä merkit selviävät opastetaulun ketjusta, joka alkaa indeksistä $P[4] = c$. Ketjun tietosisällön mukaan pitää ensiksi ohittaa rivillä 5 oleva merkki ”F”, minkä jälkeen jatketaan seuraamalla opastetietuetta *P*[3], jonka mukaisesti vielä rivien 1–3 merkit ”C”, ”B” ja ”F” ohitetaan käännetyssä järjestyksessä. Syötteiden PYAn ”DAC” muodostaisivat siten merkit *X*[4], *X*[6] ja *X*[7] sekä *Y*[1], *Y*[3] ja *Y*[5].

Esimerkistä havaitaan myös, että ensimmäinen dominantti 2-täsmäys löytyy vasta riviltä 4. Jos algoritmissa olisi menty kirjaamaan *P*[0]:n arvoksi heti -1, kun 2-täsmäystä ei toiselta riviltä löytynyt, täsmäyksestä (4, 7) katkeaisi yhteys sen generoineeseen edeltäjään (1, 5) opastetaulussa. Liian aikaisen päivityksen aiheuttama virhetilanne on merkitty kuvaan näkyviin. Tästä syystä pitää viivyttää vektorin *P* päivittämistä, kunnes kaikki luokan *k* kelvolliset dominantit täsmäykset on löydetty, ja päivitykset jätetään siihen asti odottamaan apuvektoriin *UusiP*.

3.1.2.2 Suorituksen vaiheet

Algoritmin toiminta kokonaisuudessaan jakautuu kolmeen vaiheeseen. *Alustusvaiheessa* nollataan vektorien *F* ja *G* sekä *P*[0]:n sisältö, ja muut *P*:n indeksipaikat alustetaan arvolla -1. Lisäksi muodostetaan esiintymälistat *Y*:n symboleille. *Varsinainen PYAn etsintä* tapahtuu kahdessa sisäkkäisessä silmukassa. Näistä ulompi etsii korkeuskäyrän *k* kelvolliset dominantit täsmäykset luokka kerrallaan. Silmukan joka kierroksen alussa alustetaan apumuuttujat *imax* ja *jmin* arvoihin 0 ja *n*+1. Tämän jälkeen käynnistyy

sisempi silmukka, jossa tehdään kutakin k:n arvoa kohti $Min\{e, m - k + 1\}$ kierrosta[28]. Sisemmän silmukan laskurina toimii h, joka kuvaa ohitettujen rivien lukumäärää.

Kultakin tarkasteltavalta riviltä etsitään dominantteja k-täsmäyksiä. Jos tällaista ei löydy, asetetaan $F[h]$:lle arvo *imax* ja $G[h]$:lle arvo *jmin*, jotka ovat viimeksi löydetyn k-täsmäyksen rivi- ja sarakeindeksit[29]. Lisäksi $UusiP[h]$ saa arvokseen roskaosoittimen -1 tiedoksi siitä, ettei merkki $X[k + h]$ muodostanut dominanttia k-täsmäystä. Jos riviltä $k+h$ kuitenkin löydetään k-täsmäys, päivitetään sen rivi- ja sarakeindeksit muuttujiin *imax* ja *jmin* sekä samalla $F[h]$:hon ja $G[h]$:hon. Jos täsmäyksen generoinut k–1 -täsmäys sijaitsee tarkalleen riviä ylempänä, ei edeltäjäviittaustietoja tarvitse päivittää. Muussa tapauksessa sen sijaan löydetyn k-täsmäyksen ja sen edeltäjän väliin jäi ainakin yksi rivi, jonka ylitse täytyy hypätä. Tällöin muodostetaan uusi opastetietue $UusiP[h]$, jolle varataan muistia dynaamisesti. Opastetaulun muistipaikkaan $Opaste[UusiP[h]]$ talletetaan nyt kolmikko $<i–1, lkm, P[h – lkm]>$, joka kertoo riviohituksen alkukohdan, pituuden sekä linkin seuraavaan opastetietueeseen.

Jos sisemmän silmukan suorituksen aikana löydettiin ainakin yksi dominantti kelvollinen k-täsmäys, kopioidaan silmukan päätyttyä vektorin *UusiP* sisältö viimein vektoriin P ja siirrytään ulommassa silmukassa etsimään k+1 -täsmäyksiä. Ellei kuitenkaan luokkaan k saatu yhtään edustajaa, vektorin *UusiP* kopiointi jätetään tekemättä ja algoritmin varsinainen laskentavaihe päättyy.

Silmukoiden lopetettua toimintansa alkaa *PYAn keräämisvaihe*. Ensiksi on selvitettävä, mikä on pienin sellainen vektorin P indeksi h, jossa osoittimena on jotain muuta kuin roskaosoitin -1. Ellei tällaista indeksiä ole olemassa, algoritmi ei annetulla parametrin ε arvolla löytänyt ratkaisua, eli syötteiden PYA on lyhyempi kuin $m - \varepsilon$. Jos vaatimukset täyttävä indeksi h kuitenkin löytyy, tiedetään asetetun PYA-ongelman ratkenneen, ja PYAn pituudeksi saadaan $k - 1$. Muistipaikassa $F[h]$ oleva arvo kuvaa nyt viimeisen PYAan kuuluvan merkin rivi-indeksiä. Koko PYA-jono saadaan artikkelin kirjoittajan mukaan[30] koottua selaamalla läpi $Opaste[P[h]]$:sta alkava linkkiketju läpi ja kirjaamalla poistettavien X:n merkkien indeksit erilliseen vektoriin *ohita*. Kun kaikki ohitettavat X-indeksit on jo ehditty kirjata, voidaan selata X alusta loppuun päin riville $F[h]$ asti. Jos indeksiä i ei löydy vektorista *ohita*, talletetaan $X[i]$ PYA-jonoon, muutoin se jätetään tallentamatta.

3.1.2.3 Algoritmin analyysi

Koska HI2 löytää onnistuessaan aina saman ratkaisun kuin HI1, ei esimerkin 3.1 kaltaista esitystä algoritmin toiminnasta tässä anneta. Kyseisen esimerkin syötteille HI2

[28] Näin varmistutaan, ettei yritetä tarkastella X:n merkkejä, joiden indeksi > m. Tätä ei artikkelissa kuitenkaan ole huomioitu.
[29] Tai pseudoarvot 0 ja n+1, ellei k-täsmäyksiä vielä ole ehditty löytää.
[30] PYA-jono saataisiin koottua haluttaessa jo seurattaessa opastetietueita lopusta alkuun, jolloin vältyttäisiin ylimääräiseltä vektorin X selaamiselta.

löytää ratkaisun, mikäli parametriksi ε valitaan 8 tai sitä suurempi arvo, koska X on pituudeltaan 17 ja PYAn pituus on 9. Jos parametrille ε annetaan arvoksi 8, jäisivät dominantti 3-täsmäys paikassa (14, 3) sekä dominantti 4-täsmäys paikassa (16, 4) tarkastelematta.

Algoritmin alustustoimenpiteet vaativat ajan $O(n)$, sillä niitä dominoi esiintymälistojen perustaminen syötevektorin Y merkeille. Laskentavaiheessa ulompaa silmukkaa suoritetaan – olettaen, että algoritmi löytää ratkaisun – p kertaa. Sisempää silmukkaa puolestaan suoritetaan jokaista täsmäysluokkaa k kohden enintään ε kertaa. Funktion *SeurEs* kutsuminen sisemmässä silmukassa vaatii työmäärän $O(\log n)$, jos esiintymälista toteutetaan vektorimuotoisena ja siihen sovelletaan puolitushakua. Siten algoritmin aikakompleksisuus on suuruusluokkaa $O(n+p\varepsilon \log n)$.

Jos algoritmin halutaan aina löytävän ongelmalle ratkaisun, eli silloinkin, kun parametri ε valitaan ensi yrityksellä liian pieneksi, voitaisiin ajatella kokeiltavaksi ε:n paikalle kakkosen potensseja järjestyksessä ykkösestä alkaen niin kauan, kunnes $2^{\varepsilon} \geq m - p$. Tällöin saadaan kokonaistyömääräksi $2pl \log n + 4pl \log n + \ldots + \varepsilon pl \log n$, missä l on aikakustannuksen multiplikatiivinen kerroin yhtä suorituskertaa kohti. Koska termeistä muodostuu geometrinen sarja, sen summan ylärajaksi saadaan $2pel \log n$. Koska $\varepsilon < 2(m+1-p)$, niin PYA saadaan ratkaistuksi ajassa $O(n + p(m+1-p) \log n)$.

Algoritmi tarvitsee muistitilaa – paitsi syötevektoreita varten, niin myös vektoreille F ja G sekä polkujen ylläpitoa varten opastevektoreille P ja *UusiP* sekä taulukolle *Opaste*. Syötevektoreiden viemä muistitila on suuruusluokkaa $O(n)$, Vektoreissa P ja *UusiP* on kummassakin e indeksipaikkaa, ja opastetauluun tallennettavien viittausten teoreettinen kokonaismäärä on siten $\varepsilon(\varepsilon - 1)/2 = O(\varepsilon^2)$. Algoritmin kokonaistilantarve on siten $O(Max\{n, \varepsilon^2\})$.

Algoritmi suorittaa verrattain paljon turhaa laskentaa, jos syötevektoreilla ei ole pitkää yhteistä alkuliitettä (kts. liite 11.1.2). Jos nimittäin etsittäessä korkeuskäyrän k kelvollisia täsmäyksiä riveiltä $k..k+\varepsilon$ havaitaan, ettei yksikään niistä sijaitse ennen riviä $k + z$, missä $0 < z \leq \varepsilon$, seuraa k-täsmäyksen määritelmän perusteella, ettei luokkaan $k + u$ kuuluvia täsmäyksiä voi löytyä rivin $k + z + u$ yläpuolelta. Mitä enemmän z lähestyy ε:ia, sitä useampia turhia rivejä tutkitaan jokaisella sisemmän silmukan kierroksella. Tämä tehottomuuden lähde olisi helposti vältettävissä tutkimalla vaikkapa vektorin F arvoja ulomman silmukan kierroksen päätteeksi luokan k kaikkien kelvollisten dominanttien täsmäysten löydyttyä. Luokan k+1 edustajien etsiminen aloitettaisiin vasta riviltä k+z, missä z on pienin vektorin F indeksi, johon on tallennettu nollaa suurempi arvo. Tarkasteltaessa uudelleen esimerkkiä 3.4 nähdään, ettei luokan 3 täsmäyksiä kannata vielä alkaa etsiä riveiltä 3 ja 4, kuten algoritmissa tehtäisiin, sillä kyseisen luokan täsmäyksiä voi löytyä ainoastaan ylimmän 2-täsmäyksen alapuolelta, eli riviltä 5 lähtien (toisen korkeuskäyrän valmistuttua $F[0] = F[1] = 0$, mutta $F[2] = 4$).

3.1.3 Hsun ja Dun I algoritmi (HD1)

3.1.3.1 Jalostettu versio Hirschbergin I algoritmista

Hsu ja *Du* esittelivät vuonna 1984 [Hsu84] kaksi PYA-algoritmia, joista vain ensimmäinen, tekijöiden *A1*:ksi nimeämä algoritmi, suorittaa laskentaa korkeuskäyrä kerrallaan. Algoritmista käytetään tässä työssä lyhennettä *HD1*, ja sen lähtökohtana on ollut aliluvussa 3.1.1 esitelty Hirschbergin I algoritmi, johon tekijät esittävät dominanttien täsmäysten etsintätapaan kohdistuvia parannusehdotuksia. Sen sijaan PYAn keräämisvaiheeseen he eivät esitä muutoksia, koska kyseisen vaiheen asymptoottinen kustannus $O(n)$ on Hirschbergin I algoritmissa verrattain vähämerkityksinen kokonaistyömäärään nähden.

HD1-algoritmin laskentamalli muistuttaa hyvin paljon HI1:n suoritustapaa. Tosin dominanttien *k*-täsmäysten kirjaamiseen ei käytetä Hirschbergin tapaan 2-ulotteista matriisia *D*, vaan ne korvataan esiintymälistoja sisältävällä vektorilla sekä muuttujalla *kynnysarvo*, joka kuvaa vasemmanpuoleisimman löydetyn dominantin *k*-täsmäyksen sarakeindeksiä. Kynnysarvoksi alustetaan ∞ aina alettaessa etsiä uuden korkeuskäyrän pisteitä. Hsu ja Du kiinnittävät artikkelissaan huomiota erityisesti siihen, ettei HI1:ssä hyödynnetä mitenkään tietoa täsmäyksen luokan yksikäsitteisyydestä. Jos nimittäin tiedetään täsmäyksen (*i*, *j*) kuuluvan jo korkeuskäyrälle *k*, on turhaa asettaa sitä enää ehdolle etsittäessä seuraaville korkeuskäyrille kuuluvia täsmäyksiä. HD1:ssä asetetaan *rivi- ja merkkikohtaiset rajat* määrittämään, mitkä x_i:n esiintymistä *Y*:ssä ovat kelvollisia muodostamaan luokan *k* täsmäyksen rivillä *i*.

Vasen raja on toteutettu rivinumeroin indeksoidulla vektorilla *alaraja*[1..*m*], joka ilmaisee merkin x_i ensimmäisen *käyttökelpoisen esiintymän* järjestysnumeron rivillä *i*. Vektorin jokaiseen positioon asetetaan alkuarvoksi ykkönen, ja algoritmin suorituksen edetessä vektoriin tallennetut arvot eivät voi milloinkaan pienentyä. Ajatuksena on, että dominantin *k*-täsmäyksen (*i*, *j*) löydyttyä ei enää myöhemmin yritetä löytää samaiselta riviltä luokkaan $l > k$ kuuluvaa dominanttia täsmäystä sarakkeesta *j* tai sen vasemmalta puolelta, sillä tällaisia ei voi olla olemassa lemman 2.3 perusteella. Merkin x_i esiintymä rivillä *i* on käyttökelpoinen niin kauan, kunnes sen luokka saadaan algoritmissa selvitettyä, minkä jälkeen se rajataan tarkastelujen ulkopuolelle.

Merkkikohtaista oikeaa rajaa ylläpidetään vektorissa *yläraja*[1..σ]. Saavuttaessa riviltä $i - 1$ riville *i* paikkaan *yläraja*[x_i] tallennettu arvo ilmaisee, mikä on merkin x_i järjestysnumeroltaan oikeanpuoleisin esiintymä, jonka jälkeen rivillä *i* ei voi enää sijaita *k*-täsmäyksiä. Vektori alustetaan kunkin korkeuskäyrän etsinnän alkaessa arvoon *frekv*[x_i], joka kuvaa merkin x_i esiintymien lukumäärää vektorissa *Y*. Alustusvaiheessa oletetaan luokkaan 0 kuuluvan pseudotäsmäyksen sijaitsevan paikassa (0, 0).

3.1.3.2 Neljä eri tapausta rivien käsittelyssä

Etsittäessä luokan *k* dominantteja täsmäyksiä riviltä *i* algoritmi erittelee neljä tapausta. *Ensimmäisessä tapauksessa* paikkaan *alaraja*[*i*] tallennettu arvo on suurempi kuin merkin x_i esiintymiskertojen lukumäärä rivillä *i*, eli yhtään käyttökelpoista merkin esiintymää ei rivillä enää ole. Tällöin voidaan rivin käsittely lopettaa saman tien tarpeettomana. Kaikissa muissa tapauksissa (numerot 2–4) käyttökelpoisia x_i:n esiintymiä kyseisellä rivillä on vielä jäljellä, eli on voimassa $alaraja[i] \leq frekv[x_i]$.

Tapauksessa 2 riviltä *i* ei löydy luokan *k* täsmäystä. Tällöin kopioidaan $yläraja[x_i]$ samaan indeksiarvoon kuin *alaraja*[*i*], jolloin jatkettaessa dominanttien *k*-täsmäysten etsintää seuraavilta riveiltä tiedetään, etteivät merkin x_i esiintymät $yläraja[x_i]$:stä eteenpäin voi enää muodostaa *k*-täsmäystä[31].

Tapauksessa 3 riviltä *i* löydetään tarkalleen yksi luokan *k* ei-dominantti täsmäys. Tästä pystytään epäsuorasti päättelemään, että jollain edeltävällä rivillä $i' < i$ on oltava voimassa $x_{i'} = x_i$ ja merkki muodostaa siellä dominantin *k*-täsmäyksen[32]. Nytkin kopioidaan $yläraja[x_i]$ arvoon *alaraja*[*i*], sillä tiedetään, ettei mainitusta x_i:n esiintymälistan indeksipaikasta lähtien enää kannata etsiä myöhempiä dominantteja *k*-täsmäyksiä. Samalla siirretään *alaraja*[*i*]:tä yhdellä esiintymällä eteenpäin *X*[*i*]:n täsmäyslistassa, sillä nyt tiedetään juuri löydetyn ei-dominantin täsmäyksen (*i*, *j*) kuuluvan luokkaan *k*, joten se ei voi kelvata enää millekään seuraavista korkeuskäyristä.

Viimeisessä eli neljännessä tapauksessa löydetään riviltä *i* sarakkeesta $EsLista[x_i, alaraja[i]] = j$ dominantti *k*-täsmäys, jota voi mahdollisesti seurata useita ei-dominantteja saman luokan täsmäyksiä. Koska muuttuja $yläraja[x_i]$ osoittaa riville *i* saavuttaessa aina merkin x_i oikeanpuoleisimman esiintymän, jonka jälkeen *k*-täsmäyksiä ei voi enää esiintyä, voidaan nyt x_i:n esiintymälistasta indeksialueelta $alaraja[i]+1..yläraja[x_i]$ etsiä puolitushaulla suurin sellainen indeksi[33], jossa esiintyvä arvo on pienempi tai yhtä suuri kuin muuttujan *kynnysarvo*, joka kuvaa korkeuskäyrän *k* sijaintia rivillä *i*–1. Tämä indeksi otetaan talteen apumuuttujaan *t*. Tämän jälkeen päivitetään uudeksi kynnysarvoksi *j*. Koska nyt kaikki x_i:n esiintymät väliltä $j..EsLista[x_i, t]$ kuuluvat luokkaan *k*, pitää uudeksi $yläraja[x_i]$:ksi päivittää nykyinen *alaraja*[*i*], sillä seuraavien rivien *k*-täsmäykset sijaitsevat nyt väkisin siitä vasemmalle. Vastaavasti *alaraja*[*i*] päivitetään arvoon *t*+1, koska se on nyt pienin indeksi, josta voi löytyä luokan *k*+1 täsmäys riviltä *i*. Seuraavassa esitetään esimerkki HD1-algoritmin toiminnan havainnollistamiseksi.

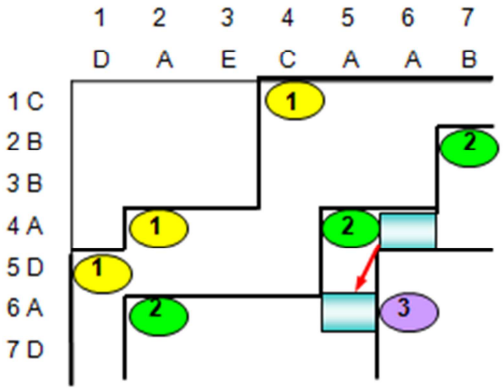
---

[31] Tämä on ilmeistä, sillä dominanttien *k*-täsmäysten sarakeindeksit pienenevät rivi-indeksin kasvaessa. Jos vektorin *Y* indeksi $yläraja[x_i]$ oli jo rivillä *i* luokan *k* kynnysarvoa suurempi eli liian suuri muodostamaan *k*-täsmäystä, ei se enää kelpaa luokkaan *k* millään seuraavistakaan riveistä $u > i$.

[32] Jos rivin ensimmäinen *k*-täsmäys on ei-dominantti, täytyy määritelmän mukaisesti samalla sarakkeella sijaita ylempänä dominantti *k*-täsmäys, jonka muodostaa väistämättä sama merkki.

[33] Jos hakualue on tyhjä, puolitushaku palauttaa indeksin *alaraja*[*i*].

Esimerkki 3.5: X = ”CBBADAD”, Y = ”DAECAAB”, $\sigma = 5$, $p = 3$, PYA = ”CAA”.



Punainen nuoli kuvaa puolitushaun hakualueen rajojen siirtymistä vasemmalle toisella korkeuskäyrällä laskennan edetessä riviltä 4 riville 6.

2-täsmäyksen (4, 5) löydyttyä lähdetään etsimään riviltä 4 ensimmäistä ”A”:n esiintymää, joka voisi kelvata jonkin luokan C_k ($k > 2$) edustajaksi. Koska (4, 5) on ensimmäinen merkin ”A” muodostama 2-täsmäys, haun kohteena ovat ”A”:n esiintymät paikasta 6 lähtien. Hakualue on värjätty **sinertävällä** suorakaiteella. Koska *EsLista*[A, 3] = 6 on viimeinen merkin ”A” esiintymä, jonka Y-indeksi ≤ *kynnysarvo* = 7, päivittyy *alaraja*[4]:si 3 + 1 = 4, eli rivin 4 käsittely voidaan lopettaa, ja *yläraja*[A] saa alarajan vanhan arvon 2. Haettaessa myöhemmin täsmäyksen (6, 2) löydyttyä ”A”:n viimeistä 2-täsmäystä riviltä 6, puristuu uusi hakualue sarakkeeksi 5, sillä se on ainoa ”A”:n esiintymä indeksialueella *alaraja*[6]+1..*kynnysarvo* = 3..5. Merkkien hakualueet eivät milloinkaan leikkaa toisiaan yhden korkeuskäyrän sisällä.

Tietorakenteiden sisältämät arvot 2-täsmäyksiä generoitaessa ennen kunkin rivin käsittelyä ja sen jälkeen:

Kynnysarvojen päivitykset: rivillä 1: kynnysarvo = ∞ (ei päivitystä)
rivillä 2: ∞ ⇒ 7 (löytyi dominantti 2-täsmäys (2, 7))
rivillä 3: edelleen 7 (ei päivitystä)
rivillä 4: 7 ⇒ 5 (löytyi dominantti 2-täsmäys (4, 5))
rivillä 5: edelleen 5 (ei päivitystä)
rivillä 6: 5 ⇒ 2 (löytyi dominantti 2-täsmäys (6, 2))
rivillä 7: edelleen 2 (ei päivitystä)

Alarajan päivitykset: $alaraja_1$ = frekv(”C”) + 1 = 2 (ennallaan, ei uusia täsmäyksiä)
$alaraja_2$: 1 ⇒ 2 = frekv(”B”) + 1 (ei enää uusia 2-täsm.)
$alaraja_3$: 1 ⇒ 2 (päivittyy, koska löytyi merkin ”B” muodostama ei-dominantti täsmäys)
$alaraja_4$: 2 ⇒ 4 (ei voi löytyä enää uusia täsmäyksiä)
$alaraja_5$ = 2 (pysyy ennallaan, ei enää uusia täsmäyksiä)
$alaraja_6$: 1 ⇒ 3 (EsLista[”A”, 3] = 6 > 5)
$alaraja_7$ = 2 (pysyy ennallaan, ei enää uusia täsmäyksiä)

Ylärajan päivitykset: rivillä 1: $yläraja_{'C'}$ = frekv(”C”) = 1 (pysyy ennallaan)
rivillä 2: $yläraja_{'B'}$: 1 ⇒ 1 ((2, EsLista[”B”, 2]) uusi D_2-täsm.)
rivillä 3: $yläraja_{'B'}$: 1 ⇒ 1 (edellisen 2-täsm. muod. ”B”)
rivillä 4: $yläraja_{'A'}$: 3 ⇒ 2 ((4, EsLista[”A”, 2]) uusi D_2-täsm.)
rivillä 5: $yläraja_{'D'}$: 1 ⇒ 1 (ei uutta D_2-täsmäystä)
rivillä 6: $yläraja_{'A'}$: 2 ⇒ 1 ((6, EsLista[”A”, 1]) uusi D_2-täsm.)
rivillä 7: $yläraja_{'D'}$: 1 ⇒ 1 (ei uutta D_2-täsmäystä)

3.1.3.3 Analyysi

HD1-algoritmi löytää PYAlle saman esiintymän kuin HI1. Kyseinen ratkaisu on kuvattu esimerkissä 3.1. Tarkasteltaessa HD1-algoritmin kompleksisuutta havaitaan, että algoritmin ulompaa silmukkaa (merkitty liitteen kohdan 11.1.3 pseudokoodissa nimiöllä *S1*) suoritetaan p+1 kertaa, eli suoritus päättyy, kun havaitaan, ettei tarkasteltavalle korkeuskäyrälle k löydetä yhtään (dominanttia) täsmäystä. Vastaavasti jokaista ulomman silmukan kierrosta kohti tutkitaan sisemmässä silmukassa *S2* vuoron perään kaikki X:n m merkkiä. Pelkkien silmukoiden suorituskustannus on siten $O(pm)$. Pelkästään *S1*:n sisällä suoritettavat muut lauseet kuin silmukka *S2* ovat vakioaikaisia. Samoin *S2*:n sisällä suoritettavat lauseet ovat vakioaikaisia paitsi puolitushaku, jota joudutaan käyttämään tarkalleen silloin, kun tarkasteltavalta riviltä i löydetään dominantti k-täsmäys. Pahimmassa tapauksessa niitä on tarkasteltavassa ongelmassa yhteensä $\frac{1}{2}m(m + 1)$ kappaletta aliluvussa 2.1.3 tehdyn analyysin perusteella, ja tällöin on välttämättä voimassa $p = m$, jolloin myös silmukan *S1* kierrosten määrä maksimoituu. Yksittäisen puolitushaun kustannus on enintään $\log z$+1, missä z on hakualueen pituus. Algoritmin yhdellä ulomman silmukan kierroksella joudutaan pahimmassa tapauksessa tutkimaan puolitushauissa yhteensä n:n mittainen alue, sillä kutakin k:n arvoa kohti hakualueet ovat keskenään erillisiä. Mitä tasaisemmin tämä alue jakautuu riveittäin, sitä enemmän puolitushaku alueen ylitse maksaa Lagrangen kertoimien nojalla [Hil74].

Koska ulompaa silmukkaa suoritetaan m+1 kertaa[34], löytyy yksinkertaistetun analyysini[35] perusteella jokaista k:n arvoa kohti $\frac{1}{2}m$ dominanttia täsmäystä, joiden puolitushakualueiden kokonaispituus on n. Siten yhtä korkeuskäyrää kohti suoritettavien puolitushakujen kustannuksen yläraja olisi $\frac{1}{2}m(1+\log(2n/m)) = O(m \log(n/m)+m)$. Kun tasamittaisiksi oletetut haut toistetaan vielä jokaista ulomman silmukan p kierrosta kohti ja huomioidaan lisäksi alustuksen ja PYAn keräämisen kustannus $O(n)$, saadaan algoritmin aikakompleksisuudeksi $O(n+pm \log(n/m)+pm)$. Täsmälleen samaan ylärajaan päästään Hsun ja Dun artikkelissa. Aikakompleksisuuslausekkeen mukaisesti HD1 toimii HI1:tä tehokkaammin, kun $n >> m$, kun taas yhtä pitkillä syötteillä kertalukueroa ei saavuteta.

Algoritmi tarvitsee syötevektoriensa X ja Y lisäksi muistitilaa esiintymälistoille, vektoreille *alaraja*[1..m] sekä *yläraja*[1..σ] sekä *linkkivektorille*, jossa on p indeksipaikkaa, joista kuhunkin paikkaan $k \in 1..p$ on tallennettu linkitetty lista algoritmin löytämistä dominanteista k-täsmäyksistä. Koska linkkivektoria lukuun

[34] Viimeinen kierros on tosin aina tulokseton.

[35] Teoreettisesti hakualueiden täydellinen tasainen jakautuminen $\frac{1}{2}m(m + 1)$:n dominantin täsmäyksen kesken ei tietystikään ole mahdollista, koska esimerkiksi löydettäessä rivin 1 ainoa dominantti täsmäys on hakualueen pituus *frekv*[x_i], ja myöhemmillä riveillä sama merkki ei enää muodosta dominanttia 1-täsmäystä. Tällä menettelyllä saadaan kuitenkin yläraja puolitushakujen kokonaiskustannukselle.

ottamatta kaikki rakenteet ovat tilakompleksisuudeltaan ≤ $O(n)$, saadaan algoritmin kokonaistilantarpeeksi $O(n+d)$.

Vaikka HD1 toimii asymptoottisesti tehokkaammin kuin sen kehitystyön lähtökohtana ollut HI1, on algoritmiin jäänyt kaikesta huolimatta myös käytännön suoritusaikaa tarpeettomasti hidastava piirre. HD1:n sisempi silmukka aloittaa nimittäin aina toimintansa riviltä 1 riippumatta siitä, mitä korkeuskäyrää ollaan rakentamassa, vaikka *k*. korkeuskäyrän pisteitä ei voi koskaan löytyä riviä *k* ylempää. Etsintää ei välttämättä tarvitsisi aloittaa vielä edes riviltä *k*, vaan järkevä aloituskohta kyseisen korkeuskäyrän tutkimiselle oli ylintä *k*–1 täsmäystä seuraava rivi. Tieto tästä rivistä kirjataan muistiin HI1:ssä muuttujaan *YlinRivi*, mutta HD1:ssä sitä ei ole muistettu rekisteröidä. Mitä pidempi syötteiden PYA on ja mitä kauempana se sijaitsee vektorin *X* alusta lukien, sitä enemmän tehdään turhaa työtä.

3.1.4 Apostolicon ja Guerran I algoritmi (AG1)

3.1.4.1 Uusia tietorakenteita laskennan avuksi

Alberto Apostolico ja *Concettina Guerra* esittelivät vuonna 1987 [Apo87] kaksi PYAn ratkaisevaa algoritmia, joista toinen – tekijöiden *Algoritmi 3:ksi* nimeämä ja tässä työssä lyhenteellä *AG1* tunnistettava – suorittaa laskentaa korkeuskäyrittäin. Algoritmi muistuttaa käsitteellisesti ja toiminnallisesti hyvin paljon kolmea vuotta aikaisemmin julkaistua HD1:tä, vaikka omien sanojensa mukaan tekijät ovatkin lähteneet kehittämään tätä varhaisempaa, jo vuonna 1977 julkaistua HI1-algoritmia.

AG1 täydentää aikaisempia korkeuskäyrittäin PYAn ratkaisevia menetelmiä lähinnä kolmella uutuudella. Ensimmäinen niistä on *esiintymälistojen käänteiskuvaus*. Edellisessä aliluvussa esitellyssä HD1-algoritmissa jouduttiin perustamaan kaksi apuvektoria: *alaraja*[1..*m*] ja *yläraja*[1..σ]. Alarajavektori perustetaan yhtäläisesti myös AG1:ssä, mutta merkkikohtaisen ylärajavektorin tilalle on ilmestynyt *n*+1 -paikkainen vektori *monesko_symboli*. Kyseisen vektorin positioon *j* tallennettu arvo osoittaa, *kuinka mones merkin y_j esiintymistä sijaitsee vektorin Y indeksissä j*. Menetelmässä tarvitaan tätä vektoria, jotta päästäisiin helposti käsiksi haluttuun kohtaan kulloinkin tarkasteltavan merkin x_i esiintymälistassa. Vektori pystytään rakentamaan ajassa $O(n)$ selaamalla pidempi syötevektori *Y* ja merkkien esiintymälistat kertaalleen läpi.

Loput kaksi artikkelissa esitellyistä uusista tietorakenteista – *lähiesiintymätaulukko* ja *lähiesiintymävektori*[36] – tarjoavat uuden vaihtoehdon luokan *k* kaikkien täsmäysten rajaamiseksi kulloinkin tarkasteltavalla rivillä *i*. Tämä on Hsun ja Dun I algoritmin tapaan tarpeen tarkalleen silloin, kun riviltä *i* löydetään dominantti *k*-täsmäys. Tavoitteena on saada vastaus kysymykseen, mikä on muuttujan *kynnysarvo* osoittaman paikan jälkeen vektorin *Y* seuraava indeksi, jossa esiintyy merkki x_i, kun kynnysarvolla

[36] Lähiesiintymätaulukko ja -vektori ehdittiin jo esitellä tarkemmin aliluvuissa 2.4.2 ja 2.4.3.

tarkoitetaan korkeuskäyrän k sijaintia rivillä i–1. Vastaava operaatio suoritettiin edeltäneissä algoritmeissa soveltamalla joko ajassa $O(n)$ toimivaa lineaarihakua (HI1) tai ajassa $O(\log n)$ toimivaa puolitushakua (HI2, HD1) merkin x_i vektorimuotoiseen täsmäyslistaan. Lähiesiintymävektorista haluttu vastaus saadaan asymptoottisesti näitä menetelmiä nopeammin ajassa $O(\log \sigma)$, ja lähiesiintymätaulukosta vastaus saadaan peräti vakioajassa $O(1)$. AG1 asettaa vektorin Y oikeaan reunaan paikkaan n+1 kuvitteellisen *jokerimerkin*, jonka kanssa täsmäävät kaikki Y:stä haettavat symbolit. Näin taataan, että haku lähiesiintymätaulukosta tai -vektorista onnistuu aina. Useimmissa tutkielman testiajoista on AG1-algoritmin hakurakenteeksi valittu toiminnaltaan nopein eli lähiesiintymätaulukko, jonka mukainen vaihtoehto esitetään myös liitteen kohdassa 11.1.4 olevassa pseudokoodissa. Aliluvussa 8.1 tarkastellaan tarkemmin hakua tukevia vaihtoehtoisia tietorakenteita, ja vielä myöhemmin aliluvussa 9.5 esitellään empiirisissä tutkimuksissa saavutettuja mittaustuloksia siitä, miten valittu hakutietorakenne vaikuttaa menetelmän tehokkuuteen.

3.1.4.2 Analyysi

AG1-algoritmi toimii siis hakutietorakenteen perustamista ja dominantin täsmäyksen löytymistä seuraavaa hakua lukuun ottamatta samoin kuin edellisessä aliluvussa esitelty HD1-algoritmi. Myös algoritmin palauttama PYAn instanssi on sama kuin aikaisemmin esitellyissä menetelmissä.

Tarkasteltaessa AG1:n pseudokoodia havaitaan, että enin osa algoritmin käyttämästä ajasta kuluu silmukoiden *S1*, *S2* ja *S3* suorittamiseen. Silmukassa *S1* rakennetaan lähiesiintymätaulukko, jossa jokaista Y:n indeksipaikkaa kohti asetetaan syöttöaakkoston koon eli σ:n verran arvoja. Yksittäisen arvon asettaminen taulukkoon on vakioaikainen operaatio. Silmukan *S1* eli lähiesiintymätaulukon rakentamisen kustannus on siten $O(\sigma n)$. Silmukkaa *S2* suoritetaan p+1 kertaa: kerran kutakin etsittävää täsmäysluokkaa D_k kohti. Kaikki muut siinä suoritettavat lauseet paitsi sisempi silmukka *S3* ovat vakioaikaisia. Silmukkaa *S3* puolestaan suoritetaan enintään m kierrosta yhtä ulomman silmukan *S2* kierrosta kohti tutkimalla, löytyykö riveiltä $k..m$ luokan D_k dominantteja täsmäyksiä. Silmukassa *S3* suoritettavat kaikki lauseet ovat vakioaikaisia, jos hakurakenteena käytetään lähiesiintymätaulukkoa. Siten koko algoritmin aikakompleksisuus on suuruusluokkaa $O(Max\{pm, \sigma n\})$, joista jälkimmäinen kuvaa lähiesiintymätaulukon perustamiskustannusta. Jos lähiesiintymätaulukon asemesta käytettäisiin $O(n)$:n mittaista lähiesiintymävektoria, jokainen hakuoperaatio vaatisi ajan $O(\log \sigma)$. Silloin aikakompleksisuutta kuvaisi lauseke $O(n + pm \log \sigma)$.

Algoritmissa varataan syötemerkkijonojen lisäksi muistia esiintymälistoille ja sen käänteiskuvaukselle, linkkivektorille, johon dominantit täsmäykset tallennetaan, sekä hakutietorakenteelle. Linkkivektorin muistintarve on suuruusluokkaa $O(d)$. Muut

tarvittavat tietorakenteet vaativat muistia $O(n)$, paitsi lähiesiintymätaulukko, jonka koko on $O(\sigma n)$. AG1:n tilakompleksisuus on siten $O(d + \sigma n)$ käytettäessä lähiesiintymätaulukkoa ja $O(d + n)$ turvauduttaessa lähiesiintymävektoriin. Näistä rajoista voi päätellä, että algoritmin muistintarve kuten samalla myös esiprosessointivaiheen suoritusaika kasvavat nopeasti, kun syöttöaakkoston koon annetaan kasvaa, ellei lähiesiintymätaulukkoa korvata lähiesiintymävektorilla. Siten olisi ennustettavissa, että suurella syöttöaakkostolla lähiesiintymätaulukon mahdollistaman nopean haun hyödyt rapautuisivat hankaloituneen esiprosessoinnin myötä.

3.1.5 Apostolicon ja Guerran lineaaritilainen algoritmi (AGL)

3.1.5.1 Ensimmäinen korkeuskäyrittäinen lineaaritilainen PYA-algoritmi

A. Apostolicon ja C. Guerran korkeuskäyrittäin prosessoivasta lineaaritilaisesta algoritmista [Apo85] käytetään tässä työssä lyhennettä *AGL*. Varhaisimmille lineaaritilaisille PYAn ratkaiseville menetelmille on ominaista, että ne muistuttavat paljolti jotain aikaisemmin kehitettyä ei-lineaarista algoritmia [Hir75][Kum87][Apo92]. Myöskään AGL ei tee poikkeusta tässä suhteessa, vaan se on hyvin voimakkaasti sidoksissa samojen tekijöiden korkeuskäyrittäin prosessoivaan algoritmiin AG1 [Apo87][37], joka on puolestaan tehostettu muunnos *Hirschbergin I* algoritmista [Hir77].

Apostolicon ja Guerran lineaaritilainen algoritmi puolittaa jokaisella rekursiotasolla vektorin *X* pituuden aivan samoin kuin Hirschbergin lineaaritilainen algoritmi (*HIL*)[38]. Kullakin kutsukerralla tarkasteltavat alueet ovat identtiset myöhemmin esiteltävän HIL:n valitsemien kanssa: mikäli alue koostuu yhteensä *i* rivistä ja *j* sarakkeesta, ratkaistaan ensiksi alueen *yläpuoliskon* eli sen $\lfloor i/2 \rfloor$ ensimmäisen rivin ja kaikkien *j* sarakkeen välinen PYA, ja tämän jälkeen alueen *alapuoliskon* eli $\lceil i/2 \rceil$ alimman rivin ja samaisen sarakealueen välinen PYA. Alapuoliskon PYAa määrättäessä käsitellään kumpaakin alueen syötejonoa *käännetyssä järjestyksessä*. Osaratkaisut pyritään tämän jälkeen yhdistämään sellaisen sarakkeen *k* kohdalta, joka *maksimoi eri puoliskojen PYAn pituuksien summan* siten, että toisiinsa yhdistetyt osaratkaisut kelpaavat koko tarkasteltavan alueen PYAn ratkaisuksi. Sarakkeen *k* valitsemismenettelyyn palataan tarkemmin hetken päästä.

Sen sijaan ositteiden PYAn pituuksien määrääminen tapahtuu samoin kuin AG1-algoritmissa kolmea poikkeusta lukuun ottamatta. Ensimmäisenä eroavaisuutena on, ettei kaikkia tutkittavasta ositteesta löytyviä dominanttitäsmäyksiä kirjata muistiin, sillä

[37] Lineaaritilaisen menetelmän julkistamisvuosi on kuitenkin kaikesta huolimatta varhaisempi kuin samojen tekijöiden samankaltaisen ei-lineaarisen algoritmin. Tämä selittyy sillä, että ei-lineaariset menetelmät esittelevän laajan teknisen raportin tarkastusvaihe lehdessä kesti yli kaksi vuotta.

[38] Hirschbergin lineaaritilainen algoritmi erityispiirteineen esitellään myöhemmin riveittäin prosessoivia menetelmiä kuvaavan aliluvun kohdassa 3.2.1. Kyseinen algoritmi ehdittiin julkaista kymmenen vuotta ennen AGL:n esittelyä. Mikäli HIL:n toimintatapa ei ole lukijalle entuudestaan tiedossa, tulee AGL:n toiminnan ymmärtäminen jossain määrin helpommaksi tutustumalla ensiksi HIL-algoritmiin.

niiden lukumäärää ei voida rajoittaa ylhäältä termillä $O(n + m)$. Jokaista ositteesta löytyvää korkeuskäyrää kohti kirjataan ainoastaan sen *vasemmanpuoleisimman edustajan sijaintisarake* ositteessa. Tämän voidaan tulkita vastaavan ositteen viimeisen rivin kynnysarvojen kirjaamista, mikäli prosessointi tapahtuisi riveittäin. Toiseksi, lähiesiintymätaulukon[39] käyttäminen *Y*-vektorin merkkien hakemiseen ei enää tule kyseeseen, sillä taulukon konstruoimiseksi tarvittava muistitila ei ole lineaarinen syötteiden pituuksien suhteen vaan suuruusluokkaa $O(\sigma n)$. Hakurakenteeksi valitaan siten *lähiesiintymävektori*[40], jota täydentämään tarvitaan syöttöaakkoston merkkien esiintymälistat sekä niiden käänteiskuvaus, jotka kaikki vaativat lineaarisen muistitilan. Kolmas eroavuus liittyy jälkimmäisen ositteen käsittelyyn. Sitä on prosessoitava *lopusta alkuun päin*, mikä tuo uudelleen mieleen Hirschbergin lineaaritilaisen algoritmin laskentatavan. AGL tuntuu siten edellä kuvattujen ominaisuuksiensa perusteella selvästi HIL:n ja AG1:n hybridiltä: dominanttitäsmäyksien etsintä on perua AG1:stä ja rekursiivinen ositustekniikka puolestaan HIL:stä.

Kun kummankin ositteen PYAn pituudet $p_{alkuosa}$ ja $p_{loppuosa}$ sekä osittеisiin liittyvät minimaaliset kynnysarvot $KA_{alkuosa}$ ja $KA_{loppuosa}$ on selvitetty, saadaan selville koko ongelman PYAn pituus etsimällä sarake *k*, joka toteuttaa ehdon

$$k = KA_{alkuosa}[i] \mid \{Max\ (i + j) \mid KA_{alkuosa}[i] < KA_{loppuosa}[j],\ 0 \le i \le p_{alkuosa},\ 0 \le j \le p_{loppuosa}\}.$$

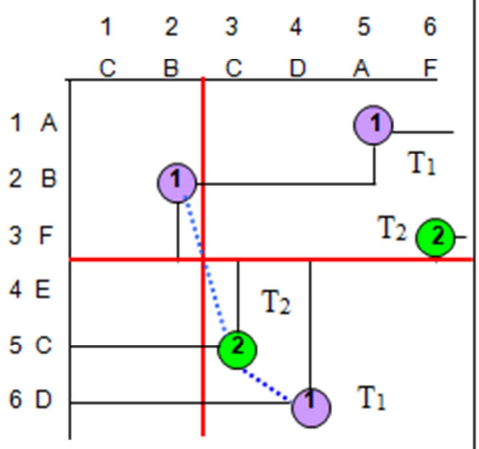
Edellisessä ehdossa oletetaan, että loppuosan käsittelyssä merkkien alkuperäiset indeksit säilyvät kynnysarvoja tallennettaessa. Mikäli ehdon täyttäviä indeksejä *i* on useita, voidaan sopia, että niistä valitaan vasemmanpuoleisin. Rivikohtainen leikkauspiste määräytyy siten täysin samoin kuin aliluvussa 3.2.1 esiteltävässä HIL-algoritmissa. Prosessointi AGL:ssä jatkuu tästä eteenpäin rekursiivisesti saman kaavan mukaan kuin HIL:ssä, ja yhden ositteen sisällä tarkastelu etenee kuten AG1:ssä. Koska algoritmi valitsee täysin saman PYAn instanssin kuin HIL, erillistä esimerkkiä 17 x 17 -kokoiselle aineistolle ei tässä esitetä. Sen sijaan näytetään pienempi esimerkki, josta käy ilmi, miten ongelman osittaminen aliongelmiksi tapahtuu.

---

[39] *Lähiesiintymätaulukosta* (kts. aliluku 2.4.2) saadaan vakioajassa selville minkä tahansa syöttöaakkostoon kuuluvan merkin ensimmäinen sijaintipaikka syötemerkkijonossa *Y i*:nnen position jälkeen. Ellei merkkiä enää esiinny indeksipaikan *i* jälkeen, määritellään sijaintipaikaksi $n + 1$.

[40] *Lähiesiintymävektori* (kts. aliluku 2.4.3) on tiivistetyssä muodossa oleva lähiesiintymätaulukko, jonka *i*. positioon on tallennettuna tieto siitä, mistä löytyy merkin s_g, missä $g = (i\ MOD\ \sigma) + 1$, seuraava esiintymä syötevektorista paikan *i* jälkeen.

Esimerkki 3.6: $X[1..6]$ = ”ABFECD”, $Y[1..6]$ = ”CBCDAF”, $p = 3$, PYA = ”BCD”.



Esimerkin ylemmästä ositteesta löydetään kahden mittainen PYA. Ositteen kynnysarvoiksi D_k-luokille 0..2 määräytyvät minimisarakkeet, joista löytyy D_k-täsmäys. Tällöin $KA_{alkuosa}[0..2]$ saa arvot 0, 2 ja 6. Myös jälkimmäisestä ositteesta löytyy kahden mittainen PYA. Siellä $KA_{loppuosa}[0..2]$ saa arvot 7, 4 ja 3. Nyt $KA_{alkuosa}[1] = 2 < KA_{loppuosa}[2] = 3$. $1 + 2 = 3$ on suurin KA-vektoreiden indeksipareista saatava summa $i + j$, joka toteuttaa ehdon $KA_{alkuosa}[i] < KA_{loppuosa}[j]$, joten osituspisteeksi valitaan $(\lfloor m/2 \rfloor, KA_{alkuosa}[1]) = (3, 2)$. Alkuperäisen ongelman PYA-jono on merkitty kuvaan sinisellä katkoviivalla ja osituspisteen lohkomat alueet punaisilla lihavoiduilla viivoilla.

3.1.5.2 Analyysi

Koska AGL sivuaa toteutukseltaan voimakkaasti algoritmeja AG1 ja HIL edellä kuvailluin poikkeuksin, esitetään AGL:stä liitteen kohdassa 11.1.5 ainoastaan karkean tason pseudokoodilistaus. Apostolicon ja Guerran lineaaritilainen algoritmi selvittää ensimmäisellä kutsukerralla alkuperäisen ongelman PYAn pituuden käyttämällä AG1-algoritmia, josta ei-lineaaritilaisuuden aiheuttavat *lähiesiintymätaulukko* sekä dominanttien täsmäysten kirjaamiseen käytettävä *linkkivektori* on poistettu. Tähän vaiheeseen kuluu aikaa $O(n + pm \log \sigma)$ eli suuruusluokaltaan saman verran kuin AG1:n suorittamiseen lähiesiintymävektoritoteutusta käyttäen. Siirryttäessä rekursiossa tasolta k tasolle $k + 1$ ratkaistavan ongelman koko puolittuu. Tällöin aikakompleksisuuslauseke voidaan kehittää auki geometriseksi summaksi $(n + pm \log \sigma) + (n + pm \log \sigma)/2 + (n + pm \log \sigma)/4 + \ldots$. Rekursiotasoja kertyy puolestaan $\log m + 1$ kappaletta. Ratkaisemalla sarjan summa havaitaan, että työn määrä likimain kaksinkertaistuu verrattuna ei-lineaaritilaiseen ratkaisemiseen. Lisäksi rekursiivisista kutsuista muodostuu suuruusluokkaa $m \log m$ oleva vähimmäiskustannus. Siten AGL-algoritmin suoritusajaksi saadaan $O(m \log m + n + pm \log \sigma)$.

Syötevektoreiden X ja Y sekä hakua varten tarvittavien aputietorakenteiden varastointiin käytetään tilaa $O(n)$. Koska AGL ei pidä kirjaa dominanttitäsmäysten lukumäärästä ositteissa vaan hävittää korkeuskäyrien sisältämän informaation muilta osin kuin minimaalisten kynnysarvojen osalta, laskentavaihe voidaan suorittaa lineaarisessa tilassa syötteiden suhteen. Koska rekursiopinoonkaan ei tarvitse säilöä muuta tietoa kuin itse PYA-jono, algoritmi suoriutuu kokonaisuudessaan lineaarisessa muistitilassa, kuten oli tavoitteena.

AGL-algoritmin käytännön suoritusaikaa olisi mahdollista tehostaa. Puolitettaessa syötteen *X* pituutta on kynnysarvovektorissa tallella tieto PYAn pituudesta *p* seuraavaksi muodostuvissa kahdessa ositteessa. Jos ositus tehdään siten, että ainakin jommassakummassa $p = 0$, ei kyseistä ositetta kannata tutkia. Tällöin vähintään toinen algoritmin lopussa esiintyvistä rekursiivisista kutsuista voidaan jättää suorittamatta. Mitä suurikokoisempi nollan mittaisen PYAn sisältävä osite löydetään, sitä enemmän pystytään tällä tekniikalla välttämään tuloksetonta laskentaa.

3.1.6 Chinin ja Poonin algoritmi (CPO)

3.1.6.1 Lisää tehoa riviltä toiselle siirtymiseen: symbolijärjestystaulukko

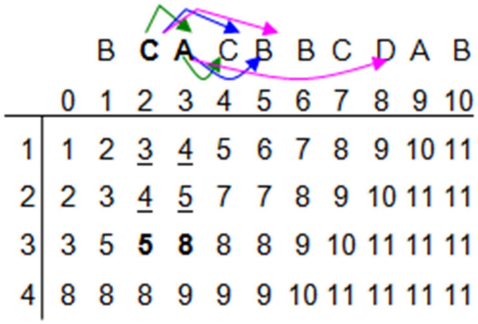
Viimeisenä korkeuskäyrä kerrallaan laskentaa suorittavana PYA-algoritmina esitellään vuonna 1990 julkaistu *Chinin* ja *Poonin* algoritmi [Chi90]. Algoritmi tunnetaan alkuperäisartikkelissa nimellä *FIND_LCS*, ja siitä käytetään tässä työssä lyhennettä *CPO*. Menetelmän perusajatus on sama kuin sen edeltäjilläkin, eli etsitään järjestyksessä kunkin korkeuskäyrän dominantit täsmäykset riveittäin ylhäältä alaspäin. Myös CPO käyttää laskennassa hyväksi pseudotäsmäystä (0, 0). Sen sijaan täsmäysten etsintätekniikka on aikaisempia menetelmiä jalostetumpi. Edellä esitelty AG1-algoritmi käyttää hyväksi dominantin täsmäyksen löydyttyä vektoriin *Y* perustuvaa lähiesiintymätaulukkoa. Sen avulla pystyttiin hyppäämään suoraan löytynyttä dominanttia *k*-täsmäystä seuranneiden, mahdollisesti useiden ei-dominanttien täsmäysten ylitse. Lähiesiintymätaulukkoa käytetään myös Chinin ja Poonin algoritmissa, mutta *kahteen eri tarkoitukseen*. CPO:ssa lähiesiintymätaulukon avulla ei etsitä pelkästään (AG1:n tavoin) symbolin x_i lähintä sijaintipaikkaa vektorissa *Y* annetusta indeksistä *j* lähtien, vaan tämän lisäksi sitä tarvitaan myös määrättäessä rivejä, joilta kannattaa hakea *ehdokkaita korkeuskäyrän k dominanteiksi täsmäyksiksi*. Tämän perään vielä tarkastetaan, täyttääkö löydetty täsmäys sille asetetut kriteerit.

Algoritmi AG1 käy yhden korkeuskäyrän sisällä systemaattisesti läpi kaikki rivit väliltä *k..m* ja testaa, voidaanko kulloinkin tarkasteltavana olevalta riviltä löytää luokkaan *k* kuuluvia täsmäyksiä. Sekä ei-dominantin että dominantin täsmäyksen löytyminen aiheuttaa algoritmissa kirjanpitoa: alarajavektorin arvoja joudutaan aina päivittämään. CPO on kuitenkin algoritmeista ensimmäinen, jossa voidaan hypätä jo AG1:ssä ohitettavien ei-mielenkiintoisten sarakkeiden lisäksi myös *tarpeettomien rivien ylitse*. Tämä ominaisuus on saatu aikaan esittelemällä uusi, lähiesiintymätaulukon kaltainen tietorakenne, josta käytetään nimitystä *symbolijärjestystaulukko*[41]. Sen rivi-indekseinä ovat syöttöaakkoston symbolien järjestysnumerot $1..\sigma$ ja sarakeindekseinä

[41] Tekijät käyttävät kyseisestä tietorakenteesta heikohkosti valaisevaa nimeä *α-CLOSEST'*.

vektorin X indeksipaikat laajennettuina indeksillä 0. Paikassa *SymbJTaulu*[*j*, *i*] oleva arvo kertoo, *miten monta riviä on edettävä alaspäin riviltä i, jotta kohdattaisiin järjestyksessä j. erilainen X:n symboli* x_i*:n jälkeen*[42]. Seuraava esimerkki havainnollistaa, mistä symbolijärjestystaulukossa on kyse.

Esimerkki 3.7: Σ={ ”A”,”B”,”C”,”D” }, σ = 4, X =”BCACBBCDAB”, *pituus*(X) = 10, *X-vektorin symbolijärjestystaulukko näyttäisi seuraavanlaiselta.* ***Vihreä*** *nuoli osoittaa tarkasteltavasta sarakkeesta 2 tai 3 lähtien ensimmäisen kohdatun merkin,* ***sininen*** *nuoli toisen erillisen merkin ja* ***vaaleanpunainen*** *nuoli kolmannen erillisen merkin sijaintisarakkeen.*



Tarkasteltakoon vaikkapa lihavoituja arvoja taulukon soluissa [3, 2] ja [3, 3]. Soluun [3, 2] tallennettu arvo kertoo, mihin X:n positioon asti on edettävä, jotta kohdattaisiin kolme toisistaan eroavaa syötevektorin X merkkiä x_2:n jälkeen. Ensimmäinen merkki löydetään tietystikin aina heti seuraavasta indeksipaikasta, eli se olisi tässä tapauksessa X[3]:sta löytyvä ”A”. Koska X[4]:stä löytyy ”C” ja x_5 = ”B”, se on nyt kolmas erilainen löydetty merkki x_2:n jälkeen, joten paikkaan *SymbJTaulu*[3, 2] tallentuu arvo 5. Vastaavalla periaatteella paikkaan [1, 2] tallentuu kolmonen ja paikkaan [2, 2] nelonen. Arvo 8 solussa [3, 3] osoittaa puolestaan kolmannen erillisen X:n merkin sijainnin x_3:n jälkeen. Merkeistä ensimmäinen, tällä kertaa ”C”, löytyy taas tietysti heti seuraavalta riviltä 4 ja toinen eli ”B” heti tämän perään riviltä 5. Sen sijaan riviltä 6 löytyy uudestaan ”B”, ja rivillä 7 törmätään toistamiseen merkkiin ”C”. Kolmas erilainen merkki kohdataankin siten vasta rivillä 8, josta löytyy ”D”. Täten asetetaan *SymbJTaulu*[3, 3] = *8*. Esimerkissä 3.7 merkkien yläpuolisten, murtoviivoilla merkittyjen nuolten lähtörivinä X:ssä on 2 ja alapuolisten, kaariviivoin piirrettyjen nuolten lähtörivinä 3. Ensimmäinen kohdattu merkki kummastakin aloituspisteestä lukien on merkitty **vihreällä**, toinen **sinisellä** ja kolmas **vaaleanpunaisella** nuolella.

Arvo m+1 jossakin symbolijärjestystaulukon solussa [j, i] tarkoittaa, ettei rivin i jälkeen enää kohdata vektorin X loppuosassa vähintään j:tä erilaista merkkiä, vaan X ehtii loppua kesken. Taulukon rakentaminen tapahtuu esiprosessointivaiheessa. Sen viimeinen rivi alustetaan mielekkäästi pseudoarvolla m+1. Muut rivit täytetään silmukassa, jonka laskuri i pienenee m–1:stä kohti nollaa. Kaikkien näiden rivien täyttö

[42] Tässä yhteydessä olisi oikeaoppisempaa puhua sarakkeesta i, mutta koska i tarkoittaa sijaintia vektorissa X, jonka merkkejä edustavat tutkielman algoritmien kuvauksissa aina rivit, pitäydytään tässä tulkinnassa.

noudattelee seuraavaa systematiikkaa. Ensiksikin, koska ensimmäinen merkki kulloinkin tarkasteltavan rivin i jälkeen[43] kohdataan jo heti seuraavalla rivillä, asetetaan *SymbJTaulu*[1, i]:n arvoksi i+1 kaikilla i:n arvoilla. Tämän jälkeen pitää etsiä, kuinka monentena seuraavan rivin merkki x_{i+1} löydettäisiin, jos etsintä aloittaisiinkin vasta riviltä i+2. Oletetaan, että se on järjestyksessä u. erilainen merkki ($1 \le u \le \sigma$) rivin i+1 jälkeen. Tällöin tiedetään, että rivin i+1 jälkeen numerojärjestyksessä 1, 2, ..., u–1 kohdatut syöttöaakkoston eri merkit kohdataankin nyt järjestyksessä 2, 3, …, u, kun etsintä aloitetaankin jo riviltä i+1. Täten voidaan kopioida *SymbJTaulu*[z, i]:lle arvot taulukon indeksipaikasta *SymbJTaulu*[z–1, i+1] jokaiselle z:lle väliltä 2..u.[44] Sen sijaan ne merkit, jotka esiintyivät riviltä i+2 lähtien järjestyksessä vasta u+1:nä tai myöhemmin, säilyttävät rivin i+1 jälkeisen löytymisjärjestyksensä: niiden lähimmät esiintymät ovat jo riviltä i+2 lähtien sijainneet kauempana kuin merkin x_{i+1} seuraava esiintymä, eli *SymbJTaulu*[z, i] saa saman arvon kuin *SymbJTaulu*[z, i+1] jokaisella z:n arvolla väliltä u+1..σ.[45]

Vertaillaan vielä asian selventämiseksi kolmatta ja neljättä saraketta esimerkin 3.7 taulukossa. Merkin x_4 = ”C” jälkeen kohdataan ensimmäinen ”B” paikassa 5, ensimmäinen ”C” paikassa 7, ensimmäinen ”D” paikassa 8 ja ensimmäinen ”A” paikassa 9, joten taulukon neljäs sarake sisältää arvot 5, 7, 8 ja 9. Sarakkeen 4 valmistuttua asetetaan sarakkeen 3 ylimpään positioon arvo 4, koska *SymbJTaulu*[1, i] saa aina arvon i+1. Nyt pitää tutkia, mistä vektorin X indeksistä löytyy seuraava ”C” paikan 4 jälkeen, ja käy ilmi, että seuraava ”C”:n esiintymä on paikassa 7, ja ”C” on järjestyksessä toinen erilainen merkki indeksistä 5 lähtien. Siten kopioidaan *SymbJTaulu*[2, 3]:een arvo 5 vasemmalle alaviistoon paikasta *SymbJTaulu*[1, 4]. Sen sijaan arvot paikkoihin *SymbJTaulu*[3, 3] ja *SymbJTaulu*[4, 3] saadaan kopioimalla ne suoraan vasemmalle paikoista *SymbJTaulu*[3, 4] ja *SymbJTaulu*[4, 4]: merkit ”A” ja ”D” ovat järjestyksessä yhtä monennet havaitut merkit, aloitettiin niiden etsintä jo indeksistä 4 tahi vasta indeksistä 5 lähtien.

3.1.6.2 Täsmäysten vaikutusalueet

Chinin ja Poonin algoritmin lähtökohtana on havainto, ettei yksittäistä luokan k dominanttia täsmäystä (i, j) voi seurata useampi kuin yksi saman symbolin s_t ($1 \le t \le \sigma$) muodostama luokan k+1 dominantti täsmäys. Tämä ominaisuus on dominantin täsmäyksen määritelmän ilmeinen seuraus. Jokaiselle k-dominantille täsmäykselle muodostuu nk. *vaikutusalue*, jolle rajoitetaan sitä mahdollisesti seuraavien dominanttien k+1 -täsmäysten etsintä. Vaikutusalueen ylärajan muodostaa rivi i+1 ja

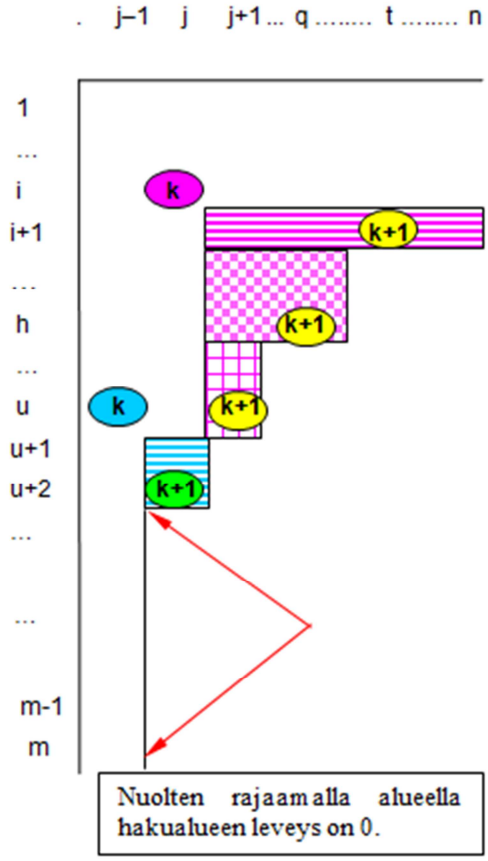
[43] Kannattaa huomioida, ettei tarkasteltavalla merkillä x_i itsellään ole mitään vaikutusta taulukon loppupään arvoihin, vaan ensimmäinen vaikuttava merkki on vasta x_{i+1}.
[44] Jos merkit x_{i+1} ja x_{i+2} ovat samat, mainittua kopiointia ei tehdä.
[45] Jos x_{i+1} oli viimeinen erilainen symboli riviltä i+2 lähtien, kopiointia ei suoriteta.

vasemman rajan sarake j+1. Sen alarajan muodostaa (i, j):tä lähinnä seuraavan alemman k-dominantin täsmäyksen (u, v) rivinumero u[46] tai – jos (i, j) on alin dominantti k-täsmäys – rivi m. Vaikutusalueen oikeana rajana toimii q–1, missä q on riveiltä 1..i löydetyn alimman dominantin k+1 -täsmäyksen sarakeindeksi. Ellei yhtään dominanttia k+1 -täsmäystä ole vielä ehditty löytää, oikeana rajana toimii viimeinen sarake n.

Dominantin k-täsmäyksen (i, j) vaikutusalueeksi muodostuu siten suorakaiteen muotoinen alue [i+1..u, j+1..q–1]. Tältä alueelta pitää tutkia rivejä symbolijärjestys-taulukon sarakkeeseen i tallennettujen arvojen mukaisessa järjestyksessä, kunnes jollain $z \in [1..\sigma]$ *SymbJTaulu*[z, i] > u, tai kaikkien syöttöaakkostoon kuuluvien merkkien ensimmäiset esiintymät rivin i jälkeen on jo ehditty tutkia. Sitä mukaa kun uusia dominantteja k+1 -täsmäyksiä löydetään, niistä asetetaan edeltäjälinkki täsmäyksen generoineeseen isäsolmuun (i, j). Samalla siirtyy vaikutusalueen oikean rajan määräävä sarakeindeksi q uuden dominantin k+1 -täsmäyksen vasemmalle puolelle entisestä paikastaan. Rivin u jälkeen vaikutusalueen ala- ja vasen raja määrätään aina uudelleen. Mielivaltaisen dominantin k-täsmäyksen vaikutusalueelta tutkitaan CPO-algoritmissa siten enintään $Min\{\sigma, u-i\}$ riviä. Tästä voi intuitiivisesti päätellä, että mitä pidempi syötevektori X on ja mitä suppeampi syöttöaakkoston koko on, sitä useampi rivi pystytään jättämään tarkastelematta etsittäessä korkeuskäyrälle k kuuluvia dominantteja täsmäyksiä. Esimerkiksi jos $\sigma = 2$, joudutaan k:n arvolla 1 tutkimaan vain kaksi X:n merkkiä ja k:n arvolla 2 enintään neljä merkkiä. Seuraava esimerkki valaissee vaikutusalueen käsitettä CPO-algoritmissa.

[46] Sama formaalisti: $\exists(i, j), (u, v) \in D_k \mid (i < u) \wedge (j > v) \wedge \neg\exists(s, t) \in D_k \mid (i < s < u), (j > t > v)$.

Esimerkki 3.8: *Dominanttien k-täsmäysten vaikutusalueet.*



Dominantin k-täsmäyksen (i,j) vaikutusaluetta on havainnollistettu vaaleanpunaisilla suorakulmioilla, jotka ulottuvat riviltä i+1 aina riville u asti. Näiltä riveiltä löytyy täsmäykselle (i, j) kolme k+1-dominanttia seuraajaa, jotka sijaitsevat positioissa $(i$+1, $t)$, (h, q) ja $(u, j$+1), i+1 < h < u ja $t > q > j$+1. Kyseiset k+1-seuraajat on värjätty kuvassa keltaisiksi. Vaikutusalueen oikea reuna siirtyy aina uuden löytyneen dominantin k+1 -täsmäyksen vasemmalle puolelle seuraavalla rivillä, mikä näkyy suorakulmioiden kutistumisena oikealta. Seuraavan dominantin k-täsmäyksen vaikutusaluetta symbolisoi vaaleansininen suorakulmio. Koska $(u, j$–1) on kuvan alin dominantti k-täsmäys, sen vaikutusalue jatkuu viimeiselle riville asti, mutta rivin u+2 jälkeen alue puristuu leveyssuunnassa kokoon, kun sarakkeesta j löytyy dominantti k+1 -täsmäys. Sekä vaaleanpunaisilta että -sinisiltä alueilta etsitään kutakin aakkoston symbolia vain sen ylimmältä sijaintiriviltään alueella, ja tältä riviltä valitaan merkin vasemmanpuoleisin esiintymä ko. alueella, mikäli sellainen on olemassa.

3.1.6.3 Aika- ja tilavaativuudesta

Esiprosessointivaiheessa perustetaan silmukoissa *S1* ja *S2* sekä lähiesiintymä- että symbolijärjestystaulukot, joista kummankin alustaminen vaatii työmäärän $O(\sigma n)$. Silmukkaa *S3* suoritetaan $p + 1$ kertaa eli yksi kierros enemmän kuin PYAlla on pituutta. Silmukan *S3* sisällä kaikki muut lauseet ovat suoritusajaltaan vakioaikaisia paitsi sisempi silmukka *S4*. Siinä käydään läpi vuoron perään kukin D_k-täsmäys ja etsitään niille seuraajia korkeuskäyrältä k+1. Silmukan *S4* sisältämät lauseet ovat samoin vakioaikaisia paitsi sen sisällä oleva silmukka *S5*, jossa yhtä D_k-täsmäystä kohti tutkitaan sen vaikutusalueelta enintään σ riviä. Silmukoiden *S3* – *S5* kokonaiskustannus voidaan siten lausua dominanttien täsmäysten määrään perustuen termillä $O(\sigma d)$. Algoritmin pseudokoodilistaus löytyy liitteen kohdasta 11.1.6.

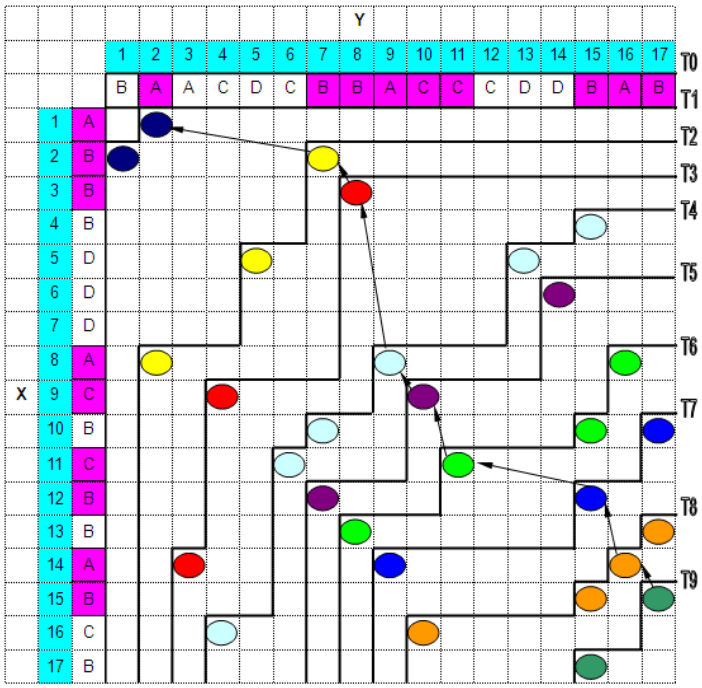
Tarkempi analyysi osoittaa, että pelkästään $O(\sigma d)$ olisi silmukoiden *S3* – *S5* kokonaiskustannukseksi tarpeettoman väljä. Silmukoissa ei nimittäin tarvitse myöskään koskaan tehdä enempää työtä kuin $O(pm)$, sillä silmukat *S4* ja *S5* käyvät yhteensä enintään m riviä jokaista korkeuskäyrää kohti. Siten silmukoiden *S3* – *S5* kokonaiskustannus on $O(Min\{pm, \sigma d\})$.

Algoritmin loppuprosessoinnissa kerätään PYA lopusta alkuun päin etsimällä korkeuskäyrän p ylin täsmäys ja peruuttamalla sieltä edeltäjälinkkejä pitkin kohti jonon alkua. Algoritmi löytää useista PYAn instansseista aina ylimmän. PYAn keräämisvaiheen kustannus on siten $O(p)$, ja esiprosessoinninkin huomioiva algoritmin kokonaisaikakompleksisuus on siten $O(n\sigma + Min\{pm, \sigma d\})$. CPO:ssa varataan muistia syötevektoreiden lisäksi lähiesiintymä- ja symbolijärjestystaulukoille sekä linkkivektorille, johon kerätään kaikki algoritmin löytämät dominantit täsmäykset. Menetelmän tilakompleksisuus on siten $O(n\sigma + d)$.

CPO:ssa käytettävät lähiesiintymä- ja symbolijärjestystaulukot takaavat nopeat hakuoperaatiot, mutta vastaavasti menetelmän esiprosessoinnista tulee kallista, kun syötevektorit ovat pitkiä ja syöttöaakkoston koko suuri. Tekijät siten ymmärrettävästi suosittelevat algoritmiaan käytettäväksi lähinnä silloin, kun syöttöaakkosto on pieni, jolloin esiprosessointivaihe ei ehdi kuluttaa tehostuneen haun mukanaan tuomia hyötyjä.

Algoritmin suoritusaikaa olisi mahdollista lyhentää tiukentamalla silmukan *S5* alkuehtoa vielä lisävaatimuksella *MaxSarake* > *MinSarake*. Jos nimittäin tarkasteltavan *k*-täsmäyksen vaikutusalue on jo ehtinyt puristua leveyssuunnassa tyhjäksi kuten esimerkissä 3.8 rivin *u*+2 jälkeen, ei kyseinen täsmäys pysty enää generoimaan itselleen seuraajia korkeuskäyrällä *k*+1. Silmukkaehdon tiukennuksesta voisi olettaa olevan hyötyä, jos σ on kooltaan iso. Lopuksi esitetään vielä, miten CPO löytää ratkaisun esimerkin 3.1 mukaisille syötemerkkijonoille.

Esimerkki 3.9: *CPO-algoritmin löytämä ratkaisu esimerkin 3.1 syötteille.*



3.2 Riveittäin tapahtuva ratkaiseminen

Edellä tutustuttiin pisimmän yhteisen alijonon ongelman ratkaisemiseen *korkeuskäyrä kerrallaan*. Tätäkin vallitsevampi lähestymistapa PYA-algoritmeissa on kuitenkin prosessointi *riveittäin*. Korkeuskäyrittäin tapahtuneessa käsittelyssä ei kertaalleen löydettyjä korkeuskäyrien pisteitä enää tutkittu uudelleen, mutta sen sijaan kummankin syötevektorin merkkejä jouduttiin selaamaan toistuvasti: yleensä $p + 1$ kertaa. Riveittäisessä käsittelyssä ajatuksena on puolestaan se, että kutakin syötevektorin X merkkiä x_i tarkastellaan ainoastaan kertaalleen. Kaikki sen muodostamat dominantit täsmäykset etsitään ja mahdollisesti kirjataan ennen siirtymistä X:ssä eteenpäin. Sen sijaan vektoria Y joudutaan näissäkin menetelmissä selaamaan läpi toistuvasti, ja saman korkeuskäyrän pisteitä haetaan useilta eri riveiltä.

Tutustuessamme PYAn ratkaisemiseen dynaamisella ohjelmoinnilla aliluvussa 2.2 tarkastelimme *Wagnerin* ja *Fischerin* syötejonojen kaikki indeksiparit tutkivaa WFI-algoritmia. Itse asiassa olisi oikeutetumpaa esitellä kyseinen algoritmi vasta käsillä olevan aliluvun alkajaisiksi, sillä menetelmä suorittaa laskentaa rivi kerrallaan. Koska tätä algoritmia tarvittiin kuitenkin havainnollistamaan PYA-ongelman peruskäsitteitä, lähdetään tässä aliluvussa liikkeelle kyseisen menetelmän lineaaritilaisesta versiosta, *Hirschbergin lineaaritilaisesta* algoritmista.

3.2.1 Hirschbergin lineaaritilainen algoritmi (HIL)

3.2.1.1 Lineaaritilainen variantti Wagnerin ja Fischerin algoritmista

Hirschberg [Hir75] julkaisi ensimmäisenä vuonna 1975 *lineaarisessa muistitilassa toimivan PYA-algoritmin.* Algoritmi on nimetty alkuperäisartikkelissa nimellä *ALG C,* ja tässä työssä siitä käytetään lyhennettä *HIL*. Sen esittelemisen aikoihin ainoa yleisesti tunnettu menetelmä pisimmän yhteisen alijonon ongelman ratkaisemiseksi oli dynaamiseen ohjelmointiin perustuva Wagnerin ja Fischerin algoritmi [Wag74]. Luotettavasti toimivan WFI-algoritmin heikkoina kohtina ovat suuruusluokkaa $\Omega(mn)$ olevat aika- ja tilakompleksisuudet. Tietokoneiden työmuistiresurssien ollessa niukkoja ja kalliita oli etenkin tilakompleksisuuden tehostaminen 1970-luvulla enemmän kuin toivottavaa. Niinpä onkin sangen ymmärrettävää, että Hirschbergin tuolloin kehittämässä algoritmissa keskitytään juuri muistitilan tarpeen radikaaliin vähentämiseen aikaisemmasta. Suoritusaikaan uusi algoritmi ei sen sijaan tuonut vielä minkäänlaista lisätehoa. HIL-algoritmia voidaankin kiistatta pitää eräänlaisena Wagnerin ja Fischerin algoritmin lineaaritilaisena varianttina laskentatapansa konservatiivisuuden johdosta.

Hirschbergin algoritmisesti tärkein uusi havainto perustuu WFI:ssä käytettävän matriisin *M* arvojen evaluointiin. Tarkastellaan seuraavaksi matriisin yksittäisen solun *M*[*i, j*] arvojen määräytymistä Wagnerin ja Fischerin algoritmissa.

$$1° \ (x_i = y_j) \ \Rightarrow \ M[i, j] = M[i–1, j–1] + 1$$

$$2° \ (x_i \neq y_j) \ \Rightarrow \ M[i, j] = Max\{M[i, j–1], M[i–1, j]\}$$

WFI:n kahden laskentasäännön perusteella soluun *M*[*i, j*] asetettava arvo määräytyy yksinomaan joko sen kulmittain ylävasempaan naapurisoluun (*M*[*i*–1, *j*–1]) tai sekä lähinnä vasempaan (*M*[*i, j*–1]) että yläpuoleiseen (*M*[*i*–1, *j*]) soluun tallennettujen arvojen perusteella. Toisin sanoen, kulloinkin tarkasteltavan solun kannalta ovat merkityksellisiä ainoastaan nykyrivillä *i* ja edellisellä rivillä *i* – 1 sijaitsevien solujen sisältämät arvot. Aikaisempien rivien 0..*i*–2 ($1 < i \leq m$) arvot ovat PYAn pituuden määräämisen kannalta jo tarpeettomia, ja ne voidaan unohtaa. Kun laskenta on edennyt matriisin oikean alanurkan soluun *M*[*m, n*] asti, saadaan koko syötemerkkijonojen PYAn pituus selville. Pituuden tullessa selvitetyksi ovat ainoastaan matriisin kahden viimeisen rivin *m* ja *m* – 1 tiedot tarpeellisia. Täten (*m* + 1) ∗ (*n* + 1):n suuruinen matriisi *M* voidaan HIL-algoritmissa kutistaa kahdeksi (*n* + 1):n mittaiseksi *vektoriksi*, joista ensimmäinen pitää muistissa nykyriviä edeltävän rivin ja toinen nykyrivin tiedot[47]. Laskentavaiheessa tarvittavan muistitilan määrä pienentyy siten voimakkaasti WFI:n *O*(*mn*):stä *O*(*n*):ään. PYAn pituuden laskemiseksi tehtävän työn määrä ei kuitenkaan muutu, sillä WFI:n matriisiesityksen kaikki solut käydään läpi myös HIL-algoritmissa syötteistä riippumatta. HIL ei siten suoritusajassa mitattuna vaikuta tehokkaalta algoritmilta pitkille syötteille, joiden PYAn ratkaisemisen nopeudella on soveltajalle oleellisen suuri merkitys.

3.2.1.2 PYAn ratkaisuksi kelpaavan jonon muodostaminen

Mikäli PYAn pituuden lisäksi halutaan selvittää myös jokin PYAn muodostavista jonoista, aiheuttaa WFI:ssä käytettävästä matriisista *M* luopuminen HIL:ssä jonkin verran mutkia matkaan – toisin kuin pelkkää PYAn pituutta määrättäessä. HIL-algoritmissa menetetään nimittäin matriisin *M* myötä mahdollisuus selvittää PYAn muodostava jono matriisiin tallennettuja arvoja tutkimalla kuten WFI:ssä, sillä ainoastaan kahden viimeisen rivin tiedot säilytetään. Hirschbergin lineaaritilaisessa algoritmissa matriisin puuttuminen on korvattu rekursiivisella, ns. *hajota ja hallitse* (lat. *divide et impera*) -tekniikkaan nojautuvalla PYA-jonon konstruointimenetelmällä.

[47]Analysoitaessa algoritmin toimintaa tarkemmin havaitaan, että algoritmi voidaan helposti muuntaa toimimaan myös pelkästään yhden matriisin *M* rivin turvin.

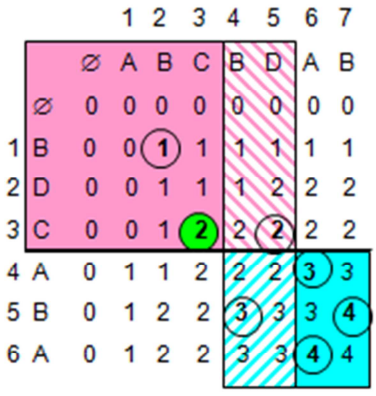
Hakualueen puolitustekniikka

Syötevektoreiden X ja Y pisimmän yhteisen alijonon S voidaan olettaa muodostuvan kahdesta osajonosta S_1 ja S_2, joille on voimassa $pituus(S_1) + pituus(S_2) = pituus(S) = p$. Hirschbergin lineaaritilaisessa algoritmissa osajono S_1 määrätään valitsemalla siihen tarkalleen ne S:n merkit, jotka on saatu syötevektorin X indeksipaikoista $1..\lfloor m/2 \rfloor$. Merkitään kyseistä X:n indeksialuetta X_1:llä. Sijaitkoot tällä alueella S:ään valituista merkeistä k ensimmäistä. Siten $pituus(S_1) = k$. Tällöin osajonon S_2 pituudeksi määräytyy vastaavasti $p - k$, ja siihen kuuluvat merkit ovat peräisin X:n indeksialueelta $\lfloor m/2 \rfloor+1..m = X_2$. Koska PYA-jonoa $S[1..p]$ sekä syötevektoreita $X[1..m]$ ja $Y[1..n]$ sitoo toisiinsa funktio

$$f\colon S[1..p] \rightsquigarrow X[1..m] \times Y[1..n]\colon$$
$$(f(i) = (k, l) \Leftrightarrow s_i = x_k = y_l, (1 \le i \le p), (1 \le k \le m), (1 \le l \le n)).$$
$$\text{Lisäksi, kun } (1 \le i < j \le p),\ f(j) = (u, v) \Rightarrow (u > k) \wedge (v > l).$$

eli S:ään kuuluvien merkkien sijaintipaikkojen indeksit syötevektoreissa X ja Y kasvavat monotonisesti ja aidosti S:n indeksin kasvaessa, voidaan jono S osittaa — paitsi syötevektorin X — niin myös syötevektorin Y indeksien suhteen. Mahdollisiksi osituskohdiksi kelpaavat sarakkeet $Yind(s_k)..Yind(s_{k+1})–1$, missä merkintä $Yind(s_k)$ tarkoittaa syötevektorin Y sitä indeksiä, josta PYA-jonon S k. eli sen alkuliitteen S_1 viimeinen merkki on valittu. Tällöin syötevektoreiden alkuliitteiden $X[1..\lfloor m/2 \rfloor]$ ja $Y[1..Yind(s_k)]$ välisen PYAn pituus on k. Vastaavasti loppuliitteiden $X[\lfloor m/2 \rfloor+1..m]$ ja $Y[Yind(s_k)+1..n]$ PYAn pituuden on oltava $p - k$, sillä jonon S_2 ensimmäisen merkin X-indeksi $\ge \lfloor m/2 \rfloor + 1$ edellä tehdyn X:n osituksen perusteella. Koska jonon S_1 viimeisen merkin Y-indeksi puolestaan on $Yind(s_k)$, täytyy jonon S_2 ensimmäisen merkin sijaintipaikka Y:ssä eli $Yind(s_{k+1})$ olla $Yind(s_k)$:n oikealla puolella, jotta S olisi syötevektoreiden X ja Y alijono. Täten on olemassa vähintään yksi katkaisupiste $(\lfloor m/2 \rfloor, Yind(s_k))$, jonka suhteen syötteistä muodostuva matriisi M voidaan jakaa neljään suorakulmioon, joista ainoastaan kahta tarvitsee tutkia edelleen PYA-jonon määräämiseksi. Seuraava esimerkki valaissee tilannetta lukijalle.

Esimerkki 3.10: X =”BDCABA”, Y =”ABCBDAB”, p = 4, S_1= ”BC”, S_2 ”AB” / ”BA”



		1	2	3	4	5	6	7
		∅	A	B	C	B	D	A	B
	∅	0	0	0	0	0	0	0	0
1	B	0	0	1	1	1	1	1	1
2	D	0	0	1	1	1	2	2	2
3	C	0	0	1	2	2	2	2	2
4	A	0	1	1	2	2	2	3	3
5	B	0	1	2	2	3	3	3	4
6	A	0	1	2	2	3	3	4	4

Kuvassa vihreällä pallolla merkitty piste (3,3) jakaa PYA-jonon ”BCAB” tai ”BCBA” osajonoihin S_1=”BC” ja S_2=”AB” tai ”BA”. Jono S_1 muodostuu punaisen suorakulmion peittämistä alkuliitteistä X[1..3] = ”BDC” ja Y[1..3] = ”ABC” ja S_2 sinisen suorakulmion rajaamista loppuliitteistä X[4..6] = ”ABA” ja Y[4..7] = ”BDAB”. Myös pisteet (3,4) ja (3,5) kelpaisivat katkaisupisteiksi. Valittaessa piste (3,5) voitaisiin löytää myös ratkaisu ”BDAB”. Tällöin alkuliitteet pitenisivät ja loppuliitteet lyhenisivät raidoitettujen alueiden verran. Värjättyjen suorakulmioiden ulkopuolisten pisteiden tarkastelu on jatkossa tarpeetonta yhden PYA-jonon esiintymän löytämiseksi.

Takautuva PYAn pituuden laskenta

Edellä saatiin osoitettua, että koko syötevektoreiden välinen PYA saadaan muodostettua jakamalla sekä X- että Y-syötejono sopivasti kahteen osajonoon ja ratkaisemalla niiden PYAt. Vektorin X pituus voidaan yksinkertaisesti puolittaa, mutta sopivan katkaisukohdan löytäminen Y:lle on pulmallisempaa. Sen selvittämiseksi on nimittäin tiedettävä, montako merkkiä syötejonosta Y pitää valita PYA-jonon ositteisiin S_1 ja S_2 puolitettaessa X:n pituus. Asia kuitenkin ratkeaa pienellä ajatustyöllä. Syötevektorin X alkupuoliskon ja koko Y-vektorin välinen PYA saadaan ratkaisemalla alkuperäisestä ongelmasta puolet. Tällöin saadaan selville $X[1..\lfloor m/2 \rfloor]$:n ja Y:n kaikkien erimittaisten alkuliitteiden välisten pisimpien yhteisten alijonojen pituudet. Ne otetaan talteen vektoriin *alkuosa*[0..n]. Vektorin *alkuosa* indeksi symbolisoi Y:n alkuliitteen pituutta. Tarkastellaan seuraavaksi alkuperäisen ongelman loppupuoliskoa eli indeksialueita X_2 = $X[\lfloor m/2 \rfloor+1..m]$ ja $Y[1..n]$. Koska kahden merkkijonon välisen PYAn pituus on sama riippumatta siitä, prosessoidaanko jonoja alusta loppuun vai lopusta alkuun päin [Muk80][NKY82], voidaan X_2:n ja koko Y-vektorin välinen PYA ratkaista muodostamalla X_2:sta ja Y:stä käänteiset jonot $X^*[m..\lfloor m/2 \rfloor+1]$ ja $Y^*[n..1]$ ja ratkaisemalla niiden välinen PYA. Prosessoitaessa käänteisten jonojen viimeistä riviä eli $\lfloor m/2 \rfloor+1$:tä tallentuvat vektoriin *loppuosa*[0..n] $X^*[m..\lfloor m/2 \rfloor+1]$:n ja Y^*:n erimittaisten alkuliitteiden välisten pisimpien yhteisten alijonojen pituudet. Vektorin *loppuosa* indeksi kuvaa valitun Y^*:n alkuliitteen pituutta.

Jonon ratkaisutekniikka

Lähdettäessä ratkaisemaan sekä syötemerkkijonojen PYAn muodostavaa jonoa että sen pituutta halkaistaan siis jono X mahdollisimman keskeltä kahdeksi ositteeksi X_1 ja X_2, joiden pituuksiksi saadaan selvästikin $\lfloor m/2 \rfloor$ ja $\lceil m/2 \rceil$. Kun kummankin puoliskon PYAn pituudet on saatu laskettua vektoreihin *alkuosa* ja *loppuosa*, lasketaan vektoreiden

alkuosa indekseissä *j* ja vektorin *loppuosa* indekseissä $n - j$ ($0 \le j \le n$) olevat arvot yhteen. Ainakin yhden tällä tavoin muodostetuista pareittaisista summista on oltava arvoltaan *p*, koska edeltäneen tarkastelun perusteella on olemassa piste ($\lfloor m/2 \rfloor$, *j*), joka osittaa syötevektorit *X* ja *Y* alku- ja loppuliitteiksi, joiden yhteenlasketun PYAn pituus on koko alkuperäisen ongelman PYAn pituus *p*. Vastaavasti yksikään pareittainen summa ei voi olla *p*:tä suurempi, sillä silloin alkuperäisellä ongelmalla olisi oltava *p*:tä pidempi pisin yhteinen alijono, mikä on vastoin alussa tehtyä oletusta. HIL-algoritmissa valitaan katkaisupisteen *Y*-indeksiksi vektorin *alkuosa pienin sellainen indeksi j*, jolla summa $alkuosa_j + loppuosa_{n-j}$ maksimoituu. Esimerkin 3.10 mukaisilla syötteillä tallentuisivat vektoriin *alkuosa*[0..*n*] arvot 0, 0, 1, 2, 2, 2, 2, 2 ja vektoriin *loppuosa*[0..*n*] arvot 0, 1, 2, 2, 2, 2, 2, 3. Tällöin pareittaisiksi summiksi $alkuosa_j + loppuosa_{n-j}$ muodostuisivat 3, 2, 3, 4, 4, 4, 3, 2, joten katkaisupisteeksi valittaisiin (3, 3) ja PYAn pituudeksi saataisiin 4, kuten kyseisestä esimerkistä ilmenee.

Kun katkaisupiste ($\lfloor m/2 \rfloor$, *j*) on saatu selville, tiedetään, että ainakin yksi PYAn esiintymistä voidaan muodostaa tarkastelemalla vain alkuliitteiden $X[1..\lfloor m/2 \rfloor]$ ja $Y[1..j]$ sekä loppuliitteiden $X[\lfloor m/2 \rfloor+1..m]$ ja $Y[j+1..n]$ välisiä täsmäyksiä. Tarkasteltavat alueet sijaitsevat matriisin *M* vasemmassa yläkulmassa, josta kootaan PYAn alkuliite S_1, ja oikeassa alakulmassa, josta kerätään PYAn loppuliite S_2. Selvästikin PYAn alku- ja loppuliitteille S_1 ja S_2 pätevät samat ominaisuudet kuin koko PYA-jonolle *S*: nekin voidaan jakaa kahteen osajonoon S_{11} ja S_{12} sekä S_{21} ja S_{22} puolittamalla alueet syötevektorin *X* suhteen. Siten indeksialue X_1 jakaantuu alueiksi $X_{11} = X[1..\lfloor m/4 \rfloor]$ ja $X_{12} = X[\lfloor m/4 \rfloor+1..\lfloor m/2 \rfloor]$ ja X_2 alueiksi $X_{21} = X[\lfloor m/2 \rfloor+1..\lfloor 3m/4 \rfloor]$ ja $X_{22} = X[\lfloor 3m/4 \rfloor+1..m]$. Jokaiselle em. alueista haetaan jälleen sopiva katkaisupiste *Y*:n suhteen. Tällä tavoin jatketaan alueiden koon systemaattisesti pienentyessä niin pitkään, kunnes ratkaistava aliongelma muuttuu *triviaaliksi* eli saavutetaan *rekursion kanta*. Tällöin joko vektorin *X* osite on enää yhden mittainen tai jonon *Y* osite ei sisällä enää yhtään merkkiä, jolloin ositteen PYAn pituus on selvästikin nolla. Ensimmäisessä tapauksessa ositteen PYA on yhden mittainen, jos siinä esiintyvä *X*:n merkki löytyy tarkasteltavasta *Y*:n ositteesta; muutoin tällaisenkin ositteen PYAn pituudeksi tulee nolla.

Pisteiden valitseminen PYA-jonoon tapahtuu ainoastaan saavutettaessa rekursion kanta. Olettaen, että tuolloin tarkasteltavaan *X*:n ositteeseen kuuluu *tarkalleen yksi* ja *Y*:n ositteeseen *vähintään yksi merkki*, selataan ositteen sarakkeita järjestyksessä niin pitkään, kunnes ositteen ainoa *X*:stä valittu merkki täsmää jollain sarakkeella olevaan *Y*:n merkkiin (tällöin ositteen PYAn pituus on 1) tai *Y*:n merkit loppuvat kesken (alueen PYA on tyhjä jono). Palataan vielä hetkeksi esimerkkiin 3.10. Kuvassa punaiseksi värjätty alue $X[1..3]$, $Y[1..3]$ ositettaisiin seuraavaksi pisteen (1, 2) kohdalta, sillä sen valitseminen osituspisteeksi maksimoi alueen PYAn pituudeksi 2: minkä tahansa muun katkaisukohdan valitseminen johtaisi esimerkissä tätä lyhyempään PYAan. Osituksen tapahduttua paikassa (1, 2) jää tämän yläpuoleiseen ositteeseen jäljelle enää yksi *X*:n merkki, ”B”, joten alueen jakamisyritys riveittäin kahtia edellisessä kappaleessa

esitetyllä tavalla tuottaa ensimmäiseksi syötteeksi *tyhjän alueen X*[1..0], *Y*[1..2], jonka käsittely lopetetaan saman tien tilanteen paljastuttua. Pisteeseen (1, 2) oikeasta alakulmastaan rajoittuvasta alueesta jääkin siten tarkasteltavaksi enää jälkimmäinen osa eli *X*[1..1], *Y*[1..2]. Tämäkin on algoritmin toiminnan kannalta perustapaus, sillä vektorin *X* alue on puristunut *yhteen merkkiin X*[1], jota nyt etsitään ositteeseen kuuluvista *Y*-indekseistä. Tällöin piste (1, 2) tulee valituksi PYAn ensimmäiseksi täsmäykseksi. Samaa tekniikkaa sovelletaan jatkossa aina, kun tarkasteltava alue on kutistunut ainakin toiselta dimensioltaan riittävän pieneksi.

3.2.1.3 HIL-algoritmin toteutus

Liitteen kohdassa 11.2.1 esiteltävä Hirschbergin lineaaritilaisen algoritmin pseudokoodilistaus noudattelee täysin alkuperäisartikkelissa [Hir75] esitettyä ratkaisumenettelyä. Koska Hirschberg on valinnut rekursion ensimmäiseksi kantatapaukseksi tilanteen, jossa $n = 0$, olen pitänyt tämän voimassa myös algoritmin omassa toteutuksessani siitäkin huolimatta, että rekursion aikana työni alussa esitettyä oletusta $m \leq n$ voidaan joutua (ja lähestulkoon aina joudutaankin!) rikkomaan. Sen sijaan Hirschbergin esittämää PYAn osaratkaisujonojen yhdistämistekniikkaa on ohjelmoidussa versiossa yksinkertaistettu artikkelissa olevasta versiosta siten, että PYA-jono kootaan kasvavassa indeksijärjestyksessä alusta loppuun pelkästään yhteen tulosvektoriin. Tähän on päädytty, jotta PYA-jonoille tehty muistinvaraus olisi yhdenmukainen tässä tutkielmassa toteutettavien muiden PYA-algoritmien kanssa ja jotta vältyttäisiin alkuperäisversiossa esiintyvien, PYAn osaratkaisujen yhdistämiseksi tarvittavien apuvektorien *S*1 ja *S*2 perustamiselta.

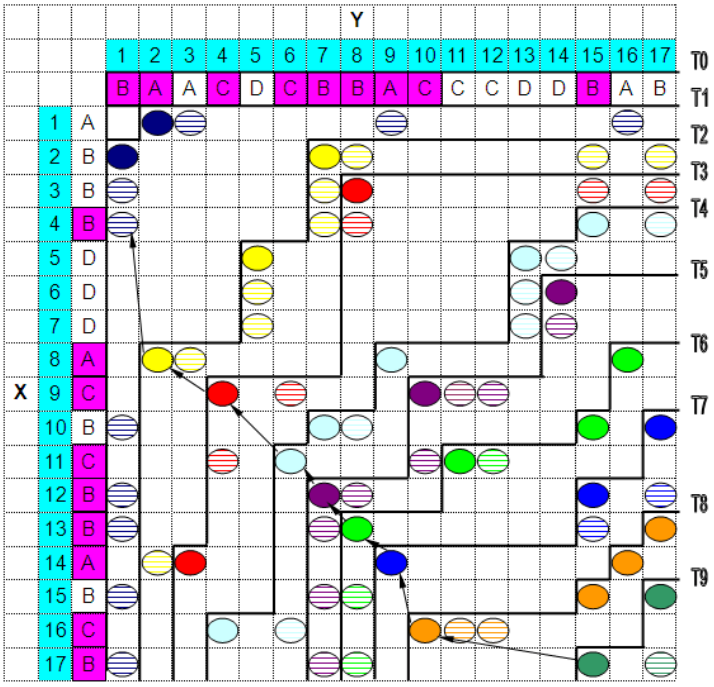
Hirschbergin lineaaritilaisen algoritmin toteutus on varsin yksinkertainen. Pääalgoritmissa kutsutaan ainoastaan proseduuria *RatkaiseRekursiivisestiHIL*, jonka lopetettua toimintansa lista *S* sisältää PYA-jonon, jonka pituus selvitetään vielä ennen algoritmin toiminnan loppumista. Edellä mainitun proseduurin suoritus päättyy, jos syötevektorin *X* osite kutistuu yhden mittaiseksi tai *Y*:n ositteesta tulee tyhjä jono. Muussa tapauksessa ositetaan käsiteltävä ongelma kahtia *X*:n suhteen ja määrätään katkaisupiste *Y*:lle kutsumalla proseduuria *LaskePituudet* ongelman alkuliitteille sekä käännetyille loppuliitteille. Tämän jälkeen seuraa kaksi uutta kutsua proseduurille *RatkaiseRekursiivisestiHIL*: yksi kumpaakin muodostunutta aliongelmaa kohti. Lopuksi yhdistetään osaongelmien tulokset.

3.2.1.4 Esimerkki

Seuraavassa kuvassa on havainnollistettu Hirschbergin lineaaritilaisen algoritmin toimintaa yksittäiselle 17 x 17 -kokoiselle esimerkkiaineistolle. Syöttöaakkosto koostuu

joukkoon {”A”, ”B”, ”C”, ”D”} kuuluvista merkeistä. Syötteiden PYAn pituudeksi saadaan 9. Algoritmin valitsema PYA-jono on merkitty kuvaan nuolilla. Koska HIL-algoritmi suorittaa osaongelmiin jaon aina mahdollisimman vasemmalta, algoritmi valitsee PYAn esiintymistä sen, joka kulkee matriisissa *M* alimpana. Mukaan hyväksytään myös ei-dominantteja täsmäyksiä kuten paikassa (4, 1) sijaitseva.

Esimerkki 3.11: *HIL-algoritmin löytämä ratkaisu esimerkin 3.1 syötteille*



Seuraavassa ovat vielä esitettyinä kaikki proseduurin *RatkaiseRekursiivisestiHIL* kutsut esimerkin syötejonoille sekä algoritmin asettamat osituspisteet, kunnes ensimmäinen PYAan kuuluva merkki löydetään.

1. 17, 17, X[1..17], Y[1..17], S (alkuperäinen ongelma)
2. 8, 2, X[1..8], Y[1..2], S1 (osituspiste (8, 2))
3. 4, 1, X[1..4], Y[1..1], S11 (osituspiste (4, 1))
4. 2, 0, X[1..2], Y[1..0], S111 (osituspiste (2, 0))
   Y[1..0] = ∅ ⇒ S111 = ∅ (kutsu 4 päättyy tähän)
5. 2, 1, X[3..4], Y[1..1], S112 (osituspiste (2, 0))
6. 1, 0, X[3..3], Y[1..0], S1121 (osituspiste (3, 0)
   Y[1..0] = ∅ ⇒ S1121 = ∅ (kutsu 6 päättyy tähän)
7. 1, 1, X[4..4], Y[1..1], S1122 (osituspiste (3, 0))
   X[4] = ’B’ = Y[1] ⇒ S1122 = ’B’ (kutsu 7 päättyy tähän)

   S112 = S1121 || S1122 = ∅ || ’B’ = ’B’ (kutsu 5 päättyy tähän)

   S11 = S111 || S112 = ∅ || ’B’ = ’B’ (kutsu 3 päättyy tähän)

3.2.1.5 Aika- ja tilavaativuus

HIL-algoritmissa alustustoimenpiteet ovat minimaaliset: ainoastaan proseduurissa *LaskePituudet* pitää alustaa kaksi pituudeltaan $(n + 1)$:n mittaista vektoria. Alustuksen kustannus on siten $O(n)$. Tarkastellaan seuraavaksi laskentavaihetta. Ylivertaisesti vallitsevin osa ajasta käytetään proseduurin *LaskePituudet* suorittamiseen, jossa joudutaan käymään lävitse kulloisenkin tarkasteltavan aliongelman syötejonoista muodostettavissa olevan matriisin M kaikki solut, mihin kuluu aikaa $O(Max\{1, uv\})$, missä $(1 \leq u \leq m)$ ja $(0 \leq v \leq n)$ ovat tarkasteltavan aliongelman dimensioiden pituudet[48]. Suoritettaessa kyseistä proseduuria ensimmäistä kertaa käydään läpi koko alkuperäisen ongelman solut, eli aikaa tarvitaan $O(mn)$. Proseduurissa *RatkaiseRekursiivisestiHIL* suoritetaan ongelman triviaalisuuden testaus, etsitään uuden aliongelman katkaisupiste pareittaisten summien avulla sekä yhdistetään lopputulokset. Näistä kaikista toimenpiteistä selvitään ajassa $O(m + n)$.

Proseduuri *RatkaiseRekursiivisestiHIL* muodostaa jokaisella kutsukerralla itsestään kaksi uutta aktivaatiota, ellei paraikaa tarkasteltava aliongelma osoittaudu triviaaliksi. Jos syötteen pituus m on jokin kakkosen potenssi ja $n > 0$, ovat seuraavissa aktivaatioissa syötteiden pituudet $m/2$ ja k sekä $m/2$ ja $n - k$. Täten uusien aliongelmien kooksi saadaan $m/2(k + n - k) = mn/2$, joten jokaisen uuden kutsuparin myötä aliongelman koko puolittuu. Tällöin kokonaissuoritusaikaa kuvaa geometrinen sarjakehitelmä $mn + mn/2 + mn/4 + \ldots \approx 2mn$, joten aikakompleksisuudeksi saadaan $O(mn)$ kuten WFI-algoritmillakin. Jos puolestaan m ei ole kakkosen potenssi, voidaan alkuperäisen ongelman kokoa rajoittaa ylhäältä päin termillä $2mn$. Tällöin aikakompleksisuus voi asymptoottisesti korkeintaan kaksinkertaistua, mikä ei siten muuta saavutettua ordo-lauseketta lainkaan.

Algoritmissa tarvitaan muistitilaa yksinkertaisten muuttujien lisäksi syötevektoreille X ja Y sekä aliongelmien PYA-pituuksien laskemiseksi tarvittaville kahdelle vektorille, joiden pituus on $n + 1$. Proseduurista *RatkaiseRekursiivisestiHIL* muodostuu korkeintaan $m + m/2 + m/4 + \ldots + 1 = 2m - 1$ aktivaatiota, kun m on kakkosen potenssi. Ajon aikana muodostuvan rekursiopinon koko on suuruusluokkaa $O(\log m)$. Koska kaikki proseduurin käyttämät muuttujat vievät vakiomäärän muistia, tulee algoritmin kokonaistilantarpeeksi tavoiteltu lineaarinen $O(n)$.

3.2.2 Huntin ja Szymanskin algoritmi (HSZ)

3.2.2.1 Tavoitteena tehostettu versio Wagnerin ja Fischerin menetelmästä

Hunt ja *Szymanski* [Hun77] esittelivät riveittäin prosessoivan PYA-algoritminsa vuonna 1977, eli samana vuonna, jolloin *Hirschberg* julkaisi kahta korkeuskäyrä kerrallaan

[48] Aliongelman kompleksisuus on $O(1)$, jos syötevektorista Y saatu osite on tyhjä.

PYAn ratkaisevaa algoritmia käsitelleen tutkimuksensa [Hir77]. Tekijät käyttävät algoritmistaan nimeä *Algoritmi 2*, ja tässä työssä siitä käytetään merkintää *HSZ*.

HSZ-algoritmin teoreettiset perusteet ja tärkeimmät tietorakenteet ovat ehtineet tulla tutuiksi jo aikaisemmin tarkasteltaessa PYA-ongelman perusteita työn luvussa 2. Algoritmi nojautuu tekijöiden itsensä esittämiin kolmeen, kynnysarvojen ominaisuuksia koskevaan lemmaan, jotka käsiteltiin ja verifioitiin aliluvussa 2.3. Tietorakenteiden valintansa puolesta HSZ on verrattain yksinkertainen ja suoraviivainen. Täsmäysten löytämiseksi algoritmissa perustetaan *X*:n merkeille *esiintymälistat* niiden sijaintipaikoista *Y*:ssä. Esiintymälistoissa täsmäysten *Y*-indeksit ovat vähenevässä järjestyksessä. Syötevektorien ja esiintymälistojen lisäksi algoritmi vaatii ei-vakiotilaista muistia ainoastaan *kynnysarvovektorille* sekä *linkkivektorille*, jossa ylläpidetään löydettyjä täsmäysketjujen muodostamia polkuja PYAn palauttamiseksi laskennan päätyttyä. Kynnysarvovektorin nollas positio alustetaan nollaksi pseudotäsmäyksen $D_0 = (0, 0)$ sijaintisarakkeen mukaisesti, ja muihin indekseihin asetetaan alkuarvoksi $n + 1$, jonka tulkinta on ”määrittelemätön”.

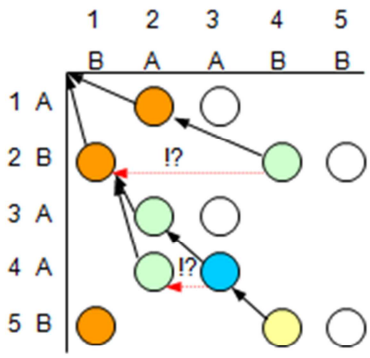
3.2.2.2 Toiminnan kuvaus

Algoritmin varsinainen laskenta tapahtuu kahdessa sisäkkäisessä silmukassa. Ulommassa niistä tutkitaan vuoron perään kukin *X*:n merkki eli matriisin *M* yksi rivi. Sisemmässä silmukassa tutkitaan rivin i ($1 \le i \le m$) kaikki täsmäykset lopusta alkuun päin rullaamalla merkin x_i esiintymälista läpi. Algoritmi selvittää kunkin täsmäyksen (x_i, y_j) kohdalla, mihin luokkaan se kuuluu. Tämä tapahtuu kohdistamalla puolitushaku[49] kynnysarvovektoriin, josta etsitään sellainen indeksi k, joka täyttää ehdon $kynnysarvo[k–1] < j \le kynnysarvo[k]$. Haku on aina tuloksellinen pysäytysalkioiden 0 ja n+1 ansiosta. Mikäli äskeisessä epäyhtälössä puolitushaun palauttamalla indeksillä k toteutuu yhtäsuuruus, täsmäys on luokan C_k ei-dominantti täsmäys[50] eikä se aiheuta lisätoimenpiteitä. Muussa tapauksessa löydetty täsmäys *saattaa olla dominantti*[51], ja $kynnysarvo[k]$:lle asetetaan uudeksi arvoksi j merkkinä siitä, että k. korkeuskäyrä siirtyy vasemmalle rivillä i. Lisäksi täsmäyssolmu (i, j) tallennetaan *linkkivektoriin* indeksiin k, josta asetetaan osoitin täsmäyksen generoineeseen $D_{k–1}$ -edeltäjään, jotta PYAan kuuluvat merkit pystytään löytämään laskentasilmukoiden lopetettua toimintansa.

[49] Puolitushakua voidaan käyttää, sillä lemman 2.1 perustella kynnysarvovektorin arvot ovat aina ei-vähéneviä.
[50] Kuvatun kaltaisia k-täsmäyksiä voi rivillä i olla korkeintaan yksi, jolloin luokan k korkeuskäyrä kulkee rivillä i pystysuoraan alaspäin.
[51] Algoritmi ei pysty päättämään, onko tarkasteltavalla rivillä mahdollisesti useita k-täsmäyksiä, joista selvästikin vain vasemmanpuoleisin on dominantti. Täten kynnysarvo- ja linkkivektorin indeksiin k tehdään yhdellä rivillä niin monta perättäistä päivitystä kuin k-täsmäyksiä riittää, vaikka vain viimeinen niistä on todellisuudessa tarpeellinen.

Koska jokaisen rivin täsmäykset tutkitaan lopusta alkuun, joudutaan pysähtymään rivin kaikkien *k*-täsmäysten kohdalla, vaikka selvästikin vain ensimmäinen näistä on dominantti. Tekijöiden esittämä tutkimisjärjestys on kuitenkin siinä mielessä järkevä, että se *takaa linkkivektorin päivitysten laillisuuden*. Seuraava esimerkki selvittää, mitä tapahtuisi, jos rivin täsmäysten tutkimisjärjestys olisikin päinvastainen.

Esimerkki 3.12: *X*[1..5] = ”ABAAB”, *Y*[1..5] = ”BAABB”, *p* = 4, *PYA* = ”BAAB”. *Esimerkki pulmatilanteesta, joka voisi syntyä, jos rivejä prosessoitaisiin vasemmalta oikealle HSZ-algoritmissa.*



Esimerkissä ensimmäisen rivin jälkeen *linkkivektori*[1] sisältää solmun (1, 2). Toiselta rivillä löydettäisiin aluksi D_1-täsmäyspaikasta (2, 1), josta asetetaan edeltäjälinkki pseudosolmuun (0, 0). Jos täsmäys paikassa (2, 1) vietäisiin nyt heti *linkkivektori*[1]:een, linkittyisi D_2-täsmäys (2, 4) solmun (2, 1) perään **punaisen** pisteviivanuolen mukaisesti. Vastaavanlainen tilanne toistuisi vielä rivillä 4. Virheellisten päivitysten ongelma vältetään kuitenkin HSZ-algoritmissa selaamalla kunkin rivin täsmäykset oikealta vasemmalle.

Kun viimeisenkin rivin kaikki täsmäykset on tutkittu, HSZ-algoritmi etsii kynnysarvovektorista suurimman sellaisen indeksin *k*, jossa esiintyy pienempi arvo kuin *n*+1. Kyseinen *k* on nyt samalla PYAn pituus *p*. Lopuksi algoritmi palauttaa vielä ratkaisun PYAlle lähtemällä liikkeelle linkkivektorin positiosta *p* ja tulostamalla ko. paikasta alkavan linkitetyn listan ensimmäistä täsmäystä vastaavan merkin jommastakummasta syötevektorista. PYA-jonon merkit tulostetaan siten lopusta alkuun päin, ja algoritmi palauttaa useasta ratkaisusta alimman mahdollisen, pelkästään dominanteista täsmäyksistä muodostuvan polun kuten *Hirschbergin I* algoritmi aliluvussa 3.1.1.

3.2.2.3 Algoritmin aika- ja tilavaativuus

Alustustoimenpiteistä selvitään ajassa $O(n)$, joka kuluu esiintymälistojen muodostamiseen. Kynnysarvovektorin alustaminen on tätä helpompi tehtävä ja vaatii aikaa $O(m)$. Laskentavaiheessa silmukat *S1* ja *S2* käyvät yhdistettyinä läpi kaikki syötevektoreiden väliset täsmäykset, joita on *r* kappaletta. Kutakin täsmäystä kohti joudutaan suorittamaan puolitushaku kynnysarvovektorista, jonka kustannus on $O(\log m)$. Vaikkei täsmäyksiä esiintyisi ongelmassa yhtäkään, joka tapauksessa kaikki rivit joudutaan selaamaan läpi ulommassa silmukassa. Siten laskentavaiheen kustannukseksi saadaan $O(m + r \log m)$. Aliluvussa 2.1.4 todettiin täsmäysten

enimmäismääräksi mn, joten jos nyt $m = n$, olisi HSZ:n pahimman tapauksen aikakompleksisuus suuruusluokkaa $O(n^2 \log n)$, joka on tähänastisista menetelmistä huonoin: se häviää jopa Wagnerin ja Fischerin naiiville algoritmille! Käytännön sovelluksissa tilanne on usein kuitenkin huomattavasti valoisampi. Etenkin syöttöaakkoston koon kasvaessa täsmäyksiä esiintyy verrattain harvoin, ja jos esimerkiksi $r \approx n$ ja syötejonot ovat yhtä pitkät, algoritmi suoriutuu tehtävästään ajassa $O(n \log n)$, joka on kertaluokkaa parempi kuin esimerkiksi WFI:n ja HI1:n $O(n^2)$. Alle $O(n)$:n suuruiseen työmäärään ei kuitenkaan milloinkaan päästä, sillä kumpikin syötejonoista on joka tapauksessa käytävä ainakin kertaalleen läpi. HSZ:n pseudokoodilistaus löytyy liitteestä kohdasta 11.2.2.

Algoritmin rivien käsittelysuunnasta (oikealta vasemmalle) johtuva ei-dominanttien täsmäysten kirjaaminen linkkivektoriin sekä tilapäisesti myös kynnysarvovektoriin hidastuttavat menetelmää oleellisesti, kun täsmäyksiä on paljon. Aliluvussa 3.2.7 esiteltävässä, HSZ-algoritmiin perustuvassa *Kuon* ja *Crossin* [Kuo89] algoritmissa kyseinen tehottomuuden lähde on eliminoitu.

HSZ-algoritmi vaatii syötevektoreiden lisäksi muistitilaa esiintymälistoille, kynnysarvovektorille sekä linkkivektorille. Kaikkia muita paitsi viimeksi mainittua varten riittäisi $O(n)$:n suuruinen muistitila. Linkkivektoriin joudutaan kuitenkin taltioimaan tiedot kaikista täsmäyksistä, joita se ei suoraan pysty havaitsemaan ei-dominanteiksi, joten algoritmin tilakompleksisuus on kokonaisuudessaan suuruusluokkaa $O(r+n)$.

3.2.3 Mukhopadhyayn algoritmi (MUK)

3.2.3.1 Toisenlainen lähestymistapa Huntin ja Szymanskin menetelmälle

Amar Mukhopadhyay [Muk80] esitteli rivi kerrallaan etenevän PYA-algoritmin vuonna 1980. Häntä voidaan pitää tähän mennessä mainituista PYA-menetelmien kehittäjistä sikäli epäonnisimpana, että hänen menetelmänsä muistuttaa teknisesti hyvin paljon jo kolme vuotta aikaisemmin julkaistua *Huntin* ja *Szymanskin* algoritmia, jonka olemassaolosta tekijä sai tietää vasta lähetettyään oman käsikirjoituksensa lehteen. Algoritmissa, josta Mukhopadhyay käyttää nimitystä *Algoritmi 2* ja joka tässä työssä tunnistetaan lyhenteellä *MUK*, käydään HSZ:n tapaan läpi kaikki syötteistä muodostuvan matriisin M rivit, tutkitaan tarkasteltavan rivin kaikki täsmäykset ja selvitetään, mihin luokkaan ne kuuluvat.

Vaikka MUK-algoritmin toiminnan perusperiaatteet ovat identtiset HSZ:n kanssa, toteutukseltaan algoritmit eroavat toisistaan kuitenkin selkeästi. HSZ-algoritmissa prosessointi aloitetaan syötevektorin X ensimmäisestä merkistä, ja kutakin X:n merkkiä x_i ($1 \le i \le m$) kohti selataan merkin yhteen suuntaan linkitettyyn esiintymälistaan

tallennetut esiintymät *Y*:ssä oikealta vasemmalle eli lopusta alkuun päin[52]. MUK:ssa rivit – eli syötevektorin *X* merkit – käsitellään kuitenkin lopusta alkuun päin aloittamalla merkistä x_m. Kulloinkin tarkasteltavan merkin x_i esiintymät *Y*:ssä selataan puolestaan vasemmalta oikealle eli alusta loppuun päin. Tästä seuraa, että MUK-algoritmi muodostaa korkeuskäyrät alhaalta ylöspäin, ja täsmäysluokan numeron kasvaessa ne siirtyvät asteittain kohti vasenta yläkulmaa: siis täysin päinvastoin kuin aikaisemmin esitellyissä menetelmissä. Lisäksi on huomion arvoista, *ettei MUK:ssa käytetä dynaamisia tietorakenteita mihinkään tarkoitukseen*: kaikki rakenteinen tieto säilötään vektoreihin. MUK:ssa myös ei-dominanttiin täsmäykseen törmääminen aiheuttaa algoritmissa aina kirjanpitoa, kun taas HSZ:ssa se ei ollut tarpeen, jos kohdataan rivin ainoa ei-dominantti *k*-täsmäys.

MUK:ssa vektorin *X* sisältämien merkkien esiintymälistat talletetaan vektoriin *EsListat*, jonka pituus on $n+\sigma_{xy}$, missä jälkimmäinen termi kuvaa niiden syöttöaakkoston Σ merkkien lukumäärää, jotka esiintyvät kummassakin syötevektorissa *X* ja *Y*. Esiintymälistavektori jakautuu σ_{xy} loogiseen osavektoriin, joista jokaisen pituus on $1 + frekv[s_g]$ $(1 \leq g \leq \sigma_{xy})$, missä jälkimmäinen termi tarkoittaa merkin s_g esiintymiskertojen lukumäärää syötevektorissa *Y*. Kunkin osavektorin $1..\sigma_{xy}$ ensimmäisessä positiossa pidetään muistissa arvoa $frekv[s_g]$, kun taas osavektorin loppuosa sisältää merkin s_g sijaintipaikat *Y*:ssä kasvavassa järjestyksessä.

Jotta päästäisiin nopeasti käsiksi kutakin merkkiä kuvaavan osavektorin alkamiskohtaan, tarvitaan avuksi toista apuvektoria nimeltä *ListanAlku*. Kyseisen vektorin pituus on *m* kuten syötejonolla *X*, ja sen paikkaan *i* tallennettu arvo ilmaisee, mistä kohdin vektoria *EsListat* alkaa merkin x_i esiintymistietoja *Y*:ssä kuvaava osavektori. Jos merkkiä x_i ei esiinny *Y*:ssä kertaakaan, sijoitetaan alustuksessa paikkaan $ListanAlku[i]$ arvo 0, ja vastaavan merkin osavektori puuttuu tällöin vektorista *EsListat*.

Algoritmin laskemien tulosten ylläpitoa varten tarvitaan avuksi vielä kolme muutakin vektoria. Ensimmäinen niistä on jo HSZ-menetelmästä tuttu *kynnysarvovektori*, jonka pituus on *m*+1. HSZ:lle käänteisen laskentajärjestyksen takia MUK asettaa vektorin indeksiin 0 alkuarvon *n*+1 nollan asemesta. Vastaavasta syystä sitä mukaa kuin kynnysarvovektorin indeksit kasvavat, niihin tallennetut arvot pienenevät. Vektorin muita kuin nollatta positiota ei alusteta, vaan erillinen muuttuja *pyapituus* estää määrittelemättömien arvojen lukemisen kynnysarvovektorista.

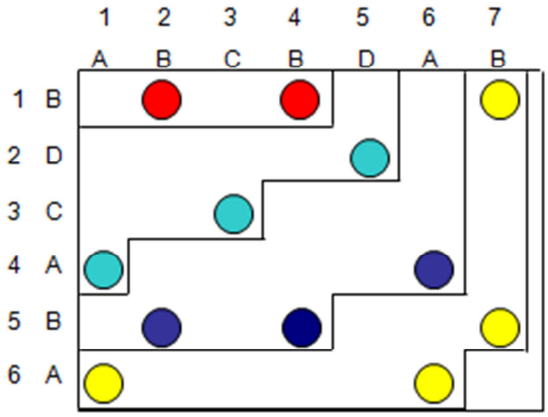
Laskennan aikana saavutetut tulokset tallennetaan vektoriin *RivienTiedot*, jolla on pituutta enintään *m*+*r*, missä *r* kuvaa täsmäysten määrää tarkasteltavassa ongelmassa. Tämäkin vektori jakautuu loogisesti enintään *m* eri jaksoon, joista kukin kuvaa rivikohtaisia kynnysarvoja. Kukin sen ei-tyhjistä osavektoreista alkaa tiedolla symbolin x_i esiintymälistaa kuvaavan osavektorin alkukohdasta. Tätä seuraavat arvot kuvaavat kyseisen rivin täsmäysten luokkia. Jos tietty *X*:n merkki x_i ei muodosta täsmäystä

[52] Mukhopadhyayn artikkelissa merkintä *n* kuvaa *X*:n ja *m* vastaavasti *Y*:n pituutta, mutta niiden merkitykset on vaihdettu päittäin käsillä olevan työn määritelmien mukaisiksi.

yhdenkään *Y*:n merkin kanssa, sitä kuvaava osavektori jää tyhjäksi, kuten jäi myös samaisen merkin osavektori vektorissa *EsListat*.

Vielä on tarpeen perustaa yksi apuvektori – *RivinAlku* – jonka pituus on *m*. Tätä vektoria tarvitaan kuvaamaan, mistä kohtaa rivin *i* täsmäyksiä koskevat tiedot alkavat tulosvektorissa *RivienTiedot*. Mikäli tietyllä rivillä *i* ei sijaitse yhtään täsmäystä, asetetaan *RivinAlku*[*i*] arvoksi 0. Seuraavan esimerkin tarkoituksena on havainnollistaa, millaisia arvoja edellä mainitut viisi vektoria saavat ratkaistaessa PYAa annetuille syötteille.

Esimerkki 3.13: *X* = ”BDCABA”, *Y* = “ABCBDAB”, σ = 4 (A=1, B=2, C=3, D=4), *p* = 4.



Oheiseen kuvaan on merkitty ongelman kaikki täsmäykset neljällä eri värillä värjätyillä palloilla. Pallon väri osoittaa täsmäyksen luokan (1 = **kelt.**, 2 = **sin.**, 3 = **turk.** ja 4=**pun.**). Piirretyt murtoviivat ilmaisevat korkeuskäyrien muotoa algoritmin suorituksen edetessä nyt alimmalta riviltä ylimmälle.

Mukhopadhyayn algoritmin tietorakenteiden sisältämät arvot algoritmin suorituksen päätyttyä ovat nähtävissä seuraavassa taulukossa. Lihavoidut lukuarvot toimivat esiintymälista- ja rivitietovektoreissa kuhunkin symboliin tai riviin liittyvien tietojen erotinkenttinä. Kysymysmerkit kynnysarvovektorin loppuosassa indikoivat kyseisten positioiden pysyvän alustamattomina esimerkin mukaisilla syötteillä.

kynnysarvot[0..7] = 8 7 6 5 4 ? ? ? (suurin arvoa T_k sisältävä sarake)
EsListat[1..11] = **2** 1 6 **3** 2 4 7 **1** 3 **1** 5 (merkkien esiintymälistat)
ListanAlku[1..6] = 4 10 8 1 4 1 (x_i:n esiintymälistan alkukohta)
RivienTiedot[1..18] = **1** 1 1 **4** 2 2 1 **1** 3 2 **8** 3 **10** 3 **4** 4 4 1
RivinAlku[1..6] = 15 13 11 8 4 1 (rivin *i* täsmäystietojen alkupositiot)

3.2.3.2 Toiminnan kuvaus ja analyysi

Vektoreiden *EsListat* ja *ListanAlku* arvot asetetaan alustusvaiheessa, eikä niitä päivitetä suorituksen aikana. Löydettäessä täsmäys riviltä *i* tutkitaan ensiksi, pidentääkö se aikaisemmin löydettyä PYAa. Esimerkissä 3.13 täsmäys (5, 2) pidentää PYAn ensi kertaa kahden mittaiseksi, sillä täsmäyksen sarakeindeksi 2 on pienempi kuin luokan 1 kynnysarvo 6 alimman rivin käsittelyn päätyttyä. Jos PYA pitenee tarkasteltavalla rivillä, kasvatetaan muuttujan *pyapituus* arvoa yhdellä ja nostetaan lippu, joka estää PYAn pituuden kasvattamisen toistamiseen samalla rivillä. Lippu kumotaan aina siirryttäessä tarkastelemaan uutta riviä. Ellei löydetty täsmäys paikassa (*i*, *j*) kuulunutkaan tarkasteluhetkellä korkeimpaan luokkaan, etsitään puolitushaulla

kynnysarvovektorista indeksi *k*, jolle on voimassa *kynnysarvot*[*k*+1] ≤ *j* < *kynnysarvot*[*k*], ja asetetaan kyseinen *j* luokan *k*+1 (uudeksi) kynnysarvoksi[53]. Täsmäyksen luokka tulostetaan myös vektoriin *RivienTiedot*. Hetki sitten tarkastelemamme täsmäys (5, 2) esimerkissä 3.13 päivittäisi *kynnysarvot*[2]:n arvoksi 2, ja vektoriin *RivienTiedot* paikkaan 5 tulostuisi niin ikään arvo 2.[54]

Kun algoritmi on ehtinyt käsitellä ylimmänkin rivin vasemmalta oikealle, saadaan PYAn pituus selville suoraan muuttujasta *pyapituus*. Mukhopadhyay ei esitä mitään eksaktia pseudokoodia itse PYA-jonon palauttamiseksi, vaan hahmottelee pelkästään sen suuntaviivat. PYAn täyttäminen etenee syötejonojen alusta loppuun päin, mutta jonon *k*. alkio onkin nyt jokin luokan *p*–*k*+1 täsmäyksen muodostaneista merkeistä johtuen algoritmin laskentajärjestyksestä. Jonon hakemiseksi käytetään hyväksi vektoria *RivienTiedot*. Vektorin indeksialueelta *RivinAlku*[*i*]+1..*RivinAlku*[*i*+1]–1 löydetään riviä *i* koskevat tiedot[55]. Jos riviltä *i* löytyy C_{p-k+1} täsmäys, viedään se PYA-jonon *k*. alkioksi. Ellei tällaisia riviltä *i* kuitenkaan löytynyt, jatketaan niiden etsintää seuraavilta riveiltä niin pitkään, kunnes sellainen ensi kertaa löytyy riviltä *i*+*u*, (0 < *u* < *m*–*i*), jolloin tallennetaan *PYA*[*k*]:hon merkki x_{i+u}. PYAn *k*. merkin löydyttyä käynnistetään riviä alempaa nyt PYAn *k*+1. alkion etsiminen, joka selvästikin muodostaa luokan *p*–(*k*+1)+1 = *p*–*k* täsmäyksen. PYAn kerääminen jatkuu samalla periaatteella, kunnes lopulta sen viimeinen eli *p*. merkki on löydetty 1-täsmäysten joukosta. Liitteen kohdassa 11.2.3 esitetään algoritmin pseudokoodi, jotta lukija voisi halutessaan simuloida algoritmin toimintaa.

Analysoitaessa algoritmin kompleksisuutta todetaan, että esiprosessointivaihe on aikakustannukseltaan $O(n)$, sillä syöttöaakkoston merkkien esiintymälistat voidaan koota lajitelluiksi osavektoreiksi käyttämällä lineaariaikaista *laskentalajittelua* [CLR93, 175 – 178], mikäli $\sigma = O(n)$. Varsinainen laskenta suoritetaan silmukoissa *S1* ja *S2*, joissa yhteensä käydään läpi tarkasteltavan ongelman kaikki täsmäykset, ja täsmäyksen luokan etsimiseksi saatetaan joutua tekemään puolitushaku, jonka kompleksisuus on $O(\log p)$. Muilta osin silmukoiden sisältämät lauseet ovat vakioaikaisia. Siten silmukoiden kokonaiskustannus on suuruusluokkaa $O(r \log p)$. Loppuprosessoinnissa joudutaan pahimmassa tapauksessa etsimään joka riviltä PYA-jonoon kelpaavia täsmäyksiä soveltamalla puolitushakua vektoriin *RivienTiedot*. Koska pahimmassa tapauksessa yhdellä rivillä on *p* täsmäystä ja tutkittavia rivejä on *m* kappaletta, vaatii PYAn rakentamisvaihe suoritusaikaa kokonaisuudessaan $O(m \log p)$. Siten koko algoritmin yhteenlasketuksi aikakompleksisuudeksi saadaan $O(n + r \log p + m \log p)$.

---

[53] Päivitys tehdään siitäkin huolimatta, vaikka tallennettava arvo olisi sama kuin nykyinen, eli löydetään rivin *i* ainoa *k*-täsmäys, joka on samalla ei-dominantti.

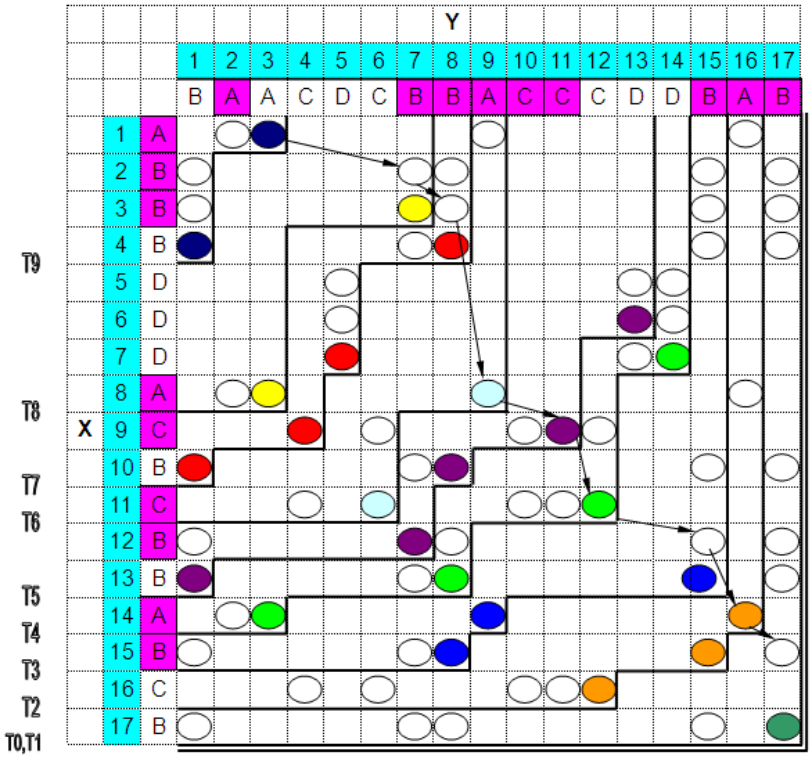
[54] Vektorin *RivienTiedot* paikat 1–3 on varattu rivin 6 tietoja varten: paikka 1 osoittaa merkin ”A” osavektorin alkukohdan 1 vektorissa *EsListat*, ja paikat 2 ja 3 täsmäysten (6, 1) ja (6, 6) luokat 1 ja 1. Paikkaan 4 tallentuu rivin 2 merkin ”B” osavektorin alkukohdan 4 vektorissa *EsListat*, joten täsmäyksen (5, 2) tiedot sijoitetaan paikkaan *RivienTiedot*[5].

[55] Jos rivillä *i* ei ole täsmäyksiä, hakualue on tyhjä, ja jos rivi on *m*, indeksialueen oikea raja on *pituus*(*RivienTiedot*).

Algoritmi vaatii muistitilaa syötevektoreidensa lisäksi edellä esitellyille apuvektoreille. Esiintymälistavektorin pituus on $O(n + \sigma_{xy})$, ja sen osavektorien alkukohdat osoittavan vektorin *ListanAlku* pituus on $O(m)$. Kynnysarvovektoria varten tarvitaan samoin $O(m)$ verran tilaa, koska joudutaan varautumaan pahimpaan tilanteeseen, että $p = m$. Täsmäystietojen ylläpitoa varten perustettava vektori *RivienTiedot* vaatii muistitilaa $O(m+r)$, ja sen osavektorien alkukohdan tunnistamiseksi tarvittava vektori *RivinAlku* on kooltaan $O(m)$. Siten algoritmin kokonaistilantarve on suuruusluokkaa $O(n+r)$, joka on sama kuin HSZ-algoritmilla.

Lopuksi esitetään vielä esimerkki, millaisen ratkaisun MUK-algoritmi löytää esimerkin 3.1 kaltaisille syötteille. Huomion arvoista on, että algoritmi voi valita PYAan mukaan myös ei-dominantteja täsmäyksiä kuten edellä esitelty HIL.

Esimerkki 3.14: *MUK-algoritmin löytämä ratkaisu esimerkin 3.1 syötteille. Valkoiset soikiot kuvaavat ei-dominantteja täsmäyksiä.*



3.2.4 Hsun ja Dun II algoritmi (HD2)

3.2.4.1 Lisää tehoa kynnysarvovektorin käsittelyyn

Hsu ja *Du* julkaisivat vuonna 1984 ilmestyneessä artikkelissaan [Hsu84] kaksi PYAn ratkaisevaa algoritmia. Näistä ensimmäinen, korkeuskäyrä kerrallaan laskentaa

suorittava, esiteltiin aliluvussa 3.1.3. Sen sijaan toinen algoritmeista, josta tekijät käyttävät nimitystä *Algoritmi 2* ja joka tässä työssä tunnistetaan lyhenteellä *HD2*, etenee riveittäin. Kehittäessään HD2-algoritmia tekijöiden ajatuksena on ollut Huntin ja Szymanskin HSZ-algoritmin tehostaminen. Samoin kuin HSZ niin myös HD2 käy ulommassa silmukassa läpi vuoron perään kaikki rivit väliltä 1..*m*.

Hsu ja Du kiinnittävät huomiota yhteen HSZ:n heikkouksista – täsmäysten luokkien jossain määrin tehottomaan selvittämiseen. Aina kun HSZ:ssa löydetään tarkasteltavalta riviltä i ($1 \leq i \leq m$) täsmäys sarakkeesta j ($1 \leq j \leq n$), etsitään puolitushaulla kynnysarvovektorista sellainen indeksi k ($1 \leq k \leq m$), jolle on voimassa *kynnysarvot*[k–1] < $j \leq$ *kynnysarvot*[k]. Jokainen puolitushaku maksaa selvästikin $O(\log m)$, koska hakualueena on koko kynnysarvovektori[56].

Tekijät esittävät haun tehostamiseksi *osittaista limitysoperaatiota*[57], jossa rivillä i sijaitsevien täsmäysten Y-indeksejä ja rivin i–1 laskennan päättymisen jälkeisiä kynnysarvoja verrataan toisiinsa. Oletetaan nyt, että z on pienin kynnysarvovektorin indeksi, jonka arvo on rivin i–1 käsittelyn jälkeen määrittelemätön (joko n+1 tai ∞). Seuraavaksi tutkitaan sisemmässä silmukassa laskevassa järjestyksessä vuoron perään kukin kynnysarvovektorin indekseistä z, z–1, …, 2, 1 ja etsitään merkin x_i täsmäyslistasta *pienintä sellaista arvoa* j, jolle on voimassa *kynnysarvot*[k–1] < j < *kynnysarvot*[k], ($1 \leq k \leq z$). Jos x_i:n esiintymälistat ovat lajiteltuina kasvavaan suuruusjärjestykseen, voidaan haku toteuttaa puolitushakuna, jonka kohdealueena on aluksi koko x_i:n esiintymälista. Ellei tällaista j:tä ole olemassa, tiedetään, ettei riviltä i löydy luokan z dominanttia täsmäystä lemman 2.3 perusteella. Muussa tapauksessa paikasta (i, j) löytyi uusi dominantti z-täsmäys. Tällöin *kynnysarvot*[z]:n arvoksi päivitetään j, ja lisäksi luodaan uusi solmu (i, j) *linkkivektori*[z]:sta alkavan listan alkuun, ja solmusta asetetaan edeltäjälinkki *linkkivektori*[z–1]:n alkusolmuun. Lisäksi kirjataan muistiin indeksi u, joka osoittaa, monentenako arvo j esiintyi x_i:n täsmäyslistassa[58].

Kun silmukassa lähdetään toiselle kierrokselle, jolloin k saa arvon z–1, yritetään löytää pienin Y-indeksi j siten, että $x_i = y_j$ ja *kynnysarvot*[z–2] < j < *kynnysarvot*[z–1]. Nyt on kuitenkin selvää, että kyseinen j voi löytyä x_i:n esiintymälistasta ainoastaan edellisen kierroksen haun palauttaman indeksin u vasemmalta puolelta, joten arvoa j haetaan x_i:n esiintymälistan indeksialueelta 1..u–1. Tällä tavoin jatketaan, kunnes kynnysarvovektorin kaikki positiot ykköseen asti on tutkittu, tai merkin x_i esiintymälista on puristunut tyhjäksi sen oikean rajan siirtojen ansiosta. Seuraavaksi esitetään

---

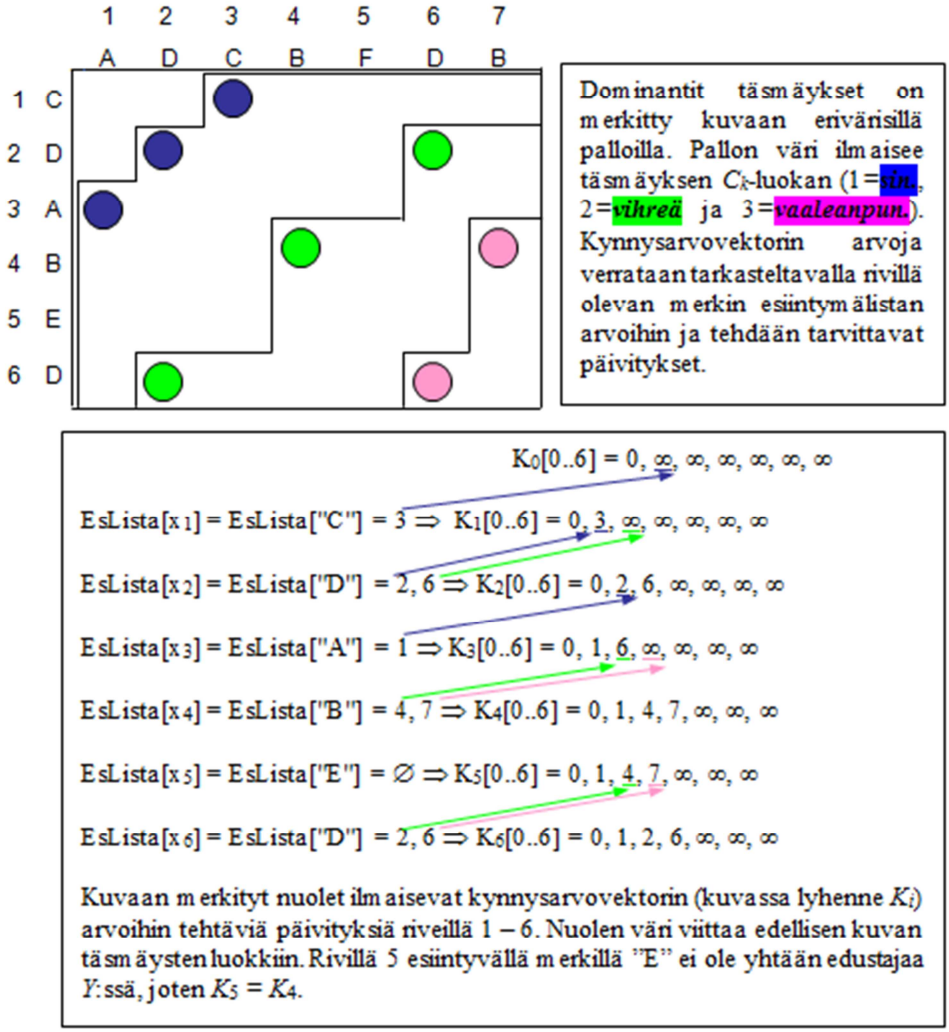
[56] Hakualueen ylärajaksi olisi järkevää asettaa m:n sijasta kynnysarvovektorin pienin indeksi, jossa esiintyy arvo n + 1 tai ∞. Tällöin puolitushaun kustannuksen yläraja olisi $O(\log p)$.

[57] Pelkkä limitys olisi sikäli kyseenalainen nimitys tässä suoritettavalle toiminnolle, sillä lopputulokseksi saadaan vain päivitetty kynnysarvovektori, eikä suinkaan vektoria, jonka pituus olisi kynnysarvovektorin ja x_i:n esiintymälistan pituus.

[58] Jos haku epäonnistui, u:n arvoksi asetetaan pienin x_i:n esiintymälistan indeksi, jolle *EsLista*[X[i], u] ≥ *kynnysarvot*[z–1]. Ellei tällaista ole, u = *pituus*(*EsLista*(X[i]))+1.

esimerkki, jonka tarkoituksena on havainnollistaa algoritmissa suoritettavan osittaisen limityksen etenemistä.

Esimerkki 3.15: *Osittaisen limityksen eteneminen HD2-algoritmissa, kun* X = ”CDABED”, Y = ’”ADCBFDB” ja p = 3.



Kannattaa huomioida, että algoritmissa voitaisiin edellä esitetty osittainen limitys suorittaa myös *päinvastoin*, eli etsimällä kutakin rivin i täsmäystä kohti puolitushaulla kynnysarvovektorista luokka, johon täsmäys kuuluu. Intuitiivisesti tuntuisi järkevältä suorittaa operaatio pienemmältä joukolta suuremmalle päin, eli jos $z \leq frekv[x_i]$, kannattaa osittainen limitys suorittaa ensiksi kuvatulla tavalla ja muussa tapauksessa jälkimmäisellä. Tekijät suosittelevat käyttämään *Hwangin* ja *Linin limitysalgoritmia* [Hwa72], ellei ole ennalta ennustettavissa, miten suuresti z ja $frekv[x_i]$ vaihtelevat toisiinsa nähden algoritmin suorituksen aikana.

3.2.4.2 Analyysi

Hsu ja Du eivät ole esittäneet artikkelissaan mitään pseudokoodia HD2-algoritmille. Liitteen kohdassa 11.2.4 esitetään sille kuitenkin yksi mahdollinen toteutus. Vektorimuotoisten esiintymälistojen rakentamisen kustannus on $O(n)$, kun syöttöaakkosto oletetaan tunnetuksi. Kynnysarvovektorin alustus vaatii puolestaan ajan $O(m)$, joten koko esiprosessoinnin kustannus on rajoitettavissa termillä $O(n)$. Silmukassa *S1* käydään läpi kaikki syötevektorin *X* merkit. Silmukan sisällä olevat lauseet ovat vakioaikaisia sisempää silmukkaa *S2* lukuun ottamatta, jota suoritetaan enintään *p* kertaa. Myös *S2*:n sisällä olevat lauseet ovat vakioaikaisia lukuun ottamatta puolitushakua, jossa suoritetaan kynnysarvovektorin ja rivin *i* täsmäysten välinen osittainen limitys. Tekijät väittävät artikkelissaan, että yksittäisen puolitushaun kustannukseksi saataisiin $O(\log(frekv[x_i]/z) + 1)$, missä $frekv[x_i]$ kuvaa täsmäysten määrää rivillä *i* ja *z* PYAn pituutta rivin *i*–1 jälkeen. Tässä vaikuttaisi olevan taustaoletuksena, että jokaisen puolitushaun kohdalla hakualue kutistuu. Näin ei todellisuudessa kuitenkaan välttämättä tapahdu. Jos vaikkapa riville *i* tultaessa PYAn pituus on $2k$, ja merkin x_i täsmäyslistan suurin indeksi on pienempi kuin luokan *k* kynnysarvo, joudutaan puolitushaussa asettamaan hakualueeksi x_i:n *koko esiintymälista k* kertaa peräkkäin. Täten vaikuttaisi, että pahimmassa tapauksessa koko algoritmin aikakompleksisuus olisi $O(n + mp \log q + mp)$, missä *q* on *Y*:ssä eniten esiintyvän *X*:n symbolin frekvenssi, tekijöiden ehdottaman lausekkeen $O(n + mp \log(n/p) + mp)$ sijaan, joka edellyttäisi hakualueen jatkuvaa tasaista lyhenemistä.

Algoritmissa vaaditaan HSZ:n tapaan vakiomäärää enemmän muistia syötevektoreiden lisäksi ainoastaan kynnysarvo- ja linkkivektorille, joista jälkimmäiseen kerätään kaikki dominantit täsmäykset. Siten HD2-algoritmin tilantarve on suuruusluokkaa $O(n + d)$.

3.2.5 Allisonin ja Dixin bittioperaatioihin perustuva algoritmi (ADI)

3.2.5.1 Bittioperaatioilla lisätehoa WFI-algoritmiin

Lloyd Allison ja *Trevor I. Dix* esittelivät vuonna 1986 ensimmäisen PYA-menetelmän, joka perustuu *bittijonojen tarkasteluun* [All86]. Kyseinen menetelmä, josta tässä työssä käytetään lyhennettä *ADI*, muistuttaa toimintaperiaatteeltaan pitkälti dynaamiseen ohjelmointiin perustuvaa Wagnerin ja Fischerin WFI-algoritmia [Wag74]. Allison ja Dix eivät lehtiartikkelissaan esitä menetelmästään mitään pseudokoodia, vaan algoritmin toiminta on kuvattu ainoastaan tekstuaalisesti. Tähän työhön on kuitenkin sisällytetty myös algoritmin kuvausta myötäilevä pseudokoodilistaus, joka löytyy liitteestä kohdasta 11.2.5.

Kuten aliluvusta 2.2 muistamme, WFI lähtee ratkaisemaan PYA-ongelmaa vertaamalla vuoron perään rivi kerrallaan syötejonon *X* kutakin merkkiä kaikkien syötejonon *Y* merkkien kanssa, eli testaamalla syötteistä muodostuvan matriisin *M* kaikki indeksiparit. Kehittyneemmässä HSZ-algoritmissa puolestaan rajoituttiin tarkastelemaan pelkkiä täsmäyksiä muodostamalla vektorin *Y* merkeistä yhteen suuntaan linkitetty esiintymälista. ADI lähtee aputietorakenteensa puolesta kuitenkin omille teilleen perustamalla syötevektorin *X* jokaista eri symbolia kohti ns. *esiintymäbittijonon* (lyh. *EBJ*), jonka pituus on *n*. Symbolia $X[i]$ $(1 \leq i \leq m)$ edustavan jonon paikassa j $(1 \leq j \leq n)$ oleva arvo 1 kertoo, että syötevektoreiden merkit x_i ja y_j *täsmäävät*. Muussa tapauksessa arvona esiintyy 0. Huomion arvoista on, että ADI-algoritmi etenee matriisissa *M* oikeasta alakulmasta lähtien rivi kerrallaan *oikealta vasemmalle* ja *alhaalta ylös*, eli rivinumerot kasvavat alhaalta ylöspäin ja sarakenumerot oikealta vasemmalle edettäessä. Jonot *X* ja *Y* esitetään matriisissa siten *käänteisessä järjestyksessä* eli lopusta alkuun päin. Yhtä symbolia kuvaavan esiintymäbittijonon pituus on $\lceil n/w \rceil$, missä *w* ilmaisee koneen sananpituuden bitteinä. Syötevektorin *Y* paikkaan *j* liittyvä esiintymäbittijonon bitti *b* löytyy konesanan $\lfloor j/w \rfloor$ indeksipaikasta *j* MOD *w*. Selvästikin, mitä enemmän bittejä mahtuu konesanaan, sitä vähemmän konesanoja tarvitaan esiintymäbittijonojen tallentamista varten. Seuraavassa esimerkissä havainnollistetaan esiintymäbittijonon rakennetta.

Esimerkki 3.16: *X*[1..6] =”BDCABA”, *Y*[1..7] =”ABCBDAB”, Σ={”A”,”B”,”C”,”D”}

Merkkijonon X symboleista kerätyt esiintymäbittijonot ovat:

EBJ[“A”] = 1000010 (“A” esiintyy syötejonon *Y* paikoissa 1 ja 6),
vastaavalla tavalla
EBJ[“B”] = 0101001, *EBJ*[“C”] = 0010000 ja *EBJ*[”D”] = 0000100.

Aikaisemmin tarkastellun perusteella tiedetään, että siirryttäessä matriisissa *M* rivi ylöspäin – eli kasvatettaessa *X*:n alkuliitteen pituutta yhdellä – PYA voi pidentyä korkeintaan yhdellä aikaisemmasta arvostaan. Lisäksi tiedetään, että sitä mukaa, kun otetaan uusi *X*:n merkki x_{i+1} käsittelyyn, saatetaan *k*:n mittainen *X*:n ja *Y*:n yhteinen alijono löytää valitsemalla *Y*:stä vähemmän merkkejä kuin yhtä lyhyemmän *X*:n alkuliitteen $X[1..i]$ kanssa. ADI-algoritmissa matriisi *M* on kooltaan $(m+1)$ x *n*, mutta sen soluihin tallennetaan pelkästään *nollia tai ykkösiä*. Pelkistä nollista koostuvaa *alinta eli nollatta riviä* tarvitaan alustusta varten. Soluun (i, j) tallennetaan ykkönen tarkalleen silloin, kun (i, j) esiintyy jonkin korkeuskäyrän – samantekevää, monennenko – *pystysuoran osan pisteenä*. Muulloin soluun asetetaan arvoksi nolla. Ykkösbittien lukumäärä matriisin *M* rivillä *i* kuvastaa alkuliitteen $X[1..i]$ ja koko *Y*-vektorin välisen PYAn pituutta. Matriisin kukin rivi jakautuu loogisesti *segmentteihin*, joiden

rajoittimina toimivat rivillä esiintyvät ykkösbitit[59]. Jokaisesta ykkösbitistä lähtien riviä vasemmalta oikealle selattaessa alkaa uusi segmentti, joka päättyy juuri ennen seuraavaa ykkösbittiä.

Matriisin *i*. rivin arvot saadaan laskettua käyttämällä hyödyksi riville *i*–1 sekä esiintymäbittijonoon *EBJ*[*X*[*i*]] tallennettuja arvoja. Näille muodostetaan ensinnä indeksikohtaisesti pareittainen TAI-operaatio, jonka tulos tallennetaan muuttujaan *z*. Siinä esiintyy ykkösbittejä sekä kaikissa niissä indekseissä j ($1 \leq j \leq n$), joissa nykyrivillä tarkasteltava merkki x_i täsmää y_j:n kanssa, että indekseissä, joissa matriisin *M* rivillä *i*–1 esiintyy ykkönen. Viimeksi mainitut ykkösbitit ovat tulkittavissa *täsmäysluokkien kynnysarvojen sijaintipaikoiksi* rivillä *i*–1. Tämän jälkeen matriisin riville *i*–1 tehdään *looginen siirto vasemmalle* siten, että oikeanpuoleisimpaan segmenttiin syötetään ykkönen, ja operaation tulos tallennetaan tilapäismuuttujaan *t*. Vastaavasti rivin vasemmassa reunassa sijainnut bitti menetetään. Apumuuttujassa *t* sijaitsee selvästikin nyt ykkönen tarkalleen *kunkin segmentin oikeanpuoleisimmassa positiossa*. Sitten muuttujasta *z* vähennetään edellä lasketun tilapäismuuttujan *t* arvo, ja vähennyslaskun tulos talletetaan muuttujaan *z'*. Kyseisen muuttujan bitit poikkeavat arvoiltaan muuttujan *z* biteistä kunkin segmentin oikeasta reunasta lukien aina siihen indeksiin asti, jossa kohdataan *segmentin oikeanpuoleisin ykkönen* muuttujassa *z*. Kun seuraavana toimenpiteenä muodostetaan muuttujien *z* ja *z'* välinen poissulkeva TAI-operaatio (engl. XOR, merkintä $\oplus$), saadaan tämän tuloksena bittijono *u*, jossa esiintyy ykkösbitti tarkalleen niissä positioissa, joissa *z*:n ja *z'*:n vastinbitit poikkeavat toisistaan. Näiden bittien arvot kääntyivät siten *päinvastaiseksi* operaatiota *z* – *z'* suoritettaessa.

Kun lopulta muodostetaan vielä muuttujien *z* ja *u* välinen looginen JA-operaatio, saadaan lopputulokseksi merkkijono, jossa rivin *i*–1 segmenttien ykkösbitit ovat joko säilyneet paikoillaan tai siirtyneet kohti niiden oikeaa reunaa. Lisäksi pelkkiä nollia sisältäneeseen ensimmäiseen segmenttiin ilmestyy *ykkönen*, jos PYA piteni viimeksi tarkastellun merkin x_i vaikutuksesta. Muodostunut jono edustaa nyt matriisin *M* riviä *i*, jonka segmenttirajat määräytyvät uudelleen ykkösbittien mahdollisesti muuttuneen sijainnin mukaan. Seuraavassa esimerkissä 3.17 nähdään, miltä matriisi *M* näyttää syötejonojen ollessa samat kuin esimerkissä 3.16. Esimerkin loppuosassa tarkastellaan lähemmin erityisesti rivien M_1 ja M_5 muodostamista riveihin M_0 ja M_4 sekä merkin "B" esiintymäbittijonoon perustuen.

Käytännön syötejonot ovat niin pitkiä, etteivät matriisien rivit – sen kummemmin kuin esiintymäbittijonotkaan – tietystikään mahdu yhden konesanan sisään, vaan näitä tarvitaan useita. Tällöin pitää bittioperaatioiden oikeellisuuden kannalta huomioida muistinumero- ja lainausbittien kulkeutuminen konesanalohkojen ylitse. Näiden käsittelyyn tarvittavan tekniikan tarkempi esittely kuitenkin sivuutetaan tässä.

Matriisin *M* rivien arvojen määrääminen voidaan kiteytetysti esittää artikkelissa [All86] annetun kaavan avulla seuraavasti. Oletuksena on, että algoritmissa perustetaan *n* kappaleesta nollia koostuva matriisin alustusrivi M_0:

[59] On mahdollista, että matriisin *M* rivin *i* vasemmanpuoleisin segmentti sisältää pelkkiä nollia.

M_i $(1 \le i \le m) = z \wedge ((z - (M_{i-1} <<1)) \oplus z)$, missä $z = M_{i-1} \vee EBJ[X[i]]$ ja <<1 tarkoittaa loogista siirtoa vasemmalle, kun oikeaan reunaan syötetään ykkönen.

Esimerkki 3.17: $X[1..6]$ = ”BDCABA”, $Y[1..7]$ = ”ABCBDAB”, Σ = {A, B, C, D}

Matriisin M sisältö eri rivien tultua käsitellyiksi. Sarake p kuvaa ykkösbittien määrää eli PYAn pituutta rivin tultua käsitellyksi. Syötejonot X ja Y on käännetty päinvastaisiksi eli X:ksi ja Y*:ksi. Dominantit täsmäykset näkyvät matriisissa lihavoituina ykkösinä, muut ykköset kuvaavat kynnysarvojen siirtymistä päivittymättöminä riviltä toiselle. Pystyviivamerkinnät* (|) *matriisin riveillä kuvaavat rivin segmenttirajoja.*

				Y^*								
	B	A	D	B	C	B	A					
i	M_i								p	X^*		
6	0	**1**	0	1	0	1	1		4	A	Dominantti 4-täsmäys (6,5) ”BCBA”	
5	**1**	0	0	**1**	0	**1**	1		4	B	Dominantti 2-täsmäys (5,2) ”AB” + Dominantti 3-täsmäys (5,4) ”BCB” + Dominantti 4-täsmäys (5,7) ”BCAB”	
4	0	**1**	0	0	1	0	**1**		3	A	Dominantti 1-täsmäys (4,1) ”A” + Dominantti 3-täsmäys (4,6) ”BCA”	
3	0	0	0	0	**1**	1	0		3	C	Dominantti 2-täsmäys (3,3) ”BC”	
2	0	0	**1**	0	0	1	0		2	D	Dominantti 2-täsmäys (2,5) ”BD”	
1	0	0	0	0	0	**1**	0		1	B	Dominantti 1-täsmäys (1,2) ”B”	
0	0	0	0	0	0	0	0		0	∅	Alustusrivi	

Rivin M_1 muodostaminen rivin M_0 ja merkin ”B” esiintymäbittien turvin; rivi M_0 kuuluu kokonaisuudessaan yhteen ja samaan segmenttiin (ei yhtään ykkösbittiä). Riviltä 1 löydetään esimerkissä dominantti 1-täsmäys paikasta (1, 2).

M_0:	0	0	0	0	0	0	0	Rivin M_0 sisältö matriisissa M
EBJ[”B”]:	1	0	0	1	0	1	0	Merkin x_1 = ”B” esiintymät Y^*:ssä
z:	1	0	0	1	0	1	0	$z := M_0 \vee EBJ$[”B”]
$t := (M_0 <<1) \vee 1$:	0	0	0	0	0	0	1	M_0:n looginen siirto vasemmalle, oikeaan reunaan syötetään 1
z':	1	0	0	1	0	0	1	$z' := z - t$
$u := z \oplus z'$:	0	0	0	0	0	1	1	ero kahdessa viimeisessä bitissä
M_1:	0	0	0	0	0	**1**	0	$M_1 := z \wedge u$

Rivin M_5 muodostaminen rivin M_4 ja merkin "B" esiintymäbittien turvin; rivi M_4 muodostuu neljästä segmentistä, joiden positiot ovat 1, 2–4, 5–6 ja 7. Riviltä löydetään luokkien 2, 3 ja 4 täsmäykset paikoista (5, 2), (5, 4) ja (5, 7).

M_4:	0 |	1	0	0 |	1	0 |	1	Rivin M_4 sisältö matriisissa M
EBJ["B"]:	1	0	0	1	0	1	0	Merkin x_5 = "B" esiintymät Y*:ssä
z:	1	1	0	1	1	1	1	$z := M_4 \vee EBJ$["B"]
$t := M_4 \ll 1$:	1	0	0	1	0	1	1	M_4:n looginen siirto vasemmalle, oikeaan reunaan syötetään 1
z':	0	1	0	0	1	0	0	$z' := z - t$
$u = z \oplus z'$:	1	0	0	1	0	1	1	ero biteissä 1, 4, 6 ja 7
M_5:	**1**	0	0 |	**1**	0 |	**1** |	1	$M_5 := z \wedge u$

3.2.5.2 PYAn kerääminen ja algoritmin analyysi

ADI-algoritmin löytämä PYAn pituus saadaan selville laskemalla yhteen matriisin *M* ylimmälle eli riville *m* tallennettujen ykkösbittien lukumäärä. Ratkaisuksi kelpaava jono voidaan löytää samankaltaisella tekniikalla kuin Wagnerin ja Fischerin WFI-algoritmissa [Wag74], mikäli matriisi *M* on tarkoitus tallentaa kokonaisuudessaan. Jos algoritmista halutaan puolestaan lineaaritilainen säilyttämällä matriisista *M* ainoastaan kaksi viimeisintä riviä, turvaudutaan samanlaiseen jononhakumenettelyyn kuin Hirschbergin lineaaritilaisessa HIL-algoritmissa [Hir75]. Liitesivuilla kohdassa 11.2.5 on nähtävissä algoritmin karkean tason pseudokoodilistaus, joka alkuperäisartikkelista puuttuu. Siinä oletetaan, että matriisi *M* säilytetään muistissa kokonaisuudessaan, jolloin ratkaisusta tulee nopeampi kuin lineaaritilaisesta, jossa matriisin kahden viimeisimmän rivin tietojen säilyttäminen riittää. Koska ADI:n löytämä ratkaisu PYAlle palautuu algoritmeissa WFI ja HIL esitettyihin, erillistä laajaa esimerkkiä ADI:n toiminnasta 17x17-kokoiselle syöteaineistolle ei erikseen esitetä.

ADI-algoritmin aika- ja tilavaativuudesta

ADI-algoritmissa muodostetaan esiprosessointivaiheessa esiintymäbittivektorit kutakin syöttöaakkoston symbolia kohti. Vektorien vaatima tila on kokonaisuudessaan suuruusluokkaa $O(\lceil n/w \rceil \sigma)$. Koska *EBJ*-vektorit kuitenkin alustetaan nollilla, monet ohjelmointiympäristöt tukevat kyseistä toimenpidettä, minkä johdosta nollauksen käytännön vaikutus suoritusaikaan jää vähäiseksi. Tämän jälkeen kutakin syöttöaakkoston merkkiä edustavaan *EBJ*-vektoriin on kuitenkin asetettava paikoilleen vielä ykkösbitit niihin indekseihin, joissa esiintyy merkin muodostama täsmäys vektorissa *Y*. Tämä vaatii vektorin *Y* selaamisen kertaalleen läpi ajassa $O(n)$. Itse prosessointivaiheessa käsitellään jokainen syötemerkkijonon *X* merkki, ja sitä kohti tutkitaan yksi matriisin rivi, jonka pituus on *n*. Koska sen sisältönä on kuitenkin vain

nollia tai ykkösiä, saadaan yksittäinen rivin positio mahdutettua yhteen bittiin. Siten matriisin yhtä riviä kohti tarvitaan vain $\lceil n/w \rceil$ konesanan käsittelyä, missä w kuvaa konesanan pituutta. Termi $\lceil n/w \rceil$ on karkeasti arvioituna edelleen suuruusluokkaa $O(n)$, joten varsinaisen prosessoinnin kustannuksen epätarkaksi ylärajaksi voitaisiin määritellä $O(mn)$ eli sama kuin naiivissa Wagnerin ja Fischerin algoritmissa. Koska yhden konesanan sisällä laskenta tapahtuu kuitenkin rinnakkaisesti, paranee ADI:n tehokkuus WFI:hin nähden käytännössä selkeästi sitä mukaa, mitä pidempiä koneen muistisanat ovat – ts. mitä suuremmaksi termi w kasvaa. Menetelmän kokonaisaikakompleksisuutta kuvaakin siten parhaiten lauseke $O(n\sigma + m\lceil n/w \rceil)$.

Algoritmi tarvitsee syötevektoreidensa lisäksi aputietorakenteikseen matriisin M sekä esiintymäbittijonot sisältävän taulukon *EBJ*. Matriisi M on kooltaan $O(m\lceil n/w \rceil)$, mikäli se halutaan tallentaa kokonaisuudessaan. Jos siitä riittävät minimaaliset kaksi riviä, sen kooksi tulee pelkästään $O(\lceil n/w \rceil)$, eli ADI voidaan saada siten toimimaan lineaarisessa muistitilassa. On kuitenkin muistettava, että taulukko *EBJ* vaatii tilan $O(\lceil n/w \rceil \sigma)$. Jos syöttöaakkosto on isokokoinen, on tällä termillä huomattava merkitys algoritmin tilantarvetta arvioitaessa.

3.2.6 Apostolicon ja Guerran II algoritmi (AG2)

3.2.6.1 Rajoittuminen dominanttien täsmäysten tarkasteluun

Samoin kuin Hsu ja Du, myös *Apostolico* ja *Guerra* esittelivät kaksi PYAn ratkaisevaa menetelmää yhdessä ja samassa artikkelissaan: yhden korkeuskäyrittäin ja yhden riveittäin prosessoivan algoritmin [Apo87]. Apostolico ja Guerra käyttävät rivi kerrallaan laskentaa suorittavasta menetelmästään nimitystä *HS 1*, joka antaa ymmärtää, että algoritmi olisi parannusehdotus Huntin ja Szymanskin algoritmiin. Käsillä olevassa työssä samainen algoritmi kantaa nimeä *AG2*.

Siinä, missä Hsu ja Du kiinnittivät huomiota HSZ:n tehottomuuteen täsmäysten luokkien määräämisessä, Apostolico ja Guerra pystyvät rajoittamaan riveittäisen tarkastelun pelkästään dominantteihin täsmäyksiin. Samalla he muuttavat rivien käsittelysuunnan: dominantteja täsmäyksiä etsitään rivillä vasemmalta oikealle toisin kuin HSZ:ssa ja HD2:ssa. AG2-algoritmin logiikan varsinainen ydin liittyy seuraavaan kynnysarvoja koskevaan havaintoon.

Lemma 3.1: *Jos riviltä i löydetään sarakkeesta j mielivaltaisen luokan k* ($1 \leq k \leq p$) *dominantti täsmäys, ei tällä sarakkeella voi myöhemmin esiintyä minkään luokan dominanttia täsmäystä, ennen kuin samainen k. korkeuskäyrä on ehtinyt siirtyä seuraavan kerran vasemmalle.*

Todistus: Sijaitkoon rivin i jälkeen seuraava dominantti k-täsmäys rivillä u sarakkeessa v ($i < u < m$), ($1 \leq v < j$). Tällöin sarakkeen j tarkastelu tulee rivin i jälkeen uudelleen ajankohtaiseksi vasta rivillä u+1, sillä tekemämme oletuksen nojalla ei riveiltä i+1..u–1 löydy dominantteja k-täsmäyksiä. Myöskään minkään muun luokan $l \neq k$ dominanttitäsmäyksiä ei näiltä riveiltä voi löytyä lemman 2.1 perusteella. Vielä pitää erikseen tutkia tilanne rivillä u. Selvästikään paikassa (u, j) ei voi sijaita dominanttia k-täsmäystä, koska sellainen jo löytyi samalta riviltä vasemmalta paikasta (u, v). Toisaalta (u, j) ei voi olla myöskään dominantti k+1 -täsmäys, sillä lemman 2.2 perusteella $Kynnys_{u-1,k} = j < Kynnys_{u,k+1}$. Täten on selvää, ettei sarakkeella j esiinny dominantteja täsmäyksiä riveillä i+1..u. □

Äskeistä tulosta tekijät hyödyntävät esittelemällä uudentyyppisen aputietorakenteen: syöttöaakkoston merkkien *dynaamisen esiintymälistan*. Sitä mukaa kun riviltä i löydetään uusi dominantti k-täsmäys indeksistä j, poistetaan j merkin x_i aktiivisesta esiintymälistasta, jonne se palautetaan vasta, kun seuraava dominantti k-täsmäys on ehditty löytää. Tämä menettely tarkoittaa toisin sanoen, että merkin x_i esiintymä vektorin Y indeksissä j on aktiivinen tarkalleen silloin, kun j ei esiinny minkään luokan kynnysarvona. Tällä tavoin taataan, että merkin aktiivisessa esiintymälistassa ei pidetä esiintymiä, joiden tarkastelu nykyhetkellä on turhaa.

AG2 käy ulommassa silmukassa läpi jokaisen rivin ylhäältä alas, ja sisemmässä silmukassa lähdetään selaamaan vasemmalta oikealle merkin x_i aktiivisia esiintymiä vektorissa Y. Jos merkin aktiivinen esiintymälista ei ole tyhjä[60], ensimmäinen sen esiintymistä j muodostaa nyt varmasti jonkin luokan dominantin täsmäyksen. Sen luokka k, jolle on selvästikin voimassa $kynnys[k–1] < j < kynnys[k]$, saadaan selville hakuoperaatiolla *dynaamisesta kynnysarvolistasta*. Tämän jälkeen päivitetään k. luokan kynnysarvoksi j, ja samalla listasta poistetaan luokan vanha kynnysarvo $h > j$, ellei se ollut ennestään määrittelemätön eli n+1. Uudesta dominantista k-täsmäyksestä (i, j) asetetaan HSZ:n ja HD2:n tapaan myös edeltäjäosoitin *linkkivektori*[k–1]:een, mutta solmun asettamista *linkkivektori*[k]:n alkuun joudutaan kuitenkin viivyttämään, ettei kehkeytyisi esimerkissä 3.12 kuvattua virhetilannetta. Kuten jo edellä todettiin, esiintymä j poistetaan nyt x_i:n aktiivisesta täsmäyslistasta, ja etsitään, mistä löytyy merkin x_i seuraava esiintymä vanhan kynnysarvoindeksin h jälkeen. Tällä tavoin päästään ohittamaan sarakkeissa j+1..h mahdollisesti esiintyvät ei-dominantit luokan k täsmäykset. Jos paikan h jälkeisen esiintymän indeksi $t < n + 1$, eli sellainen vielä löytyy riviltä i, otetaan se seuraavaksi käsittelyyn. Ennen sen käsittelyä palautetaan vielä kynnysarvolistasta poistettu indeksi h merkin $Y[h]$ aktiiviseen esiintymälistaan, sillä se voi mahdollisesti jo seuraavasta rivistä lähtien muodostaa uuden, luokkaan $l > k$ kuuluvan dominantin täsmäyksen. Sisempi silmukka lopettaa toimintansa, kun x_i:n aktiivisesta esiintymälistasta luetaan lopulta pysäytysalkiota edustava arvo $n + 1$.

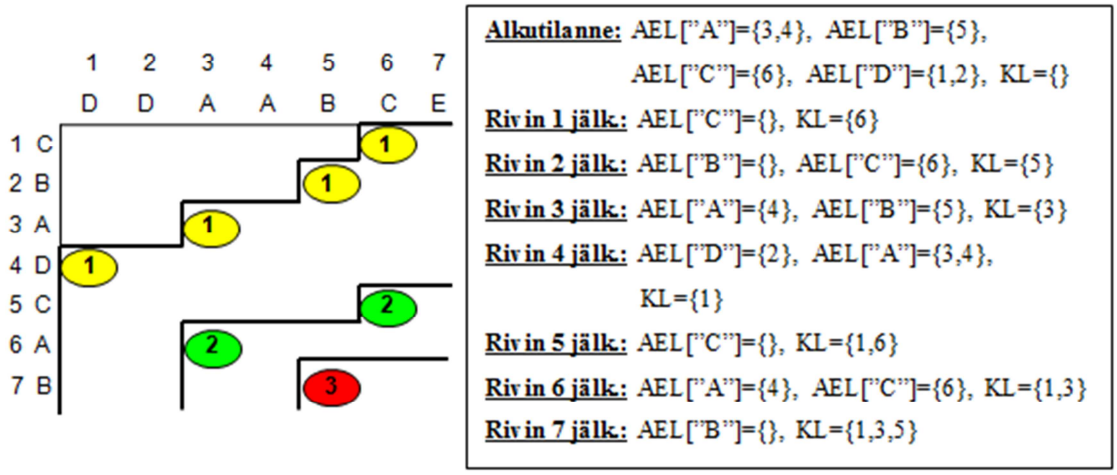
[60] Aktiiviset esiintymälistat päättyvät pseudotäsmäykseen $n + 1$.

3.2.6.2 Analyysi

AG2-algoritmin laskentavaihe päättyy, kun *X*:n viimeisenkin merkin aktiivinen esiintymälista on selattu loppuun asti. Algoritmi palauttaa PYAlle saman ratkaisun kuin HSZ ja HD2, joten erillistä isoa esimerkkiä menetelmän toiminnasta koko syötteelle ei esitetä. Sen sijaan seuraavassa annetaan näyte siitä, miten aktiivisten esiintymälistojen sisältö muuttuu algoritmin suorituksen edetessä. Liitteen kohdassa 11.2.6 on listattuna algoritmin pseudokoodi ilman PYA-jonon keräämiseksi tarvittavia toimenpiteitä, koska ne on algoritmin kuvauksessa esitetty virheellisesti[61].

Esimerkki 3.18: *X*[1..7] = ”CBADCAB”, *Y*[1..7] = ”DDAABCE”, *p* = 3, *PYA* = ”ACB”.

Aktiivisen esiintymä- ja dynaamisen kynnysarvolistojen kehittyminen AG2-algoritmin suorituksen edetessä. Aktiivisia esiintymälistoja ilmaisee esimerkissä tunnus AEL, kynnysarvolistaa KL.



Algoritmin aika- ja tilavaativuus

AG2:n esiprosessointivaiheessa perustetaan kutakin *X*:ssä esiintyvää merkkiä kohti dynaaminen esiintymälista merkin sijaintipaikoista vektorissa *Y*. Tästä operaatiosta selviydytään ajassa $O(n)$, koska Σ oletetaan tunnetuksi. Dynaamisen kynnysarvovektorin perustamiskustannus on vakioaikainen, sillä sinne tarvitsee asettaa vain reuna-alkiot 0 ja n+1.

Algoritmin varsinainen laskenta tapahtuu kahden sisäkkäisen silmukan *S1* ja *S2* sisällä. Silmukassa *S1* tutkitaan vuoron perään jokainen rivi eli syötevektorin *X i*. merkki. Sisemmässä silmukassa *S2* tutkitaan puolestaan ainoastaan x_i:n aktiiviseen esiintymälistaan kuuluvia *Y*-indeksejä. Näistäkin jäävät tarkastelun ulkopuolelle vielä ei-dominantin täsmäyksen muodostavat pisteet, joten *S2*:ssa suoritetaan tarkalleen niin

[61] Juuri esimerkissä 3.12 kuvatun kaltainen virhetilanne voi syntyä noudattamalla AG2:n pseudokoodia, jossa tarvetta päivitysten viivyttämiselle ei ole huomioitu. Tutkimusryhmämme esittämä teknisesti korjattu ratkaisu, joka soveltuu käytettäväksi myös AG2:ssa, on kuvattu *Kuon* ja *Crossin* algoritmin yhteydessä aliluvussa 3.2.7.

monta kierrosta kuin ulomman silmukan laskurin arvon mukaisella rivillä *i* on dominantteja täsmäyksiä. Niiden kohdalla joudutaan suorittamaan hakuoperaatioita sekä uuden että aikaisemmin löydetyn *k*-täsmäyksen muodostaneen merkin aktiivisesta esiintymälistasta ja lisäksi vielä kynnysarvolistasta. Jos nämä tietorakenteet esitetään esimerkiksi käyttämällä *2-3-* tai *AVL-puita*, on dominanttien täsmäysten etsinnän kokonaiskustannus suuruusluokkaa $O(d \log n)$ [Meh84], jota voidaan algoritmin tekijöiden mukaan puristaa vieläkin tiukemmaksi suuruusluokkaan $O(d \log\log n)$ asti käyttämällä *van Emde Boasin* esittämää menetelmää kokonaislukujen käsittelemiseksi [Emd77], tai vaihtoehtoisesti suuruusluokkaan $O(m \log n + d \log(2mn/d))$ käyttämällä aputietorakenteena *sormipuita* (engl. *finger-tree*) [Meh84]. Koska hakuoperaatioita lukuun ottamatta silmukoiden kaikki muut lauseet ovat vakioaikaisia, saadaan algoritmin aikakompleksisuudeksi $O(n + d \log n)$ tai $O(n + d \log\log n)$ riippuen listojen käsittelyyn valitusta tietorakenteesta. Menetelmän pitäisi teoreettisesti toimia hyvin, kun ongelmassa on verrattain vähän dominantteja täsmäyksiä.

Algoritmi tarvitsee ei-vakiomääräistä tilaa syötevektoreille, esiintymälistoille ja linkkivektorille. Ensin mainitut vievät muistia yhteensä $O(n)$, kun taas linkkivektorin vaatima muistitila riippuu dominanttien täsmäysten kokonaismäärästä *d*. Siten koko algoritmin yhteenlaskettu tilantarve on $O(n + d)$.

3.2.7 Kuon ja Crossin algoritmi (KCR)

3.2.7.1 Pyrkimyksenä HSZ-algoritmin kirjanpidon vähentäminen

Kuo ja *Cross* esittelivät vuonna 1989 [Kuo89] parannetun versionsa Huntin ja Szymanskin algoritmista. Heidän algoritmistaan käytetään tässä työssä lyhennettä *KCR*. Artikkelin lähdeluettelon perusteella tekijät eivät ole vielä huomioineet HSZ:n jälkeen vuonna 1987 julkaistua AG2-algoritmia, jonka lähestymistapa PYAn määräämiseksi on perusajatukseltaan identtinen KCR:n kanssa. Teoreettisesti AG2 on jopa KCR:ää kehittyneempi menetelmä, sillä se rajoittaa tarkastelun yksinomaan rivien dominantteihin täsmäyksiin, kun taas KCR:ssä käydään edelleen kaikki täsmäykset läpi menetelmän esikuvana toimivan HSZ:n tapaan.

Oleellisena parannuksena HSZ:n laskentamalliin nähden Kuo ja Cross korostavat rivien täsmäysten tarkastelua päinvastaisessa järjestyksessä eli vasemmalta oikealle. Koska PYA-ongelman teoreettisten ominaisuuksien perusteella on selvää, että tarkasteltavan rivin *k*-täsmäyksistä ainoastaan *vasemmanpuoleisin* voi olla dominantti, on turhaa päivittää HSZ:n tapaan useammin kuin kertaalleen luokan *k* kynnysarvoa yhden rivin käsittelyn aikana. Kuon ja Crossin algoritmissa tämä HSZ:n kiusallinen piirre onkin korjattu.

KCR:ssä muodostetaan *X*:n merkkien esiintymälistat niiden sijaintipaikoista *Y*:ssä *kasvavaan suuruusjärjestykseen*, eli päinvastoin kuin HSZ:ssa. Kynnysarvovektori sen

sijaan alustetaan samoin kummassakin menetelmässä. Rivin käsittelyssä tarvitaan avuksi muuttujaa *tilap*, johon otetaan talteen *kynnysarvovektorin viimeksi päivitetyn indeksipaikan k päivitystä edeltänyt arvo nykyisen rivin käsittelyn aikana*. Muuttuja alustetaan nollaksi aina rivin vaihtuessa algoritmin ulommassa silmukassa.

Algoritmin sisemmässä silmukassa käydään läpi yksittäisen rivin kaikki täsmäykset. Rivillä i ($1 \le i \le m$) otetaan kukin merkin x_i täsmäyskohdista j vektorissa Y vuoron perään tutkittavaksi. Jos nyt $j \le tilap$, täsmäys (i, j) kuuluu samaan luokkaan kuin rivin edellinenkin täsmäys. Silloin (i, j) on välttämättä ei-dominantti, ja se ei siten aiheuta tarvetta lisätoimenpiteille. Muussa tapauksessa algoritmi selvittää, mihin luokkaan indeksipaikassa j oleva täsmäys kuuluu. Tämä tapahtuu etsimällä lineaarihaulla kynnysarvovektorista pienin sellainen indeksi $k \in [1..p+1]$, jolle on voimassa $j \le kynnys[k]$. Paikassa $kynnys[k]$ oleva arvo otetaan talteen muuttujan *tilap* uudeksi arvoksi, ennen kuin se päivitetään arvoon j. Täsmäystä (i, j) varten generoidaan uusi linkkisolmu PYA-jonojen ylläpitoa varten, ja solmusta asetetaan isäosoitin $linkkivektori[k–1]$:n alkusolmuun. Sen sijaan linkkisolmua itseään ei viedä vielä tässä vaiheessa $linkkivektori[k]$:hon, vaan sitä vastoin asetetaan apumuuttujan *edellpäiv* arvoksi k muistuttamaan, että luokan k linkkivektori pitää myöhemmin päivittää[62].

Kun täsmäys on käsitelty edellä kuvatulla tavalla, siirrytään rivin seuraavaan täsmäykseen. Kannattaa huomioida, ettei rivin myöhempien täsmäysten luokkaa selvitettäessä tarvitse enää selata kynnysarvovektoria alusta alkaen, vaan voidaan jatkaa eteenpäin siitä indeksiarvosta k, johon edellinen haku pysähtyi, sillä siirryttäessä rivillä oikealle täsmäysten luokat eivät voi pienentyä. Kun riviltä i on ehditty löytää järjestyksessä seuraava kirjattava täsmäys – olkoon sen luokka $u > k$ – ja asettaa sen isälinkki $linkkivektori[u–1]$:n alkusolmuun, asetetaan linkkivektoriin yhä päivitystä odottanut k-täsmäys lopulta $linkkivektori[k]$:n alkuun. Syy linkkivektorin päivityksen myöhäisyydelle ilmenee tarkastelemalla esimerkkiä 3.12. Esimerkiksi luokan 2 täsmäys paikassa (2, 4) kirjautuisi virheellisesti samalta riviltä löytyneen 1-täsmäyksen (2, 1) perään, jos (2, 1) olisi viety $linkkivektori[1]$:n alkuun heti sen löytämishetkellä. Nyt kuitenkin viivytetään (2, 1):n kirjaamista $linkkivektori[1]$:een siihen saakka, kunnes seuraavan samalta riviltä löytyneen toimenpiteitä aiheuttavan täsmäyksen, eli esimerkissä (2, 4):n, isälinkki on ehditty asettaa ylemmällä rivillä sijaitsevaan edeltäjäänsä eli täsmäykseen (1, 2). Tällä tavoin menettelemällä eivät PYAn löytämiseksi ylläpidettävät täsmäysketjut pääse sekoamaan[63]. Rivin viimeinen kynnysarvovektoriin päivityksen aiheuttanut täsmäys asetetaan linkkivektoriin vasta sisemmästä silmukasta poistumisen jälkeen, eli juuri ennen uuden rivin ottamista tarkasteluun.

---

[62] Jos muuttujan *edellpäiv* arvo on 0, ei linkkivektorin päivityksiä ole odottamassa. Nolla sopii siten oivallisesti kyseisen muuttujan alkuarvoksi uudelle riville siirryttäessä.

[63] Kuvattua päivitysongelmaa esiintyy tosin vain, jos samalla rivillä joudutaan päivittämään kahden perättäisen luokan k ja $k+1$ kynnysarvoja.

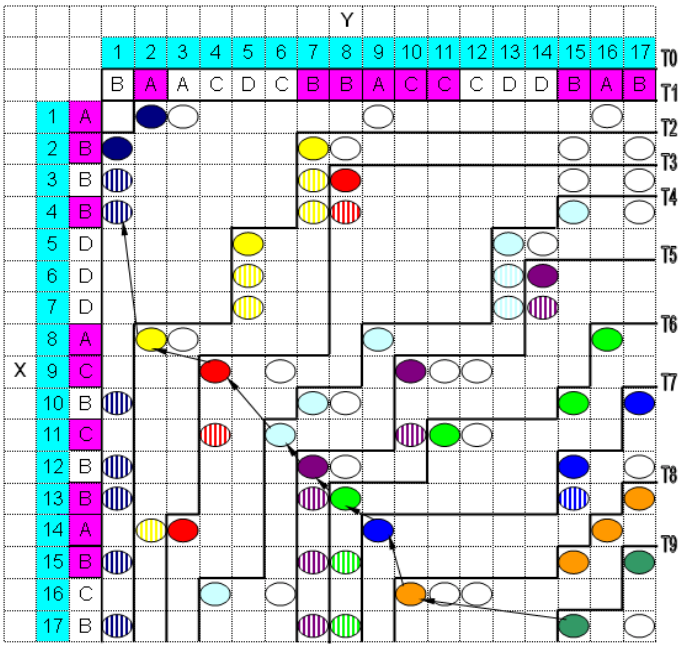
3.2.7.2 Algoritmiin liittyviä huomioita ja esimerkki

Liitesivuilla kohdassa 11.2.7 esitetään KCR:n pseudokoodi niiltä osin kuin se poikkeaa oleellisesti Huntin ja Szymanskin algoritmista. Tekijät eivät itse ole esityksessään kiinnittäneet huomiota PYAn ratkaisupolkujen ylläpitoon vaan keskittyvät ainoastaan PYAn pituuden ratkaisemiseen. Siten jonojen ylläpidon osuus on oma ehdotukseni algoritmin täydentämiseksi. Samaa päivitysten kirjaamistekniikkaa voitaisiin soveltaa myös edellä esitettyyn AG2-algoritmiin.

Siitä huolimatta, että KCR pystyy HSZ:sta poiketen tunnistamaan huomattavan osan rivin ei-dominanteista täsmäyksistä, menetelmä ei rajaa kaikkia ei-dominantteja täsmäyksiä jatkotarkastelun ulkopuolelle. KCR nimittäin kirjaa muistiin kultakin riviltä jokaisen luokan C_k ensimmäisen täsmäyksen riippumatta siitä, onko se dominantti vai ei, sillä algoritmi ei erikseen enää testaa paikassa j olevan täsmäyksen luokan selvittyä, esiintyykö j jo ennestään asianomaisen luokan kynnysarvona. Graafisesti tarkasteltuna tämä tarkoittaa, että jokaisen korkeuskäyrän pystysuorilla osuuksilla sijaitsevat ei-dominantitkin täsmäykset rekisteröidään linkkivektoriin, ja samalla kynnysarvovektoriin voidaan päivittää eri riveillä sama arvo toistuvasti. Tästä seuraa, että PYA-jonoon voi päätyä myös ei-dominantteja täsmäyksiä.

Seuraavassa esimerkissä on kuvattu, millaisen ratkaisun KCR löytäisi esimerkin 3.1 syötejonoille. Siitä myös ilmenee, minkä verran algoritmi kirjaa ei-dominantteja täsmäyksiä linkkivektoriin. Algoritmi valitsee PYAlle aina mahdollisimman alhaalla sijaitsevan ratkaisupolun, ja siihen kuuluu oheisessa esimerkissä yksi ei-dominantti täsmäys paikassa (4, 1). Algoritmi löytää myös kaikki k-täsmäykset, jotka esiintyvät rivin ensimmäisen dominantin k-täsmäyksen jälkeen. Näitä ei kuitenkaan kirjata muistiin linkki- eikä kynnysarvovektoriin toisin kuin HSZ:ssa.

Esimerkki 3.19: *KCR:n löytämä PYAn ratkaisupolku esimerkin 3.1 syötejonoille. Pystyraidoitetut soikiot ilmaisevat luokkansa ainoita ja samalla ei-dominantteja täsmäyksiä tarkasteltavalla rivillä. Muut ei-dominantit täsmäykset on merkitty kuvaan valkoisin ovaalein.*



3.2.7.3 KCR-algoritmin analyysi

Algoritmin esiprosessointivaiheessa joudutaan perustamaan X:n merkkien esiintymälistat kuten myös kynnysarvo- ja linkkivektori sekä alustamaan ne. Tähän tarvitaan aikaa yhteensä $O(n)$. Laskentavaiheessa silmukka *S1* käy läpi kaikki rivit ja sisempi silmukka *S2* kunkin yksittäisen rivin kaikki täsmäykset. Siten *S1* ja *S2* käsittelevät yhdessä syötejonojen kaikki täsmäykset, mihin kuluu aikaa $O(r)$. Jokaista riviä kohti joudutaan kynnysarvovektori selaamaan pahimmassa tapauksessa indeksistä 1 indeksiin p asti. Koska rivejä on m kappaletta, on kynnysarvovektorin selaamisen kustannuksen yläraja $O(pm)$. PYAn palauttaminen tapahtuu ajassa $O(m)$. Siten koko algoritmin aikakompleksisuutta rajoittaa ylhäältä lauseke $O(r + n + pm)$.

KCR vaatii ei-vakiotilaista muistia syötevektoreiden lisäksi vain esiintymälistoille, kynnysarvovektorille ja linkkivektorille solmuineen. Viimeksi mainittua lukuun ottamatta kaikki saadaan mahtumaan tilaan $O(n)$. Sen sijaan linkkivektoriin voi HSZ:n tavoin päätyä pahimmassa tapauksessa $O(r)$ täsmäystä, joten algoritmin tilakompleksisuus on suuruusluokkaa $O(n + r)$. Jos korkeuskäyrien pystysuorien osuuksien ei-dominantteja täsmäyksiä ei kirjattaisi, algoritmi toimisi tilassa $O(n + d)$.

3.2.8 Rickin II algoritmi (RI2)

3.2.8.1 Uutuutena rivillä käsiteltävien kynnysarvoluokkien rajaaminen

Viimeisenä rivi kerrallaan prosessoivista PYAn ratkaisevista menetelmistä esitellään *Rickin II algoritmi* vuodelta 1994 [Ric94]. Tekijä käyttää kyseisestä algoritmista nimitystä *Algoritmi 4*, ja tässä työssä siitä käytetään lyhennettä *RI2*.

RI2-algoritmin lähestymistapa PYAn ratkaisemiseksi eroaa varsin paljon samassa artikkelissa esitellystä, mutta tässä työssä prosessointitapansa vuoksi vasta aliluvussa 3.4.1 esiteltävästä RI1-algoritmista. Kun RI1:ssä etsitään dominantteja täsmäyksiä vuoron perään riveiltä ja sarakkeilta, RI2 on puhtaasti riveittäin laskentaa suorittava menetelmä. Siten on ilmeistä, että siinä ei myöskään ole käyttöä sarakesuuntaiselle lähiesiintymätaulukolle kuten RI1:ssä. Algoritmin erikoisuutena on *dynaamistyyppinen kynnysarvovektori*, joka kuitenkin esitetään staattisena. Vektorissa on m+1 indeksipaikkaa kuten tavanomaisissakin kynnysarvovektoreissa, mutta *kynnysarvon* lisäksi siinä esiintyy neljä ylimääräistä tietokenttää, jotka ovat *edellinen*, *seuraava*, *uusiarvo* ja *rivi*. Ennen kuin lähdetään tarkemmin kuvaamaan, mihin näitä lisäkenttiä käytetään, tarkastellaan Rickin esittämää lemmaa, jonka perusteella saadaan selville, *minkä eri täsmäysluokkien kynnysarvot voivat päivittyä rivillä i* ($1 \le i \le m$).

Lemma 3.2: *Olkoot X ja Y kaksi syötemerkkijonoa, joiden pituudet ovat m ja n. Oletetaan, että* i_1 *ja* i_2 *ovat kaksi vektorin X indeksiä siten, että* $i_1 < i_2$, $x_{i_1} = x_{i_2}$ *ja* ($\forall i$*:* $i_1 < i < i_2 \mid x_i \neq x_{i_1}$). *Tällöin on voimassa*

$$kynnys_{i_2-1,k} = kynnys_{i_1-1,k} \Rightarrow kynnys_{i_2,k+1} = kynnys_{i_2-1,k+1}.$$

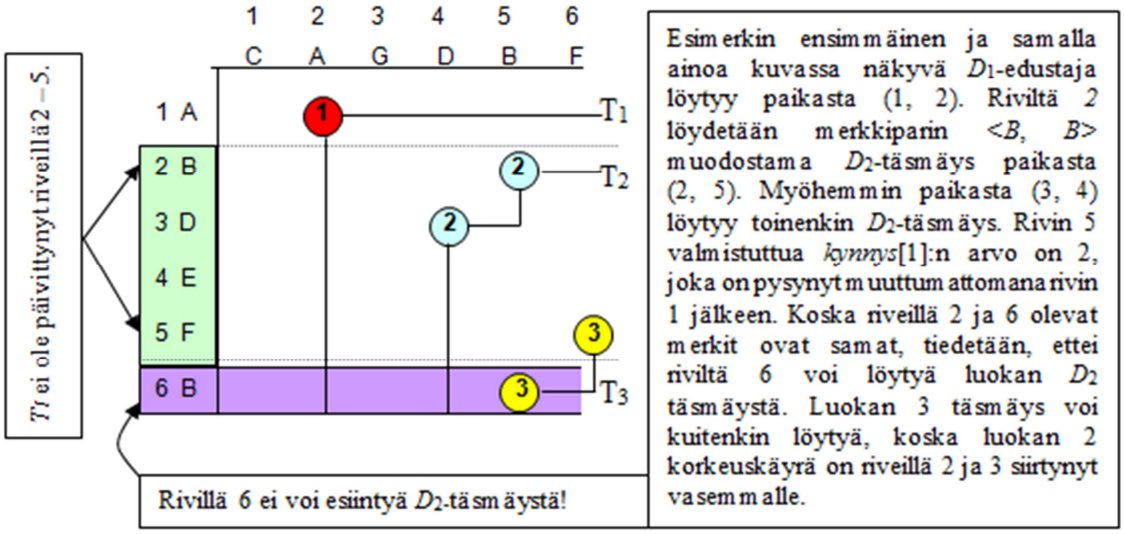
Lemman 3.2 mukaan luokan k + 1 kynnysarvo *ei voi päivittyä* rivillä i_2, ellei luokan k kynnysarvo ole pienentynyt kertaakaan tätä edeltäneillä riveillä i_1.. i_2–1, kun näistä ylin eli i_1 on ainoa, jolla esiintyy sama symboli kuin rivillä i_2. Todistetaan lemman sisältämä väite seuraavassa oikeaksi.

Todistus: Oletetaan, että luokan k kynnysarvo on u rivin i_1–1 tultua käsitellyksi, ja riviltä i_1 löydetään dominantti k+1 -täsmäys sarakkeesta $v > u$. Selvästikin v on nyt ensimmäinen merkin x_{i_1} täsmäys indeksin u jälkeen dominantin täsmäyksen määritelmän perusteella, mikä merkitsee, ettei x_{i_1} täsmää yhdenkään Y:n symbolin kanssa sarakkeilla u+1..v–1. Kun tullaan myöhemmin riville i_2, kohdataan sama merkki kuin rivillä i_1. Ellei rivin i_1–1 jälkeen luokan k kynnysarvo ole muuttunut miksikään – eli graafisesti tulkittuna ei kyseisen luokan korkeuskäyrä ole siirtynyt rivin jälkeen vasemmalle – on merkin x_{i_2} lähin esiintymä *kynnys*[k]:n jälkeen samassa paikassa kuin jo rivillä i_1, joten uutta k+1 -täsmäystä ei riviltä i_2 voi löytyä. Jos puolestaan riviltä i_1 ei

löytynyt dominanttia *k*-täsmäystä, ei sellaista voi edellä mainitusta syystä esiintyä myöskään rivillä i_2. □

Lemmasta 3.2 seuraa, ettei riviltä *i* kannata etsiä sellaisten luokkien *k* dominantteja täsmäyksiä, joiden edeltäjäluokkien *k*–1 kynnysarvot eivät ole päivittyneet rivin *t*–1 jälkeen, missä *t* on lähin *i*:tä edeltävä rivi, jossa esiintyy symboli x_i. Seuraavassa esitetään pieni esimerkki, joka selkeyttänee lemman 3.2 sisältöä.

Esimerkki 3.20: *X*[1..*m*] =”ABDEFB…” *Y*[1..*n*] = ”CAGDBF…”.

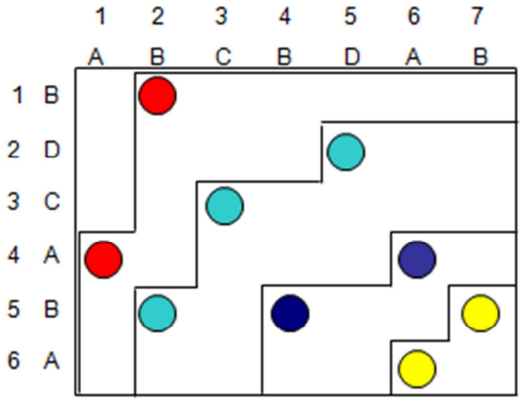


Algoritmi pyrkiikin siten rajoittamaan rivillä *i* kynnysarvovektorin tutkimisen ainoastaan niihin sen positioihin, jotka voivat päivittyä rivillä *i*. Jotta tämä olisi mahdollista, joudutaan pitämään kirjaa siitä, millä rivillä luokan *k* kynnysarvoa on viimeksi päivitetty. Tämä tieto säilötään kynnysarvovektorin kenttään nimeltä *rivi*. Lisäksi pitää muistaa, millä rivillä tarkasteltavana oleva merkki on esiintynyt aikaisemmin. Tähän tarkoitukseen perustetaan vektori *viim_es*, jonka pituus on σ.

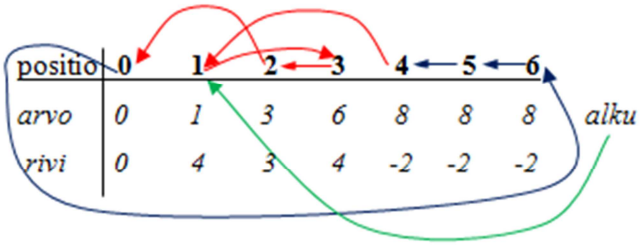
Kynnysarvovektorin luokat asetetaan dynaamiseen järjestykseen sen mukaisesti, missä järjestyksessä sen positioita on päivitetty. Apumuuttujaan *alku* tallennetaan sen luokan numero, jonka kynnysarvoa on päivitetty viimeksi. Kynnysarvoista rakennetaan vektoriin *kahteen suuntaan linkitetty rengaslista*. Uusimmasta päivitetystä luokasta päästään kenttää *seuraava* pitkin etenemään tätä lähinnä aikaisemmin päivitettyyn positioon. *Edeltäjä*-kenttiä tarvitaan ainoastaan vektorin dynaamisuuden ylläpitoa varten. Uuden löydetyn dominantin *k*-täsmäyksen indeksiä *j* ei saada tallentaa suoraan varsinaiseen kynnysarvolle varattuun kenttään, vaan se sijoitetaan tilapäisesti kenttään *uusiarvo*. Tämä on tarpeen, jotta rivin *i* täsmäysten indeksiarvoja voitaisiin verrata edellisen rivin kynnysarvoihin, eikä virheellisesti mahdollisesti vasta rivin *i* käsittelyn aikana löytyneisiin. Sama on tarpeen päivitysten viivyttämistä varten *linkkivektorin* päivittämiseksi, sillä kynnysarvojen tutkimisjärjestys rivillä *i* määräytyy yksinomaan

niiden edeltäjäluokkien käännetyn päivitysjärjestyksen perusteella. Jos vaikkapa rivin *i*–1 aikana päivitettiin luokkien 4, 1 ja 3 kynnysarvot mainitussa järjestyksessä, etsitään riviltä *i* ensinnä luokan 4, ja tämän perään luokkien 2 ja 5 dominantteja täsmäyksiä ennen muiden mahdollisten luokkien tarkastelua. Tiedot riviltä *i* löydetyistä uusista dominanteista täsmäyksistä viedään löytymisjärjestyksessä odottamaan vektoriin *muutokset*, josta ne asetetaan oikeille paikoilleen dynaamiseen kynnysarvovektoriin ennen seuraavalle riville siirtymistä. Seuraavassa vielä esimerkki, miten seuraajalinkit ja rivinumerot kulkevat mukana RI2-algoritmin kynnysarvovektorissa.

Esimerkki 3.21: $X[1..6]$ = ”BDCABA”, $Y[1..7]$ = ”ABCBDAB”, $\sigma = 4$, $p = 4$.



Kynnysarvovektorin tila 4. rivin käsittelyn jälkeen:



Tultaessa neljännelle riville *kynnys*[1] = 2 ja *kynnys*[2] = 3. Päivitysjärjestys viimeisimmästä päivityksestä alkaen on *kynnys*[2] (päivitys rivillä 3) -> *kynnys*[1] (päivitys rivillä 1). Riviltä 4 löydetään uusi 3-täsmäys (4, 6), ja sen jälkeen vielä ensimmäinen 1-täsmäys indeksipaikasta (4, 1). Näiden johdosta kynnysarvovektorin positioiden päivitysjärjestys muuttuu rivin 4 jälkeen muotoon 1->3->2. Alempaan kuvaan on piirretty kaikkien luokkien rivi- ja sarakearvot sekä seuraajalinkit. Punaisella merkityt linkit on päivitetty rivin 4 kohdalla. Kynnysarvovektorin paikkojen 0, 4, 5 ja 6 arvot eivät ole muuttuneet kertaakaan.

Algoritmi pitää kirjaa PYAn pituudesta muuttujassa *max*. Kun syötevektorin *X* viimeisenkin rivin dominantit täsmäykset on saatu kirjattua, tulostetaan muuttujan *max* arvo ja palautetaan PYA-jono. Aivan kuten RI1-algoritmin yhteydessä aliluvussa 3.4.1, nytkään ei Rick esitä mitään pseudokoodia PYAn ratkaisupolkujen yläpitoa varten, mutta se voitaisiin toteuttaa samaan tapaan kuin KCR-algoritmin yhteydessä. Tähän työhön on sisällytetty RI2:n pseudokoodi liitesivujen kohdassa 11.2.8.

3.2.8.2 Algoritmin analyysi

RI2:n esiprosessointivaiheessa joudutaan perustamaan lähiesiintymätaulukko vektorin *X* sijaintipaikoista *Y*:ssä. Tähän työhön tarvitaan aikaa $O(n\sigma)$. Lisäksi joudutaan alustamaan vektorit *kynnys* ja *viim_es*, joiden pituudet ovat $m+1$ ja σ. Lähiesiintymätaulukon perustamiskustannus peittää kuitenkin alleen niiden

alustamistyön kustannuksen. Pituudeltaan *m* olevaa vektoria *muutokset* ja täsmäyspolkujen ylläpitämiseen tarvittavaa linkkivektoria ei tarvitse alustaa. Siten esiprosessoinnin kustannus on $O(n\sigma)$.

Varsinainen laskentavaihe tapahtuu silmukoissa *S1 – S3*. Niistä toisiinsa nähden peräkkäin sijoittuneet *S2* ja *S3* ovat *S1*:n sisällä. Ulompaa silmukkaa *S1* suoritetaan *m* kierrosta eli kutakin riviä kohti, ja silmukan lauseista muut kuin silmukat *S2* ja *S3* ovat vakioaikaisia. Silmukkaa *S2* suoritetaan yhtä riviä kohti korkeintaan niin monta kertaa kuin PYAlle on kertynyt pituutta tarkasteluhetkeen mennessä, sillä siinä käydään läpi kynnysarvovektorista tarkalleen ne positiot, joiden arvot voivat päivittyä tutkittavalla rivillä. Silmukkaa *S3* suoritetaan korkeintaan yhtä monta kierrosta riviä kohti kuin *S2*:ta, sillä siinä vain korjataan kynnysarvovektori kuntoon päivittyneiden indeksien osalta. Siten kaikkien silmukoiden kokonaiskustannus on suuruusluokkaa $O(pm)$. Toisaalta, silmukoiden suoritusten aikana kutakin kynnysarvovektoriin tallennettua arvoa — toisin sanoen dominanttia täsmäystä — voidaan joutua tarkastelemaan korkeintaan kertaalleen jokaista syöttöaakkoston merkkiä kohti, joten myös $O(d\sigma)$ kelpaisi silmukoiden kokonaistyömäärää kuvaavaksi ylärajatermiksi. Loppuprosessointi, PYAn palauttaminen, vie aikaa ainoastaan $O(p)$, joten koko algoritmin suorituskustannus on $O(n\sigma + Min\{pm, d\sigma\})$.

Algoritmissa tarvitaan ei-vakiotilainen määrä muistia syöte-, kynnysarvo-, muutos- ja merkkien viimeisiä esiintymäkohtia kuvaaville vektoreille, linkkivektorille sekä lähiesiintymätaulukolle. Näistä kaksi viimeksi mainittua peittävät alleen muiden tietorakenteiden tilantarpeen. Lähiesiintymätaulukkoa varten tarvitaan muistia $O(n\sigma)$ ja linkkivektoria varten $O(d)$. RI2-algoritmi toimii siten muistitilassa $O(n\sigma + d)$.

3.3 Syötteiden loppuliitteitä laajentavat algoritmit

Tässä aliluvussa tarkastellaan kahta PYA-algoritmia, joiden toiminta ei perustu sen kummemmin korkeuskäyrä kuin rivikään kerrallaan tapahtuvaan täsmäysten etsintään, vaan *syötejonojen loppuliitteiden askeltavaan laajentamiseen*. Kumpikin menetelmä esiteltiin 1980-luvulla, ja niistä uudempi on lisäksi lineaaritilainen.

3.3.1 Nakatsun, Kambayashin ja Yajiman algoritmi (NKY)

3.3.1.1 Uusi laskennan etenemismalli: diagonaali kerrallaan

Narao Nakatsu, *Yahiko Kambayashi* ja *Shuzo Yajima* julkaisivat vuonna 1982 PYA-algoritmin, joka poikkeaa toimintaperiaatteeltaan voimakkaasti kaikista aliluvuissa 3.1

ja 3.2 esitellyistä algoritmeista [NKY82]. Tekijät ovat nimenneet algoritminsa nimellä $O(n(m-p))$ sen asymptoottista suoritusaikaa kuvaavan lausekkeen mukaan, ja tässä työssä se tunnetaan lyhenteellä *NKY*. Kaikkien edellä esiteltyjen algoritmien tapaan myös NKY:n laskentavaiheesta huolehtii kaksi sisäkkäistä silmukkaa. Kun edellisissä menetelmissä yhden ulomman silmukan kierroksen aikana käsiteltiin joko yksi *korkeuskäyrä* tai lyhyemmän syötevektorin yksi *rivi*, NKY:ssä laskenta eteneekin *diagonaali kerrallaan* syötejonojen *lopusta alkuun päin*.

Algoritmin teoreettisena perustana ovat kolme *syötejonojen loppuliitteitä* koskevaa lemmaa 3.3 – 3.5, jotka muistuttavat melkoisesti kynnysarvoja käsitelleitä lemmoja 2.1 – 2.3. Niissä esiintyvällä merkinnällä $L_i(k)$ tarkoitetaan *suurinta sellaista syötejonon Y indeksiä j, josta alkava loppuliite Y*[*j*..*n*] *muodostaa k:n mittaisen PYAn vektorin X loppuliitteen X*[*i*..*m*] *kanssa*[64]. Toisin sanoen, *Y*[*j*..*n*] on *lyhin* mainitun vaatimuksen täyttävä *Y*:n loppuliite[65]. Ellei tällaista *j*:tä ole olemassa, määritellään $L_i(k)$:n arvoksi 0. Lemmojen käsittelyn helpottamiseksi voidaan määritellä, että mille tahansa *i*:n arvolle $L_i(m+1) = 0$, koska PYAn pituus ei selvästikään voi ylittää *X*:n pituutta *m*. Samoin on triviaalisti voimassa $L_i(0) = n + 1$ *i*:n arvosta riippumatta, sillä yhtään merkkiä *Y*:n lopusta ei tarvita tyhjän PYAn muodostamiseksi loppuliitteen *X*[*i*..*m*] kanssa[66]. Lemmat esitellään ja todistetaan oikeiksi seuraavassa, jotta lukijan olisi helpompi hahmottaa algoritmin toimintaa ja vakuuttua sen oikeellisuudesta. Todistukset löytyvät lähes samanlaisina myös tekijöiden alkuperäisartikkelista.

Lemma 3.3: $\forall i \in [1..m], \exists z \mid z = MIN_u \in [1..m+1], u \le k \le m+1$:

$$L_i(u) = 0 \wedge L_i(0) > L_i(1) > ... > L_i(u-1) > L_i(u) \wedge L_i(k) = 0.$$[67]

Lemman 3.3 asiasisältönä on, että mistä tahansa *X*:n loppuliitteen alkamiskohdasta *i* alkaen lasketut $L_i(k)$-arvot ovat *aidosti väheneviä*, kunnes *k* saavuttaa arvon *u*, jonka mittaista yhteistä alijonoa *X*[*i*..*m*] ei enää pysty muodostamaan *Y*:n kanssa.

Todistus: Lemman paikkansapitävyys on helppo todeta suoraan merkinnän $L_i(k)$ määritelmän perusteella. Oletetaan ensiksi, että $L_i(k) = t > 0$. Tällöin vektorin *Y* indeksissä *t* oleva merkki *Y*[*t*] on loppuliitteiden *X*[*i*..*m*] ja *Y*[*t*..*n*] *k*:n mittaiseen PYAan *ensimmäinen* syötejonosta *Y* otettava merkki. Ellei kyseisessä paikassa olevaa merkkiä tarvittaisi, siitä seuraisi $L_i(k)$:n minimaalisuuden nojalla, että $L_i(k) \neq t$, jolloin päädytään

[64] NKY-algoritmia käsittelevissä artikkeleissa (mm. [NKY82], [BHV03]) käytetään merkinnän $L_i(k)$ asemesta merkintää $L_i(j)$. Koska sulkulausekkeen sisällä oleva symboli *j* tarkoittaa tässä kuitenkin loppuliitteiden *X*[*i*..*m*] ja *Y*[$L_i(j)$..*n*] yhteisen alijonon pituutta eikä yksittäistä *Y*-vektorin indeksiä, jota *j* normaalisti ilmaisee, on päädytty valitsemaan tämän työn muiden osien kanssa yhdenmukaisempi merkintätapa.

[65] Arvo $L_i(k)$ voidaan siten tulkita luokan *j* kynnysarvon sijaintikohdaksi rivillä *i* käännettyjen syötejonojen *X** ja *Y** alkuliitteillä *X**[*m*..*i*] ja *Y**[*n*..$L_i(k)$].

[66] Näitä lisämäärittelyjä ei ole tehty alkuperäisartikkelissa [NKY82], vaan siinä oletetaan lemmoissa käsiteltävän aina *nollasta eroavia* $L_i(k)$-arvoja, jotka ovat samalla $\le n$.

[67] Kvanttorimerkintä MIN_u tarkoittaa pienintä *u*:ta, joka täyttää sille asetetut ehdot [Rai94].

ristiriitaan $L_i(k)$:n määritelmän kanssa. Siispä $Y[t..n]$ on *lyhin* Y:n loppuliite, joka muodostaa j:n mittaisen PYAn $X[i..m]$:n kanssa. Tästä seuraa väistämättä, että j:tä pidemmän PYAn muodostamiseksi samaisen loppuliitteen $X[i..m]$ kanssa Y:stä pitää valita ainakin yksi lisämerkki ennen indeksiä t, eli siten $L_i(k+1) < L_i(k) = t$ aina, kun $L_i(k) \neq 0$. Jos puolestaan alun perin $L_i(k) = 0$, tällöin loppuliitteiden $X[i..m]$ ja $Y[1..n]$ – eli samalla koko Y-vektorin – muodostaman PYAn pituus $< k$. Silloin varmasti myös arvot $L_i(k+1)$, $L_i(k+2)$, …, $L_i(m+1)$ ovat kaikki nollia. Täten lemman 3.3 voi todeta pitävän paikkansa. □

Lemma 3.4: $\forall i \in [1..m-1],\ \forall k \in [1..m]: L_{i+1}(k) \leq L_i(k)$.

Lemman 3.4 sanoma on, että k:n mittaisen PYAn muodostamiseksi tarvittavan minimaalisen Y:n loppuliitteen pituus pysyy ennallaan tai lyhenee, jos tarkasteltavaa X:n loppuliitettä pidennetään yhdellä nykyisestään. Tämäkin lemma on verrattain suoraviivaisesti todennettavissa käyttämällä hyväksi $L_i(k)$-arvojen määritelmää.

Todistus: Oletetaan, että $L_{i+1}(k) = t \geq 0$. Tällöin loppuliite $X[i..m]$ sisältää *osajononaan* yhtä merkkiä lyhyemmän loppuliitteen $X[i+1..m]$, joten k:n mittaisen PYAn muodostaminen $X[i..m]$:n kanssa ei vaadi Y:stä missään tapauksessa ainakaan pidempää loppuliitettä kuin sen muodostaminen osajonon $X[i+1..m]$ kanssa. Täten on todistettu alkuperäinen väite $L_{i+1}(k) \leq L_i(k)$. □

Lemma 3.5: *$L_i(k) = Max\{h, L_{i+1}(k)\}$, missä h on suurin Y-indeksi siten, että $x_i = y_h$ ja $L_{i+1}(k) < h < u$, missä $u = L_{i+1}(k-1)$. Ellei tällaista h:ta ole olemassa, $L_i(k) = L_{i+1}(k)$.*

Edellä todistetun lemman 3.4 perusteella on selvää, että $L_{i+1}(k) \leq L_i(k)$. Lemma 3.5 määrittelee nyt yksikäsitteisesti, miten arvo $L_i(k)$ pystytään laskemaan, kun tiedetään entuudestaan arvot $L_{i+1}(k) = t$ ja $L_{i+1}(k-1) = u$, $u > t$. Sen mukaan $L_i(k) > L_{i+1}(k)$ tarkalleen silloin, kun x_i muodostaa täsmäyksen Y:n merkin kanssa indeksialueella $t+1..u-1$. Myös lemman 3.5 sisältämä väite on helpohkosti todistettavissa.

Todistus: Oletetaan ensiksi, että vektorista Y indeksiväliltä $t+1..u-1$ löytyy vähintään yksi merkki, jonka kanssa x_i täsmää. Tällöin niistä pitää merkinnän $L_i(k)$ minimaalisuuden takia valita *oikeanpuoleisin* eli *se, jonka indeksi on suurin*, jotta Y:n loppuliitteen pituus olisi lyhin mahdollinen k:n mittaisen PYAn muodostamiseksi $X[i..m]$:n kanssa. Merkitään kyseisen täsmäyksen indeksiä h:lla. Toisin sanoen, merkin x_i kanssa täsmäävä merkki y_h sijaitsee pienemmässä indeksissä kuin $L_{i+1}(k-1) = u$. Tämä tarkoittaa sitä, että täsmäyksen (x_i, y_h) perään mahtuu yhä loppuliitteiden $X[i+1..m]$ ja

$Y[u..n]$ välinen k–1:n mittainen PYA. Mutta samanaikaisesti on voimassa myös $h > L_{i+1}(k)$, mikä merkitsee, että k:n mittaisen PYAn muodostamiseksi tarvittava Y:n loppuliite *lyhenee*, kun X:n loppuliite pitenee yhdellä $X[i+1..m]$:stä $X[i..m]$:ään. Jos puolestaan x_i:llä ei olekaan täsmäystä indeksialueella $t+1..u–1$, X:n loppuliitteen laajentamisesta ei ollut hyötyä, vaan lemman 3.4 mukaisesti tarvitaan yhtä pitkä Y:n loppuliite k:n mittaisen PYAn muodostamiseksi sekä $X[i..m]$:n kuin jo aiemmin $X[i+1..m]$:n kanssa. Siten voidaan koko väitteen todeta pitävän paikkansa. □

3.3.1.2 Teorian soveltaminen käytäntöön

NKY-algoritmin toiminta perustuu suoraan edellä esiteltyihin kolmeen lemmaan. Algoritmin esiprosessointivaihe on minimaalinen: menetelmä ei tarvitse avukseen mitään erityisiä tietorakenteita, vaan se lähtee suoraan prosessoimaan syötevektoreitaan käyttämällä lineaarihakua. Ainoastaan kooltaan $(m+1)m$ oleva taulukko M joudutaan perustamaan $L_i(k)$-arvojen muistiin tallentamista varten. Kukin laskettava arvo $L_i(k)$ sijoitetaan taulukkoon paikkaan $M[i, k]$. Pysäytysalkioiksi matriisiin M joudutaan tallentamaan nollat soluihin $M[i+1, k]$, kun sisemmän silmukan laskuri saa ensimmäistä kertaa arvon k.[68]

Algoritmin ulomman silmukan laskuri *diagpos* aloittaa arvosta m, ja jokaisen uuden kierroksen alkaessa laskurin alkuarvoa pienennetään yhdellä. Muuttujan *diagpos* arvo kopioidaan kierroksen aluksi muuttujalle i, jonka arvo kuvaa, mistä indeksistä alkavaa vektorin X loppuliitettä ollaan paraikaa tutkimassa. Sisemmässä silmukassa laskurina esiintyy ykkösestä aloittava j, joka kuvaa paraikaa etsittävän yhteisen alijonon pituutta.

Ulomman silmukan ensimmäisen kierroksen aikana määrätään sisemmässä silmukassa järjestyksessä arvot $L_m(1)$, $L_{m–1}(2)$, $L_{m–2}(3)$, …, $L_{m–k+1}(k)$, kunnes lopulta joko jollain k:n arvolla $L_{m–k+1}(k)$ saa arvon 0, eli $X[m–k+1..m]$ ei enää muodosta k:n mittaista PYAa edes koko vektorin Y kanssa, tai sitten k ehtii kasvaa arvoon $m+1$, jolloin koko syötevektori X esiintyi jo Y:n alijonona. Jälkimmäisessä tapauksessa algoritmin suoritus voidaan jo lopettaa, koska tuolloin selvästikin $p = m$.

Oletetaan seuraavaksi, että ensimmäisen ulomman silmukan kierroksen aikana löydettiin PYA, jonka pituus < m–1.[69] Tieto toistaiseksi löytyneen PYAn pituudesta tallennetaan muuttujaan *maxpituus*, joka on algoritmin alussa asetettu nollaksi. Ulomman silmukan *toisen kierroksen* aikana lasketaan sisemmässä silmukassa arvot $L_{m–1}(1)$, $L_{m–2}(2)$, $L_{m–3}(3)$, …, $L_{m–k}(k)$, kunnes joko syötevektori X loppuu kesken, tai jollain k:n arvolla $L_{m–k}(k)$ saa arvon 0. Arvojen määrääminen tapahtuu suoraan lemmaan 3.5 perustuen, eli etsitään merkille $x_{m–k}$ täsmäystä vektorista Y indeksialueelta $t+1..u–1$,

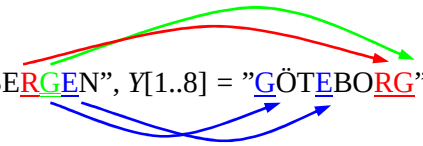
[68] Matriisin M rivit ja sarakkeet vaihtavat tässä esityksessä paikkaa NKY:n esitykseen nähden, jotta indeksi i viittaisi riveihin ja j sarakkeisiin kuten aiemminkin tässä työssä.
[69] Perustelut sille, miksi algoritmin ulomman silmukan suoritus voidaan lopettaa jo p:n saavutettua arvon m–1, esitetään hetkeä myöhemmin.

missä $t = L_{i+1}(k)$ ja $u = L_{i+1}(k–1)$. Täsmäyksen löytyessä valitaan niistä oikeanpuoleisin ja asetetaan sen indeksi h $L_i(k)$:n arvoksi. Ellei täsmäystä vaaditulta alueelta löydy, asetetaan $L_i(k) = L_{i+1}(k) = t$. Ajatuksena on siis tutkia, lyhentääkö X:n loppuliitteen kasvattaminen yhdellä ulomman silmukan edellisen kierroksen aikana laskettujen $L_{i+1}(k)$-arvojen mukaisia Y:n minimaalisia loppuliitteitä k:n mittaisen PYAn muodostamiseksi.

Kolmannella ulomman silmukan kierroksella – mikäli sellainen vielä on tarpeen suorittaa – lähdettäisiin nyt laskemaan arvoja $L_{m–2}(1)$, $L_{m–3}(2)$, $L_{m–4}(3)$, …, $L_{m–k–1}(k)$. Yleisesti, ulomman silmukan z. kierroksella määrätään arvot $L_{m–z–k+2}(k)$, ja kierroksen aikana sisempää silmukkaa suoritetaan niin pitkään, kunnes joko syötevektori X loppuu kesken tai kohdataan ensimmäinen k, jolle $L_{m–z–k+2}(k) = 0$. Kierroksen päättyessä vektorin loppuliite $X[m–k–z+3..m]$ muodostaa k–1:n pituisen PYAn syötevektorin Y kanssa.

Ulomman silmukan jokaisen kierroksen päättyessä laskurin k viimeisin arvo osoittaa kierroksen aikana löytyneen PYAn pituuden ykkösellä lisättynä. Jos se ylittää edellisten kierroksien aikana löytyneen pituuden, päivitetään uusi tieto muuttujaan *maxpituus*. Ulomman silmukan suorittaminen voidaan lopettaa, kun aikanaan tullaan tilanteeseen, jossa *maxpituus* ≥ *diagpos*. Tällöin tiedetään, ettei PYA voi enää seuraavien kierrosten aikana pidentyä, koska syötevektorin lopusta jätetään jokaisella ulomman silmukan kierroksella viimeiset *m–diagpos* merkkiä tutkimatta. Tästä on pääteltävissä, että *NKY* soveltuu hyvin käytettäväksi syötteille, joiden PYA on pitkä eli ≈ *m*. Seuraavassa esitetään vielä esimerkki, joka havainnollistanee $L_i(k)$-arvojen kehittymistä algoritmin suorituksen edetessä.

Esimerkki 3.22: $X[1..6]$ = ”BERGEN”, $Y[1..8]$ = ”GÖTEBORG”, $p = 3$, PYA = ”ERG”



Tarkastellaan aluksi X:n loppuliitettä $X[4..6]$ = ”GEN”. Havaitaan, että $L_4(1) = 8$, sillä x_4 = ”G” esiintyy viimeisenä merkkinä vektorissa Y (**vihreä** nuoli). $L_4(2)$ on puolestaan 1, sillä ”N”:ää ei esiinny Y:ssä lainkaan, ja alijono ”GE” alkaa Y:ssä viimeisen (ja samalla ainoan) kerran position 1 kohdalta (**siniset** nuolet). Lemman 3.4. perusteella on nyt voimassa $L_3(2) = Max\{L_4(2), h\}$, missä $x_3 = y_h$ ja $h < L_4(1) = 8$. Koska esimerkissä x_3 = ”R” = y_7 ja $L_4(2) = 1 < h = 7 < L_4(1) = 8$, saadaan $L_3(2)$:n arvoksi 7. Siten X:n loppuliitteen pidentäminen kolmen pituisesta ”GEN”:stä neljän mittaiseksi ”RGEN”:ksi vähentää Y:n loppuliitteen merkkien määrää, joka tarvitaan kahden mittaisen PYAn konstruoimiseksi syötevektoreiden loppuliitteistä, kahdeksasta kahteen (**punainen** nuoli).

3.3.1.3 Esimerkki ja algoritmin analyysi

Seuraavassa on NKY:n toimintaa kuvaava esimerkki, jonka syötejonot ovat samat kuin esimerkissä 3.1. PYA pystytään palauttamaan matriisista M etsimällä sarake, jonka

numero on sama kuin PYAn pituus eli muuttujan *maxpituus* arvo algoritmin laskentavaiheen päättyessä[70]. Sarakkeelta etsitään alin rivi, jolla esiintyy nollasta poikkeava arvo. Olkoon solun indeksi (*i*, *k*). Soluun tallennettu arvo kuvaa ensimmäisen PYAan kuuluvan merkin *Y*-indeksiä. Merkin vastinparin sijainnin vektorissa *X* osoittaa solun rivi-indeksi. Seuraava edustaja PYA-jonoon saadaan etenemällä paikasta (*i*, *k*) vasemmalle alaviistoon soluun (*i*+1, *k*–1), ja kyseistä saraketta pitkin edetään alaspäin taas niin pitkään, kunnes kohdataan viimeisen kerran sama arvo kuin sarakkeelle saavuttaessa. Tuon solun rivinumero kertoo jälleen täsmäyksen *X*- ja siihen tallennettu arvo *Y*-indeksin. PYAn keräämistä jatketaan samalla tavalla, kunnes sen viimeinenkin merkki on löydetty tutkimalla edellä mainitulla periaatteella sarakkeen 1 sisältöä.

Esimerkki 3.23: *$L_i(k)$-arvojen tallentuminen matriisiin M ratkaistaessa PYA esimerkin 3.1 mukaisille syötteille käyttämällä NKY-algoritmia.*

			Y															
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
			B	A	A	C	D	C	B	B	A	C	C	C	D	D	B	A	B
	1	A										0							
	2	B									1	0							
	3	B								7	1	0							
	4	B							8	3	(1)	0							
	5	D						9	5	3	0								
	6	D					13	9	5	3	0								
	7	D				14	11	9	5	3	0								
	8	A			15	12	11	9	4	(3)	0								
X	9	C		16	15	12	11	6	(4)	0									
	10	B	17	16	15	12	8	6	1	0									
	11	C	17	16	15	12	7	(6)	0										
	12	B	17	16	15	8	(7)	0											
	13	B	17	16	15	(8)	1	0											
	14	A	17	16	(9)	3	0												
	15	B	17	(15)	8	0													
	16	C	17	12	0														
	17	B	(17)	0															
	18		0																

NKY-algoritmin aika- ja tilavaativuudesta

NKY:n pseudokoodi löytyy liitteen kohdasta 11.3.1. Kuten jo algoritmin toimintaa kuvattaessa todettiin, NKY:n esiprosessointivaiheessa tarvitsee alustaa ainoastaan yksinkertaisia muuttujia ja varata staattinen muistialue taulukkoa *M* varten. Siten algoritmin alustustoimenpiteet ovat vakioaikaisia.

[70] lukuun ottamatta erikoistilannetta, että *maxpituus* = 0

NKY:n laskentavaiheesta huolehtivat kolme sisäkkäistä silmukkaa *S1* – *S3*. Näistä ulointa eli *S1*:tä suoritetaan niin pitkään kuin PYA voi vielä pidentyä, eli *S1*:n *i*. kierroksen alkamiseen mennessä löydetyn PYAn pituus on korkeintaan $m–i$. Pituudeltaan p:n mittaisen PYAn on pakko löytyä viimeistään kierroksella $m–p+1$, koska kyseinen kierros, jossa *X*:n selaaminen aloitetaan indeksistä p alkuun päin, on viimeinen, jossa vektorista *X* vielä käsitellään vähintään p merkkiä. Siten ulompaa silmukkaa suoritetaan enintään $m - p + 1 = O(m - p)$ kertaa. Silmukan *S1* muut lauseet paitsi sisempi silmukka *S2* ovat vakioaikaisia. Siinä puolestaan otetaan vuoron perään kukin *X*:n merkeistä käsittelyyn niin kauan, kunnes vektori loppuu kesken, tai kohdasta *i* alkavan loppuliitteen $X[i..m]$ $L_i(k)$-arvoksi tulee 0. Vieläkin sisempänä oleva silmukka *S3* vain siirtää jatkuvasti *Y*-vektorin indeksiä yhdellä alkua kohti. Vaikka *S2*:ssa tarkasteltava *X*:n symboli vaihtuu, *Y*:ssä edetään jatkuvasti alkuun päin palaamatta milloinkaan takaisin. Pahimmassa tapauksessa silmukoissa *S2* ja *S3* kelataan yhdessä koko vektori *Y* lopusta alkuun, eli niiden suorittaminen maksaa $O(n)$. Siten koko laskentavaiheen kustannukseksi saadaan $O(n(m–p))$. PYA-jonon keräämisvaiheen kustannus $O(p)$ ei enää kasvata algoritmin asymptoottista kokonaissuoritusaikaa laskentavaiheen kustannuksesta.

Rick on vuonna 1994 ilmestyneessä tutkimusraportissaan [Ric94] todennut, että NKY:n suoritusaikaa voitaisiin rajoittaa ylhäältä lausekkeella $O(n\sigma+p(m–p))$, jos algoritmia varten perustettaisiin esiprosessointivaiheessa lähiesiintymätaulukko vektorille *Y*. Aakkoston koon ollessa pieni, PYAn ollessa lyhyt ja *Y*:n ollessa huomattavasti syötejonoa *X* pidempi saatettaisiin säästää käytännön suoritusajassa verrattain paljonkin esiprosessoimalla NKY:tä Rickin ehdottamalla tavalla.

Algoritmille teettävät siten eniten työtä syötejonot, joissa ei esiinny ainoatakaan täsmäävää merkkiparia, eli joiden $p = 0$. Tällöin NKY:ssä joudutaan selaamaan vektori *Y* kokonaisuudessaan läpi jokaista *X*:n merkkiä kohti, eli suoritusajaksi saadaan $O(mn)$. Parhaimmassa tapauksessa NKY ratkaisee PYA-ongelman kuitenkin jopa ajassa $O(n)$. Näin tapahtuu, kun $p = m$.

Algoritmi tarvitsee ei-vakiomääräistä muistitilaa vain syötejonoilleen sekä matriisille *M*, jota tarvitaan $L_i(k)$-arvojen tallentamiseen. Taulukolle varataan muistia $O(m^2)$, joka on samalla koko algoritmin tilavaativuus. Tekijät huomauttavat, että taulukosta vähintään likimain puolet, eli oikea alakolmio jää aina käyttämättä. Haluttaessa säästää tarvittavaa muistitilaa pitää kuitenkin *matriisin indeksointi organisoida uudelleen* [BHV03]. Tätä tarkastellaan lähemmin aliluvussa 8.2.

3.3.2 Kumarin ja Ranganin algoritmi (KRA)

3.3.2.1 Lineaaritilainen muunnelma NKY-algoritmista

Yleistä

Kumar ja *Rangan* [Kum87] esittelivät vuonna 1987 lineaarisessa muistitilassa toimivan PYA-algoritminsa, josta tässä työssä käytetään tunnistetta *KRA*. Siinä lasketaan aluksi pelkkä syötemerkkijonojen PYAn pituus tekemättä minkäänlaista kirjanpitoa ratkaisupolkujen ylläpitämiseksi. Tällä menettelyllä vältytään $n(m+1)$-kokoisen matriisin perustamiselta – toisin kuin menetelmän ilmeisenä esikuvana olevassa, *Nakatsun, Kambayashin* ja *Yajiman* (NKY) viisi vuotta aikaisemmin julkaisemassa algoritmissa [NKY82]. NKY-algoritmin suoritusaika on samaa suuruusluokkaa kuin KRA:n eli $n(m–p)$, mutta sen muistintarve on luokkaa $O(m^2)$, mikä rajoittaa oleellisesti NKY:n käyttökelpoisuutta pitkille syötejonoille. Kumarin ja Ranganin algoritmissa PYAn laskentavaihe noudattaa matriisin ylläpitoa lukuun ottamatta täysin NKY:stä tuttuja piirteitä eli jonojen prosessointia lopusta alkuun diagonaaleittain. Huomion arvoista on kuitenkin, ettei tekijöiden ehdotuksesta huolimatta Kumarin ja Ranganin algoritmissa ole mitenkään välttämätöntä selvittää PYAn pituutta typistetyllä NKY-menetelmällä, vaan se voitaisiin ratkaista millä tahansa PYAn pituuden määräävällä algoritmilla! Tällaista valinnan vapautta ei ole tarjolla HIL-algoritmissa[71] [Hir75], jossa ensimmäisen kutsukerran aikana on ratkaistava syötejonon X suhteen puolitettujen ongelmien pisimpien yhteisten alijonojen pituudet erisuuntaisesti Wagnerin ja Fischerin [Wag74] tapaan ja kirjattava saadut tulokset muistiin jonon Y jokaista m:n mittaista alku-loppuliiteparia[72] kohti.

Kumarin ja Ranganin algoritmin suurena etuna HIL:ään verrattuna on juuri riippumattomuus PYAn laskentatavasta. Tämä johtaa väistämättä myös HIL:stä poikkeavaan tapaan PYA-jonon määräämiseksi. HIL:ssä jokaisella rekursioaskeleella puolitettiin tarkasteltavan ongelman X-vektorin suuntainen dimensio. Myös KRA:ssa aliongelman X-vektorin pituus lyhenee rekursion edetessä aina syvemmälle, mutta osituskriteerinä toimii ns. *roskamäärä* (engl. *amount of waste*). Roskamäärällä tarkoitetaan niiden syötejonon X merkkien lukumäärää, jotka eivät kuulu syötteiden X ja Y väliseen PYAan. Seuraava esimerkki valaisee roskamäärän käsitettä.

[71] Hirschbergin lineaaritilainen algoritmi esiteltiin aliluvussa 3.2.1.
[72] Y:n alkuliitteen pituuden ollessa k ($0 \le k \le n$) on sen loppuliitteen pituus $n – k$.

Esimerkki 3.24: X = ”HELSINKI”, Y = ”BERLIINI”, $m = n = 8$, $p = 5$, PYA = ”ELINI”
$\Rightarrow$ *syötevektorin X roskamäärä on* $8 - 5 = 3$.

		0	1	2	3	4	5	6	7	8
			B	E	R	L	I	I	N	I
0		0	0	0	0	0	0	0	0	0
1	H	0	0	0	0	0	0	0	0	0
2	E	0	0	**1**	1	1	1	1	1	1
3	L	0	0	1	1	**2**	2	2	2	2
4	S	0	0	1	1	2	2	2	2	2
5	I	0	0	1	1	2	**3**	3	3	3
6	N	0	0	1	1	2	3	3	**4**	4
7	K	0	0	1	1	2	3	3	4	4
8	I	0	0	1	1	2	3	4	4	**5**

Oheiseen kuvaan on merkitty syötejonojen X = ”HELSINKI” ja Y = ”BERLIINI” PYAn muodostavat täsmäykset punaisin palloin. PYAan kuuluvat vastinmerkit kummastakin syötejonosta on maalattu **vihreiksi**. Ne syötevektorin X merkit, jotka eivät muodosta esimerkkisyötteiden PYAan kuuluvaa täsmäystä, on lihavoitu **sinisellä** värillä. Kyseiset kolme merkkiä ”H”, ”S” ja ”K” ovat nyt roskaa, joten esimerkin roskamääräksi saadaan 3.

HIL-algoritmin käsittelyn yhteydessä osoitettiin, että yksittäinen PYA-ongelma voidaan osittaa tietyistä kohdin X- ja Y-vektoria kahteen pienempään osaongelmaan siten, että yhdistämällä näiden ratkaisut saadaan kelvollinen ratkaisu alkuperäiselle ongelmalle. Jos vektori X katkaistaan mielivaltaisesta kohdasta i, ($1 \le i \le m$), on Y:n katkaisun tapahduttava siten, että X:n alkuosan viimeinen PYAan kuuluva täsmäys mahtuu Y:n alkuosaan. Vastaavasti X:n loppuosan ensimmäisen täsmäyksen on mahduttava Y:n loppuosaan. Kyseiset kriteerit täyttäviä pisteitä löytyy kutakin mahdollista i:n valintaa kohti vähintään yksi. Näitä pisteitä kutsutaan Kumarin ja Ranganin algoritmissa ns. *laillisiksi leikkauspisteiksi* (engl. *valid cut*).

Roskan eliminointi

Kun HIL:ssä kullakin osituskerralla syötevektorin X koko puolitetaan, KRA:ssa osituksen kriteerinä on *roskamäärän puolittaminen* rekursion edetessä. Roskamäärä φ saadaan laskettua suoraviivaisesti kaavalla $\varphi = m - p$, kunhan syötevektorin X ja $PYA(X, Y)$:n pituudet tiedetään. Pulmana on kuitenkin, mistä kohdin vektorit X ja Y pitää jakaa kahtia, jotta ensimmäisen aliongelman roskamääräksi saataisiin juuri $\lceil \varphi/2 \rceil$. Ratkaisu ongelmaan löytyy tarkastelemalla NKY-algoritmin toimintaa. Kyseisessä algoritmissa tutkitaan PYAn pituuden selvittämiseksi tarkalleen $m - p + 1$ diagonaalia lopusta alkuun päin, eli tarkasteltavia diagonaaleja on yhteensä $\varphi + 1$ kappaletta. Tutkimalla ainoastaan $\lfloor \varphi/2 \rfloor + 1$ lävistäjää saadaan selville, miten pitkä PYA voidaan muodostaa ohittamalla vektorin X lopusta lukien tarkalleen $\lfloor \varphi/2 \rfloor$ merkkiä. Kun kaikki $\lfloor \varphi/2 \rfloor + 1$ lävistäjää on tutkittu, talletetaan tieto näitä lävistäjiä pitkin saavutetun PYAn pituudesta muuttujaan $p_{loppuosa}$. PYAn pituustiedon lisäksi on tarpeen kirjata muistiin, mistä Y:n indeksistä alkava loppuliite riittää muodostamaan u:n ($0 \le u \le p$) mittaisen PYAn X:n loppuosan $X[m - \lfloor \varphi/2 \rfloor - u + 1..n]$ kanssa. Näitä tietoja tarvitaan roskamäärän puolittavan osituspisteen määräämiseksi. Tiedot kirjataan vektoriin $loppuosa[0..p]$.

Paikkaan $loppuosa_0$ tallentuu aina arvo $n + 1$, sillä nollan mittaisen PYAn tuottamiseksi ei tarvita Y:n lopusta yhtään merkkiä. NKY-algoritmissa merkintä $L_i(k)$ tarkoittaa, montako merkkiä pitkä Y:n loppuliite vähintään tarvitaan muodostamaan k:n mittainen PYA $X[i..m]$:n kanssa. Vektoriin *loppuosa* edellä viedyt arvot ovat toisin sanoen NKY:ssä laskettavat $L_i(k)$-arvot i:n arvoilla $m - \lfloor \varphi/2 \rfloor - u + 1$ ja j:n arvoilla u, kun $u \in [0..p_{loppuosa}]$. Jokaiseen tällä tavoin valittuun X:n ja Y:n loppuliitepariin liittyy nyt selvästikin $\lfloor \varphi/2 \rfloor$:n suuruinen roskamäärä.

Seuraavaksi pitäisi vielä tutkia syötejonojen *alkuliitteitä*, joiden roskamääräksi saataisiin tarkalleen $\lceil \varphi/2 \rceil$. Määritellään seuraavaksi, että merkintä $L_i^*(j)$ tarkoittaa lyhimmän sellaisen Y:n alkuliitteen pituutta, joka riittää muodostamaan j:n mittaisen PYAn X:n alkuliitteen $X[1..i]$ kanssa. Vaikkakin NKY prosessoi merkkijonoja lopusta alkuun päin, tämä ei osoittaudu ongelmaksi vaaditun kaltaisten alkuliitteiden PYAn pituuksien määräämiselle. Lyhyellä päättelyketjulla voidaan nimittäin osoittaa seuraavan lemman todenperäisyys.

Lemma 3.6: $L_i^*(k)_{(X,\ Y)} = n + 1 - L_{m-i+1}(k)_{(X^*,\ Y^*)}$, *missä X^* ja Y^* tarkoittavat syötejonoja X ja Y käänteisessä järjestyksessä. Alaindeksi merkinnän $L_i(k)$ yhteydessä ilmaisee, määrätäänkö $L_i(k)$-arvo alkuperäisille vai käännetyille syötejonoille.*

Todistus: Mikäli jonot X ja Y käännetään, muuttuvat niiden alkuliitteet jonojen X^* *ja* Y^* *käännetyiksi loppuliitteiksi.* Tällöin erityisesti alkuliitteet $X[1..i]$ *ja* $Y[1..L_i^*(k)_{(X,Y)}]$ kuvautuvat merkkijonojen X^* ja Y^* käännetyiksi loppuliitteiksi $X^*[m..m-i+1]$ ja $Y^*[n..n-L_i^*(k)_{(X,\ Y)}+1]$. Koska kahden jonon välisen pisimmän yhteisen alijonon pituus ei muutu tarkastelusuuntaa vaihdettaessa, on myös $X^*[m-i+1..m]$:n ja $Y^*[n - L_i^*(k)_{(X,\ Y)}+1..n]$:n PYAn pituus k, ja samalla $n - L_i^*(k)_{(X,\ Y)} + 1$ on viimeinen Y^*:n indeksi, josta alkaen voidaan muodostaa *i:n mittainen yhteinen alijono* $X[m+1-i..m]$:n kanssa. Siten $L_{m-i+1}(k)_{(X^*,\ Y^*)} = n + 1 - L_i^*(k)_{(X,\ Y)}$, josta termejä siirtelemällä saadaan edelleen $L_i^*(k)_{(X,\ Y)} = n + 1 - L_{m-i+1}(k)_{(X^*,\ Y^*)}$.

Lemman 3.6 avulla pystytään siis rakentamaan silta termien $L_i(k)$ ja $L_i^*(k)$ välille. Termien välisen yhteyden löydyttyä ovat nyt käytettävissä välineet syötemerkkijonojen sellaisten alkuliitteiden konstruoimiseksi, joiden roskamääräksi tulee vaadittu $\lceil \varphi/2 \rceil$. Vaaditun kaltaiset alkuliitteet löydetään suorittamalla NKY-algoritmia $\lceil \varphi/2 \rceil + 1$ kierrosta käännetyille X- ja Y-syötejonoille. Suorituksen päätyttyä tiedetään, miten pitkä PYA voidaan syötejonojen alkuliitteistä muodostaa, jos tarkalleen $\lceil \varphi/2 \rceil$ merkin pitää tulla ohitetuiksi X:stä. Kyseinen tieto tallentuu muuttujaan $p_{alkuosa}$. Lisäksi vektoriin *alkuosa*$[0..p]$ ovat tallennettuina NKY-algoritmin laskemat $L_i(k)$-arvot X^*:lle ja Y^*:lle i:n arvoilla $m - \lceil \varphi/2 \rceil - u + 1$ ja k:n arvoilla u, kun $u \in [0..p_{alkuosa}]$. Paikkaan $alkuosa_0$ tallentuu pseudoarvo $n + 1$. Jokaiseen tällä tavoin valittuun X^*:n ja Y^*:n loppuliitepariin liittyy nyt $\lceil \varphi/2 \rceil$:n suuruinen roskamäärä. Käännetyille jonoille lasketuista $L_i(k)$-arvoista

päästään siirtymään vastaaviin $L_i^*(k)$-arvoihin lemman 3.6 mukaisesti vähentämällä kyseiset arvot $n + 1$:stä yksi kerrallaan.

Kun alkuperäisten syötteiden alku- ja loppuliitteiden roskamäärät on saatu halutun suuruisiksi, on seuraavaksi vuorossa sopivan osituskohdan määrääminen vektoreita *alkuosa* ja *loppuosa* tarkastelemalla. Rajoituksena alku- ja loppuosien yhteen liittämiselle ovat luonnollisestikin alijono-ominaisuuden säilyminen; ts. alku- ja loppuosista saatavien jonojen indeksit eivät saa mennä ristikkäin. Lisäksi ositteiden yhteenlasketun PYA-pituuden on oltava alkuperäisten syötteiden PYAn pituus eli p. Leikkauspiste (i, j) täyttää siten vaatimukset

1) $i = k + \lceil \varphi/2 \rceil : (k \leq p_{alkuosa}) \wedge (p - k \leq p_{loppuosa}) \wedge (alkuosa_k < loppuosa_{p-k})$
   ja
2) $j = alkuosa_k$.

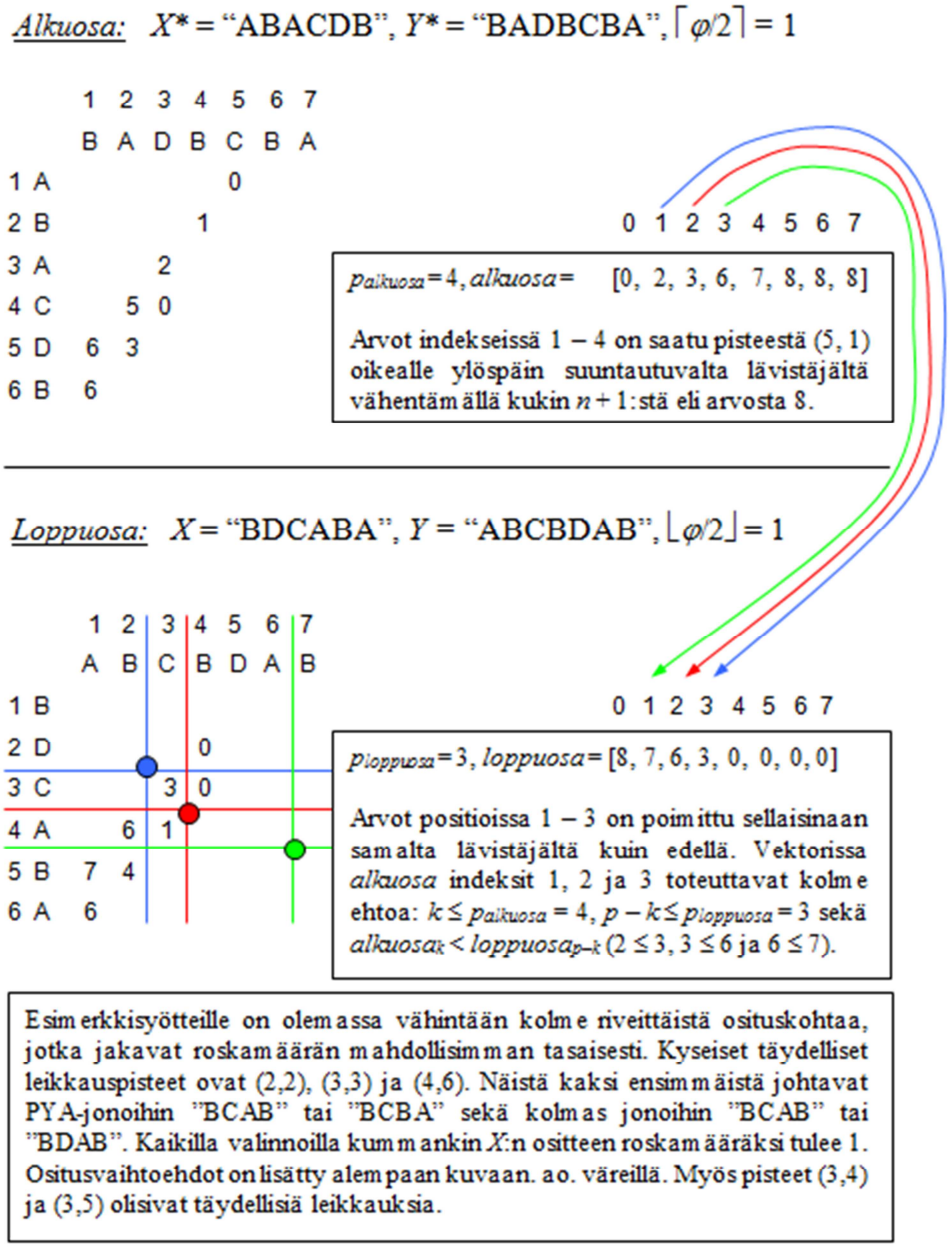
Yllä olevat vaatimukset täyttäviä pisteitä on aina olemassa vähintään yksi. Näistä pisteistä käytetään nimitystä *täydellinen leikkaus* (engl. *perfect cut*). Kaikki täydellisen leikkauksen kriteerin täyttävät pisteet ovat samalla laillisia leikkauspisteitä. Kannattaa kuitenkin huomioida, että yhden täydellisen leikkauksen selvittäminen syötteitä kohti riittää, eli kaikkia täydellisiä leikkauksia ei yritetäkään hakea. Esimerkissä 3.25 simuloidaan täydellisen leikkauksen etsintämenettelyä KRA-algoritmissa.

Koska täydellinen leikkaus on määritelty KRA:ssa aina, kun syötteiden roskamäärä on vähintään kahden suuruinen[73], voidaan PYA-jonon määräämisessä edetä tekemällä rekursiivisia kutsuja alkuperäisen ongelman aliositteille – ensiksi alku- ja sitten loppuosalle. Kun lopulta päädytään tilanteeseen, että roskamäärä on pienentynyt ykköseksi tai nollaksi[74], on saavutettu rekursion kanta. Jos roskamäärä on nolla, hyväksytään kaikki tarkasteltavan ositteen X:ään kuuluvat merkit PYA-jonoon. Muussa tapauksessa joudutaan ositteen X- ja Y-jonoja tutkimaan pareittain niin pitkään, kunnes ainoa PYAan kuulumaton X:n merkki löytyy. Tätä merkkiä lukuun ottamatta kaikki muut X:n ositteen merkit hyväksytään PYAan. Kun viimeinenkin rekursiivinen kutsu on saatu suoritettua loppuun, eli alku- ja loppuosien PYA-jonot on katenoitu, algoritmin suoritus päättyy.

[73] Tällöin kummankin aliositteen roskamäärä on nollasta poikkeava, eli ≥ 1.

[74] Roskamäärää nolla voi esiintyä vain jälkimmäisessä ositteessa, ja se on mahdollista tarkalleen silloin, kun alkuositteen roskamäärä on ollut 1.

Esimerkki 3.25: $X[1..6]$ = ”BDCABA”, $Y[1..7]$ = ”ABCBDAB”, $p = 4$, $\varphi = 2$.



3.3.2.2 Algoritmin toteutuksen kuvaus

Kumarin ja Ranganin algoritmi on lohkottu liitteen kohdassa 11.3.2 kuuteen proseduuriin. Pääalgoritmi sisältää ainoastaan proseduurien *LaskePYAnPituus* sekä *RatkaisePYA-jono* kutsut. Ensin mainittua suoritetaan tarkalleen kerran. Jälkimmäinen kutsuu itseään niin pitkään, kunnes ositteen roskamäärä on pienentynyt alle kahden. Tällöin ratkaistaan triviaali ongelma proseduurissa *RatkaiseTriviaaliTapaus* kopioimalla tarkasteltavan ositteen X:n merkeistä vähintään $m - 1$ kappaletta PYA-jonoon S. Muussa tapauksessa kutsutaan proseduuria *EtsiTäydellinenLeikkaus* nykyisen ongelman osittamiseksi. Kyseinen proseduuri sisältää kaksi kutsua proseduurille *TutkiOsite*. Viimeksi mainitun proseduurin tehtävänä on vain kontrolloida, montako

diagonaalia aliongelmasta prosessoidaan, sekä välittää tuloksena saadut $L_i(k)$-arvot kutsujalleen. Proseduuri *TutkiYksiDiagonaali* on yleiskäyttöisin. Sitä käytetään sekä PYAn pituuden laskemiseksi, täydellisen leikkauksen etsimiseksi että triviaalin ongelman ratkaisemiseksi. Kyseinen proseduuri huolehtii yksittäisen diagonaalin $L_i(k)$-arvojen laskennasta NKY-algoritmin tapaan.

Esimerkki

Kumarin ja Ranganin algoritmin löytämää ratkaisua yhdelle mahdolliselle syötemerkkijonoparille on hahmoteltu esimerkissä 3.26. Kuvaan on merkitty myös algoritmin löytämän täydellisen leikkauksen mukainen osituskohta proseduurin *RatkaisePYA-jono* ensimmäisellä kutsukerralla. Kyseinen piste (5, 1) on merkitty kuvaan **mustalla** pyörylällä. Seuraavalla rekursiotasolla tutkitaan ainoastaan mainittuun pisteeseen oikeasta alakulmastaan rajoittuva vasemman yläkulman alue X[1..5], Y[1..1] (värjätty **vaaleanvihreäksi**) ja oikea alapuoleinen alue X[6..17], Y[2..17] (värjätty **vaaleansiniseksi**). Ensimmäisestä ositteesta PYAan valitaan vain yksi piste, (4, 1), kun taas loput PYAan valittavat kahdeksan pistettä saadaan jälkimmäisestä, suuremmasta ositteesta. Ositteiden roskamäärät saadaan vähentämällä ositteen X:n suuntaisen dimension pituudesta ositteesta PYAan valittujen merkkien lukumäärä. Alueiden roskamääriksi saadaan siten 5 – 1 = 4 ja 12 – 8 = 4, joten piste (5, 1) todellakin tasapainottaa ositteet roskamäärien suhteen. Koska algoritmissa pyritään roskamäärien tasapainottamiseen etsimällä ensisijaisesti ylimpänä ja toissijaisesti vasemmanpuoleisimpana sijaitseva täydellinen leikkauspiste, PYAan tulee esimerkkitapauksessa valituiksi enemmän pisteitä loppu- kuin alkuositteesta. KRA-algoritmi löytää esimerkissä täsmälleen saman ratkaisun kuin ei-lineaarisessa muistitilassa toimiva NKY-algoritmi.

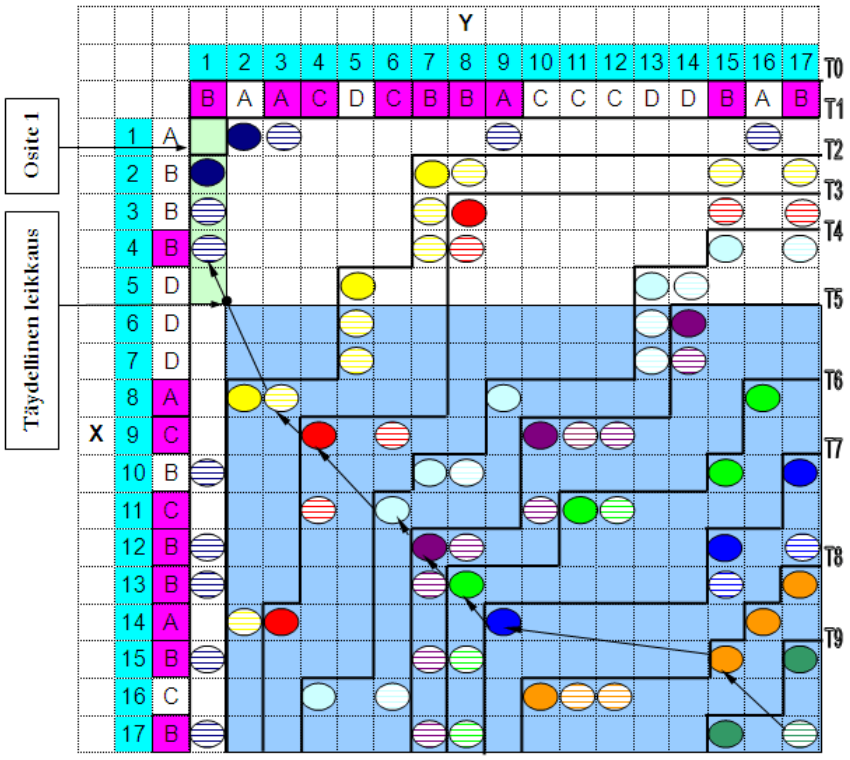
3.3.2.3 Aika- ja tilavaativuus

KRA-algoritmissa ratkaistaan ensiksi PYAn pituus, johon tarvitaan aikaa $O(n(m–p))$, mikäli pituuden laskemiseksi käytetään muistinkäytöltään riisuttua Nakatsun, Kambayashin ja Yajiman algoritmia. Tarkastellaan seuraavaksi KRA-algoritmin muiden suoritusvaiheiden aikakompleksisuutta.

Triviaalin tapauksen ratkaisemiseksi kutsutaan kertaalleen proseduuria *TutkiOsite*. Koska triviaalia tapausta ratkaistaessa roskamäärä on korkeintaan 1, joudutaan proseduurista *TutkiOsite* tekemään korkeintaan kaksi proseduurin *TutkiYksiDiagonaali* kutsua. Proseduurin *TutkiYksiDiagonaali* suorittaminen yhden kerran vaatii aikaa $O(n + m)$, sillä kummankin syötevektorin tarkasteltava osite voidaan joutua käymään läpi lopusta alkuun monotonisesti. Tämän jälkeen valitaan PYA-jonoon ositteen kaikki merkit mahdollisesti yhtä lukuun ottamatta, jolloin syötejono X on käytävä kertaalleen

läpi. Triviaalin ongelman ratkaisemisen kompleksisuus on siten kokonaisuudessaan suuruusluokkaa $O(n + m)$. Menetelmän pseudokoodi löytyy liitteestä kohdasta 11.3.2.

Esimerkki 3.26: *KRA:n löytämä PYAn ratkaisupolku esimerkin 3.1 syötejonoille.*



Täydellisen leikkauksen laskeva proseduuri *EtsiTäydellinenLeikkaus* kutsuu kahdesti proseduuria *TutkiOsite*, jossa selvitetään alku- ja loppuliitteiden $L_i(k)$-arvot roskamäärän ollessa φ. Täydellisen leikkauksen selvittäminen vie siten aikaa $O(n\varphi)$. Alkuperäiselle ongelmalle tähän tarvitaan $O(n(m–p))$:n suuruinen aika. Seuraavilla kerroilla ositteen roskamäärä pienenee aina puoleen edellisen kutsun roskamäärästä. Jos koko ongelman roskamäärä on kakkosen potenssi, rekursiotasoja on $\log \varphi$ kappaletta. Roskamäärä saa rekursion edetessä siten selvästikin arvot φ, $\varphi/2$, $\varphi/4$, … . Laskemalla yhteen lukujonon termit saadaan tulokseksi geometrinen sarja, jonka summaksi tulee 2φ. Koska roskamäärän puolittuminen puolittaa myös ongelman ratkaisemiseen kuluvan ajan, muodostuu täydellisen leikkauksen etsinnän kustannukseksi $2\varphi n$. Edelleen, koska jo PYAn pituuden ratkaisemiseen tarvittiin aikaa $O(n(m–p))$, voidaan havaita, että PYAn pituuden laskemisen jälkeiset suoritusvaiheet eivät enää lisää algoritmin asymptoottista suoritusaikaa. Siten Kumarin ja Ranganin algoritmin kokonaissuoritusajaksi määräytyy $O(n(m–p))$, joka on identtinen NKY:n aikakompleksisuuden kanssa. Vaikka KRA:n ja NKY:n aikakompleksisuudet ovatkin samaa suuruusluokkaa, PYAn rekursiivisen ratkaisemisen KRA:ssa pitäisi johtaa käytännössä tuntuvasti pidempiin suoritusaikoihin NKY:hyn verrattuna.

Algoritmissa varataan tilaa syötevektoreiden lisäksi täydellisten leikkauspisteiden laskentaa varten tarvittaville vektoreille *alkuosa* ja *loppuosa*. Lisäksi tarvitaan tilaa PYA-jonoa *S* varten. Kyseisten tietorakenteiden vaatima tila on suuruusluokkaa $O(n+m)$. Koska sekä PYAn pituuden laskenta että täydellisten leikkausten määrääminen tapahtuu ilman NKY:ssä käytettävää matriisia, pysyy koko algoritmin tilakompleksisuus lineaarisena eli termin $O(n+m)$ mukaisena. NKY:hyn verrattuna KRA:n voisi siten olettaa suoriutuvan tehtävästään selkeästi vähemmin muistiresurssein.

3.4 Monisuuntaisesti prosessoivat PYA-algoritmit

Käsillä olevan pääluvun aliluvuissa 3.1 – 3.3 tarkasteltiin PYA-algoritmeja, jotka käsittelevät syötevektoreidensa indeksipareja havainnollistavaa kaksiulotteista matriisia *M* joko yksi rivi, korkeuskäyrä tai diagonaali kerrallaan. Seuraavassa kuvataan kaksi PYA-algoritmia, jotka käsittelevät kyseistä matriisia *sekä rivi- että sarakesuuntaisesti*. Tähän ryhmään luokitellaan kuuluviksi Bonnin yliopistossa kehitetyt kaksi menetelmää: *Rickin I algoritmi* (RI1) [Ric94] sekä lineaarisessa muistitilassa toimiva *Goemanin ja Clausenin algoritmi* (GCL) [Goe99].

3.4.1 Rickin I algoritmi (RI1)

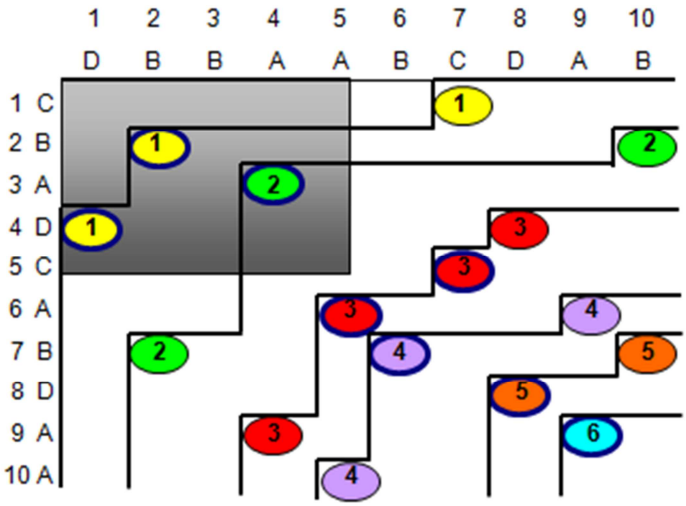
3.4.1.1 Uusi teoreettinen käsite: minimaalinen todistaja …

Claus Rick esitteli vuonna 1994 teknisessä raportissaan [Ric94] kaksi PYA-algoritmia. Näistä ensimmäinen kantaa hänen julkaisussaan nimeä *Algoritmi 3*, ja siitä käytetään tässä työssä lyhennettä *RI1*. Algoritmi on alun perin suunniteltu parantamaan riveittäin prosessoivien HSZ- ja AG2-algoritmien suorituskykyä.

Rick kiinnittää raportissaan huomiota kahteen tehottomuuden lähteeseen Huntin ja Szymanskin algoritmissa. Näistä ensimmäinen on ongelman kaikkien täsmäysten tarkasteleminen pelkkien dominanttien täsmäysten sijaan. Tästä hidastuttavasta piirteestä Apostolicon ja Guerran II algoritmissa onkin jo päästy eroon. Toisena heikkona kohtana Rick mainitsee systemaattisesti jokaista täsmäystä kohti toistettavan puolitushakuoperaation kynnysarvovektorissa. Hän tarjoaa lääkkeiksi näiden pulmien ratkaisemiseksi menetelmiä, joiden avulla pystytään paitsi rajoittamaan tarkastelu pelkkiin dominantteihin täsmäyksiin, niin myös *tunnistamaan tietyn luokan D_k kaikkien täsmäysten jo löytyneen*. Viimeksi mainittua piirrettä ei ole sisältynyt vielä mihinkään edellä esiteltyyn PYA-algoritmiin. Ennen algoritmin tärkeimpien piirteiden tarkempaa esittelyä tarkastellaan kuitenkin kahta tekijän teoreettista havaintoa.

Aliluvussa 2.3.1 ehdittiin jo todistaa, että PYAn ratkaisemiseksi voidaan rajoittua tarkastelemaan yksinomaan *dominantteja täsmäyksiä*. Rick tuo artikkelissaan kuitenkin ilmi, *ettei välttämättä niistäkään kaikkia tarvita ongelman ratkaisemiseksi*, vaan tietyt dominanteista täsmäyksistä ovat toisia tärkeämpiä. Hyvänä johdatuksena tämän väitteen sanoman ymmärtämiselle toimii seuraava esimerkki. Siinä *oletetaan etukäteen tiedettävän*, että PYAn pituus on 6.

Esimerkki 3.27: *Minimaalisten todistajien erottaminen muista dominanteista täsmäyksistä.* X = ”CBADCABDAA”, Y = ”DBBAABCDAB”, $p = 6$, $m = n = 10$, PYA = ”BAABDA”.



Esimerkissämme x_1:n eli ”C”:n vasemmanpuoleisin esiintymä vektorissa Y sijaitsee sarakkeessa 7, joten paikasta (1, 7) löytyvä 1-täsmäys on selvästikin dominantti. Tästä huolimatta se ei kuitenkaan voi kuulua tarkasteltavan ongelman PYAan, sillä sen oikealla puolella sijaitsee enää kolme jonoon Y kuuluvaa merkkiä, kun niitä oletuksen mukaan tarvitsisi löytyä vielä viisi kappaletta, jotta (1, 7) voisi kelvata PYA-jonon ensimmäiseksi täsmäykseksi! Vastaavanlaisesta syystä myöskään esimerkiksi paikassa (7, 2) sijaitseva dominantti 2-täsmäys ei voi kuulua PYAan, sillä sen jälkeen on enää kolme X:n merkkiä käytettävissä, mikä ei riitä kuuden mittaisen PYAn muodostamiseen kyseisen täsmäyksen kautta.

Täsmäyksen luokan määritelmän perusteella on selvää, että luokkaan k ($1 \leq k \leq p$) kuuluvia (dominantteja) täsmäyksiä voi löytyä aikaisintaan riviltä ja sarakkeelta k[75]. Lisäksi, jotta dominantti k-täsmäys (i, j) voisi kuulua tarkasteltavan PYA-ongelman ratkaisuun, eli sen perään mahtuisi vielä riittävä määrä — $p–k$ kappaletta — merkkejä kummastakin syötejonosta, pitää sen rivi- ja sarakeindeksien täyttää myös seuraavat *ylärajaehdot*: $i \leq k + m – p$ ja $j \leq k + n – p$. Niitä dominantteja k-täsmäyksiä, joiden rivi- ja sarakeindeksit täyttävät kyseiset vaatimukset, kutsutaan luokan k *minimaalisiksi todistajaksi* (engl. *minimal k-witness*). Jokaisesta C_k-luokasta löytyy väistämättä ainakin yksi minimaalinen todistaja. Muutoinhan ongelman PYA ei

[75] Muussa tapauksessa X:n tai Y:n alkuliitteen pituus, joka on k:ta pienempi, olisi riittämätön.

sisältäisi yhtään luokan C_k edustajaa, mikä on vastoin PYAn määritelmää. Mitä pidempi tarkasteltavan ongelman PYA on suhteessa syötevektoreiden pituuteen, sitä matalammaksi ja kapeammaksi puristuu rivi- ($k \le i \le k+m-p$) ja sarakealueista ($k \le j \le k+n-p$) muodostuva suorakulmio, josta luokan k minimaalisia todistajia voi löytyä. Minimaaliset todistajat on esimerkissä 3.27 erotettu paksuin **tummansinisin** reunamerkinnöin muista dominanteista täsmäyksistä. Luokan 1 minimaalisten todistajien mahdollinen esiintymisalue on sävytetty **harmaaksi**. Esimerkissä ainoastaan kaksi PYA-jonoon kuulumatonta dominanttia täsmäystä – paikoissa (4, 1) ja (5, 7) sijaitsevat – ovat minimaalisia todistajia.

Äskeisen graafisen esityksen perusteella tuntuisi ilmeisen tavoiteltavalta, että PYA-algoritmi pystyisi erottelemaan minimaaliset todistajat muista dominanteista täsmäyksistä. Asian tekee kuitenkin hankalaksi käsitteen riippuvuus PYAn pituudesta p, joka on juuri se tuntematon parametri, jonka arvo algoritmin on tarkoitus laskea! Myöhemmin luvussa 6 esitellään tutkimusryhmäni kehittämä tekniikka, jonka avulla ainakin osa turhiksi osoittautuvista dominanttitäsmäyksistä voidaan jättää tutkimatta.

Rick käyttää hyväksi tietoa minimaalisten todistajien olemassaolosta kussakin luokassa k määrätessään PYAn pituuteen perustuvaa ylärajaa dominanttien täsmäysten kokonaismäärälle tarkasteltavassa ongelmassa. Oletettakoon, että paikassa (i, j) sijaitsee dominantti k-täsmäys, joka on samalla minimaalinen todistaja. Tällöin voi esiintyä korkeintaan $i–k$ kappaletta saman luokan dominantteja täsmäyksiä, jotka sijaitsevat rivin i yläpuolella. Vastaavasti rivin i alapuolella niitä voi esiintyä korkeintaan $j–k$ kappaletta, eli tarkalleen jokaisessa sarakkeessa väliltä $k..j–1$. Siten dominantteja k-täsmäyksiä voi esiintyä enintään $(i–k)+(j–k)+1$ kappaletta täsmäys (i, j) mukaan lukien. Koska oletuksen mukaan (i, j) on minimaalinen todistaja, sen alin mahdollinen sijaintirivi on $m–(p–k)$. Vastaavasti oikeanpuoleisin sarake, jossa (i, j) voi sijaita, on $n–(p–k)$. Sijoittamalla kyseiset arvot i:n ja j:n paikalle saadaan luokan k dominanttien täsmäysten enimmäismäärälle d ylärajaksi siten $[(m–(p–k))–k]+[((n–(p–k)–k]+1 = m+n–2p+1$. Koska luokkia k on yhteensä p kappaletta, saadaan tulokseksi $d \le p(m+n–2p+1)$. Kyseinen yläraja on tosin kovin väljä, jos p on lyhyt ja $n >> m$. Aikaisemman, aliluvussa 2.3.1 tehdyn analyysin perusteella myös termi $pm–½p(p–1)$ kelpaa d:lle ylärajaksi, sillä ko. lauseke saadaan laskemalla summa $m+(m–1)+ \ldots +(m–p+1)$, jossa k. yhteenlaskettava $(m–k+1)$ kuvaa luokan i dominanttien täsmäysten enimmäismäärää. Tämä yläraja on puolestaan väljä, jos $p \approx m \approx n$. Ottamalla näiden kahden lausekkeen $p(m+n–2p+1)$ ja $pm–½p(p–1)$ *minimi* päästään usein lähemmäksi d:n todellista arvoa kuin yksittäisellä lausekkeella.

3.4.1.2 … ja uusi laskentamalli: vuoron perään rivi ja sarake kerrallaan

Vaikka RI1-algoritmi ei kykenekään tunnistamaan löytämäänsä täsmäystä minimaaliseksi todistajaksi, se pystyy kuitenkin selvittämään, *milloin jonkin luokan kaikki dominantit täsmäykset on jo ehditty löytää*. Etenemällä vuoron perään ensinnä riviä i ja sitten saraketta i pitkin ($1 \leq i \leq m$) ennen riville i+1 ja sittemmin sarakkeelle i+1 siirtymistä taataan, että kunkin luokan k ($1 \leq i \leq p$) dominantit täsmäykset löydetään *ei-vähenevien minimi-indeksien*[76] *mukaisessa järjestyksessä*. Luokan k korkeuskäyrän ominaisuuksien perusteella[77] tiedetään, että sille kuuluvista dominanteista täsmäyksistä ylimpänä sijaitsevalla on suurin sarakeindeksi, ja vastaavasti niistä vasemmanpuoleisimmalla on suurin rivi-indeksi. Kun on kyse luokasta D_k, on sen kaikkien täsmäysten sekä rivi- että sarakeindeksien oltava $\geq k$. Mikäli pisteessä (k, k) sijaitsee k-täsmäys, se on samalla luokkansa ainoa dominantti täsmäys.

Oletetaan nyt, että korkeuskäyrälle k kuuluu vähintään kaksi dominanttia täsmäystä, jotka sijaitsevat pisteissä (k, u) ja (v, k) ($k < u \leq n$, $k < v \leq m$). Tällöin ne ovat mainitussa järjestyksessä algoritmin laskentatavan ansiosta *ensimmäiset* RI1:n löytämät luokan D_k täsmäykset, sillä RI1 etsii niitä ensiksi riviltä k ja seuraavaksi sarakkeelta k. Kun myöhemmin rivi $k+h$ ($h > 0$) otetaan aikanaan käsittelyyn, ei siltä voida enää löytää dominanttia k-täsmäystä, jonka sarakeindeksi t olisi pienempi kuin $k+h$, sillä kyseinen k-täsmäys olisi löydetty jo käsiteltäessä sarake $t < k+h$. Tästä voidaan päätellä, että algoritmin rivi- ja sarakelaskurin i arvon kasvaessa kasvaa myös löydettävien D_k-täsmäysten minimi-indeksi, joka tarkoittaa riviä prosessoitaessa rivinumeroa ja saraketta prosessoitaessa sarakenumeroa. Samalla niiden *maksimi-indeksit*[78] pienenevät, kun k. korkeuskäyrä siirtyy rivillä i vasemmalle tai sarakkeella i ylöspäin. Tosin sanoen rakenteilla olevan k. korkeuskäyrän osat alkavat lähestyä toisiaan laskurin i arvon kasvaessa. Kun lopulta laskuri i saavuttaa arvon z, missä $z = Min\{maxind(f, g) \mid (f, g) \in D_k\}$, on viimeinenkin luokan D_k täsmäyksistä löydetty. Tätä aikaisemmin on välttämättä myös jo kaikkien edellisten korkeuskäyrien $l < k$ pisteet löydetty, sillä niille kuuluvat täsmäykset sijaitsevat jokaisella rivillä luokan k pisteiden vasemmalla puolella ja jokaisella sarakkeella niiden yläpuolella.

Edellisen perusteella RI1-algoritmi löytää riviä i prosessoidessaan kaikki sellaiset korkeuskäyrän k pisteet (i, j), missä $i \leq j$. Nämä pisteet sijaitsevat syötevektoreista muodostuvan matriisin *M päälävistäjällä tai sen yläpuolella*. Vastaavasti ne D_k-täsmäykset, joiden rivinumero on sarakenumeroa suurempi, löytyvät prosessoitaessa saraketta j. Kyseiset pisteet sijaitsevat matriisin M päälävistäjän alapuolella. Seuraava esimerkki valaissee asiaa käytännössä.

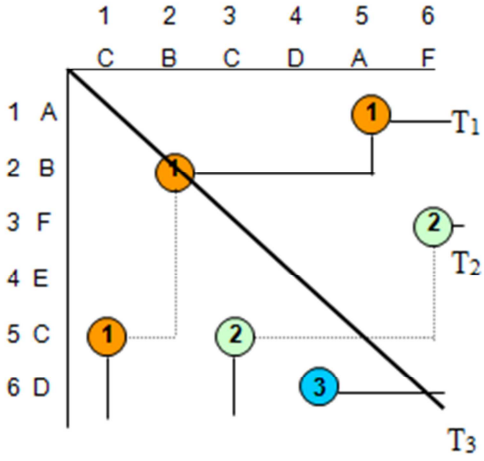
---

[76] Määritellään, että täsmäyksen (i, j) *minimi-indeksi* = $minind(i, j) = i$, jos $i \leq j$, ja muulloin j.

[77] Korkeuskäyrien ominaisuuksia käsiteltiin aliluvussa 2.3.2.

[78] Määritellään, että täsmäyksen (i, j) *maksimi-indeksi* on $maxind(i, j) = i$, jos $i \geq j$, ja muulloin j.

Esimerkki 3.28: $X[1..m]$ = ”ABFECD…” $Y[1..n]$ = ”CBCDAF…”.



Esimerkin D_1-täsmäykset löytyvät järjestyksessä (1,5) riviltä 1, (5,1) sarakkeesta 1 ja (2,2) riviltä 2. Nyt ovat kaikki D_1-täsmäykset löytyneet, sillä saraketta 2 tutkittaessa etsinnät kohtaavat, ja päädiagonaali leikkaa käyrän T_1 pisteessä (2,2) = (k+1, k+1). Vastaavasti T_2 saadaan valmiiksi 4. sarakkeen tultua tutkituksi, sillä 5. rivin täsmäys(5,3) onjo ehditty löytää saraketta 3 tutkittaessa, ja $Min\{maxind(3,6), maxind(5,3)\}$ = 5. Päädiagonaali leikkaa nyt T_2:n sen vaakasuoran osan pisteessä (5,5) = (k+3, k+3).

Tieto luokan k kaikkien dominanttien täsmäysten löytymisestä on algoritmissa hyödyllinen, sillä tämän jälkeen ei enää tarvitse testata luokkien 1..k kynnysarvoja. Algoritmissa muuttuja k sisältää tiedon alimmasta *aktiivisesta täsmäysluokasta*, eli sellaisesta, jota ei ole vielä käsitelty loppuun. Mitä varhaisemmassa vaiheessa korkeuskäyrät saadaan valmiiksi, sitä nopeammin voidaan kynnysarvovektoreiden alkupään positioita ohittaa laskennassa. Kannattaa huomioida, että RI1 tarvitsee käyttöönsä kaksi kynnysarvovektoria. Ensimmäinen niistä eli *rivikynnys* tarvitaan perinteiseen tapaan eli rivien suuntaisesti prosessoimalla löydettyjen dominanttien täsmäysten sarakenumeroiden kirjaamiseen ja toinen, *sarakekynnys*, sarakkeittain prosessoiden löydettyjen dominanttien täsmäysten rivinumeroiden kirjaamiseen.

Riviltä i aloitetaan täsmäysten tutkiminen merkin x_i ensimmäisestä esiintymästä, jonka Y-indeksi $j \geq i$. Indeksiä verrataan riveittäisen kynnysarvovektorin arvoon paikassa k, joka siis edustaa alinta mahdollisesti vielä keskeneräistä korkeuskäyrää. Tällöin on erotettava kolme tapausta. Jos osoittautuu, että 1) $j < rivikynnys[k]$, löytyy uusi luokan k dominantti täsmäys, jota varten perustetaan uusi linkkisolmu PYAn ratkaisupolkujen ylläpitoa varten. Linkkivektorin päivityksiä joudutaan jälleen viivyttämään KCR:n tapaan, koska prosessoinnin suunta on vasemmalta oikealle. *Rivikynnys*[k]:n uudeksi arvoksi asetetaan j, ja etsitään riviltä i seuraava täsmäys *rivikynnys*[k]:ssa ennen päivitystä sijainneen indeksiarvon jälkeen, sillä kaikki sitä edeltävät rivin täsmäykset ovat nyt ei-dominantteja k-täsmäyksiä, eli niiden yli voidaan huoletta hypätä. Seuraavan täsmäyksen sijaintipaikka löytyy vakioajassa *rivin suuntaisesta* lähiesiintymätaulukosta[79]. Jos edellä toteutui kuitenkin 2) $j = rivikynnys[k]$, löydettiin rivin ainoa luokan k täsmäys, ja se on ei-dominantti, minkä johdosta etsitään rivin seuraava täsmäys. Viimeisenä mahdollisuutena on, että 3) $j > rivikynnys[k]$. Tällöin joudutaan etsimään lineaarihaulla *rivikynnys*-vektorista pienin indeksi, jolle on voimassa $j \leq rivikynnys[k]$ ja jatketaan tilanteen mukaan kuten

[79] Jos syöttöaakkosto on kooltaan iso, Rick suosittaa käytettäväksi lähiesiintymävektoria.

tapauksessa 1 tai 2. Rivin *i* käsittely päättyy, kun lähiesiintymätaulukosta haku palauttaa ensi kerran pysäytysalkiota kuvaavan indeksin *n*+1.

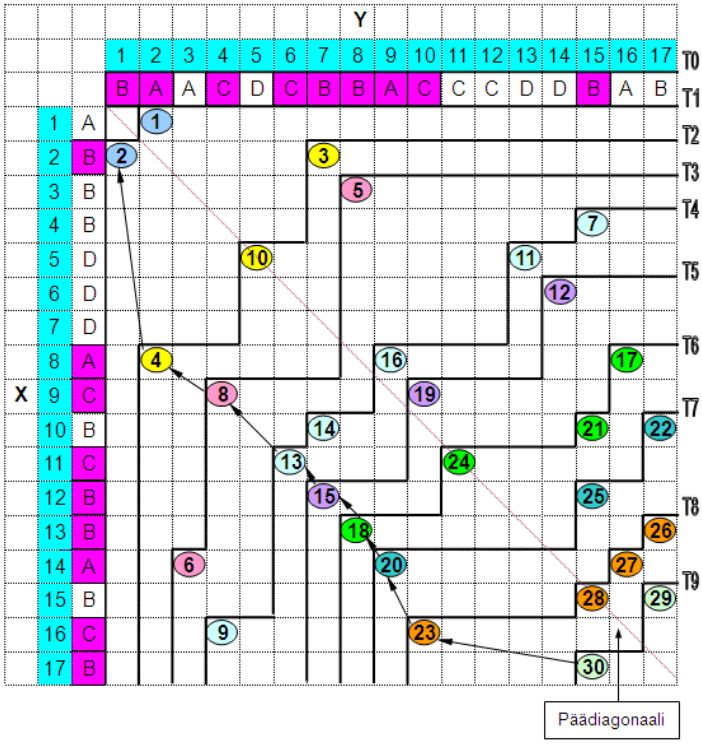
Sarakkeella *i* edetään analogisesti rivin käsittelyn kanssa. Vektorin *rivikynnys* tilalle vain on ilmestynyt *sarakekynnys*, ja merkin y_i esiintymät sarakkeelta *i* löytyvät tarkoitusta varten perustetusta *sarakkeen suuntaisesta* lähiesiintymätaulukosta (tai -vektorista). Siinä missä *rivikynnys*[*k*]:hon tallennettiin luokan *k vasemmanpuoleisimman* täsmäyksen *Y*-indeksi riviin *i* mennessä *päädiagonaalilla tai sen yläpuolella*, tallentuu *sarakekynnys*[*k*]:hon luokan *k ylimmän* täsmäyksen *X*-indeksi *päälävistäjän alapuolella*. Täsmäyksen sijaintia sarakkeella *i* kuvaa *j*:n sijaan muuttuja nimeltä *rivi*. Rivin käsittelyssä esiintyneille tapauksille 1) – 3) on täsmälleen samat vastineet myös sarakkeen käsittelyssä; jos *rivi* < *sarakekynnys*[*k*], löytyy dominantti *k*-täsmäys; jos *rivi* = *sarakekynnys*[*k*], löytyy pelkästään yksi ei-dominantti *k*-täsmäys; jos *rivi* > *sarakekynnys*[*k*], kyseessä on jonkin *k*:ta ylemmän luokan täsmäys. Pysäytysalkiona sarakesuuntaisessa lähiesiintymätaulukossa toimii *m*+1.

Ennen kunkin rivin *i* käsittelyä algoritmi tutkii vielä, onko alimman aktiivisen luokan *k* dominanttia täsmäystä jo ehditty löytää riviltä *i* sarakekohtaisessa prosessoinnissa. Jos näin on tapahtunut, on luokan *k* kaikki dominantit täsmäykset jo löydetty, eli vastedes etsitään vain luokan *k*+1 ja sitä seuraavien luokkien täsmäyksiä. Graafisesti esitettynä tämä tarkoittaa, että päädiagonaali leikkaa korkeuskäyrän *k* sen *vaakasuoralla osuudella*. Vastaavasti ennen kunkin sarakkeen *i* tutkimista testataan, onko jo kyseiseltä sarakkeelta ehditty löytää D_k-täsmäystä rivin suuntaisessa käsittelyssä. Myös tuolloin luokan D_k kaikki täsmäykset on löydetty, ja sen käsittely voidaan lopettaa tarpeettomana. Graafisesti ilmaistuna tämä tarkoittaa, että päälävistäjä leikkaa korkeuskäyrän *k* sen *pystysuoralla osuudella* tai *kulmapisteessä*.

3.4.1.3 Algoritmin analyysi

Seuraavassa esitetään esimerkki, joka havainnollistaa, missä järjestyksessä RI1 löytää dominantit täsmäykset esimerkin 3.1 mukaisille syötejonoille. Liitteen kohdassa 11.4.1 esitetty menetelmää kuvaava pseudokoodi palauttaa PYAlle aina mahdollisemman lähellä matriisin vasenta alakulmaa sijaitsevan esiintymän, joka voidaan koota dominantteja täsmäyksiä käyttämällä, eli se löytää PYAlle saman ratkaisun kuin HI1.

Esimerkki 3.29: *Dominanttien täsmäysten löytymisjärjestys RI1-algoritmissa sekä sen palauttama PYAn instanssi esimerkin 3.1 mukaisille syötteille.*



Rickin I algoritmissa perustetaan esiprosessointivaiheessa kummankin syötevektorin suuntaiset lähiesiintymätaulukot. Näiden alustamiseen kuluu aikaa yhteensä $O(\sigma m) + O(\sigma n) = O(\sigma n)$. Lisäksi on tarpeen alustaa kumpikin kynnysarvovektori, mikä onnistuu ajassa $O(m)$. Siten koko esiprosessointivaiheen aikakompleksisuus on $O(\sigma n)$[80].

Varsinainen laskentavaihe tapahtuu silmukoissa *S1* – *S3*, joista *S2* ja *S3* sijaitsevat toisiinsa nähden peräkkäin pääsilmukan *S1* sisällä. Pääsilmukka huolehtii yhteisen rivi- ja sarakelaskurin *i* arvon kasvattamisesta 1:stä *m*:ään, joten sitä suoritetaan aina *m* kierrosta. Sen sisällä suoritettavat *S2*:n ja *S3*:n ulkopuoliset lauseet ovat vakioaikaisia. Osassa niistä testataan, onko tietyn korkeuskäyrän kaikki täsmäykset löydetty, ja osaa tarvitaan linkkivektoriviittausten päivittämiseen. Silmukassa *S2* käydään läpi kaikki rivin *i* dominantit täsmäykset indeksistä *i* alkaen, ja silmukka *S3* tekee vastaavan työn sarakkeen *i* dominanteille täsmäyksille indeksistä *i* + 1 lähtien.

Tarkasteltaessa yhtä ulomman silmukan kierrosta havaitaan, että sen aikana sekä *S2*:ssa että *S3*:ssa voidaan kummassakin tehdä korkeintaan *p* kierrosta, sillä joka kerta, kun näissä silmukoissa otetaan uusi täsmäys käsittelyyn, kasvaa myös rivi- tai sarakekohtaisen kynnysarvovektorin laskuri ykkösen verran. Jos rivillä on täsmäyksiä enemmän kuin *p* kappaletta — sanokaamme yhteensä *z* täsmäystä — tällöin niistä

[80] Jos lähiesiintymätaulukot korvataan lähiesiintymävektoreilla, esiprosessoinnin kustannus pienenee $O(n)$:ään.

vähintään $z–p$ jää tutkimatta ei-dominantteina täsmäyksinä. Siten silmukoiden *S1 – S3* kokonaistyömäärä voidaan lausua termillä $O(pm)$.

Vaihtoehtoinen yläraja kolmen silmukan kokonaistyömäärälle saadaan tarkastelemalla, miten monta kertaa kutakin yksittäistä kynnysarvovektorien positiota k ($1 \le k \le p$) voidaan enintään päivittää. Selvästikään silmukan *S1* kierroksilla 1..k–1 ei päivityksiä kyseiseen indeksipaikkaan tehdä. Tarkasteltaessa minimaalisia todistajia todettiin, että PYAn pituuden ollessa p pitää luokan k dominantin täsmäyksen löytyä viimeistään riviltä $k+(m–p)$ ja sarakkeelta $k+(n–p)$. Jos nyt sarake $k+(n–p) \le m$, niin silloin päädiagonaali leikkaa k. korkeuskäyrän viimeistään *S1*:n laskurin eli i:n arvolla $k+n–p$ eli pisteessä $(k+n–p, k+n–p)$, koska viimeisenä löydettävä D_k-täsmäys on paikassa $Min\{maxind(f, g) \mid (f, g) \in D_k\}$. Tätä suuremmilla i:n arvoilla ei kynnysarvovektoreiden positiota k enää tutkita, joten indeksin k tutkimiskertojen enimmäismääräksi saadaan tällöin $k+n–p–k+1 = n–p+1$ kertaa. Jos puolestaan $k+(n–p) > m$, joudutaan luokan k täsmäyksiä etsimään pahimmassa tapauksessa kierroksesta k alkaen silmukan *S1* loppuun asti eli yhteensä $m–k+1$ kierrosta, jolloin myös saadaan $k+n–p > m \Leftrightarrow m–k < n–p \Leftrightarrow m–k+1 < n–p+1$. Yhtä kynnysarvovektorien positiota k voidaan siis joutua tutkimaan enintään $n–p+1$ kertaa, joten kaikkia positioita yhteensä tutkitaan enintään $p(n–p)+p$ eli $O(p(n–p))$ kertaa.

PYA-jono pystytään palauttamaan ajassa $O(p)$ seuraamalla linkkivektorin osoittimia positiosta p lähtien alkuun päin. Algoritmin aikakompleksisuus on kokonaisuudessaan siten $O(n\sigma + Min\{mp, p(n–p)\})$[81].

Algoritmi tarvitsee ei-vakiotilaisen määrän muistia syötevektoreidensa lisäksi kummallekin lähiesiintymätaulukolle, joiden koot ovat $O(n\sigma)$ ja $O(m\sigma)$, sekä kummallekin $O(m)$-mittaiselle kynnysarvovektorille. Näistä termi $O(n\sigma)$ on dominoiva. Lisäksi jokainen löydetty dominantti täsmäys pitää tallentaa linkkivektoriin, jonka tilantarve solmuineen on $O(d)$. Algoritmin kokonaistilantarve on siten $O(n\sigma+d)$[82]. Liitteestä löytyvää RI1:n pseudokoodia on täydennetty *linkkisolmujen käsittelyllä*, sillä Rick ei artikkelissaan esitä mitään ehdotelmaa ratkaisuksi kelpaavan PYA-jonon keräämistä ja palauttamista varten.

Mikäli RI1-algoritmista halutaan saada aikaan *lineaaritilainen versio*, joka säilyttää menetelmän asymptoottisen aikakompleksisuuden ja joka pidentää sen käytännön ajoaikaakin enintään kaksinkertaiseksi, pitää algoritmin laskenta muuttaa *kaksisuuntaiseksi*. *Rickin lineaaritilaisessa algoritmissa* [Ric00_1] on ajatuksena rakentaa korkeuskäyriä sekä vasemmasta ylä- että oikeasta alakulmasta aloittaen. Eri suunnista muodostettavat korkeuskäyrät alkavat vähitellen lähestyä ja lopulta sivuta tai leikata toisiaan matriisin keskustassa. Siinä vaiheessa, kun kummastakaan suunnasta edettäessä ei enää löydetä ylimmän luokan korkeuskäyrälle pisteitä, jotka eivät olisi olleet saavutettavissa jo toisesta kulmasta lähtien etsittäessä, on PYAn pituus tiedossa.

---

[81] Jos käytetään lähiesiintymävektoria, vastaava lauseke on $O(Min\{mp \log n, p(n–p) \log n\})$.

[82] Lähiesiintymävektoria käytettäessä tilantarve on $O(n+d)$.

Tällaisen keskimmäisen täsmäyksen suhteen alkuperäinen ongelma voidaan osittaa kahdeksi pienemmäksi. Koska edellä mainitussa artikkelissa ei kuitenkaan ole annettuna menetelmän pseudokoodia ja koska seuraavaksi esiteltävä *Goemanin ja Clausenin algoritmi* perustuu juuri kaksisuuntaiseen laskentaan ja antaa siten menettelystä selkeän kuvan aina rekursiivista jonon palautusvaihetta myöten, ei Rickin lineaaritilaista algoritmia käsitellä tässä työssä tarkemmin enää erikseen.

3.4.2 Goemanin ja Clausenin algoritmi (GCL)

3.4.2.1 Yleistä

Viimeisenä varsinaisena PYA-algoritmina esitellään *Goemanin ja Clausenin algoritmi* [Goe99]. Se julkistettiin vuonna 1999, ja siitä käytetään tässä työssä lyhennettä *GCL*. Algoritmi esitellään nyt varsin perusteellisesti sen sisältämien monien uusien teknisten havaintojen ja yksityiskohtien vuoksi, sekä eritoten siksi, että menetelmän pitäisi teoreettisten ominaisuuksiensa perusteella olla erittäin kilpailukykyinen sekä suoritusajaltaan että muistinkäytöltään. Perinpohjaista käsittelyä puoltanee myös algoritmin monimutkainen toteutus, jonka ymmärtäminen on työlästä ilman lukuisia, havainnollistavia esimerkkejä. Lisäksi tekijöiden esittämän pseudokoodin paikoittaiset puutteet tekevät algoritmin ohjelmoinnista haasteellisen ja visaisen tehtävän.

Koska GCL on kehitetty samassa yliopistossa kuin edellä esitelty Rickin I algoritmi, on varsin ymmärrettävää, että GCL on perinyt edeltäjältään muutamia oleellisia piirteitä. Ensiksikin, kumpainenkin menetelmä suorittaa PYAn etsimiseksi *sekä rivi- että sarakesuuntaista* laskentaa. Myös menetelmien käyttämät aputietorakenteet muistuttavat toisiaan: sekä RI1 että GCL käyttävät laskennassa avuksi sekä rivi- että sarakesuuntaisia *lähiesiintymätaulukoita* ja *kynnysarvovektoreita*. Edelleen, kumpikin menetelmä pystyy havaitsemaan, milloin rivillä tai sarakkeella tarkasteltavan *korkeuskäyrän k kaikki pisteet tiedetään jo löydetyiksi*, eli milloin tietyn korkeuskäyrän pisteiden etsintä voidaan lopettaa tarpeettomana.

Selkein ero menetelmien RI1 ja GCL toimintatavoissa liittyy etenemiseen syötejonoja pitkin. Siinä missä RI1 käsittelee kumpaakin syötejonoa *X* ja *Y* yksistään niiden alusta loppuun päin, GCL tarkastelee kumpaakin jonoa *myös päinvastaiseen suuntaan*: *lopusta alkuun päin*. Jotta dominanttien täsmäysten etsintä tapahtuisi vakioajassa siitä riippumatta, edetäänkö riviä vai saraketta pitkin ja onko prosessoinnin suunta alusta loppuun vai lopusta alkuun päin, GCL tarvitsee avukseen *neljää eri lähiesiintymätaulukkoa ja kynnysarvovektoria*: yhden kutakin mahdollista matriisin dimensio- (rivi/sarake) ja etenemissuuntayhdistelmää (eteen-/taaksepäin) varten.

3.4.2.2 Tietorakenteet

Taulukoiden *LähiEsTauluY*[1..σ, 1..n+1] ja *LähiEsTauluX*[1..σ, 1..m+1] merkitys on täysin sama kuin Rickin I algoritmissa. Paikkaan *LähiEsTauluY*[x_i, j] tallennettu arvo u kertoo, miltä sarakkeelta $u \geq j$ — ts. mistä vektorin Y indeksistä — löytyy merkin x_i lähin esiintymä indeksistä j lähtien. Vastaavasti paikassa *LähiEsTauluX*[y_j, i] esiintyvä arvo v ilmaisee, miltä riviltä $v \geq i$ — ts. mistä vektorin X indeksistä — löytyy merkin y_j lähin esiintymä indeksistä i lähtien. Algoritmissa GCL tarvitaan näiden lisäksi vielä lähiesiintymätaulukoita *LähiEsTauluY**[1..σ, 0..n] sekä *LähiEsTauluX**[1..σ, 0..m]. Näistä ensin mainitussa soluun (x_i, j) tallennettu arvo u osoittaa, mistä vektorin Y paikasta $u \leq j$ löytyy merkin x_i viimeinen sellainen esiintymä, jonka indeksi on korkeintaan yhtä suuri kuin j. Vastaavasti *LähiEsTauluX**[y_j, i] = v tarkoittaisi, että v on suurin sellainen indeksi $\leq i$, josta löytyy symboli y_j vektorista X. Seuraava esimerkki havainnollistaa, miltä käännetystä syötevektorista muodostettu taulukko *LähiEsTauluY** näyttäisi syötejonolle Y = ”DBBAABCDAB”.

Esimerkki 3.30: *Käännetyn lähiesiintymätaulukon LähiEsTauluY**[0..n] *sisältö Goemanin ja Clausenin algoritmissa, kun n = 10, syötejonona Y esiintyy* ”DBBAABCDAB” ja Σ = {“A”, “B”, “C”, “D”}.

	D	B	B	A	A	B	C	D	A	B
	0	1	2	3	4	5	6	7	8	9	10
A	0	0	0	0	4	5	5	5	5	9	9
B	0	0	2	3	3	3	6	6	6	6	10
C	0	0	0	0	0	0	0	7	7	7	7
D	0	1	1	1	1	1	1	1	8	8	8

*LähiEsTauluY**[“A”..”D”, 0..n]

Kuten jo edellä todettiin, GCL-algoritmissa muodostetaan myös neljä erillistä *kynnysarvovektoria*, jotka tässä työssä on nimetty tunnuksin *Rivikynnys*, *Sarakekynnys*, *Rivikynnys** ja *Sarakekynnys**. Näistä kaksi ensin mainittua ovat käyttötarkoitukseltaan samoja kuin algoritmissa RI1, mutta erona on kuitenkin niille varatun muistitilan koko. Rickin I algoritmissa niille varataan kummallekin m+1 muistipaikkaa, eli vektoreiden pituus ylittää yhdellä PYAn teoreettisen maksimipituuden. Sen sijaan Goemanin ja Clausenin algoritmissa näille kahdelle kynnysarvovektorille varataan muistista vain $\lceil m/2 \rceil$+1 paikkaa. Tämä perustuu siihen, että GCL-algoritmin pääsilmukkaa suoritetaan ainoastaan $\lceil m/2 \rceil$ kertaa, kun taas RI1:n pääsilmukassa tarkastellaan kaikki rivit 1..m. GCL:ssä rivit $\lceil m/2 \rceil$+1..m käsitellään *käänteisessä järjestyksessä*, jolloin matriisin oikeasta alanurkasta alkavien korkeuskäyrien tiedot tallennetaan vektoreihin *Rivikynnys** ja *Sarakekynnys**. Kummassakin algoritmissa kynnysarvovektoreiden nollatta indeksipaikkaa käytetään pysäytysalkion tallentamiseen.

3.4.2.3 Pääsilmukan toiminta ja lyhyt pseudokoodi

Suorituksen eteneminen riveillä ja sarakkeilla

Oletetaan seuraavaksi, että m on pariton. Käsiteltäessä riviä i ($1 \le i \le \lceil m/2 \rceil$) alusta loppuun, paikkaan *Rivikynnys*[k] ($0 \le k \le \lceil m/2 \rceil$) tallennettu arvo u ($0 \le u \le n + 1$) osoittaa, *millä sarakkeella korkeuskäyrä k leikkaa pystysuunnassa rivin i*. Ellei rivillä i esiinny laisinkaan korkeuskäyrää k, tulee *Rivikynnys*[k]:n arvoksi $n + 1$. Vastaavalla tavalla, käsiteltäessä saraketta j ($1 \le j \le \lceil (m-1)/2 \rceil$), paikasta *Sarakekynnys*[k] ($0 \le k \le \lceil (m-1)/2 \rceil$) löydettävä arvo v ($0 \le v \le m$) ilmaisee, *millä rivillä korkeuskäyrä k leikkaa vaakasuorassa sarakkeen j*[83]. Ellei tällaista riviä ole olemassa — ts. korkeuskäyrää k ei esiinny sarakkeella j — *Sarakekynnys*[k] saa arvon m+1 sarakkeella j. Sitä mukaa kun rivi- ja sarakenumerot kasvavat käsiteltäessä niitä alusta loppuun päin, kynnysarvovektoreihin tallentuvat arvot indeksipaikoissa $1..\lceil m/2 \rceil$ joko pysyvät ennallaan tai pienenevät. Sekä *Rivikynnys*[0] että *Sarakekynnys*[0] alustetaan pseudoarvoon 0, sillä nollan mittaisen PYAn muodostamiseksi riittää selvästikin jo kaksi tyhjää merkkijonoa.

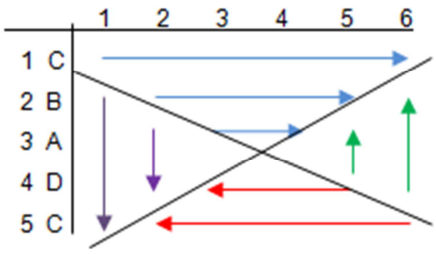
Rivit $\lceil m/2 \rceil+1..m$ algoritmi GCL käsittelee yksinomaan lopusta alkuun päin. Tällöin vektoriin *Rivikynnys**[k] ($0 \le k \le \lfloor m/2 \rfloor$) tallennettava arvo u ($0 \le u \le n+1$) pääsilmukan indeksiarvolla i ($1 \le i \le \lfloor m/2 \rfloor$) osoittaa, millä sarakkeella korkeuskäyrä k leikkaa pystysuunnassa rivin $m+1-i$. Ellei tällaista saraketta ole olemassa, paikassa *Rivikynnys**[k] esiintyy mainitulla rivillä oltaessa alustusarvo 0. Paikkaan *Sarakekynnys**[k] tallennettu arvo v kuvaa puolestaan analogisesti, millä rivillä korkeuskäyrä k leikkaa vaakasuorassa sarakkeen $n+1-j$. Jos tällaista riviä ei ole määritelty, esiintyy *Sarakekynnys**[k]:n arvona alustusarvo 0. Positiot *Rivikynnys**[0] ja *Sarakekynnys**[0] alustetaan pseudoarvoilla $n + 1$ ja $m + 1$, sillä nollan mittaisen PYAn muodostamiseksi ei tarvita syötejonojen lopusta ainoatakaan merkkiä. Sitä mukaa kun pääsilmukan indeksi i kasvaa – eli lopusta alkuun päin tarkasteltavien rivien ja sarakkeiden numerot pienenevät – näihin kahteen kynnysarvovektoriin tallentuvat arvot indeksipaikoissa $1..\lfloor m/2 \rfloor$ joko pysyvät ennallaan tai kasvavat.

Goemanin ja Clausenin algoritmi on toteutettu artikkelissa [Goe99] siten, että rivien ja sarakkeiden tarkastelu aloitetaan *oikeasta alakulmasta*, mikäli lyhyemmän syötejonon X pituus m on *parillinen*, ja *vasemmasta yläkulmasta*, jos se on *pariton*. Tällä tavoin taataan, että jokainen syötejonon X riveistä tulee tutkituksi tarkalleen kertaalleen. Olettaen, ettei kyseessä ole algoritmin pääsilmukan ensimmäinen suorituskierros X:n pituuden m ollessa pariton, GCL-algoritmissa käsitellään yhden suorituskierroksen aikana ensinnä *alin toistaiseksi tutkimaton rivi* ja *oikeanpuoleisin yhä tutkimaton sarake*, ja niiltä etsitään *tarkastelusuunnasta katsottuna* kaikki

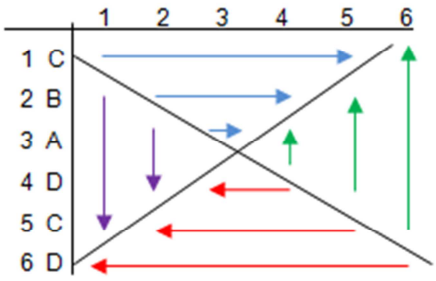
[83] Silloin, kun rivien määrä on pariton, lopetetaan sarakkeiden käsittely ylhäältä alaspäin jo sarakkeen $\lceil (m-1)/2 \rceil$ tultua käsitellyksi (koko matriisi tulee riittämiin selatuksi rivin $\lceil m/2 \rceil$ tutkimisen jälkeen).

dominantit täsmäykset[84]. Tämän jälkeen siirrytään tutkimaan syötteen vasemmasta yläkulmasta lukien *ylintä riviä* ja *vasemmanpuoleisinta saraketta*, joita ei vielä ole ehditty käsitellä[85] ja etsitään vuorostaan niiltä kaikki dominantit täsmäykset. Jokainen pääsilmukan suorituskierros kasvattaa ylimmän tarkasteltavan rivin ja vasemmanpuoleisimman sarakkeen numeroa ykkösellä, ja ellei kyseessä ollut silmukan ensimmäinen kierros kun *m* on pariton, lisäksi alimman tarkasteltavan rivin ja oikeanpuoleisimman sarakkeen numeroa pienennetään yhdellä. Täten jokaisen suorituskierroksen jälkeen tutkimattoman alueen koko kutistuu reunoiltaan sekä kahdella rivillä että sarakkeella[86]. Seuraava esimerkki auttaa lukijaa hahmottamaan GCL-algoritmin rivien ja sarakkeiden prosessointijärjestystä.

Esimerkki 3.31: *Alueiden tutkimisjärjestys GCL-algoritmissa, kun rivien määrä on 1) pariton tai 2) parillinen*



1) Rivi 1: sarakkeet 1-6 { (1,1), (1,2), ..., (1,6) }
2) Sarake 1: rivit 2-5 { (2,1), (3,1), (4,1), (5,1) }
3) Rivi 5: sarakkeet 6-2 { (5,6), (5,5), ... , (5,2) }
4) Sarake 6: rivit 4-2 { (4,6), (3,6), (2,6) }
5) Rivi 2: sarakkeet 2-5 { (2,2), (2,3), (2,4), (2,5) }
6) Sarake 2: rivit 3-4 { (3,2), (4,2) }
7) Rivi 4: sarakkeet 5-3 { (4,5), (4,4), (4,3) }



1) Rivi 6: sarakkeet 6-1 { (6,6), (6,5), ..., (6,1) }
2) Sarake 6: rivit 5-1 { (5,6), (4,6), ..., (1,6) }
3) Rivi 1: sarakkeet 1-5 { (1,1), (1,2), ... , (1,5) }
4) Sarake 1: rivit 2-5 { (2,1), (3,1), (4,1), (5,1) }
5) Rivi 5: sarakkeet 5-2 { (5,5), (5,4), (5,3), (5,2) }
6) Sarake 5: rivit 4-2 { (4,5), (3,5), (2,5) }
7) Rivi 2: sarakkeet 2-4 { (2,2), (2,3), (2,4) }
8) Sarake 2: rivit 3-4 { (3,2), (4,2) }
9) Rivi 4: sarakkeet 4-3 { (4,4), (4,3) }

GCL-algoritmi jakaa syötteistä muodostuvan matriisin *M* selvästikin neljään osaan, joiden rajoina esiintyvät esimerkissä 3.31 nähtävissä olevat lävistäjät. Menetelmä pitää huolen siitä, ettei mitään aluetta tutkita useampaan otteeseen kuin kertaalleen. Kulloinkin tarkasteltavalla alueella pysähdytään ainoastaan sellaisten solujen kohdalla, joissa esiintyy täsmäys, mutta algoritmi ei kuitenkaan takaa, että täsmäys olisi tarkastelusuunnasta katsottuna välttämättä aina dominantti.

---

[84] Kannattaa huomioida, että täsmäys, joka on dominantti prosessoitaessa syötteitä alusta loppuun päin, ei välttämättä ole dominantti täsmäys edettäessä lopusta alkuun päin – ja päinvastoin.

[85] Silloin, kun *m* on pariton, käsitellään uloimman silmukan ensimmäisellä suorituskierroksella ainoastaan ylin rivi ja vasemmanpuoleisin sarake. Alkuperäisartikkelin [Goe99] mukaisessa pseudokoodissa tehdään tuolloin hyppy pääsilmukan rungon sisään. Tämä joudutaan kiertämään sopivaa kontrollirakennetta käyttäen kielissä, joissa keskelle toistorakenteen runkoa hyppääminen ei ole laillista.

[86] Jos *m* on pariton, alue kutistuu silmukan ensimmäisen kierroksen jälkeen vain yhdellä rivillä ylhäältä ja yhdellä sarakkeella vasemmalta. Vastaavasti viimeisellä kierroksella tutkitaan vain rivi $\lceil m/2 \rceil$.

Koska Goemanin ja Clausenin algoritmin oikeellisuuden ymmärtäminen vaatii analyyttistä otetta, joudutaan sen rivi- ja sarakeoperaatioita sekä PYAn ratkaisuksi kelpaavan jonon määräämistä tarkastelemaan yksityiskohtaisesti. Jotta lukija saisi kuitenkin yleiskäsityksen algoritmin päävaiheista ilman rekursiivista jonon palautusvaihetta, esitetään seuraavassa GCL:n tiivistetty pseudokoodi ennen tarkempaa algoritmin logiikkaan perehtymistä. Pseudokoodista esitetään myöhemmin liitteessä kohdassa 11.4.2 täydellisempi versio, jossa huomioidaan myös operaatiot PYAn ratkaisuksi kelpaavan jonon etsimiseksi rekursiivisesti.

ALGORITMI GCL (X, m, Y, n, σ) (**karkean tason kuvaus):**

Perusta vektoreiden X ja Y lähiesiintymätaulukot molempisuuntaisesti.
Perusta ja alusta kynnysarvovektori jokaista matriisin neljännestä kohti.
Alusta muuttujat matriisin eri ositteiden PYAn pituudet D_Y (ylä), D_V (vasen), D_Δ (ala) ja D_O (oikea) nolliksi ja eri kulmista lähtien etsittävät alimmat korkeuskäyrät T_{YV} (ylävasemmalta) ja $T_{\Delta O}$ (alaoikealta) ykkösiksi.
ylinrivi:= 1; /* Ylimmän tarkasteltavan rivin määräämiseen tarvittava laskurimuuttuja. */
alinrivi := m; /* Alimman tarkasteltavan rivin määräämiseen tarvittava laskurimuuttuja. */
vasensarake := 1; /* Vasemmanpuoleisimman tarkasteltavan sarakkeen osoittava laskurimuuttuja. */
oikeasarake := n; /* Oikeanpuoleisimman tarkasteltavan sarakkeen osoittava laskurimuuttuja. */
IF m on pariton
    Hyppää algoritmissa riville **YV**.[87]
S1: WHILE ylinrivi ≤ alinrivi /* Etsitään matriisista täsmäyksiä vuorotellen eri ositteista. */
    A: Etsi riviltä alinrivi sarakkeilta oikeasarake..vasensarake oikealta vasemmalle edeten täsmäyksiä luokkiin $T_{\Delta O}$, …, D_Δ+1 ja päivitä tarvittaessa alaositteen kynnysarvovektoria Rivikynnys*.
    IF alueen PYA pitenee eli löytyy luokan D_Δ + 1 täsmäys
        D_Δ := D_Δ + 1; /* Kasvatetaan alaositteen PYAn pituutta yhdellä. */
            IF Sarakekynnys[D_V] ≥ alinrivi /* Testataan, onko täsmäys vertailukelvoton vasemman ositteen ylimmän täsmäysluokan vasemmanpuoleisimman edustajan kanssa. */
                D_V := D_V – 1; /* On: lyhennetään PYAn pituutta vasemmassa ositteessa.
    Etsi sarakkeelta oikeasarake riveiltä alinrivi–1..ylinrivi alhaalta ylös edeten täsmäyksiä luokkiin $T_{\Delta O}$, …, D_O+1 ja päivitä tarvittaessa oikean ositteen kynnysarvovektoria Sarakekynnys*.
    IF alueen PYA pitenee eli löytyy luokan D_O + 1 täsmäys
        D_O := D_O + 1; /* Kasvatetaan oikean ositteen PYAn pituutta yhdellä. */
            IF Rivikynnys[D_Y] ≥ oikeasarake /* Testataan, onko täsmäys vertailukelvoton yläositteen ylimmän täsmäysluokan alimman edustajan kanssa. */
                D_Y := D_Y – 1; /* On: lyhennetään PYAn pituutta yläositteessa.
    /* Testataan, voidaanko alaoikealla kulkeva korkeuskäyrä $T_{\Delta O}$ terminoida. */
    IF (Rivikynnys*[$T_{\Delta O}$] = oikeasarake) /* Luokka $T_{\Delta O}$ on täyttynyt alaositetta tutkittaessa? */
        IF (D_O < $T_{\Delta O}$) /* Oikeassa ositteessa ei yhtään täyttyneen luokan täsmäystä? */
            IF Rivikynnys[D_Y] ≥ oikeasarake /* Yläos. ylimmän luokan alin täsmäys vertailukelvoton? */
                D_Y := D_Y – 1; /* On: lyhennetään PYAn pituutta yläositteessa.
            ELSE
                D_O := $T_{\Delta O}$; /* Korjataan D_O ajan tasalle. */
            $T_{\Delta O}$:= $T_{\Delta O}$ + 1; /* Alimman oikeasta alakulmasta tarkasteltavan täsmäysluokan päivitys. */

[87] Hyppy silmukan sisään on laiton operaatio useimmissa ohjelmointikielissä, ja se pitää tässä tapauksessa kiertää estämällä silmukan *S1* alkuosan suoritus ehtolauseella tultaessa silmukkarakenteeseen ensimmäistä kertaa samalla, kun *m* on pariton. Alkuperäisartikkelissa hyppykäskyä on kuitenkin käytetty tässä esitetyllä tavalla.

ELSE IF (Sarakekynnys*[T_{AO}] = alinrivi) /* Luokka T_{AO} täyttynyt oikeaa ositetta tutkittaessa? */
IF (D_A < T_{AO}) /* Alaositteessa ei yhtään täyttyneen luokan täsmäystä. */
IF Sarakekynnys[D_V] ≥ alinrivi /* Vasemman ositteen ylimmän luokan oikeanpuoleisin täsmäys vertailukelvoton? */
D_V := D_V – 1; /* On: lyhennetään PYAn pituutta vasemmassa ositteessa.
ELSE
D_A := T_{AO}; /* Korjataan D_A ajan tasalle. */
B: T_{AO} := T_{AO} + 1; /* Alimman tarkasteltavan täsmäysluokan päivitys oik. alakulm. lähtien. */
alinrivi := alinrivi – 1; /* Leikataan alin rivi … */
oikeasarake := oikeasarake – 1; /* … ja oikeanpuoleisin sarake jatkotarkastelujen ulkopuolelle. */
YV: Suorita vastaavat toimenpiteet kuin edellä nimiöiden A ja B väliin jäävillä riveillä, mutta vaihda seuraavat muuttujat keskenään ja huomioi tarkastelun kulkusuunnan vaihtuminen edellä esitetyn peilikuvaksi.
- Rivikynnys* ←→ Rivikynnys /* Tutkitaan rivejä yläositteesta, ei alaositteesta. */
- Sarakekynnys* ←→ Sarakekynnys /* Tutkitaan sarakkeita vasemmalta, ei oikealta. */
- D_A ←→ D_Y, D_O ←→ D_V, T_{AO} ←→ T_{YV} /* Ylimpien täsmäysluokkien vastinmuuttujat. */

ylinrivi := ylinrivi + 1; /* Leikataan ylin rivi … */
vasensarake := vasensarake + 1; /* … ja vasemmanpuoleisin sarake jatkotarkastelujen ulkopuolelle. */
ENDWHILE (S1)

/* Määrätään PYAn pituus. */
IF ((D_Y > D_V) **AND** (D_A > D_O)) /* Yläositteessa pidempi PYA kuin vasemmassa ositteessa ja alaositteessa pidempi PYA kuin oikeassa ositteessa? */
IF (T_{YV} ≤ D_V) /* Kyllä. Onko T_{YV}:n arvo enintään D_V:n suuruinen? */
T_{YV} := D_V + 1; /* On: asetetaan T_{YV} tarvittaessa yhtä isommaksi kuin D_V. */
IF (T_{AO} ≤ D_O) /* Entä onko T_{YV}:n arvo enintään D_V:n suuruinen? */
T_{AO} := D_O + 1; /* On: asetetaan T_{AO} tarvittaessa yhtä isommaksi kuin D_O. */
u := D_Y; /* Aloitetaan ei-terminoitujen käyrien vertailu yläositteen ylimmästä … */
v := T_{AO}; /* … ja alaositteen alimmasta luokasta, jota ei löydy oikeasta ositteesta. */
WHILE niin kauan kuin (u ≥ T_{YV}) **AND** (v ≤ D_A) /* Jatketaan, kunnes käyrät loppuvat. */
IF (Rivikynnys[u] ≥ Rivikynnys*[v]) /* Ovatko käyrät vertailukelvottomat? */
u := u – 1; /* Kyllä: PYAn pituus ei muutu, sillä summa u + v ei muutu. */
v := v + 1; /* Siirrytään alaositteessa luokkaa ylemmäs. */
p := u + D_A; /* PYAn pituus on u + alaositteen PYAn pituus. */
ELSE /* (D_Y ≤ D_V) tai (D_A ≤ D_O) */
p := Max{D_V + D_A, D_Y + D_O}; /* PYA saadaan kulkemalla vasemmalta alas tai ylhäältä oikealle. */
Tulosta PYAn pituus p.
END (ALGORITMI GCL).

Tarkasteltaessa Rickin I algoritmia [Ric94] todettiin, että kaikki tiettyyn luokkaan *k* kuuluvat dominantit täsmäykset ovat löytyneet, kun tarkasteltavan rivin tai sarakkeen laskuri saavuttaa arvon, joka on $z = Min\{maxind(f, g) \mid (f, g) \in D_k\}$[88]. Koska RI1:ssä tarkasteltavien rivien ja sarakkeiden numerot ovat järjestyksessä ei-vähéneviä, laskurin arvosta *z* lähtien löydetään täsmäyksiä pelkästään paikoista (z+u, z+v), missä $u, v \geq 0$. Kyseiset täsmäykset – matriisin päädiagonaalin pistettä (z, z) lukuun ottamatta – eivät sijaintinsa perusteella selvästikään voi olla enää luokan *k* dominantteja täsmäyksiä, sillä edellä tehdyn oletuksen mukaan jo ennen rivin tai viimeistään ennen sarakkeen *z*

[88] kts. aliluku 3.4.1.2

tutkimista löydetty k-täsmäys (f, g), missä f, $g \leq z$, oli dominantti, ja dominantin täsmäyksen määritelmän perusteella kaikille muille luokkaan k kuuluville dominanteille täsmäyksille (q, r) pitää olla voimassa joko (($q < f$) $\wedge$ ($r > g$)) tai (($q > f$) $\wedge$ ($r < g$)). Paikassa (z+u, z+v) sijaitsevat täsmäykset eivät siten enää täytä määritelmän mukaista kriteeriä, kun u tai $v > 0$.

GCL-algoritmissa jokaista tarkasteltavaa rivinumeroa i kohti ($1 \leq i \leq \lceil m/2 \rceil$) tutkittava matriisin yläpuoliskon sarakealue on $i..n$+1–i, mikäli m on pariton. Sarakealueen kutistuminen kummastakin reunasta rivilaskurin i kasvaessa yhdellä on ilmeistä algoritmin laskentatavan ansiosta, sillä rivin i käsittelyä seuraavat järjestyksessä sarakkeen i, rivin m+1–i ja sarakkeen n+1–i tarkastelu ennen riville i+1 siirtymistä. Etsittäessä täsmäyksiä sarakkeelta i osuu matkan varrelle myös matriisin solu (i+1, i), joten rivin i+1 tarkastelun alkaessa kyseisen solun uudelleen tutkiminen olisi turhaa. Vastaavasti saraketta n+1–i läpi käytäessä osutaan tarpeen vaatiessa soluun (i+1, n), joten selvästikin rivin i+1 tarkastelua voidaan myös oikeasta reunasta rajoittaa yhden sarakkeen verran. Samankaltainen tilanne toistuu aina i:n arvon kasvaessa yhdellä. Jos puolestaan m on parillinen, rivillä i sarakkeiden tarkastelu rajoittuu indekseihin $i..n–i$, eli rivillä i tarkasteltava alue on oikeasta reunastaan yhdellä kapeampi kuin m:n ollessa pariton. Syynä tähän on algoritmin suorituksen käynnistyminen oikeasta alakulmasta, jolloin rivi m+1–i ja sarake n+1–i ehditään tutkia ennen riville i siirtymistä. Tällöin mahdollisesti solussa (i, n+1–i) esiintyvä täsmäys kirjataan jo ennen rivin i käsittelyä, joten rivin i oikeaksi rajaksi voidaan tällöin turvallisesti asettaa $n–i$ n+1–i:n sijaan. Yleisesti, saavuttaessa tarkastelemaan mitä tahansa matriisin M riviä (tai saraketta), tarkastelun ulkopuolelle jätetään kaikki ne solut, joihin on pystytty etenemään jo aikaisemmin prosessoimalla matriisia sarakkeittain (tai riveittäin). Edellä esitetty esimerkki 3.31 kuvaa rivien ja sarakkeiden tarkastelualueita sen mukaan, onko lyhyemmän syötevektorin X pituus m pariton vai parillinen. Kuvassa nuolen suunta esittää prosessoinnin etenemissuuntaa. Kannattaa huomioida, että i:n saavuttaessa maksimiarvonsa $\lceil m/2 \rceil$ jää sarake $\lceil m/2 \rceil$ tarkastelematta sen indeksirajojen muodostaessa jo tyhjän alueen. Samoin käy sarakkeen n+1–$\lceil m/2 \rceil$ tarkastelulle silloin, kun m on pariton.

Riippumatta siitä, tarkastellaanko paraikaa matriisin riviä vai saraketta ja siitä, kumpaan suuntaan ollaan etenemässä, GCL pyrkii löytämään tarkastelualueeltaan dominantteja täsmäyksiä aina niiden kasvavan luokkanumeron mukaan aloittamalla ensimmäisestä sellaisesta luokasta, jota ei vielä ole ehditty *terminoida* kaikkien siihen kuuluvien täsmäysten löytymisen takia. Viimeksi loppuun käsitelty dominanttien täsmäysten luokka määräytyy kuitenkin erikseen prosessoinnin alkukohdan mukaan, ja *alimmista vielä tarkasteltavista täsmäysluokista* käytetään merkintöjä T_{YV} (ylhäältä vasemmalta) ja T_{AO} (alhaalta oikealta), ja näille on voimassa $1 \leq T_{YV}, T_{AO} \leq i$.[89] Vastaavasti *ylin täsmäysluokka*, johon edustajia voidaan tarkasteltavalta riviltä tai

[89] Muuttujien T_{YV} ja T_{AO} alkuarvoksi asetetaan ykkönen.

sarakkeelta etsiä, on ykkösen verran suurempi kuin alueelta tarkasteluhetkeen mennessä löydetyn PYAn pituus. Ylin täsmäysluokka määräytyy erikseen sekä kullekin dimensiolle että etenemissuunnalle, ja siitä käytetään merkintöjä D_Y (riveittäin ylhäältä), D_V (sarakkeittain vasemmalta), D_A (riveittäin alhaalta) ja D_O (sarakkeittain oikealta). Kaikkien neljän viimeksi mainitun muuttujan alkuarvoksi asetetaan nolla.

Korkeuskäyrien etsintä ja terminointi

Olettaen, että m on pariton ja algoritmin pääsilmukan suoritus on juuri käynnistynyt, etsitään aluksi symbolin x_1 *ensimmäistä esiintymää* Y:stä. Jos tällainen on olemassa, se on samalla selvästikin dominantti 1-täsmäys. Täsmäyksen sarakeindeksi tallennetaan muistipaikkaan *Rivikynnys*[1]. Ellei merkkiä x_1 esiinny kertaakaan vektorissa Y, ei mainittua kynnysarvovektoria päivitetä. Koska riviltä 1 ei voida löytää korkeampien luokkien täsmäyksiä, siirrytään seuraavaksi tarkastelemaan saraketta 1, jolta etsitään merkin y_1 *ensimmäistä esiintymää* X:ssä riviltä 2 lähtien[90]. Jos sellainen löytyy, se kirjataan muistiin paikkaan *Sarakekynnys*[1]. Tähän asti laskenta on edennyt aivan kuin algoritmissa RI1. Nyt tiet kuitenkin eroavat, sillä GCL ei siirrykään riville 2 kuten RI1, vaan se lähtee seuraavaksi etsimään merkin x_m *viimeistä esiintymää* vektorista Y. Esiintymä – olettaen, että sellainen löytyy sarakkeilta $2..n$[91] – kirjataan muistiin paikkaan *Rivikynnys**[1]. Myös tämä on dominantti täsmäys, kun tarkastelu aloitetaan oikeasta alakulmasta. Sama pätee merkin y_n *viimeiseen esiintymään* vektorissa X, ja tieto sen sijaintirivistä tallennetaan muuttujaan *Sarakekynnys**[1], kunhan se löytyy riveiltä $2..m–1$.[92]

Kun matriisista on ehditty tutkia uloimmat rivit ja sarakkeet, siirrytään tarkastelemaan matriisin toiseksi uloimpia rivejä ja sarakkeita edellä kuvatussa järjestyksessä. Näillä voi esiintyä myös luokan 2 dominantteja täsmäyksiä, jos edellisellä kierroksella vastaavalla etenemistavalla[93] löydettiin 1-täsmäys. Myös dominanttien 1-täsmäyksien etsintää jatketaan, ellei luokkaa 1 ehditty *terminoida* jo edellisellä kierroksella. Näin olisi käynyt, jos olisi löytynyt dominantti 1-täsmäys joko paikasta (1, 1) tai (m, n). Ensin mainitussa tapauksessa olisi jatkossa jo turhaa etsiä 1-täsmäyksiä matriisin vasemmasta yläkulmasta lukien, ja jälkimmäisessä tapauksessa niiden etsintä oikeasta alakulmasta alkaen olisi tarpeetonta. Tällöin tilanteen mukaan joko muuttujan T_{YV} tai T_{AO} – tai mahdollisesti jopa molempien – arvoa olisi kasvatettu 1:stä 2:een merkkinä siitä, että luokka 1:n käsittely on jo saatu päätökseen

[90] Jos symboli y_1 olisi myös X:ssä heti ensimmäisenä, merkkien muodostama täsmäys olisi löytynyt jo edellä riviltä 1.

[91] Paikassa (m, 1) sijaitseva täsmäys olisi tarvittaessa jo kirjattu käsiteltäessä saraketta 1, eli tarkalleen silloin, kun se on sarakkeen 1 ainoa 1-täsmäys rivin 1 alapuolella. Jos sarakkeelta 1 löytyi luokan 1 dominantti täsmäys tätä ylempää, voidaan mahdollinen täsmäys (m, 1) ohittaa ei-dominanttina.

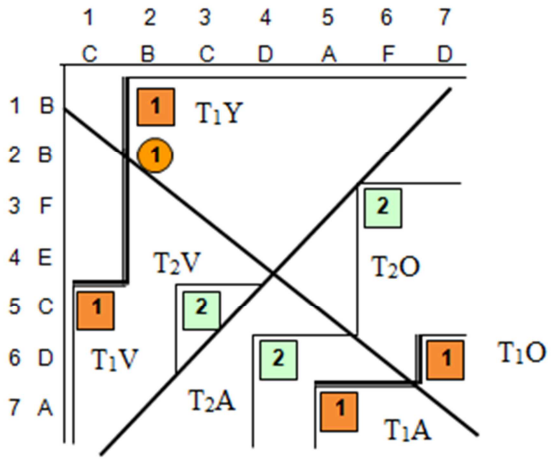
[92] Solut (1, n) ja (m, n) on jo tutkittu rivien 1 ja m tarkasteluissa edellä, mikäli se oli tarpeen. Kannattaa huomioida, että jos solussa (1, n) sijaitsee sarakkeen n ainoa 1-täsmäys, mutta riviltä 1 löydettiin dominantti 1-täsmäys jo sarakkeesta, jonka indeksinumero $j < n$, ei täsmäyksen (1, n) kirjaaminen ole enää hyödyllistä, sillä se on ylhäältä alaspäin tarkastelussa ei-dominantti 1-täsmäys, jonka (1, j) peittää.

[93] Saraketarkasteluissa 2-täsmäyksiä voi löytyä myös silloin, jos 1-täsmäys löytyi rivitarkastelulla joko paikasta (1, 1) tai (m, n). Tällöin luokka 1 ehti jo terminoitua tarkastelusuunnastaan.

asianomaisesta tarkastelusuunnasta. Kuten jo edellä mainittiin, täsmäysluokan k terminointi tapahtuu silloin, kun pääsilmukan laskuri on saavuttanut vasemmasta yläkulmasta aloitettaessa arvon $z = Min\{maxind(i, j) \mid (i, j) \in D_k\}$. Jos aloitus on tapahtunut oikeasta alakulmasta, luokka k täyttyy analogisesti, kun $z^* = 1 + Min\{Max\{m - i, n - j\} \mid \forall(i, j) \in D_{k^*}\}$[94].

Seuraava esimerkki 3.32 valaisee lukijalle GCL-algoritmin täsmäysten etsintämenettelyä. Korkeuskäyrän T_i perään merkitty lyhenne Y, V, A tai O ilmaisee sen lävistäjien rajaaman matriisin neljänneksen, jonka alueelta korkeuskäyrä on löytynyt.

Esimerkki 3.32: *Täsmäysten etsintästrategia GCL-algoritmissa, kun m on pariton.*



Esimerkissä 3.32 syötejonon X pituus $m = 7$, joten GCL-algoritmissa tarkastellaan ensiksi ylintä riviä 1 kokonaisuudessaan ja tämän jälkeen vasemmanpuoleisinta saraketta paikkaa (1, 1) lukuun ottamatta. Riviltä 1 löytyy merkkien ”B” muodostama luokan 1 dominantti täsmäys paikasta (1, 2), ja tämän jälkeen löydetään sarakkeelta 1 merkkien ”C” dominantti 1-täsmäys paikasta (5, 1). Seuraavaksi löydetään dominantteja 1-täsmäyksiä alimmalta riviltä paikasta (7, 5) ja lopuksi vielä oikeanpuoleisimmasta sarakkeesta paikasta (6, 7). Riviltä 2 löydetään pelkästään ei-dominantti 1-täsmäys paikasta (2, 2), jonka algoritmi kuitenkin rekisteröi. Sen sijaan sarakkeen 2 tarkastelu ei osoittaudu hedelmälliseksi: merkkiä ”B” ei esiinny vektorissa X riveillä 3..6. Koska riviltä 1 löytyneen täsmäyksen (2, 2) ansiosta $Min\{maxind(i, j) \mid (i, j) \in D_1\} = 2$, ei ole enää mahdollista löytää uusia 1-täsmäyksiä matriisin vasemmasta yläkulmasta alkaen etenemällä[95]. Siten luokka 1 voidaan tältä suunnalta *terminoida*, ja muuttujan T_{YV} uudeksi arvoksi asetetaan 2. Oikeasta alakulmasta edettäessä ei riviltä eikä sarakkeelta 6

[94] Dominanttien täsmäysten joukkoon D_k^* kuuluvat kaikki käännettyjen syötejonojen X^* ja Y^* dominantit k-täsmäykset.

[95] GCL-algoritmin rivillä 2 havaitsemaa täsmäystä (2, 2) ei olisi välttämättä edes tarvinnut kirjata, sillä jo dominantin 1-täsmäyksen löytyminen paikasta (1, 2) olisi taannut luokan 1 valmistumisen i:n arvolla 2. Alkuperäisartikkelin versio kirjaa kuitenkin luokan k ei-dominantin täsmäyksen, jos tarkasteltavalla rivillä (sarakkeella) on tarkalleen yksi tällainen täsmäys. Tästä syystä myös joukkomerkintä D_k on GCL:n yhteydessä sikäli harhaanjohtava, että kyseiseen joukkoon otetaan mukaan myös ei-dominantteja täsmäyksiä. Joukon D_k laajentaminen tässä tapauksessa ei kuitenkaan aiheuttane merkittävää sekaannusta.

löydetä uusia 1-täsmäyksiä, mutta sen sijaan löytyy dominantti 2-täsmäys ensiksi paikasta (6, 4) riveittäisessä ja myöhemmin vielä paikasta (3, 6) sarakkeittaisessa tarkastelussa. Myös merkkijonoja lopusta alkuun päin tarkasteltaessa voidaan täsmäysluokka 1 terminoida, sillä $z^* = 2 = 1 + Min\{Max\{m - i, n - j\} \mid \forall(i, j) \in D_{k^*}\}$ paikasta (6, 7) löytyneen täsmäyksen ansiosta. Luokkien terminointia kuvaa esimerkissä kaksinkertaisella viivalla korostettu korkeuskäyrän osuus, joka yhdistää samaisesta kulmasta aloitetut luokkien D_1 ja D_{1^*} täsmäysten rivi- ja sarakesuuntaiset etsinnät.

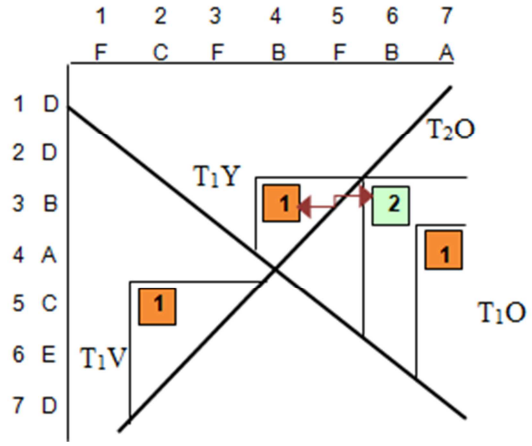
Esimerkin viimeinen dominantti 2-täsmäys (5, 3) löydetään lopulta prosessoitaessa saraketta 3 ylhäältä alaspäin *i*:n arvolla 3. Kannattaa huomioida, että kyseisen täsmäyksen edeltäjänä luokassa D_1 esiintyy (2, 2), joka löydettiin prosessoitaessa riviä 2 — ts. se ei sijaitse samassa lävistäjien jakamassa matriisin *M* etsintäneljänneksessä täsmäyksen (5, 3) kanssa. Sen sijaan ainoa (5, 3):n kanssa samassa neljänneksessä sijaitseva 1-täsmäys (5, 1) ei selvästikään kelpaisi tämän edeltäjäksi *X*:n ja *Y*:n yhteiseen alijonoon, sillä niiden *X*-indeksit ovat samat. Täsmäyksen (5, 3) kelpaaminen luokkaan D_2 selittyy täsmäysluokan D_1 *täyttymisellä ennen rivi-/sarakelaskurin kasvamista arvoon 3*. Jos nimittäin luokka D_{k-1} ($k \geq 1$) on saatu käsiteltyä loppuun[96] ennen riviä tai saraketta *i*, on tällöin oltava voimassa $Min\{maxind(i, j) \mid (i, j) \in D_{k-1}\} < i$. Siten luokassa k–1 esiintyy väistämättä ainakin yksi täsmäys, jonka sekä *X*- että *Y*-indeksit ovat paraikaa tarkasteltavaa rivi- tai sarakenumeroa *i* pienempiä. Tämä havainto on todistettu tekijöiden artikkelin [Goe99] lemmassa 3.1. Samalla perusteella voidaan yleisesti todeta, että missä tahansa matriisin *M* neljänneksessä voi esiintyä rivillä tai sarakkeella *i* luokan D_k ($1 \leq k \leq i$) täsmäys, vaikkei siinä esiintyisi yhtään mielivaltaiseen luokkaan 1..k–1 kuuluvaa täsmäystä. Tällöin on kuitenkin edellytyksenä, että jokainen luokista 1..MAX_l ($1 \leq l < k$), joiden edustajaa ei esiinny tarkasteltavassa neljänneksessä[97], on jo *terminoitu* riville *i* tultaessa.

GCL-algoritmin oikeellisuuden yhtenä peruspilarina on *silmukkainvariantissa taata, etteivät eri suunnista edettäessä muodostuvat korkeuskäyrät risteä eivätkä kohtaa toisiaan yhden rivi-sarakeparin tultua käsitellyksi* jommastakummasta suunnasta algoritmin pääsilmukan yhden kierroksen aikana. Esimerkissä 3.32 ei kyseistä konfliktitilannetta esiintynyt silmukan minkään kierroksen 1..$\lceil m/2 \rceil$ eli 1..4 aikana, mutta katsotaan seuraavaksi esimerkkiä 3.33, jossa silmukkainvariantti tilapäisesti rikkoutuu algoritmin pääsilmukan kierroksen suorituksen ollessa kesken.

[96] Pelkästä pseudotäsmäyksestä (0, 0) koostuvan luokan D_0 oletetaan valmistuneen jo ennen pääsilmukan ensimmäistä suorituskierrosta.

[97] Kvanttorimerkinnällä MAX_l tarkoitetaan tässä suurinta sellaista indeksiä *l*, joka toteuttaa asetetun ehdon [Rai94].

Esimerkki 3.33: *Erisuuntaisten korkeuskäyrien kohtaaminen GCL-algoritmissa yhden rivi-sarakeparin tarkastelun aikana.*



Samoin kuin esimerkissä 3.32, on syötteen *X* pituus *m* nytkin pariton 7. Siten GCL-algoritmi yrittää löytää aluksi merkille x_1 = “D” vasemmanpuoleisinta esiintymää *Y*:ssä sekä merkille y_1 = “F” ylintä esiintymää *X*:ssä, mutta tällaisia ei kuitenkaan löydy. Merkin x_7 = “D” oikeanpuoleisimman esiintymän etsintä ei edellisen perusteella selvästikään voi tuottaa tulosta, mutta paikasta (4, 7) löydetään dominantti 1-täsmäys etsittäessä merkin y_7 = ”A” viimeistä esiintymää *X*:ssä. Rivin 2 selaaminen ei tuota tulosta, joten seuraava onnistuva haku tapahtuu sarakkeella 2, jonka positiosta (5, 2) löytyy toinen dominantti 1-täsmäys. Tämän jälkeen rivi 6 tutkitaan laihoin tuloksin, mutta sen jälkeen löydetään luokan 2 dominantti täsmäys paikasta (3, 6). Täsmäyksen rivi-indeksi 3 ilmaisee samalla, mistä indeksistä alkaa *lyhin X:n loppuliite, joka sisältää Y:n loppuliitteen* ”BA”[98].

Erisuuntaisten korkeuskäyrien kohtaaminen

Kun seuraavaksi lähdetään tutkimaan riviä 3, löydetään sieltä tarkastelualueen ensimmäinen dominantti 1-täsmäys paikasta (3, 4). Nyt havaitaan kuitenkin, että sen oikealla puolella samalla rivillä sijaitsee jo dominantti 2-täsmäys (3, 6), joka ehdittiin löytää selattaessa saraketta 6 alhaalta ylöspäin riveillä 5..3. Siitä huolimatta, että nämä kaksi täsmäystä sijaitsevat tarkasteluhetkellä eri korkeuskäyrillä, ne eivät voi samanaikaisesti kuulua jonojen *X* ja *Y* PYAan, sillä niiden indeksiparit *eivät ole vertailukelpoisia* eli ne eivät ole lajiteltavissa täydelliseen järjestykseen keskenään.

Edellä kuvatun tilanteen hallitsemiseksi — eli keskenään ristiriitaisten korkeuskäyrien uudelleen järjestelemiseksi — GCL-algoritmi testaa aina, kun jollakin matriisin neljänneksistä — merkittäköön tätä aluetta tunnuksella *A* — PYA pitenee yhdellä, onko löydetty täsmäys vertailukelpoinen korkeimman löydetyn täsmäysluokan

[98] Tämä voitaisiin tulkita NKY-algoritmissa käytetyn merkinnän $L_i(k)$ käänteiskuvaukseksi, joka vastaa kysymykseen $L_z(k) = n + 1 - k$. Merkityksenä olisi: *”Mistä alkaa lyhin X:n loppuliite, joka sisältää Y:n viimeiset k merkkiä alijononaan?*”. Merkinnässä vain *z* esiintyy ratkaistavan asemassa.

l edustajan kanssa matriisin *siinä viereisessä neljänneksessä B, jossa sekä haun dimensio että suunta ovat päinvastaiset*. Toisin sanoen, jos ollaan prosessoimassa rivi kerrallaan vasemmalta oikealle matriisin yläneljännestä (alue *A*), vertailualueena *B* on matriisin oikea neljännes, jossa edetään sarake kerrallaan lopusta alkuun päin. Tällöin vertailukelpoisuus saadaan selville tutkimalla arvoa *Sarakekynnys**[D_O]. Korkeuskäyrien ristiriitaisuutta testattaessa matriisin *ylä- ja oikea neljännes* muodostavat siten keskenään *vertailuparin*, ja toisen samankaltaisen parin muodostavat matriisin *ala- ja vasen neljännes*.

Mikäli tarkasteltavat kaksi täsmäystä alueilla *A* ja *B* ovat vertailukelpoiset keskenään, ei alueelta *A* löydetyn täsmäyksen kynnysarvovektoriin kirjaamisen ohella tarvitse tehdä mitään erityistä. Tämä tarkoittaa, että alueiden korkeimpien täsmäysluokkien edustajat voivat kuulua samaan PYA-jonoon. Näin käy esimerkissä 3.32, kun vasemmasta neljänneksestä (alue *A*) löydetään ensimmäinen luokan 2 dominantti täsmäys paikasta (5, 3). Sen sarakeindeksiä verrataan matriisin alaneljänneksestä löydetyn 2-täsmäyksen (6, 4) sarakeindeksiin. Koska 3 < 4, täsmäykset voivat kuulua samaan PYA-jonoon[99]. Mutta muussa tapauksessa kaikki aiemmin löydetyt vastinparineljänneksen *B* ylimmän korkeuskäyrän pisteet liitetään alueelle *A* perustetun uuden korkeuskäyrän jatkeeksi. Lisäksi korkeimman täsmäysluokan *l* olemassaolo alueella *B* lakkautetaan, ja sen arvoksi päivittyy *k*–1.

Palataan takaisin esimerkin 3.33 tilanteeseen, jossa matriisin yläneljänneksestä (alue *A*) löydetään 1-täsmäys (3, 4). Koska se on vertailukelvoton oikean neljänneksen (alue *B*) 2-täsmäyksen (3, 6) kanssa, viimeksi mainittu täsmäys siirretään kuuluvaksi luokkaan 1 *samalle korkeuskäyrälle kuin* (3, 4). Samalla alueen *B* korkeimman täsmäysluokan numero päivitetään kakkosesta ykköseen. Tehdyn toimenpiteen laillisuus on esimerkistä melko mutkattomasti todennettavissa intuitiivisesti. Ensinnäkin tiedetään, että kaikkien oikeasta neljänneksestä löytyneiden täsmäysten sarakenumerot ovat suurempia kuin 4 GCL-algoritmin rivien ja sarakkeiden käsittelyjärjestyksen ansiosta. Koska (3, 4) on ensimmäisenä löydetty eli ylin matriisin yläneljänneksen 1-täsmäys, voi tämän oikealta puolelta löytyä muita 1-täsmäyksiä ainoastaan sillä ehdolla, että ne sijaitsevat joko samalla rivillä kuin (3, 4) tai tätä ylempänä, eli alueen *A* korkeuskäyrän jatkeeksi liitettävien alueen *B* täsmäysten on oltava sen kanssa vertailukelvottomia. Esimerkissämme alun perin dominantiksi 2-täsmäykseksi saraketarkastelussa kirjautunut (3, 6) kelpaa selvästikin yläneljänneksen korkeuskäyrälle 1 ei-dominantiksi täsmäykseksi.

Vertailukelvottomuus ei kuitenkaan yksistään ole vielä riittävä ehto korkeuskäyrän pisteiden siirtämiselle toiselle alueelle, vaan lisäksi on varmistuttava siitä, että alueelta *B* liitettävät täsmäykset kuuluvat *B*:n *ylimpään täsmäysluokkaan*. Mietitään mielenkiinnosta esimerkin 3.34 avustamana, mitä olisi tapahtunut, jos esimerkin 3.33 syötejonon *Y*:n neljäs merkki ”B” olisikin vaihtunut ”A”:ksi. Tällöin yläneljänneksen

[99] Pelkän sarakeindeksin testaaminen riittää, sillä saraketta 3 pitkin ei edetä niille riveille asti, joilta on ehditty etsiä täsmäyksiä oikeasta alakulmasta aloittaen.

(alue *A*) ainoa dominantti täsmäys olisi löytynyt rivitarkastelussa vasemmalta oikealle vasta paikasta (4, 4). Nyt oikeasta neljänneksestä (alue *B*) löytyisi kaksi täsmäystä, jotka ovat tämän kanssa vertailukelvottomia, eli pisteet (3, 6) ja (4, 7). Näistä kuitenkin vain alun perin ylimpään eli luokkaan 2 kuulunut täsmäys (3, 6) voidaan liittää samalle korkeuskäyrälle kuin (4, 4), sillä piste (4, 7) on vertailukelpoinen pisteen (3, 6) kanssa ($4 > 3 \wedge 7 > 6$), joten se ei voi kuulua samalle korkeuskäyrälle kuin (3, 6) ja (4, 4).

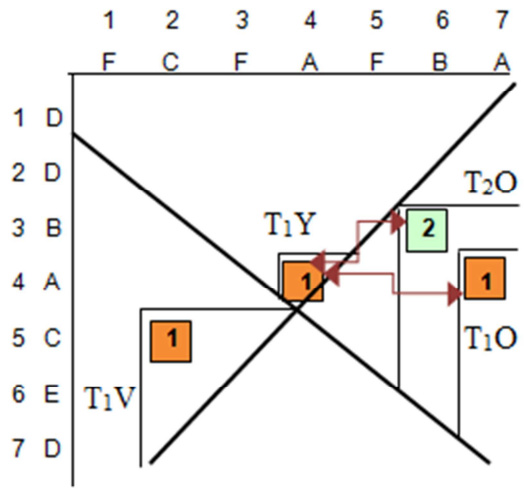
Kannattaa huomioida, että jos oikeassa neljänneksessä useampi kuin yksi täsmäys kuuluu alueen ylimmälle korkeuskäyrälle D_O, jota ollaan liittämässä yläneljänneksen ylimpään korkeuskäyrään D_Y, kaikki luokan D_O täsmäykset siirretään samalla. Näin voidaan turvallisesti tehdä, sillä näistä viimeksi löytyneen rivinumero on kaikkein suurin ja sarakenumero kaikkein pienin, ja siirryttäessä oikealle päin täsmäysten rivinumerot ovat ei-kasvavia.

Erityisesti on huomion arvoista, ettei alueen *B* korkeimman täsmäysluokan numerolla *l* ole täsmäysten siirtohetkellä alueen *A* korkeuskäyrälle mitään merkitystä, kunhan se vain on nollasta eroava. Tämä selittyy sillä, että alueen *B* luokkiin 1..*l*–1 kuuluvat täsmäykset ovat yhä edelleen saavutettavissa alueelle *A* siirrettyjen, alueella *B* aiemmin luokkaan *l* kuuluneiden täsmäysten kautta. Siten GCL toimii oikeellisesti – kasvatettuaan alueen *A* ylimmän korkeuskäyrän numeroa *k* yhdellä – pienentämällä tämän perään alueen *B* vastaavaa arvoa *l* ykkösen verran. Korkeuskäyrän siirron toteutuessa PYAn kokonaispituus ei siten muutu miksikään.

Esimerkki 3.34 havainnollistaa tilannetta. Siinä löydetään matriisin yläneljänneksestä ensimmäinen 1-täsmäys paikasta (4, 4), mutta samalla havaitaan, että oikean neljänneksen ylimmän luokan 2 täsmäys ei sijaitse rivin 4 alapuolella. Täten siirretään oikean neljänneksen 2-täsmäys (3, 6) kuuluvaksi yläneljänneksen luokkaan 1. Kyseisen pisteen kautta on oikean neljänneksen dominantti 1-täsmäys paikassa (4, 7) yhä edelleen saavutettavissa, vaikka se ei kelpaakaan ensimmäisen yläneljänneksestä löytyneen täsmäyksen (4, 4) seuraajaksi.

Edellä keskityttiin tarkastelemaan pelkästään tilannetta, jossa matriisin yläneljänneksestä löydetään ensimmäinen uuden korkeuskäyrän piste, jonka sijaintia verrattiin oikean neljänneksen korkeimman täsmäysluokan pisteen rivinumeroon. Vastaavanlainen tarkastelu pätee *symmetrisesti myös päinvastoin*: aina, kun *oikeasta neljänneksestä* (saraketarkastelussa alhaalta ylöspäin) löydetään ensimmäinen korkeuskäyrän $l = D_O$ piste, pitää selvittää, sijaitseeko yläneljänneksen korkeimman täsmäysluokan D_Y vasemmanpuoleisin löydetty piste *tarkasteltavassa sarakkeessa tai sen oikealla puolella*. Asia selviää tutkimalla arvo paikasta *Rivikynnys*[D_Y]: jos sijaitsee, siirretään yläneljänneksen korkeuskäyrän D_Y pisteet oikean neljänneksen korkeuskäyrän D_O jatkeeksi oikealle ylös, ja pienennetään muuttujan D_Y arvoa yhdellä. Menettely on täysin analoginen edellä esitetyn kanssa: ainoastaan alueet *A* ja *B* vaihtavat tarkastelussa paikkaa keskenään.

Esimerkki 3.34: *Korkeuskäyrien risteäminen GCL-algoritmissa yhden rivi-sarakeparin tarkastelun aikana.*



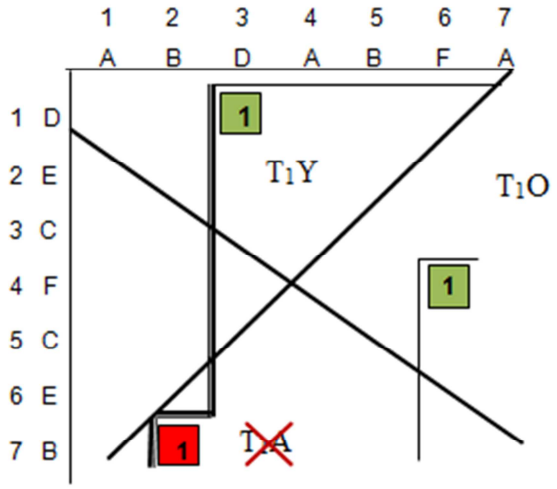
Esimerkissä löydetään järjestyksessä ensin 1-täsmäykset paikasta (4, 7) sarakkeelta 7 ja paikasta (5, 2) sarakkeelta 2. Näiden jälkeen löydetään saraketta 6 tutkittaessa vielä 2-täsmäys paikasta (3, 6), ennen kuin löydetään ensimmäinen ja samalla ainoa 1-täsmäys yläneljänneksestä paikasta (4, 4). Tämä on selvästi vertailukelvoton kummankin oikeasta neljänneksestä löytyvän täsmäyksen kanssa. Niistä kuitenkin vain korkeimpaan luokkaan 2 kuuluva eli (3, 6) siirretään nyt luokan 1 täsmäykseksi yläneljännekseen samalle korkeuskäyrälle kuin (4, 4). Sen sijaan täsmäys (4, 7) jää nykyiselle korkeuskäyrälleen, sillä se kuuluu luokkaan 1 ja on vertailukelpoinen siirrettävän täsmäyksen (3, 6) kanssa.

Edellä ehdittiin todeta, että matriisin ylä- ja oikean neljänneksen ohella myös matriisin *vasen ja alaneljännes* muodostavat keskenään tarkasteluparin. Näilläkin alueilla tutkitaan kutakin riviä ja saraketta käsiteltäessä, kohtaavatko/risteävätkö vastinalueiden korkeuskäyrät vai eivät. Perustettaessa rivitarkastelussa *alaneljännekseen* uusi korkeuskäyrä D_A on tutkittava käyttämällä hyväksi arvoa *Sarakekynnys*[D_V], sijaitseeko vasemman neljänneksen ylimmän luokan D_V täsmäys *tarkasteltavalla rivillä tai sen alapuolella*. Jos tällainen tilanne syntyy, siirretään vasemman neljänneksen korkeuskäyrän D_V pisteet alaneljänneksen korkeuskäyrän D_A jatkeeksi vasemmalle alas, ja pienennetään muuttujan D_V arvoa yhdellä. Kun löydetään puolestaan uusi korkeuskäyrä D_V *vasemmasta neljänneksestä*, pitää selvittää muistipaikasta *Rivikynnys**[D_A], sijaitseeko alaneljänneksen korkeimman luokan D_A dominantti täsmäys *tarkasteltavalla sarakkeella tai sen vasemmalla puolella*. Mikäli näin on, lakkautetaan korkeuskäyrä D_A alaneljänneksestä pienentämällä D_A:n arvoa yhdellä ja siirtämällä käyrän pisteet vasemman neljänneksen korkeuskäyrälle D_V.

Ennen kunkin täsmäysluokan *lopullista terminoimista* on vielä selvitettävä, onko täyttyneellä täsmäysluokalla k edustaja matriisin *molemmilla alueilla*, joilla merkkijonojen selaus tapahtuu alusta loppuun päin, eli pareittain *ylä- ja vasemmassa neljänneksessä* ja vastaavaan tapaan yhtäläisesti *ala- ja oikeassa neljänneksessä*. Tarkastellaan tilannetta ensiksi edettäessä merkkijonoja alusta loppuun. Alinta vielä loppuun käsittelemätöntä täsmäysluokkaa matriisin ylä- ja vasemmassa neljänneksessä ilmaisee muuttuja T_{YV}. Kun pääsilmukan laskuri i on jonkin rivi-sarakeparin käsittelyn

jälkeen saavuttanut arvon $z = Min\{maxind(i, j) \mid (i, j) \in D_{T_{YV}}\}$, on kaikki luokan T_{YV} dominantit täsmäykset löydetty. Mikäli sillä luokan $D_{T_{YV}}$ täsmäyksellä (i, j), jolla rivi- ja sarakenumeron maksimina esiintyy z, on voimassa $i \leq j$, on täsmäys löytynyt *yläneljänneksestä* riveittäisessä tarkastelussa. Tällöin pitää tarkastaa, onko juuri täyttyneellä täsmäysluokalla T_{YV} edustajaa *myös vasemmassa neljänneksessä*. Jos tällainen löytyy, ei korkeuskäyrää T_{YV} sittemmin enää tarkastella. Mutta ellei tällaista edustajaa ole olemassa, pitää tutkia vielä lisäksi, onko *alaneljänneksen* korkeimman eityhjän täsmäysluokan D_A kynnysarvo $\leq z$. Mikäli näin on, voidaan kyseinen täsmäysluokka lakkauttaa, sillä kaikki siihen kuuluvat täsmäykset ovat vertailukelvottomia juuri edellä ylä- ja vasemmasta neljänneksestä terminoidun korkeuskäyrän T_{YV} kanssa, ja ne voidaan liittää sen loppuun matriisin vasempaan alakulmaan. Seuraava esimerkki valaisee asiaa tarkemmin.

Esimerkki 3.35: *Täsmäysluokan terminointi vasemmasta yläkulmasta alkaen edettäessä silmukkalaskurin i arvolla 3, kun matriisin vasemmassa neljänneksessä ei ole yhtään täyttyvän täsmäysluokan $T_{YV} = 1$ edustajaa, mutta alaneljänneksen ylimmän luokan täsmäyksen sarakeindeksi ≤ 3.*



Esimerkissä 3.35 GCL-algoritmi löytää dominantit 1-täsmäykset ensinnä riviltä 1 paikasta (1, 3) ja sitten riviltä 7 paikasta (7, 2) kierroslaskurin *i* arvolla 1, ja laskurin arvolla 2 löytyy vielä yksi dominantti 1-täsmäys lisää sarakkeelta 6 paikasta (4, 6). Kun seuraavaksi silmukkalaskuri *i* saavuttaa arvon 3, tutkitaan matriisin kolmas rivi ja sarake, mutta tässä tapauksessa tuloksetta. Tämän jälkeen todetaan, että laskurin arvo 3 on sama kuin ainoan vasemmasta yläkulmasta alkavin tarkasteluin löytyneen 1-täsmäyksen (1, 3) maksimi-indeksi. Täten täsmäysluokka 1 voidaan nyt terminoida tarkastelusuunnastaan, kun sekä kolmas rivi että sarake ovat jo käsitellyt. Koska vasemmassa neljänneksessä ei ole yhtään 1-täsmäystä, on vielä tutkittava, sijaitseeko alaneljänneksen ylimmän täsmäysluokan edustajaa sarakkeella, jonka numero ≤ 3. Tällainen, luokkaan 1 kuuluva täsmäys, löytyy paikasta (7, 2). Koska (1, 3) on yläneljänneksen vasemmanpuoleisin 1-täsmäys, ja vasemmassa neljänneksessä ei 1-täsmäyksiä esiinny, kelpaa (7, 2) nyt matriisin ylimmältä riviltä alkavan luokan 1 korkeuskäyrän pisteeksi. Pisteen siirron tapahtuessa pienennetään samalla alaneljänneksen korkeimman täsmäysluokan arvoa D_A ykkösestä nollaan, eli korkeuskäyrä T_1A lakkaa olemasta olemassa.

Myös edellä kuvatussa tilanteessa GCL-algoritmi toimii symmetrisesti. Jos nimittäin tilanne onkin sellainen, että terminoituvan täsmäysluokan T_{YV} kaikki edustajat löytyvät *vasemmasta neljänneksestä* – eli yläneljännes on tyhjä – pitää tutkia, millä rivillä sijaitsee ylimmän luokan D_O täsmäys *oikeassa neljänneksessä*. Tieto tästä on tallennettuna paikkaan *Sarakekynnys**[D_O]. Mikäli mainittu täsmäys sijaitsee samalla rivillä tai ylempänä kuin korkeuskäyrän T_{YV} ylin täsmäys, eli sen rivi-indeksi $\leq z$, lakkautetaan täsmäysluokka D_O ja siirretään sen pisteet korkeuskäyrän T_{YV} alkuun oikeaan yläkulmaan. Korkeuskäyrän T_{YV} käsittely on nyt lopullisesti saatu päätökseen.

Oikeasta alakulmasta alkaneiden korkeuskäyrien terminointi noudattelee samoja periaatteita kuin edellä kuvatut toimenpiteet vasemmasta yläkulmasta alkaneiden korkeuskäyrien terminoimiseksi. Kun silmukkalaskuri i on saavuttanut arvon $z^* = 1 + Min\{Max\{m - i, n - j\} \mid \forall(i, j) \in D_{T_{AO}}\}$, on kaikki korkeuskäyrän T_{AO} dominantit täsmäykset löydetty[100]. Olettaen, että kyseisen luokan täsmäyksiä löytyi ainoastaan matriisin *alaneljänneksestä*, pitää tutkia, millä sarakkeella *yläneljänneksen* ylimmän täsmäysluokan D_Y korkeuskäyrä leikkaa rivin i. Jos se tapahtuu sarakkeella, jonka numero $\geq n + 1 - z^*$, lakkautetaan täsmäysluokka D_Y yläneljänneksestä, ja sen pisteet siirretään alaneljänneksestä alkavan korkeuskäyrän T_{AO} jatkeeksi matriisin oikeaan yläkulmaan. Mikäli luokan T_{AO} kaikki edustajat löytyvät sitä vastoin *oikeasta neljänneksestä*, pitää selvittää, millä rivillä *vasemman neljänneksen* ylin korkeuskäyrä D_V leikkaa sarakkeen z^*. Jos kyseinen rivi on numeroltaan $\leq m + 1 - z^*$, lakkautetaan täsmäysluokka D_V vasemmasta neljänneksestä ja siirretään kaikki sen täsmäykset oikean neljänneksen luokkaan $D_{T_{AO}}$. GCL-algoritmin toiminnan kannalta on oleellista, että ennen tarkastelun siirtämistä toiseen kulmaan taataan, etteivät mitkään kaksi erillistä korkeuskäyrää kohtaa toisiaan tai risteä keskenään missään kohdin matriisia, eikä mitään jo kertaalleen löydettyä täsmäystä poisteta kesken algoritmin suorituksen.

Esimerkkien 3.32 – 3.35 avulla olemme havainneet, että GCL pyrkii suorituksensa aikana *yhdistämään eri alueiden korkeuskäyriä toisiinsa*. Esimerkissä 3.32 näin tapahtui, kun havaittiin tietyn täsmäysluokan *k kaikkien edustajien jo löytyneen* tarkasteltaessa syötejonoja joko vasemmasta ylä- tai oikeasta alakulmasta lähtien. Tällöin luokan *k* korkeuskäyrät ylä- ja vasemmassa neljänneksessä – tai vaihtoehtoisesti ala- ja oikeassa neljänneksessä – yhdistyivät. Esimerkeissä 3.33 ja 3.34 puolestaan lakkautettiin ylin korkeuskäyrä tarkastelualueen siitä naapurineljänneksestä, jossa sekä prosessoinnin suunta että käsiteltävä dimensio ovat keskenään päinvastaiset. Näin toimittiin, mikäli kahden vierekkäisen ositteen korkeuskäyrät olivat kehittyneet toisiinsa nähden vertailukelvottomiksi. Tällöin keskenään vertailukelvottomat, kaksi eri suunnista alkavaa korkeuskäyrää yhdistettiin yhdeksi siirtämällä lakkautetun käyrän pisteet tarkasteltavan neljänneksen ylimmälle korkeuskäyrälle. Kolmas mahdollinen edellytys korkeuskäyrien yhdistämiselle nähtiin esimerkissä 3.35. Siinä kaksi rivi- tai

[100] Luokkaan $D_{T_{AO}}$ saattaa tässä kuulua myös ei-dominantteja täsmäyksiä samaan tapaan kuin edellä luokkaan $D_{T_{YV}}$. Kts. tarkemmin alaviite 95.

sarakesuuntaista, mutta keskenään *päinvastaisista kulmista* alkaen tarkasteltua korkeuskäyrää osoittautuivat keskenään vertailukelvottomiksi tilanteessa, jossa ensimmäinen käyristä on juuri terminoitunut tarkastelualueellaan. Tällöin toinen käyristä lakkautetaan, ja pisteet siirretään ensin mainitulle käyrälle.

Tarkasteltaessa korkeuskäyrien yleisiä ominaisuuksia aliluvussa 2.3.2 todettiin, etteivät korkeuskäyrät milloinkaan sivua eivätkä leikkaa toisiaan. Edelleen, yhdelle ja samalle korkeuskäyrälle kuuluvat kaikki pisteet ovat keskenään *vertailukelvottomia*: niistä vain tarkalleen yksi hyväksytään kerrallaan mukaan syötteiden PYAan. Jotta GCL-algoritmi laskisi PYAn pituuden oikein, pitää varmistua, että *kaikki tarkastelusuunnan dominantit täsmäykset kuuluvat jollekin korkeuskäyrälle*, ja lisäksi vielä *korkeuskäyrien lukumäärä* syötejonojen muodostamassa matriisissa *M* on *minimaalinen*. Ensin mainittu vaatimus täyttyy suoraan algoritmin prosessointitavan perusteella: ensimmäiset $\lceil m/2 \rceil$ riviä kun tutkitaan täsmälleen samoin kuin Rickin I algoritmissa, ja myös alimmaisten $\lfloor m/2 \rfloor$ rivin tutkiminen noudattaa samoja periaatteita: vain merkkien tarkastelujärjestys on päinvastainen. Ongelmana on vain, miten algoritmin pääsilmukan suorituksen päättymiseen mennessä vielä terminoimattomat, eri suunnista löydetyt korkeuskäyrät, saadaan yhdistettyä toisiinsa, jotta korkeuskäyrien kokonaismäärä matriisissa saataisiin minimoitua. Kun tämä tavoite on saavutettu, saadaan selville PYAn pituus ja voidaan aloittaa ratkaisuksi kelpaavan jonon kerääminen. Jo loppuun käsitellyt korkeuskäyrät jäävät ennalleen, ja niiltä kultakin löydetään edustaja siten myös ratkaisuksi kelpaavaan jonoon.

3.4.2.4 PYAn pituuden määrääminen

Keskeneräisten korkeuskäyrien käsittelyä varten pääsilmukan suorituksen päätyttyä algoritmi huomioi neljä eri tapausta, joita tarkastellaan seuraavaksi.

Tapaus 1

Ensimmäisenä tapauksena tarkastellaan tilannetta, jossa matriisin *M vasemmasta neljänneksestä* löydetyn ylimmän korkeuskäyrän D_V numero on vähintään yhtä suuri kuin *yläneljänneksestä* löydetyn D_Y:n, ja samalla *alaneljänneksestä* löytynyt ylin korkeuskäyrä D_A on numeroltaan ainakin samansuuruinen kuin oikeasta neljänneksestä löydetty D_O. GCL-algoritmin prosessointitavan perusteella tiedetään, että kaksi samannumeroista korkeuskäyrää ylä- ja vasemmassa neljänneksessä sekä vastaavasti ala- ja oikeassa neljänneksessä ovat pareittain vertailukelvottomia keskenään. Täten vasemmasta yläkulmasta edettäessä alueen PYAn pituus on $Max\{D_Y, D_V\}$ ja oikeasta alakulmasta edettäessä vastaavasti $Max\{D_A, D_O\}$. Koska sen sijaan yksikään vasemman neljänneksen korkeuskäyristä ei ole vertailukelvoton

minkään alaneljänneksen korkeuskäyrän kanssa[101], saadaan PYAn pituus laskettua yksinkertaisesti muodostamalla summa $p = D_V + D_A$. Kuvatun kaltainen tilanne syntyy vaikkapa esimerkissä 3.32, jossa $D_V = 2 \geq D_Y = 1$ ja $D_A = 2 \geq D_O = 2$.

Tapaus 2

Toisena tapauksena GCL huomioi tilanteen, jossa yläneljänneksen ylin korkeuskäyrä D_Y on numeroltaan aidosti suurempi kuin vasemman neljänneksen D_V, ja samalla oikean neljänneksen ylimmän korkeuskäyrän numero D_O on ainakin yhtä suuri kuin alaneljänneksen vastaava arvo D_A. Edellisen tapauksen tarkastelun perusteella voidaan arvot D_V ja D_A jättää nyt huomiotta. Näin voidaan toimia, sillä ylä- ja oikean neljänneksen korkeuskäyrien vertailukelvottomuus testataan aina, kun jommaltakummalta alueelta löytyy uusi korkeuskäyrä, ja keskenään vertailukelvottomat ylimmät korkeuskäyrät tuolloin yhdistetään. Täten taataan, etteivät näihin neljänneksiin jäljelle jäävät korkeuskäyrät ole enää vertailukelvottomia keskenään, vaan kustakin niistä voidaan valita yksi edustaja PYAan. Näin ollen PYAn pituudeksi p saadaan nyt $D_Y + D_O$. Kuvatun kaltainen tilanne syntyisi, jos esimerkissä 3.34 jonon X viides merkki ”C” vaihdettaisiin ”E”:ksi. Tällöin olisi voimassa $D_Y = 1 > D_V = 0$ ja $D_O = 1 \geq D_A = 0$[102].

Tapaus 3

Kahdessa ensimmäisessä tapauksessa ei esiintynyt tilannetta, jossa molemmista suunnista edettäessä ylin korkeuskäyrä olisi löydetty *matriisin samasta dimensiosta*. Näin käy kuitenkin tapauksissa 3 ja 4. Kolmannessa tapauksessa vasemman neljänneksen ylin korkeuskäyrä on numeroltaan vähintään yhtä suuri kuin yläneljänneksen ($D_Y \leq D_V$), ja oikeasta neljänneksestä löydetään suurinumeroisempi korkeuskäyrä kuin alaneljänneksestä ($D_A < D_O$). Tällöin vasemmasta yläkulmasta edettäessä ainakin kaikki ne korkeuskäyrät, joiden numero $> D_Y$, ovat jääneet terminoitumatta pääsilmukan suorituksen aikana. Koska näitä korkeuskäyriä ei esiinny yläneljänneksessä, on kaikille niille kuuluvilla vasemman neljänneksen dominanteilla täsmäyksillä (k, l) oltava voimassa $k > \lceil m/2 \rceil$ ja $l \leq \lceil m/2 \rceil$. Tarkastellaan nyt vuorostaan tilannetta matriisin alapuoliskolla. Koska oletuksen mukaan $D_A < D_O$, ovat ainakin täsmäysluokat, joiden numero $> D_A$, jääneet alueella terminoimatta. Koska näiden edustajia esiintyy pelkästään oikeassa neljänneksessä, on kaikille kyseisille täsmäyksille (k', l') oltava voimassa $k' \leq \lceil m/2 \rceil$ ja $l' \geq \lceil m/2 \rceil$. Koska jokaiselle mahdolliselle täsmäysparille (k, l) ja (k', l') on nyt voimassa $k > k'$ ja $l \leq l'$, ovat kyseiset täsmäysparit vertailukelvottomia keskenään. Jos nyt $D_V \geq D_O$, saadaan PYAn pituudeksi $p = D_V + D_A$, sillä vasemman ositteen korkeuskäyrät ovat jo aikaisemmin todetun perusteella

[101] Vastinparialueiden A ja B korkeuskäyrien välinen vertailukelvottomuus testataan aina löydettäessä jommaltakummalta uusi korkeuskäyrä. Jos ne ovat keskenään vertailukelvottomat, ne yhdistetään yhdeksi.

[102] Kannattaa huomioida, että riviä 3 käsiteltäessä alun perin oikeasta neljänneksestä löytynyt 2-täsmäys (3, 6) siirrettäisiin yläneljänneksen 1-täsmäykseksi. Täten $D_Y = D_O = 1$.

vertailukelpoisia alaneljänneksen käyrien kanssa. Jos puolestaan $D_V < D_O$, määräytyy PYAn pituudeksi vastaavasti $p = D_Y + D_O$, sillä ylä- ja oikean neljänneksen korkeuskäyrien vertailukelpoisuus on taattu algoritmin suorituksen aikana. Edellä nähty esimerkki 3.34 edustaa sellaisenaan tapausta 3.

Tapaus 4

Mutkikkain tilanne syntyy neljännessä tapauksessa, jossa kummastakin suunnasta prosessoitaessa rividimensioilta löytyy aidosti korkeampi ylin täsmäysluokka kuin saraketarkastelussa, ts. $D_Y > D_V$ ja $D_A > D_O$. Tällöinkin jo terminoidut korkeuskäyrät säilyvät myös lopullisessa ratkaisussa, ja PYAn pituus p alustetaan arvoon $Min\{D_Y, D_V\} + Min\{D_A, D_O\}$. Sen sijaan keskeneräisiksi jääneiden käyrien tarkastelussa GCL:ssä lähdetään liikkeelle täsmäysluokasta T_{AO} (= D_O + 1), joka on oletuksen perusteella edustettuna matriisin alaneljänneksessä. Sen sarakeindeksiä v verrataan matriisin yläneljänneksen korkeimman täsmäysluokan D_Y sarakeindeksiin u. Jos $u < v$, täsmäykset ovat vertailukelpoiset, PYAn pituus kasvaa yhdellä, apumuuttujan $\tilde{u}$ arvoksi asetetaan u ja apumuuttujan $\tilde{v}$ arvoksi v, ja siirrytään matriisin alaneljänneksessä seuraavalle korkeuskäyrälle kasvattamalla muuttujan v arvoa yhdellä. Jos nyt kuitenkin $u \geq v$, luokat D_Y ja D_O + 1 ovat keskenään vertailukelvottomat, joten ne yhdistetään yhdeksi PYAn pituuden samalla kasvamatta. Samalla asetetaan $u := u - 1$ ja $v := v + 1$. Tällä tavoin jatketaan, kunnes joko 1) yläneljänneksen korkeuskäyrien laskuri u saavuttaa arvon $T_{YV} - 1$ (= D_V) tai 2) alaneljänneksen korkeuskäyrien laskurina toimiva v saavuttaa arvon D_A + 1 eli ylittää alueelta löytyneiden korkeuskäyrien kokonaismäärän. Jos molemmat lopetusehdot toteutuvat samanaikaisesti, toimitaan tilanteen 1 mukaisesti. Mikäli korkeuskäyrien vertailun yhteydessä kaksi käyrää T_uY ja T_vA olivat ainakin jollain u:n ja v:n arvoilla vertailukelpoiset – eli niiden numerot otettiin edellä talteen muuttujiin $\tilde{u}$ ja $\tilde{v}$ – käsitellään kyseistä tilannetta erikoistilanteena 3) riippumatta siitä, kumpi lopetusehdoista toteutui vertailun päättyessä.

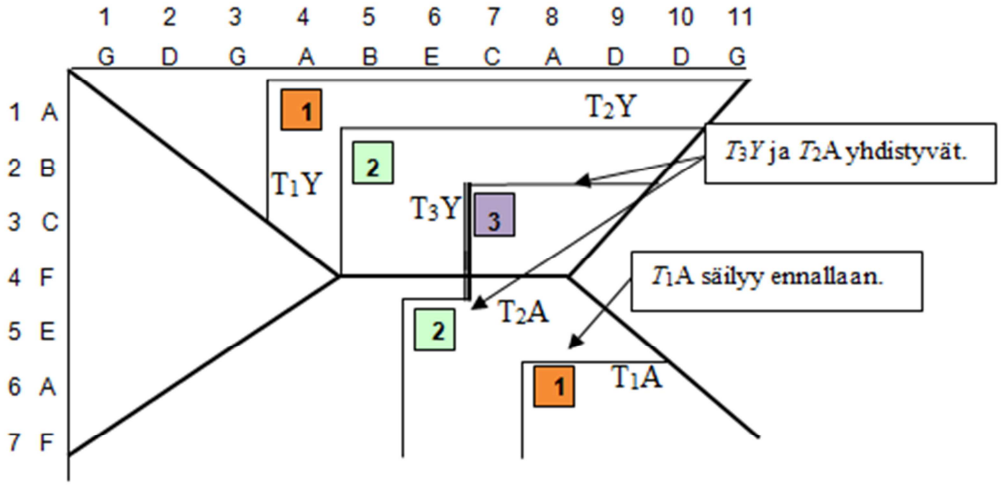
Tilanteessa 1 ylä- ja alaneljänneksen korkeuskäyrien vertailu edellä päättyi siis muuttujan u saavutettua arvon D_V siten, etteivät mitkään pareittain vertailluista korkeuskäyristä T_uY ja T_vA olleet vertailukelpoiset keskenään. Tällöin korkeuskäyrät $T_1..T_{D_V}$ olivat löytyneet matriisin yläpuoliskolta sekä rivi- että saraketarkasteluissa, ja ne on löytymisvaiheessaan siten todettu vertailukelpoisiksi alaneljänneksen korkeuskäyrien kanssa. Täten summa $u + D_A$ on tässä tilanteessa arvoltaan sama kuin $D_V + D_A$.

Jos sen sijaan vertailun päättymisen aiheutti *tilanne 2* eli korkeuskäyrien loppuminen alaneljänneksestä samalla, kun $u \geq T_{YV}$, saatiin kaikki alaneljänneksen terminoimattomat korkeuskäyrät yhdistettyä yläneljänneksen ylimpien korkeuskäyrien kanssa. Siten summa $u + D_A$ on arvoltaan tällöin sama kuin $D_Y + D_O$, sillä GCL takaa algoritmisesti ylä- ja oikean neljänneksen korkeuskäyrien vertailukelpoisuuden.

Tilanteessa 3 puolestaan yläneljänneksen korkeuskäyrä T_uY ja alaneljänneksen käyrä T_vA olivat ainakin yhden kerran vertailukelpoiset jollain vertaillulla

indeksiyhdistelmällä (u, v). Tällöin alaneljänneksen korkeuskäyrää T_vA ei voitu yhdistää mihinkään yläneljänneksen korkeuskäyristä. Olkoon ($\tilde{u}$, $\tilde{v}$) viimeinen tällainen vertailtu indeksipari. PYAn pituus saadaan lopulta näiden muuttujien summasta. Kannattaa huomioida, että summa $u + v$ kasvaa edellisestä arvostaan aina tarkalleen ykkösen verran, sillä korkeuskäyrien ollessa vertailukelpoiset vain muuttujan v arvoa kasvatetaan, kun taas u jää ennalleen. Muutoin summa $u + v$ ei muutu, sillä samalla kun v kasvaa yhdellä, u vastaavasti pienenee yhdellä. Seuraavassa esitetään vielä esimerkki siitä, miten keskeneräisten korkeuskäyrien yhdistäminen etenee tapauksessa 4.

Esimerkki 3.36: *Korkeuskäyrien yhdistäminen tapauksessa 4* ($D_Y > D_V$ ja $D_A > D_O$).



Täsmäyksiä löydetään pelkästään ylä- ja alaneljänneksistä, ja ainoastaan ylävasemmalta alkava korkeuskäyrä 1 ehtii terminoitua täsmäyksen (1, 4) ansiosta ennen pääsilmukan suorituksen päättymistä. Ylhäältä löytyy yksi edustaja kuhunkin täsmäysluokkaan 1 – 3 ja alhaalta luokkiin 1 – 2, joten $D_Y = 3$, $D_A = 2$ ja $D_V = D_O = 0$. Täten muodostuu tapaus 4, joten laskurimuuttujat u ja v alustetaan arvoihin 3 ($=D_Y$) ja 1 ($=T_{AO} = D_O + 1$). Korkeuskäyrien yhdistäminen alkaa vertaamalla T_3Y:n ja T_1A:n sarakeindeksejä. Koska $Rivikynnys[3] = 7 < Rivikynnys*[1] = 8$, T_3Y ja T_1A ovat vertailukelpoiset keskenään, joten niitä **ei voi yhdistää**. Nykyiset u ja v otetaan talteen päivittämällä muuttujat $\tilde{u}$ (arvo ei tässä muutu) ja $\tilde{v}$. Seuraavaksi v kasvaa arvoon 2 ja yritetään käyrien T_3Y ja T_2A yhdistämistä. Tämä onnistuu, koska $Rivikynnys[3] = 7 \geq Rivikynnys*[2] = 6$. Tämän jälkeen korkeuskäyrät loppuvat kesken alaneljänneksestä, joten PYAn pituudeksi saadaan $p = \tilde{u} + \tilde{v} = 3 + 1 = 4$.

Liitesivuilla kohdassa 11.4.2 on nähtävissä alkuperäisartikkelissa esitellyn pohjalta muunneltu GCL-algoritmin pseudokoodilistaus. Kannattaa huomioida, että alkuperäisartikkelissa *ei ole eksaktisti otettu kantaa siihen, miten jokin PYAn ratkaisuksi kelpaava jono saadaan muodostettua*. Tästä huolimatta siinä kuitenkin kerätään riittävät tiedot ratkaisun löytämiseksi.

3.4.2.5 PYAn esiintymän palauttaminen rekursiivisesti

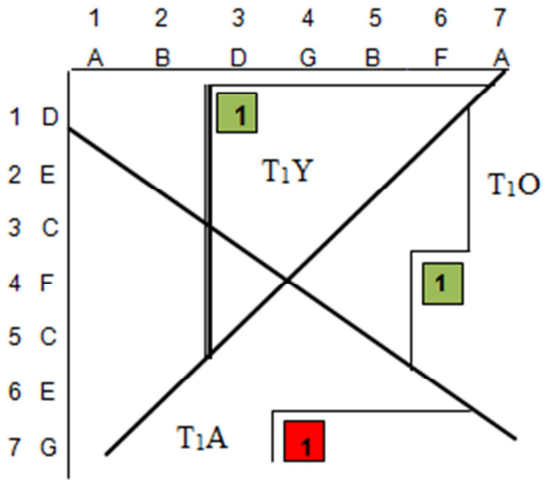
GCL-algoritmin tehokkuus lähinnä sitä muistuttavaan Rickin I algoritmiin verrattuna perustuu pitkälti niukkaan täsmäysketjujen kirjanpitoon. Siinä missä RI1:ssä jokainen löydetty dominantti täsmäys linkitettiin myös edeltäjänsä kanssa PYAn

esiintymän löytämiseksi, GCL ylläpitää riittävää tietoa PYA-jonon keräämiseksi esittelemällä ainoastaan muutaman tarpeellisen yksinkertaisen muuttujan! Näistä muuttujista neljä on muotoa (*u*, *v*) olevia *pisteparimuuttujia*, joita tarvitaan ylimmän luokan dominantin täsmäyksen kirjaamista varten kussakin matriisin eri neljänneksessä: c^Y (yläneljännes), c^O (oikea neljännes), c^A (alaneljännes) ja c^V (vasen neljännes). Sitä mukaa kun tarkasteltavasta neljänneksestä löydetään uuden ylimmän täsmäysluokan edustaja – toisin sanoen PYAn pituus jossain matriisin neljänneksistä kasvaa, kun se vastinparineljänneksessä pysyy ennallaan – päivitetään kyseisen neljänneksen pisteparimuuttujan arvoksi löytyneen täsmäyksen koordinaatit. Muuttujan aikaisempi arvo kirjoitetaan yli. Esimerkissä 3.36 muuttuja c^Y saisi rivillä 1 arvokseen (1, 4), joka päivittyisi seuraavilla riveillä arvoihin (2, 5) ja (3, 7). Edellä mainittuja muuttujia joudutaan päivittämään myös tapauksessa, jossa *kaikki seuraavat kolme ehtoa täyttyvät samanaikaisesti*: 1) jokin täsmäysluokka k ($1 \le k \le \lceil m/2 \rceil$) *terminoidaan* tarkastelukulmastaan katsottuna (ylävasemmalta tai alaoikealta), ja terminoitumisen aiheuttaa dimensiota *A* (joko riviä tai saraketta) tutkittaessa löytynyt täsmäys, 2) samasta kulmasta lähtien tarkasteltavalla viereisellä alueella *ei esiinny* luokan *k* täsmäystä sekä 3) päinvastaisesta kulmasta aloitettaessa ylimmän täsmäysluokan edustaja samalla dimensiolla *A* on *vertailukelpoinen* sen täsmäyksen kanssa, joka aiheutti luokan *k* terminoitumisen (ei siis synny esimerkin 3.35 kaltaista tilannetta matriisin alalaidassa). Jos esimerkiksi luokka *k* terminoituu yläneljänneksen täsmäyksen (*u*, *v*) ansiosta[103], ja luokalla *k* ei ole edustajaa vasemmassa neljänneksessä, tulee (*u*, *v*):stä nyt kelvollinen edeltäjä kaikille alapuoliskon täsmäyksille, kunhan sieltä löytynyt ylimmän luokan täsmäys sijaitsee sarakeindeksissä > *k*. Tarkastellaan asian selventämiseksi esimerkkiä 3.37, jossa esimerkin 3.35 syötejonoja on muutettu yhden merkin verran.

Tämän lisäksi on tarpeen pitää kirjaa siitä, *monesko neljänneksen täsmäysketjun jäsenistä viimeksi löydetty korkeimman luokan täsmäys oli*. Edellä jo ehdittiin todeta, että ylä- ja oikean neljänneksen sekä vastaavasti vasemman ja alaneljänneksen täsmäysketjut ovat aina vertailukelpoiset keskenään. Siten oikean neljänneksen ylimmän täsmäysluokan edustaja c^O voidaan linkittää yläneljänneksen ylimmän luokan edustajan c^Y perään siten, että ylä- ja oikean neljänneksen yhdistetty täsmäysketju tuottaa syötejonojen laillisen yhteisen alijonon. Ketjussa rajatäsmäyksen c^Y järjestysnumero otetaan talteen muuttujaan $vasenpos^{YO}$, ja lisäämällä tähän ykkönen saadaan vastaavasti toisen rajatäsmäyksen c^O järjestysnumero. Vasemman ja alaneljänneksen pisteitä yhdistävän täsmäysketjun rajapisteen c^V järjestysnumero otetaan talteen puolestaan muuttujaan $vasenpos^{AV}$, ja lisäämällä tähän ykkönen saadaan alueen oikeanpuoleisen rajatäsmäyksen c^A järjestysnumero.

[103] Tällöin on voimassa $k = Max\{u, v\}$.

Esimerkki 3.37: *Täsmäysluokan terminointi vasemmasta yläkulmasta alkaen edettäessä silmukkalaskurin i arvolla 3, kun matriisin vasemmassa neljänneksessä ei ole yhtään täyttyvän täsmäysluokan $T_{YV} = 1$ edustajaa, mutta alaneljänneksen ylimmän luokan täsmäyksen sarakeindeksi > 3.*



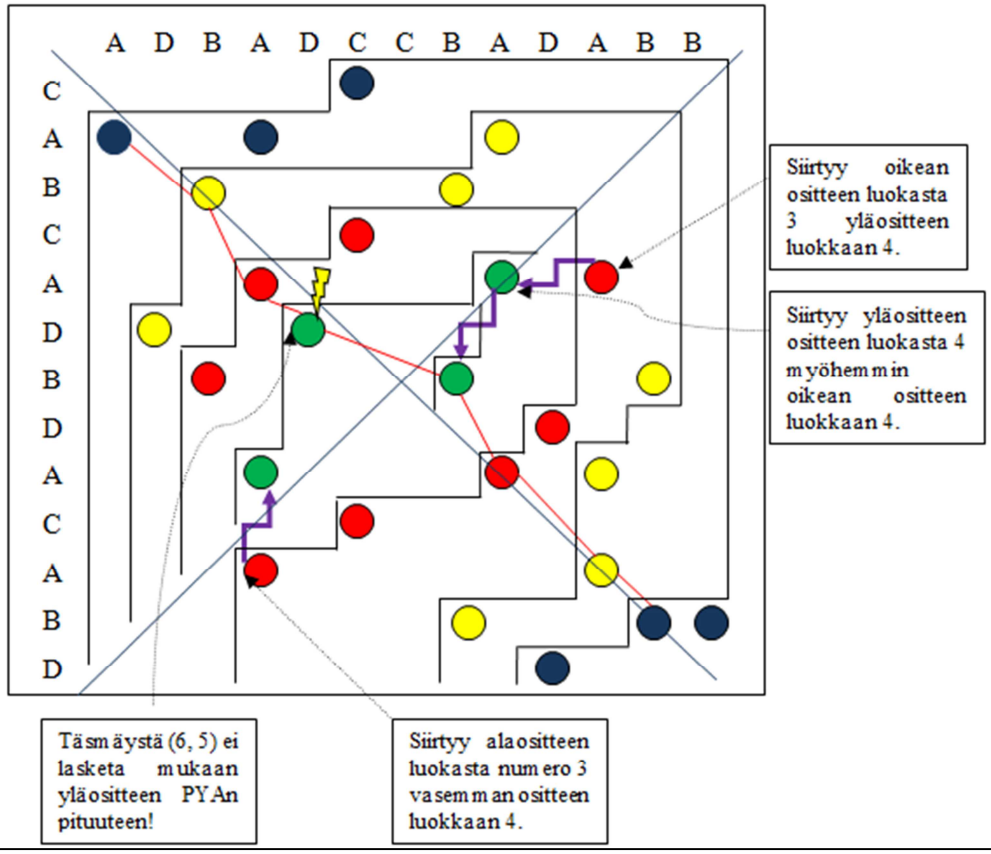
Esimerkkiin 3.35 verrattuna syötejonot ovat yhtä merkkiä lukuun ottamatta samat: x_7 samoin kuin y_4 ovat vaihtuneet ”B”:stä ja ”A:”sta ”G”:ksi. Algoritmi löytää täsmäykset järjestyksessä (1, 3), (7, 4) ja (4, 6). Kun suoritus aikanaan eteneematriisissa riville 3 asti, voidaan vasemmasta yläkulmasta lukien luokka 1 terminoida, koska täsmäyksen (1, 3) maksimikoordinaatti on 3. Täsmäysluokalla 1 ei ole edustajaa vasemmassa neljänneksessä, joten pitää vielä tutkia, onko alaneljänneksen ylimmällä täsmäysluokalla – sekin on tässä numeroltaan 1 – edustajaa sarakkeella, jonka indeksi ≤ 3, joka olisi vertailukelvoton täsmäyksen (1, 3) kanssa. Koska näin ei asia ole, täsmäystä (7, 4) ei nyt liitetä yläpuoliskolla terminoituneeseen täsmäysluokkaan 1. Sen sijaan pitää linkittää täsmäykset (1, 3) ja (7,4) peräkkäin samaan täsmäysketjuun, sillä (1, 3) kelpaa täsmäyksen (7, 4) edeltäjäksi.

Ratkaisun kokoaminen tapahtuu rekursiivisten kutsujen avulla. Artikkelissa ei esitetä mitään pseudokooditasoista eksaktia menettelyä sille, millä eri parametrien arvoilla kutsuminen tapahtuu ja miten siinä huomioidaan erikoistilanteet, kuten jommankumman rajapisteen puuttuminen sen tähden, että tarkasteltavan alueen PYAn pituus on jo puristunut korkeintaan yhden mittaiseksi. Tekniikka noudattelee kuitenkin pitkälti aikaisemmin julkaistujen lineaaristen PYA-menetelmien yhteydessä [Apo85][Hir75][Kum87] esiteltyä. Ajatuksena on, että yksi matriisin ylä- tai vasemman ositteen sekä yksi ala- tai oikean ositteen PYAan kuuluva täsmäys sijoitetaan paikalleen PYAn ratkaisujonoon, ja ne jäävät jatkotarkastelujen ulkopuolelle. Ensimmäiseen rekursiivisen kutsupariin tulevat nyt selvästikin mukaan alueet, jotka sijaitsevat matriisissa M ensin mainitusta täsmäyksestä katsottuna ylävasemmalla ja jälkimmäisestä alaoikealla.

Ratkaisujonoa määrättäessä on kiinnitettävä tarkoin huomiota erikoistilanteisiin, joissa eri ositteiden rajatäsmäyksiä edustavat muuttujat c^Y, c^V, c^A ja c^O viittaavat täsmäyksiin, jotka on jo ehditty siirtää toisen alueen korkeuskäyrälle kuuluviksi. Lisäksi on oleellista havaita, että vaikka alkuperäisillä syötejonoilla olisikin voimassa $m \leq n$, saattavat rekursiivisessa kutsussa osat vaihtua, eli katkaistusta X:stä tulee katkaistua Y:tä pidempi, jolloin osoittimet syötejonoihin ja niihin perustuviin lähiesiintymätaulukoihin

– kuten myös syötejonoista tutkittavien osien alku- ja loppupositiot ja pituudet – pitää vaihtaa keskenään. Tulosjonolle tilaa varattaessa pitää huomioida jokaisen rekursiivisen kutsun yhteydessä, tuottaako kierros tulosjonoon kaksi uutta täsmäystä paikoilleen vai ainoastaan yhden. Alkuperäisartikkelissa [Goe99] esitetyssä versiossa aiheuttaa pulmatilanteita myös se, ettei terminoinnin yhteydessä samasta kulmasta lähtien tarkasteltavan viereisen ositteen PYAa päivitetä T_{YV}:n (tai T_{AO}:n) mittaiseksi, jos alueella ei ole kyseisen luokan täsmäystä. Seuraava esimerkki valaisee asiaa:

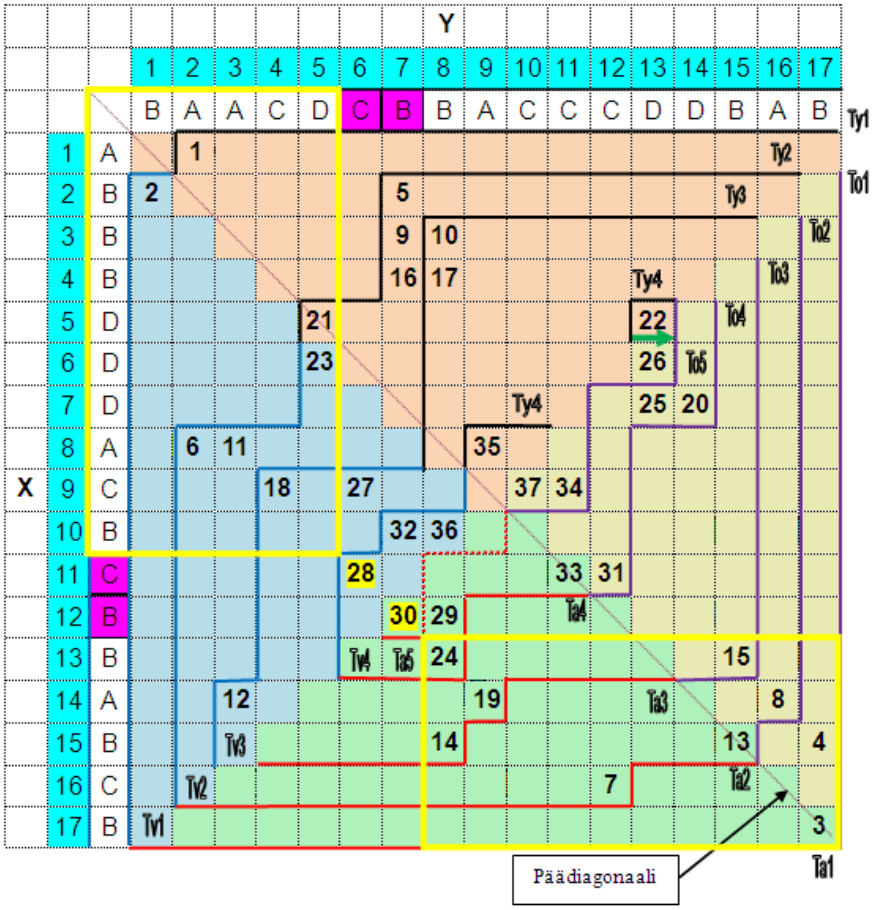
Esimerkki 3.38: *Esimerkki syötejonoista, joilla alkuperäisartikkelin mukainen PYAn pituuden laskenta ei tuota oikeaa lopputulosta:* X[1..13] = ”CABCADBDACABD”, Y[1..13] =”ADBADCCBADABB”, $\sigma = 4$, $p = 8$, PYA = ”ABADBDAB”



Jos syötejonot ovat esimerkin 3.38 mukaiset, ei artikkelissa esitetyllä pseudokoodiversiolla saada ratkaisuksi kahdeksan mittaista PYAa kuten pitäisi, vaan artikkelin versio löytäisi tapauksessa vain seitsemän merkin mittaisen PYAn. Tämä selittyy sillä, että vasempaan ositteeseen kuuluvaa luokan 4 dominanttia täsmäystä pisteessä (6, 5) ei kirjata mukaan yläositteen PYAan, kun neljäs korkeuskäyrä terminoituu sarakkeen 6 käsittelyn jälkeen. Täsmäys kuitenkin sijaitsee ylemmällä rivillä kuin oikean ositteen korkeimman luokan 4 edustaja paikassa (7, 8), eli se kelpaisi oikean ositteen täsmäysketjun edeltäjäksi. Nyt saadaan ositteiden PYAn pituuksiksi kuitenkin $D_Y = 3$, $D_V = 4$, $D_A = 3$ ja $D_O = 4$, joten artikkelin mukainen algoritmiversio määrää PYAn pituudeksi nyt summan $D_V + D_A = 4 + 3 = 7$. **Punaisella** viivalla merkittyä PYA-jonoa ei löydetä. Mielestäni algoritmia pitäisi tarkistaa korjaamalla PYAn pituus arvoon T_{YV} samasta yläkulmasta (tai T_{AO} samasta alakulmasta) alkavassa viereisessä ositteessa terminoinnin tapahtuessa. Esimerkissä täsmäyksiä kuvaavien pallojen väri osoittaa täsmäysluokkaa, johon se on asetettu ennen mahdollista siirtoa (**sininen** = 1, **keltainen** = 2, **punainen** = 3 ja **vihreä** = 4). Korjauksen jälkeen PYAn pituudeksi saataisiin $D_Y + D_O = 4 + 4 = 8$. Tämän työn pseudokoodiin mainittu muutosehdotus on otettu mukaan.

Esimerkissä 3.39 näytetään GCL-algoritmin toiminta yhdelle syötejonoparille ensimmäisen kutsun aikana, jona PYAn pituus saadaan ratkaistua. Syötejonot ovat samat kuin esimerkissä 3.1.

Esimerkki 3.39: *Täsmäysten löytymisjärjestys GCL-algoritmissa ensimmäisen kutsukerran aikana. Jokainen näistä aiheuttaa myös päivityksen kynnysarvovektoriin* (*sama arvo saatetaan kirjoittaa toistuvasti*).



Esimerkistä 3.39 nähdään, minkä syötejonojen pisteparien kohdalla GCL-algoritmi joutuu tekemään päivityksen johonkin neljästä kynnysarvovektorista. Soluihin tallennetut numerot kertovat järjestyksen, jossa täsmäykset löydetään. Eri tarkastelualueet on kuvattu toisistaan erottuvin pohjavärein – kuten myös niiltä löytyneet korkeuskäyrät. Alueiden tarkastelu etenee reunoilta matriisin keskustaa kohti. Viimeisenä etsitään rivillä 9 esiintyvää merkkiä ”C” pelkästään sarakkeesta 9. Algoritmin edetessä pitkin rivejä ja sarakkeita ehditään sekä ylävasemmalta että alaoikealta alkavat korkeuskäyrät 1 – 4 terminoida. Sen sijaan oikean ja alaneljänneksen korkeuskäyrä 5 jää vajaaksi, sillä algoritmin ensimmäinen kutsu päättyy ennen riville 9 etenemistä oikeasta alakulmasta lähtien, ja vasta silloin pisteet (9, 10) ja (12, 7) sisältävät korkeuskäyrät voitaisiin yhdistää keskenään. **Vihreällä** nuolella alleviivattu täsmäys paikassa (5, 13) kirjataan alun perin luokan 4 täsmäykseksi yläneljänneksessä, mutta se siirretään oikean ositteen viidennen korkeuskäyrän pisteeksi täsmäyksen (6, 13) löydyttyä. Koska oheisessa esimerkissä $D_Y = D_v = 4$ ja $D_A = D_O = 5$, saadaan PYAn pituudeksi 4 + 5 = 9. Haluttaessa palauttaa myös PYA-jono voidaan jako kahteen alueeseen tehdä keltaisella merkittyjen pisteparien (11, 6) ja (12, 7) toimiessa rajoina. Ensimmäinen alueista on siten X[1..10], Y[1..5] ja jälkimmäinen vastaavasti X[13..17], Y[8..17], ja ne näkyvät kuvassa keltaisilla suorakulmiolla rajattuina.

3.4.2.6 Aika- ja tilavaativuudesta

Tarkastellaan aluksi pahimman tapauksen suoritusaikakustannusta pelkän PYAn pituuden määräämiseksi ilman jonon hakemista rekursiivisten kutsujen avulla. Alustusvaiheessa GCL:ssä rakennetaan neljä lähiesiintymätaulukkoa, joiden perustaminen vaatii työtä $O(n\sigma)$. Algoritmin pääsilmukassa suoritetaan yhteensä $\lceil m/2 \rceil$ kierrosta. Yhden kierroksen aikana joudutaan etsimään korkeintaan luokkiin 1..p+1 kuuluvia täsmäyksiä, jotka löytyvät vakioajassa lähiesiintymätaulukoiden ansiosta. Siten pääsilmukan suorituskustannus on yhtäältä suuruusluokkaa $O(mp)$. Toisaalta, aivan samoin kuin Rickin I algoritmia [Ric90] analysoitaessa aliluvussa 3.4.1.3 todettiin, kutakin täsmäysluokkaa joudutaan tarkastelemaan enintään $n - p + 1$ kertaa, minkä jälkeen se terminoituu. Siten PYAn etsinnän kustannukseksi saadaan täsmälleen sama lauseke kuin RI1-algoritmille eli $O(n\sigma + Min\{mp, p(n - p)\}$.

Tarkastellaan seuraavaksi rekursiivisten kutsujen vaikutusta suoritusaikaan, kun myös ratkaisuksi kelpaava jono halutaan etsiä[104]. Oletetaan seuraavassa, että PYA jakautuu matriisin ylä- ja alakolmioihin siten, että viimeinen sen yläkolmioon kuuluva, PYAn ratkaisuun kelpaava täsmäys sijaitsee paikassa (k, l): $(1 \le k \le \lceil m/2 \rceil) \vee (1 \le l \le \lceil m/2 \rceil)$ ja ensimmäinen alakolmioon kuuluva, PYA-jonoon kelpaava täsmäys vastaavasti paikassa (k', l'): $(\lceil m/2 \rceil + 1 \le k' \le m) \vee (1 \le k \le \lceil m/2 \rceil)$. PYAan kuuluvien pisteiden ominaisuuksien perusteella on selvästikin oltava voimassa ($k < k'$) ja ($l < l'$). Tällöin rekursiiviset kutsut kohdistetaan alueille (X[1..k–1], Y[1..l–1]) ja (X[k'+1..m], Y[l'+1..n]). Kummankin alueen lyhyemmän dimension pituus on korkeintaan $\lceil m/2 \rceil$ pisteiden (k, l) ja (k', l') valinnan perusteella. Olkoon PYAn pituus ensin mainitulla alueella p_1 ja jälkimmäisellä alueella p_2, $p = p_1 + p_2 + 2$. Tällöin kahden ensimmäisen rekursiivisen kutsun yhteenlaskettu kustannus on lausekkeessa esiintyvä termi mp huomioiden

$$d(p_1 + 1)(\lceil m/2 \rceil - 1) + d(p_2 + 2)(\lceil m/2 \rceil - 1) \le dp(m/2),$$

missä d on positiivinen kerroin.

Rekursiotasoja tarvitaan enintään i kappaletta, jolloin oikean puolen summaksi saadaan alkuperäinen, ylimmän tason kutsu mukaan lukien $dpm + dp(m/2) + dp(m/4) + dp(m/8) + \ldots + dp(m/2^i)$, josta saadaan geometrisen sarjan summaksi $2dpm$.

Tarkastellaan seuraavaksi termiä $p(n - p)$. Oletetaan ensiksi, että $p \le gm$, missä $g = \frac{1}{2}(\sqrt{5} - 1) \approx 0.618$. Tällöin

$$\begin{aligned} 2dpm &\le (2d(1 - g)mp)/(1 - g)) = (2d(m - gm)p)/(1 - g) \\ &\le (2d(m - p)p)/(1 - g). \end{aligned}$$

[104] Vastaava analyysi löytyy alkuperäisartikkelin [Goe99] luvusta 5.

Nyt oletetaan, että $p > gm$. Olkoon $h = Max\{k - 1, l - 1\}$ ja $h' = Max\{m - k', n - k'\}$. Selvästikin $h + h' \leq n - 2$. Lisäksi $p_1, p_2 \leq \lceil m/2 \rceil - 1$, koska p_1 kertyy matriisin ylä- ja p_2 alakolmion täsmäyksistä. Tällöin kahden ensimmäisen rekursiivisen kutsun kustannus on enintään

$$
\begin{aligned}
& d(\lceil m/2 \rceil - 1 + p_1(h - p_1)) + d(\lceil m/2 \rceil - 1 + p_2(h' - p_2)) \\
& \leq d(m + p_1(h - p_1)) + p_2(h' - p_2)) \\
& \leq d(m + (\lceil m/2 \rceil - 1)(h - p_1 + h' - p_2)) \\
& \leq d(m + (\lceil m/2 \rceil - 1)\ (n - p)) \\
& \leq d(m + (p(n - p)/2g))).
\end{aligned}
$$

Vastaavasti, *i*. rekursiotasolla tehdään enintään $d(m + p(n - p)/(2g)^i)$ operaatiota. Tästä saadaan operaatioiden kokonaismääräksi

$$d(m \log m + p(n - p)/(1 - (1/(2g))) = d(m \log m + 2p(n - p)/(1 - g)).$$

Kumpi tahansa termeistä mp tai $p(n - p)$ tuleekin valituksi, on GCL-algoritmin aikavaativuus luokkaa $O\{n\sigma + Min\{mp, m \log m + p(n - p)\}$. Huonoimmillaan edellä esiintynyt vakiokerroin $2/(1 - g)$ on enintään 5.25:n suuruinen.

Algoritmi vaatii tilaa paitsi syötevektoreille *X* ja *Y*, niin myös lähiesiintymätaulukoille, kynnysarvovektoreille ja rekursiopinolle, jos PYAn esiintymä halutaan palauttaa. Syötevektorit mahtuvan tilaan $O(n)$, ja rekursiopinon koko on ainoastaan $O(\log m)$. Tilavaativuudessa enin merkitys kohdistuu GCL-algoritmissa siten $O(n\sigma)$-kokoisten lähiesiintymätaulukoiden muodostamiseen. Mitä pienempi syöttöaakkosto, sitä niukemmin tilaa tarvitaan, mutta ison syöttöaakkoston uskoisi vievän mennessään muilta osin tosiaankin lineaaritilaisen menetelmän tilansäästöt verrattain nopeasti.

4 PYA-ongelman yhteys kahden merkkijonon välisen lyhimmän editointietäisyyden ongelmaan

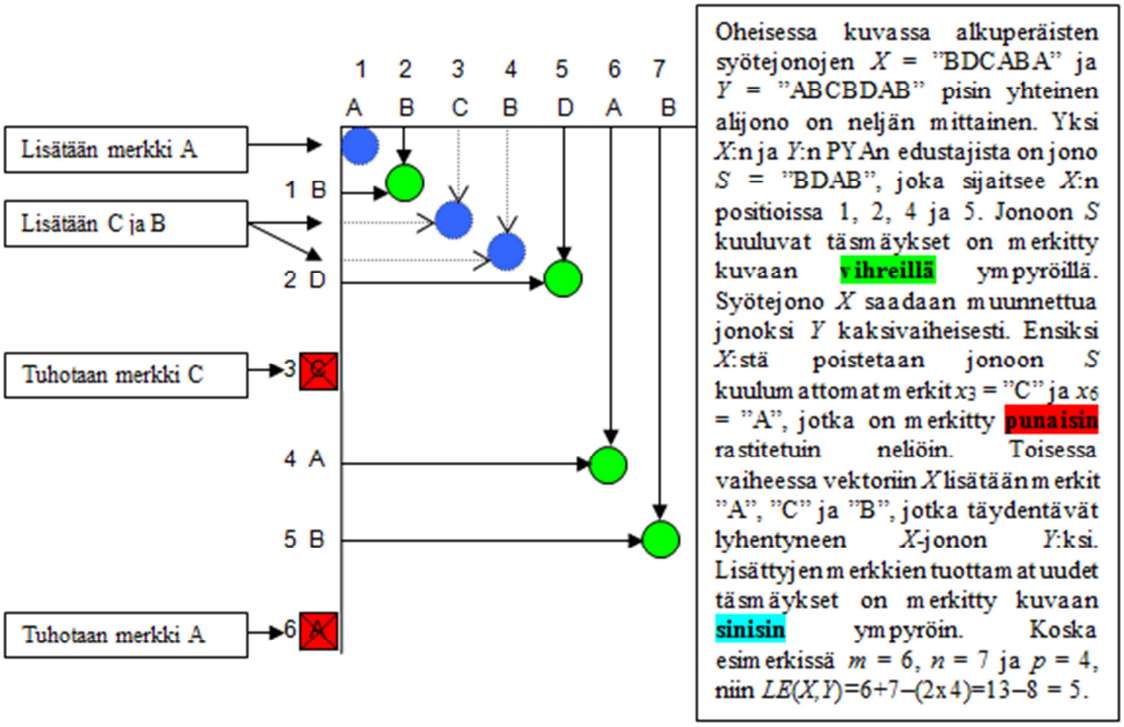
Luvussa 1 sivuttiin lyhyesti kahden merkkijonon välisen *lyhimmän editointietäisyyden* ongelmaa. Ongelman ratkaisemiseksi pyritään löytämään kustannukseltaan *halvin operaatiojono, joka muuntaa syötejonon X sisällön yhdenmukaiseksi syötejonon Y kanssa*. Tähän tarkoitukseen käytettävissä olevat operaatiot ovat *uuden merkin lisääminen* vektorin *X* mielivaltaiseen paikkaan, *olemassa olevan merkin poistaminen* vektorista *X* ja jonkin *X*:n merkin *vaihtaminen toiseksi*. Jotta ongelma olisi hyvin määritelty, pitää edellä mainituille operaatioille määrätä myös *kustannukset*. Tarkastelun pitämiseksi yksinkertaisena voidaan sopia, että yhden merkin *lisääminen tai poistaminen* vaatii työtä *yhden yksikön* verran[105]. Merkin *vaihtaminen toiseksi* vaatii puolestaan *kaksi yksikköä* työtä, sillä kyseisen operaation voi olettaa koostuvan ensinnä yhden merkin poistamisesta ja sen jälkeen uuden merkin lisäämisestä samaan paikkaan (tai päinvastoin – lisäämällä merkki vaihdettavan viereen jommallekummalle puolella ja poistamalla vaihdettava merkki tämän jälkeen). Usein myös kirjallisuudessa merkin vaihtaminen palautuu yhteen poisto-lisäys -pariin. Tästä syystä vaihto-operaatio jatkotarkasteluissa tulkitaankin kahdeksi erilliseksi operaatioksi, joista ensimmäinen on poisto ja jälkimmäinen on lisäys.

Intuitiivisesti ajateltuna on ilmeistä, että kummankin syötevektorin pisimpään yhteiseen alijonoon kuuluvia merkkejä ei tarvitse poistaa eikä vaihtaa miksikään muunnettaessa jonoa *X* jonoksi *Y*. Siten jonosta *X* ei jouduta *poistamaan* useampia merkkejä kuin ne, jotka eivät kuulu jonojen PYAan. Näitä merkkejä on *X*:ssä $m - p$ kappaletta. Vastaavasti *X*:ään joudutaan *lisäämään* ne *Y*:n merkit, jotka jäivät vektorista PYAn ulkopuolelle. Kyseisten merkkien lukumäärä on $n–p$. Siten jonon *X* muuntaminen jonoksi *Y* vaatii $m–p+n–p = m+n–2p$ operaatiota. Operaatioiden kokonaismäärä kuvaa syötejonojen *X* ja *Y* välistä *lyhintä editointietäisyyttä* (lyh. *LE*). Toisin sanoen, $LE(X, Y) = m+n–2p$. Esimerkki 4.1 toimii pienenä hahmotelmana sille, miten kahden syötejonon lyhin editointietäisyys pystytään määräämään.

Mikäli käytettävissä on algoritmi, joka ratkaisee kahden jonon *LE*:n, voidaan sitä käyttää myös jonojen PYAn pituuden ratkaisemiseksi, sillä edellä esitetystä yhtälöstä saadaan PYAn pituudeksi ratkaistua $p = ½(m+n–LE(X, Y))$. Samasta syystä PYAn pituuden ratkaisevaa algoritmia voitaisiin käyttää jonojen lyhimmän editointietäisyyden määräämiseen. Myös itse *PYAn instanssin* palauttaminen *LE*:n ratkaisevaa algoritmia käyttäen on mahdollista lisäämällä algoritmiin tarpeellinen kirjanpito tätä varten.

[105] Merkin lisäyksen ja poiston todelliset kustannukset tietokoneilla suoritettuina eivät ole välttämättä samat, mutta ellei näin oleteta, algoritmin analyysistä tulee monimutkaisempaa ja samalla suorittimesta riippuvaa. Siten erilaisten painojen käyttäminen lisäyksen ja poiston kustannuksille eivät antaisi objektiivisesti oleellisesti tarkempaa kokonaiskuvaa.

Esimerkki 4.1: *Kahden jonon PYAn ja lyhimmän editointietäisyyden yhteys toisiinsa.*



Kannattaa huomioida, että viimeksi esitettyä kaavaa käyttämällä p:n arvoksi saadaan aina kokonaisluku yhtälön oikean puolen kertoimena esiintyvästä puolikkaasta huolimatta. Tämä voidaan seuraavassa vielä todistaa lyhyesti. Oletetaan ensiksi, että m ja n ovat *molemmat parillisia tai parittomia,* jolloin summasta $m+n$ tulee parillinen. Tällöin vektorin X muuntaminen vektorin Y kanssa samanlaiseksi vaatii, että vektorin X uusi pituus $n = m + 2k$ ($k \in N$). Ellei näin olisi, jonojen X ja Y pituuksien pariteetit eivät enää olisikaan keskenään samat jonon X muuntamisoperaatioiden päätyttyä, mikä on ristiriidassa oletuksen kanssa. Siispä X:n pituus voi kasvaa vain parillisen merkkimäärän $2k$ verran. Jokaista tämän määrän ylittävää X:ään tehtävää merkin lisäysoperaatiota kohti joudutaan X:stä vastaavasti poistamaan jostain toisesta paikasta yksi merkki, ettei X:n pituus enää muuttuisi. Siten jokaista $2k$:n ylittävää merkin lisäystä kohti joudutaan tekemään myös yksi merkin poisto, joten $LE(X, Y)$ kasvaa kyseisen operaatioparin ansiosta kahdella. Samasta syystä jokainen X:ään kohdistuva poisto-operaatio aiheuttaa yhden lisäysoperaation vektoriin X: jälleen LE:n arvo kasvaa kahdella. Siten LE on aina parillinen, kun m ja n ovat pariteetiltaan samat, joten yhtälössä p:n arvoksi saadaan kokonaisluku. Jos puolestaan tarkalleen jompikumpi syötejonojen pituuksista m tai n on pariton, tulee myös summasta $m+n$ pariton. Tällöin oletuksen perusteella on oltava voimassa $m < n$ ja tiedetään, että X:ään joudutaan lisäämään pariton määrä eli $2k+1$ kappaletta merkkejä, jotta sen pituudeksi saataisiin n. Jotteivät nytkään mahdolliset ylimääräiset lisäys- ja poisto-operaatiot muuttaisi X:n pituutta kokonaisvaikutukseltaan mihinkään $m+2k+1$:stä, pitää poistoja ja lisäyksiä olla yhtä monta. Silloin LE:n arvoksi

saadaan pariton, joten yhtälön koko oikean puolen arvoksi tulee parillinen, minkä johdosta *p*:n arvoksi tulee nytkin kokonaisluku.

4.1 Lyhimmän editointietäisyyden laskevat algoritmit

Kahden merkkijonon välisen lyhimmän editointietäisyyden määräämiseksi on kehitetty muutamia algoritmeja, joista kolme esitetään seuraavissa aliluvuissa. Asymptoottisesti tehokasta[106], mutta vain hyvin pitkille syötteille soveltuvaa *Masekin ja Patersonin algoritmia* [Mas80] ei työssä käsitellä. Koska myöhemmin esiteltävissä ajoaikojen vertailuissa on haluttu kilpailuttaa juuri PYAn ratkaisevia algoritmeja keskenään, on *LE*:n ratkaisevia algoritmeja modifioitu siten, että ne palauttavat PYAn ja sen pituuden. Vastaavasti algoritmeissa on luovuttu komentojonojen *lisäysoperaatioiden* kirjaamisesta, sillä niiden ylläpito ei ole tarpeen PYAa ratkaistaessa. Sen sijaan tietoja *X*:stä *poistetuista* merkeistä on ylläpidettävä.

Yhteenvetotaulukko kaikista luvussa 4 esiteltävistä algoritmeista löytyy työn tulososasta aliluvusta 9.2. Kyseisessä taulukossa alun perin kahden merkkijonon välisen lyhimmän editointietäisyyden määräämiseen kehitetyt algoritmit on merkitty violetilla värikoodilla.

4.1.1 Millerin ja Myersin algoritmi (MMY)

4.1.1.1 Yleistä

Ensimmäisen tässä työssä esiteltävän, erityisesti kahden merkkijonon välisen lyhimmän editointietäisyyden ratkaisemiseksi kehitetyn algoritmin, esittivät *Miller* ja *Myers* [Mil85] vuonna 1985. Tekijät käyttävät algoritmistaan nimitystä *fcomp*, ja tässä työssä siitä käytetään lyhennettä *MMY*. Algoritmissa on jonkin verran piirteitä *Wagnerin* ja *Fischerin* WFI-algoritmista, sillä sen laskentamallin *tausta-abstraktiona* toimii WFI:n syötevektoreistaan muodostama, $(m+1)$ x $(n+1)$ -kokoinen matriisi *M*. Matriisin soluihin (i, j), $(0 \leq i \leq m,\ 0 \leq j \leq n)$ talletettavat arvot eivät MMY:ssä kuitenkaan edusta syötejonojen alkuliitteiden $X[1..i]$ ja $Y[1..j]$ välisen PYAn pituutta vaan niiden välistä *lyhintä editointietäisyyttä*. Lisäksi on huomionarvoista, että MMY:ssä joudutaan kaikissa matriisin soluissa vierailemaan ainoastaan *pahimmassa tapauksessa*, eli silloin, kun syötejonoilla ei ole yhtään yhteistä merkkiä. Algoritmin laskentavaihe päättyy, kun matriisin oikeassa alanurkassa sijaitsevaan soluun (m, n) tallennetaan arvo, joka on samalla $LE(X, Y)$. Tästä arvosta voidaan edelleen johtaa PYAn pituus *p* sijoittamalla

[106] Menetelmän aikakompleksisuus on $O(n^2 / \log_\sigma n)$.

syötejonojen pituuksien ja *LE*:n arvot tämän luvun alussa oikeaksi todistettuun yhtälöön.

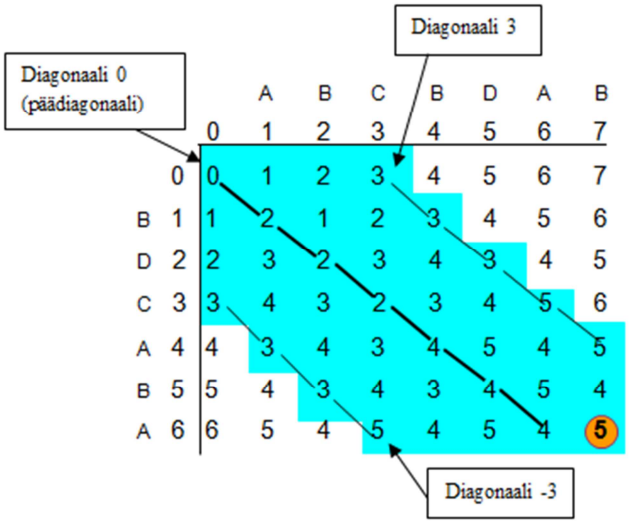
Matriisin *diagonaalit* on numeroitu siten, että matriisin *nollas diagonaali* eli *päädiagonaali* alkaa solun (0, 0) kohdalta, ja sen varrella sijaitsevien solujen indeksit ovat muotoa (*i*, *i*). Jos syötevektorit ovat yhtä pitkät, nollas diagonaali päättyy matriisin oikeaan alakulmaan. *Päädiagonaalin yläpuolella* sijaitsevien, solusta (0, *j*) alkavien lävistäjien järjestysnumero on *j*, ja niiden varrella sijaitsevat solut on indeksoitu (*i*, *i*+*j*). Sen sijaan *päädiagonaalin alapuoleisten*, solun (*i*, 0) kohdalta alkavien diagonaalien järjestysnumero on –*i*, ja niille kuuluvien solujen indeksit ovat muotoa (*i* + *j*, *j*).

4.1.1.2 Matriisiabstraktio ja laskentasäännöt

Algoritmi ei suorita jonoille minkäänlaista esiprosessointia. Sen laskennan lähestymistapa on ahnas, ja algoritmi tutkii ensi töikseen, miten pitkä *yhteinen alkuliite* syötevektoreilla *X* ja *Y* on. Tämä tapahtuu etsimällä täsmäyksiä merkkipareille (x_i, y_i) kasvaville *i*:n arvoille ykkösestä alkaen, kunnes joko käy ilmi, että koko *X* esiintyy *Y*:n alkuliitteenä, eli merkkiparit täsmäävät aina indeksiin (*m*, *m*) asti, tai sitten jollakin arvolla *z* ($0 \leq z < m$) osoittautuu, että $x_{z+1} \neq y_{z+1}$[107]. Muuttujan *z* arvo otetaan talteen nollanteen indeksipaikkaan vektorissa *diagpos*. Kyseisen vektorin pituus on *m*+*n*+1, ja sen indeksi kuvaa, *monettako matriisin M lävistäjää* ollaan käsittelemässä, ja soluun tallennettava arvo ilmaisee, *kuinka pitkälle* onnistuttiin etenemään tarkasteltavaa diagonaalia pitkin ennen kuin törmättiin ensimmäiseen ei-täsmäävään merkkipariin. Jos heti alkuun $x_1 \neq y_1$, tallentuisi *diagpos*[0]:n arvoksi 0. Jos taas päädiagonaalia pitkin päästiin etenemään suoraan alimmalle riville asti, *diagpos*[0] saa tällöin arvon *m*, mikä on merkki siitä, että diagonaali on jo ehditty käsitellä loppuun asti. Todellisuudessa kaikki se informaatio, jota MMY-algoritmi tarvitsee abstraktionaan olevasta matriisista *M*, tallennetaan laskennan aikana vektoriin *diagpos*. Seuraavassa esimerkissä kuitenkin esitetään tarkasteltavan asian selventämiseksi, miltä matriisin sisältö näyttäisi, jos sen kaikki solut täytettäisiin kyseessä olevan ongelman syötemerkkijonojen osalta.

[107] Jos koko vektori *X* esiintyy vektorin *Y* alkuliitteenä, muuttuja *z* kasvaa arvoon *m*.

Esimerkki 4.2: *Matriisiabstraktio Millerin ja Myersin algoritmissa: kuhunkin soluun* (i, j), $(0 \leq i \leq m, 0 \leq j \leq n)$, *tallennetut arvot kuvaavat alkuliitteiden* $X[1..i]$ *ja* $Y[1..j]$ *välistä lyhintä editointietäisyyttä.*



Kuvan esimerkissä on nähtävissä matriisiin *M* tallentuvat arvot syötteille ”BDCABA” ja”ABCBDAB”. Turkoosilla maalattu alue esittää diagonaalien –3..3 väliin jäävää aluetta, jonka ulkopuolella ei voi esiintyä nelosta pienempiä arvoja. Kahden päällekkäisen tai vierekkäisen solun sisältämät arvot poikkeavat toisistaan ykkösen verran. Kuljettaessa diagonaaleja pitkin alaspäin niiden solujen arvot pysyvät joko ennallaan tai kasvavat kahdella. Matriisin oikean alakulman **oranssilla** pallolla ympyröity arvo 5 esittää koko syötteiden lyhintä editointietäisyyttä.

Useimmiten ei kuitenkaan koko X esiinny Y:n alkuliitteenä, joten oletetaan seuraavaksi, että diagonaalin 0 tutkiminen pysähtyy muuttujan z arvon ollessa $< m$. Tämä tarkoittaa käytännössä sitä, että alkuliitteiden $X[1..z]$ ja $Y[1..z]$ välinen lyhin editointietäisyys on nolla, eli nuo alkuliitteet ovat identtiset. Tällöin matriisin jokaiseen päädiagonaalin soluun (i, i), missä $i \leq z$, voitaisiin sijoittaa arvo 0. Koska tiedetään, etteivät merkit x_{z+1} ja y_{z+1} täsmänneet, on alkuliitteiden $X[1..z+1]$ ja $Y[1..z+1]$ *LE*:n oltava 2, sillä merkki $X[z+1]$ joudutaan nyt vaihtamaan merkiksi $Y[z+1]$. Yleisestikin on samasta syystä voimassa, että millä tahansa diagonaalilla olevat arvot *joko pysyvät ennallaan tai kasvavat kahdella*, kun sitä pitkin edetään solusta (i, j) soluun $(i+1, j+1)$, missä $(0 \leq i < m, 0 \leq j < n)$. Kun paikassa $(i+1, j+1)$ olevat merkit täsmäävät, tällöin $M[i+1, j+1] = M[i, j]$, ja muulloin $M[i+1, j+1] = M[i, j] + 2$.

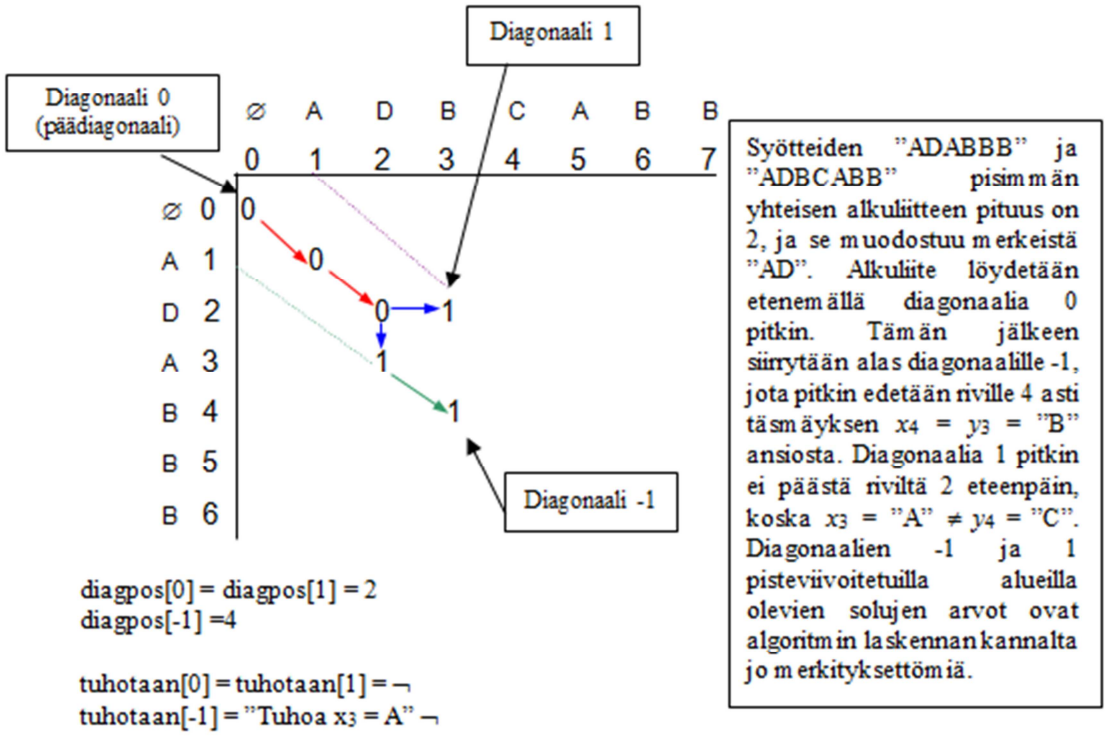
Kun on käynyt ilmi, että diagonaalia 0 pitkin ei pystytä etenemään riviä z pidemmälle $(z < m)$, siirrytään tutkimaan vuoron perään *päädiagonaalista ykkösen etäisyydellä* sijaitsevia lävistäjiä -1 ja 1 mainitussa järjestyksessä. Diagonaalille -1 siirrytään *suoraan alaspäin* diagonaalin 0 viimeisen täsmäyksen muodostaneesta indeksiparista (z, z)[108], josta saavutaan soluun $(z+1, z)$. Tämä tarkoittaa sitä, että X:n alkuliitettä pidennetään yhdellä Y:n alkuliitteen pysyessä ennallaan. Merkki $X[z+1]$ on ylimääräinen, eli alkuliitteiden lyhin editointietäisyys kasvaa nollasta yhteen. Solusta $(z+1, z)$ yritetään edetä nyt diagonaalia -1 pitkin, kunnes kohdataan jälleen ensimmäinen ei-täsmäävä merkkipari (x_{z+1+u}, y_{z+u}) $(u \geq 1)$, tai sitten diagonaalin kaikkia soluindeksejä vastaavat merkkiparit täsmäävät aina viimeisen rivin (x_m, y_{m-1}) mukaan lukien. Viimeinen rivi $z+u$, jolle asti diagonaalia -1 pitkin päästiin etenemään, tallennetaan nyt paikkaan *diagpos*[-1]. Matriisin M kaikkiin diagonaalilla -1 sijaitseviin soluihin alkaen riviltä $z+1$ aina riville $z+u$ asti voitaisiin nyt tallentaa ykkönen, sillä kyseisiä soluja vastaavien syötejonojen alkuliitteiden välinen lyhin

[108] Rivinumero z löydetään tallennettuna paikasta *diagpos*[0].

editointietäisyys on nyt 1: ainoastaan merkki x_{z+1} jouduttaisiin tuhoamaan. Edelleen tiedetään, että matriisin soluun $M[z+1+u, z+u]$[109] tallennettaisiin myöhemmin arvo 3, sillä (x_{z+1+u}, y_{z+u}) ei ollut täsmäävä merkkipari, joten x_{z+1+u}:n tilalle jouduttaisiin vaihtamaan y_{z+u}, mikä kasvattaisi editointietäisyyden kahdella yksiköllä yhdestä kolmeen.

Seuraavaksi tutkitaan diagonaali 1, jolle päästään diagonaalilta 0 etenemällä *suoraan oikealle* solusta (z, z) soluun (z, z+1). Siirtyminen matriisin rivillä yhden sarakkeen verran oikealle tarkoittaa Y:n alkuliitteen pidentämistä yhdellä X:n alkuliitteen pituuden pysyessä ennallaan. Tällöin alkuliitteiden välinen editointietäisyys kasvaa nollasta yhteen, koska nyt pitäisi X:ään *lisätä* y_{z+1}:n kanssa täsmäävä merkki. Nyt vuorostaan diagonaalia 1 pitkin edetään solusta (z, z+1) alkaen alaviistoon niin kauan, kunnes kohdataan ensimmäinen ei-täsmäävä merkkipari ($z+v$, $z+1+v$) ($v \geq 1$), tai vaihtoehtoisesti jompikumpi syötejonoista loppuu kesken. Näin käy, kun diagonaalin kaikkia soluindeksejä vastaavat merkkiparit täsmäävät joko viimeisen rivin täsmäys (x_m, y_{m+1}) tai oikeanpuoleisimman sarakkeen täsmäys (x_{n-1}, y_n) mukaan lukien[110]. Viimeisen täsmäyksen rivinumero talletetaan paikkaan *diagpos*[1]. Seuraava esimerkki havainnollistaa, miten diagonaalien 0, -1 ja 1 tutkiminen etenee.

Esimerkki 4.3: *Laskennan eteneminen Millerin ja Myersin algoritmissa.*



[109] olettaen tietystikin, ettei solu sijaitse jo matriisin alareunan ulkopuolella

[110] Kannattaa huomioida, että edettäessä päädiagonaalin alapuolella olevia lävistäjiä pitkin ei voida koskaan törmätä matriisin oikeaan reunaan, sillä oletuksen mukaan $m \leq n$. Sen sijaan, kun $m < n$, voidaan päädiagonaalin yläpuolisia lävistäjiä [1..n–m–1] pitkin edettäessä törmätä pelkästään matriisin alareunaan, kun taas lävistäjiä [n–m..n] pitkin voidaan saavuttaa matriisin oikea reuna.

Ellei algoritmi edennyt diagonaaleja 0, -1 ja 1 käsitellessään vieläkään oikean alakulman soluun (*m*, *n*) asti, lähdetään seuraavaksi tutkimaan kaikkia niitä diagonaaleja, joissa voi *LE*:n arvona esiintyä 2. Editointietäisyyteen 2 päästään ainoastaan joko kasvattamalla vektorin *X* tai *Y* etuliitteen pituutta kahdella tai vaihtamalla jokin vektorin *X* merkeistä toiseksi. Arvoa 2 voi siten esiintyä ainoastaan diagonaaleilla -2, 0 ja 2, ja ne tutkitaan tässä järjestyksessä. Diagonaaleja pitkin edetään samaan tapaan kuin aiemminkin, eli jatketaan niin pitkälle kuin perättäisiä täsmäyksiä niillä riittää, tai törmätään matriisin ala- tai oikeaan reunaan. Ensimmäiseksi käsiteltävälle diagonaalille -2 siirryttäisiin suoraan alaspäin diagonaalilta -1 riviltä *diagpos*[-1] eli samaan tapaan kuin aikaisemmin siirryttiin alaspäin diagonaalilta 0 diagonaalille -1. Vastaavasti kolmikosta -2, 0, 2 viimeisenä käsiteltävälle diagonaalille 2 päästäisiin siirtymällä oikealle diagonaalilta 1 riviä *diagpos*[1] pitkin aivan samoin kuin aikaisemmin siirryttiin diagonaalilta 0 diagonaalille 1. Sen sijaan keskimmäiselle diagonaalille 0 olisi nyt tarjolla kaksi vaihtoehtoista siirtymistapaa: joko *oikealle diagonaalilta* -1 riviä *diagpos*[-1] pitkin tai *alaspäin diagonaalilta* 1 riviltä *diagpos*[1]. Siirtymistavoista valitaan se, kumpi vie kauemmas diagonaalilla 0. Esimerkissä 4.3 voitaisiin diagonaalille 0 siirtyä seuraavaksi etenemällä joko oikealle diagonaalilta -1 soluun (4, 4) tai alas diagonaalilta 1 soluun (3, 3). Koska ensin mainittu siirtymä veisi esimerkissä pidemmälle diagonaalilla 0 kuin jälkimmäinen, siirrytään oikealle diagonaalilta -1. Yleisesti, kun ollaan siirtymässä tarkastelemaan diagonaalia *d*, ja sekä diagonaalia *d*–1 että *d*+1 pitkin on jo ehditty kulkea, siirtyminen tapahtuu siltä lävistäjältä, jolta päästään etenemään kauemmas – toisin sanoen suuremmalle rivinumerolle – diagonaalilla *d*.

4.1.1.3 Tekninen toteutus

MMY-algoritmin lyhimpien editointietäisyysarvojen 0, 1, 2 jne. määrääminen tapahtuu kahdessa sisäkkäisessä silmukassa, joista ulompi käy läpi kasvavia *etäisyysarvoja* ja sisempi – samaten järjestyksessä pienimmästä suurimpaan – kaikki *diagonaalit*, joilla voi esiintyä ulomman silmukan laskurin mukaisia etäisyysarvoja. Kun ulomman silmukan laskurina on $k \geq 0$, sisemmässä silmukassa tutkitaan järjestyksessä diagonaalit $-k$, $-k+2$, $-k+4$, …, $k-2$, k, ellei niistä uloimpia ole jo ehditty rajata tarpeettomina tarkastelujen ulkopuolelle[111]. Koska kutakin etäisyysarvoa *D* kohti tutkittavien diagonaalien numeroilla on *sama pariteetti kuin itse etäisyysarvolla*, taataan, ettei etäisyysarvojen määräämiseksi välttämättä tarvittavia tietoja menetetä liian aikaisin. Tämä perustuu huomioon, että parillisella *D*:n arvolla tarvitaan siirtymäsääntöjen mukaan avuksi vain parittomia *diagpos*-vektorin indeksejä, joiden tietosisältö ei kierroksen aikana muutu. Vastaavasti *D*:n ollessa pariton, vain parillisia *diagpos*-vektorin indeksejä joudutaan käyttämään apuna laskennassa.

[111] Turhien diagonaalien pois rajaamisesta kerrotaan tarkemmin seuraavassa kappaleessa.

Kun jotain diagonaalia *d* pitkin on jo ehditty edetä taustalla olevan matriisin *alareunaan asti sarakkeen n vasemmalla puolella*, on tarpeetonta tutkia jatkossa sen alapuolella sijaitsevia diagonaaleja *d*–1, *d*–2 jne., sillä ne eivät sijaintinsa tähden enää voi vaikuttaa arvon *M*[*m*, *n*] määräytymiseen. Tällöin asetetaan *alimmaksi relevantiksi diagonaaliksi d*+1. Vastaavasti, jos jotain diagonaalia *f* pitkin edettäessä törmätään matriisin *oikeaan reunaan rivin m yläpuolella*, tulee analogisesti diagonaalien *f*+1, *f*+2 jne. tutkimisesta turhaa. Tuolloin asetetaan *ylimmäksi relevantiksi diagonaaliksi f*–1.

Joka kerta, kun algoritmissa siirrytään alaspäin diagonaalilta *d* diagonaalille *d*–1, lisätään rivinumero *diagpos*[*d*]+1 ensimmäiseksi linkitettyyn listaan, joka alkaa vektorin *tuhotaan* indeksistä *d*–1. Kyseisessä vektorissa on *m*+*n*+1 indeksipaikkaa samoin kuin vektorissa *diagpos*, ja sen *i.* positioon ($-m \le i \le n$) kerätään listaa niistä *X*:n merkeistä, jotka on jouduttu poistamaan edettäessä matriisin vasemmasta yläkulmasta tarkasteltavana olevaan soluun. Palattakoon jälleen esimerkkiin 4.3. Kun on edetty diagonaalia -1 pitkin soluun (4, 3), on matkan varrella kyseiseen soluun jouduttu yhden kerran siirtymään alaspäin. Siirtyminen tapahtui riviltä 2 riville 3 diagonaalilta 0 diagonaalille -1. Tuolloin *X*:n kolmas merkki ”A” joudutaan poistamaan, joten listan *tuhotaan*[-1] alkuun asetetaan poistettavan *X*:n merkin rivinumero 3.

Kun algoritmin laskenta aikanaan etenee diagonaalilla *n*–*m* sijaitsevaan soluun (*m*, *n*) asti, saadaan *LE*(*X*, *Y*):n arvoksi ulomman silmukan laskurin mukainen etäisyysarvo. PYA-jono pystytään palauttamaan selaamalla vektori *X* lopusta alkuun päin ja tulostamalla siitä kaikki ne merkit, joiden indeksiarvoa ei esiinny paikasta *tuhotaan*[*n*–*m*] alkavassa listassa.

MMY-algoritmin edellä esitetyt siirtymissäännöt diagonaalia pitkin, alemmalta diagonaalilta suoraan oikealle ja ylemmältä diagonaalilta suoraan alas voidaan koota kolmeksi alkuperäisartikkelissa [Mil85] esitetyksi lemmaksi, jotka esitetään seuraavassa[112]. Kannattaa huomioida, että algoritmin toimivuuden kannalta on oleellista, että etäisyysarvo *k* asetetaan kaikille sen sisältäville diagonaaleille ennen, kuin yhtään etäisyysarvoista *k*+1 on ehditty asettaa. Koska pienintä etäisyysarvoa eli nollaa voi esiintyä ainoastaan matriisin päädiagonaalilla, pitää algoritmin suoritus aloittaa soveltamalla lemmaa 4.3 eli etenemistä diagonaalia pitkin.

Lemma 4.1: *Siirrytään matriisissa M oikealle päin riviä i pitkin. Oletetaan, että seuraavat kolme ehtoa ovat voimassa:*

i) $M_{i,j-1}$ *on laskettuna.*

ii) Tiedetään komentojono Z, joka kuvaa, mitkä X[1..*i*]*:n merkeistä eivät kuulu X*[1..*i*]*:n* ja *Y*[1..*j*–1]*:n PYAan.*

iii) $M_{i,j}$ *on tuntematon*

[112] Lemmojen todistukset on kuitenkin sivuutettu tässä työssä, sillä niiden paikkansapitävyys on verrattain ilmeinen. Myös alkuperäisartikkelista laskentasääntöjen eksplisiittiset todistukset puuttuvat.

Tällöin $M_{i,j} = M_{i,j-1} + 1$ ja Z on sama komentojono kuin ehdossa ii)[113], *mutta se ilmaisee nyt, mitkä $X[1..i]$:n merkeistä eivät kuulu $X[1..i]$:n* ja *$Y[1..j]$:n PYAan.*

Lemma 4.2: *Siirrytään matriisissa M alaspäin saraketta j pitkin. Oletetaan, että seuraavat kolme ehtoa ovat voimassa:*

i) $M_{i-1,j}$ on laskettuna
ii) Tiedetään komentojono Z, joka kuvaa, mitkä $X[1..i-1]$:n merkeistä eivät kuulu $X[1..i-1]$:n ja *$Y[1..j]$:n PYAan.*
iii) $M_{i,j}$ on tuntematon

Tällöin $M_{i,j} = M_{i-1,j} + 1$ ja komentojonon Z alkuun on lisättävä komento ”Tuhoa i. symboli vektorista X”. Komentojono Z ilmaisee nyt, mitkä $X[1..i]$:n merkeistä eivät kuulu $X[1..i]$:n ja *$Y[1..j]$:n PYAan.*

Lemma 4.3: *Siirrytään matriisissa M pitkin diagonaalia. Oletetaan, että seuraavat kolme ehtoa ovat voimassa:*

i) Matriisin solun $M_{i-1,j-1}$ arvo on laskettuna.
ii) Tiedetään komentojono Z, joka kuvaa, mitkä $X[1..i-1]$:n merkeistä eivät kuulu $X[1..i-1]$:n ja *$Y[1..j-1]$:n PYAan.*
iii) $x_i = y_j$

Tällöin $M_{i,j} = M_{i-1,j-1}$, ja Z on sama komentojono kuin ehdossa ii), mutta se ilmaisee nyt, mitkä $X[1..i]$:n merkeistä eivät kuulu $X[1..i]$:n ja $Y[1..j]$:n PYAan.

Liitteen kohdassa 11.5.1 esitetään MMY-algoritmin pseudokoodilistaus. Algoritmia on muunneltu alkuperäisversiostaan siten, että se palauttaa syötejonojensa PYAn ja sen pituuden. Sen sijaan komentojonoa syötejonon X muuttamiseksi Y:n kaltaiseksi ei esitetä.

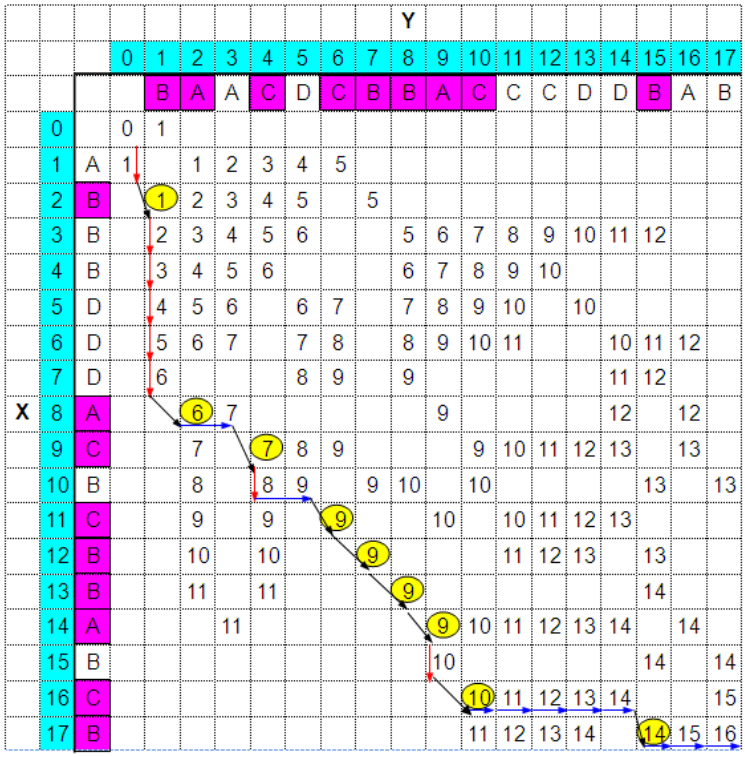
Seuraavaksi esitetään esimerkki MMY-algoritmin toiminnasta kahdelle 17 merkin pituiselle syötevektorille. Algoritmin löytämällä ratkaisupolulla siirtymiset alaviistoon diagonaalia pitkin on merkitty kuvaan **mustilla** nuolilla, oikealle riviä pitkin **sinisillä** nuolilla ja alas saraketta pitkin **punaisilla** nuolilla. Matriisin oikea alakulma saavutetaan etäisyysarvolla 16 diagonaalilla 0. PYAn pituudeksi saadaan 9 sijoittamalla yhtälöön $p = ½(m+n-LE)$ kummankin syötejonon pituus 17 ja niiden lyhin editointietäisyys 16.

PYA-jonoon eivät tule valituiksi merkit niiltä riveiltä, joille ratkaisupolulla on siirrytty punaisella nuolella. Nämä rivinumerot löytyvät nyt laskevassa

[113] Jos haluttaisiin selvittää myös merkkijonon X merkkijonoksi Y muuntava komentojono eikä pelkästään jonojen PYAa, pitäisi komentojonon Z loppuun liittää vielä komento ”Lisää Y:n j. symboli vektoriin X”. Tämä ei kuitenkaan ole tarpeen pelkästään PYAa ratkaistaessa.

suuruusjärjestyksessä vektorin *tuhotaan* indeksipaikasta 0. Matriisiin on merkitty kaikkien algoritmin suorituksen aikana tutkittujen alkuliiteparien lyhimmät editointietäisyydet. Tyhjiä ruutuja vastaavia alkuliitepareja ei jouduta tutkimaan. Esimerkkisyötteillä yhteensä 130 alkuliiteparin lyhimmät editointietäisyydet jouduttaisiin laskemaan, mikä vastaa hieman yli 40 %:a taulukon soluista.

Esimerkki 4.4: *MMY-algoritmin löytämä ratkaisu esimerkin 3.1 syötejonojen PYAlle.*



4.1.1.4 Aika- ja tilavaativuudesta[114]

Kuten jo edellä todettiin, MMY-algoritmissa ei ole mitään varsinaista esiprosessointivaihetta, vaan sen ajankäyttö koostuu yksinomaan *LE*:n määräämisestä ja PYAn keräämisestä. Algoritmin toiminnallisen ytimen muodostavat kaksi sisäkkäistä silmukkaa *S1* ja *S2*. Näistä ulompaa eli *S1*:tä joudutaan suorittamaan yhtä monta kierrosta kuin syötejonojen lyhin editointietäisyys on. Vastaavasti sisempää silmukkaa *S2* suoritetaan enintään jokaista sellaista diagonaalia kohti, joilla voi esiintyä ulomman silmukan etäisyysarvoa *k*. Kyseeseen tulevia diagonaaleja on yhteensä *k*+1 kappaletta. Silmukasta *S2* kutsutaan vielä proseduuria *Tutki_diagonaali_k*, joka pyrkii löytämään

[114] Tekijöiden esittämässä analyysissä algoritmin kompleksisuus riippuu syötejonojen *LE*:stä. Tässä esitettävässä analyysissä kyseisestä termistä on päästy eroon käyttämällä hyväksi tietoa, että *LE* voidaan korvata summalla $m + n - 2p$.

perättäisiä täsmäyksiä kulloinkin tutkittavalta diagonaalilta. Pahimmassa tapauksessa tarkasteltava diagonaali joudutaan tutkimaan loppuun asti.

Tarkastellaan seuraavaksi, mitä tahtia tutkittavien diagonaalien määrä kasvaa ulomman silmukan laskurina toimivan etäisyysarvon *k* kasvaessa. Ensimmäisellä kierroksella *k*:n arvolla 0 tutkitaan ainoastaan päädiagonaalia. Toisesta kierroksesta lähtien tulee joka kierroksella tarkasteltaviksi kaksi uutta diagonaalia: -*k* ja *k*. Kaikilla näiden väliin jäävillä diagonaaleilla on ehditty vierailla jo aikaisemmin. Koska ulomman silmukan suorittaminen päättyy *k*:n arvolla $LE(X, Y)$, ehditään suorituksen aikana tutkia korkeintaan $2LE+1$ diagonaalia[115]. Koska $LE = m + n - 2p \leq 2(n-p) = O(n-p)$, voidaan kyseisellä ylärajalla rajoittaa tutkittavien diagonaalien lukumäärää. Koska kukin diagonaaleista on pituudeltaan enintään lyhyemmän syötevektorin mittainen, rajoittaa algoritmin laskentavaiheen kustannusta ylhäältä siten lauseke $O(m(n-p))$. PYA-jonon keräämisvaihe vaatii ainoastaan yhden syötevektorin *X* selauskerran, eli sen suorituskustannus $O(m)$ on laskentavaiheen kustannukseen nähden vähäinen. Koko algoritmin aikakompleksisuus on siis $O(m(n-p))$, joka on aivan sama kuin eri lähestymistavalla toimivalla NKY-algoritmilla. MMY toimii siten nopeasti, kun syötejonojen PYA on pitkä. Pahimmassa tapauksessa työtä tehdään $O(mn)$.

Algoritmi vaatii syötejonojensa tallentamisen lisäksi ei-vakiomääräistä muistitilaa vektoreita *diagpos* ja *tuhotaan* varten. Kumpainenkin vektoreista on $(m+n+1)$:n mittainen. Ensin mainittuun tallennetaan kokonaislukuja, mutta jälkimmäisessä ylläpidetään linkitetyissä listoissa komentojonoja, joista käy ilmi, mitkä *X*:n merkeistä pitää poistaa PYAn rakentamiseksi. Koska oletuksemme mukaan $m \leq n$, voi etäisyysarvoa *k* kohti enintään puolet operaatioista olla poistoja vektorista *X*; muutoinhan *X* lyhenisi komentojonon vaikutuksesta. Siten yhden komentojonon pituus voi olla enintään $\frac{1}{2}LE$. Koska diagonaaleja joudutaan tutkimaan edellä esitetyn analyysin perusteella enintään $2LE+1$, on komentojonolistojen pituuksien summan verrattain karkea[116] yläraja $O(LE^2) = O((n-p)^2)$. Algoritmin tilantarve pysyy sen mukaan lineaarisena vain, jos $p \geq n - \sqrt{n}$.

4.1.2 Myersin algoritmi (MYE)

4.1.2.1 Lineaaritilainen lyhimmän editointietäisyyden laskeva menetelmä

Eugene W. Myers esitti vuonna 1986 parannetun versionsa [Mye86] vain vuotta aiemmin yhdessä *Millerin* kanssa kehittämästään MMY-algoritmista, jota tarkasteltiin juuri edellä. Myers esittelee artikkelissaan erillisen algoritmin *LCS*, jonka tarkoituksena

[115] Vähempikin määrä riittää, kun osutaan jo ennen kierrosta *LE* matriisin ala- tai oikeaan reunaan, jolloin tarpeettomia diagonaaleja voidaan jo jättää tutkimatta.
[116] Diagonaalille, jonka numero > $n - m + p$, ei enää mahdu $m - p$ tuhoamisoperaatiota.

on ratkaista erityisesti syötejonojensa PYA. Tässä työssä algoritmista käytetään tunnistetta *MYE*.

MYE-algoritmin toimintatapa muistuttaa voimakkaasti edeltäjänsä MMY:n laskentamallia, jonka ytimenä olivat kolme etenemissääntöä: joko riviä pitkin oikealle, saraketta pitkin alas tai diagonaalia pitkin oikealle alaviistoon. Nämä piirteet ovat periytyneet myös MYE:hen – samoin kuin esiprosessoinnin minimaalisuus ja ajatus taustalla olevasta matriisista *M*. Siinä on kuitenkin myös kaksi uutta teknistä ominaisuutta edeltäjäänsä verrattuna. Ensimmäinen niistä on *lyhimmän editointietäisyyden laskennan kaksisuuntaisuus*. Kun MMY:ssä lähdetään liikkeelle ainoastaan matriisin *M* vasemmasta yläkulmasta[117] ja pyritään kolmea siirtymissääntöä noudattamalla etenemään maalisolmuun (*m*, *n*) asti, MYE laskee ulomman silmukan kutakin etäisyysarvoa 0, 1, 2, … kohti etäisyydet *myös maalisolmusta alkuun päin* ennen seuraavan etäisyysarvon mukaisten lävistäjien tutkimista. Toisena tärkeänä erona MYE:ssä on edeltäjäänsä verrattuna vektorin *tuhotaan* puuttuminen. Kyseistä linkitettyjä listoja varastoivaa vektoria tarvitaan MMY:ssä *X*:stä poistettavien symbolien kirjaamiseen ylläpidettävillä hakupoluilla. Koska se on MMY:ssä ainoa tietorakenne, jonka takia algoritmi ei toimi lineaarisessa muistitilassa suhteessa syötejonojensa pituuteen, saadaan MYE vektorin *tuhotaan* poistamisen ansiosta toimimaan lineaarisessa muistitilassa.

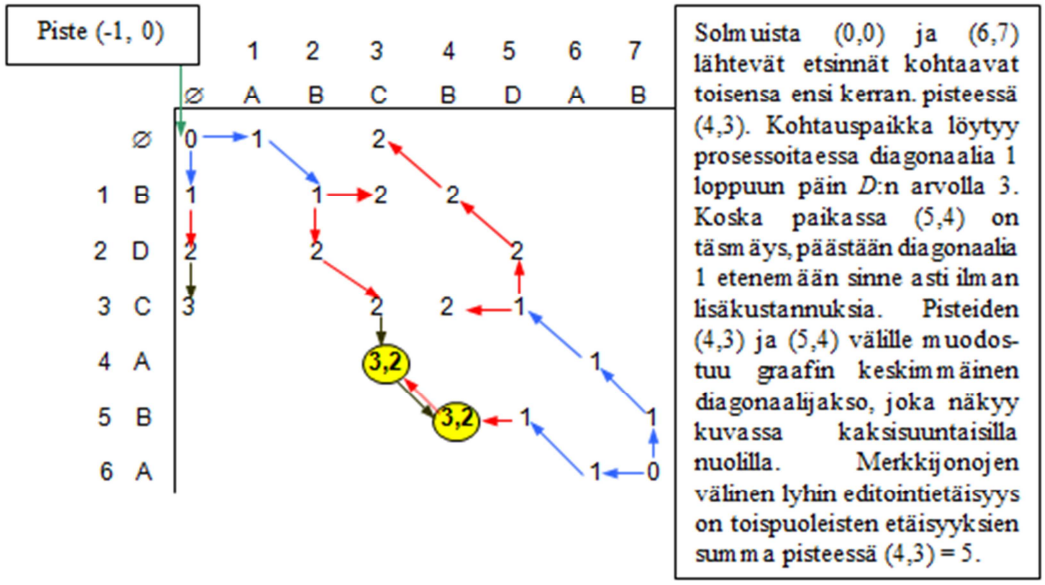
MYE:ssä joudutaan diagonaalien kaksisuuntaisen prosessoinnin tähden perustamaan kaksi erillistä $m+n+1$:n mittaista vektoria, jonka positioihin $-m..n$ tallennetaan tieto siitä, *miten pitkälle kutakin lävistäjää pitkin on ehditty edetä taustalla olevassa matriisissa M*. Diagonaalia pitkin saadaan MMY:n tapaan edetä pisteestä (x_i, y_j) *alaviistoon* niin kauan kuin on voimassa $x_{i+1} = y_{j+1}$. Kun MMY:ssä tähän tarkoitukseen riitti pelkästään vektori *diagpos*, sen tilalla MYE:ssä ovat vektorit *DE* ja *DT*, joista ensin mainitun positioon i ($-m \le i \le n$) kirjataan diagonaalia i pitkin ylhäältä alaspäin etenemällä saavutettu suurin rivinumero. Vastaavasti *DT*:n samaiseen positioon säilötään tieto siitä, mikä on *pienin rivinumero*, jolle asti diagonaalia i on ehditty edetä diagonaalin lopusta alkuun päin. Vektori *DE* alustetaan nollilla ja vektori *DT* puolestaan arvoilla $m+1$. Koska MYE-algoritmissa ei paluusuuntaista prosessointia varten ole käytettävissä mitään (0, 0):n kaltaista pseudotäsmäystä, vaan aloitus tapahtuu suoraan solmusta (*m*, *n*), on diagonaalien *DT* sisältämillä arvoilla eri tulkinta kuin vektoriin *DE* tallentuvilla arvoilla. Paluusuuntaan prosessoitaessa pisteestä (x_i, y_j) saadaan edetä *yläviistoon* pisteeseen (x_{i-1}, y_{j-1}) niin kauan, kuin $x_i = y_j$. Siinä missä $DE[k] = u$ kuvaa alkuliitteiden $X[1..u]$ ja $Y[1..u+k]$ ($0 \le u \le m$, $-m \le k \le n$) *lyhintä editointietäisyyttä*, arvo $DT[k] = u$ osoittaakin *halvimman polun pituuden* pisteestä (x_u, y_{u+k}) matriisin *M* oikean alakulman pisteeseen (*m*, *n*).

[117] Algoritmissa MYE on lisäksi erityinen pseudosolmu (-1, 0), josta siirtymällä suoraan alaspäin päästään laskennan varsinaiseen alkusolmuun (0, 0) diagonaalille 0.

Kannattaa huomioida, että tiettyä ulomman silmukan laskurin etäisyysarvoa D voi esiintyä vektorissa DE lävistäjillä $-D$, $-D+2$, $-D+4$, …, $D-2$, D, mutta näitä etäisyysarvoja mahdollisesti sisältävät diagonaalit vektorissa DT ovatkin sitä vastoin $\Delta-D$, $\Delta-D+2$, $\Delta-D+4$, …, $\Delta+D-2$, $\Delta+D$, missä $\Delta = n-m$ eli syötteiden pituuksien erotus – s. o. syötteiden välinen teoreettisesti pienin mahdollinen editointietäisyys. Tämä on intuitiivisesti ajatellen selvää, sillä lopusta alkuun laskentaa suoritettaessa ainoastaan lävistäjällä $n-m$, joka päättyy oikean alanurkan soluun (m, n), voi esiintyä editointietäisyyttä 0. Jos syötejonot ovat samanpituiset keskenään, tutkittavat vastindiagonaalit ovat etäisyysarvoittain samat.

Yhtä ulomman silmukan kierrosta numero D kohti edetään ensinnä MMY:n tapaan kutakin mahdollista diagonaalia pitkin ylhäältä alas niin pitkälle kuin täsmäyksiä sillä riittää. Tämä tapahtuu MYEn ensimmäisessä sisemmässä silmukassa. Sen sijaan tämän jälkeen tutkitaan kaikki etäisyysarvoa D sisältävät diagonaalit lopusta alkuun päin toisessa sisemmässä silmukassa, ennen kuin siirrytään tutkimaan seuraavaa etäisyysarvoa $D+1$. Kun jossain vaiheessa päädytään tilanteeseen, että vastakkaissuuntaiset etsinnät kohtaavat toisensa jollain diagonaalilla, on löydetty koko syötejonojen lyhin editointietäisyys. Jos kohtaaminen tapahtuu menosuuntaan edettäessä kohti maalisolmua (m, n), on jonojen välinen LE pariton, ja sen arvo on $2D - 1$. Muussa tapauksessa LE on parillinen, ja sen pituus saadaan kertomalla kahdella ulomman silmukan laskurin arvo D. Mikäli kohtaamispaikassa sijaitsee vähintään yksi täsmäys, eli siihen on saavuttu toisesta suunnasta etenemällä diagonaalia pitkin ala- tai yläviistoon, käytetään kohtaamispaikasta alkavasta, jompaankumpaan suuntaan diagonaalia pitkin aukeavasta täsmäysten ketjusta nimitystä *optimaalisen polun keskimmäinen diagonaalijakso* (engl. *middle snake*). Ellei kohtaamispaikan ympäristössä ole yhtään täsmäystä, keskimmäinen diagonaalijakso puristuu pelkäksi pisteeksi. Seuraavassa esitetään esimerkki diagonaalien tutkimisen etenemisestä MYE-algoritmissa. Kuviossa esiintyvien nuolten väri kuvaa tarkasteltavaa etäisyysarvoa pisteeseen saavuttaessa: 0 = **vihreä**, 1 = **sininen**, 2 = **punainen** ja 3 = **musta**.

Esimerkki 4.5: *MYE-algoritmin prosessoinnin eteneminen, kun X* = ”BDCABA”, *Y* =”ABCBDAB”, *p* = 4, *D* = 5.



Keskimmäisen diagonaalijakson löytymistä testataan joka kerta, kun ulomman silmukan laskurin ilmaiseman *etäisyysarvon D alin esiintymisrivi* on selvitetty *menosuunnassa* jollekin diagonaaleista $k \in [-D, -D+2, \ldots D-2, D]$ ja talletettu paikkaan $DE[k]$. Samoin toimitaan, kun *paluusuunnassa* mille tahansa diagonaaleista $k \in [\Delta-D, \Delta-D+2, \ldots, \Delta+D-2, \Delta+D]$ saman *etäisyysarvon D ylin sijaintirivi* on tullut selvitetyksi ja otettu talteen paikkaan $DT[k]$. Etsintöjen kohtaaminen tunnistetaan siitä, kun ensimmäistä kertaa toteutuu $DE[k] \geq DT[k]$. Seuraava lemma määrittää keskimmäisen diagonaalijakson ominaisuuksia. Lukijan työn helpottamiseksi myös sen todistus esitetään tässä[118].

Lemma 4.4: *Merkitään keskimmäisen diagonaalijakson ylempää päätepistettä koordinaatein* (κ_1, λ_1) *ja alempaa vastaavasti* (κ_2, λ_2). *Jos ne eivät esitä samaa pistettä, muodostavat mahdollisesti jakson ylintä pistettä* (κ_1, λ_1) *lukuun ottamatta kaikki sen indeksiparit täsmäyksen, ja lisäksi tiedetään, että kyseiset täsmäykset kuuluvat samalla syötejonojen PYAan.*

Todistus: Oletetaan ensin, että keskimmäinen diagonaalijakso löytyy *menosuuntaisen* prosessoinnin aikana diagonaalilta k ja siihen kuuluu vähintään yksi täsmäys. Sen on selvästikin täytynyt löytyä jo prosessoitaessa samaa diagonaalia aikaisemmin lopusta alkuun päin, sillä paluusuuntaisessa prosessoinnissa diagonaalia pitkin siirrytään pisteestä (x_i, y_j) yläviistoon niin pitkään kuin $x_i = y_j$, eli koko keskimmäinen diagonaalijakso olisi ehditty selata lopusta alkuun asti aina pisteeseen (κ_1, λ_1) asti, jonka merkit eivät enää muodosta täsmäystä. Jos puolestaan keskimmäinen diagonaalijakso

[118] Todistus on esitetty jo alkuperäisartikkelissa [Mye86].

löydetään *paluusuuntaisella* prosessoinnilla, on sen päätepisteen täytynyt löytyä aikaisemmin prosessoitaessa samaa diagonaalia menosuuntaan. Tällöin diagonaalia pitkin on edetty pisteestä (i, j) pisteeseen $(i+1, j+1)$ aina niin kauan kuin $x_{i+1} = y_{j+1}$ Siten keskimmäisen diagonaalijakson kaikki muut pisteet paitsi sen alkupiste (κ_1, λ_1) muodostavat välttämättä täsmäyksen. Koska tarkasteltava etäisyysarvo on pienin, jolla etsinnät kohtaavat, polulla minimoituu syötejonojen välinen editointietäisyys. Luvun 4 alussa esitetyn yhtälön perusteella syötejonojen PYA pitenee samalla kun niiden *LE* lyhenee. Siten polun kaikki täsmäykset – erityisesti keskimmäisen diagonaalijakson täsmäykset – kuuluvat syötejonojen PYAan. □

4.1.2.2 PYAn rekursiivinen määrääminen

Optimaalisen polun keskimmäisen diagonaalijakson löydyttyä PYAn pituus *p* saadaan laskettua saman tien edellä esitetyn yhtälön perusteella, mutta ratkaisuksi kelpaavan jonon etsintä teettää nyt enemmän töitä kuin MMY:ssä, sillä toisin kuin MMY-algoritmi, MYE ei muista vektorista *X* tuhottuja merkkejä optimaalisella polulla alkupisteestä (0, 0) loppupisteeseen (m, n). PYAn määrääminen vaatii MYE-algoritmin kutsumista *rekursiivisesti*. Algoritmin taustalla oleva matriisi *M* jaetaan sitä varten kolmeen alueeseen, jotka ovat sijaintinsa puolesta seuraavanlaiset:

1) *keskimmäistä diagonaalijaksoa edeltävät alkuliitteet* $X[1..\kappa_1]$, $Y[1..\lambda_1]$ *sen mahdollisesti täsmäystä muodostamaton*[119] *alkupiste* (κ_1, λ_1) *mukaan lukien*
2) *keskimmäisen diagonaalijakson täsmäysalueen* $X[\kappa_1+1..\kappa_2]$, $Y[\lambda_1+1..\lambda_2]$ *merkit*
3) *keskimmäistä diagonaalijaksoa seuraavat loppuliitteet* $X[\kappa_2+1..m]$, $Y[\lambda_2+1..n]$.

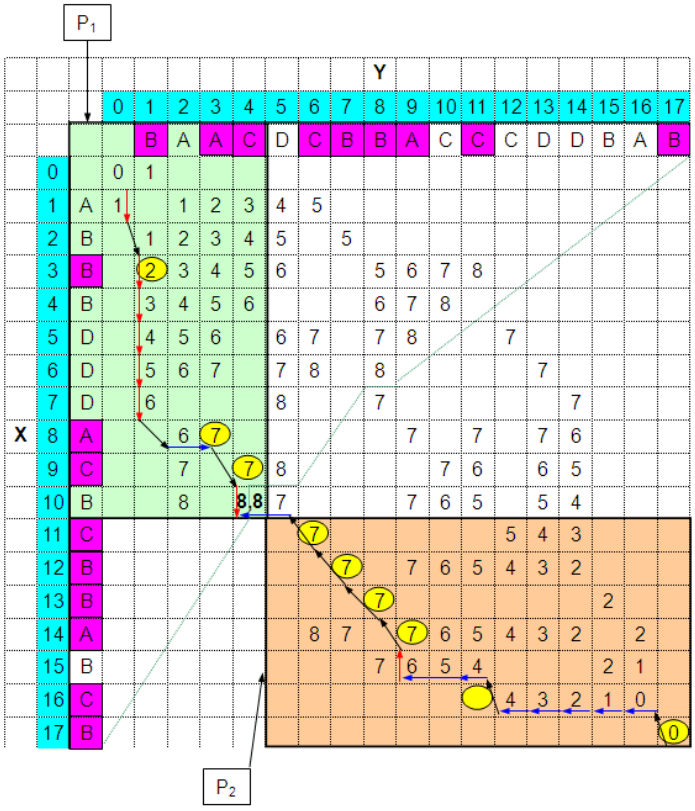
Edellä mainituista alueista 1) sijaitsee matriisin *M* vasemmassa yläkulmassa, 3) oikeassa alakulmassa, ja näitä kahta aluetta yhdistää toisiinsa keskimmäinen diagonaalijakso 2) ilman ylempää päätepistettään. Mikä tahansa edellä mainituista alueista voi jäädä myös tyhjäksi. MYE-algoritmi muodostaa edellä kuvatun alueisiin jaon tapahduttua rekursiivisen kutsun ensiksi alueelle 1. Kun koko alue on tullut rekursiivisesti käsiteltyä, liitetään PYAan alueeseen 2 kuuluvat merkit, minkä jälkeen MYE-algoritmia kutsutaan rekursiivisesti alueelle 3. Rekursio alkaa palautua, kun tarkasteltavan alueen jompikumpi syötevektoreista tyhjenee, tai osavektoreiden lyhin editointietäisyys kutistuu korkeintaan yhden mittaiseksi. Ensin mainitussa tapauksessa tarkasteltavan alueen PYAn pituus on triviaalisti nolla, ja jälkimmäisessä tapauksessa alueen PYA on sama kuin tarkasteltavan aliongelman lyhyempi osavektori.

[119] Myös keskimmäisen diagonaalijakson alkupiste (κ_1, λ_1) voi muodostaa täsmäyksen, jos se sijaitsee tarkasteltavan alueen vasemmassa reunassa tai yläreunassa.

MYE-algoritmin pseudokoodilistaus on nähtävissä liitesivuilla kohdassa 11.5.2. Huomion arvoista on, ettei artikkelissa [Mye86] esitetyssä algoritmissa ole mukana mitään rajatestejä sille, pysytäänkö diagonaaleja pitkin edettäessä syötevektoreiden määritellyllä indeksialueella. Niitä ei ole asetettu myöskään liitteessä esitettyyn pseudokoodiin.

Seuraavassa esitetään vielä MYE-algoritmin ensimmäisen kutsun aikana muodostuva aluejako sekä algoritmin löytämä PYAn instanssi esimerkin 3.1 kaltaiselle syöteaineistolle (**keltaisilla** soikioilla merkityt solut vastaavat PYAn merkkien indeksipareja). Matriisin soluihin tallentuvat luvut ovat toispuoleisia, yhteen suuntaan laskettuja etäisyysarvoja. Keskimmäisen diagonaalijakson soluihin on merkitty etäisyysarvot kummastakin suunnasta saavuttaessa.

Esimerkki 4.6: *PYAn ratkaiseminen MYE-algoritmilla esimerkin 3.1 mukaiselle syöteaineistolle. Kuvassa näkyy aluejako ensimmäisen rekursiivisen kutsun jälkeen. Algoritmia joudutaan kutsumaan seuraavalla rekursiotasolla alueille P1 ja P2. **Vihreä** pisteviiva erottelee erisuuntaisilla etsinnöillä tutkitut alueet toisistaan.*



4.1.2.3 Aika- ja tilavaativuudesta

Myersin algoritmin alustusvaihe on MMY:n tavoin nopea, koska se ei kerää mitään erityisiä tietoja syötejonoistaan ennen laskentavaiheen alkua. Ainoastaan $m+n+1$

-paikkaiset vektorit *DE* ja *DT* joudutaan alustamaan. Koko esiprosessointi tapahtuu siten ajassa $O(n)$.

Laskentavaiheessa joudutaan ulommassa silmukassa käymään läpi etäisyysarvot 0, 1, 2, …, $\lceil D/2 \rceil$. Näiden aikana joudutaan tutkimaan yhteensä enintään $2\lceil D/2 \rceil+1 \leq 2(D/2 + 1)+1 = 2D+3 = LE+3$ diagonaalia, joista jokaisen pituus on enintään *m*. Koska $LE = ½(m+n–2p)$, on PYAn pituuden laskentavaiheen suorituskustannus $O(m(n–p))$, eli päästään samaan teoreettiseen ylärajaan kuin algoritmissa MMY[120]. Käytännössä on kuitenkin odotettavissa, että MYE toimisi edeltäjäänsä MMY:tä nopeammin, sillä jos syötejonojen välinen etäisyys < $(m+n)/2$, MYEssä tutkitaan parhaimmassa tapauksessa vain noin puolet diagonaaleista. Esimerkiksi syötejonojen ollessa keskenään yhtä pitkät ja niiden editointietäisyyden ollessa 2*k*, voidaan MMY:ssä joutua tutkimaan kaikkia diagonaaleja väliltä [-2*k*..2*k*], kun taas pelkästään diagonaalien [-*k*..*k*] tutkiminen riittäisi MYE:ssä.

PYA-jonon palauttaminen ei MYE:ssä ole kuitenkaan niin vaivatonta kuin MMY:ssä, sillä MYE ei pidä kirjaa hakupolkujen varrella tuhotuista *X*:n merkeistä. Siten on laskettava, mikä on rekursiivisten kutsujen yhteenlaskettu kustannus. Seuraavalla todistuksella pystytään osoittamaan, ettei PYA-jonon keräämisvaiheella ole vaikutusta MYE-algoritmin asymptoottiseen suoritusaikaan. Koko algoritmin suoritusaikaa kuvaa rekursioyhtälö

$$(i)\ T(P, D) \leq \begin{cases} \alpha PD + T(P_1, \lceil D/2 \rceil) + T(P_3, \lfloor D/2 \rfloor), \text{ kun } D > 1 \\ \beta P, \text{ kun } D \leq 1. \end{cases}$$

Rekursioyhtälössä *P* kuvaa kutsussa esiintyvien syötejonojen yhteenlaskettua pituutta *m*+*n*, ja α sekä β ovat sopivasti valittuja vakiokertoimia. Kun $D > 1$, tulee termistä $2D/3$ arvoltaan suurempi kuin $\lceil D/2 \rceil$, ja vastaavasti $D/2$ on aina $\geq \lfloor D/2 \rfloor$. Edelleen oletetaan, että algoritmin suoritusaika on vakio silloin, kun syötejonojen välinen etäisyys on korkeintaan 1. Rekursioyhtälö voidaan kirjoittaa nyt uudelleen muotoon

$$(ii)\ T(P, D) \leq \begin{cases} \alpha PD + T(P_1, 2D/3) + T(P_3, D/2), \text{ kun } D > 1 \\ \beta P, \text{ kun } D \leq 1. \end{cases}$$

Myersin artikkelissa todetaan, että epäyhtälön ylemmän rivin rekursiiviset termit voidaan yhdistää yhdeksi termiksi $T(P, 2D/3)$ perustelematta väitettä sen kummemmin. Selvyyden vuoksi väite kuitenkin todistetaan seuraavassa. Merkitään alueen 1)

[120] Samoin kuin MMY-algoritmissa, ei MYE:ssä oteta kantaa siihen, kumpi syötejonoista *X* ja *Y* on pidempi. Aliluvussa 2.1 tekemämme oletuksen mukaan kuitenkin $m \leq n$ aina, mistä seuraa, että diagonaalin pituus on aina $\leq m$. Rekursiivisissa algoritmeissa tämä pätee ainakin alkutilanteessa.

syötteiden pituutta muuttujalla P_1 ja alueen 3) syötteiden pituutta muuttujalla P_3. Koska alueiden 1) ja 3) syötteiden yhteenlaskettu pituus P_1+P_2 on enintään $m+n = P$ löydetyn keskimmäisen diagonaalijakson pituudesta riippumatta, voidaan ylemmän rivin epäyhtälössä jälkimmäistä termiä $T(P_3, D/2)$ arvioida ylöspäin kirjoittamalla se muotoon $T(P–P_1, D/2)$. Täten epäyhtälö saa muodon

$$(iii)\ T(P, D) \leq \begin{cases} \alpha PD + T(P_1, 2D/3) + T(P - P_1, D/2), \text{ kun } D > 1 \\ \beta P, \text{ kun } D \leq 1. \end{cases}$$

Iteroimalla epäyhtälöä päästäisiin seuraavaan muotoon[121]:

$T(P, D) \leq \alpha PD+[2\alpha P_1D/3 + \alpha PD/2 - \alpha P_1D/2] + T(P_{11}, 4D/9) + T(P_{13}, D/3) + T(P_{31}, D/3) + T(P_{33}, D/4)$. Tarkastellaan iteraatiokierroksella syntyneiden, hakasulkeiden sisälle merkittyjen uusien ei-rekursiivisten termien summaa. Rekursiiviset termit voidaan jättää huomiotta. Ottamalla vakiokerroin α yhteiseksi tekijäksi saadaan hakasulkulauseke muotoon

$$[\ ] = \alpha(2P_1D/3 + PD/2 - P_1D/2) = \alpha(4P_1D/6 + 3PD/6 - 3P_1D/6) = \alpha/6(P_1D+3PD)$$

Jos kohdan (*ii*) rekursioyhtälön ylin rivi muunnettaisiin Myersin ehdotuksen mukaisesti muotoon $\alpha PD+T(P_1, 2D/3)$, saataisiin sille seuraavalla iteraatiokierroksella muoto $T(P, D) \leq \alpha PD + [2\alpha PD/3] + T(P, 4D/9)$. Ainoa kierroksella syntynyt ei-rekursiivinen termi on $2\alpha PD/3$, josta laventamalla saadaan $4\alpha PD/6$. Jotta Myersin väite pitäisi paikkansa, pitää olla voimassa $4\alpha PD/6 \geq \alpha/6(P_1D+3PD)$. Ratkaistaan kyseinen epäyhtälö:

$$4\alpha PD/6 \geq \alpha/6(P_1D+3PD) \Leftrightarrow 4PD \geq P_1D+3PD \Leftrightarrow PD \geq P_1D \Leftrightarrow P \geq P_1.$$

Koska kumpikaan alueista 1) ja 3) ei voi olla kooltaan alkuperäistä aluetta suurempi, voidaan väitteen todeta pitävän paikkansa, eli $T(P, D) \leq \alpha PD+T(P, 2D/3)$. □

Siten koko algoritmin suoritusaikaa kuvaa rekursioyhtälö

[121] Muodon rekursiivisissa termeissä on alueen 2 käsittelykustannuksia varten säilytetty merkintä $T(P_3, D/2)$ luettavuuden parantamiseksi.

$$(iv)\ T(P, D) \leq \begin{cases} \alpha PD + T(P, 2D/3), \text{ kun } D > 1 \\ \beta P, \text{ kun } D \leq 1. \end{cases}$$

Iteroimalla epäyhtälön ylempää riviä, kunnes etäisyys D saavuttaa arvon ≤ 1, saadaan kierrosten aikana syntyvien ei-rekursiivisten termien summaksi $\alpha PD + 2/3\alpha PD + 4/9\alpha PD + \ldots$. Termit muodostavat suppenevan *geometrisen sarjan*, jonka summaksi saadaan $3\alpha PD$. Siten koko algoritmin suorituskustannukseksi saadaan $3PD + \beta P$, joka on suuruusluokkaa $O(PD)$. Toisin sanoen rekursiiviset kutsut eivät heikennä MYE:n asymptoottista suoritusaikaa. Koska oletuksemme mukaan $m \leq n$ ja $D = m + n - 2p$, voidaan suoritusaikalausekkeen $O(PD)$ termi P korvata termillä m ja D termillä $n - p$[122]. Siten MYE-algoritmin asymptoottinen suorituskustannus on sama kuin MMY:n eli $O(m(n-p))$.

Algoritmi vaatii ei-vakiotilaista muistia syötevektoreidensa lisäksi vektoreille *DE* ja *DT*, joiden kummankin pituus on $m + n + 1$. Rekursiotasoja tarvitaan etäisyysarvon 1 saavuttamiseksi $\lceil \log D \rceil$ kappaletta, ja eri tasoilla voidaan käyttää hyväksi globaaleja vektoreita *DE* ja *DT*, sillä aikaisempien kutsujen aikana laskettuja kyseisten vektoreiden arvoja ei tarvitse muistaa. Siten koko algoritmi toimii lineaarisessa muistitilassa $O(n)$.

4.1.3 Wun, Manberin, Myersin ja Millerin algoritmi (WMM)

4.1.3.1 Uusi laskennallinen käsite: puristettu etäisyys

Viimeisenä alun perin kahden merkkijonon lyhimmän editointietäisyyden ratkaisemiksi kehitetyistä algoritmeista esitellään vuonna 1990 ilmestynyt *Wun, Manberin, Myersin ja Millerin* algoritmi [WMM90]. Tekijät käyttävät algoritmistaan nimitystä *Compare*, ja siitä käytetään käsillä olevassa työssä lyhennettä *WMM*. Aikaisemmin esitellystä MMY- ja MYE-algoritmeista tutut piirteet, kuten abstraktio taustalla olevasta matriisista, minimiin supistettu esiprosessointi, diagonaalien numerointi sekä siirtymissäännöt diagonaalilta toiselle, paistavat läpi myös WMM:stä, mihin osaltaan on varmastikin vaikuttanut edellä mainittujen kahden menetelmän kirjoittajien osallistuminen WMM:n kehittämiseen. Erityisesti MMY:n kanssa WMM:llä on runsaasti yhtäläisyyksiä. Tekijöiden julkaisema algoritmi laskee yksinomaan kahden merkkijonon *LE*:n pituuden. Tässä esiteltävä WMM-algoritmin versio on muunnettu alkuperäisartikkelissa esitetystä siten, että se ratkaisee PYAn pituuden sekä jonkin sen esiintymistä.

Vaikka WMM:ssä on useita edeltäjiltään perittyjä piirteitä, algoritmissa esitellään myös yksi täysin uusi käsite – *puristettu etäisyys* (engl. *compressed distance*) – jonka varaan koko algoritmin laskentamalli pitkälti rakentuu. Kahden merkkijonon

[122] kts. alaviite 120

puristetulla etäisyydellä *P* tarkoitetaan *niiden merkkien lukumäärää, jotka joudutaan tuhoamaan vektorista X haluttaessa muuntaa sitä Y:n kaltaiseksi.* Puristettu etäisyys voidaan laskea kaavasta $P = ½(LE–\Delta)$, missä $\Delta = n–m$ eli syötemerkkijonojen pituuksien erotus. Jos syötejonot ovat keskenään yhtä pitkät, on puristettu etäisyys tarkalleen puolet jonojen lyhimmästä editointietäisyydestä: jokaista *X*:stä poistettavaa merkkiä kohti on lisättävä tilalle toinen, jotta *X*:n pituus ei muuttuisi. Jos taasen *Y* on jonoa *X* pidempi, pitää *X*:ään lisätä välttämättä Δ kappaletta merkkejä, jotta jonoista saataisiin keskenään yhtä pitkät. Tämän lukumäärän ylittävistä operaatioista pitää taas selvästikin puolet olla poisto- ja puolet lisäysoperaatioita.

WMM-algoritmissa tarvitaan aputietorakenteeksi Millerin ja Myersin algoritmin tavoin vektoria *diagpos*, joka on indeksoitu välille *-m..n*. Sen *k*. positioon ($-m \le k \le n$) kirjataan tieto suurimmasta sarakenumerosta, jonne on edetty diagonaalia *k* pitkin. Samoin algoritmin prosessointi on MMY:n tapaan vain yksisuuntaista, eli lähdetään liikkeelle pseudotäsmäyksestä (0, 0) ja pyritään etenemään kohti maalisolmua (*m*, *n*). Niin ikään kuten MMY, myöskään WMM ei sisällä rekursiivisia kutsuja. Siten siinä joudutaan pitämään kirjaa *X*:stä poistetuista merkeistä kullakin hakupolulla, mihin tarvitaan avuksi MMY:stä tuttua vektoria *tuhotaan*. Kyseinen vektori on indeksoitu vektorin *diagpos* tapaan välille *-m..n*, ja sen *k*. positioon tallennetaan linkitetty lista niistä *X*:n merkeistä, joiden ylitse on jouduttu hyppäämään edettäessä alkusolmusta (0, 0) pisteeseen (*diagpos*[*k*]*–k, diagpos*[*k*]). Saavuttaessa aikanaan diagonaalilla Δ sijaitsevaan maalisolmuun (*m*, *n*) voidaan PYAn muodostavat *X*:n merkit listata lopusta alkuun päin hyppäämällä niiden *X*:n indeksien ylitse, jotka esiintyvät listassa *tuhotaan*[Δ].

Mikä selkeimmin erottaa WMM:n muista lyhimmän editointietäisyyden laskevista algoritmeista eli MMY:stä ja MYE:stä on *diagonaalien käsittelyjärjestys*, joka WMM:ssä noudattaa *puristetun etäisyyden* mukaista nousevaa suuruusjärjestystä. Toisin sanoen, ensiksi tutkitaan kaikki sellaiset matriisin diagonaalit, joilla voi esiintyä puristettuna etäisyytenä arvoa 0. Määritelmänsä mukaisesti puristettu etäisyys on 0 tarkalleen silloin, kun tarkasteltavasta *X*:n alkuliitteestä ei jouduta hävittämään yhtään merkkiä sen muuntamiseksi yhdenmukaiseksi tarkasteltavan *Y*:n alkuliitteen kanssa. Selvästikin tämä on mahdollista ainoastaan diagonaaleilla $0..\Delta$, sillä negatiivisilla diagonaaleilla on *X*:stä ehditty jo poistaa vähintään yksi merkki, ja jos diagonaalin numero $> \Delta$, sieltä ei enää pystytä etenemään maalisolmuun siirtymättä ainakin kertaalleen pieninumeroisemmalle diagonaalille, mikä merkitsee tulevaisuudessa väistämättä tapahtuvaa *X*:n merkin poistoa. Kulloinkin tarkastelun kohteena oleva puristettu etäisyys toimii algoritmin ulomman silmukan laskurina.

Puristetun etäisyyden ollessa *k* tutkitaan *kaikki* diagonaalit väliltä $-k..k+\Delta$. Kannattaa huomioida, että vaikka lyhintä editointietäisyyttä *D* voi esiintyä ainoastaan joka toisella lävistäjällä väliltä *-D..D*, tiettyä puristettua etäisyysarvoa *k* esiintyy *jokaisella* tutkittavan välin diagonaalilla sen pariteetista riippumatta. Syy tähän on

helposti ymmärrettävissä. Tarkastellaan vaikkapa diagonaalia 0: niin pitkään kuin X:n ja Y:n alkuliitteet ovat identtiset, on niiden välinen lyhin editointietäisyys – ja samalla tietysti myös puristettu etäisyys – nollan mittainen. Olkoot kyseiset alkuliitteet $X[1..i]$ ja $Y[1..i]$ ($1 \leq i \leq m-1$). Kun lopulta löydetään alkuliitteiden ensimmäinen ei-täsmäävä merkkipari (x_{i+1}, y_{i+1}), kasvaa niiden LE nollasta kahteen, koska x_{i+1} joudutaan poistamaan ja vaihtamaan sen tilalle y_{i+1}. Mutta samaisten alkuliitteiden puristettu etäisyys on kuitenkin vain 1, sillä vain yksi X:n merkki on jouduttu poistamaan. Siten puristettua etäisyysarvoa k esiintyy kaikilla diagonaaleilla $-k..k+\Delta$. □

Puristetut etäisyysarvot k lasketaan matriisin diagonaaleille kolmessa eri vaiheessa. Ensiksi ne asetetaan diagonaaleille $-k..\Delta-1$ järjestyksessä pieninumeroisimmasta suurimpaan. Kaikki nämä diagonaalit sijaitsevat maalisolmuun (m, n) päättyvästä lävistäjästä Δ katsottuna *alavasemmalla*. Toisessa vaiheessa arvo k asetetaan laskevassa järjestyksessä samaisen lävistäjän *yläoikealla* sijaitseville diagonaaleille $k+\Delta..\Delta+1$, ja viimeisessä vaiheessa asetetaan arvo k *maalilävistäjälle* Δ.

Tarkastellaan ensiksi arvojen k asettamista maalilävistäjän alapuolelle. Siirtyminen diagonaalille $-k$ tapahtuu aina pystysuoralla transitiolla alas diagonaalilta $-k+1$. Sen sijaan diagonaaleille $u \in [-k+1..\Delta-1]$ voidaan siirtyä kahdella tavalla: joko etenemällä *oikealle* diagonaalilta $u-1$ tai *alas* diagonaalilta $u+1$ sen mukaan, kumpaa kautta diagonaalille u päästään kauemmas. Tarvittavat tiedot löytyvät vektorista *diagpos* positioista $u-1$ ja $u+1$. Siirryttäessä riviä pitkin oikealle ei puristettu etäisyys muutu miksikään, sillä siirtyminen oikealle oltaessa maalilävistäjän alapuolella tarkoittaisi X:ään kohdistuvaa merkin lisäysoperaatiota. Sen sijaan siirtyminen alaspäin samalla alueella tarkoittaisi merkin poistoa X:stä eli puristetun etäisyyden kasvamista yhdellä entisestä. Siten on tärkeää, että asetettaessa puristettua etäisyysarvoa k diagonaalille u muistipaikasta $diagpos[u-1]$ löydetään tieto arvon k oikeanpuoleisimmasta sijaintisarakkeesta diagonaalilla $u-1$. Vastaavasti muistipaikasta $diagpos[u+1]$ olisi saatava selville, mikä on diagonaalilla $u+1$ oikeanpuoleisin sarake, jolla esiintyy puristettua etäisyysarvoa $k-1$, sillä siirtyminen alaspäin tältä diagonaalilta kasvattaa puristetun etäisyyden k:n mittaiseksi. Jotta kaikki vaaditut tiedot olisivat käytettävissä, on maalidiagonaalin alapuolella puristetut etäisyysarvot asetettava kullekin diagonaalille järjestyksessä pieninumeroisimmasta alkaen, jottei myöhemmin tarvittavia tietoja tuhottaisi liian aikaisin.

Tilanne maalilävistäjän yläpuolella on äsken esitetyn peilikuva. Uloimmalle diagonaalille k päästään siirtymään ainoastaan edelliseltä diagonaalilta $k-1$. Nyt kannattaa kuitenkin huomioida, että oltaessa diagonaalin Δ yläpuolella siirtyminen riviä pitkin yhdellä sarakkeella oikealle päin kasvattaa puristettua etäisyyttä ykkösen verran. Tämä johtuu siitä, että siirtymällä oikealle loitonnutaan maalidiagonaalista, ja sille ei pystytä enää palaamaan takaisin siirtymättä jossain vaiheessa alaspäin jotain saraketta pitkin. Täten ikään kuin maksetaan etukäteen vasta myöhemmin tapahtuvasta X.n merkin tuhoamisesta. Siirtyminen muille maalidiagonaalin yläpuolisille lävistäjille $u \in [\Delta+1..k-1]$ tapahtuu samaan tapaan kuin sen alapuolisillekin lävistäjille, eli joko

lävistäjältä u–1 oikealle tai lävistäjältä u+1 alas. Koska siirtyminen oikealle lisää puristettua etäisyyttä yhdellä, on *diagpos*[u–1]:ssä oltava saatavilla tieto oikeanpuoleisimmasta sarakkeesta, jolla diagonaalilla u–1 esiintyy puristettua etäisyysarvoa k–1. Alaspäin siirtymisistä maalidiagonaalin yläpuolella on sen sijaan maksettu jo menomatkalla, joten siirtyminen alaspäin ei lisää puristettua etäisyyttä. Siten paikasta *diagpos*[u+1] pitää löytyä tieto puristetun etäisyysarvon k oikeanpuoleisimmasta sijaintisarakkeesta diagonaalilla u+1. Jotta kaikki tarvittavat tiedot olisivat käytettävissä, on maalidiagonaalin yläpuoliset lävistäjät tutkittava numeroittain vähenevässä suuruusjärjestyksessä.

Viimeiseksi asetetaan puristettu etäisyysarvo k maalilävistäjälle Δ. Kyseinen lävistäjä on sikäli erikoisasemassa, että sille voidaan siirtyä kummalta tahansa diagonaaleista Δ–1 tai Δ+1 puristetun etäisyyden muuttumatta. Siten tälle diagonaalille arvoa k asetettaessa pitää olla tiedossa arvon k oikeanpuoleisin sijaintisarake sekä sen edeltävällä diagonaalilla Δ–1 että seuraavalla diagonaalilla Δ+1. Siirtyminen tapahtuu nytkin siltä diagonaalilta, jolta päästään etenemään kauemmas maalidiagonaalille. Vaadittujen tietojen on jälleen oltava käytettävissä muistipaikoissa *diagpos*[Δ–1] ja *diagpos*[Δ+1]. Puristettu etäisyysarvo k on siten asetettava viimeiseksi lävistäjälle Δ.

Kun on jo ehditty puristetun etäisyysarvon k asettamiseksi siirtyä diagonaalille u joko vasemmalta tai ylhäältä pisteeseen (j–u, j), edetään diagonaalia pitkin alaspäin niin pitkään kuin $x_{j-u+z} = y_{j+z}$. Viimeinen täsmäyksen muodostanut sarakenumero[123] tallennetaan paikkaan *diagpos*[u]. Diagonaalia pitkin eteneminen on WMM-algoritmissa toteutettu kutsumalla erillistä funktiota *Etene_diag*.

4.1.3.2 Arvojen asettaminen diagonaaleille

Kaikki edellä kuvatut laskentasäännöt on tekijöiden esittämässä artikkelissa koottu lemmaksi, jonka tapauksista yksi on artikkelissa myös todistettu. Lemman sanomana on, miten puristetun etäisyysarvon k sijainti määräytyy rekursiivisesti kullekin diagonaaleista $u \in [-k, \Delta+k]$. Koska kuitenkin edellä esitetty kuvaus vakuuttanee lukijan riittämiin laskentasääntöjen oikeellisuudesta, lemman todistus sivuutetaan tässä.

[123] Ellei paikassa sijaitse yhtään täsmäystä, asetetaan *diagpos*[u]:n arvoksi aloitussarake j.

Lemma 4.5:

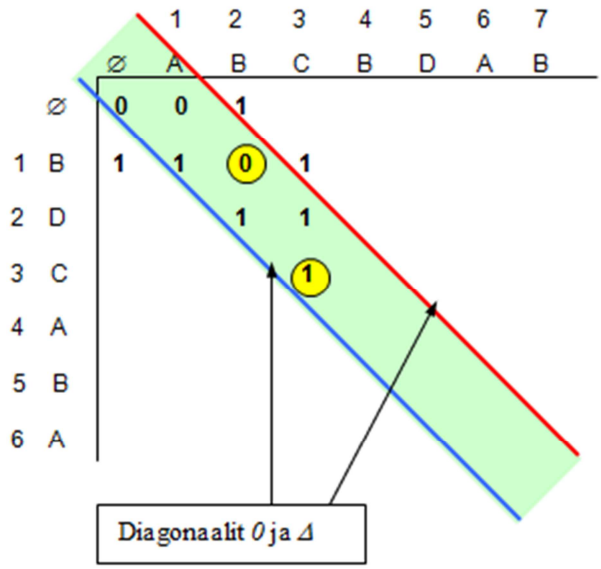
$$diagpos(u, k) = \begin{cases} Etene\_diag(u, Max\{diagpos(u-1, k) + 1, diagpos(u+1, k-1)\}), \\ \quad \text{jos } u \in [-k, \Delta-1], \\ Etene\_diag(u, Max\{diagpos(u-1, k) + 1, diagpos(u+1^{124}, k)\}), \\ \quad \text{jos } u = \Delta, \\ Etene\_diag(u, Max\{diagpos(u-1, k-1) + 1, diagpos(u+1, k)\}), \\ \quad \text{jos } u \in [\Delta+1, k], \end{cases}$$

missä

$Etene\_diag(u, j) = MAX\{z \mid x_{j+1-u}, \ldots, x_{z-u} = y_{j+1}, \ldots, y_z\}$. Ellei *tällaista z:aa ole olemassa, z = j.*

Seuraavaksi esitetään esimerkki, joka havainnollistaa puristetun etäisyyden käsitettä. Siihen on merkitty näkyviin esimerkkisyötteiden kaikki puristetut etäisyysarvot 0 ja 1. Koska syötevektorien pituusero on 1, nollia voi esiintyä tarkalleen diagonaaleilla 0 ja 1 ja ykkösiä näiden diagonaalien lisäksi myös lävistäjillä -1 ja 2.

Esimerkki 4.7: *X* = ”BDCABA”, *Y* = ”ABCBDAB”



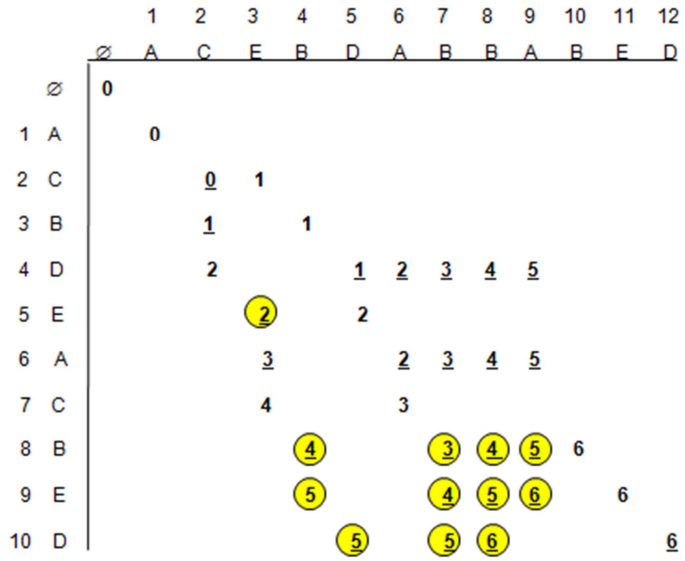
Kuvassa ovat nähtävissä kaikki syötteistä *X* ja *Y* muodostetun matriisin *M* pisteet (*i*, *j*), joiden puristettu etäisyys ≤ 1. Näiden pisteiden kautta voitaisiin teoriassa päästä kulkemaan alkusol-musta (0, 0) maalisolmuun (*m*, *n*) tuhoamalla *X*:stä enintään yksi merkki. Kaikki arvoa nolla sisältävät solut esiintyvät diagonaalien *0* ja *Δ* rajaamalla **vihreällä** alueella. Kannattaa huomioida, että vaikka ”B” on ”ABC”:n alijono, ei pisteen (1, 3) kautta päästä maalisolmuun ilman vähintään yhden *X*:n merkin tuhoamista, sillä piste (1,3) sijaitsee diagonaalin *Δ* = 1 oikealla puolella. Täsmäyksen sisältävät pisteet (*i*, *j*) on merkitty **keltaisin** palloin.

Seuraavat kaksi esimerkkiä on lainattu alkuperäisartikkelista [WMM90]. Ne on otettu mukaan tähän työhön paitsi havainnollisuutensa, niin myös alkuperäismuodossa esiintyvien, ymmärtämistä haittaavien painovirheiden vuoksi. Esimerkeissä esitetään vertailun vuoksi lyhimmät editointietäisyydet sekä puristetut etäisyydet kahdelle syötemerkkijonolle. Esimerkissä 4.8 on esitetty lyhimmät editointietäisyydet MMY:n

[124] Lemman tapaukseen, jossa *k* = *Δ*, oli pullahtanut artikkelissa painovirhe (plus-merkki vaihtunut miinukseksi).

käyttämällä tekniikalla. Artikkelissa virheellisiä arvoja sisältäneet solut on rengastettu **keltaisilla** ympyröillä. Esimerkissä 4.9 ovat sen sijaan WMM:n laskemat puristetut etäisyydet samalle syötemerkkijonoparille. Kummastakin esimerkistä on alkuperäisversioon verrattuna jätetty pois etäisyysarvot niistä soluista, joihin algoritmien suorituksen aikana ei jouduta etenemään. Alleviivatut arvot joudutaan tallentamaan vektoriin *diagpos*. Syötejonojen lyhin editointietäisyys on 6, ja puristettu etäisyys on 2. Esimerkin syöteaineistoilla WMM tuntuisi tutkivan huomattavasti vähemmän soluja kuin MMY.

Esimerkki 4.8: *Lyhimmän editointietäisyyden laskeminen Millerin ja Myersin algoritmia käyttämällä syötteille X =* ”ACBDEACBED” *ja Y =* ”ACEBDABBABED”.



Esimerkki 4.9: *Puristetun etäisyyden laskeminen Wun, Manberin, Myersin ja Millerin algoritmilla syötteille, missä syötevektori X =* ”ACBDEACBED” ja *Y =* ”ACEBDABBABED”.

		1	2	3	4	5	6	7	8	9	10	11	12
		∅	A	C	E	B	D	A	B	B	A	B	E	D
	∅	0												
1	A		0											
2	C			0	0									
3	B			1		0								
4	D			2			0	0	1	2				
5	E				2		1							
6	A							1	1	2				
7	C							2						
8	B								2	2	2	2		
9	E												2	
10	D													2

Seuraavassa annetaan näyte WMM-algoritmin toiminnasta. Matriisiin tallennetut numeroarvot kuvaavat alkuliitteiden puristettuja etäisyyksiä. **Keltaiset** soikiot kuvaavat täsmäyksiä, jotka menetelmä valitsee mukaan löytämäänsä PYAan.

Esimerkki 4.10: *PYAn ratkaiseminen kyseistä tarkoitusta varten muunnellulla WMM-algoritmilla esimerkin 3.1 mukaiselle syöteaineistolle.*

											Y								
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
				B	A	A	C	D	C	B	B	A	C	C	C	D	D	B	A	B
	0		0	1																
	1	A	1		1	2	3	4	5											
	2	B		1	1	2	3	4		5										
	3	B		2	2	2	3	4			5	6	7	8						
	4	B		3	3	3	3				5	6	7	8						
	5	D		4	4	4		3	4		5	6	7	8						
	6	D		5	5	5		4	4		5	6	7	8						
	7	D		6				5	5		5	6	7	8						
X	8	A			6	6					5	6								
	9	C			7	6	6	6					6	7	8					
	10	B			8		7	7		6	6	6								
	11	C					8		7				6	6	7	8				
	12	B								7										
	13	B									7									
	14	A										7	7	7	7	7	7			
	15	B										8						7	8	
	16	C											8	8	8	8	8			
	17	B																8	8	8

Laskettaessa esimerkkimatriisin ei-tyhjien solujen lukumäärä saadaan tulokseksi 98. Tämä on hieman vähemmän kuin MYE-algoritmin vastaava lukumäärä 105 mutta huomattavasti vähemmän kuin MMY-algoritmin läpikäymät 130 solua. Esimerkki ei kuitenkaan ole edes erityisen suosiollinen WMM-algoritmille, sillä syötejonot ovat yhtä pitkiä. Mitä suuremmat pituuserot syötteillä on, sitä selkeämmin odottaisi WMM-algoritmin voittavan tehokkuudessa muut lyhimmän editointietäisyyden laskevat menetelmät!

4.1.3.3 WMM-algoritmin aika- ja tilavaativuudesta

WMM-algoritmin pseudokoodi löytyy liitesivuilta kohdasta 11.5.3. Kuten MMY:n ja MYE:n, niin myös WMM-algoritmin esiprosessointi on minimaalinen. Ainoastaan vektorin *diagpos*, jonka pituus on $m+n+1$, jokainen positio joudutaan alustamaan arvolla -1. Siten esiprosessoinnin kustannus on $\mathcal{O}(n)$.

Laskentavaiheessa joudutaan tutkimaan kaikki matriisin diagonaalit, joilla esiintyy puristettuja etäisyysarvoja $0..P$, missä P on koko syötejonojen puristettu etäisyys. Näitä diagonaaleja on $2P+\Delta+1 = LE+1$, ja jokaisen tutkittavan diagonaalin pituus on $\leq m$. Koska $LE = m+n-2p = \mathcal{O}(n-p)$, saadaan algoritmin laskentavaiheen suoritus-kustannukseksi $m(n-p)$.

PYAn kerääminen voidaan suorittaa rullaamalla vektori X kertaalleen lopusta alkuun päin ja tulostamalla käänteisessä järjestyksessä kaikki ne merkit, joiden indeksejä ei löydy muistipaikasta *tuhotaan*[Δ] alkavasta linkitetystä listasta. Siten koko algoritmin suorituskustannus on $\mathcal{O}(m(n-p))$, joka on asymptoottisesti sama kuin edeltäjillään MMY ja MYE. Käytännössä WMM toimii kuitenkin tehokkaammin kuin edeltäjänsä, sillä WMM tutkii usein vain puolet diagonaaleista verrattuna MMY-algoritmiin. Erityisen hyvin WMM:n paremmuuden kilpailijoihinsa nähden pitäisi tulla ilmi silloin, kun PYAn pituus $p \approx m$ ja samalla $m \approx n$, jolloin päästään lähelle lineaarista suoritusaikaa.

WMM varaa esiprosessointivaiheessa ei-vakiotilaisen määrän muistia syötejonojensa lisäksi ainoastaan vektoreille *diagpos* ja *tuhotaan*. Näistä kumpainenkin on pituudeltaan $m+n+1$ eli $\mathcal{O}(n)$. Sen sijaan vektoriin *tuhotaan* tallennettavien hakupolkujen pituuksien karkeaksi ylärajaksi saadaan $\mathcal{O}(P^2)$, jota voidaan ylhäältä rajoittaa edelleen termillä $\mathcal{O}((n-p)^2)$ MMY:n tapaan.

5 Heuristiikat

Tähän mennessä on tarkasteltu yksinomaan algoritmeja, jotka laskevat PYAn tarkan pituuden p ja palauttavat lisäksi jonkin ratkaisuksi kelpaavan PYA-jonon. Tietyissä PYA-ongelman sovelluskohteissa kuten *tiedostojen versionhallinnassa* onkin välttämätöntä, että ongelmalle saavutetaan aina tarkka ratkaisu. Sama pätee *kuvien tiivistämiseen* silloin, kun tiivistyksestä halutaan *häviötön*.

Jos käytettävissä on ennakkotietoa tarkasteltavasta PYA-ongelman instanssista, ongelmaa varten käyttökelpoisen tarkan algoritmin valinta helpottuu melkoisesti. Jos tiedetään vaikkapa, että valtaosa lyhemmän syötevektorin X merkeistä kuuluu syötejonojen PYAan, kannattaa valita algoritmi, joka toimii hyvin pitkälle PYAlle. Tällaisia algoritmeja ovat erityisesti ne, joiden suoritusaikalausekkeessa termi p (PYAn pituus) esiintyy negatiivisena. Vastaavasti, jos tiedetään syöttöaakkoston koon olevan pieni, pysyy kompleksisuudeltaan termistä σ riippuvien algoritmien suoritusaika kohtuullisena. Nämä menetelmät käyttävät laskennassa apuna lähiesiintymätaulukkoa. Silloin puolestaan, kun täsmäysten kokonaismäärän r tiedetään olevan niukka, voidaan käyttää algoritmia, jonka kompleksisuuteen vaikuttaa oleellisesti täsmäysten kokonaismäärä. Syötejonojen ollessa hyvin pitkiä on taas turvauduttava menetelmään, joka vaatii niukalti muistitilaa syötejonojensa lisäksi. Yleisesti ottaen aika- ja tilavaativuuslausekkeet avustavat merkittävästi hyvän menetelmän valitsemisessa silloin, kun on välttämättä löydettävä tarkka ratkaisu PYA-ongelmalle.

Hyvin niukkakin etukäteistietämys syötejonojen X ja Y ominaisuuksista saattaa helpottaa tuntuvasti PYA-ongelman ratkaisemista. Esimerkiksi jos tiedettäisiin ennalta, että syötejonojen $X[1..m]$ ja $Y[1..n]$ pisteparin (i, j) $(1 \leq i \leq m,\ 1 \leq j \leq n)$ muodostama täsmäys kuuluu välttämättä jonojen PYAan, voitaisiin alkuperäinen ongelma osittaa kahdeksi aliongelmaksi ratkaisemalla alkuliiteparin $X[1..i–1]$ ja $Y[1..j–1]$ ja vastaavasti loppuliiteparin $X[i+1..m]$ ja $Y[j+1..n]$ välinen PYA ja yhdistämällä saadut ratkaisut. Mitä keskemmällä syötejonoista muodostettavaa, WFI-algoritmin yhteydessä esitettyä matriisia M piste (i, j) sijaitsee, sitä enemmän voitaisiin täsmäyksiä jättää tutkimatta tarkkaa ratkaisua etsittäessä.

Mutta jollei syötteiden ominaisuuksia ole ennalta aavistettavissa, on tehokkaasti toimivan tarkan algoritmin valitseminen mahdotonta, ja käytettävä menetelmä joudutaan ratkaisemaan miltei arpomalla. Ennemmin tai myöhemmin voidaan joutua toteamaan algoritmin valinnan olleen epäonnistunut joko muistitilan loppumisen tai kiusallisen pitkän suoritusajan takia. Turvallisempi lähestymistapa olisikin tässä tapauksessa lähteä liikkeelle suorittamalla *nopea esiprosessointi*, jonka aikana kerättäisiin tietoja syötteiden ominaisuuksista ennen tarkan algoritmin soveltamista.

Syötejonojen ollessa tuntemattomia voi olla riittävää saada laskettua PYAn pituudelle ensinnä *likiarvo*, ennen kuin halutaan tehdä pidemmälle meneviä johtopäätöksiä syötejonojen ominaisuuksista [BHR98]. Esimerkkinä tästä voisi mainita

vaikkapa bioinformatiikassa tutkittavat *DNA*- ja *proteiiniketjut*. Voidaan vaatia, että ketjujen *samankaltaisuuden asteen*, jonka yhtenä mittana voidaan käyttää niiden PYAn pituutta, on oltava riittävän korkea, jotta ketjuja kannattaa lähteä analysoimaan pidemmälle. Tällöin olisi hyödyllistä saada laskettua jonojen PYAn pituudelle *alaraja*, jonka pituinen PYA vähintään on. Jos alaraja saadaan laskettua tuntuvasti vähemmin resurssein kuin tarkka ratkaisu, ja analysoitavia aineistoja on paljon, voitaisiin säästää paljon työtä jättämällä sellaiset ketjut, joiden PYA ei ole vähintään vaaditun alarajan mittainen, jatkotarkastelujen ulkopuolelle. Vastaavasti silloin, kun syötejonot ovat pitkiä, saattaa PYAn pituuden *ylärajan* laskeminen olla tarpeen, jotta voitaisiin tietää, paljonko *p*:stä riippuville tietorakenteille joudutaan varaamaan muistia. PYAn ala- ja ylärajan laskevat menetelmät kuuluvat ns. *heuristisiin algoritmeihin*.

Yleisesti ottaen heuristiset algoritmit voidaan jakaa kolmeen ryhmään. Ensimmäiseen ryhmään kuuluvat algoritmit, jotka löytävät *aina kelvollisen ratkaisun, mutteivät välttämättä optimaalista* (vrt. syvyyshaku äärellisestä graafista) [Bob00, 179 – 182]. Toisen ryhmän muodostavat algoritmit, jotka ratkaisevat asetetun ongelman *likimain oikein*, mutta tietyllä, yleensä hyvin pienellä todennäköisyydellä, *vastaus voi olla virheellinen*, tai vaihtoehtoisesti algoritmille asetetaan *rajat, joiden sisällä tulos voi poiketa tarkasta* (vrt. alijoukkojen summan approksimointialgoritmi) [CLR93, 980–983]. Kolmanteen ryhmään kuuluvat heuristiset algoritmit toimivat jonkin *strategian ohjaamina*. Parhaassa tapauksessa ratkaisu löytyy hyvin nopeasti, mutta huonossa tapauksessa maalia ei löydetä koskaan (vrt. syvyyshaku äärettömästä graafista) [Bob00, 182]. PYAn pituuden arvioimiseksi kehitetyt ylä- ja alarajan muodostavat algoritmit kuuluvat ensimmäiseen ryhmään. Niiden laskeman alarajan mittainen yhteinen alijono on aina löydettävissä, ja vastaavasti jonojen PYAn pituus ei koskaan ylitä ylärajaa. Sen sijaan laskettujen rajojen etäisyys todellisesta PYAn pituudesta voi vaihdella melkoisesti riippuen syötteistä ja käytetystä algoritmista.

Jotta heuristisen algoritmin käyttäminen tarkan asemesta olisi kannattavaa ja mielekästä, sillä pitää saavuttaa etua tarkkaan menetelmään verrattuna joko suoritusajassa tai muistinkulutuksessa mitattuna. Jotta tämä olisi mahdollista, alun perin asetetun ongelman heuristisen ratkaisun pitää olla *jollain tavoin helpommin laskettavissa kuin tarkan*. PYA-ongelmaa voidaan helpottaa useilla eri tavoilla. Vaihtoehtoisia lähestymistapoja ovat mm. ongelmassa esiintyvien *syötejonojen lyhentäminen* alkuperäisistä, *syöttöaakkoston koon pienentäminen*, tarkasteltavien *täsmäysten lukumäärän rajoittaminen* ja syötejonojen *merkkien järjestyksen höllentäminen*. Kaikkia näitä keinoja alkuperäisen ongelman helpottamiseksi käytetään tässä luvussa esiteltävissä heuristisissa PYA-menetelmissä, jotka laskevat PYAn pituudelle joko ala- tai ylärajan.

Seuraavassa tutustutaan heuristisiin PYA-menetelmiin: ensinnä ylä- ja sittemmin alarajojen laskentaan. Tarkoituksena on esitellä kolme yläraja- ja kuusi alarajaheuristiikkaa [BHR98], joista puolet on tutkimusryhmämme kehittämiä. Myöhemmin luvuissa 6 – 8 näytetään, miten PYA-heuristiikkojen laskemia rajoja

voidaan käyttää suoraan hyväksi myös tarkoissa PYA-menetelmissä [BHV03][Ber05]. Heuristisella esiprosessoinnilla saadaan usein voimakkaasti pienennettyä niiden muistintarvetta sekä nopeutettua niiden käytännön suoritusaikaa. Viimeksi mainittu piirre edellyttää heuristiikoilta nopeaa laskentaa, jotta siihen tarvittava aika lisättynä esiprosessoinnin ansiosta tehostuneen tarkan menetelmän ajoaikaan alittaa suoran tarkan ratkaisun vaatiman suoritusajan.

5.1 PYAn ylärajan laskevat menetelmät

Syötemerkkijonojen PYAn pituuden *ylärajalla* tarkoitetaan *merkkien lukumäärää, jota syötteiden PYAn pituus p ei voi ylittää*. Tässä työssä heuristisesti laskettavan PYAn ylärajan pituudesta käytetään merkintää $p_{ylär}$. Selvästikin lyhemmän syötejonon pituus *m* kelpaa syötteiden PYAn *triviaaliksi ylärajaksi*. Vastaavasti ylärajan on oltava aina vähintään yhtä suuri kuin jonojen *X* ja *Y* todellinen PYAn pituus, sillä muuten ei kyseessä olisi enää *p*:n yläraja. Siten ylärajan sallittua vaihteluväliä kuvaa kaksoisepäyhtälö $p \leq p_{ylär} \leq m$. Huomion arvoista on myös, että ylärajaheuristiikat *eivät ylärajan lisäksi palauta tuloksenaan mitään merkkijonoa* kuten tarkat PYA algoritmit, sillä tälle ei ole mitään suoranaista käyttöä.

Silloin, kun tarkasteltavat syötejonot ovat pitkiä, on jokainen toimenpide tarkan PYA-algoritmin vaatiman muistitilan vähentämiseksi enemmän kuin tervetullut. Laskemalla $p_{ylär}$ ennen tarkan algoritmin soveltamista voidaan kaikkien parametrista *p* riippuvien tietorakenteiden koko rajoittaa etukäteen lasketun arvon $p_{ylär}$ mukaiseksi. Sen sijaan algoritmin suoritusaikaa pystytään ylärajan laskemisella pienentämään yksinomaan silloin, kun se on tarkka eli $p_{ylär} = p$. Seuraavassa luodaan katsaus kolmeen erilaiseen tapaan PYAn pituuden ylärajan määräämiseksi.

5.1.1 Merkkien minimifrekvenssien summan laskeminen (MFS)

Ensimmäisenä ei-triviaalina menetelmänä PYAn pituuden ylärajan laskemiseksi esitellään *Bergrothin, Hakosen ja Raidan* vuonna 1998 ehdottama *merkkien minimifrekvenssien summan laskeminen* [BHR98, 34]. Alkuperäisartikkelissa siitä käytetään lyhennettä *U3*, ja tässä työssä se tunnetaan lyhenteellä *MFS*.

MFS-heuristiikka ei huomioi syötejonojen merkkien järjestystä, vaan laskee sen sijaan pelkästään syöttöaakkoston eri symbolien pareittaiset minimifrekvenssit kummassakin syötevektorissa. Selvästikin syötteiden PYAssa voi esiintyä kutakin yksittäistä aakkoston symbolia enintään yhtä monta kertaa kuin sitä esiintyy siinä syötejonossa, jossa merkki on harvinaisempi. Toisin sanoen, jos merkkiä s_i ($1 \leq i \leq \sigma$) esiintyy *u* ($0 \leq u \leq m$) kappaletta vektorissa *X* ja *v* ($0 \leq v \leq n$) kappaletta vektorissa *Y*, voi jonojen PYAssa *S* esiintyä merkkiä s_i enintään $Min\{u, v\}$ kappaletta. Siten koko

yläraja saadaan määrättyä laskemalla kaikkien merkkien frekvenssit erikseen kummassakin syötevektorissa ja summaamalla niiden minimiarvot:

$$p_{ylär}(\text{MFS}) = \sum_{i=1}^{\sigma} Min\{frekvX(s_i), frekvY(s_i)\},$$

missä $frekvX(s_i)$ tarkoittaa merkin s_i esiintymiskertojen määrää vektorissa X ja $frekvY(s_i)$ vastaavasti samaisen merkin esiintymien lukumäärää vektorissa Y. MFS-heuristiikalla lasketun ylärajan laatu riippuu voimakkaasti siitä, miten paljon merkkien jakaumat syötevektoreissa muistuttavat toisiaan. Jos ne ovat likimain samanlaiset, menetelmä pienentää PYAn ylärajaa ainoastaan hieman triviaalista ylärajasta m. Siten heuristiikan laskeman tuloksen informaatioarvo voi olla vähäinen, jos syötejonojen merkkijakaumat ovat lähellä toisiaan. Toisena ääripäänä voisi mainita tapauksen, jossa syötejonoilla ei ole lainkaan yhteisiä merkkejä. Tällöin heuristiikan palauttama yläraja on sama kuin jonojen todellinen PYAn pituus eli 0. Joka tapauksessa MFS-heuristiikka toimii hyvin silloin, kun merkkijakaumat syötevektoreiden välillä poikkeavat selvästi toisistaan. Seuraavassa esitetään esimerkki MFS-heuristiikan toiminnasta.

Esimerkki 5.1: *MFS-ylärajaheuristiikan toiminta, kun X* = ”BDDBCA”, m = 6, *Y* = ”ABCBDAB”, n = 7, p = 3 ja *PYA* = ”BDB” / ”BDA” / ”BBA” / ”BCA”.

		1	2	3	4	5	6	7
		A	B	C	B	D	A	B
1	B		*		*			*
2	D					*		
3	D					*		
4	B		*		*			*
5	C			*				
6	A	*					*	

Silmukoiden suorituksen jälkeen:

	A	B	C	D
Xfrekvenssi =	**1**	**2**	**1**	2
Yfrekvenssi =	2	3	**1**	**1**

	A	B	C	D
XYfrekvmin =	1	2	1	1

$\Rightarrow p_{ylär} = 5$, *triviaali yläraja* $= m = 6$

Aika- ja tilavaativuus

Ylärajaheuristiikka MFS on toteutukseltaan yksinkertainen, ja siihen kuuluu viisi perättäistä silmukkaa. Ennen varsinaista laskentavaihetta kummankin syötevektorin merkkien frekvenssivektorit nollataan ensimmäisen silmukan sisällä ajassa $O(\sigma)$.

Seuraavissa kahdessa silmukassa selataan kumpikin syötevektoreista ja lasketaan niiden merkkien esiintymiskertojen lukumäärät. Nämä silmukat toimivat yhteensä ajassa $O(n)$, koska Y on aina vähintään yhtä pitkä kuin X oletuksemme mukaan. Neljännessä silmukassa otetaan kunkin merkin minimifrekvenssit talteen vektoriin *XYfrekvmin*, johon tallennetut arvot summataan lopulta yli käytetyn aakkoston viidennessä silmukassa, jonka lopetettua toimintansa ylärajan laskenta on saatu valmiiksi. Neljäs ja viides silmukka vievät yhteensä ajan $O(\sigma)$, joten koko algoritmin suoritusaika on $O(n + \sigma)$.

Algoritmi tarvitsee ei-vakiotilaista muistia ainoastaan syötevektoreilleen, sekä pituudeltaan syöttöaakkoston koosta riippuville vektoreille *Xfrekvenssi*, *Yfrekvenssi* ja *XYfrekvmin*. Siten MFS-heuristiikan tilavaativuus on myös $O(n + \sigma)$. Menetelmän pseudokoodi löytyy liitteestä kohdasta 11.6.1.

5.1.2 Syöttöaakkoston koon supistaminen (SKS)

Toisena ylärajaheuristiikkana esitetään menetelmä, joka perustuu alkuperäisen ongelman *syöttöaakkoston koon supistamiseen* [BHR98, 33–34]. Heuristiikasta käytetään alkuperäisartikkelissa nimeä *U1*, ja tässä työssä sen lyhenteenä on *SKS*.

Edellä esiteltyyn MFS-heuristiikkaan verrattuna SKS tekee huomattavasti enemmän työtä, sillä SKS *säilyttää myös alkuperäisten syötejonojen merkkijärjestyksen*. Kuitenkin ainoastaan osa alkuperäisten syötejonojen X ja Y merkeistä säilytetään ennallaan. SKS-heuristiikassa näyttelee tärkeää osaa parametri σ', joka kuvaa uuden syöttöaakkoston kokoa. Alkuperäisen aakkoston Σ merkit muunnetaan uudelle aakkostolle Σ' kuvaamalla ne *ekvivalenssiluokkiin* $<s_1>$, $<s_2>$, …, $<s_{\sigma'}>$. Mikäli alkuperäisen syöttöaakkoston merkkien ASCII-koodiarvojen oletetaan olevan väliltä $1..\sigma$, saadaan kunkin merkin kuvasymboli uudessa, alkuperäistä pienemmässä aakkostossa laskemalla summa $(i–1) \text{ MOD } \sigma' + 1$, missä $1 \leq i \leq \sigma$ edustaa alkuperäisen symbolin ASCII-koodiarvoa. Täten järjestyksessä *ensimmäiset σ' symbolia kuvautuvat itselleen*. Sen sijaan tämän jälkeiset alkuperäisen aakkoston symbolit kuvautuvat merkiksi, jonka koodiarvo on σ':n monikerran etäisyydellä alkuperäisestä. Jos esimerkiksi $\sigma = 24$ ja $\sigma' = 8$, kuvautuvat ASCII-merkit 1, 9 ja 17 merkiksi 1, merkit 2, 10 ja 18 merkiksi 2 jne. Lopulta ylimpään ekvivalenssiluokkaan <8> kuvautuisivat merkit 8, 16 ja 24. Osamäärä σ'/σ osoittaa, missä suhteessa alkuperäinen aakkosto kutistuu: mitä lähemmäs ykköstä osamäärä tulee, sitä vähemmän alkuperäisen aakkoston merkkejä kuvataan toisiksi, ja osamäärän lähestyessä nollaa valtaosa alkuperäistä merkeistä joudutaan muuntamaan toisiksi. Jotta kuvaus f: Σ ---> Σ' olisi mielekäs, pitää seuraavan ehdon toteutua:

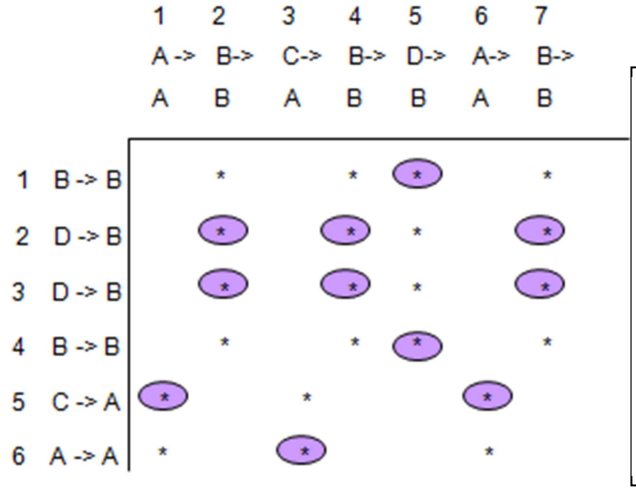
$$2 \leq \sigma' < \sigma.$$

Ellei epäyhtälön oikea puoli pitäisi paikkaansa, kyseessä ei olisi enää kutistava kuvaus. Vastaavasti kuvauksen kohdeaakkoston koon pitää olla vähintään 2. Jos nimittäin kohdeaakkostolle sallittaisiin kooksi 1, tulisi ylärajasta triviaali *m*.

Koska lähdeaakkoston kunkin yksittäisen symbolin kaikki esiintymät kuvautuvat aina yhden määrätyn symbolin esiintymiksi kohdeaakkostolla, säilyvät alkuperäisen ongelman sisältämät täsmäykset myös kutistetun aakkoston mukaisilla syötejonoilla, joten yhtään aikaisempaa täsmäystä ei menetetä kuvauksen ansiosta. Sen sijaan on mahdollista, että uudet syötejonot muodostavat keskenään myös sellaisia täsmäyksiä, joita alkuperäisessä ongelmassa ei esiintynyt. Edellä esitetyssä esimerkissä vaikkapa merkit $x_i = 1$ ja $y_j = 9$ muodostaisivat uusilla syötejonoilla täsmäyksen, jota alun perin ei ollut olemassa. Heuristiikan kuvauksen aiheuttamista ylimääräisistä, ei-todellisista täsmäyksistä käytetään nimitystä *valetäsmäys* tai *huti*. Mitä pienemmäksi osamäärä σ'/σ tulee, sitä pienempi on kohdeaakkosto suhteessa lähdeaakkostoon ja sitä enemmän uuteen ongelmaan muodostuu valetäsmäyksiä.

Seuraavassa esitetään esimerkki, jossa alun perin neljän kokoinen syöttöaakkosto puristetaan kahden kokoiseksi. Tämä tarkoittaa aakkostolla Σ = {”A”, ”B”, ”C”, ”D”} sitä, että merkit ”A” ja ”B” säilyvät ennallaan, mutta jokainen ”C” vaihdetaan ”A”:ksi ja jokainen ”D” puolestaan ”B”:ksi.

Esimerkki 5.2: *SKS-ylärajaheuristiikan toiminta, kun X* = ”BDDBCA”, *m* = 6, *Y* = ”ABCBDAB”, *n* = 7, *p* = 3 ja *PYA* = ”BDB” / ”BDA” / ”BBA” / ”BCA”.



Oheisesta kuvasta käy ilmi, miten alkuperäisten syöte-jonojen merkit vaihtuvat, kun aakkoston kokoa supistetaan 4 ---> 2. Ilman aakkoston kuvausta ongelmassa esiintyisi 11 täsmäystä, mutta tehty kuvaus tuo tullessaan myös 11 *valetäsmäystä*, jotka on merkitty kuvaan **violetein** soikioin. Niiden ansiosta myös PYA pitenee alkuperäisestä arvostaan kolme neljän mittaiseksi, ja sen muodostavat joko merkit ”BBBA” tai ”BBBB”.

Syöttöaakkoston kutistamisen tultua valmiiksi sovelletaan uusille syötejonoille *X’* ja *Y’* nyt tarkkaa PYA-algoritmia. Koska kaikki alkuperäiset täsmäykset säilyvät, ei uusien jonojen PYA ole milloinkaan lyhyempi kuin alkuperäisten jonojen. Sen sijaan valetäsmäykset voivat kerryttää PYAlle *lisäpituutta*, joten tulokseksi saatava arvo on selvästikin PYAn pituuden yläraja. Koska syöttöaakkoston supistaminen saa aikaan *syötteiden merkkikokoelman pienenemisen*, nopeuttaa aakkoston supistaminen

ainoastaan sellaisia PYA-algoritmeja, joiden aika- tai tilakompleksisuuslausekkeessa esiintyy parametri σ. Tällaisia algoritmeja ovat erityisesti *lähiesiintymätaulukkoa* aputietorakenteenaan käyttävät menetelmät, kuten AG1, CPO, RI1, RI2 ja GCL. Samalla kuitenkin *täsmäysten lukumäärä lisääntyy*, joten siihen eli termiin r sidoksissa olevien menetelmien ajoaika voi jopa heikentyä tehdyn aakkoston kutistamisen ansiosta.

Aika- ja tilavaativuudesta

SKS-heuristiikan toiminnollinen ydin on merkkien kuvaus alkuperäiseltä syöttöaakkostolta Σ supistetulle syöttöaakkostolle Σ'. Tämä saadaan tehtyä ajassa $O(n)$, kun kohdeaakkoston koko on tiedossa. Ratkaistaessa tarkkaa PYAa uusille merkkijonoille X' ja Y' jollain kompleksisuudeltaan *aakkoston koosta riippuvalla tarkalla algoritmilla* ei menetelmän teoreettinen kompleksisuus parane, mutta käytännön ajoaika voi supistua osamäärän σ'/σ määräämässä suhteessa.

Jos puolestaan käytettävän tarkan PYA-menetelmän ajoaika *ei suoraan riipu syöttöaakkoston koosta*, kannattaa kuitenkin huomioida, että syöttöaakkoston supistuminen generoi mukanaan jopa runsaastikin valetäsmäyksiä. Siten *täsmäysten lukumäärään* herkästi reagoivat algoritmit kuten HSZ, MUK ja KCR saattaisivat suoriutua tehtävästään jopa hitaammin muunnetuille kuin alkuperäisille syötejonoille. Ylimääräistä muistitilaa SKS varaa ainoastaan kohdevektoreille X' ja Y', joten heuristiikan suorittaminen ei vaikuta tarkan menetelmän tilakompleksisuuteen. Menetelmän karkean tason pseudokoodi on nähtävissä liitteessä kohdassa 11.6.2.

5.1.3 Ongelman osittaminen erillisiksi aliongelmiksi (OEA)

Kolmantena ylärajatekniikkana esitellään heuristiikka, joka osittaa alkuperäisen ongelman *erillisiksi aliongelmiksi* [BHR98, 34]. Heuristiikasta käytetään alkuperäisartikkelissa merkintää *U2*, ja tässä työssä sen lyhenteenä toimii *OEA*.

Siinä missä MFS-heuristiikka jätti syötteiden symbolien järjestyksen täysin huomiotta ja SKS-heuristiikka säilytti sen täydellisesti, OEA asettuu näiden lähestymistapojen välimaastoon säilyttämällä syötejonojen merkkien *osittaisen järjestyksen*. OEA jakaa alkuperäisen syöttöaakkoston Σ k ($k \geq 2$) erilliseen ositteeseen: $\Sigma_1, \Sigma_2, \ldots, \Sigma_k$. Syötejonoista X ja Y muodostetaan samalla k kappaletta alijonoja $X_1, X_2, \ldots X_k$ ja $Y_1, Y_2, \ldots Y_k$. Alijonoihin X_i ja Y_i ($1 \leq i \leq k$) kopioidaan järjestyksessä vektorien X ja Y kaikki ne merkit, jotka kuuluvat syöttöaakkoston ositteeseen Σ_i. Tällä tavoin menettelemällä muodostuu alkuperäisen ongelman pohjalta *k kappaletta itsenäisiä aliongelmia*, joiden sisällä on säilynyt merkkien alkuperäinen järjestys syötevektoreissa. Jos esimerkiksi Σ_1 koostuisi merkeistä ”A”, ”B”, ”C” ja ”D”, sisältäisi vektori X_1 kaikki

X:n sisältämät kyseiset aakkosmerkit alkuperäisessä järjestyksessään, ja vastaavat *Y*:n merkit löytyisivät alkuperäisessä järjestyksessään vektorista Y_1.

Tämän jälkeen ratkaistaan PYA tarkkaa menetelmää käyttämällä kullekin *k* aliongelmalle. Jokaisen aliongelman koko on nyt pienempi kuin alkuperäisen ongelman: sekä syötejonojen pituus että syöttöaakkoston koko ovat alkuperäistä pienemmät, käytännössä näiden lisäksi myös täsmäysten lukumäärä[125]. Mitä tasaisemmin ositteiden koot ovat jakautuneet, sitä nopeammin muodostuneet aliongelmat ratkeavat, joten yleisesti on kannattavaa pyrkiä ainakin likimain tasaiseen ositukseen. Tasaisesti jakautuneiden aliongelmien ratkaisemiseen kuluva yhteenlaskettu aika alittaa käytännössä selvästi alkuperäisen ongelman tarkkaan ratkaisemiseen kuluvan ajan — etenkin, jos alkuperäiset syötejonot ovat pitkiä ja syöttöaakkoston koko on iso. PYAn yläraja saadaan laskemalla yhteen jokaisen aliongelman PYAn pituus. Osittaisen merkkijärjestyksen säilymisen ansiosta mitään alkuperäisen ongelman täsmäystä ei menetetä. Sen sijaan summa usein ylittää todellisen PYAn pituuden, koska eri ositteista saatavat täsmäykset menevät herkästi ristikkäin keskenään, kuten seuraava esimerkki osoittaa. Esimerkin syötejonot ovat samat kuin esimerkissä 3.1.

Esimerkki 5.3: *Ylärajan laskenta OEA:lla, kun X =* ”ABBBDDDACBCBBABCB”, *Y =* ”BAACDCBBACCCDDBAB”, *m = n =* 17, *p =* 9 ja *PYA =* ”ABBACCBAB” / ”BACCBBACB” / ”BACCBBABB”.

Oletetaan, että merkistön ositus tapahtuu seuraavasti:

{”A”, ”B”} ---> Σ_1, {”C”, ”D”} ---> Σ_2

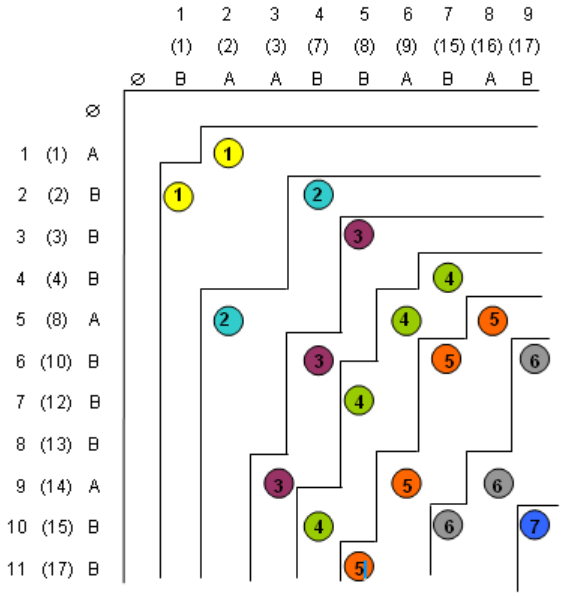
$\Rightarrow$ *Saadaan seuraavanlaiset aliongelmien syötejonot:*

X_1 = ”ABBBABBBABB”, Y_1 = ”BAABBABAB”
X_2 = ”DDDCCC”, Y_2 = ”CDCCCCDD”

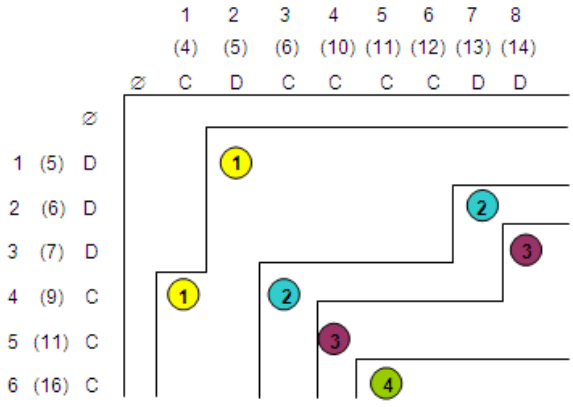
Seuraavassa annetaan aliongelmien dominantit täsmäykset. Kannattaa huomioida, että ensimmäisen aliongelman syötejonoja X_1 ja Y_1 ei ole vaihdettu päittäin, vaikka nyt X_1 onkin jonoa Y_1 pidempi vastoin yleistä oletustamme. Sulkumerkkien sisään on merkitty symbolien sijaintipaikka alkuperäisessä syötevektorissa.

[125] Täsmäysten määrä on sama kuin alkuperäisessä ongelmassa, jos $p = 0$, tai aakkoston osituksen valinta on niin onneton, että kaikki alkuperäisen ongelman täsmäykset (eli täsmäyksen muodostavat symbolit) ovat joutuneet samaan ositteeseen.

Aliongelma 1: X_1 = ”ABBBABBBABB”, Y_1 = ”BAABBABAB”,
p_1 = 7, PYA$_1$ = ”ABBABAB”



Aliongelma 2: X_2 = ”DDDCCC”, Y_2 = ”CDCCCCDD”,
p_2 = 4, PYA$_2$ = ”DCCC”



Yhdistettäessä osaratkaisujen PYAn pituudet saadaan $p_{ylär}$ = 7 + 4 = 11. *Aliongelmien täsmäykset muodostuvat alkuperäisten syötejonojen indeksipareista* (1, 2), (2, 7), (3, 8), (5, 5), (8, 9), (9, 6), (10, 15), (11, 10), (14, 16), (15, 17) *ja* (16, 11). *Näistä voidaan muodostaa seitsemän mittainen jono, jossa kumpikin koordinaateista on kasvavassa järjestyksessä. Kyseiset koordinaatit on alleviivattu. Muut neljä pisteparia ovat jonoon nähden kilpailevia eli ristiriitaisia*: *Y-koordinaatin arvo liian pieni.*

OEA-heuristiikka toimii sitä nopeammin, mitä useampaan ositteeseen ja tasaisemmin syöttöaakkosto jaetaan. Nopeudella on kuitenkin vastapainona ylärajan laadun heikkeneminen: mitä enemmän ositteita, sitä enemmän keskenään ristiriitaisia täsmäyksiä kuuluu eri osaratkaisuihin, jolloin yhteenlasketun ylärajan pituus kasvaa.

Luonnollisen kielen aineistoille voisi olla mielekästä jakaa symbolit kahteen luokkaan: *yleisimmin esiintyviin* ja *harvinaisempiin*. Tällöin yleisimpien merkkien ryhmään kannattaa valita niin monta symbolia, kunnes ne peittävät likimain 50 % kaikista syötejonojen merkeistä. Loput merkit asetetaan harvinaisten merkkien kategoriaan. Tällöin saadaan syötejonojen pituudet likimain puolitettua, ja kun ositteita on vain kaksi, ja on oletettavaa, että PYA koostuu enimmäkseen yleisimmin tekstissä esiintyvistä symboleista, ei ristiriitaisia täsmäyksiä pitäisi normaalisti esiintyä huomattavaa määrää. Liitteen kohdassa 11.6.3 olevassa OEA-heuristiikan pseudokoodissa oletetaan, että ositus tapahtuu kahteen luokkaan siten, että ensimmäiseen sijoitetaan k ja toiseen loput σ–k merkkiä.

Aika- ja tilavaativuudesta

Heuristiikka joutuu ensi töikseen selvittämään, mitkä merkit jaetaan mihinkin ositteisiin. Tätä varten joudutaan suorittamaan kekolajittelu [CLR90, 140–152], jonka kustannus on enintään $O(\sigma \log \sigma)$. Ositusta varten tarvittavien lisävektorien yhteispituudet ovat m ja n, joten niiden täyttäminen ei vaikuta tarkan menetelmän kokonaiskustannukseen, joka kaikilla algoritmeilla riippuu termistä n. Tarkan algoritmin asymptoottiseen suoritusaikaan joudutaan siten lisäämään edellä mainittu aakkoston kekolajittelun kustannus. Muilta osin aikakompleksisuus pysyy ennallaan. Käytännön suoritusaika kuitenkin nopeutuu, jos tarkan algoritmin suoritusaika riippuu syöttöaakkoston koosta σ tai syötteiden pituuksien tulotermistä mn.

Heuristiikan käyttö vaikuttaa tarkan algoritmin asymptoottiseen tilavaativuuteen ainoastaan siten, että joudutaan perustamaan aakkoston koosta riippuvat frekvenssivektorit *Xfrekvenssi*, *Yfrekvenssi* ja *XYfrekvmin*. Tämä koskee luonnollisestikin vain niitä tarkkoja algoritmeja, joille syöttöaakkoston koolla ei ole jo ennestään vaikutusta, ja niiden tilakompleksisuuteen joudutaan lisäämään siten uutena additiivisena terminä $O(\sigma)$.

5.2 Alarajan määräävät menetelmät

Bergroth, Hakonen ja Raita tarkastelivat vuoden 1998 konferenssiartikkelissaan myös *alarajaheuristiikkoja* [BHR98, 35–37]. PYAn pituuden alaraja antaa *takeet* siitä, miten pitkä syötejonojen X ja Y PYA *vähintään* on. Alarajan pituudesta käytetään merkintää p_{alar}. PYAn *triviaaliksi alarajaksi* kelpaa 0, sillä tyhjä merkkijono esiintyy jokaisen syötemerkkijonoparin yhteisenä alijonona.

Alarajaheuristiikat eroavat perustavassa määrin ylärajamenetelmistä siten, että ne pystyvät palauttamaan PYAn vähimmäispituuden lisäksi myös *jonkin merkkijonon, joka kelpaa syötevektorien yhteiseksi alijonoksi*. Tähän rinnastettavaa ominaisuutta ylärajaheuristiikoilla ei sellaisenaan ole, koska MFS kadottaa merkkien sijaintitiedot syötevektoreissa, SKS muodostaa todellisten täsmäysten lisäksi myös valetäsmäyksiä, ja OEAn eri ratkaisupolkujen pisteiden koordinaatit voivat olla (ja lähes aina myös ovat!) kilpailevia keskenään.

Koska alaraja antaa tiedon PYAn vähimmäispituudesta, se saattaa jo yksinään antaa riittävän selkeän kuvan PYAn tarkasta pituudesta. Jos alarajan osuus lyhemmän syötejonon pituudesta ylittää jonkin kriittisen prosenttiosuuden, se saattaa olla riittävä ehto jonojen tarkemmalle tutkimiselle. Tämä tietystikin edellyttää, että alaraja on *luotettava* eli *laadultaan riittävän tiukka*. Mikäli alaraja jää kovin kauas PYAn todellisesta osuudesta p/m, esitetty menettely johtaa ennemmin tai myöhemmin jatkotarkastelun kannalta mielenkiintoisten jonojen hylkäämiseen aiheetta. Toisaalta, luotettavalla ja tiukan rajan tuottavalla ylärajamenetelmällä voitaisiin karsia myöhempien tarkastelujen ulkopuolelle sellaiset syötejonoparit, joille PYAn yläraja on varmuudella liian lyhyt.

Tärkeimpänä alarajan laskennasta saatavana hyötynä on kuitenkin sen mukanaan tuoma mahdollisuus *nopeuttaa tarkan algoritmin käytännön suoritusaikaa*. Käytettävissä oleva tieto PYAn vähimmäispituudesta nimittäin auttaa rajaamaan *osaa* (*dominanteista*) *täsmäyksistä tarkastelujen ulkopuolelle*. Mitä paremman likiarvon PYAn pituudelle alaraja muodostaa, sitä tehokkaammin voidaan kunkin luokan täsmäysten tarkastelua rajoittaa kohti *minimaalisten todistajien joukkoa* Rickin [Ric94] analyysin mukaisesti[126].

Seuraavassa esitellään yhteensä kuusi PYAn alarajaheuristiikkaa. Näistä kaksi ensimmäistä perustuvat syötejonojen merkkien pareittaisten minimifrekvenssien laskemiseen ja kolmas sijainniltaan ”taloudellisimman täsmäyksen” ahnaaseen valintaan. Neljäs ja viides menetelmä käyttävät hyödykseen edellisessä aliluvussa esitettyjä ylärajamenetelmiä SKS ja OEA, ja viimeisen heuristiikan strategia perustuu mahdollisimman pitkien perättäisten täsmäysketjujen ahnaaseen etsintään.

5.2.1 Syötejonojen yleisimmän merkin frekvenssi (YMF)

Primitiivinen heuristiikka palauttaa alarajaksi syötejonojen *X* ja *Y sen merkin pareittaisen minimifrekvenssin, jota jonoissa esiintyy eniten*. Menetelmä esiintyy artikkelissa [BHR98, 35] nimellä *L1*, ja se tunnetaan tässä työssä lyhenteellä *YMF*.

Heuristiikan toimintaperiaate on sangen suoraviivainen. Kutakin syöttöaakkoston symbolia s_i ($1 \le i \le \sigma$) kohti lasketaan sen esiintymiskertojen lukumäärä vektoreissa *X* ja

[126] kts. aliluku 3.4.1.1

Y, ja näistä otetaan talteen minimi muistipaikkaan *XYminfrekv*[*i*]. Kun tämä on suoritettu jokaiselle syöttöaakkoston symbolille, etsitään edellä mainitun vektorin suurin frekvenssi, joka on samalla heuristiikan palauttama alaraja, ja tallennetaan se muuttujaan *MaxFrekvenssi*. Sijaitkoon maksimifrekvenssi paikassa *XYminfrekv*[*k*] ($1 \leq k \leq \sigma$). Tällöin YMF:n muodostaman alarajan mittainen *X*:n ja *Y*:n yhteinen alijono koostuu *MaxFrekvenssi* kappaleesta merkkiä s_k, jotka asetetaan loppuprosessointivaiheessa vektoriin *S*.

Mikäli syötevektorien merkkijakaumat ovat keskenään samankaltaiset ja lisäksi tasaiset, on ilmeistä, että YMF:n laskema alaraja on laadultaan kovin vaatimaton ja jää kauas PYAn todellisesta pituudesta. Sen sijaan syötevektorien jakaumien ollessa hyvin erilaiset ja vieläpä vinosti jakautuneet, voidaan menetelmällä saada nopeasti verrattain kelvollinen arvio PYAn pituudesta.

Tarkastellaan vielä esimerkkiä 5.1. Jos YMF-heuristiikkaa sovellettaisiin sen mukaisille syötejonoille, kävisi ilmi, että yleisin molemmissa syötevektoreissa esiintyvä aakkoston symboli on ”B”, joka esiintyy kummassakin vektorissa *vähintään kahdesti* – *X*:ssä kaksi ja *Y*:ssä kolme kertaa. Siten YMF palauttaisi alarajaksi 2 ja yhteiseksi alijonoksi ”BB”, kun PYAn pituus esimerkissä olisi 3.

Liitteen kohdassa 11.7.1 esitetään menetelmän pseudokoodi. YMF eroaa ylärajamenetelmästä MFS ainoastaan siten, että MFS kerää ylärajan pituuteen jokaisen symbolin pareittaisen minimifrekvenssin, kun taas YMF etsii alarajaa laskiessaan niistä vain maksimin. Lisäksi YMF palauttaa löytämänsä yhteisen alijonon, jonka kerääminen vie ajan $O(p_{alar})$. Koska YMF tarvitsee käyttöönsä täsmälleen samat tietorakenteet kuin ylärajaheuristiikka MFS ja tämän lisäksi vain $O(p_{alar})$-mittaisen tulosvektorin, ovat molempien menetelmien aika- ja tilakompleksisuudet keskenään samat, eli $O(n+\sigma)$.

5.2.2 Chinin ja Poonin heuristinen algoritmi (CPH)

Aliluvun 3.1.6 tarkan PYA-algoritmin kehittäjät *Chin* ja *Poon* ovat vuonna 1994 esittäneet myös heuristisen PYAn alarajamenetelmän [Chi94; BHR98, 35]. Tekijät kutsuvat sitä nimellä *CNT*, ja tässä työssä se kantaa nimeä *CPH*.

Edellä esitellyn YMF-heuristiikan tavoin myös CPH turvautuu syötejonojen merkkien frekvenssien tarkasteluun. Aluksi lasketaan syöttöaakkoston eri symbolien esiintymiskertojen lukumäärät kummassakin syötevektorissa. Tästä eteenpäin tiet kuitenkin eroavat, sillä CPH ei enää laskekaan erilliseen vektoriin merkkikohtaisia pareittaisia minimifrekvenssejä, vaan käynnistää syötevektorien selaamisen alusta loppua kohti. Kumpaakin vektoria varten asetetaan kursorit *i* ja *j*, jotka osoittavat aluksi syötevektoreiden *X* ja *Y* indeksiin 1. Jos molempien vektorien ensimmäiset symbolit ovat samat, kasvatetaan alarajan pituutta yhdellä ja siirretään kumpaakin kursoria yhdellä eteenpäin. Jos taas merkit eivät täsmää, tutkitaan seuraavaksi, miten monta kertaa *kumpikin tarkastelluista merkeistä* – x_1 ja y_1 – *vähintään esiintyy molempien*

vektorien loppuosassa kursorien nykypositiot mukaan lukien. Se, kumpaa esiintyy vähemmän, tulkitaan harvinaisemmaksi merkiksi, ja silloin siinä vektorissa, jossa kohdattiin harvinaisempi merkki, siirretään kursoria yhdellä eteenpäin ja pienennetään juuri tarkastellun merkin frekvenssiä vektorin loppuosassa yhdellä. Toisen vektorin kursori jätetään ennalleen. Jos merkkejä esiintyy molempien vektorien loppuosassa yhtä monta, voidaan sopia, että edetään vektorissa *Y* yksi positio eteenpäin, koska *Y* voi sisältää merkkejä enemmän kuin *X*. Tällä tavoin jatketaan, kunnes jompikumpi syötevektoreista loppuu kesken[127]. Seuraavassa annetaan pieni esimerkki heuristiikan etenemisstrategiasta. CPH-heuristiikasta on esitelty myös *pelkistetympi versio*, johon palataan lyhyesti myöhemmin esimerkissä 5.5.

Esimerkki 5.4: *Chinin ja Poonin heuristiikan suorituksen eteneminen syötejonoille* X = ”BDDBCA” *ja* Y = ”ABCBDAB”.

1 2 3 4 5 6 1 2 3 4 5 6 7

Vertailtava x_i:n ja y_i:n minimifr.

merkkipari	*X*[*i..m*] ja *Y*[*j..n*]	*X-kursori*	*Y-kursori*	*Alarajan pituus*
(1, 1) (B, A)	*2, 1*	*1*	*1 -> 2*	*0*
(1, 2) (B, B)	*2, 2*	*1 -> 2*	*2 -> 3*	*1*
(2, 3) (D, C)	*1, 1*	*2*	*3 -> 4*	*1*
(2, 4) (D, B)	*1, 1*	*2*	*4 -> 5*	*1*
(2, 5) (D, D)	*1, 1*	*2 -> 3*	*5 -> 6*	*2*
(3, 6) (D, A)	*0, 1*	*3 -> 4*	*6*	*2*
(4, 6) (B, A)	*1, 1*	*4*	*6 -> 7*	*2*
(4, 7) (B, B)	*1, 1*	*4 -> 5*	*7 -> 8*	*3*
(5, 8) (C, ∅)	*0, 0*	*suoritus päättyy*		*3*

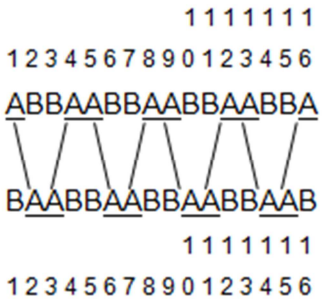
Tarkasteltaessa yllä olevaa esimerkkiä havaitaan, että paikassa (1, 1) olevat symbolit ”B” ja ”A” eivät täsmää keskenään. Niinpä joudutaan laskemaan, kumpi näistä kahdesta merkistä esiintyy kummankin syötevektorin loppuosassa useammin aloituspositio (1, 1) mukaan lukien. Koska ”B”:tä esiintyy *X*:ssä kaksi ja *Y*:ssä kolme kertaa, on ”B”:n minimifrekvenssi siten 2. Sen sijaan ”A”:ta esiintyy *X*:ssä vain kerran ja *Y*:ssä kahdesti, joten ”A”:n minimifrekvenssi on 1. Täten ”A” tulkitaan paikassa (1, 1) esiintyneistä symboleista harvinaisemmaksi, joten siirrytään siinä vektorissa, josta ”A” luettiin, yksi positio eteenpäin. Siten *X*-kursori jää paikoilleen ja *Y*-kursoria siirretään yksi askel eteenpäin, jolloin seuraava vertailtava merkkipari löytyy paikasta (1, 2). Siinä esiintyy merkkien ”B” muodostama täsmäys, joten molempia kursoreita siirretään eteenpäin, ja samalla alarajan pituus kasvaa ykköseen. Seuraavaksi tutkittavassa paikassa (2, 3) ei esiinny täsmäystä, koska ”D” ≠ ”C”, mutta kummankin

[127] Olisi tietystikin myös mahdollista siirtyä eteenpäin aina siinä vektorissa, jossa merkkejä on *tarkasteluhetkellä* jäljellä enemmän, jos harvinaisempi kohdatuista merkeistä esiintyy kummassakin vektorissa vielä yhtä monta kertaa. Voi siis hyvinkin käydä niin, että *Y*:ssä on vähemmän merkkejä jäljellä kuin *X*:stä tarkastelun edetessä, vaikka alun perin *Y* oli vähintään yhtä pitkä kuin *X*.

merkin minimifrekvenssi syötevektorien loppuosassa on sama. Tällöin siirretään sopimuksen mukaan *Y*-kursoria yhdellä eteenpäin, ja samalla ”C”:n frekvenssi *Y*:n loppuosassa pienenee yhdellä. Tällä tavoin jatketaan, kunnes tullaan ulos jommastakummasta vektorista, kuten esimerkissä tapahtuu täsmäyksen (4, 7) käsittelyn jälkeen. Esimerkissä CPH löytää kolmen mittaisen alarajan, joka jää YMF:n laskeman alarajan 2 ja todellisen PYAn pituuden 4 väliin.

Koska CPH siirtää kummankin syötevektorin kursoria eteenpäin joka kerta kun kohdataan täsmäys, on ilmeistä, että menetelmä löytää syötejonoille laillisen yhteisen alijonon, jonka pituus siten kelpaa PYAn pituuden alarajaksi. Samoin kuin YMF:n, niin myös CPH:lla lasketun alarajan laatu saattaa jäädä kuitenkin heikoksi. Näin kävisi erityisesti silloin, jos CPH:sta käytettäisiin alkuperäisartikkelissa [Chi94] esitettyä *yksinkertaistettua versiota, joka ei päivitä merkkien frekvenssejä laskennan käynnistyttyä*, vaan tekee päätökset suoraan koko syötejonojen merkkien *alkuperäisfrekvensseihin* perustuen. Jos ne ovat samat, tulkitaan aakkosissa jälkimmäisenä oleva harvinaisemmaksi merkiksi. Menetelmän kehittäjien seuraava esimerkki on osoitus tästä [Chi94, 297].

Esimerkki 5.5: *Chinin ja Poonin pelkistetyn heuristiikan suorituksen eteneminen syötejonoille* X = ”ABBAABBAABBAABBA” *ja* Y = ”BAABBAABBAABBAAB”, p = 14, PYA = ”ABBAABBAABBAAB”, *kun merkkien frekvenssejä ei päivitetä suorituksen aikana: frekv*[”A”] = *frekv*[”B”] = 8.



Kummassakin syötevektorissa on kahdeksan ”A”:ta ja ”B”:tä. Koska ensimmäiset merkit eivät täsmää, siirretään Y-kursoria eteenpäin paikkaan 2. Paikasta (*1, 2*) *löydetään ”A”-symbolien täsmäys, jonka johdosta alarajan pituus kasvaa nollasta yhteen. Sieltä edetään paikkaan* (*2, 3*), *jossa X:stä luetaan ”B” ja Y:stä ”A”. Koska ”B” tulkitaan jatkuvasti harvinaisemmaksi symboliksi, joudutaan X:ssä ohittamaan kaksi merkkiä, kunnes paikasta* (*4, 3*) *löydetään seuraava täsmäys. Paikassa* (*5, 4*) *epäonnistutaan jälleen, mikä johtaa kahden ”B”:n ohittamiseen vektorista Y. Koska ”B” säilyy harvinaisempana merkkinä, ei CPH:n yksinkertaistettu versio löydä esimerkissä*

niiden muodostamia täsmäyksiä, vaan heuristiikka löytää vain kahdeksan mittaisen, pelkistä ”A”-symboleista koostuvan alijonon.

Liitesivuilla kohdassa 11.7.2 esitetään frekvenssejä suorituksen aikana päivittävän CPH-heuristiikan pseudokoodilistaus. Esimerkissä 5.5 tarkasteltua yksinkertaistettua versiota ei jatkossa enää käsitellä.

Aika- ja tilavaativuudesta

CPH:n pseudokoodista havaitaan, että sen kolme ensimmäistä silmukkaa ovat identtiset YMF-heuristiikan kanssa, ja ne vievät yhteensä ajan $O(n + \sigma)$. Viimeinen silmukka käy puolestaan läpi syötevektoreiden merkkejä, kunnes jompikumpi loppuu kesken. Tämän silmukan suoritusaika on $O(n)$, joten se ei lisää asymptoottista kokonaistyömäärää. CPH toimii siis ajassa $O(n + \sigma)$ kuten YMF-alarajaheuristiikkakin.

Vaikka CPH:n ja YMF:n asymptoottiset suoritusajat ovat samat, käytännössä YMF on nopeampi, sillä siinä tehdään ainoastaan yksi syötevektoreiden selaus. CPH:ssa ne joudutaan käymään läpi kahdesti: ensinnä merkkien frekvenssien laskentaa ja myöhemmin täsmäyksien etsimistä varten. Tällä ei kuitenkaan ole liiemmin merkitystä, koska syötevektorien selaamiskustannus on lineaarinen. Kannattaa huomioida, että CPH ei milloinkaan tuota huonompaa alarajaa kuin YMF, sillä kursorien siirtelysäännöt takaavat, että ainakin yleisimmän merkin kaikki täsmäykset tulevat mukaan alarajaan. Syötteiden merkkijärjestyksen ollessa suotuisa on odotettavissa, että CPH:n tuottama alaraja on jopa huomattavastikin YMF:n laskemaa tulosta tarkempi.

CPH-heuristiikka tarvitsee syötejonojensa lisäksi tilaa ainoastaan kahdelle frekvenssivektorille, joiden pituus riippuu syöttöaakkoston koosta $O(\sigma)$, sekä tulosvektorille *S*, jonka pituus on $O(p)$. Siten CPH:n kokonaistilavaativuus on sama kuin sen asymptoottinen suoritusaika eli $O(n + \sigma)$.

5.2.3 PYAMAX-heuristiikka (PMX)

Kolmantena alarajamenetelmänä esiteltävä PYAMAX-heuristiikka näki päivänvalon 1990-luvun puolivälissä kahdella eri taholla toisistaan riippumatta. Ensimmäisen kerran kyseisen algoritmin perusteet kuvataan lyhyesti *Fraserin* vuonna 1995 ilmestyneessä väitöskirjassa [Fra95, 81–82], ja tätä melkoisesti laajempi samaa heuristiikkaa koskeva katsaus löytyy *Johtelan* ym. vuotta myöhemmin ilmestyneestä tutkimuksesta [Joh96]. Menetelmää on tarkasteltu vertailumielessä lisäksi myöhemmin vuonna 1998 ilmestyneessä tutkimusryhmämme konferenssiartikkelissa [BHR98, 35]. Näissä julkaisuissa ei kuitenkaan ole esitettynä heuristiikan yksityiskohtaista pseudokoodia, joka sisällytetään tämän työn liiteosan kohtaan 11.7.3. Fraser käyttää heuristiikasta nimitystä *BestNext*, kun taas Johtelan artikkelissa vastaavan heuristiikan nimityksenä on

path. Tutkimusryhmässämme heuristiikalle on vakiintunut nimitykseksi *PYAMAX*, josta tässä työssä käytetään lyhennettä *PMX*.

PYAMAXin toiminta-ajatus muistuttaa melkoisesti *Chinin ja Poonin tarkan algoritmin* eli CPO:n käyttämää laskentatapaa. Heuristiikka etsii täsmäyksiä korkeuskäyrä kerrallaan. Siinä missä CPO kirjaa jokaista korkeuskäyrää k ($1 \leq k \leq p$) kohti kaikki sille kuuluvat dominantit täsmäykset, PMX valitsee niistä kuitenkin ainoastaan *yhden*. Täsmäyksen valinta tapahtuu ahnaasti: aina *paikallisesti taloudellisin dominantti täsmäys* huomioidaan ja kirjataan kuuluvaksi heuristiseen PYA-jonoon. Täsmäyksen taloudellisuudelle voidaan esittää useita määritelmiä (vrt. [Joh96, 132]). Niistä käsitteellisesti yksinkertaisin lienee Fraserin esittämä kriteeri [Fra85, 81]. Sen mukaan tarkasteltavan korkeuskäyrän k täsmäyksistä (i, j) ($1 \leq i \leq m$, $1 \leq j \leq n$) valitaan se, joka maksimoi täsmäyskohdan jälkeisten syötevektorien loppuliitteiden yhteisen minimipituuden $Min\{m–i,\ n–j\}$. Käyttämällä kyseistä valintakriteeriä tulee valituksi aina se luokan k täsmäys, jonka kautta voidaan teoriassa muodostaa pisin mahdollinen yhteinen alijono, sillä PYA voi pidentyä k:sta enintään $Min\{m–i,\ n–j\}$:n verran. Jos kumpikin syötejonoista on yhtä pitkä, pyrkii *PMX* valitsemaan siten luokan k sellaisen dominantin täsmäyksen, joka sijaitsee lähimpänä syötejonoista muodostuvan matriisin päälävistäjää. Jos puolestaan $m < n$, tulee valituksi sellainen dominantti täsmäys, joka sijaitsee mahdollisimman lähellä diagonaalien 0 ja Δ rajoittamaa vyötä ($\Delta = n–m$).

Lähdettäessä etsimään luokan k taloudellisinta dominanttia täsmäystä etsintä käynnistetään syötejonoista X ja Y muodostetussa matriisissa heuristiikan valitseman k–1 -täsmäyksen oikealta alapuolelta[128]. Sijaitkoon mainittu k–1 -täsmäys pisteessä (i, j). Ensiksi etsitään riviltä i+1 lähtien ensimmäinen X:n symboli, joka täsmää Y:n merkin kanssa sarakkeen j jälkeen, ja valitaan ehdon täyttävistä täsmäyksistä *vasemmanpuoleisin*. Olkoot täsmäyksen koordinaatit (u, v) ($i < u \leq m$, $j < v \leq n$). Tämän jälkeen tutkitaan puolestaan Y:n symboleja ja etsitään sarakkeelta j+1 lähtien niistä ensimmäinen, joka täsmää X:n merkin kanssa rivin i jälkeen, ja valitaan ehdon täyttävistä täsmäyksistä *ylin*. Sovitaan, että kyseinen täsmäys sijaitsee paikassa (w, z) ($i < w \leq m$, $j < z \leq n$). Löydetyt pisteet (u, v) ja (w, z) sijaitsevat nyt suorakulmion muotoisen alueen oikeassa ylä- ja vasemmassa alakulmassa[129]. Tältä alueelta haetaan nyt se dominantti k-täsmäys, joka maksimoi syötejonojen lyhemmän loppuliitteen pituuden. Jos useammalla kuin yhdellä k-täsmäyksellä on sama maksimaalinen loppuliitteiden minimipituus, valitaan näistä se täsmäys, joilla toinen dimensioista on pidempi. Jos pidemmätkin loppuliitteiden dimensiot ovat täsmäyksillä identtiset, voidaan sopia, että valitaan tällöin ehdokkaista ylimpänä sijaitseva. Heuristisen PYAn pituus kasvaa yhdellä, ja valittu täsmäys viedään heuristisen PYAn ratkaisun k. merkiksi. Tämän jälkeen aletaan etsiä luokkaan k+1 kuuluvia täsmäyksiä valitun

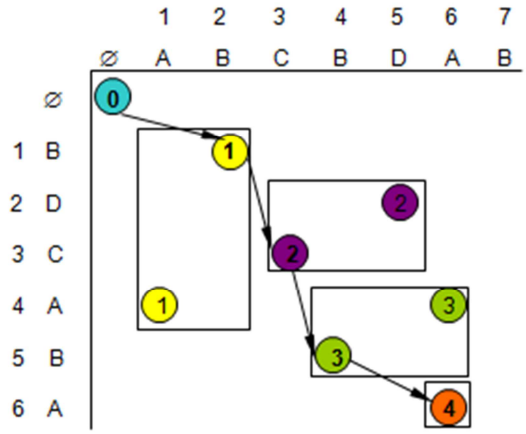
[128] PMX olettaa luokan 0 pseudotäsmäyksen sijaitsevan pisteessä (0, 0).

[129] Jos pisteet (u, v) ja (w, z) ovat samat, suorakulmio puristuu yhdeksi pisteeksi, joka on samalla etsitty taloudellisin k-täsmäys. Ellei vaatimukset täyttävää pistettä (u, v) löydy laisinkaan, ei tarkasteltavalla alueella sijaitse enää yhtään täsmäystä, joten heuristiikan suoritus voidaan lopettaa.

k-täsmäyksen oikealta alapuolelta. Tällä tavoin jatketaan, kunnes johonkin luokkaan ei enää löydetä lainkaan täsmäyksiä. Tällöin algoritmin suoritus päättyy, ja heuristisen PYAn pituus ja ratkaisujono löytyvät muuttujista p_{heur} ja S. Heuristiikan löytämä ratkaisu on selvästikin syötejonojen alijono, sillä kummankin syötejonon kursoria siirretään aina eteenpäin uuden täsmäyksen tultua valituksi siihen.

Aputietorakenteena PMX-heuristiikka käyttää sekä vektorin X että Y esiintymälistoja, jotka on rakennettu yhdeksi ketjuksi, jonka pituus on $m + n + 1$. Alkusolmu toimii samalla listojen pysäytysalkiona. Esiintymälistoilla taataan, ettei sen kummemmin vektoria X kuin Y jouduta selaamaan läpi esiprosessoinnin jälkeen useampaan otteeseen kuin kerran, mikä mahdollistaa algoritmin nopean suorituksen. Seuraavassa näytetään pieni esimerkki PMX-heuristiikan toiminnasta.

Esimerkki 5.6: *PYAMAX-heuristiikan toiminta: X* = ”BDCABA”, *Y* = ”ABCBDAB”.



PYAMAX-heuristiikka valitsee 1-täsmäyksen (1, 2), sillä sen perään mahtuu vielä viiden mittainen yhteinen alijono, kun taas valinnan (4, 1) jatkeeksi mahtuisi enää vain kaksi *X*:n merkkiä. Analogisesta syystä 2-täsmäys (3, 3) on parempi valinta kuin tätä ylempi (2, 5). Valinta 3-täsmäysten (4, 6) ja (5, 4) välillä kallistuu jälkimmäisen hyväksi, sillä minimien ollessa samat valitaan se täsmäys, jolle toinen dimensio on pidempi (3 > 2). Luokan 4 täsmäykselle on tarjolla enää yksi rivi. PMX löytää tässä esimerkissä jopa tarkan PYAn pituuden 4 ja jonon ”BCBA”.

Ongelman instansseissa, joissa jokin PYAn ratkaisuista sijaitsee lähellä matriisin päälävistäjän ja maalilävistäjän välistä aluetta, PMX tuottaa yleensä kirkkaasti edellä kuvattujen kahden alarajamenetelmän laskemaa alarajaa laadukkaamman ja luotettavamman arvion PYAn pituudelle. PMX:n käytännön suoritusaika on lisäksi niin paljon tarkkoja menetelmiä nopeampi, että se on osoittautunut erittäin varteenotettavaksi vaihtoehdoksi tarkan PYAn määrääville algoritmeille, mikäli tuloksena saatavan PYAn pituuden sallitaan olevan tiettyjen rajojen puitteissa virheellinen. Mikäli PYA on jakautunut syötteistä muodostettavan matriisin päälävistäjän tuntumaan, menetelmä on käytännön testeissä löytänyt huonoimmillaankin – PYAn prosenttiosuuden *X*:n pituudesta ollessa 50 – puolet PYAn pituudesta [BHR98, 38–39]. Kaikesta huolimatta PMX-heuristiikallakin on kuitenkin heikko kohtansa: se on nimittäin herkkä syötejonojen PYAn *sijainnille matriisissa*. Tähän ominaisuuteen palataan uudelleen tarkemmin luvuissa 6 ja 7.

PMX-heuristiikan aika- ja tilavaativuudesta

PMX-heuristiikan esiprosessointivaiheen kustannus on $O(n + \sigma)$, sillä se tarvitsee käyttöönsä kaikille syöttöaakkoston symboleille molempien syötevektorien suuntaiset esiintymälistat, jotka rakennetaan selaamalla kumpikin syötevektori alusta loppuun. Laskentavaiheessa kunkin korkeuskäyrän k pisteitä generoidaan ainoastaan yhden edellisen korkeuskäyrän k–1 pisteen perusteella. Koska jokainen syöttöaakkoston merkeistä voi tällöin muodostaa enintään yhden dominantin täsmäyksen luokkaa kohti, voi yhdessä luokassa esiintyä enintään σ dominanttia täsmäystä, joista vain yksi valitaan ja muut joutuvat unohduksiin. Kaiken kaikkiaan PMX:ssä tutkitaan siten enintään σp_{alar} kappaletta dominantteja täsmäyksiä. Silmukoiden suoritusten aikana vektoreissa X ja Y ei peräännytä kertaakaan, vaan ne selataan kertaalleen läpi. Heuristiikassa ei ole mitään erillistä loppuprosessointia, vaan heuristisen PYAn pituus ja sen mittainen yhteinen alijono ovat jo laskettuina, kun viimeinenkin löytynyt korkeuskäyrä on tutkittu. Siten algoritmin kokonaissuoritusaika on $O(n + \sigma + \sigma p_{alar})$. Tämän perusteella heuristiikan pitäisi toimia ainakin pienillä syöttöaakkostoilla erittäin nopeasti.

Heuristiikka vaatii ei-vakiomääräistä muistitilaa ainoastaan syötejonoille ja esiintymälistoille. Siten PMX-heuristiikan tilakompleksisuus on $O(n + \sigma)$ eli samaa suuruusluokkaa YMF:n ja CPH:n kanssa. Koska tilakompleksisuudessa esiintyy vain ensimmäisen asteen termejä, PMX:n voisi olettaa olevan varsin käyttökelpoinen heuristiikka, vaikka syötejonot olisivat hyvinkin pitkiä. Muistitilan loppuminen kesken ei siis aiheudu ongelmaksi normaalin kokoisilla PYA-ongelman instansseilla.

5.2.4 Alarajan laskeminen ylärajamenetelmän avulla

Edellä esitellyt kolme alarajaheuristiikkaa – YMF, CPH ja PMX – ovat siinä mielessä puhtaasti heuristisia PYA-algoritmeja, että ne eivät käytä laskennassa hyväkseen mitään tarkkaa PYA-menetelmää. Tarkastellessamme sen sijaan aliluvussa 5.1 ylärajaheuristiikkoja totesimme, että SKS- ja OEA-heuristiikat tarvitsevat avukseen tarkkaa algoritmia. Kyseiset heuristiikat helpottivat alkuperäistä PYA-ongelmaa pienentämällä syöttöaakkoston kokoa (SKS, OEA) sekä lisäksi osaongelmien syötejonojen pituuksia (OEA). Näitä kahta ylärajamenetelmää voidaan verraten helposti jalostaa siten, että ylärajan avulla voidaan muodostaa haluttaessa myös alaraja.

Ylärajan laskevat menetelmät eivät palauta ratkaisunaan mitään jonoa, sillä sille ei sinänsä ole mitään kovin hyödyllistä tulkintaa. Kuitenkin esimerkiksi SKS:n voisi hyvin määrätä tulostamaan ylärajan lisäksi myös (X, Y)-indeksiparit, joista löytyivät menetelmän laskeman ylärajan mittaiseen jonoon kuuluvat täsmäykset. Ongelmana on kuitenkin se, että jonon täsmäykset ovat ainoastaan ani harvoin[130] kaikki laillisia, eli

[130] Kaikki täsmäykset voivat olla todellisia ainoastaan silloin kun $p = p_{ylär}$.

mukaan mahtuu mahdollisesti useitakin aakkoston supistamisesta johtuvia *valetäsmäyksiä*. Vastaavasti OEA:n eri osaongelmien ratkaisujen yhdistäminen ei onnistu aivan suoraviivaisesti sen tähden, että niiden täsmäysten indeksiparit voivat olla keskenään *kilpailevia*[131]. Kumpaakin ylärajaheuristiikkaa voidaan kuitenkin täydentää sopivalla loppuprosessoinnilla, joka suodattaa ylärajaheuristiikkojen löytämistä ”ratkaisujonoista” alkuperäisten syötejonojen kelvollisen yhteisen alijonon. Seuraavissa kahdessa aliluvussa tutustutaan, miten SKS- ja OEA-ylärajaheuristiikkojen laskentaa voidaan hyödyntää alarajan määräämiseksi. Kumpikin näistä on tutkimusryhmäni kehittämä [BHR98].

5.2.4.1 Syöttöaakkostoltaan supistetun ongelman täsmäysten tarkastaminen (SST)

Kuten jo edellä ehdittiin todeta, SST-alarajaheuristiikan juuret ulottuvat suoraan aliluvussa 5.1.2 esitellyn SKS-ylärajaheuristiikan perusteisiin. Käytettäessä SKS-heuristiikkaa ylärajan laskentaan supistetaan syöttöaakkosto kooltaan murto-osaan alkuperäisestä. Osamäärä σ'/σ kuvaa uuden ja alkuperäisen syöttöaakkoston koon suhdetta. Jos osamäärä on esimerkiksi ¼, vastaa jokainen kohdeaakkoston merkki neljää alkuperäisen aakkoston merkkiä.

Kun aakkoston uudelleenkuvauksen ansiosta muuttuneiden syötejonojen PYA ratkaistaan tarkalla PYA-algoritmilla, saadaan PYAlle selvästikin *yläraja*, sillä merkistön kuvaus säilyttää kaikki alkuperäisen ongelman täsmäykset, mutta se tuo mitä ilmeisimmin mukanaan myös uusia, ns. valetäsmäyksiä. Ottamalla talteen SKS:n laskeman ylärajan pituus sekä palauttamalla ylärajan mukaisen jonon indeksit kummassakin syötejonossa voidaan siitä puhdistaa pois kaikki valetäsmäykset. Tällä tavoin saadaan muodostettua PYAlle *alaraja*. Heuristiikasta, joka muodostaa syötejonojensa PYAn alarajan hyödyntämällä SKS-ylärajaheuristiikan laskemia tuloksia, käytetään tässä työssä lyhennettä *SST*. Alkuperäisartikkelissa [BHR98] heuristiikka tunnetaan nimellä *L4* ja sen *vahvistettu versio* nimellä *L4'*.

SST-heuristiikka edellyttää siis esiprosessointina SKS-heuristiikan suorittamista. SKS palauttaa tuloksenaan PYAn ylärajan sekä lisäksi *X*- ja *Y*-indeksit, joista ylärajan mittainen yhteinen alijono löytyi, kun alkuperäisten syötejonojen merkit oli ensinnä kuvattu supistetulle aakkostolle Σ'. Kukin SKS-heuristiikan palauttama indeksipari tutkitaan ja selvitetään, onko siinä täsmäystä myös alkuperäisillä eikä ainoastaan muunnetuilla syötejonoilla. Jos tutkittu täsmäys oli aito, kasvatetaan alarajan pituutta yhdellä ja viedään täsmäyksen muodostanut merkki alarajaa vastaavaan jonoon. Jos taas kyseessä oli merkistön kuvauksesta johtunut valetäsmäys, se vain yksinkertaisesti ohitetaan.

[131] kts. esimerkki 5.3

Tarkastellaan uudelleen esimerkkiä 5.2. Siinä SKS-heuristiikka palauttaa neljän mittaisen ylärajan, joksi saadaan joko ”BBBB” indekseistä (1, 2), (2, 4), (3, 5) ja (4, 7) tai ”BBBA” indekseistä (1, 2), (2, 4), (3, 5) ja (5, 6) edellyttäen, että käytettävä tarkka PYA-menetelmä huomioi ainoastaan dominantit täsmäykset[132]. Ensin mainitusta jonosta täsmäykset (1, 2), (3, 5) ja (4, 7) ovat aitoja ja vastaavat kummassakin alkuperäisessä syötejonossa merkkejä ”BDB”, kun sitä vastoin (2, 4) on valetäsmäys, sillä x_2 = ”D” ≠ y_4 = ”B”. Jälkimmäisestä jonosta sen sijaan (2, 4):n lisäksi myös paikassa (5, 6) on valetäsmäys, sillä x_5 = ”C” ≠ y_6 = ”A”. Siten SST-heuristiikka palauttaisi esimerkissä PYAn alarajaksi joko 3 tai 2 riippuen siitä, mitkä täsmäykset SKS-heuristiikan soveltama tarkka algoritmi valitsee. Jos heuristiikka sattuu palauttamaan jonon ”BBBB”, saadaan alarajaksi jopa tarkka PYAn pituus 3. Esimerkin perusteella on ilmeistä, että mitä enemmän SKS valitsee ylärajaansa valetäsmäyksiä, sitä ilmeisimmin SST ei saavuta tarkkaa PYAn pituutta vaan ainoastaan alarajan sille.

SST-heuristiikan vahvistaminen

SST-heuristiikan määräämää alarajaa voidaan yrittää myös jälkikäteen vahvistaa. Tarkastellaan SKS-heuristiikan valitsemia täsmäysketjuja, joiden ensimmäinen ja viimeinen täsmäys[133] ovat aitoja mutta keskimmäiset huteja. Olkoon ylemmän aidon täsmäyksen sijaintipaikka (*i*, *j*) ja alemman (*u*, *v*). Koska SKS ratkaisee PYAn muunnetuille syötemerkkijonoille, ei aitojen täsmäysten välissä voi sijaita useampia todellisia täsmäyksiä kuin niiden väliin jäävien valetäsmäysten määrän verran. Alueelta *X*[*i*+1..*u*–1], *Y*[*j*+1..*v*+1] voitaisiin nyt etsiä tarkan algoritmin avulla hutien tilalle *kelvollisia täsmäyksiä*, jotka löytyessään pidentäisivät alarajaa ja parantaisivat siten sen laatua. Koska tällaisten alueiden koko on verrattain pieni, ei tarkan algoritmin kutsuminen näille alueille vaadi yleensä suurta lisätyömäärää. Patologisessa tapauksessa on tosin tietysti mahdollista, että kaikki SKS:n löytämät täsmäykset olivat huteja. Tällöin alarajan 0 vahvistamiseksi ehdotetulla tavalla joudutaan loppujen lopuksi kutsumaan tarkkaa algoritmia koko syötejonoille! Liitteenä kohdassa 11.7.4 esitettävässä SST-heuristiikan pseudokoodissa on erotettuna kahdella tähdellä (**) ne rivit, jotka ovat tarpeen alarajan vahvistamiseksi edellä kuvatulla tekniikalla.

SST-heuristiikan aika- ja tilavaativuudesta

SST-heuristiikan aikakompleksisuus riippuu pitkälti siitä, mitä tarkkaa algoritmia esiprosessointina suoritettava SKS-ylärajaheuristiikka käyttää avukseen. Ellei valetäsmäyksien tilalle yritetä löytää kelvollisia täsmäyksiä, on aikakompleksisuus

[132] Muussa tapauksessa jonoa ”BBBA” vastaava yläraja voisi lisäksi muodostua myös indeksipareista {(1, 2), (2, 4), (3, 5), (6, 6)}, {(1, 2), (2, 4), (4, 5), (5, 6)}, {(1, 2), (2, 4), (4, 5), (6, 6)}, {(1, 2), (3, 4), (4, 5), (5, 6)}, ({1, 2), (3, 4), (4, 5), (6, 6)}, {(2, 2), (3, 4), (4, 5), (5, 6)} tai {(2, 2), (3, 4), (4, 5), (6, 6)}.

[133] Jos ensimmäinen täsmäyksistä on valetäsmäys, asetetaan tutkittavan suorakulmion vasemmaksi yläkulmaksi piste (1, 1), ja jos viimeinen niistä on valetäsmäys, asetetaan suorakulmion oikeaksi alakulmaksi piste (*m*, *n*).

identtinen SKS:n kanssa, sillä ainoa lisätyö on ylärajaan kuuluvien täsmäysten laillisuuden tutkiminen, joka vie aikaa $O(p_{ylär})$. Jos valetäsmäysketjujen ympärille muodostuvat suorakulmiot halutaan tutkia, tulee lisätyötä jokaista suorakulmiota varten niin paljon, kuin sovellettava tarkka algoritmi sitä vaatii. Pahimmassa tapauksessa joudutaan suorittamaan tarkka algoritmi koko syötejonoille, jos ylärajaheuristiikan palauttamat kaikki täsmäykset ovat valetäsmäyksiä. Siten alarajan vahvistaminen on kannattavaa ainoastaan silloin, kun jo SST-perusheuristiikan laskema alaraja on verrattain pitkä. Tällöin ei ole riskiä, että valetäsmäysten korvaamistarkastelu veisi kohtuuttomasti aikaa.

Koska SST ei tarvitse mitään oleellisia lisätietorakenteita SKS-ylärajaheuristiikan vaatimien lisäksi, myös SST:n tilavaativuus määräytyy suoraan SKS:n käyttämän tarkan algoritmin perusteella. Alarajan vahvistamisessa tarvittavalla apuvektorilla Z ei ole vaikutusta heuristiikan asymptoottiseen tilankäyttöön.

5.2.4.2 Aliongelmien tulosten yhdistämisheuristiikka (ATY)

Aliluvussa 5.1.3 ehdittiin tutustua OEA-ylärajaheuristiikkaan, joka perustuu syötejonojen merkkien ryhmittelyyn eri aliluokkiin. Kriteerinä aliluokkiin jakamisessa on *merkkien yleisyys* kummassakin syötejonossa. Yksinkertaisinta on toteuttaa syöttöaakkoston jako kahteen aliluokkaan: *yleisiin* ja *harvinaisiin* merkkeihin. Jos kummankin syötejonon merkkijakauma on samanlainen ja vieläpä tasainen, kannattaa aakkosto jakaa kahtia, jotta kumpaankin ositteeseen tulisi likimain yhtä paljon merkkejä. Tällöin muodostuu kaksi aliongelmaa, joiden syötejonot ovat pituudeltaan melko tarkalleen puolet alkuperäisistä pituuksista m ja n.

Sen sijaan, jos molemmat syötejonot edustavat samaa luonnollista kieltä, usein jo kahdeksan yleisintä merkkiä muodostavat esiintymistiheydeltään noin 2/3 sen kaikista symboleista [ROA09]. Tällöin voidaan hyvällä syyllä olettaa, että myös jonojen pisimpään yhteiseen alijonoon kuuluvista merkeistä selvä enemmistö on näitä ns. yleisiä merkkejä. Tällöin on perusteltua jakaa luonnollisen kielen aakkosto kahteen ositteeseen siten, että ensimmäiseen kuuluvat yleiset ja jälkimmäiseen harvinaiset merkit. Ensimmäisen aliongelman syötejonot ovat pituudeltaan lähes kaksinkertaiset jälkimmäiseen verrattuna, mutta vastapainona syöttöaakkoston koko 8 on vain runsas kolmannes harvinaisten merkkien lukumäärästä. Jos aliongelmien PYAn ratkaisemiseksi käytetään nyt algoritmia, jonka suoritusaika riippuu oleellisesti syöttöaakkoston koosta, saadaan sekä tasaisella että luonnollisen kielen merkkijakaumalla ajoaikaa nopeutettua alkuperäisen ongelman ratkaisemiseen tarvittavasta ajasta [BHR98, 39].

Pelkkä OEA-heuristiikan jokaisen yksittäisen aliositteen PYA kelpaa sellaisenaan alkuperäisen ongelman PYAn alarajaksi, sillä aliongelmiin jako säilyttää alkuperäisessä ongelmassa esiintyneen symbolien keskinäisen järjestyksen. Siten esimerkiksi yksinomaan yleisimpien merkkien ositteen PYAn ratkaiseminen tuottaa laillisen

alarajan alkuperäisten syötejonojen PYAn pituudelle. Pelkästään yleisimpien merkkien ositteeseen perustuvaa alarajaa symbolisoi vuonna 1998 julkaistussa konferenssiartikkelissa [BHR98, 35–36] lyhenne *L5*. Heuristiikasta käytetään tässä työssä lyhennettä *ATY*. Heuristiikka palauttaa alarajan lisäksi ne yleisimpien merkkien ositteen indeksiparit, joissa esiintyvät merkit kuuluvat ositteen PYAan.

Pulmallista ATY-heuristiikassa on kuitenkin se, että alkuperäisten syötejonojen PYAn pituuden alarajaksi saadaan edellä kuvatulla aakkoston ositusmenettelyllä enintään 50 – 67 % lyhyemmän syötejonon *X* pituudesta. Jos syötejonot ovat likimain identtiset, alarajan laatu jää verraten heikoksi. Alarajan laatua parantamaan on ATY-heuristiikasta kehitetty myös *vahvistettu versio* [BHR98, 36], joka esitellään seuraavaksi. Heuristiikkaa käsittelevässä konferenssiartikkelissa siitä käytetään lyhennettä *L5'*.

ATY-heuristiikan vahvistaminen

Samoin kuin SST- niin myös ATY-heuristiikan palauttaman tuloksen laatua voidaan yrittää parantaa *loppuprosessoinnilla*. Siinä missä SST:ssä pyrittiin korvaamaan valetäsmäyksiä oikeilla, ATY pyrkii löytämään harvinaisten merkkien muodostamia täsmäyksiä niiltä syötejonojen indeksialueilta, jotka jäävät yleisimpien merkkien muodostamien täsmäysten väliin. Oletetaan, että kaksi perättäistä yleisimpien merkkien muodostamaa täsmäystä löytyy alkuperäisistä syötejonoista paikoista (*i*, *j*) ja (*u*, *v*), missä $i < u-1$ ja $j < v-1$. Tällöin näiden kahden täsmäyksen väliin voi teoriassa mahtua ainakin yksi harvinaisten merkkien muodostama täsmäys. Alarajaa voitaisiin siten nyt yrittää vahvistaa kopioimalla kaikki syötteiden harvinaiset merkit väliltä $X[i+1..u-1]$ ja $Y[j+1..v-1]$ vektoreihin *X'* ja *Y'* ja ratkaisemalla niiden välinen PYA. ATY-heuristiikan alaraja kasvaa alueen PYAn pituuden verran, ja sen muodostavat merkit liitetään alarajaan kuuluviksi merkeiksi täsmäysten (*i*, *j*) ja (*u*, *v*) väliin. Vastaavanlainen tarkastusoperaatio suoritetaan kaikkien perättäisten yleisimpien merkkien täsmäysten välisille alueille, kunhan sen molemmat dimensiot ovat ei-tyhjät. Samoin myös ensimmäistä täsmäystä edeltävä ja viimeistä täsmäystä seuraava alue pitää tutkia.

Tarkastellaan uudelleen esimerkkiä 5.3. Siinä ensimmäisen aliongelman, jonka $p = 7$ ja $\Sigma' = \{$"A", "B"$\}$, ratkaisuksi kelpaava PYA-jono löytyy indeksipareista (1, 2), (2, 7), (3, 8), (8, 9), (10, 15), (14, 16) ja (15, 17). Näin saatua alarajaa voitaisiin nyt yrittää vahvistaa ratkaisemalla alueen $X[9]$ ja $Y[10..14]$ eli alkuperäisten syötteiden osajonojen "C" ja "CCCDD" välinen PYA. Sen pituus $p = 1$, ja (dominantti) täsmäys löytyy paikasta (9, 10). Tämä selvästikin "mahtuu" ensimmäisen aliongelman täsmäysten (8, 9) ja (10, 15) väliin, joten vahvistamisen ansiosta heuristinen PYA pitenee seitsemästä kahdeksaan tarkan PYAn pituuden ollessa yhdeksän.

Kannattaa huomioida, että merkkien $X[9]$ ja $Y[10..14]$ rajoittama suorakaiteen muotoinen alue on esimerkissä todellakin ainoa, joka joudutaan tutkimaan alarajan vahvistamiseksi, sillä aliongelmassa merkeistä $X[1..3]$ kukin muodostaa täsmäyksen, ja

niistä viimeinen täsmää *Y*[8]:n kanssa. Seuraava täsmäyksen muodostava *X*:n merkki on vasta *X*[8], mutta koska se täsmää merkin *Y*[9] kanssa, ei kahden perättäisen täsmäyksen (3, 8) ja (8, 9) väliin jää yhtään tilaa vektorissa *Y*. Tutkittavan alueen *X*[9], *Y*[10..14] jälkeen *X*[10] muodostaa täsmäyksen *Y*[15]:n kanssa, ja *X*[14] ja *X*[15] täsmäävät *Y*:n kahden viimeisen merkin kanssa. Siten *X*[9], *Y*[10..14] on syötemerkkijonojen ainoa indeksialue, joka joudutaan tutkimaan PYAn alarajan vahvistamiseksi.

ATY-alarajaheuristiikka vahvistamattomana vastaa likipitäen OEA-ylärajaheuristiikan suoritusta pelkästään yhdelle, yleisimpien merkkien ositteelle. Ainoana lisäyksenä on, että myös ratkaisujono alkuperäisine indekseineen palautetaan. Vastaavasti vahvistaminen tapahtuu periaatteessa samaan tapaan kuin edellä esitetyssä SST-heuristiikassa: tutkitaan täsmäysten väliin jäävät, pelkästään harvinaisista merkeistä koostuvat alueet[134]. Täten ATY-heuristiikan pseudokoodia ei tässä erikseen esitetä.

ATY-heuristiikan aika- ja tilavaativuudesta

Mikäli ATY-heuristiikkaa käytetään vahvistamattomana, sen aikavaativuus on sama kuin OEA-heuristiikan ensimmäisen[135] aliongelman vaatima suoritusaika, sillä vain yksi OEA:n aliongelmista pitää ratkaista. Se, että ATY palauttaa vastauksessaan myös ratkaisujonon merkkien indeksit alkuperäisissä syötejonoissa, ei vaikuta menetelmän asymptoottiseen suoritusaikaan. Jos sen sijaan alarajaa vahvistetaan, on SST-heuristiikan tavoin pahin tapaus se, että yleisimpien merkkien aliositteen PYA on tyhjä. Tällöin joudutaan ratkaisemaan tarkka PYA syötejonojen kaikille harvinaisille merkeille[136]. Kyseisen operaation lisäkustannus riippuu suoraan käytettävästä tarkasta algoritmista.

Vahvistamaton ATY-heuristiikka vaatii OEA-ylärajaheuristiikkaan verrattuna lisää tilaa ainoastaan indeksivektoreille, joiden avulla pystytään selvittämään, missä paikoissa yleisimpien merkkien aliositteen PYAan kuuluvat täsmäykset sijaitsevat alkuperäisissä syötevektoreissa. Koska näiden kummankin pituus on $O(p_{ylär})$, ei indeksivektoreilla ole vaikutusta asymptoottiseen tilantarpeeseen, joka on aina OEA-heuristiikalla vähintään $O(n+\sigma)$ käytettävästä tarkasta menetelmästä riippumatta. Vahvistetun version tilankulutus on suurimmillaan silloin, jos yleisimpien merkkien ositteen PYAn pituus on 0, eli joudutaan kutsumaan tarkkaa menetelmää syötejonojen kaikille harvinaisille merkeille. Tilankulutuksen suuruus riippuu tällöin siitä, mitä tarkkaa PYA-menetelmää sovelletaan alarajan vahvistamista varten.

---

[134] Alueiden yleisiä merkkejä ei selvästikään kannata enää tarkastella uudelleen, sillä jos ne muodostaisivat täsmäyksiä, ne olisivat jo löytyneet yleisimpien merkkien PYAa etsittäessä.

[135] Olettaen, että juuri ensimmäisessä OEA-ylärajaheuristiikan aliositteessa käsitellään yleisimmät merkit.

[136] Jos yleisimpien merkkien ositteen PYA on tyhjä jono, niin myös harvinaisten merkkien ositteen PYAn pituudeksi tulee väistämättä 0 yleisimpien merkkien määritelmän perusteella (kaikkien symbolien parittaisten minifrekvenssien on oltava nollia). Tällöin alarajan vahvistaminen on ilmeisen turhaa.

5.2.5 FASTA-heuristiikka (FTA)

Viimeisenä kahden merkkijonon PYAn pituuden alarajaheuristiikoista esitellään seuraavassa muunnelma[137] alun perin molekyylibiologian tarpeisiin kehitetystä *FASTA*-heuristiikasta [PLi88, 2444–2448]. Muunnettu FASTAn versio eli *L6* [BHR98, 36] tunnetaan tässä työssä lyhenteellä *FTA*. Se on kolmen ensimmäisen alarajaheuristiikan YMF, CPH ja PMX tavoin puhtaasti heuristinen menetelmä, eli se ei tarvitse avukseen mitään tarkkaa PYA-algoritmia, kuten SST ja ATY tarvitsevat.

FTA-heuristiikka perustuu PMX:n tavoin ahnaaseen täsmäysten valintaan, mutta näiden kahden menetelmän toimintatavat ovat tästä yhteisestä piirteestä huolimatta varsin kaukana toisistaan. Siinä missä PMX pyrki löytämään aina paikallisesti taloudellisimman täsmäyksen – eli sellaisen, joka maksimoi täsmäystä seuraavien syötejonojen loppuliitteiden minimipituuden – FTA pyrkiikin ensi töikseen löytämään syötteidensä *pisimmän yhteisen osajonon*. Heuristiikalle pitää antaa syötteenä ns. *rajaparametri*, joka kuvaa, miten pitkä hyväksyttävän osajonon pitää vähintään olla. Parametrin valinta vaikuttaa ratkaisevasti sekä algoritmin ajankäyttöön että sen laskeman alarajan laatuun. Turvallinen valinta rajaparametrin arvoksi on selvästikin *ykkönen*, jolloin jo yksittäinen täsmäys tulee noteeratuksi, mutta tällöin heuristiikan suoritus voi kestää pitkään. Jos taas parametrin arvoa kasvatetaan ykkösestä arvoon k, heuristiikka hylkää kaikki tätä lyhyemmät perättäisten täsmäysten ketjut. Suoritus nopeutuu melkoisesti, mutta tällä on hintansa: jos syötejonojen täsmäykset ovat sirottuneet satunnaisesti ympäri niistä muodostettua matriisia, voi huonossa tapauksessa käydä niin ikävästi, ettei jonoilla ole yhtään vähintään k:n mittaista yhteistä osajonoa. Tällöin heuristiikka palauttaisi nopeasti tuloksen 0, joka olisi informaatioarvoltaan mitätön. Parhaimpana vaihtoehtoehtona rajaparametrin valitsemiseksi saattaisi olla sen dynaaminen muuttaminen algoritmin suorituksen aikana – mitä pienempikokoinen syöte, sitä pienempi rajaparametri asetetaan. Tällöin voitaisiin ensimmäisiä valintoja tehtäessä karsia ylläpidettäviä vaihtoehtoja tehokkaasti, kun taas syötteen koon pienentyessä pieni rajaparametrin arvo estäisi hukkaamasta lyhyitä täsmäysketjuja, joiden löytymisellä on kuitenkin ilmeinen vaikutus saavutettavan alarajan laatuun.

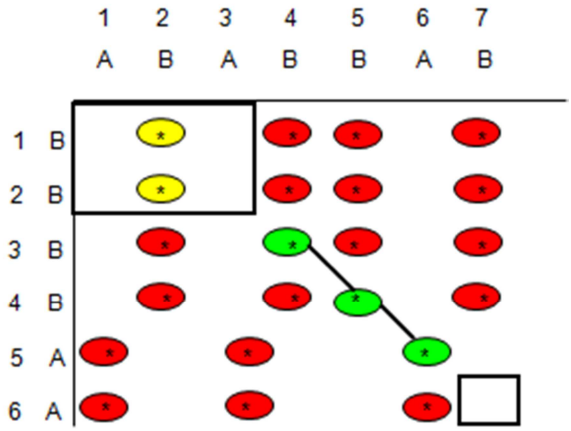
Heuristiikka pitää kirjaa kunkin löytämänsä täsmäysketjun alkamiskohdasta sekä sen pituudesta, kunhan täsmäysketju on vähintään asetetun rajaparametrin mittainen. Menetelmä valitsee ahnaasti näistä pisimmän ja tekee siten päätöksen, että PYAn alaraja muodostetaan käyttämällä kyseistä täsmäysjaksoa. Valittu jakso – olkoon sen alkupisteen koordinaatit (i, j) ja loppupisteen vastaavasti (u, v) – jakaa alkuperäisen ongelman kahteen pienempään suorakulmion muotoiseen ositteeseen. Näistä toinen

[137] Molekyylibiologiassa DNA- ja proteiiniketjujen välisen etäisyyden mittaamisessa käytetään muitakin kriteereitä kuin tietojenkäsittelyssä, joten etäisyyskäsitteet eivät vastaa toisiaan yksi yhteen [BHR98, 36].

alkaa matriisin vasemmasta yläkulmasta ja päättyy pisteeseen (i–1, j–1). Toinen ositteista sijoittuu puolestaan täsmäysjakson oikeaan alanurkkaan ja alkaa pisteestä (u+1, v+1) päättyen maalisolmuun (m, n). Mikäli pisin täsmäysjakso ei ole yksikäsitteinen, vaan maksimin mittaisia löytyy useita, valitaan niistä se, joka sijaitsee lähinnä pää- ja maalidiagonaalin väliin jäävää aluetta, jotta alaraja voisi kehittyä mahdollisimman pitkäksi. Ilmeistä on, että pisimmän täsmäysjakson sijainti vaikuttaa ratkaisevasti siihen, miten hyvän alarajan FTA-heuristiikka pystyy löytämään. Jos todellinen PYA kulkee samaista jaksoa pitkin, valinta on onnistunut ja johtanee laadukkaaseen alarajaan. Jos taas ajaudutaan kauas todellisen PYAn sijaintipaikasta, mitä ilmeisimmin alarajasta tulee laadultaan vaatimaton.

FTA-heuristiikkaa kutsutaan pisimmän täsmäysjakson löydyttyä rekursiivisesti sen kummankin päätepisteen sekä matriisin vasemman ylä- ja oikean alakulman rajoittamille alueille. On selvää, että näiltä alueilta löytyvät täsmäysketjut voidaan liittää aikaisemmin löydetyn pisimmän täsmäysjakson alkuun ja loppuun, sillä alueiden koordinaatit eivät ole päällekkäisiä keskenään. Tällä tavoin jatketaan, kunnes rajaparametrin mittaisia täsmäysketjuja ei löydy enää tutkittavilta alueilta. Seuraava esimerkki valaissee FTA-heuristiikan toimintaa.

Esimerkki 5.7: *FTA-ylärajaheuristiikan toiminta, kun* X = ”BBBBAA”, m = 6, Y = ”ABABBAB”, n = 7, p = 4 ja *PYA* = ”BBBB” / ”BBBA”.



Jonossa X esiintyy kolmenlaisia pituudeltaan kolmen mittaisia osajonoja: ”BBB”, ”BBA” ja ”BAA”. Näistä ainoastaan ”BBA” esiintyy Y:ssä. Tästä pidempää yhteistä osajonoa ei löydy, joten FTA valitsee täsmäysjakson M[3..5, 4..6]. Se on merkitty **vihreillä** soikioilla ja niitä yhdistävillä viivoilla. Tämän jälkeen tutkitaan matriisin vasemman ylä- ja oikean alakulman laatikoiden sisään jäävät alueet, joiden täsmäykset on värjätty **keltaisiksi**. **Punaisia** täsmäyksiä ei enää tarkastella myöhemmin.

Jos esimerkissä 5.7 rajaparametrin arvoksi asetetaan ykkönen, FTA löytää jopa tarkan PYAn ”BBBA”: ensiksi löytyy kolmen mittainen täsmäysjakso ”BBA”, ja myöhemmin rekursiivisella kutsulla vielä tämän alkuun joko täsmäys (1, 2) tai (2, 2) eli ”B”. Parametrin arvoilla 2 ja 3 löydettäisiin kuitenkin enää kolmen mittainen yhteinen alijono ”BBA”, sillä tällöin ei enää yhden mittainen pelkän merkkiparin ”B” muodostama täsmäys vasemmassa ylänurkassa kelpaisi. Parametrin arvolla 4 FTA palauttaisi jo triviaalin alarajan 0, koska syötejonoilla ei ole missään vähintään neljän mittaista yhteistä osajonoa.

Eri osajonojen sijaintipaikat vektorissa *Y* kerätään hajautustauluun siten, että *Y*:stä luetaan paikasta 1 lähtien aina rajaparametrin mittainen määrä merkkejä peräkkäin, ja lasketaan niiden osoite *hajautustaulussa*. Jos *Y* alkaisi vaikkapa merkeillä ”ABCA” ja rajaparametrina olisi 2, löytyisivät *Y*:n alusta merkkijonot ”AB”, ”BC” ja ”CA”. Jokaista mahdollista rajan mittaista merkkiyhdistelmää kohti kirjataan sarakkeet, mistä kyseinen merkkiyhdistelmä löytyy, muistiin hajautustauluun. Hajautusfunktiona voi käyttää vaikkapa merkkien bittiesityksen välisen XOR-operaation tulosta. Taulun valmistuttua lähdetään selaamaan vektoria *X* alusta loppuun samoin rajan osoittama merkkimäärä kerrallaan. Nyt löydetään hajautustaulusta kaikki ne paikat, missä vastaavat perättäiset merkit täsmäävät *Y*:ssä. Lisäksi pitää huomioida, että jos rajan mittainen täsmäys löytyy paikasta (*u*, *v*), mutta vähintään vastaavan mittainen täsmäysketju päättyy pisteeseen (*u*–1, *v*–1), pitää tätä aikaisemmin löytyneen täsmäysketjun pituutta kasvattaa nyt rajan verran, jotta täsmäysketjujen pituudet pysyisivät ajan tasalla.

Aika- ja tilavaativuudesta

FASTA-heuristiikka on herkkä rajaparametrin valinnalle, hajautusfunktion osoitteiden jakautumiselle sekä syötteiden merkkijakaumalle. Mitä pienempi rajaparametri, mitä pienempi syöttöaakkosto ja mitä enemmän täsmäyksiä, sitä todennäköisemmin hajautustauluun muodostuu harvoihin osoitteisiin pitkiä ketjuja, jolloin sieltä hakeminen tulee kalliiksi. Jos molemmissa syötevektoreissa on pelkästään yhtä merkkiä, esiprosessoinnin kustannukseksi tulee $O(mn)$. Silloin ei ole suurta lohtua siitä, että itse laskenta sujuu tämän jälkeen rivakasti. Yleisesti algoritmin suoritusajalle saadaan karkeaksi ylärajaksi $O(n(m/raja)^2)$ [BHR98, 36].

Hajautustaulun rakentamiseen ja sieltä hakuun kuluva aika on keskimäärin $O(mn / (z \cdot raja))$, missä *z* on hajautustaulun osoitteiden määrä, kun taulu on kohtalaisen tasaisesti täytetty. Täsmäyslistojen määrä maksimoituu silloin, kun jokaista rajan mittaista täsmäysketjua seuraa tarkalleen yksi piste, jossa ei ole täsmäystä – muutoinhan muodostuisi pidempi yhtenäinen täsmäysketju. Pahimmillaan täsmäysten listan pituus on siten syöttöaakkoston koolla 2 ja rajaparametrilla 1. Tilakompleksisuus voidaan lausua yleisesti muodossa $O(mn / (raja + 1))$. Menetelmän pseudokoodi löytyy liitteestä kohdasta 11.7.5.

6 Heuristiikkojen hyödyntäminen tarkoissa PYA-algoritmeissa

Edellä tarkasteltiin kolmenlaisia PYA-algoritmeja: suoraan kahden merkkijonon PYAn määrääviä menetelmiä (luku 3); PYAn ratkaisemista varten muunnettuja, alun perin kahden merkkijonon lyhimmän editointietäisyyden laskevia menetelmiä (luku 4) ja heuristisia algoritmeja PYAn ylä- tai alarajan määräämiseksi (luku 5). Seuraavassa tutkitaan, miten heuristisia menetelmiä voidaan *upottaa osaksi tarkkaa PYA-algoritmia*. Tavoitteena on saavuttaa oleellisia säästöjä tarkan menetelmän suoritusajassa sekä mahdollisesti myös tilankäytössä. Aluksi esitellään yleiset perusperiaatteet, millä tavalla *ennakkotieto PYAn ylä- ja/tai alarajasta* edesauttaa tarkan algoritmin tehokkaampaa suoritusta. Asiaa tarkastellaan kutakin kolmea PYA-algoritmien laskentastrategiaa kohti. Jälkimmäisessä aliluvussa selvitetään, voidaanko tarkka algoritmi saada toimimaan vieläkin tehokkaammin laskemalla heuristinen alaraja *toistuvasti* tarkan algoritmin suorituksen jo ollessa käynnissä. Tämä luku perustuu pitkälti kahteen tutkimusryhmämme työhön [BHV03] [Ber05].

6.1 Tarkkojen menetelmien heuristinen esiprosessointi

6.1.1 Alarajan hyödyntäminen

Tarkastellessamme varsinaisten PYA-algoritmien kehitystä luvussa 3 havaitsimme, että menetelmien tehostumisen salaisuutena on pitkälti ollut niiden suorittaman *laskennan aikaisen kirjanpidon väheneminen*. Ensimmäinen PYA-algoritmi vuodelta 1974 – *Wagnerin ja Fischerin* WFI – laski ja tallensi syötejonojen *kaikkien pareittaisten osajonojen* PYAn pituuden [Wag74]. Kolme vuotta myöhemmin *Hunt ja Szymanski* oivalsivat, että tarkastelu voidaan rajoittaa pelkästään niihin syötejonojen indeksipareihin, joissa sijaitsee *täsmäys* [Hun77]. Vielä samana vuonna *Hirschberg* totesi, että PYA-ongelma pystytään ratkaisemaan kirjaamalla muistiin pelkät *dominantit täsmäykset*, eli välttämättä tarkastelua vaativien syötejonojen pisteparien lukumäärä väheni entisestään [Hir77]. Vuonna 1994 *Rick* osoitti myös osan dominanteista täsmäyksistä olevan turhia, mikäli PYAn pituus on entuudestaan tiedossa. Hän otti käyttöön *minimaalisen todistajan* käsitteen. Minimaaliset todistajat muodostavat *dominanttien täsmäysten alijoukon, joka on edelleen tarpeeksi peittävä, jotta PYA-ongelma pystyttäisiin ratkaisemaan pelkästään siihen kuuluviin täsmäyksiin turvautuen* [Ric94]. Kiusallisena mutkana matkassa oli kuitenkin oletus siitä, että PYAn pituuden olisi oltava tiedossa, ennen kuin minimaalisten todistajien joukko on selvitettävissä.

Vaikkei *tarkkaa* PYAn pituutta olisikaan tiedossa, sen asemesta on mahdollista nopeasti laskea pituudelle *alaraja* luvun 5 perusteella. Sen laskeminen esiprosessointina

ei vaikuta tarkan PYA-algoritmin asymptoottiseen suoritusaikaan kuin mahdollisesti termin σ verran, mikäli tyydytään käyttämään jotain nopeimmista alarajaheuristiikoista YMF, CPH tai PMX. Ennakolta laskettua alarajaa voidaan käyttää hyväksi rajoittamaan dominanttien täsmäysten joukkoa niin, että se lähestyy kooltaan minimaalisten todistajien joukkoa. Mitä tiukempi alaraja saadaan, sitä vähemmän turhia dominanttitäsmäyksiä joudutaan tutkimaan [BHR98, 40; BHV03, 296–297].

6.1.1.1 Alarajan hyödyntäminen rivi kerrallaan prosessoitaessa

Rivi kerrallaan prosessoivissa menetelmissä tieto alarajan pituudesta mahdollistaa *rivin käsittelyn katkaisun* sekä alku- että loppupäästä. Oletetaan ensiksi, että ollaan prosessoimassa *vasemmalta oikealle* päin riviä i ($1 \leq i \leq m$), jolta löydetään (dominantti) k-täsmäys sarakkeesta j ($0 < k \leq j \leq n$). Lisäksi oletetaan, että tarkasteltavan ongelman PYAn pituuden alarajaksi on ennalta laskettu $k+h$ ($0 < h \leq m–k$). Jos nyt havaitaan, että juuri löytyneen k-täsmäyksen indeksi $j > n–h$, osoittautuu kyseisen täsmäyksen kirjaaminen turhaksi. Tämä selittyy sillä, että sarakkeen j jälkeen on syötejonosta Y jäljellä enää korkeintaan h–1 merkkiä. Siten pituudeltaan vähintään alarajan $k+h$ mittainen PYA ei enää mahdu kulkemaan pisteen (i, j) kautta, vaan kyseinen (dominantti) täsmäys sijaitsee *liian lähellä matriisin oikeaa reunaa*.

Koska luokan k kynnysarvojen määritelmän[138] perusteella $Kynnys_{i,k} < Kynnys_{i,k+1}$ aina, kun $Kynnys_{i,k} < n+1$, siitä seuraa, että luokan k dominantin täsmäyksen osoittauduttua turhaksi myös kaikkiin järjestysnumeroltaan k:ta ylempiin luokkiin mahdollisesti kuuluvien täsmäysten kirjaaminen rivillä i käy yhtäläisesti tarpeettomaksi. Sama pätee lisäksi kaikkiin niihin dominantteihin täsmäyksiin, jotka matriisin myöhemmillä riveillä seuraavat näitä turhiksi osoittautuneita täsmäyksiä. Rivin i käsittely voidaan edellisen analyysin perusteella siten lopettaa heti, kun siltä löydetään ensimmäinen mihin tahansa luokkaan k kuuluva täsmäys, jonka sijaintisarake j ei toteuta ehtoa $n–j \geq h$, missä $h = p_{alar}–k > 0$.

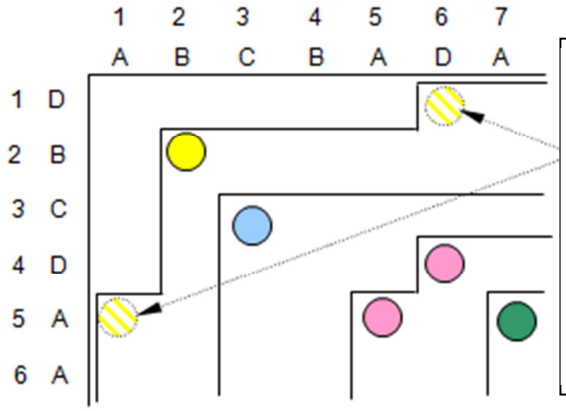
Vastaavankaltainen tilanne kuin edellä rivin i lopussa voi syntyä myös lähestyttäessä syötteistä muodostetun matriisin alareunaa. Jotta rivillä i olisi hyödyllistä kirjata luokkaan k kuuluva, sarakkeessa j sijaitseva täsmäys, pitää myös syötejonossa X esiintyä kyseisen rivin jälkeen riittävästi merkkejä, jotta vähintään $k + h$:n mittainen PYA mahtuisi kulkemaan pisteen (i, j) kautta. Graafisesti tulkittuna piste (i, j) ei siis saa sijaita myöskään *liian lähellä matriisin alareunaa*, jotta sen kirjaaminen olisi yhä hyödyllistä. Tämä tarkoittaa sitä, että *alimman täsmäysluokan k*, jota kannattaa tarkastella rivillä i, on täytettävä *vaatimus* $m–i \geq h$, kun $h = p_{alar}–k$. Toisin sanoen,

[138] kts. lemma 2.1

k-täsmäys (*i*, *j*) on tarpeen kirjata muistiin, eli se *voi kuulua*[139] luokan *k* minimaalisiin todistajiin *tarkalleen silloin*, kun $Min\{m–i, n–j\} \geq p_{alar}–k$.

Heuristisen alarajan hyväksikäyttö tarjoaa vasemmalta oikealle päin riveittäin prosessoiville PYA-algoritmeille edellisen analyysin perusteella mahdollisuuden rajata tarkastelujen ulkopuolelle ne dominantitkin täsmäykset, jotka *varmasti* eivät kuulu ongelman minimaalisiin todistajiin käytettävissä olevaan alarajatietoon nojautuen. Matriisiesityksessä tämä tarkoittaa paitsi lähellä oikeaa reunaa niin myös alimmilla riveillä sijaitsevien matalimpien täsmäysluokkien edustajien jättämistä pois laskuista. Mitä tiukempi alaraja on – eli mitä lähempänä se on PYAn todellista pituutta – sitä tehokkaammin tieto PYAn alarajasta suodattaa tarpeettomia (dominantteja) täsmäyksiä tarkastelujen ulkopuolelle. Tämä pitää paikkansa erityisesti silloin, kun PYAn pituus $p \approx m \approx n$, jolloin matriisin alue, jolla PYAan kelpaavat dominantit täsmäykset eli minimaaliset todistajat voivat sijaita, jää hyvin kapeaksi. Alarajatiedon hyödyntäminen riveittäin prosessoivissa algoritmeissa nopeuttaa selvästikin tarkan algoritmin suoritusaikaa, mutta se säästää myös muistitilaa silloin, kun osa PYA-jonojen ylläpitämistä varten tallennettavista linkkivektorien solmuista jätetään perustamatta. Seuraava esimerkki valaisee, miten alarajatietoa voidaan käyttää hyödyksi riveittäin vasemmalta oikealle prosessoivissa tarkoissa PYA-algoritmeissa.

Esimerkki 6.1: *Heuristisen alarajan vaikutus riveittäisen PYA-algoritmin toimintaan, kun X* = ”DBCDAA”, *Y* = ”ABCBADA”, *PYA* = ”BCDA” / ”BCAA”, $p = 4$.



Esimerkissä PYA on neljän mittainen. Mikäli sen heuristiseksi alarajaksi on ennalta laskettu 3 tai 4, rajautuvat nuolilla merkityt dominantit 1-täsmäykset paikoista (1, 6) ja (5, 1) pois tarkasteluista, sillä kumpikin mainituista täsmäyksistä voi kuulua enintään kahden mittaiseen PYAan. Tämä perustuu siihen, että täsmäystä (1, 6) seuraa enää yksi *Y*:n ja täsmäystä (5, 1) puolestaan vain yksi *X*:n merkki.

Jos algoritmin etenemissuunta riveillä on kuitenkin päinvastainen eli *oikealta vasemmalle* kuten HSZ:ssa, ei edellä mainittuja ehtoja toteuttamattomia täsmäyksiä nytkään tarvitse kirjata. Koska riviä *i* prosessoitaessa PYA voi pidentyä edellisen rivin arvostaan *k* ainoastaan yhdellä, voi oikeanpuoleisin kelvollinen *k*+1 -täsmäys sijaita rivillä *i* sarakkeessa *j*, jolle pätee $n – j \geq h – 1$, kun $h = p_{alar} – k$. Tämä tarkoittaa sitä, että

[139] Tässä tarkastelussa dominantin täsmäyksen (*i*, *j*) kuuluminen minimaalisten todistajien joukkoon perustuu suoraan lasketun alarajan mukaiseen arvioon PYAn pituudesta. Jos $p_{alar} < p$, kuten yleensä on asian laita, ei ole takeita sille, että piste (*i*, *j*) kuuluisi samalla myös tarkan ratkaisun minimaalisiin todistajiin. Toisin sanoen, pisteen (*i*, *j*) kuuluminen alarajan minimaalisiin todistajiin on välttämätön, mutta ei riittävä ehto sille, että ko. piste esiintyisi tarkassa ratkaisussa minimaalisen todistajan asemassa.

merkin x_i esiintymälistaa voidaan käydä läpi lopusta alkuun päin tekemättä mitään kirjauksia aina niin pitkälle, kunnes löydetään kyseisen merkin oikeanpuoleisin esiintymä, jonka sarakenumero $j \leq n - h + 1$. Vastaavasti merkin x_i täsmäyslistan selaaminen voidaan lopettaa heti, kun löydetään samaisen merkin muodostama oikeanpuoleisin täsmäys, joka kuuluu sellaiseen luokkaan l, jolle pätee $m - i < h$, missä $h = p_{alar} - l$.

6.1.1.2 Alarajan hyödyntäminen korkeuskäyrä kerrallaan prosessoitaessa

Mikäli algoritmin prosessointitapa olisi ollut riveittäisen asemesta *korkeuskäyrä kerrallaan* etenevä, aiheuttaisi heuristisen alarajan soveltaminen tarkkaan menetelmään *korkeuskäyrien lyhenemisen* niiden alkuperäisestä pituudestaan. Tarkastellaan uudelleen esimerkkiä 6.1. Lähdettäessä etsimään 1. korkeuskäyrän pisteitä löydetään niistä ensiksi dominantti täsmäys (1, 6). Tämä voidaan kuitenkin jättää huomiotta, jos PYAn pituuden tiedetään olevan vähintään kolmen mittainen. Siten vasta seuraavaksi löytyvä dominantti täsmäys (2, 2) hyväksyttäisiin sijaintinsa puolesta kelvolliseksi 1. korkeuskäyrän pisteeksi. Myös kolmas ehdokas eli pisteessä (5, 1) sijaitseva dominantti täsmäys hylättäisiin vastaavalla kriteerillä kuin (1, 6). Täten 1. korkeuskäyrälle jäisi vain yksi ainoa noteerattava täsmäys eli (2, 2). Koska jokaista yksittäistä k. korkeuskäyrän dominanttia täsmäystä voi seurata pahimmassa tapauksessa jopa σ dominanttia k+1 -täsmäystä [Chi90], voidaan säästää huomattavan paljon ajoaikaa ja muistitilaa, kun edes osa turhista dominanteista täsmäyksistä jätetään tutkimatta.

6.1.1.3 Tekniikan soveltaminen NKY-algoritmiin

Tarkastellaan seuraavassa heuristisen alarajan merkitystä vielä *diagonaali kerrallaan* prosessoivalle NKY-algoritmille[140] [NKY82]. Aluksi todistetaan oikeaksi lemma, johon nojautuen osa käsittelyn kohteena olevasta diagonaalista voidaan jättää tutkimatta. Kyseinen lemma todistuksineen on esitetty aiemmin työryhmämme artikkelissa [BHV03, 296–297].

Lemma 6.1: *NKY-algoritmissa voidaan nykyisen diagonaalin tutkiminen keskeyttää, mikäli käy ilmi, ettei sitä pitkin jatkettaessa voida enää millään saavuttaa vähintään lasketun alarajan mittaista PYAa. Tällaiseen tilanteeseen ajaudutaan tarkalleen silloin, kun tarkasteltavalla diagonaalilla $L_i(k)$-arvo alittaa ensimmäistä kertaa rajan $p_{alar} - k + 1$ jollekin $k \in [1..p_{alar}-1]$.*

[140] kts. aliluku 3.3.1

Todistus: NKY-algoritmista muistetaan, että merkintä $L_i(k)$ tarkoittaa syötevektorin Y viimeistä indeksiä, josta alkava loppuliite $Y[L_i(k)..n]$ muodostaa vielä tarkalleen k:n mittaisen PYAn syötevektorin X loppuliitteen $X[i..m]$ kanssa. Jotta käsiteltävää diagonaalia pitkin voitaisiin löytää vähintään alarajan mittainen yhteinen alijono, pitää syötevektorin alkuliitteen olla tällöin väistämättä vähintään $p_{alar}–k+1$:n mittainen. Muutoin Y:n merkit loppuisivat kesken ennen aikojaan, joten nykyisen diagonaalin tutkiminen voidaan lopettaa, kun todetaan, että jollekin k:n arvolle $L_i(k) < p_{alar}–k+1$. □

Edellisen analyysin perusteella myös NKY-algoritmin kohdalla hyvä alaraja-approksimaatio PYAn pituudelle vähentää menetelmän suorittamaa tarpeetonta työtä ja säästää siten ajoaikaa. Asian toteamiseksi tarkastellaan seuraavaksi esimerkkiä 3.23, jossa $m = n = 17$ ja $p = 9$. Oletetaan nyt, että $p_{alar} = 8$. NKY-algoritmi laskee aluksi arvot $L_{17}(1) = 17$, $L_{16}(2) = 12$ ja $L_{15}(3) = 8$. Jotta vähintään kahdeksan mittainen PYA voisi löytyä paraikaa tutkittavalta ensimmäiseltä diagonaalilta, pitäisi algoritmin seuraavaksi määräämän arvon $L_{14}(4)$ olla nyt ≥ 5. Koska merkkiä x_{14} = ”A” ei indeksin $L_{15}(3) = 8$ vasemmalta puolelta kuitenkaan enää löydy positioon 5 mennessä, tiedetään, ettei diagonaalille enää mahdu alarajan saavuttamiseksi tarvittavia kahdeksaa täsmäystä. Siten sen neljän mittainen alkuliite voidaan jättää toistaiseksi tutkimatta.

Ajateltaessa NKY-algoritmin muistintarvetta kannattaa huomioida, että siinä perustetaan aina suuruusluokkaa $O(m^2)$ oleva matriisi $L_i(k)$-arvojen tallentamista varten [NKY82]. Alarajan ollessa tiedossa etukäteen on kuitenkin selvää, että matriisin diagonaaleista korkeintaan $m–p_{alar}+1$ ensimmäistä joudutaan käymään läpi – PYA löytyy varmasti siihen mennessä. Organisoimalla ja indeksoimalla uudelleen algoritmin aputietorakenteenaan käyttämä matriisi selviydytään usein kuitenkin huomattavasti menetelmän varaamaa $m(m+1)$ yksikköä vähäisemmällä muistitilalla. NKY-algoritmia analysoidaan vielä yksityiskohtaisemmin aliluvussa 8.2, jossa tarkastellaan heuristisen esiprosessoinnin ohella myös *tietorakenteiden valinnan* vaikutusta algoritmin suoritusaikaan [BHV03].

6.1.2 Ylärajan hyödyntäminen

6.1.2.1 Rivi tai korkeuskäyrä kerrallaan prosessoivat menetelmät

Ylärajamenetelmien kytkemistä osaksi tarkkaa PYA-algoritmia on tutkittu vähemmän kuin alarajaheuristiikkojen vastaavaa soveltamista [BHV03, 296]. Tärkeimpänä syynä tähän on se, että yksinkertaisinta MFS-heuristiikkaa lukuun ottamatta tunnetuimmat ylärajamenetelmät tarvitsevat avukseen tarkkaa PYA-algoritmia, minkä johdosta *ylärajan laskenta vie usein enemmän aikaa kuin alarajan määrääminen* [BHR98, 39]. MFS suoriutuu ylärajan laskennasta kuitenkin niin nopeasti, ettei sen upottaminen esiprosessointina osaksi tarkkaa PYA-algoritmia käytännössä vaikuta tämän asymptoottiseen suoritusaikaan. Toisena merkittävänä ylärajaheuristiikkojen vähäisen

tarkastelun syynä voidaan pitää sitä, *ettei ylärajan laskennasta ole yleensä sanottavaa hyötyä rivi tai varsinkaan korkeuskäyrä kerrallaan prosessoiville menetelmille*. Rivi kerrallaan prosessoivien algoritmien suoritus voitaisiin tosin pysäyttää heti, kun löydetään ylärajan mittainen PYA, mutta koska yläraja on tarkalleen PYAn pituinen vain ani harvoin, tästä hyödystä ei juurikaan päästä nauttimaan käytännössä. Tietoa ylärajan pituudesta voitaisiin tosin käyttää hyväksi myös tarkan PYA-algoritmin tietorakenteiden kokoa määrättäessä siten, ettei PYAn pituudesta p riippuvia staattisia apuvektoreita varten tarvitsisi varata muistia pahimman varalta kuin muuttujan $p_{ylär}$ arvoon sidottu määrä p:n teoreettisesta maksimista m riippuvan määrän asemesta. Todellisuudessa kuitenkin riveittäisissä menetelmissä enin määrä muistia käytetään joko lähiesiintymätaulukoiden perustamiseen tai täsmäysten sekä niiden välisten linkkien ylläpitoon, joten ennakkotieto PYAn pituuden ylärajasta ei merkitsisi menetelmän muistintarpeen mullistavaa pienenemistä. Rivi kerrallaan prosessoivia algoritmejakin vähäisempi hyöty ylärajan laskemisesta olisi korkeuskäyrä kerrallaan laskentaa suorittaville algoritmeille, sillä niiden suoritushan pysähtyy muutenkin jo heti, kun tarkasteltavalle korkeuskäyrälle ei löydetä enää yhtään täsmäystä!

6.1.2.2 Ylärajan merkitys NKY-algoritmin tehostajana

Ylärajaheuristiikan merkitys diagonaali kerrallaan prosessoivalle NKY-algoritmille on kuitenkin täysin toista suuruusluokkaa kuin rivi tai korkeuskäyrä kerrallaan prosessoiville. NKY-algoritmin suorittaman laskennan kaikki välitulokset nimittäin tallennetaan matriisiin, jolle algoritmi varaa tilaa aina $m(m+1)$ yksikköä. Jos ennalta tiedetään, miten pitkä tarkasteltavan ongelman PYA korkeintaan on, voitaisiin matriisin *diagonaalien pituutta* rajoittaa ylhäältä termillä $p_{ylär}$ kaikilla niillä diagonaaleilla, joiden järjestysnumero on väliltä $0..m+1-p_{ylär}$. Tämän jälkeen jokaisen seuraavan lävistäjän pituus lyhenee aina yhdellä siihen saakka, kunnes saavutetaan lävistäjä p_{alar}, mihin mennessä NKY jo löytää ratkaisun. Siten NKY:n avukseen tarvitseman matriisin kokoa voitaisiin heuristisiten rajojen ohjaamina rajoittaa lausekkeella[141] $p_{ylär}(m + 2 - p_{ylär}) + \frac{1}{2}(p_{ylär} - 1 + p_{alar})(p_{ylär} - p_{alar})$ [BHV03].

Palataan asian havainnollistamiseksi vielä uudelleen esimerkkiin 3.23, jossa kummankin syötejonon pituus on 17 ja PYA on yhdeksän mittainen. Oletetaan kuten hieman aikaisemminkin, että p:n alarajaksi olisi laskettu 8, mutta tämän lisäksi vielä ylärajaksi olisi saatu arvo 12. Alarajan perusteella tiedetään, että matriisista tarvitaan ainoastaan diagonaaleja 0 – 10, sillä kymmenes eli järjestysnumeroltaan $m+1-p_{alar}$ oleva diagonaali on viimeinen, jonka pituus ≥ 8, eli sille mahtuu vielä alarajan mittainen PYA.

[141] Lausekkeen ensimmäinen yhteenlaskettava kuvaa, mikä on ylärajan mittaisten diagonaalien yhteispituus, ja jälkimmäinen kuvaa muistipaikkojen yhteismäärää lyhyemmillä diagonaaleilla, joita on kaiken kaikkiaan $p_{ylär} - p_{alar}$ kappaletta.

Vastaavasti diagonaalit[142] 0 – 6 ovat ainoat, joille mahtuu tarvittaessa ylärajan mukainen eli 12:n pituinen PYA. Jokaisen viimeksi mainitun diagonaalin pituudeksi riittää siis 12, ja tämän jälkeisistä lävistäjistä 7 – 10 on kukin yhdellä edeltäjäänsä lyhyempi. Siten esitetyillä ylä- ja alarajan arvoilla saataisiin esimerkissä matriisin tarpeelliseksi enimmäiskooksi 7 * 12 + 11 + 10 + 9 + 8 = 122 muistipaikkaa, kun taas ilman heuristista esiprosessointia NKY varaisi tarpeettomasti 18 * 17 = 306 muistipaikkaa sisältävän matriisin!

PYAn ylärajan ollessa lyhyt selvitään heuristisia ala- ja ylärajoja hyödyntämällä alkuperäiseen NKY-algoritmiin verrattuna huomattavan lyhyillä diagonaaleilla, ja vastaavasti PYAn ollessa pitkä, diagonaaleja tarvitaan lukumääräisesti vain murto-osa alkuperäisen NKY:n käyttöönsä varaamista. Muistitilan säästäminen vaatii kuitenkin muutoksia NKY:n käyttämän matriisin esitysmuotoon [BHV03]. Tätä tarkastellaan syvällisemmin aliluvussa 8.2.1.

6.2 Heuristiikan toistuva soveltaminen

Edellisessä aliluvussa tarkasteltiin heuristisen ylä- tai alarajan *staattista upottamista* tarkkoihin PYA-algoritmeihin. Raja(t) laskettiin *esiprosessointivaiheessa* ja *vain kertaalleen* ennen varsinaisen PYA-algoritmin suoritusta. Siten heuristiikan laskeman rajan *laatu* vaikuttaa pitkälti siihen, minkä verran tarkan algoritmin työtä käytettävissä oleva tieto PYAn ala- ja/tai ylärajasta pystyy helpottamaan. Koska heuristiikka toimii usein huomattavasti tarkkaa algoritmia nopeammin, on selvää, ettei se hyödynnä läheskään kaikkea sitä informaatiota, jota tarvitaan tarkan PYAn laskemiseksi. Siten on verrattain vähäiselläkin ajatustyöllä konstruoitavissa useita esimerkkisyötteitä, joissa tarkastelun kohteena oleva heuristiikka harhautuu kauas oikealta ratkaisupolulta. Suorittamalla heuristisen rajan laskenta *uudelleen* tarkan algoritmin suorituksen jo käynnistyttyä voidaan aiemmin laskettuja rajoja saada mahdollisesti tiukennettua [Ber05]. Koska ylärajaheuristiikkojen käytöstä saatavat hyödyt ovat edellisen aliluvun perusteella yleisesti ottaen vähäisemmät kuin alarajaheuristiikkojen soveltamisesta saatavat, keskitytään seuraavassa tarkastelemaan ainoastaan alarajan laskevia menetelmiä.

6.2.1 PYAMAX-heuristiikan puutteet

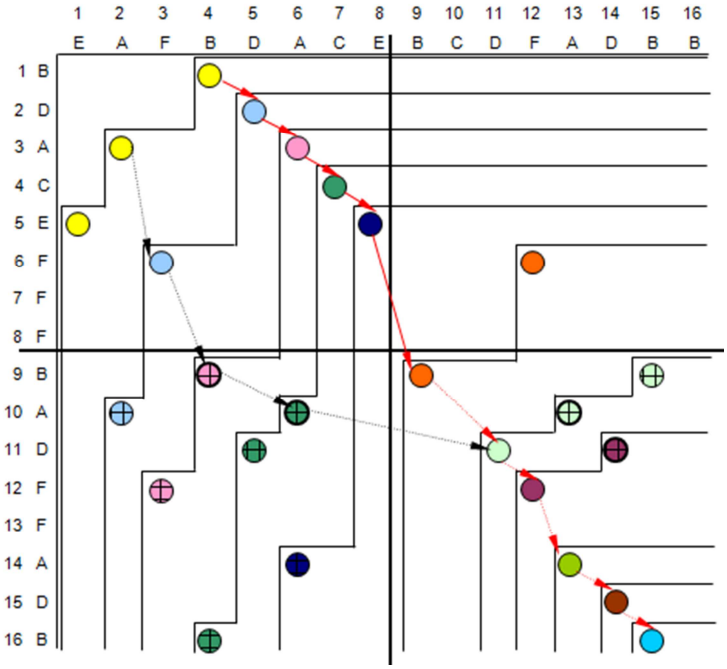
Esiteltäessä alarajaheuristiikkoja aliluvun 5.2 eri alakohdissa kiinnitettiin huomiota useimpien heuristiikkojen heikkoihin kohtiin. Pelkästään syötejonojen merkkien frekvensseihin perustuvien menetelmien – eli YMF:n ja CPH:n – laskeman alarajan

[142] NKY:ssä diagonaalien numerointi tapahtuu siten, että 1. diagonaali alkaa merkin x_m kohdalta, ja yleisesti k. diagonaali alkaa X-indeksin $m+1-k$ kohdalta. Nollas diagonaali tarvitaan pelkkien pysäytysalkioiden asettamiseksi, ja se alkaa pseudomerkin x_{m+1} kohdalta.

laatu riippuu hyvin voimakkaasti syötejonojen merkkijakaumasta. Tarkkaa algoritmia avukseen tarvitsevat heuristiikat SST ja ATY tuottavat PYAn pituudelle edellä mainittuja kahta menetelmää paremman approksimaation, mutta laadusta joudutaan maksamaan frekvenssipohjaisiin menetelmiin verrattuna selvästi pidemmän ajoajan muodossa. FTA-heuristiikan laatuun vaikuttaa puolestaan ratkaisevasti se, mistä päin syötejonoja löytyy pisin yhtenäinen täsmäysten ketju, ja kuuluuko se tarkasteltavan ongelman PYAan vai ei.

Sen sijaan PMX-heuristiikan heikkoja kohtia ei vielä ehditty analysoida kovinkaan syvällisesti. Aikaisemmissa vertailuissa se on kuitenkin osoittautunut käytännössä luotettavaksi, nopeaksi ja siten yleisesti ottaen käyttökelpoisimmaksi alarajaheuristiikaksi [BHR98, 38–40]. Näiden havaintojen perusteella tuntuisi parhaalta vaihtoehdolta turvautua PMX-heuristiikan käyttämiseen, ja näin onkin toimittu artikkeleissa, joissa käsitellään heuristiikkojen upottamista osaksi tarkkoja PYA-algoritmeja [BHV03, Ber05]. Koska PMX valitsee kustakin luokasta ainoastaan *yhden dominantin täsmäyksen*, sen voi otaksua toimivan käytännössä nopeasti mihin tahansa tarkkaan menetelmään verrattuna. Lisäksi se pyrkii valitsemaan kustakin luokasta aina *paikallisesti taloudellisimman täsmäyksen*, mikä takaa sen, ettei syötevektoreita käsitellä tuhlailevasti täsmäyksiä valittaessa. PMX:n kuningasajatuksena on pyrkiä tekemään tarkasteltavalta korkeuskäyrältä katsottuna tilaa aina teoreettisesti mahdollisimman pitkän yhteisen alijonon muodostamiselle. Vaikka ajatus sinänsä on usein mielekäs, ei periaatteen soveltaminen kuitenkaan aina johda toivottuun lopputulokseen, kuten seuraava esimerkki osoittaa [Ber05].

Esimerkki 6.2: *PYAMAX-heuristiikan löytämän alarajan ja tarkan PYAn sijainti, kun* X = ”BDACEFFFBADFFADB”, Y = ”EAFBDACEBCDFADBB”, *PYA* = ”BDACEBDFADB”, $p = 11$ ja $p_{alar} = 9$



Esimerkissä 6.2 syötejonojen PYAn pituus on 11, josta PYAMAX kadottaa kaksi täsmäystä hyppäämällä luokan 4 dominantista täsmäyksestä suoraan luokan 7 edustajaan, josta eteenpäin se löytää kaikki loput tarkkaan ratkaisuun kuuluvat dominantit täsmäykset. Jos kuitenkin tutkitaan matriisista vain sen vasenta yläneljännestä, havaitaan, että syötejonojen alkuliitteiden X[1..8] ja Y[1..8] PYA on viiden mittainen ”BDACE”, mutta PMX löytää alueelta vain kahden pituisen yhteisen alijonon ”AF”, jolla ei ole yhtään yhteistä täsmäystä tarkan ratkaisun kanssa! Syynä tähän on PMX:n ahnas täsmäysten valitsemistapa, jonka ansiosta dominanttia 1-täsmäystä (3, 2) pidetään taloudellisempana valintana kuin kahta muuta ehdokasta (1, 4) ja (5, 1). Koska PMX ei milloinkaan peru kertaalleen tekemiään valintoja, kolmen perättäisen täsmäyksen ketjua (1, 4), (2, 5) ja (3, 6) ei enää koskaan löydetä. Etsittäessä seuraavaksi 2-täsmäyksiä pisteestä (3, 2) lähtien, edellä mainittua ketjua jatkava täsmäys (4, 7) tulkitaan PMX:ssä nyt luokan 2 dominantiksi täsmäykseksi, koska (1, 4) ”unohdettiin”. Nyt vuorostaan merkkien ”F” muodostama dominantti täsmäys paikassa (6, 3) osoittautuu kuitenkin paikallisesti parhaaksi valinnaksi, joten lisäksi täsmäykset (4, 7) ja (5, 8) jäävät kirjaamatta. Täsmäyksen (11, 11) kohdalla heuristiikan valitsema polku ”löytää onnekkaasti” tarkan PYAn ratkaisupolun, jota se ei tässä esimerkissä enää myöhemmin kadota. PYAn ratkaisupolku on merkitty kuvaan **punaisilla** nuolilla.

6.2.2 Alarajan laadun parantaminen toistetulla laskennalla

Edellisessä esimerkissä PMX-heuristiikka ajautui heti laskennan alkutaipaleella sivuraiteille valitsemalla lokaalisti parhaimmalta vaikuttaneen täsmäyksen paikasta (3, 2) oikean valinnan (1, 4) sijaan. Pituudeltaan vaatimattomaksi jäänyttä alarajaa voidaan kuitenkin yrittää parantaa *toistamalla heuristisen rajan laskenta kesken tarkan algoritmin suorituksen*. Vuodelta 2005 peräisin olevassa *Bergrothin* konferenssiartikkelissa [Ber05] on tarkasteltu alarajan toistuvaa laskentaa rivi kerrallaan prosessoivalle *Kuon* ja *Crossin* algoritmille (KCR), mutta menettely voitaisiin yleistää mille tahansa riveittäin etenevälle PYA-algoritmille.

Keskeisenä ajatuksena on esitellä rajaparametri *toisto*, jonka arvo kuvaa, *miten monen rivin prosessoinnin jälkeen tarkan algoritmin suoritus keskeytetään uuden alarajan laskemista varten*. PYAn pituuden alaraja lasketaan ensi kertaa aina ennen tarkan algoritmin suorittamista koko syötemerkkijonoille X ja Y. Merkitään ensimmäistä laskettua alarajaa tunnuksella p_{alar}. Kun tarkan algoritmin laskenta on edennyt rajaparametrin ilmaisevan rivimäärän verran, lasketaan alaraja uudelleen syötejonojen loppuliitteille $X[toisto+1..m]$ ja $Y[toisto+1..n]$, eli rajaparametrin osoittama määrä merkkejä jätetään kummankin syötejonon alusta huomiotta. Merkitään mainittujen loppuliitteiden PYAn alarajaa tunnuksella $p_{alar}U$. Tämän jälkeen etsitään kynnysarvovektorista, mikä on tarkasteluhetkeen mennessä *korkein täsmäysluokka k, jolle on voimassa* $Kynnys_k \leq toisto$. Tällöin alkuliitteiden X[1..*toisto*] ja Y[1..*toisto*]

PYAn pituus on k, ja nyt selvästikin koko alkuperäisen ongelman PYAn pituuden uudeksi alarajaksi kelpaisi summalauseke $k+p_{alar}U$, sillä loppuliitteiden alarajaan $p_{alar}U$ kuuluvien kaikkien täsmäysten X- ja Y-indeksit ovat aidosti suurempia kuin alkuliitteiden k:n mittaisten tarkan PYAn muodostaneiden täsmäysten. Jos edellä mainitun summalausekkeen arvo ylittää termin p_{alar} arvon, saadaan alkuperäiselle ongelmalle alussa laskettua *tiukempi alaraja*, joka rajoittaa entistä voimakkaammin aluetta, jolta kannattaa etsiä täsmäyksiä. Päinvastaisessa tapauksessa uuden alarajan laskennasta ei ollut hyötyä, ja vanha alarajan arvo säilyy voimassa. Muuttujassa p_{alar} säilytetään vastedes pisintä tarkasteluhetkeen mennessä löydettyä alarajaa.

6.2.3 Esimerkkejä alarajan toistetusta laskennasta

Palataan hetkeksi esimerkkiin 6.2. Oletetaan, että parametrille *toisto* annettaisiin arvoksi 8. Koska kumpikin syötejonoista on 16:n mittainen, alaraja laskettaisiin paitsi esiprosessointina, niin myös kahdeksannen rivin käsittelyn jälkeen. Käytettäessä PMX-heuristiikkaa saadaan esiprosessoinnin tuloksena PYAlle alarajaksi 9. Heuristinen ratkaisu on esitetty esimerkissä täsmäyksestä (3, 2) alkavilla, pisteviivoin merkityillä nuolilla. Kun *riveittäin* prosessoivalla tarkalla algoritmilla – esimerkiksi KCR:llä – on ehditty käsitellä kahdeksan ensimmäistä riviä, on alkuliitteiden $X[1..8]$ ja $Y[1..8]$ välisen PYAn pituudeksi saatu 5, eli ts. $k = \{\text{MAX } u \mid Kynnys_u \leq 8\} = 5$. Kutsuttaessa PYAMAX-heuristiikkaa loppuliitteille $X[9..16]$ ja $Y[9..16]$ saadaan niiden PYAn alarajaksi eli $p_{alar}U$:n arvoksi 6. Loppuliitteiden heuristiseen PYAan kuuluvat täsmäykset alkavat kohdasta (9, 9), ja niiden loput viisi täsmäystä ovat samat kuin esiprosessointina lasketussa alarajassa. Täsmäyksestä (9, 9) alkava pisteviivoitettu nuoliketju kuvaa PMX-heuristiikan syötejonojen loppuliitteille löytämää ratkaisua. Koska nyt summa $k+p_{alar}U = 5+6 = 11 > p_{alar} = 9$, saadaan alkuperäistä alarajaa 9 tiukennettua arvoon 11, joka esimerkissä sattuu olemaan jopa PYAn tarkka arvo. Kun seuraavaksi lähdetään jatkamaan tarkan PYAn etsintää riviltä 9 eteenpäin, voidaan nyt karsia kaikki sellaiset dominantit täsmäykset, joiden kautta 11:n mittainen PYA ei mahdu kulkemaan. Ne alkuperäisen ongelman PYAn ratkaisupolun osat, jotka eivät kuulu kumpaankaan heuristiseen ratkaisuun, on esitetty kuvassa yhtenäisillä (ei piste-) viivoilla merkityillä nuolilla. Heuristisen alarajan toistetun laskennan ansiosta tarkastelujen ulkopuolelle jäävät tarkan algoritmin dominantit täsmäykset on merkitty *ristikkokuvioinnilla*. Kyseisistä täsmäyksistä *vahvistetuin reunaviivoin merkityt* olisi kuitenkin jouduttu tutkimaan, jos alarajaa ei olisi päivitetty kesken laskennan. Kannattaa huomioida, että alarajan päivityksen tapahduttua rivin 8 jälkeen selvä enemmistö kaikista toistaiseksi tutkimattomista dominanteista täsmäyksistä jää nyt jatkotarkastelujen ulkopuolelle.

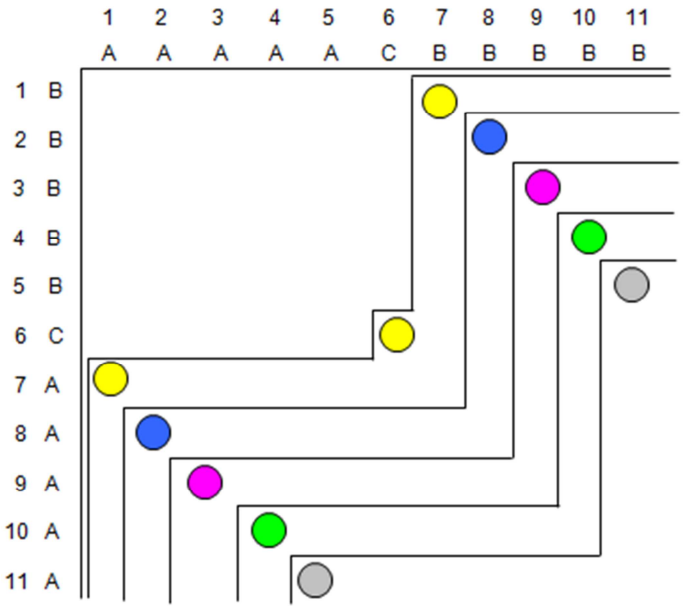
Esimerkissä 6.2 heuristinen alarajan laskenta ehdittiin toistaa ainoastaan kerran ennen syötejonojen loppumista. Mikäli kuitenkin termi $z = \lfloor (m-1)/toisto \rfloor > 1$, alarajan

laskenta toistetaan esiprosessoinnin jälkeen yhteensä z kertaa, ja i. kerralla ($1 \leq i \leq z$) alaraja lasketaan loppuliitteille $X[1+i \cdot toisto..m]$ ja $Y[1+i \cdot toisto..n]$. Jos millä tahansa i:n arvolla $p_{alar} < \text{PYA}(X[1..i \cdot toisto], Y[1..i \cdot toisto]) + p_{alar}U$, päivitetään alarajaksi uusi, tiukennettu arvo.

Rajaparametrin valinnalla on suora vaikutus paitsi kulloinkin käytettävissä olevan alarajan laatuun, niin myös heuristiikalla tehostetun tarkan algoritmin suoritusaikaan. Mikäli alarajaa lasketaan uudelleen kovin harvoin, voidaan pitkään joutua kärsimään aikaisemmin lasketusta huonolaatuisesta alarajasta, ja tarkka algoritmi joutuu suorittamaan vastaavasti enemmän laskentatyötä. Jos taas alarajaa yritetään päivittää liian ahkerasti, alkaa myös heuristisen laskennan teettämä lisätyö vähitellen pidentää suoritusaikaa. Näin käy silloin, kun syötteiden PYA on verrattain lyhyt, jolloin alarajatiedosta ei muutenkaan ole paljoa hyötyä, sekä silloin, kun jo aikaisemmat alarajat ovat olleet hyvälaatuisia, eli toistetulla laskennalla ei enää juurikaan voida tiukentaa aiemmin laskettua alarajaa. Perusteellinen empiirinen testaaminen edesauttaa parhaiten sopivan arvon löytämistä rajaparametrille.

Edellä todettiin, että PMX-heuristiikan toistuva suorittaminen saattaa auttaa mahdollisesti viiveellä toipumaan laadultaan heikon alarajan aiheuttamasta tehottomuudesta. Alarajan laskennan toistaminen ei ole kuitenkaan mikään ihmelääke, joka tehoaisi kaikkiin mahdollisiin PYA-ongelman esiintymiin. Tarkastellaan seuraavaksi esimerkkiä 6.3, joka tekee tyhjiksi toistettuihinkin heuristisiin alarajoihin kohdistetut toiveet.

Esimerkki 6.3: *Heuristisen alarajan vaikutus riveittäisen PYA-algoritmin toimintaan, kun* X = ”BBBBBCAAAAA”, Y = ”AAAAACBBBBB”, *PYA* = ”AAAAA” / ”BBBBB”, $p = 5$



Lähtiessään laskemaan alarajaa esimerkin 6.3 syötejonoille PMX-heuristiikka punnitsisi aluksi kolmen 1-täsmäysten (1, 7), (6, 6) ja (7, 1) keskinäistä paremmuutta. Näistä (6, 6) osoittautuu parhaaksi, sillä sen valitseminen tarjoaa mahdollisuuden löytää kuuden mittainen PYA, kun taas muiden ehdokkaiden valinta mahdollistaa korkeintaan viiden pituisen PYAn löytymisen. Nyt kuitenkin täsmäys (6, 6) osoittautuu melkoiseksi kiusankappaleeksi, sillä sen tultua valituksi ei PMX löydä enää yhtään luokan 2 täsmäystä, joten alarajan pituudeksi saadaan esiprosessoinnin jälkeen laadultaan onneton 1. Jos heuristinen alaraja laskettaisiin nyt uudelleen, kun puolet riveistä eli ylöspäin pyöristettynä 6 riviä on tutkittu tarkkaa menetelmää soveltaen, on alkuliitteiden X[1..6] ja Y[1..6] PYAn pituus 1. Loppuliitteiden X[7..11] ja Y[7..11] heuristinen (ja nyt samalla myös tarkka) PYA on tyhjä merkkijono, joten alaraja ei paranisi yhtään toistetun laskennan ansiosta! Vaikka esimerkki on voimakkaasti kärjistetty ja sen mukainen tilanne esiintyy todellisilla syötteillä erittäin harvoin, toimii se kuitenkin *varoituksena, ettei heuristisen alarajan laskenta edes toistetustikaan suoritettuna aina välttämättä lyhennä syötteiden tarkan PYAn määräämiseen kuluvaa aikaa*.

7 Heuristiikkojen luotettavuuden parantaminen

Kahdessa edellisessä luvussa tutustuttiin heuristiikkojen käyttöön PYA-ongelmassa. Luvussa 5 tarkasteltiin heuristiikkoja sellaisinaan, kun taas luvussa 6 ne upotettiin osaksi tarkkaa PYA-algoritmia. Käytettäessä alarajaheuristiikkoja yksinään saadaan toisinaan tulokseksi laadultaan verrattain vaatimaton alaraja, joka ei sellaisenaan anna paljoakaan informaatiota, joka edistäisi päätöksentekoa siitä, kannattaako tarkastelun kohteina olevia syötemerkkijonoja analysoida lisää. Jos väljäksi jäänyttä alarajaa yritetään puolestaan hyödyntää osana tarkkaa algoritmia, ei tämän suoritus paljoakaan nopeudu käytettävissä olevan alarajatiedon ansiosta. Mikäli taas alarajan sijaan onkin laskettuna pelkkä heuristinen yläraja jonka arvo $\approx m$, ei nytkään voida tehdä mitään luotettavia johtopäätöksiä syötejonojen PYAn pituuden suuruusluokasta. Tässä luvussa esitetään kolme menettelyä, miten heuristisen laskennan luotettavuutta voidaan yrittää parantaa. Ensimmäinen niistä perustuu *sekä ylä- että alarajan laskemiseen* esiprosessointina, toinen pyrkii *uudistamaan alarajaheuristiikan muodostamaa näkymää* syötejonoihin, ja kolmas pyrkii *yhdistämään kahden alarajaheuristiikan tulokset*. Näistä alarajaheuristiikan näkymän siirtämistä on ehditty jo tarkastella aikaisemmassa artikkelissani [Ber07].

7.1 Sekä ala- että ylärajan määrääminen esiprosessointina

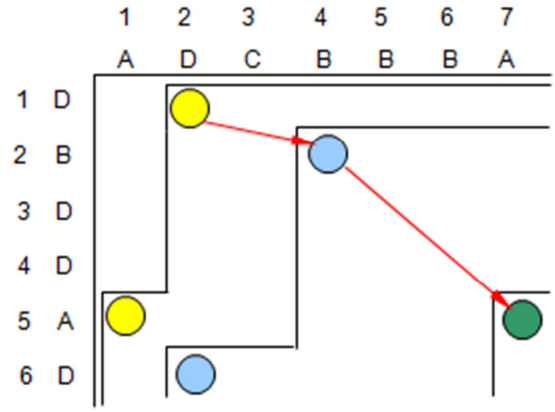
Laskemalla esiprosessointivaiheessa PYAn pituudelle *sekä ala- että yläraja* saadaan syötejonojen PYAn pituudesta usein täsmällisempää tietoa kuin laskettaessa niistä pelkästään jompikumpi. Tällöin saamme nimittäin selville PYAn pituuden *p vaihteluvälin*. Aliluvuissa 5.1 ja 5.2 ehdittiin mainita, että kaksi ala- ja ylärajaheuristiikkaparia on verraten lähellä toisiaan. Alkuperäisen ongelman syöttöaakkoston kokoa supistava SKS-ylärajaheuristiikka voidaan laajentaa SST-alarajaheuristiikaksi rajaamalla sen laskemasta tuloksesta pois valetäsmäykset. Vastaavasti OEA-ylärajaheuristiikan laskema mikä tahansa yksittäinen – esimerkiksi yleisimpien merkkien ositteen – osatulos kelpaa alarajaksi. Näiden kahden ylärajaheuristiikan pulmana on kuitenkin se, että niiden käyttäminen vaatii tarkan PYA-algoritmin suorittamista, joka usein vaatii liian paljon aikaa, vaikkakin heuristiikkojen ratkaisemat ongelmat ovat helpottuneet alun perin asetetusta[143].

Tarkasteltaessa alarajamenetelmiä todettiin, että niistä PMX on usein hyvä valinta, koska se toimii nopeasti ja palauttaa silti useimmiten laadukkaan tuloksen eli hyvän approksimaation *p*:lle. Aliluvussa 6.2 nähtiin kuitenkin esimerkkejä, joissa kyseinen heuristiikka ei löydä läheskään optimaalista ratkaisua. Tällöin syötteiden PYA voi olla

[143] Edellyttäen, että SKS- ja OEA-heuristiikkojen apunaan käyttämän algoritmin suoritusaika riippuu syöttöaakkoston koosta.

pitkä, vaikka alaraja jäisikin lyhyeksi. Jos PMX-heuristiikan laskema alaraja on *absoluuttisesti mitattuna* lyhyt (<< *m*), se ei kuitenkaan mitenkään takaa sitä, että sen laskenta olisi epäonnistunut eli käytännössä ajautunut pois tarkalta ratkaisupolulta, vaan tarkasteltavien syötteiden PYA voi tosiaankin olla lyhyt. Tällaista tulosta voitaisiin yrittää tarkentaa ottamalla avuksi ylärajaheuristiikoista teoreettisesti nopein MFS, joka pitää kirjaa vain merkkien pareittaisista frekvensseistä syötejonoissa. Jos myös yläraja jää lyhyeksi, voidaan syötejonot jättää vaille jatkotarkasteluja, jos ollaan kiinnostuneita vain jonoista, joiden PYA on pitkä suhteessa jonon *X* pituuteen *m*. Seuraava esimerkki kuvastaa tilannetta.

Esimerkki 7.1: *Heuristisen ala- ja ylärajan laskenta esiprosessointina, kun* X = ”DBDDAD”, Y = ”ADCBBBA”, PYA = ”DBA”, $p = p_{alar} = p_{ylär} = 3$.



PMX valitsisi 1-täsmäyksen (1, 2) ja 2-täsmäyksen (2, 4) ja löytäisi samalla tarkan PYAn pituuden 3. Koska aakkoston merkkien {A, B C, D} minimifrekvenssit syötevekto-reissa ovat 1, 1, 0 ja 1, palauttaisi MFS-heuristiikka samoin ylärajaksi arvon 3. Mikäli oltaisiin kiinnostuneita vain syötteistä, joiden p on vaikkapa vähintään 80 % jonon X pituudesta, voitaisiin esimerkin syötejonot jättää vaille jatkotarkasteluita.

Edellinen esimerkki antaa aihetta miettiä, onko ylärajan laskeminen esiprosessoinnissa aina kannattavaa, koska se onnistuu nopeasti käyttämällä MFS-heuristiikkaa. Jos oltaisiin kiinnostuneita yksinomaan syötejonoista, joilla PYAn suhteellista osuutta kuvaava osamäärä p/m on lähellä ykköstä, riittäisi yksistään jo liian alhainen ylärajan arvo lopettamaan syötteiden käsittelyn tarpeettomana. Ylärajan laskenta onkin perusteltua, jos on odotettavissa, että molemmat syötejonot eivät välttämättä noudata keskenään likimain samanlaista merkkijakaumaa. Jos syötejonojen merkkien frekvenssijakaumat kuitenkin muistuttavat läheisesti toisiaan tai syötejonojen välinen pituusero on suuri ($n >> m$), toimii MFS-ylärajaheuristiikka verraten vaatimattomasti ja ylärajoista tulee korkeita myös silloin, kun syötejonojen todellinen PYA on lyhyt. Lisäksi sovelluksissa, jossa p:n arvo on välttämättä laskettava, ei ylärajan laskemisesta ole liiemmälti hyötyä, ellei tarkan PYAn laskennassa haluta käyttää diagonaali kerrallaan prosessoivaa NKY-algoritmia[144].

[144] Perustelut tälle väitteelle löytyvät aliluvusta 6.1.2.

7.2 PYAMAX-heuristiikan syötejononäkymän siirtäminen

Aliluvussa 6.2 esiteltiin esimerkkejä, joissa PMX-heuristiikan laskema alaraja jää laadultaan heikoksi. Yhtenä ehdotuksena ongelman ratkaisemiseksi tarjottiin alarajan toistuvaa laskentaa, mutta kuten esimerkki 6.3 paljasti, tästäkään ei välttämättä ole aina apua, koska PMX-heuristiikka ei milloinkaan huomioi muita kuin paikallisesti parhaan dominantin täsmäyksen. Tietysti voitaisiin yhtenä ratkaisuna ajatella lisättävän heuristiikan kirjaamia pisteitä yhtä valittua täsmäystä kohti yhdestä kahteen tai jopa useampaankin, mutta tämä heijastuisi väistämättä heikentyneenä suoritusaikana: tarvittavan kirjanpidon määrä lisääntyisi huomattavasti.

Esimerkeille, joissa PMX toimii huonosti, on yhteistä perättäisten täsmäysketjujen kasaantuminen muualle kuin päälävistäjän välittömään läheisyyteen: pahimmassa tapauksessa ne löytyvät matriisin oikeasta ylä- tai vasemmasta alanurkasta. Heuristiikan luotettavuutta voidaan yrittää parantaa tällaisten syötteiden varalta *laskemalla alaraja esiprosessointivaiheessa useaan kertaan* vaihtamalla syötejonoista kulloinkin tarkasteltavia alueita [Ber07]. Sen sijaan *PMX-heuristiikan logiikkaan ei tehdä mitään muutoksia*. Aluksi lasketaan normaaliin tapaan alaraja koko syötejonoille ja se otetaan talteen muuttujaan p_{alar}. Tämän jälkeen syötejonosta *Y poistetaan alkuliite*, jonka pituus on ennalta kiinnitettävän rajaparametrin z suuruinen, ja lasketaan alaraja uudelleen syötejonolle X ja loppuliitteelle $Y[z+1..n]$. Tällä tavoin saadaan PYAn pituudelle mahdollisesti aiempaa tarkempi alaraja, mikäli syötteiden X ja Y PYAn muodostava jono sijaitsee matriisissa päälävistäjän oikealla puolella. Saatua uutta alarajaa $p_{alar}U$ verrataan aikaisemmin laskettuun, ja mikäli uusi alaraja on aikaisemmin löydettyä pidempi, päivitetään muuttujan p_{alar} arvoksi $p_{alar}U$. Tämän jälkeen syötejonon Y alusta leikataan pois z merkkiä lisää, lasketaan seuraava alaraja jonoille X ja $Y[2z+1..n]$ ja päivitetään muuttujan p_{alar} arvoa, mikäli siihen on aihetta. Tällä tavoin jatketaan, kunnes syötejono Y kutistuu tyhjäksi[145]. Viimeksi lasketaan siten alaraja jonoille X ja $Y[qz+1..n]$, missä $q = \lceil n/z \rceil$. Mitä isompi osuus vektorin Y alusta pätkitään pois, sitä lähempänä matriisin oikeaa ylänurkkaa sijaitsevan PYA-jonon näkymältään uudistettu PMX-heuristiikka löytää.

Kun edellä kuvatulla menettelyllä vektorin Y loppuliite on kutistunut tyhjäksi, lähdetään suorittamaan vastaavanlaisia operaatioita syötejonon X erimittaisille loppuliitteille ja koko syötevektorille Y. Aluksi vektorista X poistetaan z merkkiä, sitten $2z$ merkkiä ja niin edelleen, kunnes lopulta vektorin X merkit loppuvat kesken. Jos jokin lasketuista alarajoista osoittautuu jälleen pidemmäksi kuin yksikään aikaisemmista, kirjataan pisin löytynyt alaraja muistiin muuttujaan p_{alar}. Lyhentämällä asteittain X:n loppuliitettä PMX-heuristiikka pystyy pääsemään nyt sellaisten PYA-jonojen jäljille, jotka sijaitsevat lähellä matriisin vasenta alakulmaa.

---

[145] Loppuliitteiden lyhentäminen voidaan lopettaa heti, kun on selvää, ettei alaraja voi enää pidentyä niitä lyhentämällä ($p_{alar} \geq n - kz$), missä k ilmaisee, montako kertaa Y:n loppuliitettä on ehditty lyhentää. Vastaava havainto pätee, kun lyhennetään myöhemmin X:n loppuliitteitä.

Tarkastellaan seuraavaksi aiheeseen liittyen uudelleen esimerkkiä 6.3. Oletetaan, että parametrin *z* arvoksi valittaisiin 6. PYAMAX-heuristiikka laskisi ensiksi alarajan koko syötejonoille, ja tulokseksi saataisiin 1, kuten jo esimerkkiä ensi kertaa tarkasteltaessa todettiin. Seuraavaksi laskettaisiin alaraja syötejonolle *X* ja jonon *Y* loppuliitteelle *Y*[7..11], jolloin löydettäisiin jo tarkka ratkaisu $p = 5$. Lopuksi laskettaisiin vielä alaraja jonoille *X*[7..11] ja *Y*, mutta alaraja ei enää pitenisi. Kannattaa kuitenkin huomioida, että jos *z*:n arvoksi olisi asetettukin 5, ei heuristinen PYA pitenisi yhtään arvosta 1! Tällöin koko jonon *X* ja loppuliitteen *Y*[6..11] alarajaksi olisi saatu edelleen vain 1, sillä PMX pitää täsmäystä (6, 6) parempana kuin täsmäystä (1, 7). Kolmannella kierroksella laskettaisiin alaraja jonon *X* jonon *Y* pelkän viimeisen merkin *Y*[11] kanssa, mutta tämä jätetään tekemättä, sillä alaraja ei enää voisi pidentyä ykkösestä. Neljännellä kierroksella laskettaisiin PYAn alaraja loppuliitteelle *X*[6..11] ja koko *Y*-vektorille, mutta harmia aiheuttava täsmäys (6, 6) osoittautuisi jälleen parhaaksi, ja alaraja ei pitenisi nytkään. Kun lopulta vektorista *X* on jäljellä enää viimeinen merkki *X*[11], alarajan päivittämisyritykset lopetetaan tuloksettomina.

Esimerkin perusteella nähdään, että menetelmä on sangen herkkä rajaparametrin *z* valinnalle: pienikin muutos *z*:n arvossa voi vaikuttaa ratkaisevasti saavutettavan alarajan laatuun. Mitä pienempi arvo *z*:n paikalle asetetaan, sitä suuremmaksi kasvaa toistojen määrä ja samalla heuristiikan kokonaisajoaika. Parametrin arvon pienentäminen parantaa yleensä alarajan laatua, mutta patologisessa tapauksessa ei tästä kuitenkaan ole hyötyä, kuten edellinen esimerkki meille osoitti.

7.3 Alarajaheuristiikkojen tulosten yhdistäminen

Viimeisenä ratkaisuvaihtoehtona alarajan laadun parantamiseksi tarkastellaan *alarajaheuristiikkojen kombinointia*. Tavoitteena on jälleen yrittää mahdollisuuksien mukaan toipua PMX-heuristiikan toisinaan tuottamasta huonosta alarajasta. Kuten olemme nähneet, lääkkeeksi eivät välttämättä kelpaa heuristiikan uudelleenlaskenta kesken algoritmin suorituksen, ylärajan hyödyntäminen alarajan lisäksi eikä myöskään alarajaheuristiikan syötenäkymän siirtäminen.

Halvaksi ratkaisuksi PMX:n laatuongelmalle voisi osoittautua YMF-heuristiikan soveltaminen PMX:n ohella. YMF laskee ainoastaan syötejonojen pareittain yleisimmän merkin minimifrekvenssin, joten laskentaan kuluu aikaa vain niukalti. On siten kokonaisajankäytön kannalta melko lailla yhdentekevää, lasketaanko alaraja yksistään PMX-heuristiikalla vaiko tämän lisäksi myös YMF-heuristiikkaa käyttäen. Valitsemalla näiden kahden heuristiikan palauttamasta alarajasta pidempi saatetaan päästä parempaan tulokseen silloin, kun syöte on PMX-heuristiikan kannalta epäedullinen.

Tarkastellaan vielä kertaalleen esimerkkiä 6.3. Kummassakin syötejonossa merkkiä ”A” ja ”B” esiintyy viidesti, kun taas ”C”:tä vain kerran. YMF-heuristiikka palauttaisi

tällöin alarajaksi arvon 5, kun taas PMX:n laskenta epäonnistuu, ja sen laskeman alarajan pituus on vain 1. PMX- ja YMF heuristiikkojen tulosten yhdistäminen tuottaa käytännössä parhaan tuloksen juuri silloin, kun syötejonoilla on pitkä PYA, joka koostuu vain yhdestä merkistä, mutta jono sijaitsee kaukana jonoista muodostettavan matriisin päälävistäjältä.

Kahden alarajaheuristiikan yhdistämisellä mahdollisesti saavutettavan alarajan laadun paranemista ei tässä työssä ole empiirisesti testattu, kuten ei myöskään aliluvussa 7.2 esiteltyä PYAMAX-heuristiikan syötenäkymän siirtämisen vaikutusta alarajan laatuun. Sen sijaan NKY-algoritmia tehostetaan aliluvussa 8.2.4 sekä ylä- että alarajaheuristiikkaa hyödyntämällä.

8 Hakutietorakenteen valinta ja välitulosten huomiointi

Tässä luvussa on tarkoituksena analysoida, millä tavoin tarkan PYA-algoritmin käyttämän *hakutietorakenteen valinnalla* on vaikutusta menetelmän suoritusaikaan [BHR00, Ber06]. Tarkastelun kohteeksi otetaan ensimmäisessä aliluvussa *riveittäin* prosessoivat menetelmät yleisesti. Jälkimmäisessä aliluvussa puolestaan pohditaan, millä tavoin *diagonaali kerrallaan* prosessoivaa NKY-algoritmia [NKY82] voitaisiin tehostaa järkeistämällä muistinkäyttöä ja kiinnittämällä huomiota sen laskemiin välituloksiin, mitä on selvitetty tutkimusryhmämme artikkelissa [BHV03].

8.1 Hakutietorakenteiden merkitys rivi kerrallaan prosessoivissa algoritmeissa

Kaikille rivi kerrallaan laskentaa suorittaville PYA-menetelmille on yhteistä se, että niiden suorituksen aikana joudutaan systemaattisesti etsimään tarkasteltavalla rivillä i ($1 \leq i \leq m$) sijaitsevan merkin x_i esiintymiä vektorista Y. Asetettava kysymys kuuluu seuraavasti: ”Mistä löytyy merkin x_i seuraava (edellinen) esiintymä Y:stä paikan j jälkeen (ennen indeksiä j)?” Haun perusajatus on sama riippumatta siitä, etsitäänkö pelkästään *dominantteja* (HD2, AG2, RI1, RI2, GCL)[146] vai *kaikkia täsmäyksiä* (HSZ, MUK, KCR). Myöskään riveittäisen prosessoinnin *suunnalla* – vasemmalta oikealle tai päinvastoin – ei ole merkitystä. Erityisesti silloin, kun halutaan ratkaista pelkästään PYAn *pituus*, edellä mainitut hakuoperaatiot tuottavat valtaosan menetelmän laskentavaiheen kustannuksista. Seuraavassa on tarkoitus miettiä, millä tavoin käytettävä hakutietorakenne kannattaisi valita, jotta algoritmi toimisi mahdollisimman tehokkaasti. Asiaa analysoidaan syötteisiin liittyvien erilaisten parametrien osalta. Huomion arvoista on, että tarkasteltaviin algoritmeihin ei samalla olla ehdottamassa mitään muita teknisiä muutoksia, vaan tarkastelun kohteena ovat yksinomaan algoritmeihin sovellettavat hakutietorakenteiden eri vaihtoehdot.

Myöhemmin aliluvussa 9.5 esitellään, miten saman algoritmin eri hakutietorakenteilla varustetut versiot menestyvät testiajoissa, kun niitä kilpailutetaan toisiaan vastaan. Tietorakenteiden osalta on tässä työssä kilpailutettu algoritmia AG2 lähinnä siksi, että Apostolico ja Guerra tarkastelivat AG2:n esittelyn yhteydessä vaihtoehtoisia tapoja haun toteuttamiseksi [Apo87]. Tietorakenteiden tehokkuuseroja on selvitetty myös tutkimusryhmämme aikaisemmissa julkaisuissa algoritmien AG2 [BHR00], NKY [BHV03] ja KCR [Ber06] osalta. Näistä kahdessa viimeksi mainitussa työssä muutamiin algoritmiversioihin on upotettu lisäksi *heuristinen esiprosessointi*,

[146] Näistä menetelmistä RI1 ja GCL suorittavat lisäksi sarakesuuntaista laskentaa, johon riveittäistä prosessointia koskevat havainnot ovat yleistettävissä analogisesti.

jotta nähtäisiin, millainen on tietorakenteiden kehittämisen ja heuristiikkojen soveltamisen kokonaisvaikutus algoritmin tehokkuudelle.

8.1.1 Lineaarihaun kohdistaminen suoraan syötevektoriin *Y*

Algoritmin esiprosessointia ajatellen vähimmällä alustustyöllä päästään, jos syötejonosta *Y ei kerätä minkäänlaisia lisätietoja* ennen varsinaisen laskentavaiheen käynnistämistä. Tämä tarkoittaa sitä, että rivi kerrallaan prosessoiva menetelmä etsisi merkille x_i täsmäyksiä vuoron perään *jokaisesta Y:n positiosta*. Tällöin riveittäinen PYA-algoritmi palautuisi hyvin lähelle *Wagnerin ja Fischerin* naiivia algoritmia. Tärkeimpänä poikkeuksena WFI:hin nähden olisi kuitenkin se, etteivät täsmäämättömät merkkiparit aiheuttaisi minkäänlaista kirjanpitoa, kun taas WFI kirjaa muistiin jokaisen alkuliiteparin $X[1..i]$ ja $Y[1..j]$ ($1 \leq i \leq m$, $1 \leq j \leq n$) välisen PYAn pituuden, vaikkei indeksiparissa (i, j) olisikaan täsmäystä.

Suoran lineaarihaun soveltaminen pidempään syötevektoriin johtaisi täten aikavaativuuteen $\Omega(mn)$, mikäli hakualuetta vektorista *Y* ei rajoiteta millään tavalla. Yksinkertaisen toteutuksensa ansiosta lineaarihaku on käytännössä käyttökelpoinen lyhyillä syötejonoilla, jolloin vältetään esiprosessoinnista aiheutuneet kustannukset. Lisäksi, jos käytettävä algoritmi kirjaa kaikki täsmäykset, ja niitä sattuu olemaan tarkasteltavassa ongelman instanssissa tiheälti, ei hakua tukevia tietorakenteita käyttämällä juurikaan selvittäisi nopeammin, sillä vektorissa *Y* päästään tällöin etenemään vain hyvin lyhyitä askeleita kerrallaan, ja aputietorakenteiden tarvitsemien lista- tai vektorioperaatioiden kutsuminen vie helposti vähintään saman ajan, jonka kuluessa suoraa lineaarihakua soveltamalla etsittävä kohde olisi jo löytynyt. Sen sijaan jos syötteet ovat pitkiä, ongelmassa esiintyy vain harvoja täsmäyksiä, tai ollaan kiinnostuneita vain dominanteista täsmäyksistä, kostautuu esiprosessoinnin pois jättäminen väistämättä huomattavasti pidentyneinä hakuaikoina.

8.1.2 Syötejonosta *Y* rakennettu esiintymälista

Teknisesti yksinkertaisin tietorakenne merkin x_i täsmäyskohtien kirjaamiseksi vektorissa *Y* on *esiintymä-* eli *täsmäyslista*[147], joka kootaan erikseen jokaista[148] syöttöaakkoston symbolia kohti. Kyseessä on yhteen suuntaan linkitetty, joko nousevaan tai laskevaan suuruusjärjestykseen lajiteltu lista, jonne on ennen laskentavaiheen käynnistymistä viety kaikkien syöttöaakkoston merkkien

[147] kts. aliluku 2.4.1

[148] Periaatteessa riittäisi muodostaa täsmäyslistat ainoastaan niille syöttöaakkoston symboleille, joita esiintyy vektorissa *X*. Tämä edellyttäisi kuitenkin ennakkotietoa *X*:ssä esiintyvistä merkeistä.

esiintymiskohdat Y:ssä. Listan muodostaminen edellyttää vektorin Y selaamista kertaalleen jompaankumpaan suuntaan, joten se saadaan muodostettua ajassa $O(n)$, eli sen rakentaminen ei vaikuta algoritmin asymptoottiseen aikakompleksisuuteen. Selaus tehdään päinvastaiseen suuntaan algoritmin riveittäisen laskennan etenemissuuntaan nähden, jotta merkin s_i ($1 \leq i \leq \sigma$) uusin löydetty esiintymäkohta voitaisiin lisätä listaan aina ensimmäiseksi. Näin saadaan selauksen päätyttyä aikaan haluttuun järjestykseen lajiteltu merkkikohtainen täsmäyslista.

Kaikki luvussa 3.2 esitellyt riveittäiset PYA-algoritmit etenevät rivejä pitkin yksinomaan vasemmalle tai oikealle: milloinkaan ei jouduta perääntymään rivin tarkastelun ollessa kesken. Siten yhteen suuntaan linkitetty rakenne on riittävä hakuoperaatioita ajatellen, eli listaa ei saman rivin käsittelyn aikana jouduta kelaamaan alusta loppuun kuin korkeintaan kertaalleen. Rivin käsittelyn tultua valmiiksi siirretään tutkitun merkin esiintymälistan osoitin takaisin listan alkuun. Esiintymälistaa käyttämällä voidaan selvästikin ohittaa riviltä kaikki ne Y-indeksit, joissa ei esiinny täsmäystä. Pahimmassa tapauksessa joudutaan nyt tutkimaan yhteensä r indeksiparia, missä r on täsmäysten kokonaismäärä ongelmassa.

Jos tarkasteltavassa ongelmassa on täsmäyksiä tiheässä – näin käy usein syöttöaakkoston ollessa pieni – ei täsmäyslistoja pitkin pystytä etenemään paljoakaan nopeammin kuin kohdistamalla lineaarihaku suoraan syötevektoriin Y. Jos kuitenkin täsmäykset ovat harvassa, esiintymälistaa käyttämällä selvitään hakuoperaatioista huomattavasti vähemmällä työllä kuin lineaarihakua soveltamalla. Jos ollaan etsimässä pelkästään dominantteja täsmäyksiä, saatetaan joutua kuitenkin testaamaan useita perättäisiä täsmäyslistan indeksejä, ennen kuin kriteerit täyttävä täsmäys lopulta löytyy, jos sellainen ylipäätään on vielä löytyäkseen.

8.1.3 Vektorimuotoinen esiintymälista syötejonosta Y

Kolmantena varteenotettavana vaihtoehtona hakutietorakenteeksi voidaan mainita syötejonoon Y perustuva *vektorimuotoinen esiintymälista*. Sillä tarkoitetaan esiintymälistaa, joka on tallennettu dynaamisen tietorakenteen sijasta staattiseen vektoriin.

Kuten edellä todettiin, rivi kerrallaan laskentaa suorittavissa PYA-algoritmeissa ei riviä prosessoitaessa jouduta perääntymään kertaakaan. Täten vektorista ei nyt ole hyötyä siinä mielessä, että se tukee hakusuunnan vaihtamista kesken kaiken. Vektorin hyödyt ilmenevät kuitenkin silloin, kun etsitään *dominanttia k-täsmäystä*, jota edeltää pitkä ei-dominanttien täsmäysten ketju. Jos esiintymälista olisi tällöin perustettu yhteen suuntaan linkitettyä listarakennetta käyttämällä, ei olisi tarjolla mitään parempaa ratkaisua dominantin täsmäyksen etsimiseksi kuin selata täsmäyslistaa lineaarihaulla

riittävän pitkään[149]. Vektoriesitys sen sijaan tarjoaa mahdollisuuden *puolitushakuun*, kunhan tiedetään indeksialue, jolta dominanttia täsmäystä ollaan hakemassa.

Kannattaa huomioida, että vektorimuotoisen esiintymälistan rakentaminen on työläämpää kuin linkitetyn listan, sillä ennen sen kokoamista pitäisi kerätä tiedot siitä, miten monta kertaa mikin syöttöaakkoston symboleista esiintyy syötejonossa *Y*, jotta sen muistinvaraus voitaisiin suorittaa taloudellisesti – esimerkiksi sijoittamalla kaikkien merkkien esiintymälistat yhdeksi pötköksi peräkkäin. Lisäksi tarvitaan erillinen vektori, joka osoittaa kunkin merkin esiintymälistan alkukohdan vektorissa, jotta haku osattaisiin kohdistaa oikeaan vektorin osaan. Vektorimuotoisen esiintymälistan rakennusvaiheen asymptoottinen kustannus on tosin vain $O(n)$, mutta sen konstruoiminen vaikuttaa intuitiivisesti kuitenkin sen verran monimutkaiselta, että pitkän päälle otaksuisi yksisuuntaisen linkitetyn rakenteen olevan tehokkaampi ja suoraviivaisempi valinta haun aputietorakenteeksi.

8.1.4 Syötejonoon *Y* perustuva lähiesiintymätaulukko

Lähiesiintymätaulukon[150] [Apo87] soluun *LähiEsTaulu*[$s_{X[i]}$, *j*] on talletettu rivillä *i* ($1 \leq i \leq m$) sijaitsevan symbolin ensimmäinen esiintymä vektorissa *Y* paikan *j* ($1 \leq j \leq n$) jälkeen. Toisin sanoen lähiesiintymätaulukosta löydetään *vakioajassa* merkin x_i esiintymä mielivaltaisen indeksin jälkeen vektorista *Y*, eli taulukko tukee *suorasaantia*. Mitä useammin hakuoperaatioita joudutaan PYA-ongelman laskentavaiheen aikana suorittamaan, sitä suurempi hyöty saavutetaan käyttämällä suorasaannin mahdollistavaa aputietorakennetta.

Ongelmaksi koituu kuitenkin taulukon *perustamiskustannus*, joka riippuu tulotermistä σn. Mitä suurempi syöttöaakkoston koko on, sitä kalliimmaksi alustusvaihe muodostuu. Jos kuitenkin aakkoston koon tiedetään olevan pieni ja syötejonot ovat pitkiä, on lähiesiintymätaulukon perustaminen intuitiivisesti mielekästä. Tämä pätee erityisesti silloin, kun algoritmi etsii ainoastaan dominantteja täsmäyksiä, sillä pienellä syöttöaakkostolla täsmäysten määrä on verrattain iso, joten kaikkien tarpeettomien täsmäysten ohittaminen säästää tuntuvasti ajoaikaa. Aakkoston koon kasvaessa lähiesiintymätaulukon käyttökelpoisuus kuitenkin heikkenee asteittain. Lisäksi on huomioitava, ettei lauseke $\Theta(\sigma n)$ kuvaa vain lähiesiintymätaulukon alustamisen kustannusta vaan myös sen tarvitsemaa muistitilan määrää; vaikka verrattain pitkää alustusaikaa pystyttäisiinkin vielä sietämään, liiallinen muistinkulutus saattaa jo aikaisemmin muodostua pullonkaulaksi rakenteen käyttämiselle.

---

[149] Periaatteessa on kuitenkin mahdollista kuorruttaa lista monikerroksisilla osoittimilla ns. *skip-listaksi* [Rai96], jonka avulla pystytään jäljittelemään puolitushakua linkitetyille listoille.

[150] kts. aliluku 2.4.2

8.1.5 Lähiesiintymävektorin konstruoiminen syötejonosta *Y*

Viimeisenä hakua nopeuttavana tietorakenteena tarkastellaan aliluvussa 2.4.3 esiteltyä *lähiesiintymävektoria*. Sitä voidaan pitää lähiesiintymätaulukon tiivistettynä versiona, sillä se on saatu aikaan puristamalla lähiesiintymätaulukko 1-ulotteiseksi vektoriksi, jonka pituus on n+1. Kokoon puristamisen myötä vektorin kutakin yksittäistä indeksipaikkaa j kohti on saatavilla tieto ainoastaan *yhden aakkoston symbolin lähimmästä esiintymästä paikan j jälkeen*. Ellei paikkaan *LähiEsVekt*[j] tallennettu informaatio merkin lähimmästä sijaintipaikasta indeksin j jälkeen koske juuri rivillä i esiintyvää symbolia x_i, joudutaan lähiesiintymävektorissa peruuttamaan lähimpään indeksipaikkaan j', johon on tallennettu merkin x_i lähin esiintymä paikasta j' eteenpäin. Jos nyt h = *LähiEsVekt*[j'] > j, etsitty indeksipaikka on löytynyt. Muutoin joudutaan vielä hakemaan puolitushakua käyttäen x_i:n vektorimuotoisesta[151] esiintymälistasta vasemmanpuoleisin esiintymä, jonka indeksi > j.

Lähiesiintymävektorin ja sitä täydentämään tarvittavan vektorimuotoisen esiintymälistan rakentamiskustannus ja tilantarve ovat lineaarisia pidemmän syötejonon pituuden suhteen eli suuruusluokkaa $O(n)$. Siten lähiesiintymävektoria voidaan käyttää hakutietorakenteena hyvinkin pitkille syötejonoille. Erityisesti silloin, kun täsmäyksiä on paljon ja algoritmi rekisteröi pelkästään dominantteja täsmäyksiä, päästään lähiesiintymävektoria käyttämällä ohittamaan useat tarpeettomat täsmäykset. Toisaalta, koska tietorakenteesta haun kustannusta kuvaa termi $\log \sigma$, joudutaan algoritmien, joiden asymptoottinen suorituskustannus riippuu dominanttien täsmäysten lukumäärästä d, ajoaika kertomaan kyseisellä termillä. Tämä antaa aihetta olettaa, että lähiesiintymätaulukon korvaaminen lähiesiintymävektorilla pidentää merkittävästi algoritmin laskentavaiheen suoritusaikaa, kun taas esiprosessointiin kuluva aika sekä muistinkulutus vähenevät jyrkästi. Lähiesiintymävektori saattaa siten olla oikea valinta hakutietorakenteeksi silloin, kun syöttöaakkosto on iso tai täsmäyksiä on runsaasti suhteessa dominanttien täsmäysten lukumäärään. Sen sijaan menetelmille, jotka rekisteröivät kaikki täsmäykset, lähiesiintymävektori vaikuttaa huonolta valinnalta: usein toistuvat hakuoperaatiot tulevat pitkän päälle kalliimmiksi kuin muita hakutietorakenteita käyttämällä.

8.2 Nakatsun, Kambayashin ja Yajiman algoritmin tehostaminen

Nakatsun, Kambayashin ja *Yajiman* NKY-algoritmia [NKY82] tarkasteltiin edellä aliluvussa 3.3.1, jossa esiteltiin ja kuvattiin algoritmin perusominaisuudet sekä analysoitiin sen aika- ja tilavaativuutta. Algoritmiin palattiin uudelleen aliluvussa 6.1,

[151] Lähiesiintymävektorin käyttäminen edellyttää myös vektorimuotoisten esiintymälistojen perustamista, mikäli puolitushakua halutaan soveltaa.

jossa pohdittiin, miten tietoa heuristisesta ylä- ja alarajasta voitaisiin käyttää algoritmissa hyväksi. Tuolloin ilmeni, että heuristisen rajojen avulla voidaan NKY:n muistintarvetta pienentää melkoisesti, mikäli sen tietorakenteet uudelleenorganisoidaan. Tarkoituksena onkin seuraavassa selvittää, millä tavoin NKY:tä voitaisiin nopeuttaa ohjelmointiteknisin keinoin.

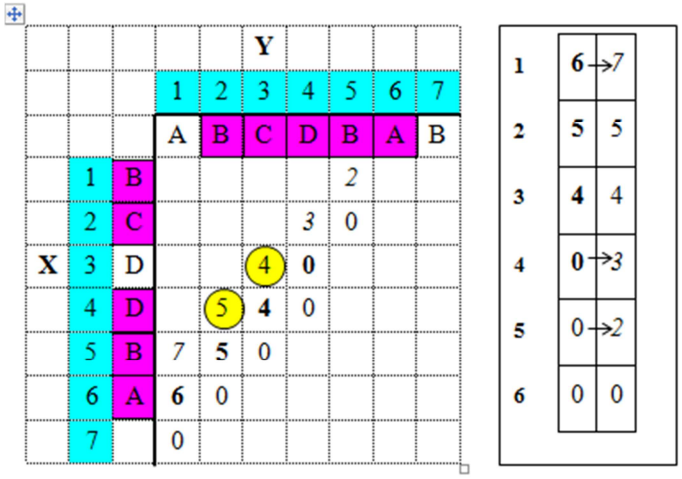
Ennen kuin käytetään enempää tarmoa NKY-algoritmin tarkasteluun, olisi syytä perustella, minkä tähden juuri kyseiseen algoritmiin kannattaa kiinnittää erityistä lisähuomiota. NKY on algoritmi, jossa on yleiskäyttöisyyden kannalta monia suotuisia piirteitä. Ensinnäkin se on toteutukseltaan verrattain yksinkertainen, eli sen käyttöönotto ei vaadi monimutkaista ohjelmointityötä. Toiseksi, algoritmin esiprosessointivaihe on minimaalisen niukka, eli se soveltuu käytettäväksi myös lyhyillä syötteillä, joilla pitkä esiprosessointi veisi suhteettoman suuren osuuden kokonaisajankäytöstä. Kolmanneksi, algoritmin teoreettinen suoritusaika ei ole riippuvainen sen kummemmin (dominanttien) täsmäysten lukumäärästä kuin syöttöaakkoston koostakaan, joille parametreille monet PYA-algoritmeista ovat herkkiä. NKY:n suoritusaikaan vaikuttaa ratkaisevasti vain PYAn pituuden suhteellinen osuus lyhyemmän syötejonon pituudesta eli osamäärä p/m: mitä lähempänä ykköstä osamäärä on, sitä nopeammin NKY selviytyy tehtävästään. Useissa tarkasteltavissa ongelmissa kuten versionhallinnassa ja biosekvenssien vertailussa PYAn voidaan ennalta olettaa olevan pitkä, joten tässäkin mielessä NKY kestää hyvin vertailun.

Algoritmilla todettiin olevan kuitenkin yksi kiusallinen pullonkaula: suuri muistinkulutus. Kyseinenkin ongelma aktualisoituu pelkästään silloin, kun halutaan palauttaa myös jokin PYAn ratkaisuksi kelpaava jono. Pelkästään PYAn pituuden määräämiseksi selviydytään lineaarisella muistitilalla, kuten KRA-algoritmissa [Kum87]! Jotta voisimme vakuuttua asiasta, tarkastellaan esimerkkiä 8.1.

Esimerkin 8.1 mukaisille syötejonoille NKY-algoritmi laskee ulomman silmukan ensimmäisen kierroksen aikana järjestyksessä arvot $L_6(1) = 6$, $L_5(2) = 5$ ja $L_4(3) = 4$, jotka tallennetaan matriisin soluihin (6, 1), (5, 2) ja (4, 3)[152]. Seuraavaksi laskettavasta arvosta $L_3(4)$ tulee määrittelemätön, sillä merkkiä ”D” ei enää esiinny alkuliitteessä $Y[1..3]$. Ensimmäisen kierroksen aikana lasketut $L_i(k)$-arvot on merkitty matriisiin **lihavoiduilla** ja toisen kierroksen aikana lasketuista $L_{i-1}(k)$-arvoista ne, jotka ovat suurempia kuin vastaavat $L_i(k)$-arvot, *kursivoiduilla* numeroilla. Kun toinen ulomman silmukan kierros käynnistyy laskemalla $L_5(1) = 7$, NKY tallentaa sen paikkaan (5, 1). Paikkaan (6, 1) edellisellä kierroksella vietyä arvoa $L_6(1) = 6$ ei saada kirjoittaa yli, jos halutaan ylläpitää myös *PYAn ratkaisupolkua*. Ei ole nimittäin minkäänlaisia takeita siitä, että täsmäys (5, 7) olisi parempi valinta PYA-jonoon mukaan otettavaksi kuin kierros aikaisemmin löytynyt (6, 6), jota ei yli kirjoittamisen tapahduttua enää muistettaisi.

[152] Yleisesti, NKY tallentaa laskemansa $L_i(k)$-arvon matriisin soluun $M[i, k]$.

Esimerkki 8.1: *Pelkän PYAn pituuden ratkaiseminen NKY-algoritmilla, kun p = 5, X* = ”BCDDBA”, *Y* = ”ABCDBAB” *ja PYA* = “BCDBA”.



Sen sijaan, jos olisimme kiinnostuneita *vain PYAn pituudesta p*, meille riittäisi arvoa $L_i(k)$ laskettaessa pelkästään tieto, että *k*:n mittaisen PYAn muodostaminen *X*[*i..m*]:n ja vektorin *Y* loppuliitteen kanssa on *jollain tavalla mahdollista*. Mitä lyhyempi *Y*:n loppuliite sen muodostamiseksi tarvitaan, sitä parempi. Se, paljonko tähän tarkoitukseen *tarkalleen* tarvitaan merkkejä vektorin *X* lopusta ja *mitkä X:n loppuliitteen merkit* siihen osallistuvat, on tässä yksinkertaisemmassa ongelmanasettelussa samantekevää. Niinpä paikkaan (5, 1) tallennettu arvo 7 voitaisiin pelkästään *p*:tä määrättäessä asettaa nyt paikkaan (6, 1) viedyn arvon 6 tilalle, koska nyt jo pelkkä *Y*:n viimeinen merkki riittää yhden mittaisen PYAn muodostamiseksi *X*[5..6]:n kanssa. Seuraavaksi laskettavat $L_4(2) = 5$ ja $L_3(3) = 4$ ovat arvoiltaan samat kuin kierrosta aikaisemmin lasketut $L_5(2)$ ja $L_4(3)$. Sen sijaan osoittautuu, että $L_2(4) = 3 > L_3(4) = 0$ sekä $L_1(5) = 2 > L_2(5) = 0$, joten aikaisemmat nollat indeksipareissa (3, 4) ja (2, 5) päivittyisivät arvoiksi 3 ja 2. Toisen kierroksen päätyttyä paikasta (6, 1) alkava diagonaali sisältäisi *p*:tä määrättäessä siten arvot 7, 5, 4, 3 ja 2. Näistä järjestyksessä *i*. ($1 \leq i \leq 5$) arvo kertoo nyt, miten pitkä *Y*:n loppuliite tarvitaan *i* mittaisen PYAn muodostamiseksi vektorin *X* loppuliitteen kanssa, jonka pituus on *i*+1. Sen sijaan jonosta 7, 5, 4, 3, 2 ei pystytä enää päättelemään, mitkä merkit PYAn muodostavat.

Algoritmin suoritus päättyisi edellisessä esimerkissä ulomman silmukan toisen kierroksen jälkeen, sillä paikasta (4, 1) alkava 3. diagonaali on jo liian lyhyt, jotta PYA voisi vielä pidentyä arvosta 5. Koska arvo $L_i(k)$ voidaan PYAn pituutta määrättäessä unohtaa edellisen tarkastelun perusteella heti, kun arvo $L_{i-1}(k)$ on laskettu, riittäisi pelkästään yksi pituudeltaan *m* oleva vektori *Y*:n optimaalisten loppuliitteiden pituuksien kirjaamiseksi: matriisi on siis tarpeeton pelkän *p*:n laskennassa! Vektori pitää ennen laskennan käynnistämistä alustaa nollilla samasta syystä, kuin niitä alkuperäisessä algoritmissa asetetaan tarkasteltavan diagonaalin oikealle puolelle, kun PYA voi pidentyä aiemmasta arvostaan. Sen sisältö ulomman silmukan 1. ja 2. kierroksen jälkeen on merkitty näkyville esimerkin 8.1 oikeanpuoleiseen kuvaan.

Yleisesti, kun ulomman silmukan *k*. kierros ($1 \le k \le m$) on saatu päätökseen, on vektorin *i*. ($1 \le i \le m$) indeksiin tallennettu tieto siitä, miten pitkä on *Y*:n lyhin loppuliite, joka muodostaa loppuliitteen $X[m+1-k..m]$ kanssa *i*:n mittaisen PYAn.

Olemme siis todenneet, että NKY toimii muistitilassa $O(n)$, jos PYAn mahdollisia ratkaisupolkuja ei tarvitse ylläpitää. Seuraavassa palataan kuitenkin tarkastelemaan tilannetta, että myös jokin PYAn ratkaisuksi kelpaavista jonoista pitää palauttaa. Seuraavissa aliluvuissa selvitetään tärkeimpiä ohjelmointiteknisiä[153] ratkaisuja sille, miten alkuperäistä NKY-algoritmia voidaan tehostaa [BHV03, 294–298].

8.2.1 Matriisin uudelleenorganisointi

Tarkasteltaessa NKY-algoritmin pseudokoodia liitteen kohdassa 11.3.1 havaitaan, että algoritmin aputietorakenteena käytettävä matriisi *M* on indeksoitu syötejonojen mukaan järjestyksessä (*Xindeksi*, *Yindeksi*). Tallennettaessa *Y*:n optimaalisten loppuliitteiden arvoja matriisiin sitä täytetään kuitenkin aina *diagonaali* kerrallaan, sillä arvon $L_i(k)$ jälkeen lasketaan seuraavaksi aina $L_{i-1}(k+1)$ edellyttäen, että arvo $L_i(k)$ on määritelty. Tämä tarkoittaa sitä, että jokaista arvoa tallennettaessa joudutaan matriisissa siirtymään pois *sekä nykyiseltä riviltä että sarakkeelta*. Tämä on helposti havaittavissa esimerkistä 8.1, jossa ensimmäisen diagonaalin arvot 6, 5, 4 ja 0 sijaitsevat kukin eri rivillä ja sarakkeella.

Jos matriisi indeksoitaisiin sen sijaan *diagonaaleittain* siten, että *jokainen diagonaali muodostaisi yhden matriisin sarakkeen*, samalle diagonaalille kuuluvien arvojen tallentamista varten ei tarvitsisi siirtyä sarakkeelta pois kertaakaan. Uudelle sarakkeelle siirrytään vasta silloin, kun seuraavan diagonaalin käsittely alkaa. Indeksointi toteutettaisiin siis siten, että matriisin rivinumero — $L_i(k)$-arvon parametri *k* — kertoo *paraikaa etsittävän loppuliitteiden PYAn pituuden* ja sarakenumero *diagonaalin järjestysnumeron*, jota esittää lauseke $m + 2 - i - k$. Matriisin vasen sarake edustaa siis ensimmäistä eli pisintä, syötejonojen indekseillä kuvattuna paikasta (*m*, 1) alkavaa diagonaalia. Uudessa esitysmuodossa ei erillistä nollatta diagonaalia tarvita, jos matriisi alustetaan nollilla, mikä on useiden ohjelmointikielten tukemana vakioaikainen operaatio [Meh84, luku III]. Kuten NKY:n pseudokoodista käy ilmi, arvon $L_i(k)$ laskennassa tarvitaan avuksi edellisen diagonaalin arvoa $L_{i+1}(k)$. Tämä arvo löytyisi uudessa matriisiesityksessä suoraan tarkasteltavan solun vasemmalta puolelta, eli tämänkään takia ei tarvitsisi siirtyä matriisissa pois samalla sekä nykyriviltä että -sarakkeelta. Jos uutta indeksointitapaa käytettäisiin esimerkissä 8.1, löydettäisiin toisen

[153] Viitteenä olevassa artikkelissa on alkuperäisestä NKY:stä lisäksi poistettu muutama turha muuttujan päivitys tiukentamalla silmukkaehtoja sekä erottelemalla pääsilmukka kahteen loogisesti autonomiseen toimintavaiheeseen: aiemmin löydettyjen $L_i(k)$-arvojen kasvattamiseen sekä PYAn pitenemiseen aikaisemmasta arvostaan. Näiden tarkempi kuvaus kuitenkin sivuutetaan tässä.

diagonaalin soluun (1, 2) tallennettavan arvon $L_5(1) = 7$ laskennassa apuna tarvittava $L_6(1) = 6$ ensimmäisen diagonaalin solusta (1, 1).

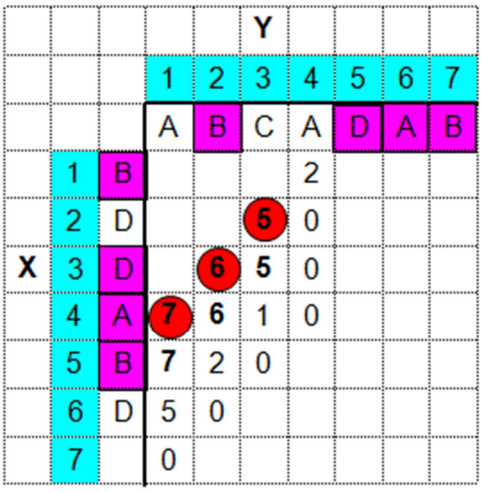
Matriisin uudelleenorganisointi saattaisi ensivaikutelmalta tuntua melko kosmeettiselta ja vähäiseltä muutokselta. Koska pitkillä syötejonoilla matriisiin tallennus- ja sieltä hakuoperaatioita tehdään kuitenkin runsaasti, pienikin muistinkäytön järkeistäminen voi tuoda suuria säästöjä ajoaikaan. Mitä paikallisempia suoritettavat muistioperaatiot ovat, sitä tehokkaammin erityisesti välimuisti toimii. NKY:n perinteisen indeksointitavan ja matriisin uudelleenorganisoinnin välisten tehokkuusvertailujen empiirisiä testituloksia esitellään aliluvussa 9.4.1.

8.2.2 Optimaalisten loppuliitteiden havaitseminen syötejonossa *Y*

Toisinaan saattaa syötejonosta *Y* löytyä *pitkä loppuliite*, joka esiintyy kokonaisuudessaan alijonona syötejonossa *X*. Käytettäessä NKY-algoritmin mukaista laskentastrategiaa tämän pystyisi havaitsemaan siitä, että jostakin *i*:n arvosta lähtien perättäisille *k*:n arvoille 1, 2, .., *z* on voimassa $L_i(1) = n$, $L_i(2) = n–1$, $L_i(3) = n–2$, …, $L_i(z) = n–z+1$. Tällöin on selvää, että *Y*:n loppuliitteitä, joiden pituus on väliltä 1..*z*, ei enää pystytä valitsemaan tehokkaammin. Siten kaikilla seuraavilla diagonaaleilla olisi jo turhaa laskea arvoja $L_{i–u}(1)$, $L_{i–u–1}(2)$, … $L_{i–u–2}(3)$, …, $L_{i–u–z+1}(z)$, kun $u > 1$. NKY:tä ei kuitenkaan ole rakennettu tunnistamaan *Y*:n optimaalisten loppuliitteiden löytymistä, vaan se jatkaa sitkeästi $L_i(k)$-arvojen laskentaa myös *k*:n arvoille 1..*z* kullakin diagonaalilla aina suorituksensa päättymiseen asti. Mitä pidempi optimaalinen loppuliite *Y*:stä löydetään – ts. mitä suuremmaksi *z* kasvaa – sitä enemmän NKY suorittaa hyödytöntä laskentaa. Tarkastellaan asian selventämiseksi esimerkkiä 8.2.

Kun nykyisen diagonaalin *d* läpikäynnin päätyttyä tiedetään *Y*:stä löytyneen *z*:n pituisen optimaalisen loppuliitteen, voidaan diagonaalille *d*+1 saavuttaessa siirtää syötevektorin *Y* kursoria viimeisestä positiosta *n* yhteensä *z merkkiä alkua kohti*. Vastaava menettely pätee syötevektorille *X*: kun sen tarkastelu diagonaalilla *d* aloitettaisiin normaalisti positiosta *m*+1–*d*, aloituskohtaa siirretään nyt samoin *z* merkkiä alkuun päin paikkaan *X*[*m*+1–*d*–*z*] asti, ja ensimmäinen diagonaalilla laskettava $L_i(k)$ arvo on vastaavasti $L_{m+1–d–z}(z+1)$. Mitä varhaisemmassa vaiheessa prosessoinnin aikana *z*:n arvo kasvaa, sitä pidempiä osuuksia vektoreista *X* ja *Y* voidaan jättää algoritmin ulomman silmukan myöhemmillä kierroksilla tarkastelematta. Kannattaa lisäksi huomioida, että optimaalisten loppuliitteiden etsiminen on täysin tuloksetonta ainoastaan silloin, kun syötevektorin viimeinen merkki y_n ei täsmää yhteenkään *X*:n merkkiin viimeistä edelliseen ulomman silmukan kierrokseen mennessä. Vain kyseisessä tapauksessa ei vektoreiden lopusta pystytä ohittamaan yhtään ylimääräistä merkkiä koko laskentavaiheen aikana.

Esimerkki 8.2: *Optimaalisen loppuliitteen löytymien syötevektorista Y, kun p = 4, X* = ”BDDABD”, *Y* = ”ABCADAB” *ja PYA* = “BDAB”.



Diagonaalilta 2 löytyy kolmen mittainen *Y*:n optimaalinen loppuliite, joka esiintyy *X*:ssä alijonona. Kolmannen diagonaalin arvot $L_4(1)$, $L_3(2)$ ja $L_2(3)$, jotka on merkitty kuvaan **punaisin** pallukoin, voitaisiin jättää laskematta, koska pituudeltaan enintään kolmen mittaisen PYAn muodostamiseen tarvittavaa *Y*:n loppuliitettä ei enää voida kutistaa lyhyemmäksi.

8.2.3 Yhtenäisten täsmäysketjujen havaitseminen syötejonossa *Y*

Melko lailla *Y*:n optimaalisen loppuliitteen löytymistä muistuttava tilanne voi kehkeytyä myös toisaalla vektorissa *Y*. Oltaessa prosessoimassa diagonaalia *d* ja laskemassa arvoa $L_i(k)$ voi syntyä tilanne, että $L_i(k) = L_{i+1}(k)$, mikä tarkoittaa sitä, että *k*:n mittaisen PYAn löytymiseksi vaadittavan *Y*:n loppuliitteen pituus ei lyhene, vaikka *X*:n loppuliite pitenee yhdellä. Tällöin joudutaan suorittamaan arvon $L_{i+1}(k)$ kopiointi edelliseltä diagonaalilta *d*–1 nykyiselle diagonaalille. Jos nyt diagonaalilla *d*–1 arvon $L_{i+1}(k)$ sijaintipaikasta alkaa *z*:n *perättäisen täsmäyksen ketju*, eli diagonaalin perättäiset arvot $L_{i+1}(k)$, $L_i(k+1)$, $L_{i-1}(k+2)$, …, $L_{i-z+1}(k+z-1)$ eroavat toisistaan ainoastaan *ykkösen verran*, siitä seuraa, ettei myöskään nykyisellä diagonaalilla *d* mikään arvoista $L_{i-1}(k+1)$, $L_{i-2}(k+2)$, …, $L_{i-z}(k+z-1)$ voi muuttua $L_i(k)$-arvojen määritelmän nojalla[154]. Tällöin kummastakin syötevektorista voidaan ohittaa seuraavat *z* merkkiä tarpeettomina.

Palataan asian selventämiseksi vielä uudelleen esimerkkiin 8.1. Edettäessä toista diagonaalia pitkin asetetaan aluksi $L_5(1) = 7$. Tämän jälkeen $L_4(2)$:n arvoksi joudutaan asettamaan edellisen diagonaalin $L_5(2) = 5$, sillä merkkiä x_4 = ”D” ei löydy paikasta 6, ja $L_i(k)$-arvojen määritelmän mukaisesti pitää olla voimassa $L_5(1) > L_4(2) \geq L_5(2)$. Solusta (5, 2) alkaa diagonaalilla 1 kahden perättäisen täsmäyksen ketju, joka päättyy soluun (4, 3) ja koostuu arvoista $L_5(2)$ ja $L_4(3)$. Täten diagonaalilla 2 arvojen $L_4(2)$ ja $L_3(3)$ on oltava viimeksi mainittujen kanssa samat ja ne voidaan ohittaa. Kyseiset arvot on merkitty kuvaan **keltaisin** palloin. Diagonaalilla 2 siirryttäisiin siten seuraavaksi etsimään arvoa $L_2(4)$, joksi määräytyisi 3.

Pitkän yhtenäisen täsmäysjakson löytymisen testaaminen ei saa aikaan niin suuria ajoaikasäästöjä kuin optimaalisten loppuliitteiden havaitseminen, sillä kunkin

[154] kts. lemmat 3.3 – 3.5

diagonaalijakson pituuden kirjaaminen muistiin tulee hankalaksi. Niinpä onkin selvitettävä testaamalla toistuvasti, eroavatko edellisen diagonaalin kaksi perättäistä arvoa $L_{i+1}(k)$ ja $L_i(k+1)$ toisistaan vain yhdellä, jos arvo $L_{i+1}(k)$ on jouduttu kopioimaan seuraavalle diagonaalille $L_i(k)$:n arvoksi. Koneissa, joissa muistipaikan arvon testaaminen on alkeisoperaationa halvempi kuin muuttujan arvon asettaminen, yhtenäisten täsmäysjaksojen huomioiminen lyhentää jonkin verran ajoaikaa, jos syötejonojen PYA on pitkä.

8.2.4 Heuristisen rajojen vaikutus NKY:n muistinkäyttöön

Edellisissä kolmessa aliluvussa kuvailtiin tekniikoita, miten NKY:n suoritusta voitaisiin tehostaa esittelemällä algoritmin laskentatapaa paremmin tukeva indeksointi sekä vähentämällä tarpeetonta laskentaa *Y*:n optimaalisten loppuliitteiden ja pitkien täsmäysjaksojen tunnistamisen ansiosta. Nämä muutokset NKY-algoritmiin eivät kuitenkaan vielä ratkaisseet sen pahinta puutetta eli *suurta muistintarvetta*, jos myös PYA-jono pitää ratkaista. Seuraavassa tarkastellaan, miten NKY:n muistinkulutusta voitaisiin pienentää *heuristisen ylä- ja alarajojen* avulla [BHV03]. Asiaa ehdittiin jo lyhyesti sivuta aliluvussa 6.1.

Kuten aliluvusta 3.3.1 muistetaan, alkuperäinen NKY varaa aina käyttöönsä kooltaan O(*mn*) olevan matriisin aputietorakenteekseen [NKY82]. Koska algoritmin ulointa silmukkaa suoritetaan kuitenkin enintään *m* kierrosta, ja jokaisella kierroksella diagonaalien pituus lyhenee yhdellä, teoriassakin vain puolet matriisin muistitilasta riittäisi. Ongelmana on kuitenkin, miten matriisin muistinvaraus kannattaisi suorittaa.

PYAn *heuristinen yläraja* antaa meille tiedon siitä, miten pitkä syötejonojen PYA *korkeintaan* on. Selvästikään yhdenkään diagonaalin ei tarvitsisi olla ylärajaa pidempi, jotta sille mahtuisi riittävästi nollasta eroavia $L_i(k)$-arvoja. Vastaavasti *alaraja* osoittaa, miten pitkä PYA *vähintään* on. Siitä pystytään päättelemään, miten monta diagonaalia korkeintaan joudutaan käsittelemään, ennen kuin ratkaisu löytyy ja algoritmi lopettaa toimintansa. Diagonaalille, jonka järjestysnumero on d, mahtuu yhteensä $m+1-d$ kappaletta $L_i(k)$-arvoja. Olettaen, että aputietorakenne alustetaan nollilla, eli erillistä nollatta diagonaalia ei tarvita, on diagonaalien $1..m+1-p_{ylär}$ oltava pituudeltaan heuristisen ylärajan mittaisia. Tämän jälkeen jokainen seuraava diagonaali on yhdellä lyhyempi, ja viimeinen tarvittava diagonaali on $m + 1 - p_{alar}$.

Diagonaalit voidaan koota yhdeksi *vektoriksi*, jonka yhteispituus on $p_{ylär}(m + 1 - p_{ylär}) + (p_{ylär} - 1 + p_{alar})(p_{ylär} - p_{alar})/2$. Vektorissa diagonaalit ovat edustettuina kasvavassa numerojärjestyksessä siten, että sen indeksipaikat $1..p_{ylär}$ on varattu 1. diagonaalin käyttöön. Toisen diagonaalin käytössä ovat vastaavasti paikat $p_{ylär}+1..2 \cdot p_{ylär}$. Viimeinen diagonaaleista, jolle mahtuu heuristisen ylärajan verran $L_i(k)$-arvoja, on järjestyksessä diagonaali numero $m+1-p_{ylär}$. Jokaisen yksittäisen diagonaalin muistialue on yhtenäinen, ja kahden perättäisen diagonaalin d ja $d + 1$ muistialueiden

alkukohtien etäisyys on sama kuin ylärajan pituus, jos diagonaalin järjestysnumero on väliltä $1..m + 1 - p_{ylär}$. Jos järjestysnumero on tätä suurempi, saadaan alkukohtien etäisyys lausekkeesta $m - d + 1$.

Vaikka NKY:n käyttämä matriisi puristetaan *1-ulotteiseksi vektoriksi*, ei aliluvussa 8.2.1 esitelty matriisin uudelleenorganisointitekniikka kuitenkaan vesity saman tien, sillä kuten edellä todettiin, vektorissa jokaisen diagonaalin muistialue on peräkkäinen. Samoin, kun verrataan perättäisten diagonaalien $L_{i+1}(k)$ ja $L_i(k)$ arvoja, niiden indeksien välinen etäisyys pysyy samana koko tutkittavan diagonaalin käsittelyn aikana. Esitetään lopuksi vielä esimerkki, miten NKY:n muistinkäyttö muuttuisi esimerkin 3.23 syöteaineistolle, jos muistinvaraus tehtäisiin edellä ehdotetulla tavalla, kun heuristiseksi ylärajaksi olisi laskettuna 12 ja alarajaksi 8.

Esimerkki 8.3: *Muutokset NKY-algoritmin laskennassa ja muistinkäsittelyssä esimerkin 3.23 mukaisille syötteille, kun algoritmia modifioidaan kombinoimalla aliluvussa 8.2 esitettyjä tekniikoita.*

										Y								
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
			B	A	A	C	D	C	B	B	A	C	C	C	D	D	B	A	B
	1	A	**17**	*17*	17	*17*	*17*	*17*	*17*	*17*	0	0							
	2	B	**12**	**15**	**16**	*16*	*16*	*16*	*16*	*16*	0	0							
	3	B	**8**	**9**	**15**	*15*	*15*	*15*	*15*	*15*	0	0							
	4	B	*3*	**8**	8	**12**	12	12	12	**14**	0	0							
	5	D	*1*	**7**	7	**8**	**11**	11	11	**13**	0	0							
	6	D	0	**6**	6	6	**9**	9	9	9	0	0							
	7	D	0	*1*	**4**	4	**5**	5	5	**8**	0	0							
	8	A	0	0	**3**	3	3	3	3	**7**	0	0							
X	9	C	0	0	0	0	0	**1**	1	1	0								
	10	B	0	0	0	0	0	0	0	0									
	11	C	0	0	0	0	0	0	0										
	12	B	0	0	0	0	0	0											

Esimerkissä 8.3 on kuvattu, miten NKY:n laskenta ja muistinkäyttö muuttuvat, kun siihen tehdään kaikki ne muutosehdotukset, joita esiteltiin asteittain luvussa 8.2: *matriisin uudelleenorganisointi*, *Y:n optimaalisten loppuliitteiden havaitseminen*, *perättäisten täsmäysketjujen tunnistaminen* sekä *heuristisiten rajojen laskenta*. Solut, joissa ei vierailla kertaakaan, säilyttävät arvonaan alustuksessa saamansa nollan. **Lihavoidulla** on merkitty arvot, jotka tehostavat *Y*:n loppuliitteen valintaa aikaisemmasta. Ainoastaan näillä arvoilla on merkitystä PYAa aikanaan kerättäessä. Pelkällä *kursiivilla* merkityt arvot kuuluvat jo kertaalleen löydettyyn *Y*:n optimaaliseen loppuliitteeseen, ja niitä ei modifioitu NKY-algoritmi enää laske. *Kursivoidut*

alleviivatut arvot jäävät tarkastelematta alarajakatkaisun takia. Esimerkiksi ensimmäisen diagonaalin tarkastelu lopetetaan, kun on havaittu, että $L_1(4) < 5$, sillä näin ollen kyseiselle diagonaalille ei enää mahdu alarajan vaatimia kahdeksaa täsmäystä. Pelkästään alleviivatut arvot on saatu kopioimalla edelliseltä diagonaalilta yhtenäinen täsmäysjakso, joka voi olla jopa vain yhden merkin mittainen. PYA-jono muodostuu täsmälleen samoista merkeistä kuin esimerkissä 3.23.

Muistia jouduttaisiin varaamaan $L_i(k)$-arvojen tallentamista varten edellä mainituin oletuksin PYAn pituuden vaihteluvälistä ainoastaan 110 yksikköä, kun taas alkuperäinen NKY olisi varannut käyttöönsä 18 * 17 = 306 solua sisältävän matriisin. Muistitilan säästö olisi siten yli 60 prosenttia heuristisesti esiprosessoidun ja aputietorakenteiden indeksointitavaltaan uudistetun version hyväksi.

9 Testiajojen tuloksia

Edellisissä luvuissa tarkasteltiin PYA-ongelmaa monesta eri näkökulmasta. Aluksi tehtiin kattava *tarkkojen PYA-algoritmien* esittely luvussa 3. Seuraavaksi näiden algoritmien joukkoa laajennettiin esittelemällä PYAn ratkaisemiseen soveltuvat versiot alun perin *kahden merkkijonon välisen lyhimmän editointietäisyyden* ratkaisemiseksi kehitetyistä algoritmeista luvussa 4. Kaikille tähän laajennettuun joukkoon kuuluville algoritmeille on yhteistä se, että ne ratkaisevat sekä PYAn *tarkan pituuden* että lisäksi palauttavat yhden, ongelman ratkaisuksi kelpaavan *PYA-jonon esiintymän*. Yksi tarkkoihin menetelmiin liittyvistä mielenkiintoisimmista kysymyksistä on selvittää, *miten tehokkaasti menetelmät suoriutuvat tehtävästään erityyppisille käytännön syöteaineistoille*. Algoritmien aika- ja tilavaativuuslausekkeet antavat tosin jo melko hyvän käsityksen siitä, miten käyttökelpoisia eri menetelmät ovat silloin, kun syötteiden koon ja muiden suorituksen kannalta kriittisten parametrien arvojen annetaan kasvaa rajatta. Pulmana on kuitenkin se, että lausekkeissa vakiokertoimet ja muuttujien korkeinta astetta alemmat termit jäävät huomiotta, ja niiden vaikutus voi olla varsin huomattava monilla mahdollisilla käytännön syötteillä. Käsillä olevan luvun aliluvussa 9.2 keskitytään kilpailuttamaan *tarkkoja, heuristisesti tehostamattomia PYA-algoritmeja* keskenään.

Tarkkojen PYA-algoritmien kentän kartoituksen jälkeen siirryttiin tarkastelemaan *heuristisia PYA-menetelmiä* luvussa 5. Näistä *ylärajamenetelmät* laskevat pelkästään PYAn pituuden ylärajan palauttamatta ratkaisuna sen lisäksi mitään merkkijonoa. Sen sijaan *alarajamenetelmät* eivät laske pelkästään tarkasteltavan ongelman PYAn vähimmäispituutta, vaan ne myös palauttavat jonkin alarajan mittaisista kyseisen ongelman yhteisistä alijonoista. Jotta heuristiset menetelmät olisivat käytännössä hyödyllisiä, niiden pitää pystyä laskemaan laadultaan riittävän informatiivinen ylä- tai alaraja verraten nopeasti. Tämän lisäksi olisi suotavaa, että heuristiikat eivät olisi herkkiä syötteiden ominaisuuksille, kuten merkkijakaumalle, syöttöaakkoston koolle, PYAn suhteelliselle osuudelle, (dominanttien) täsmäysten lukumäärälle jne. Siten *pelkästään heuristisiten menetelmien vertailulle* on omistettu aliluku 9.3. Siinä pyritään antamaan lukijalle osviittaa siitä, mihin heuristisiin menetelmiin on luotettavinta turvautua silloin, kun tarkan ratkaisun etsiminen vaatii liian paljon aika- ja/tai tilaresursseja.

Puhtaasti heuristisiten menetelmien esittelyä seurasi luvussa 6 analyysi, jossa *heuristiikat upotettiin osaksi tarkkoja PYA-menetelmiä*. Tavoitteena oli ohjata ja järkeistää tarkan algoritmin toimintaa estämällä sitä laskemasta välituloksia, joiden tiedetään heuristisen esiprosessoinnin valossa olevan turhia tarkan PYAn ratkaisemisen kannalta. Ehdotuksia heuristiikkojen luotettavuuden parantamiseksi tarkasteltiin sittemmin luvussa 7. Mielenkiintoa herättää erityisesti kysymys, *onko heuristiikoista sanottavaa hyötyä tarkan PYA-algoritmin esiprosessoinnissa ja sen laskennan edettyä jo pidemmälle*. Tähän kysymykseen etsitään vastausta aliluvussa 9.4.

Luvussa 8 päädyttiin vielä tarkastelemaan eri *hakutietorakenteiden vaikutusta tarkan algoritmin suoritukseen*. Useimmissa PYA-algoritmeissa käytettävä hakutietorakenne voitaisiin verrattain helposti vaihtaa toiseksi: esimerkiksi *esiintymälistan* sijaan saatettaisiin rakentaa vaikkapa suorasaantia tukeva *lähiesiintymätaulukko*, tai vaihtoehtoisesti voitaisiin olla käyttämättä mitään haun aputietorakenteita ja soveltaa pelkkää *lineaarihakua* suoraan syötevektoriin. Eri *hakuratkaisujen välistä tehokkuutta* tarkastellaan lähemmin tämän luvun viimeisessä aliluvussa 9.5.

Ennen varsinaisten testituloksien esittelyä selostetaan aliluvussa 9.1, miten eri algoritmien testaaminen ja toistensa kanssa kilpailuttaminen toteutettiin. Tämä edesauttaa lukijaa vakuuttumaan testiajojen tulosten vertailukelpoisuudesta.

9.1 Yleistä testausasetelmasta

Kaikki testiajoihin mukaan valitut algoritmit – niin alkuperäiset tarkat menetelmät, heuristiikat kuin heuristiikalla vahvistetutkin – ohjelmoitiin testiajojen suorittamista varten *C-ohjelmointikielellä*. Testaus suoritettiin *Linux-käyttöjärjestelmän* version *2.6.30.5* alaisuudessa *Intel Xeon -työasemaa* käyttäen. Käännettäessä lähdekoodeja käytettiin *gcc-kääntäjän versiota 3.4.6* ja *käännösoptiota -O3*. Kone sisältää kaksi 1.8 GHz:n kellotaajuista prosessoria ja 1 gigatavun keskusmuistia. Testien yhteydessä raportoidut algoritmien suoritusajat on mitattu käyttöjärjestelmästä riippuvien *kellosykäysten* (engl. *clock tick*) määränä, joita testausympäristössä yhteen sekuntiin mahtui sata. Toisin sanoen, luvussa esiteltävien testien mittaustulokset voidaan tulkita tässä tapauksessa yhtäläisesti sadasosasekunneiksi. Muistinkulutusta mitattaessa perusyksikkönä on puolestaan *kilotavu*.

Algoritmeja ohjelmoitaessa niiden tarvitsemat tietorakenteet ja pseudokoodissa esiintyneet komennot pyrittiin toteuttamaan mahdollisimman autenttisesti alkuperäisartikkeleissa kuvatun mukaisesti. Dynaamisten tietorakenteiden muistinvaraus järkeistettiin siten, että muistia varattiin kerralla yksi isompi lohko sen sijaan, että sitä olisi varattu toistuvasti vain tietyllä hetkellä tarvittava määrä lisää. Tällä tavoin saatiin algoritmien dynaamista muistinkäyttöä tehostettua ja yhdenmukaistettua.

Algoritmeille annettiin testiajoissa sekä *tasaista* että harmonisiin lukuihin perustuvaa vinoa, ns. *Zipfin merkkijakaumaa* [Zip35] kummassakin jonossa X ja Y noudattavia syötteitä. Tasaisella jakaumalla syöttöaakkoston kunkin merkin esiintymistodennäköisyys on sama: $1/\sigma$, kun taas Zipfin jakaumalla jokaisella merkillä se on erisuuri. Erityisesti järjestyksessä i. yleisimmän merkin ($1 \leq i \leq \sigma$) esiintymisen todennäköisyys on $1/(i \cdot H_\sigma)$, missä H_σ tarkoittaa σ. *harmonista lukua*, joka saadaan laskemalla summa $\sum_{i=1}^{\sigma} \frac{1}{i}$. Tämän lisäksi syötteinä käytettiin luonnollista kieltä sisältäviä dokumentteja [Ber08][Hal08][Kal07]. Tasaista ja Zipfin jakaumaa

noudattavilla syötteillä syöttöaakkoston koon annettiin vaihdella 2:sta 256 merkkiin. Täten pystytään havainnoimaan, miten herkkiä eri menetelmät ovat *syöttöaakkoston koolle* ja *merkkijakaumalle*.

Syötejonojen pituudeksi valittiin *10 000 merkkiä* lukuun ottamatta yhtä testiasetelmaa, jossa *X*:n pituudeksi valittiin *5 000* ja *Y*:n pituudeksi *10 000* merkkiä. Yhtä ja samaa aineistoa kohti sama algoritmi suoritettiin *viisi kertaa peräkkäin*, jotta mahdolliset suoritusaikojen mittausvirheet olisivat tasapainottuneet. Jokaista tasaista tai Zipfin jakaumaa noudattavalle aineistotyypille generoitiin *10 eri syötejonoparia*. Kuvissa esiintyvät ajoajat ovat siten viiden perättäisen suorituskerran keskiarvoja 10 eri syöteaineistolle. Edellä mainittua poikkeuksellista asetelmaa lukuun ottamatta kumpikin syötejono valittiin yhtä pitkäksi, eli $m = n$. PYAn pituuden prosenttiosuuden annettiin keinotekoisille syötejonoille vaihdella 30 prosentista 90 prosenttiin, jotta nähtäisiin, minkä verran PYAn suhteellisella osuudella on vaikutusta algoritmien tehokkuuteen.

Tasaista ja Zipfin jakaumaa noudattavat, testiajojen suorittamiseksi generoidut syötejonot luotiin käyttämällä tarkoitusta varten kehitettyä *satunnaislukuja tuottavaa algoritmia*. Kun haluttiin muodostaa esimerkiksi tasaista merkkijakaumaa noudattavat syötejonot, joiden aakkoston koko olisi 32 ja PYAn pituus 50 %, asetettiin ensiksi kumpaankin syötejonoon samassa järjestyksessä peräkkäin 5 000 merkkiä, jotka olivat määräytyneet tuottamalla tasaista jakaumaa välillä 1 – 32 noudattavia satunnaislukuja. Tämän jälkeen sijoitettiin syötejonoon *X* mielivaltaisiin paikkoihin loput 5 000 merkkiä generoimalla lisää tasaista merkkijakaumaa noudattavia satunnaislukuja edellä mainitulta väliltä. Vektori *Y* muodostettiin samalla periaatteella. Eri syötejonoihin asetetut täytemerkit sijoitettiin niihin siten täysin toisistaan riippumatta. Tehtyjen operaatioiden ansiosta syötejonojen PYAn pituudeksi tuli selvästikin vähintään 50 %, koska aluksi kumpaankin jonoon syötetyt 5 000 merkkiä kuuluvat syötejonojen yhteiseen alijonoon. On kuitenkin erittäin ilmeistä, että jälkeenpäin vektoreihin syötetyt täytemerkit aiheuttavat jonojen PYAn pitenemisen asetetusta tavoitellusta prosenttiosuudesta. Niinpä likimain haluttuun PYAn prosenttiosuuteen pääsemiseksi jouduttiinkin PYAn osuutta kuvaavaa parametria säätämään jonkin verran tavoiteltua pienemmäksi, jotta täytemerkkien lisäämisen jälkeen PYA olisi ollut vaaditun mittainen. Näin oli tarpeen toimia lähinnä syöttöaakkoston koon ollessa pieni, jolloin todennäköisyys PYAn tuntuvalle pitenemiselle täytemerkkien lisäämisen ansiosta on iso. Syöttöaakkoston koon kasvaessa täytemerkkien vaikutus PYAn pituuteen heikkenee vähitellen.

9.2 Tarkat algoritmit

Tarkkojen PYA-algoritmien välisiä tehokkuuseroja tarkasteltiin jo *Bergrothin, Hakosen* ja *Raidan* tutkimuksessa [BHR00, 47]. Siinä vertailuun otettiin mukaan suuri osa tämän työn luvuissa 3 ja 4 esitetyistä tarkoista menetelmistä: siis myös alun perin kahden

merkkijonon välisen lyhimmän editointietäisyyden laskentaa varten kehitetyistä menetelmistä. Poikkeuksen muodostivat *Hirschbergin II algoritmi* HI2 [Hir77, 669–673], *Goemanin ja Clausenin algoritmi* GCL [Goe99] sekä sen ohella kaikki muutkin tässä työssä esitellyt *lineaarisessa muistitilassa* toimivat algoritmit (ADI, AGL, HIL, KRA, MYE), jotka jätettiin tarkastelujen ulkopuolelle[155, 156]. Syynä tähän oli HI2:n kohdalla algoritmin kyvyttömyys löytää ehdoitta tarkkaa ratkaisua, ellei siinä aina varauduta pahimpaan eli nollan mittaiseen PYAan asettamalla rajaparametrille ε arvo m. Tällöin algoritmista tulee yleisesti ottaen kuitenkin kohtuuttoman hidas. Samaisesta syystä HI2 on jätetty myös tämän työn testiajojen ulkopuolelle. Analyysiin mukaan otettujen kaikkien menetelmien lyhenteet, niiden kehittäjät, julkaisemisvuosi sekä niiden teoreettiset aika- ja tilavaativuuslausekkeet on vielä yhteenvedonomaisesti koottu seuraavaan taulukkoon. Algoritmit on lajiteltu ensisijaisesti niiden toimintaperiaatteen ja toissijaisesti niiden julkaisuvuoden mukaan. Toimintaperiaatetta ilmaisee solun värikoodi: **sininen** = riveittäinen, **punainen** = korkeuskäyrittäinen, **keltainen** = syötejonojen loppuliitteitä laajentava, **vihreä** = monisuuntaisesti prosessoiva ja **violetti** = varsinaisesti editointietäisyyden laskentaan kehitetty menetelmä[157]. Samaa ryhmittelyä on käytetty myöhemmin kaikissa aliluvun 9.2 kuvissa.

[155] Mainitussa artikkelissa jätettiin lineaaritilaiset menetelmät systemaattisesti tarkastelematta artikkelin tiukan sivumäärärajoituksen tähden.

[156] Goemanin ja Clausenin algoritmia käsittelevä artikkeli ei ollut saatavilla artikkelin BHR00 kirjoittamisen aikoihin. Se olisi kuitenkin jäänyt käsittelemättä lineaaritilaisuutensa tähden.

[157] Wagnerin ja Fischerin algoritmi on luokiteltu tässä työssä riveittäin laskentaa suorittaviin, mutta se voitaisiin yhtäläisesti sijoittaa *lyhimmän editointietäisyyden* laskentaa varten kehitettyihin (vrt. tekijöiden artikkelin nimike ”*The string to string correction problem*”).

Taulukko 9.1: *Testiajoihin mukaan otetut tarkkojen PYA-menetelmien lyhenteet, menetelmien kehittäjät ja julkaisuvuodet sekä aika- ja tilavaativuudet. Merkintä (L) kirjoittajia kuvaavassa sarakkeessa ilmaisee menetelmän olevan lineaaritilainen.*

Algor.	*Tekijät ja ilmestymisvuosi*	*Aikavaativuus*	*Tilavaativuus*
WFI	*Wagner – Fischer, 1974*	$O(mn)$	$O(mn)$
HIL	*Hirschberg (L), 1975*	$O(mn)$	$O(n)$
HSZ	*Hunt – Szymanski, 1977*	$O(n + r \log m)$	$O(n + r)$
MUK	*Mukhopadhyay, 1980*	$O(n + r \log p + m \log p)$	$O(n + r)$
HD2	*Hsu – Du, 1982*	$O(mp \log (n/p) + mp + n)$[158]	$O(n + d)$
ADI	*Allison – Dix (L), 1986*	$O(n\sigma + m\lceil n/w \rceil)$	$O(\lceil n/w \rceil \sigma)$
AG2	*Apostolico – Guerra, 1987*	$O(n + d \log n)$ tai $O(n + d \log \log n)$[159]	$O(n + d)$
KCR	*Kuo – Cross, 1989*	$O(r + n + pm)$	$O(n + d)$
RI2	*Rick, 1994*	$O(n\sigma + Min\{pm, d\sigma\})$	$O(n\sigma + d)$
HI1	*Hirschberg, 1977*	$O(pn)$	$O(Max\{n, m^2\})$
HD1	*Hsu – Du, 1982*	$O(pm \log (n/m) + pm + n)$	$O(n + d)$
AGL	*Apostolico – Guerra (L), 1985*	$O(m \log m + pm \log s + n)$	$O(n)$
AG1	*Apostolico – Guerra, 1987*	$O(Max\{pm, \sigma n\})$	$O(n\sigma + d)$
CPO	*Chin – Poon, 1990*	$O(n\sigma + Min\{pm, \sigma d\})$	$O(n\sigma + d)$
NKY	*Nakatsu – Kambayashi – Yajima, 1982*	$O(n(m–p))$	$O(m^2)$
KRA	*Kumar – Rangan (L), 1987*	$O(n(m–p))$	$O(n)$
RI1	*Rick, 1994*	$O(Min\{mp, p(n – p)\} + n\sigma)$	$O(n\sigma + d)$
GCL	*Goeman – Clausen (L), 1999*	$O(Min\{mp, p(n – p) + n\sigma\})$	$O(n\sigma + \log m)$
MMY	*Miller – Myers, 1985*	$O(m(n–p))$	$O((n–p)^2)$
MYE	*Myers, 1986*	$O(m(n–p))$	$O(n)$
WMM	*Wu – Manber – Myers – Miller, 1990*	$O(m(n–p))$	$O((n–p)^2)$

9.2.1 Testiajot keinotekoisesti generoiduille syöteaineistoille

Tarkoille PYA-algoritmeille suoritetut testiajot kohdistettiin ensiksi *keinotekoisesti generoiduille syötejonoille*, jotka noudattivat joko *tasaista* tai vaihtoehtoisesti *Zipfin*

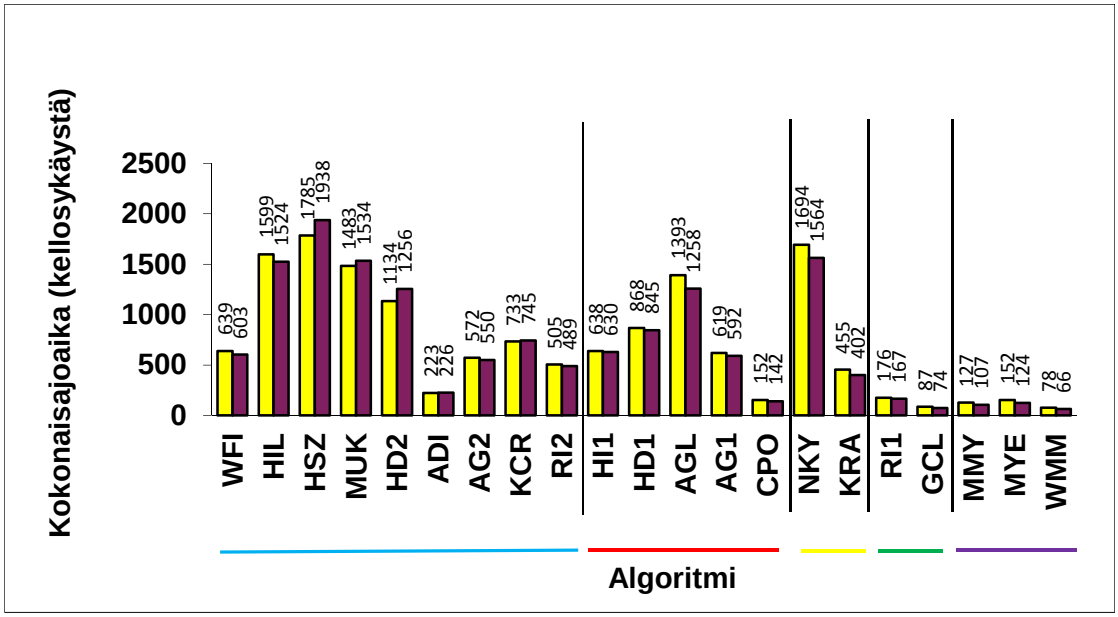
[158] Itse suorittamani analyysin mukaan aikavaativuus olisi $O(n + mp \log q + mp)$: kts. aliluku 3.2.4.2.
[159] Riippuu esiintymä- ja kynnysarvolistojen esittämiseen käytettävästä tietorakenteesta.

merkkijakaumaa. Seuraavassa analysoidaan testiajojen tulokset kutakin käytettyä syöttöaakkoston kokoa (2, 4, 20, 32 ja 256), jakaumatyyppiä (tasainen, Zipf) ja PYAn prosenttiosuutta (p / m)*100 % kohti.

9.2.1.1 Syöttöaakkoston koko 2

Kuvassa 9.1 esitetään tarkkojen alkuperäisten PYA-algoritmien testiajojen suoritusajat, kun syöttöaakkoston kooksi on valittu 2. *Binääriaakkoston* tarkastelu on mielenkiintoista, sillä sitä käytetään tietojenkäsittelyssä ja logiikassa *totuusarvotyyppisen tiedon* (on/ei) esittämiseen. Lisäksi binääriaakkoston syötejonoille on tyypillistä *suuri täsmäysten määrä*, kun merkkijakauma kummassakin jonossa on tasainen. Pystyakseli kuvaa suoritusaikaa, ja vaaka-akselilla algoritmit on eroteltu vasemmalta oikealle pystyviivoin viiteen ryhmään: *rivi kerrallaan* prosessoiviin, *korkeuskäyrittäin* eteneviin, *syötejonojen loppuliitteitä laajentaviin* varsinaisiin PYA-algoritmeihin (NKY ja KRA), *kaksisuuntaisesti prosessoiviin* (RI1 ja GCL) sekä *editointietäisyyden laskevista algoritmeista johdettuihin* PYA-menetelmiin. **Keltainen** ajoaikapylväs kuvaa *tasaista* ja **viininpunainen** *Zipfin* syötejakaumaa. PYAn suhteelliseksi osuudeksi on valittu 80 %. Useimmille PYA-algoritmeille teoreettisesti epäedullisin PYA-osuus on noin 50 %, sillä se mahdollistaa useiden kilpailevien PYA-jonojen esiintymisen ja lisää siten algoritmin tekemän kirjanpidon tarvetta. Syöttöaakkoston koolle 2 ei kuitenkaan pystytty tuottamaan 80 %:a lyhyempää PYAa syötteiden merkkijonon ollessa tasainen, joten tällä perusteella on taulukossa päädytty käyttämään PYA-osuutta 80 % hankalimman mahdollisen eli 50 %:n asemesta.

Kuvasta 9.1 havaitaan, että saman tehtävän ratkeamiseen kuluva aika vaihtelee suuresti eri PYA-algoritmeilla. *Tasaisella merkkijakaumalla* suoritusaikojen ero nopeimman eli *Wun, Manberin, Millerin ja Myersin algoritmin* (WMM) ja hitaimman eli *Huntin ja Szymanskin algoritmin* (HSZ) välillä on peräti yli 22-kertainen. Näistä WMM:n sijoittuminen kärkeen on varsin ymmärrettävää, sillä syötejonojen PYAn pituus on neljä viidennestä X:n pituudesta, joten menetelmä pystyy jättämään varsin huomattavan osan diagonaaleista käsittelemättä. Yleisestikin kaikki kolme alun perin kahden syötejonon lyhimmän editointietäisyyden määräämiseen kehitettyä algoritmia menestyvät testiajossa kiitettävästi.



Kuva 9.1: *Tarkkojen PYA-algoritmien suoritusajat, kun m, n = 10 000, σ = 2, p ≈ 8 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*). *Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

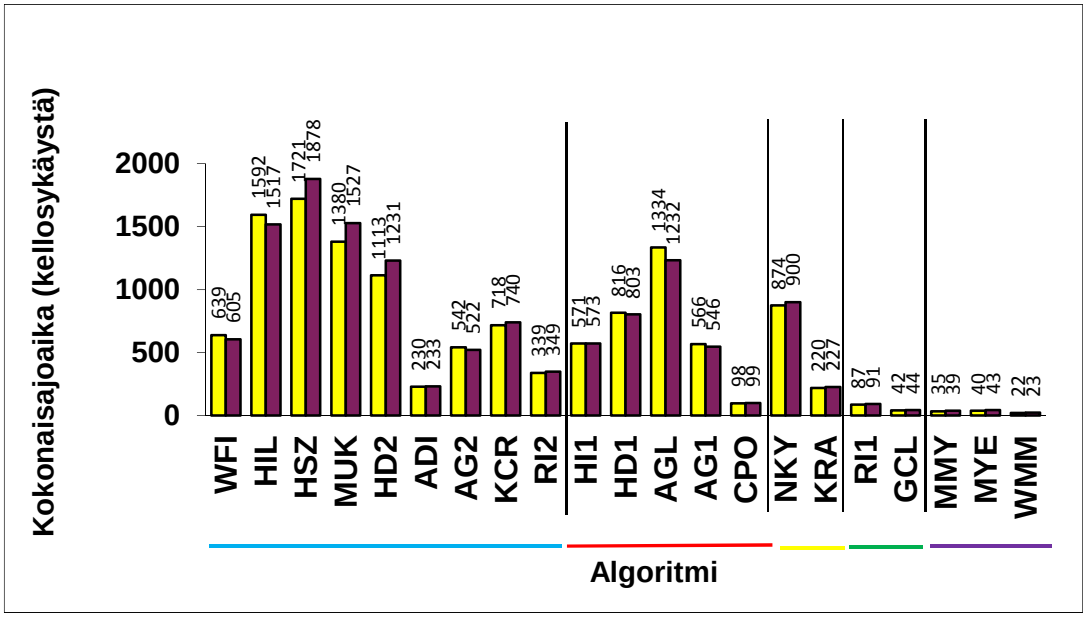
Toiseksi nopeimmaksi menetelmäksi kuvan aineistoille osoittautuu neljää lähiesiintymätaulukkoa käyttävä, mutta syötejonojen suhteen lineaarisessa muistitilassa toimiva *Goemanin ja Clausenin algoritmi* (GCL). Myöskään tämä tulos ei ole yllättävä, sillä menetelmä erottelee tehokkaasti eri suunnista lähtien dominantit täsmäykset pitäen samalla suorituksen aikaisen kirjanpidon niukkana. WMM ja GCL ovat ainoat menetelmät, joiden suoritusaika alittaa 100 yksikköä. Seuraavaksi parhaiksi menetelmiksi osoittautuivat testiajoissa kaksi muuta etäisyydenlaskentamenetelmää MMY ja MYE, pienille syöttöaakkostoille tekijöidensä mukaan kehitetty korkeuskäyrittäinen CPO sekä toinen kaksisuuntaisesti prosessoivista menetelmistä eli RI1. Näiden neljän ajoajat alittavat vielä 200 yksikköä. Niukasti kyseisen rajan huonommalle puolelle päätyy paras riveittäin prosessoivista menetelmistä: bittirinnakkaisuuteen perustuva ADI. Sen sijaan tämän jälkeen ajoajat heikkenevät tuntuvasti. Paras loppuliitteitä laajentava algoritmi KRA vaatii ADI:hinkin verrattuna yli kaksinkertaisen laskenta-ajan, ja ero parhaaseen menetelmään on miltei 6-kertainen.

Mielenkiintoista on havaita myös, että teknisesti primitiivisimmät menetelmät WFI ja HIL eivät suoriudukaan testistä kaikkein heikoimmin, kuten voisi ennakkoon aavistella. Heikoimman testituloksen saavuttaneen HSZ:n suoritusaika on kaikin puolin järkeenkäypä, sillä menetelmä pysähtyy riveillä kaikkien täsmäysten kohdalle ja etsii puolitushaulla, minkä luokan edustajia ne ovat[160]. Kun täsmäyksiä on runsaasti, tapahtuu riveillä eteneminen jopa hitaammin kuin pelkkää lineaarista selausta noudattaen WFI:n tapaan. Myös NKY:n suoritusaika on luokattoman heikko, mikä

[160] HSZ:n aikakompleksisuus on $O(r \log n)$, joka on heikompi kuin $O(n^2)$, kun täsmäysten lukumäärä $r = \omega(n^2/\log n)$.

myös selittynee täsmäysten paljouden aiheuttamalla runsaalla kirjanpitotyöllä suorituksen aikana. Kevyt esiprosessointi ei selvästikään pelasta esimerkin testiasetelmassa HSZ:aa, MUK:ta ja NKY:tä, kun taas toisia minimaaliseen esiprosessointiin perustuvia – lyhimmän editointietäisyyden laskentaan kehitettyjä menetelmiä – suosii voimakkaasti suhteellisesti pitkän PYAn löytyminen.

Zipfin jakauman kohdalla havaitaan, että kovin voimakkaita muutoksia suuntaan tai toiseen algoritmien suoritusajoissa ei ilmene. Samat menetelmät, jotka menestyivät parhaiten tasaisella jakaumalla, saavuttivat hyvän tuloksen myös vinolla merkkijakaumalla. Suurimmat muutokset olivat 10 – 15 prosentin suuruusluokkaa. Absoluuttisin arvoin mitaten ajoaika lisääntyi eniten HSZ:lla ja HD2:lla, ja vastaavasti AGL:n ja NKY:n suoritusajat vähenivät eniten verrattuna tasaisen jakauman mukaisiin ajoaikoihin. Suhteellisesti nopeutuminen oli kuitenkin voimakkainta jo tasaisella jakaumalla parhaimmistoon kuuluneella nelikolla GCL, MMY, MYE ja WMM: parannusta näille kertyi yli 10 %.



Kuva 9.2: *Tarkkojen PYA-algoritmien suoritusajat, kun m, n = 10 000, σ = 2, p ≈ 9 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*). *Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

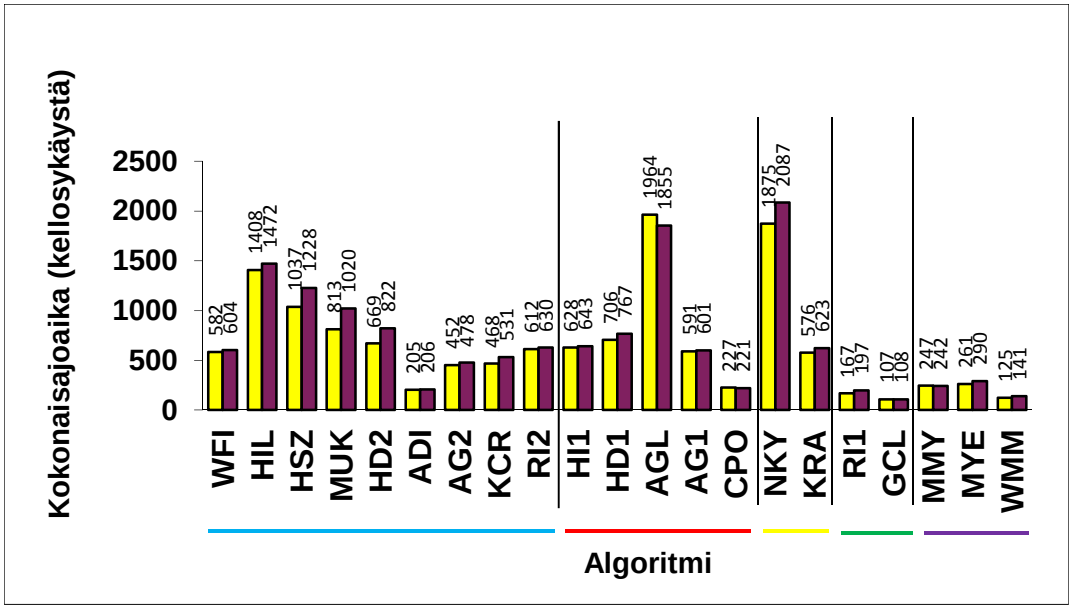
Kuvassa 9.2 on PYAn pituus suhteessa *m*:ään kasvanut 80 %:sta 90 %:iin. Kautta linjan on nähtävissä ajoaikojen nopeutumista edellisen kuvan tilanteeseen verrattuna, joskin pientä päinvastaistakin kehitystä on tapahtunut WFI:n ja ADI:n kohdalla. Absoluuttisesti mitattuna selvästi voimakkaimmin on nopeutunut NKY, jonka ajoajasta katosi lähes puolet PYAn pituuden kasvettua 10 prosenttiyksiköllä. Melkoisesta ajoajan paranemisesta huolimatta menetelmä on kaukana parhaimmistosta, johon kuuluvat samat kuusi algoritmia kuin kuvassa 9.1. Suhteellisesti mitattuna tehokkain algoritmi WMM nopeutui edellisen kuvan tilanteesta jopa yli 60 %! Kärkikolmikon muodostavat

nyt kaikki lyhimmän editointietäisyyden laskentaan kehitetyt menetelmät järjestyksessä WMM, MMY ja lineaaritilainen MYE, jonka kanssa lähes samoihin aikoihin pystyy monisuuntaisesti prosessoiva GCL. Kuvan 9.1 viidenneksi ja kuudenneksi nopeimmat menetelmät CPO ja RI1 ovat nyt vaihtaneet paikkaa keskenään RI1:n nopeuduttua lähes puolella edellisistä mittauksista. Nyt kaikki kuusi parasta menetelmää suoriutuvat tehtävästä alle 100 aikayksikön. Seitsemänneksi sijoittuvan ADI:n suoritusajat tuntuvat pysyneen lähes ennallaan, eli PYAn osuuden kasvamisella ei näyttäisi olevan juurikaan vaikutusta ADI:n toimintaan. Huonoiten tämänkin kuvan esittämässä testaustilanteessa menestyy HSZ, jota seuraavat HIL ja MUK. Sen sijaan menetelmiä HSZ ja MUK logiikaltaan pitkälti muistuttava KCR, josta kuitenkin turhien täsmäysten kirjaamista on karsittu edeltäjiinsä verrattuna, suoriutuu näistä testiajoista lähes puolta nopeammin. Kaikkien menetelmien käyttäytyminen tasaista tai Zipfin jakaumaa noudattavilla syötteillä on melko lailla yhdenmukaista kuvan 9.1 kanssa: yli 10 prosentin eroja ajoajoissa eri jakaumien välillä ei juurikaan esiinny.

Yhteenvetona kuvien 9.1 ja 9.2 perusteella voidaan todeta, että binääriaakkostolle tuntuisi olevan kannattavaa soveltaa periaatteessa minkä tahansa kahden merkkijonon lyhimmän editointietäisyyden laskentaan kehitetyn algoritmin (WMM, MYE, MMY) PYAn ratkaisevaa versiota. *Wun, Manberin, Myersin ja Millerin* algoritmi WMM selviytyi voittajana kaikista binääriaakkoston testiajoista, mutta toisaalta *Myersin* algoritmin käyttämistä puoltaa sen lineaaritilaisuus. Vaihtoehtoisesti voitaisiin ajatella käytettävän jotain algoritmeista GCL, RI1, CPO tai ADI. Tällöin vältettäisiin WMM:n toivottoman hidas suoritusaika hypoteettisille syötejonopareille, joissa *X* ja *Y* sisältävät hyvin vähän yhteisiä merkkejä. Viimeksi mainituista menetelmistä GCL ja ADI ovat lisäksi lineaaritilaisia. Minkä tahansa muiden kuin edellä lueteltujen menetelmien käyttäminen binääriaakkostosta muodostetuille syötejonoille näyttäisi tehtyjen testiajojen valossa olevan vaikeasti perusteltavissa.

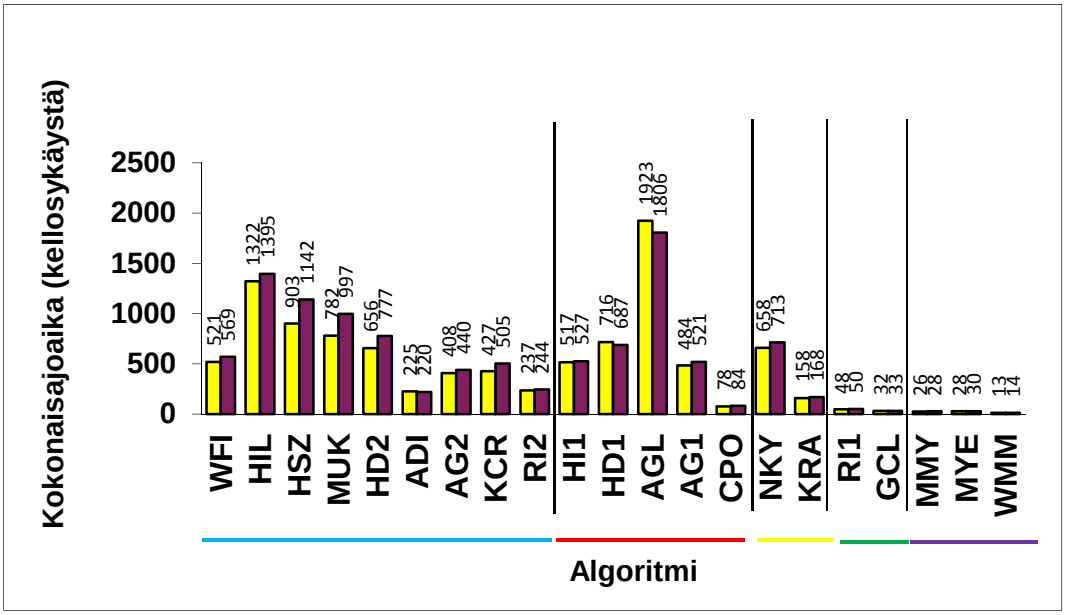
9.2.1.2 Syöttöaakkoston koko 4

Neljän kokoisen syöttöaakkoston tekee erityisen mielenkiintoiseksi *DNA-ketjuihin* liittyvä tutkimus, sillä ”DNA-aakkosto” koostuu juuri neljästä symbolista: ”A” = *adeniini*, ”C” = *sytosiini*, ”G” = *guaniini* ja ”T” = *tymiini*. Tälle aakkostotyypille suoritettiin sekä tasaista että Zipfin jakaumaa noudattaville aineistoille testiajot, joissa PYA-osuuksina esiintyivät 70 % ja 90 %. Lyhyemmän kuin 70 %:n PYAn muodostaminen neljän kokoiselle syöttöaakkostolle osoittautui jälleen mahdottomaksi tehtäväksi merkkijakauman ollessa tasainen.



Kuva 9.3: *Tarkkojen PYA-algoritmien suoritusajat, kun m, n = 10 000, σ = 4, $p \approx 7\,000$ ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*). *Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

Riveittäin prosessoivien menetelmien suoritusajat näyttävät Rickin II algoritmia lukuun ottamatta lyhentyneen binääriaakkoston mukaisista. Tämä on selitettävissä etenkin täsmäysten kokonaismäärän pienenemisellä. Ryhmästä erottuu selvästi edukseen *Allisonin ja Dixin* bittirinnakkaisalgoritmi ADI, joka on toistaiseksi reagoinut erittäin passiivisesti syötejonojen eri parametrien muutoksiin. Korkeuskäyrittäisistä menetelmistä HD1 nopeutuu jonkin verran, kun taas muiden algoritmien – erityisesti AGL:n – kohdalla tapahtuu heikkenemistä. Aakkoston koon kaksinkertaistuminen voi aikaisempia tilanteita alhaisemman PYA-osuuden ohella olla osasyynä CPO:n hidastumiselle. NKY ei selvästikään menesty hyvin, kun PYA on lyhyt ja aakkosto samalla verraten pieni. Monisuuntaisesti prosessoivien ja lyhimmän editointietäisyyden laskevien menetelmien ryhmät vievät jälleen kerran voiton. Nopeimmaksi algoritmiksi osoittautuu tällä kertaa *Goemanin ja Clausenin algoritmi*, jonka suoritusaika jää aavistuksen verran 100 aikayksikön huonommalle puolelle. GCL:n lisäksi myös WMM ja RI1 alittavat 200 aikayksikön rajan, ja näitä seuraavat lähinnä ADI, CPO, MMY ja MYE, eli kärkiseitsikko on aivan sama kuin binääriaakkostollekin. Zipfin jakaumalla suoritusajat ovat yleensä hieman pidempiä kuin tasaisella – tämä näkyy selvimmin HSZ:n ja MUK:n suoritusajoissa. Siitä huolimatta, että nuo kaksi menetelmää toimivat kuvan 9.3 testissä kovin vaatimattomasti, niiden suoritusajat ovat kuitenkin selvästi kutistuneet binääriaakkoston mukaisista täsmäysten kokonaismäärän vähenemisen tähden. AGL ja NKY vaikuttaisivat olevan epäedullisimmat valinnat PYA-algoritmeiksi kuvan 9.3 mukaisessa testiasetelmassa.

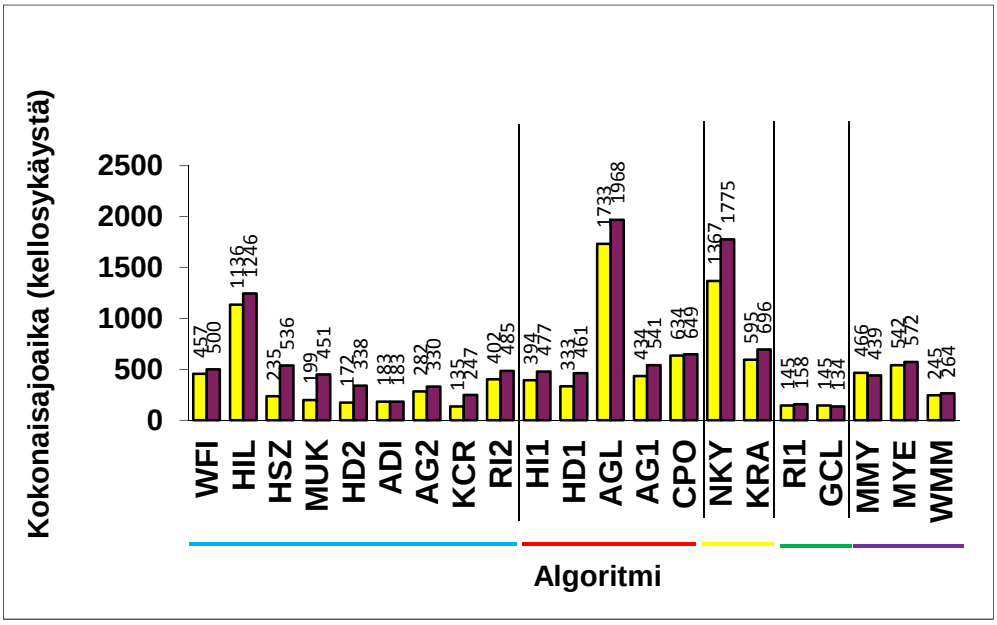


Kuva 9.4: *Tarkkojen PYA-algoritmien suoritusajat, kun m, n = 10 000, σ = 4, p ≈ 9 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*). *Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

Annettakoon seuraavaksi PYA-osuuden pidentyä 70 %:sta 90 %:iin. Kärkiryhmän osalta asetelma muistuttaa varsin selkeästi kuvan 9.2 mukaista tilannetta, jossa PYA-osuus oli sama mutta syöttöaakkostona oli puolta suppeampi binääriaakkosto. Wun, Manberin, Myersin ja Millerin algoritmi osoittautuu nyt jopa puolta nopeammaksi kuin sijoille 2 ja 3 sijoittuvat MMY ja MYE. Myöskään monisuuntaisesti prosessoivat menetelmät GCL ja RI1 eivät ylitä edes 50 aikayksikön työmäärää, mutta silti niistä heikommin suoriutuva Rickin I algoritmi vie jo yli 3½-kertaisen ajoajan voittajaan verrattuna. Viimeisenä algoritmina CPO alittaa 100 aikayksikön rajan. Jälleen kerran ADIn saavuttama ajoaika on hyvin stabiili edellisten testiajojen tulosten kanssa: ajoaika pysyttelee sitkeästi runsaassa 200 yksikössä. Huomattavana erona muutoksessa kuvan 9.2 tilanteeseen voidaan havaita, että kolmen ensimmäisen ryhmän algoritmien ajoajat ovat lähes systemaattisesti lyhentyneet muutamaa poikkeusta lukuun ottamatta. Erityisen suuri muutos on ns. *HSZ-perheen* menetelmissä (HSZ, MUK ja KCR). Näillä menetelmillä, samoin kuin HD2:lla, Zipfin jakaumaa noudattavan syötteen ratkaisuaika on vähintään 10 % pidempi kuin tasaisella jakaumalla; MUK vaatii Zipfin jakaumalla jopa yli 20 % enemmän ajoaikaa. Muiden menetelmien kohdalla ajoaikojen erot eri jakaumien välillä ovat vähäisehköt. Kuvan 9.4 perusteella – samoin kuin edellistenkin kolmen kuvan perusteella – olisi vaikeaa suositella neljän kokoiselle syöttöaakkostolle muita algoritmeja kuin kahteen viimeiseen ryhmään kuuluvia tai CPO:ta. Mikäli on olemassa riski pienemmästä kuin 70 %:n PYA-osuudesta, monisuuntaisesti prosessoivat algoritmit GCL ja RI1 tuntuisivat turvallisimmilta vaihtoehdoilta. Näistä ensin mainittu selviytyy tehtävästään lisäksi vieläpä lineaarisella muistinkäytöllä.

9.2.1.3 Syöttöaakkoston koko 20

Seuraavaksi tarkastellaan syötejonoja, joilla aakkoston kokona on 20. Tätäkään aakkoston kokoa ei ole valittu testiajoihin sattumanvaraisesti, vaan se on relevantti *proteiiniketjujen samankaltaisuuden tutkimuksessa*: erilaisten aminohappojen lukumäärä on 20. Kyseiselle symbolimäärälle pystytään jo tuottamaan sekä tasaista että Zipfin jakaumaa noudattavia syötejonoja, joiden PYA-osuutena on 50 %. PYAn pituuden ollessa puolet lyhyemmän syötejonon pituudesta saavutetaan useiden PYA-algoritmien ajankäytön teoreettisesti pahin tapaus. Algoritmit, joiden aikavaativuudessa esiintyy lauseke $p(n - p)$, ovat tuolloin huonoimmillaan. Näihin algoritmeihin kuuluvat kumpikin monisuuntaisesti prosessoivista eli RI1 ja GCL. Kun samalla syöttöaakkoston kokokin on ehtinyt viisinkertaistua sitten edellisen aliluvun, pitäisi myös samaisten menetelmien esiprosessointivaiheen tulla tuntuvasti aiempaa työläämmäksi. Siten on mielenkiintoista saada näyte näiden menetelmien toimivuudesta epäedullisille syötteille.



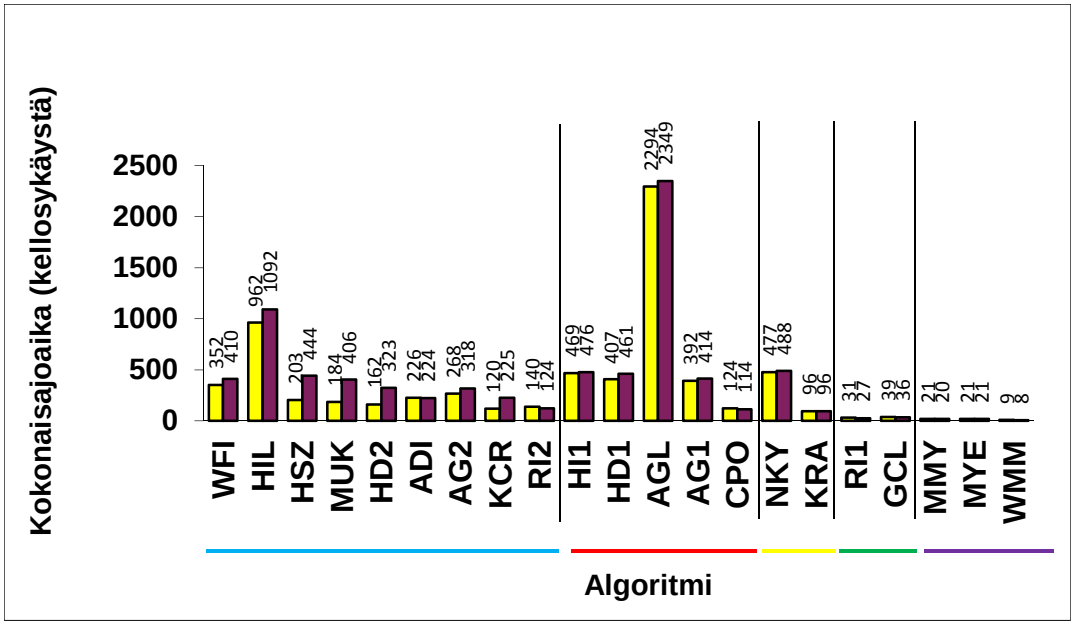
Kuva 9.5: *Tarkkojen PYA-algoritmien suoritusajat, kun m, n = 10 000, σ = 20, $p \approx 5\,000$ ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*). *Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

Kuva 9.5 välittää melko lailla muuttuneet tulokset eri algoritmien keskinäisestä paremmuudesta edellisiin tuloksiin verrattuna. Tarkastellaan ensiksi *tasaisen jakauman* tuloksia. Ensimmäistä kertaa parhaaksi menetelmäksi selviytyy riveittäin prosessoiva algoritmi KCR, jonka suorituskyky edellisissä testiajoissa on kärsinyt suuresta täsmäysten määrästä syöttöaakkoston koon ollessa pieni. Seuraavalle sijalle 2 sijoittuvat — täysin ennakko-odotusten vastaisesti — tasapäin algoritmit RI1 ja GCL, joiden pitäisi edellä esitetyn analyysin perusteella menestyä nyt vaatimattomasti. Menetelmät

häviävät ajoajassa KCR:lle kuitenkin vain alle 10 %. Tulos on merkki siitä, että lähiesiintymätaulukoiden alustaminen 20:n kokoiselle aakkostolle ei tee tyhjäksi niistä saavutettuja laskennallisia etuja — vaikka niitä olisi neljä kappaletta kuten GCL:ssä — jos algoritmi muuten toimii tehokkaasti. Mielenkiintoa herättää tosin havainto, että nyt RI1 saavuttaa ensimmäistä kertaa ajoajassa vähintään yhtä hyvän tuloksen kuin GCL. Tämä saattaisi kuitenkin olla merkki siitä, että neljän lähiesiintymätaulukon perustaminen kahden sijaan alkaa jo jossain määrin rasittaa GCL:ää aakkoston kasvamisen myötä.

Loput kolme 200 aikayksikön alittajaa olivat kaikki riveittäin prosessoivia menetelmiä järjestyksessä HD2, ADI ja MUK. Myös HSZ saavuttaa huomattavasti paremman tuloksen kuin kertaakaan aikaisemmissa testeissä. Korkeuskäyrä kerrallaan prosessoivista menetelmistä HI1, HD1 ja AG1 suoriutuvat myös nopeammin kuin aiemmin, mutta niiden ajoaika on silti vähintään 2½-kertainen voittaja KCR:ään verrattuna. Huomion arvoista on pienillä aakkostoilla hyvin toimineen CPO:n selkeä heikkeneminen. Syötejonojen loppuliitteitä laajentavien algoritmien suoritusajat, eritoten NKY:n, jäävät heikoiksi. Kuvasta havaitaan myös aikaisemmissa kuvissa poikkeuksetta kärkipäähän sijoittuneiden lyhimmän editointietäisyyden laskentaan kehitettyjen algoritmien suoritusaikojen heikentyminen. Tämä selittyy loogisesti tähän mennessä alhaisimmalla testatulla PYA-osuudella, joka pakottaa menetelmät tutkimaan useampia diagonaaleja kuin PYAn ollessa pidempi suhteessa lyhemmän syötejonon pituuteen.

Zipfin jakaumalla kärkikolmikoksi saadaan GCL, RI1 ja ADI. Lisäksi kuvasta paljastuu neljän riveittäisen menetelmän — HSZ:n, MUK:n, HD1:n ja tasaisen jakauman testin voittaneen KCR:n — herkkyys merkkijakauman tyypille. Edellä mainituista menetelmistä HSZ:n ja MUK:n ajoajat vinolle jakaumalle ovat yli kaksinkertaiset, eivätkä paljoa vähemmällä kasvulla selviä myöskään HD2 ja KCR. Uskottavimpana selityksenä ilmiölle voidaan pitää täsmäysten lukumäärän kasvua, kun jakauma muuttuu tasaisesta vinoksi. Lisäksi loppuliitteitä laajentavan NKY:n taantuminen Zipfin jakaumalla on voimakasta etenkin absoluuttisesti mitattuna. Suhteellisesti muutos on kuitenkin vähäisempi kuin eniten heikentyneillä riveittäisillä menetelmillä. Zipfin jakaumalla ainoastaan kärkikolmikko alittaa ajoajassa 200 aikayksikön rajan, ja alle 300 yksikön selviytyvät tehtävästä sen lisäksi ainoastaan KCR ja WMM.



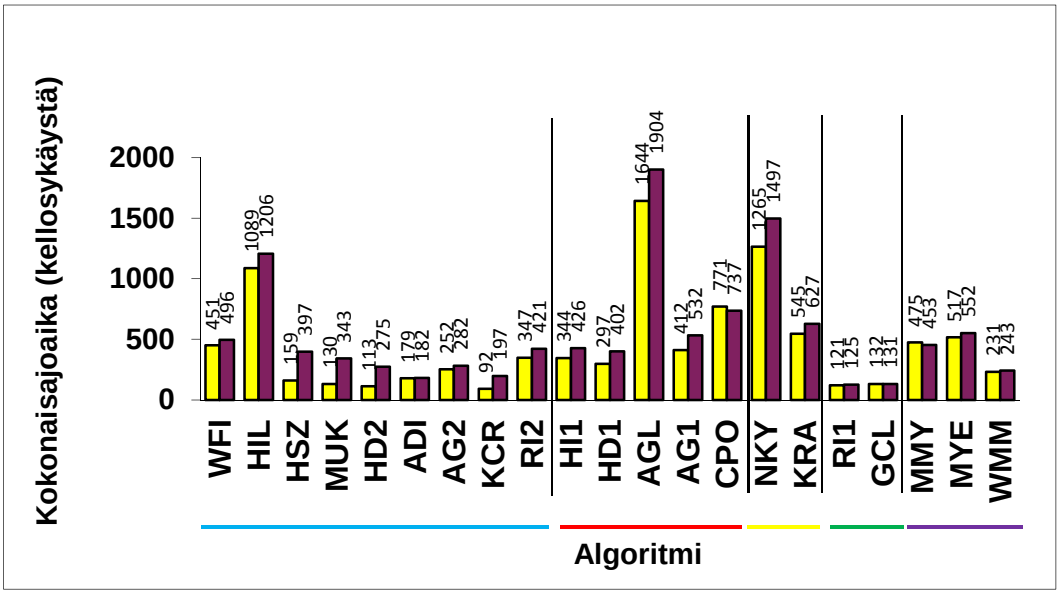
Kuva 9.6: *Tarkkojen PYA-algoritmien suoritusajat, kun m, n = 10 000, σ = 20, p ≈ 9 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*). *Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

Kuvassa 9.6 PYAn osuus on kasvanut 50 %:sta 90 %:iin. Tällaisen muutoksen pitäisi suosia etenkin lyhimmän editointietäisyyden laskevia algoritmeja. Ennakko-odotusten mukaisesti niihin lukeutuva WMM osoittautuu nopeimmaksi, ja seuraavat sijat menevät MMY:lle ja MYElle. Vaikka viimeksi mainittu kaksikon ajoaikatulos on sinänsä erinomainen: vain 21 yksikköä, se käyttää laskenta-aikaa silti jo yli kaksinkertaisesti WMM:n tarvitsemaan häikäisevän vähäiseen 9 yksikköön verrattuna. Editointietäisyyden laskevien menetelmien jälkeen seuraavaksi parhaiten menestyy kaksikko RI1 ja GCL, jotka kumpikin alittavat 40 aikayksikköä. Niukasti alle 100 yksikön selviytyy vielä Kumarin ja Ranganin algoritmi KRA, joka toimii lineaarisessa tilassa.

Kuuden parhaan algoritmin järjestys pysyy identtisenä, tarkasteltiinpa sitten tasaista tai Zipfin jakaumaa. Osalla rivi kerrallaan prosessoivista algoritmeista – samoin kuin kuvassa 9.5 – ajoaika kasvaa tuntuvasti siirryttäessä tasaisesta jakaumasta Zipfin jakaumaan. Ilmeisenä syynä tälle on täsmäysten lukumäärän kasvaminen kuten edelläkin. Yleisenä havaintona voidaan todeta, että useille algoritmeille suhteellisesti pitkä syötejonojen PYA on suotuisampi syöte kuin sellainen, jossa PYA on lyhyempi. Kuvan 9.6 perusteella tuntuisi vaikealta suositella käytettäväksi muita kuin lyhimmän editointietäisyyden laskentaan tai monisuuntaisesti prosessoivia algoritmeja, kun syöttöaakkoston koko on 20 ja PYAn pituus alkaa lähestyä syötejonon *X* pituutta *m*. Ellei PYAn pituudesta ole kuitenkaan mitään ennakkotietoja, vaikuttaisivat monisuuntaisesti prosessoivat RI1 ja GCL turvallisimmilta valinnoilta, sillä ne takaavat nopean suorituksen myös PYAn ollessa jopa vain 50 % *X*:n pituudesta.

9.2.1.4 Syöttöaakkoston koko 32

Kuvassa 9.7 on syöttöaakkoston koon annettu kasvaa 20:stä 32:een. ja PYAn osuudeksi on kiinnitetty 50 %. Tämän kokoiseen aakkostoon mahtuvat useimpien *eurooppalaisten kielten kaikki aakkosmerkit*, kunhan isot ja pienet kirjaimet samaistetaan[161]. Aakkoston koon kasvun voisi hyvällä syyllä otaksua heikentävän niiden menetelmien käyttökelpoisuutta, jotka hyödyntävät perustamiskustannukseltaan $O(n\sigma)$ olevaa lähiesiintymä/symbolijärjestystaulukkoa.



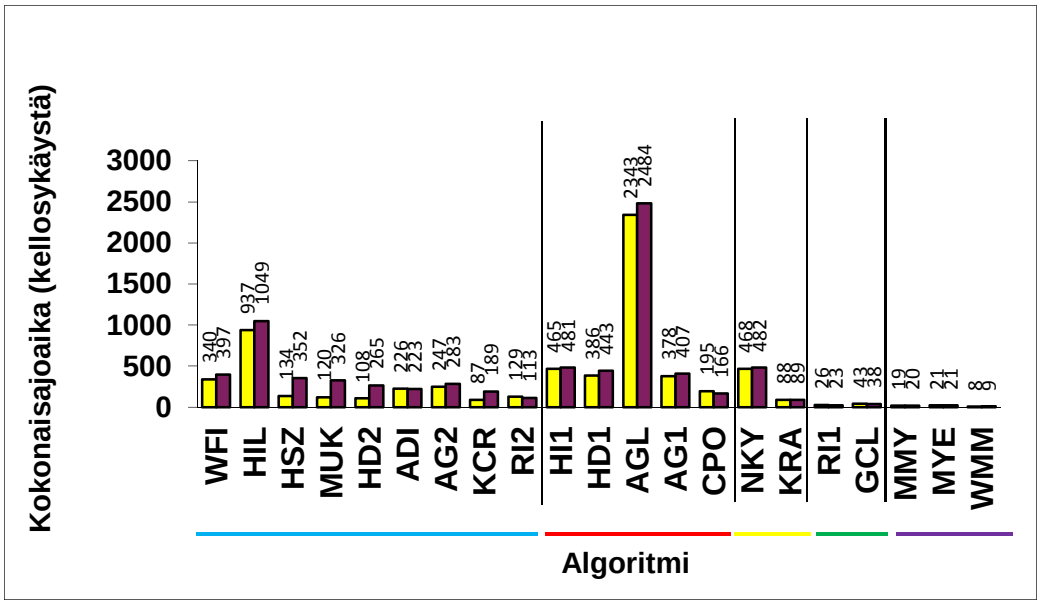
Kuva 9.7: *Tarkkojen PYA-algoritmien suoritusajat, kun m, n = 10 000, σ = 32, p ≈ 5 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*). *Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

Edellä esitetty hypoteesi suorasaantia tukevien taulukkojen vaatiman esiprosessointityön vaikeutumisesta näyttäisi kuvassa 9.7 toteutuvan CPO:n kohdalla, jonka suoritusaika on systemaattisesti kasvanut aakkoston kasvun myötä. Kuitenkin voidaan todeta, että esiprosessointi ei vaikuttaisi olevan suoritusajan kannalta mitenkään dominoiva tekijä. Esimerkiksi kuvan 9.5 tilanteeseen verrattuna, jossa PYAn osuus oli samoin 50 % mutta aakkoston koko 20, suoritusaika on kuvassa 9.7 kasvanut hädin tuskin viidenneksellä, kun taas aakkoston koko on kasvanut peräti 60 %. Sitä vastoin hypoteesi ei lainkaan pidä paikkaansa monisuuntaisten RI1:n ja GCL:n kohdalla, joiden suoritusajat ovat kuvaan 9.5 verrattuna jopa hieman nopeutuneet! Sama pätee muihinkin lähiesiintymätaulukoita käyttäviin algoritmeihin AG2 ja RI2. Tämä havainto kielii CPO:n tehottomasta toteutuksesta AG2:een, RI1:een ja RI2:een verrattuna.

[161] Esimerkiksi englannin kielessä on 26, suomen ja ruotsin kielissä 29, saksan kielessä 30 ja liettuan kielessä 32 aakkossymbolia.

Tasaista jakaumaa sovellettaessa seitsemän nopeinta algoritmia, joista jokainen alitti 200 aikayksikön rajan, ovat kaikki joko riveittäin tai kaksisuuntaisesti prosessoivia. Niiden keskinäinen paremmuusjärjestys on KCR, HD2, RI1, MUK, GCL, HSZ ja ADI. Parhaiten menestynyt *Kuon ja Crossin algoritmi* alittaa täpärästi myös 100 aikayksikön rajan. GCL:n ja RI1:n mahtuminen nytkin kärkiryhmään on selkeä todiste siitä, ettei edes 32:n kokoinen syöttöaakkosto ole vielä liian iso vesittämään lähiesiintymätaulukoista saatavia hyötyjä laskennassa. Ero RI1:n hyväksi on noin 10 prosentin luokkaa, joten neljän taulukon perustamiskulut alkavat kuormittaa GCL:ää suhteessa RI1:een. Yksikään korkeuskäyrittäinen menetelmä ei osoittautunut kuvan testiajossa hääppöiseksi, ja PYAn vain maltillinen prosenttiosuus 50 pitää lyhimmän editointietäisyyden laskentaan alun perin erikoistetut menetelmät poissa aivan parhaiden joukosta. AGL, NKY ja HIL kuuluivat algoritmeista nytkin hitaimpien joukkoon.

Zipfin jakaumalla algoritmien ajoajat heikkenevät kautta linjan – poikkeuksen muodostavat vain CPO, GCL ja MMY. Erityisen voimakasta heikkeneminen on HSZ-perheen menetelmillä sekä HD2:lla, joiden ajoaika Zipfin jakaumalla on yli kaksinkertainen tasaista jakaumaa noudattaviin syötteisiin verrattuna. Parhaimmaksi menetelmäksi osoittautuvat nyt RI1 ja GCL. Näiden lisäksi ainoastaan ADI ja KCR alittavat 200 aikayksikön rajan. Merkille pantavaa on jälleen kerran monisuuntaisesti prosessoivien menetelmien – GCL:n ja RI1:n – lisäksi myös ADI:n vahva riippumattomuus syötejonojen merkkijakaumasta: ero eri jakaumien välillä ADI:n suoritusajoissa on ainoastaan kolme yksikköä.



Kuva 9.8: *Tarkkojen PYA-algoritmien suoritusajat, kun m, n = 10 000, σ = 32, p ≈ 9 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*). *Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

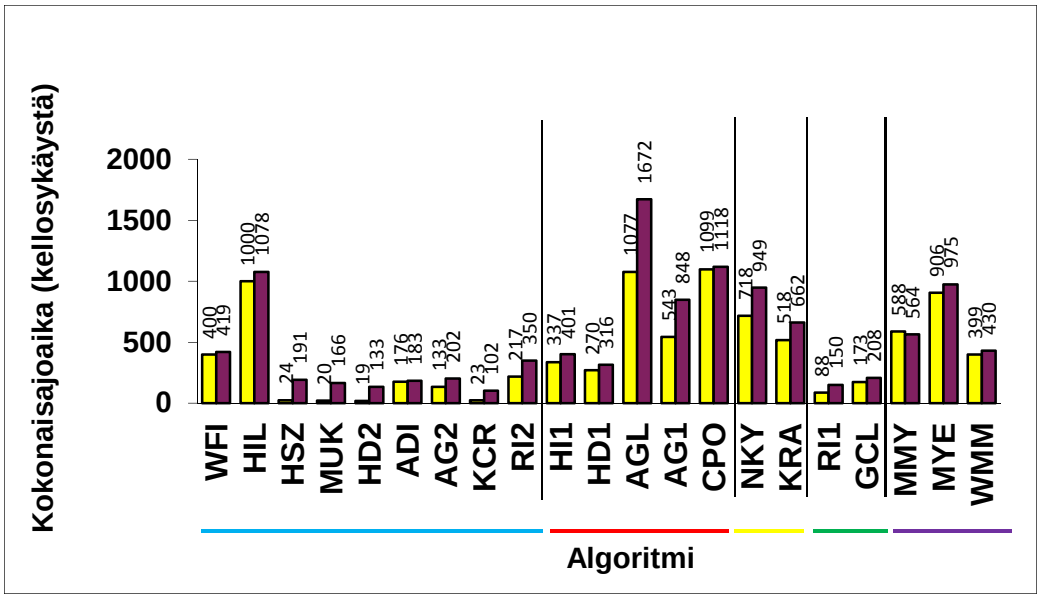
Kuvassa 9.8 testausasetelma on muuten kuin kuvassa 9.7, mutta PYAn prosenttiosuus on nyt kasvanut 50 %:sta 90 %:iin. Vastaavasti kuvan 9.6 mukaisesta tilanteesta kuvaan 9.8 siirryttäessä on muuttunut ainoastaan syöttöaakkoston koko 20:sta 32:een. Pylväiden keskinäinen suuruus kuvassa 9.8 on melko lailla samankaltainen kuin kuvassa 9.6. Alun perin lyhimmän editointietäisyyden mittaamiseksi kehitetyt algoritmit osoittautuvat jälleen parhaiksi, kun PYAn osuus on 90 %, mikä on hyvin sopusoinnussa algoritmien suoritusaikalausekkeen kanssa. Ne eivät myöskään millään tavoin tuntuisi olevan herkkiä merkkijakauman tyypille, vaan niiden suoritusajat tasaisen ja Zipfin jakauman syötteille ovat käytännöllisesti katsottuina samat. Tälläkin syöteaineistolla kaksisuuntaisesti prosessoivat RI1 ja GCL sijoittuvat aivan lähelle kärkeä. Riveittäin prosessoivista menetelmistä ovat naiivia WFI:tä ja siitä johdettua HIL:ää lukuun ottamatta tasaisella jakaumalla lähes kaikki parhainta korkeuskäyrä kerrallaan etenevää menetelmää – CPO:ta – tehokkaampia. Toteamisen arvoista on myös se, että jonojen loppuliitteitä laajentavista menetelmistä lineaaritilainen KRA voittaa suoritusajassa kirkkaasti ”esikuvansa” – ei-lineaaritilaisen NKY:n – rekursiivisista kutsuistaan huolimatta. Ongelman ratkeamiseen Zipfin jakaumalla kuluu muutamaa poikkeusta lukuun ottamatta enemmän aikaa kuin tasaisella jakaumalla. AGL:n tehottomuus tulee tässäkin testitilanteessa hyvin voimakkaasti esiin.

9.2.1.5 Syöttöaakkoston koko 256

Laajin tässä työssä testiajoihin otettu syöttöaakkosto on kooltaan 256-merkkinen. Yhtä monta merkkiä mahtuu *8-bittiseen ASCII-kooditauluun*. Tämä on samalla suurin merkkikokoelma, jonka työryhmämme kehittämä testausympäristö syötteille hyväksyy. Aakkoston käsittäessä 256 merkkiä on mahdollista muodostaa aikaisempia testiajoja pienempiä PYA-osuuksia sekä tasaiselle että Zipfin merkkijakaumalle. Siten tätä laajinta aakkostoa on testattu muista poiketen kolmelle eri PYA-osuudelle: 30, 50 ja 90 prosentin PYAlle.

Kuva 9.9 esittää testiajojen tuloksia, kun syöttöaakkoston kooksi asetettiin 256 merkkiä ja PYAn prosenttiosuudeksi vähäisehkö 30 %. Kuvasta voidaan havaita, että verraten alhainen PYAn osuus heikentää voimakkaasti syötejonojen loppuliitteitä laajentavia sekä lyhimmän editointietäisyyden laskentaan alun perin erikoistuneita algoritmeja ja tekee niistä tehottomia. Korkeuskäyrä kerrallaan etenevien pitäisi teoreettisesti menestyä vertailussa hyvin, kun PYA on lyhyt, mutta tästä huolimatta riveittäiset menetelmät osoittautuvat näitä selvästi tehokkaammiksi. Nopeimmaksi algoritmiksi *tasaisella merkkijakaumalla* osoittautuu esiintymälistojen ja kynnysarvovektorin osittaista limitystä käyttävä *Hsun ja Dun II algoritmi*. Ero lähimpiin kilpailijoihin eli MUK:hon, KCR:ään ja HSZ:aan on kuitenkin lähes olematon. Sen sijaan viidenneksi paras menetelmä aineistolle — RI1 — vaatii parhaaseen nelikkoon verrattuna jo noin viisinkertaisen suoritusajan. CPO:lla on

kyseenalainen kunnia ottaa tällä testiaineistolla itselleen useimmiten AGL:n hallussa ollut hitaimman algoritmin asema.

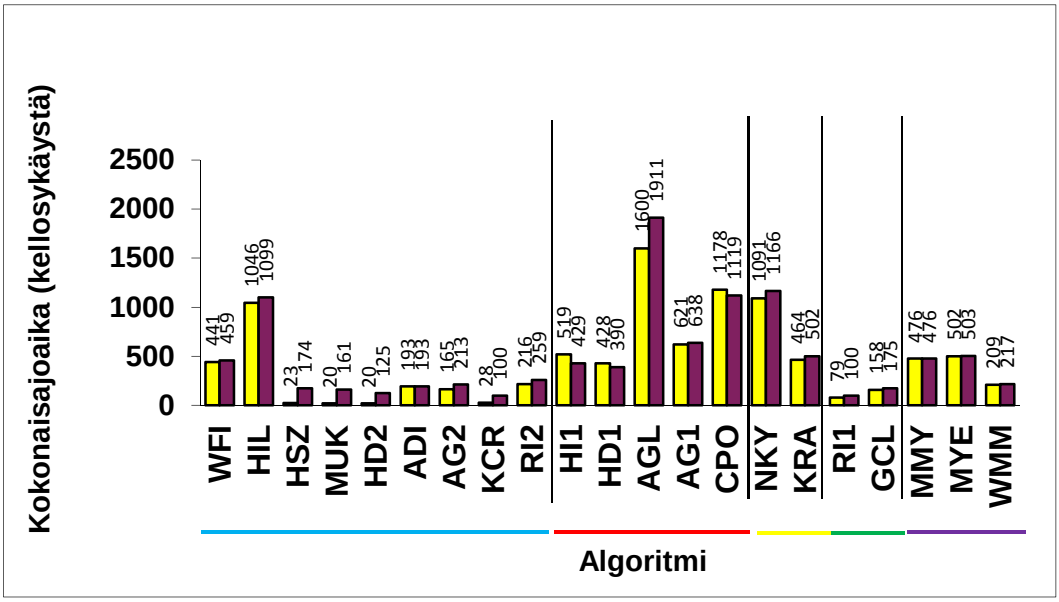


Kuva 9.9: *Tarkkojen PYA-algoritmien suoritusajat, kun m, n = 10 000, σ = 256, p ≈ 3 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*). *Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

Suuri syöttöaakkoston koko ja vähäinen täsmäysten lukumäärä suosivat selvästikin menetelmiä, jotka käyttävät hyväkseen esiintymälistoja, joita pitkin ne voivat laskentavaiheessa edetä tällöin varsin nopeasti. Edelleenkin RI1:n saavuttamat ajoajat ovat erittäin kilpailukykyisiä, vaikka kuvan 9.9 mukaisessa syötteessä se joutuu perustamaan esiprosessointivaiheessa kaksi lähiesiintymätaulukkoa, joihin talletetaan yhteensä likimain 5 120 000 kokonaislukua!

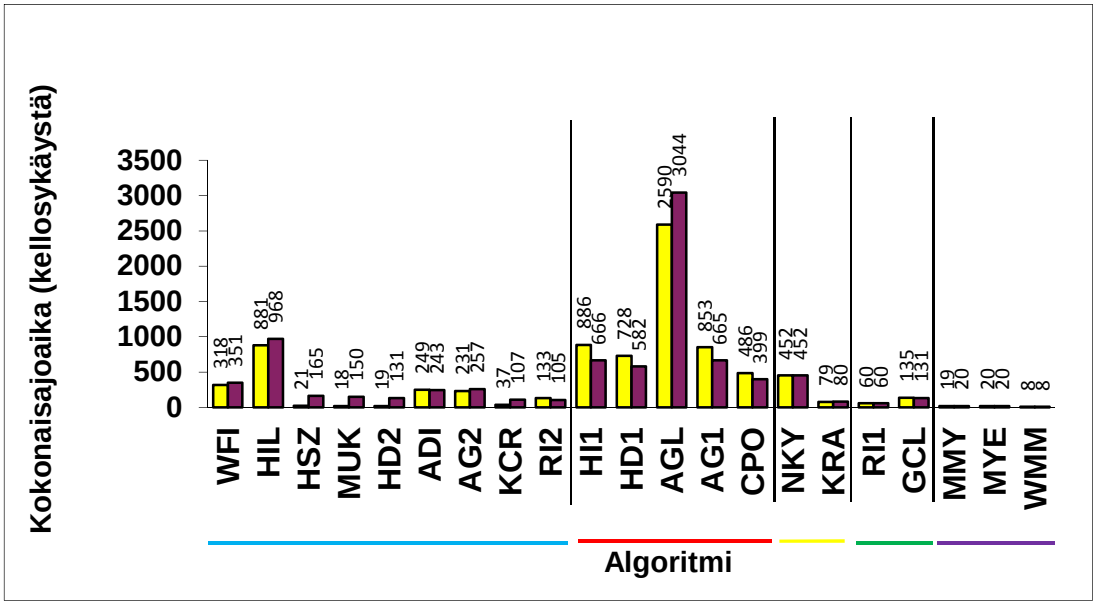
Zipfin jakaumalla RI1 osoittautuu jopa kolmanneksi parhaaksi menetelmäksi KCR:n ja HD2:n perään. Turha työ tuntuisi algoritmin suorittamassa laskennassa selvästikin olevan minimoitu, sillä esiprosessointivaihe esimerkiksi HSZ-perheen algoritmeilla (HSZ, MUK ja KCR) on huomattavasti tätä vähäisempää, ja tästä huolimatta RI1 pystyy voittamaan niistä muut paitsi KCR:n vinolla merkkijakaumalla. Sen sijaan vaikuttaisi siltä, että GCL-algoritmin neljä muhkean kokoista lähiesiintymätaulukkoa, joihin talletetaan alustusvaiheessa jo yli 10 miljoonaa kokonaislukua, alkavat yhä selvemmin näkyä hidastuttavina tekijöinä algoritmin suoritusajassa, mutta edelleenkään menetelmän ajoaikoja ei ole häpeämistä: Zipfin jakaumalla GCL ei häviä kovinkaan paljoa esimerkiksi HSZ:lle, jonka esiprosessointi on niukanlainen! Verrattaessa ajoaikojen pituuksia eri merkkijakaumatyyppien välillä havaitaan, että erityisesti HSZ-perheen algoritmit suoriutuvat suhteellisesti huomattavasti lyhyemmässä ajassa tasaista jakaumaa kuin vinoa, Zipfin jakaumaa noudattavista syötteistä. Mukhopadhyayn algoritmilla suoritusaika venyy yli 8-

kertaiseksi siirryttäessä tasaisesta jakaumasta Zipfin jakaumaan. Sama ilmiö oli nähtävissä jo aikaisemmissa kuvissa – joskaan ei ihan näin korostetusti.



Kuva 9.10: *Tarkkojen PYA-algoritmien suoritusajat, kun m, n = 10 000, σ = 256, p ≈ 5 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*)*. Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

Kuvassa 9.10 PYAn prosenttiosuus on pidentynyt edellisen kuvan 30 %:sta 50 %:iin. Parhaimmin tällä syöteaineistolla toimivat molemmilla merkkijakaumilla samat viisi algoritmia kuin edellisessäkin kuvassa. *Tasaisella jakaumalla* voiton vie MUK tasapäin HD2:n kanssa kannoillaan HSZ ja KCR. *Zipfin jakaumalla* nopeimmiksi menetelmiksi osoittautuvat puolestaan KCR ja RI1. Korkeuskäyrä kerrallaan prosessoivat menetelmät vaikuttavat tälläkin testiaineistolla olevan kautta linjan tehottomia. Syötejonojen loppuliitteitä laajentavat algoritmit ovat 50 %:n osuudella edelleen liian hitaita, jotta niitä kannattaisi soveltaa esimerkin kaltaisille syötejonoille. Sen sijaan on huomion arvoista WMM:n ajoajan supistuminen runsaaseen puoleen kuvan 9.9 vastaavista arvoista. PYAn osuuden kasvaessa yli 50 prosentin näyttäisi lyhimmän editointietäisyyden laskevista menetelmistä — etenkin WMM:stä — tulevan verrattain pian kilpailukykyisiä.



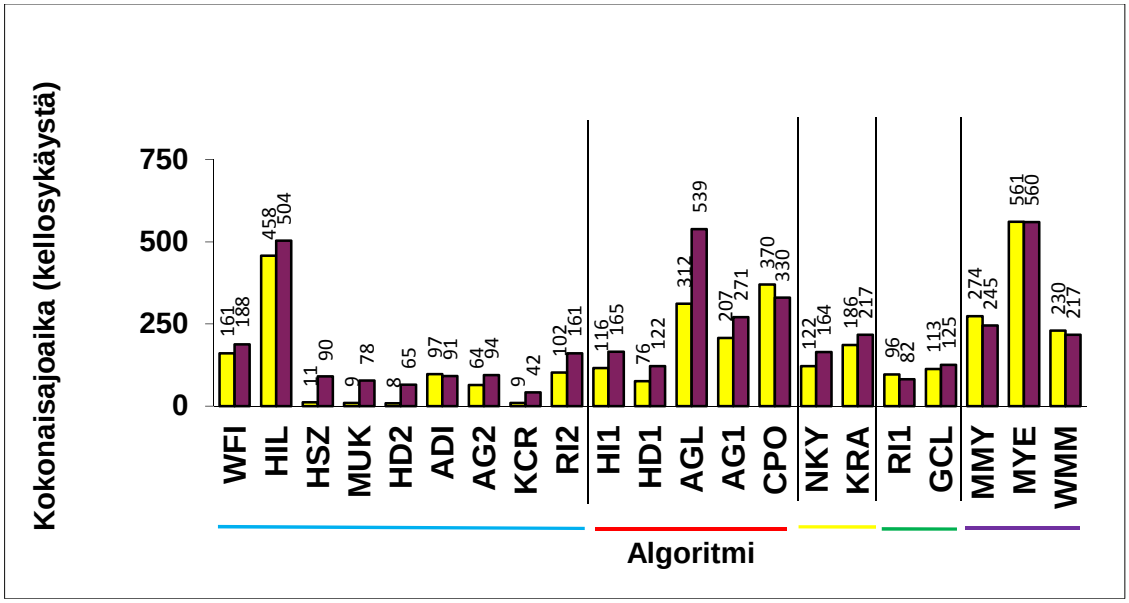
Kuva 9.11: *Tarkkojen PYA-algoritmien suoritusajat, kun m, n = 10000, σ = 256, p ≈ 9 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*). *Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

Viimeisenä testiajona keinotekoisesti generoiduille yhtä pitkille syötteille tarkastellaan kuvassa 9.11 tilannetta, jossa sekä syöttöaakkosto (σ = 256) että PYA-osuus (90 %) ovat isot. Kovin yllättäviä havaintoja ei tästä kuvasta ole tehtävissä, ellei sellaisena halua mainita korkeuskäyrittäisten menetelmien lähes systemaattisesti lyhyempiä suoritusaikoja Zipfin jakaumaa noudattavilla syötemerkkijonoilla tasaisen jakauman suoritusaikoihin verrattuna. Lyhimmän editointietäisyyden laskentaa varten kehitetyt algoritmit ovat hyvin nopeita molemmille jakaumatyypeille, ja WMM on menetelmistä nopein. Riveittäisistä menetelmistä MUK, HD2, HSZ ja KCR menestyvät *tasaisella jakaumalla* myös varsin hyvin, mutta *Zipfin jakauma* lisää niiden työmäärää sen verran, että jakaumiin ilmeisen neutraalisti reagoiva RI1 voittaa edellä mainitun nelikon syötemerkkien jakauman ollessa vino. Samoin syötejonojen loppuliitteitä laajentava *Kumarin ja Ranganin algoritmi* sekä *Rickin II algoritmi* menestyvät Zipfin jakaumalla HSZ-perheen algoritmeja ja HD2:ta paremmin. Neljän suurikokoisen lähiesiintymätaulukon alustamisestaan huolimatta GCL:n suoritusajat ovat kaikkea muuta kuin huonoja, joskin ero hyvin nopeisiin lyhimmän editointietäisyyden laskeviin menetelmiin on huomattava. Korkeuskäyrä kerrallaan laskentaa suorittavat menetelmät voisi luokitella ominaisuuksiltaan kategorisesti onnettomiksi kuvan 9.11 kaltaisille syötteille. Etenkin AGL:n suoritus kestää kohtuuttoman pitkään.

Keskenään eripituiset syötejonot

Ennen siirtymistä käsittelemään luonnollista kieltä noudattavia syöteaineistoja tarkastellaan vielä yhtä testitapausta, jossa syötejonot *X* ja *Y* ovat *keskenään eripituiset*. PYAn osuudeksi asetettiin 30 % kuten kuvassa 9.9, mutta nyt *X*:n pituus puolitettiin

vain 5 000 merkin mittaiseksi *Y*:n pituuden pysyessä ennallaan 10 000 merkissä. Samalla PYAn absoluuttinen pituus lyheni noin 3 000 merkistä likimain 1 500 merkkiin. Seuraavaksi nähtävä kuva 9.12 esittää mainituille syötteille tehtyjen testiajojen tuloksia.



Kuva 9.12: *Tarkkojen PYA-algoritmien suoritusajat, kun m = 5000, n = 10000, σ = 256, p ≈ 1 500 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*). *Algoritmit on ryhmitelty pystyviivoin ja ryhmän alapuolelle merkityin värein taulukon 9.1 mukaisesti.*

Verrattaessa kuvan 9.12 esittämiä tuloksia kuvan 9.9 tuloksiin ei ole havaittavissa mitään kovin dramaattisia muutoksia. Useimpien algoritmien suoritusajat ovat lyhentyneet vähintään puolella kuvan 9.9 mukaisesta tilanteesta. Koska tarkasteltavia rivejä on kuitenkin vain puolet kuvan 9.9 asetelmaan verrattuna, ajoaikojen puolittuminen on hyvin ymmärrettävissä. Erityisen hyvin havainto pitää paikkansa *rivi kerrallaan* prosessoiville menetelmille, joista parhaan nelikon HD2, MUK, KCR ja HSZ ajoajat pyörivät 10 yksikön tuntumassa. Samat neljä menetelmää menestyivät *tasaisella jakaumalla* parhaiten myös kuvassa 9.9. *Zipfin jakaumalla* nelikon sisäinen järjestys säilyy, mutta KCR osoittautuu nyt muita selkeästi nopeammaksi: toiseksi paras HD2 vaatii Kuon ja Crossin algoritmiin verrattuna 1½-kertaisen suoritusajan.

Myös *korkeuskäyrittäin* prosessoivat menetelmät tehostuvat selkeästi jonon *X* lyhenemisen myötä, joskin niiden ajoajat ovat lähinnä HD1:tä lukuun ottamatta melko vaatimattomia. *Syötejonojen loppuliitteitä laajentavista* algoritmeista NKY tehostuu erittäin voimakkaasti: menetelmän ajoaika kutistuu alle viidesosaan kuvan 9.9 tilanteesta. Havainto lienee ainakin osaksi selitettävissä sillä, että kun jonon *X* pituus on puolittunut, joudutaan *X*:n kursoria siirtämään kohti jonon alkua vähempiä kertoja yhden diagonaalin käsittelyn aikana kuin ilman *X*:n pituuden puolittamista. Lisäksi ratkaisu löydetään nyt jo noin 3 500 diagonaalin tutkimisella, kun niitä kuvan 9.9 asetelmassa olisi tutkittu suunnilleen 7 000.

Poikkeukselliselta vaikuttavan mittaustuloksen saa aikaan *monisuuntaisesti* prosessoiva RI1-algoritmi, jolla ajoaika tasaisella jakaumalla muista algoritmeista poiketen pitenee jonkin verran verrattuna kuvaan 9.9. Näin käy siitäkin huolimatta, että vektorin *X* suuntaisen lähiesiintymätaulukon perustamiskustannus on halvempi kuin aikaisemmassa testiajossa. Tulos on parhaiten ymmärrettävissä siten, että alimpiin täsmäysluokkiin kertyy nyt enemmän edustajia kuin syötejonojen ollessa keskenään yhtä pitkät, mikä lisää linkkivektorien päivitystyötä. Sen sijaan Zipfin jakaumalla sama ilmiö ei toistu, ja tuolloin RI1:stä tulee jo neljänneksi paras menetelmä heti KCR:n, HD2:n ja MUK:n jälkeen. Niukanlaiseen dominanttien täsmäysten kirjanpitoon perustuva GCL sen sijaan toimii tarkastelluilla syöteaineistoilla kummallakin jakaumalla selvästi tehokkaammin kuin kuvassa 9.9: linkkivektorin päivitysten lisääntynyt tarve tasaisella jakaumalla ei häiritse GCL:n toimintaa. Sitä vastoin neljän isokokoisen lähiesiintymätaulukon perustaminen rapauttaa jonkin verran menetelmän tehoa, kuten muissakin paraikaa tarkasteltavan aliluvun testitapauksissa.

Odotettuun tapaan käyttäytyvät myös alun perin *lyhimmän editointietäisyyden* laskentaa varten kehitetyt kolme algoritmia. Niiden ajoajat lyhenivät noin puolella kuvan 9.9 mukaisiin tuloksiin verrattuna. Tästä huolimatta kyseiset menetelmät eivät sijoittuneet kärkeen myöskään kuvan 9.12 testiajossa: ainoastaan 30 %:n PYA laajentaa näiden algoritmien tarkasteleman alueen liian väljäksi, jotta laskenta voisi edetä tehokkaasti.

9.2.2 Luonnollisen kielen syöteaineistot

Edellisten alilukujen testiajoja varten tuotettiin keinotekoisesti generoimalla syöteaineistoja, joiden aakkoston koko, merkkijakauman tyyppi ja PYAn prosentuaalinen osuus pystyttiin ennalta kiinnittämään toivotun kaltaisiksi. Kyseisten parametrien arvoja tarkoituksenmukaisesti säätämällä saatiin jo vähintäänkin suuntaa-antava käsitys eri PYA-algoritmien käyttökelpoisuudesta mainittujen parametrien eri yhdistelmillä. Edellä käytettyjä keinotekoisesti muodostettuja syötejonopareja yhdistää kuitenkin systemaattisesti piirre, että *molemmat syötejonot noudattavat keskenään aina samanmuotoista merkkijakaumaa*. Käytännön sovelluksissa tilanne ei kuitenkaan aina ole tällainen, vaan syötejonojen merkkijakaumat voivat olla melko lailla toisistaan riippumattomia.

9.2.2.1 Kumpikin syötejono samasta dokumentista

Jotta saataisiin tuntumaa myös siihen, miten eri menetelmät käyttäytyvät *kahdelle keskenään verrattain heterogeeniselle syötejonolle*, muodostettiin kaksi 10 000 merkin mittaista syötejonoa luonnollista kieltä noudattavasta *suomenkielisestä*

harjoitustyöraportista [Ber08], jossa esiintyy paikoitellen myös *hollanninkielisiä* sanoja. Tekstissä kummassakin syötejonossa yleisimmin esiintyvät merkit olivat *blanko* sekä pieni *a*-kirjain, joiden kumulatiivinen frekvenssi oli noin 20 % kaikista merkeistä. Seuraavassa esitetään näyte merkkijonojen sisältämistä merkeistä[162].

3. Spoorloos

Toisena esitetty elokuva Spoorloos (suom. ”Jäljettömiin”) on kuvaus nuoresta amsterdamilaisesta miehestä nimeltä Rex Hofman (Gene Bervoets) ja naisesta nimeltä Saskia Wagter (Johanna ter Steege), jotka lähtevät yhdessä viettämään kesälomaansa Ranskaan vanhalla Peugeot 404 -henkilöautolla, jonka kattotelineelle on asetettu molempien polkupyörät.

tunnelissa, koska Rex ei Saskian varoituksista huolimatta tankkaa autoa ajoissa. Pariskunnan kesken syntyy riita, ja Rex lähtee kävellen hakemaan polttoainetta irtosäiliöllä. Saskia jää yksinään peloissaan odottelemaan auton luo, ja tunnelissa ilman varoituskolmiota seisova auto aiheuttaa vaaratilanteita. Pitkänpuoleisen odottelun jälkeen matka jatkuu aluksi kinastellen, mutta saavuttuaan aikanaan isolle ranskalaiselle huoltoasemalle Rex ja Saskia sopivat riitansa, ja pitävät siellä leppoisan tauon.

Kuva 9.13: *Näyte testiajoissa käytetystä ensimmäisestä luonnollisen kielen aineistosta. Poikkiviivan yläpuolella on nähtävissä X:n ja alapuolella Y:n sisältöä.*

Merkkijonojen muodostamisperiaate oli varsin yksinkertainen: dokumentin järjestyksessä 10 000 ensimmäistä merkkiä kopioitiin syötevektoriin *X* ja seuraavat 10 000 syötevektoriin *Y*. Myös muut kuin kirjainmerkit – numerot, väli- ja erikoismerkit sekä blankot – hyväksyttiin jonoihin mukaan. Isot ja pienet kirjainmerkit tulkittiin *eri merkeiksi*. Merkkijonojen yhteisen syöttöaakkoston kooksi määräytyi lopulta 75, ja PYAn pituudeksi saatiin 4 061 eli runsaat 40 % syötejonojen pituudesta. Testiajo toistettiin samalla aineistolla kullekin algoritmille 5 kertaa mittausvirheiden tasapainottamiseksi, ja sen tulokset ovat nähtävissä seuraavassa taulukossa 9.2. Taulukosta käy ilmi myös eri menetelmien muistinkulutus. Algoritmit on taulukossa ryhmitelty toimintatapansa mukaan riveittäin, korkeuskäyrittäin ja diagonaaleittain toimiviin, monisuuntaisesti prosessoiviin ja lyhimmän editointietäisyyden laskentaan alun perin kehitettyihin, järjestyksessä ylhäältä alaspäin. Lihavoidut pitkät poikkiviivat toimivat ryhmien välisinä erottimina.

[162] Tilan säästämiseksi useat perättäiset rivinvaihdot on hävitetty kuvan tekstinäytteestä.

Taulukko 9.2: *Tarkkojen PYA-algoritmien suoritusajan ja muistinkulutuksen vertailu, kun m, n = 10 000, σ = 75, p = 4 061 ja syötejonot ovat peräisin yhdestä ja samasta luonnollista kieltä sisältävästä dokumentista. Parhaimmat tulokset on merkitty mitalien (**kulta**, **hopea**, **pronssi**) värein, huonoin mustalla varjostuksella.*

Algoritmi	*Suoritusaika*	*Muistinkulutus (kilotavua)*
WFI	*457*	*390 742*
HIL	*1 148*	*366*
HSZ	*286*	*53 516*
MUK	*243*	*24 975*
HD2	*200*	*47 969*
ADI	*178*	***199***
AG2	*263*	*36 212*
KCR	***155***	*52 344*
RI2	*432*	*69 532*
HI1	*442*	***393 673***
HD1	*349*	*47 227*
AGL	***2 175***	*828*
AG1	*579*	*58 361*
CPO	*806*	*50 119*
NKY	*1 681*	*195 410*
KRA	*735*	***356***
RI1	***167***	*39 103*
GCL	***175***	*11 895*
MMY	*484*	*81 133*
MYE	*735*	***156***
WMM	*338*	*28 479*

Taulukkoon 9.2 tallennettuja suoritusaikoja ei pystytä suoraan vertaamaan mihinkään kuvien 9.1 – 9.12 suoritusajoista, sillä PYA-osuutta 40 %, syöttöaakkoston kokoa 75 eikä pareittain heterogeenista merkkijakaumaakaan esiinny missään aikaisemmista testiajoista. Suuria yllätyksiä taulukon esittämistä suoritusajoista ei kuitenkaan paljastu. Mitalien värein korostettu kärkikolmikko KCR, RI1 ja GCL on tullut esiin jo aikaisemmissakin testiajoissa vaihtelevassa järjestyksessä. Nytkin rivi kerrallaan prosessoivat menetelmät osoittautuvat yleisesti ottaen korkeuskäyrämenetelmiä tehokkaammiksi, vaikka PYA-osuus on esimerkkisyötteillä lyhyt. Alhainen PYA-osuus heikentää myös syötteiden loppuliitteitä laajentavia menetelmiä. Lyhimmän editointietäisyyden laskevista algoritmeista WMM saavuttaa varsin siedettävän tuloksen. AGL osoittautuu jälleen kerran erittäin tehottomaksi menetelmäksi, mitä symbolisoi mustalla korostettu solu ajoaikasarakkeessa.

Tarkasteltaessa menetelmien muistinkulutusta havaitaan, että lineaaritilaiset menetelmät erottuvat selvästi muista edukseen. Voittajaksi selviytyy lyhimmän editointietäisyyden laskentaan kehitetty *Myersin algoritmi*, jota seuraavat ADI, KRA, HIL ja AGL. Matkaa kuudenteen, myös lineaaritilaiseen GCL:ään kertyy jo melkoisesti, sillä GCL tarvitsee tunnetusti avukseen neljä lähiesiintymätaulukkoa. Silti se voittaa selvästi tilantarpeensa puolesta vertailun kaikki ei-lineaaritilaiset algoritmit. Niistä parhaiten menestyvien MUK:n ja WMM:n muistinkulutus on jo yli 150-kertainen verrattuna parhaaseen MYE:hen, ja muistintarve lineaaritilaisista eniten muistia varaavaan GCL:äänkin verrattuna on lähes kaksinkertainen. Tehottomimmin muistia ei odotusten vastaisesti nytkään käytä naiivi WFI, vaan Hirschbergin I algoritmi vaatii muistia vielä tätäkin enemmän. Huomion arvoista on myös, että lähiesiintymätaulukkoja käyttävien RI1:n, AG2:n ja CPO:n muistinkulutus on varsin kilpailukykyinen muihin menetelmiin paitsi lineaaritilaisiin verrattuna.

9.2.2.2 Syötejonot kahdesta eri dokumentista

Luonnollisen kielen mukaisille syötejonoille tehtiin lisäksi myös *toinen testiajo* (kts. taulukko 9.3). Tälläkin kertaa kummankin syötejonon pituudeksi valittiin 10 000 merkkiä, mutta nyt jono *X* täytettiin *eri tekstidokumentista saaduilla merkeillä kuin Y*. Kumpikin suomeksi kirjoitettu dokumentti [Hal08][Kal07] käsitteli samaa aihetta: *toriumin käyttämistä ydinpolttoaineena*. Ensimmäisen dokumentin alusta kopioitiin vektoriin *X* 10 000 ensimmäistä merkkiä alkuperäisessä järjestyksessään, ja toisen dokumentin alusta vietiin sama määrä merkkejä vastaavalla tavalla vektoriin *Y*. Testissä haluttiin tutkia, miten syöttöaakkoston koon pienentäminen heijastuu saavutettuihin mittaustuloksiin; aakkostoa saatiin pienennettyä *samaistamalla kunkin aakkossymbolin isot ja pienet kirjaimet keskenään*. Tällä tavoin päästiin edellä esitettyä 75 merkin aakkostoa pienempään aakkoston kokoon 51. Sen sijaan PYAn pituus ei juurikaan muuttunut taulukon 9.2 mukaisesta tilanteesta: PYAn pituudeksi saatiin nyt 4 142 eli PYA piteni 81 merkillä. Syötejonojen merkkijakaumasta tuli odotetusti jälleen vino. Tälläkin kertaa pareittain yleisimmäksi merkiksi osoittautui välilyönti, jota esiintyi molemmissa syötejonoissa vähintään 924 kertaa. Vain hieman blankoa harvemmin esiintyi tekstissä kirjain *t*, jonka pareittaiseksi minimifrekvenssiksi saatiin 909. Seuraavassa on nähtävissä ote syötejonojen sisällöstä. Ote syötejonojen sisällöstä on nähtävissä taulukkoa 9.3 edeltävässä kuvassa 9.14.

torium ydinpolttoaineena

Jarmo Kalilainen

maailman uraanivarojen vähentyessä on alettava kiinnittää huomiota myös vaihtoehtoisiin fissioenergian lähteisiin. toriumia voidaan pitää vaihtoehtona uraanille. luonnosta löytyvä torium on fertiiliä ja siitä voidaan neutronia absorption kautta konvertoida fissiiliä ydinpolttoaineeksi sopivaa 233U:a.toriumissa on uraaniin nähden monia hyötyjä, kuten runsaammat polttoainevarat sekä torium-polttoainesyklin mahdollistama proliveraation esto. torium tuo mukanaan kuitenkin myös uusia ongelmia ja haasteita polttoainesykliin, kuten konversiossa syntyvän väliytimen protaktiniumin pitkä puoliintumisaika, ja sivutuot

torium - vaihtoehtoinen ydinpolttoaine

nykyiset ydinpolttoaineet

ydinreaktoreissa nykyisin käytetty polttoaine koostuu uraanin isotoopeista 235u ja 238u, joista 235u on fissiili eli halkeava ja 238u fertiili eli hyötämiskelpoinen. luonnosta löytyvä uraani koostuu pääosin isotoopista 238u, isotoopin 235u pitoisuus on vain noin 0,7 %. yleisimmän reaktorityypin eli kevytvesireaktorin polttoaineena käytetään uraania, jota on ketjureaktion ylläpitämiseksi väkevöity 235u-isotoopin suhteen siten, että sen osuus on 3

Kuva 9.14: *Näyte testiajoissa käytetystä toisesta luonnollisen kielen aineistosta. Poikkiviivan yläpuolella on nähtävissä X:n ja alapuolella Y:n alkupää.*

Taulukko 9.3: *Tarkkojen PYA-algoritmien suoritusajan ja muistinkulutuksen vertailu, kun m, n = 10 000, σ = 51, p = 4 142 ja syötejonot ovat peräisin luonnollista kieltä sisältävistä kahdesta eri dokumentista. Parhaimmat tulokset on merkitty mitalien värein, huonoin mustalla varjostuksella.*

Algoritmi	***Suoritusaika***	***Muistinkulutus (kilotavua)***
WFI	*450*	*390 742*
HIL	*1 152*	***355***
HSZ	*274*	*50 235*
MUK	*228*	*23 442*
HD2	*194*	*45 508*
ADI	*178*	***193***
AG2	*258*	*33 282*
KCR	***151***	*49 414*
RI2	*423*	*64 610*
HI1	*444*	***392 735***
HD1	*337*	*43 633*
AGL	***2 123***	*793*
AG1	*574*	*54 103*
CPO	*851*	*45 587*
NKY	*1 637*	*195 410*
KRA	*739*	*358*
RI1	***163***	*35 236*
GCL	***158***	*8 132*
MMY	*476*	*81 992*
MYE	*714*	***156***
WMM	*329*	*26 448*

Vertailtaessa taulukossa 9.3 esitettyjä tuloksia taulukon 9.2 tulosten kanssa havaitaan, että syöttöaakkoston koon pieneneminen 75:stä 51:een sai aikaan verrattain vähän silmiinpistäviä muutoksia algoritmien välisessä paremmuudessa. Suoritusajoissa mitattuna kärkikolmikko pysyi samana Kuon ja Crossin algoritmin (KCR) osoittautuessa nytkin nopeimmaksi. Toisen ja kolmannen sijan saavuttaneiden GCL:n ja RI1:n keskinäinen järjestys vaihtui edellisen taulukon mukaisesta tilanteesta. Muutokset suoritusajoissa ovat kauttaaltaan melko niukkoja. Ehkäpä voimakkaimpana muutoksena edelliseen taulukkoon verrattuna voidaankin havaita GCL:n muistitilan tarpeen selkeä prosentuaalinen supistuminen: menetelmän muistintarve pieneni yli neljänneksellä. Lähimainkaan yhtä voimakasta muistinkäytön suhteellista tehostumista ei ole havaittavissa muilla lähiesiintymätaulukoita käyttävillä menetelmillä.

Luonnollisen kielen syöteaineistoille tehdyissä kahdessa testiajossa KCR osoittautui kummassakin nopeimmaksi menetelmäksi, mutta sen muistinkulutus on verrattain iso. Muista parhaimmistoon kuuluvista GCL pystyy suoriutumaan tehtävästä selvästi KCR:ää vähäisemmällä työmuistin määrällä, ja myös RI1:n muistintarve on tätä pienempi. Nopeus ja vähäinen muistintarve yhdistyvät kuitenkin ylivoimaisesti parhaiten bittirinnakkaisuuteen perustuvassa ADI-algoritmissa, joka on osoittautunut kaikissa aiemmissakin testiajoissa varsin riippumattomaksi syötejonojen muista ominaisuuksista kuin sen pituudesta. Mikäli syötejonojen aakkoston koosta, täsmäysten määrästä ja merkkijakaumasta ei ole saatavilla ennakkotietoja, tuntuisi ADI tähänastisten mittausten valossa hyvin turvalliselta valinnalta etenkin luonnollista kieltä noudattaville syöteaineistoille.

9.2.3 Yhteenveto testiajoista tarkoille PYA-algoritmeille

Yhteenvetona kuvien 9.1 – 9.12 ja taulukoiden 9.2 – 9.3 testiajoista voidaan tiivistetysti todeta, että lyhimmän editointietäisyyden laskevat menetelmät – ja niistä erityisesti WMM – ovat erinomaisia valintoja silloin, kun PYAn tiedetään olevan hyvin pitkä ja syötejonojen olevan keskenään likimain yhtä pitkiä. Sen kummemmin syöttöaakkoston koolla kuin syötteiden merkkijakaumallakaan ei tuntuisi olevan merkitystä näille menetelmille. Jos puolestaan tiedetään, että syöttöaakkosto on taatusti iso mutta PYAn osuus on tuntematon, tuntuisi järkevältä turvautua johonkin algoritmeista GCL, HD2, HSZ, MUK tai RI1, sillä lyhyt PYA tekee alun perin lyhimmän editointietäisyyden laskentaan kehitetyistä menetelmistä hitaita.

Pieni syöttöaakkosto ja samanaikainen suuri täsmäysten määrä ovat kiusallisia erityisesti HSZ-perheen algoritmeille. Siten GCL ja RI1 vaikuttavat hyviltä vaihtoehdolta tällaisille syötteille, sillä syöttöaakkoston koon ollessa pieni ei ole pelkoa niiden aputietorakenteena tarvittavien lähiesiintymätaulukoiden kokojen kasvamisesta kohtuuttoman isoiksi.

On selvää, että ennemmin tai myöhemmin GCL:n ja RI1:n paljon tilaa ja alustustyötä vaativat lähiesiintymätaulukot vaativat resursseja kohtuuttoman paljon, kun syöttöaakkoston annetaan kasvaa kasvamistaan, mutta hyvin useille käytännön syötteille GCL ja RI1 ovat turvallisia ja usein myös parhaita valintoja, sillä syöttöaakkoston kokoa lukuun ottamatta ne tuntuisivat olevan erittäin joustavia kaikille syötteiden ominaisuuksille. Ne ovat selkeästi parhaita valintoja silloin, kun syötteistä ei ole saatavilla mitään ennakkotietoja, kunhan voidaan olettaa, ettei syöttöaakkosto ole kohtuuttoman iso[163]. Tehtyjen testiajojen perusteella on sen sijaan vähintäänkin kyseenalaista suositella NKY:tä tai korkeuskäyrä kerrallaan prosessoivia menetelmiä CPO:ta lukuun ottamatta[164] sovellettavaksi millekään PYA-ongelman instanssille.

Luonnollista kieltä sisältävät syöteaineistot ovat sikäli kiusallisia, että niiden PYAn pituus ja merkkijakauma ovat toisinaan vaikeasti ennustettavissa etukäteen. Sitä vastoin niiden syöttöaakkoston koko on arvioitavissa kohtalaisen tarkasti. Siten on perusteltua valita luonnollisen kielen mukaisten syötejonojen PYAn ratkaisemiseksi algoritmi, joka ei ole herkkä merkkijakauman eikä PYA-osuuden vaihteluille. Menetelmien teoreettisen taustan ja kuvien 9.1 – 9.12 ja taulukoiden 9.2 – 9.3 havaintojen tukemina *Allisonin ja Dixin*, *Goemanin ja Clausenin sekä Rickin I algoritmi* tuntuisivat tällöin turvallisimmilta vaihtoehdoilta. Erityisesti silloin, kun halutaan päästä kohtalaisen nopeaan suoritusaikaan ilman pelkoa muistitilan loppumisesta kesken, ADI vaikuttaisi yleisesti turvallisimmalta valinnalta, kun tietoja syötejonojen ominaisuuksista on korkeintaan niukalti saatavilla etukäteen.

9.3 Testiajot heuristiikoille

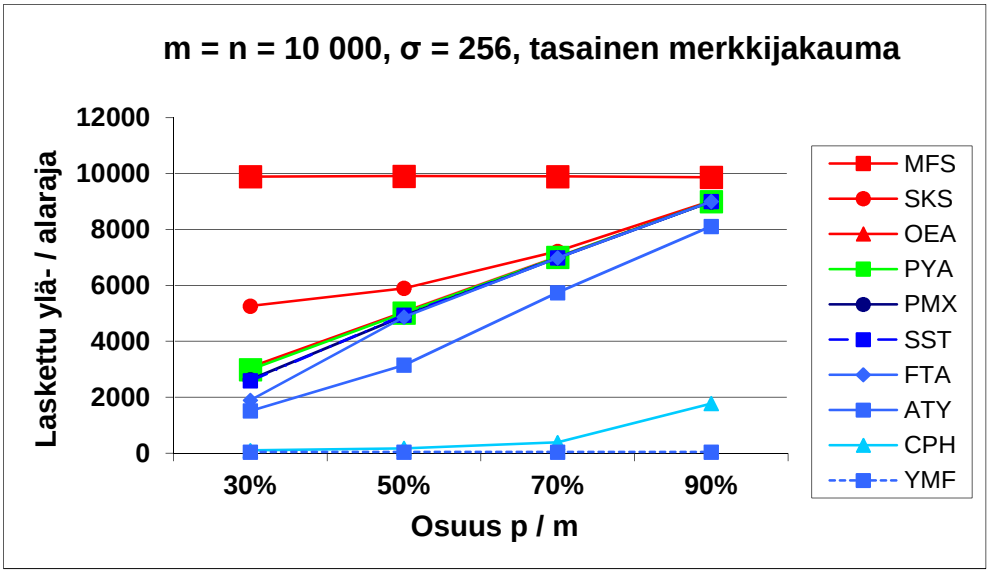
Pisimmän yhteisen alijonon ylä- ja alarajaheuristiikkojen tehokkuutta vertailtiin ensi kertaa tutkimusryhmämme artikkelissa vuodelta 1998 [BHR98, 38–40]. Työn tarkoituksena oli selvittää, miten tarkkoja arvioita PYAn pituudelle erilaisilla heuristisilla menetelmillä saavutetaan, ja paljonko rajojen laskentaan kuluu aikaa tarkkoihin algoritmeihin verrattuna. Vertailuja suoritettiin sekä *tasaista* että *Zipfin jakaumaa* noudattaville keinotekoisille aineistoille ja lisäksi *luonnollista kieltä* sisältäville tekstitiedostoille.

Tässä työssä testiaineistojen syötejonot valittiin 10 000 merkin mittaisiksi kuten edellä tarkkoja PYA-menetelmiä vertailtaessa. Syöttöaakkoston kooksi valittiin keinotekoisesti generoiduille aineistoille 256, ja PYAn osuuden lyhyemmän syötejonon pituudesta annettiin vaihdella 30 prosentista 90 prosenttiin. Testiaineistojen generointi tapahtui samaan tapaan kuin aliluvussa 9.2 esitettiin. Ylärajaheuristiikalle SKS sekä siihen perustuvalle alarajaheuristiikalle SST valittiin kohdeaakkoston kooksi 8, eli aakkoston koko supistui kuvauksessa suhteessa 1/32:aan alkuperäisestä. Vastaavasti

[163] Ainakaan 256 ei osoittautunut vielä kohtuuttoman suureksi syöttöaakkoston kooksi RI1:lle ja GCL:lle.
[164] CPO saattaa soveltua käytettäväksi hyvin silloin, kun syöttöaakkosto on pieni.

ylärajaheuristiikassa OEA ja siihen perustuvassa alarajaheuristiikassa ATY jaettiin alkuperäiset merkit yleisiin ja harvinaisiin siten, että yleisimpien merkkien kategoriaan valittiin 8 merkkiä ja loput 248 merkkiä harvinaisiin. Alarajaheuristiikoista ATY ja SST otettiin vertailuun mukaan ainoastaan *vahvistetut versiot*[165], joiden laskeman alarajan laatu ei milloinkaan ole huonompi kuin vastaavien menetelmien vahvistamattomien versioiden tuottama. FTA-heuristiikkaa suoritettiin testiajoissa rajaparametrilla 1, jolla saavutetaan PYAn pituudelle paras approksimaatio, joskin se näkyy väistämättä myös pidempinä suoritusaikoina. Kaikissa laskennan aikana jotain *tarkkaa PYA-algoritmia* avukseen tarvitsevissa heuristiikoissa[166] on hyödynnetty *Rickin I algoritmia* (RI1).

Kannattaa huomioida, että *tarkkaa menetelmää laskennassa apuna käyttävien alarajaheuristiikkojen* palauttamat tulokset riippuvat tarkan menetelmän valitseman osaratkaisun PYAn sijainnista silloin, kun se ei ole yksikäsitteinen! Esimerkiksi SKS-heuristiikka voi palauttaa täysin erilaisen valetäsmäyksiä sisältävän ratkaisun ylärajalle riippuen siitä, onko tarkkana algoritmina käytetty esimerkiksi RI1:tä vai CPO:ta. Itse ylärajan pituuteen tarkan menetelmän valinnalla ei ole luonnollisestikaan ole merkitystä, mutta sen sijaan valetäsmäysten lukumäärä ja niiden korvaaminen aidoilla täsmäyksillä alarajaa vahvistettaessa riippuvat ylärajan muodostaneiden pisteiden sijainnista syötejonoissa. Tässä työssä ei kuitenkaan ole perehdytty analysoimaan tarkan menetelmän valinnan vaikutusta heuristiikan toimintaan syvemmin, vaan siirryttäköön seuraavaksi tarkastelemaan PYA-heuristiikoille tehtyjen testiajojen tuloksia.



Kuva 9.15: *Heurististen PYA-algoritmien ylä-/alarajojen laadun vertailua, kun m, n = 10 000, σ = 256, merkkijakauma on tasainen, ja PYAn osuus p / m vaihtelee rajoissa 30 % – 90 %.*

[165] kts. aliluvut 5.2.4.1 (SST) ja 5.2.4.2 (ATY)
[166] Tällaisia heuristiikkoja ovat ylärajamenetelmät OEA ja SKS sekä alarajamenetelmät ATY ja SST.

Tasainen jakauma

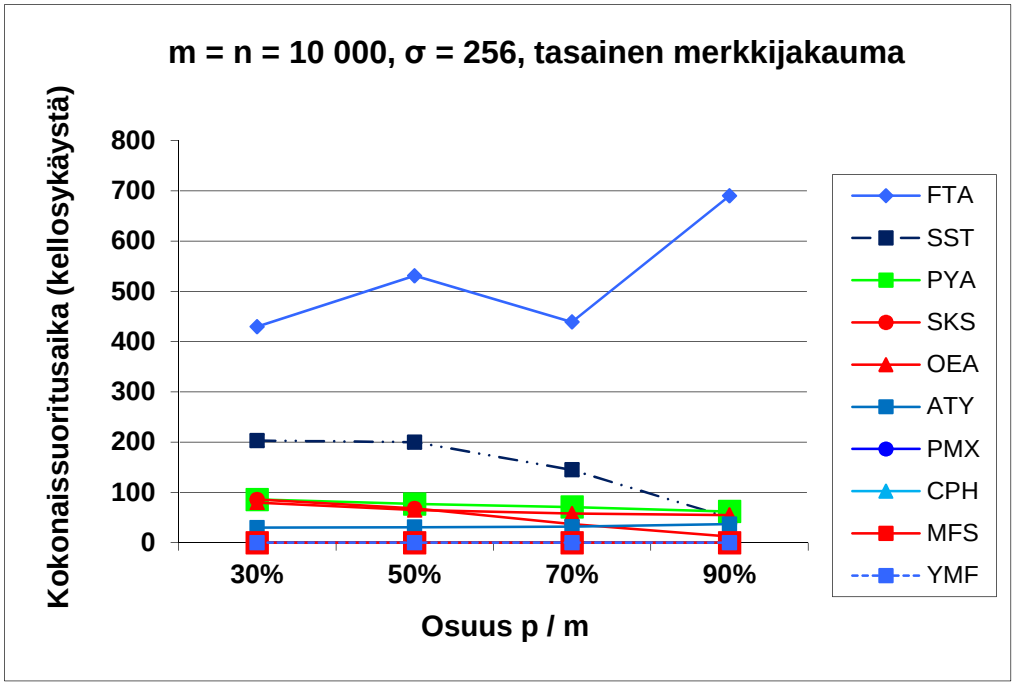
Kuvassa 9.15 on nähtävissä kaikkien luvussa 5 esiteltyjen heuristisiten menetelmien laskemat ylä-/alarajat PYAn pituudelle syöttöaakkoston merkkijakauman ollessa *tasainen*. Ylärajamenetelmiä esittävät kuvassa kolme **punaista** käyrää, tarkan PYAn pituuden osoittaa **vihreä** käyrä, ja **sinisen** eri sävyillä piirretyt kuusi käyrää kuvaavat eri alarajamenetelmillä saavutettuja alarajoja PYAn pituudelle.

Tarkasteltaessa *ylärajamenetelmiä* havaitaan, että merkkien pareittaisten minimifrekvenssien summan laskentaan perustuva MFS-heuristiikka tuottaa lähes triviaalin ylärajan *m* PYAn osuudesta riippumatta. Tulos on hyvin ymmärrettävä, sillä tarkasteltavan tapauksen molemmat syötejonot noudattelevat samanlaista tasaista merkkijakaumaa, joten pareittaiset frekvenssivaihtelut kunkin yksittäisen merkin välillä eri syötejonoissa ovat erittäin vähäisiä. Sama merkkien frekvenssijakauma molemmissa syötejonoissa on MFS:n laadun kannalta epäedullisin mahdollinen syöte. Lähdeaakkoston symbolit esimerkissä kahdeksaan ekvivalenssiluokkaan kuvaava SKS erehtyy valitsemaan pienillä PYA-osuuksilla ratkaisuunsa huomattavan paljon valetäsmäyksiä, mistä syystä esimerkiksi 30 %:n PYA-osuudella ylärajan pituudeksi tulee noin 1.75-kertainen PYAn todelliseen pituuteen verrattuna. Sen sijaan PYAn prosentuaalisen osuuden lähestyessä 70 %:a yläraja tiukkenee hyvin lähelle tarkkaa PYAn pituutta. Tällöin tai pidemmillä PYA-osuuksilla enää harvat SKS:n löytämistä täsmäyksistä ovat tarkkaan ratkaisuun kelpaamattomia eli huteja. Syötteen merkit yleisiin ja harvinaisiin jakava OEA näyttäisi sen sijaan toimivan luotettavasti jo pienilläkin PYA-osuuksilla. Menetelmän kannalta on ratkaisevaa, miten hyvin yleisten ja harvinaisten merkkien osaratkaisut ovat yhdistettävissä toisiinsa. Mitä vähemmän niiden välillä on *kilpailevia* eli keskenään peräkkäin sopimattomia täsmäyksiä, sitä tiukempi ylärajasta tulee.

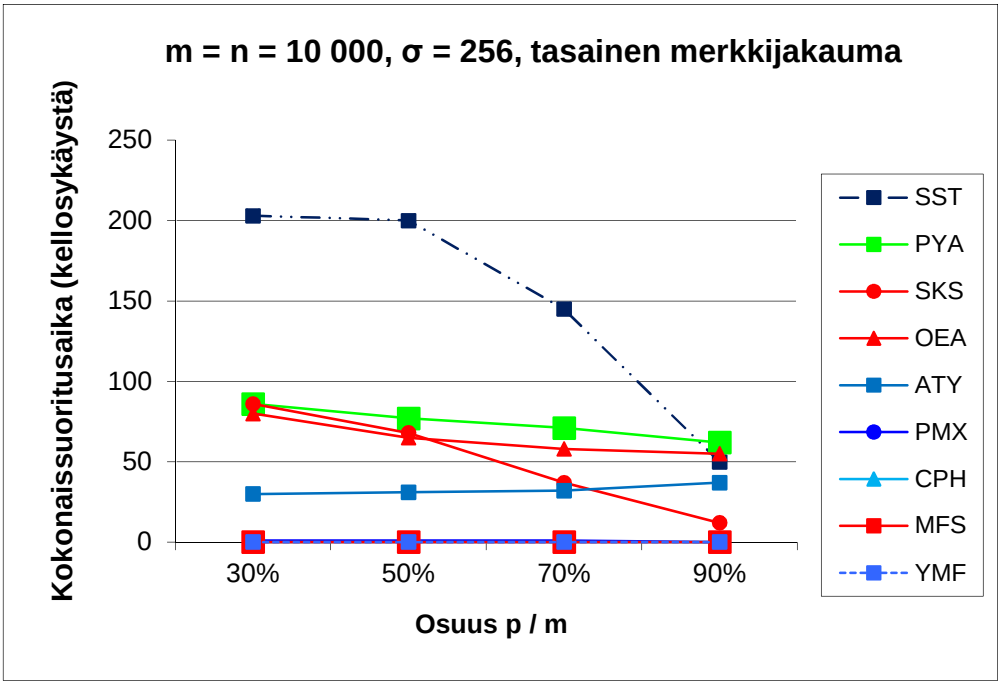
Alarajaheuristiikat näyttäisivät jakautuvan kahteen jyrkästi erottuvaan ryhmään. Hyvin lähelle todellista PYAn pituutta päästään heuristiikalla PMX, joka ei vaadi tarkan PYA-algoritmin kutsumista. Tarkkaa menetelmää avukseen tarvitseva SST tuottaa esimerkkiaineistoilla niin ikään laadukkaan alarajan PYAlle, kuten myös FTA-heuristiikka, jonka laatuun vaikuttavat ratkaisevasti algoritmin tekemien ahnaiden valintojen onnistuminen. Se ajautuu vielä 30 %:n PYA-osuudella selvästi loitommas oikeasta ratkaisusta PMX:ään ja SST:hen verrattuna. Melko lailla näitä heikompi, mutta kuitenkin PYAn pituuden kasvaessa tarkentuva approksimaatio PYAn pituudelle saadaan ATY-heuristiikalla, jonka laskeman alarajan laatu määräytyy ratkaisevasti sen perusteella, miten kahdeksaan yleisimpään merkkiin perustuva alaraja syötteistä muodostetussa matriisissa sijoittuu. ATY-menetelmää rasittaa ilmiselvästi tasainen merkkijakauma molemmissa syötejonoissa, sillä tuolloin kaikki merkit esiintyvät likimain yhtä suurella todennäköisyydellä, jolloin merkkien luokittelu ”yleisiin ja harvinaisiin” on pikemminkin hiusten halkomista kuin oikean kuvan tilanteesta antavaa. Sen sijaan *frekvenssiperustaiset yksinkertaiset alarajamenetelmät* YMF ja CPH osoittautuvat esimerkkiaineistossa kelvottomiksi. Koska syöttöaakkosto on iso ja

merkkijakauma on tasainen, on selvää, ettei pelkkää yleisintä merkkiä tarkastelemalla (YMF) saada kunnollista alarajaa PYAn pituudelle, sillä kunkin merkin teoreettinen esiintymistodennäköisyys on 1/256. Samat taustatekijät ovat syynä myös Chinin ja Poonin CPH-heuristiikan epäonnistumiseen. CPH:n löytämän alarajan pituutta kuvaava käyrä sulautuu lähes yhteen YMF-heuristiikan käyrän kanssa pienillä prosenttiosuuksilla. Kuitenkin 90 %:n PYA-osuudella CPH erottuu jo selvästi edukseen YMF:stä, mutta edelleenkin sen löytämä osuus tarkan PYAn pituudesta jää vaatimattomaksi.

Edellisen kuvan perusteella ilmenee tiivistetysti, että erityisesti PMX ja SST tuottavat PYAn pituudelle hyvän alarajan ja vastaavasti OEA hyvälaatuisen ylärajan, kun syötteiden merkkijakauma on tasainen ja syöttöaakkoston koko on 256. Nyt olisi seuraavaksi mielenkiintoista selvittää, miten paljon rajojen laskentaan kuluu aikaa. Ajoaikoja esittävään seuraavaan kuvaan 9.16a on otettu mukaan myös vastaavanlaisella syöteaineistolla tarkan PYA-algoritmin (merkitty ”PYA”) Rickin I algoritmin (RI1) suoritusaika. Kyseinen menetelmä päätettiin ottaa verrokkimenetelmäksi sen ansiosta, että se toimii verrattain tehokkaasti ja ilman jyrkkiä ajoajan heilahteluja syötejonojen merkkijakauman ja PYAn prosentuaalisen osuuden vaihdellessa. Testiajo samaa aineistoa käyttäen suoritettiin *viidesti peräkkäin* mittausvirheiden tasapainottamiseksi aivan kuten aliluvun 9.2 testiasetelmissa. Tulokset ovat mitattujen ajoaikojen keskiarvoja. Koska yhden alarajamenetelmän – FTA:n – ajoajat ylittävät voimakkaasti muiden menetelmien suoritusajat ja siten vääristävät niiden keskinäisiä vertailuja, vastaava testausasetelma ilman FTA-menetelmää toistetaan heti perään kuvassa 9.16b.

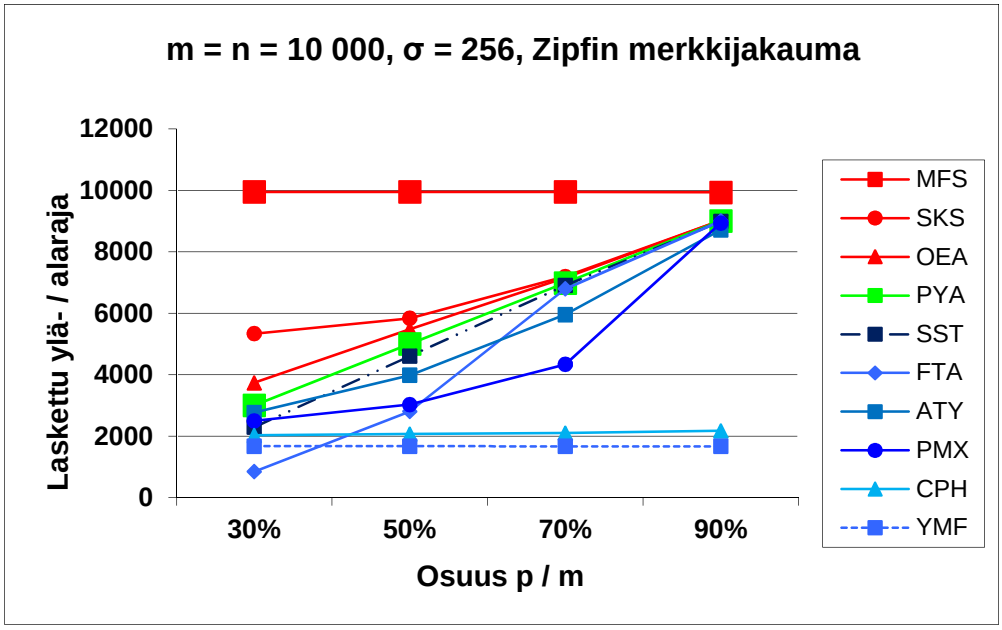


Kuva 9.16a: *Heuristisen PYA-algoritmien ajoaikojen vertailua, kun m, n = 10 000, σ = 256, merkkijakauma on tasainen, ja PYAn osuus vaihtelee rajoissa 30 % – 90 %.*



Kuva 9.16b: *Heurististen PYA-algoritmien ajoaikojen vertailua, kun m, n = 10 000, σ = 256, merkkijakauma on tasainen, ja PYAn osuus vaihtelee rajoissa 30 % – 90 %. FTA-alarajaheuristiikkaa edustava käyrä on poistettu.*

Kuvasta 9.16a nähdään, että ajoaikojen vaihtelut eri heurististen PYA-menetelmien välillä ovat melkoisia: parhaimpien menetelmien suoritusaikaa kuvaava käyrä on lähes nollan aikayksikön tuntumassa, kun taas alarajamenetelmistä FTA vaatii rajaparametrilla 1 kohtuuttoman paljon suoritusaikaa. Siten muiden menetelmien vertailuja ajatellen kuva 9.16b lienee lukijalle havainnollisempi. *Ylärajamenetelmistä* ainoastaan lähes triviaalin ylärajan palauttanut MFS toimii erittäin nopeasti: alle yhden aikayksikön. Sen sijaan muut ylärajamenetelmät toimivat vain hädin tuskin tarkkaa algoritmia nopeammin. PYAn osuuden ollessa 30 % SKS:n ajoaika jopa sivuaa tarkan RI1-menetelmän ajoaikaa! Syynä tähän on mitä ilmeisimmin täsmäysten lukumäärän lisääntyminen siten, että heuristisen PYAn pituus lähestulkoon saavuttaa heuristiikan taustalla olevan RI1-algoritmin teoreettisen aikavaativuuden mukaisen pahimman tapauksen eli 50 prosenttia *X*:n pituudesta. Aakkoston koon puristuminen 256:sta 32:een vesittyy siten täsmäysten määrän kasvulla. *Alarajamenetelmistä* tarkkaa menetelmää avukseen käyttävä SST tarvitsee edellä mainitulla perusteella parasta tarkkaa algoritmia enemmän suoritusaikaa. Sen sijaan ATY selviytyy tehtävästä tarkkaa menetelmää nopeammin, sillä siinä tarkkaa menetelmää kutsutaan alkuperäistä pienemmille osaongelmille. Sitä vastoin kolmikko YMF, CPH ja PMX alittaa tuntuvasti parhaan algoritmin käyttämän ajan.

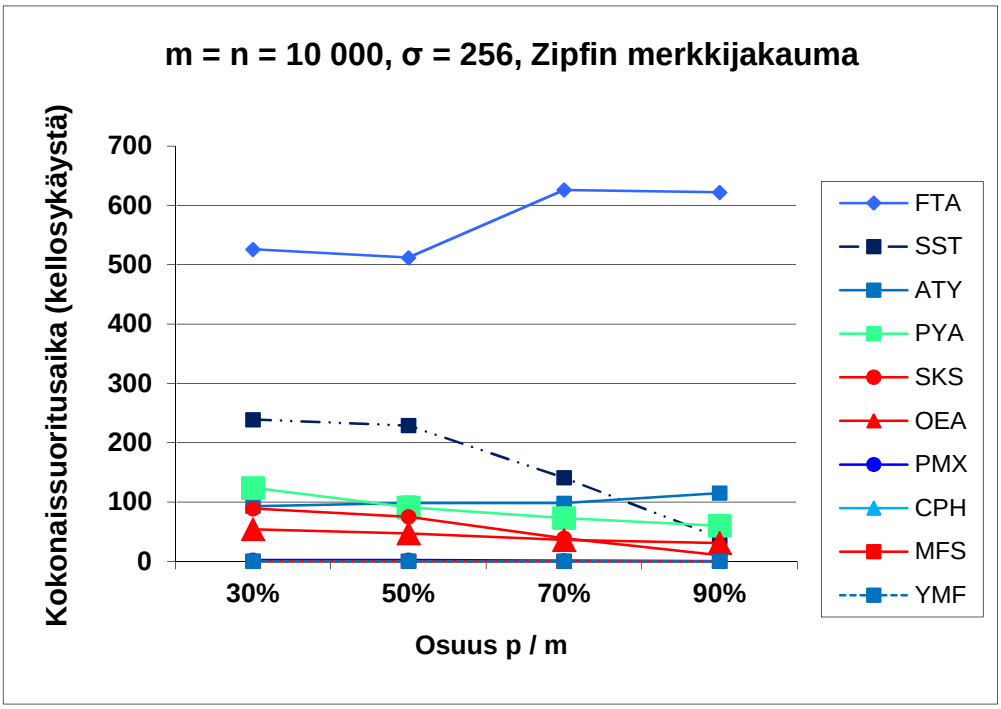


Kuva 9.17: *Heurististen PYA-algoritmien ylä-/alarajojen laadun vertailua, kun $m = n = 10\,000$, $\sigma = 256$, merkkijakauma on Zipfin jakauma, ja PYAn osuus vaihtelee rajoissa 30 % – 90 %.*

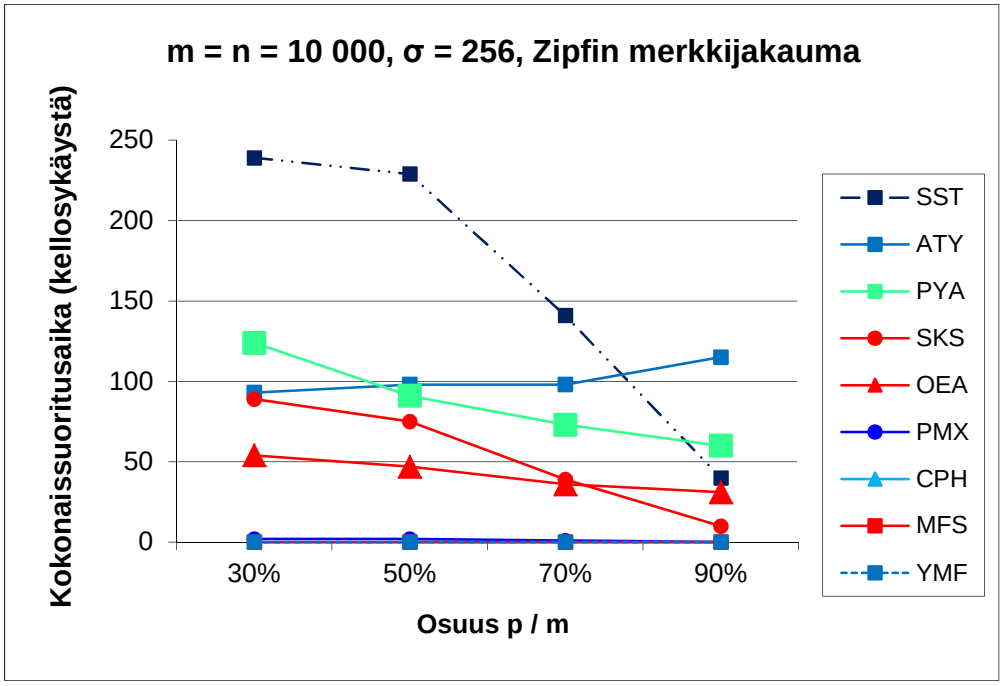
Zipfin jakauma

Kuva 9.17 poikkeaa testiasetelmaltaan kuvasta 9.15 merkkijakaumansa puolesta: se on nyt vaihtunut tasaisesta *Zipfin jakaumaksi*. Koska kuitenkin molemmat syötejonot noudattavat keskenään samaa jakaumaa, ei yksinkertaisin ylärajaheuristiikka MFS nytkään anna tuloksena kelpo ylärajaa, vaan tulos on likimain triviaali 10 000. Myös muut ylärajaheuristiikat käyttäytyvät samantapaisesti kuin kuvassa 9.15. Alarajojen kohdalla on sen sijaan enemmän muutoksia. Koska yleisimmän merkin frekvenssi on nyt huomattavasti isompi kuin tasaisen jakauman $10\,000 / 256 \approx 39$, saavutetaan primitiivisintä alarajaheuristiikkaa soveltamalla likimain 1 600:n mittainen alaraja, joka tosin on edelleen vaatimaton kaikilla PYA-osuuksilla. Myös CPH-heuristiikan alarajan laatu paranee tasaisen jakauman mukaisesta arvostaan, mutta edelleen myös CPH:n laatu on verrattain heikko. FTA:n tekemät ahnaat valinnat eivät onnistu esimerkkiaineistojen pienten PYA-osuuksien kohdalla toivotulla tavalla, mutta 70 %:n PYA-osuudella alarajan laatu paranee huomattavasti ja päästään lähelle tarkkaa arvoa. Zipfin jakauma tuottaa myös PMX:lle selvästi enemmän kiusaa kuin tasainen jakauma, ja sen laskemat alarajat jäävät nyt siten selvästi loitommas tarkoista arvoista. Laadukkaimmat alarajat saavutetaan nytkin vahvistetulla SST-heuristiikalla, ja myös ATY-heuristiikalla päästään kohtalaiseen tarkkuuteen PYA-osuudesta riippumatta. FTA:n onnistuminen valinnoissaan riippuu jälleen erittäin voimakkaasti syöteaineistosta: ensimmäisen diagonaalijakson valinta vaikuttaa ratkaisevasti alarajan laatuun. Siten on ilmeistä, että PYAn ollessa lyhyt heuristiikka joutuu helposti hakoteille. PMX toimii kautta linjan melko luotettavasti, mutta 50 %:n ja 70 %:n PYA-osuuksilla jää kuitenkin verrattain paljon todellisesta PYAn pituudesta löytymättä. Frekvenssiperustaisten CPH:n ja YMF:n laskemat tulokset jäävät edelleen laadultaan

vaatimattomiksi, vaikka ne tosin tuottavatkin Zipfin jakaumalla selvästi pidemmän alarajan kuin tasaisella jakaumalla. Syöttöaakkoston koon ollessa suuri voi YMF toimia lähinnä vain heikkotehoisena indikaattorina syötteiden yhteisestä merkkijakaumasta: selvästi osamäärää m/σ pidempi alaraja vahvistaa kummankin syötejonon jakauman olevan vino, mutta päinvastaisessa tapauksessa ei pystytä päättelemään juuri mitään, sillä pieni alaraja voi kertoa paitsi jakauman tasaisuudesta niin myös siitä, että syötteillä on vain yksinkertaisesti hyvin vähän yhteisiä merkkejä taustalla olevasta jakaumasta riippumatta.



Kuva 9.18a: *Heuristisen PYA-algoritmien ajoaikojen vertailua, kun m = n = 10 000, σ = 256, merkkijakauma on Zipfin jakauma, ja PYAn osuus vaihtelee rajoissa 30 % – 90 %.*



Kuva 9.18b: *Heurististen PYA-algoritmien ajoaikojen vertailua, kun m = n = 10 000, σ = 256, merkkijakauma on Zipfin jakauma, ja PYAn osuus vaihtelee rajoissa 30 % – 90 %. FTA-alarajaheuristiikkaa edustava käyrä on poistettu.*

Kuvat 9.18a ja 9.18b paljastavat lopulta, miten nopeasti heuristiset ylä- ja alarajat ovat laskettavissa, kun syötejonot noudattavat *Zipfin jakaumaa*. Kuvasta 9.18b on jälleen FTA-heuristiikka jätetty pois (kuten kuvasta 9.16b edellä), sillä sen saavuttamat huonot mittaustulokset vaikeuttavat jälleen tarpeettomasti muiden menetelmien välisiä vertailuja. Toisin kuin tasaisella jakaumalla kuvissa 9.16a ja 9.16b, ylärajaheuristiikat ovat nyt kaikki tarkkaa PYA-algoritmia nopeampia. Erityisen nopea on MFS, jonka laatu tosin esimerkkisyötteille oli onneton. Alarajaheuristiikoista tarkkaa PYA-menetelmää avukseen tarvitseva SST on pisintä sekä ATY lyhintä PYA-osuutta lukuun ottamatta kuitenkin tarkkaa RI1-algoritmia hitaampi, kuten kaikilla PYA-osuuksilla myös FTA, jonka suoritusajat rajaparametrin arvolla 1 ovat kohtuuttoman pitkiä: vähintään 500 aikayksikköä. Toiseen ääripäähän sijoittuvat nytkin primitiiviset YMF- ja CPH-heuristiikat kuten myös PMX, jonka laskemat alarajat ovat laadullisesti tuntuvasti edellä mainittuja kahta menetelmää parempia. Heuristiikkojen MFS, YMF, CPH ja PMX ajoaikoja kuvaavat käyrät menevät kuvassa lähestulkoon päällekkäin vain aavistuksen verran nollan yläpuolella.

Luonnollinen kieli

Lopuksi tarkastellaan vielä heuristiikkojen löytämien ylä- ja alarajojen laatua luonnollista kieltä noudattavalle aineistolle. Vertailun tulokset ovat esillä seuraavissa taulukoissa 9.4 ja 9.5. Kyseessä ovat samat aineistot kuin taulukoissa 9.2 ja 9.3 kuvatut, joissa niitä sovellettiin tarkoille PYA-algoritmeille. Taulukon 9.4 lähteenä olleen *Hollanti-Belgia -aiheisen tekstitiedoston* 20 000 ensimmäisestä merkistä muodostettujen

kahden syötejonon merkkijakauma on varsin vino, sillä pelkästään kahta yleisintä merkkiä – *blankoa* ja *pientä a-kirjainta* – esiintyy kummassakin syötejonossa yhteensä jo vähintään 2 166 kappaletta eli yli viidennes kaikista merkeistä.

Taulukko 9.4: *PYA-heuristiikkojen vertailua, kun m, n = 10 000, σ = 75, p = 4 061 ja syötejonot ovat peräisin luonnollista kieltä sisältävästä dokumentista. Taulukko esittää löytyneiden ylä- ja alarajojen pituuksia sekä niiden suhdetta tarkan PYAn pituuteen. Kolme ylintä menetelmää ovat yläraja- ja alimmat kuusi alarajaheuristiikkoja.*

Heuristiikka	***Ala- / ylärajan pituus***	***Osuus PYAn pituudesta***
MFS	***9 355***	***230.6 %***
SKS	*5 596*	*137.8 %*
OEA	***4 880***	***120.2 %***
TARKKA PYA	***4 061***	***100.0 %***
YMF	*1 147*	*28.2 %*
CPH	*2 063*	*50.8 %*
PMX	*3 274*	*80.6 %*
SST	*3 398*	*83.7 %*
ATY	***3 759***	***92.6 %***
FTA	***883***	***21.7 %***

Koska molempien syötejonojen merkkijakaumat ovat kuitenkin verrattain samanlaiset keskenään – jakaumat ovat samalla tavoin vinoja – ei *ylärajamenetelmistä* MFS nytkään pysty saamaan aikaan laadukasta arviota PYAn pituudelle, vaan sen muodostama yläraja 9 355 on pituudeltaan yli kaksinkertainen todelliseen PYAan verrattuna. Selkeästi tätä parempaan approksimaatioon päästään merkkejä eri ekvivalenssiluokkiin osittavalla SKS- ja erityisesti yleisyyden mukaan merkkejä luokittelevalla OEA-heuristiikalla, jonka laskema yläraja ylitti tarkan PYAn pituuden noin 20 prosentilla.

Alarajamenetelmistä yksinkertaisin, pelkästään syötevektorien merkkien pareittaisten minimifrekvenssien maksimin etsivä YMF löytää PYAn pituudesta reilut 28 %, joka vastaa syötejonojen sisältämien välilyöntien pareittaisen minimifrekvenssin suhdetta jonojen PYAn pituuteen testiaineistossa. Yksinkertaisuudestaan huolimatta YMF ei kuitenkaan osoittaudu tulokseltaan heikoimmaksi alarajamenetelmäksi, vaan selkeästi pienemmän osuuden PYAn pituudesta löytää FTA-heuristiikka, joka ei tarkasteluilla syötejonoilla pääse onnistuneesti PYAn sijainnin jäljille. Frekvenssiperustaisista menetelmistä kehittyneempi, CPH, löytää PYAn pituudesta esimerkissä hieman yli puolet. Keinotekoisesti generoiduilla jonoilla nopeasti toiminut ja laadukkaan alarajan löytänyt PMX-heuristiikka saavuttaa PYAn pituudesta nytkin neljä viidesosaa. Laadultaan tätäkin paremmiksi heuristisiksi alarajamenetelmiksi osoittautuivat ylärajaheuristiikasta SKS kehitetty SST sekä etenkin ylärajaheuristiikasta

OEA kehitetty ATY, joka esimerkkisyötteillä löysi tarkan PYAn pituudesta jopa yli 90 %. Kummastakin viimeksi mainitusta menetelmästä käytettiin *vahvistettua versiota*[167].

Taulukko 9.5 esittää heuristiikkojen toimintaa kahdelle tekstidokumentille, jotka käsittelivät *toriumin käyttämistä ydinpolttoaineena*. Ylärajamenetelmien keskinäinen paremmuusjärjestys ei muutu, ja rajat ovat aavistuksen verran tiukempia kuin edellisen taulukon aineistolle. Alarajat ovat puolestaan hieman väljempiä kuin edellisessä esimerkissä kahta menetelmää — SST:tä ja FTA:ta — lukuun ottamatta. Viimeksi mainitun kohdalla alaraja piteni tuntuvasti, mutta siitä huolimatta sen laatu — vain 38 % tarkasta PYAn pituudesta — on kovin vaatimaton. Parhain alaraja, likimain 90 % PYAn todellisesta pituudesta, saadaan nytkin käyttämällä vahvistettua ATY-heuristiikkaa, joka laskee aluksi alarajan pelkästään kahdeksalle yleisimmälle merkille ja täydentää tätä harvinaisten merkkien osaratkaisun vertailukelpoisilla täsmäyksillä.

Taulukko 9.5: *PYA-heuristiikkojen vertailua, kun m, n = 10 000, σ = 51, p = 4 142 ja syötejonot ovat peräisin kahdesta luonnollista kieltä sisältävästä dokumentista. Taulukko esittää löytyneiden ylä- ja alarajojen pituuksia sekä niiden suhdetta tarkan PYAn pituuteen. Kolme ylintä menetelmää ovat yläraja- ja alimmat kuusi alarajaheuristiikkoja.*

Heuristiikka	***Ala- / ylärajan pituus***	***Osuus PYAn pituudesta***
MFS	***9 161***	***221.1 %***
SKS	*5 544*	*133.8 %*
OEA	***4 901***	***118.3 %***
TARKKA PYA	***4 142***	***100.0 %***
YMF	***924***	***22.3 %***
CPH	*1 671*	*40.3 %*
PMX	*3 272*	*79.0 %*
SST	*3 577*	*86.4 %*
ATY	***3 785***	***91.4 %***
FTA	*1 575*	*38.0 %*

Yhteenveto

Kuvien 9.15 – 9.18 ja taulukoiden 9.4 – 9.5 perusteella todetaan, että selvästi lupaavimmalta heuristiselta menetelmältä tuntuisi *alarajaheuristiikka* PYAMAX (PMX). Sen laskemat alarajat ovat verrattain laadukkaita, joskin ATY:tä ja SST:tä käyttämällä pystytään usein vielä tarkempiin rajoihin. Huomion arvoista on ATY:n selkeästi parempi menestys luonnollisen kielen kuin tasaisen jakauman aineistoille: kahdeksan yleisintä merkkiä peittävät luonnollisella kielellä jo yli selvästi yli puolet kaikista merkkiesiintymistä [ROA09], joten on ilmeistä, että sen myötä ATY:n palauttaman alarajan tarkkuus kasvaa. SST:n ja ATY:n haittapuoleksi koituu kuitenkin tarkan algoritmin tarve niiden laskennassa, mikä tekee niistä käytännössä liian hitaita.

[167] kts. aliluku 5.2.4

Nopeasti prosessoivien YMF:n ja CPH:n laskemien alarajojen laatu on kovin heikko, ja FTA:n käytön sulkee pois menetelmän laskemien rajojen voimakkaasti ailahtelevan laadun lisäksi kohtuuttoman pitkä suoritusaika, jos menetelmän rajaparametriksi valitaan ykkönen. Menetelmä tuntuisi olevan kelvollinen ainoastaan silloin, kun PYAn tiedetään olevan hyvin pitkä, mutta toisaalta kyseisessä asetelmassa myös monet tarkat algoritmit – erityisesti lyhimmän editointietäisyyden laskevat – toimivat hyvin nopeasti.

Ylärajamenetelmistä parhaat eli OEA ja SKS vaativat myös avukseen tarkkaa PYA-algoritmia, mikä hidastaa näiden suoritusta oleellisesti. Hyvin nopea ja yksinkertainen frekvenssiperustainen MFS-heuristiikka taas osoittautuu hyödyttömäksi, jos kumpikin syötejonoista noudattaa edes likimain samankaltaista merkkijakaumaa. Jos kyseinen oletus ei kuitenkaan pidä paikkaansa, niin MFS:n kuten myös yksinkertaisimpien alarajaheuristiikkojen käytettävyys paranee merkittävästi. Saadut testitulokset puoltanevat kuitenkin riittämiin pelkän PMX-heuristiikan ottamista tarkasteluun seuraavassa aliluvussa, jossa sovelletaan heuristista laskentaa osana tarkan PYA-algoritmin toimintaa.

9.4 Testiajot jalostetuille tarkoille PYA-algoritmeille

Seuraavaksi on tarkoitus empiirisillä testiajoilla selvittää, missä määrin tarkkojen PYA-algoritmien suoritusta voidaan tehostaa joko ohjelmointiteknisesti, järkeistämällä algoritmin muistinkäyttöä, lisäämällä sen havainnointia syötejonojen ominaisuuksista sekä liittämällä algoritmin tueksi heuristinen esiprosessointi. Tarkastelu kohdistuu *kolmeen tarkkaan algoritmiin*: NKY:hyn, KCR:ään ja RI1:een. Näitä algoritmeja on päädytty kehittämään siksi, että ne ovat verrattain yksinkertaisia implementoitaviksi ja lisäksi ne eroavat toisistaan laskentamallinsa puolesta. Ensin mainittua muunnellaan vaiheittain hienosäätämällä, jälkimmäisiä puolestaan pelkästään heuristiikan turvin.

9.4.1 Testiajot NKY-algoritmin eri kehitysversioille

Tutkimusryhmämme artikkelissa [BHV03] tarkasteltiin *Nakatsun, Kambayashin* ja *Yajiman* algoritmin tehostamista kuudella erilaisella, toisiaan täydentävällä tavalla: 1) *tiukentamalla diagonaalien käsittelyn silmukkaehtoa ja poistamalla PYAn pituutta kuvaavan muuttujan tarpeettomat päivitykset*, 2) *jakamalla algoritmin uloin silmukka kahteen erilliseen, keskenään loogisesti yhtenäiseen osaan*[168], 3) *uudistamalla algoritmin käyttämän tietorakenteen indeksointi*[169], 4) *käyttämällä hyväksi heuristista*

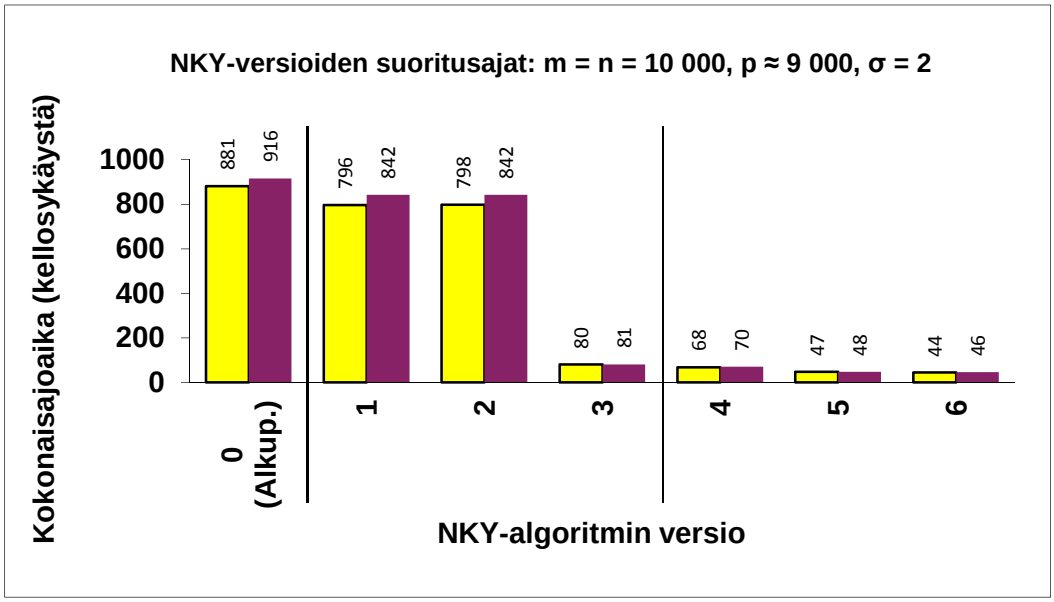
[168] Silmukan alkuosassa pyritään tehostamaan $L_i(k)$-arvoja k:n arvoille 1, 2, …, p_t, missä p_t kuvaa tarkasteluhetkeen mennessä löydetyn PYAn pituutta. Silmukan jälkimmäisessä osassa keskitytään sen sijaan PYAn pidentämiseen arvosta p_t.

[169] Matriisin uudelleenorganisointi on esitelty tämän työn aliluvussa 8.2.1.

esiprosessointia[170], 5) *havaitsemalla perättäisten täsmäysten ketjujen muodostuminen*[171] sekä 6) *rajoittamalla tuloksetonta laskentaa havainnoimalla optimaalisten välitulosten saavuttaminen*[172]. Artikkelissa algoritmia kehitettiin asteittain edellä mainitussa järjestyksessä, eli esiprosessointi upotettiin algoritmiin mukaan vasta kolmanneksi viimeisessä kehitysvaiheessa. Siten juuri esiprosessoinnista saatavaa hyötyä ei pystytty selkeästi mittaamaan. Näin oli kuitenkin järkevää toimia, sillä heuristisen rajojen laskenta tarjoaa mahdollisuuden aputietorakenteena käytettävän *matriisin koon rajoittamiseen* NKY:n artikkelissa alun perin esitetystä, ja rajoittamisen edellytyksenä on sitä tukeva indeksointi. Vastaavasti kaksi ensiksi tehtyä parannusta eivät muuta mitenkään algoritmin peruslogiikkaa, vaan ainoastaan selkeyttävät ohjelmakoodia ja vähentävät turhia muuttujien päivityksiä. Edellä esitetyllä numeroinnilla 1 – 6 viitataan seuraavissa kuvissa NKY:n eri versioihin. Algoritmin alkuperäisversiota edustaa versionumero 0.

9.4.1.1 Syöttöaakkoston koko 2

Kuvasta 9.19 ilmenee, millä tavoin eri tekniset parannukset vaikuttavat NKY-algoritmin suoritusaikaan, kun syöttöaakkoston koko on 2 ja PYAn prosenttiosuus on 90 % syötejonojen pituudesta. Alkuperäisversio on erotettu pystyviivalla ensimmäisistä parannetuista versioista, jotka puolestaan on erotettu toisella pystyviivalla heuristisesti esiprosessoiduista versioista.



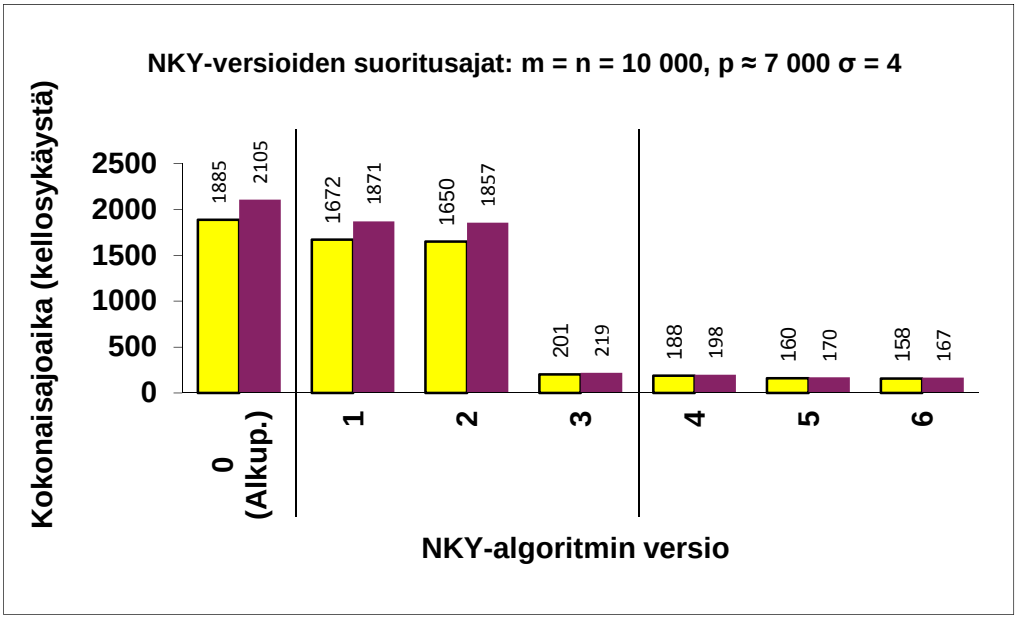
Kuva 9.19: *NKY-algoritmin eri versioiden suoritusajat, kun m, n = 10 000, σ = 2, p ≈ 9 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*).

[170] Heuristisen esiprosessoinnin hyödyntämisen vaikutusta NKY:n suoritusaikaan ja muistinkäyttöön on tarkasteltu aliluvussa 8.2.4.
[171] Perättäisten täsmäysten ketjujen muodostumisen havainnointia on käsitelty aliluvussa 8.2.3.
[172] Optimaalisten loppuliitteiden löytymistä on selvitetty aliluvussa 8.2.2.

Heti ensimmäisen kehitysvaiheen vähäisiltä vaikuttavat muutokset, eli algoritmin silmukkaehdon tiukentaminen ja PYAn pituutta edustavan muuttujan turhien päivitysten karsiminen lyhentävät menetelmän ajoaikaa jo noin 10 % tasaisella jakaumalla. Zipfin jakaumalla muutos jää hieman vähäisemmäksi. Vaiheella 2, jossa NKY:n uloin silmukka ositetaan kahteen toiminnallisesti erilaiseen tarkoitukseen, ei tunnu esimerkkiaineistolle olevan hyötyvaikutusta – itse asiassa tasaisella jakaumalla tulokset ovat niukasti heikompia kuin versiolla 1. Sen sijaan kehitysvaiheen 3 eli matriisin uudelleenorganisoinnin vaikutus algoritmin suoritusaikaan on suorastaan dramaattinen: algoritmin ajoaika lyhenee noin yhteen kymmenesosaan jo vaiheiden 1 ja 2 muutokset sisältävästä parannetusta versiosta, ja ajoaikaa säästyy alkuperäisversioon nähden jopa yli 90 %! Ajoajan nopeutuminen on suurelta osin selitettävissä välimuistioperaatioiden tehostumisella, kun haku- ja päivitysoperaatiot ovat hyvin paikallisia – toisin kuin versioissa 1 – 2. Vaiheiden 4 – 6 eli heuristisen esiprosessoinnin sekä perättäisten täsmäysketjujen ja optimaalisten loppuliitteiden havaitsemisen absoluuttinen vaikutus NKY:n kokonaissuoritusaikaan tuntuu vähäpätöiseltä vaiheen 3 mullistavaan muutokseen nähden, mutta tästä huolimatta nämä vaiheet yhdessä vähentävät vielä yli 40 % version 3 suoritusajasta. Vertailtaessa jalostetuimman version 6 tasaisen jakauman suoritusaikaa 44 ja Zipfin jakauman suoritusaikaa 46 kuvan 9.2 suoritusaikoihin havaitaan, että NKY:n kehittynein versio sijoittuisi algoritmien välisessä vertailussa peräti viidennelle sijalle, ja se häviäisi selkeästi ainoastaan vertailun voittajalle WMM:lle! Alkuperäinen versio NKY:stä olisi puolestaan seitsemänneksi heikoin yhteensä 21 algoritmin vertailussa.

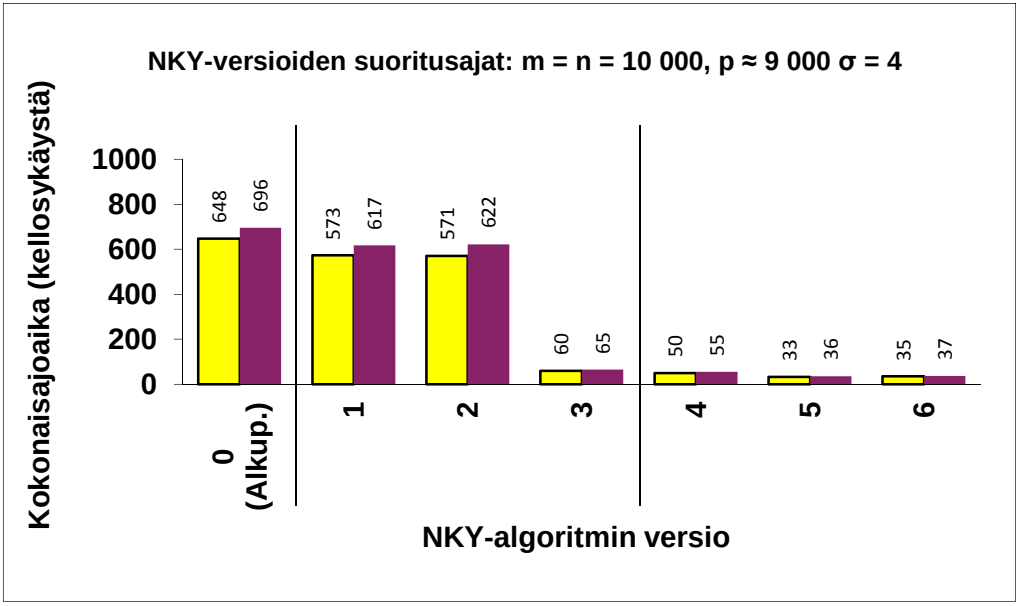
9.4.1.2 Syöttöaakkoston koko 4

Seuraavaksi tarkastellaan kuvassa 9.20, miten tilanne muuttuu aakkoston koon kahdentuessa 2:sta 4:ään ja samalla PYAn osuuden pienentyessä 90 %:sta 70 %: iin syötejonojen pituudesta.



Kuva 9.20: *NKY-algoritmin eri versioiden suoritusajat, kun m, n = 10 000, σ = 4, p ≈ 7 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*).

Kuvan pylväiden korkeuksien muutokset muistuttavat melko lailla edellisen kuvan mukaista tilannetta. Nyt tosin myös vaiheessa 2 ajoajat hieman lyhenevät. Vaiheiden 1 – 2 ajoaikaa lyhentävä vaikutus on nytkin kiistaton, mutta ne eivät riitä tekemään NKY:stä kilpailukykyistä algoritmia annetun tyyppisille syötteille, sillä menetelmän alkuperäisversio oli kuvan 9.3 vertailussa jakaumasta riippuen joko huonoin tai toiseksi huonoin! Vaihe 3 kuitenkin sekoittaa asetelman jälleen melkoisesti, sillä se yhdessä edellisten vaiheiden muutosten kanssa kutistaa NKY:n alkuperäisversion vaatimasta ajoajasta jopa yli 85 %! Vastaavasti vaiheiden 4 – 6 vaikutus näyttäisi pylväiden korkeuksia katsottaessa jopa yhdentekevältä, mutta todellisuudessa pisimmälle jalostettu versio on siltikin vielä viidenneksen verran versiota 3 nopeampi. Syötejonojen jakaumalla ei tuntuisi olevan sanottavaa merkitystä ajoaikojen kestoon kuin versioissa 0 – 2, joissa Zipfin jakaumalla suoritus etenee jonkin verran hitaammin. Samoin kuin edellisen kuvan kohdalla voidaan jälleen tehdä hätkähdyttävä havainto, että vielä kuvassa 9.3 varsin surkeasti toiminut NKY-algoritmi muuttuu esitettyjen parannusten myötä ehdottomaan kärkiryhmään kuuluvaksi: menetelmästä jalostetuimmasta versiosta tulee vertailun kolmanneksi paras!

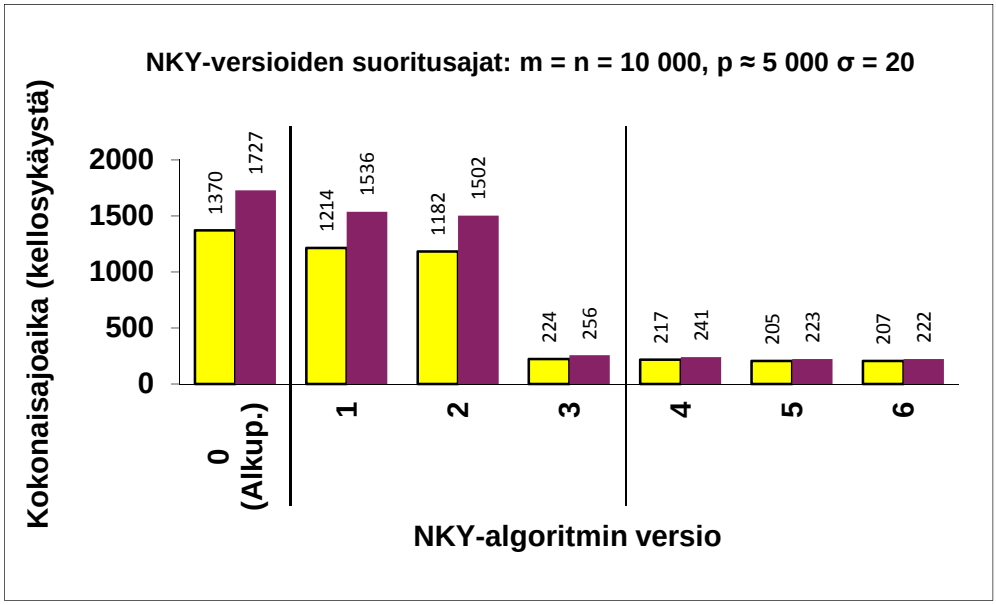


Kuva 9.21: *NKY-algoritmin eri versioiden suoritusajat, kun m, n = 10 000, σ = 4, p ≈ 9 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*).

Ylempänä näkyvässä kuvassa 9.21 nähdään kuvaa 9.4 vastaava tilanne, jossa syöttöaakkoston kokona on 4 ja PYAn prosenttiosuutena 90 %. Eri kehitysvaiheiden aikaan saamat muutokset ovat aivan samansuuntaisia kuin kahdessa edellisessäkin kuvassa. Tosin viimeinen kehitysvaihe eli optimaalisten loppuliitteiden etsintä osoittautuu tarkastelluilla syötteillä jopa hiuksenhienosti tappiolliseksi, sillä version 5 ajoajat ovat sitä nopeammat. Mainituilla syötteillä kuvassa 9.4 alkuperäinen NKY osoittautui selkeästi kelvottomaksi algoritmiksi useimpiin kilpailijoihinsa verrattuna, mutta sen parhaat versiot 5 ja 6 yltäisivät Zipfin jakaumalla nytkin lähes mitalisijalle asti: vain lyhimmän editointietäisyyden laskentaan perustuvat WMM, MYE ja MMY sekä monisuuntaisesti prosessoiva GCL suoriutuvat samaisesta tehtävästä nopeammin.

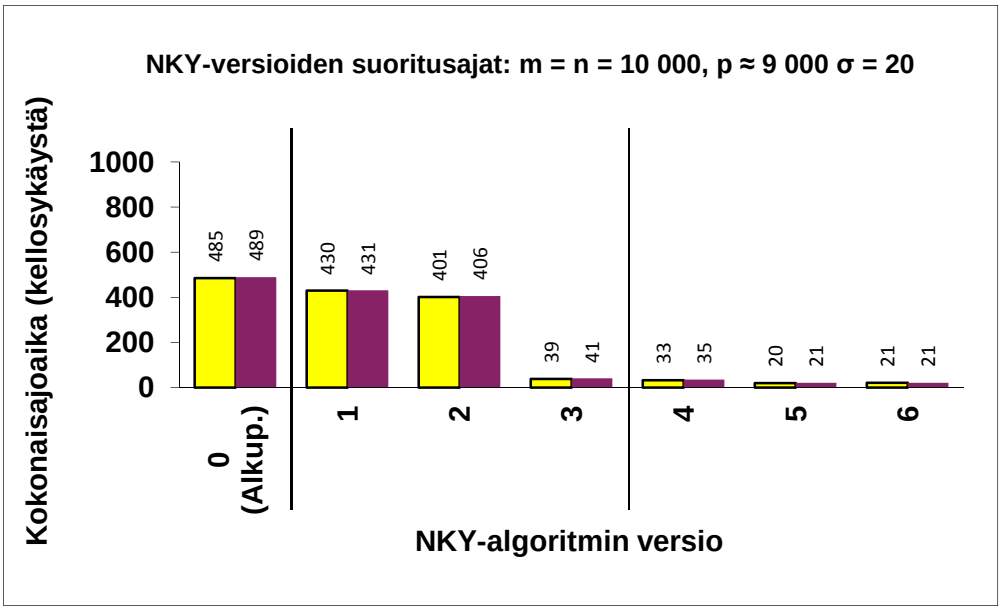
9.4.1.3 Syöttöaakkoston koko 20

Kuvassa 9.22 syöttöaakkoston kooksi on valittu 20 ja PYAn prosenttiosuudeksi 50. Mitään mainittavia muutoksia eri versioiden aikaansaamissa ajoajan säästöissä ei nytkään ole havaittavissa, vaan ne ovat varsin yhdenmukaisia kuvien 9.19 – 9.21 esittämien tilanteiden kanssa.



Kuva 9.22: *NKY-algoritmin eri versioiden suoritusajat, kun m, n = 10 000, σ = 20, p ≈ 5 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*).

Itse PYAn ratkaisemiseen kuluva aika on neljällä jalostetuimmalla NKY-versiolla pidempi kuin aikaisemmissa kolmessa kuvassa, mikä on johdonmukaista NKY-algoritmin suosiessa korkeita PYA-prosenttiosuuksia. Pisimmälle kehitetty versio kuuluisi kuvan 9.5 vertailuun mukaan otettaessa parhaimmistoon ja häviäisi molemmilla jakaumilla selkeästi ainoastaan monisuuntaisesti prosessoiville GCL:lle ja RI1:lle sekä riveittäin prosessoivalle, bittirinnakkaisuuteen perustuvalle ADI:lle, kun taas versiot 0 – 2 jäisivät vertailussa heikoimmin menestyvien joukkoon.

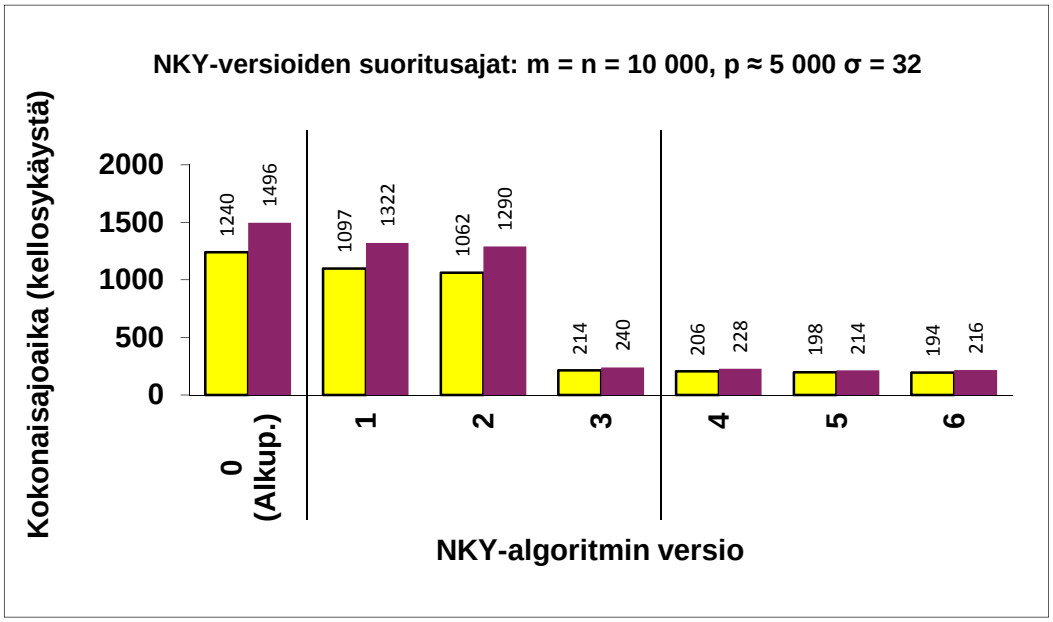


Kuva 9.23: *NKY-algoritmin eri versioiden suoritusajat, kun m, n = 10 000, σ = 20, p ≈ 9 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*).

Kuvassa 9.23 syöttöaakkoston koko pysyi ennallaan, mutta PYAn prosenttiosuuden annettiin vaihtua 50:stä 90:een. Odotetusti ajoajat lyhenivät algoritmin kaikilla versioilla. Pisimmälle kehitetyt versiot 5 ja 6 voittavat suorituskyvyssä kaikki muut algoritmit kuvan 9.6 esittämässä asetelmassa paitsi lyhimmän editointietäisyyden laskentaan alun perin erikoistettuja WMM:ää, MYE:tä ja MMY:tä. Näistäkin kaksi viimeksi mainittua kilpailevat samassa sarjassa kuin NKY:n parhaat variantit, ja ainoastaan WMM:n eduksi tulee selvä ero.

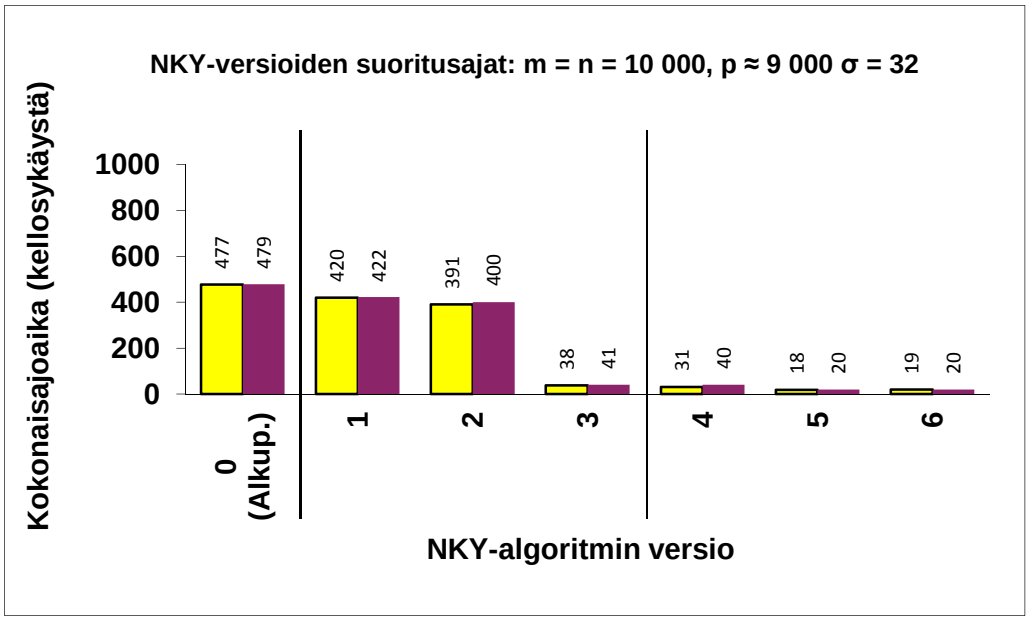
9.4.1.4 Syöttöaakkoston koko 32

Tarkastellaan seuraavaksi syöttöaakkoston kokoa 32 aloittamalla PYA-osuudesta 50 %. Testiajojen tulokset ovat nähtävissä kuvassa 9.24. Kuva on melko lailla toisinto kuvasta 9.22, jossa syöttöaakkosto oli pienempi – 20 merkin – mutta muilta osin syötejonojen ominaisuudet olivat samat. Syöttöaakkoston koon kasvaessa täsmäysten määrä harvenee, joten kauttaaltaan kuvan 9.24 esittämät ajoajat ovat hieman nopeampia kuin kuvassa 9.22. Erot merkkijakaumien välillä ovat myös samansuuntaiset – laskenta on jossain määrin hitaampaa Zipfin kuin tasaista jakaumaa noudattaville syötteille.



Kuva 9.24: *NKY-algoritmin eri versioiden suoritusajat, kun m, n = 10 000, σ = 32, p ≈ 5 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*).

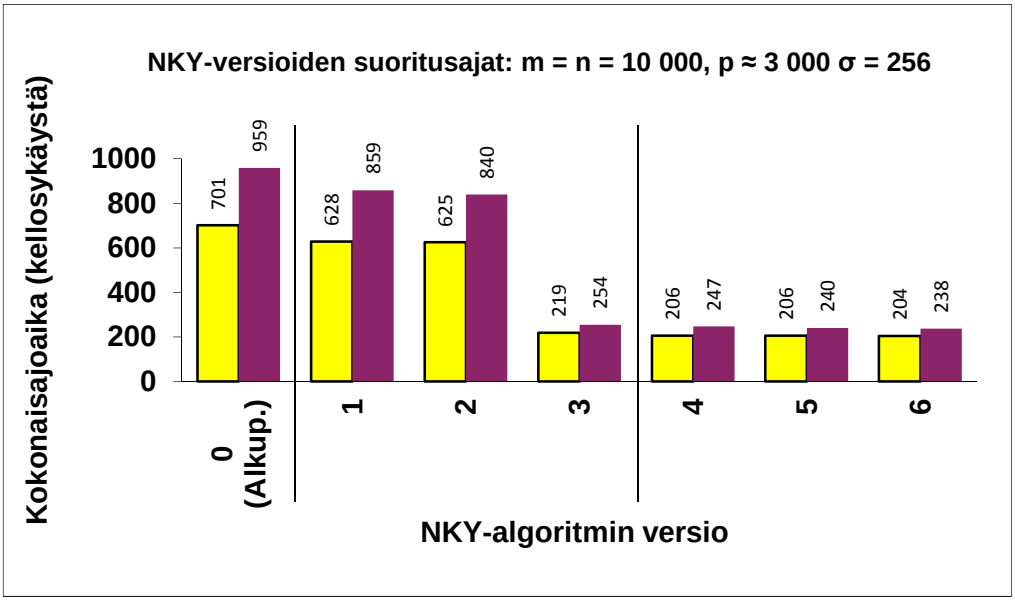
Kun syötejonojen PYA-osuuden annetaan kasvaa 50:stä 90 prosenttiin, tulokset ovat hyvin samansuuntaiset kuin vastaavalla prosenttiosuudella aakkoston koolla 20. Parhaimpien NKY-versioiden suorituskyky vetää vertoja lähes kaikille vastinkuvan 9.8 algoritmeille, kun taas alkuperäinen NKY oli samaisessa kuvassa kolmanneksi huonoin.



Kuva 9.25: *NKY-algoritmin eri versioiden suoritusajat, kun m, n = 10 000, σ = 32, p ≈ 9 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*).

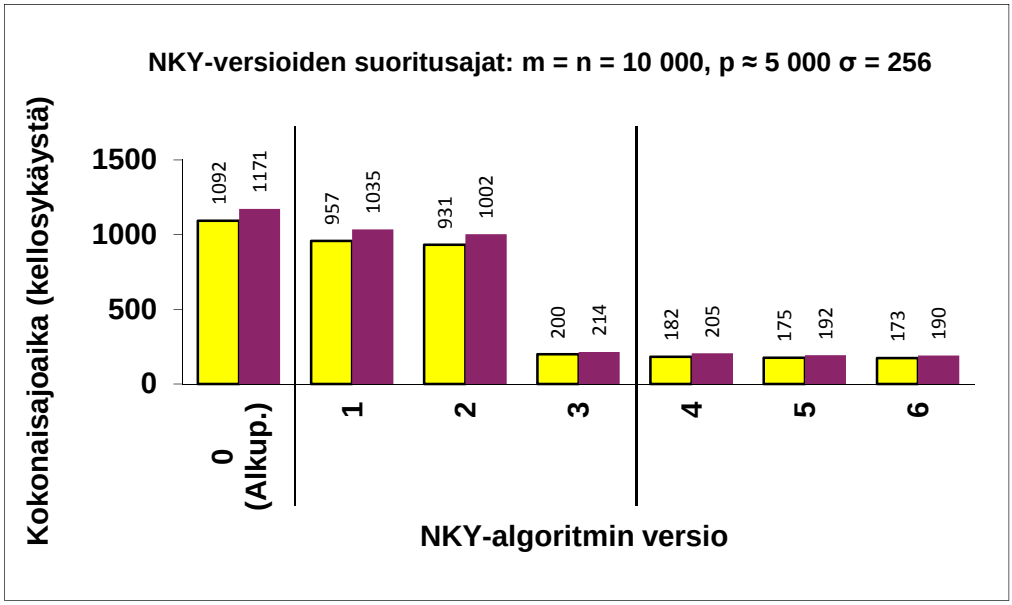
9.4.1.5 Syöttöaakkoston koko 256

Tarkastellaan seuraavaksi NKY:n eri versioiden toimintaa vielä 256:n kokoisella syöttöaakkostolla. Kuvassa 9.26 PYAn osuudeksi on kiinnitetty 30 %. Nyt parhaimman version aikaansaama ajoajan säästö alkuperäiseen NKY-algoritmiin verrattuna on ”vain” likimain 70 %, kun tehokkuuden lisäys on aikaisemmissa kuvissa ollut noin 90 %:n suuruusluokkaa. Tämä selittynee sillä, että NKY on teoreettisen aikakompleksisuutensa perusteella huonoimmillaan PYAn ollessa lyhyt, ja yhdessäkään aiemmassa esimerkissä ei PYAn prosenttiosuus ole ollut näin alhainen. Siten on ymmärrettävää, että parhainkin NKY-versio häviää suoritusajassa useimmille rivi kerrallaan prosessoiville menetelmille, kuten vertailtaessa sen ajoaikaa kuvan 9.9 sisältämien muiden algoritmien tuloksiin käy ilmi. Kuitenkin se voittaa kaikki korkeuskäyrä kerrallaan prosessoivat ja alun perin lyhimmän editointietäisyyden laskentaa varten kehitetyt menetelmät. Matriisin uudelleenorganisoinnin jälkeisillä uudistuksilla ei kuvan 9.26 syöteaineistoilla ole mainittavia vaikutusta ajoaikaan. Tämä selittyy siten, että pienellä PYA-osuudella heuristinen alaraja jää usein varsin epäinformatiiviseksi ja leikkaa vain niukalti tarkasteltavia merkkejä syötejonon *Y* alkuosasta. Myöskään pitkiä yhtenäisiä täsmäysketjuja tai pitkiä optimaalisia loppuliitteitä tuskin esiintyy PYA-osuuden ollessa vain 30 %.



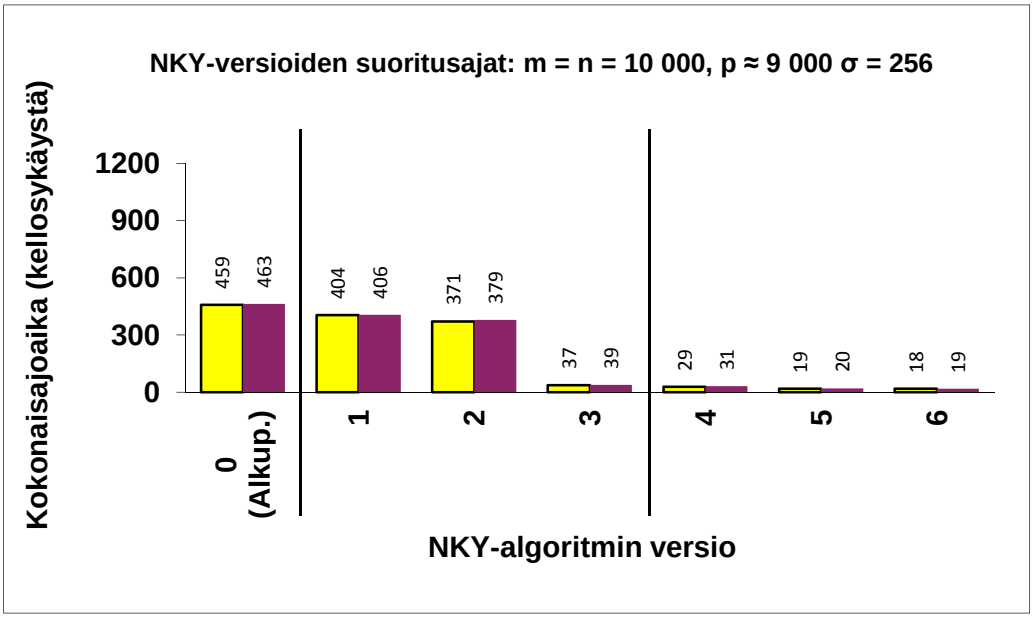
Kuva 9.26: *NKY-algoritmin eri versioiden suoritusajat, kun m, n = 10 000, σ = 256, p ≈ 3 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*).

PYAn osuuden pidentyessä 30 %:sta 50 %:iin NKY:n eri kehitysversiot käyttäytyvät samantapaisesti, mikä on nähtävissä kuvassa 9.27. Kuitenkin on myös havaittavissa, että kaikkien alkuperäistä matriisia käyttävien NKY-versioiden 0 – 2 ajoajat ovat 50 %:n PYA-osuudella selkeästi pidemmät kuin 30 %:n PYA-osuudella. Tästä huolimatta matriisin uudelleenorganisointi saa osat vaihtumaan, ja NKY:n kolmosversio toimiikin 50 %:n PYA-osuudella jo noin 10 % nopeammin kuin PYAn osuuden ollessa 30 %. Lisäksi on nähtävissä, että versiot 4 – 6 osoittautuvat jo selvästi hyödyllisiksi. Koska PYAan kuuluu nyt 2 000 merkkiä enemmän kuin kuvan 9.26 syötteillä, rajoittaa alarajaheuristiikka jo tehokkaammin turhien merkkien tarkastelua syötevektorissa *Y*. Parhaimmaksi tarkastelluilla syötteillä osoittautunut NKY:n viitosversio häviäisi kuitenkin tehokkuudessa vielä useimmille riveittäisille ja molemmille monisuuntaisesti prosessoiville menetelmille, mikä käy ilmi kuvasta 9.10.



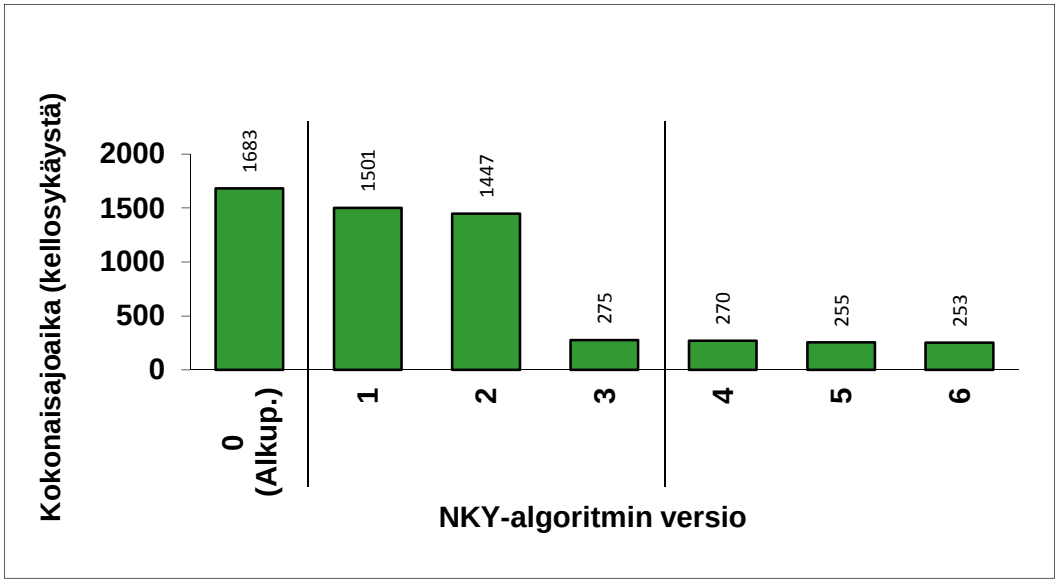
Kuva 9.27: *NKY-algoritmin eri versioiden suoritusajat, kun m, n = 10 000, σ = 256, p ≈ 5 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*).

Tarkastellaan vielä tilannetta, jossa PYA-osuus on kasvanut 90 prosenttiin, kun syöttöaakkoston kokona on edelleen 256. Tällä syöteaineistolla jokainen NKY:n versioista 3 – 6 käyttää suoritusaikaa enintään 9 % alkuperäisversion vaatimasta ajoajasta. Lisäksi kehitetyimmät versiot 5 ja 6 puristavat pois vielä miltei puolet version 3 suoritusajasta, sillä versioihin 4 – 6 sisällytetty PMX-heuristiikka toimii hyvin, kun PYAn prosentuaalinen osuus jonon *X* pituudesta on iso. NKY:stä tulisi parannusten myötä yksi tehokkaimmista algoritmeista kuvassa 9.11 tehdyssä vertailussa.



Kuva 9.28: *NKY-algoritmin eri versioiden suoritusajat, kun m, n = 10 000, σ = 256, p ≈ 9 000 ja merkkijakauma on tasainen* (*keltaiset pylväät*) */ Zipfin jakauma* (*viininpunaiset pylväät*).

Tarkastellaan lopuksi kuvassa 9.29, miten modifioitu NKY-algoritmi käyttäytyy, kun sille annetaan syötteeksi työssä jo aikaisemmin tarkasteltu *luonnollisen kielen aineisto*, jossa PYA-osuus on runsaat 40 prosenttia ja joka käsitteli Alankomaiden ja Belgian elokuvia. Mitään yllätyksiä ei ole havaittavissa, vaan matriisin uudelleenorganisointi vaiheessa 3 tehostaa jälleen kerran hyvin selkeästi algoritmin toimintaa. Vaiheet 4 – 6 puristavat yhteensä ajoaikaa kokoon tämän jälkeen enää vajaat 10 prosenttia. Koska PYA on verraten lyhyt, ei näiden versioiden sisältämän PMX-heuristiikan käytöstä ole sanottavaa hyötyä laskennan nopeuttamiseksi. Pisimmälle jalostettu NKY:n kuutosversio sijoittuisi taulukon 9.2 algoritmien vertailussa kunniakkaalle seitsemännelle sijalle, eli toteutetut uudistukset pystyivät tekemään alun perin varsin vaatimattomasti toimineesta NKY:stä kaikin puolin kilpailukykyisen menetelmän myös luonnollisen kielen mukaisille syötteille. Lisäksi on vielä huomion arvoista, että NKY ei käytä prosessointivaiheessaan apuna minkäänlaisia hakua tehostavia tietorakenteita, vaan vektoria *Y* pitkin edetään naiivilla lineaarihaulla! Jatkotutkimuksia ajatellen olisi hyvin mielenkiintoista selvittää, miten hakua voitaisiin nopeuttaa esimerkiksi yksinkertaisten esiintymälistojen avulla, joiden muodostaminen on verrattain halpaa niistä saatavaan hyötyyn verrattuna.



Kuva 9.29: *NKY-algoritmin eri versioiden suoritusajat, kun m, n = 10 000, σ = 75, p = 4 061 ja merkkijakauma on luonnollisen kielen mukainen.*

Muistinkäyttö

Taulukko 9.6 kuvastaa NKY:n eri versioiden muistinkäyttöä kuvien 9.19 – 9.29 testausasetelmille. Jokaista kuvaa kohti löytyy taulukosta kolme eri arvoa. Niistä ensimmäinen osoittaa *alkuperäisen NKY-algoritmin muistinkulutuksen*. Kyseinen arvo riippuu ainoastaan syötejonojen pituudesta, eli esimerkiksi syötejonojen merkkijakaumalla tai aakkoston koolla ei ole merkitystä. Versioiden 1 – 3 tarvitsemaa

muistin määrää ei ole taulukoitu, sillä se poikkeaa vain niukalti alkuperäisen NKY:n vastaavasta arvosta[173]. Muutos johtuu muutamista yksinkertaisista muuttujista, jotka on jouduttu lisäämään alkuperäisversiota kehitettäessä. Nämäkin kaikki versiot ovat stabiileja käytetyn merkkijakauman ja syöttöaakkoston koon suhteen. Toinen arvo edustaa puolestaan versioiden 4 – 6 keskimääräistä muistinkulutusta[174], joka – toisin kuin edellisillä versioilla – vaihtelee syötteittäin, vaikka niiden pituus pysyisi samana. Syynä tähän on se, että kulloinkin allokoitavan aputietorakenteen kokoon vaikuttavat heuristisen ala- ja ylärajan sanelemat PYAn vähimmäis- ja enimmäispituus. Kun PYAn enimmäispituus on pieni, riittää kunkin diagonaalin käyttöön varata enintään ylärajan osoittama määrä muistipaikkoja. Vastaavasti, kun PYAn tiedetään olevan pitkä, tarvitaan ainoastaan harvoja diagonaaleja, jotka voi joutua tutkimaan, sillä PYA löytyy tällöin väkisin jo verrattain aikaisessa vaiheessa. Muutoin diagonaalit kävisivät liian lyhyiksi.

Taulukko 9.6: *Alkuperäisen ja heuristiikoilla vahvistettujen NKY-versioiden muistinkulutuksen vertailua, kun m, n = 10 000. Alimman rivin syöteaineistona on luonnollista kieltä sisältävä teksti. Muiden rivien aineistotyypit on sitä vastoin generoitu sekä tasaiselle että Zipfin jakaumalle, ja esitetyt luvut ovat keskiarvoja.*

Kuva	***Aineisto***	***NKY* (*kilotavua*)**	***Versiot 4 – 6* (*kilotavua*)**	***Säästö heuristiikoilla***
9.19	*σ = 2, PYA-osuus 90 %*	*195 410*	*74 602*	*61.8 %*
9.20	*σ = 4, PYA-osuus 70 %*	*195 410*	*122 781*	*37.2 %*
9.21	*σ = 4, PYA-osuus 90 %*	*195 410*	*98 882*	*49.4 %*
9.22	*σ = 20, PYA-osuus 50 %*	*195 410*	*166 911*	*14.6 %*
9.23	*σ = 20, PYA-osuus 90 %*	*195 410*	*60 350*	*69.1 %*
9.24	*σ = 32, PYA-osuus 50 %*	*195 410*	*170 760*	*12.6 %*
9.25	*σ = 32, PYA-osuus 90 %*	*195 410*	*47 435*	*75.7 %*
9.26	*σ = 256, PYA-osuus 30 %*	*195 410*	*182 178*	*6.8 %*
9.27	*σ = 256, PYA-osuus 50 %*	*195 410*	*161 974*	*17.1 %*
9.28	*σ = 256, PYA-osuus 90 %*	*195 410*	*39 549*	*79.8 %*
9.29	*σ = 75, PYA-osuus 41 %, luonnollinen kieli*	*195 410*	*173 679*	*11.1 %*

Taulukosta on selvästi nähtävissä heuristisen ylä- ja alarajojen hyödyllisyys NKY:n muistintarpeen kannalta. Mitä pidempi PYAn prosentuaalinen osuus on, sitä suurempi muistitilan säästö saavutetaan heuristisella esiprosessoinnilla. Myös silloin, kun PYAn prosenttiosuus jää hyvin lyhyeksi – vaikkapa 30 % mittaiseksi – saadaan muistitilaa säästymään kuitenkin yli 5 prosenttia.

[173] Esimerkkiaineistoilla versioiden 1 – 3 muistintarve olisi 195 449 kilotavua.
[174] Muistinkulutus on laskettu versioille 4 – 6 tasaisen ja Zipfin jakauman keskiarvona; luonnollisella kielellä testattuja aineistoja on ainoastaan yksi (taulukon alin rivi).

9.4.2 Testiajot alarajaheuristiikalla täydennetylle KCR-algoritmille

Vuonna 2005 ilmestyneessä artikkelissa [Ber05] suoritettiin vertailuja *Kuon ja Crossin* algoritmin alkuperäiselle ja heuristisella prosessoinnilla vahvistetuille versioille. Heuristista laskentaa sovellettiin algoritmille kahdella eri tavalla: sekä yksistään *kertaluonteisena esiprosessointina* että *toistetusti*. Seuraavassa taulukossa vertaillaan alkuperäisen KCR-algoritmin ajankäyttöä sen modifioitujen versioiden kanssa, joihin on upotettu heuristisen alarajan laskenta.

Taulukko 9.7: *Suoritusajat Kuon ja Crossin alkuperäiselle, kertaalleen heuristisen esiprosessoinnin suorittavalle sekä toistuvasti alarajan laskevalle versiolle, kun uudistettu alarajan laskenta tapahtuu 1 000 rivin välein. Alimman rivin syöteaineistona on ollut luonnollista kieltä sisältävä teksti. Muiden rivien aineistotyypit on sitä vastoin generoitu sekä tasaiselle että Zipfin jakaumalle.*

Aineisto	***KCR* (*tas. / Zipf*)**	***KCR* + *staatt. PMX* (*tas. / Zipf*)**	***KCR* + *dyn. PMX* (*tas. / Zipf*)**	***Ajan säästö heuristiikkaa käyttämällä %***
σ = 2, PYA-osuus 90 %	***720* / *722***	*340 / 310*	***251* / *245***	*52.8 – 66.1*
σ = 4, PYA-osuus 70 %	***465* / *533***	*304 / 322*	***253* / *286***	*34.6 – 46.3*
σ = 4, PYA-osuus 90 %	***424* / *499***	*234 / 282*	***171* / *208***	*43.5 – 59.7*
σ = 20, PYA-osuus 50 %	***135* / *244***	*108 / 187*	***96* / *172***	*20.0 – 29.5*
σ = 20, PYA-osuus 90 %	***120* / *226***	***37*** / *93*	***37* / *82***	*58.8 – 69.2*
σ = 32, PYA-osuus 50 %	***94* / *197***	*79 / 157*	***67* / *139***	*16.0 – 29.4*
σ = 32, PYA-osuus 90 %	***87* / *189***	***24*** / *59*	*26 / **57***	*68.8 – 72.4*
σ = 256, PYA-osuus 30 %	*22 / **103***	***19* / *89***	***23*** / *95*	*-4.5 – 13.6*
σ = 256, PYA-osuus 50 %	***29* / *100***	***18*** / *85*	*24 / **75***	*15.0 – 37.9*
σ = 256, PYA-osuus 90 %	***38* / *107***	***7* / *30***	*11 / 32*	*70.1 – 81.6*
σ = 75, PYA-osuus 41 %, luonnollinen kieli	***155***	*133*	***129***	*14.2 – 16.8*

Tarkasteltaessa taulukkoa 9.7 havaitaan, että yhtä aineistotyyppiä lukuun ottamatta millään tutkitulla syöteaineistolla kumpikaan heuristisista versioista ei toimi hitaammin kuin alkuperäinen Kuon ja Crossin algoritmi. Jokaisen syöteaineiston paras suoritusaika on merkitty taulukkoon **vihreällä** ja vastaavasti huonoin **punaisella** värillä. Syöttöaakkoston koolla 256 ja PYA-osuuden ollessa 30 % toistetusti alarajan laskeva versio vaatii tasaisella jakaumalla marginaalisesti pidemmän suoritusajan kuin alkuperäinen KCR. Sen sijaan vain kertaalleen alarajan laskeva heuristinen versio säästäisi tälläkin syötteellä ajoaikaa noin kahdeksasosan. Yleisesti tarkastellen *heuristiikkojen aikaansaama ajoaikojen nopeutuminen on vähäisintä silloin, kun PYA-*

osuus on pieni, mille on ilmeiset teoreettiset perusteet, kuten aliluvusta 6.1 käy ilmi. Tuolloin heuristiikan laskema alaraja on verrattain lyhyt, jolloin sen turvin ei pystytä katkaisemaan rivien tarkastelua kovinkaan merkittävässä määrin. Lisäksi taulukosta havaitaan, että *alarajan toistuvasta laskennasta on parhaiten hyötyä syöttöaakkoston koon ollessa pieni*. Tähän puolestaan vaikuttaa se tosiasia, että tuolloin täsmäyksiä esiintyy syötteiden merkkiparien välillä verrattain tiheälti, joten heuristisen KCR-versioiden taustalla toimiva PMX-heuristiikka ei välttämättä heti pääse todellisen PYA-jonon jäljille, mutta alaraja voi tarkentua oleellisesti, kun laskenta toistetaan aina 1 000 rivin tultua käsitellyksi. Toiston suorittaminen tätä tiheämmin alkaa vähitellen tuntua jo ajoajan pidentymisenä, ja vastaavasti toistovälin harventaminen on kiusallista silloin, jos aikaisemmin laskettu alaraja jää laadultaan kovin vaatimattomaksi.

Kaiken kaikkiaan, jo kuvien 9.1 – 9.12 vertailuissa yleisesti varsin hyvin menestynyt KCR-algoritmi tehostuu taulukon 9.7 mittaustulosten perusteella vielä huomattavasti, kun siihen upotetaan heuristisen alarajan laskenta. Syöttöaakkoston koon ollessa pieni tuntuisi olevan kannattavaa käyttää dynaamista, usein toistuvaa alarajan laskentaa. Sen sijaan aakkoston koon kasvaessa pelkästään yksi heuristisen alarajan laskentakerta saattaa johtaa parhaaseen tulokseen.

Seuraavaan taulukkoon 9.8 on kerätty edellä tarkasteltujen kolmen KCR-version *muistinkulutukset* tarkastelluille syöteaineistoille. Esitetyt luvut ovat tasaisen ja Zipfin jakaumien keskiarvoja. Kumpaakin jakaumaa kohti suoritettiin kymmenen testiajoa. Taulukosta ilmenee, että syötetyypistä riippumatta heuristiikkojen käyttö vähentää huomattavasti KCR-algoritmin muistintarvetta. Muistitilaa säästyy suhteellisesti sitä enemmän, mitä pidempi tarkasteltavan syötetyypin PYA-osuus on. Tämä ominaisuus pätee syöttöaakkoston koosta riippumatta. Vastaavasti vähiten muistitilaa säästyy lyhyillä PYA-osuuksilla, jolloin verrattain harvoja täsmäyksiä voidaan jättää tarkastelujen ulkopuolelle, koska PMX-heuristiikan laskema alaraja on tuolloin lyhyt. Taulukko näyttäisi antavan tukea hypoteesille, että heuristiikan dynaaminen soveltaminen tehostaisi parhaiten KCR:n muistinkäyttöä syötejonojen tyypistä riippumatta: systemaattisesti kaikilla syötetyypeillä toistuvasti PYAn pituuden alarajan laskeva KCR-versio vaatii vähiten muistia.

Taulukko 9.8: *Alkuperäisen ja heuristiikoilla vahvistettujen KCR-versioiden muistinkulutuksen vertailua, kun m, n = 10 000. Alimman rivin syöteaineistona on luonnollista kieltä sisältävä teksti. Muiden rivien aineistotyypit on sitä vastoin generoitu sekä tasaiselle että Zipfin jakaumalle, ja esitetyt luvut ovat keskiarvoja.*

Aineisto	***KCR***	***KCR + staatt. PMX***	***KCR + dyn. PMX***	***Muistitilan säästö heuristiikkaa käyttämällä %***
σ = 2, PYA-osuus 90 %	***321 998***	*116 891*	***76 004***	*63.7 – 76.4*
σ = 4, PYA-osuus 50 %	***195 758***	*113 194*	***90 049***	*42.2 – 46.0*
σ = 4, PYA-osuus 90 %	***179 504***	*86 275*	***51 313***	*48.1 – 71.4*
σ = 20, PYA-osuus 50 %	***66 096***	*50 715*	***42 465***	*23.3 – 35.8*
σ = 20, PYA-osuus 90 %	***53 850***	*12 969*	***9 700***	*75.9 – 82.0*
σ = 32, PYA-osuus 50 %	***48 922***	*38 446*	***31 251***	*21.4 – 36.1*
σ = 32, PYA-osuus 90 %	***40 086***	*5 967*	***5 310***	*85.1 – 86.8*
σ = 256, PYA-osuus 30 %	***19 404***	*16 591*	***15 475***	*14.5 – 20.2*
σ = 256, PYA-osuus 50 %	***17 288***	*13 445*	***10 618***	*22.2 – 38.6*
σ = 256, PYA-osuus 90 %	***15 155***	*2 083*	***2 000***	*86.3 – 86.8*
σ = 75, PYA-osuus 41 %, luonnollinen kieli	***52 344***	*43 907*	***39 219***	*16.1 – 25.1*

9.4.3 Testiajot alarajaheuristiikalla täydennetylle RI1-algoritmille

Edellä tarkasteltiin, minkä verran heuristisella esiprosessoinnilla on vaikutusta NKY- ja KCR-algoritmien toimintaan. Molemmille menetelmille – erityisesti KCR:lle – heuristiikan käyttö osoittautui erittäin hyödylliseksi sekä ajoajan nopeuttajana että muistinkulutuksen vähentäjänä. Kyseisten kahden algoritmin perusversioilla oli kuitenkin heikko kohtansa: KCR toimii hitaasti, kun syöttöaakkosto on pieni ja täsmäyksiä paljon, ja NKY:n perusversio ei osoittautunut kilpailukykyiseksi oikeastaan millään syötteillä. Seuraavaksi testataan heuristiikan käyttökelpoisuutta menetelmälle, jonka perusversio ei ole toiminut huonosti vielä millekään aineistolle. Yksi tällaisista harvoista algoritmeista on *Rickin I algoritmi* (RI1), johon lisätään heuristinen esiprosessointi. Esiprosessointi on toteutettu ainoastaan staattisesti. Taulukossa 9.9 vertaillaan ajoaikojen eroja suorittamalla RI1-algoritmia ensiksi alkuperäisessä muodossaan ja sitten staattisella heuristisella esiprosessoinnilla vahvistettuna.

Taulukko 9.9: *Suoritusajat Rickin I algoritmin alkuperäiselle sekä kertaalleen heuristisen esiprosessoinnin suorittavalle versiolle. Alimman rivin syöteaineistona on ollut luonnollista kieltä sisältävä teksti. Muiden rivien aineistotyypit on sitä vastoin generoitu sekä tasaiselle että Zipfin jakaumalle.*

Aineisto	***RI1* (*tas. / Zipf*)**	***RI1* + staatt. *PMX* (*tas. / Zipf*)**	***Ajan säästö heuristiikkaa käyttämällä %***
σ = 2, PYA-osuus 90 %	***87 / 92***	***58 / 60***	*33.3 – 34.8*
σ = 4, PYA-osuus 70 %	***165 / 190***	***132 / 147***	*20.0 – 22.6*
σ = 4, PYA-osuus 90 %	***47 / 53***	***29 / 32***	*38.3 – 39.6*
σ = 20, PYA-osuus 50 %	***145 / 159***	***114 / 120***	*21.4 – 24.5*
σ = 20, PYA-osuus 90 %	***31 / 26***	***7 / 9***	*65.4 – 77.4*
σ = 32, PYA-osuus 50 %	***118 / 126***	***97 / 101***	*17.8 – 19.8*
σ = 32, PYA-osuus 90 %	***25 / 23***	***8 / 9***	*60.1 – 68.0*
σ = 256, PYA-osuus 30 %	***89 / 153***	***86 / 143***	*3.4 – 6.5*
σ = 256, PYA-osuus 50 %	***77 / 103***	***70 / 94***	*8.7 – 9.1*
σ = 256, PYA-osuus 90 %	***60 / 62***	***57 / 56***	*5.0 – 9.7*
σ = 75, PYA-osuus 41 %, luonnollinen kieli	***168***	***143***	*14.9*

Taulukon 9.9 tulokset kertovat selkeää kieltä heuristisen esiprosessoinnin hyödyistä RI1:n ajoajan nopeuttajana. Aakkoston koon ollessa enintään 32 heuristiikan käyttäminen tehostaa algoritmin toimintaa likimain vähintään viidenneksellä, ja 90 %:n PYA-osuudella ero PMX-heuristiikalla vahvistetun version hyväksi on huomattavan iso. Sen sijaan alhainen PYA-osuus yhdistettynä vielä laajaan syöttöaakkostoon syö heuristisen esiprosessoinnin edut melko lailla kuiviin. Tuolloin alaraja on yleensä melko niukalti informatiivinen lyhyen PYA-osuuden tähden, ja vastaavasti tarkan algoritmin suoritusaikaa dominoi melkoisesti kahden lähiesiintymätaulukon muodostaminen, miltä ei heuristiikkaa käyttämällä vältytä. Kuitenkaan yhdessäkään tapauksessa ei havaittu heuristisella esiprosessoinnilla täydennetyn version tuottavan hitaampaa suoritusaikaa kuin alkuperäisversio. Erityisesti aakkoston koilla 20 ja 32 ja pitkällä PYA-osuudella heuristiikkaa hyödyntävä variantti on erittäin nopea ja olisi kuvien 9.6 ja 9.8 asetelmissa parhain menetelmä.

Taulukko 9.10: *Alkuperäisen ja staattisella heuristiikalla vahvistettujen RI1-versioiden muistinkulutuksen vertailua, kun m, n = 10 000. Alimman rivin syöteaineistona on luonnollista kieltä sisältävä teksti. Muiden rivien aineistotyypit on sitä vastoin generoitu sekä tasaiselle että Zipfin jakaumalle, ja esitetyt luvut ovat keskiarvoja.*

Aineisto	***RI1***	***RI1 + staatt. PMX***	***Muistitilan säästö heuristiikkaa käyttämällä %***
σ = 2, PYA-osuus 90 %	***49 260***	***31 072***	*36.9*
σ = 4, PYA-osuus 50 %	***70 996***	***52 926***	*25.5*
σ = 4, PYA-osuus 90 %	***28 153***	***16 885***	*40.0*
σ = 20, PYA-osuus 50 %	***36 323***	***29 872***	*17.8*
σ = 20, PYA-osuus 90 %	***7 899***	***3 276***	*58.5*
σ = 32, PYA-osuus 50 %	***27 030***	***22 606***	*16.4*
σ = 32, PYA-osuus 90 %	***6 686***	***3 522***	*47.3*
σ = 256, PYA-osuus 30 %	***31 012***	***29 682***	*4.3*
σ = 256, PYA-osuus 50 %	***25 733***	***24 403***	*5.2*
σ = 256, PYA-osuus 90 %	***20 905***	***20 436***	*2.2*
σ = 75, PYA-osuus 41 %, luonnollinen kieli	***39 103***	***35 236***	*9.9*

Taulukko 9.10 valaisee heuristisen esiprosessoinnin vaikutusta RI1:n muistinkulutukseen. Heuristiikan käyttö heijastuu myönteisesti myös algoritmin tarvitsemaan muistin määrään. Mikäli aakkoston koko on enintään 32:n suuruinen, saavutetaan tarkastelluilla aineistotyypeillä vähintään 15 prosentin suuruinen muistitilan säästö verrattuna siihen, että käytettäisiin RI1:tä ilman heuristiikkaa. Isolla aakkostolla alkaa puolestaan aputietorakenteiden koko dominoida niin runsaasti tarkan algoritmin koko muistinkulutusta, ettei siitä enää heuristiikkaa käyttämällä saada mainittavasti nipistettyä pois. Kuitenkaan missään testiasetelmassa heuristiikan käyttö ei lisää muistinkulutusta. Vielä 75 merkin luonnollisen kielen aakkostolla muistitila supistui lähes 10 % alkuperäisestä.

Jos halutaan verrata heuristisen KCR:n ja RI1:n ajan- ja muistinkäyttöä, havaitaan, että luonnollisen kielen 75-merkkinen aakkosto tuntuisi toimivan ikään kuin vedenjakajana. Tätä pienemmillä aakkostoilla RI1 on selkeästi KCR:ää parempi valinta, ja se on sikäli hyvin turvallinen ratkaisu, että menetelmä sietää hyvin myös binääriaakkostoa, jolle KCR on heuristisestikin vahvistettuna tuntuvasti tehottomampi. Sen sijaan 256:n kokoisella aakkostolla RI1:n lähiesiintymätaulukoiden muodostamisesta aiheutuva taakka kasvaa niin paljon, että KCR alkaa voittaa sen sekä ajoajan että muistintarpeen suhteen mitattuna. Tästä huolimatta RI1 on kuitenkin hyvin käyttökelpoinen menetelmä myös 256:n aakkostolla. Mitä ilmeisimmin RI1:tä koskevat

havainnot heuristiikan käytön hyödyllisyydestä voitaisiin laajentaa koskemaan myös toista monisuuntaisesti prosessoivaa menetelmää: *Goemanin ja Clausenin algoritmia*.

9.5 Hakutietorakenteiden tehokkuusvertailu

Aliluvussa 8.1 analysoitiin vaihtoehtoisia tapoja rivi kerrallaan prosessoivien menetelmien hakurakenteen toteuttamiseksi. Analyysiä täydentämään muodostettiin *Apostolicon ja Guerran II algoritmista* [Apo87] kuusi erilaista versiota, joista kussakin hakua tukeva aputietorakenne toteutettiin eri tavalla. Muilta osin algoritmiversiot ovat identtiset keskenään. Algoritmi AG2 valittiin tarkastelun kohteeksi erityisesti siitä syystä, että sen kehittäjät kiinnittivät artikkelissaan paljolti huomiota vaihtoehtoisiin ratkaisuihin haun toteuttamiseksi. Haun tarkoituksena on löytää *rivin seuraava dominantti täsmäys tietystä vektorin Y indeksistä j lähtien*.

Vertailun kohteiksi otettiin seuraavat vaihtoehtoiset ratkaisut: 1) suora *lineaariselaus* vektoriin *Y* ilman mitään aputietorakenteita, 2) *yhteen suuntaan linkitetty lista* vektorin *X* merkkien esiintymäkohdista vektorissa *Y*, 3) *vektoriin* tallennettu esiintymälista, jota käsitellään kuitenkin *lineaarihaulla* 4) *puolitushaku* vektoriin tallennetusta esiintymälistasta, 5) vektorin *Y* suuntainen *lähiesiintymätaulukko* ja 6) vektorin *Y* suuntainen *lähiesiintymävektori*. Testiajo suoritettiin jokaiselle algoritmiversiolle kaksivaiheisesti. Ensimmäisessä vaiheessa algoritmin suoritus lopetettiin heti esiprosessointivaiheen päätyttyä. Täten saatiin selville, paljonko eri vaihtoehtoisten ratkaisujen vaatimat alustustoimenpiteet vievät aikaa, kun kummankin syötejonon pituudeksi valittiin 10 000 ja syöttöaakkoston kooksi 256. Tämän jälkeen toisessa vaiheessa PYA-ongelma ratkaistiin käyttämällä tasaista merkkijakaumaa. PYAn osuudeksi kiinnitettiin 50 %. Seuraava taulukko kuvastaa testiajon tuloksia.

Taulukko 9.11: *AG2-algoritmin suoritusajat ja vaihtoehtoisten hakua tukevien tietorakenteiden alustusajat, kun m, n = 10 000, σ = 256, p = 5 000 ja syötejonot noudattavat tasaista merkkijakaumaa*

Hakutietorakenne	***Alustusaika***	***Ratkaisuaika***
lineaarinen selaus	***0***	***164***
linkitetty lista	*0*	***60***
esiintymälista vektorissa	*0*	*124*
vektori + puolitushaku	*0*	*122*
lähiesiintymätaulukko	***26***	*159*
lähiesiintymävektori	*1*	*132*

Taulukosta 9.11 nähdään, että käytettiinpä lähiesiintymätaulukkoa lukuun ottamatta mitä tahansa muuta ratkaisua hakurakenteen toteuttamiseksi, alustuksen kustannus on merkityksetön sekä absoluuttisesti mitattuna että suhteessa algoritmin

kokonaissuoritusaikaan. Lineaarista selausta sovellettaessa vältytään tyystin alustuskustannuksilta, sillä sitä varten ei tarvitse perustaa mitään erillistä tietorakennetta, vaan haku kohdistetaan suoraan syötejonoihin. Samalla kuitenkin havaitaan, että myös kaikki 1-ulotteiset aputietorakenteetkin vaativat ainoastaan enintään yhden yksikön alustusaikaa, joten resursseja ei juuri hukata, vaikka tällainen tietorakenne perustettaisiin. Suorasaannin mahdollistava 2-ulotteinen lähiesiintymätaulukko vaatii kuitenkin yli 25-kertaisen luontikustannuksen yksinkertaisempiin rakenteisiin verrattuna.

Tarkasteltaessa ratkaisuaikoja havaitaan odotetusti, että ”liiallinen laiskuus” eli hakua tukevan aputietorakenteen kokonaan perustamatta jättäminen kostautuu selvästi pidentyneenä ajoaikana. Silmiin pistävää on kuitenkin yksinkertaisen linkitetyn listan tehokkuus: sitä käyttämällä AG2-algoritmin ajoajan kesto on vain puolet kehittyneempien tietorakenteiden avulla toteutettujen versioiden vaatimasta suoritusajasta. Vektoritoteutuksen kannalta vaikuttaisi samantekevältä, sovelletaanko siihen puolitushakua vai ei. Lähiesiintymätaulukon suurehko alustuskustannus syö rakenteen mukanaan tuoman vakioaikaisen haun edut, ja sen yksiulotteiseksi puristettu versio — lähiesiintymävektori — vaatii vastaavasti usein niin monta lisälaskutoimitusta, että siitä tulee kaksiulotteista taulukkoa tehottomampi kokonaiskustannukseltaan.

10 Yhteenveto

Käsillä olevassa työssä luotiin katsaus kahden merkkijonon pisimmän yhteisen alijonon ongelmaan, sen ominaisuuksiin ja ratkaisemistapoihin. Koska työn lähestymistapa on paitsi tutkimuksellinen niin samalla myös oppi- tai käsikirjamainen, siihen haluttiin sisällyttää kattava näkymä jo tunnettuihin ja olemassa oleviin ongelman ratkaisutapoihin ja niitä tukeviin tietorakenteisiin. Siten johdantoluvun ja ongelmalle tyypillisten käsitteiden ja määritelmien esittelyn jälkeen esiteltiin luvussa 3 joukko kahden merkkijonon PYAn ratkaisevia algoritmeja. Näkymää laajennettiin luvussa 4 esittelemällä lisäksi kolme PYA-algoritmia, jotka on alun perin kehitetty kahden merkkijonon välisen lyhimmän editointietäisyyden määräämiseen. Näiden lukujen perusteella lukija voi muodostaa kuvan siitä, millaisia menetelmiä on tarjolla PYA-ongelman ratkaisemiseksi ja millaisille syötteille eri menetelmät teoreettisesti parhaiten soveltuvat.

Työn eräs tutkimuksellisesti tärkeimmistä tavoitteista on ollut selvittää, millä tavoin nykyisten algoritmien suoritusaikaa voitaisiin nopeuttaa ja niiden muistitilan tarvetta pienentää. Useissa käytännön sovelluksissa saattaa nopeasti toimiva algoritmi vaatia kohtuuttoman suuren määrän muistia, tai vastaavasti siedettävässä muistitilassa toimivan algoritmin suoritus saattaa kestää kiusallisen pitkään. Luvussa 5 esitetyt *heuristiset menetelmät* voivat tarjota käyttökelpoisen ratkaisun PYA-ongelmaan silloin, kun ongelman tarkkaa ratkaisua ei välttämättä – ainakaan heti – tarvitse saavuttaa, vaan jo hyvä likiarvo PYAn pituudesta on riittävän informatiivinen. Likiarvo muodostaa PYAn pituudelle joko ylä- tai alarajan. Osa heuristisista menetelmistä on hyvin nopeita, ja lisäksi niiden muistinkulutus on vähäinen.

Luvussa 6 tarkasteltiin, miten heuristista ylä- tai alarajaa voidaan käyttää hyväksi tarkan algoritmin suorituksen tehostamiseksi. Siinä todettiin, että useissa tähänastisissa tarkoissa algoritmeissa ehdoitta kirjattavista dominanteistakaan täsmäyksistä kaikki eivät ole välttämättömiä, mikäli käytössä on riittävästi tietoa PYAn vähimmäispituudesta. Mitä tarkemmin PYAn vähimmäispituus tiedetään, sitä useampia täsmäyksiä voidaan tarkkaa ratkaisua etsittäessä ohittaa matriisin rivien alku- ja loppuosista. Vastaavasti heuristista ylärajaa voidaan käyttää hyväksi varattaessa muistia tarkan algoritmin käyttämälle aputietorakenteelle. Tämä pätee erityisesti NKY-algoritmiin, jonka perusversio varaa useimmiten paljolti tarpeetonta muistitilaa.

Luvussa 7 nähtiin, että heuristisiin menetelmiin liittyy myös useita puutteita. Mitä enemmän alarajan pituus lähestyy nollaa tai ylärajan pituus arvoa m, sitä hämärämmän kuvan heuristiikka antaa PYAn todellisesta pituudesta. Luvussa tarkasteltiin heuristiikkojen pahimpia puutteita, jotka heikentävät niiden laskemien rajojen luotettavuutta. Lisäksi siinä esitettiin ehdotuksia heuristiikkojen luotettavuuden parantamiseksi.

Luvussa 8 kiinnitettiin huomiota siihen, miten tarkan PYA-algoritmin käyttämän hakutietorakenteen valinnalla on vaikutusta algoritmin suoritusaikaan ja

muistinkulutukseen. Monissa algoritmeissa tietorakenne on verrattain vähällä työllä vaihdettavissa toiseksi. Yksinkertainen aputietorakenne hakua varten – esimerkiksi yhteen suuntaan linkitetty lista – on helppo toteuttaa ja vaatii vain niukalti muistia. Toisena äärimmäisyytenä voisi mainita vakioaikaista hakua tukevan lähiesiintymätaulukon, jonka rakentamiskustannus on kuitenkin $O(n\sigma)$. Jotta algoritmi toimisi tehokkaasti, on oleellista ennen kaikkea tietorakenteen soveltuvuus algoritmin käyttöön. Kaikenlainen ylimitoitettu laskenta – tapahtuupa se sitten tietorakenteen alustamisvaiheessa taikka vasta hakuvaiheessa – johtaa pitkän päälle tehottomaan lopputulokseen. Luvussa kiinnitettiin lisäksi huomiota siihen, että matriisiin tehtävien päivitysten paikallisuus tulee taloudellisemmaksi kuin jatkuva siirtyminen sekä riviltä että sarakkeelta toiselle. Päivitysten ollessa paikallisia välimuistin operaatiot ovat nopeita. Edelleen samaisessa luvussa tarkasteltiin, miten laskennan tehokkuutta voidaan lisätä tekemällä syötejonoista havaintoja, joiden avulla pystytään välttämään tietorakenteisiin tallennettujen arvojen turhia testauksia ja päivityksiä.

Luku 9 empiirisine testituloksineen antoi vastauksia useimpiin aikaisempien lukujen yhteydessä esille nousseisiin kysymyksiin. Tarkoista, jalostamattomista PYA-algoritmeista osoittautuivat luotettavimmaksi sekä rivi- että sarakesuuntaista prosessointia suorittavat *Goemanin ja Clausenin* sekä *Rickin I algoritmi*, jonka suoritusaika kuuluu parhaimmistoon kaikilla testatuilla syötejonoilla, joiden pituus oli 10 000 merkkiä. Myös riveittäin etenevä, bittirinnakkaisuuteen perustuva ADI osoittautui käyttökelpoiseksi menetelmäksi kaikissa testiajoissa, joskaan se ei sijoittunut missään testiasetelmassa aivan nopeimpien menetelmien joukkoon. Muista rivi kerrallaan laskentaa suorittavista menetelmistä KCR, MUK, HSZ, HD2 ja AG2 toimivat verrattain hyvin, kunhan syöttöaakkosto ei jää aivan pieneksi. Alle 20 merkin aakkostolla täsmäysten suuri lukumäärä sen sijaan heikentää niiden tehoa. Diagonaali kerrallaan etenevät, alun perin lyhimmän editointietäisyyden laskentaan kehitetyt menetelmät toimivat hyvin nopeasti edellyttäen, että PYA-osuus on iso. Sen sijaan PYAn ollessa lyhyt kyseiset menetelmät ovat tehottomia. Korkeuskäyrä kerrallaan prosessoivat menetelmät eivät menesty juuri missään testiajossa kovin hääppöisesti. Selkeimpänä syynä tähän näyttää olevan toimintatavalle hyvin räätälöidyn hakutietorakenteen puuttuminen. Yhden ulomman silmukan kierroksen aikana joudutaan siirtymään sekä riveillä että sarakkeilla, mikä ei ole hakuoperaatioita ajatellen kovinkaan taloudellista. Täsmälleen samasta syystä diagonaali kerrallaan etenevä NKY-algoritmi osoittautuu jopa odottamattoman heikoksi. Testiajoissa tasaista jakaumaa noudattaville syöteaineistoille algoritmit toimivat yleisesti jonkin verran nopeammin kuin vinolle Zipfin jakaumalle tai luonnolliselle kielelle.

Heuristisista menetelmistä luotettavimmaksi ja samalla erittäin nopeaksi osoittautui PMX-heuristiikka, joka pyrkii ahnaasti valitsemaan aina paikallisesti parhaalta valinnalta vaikuttavan täsmäyksen. Menetelmä toimii verrattain luotettavasti, jos PYA-jono sijaitsee matriisissa lähellä päälävistäjää. Muussa tapauksessa menetelmä voi kuitenkin epäonnistua melko pahastikin. Koska kuitenkin useimmat muut ala- ja

ylärajaheuristiikat ovat PMX:ään verrattuna joko primitiivisempiä ja samalla hyvin herkkiä syötteiden merkkijakaumalle, tai vaativat avukseen tarkkaa PYA-algoritmia ja ovat sen takia hitaita, osoittautuu PMX heuristiikoista parhaaksi valinnaksi.

Aliluvussa 9.4 kehitettiin testiajoissa erittäin huonosti menestynyttä NKY-algoritmia asteittain mm. poistamalla siitä tarpeettomia päivityksiä, organisoimalla sen käyttämä tietorakenne uudelleen, upottamalla siihen heuristinen esiprosessointi ja lisäämällä algoritmin tekemää havainnointia syötejonoista. Erityisesti matriisin indeksointitavan muuttaminen sai aikaan mullistavia tuloksia: NKY:n ajoaika saatiin lyhenemään murto-osaan alkuperäisestään. Perusversiona lähes kaikissa testeissä huonosti menestynyt algoritmi pesi melkoisesti kasvojaan tehtyjen uudistusten myötä, ja siitä tuli menetelmänä varsin kilpailukykyinen muiden kanssa. Heuristiikkojen osuus korostui puolestaan algoritmin vaatiman muistitilan pienentymisenä. Samaisessa aliluvussa kehitettiin myös jo perusversiona testeissä hyvin menestynyttä KCR-algoritmia lisäämällä siihen joko yhteen kertaan tai toistuvasti suoritettava heuristisen alarajan laskenta. Ajansäästöt olivat heuristisen esiprosessoinnin käyttöönoton myötä kiistattomia. Sama pätee muistinkulutukseen: huonoimmassakin tapauksessa noin 15 % perusversion tarvitsemasta muistitilasta pystyttiin säästämään. Myös yhteen parhaimmista menetelmistä, sekä riveittäin että sarakkeittain etenevään RI1-algoritmiin, sovellettiin staattista heuristista esiprosessointia. Siitä huolimatta, että menetelmä jo sinällään on tehokas, saatiin heuristiikan avulla aikaan vielä huomattavia lisäsäästöjä suoritusajassa ja muistinkulutuksessa etenkin, jos syöttöaakkosto pysyy enintään luonnollisen kielen merkistön suuruisena ja PYAn prosentuaalinen osuus ei jää kovin lyhyeksi.

Luvun 9 päätteeksi tarkasteltiin vielä hakutietorakenteiden perustamiskustannuksia sekä AG2-algoritmin ajoaikoja eri tietorakenteita käyttäen. Syötejonojen pituudeksi valittiin nytkin 10 000. Alustuskustannukset olivat jokseenkin yhdentekevät lukuun ottamatta suorasaannin mahdollistavaa lähiesiintymätaulukkoa, jonka perustamiseksi vaadittiin jo yli 25-kertainen työmäärä. Ainakaan AG2-algoritmilla ei lähiesiintymätaulukon perustamiskustannusta saatu kurottua laskentavaiheen aikana umpeen, sillä yhteen suuntaan linkitettyjen listojen avulla toteutettua esiintymälistaa käyttäen ajoajaksi saatiin 60 kellosykäystä, kun taas lähiesiintymätaulukkoa käyttämällä niitä tarvittiin peräti 159. Ero esiintymälistaa käyttävän version hyväksi siis jopa kasvoi suorituksen aikana. Tulosta ei kuitenkaan voida pitää anomaliana, sillä testiasetelmassa syöttöaakkoston kokona oli 256 ja PYA-osuutena 50 %. Yhteen suuntaan linkitettyä listaa pitkin rullaaminen on tällöin varsin nopeaa ja siten halpaa, kun taas lähiesiintymätaulukon oikeaan soluun pääseminen vie oman aikansa. Mitä vähemmän linkitetyn listan solmuja ohitetaan, sitä vähemmän on hyötyä lähiesiintymätaulukosta. Kannattaa kuitenkin mainita, että tulos voisi näyttää melko lailla toisenlaiselta, jos syöttöaakkosto olisi pieni ja ohitettavia täsmäyksiä useita. Tuolloin lähiesiintymätaulukon perustaminen mitä ilmeisimmin maksaisi itsensä suoritusaikana korkoineen takaisin.

Suoritettujen testiajojen perusteella vaikuttaa ilmeiseltä, että heuristista esiprosessointia kannattaa soveltaa tarkkoihin PYA-algoritmeihin melko lailla varauksetta. Vain ani harvoissa testiajoissa heuristinen esiprosessointi johti tarkan menetelmän ajoajan pitenemiseen. Koska tarkan algoritmin pitää pystyä ratkaisemaan asetettu PYA-ongelma kaikille mahdollisille syötejonopareille, on jo alun perin tehokkaasti toteutettua ja turhan laskennan minimoivaa algoritmia (esimerkiksi RI1:tä ja GCL:ää) vaikeaa tehostaa nykyisestään hyödyntämättä menetelmässä mitenkään kulloinkin tarkasteltavien syötejonojen ominaisuuksia. Tätä taustaa vasten voidaan todeta, että luotettavat ja samalla nopeasti toimivat ylä- ja alarajaheuristiikat tuntuisivat lupaavimmilta kehityskohteilta sekä tärkeiltä edellytyksiltä ja osatavoitteilta matkalla kohti entistä tehokkaampia, yleisesti sovellettavia tarkkoja PYA-algoritmeja.

11 Liite: Algoritmien ja heuristiikkojen pseudokoodit

11.1 Korkeuskäyrittäin laskentaa suorittavat menetelmät

11.1.1 Hirschbergin I algoritmi (HI1)

ALGORITMI HI1 (X, m, Y, n, σ):

```
FOR g := 1, 2, … → σ  /* Käydään läpi kaikki syöttöaakkoston merkit. */
        Frekvenssit[g] := 0;  /* Alustetaan aakkoston merkkien frekvenssit nolliksi. */
        Esiintymät[g, 0] := 0;  /* Viedään kunkin merkin esiintymälistaan nolla pysäytysalkioksi. */
FOR j := 1, 2, … → n  /* Selataan vektori Y alusta loppuun. */
        Frekvenssit[Y[j]] := Frekvenssit[Y[j]] + 1;  /* Kasvatetaan kohdatun Y:n merkin frekvenssiä.  */
        Esiintymät[Y[j], Frekvenssit[Y[j]]] := j; /* Viedään nykyinen Y-indeksi merkin täsmäyslistaan. */
FOR g := 1, 2, … → σ  /* Tutkitaan vielä frekvenssivektori kertaalleen läpi. */
        IF Frekvenssit[g] = 0   /* Ellei symbolia s_g esiinny lainkaan vektorissa Y … */
            Esiintymät[g, 1] := 0;  /* … asetetaan merkin esiintymälistaan vielä toinen nolla. */
FOR i := 0, 1, … → m
        D[0, i] := 0;  /* Asetetaan  pseudosolmut eri korkeuskäyrille. */
YlinRivi := 0;  /*  Haettavan täsmäyksen X-indeksin on oltava suurempi kuin YlinRivi. */
löytyi := tosi;   /*  Kontrolloi, löytyikö silmukassa S1 uusia dominanttitäsmäyksiä kierroksen aikana. */
k := 1;  /*  Lähdetään rakentamaan 1. korkeuskäyrää. */
S1: WHILE (löytyi = tosi) /* Etsitään k-täsmäyksiä vain, jos löytyi k-1 täsmäyksiä. */
    FOR g := 1, 2, … → σ
        MoneskoEsiintymä[g] := Frekvenssit[g]; /* Asetetaan osoittimet täsmäyslistojen loppuun. */
    löytyi := epätosi;  /* Silmukan suoritus päättyy, ellei uusia dominantteja täsmäyksiä löydy. */
    alaraja := D[k-1, YlinRivi];  /* Asetetaan indeksialaraja uudelle dominantille täsmäykselle. */
    yläraja := n + 1;  /*  Asetetaan vastaava yläraja. */
    S2: FOR i :=  YlinRivi + 1, YlinRivi + 2, … → m /* Tutkitaan rivit alareunaan asti. */
        S3: WHILE (Esiintymät[X[i], MoneskoEsiintymä[X[i] – 1] > alaraja)
            MoneskoEsiintymä[X[i]] := MoneskoEsiintymä[X[i]] – 1; /* Haetaan lähin esiintymä. */
        ENDWHILE (S3)
        IF (yläraja > Esiintymät[X[i], MoneskoEsiintymä[X[i]]] > alaraja)  /* k-dominantti? */
            yläraja := Esiintymät[X[i], MoneskoEsiintymä[X[i]]];  /* Kyllä: löydettiin uusi täsmäys. */
            D[k, i] := yläraja;   /* Asettaa riville i korkeuskäyrän k leikkauskohdan. */
            IF (löytyi = epätosi)  /* Löydettiin ensimmäinen korkeuskäyrän k täsmäys. */
                UusiYlinRivi := i;  /*  Määrätään aloitusrivi k+1. korkeuskäyrän etsinnälle. */
                löytyi := tosi;  /*  Ainakin yksi dominantti k-täsmäys löydetty. */
        ELSE  /* Haettavan merkin esiintymää ei löytynyt vaaditulta sarakealueelta. */
            D[k, i] := 0;  /* Riviltä i ei löytynyt dominanttia k-täsmäystä. */
        IF (D[k–1, i] > 0)  /* Löytyikö i:nneltä riviltä dominantti k-1 -täsmäys? */
            alaraja := D[k–1, i];  /* Kyllä: asetetaan sarakealaraja i:nnelle riville.
    ENDFOR (S2)
    k := k+1;
    IF (löytyi = tosi) /* Löydettiinkö ainakin yksi luokan k täsmäys? */
        YlinRivi := UusiYlinRivi; /* Kyllä: asetetaan silmukan S2 laskurille seuraavan kierroksen alkuarvo. */
ENDWHILE (S1)
p := k – 1;  /* PYAn pituus on silmukan S1 suorituskertojen määrä ykkösellä vähennettynä. */
```

```
k := p;  /* Aloitetaan PYA-jonon kerääminen lopusta alkua kohti. */
FOR i := m, m – 1, … → 1  /* Kerätään PYA lopusta alkuun päin. */[175]
    IF (D[k, i] > 0)
        PYA[k] := X[i];  /* Tulostetaan PYA-jonon k:s merkki. */
        k := k–1;  /*  Siirrytään etsimään edeltävää PYAan kuuluvaa merkkiä. */
Tulosta vektoriin PYA kerätty PYA-jono ja sen pituus p.
END (ALGORITMI HI1).
```

11.1.2 Hirschbergin II algoritmi (HI2)

```
ALGORITMI HI2 (X, m, Y, n, σ):
ε := ?;   /*  Käyttäjä asettaa ylärajan X:n ja PYA(X, Y):n pituuden erotukselle. */
FOR i := 0, 1, … → ε
        F[h] := 0;  /*  Alustetaan F-vektori nollilla. */
        G[h] := 0;  /*  Tehdään samoin G-vektorille. */
P[0] := 0;  /* Jos P[i] = 0, on palattu polun i alkupisteeseen.  */
FOR i := 1, 2, … → h
        P[h] := -1;  /*  Jos P[i] = -1, polku ei ole PYAn edustaja. */
k := 0;  /*  Alustetaan korkeuskäyrä nollaksi. */
löytyi := tosi;  /*  Looginen muuttuja löytyi kontrolloi, löytyikö Dk–täsmäyksiä vai ei. */
S1: WHILE (löytyi = tosi)  /* Etsitään kelvolliset dominantit k-täsmäykset. */
    löytyi := epätosi;   /*  Kontrollimuuttuja löytyi asetetaan silmukan alussa epätodeksi. */
    imax := 0;   /*  Maksimirivi, jolta löytyy edellinen k-täsmäys; alustetaan nollaksi. */
    jmin := n + 1;  /*  Minimisarake, josta löytyy edellinen k-täsmäys; alustetaan n+1:ksi.  */
    h := 0;  /*  Alustetaan seuraavan silmukan S2 laskuri. */
    k := k + 1;  /*  Aloitetaan luokan Dk kelvollisten täsmäysten etsiminen. */
    S2: WHILE ((k+h ≤ m) AND (h ≤ ε)) /* Tutkitaan rivit k..k+h, mikäli olemassa. */
        i := h + k;  /* Muuttuja i ilmoittaa käsiteltävän rivin eli X-syötevektorin indeksin. */
        j := SeurEs(xi, G[h]);  /*  Etsitään G[h]:ta seuraava xi:n esiintymä Y:n esiintymälistasta. */
        E1: IF (j ≥ jmin)  /* Xi:n esiintymää ei löytynyt Y:stä tai se löytyi liian kaukaa. */
            F[h] := imax;  /*  Asetetaan F[h]  edellisen k+1 –täsmäyksen riviarvoon. */
            G[h] := jmin;  /*  Asetetaan G[h] edellisen k+1 –täsmäyksen sarakearvoon. */
            uusi_P[h] := -1;  /*  Asetetaan h hyppyä sisältävä polku kelvottomaksi. */
        ELSE  /* Löydettiin uusi kelvollinen Dk-täsmäys. */
            löytyi := tosi;   /* Tehdään vielä ainakin yksi silmukan S1 kierros lisää. */
            lkm :=  (i – 1) – F[h];  /*  Montako riviä hypättiin edeltäjätäsmäyksestä?  */
            E2: IF (lkm = 0)
                uusi_P[h] := P[h]  /*  Ei hyppyjä, asetetaan linkki edeltäjän opastetietueeseen. */
            ELSE  /* Vähintään yksi yli hypätty rivi edeltäjäsolmun jälkeen. */
                LuoUusiOpastetietue(uusi_P[h]);   /* Luodaan dynaamisesti uusi opastetietue. */
                Opaste[uusi_P[h]] := <i – 1, lkm, P[h – lkm]>;   /* Tiedot tietueeseen. Kenttien nimet:
                                        <rivi, lkm, osoitinkenttä> */
            ENDIF (E2)
            imax := i;  /*  Ylin rivi, jolta on löydetty kelvollinen Dk-täsmäys. */
            jmin := j;   /*  Vasemmanpuoleisin sarake, josta on löydetty kelvollinen Dk-täsmäys. */
```

[175] Artikkelissa on tässä kohdin painovirhe: laskurin *i* alkuarvon pitää olla *m*, ei *m* + 1.

```
            F[h] := i;   /* Tallennetaan h:lla hypyllä löydetyn täsmäyksen rivinumero F[h]:hon. */
            G[h] := j;  /*  Tallennetaan vastaavan täsmäyksen sarakenumero G[h]:hun. */
        ENDIF (E1)
        h := h+1;  /*  Kasvatetaan silmukan S2 laskuria.  */
    ENDWHILE (S2)
    IF (löytyi = tosi)  /* Löytyikö kelvollisia Dk-täsmäyksiä? */
        FOR i := 1, 2, … → ε
            Tuhoa(P[i]);  /* Tuhotaan vanhan P-vektorin sisältö … */
            P[i] := uusi_P[i];  /* … ja korvataan se uusilla tiedoilla. */
ENDWHILE (S1)
z := Min{ h | P[h] ≥ 0 }, ellei tällaista h:ta ole, z := -1;  /*  Monellako hypyllä saavutettiin PYA-jono? */
p := k – 1;  /*  Silmukka S1 kasvatti k:n arvoa yhdellä isommaksi kuin PYAn pituus on. */
IF ((z < 0) OR (p < m – ε))  /* ε:n arvo liian pieni ratkaisun löytämiseksi. */
    Tulosta("PYA liian lyhyt, jotta se voitaisiin laskea parametrin ε nykyisellä arvolla.")
ELSE
    KerääPYA(F, P, X, lkm, p, rivi, z);   /* Aliohjelma kerää algoritmin löytämän PYA-jonon vektoriin S. */
END (ALGORITMI HI2).

PROSEDUURI KerääPYA (F, P, X, lkm, p, rivi, z):
ohita[z + 1] := 0;  /* Tehdään tarvittava alustus vektorille ohita. */
viim_x_täsmäys := F[z];  /*  Alin rivi, jolta saadaan valittuun PYAan kuuluva merkki. */
v := P[z];  /*  v osoittaa nyt alimman täsmäyksen opastetietueeseen. */
S3: WHILE (v ≠ 0)  /* Kun v = 0, PYA-jono on valmis. */[176]
    rivejä := lkm[v];  /* Montako seuraavaa riviä hypätään yli?  */
    sijainti := rivi[v];  /* Osoittaa nykyisen sijaintirivin. */
    WHILE niin kauan kuin (rivejä > 0)
        ohita[z] := sijainti;  /* Muuttujan sijainti mukainen rivi ohitetaan. */
        z := z – 1;  /*  Pienennetään yli hypättävien rivien laskuria. */
        sijainti := sijainti – 1;  /* Siirrytään edelliselle riville. */
        rivejä := rivejä – 1;  /*  Vähennetään perättäisten ohitettavien rivien laskuria. */
    v := osoitinkenttä[v];     /* Siirrytään edeltäjäsolmun opastetietueeseen. */
ENDWHILE (S3)
z := 1;  /*  Alustetaan arvo z uudelleen. */
k := 1;  /*  Samoin k. */
FOR i := 1, 2, … → viim_x_täsmäys
    IF (i = ohita[z])  /* Tutkitaan, pitääkö rivillä i oleva merkki ohittaa PYAssa? */
        z := z + 1  /* Pitää. Kasvatetaan samalla ohitettujen merkkien laskuria. */
    ELSE
        PYA[k] := xi;  /* Asetetaan rivillä i oleva X:n merkki PYA-jonoon S. */
        k := k + 1;  /* Kasvatetaan PYAn pituutta. */
Tulosta vektoriin PYA kerätty PYA-jono ja sen pituus p.
END (PROSEDUURI KerääPYA)
```

[176] Artikkelista puuttuu tältä riviltä 0 vertailuoperaattorin "=" oikealta puolelta.

11.1.3 Hsun ja Dun I algoritmi (HD1)

ALGORITMI HD1 (X, m, Y, n, σ)**:**

Rakenna vektorimuotoiset esiintymälistat vektorissa X esiintyvien merkkien sijaintipaikoista vektorissa Y. Lajittele esiintymät nousevaan suuruusjärjestykseen.

Talleta kunkin X:ssä esiintyvän symbolin esiintymiskertojen lukumäärä vektoriin frekv[1..σ].

```
FOR i := 1 → m
        alaraja[i] := 1;  /*  Alustetaan rivikohtaiset alarajat. */
k := 0;   /* Alustetaan korkeuskäyrän numero. */
S1: REPEAT  /* Toistetaan niin pitkään, ... */
    FOR i := 1, 2, …→ m
            yläraja[xi] := frekv[xi];  /* Asetetaan ylärajaksi kunkin merkin viimeinen esiintymä. */
    k := k + 1;  /*  Siirrytään seuraavalle korkeuskäyrälle. */
    linkkivektori[k] = ∅;  /* Alustetaan k:s korkeuskäyrä tyhjäksi. */
    kynnysarvo := ∞;  /*  Edeltäviä k-dominantteja täsmäyksiä ei vielä ole yhtään. */
    S2: FOR i := 1, 2, … → m  /* Rullataan kaikki rivit läpi. */
        IF (alaraja[i] ≤ frekv[xi])  /* Onko rivillä i vielä merkin xi käyttökelpoisia esiintymiä? */
            j := EsLista[xi, alaraja[i]];  /* Kyllä: j on rivin 1. vapaan esiintymän sarakeindeksi. */
            V1: CASE j OF
                TAPAUS 1:  (j > kynnysarvo)  /* Ei ollut dominantti k-täsmäys. */
                    yläraja[xi] := alaraja[i];  /* Asetetaan nykyinen alaraja uudeksi ylärajaksi. */
                TAPAUS 2:  (j = kynnysarvo)  /* Rivillä on yksi ei-dominantti k-täsmäys. */
                    yläraja[xi] := alaraja[i];  /* Asetetaan nykyinen alaraja uudeksi ylärajaksi. */
                    alaraja[i] := alaraja[i] + 1;  /* Siirretään alarajaa yhdellä esiintymällä. */
                TAPAUS 3:  (j < kynnysarvo)  /* Nyt löytyi uusi dominantti k-täsmäys. */
                    UusiSolmu(linkkivektori[k], (i, j)); /* Viedään solmu Dk-täsmäysten listaan. */
                    t := Puolitushaku(EsLista[xi], alaraja[i] + 1, yläraja[xi], kynnysarvo);
                        /* Haetaan viimeinen xi, jonka Y-indeksi < kynnysarvo, EsLista[xi]:stä
                           väliltä [alaraja[i] + 1..yläraja[xi]]. Ellei tällaista ole, t = alaraja[i]. */
                        kynnysarvo := j;  /* Päivitetään kynnysarvo ajan tasalle. */
                        yläraja[xi] := alaraja[i];  /* Asetetaan nykyinen alaraja uudeksi ylärajaksi. */
                        alaraja[i] := t + 1;  /* Päivitetään alaraja 1. käyttökelpoiseen esiintymään. */
            ENDCASE (V1)
    ENDFOR (S2)
S1: UNTIL (linkkivektori[k] = ∅);   /* …, kunnes ei enää löydy yhtään k-täsmäystä. */
```

Ratkaise PYA-jono kuten Hirschbergin I algoritmissa.

Tulosta PYA ja sen pituus.

END (ALGORITMI HD1).

11.1.4 Apostolicon ja Guerran I algoritmi (AG1)

ALGORITMI AG1 (X, m, Y, n, σ)**:**

```
/* Konstruoidaan lähiesiintymätaulukko Y-vektorista. */
FOR g := 1, 2, … → s
    LähiEsTauluY[σg, n] := n+1;   /* Alustetaan lähiesiintymätaulukon oikeanpuoleisin sarake. */
S1: FOR j := n–1, n–2, … → 0  /* Rakennetaan lähiesiintymätaulukko valmiiksi. */
```

```
        FOR g := 1, 2, … → σ
            LähiEsTauluY[σg, j] := LähiEsTauluY[σg, j+1];    /* Kopioidaan oikeanpuol. sarakkeen sisältö. */
        LähiEsTauluY[Y[j+1], j] := j+1;  /* Korjataan tilanne seuraavassa positiossa olevan merkin osalta. */
ENDFOR (S1)
FOR i := 1, 2, … → m
    alaraja[i] := 1;   /* Alustetaan rivien käyttökelpoiset esiintymät alkuarvolla 1. */
k := 0;  /* Alustetaan korkeuskäyrän numero. */
löytyi := tosi;  /* Merkitään pseudoluokan D0 edustaja löytyneeksi. */
S2: WHILE ((k ≤ m) AND (löytyi = tosi))  /* Etsintöjä jatketaan, jos luokka Dk ≠ ∅. */
    löytyi := epätosi;  /* Kumotaan tieto löytyneistä Dk+1 –täsmäyksistä. */
    kynnysarvo :=  n + 1;  /* Alustetaan kynnysarvo 'äärettömäksi'. */
    k := k + 1;  /* Siirrytään seuraavalle korkeuskäyrälle. */
    linkkivektori[k] := ¬;  /* Asetetaan Dk-täsmäysten lista tyhjäksi. */
    S3: FOR i := k, k+1, … → m  /* Etsitään Dk+1 –täsmäyksiä riveiltä k → m. */
        entinen := kynnysarvo;  /* Otetaan nykyinen kynnysarvo talteen. */
        j := EsLista[xi, alaraja[i]];  /* Haetaan xi:n ensimmäinen vapaa esiintymä riviltä i. */
        IF (j < kynnysarvo)  /* Löytyikö esiintymä kynnysarvon vasemmalta puolelta? */
            UusiSolmu(linkkivektori[k], (i, j))  /* Kyllä. Viedään täsmäys Dk-listan loppuun. */
            kynnysarvo := j;   /* Päivitetään uudeksi kynnysarvoksi täsmäyksen Y-indeksi. */
            löytyi := tosi;  /* Nostetaan löydettyjen Dk-täsmäysten lippu. */
        IF (xi = yentinen)  /*  Muodostaako sama merkki edellisen k-täsmäyksen? */
            alaraja[i] := monesko_symboli[entinen] + 1  /* Kyllä, sitä seuraava on vapaa. */
        ELSE
            IF (j < entinen)  /* Ei, mutta löytyi eri merkkien välinen Dk-täsmäys. */
                alaraja[i] := monesko_symboli[LähiEsTauluY[xi, entinen]];
            ELSE  /* Ei löytynyt uutta Dk-täsmäystä. Ei tehdä mitään. */
    ENDFOR (S3)
ENDWHILE (S2)
Ratkaise PYA-jono kuten Hirschbergin I algoritmissa.
Tulosta PYA ja sen pituus.
END (ALGORITMI AG1).
```

11.1.5 Apostolicon ja Guerran lineaaritilainen algoritmi (AGL)

```
ALGORITMI AGL (X, m, Y, n, σ):
RatkaisePYA-jono(X, 1, m, m, Y, 1, n, n, PYA);
p := pituus(PYA);
Tulosta muuttujat p ja PYA.
END (ALGORITMI AGL).

PROSEDUURI RatkaisePYA-jono(X, Xalku, Xloppu, m, Y, Yalku, Yloppu, n, var PYA):
/* Tässä oletetaan, että edellä esiteltyä algoritmia AG1 voidaan kutsua määräämällä X:lle (2. parametri) ja Y:lle
   (5. parametri) mielivaltaiset alkuindeksit tavanomaisen ykkösen asemesta. Lisäksi oletetaan ositteen
   kynnysarvovektori (7. parametri) ja PYAn pituus (8. parametri) AG1:n muuttujaparametreiksi. */
IF ((n = 0) OR (m = 1))
    Ratkaise triviaali aliongelma HIL-algoritmissa esitetyllä tavalla.
```

ELSE /* Puolitetaan ongelma X:n suhteen. Määrätään X:n alkuliitteen X[Xalku..Xalku + ⌈m/2⌉ – 1] ja Y:n erimittaisten alkuliitteiden sekä X:n käännetyn loppuliitteen X*[Xloppu..Xloppu – ⌊m/2⌋ + 1] ja Y:n eripituisten käänteisten loppuliitteiden kynnysarvot AG1-algoritmilla, josta dominanttitäsmäysten kirjanpito on poistettu. Hakurakenteena käytetään lähiesiintymävektoria tarvittavine aputietorakenteineen. */

AG1(X, Xalku, ⌈m/2⌉, Y, Yalku, n, $KA_{alkuosa}$, $p_{alkuosa}$); /* Ensin ositteille X[1..Xalku + ⌈m/2⌉ – 1] ja Y[Yalku..Yloppu] … */

AG1(X, Xalku+⌈m/2⌉, ⌊m/2⌋, Y, Yalku, n, $KA_{loppuosa}$, $p_{loppuosa}$); /*… ja tämän jälkeen vielä ositteille X*[Xloppu..Xloppu – ⌊m/2⌋ + 1] ja Y*[Yloppu..Yalku]. */

i := EtsiMaksimi_iplusj($KA_{alkuosa}$, $p_{alkuosa}$, $KA_{loppuosa}$, $p_{loppuosa}$, n); /* Etsitään Y:n osituskohta, joka maksimoi alku- ja loppuosien pisimpien yhteisten alijonojen summan.*/

RatkaisePYA-jono(X, Xalku, Xalku+⌈m/2⌉–1, ⌈m/2⌉, Y, Yalku, Yalku+$KA_{alkuosa}$[i]–1, $KA_{alkuosa}$[i], S1); /* Rekursiivinen kutsu alkuosalle. */

RatkaisePYA-jono(X, Xalku + ⌈m/2⌉, Xloppu, ⌊m/2⌋, Y, Yalku + $KA_{alkuosa}$[i], Yloppu, Yloppu – $KA_{alkuosa}$[i], S2); /* Rekursiivinen kutsu loppuosalle. */

PYA = S1 || S2; /* Yhdistetään saadut osaratkaisut. */

END (PROSEDUURI RatkaisePYA-jono)

FUNKTIO EtsiMaksimi_iplusj($KA_{alkuosa}$, $p_{alkuosa}$, $KA_{loppuosa}$, $p_{loppuosa}$, n)**:**

maksimipya := -1; /* Alustetaan tarkasteltavan ongelman PYAn pituus negatiiviseksi. */

i := 0;

S1: WHILE ((i ≤ n) **AND** ($KA_{alkuosa}$[i] ≤ n)) /* Muodostetaan paristtaiset summat … */

j := 0;

S2: WHILE ($KA_{alkuosa}$[i] < n – $KA_{loppuosa}$[j]) /* … alku- ja loppuosien PYA-pituuksista, ja … */

IF (i + j > maksimipya); /* … selvitetään niiden maksimi. */

maksimipya := i + j; /* Päivitetään aliongelman PYAn uusi maksimipituus. */

katkYind := j; /* Muistetaan vasemmanpuoleisimman maksimikohdan Y-indeksi. */

ENDIF

j := j + 1; /* Siirrytään eteenpäin loppuosan kynnysarvovektorissa. */

ENDFOR (S2)

i := i + 1; /* Siirrytään eteenpäin alkuosan kynnysarvovektorissa. */

ENDFOR (S1)

EtsiMaksimi_iplusj := katkYind;

END (FUNKTIO EtsiMaksimi_iplusj)

11.1.6 Chinin ja Poonin algoritmi (CPO)

ALGORITMI CPO (X, m, Y, n, σ)**:**

/* Konstruoidaan lähiesiintymätaulukko Y-vektorista. */

FOR g := 1, 2, … → σ

LähiEsTaulu[$σ_g$,n] := n+1; /* Alustetaan lähiesiintymätaulukon oikeanpuoleisin sarake. */

S1: FOR j := n–1, n–2, … → 0 /* Rakennetaan lähiesiintymätaulukko valmiiksi. */

FOR g := 1, 2, … → s

LähiEsTaulu[$σ_g$, j] := LähiEsTaulu[$σ_g$, j+1]; /* Kopioidaan oikeanp. sarakkeen sisältö. */

LähiEsTaulu[Y[j+1], j] := j+1; /* Korjataan tilanne seuraavassa positiossa olevan merkin osalta. */

ENDFOR (S1)

```
/*  Konstruoidaan symbolijärjestystaulukko X-vektorista. */
FOR i := 1, 2, … → σ
    SymbJTaulu[i, m] := m+1;   /*  Alustetaan  symbolijärjestystaulukon oikeanpuoleisin sarake. */
S2: FOR i := m – 1, m – 2, … → 0 /* Rakennetaan symbolijärjestystaulukko valmiiksi. */
    SymbJTaulu[1, i] := i+1;  /* Taulukon ylin rivi sisältää järjestyksessä arvot ykkösestä m+1:een. */
    z := 1;   /*  z on laskuri, joka voi saada arvoja 1:stä aakkoston kokoon, s:ään. */
   WHILE ((z < σ) AND (SymbJTaulu[z, i+1] ≤ m) AND (X[i+1] ≠ X[SymbJTaulu[z, i+1]]))
        z := z + 1;   /*  Montako erilaista X:n merkkiä esiintyy paikan i+1 jälkeen ennen kuin merkki
                          X[i+1] kohdataan uudelleen tai syöte loppuu kesken? Muuttuja z sisältää
                          tämän tiedon ykkösellä lisättynä.  */
       SymbJTaulu[z, i] := SymbJTaulu[z-1, i+1]; /* Kopioidaan arvo oikealta yhtä riviä ylempää. */
    FOR g := z+1, z+2, … → σ  /* Jokaista  aakkoston eri merkkiä ei kohdattu. */
        SymbJTaulu[g, i] := SymbJTaulu[g, i+1];   /*  Kopioidaan arvo oikealta samalta riviltä. */
ENDFOR (S2)
LuoUusiSolmu(solmu);  /* Luodaan uusi solmu, johon voidaan tallentaa D0:n sijainti. */
solmu^.rivi := 0;   /* Asetetaan alkusolmulle rivinumero …  */
solmu^.sarake := 0;  /*  … ja sarakenumero. */
solmu^.edellinen := solmu;   /*  Solmun (0,0) edeltäjäksi asetetaan solmu (0,0) itse. */
solmu^.seuraava := →↓;    /* Kenttä seuraava asetetaan tyhjäksi. */
D0 := solmu;  /* Asetetaan D0 osoittamaan solmuun (0, 0). */
k := 0;      /*  Aloituskorkeuskäyrän numero  on nolla.  */
S3: WHILE Dk ≠ nil  /* Otetaan k. kontuuri käsiteltäväksi, jos sillä on täsmäyksiä. */
    MaxSarake := n+1;   /* Asetetaan sarake, jota pienemmästä uusi täsmäys on löydettävä. */
    LuoUusiSolmu(UusiSolmu);   /* Varataan muistia Dk+1 –täsmäyksen kirjaamista varten. */
    Dk+1 := UusiSolmu;      /* Perustetaan tyhjä k+1 -täsmäysten lista.  */
    solmu := Dk;    /* Asetetaan muuttuja solmu osoittamaan Dk-täsmäyslistan alkuun. */
    S4: WHILE solmu ≠ nil  /* Kontuurin Dk solmut käydään läpi järjestyksessä.*/
        IF (solmu^.seuraava ≠ nil)     /* Onko kyseessä korkeuskäyrän Dk viimeinen solmu?  */
            MaxRivi := solmu^.seuraava^.rivi;  /* Ei. Etsitään seuraavan solmun sijaintirivi. */
        ELSE
            MaxRivi := m;  /* Kyllä. Voidaan edetä tarvittaessa X-vektorin loppuun asti. */
        merkki := 1;   /* Otetaan lähin mahdollinen X:n merkki käsittelyyn. */
        sarake := solmu^.sarake; /* Otetaan laajennettavan solmun Y-indeksi talteen. */
        MinSarake := sarake + 1; /* Asetetaan alin hyväksytty sarake, josta k+1 –täsmäys voi löytyä. */
        rivi := SymbJTaulu(merkki, solmu^.rivi);  /* Etsitään lähin vielä kohtaamaton X:n merkki.*/
        S5: WHILE (rivi ≤ MaxRivi) AND (merkki ≤ s)
            sarake := LähiEsTaulu(X[rivi], sarake);   /* Etsitään kyseisen merkin lähin täsmäys Y:stä.*/
            IF (sarake < MaxSarake)   /*  Löydettiin dominantti k+1 -täsmäys. */
                UusiSolmu^.rivi := rivi;   /*  Tallennetaan uuden täsmäyksen X-indeksi … */
                UusiSolmu^.sarake := sarake;   /*  … kuten myös Y-indeksi.  */
                UusiSolmu^.edellinen := solmu;   /*  Asetetaan linkki edeltäjäsolmuun. */
                LuoUusiSolmu(UusiSolmu);   /*  Varataan muistia uudelle Dk+1 –täsmäykselle. */
                UusiSolmu := UusiSolmu^.seuraava;  /* Siirretään osoitinta UusiSolmu eteenpäin. */
                MaxSarake := sarake;   /*  Rajoitetaan hyväksyttävää sarakehaarukkaa. */
            merkki := merkki + 1;   /*  Valmistaudutaan valitsemaan uusi merkki X-vektorista. */
            rivi := SymbJTaulu(merkki, solmu^.rivi);  /* Etsitään lähin vielä kohtaamaton X:n merkki.*/
        ENDWHILE (S5)
        UusiSolmu := →↓; /* Terminoidaan lista Dk+1. */
        solmu := solmu^.seuraava;  /* Siirrytään korkeuskäyrän k seuraavaan solmuun. */
```

```
    ENDWHILE (S4)
    k := k+1;   /* Ei enää solmuja kontuurilla k. Siirrytään tutkimaan seuraavaa kontuuria. */
ENDWHILE (S3)
/* Rakennetaan PYA kulkemalla taaksepäin pisintä polkua. */
k := k – 1;   /* Algoritmi kasvattaa k:n arvoa yhtä suuremmaksi kuin muodostuneen PYAn pituus on. */
solmu := Dk;  /*  Solmu osoittaa nyt ylimmän korkeuskäyrän alkuun. */
Tulosta PYAn pituus k.
WHILE (k ≥ 1)  /*  Kerätään PYAaan kuuluvat merkit S-vektoriin. */
    S[k] := X[solmu^.rivi];   /*  Tallennettaan PYAan kuuluva k:s merkki. */
    solmu := solmu^.edellinen;  /*  Siirrytään linkkiä pitkin edeltäjäsolmuun. */
    k := k – 1;  /* Pienennetään PYA-jonon indeksilaskuria. */
Tulosta PYA-jono S.
END (ALGORITMI CPO).
```

11.2 Riveittäin laskentaa suorittavat menetelmät

11.2.1 Hirschbergin lineaaritilainen algoritmi (HIL)

```
ALGORITMI HIL (X, m, Y, n):
RatkaiseRekursiivisestiHIL(m, n, X, Y, S);  /* PYAn muodostavan jonon määrääminen. */
pyapituus := pituus(S);  /* PYAn pituus on listan S solmujen lukumäärä. */
Tulosta PYA ja sen pituus muuttujista S ja pyapituus.
END (ALGORITMI HIL).

PROSEDUURI RatkaiseRekursiivisestiHIL(m, n, X, Y, var S):
E1: IF (n = 0)   /* Rekursion 1. kantatapaus: onko syötejonon Y osite tyhjä jono? */
        S := ∅;  /* Kyllä. Samoin myös ositteen PYA-jono on tyhjä. */
    ELSE
     E2: IF (m = 1)  /* Onko vektori X kutistunut yhden pituiseksi? */
            j := 0;  /* Kyllä. Yritetään löytää xi:n kanssa täsmäävää merkkiä … */
            REPEAT  /* … Y:n tarkasteltavasta ositteesta niin pitkään … */
                j := j + 1;  /*  … kunnes sellainen löytyy … */
            UNTIL ((j > n) OR (X[i] = Y[j]));  /* … tai Y:n osite loppuu kesken. */
            IF (j ≠ n + 1)  /* Löytyikö etsittyä merkkiä Y:stä? */
                S := xi  /*  Kyllä. Otetaan se PYAan. */
            ELSE
                S := ∅  /*  Alueella ei esiintynyt PYAan kuuluvaa merkkiä. */
        ELSE
          i := ⌊m / 2⌋;  /* Halkaistaan ongelma puolittamalla X kahteen ositteeseen. */
          LaskePituudet(i, n, X[1..i], Y[1..n], alkuosa);  /* Ratkaistaan alkuosan PYA … */
          LaskePituudet(m–i, n, X*[m..i+1], Y*[n..1], loppuosa)  /* … ja loppuosan PYA. */
          p := -1;  /* Tehdään alustus PYAn pituudelle. */
          S1: FOR j := 0, 1, … → n  /* Muodostetaan parittaiset summat  … */
              parisumma := alkuosa[j] + loppuosa[n–j];  /* … alku- ja loppuosien PYA-pituuksista ja … */
              IF (parisumma > p)   /* selvitetään niiden maksimi. */
```

```
                p := parisumma;
                katkYind := j  /* Muistetaan vasemmanpuoleisimman maksimikohdan Y-indeksi. */
            ENDFOR (S1)
            RatkaiseRekursiivisestiHIL(i, katkYind, X[1..i], Y[1..katkYind], S1); /* Rekursio alkuosalle ja … */
            RatkaiseRekursiivisestiHIL(m–i, n–katkYind, X*[m..i+1], Y*[n..katkYind+1], S2);/* loppuosalle.*/
            S := S1 || S2  /* Yhdistetään osaratkaisut katenoimalla jonot S1 ja S2. */
        ENDIF (E2)
    ENDIF (E1)
END (PROSEDUURI RatkaiseRekursiivisestiHIL)

PROSEDUURI LaskePituudet(m, n, X, Y, var nykyrivi):
FOR j := 0, 1, … → n
        nykyrivi[j] := 0;  /* Alustetaan nykyrivin positiot nolla-arvoilla. */
S2: FOR i := 1, 2, … → m  /* Tutkitaan koko X-vektorin osite loppuun asti. */
        FOR j := 1, 2, … → n
            edellrivi[j]  := nykyrivi[j];  /* Kopioidaan viimeksi tutkitun rivin tiedot vektoriin edellrivi. */
    IF (X[i] = Y[j])  /* Täsmäävätkö merkit xi ja yj keskenään? */
        nykyrivi[j] := edellrivi[j–1] + 1  /* Kyllä. Kopioidaan arvo ylävasemmalta yhdellä lisättynä. */
    ELSE
        nykyrivi[j] := Max{nykyrivi[j–1], edellrivi[j]} /* Eivät. Otetaan vasemman ja ylemmän maksimi. */
ENDFOR (S2)
END (PROSEDUURI LaskePituudet)
```

11.2.2 Huntin ja Szymanskin algoritmi (HSZ)

```
ALGORITMI HSZ (X, m, Y, n, σ):
FOR g := 1, 2, … → σ  /* Alustetaan merkkien esiintymälistat tyhjiksi. */
        EsLista[g] := nil;
FOR j := 1, 2, … → n /* Täytetään esiintymälistat käänteisessä järjestyksessä.*/
        EsSolmu := LuoUusiSolmu(j);  /* Perustetaan uusi solmu, jonka sijaintikenttään viedään j …  */
        LisääEnsimmäiseksi[EsLista[yj], EsSolmu];  /* .. ja joka viedään yj:n esiintymälistan alkuun. */
Kynnysarvot[0] = 0;  /* Asetetaan alkuarvo kynnysarvovektorin nollapositiolle. */
FOR k := 1, 2, … → m
        Kynnysarvot[k] := n+1;   /* Alustetaan kynnysarvovektorin muut positiot. */
        linkkivektori[k] := nil;   /*  Asetetaan linkkivektoriin pysäytysarvot. */
S1: FOR i := 1, 2, … → m
    EsSolmu := EsLista[X[i]];   /*  Tutkitaan xi:n esiintymälistan 1. solmu. */
    S2: WHILE (EsSolmu ≠ nil)  /* Käydään xi:n kaikki esiintymät läpi. */
        j := EsSolmu^.sijainti;   /*  Otetaan nykyisen esiintymän Y-indeksi talteen. */
        Etsi puolitushaulla väliltä 1..m pienin k, jolle pätee Kynnysarvot[k–1] < j ≤ Kynnysarvot[k]
        IF  (j < Kynnysarvot[k])   /* Päivittääkö uusin esiintymä jotain kynnysarvoa? */
            Kynnysarvot[k] := j;   /* Kyllä. Tehdään päivitys paikkaan Kynnysarvot[k]. */
            IF (linkkivektori[k] = nil) /* Löydettiinkö ensimmäinen k-täsmäys. */
                linkkivektori[k]^.ens := LuoUusiLinkkiSolmu(i, j, linkkivektori[k–1])
            linkkivektori[k] := LuoUusiLinkkiSolmu(i, j, linkkivektori[k–1])  /* Linkki edeltäjään. */
    EsSolmu := EsSolmu^.seuraava;
    ENDWHILE (S2)
ENDFOR (S1)
```

```
/* Ratkaistaan PYAn muodostava jono. */
k := Max{t | Kynnysarvot[t] < n + 1};  /* Muuttujaan k tallentuu PYA(X, Y):n pituus. */
Tulosta PYAn pituus muuttujasta k.
PYA_osoitin := linkkivektori[k]^.ens;  /* Asetetaan osoitin linkkivektorin k. position 1. solmuun. */
WHILE (PYA_osoitin ≠ 0)  /* Palautetaan lopuksi PYA. */
        PYA[k] := X[PYA_osoitin^.xind];  /* Tulostetaan yksi PYAn merkki kerrallaan lopusta alkaen. */
        k := k–1;  /* Pienennetään silmukkalaskuria. */
        PYA_osoitin := PYA_osoitin^.isä;  /*  Siirrytään PYAssa alkuun päin isälinkkiä pitkin. */
Tulosta PYA-jono vektorista PYA.
END (ALGORITMI HSZ).
```

11.2.3 Mukhopadhyayn algoritmi (MUK)

```
ALGORITMI MUK (X, m, Y, n, σ):
Muodosta vektorit EsListat ja ListanAlku selaamalla Y-vektori kahdesti alusta loppuun.
pyapituus := 0;  /* Alustetaan PYAn pituus nollaksi. */
RTind := 1;  /* Alustetaan vektoriin RivienTiedot liittyvä indeksilaskuri. */
kynnysarvot[0] := n + 1;  /* Alustetaan kynnysarvovektorin nollas positio. */
S1: FOR i := m, m–1, … → 1
    pyapitenee := epätosi;   /* Ilmaisee, onko PYA pidentynyt aikaisemmasta arvostaan. */
    edellpya := pyapituus;  /* Muuttuja edellpya ilmaisee PYAn pituuden riviä aikaisemmin. */
    alku := ListanAlku[i];  /* Muuttuja alku ilmaisee i. merkin täsmäyslistan pituuden. */
    IF (alku = 0)  /* x_i ei täsmää yhteenkään Y:n merkkiin */
        RivinAlku[i] := 0  /* i:nnen rivin täsmäysluokkien lista asetetaan puuttuvaksi. */
    ELSE
        RivinAlku[i] := RTind;  /* Asetetaan alkupositio i:nnen rivin täsmäystietojen listalle. */
        lkm := EsListat[alku];  /* Montako täsmäystä riviltä i löydetään? */
        RivienTiedot[RTind] := alku;  /*  Kirjataan i:nnen merkin täsmäyslistan alkukohta. */
        RTind := RTind + 1;
        S2: FOR j := alku + 1, alku + 2, … → alku + lkm
            E1: IF (EsListat[j] < kynnysarvot[edellpya]) /* Piteneekö PYA edellisestä? */
                E2: IF (pyapitenee = epätosi)  /* Kyllä vaan. Onko PYA jo pidentynyt nykyrivillä? */
                        pyapituus := pyapituus + 1;  /* Ei ole. Pidennetään PYAa yhdellä. */
                        pyapitenee := tosi;  /* PYA ei enää voi pidentyä rivillä i. */
                ENDIF (E2)
                RivienTiedot[RTind] := pyapituus;
                RTind := RTind + 1;
                kynnysarvot[pyapituus] := EsListat[j];
            ELSE
                Etsi puolitushaulla sellainen k, että EsListat[j] < kynnysarvot[k], kun k ∈ [0..edellpya-1].
                RivienTiedot[RTind] := k + 1;  /* Tallennetaan täsmäyksen luokka. */
                RTind := RTind + 1;[177]  /* Siirretään vektorin RivienTiedot positiolaskuria. */
                kynnysarvot[k + 1] := EsListat[j]  /* Päivitetään kynnysarvovektoria. */
            ENDIF (E1)
        ENDFOR (S2)
```

[177] Tämä lause on jäänyt vahingossa puuttumaan alkuperäisartikkelista. Lauseen pois jättäminen aiheuttaisi vektorin *RivienTiedot* sisällön sekoamisen päälle kirjoittumisten tähden.

ENDFOR (S1)
PYAnKerääminen(RivinAlku, RivinTiedot, X, m, n, pyapituus);
END (ALGORITMI MUK).

PROSEDUURI PYAnKerääminen (RivinAlku, RivinTiedot, X, m, n, p)**:**
i := 1; /* Aloitetaan tarkastelu matriisin M ylimmältä riviltä. */
k := p; /* Etsitään aluksi luokkaa C_p edustava täsmäys. */
WHILE niin pitkään kuin ((i ≤ m) **AND** (k > 0)) /* Niin pitkään kunnes PYA-jono on valmis ... */
Etsi puolitushakua käyttämällä, löytyykö riviltä i luokan C_k edustajia. Asia selviää etsimällä k:ta vektorista RivienTiedot indeksialueelta, jonka rajoina ovat RivinAlku[i]+1..RivinAlku[i+1]-1.
Jos i = m, ylärajaksi valitaan pituus(RivinAlku).
IF (k löytyi edellisessä puolitushaussa) /* Onko nykyrivillä i luokan C_k edustajia? */
PYA[p – k + 1] = X[i]; /* Kyllä. Täsmäys kelpaa PYAan. */
k := k – 1; /* Lähdetään etsimään seuraavaa PYAan kuuluvaa täsmäystä. */
i := i + 1; /* Siirrytään seuraavalle riville. */
Tulosta PYA-jono vektorista PYA ja sen pituus p.
END (PROSEDUURI PYAnKerääminen)

11.2.4 Hsun ja Dun II algoritmi (HD2)

ALGORITMI HD2 (X, m, Y, n, σ)**:**
Perusta vektorimuotoiset esiintymälistat kaikkien X:ssä esiintyvien merkkien sijaintipaikoista Y:ssä.
kynnysarvot[0] := 0; /* Alustetaan kynnysarvovektorin nollas positio. */
FOR k := 1, 2, … → m
kynnysarvot[k] := ∞; /* Alustetaan kynnysarvovektorin muut positiot arvolla ääretön. */
maxpituus := 0; /* Alustetaan PYAn pituus nollaksi. */
FOR j := 1, 2, … → σ
frekv[s_j] := 0; /* Alustetaan syöttöaakkosten merkkien frekvenssit nolliksi. */
FOR j := 1, 2, … → n
frekv[y_i] := frekv[y_i] + 1; /* Alustetaan syöttöaakkosten merkkien frekvenssit nolliksi. */
S1: FOR i := 1, 2, … → m
k := maxpituus; /* Asetetaan yläraja kierroksen aikana tarkasteltaville kynnysarvovektorin positioille. */
u := frekv[x_i]; /* Selvitetään, montako kertaa x_i esiintyy Y:ssä.
S2: WHILE (k ≥ 0) **AND** (u > 0)
j := Etsi(EsLista[x_i], kynnysarvot[k], kynnysarvot[k+1], 1, u); /* Etsitään x_i:n täsmäyslistasta indeksialueelta 1..u pienin j, joka täyttää ehdon kynnysarvot[k] < j < kynnysarvot[k+1] käyttämällä puolitushakua. Ellei tällaista j:tä ole olemassa, j = 1 + Min{t|EsLista[x_i, t] ≥ kynnysarvot[k]}. Ellei tällaista t:täkään ole olemassa, j := frekv[x_i] + 1 */
IF kynnysarvot[k] < j < kynnysarvot[k+1] /* Löytyikö vaatimukset täyttävä j? */
kynnysarvot[k+1] := EsLista[x_i, j];
/* Kyllä, päivitetään kynnysarvovektorin k+1. position arvoa. */
IF (linkkivektori[k+1]^.ens = nil) /* Löytyikö ensimmäinen k+1 –täsmäys? */
linkkivektori[k+1]^.ens := LuoUusiSolmu(i, EsLista[x_i, j, linkkivektori[k])
linkkivektori[k+1] := LuoUusiSolmu(i, EsLista[x_i, j], linkkivektori[k]);
/* Linkitetään uusi D_{k+1}-täsmäys lähimpään D_k-edeltäjäsolmuunsa. */
IF (k = maxpituus)
maxpituus := maxpituus + 1; /* PYA piteni yhdellä aikaisemmasta. */

```
        k := k – 1;  /* Valmistaudutaan aloittamaan luokan D_{k–1} täsmäyksen etsiminen. */
        u := j – 1; /* Siirretään merkin x_i hakualueen oikeaa rajaa vasemmalle sen esiintymälistassa. */
    ENDWHILE (S2)
ENDFOR (S1)
/* Ratkaistaan PYAn muodostava jono. */
k := maxpituus;  /* Muuttujaan k tallentuu PYA(X, Y):n pituus. */
PYA_osoitin := linkkivektori[k]^.ens;  /* Asetetaan osoitin linkkivektorin k:nnen position 1. solmuun. */
WHILE (k ≠ 0)  /* Palautetaan lopuksi PYA. */
        PYA[k] := X[PYA_osoitin^.xind];  /* Tulostetaan yksi merkki kerrallaan lopusta alkaen. */
        k := k–1;  /* Pienennetään silmukkalaskuria. */
        PYA_osoitin := PYA_osoitin^.isä;  /*  Siirrytään PYAssa alkuun päin isälinkkiä pitkin. */
Tulosta PYA-jono vektorista PYA ja sen pituus muuttujasta maxpituus.
END (ALGORITMI HD2).
```

11.2.5 Allisonin ja Dixin algoritmi (ADI)

```
ALGORITMI ADI (X, m, Y, n):
FOR i := 1, 2, … → σ /* Muodostetaan esiintymäbittijonot kullekin merkille … */
    EBJ[i, *] := 0;                    /*  … ja alustetaan ne nollilla. */
FOR j := n, n–1, … → 1  /* Asetetaan EBJ-taulukkoon ykköset paikoilleen. */
    EBJ[Y[j], j] := 1;  /* Asetetaan EBJ:hin merkin Y[j] riville ykkönen kohtaan j. */
FOR j := 1, 2, … → n
    M[0, j] := 0;  /* Alustetaan matriisin M nollas rivi nollilla. */
S1: FOR i := 1, 2, … → m  /* Syötevektori X tutkitaan alusta loppuun. */
    edellrivi := M[i–1]; /* Matriisin rivi M_i perustuu sekä riviin M_{i–1} … */
    z := edellrivi OR EBJ[X[i]];
   S2: FOR jokainen M_{i–1}:n segmentti
        Tee segmentille looginen siirto vasemmalle ja sijoita sen oikeaan reunaan ykkönen. Tallenna muutokset
       bittijonoon t vastaavaan paikkaan kuin bittijonossa M_{i–1}.
   ENDFOR (S2)
    z' := z – t; /* Vähennetään z:sta edellä loogisista siirroista koostettu bittijono t ja viedään tulos z':uun. */
    u := z ⊕ z';  /* Muodostetaan z:n ja z':n poissulkeva TAI-operaatio. */
    M_i := z AND u;  /* Matriisin rivi M_i saadaan z:n ja u:n JA-operaatiolla. */
ENDFOR (S1)
p := ykkösbittien lukumäärä rivillä M_m  /* PYAn pituus = ykkösten lukumäärä matriisin ylimmällä rivillä. */
Ratkaise PYA samaan tapaan kuin algoritmissa HIL.
Tulosta PYA-jono ja sen pituus.
END (ALGORITMI ADI).
```

11.2.6 Apostolicon ja Guerran II algoritmi (AG2)

Algoritmissa AG2 käytettyjä merkintöjä:

Merkintä *AEL*[x_i] tarkoittaa *X*:n *i.* merkin aktiivista esiintymälistaa *Y*:ssä.

Funktio *Ensimmäinen*(*L*) palauttaa argumenttinaan saamansa listan ensimmäisen alkion. Jos lista on tyhjä, funktio palauttaa arvon $n + 1$.

Funktio *Etsi*(*u, L*) palauttaa listasta *L* pienimmän alkion, joka on suurempi kuin *u*. Jos lista on tyhjä, tai hakuehdon toteuttavaa alkiota ei löydy listasta, funktio palauttaa arvon $n + 1$.

Funktio *Lisää*(*u, L*) lisää alkion *u* listaan *L*.

Funktio *MoneskoSolmu*(*u, L*) palauttaa solmun järjestysnumeron listassa *L*. Jos solmu *u* sijaitsee viimeisenä funktio palauttaa arvon $n + 1$.

Funktio *Poista*(*u, L*) poistaa alkion *u* listasta *L*.

Funktio *Päivitä*(*u, j, L*) muuttaa listan *j*:ntenä esiintyvän solmun arvon *u*:ksi.

ALGORITMI AG2 (X, m, Y, n, σ) **(ilman jonon ylläpitoa varten tehtävää kirjanpitoa):**

Perusta jokaista X:n merkkiä kohti nousevaan suuruusjärjestykseen aktiivinen dynaaminen esiintymälista AEL merkin sijaintipaikoista vektorissa Y.
Alusta dynaamisen kynnysarvovektorin KL luokka 0 pseudoarvoilla 0 ja luokka 1 määrittelemättömäksi arvolla n + 1.

```
S1: FOR i := 1, 2, … m   /* Tarkastellaan kaikki X:n merkit. */
    ensimm := Ensimmäinen(AEL[xi]);   /* Valitaan xi:n aktiivisen esiintymälistan 1. esiintymä. */
    IF (ensimm = n + 1)  /* Onko merkin xi aktiivinen esiintymälista tyhjä? */
        löytyy := epätosi;  /* Kyllä. Ei kannata tutkia riviä i, sillä sieltä ei löydy dominanttitäsmäyksiä. */
    ELSE
        löytyy := tosi;   /* S2:ta on suoritettava ainakin kerran. */
    S2: WHILE (löytyy = tosi)  /* Tutkitaan riviä i niin pitkään … */
        entinen := Etsi(ensimm, KL);  /* entinen = Min{ z | z ∈ KL: z > ensimm} */
        j := MoneskoSolmu(entinen, KL);  /* Mihin luokkaan uusi dominantti täsmäys kuuluu? */
        Poista(ensimm, AEL[xi]);  /* Poistetaan ensimm merkin xi aktiivisesta täsmäyslistasta. */
        IF (j = n + 1)  /* PYA pitenee löydetyn dominanttitäsmäyksen ansiosta. */
            löytyy := epätosi;   /* Riviltä i ei enää voi löytyä uusia dominanttitäsmäyksiä. */
            Lisää(ensimm, KL) /* Lisätään arvo ensimm kynnysarvolistaan oikealle paikalleen. */
        ELSE  /* Löydetty täsmäys siirtää jonkin aiemmin löydetyn luokan kontuuria. */
            Päivitä(ensimm, j, KL);  /* Päivitetään kynnysarvovektoria. */
            merkki := Y[entinen];  /* Selvitetään, mikä merkki muodosti edeltäjätäsmäyksen.*/
            ensimm := Etsi(entinen, AEL[merkki]);  /*Min { z | z ∈ AEL[merkki]: z > entinen } */
            Lisää(entinen, AEL[merkki]);  /* Aktivoidaan vanha kynnysarvo uudelleen. */
            IF (ensimm = n + 1)  /*Onko jokin xi:n esiintymä vanhaa kynnysarvoa kauempana? */
                löytyy := epätosi;  /* Ei. Rivin i käsittely voidaan lopettaa. */
    ENDWHILE (S2)  /* … kuin riviltä i löytyy dominanttitäsmäyksiä. */
ENDFOR (S1)
END (ALGORITMI AG2).
```

11.2.7 Kuon ja Crossin algoritmi (KCR)

ALGORITMI KCR (X, m, Y, n, σ) **(muutokset HSZ-algoritmiin):**

```
/*  Esiintymälistojen muodostaminen  */
  Kuten HSZ:ssa, mutta silmukkalaskuri j saa arvot 1, …, n nousevassa järjestyksessä.
linkkivektori[0] := (0, 0); /* Asetetaan pseudosolmu linkkivektori[0]:n alkuun. */
/* Silmukat S1 ja S2 korvataan seuraavilla uusilla versioilla. */
S1: FOR i := 1, 2, … → m
    k := 0;  /* Alustetaan PYAn pituus nollaksi. */
    tilap := 0;  /* Muuttujaan tilap otetaan talteen arvo Kynnysarvot[k] ennen päivitystä. */
    edellpäiv := 0;  /* Ilmoittaa, mihin kohtaan Kynnysarvot[k]:ssa tehdään seuraava viivytetty päivitys. */
    EsSolmu := EsLista[xi];  /* EsSolmu osoittaa merkin xi esiintymälistan alkuun. */
    S2: WHILE xi:n esiintymälistassa on solmuja
        j := EsSolmu[xi]^.sijainti;  /* esiintymän indeksi Y-vektorissa */
        IF (j > tilap)  /* Onko saman luokan täsmäys kuin listan edellinen täsmäys? */
            REPEAT  /* Ei ole. Selvitetään täsmäyksen luokka … */
                    k := k+1  /*  .. lineaarihaulla siirtymällä kynnysarvovektorissa … */
            UNTIL (j ≤ Kynnysarvot[k]);  /* … riittävän kauas oikealle. */
            apu := LuoUusiLinkkisolmu(i, j, linkkivektori[k–1]);  /* isälinkin asetus */
            IF (edellpäiv > 0)   /* Onko viivytettyjä päivityksiä tekemättä kynnysarvovektoriin? */
                IF (linkkivektori[edellpäiv] = nil)  /* Kyseessä luokkansa 1. täsmäys? */
                    linkkivektori[edellpäiv]^.ens := linkkisolmu;
                linkkivektori[edellpäiv] := linkkisolmu;   /* Toteutetaan linkkivektorin päivitys.
            linkkisolmu := apu;
            tilap := Kynnysarvot[k];  /*  Seuraavan päivittyvän täsmäyksen Y-indeksin on oltava
                                          > Kynnysarvot[k]. */
            Kynnysarvot[k] := j;   /*  Päivitetään nyt Kynnysarvot[k] uuteen arvoonsa. */
            edellpäiv := k;  /*  Seuraava viivytetty päivitys tullaan tekemään luokkaan k. */
        EsSolmu := EsSolmu^.seuraava
    ENDWHILE (S2)
    IF (edellpäiv > 0)  /* Käsitellään viimeinen viivytetty päivitys, jos sellainen vielä löytyy. */
        IF (linkkivektori[edellpäiv]^.ens = nil)  /* Kyseessä luokkansa 1. täsmäys? */
            linkkivektori[edellpäiv]^.ens := linkkisolmu;
        linkkivektori[edellpäiv] := linkkisolmu   /* Toteutetaan linkkivektorin päivitys.
                                                     Tämän jälkeen rivi i on loppuun käsitelty. */
ENDFOR (S1)
END (ALGORITMI KCR).
```

11.2.8 Rickin II algoritmi (RI2)

ALGORITMI RI2 (X, m, Y, n, σ) **(ilman linkkivektorien ylläpitoa):**

```
Muodosta lähiesiintymätaulukko pidemmästä syötevektorista Y.
FOR i := 1, 2, … → m  /* Alustetaan kynnysarvovektorin eri positiot. */
        kynnys[i].seur := i – 1;  /* position seuraajalinkki */
        kynnys[i].edell := i + 1;  /* position edeltäjälinkki */
```

```
        kynnys[i].arvo := n + 1;  /* Minimaalinen sarake, jota pitkin korkeuskäyrä T_k kulkee. */
        kynnys[i].uusiarvo := n + 1;  /* seuraavassa päivityksessä asetettava sarakearvo */
        kynnys[i].rivi := -2;  /* Millä rivillä kynnys[i] on päivitetty viimeksi? */
kynnys[0].seur := m;  /* Nollannen position seuraaja on m. positio: rengaslistatoteutus. */
kynnys[m].edell := 0;  /* Asetetaan vastaavasti m. position edeltäjä nollaksi. */
kynnys[0].arvo := 0;  /* kynnys[0].arvo ei päivity enää myöhemmin. */
kynnys[0].uusiarvo := 0;  /*  Periaatteessa turha asetus: ei käytetä koskaan hyväksi[178]. */
kynnys[0].rivi := 0;  /* Pseudotäsmäyksen (0, 0) sijaintirivi. */
FOR i := 1, 2, … → s /* Alustetaan merkkien viimeiset esiintymät. */
        viim_es[i] := -1;  /* Yhtään merkkiä ei ole vielä kohdattu. */
max := 0;  /* Muuttujaan max asetetaan PYAn pituus. */
alku := 0;  /*  Viimeksi on päivitetty kynnysarvovektorin positiota 0. */
S1: FOR i := 1, 2, … → m  /* Syötevektori X tutkitaan alusta loppuun. */
    q := alku;  /* Asetetaan q osoittamaan kynnysarvovektorin listaesityksen alkuun. */
    z := 1;  /* Laskuri z pitää kirjaa, monettako listan solmua ollaan tutkimassa. */
    S2: WHILE (kynnys[q].rivi ≥ viim_es[x_i]) /* Voiko esiintyä q + 1 -täsmäystä? */
        /* Kyllä: haetaan tarpeelliset x_i:n esiintymät. */
        j := LähiEsTauluY[x_i, kynnys[q].arvo + 1];  /* Otetaan lähin niistä talteen muuttujaan j. */
        IF (j < kynnys[q + 1].arvo)  /* Löydettiinkö uusi dominantti q + 1 -täsmäys? */
            kynnys[q + 1].uusiarvo := j;  /* Kyllä: kirjataan löytymispaikan sarakeindeksi muistiin.*/
            muutokset[p] := q + 1;  /* Viedään löydetty D_{q+1} –täsmäys päivitettävien jonoon. */
            z := z + 1;  /* Siirretään vektorin muutokset laskuria eteenpäin. */
        q := kynnys[q].seur  /* Haetaan järjestyksessä seuraava luokka kynnysarvolistasta. */
    ENDWHILE (S2)
    S3: FOR j := 1, 2, … → p – 1  /* Käydään päivitettävien jono läpi. */
        q := muutokset[j];  /* Otetaan sieltä j:s alkio. */
        IF (alku ≠ q)  /* Tehdäänkö päivitys samaan luokkaan, jota päivitettiin viimeksi edellä? */
            kynnys[kynnys[q].seur].edell := kynnys[q].edell;
                /* Ei. Poistetaan positio kynnys[q] vanhasta paikastaan näillä kahdella lauseella. */
            kynnys[kynnys[q].edell].seur := kynnys[q].seur;
            kynnys[q].seur := alku;  /* kynnys[q] siirretään listan alkuun …  */
            kynnys[q].edell := kynnys[alku].edell;
            kynnys[kynnys[alku].edell].seur := q;
            kynnys[alku].edell := q;
            alku := q  /* .. näiden viiden lauseen turvin. */
        kynnys[q].arvo := kynnys[q].uusiarvo;  /*  Päivitetään kynnys[q] uuteen … */
        kynnys[q].rivi := i;  /* … sarake- ja riviarvoonsa. */
    ENDFOR (S3)
    IF (kynnys[max + 1].arvo < n + 1)  /* Kasvoiko PYAn pituus rivin i tultua käsitellyksi? */
        max := max + 1;   /* Kyllä. Päivitetään pituus uuteen arvoonsa. */
    viim_es[x_i] := i;  /* Kirjataan symboli x_i esiintyneeksi viimeksi rivillä i. */
ENDFOR (S1)
Tulosta('PYAn pituus on: ', max);  /* Ohjelman suoritus päättyy tähän. */
END (ALGORITMI RI2).
```

[178] Kyseinen arvo asetetaan kuitenkin Rickin teknisessä raportissa [Ric94].

11.3 Syötejonojen loppuliitteitä laajentavat algoritmit

11.3.1 Nakatsun, Kambayashin ja Yajiman algoritmi (NKY)

ALGORITMI NKY (X, m, Y, n)**:**

```
diagpos := m;   /* aloitetaan X-vektorin tarkastelu sen viimeisestä positiosta */
maxpituus := 0;   /* alustetaan PYA(X, Y):n pituus nollaksi */
/* Seuraavaa silmukkaa toistetaan niin pitkään kuin syötemerkkijonojen PYA voi pidentyä. */
S1: WHILE (maxpituus < diagpos)
    i := diagpos;  /* i ilmoittaa nykysijainnin X-vektorissa. */
    j := 1;  /* j kertoo, mikä on seuraava saavutettavissa oleva PYAn pituus. */
    ylinYind := n + 1;  /* Ilmoittaa pienimmän Y-indeksin, jota ei enää tutkita kierroksella. */
    /* Seuraavassa silmukassa lasketaan matriisin M arvot yhdelle antidiagonaalille. */
    S2: WHILE ((i ≠ 0) AND (ylinYind ≠ 0))
        IF ((diagpos = m) OR (j > maxpituus))
            M[i+1, j] := 0;  /* Viedään matriisiin pysäytysalkio. */
        alinYind := Max{1, M[i+1, j]};  /* pienin tarkasteltava Y-indeksi */
        posY := ylinYind – 1;  /* posY ilmoittaa nykysijainnin Y-vektorissa. */
        /* Seuraavassa silmukassa etsitään X:stä luetulle merkille lähin täsmäys Y:stä. */
        S3: WHILE ((posY ≥ alinYind) AND (x_i ≠ y_posY))
            posY := posY – 1;  /* Siirrytään alkua kohti Y-vektorissa etsien X[i]:ää. */
        ENDWHILE (S3)
        /* Löytyikö täsmäystä ja jos löytyi, niin oliko se riittävän läheltä? */
        IF (posY ≥ alinYind)
            ylinYind := posY;   /* x_i löytyi vielä Y-vektorin alkuosasta riittävän läheltä. */
        ELSE
            ylinYind := M[i+1, j];   /*  ei löytynyt, tai aikaisempi polku parempi. */
        M[i, j] := ylinYind;  /* Viedään ylinYind matriisiin M. */
        /* PYA ei pidentynyt edettäessä käsiteltävää diagonaalia pitkin. */
        IF (ylinYind = 0)
            j := j – 1;  /* PYA ei pidentynyt tällä kierroksella. */
        maxpituus := Max{maxpituus, j};  /*  Päivitetään PYAn pituus. */
        j := j + 1;  /* seuraava mahdollinen PYAn pituus. */
        i := i – 1;  /*  Siirrytään X:ssä 1 merkki alkua kohti. */
    ENDWHILE (S2)
    diagpos := diagpos – 1;  /* Rajoitetaan X:n tarkastelualuetta yhtä kapeammaksi. */
ENDWHILE (S1)
/* PYAn muodostavien syötemerkkijonon indeksipaikkojen kerääminen vektoreihin PYAX ja PYAY.
   PYA-jono kerätään vektoreihin PYAX ja PYAY indekseihin [1..maxpituus] käännetyssä
   järjestyksessä. */
IF (ylinYind = 0)   /* Jouduttiinko ulos Y-vektorista? */
    i := i + 2;  /*  Kyllä vaan. Siirretään X-kursoria kahdella eteenpäin. */
ELSE
    i := i + 1;  /* Ei jouduttu. Siirretään X-kursoria yhdellä eteenpäin. */
pyapos := maxpituus;   /*  Asetetaan pyapos osoittamaan PYA-jonon viimeiseen indeksiin. */
S4: WHILE niin pitkään kuin (pyapos > 0)
    S5: WHILE niin pitkään kuin (M[i, pyapos] = M[i+1, pyapos])
```

```
        i := i + 1;   /* Siirrytään seuraavalle riville, kunnes aloitusehto ei enää toteudu. */
    ENDWHILE (S5)
    PYAX[pyapos] := i;   /* Otetaan PYAan kuuluvan täsmäyksen X-indeksi talteen. */
    PYAY[pyapos] := M[i, pyapos];  /*  Tehdään samoin täsmäyksen Y-indeksille. */
    i := i + 1;   /* Siirrytään seuraavalle riville. */
    pyapos := pyapos – 1;   /*  Siirrytään nykyisestä vasemmanpuoleiseen sarakkeeseen. */
ENDWHILE (S4)
Tulosta PYA-jonon pituus maxpituus sekä sen merkit X[PYAX] kullekin indeksille väliltä 1..maxpituus.
END (ALGORITMI NKY).
```

11.3.2 Kumarin ja Ranganin algoritmi (KRA)

```
ALGORITMI KRA (X, m, Y, n):
LaskePYAnPituus(X, Y, m, n, p);  /* Määrätään aluksi pelkkä PYAn pituus, … */
Tulosta PYAn pituus p.
RatkaisePYA-jono(X, Y, m, n, p, S);  /* … jonka jälkeen määrätään PYAn muodostava merkkijono. */
Tulosta PYA-jono vektorista S.
END (ALGORITMI KRA).

PROSEDUURI LaskePYAnPituus(X, Y, m, n, var p):
p := 0;  /* Alustetaan PYAn pituus nollaksi. */
Xyläraja := m + 1;  /* Lähdetään käsittelemään X:ää lopusta alkuun päin. */
WHILE (Xyläraja > p)  /* Yritetään tehostaa Y:n loppuliitteen valintaa. */
        Xyläraja := Xyläraja – 1;  /* Siirrytään X:ssä yksi merkki alkuun päin. */
        TutkiYksiDiagonaali(X, Y, m, n, edelldiag, nykdiag, p, X);  /* Lasketaan L_i(k)-arvot. */
        FOR j :=  0, 1, …,  → p
                edelldiag[j] := nykdiag[j];  /* Kopioidaan uusimman diagonaalin arvot lähtöarvoiksi. */
p := Xyläraja;  /* PYAn pituus on nyt selvillä. */
END (PROSEDUURI LaskePYAnPituus)

PROSEDUURI TutkiYksiDiagonaali(X, Y, m, n, edelldiag, nykdiag, var pyapituus, Xyläraja):
j := 1;  /* Alustetaan loppuliitteiden PYAn pituudeksi 1. */
i := Xyläraja;  /*  Aloituskohta vektorin X tarkastelulle. */
lopetetaan := epätosi;  /* Kontrollimuuttuja, joka rajoittaa alkavan silmukan toistokertojen määrää. */
nykdiag[0] := n + 1;  /*  Nollan mittaiseen PYAan ei tarvita yhtään Y:n merkeistä. */
WHILE ((i > 0) AND (lopetetaan = epätosi))  /* Merkkejä yhä sekä X:ssä että Y:ssä. */
        IF (j > pyapituus)  /* Yritetään kasvattaa aikaisemmin löydettyä PYAn maksimipituutta? */
            Yalaraja := 0   /* Kyllä. Etsittävä merkin tulee löytyä Y-kursorin vasemmalta puolelta. */
        ELSE  /* Ei. Aiemmin on jo löydetty j:n mittainen PYA. */
            Yalaraja := edelldiag[j];  /* Merkin tulee löytyä edelldiag[j]:n oikealta puolelta. */
        posY := nykdiag[j – 1] – 1;  /* Siirrytään Y:ssä yhden merkin verran kohti alkua. */
        WHILE ((posY > Yalaraja) AND (X[i] ≠ Y[posY])) /* Merkkejä muttei täsm. */
                posY := posY – 1;  /* Siirrytään Y:ssä alkua kohti. */
        tilap := Max{posY, Yalaraja}  /* j-täsmäyksen oikeanpuoleisin kelvollinen sijaintipaikka. */
        IF (tilap = 0)  /*  Ei löydy pidempää PYAa tutkittavalta lävistäjältä. */
            lopetetaan := tosi  /* Silmukan suoritus päättyy. */
        ELSE
```

```
            nykdiag[j] := tilap;  /* Päivitetään nykydiagonaalin Li(k)-arvo. */
            i := i – 1;  /* Siirrytään X:ssä yksi merkki alkuun päin. */
            j := j + 1;  /* Lähdetään etsimään yhtä pidempiä Y:n loppuliitteitä. */
pyapituus := j – 1;  /* Maksimi kohdattu PYAn pituus diagonaalilla nykdiag. */
END (PROSEDUURI TutkiYksiDiagonaali)

PROSEDUURI RatkaiseTriviaaliTapaus(X, Y, m, n, p, var S):
TutkiOsite(X, Y, m, n, m–p, diagonaali, pyapituus);  /* Selvitetään ositteen PYAn pituus. */
i := 1;  /* Tehdään tarpeellinen alustus. */
WHILE ((i ≤ p) AND (X[i] = Y[diagonaali[p–i+1])) /* Merkkien on täsmättävä. */
        S[i] = X[i];  /* Kopioidaan ositteen X-syötteen i:s merkki PYA-jonoon S. */
        i := i + 1;  /* Siirrytään X:n seuraavaan merkkiin. */
i := i + 1;  /* Ohitetaan mahdollinen ositteen PYAan kuulumaton merkki. */
WHILE (i ≤ m)  /* Toistetaan, kunnes syöte X loppuu. */
        S[i–1] := X[i];  /* Kopioidaan kohdattu X:n merkki PYA-jonoon. */
        i := i + 1;  /* Siirrytään X:ssä eteenpäin. */
END (PROSEDUURI RatkaiseTriviaaliTapaus)

PROSEDUURI EtsiTäydellinenLeikkaus(X, Y, m, n, φ, var u, var v):
TutkiOsite(X*, Y*, m, n, ⌈φ/2⌉, alkuosa, palkuosa);  /* Tutkitaan käännettyjen jonojen alkuosan PYA. */
FOR j := 0, 1, … → palkuosa  /* Käydään läpi vektori alkuosa, ja muutetaan … */
        alkuosa[j] := n + 1 – alkuosa[j]; /* … Lm–i+1(j)(X*, Y*) -arvot Li*(j)(X, Y) -arvoiksi. */
TutkiOsite(X, Y, m, n, ⌊φ/2⌋, loppuosa, ploppuosa); /* Tutkitaan syötejonojen loppuliitteiden PYA. */
löytyi := epätosi;  /* Kontrolloi, onko täydellinen leikkaus löydetty. */
k := 0;  /* Silmukan laskurimuuttuja alustetaan tässä. */
S1: WHILE (löytyi = epätosi)  /* Täydellinen leikkaus löytyy tässä silmukassa. */
        IF ((k ≤ palkuosa) AND (p – k ≤ ploppuosa) AND (alkuosa[k] < loppuosa[k]))  /* TL:n ehdot. */
            löytyi := tosi;  /* Täydellisen leikkauksen etsintä voidaan lopettaa. */
            u := k + ⌈φ/2⌉;  /* Leikkauksen rivi-indeksi on u … */
            v := alkuosa[k];  /* … ja sarakeindeksi on v. */
        ELSE
            k := k + 1;  /* k:n arvoa kasvatetaan, koska edellinen ei toteuttanut TL:n ehtoja. */
ENDWHILE (S1)
END (PROSEDUURI EtsiTäydellinenLeikkaus)

PROSEDUURI TutkiOsite(X, Y, m, n, roskamäärä, var diagonaali, pyapituus):
pyapituus := 0;  /*  Tehdään tarpeellinen alustus. */
FOR Xyläraja := m, m – 1, … → m – roskamäärä  /* Ratkaistaan ositteen PYA diagonaali */
        TutkiYksiDiagonaali(X, Y, m, n, edelldiag, nykdiag, pyapituus, Xyläraja); /* … kerrallaan. */
        FOR j := 0, 1, … → pyapituus
            edelldiag[j] := nykdiag[j];  /* Määritellään viimeksi tutkittu diagonaali edelliseksi. */
FOR j := 0, 1, … → pyapituus
        diagonaali[j] := edelldiag[j];  /* Palautetaan ositteen viimeisen diagonaalin arvot kutsujalle. */
END (PROSEDUURI TutkiOsite)

PROSEDUURI RatkaisePYA-jono(X, Y, m, n, p, var S):
φ := m – p; /* Muuttuja φ esittää tarkasteltavan ongelman roskamäärää. */
IF (φ < 2)  /* Onko kyseessä triviaali osaongelma? */
```

```
    RatkaiseTriviaaliTapaus(X, Y, m, n, m–p, L, p, S);  /* On. Saavutettiin rekursion kanta. */
ELSE
    EtsiTäydellinenLeikkaus(X, Y, m, n, φ, u, v);  / Ei. Ongelma ositettava vielä pienemmäksi. */
    RatkaisePYA-jono(X[1..u], Y[1..v], u, v, u–⌈φ/2⌉, S1);  /*  Rekursiiviset kutsut alkuosalle … */
    RatkaisePYA-jono(X[u+1..m], Y[v+1..n], m–u, n–v, m–u–⌊φ/2⌋, S2);  /* … ja loppuosalle. */
    S = S1 || S2  /* Yhdistetään saadut osaratkaisut katenoimalla jonot S1 ja S2. */
END (PROSEDUURI RatkaisePYA-jono)
```

11.4 Monisuuntaisesti laskentaa suorittavat algoritmit

11.4.1 Rickin I algoritmi (RI1)

ALGORITMI RI1 (X, m, Y, n, σ) **(PYA-jonon palauttava, täydennetty versio):**

```
Muodosta syöttöaakkoston symboleille LähiEsTauluX symbolien esiintymäkohdista X-vektorissa[179].
Muodosta syöttöaakkoston symboleille LähiEsTauluY symbolien esiintymäkohdista Y-vektorissa.
löytyi := epätosi;  /* Vielä ei ole löytynyt yhtään täsmäystä, joten lasketaan sitä merkitsevä lippu. */
valmis := epätosi;  /* Yhtään ei-triviaalia Dk-luokkaa ei ole käsitelty vielä loppuun. */
rivilinkki[0] := nil; /* Alustetaan luokan 0 rivi- … */
sarakelinkki[0] := nil;  /* … ja sarakelinkit tyhjillä osoittimilla. */
rivikynnys[0] := 0;  /* Alustetaan riveittäisen kynnysarvovektorin nollas positio nollaksi. */
sarakekynnys[0] := 0;  /* Alustetaan sarakekohtaisen kynnysarvovektorin nollas positio nollaksi. */
FOR k := 1, 2, … → m
        rivikynnys[k] := n  + 1;  /* Alustetaan rivikohtainen kynnysarvovektori alkuarvolla n + 1. */
        sarakekynnys[k] := m + 1;  /* Alustetaan sarakekoht. kynnysarvovektori alkuarvolla m + 1. */
 seur := 1;  /*  Muuttuja seur ilmoittaa seuraavaksi loppuun käsiteltävän korkeuskäyrän numeron. */
S1: FOR i := 1, 2, … → m  /* Käsitellään matriisin M kaikki rivit. */
    IF (sarakekynnys[seur] = i)  /* Onko riviltä i jo löydetty luokkaan Dseur kuuluva täsmäys? */
        j := LähiEsTauluY[xi, rivikynnys[seur] + 1] /*  Kyllä, aletaan etsiä luokan Dseur+1 täsmäyksiä. */
        valmis := tosi;   /*  Luokka Dseur saatiin valmiiksi. */
        seur := seur + 1  /* Seuraavaksi loppuun käsiteltävä korkeuskäyrän numero päivittyy. */
    ELSE  /* Luokan Dseur täsmäyksiä voi vielä löytyä. */
        j := LähiEsTauluY[xi, i];  /* Etsitään xi:n 1. täsmäys Y:ssä paikasta i lähtien. */
    k := seur;  /*  Verrataan arvoa alimman aktiivisen korkeuskäyrän nykysijaintiin. */
    S2: WHILE (j ≠ n + 1)  /* Tutkitaan kaikki xi:n esiintymät Y:ssä. */
        IF (j < rivikynnys[k])  /* Onko kohdattu uusi luokan k dominantti täsmäys? */
            tilap := rivikynnys[k];  /* Kyllä. Otetaan rivikynnys[k]:n vanha arvo talteen. */
            rivikynnys[k] := j;  /* Asetetaan uusi arvo. */
            LuoUusiSolmu(uusisolmu);  /* Perustetaan uusi täsmäyssolmu. */
            uusisolmu^.isä := rivilinkki[k – 1];  /* Asetetaan isälinkki edeltäjään. */
            uusisolmu^.ind := i;   /* Talletetaan sen rivin numero, jolta täsmäys löytyi. */
            j := LähiEsTauluY[xi, tilap + 1]  /*  Haetaan xi:n seuraava esiintymä Y:stä. */
        ELSE  /* Löydetty täsmäys ei kuulunut luokkaan Dk. */
```

[179] Lähiesiintymätaulukoiden kokojen on oltava $\sigma^*(m+2)$ ja $\sigma^*(n+2)$ artikkelissa esitettyjen mittojen $\sigma^*(m+1)$ ja $\sigma^*(n+1)$ asemesta. Muussa tapauksessa ohjelman suoritus keskeytyy taulukoiden ylivuotoon rivin (sarakkeen) viimeisen täsmäyksen löydyttyä suoritettaessa lausetta *j* := *LähiEsTaulu*[x_i, *tilap*+1] tai *rivi* := *LähiEsTauluX*[y_i, *tilap*+1]).

```
        IF (j = rivikynnys[k])  /* Oliko vanha rivikynnys[k] sama kuin täsmäyksen Y-indeksi? */
            j := LähiEsTauluY[x_i, j + 1]  /* Kyllä. Etsitään x_i:n seuraava esiintymä Y:stä. */
    IF (löytyi = tosi)  /* Onko riviltä i jo aiemmin löydetty dominantteja täsmäyksiä? */
        rivilinkki[k – 1] := päivitä;  /* On. Lisätään edellinen täsmäys luokan k-1 listaan.*/
        löytyi := epätosi;  /* Lasketaan kirjaamattomien täsmäysten lippu. */
    IF (uusisolmu <>  nil)  /*  Löytyikö silmukan nykykierroksella D_k-täsmäystä? */
        päivitä := uusisolmu;  /* Kyllä. Asetetaan solmu odottamaan päivitystä. */
        löytyi := tosi;  /* Lippu viivytetystä päivityksestä nostetaan. */
        uusisolmu := nil;  /*  Asetetaan osoitin uusisolmu tyhjäksi. */
    k := k+1;  /*  Siirrytään tutkimaan seuraavaa korkeuskäyrää. */
ENDWHILE (S2)
IF (löytyi = tosi)  /* Löytyikö riviltä i yhtään uutta dominanttia täsmäystä? */
    rivilinkki[k – 1] := päivitä;  /* Kyllä. Päivitetään tässä niistä viimeinen. */
    löytyi := epätosi;  /* Kumotaan tieto päivittämättömistä täsmäyksistä. */
IF (valmis = tosi)  /* Onko jonkin luokan kaikki täsmäykset löydetty? */
    rivilinkki[seur – 1] := sarakelinkki[seur – 1];  /* On. Päivitetään edeltäjätieto riville. */
    rivikynnys[seur – 1] := sarakekynnys[seur – 1];
        /* Kopioidaan Rivikynnys[seur - 1]:lle arvo vektorista Sarakekynnys. */
    valmis := epätosi;  /* Ei enää luokkia, joiden kaikki täsmäykset on löydetty. */
IF (rivikynnys[seur] = i)  /* Onko sarakkeesta j jo löydetty luokan D_seur edustaja? */
    rivi := LähiEsTauluX[y_i, sarakekynnys[seur] + 1];
        /*  Kyllä, aletaan etsiä luokan D_seur+1 täsmäyksiä. */
    valmis := tosi;  /*  Luokka D_seur saatiin valmiiksi. */
    seur := seur + 1;  /* Seuraavaksi loppuun käsiteltävän korkeuskäyrän numero päivittyy. */
ELSE  /* Luokan D_seur täsmäyksiä voi vielä löytyä. */
    rivi := LähiEsTauluX[y_i, i + 1];  /* Etsitään y_i:n 1. täsmäys X:ssä paikasta i + 1 lähtien. */
k := seur;  /*  Verrataan arvoa alimman aktiivisen korkeuskäyrän nykysijaintiin. */
S3: WHILE (rivi ≠ m + 1)  /* Tutkitaan kaikki y_i:n esiintymät X:ssä. */
    IF (rivi < sarakekynnys[k])  /* Onko kohdattu uusi luokan k dominantti täsmäys? */
        tilap := sarakekynnys[k];  /* Kyllä. Otetaan sarakekynnys[k]:n vanha arvo talteen. */
        sarakekynnys[k] := rivi;  /* Asetetaan kynnysarvovektoriin uusi arvo. */
        LuoUusiSolmu(uusisolmu);  /* Perustetaan uusi täsmäyssolmu. */
        uusisolmu^.isä := sarakelinkki[k – 1];  /* Asetetaan isälinkki edeltäjään. */
        uusisolmu^.ind := rivi;  /* Talletetaan sen rivin numero, jolta täsmäys löytyi. */
        rivi := LähiEsTauluX[y_i, tilap + 1]  /*  Haetaan y_i:n seuraava esiintymä Y:ssä. */
    ELSE  /* Löydetty täsmäys ei kuulunut luokkaan D_k. */
        IF (rivi = sarakekynnys[k]) /*Oliko sarakekynnys[k] sama kuin täsmäyksen X-indeksi? */
            rivi := LähiEsTauluX[y_i, rivi + 1];  /* Oli. Etsitään y_i:n seuraava esiintymä. */
    IF (löytyi = tosi)  /* Onko sarakkeesta aiemmin löydetty dominanttitäsmäyksiä? */
        sarakelinkki[k – 1] := päivitä; /* On. Lisätään edellinen luokan k-1 listaan.*/
        löytyi = epätosi;  /* Lasketaan kirjaamattomien täsmäysten lippu. */
    IF (uusisolmu ≠  nil)  /*  Löytyikö silmukan nykykierroksella D_k-täsmäystä? */
        päivitä := uusisolmu;  /* Kyllä. Asetetaan solmu odottamaan päivitystä. */
        löytyi := tosi;  /* Lippu viivytetystä päivityksestä nostetaan. */
        uusisolmu := nil;  /* Asetetaan  osoitin uusisolmu tyhjäksi. */
    k := k + 1;  /* Siirrytään tutkimaan seuraavaa korkeuskäyrää. */
ENDWHILE (S3)
IF (löytyi = tosi)  /* Löytyikö sarakkeesta i yhtään uutta dominanttia täsmäystä? */
    sarakelinkki[k – 1] := päivitä;  /* Kyllä. Päivitetään tässä niistä viimeinen. */
    löytyi := epätosi;  /* Kumotaan tieto päivittämättömistä täsmäyksistä. */
```

```
    IF (valmis = tosi)  /* Onko jonkin luokan täsmäyksistä kaikki löydetty? */
        sarakelinkki[seur – 1] := rivilinkki[seur – 1];  /* On. Päivitetään edeltäjätieto sarakkeelle. */
        sarakekynnys[seur – 1] := rivikynnys[seur – 1];
            /*  Kopioidaan sarakekynnys[seur – 1]:lle arvo vektorista rivikynnys. */
        valmis := epätosi;  /* Ei enää luokkia, joiden kaikki täsmäykset on löydetty. */
ENDFOR (S1)
/* Ratkaistaan PYA-jono ja sen pituus. */
IF (rivikynnys[1] ≠ n + 1)
    /* Oliko koko probleemassa yhtään peittävää täsmäystä? */
    k := m;  /* Kyllä. Etsitään, monenko luokan täsmäyksiä mahtoi löytyä. Asia selviää … */
    WHILE (rivikynnys[k] = n + 1)  /*  … silmukassa, jonka lopetettua … */
        k := k – 1;  /*  … k ilmaisee riveittäin löytyneiden täsmäysluokkien määrän. */
    p := k;  /* Riveittäin päästiin kauemmas. Asetetaan PYAlle pituus. */
    uusisolmu := rivilinkki[p]  /* Siirrytään rivilinkki[p]:n listan loppuun. */
    /* Lähdetään keräämään PYA-jonoa p. merkistä alkuun päin. */
    FOR k := p, p–1, … → 1
        PYA[k] := uusisolmu^.ind;   /* uusisolmu^.ind ilmaisee täsmäyksen rivinumeron. */
        uusisolmu := uusisolmu^.isä  /* Siirrytään isälinkkiä pitkin edeltäjäsolmuun. */
ELSE
    p := 0;  /* Ei yhtään täsmäystä koko ongelmassa: PYA on nollan mittainen. */
Tulosta PYA-jono vektorista PYA ja sen pituus p.
END (ALGORITMI RI1).
```

11.4.2 Goemanin ja Clausenin algoritmi (GCL)

```
PROSEDUURI RatkaisePYA(X, Xalku, Xloppu, m, Y, Yalku, Yloppu, n, σ, LähiEsTauluY,
                       LähiEsTauluX, LähiEsTauluY*, LähiEsTauluX*, S, p):
VaihtoTehty = 0;
E1: IF (m > n)
    Vaihda syötevektorit X ja Y keskenään
    Vaihda muuttujat m ja n, Xalku ja Yalku sekä Xloppu ja Yloppu keskenään
    Vaihda taulukot LähiEsTauluX ja LähiEsTauluX* sekä LähiEsTauluY ja LähiEsTauluY* päittäin. */
    VaihtoTehty = 1; /* Muistetaan, että vaihdot on tehty. */
ENDIF (E1)
Rivikynnys[0] := Yalku – 1; /* Alustetaan Rivikynnys[0] tarkastelualueen vasemman reunan ulkopuolelle. */
Sarakekynnys[0] := Xalku – 1; /* Alustetaan Sarakekynnys[0] tarkastelualueen yläreunan ulkopuolelle. */
Rivikynnys*[0] := Yloppu + 1; /* Alustetaan Rivikynnys*[0] tarkastelualueen oikean reunan ulkopuolelle. */
Sarakekynnys*[0] := Xloppu + 1; /* Alustetaan Sarakekynnys*[0] tarkastelualueen alareunan ulkopuolelle. */
c^Y.rivi = c^V.rivi := Xalku – 1; /* Alustetaan pisteiden c^Y ja c^V rivikoordinaatit yläreunan ... */
c^Y.sarake = c^V.sarake := Yalku – 1;  /* … ja sarakekoordinaatit vasemman reunan ulkopuolelle. */
c^Λ.rivi = c^O.rivi := Xloppu + 1; /* Alustetaan pisteiden c^Λ ja c^O rivikoordinaatit alareunan … */
c^Λ.sarake = c^O.sarake := Yloppu + 1;  /* … ja sarakekoordinaatit oikean reunan ulkopuolelle. */
S1: FOR u := 1, 2, … →⌈m/2⌉  /* Alustetaan menosuunnan kynnysarvovektorit: … */
    Rivikynnys[u] := Yloppu + 1;  /* … rivien käsittelyyn vasemmalta oikealle ja … */
    Sarakekynnys[u] := Xloppu + 1;  /* … sarakkeiden käsittelyyn ylhäältä alas vas. yläkulmasta lähtien. */
ENDFOR (S1)
S2: FOR u := 1, 2, … →⌊m/2⌋  /* Alustetaan paluusuunnan kynnysarvovektorit: … */
```

```
    Rivikynnys*[u] := Yalku – 1;  /* … rivien käsittelyyn oikealta vasemmalle ja … */
    Sarakekynnys[u] := Xalku – 1;  /*  … sarakkeiden käsittelyyn alhaalta ylös oikeasta alakulmasta lähtien. */
ENDFOR (S2)
ylinrivi := Xalku;  /* matriisin ylin toistaiseksi tutkimaton rivi */
vasensarake:= Yalku; /* matriisin vasemmanpuoleisin toistaiseksi tutkimaton sarake */
alinrivi := Xloppu;   /* matriisin alin toistaiseksi tutkimaton rivi */
oikeasarake := Yloppu; /* matriisin oikeanpuoleisin toistaiseksi tutkimaton sarake */
T_YV := 1;  /* alin täsmäysluokka, johon vielä etsitään uusia edustajia vasemmasta yläkulmasta lähtien */
D_Y := 0;  /* korkein matriisin yläositteesta löytynyt täsmäysluokka */
D_V := 0;  /* korkein matriisin vasemmasta ositteesta löytynyt täsmäysluokka */
T_AO := 1; /* alin täsmäysluokka, johon vielä etsitään uusia edustajia oikeasta alakulmasta lähtien */
D_A := 0; /* korkein matriisin alaositteesta löytynyt täsmäysluokka */
D_O := 0; /* korkein matriisin oikeasta ositteesta löytynyt täsmäysluokka */
E2: IF (m on pariton) /* Testataan syötejonon X pituuden pariteettia. */
    siirry algoritmissa kohtaan YV; /* Hypätään keskelle pääsilmukkaa S1: aloitus vasemmasta yläkulm.[180] */
ENDIF (E2)
S3: WHILE (ylinrivi ≤ alinrivi) AND (vasensarake ≤ oikeasarake) /* Algoritmin pääsilmukka alkaa tästä. */
    k := LähiEsTauluY*[x_alinrivi, oikeasarake]; /* Haetaan alueen alimman rivin oikeanpuoleisin täsmäys. */
    u := T_AO;  /* Lähdetään liikkeelle alimmasta ei-täyttyneestä täsmäysluokasta. */
    S4: WHILE u ≤ D_A /* Käydään tarvittaessa läpi kaikki alaositteen täsmäysluokat. */
        j := Rivikynnys*[u];  /* Tutkitaan, mistä sarakkeesta löytyy luokan u oikeanpuoleisin täsmäys. */
        E3: IF (k ≥ j) /* Löytyikö nykyrivin täsmäys sarakkeesta j tai sen oikealta puolelta? */
            Rivikynnys*[u] := j; /* Löytyi: päivitetään luokan u kynnysarvo ajan tasalle. */
            k := LähiEsTauluY*[x_alinrivi, j – 1]; /* Etsitään rivin seuraava täsmäys sarakkeen j vas. puolelta. */
        ENDIF (E3)
        u := u + 1;  /* Siirrytään tarkastelemaan seuraavaa täsmäysluokkaa. */
    ENDWHILE (S4)
    E4: IF (k ≥ vasensarake) /* Löytyikö halualueelta täsmäys, joka ei kelvannut luokkiin T_AO..D_A? */
        D_A := u; /* Kyllä: alaositteen PYA kasvoi yhdellä. */
        Rivikynnys*[D_A] := k; /* Päivitetään paluusuunnan kynnysarvovektorin luokan D_A tiedot. */
        E5: IF (Sarakekynnys[D_V] ≥ alinrivi) /* Vasemman ositteen ylimmän luokan täsmäys ei ylempänä? */
            D_V := D_V – 1;  /* Ei ole: kumotaan luokka D_V vasemmasta ositteesta ja siirretään se D_A:han. */
        ELSE  /* On: ainakin yksi luokan D_V täsmäys sijaitsee nykyistä riviä ylempänä. */
            c^A.rivi := alinrivi; /* Otetaan talteen nykyriviltä löytyneen D_A-täsmäyksen rivinumero, … */
            c^A.sarake := k;   /* … sitten vielä sen sarakenumero ... */
            E6: IF (D_V > 0)  /* Onko vasemman neljänneksen PYAn pituus > 0? */
                c^V.rivi := Sarakekynnys[D_V]; /* On: otetaan talteen D_V:n ylimmän edustajan sijaintirivi  … */
                c^V.sarake := LähiEsTauluY*[x_sarakekynnys[DV], vasensarake-1];  /* … ja sitten sarakenumero ... . */
            ENDIF (E6)
            vasenpos^AV := D_A;   /* Lopuksi muistetaan luokan D_A numero. */
        ENDIF (E5)
    ENDIF (E4)
    k := LähiEsTauluX*[y_oikeasarake, alinrivi – 1];  /* Haetaan alueen oikeanpuoleisimman rivin alin täsmäys. */
    u := T_AO; /* Lähdetään liikkeelle alimmasta ei-täyttyneestä täsmäysluokasta matriisin oik. ositteessa. */
    S5: WHILE u ≤ D_O /* Käydään tarvittaessa läpi kaikki oikean ositteen täsmäysluokat. */
        j := Sarakekynnys*[u]; /* Tutkitaan, miltä riviltä löytyy luokan u alin täsmäys. */
        E7: IF (k ≥ j) /* Löytyikö nykysarakkeen täsmäys riviltä j tai sen alapuolelta? */
```

[180] kts. alaviite 87

```
        Sarakekynnys*[u] := k; /* Löytyi: päivitetään luokan u kynnysarvo ajan tasalle. */
        k := LähiEsTauluX*[y_oikeasarake, j – 1]; /* Etsitään sarakkeen seuraava täsmäys rivin j yläpuolelta. */
    ENDIF (E7)
    u := u + 1; /* Siirrytään tarkastelemaan seuraavaa täsmäysluokkaa. */
ENDWHILE (S5)
E8: IF (k ≥ ylinrivi) /* Löytyikö halualueelta täsmäys, joka ei kelvannut luokkiin T_AO.. D_O? */
    D_O := u; /* Kyllä: oikean ositteen PYA kasvoi yhdellä. */
    SarakekynnysX*[D_O] := k; /* Päivitetään paluusuunnan kynnysarvovektorin luokan D_A tiedot. */
    E9: IF (Rivikynnys[D_Y] ≥ oikeasarake) /* Yläositteen ylimmän luokan täsmäys ei vasempana? */
        D_Y := D_Y – 1;  /* Ei ole: kumotaan luokka D_Y yläositteesta ja siirretään se D_O:hon. */
    ELSE /* On: ainakin yksi luokan D_Y täsmäys sijaitsee nykyisestä sarakkeesta vasemmalle. */
        c^O.rivi := k; /* Otetaan talteen nykyriviltä löytyneen luokan D_O täsmäyksen rivinumero, … */
        c^O.sarake := oikeasarake; /* … sitten vielä sen sarakenumero ... */
        E10: IF (D_Y > 0) /* Onko yläneljänneksen PYAn pituus > 0? */
            c^Y.rivi := LähiEsTauluX*[y_Rivikynnys[DY], ylinrivi-1]; /* On: D^Y:n alimman edustajan rivi  … */
            c^Y.sarake := Rivikynnys[D_Y];   /* … ja sen sarakenumero otetaan talteen. */
        ENDIF (E10)
        vasenpos^YO := D_O; /* Lopuksi muistetaan luokan D_O numero. */
    ENDIF (E9)
ENDIF (E8)

/* Testataan, voidaanko ala- ja oikean neljänneksen korkeuskäyrät terminoida. */

E11: IF (Rivikynnys*[T_AO] = oikeasarake) /* Löytyikö luokan T_AO täsmäys mahdollisimm. oikealta? */
    Sarakekynnys*[T_AO] := LähiEsTauluX[y_oikeasarake, alinrivi]; /* Kyllä: päivitetään Sarakekynnys.  */
    E12: IF (T_AO > D_O)  /* Entä onko alaositteen PYA pidempi kuin oikean ositteen? */
        D_O := T_AO;  /* On. Päivitetään D_O ajan tasalle: samaksi kuin T_AO. */
        E13: IF (Rivikynnys[D_Y] ≥ oikeasarake) /* On: entä ovatko D_Y:n edustajat oik. reunassa? */
            D_Y := D_Y – 1 /* Kyllä ovat: kumotaan luokka D_Y yläositteesta ja siirretään se D_A:han. */
        ELSE /* Eivät: ainakin yksi D_Y-täsmäys sijaitsee hakualueen reunasta vasemmalle. */
            c^O.rivi := LähiEsTauluX[y_oikeasarake, alinrivi]; /* Tallennetaan D_A-täsmäyksen rivi, … */
            c^O.sarake := oikeasarake; /* … sitten vielä sen sarakenumero pisteeseen c^O. */
            E14: IF (D_Y > 0) /* Onko yläneljänneksen PYAn pituus > 0? */
                c^Y.rivi := LähiEsTauluX*[y_Rivikynnys[DY], ylinrivi-1]; /* On: Otetaan talteen rivi … */
                c^Y.sarake := Rivikynnys[D_Y];   /* … ja sarakenumero D_Y:n alimmalta edustajalta. */
            ENDIF (E14)
            vasenpos^YO := D_Y; /* Lopuksi asetetaan yläosan rajatäsmäyksen luokan numero. */
        ENDIF (E13)
    ENDIF (E12)
    T_AO := T_AO + 1  /* Päivitetään alimman tarkasteltavan täsmäysluokan numero alaosassa. */
ELSE
    E15: IF (Sarakekynnys*[T_AO] = alinrivi) /* Ei: entä mahdollisimman alhaalta? */
        Rivikynnys*[Tao] = LähiEsTauluY[x_alinrivi, oikeasarake+1]; /* Kyllä: päivitetään Rivikynnys. */
        E16: IF (T_AO > D_A) /* Entä onko oikean ositteen PYA pidempi kuin alaositteen? */
            D_A := T_AO;  /* On. Päivitetään D_A ajan tasalle: samaksi kuin T_AO. */
            E17: IF (Sarakekynnys[D_V] ≥ alinrivi) /* On: ovatko D_V:n edust. vasemm. alareun.? */
                D_V := D_V–1 /* Kyllä: kumotaan D_V vasemm. ositteesta ja siirretään se D_O:hon. */
            ELSE /* Ei: ainakin yksi D_V-täsmäys sijaitsee hakualueen reunan yläpuolella. */
                c^A.rivi := alinrivi; /* Tallennetaan D_A-täsmäyksen rivi, … */
                c^A.sarake := LähiEsTauluY[x_alinrivi, oikeasarake + 1]; /* … sitten vielä sen sarake. */
```

E18: IF ($D_V > 0$) /* Onko vasemman neljänneksen PYAn pituus > 0? */
c^V.rivi := Sarakekynnys[D_V]; /* Muistetaan D_V:n ylimm. edust. sijaintir. … */
c^V.sarake := LähiEsTauluY*[$x_{Sarakekynnys[D_V]}$, vasensarake–1]; /* … ja sar. */
ENDIF (E18)
$vasenpos^{AV}$:= D_O; /* Asetetaan lopuksi luokan D_O rajatäsmäyksen numero. */
ENDIF (E17)
ENDIF (E16)
T_{AO} := T_{AO} + 1 /* Päivitetään alimman tarkasteltavan täsmäysluokan numero alaosassa. */
ENDIF (E15)
ENDIF (E11)
alinrivi := alinrivi – 1;[181] /* Nostetaan hakualueen alareunaa yhdellä rivillä ylöspäin ... */
oikeasarake := oikeasarake – 1;[182] /* … ja siirretään sen oikeaa reunaa yhdellä sar. vasemmalle. */

YV:

E19: IF (vasensarake ≤ oikeasarake) /* Onko vielä tutkimattomia sarakkeita jäljellä? */
k := LähiEsTauluY[$x_{ylinrivi}$, vasensarake]; /* On: haetaan alimman rivin oikeanpuoleisin täsmäys. */
u := T_{YV}; /* Lähdetään liikkeelle alimmasta ei-täyttyneestä täsmäysluokasta matriisin ylävasemm. */
S6: WHILE u ≤ D_Y /* Käydään tarvittaessa läpi kaikki yläositteen täsmäysluokat. */
j := Rivikynnys[u]; /* Tutkitaan, mistä sarakkeesta löytyy luokan u vasemmanpuoleisin täsm. */
E20: IF (k ≤ j) /* Löytyikö nykyrivin täsmäys sarakkeesta j tai sen vasemmalta puolelta? */
Rivikynnys[u] := j; /* Löytyi: päivitetään luokan u kynnysarvo ajan tasalle. */
k := LähiEsTauluY[$x_{ylinrivi}$, j + 1]; /* Etsitään rivin seuraava täsm. sarakkeen j oik. puol. */
ENDIF (E20)
u := u + 1; /* Siirrytään tarkastelemaan seuraavaa täsmäysluokkaa. */
ENDWHILE (**S6**)
E21: IF (k ≤ oikeasarake) /* Löytyikö halualueelta täsmäys, joka ei kelvannut luokkiin T_{YV}..D_Y? */
D_Y := u; /* Kyllä: yläositteen PYA kasvoi yhdellä. */
Rivikynnys[D_Y] := k; /* Päivitetään menosuunnan kynnysarvovektorin luokan D_Y tiedot. */
E22: IF (Sarakekynnys*[D_O] ≤ ylinrivi) /* Oikean ositteen ylimm. luokan täsm. ei alempana? */
D_O := D_O – 1; /* Ei ole: kumotaan luokka D_O oikeasta ositteesta ja siirretään se D_Y:hyn. */
ELSE /* On: ainakin yksi luokan D_O täsmäys sijaitsee nykyistä riviä alempana. */
c^Y.rivi := ylinrivi; /* Otetaan talteen nykyriviltä löytyneen D_Y-täsmäyksen rivinumero … */
c^Y.sarake := k; /* … ja sitten vielä sen sarakenumero. */
E23: IF ($D_O > 0$) /* Onko oikean neljänneksen PYAn pituus > 0? */
c^O.rivi := Sarakekynnys*[D_O]; /* On: lisäksi talletetaan D_O-täsmäyksen rivinum., … */
c^O.sarake := LähiEsTauluY[$x_{Sarakekynnys*[DO]}$, oikeasarake+1] /*ja sitten sarakenum. */
ENDIF (E23)
$vasenpos^{YO}$:= D_Y; /* … ja lopuksi sen luokan numero. */
ENDIF (E22)
ENDIF (E21)
k := LähiEsTauluX[$y_{vasensarake}$, ylinrivi + 1];[183] /* Haetaan alueen vasemmanpuol. rivin ylin täsm. */
u := T_{YV}; /* Lähdetään liikkeelle alimmasta ei-täyttyneestä täsmäysluokasta matriisin ylävas. */

---

[181] Alkuperäisartikkelissa esiintyy tässä kohtaa asetuslause ”*ylinrivi* := *ylinrivi* + 1”. Jos tuo asetus suoritettaisiin, vaikuttaisi, ettei syötejonon *X* ensimmäistä merkkiä tutkittaisi välttämättä kertaakaan, jos jonon pituus *m* on parillinen.

[182] Alkuperäisartikkelissa esiintyy tässä kohtaa asetuslause ”*vasensarake* := *vasensarake* + 1”. Jos kyseinen asetus suoritettaisiin, vaikuttaisi, ettei syötejonon *Y* ensimmäistä merkkiä tutkittaisi välttämättä kertaakaan silloin, kun syötejonon *X* pituus *m* on parillinen.

[183] Alkuperäisartikkelissa tällä rivillä esiintyy asetuslause ”*k* := *LähiEsTauluX*[$y_{vasensarake}$, *ylinrivi*]:”. Se on kuitenkin ristiriitainen artikkelissa esitetyn algoritmin kuvauksen kanssa, jonka mukaan mitään matriisin solua ei ole tarpeen tutkia kahdesti, ja paikka *X*[*ylinrini*, $y_{vasensarake}$] on jo ehditty kertaalleen tutkia.

S7: WHILE u ≤ D_V /* Käydään tarvittaessa läpi kaikki vas. ositteen täsmäysluokat. */
j := Sarakekynnys[u]; /* Tutkitaan, miltä riviltä löytyy luokan u ylin täsmäys. */
E24: IF (k ≤ j) /* Löytyikö nykysarakkeen täsmäys riviltä j tai sen yläpuolelta? */
Sarakekynnys[u] := k; /* Löytyi: päivitetään luokan u kynnysarvo ajan tasalle. */
k := LähiEsTauluX[$y_{vasensarake}$, j + 1];/* Etsitään sarakkeen seuraava täsmäys rivin j alap. */
ENDIF (E24)
u := u + 1; /* Siirrytään tarkastelemaan seuraavaa täsmäysluokkaa. */
ENDWHILE (**S7**)
E25: IF (k ≤ alinrivi) /* Löytyikö halualueelta täsmäys, joka ei kelvannut luokkiin T_{YV}.. D_V? */
D_V := u; /* Kyllä: vasemman ositteen PYA kasvoi yhdellä. */
Rivikynnys[D_V] := k; /* Päivitetään menosuunnan kynnysarvovektorin luokan D_V tiedot. */
E26: IF Rivikynnys[D_A] ≤ vasensarake /* Alaositteen yl. luokan täsmäys ei oikeampana? */
D_A := D_A – 1; /* Ei ole: kumotaan luokka D_A alaositteesta ja siirretään se D_V:hen. */
ELSE /* Ei: ainakin yksi D_A-täsmäys sijaitsee hakualueen reunasta oikealle. */
c^V.rivi := k; /* Otetaan talteen nykyriviltä löytyneen luokan D_V täsmäyksen rivinro … */
c^V.sarake := vasensarake; /* … ja sitten vielä sen sarakenumero. */
E27: IF (D_A > 0) /* Onko alaneljänneksen PYAn pituus > 0? */
c^A.rivi := LähiEsTauluY[$y_{Rivikynnys*[DA]}$, alinrivi + 1]; /* Muistetaan rivinumero ja … */
c^A.sarake := Rivikynnys*[D_A] /* … sarake D_A:n ylimmältä edustajalta. */
ENDIF (E27)
vasenpos^{AV} := D_V; /* … ja lopuksi sen luokan numero. */
ENDIF (E26)
ENDIF (E25)

/* Testataan, voidaanko ylä- ja vasemman neljänneksen korkeuskäyrät terminoida. */

E28: IF (Rivikynnys[T_{YV}] = vasensarake) /* Löytyikö luokan T_{YV} täsm. mahdoll. vasemmalta? */
Sarakekynnys[T_{YV}] := LähiEsTauluX*[$y_{vasensarake}$, ylinrivi]; /* Kyllä: päivitetään Sarakekynnys. */
E29: IF (T_{YV} > D_V) /* Entä onko yläositteen PYA pidempi kuin vasemman ositteen? */
D_V = $T_{YV;}$ /* On. Päivitetään D_V ajan tasalle: samaksi kuin T_{YV}. */
E30: IF (Rivikynnys*[D_A] ≤ vasensarake) /* On: ovatko D_A:n edustajat vas. reunassa? */
D_A := D_A – 1 /* Kyllä: kumotaan D_A alaositteesta ja siirretään se D_Y:hyn. */
ELSE /* Ei: ainakin yksi D_A-täsmäys sijaitsee hakualueen vasemmasta reunasta oikealle. */
c^V.rivi := LähiEsTauluY*[$y_{vasensarake}$, ylinrivi]; /* Tallennetaan D_A-täsmäyksen rivi, … */
c^V.sarake := vasensarake; /* … sitten vielä sen sarakenumero ... */
E31: IF (D_A > 0)
c^A.rivi := LähiEsTauluX[$y_{Rivikynnys[DA]}$, alinrivi + 1]; /* Kirjataan rivinumero ja … */
c^A.sarake := Rivikynnys[D^T]; /* … sitten vielä sarake D_A:n ylimmältä edustajalta. */
ENDIF (E31)
vasenpos^{AV} := D_V; /* Asetetaan lopuksi uusi loppupositio vas./alaneljänn. PYAlle */
ENDIF (E30)
ENDIF (E29)
T_{YV} := T_{YV} + 1; /* Päivitetään alimman tarkasteltavan täsmäysluokan numero alaosassa. */
ELSE
E32: IF (Sarakekynnys[T_{YV}] = ylinrivi) /* Ei: entä mahdollisimman ylhäältä? */
Rivikynnys[T_{YV}] = LähiEsTauluX*[$x_{ylinrivi}$, vasensarake – 1] /* Kyllä: päivit. Rivikynnys. */
E33: IF (T_{YV} > D_Y) /* Entä onko vasemman ositteen PYA pidempi kuin yläositteen? */
D_Y = T_{YV}; /* On. Päivitetään D_Y ajan tasalle: samaksi kuin T_{YV}. */
E34: IF (Sarakekynnys*[D_O] ≤ ylinrivi) /* Ovatko D_O:n edustajat oikeassa ylär.? */

D_O := D_O – 1 /* Kyllä: kumotaan D_O oikeasta ositteesta ja siirretään se D_V:hen. */
ELSE /* Ei: ainakin yksi D_O-täsmäys sijaitsee hakualueen yläreunan alapuolella. */
c^Y.rivi := ylinrivi; /* Tallennetaan D_Y-täsmäyksen rivi … */
c^Y.sarake :=LähiEsTauluY*[$x_{ylinrivi}$, vasensarake – 1]; /* … ja sitten sarakenro. */
E35: IF (D_O > 0) /* Onko alaneljänneksen PYAn pituus > 0? */
c^O.rivi := Sarakekynnys*[D_O]; /* On: lisäksi tallet. D_O-täsmäyksen rivi … */
c^O.sarake := LähiEsTauluY[$x_{Sarakekynnys*[D_O]}$, oikeasarake + 1] /* ja sarake. */
ENDIF (E35)
$vasenpos^{YO}$:= D_Y; /* Asetetaan uusi loppupositio ylä-/oik. neljänn. PYAlle */
ENDIF (E34)
ENDIF (E33)
T_{YV} := T_{YV} + 1 /* Päivitetään alimman tarkasteltavan täsmäysluokan numero alaosassa. */
ENDIF (E32)
ENDIF (E28)
ylinrivi := ylinrivi + 1;[184] /* Lasketaan hakualueen yläreunaa yhdellä rivillä alaspäin ... */
vasensarake := vasensarake + 1;[185] /* … ja siirretään sen vasenta reunaa yhdellä sarakkeella oik. */
ENDWHILE (**S3**)

/* Määrätään PYAn pituus ja rekursiivisesti tutkittavien alueiden rajat. Rajatäsmäykset viedään jonoon S. */

pyapituusylävasen := 0; /* Alustetaan ylävasemman alueen PYAn pituudeksi nolla … */
pyapituusalaoikea := 0; /* … ja tehdään samoin alaoikean alueen PYAn pituudelle. */
E36: IF ((D_Y = 0) **AND** (D_V = 0) **AND** (D_Δ = 0) **AND** (D_O = 0) /* Alueella ei yhtään täsmäystä? */
tapaus = 0; /* Annetaan tapaukselle numeroksi 0: aluetta ei kannata tutkia lisää. */
rekursio1rivi := Xalku – 1; /* Rajataan ylävasen alue tarkastelujen ulkopuolelle … */
rekursio1sarake := Yalku – 1; /* … näillä kahdella lauseella. */
rekursio2rivi := Xloppu + 1; /* Rajataan alaoikea alue tarkastelujen ulkopuolelle … */
rekursio2sarake := Yloppu + 1; / * … näillä kahdella lauseella. */
ELSE IF ((D_Y > D_V) **AND** (D_Δ > D_O)) /* Onko yläreunan PYA pidempi kuin vasemman ja alareunan PYA pidempi kuin oikean reunan? */
tapaus := 1; /* On: numeroidaan tapaus numerolla 1. */
E37: IF (T_{YV} ≤ D_V) /* Onko ensimmäisen yläv. ei-terminoidun luokan numero ≤ vas. ositteen PYA? */
T_{YV} := D_V + 1; /* On: asetetaan uudeksi alimmaksi M:n yläpuoliskon täsmäysluokaksi D_V + 1. */
ENDIF (E37)
E38: IF ($T_{\Delta O}$ ≤ D_O) /* Onko ensimmäisen alaoik. ei-termin. luokan numero ≤ oik. ositteen PYA? */
$T_{\Delta O}$:= D_O + 1; /* On: asetetaan uudeksi alimmaksi M:n alapuoliskon täsmäysluokaksi D_O + 1. */
ENDIF (E38)
u := D_Y; /* Alustetaan u:n alkuarvoksi yläositteen PYAn pituus. */
v := $T_{\Delta O}$; /* Alustetaan v:n arvoksi ensimmäinen täsmäysluokka, jolle ei ole edustajaa oikealla. */
vertailukelpoisia := 0; /* Vertailukelpoisia ylä- ja alaositteen korkeuskäyriä ei ole vielä löytynyt. */
S8: WHILE ((u ≥ T_{YV}) **AND** (v ≤ D_Δ)) /* Käydään läpi ei-terminoidut korkeuskäyrät. */
E39: IF (Rivikynnys[u] ≥ Rivikynnys*[v]) /* Voidaanko vertailtavat korkeuskäyrät yhdistää toisiinsa? */
u := u – 1; /* Kyllä voidaan – PYAn pituus ei muutu, peruutetaan yläosassa edell. luokkaan. */

---

[184] Alkuperäisartikkelissa esiintyy tässä kohtaa asetuslause ”*alinrivi* := *alinrivi* – 1”. Jos tuo asetus suoritettaisiin, ei syötejonon *Y* viimeistä merkkiä tutkittaisi koskaan, jos jonon *X* pituus *m* on pariton.

[185] Alkuperäisartikkelissa esiintyy tässä kohtaa asetuslause ”*oikeasarake* := *oikeasarake* – 1”. Jos kyseinen asetuslause suoritettaisiin, vaikuttaisi siltä, ettei syötejonon *Y* viimeistä merkkiä tutkittaisi välttämättä kertaakaan silloin, kun syötejonon *X* pituus *m* on pariton.

ELSE /* Ei voida: perustetaan uusi korkeuskäyrä. */
vertailukelpoisia := 1; /* Nostetaan lippu merkiksi, että vertailukelpoinen käyräpari löytyi. */
pyapituusylävasen := u – 1; /* Otetaan u – 1 talteen ennen siirtymistä alaosassa käyrälle v + 1. */
pyapituusalaoikea := v – 1; /* Otetaan myös v - 1:n arvo talteen ennen mainittua siirtymistä. */
vasenpos^{YO} := u; /* Asetetaan uusi loppupositio ylä-/oik. neljänn. PYAlle */
ENDIF (E39)
v := v + 1; /* Siirrytään alapuoliskossa seuraavan täsmäysluokan korkeuskäyrälle. */
ENDWHILE (**S8**)

E40: IF ((vertailukelpoisia = 0) **AND** (u < T_{YV})) /* Terminoimattomat korkeuskäyrät loppuivat yläositteesta, ja mikään ala- ja yläositteen korkeuskäyräpari ei ollut vertailukelpoinen keskenään: p = D_V + D_A. */
rekursio1rivi := c^V.rivi – 1; /* Asetetaan vasemmasta yläkulmasta tutkittavan alueen alin rivi ja … */
rekursio1sarake := c^V.sarake – 1; /* … oikeanpuoleisin sarake. Alue alkaa paikasta (Xalku, Yalku). */
pyapituusylävasen := vasenpos^{AV} – 1; /* Näin monta täsmäystä tiedetään löytyvän alueelta. */
E41: IF (c^V.rivi ≥ Xalku) /* Löytyikö tutkitulta alueelta ainakin yksi kirjattu täsmäys vasemmalta? */
u := 1; /* Kyllä: valmistaudutaan viemään se paikalleen PYA-jonoon rekursion palautuessa. */
ELSE
tapaus := 4; /* Kaikki tutkitulta alueelta löytyneet täsmäykset sijaitsevat alaositteessa. */
ENDIF (E41)
rekursio2rivi := c^A.rivi + 1; /* Asetetaan oikeasta alakulmasta tutkittavan alueen ylin rivi ja … */
rekursio2sarake := c^A.sarake + 1; /* … vasen sarake. Alue päättyy paikkaan (Xloppu, Yloppu). */
pyapituusalaoikea := D_A – 1 – (vasenpos^{AV} – D_V); /* Alaosan PYAn pituus asetetaan tässä. */
E42: IF ((c^A.rivi > Xloppu) **OR** (c^A.rivi < rekursio1rivi + 2) **OR** (cA.sarake < rekursio1sarake + 2))
/* Eikö vasemmassa ositteessa ole yhtään täsmäystä? */
tapaus := 4; /* Ei ole: kaikki osaratkaisun täsmäykset tulevat alaositteesta. */
ENDIF (E42)
ELSE IF (vertailukelpoisia = 0) /* Terminoimattomat käyrät loppuivat ensinnä alaositteesta, ja mikään tutkittu ala- ja yläositteen korkeuskäyräpari ei ollut vertailukelpoinen keskenään: D_Y + D_O. */
rekursio1rivi = c^Y.rivi – 1; /* Asetetaan vasemmasta yläkulmasta tutkittavan alueen alin rivi ja … */
rekursio1sarake = c^Y.sarake – 1; /* … oikeanpuoleisin sarake. Alue alkaa paikasta (Xalku, Yalku). */
E43: IF (c^Y.rivi < Xalku) /* Eikö tutkitulta alueelta löytynyt yhtään kirjattua täsmäystä ylhäältä? */
tapaus = 4; /* Ei. Kaikki tutkitulta alueelta löytyneet täsmäykset sijaitsevat alaositteessa. */
ENDIF (E43)
pyapituusylävasen := vasenposYO – 1;
rekursio2rivi := c^O.rivi + 1;
rekursio2sarake := c^O.sarake + 1;
pyapituusalaoikea := D_O – 1 – (vasenpos^{YO} – D_Y); /* Oikean ositteen PYAn pituus asetetaan tässä. */
E44: IF ((c^O.rivi < rekursio1rivi + 2) **OR** (c^O.sarake < rekursio2sarake + 2) **OR** (c^O.rivi > Xloppu))
/* Eikö alaosassa ole yhtään täsmäystä, joka sijaitsee aidosti oikealla yläositteen viimeisestä? */
tapaus := 5; /* Ei ole: oikean ositteen tutkiminen on jatkossa tuloksetonta. */
ENDIF (E44)
ELSE /* Nyt muuttujan vertailukelpoisia arvo on 1. */
E45: IF (((pyapituusylävasen + pyapituusalaoikea + 2) ≤ (D_V + D_A)) **OR**
((pyapituusylävasen + pyapituusalaoikea + 2) ≤ (D_Y + D_O)))
/* Eikö pelkistä ylä- ja alaositteista muodostettavissa oleva PYA ollut riittävän pitkä? */
E46: IF ((D_V + D_A) ≥ (D_Y + D_O)) /* Ei: kumpi summa on isompi: D_V + D_A vai D_Y + D_O? */
rekursio1rivi := c^V.rivi – 1; /* Ensin mainittu on vähintään yhtäsuuri: asetetaan vasemmasta yläkulmasta tutkittavan alueen alin rivi ja … */
rekursio1sarake := c^V.sarake – 1; /* … oikea sarake. Alue alkaa paikasta (Xalku, Yalku). */
rekursio2rivi := c^A.rivi + 1; /* Asetetaan oikeasta alakulmasta tutkittavan alueen ylin … */

```
            rekursio2sarake := c^A.sarake + 1;  /* … rivi ja vasen sarake. Päättyy (Xloppu, Yloppu). */
            pyapituusylävasen := vasenpos^AV – 1;  /* PYAn pituus ylävasemmassa alueessa. */
            pyapituusalaoikea := D_A – 1 – (vasenpos^AV – D_V); /* Asetetaan PYAn pituus alaoikealla. */
            u := vasenpos^AV  /* Asetetaan alavasemman rajatäsmäyksen indeksi. */
         ELSE /* D_Y + D_O oli summista isompi. */
            rekursio1rivi := c^Y.rivi – 1;  /* Asetetaan vasemmasta yläkulmasta tutkittavan alueen alin */
            rekursio1sarake := c^Y.sarake – 1; /* … rivi ja oikea sarake. Alkukohtana (Xalku, Yalku). */
            rekursio2rivi := c^O.rivi + 1;  /* Asetetaan oikeasta alakulmasta tutkittavan alueen ylin … */
            rekursio2sarake := c^O.sarake + 1;  /* … rivi ja vasen sarake. Päättyy (Xloppu, Yloppu). */
            pyapituusylävasen := D_Y – 1;  /* PYAn pituus ylävasemmassa alueessa. */
            pyapituusalaoikea := D_A – 1 – (vasenpos^YO – D_Y); /* Asetetaan PYAn pituus alaoikealla. */
            u := vasenpos^YO  /* Asetetaan yläoikean rajatäsmäyksen indeksi. */
         ENDIF (E46)
      ELSE  /* Ehtolauseen E45 ehto epätosi. */
         rekursio1rivi := LähiEsTauluX*[Y[LähiEsTauluY[pyapituusylävasen + 1]]][alinrivi] – 1;
         rekursio1sarake := LähiEsTauluY[pyapituusylävasen + 1] – 1;
         rekursio2rivi := LähiEsTauluX[Y[LähiEsTauluY*[pyapituusalaoikea + 1]]][ylinrivi] + 1;
         rekursio2sarake := LähiEsTauluY*[pyapituusalaoikea + 1] + 1;
      ENDIF (E45)
   ENDIF (E40)
ELSE  /* Ei toteutunut E36:n tapaus 0 eikä tapaus 1. */
   E47: IF ((D_V + D_A) ≥ (D_Y + D_O))  /* PYA kertyy vasemman ja alaositteen täsmäyksistä. */
      tapaus = 2;
      E48: IF ((c^V.rivi < Xalku) OR (vasenpos^AV = 0)) /* Vasemmalla ei yhtään kirjattua täsmäystä. */
         rekursio1rivi := Xalku – 1; /* Rajataan ylävasen alue tarkastelujen ulkopuolelle … */
         rekursio1sarake := Yalku – 1;  /* … näillä kahdella lauseella. */
         piste1 = 0; /* Keskimmäistä vasemmanpuoleista täsmäystä ei ole olemassa. */
      ELSE
         rekursio1rivi := c^V.rivi – 1;  /* Asetetaan vasemmasta yläkulmasta tutkittavan alueen alin  … */
         rekursio1sarake := c^V.sarake – 1; /* … rivi ja oikea sarake. Alueen alkukohta (Xalku, Yalku). */
         pyapituusylävasen = vasenpos^AV – 1; /* PYAn pituus ylävasemmassa alueessa. */
         piste1 = 1;  /* Vasemmalta puolelta löytyi keskimmäinen täsmäys. */
      ENDIF (E48)
      E49: IF ((c^A.rivi > Xloppu) OR (vasenpos^AV = D_V + D_A))/* Alhaalla ei yhtään kirjattua täsmäystä. */
         rekursio2rivi := Xloppu + 1;  /* Rajataan alaoikea alue tarkastelujen ulkopuolelle … */
         rekursio2sarake := Yloppu + 1; / * … näillä kahdella lauseella. */
         piste2 = 0; /* Keskimmäistä alaosan täsmäystä ei ole olemassa. */
      ELSE
         rekursio2rivi := c^A.rivi + 1;  /* Asetetaan oikeasta alakulmasta tutkittavan alueen ylin … */
         rekursio2sarake := c^A.sarake + 1;  /* … rivi ja vasen sarake. Alue päättyy (Xloppu, Yloppu). */
         pyapituusalaoikea := D_A – 1 – (vasenpos^AV – D_V); /* Asetetaan PYAn pituus alaoikealla. */
         piste2 = 1; /* Oikealta puolelta löytyi keskimmäinen täsmäys. */
      ENDIF (E49)
   ELSE  /* Ehtolauseessa E47 D_Y + D_O > D_V + D_A */
      tapaus = 3;
      E50: IF ((c^Y.rivi < Xalku) OR (vasenpos^YO = 0))  /* Yläosassa ei yhtään kirjattua täsmäystä. */
         rekursio1rivi := Xalku – 1; /* Rajataan ylävasen alue tarkastelujen ulkopuolelle … */
         rekursio1sarake := Yalku – 1;  /* … näillä kahdella lauseella. */
         piste1 = 0; /* Keskimmäistä vasemmanpuoleista täsmäystä ei ole olemassa. */
      ELSE
```

rekursio1rivi := c^{Y}.rivi – 1; /* Asetetaan vasemmasta yläkulmasta tutkittavan alueen alin … */
rekursio1sarake := c^{Y}.sarake – 1; /* … rivi ja oikea sarake. Alueen alkukohta (Xalku, Yalku). */
pyapituusylävasen = $vasenpos^{YO}$ – 1; /* PYAn pituus ylävasemmassa alueessa. */
piste1 = 1; /* Vasemmalta puolelta löytyi keskimmäinen täsmäys. */
ENDIF (E50)
E51: IF ((c^{O}.rivi > Xloppu) **OR** ($vasenpos^{YO}$ = D_Y + D_O))/* Oikealla ei yhtään kirjattua täsm. */
rekursio2rivi := Xloppu + 1; /* Rajataan alaoikea alue tarkastelujen ulkopuolelle … */
rekursio2sarake := Yloppu + 1; / * … näillä kahdella lauseella. */
piste2 = 0; /* Keskimmäistä alaosan täsmäystä ei ole olemassa. */
ELSE
rekursio2rivi := c^{O}.rivi + 1; /* Asetetaan oikeasta alakulmasta tutkittavan alueen ylin … */
rekursio2sarake := c^{O}.sarake + 1; /* … rivi ja vasen sarake. Alue päättyy (Xloppu, Yloppu). */
pyapituusalaoikea := D_O – 1 – ($vasenpos^{YO}$ – D_Y); /* Asetetaan PYAn pituus alaoikealla. */
piste2 = 1; /* Oikealta puolelta löytyi keskimmäinen täsmäys. */
ENDIF (E51)
ENDIF (E47)
ENDIF (E36)
E52: IF (VaihtoTehty = 1) /* Vaihdettiinko X ja Y päittäin kutsukerran alussa? */
Palauta kaikki päittäin vaihdetut muuttujat ennalleen vaihtamalla ne uudelleen keskenään. */
VaihtoTehty = 0; /* Kumotaan vaihdon voimassaolo. */

/* Tehdään tulosvektoriin tilaa kierroksen aikana löydettyjen täsmäysten tallentamiseen siirtämällä sen indeksiosoittimia, jos vasemmalla ylhäällä sijaitsevasta alueesta tiedetään vielä löytyvän täsmäyksiä. */

ENDIF (E52)
E53: IF ((rekursio1rivi ≥ Xalku) **AND** (rekursio1sarake ≥ Yalku))
S := S + pyapituusylävasen; /* Siirretään tulosvektorin S osoitinta eteenpäin vasemman yläositteen PYAn pituuden verran. */
/* Käynnistetään rekursiivinen kutsu vasemmassa yläkulmassa sijaitsevan alueen tutkimiseksi. */
RatkaisePYA(X, Xalku, rekursio1rivi, rekursio1rivi–Xalku+1, Y, Yalku, rekursio1sarake, rekursio1sarake–Yalku+1, σ, LähiEsTauluX, LähiEsTauluX, LähiEsTauluY*, LähiEsTauluX*, S–pyapituusylävasen, p);
ENDIF (E53)

/* Testataan, löytyikö ainakin yksi kirjattava täsmäys tutkitun alueen ylä- tai vasemmasta ositteesta? */

E54: IF (((tapaus = 1) **AND** (u > 0)) **OR** ((tapaus = 2) **AND** (piste1 = 1)) **OR** (tapaus = 3) **AND** (piste1 = 1)) **OR** (tapaus = 5))
S^ = X[rekursio1rivi + 1]; /* Sijoitetaan vektorin S nykyosoittimen kohdalle symboli vektorista X. */
S := S + 1; /* Siirrytään yksi positio eteenpäin S:ssä. */
p := p + 1; /* Kasvatetaan muuttujaparametrina olevaa PYAn pituutta yhdellä. */
ENDIF (E54)

/* Testataan, löytyikö ainakin yksi kirjattava täsmäys tutkitun alueen ala- tai oikeasta ositteesta? */

E55: IF ((tapaus = 1) **OR** ((tapaus = 2) **AND** (piste2 = 1)) **OR** (tapaus = 3) **AND** (piste2 = 1)) **OR** (tapaus = 4))
S^ = X[rekursio2rivi - 1]; /* Sijoitetaan vektorin S nykyosoittimen kohdalle symboli vektorista X. */
S := S + 1; /* Siirrytään yksi positio eteenpäin S:ssä. */
p := p + 1; /* Kasvatetaan muuttujaparametrina olevaa PYAn pituutta yhdellä. */
ENDIF (E55)

/* Tehdään tulosvektoriin tilaa kierroksen aikana löydettyjen täsmäysten tallentamiseen siirtämällä sen indeksiosoittimia, jos oikealla alhaalla sijaitsevasta alueesta tiedetään vielä löytyvän täsmäyksiä. */

E56: IF ((rekursio2rivi ≤ Xloppu) **AND** (rekursio2sarake ≤ Yloppu))
S := S + pyapituusalaoikea; /* Siirretään tulosvektorin S osoitinta eteenpäin vasemman yläositteen PYAn pituuden verran. */
/* Käynnistetään rekursiivinen kutsu oikeassa alakulmassa sijaitsevan alueen tutkimiseksi. */
RatkaisePYA(X, rekursio2rivi, Xloppu, Xloppu–rekursio2rivi+1, Y, rekursio2sarake, Yloppu, Yloppu–rekursio2sarake+1, σ, LähiEsTauluX, LähiEsTauluX, LähiEsTauluY*, LähiEsTauluX*, S–pyapituusalaoikea, p);
ENDIF (E56)
END (**PROSEDUURI RatkaisePYA**)

ALGORITMI GCL (X, m, Y, n, σ)**:**

Muodosta syöttöaakkoston symboleille taulukot LähiEsTauluX ja LähiEsTauluY vasemmasta yläkulmasta alkavaa syötejonojen tarkastelua varten.
/* LähiEsTauluX ja LähiEsTauluY globaaleja muuttujia, joita ei alustuksen jälkeen päivitetä. */
Muodosta syöttöaakkoston symboleille taulukot LähiEsTauluX* ja LähiEsTauluY* oikeasta alakulmasta alkavaa syötejonojen tarkastelua varten.
/* LähiEsTauluX* ja LähiEsTauluY* globaaleja muuttujia, joita ei alustuksen jälkeen päivitetä. */
RatkaisePYA(X, 1, m, m, Y, 1, n n, σ, LähiEsTauluY, LähiEsTauluX, LähiEsTauluY*, LähiEsTauluX*, S, p); /* S on muuttuja, johon PYA-jono kerätään. */
p := pituus(S);
Tulosta PYA-jono S ja sen pituus p.
END (ALGORITMI GCL).

11.5 Lyhimmän editointietäisyyden määräämiseen kehitetyt algoritmit

11.5.1 Millerin ja Myersin algoritmi (MMY)

ALGORITMI MMY (X, m, Y, n) **(PYAn määräävä versio):**

rivi := Min{i | X[i+1] ≠ Y[i+1], kun 0 ≤ i ≤ m–1}; Ellei tällaista i:tä ole, rivi := m; /* Edetään päälävistäjää pitkin niin pitkälle kuin perättäisiä täsmäyksiä löytyy: */
diagpos[0] := rivi; /* Ilmoittaa alimman rivin, jonne asti edettiin kulkemalla päädiagonaalia pitkin. */
tuhotaan[0] := ¬; /* Ilmoittaa X-vektorista poistettavat merkit edettäessä diagonaalia 0 pitkin. */
IF (rivi = m) /* Saavuttiin viimeiselle riville: PYA on lyhyempi merkkijonoista eli X. */
alindiag := 1 /* Muuttuja alindiag ilmoittaa alimman diagonaalin, jota kannattaa vielä tutkia. */
ELSE /* Ei edetty vielä alimmalle riville asti. Asetetaan alindiag arvoon –1 merkiksi, … */
alindiag := -1; /* … että silmukka S2 aloittaisi seuraavalla kerralla diagonaalilta –1. */
IF (rivi = n) /* Saavuttiin oikeanpuoleisimpaan sarakkeeseen: merkkijonot X ja Y ovat identtiset. */
ylindiag := -1 /* Muuttuja ylindiag ilmoittaa ylimmän diagonaalin, jota vielä kannattaa tutkia. */
ELSE /* Ei edetty vielä oikeanpuoleisimpaan sarakkeeseen asti. Asetetaan ylindiag arvoon 1 merkiksi … */
ylindiag := 1; /* ... että silmukka S2 lopettaisi seuraavalla kerralla diagonaalin 1 jälkeen. */

```
IF (alindiag > ylindiag)  /* Diagonaalien ylä- ja alarajat ristikkäiset? */
    Tulosta ('Merkkijonot ovat identtiset. Ohjelman suoritus lopetetaan.');
ELSE  /* Syötemerkkijonojen välinen etäisyys vähintään 1. */
    etäisyys := 0;  /*  Alustetaan etäisyysmuuttujan arvoksi 0. */
    S1: WHILE (ylindiag ≥ alindiag)
        etäisyys := etäisyys + 1; /* Sijoitetaan etäisyysarvot 2 tarkasteltaville diagonaaleille. */
        k := alindiag;   /* Asetetaan aloitusdiagonaali silmukalle S2. */
        S2: WHILE (k ≤ ylindiag) /* Tutkitaan haarukan [alindiag, ylindiag] diagonaaleista joka toinen. */
            Tutki_diagonaali_k(X, m, Y, n, k, etäisyys, diagpos, rivi, sarake, tuhotaan);
            IF ((rivi = m) AND (sarake = n)) /* Saavuttiin matriisin oikeaan alanurkkaan. */
                Tulosta('Merkkijonojen X ja Y välinen etäisyys on: ', etäisyys);
                Tulosta('PYAn pituus on: ', (m+n–etäisyys) / 2));
            ELSE IF (rivi = m)   /* Saavuttiin matriisin viimeiselle riville. */
                alindiag := k + 2;   /* Ei tutkita enää k:nnen lävistäjän vasemmalta puolelta. */
            ELSE IF (sarake = n)  /* Saavuttiin matriisin viimeiseen sarakkeeseen. */
                ylindiag := k – 2;   /* Ei tutkita enää k:nnen lävistäjän oikealta puolelta. */
            k := k + 2;  /* Vain joka toinen diagonaali väliltä [alindiag, ylindiag] tutkitaan. */
        ENDWHILE (S2)
        alindiag := alindiag – 1;  /* Asetetaan seuraavan diagonaalihaarukan alaraja S2:ta varten. */
        ylindiag := ylindiag + 1  /* Asetetaan seuraavan diagonaalihaarukan yläraja S2:ta varten. */
    ENDWHILE (S1)
Tulostetaan PYA-jono S listaamalla vektorin X merkit kaikista indeksipaikoista, joihin ei kohdistunut
tuhoamisoperaatiota listassa tuhotaan[k].
END (ALGORITMI MMY).

PROSEDUURI Tutki_diagonaali_k(X, m, Y, n, k, etäisyys, var diagpos, rivi, sarake, var tuhotaan):
/* Proseduurissa etsitään diagpos[k]:lle uusi arvo */
IF  ((k = -etäisyys) OR ((k ≠ etäisyys) AND (diagpos[k+1] ≥ diagpos[k–1])))
    /* Tutkittavan lävistäjän oikeanpuoleisella lävistäjällä on edetty vähintään yhtä pitkälle kuin sen
       vasemmanpuoleisella lävistäjällä: sovelletaan lemmaa 4.1. */
    rivi := diagpos[k+1] + 1;  /* Tullaan yksi rivi k+1:nneltä diag. alaspäin, niin päästään k:nnelle. */
    tuhotaan[k] := tuhotaan[k+1] + "Tuhoa merkki X[rivi]"  /* Poistetaan tällä rivillä oleva merkki. */
ELSE  /* Päinvastainen tilanne: vasemmanpuoleista lävistäjää pitkin on edetty pidemmälle kuin
         oikeanpuoleista. */
    rivi := diagpos[k–1];   /*  Jatketaan riviltä, jonne k-1:sen diagonaalin tarkastelu oli pysähtynyt. */
sarake := rivi + k;   /* Sarakkeen numero diagonaalilla k. */
/* Seuraavaksi edetään solusta M[rivi, sarake] niin pitkälle viistoon alaspäin kuin
   rivi < m, sarake < n ja X[rivi+1] = Y[rivi+1]. */
S3: WHILE ((rivi < m) AND (sarake < n) AND (X[rivi+1] = Y[sarake+1]))
    rivi := rivi + 1;
    sarake := sarake + 1;
ENDWHILE (S3)
diagpos[k] := rivi;
END (PROSEDUURI Tutki_diagonaali_k)
```

11.5.2 Myersin algoritmi (MYE)

```
FUNKTIO Etäisyys (X, m, Y, n, κ1, κ2, λ1, λ2):
Δ := n – m;  /* Δ osoittaa syötevektorien pituuseron. */
FOR i := –m, –m + 1, … → n  /* Alustetaan etsintäpositiot diagonaaleilla. */
        DE[i] := 0; /* Alustetaan menosuuntaisten diagonaalien alin löytymisrivi. */
        DT[i] := m + 1; /* Alustetaan paluusuuntaisten diagonaalien ylin löytymisrivi. */
S1: FOR D := 0, 1, … →⌈(m+n)/2⌉ /* Käydään läpi tarvittavat etäisyysarvot. */
    /* Prosessoinnin suunta on ylhäältä alaspäin. */
  S2: FOR k := –D, –D + 2, … → D  /* D:n eri diagonaalit tutkitaan kahden hyppäyksin. */
    Siirry diagonaalille k samaan tapaan kuin algoritmissa MMY eli joko diagonaalilta k + 1 suoraan
    alas riviltä DE[k+1] tai diagonaalilta DE[k–1] oikealle rivillä DE[k–1]
    Etene mahdollisimman pitkälle diagonaalia k pitkin.
    Tallenna lopetusrivin numero DE[k]:hon.
    IF (Δ on pariton) AND (k ∈ [ Δ - (D – 1), Δ + D – 1])
        IF päästään vähintään riville DT[k] asti
            κ1 := X[DT[k]];  /* Otetaan talteen keskimmäisen diagonaalijakson päätepisteet: κ1, */
            κ2 := X[DE[k]]; /* κ2, */
            λ1 := Y[DT[k]+k]; /* λ1 */
            λ2 := Y[DE[k]+k]; /* ja λ2. */
            Etäisyys := 2D – 1;  /* Asetetaan etäisyydelle saatu arvo. */
  ENDFOR (S2)
    /* Prosessoinnin suunta on alhaalta ylöspäin. */
  S3: FOR k := –D, –D + 2, … → D /* D:n eri diagonaalit kahden hyppäyksin. */
    Siirry diagonaalille k joko vasemmalle rivillä DT[k+1] tai suoraan ylöspäin riviltä DT[k–1]
    Kulje mahdollisimman pitkälle taaksepäin diagonaalia k pitkin.
    Tallenna lopetusrivin numero DT[k]:hon.
    IF (Δ on parillinen) AND (k + Δ ∈ [-D, D])
        IF päästään vähintään riville DE[k] asti
            κ1 := X[DT[k]]; /* Otetaan talteen keskimmäisen diagonaalijakson päätepisteet: κ1,  */
            κ2 := X[DE[k]]; /* κ2, */
            λ1 := Y[DT[k]+k]; /* λ1 */
            λ2 := Y[DE[k]+k];  /* ja λ2. */
            Etäisyys := 2D;  /* Asetetaan etäisyydelle saatu arvo. */
  ENDFOR (S3)
ENDFOR (S1)
END (FUNKTIO Etäisyys)

ALGORITMI MYE (X, m, Y, n):
p := 0;
Ratkaise_PYA (X, m, Y, n, p)  /* Kutsutaan proseduuria Ratkaise_PYA koko PYA-ongelmalle. */
Tulosta('PYAn pituus on: ', p);
END (ALGORITMI MYE).

PROSEDUURI Ratkaise_PYA(X, m, Y, n, var p):
IF ((m > 0) AND (n > 0))  /* Kummankin syötejonon oltava ei-tyhjä. */
    D := Etäisyys(X, n, Y, n, κ1, κ2, λ1, λ2);  /* Ratkaistaan ositteen lyhin editointietäisyys. */
```

```
    IF (D > 1)  /* Ositteen käsittely ei ole vielä valmis. */
        Ratkaise_PYA(X[1..κ1], κ1, Y[1..λ1], λ1)  /* Tutkitaan aliosite 1, … */
        Tulosta(X[κ1+1.. κ2]);  /* … tulostetaan keskimmäinen diagonaalijakso … */
        p := p + pituus(X[κ1+1.. κ2]);
        Ratkaise_PYA(X[κ2+1..m], m–κ2, Y[λ2+1..n], n–λ2)  /* ... ja tutkitaan aliosite 2. */
    ELSE   /* Ositteen PYA on lyhyempi kutsun syötejonoista. */
        IF (m < n)
            Tulosta(X[1..m];)
            p := p + m;
        ELSE
            Tulosta(Y[1..n]);
            p := p + n;
END (PROSEDUURI Ratkaise_PYA)
```

11.5.3 Wun, Manberin, Myersin ja Millerin algoritmi (WMM)

```
FUNKTIO Etene_diag(X, Y, k, j):
    i := j – k;  /* Määrätään aloitusrivi. */
    WHILE niin pitkään kuin ((i < m) AND (j < n) ja (X[i + 1] = Y[j + 1]))
        i := i + 1;  /* Siirrytään eteenpäin vektoreissa X … */
        j := j + 1;  /*  … ja Y niin pitkään, kuin perättäisiä täsmäyksiä löytyy. */
    Etene_diag := j  /* Funktio palauttaa arvonaan viimeisen aloitusehdon toteuttavan Y-indeksin. */
END (FUNKTIO Etene_diag)

ALGORITMI WMM (X, m, Y, n):
FOR i := –m, –m + 1, … → n
    diagpos[i] := -1;  /* Alustetaan diagpos-vektorin indeksipaikat alkuarvolla –1. */
tuhotaan[0] := ¬;  /* Ilmoittaa X-vektorista poistettavat merkit edettäessä diagonaalia 0 pitkin. */
t := -1;   /* Alustetaan puristetun etäisyyden laskuri. */
Δ := n – m;   /* Määrätään diagonaali, jolla maalisolmu (m, n) sijaitsee. */
S1: REPEAT
    t := t + 1;  /* Siirrytään tutkimaan uutta puristettua etäisyysarvoa. */
    FOR k := –t, –t + 1, … →Δ – 1  /* Käsitellään Δ:n alapuoleiset diagonaalit. */
        diagpos[k] := Etene_diag(X, Y, k, Max{diagpos[k – 1] + 1, diagpos[k + 1]})
        IF Max{diagpos[k – 1] + 1, diagpos[k + 1]} = diagpos[k + 1]
            tuhotaan[k] := tuhotaan[k+1] + "Tuhoa merkki X[diagpos[k + 1] + 1]"
            /* Poistetaan tällä rivillä oleva merkki. */
    FOR k := Δ + t, Δ + t – 1, … → Δ – 1  /* Käsitellään Δ:n yläpuoleiset diagonaalit. */
        diagpos[k] := Etene_diag(X, Y, k, Max{diagpos[k – 1] + 1, diagpos[k + 1]})
        IF Max{diagpos[k – 1] + 1, diagpos[k + 1]} = diagpos[k + 1]
            tuhotaan[k] := tuhotaan[k+1] + "Tuhoa merkki X[diagpos[k + 1] + 1]"
    diagpos[Δ] := Etene_diag(X, Y, k, Max{diagpos[k – 1] + 1, diagpos[k + 1]});
        /* Käsitellään diagonaali Δ. */
S1: UNTIL (diagpos[Δ] = n)
Tulosta('PYAN pituus on: ', m – t);
```

Tulostetaan PYA-jono S listaamalla vektorin X merkit kaikista indeksipaikoista, joihin ei kohdistunut tuhoamisoperaatiota listassa tuhotaan[Δ].

END (ALGORITMI WMM).

11.6 PYAn pituuden ylärajan laskevat heuristiikat

11.6.1 Minimifrekvenssien summan laskeminen (MFS)

YLÄRAJAHEURISTIIKKA MFS (X, m, Y, n, σ)**:**

FOR i := 1, 2, … → σ /* frekvenssilaskurien nollaaminen */
Xfrekvenssi[i] := 0; /* Nollataan symbolin s_i frekvenssi X:ssä. */
Yfrekvenssi[i] := 0; /* Nollataan symbolin s_i frekvenssi Y:ssä. */
FOR i := 1, 2, … → m /* Selataan vektori X läpi alusta loppuun. */
Xfrekvenssi[X[i]] := Xfrekvenssi[X[i]] + 1; /* Kasvatetaan indeksistä i löytyneen merkin frekvenssiä. */
FOR i := 1, 2, … → n /* Selataan vektori Y läpi alusta loppuun. */
Yfrekvenssi[Y[i]] := Yfrekvenssi[Y[i]] + 1; /* Kasvatetaan indeksistä i löytyneen merkin frekvenssiä. */
FOR i := 1, 2, … →σ /* Otetaan talteen merkkien minimifrekvenssi vektoreissa. */
XYfrekvmin[i] := Min(Xfrekvenssi[i], Yfrekvenssi[i]); /* Tallennetaan pareitt. frekvenssien minimit. */
$p_{ylär}$:= 0;
FOR i := 1, 2, … → σ /* Lasketaan minimifrekvenssien summa yli merkistön. */
$p_{ylär}$:= $p_{ylär}$ + XYfrekvmin[i];
Tulosta PYAn ylärajan pituus muuttujasta $p_{ylär}$.
END (YLÄRAJAHEURISTIIKKA MFS).

11.6.2 Syöttöaakkoston koon supistaminen (SKS)

YLÄRAJAHEURISTIIKKA SKS (X, m, Y, n, σ)**:**

FOR i := 1, 2, … → m /* Kuvataan X:n merkit vektoriin X' … */
X'[i] := (X[i] – 1) MOD σ' + 1; /* … suorittamalla oheinen jakojäännös ja summaus. */
FOR i := 1, 2, … → n /* Kuvataan Y:n merkit vektoriin Y' … */
Y'[i] := (Y[i] – 1) MOD σ' + 1; /* … suorittamalla oheinen jakojäännös ja summaus. */
Ratkaise X':n ja Y':n PYA käyttämällä jotain (asymptoottisesti aakkoston koosta riippuvaa) algoritmia.
$p_{ylär}$:= p(X', Y');
Tulosta PYAn ylärajan pituus muuttujasta $p_{ylär}$.
END (YLÄRAJAHEURISTIIKKA SKS).

11.6.3 Ongelman osittaminen erillisiksi aliongelmiksi (OEA)

YLÄRAJAHEURISTIIKKA OEA (X, m, Y, n, σ)**:**

FOR i := 1, 2, … → σ
Xfrekvenssi[i] := 0; /* Nollataan s_i:n frekvenssi X:ssä. */
Yfrekvenssi[i] := 0; /* Nollataan s_i:n frekvenssi Y:ssä. */
FOR i := 1, 2, … → m /* Lasketaan X:n merkkien frekvenssit. */
Xfrekvenssi[X[i]] := Xfrekvenssi[X[i]] + 1;
FOR i := 1, 2, … → n /* Lasketaan Y:n merkkien frekvenssit. */
Yfrekvenssi[Y[i]] := Yfrekvenssi[Y[i]] + 1;

FOR i := 1, 2, … →σ /* Otetaan minimi merkkien esiintymistä X:ssä ja Y:ssä. */
XYfrekvmin[i] := Min(Xfrekvenssi[i], Yfrekvenssi[i]);
Lajittele kekolajittelun avulla XYfrekvmin-vektorin k:n ensimmäisen merkin frekvenssit nousevaan suuruusjärjestykseen siten, että tiedetään, mihin merkkiin mikin frekvensseistä liittyy lajittelun jälkeenkin.
Käy läpi loput σ – k indeksipaikkaa vektorista XYfrekvmin ja pidä yllä minimikekoa k:sta yleisimmästä merkistä ja niiden frekvenssistä. Kyseisiä merkkejä indikoi merkintä {yleisimmät}.
p := 1; /* vektorin X_1 indeksilaskuri */
q := 1; /* vektorin X_2 indeksilaskuri */
u := 1; /* vektorin Y_1 indeksilaskuri */
v := 1; /* vektorin Y_2 indeksilaskuri */
FOR i := 1, 2, … → m /* Asetetaan vektorin X merkit ositteisiin X_1 ja X_2. */
IF (X[i] ∈ {yleisimmät}) /* Oliko yleiseksi luokiteltava merkki? */
X_1[p] := X[i]; / /* Oli: viedään se vektoriin X_1 … */
p := p + 1; /* … ja kasvatetaan sen indeksilaskuria p. */
ELSE
X_2[q] := X[i] /* Ei, vaan harvinainen: viedään se vektoriin X_2 … */
q := q + 1; /* … ja kasvatetaan sen indeksilaskuria q. */
FOR i := 1, 2, … → n /* Asetetaan vektorin Y merkit ositteisiin Y_1 ja Y_2. */
IF (Y[i] ∈ {yleisimmät}) /* Oliko yleiseksi luokiteltava merkki? */
Y_1[u] := Y[i]; / /* Oli: viedään se vektoriin Y_1 … */
u := u + 1; /* … ja kasvatetaan sen indeksilaskuria u. */
ELSE
Y_2[v] := Y[i] /* Ei, vaan harvinainen: viedään se vektoriin Y_2 … */
v := v + 1; /* … ja kasvatetaan sen indeksilaskuria v. */
Ratkaise X_1:n ja Y_1:n PYA käyttämällä jotain tarkkaa (syöttöaakkoston koosta riippuvaa) algoritmia.
Talleta 1. aliongelman PYAn pituus muuttujaan $p_1$ylär.
Ratkaise X_2:n ja Y_2:n PYA käyttämällä jotain tarkkaa (syöttöaakkoston koosta riippuvaa) algoritmia.
Talleta 2. aliongelman PYAn pituus muuttujaan $p_2$ylär.
$p_{ylär}$:= $p_1$ylär + $p_2$ylär;
Tulosta PYAn ylärajan pituus muuttujasta $p_{ylär}$.
END (YLÄRAJAHEURISTIIKKA OEA).

11.7 PYAn pituuden alarajan laskevat heuristiikat

11.7.1 Syöttöaakkoston yleisimmän merkin frekvenssi (YMF)

ALARAJAHEURISTIIKKA YMF (X, m, Y, n, σ)**:**

FOR i := 1, 2, … → σ /* frekvenssilaskurien nollaaminen */
Xfrekvenssi[i] := 0; /* Nollataan symbolin s_i frekvenssi X:ssä. */
Yfrekvenssi[i] := 0; /* Nollataan symbolin s_i frekvenssi Y:ssä. */
FOR i := 1, 2, … → m /* Selataan vektori X läpi alusta loppuun. */
Xfrekvenssi[X[i]] := Xfrekvenssi[X[i]] + 1; /* Kasvatetaan indeksistä i löytyneen merkin frekvenssiä. */
FOR i := 1, 2, … → n /* Selataan vektori Y läpi alusta loppuun. */
Yfrekvenssi[Y[i]] := Yfrekvenssi[Y[i]] + 1; /* Kasvatetaan indeksistä i löytyneen merkin frekvenssiä. */
MaxFrekvenssi := 0; /* Ylläpitää tietoa suurimmasta arvosta vektorissa XYfrekvmin: alustus nollaksi. */
FOR i := 1, 2, … → σ /* Otetaan talteen merkkien minimifrekvenssi vektoreissa. */
XYfrekvmin[i] := Min(Xfrekvenssi[i], Yfrekvenssi[i]); /* Tallennetaan pareittaisten frekv. minimit. */

IF (XYfrekvmin[i] > MaxFrekvenssi) /* Löytyikö uusi frekvenssin maksimiehdokas? */
MaxFrekvenssi := XYfrekvmin[i]; /* Kyllä: otetaan sekä arvo … */
MaxIndeksi := i; /* … että sen sijaintipaikka talteen. */
p_{alar} := MaxFrekvenssi;
FOR i := 1, 2, … → MaxFrekvenssi /* Yhteinen alijono saadaan …. */
S[i]:= $s_{MaxIndeksi}$; /* … syöttämällä vektoriin S merkkiä $s_{MaxIndeksi}$ MaxFrekvenssi kappaletta. */
Tulosta alarajan pituus p_{alar} sekä sen mittainen X:n ja Y:n yhteinen alijono S.
END (ALARAJAHEURISTIIKKA YMF).

11.7.2 Chinin ja Poonin heuristinen algoritmi (CPH)

ALARAJAHEURISTIIKKA CPH (X, m, Y, n, σ)**:**

FOR i := 1, 2, … → σ /* frekvenssilaskurien nollaaminen */
Xfrekvenssi[i] := 0; /* Nollataan symbolin s_i frekvenssi X:ssä. */
Yfrekvenssi[i] := 0; /* Nollataan symbolin s_i frekvenssi Y:ssä. */
FOR i := 1, 2, … → m /* Selataan vektori X läpi alusta loppuun. */
Xfrekvenssi[X[i]] := Xfrekvenssi[X[i]] + 1; /* Kasvatetaan indeksistä i löytyneen merkin frekvenssiä. */
FOR i := 1, 2, … → n /* Selataan vektori Y läpi alusta loppuun. */
Yfrekvenssi[Y[i]] := Yfrekvenssi[Y[i]] + 1; /* Kasvatetaan indeksistä i löytyneen merkin frekvenssiä. */
i : = 1; /* Asetetaan X-kursori. */
j := 1; /* Asetetaan Y-kursori. */
k := 1; /* Asetetaan tulosvektorin S kursori. */
WHILE (i ≤ m) **AND** (j ≤ n) /* Suoritetaan niin kauan kuin syötevekt. on merkkejä. */
IF (X[i] = Y[j]) /* Löytyikö täsmäys? */
S[k] := X[i]; /* Kyllä: viedään sen muodostanut merkki tulosvektoriin S. */
Xfrekvenssi[X[i]] := Xfrekvenssi[X[i]] – 1; /* Pienennetään merkin frekvenssiä X:ssä … */
Yfrekvenssi[Y[j]] := Yfrekvenssi[Y[j]] – 1; /* … ja sitten Y:ssä. */
i := i + 1; /* Siirrytään eteenpäin X:ssä, … */
j := j + 1; /* … Y:ssä … */
k := k + 1; /* … ja tulosvektorissa S. */
ELSE
IF Min(Xfrekvenssi[X[i]], Yfrekvenssi[X[i]]) < Min(Xfrekvenssi[Y[j]], Yfrekvenssi[Y[j]])
Xfrekvenssi[X[i]] := Xfrekvenssi[X[i]] – 1; /* X[i] harvin.: pienennetään sen frekv. ... */
i := i + 1; /* … ja siirretään X-kursoria. */
ELSE
Yfrekvenssi[Y[j]] := Yfrekvenssi[Y[j]] – 1; /* Y[j] harvin.: pienennetään sen frekv. ... */
j := j + 1; /* … ja siirretään Y-kursoria. */
p_{alar} := k – 1; /* Alarajan pituus on selvitetty. */
Tulosta alarajan muodostavat merkit vektorista S sekä sen pituus p_{alar}.
END (ALARAJAHEURISTIIKKA CPH).

11.7.3 PYAMAX-heuristiikka (PMX)

ALARAJAHEURISTIIKKA PMX (X, m, Y, n, σ)**:**

Varataan muistia dynaamisesti m + n + 1 solmun verran. Jokaista syöttöaakkoston merkkiä kohti perustetaan X- ja Y-vektorin täsmäyslistat Xmerkit[s_g] ja Ymerkit[s_g] ($1 \leq g \leq \sigma$), jotka ilmaisevat aakkoston eri merkkien sijaintipaikat kummassakin syötevektorissa alusta loppua kohti. Dynaamisen rakenteen ensimmäistä solmua käsitellään pysäytyssolmuna, ja siihen viitataan tunnuksella alku. Kun vektorin jonkin merkin täsmäyslista osoittaa pysäytyssolmuun, ei merkillä ole enää esiintymiä kyseisen vektorin loppuosassa.

```
alku^.indeksi := n + 1; /* Muuttuja alku osoittaa listan pysäytyssolmuun, jonka indeksinä on n + 1. */
FOR i := 1, 2, … → σ  /* Alustetaan vektorien merkkien esiintymälistat tyhjiksi … */
        Xmerkit[i] := Ymerkit[i] := alku;  /* … asettamalla ne viittaamaan pysäytyssolmuun. */
solmu := alku^.seuraava;  /* Paikasta alku^.seuraava alkavat syötevektorien merkkien sijaintilistat. */
FOR i := m, m – 1, … → 1  /* Perustetaan X:ssä esiintyvien symbolien täsmäyslistat. */
        tilap := solmu;   /*  Asetetaan lisäosoitin tarkasteltavaan solmuun. */
        tilap^. indeksi := i;  /* Asetetaan X:n merkin indeksi esiintymälistaan. */
        tilap^.seuraava := Xmerkit[X[i]];  /* Asetetaan tarkasteltava solmu listan Xmerkit[X[i]] alkuun. */
        X[merkit[X[i]] := tilap;  /* Siirretään lista X[merkit[X[i]] alkamaan tarkasteltavasta solmusta. */
        solmu := solmu^.seuraava /* Siirrytään seuraavan solmun kohdalle listassa. */
FOR i := n, n – 1, … → 1  /* Perustetaan Y:ssä esiintyvien symbolien täsmäyslistat. */
        tilap := solmu;   /*  Asetetaan lisäosoitin tarkasteltavaan solmuun. */
        tilap^. indeksi := i;  /* Asetetaan Y:n merkin indeksi esiintymälistaan. */
        tilap^.seuraava := Ymerkit[Y[i]];  /* Asetetaan tarkasteltava solmu listan Ymerkit[Y[i]] alkuun. */
        Y[merkit[Y[i]] := tilap;  /* Siirretään lista Y[merkit[Y[i]] alkamaan tarkasteltavasta solmusta. */
        solmu := solmu^.seuraava /* Siirrytään seuraavan solmun kohdalle listassa. */
p_alar := 0;  /* Alustetaan heuristisen PYAn pituus nollaksi. */
lopetetaan := epätosi;  /* Estetään ohjelman suorituksen keskeytyminen alkuunsa. */
Xpositio := 1;  /* Asetetaan aloituskohta X-vektorin tutkimiselle. */
Ypositio := 1:  /* Asetetaan aloituskohta Y-vektorin tutkimiselle. */
WHILE (lopetetaan = epätosi)
      minX := m + 1;  /* Toistaiseksi pieninumeroisin rivi, jolta on löydetty täsmäys. */
      minY := n + 1;  /*  Toistaiseksi pieninumeroisin sarake, jolta on löydetty täsmäys. */
      lopetetaan := tosi;  /* Nostetaan lopetuslippu ylös. */
      WHILE (Xpositio ≤ m) AND (Ymerkit[X[Xpositio]] = alku) /* Löytyikö X[i] Y:stä? */
            Xmerkit[X[Xpositio]] := Xmerkit[X[Xpositio]]^.seuraava; /* Ei: siirretään X[i]-kursoria. */
            Xpositio := Xpositio + 1;  /* Siirrytään vektorissa X yksi merkki eteenpäin. */
       IF (Xpositio ≤ m)
            minX := Xpositio;  /* Ylin rivi, jolla sijaitsevalla merkillä on vielä vastine Y:n lopussa. */
            oikea := Ymerkit[X[Xpositio]]^. indeksi;  /* Y:n sarake, josta X:n merkki löydettiin. */
            lopetetaan := epätosi;  /* Täsmäys löytyi: ei voida lopettaa vielä ohjelman suoritusta. */
       IF (lopetetaan = epätosi)  /* Löydettiin k-täsmäys. */
            WHILE (Xmerkit[Y[Ypositio]] = alku) /* Y:n merkki löytyy nyt varmasti. */
                  Ymerkit[Y[Ypositio]] := Ymerkit[Y[Ypositio]]^.seuraava; /* Siirretään Y[i]-kursoria. */
                  Ypositio := Ypositio + 1;  /* Siirrytään vektorissa Y yksi merkki eteenpäin. */
            minY := Ypositio;  /* Vasemmanpuoleisin sarake, jonka merkillä on vastine X:ssä. */
            AlinRivi := Xmerkit[Y[Ypositio]]^.indeksi;  /* Rivi, jolla esiintyy merkki Y[Ypositio]. */
            maksimi := 0;  /* Alustetaan maksimaalisen yhteisen minimiloppuliitteen pituus nollaksi. */
            MaxYind := oikea + 1;  /* Tutkittavan alueen oikeaa reunaa seuraava sarake. */
             FOR i := Xpositio, Xpositio + 1, ... → AlinRivi /* Tutkitaan suorakulmion rivit. */
                   IF (Ymerkit[X[i]]^.indeksi < MaxYind)  /* Onko täsmäystä laatikon sisällä? */
```

MinEtäisyys := Min{m–i, n–Ymerkit[X[i]]^.indeksi} /* On: lask. minimiet. */
IF MinEtäisyys > maksimi /* Löytyikö uusi maks. minimietäisyys? */
maksimi := MinEtäisyys; /* Kyllä: kirjataan se muistiin .... */
maxXind := i; /* … kuten myös sen X- … */
maxYind := Ymerkit[X[i]]^.indeksi; /* … ja Y-koordinaatti. */
p_{alar} := p_{alar} + 1; /* Heuristisen PYAn pituus kasvaa yhdellä. */
S[p_{alar}] := X[maxXind];
WHILE (Xpositio ≤ maxXind) /* Siirretään X-kursori ohi kirjatun täsmäyksen. */
Xmerkit[X[Xpositio]] := Xmerkit[X[Xpositio]]^.seuraava;/* Siirretään X[i]-kursoria. */
Xpositio := Xpositio + 1; /* Siirrytään X:ssä yksi positio eteenpäin. */
WHILE (Xpositio ≤ maxXind) /* Siirretään Y-kursori ohi kirjatun täsmäyksen. */
Ymerkit[Y[Ypositio]] := Ymerkit[Y[Ypositio]]^.seuraava; /* Siirretään Y[i]-kursoria. */
Ypositio := Ypositio + 1; /* Siirrytään Y:ssä yksi positio eteenpäin. */

Tulosta alarajan muodostavat merkit vektorista S sekä sen pituus p_{alar}.
END (ALARAJAHEURISTIIKKA PMX).

11.7.4 Syöttöaakkostoltaan supistetun ongelman täsmäysten tarkastaminen (SST)

ALARAJAHEURISTIIKKA SST (X, m, Y, n, σ)**:**

Laske PYAn pituuden yläraja käyttämällä SKS-ylärajaheuristiikkaa. Otetaan ylärajan pituus talteen muuttujaan $p_{ylär}$ ja sen täsmäysten indeksit muuttujaan lista. Täsmäyksen X-koordinaattia esittää kenttä Xind ja y-koordinaattia kenttä Yind.

p_{alar} := 0; /* Alustetaan heuristisen PYAn pituus nollaksi. */
** Xedell := 0; /* Edellisen täsmäyksen X-koordinaatti: alustetaan nollaksi (0, 0):n mukaan. */
** Yedell := 0; /* Edellisen täsmäyksen X-koordinaatti: alustetaan nollaksi (0, 0):n mukaan. */
** valetäsmäys := epätosi; /* Kontrollimuuttuja sille, kohdattiinko valetäsmäys. */
FOR k := 1, 2, … → $p_{ylär}$ /* Käydään läpi ylärajan muodostaneet täsmäykset. */
i := lista^.Xind; /* Luetaan täsmäyksen X-indeksi … */
j := lista^.Yind; /* ... ja sitten Y-indeksi. */
lista := lista^.seuraava; /* Edetään listassa seuraavaan täsmäykseen. */
IF (X[i] = Y[j]) /* Oliko kyseessä aito täsmäys? */
** **IF** (valetäsmäys = tosi) /* Kyllä: entä oliko tätä ennen esiintynyt valetäsmäys? */
** Laske tarkka PYA osajonoille X[Xedell + 1..i – 1], Y[Yedell + 1..j – 1]. /* Kyllä oli. */
** Olkoon saadun tarkan PYAn pituus z ja sijaitkoot sen merkit vektorissa Z.
** **FOR** u := 1, 2, … → z /* Liitetään suorakulmion täsmäykset. */
** p_{alar} := p_{alar} + 1; /* Alaraja pitenee yhdellä. */
** S[p_{alar}] := Z[u];
** valetäsmäys := epätosi; /* Viimeksi tutkittu täsmäys ei ollut valetäsmäys. */
p_{alar} := p_{alar} + 1; /* Edellinen täsmäys ei ollut huti. Alaraja pitenee yhdellä. */
S[p_{alar}] := X[i]; /* Viedään merkki X[i] ratkaisuun. */
** Xedell := i; /* Juuri kirjatun täsmäyksen rivi. */
** Yedell := j; /* Juuri kirjatun täsmäyksen sarake. */
** **ELSE** /* Kohdattiin valetäsmäys ... */
** valetäsmäys := tosi; /* … ja nostetaan lippu sen merkiksi. */
** **IF** (valetäsmäys = tosi) /* Oliko viimeinen täsmäys valetäsmäys? */
** Laske tarkka PYA osajonoille X[Xedell + 1..m], Y[Yedell + 1..n]. /* Kyllä oli. */

** Olkoon saadun tarkan PYAn pituus z ja sijaitkoot sen merkit vektorissa Z.
** **FOR** u := 1, 2, … → z /* Liitetään suorakulmion täsmäykset. */
** $p_{alar} := p_{alar} + 1$; /* Alaraja pitenee yhdellä. */
** $S[p_{alar}] := Z[u]$;
Tulosta alarajan muodostavat merkit vektorista S sekä sen pituus p_{alar}.
END (ALARAJAHEURISTIIKKA SST).

11.7.5 FASTA-heuristiikka (FTA)

ALARAJAHEURISTIIKKA FTA (X, m, Y, n)**:**

raja := t; /* Parametri, jonka käyttäjä itse määrää (>0). */
FOR i := 1, 2, … → n
Aloita merkkien lukeminen vektorista Y paikasta i. Lue Y:stä merkkejä parametrin raja osoittama määrä ja laske merkkien osoitteeksi hajautustaulussa niiden bittiesityksen XOR-operaation tulos. Vie luettu osajono laskettuun osoitteeseen ja tallenna esiintymän alkukohdaksi i. Jos sama merkkijono on taulussa jo ennestään, perusta sille vain uusi esiintymäkohta.
FOR i := –m, –m + 1, … → n
diagonaalit[i] := []; /* Perustetaan diagonaalikohtaiset täsmäyslistat. Diagonaalin numero määräytyy vähentämällä sarakkeen koordinaatista rivikoordinaatti. Mitä pienempi diagonaalin itseisarvo on, sitä lähempänä päälävistäjää ollaan. */
i := 1; /* Alustetaan X-vektorin laskuri. */
S1: WHILE niin pitkään kuin (raja + i – 1 ≤ m)
Lue X:stä parametrin raja osoittama määrä merkkejä. Aloita paikasta i. Laske merkkien bittiesityksen XOR-operaation tulos ja etsi hajautustaulun kyseisestä osoitteesta, löytyykö sieltä samaa merkkijonoa.
IF löytyi
S2: REPEAT
Lisää matriisiin M rajan mittainen täsmäys, joka alkaa paikasta (i, j), missä j on esiintymän alkukoordinaatti Y:ssä. Kyseinen täsmäys viedään listaan diagonaalit[j – i].
Tutki, päättyykö listan diagonaalit[j – i] viimeinen täsmäys paikkaan (i – 1, j – 1).
IF diagonaalit[j – i]^.viimeinen^.Xloppu = i – 1 /* Jatkaako aiempaa t-ketjua? */
täsmäys := diagonaalit[j – i]^.viimeinen; /* Kyllä vaan. */
täsmäys^.pituus := täsmäys^.pituus + raja; /* Päivitetään täsmäyksen pit. */
täsmäys^.Xloppu := täsmäys^.Xloppu + raja; /* Sitten X:n loppukoord. … */
täsmäys^.Yloppu := täsmäys^.Yloppu + raja; /* … ja sitten Y:n loppukoord. */
ELSE /* Lisätään uusi täsmäyssolmu diagonaalille j – i */
täsmäys := diagonaalit[j – i]^.viimeinen^.seuraava; /* Perust. uusi solmu. */
täsmäys^.pituus := raja; /* Kirjataan jakson pituus, … */
täsmäys^.Xalku := i; /* … alkamiskohta X:ssä, … */
täsmäys^.Yalku := j; /* … alkamiskohta Y:ssä, … */
täsmäys^.Xloppu := i + raja – 1; /* … loppumiskohta X:sssä ja … */
täsmäys^.Yloppu := j +raja – 1; /* … vielä loppumiskohta Y:ssä .*/
diagonaalit[j – i]^.viimeinen := täsmäys; /* Suora yhteys viim. solmuun. */
S2: UNTIL kaikki esiintymät Y:ssä on käsitelty
i := i + raja;
ENDWHILE (S1)

Linkitä diagonaalit, jotka sisältävät täsmäyksiä, yhdeksi listaksi, joka on nimeltään täsmäykset. Linkittäminen alkaa diagonaalilta 0 ja jatkuu järjestyksessä diagonaaleille 1, -1, 2, -2, …, -m, m, … n.
p_{alar} := 0; /* Alustetaan alarajan pituus nollaksi. */
Suorita rekursiivinen proseduuri MuodostaAlijono(täsmäykset, p_{alar}).
Tulosta alarajan muodostava jono S ja sen pituus p_{alar}.
END (ALARAJAHEURISTIIKKA FTA).

PROSEDUURI MuodostaAlijono(täsmäykset, p_{alar}, **var** S)**:**
/* Käydään läpi matriisin M kaikki riittävän pitkät täsmäysketjut ja etsitään pisin löydetty täsmäävä osajono. */
maxpituus := 0; /* Alustetaan pisin täsmäysjakso pituudeltaan nollaksi. */
täsmäys := täsmäykset; /* Siirrytään täsmäysten listan alkuun. */
WHILE täsmäyksiä riittää
pituus := täsmäys^.pituus;
IF (pituus > maxpituus) /* Päivitetään maksimin pituisen täsmäysketjun sijaintitiedot. */
maxpituus := pituus; /* Tallennetaan uusi maksimi, … */
maxXalku := täsmäys^.Xalku; /* … sen alkukohta X:ssä, … */
maxYalku := täsmäys^.Yalku; /* … sen alkukohta Y:ssä, … */
maxXloppu := täsmäys^.Xloppu; /* … sen loppukohta X:ssä … */
maxYloppu := täsmäys^.Yloppu; /* … ja sen loppukohta Y:ssä. */
täsmäys := täsmäys^.seuraava; /* Siirrytään seuraavaan täsmäykseen. */
täsmäys := täsmäykset; /* Palataan takaisin täsmäysten listan alkuun. */
WHILE täsmäyksiä riittää /* Pisin täsmäysjakso on selvillä: poistetaan turhat täsmäykset listasta sen valinnan mukaan. */
IF (täsmäys^.Xalku < maxXalku) **AND** (täsmäys^.Yalku < maxYalku) /* Voi mahtua alkuun. */
Xloppu := Min(Xloppu, maxXalku – 1); /* Pätkitään tarpeen mukaan lopusta. */
Yloppu := Min(Yloppu, maxYalku – 1); /* Tehdään samoin Y:n mukaiselle loppupisteelle. */
täsmäys^.pituus := Xloppu – Xalku + 1; /* Päivitetään täsmäyksen pituus ajan tasalle. */
Vie täsmäys listaan Alkuosa.
ELSE
IF (täsmäys^.Xloppu > maxXloppu) **AND** (täsmäys^.Yloppu > maxYloppu)/* Loppuun? */
Xalku := Max(Xalku, maxXloppu + 1) /* Pätkitään tarpeen mukaan alusta. */
Yalku := Max(Yalku, maxYloppu + 1) /* Tehdään samoin Y:n mukaiselle loppupist. */
täsmäys^.pituus := Xloppu – Xalku + 1; /* Päivitetään täsm. pituus ajan tasalle. */
Vie täsmäys listaan Loppuosa.
ELSE /* Täsmäys on jo kokonaisuudessaan päällekkäinen valitun jakson kanssa. */
Vie täsmäys roskakoriin. Sitä ei tarvita enää mihinkään.
IF Alkuosan lista ei ole tyhjä /* Onko vasemmassa yläkulmassa vielä täsmäyksiä jäljellä? */
Suorita rekursiivinen proseduuri MuodostaAlijono(Alkuosa, p_{alar}). /* On: tutkitaan kyseinen alue. */
FOR j := 0, 1, … → maxpituus – 1 /* Liitetään pisin täsmäysjakso heurist. PYAn. */
p_{alar} := p_{alar} + 1; /* Kasvatetaan alarajan pituutta yhdellä. */
S[p_{alar}] := X[maxXalku + 1]; /* Liitetään jakson j. merkki alarajajonoon. /
IF Loppuosan lista ei ole tyhjä /* Onko oikeassa alakulmassa vielä täsmäyksiä jäljellä? */
Suorita rekursiivinen proseduuri MuodostaAlijono(Loppuosa, p_{alar}). /* On: tutkitaan alue. */
END (PROSEDUURI MuodostaAlijono)

Kirjallisuusluettelo

[Aho76] Aho, A. V. & Hirschberg, D. S. & Ullman, J. D.: *Bounds on the Complexity of the Longest Common Subsequence Problem*, Journal of the Association for Computing Machinery, Vol. 23, n:o 1, tammikuu 1976, sivut 1 – 12.

[Alg10] Verkkosivu http://www.thealgorithmist.com/showthread.php/133-LCS-of-three-strings, linkin toimivuus testattu 2012-05-23.

[All86] Allison, L. & Dix, T. I.: *A Bit-String Longest-Common-Subsequence algorithm*, Information Processing Letters 23 (1986), North-Holland, 1986, sivut 305 – 310.

[Apo85] Apostolico, A. & C. Guerra: *A Fast Linear Space Algorithm for Computing Longest Common Subsequences*, Purdue University, tekninen raportti 546, 1985.

[Apo87] Apostolico, A. & Guerra, C.: *The Longest Common Subsequence Problem Revisited*, Algorithmica (1987) 2, sivut 315 – 336.

[Apo92] Apostolico, A. & Browne, S. & Guerra, C.: *Fast linear-space computations of longest common subsequences*, Theoretical Computer Science 92 (1992), Elsevier, 1992, sivut 3 – 17.

[Bae96] Baeza-Yates, R. A. & Gavaldà, R. & Navarro, G.: *Bounding the Expected Length of Longest Common Subsequences and Forests*, Proceedings of WSP'96, Recife, Brasilia, elokuu 1996, sivut 1 – 15.

[Ber05] Bergroth, L. *Utilizing Dynamically Updated Estimates in Solving the Longest Common Subsequence Problem,* Proceedings of SPIRE 2005, Buenos Aires, Argentiina, Springer-Verlag, marraskuu 2005, sivut 301 – 314.

[Ber06] Bergroth, L. *Duomenų struktūrų poveikis algoritmo efektyvumui ieškant dviejų sekų bendrą ilgiausią posekį* (suomeksi: *Tietorakenteiden vaikutus algoritmin tehokkuuteen etsittäessä kahden merkkijonon pisintä yhteistä alijonoa*), Vadyba 2006, Nr. 2(9), Vakarų Lietuvos verslo kolegija: Mokslo tiriamieji darbai, Klaipėda, Liettua, lokakuu 2006, sivut 15 – 19.

[Ber07] Bergroth, L.: *Pagerintas euristinis algoritmas dviejų sekų bendro ilgiausio posekio paieškai* (suomeksi: *Parannettu heuristinen algoritmi kahden merkkijonon pisimmän yhteisen alijonon etsintään*), Vadyba 2007, Nr. 2(11), Vakarų Lietuvos verslo kolegija: Mokslo tiriamieji darbai, Klaipėda, Liettua, lokakuu 2007, sivut 32 – 36.

[Ber08] Bergroth, L.: *Elokuvia Alankomaista ja Belgiasta*, kurssiin Alankomaiden ja Belgian maantuntemus liittyvä harjoitustyö, Helsingin yliopisto, saksalainen laitos, hollannin kieli ja kulttuuri, kesäkuu 2008.

[BHR00] Bergroth L., Hakonen H., Raita T. *A Survey of Longest Common Subsequence Algorithms*. Proceedings of SPIRE 2000, A Coruña, Espanja, IEEE Computer Society, 2000, sivut 39 – 48.

[BHR98] Bergroth, L. & Hakonen H. & Raita T.: *New Approximation Algorithms for Longest Common Subsequences*, Proceedings of SPIRE 1998, Santa Cruz de la Sierra, Bolivia, syyskuu 1998, sivut 32 – 40.

[BHV03] Bergroth, L. & Hakonen, H. & Väisänen, J: *New Refinement Techniques for Longest Common Subsequence Algorithms*, Proceedings of SPIRE 2003, Manaus, Brasilia, lokakuu 2003, sivut 287 – 303.

[Bob00] Boberg, J.: Kurssin *Johdatus tietojenkäsittelytieteeseen* luentomoniste, Turun yliopisto, täydennyskoulutuskeskus, elokuu 2000.

[Chi90] Chin, F. Y. L. & Poon, C. K.: *A Fast Algorithm for Computing Longest Common Subsequences of Small Alphabet Size*, Journal of Information Processing, Vol. 13, n:o 4, 1990, sivut 463 – 469.

[Chi94] Chin, F. & Poon, C. K.: *Performance Analysis of Some Simple Heuristics for Computing Longest Common Subsequences*, Algorithmica (1994) 12: sivut 293 – 311.

[Chv75] Chvátal, V. & Sankoff, D.: *Longest Common Subsequences of Two Random Sequences*, Journal of Applied Probability 12, 1975, sivut 306 – 315.

[CLR93] Cormen, T. H. & Leiserson, C. E. & Rivest, R. L.: *Introduction to Algorithms*, 9. painos, The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993.

[Cro01] Crochemore, M. & Iliopoulos, C. S. & Pinzon, Y. J. & Reid, J. F.: *A Fast and Practical Bit-Vector Algorithm for the Longest Common Subsequence Problem*, Information Processing Letters, Volume 80, Issue 6, joulukuu 2001, sivut 279 – 285.

[Dan94] Dančík, V.: *Expected Length of Longest Common Subsequences*, Väitöskirja, University of Warwick, syyskuu 1994.

[Dan95] Dančík, V. & Paterson, M.: *Upper Bounds for the Expected Length of a Longest Common Subsequence of Two Binary Sequences*, Random Structures & Algorithms 6, 4, sivut 449 – 458.

[Dek78] Deken, J. G.: *Some Limit Results for Longest Common Subsequences*, Discrete Mathematics 26 (1979), North-Holland, sivut 17 – 31.

[Emd77] van Emde Boas, P.: *Preserving order in a forest in less than logarithmic time and linear space*, Information Processing Letters 6 / 1977, sivut 80 – 82.

[Fra92] Fraser, C. B. & Irving, R. W.: *Two algorithms for the longest common subsequence of three* (*or more*) *strings*, Combinatorial pattern matching, Lecture notes in Computer Science, Volume 644/1992, Springer-Verlag, sivut 214 – 229.

[Fra95] Fraser, C. B.: *Subsequences and Supersequences of Strings*, University of Glasgow, Computing Science, väitöskirja, 1995.

[Fre75] Fredman, M. *On Computing the Length of Longest Increasing Subsequences*, Discrete Mathematics 11 (1975), North-Holland, sivut 29 – 35.

[Goe99] Goeman, H. & Clausen, M.: *A New Practical Linear Space Algorithm for the Longest Common Subsequence Problem*, Proceedings of the Prague Stringology Club Workshop '99, Praha 1999.

[Hal08] Halinen, J: *Torium – vaihtoehtoinen ydinpolttoaine*, julkaisussa Energiateollisuus ry: Ydinvoima ja innovaatiot, ISBN 978-952-5615-25-8, Helsinki 2008, sivut 23 – 27.

[Hil74] Hillier, F. S. & Lieberman, G. J.: *Introduction to Operations Research*, Holden-Day, San Francisco, 1967.

[Hir75] Hirschberg, D. S.: *A Linear Space Algorithm for Computing Maximal Common Subsequences*, Communications of the ACM, Vol. 18, nro 6, kesäkuu 1975, sivut 341 – 343.

[Hir77] Hirschberg, D. S.: *Algorithms for the Longest Common Subsequence problem*, Journal of the Association for Computing Machinery, Vol. 24, n:o 4, lokakuu 1977, sivut 664 – 675.

[Hir78] Hirschberg, D. S.: *An Information-Theoretic Lower Bound for the Longest Common Subsequence Problem*, Information Processing Letters, Vol. 7, n:o 1, tammikuu 1978, sivut 40 – 41.

[Hsu84] Hsu, W. J. & Du, M. W.: *New Algorithms for the LCS Problem*, Journal of Computer and System Sciences 29, 1984, sivut 133 – 152.

[Hun77] Hunt, J. W. & Szymanski, T. G.: *A Fast Algorithm for Computing Longest Common Subsequences*, Communications of the ACM, Vol. 20, nro 5, toukokuu 1977, sivut 350 – 353.

[Hwa72] Hwang, F. K. & Lin, S.: *A simple algorithm for merging two disjoint linearly ordered sets*, SIAM J. Comput. 1 (1972), sivut 31 – 39.

[Ili02] Iliopoulos, C. S. & Pinzon, Y. J.: *Recovering an LCS in* $O(\frac{n^2}{\omega})$ *time and space*, Revista Colombiana de Computación / Colombian Journal of Computation – RCC, Vol. 3, n:o 1, 2002, sivut 41 – 51.

[Irv92] Irving, R. W. & Fraser, C. B.: *Two Algorithms for the Longest Common Subsequence of Three* (*or More*) *Strings*, Combinatorial Pattern Matching, Lecture Notes in Computer Science, 1992, Volume 644/1992, Springer-Verlag, sivut 214 – 229.

[Jia95] Jiang, T. & Li, M.: *On the Approximation of Shortest Common Supersequences and Longest Common Subsequences*, SIAM: Society for Industrial and Applied Mathematics, Vol. 24, n:o 5, lokakuu 1995, sivut 1122 – 1139.

[Joh96] Johtela, T., Smed, J., Hakonen, H., Raita, T.: *An Efficient Heuristic for the LCS problem*, Proceedings of the Third South American Workshop on String Processing, WSP'96, Recife, Brasilia, elokuu 1996, sivut 126 – 140.

[Kal07] Kalilainen, J.: *Torium ydinpolttoaineena*, tekstidokumentti Internetissä, verkkosivu http://www.tkk.fi/Units/AES/courses/crspages/Tfy-56.181_07/Kalilainen_text.doc, linkin toimivuus testattu 2012-05-23.

[Kiw05] Kiwi, M. & Loebl, M. & Matoušek, J.: *Expected Length of the Longest Common Subsequence for Large Alphabets*, Advanced Mathematics 197, 2, sivut 480 – 498.

[Kum87] Kiran Kumar, S. & Pandu Rangan, S.: *A Linear Space Algorithm for the LCS Problem*, Acta Informatica 24, 1987, sivut 353 – 362.

[Kuo89] Kuo, S. & Cross, G. R.: *An Improved Algorithm to Find the Length of the Longest Common Subsequence of Two Strings*, ACM SIGIR Forum, Spring / Summer 1989, Vol. 23, n:o. 3 – 4, 1989, sivut 89 – 99.

[Lue09] Lueker, G. S.: *Improved Bounds on the Average Length of Longest Common Subsequences*, Journal of the ACM, Vol. 56, n:o 3, artikkeli 17, toukokuu 2009.

[Mai78] Maier, David: *The Complexity of Some Problems on Subsequences and Supersequences*, Journal of the Association for Computing Machinery, Vol. 25, n:o 2, huhtikuu 1978, sivut 322 – 336.

[Mas80] Masek, W. J. & Paterson, M. S.: *A faster algorithm for computing string edit distances*, Journal of Computer and System Sciences 20 (1), 1980, sivut 18 – 31.

[Mau74] Maurer, H.: *Datenstrukturen und Programmierverfahren*, B.G. Teubner, Stuttgart, 1974, sivut 95 – 96.

[Meh84] Mehlhorn, K.: *Data Structures and Algorithms 1: Sorting and Searching*, EACTS Monographs on TCS, Springer-Verlag, Berlin, 1984.

[Mil85] Miller, W. & Myers, E. W.: *A File Comparison Program*, Software – Practice and Experience, Vol. 15(11), marraskuu 1985, sivut 1025 – 1040.

[Muk80] Mukhopadhyay, A.: *A Fast Algorithm for the Longest-Common-Subsequence Problem*, Information Sciences 20, Elsevier North Holland Inc., 1980, sivut 69 – 82.

[Mye86] Myers, E. W.: An O(ND) *Difference Algorithm and Its Variations*, Algorithmica (1986) 1: Springer-Verlag, sivut 251 – 266.

[NKY82] Nakatsu, N. & Kambayashi, Y. & Yajima, S.: *A Longest Common Subsequence Suitable for Similar Text Strings*, Acta Informatica 18, Springer-Verlag, 1982, sivut 171 – 179.

[PBL08] *Pôle Bioinformatique Lyonnais*, verkkosivu http://pbil.univ-lyon1.fr/alignment.html, linkin toimivuus testattu 2012-05-23.

[PLi88] Pearson, W. R. & Lipman, D. J.: *Improved Tools for Biological Sequence Comparison*, Proc. Natl. Acad. Sci. USA 85, huhtikuu 1988.

[Rai94] Raita, T.: Kurssin *Ohjelmoinnin metodiikka luentomoniste*, Turun yliopisto, 1994, sivut 17 – 18.

[Rai96] Raita, T.: *Tietorakenteiden erikoiskurssin luentomoniste*, Turun yliopisto, 1996, sivut 15 – 17.

[Ric00_1] Rick, C.: *Efficient Computation of All Longest Common Subsequences*, Algorithm Theory – SWAT2000: Lecture Notes in Computer Science, Springer-Verlag Berlin / Heidelberg, Vol. 1851 / 2000, tammikuu 2000, sivut 687 – 697.

[Ric00_2] Rick, C.: *Simple and Fast Linear Space Computation of Longest Common Subsequences*, Information Processing Letters, Vol. 75 (6), Elsevier North-Holland, 2000, sivut 275 – 281.

[Ric94] Rick, C.: *New Algorithms for the Longest Common Subsequence Problem*, Institut für Informatik der Universität Bonn, Research Report No. 85123-Cs, lokakuu 1994.

[ROA09] Rovaniemen ammattikorkeakoulun kurssin *Salausmenetelmät: osa 2* verkkosivu, osoite http://ta.ramk.fi/~jouko.teeriaho/krypto2006/salausmenetelmat2_2klassiset.pdf, linkin toimivuus testattu 2012-05-23.

[Sch61] Schensted, C.: *Longest increasing and decreasing subsequences*, Canadian Journal of Mathematics 13, 1961, sivut 179 – 191.

[Sim89] Simon, Imre: *Sequence Comparison: Some Theory and Some Practice*, Electronic Dictionaries and Automata in Computational Linguistics, Lecture Notes in Computer Science, 1989, Volume 377/1989, Springer-Verlag, sivut 79 – 92.

[Wag74] Wagner, R. A. & Fischer, M. J.: *The String to String Correction Problem*, Journal of the Association of Computing Machinery, Vol. 21, nro 1, 1974, sivut 168 – 173.

[WMM90] Wu, S. & Manber, U. & Myers, G. & Miller, W.: *An O(NP) Sequence Comparison Algorithm*, Information Processing Letters 35 (1990), North-Holland, sivut 317 – 323.

[Won76] Wong, C. K. & Chandra, A. K.; *Bounds for the String Editing Problem*, Journal of the Association for Computing Machinery, Vol. 23, n:o 1, tammikuu 1976, sivut 13 – 16.

[Zip35] Zipf, G. K.*The Psychobiology of Language*. Houghton – Mifflin, 1935.

Turku Centre for Computer Science
TUCS Dissertations

114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Communication and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming
128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen

Turku Centre *for* Computer Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www. tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Division for Natural Sciences and Technology
- Department of Information Technologies

ISBN 978-952-12-2756-1
ISSN 1239-1883

Lasse Bergroth

Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen