Tampereen yliopisto

Antti Pham

MERKKIJONOHAUN TEKNIIKAT JA ALGORITMIT LUONNOLLISEN TEKSTIN NOPEAAN HAKUUN

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Huhtikuu 2023

TIIVISTELMÄ

Antti Pham: Merkkijonohaun tekniikat ja algoritmit luonnollisen tekstin nopeaan hakuun
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Huhtikuu 2023

Merkkijonohaut ovat tietojenkäsittelytieteen yksi klassisista aiheista. Merkkijonohakua on tutkittu tietojenkäsittelytieteen alusta lähtien, ja ne ovat perustana hakukoneille, jotka ovat nykyiselle yhteiskunnalle tärkeitä. Merkkijonohaussa *teksti* on merkkijono, josta *hakusanoja* etsitään. Tarkastelen luonnollisen kielen merkkijonohakuja, jotka löytävät *kaikki* hakuosumat *tarkalla merkkijonohaulla*, eli hakuosumaa ei tule yhdenkään merkin erosta hakusanassa. Työssäni tutkin merkkijonohaun tekniikoita ja esitän nopeimpia hakualgoritmeja, kun tekstin esikäsittely sallitaan ja kun sitä ei sallita. Tutkimukseni jakautuu siis kolmeen osaan: merkkijonohaun tekniikat, nopeimmat hakualgoritmit ilman tekstin esikäsittelyä ja nopeimmat hakualgoritmit tekstin esikäsittelyllä.

Selvitän työssäni merkkijonohaun tekniikoita tarkastelemalla naiivia algoritmia, Boyer-Moore-algoritmia, bittirinnakkaisuutta käyttäviä algoritmeja, suodattavia algoritmeja ja suffiksitaulukon hakualgoritmeja. Otan näistä tarkempaan tarkasteluun naiivin algoritmin ja Boyer-Mooren. Osoitan, miten vertailujärjestyksen muutoksella naiivi algoritmi saadaan varsin tehokkaaksi, ja selitän, miten Boyer-Mooren käänteisestä vertailujärjestyksestä saadaan huonon merkin ja hyvän suffiksin sääntö havaintojen perusteella. Työssäni saan tekniikoiksi vertailujärjestyksen muuttaminen, huonon merkin sääntö, hyvän suffiksin sääntö, hakusanan esikäsittely, parhaan hakualgoritmin valitseminen, suodatus, bittirinnakkaisuus ja suffiksitietorakenne.

Selvitän nopeimmat merkkijonohakualgoritmit kirjallisuudesta. Saan työssäni selville, että EPSMA (Exact Packed String Matching AVX2) ja SSEFA (Streaming SIMD Extensions Filter AVX2) ovat nopeimpia merkkijonohakualgoritmeja, jotka eivät esikäsittele tekstiä mutta voivat esikäsitellä hakusanaa. Yhteistä näissä molemmissa algoritmeissa on se, että ne käyttävät suodatusta ja bittirinnakkaisuutta.

Kun sallitaan tekstin esikäsittely, voidaan käyttää suffiksitietorakennetta, jolla pystyy poistamaan merkkijonohaun aikavaatimuksen riippuvuuden tekstin pituudesta. Valitsen suffiksitietorakenteista suffiksitaulukon, ja esitän kirjallisuudesta useita tapoja hakea merkkijonoja suffiksitaulukolla. Nopeimman merkkijonohaun voi toteuttaa Burrows-Wheeler-muunnoksella, jonka avulla merkkijonohaut onnistuvat hakusanan pituuden suhteen lineaarisessa ajassa.

Avainsanat: nopea, tarkka merkkijonohaku, algoritmi, tietorakenne, suffiksitaulukko

SISÄLLYSLUETTELO

1 Johdanto

Merkkijonohaut ovat tietojenkäsittelytieteen yksi klassisista aiheista. Merkkijonohaut ovat olleet mielenkiinnon kohteena tämän tieteenalan alusta lähtien, ja merkkijonojen hakualgoritmeja vieläkin kehitetään. Merkkijonohaut ovat perustana koko nykyiselle yhteiskunnalle, sillä tiedonhakua vaaditaan kaikissa aloissa. Lisäksi digitalisaation ja kasvavan datan määrän vuoksi nopeilla merkkijonohauilla on suuri merkitys.

Merkkijonohaussa etsitään pitkästä luonnollisesta *tekstistä*, missä kaikissa kohdissa esiintyy *hakusana*. *Tekstin* määrittelen työssäni merkkijonoksi, josta *hakusanoja* etsitään. Merkkijonohakuja voidaan luokitella eri tavoilla. Merkkijonohakua voi suorittaa siten, että löytää vain yhden hakuosuman tai sitten kaikki hakuosumat. Lisäksi sen voi luokitella tarkkoihin ja likimääräisiin hakuihin. (Hakak ym., 2019) Työni tarkastelee merkkijonohakuja, jotka löytävät *kaikki* hakuosumat *tarkalla haulla*, eli hakuosumaa ei tule yhdenkään merkin erosta hakusanassa.

Mailund (2020) esittää kirjassaan tärkeimpiä tapoja toteuttaa merkkijonohakuja. Kirjassa kerrotaan, että merkkijonohakuja voidaan tehdä merkkijonohakualgoritmeilla tai tietorakenteilla. Merkkijonohakualgoritmi määritellään työssäni merkkijonohaun algoritmiksi, joka yrittää löytää kaikki hakuosumat hakusanalle samalla, kun teksti käydään läpi. Merkkijonohakualgoritmi ei siis saa esikäsitellä tekstiä. Merkkijonohaku tietorakenteella eroaa edellisestä siten, että se saa esikäsitellä tekstiä esimerkiksi käymällä tekstin läpi ja tallentamalla sen eri muotoon, ennen kuin aloitetaan merkkijonohaku. Työssäni otan nämä molemmat tavat huomioon.

Työni tarkastelee tilannetta, jossa luonnollisen kielen tekstiä on kerätty runsaasti eikä se muutu merkkijonohakujen aikana. Datasta pitäisi etsiä hakuosumia suurella määrällä eri hakusanoilla mahdollisimman nopeasti. Lisäksi tekstin pituus on paljon suurempi kuin hakusanan pituus. Tämä voisi vastata esimerkiksi kyselyiden suorittamista tietokannassa, kun kyselyitä tulee jatkuvasti.

Edellä mainitusta tilanteesta tutkin, *mitä tekniikoita merkkijonohaussa on ja mitkä ovat nopeimmat merkkijonohaun algoritmit*. Nopeimmilla algoritmeilla tarkoitan algoritmeja, joiden suoritusajat ovat ajaltaan lyhyimpiä. Koska tarkastelen merkkijonohakualgoritmien ja tietorakenteiden näkökulmista, tutkimuskysymykseni on kolmiosainen:

1. *mitä tehokkaita tekniikoita merkkijonohaussa on*
2. *mitkä ovat nopeimmat hakualgoritmit, jotka löytävät kaikki hakuosumat tarkalla haulla, kun tekstiä ei saa esikäsitellä*
3. *mitkä ovat nopeimmat hakualgoritmit, jotka löytävät kaikki hakuosumat tarkalla haulla, kun tekstiä saa esikäsitellä.*

Työni esittelee toisessa luvussa tutkimusmenetelmän, joka on kirjallisuuskatsaus. Kolmannessa luvussa esitän määritelmät ja merkintätavat, joita käytän työssäni. Neljännessä luvussa selvitän merkkijonohaun tekniikoita käymällä tarkasti läpi naiivin algoritmin ja Boyer-Moore-algoritmin. Viidennessä luvussa selvitän nopeimmat merkkijonohakualgoritmit, EPSMA ja SSEFA.

Kuudennessa luvussa esitän merkkijonohaun algoritmeja, kun sallitaan tekstin esikäsittely. Tekstin esikäsittelyssä käytän suffikseihin perustuvaa merkkijonon tietorakennetta, suffiksitaulukkoa. Nopein merkkijonohaun algoritmi suffiksitaulukosta käyttää Burrows-Wheeler-muunnosta.

2 Tutkimusmenetelmä

Työni tutkimusmenetelmä on kirjallisuuskatsaus. Etsin kirjallisuutta eri hakukoneista ja julkaisufoorumeista. Hain Andor- ja Google Scholar -hakukoneilla, ja käytin seuraavia tietokantoja: ACM Digital Library, IEEE Xplore, ScienceDirect (Elsevier) ja SpringerLink.

Hain lähteitä mukauttaen loogisia operaattoreita sopimaan eri hakukoneisiin ja tietokantoihin tällä hakusanalla: *(pattern OR string) AND (match OR search) AND algorithm AND exact AND (NOT approximate)*. Hakusanan alkuosan avainsanat valitsin tutkimuskysymykseni nojalla käyttäen synonyymejä. Hakusanat ”exact” ja ”NOT approximate” lisäsin rajatakseni pois likimääräiset merkkijonohaut, koska tutkimuskysymykseni koskee ainoastaan tarkkoja merkkijonohakuja. Osumia tuli paljon näillä hakusanoilla, joten rajasin julkaisuja vuodesta 2013 eteenpäin. Tein myös toisen hakukierroksen lisättyäni tarkentavan hakusanan ”*AND fast*”, jotta löytäisin nopeita merkkijonohaun algoritmeja. Merkkijonohaun tekniikoita en etsinyt suoraan hakusanalla vaan löysin ne seuraamalla lähteiden lähteitä.

Kun löysin sopivia lähteitä, seurasin niiden lähteitä jatkuvasti taakse sitä mukaa kuin sopivia lähteitä vielä löytyi. Kuitenkin yksi käyttämistäni lähteistä on kirja, johon ei ole merkattu lähteitä, joten en pystynyt kirjan lähteistä hakemaan sen lähteitä, kun löysin kiinnostavia algoritmeja ja käsitteitä. Tällöin hain alkuperäisjulkaisun hakukoneilla ja tietokannoilla avainsanoilla.

Lähteen totesin sopivaksi, jos se liittyi luonnollisen kielen tarkkaan merkkijonohakuun ja siinä verrataan eri merkkijonohaun algoritmeja. Jätin kuitenkin lähteitä pois, jos lähteessä esitellään uusi algoritmi vertailematta riittävän perusteellisesti muita algoritmeja. Nämä tein tarkastelemalla hakuosumien otsikon, tiivistelmän, yhteenvetoluvun ja kuvaajat.

3 Määritelmät ja merkintätavat

$\boldsymbol{A[i]}$ tarkoittaa taulukon A alkiota, joka sijaitsee indeksissä i. Taulukon ensimmäinen alkio on indeksissä 0. Esimerkiksi taulukolle $B = [1, 2, 3, 4]$ pätee $B[2] = 3$.

$\boldsymbol{A[i..j]}$ tarkoittaa taulukon A alitaulukkoa indeksistä i indeksiin j siten, että alkio $A[j]$ ei ole mukana alijonossa. Eli taulukolle B pätee $B[0..2] = [0, 1]$.

Merkki merkataan lainausmerkkeihin siten, että itse merkki on kursivoitu: "A".

Merkkijonoille käytetään samaa merkintää kuin taulukoille, eli merkkijonolle $C = \text{"ABCD"}$ pätee $C[1] = \text{"}B\text{"}$.

Prefiksi määritellään merkkijonon aluksi, kun koko merkkijono ja tyhjä merkkijono otetaan mukaan. Merkkijonon T kaikki prefiksit ovat $T[i..n]$, kun n on merkkijonon pituus ja $0 \leq i \leq n$ (Taulukko 1). Prefiksejä on siis yhteensä $n + 1$ kappaletta.

Taulukko 1. Merkkijonon $T = "AABAADAAAAD"$ kaikki prefiksit.

```
 T[0..0] = ""
 T[0..1] = A
 T[0..2] = AA
 T[0..3] = AAB
 T[0..4] = AABA
 T[0..5] = AABAA
 T[0..6] = AABAAD
 T[0..7] = AABAADA
 T[0..8] = AABAADAA
 T[0..9] = AABAADAAA
T[0..10] = AABAADAAAA
T[0..11] = AABAADAAAAD
```

Suffiksi määritellään merkkijonon päätteeksi, kun koko merkkijono ja tyhjä merkkijono otetaan mukaan. Merkkijonon T kaikki suffiksit ovat $T[i..n]$, kun $0 \leq i \leq n$ (Taulukko 2). Suffikseja on myös yhteensä $n + 1$ kappaletta.

Taulukko 2. Merkkijonon $T = "AABAADAAAAD"$ kaikki suffiksit.

```
 T[0..n] = AABAADAAAAD
 T[1..n] = ABAADAAAAD
 T[2..n] = BAADAAAAD
 T[3..n] = AADAAAAD
 T[4..n] = ADAAAAD
 T[5..n] = DAAAAD
 T[6..n] = AAAAD
 T[7..n] = AAAD
 T[8..n] = AAD
 T[9..n] = AD
T[10..n] = D
T[11..n] = ""
```

Väreillä havainnollistetaan esimerkin hakusanan ja tekstin merkkien yhtäsuuruus esimerkeissä. Hakusanan vihreä merkki tarkoittaa sitä, että sen tiedetään sopivan sitä vastaavan tekstin tai kuvion vihreän merkin kanssa. Vastaavasti hakusanan punainen merkki ja sitä vastaava tekstin punainen merkki merkitsevät sitä, että ne eroavat toisistaan.

4 Merkkijonohaun tekniikat

Tutkin merkkijonohaun tekniikoita käymällä yksityiskohtaisesti läpi yleisiä algoritmeja. Tekniikat ovat tärkeitä, koska ne ovat olleet perustana muille merkkijonohakualgoritmeille ja osa niistä käytetään vieläkin. En voi kuitenkaan käydä kaikkia tekniikoita läpi, koska niitä on paljon (Hakak ym., 2019). Sen vuoksi käyn tässä läpi vain naiivia algoritmia, joka toimii hyvänä johdantona merkkijonohakuihin, ja klassista Boyer-Moore-algoritmia.

Naiivi algoritmi on yksinkertaisin merkkijonohakualgoritmi, ja sen toteutus näkyy lähestulkoon kaikissa ohjelmointikielten kirjastoissa, koska se on helppo toteuttaa. Tämä ei kuitenkaan

tarkoita, että se on kaikista huonoin hakualgoritmi. Sen vahvuus on sen yksinkertaisuudessa, jonka vuoksi se on helppo optimoida (Tarhio ym., 2017).

Boyer-Moore on klassinen merkkijonohakualgoritmi ja se on ollut varhaisen merkkijonohakualgoritmin tutkimuksien yleinen vertailukohde (Hume & Sunday, 1991). Sen tekniikoita on sovellettu moniin muihin hakualgoritmeihin, ja siitä on monta variaatiota (Hakak ym., 2019), joten se on yksi merkittävimmistä hakualgoritmeista. Boyer-Moore ja sen yksinkertaistettu versio, lisättiin C++17-standardissa standardikirjastoon (ISO, 2017). Boyer-Moore on siis klassinen algoritmi, jota vieläkin käytetään.

Seuraavaksi käydään edellä mainitut algoritmit yksityiskohtaisesti läpi ja selvitetään merkkijonohakuun liittyviä tekniikoita tutkimalla, miten nämä algoritmit toimivat. Läpikäynnissä ei kuitenkaan käydä kaikkia mahdollisia tapauksia läpi vaan painotus on algoritmien tekniikoiden ymmärtämisessä.

4.1 Naiivi algoritmi

Esitän naiivin algoritmin, joka toimii ajassa $O(nm)$, missä n on tekstin pituus ja m on hakusanan pituus. Algoritmi käy tekstin jokaisen m-pituisen alijonon läpi ja vertaa sen merkkejä hakusanan merkkeihin. Alijonon ja hakusanan vertailu toteutetaan tarkastelemalla alijonon ja hakusanan merkit ensimmäisestä merkistä viimeiseen merkkiin. Algoritmin aikavaatimus on $O(nm)$, koska alijonoja on n kappaletta ja jokaisen alijonon kohdalla verrataan hakusanan kaikki m merkkiä. Todellisuudessa alijonoja on $n - m + 1$ kappaletta, mutta kirjallisuudessa yleensä sievennetään alijonojen määrä $n - m + 1 \approx n$.
Olen havainnollistanut algoritmin toiminta, kun $T = "AABAADAAAAD"$ ja $H = "AAD"$ (

Esimerkki 1). Esimerkit eivät ole luonnollista tekstiä, koska yritän esittää naiivin algoritmin haittapuolet toistoa sisältävällä tekstillä ja esimerkin tulisi olla riittävän lyhyt, jotta esimerkit eivät olisi liian pitkiä.

Esimerkissäni käytettiin hakusanaa ja tekstiä, jotka sisältävät paljon toistoa. Tämä johti siihen, että tehtiin 14 onnistunutta vertailua (esimerkissä vihreällä) ja 7 epäonnistunutta vertailua (esimerkissä punaisella), eli 21 vertailua yhteensä. Tekstin T pituus on 11, joten algoritmi teki noin 1,9 vertailua jokaista merkkiä kohti. Tämä ei ole kovin tehokasta.

Esimerkki 1. Naiivissa algoritmissa jokaisella rivillä tarkastellaan merkkijonon *T* eri alijonoja hakusanaan *H* vertailemalla merkkejä järjestyksessä vasemmalta oikealle. Vihreä väri tarkoittaa sitä, että alijonon ja hakusanan vastaavat merkit ovat samat, ja punainen väri tarkoittaa sitä, että merkit ovat erilaiset. Hakusana esiintyy tekstin kohdissa 3 ja 8.

T = AABAADAAAAD
T[0..3] = AAB H = AAD
T[1..4] = ABA H = AAD
T[2..5] = BAA H = AAD
T[3..6] = AAD H = AAD
T[4..7] = ACA H = AAD
T[5..8] = CAA H = AAD
T[6..9] = AAA H = AAD
T[7..10] = AAA H = AAD
T[8..11] = AAD H = AAD

Naiivia algoritmia voidaan optimoida, kun hakusanaa ja alijonoja ei verrata ensimmäisestä merkistä viimeiseen vaan verrataan merkkien esiintymistiheyden käänteisessä järjestyksessä. Voidaan tarkastella merkkien esiintymistiheyttä ja selvittää sen avulla optimaalinen vertaamisjärjestys. (Hume & Sunday, 1991) Tämän voi tehdä käymällä läpi tekstin *T* tai jonkin luonnollisen tekstin kaikki merkit läpi ja sen perusteella asettaa hakusanan vertaamisjärjestys.
Otetaan esimerkiksi sama teksti *T* ja hakusana *H* kuin aikaisemmin (

Esimerkki 1). Ensin pitää laskea merkkien esiintymistiheys tekstistä: merkki ”A” esiintyy 8 kertaa, ”D” esiintyy 2 kertaa ja ”B” esiintyy 1 kerran. Siis optimaalinen vertaamisjärjestys on ensin ”B”, sitten ”D” ja lopuksi ”A”, joten hakusanan $H = "AAD"$ yksi tarkastelujärjestys voisi olla $H[2]$, $H[0]$, $H[1]$.

Algoritmi toimii nyt paljon tehokkaammin (Esimerkki 2). Paranneltu naiivi algoritmi teki yhteensä 13 vertailua 11-pituiseen tekstiin, mistä saadaan noin 1,18 vertailua jokaista merkkiä kohden. Optimointi siis paransi algoritmin tehokkuutta merkittävästi. Tästä saadaan työn ensimmäinen tekniikka: vertailujärjestyksen muuttaminen.

On yritetty selvittää, kuinka monta vertailua keskimäärin tehdään tekstin merkkiä kohti parannellulla naiivilla algoritmilla (Hume & Sunday, 1991). Tämä on testattu Kuningas Jaakon Raamatun yhden megatavun osalla ja vastaukseksi saatiin 1,08 vertausta jokaista merkkiä kohti, mikä on tehokas.

Esimerkki 2. Parannellussa naiivissa algoritmissa vertailujärjestys on eri. Ensin vertaillaan viimeistä merkkiä, sitten ensimmäistä merkkiä ja lopuksi keskimmäistä merkkiä.

T = AABAADAAAAD
T[0..2] = AAB H = AAD
T[1..3] = ABA H = AAD
T[2..4] = BAA H = AAD
T[3..5] = AAD H = AAD
T[4..6] = ACA H = AAD
T[5..7] = CAA H = AAD
T[6..8] = AAA H = AAD
T[7..9] = AAA H = AAD
T[8..10] = AAD H = AAD

4.2 Boyer-Moore

Nopeasti ajatellen lineaariaikainen merkkijonohakualgoritmi, kuten Knuth-Morris-Pratt-algoritmi (Knuth ym., 1977), joka toimii ajassa $\Theta(n)$, vaikuttaisi olevan tehokkain merkkijonon hakualgoritmi, koska pitäähän n-pituisesta merkkijonosta tarkastella kaikki merkit. Tämän luvun tarkastelun kohde, Boyer-Moore, on siitä mielenkiintoinen, että se voi toimia vielä tehokkaammin ajassa $\Omega\left(\frac{n}{m}\right)$, sillä kustannuksella, että sen asymptoottinen yläraja on $O(nm)$ (Boyer & Moore, 1977). Huonommasta ylärajasta huolimatta Boyer-Moore on yleensä nopeampi kuin lineaarinen algoritmi, kun merkkijonot ovat luonnollista tekstiä.

Boyer-Mooren tehokkuus perustuu siihen, että sen vertailujärjestys on muutettu. Tässä siis käytetään samankaltaista tekniikkaa kuin optimoidussa naiivissa algoritmissa. Nyt kuitenkin vertailujärjestys on viimeisestä merkistä ensimmäiseen merkkiin. Kun vertailujärjestys asetetaan tällaiseksi, voidaan havaintojen perusteella keksiä optimointeja algoritmille (Boyer & Moore, 1977).

Seuraavissa luvuissa tarkastelen m'-pituista hakusanaa $H' =$ "anpanman" ja n'-pituista tekstiä $T' =$ "etsittävä_sana_ei_ole_pan_eikä_onpanman_vaan_anpanman" ja esitän havaintoja, joiden perusteella muodostetaan Boyer-Mooren kaksi sääntöä: huonon merkin sääntö (bad character rule), ja hyvän suffiksin sääntö (good suffix rule). Seuraavissa luvuissa mainitut havainnot ja säännöt ovat Boyer-Moore-algoritmin julkaisusta (Boyer & Moore, 1977), mutta olen selittänyt omalla tavallani, miten havainnoista päästään sääntöihin.

4.2.1 Huonon merkin sääntö

Ensimmäinen vertailu kohdistuu merkkeihin $H' =$ "n" ja $T'[7] =$ "v" (Esimerkki 3). Kun tutkitaan näitä kahta merkkiä ja yritetään selvittää, mikä olisi paras kohta jatkaa vertailua, huomataan, että hakusanassa H' ei ole merkkiä "v". Naiivissa algoritmissa vertailu jatkuisi seuraavasta alijonosta $T'[1..9]$, mutta koska tiedetään, että hakusanassa ei ole merkkiä "v", voidaan suoraan todeta, että alijonojen $T'[1..9], T'[2..10], \ldots, T'[7..15]$ vertailut voidaan sivuuttaa (Boyer & Moore, 1977). Paras kohta jatkaa vertailua on siis siirtyä m' merkkiä eteenpäin kohtaan $T'[8..16]$ (Esimerkki 3).

Esimerkki 3. Tekstissä T' on merkki "v", joka ei esiinny hakusanassa, joten voidaan siirtyä m' merkkiä eteenpäin.

```
      T' = etsittävä_sana_ei_ole_pan_eikä_onpanman_vaan_anpanman
T'[0..8] = etsittäv
      H' = anpanman
       T'[8..16] = ä_sana_e
             H' = anpanman
```

Tämä havainto voidaan myös yleistää merkeille, jotka ovat hakusanassa tarkastelemalla hakusanan pisintä suffiksia, joka ei sisällä tekstin epäonnistuneen vertailun merkkiä (Boyer & Moore, 1977). Esimerkiksi, jos hakusanan H' merkkiä verrataan tekstin merkkiin "p", kannattaa siirtyä 5 merkkiä eteenpäin, koska hakusanan suffiksi $H'[3..m'] =$ "anman" on pisin suffiksi, joka ei sisällä merkkiä "p", ja se on 5 merkin pituinen.

Kun tätä havainnon yleistystä tarkastellaan, huomataan, että tekstin T' ja hakusanan H' merkit "p" menevät täydellisesti kohdakkain (Esimerkki 4). Siispä toinen tapa selittää tämä havainto on merkkien $T'[i]$ ja $H'[j]$ vertailun epäonnistuessa siirtää hakusana H' eteenpäin niin kauan, kunnes merkki $T'[i]$ ja hakusanan H' merkki ovat samat tai kunnes on edetty m' merkkiä, eli hakusanan merkit on käyty läpi.

Esimerkki 4. Tekstissä on merkki $T'[22] = "p"$, joka esiintyy hakusanassa H'. Kun edetään viisi merkkiä eteenpäin, saadaan tekstin merkki $T'[22] = "p"$ ja hakusanan merkki $H'[2] = "p"$ kohdakkain.

```
T' = etsittävä_sana_ei_ole_pan_eikä_onpanman_vaan_anpanman
T'[15..23] = ei_ole_p
H' = anpanman
T'[20..28] = e_pan_ei
H' = anpanman
```

Tästä havainnosta saadaan Boyer-Mooren ensimmäinen sääntö ja työn uusi tekniikka: huonon merkin sääntö (Boyer & Moore, 1977). Huonon merkin säännön perusteella voidaan luoda hakusanalle H' huonon merkin taulukko, jota voidaan käsitellä $delta_1$-funktiolla ennen merkkijonohakua (Taulukko 3). Taulukkoon on laskettu etukäteen siirtymät kaikille merkeille edellisten havaintojen mukaisesti.

Taulukko 3. Huonon merkin taulukkoa voidaan käsitellä $delta_1$-funktiolla hakusanalle $H' =$ "anpanman". Taulukko määrittää jokaiselle merkille, kuinka monta merkkiä kannattaa siirtyä eteenpäin merkkijonohaussa vertailussa epäonnistuneen tekstin merkin perusteella.

merkki	"a"	"n"	"p"	"m"	**muu**
$delta_1$	1	0	5	2	8

Huonon merkin sääntö kattaa myös tilanteen, jossa merkkien $T'[i]$ ja $H'[j]$ epäonnistunut vertailu ei tapahdu hakusanan viimeisellä merkillä vaan jossain hakusanan keskellä. Tällaisessa tilanteessa siirrytään eteenpäin $delta_1(T'[i])$ merkkiä kohdan i suhteen. Siis seuraava vertailu kohdistuu merkkeihin $T'[i + delta_1(T'[i])]$ ja $H'[m' - 1]$ (Esimerkki 5).

Esimerkki 5. Kun merkin $T'[22] = "p"$ vertailu epäonnistuu (merkattu punaisella), huonon merkin säännön nojalla täytyy siirtyä epäonnistuneelta merkiltä $delta_1("p") = 5$ merkkiä eteenpäin, joten seuraava vertailu kohdistuu merkkiin $T'[22 + 5] = T'[27]$. Siis seuraava väli on $T'[20..28]$.

```
T' = etsittävä_sana_ei_ole_pan_eikä_onpanman_vaan_anpanman
T'[17..25] = _ole_pan
H' = anpanman
T'[20..28] = e_pan_ei
H' = anpanman
```

4.2.2 Hyvän suffiksin sääntö

Hyvän suffiksin sääntöä merkataan funktiolla $delta_2$. Sääntö tulee seuraavien havaintojen perusteella. Luodaan kuvio aiempien vertailujen perusteella, ja etsitään hakusanasta viimeisin kohta, joka sopii kuvioon. Tässä sallitaan myös kuvion suffiksin löytyminen hakusanan prefiksistä, jos

kuvion suffiksin ja hakusanan prefiksi ovat samanpituisia. Koska kuvio sisältää aikaisempien vertailujen tiedot, merkkijonohaussa voidaan välttää samojen merkkien vertailuja ja täten nopeuttaa merkkijonohakua. (Boyer & Moore, 1977)

Tarkastellaan tilannetta, jossa vertailu epäonnistuu hakusanan kolmanneksi viimeisen merkin kohdalla, $H'[m' - 3] = "m"$. Tässä tärkeä havainto on se, että aikaisempi vertailu epäonnistui merkille $H'[m' - 3] = "m"$ mutta onnistui merkeille $H'[m' - 2] = "a"$ ja $H'[m' - 1] = "n"$. Tästä saadaan kolmemerkkinen kuvio "~~m~~an": ei merkki "m", merkki "a" ja merkki "n".

Kun verrataan kuviota hakusanaan H', saadaan selville, että viimeisin kohta, johon kuvio sopii, on kohta $H'[m' - 6..m' - 3] = "pan"$ (Esimerkki 6). Tässä siis tarkastellaan viimeisintä kohtaa, koska vertailujärjestys on viimeisestä merkistä ensimmäiseen. Koska kohta $H'[m' - 6..m' - 3]$ löytyy silloin, kun kuviota siirretään 3 merkkiä viimeisestä merkistä, voidaan päätellä, että kannattaa siirtää väliä 3 merkkiä eteenpäin, kun kolmanneksi viimeisimmän merkin vertailu epäonnistuu (Esimerkki 7) (Boyer & Moore, 1977).

Esimerkki 6. Hakusanassa H' viimeisin kohta, jossa kuvio "~~m~~an" esiintyy, löytyy, kun siirretään kuviota 3 merkkiä taaksepäin.

```
                H’ = anpanman
H’[m’-4..m’-1] = nma
          kuvio: man
H’[m’-5..m’-2] = anm
          kuvio: man
H’[m’-6..m’-3] = pan
          kuvio: man
```

Esimerkki 7. Kun merkin $T'[i] = "p"$ vertailu epäonnistuu (merkattu punaisella), hyvän suffiksin säännön nojalla voidaan siirtyä väliltä $T'[17..25]$ kolme merkkiä eteenpäin välille $T'[20..28]$, koska kuvio "~~m~~an" sopii tähän väliin eikä mihinkään aikaisempaan väliin. Sininen väri merkitsee sitä, että merkki ei ole "m".

```
T’ = etsittävä_sana_ei_ole_pan_eikä_onpanman_vaan_anpanman
T’[17..25] = _ole_pan
        H’ = anpanman
         i = 22
T’[20..28] = e_pan_ei
        H’ = anpanman
```

Tarkastellaan vielä tilanne, jossa vertailu epäonnistuu kohdassa $H'[m' - 2] = "a"$. Kuvio on nyt siis "~~a~~n": ei merkki "a" ja merkki "n". Kuvion vertailussa sallitaan myös kuvion suffiksin löytyminen hakusanan prefiksistä, jos kuvion suffiksi on yhtä pitkä kuin hakusanan prefiksi. Nyt huomataan, että kuvion etsintä jatkuu siihen asti, kunnes kuvion tyhjää suffiksia verrataan tyhjään prefiksiin (Esimerkki 8). Täten, jos tekstin ja hakusanan H' vertailu epäonnistuu hakusanan toiseksi viimeisessä merkissä $H'[m' - 2]$, optimaalisinta olisi siirtää tekstin tarkasteltavaa väliä 8 merkkiä eteenpäin (Esimerkki 9).

Esimerkki 8. Kuvio "~~a~~n" ei esiinny hakusanassa H' hyvän suffiksin säännön mukaisesti.

```
                H’ = anpanman
     H’[m’-3..m’-1] = ma
                   kuvio: an
    H’[m’-4..m’-2] = nm
                   kuvio: an
   H’[m’-5..m’-3] = an
                   kuvio: an
  H’[m’-6..m’-4] = pa
                   kuvio: an
 H’[m’-7..m’-5] = np
                   kuvio: an
H’[m’-8..m’-6] = an
                   kuvio: an
H’[m’-8..m’-7] = a
                   kuvio: n
H’[m’-8..m’-8] = ""
                   kuvio: ""
```

Esimerkki 9. Kun merkin $T'[i] = "o"$ vertailu epäonnistuu (merkattu punaisella), hyvän suffiksin säännön nojalla voidaan siirtyä väliltä $T'[25..32]$ kahdeksan merkkiä eteenpäin välille $T'[32..40]$, koska kuvio "~~a~~n" ei sovi mihinkään aikaisempaan väliin.

```
      T’ = etsittävä_sana_ei_ole_pan_eikä_onpanman_vaan_anpanman
                          T’[25..32] = _eikä_on
                                 H’ = anpanman
                                  i = 30
                          T’[32..40] = e_pan_ei
                                 H’ = anpanman
```

Nyt kuitenkin huonon merkin sääntö on määritelty siten, että kannattaa vertailla merkkiä $T'[i + delta_1(T'[i])]$, kun merkin $T'[i]$ vertailu epäonnistuu, mutta hyvän suffiksin sääntö siirtää tällä hetkellä väliä eikä indeksiä i. Muokataan hyvän suffiksin sääntöä vielä siten, että sekin kertoo, kuinka monta merkkiä kannattaa siirtyä eteenpäin. Muutos on yksinkertainen: lisätään jokaiselle siirtymälle arvo $m' - j - 1$, eli alkioiden $H'[j]$ ja $H'[m' - 1]$ etäisyys. Tällöin hyvän suffiksin sääntö kertoo, että kannattaa vertailla merkkiä $T'[i + delta_2(j)]$, kun merkkien $T'[i]$ ja $H'[j]$ vertailu epäonnistuu.

Muutoksen jälkeen edelliset esimerkit vieläkin toimivat. Välin kolmen merkin siirto muuttuu epäonnistuneen vertailukohdan i viiden merkin siirroksi (Esimerkki 7), ja välin kahdeksan merkin siirto muuttuu epäonnistuneen vertailukohdan i yhdeksän merkin siirroksi (Esimerkki 9).

Tästä saadaan Boyer-Mooren toinen sääntö: hyvän suffiksin sääntö (Boyer & Moore, 1977). Samoin kuin huonon merkin säännölle tehtiin, hyvän suffiksin sääntö voidaan laskea etukäteen jokaiselle hakusanan kohdalle hyvän suffiksin taulukkoon (Taulukko 4).

Taulukko 4. Hyvän suffiksin taulukko $delta_2$ hakusanalle $H' =$ "anpanman". Taulukko määrittää jokaiselle merkille, kuinka monta merkkiä kannattaa siirtyä eteenpäin merkkijonohaussa, kun vertailu epäonnistuu.

alkio	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
H'	"a"	"n"	"p"	"a"	"n"	"m"	"a"	"n"
$delta_2$	13	12	11	10	9	5	9	1

4.2.3 Algoritmin toiminta

Boyer-Moore on yksinkertainen, kun edellä mainitut kaksi taulukkoa on laskettu. Se toimii hyvin samankaltaisesti kuin paranneltu naiivi algoritmi, mutta kun vertailu epäonnistuu, valitaan säännöistä se, jolla siirrytään eniten merkkejä eteenpäin (Esimerkki 10) (Boyer & Moore, 1977).

Lasketaan Boyer-Moorellekin, kuinka monta vertailua keskimäärin tehdään tekstin merkkiä kohti. Kun tarkastellaan tekstiä T' (Esimerkki 10), voidaan laskea, että esimerkissä vertailu epäonnistuu 8 kertaa ja onnistuu 19 kertaa, eli vertaillaan yhteensä 27 merkkiä. Koska tekstin pituus m' on 53, Boyer-Moore vertailee keskimäärin noin 0,51 merkkiä. Boyer-Moore on siis paljon tehokkaampi kuin naiivi algoritmi Boyer-Mooren sääntöjen vuoksi. Tämä ei kuitenkaan riitä todistamaan, että Boyer-Moore on nopeampi, koska sääntöjen esikäsittelyä taulukoihin ei huomioida ja käytännössä on monia muita tekijöitä, jotka vaikuttavat tulokseen kuin pelkästään vertailujen määrä.

Esimerkki 10. Jokaisessa vaiheessa vertaillaan tekstiä T' ja hakusanaa H'. Epäonnistuneen vertailun (merkattu punaisella) kohtaan i lisätään huonon merkin tai hyvän suffiksin säännöstä suurempi arvo, eli $i_{next} = i + \max(delta_1(T'[i]), delta_2(i))$. Tällöin seuraava tarkasteltava väli on $T'[i_{next} - 7 .. i_{next} + 1]$.

```
          T’ = etsittävä_sana_ei_ole_pan_eikä_onpanman_vaan_anpanman
    T’[0..8] = etsittäv
          H’ = anpanman
           i = 7
delta1("v") = 8
delta2( 7 ) = 1

           T’[8..16] = ä_sana_e
                 H’ = anpanman
                  i = 15
        delta1("e") = 8
        delta2( 7 ) = 1

                  T’[16..24] = i_ole_pa
                         H’ = anpanman
                          i = 23
                delta1("a") = 1
                delta2( 7 ) = 1

                   T’[17..25] = _ole_pan
                          H’ = anpanman
                           i = 22
                 delta1("p") = 5
                 delta2( 5 ) = 5

                     T’[20..28] = e_pan_ei
                            H’ = anpanman
                             i = 27
                   delta1("i") = 8
                   delta2( 7 ) = 1

                            T’[28..36] = kä_onpan
                                   H’ = anpanman
                                    i = 33
                          delta1("p") = 5
                          delta2( 5 ) = 5

                               T’[31..39] = onpanman
                                      H’ = anpanman
                                       i = 31
                             delta1("o") = 8
                             delta2( 0 ) = 13

                                    T’[37..45] = an_vaan_
                                           H’ = anpanman
                                            i = 44
                                  delta1("_") = 8
                                  delta2( 7 ) = 1

                                           T’[37..45] = anpanman
                                                  H’ = anpanman
```

5 Nopeimmat merkkijonohakualgoritmit

Merkkijonohakuja on tutkittu paljon kirjallisuudessa, ja algoritmeja on luotu suuri määrä (Hakak ym., 2019). Algoritmien vertailu ei ole suoraviivaista, koska jokainen algoritmi toimii eri tavalla, kun tekstin sisältöä, tekstin pituutta ja hakusanan pituutta vaihdellaan. Tämän lisäksi algoritmin nopeuteen vaikuttaa tekniikoiden tehokkuus, algoritmin toteutus koodissa ja laitteisto.

Aydoğmuş ja Külekci (2019) ovat tehneet julkaisun, jossa verrataan kaksikymmentä nopeaa merkkijonohakualgoritmia. Vertailussa käytetään englannin kielistä dataa, joka koostuu Kuningas Jaakon Raamatusta ja The World Factbookista (Aydoğmuş & Külekci, 2019; Faro ym., 2016). Vertailussa mitataan merkkijonohauissa kuluva aika, kun hakusanan pituutta m vaihdellaan. Vertailussa tarkastellaan seitsemäätoista algoritmia, kun $m < 64$, ja yhdeksäätoista algoritmia, kun $m \geq 64$.

Aiemmista käydyistä algoritmeista naiivi algoritmi on vertailussa mukana. Sitä on kuitenkin optimoitu vielä enemmän muun muassa bittirinnakkaisuudella, jolla pystyy käsitellä 32 merkkiä yhdellä käskyllä (Tarhio ym., 2017). Monien optimointien jälkeen se on seitsemäntoista algoritmin joukosta toiseksi tai kolmanneksi nopein algoritmi, kun $m < 64$, ja ylsi nopeimmaksi algoritmiksi, kun $6 \leq m \leq 8$. (Aydoğmuş & Külekci, 2019) Se pärjää vertailussa siis erinomaisesti. Naiivin algoritmin etu on sen yksinkertaisuudessa, minkä vuoksi se on helppo optimoida ohjelmisto- ja laitteistotasolla (Tarhio ym., 2017).

Sitä vastoin Boyer-Moore ei ole käytännössä nopea. Koska vertailussa tutkitaan ainoastaan nopeita algoritmeja ja Boyer-Moore ei ole mukana mutta optimoitu naiivi algoritmi on (Aydoğmuş & Külekci, 2019), voidaan tulkita, että vertailun tekijät pitävät optimoitua naiivia algoritmia nopeampana kuin Boyer-Moorea. Yksi syy saattaa olla se, että Boyer-Moorea on vaikea tehostaa laitteistotasolla niin kuin naiivia algoritmia.

Palataan takaisin aikaisempaan julkaisuun (Aydoğmuş & Külekci, 2019) ja selvitetään nopeimmat algoritmit. Tuloksena saadaan, että seitsemäntoista algoritmin joukosta EPSMA (Exact Packed String Matching AVX2) on nopein, kun $m < 64$, ja yhdeksäntoista algoritmin joukosta SSEFA (Streaming SIMD Extensions Filter AVX2) on nopein, kun $m \geq 64$.

EPSMA on EPSM:n (Exact Packed String Matching) muokattu variaatio, joka käyttää AVX2-käskykantaa (Aydoğmuş & Külekci, 2019). EPSM-algoritmi (Faro & Külekci, 2014) koostuu kolmesta hakualgoritmista, joista nopein valitaan hakusanan pituuden perusteella ja jotka lisäksi saattavat toimia eri tavalla riippuen hakusanan pituudesta. EPSM voi toimia kuin suodattava algoritmi tai melkein kuin optimoitu naiivi algoritmi riippuen hakusanan pituudesta. Merkkijonohakualgoritmit ovat yleensä nopeimpia vain tietyn tekstin ja hakusanan pituuden alueilla. EPSM:n ideana on yhdistää eri hakualgoritmeja yhden algoritmin alle päästäkseen nopeimmaksi algoritmiksi mahdollisimman monella tekstin ja hakusanan pituuden alueella.

SSEFA on tehty SSEF:stä (Streaming SIMD Extensions Filter) ja siinäkin käytetään AVX2-käskykantaa (Aydoğmuş & Külekci, 2019). SSEF (Külekci, 2009) on suodattava algoritmi ja se toimii parhaiten pitkillä hakusanoilla. Mitä pidempi hakusana on, sitä lyhyempi SSEF:n ajoaika

on. Selvä huono puoli tässä on se, että SSEF ei toimi, kun hakusanan pituus m on liian pieni: 128-bittiselle rekisterille alaraja on $m \geq 32$, ja 256-bittiselle rekisterille $m \geq 64$.

EPSMA ja SSEFA käyttävät suodatusta (filter). Suodattavat hakualgoritmit ovat kaksiosaisia algoritmeja. Ensin ne käyttävät suodatinta, jolla pystyy vakioajassa tarkistamaan kohtia, joissa hakusana ei esiinny. Suodatin toteutetaan siten, että se on mahdollisimman nopea sillä kustannuksella, että se saattaa olla epätarkka. Kun suodatuksen jälkeen löydetään kohta, joka saattaa olla hakusana, siirrytään seuraavaan vaiheeseen ja verrataan tekstin ja hakusanan merkkejä. Siis jokaista tekstin merkkiä kohti käytetään vakioaikaista suodatusta ja, jos suodatuksen mukaan hakusana voi esiintyä tekstissä, verrataan lineaariaikaisesti tekstiä ja hakusanaa. (Faro & Külekci, 2014; Külekci, 2009)

Yksi mahdollinen suodatin on rullaava hajautus, jolla voi päivittää hajautusarvon vakioajassa (Karp & Rabin, 1987). Yksi rullaavan hajautuksen esimerkki on yhteenlasku merkkien ASCII-arvoista. Hajautusarvoa voi tällöin päivittää lisäämällä tekstin seuraavan merkin ASCII-arvo ja vähentämällä nykyisen kohdan ASCII-arvo. Kun tekstin hajautusarvo vastaa hakusanan hajautusarvoa, pitää vielä tarkistaa vertailemalla tekstin ja hakusanan merkkejä, koska hajautuksissa voi tapahtua törmäyksiä.

EPSMA ja SSEFA ovat olemassa olevien algoritmien muunnelmia, jotka käyttävät SIMD-arkkitehtuurin (single instruction, multiple data) AVX2-käskykantaa (Advanced Vector Extensions 2) (Aydoğmuş & Külekci, 2019). Etu tässä käskykannassa on se, että se tukee 256-bittistä rekisteriä. Koska merkkijonon yksi merkki vie 8 bittiä, 256-bittisellä rekisterillä voi käsitellä jopa $\frac{256}{8} = 32$ merkkiä samaan aikaan. Tämän tapaista tekniikkaa, jossa hyödynnetään rekisterin suurta kokoa, kutsutaan bittirinnakkaisuudeksi.

Nopeimmat merkkijonohakualgoritmit ovat siis EPSMA ja SSEFA, ja näiden jälkeen tulee optimoitu naiivi algoritmi. Yhteinen piirre näissä kaikissa on bittirinnakkaisuus, jolla pystyy käsittelemään enemmän dataa kerrallaan. Tämä vaikuttaa siltä, että käytännössä nopein algoritmi on sellainen, jota pystytään nopeuttamaan helposti. Tämän vuoksi nopeimmat algoritmit pyrkivät käyttämään yksinkertaisia tekniikoita, joita on helppo nopeuttaa suuremmilla rekistereillä.

6 Nopeimmat merkkijonohaut tietorakenteella

Aikaisemmin on esitelty algoritmeja, jotka eivät esikäsittele tekstiä, joten algoritmien asymptoottinen yläraja on $O(n)$, koska tekstin kaikki merkit täytyy lukea. Tässä luvussa käsitellään tietorakenteita, joilla pääsee parempiin nopeuksiin kuin $O(n)$. Haittapuolena tietorakenteissa on se, että niiden muodostamisessa menee aikaa, mutta koska työssäni sallitaan tekstin esikäsittely, muodostamisessa kuluvaa aikaa ei oteta huomioon.

6.1 Suffiksitietorakenteet

Suosituimmat tietorakenteet merkkijonojen hakuun ovat suffiksipuu ja suffiksitaulukko (Mailund, 2020). Ne säilyttävät tekstin kaikki suffiksit (Taulukko 2) jossain järjestyksessä. Suffiksit

ovat tärkeä tieto merkkijonohauille, kuten aikaisemmin Boyer-Mooren hyvän suffiksin säännössä on havaittu. Nyt suffiksit eivät ole ainoastaan osa hakua, vaan koko haku perustuu ainoastaan suffikseihin.

Etu suffiksitietorakenteissa on se, että haussa voidaan hakea tekstin jokaisesta kohdasta samaan aikaan. Tämä johtuu siitä, että tekstin kaikki suffiksit ovat toisilla sanoilla alijonoja, jotka alkavat tekstin jokaisesta kohdasta (ks. Taulukko 2). Täten haku tekstin kaikista kohdista samaan aikaan on mahdollista. Tällä tavalla voidaan päästä aikavaatimukseen, joka ei riipu tekstin koosta n.

Ensimmäinen suffiksitietorakenne on suffiksipuu, joka tallentaa tekstin kaikki suffiksit tiettyyn järjestykseen puuhun. Puun kaariin on tallennettu tietoa tekstistä siten, että polun kaaret muodostavat tekstin yhden suffiksin, kun juuresta siirrytään jotakin polkua pitkin puun lehteen asti. Voidaan siis selvittää, onko hakusana tekstissä vertaamalla hakusanaa tekstin suffiksiin, mikä onnistuu ajassa $O(m)$. (Weiner, 1973)

Toinen suffiksitietorakenne on suffiksitaulukko, joka on alun perin kehitetty suffiksipuusta yrityksenä vähentää muistin käyttöä. Suffiksitaulukko on tekstistä muodostettu aakkosjärjestykseen järjestetty taulukko suffikseista. Haittapuolena merkkijonohaku suffiksitaulukosta ei ole yhtä suoraviivaista kuin suffiksipuusta, eikä suffiksitaulukon alkuperäisessä julkaisussa (Manber & Myers, 1993) ole hakualgoritmia, joka toimisi ajassa $O(m)$. (Manber & Myers, 1993) Myöhemmin on kuitenkin keksitty algoritmi, jolla pystyy hakemaan merkkijonoja ajassa $O(m)$ (Li & Durbin, 2009).

Kun vertaillaan molempia tietorakenteita, huomataan, että ne ovat ominaisuuksiltaan samankaltaisia. Niillä on sama muodostamisaika $O(n)$ (Mailund, 2020; Nong ym., 2011), molemmat pystyvät ratkaisemaan samoja ongelmia (Abouelhoda ym., 2004) ja molempien asymptoottinen yläraja merkkijonohaulle on $O(m)$ (Li & Durbin, 2009; Weiner, 1973). Käytännössä ratkaiseva ero tulee muistin käytössä. Vaikka tilavaatimus on molemmille $O(n)$, suffiksitaulukoiden muistin käyttö on kolmasosasta viidesosaan verrattuna suffiksipuuhun (Manber & Myers, 1993). Suffiksitaulukko säästää siis paljon enemmän muistia verrattuna suffiksipuuhun. Tämän vuoksi otan työssäni tarkempaan tarkasteluun suffiksitaulukon.

6.2 Merkkijonohaku suffiksitaulukosta

Suffiksitaulukko tekstistä T muodostetaan tallentamalla sinne kaikki suffiksit (Taulukko 2) aakkosjärjestyksessä. Aakkosjärjestyksessä tulee huomioida se, että merkkijonon pääte määritellään aakkosjärjestyksessä kaikista pienimmäksi. Suffiksien tiedot ei tallenneta merkkijonoina suffiksitaulukkoon vaan indekseinä. Näin saadaan muodostettua suffiksitaulukko SA (suffix array) (Taulukko 5).

Taulukko 5. Merkkijonon $T =$ "AABAADAAAAD" suffikseista muodostettu suffiksitaulukko SA, kun merkkijonon päätettä merkataan merkillä "$".

SA[0] = 11	T[SA[0]..n] = $
SA[1] = 6	T[SA[1]..n] = AAAAD$
SA[2] = 7	T[SA[2]..n] = AAAD$
SA[3] = 0	T[SA[3]..n] = AABAADAAAAD$
SA[4] = 8	T[SA[4]..n] = AAD$
SA[5] = 3	T[SA[5]..n] = AADAAAAD$
SA[6] = 1	T[SA[6]..n] = ABAADAAAAD$
SA[7] = 9	T[SA[7]..n] = AD$
SA[8] = 4	T[SA[8]..n] = ADAAAAD$
SA[9] = 2	T[SA[9]..n] = BAADAAAAD$
SA[10] = 10	T[SA[10]..n] = D$
SA[11] = 5	T[SA[11]..n] = DAAAAD$

Naiivi tapa hakea suffiksitaulukosta olisi tehdä puolitushaku hakusanalla. Jos miettii nopeasti, voi erehtyä, että tällöin aikavaatimus olisi $O(\log n)$, koska puolitushaussa tarkastellaan $\log n$ väliä. Kuitenkin pitää huomioida myös se, että jokaisessa vaiheessa verrataan m-pituista hakusanaa suffiksitaulukon suffiksiin, joten aikavaatimus on oikeastaan $O(m \log n)$, mikä on hidas.

Suffiksitaulukon alkuperäisessä julkaisussa (Manber & Myers, 1993) on keksitty parempi algoritmi. Tämä menetelmä käyttää myös puolitushakua. Kuitenkin Manber ja Myers (1993) osoittavat, että vertailut voidaan optimoida vakioaikaan soveltamalla pisintä yhteistä prefiksiä ja tarkastelemalla hakusanan jokainen merkki vain kerran. Tällöin aikavaatimus on $O(m + \log n)$. Tämä ei kuitenkaan ole riittävän nopea, koska suffiksitaulukon koko vaikuttaa hakunopeuteen. Nykypäivänä dataa voi olla niin suuri määrä, että haun riippuvuus tekstin koosta n voi olla riittävä syy välttää käyttämästä algoritmia. Tällöin suffiksipuun aikavaatimus, $O(m)$, vaikuttaa paremmalta vaihtoehdolta, vaikka siinä menisi monin kerroin enemmän muistia.

Toinen tapa on käyttää *tehostettua suffiksitaulukkoa* (enhanced suffix array). Tehostettu suffiksitaulukko toteuttaa suffiksipuun kaikki toiminnot samalla tai vähän huonommalla aikavaatimuksella, kun merkistön koko on vakio. Merkkijonohaussa aikavaatimus on $O(m + z)$, missä z on hakuosumien määrä. (Abouelhoda ym., 2004) Tämä on nopea, mutta suuressa määrässä datassa voi olla paljon hakuosumia, mikä voi hidastaa paljon hakunopeutta etenkin, jos hakusana on yleinen, kuten välilyönti " ". Käytännössä jokainen hakuosuma käsitellään jollakin tavalla tai hakuosumia rajoitetaan tiettyyn määrään. Tällöin z ei vaikuttaisi käytännön nopeuteen merkittävästi. Tutkimuskysymykseeni kuitenkin kuuluu kaikkien kohtien löytäminen, joten z täytyy ottaa huomioon. Lisäksi tämä merkkijonohaku olettaa, että merkistön koko on vakio. Esimerkiksi kiinan tai japanin kielessä tämä vakio voi osoittautua suureksi, mikä hidastaa merkkijonohakua. Tämäkään ei ole siis sopiva hakualgoritmi.

Nopein tapa hakea suffiksitaulukosta tulee oikeastaan tiedonpakkausmenetelmästä, vaikka tiedonpakkaus ja merkkijonohaku ovat kaksi eri aihetta. Burrows-Wheeler-muunnos on lohkopakkaukseen perustuva tiedonpakkausalgoritmi, ja se voidaan laskea lineaarisessa ajassa, kun suf-

fiksitaulukko on muodostettu (Burrows & Wheeler, 1994). Burrows-Wheeler-muunnoksella pystyy tehokkaasti etsimään merkkijonoja ajassa $O(m)$, kun merkkijonoa etsitään käänteisesti viimeisestä merkistä ensimmäiseen merkkiin (Li & Durbin, 2009). Tässä näkyy tekniikka, joka on käsitelty optimoidun naiivin algoritmin ja Boyer-Moore-algoritmin yhteydessä: vertailujärjestyksen muuttaminen. Tämä mahdollistaa suffiksipuun kaltaisen aikavaatimuksen $O(m)$, jota paremmaksi eivät merkkijonohaut pääse, koska hakusanan kaikki m merkkiä pitää lukea. Burrows-Wheeler-muunnokseen perustuva merkkijonohaku on siis tehokkain algoritmi suffiksitaulukoilla.

7 Yhteenveto

Tutkimuskysymykseni ovat siis

1. *mitä tehokkaita tekniikoita merkkijonohaussa on*
2. *mitkä ovat nopeimmat hakualgoritmit, jotka löytävät kaikki hakuosumat tarkalla haulla, kun tekstiä ei saa esikäsitellä*
3. *mitkä ovat nopeimmat hakualgoritmit, jotka löytävät kaikki hakuosumat tarkalla haulla, kun tekstiä saa esikäsitellä.*

Työssäni tutkittiin tekniikoita tarkastelemalla naiivia algoritmia, Boyer-Moorea, EPSMA:ta, SSEFA:ta ja suffiksitaulukon hakualgoritmeja. Kun paranneltiin naiivia algoritmia *vertailujärjestystä muuttamalla*, saatiin tehokkaampi algoritmi. Tästä siirryttiin Boyer-Moore-algoritmiin, joka perustuu edelliseen tekniikkaan, vertailujärjestyksen muuttamiseen. Havaintojen kautta saatiin Boyer-Mooren säännöt: *huonon merkin sääntö* ja *hyvän suffiksin sääntö*. Näillä säännöillä *hakusanoja esikäsiteltiin* ja laskettiin taulukkoon talteen, jotta niitä ei tarvitsisi laskea jatkuvasti uudestaan. EPSMA on kokoelma muista *hakualgoritmeista, joista paras valitaan* merkkijonohaun parametrien perusteella. SSEFA on *suodattava* algoritmi, jotka ovat kaksiosaisia: ensin löydetään tekstistä mahdollinen kohta, jossa hakusana saattaa esiintyä, sitten tarkistetaan tämä kohta vertaamalla tekstin ja hakusanan merkkejä. *Bittirinnakkaisuudella* pystytään SIMD-arkkitehtuurilla käsittelemään useita merkkejä yhdellä käskyllä. *Suffiksitietorakenteesta* huomattiin, että kaikkien suffiksien tallentaminen, mahdollistaa merkkijonohaun kohdistuvan jokaiseen kohtaan samaan aikaan, mikä poistaa merkkijonohaun riippuvuuden tekstin koosta n. Tekniikoiksi sain siis vertailujärjestyksen muuttaminen, huonon merkin sääntö, hyvän suffiksin sääntö, esikäsittely, parhaan hakualgoritmin valitseminen, suodatus, bittirinnakkaisuus ja suffiksitietorakenne.

Tämän lisäksi kävin naiivin algoritmin ja Boyer-Mooren läpi. Läpikäynti oli sillä tarkkuudella, että lukija ymmärtää algoritmien tekniikoita ja pystyisi ehkä toteuttamaan algoritmit itse. Johdatin Boyer-Mooren havainnot sääntöihin omalla tavallani ja esitin esimerkkejä Boyer-Mooren sääntöjen ymmärtämiseen.

Kuitenkin työni laajuuden vuoksi tekniikoista jäi väkisin muita tärkeitä tekniikoita käsittelemättä, ja tämän takia työni ei sisällä kaikkia tärkeitä tekniikoita, koska tekniikoita merkkijonohaussa on paljon. Esimerkiksi Knuth-Morris-Pratt-algoritmin automaatit (Knuth ym., 1977) jäivät

käsittelemättä. Työni kuitenkin käsittelee riittävästi tärkeitä ja oleellisia tekniikoita, joten totean, että tämä on toimiva kompromissi.

Selvitin kirjallisuudesta nopeimmat hakualgoritmit, kun tekstiä ei saa esikäsitellä. Nopeusvertailuun (Aydoğmuş & Külekci, 2019) perustuen nopeimmaksi merkkijonohakualgoritmeiksi esitin EPSMA-algoritmin, kun hakusanan pituudelle pätee $m < 64$, ja SSEFA-algoritmin, kun $m \geq 64$. Nämä kaksi algoritmia tulivat nopeimmaksi, koska näitä on helppo optimoida laitteistotasolla bittirinnakkaisuudella.

Kuitenkin työssäni käytin nopeimpaan hakualgoritmiin vain yhtä lähdettä (Aydoğmuş & Külekci, 2019), koska EPSMA ja SSEFA ovat uusia algoritmeja eikä niitä esitetä muissa julkaisuissa. On kuitenkin vanhempia julkaisuja, jossa vertaillaan muun muassa EPSM:ä ja SSEF:ä (Faro & Külekci, 2014), mutta nämä eivät käytä AVX2-käskykantaa. Koska työssäni yritän selvittää nopeimmat algoritmit, on pakko käyttää tätä yhtä lähdettä.

Kun tekstiä saa esikäsitellä, nopean aikavaatimuksen $O(m)$ merkkijonohakuun päästiin Burrows-Wheeler-muunnoksella, joka voidaan laskea helposti suffiksitaulukosta (Li & Durbin, 2009). Etu tekstin esikäsittelyssä ja suffiksitietorakenteissa oli se, että haku ei enää riipu tekstin pituudesta n.

Työni vastasi tutkimuskysymykseni kaikkiin osiin, joten nyt osataan toimia johdannossa mainitsemassa tilanteessa, jossa *luonnollisen kielen tekstiä on kerätty runsaasti eikä se muutu merkkijonohakujen aikana* ja *datasta pitäisi etsiä hakuosumia suurella määrällä eri hakusanoilla mahdollisimman nopeasti*.

Lähdeluettelo

Abouelhoda, M. I., Kurtz, S., & Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, *2*(1), 53–86. https://doi.org/10.1016/S1570-8667(03)00065-0

Aydoğmuş, M. A., & Külekci, M. O. (2019). Optimizing Packed String Matching on AVX2 Platform. Teoksessa H. Senger, O. Marques, R. Garcia, T. Pinheiro de Brito, R. Iope, S. Stanzani, & V. Gil-Costa (Toim.), *High Performance Computing for Computational Science – VECPAR 2018* (ss. 45–61). Springer International Publishing. https://doi.org/10.1007/978-3-030-15996-2_4

Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, *20*(10), 762–772. https://doi.org/10.1145/359842.359859

Burrows, M., & Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. *SRC Research Report, 124*.

Faro, S., & Külekci, M. O. (2014). Fast and flexible packed string matching. *Journal of Discrete Algorithms*, *28*, 61–72. https://doi.org/10.1016/j.jda.2014.07.003

Faro, S., Lecroq, T., Borzì, S., Mauro, S. D., & Maggio, A. (2016). *SMART (String Matching Algorithm Research Tool)*. https://github.com/smart-tool/smart

Hakak, S. I., Kamsin, A., Shivakumara, P., Gilkar, G. A., Khan, W. Z., & Imran, M. (2019). Exact String Matching Algorithms: Survey, Issues, and Future Research Directions. *IEEE Access*, *7*, 69614–69637. https://doi.org/10.1109/ACCESS.2019.2914071

Hume, A., & Sunday, D. (1991). Fast string searching. *Software: Practice and Experience*, *21*(11), 1221–1248. https://doi.org/10.1002/spe.4380211105

ISO. (2017). *ISO/IEC 14882:2017* (5. p.). International Organization for Standardization. https://www.iso.org/standard/68564.html

Karp, R. M., & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, *31*(2), 249–260. https://doi.org/10.1147/rd.312.0249

Knuth, D. E., Morris, Jr., J. H., & Pratt, V. R. (1977). Fast Pattern Matching in Strings. *SIAM Journal on Computing*, *6*(2), 323–350. https://doi.org/10.1137/0206024

Külekci, M. O. (2009). Filter Based Fast Matching of Long Patterns by Using SIMD Instructions. *Proceedings of the Prague Stringology Conference 2009*, 118–128.

Li, H., & Durbin, R. (2009). Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, *25*(14), 1754–1760. https://doi.org/10.1093/bioinformatics/btp324

Mailund, T. (2020). *String Algorithms in C: Efficient Text Representation and Search*. Apress. https://doi.org/10.1007/978-1-4842-5920-7

Manber, U., & Myers, G. (1993). Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, *22*(5), 935–948. https://doi.org/10.1137/0222058

Nong, G., Zhang, S., & Chan, W. H. (2011). Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Transactions on Computers*, *60*(10), 1471–1484. https://doi.org/10.1109/TC.2010.188

Tarhio, J., Holub, J., & Giaquinta, E. (2017). Technology beats algorithms (in exact string matching). *Software: Practice and Experience*, *47*(12), 1877–1885. https://doi.org/10.1002/spe.2511

Weiner, P. (1973). Linear pattern matching algorithms. *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, 1–11. https://doi.org/10.1109/SWAT.1973.13