View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Helsingin yliopiston digitaalinen arkisto

Tietojenkäsittelytiede **31** Marraskuu 2010 sivut 57–69
Toimittaja: Antti Valmari
© kirjoittaja(t)

Paikallinen laskettavuus

Jukka Suomela
Helsingin yliopisto
Tietotekniikan tutkimuslaitos HIIT
jukka.suomela@cs.helsinki.fi

Tuhannen nimen lista on hitaampi aakkostaa kuin sadan nimen lista. Yleensäkin kun syötteen koko kasvaa, myös tyypillisen algoritmin ajoaika kasvaa. Vain kaikkein triviaaleimmat ongelmat pystytään ratkaisemaan vakioajassa riippumatta syötteen koosta.

Hajautettujen algoritmien osalta tilanne ei kuitenkaan ole näin suoraviivainen: on olemassa useita ongelmia, joiden ratkaisemiseen on olemassa vakioajassa toimiva hajautettu algoritmi.

1 Verkko-ongelmia

Hajautettujen algoritmien perusidea on täysin erilainen kuin tutummissa keskitetyissä algoritmeissa ja rinnakkaisalgoritmeissa. Yhteistä kaikille algoritmeille on kuitenkin se, että niitä käytetään *laskennallisten ongelmien* ratkaisemiseen.

Hajautettujen algoritmien näkökulmasta tärkeimpiä laskennallisia ongelmia ovat *verkko-ongelmat*. Näissä syötteenä on jokin verkko $\mathcal{G} = (V,E)$, joka koostuu solmuista $v \in V$ ja niitä yhdistävistä kaarista $e \in E$; ks. kuvaa 1a.

Vaikka viittaan tässä matemaattiseen verkkoteorian käsitteeseen (*graph*), voidaan aivan yhtä hyvin ajatella esimerkiksi tietoliikenneverkkoa (*network*). Usein tämän alan sovelluksissa verkon solmut vastaavat tietokoneita tai muita tietoverkkoon kytkettyjä laitteita ja kaaret vastaavat laitteiden välisiä tiedonsiirtoyhteyksiä. Jos verkossa on solmujen $u \in V$ ja $v \in V$ välillä kaari $\{u,v\} \in E$, niin laitteet u ja v voivat vaihtaa keskenään viestejä ilman muiden laitteiden apua.

Seuraavassa on kaksi esimerkkiä tunnetuista verkko-ongelmista; näitä on havainnollistettu kuvassa 1.

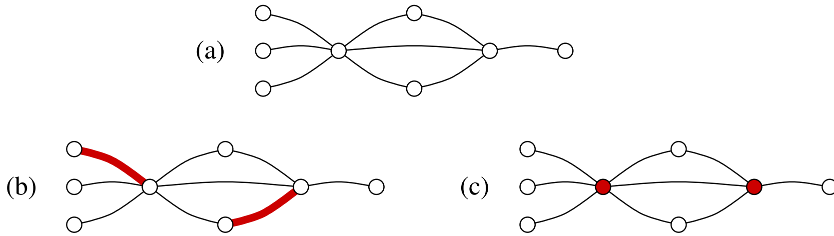
- **Pariutus** on kaarien joukko $M \subseteq E$, joka täyttää seuraavan ehdon: jokaiseen verkon solmuun liittyy enintään yksi joukon M kaari.

  Esimerkki: Yksi verkkolaite $v \in V$ voi osallistua kerralla vain yhteen tiedonsiirtoon $e \in E$. Pariutus M kuvaa tiedonsiirtoja, jotka voivat olla käynnissä samanaikaisesti toisiaan häiritsemättä.

- **Solmupeite** on solmujen joukko $C \subseteq V$, joka *peittää* kaikki kaaret: jokaisen kaaren päätepisteistä ainakin toinen on joukossa C.

  Esimerkki: Riittää, että joukon C laitteet ajavat ohjelmaa, joka valvoo laitteisiin liittyviä tiedonsiirtoyhteyksiä. Jos kaikki joukon C laitteet ovat tyytyväisiä, kaikki verkon yhteydet ovat kunnossa.

Näille ongelmille on helppoa löytää jokin ratkaisu — esimerkiksi tyhjä kaarijoukko $M = \emptyset$ on pariutus ja kaikkien solmujen joukko $C = V$ on solmupeite. Sovelluksissa halutaan kuitenkin tyypillisesti löytää



Kuva 1: (a) Verkko: ympyrät kuvaavat solmuja ja niiden väliset viivat kaaria. (b) Suurin pariutus. (c) Pienin solmupeite.

pariutus, jonka koko on mahdollisimman suuri, ja vastaavasti solmupeite, jonka koko on mahdollisimman pieni.

Tulemme tässä kirjoituksessa keskittymään erityisesti hajautettuihin algoritmeihin, jotka etsivät pieniä solmupeitteitä; luvussa 3 näemme, mitä tekemistä suurilla pariutuksilla on pienien solmupeitteiden kanssa.

2 Hajautetut algoritmit

Mutta mitä täsmälleen ottaen tarkoittaa, että meillä on olemassa hajautettu algoritmi solmupeitteen etsimiseen? Tämän ymmärtämiseksi on hyvä verrata tilannetta perinteisiin keskitettyihin algoritmeihin ja rinnakkaisalgoritmeihin.

2.1 Keskitetyt algoritmit ja rinnakkaisalgoritmit

Keskitetyt algoritmit ovat kaikille tietojenkäsittelytieteen opiskelijoille tuttu perustapaus. Esimerkiksi keskitetty algoritmi $\mathcal{A}$ solmupeitteen etsimiseen tarkoittaa seuraavaa:

- Meillä on käytettävissä yksi tietokone, joka suorittaa algoritmia $\mathcal{A}$.
- Voimme antaa tietokoneelle syötteenä minkä tahansa verkon $\mathcal{G}$. Täsmällisemmin annamme *merkkijonon*, joka kuvaa verkon rakenteen. Merkkijonossa on esimerkiksi lueteltuna verkon kaikki solmut ja niiden väliset yhteydet.
- Tietokone tuottaa tulosteena solmupeitteen C. Jälleen tuloste on merkkijono, jossa on esimerkiksi lueteltuna kaikki joukon C solmut.

Rinnakkaisalgoritmien oleellinen ero keskitettyihin algoritmeihin verrattuna on se, että laskentaa pyritään nopeuttamaan hyödyntämällä useaa suoritinta. Edelleen koko syöte on annettuna tietokoneelle yhdessä paikassa merkkijonona ja koko tuloste tuotetaan yhteen paikkaan; ainoastaan laskennan aikana tietoa ja sen käsittelyä on hajautettu erillisille suoritinyksiköille.

Tilanne on sama yleisemminkin perinteisessä *suurteholaskennassa*, jossa hyödynnetään esimerkiksi klusteria, joka muodostuu useista tietoverkolla yhteen kytketyistä tietokoneista. Edelleen klusterin käyttäjä toimittaa koko syötteen merkkijonona yhteen paikkaan ja algoritmi puolestaan tuottaa koko tuloksen merkkijonona yhteen paikkaan.

2.2 Hajautetut algoritmit

Hajautetut algoritmit eroavat edellisistä täysin. Hajautettu algoritmi $\mathcal{A}$ solmupeitteen etsimiseen tarkoittaa seuraavaa:

- Voimme ottaa mielivaltaisen määrän tietokoneita ja asentaa kuhunkin saman algoritmin $\mathcal{A}$.
- Tämän jälkeen voimme muodostaa tietokoneista mielivaltaisen verkon $\mathcal{G}$. Ver-

kon $\mathcal{G}$ solmuja ovat tietokoneet ja kaaria ovat näiden väliset tietoliikenneyhteydet. Tietokoneille ei erikseen anneta mitään syötettä.

- Tämän jälkeen käynnistämme tietokoneet. Algoritmin $\mathcal{A}$ ohjaamina tietokoneet alkavat viestiä keskenään.
- Lopulta kaikki tietokoneet pysähtyvät ja tuottavat tulosteen. Kunkin tietokoneen osalta tuloste on 0 tai 1. Solmut, jotka tulostavat ykkösen, muodostavat solmupeitteen C verkossa verkossa $\mathcal{G}$.

Hyvin oleellinen ero keskitettyyn laskentaan, rinnakkaislaskentaan tai esimerkiksi klustereilla tehtävään suurteholaskentaan on se, että syötteenä olevaa verkkoa $\mathcal{G}$ ei erikseen koodata merkkijonoksi, vaan syötteenä oleva verkko $\mathcal{G}$ on sama kuin kulloinkin tarkasteltavan tietoliikenneverkon rakenne. Vastaavasti tulosta ei koota yhdeksi merkkijonoksi, vaan riittää, että jokainen solmu tuottaa pelkästään oman osuutensa ratkaisusta.

Hajautetussa algoritmissa siis *syöte annetaan hajautetusti ja ratkaisu palautetaan hajautetusti*. Aluksi jokainen verkon solmu (tietokone) tietää vain oman yksilöllisen tunnisteensa, ja tietokoneet joutuvat vaihtamaan keskenään viestejä, jotta ne saavat selville tarkempaa tietoa verkon $\mathcal{G}$ rakenteesta. Vastaavasti algoritmin päätyttyä riittää, että jokainen solmu osaa tulostaa oman osuutensa ratkaisusta.

Jos verkko-ongelmia ratkaistaan esimerkiksi klustereilla, meillä on *kaksi eri verkkoa*: ongelman syötteenä oleva verkko $\mathcal{G}$ ja klusterissa olevien tietokoneiden välisiä tietoliikenneyhteyksiä kuvaava verkko $\mathcal{H}$. Näissä tyypillisesti verkko $\mathcal{H}$ on tunnettu ja pysyy muuttumattomana; syöte $\mathcal{G}$ sen sijaan vaihtelee tapauksesta toiseen. Hajautettujen algoritmien kohdalla meillä on vain *yksi verkko* $\mathcal{G}$, joka kuvaa siis sekä syötteen että laskentaa tekevän järjestelmän rakenteen, ja tämä verkko vaihtelee tapauksesta toiseen.

Hajautetun algoritmin tulee siis tulla toimeen, vaikka laskentaa suorittavan järjestelmän rakenne on tuntematon. Tämä on luonnollisesti tyypillinen tilanne missä tahansa laajassa tietokoneverkossa; esimerkiksi Internetissä käytettävien ohjelmistojen tulee toimia oikein vaikka verkon koko rakenne on tuntematon.

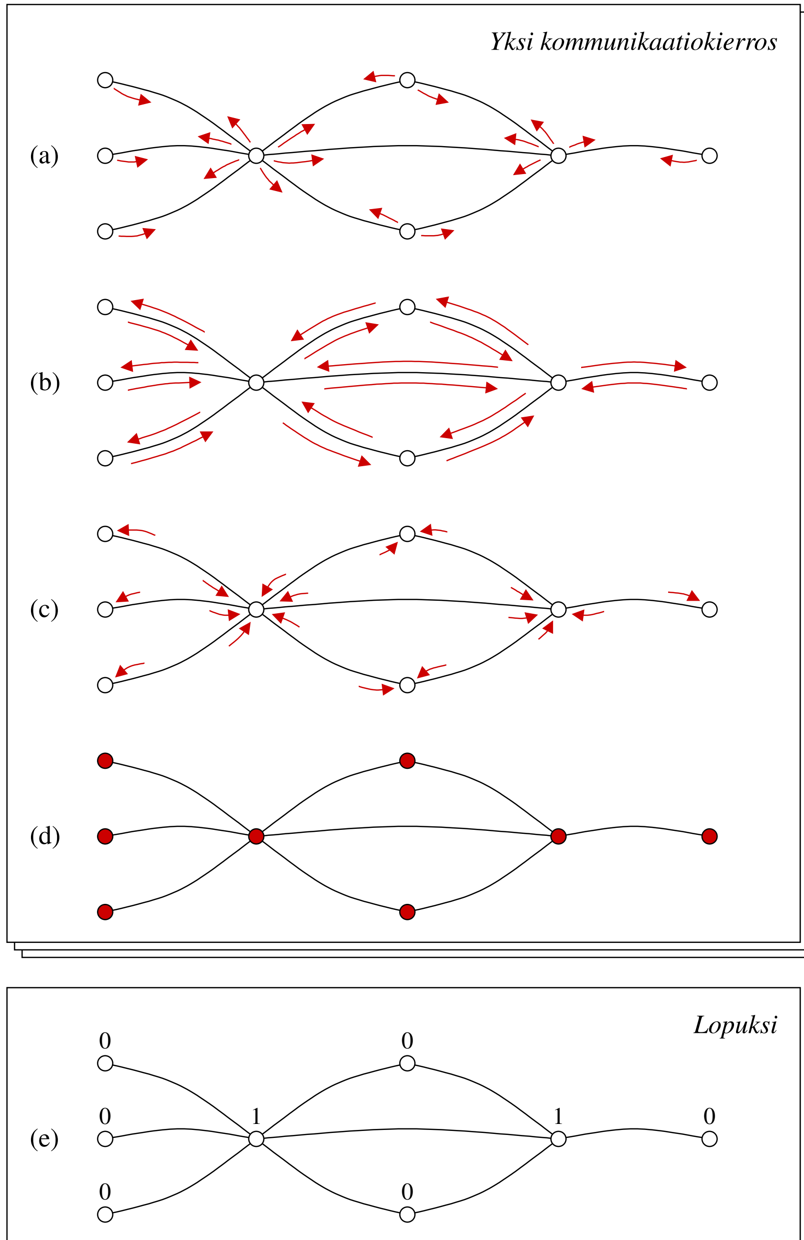
2.3 Resurssina kommunikaatio

Keskitetyissä algoritmeissa algoritmin suoritusajalla tarkoitetaan tietokoneen tekemien *laskenta-askeleiden* lukumäärää. Laskenta-askeleen täsmällinen määritelmä riippuu käytetystä laskennan mallista, mutta tyypillisesti yksittäisiä laskenta-askeleita ovat esimerkiksi muistiin kirjoittaminen, muistista lukeminen, vertailu ja yksittäinen peruslaskutoimitus.

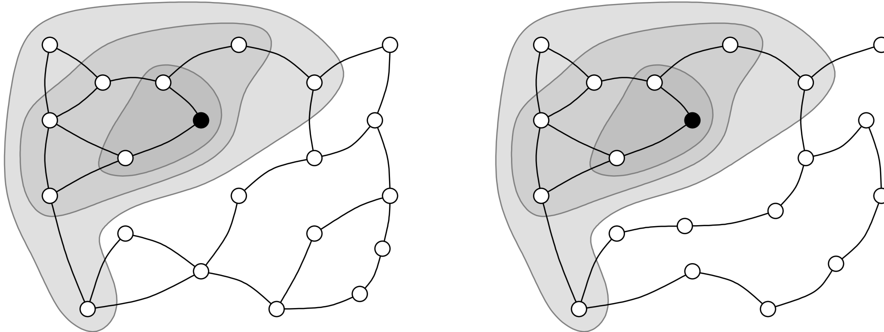
Hajautetuissa algoritmeissa keskeinen resurssi ei olekaan laskenta vaan tiedonsiirto. Ajatuksena on, että verkossa olevien tietokoneiden suorittama laskenta on suhteessa hyvin nopeaa verrattuna tietoliikenneverkossa tapahtuvaan tiedonsiirtoon. Nykyaikainen tietokone suorittaa peruslaskutoimituksia nanosekunneissa, mutta tietoliikenneyhteyksien latenssit ovat usein jopa millisekuntien suuruusluokkaa. Hajautetun algoritmin tehokkuuden kannalta onkin keskeistä, että algoritmi joutuu odottamaan verkossa tapahtuvaa tiedonsiirtoa mahdollisimman vähän.

Tämän vuoksi hajautetun algoritmin suoritusajalla tarkoitetaan algoritmin suorittamien *kommunikaatiokierrosten* lukumäärää. Yhdessä kommunikaatiokierroksessa verkon kaikki solmut suorittavat seuraavat toiminnot (tahdistettuna samanaikaisesti ja rinnakkain); ks. kuvaa 2:

(a) Solmu lähettää viestin kullekin naapurilleen.



Kuva 2: Hajautetun algoritmin suoritus koostuu kommunikaatiokierroksista. Kunkin kierroksen aikana (a) kaikki solmut lähettävät yhden viestin kullekin naapurilleen, (b) viestit etenevät verkon kaaria pitkin vastaanottajilleen, (c) solmut vastaanottavat yhden viestin kultakin naapuriltaan, ja (d) solmut suorittavat paikallista laskentaa ja päivittävät omaa tilaansa. Lopuksi (e) algoritmi päättyy ja kukin solmu tuottaa oman osansa ratkaisusta. Tässä esimerkissä algoritmi tuotti solmupeitteen C; tuloste 1 tarkoittaa, että solmu on mukana joukossa C.



Kuva 3: Harmaalla korostetut alueet ovat mustalla merkityn solmun 1-säteinen, 2-säteinen ja 3-säteinen paikallinen ympäristö. Jos hajautetun algoritmin ajoaika on kolme kierrosta, mustalla merkityn solmun tuloste riippuu ainoastaan solmun 3-säteisestä ympäristöstä. Solmu tuottaa siis saman tulosteen vasemmanpuoleisessa ja oikeanpuoleisessa esimerkkiverkossa, koska verkon rakenne on sama solmun 3-säteisen ympäristön sisällä. Verkot ovat kauempana erilaisia, mutta kolmessa kommunikaatiokierroksessa tieto näistä eroavuuksista ei ehdi mustaan solmuun asti.

(b) Viestit etenevät verkossa kaaria pitkin vastaanottajilleen.

(c) Solmu vastaanottaa viestin kultakin naapuriltaan.

(d) Solmu päivittää omaa tilaansa vastaanottamiensa viestien perusteella.

Lopulta verkon solmut tuottavat oman osansa tulosteesta ja pysähtyvät. Algoritmin suoritusaika kertoo, kuinka monen kommunikaatiokierroksen jälkeen verkon kaikki solmut ovat pysähtyneet. Nopea hajautettu algoritmi on siis sellainen, jonka suoritus tarvitsee mahdollisimman pienen määrän kommunikaatiokierroksia.

Keskitettyjen algoritmien osalta suoritusaikaa tarkastellaan yleensä vain asymptoottisessa mielessä. Ajoaikojen vakiokertoimet riippuvat käytetyn mallin yksityiskohdista, ja niinpä vakiot yleensä piilotetaan O-notaatioon. Sanotaan esimerkiksi, että algoritmin suoritusaika on $O(n)$ eli ajoaika kasvaa enintään lineaarisesti syötteen koon kasvaessa. Hajautettujen algoritmien osalta on kuitenkin aivan mielekästä tarkastella täsmällistä ajoaikaa: voidaan esimerkiksi todeta, että algoritmin ajoaika on täsmälleen 13 kierrosta.

2.4 Paikalliset ympäristöt

Hajautetun algoritmin ajoaika liittyy hyvin läheisesti solmujen *paikallisiin ympäristöihin*. Solmun T-säteiseen ympäristöön kuuluvat kaikki verkon solmut, jotka ovat lyhintä polkua pitkin enintään T kaaren päässä lähtösolmusta; näitä on havainnollistettu kuvassa 3.

Jos hajautetun algoritmin ajoaika on T, tietyn solmun v tuottama tuloste riippuu ainoastaan v:n T-säteisestä ympäristöstä. Kaikki tämän ympäristön ulkopuolella olevat solmut ovat yksinkertaisesti liian kaukana — T kierrosta ei riitä siihen, että näihin solmuihin liittyvä tieto ehtisi vaikuttaa solmun v tekemään päätökseen.

Toisaalta ajassa T solmut pystyvät keräämään kaiken tiedon T-säteisistä ympäristöistään. Tähän riittää varsin yksinkertainen algoritmi:

- 1. kierroksella kaikki solmut välittävät oman tunnisteensa naapureille. Näin yhden kierroksen jälkeen kaikki solmut tietävät 1-säteiset ympäristönsä.
- 2. kierroksella kaikki solmut välittävät 1-säteiset ympäristönsä naapureilleen.

Näin kahden kierroksen jälkeen kaikki solmut saavat selville 2-säteiset ympäristönsä.

- Jne. . .

Toistamalla tätä T kierrosta kaikki solmut saavat selville T-säteiset ympäristönsä.

Hajautettu algoritmi, jonka ajoaika on T, voidaan siis määritellä aivan yhtä hyvin kummalla tahansa seuraavista tavoista:

- Solmut vaihtavat keskenään viestejä T kommunikaatiokierroksen ajan ja julistavat tämän jälkeen tulosteensa.
- Solmun v tuottama tuloste on funktio v:n T-säteisestä ympäristöstä.

2.5 Paikalliset algoritmit

Edellä nähtiin, että nopea hajautettu algoritmi on sama asia kuin hajautettu algoritmi, jossa solmut tuottavat tulosteensa pelkästään pienisäteisen paikallisen ympäristön perusteella. Paikallisten algoritmien tutkimuksessa keskitytään tässä äärimmäiseen tapaukseen.

Paikallinen algoritmi [7, 9] on hajautettu algoritmi, jonka ajoaika T on *vakio, joka ei riipu verkon solmujen määrästä*. Toisin sanoen paikallisessa algoritmissa verkon solmut joutuvat tekemään päätöksensä pelkän vakiosäteisen lähiympäristönsä perusteella. Tällaiset algoritmit ovat erittäin skaalautuvia: jos paikallinen algoritmi toimii tehokkaasti pienessä verkossa, se toimii aivan yhtä tehokkaasti myös mielivaltaisen suuressa verkossa.

Sinällään vakioajassa toimivat hajautetut algoritmit eivät ole täysin mahdoton ajatus. Kun syötteenä olevan verkon $\mathcal{G}$ koko kasvaa, verkossa olevien tietokoneiden määrä kasvaa samaa tahtia; suuremmassa verkossa on myös enemmän rinnakkaista laskentakapasiteettia.

Haasteena on kuitenkin se, että vakioaikaisen algoritmin solmut ovat pahasti likinäköisiä, kuten kuvassa 3 havainnollistettiin. Onko siis olemassa mitään mielenkiintoisia verkko-ongelmia, joita pystytään näin rajallisessa laskennan mallissa ratkaisemaan?

3 Ongelmien välisiä yhteyksiä

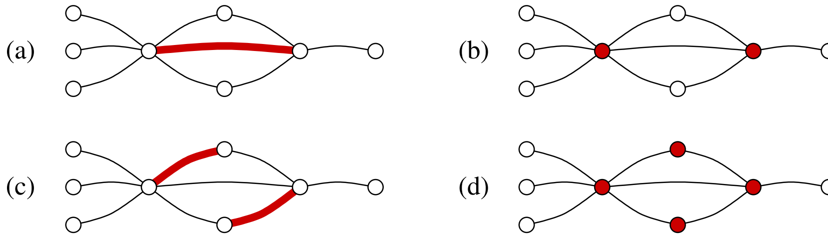
Luonnollinen lähtökohta paikallisten algoritmien suunnittelulle on soveltaa keskitetyistä algoritmeista tuttuja tekniikoita. Käymme tässä luvussa läpi erään yksinkertaisen mutta lupaavalta vaikuttavan tekniikan, jota voi käyttää pienien solmupeitteiden etsimiseen.

Luvussa 1 esittelimme kaksi verkko-ongelmaa: pariutukset ja solmupeitteet. Äkkiseltään näillä ei tunnu olevan juuri mitään yhteistä: Pariutus on maksimointiongelma ja solmupeite on puolestaan minimointiongelma. Pariutuksessa etsitään kaarien osajoukkoa, kun taas solmupeitteessä etsitään solmujen osajoukkoa. Suurimman pariutuksen löytämiseen on olemassa tehokas keskitetty algoritmi, mutta pienimmän solmupeitteen löytäminen on laskennallisesti vaikea (NP-kova) ongelma.

Osoittautuu kuitenkin, että pariutuksia voi hyödyntää myös pienien solmupeitteiden etsimisessä. Tätä varten tarvitsemme *maksimaalisen pariutuksen* käsitteen.

Pariutusta, johon ei voida enää lisätä yhtään kaarta, sanotaan maksimaaliseksi. Esimerkiksi kuvien 4a ja 4c pariutukset ovat maksimaalisia. Suurin mahdollinen pariutus on tietysti aina myös maksimaalinen, mutta maksimaalinen pariutus ei välttämättä ole suurin pariutus — tätä havainnollistaa kuvan 4a esimerkki.

Maksimaalinen pariutus on hyvin helppo löytää keskitetyllä algoritmilla: aloitetaan tyhjästä joukosta ja lisätään vain ahneesti kaaria kunnes päädytään umpikujaan. Toisaalta taas pienimmän solmupeitteen löytäminen on hyvin vaikea ongelma. Yllättäen maksimaalisen pariutuksen avul-



Kuva 4: (a) Eräs maksimaalinen pariutus. (b) Pariutuksen päätepisteistä muodostettu solmupeite. Tässä tapauksessa solmupeite on myös pienin mahdollinen. (c)–(d) Toinen esimerkki maksimaalisesta pariutuksesta ja sen perusteella muodostetusta solmupeitteestä. Saatu solmupeite ei ole optimaalinen, mutta se on aina enintään kaksi kertaa pienimmän solmupeitteen kokoinen.

la on kuitenkin mahdollista löytää *melko pieni* solmupeite. Algoritmi toimii seuraavasti: etsitään mikä tahansa maksimaalinen pariutus M ja poimitaan solmupeitteeseen C mukaan kaikki M:n kaarien päätepisteet. Tätä on havainnollistettu kuvissa 4b ja 4d.

Näin muodostettu solmujoukko C on aina solmupeite: Jos jokin kaari $e \in E$ ei olisi peitetty, olisimme voineet lisätä kaaren e mukaan pariutukseen M. Tämä taas on ristiriidassa sen kanssa, että M on maksimaalinen.

Voidaan myös osoittaa, että solmupeite C on enintään kaksi kertaa niin suuri kuin pienin mahdollinen solmupeite: Pienimmänkin solmupeitteen on peitettävä verkon kaikki kaaret, erityisesti siis myös joukon M kaaret. Pienin solmupeite joutuu sisältämään ainakin yhden solmun kutakin M:n kaarta kohti (ja mahdollisesti muitakin solmuja). Joukossa C puolestaan on vain kaksi solmua kutakin M:n kaarta kohti (eikä muita solmuja). Siis C sisältää enintään kaksi kertaa niin monta solmua kuin pienin solmupeite — toisin sanoen peite C on 2-*approksimaatio* pienimmästä solmupeitteestä.

Edellä kuvattu algoritmi on tyypillisesti ensimmäisiä esimerkkejä approksimointialgoritmien kursseilla [11]. Vaikka algoritmi on hyvin yksinkertainen — etsitään ahneesti maksimaalinen pariutus ja tulostetaan kaikki pariutuksen päätepisteet — se on myös lähestulkoon paras tunnettu algoritmi pienien solmupeitteiden löytämiseen. Ei esimerkiksi tunneta tehokasta algoritmia, joka löytäisi solmupeitteen, joka on taatusti enintään 1,99 kertaa pienimmän solmupeitteen kokoinen.

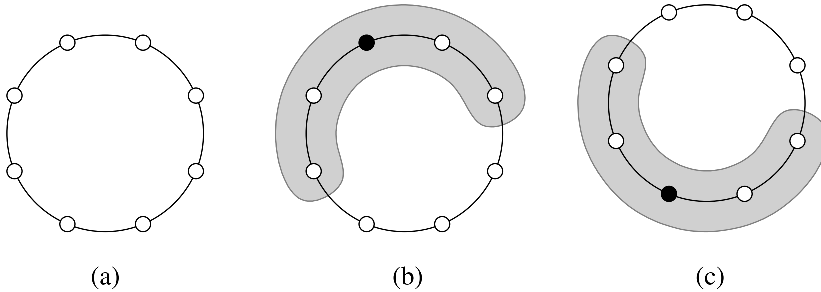
Nyt meillä on konkreettinen idea, jota voimme käyttää hajautetun algoritmin laatimisessa: Meidän tarvitsee vain suunnitella hajautettu algoritmi, joka löytää jonkin maksimaalisen pariutuksen. Tämän avulla voisimme tuottaa 2-approksimaation solmupeitteestä. Ajatus on hyvä, mutta saako tämän tehtyä paikallisella algoritmilla eli vakiomäärässä kommunikaatiokierroksia?

4 Symmetrian rikkominen

Keskeinen haaste hajautetuissa algoritmeissa on *symmetrian rikkominen*. Pahimman tapauksen verkot ovat yleensä rakenteeltaan hyvin symmetrisiä: kaikkien solmujen paikalliset ympäristöt ovat rakenteeltaan samanlaisia. Monissa ongelmissa verkon solmujen pitää tästä huolimatta tuottaa *eri* tuloste, eli hajautetun algoritmin pitää pystyä ”rikkomaan symmetria”.

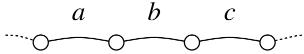
4.1 Pariutus syklissä

Hyvä (paha) esimerkki symmetrisestä verkosta on rengasmainen verkko eli *sykli*; ks. kuvaa 5. Tällöin kaikkien solmujen paikallinen ympäristö on pelkkä polku.



Kuva 5: (a) Sykli. (b)–(c) Syklissä kaikkien solmujen paikalliset ympäristöt ovat rakenteeltaan samanlaisia. Esimerkiksi minkä tahansa solmun 2-säteinen ympäristö on polku, joka koostuu neljästä kaaresta ja viidestä solmusta.

Maksimaalisen pariutuksen löytäminen puolestaan on hyvä esimerkki ongelmasta, jossa symmetrian rikkominen on välttämätöntä. Tarkastellaan esimerkiksi seuraavaa syklin osaa, jossa a, b ja c ovat peräkkäisiä kaaria:



Selvästikään kaikki kaaret a, b ja c eivät voi olla samanaikaisesti mukana pariutuksessa M. Toisaalta maksimaalisessa pariutuksessa M tulee olla näistä kaarista ainakin yksi (muuten M ei olisi maksimaalinen). Siis näiden kaarien a, b ja c osalta joudumme rikkomaan symmetrian: kaarien lähiympäristöt näyttävät samanlaisilta, mutta joudumme silti tekemään eri valintoja — ainakin yksi näistä kaarista on otettava mukaan pariutukseen ja ainakin yksi näistä on jätettävä pois.

Tässä kirjoituksessa keskitytään nimenomaan *deterministisiin hajautettuihin algoritmeihin*, jotka eivät käytä satunnaislukuja apuna. Tällöin edellä kuvatussa tilanteessa ainoaksi mahdollisuudeksi jää *symmetrian rikkominen solmujen yksilöllisten tunnisteiden perusteella*.

Jos tietäisimme, että solmujen yksilölliset tunnisteet ovat esimerkiksi kokonaislukuja $1, 2, 3, \dots$ ja luvut esiintyvät syklissä tässä järjestyksessä, ongelma olisi helppo ratkaista. Maksimaaliseen pariutukseen voitaisiin ottaa mukaan esimerkiksi kaaret $\{2, 3\}$, $\{4, 5\}$, $\{6, 7\}$ jne.

Hajautettu algoritmi ei kuitenkaan voi tietää, missä paikassa verkkoa tietyllä tunnisteella varustettu solmu sattuu sijaitsemaan. Juuri tämä tekee symmetrian rikkomisesta hankalaa.

On kyllä helppoa keksiä erilaisia ratkaisuideoita, mutta monet niistä on yhtä helppo teilata. Voisi esimerkiksi harkita algoritmia, jossa solmut, joiden tunniste on parillinen, toimivat eri tavalla kuin solmut, joiden tunniste on pariton. Mutta tämä nimenomainen algoritmi ei riitäkään, jos syklin jossain osassa peräkkäiset solmut on nimetty esimerkiksi $2, 4, 6, 8, \dots$.

Onko siis mahdollista rikkoa symmetriaa syklissä pelkkien yksilöllisten tunnisteiden avulla käyttämällä tiukasti paikallista algoritmia?

4.2 Cole–Vishkin-tekniikka

Osoittautuu, että symmetrian rikkominen syklissä on mahdollista ”melkein muttei aivan” paikallisilla algoritmeilla. On nimittäin olemassa erittäin tehokas tekniikka, jonka avulla voidaan löytää syklissä esimerkiksi maksimaalinen pariutus.

Algoritmin perusidean esittelivät Cole ja Vishkin vuonna 1986 [1], ja ideaa kehit-

tivät eteenpäin Goldberg, Plotkin ja Shannon [4]. Tämä nokkela bittimanipulointitekniikka on esitetty muun muassa Cormenin, Leisersonin ja Rivestin klassisen algoritmioppikirjan [2] ensimmäisen painoksen luvussa 30.5, ja suosittelen lämpimästi tutustumaan algoritmiin, jos se ei ole ennestään tuttu. Algoritmi esitetään yleensä muodossa, jossa se etsii syklille *värityksen* tai *riippumattoman joukon*, mutta pariutuksen tapaus on aivan vastaava.

Cole–Vishkin-algoritmi ei ole kuitenkaan paikallinen. Sen ajoaikaa ei ole rajattu millään vakiolla vaan ajoaika kasvaa, kun syklin koko kasvaa. Algoritmin ajoaika on kuitenkin äärimmäisen hitaasti kasvava funktio solmujen määrästä. Jos syklissä on n solmua ja niiden yksilölliset tunnisteet ovat kokonaislukuja $1, 2, \ldots, n$ mielivaltaisessa järjestyksessä, algoritmi löytää värityksen $O(\log^* n)$ kommunikaatiokierroksessa. Tässä $\log^* n$ on iteroitu logaritmi n:stä — se kertoo, kuinka monta kertaa luvusta n on otettava kaksikantainen logaritmi ennen kuin tulos on alle 1.

Iteroitu logaritmi kasvaa erittäin hitaasti; esimerkiksi

$$\log^* 3 = 2,$$
$$\log^* 5 = 3,$$
$$\log^* 10000 = 4,$$
$$\log^* 10^{10000} = 5.$$

Iteroitu logaritmi mistä tahansa käytännössä vastaan tulevasta luvusta on siis hyvin pieni kokonaisluku, mutta teoreetikko ei tähän tietenkään tyydy. Olisiko sittenkin mahdollista päästä aikavaativuuteen $O(1)$?

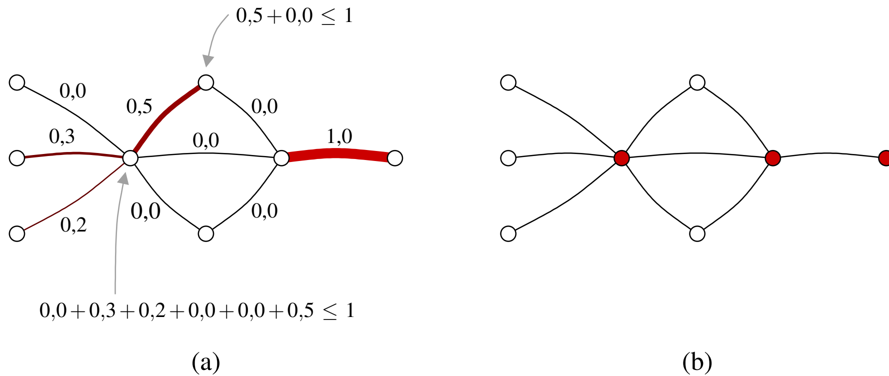
4.3 Mahdottomuustuloksia

Linial osoitti vuonna 1992 [6], että edellä kuvattu Cole–Vishkin-algoritmi on itse asiassa (vakiokertoimia vaille) paras mahdollinen algoritmi maksimaalisen pariutuksen löytämiseen. Tiukasti paikallista algoritmia ongelman ratkaisemiseen ei siis ole olemassakaan. Vaikka yrittäisi laatia kuinka ovelan paikallisen algoritmin, ja vaikka antaisi algoritmin käyttää suuren määrän kommunikaatiokierroksia — esimerkiksi 1000 kierrosta — aina löytyy syöteverkko, jossa algoritmi väistämättä tuottaa väärän ratkaisun.

Tämä tulos on luonteeltaan hyvin toisenlainen kuin tyypilliset keskitettyihin algoritmeihin liittyvät alarajatulokset. Usein ongelma osoitetaan ”vaikeaksi” esimerkiksi näyttämällä, että ongelma on NP-kova. Tällainen alarajatulos on kuitenkin *ehdollinen*: jos vaativuusluokat P ja NP eivät ole samoja, ongelmaa on mahdoton ratkaista polynomisessa ajassa, mutta emme kuitenkaan tiedä, päteekö $P \neq NP$. Linialin tulos on sen sijaan *ehdoton*: se osoittaa aukottomasti, ettei maksimaalista pariutusta voi löytää syklissä millään hajautetulla algoritmilla nopeammin kuin ajassa $\Omega(\log^* n)$.

Luonnollisesti maksimikokoisen pariutuksen etsiminen on vähintään yhtä vaikea ongelma, eikä sitäkään voi ratkaista paikallisesti. Syklissä ei ole mahdollista löytää edes ”melko suurta” pariutusta paikallisesti — ei ole olemassa esimerkiksi paikallista algoritmia, joka etsisi pariutuksen, jonka koko on vähintään sadasosa suurimman parituksen koosta [3].

Kombinatoriikasta ja yleensäkin matematiikasta kiinnostuneille mainittakoon, että tämän kaltaisten mahdottomuustulosten todistamisessa hyvin tehokas työkalu on *Ramseyn lause* [5, 8]. Ramseyn lause kertoo muun muassa, että vaikka kuinka nokkelasti yrittäisi värittää täydellisen verkon kaaria, aina verkkoon syntyy vähintään tietyn kokoinen aliverkko, jonka kaikki kaaret ovat saman värisiä. Hajautettuihin algoritmeihin sovellettuna lauseen yleistyksen voi muotoilla vaikkapa näin: vaikka kuinka nokkelasti yrittää rikkoa symmetriaa solmujen tunnisteiden avulla, aina löytyy ikävä syöte, jonka joissakin osissa symmetrian rikkominen epäonnistuu.



Kuva 6: (a) Maksimaalinen kaaripakkaus: kaariin on liitetty lukuja nollan ja ykkösen väliltä, jokaiseen solmuun liittyvien kaarien summa on *enintään* 1, ja mitään luvuista ei voi kasvattaa ilman, että tämä ehto rikkoutuisi. (b) Kun poimimme solmut, joihin liittyvien kaarien summa on *täsmälleen* 1, saamme pienen solmupeitteen.

5 Myönteisiä yllätyksiä

Tilanne näyttää siis varsin heikolta: pariutuksia ei voi löytää paikallisella algoritmilla, ja lupaavin lähestymistapa solmupeitteiden löytämiseen taas perustui nimenomaan maksimaalisiin pariutuksiin. Tästä huolimatta paikallisilla algoritmeilla on mahdollista löytää pieniä solmupeitteitä.

5.1 Työkalun vaihto

Itse asiassa paikallisilla algoritmeilla on mahdollista päästä täsmälleen samaan approksimointisuhteeseen 2 kuin mihin päästään maksimaalisten pariutusten avulla [12, 13]. Vakioajassa voidaan siis löytää solmupeite, joka on enintään kaksi kertaa pienimmän solmupeitteen kokoinen.

Keskitetty 2-approksimointialgoritmi solmupeitteeseen on varsin helppo ymmärtää, kun on ensin esitellyt oikean työkalun: maksimaaliset pariutukset. Myös paikallinen 2-approksimointialgoritmi perustuu sopivaan työkaluun, mutta kuten edellä todettiin, työkaluna ei voi käyttää maksimaalisia pariutuksia. Luontevalta tuntuisi etsiä työkalua, joka täyttää nämä vaatimukset:

- Työkalu muistuttaa jossain määrin maksimaalisia pariutuksia.
- Työkalu ei edellytä symmetrian rikkomista esimerkiksi sykleissä.
- Työkalu tästä huolimatta auttaa pienien solmupeitteiden etsimisessä.

Osoittautuu, että niin sanotut maksimaaliset kaaripakkaukset täyttävät nämä ehdot.

5.2 Maksimaaliset kaaripakkaukset

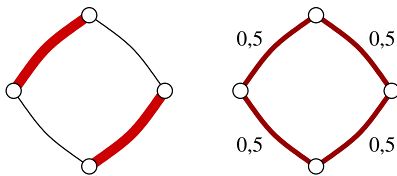
Kaaripakkauksessa verkon jokaiseen kaareen liitetään luku nollan ja ykkösen väliltä; ks. kuvaa 6a. Lukujen tulee toteuttaa seuraava ehto jokaiselle verkon solmulle v: solmuun v liittyvien kaarien summan tulee olla enintään 1.

Jokin kaaripakkaus on tietysti helppo löytää: jos esimerkiksi asetetaan kaikki luvut nolliksi, tuloksena on varmasti kaaripakkaus. Oleellista on kuitenkin tarkastella *maksimaalisia* kaaripakkauksia: sellaisia, joissa mitään lukua ei voi enää kasvattaa. Esimerkiksi kuvan 6a kaaripakkaus on maksimaalinen: jos vaikkapa luvun 0,3 tilalle vaihdettaisiin hiukan suurempi luku

0,301, verkossa olisi solmu, johon liittyvien kaarien summa olisi hiukan yli 1.

Maksimaaliset kaaripakkaukset ovat läheistä sukua maksimaalisille pariutuksille. Itseasiassa kaaripakkaus voidaan tulkita pariutuksen ”murtolukuversioksi”: pariutuksessa kaarille annetaan vain kokonaislukuja 1 (kaari on mukana pariutuksessa) tai 0 (kaari ei ole mukana pariutuksessa), kun taas kaaripakkauksessa voidaan antaa myös murtolukuja 0:n ja 1:n väliltä.

Oleellinen ero kaaripakkausten ja pariutusten välillä on kuitenkin se, että maksimaalisen pariutuksen löytäminen vaatii symmetrian rikkomista sykleissä, mutta maksimaalisen kaaripakkauksen löytäminen ei. Tätä on havainnollistettu seuraavassa kuvassa — vasemmalla maksimaalinen pariutus ja oikealla kaaripakkaus:



5.3 Kaaripakkauksen löytäminen

Osoittautuu, että tämä pieni ero riittää. Maksimaalisia pariutuksia ei voi löytää paikallisella algoritmilla, mutta maksimaalisia kaaripakkauksia *voi* löytää. Meidän tutkimusryhmämme on kehittänyt tähän ongelmaan kaksikin luonteeltaan erilaista algoritmia, joita voi lyhyesti luonnehtia seuraavasti:

- ”Puolitusalgoritmi” [12]: Algoritmi etenee vaiheissa. Kussakin vaiheessa algoritmi keskittyy ”väljiin” kaariin eli kaariin, joihin liittyviä lukuja voi vielä kasvattaa. Joka vaiheessa algoritmi poistaa yksittäiseen kaareen e liittyvästä väljyydestä joko kaiken tai täsmälleen puolet — ensimmäisessä tapauksessa kaaresta tulee ”tiukka” ja algoritmi on näiltä osin valmis; jälkimmäisessä tapauksessa kaaren käsittely jatkuu seuraavassa vaiheessa. Algoritmia analysoitaessa havaitaan, että joka solmun ympäristössä on kaaria, jotka muuttuvat tiukoiksi; algoritmi siis koko ajan edistyy kohti maksimaalista kaaripakkausta.

- ”Onnekas sattuma” [13]: Ensimmäisessä vaiheessa algoritmi kasvattaa kaariin liittyviä lukuja ahneesti mutta turvallisesti: lukuja kasvatetaan reippaasti, mutta ne muodostavat joka tapauksessa kelvollisen kaaripakkauksen.

  Ensimmäinen vaihe ei kuitenkaan välttämättä löydä maksimaalista kaaripakkausta. Osoittautuu, että täsmälleen niissä tilanteissa, joissa maksimaalista kaaripakkausta ei löydetty, onnistuttiin löytämään verkosta hiukan symmetrian rikkomisessa auttavaa tietoa (esimerkiksi solmuja, joilla on eri asteluvut). Toisessa vaiheessa voidaan sitten täydentää kaaripakkaus maksimaaliseksi hyödyntämällä luvussa 4.2 mainittua Cole–Vishkin-tekniikkaa ja ensimmäisessä vaiheessa kerättyjä tietoja.

Teknisenä yksityiskohtana mainittakoon, että nämä algoritmit ratkaisevat ongelman nimenomaan verkoissa, joissa yhteen solmuun liittyvien kaarien määrä on rajattu. Ilman tällaisia rajoituksia kaaripakkauksiakaan ei pysty löytämään.

5.4 Solmupeitteen approksimointi

Maksimaalisia kaaripakkauksia voidaan siis löytää paikallisilla algoritmeilla, mutta alkuperäisenä tavoitteenamme oli pienien solmupeitteiden löytäminen. Maksimaalisen kaaripakkauksen avulla voidaan kuitenkin helposti löytää myös pieni solmupeite.

Riittää, että poimitaan joukkoon C kaikki solmut, joihin liittyvien kaarien summa on täsmälleen 1 — tätä on havainnollistettu kuvassa 6b.

Näin muodostettu joukko C on solmupeite. Jos jonkin kaaren $e \in E$ päätepisteistä kumpikaan ei olisi joukossa C, voisimme kasvattaa kaareen e liittyvää lukua hiukan, mikä olisi ristiriidassa kaaripakkauksen maksimaalisuuden kanssa.

Voidaan myös osoittaa, että joukko C on enintään kaksi kertaa pienimmän solmupeitteen kokoinen. Kaaripakkausten avulla voi siis löytää solmupeitteen 2-approksimaation samaan tapaan kuin maksimaalisten pariutusten avulla.

6 Yhteenveto

Tässä kirjoituksessa olemme nähneet, mitä tarkoitetaan hajautetuilla algoritmeilla ja erityisesti paikallisilla algoritmeilla. Lisäksi olemme nähneet joitakin edustavia esimerkkejä paikallisiin algoritmeihin liittyvistä tuloksista:

- Suuria pariutuksia ei ole mahdollista löytää paikallisilla algoritmeilla.
- Pieniä solmupeitteitä puolestaan on mahdollista löytää paikallisilla algoritmeilla.

Oleellinen havainto on, että paikallisten algoritmien laadinnassa tarvitaan hyvin erityyppisiä tekniikoita verrattuna esimerkiksi perinteisiin keskitettyihin algoritmeihin. Myöskään intuitiomme siitä, mikä on tietokoneelle helppoa tai vaikeaa ei suoraan päde paikallisten algoritmien maailmassa:

- Maksimaalisen pariutuksen löytäminen lienee yksinkertaisin mahdollinen verkko-ongelma keskitettyjen algoritmien näkökulmasta, ja se on hyvin helppo ratkaista ahneella algoritmilla.
- Paikallisten algoritmien näkökulmasta maksimaaliset pariutukset ovat kuitenkin mahdoton ongelma — keskeinen kompastuskivi on se, että pariutus edellyttää symmetrian rikkomista.

Tämän vuoksi esimerkiksi solmupeitteen 2-approksimaation löytävä paikallinen algoritmi näyttääkin varsin erilaiselta kuin vastaava keskitetty algoritmi. Kun keskitetty algoritmi voi helposti etsiä maksimaalisen pariutuksen ja hyödyntää sitä pienen solmupeitteen löytämisessä, paikallisen algoritmin tulee hyödyntää eri tekniikoita — esimerkiksi maksimaalisia kaaripakkauksia. Lisäksi maksimaalisten kaaripakkaustenkaan löytäminen ei ole mitenkään ilmeistä paikallisilla algoritmeilla, mutta se osoittautuu kuitenkin mahdolliseksi.

7 Lopuksi

Tiukasti paikallisten algoritmien tutkimus on suhteellisen nuori ala; kiinnostus paikallisia algoritmeja kohtaan alkoi toden teolla vasta 2000-luvulla. Sekä positiivisia että negatiivisia tuloksia julkaistaan ripeää tahtia; olen koonnut näistä yhteenvetoa verkosta löytyvään kirjallisuuskatsaukseen [9].

Suuren osan alalla tehtävästä työstä voi kiteyttää yhteen lauseeseen: *oleellista on esittää oikea kysymys*. Paikallisten algoritmien kaltaisia erittäin rajoittuneita laskennan malleja tutkittaessa on hyvin helppo esittää vääriä kysymyksiä. Useimmiten vastaus kysymykseen ”voisikohan ongelman X ratkaista paikallisella algoritmilla?” on triviaalisti ”ei tietenkään!”

Tärkeää on hakea rajoja — etsiä kiintoisia ongelmia, jotka ovat juuri sillä rajalla, ettei olekaan ilmeistä, voiko ne ratkaista paikallisesti. Maksimaaliset pariutukset ja maksimaaliset kaaripakkaukset ovat hyvä esimerkki näistä: pariutukset sattuvat sijaitsemaan hiukan rajan toisella puolella, kun taas kaaripakkaukset juuri ja juuri onnistuvat paikallisesti.

Meidän tutkimusryhmässämme tehtävää tutkimusta luonnehtii monellakin tavalla rajojen hakeminen. Esimerkiksi solmupeitteen osalta tiedettiin jo ennestään,

että kahta *suurempaan* approksimointisuhteeseen on mahdollista päästä paikallisesti, kun taas kahta *pienempään* approksimointisuhteeseen ei ole mahdollista päästä. Ei kuitenkaan tiedetty, onko mahdollista päästä *täsmälleen* approksimointisuhteeseen 2. Tämän kysymyksen tutkiminen johti muun muassa luvussa 5.3 mainittuihin uusiin tekniikoihin, joiden avulla voidaan löytää kaaripakkauksia paikallisesti.

Rajojen hakeminen jatkuu edelleen. Mitenköhän hyvin voidaan esimerkiksi pieniä *dominoivia kaarijoukkoja* approksimoida paikallisilla algoritmeilla [10]?

Kiitokset

Suuret kiitokset Topi Mustolle, Joel Rybickille, Tanja Säilylle ja Antti Valmarille palautteesta ja kommenteista.

Tätä työtä ovat tukeneet Suomen Akatemia (hanke 132380) sekä Suomen Kulttuurirahasto. Osa työstä on tehty kirjoittajan vieraillessa Braunschweigin teknillisessä yliopistossa.

Viitteet

1. Richard Cole ja Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.
2. Thomas H. Cormen, Charles E. Leiserson ja Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
3. Andrzej Czygrinow, Michał Hańćkowiak ja Wojciech Wawrzyniak. Fast distributed approximations in planar graphs. *Proc. 22nd Symposium on Distributed Computing (DISC 2008)*, LNCS 5218, sivut 78–92. Springer, 2008.
4. Andrew V. Goldberg, Serge A. Plotkin ja Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM Journal on Discrete Mathematics*, 1(4):434–446, 1988.
5. Ronald L. Graham, Bruce L. Rothschild ja Joel H. Spencer. *Ramsey Theory*. John Wiley & Sons, 1980.
6. Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.
7. Moni Naor ja Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.
8. Frank P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, 30:264–286, 1930.
9. Jukka Suomela. Survey of local algorithms, lokakuu 2010. http://www.iki.fi/jukka.suomela/local-survey
10. Jukka Suomela. Distributed algorithms for edge dominating sets. *Proc. 29th Symposium on Principles of Distributed Computing (PODC 2010)*, sivut 365–374. ACM Press, 2010.
11. Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001.
12. Matti Åstrand, Patrik Floréen, Valentin Polishchuk, Joel Rybicki, Jukka Suomela ja Jara Uitto. A local 2-approximation algorithm for the vertex cover problem. *Proc. 23rd Symposium on Distributed Computing (DISC 2009)*, LNCS 5805, sivut 191–205. Springer, 2009.
13. Matti Åstrand ja Jukka Suomela. Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks. *Proc. 22nd Symposium on Parallelism in Algorithms and Architectures (SPAA 2010)*, sivut 294–302. ACM Press, 2010.