hyväksymispäivä arvosana

arvostelija

Ydinpalautukset solmupeiteongelmalle

Tuukka Korhonen

Helsinki 16.4.2019

Kandidaatintutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen osasto

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Avdelning — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen osasto

Tekijä — Författare — Author

Tuukka Korhonen

Työn nimi — Arbetets titel — Title

Ydinpalautukset solmupeiteongelmalle

Oppiaine — Läroämne — Subject

Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	16.4.2019	21 sivua + 2 liitesivua

Tiivistelmä — Referat — Abstract

Parametrisoidussa algoritmiikassa algoritmien aikavaativuutta analysoidaan syötteen koon sijasta jonkin muun parametrin mukaan. Solmupeiteongelma, jossa parametrina on halutun solmupeitteen koko, on paljon tutkittu ongelma parametrisoidussa algoritmiikassa. Ydinpalautus solmupeiteongelmalle saa syötteenä verkon ja palauttaa ytimen eli pienemmän verkon, jossa vastaus solmupeiteongelmaan on sama. Ytimen koolle on yläraja, joka riippuu vain parametrista. Esittelemme kolme ydinpalautusta solmupeiteongelmalle: Bussin palautus, kruunupalautus ja Nemhauser–Trotterin palautus. Kruunupalautus ja Nemhauser–Trotterin palautus tuottavat parametriin nähden lineaarisen kokoisen ytimen. Ne molemmat löytävät verkosta kruunun eli joukon solmuja, joille palautus ahneesti päättää, kuuluvatko ne solmupeitteeseen. Vertailemme laskennallisilla kokeilla kruunupalautusta, Nemhauser–Trotterin palautusta ja yksinkertaista palautussääntöä, joka ei ole ydinpalautus. Toisin kuin aiemmin esitetyissä koetuloksissa, tuloksiemme perusteella kruunuihin perustuvat ydinpalautukset eivät pienennä solmupeiteongelman instansseja enempää kuin yksinkertaisemmat palautussäännöt.

ACM Computing Classification System (CCS):
Theory of computation → Design and analysis of algorithms → Parameterized complexity and exact algorithms

Avainsanat — Nyckelord — Keywords

parametrisoidut algoritmit, ydinpalautus, solmupeiteongelma, kruunupalautus

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto **1**

2 Parametrisoidut algoritmit **1**

2.1 Merkinnät . . . 2

2.2 Solmupeiteongelma . . . 2

2.3 Kiintoparametriratkeavuus ja ydinpalautus . . . 2

2.4 Bussin palautus . . . 3

3 Kruunupalautus **5**

3.1 Kruunujen ominaisuuksia . . . 6

3.2 Algoritmi kruunun löytämiseen . . . 7

4 Nemhauser–Trotterin palautus **10**

4.1 Kokonaisluku- ja lineaarinen optimointi . . . 10

4.2 Solmupeite lineaarisena ohjelmana . . . 12

4.3 Muotoilu maksimiparituksena . . . 13

5 Approksimointialgoritmit **14**

5.1 Kruunupalautus approksimointialgoritmina . . . 15

5.2 Näyttö optimaalisuudesta . . . 16

6 Laskennalliset kokeet **16**

6.1 Koeasetelma . . . 17

6.2 Tulokset . . . 18

7 Yhteenveto **19**

Lähteet **20**

Liitteet

1 Laskennalliset kokeet instanssikohtaisesti

1 Johdanto

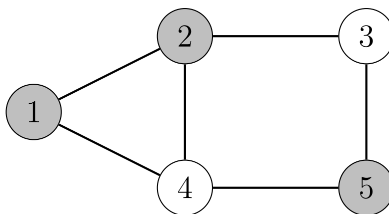
Parametrisoitu algoritmiikka on 1990-luvulla alkunsa saanut algoritmiikan kehityssuunta, jossa algoritmien aikavaativuutta analysoidaan syötteen koon sijasta jonkin muun parametrin suhteen [5]. Parametrisoidun algoritmiikan tavoitteena on tunnistaa NP-kovien ongelmien tapauksia, joiden ratkaiseminen on helpompaa kuin ongelman yleisen tapauksen. Käsittelemme parametrisoitua algoritmiikkaa solmupeiteongelman näkökulmasta. Solmupeiteongelma on toiminut ensimmäisenä ongelmana, johon parametrisoidun algoritmiikan tekniikoita on kehitetty, ja siihen toimivat algoritmit ovat sekä tehokkaita että elegantteja [17].

Keskeinen parametrisoidun algoritmiikan käsite on ydinpalautus, joka tarkoittaa esikäsittelyalgoritmia, joka pienentää syötettä niin, että sen koko riippuu parametrista. Käsittelemme ydinpalautuksia solmupeiteongelmalle keskittyen kruunupalautukseen ja Nemhauser–Trotterin palautukseen. Ne etsivät verkosta rakenteen, jota kutsutaan kruunuksi, ja poistavat sen. Luvut 2–4 analysoivat ydinpalautuksia solmupeiteongelmalle teoreettisesti. Luku 5 yhdistää kruunuihin perustuvat ydinpalautukset approksimointialgoritmeihin, ja luku 6 vertailee algoritmeja laskennallisilla kokeilla. Laskennalliset kokeet osoittavat, että tutkielmassa käsitellyt ydinpalautukset ovat tehokkaita myös käytännössä, mutta toimivat silti huonommin kuin yksinkertainen esikäsittelyalgoritmi, jonka toiminnalle ei ole teoreettisia takeita. Tietääksemme tällaisia kokeellisia tuloksia ei ole ennen esitetty kirjallisuudessa.

2 Parametrisoidut algoritmit

Solmupeiteongelma on yksi klassisista NP-täydellisistä ongelmista [19]. Minkään NP-täydellisen ongelman ratkaisemiseen tarkasti ei tunneta polynomisessa ajassa toimivaa algoritmia, minkä seurauksena niitä on lähestytty vaihtoehtoisilla algoritmisilla tekniikoilla, kuten heuristiikoilla, approksimointialgoritmeilla, optimoiduilla täyden haun algoritmeilla ja parametrisoiduilla algoritmeilla.

Perinteisesti algoritmiikassa analysoidaan algoritmin aikavaativuutta funktiona syötteen koon suhteen. *Parametrisoidut algoritmit* sen sijaan pyrkivät kiteyttämään ongelman vaikeuden parametriin, joka voi olla pieni suurillakin syötteillä. Tavoitteena on, että parametrisoidun algoritmin aikavaativuuden riippuvuus syötteen koosta on polynominen ja NP-kovien ongelmien tapauksessa väistämätön ylipolynominen tekijä riippuu vain parametrista. Parametrisoituihin algoritmeihin läheisesti liittyvä käsite on *ydinpalautus*. Ydinpalautus on algoritmi, joka pienentää syötettä niin, että sen koon ylärajaksi tulee vain parametrista riippuva funktio. Tämä luku määrittelee solmupeiteongelman, kiintoparametrialgoritmit ja ydinpalautuksen. Lisäksi kuvaamme yksinkertaisen ydinpalautuksen solmupeiteongelmalle.



Kuva 1: Esimerkki verkosta, jolla on kolmen kokoinen solmupeite.

2.1 Merkinnät

Olkoon G suuntaamaton ja yksinkertainen verkko. Käytämme G:n solmuista merkintää $V(G)$ ja kaarista $E(G)$. Aikavaativuuksien yhteydessä käytämme solmujen ja kaarten määristä merkintöjä $n = |V(G)|$ ja $m = |E(G)|$. Solmun v *naapurit* $N(v)$ ovat ne solmut, joihin v on yhteydessä kaarella. Solmun *aste* on sen naapureiden määrä. Solmujoukon S *naapurustossa* $N(S)$ on kaikki S:n solmujen naapurit paitsi S:n solmut itse. Käytämme merkintää $G \setminus S$ verkosta G, josta on otettu pois joukossa S olevat solmut ja niiden viereiset kaaret. Joukko solmuja on *riippumaton joukko*, jos ei ole kaarta, jonka molemmat päätepisteet kuuluvat siihen. Verkko on *kaksijakoinen*, jos sen solmut voi jakaa kahteen osaan niin, että jokainen kaari on eri osien välillä.

Paritus verkossa G on joukko kaaria $M \subset E(G)$, jolle pätee, että jokainen solmu on korkeintaan yhden parituksessa olevan kaaren päätepiste. Solmujoukkojen S ja T välillä olevassa parituksessa jokaisen kaaren toinen päätepiste on joukossa S ja toinen joukossa T. Käytämme parituksen kaarten päätepisteistä eli parituksen solmuista merkintää $V(M)$, ja sanomme, että M parittaa solmun u solmuun v, jos $(u, v) \in M$.

2.2 Solmupeiteongelma

Joukko solmuja on *solmupeite*, jos siinä olevat solmut peittävät kaikki verkon kaaret. Solmu peittää kaaren, jos se on jompikumpi kaaren päätepisteistä. Kuvassa 1 on verkko, jolla on solmupeite, johon kuuluvat solmut $\{1, 2, 5\}$. Solmu 1 peittää kaaret $(1, 2)$ ja $(1, 4)$, solmu 2 peittää kaaret $(1, 2)$, $(2, 4)$ ja $(2, 3)$ ja solmu 5 peittää kaaret $(3, 5)$ ja $(4, 5)$. Tämän solmupeitteen koko on 3, eikä kyseisellä verkolla ole pienempää solmupeitettä. Solmupeiteongelmassa tavoitteena on selvittää, onko verkolla G solmupeitettä, johon kuuluu korkeintaan k solmua. Syötteenä oleva pari (G, k) on solmupeiteongelman *instanssi*. Solmupeiteongelma on NP-täydellinen, joten siihen ei tunneta polynomisessa ajassa toimivaa algoritmia. Tehokkain tunnettu algoritmi solmupeiteongelmalle toimii ajassa $O(1.1996^n)$ [27].

2.3 Kiintoparametriratkeavuus ja ydinpalautus

Parametrisoidun ongelman instanssissa on syötteen lisäksi mukana *parametri*, kokonaisluku k, joka tyypillisesti kuvaa instanssin rakennetta jollakin tavalla. Paramet-

risoidut algoritmit tavoittelevat aikavaativuutta, joka riippuu vahvasti parametrista k, mutta ei juurikaan syötteen koosta. Luonteva parametri solmupeiteongelmalle on halutun solmupeitteen koko. Se on ainoa parametri, jota käsittelemme tässä tutkielmassa. Muita parametreja solmupeiteongelmalle ovat esimerkiksi verkon puuleveys [3] ja solmupeitteen koko verrattuna maksimiparitukseen [23].

Parametrisoitu ongelma on *kiintoparametriratkeava*, jos sen ratkaisemiseen on algoritmi, jonka aikavaativuus on $O(p(|X|)f(k))$, jossa $|X|$ on syötteen koko, k parametri, p polynomifunktio ja f mikä tahansa funktio [5]. Näin algoritmin käyttämää ylipolynomista aikaa kuvaa $f(k)$, joka riippuu vain parametrista k. Tällainen algoritmi on *kiintoparametrialgoritmi*.

Toinen parametrisoidun algoritmiikan keskeinen käsite on ydinpalautus, jota voidaan pitää esikäsittelyn formalisointina. Ydinpalautus saa syötteenä ongelman instanssin (X, k) ja palauttaa instanssin (X', k'), jolle ongelman vastaus on sama kuin instanssille (X, k). Vaatimukset ydinpalautukselle ovat, että se toimii polynomisessa ajassa ja $|X'| \leq g(k)$ jollekin funktiolle g [5]. Instanssi (X', k') on *ydin*. Näin voidaankin ajatella, että ydinpalautus tiivistää ongelman vaikeuden sen ytimeen.

Lause 1 ([5])**.** *Ongelma on kiintoparametriratkeava, jos siihen on ydinpalautus.*

Todistus. Rakennamme kaksivaiheisen kiintoparametrialgoritmin ydinpalautuksen avulla. Kiintoparametrialgoritmin ensimmäinen vaihe käyttää ydinpalautusta, joka palauttaa ytimen (X', k'). Toinen vaihe ratkaisee ongelman, kun syötteenä on ydin (X', k'), käyttämällä mitä tahansa algoritmia ja palauttaa saadun vastauksen. Merkitään toisessa vaiheessa käytetyn algoritmin aikavaativuutta $T(|X'|)$:llä. Ensimmäisen vaiheen aikavaativuus on polynominen $O(p(|X|))$, ja toisen vaiheen aikavaativuus on $T(|X'|) = O(T(g(k)))$, koska $|X'| \leq g(k)$. Niinpä kokonaisuuden aikavaativuus on $O(p(|X|) + T(g(k))) = O(p(|X|)T(g(k)))$. □

Todistus tarkoittaa käytännössä, että kaikki algoritmit, jotka käyttävät ensimmäisenä vaiheena ydinpalautusta, ovat kiintoparametrialgoritmeja. Tämä kuvaakin hyvin, miten ydinpalautusta käytetään usein esikäsittelyvaiheena algoritmien suunnittelussa sekä teoreettisessa [7] että kokeellisessa [2] algoritmiikassa. Yllättävää kyllä, lause 1 pätee myös toiseen suuntaan: jos ongelma on kiintoparametriratkeava, niin siihen on ydinpalautus [5]. Tämän suunnan seuraukset ovat kuitenkin vähemmän käytännöllisiä [10].

2.4 Bussin palautus

Ensimmäisenä esimerkkinä ydinpalautuksesta solmupeiteongelmalle käsittelemme Bussin palautusta [4]. Palautus perustuu kahteen havaintoon, jotka käyttävät hyväksi halutun solmupeitteen kokoa k:

1. Jos verkossa on solmu, jonka aste on yli k, niin kyseisen solmun on pakko olla mukana solmupeitteessä, koska muuten sen kaikkien naapureiden pitäisi olla mukana solmupeitteessä, mikä on mahdotonta.

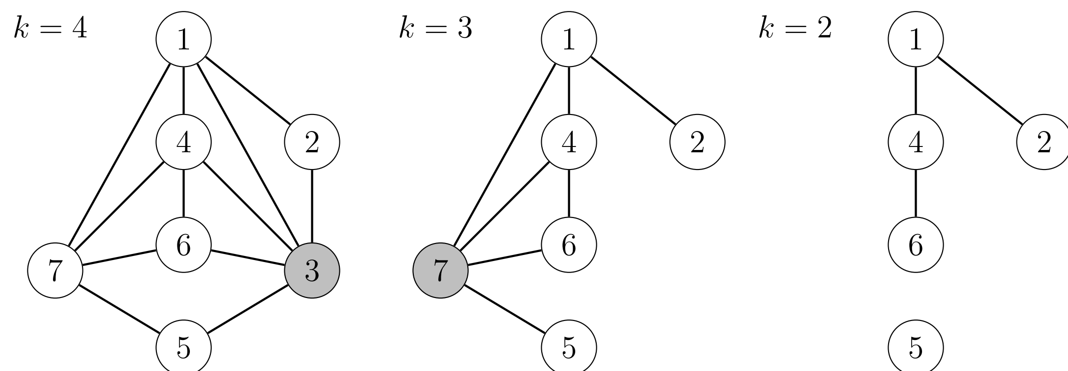
2. Jos verkon suurin solmun aste on d, niin k:n kokoinen solmupeite peittää korkeintaan kd kaarta.

Bussin palautus käyttää havainnon 1 perusteella toimivaa palautussääntöä korvaamaan ongelman instanssin uudella instanssilla niin kauan kuin mahdollista ja osoittaa havainnon 2 avulla, että instanssi, johon tätä palautussääntöä ei voi käyttää, on ydin. Havaintoon 1 perustuva palautussääntö korvaa instanssin (G, k) instanssilla $(G \setminus \{v\}, k - 1)$, jos solmun v aste on yli k. Tämä vastaa v:n valitsemista ahneesti solmupeitteeseen, koska sääntö poistaa v:n peittämät kaaret ja jäljellä olevan solmupeitteen koko saa olla korkeintaan $k - 1$. Kun tätä sääntöä ei voi enää soveltaa, verkon suurin solmun aste on korkeintaan k, jolloin havainnon 2 mukaan k:n kokoinen solmupeite peittää korkeintaan k^2 kaarta. Jos verkossa on jäljellä enemmän kuin k^2 kaarta (tai $k < 0$), niin vastaus solmupeiteongelmaan on kielteinen. Palautus ilmoittaa kielteisen vastauksen palauttamalla triviaalin *ei-instanssin*. Ei-instanssi voi olla esimerkiksi $(H, 0)$, jossa H on verkko, joka sisältää kaksi solmua ja yhden kaaren. Muussa tapauksessa palautus palauttaa palautussäännön soveltamisen jälkeen jäljellä olevan instanssin, josta se lisäksi poistaa solmut, joilla ei ole naapureita. Bussin palautus toimii polynomisessa ajassa, koska se soveltaa palautussääntöä korkeintaan n kertaa ja säännön soveltaminen on mahdollista toteuttaa polynomisessa ajassa esimerkiksi pitämällä kirjaa solmujen asteista.

Kuvassa 2 on esimerkki Bussin palautuksen soveltamisesta. Palautussääntöä käytetään kaksi kertaa: ensin solmu 3 valitaan solmupeitteeseen, koska sen aste on 5, ja sitten solmu 7 valitaan solmupeitteeseen, koska sen aste on 4. Jäljellä olevaan instanssiin (kuvassa oikealla) ei voi enää käyttää palautussääntöä. Siinä olevien kaarten määrä on $3 \leq 2^2$, joten siitä tulee Bussin palautuksen palauttama ydin.

Lause 2. *Bussin palautus tuottaa ytimen, jossa on korkeintaan $2k^2$ solmua ja k^2 kaarta.*

Todistus. Algoritmi joko palauttaa vakiokokoisen ei-instanssin tai instanssin (G', k'), jolla $|E(G')| \leq k^2$. Solmujen määrän yläraja on $2k^2$, koska jokainen solmu on jonkin kaaren päätepiste ja jokaisella kaarella on kaksi päätepistettä. □



Kuva 2: Esimerkki Bussin palautuksesta.

Seuraus 1. *Solmupeiteongelma on kiintoparametriratkeava.*

Todistus. Bussin palautus on ydinpalautus, koska se tuottaa ytimen, jossa on korkeintaan $O(k^2)$ solmua ja kaarta, ja toimii polynomisessa ajassa. Lauseen 1 mukaan tästä seuraa, että solmupeiteongelma on kiintoparametriratkeava. □

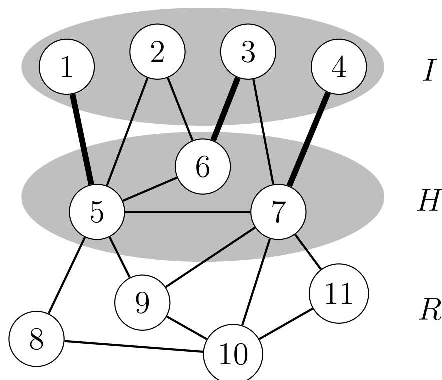
Solmupeiteongelmalle on myös muita yksinkertaisia palautussääntöjä. Tarkastelemme seuraavaksi sääntöjä, jotka poistavat kaikki solmut, joiden aste on 1 tai 2 [6]. Jos solmun aste on 1, niin palautussääntö valitsee sen naapurin ahneesti solmupeitteeseen. Asteen 2 solmujen poistaminen on hieman monimutkaisempaa. Oletetaan, että verkossa on 2-asteinen solmu v, jonka naapurit ovat $\{u, w\}$. Jos verkossa on kaari (u, w), niin joukosta $\{v, u, w\}$ vähintään kahden solmun on kuuluttava solmupeitteeseen. Solmujen u ja w ahne valinta solmupeitteeseen on tässä tapauksessa optimaalinen. Jos verkossa ei ole kaarta (u, w), niin palautussääntö korvaa solmut $\{v, u, w\}$ uudella solmulla, jonka naapureita ovat kaikki u:n ja w:n naapurit, ja vähentää vaadittua solmupeitteen kokoa yhdellä.

Nämä palautussäännöt eivät tee ydinpalautusta, koska instanssin koolle sääntöjen soveltamisen jälkeen ei ole mitään takeita. Sääntöjen soveltaminen takaa kuitenkin, että jokaisen solmun aste on vähintään 3. Bussin palautukseen yhdistettynä tämä tuottaa hieman alkuperäistä paremman ytimen, koska solmujen määrän ylärajaksi tulee $\frac{2}{3}k^2$.

Tässä luvussa esitellyt palautukset ovat toiminnaltaan yksinkertaisia, ja ne on helppo toteuttaa tehokkaasti [1]. Bussin palautus oli yksi ensimmäisistä algoritmeista, joilla esiteltiin parametrisoitujen algoritmien ja ydinpalautusten ideaa. Solmupeiteongelma onkin toiminut testialustana uusille tekniikoille parametrisoidussa algoritmiikassa [17].

3 Kruunupalautus

Vuonna 2003 esitelty kruunupalautus [13] saavuttaa pienemmän ytimen kuin Bussin palautus laajentamalla ahneen valinnan yhdestä solmusta kokonaiseen solmujoukkoon. Kruunupalautusta voidaan pitää 1-asteiset solmut poistavan säännön yleistyksenä, koska se löytää verkosta joukon solmuja, jotka eivät kuulu pienimpään solmupeitteeseen, mutta joiden naapurusto kuuluu siihen. Kruunupalautus jakaa verkon solmut kolmeen osaan, I, H ja R, valitsee osan H ahneesti solmupeitteeseen ja poistaa osan I. Palautuksesta jäljelle jäävän verkon solmut ovat siis R. Kruunupalautuksen verkosta poistamaa paria (I, H) kutsutaan *kruunuksi.* Siispä mitä suurempi kruunu on, sitä enemmän palautus pienentää instanssia. Kruunupalautus perustuu algoritmiin, joka löytää verkosta vähintään $n - 3k$ solmun kokoisen kruunun, eli kruunupalautus tuottaa ytimen, jonka koko on korkeintaan $3k$. Esittelemme tässä luvussa kruunupalautuksen, todistamme, että se on ydinpalautus, ja analysoimme sen tehokkuutta. Käytämme kruunujen ominaisuuksia myös luvussa 4 Nemhauser–Trotterin palautuksen analysoinnissa.



Kuva 3: Esimerkki jaosta I, H, R. Parituksen kaaret on lihavoitu.

3.1 Kruunujen ominaisuuksia

Kruunupalautus jakaa verkon solmut osiin I, H ja R, jotka toteuttavat seuraavat ehdot:

1. I on riippumaton joukko.
2. I:n ja R:n välillä ei ole kaaria.
3. I:n ja H:n välillä on paritus, jonka koko on $|H|$.

Kuvassa 3 on esimerkki verkon jaosta osiin I, H ja R. Kuvasta näemme, että $I = \{1, 2, 3, 4\}$ on riippumaton joukko, koska sen sisällä ei ole kaaria. Solmut, joihin I:n solmut ovat yhteydessä, ovat $H = \{5, 6, 7\}$, joten joukko H erottaa I:n ja R:n. I:n ja H:n välillä on paritus $\{(1, 5), (3, 6), (4, 7)\}$, jonka koko on $|H| = 3$. Toisin sanoen se parittaa kaikki H:n solmut. Kuvan jako täyttää siis ehdot 1–3, joten pari $(I, H) = (\{1, 2, 3, 4\}, \{5, 6, 7\})$ on kruunu. Esimerkissä H on I:n naapurusto, eli $H = N(I)$. Ehdoista 2 ja 3 seuraa, että tämä pätee millä tahansa kruunulla.

Paritus on hyödyllinen käsite solmupeiteongelmaa tarkasteltaessa, koska solmupeitteessä oleva solmu voi peittää korkeintaan yhden parituksessa olevan kaaren. Lemma 1 formalisoi tämän.

Lemma 1. *Jos C on G:n solmupeite, ja M on paritus G:ssä, niin* $|C \cap V(M)| \geq |M|$.

Todistus. Vain $V(M)$:ssä olevat solmut voivat peittää M:n kaaria. Kuitenkin jokainen $V(M)$:n solmu peittää tasan yhden M:n kaaren. Siispä $V(M)$:stä pitää valita vähintään $|M|$ solmua solmupeitteeseen, jotta se peittäisi kaikki M:n kaaret. □

Kruunupalautus perustuu havaintoon, että jos (I, H) on kruunu, niin verkolla on pienin solmupeite, jossa on kaikki H:n solmut eikä yksikään I:n solmuista. Kruunupalautus valitsee H:n solmut ahneesti solmupeitteeseen, joka peittää myös kaikki I:n viereiset kaaret, koska $N(I) = H$ ja I on riippumaton joukko. Tarkemmin sanottuna kun syötteenä on instanssi (G, k) ja G:ssä on kruunu (I, H), niin kruunupalautus palauttaa instanssin $(G \setminus (I \cup H), k - |H|)$.

Lause 3 ([1, 9, 26]). *Jos verkossa on kruunu (I, H), niin verkolla on pienin solmupeite, jonka osajoukko on H.*

Todistus. Olkoon M ehdon 3 mukainen I:n ja H:n välillä oleva paritus. Koska $V(M) \subset I \cup H$ ja $|M| = |H|$, lemman 1 mukaan verkon solmupeitteessä on vähintään $|H|$ solmua joukosta $I \cup H$. Toisaalta H peittää kaikki kaaret, jotka $I \cup H$ peittää, joten H on paras mahdollinen valinta solmupeitteeseen. $\square$

3.2 Algoritmi kruunun löytämiseen

Kruunupalautus saa syötteenä solmupeiteongelman instanssin (G, k) ja löytää kruunun (I, H), jonka koko on vähintään $n - 3k$, tai ilmoittaa, että verkon pienimmän solmupeitteen koko on suurempi kuin k [1, 9]. Jos palautus löytää kruunun, se pienentää instanssia valitsemalla H:n solmut solmupeitteeseen ja poistamalla I:n solmut. Kruunupalautuksen ensimmäinen vaihe etsii verkosta suuren riippumattoman joukon I^* maksimaalisen parituksen avulla. Tästä riippumattomasta joukosta syntyvä kruunu $(I^*, N(I^*))$ ei välttämättä täytä ehtoa 3, mutta jollekin sen osajoukolle $I \subset I^*$ pätee, että $(I, N(I))$ on kruunu. Kruunupalautus löytää osajoukon I tarkastelemalla maksimiparitusta solmujoukkojen I^* ja $N(I^*)$ indusoimassa kaksijakoisessa verkossa $G[I^*, N(I^*)]$, jonka solmut ovat $I^* \cup N(I^*)$ ja kaaret ne G:n kaaret, joiden toinen päätepiste on I^*:ssä ja toinen päätepiste $N(I^*)$:ssä.

Listaus 1 kuvaa kruunupalautuksen yksityiskohtaisesti pseudokoodina. Rivillä 1 kruunupalautus etsii parituksen M_1, joka on maksimaalinen, eli se ei ole minkään muun parituksen osajoukko. Rivi 3 rakentaa joukon I^*, jossa on paritukseen M_1 kuulumattomat solmut. I^* on riippumaton joukko, koska jos siinä olisi kaksi solmua u ja v, joiden välillä on kaari, niin $M_1 \cup \{(u, v)\}$ olisi paritus, joten M_1 ei olisi maksimaalinen. Rivillä 4 funktio Maksimiparitus etsii maksimiparituksen M_2 kaksijakoisessa verkossa $G[I^*, N(I^*)]$. Rivi 6 alustaa kruunun riippumattomaksi joukoksi joukon I_0 eli ne I^*:n solmut, joita M_2 ei parittanut.

Listaus 1: Kruunupalautus

Syöte : Solmupeiteongelman instanssi (G, k)
Tuloste: *ei* tai ydin (G', k')

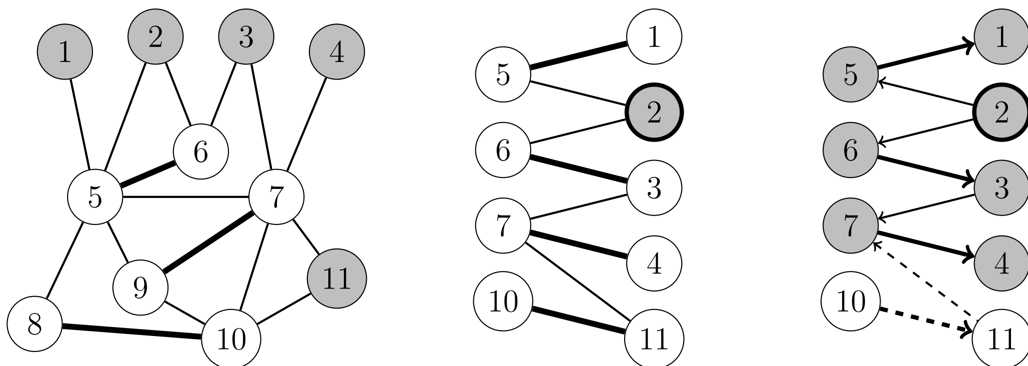
1 $M_1 \leftarrow$ Maksimaalinen-paritus(G)
2 **if** $|M_1| > k$ **then return** *ei*
3 $I^* \leftarrow V(G) \setminus V(M_1)$
4 $M_2 \leftarrow$ Maksimiparitus$(G[I^*, N(I^*)])$
5 **if** $|M_2| > k$ **then return** *ei*
6 $I_0 \leftarrow I^* \setminus V(M_2)$
7 $A \leftarrow$ Vuorotteleva-haku$(I_0, M_2, G[I^*, N(I^*)])$
8 $(I, H) \leftarrow (A \cap I^*, A \cap N(I^*))$
9 **return** $(G \setminus (I \cup H), k - |H|)$

Kruunupalautus löytää kruunun tekemällä joukon I_0 solmuista lähtevän *vuorottelevan haun* verkossa $G[I^*, N(I^*)]$ parituksen M_2 suhteen. Vuorotteleva haku kulkee joukon I^* solmuista joukon $N(I^*)$ solmuihin paritukseen kuulumattomia kaaria pitkin ja joukon $N(I^*)$ solmuista joukon I^* solmuihin parituksen kaaria pitkin. Haun voi toteuttaa esimerkiksi syvyyshaulla verkossa $G[I^*, N(I^*)]$, jossa paritukseen M_2 kuuluvat kaaret on suunnattu $N(I^*)$:stä I^*:hin ja muut kaaret I^*:stä $N(I^*)$:hin. Vuorottelevan haun saavuttamat solmut A muodostavat kruunun.

Kuva 4 näyttää esimerkin, miten kruunupalautus löytää kuvassa 3 olevan kruunun. Palautus laskee ensin maksimaalisen parituksen $M_1 = \{(5,6),(7,9),(8,10)\}$, jonka avulla se löytää riippumattoman joukon $I^* = \{1,2,3,4,11\}$. Tämän riippumattoman joukon ja sen naapuruston indusoimassa kaksijakoisessa verkossa on maksimiparitus $M_2 = \{(1,5),(3,6),(4,7),(10,11)\}$. Vuorotteleva haku lähtee paritukseen kuulumattomista I^*:n solmuista eli solmusta 2. Vuorotteleva haku saavuttaa solmut $\{1,2,3,4,5,6,7\}$, jotka muodostavat kruunun.

Lemma 2. *$H \subset V(M_2)$.*

Todistus. Oletetaan, että vuorotteleva haku löytää solmun $v \in N(I^*)$, joka ei ole parituksessa M_2. Nyt on olemassa *vuorotteleva polku*, joka lähtee parittamattomasta solmusta $u \in I_0$, päättyy parittamattomaan solmuun $v \in N(I^*)$ ja jossa joka toinen kaari kuuluu paritukseen. Polun pituus on $2p+1$, ja siinä on p paritettua ja $p+1$ parittamatonta kaarta. Muokkaamme polun avulla paritusta M_2 niin, että otamme siitä pois polussa olevat paritetut kaaret ja lisäämme siihen polussa olevat parittamattomat kaaret. Saamme parituksen, jossa on yksi kaari enemmän kuin M_2:ssa, joten M_2 ei ole maksimiparitus, mikä on ristiriita. Siispä vuorotteleva haku ei löydä yhtäkään $N(I^*)$:ssä olevaa parittamatonta solmua. □



Kuva 4: Esimerkki kruunupalautuksen toiminnasta. Vasemmalla parituksen M_1 kaaret on lihavoitu ja riippumaton joukko I^* merkitty harmaalla. Keskellä parituksen M_2 kaaret on lihavoitu ja I_0 merkitty harmaalla. Oikealla kaaret on suunnattu vuorottelevan haun mukaan ja vuorottelevan haun saavuttamat solmut merkitty harmaalla. Kaaret, joita pitkin vuorotteleva haku ei kulkenut, on merkitty katkoviivalla.

Lause 4 ([1, 9])**.** *Kruunupalautus löytää kruunun, jos G:llä on korkeintaan k:n kokoinen solmupeite.*

Todistus. Jos kruunupalautus palauttaa *ei* rivillä 2 tai rivillä 5, niin verkossa on paritus, joka on suurempi kuin k. Lemman 1 mukaan silloin verkon pienin solmupeite on suurempi kuin k. Muussa tapauksessa kruunupalautus palauttaa parin (I, H). I on riippumaton joukko, koska se on I^*:n osajoukko. Kaikki solmut, jotka ovat I:n solmujen vieressä, ovat H:ssa, koska jos vuorotteleva haku löytää solmun $v \in I$ ja $u \in N(v)$, niin joko vuorotteleva haku on saapunut solmuun v kaarta $(u, v) \in M_2$ pitkin tai vuorotteleva haku menee solmusta v solmuun u kaarta $(u, v) \notin M_2$ pitkin. Parituksen $M_3 = M_2 \cap \{(u, v) \mid u \in H, v \in I\}$ koko on $|H|$, koska lemman 2 mukaan kaikki H:n solmut ovat parituksessa M_2 ja vuorotteleva haku löytää kaikki niihin paritetut I^*:n solmut. □

Lause 5 ([1, 9])**.** *Kruunupalautuksen löytämässä kruunussa on vähintään $n - 3k$ solmua.*

Todistus. $|I^*| = |V(G)| - |V(M_1)| = n - 2|M_1| \geq n - 2k$, koska $|M_1| \leq k$. Siispä $|I_0| = |I^*| - |M_2| \geq n - 2k - k = n - 3k$, koska jokaisella M_2:ssa olevalla kaarella on tasan yksi päätepiste I^*:ssa ja $|M_2| \leq k$. $|I| \geq |I_0| \geq n - 3k$, koska vuorotteleva haku lähtee I_0:sta, joten se löytää kaikki I_0:n solmut. □

Lause 6 ([1])**.** *Kruunupalautus toimii ajassa $O(\sqrt{n}m)$.*

Todistus. Kruunupalautus löytää maksimaalisen parituksen rivillä 1 lineaarisessa ajassa käymällä kaikki kaaret läpi ja ottamalla kaaren paritukseen, jos kumpikaan sen solmuista ei vielä ole parituksessa. Rivi 4 laskee kaksijakoisen verkon maksimiparituksen Hopcroft-Karp-algoritmilla, joka toimii ajassa $O(\sqrt{n}m)$ [16]. Rivi 7 toteuttaa vuorottelevan haun syvyyshaulla, joka toimii lineaarisessa ajassa. Muut rivit tekevät yksinkertaisia joukkoja käsitteleviä operaatioita, jotka toimivat lineaarisessa ajassa. Aikavaativuutta dominoi siis maksimiparituksen löytäminen. □

Kruunupalautuksen voi toteuttaa ajassa $O(km)$ käyttämällä maksimiparitukseen algoritmia, joka löytää ajassa $O(km)$ parituksen, jossa on k kaarta [9]. Hopcroft-Karp-algoritmi toimii näin, koska se lisää jokaisella $O(m)$-ajassa toimivalla iteraatiolla paritukseen vähintään yhden kaaren, joten kruunupalautuksen aikavaativuuden voidaan sanoa olevan $O(min(k, \sqrt{n})m)$. Jos Bussin palautusta käytetään ennen kruunupalautusta, niin tämä muunnelma ei paranna aikavaativuutta, koska tällöin $\sqrt{n} \leq k$.

Seuraus 2. *Solmupeiteongelmalle on ydinpalautus, joka tuottaa ytimen, jossa on korkeintaan $3k$ solmua.*

Todistus. Kruunupalautus joko palauttaa *ei* tai löytää kruunun, jossa on vähintään $n - 3k$ solmua, ja palauttaa korkeintaan $3k$ solmun kokoisen ytimen. Lisäksi kruunupalautus toimii polynomisessa ajassa. □

Tässä luvussa esitelty kruunupalautus tuottaa ytimen, jossa on solmupeitteen kokoon nähden lineaarinen määrä solmuja. Kuitenkaan yläraja kaarten määrälle ei ole parempi kuin Bussin palautuksessa. Solmupeiteongelmalle ei tunnetakaan ydinpalautusta, jonka tuottaman ytimen kaarten määrän yläraja olisi $O(k^{2-\varepsilon})$, jossa $\varepsilon > 0$, ja tällaisen palautuksen olemassaolo kumoaisi useimpien todeksi uskoman vaativuusteorian konjektuurin [12].

4 Nemhauser–Trotterin palautus

Nemhauser–Trotterin palautus on paras tunnettu solmupeiteongelman ydinpalautus, joka on parametrisoitu solmupeitteen koon suhteen. Se tuottaa $2k$ solmun kokoisen ytimen käyttämällä lineaarista optimointia. Palautus muotoilee solmupeiteongelman kokonaislukuohjelmana ja ratkaisee tästä muokatun lineaarisen ohjelman. Solmut, joille lineaarinen optimointi antaa kokonaislukuarvot, muodostavat kruunun.

Nemhauser–Trotterin palautus esiteltiin vuonna 1975 [26], mutta silloin se muotoiltiin riippumattoman joukon laskemisen suhteen eikä silloin ollut formalisoitu parametrisoitua algoritmiikkaa. Palautuksen yhtäläisyyksiä kruunupalautukseen ei vielä tunnettu vuonna 2003, kun kruunupalautus esiteltiin ensimmäisen kerran [13], mutta myöhemmin todistettiin, että molemmat palautukset käyttävät hyväksi samaa rakennetta eli kruunua [8]. Tässä luvussa kuvaamme Nemhauser–Trotterin palautuksen algoritmina, joka löytää kruunun ja käyttää edellisessä luvussa esiteltyjä kruunujen ominaisuuksia palautuksen tekemiseen.

4.1 Kokonaisluku- ja lineaarinen optimointi

Kokonaisluku- ja lineaarinen optimointi mallintavat optimointiongelmia lineaaristen rajoitteiden avulla. Kun ongelma kuvataan lineaarisena ohjelmana, sen ratkaisemiseen voi käyttää lineaarisen optimoinnin algoritmia. *Lineaarinen ohjelma* koostuu joukosta muuttujia $\mathbf{x} = \{x_1, \ldots, x_n\}$, joukosta lineaarisia epäyhtälöitä muuttujien suhteen ja lineaarisesta *kohdefunktiosta* $obj(\mathbf{x})$. Lineaarisen ohjelman *ratkaisu* $\hat{\mathbf{x}}$ antaa muuttujille arvot, jotka toteuttavat kaikki epäyhtälöt. Käytämme ratkaisun $\hat{\mathbf{x}}$ muuttujalle $x_i \in \mathbf{x}$ antamasta arvosta merkintää $\hat{x}_i$. Ratkaisu, jolla kohdefunktio saa pienimmän mahdollisen arvon on *optimaalinen ratkaisu*. Optimaalisen ratkaisun antama kohdefunktion arvo on lineaarisen ohjelman *optimaalinen arvo*. *Kokonaislukuohjelma* on lineaarinen ohjelma, jossa muuttujat saavat vain kokonaislukuarvoja.

Lineaarinen optimointi löytää optimaalisen ratkaisun lineaariselle ohjelmalle. Se on mahdollista tehdä polynomisessa ajassa [18], mutta *kokonaislukuoptimointi* on NP-kova ongelma [19]. Kokonaislukuoptimoinnin NP-kovuus ei ole yllättävää, koska monen NP-täydellisen ongelman muotoilu kokonaislukuohjelmana on suoraviivaista. Myös solmupeiteongelma kuuluu näihin ongelmiin. Lause 7 määrittelee solmupeitteen optimointiongelman kokonaislukuohjelmana.

Lause 7. *Kokonaislukuohjelman* IP-VC(G)*, jonka määrittelevät ehdot 1–4:*

$$\begin{aligned} \textit{minimoi} \quad & \sum_{v \in V(G)} x_v & (1) \\ \textit{ehdoilla} \quad & x_u + x_v \geq 1, \quad \forall (u,v) \in E(G), & (2) \\ & x_v \geq 0, \quad \forall v \in V(G), & (3) \\ & x_v \in \mathbb{Z}, \quad \forall v \in V(G), & (4) \end{aligned}$$

optimaalinen arvo on verkon G pienimmän solmupeitteen koko.

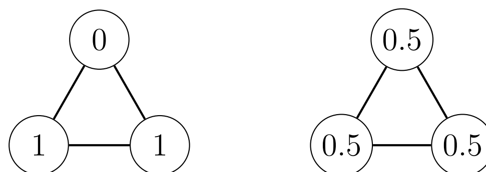
Todistus. IP-VC(G):n muuttujat $\mathbf{x} = \{x_v \mid v \in V(G)\}$ vastaavat G:n solmuja. Ratkaisu $\hat{\mathbf{x}}$ mallintaa solmupeitettä ottamalla solmun v solmupeitteesen, jos $\hat{x}_v \geq 1$. Ehdon 2 epäyhtälö takaa, että ainakin toinen kaaren (u, v) päätepisteistä on valittu solmupeitteeseen. Ehdot 3 ja 4 varmistavat, että kaikki muuttujat saavat optimaalisessa ratkaisussa arvon 0 tai 1. Kohdefunktio kuvaa siis suoraan solmupeitteen kokoa, joten optimaalinen ratkaisu vastaa G:n pienintä solmupeitettä. □

Ehto 4 tekee IP-VC(G):stä kokonaislukuohjelman. Kun se poistetaan, ehdot 1–3 määrittelevät lineaarisen ohjelman LP-VC(G). Näin saatua lineaarista ohjelmaa kutsutaan kokonaislukuohjelman *höllennykseksi.*

Lemma 3. *Kokonaislukuohjelman höllennyksen optimaalinen arvo on korkeintaan alkuperäisen kokonaislukuohjelman optimaalinen arvo.*

Todistus. Alkuperäisen kokonaislukuohjelman optimaalinen ratkaisu on myös höllennyksen ratkaisu, koska höllennys ei lisää ehtoja ratkaisulle. □

Päinvastainen tulos ei päde, eli höllennyksen optimaalinen arvo voi olla pienempi kuin alkuperäisen kokonaislukuohjelman optimaalinen arvo. Solmupeiteongelman tapauksessa pienin esimerkki tästä on kuvassa 5. Kolmioverkolla IP-VC(G):n optimaalinen arvo on 2, mutta sen höllennyksen optimaalinen ratkaisu saa arvon 1.5 asettamalla jokaiselle muuttujalle arvon 0.5. Muuttujien asettelu, joka antaa jokaiselle muuttujalle arvon 0.5, on LP-VC(G):n ratkaisu millä tahansa verkolla G. Nemhauser–Trotterin palautus pienentää solmupeiteongelman instanssia tarkalleen silloin, kun tämä ei ole optimaalinen ratkaisu.



Kuva 5: Esimerkit optimaalisista ratkaisuista IP-VC(G):lle ja LP-VC(G):lle kun G on kolmioverkko.

4.2 Solmupeite lineaarisena ohjelmana

Nemhauser–Trotterin palautus käyttää hyväksi LP-VC(G):n optimaalisten ratkaisujen rakennetta. Lineaarisen ohjelman *ääripiste* on ratkaisu $\hat{\mathbf{x}}$, jota ei voi esittää kahden muun ratkaisun, $\mathbf{y}$ ja $\mathbf{z}$, konveksina kombinaationa $\lambda\mathbf{y} + (1 - \lambda)\mathbf{z}$, jossa $0 < \lambda < 1$. Voimme olettaa, että LP-VC(G):n optimaalinen ratkaisu on ääripiste, koska lineaarisella ohjelmalla on optimaalinen ratkaisu, joka on ääripiste [15], ja tunnettujen lineaarisen optimoinnin algoritmien tuottamat ratkaisut ovat ääripisteitä [18]. Kuvassa 6 on esimerkki LP-VC(G):n optimaalisesta ratkaisusta, joka ei ole ääripiste, ja optimaalisesta ratkaisusta, joka on ääripiste. Geometrisesti tulkittuna ääripiste on lineaarisen ohjelman epäyhtälöjen muodostaman $|\mathbf{x}|$-ulotteisen monitahokkaan kärki.

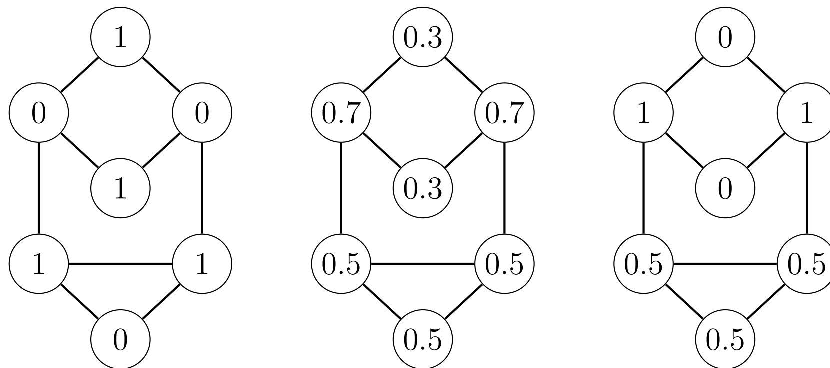
Lause 8 ([25])**.** *Jos* $\hat{\mathbf{x}}$ *on* LP-VC(G)*:n ääripiste, niin* $\hat{x}_v \in \{0, 0.5, 1\}$ *kaikilla* v*.*

Todistus. Esitämme ratkaisun $\hat{\mathbf{x}}$, jolla ei päde, että $\hat{x}_v \in \{0, 0.5, 1\}$, kahden ratkaisun konveksina kombinaationa. Olkoon $A = \{v \mid 0 < \hat{x}_v < 0.5\}$ ja $B = \{v \mid 0.5 < \hat{x}_v < 1\}$. Olkoon $d(\hat{x}_v) = \min\{|\hat{x}_v - p| \mid p \in \{0, 0.5, 1\}\}$ muuttujan arvon etäisyys lähimpään luvuista $\{0, 0.5, 1\}$. A:ssa ja B:ssä olevien solmujen arvoja voi muuttaa molempiin suuntiin vakiolla $\varepsilon = \min\{d(\hat{x}_v) \mid v \in A \cup B\}$. Rakennamme muuttujien asettelut $\mathbf{y}$ ja $\mathbf{z}$, joilla

$$y_v = \begin{cases} \hat{x}_v - \varepsilon & \text{jos } v \in A \\ \hat{x}_v + \varepsilon & \text{jos } v \in B \\ \hat{x}_v & \text{muulloin} \end{cases} \quad \text{ja} \quad z_v = \begin{cases} \hat{x}_v + \varepsilon & \text{jos } v \in A \\ \hat{x}_v - \varepsilon & \text{jos } v \in B \\ \hat{x}_v & \text{muulloin} \end{cases}.$$

Näemme tapaustarkastelulla, että $\mathbf{y}$ ja $\mathbf{z}$ toteuttavat LP-VC(G):n epäyhtälöt eli ovat sen ratkaisuja. Erityisesti A:n solmujen naapurit ovat B:ssä tai saavat arvon 1, ja B:n solmujen naapurit ovat A:ssa tai saavat arvon, joka on vähintään 0.5. Siispä $\hat{\mathbf{x}}$ ei ole ääripiste, koska $\hat{\mathbf{x}} = 0.5\mathbf{y} + 0.5\mathbf{z}$. □

Seuraus 3. *Voimme olettaa, että jos* $\mathbf{x}^*$ *on* LP-VC(G)*:n optimaalinen ratkaisu, niin* $x^*_v \in \{0, 0.5, 1\}$ *kaikilla* v*.*



Kuva 6: Esimerkkejä LP-VC(G):n ratkaisuista. Vasemmalla ratkaisu, joka on ääripiste, muttei optimaalinen. Keskellä optimaalinen ratkaisu, joka ei ole ääripiste. Oikealla optimaalinen ratkaisu, joka on ääripiste.

Nemhauser–Trotterin palautus etsii LP-VC(G):n optimaalisen ratkaisun $\mathbf{x}^*$, valitsee solmut $H = \{v \mid x_v^* = 1\}$ solmupeitteeseen ja poistaa solmut $I = \{v \mid x_v^* = 0\}$. Jäljelle jäävät siis solmut, joiden arvo on 0.5. Nemhauser–Trotterin lauseen mukaan G:llä on pienin solmupeite, jossa on kaikki H:n solmut eikä yksikään I:n solmu [26]. Todistamme vahvemman version Nemhauser–Trotterin lauseesta, eli että (I, H) on kruunu. Nemhauser–Trotterin lause seuraa tästä luvussa 3 esiteltyjen kruunujen ominaisuuksien takia.

Lause 9 ([8])**.** *Jos* $\mathbf{x}^*$ *on* LP-VC(G)*:n optimaalinen ratkaisu, niin solmujoukot* $I = \{v \mid x_v^* = 0\}$ *ja* $H = \{v \mid x_v^* = 1\}$ *muodostavat kruunun* (I, H)*.*

Todistus. I:ssä olevan solmun v naapuri $u \in N(v)$ on H:ssa, koska muuten epäyhtälö $x_u + x_v \geq 1$ ei toteutuisi. Tästä seuraa, että I täyttää kruunun ehdot 1 ja 2, eli se on riippumaton joukko ja H erottaa sen muusta verkosta.

Osoitamme vastaoletuksella ehdon 3, eli että I:n ja H:n välillä on paritus, jonka koko on $|H|$. Oletetaan, että I:n ja H:n välillä ei ole paritusta, joka parittaa kaikki H:n solmut. Hallin lauseen mukaan nyt on olemassa joukko $H_0 \subset H$, jonka naapurusto joukossa I on liian pieni parituksen muodostamiseen, eli $|H_0| > |N(H_0) \cap I|$.

Rakennamme ratkaisun $\hat{\mathbf{x}}$, joka vähentää H_0:ssa olevien solmujen arvoa 0.5:llä ja kasvattaa $N(H_0) \cap I$:ssä olevien solmujen arvoa 0.5:llä. Toisin sanoen $\hat{x}_v = 0.5$ kaikille $v \in H_0 \cup (N(H_0) \cap I)$ ja $\hat{x}_v = x_v^*$ muille v. Ratkaisu $\hat{\mathbf{x}}$ toteuttaa LP-VC(G):n epäyhtälöt, koska se vähentää vain H_0:ssa olevien solmujen arvoja ja yksikään H_0:n naapuri ei saa arvoa 0, koska ratkaisu kasvattaa sen I:ssä olevien naapureiden arvoja. Ratkaisun $\hat{\mathbf{x}}$ arvo on pienempi kuin ratkaisun $\mathbf{x}^*$ arvo, koska $|H_0| > |N(H_0) \cap I|$ eli solmuja, joiden arvoa vähennetään, on enemmän kuin solmuja, joiden arvoa kasvatetaan. Siispä $\mathbf{x}^*$ ei ole LP-VC(G):n optimaalinen ratkaisu, mikä on ristiriita. □

Lause 10 ([6])**.** *Nemhauser–Trotterin palautus on ydinpalautus, joka tuottaa* $2k$ *solmun kokoisen ytimen.*

Todistus. Palautus toimii polynomisessa ajassa, koska lineaarinen optimointi toimii polynomisessa ajassa [18]. Lemman 3 mukaan LP-VC(G):n optimaalisen ratkaisun $\mathbf{x}^*$ arvo $obj(\mathbf{x}^*)$ on pienimmän solmupeitteen koon alaraja. Siispä palautus palauttaa *ei*, jos $obj(\mathbf{x}^*) > k$. Muussa tapauksessa palautus poistaa verkosta solmut, joiden arvo $\mathbf{x}^*$:ssä ei ole 0.5, eli jäljelle jääneille solmuille $R = V(G) \setminus (I \cup H)$ pätee $x_v^* = 0.5$ kaikilla $v \in R$. Koska $obj(\mathbf{x}^*)$ on muuttujien summa, tästä seuraa, että $0.5|R| \leq obj(\mathbf{x}^*)$ eli $|R| \leq 2obj(\mathbf{x}^*) \leq 2k$. □

4.3 Muotoilu maksimiparituksena

Kun Nemhauser–Trotterin palautus esiteltiin, ei vielä tunnettu polynomiaikaista algoritmia lineaariseen optimointiin. Jotta palautus voitaisiin tehdä polynomisessa ajassa, Nemhauser ja Trotter esittivät artikkelissaan polynomiaikaisen algoritmin LP-VC(G):n ratkaisemiseen [26]. Algoritmi on yksinkertainen, ja se on osoittautunut myös käytännölliseksi tavaksi toteuttaa Nemhauser–Trotterin palautus [1].

Algoritmi laskee pienimmän solmupeitteen kaksijakoisessa verkossa G', jossa on kaksi solmua jokaista G:n solmua kohti. Tarkemmin verkko on määritelty niin, että $V(G') = \{y_v \mid v \in V(G)\} \cup \{z_v \mid v \in V(G)\}$ ja $E(G') = \{(y_u, z_v) \mid (u,v) \in E(G)\}$. Algoritmi rakentaa LP-VC(G):n ratkaisun $\mathbf{x}'$ käyttäen verkon G':n pienintä solmupeitettä C' asettamalla

$$x'_v = \begin{cases} 1 & \text{jos } y_v, z_v \in C' \\ 0 & \text{jos } y_v, z_v \notin C' \\ 0.5 & \text{muulloin} \end{cases} .$$

Toisin sanoen x'_v:n arvo riippuu siitä, kuinka monta v:tä vastaavaa solmua pienimmässä solmupeitteessä on.

Lause 11. $\mathbf{x}'$ *on* LP-VC(G)*:n optimaalinen ratkaisu.*

Todistus. Osoitamme, että $\mathbf{x}'$ on LP-VC(G):n ratkaisu. Jos $x'_v = 0$, niin y_v ja z_v eivät kuulu solmupeitteeseen C'. Tällöin jos $(u,v) \in E(G)$, niin $y_u \in C'$, koska $(y_u, z_v) \in E(G')$, ja $z_u \in C'$, koska $(y_v, z_u) \in E(G')$, joten $x'_u = 1$. Symmetrian nojalla ratkaisu toteuttaa kaikki epäyhtälöt.

Toiseen suuntaan osoitamme, että mikä tahansa LP-VC(G):n optimaalinen ratkaisu $\mathbf{x}^*$ vastaa G':n solmupeitettä C^*. Rakennamme joukon C^* niin, että $y_v \in C^*$, jos $x^*_v \geq 0.5$, ja $z_v \in C^*$, jos $x^*_v = 1$. Jos C^* ei peittäisi kaarta $(y_u, z_v) \in E(G')$, niin $x^*_u = 0$ ja $x^*_v \leq 0.5$, jolloin epäyhtälö $x_u + x_v \geq 1$ ei toteutuisi. □

Lause 12. *Nemhauser–Trotterin palautus toimii ajassa* $O(\sqrt{n}m)$.

Todistus. Palautus laskee LP-VC(G):n optimaalisen ratkaisun lauseen 11 mukaisesti käyttäen kaksijakoisen verkon pienintä solmupeitettä, jonka se laskee Kőnigin lauseen mukaisesti kaksijakoisen verkon maksimiparituksella ajassa $O(\sqrt{n}m)$. □

Nemhauser–Trotterin palautuksen ja luvussa 3 esitellyn kruunupalautuksen aikavaativuudet ovat samat. Nemhauser–Trotterin palautuksen takaama ydin on pienempi, joten teoreettisessa mielessä se on kaikin tavoin parempi kuin kruunupalautus. Käytännön toteutuksissa Nemhauser–Trotterin palautus voi kuitenkin olla hitaampi, koska verkko, jossa maksimiparitus lasketaan, voi olla suurempi [1].

5 Approksimointialgoritmit

Parametrisoitujen algoritmien lisäksi toinen lähestymistapa NP-koviin ongelmiin ovat *approksimointialgoritmit*. Approksimointialgoritmit toimivat polynomisessa ajassa, mutta eivät löydä välttämättä optimaalista ratkaisua. Algoritmi on α-*approksimointialgoritmi*, jos se löytää aina ratkaisun, jonka arvo on korkeintaan kertoimen α päässä optimaalisesta arvosta. Kutsumme tällaista ratkaisua α-*approksimoinniksi.*

Solmupeiteongelman tapauksessa α-approksimointialgoritmi löytää korkeintaan αr-kokoisen solmupeitteen, kun pienimmän solmupeitteen koko on r. Todistamme tässä luvussa, että algoritmista, joka löytää kruunun, voi muokata approksimointialgoritmin solmupeiteongelmalle. Sen lisäksi, että tästä tuloksesta seuraa 2-approksimointialgoritmi solmupeiteongelmalle, saamme myös näyttöä Nemhauser–Trotterin palautuksen optimaalisuudesta.

5.1 Kruunupalautus approksimointialgoritmina

Luvussa 3 esitelty kruunupalautus ja luvussa 4 esitelty Nemhauser–Trotterin palautus löytävät verkosta kruunun ja poistavat sen. Hyödyllinen kruunuihin perustuvien palautusten ominaisuus on, että ne säilyttävät approksimoinnin, eli approksimointi palautuksen tuottaman instanssin ratkaisulle ei huononnu, jos sitä soveltaa alkuperäiseen instanssiin. Lemma 4 formalisoi tämän.

Lemma 4. *Jos verkossa G on kruunu (I, H) ja C on verkon $G\backslash(I \cup H)$ solmupeitteen α-approksimointi, niin $C \cup H$ on G:n solmupeitteen α-approksimointi.*

Todistus. Olkoon G:n pienimmän solmupeitteen koko r. Lauseen 3 mukaan kruunupalautus säilyttää optimaalisuuden, joten $r - |H|$ on verkon $G \setminus (I \cup H)$ pienimmän solmupeitteen koko. Siispä $|C| \leq \alpha(r - |H|)$, joten $|C| + \alpha|H| \leq \alpha r$. Koska $\alpha > 1$, arvioimme vasenta puolta alaspäin ja saamme $|C| + |H| \leq \alpha r$. □

Käytämme tätä ominaisuutta rakentamaan mistä tahansa kruunun löytävästä algoritmista approksimointialgoritmin. Olkoon Kruunu algoritmi, joka löytää $n - \alpha k$ -kokoisen kruunun. Rakennamme approksimointialgoritmin Kruunu_A poistamalla verkosta kruunun niin kauan, kuin algoritmi Kruunu löytää sellaisen. Kun palautusta ei voi enää käyttää, niin Kruunu_A valitsee kaikki jäljellä olevat solmut solmupeitteeseen.

Lause 13. *Jos* Kruunu *on polynomisessa ajassa toimiva algoritmi, joka löytää vähintään $n - \alpha k$ -kokoisen kruunun tai ilmoittaa, että pienin solmupeite on suurempi kuin k, niin* Kruunu_A *on α-approksimointialgoritmi solmupeiteongelmalle.*

Todistus. Tarkastelemme Kruunu_A:n toimintaa, kun se tekee palautuksen p kertaa. Merkitään syötteenä olevaa verkkoa G_0:lla ja iteraatioilla $1, \ldots, p$ tuotettuja verkkoja $G_1, \ldots, G_p$. Iteraatiolla i algoritmi Kruunu_A etsii pienimmän k, jolla Kruunu palauttaa kruunun (I_i, H_i) syötteellä (G_{i-1}, k) ja asettaa $G_i = G_{i-1} \setminus (I_i \cup H_i)$.

Todistamme, että $H_1 \cup \ldots \cup H_p \cup V(G_p)$ on α-approksimointi G:n solmupeitteelle. Käytämme induktiota, jossa verkko G_p on perustapaus ja induktioaskel käyttää lemmaa 4. Koska Kruunu ei löytänyt verkolle G_p kruunua millään k, niin $n - \alpha k \leq 0$ kaikilla k, joilla G_p:llä on k:n kokoinen solmupeite. Toisin sanoen kun G_p:n pienimmän solmupeitteen koko on r_p, niin $|V(G_p)| \leq \alpha r_p$, eli $V(G_p)$ on α-approksimointi G_p:n solmupeitteelle. Lemman 4 mukaan jos $H_{i+1} \cup \ldots \cup H_p \cup V(G_p)$ on α-approksimointi G_i:n solmupeitteelle, niin $H_i \cup \ldots \cup H_p \cup V(G_p)$ on α-approksimointi G_{i-1}:n solmupeitteelle. Kruunu_A palauttaa solmupeitteen $H_1 \cup \ldots \cup H_p \cup V(G_p)$.

Kruunu$_A$ toimii polynomisessa ajassa, koska palautus tehdään korkeintaan n kertaa, jokaisella kerralla kutsutaan algoritmia Kruunu korkeintaan n kertaa ja Kruunu toimii polynomisessa ajassa. □

Käytännössä tässä tutkielmassa esiteltyjä algoritmeja ei tarvitse kutsua monta kertaa löytääkseen pienimmän k:n, jolla palautus ei palauta *ei*, koska algoritmeja voi muokata niin, että ne itse laskevat pienimmän mahdollisen arvon k. Esimerkiksi Nemhauser–Trotterin palautuksessa tällainen k on LP-VC(G):n optimaalinen arvo.

Seuraus 4. *Solmupeiteongelmalle on* 2*-approksimointialgoritmi.*

Todistus. Luvun 4 mukaan Nemhauser–Trotterin palautus löytää polynomisessa ajassa kruunun, jonka koko on $n - 2k$, tai ilmoittaa, että pienin solmupeite on suurempi kuin k. Lauseen 13 perusteella Nemhauser–Trotterin palautusta voidaan käyttää rakentamaan 2-approksimointialgoritmi. □

5.2 Näyttö optimaalisuudesta

Lausetta 13 voi käyttää myös toiseen suuntaan, eli osoittamaan, että suuria kruunuja on vaikea löytää. Keskeinen konjektuuri approksimointialgoritmien tutkimuksessa on *uniikkien pelien konjektuuri* [20], johon pohjautuvia approksimoinnin vaikeustuloksia on todistettu usealle ongelmalle [21]. Khot ja Regev osoittivat vuonna 2008, että solmupeiteongelmaan ei ole $2 - \varepsilon$ -approksimointialgoritmia millään $\varepsilon > 0$, jos uniikkien pelien konjektuuri on totta [22]. Lauseen 13 perusteella tästä seuraa:

Seuraus 5. *Jos uniikkien pelien konjektuuri on totta, niin ei ole algoritmia, joka löytää polynomisessa ajassa vähintään $n - (2 - \varepsilon)k$ -kokoisen kruunun tai ilmoittaa, että verkon pienin solmupeite on suurempi kuin k.*

Uniikkien pelien konjektuuria ei ole onnistuttu todistamaan tai kumoamaan, mutta sen ratkeaminen kumpaan suuntaan tahansa olisi hyvin merkittävä tulos [21]. Konjektuurin paikkansapitävyys tarkoittaisi, että Nemhauser–Trotterin palautus olisi paras kruunuihin perustuva ydinpalautus solmujen määrän suhteen. Vaikka tämän luvun tulokset koskevat vain kruunuihin perustuvia palautuksia, on vaikea kuvitella solmupeiteongelman ydinpalautusta, joka ei säilyttäisi approksimointia. Näyttö optimaalisuudesta yleistyy tarkoittamaan kaikkia solmupeiteongelman ydinpalautuksia, jos oletamme, että ne kaikki säilyttävät approksimoinnin.

6 Laskennalliset kokeet

Algoritmeja solmupeiteongelmaan on tutkittu myös kokeellisesti [1, 2, 14, 26]. Sekä kruunupalautuksella että Nemhauser–Trotterin palautuksella on tehty laskennallisia kokeita [1, 26], mutta kokeissa joko syötteinä käytetyt verkot olivat satunnaisgeneroituja [26] tai testiaineistona oli vain 4 verkkoa [1]. Tässä luvussa vertaam-

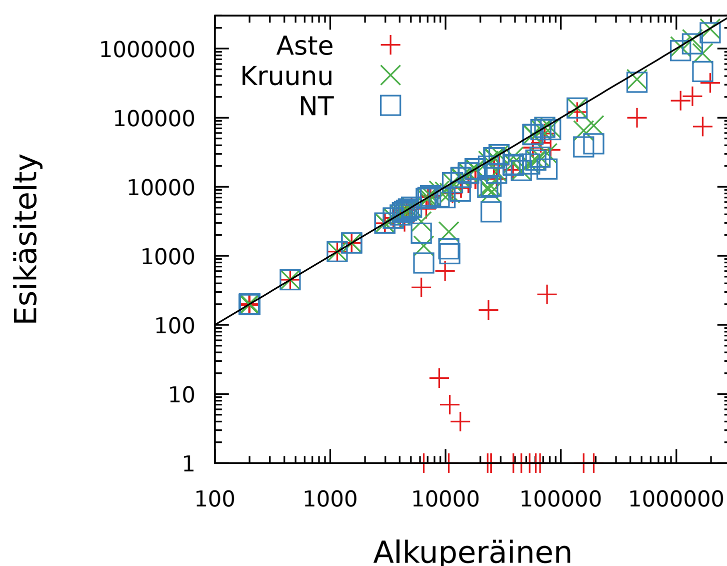
me tutkielmassa esiteltyjä algoritmeja kokeellisesti käyttäen instansseja, jotka pohjautuvat käytännön sovelluksiin. Toteutimme vertailua varten kruunupalautuksen, Nemhauser–Trotterin palautuksen ja palautussäännön, joka ei tee ydinpalautusta. Analysoimme algoritmien tekemää instanssien pienennystä ja suoritusaikoja.

6.1 Koeasetelma

Vertaamme kolmea eri algoritmia solmupeiteongelman esikäsittelyyn. Algoritmit ovat luvussa 2 kuvattu 1- ja 2-asteiset solmut poistava palautus (Aste), luvussa 3 kuvattu kruunupalautus (Kruunu) ja luvussa 4 kuvattu Nemhauser–Trotterin palautus (NT). Algoritmit saavat syötteenä verkon ja palauttavat pienennetyn version kyseisestä verkosta ja tiedon siitä, kuinka monta solmua on jo valittu solmupeitteeseen. Algoritmit eivät ota syötteenä lukua k vaan toimivat siitä riippumatta. Ne ovat siis käytännöllisiä esikäsittelyalgoritmeja, joita voisi käyttää itsenäisenä vaiheena ennen mitä tahansa solmupeiteongelman ratkaisevaa algoritmia. Kruunu ja NT tuottavat myös alarajan pienimmän solmupeitteen koolle.

Toteutimme algoritmit C++-kielellä, ja toteutuksemme ovat saatavilla julkisesti[1]. NT käyttää aliluvussa 4.3 kuvattua maksimiparitukseen pohjautuvaa algoritmia LP-VC(G):n ratkaisemiseen. Maksimiparituksen laskemiseen algoritmit Kruunu ja NT käyttävät Hopcroft-Karp-algoritmia [16]. Kokeet ajettiin laskentasolmuilla, joilla on Intel Xeon E5-2680-v4-prosessorit.

[1] `https://github.com/Laakeri/kandi`

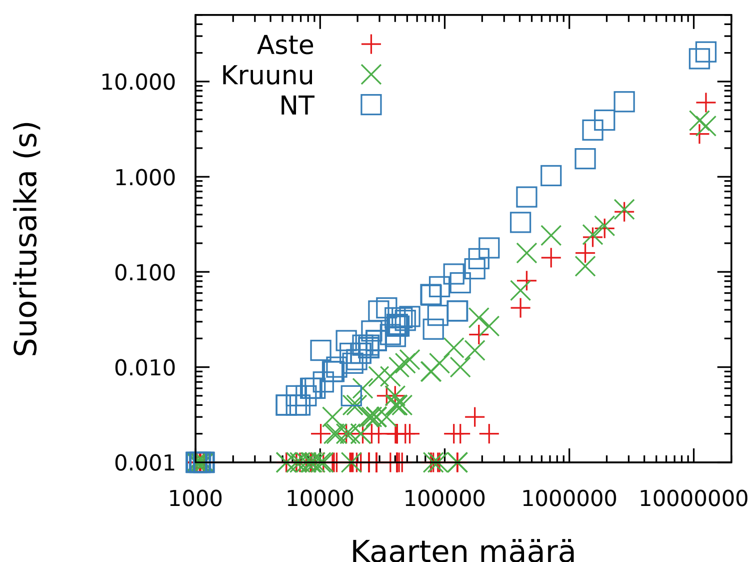


Kuva 7: Vertailu algoritmien tuottamien esikäsiteltyjen instanssien koosta. Pisteen x-koordinaatti vastaa instanssin alkuperäistä solmujen määrää ja y-koordinaatti solmujen määrää esikäsittelyn jälkeen.

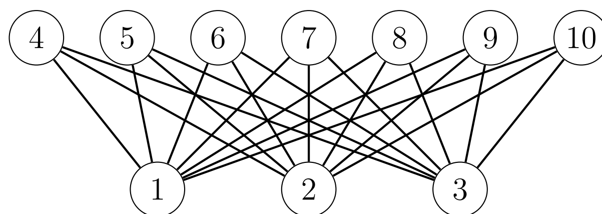
Kokeiden testiaineistona oli verkkoja PACE 2019 -kilpailusta solmupeiteongelmalle [14] ja SNAP-kokoelmasta [24]. PACE on algoritmien toteutuskilpailu, jonka tavoitteena on tuoda parametrisoitua algoritmiikkaa lähemmäksi käytännön sovelluksia [11]. SNAP-kokoelman verkot perustuvat käyttäjien välisiin suhteisiin sosiaalisessa mediassa ja tieverkkoihin Yhdysvalloissa, ja niitä on käytetty ennekin vertailemaan algoritmeja solmupeiteongelmalle [2].

6.2 Tulokset

Testasimme algoritmeja kaikilla PACE 2019 -kilpailun julkisilla solmupeiteongelman instansseilla, joita on 100, ja 15:llä SNAP-kokoelman eri kategorioista valitulla instanssilla. Algoritmit Kruunu ja NT pienensivät näistä 28 instanssia, ja Aste pienensi 95 instanssia. Liitteenä oleva taulukko esittää tarkat testitulokset instanssikohtaisesti. Tulokset osoittavat, että algoritmien välillä on selkeä paremmuusjärjestys, koska Aste pienensi instansseja aina vähintään yhtä paljon kuin NT ja NT vähintään yhtä paljon kuin Kruunu. Kuva 7 näyttää vertailun algoritmin tuottaman esikäsitellyn verkon ja syötteenä olleen verkon kokojen välillä. Aste oli ainoa algoritmi, joka pystyi pienentämään syötteenä olevia verkkoja alle 100 solmun kokoisiksi. Kuvassa 8 on vertailu algoritmien suoritusajoista testatuilla instansseilla. Kruunu on aina nopeampi kuin NT, vaikka molempien aikavaativuus on $O(\sqrt{n}m)$. Aste toimii lineaariajassa ja osoittautui vain neljässä tapauksessa hitaammaksi kuin Kruunu. Solmujen määrältä suurimmalla instanssilla, `snap_roadnet_ca`, jossa on 1 965 206 solmua ja 2 766 607 kaarta, algoritmien Aste, Kruunu ja NT suoritusajat olivat 0.428 s, 0.454 s ja 6.132 s.



Kuva 8: Vertailu algoritmien suoritusajoista suhteessa kaarten määrään.



Kuva 9: Esimerkki verkosta, jota Aste ei pienennä, mutta Kruunu ja NT pienentävät.

Kokeelliset tulokset olivat jossain määrin yllättäviä, koska teoreettisesti hyvin toimivat algoritmit Kruunu ja NT olivat kaikin tavoin huonompia kuin Aste, jonka toiminnasta ei ole mitään teoreettista takuuta. Tällaisia tuloksia ei ole saatu aiemmissa kokeissa [1]. Olisi helppoa rakentaa keinotekoisesti verkko, jossa Kruunu ja NT toimivat hyvin, mutta Aste ei toimi. Esimerkiksi kuvassa 9 oleva täydellinen kaksijakoinen verkko, jossa on toisella puolella 3 solmua ja toisella puolella 7 solmua on sellainen. Vaikuttaa kuitenkin siltä, että kaikilla testatuilla verkoilla pienin solmupeite on suhteellisen suuri, joten Kruunu ja NT eivät hyödy teoreettisista ominaisuuksistaan.

7 Yhteenveto

Olemme esitelleet kolme algoritmia ydinpalautuksen tekemiseen solmupeiteongelmalle, kun ongelma on parametrisoitu halutun solmupeitteen koon k mukaan. Algoritmeista kaksi perustuu kruunun löytämiseen verkosta. Kruunupalautus löytää kruunun maksimaalisen parituksen avulla ja tuottaa ytimen, jonka koko on korkeintaan $3k$. Nemhauser–Trotterin palautus käyttää lineaarista optimointia kruunun löytämiseen ja tuottaa ytimen, jonka koko on korkeintaan $2k$. Molemmat algoritmit toimivat ajassa $O(\sqrt{n}m)$, mutta laskennallisissa kokeissa Nemhauser–Trotterin palautus osoittautui hitaammaksi.

Kruunupalautuksen ja Nemhauser–Trotterin palautuksen esittelyn ja teoreettisen analyysin jälkeen näytimme, että niitä voi käyttää approksimointialgoritmien rakentamiseen, minkä avulla johdimme 2-approksimointialgoritmin solmupeiteongelmalle. Yhteys approksimointialgoritmeihin antaa myös teoreettista näyttöä sille, ettei Nemhauser–Trotterin palautusta parempaa kruunuihin perustuvaa ydinpalautusta olisi olemassa. Laskennalliset kokeet osoittivat, että palautuksien toteutukset ovat käytännöllisiä. Ne toimivat muutamissa sekunneissa verkoilla, joissa on miljoonia solmuja ja kaaria. Kuitenkin yksinkertaisempi palautussääntö, joka ei takaa ydinpalautusta, pienensi kaikkia testattuja instansseja enemmän kuin kruunupalautus ja Nemhauser–Trotterin palautus.

Emme pystyneet osoittamaan, että ydinpalautukset solmupeiteongelmalle olisivat hyödyllisiä käytännön tapauksissa. Yksi tämän selittävä näkökulma on, että solmupeiteongelman instanssit, joissa pienimmän solmupeitteen koko on pieni suhteessa solmujen määrään, eivät ole kovin kiinnostavia. Tällöin ydinpalautusten rooli olisi

formalisoida näiden tapausten helppous, mutta käytännössä nämä tapaukset olisivat helppoja myös muille algoritmeille. Teoreettisesti kuitenkin kruunuihin perustuvien palautusten rooli parametrisoitujen algoritmien kehityksessä on ollut merkittävä, ja Nemhauser–Trotterin palautus antaa käytännössä täydellisen vastauksen kysymykseen parhaasta ydinpalautuksesta solmupeiteongelmalle.

Lähteet

[1] F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters ja C. T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. *6th Workshop on Algorithm Engineering and Experiments*, 62–69, 2004.

[2] T. Akiba ja Y. Iwata. Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theoretical Computer Science*, 609:211–225, 2016.

[3] H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. *International Colloquium on Automata, Languages, and Programming*, LNCS 317, 105–118, 1988.

[4] J. F. Buss ja J. Goldsmith. Nondeterminism within P^*. *SIAM Journal on Computing*, 22(3):560–572, 1993.

[5] L. Cai, J. Chen, R. G. Downey ja M. R. Fellows. Advice classes of parameterized tractability. *Annals of Pure and Applied Logic*, 84(1):119–138, 1997.

[6] J. Chen, I. A. Kanj ja W. Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001.

[7] J. Chen, I. A. Kanj ja G. Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40-42):3736–3756, 2010.

[8] M. Chlebík ja J. Chlebíková. Improvement of Nemhauser-Trotter theorem and its applications in parametrized complexity. *9th Scandinavian Workshop on Algorithm Theory*, LNCS 3111, 174–186, 2004.

[9] B. Chor, M. Fellows ja D. Juedes. Linear kernels in linear time, or how to save k colors in $O(n^2)$ steps. *International Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 3353, 257–269, 2004.

[10] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk ja S. Saurabh. *Parameterized Algorithms*. Springer, 2015.

[11] H. Dell, C. Komusiewicz, N. Talmon ja M. Weller. The PACE 2017 parameterized algorithms and computational experiments challenge: The second iteration. *12th International Symposium on Parameterized and Exact Computation*, 30:1–30:12, 2017.

[12] H. Dell ja D. van Melkebeek. Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. *Journal of the ACM*, 61(4):23:1–27, 2014.

[13] M. R. Fellows. Blow-ups, win/win's, and crown rules: Some new directions in *FPT*. *International Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 2880, 1–12, 2003.

[14] J. Fichte ja M. Hecher. Parameterized algorithms and computational experiments challenge 2019. `https://pacechallenge.org/2019/`, 10.4.2019.

[15] R. E. Gomory. Some polyhedra related to combinatorial problems. *Linear Algebra and its Applications*, 2(4):451–558, 1969.

[16] J. E. Hopcroft ja R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[17] B. M. P. Jansen ja H. L. Bodlaender. Vertex cover kernelization revisited. *Theory of Computing Systems*, 53(2):263–299, 2013.

[18] N. Karmarkar. A new polynomial-time algorithm for linear programming. *16th Annual ACM Symposium on Theory of Computing*, 302–311, 1984.

[19] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 85–103, 1972.

[20] S. Khot. On the power of unique 2-prover 1-round games. *34th Annual ACM Symposium on Theory of Computing*, 767–775, 2002.

[21] S. Khot. On the unique games conjecture. *25th Annual IEEE Conference on Computational Complexity*, 99–121, 2010.

[22] S. Khot ja O. Regev. Vertex cover might be hard to approximate to within $2-\varepsilon$. *Journal of Computer and System Sciences*, 74(3):335–349, 2008.

[23] S. Kratsch ja M. Wahlström. Representative sets and irrelevant vertices: New tools for kernelization. *53rd Annual Symposium on Foundations of Computer Science*, 450–459, 2012.

[24] J. Leskovec ja A. Krevl. SNAP datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

[25] G. L. Nemhauser ja L. E. Trotter Jr. Properties of vertex packing and independence system polyhedra. *Mathematical Programming*, 6:48–61, 1974.

[26] G. L. Nemhauser ja L. E. Trotter Jr. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.

[27] M. Xiao ja H. Nagamochi. Exact algorithms for maximum independent set. *Information and Computation*, 255:126–146, 2017.

Liite 1. Laskennalliset kokeet instanssikohtaisesti

Laskennallisten kokeiden tulokset. Taulukko ilmoittaa instanssin solmujen (n) ja kaarten (m) määrän, kuinka paljon testatut algoritmit pienensivät solmujen määrää sekä algoritmien suoritusajat.

	Instanssin koko		Pienennys			Suoritusaika (s)		
Instanssi	n	m	Aste	Kruunu	NT	Aste	Kruunu	NT
pace_001	6160	40207	5810	3022	4035	0.002	0.004	0.021
pace_003	60541	48418	60541	36163	36163	0.002	0.011	0.031
pace_005	200	798	4	0	0	0.000	0.000	0.000
pace_007	8794	10121	8777	61	1696	0.002	0.001	0.015
pace_009	38452	174645	20911	16707	17467	0.003	0.015	0.108
pace_011	9877	25973	9272	1433	2939	0.002	0.003	0.024
pace_013	45307	36697	45307	28089	28089	0.001	0.008	0.022
pace_015	53610	42940	53610	32272	32272	0.001	0.010	0.027
pace_017	23541	34233	23377	25	3614	0.005	0.003	0.042
pace_019	200	862	4	0	0	0.000	0.000	0.000
pace_021	24765	19655	24765	14547	14547	0.001	0.004	0.012
pace_023	27717	133665	14736	11163	12007	0.002	0.010	0.077
pace_025	23194	18295	23194	13398	13398	0.001	0.004	0.011
pace_027	65866	52435	65866	39006	39006	0.002	0.012	0.034
pace_029	13431	16234	13427	20	4776	0.002	0.002	0.019
pace_031	200	813	0	0	0	0.000	0.000	0.000
pace_033	4410	6885	1290	0	0	0.000	0.001	0.004
pace_035	200	864	2	0	0	0.000	0.000	0.000
pace_037	198	808	2	0	0	0.000	0.000	0.000
pace_039	6795	10620	1980	0	0	0.000	0.001	0.007
pace_041	200	1023	0	0	0	0.000	0.000	0.000
pace_043	200	841	0	0	0	0.000	0.000	0.000
pace_045	200	1020	0	0	0	0.000	0.000	0.000
pace_047	200	1093	2	0	0	0.000	0.000	0.001
pace_049	200	933	2	0	0	0.000	0.000	0.000
pace_051	200	1098	0	0	0	0.000	0.000	0.001
pace_053	200	1026	0	0	0	0.000	0.000	0.000
pace_055	200	938	2	0	0	0.000	0.000	0.000
pace_057	200	1160	0	0	0	0.000	0.000	0.001
pace_059	200	961	2	0	0	0.000	0.000	0.000
pace_061	200	931	2	0	0	0.000	0.000	0.000
pace_063	200	1011	0	0	0	0.000	0.000	0.000
pace_065	200	1011	0	0	0	0.000	0.000	0.000
pace_067	200	1174	0	0	0	0.000	0.000	0.001
pace_069	200	1083	4	0	0	0.000	0.000	0.001
pace_071	200	952	0	0	0	0.000	0.000	0.000
pace_073	200	1078	0	0	0	0.000	0.000	0.000
pace_075	26300	41500	7400	0	0	0.001	0.004	0.027
pace_077	200	961	2	0	0	0.000	0.000	0.000
pace_079	26300	41500	7400	0	0	0.001	0.004	0.028
pace_081	199	1091	2	0	0	0.000	0.000	0.001
pace_083	200	1172	0	0	0	0.000	0.000	0.001
pace_085	11470	17408	3488	0	0	0.000	0.002	0.014
pace_087	13590	21240	3960	0	0	0.001	0.002	0.014
pace_089	57316	77978	20202	0	0	0.001	0.009	0.057
pace_091	200	1163	0	0	0	0.000	0.000	0.001
pace_093	200	1162	0	0	0	0.000	0.000	0.001
pace_095	15783	24663	4602	0	0	0.001	0.003	0.016
pace_097	18096	28281	5274	0	0	0.001	0.003	0.019
pace_099	26300	41500	7400	0	0	0.001	0.004	0.028
pace_101	26300	41500	7400	0	0	0.002	0.004	0.028
pace_103	15783	24663	4602	0	0	0.001	0.003	0.016
pace_105	26300	41500	7400	0	0	0.002	0.004	0.027
pace_107	13590	21240	3960	0	0	0.001	0.002	0.014
pace_109	66992	90970	23672	0	0	0.001	0.011	0.070
pace_111	450	17831	0	0	0	0.000	0.000	0.005
pace_113	26300	41500	7400	0	0	0.002	0.004	0.028
pace_115	18096	28281	5274	0	0	0.001	0.003	0.019

	Instanssin koko		Pienennys			Suoritusaika (s)		
Instanssi	n	m	Aste	Kruunu	NT	Aste	Kruunu	NT
pace_117	18096	28281	5274	0	0	0.001	0.003	0.019
pace_119	18096	28281	5274	0	0	0.001	0.003	0.019
pace_121	18096	28281	5274	0	0	0.001	0.003	0.019
pace_123	26300	41500	7400	0	0	0.002	0.004	0.028
pace_125	26300	41500	7400	0	0	0.001	0.004	0.028
pace_127	18096	28281	5274	0	0	0.001	0.003	0.019
pace_129	15783	24663	4602	0	0	0.001	0.003	0.016
pace_131	2980	5360	12	0	0	0.000	0.001	0.004
pace_133	15783	24663	4602	0	0	0.001	0.003	0.016
pace_135	26300	41500	7400	0	0	0.001	0.004	0.028
pace_137	26300	41500	7400	0	0	0.001	0.004	0.028
pace_139	18096	28281	5274	0	0	0.001	0.003	0.019
pace_141	18096	28281	5274	0	0	0.001	0.003	0.019
pace_143	18096	28281	5274	0	0	0.001	0.003	0.019
pace_145	18096	28281	5274	0	0	0.001	0.003	0.019
pace_147	18096	28281	5274	0	0	0.001	0.003	0.019
pace_149	26300	41500	7400	0	0	0.001	0.004	0.028
pace_151	15783	24663	4602	0	0	0.001	0.003	0.017
pace_153	29076	45570	8196	0	0	0.001	0.004	0.033
pace_155	26300	41500	7400	0	0	0.001	0.004	0.028
pace_157	2980	5360	25	0	0	0.000	0.001	0.004
pace_159	18096	28281	5274	0	0	0.001	0.003	0.019
pace_161	138141	227241	17446	0	0	0.002	0.027	0.179
pace_163	18096	28281	5274	0	0	0.001	0.003	0.019
pace_165	18096	28281	5274	0	0	0.001	0.003	0.019
pace_167	15783	24663	4602	0	0	0.001	0.003	0.016
pace_169	4768	8576	56	0	0	0.000	0.001	0.006
pace_171	18096	28281	5274	0	0	0.001	0.003	0.019
pace_173	56860	77264	20068	0	0	0.001	0.009	0.058
pace_175	3523	6446	2	0	0	0.000	0.001	0.005
pace_177	5066	9112	20	0	0	0.000	0.001	0.006
pace_179	15783	24663	4602	0	0	0.001	0.003	0.016
pace_181	18096	28281	5274	0	0	0.001	0.003	0.019
pace_183	72420	118362	13324	0	0	0.002	0.016	0.095
pace_185	3523	6446	0	0	0	0.000	0.001	0.004
pace_187	4227	7734	6	0	0	0.000	0.001	0.005
pace_189	7400	13600	10	0	0	0.000	0.002	0.010
pace_191	4579	8378	2	0	0	0.000	0.001	0.006
pace_193	7030	12920	4	0	0	0.000	0.002	0.009
pace_195	1150	81068	0	0	0	0.000	0.000	0.025
pace_197	1534	127011	0	0	0	0.000	0.000	0.039
pace_199	1534	126163	0	0	0	0.000	0.000	0.039
snap_as20000102	6474	12572	6474	5083	5684	0.001	0.003	0.009
snap_as_skitter	1696415	11095298	1621903	844869	1230834	2.812	3.888	17.319
snap_facebook	4039	88234	575	98	123	0.001	0.001	0.035
snap_higgs_follow	456626	12508413	357033	97157	130632	6.005	3.414	20.475
snap_higgs_reply	38683	29552	38683	11489	18384	0.002	0.008	0.039
snap_oregon_010331	10670	22002	10670	8430	9405	0.002	0.006	0.017
snap_p2p_gnutella04	10876	39994	10869	2760	9802	0.005	0.005	0.033
snap_roadnet_ca	1965206	2766607	1646553	37568	270185	0.428	0.454	6.132
snap_roadnet_pa	1088092	1541898	911512	20122	155690	0.232	0.247	3.086
snap_roadnet_tx	1379917	1921660	1175418	24863	208495	0.286	0.305	3.925
snap_soc_epinions1	75879	405740	75602	45441	57799	0.042	0.064	0.332
snap_askubuntu	157222	455691	157222	92170	119464	0.081	0.158	0.613
snap_mathoverflow	24759	187986	24759	16487	20432	0.022	0.033	0.138
snap_superuser	192409	714570	192409	115643	150453	0.141	0.242	1.028
snap_twitter	81306	1342296	47090	10094	14067	0.158	0.115	1.547