Lyhimmän polun etsintä verkossa muistihierarkiatehokkaasti

Teppo Niinimäki

Helsinki 5.4.2009

Seminaari: Muistihierarkia-algoritmit

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Teppo Niinimäki

Työn nimi — Arbetets titel — Title
Lyhimmän polun etsintä verkossa muistihierarkiatehokkaasti

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminaariraportti	5.4.2009	13 sivua

Tiivistelmä — Referat — Abstract

Tässä raportissa tutustutaan verkon yksittäisestä solmusta lähtevän lyhimmän polun (SSSP) etsintään muistihierarkiassa. Tähän liittyen esitellään kaksi prioriteettijonon toteutusta: *turnauspuut* ja *ämpärikeko*. Näitä käytetään lyhimmän polun etsintään. Lopuksi luodaan katsaus algoritmeilla saavutettaviin käytännön tuloksiin.

ACM Computing Classification System (CCS):
F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—Sorting and searching
G.2.2 [Discrete Mathematics]: Graph Theory—Path and circuit problems

Avainsanat — Nyckelord — Keywords
muistihierarkia-algoritmit, lyhimmän polun etsintä

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Lyhimmän polun etsintä on eräs keskeisimpiä verkkoihin liittyviä ongelmia. Ilmeisiä käyttökohteita ovat reittien etsiminen fyysisten sijaintien välillä esimerkiksi erilaisissa internetin reitinhakupalveluissa tai GPS-navigointilaitteissa. Muita sovelluskohteita löytyy muun muassa tietoliikenteestä ja robotiikasta. Lyhimmän polun etsintään voidaan käyttää myös osana monien muiden kombinatoristen ongelmien ratkasua.

Olkoon $G = (V, E)$ verkko, jossa on V solmua ja E kaarta, ja olkoon $c(u, v)$ kunkin kaaren $(u, v) \in E$ paino. Jos kaari ei kuulu verkkoon, sen paino on ääretön. Tässä ja jatkossa käytetään merkintöjä V ja E tarkoittamaan sekä solmujen ja kaarten joukkoja, että näiden joukkojen kokoja. Kahden solmun välisen polun pituus on polulle kuuluvien kaarten painojen summa. *Yksittäisen alkusolmun lyhimmän polun ongelmassa* (SSSP) päämääränä on löytää lyhimmät polut ennalta valitusta lähtösolmusta s verkon kaikkiin saavutettavissa oleviin solmuihin. Solmun v etäisyys solmusta s on lyhimmän näiden välisen polun pituus. Käytetään tästä etäisyydestä merkintää $d(v)$. Usein oletetaan lisäksi, että kaarten painot ovat ei-negatiivisia. Näin tehdään myös tässä raportissa käsiteltävissä algoritmeissa.

Lyhimmän polun ongelma on hyvin tunnettu perinteisessä yksitasoisen muistin mallissa (RAM-malli). Nykyaikainen tietokone sisältää kuitenkin useita eri kokoisia ja nopeuksisia muistitasoja. Koska muistitasojen väliset tiedonsiirrot vievät usein suuren osan algoritmin kuluttamasta ajasta, toimivat perinteiset algoritmit usein tehottomasti muistihierarkiassa. Ongelman ratkaisemiseksi on kehitetty useita monitasoisessa muistissa tehokkaasti toimivia algoritmeja [KS96, CR04, BFMZ04]. Tässä raportissa esitellään kaksi tällaista algoritmia ja tutustutaan saatuihin käytännön koetuloksiin.

Luvussa 2 käydään kertauksenomaisesti läpi lyhimmän polun etsintä yksitasoisessa muistissa. Kolmannessa luvussa tutustutaan kaksitasoiseen muistiin ja sen mukanaan tuomiin ongelmiin. Luvuissa 4 ja 5 esitellään kaksi muistihierarkiassa tehokkaasti toimivaa algoritmia lyhimmän polun ongelman ratkaisemiseen. Lopuksi kuudennessa luvussa luodaan katsaus testeissä saavutettuihin käytännön tuloksiin.

2 Lyhin polku yksitasoisessa muistissa

Perinteisessä yksitasoisessa muistimallissa lyhimmän polun etsintä määrätystä lähtösolmusta $s \in V$ suunnatun verkon muihin solmuihin onnistuu tunnetusti ajassa $O(E + V \log V)$ Dijkstran algoritmilla [CLRS01]. Tässä luvussa kerrataan lyhyesti Dijkstran algoritmin toimintaperiaate.

Dijkstran algoritmissa selvitetään lyhin polku verkon kaikkiin solmuihin V solmu kerrallaan. Käsittelemättömistä solmuista pidetään kirjaa minimiprioriteettijonossa Q. Solmun u prioriteetti on lyhimmän löydetyn siihen johtavan polun pituus δ_u. Algoritmin edetessä etäisyyttä δ_u päivitetään aina kun solmuu u löytyy lyhyempi reitti. Kullekin solmu-etäisyys-parille $(u, \delta_u) \in Q$ pätee siis $\delta_u \geq d(u)$, missä $d(u)$ on solmun u etäisyys solmusta s. Aluksi kaikki verkon solmut lisätään prioriteettijonoon Q ja niiden etäisyydet asetetaan äärettömiksi. Tämän jälkeen lähtösolmun etäisyydeksi δ_s asetetaan 0.

Kullakin kierroksella algoritmi poistaa joukosta Q alkion (u, δ_u), jolle δ_u on pienin. Voidaan osoittaa, että tälle pätee $d(u) = \delta_u$. Lyhimmän polun pituus solmuun u on siis saatu selville. Tämän jälkeen algoritmi päivittää lyhintä löydettyä etäisyyttä solmun u kuhunkin naapuriin $v \in Q$: Jos $d(u) + c(u, v) < \delta_u$ (eli jos reitti solmun u kautta solmuun v on lyhyempi kuin aiemmin lyhin löydetty) päivitetään etäisyys $\delta_u := d(u) + c(u, v)$. Kun algoritmia on suoritettu V kierrosta, on kaikki solmut poistettu prioriteettijonosta Q ja lyhimmät etäisyydet kaikkiin solmuihin on siis saatu selville.

Aikaa Dijkstran algoritmissa kuluu erityisesti prioriteettijonon käsittelyyn. Prioriteettijono täytyy tukea operaatioita Insert (alkion lisääminen), Delete-Min (pienimmän alkion poiminta) ja Decrease-Key (arvoon liittyvän prioriteetin/avaimen päivittäminen). Monet kekorakenteet sopivat tähän. Jos prioriteetijono toteutetaan binäärikekona, on algoritmin aikavaativuus $O((E+V) \log V)$. Fibonacci-keolla päästään aiemmin mainittuun aikavaativuuteen $O(E + V \log V)$ [CLRS01].

3 Lyhin polku muistihierarkiassa

Muistihiearakioiden mallintamiseen ja niissä suoritettavien algoritmien analysoimiseen käytetyistä malleista ehkä yleisin on Aggarwalin ja Vitterin [AV88] esittelemä kaksikerroksisen muistin malli (I/O-malli, EM-malli). Mallissa muistihierarkia koostuu kahdesta kerroksesta: nopeasta ja pienestä muistista sekä hitaasta ja suuresta

muistista. Nopean muistin koko on M. Hidas muisti voidaan ajatella tarpeeksi suureksi algoritmin tarpeisiin samaan tapaan kuin perinteisessä yksitasoisessa mallissa.

Jotta muistin sisältämää tietoa voitaisiin käsitellä, täytyy sen olla nopeassa muistissa. Muistitasojen välillä tietoa siirretään lohkoina, joiden koko on B. Alkuperäisessä Aggarwalin ja Vitterin mallissa algoritmi huolehtii itse muistisiirroista. Toisaalta voidaan myös Frigon ja kumppaneiden [FLPR99] tavoin olettaa, että laitteisto suorittaa siirrot automaattisesti ja optimaalisesti. Jälkimmäisessä tapauksessa on mahdollista kehittää muistitasoriippumattomia (cache-oblivious) algoritmeja, jotka toimivat hyvin ilman tietoa parametreista M ja B.

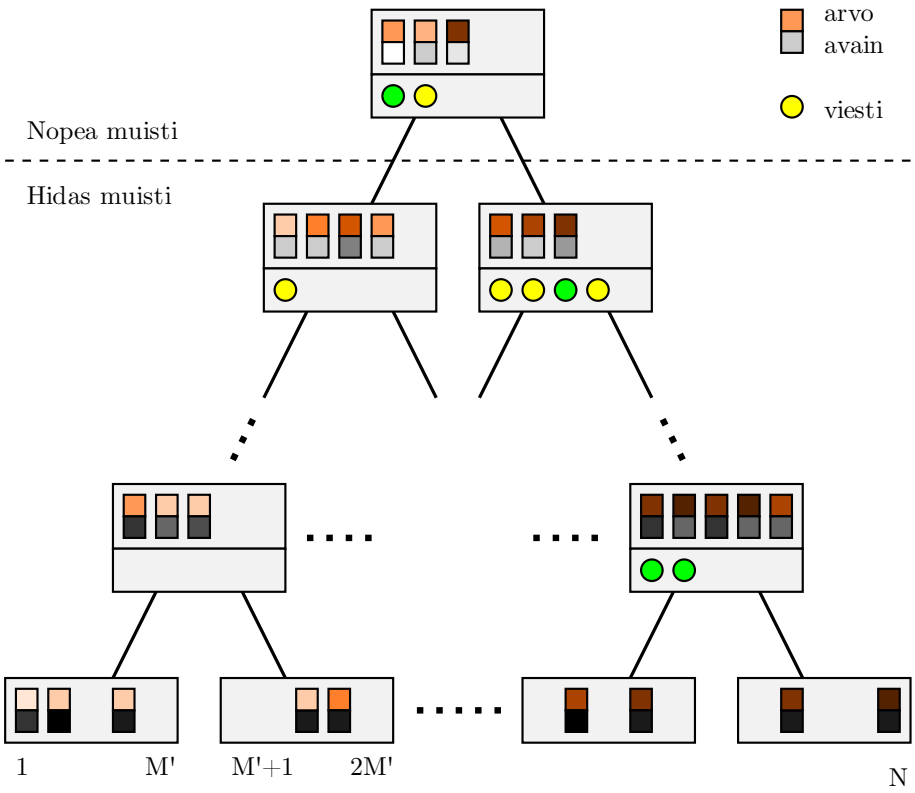
Lyhimmän polun etsintään liittyy muistihierarkiassa muutama ongelma [KM03]: Kun solmun naapurit halutaan käydä läpi, joudutaan vieruslistan hakemiseksi yleensä suorittamaan vähintään yksi lohkosiirto. Jos verkko on harva ja solmujen vieruslistat ovat tarpeeksi lyhyitä tulee tässä usein tehtyä paljon turhaa työtä. Toinen pääasiallinen ongelma on DECREASE-KEY-operaation puute useimmista muistihierarkiassa tehokkaasti toimivista prioriteettijonototeutuksista.

Ensimmäiseen ongelmaan ei ilmeisesti ole löydetty yleispätevää ratkaisua [KM03]. Yhden vieruslistan läpikäyminen vaatii $\Theta(1 + k/B)$ lohkonsiirtoa, missä k on vieruslistan koko. Näin ollen kaikkien vieruslistojen läpikäynti kertaalleen kuluttaa $\Theta(V + E/B)$ siirtoa. Esitettävät algoritmit keskittyvätkin toisen ongelman ratkaisemiseen, eli sopivan prioriteettijonon toteuttamiseen.

4 Turnauspuut

Turnauspuu (tournament tree) on täydellinen binääripuu, jonka oikeanpuolimmaisista lehdistä osa saattaa puuttua. Turnauspuun, jonka korkeus on k voidaan ajatella esittävän $N \leq 2^k$ pelaajan välisten pudotuspelien tuloksia: Kukin lehti vastaa yhtä pelaajaa ja loput solmut otteluiden voittajia. Turnauspuun koko N on kiinteä ja se täytyy olla tiedossa kun turnauspuu luodaan. Sopivilla operaatioilla varustettua turnauspuuta voidaan käyttää prioriteettijonona.

Tässä osiossa esitellään Kumarin ja Schwaben [KS96] kehittämä turnauspuun muunnelma (I/O-efficient tournament tree), joka toimii tehokkaasti kaksitasoisessa muistihierarkiassa. Käytetään siitä tästä lähin merkintää MHTP (muistihierarkiatehokas turnauspuu). Tämän jälkeen näytetään, miten MHTP:ta voidaan käyttää tehokkaaseen lyhimmän polun etsintään suuntaamattomassa verkossa, jossa kaarten painot



Kuva 1: Muistihierarkiatehokkaan turnauspuun MHTP rakenne. Arvojen ja avaimien suuruuksia on merkitty eri väreillä; mitä vaaleampi väri, sitä pienempi arvo/avain. (Perustuu Katrielin ja Meyerin [KM03] kuvaan 4.3.)

ovat ei-negatiivisia. Esittely perustuu pääosin Katrielin ja Meyerin [KM03] näkemykseen puusta ja algoritmista.

4.1 Puun rakenne

MHTP:ssä kuhunkin solmuun säilötään normaalista turnauspuusta poiketen useita alkioita. Kukin alkio on pari (x, k), joka koostuu yksilöivästä arvosta $x \in \{1, \ldots, N\}$ ja avaimesta k. Tässä N on siis puuhun tallennettavien alkioiden määrä. Avain esittää alkion prioriteettiä: mitä pienempi avain, sitä korkeammalle alkio nousee puussa. Jos avain k on ääretön, ei vastaava arvo x kuulu turnauspuun esittämään arvojoukkoon. Itse tietorakenne siis sitältää aina täsmälleen N alkiota, yhden kutankin arvoa $x \in \{1, \ldots, N\}$ kohti. Rakenteen esittämä looginen prioriteettijono sen sijaan sisältää vain ne arvot, joita vastaava avain on äärellinen. Kukin alkio sijaitsee vain yhdessä puun solmussa.

Olkoon $M' = r \cdot M$ jollain ennalta määrätyllä positiivisella vakiolla $r < 1$ ja N säilöttävien alkioiden määrä. Tällöin MHTP sisältää $\lceil N/M' \rceil$ lehtisolmua ja puun korkeus on siis $O(\log_2(N/M))$. Alkiot jaetaan tasaisesti lehtien kesken: alkiot väliltä $(i-1) \cdot M' + 1 \ldots i \cdot M'$ liittyvät i. lehteen. Sanotaan, että lehti kattaa siihen liittyvät alkiot. Vastaavasti kukin sisäsolmu kattaa kaikki sen lapsien kattamat alkiot.

Jokainen sisäsolmu sisältää vähintään $M'/2$ ja korkeintaan M' alkiota. Solmun sisältämät x alkiota ovat sen virittämän alipuun sisältämistä alkioista x pienintä. Lisäksi jokainen sisäsolmu sisältää M'-alkioisen *viestipuskurin*. Vakio r ja siitä riippuva M' on valittu siten, että juurisolmu saadaan pidettyä nopeassa muistissa. Kuvassa 1 on havainnollistettu MHTP:n rakennetta.

4.2 Puun operaatiot

MHTP tukee seuraavia operaatioita:

- DELETE-MIN palauttaa pienimmän alkion ja poistaa sen puusta.
- DELETE(x) poistaa alkion (x, k) puusta.
- UPDATE(x, k') korvaa alkion (x, k) alkiolla (x, k'), jos $k' < k$. Jos puussa ei ole arvoa x, lisätään alkio (x, k').

Operaatioiden kutsuminen lähettää *viestejä*, jotka kulkevat puussa alapäin. Viestit voivat solmuun saapuessaan lisätä, poistaa tai muokata jotain solmun sisältämää alkiota. Jos viesti halutaan lähettää edelleen eteenpäin, se varastoidaan solmun viestipuskuriin. Vasta kun viestipuskuri täyttyy, sen sisältämät viestit lähtetetään alaspäin puussa.

Operaatio DELETE-MIN poistaa pienimmän avaimen sisältävän alkion (x, k) juurisolmusta. Jos juurisolmussa ei ole tarpeeksi alkioita, täytetään se rekursiivisesti sen lapsista. Tämän jälkeen lähetetään alkion (x, ∞) lisäysviesti kohti arvoon x liittyvää lehtisolmua. Alkio lisätään ensimmäiseen viestin kohtaamaan solmuun, jossa on tilaa ja jonka jälkeläiset ovat joko tyhjiä tai sisältävät äärettömän avaimen omaavia alkioita.

Operaatiossa DELETE(x) lähetetään poistoviesti kohti arvoon x liittyvää solmua. Kun alkion (x, k) sisältävä solmu kohdataan, poistetaan alkio, muunnetaan viesti alkion (x, ∞) lisäysviestiksi ja toimitaan kuten edellä.

Operaatio UPDATE(x, k') lähettää päivitysviestin, joka etenee kunnes se kohtaa alkion (x, k) tai solmun, joka sisältää alkioita suuremmalla avaimella kuin k'. Ensimmäisessä tapauksessa löydetty alkio (x, k) korvataan solmun sisällä alkiolla (x, k'), jos $k' < k$. Jälkimmäisessä tapauksessa lisätään kohdattuun solmuun alkio (x, k) ja viesti muunnetaan alkion (x, k') poistoviestiksi, joka lähetetään eteenpäin. Lisäksi jos alkiota lisättäessä solmun kapasiteetti ylittyy, siirretään osa solmun alkioista rekursiivisesti puussa alaspäin.

Useiden alkioden ja viestien puskurointi samaan solmuun aiheuttaa sen, että päivitykset etenevät puussa laiskasti. Tämä edistää algoritmin paikallisuutta ja siten saa sen toimimaan hyvin kaksitasoisessa muistissa. Voidaan osoittaa, että mikä tahansa edellä edellä mainituista operaatioista koostuva z operaation jono voidaan suorittaa $O(z/B \cdot \log_2(N/B))$ lohkonsiirrolla. Operaatioiden tasoitettu vaativuus on siis $O(1/B \cdot \log_2(N/B))$.

4.3 Lyhimmän reitin etsintä turnauspuun avulla

Muistihierarkiassa tehokkaasti toimivan algoritmin muodostamiseksi Dijkstran algoritmin prioriteettijonon Q tietorakenne korvataan esitellyllä MHTP:llä. Aluksi puu alustetaan sisältämään alkiot $(1, \infty), (2, \infty), \ldots, (V, \infty)$ siten, että kukin sisäsolmu sisältää vähintään $M'/2$ alkiota. Tämä jälkeen lähtösolmua s vastaavan alkion avain päivitetetään nollaksi UPDATE$(s, 0)$-operaatiolla.

Algoritmissa suoritetaan V kierrosta. Kullakin kierroksella poimitaan jäljellä olevista solmuista aloitussolmua lähinnä oleva solmu u operaatiolla DELETE-MIN. Toisin kuin Dijkstran algoritmissa, jokaisen sen naapurin v etäisyys päivitetään operaatiolla UPDATE$(v, d(u) + c(u, v))$ operaation DECREASE-KEY sijaan.

Algoritmissa on kuitenkin ongelma: Jos $d(u) < d(v)$, niin solmu u poistetaan ennen v:tä ja sen avaimeksi asetetaan tällöin ∞. Jos solmujen u ja v välillä on kaari, päivitetään kuitenkin solmun v poiston yhteydessä solmun u avaimeksi $d(v)+c(v, u)$, jolloin se palautuu joukkoon Q.

Tämä ongelma kieretään pitämällä yllä toista prioriteettijonoa Q', joka tukee alkioiden lisäämistä ja pienimmän alkion poimimista tasoitetusti $O(1/B \cdot \log_2(N/B))$ lohkonsiirrolla. Tähän tai sen alle päästään useilla toteutuksilla, esimerkiksi Kumarin ja Schwaben esittelemällä kekototeutuksella [KS96].

Algoritmin alussa Q' alustetaan tyhjäksi. Algoritmin edetessä jokaisen UPDATE$(v, d(u)+c(u, v))$-operaation yhteydessä lisätään alkio $(u, d(u) + c(u, v))$ myös joukkoon Q'.

Jokaisen iteraation alussa verrataan joukon Q' pienimmän alkion (u', k') avainta k' joukon Q pienimmän alkion (u, k) avaimeen k. Jos $k \leq k'$, poistetaan (u, k) joukosta Q ja toimitaan aiemmin kuvatulla tavalla. Jos taas $k > k'$, poistetaan (u', k') joukosta Q'. Tällöin täytyy olla $d(u') < d(v) \leq d(u) + c(u, v) = k'$, joten solmu u' on jo käsitelty aiemmin. Voidaan siis suorittaa DELETE(u'), jolloin arvon u' mahdollinen uudelleenlisäys $(u', d(v) + c(v, u))$ joukkoon Q saadaan kumottua. Koska $d(u) + c(u, v) < d(v) + c(v, u)$, ehditään kumoaminen aina suorittaa, ennen kuin u' poimittaisiin joukosta Q uudelleen.

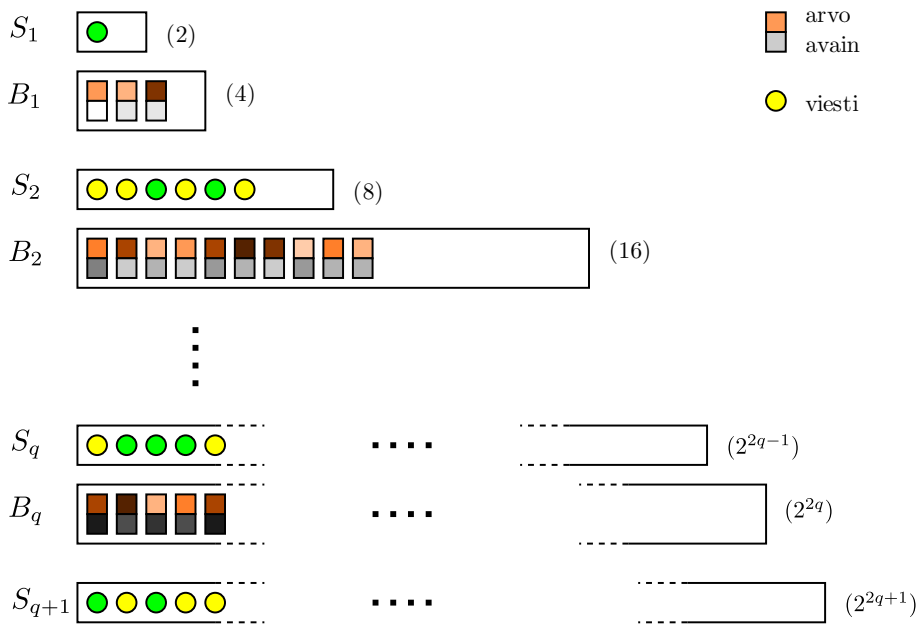
Kuten aiemmin todettiin, vaatii kaikkien verkon solmujen vieruslistojen läpikäynti $\Theta(V + E/B)$ lohkosiirtoa. Edellä mainittujen rakenteiden Q ja Q' käsittelyyn kuluu $O(E/B \cdot \log_2(E/B))$ lokonsiirtoa, joten algoritmin kokonaisvaativuus on luokkaa $O(V + E/B \cdot \log_2(E/B))$. Jotta tähän päästäisiin, täytyy parametri M' osata valita oikein. Tätä varten nopean muistin määrä M pitää olla tiedossa.

5 Ämpärikeko

Edellä esitelty algoritmi vaatii, että nopean muistin koko on tiedossa. Tämä on ongelmallista, sillä muistihierarkian parametreja B ja M ei välttämättä tiedetä. Rajoitus johtuu turnauspuusta. Tässä osiossa tutustutaan Brodalin ja kumppaneiden [BFMZ04] kehittämään *ämpärikekoon* (bucket heap), joka on MHTP:tä vastaava muistitasoriippumaton prioriteettijonototeutus. Myös Chowdhury ja Ramachandran [CR04] ovat kehittäneen vastaavan rakenteen, *puskurikeon* (buffer heap), joka perustuu samaan ideaan. Ämpärikekoa (tai vastaavasti puskurikekoa) käyttämällä edellä esitetty lyhimmän polun etsintäalgoritmi on mahdollista muuttaa muistitasoriippumattomaksi.

5.1 Ämpärikeon rakenne

Kuten turnauspuuhun, myös ämpärikekoon varastoidaan arvo-avain-pareja. Tallennettavien arvojen suuruutta ei ole kuitenkaan rajoitettu samalla tavalla; riittää että ne ovat yksilöiviä. Olkoon N kullakin hetkellä tallennettujen alkioiden määrä. Ämpärikeko koostuu *ämpäreistä* $B_1, B_2, \ldots, B_q$ ja *viestipuskureista* $S_1, S_2, \ldots, S_{q+1}$. Parametri q vaihtelee, mutta on aina korkeintaan $\lceil \log_4 N \rceil$. Kuhunkin ämpäriin B_i mahtuu 2^{2i} alkiota ja vastaavasti kuhunkin viestipuskuriin S_i mahtuu 2^{2i-1} viestiä. Koko siis aina nelinkertaistuu kun i kasvaa yhdellä. Prioriteettijonon kullakin hetkel-



Kuva 2: Ämpärikeon rakenne. Arvojen ja avaimien suuruuksia on merkitty eri väreillä; mitä vaaleampi väri, sitä pienempi avain.

lä sisältämät alkiot säilötään ämpäreihin siten, että jos B_i ja B_j ovat kaksi ämpäriä ja $i < j$ niin mille tahansa alkioille $(x, k_x) \in B_i$ ja $(y, k_y) \in B_j$ pätee $k_x \leq k_y$.

Toisin kuin turnauspuun tapauksessa, ämpärikeko sisältää kullakin hetkellä fyysisesti samat alkiot kuin sen esittämä looginen prioriteettijono. Ämpäreiden määrä vaihtelee dynaamisesti tallennettujen alkioiden määrän mukaan. Keon maksimikokoa ei siis tarvitse päättää ennalta kuten turnauspuulla. Tällä ei tosin ole merkitystä lyhimmän polun etsimisessä. Kuvassa 2 on havainnollistettu ämpärikeon rakennetta.

Kullekin ämpärille ja viestipuskurille varataan sen kapasiteettiin nähden kaksinkertainen määrä muistia väliaikaisten ylivuotojen sallimiseksi. Ämpärit ja puskurit tallennetaan muistiin peräkkäin järjestyksessä $S_1, B_1, S_2, B_2, \ldots, S_q, B_q, S_{q+1}$.

5.2 Ämpärikeon operaatiot

Ämpärikeko tukee samoja operaatioita kuin aiemmin esitelty turnauspuukin: DELETE-MIN poistaa ja palauttaa pienimmän alkion, DELETE(x) poistaa alkion x ja UPDATE(x, k) pienentää alkion x avainta tarvittaessa tai lisää x:n kekoon jos se ei ole siellä ennestään. Kuten turnauspuussa, operaatiot toteutetaan rakenteessa alapäin etenevien viestien avulla. Lisäksi käytetään kahta apuoperaatiota:

- EMPTY(S_i) tyhjentää viestipuskurin S_i toteuttamalla viestien vaatimat toimenpiteet vastaavan ämpärin B_i alkioihin ja lähettämällä jäljelle jäävät viestit eteenpäin puskuriin S_{i+1}.
- FILL(B_i) varmistaa että ämpäri B_i on täynnä ja täyttää sen tarvittaessa.

Näiden avulla muut operaatiot on helppo toteuttaa. Operaatio DELETE-MIN varmistaa, että ensimmäinen ämpäri on täynnä kutsumalla FILL(B_1). Tämän jälkeen se poistaa ämpärin pienimmän alkion ja palauttaa sen. Operaatiot DELETE(x) ja UPDATE(x, k) lisäävät vastaavan *poisto-* tai *päivitysviestin* viestipuskuriin S_1 ja kutsuvat tämän jälkeen operaatiota EMPTY(S_1).

Päivitysviesti ja poistoviesti toimivat pääosin kuten turnauspuussakin: Päivitysviesti alkiolle (x, k) etenee, kunnes se kohtaa päivitettävän alkion (x, k') tai päätyy puskuriin, jota vastaava ämpäri sisältää suurempia alkoita. Ensimmäisessä tapauksessa alkion avain päivitetään tarvittaessa, ja toisessa tapauksessa alkio (x, k) lisätään ämpäriin ja viesti muutetaan alkion x poistoviestiksi. Poistoviesti etenee, kunnes se kohtaa poistettavan alkion. Tämän jälkeen kohdattu alkio poistetaan ämpäristä. Lisäksi käytetään vielä kolmatta viestityyppiä. Alkion (x, k) *työntöviestiä* käytetään puskemaan alkioita yhtä pykälää alemmalle tasolle. Alemmalla tasolla alkio lisätään tason ämpäriin ja työntöviesti merkitään käsitellyksi.

Pääasiallisen työn tekevät kaksi apuoperaatiota. Operaatio EMPTY(S_i) käy viestipuskurin S_i viestit läpi ja suorittaa niiden vaatimat toimenpiteet. Kaikki jäljelle jääneet viestit siirretään seuraavalle tasolle puskuriin S_{i+1}, jolloin S_i tyhjenee. Jos ämpärin B_i alkioiden määrä ylitti sen kapasiteetin, lähetetään ylijäämän verran suurimpia alkioita alemmalle tasolle lisäämällä vastaavat työntöviestit seuraavan tason viestipuskuriin S_{i+1}. Jos lisäksi S_{i+1} täyttyi yli kapasiteettinsa, kutsutaan sille rekursiivisesti EMPTY(S_{i+1}). Tarvittaessa tasojen määrää q kasvatetaan yhdellä.

Operaatio FILL(B_i) tyhjentää aluksi vastaavan viestipuskurin kutsumalla EMPTY(S_i). Jos tämän jälkeen pätee $|B_{i+1}| < 2^{2i}$ eli jos B_{i+1} sisältää alkioita alle neljäsosan kapasiteetistaan, kutsutaan rekursiivisesti operaatiota FILL(B_{i+1}). Lopuksi täytetään tyhjä tila ämpärissä B_i seuraavan tason B_{i+1} pienimmillä alkioilla. Lisäksi tasojen määrää q pienennetään, mikäli ylimmät tasot tyhjenivät kokonaan.

Kuten turnauspuun tapauksessa, operaatiot suoritetaan ämpärikeossa laiskasti, jolloin se toimii hyvin muistihierarkiassa. Kaikkien operaatioiden DELETE-MIN, DELETE ja UPDATE tasoitettu lohkonsiirtojen määrä on $O(1/B \cdot \log_2(N/B))$. Tämä on sama kuin esitellyllä turnauspuulla. Ämpärikekoa varten ei kuitenkaan tarvitse tietää

muistihierarkian parametreja M tai B, jolloin sama toteutus toimii optimaalisesti parametreista riippumatta.

5.3 Lyhimmän reitin etsintä ämpärikeon avulla

Lyhimmän polun etsintä on helppo toteuttaa samaan tapaan kuin turnauspuun avulla. Pääasiallisena prioriteettijonona Q käytetään turnauspuun sijaan ämpärikekoa. Lisäksi apuprioriteettijonona Q' täytyy käyttää jotain muistitasoriippumatonta kekototeutusta. Tähän kelpaa esimerkiksi ämpärikeko itsessään tai vaikkapa Agren ja kumppaneiden [ABD+07] kehittämä toteutus.

Tuloksena saadaan aiemmin esitellyn algoritmin kanssa samalla lohkonsiirtojen määrällä $O(V + E/B \cdot \log_2(E/B))$ toimiva algoritmi. Erona aikaisempaan uusi algoritmi toimii optimaalisesti vaikka muistihierarkian parametreja ei olisikaan tiedossa.

6 Käytännön koetuloksia

Lyhimmän reitin etsinnästä muistihierarkiassa on olemassa joitakin koetuloksia. Sach ja Clifford [SC08] ovat verranneet perinteisen binäärikekoa käyttävän Dijkstran algoritmin, ämpärikekoa käyttävän algoritmin sekä *suppilokekoa* käyttävän algoritmin tehokkuutta käytännössä. Brodalin ja Fagerbergin [BF02] esittelemä suppilokeko ei tue DECREASE-KEY tai UPDATE-operaatioita, joten avaimen päivityksen sijaan täytyy kekoon aina lisätä uusi alkio INSERT-operaatiolla. Keosta jo poistetuista arvoista pidettiin testissä kirjaa bittivektorissa, jotta niiden käsittely moneen kertaan osattaisiin välttää: Kun arvo poimitaan ensimmäisen kerran keosta, se merkitään käytetyksi vaihtamalla bittivektorissa vastaavan bitin arvoa.

Testejä suoritettiin sekä keinotekoisilla että todellisilla verkoilla. Nopean muistin koko oli rajoitettu 32:een megatavuun. Pienillä verkoilla binäärikekoa käyttävä algoritmi oli nopein testatuista. Kun verkon kokoa kasvatettiin miljooniin solmuihin, ohittivat ämpärikekoa ja suppilokekoa käyttävät algoritmit selvästi perinteisen Dikstran algoritmin. Näistä kahdesta suppilokekoa käyttävä algoritmi toimi erityisesti harvoissa verkoissa hieman tehokkaammin kuin ämpärikekoa käyttävä.

Myös Chen ja kumppanit [CAR+07] ovat vertailleet erilaisia lyhimmän polun hakualgoritmeja muistihierarkiassa. Heidänkin testeissään muistihierarkiassa tehokkaasti toimimaan suunnitellut algoritmit suoriutuivat yleisesti paremmin kuin perinteiset algoritmit.

7 Yhteenveto

Lyhimmän polun etsintä on yleinen verkkoihin liittyvä ongelma. Tässä raportissa esiteltiin kaksi erilaista muistihierarkiassa tehokkaasti toimivaa Dijkstran algoritmin muunnelmaa. Ensimmäinen perustuu muistihierarkiatehokkaaseen turnauspuuhun ja vaatii, että nopean muistin koko on tiedossa. Toinen käyttää ämpärikekoa ja on muistitasoriippumaton, eikä siis tarvitse parametreja toimiakseen optimaalisesti. Kummatkin esitellyt algoritmit suorittavat $O(V + E/B \cdot \log_2(E/B))$ lohkonsiirtoa. Testeissä on lisäksi osoitettu, että teoreettisesti parempien aikavaatimusten lisäksi muistihierarkiaan optioiduilla algoritmeilla voidaan saavuttaa merkittävää nopeusetua myös käytännössä.

Käsitellyt algoritmit toimivat kaikissa suuntaamattomissa verkoissa, joissa kaarten painot eivät ole negatiivisia. Erilaisiin erikoistapauksiin on myös olemassa useita menetelmiä. Esimerkiksi tapauksiin, joissa kaarien painot ovat rajoitettuja, on kehitetty algoritmeja [MZ03, ALZ07]. Myös tasoverkkoihin erikoistuneita algoritmeja ja tietorakenteita on esitelty [JZ05, AT05].

Lähteet

ABD+07 Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B. ja Ian Munro, J., An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal of Computing*, 36,6(2007), sivut 1672–1695.

ALZ07 Allulli, L., Lichodzijewski, P. ja Zeh, N., A faster cache-oblivious shortest-path algorithm for undirected graphs with bounded edge lengths. *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, sivut 910–919.

AT05 Arge, L. ja Toma, L., External data structures for shortest path queries on planar digraphs. *Algorithms and Computation: 16th International Symposium*. Springer, 2005, sivut 328–338.

AV88 Aggarwal, A. ja Vitter, J. S., The input/output complexity of sorting and related problems. *Communications of the ACM*, 31,9(1988), sivut 1116–1127.

BF02 Brodal, G. S. ja Fagerberg, R., Funnel heap - a cache oblivious priority queue. *ISAAC '02: Proceedings of the 13th International Symposium on Algorithms and Computation*. Springer-Verlag, 2002, sivut 219–228.

BFMZ04 Brodal, G. S., Fagerberg, R., Meyer, U. ja Zeh, N., Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*. Springer, 2004, sivut 480–492.

CAR+07 Chen, M., Alam, R., Ramach, C. V., Lan, D. ja Tong, R. L., Priority queues and dijkstra's algorithm. Tekninen raportti, The University of Texas at Austin, 2007.

CLRS01 Cormen, T. H., Leiserson, C. E., Rivest, R. L. ja Stein, C. *Introduction to Algorithms*, luku 24.3: Dijkstra's algorithmy, sivut 595–601. MIT Press and McGraw-Hill, toinen painos, 2001.

CR04 Chowdhury, R. A. ja Ramachandran, V., Cache-oblivious shortest paths in graphs using buffer heap. *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2004, sivut 245–254.

FLPR99 Frigo, M., Leiserson, C. E., Prokop, H. ja Ramachandran, S., Cache-oblivious algorithms. *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1999, sivut 285–297.

JZ05 Jampala, H. ja Zeh, N., Cache-oblivious planar shortest paths. *Automata, Languages and Programming: 32nd International Colloquium*. Springer, 2005, sivut 563–575.

KM03 Katriel, I. ja Meyer, U. *Algorithms for Memory Hierarchies*, luku 4. Elementary Graph Algorithms in External Memory, sivut 62–84. Springer, 2003. 70–74.

KS96 Kumar, V. ja Schwabe, E. J., Improved algorithms and data structures for solving graph problems in external memory. *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 1996, sivut 169–176.

MZ03 Meyer, U. ja Zeh, N., I/o-efficient undirected shortest paths. *Algorithms - ESA 2003*. Springer, 2003, sivut 434–445.

Pag03 Pagh, R. *Algorithms for Memory Hierarchies*, luku 2. Basic External Memory Data Structures, sivut 14–35. Springer, 2003. 26–27.

SC08 Sach, B. ja Clifford, R., An empirical study of cache-oblivious priority queues and their application to the shortest path problem, `http://arxiv.org/pdf/0802.1026`, 2008.