Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

Petteri Turtiainen

Algoritmi interaktiiviseen järjestämiseen

Diplomityö
Espoo, 26. toukokuuta 2016

Valvoja: Eljas Soisalon-Soininen, Aalto University

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Petteri Turtiainen

Title:
Algorithm for interactive sorting

Date:	May 26, 2016	**Pages:**	57
Major:	Software Technology	**Code:**	T-106
Supervisor:	Professor Eljas Soisalon-Soininen		

Both humans and computers struggle with sorting datasets where the ordering is defined by a user's subjective preference or experience. Datasets like this include for example "top 10" lists of books or movies. Since a user's preference does not have a numerical value, the computer can not sort the dataset by itself, but it needs input from the user.

In this thesis, we solve this problem by introducing a new algorithm, which iteratively requests the user to input a relative order for a couple of items in the unsorted set. Using these partial orderings iteratively, the algorithm updates a directed graph of "greater than" relations and uses a topological sort algorithm to sort the items into a result list. The algorithm is designed to be self-repairing and tolerant of mistakes when the user makes errors in input.

Our algorithm proves to be efficient with small datasets and despite user error provides a mostly correct result. Based on this work it is possible to further develop the algorithm for more specific use cases in marketing and research or as a part of an external application which can benefit from free-form sorting.

Keywords: sorting, graph, algorithm, transitive closure, interactive, user, topological sort

Language: Finnish

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Petteri Turtiainen		
Työn nimi:	Algoritmi interaktiiviseen järjestämiseen		
Päiväys:	26. toukokuuta 2016	**Sivumäärä:**	57
Pääaine:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Eljas Soisalon-Soininen		

Sekä ihmisille että koneille on vaikeaa järjestää aineistoa, jossa järjestys määräytyy käyttäjän subjektiivisen kokemuksen kautta. Tällaisia aineistoja ovat esimerkiksi listaukset, joihin on valittu kymmenen parasta elokuvaa tai kirjaa. Koska käyttäjän kokemuksella kirjasta ei ole numeerista arvoa, tietokone ei pysty itse järjestämään aineistoa, vaan tarvitsee tähän käyttäjän syötettä.

Tässä työssä kehitimme yleiskäyttöisen interaktiivisen algoritmin, joka toistuvasti kysyy käyttäjältä tietoa järjestettävien tietoalkioiden välisestä suhteellisesta suuruusjärjestyksestä ja täydentää suuruusjärjestystä mallintavaa suunnattua verkkoa. Käyttämällä topologista järjestämistä, algoritmi järjestää tietoalkiot tuloslistaksi. Algoritmi on suunniteltu virhesietoiseksi ja itseään korjaavaksi käyttäjän virheiden varalta. Se ei myöskään ota kantaa järjestettävän aineiston tyyppiin.

Algoritmi osoittautui tehokkaaksi lyhyiden listojen järjestämisessä ja käyttäjän virheistä huolimatta pystyy tuottamaan enimmäkseen oikean lopputuloksen. Tämän työn pohjalta on mahdollista jatkokehittää algoritmia erityisempiä käyttötarkoituksia varten, kuten asiakaspalautteen ja tutkimusmateriaalin keräämiseen tai muiden sovellusten osana.

Asiasanat:	järjestäminen, verkko, algoritmi, transitiivinen sulkeuma, interaktiivisuus, käyttäjä, topologinen järjestäminen
Kieli:	Suomi

Kiitossanat

Haluan kiittää ensisijaisesti vaimoani Taijaa ja tytärtäni Tinjaa loppumattomasta kärsivällisyydestä ja tuesta, jota tämän työn kirjoitustyö on vaatinut heiltä.

Lisäksi kiitän työn valvojaa Eljas Soinsalon-Soinista työn tarkistamisesta, Le Havren yliopiston tietotekniikkalaboratio LITISin väkeä erinomaisen GraphStream kirjaston kehittämisestä sekä ERPG-kanavan väkeä kirjoitustyön vaatimasta kaikupohjasta.

Lopuksi haluan kiittää isääni, Pasi Turtiaista, joka ei ehtinyt nähdä tätä työtä kansitettuna. On hänen ansiotaan, että lähdin opiskelemaan Otaniemeen.

Sanasto

Lomistuslajittelu	Merge sort
Pikalajittelu	Quicksort
Turnaus	Tournament
Laskentalajittelu	Counting sort
Kantalukulajittelu	Radix sort
Nippulajittelu	Bucket sort
Valintaylikuormitus	Choice overload
Minimaalinen takaisinkytkentäjoukko	Minimal feedback arc set
Maksimaalinen itsenäinen joukko	Maximal independent set
Minimaalinen leikkaus	Minimal cut

Sisältö

Käytetyt termit **5**

1 Johdanto **8**

2 Perinteiset järjestämisalgoritmit **11**
2.1 Pikalajittelu 11
2.2 Lajittelu lineaarisessa ajassa 11
2.3 Yhteenveto 13

3 Verkkoalgoritmit **14**
3.1 Suuruusjärjestys verkkona 14
3.2 Turnaus 15
3.3 Dijkstran algoritmi 15
3.4 Verkon läpikäynnit 15
3.4.1 Leveyshaku 16
3.4.2 Syvyyshaku 16
3.5 Topologinen järjestäminen 16
3.6 Transitiivinen sulkeuma 17
3.7 Verkon leikkaukset 18
3.8 Yhteenveto 19

4 Dynaamiset ongelmat ja ratkaisut **21**
4.1 Perinteiset algoritmit sovellettuina interaktiivisiksi 21
4.1.1 Monkeysort 21
4.2 Inkrementaalinen topologinen järjestäminen 22
4.3 Transitiivisen sulkeuman naiivi päivittäminen dynaamisesti . . 24
4.3.1 Transitiivisen sulkeuman päivittäminen kaaren lisäyksen yhteydessä 24
4.3.2 Kaaren poisto ja transitiivisuuden päivittäminen 26
4.3.3 Probabilistiset algoritmit 26

4.3.4 Tehokkaammat transitiivisen sulkeuman deterministisesti laskevat algoritmit . . . 27
4.4 Yhteenveto . . . 27

5 Interaktiivinen järjestäminen 28
5.1 Käyttäjän syöte . . . 28
5.1.1 Käyttäjälle tarjottavan alkiojoukon koko . . . 29
5.2 Virheet syötteessä . . . 30
5.2.1 Käyttäjän syötteen ristiriidat . . . 30
5.2.1.1 Silmukoiden käsittely . . . 31
5.2.1.2 Polun heikentäminen . . . 32
5.2.2 Keskeytynyt suoritus . . . 33
5.3 Yhteenveto . . . 33

6 Algoritmin toteutus 34
6.1 Algoritmin yleiskuvaus . . . 35
6.2 Lopetusehto . . . 35
6.3 Käyttäjälle esitettävän alijoukon valinta . . . 36
6.4 Käyttäjän syöte . . . 39
6.5 Uusien kaarien lisääminen verkkoon . . . 39
6.6 Transitiivisen sulkeuman päivittäminen . . . 41
6.7 Topologinen järjestäminen ja tulosten esittäminen . . . 41

7 Analyysi 42
7.1 Virhealttius . . . 44
7.2 Virheen kehitys algoritmin edetessä . . . 44
7.3 Tiheysrajan muuttaminen . . . 45
7.4 Algoritmin skaalautuminen pidemmille listoille . . . 47
7.5 Tiheysrajan yhdistäminen käyttäjän virheeseen . . . 48
7.6 Yhteenveto . . . 49

8 Yhteenveto 51
8.1 Jatkokehitys . . . 51
8.1.1 Koneellinen järjestäminen . . . 51
8.1.2 Algoritmin jatkokehittäminen . . . 52
8.1.3 Nippulajittelu . . . 53

Luku 1

Johdanto

Perinteiset järjestämisalgoritmit vaativat järjestettäviltä alkioilta eksplisiittisen suuruusjärjestyksen tai ainakin jonkin ominaisuuden, jota voidaan koneellisesti käsitellä. Jos käyttäjä haluaa järjestää esimerkiksi kuvia tai mielipiteitä omien toiveidensa mukaiseen järjestykseen, koneelliset järjestämisalgoritmit eivät osaa käsitellä järjestettävää dataa käyttäjän toivomalla tavalla tai datasta puuttuu tyystin järjestämisen perustana käytettävä ominaisuus. Tilanteet, joissa alkioita pitäisi järjestää esimerkiksi käyttäjän mieltymysten tai arvojen mukaan, vaatisi käyttäjää antamaan vertailukelpoisen numeerisen arvon abstraktille käsitteelle, ja tämän numeerisen arvon perusteella järjestäminen voitaisiin suorittaa koneellisesti.

Tällaisia abstrakteja järjestämistehtäviä ovat esimerkiksi elokuvien arvosteleminen paremmuusjärjestykseen, kuvien järjestäminen niiden miellyttävyyden mukaan tai tehtävälistan järjestäminen käyttäjän tarpeiden mukaan. Näihin järjestämistehtäviin voi luoda tilanteeseen erikoistuneita järjestämistapoja, kuten elokuva-arvosteluissa, joissa elokuvan saamia palkintoja, pääosan esittäjiä, juonen laatua ja esteettisiä tehosteita arvioidaan ja antaa näille osa-alueille arvosanat. Näitä osa-arvosanoja painottamalla elokuville voidaan laskea vertailukelpoinen vertailuarvo ja käyttää sitä järjestämisen perusteena. Silti käyttäjälle voi olla hyvinkin vaikeaa antaa yksi selkeä numeerinen arvo monimutkaiselle kokonaisuudelle kuten elokuvalle.

Toisaalla esimerkiksi tehtävälistan tehtävät voidaan järjestää osajoukkoihin tärkeyden ja kiireellisyyden mukaisesti ja näin antaa tehtäville ainakin karkea suhteellinen tärkeysjärjestys. Ensisijaisesti käyttäjä haluaa suorittaa kiireiset tehtävät ja toissijaisesti tärkeät tehtävät. Näin tehtävät jakautuvat nelikenttään, jossa tärkein neljännes on kiireelliset ja tärkeä tehtävät. Näiden nelikentän kategorioiden avulla voidaan tehdä päätöksiä tehtävälistan tehtävien priorisoinnista, mutta se jättää auki edelleen missä järjestyksessä kategorian sisällä tehtävät ovat keskenään.

Tämänkaltaiset ad hoc lähestymistavat ovat lukittuja ongelman laatuun, eikä niitä voida helposti yleistää tukemaan arbitraarisen tietojoukon järjestämistä.

Tavoitteenamme on kehittää algoritmi, joka yhdistää edellä mainitut lähestymistavat ja antaa arbitraariselle alkiojoukolle järjestyksen riippumatta järjestettävän tiedon luonteesta. Tätä tarkoitusta varten algoritmi interaktiivisesti kyselee käyttäjältä syötettä alkioiden välisestä suhteellisesta suuruusjärjestyksestä, ja käyttäjän antaman syötteen perusteella algoritmi päättelee lopullisen suuruusjärjestyksen alkiojoukolle. Näin käyttäjä saa lopputuloksena alkioiden tarkan suuruusjärjestyksen, kuten elokuvaesimerkissä, ja toisaalta käyttäjän syötteenä antamat yksittäiset valinnat ovat yhtä yksinkertaisia kuin tehtävälistan kohdan määrittäminen tärkeäksi tai ei-tärkeäksi.

Vaikka koneellinen järjestäminen on kehitetty pitkälle, on ihmisten tekemällä lajittelulla edelleen paikkansa. Amazonin Mechanical Turk (MTurk) tarjoaa alustan, jossa ihmisten suorittamaa manuaalista työtä voi ostaa paloiteltuna pieniin HITteihin (Human Intelligence Task) [2]. Käyttäjät saavat HITtejä suorittamalla pienen korvauksen, joka on tyypillisesti suuruudeltaan muutamia senttejä minuuttien työstä. HITtejä voi tilata joustavasti suurissa määrissä Amazonin palvelualustan avulla ja sitä käytetäänkin esimerkiksi kuvantunnistamiseen ja tutkimusmateriaalin keräämiseen [2, 27].

Toinen merkittävä inhimillisen syötteen tarve on asiakaspalautteessa, jossa halutaan käyttäjän palautetta tarjotusta palvelusta tai tuotteesta. Palautteen annossa voi olla esimerkiksi tarve laittaa asiakaspalvelun osa-alueista paremmuusjärjestykseen. Tätä järjestämistä voi helpottaa käyttämällä tarpeeseen soveltuvaa algoritmia, joka selvittää käyttäjän syötteellä halutun järjestyksen.

Hankalaksi interaktiivisen järjestämisen tekee se, että käyttäjä jolta syötettä kysytään, saattaa muuttaa mieltänsä kesken järjestämisen, tehdä näppäilyvirheitä tai loogisia virheitä tai jopa jättää järjestämisalgoritmin suorittamisen kesken. Kuitenkin algoritmin pitäisi pystyä tuottamaan hyväksyttävä tulos myös näissä olosuhteissa.

Marcus et al. [27] ovat tehneet Amazonin Mechanical Turk palvelua varten järjestämisalgoritmin, joka käyttää hyväksi käyttäjien tekemiä parivertailuja listan järjestämiseen. Heidän algoritminsa perustuu suunnattuun verkkoon, johon lisätään käyttäjien syötteen perusteella suunnattuja kaaria osoittamaan solmujen välistä suuruusjärjestystä. Kun verkkoa on täytetty riittävästi, ajetaan verkkoon silmukoidenpurkualgoritmi ja verkko järjestetään topologisella järjestämisellä.

Tässä työssä teemme samankaltaisen algoritmin, mutta parannamme sitä yhden käyttäjän tarpeisiin vähentämällä kyselyiden määrää ja antamalla välitöntä palautetta algoritmin edistymisestä. Näitä käsitellään tarkemmin

luvuissa 5 ja 6. Marcus et al.:n algoritmi [27] myös keskittyi vahvasti siihen, miten järjestämisen varmuutta voidaan parantaa tilastollisesti käyttämällä useampia käyttäjiä ja arvottamalla heidän suoritustaan suhteessa muihin käyttäjiin verrattuna. Meidän algoritmimme ei luonnollisesti voi nojata siihen, että ongelman ratkaisu paranee käyttäjää vaihtamalla.

Tässä työssä pohjustamme ongelmaa luvussa *2 Perinteiset järjestämisalgoritmit* käymällä läpi perinteisiä järjestämisalgoritmeja ja perustelemalla minkä vuoksi ne eivät sovi ongelman ratkaisemiseen aukotta. Tämän jälkeen luvussa *3 Verkkoalgoritmit* mallinnamme suuruusjärjestyksen verkkorakenteena ja pohjustamme algoritmin toteutuksessa käytetyt ja vaihtoehtoisesti harkitut verkkoalgoritmit. Luvussa *4 Dynaamiset ongelmat ja ratkaisut* sovellamme verkkoalgoritmeja dynaamisten ongelmien ratkaisemiseen ja esittelemme verkkoalgoritmit, jotka on suunniteltu muuttuvan verkon käsittelyyn.

Luvussa *5 Interaktiivinen järjestäminen* taustoitamme ympäristön, jossa algoritmi tulee toimimaan, eli käyttäjän epävarmuuden ja ristiriitatilanteet, joihin algoritmin täytyy kyetä vastaamaan. Luvussa *6 Algoritmin toteutus* selostamme algoritmin toteutuksen ja toiminnan, sekä algoritmin teknisessä toteutuksessa käytetyt algoritmit aiemmista luvuista.

Algoritmin tuloksia tarkastellaan luvussa *7 Analyysi*, jota varten olemme testanneet algoritmin toimintaa epävarmalla koneellisella syötteellä ja arvioineet algoritmin tehokkuutta ja virhesietoisuutta. Lopuksi luvussa *8 Yhteenveto* kertaamme mitä työssä on saavutettu ja mitä jatkokehitysmahdollisuuksia algoritmi tarjoaa.

Luku 2

Perinteiset järjestämisalgoritmit

Tyypillisesti järjestämistehtävän työläys on verrannollinen algoritmin vaatimien vertailuoperaatioiden määrään. Tässä luvussa käymme läpi perinteisiä järjestämisalgoritmeja ja pyrimme selvittämään miksi ne eivät sovellu tässä työssä käsiteltävän ongelman ratkaisemiseen.

2.1 Pikalajittelu

Pikalajittelu on hyvin tunnettu ja tehokas järjestämisalgoritmi, jonka tavoitteena on järjestää lista mahdollisimman nopeasti. Se on rekursiivinen hajota ja hallitse -algoritmi, jossa järjestettävä alkiojoukko jaetaan kahteen osaan ja molemmat joukot järjestetään pikalajittelulla.

Algoritmi valitsee järjestettävistä alkioista yhden sarana-alkion, jonka suhteen lista jaetaan kahtia: sarana-alkiota pienemmät alkiot siirretään ensimmäiseen alilistaan ja jakoalkiota suuremmat alkiot toiseen alilistaan [9, s. 170-179]. Kumpikin alilista järjestetään rekursiivisesti pikalajittelulla ja tämän jälkeen järjestyksessä olevat alilistat yhdistetään peräkkäin jakoalkion kanssa. Cormen et al. [9, s. 171] esittää pikalajittelusta myös version, joka käyttää ainoastaan yhtä listaa järjestämiseen ja algoritmin edetessä siirtelee alkioita listan sisällä. Keskimääräiseltä aikavaativuudeltaan pikalajittelu on luokkaa $\theta(n \log n)$ [9, s. 174-178]. Jos alkiot ovat valmiiksi järjestyksessä, algoritmin suorituskyky heikkenee merkittävästi ja pahimman tapauksen aikavaatimus on $O(n^2)$.

2.2 Lajittelu lineaarisessa ajassa

Koska alkioita keskenään vertailevilla algoritmeilla pienin mahdollinen vertailuoperaatioiden määrä noudattaa kaavaa $\Omega(n \log n)$ [9, s. 193], olisi mah-

Algoritmi 2.1: Pikalajittelu

```
funktio pikajärjestys(lista)
  Jos listan pituus <= 1
    palauta lista
  Muuten valitse ja poista sarana-alkio listasta
    palauta pikajärjestys([listan saranaa pienemmät])
      + [sarana-alkio] +
      pikajärjestys([listan saranaa suuremmat])
```

dollista parantaa tehokkuutta käyttämällä tehokkaampia, lineaarisessa ajassa suoriutuvia lajittelualgoritmeja: laskentalajittelua tai kantalukulajittelua.

Laskentalajittelu [9, s. 195-197] ja kantalukulajittelu [9, s. 197-200] on suunniteltu käyttämään hyväksi järjestettävien lukujen ja luvuiksi muutettavissa olevien tekstien ominaisuuksia, kuten pituutta, merkitsevintä numeroa tai kantalukuja, joiden perusteella luvut voidaan asettaa suuruusjärjestykseen lineaarisessa ajassa. Koska ongelmamme alkioilla ei voida olettaa olevan tällaisia ominaisuuksia, nämä lajittelualgoritmit eivät sovi käyttöömme.

Nippulajittelu on kolmas Cormenin et al. [9, s. 200-212] esittelemä listan lineaarisessa ajassa järjestävä algoritmi. Se on abstraktimpi lajittelualgoritmi kuin laskentalajittelu tai kantalukulajittelu, joten se voisi sopia paremmin abstraktille syötejoukolle, jota yritämme järjestää.

Nippulajittelu toimii käymällä läpi järjestettävät alkiot ja jakamalla ne etukäteen määriteltyihin nippuihin, joilla on määriteltävissä suuruusjärjestys. Lajittelun jälkeen jokainen nippu järjestetään yksinkertaisella lisäysjärjestämisellä tai muulla nopealla lajittelualgoritmilla. Nippulajittelu olettaa, että järjestettävissä nipuissa ei järjestämisvaiheessa ole merkittävää määrää alkioita, joten nippujen sisäinen järjestäminen on nopea operaatio. Tämän jälkeen keskenään järjestyksessä olevat niput ja niiden järjestetty sisältö kootaan yhteen tuloslistaan.

Nippulajittelu voisi toimia sellaisilla syötejoukoilla, joille voidaan määritellä etukäteen järjestetyt niput. Pahimmassa tapauksessa kaikki alkiot lajitellaan samaan nippuun. Silloin tämän yhden nipun järjestämiseen kuluu $O(n^2)$ vertailua, jos käytetään lisäysjärjestämistä.

Tavoitteenamme on kehittää yleiskäyttöinen järjestämisalgoritmi, joten nippulajittelu soveltuu huonosti algoritmin perustaksi. Emme voi lähteä oletuksesta, että järjestettävillä alkioilla olisi jokin ominaisuus, jonka perusteella käyttäjä voisi jakaa ne ennalta määriteltyihin nippuihin. On kuitenkin mahdollista, että sitä voisi soveltaa lajittelemaan suuret alkiojoukot helpom-

min käsiteltäviin osajoukkoihin, joiden järjestäminen algoritmilla voisi olla nopeampaa kuin koko alkiojoukon käsittely kokonaisuudessaan.

2.3 Yhteenveto

Tässä luvussa listatut järjestämisalgoritmit ovat tehokkaita ja pitkälle kehitettyjä. Ne kuitenkin soveltuvat sellaisenaan huonosti interaktiiviseen järjestämiseen ja epävarmaan alkioiden väliseen suuruusjärjestykseen. Nippulajittelua olisi mahdollista soveltaa algoritmin yhteydessä ja mahdollisessa jatkokehityksessä; sen avulla voitaisiin tietyissä tapauksissa pienentää järjestettävän joukon kokoa jakamalla se etukäteen pienempiin nippuihin. Näin pystytään tehostamaan järjestämistä kokonaisuudessaan.

Vaihtoehtona perinteisille järjestämisalgoritmeille on käsitellä suuruusjärjestystä verkkona. Seuraavassa luvussa käsittelemme verkkoalgoritmeja, joita voi käyttää hyväksi suuruusjärjestyksen mallintamisessa ja käsittelyssä.

Luku 3

Verkkoalgoritmit

Koska perinteiset järjestysalgoritmit ovat herkkiä muutoksille ja virheille, tarvitsemme notkeamman tavan mallintaa suuruusjärjestystä. Luonnollinen tapa mallintaa suuruusjärjestystä on käyttää järjestettäviä alkioita verkon solmuina ja muodostaa näiden välille suunnattuja kaaria merkitsemään solmujen välistä suuruusjärjestystä.

3.1 Suuruusjärjestys verkkona

Määritellään suunnattu verkko, jossa ei saa esiintyä silmukoita ja kaaren (v, w) olemassa olo edustaa solmujen v ja w välistä relaatiota $v < w$. Kaaret siis osoittavat aina pienemmästä solmusta suurempaan solmuun.

Suuruusjärjestyksen mallintaminen suunnattuna verkkona vaatii sen, että verkossa ei ole silmukoita. Jos verkossa on silmukoita, pienimmän ristiriitaisen kaarijoukon etsimistä kutsutaan minimaalisen takaisinkytkentäjoukon ongelmaksi [14]. Tavoitteena on aina löytää pienin mahdollinen joukko kaaria, jotka poistamalla verkossa ei esiinny silmukoita. Merkittävä osa tässä luvussa esitellyistä verkkoalgoritmeista vaatii verkon olevan silmukaton, joten tämän ominaisuuden ylläpito on algoritmin kannalta olennaista.

Algoritmi voi ottaa kaksi mahdollista lähestymistapaa takaisinkytkentäjoukon ratkaisemiseksi: joko algoritmi käsittelee takaisinkytkentöjen poiston sellaisen vaiheen osana, jolloin verkon täytyy olla silmukaton, tai sitten algoritmi päättelee kaaren lisäyksen yhteydessä miten ristiriitaiset kaaret tulisi käsitellä. Ensimmäisessä tapauksessa voidaan soveltaa tehokasta algoritmia takaisinkytkentöjen poistamiseen [14] ja jälkimmäisessä tapauksessa voidaan esimerkiksi kaarien painoja käyttämällä vaikuttaa helposti siihen, miten algoritmi käsittelee ristiriitaisen syötteen.

3.2 Turnaus

Turnaukseksi kutsutaan suunnattua verkkoa, jossa jokaisen solmuparin välillä on täsmälleen yksi suunnattu kaari. Jos turnauksessa ei ole silmukoita, turnaus on transitiivinen, eli verkon kaarille pätee sääntö $(a,b) \wedge (b,c) \Rightarrow (a,c)$ [3, s. 3]. Turnauksessa olevien kaarien määrä on $\binom{a}{b}$, eli $\frac{1}{2}(n-1)n$, eli vaativuuslaskelmien yhteydessä täydessä turnauksessa on $O(n+\frac{1}{2}(n-1)n)$, eli luokkaa $O(n^2)$ kaarta. Lisäksi jokaisessa turnauksessa on vähintään yksi Hamiltonin polku, joka käy läpi jokaisen verkon solmun suunnattuja kaaria pitkin [3]. Täten suunnatussa turnauksessa on siis suuruusjärjestys.

Koska kappaleessa 3.5 esitellyn Tarjanin algoritmin aikavaativuus on $\Theta(V+E)$, voimme päätellä pahimman tapauksen turnaukselle. Täyden turnauksen läpikäynti on $O(n+\frac{1}{2}(n-1)n)$, eli luokkaa $O(n^2)$.

3.3 Dijkstran algoritmi

Verkon tehokas läpikäynti ja solmujen välisten polkujen löytäminen on olennaista, kun halutaan varmistaa ettei verkkoon synny silmukoita. Tähän tarkoitukseen on kehitetty useita algoritmeja [8, 9, 25, 36], mutta Dijkstran algoritmi [9] on hyvä perustavanlaatuinen esimerkki polunhakualgoritmista.

Dijkstran algoritmi aloittaa lähtösolmusta ja käy verkkoa läpi pitäen kirjaa läpikäytyjen polkujen pituuksista [9, s. 658-662]. Polun pituutena käytetään sen varrella kuljettujen kaarien painoarvoja. Algoritmi säilyttää läpikäytäviä solmuja painojen mukaisessa prioriteettijonossa, joka pitää huolen siitä että solmut käydään läpi lyhin kokonaispolku ensin. Kun algoritmi löytää kohdesolmun, se palauttaa löydetyn polun. Näin Dijkstran algoritmi löytää aina optimaalisen lyhyimmän polun. Algoritmin aikavaativuus on luokkaa $O(n \log n + m)$, missä n on solmujen lukumäärä ja m kaarien lukumäärä. Pahimmassa tapauksessa verkko on turnaus ja kaarien määrä on luokkaa $O(n^2)$, joten täydessä verkossa myös Dijkstran algoritmin keskimääräinen aikavaativuus kasvaa luokkaan $O(n^2)$.

3.4 Verkon läpikäynnit

Käyttämässämme verkossa ei ole kaarien pituuksia kuvaavia painoja. Tällaisessa tapauksessa Dijkstran algoritmi käy verkon läpi kuin se olisi leveyshakualgoritmi. Polkujen pituus on aina sama kuin polkuun kuuluvien kaarien lukumäärä.

3.4.1 Leveyshaku

Yksinkertainen leveyshaku on tehokkaampi painottamattoman verkon läpikäyntiin, koska Dijkstran algoritmiin verrattuna sillä ei ole tarvetta ylläpitää järjestettyä prioriteettijonoa [9]. Tämän vuoksi leveyshaun aikavaatimus painottamattomassa verkossa on $O(n + m)$, missä n on verkon solmujen määrä ja m on verkon kaarien lukumäärä. Kun aikavaativuutta vertaillaan Dijkstran algoritmiin, huomataan, että siitä puuttuu prioriteettijonon ylläpitoon kuluva $\log n$ termi. Vastaavasti täydessä verkossa leveyshaun aikavaativuus kasvaa luokkaan $O(n^2)$.

Leveyshaku käyttää verkon läpikäyntiin tavallista jonoa, johon työnnetään jokaisen solmun lapsisolmut ennen kuin algoritmi siirtyy käsittelemään jonon ensimmäistä solmua. Näin jokaisen solmun lapset käydään järjestyksessä läpi ennen kuin siirrytään käsittelemään solmun lapsenlapsia. Tämän vuoksi leveyshaku tuottaa polunhaussa aina lyhyimmän polun.

3.4.2 Syvyyshaku

Syvyyshaku on toinen klassinen algoritmi verkon läpikäyntiin. Se käy verkon leveyshaun tapaan, mutta jonon sijaan se käyttää pinoa läpikäytävien solmujen säilyttämiseen [9]. Syvyyshaku on myös helppo toteuttaa rekursiolla, sillä syvyyshakua voi kutsua suoraan solmun lapsisolmuille. Algoritmit merkitsee käsitellyt solmut, joten se ei käsittele samaa solmua useaan kertaan. Syvyyshaun aikavaativuus on samaa luokkaa leveyshaun kanssa: $O(n + m)$.

3.5 Topologinen järjestäminen

Suunnattu silmukaton verkko voidaan järjestää kaarien osoittamaan suuruusjärjestykseen käyttämällä yksinkertaista syvyyshakualgoritmia verkon solmujen läpikäyntiin. Cormen [9, s. 612-614] esittää algoritmin, jossa verkon solmut lisätään syvyyshaun valmistumisjärjestyksessä listaan:

Algoritmi 3.1 on samankaltainen mitä Tarjan [36] on esittänyt jo vuonna 1974. Silmukattoman verkon kaikki solmut käydään läpi syvyysjärjestämisellä ja solmut lisätään tuloslistan kärkeen valmistumisjärjestyksessä, jolloin ensimmäisen solmun hakupuun syvin solmu päätyy tuloslistan ensimmäiseksi. Tästä hakupuuta taaksepäin lukien solmut lisätään tuloslistaan ja lopulta kaikki verkon solmut on käyty topologisessa järjestyksessä läpi. Tehokkaampia algoritmeja on esitetty [5, 17, 18, 29], mutta Tarjanin algoritmi painottamattomille verkoille on yksinkertainen ja helppo toteuttaa myös verkossa, jota ei ole mallinnettu vierusmatriisina.

Algoritmi 3.1: Topologinen järjestäminen

```
List result = []
while there are unmarked nodes do
  select an unmarked node n
  visit(n)

visit(n)
  if n is not marked then
    for each node m with an edge from n to m do
      visit(m)
    mark n visited
    add n to head of result
```

Verkon järjestäminen vaatii $\Theta(V + E)$ ajan, eli algoritmin täytyy käydä läpi jokainen verkon solmu ja kaari päästäkseen lopulliseen varmuuteen solmujen keskinäisestä suuruusjärjestyksestä [9]. Toisaalta algoritmin ei tarvitse käydä läpi solmuja useaan kertaan, joten algoritmin suorituskyky on hyvin tarkkaan ennustettavissa.

3.6 Transitiivinen sulkeuma

Silmukattoman ja suunnatun verkon $G = (V, E)$ transitiivinen sulkeuma on määritelty verkkona $G+ = (V, E+)$, jossa jokaista kaarta (v, w) vasten on olemassa verkon G polku solmusta v solmuun w [9]. Transitiivinen sulkeuma siis täydentää verkkoa eteenpäin suuntautuneilla kaarilla, jotka pystytään päättelemään verkon olemassa olevien polkujen perusteella.

Määritellään että verkko joka sisältää kaikki samat kaaret kuin vastaavasta verkosta laskettu transitiivinen sulkeuma on *transitiivisesti sulkeutunut*.

Transitiivinen sulkeuma voidaan laskea naiivisti käymällä läpi jokaisesta verkon solmusta lähtevät polut ja lisäämällä puuttuvat kaaret lähtösolmusta polun varrella oleviin solmuihin. Esimerkiksi kappaleessa 3.3 esitelty Dijkstran algoritmi löytää lyhyimmät polut yhdestä solmusta kaikkiin saavutettavissa oleviin solmuihin. Suorittamalla syvyyshaku jokaiselle n solmulle verkossa, saadaan naiivin algoritmin tehokkuudeksi $O(n(n+m)$. Koska täydessä verkossa on enintään $O(n^2)$ kaarta, saadaan algoritmin pahimman tapauksen aikavaativuudeksi $O(n(n^2)) = O(n^3)$.

Nuutila [28] esittää vielä tehokkaamman algoritmin transitiivisen sulkeuman laskemiseen vahvasti yhtenäisten komponenttien avulla. Algoritmi käyt-

tää muunneltua Tarjanin algoritmia vahvasti yhtenäisten komponenttien etsimiseen ja samalla transitiivisen sulkeuman laskemiseen näiden komponenttien perusteella. Koska pyrimme pitämään verkon silmukattomana, vahvasti yhtenäisten komponenttien laskenta ei tuo meidän tapauksessamme toivottua parannusta transitiivisen sulkeuman laskemiseen.

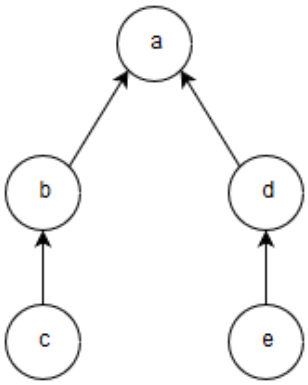
Floyd–Warshallin algoritmi käyttää vierusmatriisia ja laskee verkon transitiivisen sulkeuman kuutiollisessa ajassa $O(n^3)$ [15, 21]. Jos transitiivisen sulkeuman laskemiseksi voidaan käyttää vierusmatriisia, voidaan transitiivisen sulkeuman laskentaa käyttää matriisikertolaskua, jonka tämän hetken tehokkaimmat algoritmit suorittavat ajassa $O(n^{2.688})$ [6, 39]. Nämä on kuitenkin jätetty tämän työn ulkopuolelle.

3.7 Verkon leikkaukset

Haluamme saada selville mitkä verkon solmut ovat erillään muista, jotta voimme selvittää minkä solmujen välille kannattaa lisätä kaaria siten että verkko tiivistyy mahdollisimman paljon. Koska käytämme suunnattua verkkoa, verkon yhtenäisten komponenttien laskemisesta ei ole paljon hyötyä, sillä yhtenäiset komponentit lasketaan vain kaarien perusteella huomioimatta kaarien suuntaa [20]. Esimerkiksi verkossa 3.1 on vain yksi yhtenäinen komponentti, mutta algoritmin kannalta olisi oleellista tietää, että solmut c ja e ovat suunnattujen kaarien näkökulmasta erossa toisistaan.

Toinen vaihtoehto olisi käyttää Tarjanin vahvasti yhtenäisten komponenttien laskentaa suunnatuille verkoille [35]. Vahvasti yhtenäisten komponenttien laskeminen vaatii verkon, jossa on suunnattuja silmukoita. Pyrimme välttämään silmukoita verkossa topologisen järjestämisen vuoksi, joten tämäkään ei ole optimaalinen ratkaisu ongelmaan.

Kolmanneksi olisi mahdollista lähestyä ongelmaa laskemalla verkosta maksimaalinen itsenäinen joukko, eli suurin joukko solmuja, jotka eivät ole toistensa vieressä verkossa [10, 26]. Helpoiten tämä onnistuu muodostamalla verkosta suuntaamaton verkko ja etsimällä klikit sen komplementtiverkosta. Tässä komplementtiverkossa on siis kaari jokaisen solmuparin välillä, missä alkuperäisessä verkossa ei ole kaarta ja vastaavasti komplementtiverkossa ei ole kaarta niiden solmuparien välillä missä alkueräisessä verkossa on kaari. Jos komplementtiverkosta lasketaan maksimaalinen klikki, niin se vastaa alkuperäisessä verkossa maksimaalista itsenäistä joukkoa, eli joukkoa solmuja, joiden välillä ei ole kaaria. Tämän joukon yhdistäminen kaarilla lisäisi verkkoon merkittävästi uusia kaaria, mutta pelkästään itsenäisen joukon laskeminen ja näiden välille kaarien luominen ei annan algoritmille mahdollisuutta



Kuva 3.1: Verkko, jossa on yksi yhtenäinen komponentti.

vahvistaa olemassa olevia kaaria. Lisäksi maksimaalisen klikin löytäminen on NP-vaikea ongelma [10].

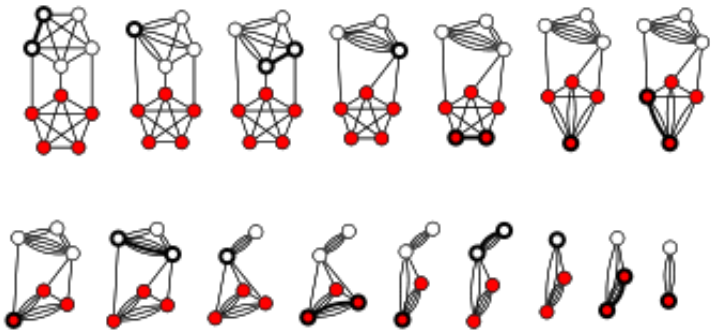
Yksinkertaisempi lähestymistapa on käsitellä tilannetta verkon minimaalisen leikkauksen ongelmana. Minimaalinen leikkaus on pienin joukko verkon kaaria, jotka poistamalla verkko jakaantuu kahteen osaverkkoon. Tällöin voidaan käänteisesti ajatella, että lisäämällä kaaria näiden osaverkkojen välille kaaria, voidaan vahvistaa verkkoa kokonaisuudessaan.

Karger [22] on esittänyt minimaalisen leikkauksen laskemiseen tehokasta satunnaistettua algoritmia, joka valitsee verkosta sattumanvaraisesti yhden kaaren ja yhdistää sen päissä olevat solmut yhdeksi supersolmuksi, jolla on vastaavat ulkoiset kaaret kuin aiemmilla kahdella erillisellä solmulla. Ainoastaan solmujen väliset kaaret poistetaan verkosta. Tästä seuraa se, että verkkoon kertyy supersolmupareja, joiden välillä on useita kaaria ja useat kaaret lisäävät todennäköisyyttä tulla valituksi seuraavassa iteraatiossa. Kuvassa 3.2 on kuvattu, miten algoritmi etenee valiten sattumanvaraisen kaaren ja yhdistäen sen solmut.

Keskimäärin Kargerin algoritmi todennäköisesti valitsee aina solmuparin, jonka välillä on eniten kaaria, joten "vahvemmin kytketty"solmupari yhdistetään supersolmuksi [22]. Kun algoritmia suoritetaan riittävän pitkään, jäljellä on vain kaksi supersolmua, jotka edustavat kahta osaverkkoa ja niiden välillä ovat kaaret, jotka kuuluvat verkon minimaaliseen leikkaukseen.

3.8 Yhteenveto

Tässä luvussa käsittelimme miten suuruusjärjestyksen voi mallintaa suunnattuna verkkona ja lopullisen suuruusjärjestyksen selvittämiseksi tavoitte-



Kuva 3.2: Kargerin algoritmin edistyminen. [38]

lemme täyttä turnausverkkoa. Kävimme läpi perustavanlaatuisia algoritmeja verkon läpikäyntiin, topologiseen järjestämiseen ja heikoimmin kytkettyjen solmujen tunnistamiseen. Seuraavassa luvussa käsittelemme muuttuvaa verkkoa, johon lisätään ja josta poistetaan kaaria, ja miten algoritmit voidaan pitää tehokkaina dynaamisessa ympäristössä.

Luku 4

Dynaamiset ongelmat ja ratkaisut

Tässä luvussa käymme läpi miten eri algoritmit ovat ratkaisseet dynaamisesti muuttuvan tiedon. Aluksi esittelemme Monkeysortin, joka perustuu pikajärjestämiseen ja interaktiivisesti kysyy käyttäjältä syötettä kuten tavoittelemamme algoritmi. Lisäksi käymme läpi eri verkkoalgoritmeja, jotka ovat erikoistuneet toimimaan muuttuvassa verkossa, johon lisätään ja josta poistetaan kaaria. Dynaamisen verkon algoritmit tyypillisesti ratkaisevat samoja ongelmia kuin edellisessä luvussa kuvatut staattiset algoritmit, ja dynaamisen verkon muutokset voidaan naiivisti ratkaista suorittamalla vastaava staattinen algoritmi verkolle jokaisen muutoksen jälkeen. Käyttämällä dynaamisia algoritmeja on kuitenkin mahdollista saavuttaa tehokkuushyötyjä, sillä tavallisesti verkon muutos ei koske kaikkia solmuja, joten algoritmille riittää osaverkon käsittely.

4.1 Perinteiset algoritmit sovellettuina interaktiivisiksi

Olisi mahdollista käyttää jotain perinteisistä järjestämisalgoritmeista kuten lomitusjärjestämistä tai pikalajittelua, ja kysyä interaktiivisesti jokaisen vertailuoperaation kohdalla käyttäjältä tämän syötettä.

4.1.1 Monkeysort

Monkeysort käyttää kappaleessa 2.1 kuvattua pikalajittelua arbitraarisen listan järjestämiseen [34]. Algoritmi täyttää järjestämisen aikana vertailutaulukkoa, jossa on listattu alkioiden välisten vertailujen tulokset. Sen vuoksi käyttäjältä voidaan kysyä kahden alkion välinen suuruusjärjestys kerran ja algoritmi muistaa tämän jatkovertailuja varten.

Pikalajittelussa pyritään absoluuttiseen tehokkuuteen, joten siinä on optimoitu nimenomaan tehtävien vertailujen määrä mahdollisimman pieneksi. Tällaisissa tehokkaissa lajittelualgoritmeissa jokainen vertailu on alkion sijoittumisen kannalta merkittävä. Jos alkioiden välisessä vertailussa tapahtuu yksikin virhe, ei lopputulos vastaa sitä mitä käyttäjä on tarkoittanut. Siispä pikalajittelu, vaikka onkin tehokas, sopii huonosti tilanteeseen missä erehtyväinen ihminen tekee vertailut.

Monkeysort tarjoilee käyttäjälle kaksi vaihtoehtoa kerrallaan vertailtavaksi ja tallettaa saadun tuloksen vertailumatriisiin myöhempää tarvetta varten [34]. Jos vertailutauluun eksyy yksikin virheellinen vertailutulos, niin se vaikuttaa merkittävästi lopputulokseen, koska käyttäjällä ei ole mahdollisuutta korjata virhettään. Otetaan esimerkiksi tilanne, jossa käyttäjä haluaa järjestää luonnolliset luvut $(1, 2, 3, 4, 5, 6, 7, 8, 9)$ ja ensimmäiseksi sarana-alkioksi valikoituu 5. Jos käyttäjä tekee virheen vertailussa 5 ja 9 välillä, 9 päätyy sarana-alkiota pienempään alilistaan. Koska pikalajittelu ei tarkista alkioiden järjestystä enää yhdistämisvaiheessa, alkio 9 voi päätyä korkeintaan alemman alilistan suurimmaksi. Tällöin sen yläpuolelle päätyvät sekä sarana-alkio 5, että kaikki ne alkiot, jotka käyttäjä on hajotusvaiheessa arvioinut sarana-alkiota suurempien alkioiden alilistaan. Jos käyttäjä ei tee muita virheitä, niin edellä kuvatun erheellisen vertailun sisältämän järjestämisen lopputulos on $(1, 2, 3, 4, 9, 5, 6, 7, 8)$.

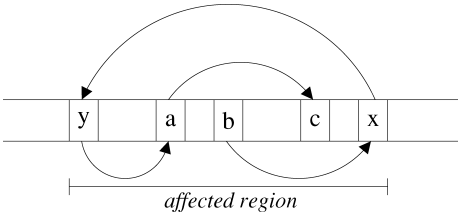
4.2 Inkrementaalinen topologinen järjestäminen

Jos suuruusjärjestystä kuvaavaan verkkoon tehdään muutoksia ja verkon suuruusjärjestys on jo valmiiksi selvitetty, tarvitsemme algoritmin laskemaan muutosoperaation aiheuttamat vaikutukset alkioiden väliseen valmiiksi laskettuun topologiseen suuruusjärjestykseen.

Koska topologinen järjestäminen perustuu yksinomaan verkon solmujen välisten kaarien ominaisuuksiin, tarvitsemme inkrementaalisen topologisen järjestysalgoritmin, joka pystyy käsittelemään sekä kaarien lisäämisen että poistamisen verkosta.

Topologisen järjestyksen päivittäminen on hyvin tunnettu ongelma, jota sovelletaan useilla eri tietotekniikan osa-alueilla, kuten piirilevyanalyysissa [1], tietorakenneanalyysissa [29] ja lukkiutumien tunnistamisessa [4].

Pearce ja Kelly [29] tuovat esille kaksi merkittävää havaintoa suunnattujen verkkojen topologisesta järjestyksestä: ensinnäkin kaaren poistaminen ei muuta verkon topologista suuruusjärjestystä millään tavalla. Solmuilla on ennen kaaren poistoa selkeä suuruusjärjestys, mutta kaaren poiston jälkeen



Kuva 4.1: PK-algoritmin tilanne, jossa valmiiseen topologiseen järjestykseen lisätään kaari (x, y) [29].

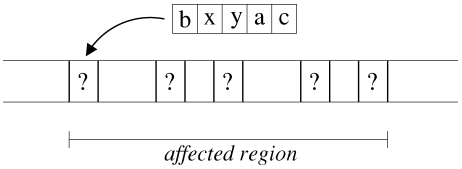
niillä ei ole enää suuruusjärjestystä, joten jo valmiiksi laskettu järjestys on validi.

Toinen Pearcen ja Kellyn [29] huomio on, että kaaren lisäyksen yhteydessä täytyy päivittää vain siinä tapauksessa, että se on ristiriidassa nykyisen järjestyksen kanssa. Jos kolmisolmuisen verkon nykyinen topologinen suuruusjärjestys on (a, b, c) ja verkkoon lisätään kaari (a, c), ei suuruusjärjestys muutu, eikä topologista järjestämisestä tarvitse laskea uudestaan. Sen sijaan kaaren (b, a) lisääminen vaatii topologisen järjestyksen päivittämisen. Tätä varten he esittelevät uuden algoritmin, jonka he nimeävät PK-algoritmiksi [29].

PK-algoritmi suorittaa lisättävästä kaaresta kaksi syvyyshakua, jotka etsivät verkosta muutoksen vaikutuspiirissä olevat solmut [29]. Toinen käy läpi solmut lisättävästä kaaresta taaksepäin ja toinen läpikäy verkkoa kaaresta eteenpäin. Näiden hakujen löytämien solmujen sijainnit täytyy laskea uudestaan. Koska syvyyshaut ovat hakeneet kaikki solmut, joiden sijainnit riippuvat toisistaan topologisessa järjestyksessä, voimme asettaa vaikutuspiirissä olevat solmut keskinäiseen suuruusjärjestykseen ja latoa ne päivitetyssä järjestyksessä vanhoille paikoilleen järjestyksessä olevassa listassa.

Kuva 4.1 esittää tilannetta, jossa valmiiksi topologisessa järjestyksessä olevaan listaan lisätään kaari (x, y) ja tämä aiheuttaa järjestyksen muutoksen. Syvyyshakujen tuloksena solmut (a, b, c, x, y) ovat yhteydessä solmuihin x ja y, joten niiden keskinäinen suuruusjärjestys täytyy laskea uudestaan. Tuloksena saadaan (b, x, y, a, c), jotka kuvassa 4.2 ladotaan takaisin suuruusjärjestykseen oikeassa järjestyksessä.

Bender et al. [5] on jatkokehittänyt yllämainittua Pearcen ja Kellyn [29] algoritmia erityisesti tilanteissa, joissa tiheään verkkoon luodaan suuruusjärjestyksen vastainen kaari. Parannettu algoritmi käyttää hyväkseen jo valmiiksi laskettua suuruusjärjestystä ja ylläpitää listaa eteenpäin suuntautuvista kaarista, joiden avulla pystytään päättelemään ne solmut, joihin uuden kaaren lisääminen vaikuttaa. Koska algoritmi käy läpi valmiiksi suuruusjär-



Kuva 4.2: Järjestettyjen alkioiden latominen aikaisemmille paikoileen [29].

jestyksessä olevaa listaa, sen ei tarvitse välittää ylimääräisten kaarien läpikäynnistä. Tämä tekee Benderin et al:n algoritmista tehokkaamman varsinkin tiheässä verkossa.

4.3 Transitiivisen sulkeuman naiivi päivittäminen dynaamisesti

Transitiivisen sulkeuman dynaamisesti päivittävällä algoritmilla on oltava kolme funktiota: lisäys, poisto ja kysely [11]. Lisäyksellä ja poistolla muokataan transitiivista sulkeumaa lisäämällä tai poistamalla alkuperäisestä verkota sekä sen transitiivisesta sulkeumasta kaaria ja kyselyllä haetaan algoritmilta tulos: onko kahden solmun välillä transitiivista suhdetta.

Verkosta voidaan tallentaa erillinen transitiivinen sulkeuma [13] tai verkkoon itseensä voidaan tehdä kaarten lisäyksiä ja poistoja, jotka pitävät huolen verkon transitiivisuuden ylläpidosta.

4.3.1 Transitiivisen sulkeuman päivittäminen kaaren lisäyksen yhteydessä

Pienessä ja hajanaisessa verkossa transitiivisen sulkeuman päivittämisoperaatio on melko kevyt, sillä ainoat solmut joihin päivitysoperaatio koskee, ovat suoria vanhempia tai jälkeläisiä niille solmuille, joiden välille uusi kaari verkkoon lisätään [13].

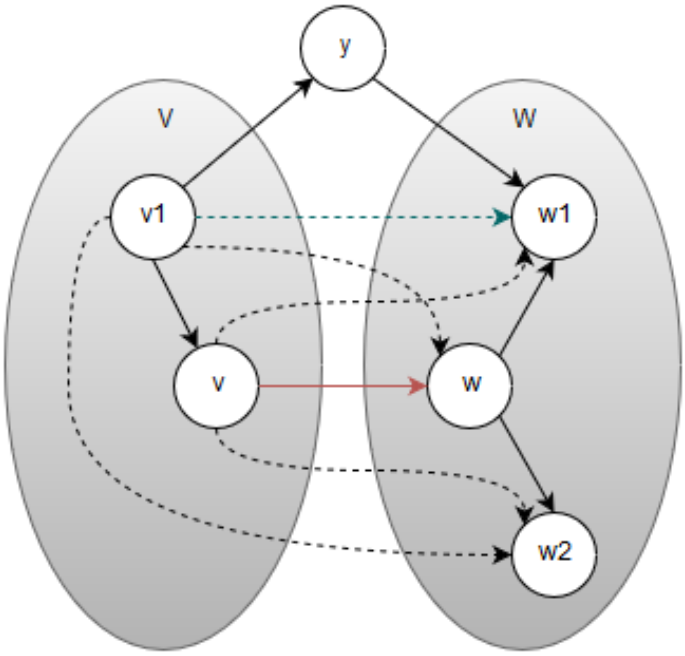
Kaareton verkko on lähtökohtaisesti transitiivisesti sulkeutunut ja määritämme jokaisen lisäys ja poisto-operaation ylläpitävän verkon transitiivisuutta käyttämällä operaatioita, jotka ylläpitävät transitiivista sulkeumaa [13]. Tällöin verkon on oltava jokaisen operaation jälkeen transitiivisesti sulkeutunut, eli verkossa ei ole sellaista polkua $(v...w)$, jolle ei olisi olemassa vastaavaa kaarta (v, w).

Kuva 4.3 havainnollistaa joukkojen V ja W suhtautumisen toisiinsa. Transitiivisen sulkeuman täydentävä algoritmi 4.1 käyttää hyväkseen edellä mai-

Algoritmi 4.1: Transitiivisen sulkeuman lisäysoperaatio

```
1 hae lähtösolmuun v saapuvien kaarien lähtösolmut V
2 hae kohdesolmusta w lähtevien kaarien kohdesolmut W
3 lisää solmut v ja w vastaaviin joukkoihin V ja W
4 jokaiselle solmulle a joukossa V
5   jokaiseen solmuun b joukossa W
6     luo transitiivinen kaari a -> b
```

nittua ominaisuutta siitä, että jokaisesta solmun esivanhemmasta joukossa V on oltava kaari solmuun v ja vastaavasti kohdesolmusta w on kaaret jokaiseen sen jälkeläiseen joukossa W. Solmu ei voi kuulua sekä joukkoon V että W, sillä muuten verkossa olisi olemassa transitiivinen kaari (w, v) ja tietäisimme, että kaaren lisääminen aiheuttaisi silmukan. Tällöin verkkoon ei voida lisätä kaarta (v, w), sillä tämän jälkeen verkko ei enää kuvaisi yksikäsitteistä suuruusjärjestystä.



Kuva 4.3: Kaaren (v, w) lisääminen transitiivisesti sulkeutuneeseen verkkoon. Transitiiviset kaaret on merkitty katkoviivoilla.

4.3.2 Kaaren poisto ja transitiivisuuden päivittäminen

Transitiivisen sulkeuman päivittäminen on hankalampi operaatio, kun kaari poistuu verkosta. Tässäkin tapauksessa voimme hyödyntää transitiivisesti sulkeutuneen verkon ominaisuutta, jonka mukaan sekä lähde että kohdesolmusta on kaari jokaiseen saavutettavissa olevaan esivanhempaan ja jälkeläiseen. Saamme siis mahdollisesti poistettavan kaarijoukon laskemalla kaikki poistetun kaaren lähtösolmusta tai sen esivanhemmista lähtevät kaaret, jotka päätyvät johonkin poistetun kaaren kohdesolmun tai sen lapsisolmuihin.

Otetaan esimerkiksi kuvan 4.3 tilanne, jossa poistetaan verkosta kaari (v, w). Jokaiselle tarkasteltavalle kaarelle lähtöjoukon V ja kohdejoukon W välillä täytyy tehdä kaksi tarkistusta. Ensinnäkin kaaren täytyy olla transitiivisesti lisätty. Toiseksi kaaren lähtösolmusta ei saa johtaa mikään muu polku kohdesolmuun. Kuvassa 4.3 on merkitty vihreällä transitiivinen kaari $(v1, w1)$, jonka olemassaolo ei riipu pelkästään kaaren (v, w) olemassa olosta, vaan se on vahvennettu polulla $(v1, y, w1)$. Tätä transitiivista kaarta ei siis tule poistaa transitiivisesta sulkeumasta.

Algoritmi 4.2: Transitiivisen sulkeuman päivittäminen poisto-operaation yhteydessä

```
1 hae lähtösolmuun v saapuvien kaarien lähtösolmut V
2 hae kohdesolmusta w lähtevien kaarien kohdesolmut W
3 lisää solmut v ja w vastaaviin joukkoihin V ja W
4 jokaiselle solmulle a joukossa V
5   jokaiseen solmuun b joukossa W
6     jos muuta polkua (a...b) ei ole olemassa
7       poista transitiivinen kaari
```

Tämän algoritmin pahimman tapauksen aikavaativuus perustuu siihen, että jokainen verkon solmu kuuluu joko joukkoon V tai W. Tällöin algoritmi käy läpi $O(\frac{n}{2}\frac{n}{2}) \approx O(n^2)$ solmua riveillä 4 ja 5. Lisäksi tarkistusvaiheessa rivillä 6 täytyy suorittaa jokin polunhakualgoritmi, kuten kappaleessa 3.4 esitelty syvyyshaku, joka suoriutuu polunhausta täydessä verkossa ajassa $O(n^2)$. Tällöin tämän naiivin päivitysoperaation aikavaativuudeksi tulee $O(n^2 * n^2) \approx O(n^4)$. Naiivi algoritmi ei siis ole järin tehokas varsinkaan täysissä verkoissa.

4.3.3 Probabilistiset algoritmit

Transitiivisen sulkeuman ylläpito on mahdollista saada tehokkaammaksi kuin optimoitu transitiivisen sulkeuman tyhjästä laskeva algoritmi, mutta ensimmäisiin yrityksiin sisältyy mahdollisuus virheisiin. Kingin ja Sagertin [24] al-

goritmi parantaa transitiivisen sulkeuman laskentaa pitämällä kirjaa solmuparien välillä olevista poluista ja pudottamalla polkujen kirjanpidossa käytettyä sanapituutta algoritmia voitiin tehostaa, mutta samalla mahdollisuus virheeseen kasvoi.

Kingin ja Sagertin algoritmi ei takaa todenmukaista tulosta, kun algoritmin tulos transitiivisen suhteen olemassa ololle on "ei", eli algoritmiin voi luottaa aina kun se löytää transitiivisen suhteen, mutta jos algoritmi ei löydä transitiivista suhdetta, suhde saattaa silti olla olemassa keskimäärin todennäköisyydellä $O(1/n^{log(n)})$ [24].

4.3.4 Tehokkaammat transitiivisen sulkeuman deterministisesti laskevat algoritmit

King [23] onnistui parantamaan aiempaa Kingin ja Sagertin [24] algoritmia ja poistamaan virheen mahdollisuuden algoritmista. Päivitetty algoritmi tallettaa jokaisesta solmusta eteen ja taaksepäin suuntautuvat puut, jotka listaavat solmusta lähteviä polkuja pitkin saavutettavat muut solmut. Lisäys- ja poisto-operaatiossa näitä puurakenteita päivitetään vastaamaan verkon rakennetta.

Yllämainittuun malliin on tehty parannuksia ja jatkokehitystä: Demetrescu ja Italiano [12] jatkokehittivät algoritmia käyttämällä tehokasta matriisihakua, joka tehosti erityisesti kaarien lisäysoperaatioita ja Frigione et al. [16] puolestaan kehittivät tätä algoritmia pidemmälle. Heidän työnsä pohjalta Roddity [31] esitti algoritmin, joka yltää päivittää transitiivisen sulkeuman ajassa $O(mn + ins * n^2 + del)$, missä n on solmujen määrä, m on verkon kaarien määrä, ins on suoritettujen lisäysten määrä ja del verkkoon tehtyjen poistojen määrä.

4.4 Yhteenveto

Tässä luvussa käytiin läpi muuttuvaa verkkoa käsitteleviä algoritmeja, joiden tavoite on tehokkaasti päivittää algoritmin lopputulosta, kun verkkoon lisätään tai siitä poistetaan kaaria. Esittelimme algoritmit transitiivisen sulkeuman ja topologisen järjestyksen ylläpitoon, sillä niitä tullaan tarvitsemaan kappaleessa 6, kun keskitymme itse algoritmin toteutukseen.

Luku 5

Interaktiivinen järjestäminen

Algoritmille syötettä tarjoava käyttäjä on erehtyväinen, joten algoritmin on sopeuduttava muuttuvaan tilanteeseen [27]. Lisäksi käyttäjä on hidas suhteutettuna tietokoneen suorittamaan laskentaan ja toisaalta käyttäjän aika ja keskittymiskyky ovat rajallisia. Koska käyttäjä on loppukädessä koko algoritmin merkittävästi hitain osa, algoritmin ei tarvitse olla tehokas erityisen suurilla syötejoukoilla. Algoritmin on kuitenkin oltava niin mukautuvainen, että se voi ottaa syötteenä epävarmaa dataa ja silti tuottaa hyödyllisen lopputuloksen. Algoritmin voidaan sanoa olevan inkrementaalinen, koska se päivittää lopputulostaan jatkuvasti saadun syötteen perusteella [1].

5.1 Käyttäjän syöte

Oletetaan, että järjestämistehtävän kohteena olevilla alkioilla on yksikäsitteinen suuruusjärjestys, eli joukossa ei ole kaksoiskappaleita ja jokaiselle alkioparille a, b pätee $(a < b) \vee (a > b)$. Jos käyttäjä ei tarkoita suuruusjärjestyksessä olevan silmukoita, voimme todeta alkioiden välisen suuruusjärjestyksen olevan luonteeltaan transitiivinen relaatio. Voimme siis hyödyntää transitiivisen sulkeuman laskentaa ja täydentämää verkkoa olemassa olevien kaarien perusteella pääteltävillä transitiivisilla kaarilla.

Algoritmin perusaskel on syötteen kysyminen käyttäjältä: käyttäjälle tarjotaan jokin järjestettävän alkiojoukon osajoukko arvioitavaksi ja käyttäjä valitsee näistä mieluisimman voittaja-alkioksi. Tällöin algoritmi osaa päätellä tarjottujen alkioiden ja näistä valikoidun voittaja-alkion välisen suuruusjärjestyksen.

Algoritmi tallettaa suuruusjärjestyksen suunnattuna verkkona ja päättelee verkon perusteella, minkä solmuparien väliltä vielä puuttuu kaari. Jotta kaikille solmuille saadaan yksikäsitteinen suuruusjärjestys, algoritmi kysyy

käyttäjältä osajoukon toisensa jälkeen, kunnes verkko on täydennetty ja algoritmi osaa sanoa järjestystehtävän alkioiden suuruusjärjestyksen.

5.1.1 Käyttäjälle tarjottavan alkiojoukon koko

Käyttäjälle tarjottavan alkiojoukon koko vaikuttaa merkittävästi sekä siihen miten tehokkaasti algoritmi päätyy toivottuun lopputulokseen, että siihen miten vaikeaa käyttäjän on valita paras tarjotusta osajoukosta. Jos osajoukon sijaan käyttäjälle tarjotaan koko joukon kaikki alkiot, ja virheetön käyttäjä valitsee aina suurimman alkion, algoritmi on ohi n iteraatiossa. Tällöin käyttäjä valitsee ensin kaikista solmuista sopivimman ja jatkaa valitsemaan lopuista solmuista sopivimman ja niin edelleen, kunnes kaikki n solmua on käyty läpi. Vastaavasti jos käyttäjälle tarjotaan valittavaksi liian pieniä osajoukkoja, käyttäjä joutuu tekemään huomattavan paljon vertailuja.

Ihmisen kyvystä tehdä tehokkaita valintoja on tehty paljon tutkimusta. Rodriguez on tutkinut oppilaille järjestettävien monivalintatestien kysymysmääriä [32]. Monivalintatehtävät ovat analoginen ongelman algoritmin tilanteen kanssa, sillä mitä tiiviimmin nopeammin opiskelija pystyy valitsemaan vastauksen kysymykseen ja jatkamaan seuraavaan, sitä enemmän kysymyksiä koepaperiin voidaan laittaa ratkaistavaksi. Hänen analyysinsa mukaan kolme vaihtoehtoa on tehokkain määrä vaihtoehtoja oppilaiden tehokkuuden ylläpitämiseen.

Psykologian puolella ongelmaa on tutkittu paljon, mutta on vaikea löytää yleispätevää parasta tapausta päätöksenteon vaihtoehtojen määrälle. Chernev et al. [7] tutki meta-analyysillään valintaylikuormitusta ja tuli siihen tulokseen, että on neljä hyvää mittaustapaa, joilla valintaylikuormitusta voi mitata, mutta tulokset vaihtelevat testitapauksittain. Hick [19] puolestaan on esittänyt Hickin lakina tunnetun ilmiön, jossa ihmisen tekemä päätöksenteko hidastuu logaritmisesti suhteessa vaihtoehtojen määrään. Tämä pätee kuitenkin pelkästään helpoissa ja intuitiivisissa päätöksissä, kuten näytettyä koodia vastaavan näppäimen painamisessa, joten se ei sovellu algoritmin tarvitsemaan monimutkaiseen evaluointiin [33].

Erinomaisena esimerkkinä monimutkaisten valintojen tekemisestä on lääkäreille tehty tutkimus, jossa koehenkilöiden piti päättää minkä hoidon he valitsisivat kuvaillulle potilaalle [30]. Redelmeierin ja Shafirin [30] havainto oli, että lääkärit, joille esitettiin kaksivaihtoehtoa: lääkitys ja vaihtoehtohoito, olivat tyypillisesti lääkityksen kannalla. Jos kolmanneksi vaihtoehdoksi lisättiin toinen samankaltainen lääkitys, eivät lääkärit kuitenkaan valinneet kumpaakaan lääkityshoidoista, vaan valitsivat kolmannen vaihtoehtohoidon, jota kahdenkeskisissä vertailuissa ei tyypillisesti valittu. Vaihtoehtojen lisääminen siis suoraan muutti lääkäreiden päätöstä.

Lopullista suuruutta käyttäjälle tarjottavien vaihtoehtojen parhaimmalle määrälle on varmasti hyvin vaikea saada, joten tässä työssä on käytetty turvallisena oletuksena kolmea vaihtoehtoa. Käyttäjälle siis tarjotaan kolme järjestettävistä alkioista ja käyttäjä valitsee niistä parhaimman. Algoritmia testatessa kolme on empiirisesti havaittu hyväksi erityisesti siksi, että vaihtoehdot on nopea lukea läpi ja valinnan teko on nopeaa esimerkiksi eliminoimalla vähiten mieluisa vaihtoehto.

5.2 Virheet syötteessä

Käyttäjän antama syöte on algoritmin keskeisin osuus ja sen yhteydessä täytyy varautua siihen, että käyttäjä voi tehdä virheitä. Virheet saattavat olla näppäilyvirheitä, loogisia päättelyvirheitä tai käyttäjä saattaa muuttaa mieltänsä algoritmin käytön aikana.

Olemme tunnistaneet kolme merkittävää tapaa, joilla käyttäjä voi aiheuttaa virheen algoritmin suorituksessa. Ensinnäkin, jos käyttäjältä kysytään solmuparin välistä järjestystä useaan kertaan, käyttäjä saattaa antaa eri kerroilla ristiriitaisen vastauksen. Toiseksi käyttäjä saattaa muodostaa verkkoon useamman solmun silmukan, jolloin silmukattomassa verkossa toimiviksi suunnitellut verkkoalgoritmit eivät enää toimikaan. Kolmanneksi käyttäjä voi jättää algoritmin suorittamisen kesken, jolloin käyttäjän siihen asti antamasta syötteestä pitää pystyä tuottamaan hyödyllisiä tuloksia.

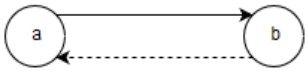
5.2.1 Käyttäjän syötteen ristiriidat

Edellytys aiemmin esitetyille algoritmeille on se, että verkossa säilyy suuruusjärjestys. Tämä puolestaan edellyttää sitä, että verkkoon ei kaaria lisätessä muodostu silmukoita.

Jos algoritmi kysyy samaa vertailua kahteen kertaan, käyttäjä voi antaa aiemmasta kannastaan poikkeavan vastauksen, eli käyttäjä on aiemmin antanut syötteen $a > b$ ja jälkeenpäin samaa alkioparia kysyttäessä käyttäjä vaihtaakin mieltymystään muotoon $b > a$

Silmukoiden tunnistusta varten on kaksi vaihtoehtoa: verkosta pidetään yllä ajantasaista transitiivista sulkeumaa (kappale 4.3), josta voidaan todeta kun verkossa on jo olemassa oleva kaari (kuva 5.1) tai transitiivinen kaari $e^- = (w, v)$ (kuvat 5.2 ja 5.3). Koska tämä kaari on vastakkaissuuntainen lisättävään kaareen nähden, verkkoon muodostuisi silmukka.

Toinen vaihtoehto on käyttää jokaisen kaaren lisäyksen yhteydessä polunhakualgoritmia, joka etsii verkosta olemassa olevan polun $p = (w, ..., v)$,



Kuva 5.1: Kaaren (b, a) lisääminen, kun solmujen välillä on jo olemassa oleva kaari (a, b).

jolloin tiedämme että kaaren lisääminen aiheuttaisi silmukan muodostumisen. Tässä lähestymistavassa on se hyvä puoli, että ristiriidan käsittelyssä olemme jo selvittäneet ristiriitaiseen polkuun p kuuluvat solmut. Huonona puolena on se, että ristiriitaisia polkuja voi olla useita kuten kuvassa 5.3.

5.2.1.1 Silmukoiden käsittely

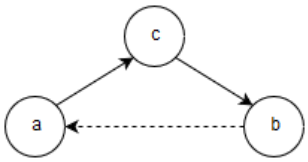
Kun verkkoon yritetään lisätä kaarta, joka loisi silmukan, täytyy ristiriita käsitellä jollain tavalla. Ristiriidan käsittelyyn on triviaalisti pääteltävissä kaksi tapaa: kaaren lisääminen voidaan estää tai ristiriitaiset polut täytyy muokata yhteensopiviksi uuden informaation kanssa.

Kaaren lisäämättä jättäminen on yksinkertaisin tapa. Oletuksena on se, että käyttäjä ei ole tehnyt virhettä aikaisemmin, joten virheen täytyy olla viimeisimmässä lisäyksessä, joten kaarta ei lisätä verkkoon. Todellisuudessa käyttäjä on kuitenkin saattanut tehdä virheen jo aikaisemmin, joten jokin aiemmin lisätyistä kaarista ja sen avulla päätellyt transitiiviset kaaret ovat virheellisiä. Esimerkiksi kuvassa 5.2 joko kaari (a, c) tai (c, b) voi olla virheellisesti lisätty.

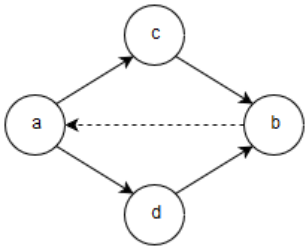
Ratkaisuna tähän aiemmin lisätyn virheellisen kaaren ongelmaan on se, että verkosta etsitään olemassa oleva polku p, joka muodostaa lisättävän kaaren kanssa silmukan. Kuvassa 5.2 tällainen polku on a-c-b(-a). Tämän kaaren olemassaolon pystyy toteamaan transitiivisen ab kaaren olemassaololla, mutta selkeyden vuoksi transitiiviset kaaret on jätetty kuvasta pois. Polun löytämiseen on tehokkaita algoritmeja, joita on esitelty kappaleessa 3.3.

Jos verkko on monimutkaisempi, voi olla tilanne, jossa silmukan muodostavia polkuja on useampia. Tällainen tilanne on esitetty kuvassa 5.3. Jos silmukan tunnistamiseen käytetään pelkästään yksinkertaista polunhakualgoritmia, niin voi olla tilanne, jossa ainoastaan polku (a, c, b) tunnistetaan virheelliseksi, vaikka virheellinen kaari voi olla myös kaari ad tai kaari db.

Kun ristiriitaiset polut on löydetty, ne poistetaan verkosta tai heikennetään.



Kuva 5.2: Kaaren (b, a) lisääminen, kun solmujen välillä on olemassa oleva polku, transitiivista kaarta (a, b) ei ole piirretty.



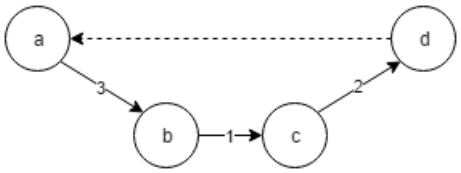
Kuva 5.3: Kaaren (b, a) lisääminen, kun solmujen välillä on useita olemassa olevia polkuja, transitiivisia kaaria ei ole piirretty.

5.2.1.2 Polun heikentäminen

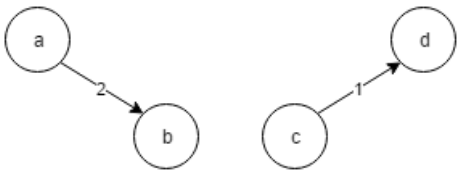
Koska haluamme algoritmin olevan mahdollisimman kestävä käyttäjän virheille, emme suoraan halua poistaa ristiriitaisen silmukan muodostavia kaaria. Tämän vuoksi lisäämme verkkoon painotukset: jokaisen kaaren painoarvo kertoo, miten monta kertaa käyttäjä on syötteellään vahvistanut kaaren olemassa olon. Jos käyttäjä tekee virheen ja luo ristiriidan, kaaria ei suoraan poisteta, vaan niiden painoarvoja pudotetaan yhdellä ja vasta kun kaaren paino putoaa nollaan, se poistetaan verkosta. Näin sellaiset kaaret, joista käyttäjä on varma, pysyvät verkossa, mutta virheelliset kaaret karsiutuvat pois.

Kuvassa 5.4 lisätään olemassa olevaan verkkoon kaari (d, a) ja se muodostaisi silmukan. Kaaren lisäämisen sijaan jokaista ristiriitaista kaarta pienennetään yhdellä, jolloin kaaren (b, c) painoarvo putoaa nollaan ja se poistetaan verkosta. Heikennyksen jälkeinen verkko on kuvattu kuvassa 5.5

Koska saattaa olla tilanne, jossa lisättävän kaaren kanssa ristiriidassa olevalla polulla on pelkästään kaaria, joiden painoarvo on suurempi kuin 1, itse polkua ei poisteta verkosta. Siispä tätä metodia käytettäessä ei voida lisätä uutta kaarta verkkoon, koska se voisi edelleen muodostaa silmukan jäljelle jääneiden heikennettyjen kaarien kanssa.



Kuva 5.4: Kaaren (d, a) lisääminen, kun solmujen välillä on olemassa oleva polku, transitiivisia kaaria ei ole piirretty.



Kuva 5.5: Verkko ristiriitaisen polun heikentämisen jälkeen

5.2.2 Keskeytynyt suoritus

Käyttäjä ei välttämättä halua listalleen täydellistä järjestystä, joten algoritmin on annettava hyödyllisiä tuloksia kesken suorituksensa. Paras tapa tämän ratkaisemiseen, on laskea käyttäjän syötteestä muodostuvasta verkosta topologista järjestys, jonka lopputulos palautetaan algoritmin tuloksena, kun käyttäjä päättää lopettaa algoritmin suorittamisen. Mutta koska haluamme algoritmilta myös välitöntä palautetta, niin on hyödyllistä käyttää jatkuvaa topologista järjestämistä algoritmin edetessä, jotta käyttäjälle voidaan näyttää syötteen vaikutukset järjestettyyn listaan.

5.3 Yhteenveto

Kävimme läpi algoritmissa käytetyt erikoistapaukset, joissa käyttäjän antama syöte aiheuttaa ristiriidan verkon nykyisessä rakenteessa. Lisäsimme verkkoon painotuksen, jotta verkon kaaria olisi mahdollista vahvistaa ristiriitoja silmällä pitäen ja määrittelimme, miten tarvittaessa kaaren painoarvoa pystyy heikentämään. Tämän jälkeen tunnistimme ristiriidat, joita syntyy, kun käyttäjän syöte ja verkkoon tallennettu tieto eivät kohtaa.

Luku 6

Algoritmin toteutus

Tässä luvussa käymme algoritmin toteutuksen vaihe vaiheelta läpi ja tarkastelemme teknisiä yksityiskohtia tämän työn piirissä olevassa teknisessä toteutuksessa.

Algoritmi käyttää suuruusjärjestyksen esittämiseen verkkoa, jonka solmuiksi lisätään järjestettävän joukon alkiot. Alustetussa verkossa ei ole kaaria solmujen välillä, mutta alustuksen jälkeen algoritmi valitsee verkon solmuista sopivan kokoisen osajoukon, joka esitetään käyttäjälle. Käyttäjä valitsee yhden, järjestyskriteerien mukaan parhaimman alkion, jolloin algoritmi luo osajoukon solmujen välille suunnattuja kaaria osoittamaan suuruusjärjestystä verkossa.

Jokaisen luodun kaaren painoarvo on lähtökohtaisesti 1. Painoarvoa käytetään kuvaamaan luottamusta kaaren todenperäisyyteen siltä varalta, että käyttäjä tekee arvotuksessaan virheitä. Jos käyttäjä antaa jo verkossa olevan kaaren toistamiseen, uutta kaarta ei lisätä, mutta olemassa olevan kaaren painoarvoa kasvatetaan yhdellä. Näin verkon hyvin tunnetut kaaret kestävät jatkossa käyttäjän virheitä.

Uusien kaarien lisäämisen jälkeen algoritmi laskee verkosta transitiivisen sulkeuman ja päättelee sen perusteella, onko verkko riittävän tiheä suorituksen päättämiseen. Jos algoritmia on tarve jatkaa, algoritmi laskee transitiivisesta sulkeumasta seuraavan solmujen osajoukon, joiden väliset kaaret kasvattavat verkon tiheyttä eniten.

6.1 Algoritmin yleiskuvaus

Algoritmi 6.1: Algoritmi manuaaliseen järjestämiseen

```
1  while (graphHasMissingEdges()) {
2    List subset = getNodesToValue();
3    Node best = askUserForBestNode(subset);
4    for (Node node : subset) {
5      if (!node.equals(best));
6        addEdge(node, best);
7    }
8    updateTransitivity(subset, best);
9    List sortedNodes = doTarjanSort();
10 }
```

6.2 Lopetusehto

```
while (graphHasMissingEdges()) { ... }
```

Algoritmia tulee suorittaa niin kauan kunnes ollaan saavutettu turnaus (kappale /refsection:turnaus). Tällöin jokaisesta verkon solmusta on suunnattu kaari jokaiseen muuhun solmuun ja verkon solmuilla on yksikäsitteinen suuruusjärjestys.

Helpoiten määrittely onnistuu käyttämällä verkon tiheyttä, joka on 1 silloin kun jokaisella verkon solmuparilla on yksi kaari välillään. Verkon tiheys lasketaan kaavalla:

$$D = \frac{|E|}{|V|(|V|-1)} \tag{6.1}$$

Jos käyttäjälle riittää osittainen varmuus järjestyksessä, voidaan verkon tiheyttä käyttää raja-arvona algoritmin päättämiselle. Pieni tiheysraja päättää algoritmin nopeammin, mutta algoritmin lopputulos on todennäköisemmin virheellinen, kun verkkoon ei ole lisätty riittävästi kaaria. Analyysissa (luku 7) käydään tarkemmin läpi tiheysrajan vaikutusta algoritmin lopputulokseen.

Päätimme käyttää verkon transitiivista sulkeumaan tiheyden laskemisen pohjana, koska silloin käyttäjän ei tarvitse antaa jokaisen solmuparin välille syötettä. Tällöin on mahdollista, että algoritmi päättelee transitiivisen sulkeuman käyttäjän virheellisen syötteen perusteella, mutta tällä saavutetaan hyötyjä, kun käyttäjän ei tarvitse manuaalisesti antaa syötettä algoritmin loppuvaiheessa jokaiselle verkon $O(n^2)$ solmuparille.

6.3 Käyttäjälle esitettävän alijoukon valinta

```
List subset = getNodesToValue();
```

Valitaan verkon transitiivisen sulkeuman perusteella ne solmut, joiden kysyminen edistää verkon täydentymistä mahdollisimman paljon. Alijoukon valinta vaikuttaa siihen miten paljon hyötyä transitiivisen sulkeuman laskennalla saavutetaan algoritmin seuraavassa vaiheessa.

Alijoukon koko määrittää sen kuinka helppo käyttäjän on tehdä päätös parhaasta alkiosta. Jos joukko on liian suuri, käyttäjä ei osaa intuitiivisesti valita parasta alkiota. Toisaalta joukko ollessa liian pieni, käyttäjä joutuu tekemään tarpeettoman monta valintaa järjestämisen aikana.

Jos haluamme, että algoritmi on mahdollisimman tehokas täydentämään verkkoa, haluamme valita syötettä varten sellaisia solmuja, jotka ovat mahdollisimman etäällä toisistaan. Tällaista solmujen osajoukkoa kutsutaan itsenäisen joukoksi (kappale 3.7), mutta itsenäisen joukon laskeminen verkolle osoittautui NP-täydelliseksi ongelmaksi.

Seuraavaksi kokeilimme toteutusta, jossa käytetään Kargerin algoritmia (kappale 3.7) jakamaan verkko kahdella minimaalisella leikkauksella kolmeen osaverkkoon ja niiden solmuista valitaan se, jolla on pienin aste. Näin saadaan kolme solmua, jotka ovat eri osissa verkkoa. Tämä johti tehokkaaseen algoritmiin, mutta vielä parempiin tuloksiin päästiin yksinkertaisella solmujen järjestämisellä.

Pseudokoodi päättelystä on algoritmissa 6.2. Algoritmi laskee solmuille vertailuarvon (v), joka on solmusta lähtevien ja siihen saapuvien ei-transitiivisten kaarien (e) painojen (w) summa lisättynä solmun astelukuun (d) neliöitynä (kaava 6.2). Tämä vertailuarvo kuvaa sitä miten tiiviisti solmu on yhteydessä muihin verkon solmuihin ja toissijaisesti sitä, miten varma algoritmi on solmusta lähtevistä kaarista. Näiden vertailuarvojen avulla solmut järjestetään listaan rivillä 5.

$$v = d^2 + \sum_{i=0}^{|e|} w(e_i) \tag{6.2}$$

Järjestettyä listaa käydään läpi pienimmästä suurimpaan ja tuloslistaan valitaan sellaiset solmut, joilla ei ole olemassa kaarta muihin tuloslistassa oleviin solmuihin.

Jos algoritmi ei löydä kolmea solmua tuloslistaan, otetaan yksi sattumanvarainen muita pienempi solmu, jolla tarvittaessa vahvistetaan olemassa olevia kaaria. Solmu valitaan verkon transitiivisesta sulkeumasta siten, että sillä on oltava transitiivinen kaari johonkin muuhun tuloslistan solmuun. Valitsemalla muita solmuja pienemmäksi arvioidun solmun varmistamme, ettei algoritmi jää lukkiumaan: oletetaan tilanne, jossa kolmen solmun verkosta puuttuu pelkästään kahden pienimmän alkion välinen kaari ja algoritmi lisää alijoukkoon mukaan aina kolmannen muita suuremman solmun. Tällöin käyttäjä valitsee aina suurimman solmun, eikä algoritmi koskaan saa syötteenään kahden pienimmän solmun välistä suuruusjärjestystä.

Olisi mahdollista käyttää vertailuarvona pelkästään transitiivisen sulkeuman solmun astetta, jolloin luontaisesti algoritmi täydentäisi ne kaaret, joita tarvitaan mahdollisimman tehokkaaseen transitiivisen sulkeuman päivittämiseen. Tämä voisi johtaa tehokkaampaan algoritmiin syöteiteraatioiden kannalta, mutta koska transitiivisen sulkeuman tiheyttä käytetään jo algoritmin päättymisehtona kohdassa 6.2, transitiiviseen sulkeumaan keskittyminen aiheuttaisi liian nopean algoritmin päättämisen. Algoritmi täyttäisi transitiivisen sulkeuman ja jos käyttäjä tekee virheen, algoritmi ei antaisi mahdollisuutta korjata sitä, vaan luottaisi virheestä transitiivisesti pääteltäviin kaariin.

Algoritmi 6.2: Vertailuarvojen laskeminen solmuille ja kysyttävän osajoukon päättely

```
1  getNodesToValue() {
2    result = [];
3
4    // Järjestetään solmut vertailuarvojen mukaan
5    nodes.sort(calculateComparisonValues());
6
7    for (Node node : nodes) {
8      if (!hasEdgeToOtherNodeIn(result)) {
9        result.add(node);
10       if (result.size() == askedNodesLimit)
11         return result;
12     }
13   }
14
15   // Lisää sattumanvarainen pienempi solmu
16   if(result.length < 2){
17     result.add(getRandomSmallNode())
18   }
19   return result;
20 }
```

6.4 Käyttäjän syöte

```
Node best = askUserForBestNode(subset);
```

Käyttäjältä kysytään, mikä osajoukon solmu on suuruusjärjestyksessä korkeimmalla.

Algoritmi ei ota kantaa siihen, miten syöte käyttäjältä saadaan, mutta olettaa käyttäjän valitsevan yhden tarjotuista solmuista parhaimmaksi. Tämän työn piirissä toteutimme algoritmille yksinkertaisen käyttöliittymän, joka tarjoaa käyttäjälle kolme painonappia, joista käyttävä valitsee mieluisimman vaihtoehdon.

Algoritmin käytettävyyttä ja jatkokehitystä on käsitelty tarkemmin luvussa 8.

6.5 Uusien kaarien lisääminen verkkoon

```
for (Node node : subset) {
  if (!node.equals(best))
    addEdge(node, best);
}
```

Algoritmissa 6.3 käydään tarkemmin läpi miten käyttäjän syöte mallinnetaan verkkoon. Jokaisesta osajoukon solmusta luodaan kaari voittajasolmuun. Jos voittajasolmun ja osajoukon solmun välillä on jo olemassa oleva kaari (rivi 2), sitä joko vahvistetaan kasvattamalla sen painoa yhdellä, tai päinvastaisessa tilanteessa sen painoa vähennetään yhdellä ja jos painoa putoaa nollaa, kaari poistetaan verkosta (rivi 3).

Algoritmi 6.3: Kaaren lisääminen verkkoon ja ristiriitaisten kaarien käsittely

```
1  addEdge(Node from, Node to) {
2    if (hasEdgeBetween(from, to)) {
3      reduceOrReinforceExistingEdge();
4    } else {
5      if (transitive.hasNoEdgeBetween(from, to)) {
6        createNewEdgeToGraph();
7      } else {
8        List conflictingEdges = DFS(to, from);
9        for (Edge edge : conflictingEdges) {
10         reduceEdgeOrDeleteIt();
11       }
12     }
13   }
14 }
```

Jos kaarta ei ole olemassa, tarkistetaan, onko verkossa olemassa olevia polkuja kohdesolmusta lähtösolmuun, jotka aiheuttaisivat verkkoon silmukan uuden kaaren lisäyksessä (rivi 5). Algoritmi tarkistaa tämän tarkastelemalla verkosta ylläpidettyä transitiivista sulkeumaa. Jos transitiivisessa sulkeumassa on olemassa kaari kohdesolmusta lähtösolmuun, on varsinaisessa verkossa polku näiden solmujen välillä ja uuden kaaren lisääminen aiheuttaisi silmukan.

Jos transitiivinen kaari löytyy, verkossa on jokin polku, jonka jokin kaari on ristiriidassa nyt lisättävän kaaren kanssa. Ei voida olla varmoja missä vaiheessa verkon täyttöä käyttäjä on tehnyt virheen, joten kaikkia mahdollisesti ristiriitaisia kaaria tulee heikentää.

Algoritmi käyttää kaikkien polkujen välisten kaarien hakemiseen syvyyshakua, joka on kuvattu kappaleessa 3.3. Syvyyshakua on muokattu hieman jolloin haku käy rekursiivisesti läpi kaikki kohdesolmusta lähtevät kaaret ja palauttaa listan niistä kaarista, jotka sijaitseva jollain polulla kohde ja lähtösolmun välillä. Algoritmi käy läpi kaikki löydetyt kaaret ja pudottaa niiden painoarvoa yhdellä. Jos kaaren painoarvo putoaa nollaan, kaari poistetaan verkosta.

Jos lähtö ja kohdesolmun välistä kaarta ei ole olemassa eikä uuden kaaren lisääminen aiheuta verkkoon silmukkaa, voidaan verkkoon luoda uusi kaari ja asettaa sen painoarvoksi 1 (rivi 6).

6.6 Transitiivisen sulkeuman päivittäminen

```
updateTransitivity(subset, best);
```

Transitiivisen sulkeuman päivittäminen jo valmiiksi transitiivisesti sulkeutuneeseen verkkoon on yksinkertainen operaatio, jos lisättävä kaari ei ole ristiriidassa aiemmin tunnettujen transitiivisten suhteiden kanssa. Tämä kaaren lisääminen on selostettu tarkemmin kappaleessa 4.3.

Lisäysoperaatiossa algoritmi listaa kaikki kaaren lähtösolmun isäsolmut lähtösolmulistaksi, ja lisätyn kaaren kohdesolmun lapsisolmut kohdesolmulistaksi. Tämän jälkeen algoritmi lisää uudet transitiiviseksi merkityt kaaret jokaisesta lähtösolmusta jokaiseen kohdesolmuun.

Jos kaaren lisäys on ristiriidassa aiemman tiedon kanssa, transitiivisten kaarien verkosta löytyy olemassa oleva kaari voittajasta johonkin alijoukon solmuun. Käyttäjä on jossain vaiheessa tehnyt virheen tai muodostanut silmukan syötteellään.

Yksinkertaisin tapa selvitä tällaisesta tilanteesta laskea verkon transitiivinen sulkeuma uudestaan. Transitiivisuuden laskevat algoritmit on kuvattu kappaleessa 3.6.

Transitiivisen sulkeuman uudelleen lasku tehdään yksinkertaisella syvyyshakualgoritmilla, laskee jokaisesta solmusta saavutettavissa olevat solmut ja lisää näiden väliltä puuttuvat kaaret transitiivisina.

6.7 Topologinen järjestäminen ja tulosten esittäminen

```
List sortedNodes = doTarjanSort();
```

Algoritmi käyttää kappaleessa 3.5 esitettyä Tarjanin algoritmia verkon topologiseen järjestämiseen ja ottaa huomioon kappaleessa 4.2 mainitut topologisen järjestyksen päivittämiseen liittyvät helpotukset: topologinen järjestys tarvitsee päivittää vain kun kaari lisätään ja kaaren lisäys on ristiriidassa olemassa olevan suuruusjärjestyksen kanssa.

Luku 7

Analyysi

Analyysia varten algoritmille annettiin tehtäväksi järjestää lista numeroita yhdestä kymmeneen. Käyttäjän syötettä simuloimaan kehitimme algoritmille koneellisen käyttäjän, joka lähtökohtaisesti valitsi aina suurimman alkion osajoukosta parhaimmaksi ja antoi tämän syötteenä algoritmille. Kaikissa testeissä on käytetty osajoukkoa, jonka koko on kolme. Tulokset voisivat olla merkittävästi erilaisia suuremmalla osajoukolla.

Jotta erehtyväistä käyttäjää voitiin simuloida, konekäyttäjän syötteeseen lisättiin virhettä siten, että tietty prosenttiosuus vastauksista olisi sattumanvaraisia. Näin pystyttiin dokumentoimaan algoritmin virheensietokyky. Samalla voitiin empiirisesti testata jäisikö algoritmi lukkiumaan ristiriitaisten syötteiden käsittelyssä.

Virheellisen syötteen lisäksi kokeilimme muuttaa algoritmin tiheysrajaa, joka vaikuttaa siihen miten nopeasti algoritmi lopettaa suorituksen ja palauttaa epävarman suuruusjärjestyksen lopputuloksena. Tämä arvo vaikuttaa siihen miten nopeasti algoritmi pysähtyy, mutta samalla se aiheuttaa virhettä ja heikentää algoritmin virheensietoa.

Algoritmin aiheuttamaa virhettä mallinnettiin laskemalla Pearsonin korrelaatio tunnetun järjestyksessä olevan listan ja algoritmin lopputuloksen kanssa. Pearsonin korrelaatio kuvaa sitä miten hyvin kahden satunnaismuuttujan välillä vallitsee lineaarinen riippuvuus [37]. Korrelaatio vaihtelee välillä $[-1, 1]$ ja arvolla 1 listat korreloivat täysin keskenään. Meidän tapauksessamme, koska listoissa on samat alkiot, korrelointi tarkoittaa listojen olevan samassa järjestyksessä. Vastaavasti Pearsonin korrelaation arvo -1 tarkoittaa sitä, että listat ovat täsmälleen päinvastaisessa järjestyksessä. Arvo 0 tarkoittaa, että listojen alkioiden arvot eivät riipu toisistaan lainkaan.

Laskimme kymmenen alkion listan kaikkien permutaatioiden Pearsonin korrelaatiot, jotta saisimme hyvän kuvan siitä minkä tyyppistä virhettä mikäkin korrelaatio kuvaa. Jos kymmenen alkion listassa vierekkäiset alkiot

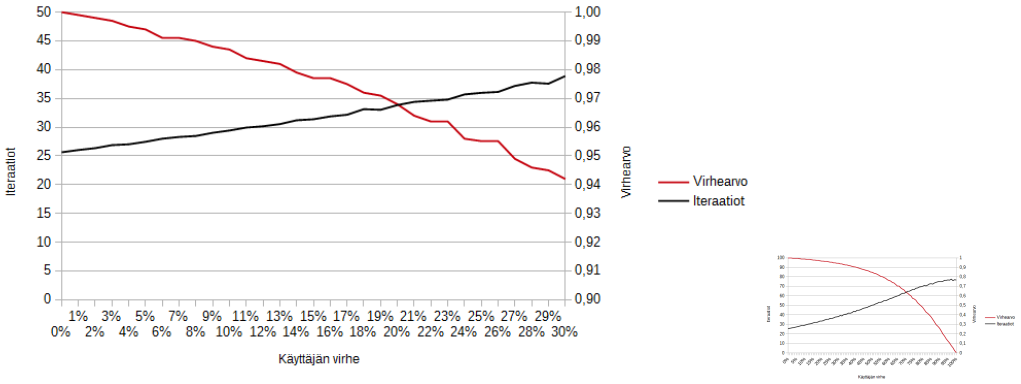
Taulukko 7.1: Pearsonin korrelaation arvot ja vastaavat esimerkkilistat

Korrelaatio	Esimerkkilista
1,000	$(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$
0,988	$(2, 1, 3, 4, 5, 6, 7, 8, 9, 10)$
0,976	$(2, 1, 4, 3, 5, 6, 7, 8, 9, 10)$
0,964	$(2, 1, 4, 3, 6, 5, 7, 8, 9, 10)$
0,952	$(2, 1, 4, 3, 6, 5, 8, 7, 9, 10)$
0,939	$(2, 1, 4, 3, 6, 5, 8, 7, 10, 9)$

vaihtavat paikkaa, korrelaatioksi tulee 0,988 ja tämä on pienin mahdollinen virhe kymmenen alkion listassa. Muut merkittävät virhearvot esimerkkeineen on listattu taulukossa 7.1. Taulukosta on korostettu punaisella värillä ne alkioparit, jotka ovat vaihtaneet paikkaa keskenään. Merkittävä korrelaatio on erityisesti 0,939, jolla listassa välttämättä yksikään alkio ei ole oikealla suuruusjärjestyksen määräämällä paikallaan, mutta lista on silti pääosin oikeansuuntaisessa järjestyksessä.

Tässä luvussa esitetyissä kuvaajissa on selkeyden vuoksi rajattu käyttäjän virhe alle 30 %:iin, tiheysraja alle 75 % ja virhearvo yli 0,9, mutta kuvaajien yhteydessä on esitetty pienellä täysimittaiset kuvaajat jotta kokonaiskuva tunnuslukujen kehityksestä tulee selväksi.

7.1 Virhealttius



Kuva 7.1: Listan järjestäminen epävarmalla syötteellä. Listan pituus 10, tiheysraja 100 %.

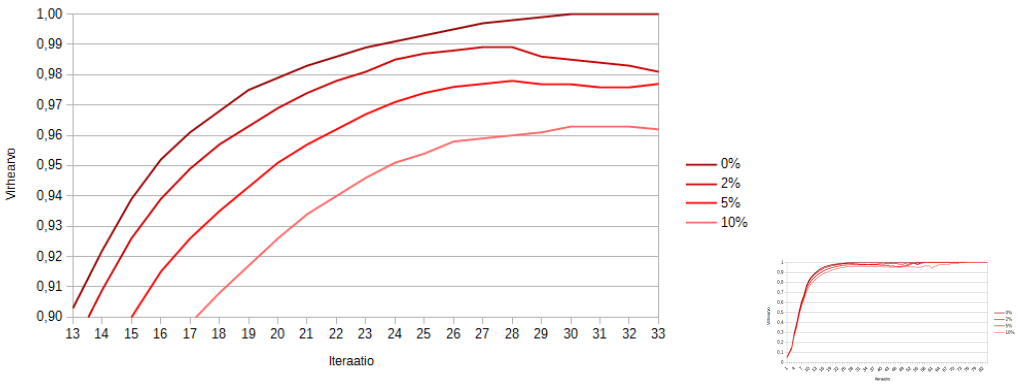
Ensimmäisessä testissä kokeiltiin algoritmin virhesietoisuutta kun käyttäjä tekee sattumanvaraisia virheitä kymmenen alkion syötteen järjestämisessä. Testasimme konekäyttäjän virheprosentit välillä 0 % - 100 % ja kirjasimme ylös algoritmin vaatimien suoritusiteraatioiden määrän ja algoritmin lopputuloksen virheen. Testin tulokset näkyvät liitteessä 7.1.

Kuten kuvasta 7.1 näkyy, algoritmin vaatimat suoritusiteraatiot virheettömällä käyttäjällä algoritmi vaatii 26 syöteiteraatiota kunnes suoritus pysähtyy. Tällöin lopputuloksessa ei ole lainkaan virhettä, joten virheluku on 1; lopputulos korreloi täysin oletetun kanssa.

Vielä 30 % käyttäjävirheellä algoritmin lopputuloksen korrelaatio on yli 0,94. Luonnollisesti tämänkin tuloksen saamiseksi algoritmi joutuu käymään läpi keskimäärin lähes 40 iteraatiota, jotta merkittävältä osin sattumanvaraisesta syötteestä selviäisi oikea lopputulos. Käyttäjän voi siis tehdä merkittävästikin virheitä lopputuloksen siitä kärsimättä, mutta algoritmi joutuu kompensoimaan epävarmaa käyttäjää kasvattamalla iteraatiomääriä.

7.2 Virheen kehitys algoritmin edetessä

Koska algoritmin yksi keskeisiä suunnitteluperiaatteita oli välitön käyttäjäpalaute suorituksen edetessä, olemme kiinnostuneita siitä, miten algoritmin tuloslistan virhe kehittyy algoritmin suorituksen aikana. Seuraavaksi tarkastelemme, miten algoritmin välitulosten virhe muuttuu algoritmia suoritettaessa ei käyttäjän virhetasoilla.



Kuva 7.2: Järjestelyvirheen kehitys kun algoritmi etenee eri käyttäjän virhetasoilla. Listan pituus 10, tiheysraja 100 %.

Käytimme neljää konekäyttäjää, joka antoivat algoritmille virheellisen syötteen todennäköisyyksillä 0%, 2%, 5% ja 10%. Tavoitteena oli nähdä, miten virheiden määrä heijastuu algoritmin välituloksiin.

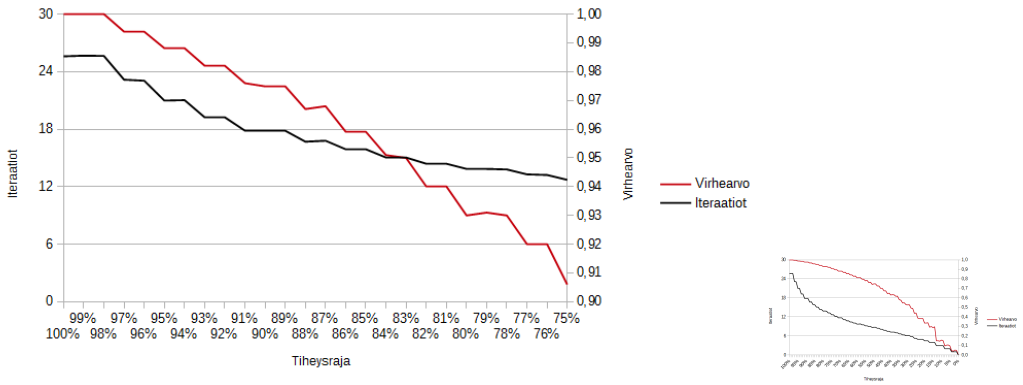
Tulokset on kuvattu kuvassa 7.2. Kuvaajasta näkyy, että virhearvo nousee jyrkästi, mutta algoritmin pysähtymiseen vaaditaan huomattavasti enemmän iteraatioita. Eli vaikka algoritmilla kuluu yli 25 iteraatiota lopputuloksen saavuttamiseen virheettömällä käyttäjällä, niin algoritmin välituloksen virhearvo ylittää 0,9:n keskimäärin jo 13. iteraatiolla. Jopa 10 % virhettä antava konekäyttäjä pääsee keskimäärin samaan tarkkuuteen 20. iteraatiolla.

Osatulokset ovat siis jo kohtuullisen tarkkoja ja algoritmilla kuluu merkittävä osa iteraatioista aiempien syötteiden tarkistamiseen ja transitiivisten suhteiden vahvistamiseen. Tätä aikaa pystytään rajoittamaan säätämällä algoritmin loppuehtoa, tiheysvaatimusta. Seuraavassa testissä kokeilimme miten tiheysvaatimuksen muuttaminen muuttaa algoritmin suorituskykyä.

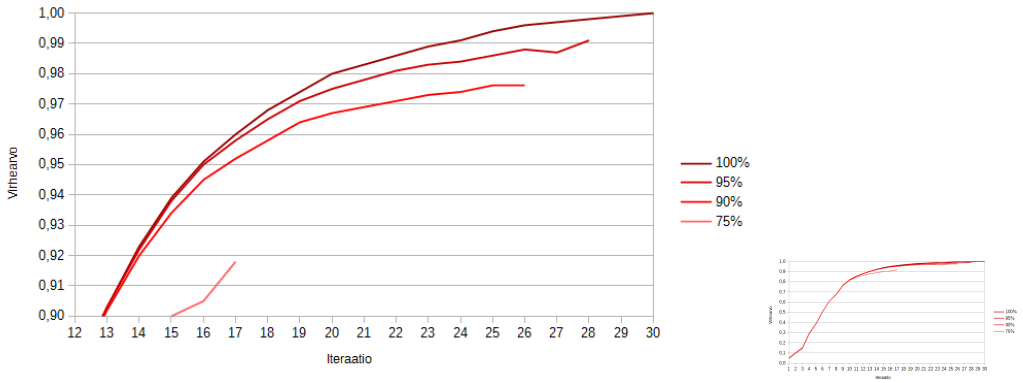
7.3 Tiheysrajan muuttaminen

Tiheysrajan säätäminen mahdollistaa algoritmin valmistumisen nopeammin, mutta samalla se tuo lopputulokseen virhettä. Vaikka käyttäjä ei tekisi lainkaan virheitä, vajaaksi jäänyt suoritus ei takaa lopputulokselle oikeaa järjestystä. Tätä virhettä on pyritty havainnollistamaan kuvissa 7.3 ja 7.4.

Kuvassa 7.3 on havainnollistettuna miten algoritmin lopputulos ja iteraatioiden määrä muuttuu kun tiheysrajaa pudotetaan sadasta prosentista nollaan. 100 % tarkoittaa samaa tilannetta kuin edellisessä testissä ja vastaa kuvan 7.1 tilannetta käyttäjän virheellä 0 %.



Kuva 7.3: Tiheysrajan vaikutus algoritmin virheeseen ja iteraatioiden määrään. Listan pituus 10, käyttäjän virhe 0 %.



Kuva 7.4: Virheen kehitys algoritmin edetessä eri tiheysrajoilla. Listan pituus 10, käyttäjän virhe 0 %.

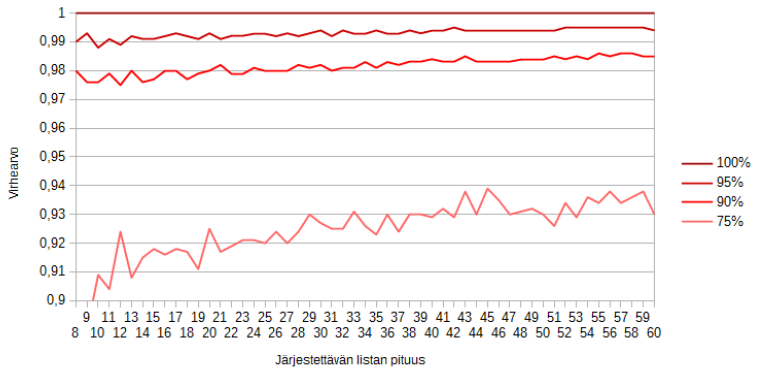
Näemme, että maltillinen parin prosentin lasku tiheysvaatimuksessa ei juuri laske algoritmin tuottamaan virhettä, mutta pudottaa merkittävästi tarvittavien iteraatioiden määrää. Tämä johtuu luonnollisesti siitä, että virheetön konekäyttäjä antaa aina oikean syötteen ja algoritmin päättelemät transitiiviset kaaret ovat aina oikeita.

Kuvassa 7.4 on selvitetty miten tiheysraja vaikuttaa algoritmin välitulosten virheeseen. Valitsimme kuvasta 7.3 sellaiset tiheysvaatimustasot joilla muutos vaikuttaa merkittävältä: 100%, 95%, 90% ja 75%.

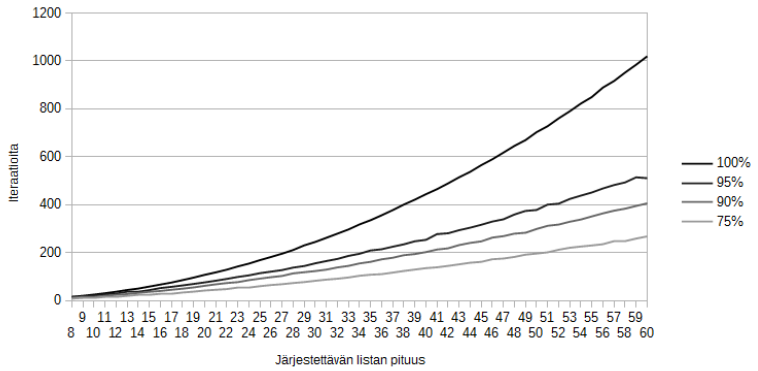
Koska tiheysvaatimusta käytetään nimenomaan algoritmin lopettamiseen, se ei luonnollisesti vaikuta suorituksen aikaisiin tuloksiin. Sen sijaan kuvasta näkyy, miten 75 % tiheysvaatimuksen käyrä erkanee muista merkittävästi

aikaisemmin: 11. iteraation kohdalla. Tämä on linjassa kuvan 7.3 kanssa, mistä näemme kuinka 75 %:n tiheysvaatimuksella suoritus keskimäärin pysähtyy 13. iteraatiolla. Iteraatiot tästä eteenpäin eivät enää muuta algoritmin tulosta ja viimeistään iteraatio numero 17 on viimeinen kyseisellä tiheysvaatimuksella.

7.4 Algoritmin skaalautuminen pidemmille listoille



Kuva 7.5: Eri pituisten listojen järjestämisen virhe eri tiheysvaatimuksilla. Käyttäjän virhe 0 %.



Kuva 7.6: Eri pituisten listojen vaatima iteraatiomäärä eri tiheysvaatimuksilla. Käyttäjän virhe 0 %.

Tähän mennessä olemme testanneet vain 10 alkion mittaisella listalla. Seuraavaksi testaamme, miten algoritmi skaalautuu suurempien listojen järjestämiseen. Algoritmi on suunniteltu ennen kaikkea virhesietoiseksi, joten oletimme, ettei algoritmi skaalaudu käytännöllisissä mittasuhteissa.

Kuvassa 7.6 kuvataan algoritmin vaatimien vertailujen määrää vertailtavan listan pituuden suhteen eri tiheysvaatimuksilla. Kuvasta on helppo huomata, että jo 27 alkion listan järjestäminen 100 % tiheysvaatimuksella vie 200 iteraatiota, joka on melko paljon käsin syötettäväksi.

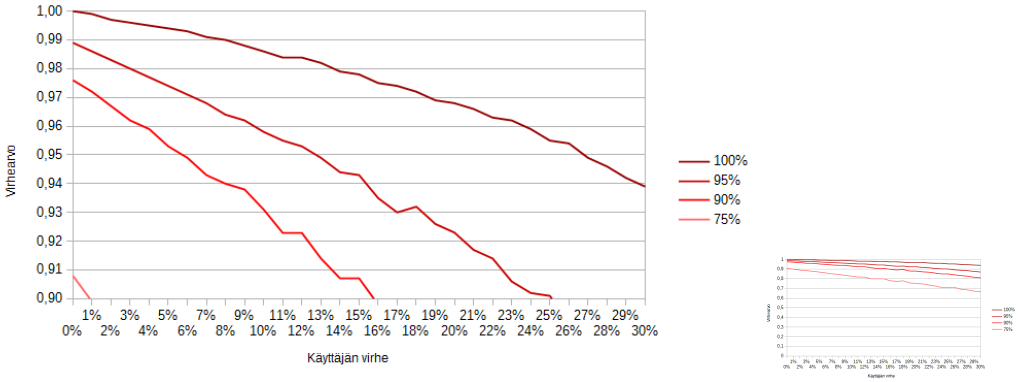
100% tiheysvaatimus vaatii huomattavasti enemmän iteraatioita pysähtyäkseen verrattuna matalampiin tiheysvaatimuksiin erityisesti pidemmillä listoilla. 60 alkion mittaisen listan järjestäminen 95 % tiheysvaatimuksella vaatii vain noin puolet niistä syötteistä, mitä täysi 100 % tiheysvaatimus tarvitsee pysähtyäkseen. Tämä johtuu ilmeisesti siitä, että suurissa verkoissa transitiivisten kaarten päättely on tehokasta, mutta täydessä verkossa on niin paljon kaaria, että kaikkien transitiivisten kaarien varmistamiseen kuluu paljon iteraatioita.

Jos tiheysvaatimuksen pudottamisella on merkittäviä tehohyötyjä, niin miten virhe skaalautuu? Kuvassa 7.5 mittasimme algoritmin lopputuloksen virhearvoa eri tiheysvaatimuksilla ja kasvavalla listan pituudella. Virheluvun kehitysluku vaikuttaa olevan pienenemään päin mitä pidempi lista on kyseessä. Intuitiivisesti tämä sopii, sillä verkon kaarien lukumäärä kasvaa neliöllisesti, jolloin prosenttipohjaisella tiheysvaatimuksella on suurempi määrä kaaria topologisen järjestyksen päättelyyn.

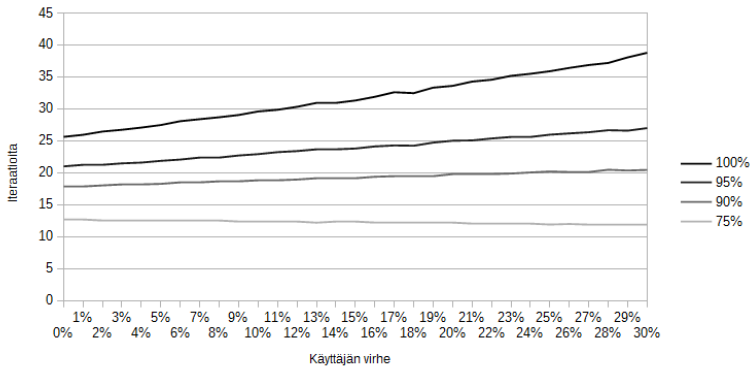
7.5 Tiheysrajan yhdistäminen käyttäjän virheeseen

Koska yllä olemme selvittäneet erikseen sekä käyttäjän virheen (kuva 7.7) että tiheysrajan (kuva 7.8) vaikutuksen algoritmin tulokseen. Viimeisessä testitapauksessa yhdistimme molemmat muuttujat ja arvioimme virheen yhteisvaikutusta.

Kuvassa 7.8 on esitetty algoritmin vaatimien iteraatioiden määrä eri tiheysvaatimuksilla, kun käyttäjä tekee virheitä syötteessä. Iteraatioiden määrä 100 % tiheysvaatimuskäyrällä odotetusti noudattaa kuvan 7.1 iteraatiokäyrää.



Kuva 7.7: Algoritmin virheen määrä eri tiheysrajoilla ja käyttäjän virhetasoilla. Listan pituus 10.



Kuva 7.8: Algoritmin kuluttamien iteraatioiden määrä eri tiheysrajoilla ja virhetasoilla. Listan pituus 10.

7.6 Yhteenveto

Algoritmin virhesietoisuus osoittautui hyväksi. Riippuen tiheysrajan suuruudestaja käyttäjän virheestä, algoritmi sai listan hyvin järjestykseen. Käyttäjä voi antaa syötteestä 5 % virheellisenä, jos algoritmin tiheysvaatimus on asetettu 95 %:iin. Tällöin algoritmi palauttaa keskimäärin tuloksen korrelaatiolla 0,974. Tämä vastaa kymmenalkioista listaa, jossa kaksi vierekkäistä alkioparia on vaihtanut paikkaa keskenään. Tuloksen saamiseksi algoritmi tarvitsee keskimäärin 21,9 syötettä. Tinkimällä tiheysrajasta säästetään keskimäärin 5,6 iteraatiota.

Koska algoritmin virhe kehittyy kuvassa 7.4 kuvatulla tavalla, käyttäjä voi jättää algoritmin suorituksen puolitiehen ja silti saada hyväksyttäviä tuloksia. Käyttäjän järjestäessä abstraktia aineistoa, on tärkeää, että algoritmi antaa välitöntä palautetta käyttäjälle suuruusjärjestyksen kehityksestä ja antaa käyttäjälle tietoa suuruusjärjestyksen varmuudesta. Tällöin käyttäjä voi tehokkaasti päättää kesken suorituksen, että on päässyt riittävään tarkkuuteen listan järjestämisessä ja voi päättää suorituksen ilman kaikkien algoritmin pysähtymiseen vaadittavien iteraatioiden suorittamista.

Käytännön sovellukset algoritmille vaativat tapauskohtaista analyysiä käyttäjien luotettavuudesta ja kärsivällisyydestä, sekä järjestettävän aineiston laadusta. Seuraavassa kappaleessa esitämme jatkokehitysideoita ja teemme yhteenvedon tuloksista.

Luku 8

Yhteenveto

Esitimme algoritmin, joka ottaa käyttäjältä syötteenä listan vapaamuotoista dataa ja kysymällä käyttäjältä syötettä data-alkioiden välisiä suhteista osaa järjestää listan käyttäjän toivomaan järjestykseen. Algoritmi ottaa huomioon käyttäjän tekemät virheet ja sallii käyttäjälle asetuksista ja aineiston koosta riippuen merkittävän virhemahdollisuuden.

Algoritmilla on merkittäviä mahdollisuuksia sellaisilla sähköisen asiakaspalvelun aloilla, joilla on tarpeen saada käyttäjiltä tietoa abstraktien asioiden välisitä suhteista ja käyttäjän kokemuksista. Algoritmia voi myös soveltaa tutkimustiedon keräämiseen testihenkilöiltä varsinkin sellaisen aineiston osalta, jossa aineiston ääripäät on hyvin tunnettu, mutta ääripäiden väliin jäävän aineiston suhteellinen järjestys on epäselvä.

8.1 Jatkokehitys

Algoritmi on jo nyt hyödyllinen, mutta sitä voi luonnollisesti jatkokehittää merkittävästi. Tärkeitä kehitysalueita on algoritmin kehittäminen yleiskäyttöisemmäksi ja käytettävyyden parantaminen.

8.1.1 Koneellinen järjestäminen

Algoritmi tarjoaa nyt hyvän interaktion käyttäjän kanssa, mutta on mahdollista laajentaa algoritmia ottamaan vastaan koneellista syötettä. Kuten kappaleessa 6.4 totesimme, algoritmin ei ota kantaa miten syöte annetaan ja luvussa 7 toteutimme koneellisen testikäyttäjän antamaan syötettä algoritmille. On siis täysin mahdollista jatkokehittää algoritmia siten, että se sopii paremmin koneelliseen järjestämiseen.

Koneellisessa järjestämisessä algoritmin virhesietoisuudella voi olla käyttöä esimerkiksi jatkuvien prosessien tarkkailussa. Jos algoritmia muutetaan siten, että se ei pääty milloinkaan, vaan jatkaa suuruusjärjestyksen päivittämistä myös silloin kun ollaan saavutettu relaatioverkon turnaus. Tällöin algoritmia voidaan käyttää tilastollisten ominaisuuksien havainnointiin ja pitkän aikavälin trendien tarkasteluun.

8.1.2 Algoritmin jatkokehittäminen

Algoritmia voi myös jatkokehittää älykkäämmäksi lopetusehdon suhteen. Jos algoritmiin lisäisi dynaamisen päättelyn, joka tarkkailee koska käyttäjän syötteessä ollaan saavutettu tilanne, että suuruusjärjestys ei muutu, voidaan algoritmin suorittaminen päättää. Nykyinen päättely nojaa täysin transitiivisen verkon tiheyteen, eikä ota huomioon suuruusjärjestyksen stabiiliutta.

Toinen merkittävä kehityskohde on algoritmin transitiivisen sulkeuman päivittämisen tehostaminen. Transitiivisen sulkeuman laskenta matriisikertolaskulla voi olla merkittävästi tehokkaampaa, mutta tässä työssä käytetty kirjasto ei valitettavasti ollut suunniteltu tukemaan matriisioperaatioita.

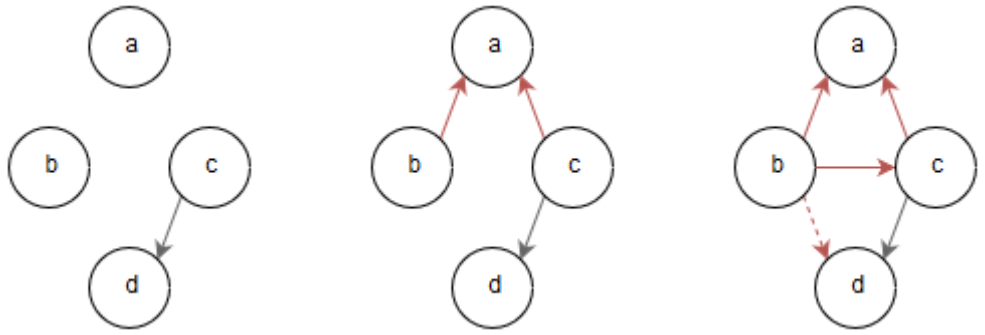
Samasta syystä algoritmin toteutuksessa luovuttiin Flod-Warshall algoritmista. Vierusmatriisin laskeminen käytetyn kirjaston avulla on aikavaativuudeltaa $O(n^2)$ operaatio. Laskeminen oli paljon tehokkaampaa käyttämällä kirjaston omaa syvyyshakuiteraattoria.

Käytettävyysnäkökulmasta algoritmissa voidaan kehittää esimerkiksi mahdollisuus peruuttaa käyttäjän tekemä valinta. Nykyisellään algoritmi nojaa itsekorjaavuuteen, mutta käyttäjän virheiden varalta kannattaisi toteuttaa jonkinlainen peruutustoiminnallisuus. Tätä ei toteutettu tämän työn puitteissa.

Analyysissa ei ole huomioitu kysyttävän alijoukon koon vaikutusta iteraatioiden määrään. Eikä myöskään sitä, että mikä on algoritmin ensimmäinen oikein järjestetty iteraatio.

Marcus et al. [27] on käyttänyt menestyksellä algoritmia, jossa usea käyttäjä antaa koko järjestettävälle tietojoukolle järjestysnumerot ja usean käyttäjän järjestysnumeroilla voidaan laskea koko tietojoukon alkioille kesimääräisen käyttäjän antama järjestys. Tässä voisi soveltaa meidän algoritmimme jatkokehityksessä siten, että algoritmin käyttäjä laittaisi alijoukon alkiot keskinäiseen suuruusjärjestykseen ja näin pysyttäisiin luomaan myös kysyttävän alijoukon sisäisiä kaaria ja näistä muodostuvia transitiivisia kaaria.

Kuvassa 8.1 on havainnollistettu ero verkossa, jossa on neljä solmua: a, b, c ja d ja olemassa oleva kaari (c, d). Jos käyttäjältä kysytään osajoukko (a, b, c) ja käyttäjä valitsee parhaimmaksi solmun a, nykytilanteessa päädytään verkkoon 8.1b. Jos käyttäjä parannetussa algoritmissa valitsisi os-



(a) Alkuperäinen verkko.

(b) Nykyinen toteutus, jossa käyttäjä valitsee vain parhaimman alkion.

(c) Parannettu versio, jossa käyttäjä asettaa osajoukon suuruusjärjestyksen.

Kuva 8.1: Parannus algoritmin kaarien lisäämiseen. Punaisella lisätyt kaaret ja katkoviivalla transitiiviset kaaret.

ajoukon suuruusjärjestykseksi $a > b > c$, niin lisäksi pystyttäisiin luomaan verkkoon kaari (b, c) ja transitiivinen kaari (b, d) (kuva 8.1c).

Lisäksi algoritmia voisi muuttaa käyttämään vielä joustavammin suunnatun verkon mahdollisuutta lisätä useita kaaria solmuparin välille. Algoritmin voisi muuttaa luopumaan ylläpidetyn verkon silmukattomuudesta ja kaarien painoista siten, että jokainen suuruusvertailu mallinnettaisiin verkkoon uutena kaarena. Topologisen järjestämisen yhteydessä käytettäisiin algoritmia, joka rikkoisi syntyneet silmukat. Tällöin käyttäjä voisi antaa myös ristiriitaisia syötteitä ja verkkoa vain kasvatettaisiin myös ristiriitaisten syötteiden perusteella.

8.1.3 Nippulajittelu

Algoritmin hyödyllisyys on kyseenalainen, kun järjestetään hyvin suuria alkiojoukkoja. Kappaleessa 2.2 käsittelimme abstraktia nippulajittelua, jolla voidaan jakaa järjestettävä joukko osajoukkoihin, joilla on keskenään jokin järjestys. Näiden osajoukkojen järjestäminen on niiden pienuuden vuoksi tehokkaampaa, ja osajoukkojen järjestämisen jälkeen ne kootaan yhdeksi tuloslistaksi.

Tätä periaatetta voisi soveltaa jatkokehityksessä, jos algoritmia kehitetään johonkin tiettyyn tarkoitukseen ja tiedetään, että järjestettävillä alkiolla on ominaisuuksia, joilla ne voidaan järjestää nippuihin.

Kirjallisuutta

[1] Alpern, B., Hoover, R., Rosen, B. K., Sweeney, P. F., and Zadeck, F. K. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 1990), SODA '90, Society for Industrial and Applied Mathematics, pp. 32–42.

[2] Amazon. Mechanical turk. `https://www.mturk.com`.

[3] Bar-Noy, A., and Naor, J. S. Sorting, minimal feedback sets, and hamilton paths in tournaments. *SIAM J. Discret. Math. 3*, 1 (Jan. 1990), 7–20.

[4] Belik, F. An efficient deadlock avoidance technique. *IEEE Trans. Comput. 39*, 7 (July 1990), 882–888.

[5] Bender, M. A., Fineman, J. T., and Gilbert, S. A new approach to incremental topological ordering. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2009), SODA '09, Society for Industrial and Applied Mathematics, pp. 1108–1115.

[6] Chan, T. M. More algorithms for all-pairs shortest paths in weighted graphs. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2007), STOC '07, ACM, pp. 590–598.

[7] Chernev, A., Böckenholt, U., and Goodman, J. Choice overload: A conceptual review and meta-analysis. *Journal of Consumer Psychology 25*, 2 (2015), 333 – 358.

[8] Cohen, E., Fiat, A., Kaplan, H., and Roditty, L. A labeling approach to incremental cycle detection. *CoRR abs/1310.8381* (2013).

[9] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[10] Dasari, N. S., Ranjan, D., and Mohammad, Z. Maximal clique enumeration for large graphs on hadoop framework. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications* (New York, NY, USA, 2014), PPAA '14, ACM, pp. 21–30.

[11] Demetrescu, C., Eppstein, D., Galil, Z., and Italiano, G. F. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook*, M. J. Atallah and M. Blanton, Eds. Chapman & Hall/CRC, 2010, ch. Dynamic Graph Algorithms, pp. 9–9.

[12] Demetrescu, C., and Italiano, G. F. Fully dynamic transitive closure: Breaking through the o(n/sup 2/) barrier. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 2000), FOCS '00, IEEE Computer Society, pp. 381–.

[13] Dong, G., Libkin, L., Su, J., and Wong, L. Maintaining transitive closure of graphs in sql. *International Journal of Information Technology 51*, 1 (1999), 46–78.

[14] Eades, P., Lin, X., and Smyth, W. F. A fast effective heuristic for the feedback arc set problem. *Information Processing Letters 47* (1993), 319–323.

[15] Floyd, R. W. Algorithm 97: Shortest path. *Commun. ACM 5*, 6 (June 1962), 345–.

[16] Frigioni, D., Miller, T., Nanni, U., and Zaroliagis, C. An experimental study of dynamic algorithms for transitive closure. *J. Exp. Algorithmics 6* (Dec. 2001).

[17] Haeupler, B., Kavitha, T., Mathew, R., Sen, S., and Tarjan, R. Faster algorithms for incremental topological ordering. In *Automata, Languages and Programming*, L. Aceto, I. Damgård, L. Goldberg, M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, Eds., vol. 5125 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 421–433.

[18] Haeupler, B., Kavitha, T., Mathew, R., Sen, S., and Tarjan, R. E. Incremental cycle detection, topological ordering, and strong component maintenance. *CoRR abs/1105.2397* (2011).

[19] Hick, W. E. On the rate of gain of information. *The Quarterly Journal of Experimental Psychology 4* (1952), 11–26.

[20] Hopcroft, J., and Tarjan, R. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM 16*, 6 (June 1973), 372–378.

[21] Ingerman, P. Z. Algorithm 141: Path matrix. *Commun. ACM 5*, 11 (Nov. 1962), 556–.

[22] Karger, D. R., and Stein, C. A new approach to the minimum cut problem. *J. ACM 43*, 4 (July 1996), 601–640.

[23] King, V. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1999), FOCS '99, IEEE Computer Society, pp. 81–.

[24] King, V., and Sagert, G. A fully dynamic algorithm for maintaining the transitive closure. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1999), STOC '99, ACM, pp. 492–498.

[25] Kong, T. T. A novel net weighting algorithm for timing-driven placement. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design* (New York, NY, USA, 2002), ICCAD '02, ACM, pp. 172–176.

[26] Luce, R. D., and Perry, A. D. A method of matrix analysis of group structure. *Psychometrika 14*, 2 (1949), 95–116.

[27] Marcus, A., Wu, E., Karger, D., Madden, S., and Miller, R. Human-powered sorts and joins. *Proc. VLDB Endow. 5*, 1 (Sept. 2011), 13–24.

[28] Nuutila, E. An efficient transitive closure algorithm for cyclic digraphs. *Inf. Process. Lett. 52*, 4 (Nov. 1994), 207–213.

[29] Pearce, D. J., and Kelly, P. H. J. A dynamic topological sort algorithm for directed acyclic graphs. *J. Exp. Algorithmics 11* (Feb. 2007).

[30] Redelmeier DA, S. E. Medical decision making in situations that offer multiple alternatives. *JAMA*, 273(4) (Jan 25 1995), 302–5.

[31] Roditty, L. A faster and simpler fully dynamic transitive closure. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2003), SODA '03, Society for Industrial and Applied Mathematics, pp. 404–412.

[32] Rodriguez, M. C. Three options are optimal for multiple-choice items: A meta-analysis of 80 years of research. *Educational Measurement: Issues and Practice 24*, 2 (June 2005), 3–13.

[33] Seow, S. C. Information theoretic models of hci: A comparison of the hick-hyman law and fitts' law. *Hum.-Comput. Interact. 20*, 3 (Sept. 2005), 315–352.

[34] Shevtsov, L. A human-driven sort algorithm (monkeysort). Verkkosivu, June 2012. Saatavilla: http://leonid.shevtsov.me/en/a-human-driven-sort-algorithm-monkeysort. Viitattu 21.12.2015.

[35] Tarjan, R. Depth first search and linear graph algorithms. *SIAM Journal on Computing* (1972).

[36] Tarjan, R. E. Edge-disjoint spanning trees, dominators, and depth-first search. Tech. rep., Stanford University, School of Humanities and Sciences, Computer Science Department, Stanford, CA, USA, 1974.

[37] Weisstein, E. W. Correlation coefficient. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/CorrelationCoefficient.html`.

[38] Wikipedia. Karger's algorithm — wikipedia, the free encyclopedia, 2016. [Online; haettu 23.3.2016].

[39] Zwick, U. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM 49*, 3 (May 2002), 289–317.