Henri Kivelä

REITINHAKUALGORITMIT

Opinnäytetyö
CENTRIA-AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma
Toukokuu 2014

Centria AMMATTIKORKEAKOULU

TIIVISTELMÄ OPINNÄYTETYÖSTÄ

Yksikkö	**Aika**	**Tekijä/tekijät**
Kokkola-Pietarsaari	Toukokuu 2014	Henri Kivelä
Koulutusohjelma		
Tietotekniikan koulutusohjelma		
Työn nimi		
REITINHAKUALGORITMIT		
Työn ohjaaja		**Sivumäärä**
Kauko Kolehmainen		39
Työelämäohjaaja		

Tämän opinnäytetyön tarkoituksena oli tutkia perus- ja reititysalgoritmeja, reititysalgoritmeista erityisesti Floyd- ja Dijkstra-algoritmeja. Tekstissä pyritään selittämään mahdollisimman tarkasti, mutta yksinkertaisesti kyseiset algoritmit. Työssä sisältää myös joitain pieniä esimerkkejä Floyd- ja Dijkstra-algoritmeja sisältävistä ohjelmista.

Opinnäytetyössä tutustutaan tarkemmin myös siihen, mitä algoritmit ovat ja missä niitä voitaisiin soveltaa. Tarkoituksena on, että tulevaisuudessa voidaan käyttää opinnäytetyötä opetusmateriaalina. Työn pohjalta pyritään tekemään yksinkertainen ja havainollistava opetusmateriaali nuorille opiskelijoille.

Asiasanat
algoritmit, Dijkstra, Floyd, reitinhaku

Centria UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Unit Kokkola-Pietarsaari	**Date** May 2014	**Author/s** Henri Kivelä
Degree programme Information Technology		
Name of thesis ROUTING ALGORITHMS		
Instructor Kauko Kolehmainen		**Pages** 39
Supervisor		

The Purpose of this thesis is to research and learn basic algorithms and routing algorithms. The focus was on routing algorithms, especially on Floyd and Dijkstra algorithms. The thesis aims to explain these algorithms in depth, but still with simple terms.

The thesis also explores algorithms in more detail and where algorithms can be applied. Second aim of this thesis was the possibility to utilize it for teaching. Based on the thesis a simple and illustrative teaching material for young students will be compiled.

Key words
algorithms, Dijkstra, Floyd, routing algorithms

TIIVISTELMÄ
ABSTRACT
SISÄLLYS

1 JOHDANTO

2 MITÄ TARKOITETAAN ALGORITMEILLÄ
  2.1 Algoritmien tehokkuuden kuvaaminen
  2.2 Klassisten ongelmien ratkaisuja algoritmeilla
  2.3 Algoritmien kuvaaminen

3 YLEISIÄ ALGORITMIEN MENETTELYTAPOJA
  3.1 Raaka voima
  3.2 Ositus ja kokoaminen
  3.3 Ahneet algoritmit
  3.4 Esimerkkejä algoritmeista
    3.4.1 Raaka voima
    3.4.2 Osita ja kokoa

4 TIETORAKENTEET
  4.1 ADT
  4.2 Puut
  4.3 Linkitetyt listat
  4.4 Graafit

5 REITIN HAKEMINEN
  5.1 Yleistä reitinhausta
  5.2 Graafien läpikäynti
    5.2.1 Syvyys ensin -haku
    5.2.2 Leveys ensin -haku

6 LYHIMMÄN REITIN HAKEMINEN
  6.1 Floyd-Warshall-algoritmi
  6.2. Dijsktran-algoritmi
  6.3 A*-algoritmi

7 POHDINTA

LÄHTEET
LIITTEET

KUVIOT

KUVIO 1. Algoritmisen ajoajan laajentuminen
KUVIO 2. Esimerkkivuokaavio, joka kuvaa sisäkkäisen ehtolauseen logiikkaa
KUVIO 3. Kuplalajittelun iterointi

KUVIO 4. Mergesortin toiminnasta vuokaavio
KUVIO 5. Esimerkki puuhierarkiasta

KUVIO 6. Esimerkki linkitetystä listasta, jossa on kokonaislukuja
KUVIO 7. Esimerkki suuntaamattomasta verkosta
KUVIO 8. Esimerkki suunnatusta graafista
KUVIO 9. Esimerkki painotetusta graafista
KUVIO 10. Esimerkki syvyys ensin -hausta
KUVIO 11. Esimerkki leveys ensin -hausta
KUVIO 12. Binääripuuesimerkki leveys ensin -hausta
KUVIO 13. Ranskan kaupungin kartta
KUVIO 14. Ranskan kaupungin kartan kuudesta kaupungista tehty vuokaavio
KUVIO 15. Lyhin reitti paikasta Poitiers paikkaan Montlucon
KUVIO 16. Solmukuvio

TAULUKOT

TAULUKKO 1. Etäisyysmatriisi kuudelle valitulle kaupungille
TAULUKKO 2. Reittimatriisi kuudelle valitulle kaupungille
TAULUKKO 3. Etäisyysmatriisi kuuden kaupungin välillä kierroksella yksi
TAULUKKO 4. Reittimatriisi kuuden kaupungin välillä kierroksella yksi
TAULUKKO 5. Etäisyysmatriisi kuuden kaupungin välillä kierroksella kaksi
TAULUKKO 6. Reittimatriisi kuuden kaupungin välillä kierroksella kaksi
TAULUKKO 7. Taulukko, joka on tehty kuvion 16 pohjalta

1 JOHDANTO

Tietojenkäsittelyssä algoritmit ovat keskeinen käsite. Tietokoneiden ohjelmoimista varten tarvitaan algoritmeja. Tutkimalla algoritmeja haetaan kahta päätavoitetta. Ensimmäisenä päätavoitteena on parempien algoritmien kehittäminen uusille laskentaongelmille. Toisena päätavoitteena on pyrkimys ymmärtää algoritmien laskentaongelmien sisäistä vaikeutta ja yleisiä rajoituksia. Tietojenkäsittelytieteessä algoritmitutkimus on keskeisessä asemassa, koska kansainvälinen matemaattisen unionin myöntämä Nevanlinna-palkinto on tähän mennessä aina myönnetty tutkijalle, jonka työ liittyy syvällisellä tavalla algoritmiteoriaan.

Perusalgoritmeja käytetään melkein jokaisessa ohjelmassa. Perusalgoritmit pitää ymmärtää, jotta voidaan luoda ja kehittää tarkempia algoritmeja. Pitää kuitenkin huomioida, että algoritmit ovat eri asia kuin ohjelma. Sama algoritmi voidaan toteuttaa samalla kielellä pienillä eroilla ja se voidaan myös toteuttaa monella eri ohjelmointikielellä. Algoritmi on siis luonteeltaan abstraktio.

Reitinhakualgoritmeja käytetään tietokonepeleissä ja se on niissä yksi keskeisimpiä algoritmeja. Reitinhakualgoritmeja on vielä nykyaikanakin haasteellista tehdä, koska ei ole olemassa mitään yleispätevää hakualgoritmia. Algoritmin haasteellisuus kasvaa, sitä mukaa mitä vaikeammin ymmärrettävä ympäristö on. Reitinhakualgoritmeista tässä työssä tutkitaan Floydia ja Dijkstraa. Reitin hakemiseen algoritmeilla tarvitaan myös graafia. Graafi on matematiikkaan ja tietojenkäsittelytieteeseen liittyvä käsite.

Graafialgoritmit ovat hyvin yleisiä reitinhaussa ja tietokonepeleissä. Reitinhaussa graafialgoritmit ovat ehkä kaikista tärkeimpiä. Työssä käydään aluksi läpi yksinkertaisempia algoritmeja ja miten ne toimivat. Myöhemmin työssä käydään läpi graafeja ja luvussa 5 tutustaan tarkemmin reitinhakualgoritmeihin. Työn lopussa käydään läpi lyhimmän reitinhakua, johon sisältyy Dijkstra ja Floyd-Warshall. Työssä on esimerkit Dijkstran-agloritmista ja Floyd-Warshall-algoritmista.

2 MITÄ TARKOITETAAN ALGORITMEILLÄ

Tietotekniikassa algoritmilla tarkoitetaan menettelyä, jolla saadaan suoritettua erityisiä tehtäviä (Skiena 2008, 1). Algoritmiin sisällytetään muutamia arvoja tai joukko arvoja. Nämä arvot saadaan sisääntulosta ja ulostulosta (Cormen, Leiserson, Rivest & Stein 2009, 5).

Algoritmeja voidaan myös katsoa siltä kannalta, että algoritmi on työkalu, jolla ratkaistaan tarkasti määriteltyjä laskennallisia ongelmia. Ongelma yleensä täsmentyy haluttuun sisääntulon tai ulostulon suhteeseen. Algoritmi kuvaa tätä tiettyä laskennallista menettelyä, jolla saavutetaan tämä sisääntulon ja ulostulon suhde. (Cormen ym. 2009, 5.)

Ennen kuin aletaan kehittään algoritmeja, pitää kuvata ongelma tarkasti. Ongelman kuvaamista helpottaa monesti kaavio, jossa ongelma on jaettu pienenpiin osaongelmiin. Monimutkaiset ongelmat on hyvä jäsenellä kaavion avulla, tällöin saadaan hyvä pohja algoritmin kehittämiselle. (Kolehmainen 2006, 7–8.)

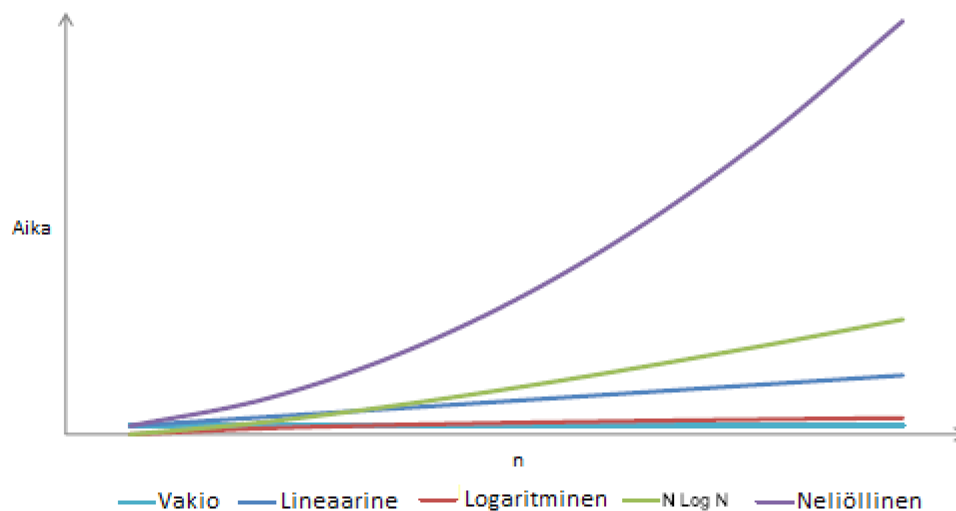
Yksinkertaisessa esimerkissä on aina hyvä miettiä, mikä on ongelma ja mikä on sisääntulo ja ulostulo ohjelmalle. Esimerkiksi ongelmana voisi olla lajittelu, sisääntulo ohjelmalle olisi järjestää n-arvoja $a_1 \dots a_n$, ja ulostulo olisi uudelleen järjestetty sisääntulosta saatu sarja $a_1 \leq a_2 \leq \dots \leq a_{n-1} \leq a_n$. Käytännön esimerkkinä voitaisiin järjestää vaikka nimiä {Mika, Anu, Lilja, Niina, Esa} tai lista numeroita {124, 242, 544, 532, 555}. Kun ongelma on määritelty, ollaan ottamassa ensimmäisiä askelia sen ratkaisemiseksi. Algoritmi on menettely, joka ottaa minkä tahansa sisääntulon ja muuttaa sen halutuksi ulostuloksi. (Skiena 2008, 1.)

Monet ohjelmat käyttävät lajittelualgoritmeja välivaiheena, joten lajittelu on perusoperaatio tietotekniikassa. Tästä johtuu, että on paljon hyviä lajittelualgoritmeja käytössä. Ohjelmaa tehtäessä yritetään aina päättää tehokkain ja nopein algoritmi. Algoritmia päättäessä tulee aina katsoa, kuinka iso määrä kohteita pitää lajitella, millaisessa järjestyksessä kohteet ovat valmiiksi ja mahdollisia rajoituksia kohteiden arvoissa. (Cormen, Leiserson, Rivest & Stein 2009, 5.)

2.1 Algoritmien tehokkuuden kuvaaminen

Algoritmien tehokkuutta voidaan kuvata O-notaatiolla. Ajoajan kompleksisuusanalyysiin käytetään O-notaatiota. Syy minkä takia O-notaatiota käytetään on se, että siitä saadaan abstrakti mittaustulos, jolla voidaan arvioida algoritmin suorituskykyä ilman matemaattisia kaavoja. (Granville & Luca Del 2008, 1.)

Kun halutaan päättää algoritmi on tärkeätä valita tehokas algoritmit, joka sopii ohjelmaan hyvin (KUVIO 1). Kuviosta on jätetty kuutiollinen $O(n^3)$ ja ekspotentiaalinen $O(2^n)$ ajoaika, jotta kuvio pyisi selkeänä. Kuutiollisia ja ekspotentiaalisia algoritmeja käytetään hyvin harvoin ja todella pieniin ongelmiin. Näitä algoritmejä on hyvä välttää aina kun mahdollista. Algoritmin suunnittelu vaiheessa, jos saadaan kuutiollinen tai ekspotentiaalinen O-notaatio on hyvä käydä mallin suunnitelu läpi uudestaan. Mallia läpi käydessä on aina hyvä yrittää löytää tehokkain ja nopein tapa ratkaista ongelma. (Granville & Luca Del 2008, 2.)



KUVIO 1. Algoritmisen ajoajan laajentuminen (mukaillen Granville & Luca Del 2008, 2)

Yleisimpiä O-notaatiota on viisi kappaletta: O(1), O(n), O(log n), O(n log n), $O(n^2)$. O(1)-notaatio on vakio, eli operaatio ei ole riippuvainen sisääntulon suuruudesta. Solmujen lisääminen operaation linkitetyn listan hännälle ei vaikuta nopeuteen, koska osoittaja on aina häntä solmulla. O(n)-notaatio on lineaarinen, eli ajoajan mutkitkuus on suhteellinen luvun n kokoon. O(log n)-notaatio on logaritminen, eli normaalisti tämä notaatio

yhdistetään algoritmeihin, jotka pilkkovat ongelman pienenpiin paloihin. O(n log n)-notaatiota yhdistetään yleensä algoritmeihin, jotka pilkkovat ongelman osiksi ja sitten ottavat tulokset ja nitovat ne takaisin kasaan. $O(n^2)$-notaatio on neliöllinen, esimerkiksi kuplalajittelu. (Granville & Luca Del 2008, 2.)

O-notaation suurin vahvuus on siinä, että se antaa mahdollisuuden hylätä asoita, kuten laitteiston. Oletetaan, että on kaksi lajittelevaa algoritmiä, toinen on neliöllinen ja toinen on logaritminen. Kun datan määrä kasvaa, logaritminen algoritmi ohittaa neliöllisen algoritmin nopeudessa. Tämä toteutuu, vaikka neliöllinen algoritmi ajettaisiin nopeammalla tietokoneella. Näin tapahtuu, koska o-notaatio eristää avaintekijän algoritmien analysoimisessa: kasvun. Neliöllinen algoritmi kasvaa paljon nopeammin kuin logaritminen algoritmi. Yleisellä tasolla sanotaan, että jossain välissä n -> ∞, ja sitten logaritminen algoritmi on nopeampi kuin neliöllinen algoritmi. (Granville & Luca Del 2008, 3.)

2.2 Klassisten ongelmien ratkaisuja algoritmeilla

Lajittelu ei ole ainut laskennallinen ongelma, jolle algoritmeja on kehitetty. Käytännön sovelluksia on tehty algoritmeilla moniin eri tarkoituksiin. Algoritmeja on yritetty soveltaa käytäntöön niin kauan kuin on tietokoneita ohjelmoitu, joten ne alkavat olemaan jo kohtuullisen kehittyneitä. Human Genome Project on edistynyt hyvin tunnistaakseen kaikki ihmisen 100 000 geeniä, jotka ovat DNA:ssa. Jotta saataisiin kaikki DNA:sta löytyvä tieto tallennettua tietokantoihin ja kehitettyä työkalut, jolla tutkia dataa, tarvitaan oikein kehittyneitä algoritmeja. Kyseisessä projektissa käytetään monenlaisia eri algoritmeja. (Cormen ym. 2009, 6.)

Internetin kehittämiseen tarvittiin monia algoritmeja, joilla sivut voivat manipuloida isoja määriä tietoa. Yksi tärkeimpiä algoritmeja on se, jolla tieto saadaan ohjattua hyviä reittejä pitkin oikeaan paikkaan. Toisena tärkeänä ovat hakualgoritmit, joilla isoista määristä tietoa voidaan hakea haluttu tieto. (Cormen ym. 2009, 6.)

Sähköisessä kaupankäynnissä käytetään paljon luottokortteja, salasanoja ja tiliotteita. Sähköinen kaupankäynti myös perustuu pitkälti siihen, että henkilökohtaisia tietoja

voidaan suojata hyvin. Näiden teknologioiden ydin on julkinen avain -salausalgoritmi ja digitaalinen allekirjoitus. Nämä perustuvat numeerisiin algoritmeihin ja numeroteoriaan. (Cormen ym. 2009, 7.)

2.3 Algoritmien kuvaaminen

Algoritmien kuvaaminen on melkein mahdotonta ilman tarkkaa kuvausta siitä, miten se toimii. Kolme yleisintä algoritmien kuvaamismenetelmää on englannin kieli, pseudokoodi ja oikea ohjelmointikieli. Pseudokoodi on näistä kolmesta mysteerisin. Pseudokoodi on helpointa selittää siten, että se on ohjelmointikieli, jossa ei tapahdu koskaan muotovirheitä. Kaikki kolme tapaa ovat käytännöllisiä, koska näillä kolmella voidaan esittää algoritmi tarkasti mutta kuitenkin tarpeeksi yksinkertaisesti. Englannin kieli on näistä kolmesta luonnollisin, mutta ei läheskään niin tarkka kuin ohjelmointikielet. Ohjelmointikielistä esimerkiksi JAVA ja C/C++ ovat tarkkoja mutta vaikeita ymmärtää ja kirjoittaa. Pseudokoodi on yleisesti käytännöllisin, koska sillä esitetään englannin kielen ja ohjelmointikielen puoliväliä. Kun valitaan, mitä käytäntöä halutaan käyttää, se riippuu pitkälti tekijästä. Tekijä valitsee yleensä niitä algoritmeja, jolla hän on tottunut ratkaisemaan ongelmia. Hyvät ohjelmoijat osaavat ratkoa ongelmat niin, että valitsevat yleensä parhaan ja tehokkaimman algoritmin. (Skiena 2008, 12.)

Pseudokoodia yleensä pidetään jäljitelmä ohjelmointikielenä, joka on suunniteltu enemmänkin luettavaksi kuin ajettavaksi ohjelmointikieleksi. Pseudokoodin tapaisia ohjelmointikieliäkin on esimerkiksi Python-ohjelmointikieli. Pythonilla ohjelmoitaessa koodi on melkein yhtä luettavaa kuin pseudokoodi. Todellisuudessa Pythonillakin ohjelmoitaessa ilman tarkempaa tuntemusta ohjelmointikielestä ainoastaan yksinkertaiset ohjelmat on helppo selvittää. (Hetland 2010, 2.)

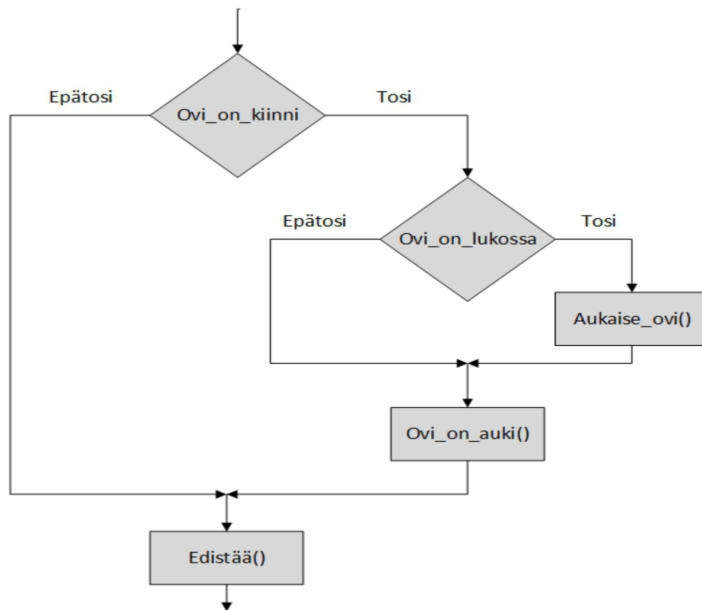
Yksinkertainen esimerkki pseudokoodista:

```
Haetaan positiivinen kokonaisluku sisääntulosta
if n > 10
    print ”Tämä voi kestää hetken”
for i = luvusta 1 lukuun n
    for j = luvusta 1 lukuun n
        print i * j
print ”Valmis”
```

(Fundamental Data Structures 2011, 14.)

Vuokaavioilla (KUVIO 2) voidaan kuvata pseudokoodin tavoin algoritmeja. Vuokaavio on diagrammi, jolla kuvataan prosessia tai operaatiota. Se sisältää useita askeleita, joilla kuvataan prosessin kulkua alusta loppuun. Vuokaavioita on monenlaisia, jotkin vuokaaviot ovat pieniä ja sisältävät vain muutamia askelia, toisaalta taas suuret ja monimutkaiset vuokaaviot sisältävät useita askelia. Vuokaaviota käytetään muihinkin asioihin kuin algoritmeihin, esimerkiksi liiketoiminnan kehittämiseen ja ongelman ratkontaan. (IBM Data Processing Techniques 1970, 1.)

Vuokaaviot yleisesti käyttävät standardoituja symboleja, jotka esittävät eri vaiheita tai toimintoja vuokaaviossa. Esimerkiksi jokainen askel kuvataan suorakulmiolla ja jokainen päätös kuvataan timanttikuviolla. Vuokaavioita voidaan tehdä käsin, mutta on olemassa monia tietokoneohjelmia, joilla voidaan luoda helposti tarkkoja vuokaavioita. (IBM Data Processing Techniques 1970, 1.)



KUVIO 2. Esimerkki vuokaavio, joka kuvaa sisäkkäisen ehtolauseen logiikkaa (mukaillen Goodrich, Tamassia & Goldwasser 2013, 19)

3 YLEISIÄ ALGORITMIEN MENETTELYTAPOJA

3.1 Raaka voima

Raaka voima on yksinkertainen, työntekoon perustuva, iteratiivinen ratkaisumalli, jos halutaan etsiä jotain tai halutaan optimoida funktiota. Tietotekniikassa raa'an voiman haut on tunnettu myös tuottaa ja testata algoritmeina. (Goodrich ym. 2013, 584.)

Raaka voima -algoritmin tutkiminen on hyvin yksinkertaista. Iteratiivista ratkaisutapaa käytetään esimerkiksi joissakin lajitteluissa, etsinnässä ja yhtälön ratkaisemisessa. Hyvä esimerkki olisi, että etsintä algoritmi sisältää kaksi sisäkkäistä silmukkaa, joista ulompi silmukka indeksoi kaikkien mahdollisten aloitusviittauksien läpi. Sisempi silmukka indeksoi läpi jokaisen merkin tekstistä vertaillen niitä mahdollisiin vastaaviin merkkeihin tekstissä. Tästä johtuu, että raaka voima -algoritmin virheettömyys seuraa suoraa tästä tyhjentävästä hausta. (Goodrich ym. 2013, 585.)

Ajoaikaa tutkittaessa huomataan, että pahimmassa tapauksessa raaka voima -algoritmi on hyvin hidas. Tämä johtuu siitä, että jos käydään isoa määrää dataa läpi, algoritmi joutuu vertailemaan niin paljon merkkejä, että ohjelma hidastuu. Raaka voima -algoritmia käytetään kuitenkin hyvin paljon, ja se on yleisesti ensimmäisiä vaihtoehtoja ongelman ratkaisuun. (Goodrich ym. 2013, 585.)

3.2 Ositus ja kokoaminen

Ositus ja kokoaminen eli hajota ja hallitse -periaate, joka oli Rooman armeijan strategia hallita suurta imperiumia. Tämä on tunnettu algoritmien suunnittelutekniikka. Idea tälle suunnittelutekniikalle on se, että yritetään ositella ongelman esiintymä useaksi pienemmäksi esiintymäksi, ja nämä osaongelmat ovat keskenään suunnilleen samankokosia. Yleensä rekursiivisesti ratkaistaan pienemmät osaongelmat, mutta muitakin tapoja käytetään. Joissakin tapauksissa alkuperäisen ongelman esiintymisen ratkaisu pitää muodostaa, ja tarvittaessa se kootaan osaongelmien ratkaisusta. (Dasgupta, Papadimitriou & Vazirani 2006, 55–56.)

Yleinen muoto osittamiselle:

```
vastaus osittava (tapaus x)
{
    if (x on "pieni")
        y = pienen tapauksen ratkaisu;
    else {
        jaetaan x osiin x1, …, xm;
        // osat yleensä tasakokoisia, ei välttämättä erillisiä
        y1:=osittava(x1);
        …
        ym:=osittava(xm);
        kokoa x:n vastaus y osista y1, ..., ym;
    }
    return y;
}
```

(mukaillen Heineman, Pollice & Selkow 2009, 49.)

Rekursiivisen funktio

Objektia sanotaan rekursiiviseksi, jos se osittain koostu tai se määritellään itsessään. Toistumista ei tapahdu pelkästään matematiikassa vaan myös jokapäiväisessä elämässä. Jokapäiväisen elämän esimerkki olisi mainoskuva, joka sisältää itsensä. Toistuminen on hyvin voimakas tekniikka matemaattisissa määritelmissä. Toistumisen voima koostuu siinä mahdollisuudessa, että loputon määrä objekteja määritellään äärellisellä lausekkeella. (Wirth 2004, 99.)

Rekursiivinen funktio menee käsi kädessä osituksen ja kokoamisen kanssa, koska molemmat antavat luonnollisen tavan luonnehtia ajoaikoja ositus- ja kokoamisalgoritmiin. Rekursiivi on yhtälö tai epäyhtälö, joka kuvaa funktion kannalta sen arvoa pienimmissä sisääntuloissa. Rekursiivi voi ottaa monia eri muotoja. Esimerkiksi rekursiivinen algoritmi voi osittaa aliongelman erisuuriksi palasiksi, kuten 2/3- ja 1/3- palasiksi. Jos ositus- ja kokoamiskohdat ottavat lineaarisen ajan, tällainen algoritmi aiheuttaisi toistumisen $T(n) = T(2n/3) + T(n/3) + \Theta(n)$. (Cormen ym. 2009, 65–66.)

Toistuminen voidaan ratkaista saavuttamalla asymptoottinen ”Θ” tai ”O” raja-arvo seuraavilla tavoilla. Ensimmäinen tapa on sijaisuusmenetelmä, jossa arvaillaan raja-arvo ja sitten käytetään matemaattista induktiota todistamaan arvaus oikeaksi. Toisena tapana on rekursiivinen puu -menetelmä, jolla muuntaa toistumisen puuksi. Puun solmut edustavat aiheutuneita kustannuksia eri tasoilla toistossa. Kolmantena tapana on mestari-menetelmä, joka tarjoaa raja-arvon toistolle muodossa: $T(n) = aT(n/b) + f(n)$. Tässä kaavassa $a \geq 1$, $b > 1$ ja $f(n)$ on annettu funktio. Tälläinen toistuminen tapahtuu usein. Edellä esitetyssä kaavassa tapahtuva toistuminen luonnehtii ositus- ja kokoamisalgoritmia, joka luo a-aliongelmia, joista jokainen on 1/b on alkuperäisen ongelman koko. Ositus- ja kokoamisaskeleet yhdessä ottavat f(n)-ajan. (Cormen ym. 2009, 66.)

3.3 Ahneet algoritmit

Ahneet algoritmit ovat taipuvaisia suoraan tyydyttämiseen. Ilman, että katsotaan pitemmälle tulevaisuuteen ja tehdään jokaisen askel loogisesti, algoritmi ei välttämättä toimi hyvin. Esimerkki ahneesta menetelmästä olisi se, että kaupan myyjä jakaisi pikkurahaa. Jotta myyjä käyttäisi mahdollisimman vähän kolikoita, antaisi hän arvokkaimmat kolikot pois ensin, ennen kuin hän siirtyisi pienenpiin kolikoihin. (Soltys 2012, 39.)

Ahne tapa toimii hyvin joissakin laskennallisissa ongelmissa mutta joissakin se taas ei toimi. Jos olisi kolikkoja, joiden arvo olisi 1, 5 ja 25. Ahne toimintapa aina tuottaa pienimmän mahdollisen määrän kolikoita. Jos on kolikoita, joiden arvo on 1, 10, 25 ja sitten 30, jakaminen ahneesti tapahtuu kuutena kolikkona (25, 1, 1, 1, 1, 1). Vähemmän ahne ja järkevämpi myyjä antaisi ainoastaan kolme kolikkoa (10, 10, 10). (Soltys 2012, 39.)

3.4 Esimerkkejä algoritmeista

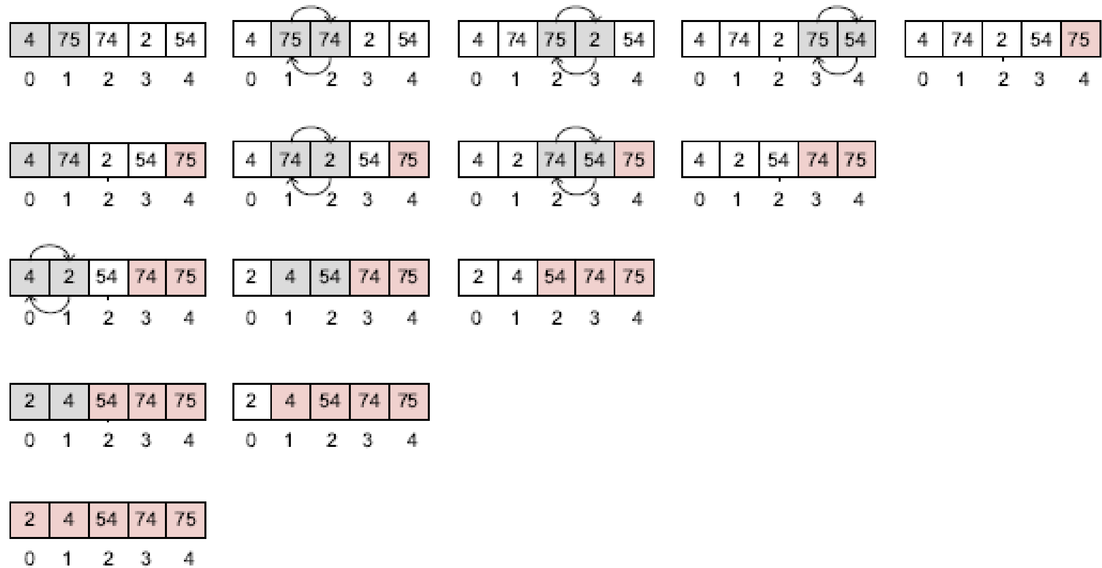
3.4.1 Raaka voima

Yksi yksinkertaisimmista lajittelualgoritmeista on kuplalajittelu. Kuplalajittelulla tarkoitetaan sitä, että verrataan jokaista alkiota joka toiseen alkioon listassa (KUVIO 3). Tämä lajittelutapa ei välttämättä kuitenkaan ole paras mahdollinen vaihtoehto tai tehokkain. Kuplalajittelussa käytetään kahta silmukkaa ongelman ratkaisuun. (Granville & Luca Del 2008, 63.)

Pseudokoodi kuplalajittelulle:

```
Algoritmi kuplalajittelu(lista)
    Alku: lista ≠ 0
    Loppu: lista on lajiteltu nousevaan järjestykseen
    for i <- 0 jotta list.Count – 1
        for j <- 0 jotta list.Count – 1
            if lista[i] < lista[j]
                vaihdetaan(lista[i], lista[j])
            end if
        end for
    end for
    return lista
end kuplalajittelu
```

(Granville & Luca Del 2008, 63.)



KUVIO 3. Kuplalajittelun iterointi (Granville & Luca Del 2008, 64)

Toinen hyvin tyypillinen raa'an menetelmän algoritmi on valintamenetelmä. Periaate valintamenetelmän selittämiselle on, että otetaan pino kortteja, joissa on numeroita. Yleisin tapa järjestellä pino on siten, että otetaan aina kortti, jossa on suurin numero, ja poistetaan se pinosta. Tätä tapaa toistetaan niin kauan, että kaikki kortit on käyty läpi. (Heineman ym. 2009, 91.)

Valintamenetelmä on kohtuullisen hidas lajittelualgoritmi, koska se tarvitsee parhaassakin tapauksessa kuutiollisen ajan. Tämä menetelmä toistuvasti suorittaa melkein samoja tehtäviä ilman, että se oppii mitään edellisestä iteroinnista. Esimerkki valintamenetelmästä on suurimman alkion etsiminen. Valintamenetelmällä menee suurin osa vertailuista hukkaan, koska jos alkio on pienempi kuin aikaisempi alkio, se ei voi olla suurin alkio taulukossa. Tästä voidaan päätellä, että valintamenetelmän vertailulla ei ole kovinkaan suurta vaikutusta suurimman alkion hakemiseen. (Heineman ym. 2009, 92.)

Ohjelmakoodi valintamenetelmälle:

```
public class Selection
{
   public static void sort(Comparable[] a)
   {                                        // Lajitellaan taulu a[] nousevaan järjestykseen
      int N = a.length;                     // Taulukon pituus
      for (int i = 0; i < N; i++)
      {                                     // Vaihdetaan a[i] pienimällä tulolla a[i+1…N]
         int min = i;                       // Pienimmän tulon indeksi
```

```
            for (int j = i+1; j < N; j++)
               if (less(a[j], a[min])) min = j;
            exch(a, i, min);
         }
      }
}
```

(Sedgewick & Wayne 2011, 249.)

3.4.2 Osita ja kokoa

Paradigmaattinen esimerkki osita ja kokoa -algoritmille on mergesort, jolla voidaan lajitella lista, jossa on paljon alkioita. Mergesort toimii siten, että lista rikotaan kahteen pienenpään listaan, jotka sitten lajitellaan rekursiivisesti ja sitten listat yhdistetään uudelleen isoksi lajitelluksi listaksi. Oletetaan, että on kaksi listaa numeroita valmiiksi lajiteltuna. Silloin meillä on lista $a_1 \leq a_2 \leq . . . \leq a_n$ ja $b_1 \leq b_2 \leq . . . \leq b_m$. Nämä kaksi listaa halutaan yhdistää lajitelluksi listaksi $c_1 \leq c_2 \leq . . . \leq c_{n+m}$. (Soltys 2012, 64.)

Pseudokoodi mergesortille:

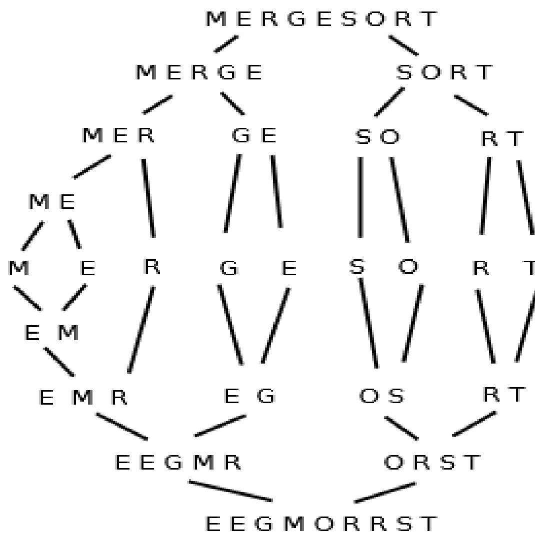
```
mergesort(alkio_tyyppi s[], int matala, int korkea)
{
   int i;                    /* laskuri */
   int keskikohta;           /* keskielementin indeksi */

   if (matala < korkea) {
      keskikohta = (matala + korkea) / 2;
      mergesort(s,matala,keskikohta);
      mergesort(s,keskikohta+1,korkea);
      merge(s, matala, keskikohta, korkea);
   }
}
```

(Skiena 2008, 122.)

Mergesortin tehokkuus riippuu siitä, kuinka hyvin kaksi lajiteltua listaa yhdistetään yhdeksi listaksi. Nämä kaksi pienempää listaa voitaisiin ketjuttaa yhteen ja kutsua jotain muuta lajittelualgoritmia, mutta tämä tuhoaisi kaiken työn, jota on tehty osalistojen lajitteluun. Sen sijaan voidaan liittää nämä kaksi listaa yhteen (KUVIO 4). Mergesort on loistava algoritmi linkitettyjen listojen lajitteluun, koska se ei turvaudu satunnaishakuihin. Sen suurin ongelma on, että se tarvitsee avustavan puskurin, kun lajitellaan taulukkoja.

Kokonaisajoaika mergesortille voidaan määrittää sillä, kuinka paljon työtä tehdään jokaiseen eri suorituspuun tasoon. (Skiena 2008, 122.)



KUVIO 4. Mergesortin toiminnasta vuokaavio (Skiena 2008, 122.)

Mergesort on klassinen osita ja kokoa -algoritmi. Aina kun pystytään rikkomaan ongelma kahteen pienenpään ongelmaan, ollaan hyvissä asemissa. Kaksi pientä ongelmaa on aina helpompi ratkaista kuin yksi iso ongelma. Hyvä keino on ottaa hyötyä siitä, että kahdella osittain ratkaistulla ongelmalla voidaan saada ratkaisu koko ongelmaan. (Skiena 2008, 122.)

4 TIETORAKENTEET

Moderni digitaalinen tietokone kehitettiin sitä varten, että sillä voitaisiin helpottaa ja nopeuttaa monimutkaisia paljon aikaa vieviä laskuja. Suurimmalla osalla nykyaikaisista sovelluksista pystyy tallentamaan suuria määriä dataa, ja sitä pystytään käsittelemään helposti. (Wirth 2004, 10.) Tietorakenteita ja algoritmeja rakentaessa pitää syöttää tarkkoja ohjeita tietokoneelle. Parhaita tapoja kommunikoida tietokoneen kanssa on käyttää korkean tason tietokonekieltä, esimerkiksi Python, C++, C#. (Goodrich ym. 2013, 2.)

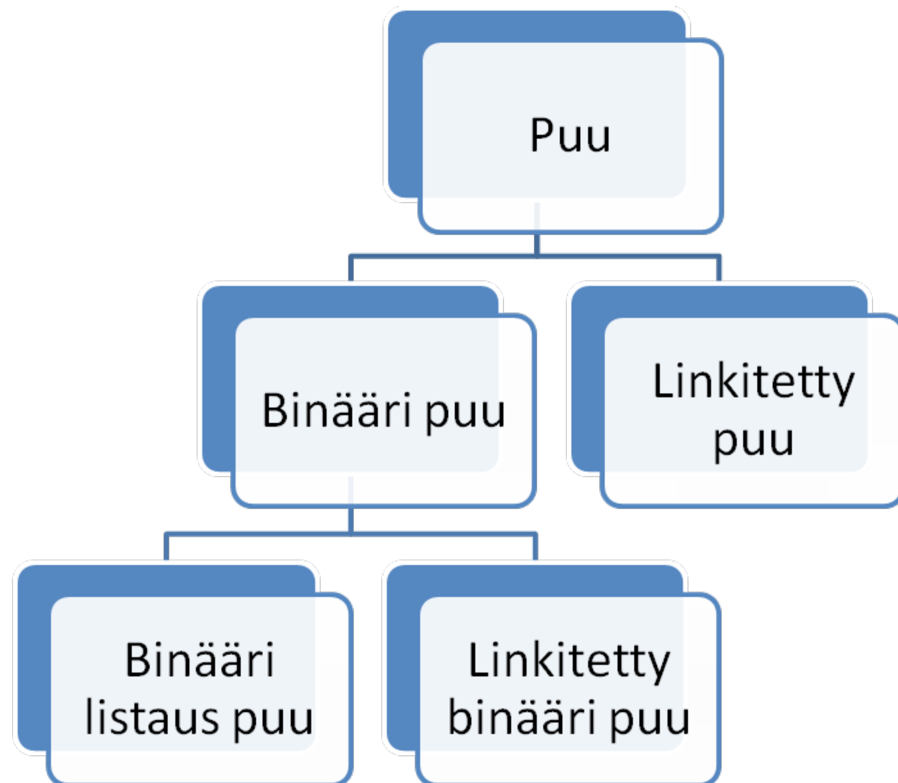
4.1 ADT

Tietotekniikassa ADT (Abstract data type) on matemaattinen malli tietyille tietorakenneluokille, joilla on samantapainen käytös. ADT määritellään epäsuorasti ainoastaan niihin operaatioihin, joissa se voidaan toteuttaa, ja operaation matemaattisiin rajoituksiin. Esimerkiksi lyhennelmä keosta voitaisiin määritellä kolmeen operaatioon. Ensimmäisenä operaationa käytetään työntöä. Työntö operaatiolla lisätään data-alkioita rakenteeseen. Toinen operaatio on veto. Veto operaatiolla otetaan data-alkioita rakenteesta. Kolmas operaatio on kurkkaus, sillä saadaan tutkittua tietorakennelman päällä oleva data. Kurkkaus operaatio ei poista mitään tietorakennelmasta, koska se pystyy tutkimaan dataa ilman sen poistamista. (Skiena 2008, 65.)

4.2 Puut

Tuotantoasiantuntijat sanovat, että läpimurrot tulevat epälineaarisesta ajattelusta. Tietotekniikassa tärkein epälineaarinen tietorakenne on puut. Puurakenne on eräänlainen läpimurto dataorganisaatioissa. Puilla pystytään toteuttamaan algoritmin isäntä paljon nopeampaa kuin lineaarisilla tietorakenteilla esimerkiksi, järjestelylistoilla tai linkitetyillä listoilla. Puilla luodaan myös luonnollinen järjestys tiedolle ja puurakenteista on tulossa yleinen rakenne tiedostojärjestelmille, graafisille käyttöjärjestelmille, web-sivuille ja muille tietotekniikan järjestelmille. (Goodrich ym. 2013, 300.)

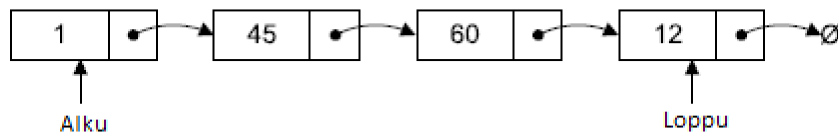
Puu on abstrakti tietotyyppi, joka tallentaa tietoa hierarkkisesti (KUVIO 5). Lukuun ottamatta ylintä alkiota kaikilla alkioilla puussa on vanhempi alkio ja nolla tai enemmän lapsialkioita. Puuta yleensä kuvataan laittamalla alkioita laatikoihin ja yhdistämällä niitä suorilla viivoilla. Yleisesti ylintä alkiota kutsutaan puun juureksi, vaikka se onkin ylimpänä puussa. (Goodrich ym. 2013, 301.)



KUVIO 5. Esimerkki puuhierarkiasta (mukaillen Goodrich ym. 2013, 303.)

4.3 Linkitetyt listat

Linkitetty lista on tietorakenne, jossa objektit on järjestetty lineaariseen järjestykseen (KUVIO 6). Toisin kuin taulukossa, jossa lineaarinen järjestys on määritetty taulukon järjestyksestä. Linkitetyn listan järjestys on määritetty taas osoittimella erikseen jokaisessa objektissa. Linkitetty lista tarjoaa yksinkertaisen ja joustavan esityksen dynaamiselle joukolle dataa. (Cormen ym. 2009, 236.)



KUVIO 6. Esimerkki linkitetystä listasta, jossa on kokonaislukuja (mukaillen Granville & Luca Del 2008, 10)

Linkitettyjä listoja on hyvä käyttää silloin, kun on tuntematon määrä alkioita talletettavaksi. Jos haluttaisiin käyttää taulukkotietorakennetta, pitäisi olla etukäteen annettu taulukon koko. Taulukossa on myös se huono puoli, että jos alkiot ylittävät taulukon koon, pitää algoritminkin kokoa muuttaa. Linkitettyä listaa pitäisi käyttää aina silloin, kun halutaan poistaa solmuja listan lopusta tai alusta. Tällä taataan se, että lista ylläpitää muuttumatonta ajoaikaa. Tämä tarkoitta sitä, että täytyy pitää osoittajat listan pää- ja häntäsolmuissa. Yksittäisiä linkitettyjä listoja pitäisi ainoastaan käyttää ohjelmissa, joissa listaan tulee peruslisäyksiä. (Granville & Luca Del 2008, 17.)

Esimerkki listan alkiosta:

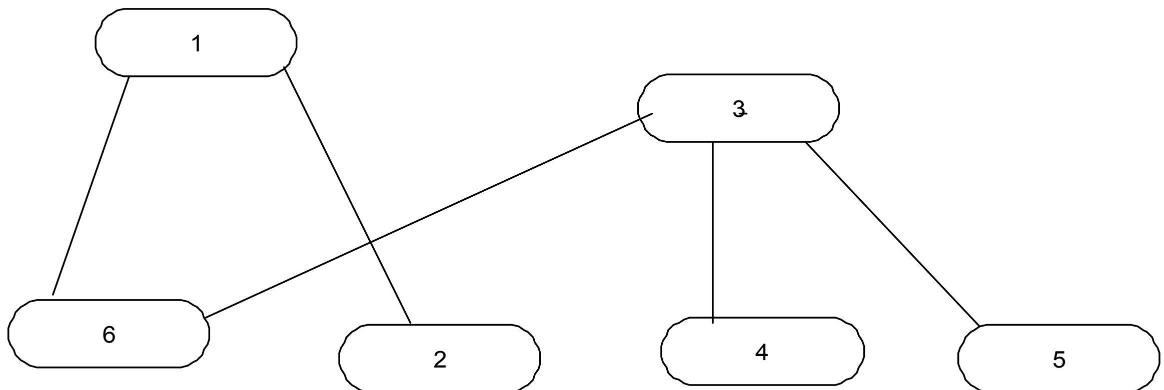
```
private class Solmu
{
    Item arvo;
    solmu next;
}
```

(Sedgewick & Wayne 2011, 142.)

4.4 Graafit

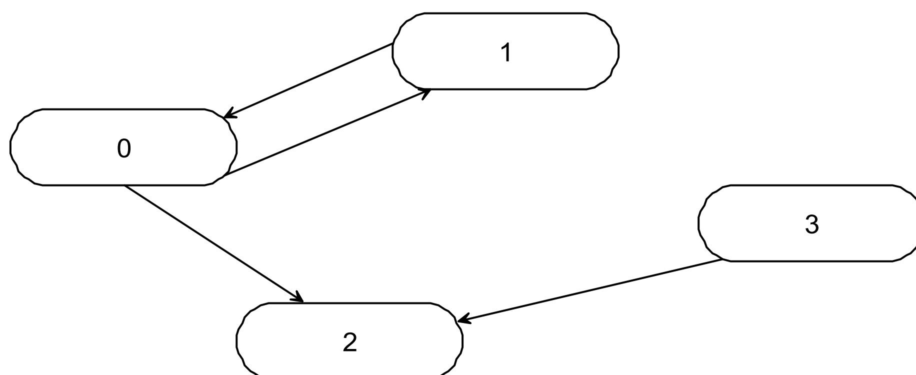
Graafit voivat esittää kaikenlaisia rakenteita ja systeemejä. Graafeilla voidaan esittää esimerkiksi kuljetusverkkoja, kommunikaatioverkkoja, proteiinien vuorovaikutusta soluissa ja ihmisten vuorovaikutusta Internetissä. Graafien ilmaisua voidaan kasvattaa lisäämällä ylimääräistä dataa, kuten painoja tai välimatkaa. Tällä tehdään mahdolliseksi uudelleen esittää monipuolisia ongelmia, kuten shakkia, tai yhteen sovittaa ihmisiä työhön, jota heidän osaamisensa parhaiten vastaa. Monissa tapauksissa, jos pystytään tekemään graafi ongelmalle, on puolet ongelmasta jo selvitetty. (Hetland 2010, 23.)

Graafi G = (V,E) määritellään kytkemällä yhteen solmut siten, että jokainen kaari kytkee kaksi solmua eivätkä kaaret kytke noita kahta solmua. Tietynlaisia graafeja esiintyy yleisesti algoritmeissa, esimerkiksi suuntaamaton graafi, suunnattu graafi ja painotettu graafi. Suuntaamattomalla graafi -mallilla tarkoitetaan sitä, että graafin solmut eivät välitä suunnasta (KUVIO 7). Näitä graafeja käytetään symmetrisen tiedon kiinni ottamiseen esimerkiksi tie, joka menee kaupunki A:sta kaupunki B:hen ja sitä voidaan matkustaa kumpaan suuntaan tahansa. (Heineman ym. 2009, 142.)



KUVIO 7. Esimerkki suuntaamattomasta verkosta

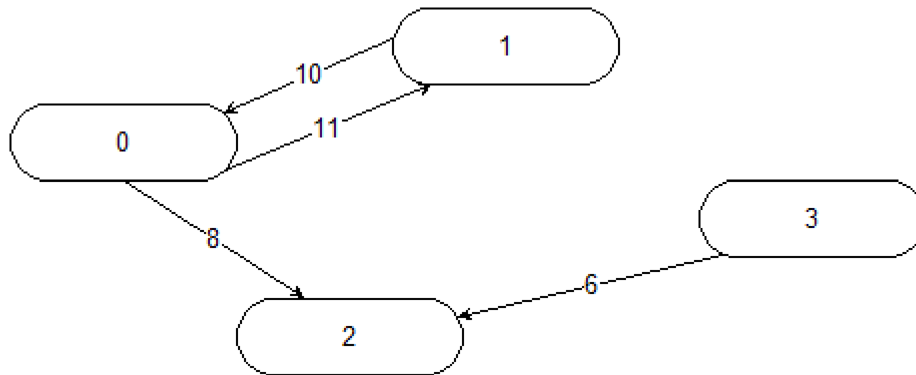
Suunnatulla graafimallilla tarkoitetaan sitä, että graafin solmujen pitää erottaa, onko solmuilla yhteyttä vai ei (KUVIO 8). Hyvä esimerkki suunnattuun graafiin on ohjelma navigaattoreissa. Navigaattorin pitää tietää, onko tie yksisuuntainen vai ei. (Heineman ym. 2009, 142.)



KUVIO 8. Esimerkki suunnatusta graafista

Painotetulla graafimallilla tarkoitetaan sitä, että numeerinen arvo eli paino liittyy solmujen yhteyteen. Joskus näitä arvoja voidaan tallentaa satunnaiseen ei-numeeriseen tietoon

(KUVIO 9). Esimerkiksi kaupunkien A ja B väliset reunat voisivat tallentaa välimatkan kaupunkien välillä, tai vaihtoehtoisesti ne voisivat tallentaa arvioidun matkan minuuteissa. (Heineman ym. 2009, 143.)



KUVIO 9. Esimerkki painotetusta graafista

5 REITIN HAKEMINEN

5.1 Yleistä reitinhausta

Ehkäpä algoritmien ohjelmoinnissa eniten tulee vastaan vaistonvaraisia graafilla selvitettäviä ongelmia. Näitä ongelmia tulee vastaan esimerkiksi karttaohjelmissa ja navigaatiojärjestelmissä. Graafimallissa solmut vastaavat risteyksiä ja reunat vastaavat teitä. Reunan painoja voitaisiin miettiä matkana tai kulkuaikana. Yksisuuntaisten teiden mahdollisuus tarkoittaa sitä, että pitää harkita reunapainotetuttuja suuntaverkkoja. (Sedgewick & Wayne 2011, 638.)

Ytimeltään reitinhakumenetelmä etsii graafissa yksi solmu kerrallaan ja tutkii vierekkäisiä solmuja niin kauan, että päämäärä löydetään. Yleisesti reitinhaussa halutaan etsiä lyhin reitti. Leveys ensin -menetelmä toimii graafista etsinnässä, ja se löytää päämäärän kyllä, jos annetaan sille tarpeeksi aikaa. Graafista etsinnässä on muitakin menetelmiä, ja nämä menetelmät tutkivat graafia ja yleensä löytävät päämäärän nopeammin kuin leveys ensin -menetelmä. (Delling, Sanders, Schultes & Wagner 2009, 1.)

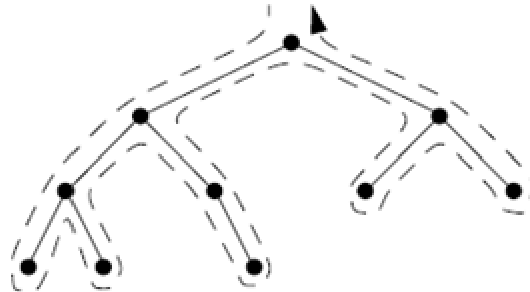
5.2 Graafien läpikäynti

5.2.1 Syvyys ensin -haku

Syvyys ensin -haku tulee englanninkielisistä sanoista depth-first search, ja siitä käytetään paljon lyhenettä DFS. Syvyys ensin -haku on hyvin käytännöllinen graafien tarpeellisuuden testaamiseen ja myös siihen, onko graafissa reittiä solmusta toiseen ja onko graafi yhdistetty. (Goodrich ym. 2013, 639.)

Syvyys ensin -haun voi selittää helpoiten siten, että haku menee niin syvälle kuin mahdollista (KUVIO 10). Tämä haku tutkii viimeisimmän solmun reunat. Kun haku pääsee jonkin haaran pohjalle, se palaa samaa tietä kuin se tuli sinne ja menee seuraavaan ei-käytyyn solmuun. Tätä jatketaan niin kauan, että on löydetty kaikki tavoitettavissa

olevat solmut. Jos puuhun jää joitain ei-löydettyjä solmuja, syvyys ensin -haku valitsee niistä uuden alkupaikan ja aloittaa haun taas. Algoritmi toistaa tätä niin kauan, että se löytää kaikki solmut. (Cormen ym. 2009, 603.)



.

KUVIO 10. Esimerkki syvyys ensin -hausta (mukaillen Goodrich ym. 2013, 640)

Pseudokoodi syvyys ensin -haulle:

Algoritmi syvyys ensin -haku(Solmu n, graafi g)
Input: ajetaan solmu n graafissa g
visited(n, g) // merkitään solmu käydyksi
neighbourSet <- neighbours(n, g);
for each neighbour **in** neighbourSet **do**
  if not isVisited(neighbour)
    syvyys-ensin(neighbour, g)

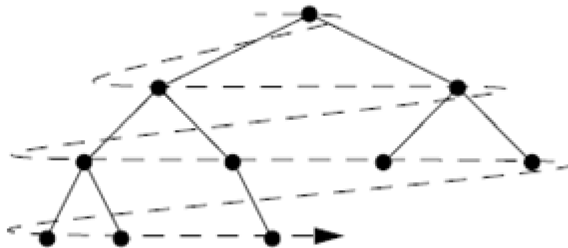
(mukaillen Heineman ym. 2009, 188.)

5.2.2 Leveys ensin -haku

Leveys ensin -haku tulee englanninkielisistä sanoista breadth-first search ja siitä käytetään yleensä lyhennettä BFS. Leveys ensin -haku on yksi yksinkertaisimmista hakualgoritmeista graafille ja arkkityyppi monille tärkeille graafialgoritmeille. Prim's -algoritmi ja Dijkstran algoritmi käyttävät samaa ideaa kuin leveys ensin -haku. (Cormen ym. 2009, 594.)

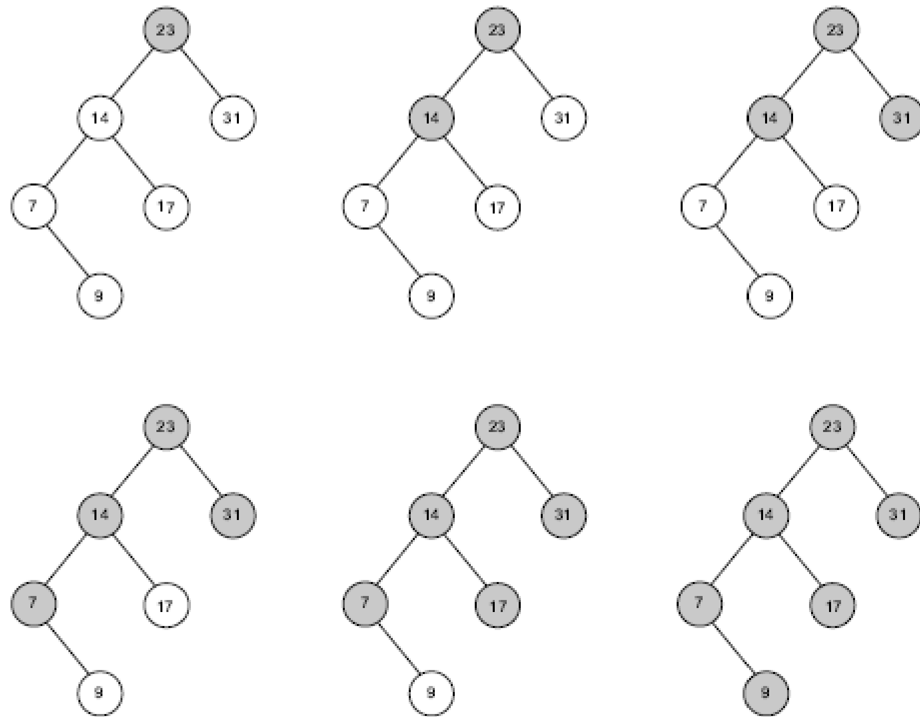
Graafissa on tunnettu lähdesolmu s, joka toimii alkupisteenä. Leveys ensin -haku tutkii järjestelmällisesti graafin reunoja (KUVIO 11), jotta se löytäisi jokaisen mahdollisen solmun graafista. Se laskee matkan lähdesolmusta kaikkiin tavoitettavissa oleviin

solmuihin. Leveys ensin -haku tuottaa myös puun jossa lähde on s ja se sisältää kaikki tavoitettavissa olevat solmut. Leveys ensin -haku toimii suunnatuissa ja suuntaamattomissa graafeissa. (Cormen ym. 2009, 594.)



KUVIO 11. Esimerkki leveys ensin -hausta (mukaillen Goodrich ym. 2013, 649)

Binääripuulla on hyvä tapa kuvailla leveys ensin -hakua (KUVIO 12). Kun Binääripuulla esitetään leveys ensin -hakua, siitä nähdään helposti, miten aloitetaan ylhäältä ja lähdetään järjestelmällisesti menemään jokaiseen solmuun erikseen (KUVIO 12). Perinteisesti leveys ensin -haku käyttää listoja käytyjen solmujen arvojen tallentamiseen ja sitten lisää solmut jonoon, joissa ei ole käyty. (Granville & Luca Del 2008, 30.)



KUVIO 12. Binääripuuesimerkki leveys ensin -hausta (Granville & Luca Del 2008, 30)

Pseudokoodi leveys ensin -haulle:

```
Algoritmi leveys-ensin(solmu n, graafi g)
Input: ajetaan solmu n graafissa g
Queue q <- empty();
visited(n,g) // Merkataan solmu käydyksi
q <- enqueue(n, q);
while not isempty(q) do
    newSolmu <- front(q)
    q <- dequeue(q);
    neighbourSet <- neighbours(newSolmu, g);
    for each neighbor in neighbourSet do
        if not isVisited(neighbour)
            visited(neigbour, g);
            q <- enqueue(neighbour, q);
```

(mukaillen Heineman ym. 2009, 156.)

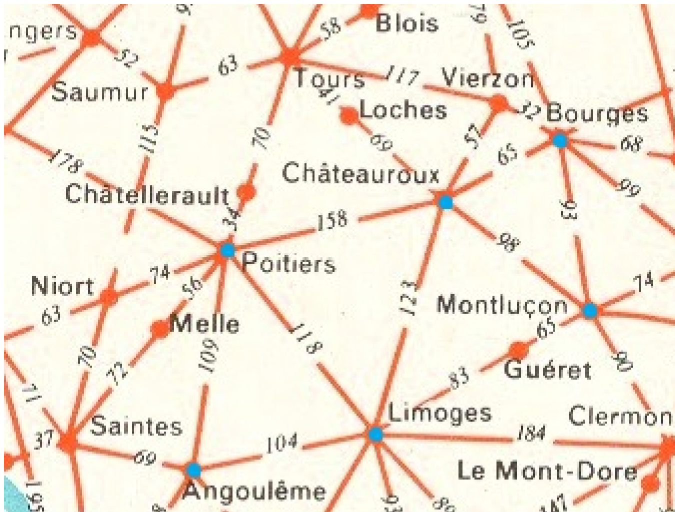
6 LYHIMMÄN REITIN HAKEMINEN

6.1 Floyd-Warshall-algoritmi

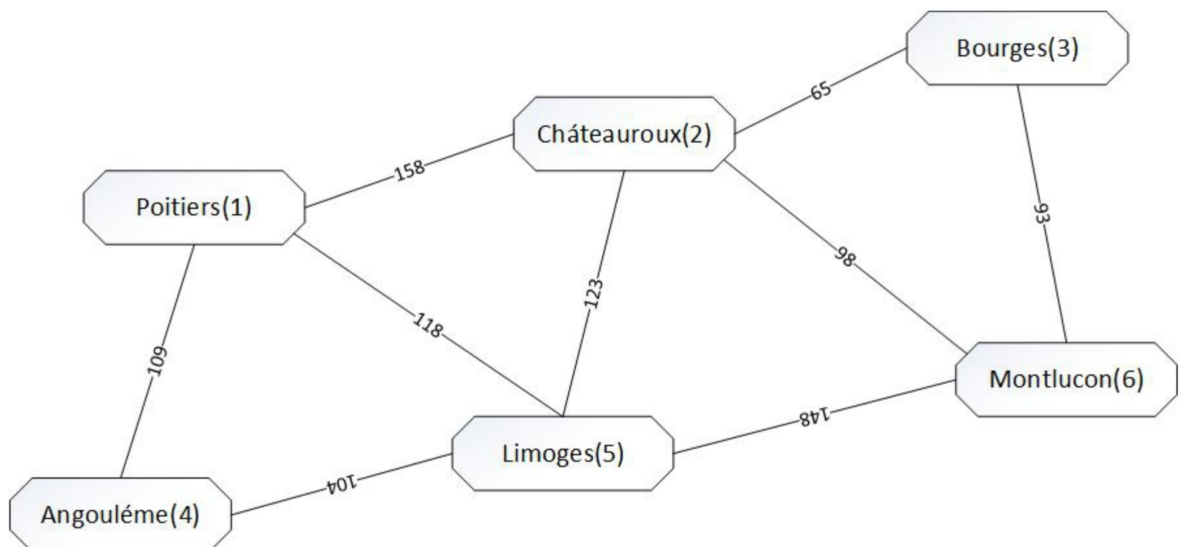
Tietotekniikassa Floyd-Warshall-algoritmi on kaavioanalyysialgoritmi, jolla etsitään lyhintä reittiä painotetussa kaaviossa, jossa on positiiviset tai negatiiviset reunapainot. Floyd-Warshall-algoritmin julkaisi nykyisessä muodossaan Robert Floyd vuonna 1962. (Cormen & Leiserson & Rivest & Stein 2009, 693.) Oletetaan, että halutaan löytää keskimmäinen solmu graafista, joka minimoi pisimmän tai keskiarvon matkasta kaikille muille solmuille. Floyd-Warshall-algoritmi on sitä varten loistava valinta. Voidaan miettiä myös, mitä käyttötarkoituksia kyseisellä algoritmilla on. Sitä voidaan esimerkiksi käyttää vaikka tietoverkon pakettien reittien hakemiseen. Tietoverkkojen pakettien reittien hakeminen vaatii algoritmia, joka laskee lyhimmän reitit kaikkien solmujen välillä annetussa graafissa. (Skiena 2008, 210.)

Floyd-algoritmi on nopea tapa rakentaa n * n etäisyysmatriisi alkuperäisestä painomatriisista. Floyd-algoritmi toimii parhaiten vierekkäismatriisitietorakenteessa, jossa ei ole yhtään ylimääräistä tuhlausta tilassa, koska sinne pitää tallentaa kaikki n^2:ssä olevat etäisyydet. Vierekkäismatriisityyppi mahdollistaa tilan suurimalle mahdolliselle matriisille ja seuraa, kuinka monta solmua on graafissa. (Skiena 2008, 210.)

Floyd-Warshall-algoritmin manuaalinen simulointi voidaan esimerkiksi esittää ottamalla kartasta (KUVIO 13) 6 kaupunkia ja laskemalla lyhimmät reitit niiden välillä. Valitut kaupungit laitetaan 7*7-taulukkoon (TAULUKKO 1). Taulukkoon on laitettu kaupunkien nimet ja kaupunkien välimatkat. Selkeyden vuoksi kartasta on hyvä tehdä vuokaavio johon tulevat välimatkat ja kaupunkien nimet. (KUVIO 14.)



KUVIO 13. Ranskan kaupungin kartta (BonjourLaFrance 2014)



KUVIO 14. Ranskan kaupungin kartan kuudesta kaupungista tehty vuokaavio

Aloituskohdalla etäisyysmatriisit (TAULUKKO 1) ja reittimatriisit (TAULUKKO 2) kuudelle kaupungille ovat taulukoissa, ja niistä lähdetään sitten tarkemmin simuloimaan reittejä. Reittien simuloinnissa taulukosta vertailemalla saadaan laskettua välimatkat ja reitit.

TAULUKKO 1. Etäisyysmatriisi kuudelle valitulle kaupungille

	Poitiers	Cháteauroux	Bourges	Montlucon	Limoges	Angouléme
Poitiers	0	158	INF	INF	118	109
Cháteauroux	158	0	65	98	123	INF
Bourges	INF	65	0	93	INF	INF
Montlucon	INF	98	93	0	83	INF
Limoges	118	123	INF	148	0	104
Angouléme	109	INF	INF	INF	104	0

TAULUKKO 2. Reittimatriisi kuudelle valitulle kaupungille

	Poitiers	Cháteauroux	Bourges	Montlucon	Limoges	Angouléme
Poitiers	1	2	3	6	5	4
Cháteauroux	1	2	3	6	5	4
Bourges	1	2	3	6	5	4
Montlucon	1	2	3	6	5	4
Limoges	1	2	3	6	5	4
Angouléme	1	2	3	6	5	4

Floyd-Warshall-algoritmien manuaalisessa simuloinnissa ensimmäisellä kierroksella saadaan laskettua ja muutettua jo muutamia loputtomuuksia, ja tämä selkeyttää jo taulukkoa. (TAULUKKO 3 & TAULUKKO 4.)

TAULUKKO 3. Etäisyysmatriisi kuuden kaupungin välillä kierroksella yksi

	Poitiers	Cháteauroux	Bourges	Montlucon	Limoges	Angouléme
Poitiers	0	158	INF	INF	118	109
Cháteauroux	158	0	65	98	123	267
Bourges	INF	65	0	93	INF	INF
Montlucon	INF	98	93	0	83	INF
Limoges	118	123	INF	148	0	104
Angouléme	109	267	INF	INF	104	0

TAULUKKO 4. Reittimatriisi kuuden kaupungin välillä kierroksella yksi

	Poitiers	Cháteauroux	Bourges	Montlucon	Limoges	Angouléme
Poitiers	1	2	3	6	5	4
Cháteauroux	1	2	3	6	5	5
Bourges	1	2	3	6	5	4
Montlucon	1	2	3	6	5	4
Limoges	1	2	3	6	5	4
Angouléme	1	5	3	6	5	4

Kierroksella kaksi tämän algoritmin simuloinnissa saadaan jo suurin osa loputtomuuksista laskettua (TAULUKKO 5 & TAULUKKO 6). Tässä työssä ei käydä enempää kuin kaksi kierrosta tätä manuaalista simulointia. Manuaalista simulointia ei yleisesti kannata tehdä välttämättä paljoa, koska pienellä ohjelmalla se on paljon nopeampaa ja helpompaa.

TAULUKKO 5. Etäisyysmatriisi kuuden kaupungin välillä kierroksella kaksi

	Poitiers	Cháteauroux	Bourges	Montlucon	Limoges	Angouléme
Poitiers	0	158	INF	INF	118	109
Cháteauroux	158	0	65	98	123	267
Bourges	223	65	0	93	188	INF
Montlucon	256	98	93	0	83	INF
Limoges	118	123	188	148	0	104
Angouléme	109	267	332	365	104	0

TAULUKKO 6. Reittimatriisi kuuden kaupungin välillä kierroksella kaksi

	Poitiers	Cháteauroux	Bourges	Montlucon	Limoges	Angouléme
Poitiers	1	2	3	6	5	4
Cháteauroux	1	2	2	6	5	5
Bourges	2	2	3	6	2	4
Montlucon	2	2	3	6	5	4
Limoges	1	2	2	6	5	4
Angouléme	1	5	1	2	5	4

Ohjelmakoodi Floyd-Warshall-algoritmista C++:lla:

```
#include <iostream>
#include <climits>
#define inf 2000000 //define INF as infinity value
using namespace std;

void FloydAPSP (int N, int C[5][5], int D[5][5], int P[5][5]);


int main(){
   int N = 5;

   int rmatrix[5][5] =
   {{   inf,   30,          40,          inf,          inf},
{30,          60,          70,          15,          inf},
{40,          70,          80,          35,          70},
{inf,         15,          35,          inf,         32},
{inf,         inf,         70,          32,          inf}};

    int imatrix[5][5] =
   { {1,2,3,4,5},
     {1,2,3,4,5},
     {1,2,3,4,5},
     {1,2,3,4,5},
     {1,2,3,4,5}};

     FloydAPSP(N, rmatrix, rmatrix, imatrix);

   cin.get();   cin.get();
   return 0;
}
void FloydAPSP (int N, int C[5][5], int D[5][5], int P[5][5])
  {
   int i,j,k;
  for (i = 0; i < N; i++) {
     for (j = 0; j < N; j++) {
      //  D[i][j] = C[i][j];
        P[i][j] = j;;
     }
   //  D[i][i] = 0.0;
   }

   for (k = 0; k < N; k++) {
     for (i = 0; i < N; i++) {
       for (j = 0; j < N; j++) {
         if (D[i][k] + D[k][j] < D[i][j]) {
           D[i][j] = D[i][k] + D[k][j];
```

```
          P[i][j] = k;
 } } } }
 for (int k = 0; k < 5; k++)
     {
        for (int s = 0; s < 5; s++)
          cout << D[k][s] << " ";
         cout << endl;
     }
for (int k = 0; k < 5; k++)
     {
        for (int s = 0; s < 5; s++)
          cout << P[k][s] << " ";
         cout << endl;
     }
 }
```

(mukaillen Heineman ym. 2009, 173–174.)

6.2. Dijsktran-algoritmi

Dijkstran-algoritmi on hyvä valinta, jos halutaan etsiä lyhyintä reittiä reuna- ja/tai solmupainotetussa kaaviossa. Annetaan erityinen aloitussolmupiste s, niin algoritmi löytää lyhyimmän reitin aloituspiste s:stä jokaiseen eri solmupisteeseen kaaviossa mukaan lukien haluttu määränpää. (Skiena 2008, 206.)

Dijkstran-algoritmi ratkaisee yhdestä lähteestä lyhyintä reittiä ongelmia painotetulla kohdistetulla graafilla $G = (V,E)$. Tässä tapauksessa kaikki reunapainot ovat ei-negatiivisia. Siten tässä tapauksessa oletetaan, että $w(u, v) \geq 0$ jokaista reunaa $(u, v) \in E$. Hyvällä täytäntöönpanolla Dijkstran-algoritmi on hyvin nopea. (Cormen ym. 2009, 658.)

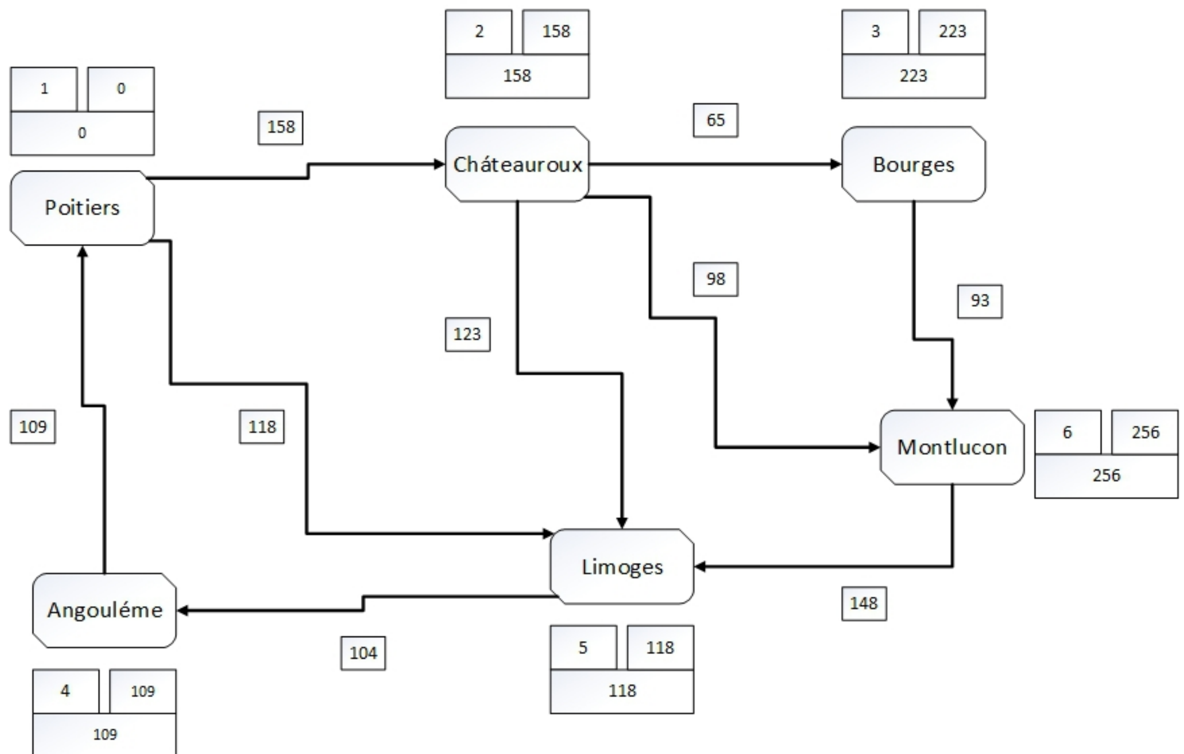
Dijkstra-algoritmi etenee kierroksittain. Jokainen kierros laatii lyhimmän reitin s:stä johonkin uuteen kärkeen. Erityisesti x on kärki, joka minimoi $dist(s, v_i) + w(v_i, x)$ kaikkien keskeneräisten $1 \leq i \leq n$, jossa $w(i, j)$ on reunan pituus i:stä j:hin ja $dist(i,j)$ on lyhimmän reitin pituus niiden välillä. (Skiena 2008, 207.)

Pseudokoodi dijkstran-algoritmille:

Algoritmi lyhin reitti Dijkstra (G, s):
Sisääntulo: Painotettu graafi G ja tästä aloitussolmu s.
Ulostulo: Jokaiselle solmulle v arvo D[v], missä D[v] on lyhimmän polun pituus solmusta s solmuun v graafissa.
Alustetaan arvot D[s] = 0 ja D[v] = ∞ jokaiselle solmulle v ≠ s.
Olkoon Q prioriteettijono, joka käsittää kaikki graafin G solmut ja jossa avaimina ovat arvot D.
While Q ≠ 0 do
    {valitaan solmu u pilveen}
    u = arvo palautetaan funktiolla Q.remove_min()
    for jokaiselle u:n vierekkäiselle solmulle v prioriteettijonosta Q do
    {suoritetaan relaksaatio-operaatio reunalle (u,v)}
    if D[u] + w(u,v) < D[v] then
        D[v] = D[u] + w(u,v)
        vaihdetaan v:n avainarvo Q:sta uudeksi arvoksi D[v]
return jokaisen solmun v arvo D[v]

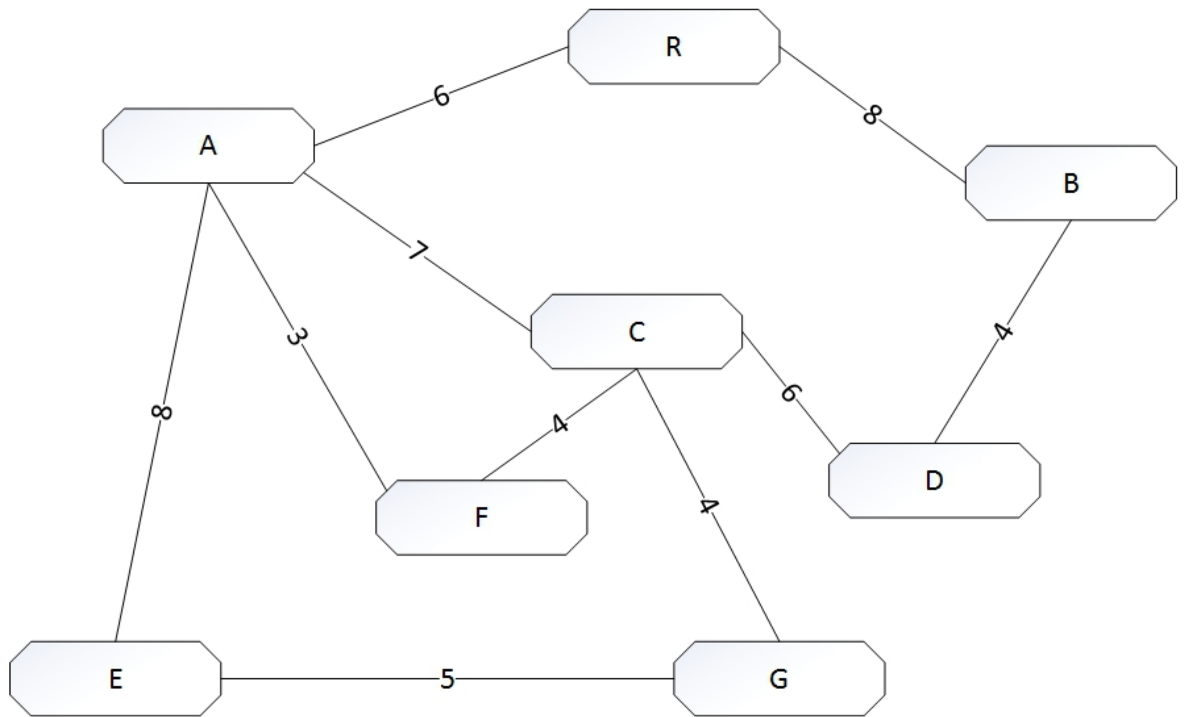
(mukaillen Goodrich ym. 2013, 662.)

Dijsktran-algoritmin manuaalinen simulointi onnistuu esimerkiksi vuokaaviolla (KUVIO 15). Vuokaavio on tehty ranskan kartasta (KUVIO 13). Vuokaaviossa simuloidaan lyhintä reittiä kaupungista Poitiers paikkaan Montlucoon. Simulointi tapahtuu siten, että katsotaan ja lasketaan, mistä menee lyhin reitti mihinkin, ja ne merkataan kaupunkien yläpuolella oleviin laatikkoihin. (KUVIO 15.) Simuloinnissa pitää ottaa huomioon jokainen kaupunki ja niiden välimatkat.



KUVIO 15. Lyhin reitti paikasta Poitiers paikkaan Montlucon

Kuviosta (KUVIO 16) tehdään taulukko muodossa (TAULUKKO 7) simulointi, josta nähdään, mikä on lyhin reitti R solmusta G solmuun. Taulukko näyttää toisen tavan Dijkstran manuaaliseen simulointiin. Simuloinissa katsotaan ensimmäiseksi solmua R ja sen naapureita A- ja B-solmuja. Katselun jälkeen lisätään päivitykset ja laitetaan kohdat jonoon.



KUVIO 16. Solmukuvio

TAULUKKO 7. Taulukko, joka on tehty kuvion 16 pohjalta

kierros numero	solmu	naapurit	päivitykset	jono
1	R	A,B	A(totta, 6, R), B(totta, 8, R)	A(totta, 6 R), B(totta, 8 R)
2	A	E,F,C,R	E(totta, 14, A)	B(totta, 8 R), E(totta, 14, A)
			F(totta, 9, A)	B(totta, 8 R), F(totta, 9, A), E(totta, 14, A),
			C(totta, 13, A) R ei muutoksia	B(totta, 8 R), E(totta, 14, A), F(totta, 9, A), C(totta, 13, A)
3	B	R,D	R ei muutoksia, D(totta, 12, D)	D(totta, 12, D), E(totta, 14, A), C(totta, 13, A)
4	F	A,C	A ei muutoksia, C(totta, 11, F)	D(totta, 12, D), E(totta, 14, A), C(totta, 11, F)
5	D	B,C	B ei muutoksia, C ei muutoksia	E(totta, 14, A), C(totta, 11, F)
6	E	A,G	A ei muutoksia, G(totta, 19, E)	C(totta, 11, F), G(totta, 19, E)
7	C	A,F,G,D	A ei muutoksia, F ei muutoksia, D ei muutoksia, G(totta, 17, C)	G(totta, 17, C)
8	G	E,C	E ei muutoksia, C ei muutoksia	tyhjä OK

Dijkstran-algoritmin simulointiohjelmalla. Ohjelma on tehty c++ kielellä.

```
#include<iostream>
#define INF 999

 using namespace std;

  int adjMatrix[6][6]
 = {0,158,INF,INF,118,109,
                    158,0,65,98,123,INF,
                    INF,65,0,93,INF,INF,
                    INF,98,93,0,83,INF,
                    118,123,INF,148,0,104,
                    109,INF,INF,INF,104,0};

  int source = 0;
  int numOfVertices=6;

class Graph
 {
 private:
 int predecessor[6],distance[6];
```

```
 bool mark[6];

 public:
 void read();
 void initialize();
 int getClosestUnmarkedNode();
 void dijkstra();
 void output();
 void printPath(int);
 };

void Graph::read()
 {
 cin>>numOfVertices;
//Solmujen määrä pitäisi aina olla enemän kuin nolla
 while(numOfVertices <= 0) {
 cin>>numOfVertices;
 }
  //Luetaan graafin vierekkäiset matriisit
for(int i=0;i<numOfVertices;i++) {
 for(int j=0;j<numOfVertices;j++) {
 cin>>adjMatrix[i][j];
 while(adjMatrix[i][j]<0) {
 cin>>adjMatrix[i][j];
 }
 }
 }
// Luetaan lähde solmu, josta pitää olla lyhimmät reitit löydetty
cin>>source;
 while((source<0) && (source>numOfVertices-1)) {
cin>>source;
 }
 }
int Graph::getClosestUnmarkedNode()
 {
 int minDistance = INF;
 int closestUnmarkedNode;
 for(int i=0;i<numOfVertices;i++) {
 if((!mark[i]) && ( minDistance >= distance[i])) {
 minDistance = distance[i];
 closestUnmarkedNode = i;
 }
 }
 return closestUnmarkedNode;
 }

void Graph::dijkstra()
 {
 initialize();
 int minDistance = INF;
```

```
int closestUnmarkedNode;
int count = 0;
while(count < numOfVertices) {
closestUnmarkedNode = getClosestUnmarkedNode();
mark[closestUnmarkedNode] = true;
for(int i=0;i<numOfVertices;i++) {
if((!mark[i]) && (adjMatrix[closestUnmarkedNode][i]>0) ) {
if(distance[i] > distance[closestUnmarkedNode]+adjMatrix[closestUnmarkedNode][i]) {
distance[i] = distance[closestUnmarkedNode]+adjMatrix[closestUnmarkedNode][i];
predecessor[i] = closestUnmarkedNode;
}
}
}
count++;
}
}

void Graph::printPath(int node)
{
if(node == source)
cout<<node<<"..";
else if(predecessor[node] == -1)
cout<<"No path from "<<source<<" to "<<node<<endl;
else {
printPath(predecessor[node]);
cout<<node<<"..";
}
}

void Graph::output()
{
for(int i=0;i<numOfVertices;i++) {
if(i == source)
cout<<source<<".."<<source;
else
printPath(i);

cout<<" -> "<<distance[i]<<endl;
}
}


void Graph::initialize()
{
for(int i=0;i<numOfVertices;i++) {
mark[i] = false;
predecessor[i] = -1;
distance[i] = INF;
}
distance[source] = 0;
```

```
}

int main()
 {
 Graph G;

 G.dijkstra();
 G.output();

 cin.get(); cin.get();
 return 0;
 }
```

(mukaillen Heineman ym. 2009, 164–165.)

6.3 A*-algoritmi

A*-algoritmi saattaa olla vaikea ymmärtää aloittelijalle. Mutta tarpeeksi asiaan tutustumalla pääsee hyvin jyvälle A*-algoritmin saloista (Lester 2005). Agenttien liikkuminen on yksi suurimpia haasteita, kun suunnitellaan realistista tekoälyä tietokonepelissä. Tämä haaste johtuu siitä, että nykyaikaiset pelit ovat enemmän ja enemmän dynaamisia luonteeltaan. Kaksi peruskomponenttia reaaliaikaiselle reitinhaulle ovat maalin suuntaan meneminen ja staattisten ja dynaamisten esteiden välttäminen realiajassa. (Graham, McCabe & Sheridan 2014, 1–2.)

A*-algoritmi toimii kolmella asialla: avoimella listalla, suljetulla listalla ja heuristiikalla. Avoimella listalla tarkoitetaan solmuja, joita pitää kokeilla, ja suljetulla listalla tarkoitetaan solmuja, joita ei tarvitse testata. Tässä tapauksessa solmulla tarkoitetaan yksinkertaisesti tietoa tietyillä tiilillä. Jokaisella tiilillä voi olla oma solmu, mutta se on erittäin epätodennäköistä, että jouduttaisiin testaamaan jokaista tiiliä kartalla. (Hatfield 2000.)

Heuristiikka on avain, jolla tehdään A*-algoritmista tehokas. Sillä tarkoitetaan arvioitua matkaa mistä tahansa tiilistä määränpäähän. Tässä tapauksessa määränpää on maalisolmu. Mitä lähempänä heuristiikka on todellista matkaa, sitä parempi reitti on. (Hartfield 2000.)

7 POHDINTA

Työn tavoitteena oli tutkia ja opiskella perusalgoritmeja ja reitinhakualgoritmeja. Perusalgoritmeista tavoitteena oli yleisellä tasolla kirjoittaa ja ottaa muutamia esimerkkejä. Reitinhakualgoritmeista tavoitteena oli tutkia yleisesti graafeja, syvyys ensin -hakua ja leveys ensin -hakua. Reitinhakualgoritmeista tarkempana tavoitteena oli lyhimmän reitinhakua. Lyhimmän reitinhakualgoritmeista tavoitteena oli tutkia Dijkstran-algoritmia ja Floyd-Warshall-algoritmia, sekä tarkastella A*-algoritmia.

Käytännön sovelluksena esiteltiin Dijkstran-algoritmin ja Floyd-Warshall-algoritmin manuaalista simulointia. Molempia algoritmien simulointi osoitti, että molemmat algoritmit löytävät kohtuullisen tehokkaasti lyhimmän reitin kohteeseen. Toisena käytännön sovelluksena käytettiin C++ ohjelmaa. C++ ohjelmalla testattiin, että manuaaliset simuloinnit ovat oikein. Valmiina olevasta materiaalista huolimatta algoritmien simulointi ja niiden graafien tekeminen voi olla hyvinkin haastavaa.

Kehittämisideana toimisi, että tämän työn pohjalta tekisin opetusmateriaalin algoritmeille. Tämä tarkoittaisi sitä, että kaikkia osa-alueita pitäisi laajentaa. Opetusmateriaalin tekeminen työstä tarkoittaisi sitä, että esimerkkien määrää pitäisi lisätä huomattavasti. Toisena kehittämisideana olisi, että työhön laitettaisiin enemmän ja yksityiskohtaisempaa tietoa A*-algoritmista.

LÄHTEET

BonjourLaFrance. 2014. Www-dokumentti. Saatavissa: http://www.bonjourlafrance.com/france-map/france_driving_distance_cities.htm. Luettu 21.4.2014.

Cormen, T., Leiserson, C., Rivest, R. & Stein, C. 2009. Introduction to Algorithms Third Edition. London. The MIT Press.

Dasgupta, S., Papadimitriou, C. H. & Vazirani U. V. 2006. Algorithms. Pdf-dokumentti. Saatavissa: http://www.cs.berkeley.edu/~vazirani/algorithms/chap2.pdf. Luettu 30.4.2014.

Delling, D., Sanders, P., Schultes, D. & Wagner D. 2009. Engineering Route Planning Algorithms. Karlsruhe. Universität Karlsruhe.

Goodrich, M., Tamassia, R. & Goldwasser, M. 2013. Data Structures and Algorithms in Python. USA. Wiley.

Graham, R., McCabe, H. & Sheridan, S. Neural Networks for Real-time Pathfinding in Computer Games. 1–2. Www-dokumentti. Saatavissa: http://www.gamesitb.com/nnpathgraham.pdf. Luettu 17.4.2014.

Granville, B. & Luca Del, T. 2008. Data Structures and Algorithms: Annotated Reference with Examples. Queensland. DotNetSlackers.

Hatfield, T. 2000. Short Description of A*. Www-dokumentti. Saatavissa: http://www-cs-students.stanford.edu/~amitp/Articles/AStar5.html. Luettu 17.4.2014.

Heineman, T., Pollice, G. & Selkow, S. 2009. Algorithms in a Nutshell. Sebastopol. O'Reilly.

Hetland, M. 2010. Python Algorithms Mastering Basic Algorithms in the Python Language. New York. Apress.

IBM Data Processing Techniques. 1970. IBM 1970.

Kolehmainen, K. 2006. Algoritmit ja mallit. Kokkola. Readme.fi.

Lester, P. 2005. A* Pathfinding for Beginners. Www-dokumentti. Saatavissa: http://www.policyalmanac.org/games/aStarTutorial.htm. Luettu 17.4.2014.

Sedgewick, R. & Wayne, K. 2011. Algorithms. Boston. Addison-Wesley.

Skiena, S. 2008. The Algorithm Design Manual. New York. Springer.

Soltys, M. 2012. An Introduction to the analysis of algorithms. Singapore. World Scientific.

Wirth, N. 2004. Algorithms and Data Structures. Www-dokumentti. Saatavissa: http://www.inf.ethz.ch/personal/wirth/AD.pdf. Luettu 2.4.2014.