# Lin-Kernighan-algoritmi kauppamatkustajan ongelmaan

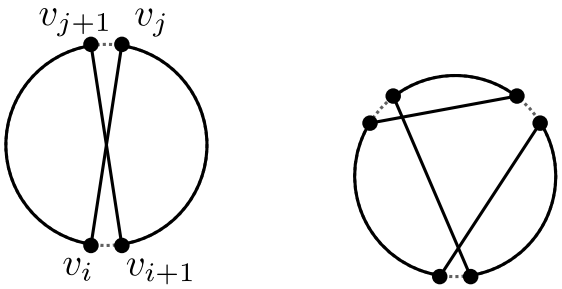
Teppo Niinimäki

Helsinki 26.2.2009

Diskreetti optimointi -kurssin raportti

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

# Sisältö




(a) (b)

Kuva 1: (a) 2-vaihto. (b) Eräs 3-vaihto.

# 1 Tehtävän kuvaus

Kauppamatkustajan ongelmassa on päämääränä löytää annetun suuntaamattoman verkon lyhin Hamiltonin kierros, eli reitti joka kulkee jokaisen solmun kautta täsmälleen kerran ja päätyy takaisin lähtösolmuun. Lyhimmän reitin etsiminen on NP-täydellinen ongelma, mutta on olemassa useita heuristiikkoja, joilla päästään lähelle optimia. Toteutin Lin-Kernighan-algoritmin, joka pyrkii löytämään mahdollisimman hyvän ratkaisun. Algoritmi aloittaa jostain validista, esimerkiksi satunnaisesti arvotusta kierroksesta, ja parantaa sitä askel kerrallaan tekemällä paikallisia muutoksia reittiin. Se ei siis yleensä löydä parasta ratkaisua, mutta päätyy paikalliseen optimiin. Toteutuksessa seurasin Applegaten ja kumppaneiden raporttia [ABCC99]. Seuraavaksi on kuvattu algoritmin toiminta pintapuolisesti. Tarkemmat yksityiskohdat löytyvät kyseisestä raportista.

## 1.1 Vaihdot ja flip-operaatio

Olkoon $v_0, \ldots, v_{n-1}$ jokin suunnattu kierros $n$-solmuisessa suuntaamattomassa verkossa (merkitään tästä lähtien $v_i = v_j$ kaikilla $i \equiv j \bmod n$). Olkoon lisäksi $c(v_i, v_j)$ solmujen $v_i$ ja $v_j$ välisen kaaren pituus. Nyt jos korvataan reitin kaaret $v_iv_{i+1}$ ja $v_jv_{j+1}$ (missä $v_i$:n ja $v_j$:n etäisyys tarpeeksi suuri) kaarilla $v_iv_j$ ja $v_{i+1}v_{j+1}$, saadaan uusi validi kierros. Jos lisäksi lisättyjen kaarien yhteenlaskettu pituus $c(v_i, v_j)+c(v_{i+1}, v_{j+1})$ on pienempi kuin poistettujen kaarien pituus $c(v_i, v_{i+1}) + c(v_j, v_{j+1})$, on uusi kierros aiempaa lyhyempi. Kutsutaan tällaista kahden kaaren korvaamista uusilla ristiin menevillä kaarilla *2-vaihdoksi*. Esimerkin 2-vaihto on esitetty kuvassa 1a.

Kuvattu 2-vaihto voidaan saadaan aikaan, kun käännetään reitin osan $v_{i+1}, \ldots, v_j$ suunta päinvastaiseksi. Tällöin saadaan siis reitti $v_0, \ldots, v_i, v_j, \ldots, v_{i+1}, v_{j+1}, \ldots, v_{n-1}$.

Merkitään tätä vaihtoa operaatiolla $flip(v_{i+1}, v_j)$.

Vastaavalla tavalla voidaan vaihtaa myös korvata useampia kaaria toisilla kaarilla. Määritellään siis vastaavalla tavalla *3-vaihto* ja *4-vaihto*, joissa korvataan reitin kolme tai neljä kaarta. Nämä voidaan tehdä suorittamalla useita $flip$-operaatioita peräkkäin. Esimerkki 3-vaihdosta on kuvassa 1b.

Määritellään lisäksi operaatiot *next* ja *prev* siten että $next(v_i) = v_{i+1}$ ja $prev(v_i) = v_{i-1}$ kaikilla $i$ (kun $v_0, \ldots, v_{n-1}$ on käsiteltävä kierros). Siis *next* palauttaa solmun seuraajan sen hetkisellä kierroksella ja vastaavasti *prev* palauttaa solmun edeltäjän.

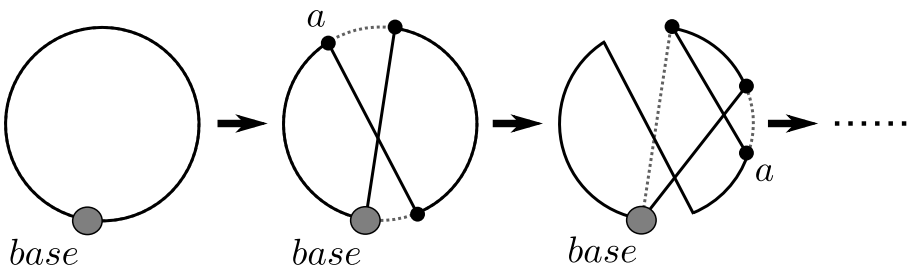
## 1.2 LK-haku

Olkoon *base* jokin valittu reitin solmu. *LK-haku* solmusta *base* toimii seuraavasti:

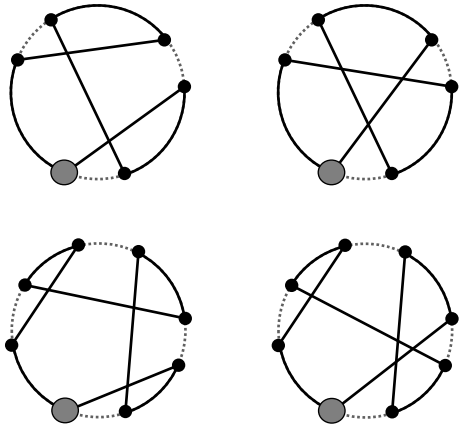
1. Valitse solmun $next(base)$ naapurustosta solmut $a$, joille pätee $delta + c(base, next(base)) - c(next(base), a) > 0$.
2. Valitse näistä yksi tai useampi solmu $a$, joille $c(prev(a), a) - c(next(base), a)$ on mahdollisimman suuri.
3. Kullekin valitulle solmulle $a$:
   (a) Aseta $delta' = delta + c(base, next(base)) - c(next(base), a) + c(prev(a), a) - c(base, prev(a))$.
   (b) Suorita 2-vaihto: $flip(next(base), prev(a))$.
   (c) Jos löydettiin parempi reitti ($delta' > 0$), hyväksi suoritettu vaihtosarja ja lopeta.
   (d) Kutsu itseä rekursiivisesti samalla *base*-solmulla ja arvolla $delta = delta'$.
   (e) Peru suoritettu 2-vaihto.

LK-haussa siis etsitään sellainen sarja *base*-solmun vieressä tehtäviä 2-vaihtoja, että reittiä saadaan lyhennettyä. Muuttujassa *delta* pidetään aina kullakin hetkellä saavutettua parannusta reitin pituudessa. Reitti voi siis huonontua väliaikaisesti ($delta < 0$), kunhan lopulta päädytään parempaan reittiin. Kuvassa 2 on esimerkki osittaisesta vaihtosarjasta.

LK-haun haarautumisaste voidaan asettaa erikseen kullekin rekursiosyvyydelle. Lisäksi rekursiosyvyyttä voidaan rajoittaa, jotta haku saadaan pysähtymään. Toinen




Kuva 2: LK-haku etenee suorittamalla sarjan 2-vaihtoja *base*-solmun vieressä.



Kuva 3: Vaihtoehtoiset LK-haun aloitustavat.

vaihtoehto syvyyden rajoittamiseen olisi estää samojen kaarten toistuva poistaminen ja/tai lisääminen.

## 1.3 Vaihtoehtoinen aloitus

LK-haku voidaan suorittaa myös tekemällä aluksi sopiva 3-vaihto tai 4-vaihto *base*-solmun vierestä ja jatkamalla tämän jälkeen normaalisti 2-vaihdoilla. Kuvassa 3 on esitetty nämä vaihtoehtoiset tavat aloittaa haku. Vaihtoehtoaloituksessa täytyy pystyä tarkistamaan, sijaitseeko annettu solmu $x$ solmujen $y$ ja $z$ välissä nykyisellä reitillä. Tähän käytetään operaatiota $sequence(y, x, z)$.

## 1.4 Lin-Kernighan-algoritmi

Lin-Kernighan-algoritmissa suoritetaan LK-hakua toistamiseen käyttäen reitin eri solmuja *base*-solmuina. Jos normaali LK-haku ei löydä parempaa reittiä, kokeillaan

vaihtoehtoista aloitusta. Aina kun solmu on *base*-solmuna, se asetetaan kieltolistalle. Listalla olevia solmuja ei saa käyttää *base*-solmuina. Toisaalta aina kun LK-haku onnistuu löytämään paremman reitin, poistetaan kaikki solmut kieltolistalta. Hakuja jatketaan, kunnes kaikki solmut ovat kieltolistalla. Saatu reitti on paikallinen optimi.

Algoritmia voidaan toistaa erilaisilla satunnaisilla aloitusreiteillä. Toinen vaihtoehto olisi iteroida algoritmia siten, että saatua reittiä muutettaisiin aina hieman iteraatioiden välissä.

# 2 Toteutuksen kuvaus

## 2.1 Ympäristö ja testiaineisto

Toteutin algoritmin C-kielellä. Standardikirjastojen lisäksi kääntämiseen vaaditaan getopt, SDL ja OpenGL. Ohjelma kääntyy ainakin gcc-kääntäjällä. Käännetyn ohjelman ajamiseksi täytyy koneella olla asennettuna SDL ja OpenGL. Kaikki tarvittava löytyy valmiiksi tietojenkäsittelytieteen laitoksen Linux-järjestelmästä. Testiaineistona käytin TSBLIB-kokoelman [Rei] ongelmia.

## 2.2 Tiedostot

Toteutus on jaoteltu tiedostoihin seuraavasti:

**lin_kernighan.c**
: Lin-Kernighan-algoritmin pääosat

**main.c**
: Pääohjelma

**Makefile**
: Kääntämiseen tarvittavat tiedot

**queue.c**
: Jonototeutus kokonaisluvuille

**tsp_loader.c**
: TSPLIB-kokoelman ongelmien lukemiseen tarvittavat funktiot

**visualize.c**
: Graafinen visualisointi

Lisäksi useimpiin c-tiedostoihin liittyy vastaava otsikkotiedosto, jossa on esitelty tiedoston tarjoamat funktiot.

## 2.3 Toteutus

Algoritmin pääfunktiona toimii *linKernighan*. Se vastaa Applegaten ja kumppaneiden raportin algoritmia 2.4. Funktio pitää taulukossa tietoa kielletyistä solmuista. Lisäksi käytössä on jono, joka sisältää solmut, joita on sallittua käyttää *base*-solmuna. Tässä tapauksessa parempi ratkaisu olisi varmaankin ollut pitää myös kielletyt solmut jonkinlaisessa jonossa, pinossa tai listassa. Nykyistä ratkaisua käytettäessä on kuitenkin helpompi tarvittaessa muuttaa algoritmia niin, että LK-haun onnistuessa kieltolistalta poistettaisiinkin vain tietyllä tavalla valitut solmut. Näitä tapoja on esitelty raportissa.

Pääfunktio *linKernighan* kutsuu toistuvasti *lkSearch*-funktiota, joka vastaa raportin algoritmia 2.3. Funktio *lkSearch* suorittaa edellä käsitellyn LK-haun. Se kutsuu funktiota *lkStep* sekä tarvittaessa funktiota *lkAltStep*.

Funktio *lkStep* vastaa raportin algoritmia 2.1 ja suorittaa LK-haun yhden rekursioaskeleen. Raportin algoritmista poiketen se ei kuitenkaan suorita Mak-Morton-siirtoja, jotka olisivat vaihtoehtoinen tapa suorittaa 2-vaihto. Se myös lopettaa suorittamisen heti, jos parempi reitti löydetään. Raportissa lopettamisehtoa ei mielestäni mainittu suoraan, mutta ilmeisesti perusversio jatkaa maksimaaliseen rekursiosyvyyteen saakka. Toinenkin vaihtoehto lopettamisehdoksi raportissa esitetään.

Funktio *lkAltStep* vastaa raportin algoritmia 2.2 ja suorittaa siten vaihtoehtoisen aloituksen LK-haulle. Vaihdot (3-vaihdot ja 4-vaihdot) suoritetaan kuitenkin erilaisilla $flip$-operaatioiden sarjoilla kuin raportissa on esitetty. Tämä johtuu siitä, että raportin esittämä tapa ei mielestäni tuota oikeaa lopputulosta (siis ainakaan sellaista kuin raportissa annettiin ymmärtää). Funktio jatkaa hakua normaalisti kutsumalla funktiota *lkStep*.

LK-haun haarautumisastetta rajoitetaan *breadth*-taulukon arvoilla (sekä muuttujilla *altBreadthA*, *altBreadthB* ja *altBreadthC*. Vain muutamalla ensimmäisellä askeleella rekursio todella haarautuu; lopuilla haarautumisaste on yksi. Haun maksimisyvyyttä rajoitetaan asettamalla haarautumisaste nollaksi taulukon lopussa.

Mahdolliset ehdokkaat solmuksi $a$ valitaan solmun $next(base)$ muutaman lähimmän naapurin joukosta. Naapurilistat kullekin solmulle lasketaan ennalta ja annetaan algoritmille parametriksi. Näin listojen muodostamiseen olisi mahdollista käyttää eri tyyppisiä heuristiikkoja. Toteutuksessani listat koostetaan lähimmistä naapureista. Parhaiden naapuriehdokkaiden järjestämiseen ja valintaan käytetään *CandidateList*-tietorakennetta.

Toteutuksessani reittiä pidetään yllä Route-tietorakenteessa. Taulukon *order* $i$. alkiossa on tallennettuna reitin $i$. solmun numero. Lisäksi käytössä on *position*-taulukko, joka sisältää order-taulukon käänteispermutaation. Näiden avulla operaatiot *next*, *prev* ja *sequence* voidaan suorittaa vakioajassa. Sen sijaan $flip(x, y)$-operaatiossa joudutaan käsittelemään kaikki reitin solmut väliltä $x, \ldots, y$, joten se on pahimmassa tapauksessa lineaarinen. Tämän lievittämiseksi käytössä on reverse-bitti, jonka ansiosta voidaankin halutessa kääntää reitin osa $next(y), \ldots, prev(x)$ ja vaihtaa reverse-bitillä koko reitin suunta päinvastaiseksi. Näin käsiteltävien solmujen määrä saadaan rajoitettua korkeintaan puoleen reitin pituudesta.

Aloituskierroksena algoritmille annetaan satunnainen kierros. Oletuksena algoritmi suoritetaan kerran ja saatu tulos palautetaan. Algoritmia on myös mahdollista iteroida toistuvasti eri satunnaisista lähtökohdista, jolloin valitaan paras saavutettu reitti.

## 2.4 Puutteet, rajoitukset ja parannusehdotukset

Algoritmi itsessään tuntuisi toimivan oikein ja pääsee yleensä muutaman prosentin päähän optimista satunnaisella aloituskierroksella. Toisaalta algoritmin saavuttamissa ratkaisuissa on usein helposti havaittavia parannuksia erityisesti kohdissa, joissa on pitkiä toisten kaarten poikki kulkevia kaaria. Esimerkiksi seuraavia asioita voisi tehdä toiminnan parantamiseksi:

- Samojen kaarien toistuvan lisäämisen/poistamisen estäminen *lkStep*-funktioon.
- Mak-Morton-siirtojen lisääminen *lkStep*-funktioon.
- Haun jatkaminen, vaikka parempi reitti on jo löydetty. Tähän liittyen myös lähderaportissa esitetty mahdollinen lisäehto jatkamiselle.
- Reitin tallentaminen parempaa tietorakenteeseen, esim. monikerroksiseen linkitettyyn listaan tai puuhun. Näin päästäisiin *flip*-operaation osalta asymptoottisesti parempaan aikavaativuuteen.

- Muutenkin nopeampien tietorakenteiden käyttö joissakin paikoissa.
- Iteroiminen siten, että algoritmia sovelletaan toistuvasti parhaan saavutetun reitin muunnelmiin.
- Muiden kuin satunnaisten reittien käyttäminen optimoinnin lähtökohtana.

Itse algoritmin lisäksi myös muita ohjelman osia voisi kehittää. TSPLIBin ongelmatyypeistä ohjelma tukee tyyppejä EUC_2D, EUC_3D, MAX_2D, MAX_3D, MAN_2D, MAN_3D, CEIL_2D ja GEO. Näistäkin vain osan toimintaa on testattu. Kun ongelma ladataan, lasketaan kaikkien kaarten painot valmiiksi taulukkoon. Tästä johtuva neliöllinen muistinkulutus estää tarpeeksi suurten instanssien käsittelemisen. Laskenta olisikin järkevää muuttaa vain tarvittaessa suoritettavaksi.

Visualisointisysteemiinkin voisi lisätä tuen 3-ulotteisille ongelmille ja eri kaarien piirtotavat erilaisille etäisyysfunktioille. Lisäksi käyttöliittymää olisi hyvä parantaa ja lisätä tarkempia tietoja antavia status-tekstejä kuvaan.

# 3 Käyttökokemuksia ja tuloksia

Yleisesti toteuttamani algoritmi tuntuisi yhdellä iteraatiolla pääsevän muutaman prosentin päähän optimista. Taulukoissa 1 ja 2 on joitakin testituloksia. Reittien pituudet on normalisoitu jakamalla ne optimireitin pituudella.

Taulukko 1: Löydetyn reitin pituus optimiin verrattuna eri ongelmilla (10 yritystä, kullakin yksi iteraatio)

| Ongelma | Lyhin reitti | Keskipituus | Pisin reitti | Keskimäärin kulutettu aika |
|---|---|---|---|---|
| tsp225 | 1.0120 | 1.0248 | 1.0506 | 0.11 |
| dsj1000 | 1.0677 | 1.0962 | 1.1373 | 0.79 |
| rl5915 | 1.1285 | 1.2137 | 1.2654 | 29.118 |

# 4 Työn kulku

Aikaa kului suurin piirtein seuraavasti eri harjoitustyön osa-alueisiin. Ajat ovat arvioita.

Taulukko 2: Löydetyn reitin pituus optimiin verrattuna eri aikarajoilla ongelma-instanssilla *dsj1000*.

| Aikaraja | Reitin pituus | Iteraatioiden määrä |
|---|---|---|
| 1 s | 1.0971 | 2 |
| 10 s | 1.0367 | 13 |
| 100 s | 1.0213 | 112 |

| Tehtävä | Aika tunteina |
|---|---|
| Aiheisiin tutustuminen ja aiheen valinta | 3,5 |
| Artikkelien tarkempi lukeminen | 4 |
| TSBLIB-parserin toteutus | 5 |
| Algoritmin ja tietorakenteiden toteutus | 11 |
| Visualisoinnin toteutus | 3 |
| Muu ohjelmointi, koodin järjestely ja kommentointi | 5 |
| Testaaminen | 1,5 |
| Esitelmän valmistelu | 2,5 |
| Raportin kirjoittaminen | 6.5 |
| Tapaamiset ja esitelmä | 6 |
| Yhteensä | 48 |

Työ sujui ilman mitään ylitsepääsemättömiä vaikeuksia. C-kielen valinta teki joidenkin osien toteutuksesta tarpeettoman monimutkaista. Pääasiallisena lähteenä käyttämäni Applegaten ja kumppaneiden raportti oli osittain hieman vaikeaselkoinen. Johnsonin ja McGeochin artikkeli[JM95], johon myös osittain tutustuin, oli toisaalta ymmärrettävämmän oloinen, mutta jätti enemmän yksityiskohtia avoimeksi.

# Lähteet

ABCC99 Applegate, D., Bixby, R., Chv'atal, V. ja Cook, W., Finding tours in the tsp. Tekninen raportti 99885, Institute for Discrete Mathematics, Universitat Bonn, 1999.

JM95 Johnson, D. S. ja McGeoch, L. A., The traveling salesman problem: A case study in local optimization, `http://www.cs.ubc.ca/~hutter/previous-earg/EmpAlgReadingGroup/TSP-JohMcg97.pdf`, 1995.

Rei Reinelt, G., TSBLIB, http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.