Pisin yhteinen alijono – onko kaksisuuntaisesta lähestymistavasta hyötyä käytännössä?

TURUN YLIOPISTO
Informaatioteknologian laitos
Pro Gradu -tutkielma
Tietojenkäsittelytiede
Tero Yli-Sipilä
Toukokuu 2015

TURUN YLIOPISTO
Informaatioteknologian laitos
Matemaattis-luonnontieteellinen tiedekunta

YLI-SIPILÄ, TERO: Pisin yhteinen alijono – onko kaksisuuntaisesta lähestymistavasta hyötyä käytännössä?

Pro Gradu -tutkielma, 61 s., 46 liites.
Tietojenkäsittelytiede
Toukokuu 2015

Pisin yhteinen alijono (PYA) tarkoittaa pisintä mahdollista yhteistä merkkijonoa, jonka merkit löytyvät samassa järjestyksessä kaikista tarkasteltavista merkkijonoista. Tässä tutkielmassa tarkastellaan PYAn määräämiseen kehitettyjä menetelmiä, varhaisempia algoritmeja sekä uudempia ja kehittyneempiä menetelmiä. Työn kokeellisessa osassa testataan, voidaanko vanhoja menetelmiä nopeuttaa uusissa menetelmissä käytetyillä keinoilla kuten merkkijonojen kaksisuuntaisella analysoinnilla. Vanhat menetelmät ovat pääsääntöisesti yksinkertaisempia kuin uudet, ja niiden vaatimat tietorakenteet eivät ole yleensä niin monimutkaisia.

Osa tutkielman algoritmeista analysoi merkkijonoja vain yhteen suuntaan, jotkut taas toimivat jo valmiiksi kaksisuuntaisesti. Osasta yhteen suuntaan etenevistä menetelmistä on tehty tätä tutkielmaa varten kaksisuuntainen versio. Tutkielmassa selvitetään algoritmien toimintaperiaatteet ja verrataan samankaltaisten lähestymistapojen yksi- ja kaksisuuntaisia versioita. Vertailujen perusteella selvitetään kaksisuuntaisuuden hyötyjä ja haittoja. Suoritettavilla testeillä selvitetään, miten uusien menetelmien monimutkaiset laskentatavat ja muistirakenteet vaikuttavat suorituskykyyn ja muistinkäyttöön vanhoihin ja suoraviivaisempiin menetelmiin verrattaessa.

Testeissä havaittiin, että kaksisuuntaisesta lähestymistavasta ei saada merkittäviä suorituskykyetuja ainakaan vain yhtä suoritinydintä käyttämällä. Pikemminkin päinvastoin, kaksisuuntainen lähestymistapa vaati osassa algoritmeista huomattavasti enemmän muistia sekä suoritusaikaa kuin vastaavalla periaatteella yhteen suuntaan prosessointi. Vain Hirschbergin lineaarisessa menetelmässä kaksisuuntaisuudesta oli selkeää hyötyä muistinkäytössä, kun rekursiivisen lähestymistavan vuoksi siinä ei tarvita suuria suorasaanti- ja tulostaulukoita. Se hyötyi kaksisuuntaisuudesta hieman myös suoritusajassa, mutta ero suoritusajoissa ei ollut niin merkittävä kuin muistinkäytössä. Testeissä huomattiin myös, että vanha ja yksinkertainen algoritmi ei ole aina hitaampi kuin uusi ja monimutkainen.

Asiasanat: pisin yhteinen alijono, merkkijonoalgoritmi, kaksisuuntaisuus, vertailu

UNIVERSITY OF TURKU
Department of Information Technology
Faculty of Mathematics and Natural Sciences

YLI-SIPILÄ, TERO: Longest common subsequence – is a bidirectional approach advantageous in practice?

Master's Thesis, 61 p., 46 app. p.
Computer Science
May 2015

Longest common subsequence (LCS) means the longest possible common string, whose characters can be found in the same order from the strings that are under the consideration. In this thesis, some LCS algorithms are examined, of which some are initial and others are more recent and advanced. In the practical part of the thesis it is tested, if it is possible to speed up the old methods with the techniques used in recent methods, such as bidirectional processing of strings. The old methods are mainly simplier than the newer ones and they don't usually require so complicated data structures.

Some of the algorithms of the thesis analyse strings only in a unidirectional way while others use already a bidirectional processing. Some of the unidirectional algorithms have been converted also to bidirectional algorithms for this thesis. In the thesis, operating principles of the algorithms are explained and unidirectional and bidirectional versions of similar methods are compared. With comparisons it is tried to sort out the benefits and drawbacks of the bidirectional processing. With performed tests it is examined, how the complex data structures and computing affect the performance and memory usage in comparison with older and more straightforward methods.

In the tests it was noticed that the bidirectional processing doesn't have significant benefits in performance, at least with using only a single core of a processor. Rather vice versa, the bidirectional processing required among some algorithms significantly more memory resources and processing time than a comparable unidirectional algorithm. Only in Hirschberg's linear method the bidirectional processing had obvious benefits in memory usage because its recursive approach doesn't require big direct access and result tables. It also got benefits from the bidirectional processing in execution time, but the differences of the performance times were not so big than in memory usage. In the tests it was also noticed that an old and simple algorithm is not always slower than a newer and more complicated algorithm.

Keywords: longest common subsequence, string algorithm, bidirectional, comparison

Sisällys

1. Johdanto

Tässä työssä tutkitaan pisimmän yhteisen alijonon (myöhemmin lyhennettynä PYA) ongelman ratkaisevia algoritmeja. Eräs PYA-algoritmien tärkeä sovelluskohde on DNA-tutkimus, jossa halutaan selvittää kahden (tai useamman) DNA-merkkijonon yhteneväisyyksiä. Pari hieman arkisempaa käyttötarkoitusta tämäntapaisille algoritmeille ovat tiedostojen pakkaus sekä erilaiset hakualgoritmit, joissa merkkijonohakujen ei tarvitse olla tarkkoja. Lisämerkit merkkijonon keskellä sallivat kaupalliset hakualgoritmit käyttävät mitä ilmeisimmin joitain tämänkaltaisia algoritmeja vertaillessaan hakusanoja yleisimpien hakusanojen listojen kanssa, jotta jokaisen haun kohdalla ei tarvitse käydä koko tietokantaa läpi. Tiedostojen pakkaamisessa taas etsitään pitkiä toistuvia palasia, joista osa voidaan karsia pois ja näin pienentää tiedostokokoa; tosin nämä algoritmit eivät välttämättä sovi tähän tarkoitukseen aivan suoraan.

Algoritmien toimintalogiikoiden esittelemisen ja tutkimisen lisäksi niitä testataan tässä tutkielmassa empiirisesti, jotta nähdään niiden suorituskyvyt käytännössä. Aiheesta on tehty aikaisempaa tutkimusta Turun yliopistossa (Ber 2000). Tuossa tutkimuksessa on testattu useampia algoritmeja suoritusajan ja muistinkäytön suhteen. Merkistön koot ja syötteiden pituudet ovat erilaisia kuin tässä työssä, sillä tämän työn merkistöt ovat pienempiä ja syötteet huomattavasti suurempia. Tässä tutkielmassa tarkoituksena on tarkastella aihetta hieman eri näkökulmasta ja samalla hiukan jatkaa em. tutkimusta. Tietotekniikan kehittymisen vuoksi nykyään algoritmeja pystytään helposti testaamaan huomattavasti isommilla syötteillä kuin 15 vuotta sitten. Tämän tutkielman syötekoot nousevat suurimmillaan jo 40 000:een, kun Bergrothin, Hakosen ja Raidan tekemässä tutkimuksessa käytettiin kooltaan huomattavasti pienempiä – 4 000:n pituisia – syötteitä.

Kolme tutkielman algoritmeista on kehitetty jo 1970-luvulla: Wagnerin ja Fischerin menetelmä (W&F 1974), Hirschbergin menetelmä (Hir 1977) ja Hirschbergin lineaarinen menetelmä (Hir 1975). Rickin menetelmä (Ric 1994) on uusin valmiina mukaan otettu menetelmä. Tässä tutkielmassa on tarkoituksena keskittyä selvittämään kaksisuuntaisen lähestymistavan vaikutuksia käytännön suoritusaikaan ja muistinkäyttöön. Tätä varten kehitettiin valmiiden yksisuuntaisten versioiden pohjalta kaksi kahteen suuntaan toimivaa menetelmää.

Tutkielman luvussa 2 selvitetään ensin pisimmän yhteisen alijonon käsitettä ja siihen liittyviä termejä. Tämän jälkeen esitellään kuuden eri algoritmin toimintaperiaatteet esimerkkien avulla. Työssä kehitetyt uudet variantit ovat Hirschbergin menetelmä kaksisuuntaisena ja Rickin menetelmä kaksisuuntaisena. Algoritmit on valittu siten, että aina kahden menetelmän toimintaperiaate olisi mahdollisimman samankaltainen ja että toinen niistä prosessoisi syötteitä vain yhteen suuntaan ja toinen kaksisuuntaisesti. Ensimmäinen pari on Wagnerin ja Fischerin menetelmä sekä Hirschbergin lineaarinen menetelmä, toinen pari koostuu Hirschbergin menetelmästä yksi- ja kaksisuuntaisena versiona ja kolmantena parina on Rickin algoritmi yksi- ja kaksisuuntaisena versiona. Luvussa 3 vertaillaan ensin pareittain näiden algoritmien suoritusaikoja ja muistinkäyttöä käytännön testeillä. Tämän jälkeen kootaan yhteen erilaisista kaksisuuntaisista lähestymistavoista saatavia hyötyjä ja niistä johtuvia haittoja. Lopuksi vertaillaan vielä kaikkia algoritmeja keskenään suoritusaikojen ja muistinkäytön osalta.

2. Pisin yhteinen alijono

Pisimmällä yhteisellä alijonolla (PYA) tarkoitetaan merkkijonojen pisintä mahdollista yhteistä osuutta. Tässä tutkielmassa tarkastellaan vain kahden merkkijonon tapausta. Pisin yhteinen osuus voi olla yhtenäinen merkkijono, tai se voi koostua useammasta pienemmästä yhteisestä osasta, jotka yhdistetään yhdeksi merkkijonoksi. Merkkien ei tarvitse esiintyä samanlaisissa ryhmissä näissä kahdessa merkkijonossa, kunhan ne ovat samassa järjestyksessä. Esimerkiksi toisessa merkkijonoista pisimpään yhteiseen alijonoon kuuluvat merkit voivat olla kaikki peräkkäin, kun taas toisessa merkkijonossa ne voivat olla yksitellen muiden merkkien seassa. Merkkijonot ”abc” ja ”cahbkc” vastaavat em. esimerkin tilannetta, jossa pisimmän yhteisen alijonon merkit a, b ja c ovat ensimmäisessä peräkkäin, kun taas toisessa niiden välissä on muita merkkejä.

Pisimmän yhteisen alijonon ehdot täyttäviä merkkijonoja voi olla useampia. Ne ovat kaikki oikeita vastauksia ongelmaan. Eri algoritmien erilaiset lähestymistavat saattavat antaa samoille merkkijonoille erilaiset ratkaisut, jotka ovat kuitenkin yhtä pitkiä. Esimerkiksi merkkijonoilla ”abcd” ja ”cdab” on kaksi erilaista kahden merkin pituista pisintä yhteistä alijonoa: ”ab” ja ”cd”. Jos pisimmän yhteisen alijonon pituus on yksi, se tarkoittaa, että jokainen molemmista merkkijonoista löytyvä merkki on oikea vastaus. Jos yhteisiä merkkejä ei ole, pisin yhteinen alijono on tyhjä merkkijono, jolloin sen pituus on nolla.

Luvussa 2.1. esitellään aiheeseen liittyviä termejä ja luvuissa 2.2.–2.7. esitellään erilaisia algoritmeja, joita käytetään pisimmän yhteisen alijonon etsimiseen.

2.1. Pisimmän yhteisen alijonon määritelmiä

Algoritmeille annettavia merkkijonoja kutsutaan seuraavassa *syötteiksi* (ja myös *syötejonoiksi*). Ensimmäisenä annettua syötettä kutsutaan myös merkkijonoksi A ja jälkimmäisenä annettua syötettä merkkijonoksi B. Näiden avulla voidaan helposti viitata merkkijonojen eri kohtiin, kuten esimerkiksi A_i tarkoittaa i. merkkiä merkkijonossa A. Vastaavasti B_j on merkkijonon B j. merkki. Syötteiden pituudet merkitään tässä tutkielmassa pienillä kirjaimilla m ja n, joista m tarkoittaa merkkijonon A pituutta ja n merkkijonon B pituutta. Monessa algoritmissa oletetaan, että ensimmäinen syöte on korkeintaan yhtä pitkä kuin toinen syöte. Toisin sanoen syötteiden ollessa eripituisia annetaan lyhempi syöte merkkijonona A ja pidempi syöte merkkijonona B. Syötteiden järjestyksellä saattaa olla merkitystä algoritmin toimintaa ajatellen. Kaikissa algoritmeissa syötteiden järjestyksellä ei ole merkitystä tuloksen oikeellisuuden kannalta, mutta väärä järjestys saattaa aiheuttaa muita ongelmia. Esimerkiksi muistinkäyttö tai suoritusaika saattaa kasvaa todella merkittävästi, kuten luvun 3 testeistä käy ilmi. Varsinkin hyvin eripituisten syötteiden järjestyksellä saattaa olla merkittäviä eroja algoritmien toiminnan kannalta.

Merkkiparilla (A_i, B_j) tarkoitetaan yksinkertaisesti kahden merkin paria, johon viittaaminen liittyy yleensä sen tarkastamiseen, ovatko nämä merkit keskenään samoja. Jos merkkiparin merkit ovat samoja, on löydetty *täsmäys*. Täsmäyksiä löytyy usein paljon enemmän kuin pisimmän yhteisen alijonon pituus on, sillä kaikki täsmäykset eivät yleensä kuulu pisimpään yhteiseen alijonoon. Esimerkiksi syötteistä ”ab” ja ”ba” löytyy kaksi täsmäystä: merkki a

merkkiparista (A_1, B_2) ja merkki b merkkiparista (A_2, B_1). Täsmäykset eivät kuulu samaan pisimpään yhteiseen alijonoon, joten pisimmän yhteisen alijonon pituus on yksi.

Osa täsmäyksistä on *minimaalisia* eli *dominantteja täsmäyksiä*. Tämä tarkoittaa, että täsmäyksen muodostavalle merkille ei löydy ns. taloudellisempaa täsmäyskohtaa. Tällöin täsmäys on löydetty mahdollisimman läheltä syötteiden alkupäätä. Esimerkiksi syötteistä ”abb” ja ”caa” löytyy kaksi täsmäystä. Merkkiparin (A_1, B_2) täsmäys on lähempänä merkkijonon B alkua kuin merkkiparista (A_1, B_3) löytyvä täsmäys, joten näistä ensimmäinen on minimaalinen täsmäys ja jälkimmäinen ei ole.

Kun merkkijonoista tehdään m*n -kokoinen taulukko, sille voidaan muodostaa nk. *korkeuskäyriä*. Tällaista taulukkoa ei muodosteta kaikissa algoritmeissa, mutta tässä tutkielmassa taulukoita on silti luotu osaan algoritmeista helpottamaan algoritmin etenemisen esittämistä. Korkeuskäyrällä tarkoitetaan käyrää, jonka reunat muodostuvat minimaalisten täsmäysten mukaan. Ensimmäisen korkeuskäyrän käännekohdista löytyvät ensimmäiset löydetyt minimaaliset täsmäykset, jotka mahdollisesti kuuluvat pisimpään yhteiseen alijonoon. Toiselta korkeuskäyrältä löytyvät seuraavat löytyneet täsmäykset, jotka jatkavat jotain jo olemassaolevista alijonoista. Jos merkkijonojen tarkastelu tapahtuu eteenpäin, eli merkkijonojen alusta loppuun, molempien merkkiparien indeksit ovat kasvaneet alijonon pidentyessä. Löytyköön esimerkiksi ensimmäinen täsmäys merkkiparista (A_{i1},B_{j1}) ja uusi täsmäys merkkiparista (A_{i2},B_{j2}), kun $i_1 \neq i_2$ ja $j_1 \neq j_2$. Jotta uusi täsmäys sopisi pisimpään yhteiseen alijonoon ja olisi toisen korkeuskäyrän täsmäys, sen tulee toteuttaa ehdot $i_1 < i_2$ ja $j_1 < j_2$. Jos tarkalleen toinen ehdoista ei täyty, uusi täsmäys kuuluu ensimmäisen täsmäyksen kanssa samalle korkeuskäyrälle ja se saattaa aiheuttaa uuden mutkan ensimmäiseen korkeuskäyrään.

Taulukko 1. Esimerkki korkeuskäyristä, joista ensimmäinen korkeuskäyrä on merkitty paksulla mustalla viivalla, toinen kaksoisviivalla ja kolmas kolmoisviivalla. Korkeuskäyrille kuuluvat täsmäykset on merkitty tähdillä, jonka perässä oleva numero ilmoittaa, kuinka mones korkeuskäyrä on kyseessä. Sulkeissa oleva täsmäys ei ole minimaalinen.

	c	b	a	c	b	a
a			$*_1$			$(*_1)$
b		$*_1$			$*_2$	
c	$*_1$			$*_2$		
d						
b		$*_2$			$*_3$	

Taulukossa 1 on kahden merkkijonon alijonojen korkeuskäyrät. Ensimmäinen täsmäys löydetään merkkiparista (A_1,B_3). Ensimmäinen löydetty täsmäys kuuluu tietysti ensimmäiselle korkeuskäyrälle. Ensimmäiseltä riviltä ei voi löytyä enempää täsmäyksiä, jotka jatkaisivat pisintä yhteistä alijonoa. Myös merkkiparista (A_1,B_6) löytyy täsmäys, mutta tämä täsmäys ei ole minimaalinen, koska samalta riviltä täsmäyksen vasemmalta puolelta on jo löytynyt yksi täsmäys lähempää merkkijonon B alkupäätä. Tämä täsmäys ei tee uutta mutkaa korkeuskäyrälle. Pisin yhteinen alijono voidaan muodostaa ilman tätä täsmäystä, koska samalta riviltä on löytynyt parempi (minimaalinen) täsmäys samalta korkeuskäyrältä. Tämän vuoksi merkkiparin (A_1,B_6) täsmäys on merkitty taulukkoon 1 suluissa olevalla tähdellä.

Korkeuskäyrien muodostamiseen ja pisimmän yhteisen alijonon selvittämiseen riittävät pelkät minimaaliset täsmäykset (Ric 1994, s. 6–8).

Taulukon 1 toiselta riviltä löytyy kaksi täsmäystä. Merkkiparin (A_2,B_2) täsmäys ei voi jatkaa ensimmäiseltä riviltä löydettyä alijonoa, koska se ei täytä ehtoja $i_1<i_2$ ja $j_1<j_2$. Tämän vuoksi se ei voi kuulua toiselle korkeuskäyrälle, joten siitä tulee toinen mutka ensimmäiselle korkeuskäyrälle. Tämä täsmäys on myös minimaalinen täsmäys, sillä ensimmäiselle korkeuskäyrälle ei löydy taloudellisempaa vaihtoehtoa tämän rivin merkille. Toisen rivin toinen täsmäys voi jatkaa merkkiparin (A_1,B_3) alijonoa, koska merkkiparin molemmat indeksit ovat suuremmat. Sen voi päätellä myös siitä, että se sijaitsee ensimmäisen käyrän ylimmästä täsmäyksestä oikealle sekä myös sen alapuolella. Näin merkkiparin (A_2,B_5) täsmäyksestä tulee toisen korkeuskäyrän ensimmäinen (minimaalinen) täsmäys.

Kolmannelta riviltä taulukosta 1 löytyy myös kaksi täsmäystä (A_3,B_1) ja (A_3,B_4). Ensimmäisen tilanne on sama kuin edellisenkin rivin ensimmäisellä: se ei voi jatkaa mitään ensimmäisen käyrän täsmäyksen yhteistä alijonoa, joten se kuuluu ensimmäiselle korkeuskäyrälle ja samalla se muodostaa myös uuden mutkan käyrälle. Jälkimmäinen kolmannen rivin täsmäyksistä voi jatkaa kahta ensimmäisen käyrän yhteisistä alijonoista, joten se kuuluu toiselle korkeuskäyrälle ja tekee uuden mutkan siihen. Nämä molemmat täsmäykset ovat myös minimaalisia. Taulukon 1 neljänneltä riviltä ei löydy yhtään täsmäystä, joten riville ei tule mitään. Viidenneltä riviltä sen sijaan löytyy taas kaksi täsmäystä, joista merkkipari (A_5,B_2) voi jatkaa yhtä ensimmäisen korkeuskäyrän täsmäysten muodostamista alijonoista. Se siis kuuluu toiselle korkeuskäyrälle. Viidennen rivin toinen täsmäys, merkkipari (A_5,B_5), voi jatkaa yhtä toisen korkeuskäyrän yhteisistä alijonoista. Siitä tulee sen vuoksi kolmannen korkeuskäyrän ensimmäinen täsmäys. Nämäkin täsmäykset ovat minimaalisia.

Korkeuskäyriä on yhtä monta kuin pisimmän yhteisen alijonon pituus on. Taulukon 1 esimerkissä pisimmän yhteisen alijonon pituus on siis kolme. Koska kolmannelle korkeuskäyrälle kuuluu vain yksi täsmäys, kuuluu tämän täsmäyksen merkki b välttämättä pisimpään yhteiseen alijonoon. Koska kolmannen korkeuskäyrän täsmäys jatkaa vain yhtä toisen korkeuskäyrän yhteisistä alijonoista (merkki c merkkiparissa (A_3,B_4)), kaksi kolmesta merkistä on selvillä. Tuo toisen korkeuskäyrän täsmäys pystyy jatkamaan kahta ensimmäisen korkeuskäyrän yhteisistä alijonoista, joten oikeita pisimpiä yhteisiä alijonoja on kaksi erilaista: ”bcb” ja ”acb”. Algoritmista riippuen lopputulos voi olla kumpi vain.

2.2. Wagnerin ja Fischerin algoritmi

Wagner–Fischer on hyvin yksinkertainen ja suoraviivainen menetelmä, joka käyttää oikeastaan raakaan voimaan perustuvaa (ns. brute force) menetelmää, sillä se käy kertaalleen läpi kaikki mahdolliset merkkiparit (A_i, B_j) (W&F 1974). Alkuperäisessä ideassa tuloksia kirjataan kahteen taulukkoon. Ensimmäinen taulukko (taulukko B) sisältää tiedon täsmäysten lukumääristä. Toiseen taulukkoon (taulukko C) merkitään reittejä, joita pitkin löytää oikeat täsmäykset pisintä yhteistä alijonoa etsiessä. Taulukko C ei ole välttämätön, jos esimerkiksi halutaan tietää pelkästään pisimmän yhteisen alijonon pituus. Pelkän taulukon B avullakin on mahdollista saada selville oikea jono, mutta siinä tapauksessa pisimmän yhteisen alijonon merkit selvittävää algoritmia täytyy muuttaa tästä alkuperäisestä ajatuksesta. Molempien taulukoiden koko on (m+1)*(n+1), joten syötteiden koon kasvaessa suurin piirtein samaa vauhtia

taulukoiden kokojen voidaan katsoa kasvavan neliöllisesti syötteiden koon kasvuvauhtiin nähden.

Wagner—Fischerin menetelmän pseudokoodi (Cor 2001):

```
LCS-LENGTH(X, Y):
    m ← length[X]
    n ← length[Y]
    for i ← 1 to m
        do B[i, 0] ← 0
    for j ← 1 to n
        do B[0, j] ← 0
    for i ← 1 to m
        do for j ← 1 to n
            do if x_i = y_j                          // Löytyi täsmäys
                then B[i, j] ← B[i – 1, j – 1] + 1    // Laske solun arvo
                    C[i, j] ← "↖"
                else if B[i – 1, j] ≥ B[i, j – 1]    // Arvo yläpuolella ≥ arvo vasemmalla
                        then  B[i, j] ← B[i – 1, j]
                              C[i, j] ← "↑"
                        else  B[i, j] ← B[i, j – 1]  // Arvo vasemmalla > arvo yläpuolella
                              C[i, j] ← "←"
    return B and C

PRINT-LCS(C, X, i, j):
    if i = 0 or j = 0
        then return
    if B[i, j] = "↖"
        then PRINT-LCS(C, X, i – 1, j – 1)
            print x_i
    else if B[i, j] = "↑"
        then PRINT-LCS(C, X, i – 1, j)
        else PRINT-LCS(C, X, i, j – 1)
```

Algoritmi lisää taulukon B nollannen sarakkeen ja rivin kaikkiin soluihin nollat. Tämän jälkeen algoritmi etsii merkkijonon A ensimmäiselle merkille täsmäyksiä järjestyksessä kaikista merkkijonon B merkeistä. Jos merkkiparista löytyy täsmäys, taulukkoon B lisätään oikealle kohdalle arvo, joka on yhden pykälän vasemmalla ja yhden rivin ylempänä olevassa alkiossa oleva luku lisättynä yhdellä. Taulukkoon C päivitetään samaan kohtaan yläviistoon vasemmalle osoittava nuoli (korvattu koodissa merkillä *x*), joka näyttää, mistä suunnasta tuohon taulukon soluun on päädytty. Jos merkkiparista ei löydy täsmäystä, vertaillaan tulossolun yläpuolella ja vasemmalla puolella olevia soluja keskenään. Suurempi arvo kopioidaan tähän soluun ja taulukkoon C lisätään vastaavaan kohtaan nuoli joko vasemmalle (korvattu koodissa merkillä *v*) tai ylöspäin (korvattu koodissa merkillä *y*) osoittamaan suunta, josta soluun on päädytty. Jos taulukossa B yläpuolella ja vasemmalla puolella olevien solujen arvot ovat samat, kopioidaan arvo ylemmästä solusta ja asetetaan taulukossa C nuoli osoittamaan ylöspäin.

Esimerkki algoritmin toiminnasta (A = ”abcdbb”, B = ”cbacbaaba”):

Algoritmille annetaan taulukossa 2 näkyvät syötteet ”abcdbb” ja ”cbacbaaba”. Ensin ylin ”nollas” rivi ja vasemmanpuoleisin ”nollas” sarake alustetaan nollilla. Merkkijonon A ensimmäistä merkkiä a verrataan vuorotellen jokaiseen merkkijonon B merkkiin. Ensimmäisestä merkkiparista ei löydy täsmäystä, ja solujen yläpuolella olevan sekä vasemmalla puolella olevan solun arvot ovat yhtäsuuria, joten taulukkoon B kopioidaan nolla yläpuolisesta solusta ja taulukkoon C laitetaan ylöspäin osoittava nuoli. Toisessa merkkiparissa sama toistuu, joten tehdään samat toimenpiteet. Kolmannesta merkkiparista (A_1, B_3) löytyy täsmäys, joten vasemmalla ylemmällä rivillä olevan solun arvoon lisätään yksi, ja tulos merkitään oikealle paikalleen taulukkoon B. Taulukkoon C laitetaan nuoli osoittamaan yläviistoon vasemmalle. Yläviistoon vasemmalle osoittava nuoli osoittaa aina yhtä pienempää numeroa kuin omassa solussa on. Suoraan vasemmalle ja suoraan ylös osoittavat nuolet osoittavat aina samanarvoiseen soluun.

Taulukko 2. Wagnerin ja Fischerin algoritmin tekemät taulukot B ja C yhdistettynä. Soluissa numero tarkoittaa arvoa taulukossa B, ja numeron vasemmalla puolella olevat nuolet ovat taulukon C merkkejä. Korkeuskäyrät on merkitty paksuilla tummilla viivoilla, täsmäykset on lihavoitu ja korkeuskäyrien mutkakohdissa olevat minimaaliset täsmäykset on alleviivattu.

	B	c	b	a	c	b	a	a	b	a
A	0	0	0	0	0	0	0	0	0	0
a	0	↑ 0	↑ 0	↖ **1**	← 1	← 1	↖ **1**	↖ **1**	← 1	↖ **1**
b	0	↑ 0	↖ **1**	↑ 1	↑ 1	↖ **2**	← 2	← 2	↖ **2**	← 2
c	0	↖ **1**	↑ 1	↑ 1	↖ **2**	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2
d	0	↑ 1	↑ 1	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2
b	0	↑ 1	↖ **2**	← 2	↑ 2	↖ **3**	← 3	← 3	↖ **3**	← 3
b	0	↑ 1	↖ **2**	↑ 2	↑ 2	↖ **3**	↑ 3	↑ 3	↖ **4**	← 4

Merkkiparin (A_1, B_4) kohdalla ei ole täsmäystä, joten verrataan vasemmalla olevan ja yläpuolella olevan solun arvoja. Vasemmanpuoleisen solun arvo on suurempi, joten se kopioidaan taulukon B soluun, ja taulukkoon C tulee vasemmalle osoittava nuoli. Ensimmäisen rivin kolmen muun täsmäyksen (merkkiparit (A_1, B_6), (A_1, B_7) ja (A_1, B_9)) kohdalla tehdään samat toimenpiteet kuin ensimmäisenkin löytyneen täsmäyksen kanssa riippumatta siitä, onko riviltä löydetty täsmäyksiä aikaisemmin. Taulukon B soluun laitetaan ykkösen verran isompi arvo kuin yläviistossa vasemmalla olevassa solussa on, ja taulukkoon C lisätään yläviistoon vasemmalle osoittava nuoli. Merkkiparien (A_1, B_5) ja (A_1, B_8) kohdalla toimitaan samalla tavalla kuin merkkiparin (A_1, B_4) kohdalla, eli täsmäystä ei löydy, ja vasemmalla olevan solun arvo on suurempi kuin yläpuolella olevan. Tämä suurempi arvo kopioidaan B:hen, ja taulukkoon C laitetaan vasemmalle osoittava nuoli.

Toisella rivillä ensimmäinen merkkipari ei tuota täsmäystä, ja vertailtavat solut ovat nollia. Taulukkoon B tulee nolla ja taulukkoon C nuoli ylöspäin. Toisen rivin toisesta merkkiparista

löytyy täsmäys, joten taulukon B soluun tulee yhtä isompi arvo kuin yläviistossa vasemmalla. Taulukkoon C laitetaan taas yläviistoon vasemmalle osoittava nuoli. Tästä tulee ensimmäisen korkeuskäyrän toinen täsmäys. Algoritmi voi valita näistä lopussa siis vain toisen pisimpään yhteiseen alijonoon. Kaksi seuraavaa merkkiparia eivät tuota täsmäyksiä ja vertailtavien solujen arvot ovat yhtäsuuret, joten taas arvo kopioidaan ylemmästä B:hen, ja C:hen tulee nuoli ylöspäin. Merkkipari (A_2, B_5) on ensimmäinen löydetty toisen korkeuskäyrän täsmäys. Yläviistossa vasemmalla olevan solun arvoon lisätään yksi, ja tulos tulee taulukkoon B. C:hen tulee vasemmalle yläviistoon näyttävä nuoli kertomaan, mistä algoritmi on tähän soluun päätynyt. Loppuriviltä löytyy vielä yksi vastaava täsmäys sekä kolme tapausta, joissa vasemmalla olevan solun arvo on suurempi kuin yläpuolella olevan. Näissä tapauksissa tehdään samat operaatiot kuin aikaisemminkin.

Algoritmi jatkaa samaa tarkastelua rivi riviltä ja merkki merkiltä, kunnes kaikki merkkiparit on käyty läpi. Neljänneltä riviltä ei löydy ollenkaan täsmäyksiä, joten koko rivin numerot kopioidaan yläpuolelta taulukkoon B, ja kaikki nuolet asetetaan osoittamaan ylöspäin taulukossa C. Algoritmi käyttää jokaisen merkkiparin kohdalla vain merkkien vertailua, ja sen tuloksen perusteella käytetään yhden tai kahden vieressä olevan solun arvoa. Taulukon B nollas rivi ei ole enää käytössä ensimmäisen rivin valmiiksi saamisen jälkeen, ensimmäinen rivi muuttuu käytännössä turhaksi toisen rivin valmiiksi saamisen jälkeen jne. Saman arvon sisältävät solut voidaan ryhmitellä piirtämällä eriarvoisten solujen väliin viivat, kuten taulukossa 2 on tehty. Viivat ovat samalla korkeuskäyriä, joiden kulmista löytyvät mahdolliset pisimpään yhteiseen alijonoon kuuluvat täsmäykset.

Kun algoritmi on saanut taulukot B ja C valmiiksi, voidaan katsoa pisimmän yhteisen alijonon pituus suoraan alimman rivin oikeanpuoleisimman solun arvosta. Alkuperäinen algoritmi tulostaa pisimmän yhteisen alijonon rekursiivisten kutsujen avulla. Kutsuja jatketaan, kunnes päädytään riville tai sarakkeeseen, jonka järjestysnumero on nolla. Kun rekursiokutsu on palautunut yläviistoon osoittavan nuolen jälkeen, se tulostaa kutsumispaikasta löydetyn merkin. Näin merkit tulevat suoraan oikeaan järjestykseen.

Vielä yksinkertaisempi ja helpommin ymmärrettävä tapa selvittää pisin yhteinen alijono on seurata taulukon C nuolia oikeasta alanurkasta lähtien. Löydetyt merkit lisätään aina ensimmäisiksi, jotta merkkien järjestys menee oikein. Viimeisen rivin oikeanpuoleisimman solun nuoli näyttää vasemmalle, joten tarkastelu siirtyy tähän soluun. Se näyttää yläviistoon, joten tiedämme, että tämä täsmäys kuuluu viimeiselle löydetylle korkeuskäyrälle. Samalla se tarkoittaa, että kyseisen täsmäyksen merkki, b, on pisimmän yhteisen alijonon viimeinen merkki. Merkki laitetaan muistiin, minkä jälkeen seurataan C-taulukon nuolia kaksi kertaa vasemmalle, kunnes tulee taas vastaan yläviistoon osoittava nuoli. Tämä täsmäys kuuluu viimeistä edeltävälle korkeuskäyrälle, joten laitetaan merkki muistiin edellisen löydöksen etupuolelle. Seuraava nuoli osoittaa ylöspäin, joten siirrytään riviä ylemmäs. Sieltä löytyy seuraava pisimpään yhteiseen alijonoon kuuluva merkki, joka laitetaan muistiin kahden löytyneen etupuolelle. Seuraavaksi siirrytään vielä yksi kerta ylöspäin, ennen kuin löytyy ensimmäisen korkeuskäyrän täsmäys. Tämä merkki laitetaan taas muistiin ja siirrytään nuolen osoittamaan suuntaan. Nyt tulee vastaan solu, jonka rivinumero on nolla, joten enempää täsmäyksiä ei voida enää löytää. Algoritmi on nyt löytänyt yhden mahdollisen pisimmistä yhteisistä alijonoista: jonon ”acbb”.

Aikavaativuus:

Algoritmi toimii kaikilla syötteillä samalla tavalla. Se käy läpi samat vertailut kaikille m*n merkkiparille taulukkoja luodessaan, minkä jälkeen selvitetään reitti ja pisin yhteinen alijono taulukon C avulla. Pisimmän reitin selvittämisen aikavaativuus on korkeintaan max(m, n), joten se ei vaikuta aikavaativuusluokkaan. Koko algoritmin aikavaativuus on siis luokkaa O(mn), joten syötteiden koon kasvaessa suurin piirtein samaa vauhtia voidaan aikavaativuuden katsoa olevan neliöllinen. Jos pelkkä tieto pisimmän yhteisen alijonon pituudesta riittää, ei tarvita kuin yksi taulukko, jossa olevat kaksi riviä riittäisivät syötteiden läpikäymiseen. Taulukon kooksi tulee siis 2*(n+1), jolloin algoritmi toimii lineaarisessa tilassa. Taulukon B päivittämiseen tarvitaan tässä tapauksessa yhtä monta kierrosta; vain taulukon C päivityskohdat jäävät pois. Pisimmän reitin laskemisen poisjättäminen vaikuttaa jonkin verran suoritusaikaan, mutta aikavaativuusluokka pysyy samana pienestä nopeutumisesta huolimatta.

2.3. Hirschbergin algoritmi

Hirschbergin algoritmilla on Wagner–Fischerin menetelmää monimutkaisempi lähestymistapa. Ennen suorituksen alkamista se kerää jälkimmäisen syötteen tiedot taulukoihin, joiden avulla se pystyy karsimaan turhia vertailuja. Taulukoihin kerätään merkkikohtaisesti tieto merkin esiintymiskohdista sekä merkkien lukumäärä. Näiden tietojen avulla algoritmi rajaa mahdollisen alueen, jolta se etsii täsmäyksiä toisesta syötteestä yksi korkeuskäyrä kerrallaan. Korkeuskäyrien täsmäyksiä etsitään kuvitteellisesta m*n -taulukosta oikealta ylhäältä vasemmalle ja alas. Uudet korkeuskäyrät sijaitsevat aina edellisiin verrattuna niistä oikealle ja alas, eli molempien syötteiden alusta loppua kohden.

Algoritmin toiminta vaatii muutaman taulukon jatkuvan ylläpitämisen ja tarkastelun. Taulukoiden koko määräytyy ennalta määritellyn merkistön ja jälkimmäisen syötteen pituuden perusteella. Merkistön täytyy sisältää kaikki mahdolliset syötteissä käytössä olevat merkit, jotta algoritmi toimisi oikein. Taulukoita on yhteensä neljä kappaletta, joista kolmea käytetään algoritmin muistina ja yksi on tulosta varten. Taulukot NB ja N sisältävät tiedon kunkin merkin kappalemäärästä. NB:ssä on syötteen B kunkin merkin kokonaislukumäärä, jota käytetään vain taulukon N alustamiseen eri vaiheiden välillä. N on taulukko, jonka sisältö muuttuu algoritmin suorituksen edetessä. Taulukko PB sisältää tiedot kunkin merkin esiintymäkohdista jälkimmäisessä syötteessä. PB ei myöskään muutu algoritmin suorituksen aikana. Taulukko D sisältää tulokset korkeuskäyrittäin. Taulukon D koko on (m+1)*(m+1), jolloin siinä on tilaa suurimmalle mahdolliselle tapaukselle. Suurin tapaus on, että koko merkkijono A sisältyy merkkijonoon B. Tässäkin eniten tilaa vaativassa tapauksessa taulukosta D on käytössä vain noin puolet, kun yleisimmissä tapauksissa taulukosta on käytössä vain pieni osa.

Hirschbergin algoritmi asettaa vaiheessa 1 ensimmäiseksi taulukoiden NB ja PB jokaisen merkin kohdalle nollan. Tämän jälkeen se käy syötteen B läpi merkki kerrallaan. Jokaisen merkin kohdalla merkin lukumäärää kasvatetaan yhdellä taulukossa NB, ja tämä merkin esiintymiskohta lisätään taulukkoon PB oikean merkin kohdalle. Algoritmin vaiheessa 2 alustetaan taulukon D ylimmäksi riviksi nollarivi sekä asetetaan muuttujaan lowcheck arvo nolla. Muuttujassa lowcheck pidetään muistissa korkeuskäyrän ensimmäisen täsmäyksen rivi kuvitteellisesta m*n -taulukosta. Seuraavia korkeuskäyriä ei siten etsitä sellaisilta riveiltä, joilla ne eivät voi sijaita. Vaiheessa 3 asetetaan ensimmäisen etsittävän korkeuskäyrän numeroksi

yksi. Vaiheessa 4 alustetaan taulukko N kopioimalla arvot taulukosta NB. Lipun (FLAG) arvoksi asetetaan nolla, joka tarkoittaa, että tältä korkeuskäyrältä ei ole vielä löytynyt täsmäyksiä. Päivitetään myös muuttujat low ja high, jotka kertovat algoritmille, miltä sarakeväliltä täsmäykset voivat löytyä m*n -taulukosta.

Hirschbergin algoritmin pseudokoodi (Hir 1977):

Hirsch(A, B):

1. NB[θ] ← 0 for θ = 1, ..., s // s = merkistön viimeinen merkki
   PB[θ, 0] ← 0 for θ = 1, ..., s
   PB[0, 0] ← 0, PB[0, 1] ← 0 // Ns. nollamerkit, joita ei löydy syötteestä B
   for j ← 1, step = 1, until n do
   NB[b_j] ← NB[b_j] + 1
   PB[b_j, NB[b_j]] ← j
2. D[0, i] ←0 for i = 0, ..., m
   lowcheck = 0
3. for k ← 1, step = 1, do
4. N[θ] ← NB[θ] for θ = 1, ..., s
   N[0] ← 1 // Ns. nollamerkit, joita ei löydy syötteestä B
   FLAG ← 0
   low ← D[k – 1, lowcheck]
   high ← n + 1
5. for i ← lowcheck + 1, step = 1, until m do
6. while PB[a_i, N[a_i] – 1] > low do N[a_i] ← N[a_i] – 1
7. if high > PB[a_i, N[a_i]] > low
   high ← PB[a_i, N[a_i]]
   D[k, i] ← high
   if FLAG = 0, then {lowcheck ← i; FLAG ← 1}
   else D[k, i] ← 0
8. if D[k – 1, i] > 0 then low ← D[k – 1, i]
9. if FLAG = 0 then go to 10.
10. k ← k – 1
    for i ← m, step = –1, until 1 do
    if D[k, i] > 0
    c_k ← a_i
    k ← k – 1

Algoritmin vaiheessa 5 aloitetaan yhteisten merkkien paikkojen etsiminen alustamalla muuttujan i arvoksi lowcheck+1. Tämä tarkoittaa, että ensimmäiseksi etsitään merkkijonon A ensimmäiselle merkille täsmäyksiä. 6. vaiheen silmukka vähentää taulukossa N ko. merkin kohdalla arvoa yhdellä niin kauan kuin arvo taulukon N arvon kohdalla taulukossa PB on suurempi kuin muuttuja low. Toisin sanoen tämä while-lause etsii taulukosta PB sarakenumeroltaan pienimmän low:ta suuremman kyseisen merkin kohdan, jos sellaista on edes mahdollista löytää. Vaiheessa 7 testataan, osuuko löydetty kohta arvojen low ja high väliin. Löydetyn kohdan osuminen tähän väliin tarkoittaa minimaalisen täsmäyksen löytymistä. Tämän jälkeen päivitetään kyseinen kohta muuttujaan high, jotta tämän merkin jälkeen

myöhemmät merkit merkkijonossa B eivät voi enää kuulua ensimmäiselle korkeuskäyrälle. Päivitetään täsmäyksen kohta myös taulukkoon D. Jos täsmäys oli ensimmäinen tältä korkeuskäyrältä löydetty, päivitetään lipun arvoksi 1 sekä lowcheck:iin täsmäyksen kohta merkkijonossa A. Jos taas löydetty kohta ei osunut arvojen low ja high väliin, kyseessä ei ole ainakaan minimaalinen täsmäys, jolloin taulukkoon D tallennetaan arvo nolla. Vaiheessa 8 tarkastetaan, onko taulukossa D nykyisen solun yläpuolella olevan solun arvo suurempi kuin 0. Jos arvo on nollaa suurempi, edellisellä korkeuskäyrällä on minimaalinen täsmäys kyseisellä rivillä. Kyseinen arvo päivitetään muuttujaan low seuraavaa kierrosta varten, jotta täsmäyksiä etsitään oikealta sarakeväliltä. Laillisten täsmäysten sarakeväli siirtyy siis vasemmalle.

Täsmäyksiä etsitään vaiheiden 6–8 mukaisesti merkkijonon A merkki kerrallaan, kunnes vaiheen 5 ehto i≤m ei enää ole voimassa. Kun merkkijonon A merkit on käyty kertaalleen läpi, on löydetty ensimmäisen korkeuskäyrän täsmäykset. Tämän jälkeen vaiheessa 9 tarkastetaan lipun arvosta, onko löytynyt yhtään täsmäystä. Jos täsmäyksiä on löytynyt, jatketaan seuraavan korkeuskäyrän etsimisellä vaiheesta 3 alkaen. Mikäli täsmäyksiä ei ole löytynyt, siirrytään vaiheeseen 10, jossa selvitetään PYAn merkit taulukon D tulosten avulla.

Esimerkki algoritmin toiminnasta (A = ”abcdbb”, B = ”cbacbaaba”):

Annetaan Hirschbergin algoritmille syötteet ”abcdbb” ja ”cbacbaaba”. Merkistö sisältää merkit a, b, c ja d. Taulukkoon NB alustetaan jokaisen neljän merkin kohdalle lukumääräksi nolla. Asetetaan nollat myös jokaiselle numerolle taulukkoon PB. Käydään merkkijonon B merkit läpi yksitellen laskien lukumäärät taulukkoon NB ja lisäämällä merkkien paikat taulukkoon PB. Taulukko PB on taulukon 4 mukainen ja NB on taulukon 5 mukainen. Asetetaan taulukkoon D ylimmälle riville (nollas rivi) jokaiseen soluun arvo nolla. Muuttuja lowcheck saa arvon nolla.

Taulukko 3. Hirschbergin algoritmin tulokset taulukossa D syötteillä ”abcdbb” ja”cbacbaaba”. Rivinumero tarkoittaa korkeuskäyrää k ja sarakkeen numero i merkin esiintymiskohtaa merkkijonossa A. Nollaa suurempi luku solussa tarkoittaa käyrän k minimaalista täsmäystä, jossa solun arvo on merkin sijainti merkkijonossa B ja sarakkeen numero on merkin sijainti A:ssa.

i / k	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1		3	2	1	0	0	0
2			5	4	0	2	0
3				0	0	5	0
4							8
5							
6							

Taulukko 4. Merkkijonon B merkkien esiintymiskohdat löytyvät taulukosta PB.

a	0	3	6	7	9
b	0	2	5	8	
c	0	1	4		
d	0	0			

Taulukko 5. Merkkijonon B eri merkkien lukumäärät taulukossa NB sekä taulukossa N sen alustamishetkellä. Merkki d on ns. nollamerkki, jota ei esiinny merkkijonossa B, joten sen arvoksi on asetettu 1.

a	4
b	3
c	2
d	1

Ensimmäisen korkeuskäyrän etsiminen alkaa asettamalla k:n arvoksi yksi. Taulukkoon N kopioidaan arvot taulukosta NB, joten se on tämän jälkeen taulukon 5 mukainen. Lipun arvoksi asetetaan nolla, low:n arvoksi solun [0,0] arvo taulukosta D ja high:n arvoksi 10. Vaiheen 5 aluksi muuttuja i saa arvon yksi, eli etsitään merkkijonon A ensimmäiselle merkille täsmäyksiä. Vaiheessa 6 etsitään siis pienintä low:ta suurempaa arvoa taulukosta PB tarkasteltavan merkin a kohdalta. Tulokseksi saadaan 3, kun N[a]=1. Koska low<3<high, löytyy minimaalinen täsmäys. Päivitetään high:n arvoksi 3, sekä kopioidaan sama arvo 3 taulukkoon D kohtaan [1,1], joka tarkoittaa täsmäyksen löytyvän ensimmäiseltä käyrältä merkkiparista (A_1,B_3). Lipun arvoksi muutetaan yksi täsmäyksen löytymisen merkiksi, minkä jälkeen päivitetään lowcheck arvoon 1. Vaiheen 8 if-lause ei toteudu, joten algoritmi siirtyy seuraavaan A:n merkkiin.

Muuttujan i arvoksi asetetaan 2, joten tässä vaiheessa etsitään ensimmäisen korkeuskäyrän täsmäyksiä merkkijonon toiselle merkille. Pienin low:ta suurempi arvo merkin b kohdalta taulukosta B on 2, joka on low:ta (arvo 0) suurempi ja high:ta (arvo 3) pienempi. Löytyi siis minimaalinen täsmäys, joten päivitetään high arvoon 2 ja laitetaan sama arvo myös taulukkoon D muistiin paikkaan [1, 2]. Muut ehtolauseet eivät enää pidä paikkaansa, joten i:tä kasvatetaan ja siirrytään merkkijonon kolmanteen merkkiin.

Merkin c kohdalta taulukosta PB löytyy low:ta suurempi arvo 1, joka osuu low:n (arvo 0) ja high:n (arvo 2) väliin. Tämän on ensimmäisen korkeuskäyrän kolmas täsmäys, joten päivitetään high:n arvoksi 1 ja lisätään arvo myös taulukkoon D omalle paikalleen soluun [1, 3]. Ehtolauseet eivät toteudu, joten i kasvaa arvoon 4. Ainoa arvo merkille d taulukossa PB on 0, joten valitaan se. Arvo ei ole kuitenkaan suurempi kuin low, joten merkitään taulukkoon D arvo 0 soluun [1, 4]. Vaiheen 8 ehtolause ei toteudu, joten siirrytään seuraavaan merkkiin. Kahdella viimeisellä kierroksella (i=5 ja i=6) taulukosta PB löydetään pienin low:ta suurempi arvo 2, mutta kummallakaan kerralla ehto low (=0)<2<high (=1) ei täyty. Taulukon D ensimmäisen korkeuskäyrän eli ensimmäisen rivin kahteen viimeiseen soluun laitetaan näin arvo 0. Vaiheen 8 ehtolause ei toteudu, joten ensimmäinen korkeuskäyrä on valmis. Koska lipun arvo ei ole nolla, jatketaan seuraavan korkeuskäyrän etsimisellä.

Toisen korkeuskäyrän etsiminen aloitetaan vaiheesta 3 päivittämällä k:n arvoksi 2. Vaiheessa 4 alustetaan taas taulukko N alkuarvoihinsa kopioimalla arvot taulukosta NB. Lipun arvoksi tulee

nolla, low:n arvoksi taulukon D solun [1, 1] arvo 3 ja high:n arvoksi 10. Täsmäysten etsiminen aloitetaan merkkijonon A toisesta merkistä. Taulukosta PB löytyy b:n kohdalta low:ta suurempi arvo 5, joka on samalla myös pienempi kuin high:n arvo. Löytyi toisen korkeuskäyrän ensimmäinen täsmäys, joten muutetaan muuttujan high arvoksi 5, tallennetaan arvo taulukkoon D soluun [2, 2], päivitetään korkeuskäyrän ensimmäisen täsmäyksen rivinumero muuttujaan lowcheck sekä asetetaan lipun arvoksi 1. Tällä kerralla vaiheen 8 ehto toteutuu, sillä edelliselle korkeuskäyrälle löytyi täsmäys samasta merkkijonon A kohdasta. Muutetaan low:n arvoksi 2.

Kolmannelle merkille c löytyy taulukosta PB juuri päivitettyä low:ta suurempi arvo 4. Tämä on myös pienempi kuin high:n arvo 5, joten toiselle korkeuskäyrälle löytyi toinen täsmäys. High:n arvoksi muutetaan 4, ja sama arvo kopioidaan myös taulukkoon D. Edelliselle korkeuskäyrälle löytyi tässäkin tapauksessa täsmäys, joten päivitetään low:n arvoksi 1. Merkkijonon neljännelle merkille d ei löydy vieläkään muuta arvoa kuin 0, joka ei ole low:ta suurempi. Laitetaan taulukkoon D arvo nolla ja siirrytään seuraavaan merkkiin. Viidennelle merkille b löytyy low:ta suurempi arvo 2, joka on samalla myös pienempi kuin high. Täsmäyksen löydyttyä päivitetään high:n arvo ja merkitään tulos myös D:hen. Edellisellä käyrällä tälle viidennelle merkille ei ollut minimaalisia täsmäyksiä, joten siirrytään viimeiseen merkkiin. Tälläkin kerralla löytyy sama low:ta suurempi arvo 2, mutta nyt se ei ole pienempi kuin high, joten muutetaan taulukon D solun [2,6] arvoksi 0. Edellisen käyrän täsmäyksiä ei ole ja lipun arvo on 1, joten algoritmi jatkaa vaiheesta 3 kolmannen korkeuskäyrän etsimisellä.

Kolmannen korkeuskäyrän etsiminen aloitetaan muuttujien alustamisella. Taulukon N arvot palautetaan alkuarvoihinsa, lipun arvoksi muutetaan 0, low:n arvoksi otetaan taulukon D solusta [2, 2] arvo 5, ja high:n arvoksi tulee taas 10. Vaiheessa 5 aloituspisteeksi määritetään merkkijonon A kolmas merkki. Merkin c suurin arvo ei ole suurempi kuin low, joten taulukkoon D laitetaan arvo 0. Edelliseltä käyrältä löytyi vastaavasta kohdasta täsmäys, joten päivitetään low:n arvoksi 4. Neljännelle merkille ei löydy vieläkään täsmäystä, joten laitetaan nolla taulukkoon D ja jatketaan seuraavalla merkillä.

Viidennelle merkkijonon A merkille b löytyy low:ta suurempi arvo 5, joka on myös pienempi kuin high. Arvo kopioidaan muuttujaan high sekä taulukkoon D. Kolmannen korkeuskäyrän ensimmäisen täsmäyksen merkiksi lipun arvoksi muutetaan 1 ja lowcheckin arvoksi tulee 5. Edelliseltä käyrältä löytyi täsmäys, joten päivitetään low:n arvoksi 2. Kuudennelle merkille, b, löytyy sama low:ta suurempi arvo 5, mutta enää se ei ole pienempi kuin high. Taulukkoon D tulee arvo 0. Edellisellä korkeuskäyrällä ei ollut tässä kohtaa minimaalisia täsmäyksiä, joten kolmas korkeuskäyrä on valmis.

Neljännen korkeuskäyrän alustukset ovat samanlaiset kuin aikaisemminkin taulukon N ja lipun sekä high:n arvojen osalta. Muuttujan low arvoksi tulee nyt 5. Vaihe 5 laittaa korkeuskäyrän etsimisen alkamaan kuudennesta merkistä. Merkin B pienin low:ta suurempi arvo taulukossa PB on 8. Se on samalla myös pienempi kuin high, joten siitä tulee neljännen korkeuskäyrän ensimmäinen täsmäys. Tulos päivitetään muuttujaan high ja taulukkoon D, minkä jälkeen lowcheckin arvoksi tulee 6 ja lipun arvoksi tulee 1. Edellisen käyrän täsmäystä ei ole ja enempää merkkejä ei ole, joten alkaa viidennen käyrän etsiminen. Vaiheessa 4 alustetaan taas muuttujat, low:n arvoksi tulee 8. Vaiheen 5 silmukkaan ei mennä sisälle, sillä lowcheck+1 on

suurempi kuin m. Vaiheen 9 tarkastuksessa lipun arvoksi jäi nyt nolla, joten siirrytään vaiheeseen 10.

Vaiheessa 10 selvitetään pisimmän yhteisen alijonon merkit. Tarkastelu aloitetaan ylimmältä korkeuskäyrältä, joka on tässä tapauksessa neljäs korkeuskäyrä. Eteneminen aloitetaan taulukon oikeasta laidasta ja edetään yksi sarake kerrallaan vasemmalle. Jos solun arvo on nollaa suurempi, kyseinen täsmäys kuuluu pisimpään yhteiseen alijonoon. Tämän jälkeen kyseiseltä käyrältä ei voida ottaa enempää pisimpään yhteiseen alijonoon kuuluvia täsmäyksiä, joten siirrytään yhtä riviä ylemmäs, edelliselle korkeuskäyrälle. Pisin yhteinen alijono on löytynyt, kun on saavutettu taulukon vasen reuna.

Tämän esimerkin neljännen korkeuskäyrän täsmäys löytyy taulukon D sarakkeesta 6, joten se on merkkijonon A kuudes merkki, b. Etsintää jatketaan kolmannelta korkeuskäyrältä sarakkeen vasemmalta puolelta. Toinen täsmäys löytyy heti sarakkeesta 5, eli merkki b kuuluu pisimpään yhteiseen alijonoon. Toisen korkeuskäyrän neljäs sarake on nolla, joten siinä ei ole täsmäystä. Kolmannesta sarakkeesta sen sijaan löytyy taas täsmäys, tällä kertaa A:n kolmas merkki, c. Ensimmäiseltä käyrältä löytyy heti viereisestä sarakkeesta täsmäys, A:n toinen merkki b. Tämän jälkeen etsintä siirtyy nollariville, jolta ei löydy enää täsmäyksiä. Löytynyt pisin yhteinen alijono on siis ”bcbb”, joka on eri kuin luvussa 2.2. Wagnerin ja Fischerin menetelmän löytämä vastaus. Pituus on kuitenkin sama, ja molemmat vastaukset ovat oikein.

Jos vertaa taulukkoa 3 taulukkoon 2, voi nähdä Hirschbergin menetelmän löytävän samat minimaaliset täsmäykset kuin Wagnerin ja Fischerin menetelmä. Taulukossa 3 ensimmäinen rivi näyttää ensimmäisen korkeuskäyrän kulmapisteiden merkkiparit (A_1,B_3), (A_2,B_2) ja (A_3,B_1), jotka ovat samat kuin taulukossa 2. Sama pätee myös muille korkeuskäyrille. Hirschbergin taulukko D sisältää samat tiedot kuin Wagner–Fischerin taulukko B, tosin hiukan erilaisessa muodossa. Taulukon 2 avulla voi muodostaa taulukon 3 ja päinvastoin, tosin ilman nuolia, jotka kuuluvat Wagner–Fischerin taulukkoon C.

Aikavaativuus:

Alussa suoritettavien taulukoiden alustuksien aikavaativuus on luokkaa $\mathrm{n}\log s$, jossa s on merkistön koko. Oletetaan, että m≤n, ne ovat suurin piirtein samaa suuruusluokkaa ja merkistön koko on huomattavasti syötteiden kokoa pienempi. Taulukoiden alustuksien suoritusaika on O(n). Vaihe 3 suoritetaan p+1 kertaa, missä p tarkoittaa korkeuskäyrien lopullista lukumäärä. Vaiheen 3 sisällä vaiheen 4 alustuksen aikavaativuus on O(s), joka on hyvin pieni verrattuna vaiheen 5 for-silmukkaan, kun s≤n. Vaiheen 5 silmukka suoritetaan maksimissaan m kertaa. Huonoimmassa tapauksessa vaiheen 6 for-lause suoritetaan n kertaa yhden korkeuskäyrän aikana, sillä taulukon N arvoja ei kasvateta vaiheen 5 sisällä. Vaiheet 7 ja 8 ovat vakioajassa suoritettavia ja vaihe 9 suoritetaan vakioajassa vaiheen 3 lopuksi. Näiden lisäksi vaiheen 10 suoritusaika on O(m). Huonoimman tapauksen aikavaativuus on siis luokkaa $\mathrm{O}(\mathrm{pn} + \mathrm{n}\log s)$, ja kun $\mathrm{p} \geq \mathrm{O}(\log \mathrm{s})$, aikavaativuus on luokkaa $\mathrm{O}(\mathrm{pn})$. Suoritusaika riippuu hyvin paljon sekä syötteiden koosta että syötteiden yhteisistä merkeistä, joten keskimääräistä suoritusaikaa ei kannata lähteä arvioimaan.

2.4. Hirschbergin lineaarinen menetelmä

Tässä luvussa esitellään toinen Hirschbergin kehittämä PYA-algoritmi, joka käyttää hajota-ja-hallitse-periaatetta. Se toimii nimensä mukaan lineaarisessa tilassa ja käyttää osittain hyväkseen samaa suoraviivaista tekniikkaa kuin Wagnerin ja Fischerin menetelmä. Hirschberg julkaisi tämän menetelmän vuonna 1975 (Hir 1975). Wagnerin ja Fischerin algoritmin tilavaativuus on neliöllinen, kun syötteiden suuruusluokat ovat suurinpiirtein samoja. Silloisilla tietokoneiden muistimäärillä tämä oli suuri ongelma. Hirschbergin lineaarinen algoritmi sen sijaan ei käytä suuria taulukoita tulosten tallentamiseen eikä siten joudu varaamaan tilaa mahdollista pahinta tapausta varten kuten luvussa 2.3. esitelty Hirschbergin algoritmi. Koska tämä menetelmä toimii lineaarisessa tilassa, se pystyy käsittelemään samalla muistimäärällä paljon suuremman kokoluokan syötteitä kuin edellisissä luvuissa esitellyt Wagnerin ja Fischerin sekä Hirschbergin algoritmit.

Hirschbergin lineaarinen menetelmä etsii täsmäyksiä kahdesta suunnasta. Se koostuu kahdesta osasta. *Algoritmi C* käyttää *algoritmin B* taulukoiden tuloksia, joiden avulla se kutsuu itseään rekursiivisesti pienempien osaongelmien ratkaisemiseksi. Algoritmi C jakaa ensimmäisen syötteen puoliksi pyöristäen alaspäin. Ensimmäiseksi etsitään alkupuolen ja kokonaisen toisen syötteen täsmäykset etuperin edeten. Tämä tapahtuu algoritmin B avulla. Algoritmi B tekee käytännössä samat operaatiot kuin luvussa 2.2. esitelty Wagner–Fischer, mutta se ylläpitää vain kahta viimeisintä riviä ja palauttaa lopussa viimeisen rivin. Seuraavaksi etsitään loppupuolen täsmäykset kokonaisen toisen syötteen kanssa, mutta tämä tapahtuu alkuperäisiin syötteisiin nähden takaperin, eli lopusta alkuun päin. Algoritmi B etsii näillekin viimeisen rivin tuloksen samalla tavalla.

Algoritmi B on palauttanut kaksi yhden rivin taulukkoa, jotka ovat L1 ja L2. Näiden taulukoiden tuloksista etsitään alijonojen yhteistä maksimipituutta laskemalla yhteen arvot mahdollisista katkaisukohdista. Summat lasketaan ottamalla L1:stä ensimmäinen arvo ja lisäämällä siihen L2:n viimeinen arvo. Viimeinen yhteenlaskettu summa saadaan siis ottamalla L1:n viimeinen arvo ja lisäämällä siihen L2:n ensimmäinen arvo. Suurimman löytyneen summan kohdalla on jälkimmäisen syötteen oikea katkaisukohta. Jos suurimpia summia on useampia, valitaan niistä ensimmäiseksi löydetty. Tämän jälkeen algoritmi C etsii oikeita täsmäyksiä kutsumalla itseään rekursiivisesti kahdelle lyhyemmälle merkkijonoparille. Algoritmi C jatkaa rekursiokutsuja ja ongelman pilkkomista pienempiin aliongelmiin, kunnes alle kahden mittaisista syötteistä ratkaistaan lopulta mahdolliset täsmäävät merkit. Lopulta näiden pienten osaongelmien palauttamat mahdolliset merkit yhdistetään pisimmäksi yhteiseksi alijonoksi.

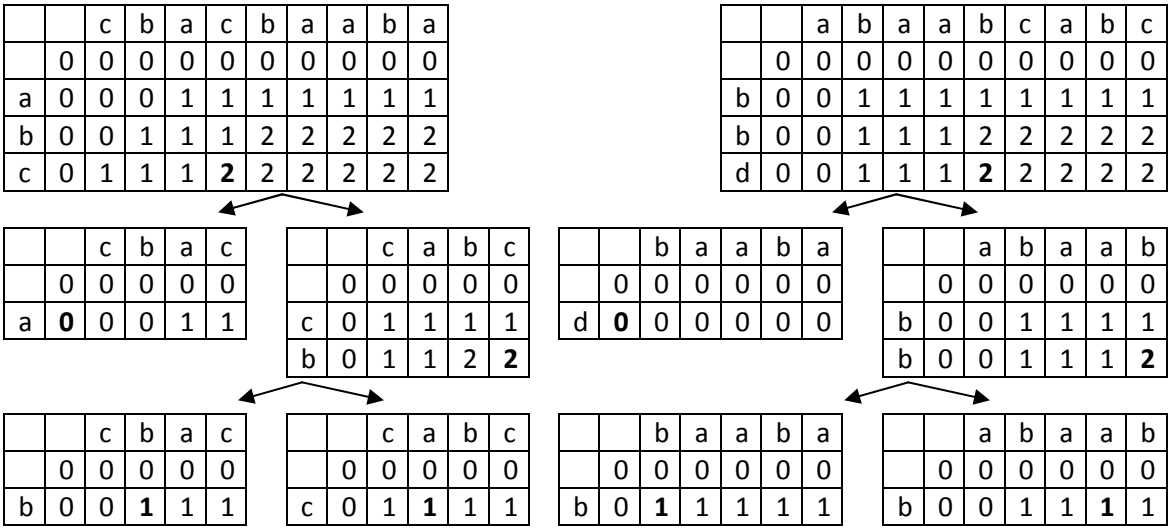
Hirschbergin lineaarisen algoritmin pseudokoodi (Hir 1975):

```
ALG B (m, n, A, B, LL):
1.  K(1, j) ←0                  [j = 0, …, n]
2.  for i ← 1 to n do
3.      K(0, j) ← K(1, j)       [j = 0, …, n]
4.      for j ← 1 to n do
            if A(i) = B(j)      then K(1, j) ← K(0, j – 1) + 1
                                else K(1, j) ← max{K(1, j – 1), K(0, j)}
5.  LL(j) ← K(1, j)             [j = 0, …, n]
```

ALG C (m, n, A, B, C):

1. if n = 0 then C ←e (e = tyhjä merkkijono)
   else if m = 1 then if ∃ j ≤ n siten, että A(1) = B(j)
   then C ← A(1)
   else C ← e
2. else i ← $\lfloor m/2 \rfloor$
3. ALG B (i, n, A_{1i}, B_{1n}, L1)
   ALG B (m – i, n, $\hat{A}_{m\,(i+1)}$, $\hat{B}_{n1}$, L2)
4. M ← $\max_{0 \le j \le n} \{L1(j) + L2(n - j)\}$
   k ← min j siten, että L1(j) + L2(n – j) = M
5. ALG C (i, k, A_{1i}, B_{1k}, C1)
   ALG C (m – i, n – k, $A_{(i+1)\,m}$, $B_{(k+1)\,n}$, C2)
6. C ← C1 + C2

Esimerkki algoritmin toiminnasta (A = ”abcdbb”, B = ”cbacbaaba”):



Kaavio 1. Hirschbergin lineaarisen menetelmän etenemisen havainnollistaminen ja algoritmin B muodostamat taulukot. Taulukoita muodostettaessa muistissa on vain kaksi riviä ja tuloksena palautetaan vain viimeinen rivi, mutta taulukot ovat tässä kokonaisuudessaan havainnollisuuden parantamiseksi. Mustat nuolet muodostavat rekursiopuun. Katkaisukohtien numerot on lihavoitu.

Kaaviossa 1 on esitetty Hirschbergin lineaarisen menetelmän eteneminen. Aluksi algoritmille C annetaan syötteet ”abcdbb” ja ”cbacbaaba”. Koska syötteet eivät täytä vaiheen 1 ehtoja triviaalista ratkaisusta, jatketaan seuraavaan vaiheeseen. Ensimmäinen syöte jaetaan puoliksi kahdeksi kolmen pituiseksi syötteeksi. Syötteen A alkuosa ja syöte B lähetetään algoritmille B. Se käyttää merkkijonojen tutkimiseen samaa tapaa kuin luvun 2.2. Wagnerin ja Fischerin algoritmi sillä erotuksella, että vastaavaa ”taulukkoa C” ei ylläpidetä ollenkaan ja ”taulukosta B” ylläpidetään aina vain tarkasteltavaa sekä edellistä riviä. Tässä ei esitellä enempää algoritmin B toimintaa, koska vastaavan algoritmin toiminta on esitelty yksityiskohtaisesti luvussa 2.2. Lopulta algoritmi B palauttaa taulukon viimeisen rivin [0, 1, 1, 1, 2, 2, 2, 2, 2, 2], joka tallennetaan riviksi L1. Myös ensimmäisen sarakkeen arvo nolla tulee mukaan. Seuraavaksi algoritmille B lähetetään alkuperäisen merkkijonon A loppupuoli takaperin sekä

merkkijono B takaperin, eli algoritmi B saa syötteet ”bbd” ja ”abaabcabc”. Tätä käänteisten syötteiden lopputulosta [0, 0, 1, 1, 1, 2, 2, 2, 2, 2] kutsutaan riviksi L2. Nämä löytyvät myös kaaviosta 1 kahdesta ylimmästä taulukosta. Taulukot on esitetty kaaviossa 1 kokonaisuudessaan, vaikka ne eivät ole sellaisina ohjelman muistissa.

Vaiheessa 4 algoritmi C etsii merkkijonolle B parasta katkaisukohtaa. Se löytyy etsimällä suurinta rivien L1 ja L2 solujen summaa. Yhdistettäessä otetaan rivin L1 solujen arvot ja vastaavasti rivin L2 solujen arvot toisesta päästä lähtien, sillä tuossa taulukossa syötteet olivat takaperin. Matemaattisesti ilmaistuna etsitään suurin summan L1(j)+L2(n–j) arvo, kun j = 0, ..., n. Jos näitä suurimpia arvoja löytyy useampia, otetaan niistä ensimmäisenä löydetty katkaisukohdaksi. Rivien arvot yhdistettynä saadaan arvot {2, 3, 3, 3, 4, 3, 3, 3, 2, 2}. Näistä suurin on viidennessä paikassa oleva arvo 4. Koska molemmissa riveissä oli mukana myös yksi ylimääräinen nolla, ensimmäinen summa kuvaa B:n tyhjää alkuliitettä. Näin jälkimmäinen merkkijono katkaistaan neljännen merkin jälkeen. Tämä yhdistettyjen arvojen joukko ei ole tällaisena koneen muistissa suoritusvaiheessa, mutta se on lisätty osaan tämän luvun kappaleista ymmärtämisen ja helppolukuisuuden parantamisen vuoksi.

Oikean katkaisukohdan löytämisen jälkeen algoritmi C kutsuu itseään kahdella rekursiivisella kutsulla. Ensimmäiselle kutsulle annetaan syötteeksi alussa katkaistun A:n alkupuoli sekä merkkijonon B merkit L1:n ja L2:n avulla löydettyyn katkaisukohtaan asti. Kun rekursiiviset kutsut ovat palauttaneet merkkijonot, ne yhdistetään ja palautetaan lopputuloksena. Kaaviossa 1 rekursiivisten kutsujen polut on merkitty mustilla nuolilla. Oikeassa algoritmissa suoritus siirtyy aina ensimmäiseen rekursioon ja jälkimmäinen rekursiokutsu jää odottamaan alkupuolen valmistumista. Vaiheen 5 peräkkäiset rekursiokutsut esitetään rinnakkain kaaviossa 1, koska ne ovat selvemmin esitettävissä sillä tavalla.

Ensimmäisen rekursiivisen kutsun syötteet ovat ”abc” ja ”cbac”. Vaiheen 1 ehdot eivät toteudu. Ensimmäinen jaetaan kahteen osaan alaspäin pyöristäen, eli alkupuoliskoon jää vain yksi merkki. Tämä merkki lähetetään jälkimmäisen syötteen kanssa algoritmille B, joka palauttaa rivin L1 = [0, 0, 0, 1, 1]. Seuraavaksi algoritmille B lähetetään takaperin sekä syötteen A loppupuolisko että syöte B, eli syötteet ovat ”cb” ja ”cabc”. Niistä saadaan tuloksena rivi L2 = [0, 1, 1, 2, 2]. Kun L1:stä ja L2:sta etsitään maksimia, saadaan arvot {2, 2, 1, 2, 1}. Näistä suurimmista arvoista 2 ensimmäinen on heti paikassa nolla. Tämä tarkoittaa, että ensimmäisen rekursiivisen kutsun syötteet ovat ”a” ja tyhjä jono. Tämä palautuu heti tyhjänä merkkijonona, sillä syötteistä ei tietenkään voi löytyä yhteisiä merkkejä toisen ollessa tyhjä. Jälkimmäiselle rekursiiviselle kutsulle annetaan syötteet ”bc” ja ”cbac”.

Kun kahden pituinen syöte ”bc” jaetaan kahtia, saadaan kaksi yhden pituista syötettä. Algoritmi B:tä kutsutaan taas kahteen kertaan, syötteinä ovat ”b” ja ”cbac” sekä ”c” ja ”cabc”. L1:stä [0, 0, 1, 1, 1] sekä L2:sta [0, 1, 1, 1, 1] etsitään taas suurinta summaa, jolloin saadaan katkaisukohdaksi merkkijonon ”cbac” puoliväli. Ensimmäisen rekursiokutsun ensimmäinen syöte ”b” on yhden pituinen, joten merkkiä etsitään toisesta syötteestä ”cb”. Merkkijonoista löytyy yhteinen merkki, joten algoritmi C palauttaa tuon merkin rekursiopinossa yhtä tasoa ylemmäs. Myös jälkimmäisen rekursiokutsun ensimmäisen syötteen ”c” pituus on yksi ja jälkimmäisen syötteen ”ac” kanssa löytyy täsmäys, joten merkki palautetaan rekursiopinossa ylöspäin. Nämä kaksi saatua vastausta yhdistetään ja palautetaan taas tasoa ylemmäs. Nyt

alkuperäisen syötteen alkupuolen täsmäykset on etsitty ja palautetaan merkkijono ”bc” ensimmäisen algoritmin C kutsun paluuarvona.

Alkuperäisten syötteiden loppuosien täsmäysten etsiminen aloitetaan puolittamalla ensimmäinen syöte alaspäin pyöristäen. Algoritmille B annetaan syötteet ”d” ja ”baaba” sekä ”bb” ja ”abaab”. Riviksi L1 saadaan [0, 0, 0, 0, 0, 0], koska yhtään täsmäystä ei löydy. Rivin L2 arvoksi tulee [0, 0, 1, 1, 1, 2]. Kun ensimmäinen on nollarivi, etsittävä suurin arvo on tietysti L2:n viimeinen arvo. Näin ensimmäiseen rekursiiviseen kutsuun annetaan syötteiksi ”d” ja tyhjä jono, joten kutsu palauttaa tyhjän jonon. Jälkimmäinen rekursiivinen kutsu saa syötteikseen ”bb” ja ”baaba”.

Kahden merkin pituinen ensimmäinen syöte jaetaan puoliksi, minkä jälkeen algoritmille B lähetetään syötteet ”b” ja ”baaba” sekä ”b” ja ”abaab”. Tulosten L1 ja L2 suurinta summaa etsitään arvojen {1, 2, 2, 2, 1, 1} joukosta. Suurimmista arvoista 2 ensimmäinen esiintyy paikassa yksi, joten jälkimmäisen syötteen katkaisukohta on heti ensimmäisen merkin jälkeen. Viimeisille rekursiivisille kutsuille annetaan syötteiksi ”b” ja ”b” sekä ”b” ja ”aaba”. Molempien ensimmäiset syötteet ovat yhden pituisia ja syötteistä löytyvät täsmäykset, joten molemmat palauttavat merkin b. Nämä yhdistetään ja palautetaan ylemmäksi rekursiopinoon. Siellä palautettu tulos lisätään tyhjän jonon perään ja palautetaan tuloksena ensimmäiselle algoritmin C kutsulle. Se yhdistää saadut osaongelmien tulokset ”bc” ja ”bb”, minkä jälkeen se palauttaa tämän vastauksena pisimmän yhteisen alijonon ongelmaan.

Aikavaativuus:

Hirschbergin lineaarisen menetelmän lopulliseen aikavaativuuteen vaikuttaa kahden algoritmin aikavaativuus. Algoritmin B aikavaativuus on luokkaa O(mn), vastaavasti kuin Wagnerin ja Fischerin algoritmilla. Algoritmi C kutsuu itseään rekursiivisesti jaettuaan syötteet pienemmiksi palasiksi. Lopulta tämä ei kuitenkaan vaikuta lopulliseen aikavaativuusluokkaan (Hir 1975), joten algoritmin lopullinen aikavaativuus on myös luokkaa O(mn).

2.5. Hirschbergin menetelmä kaksisuuntaisena

Hirschbergin menetelmä (esitelty luvussa 2.3.) etenee vain yhteen suuntaan etsiessään pisintä yhteistä alijonoa. Tässä luvussa esitellään sama menetelmä muutettuna kaksisuuntaiseksi. Se ei ole Hirschbergin tekemä, vaan menetelmä on tehty tätä tutkielmaa varten Hirschbergin alkuperäistä versiota hyväksi käyttäen. Kaksisuuntaisuuden lisäksi muita suuria eroja ei ole, sillä algoritmin toimintalogiikka on pystytty pitämään käytännössä samana molempiin suuntiin edettäessä. Muistinkäytön kannalta kaksisuuntainen menetelmä ei eroa oleellisesti alkuperäisestä, sillä takaperin suoritettaessa käytetään hyväksi samoja merkkien esiintymätaulukoita sekä samaa D-taulukkoa hyvin pienillä muutoksilla. Toiseen suuntaan edettäessä tarvitaan myös vastaavat muuttujat kuin eteenpäin täsmäyksiä etsittäessäkin, mutta niiden tallentamiseen tarvittava tila on häviävän pieni taulukkoon D verrattuna.

Korkeuskäyriä etsitään vuorotellen vasemmalta ylhäältä ja oikealta alhaalta kuvitteellisesta A*B -taulukosta. Muuttujat pitävät huolen, että algoritmi etsii täsmäyksiä vain mahdollisilta alueilta. Syötejonoja eteenpäin prosessoivaa alkuperäistä algoritmia on muutettu siten, että se pitää myös uudet muuttujat ajan tasalla ja ottaa ne huomioon. Toiseen suuntaan eli ns.

takaperin tarkasteltava osuus on pyritty tekemään toimintalogiikaltaan täysin peilikuvaksi etuperin suoritettavaan osaan nähden. Taulukkoon PB tallennetaan kaikkien merkkien esiintymiskohtien perään vielä arvo n+1. Se vastaa takaperin kuljettaessa arvoa, joka etuperin prosessoitaessa on ensimmäisenä jokaisen merkin kohdalla oleva arvo nolla. Kun tulokset menosuuntaan etenevän osan kohdalta laitetaan taulukkoon D ylhäältä vasemmalta alkaen, paluusuuntaan suoritettava osa tallentaa tulokset alhaalta oikealta alkaen. Yhdestä tarkastelusuunnasta voi löytyä enintään puolet täsmäyksistä. Pahimmassa tapauksessa löytyy täsmäys jokaisen merkin kohdalta, jolloin molempien suuntien tulokset täyttävät molemmat enintään puolet taulukon D korkeudesta. Taulukon D tila riittää siis kahteen suuntaan etenevän algoritmin tarpeisiin.

Hirschbergin kaksisuuntaisen menetelmän pseudokoodi:

```
Hirsch2w(A,B):
1.  NB[θ] ← 0      for θ = 1, ..., s          s = merkistön viimeinen merkki
    PB[θ, 0] ← 0   for θ = 1, ..., s
    for j ← 1,  step = 1,  until n do
        NB[b_j] ← NB[b_j] + 1
        PB[b_j, NB[b_j]] ← j
    PB[θ, NB[θ] + 1] ← n + 1      for θ = 1, ..., s
    PB[0, 0] ← 0, PB[0, 1] ← n + 1          // ns. nollamerkit, joita ei löydy syötteestä B
2.  D[0, i] ←0 for i = 0, ..., m
    D[m + 1, i] ← n + 1 for i = 0, …, m
    lowcheck ← 0; highcheck ← m + 1
    k ← 0; FLAG ← 1; FLAG2 ← 1
    low ← 0; highlow ← n + 1
3.  while FLAG > 0 and FLAG2 > 0 {
        k ← k + 1
4.      N[θ] ← NB[θ] for θ = 1, ..., s
        N[0] ← 1                            // ns. nollamerkit, joita ei löydy syötteestä B
        FLAG ← 0; FLAG2 ← 0
        low ← D[k – 1, lowcheck]
5.      for i ← lowcheck + 1, step = 1, until highcheck – 1  do
6.          while PB[a_i, N[a_i] – 1] > low do N[a_i] ←  N[a_i] – 1
7.          if low < PB[a_i, N[a_i]] < highlow and PB[a_i, N[a_i]] < |D[m + 2 – k, i]|
                highlow ← PB[a_i, N[a_i]]
                D[k, i] ← highlow
                if FLAG = 0, then {lowcheck ← i; FLAG ← 1}
            else D[k, i] ← – |D[k, i – 1]|
8.          if D[k – 1, i] > 0 then low ← D[k – 1, i]
9.      if lowcheck < highcheck and FLAG > 0
            then etsi käyriä takaperin (vaiheet 4* – 8*)
```

```
4*          N[θ] ← 0 for θ = 1, ..., s
            high2 ← D[m – k + 2, highcheck – 1]
            N[0] ← 0                          // ns. nollamerkit, joita ei löydy syötteestä B
5*          for j ← highcheck – 1, step = – 1, until lowcheck + 1
6*              while PB[a_j, N[a_j] + 1] < high2 do N[a_j] ← N[a_j] + 1
7*              if highlow < PB[a_j, N[a_j]] < high2 and PB[a_j, N[a_j]] > |D[k, j – 1]|
                        highlow ← PB[a_j, N[a_j]]
                        D[m – k + 1, j] ← highlow
                        if FLAG2 = 0 then {highcheck ← j; FLAG2 ← 1}
                else D[m – k + 1, j] ← – | D[m – k + 1, j + 1] |
8*              if 0 < D[m – k + 2, j] < n + 1 then high2 ← D[m – k + 2, j]
    }
    sarake1 ← lowcheck; sarake2 ← highcheck
    pisinalijono = ""
    if FLAG = 0 then
        k ← k – 1
        if k = 0 then return tyhjä merkkijono
        rivi ← m – k + 1
        while D[rivi,sarake2] ≤ D[k, sarake1] do sarake1 ← sarake1 + 1
    else
        if k = 1 then return A[lowcheck]
        rivi ← m – k + 2
        while D[k,sarake1] ≥ D[rivi, sarake2] do sarake2 ← sarake2 – 1
10. for j ← sarake1, step = – 1, until 0 do
        if D[k, j] > 0
            pisinalijono ← A[j] + pisinalijono
            k ← k – 1
10* for j ← sarake2, step = 1, until m do
        if D[rivi, j] > 0
            pisinalijono ← pisinalijono + A[j]
            rivi ← rivi + 1
    return LCS
```

Alkuperäiseen Hirschbergin algoritmiin verrattuna uusia muuttujia ovat highcheck, high2 ja FLAG2. Alkuperäinen muuttuja high korvataan muuttujalla highlow, sillä sitä käytetään etuperin kuljettaessa kuten vanhaa high:ta, ja takaperin prosessoitaessa se toimii low:n kaltaisesti. Tämä tarkoittaa sitä, että molempien suuntien etsinnät käyttävät ja päivittävät samaa muuttujaa. Muuttuja highcheck on viimeisimmän takaperin löydetyn korkeuskäyrän ensimmäisen täsmäyksen kohta taulukossa D, eli vastaava kuin lowcheck, mutta päinvastaisesta suunnasta katsottuna. Muuttuja high2 on täysin vastaava kuin low, eli takaperin etsittäessä täsmäysten on löydyttävä highlow:n ja high2:n välistä, highlow≤high2. FLAG2 on toinen lippu, joka pitää muistissa tietoa, onko kyseiselle takaperin etsittävälle korkeuskäyrälle löytynyt vielä yhtään täsmäystä. Jos molempien lippujen arvo on 1, molempiin suuntiin etsittäessä on löytynyt korkeuskäyrä, joten suoritetaan uusi etsintävaihe menosuunnasta aloittaen. Jälkimmäisen lipun arvon ollessa nolla uutta korkeuskäyrää ei ole

löytynyt paluusuuntaan sallitulta alueelta, joten voidaan selvittää lopputulos taulukosta D. Myös mikäli ensimmäisen lipun arvo on nolla, toiseenkaan suuntaan etsiessä ei voi enää löytyä uutta korkeuskäyrää. Korkeuskäyriä voi olla joko parillinen tai pariton määrä, eli viimeisen käyrän voi löytää kumpaan tahansa suuntaan etenevä osio. Tämä vaikuttaa pisimmän yhteisen alijonon laskemiseen, joten se on otettu huomioon algoritmissa.

Edellä esitetyssä pseudokoodissa huomionarvoista on se, että taulukon D indeksoinnit alkavat nollasta. Toinen erityishuomiota vaativa juttu on ns. nollamerkit, joita ei löydy jälkimmäisestä syötteestä. Nämä merkit kuuluvat tietysti merkistöön, joka sisältää ainakin kaikki syötteissä A ja B esiintyvät merkit. Näiden kohdalla lukumääräksi alustetaan 1 taulukkon NB, jotta algoritmi toimisi. Nollamerkkien alustukset ovat pseudokoodissa vielä erillisinä muistuttamassa tästä. Muuttuja highcheck ilmoittaa sarakkeen taulukossa D. Taulukossa D alhaaltapäin tarkasteltaessa oikea sarake jätetään tyhjäksi, joten highcheck tarkoittaa merkkiä $A_{highcheck + 1}$.

Hirschbergin algoritmi kaksisuuntaisena alkaa samalla tavalla kuin yksisuuntainenkin, johon tehdään muutama pieni lisäys. Vaiheessa 1 lisätään taulukkoon PB jokaisen merkin kohdalle nolla, minkä jälkeen käydään merkkijono B läpi merkki kerrallaan lisäämällä kyseisen merkin lukumäärää taulukossa NB sekä paikan taulukkoon PB. Taulukkoon PB lisätään tämän jälkeen myös arvo n+1 jokaisen merkin kohdalle. Vaiheessa 2 taulukko D:hen lisätään nollanneksi riviksi nollarivi sekä alustetaan muuttujat lowcheck, highcheck, k, FLAG, FLAG2 ja highlow. Loput tarvittavat muuttujat alustetaan vaiheessa 4, eli ennen kuin niitä tarvitaan vaiheessa 5. Vaiheessa 3 muutetaan etsittävän korkeuskäyrän numero oikeaksi, jos molempien lippujen arvo on nollaa suurempi. Tämä korvaa osittain myös alkuperäisessä yksisuuntaisessa versiossa olleen vaiheen 9 lipun arvon tarkastuksen. Vaiheessa 2 alustettiin molempien lippujen arvoksi 1, jotta algoritmi pääsee ensimmäiselle etsintäkierrokselle. Kun muuttuja k:n arvo on 1, se tarkoittaa, että etuperin etsitään ensimmäistä korkeuskäyrää ja takaperin viimeistä korkeuskäyrää. Vaiheessa 4 tehdään joka kierroksella tarvittavat muuttujien alustukset. Taulukkoon N kopioidaan arvot taulukosta NB, liput nollataan, sekä muuttujalle low etsitään oikea arvo nykyisen korkeuskäyrän etsimistä varten.

Etuperin tapahtuvan etsinnän vaiheille 4–8 ja 10 on tehty vastaavat vaiheet 4*–8* ja 10* takaperin suuntautuvalle etsinnälle. Vaiheessa 5 määritellään, mille merkkijonon A merkille lähdetään etsimään täsmäyksiä. Muuttuja lowcheck kertoo viimeksi etuperin löydetyn korkeuskäyrän ensimmäiseksi löydetyn täsmäyksen ja highcheck vastaavasti viimeksi takaperin löydetyn korkeuskäyrän ensimmäiseksi löydetyn täsmäyksen rivinumeron. Koska etsittävän täsmäyksen täytyy olla eteenpäin löydettyjen ja takaperin löydettyjen korkeuskäyrien välillä, täsmäyksen kohdan täytyy olla suurempi kuin lowcheck ja pienempi kuin highcheck. Vaiheen 6 toiminnot ovat samat kuin alkuperäisessä Hirschbergin algoritmissa, eli etsitään viimeistä low:ta suurempaa arvoa taulukon PB lopusta lähtien. Seuraavat vaiheet tarkistavat, löytyikö uusi korkeuskäyrälle k kelpaava minimaalinen täsmäys.

Vaiheeseen 7 on tullut ehtolauseeseen yksi lisäys alkuperäiseen versioon. Ei ole järkevää etsiä täsmäyksiä moneen kertaan koko alueelta, joten mahdollisen löydettävän täsmäyksen täytyy olla viimeksi takaperin löytyneen korkeuskäyrän vasemmalla puolella. Tämä tieto löydetään taulukosta D. Kun sallittua täsmäystä ei löydy, alkuperäinen algoritmi lisää taulukkoon D nollan. Tämä kaksisuuntainen algoritmi lisää vastaavassa tilanteessa vastaavaan kohtaan

edellisen samalta käyrältä löydetyn täsmäyskohdan arvon negatiivisena. Negatiivinen arvo taulukossa D kertoo, missä kohdassa korkeuskäyrä kulkee sillä rivillä. Eteenpäin etsittäessä täsmäyksen paikan täytyy olla pienempi kuin taulukosta löytyvän täsmäyksen kohdan itseisarvo, jotta se olisi viimeksi löytyneen takaperin etsityn korkeuskäyrän vasemmalla puolella. Ensimmäisellä kierroksella algoritmi etsii arvoja D:n alimmalta riviltä, jonka jokaisen solun arvo on n+1, joten kaikki löydetyt arvot kelpaavat.

Ehtojen toteutuminen vaiheessa 7 tarkoittaa korkeuskäyrälle k kelvollisen minimaalisen täsmäyksen löytymistä. Muuttuja highlow:lle annetaan täsmäyksen sarakenumero, jotta seuraavaa saman korkeuskäyrän täsmäystä etsitään nyt löydetyn vasemmalta puolelta. Samalla myös taulukkoon D päivitetään sama arvo. Jos tämä oli kyseisen korkeuskäyrän ensimmäinen löydetty täsmäys, asetetaan lipun arvoksi 1, ja lowcheckiin laitetaan muistiin, mistä kohdasta merkkijonosta A käyrän ensimmäinen täsmäys löytyi. Jos täsmäystä ei löytynyt, laitetaan taulukkoon D edellisen löytyneen täsmäyksen kohdan arvo negatiivisena tai nolla, jos täsmäyksiä ei ole vielä löytynyt.

Vaiheessa 8 tarkastetaan, löytyikö edelliseltä korkeuskäyrältä samasta kohdasta täsmäystä. Jos sellainen löytyi, päivitetään low:n arvoa siten, että seuraava nyt etsittävän käyrän piste on edellisen oikealla puolella. Kun korkeuskäyrä on löytynyt kokonaisuudessaan, vaihe 9 testaa, kannattaako etsiä toiseen suuntaan. Jos highcheck ei ole suurempi kuin lowcheck tai ensimmäinen lippu on nolla, täsmäyksiä ei voi löytyä, joten takaperin etsintää ei tarvitse enää suorittaa.

Takaperin tapahtuva korkeuskäyrien etsiminen alkaa vaiheessa 4* päivittämällä taulukon N kaikki arvot nolliksi. Tämä on täysin vastaava vaihe peilikuvana kuin etuperin edetessä arvojen alustaminen takaisin maksimiarvoonsa. Muita muuttujia ei tarvitse tässä kohtaa päivittää. Vaihe 5* on peilikuva vaiheelle 5, eli siinä määritetään merkkijonon A merkki, jolle etsitään täsmäyksiä. Takaperin etsittäessä merkit valitaan viimeisestä alkuun päin edeten. Vaiheen 6* toiminta vastaa etuperin etsimisen vaihetta 6. Tässä tapauksessa taulukosta PB yritetään etsiä oikean merkin kohdalta suurinta arvoa, joka on pienempi kuin muuttuja high2.

Vaiheen 7* tehtävänä on muokata taulukkoa D sekä muuttujia sen mukaan, onko vaiheessa 6* löydetty arvo minimaalinen täsmäys vai ei. Jos löydetty arvo on highlow:n ja high2:n välissä sekä se sijaitsee eteenpäin löydetyn korkeuskäyrän oikealla puolella, on löytynyt minimaalinen täsmäys. Arvo päivitetään muuttujaan highlow ja se lisätään oikealle paikalle taulukkoon D. Jos löydetty täsmäys oli kyseisen korkeuskäyrän ensimmäinen, muutetaan toisen lipun arvoksi 1 ja päivitetään highcheckin arvoksi kyseisen merkin esiintymiskohta. Jos vaiheen 7* ehdot eivät täyty, kyseiselle merkkijonon A merkille ei löydy ainakaan minimaalista täsmäystä. Tämän vuoksi taulukkoon D lisätään mahdollisen edellisen takaperin löydetyn täsmäyksen paikan arvo negatiivisena, aivan kuten eteenpäin etsittäessäkin.

Jos edelliseltä takaperin etsityltä korkeuskäyrältä on löytynyt saman merkin kohdalta täsmäys, päivittää vaihe 8* muuttujan high2 arvoa siten, että se etsii seuraavaa täsmäystä oikealta väliltä. Vaihetta 9* ei ole olemassa, sillä vaiheen 3 ehtolauseet testaavat, edetäänkö seuraavalle kierrokselle. Jos molempiin suuntiin on löytynyt uusi korkeuskäyrä, jatketaan etsimistä, muuten siirrytään kohtiin 10 ja 10* keräämään pisintä yhteistä alijonoa.

Koska tämä kaksisuuntainen etsiminen etsii korkeuskäyriä vuorotellen molemmista suunnista, se löytää vain n. puolet korkeuskäyristä molemmista suunnista katsottuna. Tämän vuoksi viimeksi löydetyiltä molempien suuntien korkeuskäyriltä voi löytyä useita täsmäyksiä. Jotta on voitu löytää uusi käyrä, sen sisältämien minimaalisten täsmäysten on jokaisen jatkettava vähintään yhtä viimeisimmän toiseen suuntaan löytyneen käyrän pisteistä. Tämän tiedon avulla voidaan selvittää kaksi minimaalista täsmäystä – yksi molemmilta käyriltä – joiden avulla saadaan selvitettyä sekä alku- että loppupään alijonoon kuuluvat merkit. Jos yhtään täsmäystä ja käyrää ei ole löytynyt, algoritmi palauttaa tyhjän alijonon ennen vaihetta 10. Jos on löytynyt vain yksi käyrä, palautetaan minimaalisen täsmäyksen merkki ennen vaihetta 10.

Ennen vaiheen 10 suorittamista täytyy selvittää, onko käyriä parillinen vai pariton määrä. Jos lipun arvo on nolla, on korkeuskäyriä parillinen määrä, ja vaiheessa 3 on k:n arvoa lisätty yhden kerran ”liikaa”. Tämän vuoksi k:n arvosta vähennetään yksi, jotta tuloksia katsotaan oikealta riviltä taulukosta D. Parillinen korkeuskäyrien määrä tarkoittaa, että viimeksi on löydetty käyrä takaperin etsien. Tämän käyrän ensimmäisen minimaalisen täsmäyksen on täytynyt siis jatkaa jotain viimeisen etuperin löytyneen korkeuskäyrän minimaalisista täsmäyksistä. Kun tämä sopiva täsmäys on löydetty, sen avulla vaiheessa 10 selvitetään etuperin löytyneistä käyristä alijono samalla tavalla etenemällä taulukossa vasemmalle ja ylös kuin alkuperäisessäkin Hirschbergin algoritmissa (esitelty luvussa 2.3). Takaperin löydettyjen käyrien alijono saadaan vaiheessa 10* selville täysin vastaavasti peilikuvana etenemällä taulukossa D oikealle ja alas. Jos käyriä on taas pariton määrä, viimeksi etuperin löydetyn korkeuskäyrän ensimmäinen minimaalinen täsmäys on jatkanut jotain viimeksi takaperin löydetyn korkeuskäyrän minimaalisista täsmäyksistä (eli on löytynyt täsmäys takaperin löydetyn käyrän etupuolelta siten, että viimeksi takaperin löydetyn käyrän jokin täsmäyksistä pystyy jatkamaan etuperin löydetyn käyrän alijonoa). Pisimmän yhteisen alijonon muodostaminen tapahtuu täysin samalla tavalla vaiheissa 10 ja 10* kuin parillisessakin tapauksessa.

Esimerkki algoritmin toiminnasta (A = ”abcdbb”, B = ”cbacbaaba”):

Taulukko 6. Merkkijonon B merkkien esiintymiskohdat löytyvät taulukosta PB.

a	0	3	6	7	9	10
b	0	2	5	8	10	
c	0	1	4	10		
d	0	10				

Taulukko 7. Merkkijonon B eri merkkien lukumäärät taulukossa NB sekä taulukossa N sen alustamisen jälkeen vaiheessa 4. Merkki d on ns. nollamerkki, jota ei esiinny merkkijonossa B, joten sen arvoksi on alustettu 1.

a	4
b	3
c	2
d	1

Taulukko 8. Kaksisuuntaisen Hirschbergin algoritmin tulokset syötteillä ”abcdbb” ja”cbacbaaba” ovat taulukossa D. Ylhäältä päin tarkastellessa rivinumero tarkoittaa korkeuskäyrää k ja sarakkeen numero merkin esiintymiskohtaa i merkkijonossa A. Positiivinen arvo solussa D[k, i] tarkoittaa käyrän k minimaalista täsmäystä, jossa solun arvo on merkin sijainti merkkijonossa B ja sarakkeen numero on merkin sijainti A:ssa. Alhaalta päin tarkastellessa korkeuskäyrät löytyvät riviltä 6 ylöspäin viimeisimmästä käyrästä alkaen. Oikeanpuolimmainen sarake on tyhjä, aivan kuten ylhäältä tarkasteltaessa vasemmanpuoleisin sarake. Tällöin taulukon alaosaa tarkasteltaessa sarakkeen voidaan katsoa tarkoittavan merkkijonon A merkkiä siten, että numerointi alkaa nollasta; ts. sarake 1 vastaa syötteen A toista merkkiä. Positiivinen arvo alhaalta päin lukien solussa D[k, i] tarkoittaa (m–k+1). käyrää alhaalta päin, jossa solun arvo on merkin sijainti B:ssä ja i+1 on merkin sijainti A:ssa.

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1		3	2	1	–1	–1	–1
2			5	4	–4	2	
3				0	0		
4							
5			–5	–5	5		
6		–8	–8	–8	–8	8	
7	10	10	10	10	10	10	10

Taulukoissa 6, 7 ja 8 on esitetty kaksisuuntaisessa Hirschbergin algoritmissa käytettävät taulukot samoilla aikaisemmissa luvuissa käytetyillä syötteillä. Taulukko 7, joka esittää taulukkoa NB sekä taulukkoa N vaiheen 4 alustuksen jälkeen, on täysin samanlainen kuin luvussa 2.3. esitetty vastaava taulukko 5. Taulukossa 6 on luvun 2.3. taulukkoon verrattuna jokaisen rivin viimeisenä arvona n+1. Taulukko 8 sen sijaan eroaa yhteen suuntaan etenevän algoritmin taulukosta enemmän. Siinä ylhäältä alaspäin on merkitty etuperin suoritettavan osan löytämät täsmäykset ja vastaavasti alhaalta ylöspäin on merkitty takaperin prosessoitavan osan löytämät täsmäykset. Etuperin kuljettaessa taulukon nollas sarake on tyhjä molemman suunnan aloitusrivejä lukuun ottamatta, vastaavasti takaperin etsittäessä sarake 6 on tyhjä pl. ylin ja alin rivi. Taulukon alaosan luvut ovat siis yhden pykälän verran vasemmalle, eli taulukon sarake 5 tarkoittaa merkkijonon A kuudetta merkkiä.

Annetaan Hirschbergin kaksisuuntaiseksi muutetulle algoritmille syötteet ”abcdbb” sekä ”cbacbaaba”. Vaiheessa 1 algoritmi alustaa ensimmäiseksi tarvittavat alkuarvot taulukoihin NB, PB ja D. Taulukkoon D tulee alkuperäisen ylimmän nollarivin lisäksi alimmalle riville jokaiseen soluun arvo n+1 eli 10. Kun syöte B on käyty läpi, merkkien määrät ovat taulukossa NB ja merkkien paikat on lisätty taulukkoon PB, PB:hen lisätään vielä jokaisen merkin kohdalle arvo n+1, joka on tässä esimerkissä 10. Vaiheessa 2 alustetaan muuttujat ensimmäistä etsintäkierrosta varten. Vaihe 3 muuttaa k:n arvon 1:ksi, kun etsitään ensimmäistä sekä viimeistä korkeuskäyrää. Vaiheen 4 alustukset nollavat lippujen arvot, muuttavat muuttujan low arvon oikeaksi sekä kopioivat maksimiarvot taulukosta NB taulukkoon N.

Vaiheessa 5 määritetään, että täsmäyksiä lähdetään etsimään merkkijonon A ensimmäiselle merkille. Vaiheessa 6 etsitään pienintä low:ta suurempaa arvoa merkin a kohdalta taulukosta PB. Pienin low:ta suurempi arvo on 3, joka toteuttaa vaiheen 7 ehdot. Löydetty arvo on siis minimaalinen täsmäys, joten päivitetään muuttuja highlow sekä laitetaan arvo taulukkoon D. Koska tämä oli ensimmäisen korkeuskäyrän ensimmäinen täsmäys, päivitetään myös lipun ja lowcheckin arvot. Vaiheen 8 ehto ei täyty, sillä edellistä korkeuskäyrää ei ole olemassa. Tämän jälkeen vaihe 5 vaihtaa tarkasteltavan merkin seuraavaan. Merkille b löytyy vaiheessa 6 pienin low:ta suurempi arvo 2. Tämä toteuttaa vaiheen 7 ehdot, joten täsmäyksen kohta merkitään taulukkoon D ja highlow päivitetään. Vaiheen 8 ehto ei täyty tälläkään kerralla.

Seuraavana on vuorossa merkkijonon A kolmannen merkin täsmäysten etsiminen. Vaiheessa 6 löydetään pienin low:ta suurempi arvo 1. Tämä osuu low:n ja highlow:n väliin vaiheen 7 ehdoissa ja on taulukon oikean reunan vasemmalla puolella. Arvo laitetaan taulukkoon D oikealle paikalleen ja highlow päivitetään. Vaihe 8 ei voi vieläkään toteutua. Seuraavaksi vaihe 5 päivittää etsittäväksi merkiksi neljäntenä olevan merkin d. Vaiheen 6 pienin low:ta suurempi arvo on 10, joka ei täytä vaiheen 7 ehtoja. Jos täsmäyksistä tekisi taulukon 1 mallisen taulukon, korkeuskäyrä jatkaisi suoraan alaspäin rivin 4 yli, koska tälle merkille ei löytynyt sopivaa täsmäystä. Tämän vuoksi taulukkoon D laitetaan kyseisen korkeuskäyrän viimeksi löytyneen täsmäyksen arvo negatiivisena, eli arvo –1. Vaiheen 8 ehto ei toteudu.

Seuraavana vuorossa on viides merkki b. Vaiheessa 6 etsitään pienintä low:ta suurempaa arvoa lähtien arvosta 2, joka jää myös etsityksi arvoksi. Vaiheessa 7 se ei kuitenkaan täytä enää ehtoja, joten lisätään edellisen löydetyn täsmäyksen kohta negatiivisena. Otetaan edellisestä arvosta –1 itseisarvo negatiivisena ja asetetaan se taulukkoon D, eli laitetaan tähän sama arvo kuin viereisessäkin solussa. Vaihe 8 mennään taas ohi tekemättä mitään. Viimeinenkin merkki on b aivan kuten edeltävä merkkikin, joten tässä kohdassa tehdään täysin samat operaatiot. Pienin low:ta suurempi arvo 2 ei tälläkään kertaa täytä vaiheen 7 ehtoja, joten laitetaan taulukkoon D viereisen solun itseisarvo negatiivisena, eli –1. Vaiheen 8 ehto ei toteudu. Enää ei mennä vaiheen 5 sisälle, sillä kaikki halutun välin merkit on nyt käyty läpi. Vaiheen 9 tarkastuksessa todetaan, että lowcheck on pienempi kuin highcheck ja lipun arvo on 1, joten toisesta suunnasta on mahdollista löytää uusi korkeuskäyrä. Tämän jälkeen siirrytään etsimään uusia korkeuskäyriä takaperin syötteiden lopusta lähtien.

Viimeisen korkeuskäyrän etsiminen aloitetaan vaiheesta 4* asettamalla taulukkoon N jokaisen merkin kohdalle arvo nolla. Myös muuttuja high2 päivitetään, minkä jälkeen siirrytään vaiheeseen 5*. Tarkasteltavaksi merkiksi otetaan merkkijonon A viimeinen merkki. Vaiheessa 6* etsitään suurinta high2:a pienempää arvoa merkin b kohdalta kasvattamalla taulukon N arvoa nollasta lähtien. Arvoksi tulee 8, joka on vaiheen 7* ehtojen mukainen. Se on suurempi kuin highlow, pienempi kuin high2 sekä sijaitsee etuperin löydetyn korkeuskäyrän oikealla alapuolella. Ehtojen täyttyminen tarkoittaa minimaalisen täsmäyksen löytymistä, joten päivitetään muuttuja highlow. Taulukon D alaosaa tarkasteltaessa sarake 6 jätetään tyhjäksi, joten laitetaan arvo 8 taulukkoon D oikealle paikalleen toiseksi viimeiselle riville toiseksi viimeiseen sarakkeeseen. Toiseksi viimeinen rivi tarkoittaa lopusta päin ensimmäistä etsittävää korkeuskäyrää. Soluja täytetään aloittaen oikealta ja edeten vasemmalle, sillä merkkejä käydään läpi viimeisistä alkuun päin. Päivitetään myös toinen lippu ja highcheck korkeuskäyrän ensimmäisen täsmäyksen löytymisen vuoksi. Vaiheen 8* tarkastelu etsii edellistä takaperin

etsittyä korkeuskäyrää, mutta sellaista ei löydy. Koska alimman rivin arvot ovat liian isoja vaiheen 8* ehdoille, tässä vaiheessa ei tehdä mitään.

Vaihe 5* muuttaa etsittävän merkin toiseksi viimeiseksi merkiksi, joka on myös b. Vaiheen 6* suurin high2:ta pienempi arvo on jo valmiina, sillä edellinen käsiteltävä merkki oli sama. Nyt tämä arvo ei kuitenkaan ole enää suurempi kuin päivitetty highlow, joten uutta täsmäystä ei löydy. Taulukkoon D lisätään nyt riviä alempana oleva kyseisen korkeuskäyrän kohta negatiivisena. Tämä saadaan ottamalla viereisen solun täsmäyksen kohdasta itseisarvo negatiivisena. Vaiheen 8* ehdot eivät taaskaan toteudu. Seuraavaksi vaihe 5* muuttaa tarkasteltavaksi merkiksi kolmanneksi viimeisen merkin d. Sille ei löydy täsmäyksiä, joten lisätään taulukkoon D taas arvo –8. Edellistä käyrää ei ole, joten vaihe 8* ohitetaan muuttamatta mitään.

Vaihe 5* muuttaa seuraavaksi tarkasteltavaksi merkiksi A:n kolmannen merkin c. Vaiheessa 6* löytyvä suurin high2:ta pienempi arvo on 4. Se ei kuitenkaan täytä vaiheen 7* ehtoja, joten sallittua täsmäystä ei löydy. Taulukkoon D laitetaan jälleen arvo –8. Vaiheen 8* ehdot jäävät toteutumatta, joten seuraava tarkasteltava merkki on b. Edellisten vaiheiden 6* operaatioiden jäljiltä suurin high2:ta pienempi on jo valmiina, mutta se ei täytä kohdan 7* ehtoja. Taulukkoon D tulee taas –8 eivätkä vaiheen 8* ehdot täyty tälläkään kerralla. Vaiheen 5* sisälle ei enää mennä, sillä muuttujan j arvo ei ole enää suurempi tai yhtä suuri kuin lowcheck. Tämä tarkoittaa, että tämän vasemmalta yläpuolelta ei voi enää löytyä täsmäyksiä, jotka olisivat eteenpäin löytyneiden käyrien oikealla puolella. Tämän vuoksi merkkijonon A ensimmäistä merkkiä ei käsitellä.

Kun molempiin suuntiin etsittäessä on löydetty korkeuskäyrät, molempien lippujen arvo on 1. Vaiheessa 3 k:n arvoa kasvatetaan, kun aletaan etsimään toisia käyriä molemmista suunnista. Vaiheessa 4 alustetaan taas muuttuja low, molemmat liput sekä taulukko N. Eteenpäin tapahtuvaa etsintä aloitetaan vaiheesta 5, jossa määritetään ensimmäiseksi tarkasteltavaksi merkiksi merkkijonon A toinen merkki. Vaiheessa 6 löydetään pienin low:ta suurempi arvo 5. Tämä arvo on low:n ja highlow:n välissä sekä edellisen takaperin löytyneen käyrän vasemmalla puolella, joten se toteuttaa vaiheen 7 ehdot. Päivitetään täsmäyksen löytymisen vuoksi muuttuja highlow sekä lisätään taulukkoon D arvo 5 kohtaan [2, 2]. Muutetaan myös lipun arvoksi 1 ja lowcheckin arvoksi 2, koska tämä oli kyseisen korkeuskäyrän ensimmäisen löydetyn täsmäyksen rivi. Vaiheessa 8 etsitään edellisen eteenpäin etsityn korkeuskäyrän täsmäyksiä samasta kohdasta. Sellainen löytyy, joten muutetaan edellisen käyrän täsmäyksen kohta low:n arvoksi.

Seuraavaksi vaiheessa 5 siirrytään A:n kolmanteen merkkiin. Vaiheessa 6 etsitään taas pienintä low:ta suurempaa täsmäyskohtaa. Arvoksi saadaan 4, joka on suurempi kuin low:n arvo 2, pienempi kuin highlow:n arvo 5 ja sijaitsee viimeksi takaperin löydetyn korkeuskäyrän vasemmalla puolella. Vaiheen 7 ehdot täyttyvät, joten päivitetään highlow ja lisätään arvo taulukkoon D. Edelliseltä korkeuskäyrältä löytyi täsmäys tältä riviltä, joten muutetaan low:n arvoa. Seuraavalle merkille d ei taaskaan löydy täsmäyksiä, joten kopioidaan taulukkoon D edellisen löytyneen täsmäyksen arvo negatiivisena. Myöskään edelliseltä korkeuskäyrältä ei löytynyt täsmäyksiä, joten vaiheessa 8 ei muuteta mitään.

Seuraavaksi tarkasteltavaksi merkiksi tulee A:n viides merkki vaiheessa 5. Vaiheessa 6 etsitään taas pienintä low:ta suurempaa arvoa. Muuttujan low arvo on pienempi kuin edellisellä kerralla merkin b kohdalta tarkasteltaessa. Nyt arvoksi tulee 2, joka täyttää vaiheen 7 ehdot. Tämän vuoksi päivitetään highlow ja kopioidaan arvo taulukkoon D. Vaihe 8 etsii edellisen korkeuskäyrän täsmäystä samalle merkkijonon A merkille, mutta sellaista ei ole. Vaihe 5 ei enää etene kuudenteen merkkiin, sillä se ei voi enää löytää sallittuja täsmäyksiä siltä alueelta. Vaiheessa 9 testataan ensin, onko jäljellä aluetta, josta takaperin etsivä osio voisi löytää lisää korkeuskäyriä. Tämä ehto toteutuu kuten lipun arvon ehtokin, joten algoritmi siirtyy etsimään seuraavaa käyrää.

Vaiheessa 4* muutetaan taas taulukon N arvot nolliksi ja alustetaan muuttuja high2. Vaihe 5* tekee A:n viidennestä merkistä tarkasteltavan merkin. Vaihe 6* etsii tämän merkin kohdalta suurinta high2:ta pienempää arvoa taulukosta PB. Löydetty arvo 5 täyttää vaiheen 7* ehdot, joten se on toisen takaperin löydetyn korkeuskäyrän ensimmäinen täsmäys. Päivitetään highlow, laitetaan arvo taulukkoon D sekä päivitetään lipun arvo ja highcheck. Edelliseltä takaperin löydetyltä korkeuskäyrältä ei löytynyt tämän merkin kohdalta sallittua täsmäystä, joten vaiheessa 8* ei tehdä mitään. Seuraava käsiteltävä merkki on taas d, jolle ei löydy täsmäyksiä. Kopioidaan edellisen löytyneen täsmäyksen kohdan itseisarvo negatiivisena viereisestä solusta. Edellisen käyrän täsmäyksiä ei löydy, joten siirrytään seuraavaan merkkiin. Seuraava käsiteltävä merkki on A:n kolmas merkki c. Vaihe 6* löytää arvon 4. Se ei täytä vaiheen 7* ehtoja, sillä highlow:n arvo 5 on suurempi kuin löydetty arvo. Laitetaan taulukkoon D taas arvo – 5. Vaiheen 8* ehto ei toteudu. Vaiheessa 5* ei mennä enää seuraavaan merkkiin, sillä ehdot eivät enää täyty.

Molempien lippujen arvo on yksi, eli vaiheessa 3 edetään kolmannelle kierrokselle. Muuttujan k arvoksi tulee kolme, kun etsitään mahdollista kolmatta korkeuskäyrää molemmista suunnista. Vaiheessa 4 alustetaan taas tarpeelliset muuttujat, liput sekä taulukko N. Vaihe 5 määrittää ensimmäiseksi tarkasteltavaksi merkiksi A:n kolmannen merkin. Vaihe 6 löytää pienimmän low:ta suurimman arvon 10, joka ei kuitenkaan täytä vaiheen 7 ehtoja. Koska käsiteltävälle korkeuskäyrälle ei ole vielä löytynyt täsmäyksiä, tulee taulukkoon D arvo 0. Edelliseltä käyrältä löytyi tämän merkin kohdalta täsmäys, joten low:n arvoksi muutetaan 4. Seuraavaksi etsittäväksi merkiksi tulee taas d vaiheessa 5. Sille ei löydy vieläkään täsmäystä, joten tässäkin kohdassa taulukkoon D tulee arvo 0. Vaiheen 8 ehto ei täyty. Seuraavaan merkkiin ei enää mennä vaiheessa 5, sillä reunaehdot eivät enää täyty. Toinen vaiheen 9 ehdoista ei enää täyty, joten toiseen suuntaan ei enää etsitä uutta korkeuskäyrää.

Molempien lippujen arvot ovat nollia, joten vaiheen 3 sisälle ei enää mennä. Taulukko D on nyt saatu valmiiksi, ja seuraavana on vuorossa pisimmän yhteisen alijonon etsiminen sen avulla. Ennen vaiheeseen 10 siirtymistä alustetaan muuttujia, tutkitaan käyrien lukumäärän parillisuus sekä etsitään kaksi sopivaa täsmäystä. Otetaan käyttöön muuttuja pisinalijono, johon tallennetaan löydetyt merkit. Uudet muuttujat sarake1 ja sarake2 alustetaan arvoilla lowcheck ja highcheck. Lipun arvo on 0, joten viimeinen korkeuskäyrä on löytynyt takaperin etsimällä, ja k:n arvoa on kasvatettu kerran liikaa. Tämän vuoksi k:n arvoa vähennetään yhdellä. Muuttuja k ei ole nolla, joten ei palauteta vielä mitään. Muuttujaan rivi asetetaan viimeksi takaperin löytyneen korkeuskäyrän rivinumero 5. Etsitään taulukosta D riviltä k sellaista minimaalista täsmäystä, jota viimeisen takaperin löydetyn käyrän ensimmäinen minimaalinen täsmäys

jatkaa. Etsintä aloitetaan sarakkeesta lowcheck, josta löytyy arvo 5. Se on yhtä suuri kuin vertailuarvo, joten se ei kelpaa. Etsintä jatkuu kasvattamalla yhdellä muuttujaa sarake1. Löytyy arvo 4, joka kelpaa. Nyt voidaan selvittää pisin alijono vaiheissa 10 ja 10*.

Ensin etsitään etuperin löydettyjen täsmäysten merkit aloittaen korkeuskäyrän k täsmäyksestä, joka etsittiin ennen vaihetta 10. Aloitussolusta löytyy siis nollaa suurempi arvo, joten lisätään täsmäyksen merkki c muuttujaan pisinalijono ja siirrytään yhtä riviä ylemmäs pienentämällä muuttujan k arvoa. Seuraavaksi siirrytään yksi sarake vasemmalle, josta löytyy seuraava täsmäys. Lisätään merkki b muuttujan pisinalijono ensimmäiseksi merkiksi. Enempää täsmäyksiä ei löydy etuperin löydetyiltä korkeuskäyriltä, joten siirrytään vaiheeseen 10*. Muuttujien rivi ja sarake2 avulla saadaan solu, josta lähdetään etsimään loppupään merkkejä. Tästä solusta löytyy täsmäys, joten lisätään merkki b muuttujaan pisinalijono ja kasvatetaan muuttujaa rivi. Siirrytään yksi sarake oikealle, josta löytyy taas täsmäys. Kyseisen täsmäyksen merkki b lisätään muuttujan pisinalijono viimeiseksi ja edetään seuraavalle riville. Nyt tuli vastaan rivi, jolta ei enää löydy täsmäyksiä, joten algoritmi on valmis palauttamaan muistissa olevan pisimmän yhteisen alijonon ”bcbb”.

Aikavaativuus:

Algoritmin aikavaativuus on samaa luokkaa kuin alkuperäiselläkin Hirschbergin algoritmilla. Alussa taulukoiden luomisen aikavaativuus on sama kuin alkuperäisessä versiossakin. Korkeuskäyriä etsitään kahteen suuntaan yhteensä yhtä monta kertaa (p+1, jossa p on korkeuskäyrien lopullinen lukumäärä) kuin alkuperäinen algoritmi tekee yhteen suuntaan. Algoritmin lopussa on oikean täsmäyksen etsimistä varten yksi silmukka, jonka aikavaativuus on maksimissaan O(n). Sen lisäksi alkuperäiseen verrattuna on muutama vakioaikainen testi. Ne eivät kuitenkaan vaikuta kokonaisaikavaativuuteen, joten kun $p \geq \log s$, aikavaativuus on luokkaa O(pn).

2.6. Rickin menetelmä

Claus Rick esitti vuonna 1994 (Ric 1994, s. 12–25) pisimmän yhteisen alijonon ongelmaan kaksi erilaista parannuskeinoa, jotka pystyy myös yhdistämään. Tässä työssä käsittelemme näistä ensimmäistä, O(p(n–p)) algoritmiksi nimettyä menetelmää, josta käytetään myös algoritmi 3 -nimeä. Tässä tutkielmassa ei ole mukana muita Rickin algoritmeja, joten tätä kutsutaan myöhemmin vain Rickin menetelmäksi. Menetelmän perusidea perustuu Huntin ja Szymanskin vuonna 1977 kehittämään algoritmiin (H&S 1977). Huntin ja Szymanskin menetelmässä syötteiden A ja B muodostama kuvitteellinen taulukko käydään läpi riveittäin oikealta vasemmalle solu kerrallaan, minkä jälkeen pisin yhteinen alijono saadaan samalla muodostetusta minimaalisten täsmäysten linkkilistasta. Rickin versiossa puolestaan käydään tuon A:n ja B:n kuvitteellisen taulukon rivejä läpi vasemmalta oikealle sekä sarakkeita ylhäältä alas käyttäen hyödyksi tähän tarkoitukseen luotuja aputietorakenteita. A:n ja B:n muodostama kuvitteellinen taulukko on esitetty tutkielman taulukossa 9.

Algoritmissa aputietorakenteina toimivat suorasaantitaulukot, joista löytyy nopeasti merkkikohtaisesti seuraavan mahdollisen täsmäyksen paikka. Molemmista syötteistä on tehty omat suorasaantitaulukkonsa, jotta eteneminen onnistuu nopeasti sekä rivillä eteenpäin että sarakkeessa alaspäin. Syötteen A suorasaantitaulukon CLOSEST_A koko on (m+2)*s, missä s on

kaikki merkit sisältävän merkistön koko. Vastaavasti syötteen B suorasaantitaulukon CLOSEST_B koko on (n+2)*s. Esimerkiksi CLOSEST_A[s, i] tarkoittaa merkin s seuraavaa esiintymää merkkijonon A kohdasta i lähtien. Taulukoiden muodostamiseen kuluva aika on luokkaa O(ns). Suorasaantitaulukoiden lisäksi tilaa tarvitaan vain muutamalle muuttujalle, laskennan etenemistä seuraaville col-Thresh- ja row-Thresh-*kynnysarvovektoreille* sekä alijonojen linkkilistoille. Kynnysarvo osoittaa, minkä sarakkeen kohdalla kyseinen korkeuskäyrä kulkee kyseistä riviä käsiteltäessä. Kun m≤n, vektoreiden row-Thresh ja col-Thresh pituudet ovat m+2.

Taulukko 9. Merkkijonoista A (=”abcdbb”) ja B (=”cbacbaaba”) muodostettu taulukko, jossa x merkitsee minimaalista täsmäystä. Katkoviiva esittää taulukon lävistäjää, jota pitkin algoritmi ”laskeutuu” alas. Taulukkoa ei luoda Rickin algoritmissa, vaan se on tässä helpottamassa asian esittämistä.

	c	b	a	c	b	a	a	b	a
a			x						
b		x			x				
c	x			x					
d									
b		x			x				
b								x	

Rickin menetelmä siirtyy taulukossa 9 lävistäjää pitkin askel kerrallaan yhden pykälän oikealle ja alas. Yhden askeleen aikana se etsii suorasaantitaulukoista tarkasteltavan merkin kohdalta ensin käsiteltävän rivin minimaaliset täsmäykset ja sitten toisen suorasaantitaulukon avulla saman numeroiselta sarakkeelta löytyvät minimaaliset täsmäykset. Rick ei esittänyt raportissaan linkkilistojen ylläpitoon tarvittavia koodeja. Raportissa ehdotettiin linkkien tallentamista kahteen erilliseen linkkilistaan. Tämän ehdotuksen pohjalta tähän työhön on tehty kahta erillistä linkkilistaa ylläpitävä menetelmä. Löydetyt minimaaliset täsmäykset tallennetaan linkkilistoihin rowList ja colList. Näiden listoista löytyvien linkkien avulla algoritmin lopussa selvitetään pisin yhteinen alijono.

Pseudokoodiin ei ole lisätty linkkilistojen ylläpitoa. Linkkilistat rowList ja colList sisältävät linkit nollannesta korkeuskäyrästä k. korkeuskäyrään. Linkki on olio, joka on luotu tätä tutkielmaa varten. Yksi linkki sisältää kolme tietoa: täsmäyksen kohdat merkkijonossa A ja B sekä viite edellisen korkeuskäyrän (k–1) linkkiin, jonka alijonoa kyseinen täsmäys pystyy jatkamaan. Linkkilistoihin laitetaan nollannen käyrän kohdalle ns. nollalinkki, jossa täsmäysten kohdat ovat nollia ja edellisen käyrän linkin kohdalla on tyhjää osoittava null. Korkeuskäyrän k linkit linkittyvät aina jonkin (k–1). korkeuskäyrän linkin perään, joten kaikkiin tarvittaviin täsmäyksiin löytyy aina reitti korkeimman korkeuskäyrän linkin kautta. Jonkin korkeuskäyrän k valmistuttua seuraavilta riveiltä ja sarakkeilta löytyvät täsmäykset sijaitsevat aina valmistuneen korkeuskäyrän viimeisimmän täsmäyksen oikealla ja alapuolella taulukossa 9. Pisin mahdollinen yhteinen alijono saattaa löytyä siten, että välillä täsmäys löytyy rivitarkastelussa ja välillä saraketarkastelussa. Jotta ne saadaan linkitettyä toisiinsa, kopioidaan korkeuskäyrän valmistuttua sen viimeksi löytyneen täsmäyksen linkki myös toiseen linkkilistaan.

Rickin algoritmin pseudokoodi (Ric 1994, s. 19) hieman muutettuna:

```
Rick(A, B):
(0)  Esiprosessointi: luo taulukot CLOSEST_A ja CLOSEST_B
(1)  row-Thresh[0] ← 0; col-Thresh[0] ← 0
     for k ← 1 to m + 1
          col-Thresh[k] ← m + 1; row-Thresh[k] ← n + 1
     low ← 1
(2)  for L ← 1 to m
(3)       if col-Thresh[low] = L then
               j ← CLOSEST_B[A_L, row-Thresh[low] + 1]
               low ← low + 1
          else j ← CLOSEST_B[A_L, L]
          k ← low
(4)       while j ≠ n + 1
               if j < row-Thresh[k] then
                    temp ← row-Thresh[k]
                    row-Thresh[k] ← j
                    tallenna minimaalinen täsmäys (L, j) rowListiin
                    j = CLOSEST_B[a_L, temp + 1]
               else if j = row-Thresh[k] then j ← CLOSEST_B[A_L, j + 1]
               k ← k + 1
          päivitä tarvittaessa rowList[low – 1]
(5)       if row-Thresh[low] = L then
               i ← CLOSEST_A[B_L, col-Thresh[low] + 1]
               low ← low + 1
          else i ← CLOSEST_A[B_L, L + 1]
          k = low
(6)       while i ≠ m + 1
               if i < col-Thresh[k] then
                    temp ←col-Thresh[k]
                    col-Thresh[k] ← i
                    tallenna minimaalinen täsmäys (i, L) colListiin
                    i ← CLOSEST_A[B_L, temp + 1]
               else if i = col-Thresh[k] then i ← CLOSEST_A[B_L, i + 1]
               k ← k + 1
          päivitä tarvittaessa colList[low – 1]
(7)  palauta LCS
```

Rickin alkuperäiseen koodiin on tehty yksi lisäys. Jos koodista löytyy vaiheissa (4) ja (6) sallittu minimaalinen täsmäys, ensimmäinen ehtolause toteutuu. Tällöin jälkimmäinen ei voi mitenkään toteutua, joten sitä on turha testata. Tästä syystä vaiheisiin (4) ja (6) jälkimmäisen if-lauseen eteen laitetaan else, jolloin jälkimmäinen ehto tarkastetaan vain, kun ensimmäinen ei ole toteutunut. Tämän jälkimmäisen ehdon toteutuminen tarkoittaa, että toisessa syötteessä on peräkkäin kaksi samaa merkkiä, joista ensimmäisestä löytyi jo minimaalinen täsmäys. Tämä asia huomattiin vasta testien tekemisen jälkeen, joten luvun 3 testeissä on

käytetty alkuperäistä versiota, jossa jälkimmäinen ehto on testattu uuden minimaalisen täsmäyksen löytyessäkin. Yhden vakioaikaisen ehtolauseen käsittelyllä ei pitäisi kuitenkaan olla merkittävää vaikutusta suoritusaikaan, vaikka se suoritetaankin algoritmin aikana useita kertoja.

Taulukon 9 rivejä ja sarakkeita läpi käytäessä löytyvät täsmäykset voidaan jakaa kahteen luokkaan kahdella eri tavalla. Täsmäyksien kopioiminen linkkilistaan voi tapahtua siis neljällä eri tavalla. Ensimmäisessä tapauksessa (k–1). korkeuskäyrälle ei ole löytynyt täsmäystä ja etsittävälle korkeuskäyrälle k on löytynyt jo aikaisemmin ylemmiltä riveiltä täsmäys. Tällöin riviltä voi löytyä täsmäys myös korkeuskäyrälle (k+1), minkä vuoksi luokan k täsmäyksen löytyessä otetaan linkkilistasta vanha k:n linkki väliaikaisesti talteen, jotta mahdollinen (k+1). korkeuskäyrän täsmäys liitetään oikein. Koska samalta riviltä ei ole löytynyt (k–1). korkeuskäyrän täsmäystä, linkkilistaan luotava linkki voidaan yhdistää suoraan (k–1). korkeuskäyrän linkkiin.

Toisessa tapauksessa riviltä on jo löytynyt (k–1). korkeuskäyrälle täsmäys ja aikaisemmilta riveiltä on jo löytynyt k:n täsmäys. Vanha k:n linkki täytyy tässä tapauksessa ottaa muistiin mahdollisen (k+1). korkeuskäyrän löytymistä varten, ja löydetyn k-täsmäyksen linkki täytyy yhdistää muistissa olevaan, joltain ylemmältä riviltä löytyneeseen (k–1). korkeuskäyrän linkkiin. Kolmannessa tapauksessa rivitarkastelussa löytynyt täsmäys on kyseisen korkeuskäyrän ensimmäinen täsmäys, mutta (k–1)-täsmäyksiä ei ole löytynyt samalta riviltä. Tällöin linkkilistaan voidaan vain lisätä linkki, joka osoittaa sarakkeen (k–1)-linkkiin. Neljännessä tapauksessa täsmäys on kyseisen korkeuskäyrän ensimmäinen, mutta samalta riviltä on aikaisemmin tällä kierroksella löytynyt (k–1). korkeuskäyrän täsmäys. Tällöin tämä korkeuskäyrä liitetään muistissa olevaan edellisen rivin (k–1). korkeuskäyrän linkkiin. Näissä kahdessa viimeisessä tapauksessa korkeuskäyrää (k+1) ei voi löytyä tältä riviltä. Vastaavasti saraketarkastelussa täsmäykset jakautuvat neljään eri luokkaan sen mukaan, onko saraketarkastelussa löydetty vielä käsiteltävän korkeuskäyrän k täsmäyksiä ja onko kyseiseltä sarakkeelta löydetty täsmäystä korkeuskäyrälle (k–1).

Esimerkki algoritmin toiminnasta (A = ”abcdbb”, B = ”cbacbaaba”):

Annetaan Rickin algoritmille samat syötteet, ”abcdbb” ja ”cbacbaaba” aivan kuin muillekin aikaisemmin esitetyille algoritmeille. Ensimmäiseksi luodaan suorasaantitaulukot CLOSEST_A ja CLOSEST_B, jotka on esitetty taulukoissa 10 ja 11. Merkkijonon A ensimmäinen merkki on a, joten lisätään suorasaantitaulukkoon CLOSEST_A kirjaimen a kohdalle sarakkeeseen 1 arvo 1. A:ssa ei ole enempää a-kirjaimia, joten muihin merkin a sarakkeisiin tulee arvo m+1 eli 7. A:n toinen merkki on b, joten lisätään sen kohdalle kahteen ensimmäiseen sarakkeeseen arvo 2, joka tarkoittaa ko. kohdan seuraavaa merkin esiintymää. Seuraava b löytyy paikasta 5, joten kolmannesta viidenteen sarakkeeseen lisätään soluihin arvo 5. Viimeinen b löytyy paikasta 6, joten kuudenteen sarakkeeseen laitetaan merkin b kohdalle arvo 6. Rivin loppuihin soluihin laitetaan arvoksi 7. Samat operaatiot tehdään merkeille c ja d, jolloin on saatu CLOSEST_A valmiiksi. Toinen suorasaantitaulukko muodostetaan samalla tavalla, mutta merkkien paikat katsotaan tällä kertaa merkkijonosta B. Viimeisiin soluihin tulee arvoksi n+1 eli 10. Koska merkkiä d ei löydy ollenkaan, kaikkiin sen rivin soluihin tulee arvo 10.

Taulukko 10. Merkkijonosta A = ”abcdbb” muodostettu suorasaantitaulukko CLOSEST_A.

	1	2	3	4	5	6	7	8
a	1	7	7	7	7	7	7	7
b	2	2	5	5	5	6	7	7
c	3	3	3	7	7	7	7	7
d	4	4	4	4	7	7	7	7

Taulukko 11. Merkkijonosta B = ”cbacbaaba” muodostettu suorasaantitaulukko CLOSEST_B.

	1	2	3	4	5	6	7	8	9	10	11
a	3	3	3	6	6	6	7	9	9	10	10
b	2	2	5	5	5	8	8	8	10	10	10
c	1	4	4	4	10	10	10	10	10	10	10
d	10	10	10	10	10	10	10	10	10	10	10

Taulukko 12. Vektorit row-Thresh ja col-Thresh Rickin algoritmin suorittamisen jälkeen.

	0	1	2	3	4	5	6	7
row-Thresh	0	2	4	5	8	10	10	10
col-Thresh	0	3	5	7	7	7	7	7

Vaiheessa (1) alustetaan vektorit col-Thresh ja row-Thresh siten, että molemmissa ensimmäiseen sarakkeeseen tulee arvo 0. Tämän jälkeen col-Threshin muihin sarakkeisiin laitetaan arvo m+1 ja row-Threshin muihin sarakkeisiin n+1. Taulukossa 12 ovat kynnysarvovektoreiden arvot algoritmin suorittamisen jälkeen. Muuttuja low saa arvon 1, joka tarkoittaa, että alimman etsittävän korkeuskäyrän numero on 1. Vaiheessa (2) otetaan käyttöön muuttuja L, joka tarkoittaa kulloisenkin kierroksen aloitusriviä ja -saraketta. Ensimmäiseksi L:n arvoksi laitetaan yksi, joten taulukosta 9 tarkastellaan ylimmältä riviltä ja vasemmanpuoleisesta sarakkeesta löytyviä minimaalisia täsmäyksiä.

Ensimmäiseksi etsitään A:n ensimmäiselle merkille a täsmäyksiä CLOSEST_B-taulukon avulla. Vaiheen (3) ehto ei toteudu, joten j:n arvoksi otetaan CLOSEST_B:stä merkin a kohdalta ensimmäinen luku, joka on 3. Etsittävän käyrän järjestysnumeroa seuraava muuttuja k saa arvon 1. Vaiheen (4) ehdot täyttyvät, sillä j<10. Taulukon row-Thresh[k] edellinen arvo otetaan talteen muuttujaan temp, ja sen tilalle laitetaan j:n arvo 3. Koska kyseessä on korkeuskäyrän ensimmäinen täsmäys, rivien linkkilistaan (rowList) ensimmäisen käyrän kohdalle tallennetaan uusi linkki tähän täsmäykseen. Linkki yhdistetään nollannen käyrän null-linkkiin, jotta sen tiedetään olevan ensimmäisen korkeuskäyrän linkki. Muuttujan j arvoksi muutetaan CLOSEST_B:n 11. sarakkeesta löytyvä arvo 10. Muuttuja k kasvaa yhdellä, jotta voitaisiin jatkaa toisen korkeuskäyrän täsmäysten etsimistä samalta riviltä. Vaiheen (4) while-silmukassa suorittaminen kuitenkin pysähtyy, koska j:n arvo on n+1. Nyt taulukon 9 ensimmäinen rivi on saatu tarkasteltua.

Rivitarkastelun jälkeen siirrytään saraketarkasteluun, jossa etsitään minimaalisia täsmäyksiä taulukon 9 vasemmanpuoleisesta sarakkeesta. Ensimmäisen rivin täsmäys ei löytynyt ensimmäisestä sarakkeesta, joten vaiheen (5) ehto ei toteudu. Siksi ensimmäiselle korkeuskäyrälle voi löytyä lisää minimaalisia täsmäyksiä. Muuttuja i saa arvon CLOSEST_A:n

ensimmäisestä sarakkeesta merkin c kohdalta, eli arvoksi tulee 3. Etsittävä korkeuskäyrä on ensimmäinen, joten k:n arvoksi päivitetään 1. Vaiheessa (6) while- ja if-lauseen ehdot toteutuvat. Kopioidaan vanha arvo col-Thresh-taulukosta muuttujaan temp ja korvataan taulukon arvo i:llä. Löytynyt täsmäys on tämän korkeuskäyrän ensimmäinen saraketarkastelussa, joten lisätään uusi linkki colList-taulukkoon ensimmäisen käyrän kohdalle. Yhdistetään luotu linkki null-linkkiin, jolloin tämän tiedetään olevan ensimmäisen korkeuskäyrän linkki. Taulukosta CLOSEST_A saadaan muuttujalle i uusi arvo 7. Muuttujan k arvoa kasvatetaan toisen korkeuskäyrän etsimistä varten, mutta vaiheen (6) while-silmukan suorittaminen päättyy i:n ollessa 7.

Taulukosta 9 on käyty läpi ensimmäinen rivi ja sarake, joten vaihe (2) kasvattaa L:n arvoa ja ”liukuu” lävistäjää pitkin seuraavaan alkupisteeseen. Tällä kierroksella tarkastellaan toista riviä ja saraketta. Saraketarkastelussa ei löytynyt tämän käyrän täsmäyksiä tältä riviltä, joten vaiheen (3) ehto ei täyty. Muuttuja j saa arvon 2 ja k:n arvoksi tulee taas 1. Vaiheen (4) while-lauseen ehto täyttyy, kuten myös if-lauseen ehto. Otetaan row-Thresh[1]:n vanha arvo 3 muuttujaan temp ja laitetaan tilalle j:n arvo 2. Tämä on käsiteltävän rivin ensimmäinen täsmäys, mutta rivittäinen tarkastelu on löytänyt jo tälle korkeuskäyrälle ainakin yhden aikaisemman täsmäyksen. Se tarkoittaa, että tältä käyrältä voi löytyä myös toisen korkeuskäyrän täsmäys, jota ei voi liittää nyt löytyneen täsmäyksen perään. Tämän vuoksi otetaan rowList-taulukosta ensimmäisen käyrän linkki väliaikaisesti muistiin, minkä jälkeen se korvataan taulukossa uudella linkillä. Ensimmäisen käyrän ollessa kyseessä linkki yhdistetään taas null-linkkiin. Muuttujalle j etsitään uusi arvo 5 CLOSEST_B:stä. Tämän jälkeen kasvatetaan k:n arvoa ja siirrytään etsimään toisen käyrän täsmäyksiä.

Vaiheen (4) while-silmukassa päästään toiselle kierrokselle. Toisen korkeuskäyrän pisteitä ei ole vielä löydetty. Muuttujan j arvo on sellainen, että if-lause toteutuu. Otetaan row-Thresh[2]:n vanha arvo muuttujaan temp ja korvataan se j:llä. Nyt löydetty täsmäys on toisen käyrän ensimmäinen, joten enempää täsmäyksiä ei voi enää tämän jälkeen löytyä samalta riviltä. Koska samalta riviltä löytyi jo aikaisemmin (k–1). käyrän täsmäys, täytyy k-täsmäys linkittää ylemmältä riviltä löydettyyn (k–1)-täsmäykseen. Se onnistuu, sillä väliaikaisessa muistissa on linkki edellisen rivin täsmäykseen. Tämä linkitetään siihen ja lisätään linkkilistalle rowList toisen käyrän kohdalle. Muuttuja j saa seuraavaksi arvon 10. Viimeiseksi k:n arvoa kasvatetaan yhdellä, mutta seuraavalle kierrokselle ei enää kuitenkaan siirrytä, sillä while-silmukan ehto ei toteudu.

Seuraavaksi siirrytään saraketarkasteluun, jossa tarkastelun kohteena on taulukon 9 toinen sarake. Vaiheen (5) if-lauseen ehto toteutuu, joten ensimmäinen korkeuskäyrä on nyt saatu valmiiksi. Muuttujan i arvoksi tulee 5 ja low:n arvo kasvatetaan kahteen, joten nyt alin etsittävä on toinen korkeuskäyrä. Myös k:n arvoksi tulee 2. Vaiheen (6) while- ja if-ehto toteutuvat, joten löytyi toisen käyrän täsmäys. Arvo 5 laitetaan col-Thresh[2]:een ja vanha arvo 7 otetaan muuttujaan temp. Tällä kierroksella tästä sarakkeesta ei ole löydetty (k–1)-täsmäystä, ja aikaisemmin saraketarkastelussa ei ole löytynyt k täsmäystä, joten colListiin laitetaan toisen korkeuskäyrän kohdalle uusi linkki, joka yhdistetään suoraan kohdan (k–1)-linkkiin. Muuttuja i saa arvon 7. Muuttujan k arvo kasvaa kolmeen. Koska ensimmäinen korkeuskäyrä tuli valmiiksi ennen saraketarkastelua, kopioidaan sarakelistaan viimeisin löydetty linkki rivilistasta. Tämä voidaan tehdä, sillä kaikki seuraavilta riveiltä ja sarakkeilta löytyvät toisen käyrän täsmäykset

ovat välttämättä aina oikealla ja alapuolella paikasta (2, 2) löytyneeseen täsmäykseen nähden. Vaiheen (6) while-ehto ei enää toteudu, joten siirrytään vaiheeseen (2).

Muuttujan L arvoksi tulee 3, joka tarkoittaa kolmannen rivin ja sarakkeen läpikäymistä. Vaiheen (3) if-lause ei toteudu, joten j saa arvon 4 ja k:n arvoksi tulee taas 2. Vaiheessa (4) while- ja if-lause toteutuvat. Paikasta col-Thresh[2] otetaan vanha arvo 5 muuttujaan temp, ja tilalle laitetaan 4. Korkeuskäyrälle k–1 ei löytynyt tässä rivitarkastelussa täsmäyksiä, mutta k:lle on jo löytynyt aiemmin täsmäys. Otetaan vanha k-linkki muistiin rowLististä ja korvataan se uudella linkillä, joka yhdistetään rowListin paikassa k–1 olevaan linkkiin. Muuttujan j uudeksi arvoksi tulee 10. Muuttujan k arvoa kasvatetaan, mutta while-silmukan ehto ei enää täyty. Vaiheessa (5) if-ehto ei täyty, joten i:ksi tulee 7 ja k:n arvoksi tulee 2. Vaiheen (6) while-silmukkaan ei mennä kuitenkaan sisälle, sillä ehto ei täyty.

Seuraava kierros alkaa vaiheesta (2) kasvattamalla L:n arvo neljään. Vaiheen (3) ehto ei toteudu, joten j:n arvoksi tulee 10 ja k:n arvo säilyy samana. Vaiheeseen (4) ei mennä sisälle, sillä while-silmukan ehto ei toteudu. Vaihessa (5) huomataan, että toinen korkeuskäyrä on tullut valmiiksi, joten i saa arvon 7 ja low:ta kasvatetaan. Kopioidaan low:n uusi arvo myös muuttujaan k. Vaiheen (6) while-ehto ei toteudu, koska täsmäyksiä ei voi löytyä. Koska toisen korkeuskäyrän huomattiin valmistuneen, kopioidaan nyt rowListin toisen korkeuskäyrän linkki colListiin. Vaiheessa (2) L:n arvoksi tulee nyt 5. Vaiheen (3) if-lauseen ehto ei toteudu, joten j:n arvoksi tulee 5 ja k:n arvona on edelleen 3. Vaiheen (4) while- ja if-lause toteutuvat. Otetaan row-Thresh[3]:n vanha arvo muistiin muuttujaan temp ja laitetaan tilalle j:n arvo. Täsmäys on kolmannen käyrän ja tämän rivin ensimmäinen, joten voidaan luoda uusi linkki ja yhdistää se suoraan (k–1)-linkkiin. Muuttuja j saa arvon 10 ja k:sta tulee 4. Enempää täsmäyksiä rivillä ei ole, joten vaiheen (4) silmukan ehto ei enää toteudu. Vaihessa (5) huomataan kolmannen käyrän tulleen valmiiksi rivitarkastelussa. Muuttuja i saa arvon 7 ja low sekä k saavat arvon 4. Vaiheen (6) while-silmukkaan ei mennä, mutta kopioidaan taas kolmas linkki rowLististä colListiin.

Viimeinen kierros alkaa, kun vaiheessa (2) L:n arvoksi tulee 6. Vaiheen (3) if-lause ei toteudu, sillä saraketarkastelussa ei saatu k:nnetta korkeuskäyrää valmiiksi. Muuttujan j arvoksi tulee 8 ja k:n arvona pysyy 4. Vaiheen (4) while- ja if-lauseen ehdot täyttyvät, joten otetaan vanha arvo muistiin paikasta row-Thresh[4] ja päivitetään se. Uusi rowListin linkki voidaan yhdistää suoraan (k–1). käyrän linkkiin. Muuttuja j saa arvon 10 ja k kasvaa. Silmukan suorittaminen ei jatku, sillä ehto j≠n+1 ei toteudu. Vaiheessa (5) huomataan neljännen korkeuskäyrän tulleen valmiiksi rivitarkastelussa, joten low kasvaa. Täsmäyksiä ei voi löytyä, joten vaiheen (6) while-silmukkaan ei mennä sisälle. Kopioidaan viimeisin täsmäys rowLististä myös colListiin. Tämän jälkeen vaiheen (2) for-silmukkaan ei mennä enää sisälle, joten linkkitaulukot ovat valmiita.

Tulos saadaan ottamalla korkeimman korkeuskäyrän linkki rowLististä. Koska m≤n, rowLististä löytyy aina korkeimman käyrän piste. Jos korkeimman käyrän ainoa täsmäys on löydetty saraketarkastelussa, edellä mainitun ehdon takia käyrä ehtii valmistua joka tapauksessa, jolloin täsmäys kopioidaan rowListiin. Linkin tiedon avulla saadaan pisimmän yhteisen alijonon viimeinen merkki ja tieto edellisestä linkistä. Edellisen linkin avulla löydetään toiseksi viimeinen merkki ja taas tieto edellisestä linkistä. Näin siirrytään linkkejä pitkin kohti ensimmäisen korkeuskäyrän linkkiä, josta saadaan pisimmän yhteisen alijonon ensimmäinen merkki. Tämän

esimerkin neljänneksi, eli viimeiseksi, merkiksi tulee b, kuten myös kolmanneksi merkiksi. Toisen käyrän linkki osoittaa täsmäyksen paikaksi kohdan (3, 4), josta saadaan merkiksi c. Ensimmäisen korkeuskäyrän linkki osoittaa paikkaan (2, 2), josta saadaan merkki b. Rickin algoritmi on siis löytänyt pisimmäksi yhteiseksi alijonoksi merkkijonon ”bcbb”.

Aikavaativuus:

Rickin algoritmin aikavaativuus on luokkaa O(ns+min{pm, p(n–p)}). Termi ns tulee alun esiprosessoinnista, ja loppuosa tulee minimaalisten täsmäysten etsimisestä. Pisimmän yhteisen alijonon selvittäminen linkkilistojen avulla on tämän jälkeen hyvin nopea operaatio, joka ei vaikuta aikavaativuuteen. Nämä on selvitetty perusteellisesti Rickin tutkimusraportissa (Ric 1994, s. 20–21), joten perusteluihin ei perehdytä tämän enempää tässä työssä.

2.7. Rickin menetelmä kaksisuuntaisena

Tätä tutkielmaa varten luvussa 2.6. esitellystä Rickin menetelmästä on tehty kaksisuuntainen versio. Apuna on käytetty Rickin tutkimusta (Ric 2000), joka kertoo algoritmin mahdollisesta kaksisuuntaisesta suorittamisesta lineaaritilassa. Tehty algoritmi toimii tästä poiketen ilman lineaaritilaisuutta. Muuten se toimii samoilla periaatteilla kuin yksisuuntainenkin, paitsi että se tutkii rivejä ja sarakkeita vuorotellen ylhäältä vasemmalta ja alhaalta oikealta lähtien. Rickin julkaisussa ei ole valmista koodia, joten se on luotu yksisuuntaisen menetelmän pohjalta tätä tutkielmaa varten.

Minimaalisten täsmäysten etsiminen:

Taulukko 13 esittää tässä työssä käytetyistä merkkijonoista A ja B muodostettua taulukkoa sekä täsmäysten etsimistä kaksisuuntaisen algoritmin avulla. Minimaalisia täsmäyksiä etsitään ensin ylhäältä riveittäin ja vasemmalta sarakkeittain suorasaantitaulukoiden avulla. Tämän jälkeen minimaalisia täsmäyksiä etsitään alhaalta riveittäin ja oikealta sarakkeittain käänteisten suorasaantitaulukoiden avulla. Kaksisuuntaisessa lähestymisessä otetaan huomioon jo toiseen suuntaan tarkastellut taulukon solut sekä niistä löytyneet täsmäykset. Jos eri suunnista löytyvät sellaiset keskeneräiset korkeuskäyrät, jotka eivät voi jatkaa toisiaan, ne yhdistetään siirtämällä täsmäykset listasta toiseen. Tällainen siirto voi tapahtua missä tahansa neljästä tarkastelusuunnasta. Myös jos korkeuskäyrä valmistuu, eikä samasta kulmasta lähtiessä toiseen suuntaan ole löytynyt yhtään sallittua minimaalista täsmäystä, tilanne katsotaan ensimmäisen täsmäyksen löytymisen kanssa samanlaiseksi. Tällöin toisen suunnan keskeneräinen korkeuskäyrä on mahdollista siirtää, jos se sijaitsee siten, että käyrät eivät voisi jatkaa toisiaan. Korkeuskäyrältä toiselle siirtämisen jälkeen edellinen linkki osoittaa edelleen samaan kohtaan, joten se voisi aiheuttaa virheitä pisintä yhteistä alijonoa etsittäessä. Tämän vuoksi tieto edellisestä linkistä nollataan. Siirtyneet täsmäykset sijaitsevat todennäköisimmin taulukon 13 oikeassa ylänurkassa ja vasemmassa alanurkassa. On todennäköistä, että ainakin isoilla syötteillä niiden jälkeen löytyy keskemmältä taulukkoa saman käyrän pisteitä. Osa täsmäyksistä saattaa siirtyä vielä uudelleenkin, joten on parempi etsiä oikea edeltävä täsmäys vasta tarvittaessa, kun lopussa selvitetään pisintä yhteistä alijonoa.

Täsmäysten siirtämien korkeuskäyrältä toiselle tapahtuu aina koko käyrä kerrallaan. Tällöin korkeuskäyrien lukumäärä pienenee sieltä suunnasta, josta korkeuskäyrä siirrettiin pois.

Kaikille siirrettävän korkeuskäyrän täsmäyksille ei välttämättä löydy sallittua edeltävää täsmäystä uuden suunnan edeltävältä korkeuskäyrältä. Tätä ei huomioida siirtoa tehdessä, vaan kaikki täsmäykset siirretään samalle korkeuskäyrälle. Asia huomioidaan vasta lopussa, jossa tarkastetaan tarvittaessa, löytyvätkö kyseiselle täsmäykselle sallittu edeltäjä ja seuraaja. Valmiita korkeuskäyriä tai niihin kuuluvia yksittäisiä täsmäyksiä ei siirretä, sillä jokaiselta valmiilta korkeuskäyrältä kuuluu aina välttämättä yksi täsmäys pisimpään yhteiseen alijonoon.

Taulukko 13. Merkkijonoista A (=”abdcbb”) ja B (=”cbacbaaba”) muodostettu taulukko, jossa musta x merkitsee minimaalista täsmäystä. Lihavoitu punainen x tarkoittaa löydettyä minimaalista täsmäystä, joka siirretään myöhemmin toiselle käyrälle. Harmaat nuolet esittävät täsmäysten etsimisen suuntia. Katkoviivat esittävät taulukon ”lävistäjiä”, joita pitkin algoritmi kulkee suorituksessa eteenpäin. Rivitarkastelun lähtösolut löytyvät siis katkoviivalta. Taulukkoa ei luoda Rickin algoritmissa, vaan se on tässä helpottamassa asian esittämistä.

	c	b	a	c	b	a	a	b	a
a			x						
b		x			x				
c	x			x					
d									
b		**x**			x				
b								x	

Pisimmän yhteisen alijonon selvittäminen suoritetaan kaikkien täsmäysten löytymisen jälkeen. Se ei ole lähellekään yhtä yksinkertainen operaatio kuin yhteen suuntaan etenevällä tarkastelulla. Periaate linkkien seuraamisessa on samanlainen kahteen eri suuntaan suoritettuna, mutta oikeat lähtölinkit pitää ensin etsiä. Niiden löytämiseen vaikuttavat eri suuntiin löytyneiden käyrien lukumäärä sekä myös tarkastelusuunnasta toiseen siirtyneet korkeuskäyrät. Saattaa tulla myös tilanne, jossa kaikista lähtölinkeistä ei löydy oikeata pisintä yhteistä alijonoa. Kaikki löytyneiden minimaalisten täsmäysten linkit säilytetään muistissa linkkilistoissa korkeuskäyrittäin toisin kuin yksisuuntaisessa versiossa, jossa jokaiselta korkeuskäyrältä on muistissa molempiin suuntiin vain viimeksi löytyneen täsmäyksen linkki.

PYAn selvittäminen, tapaus 1:

Pisimmän yhteisen alijonon selvittämisessä on kolme eri tapausta. Ensimmäinen tapaus on se, jossa riveittäin ylhäältä on löytynyt useampia korkeuskäyriä kuin vasemmalta saraketarkastelussa ja myös alhaalta on riveittäin löytynyt useampia korkeuskäyriä kuin oikealta sarakkeittain tarkasteltaessa. Tällöin pitää vertailla ylhäältä ja alhaalta löytyneiden keskeneräisten korkeuskäyrien kynnysarvoja row-Thresh- ja row-Thresh2-vektoreiden avulla. Tämän tarkastelun suunnittelun pohjana on käytetty apuna Goemanin ja Clausenin kaksisuuntaisen menetelmän algoritmeja (G&C 1999). Koitetaan etsiä sellaiset korkeuskäyrät, jotka pystyvät jatkamaan toisiaan ja joista löytyisi mahdollisimman pitkä yhteinen alijono. Jos löydetään korkeuskäyrät, jotka pystyvät jatkamaan toisiaan, otetaan ne muistiin ja etsitään vielä parempia (pidemmän yhteisen alijonon tuottavia) käyräpareja, kunnes korkeuskäyrät loppuvat vähintään toisesta suunnasta. Yhdenkin toisiaan jatkavan parin löytyessä voidaan helposti selvittää ja palauttaa pisin yhteinen alijono, sillä korkeuskäyrän siirron läpikäyneitä linkkejä ei voi tulla vastaan. Jos taas käyrät loppuvat kesken eikä ole löytynyt toisiaan jatkavaa korkeuskäyräparia, päivitetään sen suunnan käyrien lukumäärä, josta käyrät loppuivat ensin

kesken. Tällöin tapaus muuttuu joksikin muuksi tapaukseksi, joka selvitetään myöhemmässä vaiheessa.

Syötteillä ”aceb” ja ”abcd” saadaan esimerkiksi aikaan tilanne, jossa riveittäin löytyy molempiin suuntiin enemmän täsmäyksiä kuin sarakkeittain tarkasteltaessa. Ylhäältä on löytynyt valmistuneelle ensimmäiselle korkeuskäyrälle täsmäys merkkiparista (A_1, B_1), joka on kopioitu myös vasemmalta löytyneisiin täsmäyksiin korkeuskäyrän valmistuessa. Ylhäältä on löytynyt täsmäys myös toiselle korkeuskäyrälle merkkiparista (A_2, B_3). Alhaalta on löytynyt vain yksi täsmäys ensimmäiselle korkeuskäyrälle merkkiparista (A_4, B_2). Otetaan ylhäältä ylin korkeuskäyrä ja alhaalta alin korkeuskäyrä, joka ei ole valmistunut. Vertaillessa näiden kynnysarvoja 3 ja 2 huomataan, että nämä korkeuskäyrät eivät voi jatkaa toisiaan. Pienennetään korkeuskäyrän numeroa ylhäältä ja kasvatetaan alhaalta. Ylhäältä tulee kuitenkin valmis korkeuskäyrä vastaan, joten ei tehdä enää uutta vertailua. Keskeneräiset korkeuskäyrät loppuivat ylhäältä ensin, joten pienennetään korkeuskäyrien lukumäärää yhtä suureen arvoon kuin vasemmalta on löytynyt korkeuskäyriä. Arvosta tulee näin yksi. Tämän jälkeen tapauksesta tulee sellainen tapaus, jossa vasemmalta ja alhaalta (1+1) on löytynyt korkeuskäyriä yhteensä enemmän kuin ylhäältä ja oikealta (1+0). Jos korkeuskäyrät olisivat jatkaneet toisiaan, olisi PYA saatu helposti lähtemällä viimeisimmistä toisiaan jatkaneista täsmäyksistä linkkejä pitkin eteen- ja taaksepäin.

PYAn selvittäminen, tapaus2:

Toinen tapaus on sellainen, jossa vasemmalta ja alhaalta korkeuskäyrien lukumäärien yhteenlaskettu summa on suurempi kuin ylhäältä ja oikealta. Tähän tapaukseen päädytään myös sen jälkeen, jos ensimmäisessä tapauksessa korkeuskäyrät loppuivat ylhäältä ennen kuin löytyi toisiaan jatkava korkeuskäyräpari. Jos vasemmalta tai alhaalta ei ole löytynyt yhtään täsmäystä, selviää pisin yhteinen alijono suoraan seuraamalla linkkejä toisen suunnan ylimmästä käyrästä taaksepäin lähtien. Tämä tosin tapahtuu yleensä vain hyvin pienillä syötteillä. Normaalisti molemmista suunnista kuitenkin löytyy korkeuskäyriä, jolloin pisin yhteinen alijono selvitetään etsimällä sallitut edeltäjät ja seuraajat vasemmalta löytyneen ylimmän korkeuskäyrän listan ensimmäiselle täsmäykselle. Edeltäjät ja seuraajat yritetään selvittää edellisten täsmäysten tietojen avulla. Tämä tieto voi puuttua vain lähimpänä keskustaa olevista linkeistä. Tällaisien linkkien edeltäjät selvitetään rekursiivisesti. Tämä rekursiivinen metodi palauttaa pisimpään yhteiseen alijonoon kuuluvien täsmäysten linkit listana siihen asti, kunnes löytyy taas tieto edellisistä linkeistä. Kun löytyy ensimmäisen kerran tieto edellisestä täsmäyksestä, tämän jälkeen on loppujenkin edellisten täsmäysten kohdat oltava tiedossa. Ne voidaan selvittää siis nopeasti ilman rekursiopinon kasvattamista samaan tapaan kuin yksisuuntaisessakin versiossa. Jos täsmäyksestä lähdettäessä ei löydy sallittuja edeltäjiä tai seuraajia, rekursiivinen menetelmä palauttaa tyhjän listan. Tällöin seuraajia ja edeltäjiä etsitään järjestyksessä seuraaville ylimmän korkeuskäyrän täsmäyksille niin kauan, kunnes löytyy sallittu pisin yhteinen alijono. Kun jollekin pisteelle löytyy sallitut edeltäjät ja seuraajat, selvitetään rekursiivisesti muodostettujen listojen sekä linkkien edellisten täsmäysten tietojen avulla pisimmän yhteisen alijonon merkit. Yhden oikean vastauksen löytäminen riittää, joten sellaisen löydyttyä se palautetaan heti.

Aikaisemmin syötteistä ”aceb” ja ”abcd” muodostui siis tällainen toisen luokan tapaus, kun ensimmäisen luokan tapauksena ei löytynyt toisiaan jatkavia käyriä ja ylhäältä löytyneiden korkeuskäyrien määrää pienennettiin. Pisintä yhteistä alijonoa lähdetään etsimään järjestysnumeroltaan korkeimman vasemmalta löytyneen korkeuskäyrän täsmäyksistä. Niistä valitaan viimeksi löytynyt (ja tässä tapauksessa ainoa) täsmäys, joka löytyy merkkiparista (A_1, B_1). Tämän jälkeen korkeimman alhaalta löytyneen korkeuskäyrän pisteitä testataan, kunnes löytyy toisiaan jatkava täsmäyspari. Sellainen löytyy heti ensimmäisestä (ja tässä tapauksessa ainoasta) täsmäyksestä, joka löytyy merkkiparista (A_4, B_2). Nyt kun kaksi toisiaan jatkavaa pistettä on löytynyt, selvitetään, löytyykö näiden avulla sallittua PYAa. Vasemmalta löytyneelle täsmäykselle etsitään edeltäjiä rekursiivisesti linkkien avulla, kunnes tulee vastaan sellainen linkki, jossa on tieto edellisestä täsmäyksestä. Koska annettu ensimmäinen täsmäys sisältää jo tiedon edeltäjästä (ns. null-linkki), palautetaan se heti yhden mittaisena listana. Sama tehdään myös alhaalta löytyneestä ylimmän käyrän täsmäyksestä lähtien. Myös tässä tapauksessa tulos on sama, joten nyt tiedetään kaikki, mitä tarvitaan PYAn selvittämiseen. Alkuosa selvitetään lähtemällä merkkiparin (A_1, B_1) täsmäyksestä linkkejä pitkin taaksepäin, jolloin saadaan tulokseksi merkkijono ”a”. Loppuosa selvitetään vastaavasti merkkiparin (A_4, B_2) täsmäyksestä lähtien linkkejä pitkin eteenpäin etsien, jolloin saadaan loppuosan merkkijonoksi ”b”. Nämä yhdistämällä saadaan pisimmäksi yhteiseksi alijonoksi ”ab”.

PYAn selvittäminen, tapaus 3:

Kolmas tapaus sisältää kaikki muut kuin kahden edellisen tapauksen vaihtoehdot. Tässä tapauksessa ylhäältä ja oikealta korkeuskäyrien lukumäärien yhteenlaskettu summa on suurempi tai yhtä suuri kuin vasemmalta ja alhaalta. Tähän tapaukseen päädytään myös, jos ensimmäisessä tapauksessa korkeuskäyrät loppuivat alhaalta, ennen kuin löytyi toisiaan jatkava korkeuskäyräpari. Jos tämän tapauksen kaikki täsmäykset ovat löytyneet joko ylhäältä tai oikealta, saadaan lopputulos seuraamalla linkkejä sen suunnan ylimmästä korkeuskäyrästä lähtien. Jos mistään suunnasta ei ole löytynyt yhtään korkeuskäyrää, palautetaan tyhjä alijono. Tavallisesti täsmäyksiä on löytynyt molemmista suunnista, jolloin pisin yhteinen alijono selvitetään samalla tavalla kuin toisessa tapauksessa, tosin tällä kertaa ylhäältä ja oikealta löydettyjen täsmäysten avulla. Alkupisteeksi otetaan ensimmäiseksi ylhäältä löytyneen ylimmän korkeuskäyrän ensimmäinen täsmäys, jolle etsitään edeltäjät ja seuraajat. Jos sallittuja edeltäjiä tai seuraajia ei löydetä, otetaan seuraava ylimmän korkeuskäyrän täsmäys. Kun sallitut edeltäjät ja seuraajat löytyvät, selvitetään rekursiivisesti saatujen listojen sekä edellisten täsmäysten tietojen avulla pisin yhteinen alijono ja palautetaan se.

Kolmas tapaus saadaan esimerkiksi syötteillä ”acce” ja ”abcd”. Merkkiparista (A_1, B_1) löytyy täsmäys ensimmäiselle korkeuskäyrälle, joka myös valmistuu heti. Näin täsmäyksen linkki lisätään etuperin etsittävien rivi- ja sarakelistaan. Merkkiparista (A_2, B_3) löytyy eteenpäin etsittäessä toisen korkeuskäyrän täsmäys rivitarkastelussa. Merkkiparista (A_3, B_3) löytyy ensimmäisen korkeuskäyrän täsmäys takaperin etsittäessä. Korkeuskäyrä myös valmistuu heti, joten täsmäyksen linkki kopioidaan myös saraketarkastelun listaan. Koska saraketarkastelussa ei ollut löytynyt yhtään täsmäystä tälle korkeuskäyrälle, tarkastetaan ylhäältä löytyneen korkeimman luokan korkeuskäyrän viimeinen täsmäys. Tämä sijaitsee siten, että käyrät eivät voi jatkaa toisiaan, joten kaikki korkeuskäyrän linkit siirretään ylhäältä rivitarkastelussa löytyneiden listasta oikealta saraketarkastelussa löytyneiden listalle ensimmäisen

korkeuskäyrän kohdalle. Samalla myös tuon siirtyvän linkin tieto edellisen täsmäyksen linkistä nollataan. Tämän jälkeen päädytään siis kolmanteen lopputuloksen tapaukseen, jossa ylhäältä ja oikealta on löytynyt yhteensä vähintään yhtä paljon korkeuskäyriä kuin vasemmalta ja alhaalta.

Kolmannessa tapauksessa pisintä yhteistä alijonoa etsitään samalla tavalla kuin toisessakin tapauksessa, vain listat vaihtuvat. Valitaan ylhäältä ylimmän korkeuskäyrän viimeksi löytynyt täsmäys ja etsitään sille sallittua jatkajaa oikealta ylimmän korkeuskäyrän listalta. Listan ensimmäinen täsmäys merkkiparista (A_3, B_3) käy jatkajaksi. Ensin etsitään PYAn alkuosa. Etsiminen tapahtuu rekursiivisesti niin kauan, kunnes jostain linkistä löytyy tieto edeltäjästä. Tämä tieto löytyy heti ensimmäisestä tarkasteltavasta linkistä, joka viittaa merkkipariin (A_1, B_1), joten siirrytään tarkastelemaan loppupään löytymistä. Myös loppuosan ensimmäisestä tarkasteltavasta jatkajasta löytyy tieto edeltävän korkeuskäyrän täsmäyksestä, joten kaikki on valmista PYAn selvittämiseksi. Alkuosasta saadaan merkkijono ”a” ja loppuosasta saadaan merkkijono ”c”. Nämä yhdistetään ja lopputuloksena palautetaan merkkijono ”ac”.

Luvun 3 testeissä käytetty koodi poikkeaa pseudokoodista hiukan taaksepäin tapahtuvan etsinnän osalta. Suorasaantitaulukoiden tarkastelusuunta on päinvastainen kuin pseudokoodissa. Tämä vaikuttaa CLOSEST_A2- ja CLOSEST_B2-suorasaantitaulukoiden sisältöön siten, että solujen arvot on laskettu toisesta suunnasta. Samalla kuitenkin myös koodin indeksointia on muutettu vastaavasti, joten palautettavat arvot ovat kuitenkin samoja. Tämän ei pitäisi kuitenkaan vaikuttaa mitenkään luvussa 3 esitettävien testien suoritusaikoihin tai muistinkäyttöön. Nämä takaperin prosessoitaessa käytettävät suorasaantitaulukot on muutettu jälkeenpäin peilikuviksi etuperin suoritettavaan etsintään nähden, jotta koodi olisi helpommin ymmärrettävässä muodossa. Alkuperäisessä versiossa suorasaantitaulukoissa on lopussa kaksi ”ylimääräistä” saraketta, jotta algoritmi ei ajautuisi ulos taulukosta. Kaksisuuntaisessa versiossa paluusuunnan suorasaantitaulukoissa on alussa kaksi vastaavaa saraketta. Pseudokoodi on tehty siten, että taulukoissa 14 ja 15 vältetään negatiivisten sarakearvojen käyttö. Tämän vuoksi kaksi ensimmäistä saraketta sisältävät vain nollia. Huomionarvoista on myös se, että lopputuloksen tapauksen selvittämiseen käytetään muuttujia col1, col2, row1 ja row2, mutta niiden ylläpito on jätetty pois pseudokoodista sen lyhentämiseksi. Muuttujien ylläpito tapahtuu aina uusia korkeuskäyriä löydettäessä sekä korkeuskäyrien siirron vuoksi tapahtuvien käyrien poistamisten yhteydessä.

Rickin kaksisuuntaisen menetelmän pseudokoodi:

Rick2w(A, B):
(0) Esiprosessointi: luo taulukot CLOSEST_A ja CLOSEST_B, CLOSEST_A2, CLOSEST_B2
(1) row-Thresh[0] ← 0; col-Thresh[0] ← 0; row-Thresh2[0] ← n + 1; col-Thresh2[0] ← n + 1
    for k ← 1 to $\lceil m/2 \rceil + 1$
        col-Thresh[k] ← m + 1; row-Thresh[k] ← n + 1
        col-Thresh2[k] ← 0; row-Thresh2[k] ← 0
    low ← 1; low2 ← 1

(2) for L ← 1 to $\lceil m/2 \rceil$ {

(3) if col-Thresh[low] = L then

j ← CLOSEST_B[A_L, row-Thresh[low] + 1]

low ← low + 1

tarkista ja siirrä tarvittaessa taaksepäin saraketarkastelun ylin korkeuskäyrä

else j ← CLOSEST_B[A_L, L]

k ← low

(4) while j ≠ n + 1

if j < row-Thresh[k] & j < n + 2 – L then

temp ← row-Thresh[k]

row-Thresh[k] ← j

tallenna minimaalinen täsmäys (L, j) taulukkoon rowLinkit

tarkista ja siirrä tarvittaessa taaksepäin saraketarkastelun ylin korkeuskäyrä

j = CLOSEST_B[A_L, temp + 1]

else

if j = row-Thresh[k] then j ← CLOSEST_B[A_L, j + 1]

else if j > n + 1 – L then j ← n + 1

k ← k + 1

kopioi tarvittaessa valmistuneen korkeuskäyrän viim. täsmäys colLinkit -> rowLinkit

(5) if row-Thresh[low] = L then

i ← CLOSEST_A[B_L, col-Thresh[low] + 1]

low ← low + 1

tarkista ja siirrä tarvittaessa taaksepäin rivitarkastelun ylin korkeuskäyrä

else i ← CLOSEST_A[B_L, L + 1]

k = low

(6) while i ≠ m + 1

if i < col-Thresh[k] & i < m + 2 – L then

temp ←col-Thresh[k]

col-Thresh[k] ← i

tallenna minimaalinen täsmäys (i, L) taulukkoon colLinkit

tarkista ja siirrä tarvittaessa taaksepäin rivitarkastelun ylin korkeuskäyrä

i ← CLOSEST_A[B_L, temp + 1]

else

if i = col-Thresh[k] then i ← CLOSEST_A[B_L, i + 1]

else if i > m + 1 – L then i ← m + 1

k ← k + 1

kopioi tarvittaessa valmistuneen korkeuskäyrän viim. täsmäys rowLinkit -> colLinkit

if L*2 – 1 = m then goto step 7

(3*) if col-Thresh2[low2] = m + 1 – L then

j ← CLOSEST_B2[A_{m-L+1}, row-Thresh2[low2]]

low2 ← low2 + 1

tarkista ja siirrä tarvittaessa eteenpäin saraketarkastelun ylin korkeuskäyrä

else j ← CLOSEST_B2[A_{m-L+1}, n – L + 2]

k ← low2

```
(4*)    while j ≠ 0
            if j > row-Thresh2[k] & j > L then
                temp ← row-Thresh2[k]
                row-Thresh2[k] ← j
                tallenna minimaalinen täsmäys (m + 1 – L, j) taulukkoon rowLinkit2
                tarkista ja siirrä tarvittaessa eteenpäin saraketarkastelun ylin korkeuskäyrä
                j = CLOSEST_B2[A_{m-L+1}, temp]
            else
                if j = row-Thresh2[k] then j ← CLOSEST_B2[A_{m-L+1}, row-Thresh2[k]]
                else if j ≤ L then j ← 0
            k ← k + 1
        kopioi tarvittaessa valmistuneen korkeuskäyrän viim. täsmäys colLinkit2 -> rowLinkit2
(5*)    if row-Thresh2[low2] = n + 1 – L then
            i ← CLOSEST_A2[B_{n-L+1},col-Thresh2[low2]]
            low2 ← low2 + 1
            tarkista ja siirrä tarvittaessa eteenpäin rivitarkastelun ylin korkeuskäyrä
        else i ← CLOSEST_A2[B_{n-L+1}, m – L + 1]
        k = low2
(6*)    while i ≠ 0
            if i > col-Thresh2[k] & i > L then
                temp ←col-Thresh2[k]
                col-Thresh2[k] ← i
                tallenna minimaalinen täsmäys (i, n + 1 – L) taulukkoon colLinkit2
                tarkista ja siirrä tarvittaessa eteenpäin rivitarkastelun ylin korkeuskäyrä
                i ← CLOSEST_A2[B_{n-L+1}, temp]
            else
                if i = col-Thresh2[k] then i ← CLOSEST_A2[B_{n-L+1}, col-Thresh2[k]]
                else if i ≤ L then i ← 0
            k ← k + 1
        kopioi tarvittaessa valmistuneen korkeuskäyrän viim. täsmäys rowLinkit2 -> colLinkit2
    }
(7) if row1 > col1 & row2 > col2        // Näitä muuttujia ylläpidetään käyrien löytämisten
                                            ja poistamisten yhteydessä
        then selvitä, löytyykö riveittäin toisiaan jatkavia korkeuskäyriä
        if löytyi toisiaan jatkavat korkeuskäyrät then selvitä ja palauta pisin yhteinen alijono
        else if korkeuskäyrät loppuivat ylhäältä then row1 ← col1 else row2 ← col2
    if col1 + row2 > row1 + col2 then
        pisin yhteinen alijono saadaan vasemmalta + alhaalta, selvitä ja palauta se
    else pisin yhteinen alijono saadaan ylhäältä + oikealta, selvitä ja palauta se
```

Tässä käydään läpi Rickin kaksisuuntaisen version toiminta, mutta kaikkia kohtia ei käydä kovin tarkasti läpi, sillä algoritmin perusidea on selvitetty tarkemmin jo luvussa 2.6. Algoritmin etenemisessä käytetään muuttujia low ja low2, jotka kuvaavat alinta etsittävää käyrää kummastakin suunnasta. Algoritmi tarvitsee myös uusia muuttujia row1, row2, col1 ja col2, jotka kuvaavat sitä, mikä on kuhunkin suuntaan viimeksi löytyneen korkeuskäyrän

järjestysnumero omasta tarkastelusuunnasta. Esimerkiksi muuttujaan col1 lasketaan mukaan myös ne käyrät, jotka ovat löytyneet kokonaan rivitarkastelussa mutta joiden viimeiseksi löydetty täsmäys on kopioitu sarakelistaan käsiteltävän korkeuskäyrän valmistuessa. Vastaavasti row1 on eteenpäin rivitarkastelusta löydettyjen korkeuskäyrien lukumäärä, row2 tarkoittaa rivitarkastelua takaperin alhaalta päin ja col2 on takaperin saraketarkastelussa löydettyjen korkeuskäyrien lukumäärä. Mahdollisessa käyränsiirrossa arvo tietenkin pienenee, kun yksi käyrä poistetaan.

Esimerkki algoritmin toiminnasta (A = ”abcdbb”, B = ”cbacbaaba”):

Annetaan algoritmille syötteet ”abcdbb” ja ”cbacbaaba”. Niistä muodostetaan taulukoiden 10 ja 11 mukaiset suorasaantitaulukot eteenpäin tarkastelua varten aivan kuten yksisuuntaisessakin versiossa. Vastakkaisen suunnan tarkastelua varten tehdään omat suorasaantitaulukot CLOSEST_A2 ja CLOSEST_B2, jotka löytyvät taulukoista 14 ja 15. Niissä on kaksi ylimääräistä ns. katkaisusaraketta alussa aivan kuten toisen suunnan suorasaantitaulukoissa on lopussa. Taulukossa 16 on esitetty row-Thresh, col-Thresh, row-Thresh2 ja col-Thresh2 sellaisina kuin ne ovat algoritmin suorittamisen jälkeen.

Taulukko 14. Merkkijonosta A (=”abcdbb”)muodostettu suorasaantitaulukko CLOSEST_A2 takaperin etsimistä varten.

	0	1	2	3	4	5	6	7
a	0	0	1	1	1	1	1	1
b	0	0	0	2	2	2	5	6
c	0	0	0	0	3	3	3	3
d	0	0	0	0	0	4	4	4

Taulukko 15. Merkkijonosta B (=”cbacbaaba”) muodostettu suorasaantitaulukko CLOSEST_B2 takaperin etsimistä varten.

	0	1	2	3	4	5	6	7	8	9	10
a	0	0	0	0	3	3	3	6	7	7	9
b	0	0	0	2	2	2	5	5	5	8	8
c	0	0	1	1	1	4	4	4	4	4	4
d	0	0	0	0	0	0	0	0	0	0	0

Taulukko 16. Vektorit row-Thresh, col-Thresh, row-Thresh2 ja col-Thresh2 Rickin kaksisuuntaisen algoritmin suorittamisen jälkeen.

	0	1	2	3	4
row-Thresh	0	2	4	10	10
col-Thresh	0	3	5	7	7
row-Thresh2	10	8	5	0	0
col-Thresh2	7	0	0	0	0

Suorasaantitaulukoiden, muuttujien ja kynnysarvotaulukoiden alustamisen jälkeen ensimmäiseksi etsitään minimaalisia täsmäyksiä A:n ensimmäiselle merkille ylimmältä riviltä vaiheessa 4. Löytyy sallittu täsmäys, jossa j:n arvo on 3. Kopioidaan arvo row-Thresh-taulukkoon ja luodaan uusi linkki rivilinkkien ensimmäiselle käyrälle. Korkeuskäyrän

ensimmäisen pisteen löytyessä yritetään etsiä mahdollisia siirrettäviä korkeuskäyriä taaksepäin sarakelistasta, mutta sieltä ei voi löytyä vielä mitään. Ensimmäiseltä riviltä ei voi löytyä enempää täsmäyksiä, joten siirrytään saraketarkasteluun. Vaihe 5 tarkastaa, että ensimmäinen korkeuskäyrä ei vielä valmistunut, joten siirrytään vaiheeseen 6 etsimään täsmäyksiä. Löydetään sallittu täsmäys, jossa i:n arvo on 3. Tämä arvo kopioidaan col-Threshiin ja luodaan uusi linkki sarakelinkkien listaan ensimmäiselle käyrälle. Koska tämä oli korkeuskäyrän ensimmäinen löytynyt täsmäys, tarkastetaan alhaalta rivitarkastelusta ylimmän korkeuskäyrän kynnysarvo. Korkeuskäyriä ei ole löydetty tähän suuntaan, joten mitään ei myöskään voida siirtää. Lisää täsmäyksiä ei enää voi löytyä tähän suuntaan tästä sarakkeesta.

Tämän jälkeen lasketaan, onko vielä rivejä, joita ei ole tarkastettu. Niitä on vielä jäljellä, joten siirrytään takaperin tarkasteluun. Vaiheessa 3* tarkastetaan, onko jokin käyrä valmistunut ja asetetaan sen mukaan arvo muuttujalle j. Vaiheessa 4* etsitään minimaalisia täsmäyksiä takaperin sallitulta alueelta. Lähin täsmäys löydetään toisesta sarakkeesta oikealta eli sarakkeesta 8, joka on sallittu paikka. Laitetaan arvo muistiin row-Thresh2[1]:een ja luodaan uusi linkki ensimmäiselle käyrälle taaksepäin löydettyjen rivilinkkilistaan. Tarkastellaan myös eteenpäin saraketarkastelun kynnysarvolistaa ylimmän käyrän kohdalta, sillä tämä oli ensimmäinen tälle korkeuskäyrälle löydetty täsmäys. Kynnysarvo on kuitenkin pienempi kuin tämän rivin numero ylhäältäpäin, joten täsmäykset (ja siten nämä korkeuskäyrät) voivat jatkaa toisiaan. Tästä syystä mitään ei siirretä. Taaksepäin saraketarkastelu alkaa vaiheesta 5*, jossa todetaan, että mikään käyrä ei valmistunut. Muuttujan i arvoksi tulee 1, joten vaiheessa 6* ensimmäisen käyrän täsmäys löytyy vasta ylimmältä riviltä. Se ei kuitenkaan kelpaa, koska tuo rivi on jo käyty läpi toiseen suuntaan. Muita täsmäyksiä ei voi löytyä, joten ensimmäiset rivit ja sarakkeet on käyty läpi molemmista suunnista.

Algoritmi palaa vaiheeseen 2, josta alkaa uusi kierros. Ensin tarkastetaan, valmistuiko ensimmäinen korkeuskäyrä saraketarkastelussa. Se ei valmistunut, joten jatketaan taloudellisempien, eli lähempänä lähtöpistettä olevien, minimaalisten täsmäysten etsimistä tälle korkeuskäyrälle. Algoritmi löytää sallitun täsmäyksen toisesta sarakkeesta. Edelliseltä riviltä löytyi täsmäys tälle korkeuskäyrälle, joten otetaan kynnysarvolistasta vanha arvo väliaikaisesti muistiin ja päivitetään sen tilalle uuden löydetyn täsmäyksen sarakenumero. Luodaan myös uusi linkki rivitarkastelun linkkilistalle. Koska aikaisemmalta riviltä löytyi jo ensimmäisen korkeuskäyrän täsmäys, etsitään sallittuja täsmäyksiä toiselle korkeuskäyrälle. Sellainen löytyy sarakkeesta 5. Arvo merkitään row-Threshiin toisen käyrän kohdalle. Tämä täsmäys linkitetään rivin 1 ensimmäisen korkeuskäyrän täsmäykseen, jonka linkin tiedot ovat väliaikaisesti muistissa. Luodaan linkkilistan toiselle käyrälle uusi linkki, jossa on tiedot uudesta täsmäyksestä ja edeltävän täsmäyksen linkistä. Koska tämä oli ensimmäinen löydetty täsmäys tälle korkeuskäyrälle, tarkastetaan toisen suunnan saraketarkastelun ylimmän korkeuskäyrän kynnysarvo mahdollisen käyränsiirron vuoksi. Koska sieltä suunnasta ei ole löytynyt korkeuskäyriä, sellaisia ei voi myöskään siirtää. Enempää täsmäyksiä tältä riviltä ei löydy.

Vaiheessa 5 tarkastetaan, tuliko alin etsittävä korkeuskäyrä valmiiksi. Huomataan, että se tuli valmiiksi, joten kasvatetaan alimman etsittävän käyrän järjestysnumeroa ja laitetaan valmistuminen muistiin. Jos sarakesuuntaan valmistuneelle korkeuskäyrälle ei olisi löytynyt yhtään täsmäystä, olisi tarkastettu mahdollisten alhaalta rivitarkastelussa löytyneiden korkeuskäyrien siirtymistä. Nyt kun tähän suuntaan oli löytynyt vähintään yksi täsmäys,

tarkastelua ei tarvitse tehdä. Vaiheessa 6 etsitään toiselle korkeuskäyrälle sallittua täsmäystä, ja sellainen löydetäänkin riviltä 5. Arvo lisätään col-Thresh[2]:een sekä luodaan uusi linkki sarakelinkkilistan toiselle korkeuskäyrälle. Linkki yhdistetään ensimmäisen korkeuskäyrän täsmäyksen linkkiin. Tarkastetaan taas mahdollisten alhaalta riveittäin löytyneiden korkeuskäyrien siirtämistarve todeten, että siirtoa ei tehdä. Enempää täsmäyksiä ei tästä sarakkeesta voi löytyä. Saraketarkastelun lopuksi kopioidaan ensimmäisen korkeuskäyrän viimeisimmän riveittäin löytyneen täsmäyksen linkki myös sarakelistaan.

Kaikkia rivejä ei ole vielä käyty läpi, joten seuraavaksi täsmäyksiä etsitään takaperin alhaalta. Ensimmäinen korkeuskäyrä ei ole valmistunut, joten sille etsitään uutta taloudellisempaa täsmäystä. Sellaista ei löydy, joten siirrytään etsimään toiselle korkeuskäyrälle sallittuja täsmäyksiä. Sellainen löytyy viidenneltä riviltä, joten päivitetään row-Thresh2 ja lisätään uusi linkki takaperin riveittäin löytyneiden linkkilistalle. Koska tämä oli ensimmäinen löydetty täsmäys tälle korkeuskäyrälle, tarkastellaan eteenpäin saraketarkastelussa löytyneiden kynnysarvoja. Korkeimman korkeuskäyrän viimeisin täsmäys, joka on samalla myös taloudellisin, on löytynyt tällä hetkellä käsiteltävänä olevalta riviltä. Nämä käyrät eivät voi mitenkään jatkaa toisiaan pisintä yhteistä alijonoa muodostettaessa, joten siirretään etuperin sarakesuuntaan viimeksi löytyneen korkeuskäyrän linkkilistasta kaikki linkit takaperin rivitarkastelun toiselle korkeuskäyrälle. Näistä kaikki eivät ole uudella käyrällään välttämättä minimaalisia täsmäyksiä, mutta asian tutkimiseen ei käytetä turhaan energiaa. Siirron yhteydessä myös linkkien tiedot edeltäjistä nollataan, jotta täsmäyksen osuessa pisimpään yhteiseen alijonoon ei tule virheitä. Myös etuperin saraketarkastelussa löytyneiden korkeuskäyrien määrää pienennetään vastaamaan todellista tilannetta. Rivitarkastelussa ei löydy enempää täsmäyksiä takaperin tarkasteltaessa.

Vaiheessa 5* tarkastetaan, valmistuiko alin korkeuskäyrä, ja huomataan näin tapahtuneen. Kasvatetaan alinta etsittävää korkeuskäyrää kuvaavaa muuttujaa low2 ja laitetaan korkeuskäyrän valmistuminen muistiin. Koska takaperin saraketarkastelussa ei löytynyt valmistuneelle käyrälle täsmäyksiä, tarkastellaan toisen suunnan korkeuskäyrien mahdollista siirtotarvetta. Etuperin riveittäin löytyneen korkeimman korkeuskäyrän kynnysarvo on kuitenkin sellainen, että käyrää ei tarvitse siirtää. Yritetään etsiä täsmäystä toiselle korkeuskäyrälle, mutta sellaista ei voida löytää, kun tähän suuntaan ei löydetty ensimmäistäkään korkeuskäyrää ennen sen valmistumista. Valmistuneen korkeuskäyrän viimeisin linkki sen sijaan kopioidaan saraketarkastelun jälkeen linkkilistalle ensimmäiselle korkeuskäyrälle ainoaksi linkiksi.

Kolmannelle kierrokselle lähdettäessä tarkastetaan, onko alin korkeuskäyrä valmistunut. Se ei ole valmistunut, joten etsitään etuperin riveittäin täsmäystä toiselle korkeuskäyrälle. Sellainen löytyy sarakkeesta 4. Arvo lisätään row-Threshiin ja luodaan linkkilistalle uusi linkki. Tämän jälkeen etsitään kolmannen korkeuskäyrän täsmäyksiä, mutta sellaisia ei löydy. Ennen sarakesuuntaista tarkastelua huomataan, että alin korkeuskäyrä ei valmistunut. Saraketarkastelussa ei löydetä täsmäyksiä.

Vielä on yksi rivi tarkastamatta, joten siirrytään tarkastelemaan sitä riviä takaperin. Alin korkeuskäyrä ei valmistunut, eikä riviltä löydy yhtään täsmäystä. Vaikka ei ole enää tyhjää aluetta, jolta voisi löytyä täsmäyksiä, tarkastetaan vielä, valmistuiko alhaalta rivitarkastelun

jälkeen alin korkeuskäyrä. Se ei valmistunut, joten algoritmi on löytänyt kaikki tarvittavat täsmäykset ja ne ovat muistissa linkkilistoissa.

Viimeiseksi selvitetään pisin yhteinen alijono. Koska riveittäin löytyi useampia korkeuskäyriä kuin sarakkeittain niin ylhäältä kuin alhaaltakin, täytyy tarkastaa, pystyisivätkö jotkin korkeuskäyrät jatkamaan toisiaan. Toisiaan jatkavien korkeuskäyrien etsimisessä käytetään apumuuttujia u ja v, joita ei ole mainittu pseudokoodissa. Alustetaan muuttuja u ylhäältä löytyneiden korkeuskäyrien lukumäärällä, ja muuttujaan v tulee oikealta sarakkeittain löytyneiden korkeuskäyrien lukumäärä + 1. Ylhäältä otettiin siis ylin mahdollinen vertailtava ja alhaalta alin mahdollinen vertailtava korkeuskäyrä. Ylhäältä ensimmäiseksi vertailuun tulee toinen korkeuskäyrä, kuten myös alhaaltakin. Vertaillaan näiden kynnysarvoja ja huomataan, että molempien korkeuskäyrien viimeksi löydetyt täsmäykset voivat jatkaa toisiaan. Otetaan korkeuskäyrien järjestysnumerot muistiin ja nostetaan lippu sen merkiksi, että toisiaan jatkava mahdollisuus löytyi. Kasvatetaan v:tä eli alhaalta löydetyn korkeuskäyrän numeroa. Jatkuvuuksien etsiminen lopetetaan, sillä kolmatta korkeuskäyrää ei ole olemassa.

Kaikki pisimmän yhteisen alijonon selvittämiseen tarvittava on nyt tiedossa. Riveittäin löytyneiden toisiaan jatkavien korkeuskäyrien kaikkien edeltävien täsmäysten paikkojen on oltava tiedossa linkeissä. Lähtemällä ylhäältä viimeksi löytyneen täsmäyksen linkistä saadaan pisimmän yhteisen alijonon alkuosa väärinpäin ja alhaalta viimeksi löytyneen täsmäyksen linkistä lähtien loppuosa oikeinpäin. Linkkejä seuraamalla alkuosan täsmäykset ovat ylimmästä korkeuskäyrästä lähtien (3, 4) ja (2, 2), joten alkuosa on oikeinpäin käännetynä ”bc”. Vastaavasti loppuosan täsmäykset ovat ylimmästä korkeuskäyrästä lähtien (5, 5) ja (6, 8), joten loppuosa on ”bb”. Nämä yhdistämällä saadaan pisimmäksi yhteiseksi alijonoksi ”bcbb”.

Aika- ja tilavaativuus:

Rickin kaksisuuntainen menetelmä tekee samat operaatiot kuin yksisuuntainen versiokin, minkä lisäksi tulee muita lisätarkasteluja muutamiin kohtiin. Esiprosessointi (0) vie aikaa O(ns) ja vaiheen 1 alustukset vievät aikaa O(m). Vaiheen 2 for-silmukkaa suoritetaan O(m) kertaa. Vaiheen 2 sisällä on neljä samankaltaista vaiheparia: vaiheet 3 ja 4, vaiheet 5 ja 6, vaiheet 3* ja 4* sekä vaiheet 5* ja 6*. Näissä neljässä vaiheparissa suoritetaan samat operaatiot, joten yhden parin suoritusaikatarkastelu riittää. Otetaan tarkasteluun vaiheet 3 ja 4. Vaiheessa 3 muut paitsi korkeuskäyrän siirtäminen voidaan suorittaa vakioajassa. Vaiheessa 4 etsitään täsmäyksiä maksimissaan p korkeuskäyrälle vakioaikaisilla operaatioilla, joiden lisäksi tulevat mahdolliset korkeuskäyrän siirtämisen vaatimat operaatiot. Vaiheiden 3 ja 4 suoritusaika on O(min{pm, p(n–p)}) ilman mahdollisia korkeuskäyrien siirtoja (Ric 1994, s. 20–21). Tämä on siis myös samalla muiden edellä mainittujen vaiheparien suoritusaika poislukien mahdolliset korkeuskäyrien siirrot.

Algoritmissa korkeuskäyrien siirto suoritetaan poistamalla koko korkeuskäyrä halutusta suunnasta ja lisäämällä täsmäykset uudelle käyrälle yksitellen oikeassa järjestyksessä. Ennen lisäyksiä täsmäysten tiedot edeltäjästä poistetaan vakioaikaisella operaatiolla. Korkeuskäyriä siirretään yleensä korkeintaan muutama ja niillä on yleensä vain muutamia siirrettäviä minimaalisia täsmäyksiä. Näille on vaikea löytää kunnollista aikavaativuutta, mutta karkeita ylärajoja on mahdollista määrittää. Korkeuskäyrien siirto voi tapahtua, kun on joko löytynyt uusi korkeuskäyrä tarkastelusuuntaan tai huomataan jonkin korkeuskäyrän valmistuneen.

Koko algoritmin aikana korkeuskäyrien siirtojen hyvin karkea yläraja on luokkaa O(p) ja jokaisessa siirrossa siirtyy aivan maksimissaan m täsmäystä. Korkeuskäyrien siirtojen suoritusajalle saadaan todella karkeaksi ylärajaksi tämän algoritmin kohdalla yhteensä aikavaativuusluokka O(pm).

Kun molemmista suunnista on löytynyt riveittäin enemmän korkeuskäyriä kuin sarakkeittain tarkastellessa, pisimmän yhteisen alijonon selvittämisessä tarkastellaan korkeintaan p käyrän kynnysarvoja. Jos riveittäin löydetään toisiaan jatkavat korkeuskäyrät, selvittämisen kokonaisaika on luokkaa O(p). Kaikissa muissa tapauksissa täytyy etsiä oikea reitti pisimmän yhteisen alijonon muodostamiseksi. Etsimisen aikana kaikkia minimaalisia täsmäyksiä tutkitaan korkeintaan kertaalleen, ja lisäksi jonon merkkien selvittämiseen menee aikaa O(p), joten lopputuloksen selvittämisen aikavaativuus on luokkaa O(p+d). Aikavaativuus on sama myös ilman alun korkeuskäyrien kynnysarvotarkasteluita. Koko kaksisuuntaisen algoritmin aikavaativuus on luokkaa O(ns+min{pm, p(n–p)}+pm+d). Jos pm>p(n–p), minimilausekkeesta valitaan jälkimmäinen termi. Seuraava termi on kuitenkin pm, joka on suurempi kuin valittu pienempi suoritusaikaluokka, joten tässä tapauksessa koko minimilauseke on turha. Jos taas pm≤p(n–p), valitaan pm, jolloin termi p(n–p) on turha. Koska myöhemmin tulee joka tapauksessa termi pm, voidaan näin jättää koko minimilauseke aikavaativuusluokasta pois. Rickin algoritmin kaksisuuntaisen version lopullinen aikavaativuus on luokkaa O(ns+pm+d).

Tilavaativuudessa suorasaantitaulukoiden vaatima tila on luokkaa O(ns), ja kynnysarvotaulukot tarvitsevat tilan, joka on luokkaa O(m). Linkkilistojen yhteenlaskettu tilavaatimus sekä linkkien tilavaatimus ovat molemmat luokkaa O(d), jossa d on minimaalisten täsmäysten lukumäärä. Kaksisuuntaisen algoritmin tilavaativuus on lopulta O(ns+d). Suorasaanti- ja kynnysarvotaulukoita tulee kaksinkertainen määrä, ja linkkilistojen tilavaatimus muuttuu luokasta O(m) luokkaan O(d) verrattuna yksisuuntaiseen versioon, mutta silti tilavaativuus on samaa luokkaa kuin yksisuuntaisellakin versiolla.

3. PYA-algoritmien vertailu

Tässä luvussa testataan luvussa 2 esiteltyjen algoritmien toimintaa käytännössä. Kaikki testit on suoritettu samalla tietokoneella ja samoilla asetuksilla, jotta testien tulokset olisivat vertailukelpoisia keskenään. Testeissä on käytetty useita eri syötepareja, joista 10 paria ovat aina samankokoiset. Näin esimerkiksi mahdollisen erikoistapauksen aiheuttama suurempi tai pienempi suoritusaika ei vaikuta keskiarvoon kovin paljon. Kaikkia algoritmeja on testattu samoilla syötteillä, joten niiden väliset erot johtuvat käytännössä algoritmien lähestymistapojen eroista.

Mittaustavat:

Testeissä mitataan algoritmien suoritusaikaa millisekunnin tarkkuudella. Suoritusaika saadaan laskemalla erotus algoritmin loppumisen ja kutsumisen ajankohdista. Muistinkäyttöä mitataan Javan Runtimesta saatavien tietojen avulla. Muistinkäyttö lasketaan välittömästi algoritmin loppumisen jälkeen erotuksena totalMemory()–freeMemory(), jossa totalMemory()-kutsulla saadaan järjestelmältä tieto Javalle ositetusta muistista, ja freeMemory()-kutsulla saadaan tieto, kuinka paljon muistista on sillä hetkellä vapaana. Muistinkäyttöä mittaavat arvot eivät ole ehkä täysin tarkkoja, sillä ne ovat hetkelliset arvot algoritmin suorittamisen jälkeen. Arvot kuitenkin kuvaavat riittävän hyvin algoritmin käyttämän muistin määrää, jotta saadaan tarvittava tieto algoritmien muistinkäytön eroavaisuuksien vertailuun. Näin saatavien tulosten suuruusluokka oli sama kuin Windowsin omasta resurssienvalvonnasta seurattuna. Jokaisen mittauksen jälkeen suoritetaan Javan roskienkeräys kahteen kertaan, jotta muistissa olevat edellisen algoritmin tietueet tulisi poistettua kunnolla ennen seuraavan algoritmin suorittamista ja testaamista.

Testikokoonpano:

Koneen tyyppi: pöytätietokone
Prosessori: Intel Core I5-3570K @ 3,4 Ghz, Turbo Boost -ominaisuus kytketty pois päältä.
Keskusmuisti: 2* Corsair Vengeange Dual C DDR3 8GB, 1600 Mhz Cl10
Emolevy: Gigabyte GA-Z77X-D3H
Käyttöjärjestelmä: Windows 7 Ultimate 64-bit, Service Pack 1 (asennetut kielet suomi ja englanti)
Käyttöjärjestelmän kiintolevy: Samsung SSD 840 Series, 250 GB
Testattavien koodien kiintolevy: Samsung 1 TB SpinPoint F3 64MB
Virtalähde: LC-Power V2.2 Hyperion, 700 W
Näytönohjain: Ati Radeon HD 5870

Testitietokoneessa on Intelin kolmannen sukupolven neliydinsuoritin. Testattavat algoritmit käyttävät kerrallaan vain yhtä ydintä, joten kolme ydintä jää yli Windowsille ja muille taustalla mahdollisesti pyöriville ohjelmille ja prosesseille. Suorittimen kellotaajuudeksi asetettiin tietokoneen BIOSista vakio 3,4 GHz, ja Intelin oma ylikellotusjärjestelmä kytkettiin pois päältä testien ajaksi, jotta algoritmit suoritettaisiin samankaltaisissa olosuhteissa. Testeihin käytetyssä tietokoneessa on keskusmuistia 16 gigatavua, joista on koneen käynnistyttyä käytössä n. kaksi. Javalle ositettiin testeissä maksimissaan 14 Gt koneen keskusmuistista. Wagnerin ja Fischerin algoritmi suoritettiin 40 000:n mittaisilla syötteillä pariin kertaan, sillä

tämän tiedettiin alustavien testien perusteella olevan jo lähellä muistin maksimimäärää. Samalla myös testattiin, että oikea fyysinen keskusmuisti ei lopu kesken, sillä keskusmuistin loppuessa loppuosa ”keskusmuistista” olisi hidasta levymuistia. Koneen annettiin käydä muutama minuutti ennen testien suorittamista, jotta ylimääräiset ohjelmien päivitysten tarkastukset ym. olisi tehty ennen testien aloittamista. Kaikki ylimääräiset ohjelmat sekä levyjä tarkastelevat tiedostojen synkronointiohjelmat suljettiin testien ajaksi, jotta ne eivät vaikuttaisi lopputuloksiin. Testit suoritettiin komentokehotteesta ohjelmalla, joka tekee 10 testikierrosta, joista jokaisella kierroksella samalla syöteparilla testataan kaikki halutut algoritmit. Jokaiselle kierrokselle luodaan uudet satunnaiset yhtä pitkät syötteet. Kaikki saadut arvot tallennetaan suoraan Excel-taulukkoon, jonka avulla tuloksille lasketaan keskiarvot.

Testisyötteet:

Vertailuissa käytetyt syötteet luotiin yksinkertaisella algoritmilla, joka arpoi halutun määrän merkkejä annetusta merkistöstä. Käytössä oli kaksi erilaista merkistöä: 36 merkin pituinen merkistö (merkit a–z ja 0–9) sekä 14 merkin pituinen merkistö (merkit a–n). Vertailuissa käytettyjen syötteiden pituudet olivat 500, 1 000, 1 500, 2 000, 2 500, 5 000, 10 000, 20 000, 30 000, 40 000 ja 50 000, joista pisimmät olivat käytössä vain vertaillessa eripituisilla syötteillä.

Luvuissa 3.1.–3.3. tutkitaan samankaltaisten algoritmien toimintaa yksi- ja kaksisuuntaisena. Luvussa 3.4. tarkastellaan kaksisuuntaisuuden hyötyjä ja haittoja tämän tutkielman algoritmien perusteella. Luvussa 3.5. vertaillaan kaikkia saatuja tuloksia keskenään.

3.1. Wagner–Fischer ja Hirschbergin lineaarinen menetelmä

Wagnerin ja Fischerin kehittämä menetelmä on hyvin suoraviivainen ja yksinkertainen. Se päivittää koko ajan kahta taulukkoa, jotta lopputulos olisi lopulta helppo ja nopea löytää. Algoritmi tutkii kertaalleen jokaisen merkkiparin, joista se kirjaa tulokset. Algoritmi ei tee mitään muuta kirjanpitoa taulukoiden ylläpitämisen lisäksi, mutta toisaalta algoritmi käy läpi myös kaikki sellaiset merkkiparit, joista ei voi löytyä täsmäyksiä. Muistia algoritmi vaatii todella paljon kahden suuren neliöllisesti kasvavan tulostaulukon ylläpitämiseen. Tämä algoritmi käytti muistia eniten kaikista testattavista, ja muistin määrä loppui alustavissa testeissä ensimmäisenä kesken syötteiden pituuksia kasvatettaessa.

Hirschbergin kehittämä lineaarisen tilan menetelmä käyttää osittain samaa suoraviivaista tekniikkaa hyväksi; se tosin kutsuu menetelmää useasti pienempien syötteiden osien kanssa. Se myös lähtee tarkastelemaan syötteitä molemmista suunnista. Lineaaritilaiseksi tarkoitettu menetelmä ei tallenna isoiksi kasvavia taulukoita, vaan se ylläpitää aina vain kahta viimeisintä riviä. Tämä karsii jonkin verran prosessointia taulukoiden ylläpidosta, jonka lisäksi rekursiivisesti saatava lopputulos ei vaadi enää lopussa erillistä etsimistä. Algoritmi vaatii kuitenkin hiukan enemmän muuta prosessointia, jotta pystytään löytämään aina oikeat katkaisukohdat seuraaville rekursiivisille kutsuille. Algoritmi myös tutkii samoja osia useampaan kertaan; tosin se tiputtaa rekursiokutsuista nopeasti pois sellaisia syötteiden osia, joista ei löydy täsmäyksiä.

Testien tulosten keskiarvot ovat taulukossa 17. Pienillä syötteillä suoritusajat ovat molemmilla algoritmeilla samaa suuruusluokkaa ja hyvin lähellä toisiaan. Syötekoon kasvaessa ja

pienemmällä merkistöllä Hirschbergin lineaarinen menetelmä alkaa osoittautua hiukan tehokkaammaksi. Muistinkäytössä on valtava ero kaikilla syötteillä. Tässä tapauksessa kaksisuuntaisen lähestymistavan vaikutus suoritusaikaan alkaa näkyä isoilla syötteillä. Hirschbergin lineaarinen menetelmä on suurilla syötteen pituuksilla nopeampi kuin Wagner–Fischer. Kaksisuuntaisen menetelmän vaikutus muistinkäyttöön on vielä huomattavasti suurempi. Hirschbergin lineaarinen menetelmä pystyy laskemaan samalla muistimäärillä täysin eri kokoluokan syötteitä kuin Wagnerin ja Fischerin menetelmä.

Taulukko 17. Wagnerin ja Fischerin menetelmän ja Hirschbergin lineaarisen menetelmän suoritusaikojen sekä muistinkäytön lasketut keskiarvot, pyöristettyinä kahden merkitsevän numeron tarkkuudella tai kokonaislukuina pienillä syötteillä, 10:llä eri syötteen pituudella sekä kahdella eri merkistön pituudella. Oikeassa sarakkeessa näkyy pisimmän yhteisen alijonon keskimääräinen pituus testeissä kullakin valitulla syötepituudella.

syött. pituus	Wagner – Fischer		Hirschberg lineaarinen		PYA pituus ka.
syötteiden merkistön koko 36: a – z, 0 – 9					
500	2 ms	2 MB	5 ms	1 MB	136
1000	6 ms	6 MB	8 ms	1 MB	278
1500	12 ms	14 MB	14 ms	1 MB	418
2000	20 ms	24 MB	23 ms	2 MB	559
2500	31 ms	37 MB	34 ms	2 MB	699
5000	130 ms	150 MB	130 ms	5 MB	1405
10000	490 ms	600 MB	510 ms	17 MB	2827
20000	1900 ms	2400 MB	2100 ms	64 MB	5660
30000	4900 ms	5300 MB	4700 ms	66 MB	8495
40000	9700 ms	9400 MB	8400 ms	61 MB	11343
syötteiden merkistön koko 14: a – n					
500	3 ms	2 MB	4 ms	1 MB	203
1000	7 ms	7 MB	7 ms	1 MB	412
1500	13 ms	14 MB	13 ms	1 MB	617
2000	23 ms	25 MB	21 ms	2 MB	826
2500	35 ms	38 MB	31 ms	2 MB	1031
5000	140 ms	150 MB	120 ms	5 MB	2074
10000	550 ms	620 MB	480 ms	19 MB	4156
20000	2200 ms	2400 MB	2000 ms	68 MB	8325
30000	5400 ms	5500 MB	4500 ms	71 MB	12507
40000	19000 ms	9500 MB	8000 ms	65 MB	16677

Taulukko 18. Wagnerin ja Fischerin menetelmän ja Hirschbergin lineaarisen menetelmän suoritusaikojen ja muistinkäytön keskiarvot, pyöristettyinä kahden merkitsevän numeron tarkkuuteen, erikokoisten syötteiden järjestystä vaihdettaessa. Toisen syötteen pituus on 10 000 merkkiä ja toisen 50 000 merkkiä. Pisimmän yhteisen alijonon pituuden keskiarvo on näillä syötteillä 5905 merkkiä.

Wagner–Fischer, X ≤ Y		Wagner–Fischer, X > Y		HB lineaar., X ≤ Y		HB lineaar., X > Y	
2400 ms	3000 MB	2500 ms	3000 MB	2700 ms	72 MB	2500 ms	72 MB

Merkistön koon vaikutus näkyy molempien algoritmien suoritusajoissa. Wagnerin ja Fischerin menetelmässä suoritusaika kasvaa yllättävän paljon merkistön pienentyessä. Täsmäyksen löytyessä arvo lasketaan edellisellä sarakkeella ja rivillä olevasta solusta, joten operaatioon voisi kulua hieman enemmän aikaa kuin edellisen ja yläpuolela olevan solun vertailuun. Siksi täsmäysten tallentaminen taulukoihin saattaa viedä enemmän aikaa täsmäysten määrän kasvaessa. PYAn etsinnässä taulukossa yläviistoon liikkuminen on monimutkaisempi operaatio kuin ylös tai vasemmalle siirtyminen, joten suoritusaika saattaa kasvaa hieman tässäkin vaiheessa täsmäysten lukumäärän kasvaessa. Suuremmilla syötteillä vaikutukset kasvavat vielä suuremmiksi. Hirschbergin lineaarinen menetelmä taas näyttäisi tehostuvan pienemmällä merkistöllä hieman, mikä johtuu todennäköisesti siitä, että syötteiden jakaminen pienempiin osiin tapahtuu tasaisemmin niiden keskeltä. Algoritmien muistinkäyttöön merkistön koon muuttamisella ei ole käytännössä merkittävää vaikutusta.

Taulukossa 18 ovat tulokset testistä, jossa vaihdettiin selvästi eripituisten syötteiden järjestystä. Tavallisesti algoritmeissa oletetaan, että ensimmäinen syöte on korkeintaan yhtä pitkä kuin jälkimmäinen syöte. Wagnerin ja Fischerin menetelmässä järjestyksellä ei pitäisi olla mitään merkitystä, eikä testeissä merkittävää eroa syntynytkään. Hirschbergin lineaarisessa menetelmässäkään syötteiden järjestyksen ei pitäisi vaikuttaa kovin merkittävästi. Kun suurempi syöte annettiin ensimmäisenä, suoritusajassa havaittiin pieni tehostuminen. Tämä johtuu todennäköisesti siitä, että testisyötteet jakautuivat rekursiivisille kutsuille paremmin puoliksi. Ero on kuitenkin niin pieni, että näillä syötteen pituuksilla asialla ei ole vielä merkitystä. Todennäköisesti syötteiden pituudet ovat normaalisti paljon lähempänä toisiaan. Syötteiden järjestyksen vaihtuminen ei vaikuttanut kummankaan algoritmin muistinkäyttöön.

3.2. Hirschbergin menetelmä yksi- ja kaksisuuntaisena

Hirschbergin algoritmi etsii minimaalisia täsmäyksiä korkeuskäyrittäin. Se karsii sellaisten merkkiparien vertailut pois, mistä ei voi löytyä täsmäyksiä. Tästä syystä suurin osa merkkipareista jää yleensä kokonaan tarkastelun ulkopuolelle. Kääntöpuolena algoritmi käyttää suuria taulukoita, joiden avulla pidetään muistissa mahdolliset täsmäyskohdat sekä löytyneet täsmäykset korkeuskäyrittäin. Täsmäysten etsimisen lisäksi vaaditaan alussa esiprosessointia esiintymätaulukoiden muodostamista varten sekä jatkuvaa muuttujien ja taulukoiden päivittämistä.

Taulukossa 19 ovat testien lopputulosten keskiarvot, joista nähdään kaksisuuntaisen version suoritusaikojen olevan 20 000:n pituisiin syötteisiin asti keskimäärin n. puolitoistakertaisia yksisuuntaisen version arvoihin verrattuina. Tätä suuremmilla syötteillä ero alkaa pienentyä huomattavasti, mutta silti yksisuuntainen versio on vielä selvästi nopeampi. Vaikka kaksisuuntainen algoritmi tekee käytännössä saman työn kuin yksisuuntainenkin, kaksisuuntaisessa tulee selvästi ylimääräistä prosessointia, kun joudutaan tarkkailemaan toisen suunnan viimeisintä korkeuskäyrää joka tilanteessa. Kaksisuuntaisessa versiossa alkuprosessointi on hieman pidempi prosessi kuin yksisuuntaisessa versiossa. Suurilla syötteillä esiprosessoinnin merkitys suoritusajoissa vähenee selvästi, mikä saatta vaikuttaa hieman myös suoritusaikojen suhteellisen eron pieneneminen yksi- ja kaksisuuntaisen version välillä.

Muistinkäytössä ei ole suuria eroja näiden algoritmien välillä. Molemmissa algoritmeissa muodostetaan käytännössä yhtä suuret taulukot, joten muistinkäytön pitäisikin olla suurin

piirtein sama molemmilla menetelmillä. Alkuperäinen versio on testeissä muutamilla syötteillä taloudellisempi, kun taas kaksisuuntainen vaatii hieman vähemmän resursseja joillain muilla syötteillä. Periaatteessa kaksisuuntaisen pitäisi käyttää aina enemmän muistia, joten tulos on hieman outo, tosin erot eivät ole kovin suuria. Javan roskienkeräilijän ajaminen kahteen kertaan ei välttämättä riitä siihen, että kaikki ylimääräinen muistissa oleva poistettaisiin kunnolla – etenkin, kun maksimimuistimääräksi on määritetty näissä testeissä huomattavasti suurempi arvo kuin kumpikaan näistä algoritmeista käyttää.

Taulukko 19. Hirschbergin menetelmän yksi- ja kaksisuuntaisen version suoritusaikojen sekä muistinkäytön lasketut keskiarvot, pyöristettyinä kahden merkitsevän numeron tarkkuudella tai kokonaislukuina pienillä syötteillä, 10:llä eri syötteen pituudella sekä kahdella eri merkistön pituudella. Oikeassa sarakkeessa näkyy pisimmän yhteisen alijonon keskimääräinen pituus testeissä kullakin valitulla syötepituudella.

syött. pituus	Hirschberg alkuperäinen		Hirschberg 2-suuntainen		PYA pituus ka.
syötteiden merkistön koko 36: a – z, 0 – 9					
500	14 ms	1 MB	21 ms	1 MB	136
1000	37 ms	5 MB	54 ms	5 MB	278
1500	38 ms	9 MB	95 ms	9 MB	418
2000	60 ms	17 MB	150 ms	17 MB	559
2500	110 ms	25 MB	120 ms	25 MB	699
5000	330 ms	110 MB	540 ms	110 MB	1405
10000	1400 ms	600 MB	2100 ms	590 MB	2827
20000	5600 ms	2600 MB	8400 ms	2800 MB	5660
30000	14000 ms	5700 MB	20000 ms	5700 MB	8495
40000	26000 ms	7200 MB	31000 ms	6800 MB	11343
syötteiden merkistön koko 14: a – n					
500	20 ms	1 MB	26 ms	1 MB	203
1000	42 ms	5 MB	63 ms	5 MB	412
1500	62 ms	9 MB	132 ms	10 MB	617
2000	84 ms	20 MB	128 ms	19 MB	826
2500	130 ms	38 MB	220 ms	36 MB	1031
5000	480 ms	180 MB	750 ms	200 MB	2074
10000	2100 ms	780 MB	2900 ms	860 MB	4156
20000	8000 ms	2900 MB	12000 ms	3200 MB	8325
30000	20000 ms	6100 MB	25000 ms	4300 MB	12507
40000	42000 ms	6500 MB	49000 ms	6800 MB	16677

Taulukko 20. Hirschbergin menetelmän yksi- ja kaksisuuntaisen version suoritusaikojen ja muistinkäytön keskiarvot, pyöristettyinä kahden merkitsevän numeron tarkkuuteen, erikokoisten syötteiden järjestystä vaihdettaessa. Toisen syötteen pituus on 10 000 merkkiä ja toisen 50 000 merkkiä. Pisimmän yhteisen alijonon pituuden keskiarvo on näillä syötteillä 5905 merkkiä.

HB 1-suunt., X ≤ Y		HB 1-suunt., X > Y		HB 2-suunt., X ≤ Y		HB 2-suunt., X > Y	
6800 ms	3000 MB	9500 ms	10000 MB	9700 ms	650 MB	13000 ms	10000 MB

Hirschbergin algoritmi on hyvä esimerkki siitä, että eripituisten syötteiden järjestyksellä on suuri merkitys. Taulukossa 20 ovat tulokset samoilla syötteillä, jotka on suoritettu eri järjestyksessä. Alkuperäisen yksisuuntaisen Hirschbergin menetelmän suoritusaika kasvaa n. 40 % ja samalla muistinkäyttö yli kolminkertaistuu, kun pidempi syöte annetaankin ensimmäisenä. Kaksisuuntaisen version suoritusaika kasvaa myös vajaat 40 %, mutta muistinkäyttö kasvaa suhteessa vielä huomattavasti enemmän. Kun syötteet annetaan kokojen puolesta väärässä järjestyksessä pidempi syöte ensimmäisenä, esiprosessoinnin määrä pienenee, mutta sen osuus on kuitenkin pieni kokonaissuoritusajasta. Minimaalisten täsmäysten etsimiseen kuluva suoritusaika taas pitenee huomattavasti, ja PYAn selvittäminen isompien taulukoiden avulla vie enemmän aikaa kuin syötteiden ollessa oikeassa järjestyksessä, joten kokonaissuoritusaika kasvaa selvästi. Tulostaulukko D on paljon suurempi syötteiden ollessa väärässä järjestyksessä, joten myös muistinkäyttö kasvaa huomattavan paljon.

Kun pienempi syöte annetaan ensin, kaksisuuntainen versio vie hieman erikoisesti selvästi vähemmän muistia kuin alkuperäinen, mutta syötteiden järjestyksen vaihduttua muistia kuluu yhtä paljon. Tulospari on outo, sillä tulosparin tulokset poikkeavat toisistaan ns. ”väärään” suuntaan. Muistin mittausmenetelmä ei välttämättä ole toiminut näiden tulosten kohdalla tarpeeksi hyvin. Muistia oli käytössä huomattavasti enemmän kuin mitä algoritmit käyttivät, joten Javan roskienkeräilijän toiminta on myös mitä ilmeisimmin ollut ainakin osasyy outoihin tuloksiin. Algoritmit kuitenkin toimivat, vaikka syötteet annettaisiin ”väärässä järjestyksessä”. Syötteiden pituuksien vertailu ennen algoritmin suorittamista kuitenkin kannattaa, sillä yhdellä vertailuoperaatiolla voidaan säästää sekä suoritusajassa että etenkin muistinkäytössä todella merkittävästi resursseja. Kaksisuuntainen menetelmä käytti testien perusteella yllättävän vähän muistia verrattuna yksisuuntaiseen, kun pienempi syöte annettiin ensin. Tähän syynä on todennäköisesti Javan roskienkeräilijän toiminta.

3.3. Rickin menetelmä yksi- ja kaksisuuntaisena

Rickin menetelmä etsii minimaalisia täsmäyksiä riveittäin ja sarakkeittain. Yhdeltä riviltä tai yhdestä sarakkeesta voi löytyä useamman korkeuskäyrän täsmäyksiä. Rickin alkuperäisessä menetelmässä tehdään suorasaantitaulukot, jotka vievät suuren osan algoritmin käyttämästä muistista. Minimaalisista täsmäyksistä pidetään muistissa suoraan saavutettavina vain viimeisiä kultakin korkeuskäyrältä. Tämän lisäksi muistissa ovat linkit kaikkiin tarvittaviin edellisiin täsmäyksiin. Kaksisuuntaisessa menetelmässä tehdään omat suorasaantitaulukot myös toisen suunnan tarkastelua varten, minkä lisäksi kaikki täsmäykset ja niiden linkit pidetään muistissa loppuun asti.

Taulukosta 21 selviävät Rickin yksi- ja kaksisuuntaisten versioiden erot. Kaksisuuntaisuus vaatii enemmän alkuprosessointia uusien suorasaantitaulukoiden luomisen myötä. Kaksisuuntaisessa lähestymisessä tarkastellaan käyrien siirtämistä aina ylimmän korkeuskäyrän täsmäyksen löytyessä tai jonkin korkeuskäyrän valmistuessa pienillä vertailuilla. Näin täsmäysten etsimiseen tulee jokaiseen vaiheeseen hiukan enemmän prosessointia. Tarvittaessa myös siirretään toisesta suunnasta löytyneitä kokonaisia korkeuskäyriä toiseen taulukkoon, mikä kasvattaa suoritusaikaa huomattavasti yksisuuntaiseen verrattuna. Myös kaikkien löydettyjen minimaalisten täsmäysten linkkien päivittäminen ja pitäminen taulukoissa suoraan

saavutettavina vievät enemmän aikaa ja muistia kuin vain viimeisten löydettyjen täsmäysten linkkien päivittäminen.

Taulukko 21. Rickin menetelmän yksi- ja kaksisuuntaisten versioiden suoritusaikojen sekä muistinkäytön lasketut keskiarvot, pyöristettyinä kahden merkitsevän numeron tarkkuudella tai kokonaislukuina pienillä syötteillä, 10:llä eri syötteen pituudella sekä kahdella eri merkistön pituudella. Oikeassa sarakkeessa näkyy pisimmän yhteisen alijonon keskimääräinen pituus testeissä kullakin valitulla syötepituudella.

syött. pituus	Rick alkuperäinen		Rick 2-suuntainen		PYA pituus ka.
syötteiden merkistön koko 36: a – z, 0 – 9					
500	2 ms	1 MB	6 ms	1 MB	136
1000	2 ms	1 MB	13 ms	2 MB	278
1500	5 ms	2 MB	25 ms	3 MB	418
2000	6 ms	3 MB	32 ms	6 MB	559
2500	10 ms	5 MB	49 ms	7 MB	699
5000	36 ms	16 MB	93 ms	27 MB	1405
10000	100 ms	70 MB	430 ms	110 MB	2827
20000	320 ms	270 MB	2300 ms	410 MB	5660
30000	820 ms	540 MB	7500 ms	820 MB	8495
40000	1500 ms	930 MB	17000 ms	1400 MB	11343
syötteiden merkistön koko 14: a – n					
500	1 ms	1 MB	6 ms	1 MB	203
1000	4 ms	2 MB	18 ms	3 MB	412
1500	6 ms	3 MB	31 ms	6 MB	617
2000	9 ms	6 MB	49 ms	8 MB	826
2500	15 ms	9 MB	82 ms	14 MB	1031
5000	56 ms	32 MB	130 ms	49 MB	2074
10000	150 ms	120 MB	770 ms	180 MB	4156
20000	540 ms	510 MB	5200 ms	750 MB	8325
30000	1200 ms	1100 MB	16000 ms	1600 MB	12507
40000	2200 ms	240 MB	37000 ms	2400 MB	16677

Alkuperäisessä yksisuuntaisessa versiossa pisimmän yhteisen alijonon selvittäminen on yksinkertainen ja nopea prosessi. Kaksisuuntaisessa joudutaan etsimään mahdollisia toisiaan jatkavia korkeuskäyriä sekä testaamaan mahdollisesti monia lähtöpisteitä ennen pisimmän yhteisen alijonon selviämistä, mikä tarkoittaa paljon ylimääräistä työtä yksisuuntaiseen menetelmään verrattuna. Yksisuuntainen on suoritusajoiltaan siis selvästi kaksisuuntaista nopeampi kaikilla syötteen pituuksilla, ja ero vielä kasvaa selvästi syötteiden pituuksien kasvaessa. Pahimmillaan kaksisuuntaisen menetelmän suoritusaika kasvaa jo eri suuruusluokkaan kuin yksisuuntaisella menetelmällä. Kun merkistön kokoa pienennetään, yksisuuntaisen menetelmän suoritusaika kasvaa suurin piirtein lineaarisesti pisimmän yhteisen alijonon pidentymisen kanssa. Rickin menetelmän suoritusaikaan vaikuttaa voimakkaasti suhdeluku $\frac{\text{PYAn pituus}}{\text{m}}$. Kun aletaan lähestyä arvoa 1, algoritmi alkaa toimia tehokkaammin. Nyt PYAn pituus on kuitenkin niin kaukana syötteiden pituudesta, että PYAn piteneminen lisää suoritusaikaa. Kaksisuuntaisen menetelmän suoritusaika kasvaa vielä enemmän täsmäysten

lisääntyessä ja pisimmän yhteisen alijonon pidentyessä, etenkin isoilla syötteillä ero on jo huomattavan suuri suuremmalla merkistöllä tehtyihin testeihin.

Muistinkäytössä yhteen suuntaan etenevä menetelmä on selvästi taloudellisempi kuin kahdesta suunnasta etenevä menetelmä. Kaksisuuntainen menetelmä käyttää yleisesti n. 50 % enemmän muistia kuin yksisuuntainen. Yksisuuntainen menetelmä on jostain syystä käyttänyt huomattavan vähän muistia 40 000:n pituisilla syötteillä pienemmällä merkistön koolla. Javan roskienkeräilijän ajaminen on voinut olla ehkä tehokasta ennen tätä algoritmia, tai sitten ohjelma on jo alkanut ennakoida seuraavaa roskien poistoa ennen muistin tilanteen tarkistamista. Javalle annetun muistin määrä oli jo melkein eri kokoluokkaa kuin yksisuuntaisen algoritmin vaatima määrä. Osa eroista saattaakin johtua siitä, että kaikkea ylimäräistä ei ole tarvinnut poistaa jokaisessa välissä, koska muistia on niin paljon käytössä. Merkistön koon pienentäminen vähentää esiprosessoinnin määrää, mutta samalla täsmäykset lisääntyvät ja pisin yhteinen alijono pidentyy. Suorasaantitaulukoiden pieneneminen ei riitä kompensoimaan lisääntyneiden täsmäysten vaatimaa muistinkäyttöä, ja suoritusajan kasvu täsmäysten lisääntyessä on suurempi kuin esiprosessoinnissa säästetty aika. Näin syötemerkistön koon pienentyessä suoritusaika ja muistinkäyttö kasvavat. Yksisuuntaisen menetelmän tapauksessa muistinkäyttöä ja suoritusaikaa kasvattava vaikutus on selvästi pienempi kuin kaksisuuntaisella menetelmällä.

Rickin menetelmä perustuu siihen oletukseen, että ensimmäinen syöte on korkeintaan yhtä pitkä kuin toinen syöte. Jos syötteiden järjestyksen vaihtaa päinvastaiseksi, ohjelma kaatuu suoritettaessa, sillä joidenkin taulukoiden koko on tällaisissa tapauksissa mitoitettu liian pieneksi. Kaatuminen todettiin koittamalla suorittaa algoritmit samoilla 10 000:n ja 50 000:n pituisilla syötteillä kuin muutkin algoritmit. Jos taulukot jostain syystä hyvin pienillä syötteillä ja vain harvoilla täsmäyksillä riittäisivät, osa syötteiden merkeistä saatettaisiin tarkastella useampaan kertaan. Tästä syystä myös lopputulos saattaisi olla väärä, joten se ei ole kannattavaa missään tapauksessa. Rickin menetelmien kanssa on siis hyvin tärkeää, että syötteet annetaan oikeassa järjestyksessä. Syötteiden pituuksien tarkastaminen alussa ennen mitään muuta toimenpidettä olisi hyvä idea.

3.4. Kaksisuuntaisten menetelmien hyötyjä ja haittoja

Tässä tutkielmassa on esitelty ja tutkittu kolmea algoritmia, jotka toimivat kahteen suuntaan. Nämä ovat Hirschbergin lineaarinen menetelmä, Hirscbergin menetelmä kaksisuuntaisena ja Rickin menetelmä kaksisuuntaisena. Näistä Hirschbergin lineaarinen menetelmä on ainoa, jota ei ole kehitetty tätä työtä varten. Hirschbergin tapauksessa tarkoituksena oli testata, parantaisiko kahdesta suunnasta lähtevä tarkastelu alkuperäisen yksisuuntaisen version suoritusaikaa. Rickin yhteen suuntaan toimiva menetelmä taas on havaittu jo aikaisemmin nopeaksi (Ber 2000), ja Rick oli esittänyt jo idean kahteen suuntaan toimivasta versiosta (Ric 2000). Tosin tämä Rickin ehdotus kaksisuuntaisesta versiosta olisi ollut lineaaritilassa toimiva menetelmä, joka ei olisi pitänyt muistissa löytyneitä täsmäyksiä, vaan ne olisi etsitty lopussa uudelleen sopivien keskipisteiden löydyttyä.

Hirschbergin ja Rickin kaksisuuntaisissa versioissa on neljä yhteistä asiaa: ne pitävät kaikki löydetyt minimaaliset täsmäykset muistissa, ne tarkastelevat toisesta suunnasta saatuja viimeisimpiä tuloksia sekä niiden suoritusaika on huonompi ja ne vievät enemmän muistia kuin

niiden yksisuuntaiset vastineet. Suurempi muistinkäyttö selittyy helposti isommilla taulukoilla, joissa tiedot ovat muistissa. Suoritusaikaan vaikuttaa eniten toisen suunnan etenemisen seuraaminen, jotta samaa aluetta ei tutkittaisi turhaan useaan kertaan. Näin kaikki syötteiden muodostaman (Rickin tapauksessa kuvitteellisen) taulukon alueet tulee tutkittua käytännössä kertaalleen, mutta yksisuuntaiseen menetelmään verrattuna lisäksi tulee muutamia pieniä testejä jokaiseen vaiheeseen. Myös pisimmän yhteisen alijonon selvittäminen isommista ja monimutkaisemmista taulukoista vie enemmän aikaa kuin yhden suunnan vastaavat operaatiot. Osa näistä kahteen suuntaan tehtävistä tarkasteluista voitaisiin ehkä rinnakkaistaa, mutta osa vaatii kuitenkin tietoa toisen suunnan etenemisestä. Rinnakkaistamiseen vaadittava koodi tulisi kuitenkin vielä todella paljon monimutkaisemmaksi kuin tämä yhden asian kerrallaan suorittava koodi. Ei ole edes varmaa, että rinnakkaisuudesta saataisiin mitään konkreettista hyötyä. Tästä syystä asiaa ei mietitä tässä tutkielmassa tämän enempää.

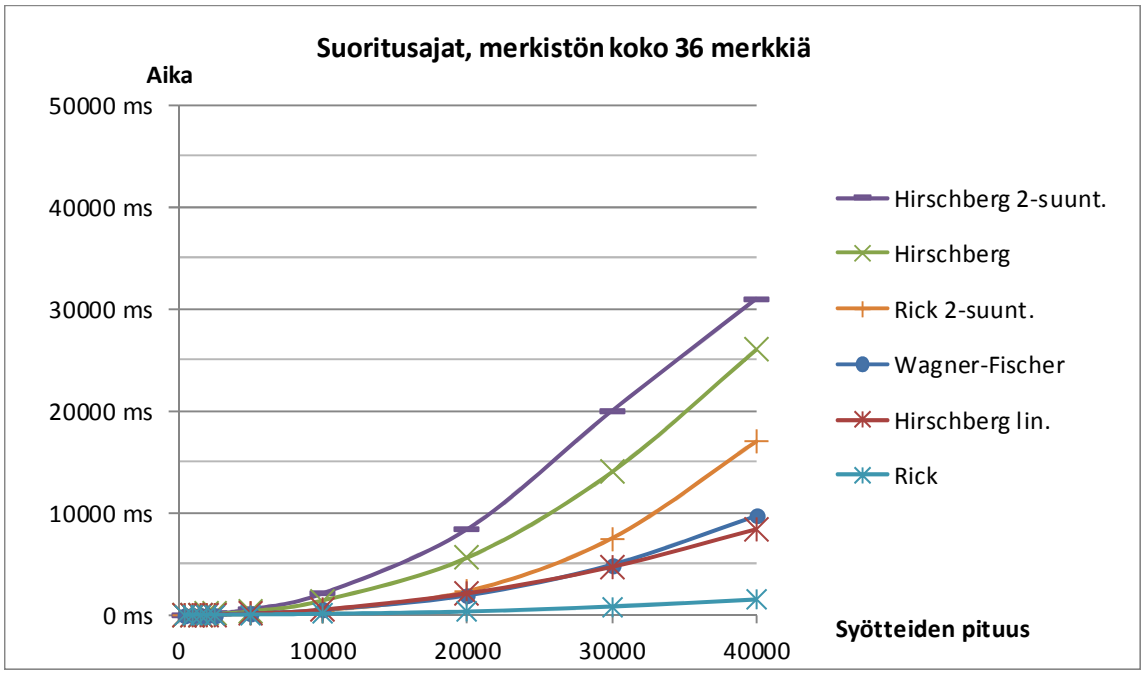
Hirschbergin lineaarinen algoritmi eroaa kahdesta muusta tutkielman kaksisuuntaisesta algoritmista. Sen lähestymislogiikka on kuitenkin jonkin verran erilainen kuin suurin piirtein vastaavaa logiikkaa yhteen suuntaan käyttävässä Wagnerin ja Fischerin algoritmissa. Rekursiivisien kutsujen vuoksi samoja alueita tutkitaan pienissä palasissa useita kertoja, mutta samalla säästyy koko ajan huomattavasti suoritusaikaa, kun ei tarvitse päivittää monta suurta taulukkoa. Hirschbergin lineaarinen menetelmä on näistä kaksisuuntaisista menetelmistä myös ainoa, jolla on yksisuuntaista verrokkiaan parempi suoritusaika. Lineaaritilaisena myös sen muistinkäyttö on hyvin taloudellista ja reilusti vastinettaan pienempi.

Kaikkien löydettyjen (minimaalisten) täsmäysten pitäminen muistissa vaatii siis reilusti sekä muistia että suoritusaikaa. Taulukoiden tekemiseen ja päivittämiseen kuluva aika on todella pitkä verrattuna pisimmän yhteisen alijonon selvittämiseen tarvittavaan aikaan valmiista taulukoista. Jos riittää pelkkä pisimmän yhteisen alijonon pituus, taulukot ovat tällöin myös turhia. Jos kaksisuuntaisessa versiossa haluaa välttää saman alueen tarkastelua useaan kertaan, vaaditaan tietoa toisen suunnan etenemisestä. Tämä taas vaatii suuria taulukoita, niiden ylläpitämistä ja arvojen tarkastelua, jotka yhdessä vievät huomattavasti enemmän resursseja kuin samalla logiikalla vain yhteen suuntaan eteneminen. Kaksisuuntaisuuden hyödyt tulevat esiin, kun tutkitaan useampia pienempiä alueita eikä pidetä kirjaa kaikista täsmäyksistä.

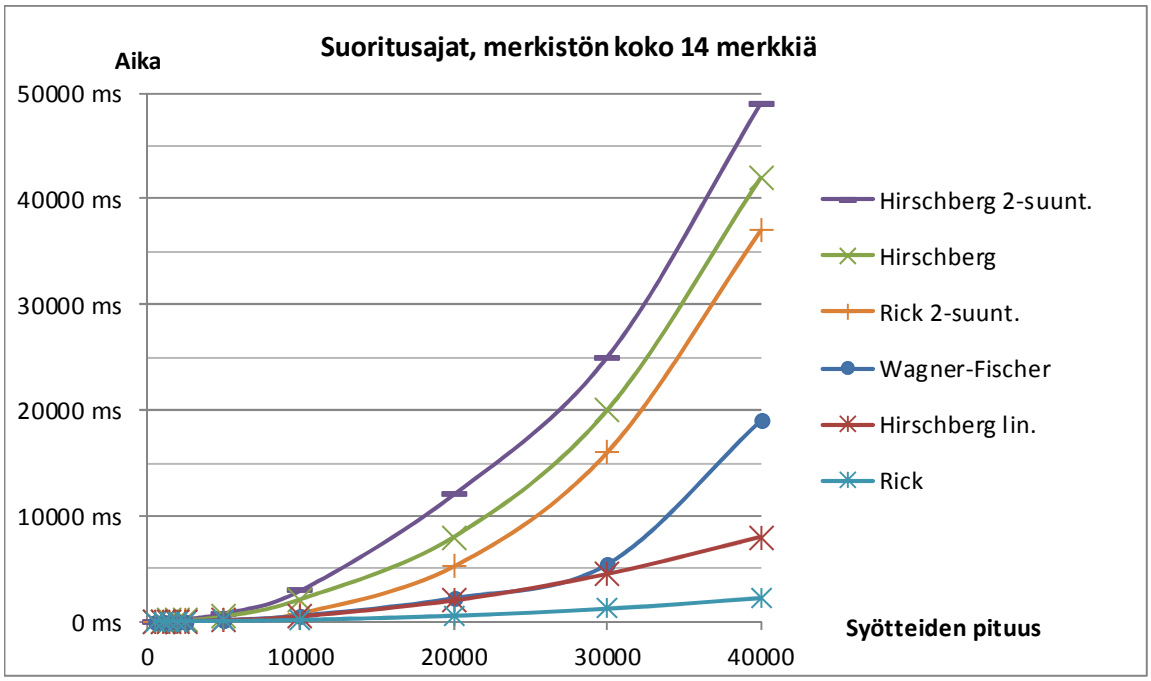
3.5. Tulosten tarkastelua

Testien perusteella tätä tutkielmaa varten tehdyt kaksisuuntaiset menetelmät olivat hyvin hitaita etsimään pisintä yhteistä alijonoa. Hirschbergistä tehty kaksisuuntainen versio oli kaikista hitain ja Rickistä tehty kaksisuuntainen versio oli kolmanneksi hitain. Näiden välissä oli suoritusaikojen perusteella Hirschbergin alkuperäinen yksisuuntainen menetelmä. Nämä kolme erottuivat selvästi toisista kolmesta syötteiden koon kasvaessa, jolloin myös täsmäysten määrä ja pisimmän yhteisen alijonon pituus kasvoivat. Suoritusajoissa Rickin yksisuuntainen menetelmä oli täysin ylivoimainen kaikkiin muihin testattaviin algoritmeihin verrattuna. Syötteiden kasvaessa suuremmiksi Hirschbergin lineaarinen menetelmä osoittautui toiseksi nopeimmaksi. Wagnerin ja Fischerin kehittämä suoraviivainen menetelmä oli suuremmalla merkistön koolla lähes yhtä nopea kuin Hirschbergin lineaarinen menetelmä, mutta

täsmäysten määrän ja pisimmän yhteisen alijonon pituuden kasvaessa ero kasvoi huomattavasti suuremmaksi Hirschbergin lineaarisen menetelmän eduksi.



Kuva 1. Algoritmien suoritusajat erilaisilla syötteiden pituuksilla, kun merkistön koko on 36 merkkiä.



Kuva 2. Algoritmien suoritusajat erilaisilla syötteiden pituuksilla, kun merkistön koko on 14 merkkiä.

Kuvissa 1 ja 2 on esitetty graafisesti syötteiden pituuden kasvamisen vaikutus suoritusaikaan kahdella eri merkistön koolla. Hirschbergin yksi- ja kaksisuuntaisen version sekä Rickin kaksisuuntaisen version käyrät nousevat huomattavasti jyrkemmin merkistön koon pienentyessä. Myös Wagner–Fischerin käyrä nousee jo vähän jyrkemmin 30 000 pituisien syötteiden kohdalla, ja 40 000 pituisien syötteiden kohdalla se nousee selvästi jyrkemmin kuin suuremmalla merkistöllä. Rickin yksisuuntaisen menetelmän kohdalla merkistön koon pieneneminen ei kasvata suoritusaikaa merkittävästi 40 000 pituisten syötteidenkään kohdalla. Tämä johtuu osittain siitä, että suorasaantitaulukoiden vaatima esiprosessointi vähenee. Hirschbergin lineaarisen menetelmän kohdalla tapahtuu jopa pieni algoritmin tehostuminen, kun merkistö supistuu.

Muistinkäytön mittausten menetelmä ei ollut ehkä paras mahdollinen, mutta toisaalta tarkoituksena ei ollut saada mahdollisimman täydellistä tarkkaa lukemaa, vaan kohtuullisen hyvät vertailukelpoiset arvot riittivät. Javalle annettiin käyttöön muistia yhteensä 14 gigatavua, joten sitä olisi ollut käytössä koko ajan vielä reilusti enemmän kuin nämä algoritmit käyttivät. Kaikki algoritmit suoritettiin peräkkäin samoilla syötteillä, joten muistissa oli mittaushetkellä mahdollisesti useamman algoritmin tietueita, vaikka Javan roskienkeräilijä ajettiinkin algoritmien suorittamisen välissä. Roskienkeräilijän kutsuminen ei kuitenkaan takaa kaiken ylimääräisen tiedon poistamista, sillä sen kutsuminen on Javalle vain suositus ajaa roskien poisto tässä välissä (Ora 2014). Roskien poisto etsii sitten joitain tietoja poistettaviksi. Vaikka sitä kutsuttaisiin pariin kertaan, se ei vielä takaa, että se olisi suoritettu tai että se olisi poistanut kaiken ylimääräisen. Koska muistin käyttöä mitataan vasta sen jälkeen, kun algoritmi on palauttanut lopputuloksen, Java on saattanut aloittaa automaattisen roskien keräämisen taustalla ilman käskyä. Muistinkäytön tulokset olivat kuitenkin sen verran lähellä toisiaan, että tuloksia voidaan hyvin pitää vertailukelpoisina toisiinsa nähden.

Hirschbergin lineaarinen menetelmä on lineaaritilaisuutensa ansiosta selvästi vähiten muistia käyttävä algoritmi. Rick oli myös selvällä erolla toiseksi vähiten muistia vaativa algoritmi. Kaksisuuntainen versio Rickistä oli kolmanneksi taloudellisin muistinkäytössä selvällä erolla molempiin suuntiin. Hirschbergin yksi- ja kaksisuuntainen versio sekä Wagnerin ja Fischerin menetelmä olivat suurimpia muistinkäyttäjiä. Niiden tulokset olivat suurin piirtein samaa suuruusluokkaa pienillä syötteillä. 40 000:n pituisilla syötteillä Wagnerin ja Fischerin menetelmä vei jo kuitenkin selvästi eniten muistia Hirschbergin yksi- ja kaksisuuntaisten tulosten ollessa hyvin lähellä toisiaan. Hirschbergin lineaarisen ja Rickin algoritmin suorittaminen onnistuu hiukan pienemmillä muistilla varustetuilla laitteillakin vielä aika isoillakin syötteillä, kun neljän muun algoritmin muistinkäyttö alkaa kasvaa kohtuuttoman suureksi jo selvästi pienemmillä syötteillä.

Merkistön koon pienentämisen vaikutus Wagnerin ja Fischerin algoritmiin sekä Hirschbergin lineaariseen menetelmään on kohtuullisen pieni, varsinkin kun muistetaan, että muistinkäytön mittaus ei anna välttämättä täysin tarkkoja lukemia. Hirschbergin yksi- ja kaksisuuntainen menetelmä käyttävät pienillä syötteillä jonkin verran enemmän muistia pienemmällä merkistöllä. 20 000 pituisten ja tätä pidempien syötteiden kanssa tulokset heittelevät hiukan molempiin suuntiin syötteiden pituudesta ja merkistön koosta huolimatta, joten tuloksista ei voi sanoa merkistön koon merkittävästi vaikuttavan käytettävän muistin määrään. Rickin algoritmissa merkistön koon pienentämisen seurauksena muistin käyttö noin kaksinkertaistuu,

mutta se on silti vielä kohtuullisella tasolla. Rickin kaksisuuntaisen version muistinkäyttö on pienemmällä merkistöllä 60 – 100 % suurempaa kuin suuremmalla merkistöllä.

Saatujen tuloksien nopeusjärjestykset vastasivat aikasemmin saatuja tuloksia niiden algoritmien osalta, joista on tehty vertailuja. Esimerkiksi Turun yliopistossa tehdyssä tutkimuksessa (Ber 2000) näistä algoritmeista oli kolme: Rick, Wagner–Fischer ja Hirschberg. Näiden suoritusajat olivat samassa järjestyksessä, vaikka testit ja syöteaineistot poikkesivatkin toisistaan.

Hirschbergin lineaarinen menetelmä on tutkielman ainoa algoritmi, jonka toiminta perustuu rekursiivisiin kutsuihin. Parissa muussakin tutkielman algoritmissa saatetaan käyttää muutamia rekursiivisia kutsuja, kuten esimerkiksi Rickin kaksisuuntaisessa versiossa sekä Wagnerin ja Fischerin algoritmissa, mutta niissä täsmäyksien etsiminen tapahtuu kuitenkin iteratiivisesti eli erilaisten silmukoiden avulla. Alkupään ja loppupään tutkimiset voitaisiin tehdä Hirschbergin lineaarisessa menetelmässä yhtä aikaa, sillä nämä suoritukset eivät vaikuta mitenkään toisiinsa. Tällaisia rekursiivisia kutsuja tulee todella paljon. Esimerkiksi toisella rekursiotasolla olisi mahdollista suorittaa jo neljää toisistaan riippumatonta tarkastelua samanaikaisesti. Nykyään tietokoneissa on useita suoritinytimiä, ja tehokkaimmissa näytönohjaimissa pieniä fysiikkalaskuja suorittavia ytimiä olisi tällä hetkellä jo useita tuhansia: esim. GeForce GTX TITAN Z -näytönohjaimessa ytimiä on 5760 kappaletta (NVI 2015). Algoritmin suorittamisen jakaminen usealle prosessoriytimelle tekee koodista jo jonkin verran monimutkaisempaa, ja näytönohjaimelle ohjelmointi vaatisi vielä paljon enemmän muutoksia niiden rajapintojen ym. vuoksi. Näiden avulla suoritusta voitaisiin saada tehostettua huomattavasti, mutta asian tutkiminen ei kuulu tämän tutkielman aihepiiriin.

4. Yhteenveto

Tässä tutkielmassa esiteltiin ensin pisimmän yhteisen alijonon määräämisen perusteet sekä kuusi eri algoritmia ongelman ratkaisemiseksi. Luvuss 2 tarkasteltiin, miten eri algoritmit ratkaisevat saman ongelman. Tutkielman tarkoituksena oli selvittää, voiko yksisuuntaisesta menetelmästä tehdä kahteen suuntaan toimivan version ja miten kaksisuuntaisella lähestymistavalla toimiva versio pärjää tehokkuudessa yksisuuntaiseen verrattuna.

Kolmannen luvun testeissä nopein oli Rickin algoritmin alkuperäinen versio. Hirschbergin kaksi- ja yksisuuntaiset menetelmät olivat selvästi hitaimmat tässä järjestyksessä. Pienemmällä merkistön koolla Rickin kaksisuuntainen oli myös hyvin hidas, ja Wagnerin ja Fischerin menetelmän ns. raa'an voiman käyttö alkoi myös näkyä tuloksissa hitautena. Kaksisuuntaisista nopein oli Hirschbergin lineaarinen menetelmä, joka oli suurilla syötteillä toiseksi nopein. Se oli myös ainoa, joka pystyi samaan tai jopa parempaan suoritukseen kuin verrokkinaan ollut yksisuuntainen versio. Hirschbergin ja Rickin algoritmeista tätä työtä varten tehdyt kaksisuuntaiset versiot eivät pärjänneet suoritusajoissa eivätkä muistinkäytössä alkuperäisille yksisuuntaisille versioille. Ne jäivät myös hyvin kauas Hirschbergin lineaarisen menetelmän tuloksista. Hirschbergin lineaarisen menetelmän vahvuutena oli se, että se ei pidä paljon mitään muistissa. Tämän vuoksi se joutuu etsimään samoja täsmäyksiä mahdollisesti useaan kertaan, mutta se säästää mm. taulukoiden ylläpidon prosessoinnissa niin paljon, että tämä kannattaa.

Hirschbergin lineaarista menetelmää lukuun ottamatta kaikki algoritmit tallensivat ainakin minimaaliset täsmäykset erilaisiin suuriin tulostaulukoihin. Wagner–Fischer tallensi kaiken lasketun tiedon tulostaulukoihin. Hirschbergin ja Rickin yksi- ja kaksisuuntaiset versiot käyttivät erilaisia suorasaantitaulukoita ym. syötteiden tietojen tallentamiseen. Nimensä mukaisesti lineaarisessa tilassa toimiva Hirschbergin menetelmä taas ylläpiti muistissa vain kahta taulukon riviä ja muodostuvaa lopputulosta, joten se vei odotetusti hyvin vähän muistia verrattuna muihin testien algoritmeihin. Rickin ja Hirschbergin kaksisuuntaisten versioiden ongelmana oli, että ne joutuivat jatkuvasti seuraamaan toiseen suuntaan tapahtuvaa etenemistä. Ne tekivät siis saman kuin yksisuuntaiset versiotkin, minkä lisäksi ne tekivät vielä paljon ylimääräistä tarkastelua.

Kaksisuuntaisesta lähestymistavasta oli hyötyä vain Hirschbergin lineaarisen menetelmän tapauksessa, jossa suurin etu oli vähäinen muistinkäyttö. Suoritusaikakin oli hyvä verrattuna moneen muuhun algoritmiin, vaikka se ei pärjännytkään Rickin algoritmille. Muissa algoritmeissa kaksisuuntaisuudesta oli enemmän haittaa kuin hyötyä. Koodista tuli monimutkaisempaa ja suorituskyky kärsi. Jos Rickin kaksisuuntainen versio ei pitäisi löydettyjä täsmäyksiä muistissa, saattaisi se olla tehokkaampi lopussa tapahtuvan ratkaisuksi kelpaavan PYAn esiintymän etsinnän lisääntyneestä prosessoinnista huolimatta.

Lopputuloksena voisi aavistaa, että kaksisuuntainen prosessointi sinänsä ilman lineaaritilaisuutta ei tuntuisi tehostavan pisimmän yhteisen alijonon ratkaisemista.

Lähteet

[Ber 2000]: Bergroth, L., Hakonen, H. & Raita, T. 2000, ”A Survey of Longest Common Subsequence Algorithms”, Proceedings of the Seventh International Symposium on String Processing and Information Retrieval, 2000, s. 39–48.

[Cor 2001]: Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. 2001, “Introduction to Algorithms” Second Edition. Cambridge (Mass.): MIT Press, s. 350–355.

[G&C 1999]: Goeman, H. & Clausen M. 1999, “A New Practical Linear Space Algorithm for the Longest Common Subsequence Problem”, Proceedings The Prague Stringology Club Workshop’99, Report DC–99–05, Department of Computer Science and Engineering, Chech Tehchnical University, 1999, s. 40–60.

[Hir 1975]: Hirschberg, Daniel S. 1975, “A Linear Space Algorithm for Computing Maximal Common Subsequences”, Journal of the Association for Computing Machinery, Vol. 18, No. 6, Kesäkuu 1975, s. 341–343.

[Hir 1977]: Hirschberg, Daniel S. 1977, “Algorithms for the Longest Common Subsequence Problem”, Journal of the Association for Computing Machinery, Vol. 24, No. 4, lokakuu 1977, s. 664–669.

[H&S 1977]: Hunt, J. W. & Szymanski, T. G. 1977, “A Fast Algorithm for Computing Longest Common Subsequences”, Communication of the Association for Computing Machinery, Vol. 20, No. 5, Toukokuu 1977, s. 350–353.

[NVI 2015]: NVIDIA 2015, GeForce GTX TITAN Z, specifications. Viittauspäivämäärä 23.5.2015.
http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-z/specifications

[Ora 2014]: Oracle 2014, System.gc, Java Platform SE 7 API. Viittauspäivämäärä 25.3.2015.
http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#gc%28%29

[Ric 1994]: Rick, Claus 1994, “New Algorithms for the Longest Common Subsequence Problem”, Research report no. 85123–CS, Department of Computer Science, University of Bonn, Germany, 1994.

[Ric 2000]: Rick, Claus 2000, “Simple and Fast Linear Space Computation of Longest Common Subsequences”, Information Processing Letters, Vol. 75, Issue 6, marraskuu 2000, s. 275–281.

[W&F 1974]: Wagner, R. A. & Fischer, M. J. 1974, “The String-to-String Correction Problem”, Journal of the Association for Computing Machinery, Vol21, No.1, tammikuu 1974, s. 168–173.

Liitteet:

Alkuperäiset tulokset: 500 merkkiä, a – z, 0 – 9												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
11 ms	3 MB	24 ms	1 MB	45 ms	1 MB	46 ms	1 MB	5 ms	1 MB	11 ms	1 MB	135
2 ms	2 MB	9 ms	1 MB	6 ms	1 MB	7 ms	1 MB	4 ms	1 MB	8 ms	1 MB	140
2 ms	2 MB	4 ms	1 MB	63 ms	1 MB	81 ms	1 MB	4 ms	1 MB	7 ms	1 MB	132
1 ms	2 MB	2 ms	1 MB	3 ms	1 MB	46 ms	1 MB	4 ms	1 MB	7 ms	1 MB	140
1 ms	2 MB	2 ms	1 MB	3 ms	1 MB	6 ms	1 MB	3 ms	1 MB	7 ms	1 MB	136
2 ms	2 MB	3 ms	1 MB	3 ms	1 MB	5 ms	1 MB	0 ms	1 MB	4 ms	1 MB	137
2 ms	2 MB	1 ms	1 MB	4 ms	1 MB	5 ms	1 MB	0 ms	1 MB	3 ms	1 MB	138
1 ms	2 MB	1 ms	1 MB	3 ms	1 MB	5 ms	1 MB	0 ms	1 MB	3 ms	1 MB	136
1 ms	2 MB	1 ms	1 MB	4 ms	1 MB	6 ms	1 MB	0 ms	1 MB	3 ms	1 MB	130
1 ms	2 MB	2 ms	1 MB	3 ms	1 MB	5 ms	1 MB	0 ms	1 MB	3 ms	1 MB	135
2 ms	2 MB	5 ms	1 MB	14 ms	1 MB	21 ms	1 MB	2 ms	1 MB	6 ms	1 MB	136
Alkuperäiset tulokset: 1 000 merkkiä, a – z, 0 – 9												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
16 ms	7 MB	29 ms	1 MB	70 ms	5 MB	187 ms	5 MB	13 ms	1 MB	29 ms	2 MB	274
6 ms	6 MB	6 ms	1 MB	15 ms	5 MB	19 ms	5 MB	2 ms	1 MB	12 ms	2 MB	281
5 ms	6 MB	6 ms	1 MB	193 ms	5 MB	185 ms	5 MB	1 ms	1 MB	12 ms	2 MB	273
5 ms	6 MB	6 ms	1 MB	14 ms	5 MB	20 ms	5 MB	1 ms	1 MB	10 ms	2 MB	274
5 ms	6 MB	5 ms	1 MB	14 ms	5 MB	21 ms	5 MB	1 ms	1 MB	10 ms	2 MB	281
5 ms	6 MB	5 ms	1 MB	13 ms	5 MB	20 ms	5 MB	1 ms	1 MB	11 ms	2 MB	276
5 ms	6 MB	5 ms	1 MB	14 ms	5 MB	21 ms	5 MB	1 ms	1 MB	11 ms	2 MB	285
5 ms	6 MB	6 ms	1 MB	13 ms	5 MB	21 ms	5 MB	1 ms	1 MB	10 ms	2 MB	278
4 ms	6 MB	5 ms	1 MB	13 ms	5 MB	21 ms	5 MB	1 ms	1 MB	11 ms	2 MB	279
5 ms	6 MB	5 ms	1 MB	13 ms	5 MB	21 ms	5 MB	1 ms	1 MB	11 ms	2 MB	281
6 ms	6 MB	8 ms	1 MB	37 ms	5 MB	54 ms	5 MB	2 ms	1 MB	13 ms	2 MB	278

Alkuperäiset tulokset: 1 500 merkkiä, a – z, 0 – 9												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
20 ms	15 MB	37 ms	1 MB	109 ms	9 MB	186 ms	9 MB	28 ms	2 MB	54 ms	3 MB	422
13 ms	14 MB	12 ms	1 MB	29 ms	9 MB	37 ms	9 MB	2 ms	2 MB	22 ms	3 MB	421
11 ms	14 MB	11 ms	1 MB	30 ms	9 MB	402 ms	9 MB	2 ms	2 MB	23 ms	3 MB	415
11 ms	14 MB	11 ms	1 MB	30 ms	9 MB	47 ms	9 MB	2 ms	2 MB	21 ms	3 MB	425
11 ms	14 MB	12 ms	1 MB	30 ms	9 MB	47 ms	9 MB	2 ms	2 MB	23 ms	3 MB	419
11 ms	14 MB	11 ms	1 MB	29 ms	9 MB	47 ms	9 MB	2 ms	2 MB	22 ms	3 MB	416
11 ms	14 MB	11 ms	1 MB	29 ms	9 MB	46 ms	9 MB	2 ms	2 MB	23 ms	3 MB	414
10 ms	14 MB	12 ms	1 MB	30 ms	9 MB	47 ms	9 MB	2 ms	2 MB	21 ms	3 MB	422
10 ms	14 MB	12 ms	1 MB	30 ms	9 MB	47 ms	9 MB	2 ms	2 MB	21 ms	3 MB	416
10 ms	14 MB	11 ms	1 MB	29 ms	9 MB	46 ms	9 MB	2 ms	2 MB	22 ms	3 MB	413
12 ms	14 MB	14 ms	1 MB	38 ms	9 MB	95 ms	9 MB	5 ms	2 MB	25 ms	3 MB	418
Alkuperäiset tulokset: 2 000 merkkiä, a – z, 0 – 9												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
31 ms	25 MB	48 ms	2 MB	125 ms	16 MB	208 ms	18 MB	34 ms	4 MB	55 ms	6 MB	549
23 ms	24 MB	20 ms	2 MB	50 ms	16 MB	602 ms	16 MB	4 ms	3 MB	42 ms	6 MB	560
18 ms	24 MB	21 ms	2 MB	54 ms	17 MB	83 ms	17 MB	3 ms	3 MB	9 ms	6 MB	564
19 ms	24 MB	21 ms	2 MB	53 ms	17 MB	81 ms	17 MB	3 ms	3 MB	39 ms	6 MB	554
18 ms	24 MB	20 ms	2 MB	53 ms	17 MB	81 ms	17 MB	3 ms	3 MB	41 ms	6 MB	555
18 ms	24 MB	20 ms	2 MB	53 ms	17 MB	81 ms	17 MB	3 ms	3 MB	38 ms	6 MB	561
18 ms	24 MB	21 ms	2 MB	53 ms	17 MB	81 ms	17 MB	3 ms	3 MB	40 ms	5 MB	548
18 ms	24 MB	21 ms	2 MB	53 ms	17 MB	82 ms	17 MB	4 ms	3 MB	40 ms	6 MB	563
19 ms	24 MB	21 ms	2 MB	54 ms	17 MB	83 ms	17 MB	3 ms	3 MB	8 ms	6 MB	564
18 ms	24 MB	20 ms	2 MB	54 ms	17 MB	82 ms	17 MB	4 ms	3 MB	7 ms	6 MB	571
20 ms	24 MB	23 ms	2 MB	60 ms	17 MB	146 ms	17 MB	6 ms	3 MB	32 ms	6 MB	559

Alkuperäiset tulokset: 2 500 merkkiä, a – z, 0 – 9												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
41 ms	39 MB	57 ms	2 MB	134 ms	26 MB	251 ms	26 MB	49 ms	5 MB	85 ms	8 MB	702
35 ms	38 MB	31 ms	2 MB	221 ms	25 MB	97 ms	26 MB	7 ms	5 MB	65 ms	7 MB	697
30 ms	37 MB	31 ms	2 MB	114 ms	26 MB	104 ms	25 MB	6 ms	5 MB	61 ms	7 MB	706
29 ms	37 MB	31 ms	2 MB	95 ms	25 MB	102 ms	25 MB	5 ms	5 MB	13 ms	7 MB	697
29 ms	37 MB	31 ms	2 MB	94 ms	25 MB	102 ms	25 MB	5 ms	5 MB	59 ms	7 MB	692
29 ms	37 MB	31 ms	2 MB	94 ms	25 MB	102 ms	25 MB	5 ms	5 MB	58 ms	7 MB	702
28 ms	37 MB	31 ms	2 MB	94 ms	25 MB	102 ms	25 MB	5 ms	5 MB	62 ms	7 MB	693
29 ms	37 MB	31 ms	2 MB	95 ms	25 MB	102 ms	25 MB	5 ms	5 MB	63 ms	7 MB	702
28 ms	37 MB	31 ms	2 MB	96 ms	25 MB	102 ms	25 MB	5 ms	5 MB	12 ms	7 MB	704
29 ms	37 MB	31 ms	2 MB	95 ms	25 MB	101 ms	25 MB	5 ms	5 MB	10 ms	7 MB	698
31 ms	37 MB	34 ms	2 MB	113 ms	25 MB	117 ms	25 MB	10 ms	5 MB	49 ms	7 MB	699
Alkuperäiset tulokset: 5 000 merkkiä, a – z, 0 – 9												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
198 ms	149 MB	155 ms	6 MB	362 ms	111 MB	574 ms	111 MB	72 ms	17 MB	267 ms	26 MB	1404
140 ms	151 MB	127 ms	4 MB	311 ms	112 MB	383 ms	112 MB	139 ms	16 MB	65 ms	26 MB	1409
117 ms	151 MB	127 ms	5 MB	335 ms	113 MB	553 ms	108 MB	20 ms	16 MB	239 ms	27 MB	1404
117 ms	151 MB	126 ms	5 MB	336 ms	113 MB	550 ms	108 MB	19 ms	16 MB	55 ms	27 MB	1407
118 ms	151 MB	128 ms	5 MB	333 ms	113 MB	549 ms	108 MB	19 ms	16 MB	53 ms	27 MB	1407
117 ms	151 MB	127 ms	5 MB	334 ms	113 MB	552 ms	108 MB	19 ms	16 MB	51 ms	27 MB	1404
117 ms	151 MB	127 ms	5 MB	333 ms	113 MB	548 ms	108 MB	19 ms	16 MB	51 ms	27 MB	1399
117 ms	151 MB	126 ms	5 MB	334 ms	113 MB	549 ms	108 MB	19 ms	16 MB	49 ms	27 MB	1404
116 ms	151 MB	127 ms	5 MB	335 ms	113 MB	550 ms	108 MB	19 ms	16 MB	50 ms	27 MB	1409
117 ms	151 MB	127 ms	5 MB	332 ms	113 MB	548 ms	108 MB	19 ms	16 MB	50 ms	27 MB	1401
127 ms	151 MB	130 ms	5 MB	335 ms	113 MB	536 ms	109 MB	36 ms	16 MB	93 ms	27 MB	1405

Alkuperäiset tulokset: 10 000 merkkiä, a – z, 0 – 9												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
741 ms	594 MB	566 ms	12 MB	1293 ms	606 MB	1940 ms	394 MB	127 ms	60 MB	837 ms	105 MB	2837
538 ms	591 MB	504 ms	18 MB	1647 ms	607 MB	1520 ms	624 MB	304 ms	76 MB	1029 ms	107 MB	2825
455 ms	606 MB	504 ms	18 MB	1364 ms	605 MB	2198 ms	605 MB	74 ms	71 MB	362 ms	107 MB	2834
455 ms	606 MB	505 ms	18 MB	1361 ms	605 MB	2187 ms	605 MB	76 ms	71 MB	290 ms	107 MB	2823
453 ms	605 MB	505 ms	18 MB	1359 ms	605 MB	2190 ms	605 MB	77 ms	71 MB	290 ms	107 MB	2833
452 ms	605 MB	505 ms	18 MB	1362 ms	605 MB	2188 ms	605 MB	76 ms	71 MB	291 ms	107 MB	2822
452 ms	605 MB	504 ms	18 MB	1358 ms	605 MB	2184 ms	605 MB	76 ms	71 MB	288 ms	107 MB	2815
451 ms	605 MB	505 ms	18 MB	1361 ms	605 MB	2190 ms	605 MB	76 ms	71 MB	288 ms	107 MB	2832
451 ms	605 MB	504 ms	18 MB	1357 ms	605 MB	2186 ms	605 MB	75 ms	71 MB	289 ms	107 MB	2826
451 ms	605 MB	504 ms	18 MB	1361 ms	605 MB	2189 ms	605 MB	75 ms	71 MB	290 ms	107 MB	2819
490 ms	603 MB	511 ms	17 MB	1382 ms	605 MB	2097 ms	586 MB	104 ms	70 MB	425 ms	107 MB	2827
Alkuperäiset tulokset: 20 000 merkkiä, a – z, 0 – 9												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
2642 ms	2363 MB	2841 ms	30 MB	5377 ms	2610 MB	7339 ms	2470 MB	439 ms	241 MB	2925 ms	385 MB	5659
2376 ms	2397 MB	2018 ms	58 MB	8033 ms	2629 MB	6855 ms	2862 MB	414 ms	233 MB	2376 ms	409 MB	5669
1799 ms	2419 MB	2020 ms	69 MB	5316 ms	2619 MB	8786 ms	2821 MB	294 ms	275 MB	2251 ms	412 MB	5648
1804 ms	2407 MB	2016 ms	69 MB	5331 ms	2610 MB	8768 ms	2815 MB	292 ms	274 MB	2280 ms	412 MB	5673
1802 ms	2403 MB	2017 ms	69 MB	5321 ms	2608 MB	8751 ms	2813 MB	291 ms	274 MB	2251 ms	411 MB	5666
1801 ms	2402 MB	2016 ms	69 MB	5330 ms	2607 MB	8800 ms	2813 MB	291 ms	274 MB	2221 ms	411 MB	5667
1801 ms	2402 MB	2018 ms	69 MB	5311 ms	2607 MB	8720 ms	2813 MB	292 ms	274 MB	2232 ms	411 MB	5625
1801 ms	2402 MB	2016 ms	69 MB	5323 ms	2607 MB	8763 ms	2813 MB	289 ms	274 MB	2204 ms	411 MB	5678
1800 ms	2402 MB	2014 ms	69 MB	5324 ms	2607 MB	8744 ms	2813 MB	295 ms	274 MB	2260 ms	411 MB	5651
1805 ms	2402 MB	2016 ms	69 MB	5319 ms	2607 MB	8750 ms	2813 MB	291 ms	274 MB	2236 ms	411 MB	5661
1943 ms	2400 MB	2099 ms	64 MB	5599 ms	2611 MB	8428 ms	2785 MB	319 ms	267 MB	2324 ms	408 MB	5660

Alkuperäiset tulokset: 30 000 merkkiä, a – z, 0 – 9												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
5874 ms	5319 MB	6328 ms	31 MB	11660 ms	4940 MB	16444 ms	4575 MB	894 ms	514 MB	8477 ms	794 MB	8479
5933 ms	5348 MB	4551 ms	63 MB	18327 ms	5987 MB	13892 ms	4155 MB	972 ms	567 MB	7564 ms	827 MB	8494
4441 ms	5325 MB	4550 ms	64 MB	13163 ms	6007 MB	20934 ms	3845 MB	783 ms	536 MB	7400 ms	834 MB	8484
4484 ms	5338 MB	4548 ms	69 MB	13226 ms	5869 MB	20766 ms	6545 MB	788 ms	528 MB	7380 ms	827 MB	8514
4955 ms	5361 MB	4546 ms	66 MB	13170 ms	6026 MB	20756 ms	6551 MB	786 ms	558 MB	7385 ms	835 MB	8493
4493 ms	5330 MB	4551 ms	69 MB	13201 ms	5811 MB	20764 ms	6532 MB	793 ms	533 MB	7406 ms	833 MB	8500
4532 ms	5314 MB	4545 ms	72 MB	13225 ms	5714 MB	20940 ms	6282 MB	785 ms	559 MB	7330 ms	789 MB	8478
4537 ms	5315 MB	4553 ms	76 MB	13248 ms	5539 MB	20817 ms	6183 MB	787 ms	517 MB	7429 ms	836 MB	8496
4569 ms	5366 MB	4545 ms	78 MB	13254 ms	5447 MB	20737 ms	6033 MB	786 ms	535 MB	7359 ms	855 MB	8500
5011 ms	5349 MB	4547 ms	75 MB	13215 ms	5639 MB	21059 ms	6205 MB	781 ms	575 MB	7370 ms	808 MB	8512
4883 ms	5337 MB	4726 ms	66 MB	13569 ms	5698 MB	19711 ms	5691 MB	816 ms	542 MB	7510 ms	824 MB	8495
Alkuperäiset tulokset: 40 000 merkkiä, a – z, 0 – 9												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
11419 ms	9420 MB	10769 ms	36 MB	21057 ms	6641 MB	28455 ms	7030 MB	1534 ms	924 MB	17417 ms	1434 MB	11339
11448 ms	9418 MB	8167 ms	64 MB	33505 ms	7291 MB	28345 ms	6745 MB	1804 ms	928 MB	16910 ms	1437 MB	11333
9265 ms	9428 MB	8165 ms	64 MB	25573 ms	7291 MB	32207 ms	6778 MB	1485 ms	928 MB	16674 ms	1438 MB	11383
9268 ms	9426 MB	8168 ms	64 MB	25572 ms	7292 MB	32056 ms	6746 MB	1484 ms	928 MB	16554 ms	1438 MB	11333
9273 ms	9428 MB	8170 ms	64 MB	25527 ms	7291 MB	32184 ms	6745 MB	1477 ms	928 MB	16485 ms	1438 MB	11355
9294 ms	9427 MB	8174 ms	64 MB	25566 ms	7291 MB	32026 ms	6746 MB	1495 ms	928 MB	16624 ms	1437 MB	11333
9304 ms	9427 MB	8185 ms	64 MB	25525 ms	7291 MB	32049 ms	6746 MB	1476 ms	928 MB	16608 ms	1438 MB	11332
9272 ms	9428 MB	8172 ms	64 MB	25643 ms	7291 MB	32016 ms	6745 MB	1482 ms	928 MB	16718 ms	1438 MB	11341
9276 ms	9426 MB	8162 ms	64 MB	25488 ms	7292 MB	32212 ms	6746 MB	1486 ms	928 MB	16709 ms	1438 MB	11329
9329 ms	9427 MB	8164 ms	64 MB	25471 ms	7291 MB	32048 ms	6746 MB	1483 ms	928 MB	16598 ms	1438 MB	11352
9715 ms	9426 MB	8430 ms	61 MB	25893 ms	7226 MB	31360 ms	6777 MB	1521 ms	928 MB	16730 ms	1437 MB	11343

Alkuperäiset tulokset: 500 merkkiä, a – n												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
12 ms	3 MB	24 ms	1 MB	59 ms	1 MB	69 ms	1 MB	7 ms	1 MB	15 ms	1 MB	200
2 ms	2 MB	5 ms	1 MB	10 ms	1 MB	10 ms	1 MB	1 ms	1 MB	8 ms	1 MB	198
1 ms	2 MB	3 ms	1 MB	93 ms	1 MB	70 ms	2 MB	1 ms	1 MB	6 ms	1 MB	208
2 ms	2 MB	2 ms	1 MB	5 ms	1 MB	69 ms	2 MB	0 ms	1 MB	6 ms	1 MB	208
1 ms	2 MB	2 ms	1 MB	5 ms	1 MB	8 ms	1 MB	0 ms	1 MB	5 ms	1 MB	199
1 ms	2 MB	1 ms	1 MB	5 ms	1 MB	7 ms	1 MB	1 ms	1 MB	5 ms	1 MB	198
2 ms	2 MB	1 ms	1 MB	5 ms	1 MB	7 ms	1 MB	0 ms	1 MB	5 ms	1 MB	209
1 ms	2 MB	1 ms	1 MB	5 ms	1 MB	7 ms	1 MB	0 ms	1 MB	4 ms	1 MB	202
1 ms	2 MB	1 ms	1 MB	6 ms	1 MB	8 ms	1 MB	1 ms	1 MB	5 ms	1 MB	206
2 ms	2 MB	1 ms	1 MB	6 ms	1 MB	7 ms	1 MB	0 ms	1 MB	5 ms	1 MB	203
3 ms	2 MB	4 ms	1 MB	20 ms	1 MB	26 ms	1 MB	1 ms	1 MB	6 ms	1 MB	203
Alkuperäiset tulokset: 1 000 merkkiä, a – n												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
16 ms	8 MB	29 ms	1 MB	75 ms	5 MB	180 ms	5 MB	15 ms	2 MB	38 ms	3 MB	415
7 ms	7 MB	5 ms	1 MB	187 ms	5 MB	257 ms	5 MB	2 ms	2 MB	18 ms	3 MB	411
6 ms	7 MB	5 ms	1 MB	20 ms	5 MB	25 ms	5 MB	11 ms	2 MB	17 ms	2 MB	407
6 ms	7 MB	4 ms	1 MB	20 ms	5 MB	24 ms	5 MB	2 ms	2 MB	17 ms	3 MB	409
6 ms	7 MB	5 ms	1 MB	20 ms	5 MB	25 ms	5 MB	2 ms	2 MB	16 ms	2 MB	408
6 ms	7 MB	5 ms	1 MB	19 ms	5 MB	24 ms	5 MB	2 ms	2 MB	4 ms	2 MB	411
6 ms	7 MB	4 ms	1 MB	20 ms	5 MB	24 ms	5 MB	2 ms	2 MB	16 ms	3 MB	419
6 ms	7 MB	4 ms	1 MB	19 ms	5 MB	23 ms	5 MB	2 ms	2 MB	16 ms	2 MB	411
6 ms	7 MB	5 ms	1 MB	19 ms	5 MB	24 ms	5 MB	2 ms	2 MB	17 ms	3 MB	411
5 ms	7 MB	4 ms	1 MB	20 ms	5 MB	24 ms	5 MB	2 ms	2 MB	17 ms	3 MB	414
7 ms	7 MB	7 ms	1 MB	42 ms	5 MB	63 ms	5 MB	4 ms	2 MB	18 ms	3 MB	412

Alkuperäiset tulokset: 1 500 merkkiä, a – n												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
22 ms	15 MB	38 ms	2 MB	96 ms	9 MB	200 ms	11 MB	31 ms	4 MB	52 ms	6 MB	626
15 ms	14 MB	10 ms	1 MB	173 ms	9 MB	51 ms	11 MB	4 ms	3 MB	36 ms	6 MB	612
13 ms	14 MB	10 ms	1 MB	45 ms	9 MB	599 ms	11 MB	4 ms	3 MB	9 ms	6 MB	619
12 ms	14 MB	10 ms	1 MB	43 ms	9 MB	67 ms	9 MB	3 ms	3 MB	35 ms	6 MB	613
12 ms	14 MB	10 ms	1 MB	43 ms	9 MB	66 ms	9 MB	3 ms	3 MB	35 ms	6 MB	614
12 ms	14 MB	10 ms	1 MB	43 ms	9 MB	66 ms	9 MB	3 ms	3 MB	34 ms	6 MB	614
12 ms	14 MB	10 ms	1 MB	43 ms	9 MB	66 ms	9 MB	3 ms	3 MB	34 ms	6 MB	616
11 ms	14 MB	10 ms	1 MB	43 ms	9 MB	67 ms	9 MB	3 ms	3 MB	34 ms	6 MB	606
12 ms	14 MB	10 ms	1 MB	43 ms	9 MB	67 ms	9 MB	3 ms	3 MB	35 ms	6 MB	626
12 ms	14 MB	10 ms	1 MB	43 ms	9 MB	66 ms	9 MB	3 ms	3 MB	7 ms	6 MB	622
13 ms	14 MB	13 ms	1 MB	62 ms	9 MB	132 ms	10 MB	6 ms	3 MB	31 ms	6 MB	617
Alkuperäiset tulokset: 2 000 merkkiä, a – n												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
34 ms	25 MB	50 ms	2 MB	146 ms	20 MB	241 ms	20 MB	47 ms	6 MB	77 ms	8 MB	827
25 ms	25 MB	18 ms	2 MB	74 ms	20 MB	87 ms	20 MB	7 ms	6 MB	64 ms	8 MB	826
22 ms	25 MB	18 ms	2 MB	79 ms	20 MB	120 ms	19 MB	5 ms	6 MB	64 ms	8 MB	826
22 ms	25 MB	18 ms	2 MB	78 ms	20 MB	118 ms	19 MB	5 ms	6 MB	13 ms	8 MB	822
21 ms	25 MB	18 ms	2 MB	78 ms	20 MB	118 ms	19 MB	5 ms	6 MB	61 ms	8 MB	826
22 ms	25 MB	18 ms	2 MB	78 ms	20 MB	118 ms	19 MB	5 ms	6 MB	62 ms	8 MB	827
20 ms	25 MB	17 ms	2 MB	77 ms	20 MB	117 ms	19 MB	5 ms	6 MB	61 ms	8 MB	822
21 ms	25 MB	17 ms	2 MB	78 ms	20 MB	119 ms	19 MB	5 ms	6 MB	60 ms	8 MB	833
21 ms	25 MB	17 ms	2 MB	78 ms	20 MB	118 ms	19 MB	5 ms	6 MB	12 ms	8 MB	823
21 ms	25 MB	18 ms	2 MB	78 ms	20 MB	119 ms	19 MB	5 ms	6 MB	12 ms	8 MB	832
23 ms	25 MB	21 ms	2 MB	84 ms	20 MB	128 ms	19 MB	9 ms	6 MB	49 ms	8 MB	826

Alkuperäiset tulokset: 2 500 merkkiä, a – n												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
45 ms	40 MB	58 ms	2 MB	182 ms	37 MB	295 ms	37 MB	69 ms	8 MB	108 ms	13 MB	1034
39 ms	38 MB	28 ms	2 MB	126 ms	37 MB	733 ms	35 MB	10 ms	8 MB	32 ms	13 MB	1040
33 ms	38 MB	27 ms	2 MB	124 ms	38 MB	146 ms	36 MB	8 ms	9 MB	94 ms	14 MB	1036
33 ms	38 MB	27 ms	2 MB	123 ms	38 MB	147 ms	36 MB	9 ms	9 MB	92 ms	14 MB	1024
33 ms	38 MB	27 ms	2 MB	123 ms	38 MB	145 ms	36 MB	8 ms	9 MB	93 ms	14 MB	1031
32 ms	38 MB	28 ms	2 MB	123 ms	38 MB	145 ms	36 MB	8 ms	9 MB	92 ms	14 MB	1028
34 ms	38 MB	27 ms	2 MB	124 ms	38 MB	145 ms	36 MB	9 ms	9 MB	93 ms	14 MB	1030
34 ms	38 MB	28 ms	2 MB	127 ms	38 MB	145 ms	36 MB	9 ms	9 MB	94 ms	14 MB	1014
34 ms	38 MB	28 ms	2 MB	123 ms	38 MB	145 ms	36 MB	9 ms	9 MB	97 ms	14 MB	1033
34 ms	38 MB	27 ms	2 MB	123 ms	38 MB	146 ms	36 MB	8 ms	9 MB	21 ms	14 MB	1039
35 ms	38 MB	31 ms	2 MB	130 ms	38 MB	219 ms	36 MB	15 ms	9 MB	82 ms	14 MB	1031
Alkuperäiset tulokset: 5 000 merkkiä, a – n												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
211 ms	153 MB	154 ms	6 MB	508 ms	118 MB	752 ms	201 MB	82 ms	31 MB	407 ms	47 MB	2064
154 ms	153 MB	115 ms	5 MB	446 ms	191 MB	538 ms	202 MB	234 ms	31 MB	102 ms	49 MB	2082
133 ms	152 MB	115 ms	5 MB	486 ms	190 MB	784 ms	196 MB	31 ms	32 MB	94 ms	49 MB	2085
130 ms	152 MB	115 ms	5 MB	483 ms	190 MB	780 ms	196 MB	31 ms	32 MB	93 ms	49 MB	2073
129 ms	152 MB	115 ms	5 MB	481 ms	190 MB	780 ms	196 MB	31 ms	32 MB	93 ms	49 MB	2065
130 ms	152 MB	115 ms	5 MB	482 ms	190 MB	781 ms	196 MB	31 ms	32 MB	93 ms	49 MB	2082
130 ms	152 MB	114 ms	5 MB	484 ms	190 MB	783 ms	196 MB	31 ms	32 MB	95 ms	49 MB	2078
130 ms	152 MB	114 ms	5 MB	481 ms	190 MB	779 ms	196 MB	30 ms	32 MB	91 ms	49 MB	2072
129 ms	152 MB	115 ms	5 MB	483 ms	190 MB	781 ms	196 MB	31 ms	32 MB	94 ms	49 MB	2073
131 ms	152 MB	115 ms	5 MB	484 ms	190 MB	781 ms	196 MB	30 ms	32 MB	94 ms	49 MB	2070
141 ms	152 MB	119 ms	5 MB	482 ms	183 MB	754 ms	197 MB	56 ms	32 MB	126 ms	49 MB	2074

Alkuperäiset tulokset: 10 000 merkkiä, a – n												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
802 ms	613 MB	563 ms	12 MB	1876 ms	587 MB	2697 ms	566 MB	194 ms	123 MB	1050 ms	175 MB	4171
599 ms	608 MB	465 ms	20 MB	2175 ms	809 MB	2456 ms	909 MB	274 ms	121 MB	1879 ms	182 MB	4167
519 ms	626 MB	468 ms	20 MB	2110 ms	807 MB	3028 ms	867 MB	131 ms	121 MB	595 ms	181 MB	4127
516 ms	626 MB	465 ms	20 MB	2103 ms	807 MB	3033 ms	887 MB	130 ms	121 MB	586 ms	181 MB	4153
516 ms	626 MB	465 ms	20 MB	2110 ms	806 MB	3047 ms	887 MB	131 ms	121 MB	590 ms	181 MB	4158
517 ms	626 MB	466 ms	20 MB	2103 ms	806 MB	3040 ms	887 MB	130 ms	121 MB	588 ms	181 MB	4167
517 ms	626 MB	465 ms	20 MB	2102 ms	806 MB	3033 ms	887 MB	130 ms	121 MB	589 ms	181 MB	4155
515 ms	626 MB	467 ms	20 MB	2110 ms	806 MB	3035 ms	887 MB	131 ms	121 MB	661 ms	181 MB	4147
518 ms	626 MB	470 ms	20 MB	2105 ms	806 MB	3042 ms	887 MB	130 ms	121 MB	576 ms	181 MB	4168
516 ms	626 MB	468 ms	20 MB	2098 ms	806 MB	3035 ms	887 MB	130 ms	121 MB	583 ms	181 MB	4143
554 ms	623 MB	476 ms	19 MB	2089 ms	785 MB	2945 ms	855 MB	151 ms	121 MB	770 ms	181 MB	4156
Alkuperäiset tulokset: 20 000 merkkiä, a – n												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
2881 ms	2436 MB	2976 ms	30 MB	7492 ms	1756 MB	9369 ms	1572 MB	618 ms	503 MB	5737 ms	731 MB	8328
2599 ms	2474 MB	1892 ms	50 MB	11612 ms	2302 MB	8540 ms	2627 MB	628 ms	477 MB	5327 ms	715 MB	8340
2049 ms	2481 MB	1892 ms	63 MB	7831 ms	1653 MB	12867 ms	1863 MB	522 ms	508 MB	5157 ms	726 MB	8307
2043 ms	2442 MB	1894 ms	76 MB	7536 ms	3278 MB	12708 ms	3658 MB	522 ms	533 MB	5135 ms	762 MB	8309
2044 ms	2439 MB	1893 ms	76 MB	7547 ms	3276 MB	12774 ms	3732 MB	520 ms	533 MB	5123 ms	761 MB	8353
2043 ms	2438 MB	1891 ms	76 MB	7529 ms	3275 MB	12702 ms	3656 MB	522 ms	533 MB	5126 ms	761 MB	8310
2043 ms	2438 MB	1894 ms	76 MB	7531 ms	3275 MB	12733 ms	3656 MB	522 ms	457 MB	5065 ms	761 MB	8300
2043 ms	2438 MB	1894 ms	76 MB	7547 ms	3275 MB	12743 ms	3732 MB	522 ms	533 MB	5177 ms	761 MB	8348
2044 ms	2438 MB	1892 ms	76 MB	7551 ms	3275 MB	12721 ms	3656 MB	521 ms	533 MB	5134 ms	761 MB	8324
2044 ms	2438 MB	1891 ms	76 MB	7528 ms	3275 MB	12714 ms	3656 MB	520 ms	533 MB	5132 ms	761 MB	8327
2183 ms	2446 MB	2001 ms	68 MB	7970 ms	2864 MB	11987 ms	3181 MB	542 ms	514 MB	5211 ms	750 MB	8325

Alkuperäiset tulokset: 30 000 merkkiä, a – n											
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
6584 ms	5489 MB	6633 ms	31 MB	16227 ms	4320 MB	22614 ms	4059 MB	1270 ms	1029 MB	17213 ms	1543 MB	12473
6284 ms	5512 MB	4272 ms	64 MB	25948 ms	4150 MB	18614 ms	5172 MB	1400 ms	1050 MB	16345 ms	1530 MB	12510
5056 ms	5483 MB	4267 ms	69 MB	19542 ms	3846 MB	26629 ms	4922 MB	1172 ms	1036 MB	16372 ms	1600 MB	12543
5084 ms	5501 MB	4272 ms	74 MB	19523 ms	7076 MB	26540 ms	4511 MB	1174 ms	1094 MB	16207 ms	1613 MB	12479
5109 ms	5519 MB	4277 ms	77 MB	19432 ms	6883 MB	26586 ms	4186 MB	1174 ms	1064 MB	16322 ms	1603 MB	12490
5134 ms	5474 MB	4280 ms	79 MB	19434 ms	6812 MB	26713 ms	3979 MB	1170 ms	1092 MB	16112 ms	1560 MB	12505
5592 ms	5460 MB	4263 ms	79 MB	19481 ms	6868 MB	26731 ms	4050 MB	1177 ms	1080 MB	16355 ms	1620 MB	12528
5188 ms	5484 MB	4275 ms	78 MB	19452 ms	6906 MB	26588 ms	4197 MB	1171 ms	1078 MB	16097 ms	1548 MB	12516
5126 ms	5501 MB	4270 ms	80 MB	19470 ms	6839 MB	26642 ms	3982 MB	1173 ms	1098 MB	16296 ms	1569 MB	12512
5192 ms	5468 MB	4266 ms	79 MB	19460 ms	6806 MB	26614 ms	4053 MB	1169 ms	1085 MB	16259 ms	1549 MB	12514
5435 ms	5489 MB	4508 ms	71 MB	19797 ms	6051 MB	25427 ms	4311 MB	1205 ms	1071 MB	16358 ms	1574 MB	12507
Alkuperäiset tulokset: 40 000 merkkiä, a – n												
Wagner–Fischer		Hirschberg lin.		Hirschberg		Hirschberg 2way		Rick		Rick 2way		PYA pituus
11975 ms	9755 MB	11164 ms	75 MB	28935 ms	6552 MB	39522 ms	7133 MB	2202 ms	256 MB	38016 ms	2400 MB	16697
12452 ms	7135 MB	7685 ms	64 MB	47049 ms	6522 MB	39698 ms	6746 MB	2512 ms	256 MB	38022 ms	2400 MB	16705
31549 ms	9707 MB	7687 ms	64 MB	80171 ms	6522 MB	51581 ms	6714 MB	2094 ms	256 MB	37223 ms	2400 MB	16689
67954 ms	9693 MB	7715 ms	64 MB	51134 ms	6489 MB	51371 ms	6712 MB	2095 ms	256 MB	37463 ms	2396 MB	16688
12262 ms	9694 MB	7715 ms	64 MB	35448 ms	6488 MB	51364 ms	6712 MB	2112 ms	223 MB	37211 ms	2396 MB	16636
11580 ms	9711 MB	7686 ms	64 MB	35484 ms	6491 MB	51540 ms	6715 MB	2096 ms	225 MB	37156 ms	2403 MB	16659
11616 ms	9693 MB	7676 ms	64 MB	35584 ms	6522 MB	51559 ms	6745 MB	2100 ms	256 MB	37370 ms	2399 MB	16694
11610 ms	9699 MB	7677 ms	64 MB	35654 ms	6521 MB	51507 ms	6744 MB	2105 ms	256 MB	37538 ms	2430 MB	16702
11609 ms	9715 MB	7675 ms	64 MB	35570 ms	6491 MB	51436 ms	6716 MB	2097 ms	225 MB	37120 ms	2404 MB	16655
11624 ms	9713 MB	7695 ms	64 MB	35572 ms	6522 MB	51345 ms	6714 MB	2097 ms	224 MB	37583 ms	2399 MB	16641
19423 ms	9452 MB	8038 ms	65 MB	42060 ms	6512 MB	49092 ms	6765 MB	2151 ms	243 MB	37470 ms	2403 MB	16677

Syötteiden järjestyksen muuttamisen alkuperäiset tulokset

Wagner–Fischer X ≤ Y		Wagner–Fischer X>Y		Hirschberg lineaarinen X ≤ Y		Hirschberg lineaarinen X>Y		PYA pituus
3195 ms	2960 MB	3562 ms	2935 MB	3071 ms	59 MB	2517 ms	59 MB	5893
2243 ms	2935 MB	2239 ms	3004 MB	2574 ms	77 MB	2508 ms	77 MB	5907
2242 ms	2972 MB	2228 ms	2959 MB	2574 ms	76 MB	2508 ms	76 MB	5915
2243 ms	2988 MB	2225 ms	2981 MB	2574 ms	75 MB	2511 ms	75 MB	5900
2243 ms	2961 MB	2226 ms	2948 MB	2573 ms	74 MB	2508 ms	74 MB	5909
2433 ms	2963 MB	2496 ms	2965 MB	2673 ms	72 MB	2510 ms	72 MB	5905
Hirschberg X ≤ Y		Hirschberg X>Y		Hirschberg 2-suuntainen X ≤ Y		Hirschberg 2-suuntainen X>Y		PYA pituus
7426 ms	730 MB	10401 ms	10025 MB	8941 ms	622 MB	13584 ms	10068 MB	5893
6696 ms	3542 MB	9405 ms	10010 MB	9930 ms	620 MB	13332 ms	10015 MB	5907
6694 ms	3484 MB	9268 ms	10006 MB	9962 ms	616 MB	13309 ms	10054 MB	5915
6691 ms	3501 MB	9202 ms	10034 MB	9906 ms	692 MB	13367 ms	10024 MB	5900
6687 ms	3532 MB	9164 ms	10061 MB	9953 ms	686 MB	13413 ms	10031 MB	5909
6839 ms	2958 MB	9488 ms	10027 MB	9738 ms	647 MB	13401 ms	10038 MB	5905

Wagnerin ja Fischerin algoritmin koodi:

```
import java.util.*;
import java.io.*;
public class WFLCS {
    public static String lcslength(char[] X, char[] Y) {
        int m = X.length;    int n = Y.length;
        char[][] b = new char[m + 1][n + 1];
        int[][] c = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++ ){
            c[i][0] = 0;
        }
        for (int j = 0; j <= n; j++ ){
            c[0][j] = 0;
        }
        for (int i = 1; i <= m; i++ ){
            for (int j = 1; j <= n; j++ ){
                if (X[i - 1] == Y[j - 1]){                  // Jos löydetään täsmäys
                    c[i][j] = c[i - 1][j - 1] + 1;
                    b[i][j] = 'x';
                }else{
                    if (c[i - 1][j] >= c[i][j - 1]){        // Jos arvo yläpuolella ≥ arvo vasemmalla
                        c[i][j] = c[i - 1][j];
                        b[i][j] = 'y';
                    }else{
                        c[i][j] = c[i][j - 1];
                        b[i][j] = 'v';
                    }
                }
            }
        }
        String lcs = PrintLCS(b, X, m, n);                  // Etsitään PYA valmiin taulukon avulla
        return lcs;
    }
    private static String PrintLCS(char[][] b, char[] X, int i, int j){
        String lcs = "";
        while( i != 0 && j != 0){
            if (b[i][j] == 'x'){                                    // Jos tästä kohdasta löytyi täsmäys...
                lcs = String.valueOf(X[i - 1]) + lcs;               // ...lisätään merkki PYAn alkuun
                i--;   j--;
            }else{
                if (b[i][j] == 'y'){
                    i--;
                }else{
                    j--;
                }
            }
        }
        return lcs;
    }
}
```

Hirschbergin algoritmi:

```
import java.util.*;
import java.io.*;
public class HirschbergLCS {
        public static String lcslength(char[] A, char[] B) {
                int m = A.length;
                int n = B.length;
                String C = "";
/*1*/           HashMap<Character, Integer> NB = new HashMap<Character, Integer>();
                HashMap<Character, Integer> N = new HashMap<Character, Integer>();
                HashMap<Character, ArrayList<Integer>> PB =
                                new HashMap<Character, ArrayList<Integer>>();
                for (char i = 'a'; i<= 'z'; i++){
                        NB.put(i, 0);
                        ArrayList<Integer> places = new ArrayList<Integer>();
                        places.add(0);
                        PB.put(i, places);
                }
                for (int i = 0; i <= 9; i++){
                        NB.put(Integer.toString(i).charAt(0), 0);
                        ArrayList<Integer> places = new ArrayList<Integer>();
                        places.add(0);
                        PB.put(Integer.toString(i).charAt(0), places);
                }
                for (int j = 0; j < n; j++){          // Lisää B:n merkit listoihin
                        NB.put(B[j], NB.get(B[j]) + 1);
                        ArrayList<Integer> lista = PB.get(B[j]);
                        lista.add(NB.get(B[j]), j + 1);
                        PB.put(B[j], lista);
                }
                for (Map.Entry<Character, Integer> entry : NB.entrySet()) {
                        Integer value = entry.getValue();
                        if (value == 0){                    // Jos kyseessä on “nollamerkki”
                                Character key = entry.getKey();
                                NB.put(key, 1);             // Lisätään 1, jottei algoritmin suoritus kaadu
                        }
                }
                for (Map.Entry<Character, ArrayList<Integer>> entry : PB.entrySet()) {
                        ArrayList<Integer> value = entry.getValue();
                        if (value.size() == 1){     // Lisätään “nollamerkeille” toinen arvo nolla
                                value.add(0);
                                Character key = entry.getKey();
                                PB.put(key, value);
                        }
                }
/*2*/           int[][] D = new int[m + 1][m + 1];  // Löytyy maks. m yhteistä alkiota
                for (int i = 0; i <= m; i++){
                        D[0][i] = 0;
                }
                int lowcheck = 0;
                int k = 0;
/*3*/           int FLAG = 1;
```

```
            while (FLAG !=0){
                k++;
/*4*/           for (Map.Entry<Character, Integer> entry : NB.entrySet()) {
                    Integer value = entry.getValue();
                    Character key = entry.getKey();
                    N.put(key, value);
                }
                FLAG = 0;
                int low = D[k - 1][lowcheck];
                int high = n + 1;
/*5*/           for (int i = lowcheck + 1; i <=m; i++){
/*6*/               ArrayList<Integer> places = PB.get(A[i - 1]);
                    while (places.get(N.get(A[i - 1]) - 1) > low){
                        N.put(A[i - 1], N.get(A[i - 1]) - 1);  // Etsitään pienin low:ta suurempi
                    }
                    int temp = places.get(N.get(A[i - 1]));
/*7*/               if (high > temp && temp > low){  // Jos löytyi sallittu täsmäys
                        high = temp;
                        D[k][i] = high;
                        if (FLAG == 0){                 // Jos 1. tälle käyrälle löydetty täsmäys
                            lowcheck = i;
                            FLAG = 1;
                        }
                    } else {                            // Ei löytynyt sallittua täsmäystä
                        D[k][i] = 0;
                    }
/*8*/               if (D[k - 1][i] > 0){   // Jos edelliselle käyrälle löytyi tästä kohdasta täsmäys
                        low = D[k - 1][i];
                    }
                }
/*9*/       }
/*10*/      k--;
            for (int i = m; i > 0; i--){        // Selvitetään PYAn merkit
                if (D[k][i] > 0){
                    C = A[i - 1] + C;
                    k--;
                }
            }
            return C;
        }
}
```

Hirschbergin lineaarinen menetelmä:

```
import java.util.*;
import java.io.*;
public class HirschbergLinearLCS {

	/**
	 * Algoritmi C selvittää PYAn algoritmilta B saaduilla taulukoilla ja kutsumalla itseään
	 * rekursiivisesti pienempiin osiin jaetuilla syötteillä.
	 */
	public static String AlgC(int aLength, int bLength, String A, String B) {
		String C = "";
/*1*/	if (bLength < 2|| aLength < 2){			// Jos triviaali tapaus
			if (bLength == 0 || aLength == 0){	// Jos ainakin toisen pituus on 0
				return C;
			}else if (aLength == 1){		// Jos A:n pituus on 1
				for (int i = 0; i < bLength; i++){	// Etsi täsmäystä A:n merkille
					if (A.charAt(0) == B.charAt(i)){
						return A;
					}
				}
				return C;
			}else{					// Jos B:n pituus on 1
				for (int i = 0; i < aLength; i++){	// Etsi täsmäystä B:n merkille
					if (A.charAt(i) == B.charAt(0)){
						return B;
					}
				}
				return C;
			}
		}else{
/*2*/		int i = aLength/2;			// A:n katkaisukohta
/*3*/		int[] L1 = AlgB(i, bLength, A.substring(0, i), B);	// Kutsu B:tä A:n alkuosalla
		String Arev = new StringBuffer(A.substring(i, aLength)).reverse().toString();
		String Brev = new StringBuffer(B).reverse().toString();
		int[] L2 = AlgB(aLength - i, bLength, Arev, Brev);	// B:lle A:n loppu käännettynä
/*4*/		int M = 0;
		int k = 0;
		for ( int j = 0; j < bLength + 1; j++){
			if(L1[j]+L2[bLength  - j] > M){		// Etsi suurin summa L1 + L2
				M = L1[j]+L2[bLength  - j];
				k = j;
			}
		}
/*5*/		String C1 = AlgC(i, k, A.substring(0, i), B.substring(0, k));	// Rekursiiviset kutsut
		String C2 = AlgC(aLength - i, bLength - k, A.substring(i, aLength),
								B.substring(k, bLength));
/*6*/		C = C1 + C2;			// Yhdistä rekursiivisten kutsujen palauttamat tulokset
		}
		return C;
	}
```

```
        /**
         * Algoritmi B laskee viimeisen rivin A * B -taulukosta ja palauttaa sen
         */
        public static int[] AlgB(int aLength, int bLength, String A, String B){
/*1*/       int[] K0 = new int[bLength+1];
            int[] K1 = new int[bLength+1];
            for(int j = 0; j < bLength+1; j++){
                K1[j] = 0;
            }
/*2*/       for(int i = 0; i < aLength; i++){
/*3*/           for(int j = 0; j < bLength+1; j++){
                    K0[j] = K1[j];
                }
/*4*/           for(int j = 0; j < bLength; j++){
                    if(A.charAt(i)==B.charAt(j)){       // Jos löytyy täsmäys
                        K1[j+1] = K0[j] + 1;
                    }else{
                        K1[j+1] = Math.max(K1[j], K0[j+1]);
                    }
                }
            }
/*5*/       return K1;                                  // Palauta viimeinen rivi
        }
}
```

Hirschbergin algoritmi kaksisuuntaisena:

```
import java.util.*;
import java.io.*;
public class Hirschberg2wayLCS {
        public static String hb2w(char[] A, char[] B) {
            int m = A.length;
            int n = B.length;
/*1*/       HashMap<Character, Integer> NB = new HashMap<Character, Integer>();
            HashMap<Character, Integer> N = new HashMap<Character, Integer>();
            HashMap<Character, ArrayList<Integer>> PB
                                    = new HashMap<Character, ArrayList<Integer>>();
            for (char i = 'a'; i<= 'z'; i++){
                NB.put(i, 0);
                ArrayList<Integer> places = new ArrayList<Integer>();
                places.add(0);
                PB.put(i, places);
            }
            for (int i = 0; i <= 9; i++){
                NB.put(Integer.toString(i).charAt(0), 0);
                ArrayList<Integer> places = new ArrayList<Integer>();
                places.add(0);
                PB.put(Integer.toString(i).charAt(0), places);
            }
```

```
            for (int j = 0; j < n; j++){            // B:n merkkien lisääminen taulukoihin
                NB.put(B[j], NB.get(B[j]) + 1);
                ArrayList<Integer> lista = PB.get(B[j]);
                lista.add(NB.get(B[j]), j + 1);
                PB.put(B[j], lista);
            }
            for (Map.Entry<Character, Integer> entry : NB.entrySet()) {
                Integer value = entry.getValue();
                if (value == 0){                    // Jos kyseessä "nollamerkki"
                    Character key = entry.getKey();
                    NB.put(key, 1);             // Lisätään 1, jottei algoritmin suoritus kaadu
                }
            }
            for (Map.Entry<Character, ArrayList<Integer>> entry : PB.entrySet()) {
                ArrayList<Integer> value = entry.getValue();
                value.add(n+1);                 // Lisätään n + 1 kaikille viimeiseksi arvoksi
                Character key = entry.getKey();
                PB.put(key, value);
            }
/*2*/       int[][] D = new int[m + 2][m + 1]; // löytyy maks. m yhteistä alkiota + 2 aloitusriviä
            for (int i = 0; i <= m; i++){
                D[0][i] = 0;
                D[m+1][i] = n + 1;
            }
            int lowcheck = 0;
            int highcheck = m;
            int k = 0;
            int FLAG = 1;
            int FLAG2 = 1;
            int low = 0;
            int highlow = n + 1;
            int high2 = n + 1;
/*3*/       while (FLAG != 0 && FLAG2 != 0){
                k++;
/*4*/           for (Map.Entry<Character, Integer> entry : NB.entrySet()) {
                    Integer value = entry.getValue();
                    Character key = entry.getKey();
                    N.put(key, value);          // Taulukon N alustus jokaisella kierroksella
                }
                FLAG = 0;
                FLAG2 = 0;
                low = D[k - 1][lowcheck];
                high2 = D[m - k + 2][highcheck];
/*5*/           for (int i = lowcheck + 1; i  <=highcheck; i++){
/*6*/               ArrayList<Integer> places = PB.get(A[i - 1]);
                    while (places.get(N.get(A[i - 1]) - 1) > low){
                        N.put(A[i - 1], N.get(A[i - 1]) - 1);  // Etsitään pienin low:ta suurempi
                    }
                    int temp = places.get(N.get(A[i - 1]));
```

```
                // Jos löytyy sallittu täsmäys
/*7*/           if (temp < Math.abs(D[m + 2 - k][i]) && temp > low && temp < highlow){
                    highlow = temp;
                    D[k][i] = highlow;
                    if (FLAG == 0){       // Tämän käyrän 1. löydetty täsmäys
                        lowcheck = i;
                        FLAG = 1;
                    }
                } else {          // Asetetaan taulukkoon käyrän kulkukohta negatiivisena
                    D[k][i] = -Math.abs(D[k][i - 1]);
                }
/*8*/           if (D[k - 1][i] > 0){          // Jos edelliselle käyrälle löytyi täsmäys tästä kohdasta
                    low = D[k - 1][i];
                }
            }
/*9*/       if(highcheck > lowcheck && FLAG == 1){     // Jos voi löytyä lisää korkeuskäyriä
/* 4* */        for (Map.Entry<Character, Integer> entry : N.entrySet()) {
                    Character key = entry.getKey();
                    N.put(key, 0);                          // Taulukon N alustus nollilla
                }
/* 5* */        for (int j = highcheck - 1; j >= lowcheck; j--){
/* 6* */            ArrayList<Integer> places = PB.get(A[j]);
                    while(places.get(N.get(A[j]) + 1) < high2){
                        N.put(A[j],N.get(A[j]) + 1);    // Etsitään suurin high2:ta pienempi
                    }
                    int f = places.get(N.get(A[j]));
                // Jos löytyy sallittu täsmäys
/* 7* */            if (Math.abs(D[k][j]) < f && f < high2 && f > highlow){
                        highlow = f;
                        D[m - k + 1][j] = highlow;
                        if (FLAG2 == 0){      // Tämän käyrän 1. löydetty täsmäys
                            highcheck = j;
                            FLAG2 = 1;
                        }
                    } else {    // Asetetaan taulukkoon käyrän kulkukohta negatiivisena
                        D[m - k + 1][j] = -Math.abs(D[m - k + 1][j + 1]);
                    }
/* 8* */            int a = D[m - k + 2][j];
                    if (0 < a && a < n + 1){   // Jos edelliselle käyrälle löytyi täsmäys tästä
                        high2 = a;
                    }
                }
            }
        }
```

```
            String pisinalijono = "";
            int sarake1 = lowcheck;
            int sarake2 = highcheck;
            int rivi = m - k + 2;
            if(FLAG == 0){                          // Jos käyrien lukumäärä on parillinen
                k--;
                if(k == 0){return pisinalijono;}
                rivi = m - k + 1;
                int jatko = D[rivi][sarake2];
            // Etsitään täsmäystä, jota viimeiseksi (takaperin) löydetty käyrä jatkaa
                while (jatko <= D[k][sarake1]){
                    sarake1++;
                }
            }else{                                  // Käyrien lukumäärä on pariton
                if(k == 1){return "" + A[lowcheck - 1];}
                int alku = D[k][sarake1];
            // Etsitään täsmäystä, jota viimeiseksi (etuperin) löydetty käyrä jatkaa
                while (alku >= D[rivi][sarake2]){
                    sarake2--;
                }
            }
        // Etsitään etuperin löydetyistä käyristä sallitut min. täsmäykset
/*10*/      for (int j = sarake1; j > 0; j--){
                if(D[k][j] > 0){
                    pisinalijono = A[j - 1] + pisinalijono;
                    k--;
                    if(k == 0){
                        break;
                    }
                }
            }
        // Etsitään takaperin löydetyistä käyristä sallitut min. täsmäykset
/* 10* */   for (int j = sarake2; j < m; j++){
                if (D[rivi][j] > 0){
                    pisinalijono = pisinalijono + A[j];
                    rivi++;
                    if(rivi == m + 1){
                        break;
                    }
                }
            }
            return pisinalijono;
    }
}
```

Rickin algoritmi:

```
import java.util.*;
import java.io.*;
public class RickLCS {
        public static String Rick(char[] A, char[] B) {
/* 0 */       String lcs = "";
              int m = A.length;
              int n = B.length;
              Map<Character, int[]> CLOSEST_A = Closest(A);       // Suorasaantitaulukon luonti
              Map<Character, int[]> CLOSEST_B = Closest(B);       // Suorasaantitaulukon luonti
              int[] row_thresh = new int[m + 1];
              int[] col_thresh = new int[m + 1];
              int low = 0;
              int temp = 0;
              int i = 0;
              int j = 0;
              int k = 0;
              ArrayList<ArrayList<Link>> linkit = new ArrayList<ArrayList<Link>>();
              ArrayList<Link> klista = new ArrayList<Link>();
              klista.add(new Link(0,0,null));
              linkit.add(klista);
              ArrayList<Link> rowList = new ArrayList<Link>();
              ArrayList<Link> colList = new ArrayList<Link>();
              rowList.add(new Link(0,0,null));
              colList.add(new Link(0,0,null));
              Link tl = new Link(0,0,null);
              boolean found;
              boolean valmis = false;
/* 1 */       col_thresh[0] = 0;
              row_thresh[0] = 0;
              for(int p = 1; p <= m; p++){
                    col_thresh[p] = m + 1;
                    row_thresh[p] = n + 1;
              }
              low = 1;
/* 2 */       for(int L = 1; L <= m; L++){
/* 3 */             int[] taulukko_B = CLOSEST_B.get(A[L - 1]);
                    if(col_thresh[low] == L){                         // Korkeuskäyrä valmistui
                          j = taulukko_B[row_thresh[low]];
                          valmis = true;
                          low++;
                    } else {
                          j = taulukko_B[L - 1];
                    }
                    k = low;
                    found = false;
```

```
/* 4 */         while(j != n + 1){
                    if(j < row_thresh[k]){
                        temp = row_thresh[k];
                        row_thresh[k] = j;
                        if(rowList.size() == k){    // Onko ko. käyrän 1. täsmäys riveittäin
                            if (found){             // Tältä riviltä löytyi täsmäys käyrälle k-1
                                rowList.add(new Link(L, j, tl));
                            }else {                 // Ei löytynyt täsmäystä käyrälle k-1
                                rowList.add(new Link(L, j, rowList.get(k - 1)));
                            }
                        }else{      // Tälle käyrälle on löytynyt aikaisemmilta riveiltä täsmäys
                            if (found){             // Tältä riviltä löytyi täsmäys käyrälle k-1
                                Link tl2 = rowList.get(k);
                                rowList.set(k, new Link(L, j, tl));
                                tl = tl2;
                            }else {                 // Ei ole löytynyt täsmäystä käyrälle k-1
                                tl = rowList.get(k);
                                rowList.set(k, new Link(L, j, rowList.get(k - 1)));
                                found = true;
                            }
                        }
                        j = taulukko_B[temp];
                    }else{
                        found = false;
                    }
                    if(j == row_thresh[k]){
                        j = taulukko_B[j];
                    }
                    k++;
                }
                if(valmis){             // Kopioidaan viimeisin täsmäys colLististä rowListiin
                    if(rowList.size() == low - 1){          // Riveittäin ei löytynyt täsmäyksiä
                        rowList.add(low - 1, colList.get(low - 1));
                    }else{
                        rowList.set(low - 1, colList.get(low - 1));
                    }
                    valmis = false;
                }
                found = false;
/* 5 */         int[] taulukko_A = CLOSEST_A.get(B[L - 1]);
                if(row_thresh[low] == L){                       // Yksi korkeuskäyrä tuli valmiiksi
                    i = taulukko_A[col_thresh[low]];
                    valmis = true;
                    low++;
                } else {
                    i = taulukko_A[L];
                }
                k = low;
```

```
/* 6 */         while(i != m + 1){
                    if(i < col_thresh[k]){
                        temp = col_thresh[k];
                        col_thresh[k] = i;
                        if(colList.size() == k){    // Onko ko. käyrän 1. täsmäys sarakkeittain
                            if (found){             // Sarakkeesta löytyi täsmäys käyrälle k-1
                                colList.add(new Link(i, L, tl));
                            }else {                 // Ei löytynyt täsmäystä käyrälle k-1
                                colList.add(new Link(i, L, colList.get(k - 1)));
                            }
                        }else{      // Käyrälle on löytynyt täsmäys aikaisemmilta sarakkeilta
                            if (found){             // Sarakkeesta löytyi täsmäys käyrälle k-1
                                Link tl2 = colList.get(k);
                                colList.set(k, new Link(i, L, tl));
                                tl = tl2;
                            }else {                 // Sarakkeesta ei löytynyt k-1 täsmäystä
                                tl = colList.get(k);
                                colList.set(k, new Link(i, L, colList.get(k - 1)));
                                found = true;
                            }
                        }
                        i = taulukko_A[temp];
                    }else{
                        found = false;
                    }
                    if(i == col_thresh[k]){
                        i = taulukko_A[i];
                    }
                    k++;
                }
                if(valmis){                 // Kopioidaan viimeisin täsmäys rowLististä colListiin
                    if(colList.size() == low - 1){          // Sarakkeittain ei löytynyt täsmäyksiä
                        colList.add(low - 1, rowList.get(low - 1));
                    }else{
                        colList.set(low - 1, rowList.get(low - 1));
                    }
                    valmis = false;
                }
            }
/* 7 */     Link h;
            lcs = "";
            h = rowList.get(rowList.size() -1);
            for(int g = rowList.size() - 1; g > 0; g--){        // Käydään läpi kaikki käyrät
                lcs = A[h.getA()-1] + lcs;
                h = h.getEd();                                  // Siirrytään edelliselle käyrälle
            }
            return lcs;
        }
```

```
	/**
	 * Luo ja palauttaa suorasaantitaulukon annetusta syötteestä.
	 */
	private static HashMap<Character, int[]> Closest(char[] T){
		HashMap<Character, int[]> CLOSEST = new HashMap<Character, int[]>();
		int m = T.length;
		for(char i = 'a'; i <= 'z'; i++){
			int taulu[] = new int[m + 2];
			for(int j = 0; j < m + 2; j++){
				taulu[j] = m + 1;
			}
			CLOSEST.put(i, taulu);
		}
		for(int i = 0; i <= 9; i++){
			int taulu[] = new int[m + 2];
			for(int j = 0; j < m + 2; j++){
				taulu[j] = m + 1;
			}
			CLOSEST.put(Integer.toString(i).charAt(0), taulu);
		}
		for(int k = 0; k < m; k++){
			char c = T[k];
			int luvut[] = CLOSEST.get(c);
			int r = k;
			while(r >= 0 && luvut[r] == m + 1){
				luvut[r] = k + 1;
				r--;
			}
			CLOSEST.put(c, luvut);
		}
		return CLOSEST;
	}
	/**
	 * Linkki-luokka pitää sisällään yhden täsmäyksen tiedot sekä viittauksen mahdollisen
	 * edellisen täsmäyksen linkkiin.
	 * Omat metodit täsmäyksen molempien koordinaattien sekä edellisen linkin
	 * viittauksen kysymiseen.
	 */
	public static class Link {
		private final int A;
		private final int B;
		private final Link ed;
		public Link(int a, int b, Link ed_linkki) {
			this.A = a;
			this.B = b;
			this.ed = ed_linkki;
		}	// “Getterit”:
		public int getA() { return A; }
		public int getB() { return B; }
		public Link getEd() { return ed;}
	}
}
```

Rickin menetelmä kaksisuuntaisena:

```
import java.util.*;
import java.io.*;
public class Rick2wayLCS {
        public static String r2w(char[] A, char[] B) {
/* 0 */       String lcs = "";
              int m = A.length;
              int n = B.length;
              Map<Character, int[]> CLOSEST_A = Closest(A); // Suorasaantitaulukoiden luonti
              Map<Character, int[]> CLOSEST_B = Closest(B);
              Map<Character, int[]> CLOSEST_A2 = ReverseClosest(A);
              Map<Character, int[]> CLOSEST_B2 = ReverseClosest(B);
              int[] row_thresh = new int[m/2 + 2];
              int[] col_thresh = new int[m/2 + 2];
              int[] row_thresh2 = new int[m/2 + 2];
              int[] col_thresh2 = new int[m/2 + 2];
              int low = 1;
              int low2 = 1;
              int temp = 0;
              int i = 0;
              int j = 0;
              int k = 0;
              boolean found = false;
              boolean valmis = false;
              ArrayList<ArrayList<Link>> rowlinkit = new ArrayList<ArrayList<Link>>();
              ArrayList<ArrayList<Link>> rowlinkit2 = new ArrayList<ArrayList<Link>>();
              ArrayList<ArrayList<Link>> collinkit = new ArrayList<ArrayList<Link>>();
              ArrayList<ArrayList<Link>> collinkit2 = new ArrayList<ArrayList<Link>>();
              ArrayList<Link> lista = new ArrayList<Link>();
              lista.add(new Link(0,0,null));
              rowlinkit.add(lista);
              collinkit.add(lista);
              lista.clear();
              lista.add(new Link(m + 1,n + 1,null));
              rowlinkit2.add(lista);
              collinkit2.add(lista);
              int row1 = 0;
              int col1 = 0;
              int row2 = 0;
              int col2 = 0;
              Link temppi = null;
/* 1 */       col_thresh[0] = 0;              // Taulukoiden alustaminen
              row_thresh[0] = 0;
              col_thresh2[0] = 0;
              row_thresh2[0] = 0;
              for(int p = 1; p <= m/2 + 1; p++){
                    col_thresh[p] = m + 1;
                    row_thresh[p] = n + 1;
                    row_thresh2[p] = 0;
                    col_thresh2[p] = 0;
              }
```

```
/* 2 */     for(int L = 1; L <= m - m/2; L++){
/* 3 */         int[] taulukko_B = CLOSEST_B.get(A[L - 1]);
                if(col_thresh[low] == L){       // Tarkastetaan valmistuiko korkeuskäyrä
                    j = taulukko_B[row_thresh[low]];
                    valmis = true;
                    low++;
                    if(row1 == low - 2 && col2 > 0 && col_thresh2[col2] <= L &&
                                col2 != low2 - 1){     // Siirretään tarvittaessa ylin käyrä
                        lista = collinkit2.remove(col2);            // Poistetaan käyrä
                        rowlinkit.add(new ArrayList<Link>());
                        for (Link li:lista){
                            li.nollaa();                    // Nollataan tiedot edellisistä linkeistä
                            rowlinkit.get(low - 1).add(li);       // Lisätään täsmäykset
                        }
                        col2--;
                        row1++;
                    }
                } else {
                    j = taulukko_B[L - 1];
                }
                k = low;
                found = false;
/* 4 */         while(j != n + 1){
                    if(j < row_thresh[k] && j < n + 2 - L){   // Jos löytyy sallittu min. täsmäys
                        temp = row_thresh[k];
                        row_thresh[k] = j;
                        if(rowlinkit.size() <= k){              // Käyrän ensimmäinen täsmäys
                            rowlinkit.add(k, new ArrayList<Link>());
                            if(found){                  // Riviltä löytyi täsmäys käyrälle k - 1
                                rowlinkit.get(k).add(new Link(L, j, temppi));
                            }else{
                                rowlinkit.get(k).add(
                                            new Link(L, j, rowlinkit.get(k - 1).get(0)));
                            }
                        }else{                  // Käyrälle löytynyt jo täsmäyksiätähän suuntaan
                            if(found){              // Riviltä löytyi täsmäys käyrälle k - 1
                                rowlinkit.get(k).add(0, new Link(L, j, temppi));
                                temppi = rowlinkit.get(k).get(1);
                            }else{
                                rowlinkit.get(k).add(0,
                                            new Link(L, j, rowlinkit.get(k - 1).get(0)));
                                temppi = rowlinkit.get(k).get(1);
                                found = true;
                            }
                        }
```

```
                    if(row1 < k && col2 > 0 && col_thresh2[col2] <= L &&
                                    col2 != low2 - 1){ // Siirretään tarvittaessa ylin käyrä
                        lista = collinkit2.remove(col2);          // Poistetaan käyrä
                        for (Link li:lista){
                            li.nollaa();          // Nollataan tiedot edellisistä linkeistä
                            rowlinkit.get(k).add(li);       // Lisätään täsmäykset
                        }
                        col2--;
                    }
                    if(row1 < k){                          // Jos 1. käyrälle löydetty täsmäys
                        row1++;
                    }
                    j = taulukko_B[temp];
                    k++;
                }else{                              // Ei löytynyt sallittua minimaalista täsmäystä
                    found = false;
                    if(j == row_thresh[k]){
                        j = taulukko_B[j];
                    }else if(j > n + 1 - L){                // Jos täsmäys löytyi "liian kaukaa"
                        j = n + 1;
                    }
                    k++;
                }
            }
            if(valmis){          // Kopioidaan viimeisin täsmäys colLinkeistä rowLinkkeihin
                if(rowlinkit.size() == low - 1){       // Riveittäin ei löytynyt täsmäyksiä
                    rowlinkit.add(new ArrayList<Link>());
                    rowlinkit.get(low - 1).add(collinkit.get(low - 1).get(0));
                    row1++;
                }else{
                    rowlinkit.get(low - 1).add(0, collinkit.get(low - 1).get(0));
                }
                valmis = false;
            }
            found = false;
/* 5 */     int[] taulukko_A = CLOSEST_A.get(B[L - 1]);
            if(row_thresh[low] == L){     // Tarkastetaan valmistuiko käyrä
                i = taulukko_A[col_thresh[low]];
                valmis = true;
                low++;
                if(col1 == low - 2 && row2 > 0 && row_thresh2[row2] <= L &&
                                row2 != low2 - 1){ // Siirretään tarvittaessa ylin käyrä
                    lista = rowlinkit2.remove(row2);         // Poistetaan käyrä
                    collinkit.add(new ArrayList<Link>());
                    for (Link li:lista){
                        li.nollaa();                  // Nollataan tiedot edellisistä linkeistä
                        collinkit.get(low - 1).add(li);              // Lisätään täsmäykset
                    }
                    row2--;
                    col1++;
                }
            } else {
```

```
                    i = taulukko_A[L];
                }
                k = low;
/* 6 */             while(i != m + 1){
                    if(i < col_thresh[k] && i < m + 2 - L){    // Löytyykö sallittu min. täsmäys
                        temp = col_thresh[k];
                        col_thresh[k] = i;
                        if(collinkit.size() <= k){              // Jos käyrän ensimmäinen täsmäys
                            collinkit.add(k, new ArrayList<Link>());
                            if(found){                  // Riviltä löytyi täsmäys käyrälle k - 1
                                collinkit.get(k).add(new Link(i, L, temppi));
                            }else{
                                collinkit.get(k).add(new Link(i, L, collinkit.get(k - 1).get(0)));
                            }
                        }else{                  // Käyrälle löytynyt jo täsmäyksiä tähän suuntaan
                            if(found){                  // Riviltä löytyi täsmäys käyrälle k - 1
                                collinkit.get(k).add(0, new Link(i, L, temppi));
                                temppi = collinkit.get(k).get(1);
                            }else{
                                collinkit.get(k).add(0,
                                        new Link(i, L, collinkit.get(k - 1).get(0)));
                                temppi = collinkit.get(k).get(1);
                                found = true;
                            }
                        }
                        if(col1 < k && row2 > 0 && row_thresh2[row2] <= L &&
                                row2 != low2 - 1){    // Siirretään tarvittaessa ylin käyrä
                            lista = rowlinkit2.remove(row2);        // Poistetaan käyrä
                            for (Link li:lista){
                                li.nollaa();            // Nollataan tiedot edellisistä linkeistä
                                collinkit.get(k).add(li);               // Lisätään täsmäykset
                            }
                            row2--;
                        }
                        if(col1 < k){                       // Jos 1. käyrälle löydetty täsmäys
                            col1++;
                        }
                        i = taulukko_A[temp];
                        k++;
                    }else{                              // Ei löytynyt sallittua minimaalista täsmäystä
                        found = false;
                        if(i == col_thresh[k]){
                            i = taulukko_A[i];
                        }else if(i > m + 1 - L){            // Jos täsmäys löytyi "liian kaukaa"
                            i = m + 1;
                        }
                        k++;
                    }
                }
```

```
                if(valmis){             // Kopioidaan viimeisin täsmäys rowLinkeistä colLinkkeihin
                    if(collinkit.size() == low - 1){  // Sarakkeittain ei löytynyt täsmäyksiä
                        collinkit.add(new ArrayList<Link>());
                        collinkit.get(low - 1).add(rowlinkit.get(low - 1).get(0));
                        col1++;
                    }else{
                        collinkit.get(low - 1).add(0, rowlinkit.get(low - 1).get(0));
                    }
                    valmis = false;
                }
                // Tarkastelu takaperin
/* 3* */        if(L*2-1 == m){ break; } // Lopetetaan tarkastelu, jos kaikki rivit on jo käyty läpi
                taulukko_B = CLOSEST_B2.get(A[m - L]);
                if(col_thresh2[low2] == m + 1 - L){      // Tarkastetaan valmistuiko käyrä
                    j = taulukko_B[n + 2 - row_thresh2[low2]];
                    valmis = true;
                    low2++;
                    if(row2 == low2 - 2 && col1 > 0 && col_thresh[col1] >= m + 1 - L &&
                                    col1 != low - 1){    // Siirretään tarvittaessa ylin käyrä
                        lista = collinkit.remove(col1);                 // Poistetaan käyrä
                        rowlinkit2.add(new ArrayList<Link>());
                        for (Link li:lista){
                            li.nollaa();                // Nollataan tiedot edellisistä linkeistä
                            rowlinkit2.get(low2 - 1).add(li);       // Lisätään täsmäykset
                        }
                        col1--;
                        row2++;
                    }
                } else {
                    j = taulukko_B[L];
                }
                k = low2;
                found = false;
/* 4* */        while(j != 0){
                    if(j > row_thresh2[k] && j > L){          // Jos löytyy sallittu min. täsmäys
                        temp = row_thresh2[k];
                        row_thresh2[k] = j;
                        if(rowlinkit2.size() <= k){           // Jos 1. käyrälle löytynyt täsmäys
                            rowlinkit2.add(k, new ArrayList<Link>());
                            if(found){                  // Riviltä löytyi täsmäys käyrälle k - 1
                                rowlinkit2.get(k).add(new Link(m + 1 - L, j, temppi));
                            }else{
                                rowlinkit2.get(k).add(
                                    new Link(m + 1 - L, j, rowlinkit2.get(k - 1).get(0)));
                            }
                        }else{                  // Käyrälle löytynyt jo täsmäyksiä tähän suuntaan
```

```
                    if(found){                    // Riviltä löytyi täsmäys käyrälle k - 1
                        rowlinkit2.get(k).add(0, new Link(m + 1 - L, j, temppi));
                        temppi = rowlinkit2.get(k).get(1);
                    }else{
                        rowlinkit2.get(k).add(0,
                            new Link(m + 1 - L, j, rowlinkit2.get(k - 1).get(0)));
                        temppi = rowlinkit2.get(k).get(1);
                        found = true;
                    }
                }
                if(row2 < k && col1 > 0 && col_thresh[col1] >= m + 1 - L &&
                        col1 != low - 1){     // Siirretään tarvittaessa ylin käyrä
                    lista = collinkit.remove(col1);              // Poistetaan käyrä
                    for (Link li:lista){
                        li.nollaa();             // Nollataan tiedot edellisistä linkeistä
                        rowlinkit2.get(k).add(li);              // Lisätään täsmäykset
                    }
                    col1--;
                }
                if(row2 < k){                            // Jos 1. käyrälle löytynyt täsmäys
                    row2++;
                }
                j = taulukko_B[n + 2 - temp];
                k++;
            }else{                              // Ei löytynyt sallittua minimaalista täsmäystä
                found = false;
                if(j == row_thresh2[k]){
                    j = taulukko_B[n + 2 - j];
                }else if(j <= L){                    // Jos täsmäys löytyi "liian kaukaa"
                    j = 0;
                }
                k++;
            }
        }
        if(valmis){      // Kopioidaan viimeisin täsmäys colLinkit2:sta rowLinkit2:een
            if(rowlinkit2.size() == low2 - 1){    // Riveittäin ei löytynyt täsmäyksiä
                rowlinkit2.add(new ArrayList<Link>());
                rowlinkit2.get(low2 - 1).add(collinkit2.get(low2 - 1).get(0));
                row2++;
            }else{
                rowlinkit2.get(low2 - 1).add(0, collinkit2.get(low2 - 1).get(0));
            }
            valmis = false;
        }
        found = false;
```

```
/* 5* */    taulukko_A = CLOSEST_A2.get(B[n - L]);
                if(row_thresh2[low2] == n + 1 - L){      // Tarkastetaan valmistuiko käyrä
                    i = taulukko_A[m + 2 - col_thresh2[low2]];
                    valmis = true;
                    low2++;
                    if(col2 == low2 - 2 && row1 > 0 && row_thresh[row1] >= n + 1 - L &&
                                row1 != low - 1){ // Siirretään tarvittaessa ylin käyrä
                        lista = rowlinkit.remove(row1);                 // Poistetaan käyrä
                        collinkit2.add(new ArrayList<Link>());
                        for (Link li:lista){
                            li.nollaa();                    // Nollataan tiedot edellisistä linkeistä
                            collinkit2.get(low2 - 1).add(li);     // Lisätään täsmäykset
                    }
                            row1--;
                            col2++;
                    }
                } else {
                    i = taulukko_A[L + 1];
                }
                k = low2;
/* 6* */    while(i != 0){
                    if(i > col_thresh2[k] && i > L){      // Löytyykö sallittu min. täsmäys
                        temp = col_thresh2[k];
                        col_thresh2[k] = i;
                        if(collinkit2.size() <= k){              // Jos 1. käyrälle löydetty täsmäys
                            collinkit2.add(k, new ArrayList<Link>());
                            if(found){                  // Riviltä löytyi täsmäys käyrälle k - 1
                                collinkit2.get(k).add(new Link(i, n + 1 - L, temppi));
                            }else{
                                collinkit2.get(k).add(
                                        new Link(i, n + 1 - L, collinkit2.get(k - 1).get(0)));
                            }
                        }else{                  // Käyrälle löytynyt jo aiemmin täsmäyksiä
                        if(found){                      // Riviltä löytyi täsmäys käyrälle k - 1
                                collinkit2.get(k).add(0, new Link(i, n + 1 - L, temppi));
                                temppi = collinkit2.get(k).get(1);
                            }else{
                                collinkit2.get(k).add(0,
                                        new Link(i, n + 1 - L, collinkit2.get(k - 1).get(0)));
                                temppi = collinkit2.get(k).get(1);
                                found = true;
                            }
                        }
                        if(col2 < k && row1 > 0 && row_thresh[row1] >= n + 1 - L &&
                                row1 != low - 1){    // Siirretään tarvittaessa ylin käyrä
                            lista = rowlinkit.remove(row1);                 // Poistetaan käyrä
                            for (Link li:lista){
                                li.nollaa();            // Nollataan tiedot edellisistä linkeistä
                                collinkit2.get(k).add(li);              // Lisätään täsmäykset
                            }
                            row1--;
                        }
```

```
                        if(col2 < k){                       // Jos 1. käyrälle löydetty täsmäys
                            col2++;
                        }
                        i = taulukko_A[m + 2 - temp];
                        k++;
                    }else{                          // Ei löytynyt sallittua minimaalista täsmäystä
                        found = false;
                        if(i == col_thresh2[k]){
                            i = taulukko_A[m + 2 - i];
                        }else if(i <= L){               // Jos täsmäys löytyi "liian kaukaa”
                            i = 0;
                        }
                        k++;
                    }
                }
                if(valmis){     // Kopioidaan viimeisin täsmäys rowLinkit2:sta colLinkit2:een
                    if(collinkit2.size() == low2 - 1){  // Sarakkeittain ei löytynyt täsmäyksiä
                        collinkit2.add(new ArrayList<Link>());
                        collinkit2.get(low2 - 1).add(rowlinkit2.get(low2 - 1).get(0));
                        col2++;
                    }else{
                        collinkit2.get(low2 - 1).add(0, rowlinkit2.get(low2 - 1).get(0));
                    }
                    valmis = false;
                }
            }
/* 7 */     Link y = null;
            lcs = "";
        // Jos molemmista suunnista löytyi riveittäin enemmän käyriä kuin sarakkeittain
            if(row1 > col1 && row2 > col2){
                boolean flag = false;
                int u = row1;                               // Yläkäyrän muuttuja
                int v = col2 + 1;                           // Alakäyrän muuttuja
                while(u != col1 && v != row2 + 1){          // Etsitään, kunnes käyrät loppuvat
                    if(row_thresh[u] < row_thresh2[v]){     // Jos käyrät jatkavat toisiaan
                        flag = true;
                        i = u;          // Otetaan toisiaan jatkavien käyrien numerot muistiin
                        j = v;
                    } else{
                        u--;                                // Edellinen käyrä ylhäältä
                    }
                    v++;                                    // Seuraava käyrä alhaalta
                }
                if(flag){       // Jos pisin PYA löytyy ylhäältä + alhaalta riveittäin, selvitetään se
                    y = rowlinkit.get(i).get(0);
                    for(int g = i; g > 0; g--){
                        lcs = A[y.getA()-1] + lcs;
                        y = y.getEd();
                    }
```

```
                    y = rowlinkit2.get(j).get(0);
                    for(int g = j; g > 0; g--){
                        lcs = lcs + A[y.getA()-1];
                        y = y.getEd();
                    }
                    return lcs;
                }else{                  // Käyrät loppuivat ilman, että mikään jatkoi toista
                    if(u == col1){      // Käyrät loppuivat ylhäältä
                        row1 = col1;
                    }else{              // Käyrät loppuivat alhaalta
                        row2 = col2;
                    }
                }
            }
            if((row1 + col2) < (row2 + col1)){  //  vasemmalta + alhaalta > ylhäältä + oikealta
                ArrayList<Link> lista1 = collinkit.get(col1);
                ArrayList<Link> lista2 = rowlinkit2.get(row2);
                ArrayList<Link> loppulista = new ArrayList<Link>();
                if(row2 == 0){                  // Jos alhaalta ei löytynyt käyriä
                    y = lista1.get(0);
                    for(int g = col1; g > 0; g--){    // Etsitään PYA vasemmalta
                        lcs = A[y.getA()-1] + lcs;
                        y = y.getEd();
                    }
                    return lcs;
                }
                if(col1 == 0){                  // Jos vasemmalta ei löytynyt käyriä
                    y = lista2.get(0);
                    for(int g = row2; g > 0; g--){   // Etsitään PYA alhaalta
                        lcs = lcs + A[y.getA()-1];
                        y = y.getEd();
                    }
                    return lcs;
                }
                for(int p = 0; p < lista1.size(); p++){  // Lähdetään vas. kork. käyrän täsmäyksistä
                    temp = -1;
                    y = lista1.get(p);          // Täsmäys, jolle haetaan edeltäjiä ja seuraajia
                    while(temp < lista2.size()){
                        temp = voiJatkaa(y, lista2, temp + 1, true); // Etsii seuraajan paikan
                        if(temp != - 1){                // Jos löytyi sallittu seuraaja
                            lista.clear();
                            lista.add(y);               // Lisätään alkupiste PYA-listaan
                            lista = etsi_ed(lista, collinkit, col1 - 1, false); // Etsitään alkuosaa
                            if(lista.size()!=0){        // Jos alkupää löytyi sallitusti
                                loppulista.clear();
                                loppulista.add(lista2.get(temp));  // Lisätään lähtöpiste
                            // Etsitään PYAn mahdollinen loppuosa
                                loppulista = etsi_ed(loppulista, rowlinkit2, row2 - 1, true);
                                if(loppulista.size() != 0){         // Jos PYAn loppuosakin
```

löytyi -> etsitään ja palautetaan se, muuten jatketaan seuraavan sallitun seuraajan etsimistä

```
                        // Alkuosan rekursiivisesti löydetyt
                            for(int h = 0; h < lista.size(); h++){
                                lcs = lcs + A[lista.get(h).getA()-1];
                            }
                            y = lista.get(0).getEd();
                        // Alkuosan loput tiedossa olevat
                            for(int h = col1 - lista.size(); h > 0; h--){
                                lcs = A[y.getA()-1] + lcs;
                                y = y.getEd();
                            }
                        // Loppuosan rekursiivisesti löydetyt
                            for(int h = loppulista.size()-1; h >= 0; h-- ){
                                lcs = lcs + A[loppulista.get(h).getA()-1];
                            }
                            y = loppulista.get(0).getEd();
                        // Loppuosan loput tiedossa olevat
                            for(int h = row2 - loppulista.size(); h > 0; h--){
                                lcs = lcs + A[y.getA()-1];
                                y = y.getEd();
                            }
                            return lcs;             // Palauta löytynyt PYA
                        }   // Sallittua loppuosaa ei löytynyt, etsitään uusi jatkaja
                    }else{                  // Alkupäätä (=> ja PYAa) ei löytynyt sallitusti
                        break;
                    }
                }else{                  //  ei löytynyt sallitusti tästä linkistä lähtiessä
                    break;
                }
            }
        }
    }else{          //  on pidempi ylhäältä + oikealta kuin vasemmalta + alhaalta
        ArrayList<Link> lista1 = rowlinkit.get(row1);
        ArrayList<Link> lista2 = collinkit2.get(col2);
        ArrayList<Link> loppulista = new ArrayList<Link>();
        if(row1 == 0){              // Jos ylhäältä ei löytynyt käyriä
            if(col2 == 0){          // Jos käyriä ei ole löytynyt yhtäkään
                return "";          // Palautetaan tyhjä merkkijono
            }else{                  // Selvitetään PYA oikealta löytyneistä täsmäyksistä
                y = lista2.get(0);
                for(int g = col2; g > 0; g--){
                    lcs = lcs + A[y.getA()-1];
                    y = y.getEd();
                }
            }
            return lcs;
        }
        if(col2 == 0){          // Jos oikealla ei ole käyriä
            y = lista1.get(0);
            for(int g = row1; g > 0; g--){   // Selvitetään PYA ylhäältä
                lcs = A[y.getA()-1] + lcs;
                y = y.getEd();
            }
```

```
                return lcs;
            }
            for(int p = 0; p < lista1.size(); p++){  // Lähdetään ylh. kork. käyrän täsmäyksistä
                temp = -1;
                y = lista1.get(p);          // Täsmäys, jolle haetaan edeltäjiä ja seuraajia
                while(temp < lista2.size()){
                    temp = voiJatkaa(y, lista2, temp + 1, true); // Etsii seuraajan paikan
                    if(temp != - 1){                    // Jos löytyi sallittu seuraaja
                        lista.clear();
                        lista.add(y);                   // Lisätään alkupiste PYA-listaan
                        lista = etsi_ed(lista, rowlinkit, row1 - 1, false); // Etsitään alkuosa
                        if(lista.size()!=0){            // Jos alkupää löytyi sallitusti
                            loppulista.clear();
                            loppulista.add(lista2.get(temp));  // Lisätään lähtöpiste
                        // Etsi PYAn mahdollinen loppuosa
                            loppulista = etsi_ed(loppulista, collinkit2, col2 - 1, true);
                            if(loppulista.size() != 0){          // Jos PYAn loppuosakin
löytyi -> selvitetään ja palautetaan se, muuten jatketaan seuraavan sallitun seuraajan etsimistä
                            // Alkuosan rekursiivisesti löydetyt
                                for(int h = 0; h < lista.size(); h++){
                                    lcs = lcs + A[lista.get(h).getA()-1];
                                }
                                y = lista.get(0).getEd();
                            // Alkuosan loput tiedossa olevat
                                for(int h = row1 - lista.size(); h > 0; h--){
                                    lcs = A[y.getA()-1] + lcs;
                                    y = y.getEd();
                                }
                            // Loppuosan rekursiivisesti löydetyt
                                for(int h = loppulista.size()-1; h >= 0; h-- ){
                                    lcs = lcs + A[loppulista.get(h).getA()-1];
                                }
                                y = loppulista.get(0).getEd();
                            // Loppuosan loput tiedossa olevat
                                for(int h = col2 - loppulista.size(); h > 0; h--){
                                    lcs = lcs + A[y.getA()-1];
                                    y = y.getEd();
                                }
                                return lcs;                 // Palauta löytynyt PYA
                            }   // Sallittua loppuosaa ei löytynyt, etsitään uusi jatkaja
                        }else{                  // Alkupäätä (=> ja PYAa) ei löytynyt sallitusti
                            break;
                        }
                    }else{                  //  ei löytynyt sallitusti tästä linkistä lähtiessä
                        break;
                    }
                }
            }
        }
        return "";          // (Jos tänne joudutaan, on tapahtunut jokin virhe)
    }
```

```
/**
 * Luo ja palauttaa suorasaantitaulukon annetusta syötteestä.
 */
private static HashMap<Character, int[]> Closest(char[] T){
	HashMap<Character, int[]> CLOSEST = new HashMap<Character, int[]>();
	int m = T.length;
	for(char i = 'a'; i <= 'z'; i++){
		int taulu[] = new int[m + 2];
		for(int j = 0; j < m + 2; j++){
			taulu[j] = m + 1;
		}
		CLOSEST.put(i, taulu);
	}
	for(int i = 0; i <= 9; i++){
		int taulu[] = new int[m + 2];
		for(int j = 0; j < m + 2; j++){
			taulu[j] = m + 1;
		}
		CLOSEST.put(Integer.toString(i).charAt(0), taulu);
	}
	for(int k = 0; k < m; k++){
		char c = T[k];
		int luvut[] = CLOSEST.get(c);
		int r = k;
		while(r >= 0 && luvut[r] == m + 1){
			luvut[r] = k + 1;
			r--;
		}
		CLOSEST.put(c, luvut);
	}
	return CLOSEST;
}

/**
 * Luo ja palauttaa suorasaantitaulukon annetusta syötteestä
 * alhaalta oikealta lähtevää takaperin tapahtuvaa tarkastelua varten.
 */
private static HashMap<Character, int[]> ReverseClosest(char[] T){
	HashMap<Character, int[]> CLOSEST = new HashMap<Character, int[]>();
	int m = T.length;
	for(char i = 'a'; i <= 'z'; i++){
		int taulu[] = new int[m + 3];
		CLOSEST.put(i, taulu);
	}
	for(int i = 0; i <= 9; i++){
		int taulu[] = new int[m + 3];
		CLOSEST.put(Integer.toString(i).charAt(0), taulu);
	}
```

```
        for(int k = m; k > 0; k--){
            char c = T[k - 1];
            int luvut[] = CLOSEST.get(c);
            int r = m + 1 - k;
            while(r > 0 && luvut[r] == 0){
                luvut[r] = k;
                r--;
            }
            CLOSEST.put(c, luvut);
        }
        return CLOSEST;
    }

    /**
     * Etsii syötteenä saadusta listasta jatkajaa annetulle täsmäykselle annetusta kohdasta
     * lähtien. Palauttaa seuraavan mahdollisen jatkajan sijainnin taulukosta tai -1, jos
     * jatkajaa ei löydy. Saa argumenttina myös tarkastelusuunnan
     * (true = ylhäältä alaspäin, false = alhaalta ylöspäin).
     */
    private static int voiJatkaa(Link ylh, ArrayList<Link> jatko, int alku, boolean alas){
        if(alas){
            for(int i = alku; i < jatko.size(); i++){
                if(jatko.get(i).getA() > ylh.getA() && jatko.get(i).getB() > ylh.getB()){
                    return i;
                }
            }
        }else{
            for(int i = alku; i < jatko.size(); i++){
                if(jatko.get(i).getA() < ylh.getA() && jatko.get(i).getB() < ylh.getB()){
                    return i;
                }
            }
        }
        return -1;
    }

    /**
     * Etsii rekursiivisesti edeltävät linkit (sallitut täsmäykset) annetun listan ensimmäiselle
     * linkille (täsmäykselle), kunnes löytyy tieto edellisestä linkistä. Saa argumenttina PYAn
     * linkkilistan, etsittävästä suunnasta tarvittavat linkkilistat, nykyisen korkeuskäyrän sekä
     * tarkastelusuunnan (true = ylhäältä alaspäin, false = alhaalta ylöspäin).
     * Jos löytyy tieto edellisestä, löytyneet täsmäykset palautetaan listana.
     * Jos jossain vaiheessa sallittua linkkiä ei löydy, palautetaan tyhjä lista.
     */
    private static ArrayList<Link> etsi_ed(ArrayList<Link> lista,
                                           ArrayList<ArrayList<Link>> linkit, int k, boolean alas){
        if(k == 0 || lista.size() == 0){return lista;}          // Palautetaan triviaali tapaus
        Link y = lista.get(0).getEd();
        if(y != null){          // Jos edeltäjä on tiedossa, palautetaan valmis rekursiivinen lista
            return lista;
        }else{                  // Etsitään sallittu edeltäjä
```

```
                    y = lista.get(0);
                    ArrayList<Link> seuraajat = linkit.get(k);
                    ArrayList<Link> palautus = new ArrayList<Link>();
                    int temp = -1;
                    while(temp < seuraajat.size()){
                    // Etsi seuraava mahdollinen jatkaja
                        temp = voiJatkaa(y, seuraajat, temp + 1, alas);
                        if(temp != -1){              // Jos jatkaja löytyi
                            for(Link linkki : lista){
                                palautus.add(linkki);
                            }
                            palautus.add(0,seuraajat.get(temp));
                            palautus = etsi_ed(palautus, linkit, k - 1, alas); // Etsitään seuraavat
                            if(palautus.size() != 0){        // Jos löytyi, palautetaan lista
                                return palautus;
                            }
                        }else{                  // Ei löytynyt sallittua jatkajaa, palautetaan tyhjä lista
                            palautus.clear();
                            return palautus;
                        }
                    }
                }
                return null;
            }
            /**
             * Linkki-luokka pitää sisällään yhden täsmäyksen tiedot sekä viittauksen mahdollisen
             * edellisen täsmäyksen linkkiin.
             * Omat metodit täsmäyksen molempien koordinaattien ja edellisen linkin
             * viittauksen kysymiseen sekä edeltäjän nollaamiseen.
             */
            public static class Link {
                private final int A;
                private final int B;
                private Link ed;
                public Link(int a, int b, Link ed_linkki) {
                    this.A = a;
                    this.B = b;
                    this.ed = ed_linkki;
                }
                // “Getterit”:
                public int getA() { return A; }
                public int getB() { return B; }
                public Link getEd() { return ed;}
                /**
                 * Nollaa tarvittaessa tiedon edellisestä linkistä esim. käyrän vaihdon yhteydessä.
                 */
                public void nollaa(){
                    this.ed = null;
                }
            }
        }
```

Vertailut tekevä algoritmi:

```
import java.util.*;
import java.io.*;
public class vertailu {
	public static void main(String[] args) throws Exception {
		/* Komentokehotteen kutsu tehtäessä vertailuja suurilla syötteillä:
		 * java -d64 -Xmx14g vertailu [lkm] [wf] [hbl] [hb] [hb2] [ri] [ri2], viimeiset 6: 1 / 0
		 */
		int lkm = 2500;
		int testattavat[] = {	// 1 => otetaan mukaan, 0 => ei oteta mukaan
			1,	// Wagner-Fischer
			1,	// Hirschberg lineaarinen
			1,	// Hirschberg alkuperäinen
			1,	// Hirschberg 2-suuntainen
			1,	// Rick alkuperäinen
			1};	// Rick 2-suuntainen
		switch (args.length) {
			case 7: testattavat[5] = Integer.parseInt(args[6]);
			case 6: testattavat[4] = Integer.parseInt(args[5]);
			case 5: testattavat[3] = Integer.parseInt(args[4]);
			case 4: testattavat[2] = Integer.parseInt(args[3]);
			case 3: testattavat[1] = Integer.parseInt(args[2]);
			case 2: testattavat[0] = Integer.parseInt(args[1]);
			case 1: lkm = Integer.parseInt(args[0]);
			default :
		}
		String merkit = "abcdefghijklmnopqrstuvwxyz0123456789";	// Kaikki sallitut merkit
		long aika;
		String X2;
		char X[];
		String Y2;
		char Y[];
		BufferedWriter printti = new BufferedWriter(new FileWriter("testit" + lkm + ".csv",
false));
		WFLCS wf = new WFLCS();
		HirschbergLinearLCS hbl = new HirschbergLinearLCS();
		HirschbergLCS hb = new HirschbergLCS();
		Hirschberg2wayLCS hb2 = new Hirschberg2wayLCS();
		RickLCS ri = new RickLCS();
		Rick2wayLCS ri2 = new Rick2wayLCS();
		int pya;
		boolean onko = true;
		int kierros = 0;
		int kierroslkm = 10;				// Testikierrosten kokonaislukumäärä
		String lcs = "";
		String lcs2 = "";
		int muisti = 0;
		double currentMemory = 0;
		// Algoritmien nimien (otsikoiden) tulostus
		printti.write(lkm + " merkkiä");
		printti.newLine();
```

```
printti.write("Wagner-Fischer;;Hirschberg lin.;;Hirschberg;;Hirschberg 2way;;
        Rick;;Rick 2way;;PYA pituus");
while(kierros < kierroslkm){
    pya = -1;
// Tiedostojen luonti
    randomgenerator.luoTaulukkoTiedostoon("G:\\Dropbox\\Gradu\\
        testitiedostot\\testi" + kierros + "merkkijono" + lkm + ".txt", lkm, merkit);
    randomgenerator.luoTaulukkoTiedostoon("G:\\Dropbox\\Gradu\\
        testitiedostot\\testi" + kierros + "merkkijono" + lkm + "_2.txt", lkm, merkit);
    BufferedReader lukija2 = new BufferedReader(new FileReader("G:\\Dropbox\\
        Gradu\\testitiedostot\\testi" + kierros + "merkkijono" + lkm + ".txt"));
    X2 = lukija2.readLine();
    X = X2.toCharArray();
    lukija2 = new BufferedReader(new FileReader("G:\\Dropbox\\Gradu\\
        testitiedostot\\testi" + kierros + "merkkijono" + lkm + "_2.txt"));
    Y2 = lukija2.readLine();
    Y = Y2.toCharArray();
    lukija2.close();
    printti.newLine();
    // Wagner-Fischer
    if(testattavat[0] == 1){
        System.gc(); System.gc();
        aika = System.currentTimeMillis();
        lcs = wf.lcslength(X, Y);
        aika = System.currentTimeMillis() - aika;
        currentMemory = (
            (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
            ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
        muisti = (int)currentMemory;
        printti.write(aika + ";" + muisti + ";");
        if(pya == -1){
            pya = lcs.length();
        }else if(pya != lcs.length()){
            onko = false;
        }
    }else{
        printti.write(";;");
    }
    // Hirschberg lineaarinen
    if(testattavat[1] == 1){
        System.gc(); System.gc();
        aika = System.currentTimeMillis();
        lcs = hbl.AlgC(X.length, Y.length, X2, Y2);
        aika = System.currentTimeMillis() - aika;
        currentMemory = (
            (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
            ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
        muisti = (int)currentMemory;
        printti.write(aika + ";" + muisti + ";");
        if(pya == -1){
            pya = lcs.length();
        }else if(pya != lcs.length()){
```

```
                onko = false;
        }
    }else{
        printti.write(";;");
    }
    // Hirschberg alkuperäinen
    if(testattavat[2] == 1){
        System.gc(); System.gc();
        aika = System.currentTimeMillis();
        lcs = hb.lcslength(X, Y);
        aika = System.currentTimeMillis() - aika;
        currentMemory = (
            (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
            ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
        muisti = (int)currentMemory;
        printti.write(aika + ";" + muisti + ";");
        if(pya == -1){
                pya = lcs.length();
        }else if(pya != lcs.length()){
                onko = false;
        }
    }else{
        printti.write(";;");
    }
    // Hirschberg 2-suuntainen
    if(testattavat[3] == 1){
        System.gc(); System.gc();
        aika = System.currentTimeMillis();
        lcs = hb2.hb2w(X, Y);
        aika = System.currentTimeMillis() - aika;
        currentMemory = (
            (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
            ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
        muisti = (int)currentMemory;
        printti.write(aika + ";" + muisti + ";");
        if(pya == -1){
                pya = lcs.length();
        }else if(pya != lcs.length()){
                onko = false;
        }
    }else{
        printti.write(";;");
    }
    // Rick alkuperäinen
    if(testattavat[4] == 1){
        System.gc(); System.gc();
        aika = System.currentTimeMillis();
        lcs = ri.Rick(X, Y);
        aika = System.currentTimeMillis() - aika;
        currentMemory = (
            (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
            ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
```

```
                    muisti = (int)currentMemory;
                    printti.write(aika + ";" + muisti + ";");
                    if(pya == -1){
                        pya = lcs.length();
                    }else if(pya != lcs.length()){
                        onko = false;
                    }
                }else{
                    printti.write(";;");
                }
                // Rick 2-suuntainen
                if(testattavat[5] == 1){
                    System.gc(); System.gc();
                    aika = System.currentTimeMillis();
                    lcs = ri2.r2w(X, Y);
                    aika = System.currentTimeMillis() - aika;
                    currentMemory = (
                       (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
                       ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
                    muisti = (int)currentMemory;
                    printti.write(aika + ";" + muisti + ";" + lcs.length());
                    if(pya == -1){
                        pya = lcs.length();
                    }else if(pya != lcs.length()){
                        onko = false;
                    }
                }else{
                    printti.write(";;" + pya);
                }
                kierros++;
            }
            if(onko){
                printti.newLine();
                printti.newLine();
                printti.close();
                System.out.println(lkm + " valmis");
            }else{
                System.out.println("Virhe: Jokin algoritmi löysi väärän pituisen PYAn");
            }
        }
}
```

Eripituisia syötteitä eri järjestyksessä testaava algoritmi:

```
import java.util.*;
import java.io.*;
public class vertailu2 {
	public static void main(String[] args) throws Exception {
		/* komentokehotteen kutsu suurilla syötteillä suoritettaessa:
		 * java -d64 -Xmx14g vertailu2 [lkm1] [lkm2] [wf] [hbl] [hb] [hb2] viimeiset 4: 1 / 0
		 */
		int lkm = 2000;
		int lkm2 = 10000;
		int testattavat[] = {	// 1 => otetaan mukaan, 0 => ei oteta mukaan
			1,	// Wagner-Fischer
			1,	// Hirschberg lineaarinen
			1,	// Hirschberg alkuperäinen
			1};	// Hirschberg 2-suuntainen
		switch (args.length) {
			case 6:	testattavat[3] = Integer.parseInt(args[5]);
			case 5:	testattavat[2] = Integer.parseInt(args[4]);
			case 4:	testattavat[1] = Integer.parseInt(args[3]);
			case 3:	testattavat[0] = Integer.parseInt(args[2]);
			case 2:	lkm2 = Integer.parseInt(args[1]);
				lkm = Integer.parseInt(args[0]);
				break;
			case 1:	lkm = Integer.parseInt(args[0]);
				lkm2 = Integer.parseInt(args[0]);
			default :
		}
		String merkit = "abcdefghijklmnopqrstuvwxyz0123456789";  // Kaikki sallitut merkit
		long aika;
		String X2;
		char X[];
		String Y2;
		char Y[];
		BufferedWriter printti = new BufferedWriter(
				new FileWriter("testit" + lkm + "," + lkm2 + ".csv", false));
		WFLCS wf = new WFLCS();
		HirschbergLinearLCS hbl = new HirschbergLinearLCS();
		HirschbergLCS hb = new HirschbergLCS();
		Hirschberg2wayLCS hb2 = new Hirschberg2wayLCS();
		RickLCS ri = new RickLCS();
		Rick2wayLCS ri2 = new Rick2wayLCS();
		int pya;
		boolean onko = true;
		int kierros = 0;
		int kierroslkm = 5;				// Testikierrosten kokonaislukumäärä
		String lcs = "";
		String lcs2 = "";
		int muisti = 0;
		double currentMemory = 0;
		// Algoritmien nimien (otsikoiden) tulostus
		printti.write(lkm + " ja " + lkm2 + " merkkiä");
		printti.newLine();
```

```
printti.write("Wagner-Fischer;;X>Y;;PYA1=PYA2;Hirschberg lin.;;X>Y;;PYA1=PYA2;
        Hirschberg;;X>Y;;PYA1=PYA2;Hirschberg 2way;;X>Y;;PYA1=PYA2;PYA pituus");
while(kierros < kierroslkm){
      pya = -1;
// Tiedostojen luonti
      randomgenerator.luoTaulukkoTiedostoon("G:\\Dropbox\\Gradu\\
            testitiedostot\\merkkijono" + lkm + "," + lkm2 +"testi" + kierros +
            ".txt", lkm, merkit);
      randomgenerator.luoTaulukkoTiedostoon("G:\\Dropbox\\Gradu\\
            testitiedostot\\merkkijono" + lkm + "," + lkm2 +"testi" + kierros +
            "_2.txt", lkm2, merkit);
      BufferedReader lukija2 = new BufferedReader(new FileReader("G:\\Dropbox\\
            Gradu\\testitiedostot\\merkkijono" + lkm + "," + lkm2 +"testi" + kierros
            + ".txt"));
      X2 = lukija2.readLine();
      X = X2.toCharArray();
      lukija2 = new BufferedReader(new FileReader("G:\\Dropbox\\Gradu\\
            testitiedostot\\merkkijono" + lkm + "," + lkm2 +"testi" + kierros +
            "_2.txt"));
      Y2 = lukija2.readLine();
      Y = Y2.toCharArray();
      lukija2.close();
      printti.newLine();
      // Wagner-Fischer
      if(testattavat[0] == 1){
            System.gc(); System.gc();
            aika = System.currentTimeMillis();
            lcs = wf.lcslength(X, Y);
            aika = System.currentTimeMillis() - aika;
            currentMemory = (
                (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
                ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
            muisti = (int)currentMemory;
            printti.write(aika + ";" + muisti + ";");
            if(pya == -1){
                  pya = lcs.length();
            }else if(pya != lcs.length()){
                  onko = false;
            }
            System.gc(); System.gc();
            aika = System.currentTimeMillis();
            lcs2 = wf.lcslength(Y, X);
            aika = System.currentTimeMillis() - aika;
            currentMemory = (
                (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
                ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
            muisti = (int)currentMemory;
            printti.write(aika + ";" + muisti + ";");
            if(pya == -1){
                  pya = lcs2.length();
            }else if(pya == lcs2.length()){
                  printti.write("true;");
```

```
        }else{
                onko = false;
                printti.write("false;");
        }
}else{
        printti.write(";;;;;");
}
// Hirschberg lineaarinen
if(testattavat[1] == 1){
        System.gc(); System.gc();
        aika = System.currentTimeMillis();
        lcs = hbl.AlgC(X.length, Y.length, X2, Y2);
        aika = System.currentTimeMillis() - aika;
        currentMemory = (
            (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
            ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
        muisti = (int)currentMemory;
        printti.write(aika + ";" + muisti + ";");
        if(pya == -1){
                pya = lcs.length();
        }else if(pya != lcs.length()){
                onko = false;
        }
        System.gc(); System.gc();
        aika = System.currentTimeMillis();
        lcs2 = hbl.AlgC(Y.length, X.length, Y2, X2);
        aika = System.currentTimeMillis() - aika;
        currentMemory = (
            (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
            ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
        muisti = (int)currentMemory;
        printti.write(aika + ";" + muisti + ";");
        if(pya == -1){
                pya = lcs2.length();
        }else if(pya == lcs2.length()){
                        printti.write("true;");
        }else{
                        onko = false;
                        printti.write("false;");
        }
}else{
        printti.write(";;;;;");
}
// Hirschberg alkuperäinen
if(testattavat[2] == 1){
        System.gc(); System.gc();
        aika = System.currentTimeMillis();
        lcs = hb.lcslength(X, Y);
        aika = System.currentTimeMillis() - aika;
        currentMemory = (
            (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
            ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
```

```
        muisti = (int)currentMemory;
        printti.write(aika + ";" + muisti + ";");
        if(pya == -1){
            pya = lcs.length();
        }else if(pya != lcs.length()){
            onko = false;
        }
        System.gc(); System.gc();
        aika = System.currentTimeMillis();
        lcs2 = hb.lcslength(Y, X);
        aika = System.currentTimeMillis() - aika;
        currentMemory = (
            (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
            ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
        muisti = (int)currentMemory;
        printti.write(aika + ";" + muisti + ";");
        if(pya == -1){
            pya = lcs2.length();
        }else if(pya == lcs2.length()){
            printti.write("true;");
        }else{
            onko = false;
            printti.write("false;");
        }
    }else{
        printti.write(";;;;;");
    }
    // Hirschberg 2-suuntainen
    if(testattavat[3] == 1){
        System.gc(); System.gc();
        aika = System.currentTimeMillis();
        lcs = hb2.hb2w(X, Y);
        aika = System.currentTimeMillis() - aika;
        currentMemory = (
            (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
            ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
        muisti = (int)currentMemory;
        printti.write(aika + ";" + muisti + ";");
        if(pya == -1){
            pya = lcs.length();
        }else if(pya != lcs.length()){
            onko = false;
        }
        System.gc(); System.gc();
        aika = System.currentTimeMillis();
        lcs2 = hb2.hb2w(Y, X);
        aika = System.currentTimeMillis() - aika;
        currentMemory = (
            (double)((double)(Runtime.getRuntime().totalMemory()/1024)/1024))-
            ((double)((double)(Runtime.getRuntime().freeMemory()/1024)/1024));
        muisti = (int)currentMemory;
        printti.write(aika + ";" + muisti + ";");
```

```
                    if(pya == -1){
                        pya = lcs2.length();
                    }else if(pya == lcs2.length()){
                            printti.write("true;" + lcs.length());
                    }else{
                        onko = false;
                        printti.write("false;" + lcs.length());
                    }
                }else{
                    printti.write(";;;;;" + lcs.length());
                }
                kierros++;
            }
            if(onko){
                printti.newLine();
                printti.newLine();
                printti.close();
                System.out.println(lkm + "," + lkm2 + " valmis");
            }else{
                System.out.println("Virhe: Jokin löysi väärän pituisen PYAn");
            }
        }
}
```

Testitiedostot luova algoritmi:

```
import java.io.*;
import java.lang.Math.*;
import java.util.Random;

public class randomgenerator {
        /**
         * Luo halutun nimisen testitiedoston, joka sisältää satunnaisia merkkejä.
         * Alkuehto: tiedostonimi (+ sijainti) annettu oikein, lkm > 0, merkit != null.
         * Loppuehto: taulukossa lkm kpl satunnaisia merkkejä annetusta merkistöstä.
         */
        public static void luoTaulukkoTiedostoon(String tiedosto, int lkm, String merkit)
                        throws Exception{
            String tulos = "";
            int pituus = merkit.length();
            Random r = new Random();
            for (int i = 0; i < lkm; i++){
                int a = r.nextInt(pituus);
                tulos = tulos + merkit.substring(a, a+ 1);
            }
            PrintWriter kirj = new PrintWriter(new FileWriter(tiedosto));
            kirj.println(tulos);
            kirj.close();
        }
}
```