Merkkijonotäsmäyksestä

Heikki Hyyrö

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Pro gradu -tutkielma
Kesäkuu 2000

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Heikki Hyyrö: Merkkijonotäsmäyksestä
Pro gradu -tutkielma, 159 sivua

Kesäkuu 2000

Tarkastelen tässä tutkielmassa muutamia tyypillisiä merkkijonotäsmäyksessä käytettyjä algoritmeja ja periaatteita.

Aluksi käsittelen eksaktia, ja sitten likimääräistä merkkijonotäsmäystä, edeten periaatteiden ja niiden toimivuuden tarkastelun pohjalta koodiesityksiin. Tämä osa sisältää esimerkiksi Boyer-Moore-algoritmin yhteydessä käytettävän ns. $delta_1$-funktion (tässä esityksessä käytetään nimitystä siirtymäfunktio f_t) alustusalgoritmin toimintaperiaatteen käsittelyn, jollaista ei tietääkseni kirjallisuudessa ole aiemmin esitetty.

Lopuksi esittelen hieman merkkijonotäsmäystä käytännössä sovellettuna DNA-sekvensseihin. Tässä osuudessa esitän muutamien koeajojen tuloksia, joiden tavoitteena oli etsiä nopeita menetelmiä laajamittaisessa aineistossa tapahtuvan kaikki-kaikkia-vastaan-tyyppisen eksaktin ja likimääräisen merkkijonovertailun suorittamiseksi. Näiden koeajojen päätehtävä tämän tekstin osalta oli toimia samalla esimerkkinä alkuosassa käsiteltyjen algoritmien käytännön tehokkuudesta, vaikka samalla esitänkin hieman yleisluontoisempiakin päätelmiä saavutettujen tulosten suhteen.

Avainsanat ja -sanonnat: Eksakti ja likimääräinen merkkijonotäsmäys, merkkijonohaku, editointietäisyys, DNA-sekvenssien vertailu.

Johdanto

Käsittelen tässä työssä merkkijonojen täsmäystä. Työ jakautuu kolmeen osaan seuraavasti:

Ensimmäisessä osassa käsitellään yleisimpiä annetun hahmo-merkkijonon eksaktien esiintymien etsimisessä käytettyjä algoritmeja. Tällä tarkoitetaan, että etsitään tekstistä hahmon kanssa täysin identtisiä tekstin osia, ts. sellaisia tekstin indeksejä *j*, joilla pätee täsmäävyys *teksti*[*j*…*j*+*m*-1] = *hahmo*[0…*m*-1], missä *m* on hahmon pituus. Tällainen etsintä on tuttu esimerkiksi tekstinkäsittelyohjelmista, joissa dokumentista voi halutessaan etsiä jonkin haluamansa sanan. Esityksessä noudatetaan koko ajan edellä ollutta tapaa, eli hahmoon ja tekstiin viitataan nimikkeiden *hahmo* ja *teksti* kautta taulukkomaisesti samaan tapaan kuin esimerkiksi C-kielessä, eli merkkien indeksointi alkaa nollasta.

Toisessa osassa käsitellään likimääräiseen merkkijonotäsmäykseen liittyviä perusmenetelmiä, joiden avulla voidaan etsiä hahmo-merkkijonon likimääräisiä esiintymiä tekstistä. Likimääräinen esiintymä eroaa ensimmäisessä kappaleessa käsitellystä eksaktista esiintymästä siten, että nyt sallitaan myös virheitä, ts. myös hahmosta poikkeavat, mutta kuitenkin tarpeeksi paljon hahmon kaltaiset merkkijonot lasketaan esiintymiksi. Edellä esiintynyt käsite ”tarpeeksi paljon hahmon kaltainen” vaatii luonnollisesti jonkin mittapuun, jonka mukaan voidaan päättää, kuinka lähellä toisiaan kaksi eri merkkijonoa ovat. Tässä käsittelyssä mittana käytetään ns. Levenshteinin editointietäisyyden käsitettä [Levenshtein, 1966], ja oletuksena on, että aina ennen itse etsintävaihetta tiedetään, kuinka suuri virhe likimääräiselle merkkijonolle sallitaan. Editointietäisyyttä tarkastellaan toisen osan alussa melko perusteellisesti. Likimääräinen haku soveltuu luonnollisesti aivan tavalliseen dokumentissa suoritettavaan hakuun, missä sallitut virheet vastaavat mahdollisia kirjoitusvirheitä. Lisäksi on sellaisia sovellusaloja, joiden perusluonteen kannalta likimääräinen täsmääminen on hyvin oleellista. Näistä mahdollisesti tärkein on erilaisten mikrobiologisten datasekvenssien käsittely, joiden yhteydessä pitää ottaa huomioon sekvensseille ominainen pieni vaihtelu (esim. mutaatiot), ja editointietäisyys soveltuu tähän melko hyvin [Navarro, 1998]. Toisessa osassa tehtävä tarkastelu perustuu kokonaisuudessaan klassiseen editointietäisyyden yhteydessä esiintyvään lähestymistapaan, joka on taulukoinnin käyttö.

Kolmannessa osassa tarkastellaan hieman yhtä käytännön kannalta merkittävimmistä ja vaativimmista sovelluskohteista merkkijonotäsmäysalgoritmeille, DNA-sekvenssien käsittelyä. Tämä tapahtuu esittelemällä projektia, joka sisälsi sekä eksaktien että

likimääräisten merkkijonotäsmäysalgoritmien soveltamista 8,7 miljoonaa merkkiä laajaan leipurin hiivan (Saccharomyces cerevisiae) DNA-aineistoon, tavoitteena etsiä aineiston kannalta sopivalla tavalla ainutlaatuisia tietynpituisia osamerkkijonoja.

Yleisesti ensimmäisen ja toisen osan tavoitteena on erilaisten merkkijonon etsinnässä käytettyjen perusperiaatteiden esittely ja analysointi. Pitäydyn näiden osalta pääasiassa algoritmien alkuperäisissä, toimintaideat varmaankin parhaiten esittelevissä versioissa, vaikka monia algoritmeja onkin kehitelty edelleen. Esitys on itse soveltavasti muotoilemaani seuraamatta siis tiukasti mitään lähdeteosta. Esimerkiksi käsittelyyn liittyvät todistukset ovat yleensä kokonaan itseni tekemiä, ja esittämäni algoritmien varsinaiset pseudokoodiesitykset rakentuvat käsittelyn pohjalta. Ne eivät siis ole suoranaisesti itse algoritmin alkuperäisten kehittäjien käsialaa, mutta koska niiden alkuperäinen toiminta-ajatus on luonnollisesti säilytetty, eivät nämä algoritmit juurikaan käytännössä poikkea ulkoasultaankaan kirjallisuudessa yleisesti esitetyistä versioista. Kaikki käsiteltävänä olevat algoritmit ovat sikäli käyttötilanteesta riippumattomia, ns. "on-line"-algoritmeja (esim. [Navarro, 1998]), että ne eivät esiprosessoi etsinnän kohteena olevaa tekstiä mitenkään eivätkä myöskään käytä hyväkseen mitään ennakkotietoja, kuten esimerkiksi tilastollista tietoa käytettävän aakkoston eri merkkien odotetusta jakaumasta tekstissä.

Kolmannen osan projektikuvaus on suppeahko, melko suoraviivainen projektin tavoitteiden, etenemisen ja tulosten esittely, sillä sen tarkoitus tämän tekstin suhteen on toimia lähinnä soveltavana esimerkkinä.

Muutama tekstissä merkkijonojen yhteydessä käytetty merkintä

- ε Tyhjä merkki, esim. ”aεb” = ”ab”.
- $\cup$ Katenointi, esim. ’a’$\cup$’b’ = ”ab”.
- $\subseteq$ Merkkijonon toiseen merkkijonoon sisältyminen, esim. ”bc” $\subseteq$ ”abcd”.

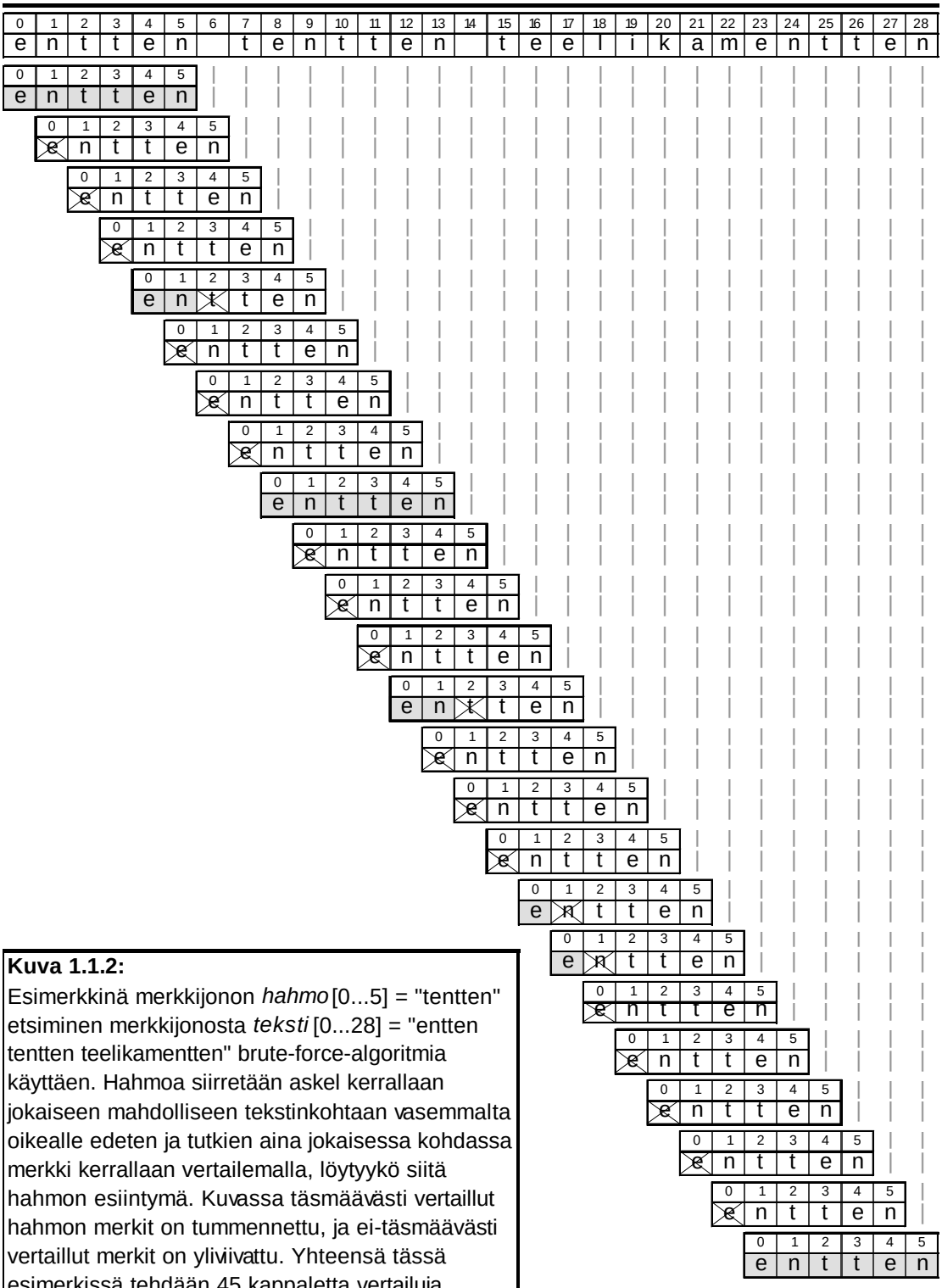
1. Eksakti merkkijonotäsmäys

1.1 Intuitiivinen, ns. ”brute force”-menetelmä

Tämä ensimmäinen käsiteltävä algoritmi kuuluu lähestymistavaltaan intuitiivisiin algoritmeihin, joista englanninkielisessä kirjallisuudessa käytetään mm. nimityksiä ”brute force” [Stephen, 1994] tai ”naive method” [Gusfield, 1997], sillä siinä hahmon etsiminen on toteutettu suoraviivaisesti kiinnittämättä juurikaan huomiota tehokkuuteen. Algoritmin toimintaperiaatteena on yksinkertaisesti verrata hahmoa jokaisen mahdollisen tekstinkohdan kanssa siirtämällä hahmoa tekstiin nähden merkki kerrallaan vasemmalta oikealle siten, että jokaisessa kohdassa tutkitaan aina merkki merkiltä vasemmalta oikealle vertailemalla, onko kyseessä hahmon esiintymä. Kuvassa 1.1.2 on esimerkki tästä, ja esitän saman esimerkin jatkossa myös muita tarkastelemiani merkkijonotäsmäysalgoritmeja soveltaen. Intuitiivisen algoritmin toteutus on siis varsin suoraviivainen (algoritmi 1.1.1).

```
int m, n;                                   // Hahmon pituus, tekstin pituus.
int i, j;                                   // Hahmoa ja tekstiä läpikäyvät indeksimuuttujat.
for(j = 0; j < n – m + 1; j++)              // Silmukka tutkii kaikki mahdolliset hahmon
{                                           // esiintymäkohdat tekstissä.
    i = 0;
    while(teksti[j+i] == hahmo[i] && i < m) // Verrataan hahmoa tekstiin.
    {
        i++;
    }
    if(i == m)                              // Löytyikö koko hahmo?
    {
        loytyi();                   // Hahmon löytymistä seuraavat toimenpiteet sovelluksesta
    }                               // riippuen.
}
```

Algoritmi 1.1.1:
Intuitiivinen merkkijonotäsmäysalgoritmi.



Kuva 1.1.2:
Esimerkkinä merkkijonon *hahmo*[0...5] = "tentten" etsiminen merkkijonosta *teksti*[0...28] = "entten tentten teelikamentten" brute-force-algoritmia käyttäen. Hahmoa siirretään askel kerrallaan jokaiseen mahdolliseen tekstinkohtaan vasemmalta oikealle edeten ja tutkien aina jokaisessa kohdassa merkki kerrallaan vertailemalla, löytyykö siitä hahmon esiintymä. Kuvassa täsmäävästi vertaillut hahmon merkit on tummennettu, ja ei-täsmäävästi vertaillut merkit on yliviivattu. Yhteensä tässä esimerkissä tehdään 45 kappaletta vertailuja.

Merkkijonoalgoritmien aikavaativuutta analysoitaessa kirjallisuudessa esiintyy hyvin usein seuraava pahimman tapauksen esimerkki:

hahmo = ”aa…aab”, pituus *m* merkkiä,
teksti = ”aaaa…aaaa”, pituus *n* merkkiä, $n \geq m$.

Nyt jokaisella pääsilmukan kierroksella sisäsilmukassa tehdään aluksi $m - 1$ täsmäävää kahden ’a’-kirjaimen välistä vertailua, minkä jälkeen vuorossa oleva kirjainten *hahmo*[*m*-1] = ’b’ ja *teksti*[*j*] = ’a’ välinen vertailu ei täsmää, ja siirrytään askel eteenpäin tutkimaan seuraavaa tekstin kohtaa. Kaikkiaan vertailuja tulee siis *m* kappaletta kierrosta kohti, ja koska pääsilmukka suoritetaan $n - m + 1$ kertaa, saadaan yhteismääräksi $(n - m + 1) \times m = n \times m - m \times m + m$ vertailua, eli aikavaativuus on $O(n \times m)$.

1.2 Morris-Pratt ja Knuth-Morris-Pratt

Intuitiivinen merkkijonohakualgoritmi voi pahimmillaan verrata yhtä tekstin merkkiä jopa kaikkia hahmon merkkejä vastaan, mikä viittaa siihen, että algoritmissa olisi tehostamisen varaa. Yhden klassisen menetelmän etsinnän nopeuttamiseksi ovat esittäneet Morris ja Pratt [Morris and Pratt, 1970].

Lähtökohtana on tutkia hahmon rakennetta ennen itse hakuprosessia siten, että saatujen tietojen pohjalta olisi ei-täsmäävän vertailun sattuessa mahdollista siirtää hahmoa tekstiin nähden enemmän kuin yksi merkki eteenpäin. Tämä tehdään muodostamalla ns. korjausfunktio, joka kertoo hahmon ei-täsmänneen merkin indeksistä riippuen, mitä hahmon merkkiä seuraavaksi verrataan, kun hahmoa siirretään tekstissä eteenpäin. Käytetään tässä korjausfunktiosta merkintää f_k.

Tarkastellaan intuitiivisen algoritmin tilannetta, jossa ei-täsmäävä vertailu tapahtuu merkkien *teksti*[*s*] ja *hahmo*[*q*] välillä ja $q > 0$. Tällöin hahmo on tekstiin nähden kohdassa *s-q*, ja merkkijonot *teksti*[*s-q*…*s*-1] ja *hahmo*[0…*q*-1] täsmäävät keskenään. Oletetaan, että tekstissä on hahmon esiintymä kohdassa (*s*– *q*) + *r*, missä $r > 0$, jolloin siis *teksti*[(*s-q*)+*r*…*s-q*+*r*+*m*-1] = *hahmo*[0…*m*-1]. Nyt jos $r < q$ eli tämä esiintymä alkaa juuri täsmäävästi vertaillun osan *teksti*[*s-q*…*s*-1] sisältä, niin pätee $0 < q\text{-}r < q$ ja *teksti*[*s-q*+*r*…*s*-1] = *hahmo*[0...*q-r*-1]. Yhdistämällä tämä tiedon *teksti*[*s-q*…*s*-1] = *hahmo*[0…*q*-1] kanssa päädytään tulokseen *hahmo*[0…*q-r*-1] = *hahmo*[*q*-(*q-r*)…*q*-1]. Edellisten perusteella voidaan päätellä, että juuri täsmätyn osan *teksti*[*s-q*+1…*s*-1] alueelta kohdasta *s* – *j*, missä $0 < j < q$, voi alkaa hahmon esiintymä vain siinä tapauksessa, että *hahmo*[0…*j*-1] = *hahmo*[*q-j*…*q*-1] (kuva 1.2.1). Tässä

Kuva 1.2.1:
Jos merkkijonot *teksti* [*s -q* …*s* -1] ja *hahmo* [0…*q* -1] täsmäävät, niin selvästi merkin *teksti* [*s -j*], $0 < j < q$, kohdalta voi alkaa hahmon esiintymä vain jos pätee *hahmo* [*q -j* …*q* -1] = *hahmo* [0…*j* -1]. Kuvassa täsmäävät osat on merkitty siten, että niistä tämä hahmon "sisäisen" täsmäävyyden osuus on vaaleamman sävyinen. Näin ollen hahmo voidaan turvallisesti siirtää seuraavan sellaisen merkin *teksti* [*s -j*] kohdalle, että pätee *teksti* [*s -j* …*s* -1] = *hahmo* [*q -j* …*q* -1] = *hahmo* [0…*j* -1]. Jos yhtään tällaista merkkiä ei ole olemassa, niin hahmo voidaan siirtää kokonaan merkin *teksti* [*s* -1] ohi. Ensimmäisessä tapauksessa on huomattava, että jo täsmääväksi tiedettyä osaa *teksti* [*s -j* …*s* -1] = *hahmo* [0…*j* -1] ei luonnollisesti enää tarvitse vertailla uudelleen, vaan silloin voidaan jatkaa merkkien *hahmo* [*j*] ja *teksti* [*s*] välisellä vertailulla.

käytetään yksinkertaisuuden vuoksi merkintää $j = q - r$.

Morris-Pratt-algoritmin korjausfunktion arvo $f_k(q)$ määräytyy edellä havaitun seikan pohjalta: Jos ei-täsmäävä vertailu tapahtuu merkin *hahmo*[q] kohdalla, voidaan hahmo siirtää tekstissä pienimmän sellaisen indeksin $s - j$ kohdalle, mikäli tällainen indeksi $s - j$ ylipäätään on olemassa, että ehdot $s\text{-}q < s\text{-}j \leq s\text{-}1$ ja *hahmo*[$0 \ldots j\text{-}1$] = *hahmo*[$q\text{-}j \ldots q\text{-}1$] ovat voimassa. Jos tällainen indeksi $s - j$ löytyy, tiedetään, että osat *teksti*[$s\text{-}j \ldots s\text{-}1$] ja *hahmo*[$0 \ldots j\text{-}1$] täsmäävät, ja siksi niitä ei enää tarvitse vertailla uudelleen, vaan vertailuja voidaan jatkaa eteenpäin merkeistä *teksti*[s] ja *hahmo*[j] alkaen. Näin ollen tässä tapauksessa on $f_k(q) = j$. Koska indeksin $s - j$ vaatimuksena on minimaalisuus, tiedetään että joko $s - j = s - q + 1$ ja tehty siirtymä on ainoastaan yhden askeleen pituinen, tai sitten alueelta *teksti*[$s\text{-}q+1 \ldots s\text{-}j\text{-}1$] ei ala mikään hahmon esiintymä. Näin ollen tämä siirtymä ei hyppää yhdenkään hahmon esiintymän yli. Jos vaaditunlaista indeksiä $s - j$ taas ei ole olemassa, tiedetään että hahmo voidaan siirtää kokonaan osan *teksti*[$s\text{-}q+1 \ldots s\text{-}1$] ohi, sillä mikään hahmon esiintymä ei ala tältä alueelta. Tällöin hahmo siirretään siis suoraan kohtaan s hyppäämättä yhdenkään hahmon esiintymän yli. Koska vertailut aloitetaan taas hahmon suhteen alusta, vertaillaan tällöin seuraavaksi merkkejä *teksti*[s] ja *hahmo*[0], ja siten nyt $f_k(q) = 0$.

Korjausfunktiota käytetään myös koko hahmon löytymistä seuraavassa siirtymässä, ja tätä tarkoitusta varten myös arvo $f_k(m)$ on määritelty, vaikka merkkiä *hahmo*[m] ei olekaan olemassa. Myös arvo $f_k(m)$ määräytyy edellä käsitellyllä tavalla, mutta sitä sovellettaessa on huomattava, että tällöin pitää myös tekstissä edetä yksi askel. Eli jos on löydetty hahmon esiintymä, jonka viimeinen merkki on *teksti*[s] (siis *hahmo*[$0 \ldots m\text{-}1$] = *teksti*[$s\text{-}m+1 \ldots s$]), tulee seuraava vertailu tehdä merkkien *hahmo*[$f_k(m)$] ja *teksti*[$s+1$] välillä. Tämä vastaa täysin tilannetta, missä hahmon pituus olisikin suurempi kuin m, ja olisi tehty ei-täsmäävä vertailu merkkien *teksti*[$s+1$] ja *hahmo*[m] välillä.

Hahmon ensimmäinen merkki aiheuttaa erikoisarvon korjausfunktiolle. Nimittäin jos ei-täsmäävä vertailu tapahtuu merkkien *hahmo*[0] ja *teksti*[s] välillä ei korjausfunktiosta ole apua, ja hahmoa tulee siirtää "brute force"-algoritmin tapaan tekstissä yksi askel eteenpäin ja aloittaa vertailut taas hahmon suhteen alusta, jolloin seuraavaksi verrataan merkkejä *hahmo*[0] ja *teksti*[$s+1$]. Tätä tilannetta vastaa korjausfunktion arvo $f_k(0) = -1$.

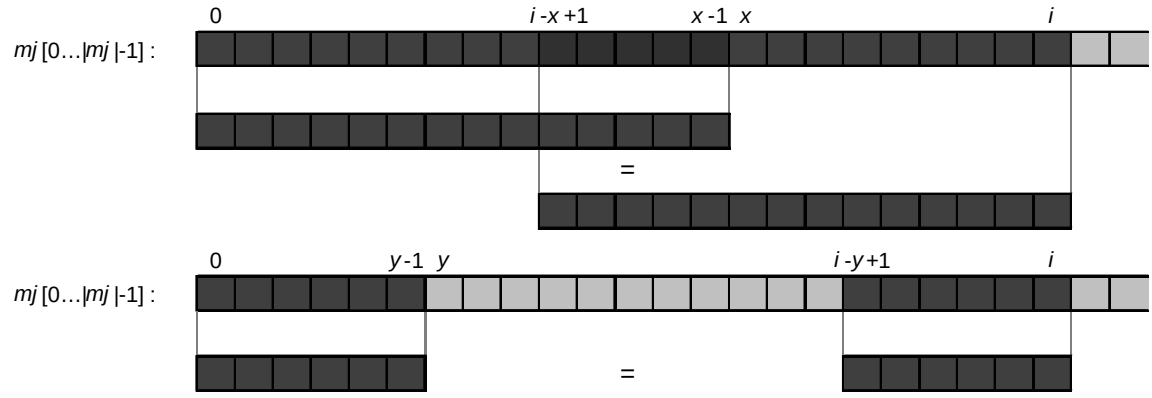
Koska indeksin $s - j$ arvon määräävät ehdot riippuvat vain luvun j arvosta, saa luku $s - j$ minimiarvonsa täsmälleen silloin, kun luku j saa maksimiarvonsa. Tämä huomioon ottaen

saadaan edellisen käsittelyn pohjalta korjausfunktion määritelmä muotoon $f_k(0) = -1$, ja $f_k(i) = \max\{k \mid (hahmo[i-k\ldots i-1] = hahmo[0...k-1] \wedge 0 < k < i) \vee (k = 0)\}$, kun $1 \leq i \leq m$.

Otetaan tarkastelun apuvälineeksi seuraava merkintä:

Määritelmä 1.2.2:
Olkoon *mj* jokin merkkijono ja $0 \leq i < |mj|$. Tällöin $R_{mj,\,i}$ on sellainen kokonaislukujen joukko, että luku *x* kuuluu joukkoon $R_{mj,\,i}$ jos ja vain jos joko $0 < x \leq i$ ja $mj[0\ldots x-1] = mj[i-x+1\ldots i]$, tai $x = 0$.

Joukko $R_{mj,\,i}$ koostuu siis kaikkien sellaisten merkkijonoa $mj[0\ldots i]$ lyhyempien merkkijonojen pituuksista (tyhjä merkkijono mukaan lukien), että kyseinen merkkijono on samalla kertaa sekä alku- että loppuosa merkkijonolle $mj[0\ldots i]$ (kuva 1.2.3). Tämäntyyppisistä merkkijonoista on käytetty nimitystä ”reuna” tai ”raja” (”border” [Crochemore and Rytter, 1994]), sillä niiden voidaan ajatella ”rajaavan” merkkijonoa $mj[0\ldots i]$ molemmilta puolilta.



Kuva 1.2.3:
Kaksi joukkoon $R_{mj,\,i}$ kuuluvaa lukua, *x* ja *y*, joilla siis pätee $hahmo[0\ldots x-1] = hahmo[i-x+1\ldots i]$ ja $hahmo[0\ldots y-1] = hahmo[i-y+1\ldots i]$.

Esitetään nyt aiemman pohjalta korjausfunktion f_k täsmällinen määritelmä käyttäen edellistä merkintää:

Määritelmä 1.2.4:
Morris-Pratt-algoritmin korjausfunktio f_k:
$f_k(0) = -1$,
$f_k(i) = \max\{k \mid k \in R_{hahmo,\,i-1}\}$, kun $1 \leq i \leq m$.

Määritelmän 1.2.4 sanallinen vastine on ”$f_k(0)$ on –1, ja kun $1 \leq i \leq m$ on arvo $f_k(i)$ yhtä kuin pisimmän sellaisen merkkijonoa *hahmo*[0…*i*-1] lyhyemmän merkkijonon pituus, että kyseinen merkkijono on samalla kertaa sekä alku- että loppuosa merkkijonolle *hahmo*[0…*i*-1].” Näin ollen seuraava lause on ilmeinen:

Lause 1.2.5:
Ehto $f_k(i) < i$ on voimassa aina kun $0 \leq i \leq m$, ja lisäksi ehto $f_k(i) \geq 0$ on voimassa kun $1 \leq i \leq m$.

Todistus:
Määritelmän 1.2.4 mukaan $f_k(i) \in R_{hahmo,i-1}$, kun $1 \leq i \leq m$, ja määritelmästä 1.2.2 nähdään, että tällöin $0 \leq f_k(i) \leq i - 1$ eli $0 \leq f_k(i) < i$. Koska lisäksi $f_k(0) = -1 < i$, on lauseen väite tosi.

Kun korjausfunktion arvot tunnetaan, voidaan Morris-Pratt-etsintäalgoritmi itsessään toteuttaa algoritmin 1.2.6 mukaisesti.

```
int m;                                          // Hahmon pituus
int n;                                          // Tekstin pituus
int f[m];                                       // Korjausfunktion arvotaulukko.
int i = 0;          // Vertailuvuorossa olevan hahmon merkin indeksi, siis alussa i  = 0.
int j = 0;          // Vertailuvuorossa olevan tekstin merkin indeksi, siis alussa j  = 0.
while(j – i  < n – m + 1)        // Käydään koko teksti läpi, j – i = hahmon paikka tekstissä.
{
    while(i < m && teksti[j] == hahmo[i])       // Tekstiä ja hahmoa täsmäävä silmukka.
    {
        i++;                                    // Edetään yksi merkki hahmossa.
        j++;                                    // Edetään yksi merkki tekstissä.
    }
    if(i == m)                                          // Löytyikö hahmo?
    {
        loytyi();                               // Löytymistä seuraavat toimenpiteet.
    }
    i = f[i];                                   // Joko hahmo löytyi tai merkki
                                                // hahmo[i] ei täsmännyt. Siis joka
                                                // tapauksessa sovelletaan arvoa f[i].
    if(i == -1)                                 // Tarkistetaan saiko i erikoisarvon –1,
    {                                           // ja jos sai niin nollataan hahmon
        i = 0;                                  // vertailukohta ja edetään yksi
        j++;                                    // merkki tekstissä.
    }
}
```

Algoritmi 1.2.6:
Morris-Pratt-merkkijonohaku.

Useimmiten Morris-Pratt-algoritmista on esitetty sellainen toteutus, jossa sisä- ja ulkosilmukoiden roolit täsmäyksen kannalta on käännetty päinvastaisiksi. Tällöin sisäsilmukka huolehtii ei-täsmäävien vertailujen yhteydessä tehtävistä siirtymistä, ja jokainen ulkosilmukan suoritus vastaa yhtä täsmäävää vertailua. Näin toimittaessa erikoisarvoa $i = -1$ ei tarvitse enää käsitellä erikseen, ja algoritmista tulee lyhyempi (algoritmi 1.2.7).

```
int m;                                          // Hahmon pituus.
int n;                                          // Tekstin pituus.
int f[m];                                       // Korjausfunktion arvotaulukko.
int i = 0;                          // Vertailuvuorossa olevan hahmon merkin indeksi, siis alussa i  = 0.
int j = 0;                          // Vertailuvuorossa olevan tekstin merkin indeksi, siis alussa j  = 0.
while(j – i  < n – m + 1)           // Käydään koko teksti läpi, j – i = hahmon paikka tekstissä.
{
    while(i >= 0 && teksti[j] != hahmo[i])      // While-silmukan ehto tutkii onko
    {                                           // kyseessä merkin i kohdalla tapahtuva
        i = f[i];                               // ei-täsmäävä vertailu, joka ei johda
    }                                           // tekstin vertailukohdan siirtoon.
    i++;                                        // Edetään yksi merkki hahmossa.
    j++;                                        // Edetään yksi merkki tekstissä.
    if(i == m)                                  // Löytyikö hahmo?
    {
        loytyi();                               // Löytymistä seuraavat toimenpiteet,
        i = f[m];                               // nyt sovelletaan arvoa f[m].
    }
}
```

Algoritmi 1.2.7:
Morris-Pratt-merkkijonohaku, lyhyempi versio.

Intuitiivisesti tuntuu luonnolliselta, että Morris-Pratt-algoritmin tulee tunnistaa aina tietyllä tapaa maksimaaliset hahmon alkuosat tekstistä, sillä jos näin ei olisi, voisi jokin hahmon esiintymä jäädä huomaamatta. Lauseissa 1.2.8 - 1.2.11 tämä ajatus puetaan tarkempaan muotoon, ja Morris-Pratt-algoritmin toimivuus todistetaan niiden pohjalta vielä täsmällisesti lauseissa 1.2.12 ja 1.2.13.

Seuraavissa tarkasteluissa oletetaan, että luvuilla i ja j on samat roolit kuin algoritmeissa 1.2.6 ja 1.2.7: i kertoo hahmon osalta vertailuvuorossa olevan merkin indeksin ja j vastaavasti tekstin osalta vertailuvuorossa olevan merkin indeksin.

Lause 1.2.8:
Morris-Pratt-algoritmin verratessa merkkejä *hahmo*[i] ja *teksti*[j] pätee aina, että joko $i = 0$, tai $i > 0$ ja *hahmo*[$0 \ldots i\text{-}1$] = *teksti*[$j\text{-}i \ldots j\text{-}1$].

Todistus:
Algoritmin alussa $i = 0$, joten silloin lauseen väite on voimassa. Tarkastellaan nyt tilannetta, jossa i ja/tai j saa uuden arvon. Käytetään lukujen i ja j vanhoista arvoista merkintöjä i_v ja j_v ja vastaavasti uusista arvoista merkintöjä i_u ja j_u sekä oletetaan, että lauseen väite on tosi arvoilla i_v ja j_v. Kun nyt verrataan merkkejä *hahmo*[i_v] ja *teksti*[j_v], määräytyvät uudet arvot i_u ja j_u seuraavien vaihtoehtojen 1), 2) tai 3) mukaan:

1) *hahmo*[i_v] = *teksti*[j_v] ja $0 \leq i_v < m - 1$:
   Nyt molempia indeksejä kasvatetaan yhdellä eli $i_u = i_v + 1$ ja $j_u = j_v + 1$. Koska lisäksi oletuksen mukaan $i_v = 0$ tai *hahmo*[0…i_v-1] = *teksti*[j_v-i_v…j_v-1], pätee nyt *hahmo*[0…i_v] = *teksti*[j_v-i_v…j_v], joten *hahmo*[0…i_u-1] = *teksti*[j_u-i_u…j_u-1].

2) *hahmo*[i_v] = *teksti*[j_v] ja $i_v = m - 1$:
   Nyt asetetaan $i_u = f_k(m)$ ja $j_u = j_v + 1$. Samoin perustein kuin kohdassa 1) pätee myös nyt, että *hahmo*[0…i_v] = *teksti*[j_v-i_v…j_v] eli *hahmo*[0…m-1] = *teksti*[j_v-m-1…j_v]. Koska määritelmän 1.2.4 mukaan $i_u = f_k(m) \in R_{hahmo,\, m-1}$, pätee joko $i_u = 0$, tai *hahmo*[0…i_u-1] = *hahmo*[m-i_u…m-1]. Yhdistämällä edelliset täsmäävyydet nähdään, että joko $i_u = 0$ tai *hahmo*[0…i_u-1] = *hahmo*[m-i_u…m-1] = *teksti*[j_v-i_u+1…j_v], mikä saadaan yhtäsuuruuden $j_v + 1 = j_u$ perusteella muotoon $i_u = 0$ tai *hahmo*[0…i_u-1] = *teksti*[j_u-i_u…j_u-1].

3) *hahmo*[i_v] ≠ *teksti*[j_v]:
   Nyt asetetaan $i_u = f_k(i_v)$. Jos $i_u = -1$, asetetaan ennen seuraavaa merkkien vertailua vielä $i_u = i_u + 1 = 0$, joten lauseen väite on voimassa. Sama pätee tietenkin myös tilanteessa, jossa heti $i_u = 0$. Tarkastellaan siis tilannetta, jossa $i_u > 0$. Tällöin indeksin j arvo ei muutu eli $j_u = j_v$. Lisäksi lauseen 1.2.5 mukaan $i_v > i_u > 0$, joten alkuoletuksen perusteella *hahmo*[0…i_v-1] = *teksti*[j_v-i_v…j_v-1]. Määritelmistä 1.2.2 ja 1.2.4 nähdään, että *hahmo*[0…i_u-1] = *hahmo*[i_v-i_u…i_v-1], joten edelliset tiedot yhdistämällä saadaan *hahmo*[0…i_u-1] = *teksti*[j_u-i_u…j_u-1].

Jokainen kohdista 1), 2) ja 3) johti tilanteeseen, jossa $i_u = 0$ tai *hahmo*[0…i_u-1] = *teksti*[j_u-i_u…j_u-1], joten koska nämä kohdat kattavat kaikki mahdolliset tilanteet, voidaan lauseen 1.2.8 väite todeta induktioperiaatteen nojalla oikeaksi.

Lause 1.2.9:
Jos mj_1 ja mj_2 ovat mielivaltaisia merkkijonoja sekä s ja q sellaisia indeksejä, että $0 \leq s < |mj_1|$ ja $0 \leq q < |mj_2|$, niin seuraava ehto on voimassa:
$\max\{k \mid k = 0 \vee mj_1[s\ldots s+k\text{-}1] = mj_2[q\text{-}k+1\ldots q]\} + 1 \geq \max\{k \mid k = 0 \vee mj_1[s\ldots s+k\text{-}1] = mj_2[(q+1)\text{-}k+1\ldots q+1]\}$.

Todistus:
Olkoon $x = \max\{k \mid k = 0 \vee mj_1[s\ldots s+k\text{-}1] = mj_2[q\text{-}k+1\ldots q]\}$ ja $y = \max\{k \mid k = 0 \vee mj_1[s\ldots s+k\text{-}1] = mj_2[(q+1)\text{-}k+1\ldots q+1]\}$. Koska molempien maksimilausekkeiden arvot ovat ≥ 0, on väite $x + 1 \geq y$ selvästi tosi, kun $y \leq 1$. Tarkastellaan siis tapausta $y > 1$. Koska nyt $y - 2 \geq 0$ ja $mj_1[s\ldots s+y\text{-}1] = mj_2[(q+1)\text{-}y+1\ldots q+1]$, pätee tällöin myös $mj_1[s\ldots s+(y\text{-}1)\text{-}1] = mj_2[(q+1)\text{-}y+1\ldots q] = mj_2[q\text{-}(y\text{-}1)+1\ldots q]$. Koska x on maksimaalinen tämän jälkimmäisen ehdon mukainen luku, seuraa tästä että $x \geq y - 1$ eli $x + 1 \geq y$, joten lauseen väite on oikea.

Lause 1.2.10:
Ehto $i + 1 \geq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k+1\ldots j]\}$ on voimassa aina, kun Morris-Pratt-algoritmi on vertaamassa merkkejä $hahmo[i]$ ja $teksti[j]$.

Todistus:
Väite on selvästi voimassa Morris-Pratt-algoritmin alussa merkkien $hahmo[0]$ ja $teksti[0]$ kohdalla. Käytetään merkintöjä i_v, j_v, i_u ja j_u samaan tapaan kuin lauseen 1.2.8 todistuksessa. Oletetaan nyt, että lauseen väite oli voimassa merkkien $hahmo[i_v]$ ja $hahmo[j_v]$ välisen vertailun kohdalla ja että seuraavaksi vertaillaan merkkejä $hahmo[i_u]$ ja $hahmo[j_u]$. Jaetaan tarkastelu seuraaviin tapauksiin merkkien $hahmo[i_v]$ ja $teksti[j_v]$ välisen vertailun mukaan:

1) $hahmo[i_v] = teksti[j_v]$ ja $0 < i_v < m - 1$:
   Tässä tapauksessa $i_u = i_v + 1$, $j_u = i_u + 1$ sekä oletuksen mukaan $i_v + 1 \geq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k+1\ldots j_v]\}$. Soveltamalla lausetta 1.2.9 sijoituksilla $mj_1 = hahmo$, mj_2 = teksti, $s = 0$ ja $q = j_v$ saadaan $i_u + 1 = (i_v + 1) + 1 \geq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k+1\ldots j_v]\} + 1 \geq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k+1\ldots j_u]\}$. Siis lauseen väite on tällöin voimassa.

2) $hahmo[i_v] = teksti[j_v]$, ja $i_v = m - 1$ tai $hahmo[i_v] \neq teksti[j_v]$ ja $0 < i_v \leq m - 1$:
   Nyt $i_u = f_k(x)$, $hahmo[0\ldots x\text{-}1] = teksti[j_u\text{-}x\ldots j_u\text{-}1]$, missä $x = i_v + 1 = m$ tai $x = i_v$

tapauksesta riippuen ja $j_u = j_v + x - i_v$. Joka tapauksessa $x > 0$, joten määritelmien 1.2.2 ja 1.2.4 perusteella tiedetään, että $f_k(x) = \max\{k \mid k \in R_{hahmo,x-1}\} = \max\{k \mid k = 0 \vee (k \leq x-1 \wedge hahmo[0\ldots k\text{-}1] = hahmo[x\text{-}k\ldots x\text{-}1])\}$. Koska lisäksi $hahmo[0\ldots x\text{-}1] = teksti[j_u\text{-}x\ldots j_u\text{-}1]$, voidaan edellinen kirjoittaa muodossa $f_k(x) = \max\{k \mid k = 0 \vee (k \leq x\text{-}1 \wedge hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\ldots j_u\text{-}1])\}$. Toisaalta joko $x = i_v + 1$ tai $x = i_v$ ja $hahmo[i_v] \neq teksti[j_v]$, mistä seuraa, että $x \geq \max\{k \mid hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\}$. Koska lisäksi $f_k(x) < x$, voidaan ehto $k \leq x\text{-}1$ poistaa tässä tapauksessa edellisestä ja kirjoittaa $f_k(x) = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\ldots j_u\text{-}1]\}$. Soveltamalla nyt lausetta 1.2.9 sijoituksilla $mj_1 = hahmo$, mj_2 = teksti, $s = 0$ ja $q = j_u$ saadaan $i_u + 1 = f_k(x) + 1 = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\ldots j_u\text{-}1]\} + 1 \geq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$. Siis lauseen väite on voimassa myös tässä tapauksessa.

3) $i_v = 0$:
Nyt joko $i_u = i_v = 0$ (tapaus $hahmo[0] \neq teksti[j_v]$) tai $i_u = i_v + 1 = 1$ (tapaus $hahmo[0] = teksti[j_v]$). Kummassakin tapauksessa $j_u = j_v + 1$ ja oletuksen mukaan $i_v + 1 \geq \max\{k \mid hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\}$. Koska tapauksessa $i_u = i_v$, $hahmo[i_v] \neq teksti[j_v]$ tiedetään, että $i_v \geq \max\{k \mid hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\}$, on myös ehto $i_u \geq \max\{k \mid hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\}$ voimassa kummassakin tapauksessa. Tästä saadaan soveltamalla lausetta 1.2.9 sijoituksilla $mj_1 = hahmo$, $mj_2 = teksti$, $s = 0$ ja $q = j_u$ ehto $i_u + 1 \geq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\ldots j_v\text{-}1]\} + 1 \geq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$. Siis lauseen väite on voimassa myös tapauksessa 3).

Koska kohdat 1) - 3) kattavat kaikki eri mahdollisuudet, voidaan lauseen väite todeta induktioperiaatteen nojalla oikeaksi.

Lause 1.2.11:
Morris-Pratt-algoritmin verratessa merkkejä $hahmo[i]$ ja $teksti[j]$ pätee toinen seuraavista ehdoista a) ja b):
$hahmo[i] = teksti[j]$ ja $i + 1 = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$,
$hahmo[i] \neq teksti[j]$ ja $i + 1 > \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$.

Todistus:
Lauseen 1.2.10 mukaan ehto $i + 1 \geq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$ on joka tapauksessa voimassa. Nyt jos $hahmo[i] = hahmo[j]$, pätee tämän ja lauseen 1.2.8

nojalla myös $i + 1 \leq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$, joten tällöin $i + 1 = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$. Toisaalta jos $hahmo[i] \neq hahmo[j]$, tiedetään, että $i + 1 \neq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$, ja siten tässä tilanteessa pätee $i + 1 > \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$. Koska näin ollen toinen ehdoista a) ja b) on aina voimassa, on lauseen väite tosi.

Lause 1.2.12:
Morris-Pratt-algoritmi kirjaa löydetyksi merkkijonon *teksti*[0…*n*-1] kohtaan *s* päättyvän merkkijonon *hahmo*[0…*m*-1] esiintymän jos ja vain jos *teksti*[*s*-*m*+1…*s*] = *hahmo*[0…*m*-1].

Todistus:
Olkoon *s* mielivaltainen tekstin indeksi, jolloin $0 \leq s < n - m + 1$. Todistetaan väite kahdessa osassa:

1) Oletetaan, että Morris-Pratt-algoritmi kirjaa tekstin kohtaan *s* päättyvän hahmon esiintymän löydetyksi. Tämä tapahtuu ainoastaan sellaisessa tilanteessa, että viimeksi on verrattu merkkejä *hahmo*[*m*-1] ja *teksti*[*s*] täsmäävästi, ja tällöin pätee lauseen 1.2.11 mukaan $m = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[s\text{-}k\text{+}1\ldots s]\}$. Täten siis *teksti*[*s*-*m*+1…*s*] = *hahmo*[0…*m*-1], ja tämä ensimmäinen osa on todistettu.

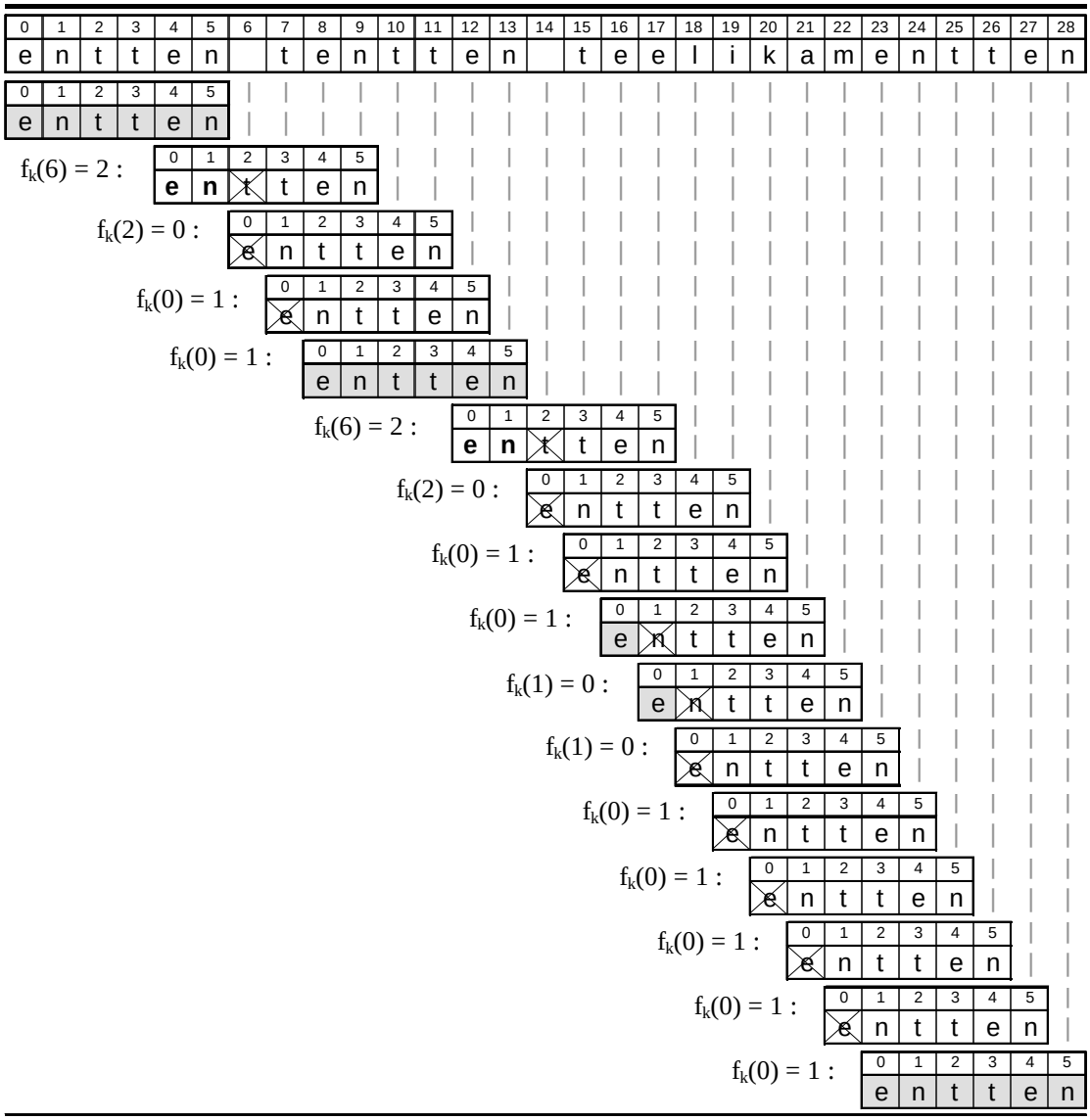
2) Oletetaan, että *teksti*[*s*-*m*+1…*s*] = *hahmo*[0…*m*-1]. Koska $s - m + 1 \leq n - m$ ja $i \geq 0$ aina Morris-Pratt-algoritmin lopetusehtoa tutkittaessa, pätee tällöin ehto $j - i < n - m + 1$ aina, kun $j \leq s - m + 1$. Toisaalta välillä $s - m + 1 < j \leq s$ tiedetään lauseen 1.2.10 sekä oletuksen *teksti*[*s*-*m*+1…*j*] = *hahmo*[0…*j*-*s*+*m*-1] pohjalta että $i + 1 \geq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\} \geq j - s + m$, joten myös tällöin ehto $j - i \leq s - m + 1 < n - m + 1$ on voimassa lopetusehdon tutkimishetkellä. Koska indeksin *j* arvoa kasvatetaan aina korkeintaan yhdellä, ei Morris-Pratt-algoritmi lopeta toimintaansa ilman, että indeksi *j* olisi saanut jossain vaiheessa arvon *s*. Tämän tapahtuessa pätee $i = m - 1$, sillä tällöin $i + 1 = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[s\text{-}k\text{+}1\ldots s]\} = m$. Koska nyt merkkien *teksti*[*s*] ja *hahmo*[*m*-1] välillä tehdään täsmäävä vertailu, kirjaa Morris-Pratt-algoritmi löydetyksi tekstin kohtaan *s* päättyvän hahmon esiintymän, ja tämä toinenkin osa on todistettu.

Lauseen väite seuraa yhdistämällä kohdat 1) ja 2).

Lause 1.2.13:
Morris-Pratt-algoritmin avulla voidaan etsiä kaikki hahmon esiintymät tekstistä lineaarisessa ajassa tekstin pituuteen nähden.

Todistus:
Tarkastellaan algoritmin 1.2.7 ulko- ja sisäsilmukoiden suorituskertojen lukumäärän ja indeksien i ja j välistä suhdetta. Alussa asetetaan $j = 0$, ja luvun j arvo ei ikinä pienene algoritmin suorituksen aikana. Toisaalta jokainen pääsilmukan kierros kasvattaa j:n arvoa yhdellä, ja pääsilmukan suorittaminen lopetetaan, kun $j - i < n - m + 1$. Tästä seuraa, että pääsilmukkaa voidaan suorittaa korkeintaan n kertaa, sillä koska aina $i \leq m$, pätee $j - i \geq n - m + 1$, kun $j \geq n$. Koska aina $f_k(i) < i$, vähentää jokainen sisäsilmukan kierros i:n arvoa vähintään yhdellä. Alussa asetetaan $i = 0$, ja ainoa tilanne, jossa i:n arvo kasvaa, on jokainen pääsilmukan suorituskerta, ja silloinkin i:n arvo kasvaa tasan yhdellä. Näin ollen sisäsilmukan suoritusehdosta $i \geq 0$ seuraa, että sisäsilmukkaa voidaan suorittaa yhteensä korkeintaan yhtä monta kertaa kuin pääsilmukkaa eli n kertaa, sillä jokaista indeksin i arvon pienentämistä kohti on täytynyt suorittaa vähintään yksi pääsilmukan kierros.
Näin ollen mitään algoritmin riviä ei suoriteta enempää kuin lineaarisesti tekstin pituuteen verrannollinen määrä, ja täten voidaan todeta että algoritmin aikavaativuus on $O(n)$.

Seuraavalla sivulla oleva kuva 1.2.14 vastaa kuvan 1.1.1 esimerkkiä käyttäen Morris-Pratt-algoritmia.



Kuva 1.2.14:
Esimerkkinä merkkijonon *hahmo*[0...5] = "tentten" etsiminen merkkijonosta *teksti*[0...28] = "entten tentten teelikamentten" Morris-Pratt-algoritmia käyttäen. Hahmoa siirretään joko seuraavaan sellaiseen kohtaan, jossa hahmon alkuosa täsmää jo täsmätyn tekstinosan kanssa, tai sitten kokonaan jo aiemmin täsmätyn osan ohi. Kuvassa täsmäävästi vertaillut hahmon merkit on tummennettu, ei-täsmäävästi vertaillut merkit on yliviivattu ja aiemmassa vaiheessa täsmätyt (ja siis uudelleen enää vertailemattomat) merkit on lihavoitu. Lisäksi jokaisen siirtymän yhteydessä on mainittu siinä sovellettu korjausfunktion arvo.Yhteensä tässä esimerkissä tehdään 33 kappaletta vertailuja.

Tähän astisessa käsittelyssä ei ole otettu vielä lainkaan huomioon korjausfunktion alustusta. Jotta Morris-Pratt-algoritmia voisi todella sanoa käytännössä lineaariseksi, pitää luonnollisesti myös sen korjausfunktio pystyä muodostamaan lineaarisessa ajassa. Seuraava tarkastelu näyttää, kuinka tämä voidaan tehdä, ja vieläpä sinänsä melko erikoisella tavalla, nimittäin soveltamalla Morris-Pratt-algoritmia itseään. Crochemore ja Rytter [Crochemore and Rytter, 1994] mainitsevat tämän ajatuksen ohimennen käsittelemättä asiaa sen tarkemmin, mutta ajatuksen soveltaminen on melko suoraviivaista aiemmin esitettyjen lauseiden pohjalta. Korjausfunktion alustaminen Morris-Pratt-algoritmilla voi nopeasti ajateltuna vaikuttaa kehäpäätelmältä, sillä algoritmihan nimenomaan pohjautuu korjausfunktion käyttöön. Mutta kuten seuraavassa käy ilmi, niin jokainen korjausfunktion arvo voidaan laskea aina ennen sen soveltamista, ja siten ei ajauduta kehäpäätelmään.

Lähdetään liikkeelle olettamalla yhtäsuuruus *teksti*[0…*m*-2] = *hahmo*[1…*m*-1], ja tarkastellaan tilannetta, jossa Morris-Pratt kasvattaa tekstin osalta vertailuvuorossa olevan merkin indeksiä j ja $j \leq m - 2$. Tällöin on täsmälleen kaksi vaihtoehtoa: Joko on tehty täsmäävä vertailu merkkien *hahmo*[i] ja *teksti*[j] välillä, tai $i = 0$ ja merkit *hahmo*[0] ja *teksti*[j] eivät täsmänneet keskenään. Ensimmäisen vaihtoehdon tilanteessa oletuksista $j \leq m - 2$ ja *teksti*[0…*m*-2] = *hahmo*[1…*m*-1] seuraa lauseen 1.2.11 kanssa, että $i + 1 = \max\{k \mid k = 0 \vee$ *hahmo*[0…*k*-1] = *teksti*[*j*-*k*+1…*j*]$\} = \max\{k \mid k = 0 \vee$ *hahmo*[0…*k*-1] = *hahmo*[*j*-*k*+2…*j*+1]$\} = \max\{k \mid k \in R_{hahmo,j+1}\} = f_k(j+2)$. Samalla tavalla saadaan jälkimmäisen vaihtoehdon tilanteessa lauseesta 1.2.11 ehto $i + 1 = 1 > f_k(j+2)$, ja koska $j + 2 > 0$, pätee lauseen 1.2.5 mukaan $f_k(j+2) \geq 0$, joten tässä tapauksessa on yhtäsuuruus $f_k(j+2) = 0$ voimassa.

Morris-Pratt-algoritmin alussa $j = 0$, ja j:n arvo muuttuu ainoastaan edellämainituissa tapauksissa, joissa sitä kasvatetaan tasan yhdellä, joten indeksi j käy algoritmin suorituksen aikana läpi kaikki arvot 0, 1, 2, …, m - 2, mikäli Morris-Pratt-algoritmin pysähtyessä pätee $j > m - 2$. Tämä jälkimmäinen seikka riippuu ainoastaan algoritmin suoritusehdosta $j - i < n - m + 1$, ja sen toteutuminen voidaan varmistaa korvaamalla alkuperäinen suoritusehto korjausfunktion laskemisen yhteydessä ehdolla $j \leq m - 2$. Tämä toimenpide ei aiheuta ongelmia, sillä oletuksen *teksti*[0…*m*-2] = *hahmo*[1…*m*-1] ansiosta $n - 1 \geq m - 2$, ja siten indeksi j pysyy tällöin aina tekstin sallimissa rajoissa, ja lisäksi korjausfunktion laskemisen kannalta ei olla kiinnostuneita siitä, mitä merkkejä tekstissä mahdollisesti vielä esiintyy merkin *teksti*[*m*-2] = *hahmo*[*m*-1] jälkeen.

Määritelmän 1.2.4 mukaan tiedetään suoraan, että $f_k(0) = -1$, mutta lauseen 1.2.5 perusteella huomataan lisäksi, että aina $0 \leq f_k(i) < 1$, eli myös arvo $f_k(1) = 0$ voidaan katsoa aina tunnetuksi. Koska merkkien *hahmo*[*i*] sekä *teksti*[*j*] kohdalla sovelletaan aina sellaista korjausfunktion arvoa $f_k(i)$, missä $i \leq j$, niin edellisen perusteella kaikki arvot $f_k(i)$ välillä $2 \leq i \leq m$ voidaan laskea ennen niiden käyttämistä soveltamalla Morris-Pratt-algoritmia siten, että ehto *teksti*[0…*m*-2] = *hahmo*[1…*m*-1] on voimassa ja algoritmin suoritus lopetetaan täsmälleen siinä tilanteessa, kun viimeinen arvo $f_k(m)$ on juuri laskettu (suoritusehto $j \leq m - 2$ takaa tämän).

Korjausfunktion arvot laskeva algoritmi saadaan edellisen käsittelyn nojalla muokattua esimerkiksi Morris-Pratt-etsintäalgoritmista 1.2.7 seuraavasti: Ensinnäkin koska *teksti*[0…*m*-2] = *hahmo*[1…*m*-1], viitataan indeksillä *j* tekstin sijaan hahmoon siten, että *j* on aina ”yhden askeleen edellä” (koska siis *teksti*[*j*] = *hahmo*[*j*+1] aina kun $0 < j \leq m - 2$). Käytännössä tämä toteutuu siten, että käytetään indeksin *j* sijaan indeksiä j^+, jolle pätee aina $j^+ = j + 1$, ja vertailut suoritetaan siis aina merkkien *hahmo*[j^+] sekä *hahmo*[*i*] välillä, ja lisäksi voidaan Morris-Pratt-algoritmin suoritusehto tällöin korvata aiemmin esitetyn ehdon $j \leq m - 2$ sijaan ehdolla $j^+ < m$. Korjausfunktion arvojen laskemiseksi lisätään alkuun suora sijoitus $f_k(1) = 0$, ja arvot $f_k(2),\ldots, f_k(m)$ lasketaan aina indeksin j^+ kasvattamisen yhteydessä. Koska $i = -1$ heti sen jälkeen kun on suoritettu ei-täsmäävä vertailu merkkien *hahmo*[0] ja *teksti*[*j*] = *hahmo*[j^+] välillä, niin myös tässä tilanteessa $f_k(j+2) = f_k(j^+1) = i + 1 = -1 + 1 = 0$, ja siten korjausfunktion arvon laskemiseksi riittää suorittaa aina indeksien j^+ ja *i* arvojen kasvattamisen jälkeen (molempia kasvatetaan aina samassa tilanteessa) sijoitus $f_k(j^+) = i$. Kun tämän lisäksi kokonaisen hahmon esiintymän löytymistä tutkiva if-lause poistetaan turhana, on korjausfunktion arvot laskeva algoritmi 1.2.15 valmis.

Yleensä Morris-Pratt-algoritmin alustusalgoritmissa ei tehdä suoraan sijoitusta $f_k(1) = 0$, vaan tämä hoidetaan esimerkiksi alustamalla muuttujat *i* ja j^+ yhtä pienemmillä arvoilla, eli $i = -1$ ja $j^+ = 0$, jolloin tämä asetus $f_k(1) = 0$ tulee hoidettua ensimmäisellä pääsilmukan kierroksella.

```
int m;                                          // Hahmon pituus.
int f[m];                                       // Korjausfunktion arvotaulukko, f[i] = fk(i).
f[0] = –1;          // Jos ei-täsmäävä vertailu heti hahmon ensimmäisen merkin kohdalla,
                    // tulee seuraava vertailu tehdä ”kohdasta –1”, joka tarkoittaa, että
                    // siirrytään tekstissä yksi merkki eteenpäin ja hahmossa ensimmäiseen
                    // merkkiin.
f[1] = 0;                                       // Etukäteen voidaan päätellä että fk(1) = 0.
int i = 0;                                      // Alustetaan hahmon indeksi arvolla i = 0.
int jplus = 1;          // Muuttuja jplus sisältää arvon j + 1, ja aina hahmo[jplus] = teksti[j].
while(jplus < m)        // Käydään kaikki korjausfunktion indeksit läpi, arvo fk(jplus+1)
{                       // lasketaan jplus:nnella kierroksella.
    while(i >= 0 && hahmo[jplus] != hahmo[i])   // While-silmukan ehto tutkii, onko
    {                                           // kyseessä merkin hahmo[i]
        i = f[i];                               // kohdalla tapahtuva ei-täsmäävä
    }                                           // vertailu, joka ei johda etenemiseen
                                                // merkkijonossa hahmo[1…m-1].
    i++;                // Hahmossa joko edetään askel tai pysytään kohdassa 0 (jos i = -1).
    jplus++;                                    // Edetään tekstissä.
    f[jplus] = i;               // Asetetaan ”f[j+2] = f[jplus+1] = i + 1” eli f[jplus] = i,
}                               // koska molempia indeksejä juuri kasvatettiin.
```

Algoritmi 1.2.15:
Morris-Pratt-algoritmin korjausfunktion alustus.

Yleensä Morris-Pratt-algoritmin alustusalgoritmissa ei tehdä suoraan sijoitusta $f_k(1) = 0$, vaan tämä hoidetaan esimerkiksi alustamalla muuttujat i ja j^+ yhtä pienemmillä arvoilla, eli $i = -1$ ja $j^+ = 0$, jolloin tämä asetus $f_k(1) = 0$ tulee hoidettua ensimmäisellä pääsilmukan kierroksella.

Algoritmin 1.2.15 lineaarisuus voitaisiin perustella Morris-Pratt-algoritmin lineaarisuuden perusteella, sillä koska aina $i < j^+$, niin myös selvästi aina $i < m$, ja siten ainoa aikavaativuuteen vaikuttava ero algoritmien 1.2.7 ja 1.2.15 välillä on pääsilmukan suoritusehto. Mutta jos esimerkiksi $n = 2m$, niin suoritusehto $j \leq m - 2$ on tosi varmasti aina silloin, kun myös algoritmin alkuperäinen suoritusehto $j - i < n - m + 1$ on tosi, ja lauseen 1.2.13 mukaan Morris-Pratt-algoritmi toimii tällöin lineaarisessa ajassa tekstin pituuteen $2m$ nähden, eli ajassa $O(m)$, joka on lineaarinen hahmon pituuteen nähden. Mutta todistetaan algoritmin 1.2.15 lineaarisuus myös täsmällisemmin:

Lause 1.2.16:
Algoritmi 1.2.15 on aikavaativuudeltaan lineaarinen hahmon pituuteen m nähden, eli $O(m)$.

Todistus:
Tarkastellaan sitä, kuinka monta kertaa algoritmissa indeksille i annetaan uusi arvo. Koska tämä tehdään sekä ulko- että sisäsilmukassa, on tämä lukumäärä lineaarisesti

verrannollinen koko algoritmissa suoritettujen operaatioiden lukumäärään. Algoritmin alussa i saa arvon 0, ja tämän jälkeen algoritmissa on aina voimassa $i \geq -1$. Koska aina on $f_k(i) < i$, vähentää jokainen suoritettu sisempi silmukka i:n arvoa vähintään yhdellä. Toisaalta nähdään, että i:n arvo kasvaa ainoastaan ulommassa silmukassa, ja silloinkin joka kierroksella aina tasan yhdellä. Tämän vuoksi sisäsilmukan suorituskertojen lukumäärä on koko ajan pienempi kuin ulkosilmukan suorituskertojen lukumäärä, sillä muuten i:n arvo menisi jossain vaiheessa pienemmäksi kuin -1. Suoritusehdon $j^+ < m$ ansiosta ulkosilmukka suoritetaan tasan m - 2 kertaa, joten sisä- ja ulkosilmukoita suoritetaan yhteensä korkeintaan m - 2 + m - 2 = 2m - 4 kertaa. Korjausfunktion arvot laskeva algoritmi 1.2.15 toimii siis lineaarisessa ajassa O(m) hahmon pituuden suhteen.

Morris-Pratt-algoritmin korjausfunktion arvoja voidaan vielä hieman tehostaa. Nimittäin jos ei-täsmäävä vertailu on tehty merkkien *teksti*[j] ja *hahmo*[i] välillä ja *hahmo*[$f_k(i)$] = *hahmo*[i], tiedetään jo etukäteen, että korjausfunktion antaman siirtymän jälkeen vertailuvuorossa olevat merkit *teksti*[j] ja *hahmo*[$f_k(i)$] = *hahmo*[i] eivät täsmää. Määritellään nyt sellainen funktion f_k optimoitu versio f_{ko}, että se hyppää näiden edellisenkaltaisten turhiksi tiedettyjen vertailujen yli. Tällöin funktio f_{ko} on muuten identtinen funktion f_k kanssa, mutta välillä $1 \leq i \leq m$ sen arvoille asetetaan lisäksi ehto *hahmo*[$f_{ko}(i)$] $\neq$ *hahmo*[i], ja tämän ansiosta $f_{ko}(i)$ voi saada myös erikoisarvon -1 kyseisellä välillä. Tällaista optimoitua korjausfunktiota käyttävää versiota Morris-Pratt-algoritmista kutsutaan Knuth-Morris-Pratt-algoritmiksi [Knuth et al., 1977], ja kirjallisuudessa nämä kaksi algoritmia usein samaistetaan Knuth-Morris-Pratt-algoritmiksi (esim. [Gusfield, 1997]).

Määritelmä 1.2.17:
Optimoitu Knuth-Morris-Pratt-algoritmin korjausfunktio f_{ko}:
$f_{ko}(0) = -1$,
$f_{ko}(i) = \max\{k \mid (k = -1) \vee (k \in R_{hahmo,\, i-1} \wedge (i = m \vee hahmo[k] \neq hahmo[i]))\}$, kun $1 \leq i \leq m$.

Algoritmin 1.2.15 pohjalta saadaan pienellä muutoksella funktion f_{ko} arvot alustava algoritmi. Luvun $f_{ko}(j)$ tulee saada arvon $i = f_k(j)$ sijaan arvo $f_{ko}(i)$ mikäli *hahmo*[j] = *hahmo*[i], ja muussa tapauksessa $f_{ko}(j) = f_k(j) = i$. Tämä on mahdollista toteuttaa, sillä koska $i = f_k(j) < j$ ja algoritmissa 1.2.15 edetään järjestyksessä j = 1, 2,…, m-1, on arvo $f_{ko}(i)$ laskettu jo ennen arvon $f_{ko}(j)$ käsittelyä. Todistetaan tämän menettelyn pätevyys vielä täsmällisemmin:

Lause 1.2.18:
Jos $1 \leq j < m$ ja *hahmo*[$f_k(j)$] = *hahmo*[j], niin $f_{ko}(j) = f_{ko}(f_k(j))$, ja muulloin $f_{ko}(j) = f_k(j)$.

Todistus:
Tarkastellaan ensin tilannetta, jossa $hahmo[f_k(j)] \neq hahmo[j]$, $j = 0$ tai $j = m$. Tällöin määritelmien 1.2.4 ja 1.2.17 nojalla pätee $f_{ko}(j) = f_k(j)$, ja lauseen väitteen jälkimmäinen osa on tosi.

Tarkastellaan nyt tilannetta, jossa $1 \leq i < m$ $hahmo[f_k(j)] = hahmo[j]$. Määritelmän 1.2.17 mukaan tällöin $f_{ko}(j) = \max\{k \mid (k = -1) \vee (k \in R_{hahmo,j-1} \wedge hahmo[k] \neq hahmo[j])\}$. Osoitetaan ensin oikeaksi ehto $f_{ko}(j) \geq f_{ko}(f_k(j))$ siten, että jaetaan käsittely seuraaviin tapauksiin luvun $f_{ko}(f_k(j))$ arvon mukaan:

1) $f_{ko}(f_k(j)) = -1$:
   Selvästi $f_{ko}(j) \geq -1$, joten tässä tapauksessa $f_{ko}(j) \geq f_{ko}(f_k(j))$.

2) $f_{ko}(f_k(j)) = 0$:
   Oletuksen mukaan $hahmo[j] = hahmo[f_k(j)]$, ja koska tässä tapauksessa $hahmo[0] \neq hahmo[f_k(j)]$, pätee myös $hahmo[0] \neq hahmo[j]$ ja siten $f_{ko}(j) \geq 0 = f_{ko}(f_k(j))$, sillä $0 \in R_{hahmo,\, j-1}$.

3) $f_{ko}(f_k(j)) > 0$:
   Tässä tapauksessa $f_{ko}(f_k(j)) \in R_{hahmo,\, f_k(j)-1}$, ja pätee $hahmo[0 \ldots f_{ko}(f_k(j))-1] = hahmo[f_k(j)-f_{ko}(f_k(j)) \ldots f_k(j)-1]$ sekä $hahmo[f_{ko}(f_k(j))] \neq hahmo[f_k(j)]$. Koska selvästi aina $f_{ko}(j) \leq f_k(j)$ ja lauseen 1.2.5 mukaan $f_k(j) < j$, pätee myös $f_{ko}(f_k(j)) \leq f_k(f_k(j)) < f_k(j)$. Näin ollen, koska $f_k(j) > 0$ ja $f_k(j) \in R_{hahmo,\, j-1}$, on täsmäävyys $hahmo[0 \ldots f_k(j)-1] = hahmo[j-f_k(j) \ldots j-1]$ voimassa. Yhdistämällä tämä aiemman kanssa saadaan $hahmo[0 \ldots f_{ko}(f_k(j))-1] = hahmo[j-f_{ko}(f_k(j)) \ldots j-1]$ eli $f_{ko}(f_k(j)) \in R_{hahmo,\, j-1}$. Koska $hahmo[f_{ko}(f_k(j))] \neq hahmo[f_k(j)]$ ja $hahmo[f_k(j)] = hahmo[j]$, pätee myös $hahmo[f_{ko}(f_k(j))] \neq hahmo[j]$, ja siten luvun $f_{ko}(j)$ maksimaalisuuden perusteella $f_{ko}(j) \geq f_{ko}(f_k(j))$.

Kohtien 1) - 3) mukaan on siis $f_{ko}(j) \geq f_{ko}(f_k(j))$. Tehdään nyt vastaoletus $f_{ko}(j) \neq f_{ko}(f_k(j))$, joka tarkoittaa juuri osoitetun seikan sekä ehtojen $f_{ko}(j) \leq f_k(j)$ ja $hahmo[i] = hahmo[f_k(j)]$ voimassaolon perusteella, että $f_{ko}(f_k(j)) < f_{ko}(j) < f_k(j)$. Koska $f_{ko}(f_k(j)) \geq -1$, voidaan tämän tilanteen tarkastelu jakaa seuraaviin kahteen tapaukseen:

1° $f_{ko}(j) = 0$:

Alkuoletuksen mukaan *hahmo*[*j*] = *hahmo*[$f_k(j)$], ja koska tässä tapauksessa *hahmo*[0] ≠ *hahmo*[*j*], pätee myös *hahmo*[0] ≠ *hahmo*[$f_k(j)$], ja siten $f_{ko}(f_k(j)) \geq 0$, sillä $0 \in R_{hahmo,\ f_k(j)-1}$. Päädyttiin ristiriitaan, sillä tämän mukaan $f_{ko}(j) > 0$.

2° $f_{ko}(j) > 0$:
Nyt *hahmo*[0…$f_{ko}(j)$-1] = *hahmo*[*j*-$f_{ko}(j)$…*j*-1] ja lisäksi myös $f_k(j) > 0$, joten *hahmo*[0…$f_k(j)$-1] = *hahmo*[*j*-$f_k(j)$…*j*-1]. Edellisten täsmäävyyksien ja tiedon $f_{ko}(j) \leq f_k(j)$ perusteella saadaan *hahmo*[0…$f_{ko}(j)$-1] = *hahmo*[$f_k(j)$-$f_{ko}(j)$…$f_k(j)$-1] eli $f_{ko}(j) \in R_{hahmo,\ f_k(j)-1}$. Koska oletuksen mukaan *hahmo*[$f_k(j)$] = *hahmo*[*j*] ja määritelmän 1.2.17 nojalla *hahmo*[$f_{ko}(j)$] ≠ *hahmo*[*j*], pätee myös *hahmo*[$f_{ko}(j)$] ≠ *hahmo*[$f_k(j)$]. Edelliset kaksi huomiota yhdistämällä nähdään arvon $f_{ko}(f_k(j))$ maksimaalisuudesta, että f_{ko} on voimassa ehto $(f_k(j)) \geq f_{ko}(j)$, joka on ristiriidassa oletukseen sisältyvän ehdon $f_{ko}(f_k(j)) < f_{ko}(j)$ kanssa.

Yhdistämällä tapauksien 1° ja 2° tulos kohtien 1) - 3) mukaisen tuloksen kanssa päästiin lopputulokseen, että jos $1 \leq j < m$ ja *hahmo*[$f_k(j)$] = *hahmo*[*j*], niin $f_{ko}(j) = f_{ko}(f_k(j))$ ja siten myös lauseen ensimmäinen osa on tosi.

Optimoidun korjausfunktion f_{ko} arvot laskevan algoritmin 1.2.19 aikavaativuuden nähdään selvästi olevan sama kuin algoritmin 1.2.15 eli O(*m*).

Tässä pätee sama kuin aiemmin esitetyn algoritmin 1.2.15 yhteydessä, eli arvon $f_{ko}(1)$ sijoitusta ei useimmiten suoriteta suoraan vaan alustamalla $i = -1$ ja $j^+ = 0$.
Algoritmin 1.2.19 toimivuudesta voi varmistua algoritmin 1.2.15 toimivuuden sekä lauseen 1.2.18 nojalla, mikäli pääsilmukan lopussa pätee aina $i = f_k(j^+)$ algoritmiin tehdyistä muutoksista huolimatta. Tämä on helppo havaita todeksi, sillä ainoa tähän vaikuttava muutos on sisäsilmukassa tehtävän sijoituksen $i = f_k(i)$ korvaaminen sijoituksella $i = f_{ko}(i)$, ja tämä ei vaikuta indeksin *i* arvoon sisäsilmukasta poistuttaessa, koska silloin on joko $i = -1$ tai *hahmo*[*j*] ≠ *hahmo*[*i*]. Nimittäin jos $f_k(i) = -1$, niin myös $f_{ko}(i) = -1$, ja toisaalta jos $i \geq 0$ ja *hahmo*[*j*] = *hahmo*[*i*], niin myös joko *hahmo*[*j*] = *hahmo*[$f_k(i)$] tai $f_k(i) = f_{ko}(i)$.

```
int m;                                  // Hahmon pituus.
int fo[m];                              // Optimoidun korjausfunktion arvotaulukko, fo[i] = f_ko(i).
fo[0] = –1;                 // Jos ei-täsmäävä vertailu heti hahmon ensimmäisen merkin kohdalla,
                            // tulee seuraava vertailu tehdä ”kohdasta –1”, joka tarkoittaa, että
                            // siirrytään tekstissä  yksi merkki eteenpäin ja hahmossa ensimmäiseen
                            // merkkiin.
int i = 0;                                          // Alustetaan hahmon indeksi arvolla i = 0.
int jplus = 1;              // Muuttuja jplus sisältää arvon j + 1, ja aina hahmo[jplus] = teksti[j].
if(hahmo[0] != hahmo[1])                // Lasketaan arvo f_ko[1] erikseen kuten arvo f_k[1]
{                                       // algoritmissa 1.2.15, mutta nyt soveltaen lausetta
    fo[1] = 0;                          // 1.2.18.
}
else
{
    fo[1] = -1;
}
while(jplus < m)            // Käydään kaikki korjausfunktion indeksit läpi, arvo f_ko(jplus+1)
{                           // lasketaan jplus:nnella kierroksella.
    while(i >= 0 && hahmo[jplus] != hahmo[i])           // While-silmukan ehto tutkii onko
    {                       // kyseessä merkin hahmo[i] kohdalla tapahtuva ei-täsmäävä
        i = fo[i];          // vertailu, joka ei johda etenemiseen merkkijonossa
    }                       // hahmo[1…m-1].
    i++;                    // Hahmossa joko edetään askel tai pysytään kohdassa 0 (jos i = -1).
    jplus++;                                    // Edetään tekstissä.
    if(jplus < m && hahmo[jplus] == hahmo[i])   // Tutkitaan, päteekö fo[jplus] = f[jplus].
    {
        fo[jplus] = fo[i];          // Asetetaan fo[jplus] = fo[i] lauseen 1.2.18 mukaisesti.
    }
    else
    {
        fo[jplus] = i;              // Asetetaan ”f[j+2] = f[jplus+1] = i + 1” eli f[jplus] = i,
    }                               // koska molempia indeksejä juuri kasvatettiin.
}
```

Algoritmi 1.2.19:
Knuth-Morris-Pratt-algoritmin optimoidun korjausfunktion alustus.

1.3 Reaaliaikainen merkkijonohaku

Määritelmän 1.2.17 mukaista Knuth-Morris-Pratt-algoritmin optimoitua korjausfunktiota f_{ko} saadaan tehostettua edelleen muuttamalla se ns. ”reaaliaikaiseksi” [Gusfield, 1997] korjausfunktioksi. Tämä tarkoittaa korjausfunktion sellaista toteutusta, että tekstiä luetaan järjestyksessä merkki kerrallaan, ja aina yksittäisen tekstin merkin kohdalla tehtävien operaatioiden aikavaativuus on vakio. Nimitys ”reaaliaikainen” viittaa siihen, että hahmon etsintä voidaan näin ollen suorittaa asymptoottista aikavaativuutta ajatellen reaaliajassa samalla, kun tekstiä luetaan merkki kerrallaan, koska myös yksittäisen tekstin merkin lukuoperaatio on aikavaativuudeltaan vakio.

Käytännössä korjausfunktion ”reaaliaikaisuus” toteutuu siten, että korjausfunktio kertoo suoraan aina kulloinkin vertailuvuorossa olevan merkin *teksti*[j] perusteella missä kohtaa hahmoa Morris-Pratt-algoritmi vertailisi täsmäävästi tämän merkin tai tapahtuisiko näin ylipäätään missään hahmon kohdassa. Jos tämä täsmäys tapahtuisi merkin *hahmo*[i] kohdalla, vertailisi Morris-Pratt-algoritmi seuraavaksi merkkejä *hahmo*[i+1] (tai *hahmo*[$f_k(m)$], kun $i = m - 1$) sekä *teksti*[j+1]. Jos taas täsmäystä ei tapahtuisi lainkaan, voitaisiin hahmo siirtää suoraan nykyisen kohdan yli, jolloin Morris-Pratt vertaisi seuraavaksi merkkejä *hahmo*[0] ja *teksti*[j+1]. Koska Morris-Pratt-algoritmissa merkkien *hahmo*[i_v] ja *teksti*[j_v] välisen täsmäävän vertailun jälkeen pätee aina ehto $i_v + 1 = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\}$, halutaan reaaliaikaisessa algoritmissa siis siirtyä hahmossa indeksin i_v kohdalta seuraavaksi sellaisen indeksin i_u kohdalle, että $i_u = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j_v]\}$.

Käytetään reaaliaikaisesta korjausfunktiosta merkintää f_r. Funktion f_r antaman siirtymän suuruus (eli seuraava hahmosta vertailtava merkki) riippuu aina merkistä *teksti*[j], joten funktio f_r on kaksipaikkainen. Merkkien *hahmo*[i] ja *teksti*[j] = λ kohdalla tapahtuneen ei-täsmäävän vertailun jälkeen sovelletaan arvoa $f_r(i, \lambda)$, jolloin seuraava vertailu tehdään merkkien *hahmo*[$f_r(i, \lambda)$] ja *teksti*[j+1] välillä. Käytännön toteutusta silmälläpitäen voi näin ollen olla järkevää sisällyttää myös täsmäävät askeleet funktioon f_r, eli $f_r(i, hahmo[i]) = i+1$, kun $i < m - 1$. Koska mitään tekstin merkkiä ei verrata kahteen kertaan, verrataan merkkien *hahmo*[i] ja *teksti*[j] välisen vertailun jälkeen aina merkkejä *hahmo*[$f_r(i, teksti[j])$] ja *teksti*[j+1] riippumatta merkkien *hahmo*[i] ja *teksti*[j] täsmäävyydestä. Tämän vuoksi algoritmin etsintävaiheessa ei tarvitse erikseen tutkia, täsmäsivätkö merkit *hahmo*[i] ja *teksti*[j] keskenään, sillä hahmon esiintymien löytämiseksi riittää tarkistaa, missä kohdissa

indeksi *i* saa arvon *m*. Tämä tapahtuu täsmälleen silloin kun merkit *hahmo*[*m*-1] ja *teksti*[*j*] on täsmätty keskenään ja siis koko hahmo löydetty.

Funktion f_r määritelmä on hieman samantapainen kuin optimoidulla korjausfunktiolla f_{ko}, mutta tässä yhteydessä korjausfunktio etenee hahmossa aina myös ei-täsmänneen vertailun jälkeen, jota vastaaviin funktionarvoihin on siis lisätty + 1, ja mukana on lisäksi täsmäävät askeleet:

Määritelmä 1.3.1:
Reaaliaikainen korjausfunktio f_r:
$f_r(0, \lambda) = 0$, kun $\lambda \neq hahmo[0]$,
$f_r(0, \lambda) = 1$, kun $\lambda = hahmo[0]$,
$f_r(i, \lambda) = \max\{k \mid (k = -1) \vee (k \in R_{hahmo,\, i-1} \wedge hahmo[k] = \lambda)\} + 1$, kun $i = m$ tai $1 \leq i < m$ ja $\lambda \neq hahmo[i]$,
$f_r(i, \lambda) = i + 1$, kun $1 \leq i < m$ ja $\lambda = hahmo[i]$.

Korjausfunktiota f_r soveltavan reaaliaikaisen merkkijonohakualgoritmin toteutus on yksinkertainen: Aloitetaan merkkien *hahmo*[0] ja *teksti*[0] kohdalta, eli alussa $i = 0$ ja $j = 0$, ja edetään tekstissä merkki kerrallaan siirtyen jokaisen tekstin etenemisaskeleen yhteydessä aina uuteen hahmon vertailukohtaan arvon $f_r(i, j)$ mukaisesti. Samalla tarkistetaan aina, löytyikö kohtaan *j* tekstissä päättyvä hahmon esiintymä, ja algoritmin suorittamista jatketaan kunnes *j* on käynyt läpi koko tekstin.

```
int m;                          // Hahmon pituus.
int n;                          // Tekstin pituus.
int a;                          // Käytetyn aakkoston koko.
int N(λ);                // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
int fr[m][a];                   // Reaaliaikaisen korjausfunktion arvotaulukko.
int i = 0;                      // Vertailuvuorossa olevan hahmon merkin indeksi, siis alussa i = 0.
int j = 0;                      // Vertailuvuorossa olevan tekstin merkin indeksi, siis alussa j = 0.
while(j < n)                    // Käydään koko teksti läpi.
{
    i = fr[i][N(teksti[j])];    // Siirrytään merkin teksti[j] perusteella suoraan seuraavan
                                // hahmon osalta mahdollisesti täsmäävän merkin kohdalle.
    if(i == m)                  // Löytyikö kohtaan teksti[j] päättyvä hahmon esiintymä?
    {
        loytyi();               // Löytymistä seuraavat toimenpiteet.
    }
    j++;                        // Edetään yksi merkki tekstissä.
}
```

Algoritmi 1.3.2:
Reaaliaikainen merkkijonohaku.

Selvästi algoritmin 1.3.2 aikavaativuus on O(n). Todistetaan vielä täsmällisesti tämän algoritmin toimivuus lauseissa 1.3.3 ja 1.3.4.

Lause 1.3.3:
Algoritmin1.3.2 pääsilmukan alussa pätee aina ehto $f_r(i, teksti[j]) = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$.

Todistus:
Algoritmin alussa $i = 0$ ja $j = 0$, $f_r(0, teksti[0]) = 0$, kun $hahmo[0] \neq teksti[0]$, ja $f_r(0, teksti[0]) = 1$, kun $hahmo[0] = teksti[0]$, joten silloin lauseen väite on selvästi tosi. Oletetaan nyt, että väite oli voimassa indeksien arvoilla i_v ja j_v eli että $f_r(i_v, teksti[j_v]) = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\}$, ja tarkastellaan seuraavaa pääsilmukan kierrosta vastaavia arvoja i_u ja $j_u = j_v + 1$. Jaetaan tarkastelu seuraaviin tapauksiin:

1) $f_r(i_v, teksti[j_v]) = 0$:
   Tässä tapauksessa $i_u = f_r(i_v, teksti[j]) = 0$. Oletuksen sekä lauseen 1.2.9 mukaan saadaan $\max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\} + 1 = 0 + 1 = 1 \geq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$. Toisaalta $\max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\} \neq 1$, kun $hahmo[0] = teksti[j_u]$, ja $\max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\} \geq 1$, kun $hahmo[0] = teksti[j_u]$. Koska $f_r(0, teksti[j_u]) = 0$, kun $hahmo[0] \neq teksti[j_u]$, ja $f_r(0, teksti[j_u]) = 1$, kun $hahmo[0] = teksti[j_u]$, pätee ehto $f_r(i_u, teksti[j_u]) = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$ aina tässä tapauksessa.

2) $hahmo[i_u] = teksti[j_u]$ ja $0 < f_r(i_v, teksti[j_v]) < m$:
   Nyt on $i_u = f_r(i_v, teksti[j_v]) = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\}$. Edellisistä seuraa ehdon $hahmo[i_u] = teksti[j_u]$ kanssa yhdistettynä, että $hahmo[0\ldots i_u] = teksti[j_u\text{-}i_u\text{+}1\ldots j_u]$ ja $f_r(i_u, teksti[j_u]) = i_u + 1$, joten nyt $\max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\} \geq f_r(i_u, teksti[j_u])$. Toisaalta lauseen 1.2.9 perusteella pätee $f_r(i_u, teksti[j_u]) = i_u + 1 = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\} + 1 \geq \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$, mistä seuraa, että tässäkin tapauksessa pätee yhtäsuuruus $f_r(i_u, teksti[j_u]) = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$.
3) $hahmo[i_u] \neq teksti[j_u]$ ja $0 < f_r(i_v, teksti[j_v]) < m$, tai $f_r(i_v, teksti[j_v]) = m$:

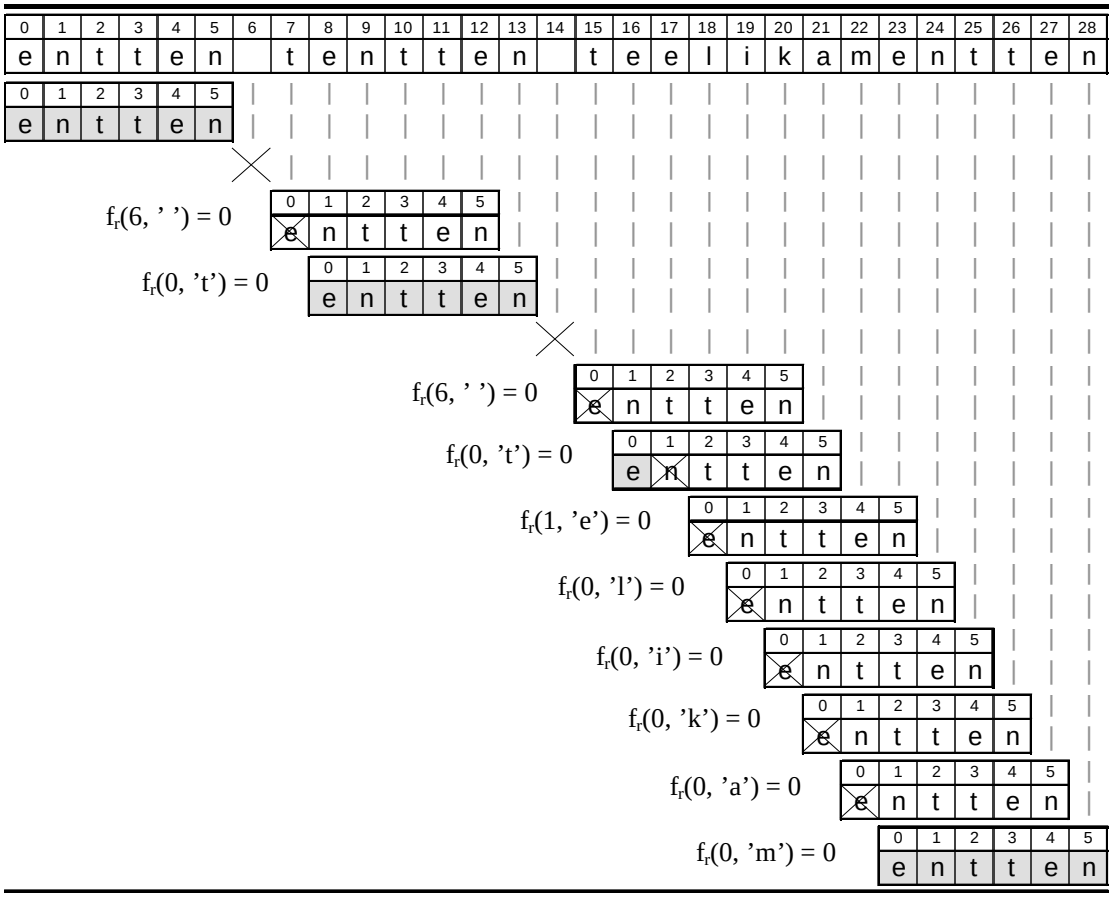
Nyt on $i_u = f_r(i_v, teksti[j_v]) = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots.j_v]\}$. Koska nyt $i_u > 0$, pätee $hahmo[0\ldots i_u\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]$, ja tämän sekä määritelmien 1.2.2 ja 1.3.1 nojalla saadaan nyt $f_r(i_u, teksti[j_u]) = \max\{k \mid (k = -1) \vee (k \in R_{hahmo,\, i_u\text{-}1} \wedge hahmo[k] = teksti[j_u])\} + 1 = \max\{k \mid (k = -1) \vee (((k = 0) \vee (hahmo[0\ldots k\text{-}1] = hahmo[i_u\text{-}k\ldots i_u\text{-}1])) \wedge (hahmo[k] = teksti[j_u]))\} + 1 = \max\{k \mid (k = -1) \vee (((k = 0) \vee (hahmo[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v])) \wedge(hahmo[k] = teksti[j_u]))\} + 1 = \max\{k \mid (k = 0) \vee (hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u])\}$.

Tapaukset 1) - 3) kattavat kaikki eri mahdollisuudet, ja jokaisessa niistä pätee myös $f_r(i_u, teksti[j_u]) = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots.j_u]\}$. Siis lauseen väite voidaan todeta oikeaksi induktioperiaatteen nojalla.

Lause 1.3.4:
Reaaliaikainen merkkijonohakualgoritmi kirjaa oikein löydetyksi kaikki hahmon esiintymät tekstissä.

Todistus:
Reaaliaikainen merkkijonohakualgoritmi käy läpi jokaisen tekstin indeksin, ja lauseen 1.3.3 nojalla aina merkin $teksti[j]$ kohdalle siirryttäessä pätee $f_r(i, teksti[j]) = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$. Selvästi $f_r(i, teksti[j]) = m$, jos ja vain jos tekstissä on merkkiin $teksti[j]$ päättyvä hahmon esiintymä, ja reaaliaikainen merkkijonohakualgoritmi kirjaa hahmon löydetyksi täsmälleen tämän ehdon pätiessä, joten jokainen hahmon esiintymä tulee kirjattua täsmälleen oikein.

Kuva 1.3.5 esittää reaaliaikaisen merkkijonon toiminnasta jo tutun esimerkin.



Kuva 1.3.5:

Merkkijonon *hahmo*[0...5] = "tentten" etsiminen merkkijonosta *teksti*[0...28] = "entten tentten teelikamentten" reaaliaikaista merkkijonohakualgoritmia käyttäen. Tekstiä luetaan merkki kerrallaan, ja hahmo siirretään aina joko seuraavaan sellaiseen kohtaan, jossa sen maksimaalinen alkuosa täsmää toistaiseksi luetun tekstinosan lopun kanssa, tai sitten kokonaan jo luetun tekstinosan ohi tämän ollessa mahdotonta. Kuvassa täsmäävästi vertaillut hahmon merkit on tummennettu, ja ei-täsmäävästi vertaillut merkit on yliviivattu. Lisäksi jokaisen siirtymän yhteydessä on mainittu siinä sovellettu korjausfunktion arvo.Yhteensä tässä esimerkissä tehdään reaaliaikaisen korjaufunktion soveltamisen muodossa täsmälleen *n* eli 29 merkkivertailua.

Reaaliaikaisen korjausfunktion f_r arvot saadaan laskettua korjausfunktion f_k arvojen avulla seuraavan lausetta 1.2.18 muistuttavan yksinkertaisen säännön mukaan.

Lause 1.3.6:
Välillä $1 \leq i \leq m$ pätee, että $f_r(i, \lambda) = i + 1$, jos $hahmo[i] = \lambda$, ja $f_r(i, \lambda) = f_r(f_k(i), \lambda)$, jos $hahmo[i] \neq \lambda$.

Todistus:
Tarkastellaan ensin tilannetta, jossa $hahmo[i] = \lambda$. Tällöin määritelmän 1.3.1 nojalla pätee aina $f_r(i, \lambda) = i + 1$, joten lauseen väittämä on tältä osin tosi.

Tarkastellaan nyt tilannetta, jossa $i = m$, tai $1 \leq i < m$ ja $hahmo[i] \neq \lambda$. Määritelmän 1.3.1 mukaan tällöin $f_r(i, \lambda) = \max\{k \mid (k = -1) \vee (k \in R_{hahmo,\, i-1} \wedge hahmo[k] = \lambda)\} + 1$.
Osoitetaan ensin, että $f_r(i, \lambda) \geq f_r(f_k(i), \lambda)$ jakamalla käsittely seuraaviin tapauksiin luvun $f_r(f_k(i), \lambda)$ arvon mukaan:

1) $f_r(f_k(i), \lambda) = 0$:
   Selvästi $f_r(i, \lambda) \geq 0$, joten tässä tapauksessa $f_r(i, \lambda) \geq f_r(f_k(i), \lambda)$.

2) $f_r(f_k(i), \lambda) = 1$:
   Määritelmästä 1.3.1 nähdään, että tässä tapauksessa $hahmo[0] = \lambda$. Siten pätee $f_r(i, \lambda) \geq 1 = f_r(f_k(i), \lambda)$, sillä $0 \in R_{hahmo,\, i-1}$.

3) $f_r(f_k(i), \lambda) > 1$:
   Tässä tapauksessa $f_r(f_k(i), \lambda) - 1 \in R_{hahmo,\, f_k(i)-1}$ sekä $f_r(f_k(i), \lambda) - 1 > 0$, joten pätee $hahmo[0 \ldots f_r(f_k(i), \lambda)-2] = hahmo[f_k(i)-f_r(f_k(i), \lambda)+1 \ldots f_k(i)-1]$ ja lisäksi $hahmo[f_r(f_k(i), \lambda) - 1] = \lambda$. Koska selvästi aina $f_r(i, \lambda) \leq f_k(i)$ ja lauseen 1.2.5 mukaan $f_k(i) < i$, pätee myös $f_r(f_k(i), \lambda) \leq f_k(f_k(i)) < f_k(i)$. Näin ollen koska $f_k(i) > 0$ ja $f_k(i) \in R_{hahmo,\, i-1}$, on täsmäävyys $hahmo[0 \ldots f_k(i)-1] = hahmo[i-f_k(i) \ldots i-1]$ voimassa. Yhdistämällä tämä aiemman kanssa saadaan $hahmo[0 \ldots f_r(f_k(i), \lambda)-2] = hahmo[i-f_r(f_k(i), \lambda)+1 \ldots i-1]$ eli $f_r(f_k(i), \lambda) - 1 \in R_{hahmo,\, i-1}$. Koska lisäksi $hahmo[f_r(f_k(i), \lambda)-1] = \lambda$, pätee luvun $f_r(i, \lambda)$ maksimaalisuuden perusteella $f_r(i, \lambda) \geq f_r(f_k(i), \lambda)$.

Kohtien 1) - 3) mukaan siis $f_r(i, \lambda) \geq f_r(f_k(i), \lambda)$. Tehdään nyt vastaoletus $f_r(i, \lambda) \neq f_r(f_k(i), \lambda)$. Sen perusteella nähdään edellinen sekä ehdot $f_r(i, \lambda) \leq f_k(i)$ ja $hahmo[i] \neq \lambda$ huomioon ottaen, että $f_r(f_k(i), \lambda) < f_r(i, \lambda) < f_k(i)$. Koska $f_r(f_k(i), \lambda) \geq 0$, voidaan tämän tilanteen tarkastelu jakaa seuraaviin kahteen tapaukseen:

1° $f_r(i, \lambda) = 1$:
Tässä tapauksessa on pakko päteä *hahmo*[0] = λ. Toisaalta jos $f_r(f_k(i), \lambda) < f_r(i, \lambda)$, niin tässä tapauksessa $f_r(f_k(i), \lambda) = 0$, mikä on mahdollista ainoastaan jos *hahmo*[0] ≠ λ. Päädyttiin siis ristiriitaan.

2° $f_r(i, \lambda) > 1$:
Nyt *hahmo*[0…$f_r(i, \lambda)$-2] = *hahmo*[i-$f_r(i)$+1…i-1], ja lisäksi myös $f_k(i) > 1$, joten *hahmo*[0…$f_k(i)$-1] = *hahmo*[i-$f_k(i)$…i-1]. Edellisten täsmäävyyksien ja tiedon $f_r(i, \lambda) \leq f_k(i)$ perusteella saadaan *hahmo*[0…$f_r(i, \lambda)$-2] = *hahmo*[$f_k(i)$-$f_r(i, \lambda)$+1…$f_k(i)$-1] eli $f_r(i, \lambda) - 1 \in R_{hahmo,\, f_k(i)-1}$. Lisäksi määritelmän 1.3.1 nojalla pätee *hahmo*[$f_r(i, \lambda)$-1] = λ, joten arvon $f_r(f_k(i), \lambda)$ maksimaalisuudesta seuraa, että $f_r(f_k(i), \lambda) \geq f_r(i, \lambda)$, mikä on ristiriita.

Koska molemmissa tapauksissa 1° ja 2° päädyttiin ristiriitaan, oli vastaoletus väärä. Siten myös lauseen loppuosan väite $f_r(i, \lambda) = f_r(f_k(i), \lambda)$, kun $1 \leq i < m$ ja *hahmo*[i] ≠ λ, on tosi.

Muotoa $f_r(0, \lambda)$ olevat arvot on helppo laskea suoraan määritelmän 1.3.1 pohjalta, ja lauseen 1.3.6 nojalla voidaan tämän jälkeen laskea kaikki muut arvot $f_r(i, \lambda)$ etenemällä järjestyksessä $i = 1, 2, \ldots, m - 1$ siten, että asetetaan $f_r(i, \lambda) = i + 1$ aina, kun *hahmo*[i] = λ, ja $f_r(i, \lambda) = f_r(f_k(i), \lambda)$ muulloin. Tämä toimenpide onnistuu halutulla tavalla, sillä koska aina $f_k(i) < i$, on arvo $f_r(f_k(i), \lambda)$ laskettu jo ennen arvon $f_r(i, \lambda)$ käsittelyä. Algoritmi 1.3.7 toimii tämän menettelyn mukaisesti ja on selvästi aikavaativuudeltaan $O(m \times |\Lambda|)$, missä Λ on käytössä oleva aakkosto, ja $|\Lambda|$ sen merkkien lukumäärä.

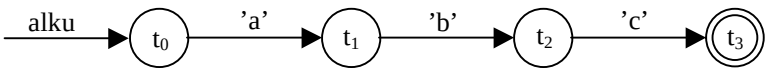
Reaaliaikainen merkkijonohakualgoritmi vastaa äärellistä tila-automaattia, joka tunnistaa merkki kerrallaan luettavasta syötteestä kaikki siinä olevat hahmon esiintymät. Käsittelen tässä asiaa lyhyesti.

Luonnollinen tapa tila-automaatin kuvaamiseen on suunnattu graafi, jossa solmut vastaavat tiloja ja ”ehdollistetut” kaaret mahdollisia siirtymiä. Kaaren yhteydessä ilmoitettu symbolijoukko kertoo, mitkä syötemerkit aiheuttavat kaaren suuntaisen siirtymän, jos kaaren alkupäässä olevaa solmua vastaava tila on aktiivinen. Lisäksi merkitään automaatin lopputilaa kaksoisreunuksella ja osoitetaan alkutila ”alku”-nuolella. Tila-automaatin määrittelyn ja kuvauksen yksinkertaistamiseksi otetaan tässä käyttöön seuraava sopimus: mikäli syötteenä saatua merkkiä vastaavaa siirtymää ei ole erikseen määritelty, tehdään tällöin aina siirtymä automaatin alkutilaan.

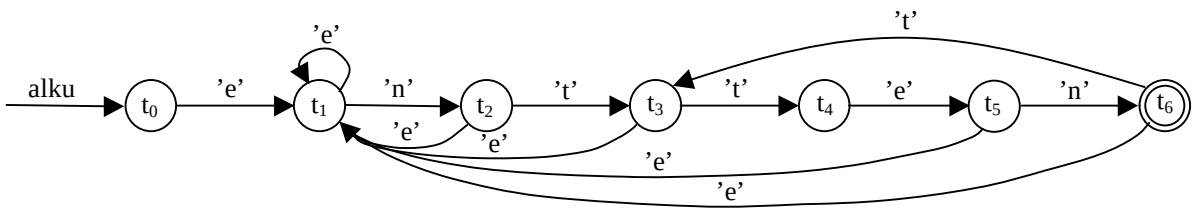
```
int m;                                  // Hahmon pituus.
int a;                                  // Käytetyn aakkoston koko.
int N(λ);          // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
int f[m];                               // Korjausfunktion arvotaulukko, f[i] = fk(i).
int fr[m][a];                           // Reaaliaikaisen korjausfunktion arvotaulukko.
int i;                                  // Hahmoa läpikäyvä indeksi.
int j;                                  // Aakkoston merkkejä läpikäyvä apuindeksi.

AlustaKorjausfunktio();            // Alustetaan korjausfunktio fk esim. algoritmilla 1.2.15.
for(j = 0; j < a; j++)                  // Käydään läpi aakkoston jokainen merkki l asettaen
{                                       // aina fr(0, l) = 0.
    fr[0][j] = 0;
}
fr[0][N(hahmo[0])] = 1;                 // Asetetaan arvo fr(0, hahmo[i]) oikeaksi.
for(i = 1; i < m; i++)                  // Käydään läpi koko hahmo.
{
    for(j = 0; j < a; j++)              // Käydään läpi aakkoston jokainen merkki.
    {
        if(N(hahmo[i]) == j)            // Tarkistetaan johtaisiko korjausfunktiolle
        {                               // laskettu arvo turhaan vertailuun merkin
            fr[i][j] = i + 1;           // hahmo[i] kohdalla tapahtuneen ei-täsmäävän
        }                               // vertailun jälkeen.
        else
        {
            fr[i][j] = fr[f[i]][j];     // hahmo[f[i]] != hahmo[i], joten myös fo[i] = f[i].
        }
    }
}
for(j = 0; j < a; j++)                  // Käydään läpi aakkoston jokainen merkki l asettaen
{                                       // aina fr(m, l) = fr(fk(m), l).
    fr[m][j] = fr[f[m]][j];
}
```

Algoritmi 1.3.7:
Reaaliaikaisen korjausfunktion f_r arvojen laskeminen korjausfunktion f_k avulla.

Otetaan nyt aluksi esimerkkinä hyvin yksinkertainen automaatti, jonka kaikkien tilojen joukko on $T = \{t_0, t_1, t_2, t_3\}$, alkutila $T_A = t_0$ ja lopputilojen joukko $T_L = \{t_3\}$, ja jolle on suoraan määritelty ainoastaan merkkijonoa ”abc” täsmäävät siirtymät $S(t_0, \text{'a'}) = t_1$, $S(t_1, \text{'b'}) = t_2$ ja $S(t_2, \text{'c'}) = t_3$, ja edellä tehdyn sopimuksen mukaan kaikissa muissa tapauksissa siirtymäfunktion arvoksi oletetaan alkutila $T_A = t_1$. Tämä automaatti voidaan kuvata seuraavanlaisella graafilla:



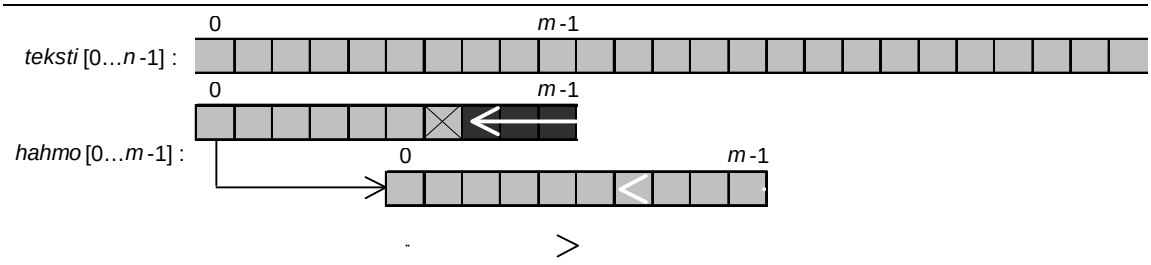
Esitetään nyt sellainen automaatti, joka tunnistaa kaikki hahmon esiintymät merkki kerrallaan luettavasta syötteestä mukaillen reaaliaikaisen merkkijonohakualgoritmin toimintaa. Määritellään tämän automaatin tilojen joukko siten, että tila t_i vastaa tilannetta, jossa syötteestä (rinnastetaan tässä etsinnän kohteena olevaan tekstiin) on tällä hetkellä täsmätty i merkkiä. Tämä tarkoittaa, että luettuaan merkin *teksti*[j] siirtyy automaatti tilasta t_i tilaan t_x jos ja vain jos $x = \max\{k \mid k = 0 \vee hahmo[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$, missä merkkijono *teksti*[0…j] muodostuu järjestyksessä automaatin siihen asti lukemista syötemerkeistä ja merkki *teksti*[j] on näistä viimeisin. Siis siirtymä tilasta t_i tilaan t_x noudattaa lauseen 1.3.3 nojalla reaaliaikaista korjausfunktiota f_r siten, että $x = f_r(i, teksti[j])$, ja siten tämän automaatin siirtymien tulee vastata funktiota f_r.



Kuva 1.3.8:
Hahmon ”entten” pohjalta laskettua reaaliaikaista korjausfunktiota f_r vastaava äärellinen tila-automaatti. Alkutilaan palaavat kaaret on jätetty pois aiemmin mainitun sopimuksen mukaisesti.

1.4 Boyer-Moore

Nimiään kantavassa algoritmissa Boyer ja Moore [Boyer and Moore, 1977] ovat ottaneet sikäli erilaisen lähestymistavan merkkijonosovitukseen, että hahmon merkkejä verrataankin lopusta alkuun. Siis ensimmäinen vertailu tehdään merkkien *hahmo*[*m*-1] ja *teksti*[*m*-1] välillä, tämän täsmätessä vertaillaan merkkejä *hahmo*[*m*-2] ja *teksti*[*m*-2] jne., mutta hahmoa kuitenkin liikutetaan tavalliseen tapaan tekstiin nähden alusta loppuun kuvan 1.4.1 mukaisesti.



Kuva 1.4.1:
Boyer-Moore-algoritmissa hahmoa siirretään tavalliseen tapaan vasemmalta oikealle tekstissä, mutta merkkien vertailut tapahtuvatkin hahmon suhteen lopusta alkuun päin. Kuvan esimerkissä täsmätyt osat on merkitty kuvioinnilla, ja hahmon etenemissuunta tummalla sekä vertailujen etenemissuunta vaalealla nuolella.

Motivaatio tähän vertailusuunnan vaihtamiseen tulee siitä, että hahmoa voidaan yleensä siirtää tekstissä turvallisesti eteenpäin vain sellaisen tekstinosan ohi, joka on jo tutkittu. Siten vasemmalta oikealle verrattaessa siirron pituus on yleensä (esim. Knuth-Morris-Pratt-algoritmissa) maksimissaan *i* merkkiä, missä *i* on alusta alkaen täsmättyjen merkkien lukumäärä (tutkittu osa päättyy merkkiin teksti[*j*+*i*-1], kun hahmo on kohdassa *j* tekstiin nähden, eli kun hahmo[0…*m*-1] = teksti[*j*…*j*+*m*-1]). Oikealta vasemmalle verrattaessa tällainen maksimisiirtymä on sen sijaan yleensä koko hahmon pituuden suuruinen, sillä silloin jo tutkittu tekstinosa päättyy aina merkkiin teksti[$j + m - 1$].

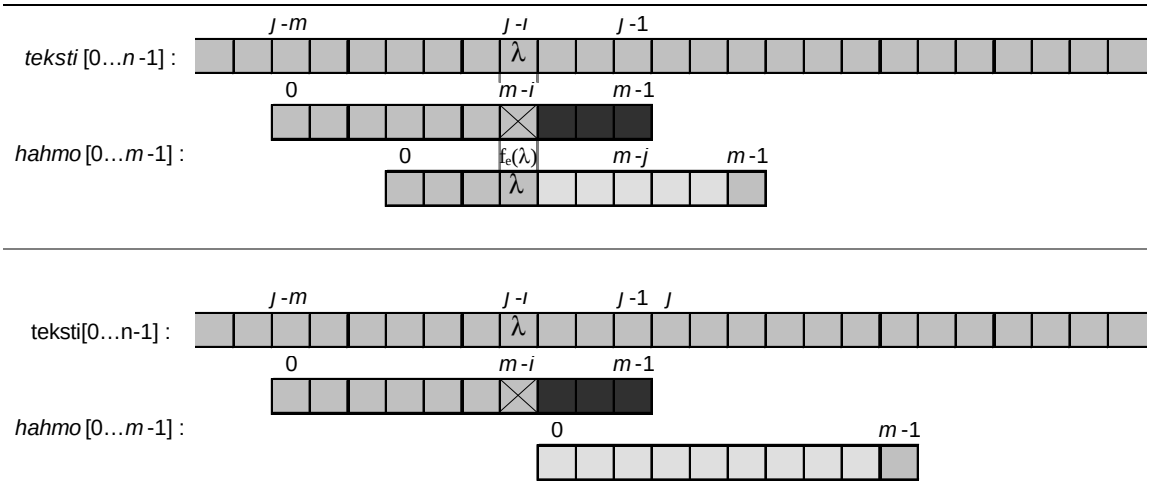
Ei-täsmäävän vertailun sattuessa pyritään siirtämään hahmoa eteenpäin mahdollisimman paljon kahden säännön mukaan, joista on käytetty nimityksiä ”ei-täsmänneen merkin sääntö” (”bad character rule” [Gusfield, 1997] tai ”occurrence heuristic” [Stephen, 1994]) ja ”täsmäävän loppuosan sääntö” (”good suffix rule” [Gusfield, 1997] tai ”match heuristic” [Stephen, 1994]). Alkuperäisessä artikkelissa [Boyer and Moore, 1977] näitä sääntöjä vastaavista siirtymäfunktioista käytettiin yksinkertaisesti nimityksiä $delta_1$ ja $delta_2$, mutta käytän tässä tarkastelussa nimityksiä hyppyfunktio ja siirtymäfunktio. Hahmon siirtämisen jälkeen vertailut aloitetaan aina hahmon viimeisestä merkistä, ja mitään aiemmin havaittuja täsmäävyyksiä ei oteta huomioon.

1.4.1 Ei-täsmänneen merkin sääntö

Ei-täsmänneen merkin säännön ideana on tutkia, missä kohtaa kunkin käytössä olevan aakkoston kirjaimen oikeanpuoleisin esiintymä hahmossa on. Tämän perusteella muodostetaan hyppyfunktio f_e, jonka avulla pyritään ei-täsmäävässä vertailussa olleesta tekstin merkistä riippuen siirtämään hahmoa mahdollisimman paljon eteenpäin. Käytetään merkin λ oikeanpuoleisimman esiintymän indeksistä hahmossa merkintää $R(\lambda)$, ja asetetaan $R(\lambda) = -1$ jos merkki λ ei esiinny hahmossa lainkaan.

Tarkastellaan tilannetta, jossa tapahtuu ei-täsmäävä vertailu merkkien *hahmo*[*m*-*i*] ja *teksti*[*j*-*i*] välillä ja $1 \leq i \leq m \leq j \leq n$. Koska vertailut tapahtuvat hahmon osalta taaksepäin (vasemmalle) alkaen merkistä *hahmo*[*m*-1], on hahmo tällöin siis kohdassa $j - m$ tekstiin nähden. Nyt jos $R($*teksti*[*j*-*i*]$) < m - i$, voidaan hahmoa siirtää $m - i -$ R(*teksti*[*j*-*i*]) merkkiä eteenpäin, jolloin joko keskenään täsmäävät merkit *hahmo*[R(*teksti*[*j*-*i*])] ja *teksti*[*j*-*i*] ovat kohdakkain (kuvan 1.4.2 yläosa) tai sitten kirjain *teksti*[*j*-*i*] ei esiinny hahmossa lainkaan ja hahmo siirrettiin kokonaisuudessaan merkin *teksti*[*j*-*i*] ohi (kuvan 1.4.2 alaosa). Tämä siirto ei hyppää yhdenkään hahmon esiintymän yli, sillä jos hahmoa siirrettäisiin askelkin vähemmän, tapahtuisi merkin *teksti*[*j*-*i*] kohdalla jälleen ei-täsmäävä vertailu. Jos $R($*teksti*[*j*-*i*]$) \geq m - i$, tapahtuisi siirtymä taaksepäin tai jäisi paikalleen, joten tällöin funktiosta $R(\lambda)$ ei ole apua ja hahmoa siirretään yksi askel eteenpäin.

Hahmon siirtymän jälkeen vertailut aloitetaan aina jälleen hahmon viimeisestä merkistä *hahmo*[*m*-1], joten hahmon siirtymän pituuden sijaan hyppyfunktio f_e kannattaa määritellä kertomaan suoraan tekstissä vertailtavana olevan merkin indeksin muutos. Tämän menettelyn mukaisesti funktion f_e arvo saadaan laskemalla yhteen hahmon siirtymä tekstiin nähden ja hahmon sisäinen vertailukohdan siirtymä merkistä *hahmo*[$m - i$] merkkiin *hahmo*[*m*-1], eli $f_e(\lambda) = m - i - R(\lambda) + i - 1 = m - 1 - R(\lambda)$. Tällöin merkkien *hahmo*[*m*-*i*] ja *teksti*[*j*-*i*] välisen ei-täsmäävän vertailun jälkeen verrataan seuraavaksi yksinkertaisesti merkkejä *hahmo*[*m*-1] ja *teksti*[$j-i+f_e($*teksti*[*j*-*i*]$)$].



Kuva 1.4.2:
Tilanne, jossa merkkien *hahmo*[*m-i*] ja *teksti*[*j-i*] = λ välillä tehdään ei-täsmäävä vertailu. Ylemmässä tapauksessa $f_e(\lambda) < m - i$, ja siten hahmoa voidaan siirtää $(m - i) - f_e(\lambda)$ merkkiä eteenpäin, jolloin merkit *teksti*[*j-i*] = λ ja *hahmo*[$f_e(\lambda)$] = λ täsmäävät. Alemmassa tapauksessa $f_e(\lambda) = -1$, eli merkki λ ei esiinny lainkaan hahmossa, ja hahmo voidaan siten siirtää kokonaan merkin *teksti*[*j-i*] = λ oikealle puolelle. Kuvissa tumma kuviointi tarkoittaa täsmättyjä merkkejä, ei-täsmäävästi vertaillun merkin päällä on ristikko, ja vaaleampi värisävy tarkoittaa hahmon osaa, josta tiedetään, että se ei sisällä lainkaan merkkiä λ.

Määritelmä 1.4.3:
Boyer-Moore-algoritmin hyppyfunktio:
$f_e(\lambda) = m - 1 - R(\lambda)$, missä $R(\lambda) = \max\{k \mid k = -1 \vee hahmo[k] = \lambda, 0 \leq k \leq m-1\}$.

Todetaan vielä funktion $f_e(\lambda)$ mukaisten siirtymien laillisuus täsmällisesti:

Lause 1.4.4:
Jos hahmo on kohdassa *j-m* tekstiin nähden ja merkit *hahmo*[*m-i*] ja *teksti*[*j-i*] eivät täsmää, niin hahmoa voidaan siirtää tekstissä eteenpäin arvon f_e(*teksti*[*j-i*]) mukaisesti hyppäämättä yhdenkään hahmon esiintymän yli.

Todistus:
Arvon f_e(*teksti*[*j-i*]) mukainen tekstin vertailukohdan siirto vastaa hahmon siirtämistä f_e(*teksti*[*j-i*]) - *i* + 1 merkillä eteenpäin. Tehdään vastaoletus, että tämä f_e(*teksti*[*j-i*]) - *i* + 1 merkin pituinen siirtymä hyppää jonkin hahmon esiintymän yli. Koska ainoastaan eteenpäin suuntautuvat siirtymät voivat hypätä hahmon esiintymän yli, on tällöin olemassa sellainen pienempi siirtymän pituus *k*, että $0 < k < f_e$(*teksti*[*j-i*]) - *i* + 1 = *m* – *i* – R(*teksti*[*j-*

i]) ≤ *m* ja *hahmo*[0…*m*-1] = *teksti*[*j*-*m*+*k*…*j*-1+*k*]. Edellisen perustella pätee 0 < *k* + *i* ≤ *m*, mistä seuraa, että *hahmo*[*m*-*i*-*k*] = *teksti*[*j*-*i*], ja toisaalta aiemmasta saadaan myös ehto R(*teksti*[*j*-*i*]) < *m* - *i* - *k*. Tämä on ristiriita arvon R(*teksti*[*j*-*i*]) määritelmän kanssa, sillä tässä tapauksessa merkki *teksti*[*j*-*i*] esiintyy hahmossa indeksin *m*-*k*-*i* kohdalla, ja *m*-*k*-*i* > R(*teksti*[*j*-*i*]). Siis vastaoletus on väärä, eli f_e(*teksti*[*j*-*i*]) - *i* + 1 merkkiä pitkä siirtymä ei hyppää yhdenkään hahmon esiintymän yli. Siten lauseen väite on oikea, koska arvon f_e(*teksti*[*j*-*i*]) soveltaminen siirtää hahmoa max{f_e(*teksti*[*j*-*i*]) - *i* + 1, 1} merkkiä eteenpäin.

Käytännössä funktion R(λ) arvoja ei kannata laskea erikseen, sillä yhtä helposti voidaan laskea suoraan funktion f_e arvot. Seuraavassa esitetty alustusalgoritmin koodiesitys toimii siten, että aluksi kaikki funktion f_e arvot alustetaan arvoilla *m* – 1 – (-1) = *m* vastaamaan tapausta, jossa merkki λ ei esiinny hahmossa lainkaan, eli R(λ) = -1. Tämän jälkeen hahmo käydään merkki kerrallaan läpi alusta loppuun siten, että aina merkin *hahmo*[*i*] kohdalla asetetaan f_e(*hahmo*[*i*]) = *m* – 1 – *i*. Toimenpiteen jälkeen arvo f_e(*hahmo*[*i*]) on laskettu oikein, sillä viimeisin kerta, kun arvoa f_e(*hahmo*[*i*]) muutetaan, on kyseisen merkin oikeanpuoleisimman esiintymän indeksin kohdalla hahmossa, eli silloin *i* = R(*hahmo*[*i*]) ja f_e(*hahmo*[*i*]) = *m* – 1 – *i* = *m* – 1 – R(*hahmo*[*i*]).

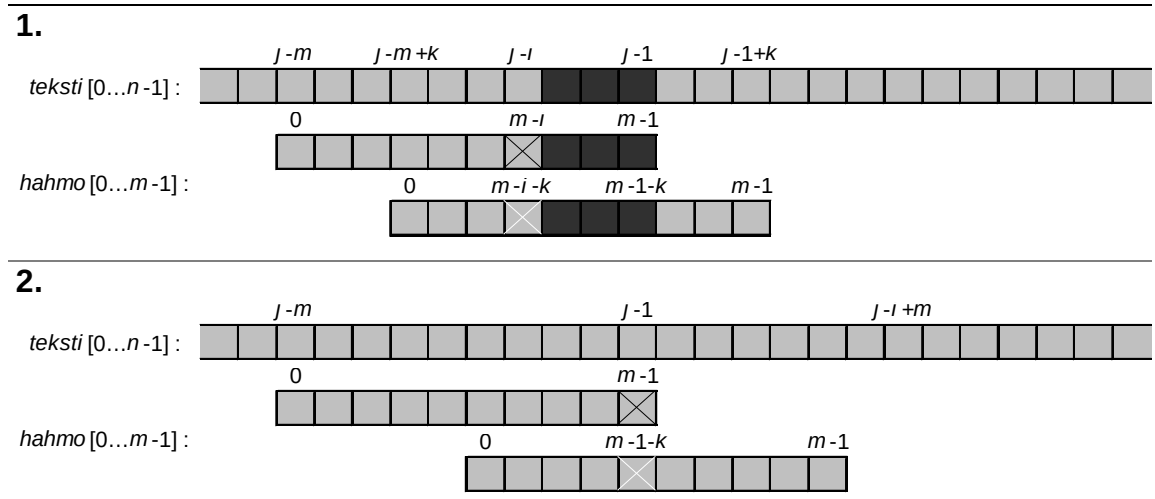
```
int i;                                  // Hahmoa läpikäyvä indeksimuuttuja.
int a;                                  // Käytetyn aakkoston merkkien lukumäärä.
int m;                                  // Hahmon pituus.
int N(λ);            // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
int f_e[a];                             // Hyppyfunktion arvotaulukko, f_e[ l] = f_e( l).

for(i = 0; i < a; i++)                  // Alustetaan kaikki funktion f_h arvot luvulla -1.
{
    f_e[i] = m;
}
for(i = 0; i < m; i++)                  // Käydään hahmon indeksit 0…m-1 läpi asettaen aina
{                                       // merkin hahmo[i] kohdalla f_e[hahmo[i]] = m-1-i, jolloin
    f_e[N(hahmo[i])] = m-1-i;           // lopussa f_e[hahmo[i]] = m-1-R(hahmo[i]).
}
```

Algoritmi 1.4.5:
Boyer-Moore-algoritmin hyppyfunktion alustus.

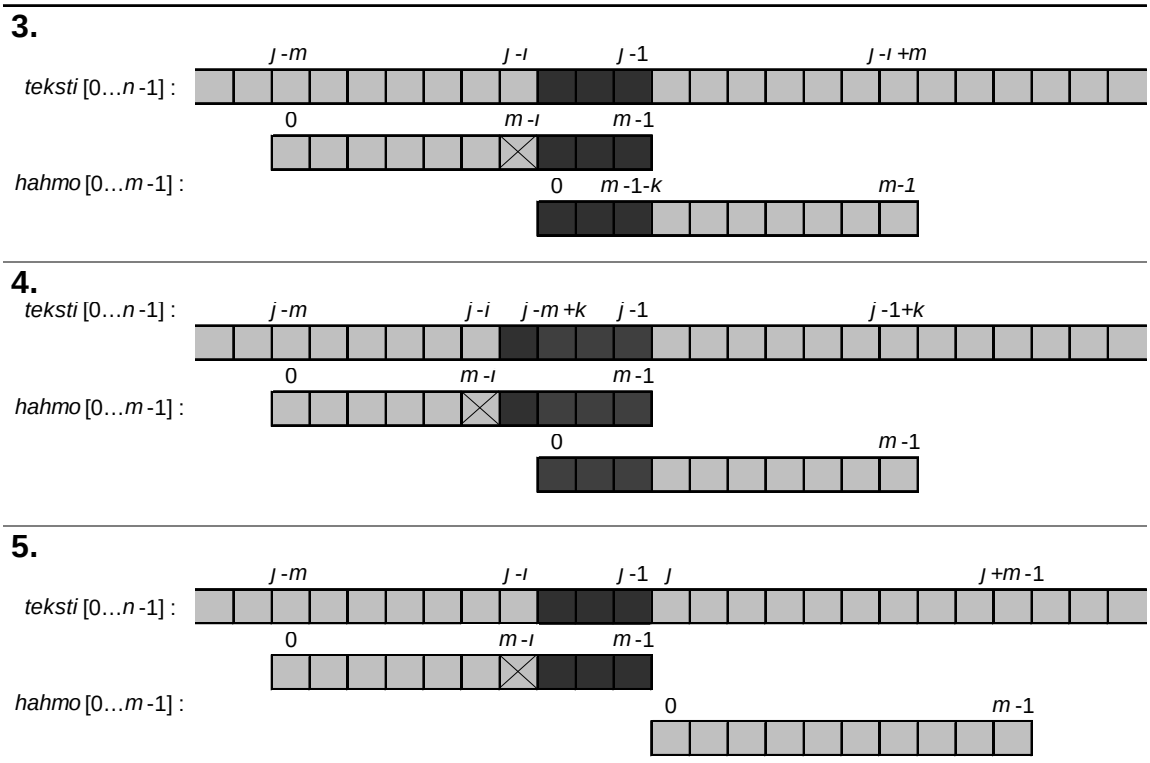
Triviaalisti edellinen algoritmi on aikavaativuudeltaan O(|Λ| + *m*), missä |Λ| on käytetyn aakkoston merkkien lukumäärä. Jos aakkoston merkkien lukumäärä oletetaan äärelliseksi vakioksi, on tämä alustusalgoritmi hahmon pituuden suhteen lineaarinen.

1.4.2 Täsmäävän loppuosan sääntö

Täsmäävän loppuosan säännössä käytettävä siirtymäfunktio on hyvin samantapainen kuin Knuth-Morris-Prattin optimoitu korjausfunktio f_{ko}. Ei-täsmäävän vertailun tapahtuessa merkkien *hahmo*[*m*-*i*] ja *teksti*[*j*-*i*] välillä tarkoituksena on siirtää hahmo seuraavaan sellaiseen tekstin kohtaan, että jo täsmäävästi vertaillun osan *teksti*[*j*-*i*…*j*-1] loppuosa on kohdakkain samanlaisen hahmon osan kanssa ja näitä täsmääviä osia edeltävät merkit ovat erilaisia. Jos hahmosta ei löydy yhtään tällaista kohtaa, voidaan hahmo siirtää tekstissä kokonaan merkin *teksti*[*j*-1] oikealle puolelle, sillä selvästi mikään myöhempi hahmon esiintymä ei voi tällöin sijaita edes osittain tekstin kohdan *teksti*[*j*-*i*…*j*-1] alueella.



Kuva 1.4.6 (jatkuu seuraavalla sivulla):
Järjestyksessä ylhäältä alas esitettynä kaikki viisi erilaista mahdollisuutta sovellettaessa täsmäävän loppuosan sääntöä, kun ei-täsmäävä vertailu on tehty merkkien *hahmo*[*m*-*i*] ja *teksti*[*j*-*i*] välillä:
1. Osajono *teksti*[*j*-*i*+1…*j*-1] = *hahmo*[*m*-*i*+1…*m*-1] esiintyy kokonaisuudessaan myös hahmon aiemmassa kohdassa *hahmo*[*m*-*i*+1-*k*…*m*-1-*k*] siten, että näitä kahta samanlaista aluetta edeltävät merkit *hahmo*[*m*-*i*] ja *hahmo*[*m*-*i*-*k*] ovat keskenään erilaiset (kuvassa merkitty tätä erilaista merkkiä *hahmo*[*m*-*i*-*k*] valkealla yliviivauksella).
2. Ei-täsmäävä vertailu tapahtuu heti ensimmäisen verratun merkin eli merkin *hahmo*[*m*-1] kohdalla, ja merkki *hahmo*[*m*-1-*k*] on oikeanpuoleisin merkin *hahmo*[*m*-1] kanssa erilainen hahmon merkki (erilainen merkki yliviivattu valkealla värillä).



Kuva 1.4.6 (jatkoa):

3. Osajono *teksti*[*j*-*i*+1…*j*-1] = *hahmo*[*m*-*i*+1…*m*-1] esiintyy myös hahmon alkuosana eli *hahmo*[0…*i*-2] = *hahmo*[*m*-*i*+1…*m*-1].
4. Osajonon *teksti*[*j*-*i*+1…*j*-1] = *hahmo*[*m*-*i*+1…*m*-1] loppuosa esiintyy myös hahmon alkuosana siten, että *hahmo*[0…*m*-1-*k*] = *hahmo*[*k*…*m*-1] ja *m*-1 ≥ *k* > *m*-*i*+1. Tämä edellinen täsmäävä osa on merkitty kuvassa vaaleammalla kuviolla.
5. Hahmosta ei löydy sellaista seuraavaa kohtaa, joka voisi olla kokonaan tai osittain tekstinosan *teksti*[*j*-*i*+1…*j*-1] = *hahmo*[*m*-*i*…*m*-1] kanssa kohdakkain ilman, että sillä alueella tapahtuisi ei-täsmäävä vertailu.

Käytetään edellisen periaatteen mukaan toimivasta ei-täsmäävän vertailun jälkeen tehtävän hahmon siirtymän antavasta funktiosta merkintää f_t. Se kannattaa määritellä tässä samalla tavalla kuin funktio f_e eli antamaan suoraan vertailukohdan siirtymän tekstissä. Tällöin funktion f_t arvo on aiempaan tapaan hahmon sisällä tapahtuvan vertailukohdan siirtymän sekä hahmon siirtymän summa.

Määritelmä 1.4.7:
Boyer-Moore-algoritmin siirtymäfunktio:
$f_t(m\text{-}i) = i - 1 + \min\{k \mid k = m \vee (0 < k \leq m\text{-}i \wedge (i = 1 \vee hahmo[m\text{-}i\text{+}1\text{-}k \ldots m\text{-}1\text{-}k] = hahmo[m\text{-}i\text{+}1 \ldots m\text{-}1]) \wedge hahmo[m\text{-}i] \neq hahmo[m\text{-}i\text{-}k]) \vee (m\text{-}i < k \leq m - 1 \wedge hahmo[0 \ldots m\text{-}1\text{-}k] = hahmo[k \ldots m\text{-}1])\}$, kun $1 \leq i \leq m$.

Välillä $1 \leq i \leq m$ arvo $f_t(m\text{-}i)$ määritellään siis pienimmäksi sellaiseksi luvuksi $i - 1 + k$, että luvun k pituinen siirtymä vastaa jotain kuvan 1.4.6 viidestä eri mahdollisuudesta. Tässä luvun k eri vaihtoehtojen vastaavuudet menevät siten, että arvo $k = m$ vastaa kuvan 1.4.6 kohtaa 5, ehto $(0 < k \leq m\text{-}i \wedge (i = 1 \vee hahmo[m\text{-}i\text{+}1\text{-}k \ldots m\text{-}1\text{-}k] = hahmo[m\text{-}i\text{+}1 \ldots m\text{-}1]) \wedge hahmo[m\text{-}i] \neq hahmo[m\text{-}i\text{-}k])$ vastaa kuvan 1.4.6 kohtia 1 ja 2, ja ehto $(m\text{-}i < k \leq m - 1 \wedge hahmo[0 \ldots m\text{-}1\text{-}k] = hahmo[k \ldots m\text{-}1])$ kuvan 1.4.6 kohtia 3 ja 4.

Lauseessa 1.4.8 todetaan siirtymäfunktion f_t antamien hahmon siirtymien toimivuus.

Lause 1.4.8:
Boyer-Moore-algoritmin siirtymäfunktion f_t mukaiset siirtymät eivät hyppää yhdenkään hahmon esiintymän yli tekstissä.

Todistus:
Jotta siirtymä voisi hypätä jonkin hahmon esiintymän yli, pitää sen selvästi olla yli yhden merkin pituinen. Merkkien $hahmo[m\text{-}i]$ ja $teksti[j\text{-}i]$ välisen ei-täsmäävän vertailun jälkeen tehdään määritelmän 1.4.7 nojalla $f_t(m\text{-}i) - i + 1$ merkkiä pitkä siirtymä, joten oletetaan, että $2 \leq f_t(m\text{-}i) - i + 1$, ja tehdään vastaoletus, että tarkastelun kohteena oleva siirtymä hyppää jonkin hahmon esiintymän yli. Tämä tarkoittaa, että on olemassa sellainen kokonaisluku y, että $0 < y < f_t(m\text{-}i) - i + 1$ ja $hahmo[0 \ldots m\text{-}1] = teksti[j\text{-}m\text{+}y \ldots j\text{-}1\text{+}y]$. Koska $f_t(m\text{-}i) - i + 1 \leq m$, tiedetään lisäksi, että $y < m$. Jaetaan tarkastelu kahteen tapaukseen luvun i arvon mukaan:

1) $i = 1$:
Tässä tilanteessa määritelmän 1.4.7 mukaan $f_t(m\text{-}i) - i + 1 = 1 - 1 + \min\{k \mid k = m \vee hahmo[m\text{-}1] \neq hahmo[m\text{-}1\text{-}k]\} - i + 1 = \min\{k \mid k = m \vee hahmo[m\text{-}1] \neq hahmo[m\text{-}1\text{-}k]\}$. Mutta koska $y < m$, pätee $hahmo[m\text{-}1\text{-}y] = teksti[j\text{-}1]$ ja siten $hahmo[m\text{-}1\text{-}y] \neq hahmo[m\text{-}1]$. Tämä on ristiriita, sillä oletuksen mukaan $y < \min\{k \mid k = m \vee hahmo[m\text{-}1] \neq hahmo[m\text{-}1\text{-}k]\}$, mutta vastaoletuksen ja edellisen mukaan y toteuttaa minimilausekkeen

luvulle k asetetut ehdot, josta seuraa ehto $\min\{k \mid k = m \vee hahmo[m\text{-}1] \neq hahmo[m\text{-}1\text{-}k]\} \geq y$.

2) $i > 1$:
Tässä tilanteessa $f_t(m\text{-}i)\ \text{-}i + 1 = \min\{k \mid k = m \vee (0 < k \leq m\text{-}i \wedge hahmo[m\text{-}i\text{+}1\text{-}k \ldots m\text{-}1\text{-}k] = hahmo[m\text{-}i\text{+}1 \ldots m\text{-}1] \wedge hahmo[m\text{-}i] \neq hahmo[m\text{-}i\text{-}k]) \vee (m\text{-}i < k \leq m - 1 \wedge hahmo[0 \ldots m\text{-}1\text{-}k] = hahmo[k \ldots m\text{-}1])\}$. Koska $y < m$, pätee $hahmo[0 \ldots m\text{-}1\text{-}y] = teksti[j\text{-}m\text{+}y \ldots j\text{-}1]$. Yhdistämällä tämä juuri tehtyjen täsmäysten perusteella tiedetyn täsmäävyyden $hahmo[m\text{-}i\text{+}1 \ldots m\text{-}1] = teksti[j\text{-}i\text{+}1 \ldots j\text{-}1]$ kanssa nähdään, että joko $y < m\text{-}i\text{+}1$ ja $hahmo[m\text{-}i\text{+}1\text{-}y \ldots m\text{-}1\text{-}y] = hahmo[m\text{-}i\text{+}1 \ldots m\text{-}1]$ tai sitten $y \geq m\text{-}i\text{+}1$ ja $hahmo[0 \ldots m\text{-}1\text{-}y] = hahmo[y \ldots m\text{-}1]$. Lisäksi tapauksessa $y < m\text{-}i\text{+}1$ on yhtäsuuruus $hahmo[m\text{-}i\text{-}y] = teksti[j\text{-}i]$ voimassa, ja tästä vuorostaan seuraa, että silloin myös $hahmo[m\text{-}i\text{-}y] \neq hahmo[m\text{-}i]$. Edellisistä seuraa määritelmän 1.4.7 mukaan, että $f_t(m\text{-}i)\ \text{-}i + 1 \leq y$, sillä y täyttää määritelmän minimilausekkeessa käytetyn muuttujan k vaatimukset. Näin ollen päädyttiin ristiriitaan oletuksen $y < f_t(m\text{-}i)\ \text{-}\ i + 1$ kanssa.

Kohtien 1) ja 2) perusteella voidaan vastaoletus todeta vääräksi, ja siten lauseen väite on todistettu.

Siirtymäfunktion f_t arvot voidaan laskea lineaarisessa ajassa ainakin kahdella perusperiaatteeltaan erilaisella menetelmällä. Alkuperäinen Boyerin ja Mooren artikkeli ei sisältänyt siirtymäfunktion alustusalgoritmia, ja ensimmäisen lineaarisen algoritmin esitti heidän sijastaan Knuth [Knuth et al., 1977]. Hänen menetelmänsä käyttää apunaan Morris-Pratt-algoritmin korjausfunktiota, mutta tätä toteutusta on luonnehdittu erittäin vaikeaselkoiseksi, ja Gusfield toteaakin, että kirjallisuudessa ei ole esiintynyt hyvää selvitystä menetelmän toiminnasta [Gusfield, 1997]. Asiaa kuvastanee myös se, että aluksi julkaistu versio ei toiminut kaikissa tapauksissa oikein, vaan vasta esimerkiksi Rytter [Rytter, 1980] on julkaissut myöhemmin korjatun version. Gusfield esittää ajatukseltaan toisen, mielestään yksinkertaisemman menetelmän [Gusfield, 1997]. Se ei ole kuitenkaan yhtä kompakti kuin ”alkuperäinen” (ja sittemmin korjattu) Knuthin esittämä alustusalgoritmi. Koska jälkimmäinen on mielestäni kaikesta edellä todetusta huolimatta kuitenkin suhteellisen suoraviivaisesti johdettavissa Morris-Pratt-algoritmin korjausfunktion f_k alustusalgoritmin pohjalta, pyrin tässä selittämään sen toimintaperiaatteen.

Otetaan käyttöön merkintä mj^R tarkoittamaan merkkijonoa mj takaperin, jolloin $|mj^R| = |mj|$ ja $mj^R[i] = mj[m\text{-}1\text{-}i]$, kun $0 \leq i \leq |mj| - 1$. Sovitaan lisäksi, että merkintä f_{kR} viittaa merkkijonoa $hahmo^R$ vastaavaan Morris-Pratt-algoritmin korjausfunktioon. Tällöin siis esimerkiksi määritelmän 1.2.17 mukaan $f_{kR}(i) = \max\{k \mid k \in R_{hahmo^R,\, i-1}\}$, kun $1 \leq i \leq m$.

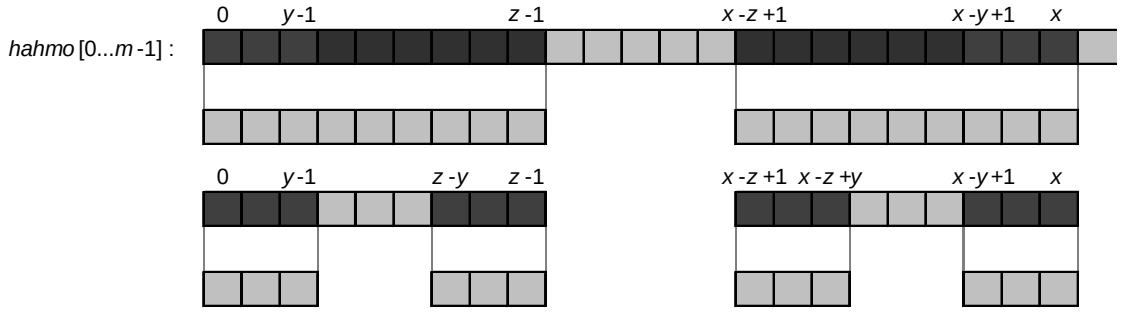
Korvaamalla määritelmässä 1.4.7 merkkijono *hahmo* merkkijonolla $hahmo^R$, saadaan Boyer-Mooren siirtymäfunktion arvo $f_t(m\text{-}i)$, missä $1 \leq i \leq m$, muotoon $f_t(m\text{-}i) = i - 1 + \min\{k \mid k = m \vee (0 < k \leq m\text{-}i \wedge (i = 1 \vee hahmo^R[k\ldots i\text{-}2+k] = hahmo^R[0\ldots i\text{-}2]) \wedge hahmo^R[i\text{-}1] \neq hahmo^R[i\text{-}1+k]) \vee (m\text{-}i < k \leq m - 1 \wedge hahmo^R[0\ldots m\text{-}1\text{-}k] = hahmo^R[k\ldots m\text{-}1])\} = i - 1 + \min\{k \mid k = m \vee (0 < k \leq m\text{-}i \wedge (i\text{-}1) \in R_{hahmo^R,\, i-2+k} \wedge hahmo^R[i\text{-}1] \neq hahmo^R[i\text{-}1+k]) \vee (m\text{-}i < k \leq m - 1 \wedge m\text{-}k \in R_{hahmo^R,\, m-1})\}$. Selvästi ainoa hankala osuus tämän arvon määrittämisessä on siinä esiintyvän minimilausekkeen ratkaiseminen. Keskitytään siis tämän tarkasteluun, ja otetaan käsittelyn ajaksi käyttöön merkintä $min_{m-i} = \min\{k \mid k = m \vee (0 < k \leq m\text{-}i \wedge (i\text{-}1) \in R_{hahmo^R,\, i-2+k} \wedge hahmo^R[i\text{-}1] \neq hahmo^R[i\text{-}1+k]) \vee (m\text{-}i < k \leq m - 1 \wedge m\text{-}k \in R_{hahmo^R,\, m-1})\}$, jolloin $f_t(m\text{-}i) = i - 1 + min_{m-i}$.

Jaetaan luvun min_{m-i} arvon laskeminen kahteen osaan siten, että tarkastellaan erikseen minimilausekkeita $min1_{m-i} = \min\{ k \mid k = m \vee (0 < k \leq m\text{-}i \wedge (i\text{-}1) \in R_{hahmo^R,\, i-2+k} \wedge hahmo^R[i\text{-}1] \neq hahmo^R[i\text{-}1+k])\}$ ja $min2_{m-i} = \min\{k \mid k = m \vee (m\text{-}i < k \leq m - 1 \wedge m\text{-}k \in R_{hahmo^R,\, m-1})\}$. Tällöin siis $min_{m-i} = \min\{min1_{m-i}, min2_{m-i}\}$ ja $f_t(m\text{-}i) = i - 1 + min_{m-i} = i - 1 + \min\{min1_{m-i}, min2_{m-i}\} = \min\{i - 1 + min1_{m-i}, i - 1 + min2_{m-i}\}$.

Aloitetaan käsittely arvosta $min1_{m-i} = \min\{ k \mid k = m \vee (0 < k \leq m\text{-}i \wedge (i\text{-}1) \in R_{hahmo^R,\, i-2+k} \wedge hahmo^R[i\text{-}1] \neq hahmo^R[i\text{-}1+k])\}$. Koska tässä $1 \leq i \leq m$ ja jälkimmäisessä osassa pätee $0 < k \leq m\text{-}i$, on ehto $0 \leq i\text{-}2+k \leq m\text{-}2$ voimassa. Siten selvästi yksi mahdollisuus tämän arvon määrittämiseksi olisi käydä läpi kaikki joukkojen $R_{hahmo^R,\, x}$, missä $x = 0, 1, \ldots, m\text{-}2$, alkiot ja tutkia mikä on pienin sellainen indeksi x, että arvo i-1 sisältyy joukkoon $R_{hahmo^R,\, x}$ ja pätee $hahmo^R[i\text{-}1] \neq hahmo^R[x+1]$, vai löytyykö sellaista indeksiä x lainkaan. Jos tällainen pienin indeksi x ja sitä vastaava joukko $R_{hahmo^R,\, x}$ löytyy, niin tiedetään että $min1_{m-i} = x - (i\text{-}2) = x - (i\text{-}1) + 1$, sillä määritelmän 1.2.1 nojalla tällöin $x - (i\text{-}1) \geq 0$ eli $x - (i\text{-}1) + 1 > 0$ ja pätee $0 < x - (i\text{-}1) + 1 \leq m\text{-}i$, $(i\text{-}1) \in R_{hahmo^R,\, (i\text{-}1)+(x\text{-}(i\text{-}1)+1)}$ sekä $hahmo^R[i\text{-}1] \neq hahmo^R[(i\text{-}1)+x\text{-}(i\text{-}1)+1]$, joten arvo $x - (i\text{-}1) + 1$ täyttää minimilausekkeessa indeksille k asetetut ehdot. Koska x on minimaalinen, ei voi olla myöskään pienempää nämä ehdot täyttävää lukua k, sillä tällöin pätisi $(i\text{-}1) \in R_{hahmo^R,\, (i\text{-}2)+k}$, $hahmo^R[i\text{-}1] \neq hahmo^R[(i\text{-}1)+k]$, ja $k < x - (i\text{-}1) + 1$ eli $(i\text{-}2) + k < x$, mikä on ristiriidassa sen kanssa, että x on pienin indeksi, jolla pätee $(i\text{-}1) \in R_{hahmo^R,\, x}$ ja $hahmo^R[i\text{-}1] \neq hahmo^R[x+1]$. Jos edellä etsitynlaista indeksiä x taas ei löydy lainkaan, eli

joukoista $R_{hahmo^R,\,0}$, $R_{hahmo^R,\,1}$, …, $R_{hahmo^R,\,m-2}$ ei löydy sellaista luvun i-1 sisältävää joukkoa $R_{hahmo^R,\,x}$, että pätisi $hahmo^R[i\text{-}1] \neq hahmo^R[x\text{+}1]$, niin selvästi $min1_{m-i} = m$, sillä tällöin minimilausekkeen $min1_{m-i}$ jälkimmäisen osan ehdot eivät voi olla voimassa millään siinä sallitulla indeksillä k.

Edellä kuvatussa arvojen $min1_{m-i}$ laskemisessa ainoa ongelma on joukkojen $R_{hahmo^R,\,0}$, $R_{hahmo^R,\,1}$, …, $R_{hahmo^R,\,m-2}$ arvojen tutkiminen. Voidaan kuitenkin havaita, että tämä saadaan toteutettua melko helposti funktion f_{kR} avulla. Jos nimittäin tutkitaan jotain joukkoa $R_{hahmo,\,x}$, missä $0 \leq x \leq m$-1, niin voidaan huomata, että jos $z \in R_{hahmo^R,\,x}$ ja $z > 0$, niin seuraavaksi suurin joukkoon $R_{hahmo^R,\,x}$ kuuluva alkio on $f_k(z)$. Kuva 1.4.9 havainnollistaa edellistä päättelyä. Todistetaan asia täsmällisemmin lauseessa 1.4.10.



Kuva 1.4.9:
Jos $y, z \in R_{hahmo,\,x}$ ja $y < z$, niin $y \in R_{hahmo,\,z-1}$, koska tällöin joko $y = 0$ tai sitten $y > 0$ ja merkkijono $hahmo[0\ldots y\text{-}1]$ esiintyy kuvan mukaisesti myös keskenään identtisten merkkijonojen $hahmo[0\ldots z\text{-}1]$ ja $hahmo[x\text{-}z\text{+}1\ldots x]$ alku- sekä loppuosana. Vastaavasti nähdään, että jos $y \in R_{hahmo,\,z-1}$ ja $z \in R_{hahmo,\,x}$, niin tällöin myös $y \in R_{hahmo,\,x}$, sillä joko $y = 0$, tai $y > 0$ ja $hahmo[0\ldots y\text{-}1]$ on merkkijonon $hahmo[0\ldots z\text{-}1]$ alkuosa ja merkkijonon $hahmo[x\text{-}z\text{+}1\ldots x]$ loppuosa eli merkkijonon $hahmo[0\ldots x]$ alku- ja loppuosa. Tässä yhteydessä käytettiin termejä alku- ja loppuosa tarkoittaen aitoja eli koko merkkijonoa lyhyempiä osamerkkijonoja. Edellisestä seuraa, että haettu lukua z pienempi luku y kuuluu joukkoon $R_{hahmo,\,z-1}$, ja toisaalta mikä tahansa joukon $R_{hahmo,\,z-1}$ luku täyttää luvun y vaatimukset. Siten täytyy päteä yhtäsuuruus $y = \max\{k \mid k \in R_{hahmo,\,z-1}\} = f_k(z)$, sillä muuten pätisi $y < f_k(z) < z$ ja $f_k(y) \in R_{hahmo,\,y-1}$, eli y ei olisikaan luvun z jälkeen seuraavaksi suurin joukon $R_{hahmo,\,z-1}$ luku.

Lause 1.4.10:
Jos $z \in R_{hahmo,\,x}$ ja $z > 0$, niin $f_k(z) = \max\{q \mid q < z \wedge q \in R_{hahmo,\,x}\}$.

Todistus:
Jos $z \in R_{hahmo,\,x}$ ja $z > 0$, niin $z \leq x$ ja $hahmo[0\ldots z\text{-}1] = hahmo[x\text{-}z\text{+}1\ldots x]$ ja lisäksi $z > f_k(z) \geq 0$. Jos $f_k(z) = 0$, pätee myös $f_k(z) \in R_{hahmo,\,x}$. Jos taas $z > f_k(z) > 0$, pätee $hahmo[0\ldots f_k(z)\text{-}1] = hahmo[z\text{-}f_k(z)\ldots z\text{-}1] = hahmo[x\text{-}f_k(z)\text{+}1\ldots x]$ eli myös tällöin $f_k(z) \in R_{hahmo,\,x}$. Edellisestä seuraa, että $f_k(z) < z$ ja $f_k(z) \in R_{hahmo,\,x}$, joten $f_k(z) \leq \max\{q \mid q < z \wedge q \in R_{hahmo,\,x}\}$. Oletetaan nyt, että $f_k(z) = \max\{q \mid q < z \wedge q \in R_{hahmo,\,x}\}$, ja tehdään vastaoletus $f_k(z) \neq \max\{q \mid q < z \wedge q \in R_{hahmo,\,x}\}$. Edellisen mukaan $f_k(z) < \max\{q \mid q < z \wedge q \in R_{hahmo,\,x}\}$, jolloin pitäisi olla olemassa jokin sellainen y, että $f_k(z) < y < z$ ja $z \in R_{hahmo,\,x}$. Tällöin pätee $hahmo[0\ldots y\text{-}1] = hahmo[x\text{-}y\text{+}1\ldots x] = hahmo[z\text{-}y\ldots z\text{-}1]$, joten $y \in R_{hahmo,\,z\text{-}1}$. Mutta tämä on ristiriita, sillä $f_k(z) < y$ ja $f_k(z) = \max\{k \mid k \in R_{hahmo,\,z\text{-}1}\}$. Siten vastaoletus on väärä ja lauseen väite tosi.

Lauseen 1.4.10 pohjalta siis tiedetään, että kaikki joukon $R_{hahmo^R,\,x}$ alkiot voidaan käydä läpi suurimmasta alkiosta pienimpään käyttäen funktiota f_{kR}. Aloitetaan kyseisen joukon suurimmasta alkiosta, joka saadaan arvosta $f_{kR}(x\text{+}1)$, ja edetään aina joukon alkiosta seuraavaksi suurimpaan soveltamalla kyseiseen alkioon funktiota f_{kR}, kunnes lopulta saavutaan joukon $R_{hahmo^R,\,x}$ pienimmän alkion 0 kohdalle. Jos tämä tehdään järjestyksessä $x = 0, \ldots, m\text{-}2$, niin aiemman mukaan aina, kun joukosta $R_{hahmo^R,\,x}$ löytyy sellainen luku y, että pätee $hahmo^R[y] \neq hahmo^R[x\text{+}1]$, ja lukua y vastaten ei ole aiemmin vielä löydetty lukua x pienempää tällaista indeksiä, niin $min1_{m\text{-}1\text{-}y} = x - (y\text{-}1)$. Koska $min1_{m\text{-}1\text{-}y} = m$, jos yhtään tällaista lukua x ei ole olemassa, niin käytännössä arvot $min1_{m\text{-}1\text{-}y}$, missä $0 \leq y \leq m\text{-}1$, voidaan laskea alustamalla ensin kaikki nämä arvot luvulla m ja asettamalla sen jälkeen $min1_{m\text{-}1\text{-}y} = \min\{min1_{m\text{-}1\text{-}y},\ x\text{-}(y\text{-}1)\}$, kun on löydetty sellainen lukupari x, y, että $y \in R_{hahmo^R,\,x}$ ja $hahmo^R[y] \neq hahmo^R[x\text{+}1]$. Tämä toimii oikein, sillä aina pätee $x\text{-}(y\text{-}1) < m$, ja lisäksi arvo $x\text{-}(y\text{-}1)$ on sitä pienempi, mitä pienempi indeksin x arvo on, joten lopulta arvo $min1_{m\text{-}1\text{-}y}$ sisältää haluttua indeksiä x vastaavan arvon. On myös mahdollista, että lukua y vastaavaa sopivaa indeksiä x ei löydy lainkaan ja arvoa $min1_{m\text{-}1\text{-}y}$ ei alustuksen jälkeen muuteta, ja silloin tämä alussa annettu arvo $min1_{m\text{-}1\text{-}y} = m$ on oikein.

Tutkitaan nyt sitä tarvitseeko jokaisesta joukosta $R_{hahmo^R,\,x}$ käydä läpi kaikki arvot $f_{kR}(x\text{+}1)$, $f_{kR}(f_{kR}(x\text{+}1))$, …, 0 vai voidaanko tämä tutkiminen jossain tilanteessa keskeyttää jo ennen viimeisen alkion 0 kohtaamista. Oletetaan, että joukot $R_{hahmo^R,\,0}$, $R_{hahmo^R,\,1}$, …, $R_{hahmo^R,\,x\text{-}1}$ on jo käsitelty ja että ollaan alkion y kohdalla joukon $R_{hahmo^R,\,x}$ tutkimisessa, missä $y > 0$. Tällöin

pätee $f_{kR}(y) \geq 0$, ja ehdot $f_{kR}(y) \in R_{hahmo^R,\, x}$ sekä $f_{kR}(y) \in R_{hahmo^R,\, y\text{-}1}$ ovat voimassa. Näin ollen tiedetään, että koska funktio f_{kR} antaa aina kummankin edellä mainitun joukon seuraavaksi pienimmän alkion, niin joukot $R_{hahmo^R,\, x}$ ja $R_{hahmo^R,\, y\text{-}1}$ ovat identtisiä sellaisten alkioiden osalta, jotka ovat arvoltaan pienempiä tai yhtäsuuria kuin $f_{kR}(y)$. Jos nyt sitten pätee $hahmo^R[y] = hahmo^R[x+1]$, niin tiedetään, että joukkoa $R_{hahmo^R,\, x}$ on turha tutkia alkiosta y alaspäin, sillä nämä jäljellä olevat alkiot on tutkittu joukon $R_{hahmo^R,\, y\text{-}1}$ yhteydessä jo aiemmin. Jos z on jokin näistä alkioista ja pätee $hahmo^R[z] \neq hahmo^R[x+1]$, niin aiemmin on pätenyt $hahmo^R[z] \neq hahmo^R[y]$, ja samalla on asetettu $min1_{m\text{-}1\text{-}z} = y$. Ja koska $y < x + 1$, ei joukon $R_{hahmo^R,\, x}$ tutkimisen jatkaminen voi siis enää muuttaa minkään tällaisen luvun $min1_{m\text{-}1\text{-}z}$ arvoa.

Edellisen perusteella voidaan nyt todeta, että jos joukko $R_{hahmo^R,\, 0} = \{0\}$ tutkitaan kokonaan, niin sen jälkeen minkään luvun $min1_{m\text{-}1\text{-}z}$ arvo ei jää muuttamatta, jos aina joukon $R_{hahmo^R,\, x}$, missä $x > 0$, tutkiminen lopetetaan kesken kohdattaessa sellainen alkio y, että pätee $hahmo^R[y] = hahmo^R[x+1]$. Tämä seuraa siitä, että oleellinen asia joukkojen tutkimisessa on arvojen $min1_{m\text{-}1\text{-}z}$ muuttuminen ja edellisen kaltainen keskeytys ei koskaan vaikuta tähän.

Algoritmilla 1.2.15 voidaan suoraviivaisesti toteuttaa edellisessä tarkastelussa kuvailtu joukkojen $R_{hahmo^R,\, 0}$, $R_{hahmo^R,\, 1}$, …, $R_{hahmo^R,\, m\text{-}2}$ tutkiminen. Nimittäin tällöin saavuttaessa sisäsilmukkaan pätee $i = f_{kR}(j^+)$, ja sisäsilmukka käy läpi joukon $R_{hahmo^R,\, j^+\text{-}1}$ alkioita järjestyksessä suurimmasta pienimpään, kunnes joko ne on käyty kokonaan läpi tai sitten vastaan tulee sellainen $i \in R_{hahmo^R,\, j^+\text{-}1}$, että ehto $hahmo^R[i] = hahmo^R[j^+]$ on voimassa. Koska algoritmissa j^+ käy läpi arvot järjestyksessä 1, 2, …, m - 1, tulevat näin käsiteltyä järjestyksessä joukot $R_{hahmo^R,\, 0}$, $R_{hahmo^R,\, 1}$, …, $R_{hahmo^R,\, m\text{-}2}$. Algoritmin 1.2.15 sisäsilmukan alussa pätee aina $i \in R_{hahmo^R,\, j^+\text{-}1}$ ja $hahmo^R[i] \neq hahmo^R[j^+]$, ja aiemman perusteella arvo $min1_{m\text{-}1\text{-}i}$ voidaan tällöin laskea säännön $min1_{m\text{-}1\text{-}i} = \min\{min1_{m\text{-}1\text{-}i},\, j^+ - (i - 1)\}$ mukaan. Algoritmin 1.4.11 laskee tämän periaatteen mukaisesti arvot $f\_t[m\text{-}1\text{-}i] = i - 1 + min1_{m\text{-}1\text{-}i}$ välillä $0 \leq i < m$, mikä vastaa siis arvojen $min1_{m\text{-}i}$ laskemista välillä $1 \leq i \leq m$.

Koska pääsilmukkaan tehdyt muutokset eivät vaikuta oleellisesti algoritmin 1.4.11 aikavaativuuteen algoritmiin 1.2.15 verrattuna ja lisäksi alkuun lisätty silmukka suoritetaan tasan m - 1 kertaa, voidaan myös algoritmin 1.4.11 aikavaativuudeksi todeta $O(m)$.

Algoritmin 1.4.11 ulkoasu poikkeaa jälleen hieman useimmiten kirjallisuudessa esitetystä muodosta (esim. [Stephen, 1994]), mutta niiden toimintalogiikka voidaan havaita samaksi. Oleellisin ero on jälleen sama kuin algoritmin 1.2.15 yhteydessä eli arvon $f_{kR}(1) = 0$ asettaminen suoralla sijoituksella pääsilmukan ulkopuolella.

```
int m;                          // Hahmon pituus.
int f_t[m];                     // Siirtymäfunktion arvotaulukko.
int fkR[m];                     // Morris-Pratt-algoritmin käänteiselle hahmolle lasketun
                                // tavallisen korjausfunktion arvotaulukko.
f_t[m-1] = 1;                   // Jos ei-täsmäävä vertailu tapahtuu heti hahmon viimeisen merkin
                                // kohdalla, siirrytään vain askel eteenpäin. Alustetaan tämä
                                // siirtymäfunktion arvo jo heti tässä alustusalgoritmin ensimmäisen
                                // osan yhteydessä.
hahmoR[m];                      // Hahmon käänteismerkkijono, eli aina hahmo[i] = hahmoR[m-1-i].
int i = 0;                      // Alustetaan hahmon indeksi arvolla i = 0.
int jplus = 1;                  // Muuttuja jplus sisältää arvon j+ = j + 1, ja aina hahmoR[jplus] =
                                // teksti[j].
int apu = 0;                    // Apumuuttuja.

for(apu  = 0; apu < m; apu++)
{
    f_t[apu] = 2*m - apu - 1;   // Alustetaan ensin ft(m-1-i) = m + i - 1 välillä i = 1…m
}                               // vastaten tapausta, että min1m-i = m.
fkR[1] = 0;                     // Etukäteen tiedetään, että fkR(1) = 0;

while(jplus < m)                // Käydään kaikki korjausfunktion indeksit läpi, arvo fkR(j+1)
{                               // lasketaan j+:nnella kierroksella.
    while(i >= 0 && hahmoR[jplus] != hahmoR[i])     // While-silmukka käy indeksillä
    {                                               // i läpi joukkoa Rhahmo, j+-1
        if(f_t[m-1-i] > jplus)                      // järjetyksessä suurimmasta,
        {                                           // eli arvosta fkR(j+) alkaen niin
            f_t[m-1-i] = jplus;                     // kauan, kun hahmoR[i] [1]
        }                                           // hahmoR[j+].
        i = fkR[i];
    }
    i++;
    jplus++;
    fkR[jplus] = i;             // Asetetaan arvo fkR(j+) algoritmin 1.2.15 mukaisesti.
}
```

Algoritmi 1.4.11:
Boyer-Moore-algoritmin siirtymäfunktion alustamisen ensimmäinen osa, eli arvojen $f_t[m\text{-}i] = i - 1 + min1_{m-i}$ laskeminen.

Tarkastellaan nyt aiemmin esitetyistä minimilausekkeista jälkimmäistä, joka oli $min2_{m-i} = \min\{k \mid k = m \vee (m\text{-}i < k \leq m\text{-}1 \wedge m\text{-}k \in R_{hahmo^R,\, m-1})\}$. Koska 0 on aina pienin joukkoon $R_{hahmo^R,\, m-1}$ kuuluva luku, on edellinen lauseke yhtäsuuri arvon $\min\{k \mid m\text{-}i < k \wedge m\text{-}k \in R_{hahmo^R,\, m-1})\}$ kanssa, ja tämä puolestaan saadaan edelleen muotoon $m - \max\{q \mid q < i \wedge q \in R_{hahmo^R,\, m-1}\}$ tekemällä sijoitus $m\text{-}k = q$. Otetaan käyttöön merkintä q_{m-i} tarkoittamaan edellisen kaltaista indeksiä $m\text{-}i$ vastaavaa lukua q, eli $q_{m-i} = \max\{q \mid q < i \wedge q \in R_{hahmo^R,\, m-1}\}$ ja $min2_{m-i} = m - q_{m-i}$. Tämän mukaan arvon $min2_{m-i}$ laskemiseksi riittää etsiä vastaava luku q_{m-i} eli suurin arvoa i pienempi luku q, joka kuuluu joukkoon $R_{hahmo^R,\, m-1}$, ja vähentää tämä q_{m-i} luvusta m.

Koska $f_{kR}(m) = \max\{k \mid k \in R_{hahmo^R, m-1}\}$ ja määritelmän 1.2.2 nojalla $0 \leq \max\{k \mid k \in R_{hahmo^R, m-1}\} \leq m$-1, pätee näin ollen yhtäsuuruus $q_{m-i} = f_{kR}(m)$ kaikilla arvoilla *i*, missä $f_{kR}(m) < i \leq m$. Jos nyt on olemassa sellainen luku *x*, että $x \in R_{hahmo^R, m-1}$ ja $x < f_{kR}(m)$, eikä olemassa yhtään sellaista lukua *y*, että $y \in R_{hahmo^R, m-1}$ ja $x < y < f_{kR}(m)$, niin *x* on ”arvon $f_{kR}(m)$ jälkeen seuraavaksi suurin joukkoon $R_{hahmo^R, m-1}$ kuuluva luku” ja kaikilla sellaisilla arvoilla *i*, missä $0 \leq x < i \leq f_{kR}(m)$, pätee $q_{m-i} = x$. Vastaavasti jos *z* on ”arvon *x* jälkeen seuraavaksi suurin positiivinen joukkoon $R_{hahmo^R, m-1}$ kuuluva luku”, pätee kaikilla arvoilla *i*, missä $0 \leq z < i \leq x$, että $q_{m-i} = z$. Samaa päättelyä voidaan jatkaa niin kauan, kuin löytyy tällainen ”seuraavaksi suurin joukon $R_{hahmo^R, m-1}$ luku”, ja kun lopulta sellaista ei enää löydy, on kaikki arvot q_{m-i} saatu laskettua. Käytännössä tämän toteuttaminen onnistuu kätevästi korjausfunktion f_{kR} avulla, sillä lauseen 1.4.10 perusteella $f_{kR}(x)$ on aina haluttu luvusta *x* seuraavaksi suurin joukon $R_{hahmo^R, m-1}$ luku.

Arvot $min2_{m-i}$ voidaan siten helposti laskea arvojen q_{m-i} pohjalta seuraavasti: Ensin asetetaan $min2_{m-i} = m - q_{m-i} = m - f_{kR}(m)$ kaikilla sellaisilla indekseillä *m-i*, että pätee $f_{kR}(m) < i \leq m$. Tämän jälkeen jatketaan asettamalla $min2_{m-i} = m - q_{m-i} = m - f_k(f_{kR}(m))$ kaikilla indekseillä *m-i*, joilla pätee $0 \leq f_{kR}(f_{kR}(m)) < i \leq f_{kR}(m)$, ja edelleen $min2_{m-i} = m - q_{m-i} = m - f_{kR}(f_{kR}(f_{kR}(m)))$ kaikilla indekseillä *m-i*, joilla pätee $0 \leq f_{kR}(f_{kR}(f_{kR}(m))) < i \leq f_{kR}(f_{kR}(m))$, ja niin edelleen, kunnes lopulta $f_{kR}(\ldots f_{kR}(m)\ldots) = 0$ ja kaikki arvot $min2_{m-i}$ on siten laskettu välillä $0 \leq m - i < m$ eli $1 \leq i \leq m$. Koska algoritmi 1.4.11 alustaa tässä tarvittavan funktion f_{kR} sekä asettaa arvot $f\_t[m-i] = i - 1 + min1_{m-i}$, saadaan arvo $f\_t[m-i] = f_t(m-i)$ laskettua algoritmin 1.4.11 suorituksen jälkeen laskemalla arvo $min2_{m-i}$ ja asettamalla $f\_t[m-i] = \min\{f\_t[m-i], i - 1 + min2_{m-i}\}$ kaikilla indekseillä $0 \leq m - i < m$ eli välillä $1 \leq i \leq m$. Algoritmi 1.4.12 toteuttaa tämän menettelyn.

Algoritmin 1.4.12 aikavaativuus on $O(m)$, sillä alussa $i = m$ ja $fkRx = f_{kR}(m) < m$, jokainen sisäsilmukan suorituskierros pienentää luvun *i* arvoa yhdellä ja vastaavasti jokainen ulkosilmukan kierros pienentää luvun fkRx arvoa vähintään yhdellä, lukujen *i* ja fkRx arvoja ei koskaan kasvateta, ja algoritmi lopettaa, kun $fkRx < 0$ (koska $f_k(0) = -1$, loppuu algoritmin suoritus sen jälkeen, kun arvo $fkRx = 0$ on käsitelty).

```
int m;                              // Hahmon pituus.
int f_t[m];                         // Siirtymäfunktion arvotaulukko.
int fkR[m];                 // Knuth-Morris-Pratt-algoritmin käänteiselle hahmolle lasketun
                            // tavallisen korjausfunktion arvotaulukko, alustettu algoritmilla 1.4.11.
hahmoR[m];                          // Hahmon käänteismerkkijono, eli aina hahmo[i] = hahmoR[m-1-i].
int i = m;                          // Arvo i jota käytetään indeksin m-i laskemisessa, alussa i = m.
int fkRx = fkR[m];                  // Kullakin hetkellä käytetty aito arvon i alaraja, eli alussa f_kR(m).

while(fkRx >= 0)                    // Jatketaan kunnes arvon i alaraja saa arvon 0.
{
    while(i > fkRx)                 // Käydään läpi i:n arvot välillä fkRx < i £ x, missä fkRx = f_kR(x).
    {
        if(f_t[m-i] > i - 1 + m - fkRx) // Päteekö f_t[m-i] > i - 1 + min2_{m-i}?
        {
            f_t[m-i] = i - 1 + m - fkRx; // Kyllä päti, ja siis asetetaan f_t[m-i] = i - 1 +
        }                           // min2_{m-i}.
        i--;                        // Pienennetään i:n arvoa yhdellä.
    }
    fkRx = fkR[fkRx];               // Siirrytään seuraavaan alarajaan, tässä tilanteessa i = fkRx =
}                                   // f_kR(x), joten se on jo valmiiksi uuden ylärajan kohdalla ja
                                    // riittää ainoastaan siirtää alaraja, eli asettaa fkRx =
                                    // f_kR(fkRx).
```

Algoritmi 1.4.12:
Boyer-Moore-algoritmin siirtymäfunktion alustamisen toinen osa, eli lopullisten arvojen $f\_t[m-i] = \min\{i - 1 + min1_{m-i}, i - 1 + min2_{m-i}\} = \min\{f\_t[m-i], i - 1 + min2_{m-i}\} = f_t(m-i)$ asettaminen algoritmin 1.4.11 laskemien arvojen pohjalta.

Boyer-Moore-algoritmi käyttää edellä algoritmien 1.4.11 ja 1.4.12 avulla laskettuja siirtymäfunktioita hyväksi siten, että aina niiden antamista siirtymistä suurempi toteutetaan. Näin ollen algoritmi etenee vertailemalla hahmoa ja senhetkistä tekstinkohtaa oikealta vasemmalle, kunnes joko löydetään kokonainen hahmon esiintymä tai merkkien *hahmo*[*i*] ja *teksti*[*j*] välillä tapahtuu ei-täsmäävä vertailu. Ensimmäisessä tapauksessa kirjataan hahmon esiintymä löydetyksi sekä siirretään hahmoa askel eteenpäin (eli tekstin vertailukohta siirtyy *m* merkkiä), ja jälkimmäisessä tapauksessa tekstin vertailukohtaa siirretään eteenpäin $\max\{f_e(teksti[j]), f_t(i)\}$ merkkiä ja vertailut aloitetaan hahmon osalta sen viimeisestä merkistä alkaen taaksepäin. Tätä menettelyä toistetaan, kunnes koko teksti on käyty läpi, ja siten Boyer-Moore-algoritmin etsintävaihe voidaan toteuttaa algoritmin 1.4.13 mukaisesti.

Boyer-Moore-algoritmissa funktiot f_e ja f_t nopeuttavat hahmon siirtämistä tekstissä, mutta ne eivät muista aiempien vertailujen perusteella, mikä osa tekstiä tiedetään jo jonkin hahmon osan kanssa täsmääväksi. Näin ollen algoritmi joutuu aina hahmon esiintymän kohdalla vertailemaan kaikki *m* hahmon merkkiä aiemmissa kohdissa tehdyistä vertailuista riippumatta. Tämän pohjalta saadaan helposti osoitettua, että pahimmillaan algoritmi toimii ajassa $O(n \times m)$.

Yksi esimerkki tällaisesta tilanteesta saadaan, kun *hahmo*[0…*m*-1] = ”atat…at” ja *teksti*[0…*n*-1] = ”atat…at”, missä tekstin pituus *n* oletetaan huomattavasti suuremmaksi kuin hahmon pituus *m* ja luonnollisesti sekä *m* että *n* oletetaan tässä tapauksessa parillisiksi. Tällöin tekstissä on (*n* – *m*) / 2 + 1 hahmon esiintymää, joten Boyer-Moore-algoritmi tekee ne kaikki tutkiessaan yhteensä ((*n* – *m*) / 2 + 1)×*m* = (*n*×*m* – m^2) / 2 + *m* vertailua, joka on asymptoottisesti verrannollinen lukuun *n*×*m*. Tällainen pahimman tapauksen tarkastelu ei kuitenkaan anna hyvää kuvaa Boyer-Moore-algoritmin käytännöllisestä aikavaativuudesta, sillä algoritmin voidaan osoittaa toimivan tarkemmin ottaen ajassa O(*n* + *k*×*m*), missä *k* on tekstissä olevien hahmon esiintymien lukumäärä (esim. [Cole, 1994]). Lisäksi Boyer-Moore-algoritmi on käytännössä useimmissa tapauksissa nopeampi kuin lineaarinen ja siten teoriassa nopeampi Knuth-Morris-Pratt-algoritmi (esim. [Stephen, 1994]).

```
int a;          //   Käytetyn aakkoston merkkien lukumäärä.
int m;      //   Hahmon pituus.
int N( l);  //   Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
int f_e[a];//   Hyppyfunktion arvotaulukko, f_e[ l] = f_e( l).
int j = m – 1;  // Tekstin vertailukohta, alustetaan arvolla m-1.
int i = m – 1; // Hahmon vertailukohta, alustetaan viimeisen merkin kohdalle.
while(i < n – 1)   // Käydään koko teksti läpi.
{
    while(i >= 0 && hahmo[i] == teksti[j])
    {
        i--;
        j--;
    }
    if(i < 0)
    {
        loytyi();   // Hahmon esiintymä löytyi.
        j = j + m + 1;// Siirretään hahmoa askel eteenpäin, tekstin vertailukohta
                            // siirtyy m askelta (tässä otetaan huomioon ”ylimääräinen” j--).
    }
    else
    {
        if(f_e[N(teksti[j])] > f_t[i]) // Tutkitaan kumpi funktioista ft ja fe antaa
        {                                          //  suuremman siirtymän merkkien hahmo[i]
            j = j + f_e[N(teksti[j])];// ja teksti[j] kohdalla, ja suoritetaan tämä
        }                                          // kyseinen siirtymä.
        else
        {
            j = j + f_t(i);
        }
    }
}
```

Algoritmi 1.4.13:
Boyer-Moore-merkkijonohaku**.**

1.5 Boyer-Moore-Horspool ja Sundayn Quick Search

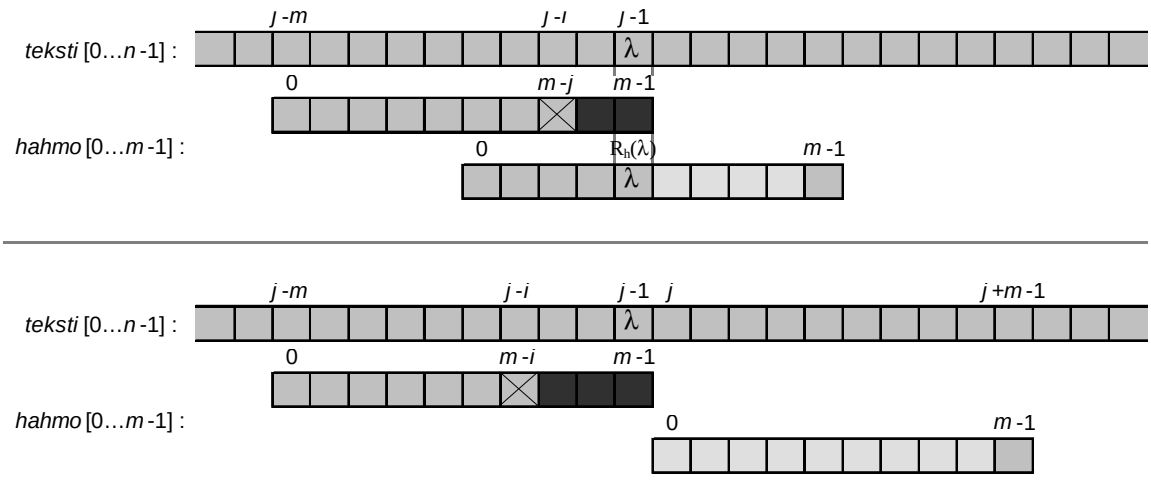
Alkuperäinen Boyer-Moore-algoritmi ei ole aina käytännöllinen, sillä funktion f_t alustusalgoritmi on turhan monimutkainen, ja siten kynnys algoritmin käyttöönottoon voi olla korkea. Lisäksi siinä joudutaan tekemään jokaisen siirtymän yhteydessä hieman lisätyötä, kun pitää aina päättää erikseen, toteutetaanko funktion f_t vai funktion f_e antama siirtymä. Horspoolin [Horspool, 1980] esittämä muunnelma tästä algoritmista sitä vastoin on yleisesti katsottu yhdeksi käytännöllisimmistä ja nopeimmista merkkijonohakualgoritmeista tavallisimmissa sovelluksissa (esim. [Stephen, 1994]), ja se on lisäksi käytännön toteutukseltaan hyvin yksinkertainen. Hänen versionsa käyttää pelkästään ei-täsmänneen merkin sääntöä, mutta sillä erotuksella alkuperäiseen, että nyt siirtymän suuruuden määrittämiseksi tarkastellaankin aina hahmon viimeisen merkin kohdalla olevaa tekstin merkkiä, riippumatta siis siitä, kuinka paljon hahmosta oli jo ehditty vertailla (kuva 1.5.3). Käytetään merkintää $R_h(\lambda)$ merkin λ oikeanpuoleisimmasta esiintymäkohdasta merkkijonossa *hahmo*[0…*m*-2], ja asetetaan $R_h(\lambda) = -1$, jos merkkiä $R_h(\lambda)$ ei ole lainkaan kyseisessä merkkijonossa. Tarkastellaan nyt tilannetta, jossa merkit *hahmo*[*m*-1] ja *teksti*[*j*-1] ovat vastakkain. Tällöin on selvää, että hahmoa voidaan siirtää *m* - 1 - R_h(*teksti*[*j*-1]) merkkiä eteenpäin hyppäämättä jonkin hahmon esiintymän yli, sillä tällöin merkit *hahmo*[R_h(*teksti*[*j*-1])] ja *teksti*[*j*-1] ovat kohdakkain tai siirryttiin kokonaan merkin *hahmo*[*j*-1] ohi, ja askeltakin pienemmän siirtymän jälkeen tehtäisiin merkin *teksti*[*j*-1] kohdalla heti ei-täsmäävä vertailu. Tässä yhteydessä tekstin vertailukohdan kokonaissiirtymä on (*hahmon siirtymä*) + (*hahmon sisäisen vertailukohdan siirtymä*), eli jos viimeisin hahmosta verrattu merkki on *hahmo*[*m-i*], siirretään edellisen periaatteen mukaan tekstin vertailukohtaa kokonaisuudessaan *m* - 1 - R_h(*teksti*[*j*-1]) + *i* - 1 merkkiä. Koska siirtymän suuruus riippuu nyt indeksistä *i*, ei Boyer-Moore-Horspool-algoritmin hyppyfunktiota määritellä funktion f_e tavoin antamaan suoraan tekstin vertailukohdan siirtymää, vaan se antaa hahmon siirtymän, joka on aina *m* - 1 - $R_h(\lambda)$ merkkiä merkin λ ollessa kohdakkain merkin *hahmo*[*m*-1] kanssa tekstissä.
Käytetään tämän hahmon siirtymän antavasta hyppyfunktiosta merkintää f_{eh}, jolloin se siis määritellään seuraavasti:

Määritelmä 1.5.1:
Boyer-Moore-Horspool-algoritmin hyppyfunktio:
$f_{eh}(\lambda) = m - 1 - R_h(\lambda)$, missä $R_h(\lambda) = \max\{k \mid k = -1 \vee hahmo[k] = \lambda, 0 \le k \le m-2\}$.

Todetaan vielä lyhyesti funktion $f_{eh}(\lambda)$ toimivuus lauseessa 1.5.2.

Lause 1.5.2:
Jos hahmo on kohdassa *j*-*m* tekstiin nähden ja merkit *hahmo*[*m*-*i*] ja *teksti*[*j*-*i*] eivät täsmää, niin hahmoa voidaan siirtää tekstissä vähintään f_{eh}(*teksti*[*j*-1]) merkkiä eteenpäin hyppäämättä yhdenkään hahmon esiintymän yli.

Todistus:
Tehdään vastaoletus, että tällainen f_{eh}(*teksti*[*j*-*i*]) merkkiä pitkä siirtymä hyppää jonkin hahmon esiintymän yli. Tämä tarkoittaa, että on olemassa sellainen siirtymän pituus *k*, että $0 < k < f_{eh}(teksti[j-i]) = m - 1 - R_h(teksti[j-i]) \leq m$ ja *hahmo*[0…*m*-1] = *teksti*[*j*-*m*+*k*…*j*-1+*k*]. Edellisestä seuraa suoraan, että *hahmo*[*m*-1-*k*] = *teksti*[*j*-1] ja R_h(*teksti*[*j*-*i*]) < *m* - 1 - *k*, mikä on ristiriita arvon R_h(*teksti*[*j*-*i*]) määritelmän kanssa, sillä *m* - 1 - *k* ≤ *m* - 2. Siis vastaoletus on väärä, ja lauseen väite on oikea.



Kuva 1.5.3:
Tilanne, jossa merkkien *hahmo*[*m*-*i*] ja *teksti*[*j*-*i*] välillä tehdään ei-täsmäävä vertailu ja hahmon viimeinen merkki *hahmo*[*m*-1] on kohdakkain merkin *teksti*[*j*-1] = λ kanssa. Ylemmässä tapauksessa $R_h(\lambda) \geq 0$, ja siten hahmoa voidaan siirtää (*m* - 1) - R_h(λ) merkkiä eteenpäin, jolloin merkit *teksti*[*j*-*i*] = λ ja *hahmo*[R_h(λ)] = λ täsmäävät. Alemmassa tapauksessa $R_h(\lambda) = -1$, eli merkki λ ei esiinny lainkaan merkkijonossa *hahmo*[0…*m*-2], ja hahmoa voidaan siten siirtää *m* merkkiä eteenpäin, kokonaan merkin *teksti*[*j*-1] = λ oikealle puolelle. Kuvissa tumma kuviointi tarkoittaa täsmättyjä merkkejä, ei-täsmäävästi vertaillun merkin päällä on ristikko ja vaaleampi värisävy tarkoittaa hahmon osaa, joka ei sisällä lainkaan merkkiä λ.

Hyppyfunktion f_{eh} voidaan todeta toimivan keskimäärin tehokkaammin kuin funktion f_e, jos tekstin ja hahmon merkkien jakaumat oletetaan satunnaisiksi, sillä selvästi aina $f_{eh}(\lambda) \geq f_e(\lambda)$. Funktion f_{eh} alustaminen voidaan toteuttaa täysin samalla periaatteella kuin aiemman hyppyfunktion f_e alustaminenkin toteutettiin algoritmissa 1.4.5, ainoa muutos on merkin *hahmo*[*m*-1] tutkimatta jättäminen.

```
int i;                                  // Hahmoa läpikäyvä indeksimuuttuja.
int a;                                  // Käytetyn aakkoston merkkien lukumäärä.
int m;                                  // Hahmon pituus.
int N(λ);          // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
int f_eh[a];                        // Hyppyfunktion arvotaulukko, f_eh[i] = f_eh(i).

for(i = 0; i < a; i++)                  // Alustetaan kaikki funktion f_eh arvot luvulla m
{                                   // vastaamaan tapausta, jossa merkki ei esiinny lainkaan
    f_eh[i] = -1;                   // merkkijonossa hahmo[0...m-2].
}
for(i = 0; i < m-1; i++)                // Käydään hahmon indeksit 0…m-2 läpi asettaen aina
{                                       // merkin hahmo[i] kohdalla f_eh[hahmo[i]] = i, jolloin
    f_q[N(hahmo[i])] = i;               // lopussa f_eh[hahmo[i]] = max{k | k = -1 Ú
}                                       // hahmo[k] = hahmo[i]}.
```

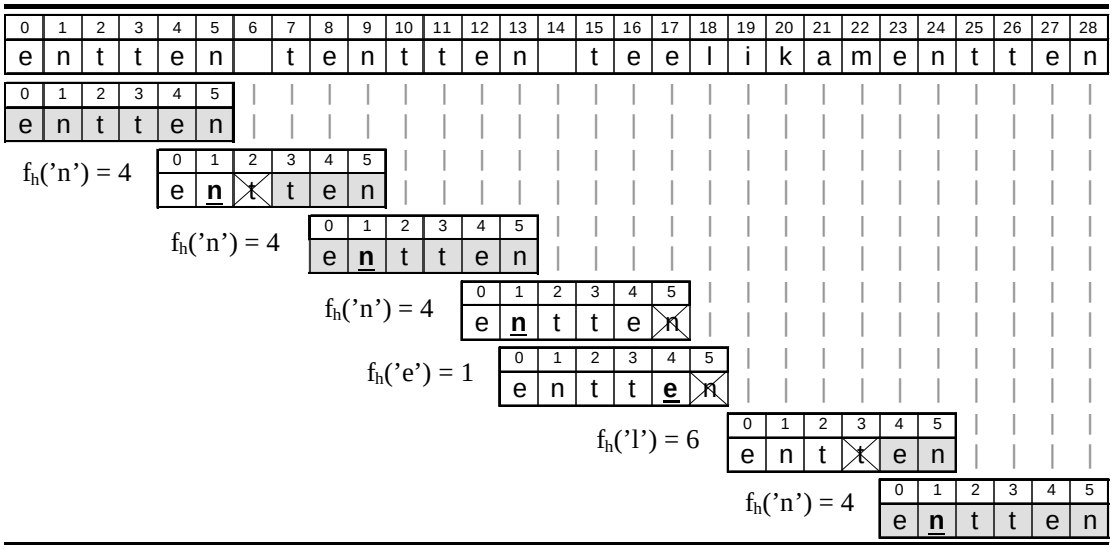
Algoritmi 1.5.4:
Boyer-Moore-Horspool-algoritmin hyppyfunktion alustus.

Algoritmin 1.5.4 aikavaativuus on selvästi sama kuin hyppyfunktion alustusalgoritmin 1.4.5, eli $O(|\Lambda| + m)$, missä $|\Lambda|$ on käytössä olevan aakkoston merkkien lukumäärä.

Boyer-Moore-Horspool-algoritmi (algoritmi 1.5.5) on hyvin samankaltainen kuin alkuperäinen Boyer-Moore-algoritmi, ja pahimmassa tapauksessa sen aikavaativuus on $O(m \times n)$.

```
int m;                              //  Hahmon pituus.
int j = m;                          //  Hahmon viimeinen merkki on aina merkin teksti[j-1] kohdalla.
int i = 0;                          //  Hahmoa läpikäyvä indeksi.
int a;                              //  Käytetyn aakkoston merkkien lukumäärä.
int N(λ);                   //  Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
int f_e[a]; //  Hyppyfunktion arvotaulukko, f_e[i] = f_e(i).

while(j <= n) // Käydään koko teksti läpi.
{
    i = 1;                          //  Hahmon vertailut aloitetaan sen viimeisestä merkistä m-i = m-1.
    while(i <= m && hahmo[m-i] == teksti[j–i])
    {
        i++;
    }
    if(i > m)
    {
        loytyi();                   // Hahmon esiintymä löytyi.
    }
    j = j + f_e[N(teksti[j-1])];    // Siirretään hahmoa hyppyfunktion mukaisesti.
}
```

Algoritmi 1.5.5:
Boyer-Moore-Horspool-algoritmi.



Kuva 1.5.6:

Esimerkkinä merkkijonon *hahmo*[0...5] = "tentten" etsiminen merkkijonosta *teksti*[0...28] = "entten tentten teelikamentten" Boyer-Moore-Horspool-algoritmia käyttäen. Hahmo siirretään joko seuraavaan sellaiseen kohtaan, missä hahmon merkki täsmää tekstissä hahmon viimeisen merkin kohdalla olevan merkin kanssa, tai tällaisen ollessa mahdotonta kokonaan kyseisen merkin ohi. Kuvassa täsmäävästi vertaillut hahmon merkit on tummennettu, ei-täsmäävästi vertaillut merkit on yliviivattu ja siirtymäperusteena toiminut hahmon merkki on lihavoitu sekä alleviivattu. Lisäksi jokaisen siirtymän yhteydessä on mainittu siinä sovellettu hyppyfunktion arvo eli hahmon siirtymän pituus.Yhteensä tässä esimerkissä tehdään 27 kappaletta vertailuja.

Sunday [Sunday, 1990] on kehittänyt Boyer-Moore-Horspool-algoritmista periaatteessa hieman parannellun version. Oletetaan jälleen, että hahmo on merkin *teksti*[*j*-*m*] kohdalla. Sundayn versio toimii Horspoolin algoritmin kanssa melkein identtisesti, mutta siinä hyppyfunktiota sovelletaankin merkin *hahmo*[*m*-1] kohdalla olevan merkin *teksti*[*j*-1] sijaan sitä seuraavaan tekstin merkkiin, joka on *teksti*[*j*] (kuva 1.5.9). Tässä yhteydessä käytettävä hyppyfunktio tarkastelee alkuperäisen Boyer-Moore-algoritmin hyppyfunktion f_e tapaan koko hahmoa, eli se määrää mahdollisimman pitkän turvallisen hahmon siirtymän funktion R(λ) avulla. Siirtymän periaate on lähes identtinen kuin Boyer-Moore-Horspool-algoritmissa, mutta nyt asetetaan merkit *teksti*[*j*] = *hahmo*[R(*teksti*[*j*])] kohdakkain, jolloin hahmoa tulee siirtää *m* - R(*teksti*[*j*]) merkkiä eteenpäin. Koska myös nyt tekstin vertailukohdan siirtymä riippuu viimeksi vertaillun hahmon merkin indeksistä, kertoo Sundayn hyppyfunktio hahmon siirtymän ja on siten melkein kuin funktioiden f_e ja f_{eh} risteytys. Käytetään tästä Sundayn hyppyfunktiosta merkintää f_q, jolloin sen määritelmä on seuraavanlainen:

Määritelmä 1.5.7:
Sundayn Quick Search -algoritmin hyppyfunktio:
$f_q(\lambda) = m - R(\lambda)$, missä $R(\lambda) = \max\{k \mid k = -1 \vee hahmo[k] = \lambda, 0 \le k \le m-1\}$.

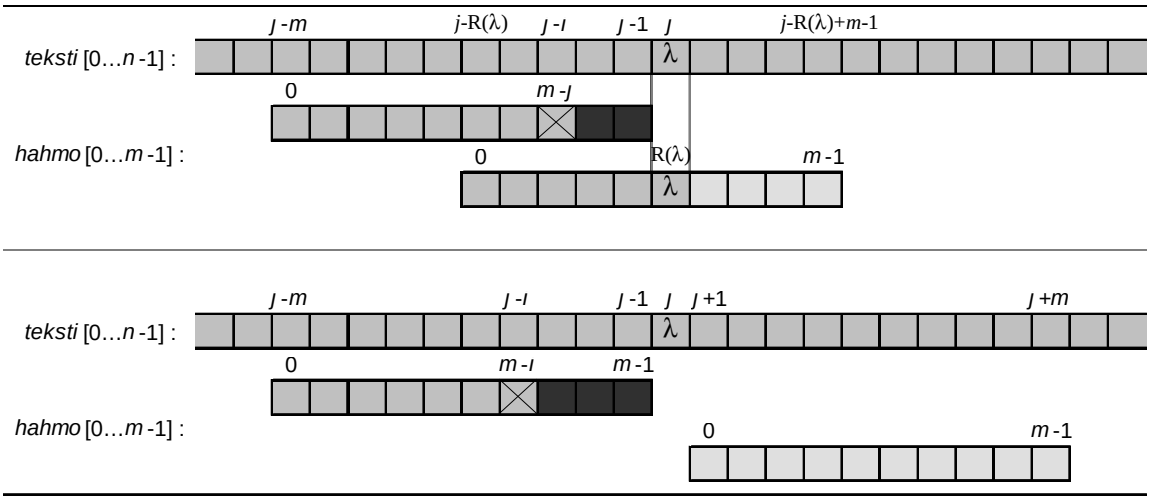
Funktion f_q alustaminen voidaan selvästi toteuttaa täysin samalla periaatteella kuin aiemman hyppyfunktion f_e alustaminenkin toteutettiin:

```
int i;                                  // Hahmoa läpikäyvä indeksimuuttuja.
int a;                                  // Käytetyn aakkoston merkkien lukumäärä.
int m;                                  // Hahmon pituus.
int N(λ);          // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
int f_q[a];                             // Hyppyfunktion arvotaulukko, f_q[i] = f_q(i).

for(i = 0; i < a; i++)                  // Alustetaan kaikki funktion f_q arvot luvulla m - (-1).
{
    f_q[i] = m + 1;
}
for(i = 0; i < m; i++)                  // Käydään koko hahmo läpi asettaen aina merkin
{                                       // hahmo[i] kohdalla f_q[hahmo[i]] = m - i, jolloin
    f_q[N(hahmo[i])] = m - i;           // lopussa f_q[hahmo[i]] = m - max{k | k = -1 Ú
}                                       // hahmo[k] = hahmo[i]}.
```

Algoritmi 1.5.8:
Sundayn Quick Search -algoritmin hyppyfunktion alustus.

Algoritmin 1.5.8 aikavaativuus on selvästi sama kuin hyppyfunktion alustusalgoritmin 1.4.5, eli $O(|\Lambda| + m)$.

Funktion f_q arvoille pätee $f_q(\lambda) \geq f_{eh}(\lambda) + 1$, joten Sundayn algoritmin voisi odottaa toimivan keskimäärin hieman nopeammin kuin Boyer-Moore-Horspool-algoritmi. Parhaassa tapauksessa päästään jo yli hahmon pituiseen eli $m + 1$ merkkiä pitkään hahmon siirtymään, mikäli merkki *teksti*[j] ei esiinny hahmossa lainkaan. Kuva 1.5.9 havainnollistaa hyppyfunktion f_q toimintaa.



Kuva 1.5.9:
Tilanne, jossa merkkien *hahmo*[m-i] ja *teksti*[j-i] välillä tehdään ei-täsmäävä vertailu ja hahmon viimeistä merkkiä seuraa tekstissä merkki *teksti*[j] = λ. Ylemmässä tapauksessa $R(\lambda) \geq 0$, ja siten hahmoa voidaan siirtää m - $R(\lambda)$ merkkiä eteenpäin, jolloin merkit *teksti*[j] = λ ja *hahmo*[$R(\lambda)$] = λ täsmäävät. Alemmassa tapauksessa $R(\lambda) = -1$, eli merkki λ ei esiinny lainkaan hahmossa, ja se voidaan siten siirtää $m + 1$ merkkiä eteenpäin, kokonaan merkin *teksti*[j-1] = λ oikealle puolelle. Kuvissa tumma kuviointi tarkoittaa täsmättyjä merkkejä, ei-täsmäävästi vertaillun merkin päällä on ristikko, ja vaaleampi värisävy tarkoittaa hahmon osaa, joka ei sisällä lainkaan merkkiä λ.

Quick Search -algoritmin periaatteen toimivuus on ilmeinen, mutta todistetaan se vielä seuraavassa lauseessa.

Lause 1.5.10:
Jos hahmo on kohdassa j-m tekstiin nähden ja merkit *hahmo*[m-i] ja *teksti*[j-i] eivät täsmää, niin hahmoa voidaan siirtää tekstissä f_q(*teksti*[j]) merkkiä eteenpäin hyppäämättä yhdenkään hahmon esiintymän yli.

Todistus:
Tehdään vastaoletus, että tämä f_q(*teksti*[j]) merkin pituinen siirtymä hyppää jonkin hahmon esiintymän yli. Tällöin on olemassa sellainen k merkkiä pitkä siirtymä, että $0 < k <$

$f_q(teksti[j]) = m - R(teksti[j]) \leq m + 1$ ja $hahmo[0\ldots m\text{-}1] = teksti[j\text{-}m+k\ldots j\text{-}1+k]$. Näin ollen pätee myös $0 \leq m - k \leq m - 1$ ja siten $hahmo[m\text{-}k] = teksti[j]$. Tämä on ristiriita funktion $R(\lambda)$ määritelmän kanssa, sillä aiemman mukaan $m - k > R(teksti[j])$. Siis vastaoletus voidaan todeta vääräksi, joten Quick Search-algoritmi ei hyppää yhdenkään hahmon esiintymän ohi.

Seuraava Quick Searchin esitys saadaan hyvin pienillä muutoksilla algoritmista 1.5.5.

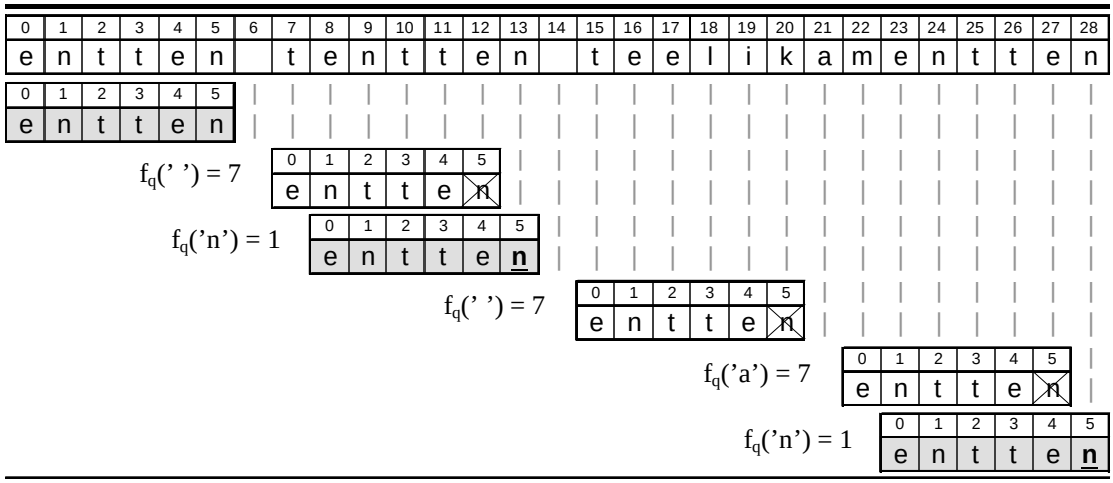
```
int m;                                  // Hahmon pituus.
int j = m;                              // Hahmon viimeinen merkki on aina merkin teksti[j-1] kohdalla.
int i = 0;                              // Hahmoa läpikäyvä indeksi.
int a;                                  // Käytetyn aakkoston merkkien lukumäärä.
int N(λ);                       // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
int f_q[a];                             // Hyppyfunktion arvotaulukko, f_q[N( l)] = f_q( l).

while(j <= n)                           // Käydään koko teksti läpi.
{
    i = 1;                              // Hahmon vertailut aloitetaan sen viimeisestä merkistä m - i = m-1.
    while(i <= m && hahmo[m-i] == teksti[j – i])
    {
        i++;
    }
    if(i > m)
    {
        loytyi();                       // Hahmon esiintymä löytyi.
    }
    if(j < n)                           // Onko hahmon viimeisen merkin jälkeen tekstissä vielä merkki?
    {
        j = j + f_q[N(teksti[j])];      // Siirretään hahmoa hyppyfunktion mukaisesti.
    }
}
```

Algoritmi 1.5.11:
Sundayn Quick Search –algoritmi.

Quick Search -algoritmin aikavaativuus on pahimmassa tapauksessa $O(m \times n)$. Käytännössä kuitenkin Quick Search- ja Boyer-Moore-Horspool-algoritmit toimivat huomattavasti nopeammin kuin esimerkiksi aikavaativuuden $O(m)$ omaava Knuth-Morris-Pratt-algoritmi, ja lisäksi algoritmien käytännön toteutuksen helppouden ansiosta niiden käyttökynnys on matala ([Lecroq, 1995], [Sunday, 1990]).

Quick Search -algoritmi on lisäksi sikäli mielenkiintoinen, että siinä ei ole hahmon siirtymien suorittamisen kannalta mitään merkitystä sillä, missä järjestyksessä hahmon merkkien vertailut suoritetaan. Tämä mahdollistaa esimerkiksi sen, että vertailut suoritetaan hahmossa

alusta loppuun, ja Quick Search-algoritmin hyppyfunktion f_q sovellus yhdistetään Knuth-Morris-Pratt-algoritmin korjausfunktion f_{ko} kanssa [Sunday, 1990]. Jos tällöin suoritetaan aina näiden kahden funktion antamista siirtymistä suurempi ja otetaan funktion f_{ko} mukaisen siirtymän jälkeen huomioon jo aiemmin täsmätty tekstinosa kuten Knuth-Morris-Pratt-algoritmissa, saadaan algoritmi, jossa yhdistyy Quick Search -algoritmin tehokkuus käytännön tilanteissa sekä Knuth-Morris-Pratt-algoritmin lineaarisuus pahimmassakin tapauksessa.



Kuva 1.5.12:
Esimerkkinä merkkijonon *hahmo*[0...5] = "tentten" etsiminen merkkijonosta *teksti*[0...28] = "entten tentten teelikamentten" Quick Search -algoritmia käyttäen. Hahmo siirretään joko seuraavaan sellaiseen kohtaan, missä hahmon merkki täsmää tekstissä hahmon viimeisen merkin perässä olevan merkin kanssa, tai sitten kokonaan kyseisen merkin ohi tällaisen ollessa mahdotonta. Kuvassa täsmäävästi vertaillut hahmon merkit on tummennettu, ei-täsmäävästi vertaillut merkit on yliviivattu, ja siirtymäperusteena toiminut hahmon merkki on lihavoitu sekä alleviivattu. Lisäksi jokaisen siirtymän yhteydessä on mainittu siinä sovellettu hyppyfunktion arvo eli hahmon siirtymän pituus.Yhteensä tässä esimerkissä tehdään 21 kappaletta vertailuja.

1.6 Karp-Rabin

Karp ja Rabin [Karp and Rabin, 1987] ovat esittäneet edellisiin algoritmeihin verrattuna aivan erityyppisen lähestymistavan ”brute force”-algoritmin tehostamiseksi. Heidän algoritminsa perustuu epäkelpojen esiintymäkohtien nopeaan havaitsemiseen ns. ”sormenjälkien” avulla. Menetelmän ydin on käytössä olevan aakkoston sanojen kuvaaminen *a*-kantaisen lukujärjestelmän luvuiksi, missä *a* on kyseisen aakkoston merkkien lukumäärä. Olkoon N tämän kuvauksen antava funktio. Esimerkiksi normaali suomalainen aakkosto kuvautuisi 29-kantaisen lukujärjestelmän luvuiksi, ja sen eri aakkosia vastaavat kymmenjärjestelmän lukuarvot voisivat olla vaikkapa järjestyksessä N(’a’) = 0, N(’b’) = 1, …, N(’ö’) = 28. Yhtä merkkiä pidemmille merkkijonoille kokonaislukukuvaus muodostetaan normaaliin tapaan, eli yksittäisten merkkien lukuarvot painotetaan kantaluvun potensseilla oikealta vasemmalle nollannesta potenssista ylöspäin. Yleisesti kirjoitettuna kuvaus N on siten muotoa $N(mj[0\ldots m\text{-}1]) = N(mj[0]) \times a^{m-1} + N(mj[1]) \times a^{m-2} + \ldots + N(mj[m\text{-}2]) \times a^1 + N(t_{m-1}) \times a^0$, missä *mj* on *m* merkkiä pitkä merkkijono. Esimerkiksi merkkijonon ”abba” lukuarvo $N(\text{”abba”}) = N(\text{’a’}) \times a^3 + N(\text{’b’}) \times a^2 + N(\text{’b’}) \times a^1 + N(\text{’a’}) \times a^0 = 0 \times 29^3 + 1 \times 29^2 + 1 \times 29 + 0 \times 1 = 870$. Laajempaa merkistöä käytettäessä (esimerkiksi välimerkit mukana) yksi luonnollinen tapa voisi olla esimerkiksi merkkien ASCII-koodien käyttäminen niiden lukuesityksinä, jolloin N(λ) = merkin λ ASCII-koodi.

Olkoon *p* jokin alkuluku. Nyt merkkijonon *hahmo*[0…*m*-1] sormenjälki määritellään yksikäsitteiseksi luvuksi $S(hahmo[0\ldots m\text{-}1]) = (N(hahmo[0\ldots m\text{-}1])) \bmod p = (N(hahmo[0]) \times a^{m-1} + N(hahmo[1]) \times a^{m-2} + \ldots + N(hahmo[m\text{-}1])) \bmod p$. Jos siis edellisen esimerkin tilanteessa *p*:*n* arvoksi valittaisiin 23, olisi sanan ”abba” sormenjälki S(abba) = N(abba) mod *p* = 26130 mod 23 = 2, sillä 26130 = 23 × 1136 + 2. Selvästi keskenään identtisillä merkkijonoilla on myös arvoiltaan identtiset sormenjäljet, ja juuri tätä ominaisuutta Karp-Rabin-algoritmi hyödyntää. Nimittäin jos jollain tekstin indeksillä *j* pätee $S(teksti[j\ldots j+m\text{-}1] \neq S(hahmo[0\ldots m\text{-}1])$, pätee varmasti myös $teksti[j\ldots j+m\text{-}1] \neq hahmo[0\ldots m\text{-}1]$, eli kyseisen indeksin kohdalla ei voi olla hahmon esiintymää. Näin ollen etsittäessä hahmon esiintymiä tekstistä voidaan aina aluksi verrata hahmon sormenjälkeä kyseisessä kohdassa olevan hahmon pituisen tekstinosan sormenjälkeen. Jos sormenjäljet täsmäävät, tutkitaan suoraan merkki merkiltä vertailemalla, onko kyseessä todellakin hahmon esiintymä, ja muuten voidaan siirtyä tekstissä suoraan yhdellä askeleella eteenpäin. Luonnollisesti monella eri merkkijonolla voi olla sama sormenjälki, jolloin tehdään turhia vertailuita, mutta tämän tilanteen todennäköisyys on sitä pienempi, mitä isompi luku *p* valitaan. Yksi käytännöllinen

valinta voisi siten olla esimerkiksi suurin 16-bittiseen kokonaislukuun mahtuva alkuluku eli $p = 32749$.

Ollakseen parannus tavalliseen ”brute force”-merkkijonotäsmäysalgoritmiin pitää Karp-Rabin-algoritmin pystyä laskemaan käyttämänsä sormenjäljet nopeasti. Tämä onnistuu tehokkaasti käyttämällä hyödyksi viimeksi aiemmin laskettua sormenjälkeä, sillä $N(teksti[i+1\ldots i+m]) = N(teksti[i+1]) \times a^{m-1} + N(teksti[i+2]) \times a^{m-2} + \ldots + N(teksti[i+m]) = a \times (N(teksti[i]) \times a^{m-1} + N(teksti[i]) \times a^{m-2} + \ldots + N(teksti[i+m\text{-}1])) + N(teksti[i+m]) - N(teksti[i]) \times a^m = a \times N(teksti[i\ldots i+m\text{-}1]) + N(teksti[i+m]) - N(teksti[i]) \times a^m$. Käyttämällä hyväksi jakojäännösoperaation assosiatiivisuutta $(q + r) \bmod p = (q \bmod p + r \bmod p) \bmod p$ voidaan täten seuraavan tekstin vertailukohdan sormenjälki laskea edellisen sormenjäljen pohjalta nojautumalla yhtälöön $S(teksti[i+1\ldots i+m]) = N(teksti[i+1\ldots i+m]) \bmod p = ((a \times N(teksti[i\ldots i+m\text{-}1])) \bmod p + N(teksti[i+m]) \bmod p - (N(teksti[i] \times a^m) \bmod p) \bmod p$. Jotta laskemisen aikana käytettävät luvut pysyisivät mahdollisimman pieninä, kannattaa edellisessä hyödyntää jakojäännöksen ominaisuutta $(q \times r) \bmod p = ((q \bmod p) \times (r \bmod p)) \bmod p$. Tämän ansiosta laskemisessa käytetty luku a^m voidaan korvata luvulla $a^m \bmod p$, joka voidaan laskea etukäteen, sillä se on ainoastaan hahmon pituudesta ja kantaluvusta riippuvainen ja pysyy siten vakiona tietyn hahmon etsinnän aikana.

Hahmoa siirretään aina askel kerrallaan eteenpäin etsien sellaista tekstinkohtaa, jonka sormenjälki täsmää hahmon sormenjäljen kanssa. Tällaisen kohdan löydyttyä tutkitaan merkki kerrallaan onko kyseessä todellinen esiintymä vai ei. Edellisiä askeleita toistetaan, kunnes koko teksti on tutkittu. Karp-Rabin-algoritmin toteutus on esitetty algoritmissa 1.6.1.

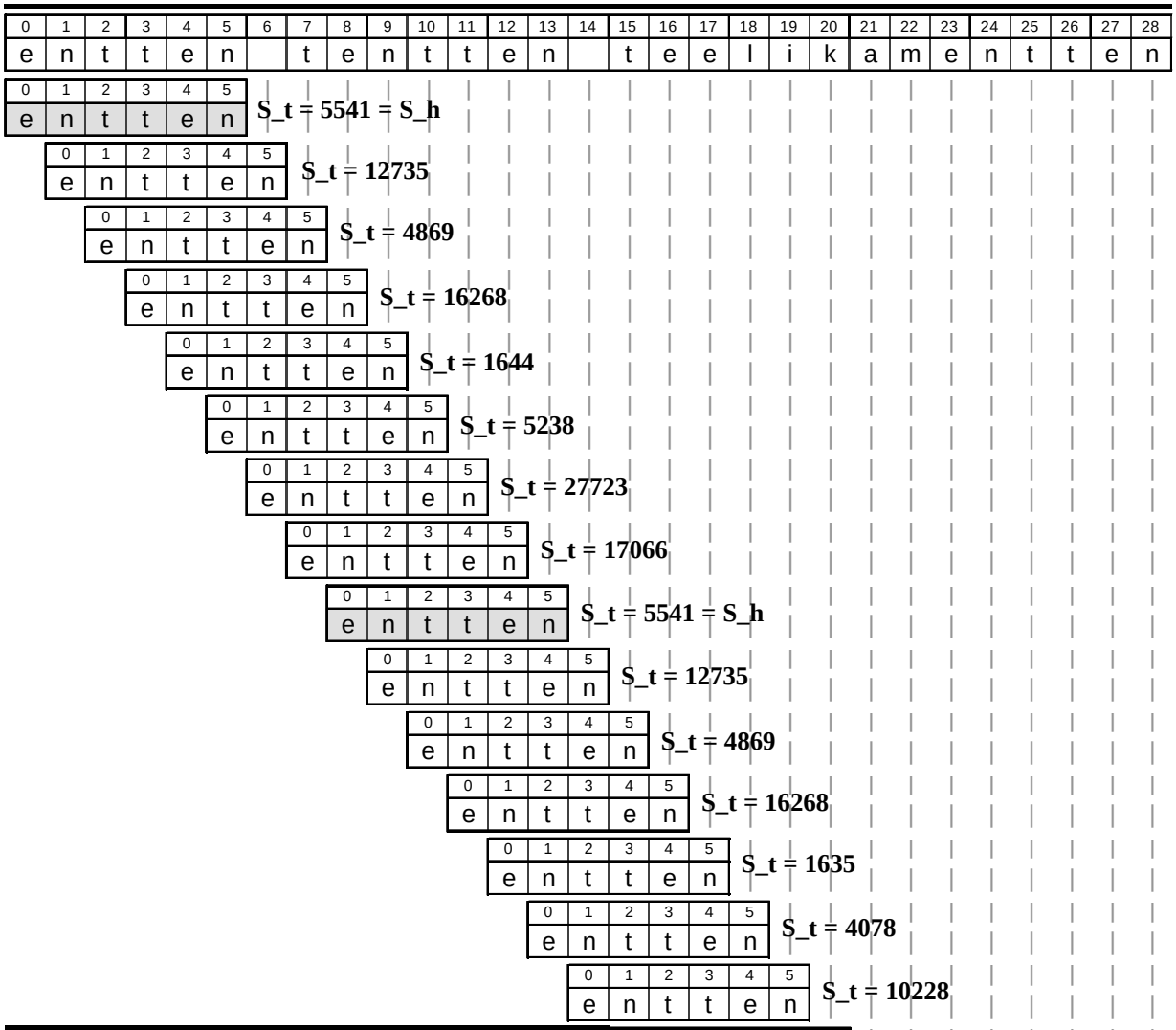
```
int S_h = 0;                                    // Hahmon sormenjälki.
int S_t = 0;                                    // Tekstin osasta laskettu sormenjälki.
int am = 1;                    // Sormenjälkien laskemisessa käytetty luku a^m alustetaan arvolla a^0.
int N(λ);                      // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
int i = 0;                                      // Hahmoa läpikäyvä indeksimuuttuja.
int j = 0;                     // Hahmon ensimmäistä merkkiä vastaavan merkin indeksi tekstissä.

for(i = 0; i < m; i++)                          // Lasketaan hahmon sekä ensimmäisen tekstinkohdan
{                                               // sormenjäljet, sekä lasketaan luvun a^m arvo.
    S_h = (S_h * a + N(hahmo[i])) % p;          // Lasketaan hahmon sormenjälki.
    S_t = (S_t * a + N(teksti[i])) % p;         // Lasketaan tekstin alkuosan sormenjälki.
    am = (am * a) % p;   // Lasketaan apuarvo a^m mod p.
}
if(S_h == S_t)                  // Tarkistetaan oliko heti tekstin alussa hahmon esiintymä.
{
    i = 0;
    while(i < m && teksti[i] == hahmo[i])       // Verrataan merkki merkiltä.
    {
        i++;
    }
    if(i == m)                                  // Täsmättiinkö m merkkiä, eli koko hahmo?
    {
        loytyi();               // Hahmon esiintymä löytyi, suoritetaan tarvittavat toimenpiteet.
    }
}
j = 1;                                          // Seuraavaksi edetään tekstin indeksistä j = 1 eteenpäin.
while(j < n – m + 1)                            // Käydään koko lopputeksti läpi.
{
    S_t = (a * S_t + N(teksti[j + m]) % p – ((am * N(teksti[j])) % p)) % p;      //  Päivitetään tämänhetkisen
                                    // tekstinosan sormenjäljen arvo edellistä  sormenjäljen arvoa hyödyntäen.
    if(S_h == S_t)                              // Tarkistetaan onko kyseessä hahmon esiintymä.
    {
        i = 0;
        while(i < m && teksti[j + i] == hahmo[i])   // Verrataan merkki merkiltä.
        {
            i++;
        }
        if(i == m)                              // Täsmättiinkö m merkkiä, eli koko hahmo?
        {
            loytyi();                           // Hahmon esiintymä löytyi, suoritetaan
        }                                       // tarvittavat toimenpiteet.
    }
    j++;                                        // Edetään tekstissä.
}
```

Algoritmi 1.6.1:
Karp-Rabin-algoritmi.

Karp-Rabin-algoritmin aikavaativuudelle saadaan melko suoraviivaisesti pahimman tapauksen analyysillä O($n \times m$). Tähän päädytään olettamalla, että jokaisella askeleella tekstin osan ja hahmon sormenjäljet täsmäävät ja että hahmon esiintymän tarkistamiseksi joudutaan tekemään keskimäärin hahmon pituuteen verrannollinen määrä vertailuja. Koska kierroksia on tekstin pituuteen n verrannollinen määrä ja hahmo on m merkkiä pitkä, on lopputuloksena $n \times m$ vertailua. Yksi patologinen esimerkki tällaisesta tilanteesta saadaan asettamalla *hahmo*[0… m-1] = ”aaa…a” ja *teksti*[0…n-1] = ”aaa…a”, missä jälleen oletetaan tekstin pituus n hahmon pituutta m huomattavasti suuremmaksi. Nyt tekstin jokaisessa kohdassa löydetään hahmon esiintymä, ja Karp-Rabin-algoritmi joutuu tutkimaan ne kaikki tehden näin ollen $(n - m + 1) \times m = n \times m - m^2 + m$ vertailua, mikä on aikavaativuudeltaan luokkaa O($n \times m$).

Kuten aiemmin mainittiin, kannattaa sormenjälkien laskennassa käytetty alkuluku p valita mahdollisimman suureksi, sillä saman sormenjäljen omaavien mutta kuitenkin ei-täsmäävien merkkijonojen lukumäärä on kääntäen verrannollinen lukuun p. Näin todennäköisyys, että tekstistä löytyy hahmon kanssa saman sormenjäljen omaava mutta ei –täsmäävä merkkijono, saadaan niin pieneksi, että käytännössä algoritmin voi todeta toimivan keskimäärin lineaarisessa ajassa tekstin ja hahmon pituuksiin nähden [Gusfield, 1997].

Luvuksi p valitaan nimenomaan alkuluku siksi, että Karp-Rabin-algoritmin tehokkuutta eli todennäköisyyttä, jolla tekstissä kohdataan hahmon kanssa erilainen mutta saman sormenjäljen omaava merkkijono, voidaan arvioida alkulukujen ominaisuuksiin pohjautuen melko hyvin [Karp and Rabin, 1987]. Lisäksi Karp ja Rabin esittävät myös adaptiivisen sormenjälkien sovellustavan, jossa sormenjälkien laskemisessa käytetty luku p vaihdetaan uuteen (ja ennen käyttämättömään) aina, kun tekstistä löydetään hahmon kanssa erilainen, mutta kuitenkin saman sormenjäljen omaava merkkijono. Kun luvun p arvot ovat aina aiemmin käyttämättömiä alkulukuja, pienenee näin toimittaessa usean turhan tekstinkohdan tarkistamisen todennäköisyys eksponentiaalisesti. Kuva 1.6.2 esittää esimerkin Karp-Rabin-algoritmin toiminnasta käytännössä.



Kuva 1.6.2:
Merkkijonon *hahmo*[0...5] = ”tentten” etsiminen merkkijonosta *teksti*[0...28] = ”entten tentten teelikamentten” Karp-Rabin-algoritmilla. Tässä p = 32749, a = 128 ja N(λ) = merkin λ ASCII-koodi. Hahmon sormenjälki S_h = 5541. Hahmoa siirretään askel kerrallaan jokaiseen mahdolliseen tekstinkohtaan vasemmalta oikealle verraten aina jokaisessa kohdassa kyseisen tekstinkohdan ja hahmon sormenjälkiä. Mikäli sormenjäljet täsmäävät, tarkistetaan merkki merkiltä vertailemalla, löytyykö siitä kohdasta hahmon esiintymä, ja muuten jatketaan eteenpäin. Kuvassa täsmäävästi vertaillut hahmon merkit on tummennettu. Tässä esimerkissä ei tapahdu yhtään ei-täsmäävää vertailua. Yhteensä suoritetaan (m-n+1) = 24 kappaletta sormenjälkivertailuja, ja merkkien välisiä vertailuja tehdään 18 kappaletta.

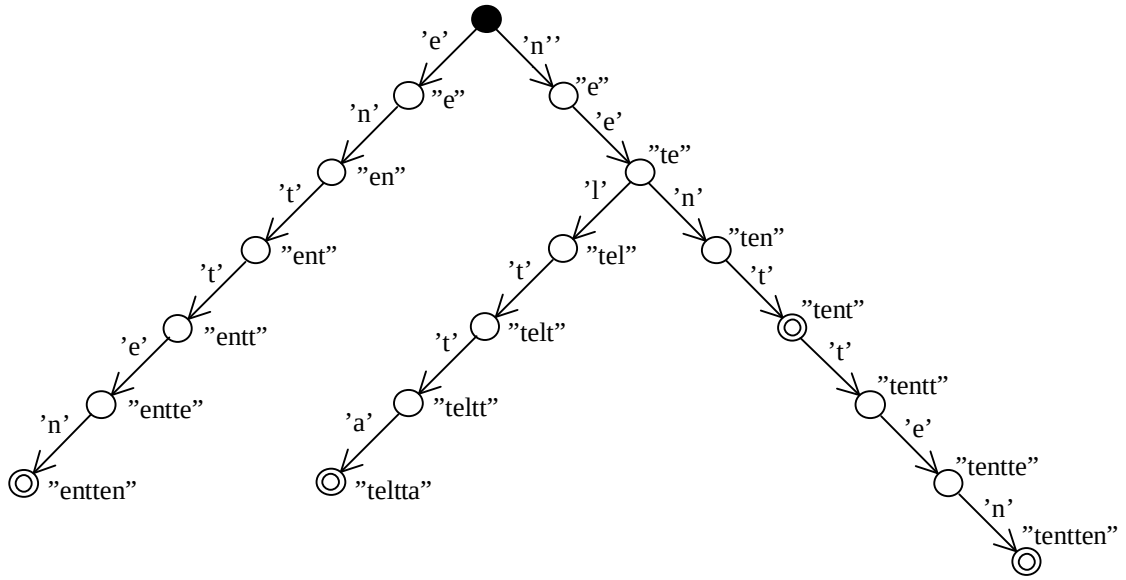
1.7 Monen hahmon samanaikainen eksakti etsintä

Jos etsittäviä hahmoja on enemmän kuin yksi, voidaan niiden etsintä luonnollisesti toteuttaa soveltamalla jotain tähän mennessä mainituista algoritmeista jokaiselle hahmolle erikseen. Tutkittaessa n merkkiä pitkää tekstiä nämä algoritmit ovat aikavaativuudeltaan parhaimmillaan $O(n)$, joten koko etsintäprosessin aikavaativuus on tällöin $O(n \times h)$, missä h on etsittävien hahmojen lukumäärä. Koska käytännössä teksti voi olla usein hyvinkin pitkä, ei tämä tunnu välttämättä kaikkein tehokkaimmalta lähestymistavalta. Toinen lähestymistapa on toteuttaa hahmojen etsintä rinnakkain siten, että aina kussakin tekstinkohdassa oltaessa verrataan tekstiä kaikkiin hahmoihin samanaikaisesti. Tämä voidaan toteuttaa avainpuun ("keyword tree" tai "trie") avulla, joka on hahmojen indeksinä toimiva suunnattu puu. Käytetään tässä käsittelyssä avainpuusta seuraavaa määritelmää:

Määritelmä 1.7.1:
Merkkijonoja $hahmo_1$, $hahmo_2$, …, $hahmo_h$ vastaava avainpuu koostuu juurisolmusta alkavista yksikäsitteisistä suunnatuista poluista seuraavien ehtojen mukaisesti:

a) Jokaista hahmoa vastaa täsmälleen yksi juuresta alkava suunnattu polku.
b) Jokainen puun solmu kuuluu jotain hahmoa vastaavalle polulle.
c) Kutakin hahmoa vastaavan polun viimeinen solmu on merkitty kyseisen hahmon loppusolmuksi.
d) Jokainen avainpuun kaari vastaa täsmälleen yhtä merkkiä.
e) Jokainen avainpuun solmu vastaa järjestyksessä alusta loppuun juuresta kyseiseen solmuun kulkevan polun sisältämien kaarien vastinmerkkien muodostamaa merkkijonoa.

Otetaan käyttöön merkintä M_v tarkoittamaan avainpuussa olevan solmun v vastaamaa merkkijonoa, jolloin avainpuussa on siis täsmälleen h loppusolmua v_1, v_2, …, v_h, ja niille pätee $M_{v1} = hahmo_1$, $M_{v2} = hahmo_2$, …, ja $M_{vh} = hahmo_h$. Tämä merkkijonon ja solmun vastaavuus on luonnollisesti kaksisuuntainen, eli $M_v = mj$ jos ja vain jos avainpuun solmu v vastaa merkkijonoa mj. Käytetään avainpuun juurisolmusta merkintää v_r, jolloin $M_{vr} = \varepsilon$, sillä juurisolmu vastaa tyhjää merkkijonoa.



Kuva 1.7.2:
Merkkijonojen ”entten”, ”tentten”, ”teltta” ja ”tent” pohjalta muodostettu avainpuu. Kuvassa juurisolmu on väritetty mustalla ja loppusolmuilla on kaksoisreunus. Lisäksi jokaisen solmun yhteydessä on ilmoitettu sitä vastaava merkkijono.

Avainpuun muodostaminen on korkealla tasolla ajateltuna hyvin yksinkertainen toimenpide, sillä selvästi riittää aloittaa juurisolmun omaavasta puusta ja lisätä siihen hahmot yksitellen siten, että aloitetaan aina juurisolmusta ja edetään merkki kerrallaan seuraten mahdollisimman pitkälle jo valmiiksi puussa olevia kyseisen merkkijonon merkkejä vastaavia kaaria, ja jos tämä ei enää onnistu, luodaan vielä käsittelemättömiä merkkejä vastaavat kaaret päätesolmuineen. Samalla aina kunkin hahmon viimeistä merkkiä vastaavan kaaren päätesolmu merkitään kyseisen hahmon loppusolmuksi. Käytännössä kuitenkin avainpuun solmujen linkitys voidaan hoitaa monella eri tavalla. Tässä esittämäni käytäntö on niistä yksinkertaisin, merkkitaulukkoihin perustuva tapa, jossa jokainen solmu sisältää koko aakkoston käsittävän osoitintaulukon. Solmun *u* osoitintaulukossa merkkiä λ vastaava osoitin osoittaa solmuun *v*, jos solmusta *u* menee merkkiä λ vastaava suunnattu kaari solmuun *v*, ja muussa tapauksessa kyseisen osoittimen arvo on määrittelemätön. Käytännössä osoitintaulukon käyttö voi viedä hyvinkin paljon muistitilaa jos aakkosto on laaja, mutta vastapainona solmusta toiseen eteneminen tiettyä merkkiä vastaavaa kaarta pitkin on nopeaa. Algoritmissa 1.7.3 jokaiseen avainpuun solmuun lisätään myös osoitin sen isäsolmuun sekä tieto siitä, mitä merkkiä vastaavan kaaren päätesolmusta on kyse, sillä näitä tietoja tarvitaan myöhemmin esitettävän Aho-Corasick-algoritmin yhteydessä.

```
char* hahmot[h];                    // Merkkijonot hahmo1, hahmo2, …, hahmoh sisältävä taulukko.
int m[h];                           // Merkkijonojen hahmo1, hahmo2, …, hahmoh pituudet.
int i = m – 1;                      // Hahmojen merkkejä läpikäyvä indeksi.
int j = 0;                          // Hahmoja läpikäyvä indeksi.
int apu = 0;                        // Apuindeksimuuttuja.
int a;                              // Käytettävän aakkoston merkkien lukumäärä.
int N(λ);                 // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
int f_q[a];                         // Hyppyfunktion arvotaulukko, f_q[i] = fq(i).
solmu *juuri;                       // Osoitin avainpuun juurisolmuun.
solmu *apusolmu;                    // Avainpuun käsittelyssä käytetty apumuuttuja.

for(apu = 0; apu < a; apu++)        // Alustetaan juurisolmusta lähtevät kaaret tyhjiksi.
{
    juuri->kaaret[apu] = NULL;
}
juuri->isasolmu = NULL;             // Juurisolmulla ei ole isäsolmua.
juuri->merkki = NULL;               // Juuri ei ole mitään merkkiä vastaavan kaaren päätesolmu.
juuri->loppu = -1;                  // Juurisolmu ei ole minkään hahmon loppusolmu.
for(j = 0; j < h; j++)              // Käydään kaikki hahmot läpi.
{
    apusolmu = juuri;               // Aloitetaan uuden hahmon lisääminen juurisolmusta.
    i = 0;                  // Aloitetaan vuorossa olevan hahmon läpikäynti sen ensimmäisestä merkistä.

    // Edetään avainpuussa niin pitkälle, kuin jo olemassa olevia kaaria pitkin pääsee
    // merkkijonon hahmo[j] merkkejä vastaavia kaaria pitkin.
    while(i < m[j] && apusolmu->kaaret[N(hahmot[j][i])] != NULL)
    {
        apusolmu = apusolmu->kaaret[N(hahmot[j][i])];   // Edetään merkkiä hahmot[j][i] vastaava
        i++;                                            // kaari sekä vastaava merkki hahmossa.
    }

    // Lisätään puuhun siitä vielä puuttumattomat hahmon loppuosaa vastaavat kaaret.
    while(i < m[j])
    {
        apusolmu->kaaret[N(hahmot[j][i])] = new solmu;  // Luodaan uusi kaari/solmu.

        // Lisätään uuteen solmuun osoitin sen isäsolmuun.
        apusolmu->kaaret[N(hahmot[j][i])]->isasolmu = apusolmu;
        apusolmu = apusolmu->kaaret[N(hahmot[j][i])];   // Siirrytään uuteen solmuun.
        apusolmu->merkki = hahmot[i][j];                // Merkitään uuteen solmuun tieto siitä,
                                                        // että mitä merkkiä vastaavaa kaarta
                                                        // pitkin siihen saavutaan.
        for(apu = 0; apu < a; apu++)            // Alustetaan uuden solmun lähtevät kaaret tyhjiksi.
        {
            apusolmu->kaaret[apu] = NULL;
        }
        apusolmu->loppu = -1;                   // Alustetaan juuri lisätty solmu ei-loppusolmuksi.
        i++;
    }
    apusolmu->loppu = j;                // Merkitään loppusolmu. Tässä toteutuksessa ei kirjata
}                                       // erikseen jokaista samaan solmuun päättyvää (ja siten
                                        // keskenään identtistä) hahmoa.
```

Algoritmi 1.7.3:
Avainpuun muodostaminen merkkijonoille $hahmo_1$, $hahmo_2$, …, $hahmo_h$.

Algoritmin 1.7.3 aikavaativuudeksi saadaan pahimman tapauksen analyysillä $O(|\Lambda| \times \Sigma m)$, missä $|\Lambda|$ on käytettävän aakkoston koko ja Σm merkkijonojen $hahmo_1$, $hahmo_2$, …, $hahmo_h$ yhteispituus. Tässä on otettava huomioon se, että solmujen välisten kaarien yhteydessä voitaisiin käyttää myös esimerkiksi lista- tai puurakenteita koko aakkoston kokoisten taulukoiden sijaan (esim. [Gusfield, 1997] tai [Bentley and Sedgewick, 1997]).

Määritelmän 1.7.1 nojalla mielivaltaiselle ei-tyhjälle merkkijonolle *mj* pätee ehto *mj* = $hahmo_g$ jollain indeksillä *g*, missä $1 \leq g \leq h$, jos ja vain jos merkkijonoista $hahmo_1$, $hahmo_2$, …, $hahmo_h$ muodostetussa avainpuussa on merkkijonoa *mj* vastaava juuresta alkava ja loppusolmuun päättyvä suunnattu polku. Siten ”brute force”-lähestymistapaa käyttäen merkkijonojen $hahmo_1$, $hahmo_2$, …, $hahmo_h$ esiintymät voidaan etsiä tekstistä siten, että jokaisen tekstin indeksin *j* kohdalla kokeillaan erikseen, päästäänkö merkkejä *teksti*[*j*], *teksti*[*j*+1], … vastaavia kaaria seuraten näiden hahmojen avainpuun juurisolmusta yhteen tai useampaan hahmon loppusolmuun. Tämän menettelyn toteutus algoritmissa 1.7.4 seuraa tarkkaan yhden hahmon etsinnässä käytettyä ”brute force”-algoritmia 1.1.2, erona on lähinnä avainpuun käyttö hahmon sijaan merkkivertailuissa.

```
int j = 0;       // Tekstinkohtia läpikäyvä indeksi.
int i = 0;       // Avainpuussa täsmättyjen tekstin merkkien laskuri-indeksi.
int a;           // Käytettävän aakkoston merkkien lukumäärä.
int N( l);       // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
char* hahmot[h];   // Etsittävien merkkijonojen taulukko, hahmot[k] = hahmo_{k+1}.
solmu *juuri;          // Osoitin avainpuun juurisolmuun.
solmu *apusolmu;   // Avainpuun käsittelyssä käytetty apumuuttuja.

for(j = 0; j < n; j++)//   Silmukka käy läpi koko tekstin alusta loppuun.
{
    apusolmu = juuri;       // Aloitetaan kaarien seuraaminen juuresta.
    i = 0;       // Vertailut aloitetaan merkistä teksti[j+i] = teksti[j].
    while(apusolmu->kaaret[N(teksti[j+i])] != NULL)// Verrataan avainpuun
    {                                                    // sisältämiä hahmoja tekstiin.
        apusolmu = apusolmu-> kaaret[N(teksti[j+i])];     // Edetään avainpuussa.
        i++;        // Edetään vastaava askel myös tekstissä (kyseisessä tekstinkohdassa).
        if(apusolmu->loppu >= 0)  // Saavuttiinko loppusolmuun?
        {
            loytyi(apusolmu->loppu);  // Rekisteröidään merkkijono
        }                                       // hahmot[apusolmu->loppu] löytyneeksi.
    }
}
```

Algoritmi 1.7.4:
”Brute force”-tyyppinen avainpuuta käyttävä usean merkkijonon etsintäalgoritmi.

Algoritmin 1.7.4 aikavaativuudeksi saadaan pahimman tapauksen analyysin nojalla $O(|\Lambda|\times\Sigma m + n\times m)$, missä m on pisimmän etsittävän hahmon pituus. Ensimmäinen termi tulee avainpuun muodostamisesta, ja jälkimmäinen seuraa siitä, että m on pisimmän avainpuussa olevan suunnatun polun pituus ja avainpuuta käytetään vertailuissa jokaisen tekstin merkin kohdalla eli n kertaa.

Avainpuun käytön yhteydessä voidaan soveltaa monia yksittäisenkin hahmon etsinnässä käytettyjä tehostamiskeinoja. Ahon ja Corasickin [Aho and Corasick, 1975] esittämä menetelmä lienee kaikkein klassisin tällainen rinnakkaishaun toteuttava algoritmi, jonka aikavaativuus on alle $O(n\times h)$. Se saadaan Morris-Pratt-algoritmista muokkaamalla sen korjausfunktion toimintaa sopivasti, kun halutaan etsiä useampaa eri hahmoa samanaikaisesti. Ajatuksena on toteuttaa ei-täsmäävän vertailun sattuessa (eli kun avainpuussa ei enää päästä pidemmälle) aiemmin täsmättyjen merkkien (eli avainpuussa kuljetun polun) perusteella suurempi kuin yhden askeleen eteneminen tekstissä. Tämä tehdään korjausfunktiota f_k matkimalla. Kun avainpuussa ei enää päästä eteenpäin, siirrytään tekstissä seuraavaan sellaiseen kohtaan, josta eteenpäin juuri täsmättyä tekstinosaa vastaa jokin avainpuun solmu, tai jos tällaista solmua ei löydy, siirrytään kokonaan juuri verratun tekstinkohdan ohi. Samalla siirrytään lisäksi avainpuussa edellä siirtymäperusteena käytettyä aiemmin täsmätyn tekstinkohdan loppuosaa vastaavaan solmuun (tai juureen jos tällaista solmua ei ollut), jolloin kyseisessä uudessa tekstinkohdassa jo täsmääväksi tiedettyä avainpuussa olevan polun alkuosaa ei turhaan kuljeta enää uudelleen. Jos avainpuussa ollaan juuren kohdalla eikä juuresta löydy sillä hetkellä tekstissä vertailuvuorossa olevaa merkkiä *teksti*[j] vastaavaa kaarta, siirrytään tekstissä eteenpäin merkkiin *teksti*[j+1]. Tämä poikkeaa Morris-Pratt-algoritmin kuvassa 1.2.1 esitetystä toimintaperiaatteesta ainoastaan siten, että nyt otetaan huomioon kaikkien hahmojen $hahmo_1$, $hahmo_2$, …, $hahmo_h$ alkuosat määriteltäessä siirtymän pituutta.

Määritellään edellä kuvaillulla tavalla toimiva Aho-Corasick-korjausfunktio f_{ac} siten, että arvo $f_{ac}(v)$ osoittaa aina, mihin solmuun avainpuussa voidaan siirtyä, jos ollaan solmun v kohdalla eikä siitä löydy vertailuvuorossa olevaa tekstin merkkiä *teksti*[j] vastaavaa etenevää kaarta. Jos v on avainpuun juurisolmu v_r, ei edellä mainitussa tilanteessa aiemman mukaan siirrytä avainpuussa lainkaan, vaan edetään sen sijaan askel tekstissä. Määritellään tätä tilannetta varten erikoisarvo $f_{ac}(v_r)$ = NULL, jolloin funktiota f_{ac} sovellettaessa tiedetään, että nolla-osoittimen tapauksessa pysytään paikallaan ja kasvatetaan tekstin indeksiä.

Määritelmä 1.7.5:
Aho-Corasick-algoritmin korjausfunktio f_{ac}:
$f_{ac}(v_r) = \text{NULL}$, kun $v = v_r$, ja muulloin
$f_{ac}(v) = u \mid (u = v_r \vee M_u = M_v[|M_v|-|M_u|\ldots|M_v|-1]) \wedge |M_u| = \max\{k \mid k = 0 \vee (0 < k < |M_v| \wedge \exists y$: $hahmo_y[0\ldots k\text{-}1] = M_v[|M_v|-k\ldots|M_v|-1])\}$.

Lauseissa 1.7.6 ja 1.7.7 todetaan korjausfunktion f_{ac} toiminnallisuudesta Morris-Pratt-algoritmin yhteydessä esitettyjä lauseita 1.2.8 ja 1.2.12 vastaavat ominaisuudet.

Lause 1.7.6:
Kun tekstiä tutkittaessa ollaan avainpuussa solmun v ja tekstissä merkin $teksti[j]$ kohdalla ja lisäksi etenemisen ollessa mahdotonta avainpuussa on aina siirrytty korjausfunktion f_{ac} mukaiseen solmuun, niin joko $v = v_r$ tai $M_v = teksti[j\text{-}|M_v|\ldots j\text{-}1]$.

Todistus:
Alussa ollaan tekstissä merkin $teksti[0]$ kohdalla ja avainpuussa solmussa v_r, ja siten lauseen väite on tällöin voimassa. Oletetaan nyt, että lause on tosi solmun v ja merkin $teksti[j_v]$ kohdalla, ja tarkastellaan tilannetta, jossa siitä siirrytään solmuun u. Olkoon $teksti[j_u]$ tässä tilanteessa tekstin osalta tarkasteltava merkki.

Jos solmusta v löytyy merkkiä $teksti[j_v]$ vastaava suunnattu kaari, pätee $j_u = j_v + 1$, $|M_u| = |M_v| + 1$ ja oletuksen perusteella $M_u = M_v \cup teksti[j_v] = teksti[j_u\text{-}|M_u|\ldots j_u\text{-}1]$, joten lauseen väite on tässä tilanteessa tosi merkin $hahmo[j_u]$ ja solmun u kohdalla.

Jos solmusta v ei löydy merkkiä $teksti[j_v]$ vastaavaa suunnattua kaarta ja $v = v_r$, pätee $u = v_r$, joten lauseen väite on tosi merkin $hahmo[j_u]$ ja solmun u kohdalla myös tässä tilanteessa.

Jos solmusta v ei löydy merkkiä $teksti[j_v]$ vastaavaa suunnattua kaarta ja $v \neq v_r$, pätee joko $u = v_r$, jolloin lauseen väite on suoraan tosi merkin $hahmo[j_u]$ ja solmun u kohdalla, tai sitten $j_u = j_v$, oletuksen perusteella $M_v = teksti[j\text{-}|M_v|\ldots j\text{-}1]$ ja määritelmän 1.7.5 nojalla $|M_u| = \max\{k \mid k = 0 \vee (0 < k < |m_v| \wedge \exists y$: $hahmo_y[0\ldots k\text{-}1] = M_v[|M_v|-k\ldots|m_v|-1])\}$ sekä $M_u = M_v[|M_v|-|M_u|\ldots|m_v|-1]$. Koska $|M_u| < |M_v|$, saadaan edelliset täsmäävyydet yhdistämällä $M_u = teksti[j_u\text{-}|M_u|\ldots j_u\text{-}1]$, joten tässäkin tilanteessa lauseen väite on tosi merkin $hahmo[j_u]$ ja solmun u kohdalla.

Edelliset kohdat yhdistäen voidaan lauseen väite todeta induktioperiaatteen nojalla oikeaksi.

Lause 1.7.7:
Kun tekstiä tutkittaessa ollaan avainpuussa solmun v ja tekstissä merkin *teksti*[j] kohdalla ja lisäksi etenemisen ollessa mahdotonta avainpuussa on aina siirrytty korjausfunktion f_{ac} mukaiseen solmuun, niin toinen seuraavista ehdoista a) ja b) on voimassa:

a) Solmusta v lähtee merkkiä *teksti*[j] vastaava suunnattu kaari sellaiseen solmuun u, että $|M_u| = |M_v| + 1 = \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$ ja $M_u = teksti[j\text{-}|M_u|\text{+}1\ldots j]$.

b) Solmusta v ei lähde merkkiä *teksti*[j] vastaavaa suunnattua kaarta, ja pätee $|M_v| + 1 > \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$.

Todistus:
Alussa avainpuussa ollaan juurisolmussa v_r ja tekstissä merkin *teksti*[0] kohdalla, ja tällöin selvästi pätee joko ehto a) tai ehto b). Tarkastellaan nyt tilannetta, jossa lauseen väite on tosi solmun v ja merkin *teksti*[j_v] kohdalla ja seuraavaksi siirrytään solmun u ja merkin *teksti*[j_u] kohdalle. Jaetaan tarkastelu kahteen osaan sen mukaan, löytyykö solmusta v merkkiä *teksti*[j_v] vastaavaa lähtevää kaarta vai ei:

1) Solmusta v lähtee merkkiä *teksti*[j_v] vastaava suunnattu kaari:
Tällöin oletuksen nojalla pätee $|M_u| = |M_v| + 1 = \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\}$, $M_u = teksti[j_v\text{-}|M_u|\text{+}1\ldots j_v]$ ja $j_u = j_v + 1$.
Jos nyt solmusta u lähtee merkkiä *teksti*[j_v+1] = *teksti*[j_u] vastaava suunnattu kaari solmuun w, niin $|M_w| = |M_u| + 1$, $M_w = M_u \cup teksti[j_u] = teksti[j_v\text{-}|M_u|\text{+}1\ldots j_u] = teksti[j_u\text{-}|M_w|\text{+}1\ldots j_u]$. Lausetta 1.2.9 hieman soveltamalla nähdään, että $|M_w| = |M_u| + 1 = \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$, joten tässä tilanteessa ehto a) on voimassa solmun u ja merkin *teksti*[j_u] kohdalla.
Jos taas solmusta u ei lähde merkkiä *teksti*[j_v+1] = *teksti*[j_u] vastaavaa suunnattua kaarta, niin jälleen lausetta 1.2.9 hieman soveltamalla tiedetään, että $|M_u| + 1 \geq \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$. Toisaalta nyt $|M_u| + 1 \neq \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$ avainpuun solmujen ja merkkijonojen vastaavuuksien yksikäsitteisyyden nojalla, ja siten on pakko päteä $|M_u| + 1 > \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$, joten tässä tilanteessa on ehto b) voimassa solmun u ja merkin *teksti*[j_u] kohdalla.

2) Solmusta v ei lähde merkkiä *teksti*[j_v] vastaavaa suunnattua kaarta:

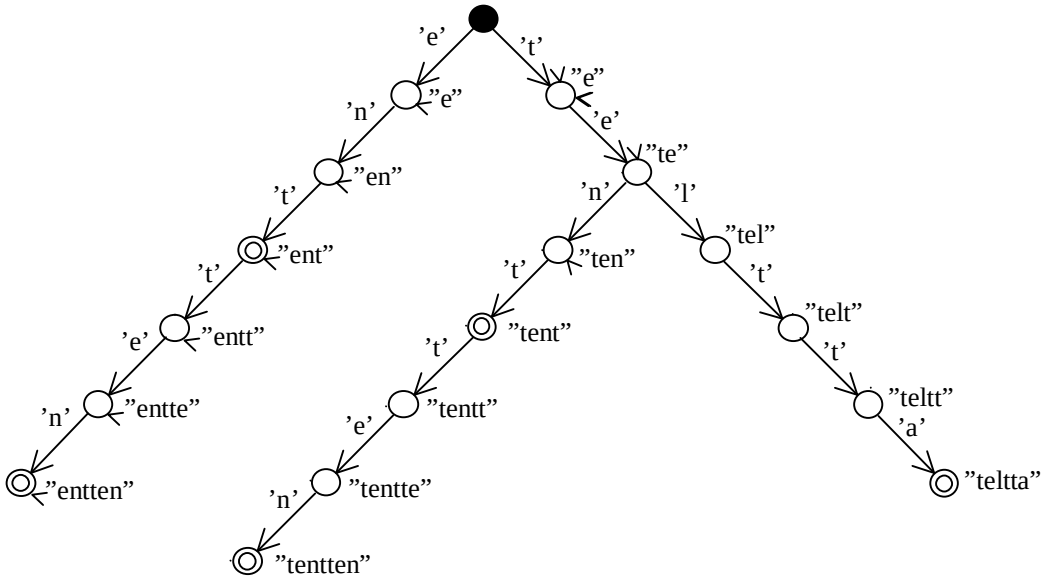
Oletuksen nojalla tässä tapauksessa pätee $|M_v| + 1 > \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\}$.

Jos $f_{ac}(v) = \text{NULL}$, pätee $v = v_r = u$, $|M_v| = |M_u| = 0$ sekä $j_u = j_v + 1$, ja siten nyt $|M_v| + 1 = 1 > \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j_v\text{-}k\text{+}1\ldots j_v]\} = 0$. Näin ollen tässä tilanteessa nähdään lausetta 1.2.9 soveltaen, että $|M_u| + 1 = 1 \geq \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$ ja yhtäsuuruus on voimassa jos ja vain jos löytyy sellainen $hahmo_y$, että $hahmo_y[0] = teksti[j_u]$, eli jos ja vain jos solmusta $v_r = u$ lähtee merkkiä $teksti[j_u]$ vastaava kaari. Tästä seuraa, että tässä tilanteessa solmun u ja merkin $teksti[j_u]$ kohdalla on voimassa joko ehto a) tai ehto b).

Jos $f_{ac}(v) \neq \text{NULL}$, pätee $u = f_{ac}(v)$, $j_u = j_v$ sekä $|M_v| + 1 > \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j\text{-}k\text{+}1\ldots j]\}$, ja nyt $|M_u| = \max\{k \mid k = 0 \vee (0 < k < |m_v| \wedge \exists y: hahmo_y[0\ldots k\text{-}1] = M_v[|M_v|\text{-}k\ldots|m_v|\text{-}1])\}$. Lisäksi tällöin $v \neq v_r$, joten lauseen 1.7.6 nojalla $M_v = teksti[j_v\text{-}|M_v|\ldots j_v\text{-}1]$. Tämän täsmäävyyden sekä tiedon $j_u = j_v$ perusteella on $\max\{k \mid k = 0 \vee (0 < k < |m_v| \wedge \exists y: hahmo_y[0\ldots k\text{-}1] = M_v[|M_v|\text{-}k\ldots|m_v|\text{-}1])\} = \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j_u\text{-}k\ldots j_u\text{-}1]\} = |M_u|$. Siten lausetta 1.2.9 soveltaen nähdään, että $|M_u| + 1 \geq \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j_u\text{-}k\text{+}1\ldots j_u]\}$ ja edellisen kohdan tapaan yhtäsuuruus on voimassa jos ja vain jos löytyy sellainen $hahmo_y$, että $hahmo_y[0\ldots|M_u|] = teksti[j_u\text{-}|M_u|\ldots j_u] = M_u \cup teksti[j_u]$, eli jos ja vain jos solmusta u lähtee merkkiä $teksti[j_u]$ vastaava kaari. Tästä seuraa jälleen, että tässä tilanteessa solmun u ja merkin $teksti[j_u]$ kohdalla on voimassa joko ehto a) tai ehto b).

Kohtien 1) ja 2) sekä induktioperiaatteen nojalla voidaan todeta, että lauseen väite on tosi.

Lauseen 1.7.7 perusteella Aho-Corasick-korjausfunktiota käytettäessä löydetään pituudeltaan maksimaaliset jonkin etsittävän hahmon kanssa osittain tai kokonaan täsmäävät tekstin osat. Morris-Pratt-algoritmissa korjausfunktion f_k vastaava ominaisuus riitti takaamaan, että mikään hahmon esiintymä ei jää löytymättä tekstistä. Useampaa hahmoa etsittäessä tämä ei kuitenkaan vielä riitä, sillä on myös mahdollista, että jokin hahmo sisältyy kokonaan toiseen hahmoon, jolloin siis etsittävien merkkijonojen joukosta löytyy esimerkiksi sellaiset $hahmo_x$ ja $hahmo_y$, että $hahmo_y = mj_1 \cup hahmo_x \cup mj_2$ sekä mj_1 ja/tai mj_2 on jokin ei-tyhjä merkkijono. Tämä mahdollistaa sellaisen tilanteen, että tekstistä on täsmätty osa $mj_1 \cup hahmo_x$, mutta silti ei päädytä missään vaiheessa merkkijonoa $hahmo_x$ vastaavaan loppusolmuun (kuva 1.7.8).



Kuva 1.7.8:
Kuvan 1.7.2 avainpuu, johon on lisätty merkkijono ”ent” sekä korjausfunktion f_{ac} arvot. Kuvassa on merkitty katkoviiva-nuolilla kaikki sellaiset korjausfunktion arvot, jotka ovat erisuuria kuin juurisolmu v_r tai NULL. Esityksen selkeyttämiseksi avainpuussa on vaihdettu merkkijonoja ”teltta” ja ”tentten” vastaavat haarat toisin päin verrattuna kuvaan 1.7.2. Jos esimerkiksi *teksti* = ”tentten teelikamentten”, siirrytään korjausfunktion f_{ac} mukaisesti merkkijonon ”tentten” täsmäyksen jälkeen avainpuussa seuraavaksi merkkijonoa ”ten” vastaavaan solmuun, ja tekstistä verrataan seuraavaksi merkkiä *teksti*[7] = ” ”. Näin merkkijonoon ”tentten” sisältyvä, merkkiin *teksti*[3] päättyvä merkkijonon ”ent” esiintymä jää noteeraamatta, sillä funktiota f_{ac} sovellettaessa tekstissä ei missään vaiheessa palata enää taaksepäin merkin *teksti*[3] kohdalle.

Edellä kuvatun mahdollisuuden takia korjausfunktiota f_{ac} sovellettaessa tulee jokaiseen avainpuun solmuun v lisätä tieto siitä, päteekö sen kohdalla ehto $M_v = mj \cup hahmo_x$, missä mj on jokin ei-tyhjä merkkijono ja $hahmo_x$ jokin etsittävistä hahmoista. Käytännössä tämä voidaan tehdä lisäämällä solmuun v hahmo-osoitin $h_{ac}(v)$, jonka arvo osoittaa pisintä edellisenkaltaista etsittävää hahmoa vastaavaan loppusolmuun u. Käytetään tässä sellaista sopimusta, että jos tällaista loppusolmua u ei ole lainkaan olemassa, niin solmun v hahmo-osoitin osoittaa avainpuun juurisolmuun v_r. Koska juurisolmu vastaa tyhjää merkkijonoa, ei sen hahmo-osoitinta ole määritelty, ja siten edellisen mukaan $h_{ac}(v_r) = v_r$. Tällöin solmun v kohdalla voidaan tutkia, minkä kaikkien hahmojen esiintymä loppuu senhetkiseen tekstinkohtaan mahdollisesti solmua v vastaavan hahmon lisäksi, sillä solmun v hahmo-osoitin $h_{ac}(v) = u$ viittaa näistä muista hahmon esiintymistä pisintä vastaavaan loppusolmuun.

Jos tällaisia muita esiintymiä on useampia kuin yksi, päästään edellisen kaltaisen solmun u hahmo-osoittimesta näistä esiintymistä seuraavaksi pisintä vastaavaan loppusolmuun ja siitä taas sitä seuraavaan jne. Lopulta ollaan sellaisessa solmussa, jonka hahmo-osoittimen arvo on v_r, eli ei ole enää jäljellä lyhyempää kyseiseen tekstinkohtaan päättyvää etsittävän hahmon esiintymää.

Määritellään nämä hahmo-osoittimet edellisen kuvailun mukaisesti, ja todetaan lauseessa 1.7.10 niiden oleellisin ominaisuus Aho-Corasick-algoritmin kannalta.

Määritelmä 1.7.9:
Aho-Corasick-algoritmin yhteydessä käytetty avainpuun solmujen hahmo-osoitin h_{ac}:
$h_{ac}(v) = u \mid (u = v_r \vee \exists x: hahmo_x = M_u) \wedge |M_u| = \max\{k \mid k = 0 \vee (0 < k < |M_v| \wedge \exists y: (hahmo_y[0\ldots k\text{-}1] = M_v[|M_v|\text{-}k\ldots|M_v|\text{-}1] \wedge k = |hahmo_y|))\}$.

Lause 1.7.10:
Merkkijono $hahmo_x$ on solmua v vastaavan merkkijonon aito loppuosa jos ja vain jos merkkijonoa $hahmo_x$ vastaava loppusolmu sijaitsee solmusta v alkavalla hahmo-osoittimien suunnatulla polulla.

Todistus:
Olkoot $u_1, u_2, \ldots, u_z$ solmusta v alkavan hahmo-osoittimien muodostaman polun solmut järjestyksessä alusta loppuun, eli $h_{ac}(v) = u_1$, $h_{ac}(u_1) = u_2$, $h_{ac}(u_2) = u_3$ jne. Koska avainpuussa on äärellinen määrä solmuja ja jokaiselle solmulle $v \neq v_r$ pätee ehto $|M_v| > |M_u|$, missä $u = h_{ac}(v)$, ja $h_{ac}(v_r) = v_r$, pätee aina $|M_{ux}| > |M_{ui+1}| \geq 0$, ja siten edellisen kaltainen polku on äärellinen. Lisäksi nähdään, että pitää olla $u_z = v_r$, ja määritelmän 1.7.9 mukaan jokaisella solmuparilla u_{x-1}, u_x, missä $1 < x \leq z$, pätee ehto $M_{ux-1} = mj_x \cup M_{ux}$, missä mj_x on jokin ei-tyhjä merkkijono ($M_{ux-1} = mj_x$ ja $M_{ux} = \varepsilon$ tapauksessa $x = z$, eli kun $u_x = u_z = v_r$).

Ei-tyhjä merkkijono $hahmo_x$ on merkkijonon M_v aito loppuosa jos ja vain jos pätee $M_v = mj \cup hahmo_x$ ja $0 < |hahmo_x| < |M_v|$, jolloin jälkimmäisen nojalla mj on ei-tyhjä merkkijono. Jos jollain edellisistä solmuista u_y pätee $|M_{uy}| = |hahmo_x|$, pätee edellisten huomioiden perusteella myös $M_v = mj_1 \cup mj_2 \cup \ldots \cup mj_y \cup M_{uy} = mj_1 \cup mj_2 \cup \ldots \cup mj_y \cup hahmo_x$, jolloin merkkijono $hahmo_x$ on merkkijonon M_v aito loppuosa.

Toisaalta jos $hahmo_x$ on merkkijonon M_v aito loppuosa, löytyy solmusta v alkavalta hahmo-osoittimien polulta sellaiset solmut u_{y-1} ja u_y, että $1 < y \leq z$ ja $|M_{uy-1}| \geq |hahmo_x| > |M_{uy}|$. Koska molemmat merkkijonot M_{uy-1} ja $hahmo_x$ ovat tällöin merkkijonon M_v aitoja

loppuosia, täytyy päteä joko $hahmo_x = M_{u_{y-1}}$, tai sitten $|M_{u_{y-1}}| > |hahmo_x|$ ja $hahmo_x$ on merkkijonon $M_{u_{y-1}}$ aito loppuosa. Edellisessä tapauksessa $M_{u_{y-1}}$ on merkkijonoa $hahmo_x$ vastaava loppusolmu, ja jälkimmäinen tilanne johtaa ristiriitaan arvon $h_{ac}(u_{y-1})$ määritelmän maksimaalisuuden kanssa, sillä merkkijonoa $hahmo_x$ vastaava loppusolmu täyttää tällöin solmulle $u_y = h_{ac}(u_{y-1})$ määritelmässä 1.7.9 asetetut ehdot, mutta $|hahmo_x| > |M_{u_y}|$. Siis edellisen mukaan solmusta v alkavalta hahmo-osoittimien muodostamalta polulta löytyy merkkijonoa $hahmo_x$ vastaava loppusolmu jos $hahmo_x$ on merkkijonon M_v aito loppuosa.

Lauseen väite voidaan todeta oikeaksi yhdistämällä edellisen tarkastelun päätelmät.

Sovelletaan nyt Aho-Corasick-algoritmia kokonaisuudessaan seuraavien sääntöjen mukaisesti:

1) Alussa lähdetään liikkeelle merkin *teksti*[0] ja avainpuun juurisolmun v_r kohdalta.

2) Merkin *teksti*[j] ja solmun v kohdalla edetään solmuun u ja merkkiin *teksti*[j+1] jos ja vain jos solmusta v lähtee merkkiä *teksti*[j] vastaava avainpuun suunnattu kaari solmuun u. Jos solmusta v ei lähde merkkiä *teksti*[j] kaarta, niin joko $v \neq v_r$, jolloin pysytään merkin *teksti*[j] kohdalla ja edetään solmuun $u = f_{ac}(v)$, tai sitten $v = v_r$ sekä $f_{ac}(v)$ = NULL, jolloin pysytään juurisolmussa edeten merkkiin *teksti*[j+1].

3) Siirryttäessä ensimmäistä kertaa merkin *teksti*[j] kohdalle tutkitaan ja ilmoitetaan heti saman tien kyseistä tilannetta vastaavan solmun v sisältämien tietojen perusteella kaikki sellaiset etsittävät hahmot, joiden esiintymä loppuu merkin *teksti*[j] kohdalle. Tämä tarkoittaa ensinnäkin, että jos v itsessään on loppusolmu, niin ilmoitetaan sitä vastaavan hahmon esiintymä löydetyksi, ja toiseksi, että ilmoitetaan solmun v hahmo-osoittimesta saavutettavissa olevia loppusolmuja vastaavat hahmon esiintymät löydetyiksi.

Edellisessä askeleessa 3) kaikki tekstin merkkiin *teksti*[j] päättyvät hahmon esiintymät tutkitaan heti sen solmun kohdalla, jossa ollaan välittömästi sen jälkeen, kun merkki *teksti*[j] on täsmätty ensimmäistä kertaa. Tämän menettelyn tarkoituksena on välttyä löytämästä samoja esiintymiä moneen kertaan, sillä muuten merkin *teksti*[j] kohdalla esiintyvän merkkijonon $hahmo_x$ esiintymä voitaisiin havaita esimerkiksi ensin sellaisen solmun v kohdalla, että $h_{ac}(v) = u$ ja $M_u = hahmo_x$, ja sitten heti perään itse solmun u kohdalla, jos $f_{ac}(v) = u$, ja solmusta v ei lähde merkkiä *teksti*[j+1] vastaavaa avainpuussa etenevää täsmäävää kaarta. Osoitetaan lauseessa 1.7.11 vielä edellisten askeleiden mukaisen Aho-Corasick-algoritmin toimivuus ennen itse käytännön toteutuksen tarkastelua.

Lause 1.7.11:
Aho-Corasick-algoritmi löytää kaikki etsittävien merkkijonojen $hahmo_1$, $hahmo_2$, …, $hahmo_h$ esiintymät tekstistä.

Todistus:
Olkoon s mielivaltainen tekstin indeksi. Todistetaan väite kahdessa osassa:

1) Oletetaan, että Aho-Corasick-algoritmi kirjaa tekstin kohtaan s päättyvän merkkijonon $hahmo_x$ esiintymän löydetyksi. Tämä tapahtuu, jos merkin $teksti[s]$ kohdalle tultaessa siirrytään sellaiseen solmuun u, että joko u on merkkijonoa $hahmo_x$ vastaava loppusolmu tai sitten solmusta u alkava hahmo-osoittimien muodostama suunnattu polku sisältää merkkijonon $hahmo_x$ loppusolmun. Koska lauseen 1.7.6 perusteella $M_u = teksti[s-|M_u|+1…s]$ ja lauseen 1.7.10 mukaan nyt $hahmo_x$ on merkkijonon M_u loppuosa, täytyy päteä $teksti[s-|hahmo_x|+1…s] = hahmo_x$. Siis Aho-Corasick-algoritmi kirjaa merkkijonon $hahmo_x$ esiintymän löydetyksi päättyen merkkiin $teksti[s]$ vain, jos todella pätee $teksti[s-|hahmo_x|+1…s] = hahmo_x$.

2) Oletetaan, että $teksti[s-|hahmo_x|+1…s] = hahmo_x[0…|hahmo_x|-1]$ eli merkkijonon $hahmo_x$ esiintymä loppuu tekstin kohtaan s. Olkoot v ja u ne avainpuun solmut, joissa ollaan heti ennen ja jälkeen saapumista merkin $teksti[s]$ kohdalle. Tällöin $u \neq v_r$, sillä muuten pätisi $v = v_r$, ja lauseen 1.7.7 perusteella olisi ehto $|M_v| + 1 = 1 > \max\{k \mid k = 0 \vee \exists y: hahmo_y[0…k-1] = teksti[s-k+1…s]\}$ voimassa, eli merkkiin $teksti[s]$ ei päättyisi minkään etsittävän (ja siten oletusarvoisesti ei-tyhjän) merkkijonon esiintymä. Toisaalta koska tekstissä edetään muun kuin juurisolmun v_r kohdalla oltaessa vain lauseen 1.7.7 ensimmäisen tapauksen mukaisesti, ovat nyt ehdot $|M_u| = |M_v| + 1 = \max\{k \mid k = 0 \vee \exists y: hahmo_y[0…k-1] = teksti[j-k+1…j]\}$ ja $M_u = teksti[j-|M_u|+1…j]$ voimassa. Jos $hahmo_x = M_u$, on u merkkijonoa $hahmo_x$ vastaava loppusolmu, ja Aho-Corasick-algoritmi kirjaa merkkijonon $hahmo_x$ esiintymän löydetyksi. Muussa tapauksessa pitää edellisen mukaan päteä ehto $|hahmo_x| < |M_u|$, ja koska lauseen 1.7.6 mukaan $M_u = teksti[s-|M_u|+1…s]$, on $hahmo_x$ merkkijonon M_u loppuosa. Siten lauseen 1.7.10 mukaan merkkijonoa $hahmo_x$ vastaava loppusolmu sijaitsee solmun u hahmo-osoittimesta alkavalla suunnatulla polulla, ja Aho-Corasick algoritmi kirjaa merkkijonon $hahmo_x$ esiintymän löydetyksi.

Kohdista 1) ja 2) seuraa, että Aho-Corasick-algoritmi kirjaa merkkijonon $hahmo_x$ esiintymän päättyväksi merkkiin $teksti[s]$ jos ja vain jos $hahmo_x = teksti[s-|hahmo_x|+1…s]$,

ja siten se löytää oikein kaikki etsittävien hahmojen esiintymät tekstistä, koska merkkijono $hahmo_x$ oli tässä mielivaltainen etsittävä merkkijono.

Varsinaisen Aho-Corasick-algoritmin etsintävaiheen esitys saadaan suoraviivaisesti mukaillen aiemmin esitettyjä askeleita 1) - 3) (algoritmi 1.7.12).

```
int j = 0;                                  // Tekstiä läpikäyvä indeksi, aloitetaan merkistä teksti[0].
int a;                                      // Käytettävän aakkoston merkkien lukumäärä.
int N(λ);                                   // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
solmu *juuri;                               // Osoitin avainpuun juurisolmuun.
solmu *apusolmu;                            // Avainpuun käsittelyssä käytetty apumuuttuja.
solmu *hahmosolmu;                          // hahmo-osoittimien käsittelyssä käytetty apumuuttuja.

apusolmu = juuri;                           // Aloitetaan kaarien seuraaminen juuresta.
while(j < n)                                //   Silmukka käy läpi koko tekstin alusta loppuun.
{
    while(j < n && apusolmu->kaaret[N(teksti[j])] != NULL)      // Täsmätään avainpuun
    {                                                           // sisältämiä hahmoja tekstiin.
        apusolmu = apusolmu-> kaaret[N(teksti[j])];             // Edetään avainpuussa.
        j++;                                                // Edetään vastaava askel myös tekstissä;
        if(apusolmu->loppu >= 0)                            // Saavuttiinko loppusolmuun?
        {
            loytyi(apusolmu->loppu);                        // Rekisteröidään merkkijono
        }                                                   // hahmot[apusolmu->loppu] löytyneeksi.
        hahmosolmu = apusolmu->hahmo_osoitin;           // Käydään läpi nykyisestä
        While(hahmosolmu != juuri)                      // solmusta alkava hahmo-osoittimien polku,
        {                                               // ja rekisteröidään kaikkien sillä olevien loppu-
            loytyi(hahmosolmu   ->loppu);               // solmujen vastinhahmot löydetyiksi.
            hahmosolmu = hahmosolmu->hahmo_osoitin;             // Edetään askel hahmo-
        }                                                       // osoitinpolulla.
    }
    if(apusolmu->f_ac != NULL)                  // Puussa ei voitu edetä, sovelletaan siis funktiota
    {                                           // f_ac nykyisen solmun kohdalla. Jos siirtymän arvo
        apusolmu = apusolmu->f_ac;              // ei ole NULL, ei olla avainpuun juurisolmussa ja
    }                                           // siten siirrytään puussa mutta ei tekstissä.
    else
    {                                               // Funktion f_ac antama siirtymä olisi NULL, joten
        j++;                                        // ollaan juurisolmun v_r kohdalla. Siis puussa ei
    }                                               // tehdä siirtymää, mutta edetään askel tekstissä.
}
```

Algoritmi 1.7.12:
Aho-Corasick, usean merkkijonon etsintäalgoritmi.

Algoritmin 1.7.12 aikavaativuus riippuu ensinnäkin siitä, kuinka monta kertaa kokonaisuudessaan hahmo-osoittimia läpikäyvää while-silmukkaa suoritetaan, ja toiseksi siitä, kuinka monta kertaa pääsilmukkaa voidaan suorittaa kokonaisuudessaan siten, että

indeksin *j* arvoa ei kasvateta. Edellisen lukumäärän ylärajana toimii se, kuinka monta eri hahmojen esiintymää tekstistä kokonaisuudessaan löydetään, ja jälkimmäiselle saadaan ylärajaksi *n* käyttäen hyvin samantapaista menettelyä kuin Morris-Pratt-algoritmin aikavaativuutta tutkittaessa. Nimittäin alussa ollaan solmun $v = v_r$ kohdalla, ja tällöin $M_v = 0$. Nyt indeksin *j* arvo pysyy ennallaan ainoastaan silloin, jos $|M_v| > 0$ ja avainpuussa ei tehdä täsmäävää siirtymää, ja tällöin arvo $|M_v|$ pienenee vähintään yhdellä funktion f_{ac} soveltamisen ansiosta (tässä siis viitataan koko ajan solmulla *v* nykyiseen avainpuun kohtaan). Toisaalta arvo $|M_v|$ kasvaa ainoastaan näiden täsmäävien siirtymien yhteydessä, ja silloinkin tasan yhdellä. Siten näitä arvoa *j* kasvattamattomia pääsilmukan kierroksia voi olla korkeintaan saman verran, kuin on tehty täsmääviä avainpuun siirtymiä. Näiden siirtymien lukumäärällä on yläraja *n*, koska jokainen täsmäävä siirtymä kasvattaa indeksin *j* arvoa ja *j*:*n* arvo ei missään vaiheessa pienene. Koska avainpuun muodostamisen aikavaativuus on $O(|\Lambda|\times\Sigma m)$, missä $|\Lambda|$ on käytettävän aakkoston koko ja Σm merkkijonojen $hahmo_1$, $hahmo_2$, …, $hahmo_h$ yhteispituus, ja myöhemmin todetaan, että tämän aikavaativuuden puitteissa on mahdollista alustaa myös avainpuun solmujen hahmo-osoittimet ja funktion f_{ac} arvot, saadaan Aho-Corasick-algoritmin kokonaisaikavaativuudeksi $O(|\Lambda|\times\Sigma m + n + k)$, missä *k* on tekstistä löytyvien esiintymien lukumäärä.

Edellä jo todettiin, että funktion f_{ac} arvot saadaan alustettua ajassa $|\Lambda|\times\Sigma m$. Tämä voidaan toteuttaa saman periaatteen mukaisesti kuin korjausfunktion f_k alustus Morris-Pratt-algoritmin yhteydessä eli soveltamalla Aho-Corasick-algoritmin toimintaperiaatetta sellaiseen tekstiin, joka vastaa sopivalla tavalla etsittävien merkkijonojen $hahmo_1$, $hahmo_2$, …, $hahmo_h$ loppuosia. Mutta ensin on hyvä todeta, että määritelmän 1.7.5 perusteella pätee selvästi $f_{ac}(v) = v_r$ jokaisen sellaisen solmun *v* kohdalla, että M_v vastaa jotain yhden merkin pituisista merkkijonoista $hahmo_1[0]$, $hahmo_2[0]$, …, $hahmo_h[0]$. Tämän perusteella siis tiedetään, että jos solmun *v* syvyys avainpuussa on 1 (eli juurisolmun v_r ja solmun *v* välisellä suunnatulla polulla on 1 kaari), niin pätee myös $f_{ac}(v) = v_r$. Näin ollen riittää tarkastella korjausfunktion f_{ac} arvojen laskemista avainpuussa vähintään syvyydellä 2 sijaitseville solmuille.

Olkoon *w* jokin merkkijonojen $hahmo_1$, $hahmo_2$, …, $hahmo_h$ muodostaman avainpuun solmu. Tarkastellaan tilannetta, jossa Aho-Corasick-algoritmia sovellettaessa siirrytään merkistä $teksti[j]$ merkin $teksti[j+1]$ kohdalle ja samalla avainpuussa siirrytään solmusta *v* solmuun *u*. Nyt lauseen 1.7.7 perusteella joko (ensimmäinen tapaus) $|M_u| = |M_v| + 1 = \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j\text{-}k+1\ldots j]\}$ tai sitten (jälkimmäinen tapaus) $|M_v| + 1 > \max\{k \mid k = 0 \vee \exists y: hahmo_y[0\ldots k\text{-}1] = teksti[j\text{-}k+1\ldots j]\}$, missä maksimilausekkeiden loppuosat sisältävät implisiittisesti ehdon $0 < k < j + 2$. Lisäksi ensimmäisessä tapauksessa $M_u = teksti[j\text{-}$

$|M_u|+1\ldots j]$. Jos tässä on voimassa täsmäävyys *teksti*$[0\ldots j] = M_w[1\ldots|M_w|-1]$, niin pätee $|M_w| = j+2$, ja ensimmäinen tapaus saa muodon $|M_u| = |M_v| + 1 = \max\{k \mid k = 0 \vee (0 < k < |M_w| \wedge \exists y$: *hahmo*$_y[0\ldots k-1] = M_w[|M_w|-k\ldots|M_w|-1]\}$ tai $|M_v| + 1 > \max\{k \mid k = 0 \vee (0 < k < |M_w| \wedge \exists y$: *hahmo*$_y[0\ldots k-1] = M_w[|M_w|-k\ldots|M_w|-1]\}$, ja lisäksi siinä pätee nyt $M_u = M_w[|M_w|-|M_u|\ldots|M_w|-1]$. Funktion f_{ac} määritelmän perusteella voidaan nyt havaita, että ensimmäisessä tapauksessa solmu u täyttää arvolle $f_{ac}(w)$ asetetut ehdot eli $f_{ac}(w) = u$. Toisaalta voidaan havaita, että yhtäsuuruus $f_{ac}(w) = u$ pätee myös jälkimmäisessä tapauksessa, sillä merkistä *teksti*$[j]$ siirrytään eteenpäin vain, jos joko tehdään täsmäävä siirtymä avainpuussa (vastaa ensimmäistä tapausta) tai jos ollaan juurisolmun kohdalla pystymättä etenemään merkkiä *teksti*$[j]$ vastaavaa kaarta pitkin. Siten tämän toisen vaihtoehdon on vastattava aiempaa jälkimmäistä tapausta, ja tällöin $u = v = v_r$ ja $|M_v| = 0$, mistä seuraa, että $1 > \max\{k \mid k = 0 \vee (0 < k\ |M_w| \wedge \exists y$: *hahmo*$_y[0\ldots k-1] = M_w[|M_w|-k\ldots|M_w|-1]\} = 0$, jolloin siis $f_{ac}(w) = v_r = u$.

Edellistä ajatusta voidaan soveltaa korjausfunktion f_{ac} arvojen yksikäsitteisyyden nojalla myös toiseen suuntaan: Jos tunnetaan arvo $f_{ac}(w) = u$ ja oletetaan ehdon *teksti*$[0\ldots j] = M_w[1\ldots|M_w|-1]$ olevan voimassa, niin Aho-Corasick-algoritmi on solmun u kohdalla välittömästi sen jälkeen, kun merkki *teksti*$[j]$ on käsitelty. Siten funktion f_{ac} arvo voidaan nyt laskea jokaiselle solmun w lapsisolmulle v jatkamalla Aho-Corasickin suoritusta tilanteesta, jossa seuraavaksi tutkittava merkki *teksti*$[j+1]$ on solmujen w ja v välisen kaaren vastinmerkki eli *teksti*$[0\ldots j+1] = M_w[1\ldots|M_w|-1] \cup M_v[|M_v|-1] = M_v[1\ldots|M_v|-1]$ ja ollaan solmun $u = f_{ac}(w)$ kohdalla. Aiemman perusteella arvo $f_{ac}(v)$ viittaa tällöin siihen solmuun, johon Aho-Corasick algoritmi siirtyy samalla, kun se siirtyy merkin *teksti*$[j+1] = M_v[|M_v|-1]$ kohdalta eteenpäin.

Tällainen menettely toimii luonnollisesti ainoastaan sillä edellytyksellä, että kaikki Aho-Corasick-algoritmin suorituksen aikana tarvitut funktion f_{ac} arvot tunnetaan ennen niiden soveltamista. Mutta tämä ei ole ongelma, sillä korjausfunktion arvo tunnetaan kaikille syvyydellä 1 sijaitseville solmuille, Aho-Corasick-algoritmi on korkeintaan syvyydessä $j+1$ ollessaan merkin *teksti*$[j+1]$ kohdalla ja aiemman mukaan funktion f_{ac} arvon laskemiseksi syvyydellä $j+2$ sijaitsevalle solmulle v tarvitsee Aho-Corasick-algoritmia suorittaa ainoastaan merkkiin *teksti*$[j+1]$ asti. Koska Aho-Corasick-algoritmi soveltaa korjausfunktiota ainoastaan sellaiseen solmuun, jonka kohdalla se on, voidaan siten korjausfunktion arvot alustaa kaikille syvyydellä $j+2$ sijaitseville solmuille, jos kaikki sitä matalammalla sijaitsevien solmujen korjausfunktion arvot jo tunnetaan. Eli koska syvyydellä 0 ja 1 olevien solmujen arvot jo tunnetaan, voidaan Aho-Corasick-algoritmia soveltaa turvallisesti merkkiin *teksti*$[2]$ asti ja siten alustaa korjausfunktion arvot myös kaikille syvyyden 2 solmuille. Tämän jälkeen samaa menettelyä voidaan jatkaa edeten askel kerrallaan syvyyksiin 3, 4, …, kunnes koko avainpuu

on käyty läpi ja korjausfunktion arvot on laskettu jokaiselle sen solmulle. Käytännössä tässä tehdään siis avainpuun leveyssuuntainen läpikäynti alkaen syvyydestä 1 siten, että aina, kun ollaan solmun *w* kohdalla ja solmusta *w* menee merkkiä λ vastaava kaari solmuun *v*, niin suoritetaan Aho-Corasick-algoritmia alkaen merkistä *teksti*[*j*+1] = λ = $M_v[|M_v|-1]$ ja solmusta $f_{ac}(w)$, kunnes algoritmin suorituksessa siirrytään tämän merkin *teksti*[*j*+1] kohdalta eteenpäin. Se avainpuun solmu, johon tämän tapahtuessa siirrytään, vastaa aiemman mukaan arvoa $f_{ac}(v)$.

Arvojen $f_{ac}(v)$ ja $h_{ac}(v)$ välillä on hieman samankaltainen suhde kuin korjausfunktioilla f_k ja f_{ko}, eli jälkimmäinen on muuten samantyyppinen kuin edellinen, mutta täyttää lisäksi jonkin lisäehdon. Siten saadaan seuraava lauseen 1.2.18 kaltainen sääntö, jonka mukaan arvoa $f_{ac}(v)$ asetettaessa on helppo asettaa myös arvo $h_{ac}(v)$.

Lause 1.7.14:
Oletetaan, että $v \neq v_r$ ja $f_{ac}(v) = w$. Tällöin $h_{ac}(v) = w$ jos *w* on jotain etsittävää merkkijonoa $hahmo_x$ vastaava loppusolmu, ja muussa tapauksessa $h_{ac}(v) = h_{ac}(w)$.

Todistus:
Määritelmistä 1.7.5 ja 1.7.9 nähdään, että jos $f_{ac}(v) = w = v_r$, niin myös $h_{ac}(v) = w = v_r = h_{ac}(v_r)$, joten tällöin lauseen väite on tosi. Oletetaan siis jatkossa, että $w \neq v_r$.

Tarkastellaan ensin tilannetta, jossa $f_{ac}(v) = w$ ja *w* on loppusolmu, eli $M_w = hahmo_x$, missä $hahmo_x$ on jokin etsittävistä merkkijonoista. Tällöin määritelmän 1.7.5 sekä edellisen nojalla $M_w = M_v[|M_v|-|M_w|\ldots|M_v|-1] = hahmo_x$ ja $|M_w| = \max\{k \mid k = 0 \vee (0 < k < |M_v| \wedge \exists y: hahmo_y[0\ldots k-1] = M_v[|M_v|-k\ldots|M_v|-1])\}$. Vertaamalla edellisiä määritelmän 1.7.9 kanssa nähdään, että tässä tapauksessa pätee $h_{ac}(v) = w$.

Tarkastellaan nyt tilannetta, jossa $f_{ac}(v) = w$ ja *w* ei ole loppusolmu. Tällöin selvästi $\max\{k \mid k = 0 \vee (0 < k < |M_v| \wedge \exists y: (hahmo_y[0\ldots k-1] = M_v[|M_v|-k\ldots|M_v|-1] \wedge k = |hahmo_y|))\} < |M_w|$, joten, koska oletusten perusteella $M_w = M_v[|M_v|-|M_w|\ldots|M_v|-1]$, pätee yhtäsuuruus $\max\{k \mid k = 0 \vee (0 < k < |M_v| \wedge \exists y: (hahmo_y[0\ldots k-1] = M_v[|M_v|-k\ldots|M_v|-1] \wedge k = |hahmo_y|))\} = \max\{k \mid k = 0 \vee (0 < k < |M_w| \wedge \exists y: (hahmo_y[0\ldots k-1] = M_w[|M_w|-k\ldots|M_w|-1] \wedge k = |hahmo_y|))\}$ ja siten pätee myös yhtäsuuruus $h_{ac}(v) = h_{ac}(w)$, sillä edellisen yhtäsuuruuden avulla saadaan tässä tapauksessa arvojen $h_{ac}(v)$ ja $h_{ac}(w)$ määritelmät identtisiksi.

Muokkamalla algoritmi 1.7.12 toimimaan avainpuun leveyssuuntaisen läpikäynnin yhteydessä ja korvaamalla merkin *teksti*[*j*+1] tutkiminen aina kulloistakin solmua *v* vastaavan

merkkijonon viimeisen merkin tutkimisella vastaten edellisessä tarkastelussa pätenyttä yhtäsuuruutta *teksti*[0…*j*+1] = M_v[|M_v|-1] saadaan algoritmin 1.7.15 esittämä korjausfunktion f_{ac} arvot alustava algoritmi 1.7.15. Tässä asetetaan lisäksi hahmo-osoittimet käyttäen lauseen 1.7.14 periaatetta eli asettamalla sijoituksen $f_{ac}(v) = w$ yhteydessä aina lisäksi $h_{ac}(v) = w$ jos u on loppusolmu, ja muuten $h_{ac}(v) = h_{ac}(w)$.

```
int i = 0;                                  // Indeksimuuttuja.
int j = 0;                                  // Apumuuttuja.
int a;                                      // Käytettävän aakkoston merkkien lukumäärä.
int N(λ);          // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
solmu *juuri;                               // Osoitin avainpuun juurisolmuun.
solmu *apusolmu;                            // Avainpuun käsittelyssä käytetty apumuuttuja.
solmu *acsolmu;                  // Funktion fac arvojen määrittelyssä käytetty apumuuttuja.
solmulista *nykyinensyvyys;                 // Avainpuun leveyssuuntaisen läpikäynnin apuna
solmulista *seuraavasyvyys;                 // käytettäviä solmulistoja.

apusolmu = juuri;                           // Aloitetaan avainpuun käsittely juurisolmusta.
juuri->hahmo_osoitin = juuri;               // Juuren hahmo-osoitin osoittaa juureen.
juuri->ac = NULL;                           // Juurta vastaava funktion f_ac arvo on NULL.
for(i = 0; i < a; i++)                  // Käydään ensin läpi syvyydellä 1 sijaitsevat solmut.
{
    if(juuri->kaaret[i] != NULL)
    {
        juuri->kaaret[i]->ac = juuri;       // Syvyydellä 1 sijaitsevalle solmulle pätee f_ac(v) = v_r.
        lisaa(juuri->kaaret[i], nykyinensyvyys);  // Muodostetaan lista ensimmäisen käsiteltävän
    }                                       // syvyyden, eli syvyyden 2 solmuista.
}
while(nykyinensyvyys != NULL)               // Jatketaan niin kauan kuin nykyinensyvyys-
{                                           // listassa on käsiteltäviä solmuja.
    while(nykyinensyvyys != NULL)           // Käsitellään kaikki nykyisen syvyyden solmut.
    {
        apusolmu = poista(nykyinensyvyys);  // Otetaan seuraava käsiteltävä solmu
                                            // nykyinensyvyys-listasta.
        j = N(apusolmu->merkki);            // Muuttujaan j asetetaan käsiteltävänä olevan
                                            // solmun tulokaarta vastaavaa merkkiä
                                            // vastaava lukuarvo.
        acsolmu = apusolmu->isasolmu;       // Aloitetaan käsiteltävää solmua vastaavan
                                            // arvon fac etsiminen sen isäsolmusta.

        // Mukaillaan Aho-Corasick algoritmin suoritusta, kunnes se joko täsmää
        // käsiteltävään solmuun saapuvan kaaren vastinmerkin, tai saavutaan
        // juurisolmuun.
        while(acsolmu->ac != NULL && acsolmu->ac->kaaret[j] == NULL)
        {
            acsolmu = acsolmu->ac;          // Ei täsmännyt, sovelletaan siis funktiota f_ac.
        }
```

Algoritmi 1.7.15 (jatkuu seuraavalla sivulla):
Aho-Corasick-algoritmin korjausfunktion f_{ac} alustaminen sekä hahmo-osoittimien asettaminen jo luodussa avainpuussa.

```
        if(acsolmu->ac == NULL)                    // Saavuttiinko juurisolmuun?
        {
            apusolmu->ac = juuri;                  // Saavuttiin, siis käsiteltävää solmua vastaava
        }                                          // funktion f_ac arvo on v_r.
        else
        {                                  // Ei saavuttu, vaan onnistuttiin täsmäämään arvon j vastinmerkki.
            apusolmu->ac = acsolmu->ac->kaaret[j];  // Käsiteltävää solmua vastaava funktion f_ac
        }                                          // arvo on siis merkin j täsmäystä vastaavan
                                                   // kaaren päätesolmu.
        if(apusolmu->ac->loppu >= 0)               // Asetetaan hahmo-osoitin lauseen 1.7.14
        {                                          // mukaisesti.
            apusolmu->hahmo_osoitin = apusolmu->ac;
        }
        else
        {
            apusolmu->hahmo_osoitin = apusolmu->ac->hahmo_osoitin;
        }
        for(i = 0; i < a; i++)                     // Lisätään kaikki käsiteltävän solmun lapsisolmut
        {                                          // seuraavalla syvyydellä olevien solmujen listaan.
            if(apusolmu->kaaret[i] != NULL)
            {
                lisaa(seuraavasyvyys, apusolmu->kaaret[i];
            }
        }
        nykyinensyvyys = seuraavasyvyys;           // Siirrytään seuraavaan syvyyteen avainpuun
                                                   // leveyssuuntaisessa läpikäynnissä.
        seuraavasyvyys = NULL;                     // Nollataan seuraavasyvyys.
    }
}
```

Algoritmi 1.7.15 (jatkoa):
Aho-Corasick-algoritmin korjausfunktion f_{ac} alustaminen sekä hahmo-osoittimien asettaminen jo luodussa avainpuussa.

Algoritmin 1.7.15 aikavaativuus on $O(|\Lambda| \times \Sigma m)$, missä $|\Lambda|$ on käytettävän aakkoston koko ja Σm merkkijonojen $hahmo_1$, $hahmo_2$, …, $hahmo_h$ yhteispituus. En todista sitä tässä täsmällisemmin, mutta asia voidaan havaita samantapaisella tarkastelulla, jota käytettiin algoritmin 1.7.12 yhteydessä. Algoritmissa 1.7.15 voidaan soveltaa funktiota f_{ac} korkeintaan niin monta kertaa, kuin avainpuussa on tehty täsmääviä askeleita, ja näitä täsmääviä askeleita voidaan tehdä avainpuun solmujen lukumäärän verran eli $O(\Sigma m)$ kappaletta. Aikavaativuudeksi tulee näin ollen $O(|\Lambda| \times \Sigma m)$, sillä leveyssuuntaisessa avainpuun läpikäynnissä käsitellään $O(\Sigma m)$ solmua, jokaisen kohdalla tehdään aikavaativuuden $O(|\Lambda|)$ omaava työmäärä, ja edellisen mukaan sisin eli funktiota f_{ac} soveltava silmukka ei aiheuta tämän aikavaativuuden ylittävää työmäärää.

2. Likimääräinen merkkijonotäsmäys

2.1 Editointietäisyys

Kahden merkkijonon mj_1 ja mj_2 editointietäisyys on pienin sellainen kokonaisluku e, että mj_1 ja mj_2 voidaan saattaa identtisiksi alla tarkemmin määriteltyjä sallittuja editointi-operaatioita käyttäen siten, että käytettyjen operaatioiden tekemän työn kustannus on korkeintaan e. Otetaan tässä käyttöön merkintä $E(mj_1, mj_2)$ tarkoittamaan merkkijonojen mj_1 ja mj_2 välistä editointietäisyyttä. Jokaiselle eri operaatiolle *op* on määritelty sen tekemä työ, josta käytetään tässä merkintää $\omega(op)$. Tässä on aina perusoletuksena, että työ ei voi olla negatiivista tai ilmaista eli jokaiselle editointioperaatiolle *op* pätee $\omega(op) > 0$. Monen operaation yhteinen työmäärä (eli kustannus) saadaan yksinkertaisesti summaamalla yhteen yksittäisten operaatioiden tekemät työt. Sallitut editointioperaatiot ovat:

- Yhden merkin poisto. Käytetään merkintää $P(mj, i)$ tarkoittamaan merkkijonon *mj* indeksin *i* kohdalla olevan merkin poistamista. Siis:
  $P(mj, i)$: $mj \rightarrow mj[0\ldots i\text{-}1] \cup mj[i\text{+}1\ldots m\text{-}1]$.

- Yhden merkin lisäys. Käytetään merkintää $L(mj, i, \lambda)$ tarkoittamaan merkin λ lisäystä merkkijonon *mj* merkin $mj[i]$ jälkeiseen kohtaan. Siis:
  $L(mj, i, \lambda)$: $mj \rightarrow mj[0\ldots i] \cup \lambda \cup mj[i\text{+}1\ldots m\text{-}1]$.

- Yhden merkin korvaus, joka vastaa yhden merkin poistoa, jonka perään tehdään heti lisäys samaan kohtaan. Käytetään merkintää $K(mj, i, \lambda)$ tarkoittamaan merkkijonon *mj* indeksin *i* kohdalla olevan merkin korvaamista merkillä λ. Siis:
  $K(mj, i, \lambda)$: $mj \rightarrow mj[0\ldots i\text{-}1] \cup \lambda \cup mj[i\text{+}1\ldots m\text{-}1]$.

Joissain yhteyksissä tietyn editointioperaation tekemä työ voidaan asettaa riippuvaksi myös sen käsittelemistä merkeistä (esim. [Needleman and Wunsch, 1970], [Ukkonen, 1985a]), jolloin esimerkiksi operaatioilla $L(mj, i, \text{'a'})$ ja $L(mj, i, \text{'b'})$ voisi olla eri kustannus riippuen merkeistä 'a' ja 'b'. Tämän käsittelyn yhteydessä perusoletuksena pidetään kuitenkin, että operaatioiden kustannukset ovat niiden käsittelemistä merkeistä riippumattomia. Lisäksi oletetaan ensinnäkin, että aina $\omega(K) < \omega(P) + \omega(L)$, sillä muuten koko korvausoperaatio jäisi turhaksi, koska se voitaisiin kustannuksia ajatellen aina korvata peräkkäin tehdyillä poisto- ja lisäysoperaatioilla. Toiseksi oletetaan, että $\omega(P) = \omega(L)$, sillä editointietäisyys on luonnollista

määritellä symmetriseksi relaatioksi, ja muuten voisi päteä esimerkiksi E(”*a*”, ”ab”) ≠ E(”ab”, ”*a*”).

Edellä esitetyssä lisäysoperaation määritelmässä on ongelmana merkin lisääminen merkkijonon *mj* alkuun. Tällaisen tilanteen käsittelemiseksi tulkitaan merkki *mj*[-1] tarkoittamaan tyhjää merkkiä ε, jolloin *mj*[-1] viittaa kohtaan ennen merkkijonon *mj* ensimmäistä merkkiä *mj*[0] ja siis operaatio L(*mj*, -1, λ) lisää merkin λ merkkijonon *mj* alkuun. Merkki ε tarkoittaa tässä siis ”näkymätöntä tyhjää tilaa”, jota ei oteta lainkaan huomioon. Esimerkiksi merkkijonot ”abba” ja ”*a*εbbε*a*” tulkitaan näin ollen identtisiksi, ja molempien pituudeksi katsotaan 4 merkkiä.

Usein oletuksena on, että kaikkien operaatioiden tekemä työ on 1, eli ω(P) = ω(L) = ω(K) = 1, jolloin operaatioiden tekemä työ = operaatioiden lukumäärä. Tässä tilanteessa merkkijonojen mj_1 ja mj_2 välinen editointietäisyys tarkoittaa pienintä sellaista lukua *e*, että mj_1 ja mj_2 voidaan saattaa identtisiksi *e* kappaletta editointi-operaatioita tekemällä. Yksinkertaisuuden vuoksi jokaisen tässä esitettävän esimerkin yhteydessä tämä oletus ω(P) = ω(L) = ω(K) = 1 on voimassa.

---

1. Korvataan merkkijonon ”antura” merkki ’n’ merkillä ’p’:
   K(”antura”, 1, ’p’): ”antura” → ”aptura”.

2. Poistetaan merkkijonon ”aptura” merkki ’t’:
   P(”aptura”, 2): ”aptura” → ”apura”.

3. Lisätään merkkijonon ”apura” loppuun merkki ’h’:
   L(”apura”, 4, ’h’): ”apura” → ”apurah”.

4. Lisätään merkkijonon ”apurah” loppuun merkki ’a’:
   L(”apurah”, 5, ’a’): ”apurah” → ”apuraha”.

Esimerkki 2.1.1:
Kun ω(P) = ω(L) = ω(K) = 1, on merkkijonojen ”antura” ja ”apuraha” editointietäisyys 4. Yllä on esitetty yksi tällä operaatioiden lukumäärällä toimiva editointien ketju.

Hahmon likimääräinen esiintymä määritellään nyt seuraavasti:

Määritelmä 2.1.2:
Hahmolla sanotaan olevan tekstin kohtaan *j* päättyvä likimääräinen esiintymä jos ja vain jos jollain positiivisella luvulla *h* pätee E(*hahmo*[0…*m*-1], *teksti*[*j*-*h*+1…*j*]) ≤ s_e, missä s_e on ennalta määrätty suurin sallittu virhe editointietäisyytenä mitattuna.

Käytännön kannalta oleellisin ero hahmon likimääräisten ja eksaktien esiintymien haussa on se, että likimääräisessä tapauksessa ei voida muodostaa tehokkaita apufunktioita turhien vertailujen määrän pienentämiseksi, ja nopea eksakti etsintä pohjautuu nimenomaan tällaisten käyttöön. Samalla itse editointietäisyyden laskeminen mahdollisimman tehokkaasti aina tietyssä hahmon sijaintikohdassa tekstin suhteen on tavallista vertailua huomattavasti monimutkaisempi ongelma. Editointietäisyyden laskemisessa dynaaminen lähestymistapa tai muunlainen ongelman alatapauksiin pilkkominen on melkeinpä käytännön pakon sanelema, sillä suoraan kokonaisten merkkijonojen välisen editointietäisyyden laskeminen johtaa helposti polynomiaaliseen tai eksponentiaaliseen laskenta-aikaan johtuen erilaisten mahdollisten korkeintaan editointietäisyyden s_e päässä merkkijonosta *mj* olevien merkkijonojen valtavasta lukumäärästä. Jos esimerkiksi käytössä olevan aakkoston koko on |Λ| ja $\lfloor s_e / \omega(K) \rfloor = k$, niin jokaista *m* merkkiä pitkää merkkijonoa kohti löytyy pelkästään korvausoperaatioita käyttäen helposti ainakin $(|\Lambda| - 1)^k \times \binom{m}{k}$ sellaista muuta merkkijonoa, jotka ovat siitä korkeintaan etäisyydellä s_e. Edellinen on laskettu luettelemalla, kuinka monta merkin korvausta jollain muulla merkillä hahmoon voidaan kohdistaa siten, että mitään hahmossa jo korvattua merkkiä ei korvata enää uudelleen. Edellisestä johtuen siis esimerkiksi sellainen ”brute force”-menetelmä, joka kävisi läpi kaikki mahdolliset tietyllä etäisyydellä tutkitusta merkkijonosta olevat muut merkkijonot, olisi laskenta-ajallisesti toivottoman hidas.

Editointietäisyyden sekä likimääräisen haun ratkaisemiseksi on kuitenkin olemassa useita kohtalaisen tehokkaita algoritmeja, ja keskityn tässä dynaamiseen taulukointiin perustuvaan lähestymistapaan, joka on käytännöllinen ja parhaimmillaan hyvinkin tehokas. Kaikki kirjallisuudesta löytämäni algoritmit perustuvat joka tapauksessa jonkinlaiseen taulukkomaiseen lähestymistapaan, mutta monet niistä on tässä sivuutettu niiden kapea-alaisuuden takia (pohjautuvat oletukseen että ω(P) = ω(L) = ω(K) = 1, näistä tehokkain esimerkki on Myersin ns. ”bitti-vektori”-algoritmi [Myers, 1998]). Lisäksi monet on sivuutettu, koska ne pohjautuvat vahvasti ns. suffiksipuun käyttöön (esimerkiksi [Landau and Vishkin, 1989]), joka on kyllä teoreettisesti tehokas mutta käytännössä tilaa vievä tietorakenne (esim. [Navarro, 1998]), ja suffiksipuun tehokas muodostaminen on jo itsessään

turhan laaja asia tarkasteltavaksi tässä yhteydessä pelkästään kattavamman käsittelyn aikaansaamiseksi.

Lisäksi käsittelen hieman likimääräisten esiintymien etsimisen tehostamisen kannalta lupaavinta tekniikkaa, jonka tavoitteena on karsia nopeasti sellaiset tekstin osat pois, joissa hahmon likimääräistä esiintymää ei ainakaan voi olla. Karsinnan jälkeen vielä mahdolliset esiintymäkohdat pitää tutkia tavalliseen tapaan, eli esimerkiksi jollain tässä osiossa esitellyistä likimääräisiä esiintymiä etsivistä algoritmeista. Aloitetaan editointietäisyyden yleisellä tarkastelulla.

Merkinnöistä

Otetaan käyttöön merkintä *mj*’ tarkoittamaan merkkijonon *mj* muotoa ennen viimeisintä siihen kohdistunutta toimenpidettä sekä merkintä mj^0 tarkoittamaan merkkijonon *mj* alkuperäistä muotoa ennen yhtään siihen kohdistunutta editointioperaatiota. Käytetään vastaavasti näiden merkkijonojen *mj*’ ja mj^0 pituuksista merkintöjä *m*’ ja m^0. Tarkoitetaan identtisyysoperaattorilla ≡ sellaista kahden merkkijonon välistä suhdetta, että $mj1[q…r] \equiv mj2[s…r\text{-}q\text{+}s]$ jos ja vain jos kyseessä on täsmälleen samojen merkkijonojen ilmentymät. Tässä ei siis tarkoiteta vain samanlaisia merkkijonoja, joita vastaa operaattori =. Esimerkiksi jos alkuperäinen $mj = mj^0$ = ”abba” ja toimenpiteen yhteydessä tehdään poisto-operaatio P(*mj*, 2), on tämän jälkeen *mj* = ”aba”, $mj' = mj^0$ = ”abba”, *m* = 3 ja *m*’ = m^0 = 4. Lisäksi tällöin $mj[0…1] \equiv mj'[0…1] \equiv mj^0[0…1]$ sekä $mj[2] \equiv mj'[3] \equiv mj^0[3]$, koska kyse on täsmälleen samojen merkkien ilmentymistä, joiden indeksit vain eivät enää täsmää suoritetun operaation ansiosta. Esimerkiksi ei siis päde $mj[1] \equiv mj'[2] \equiv mj^0[2]$, sillä merkki $mj[1]$ viittaa ensimmäiseen ’b’-kirjaimeen eli merkkeihin $mj'[1]$ ja $mj^0[1]$. Jos lisäksi tehdään vielä korvausoperaatio K(*mj*, 2, ’c’), on tämän jälkeen *mj* = ”abc”, *mj*’ = ”aba”, mj^0 = ”abba”, *m* = 3, *m*’ = 3 ja m^0 = 4.

2.2 Kahden merkkijonon välisen editointietäisyyden laskeminen

Tarkastellaan merkkijonon mj_1 muuttamista merkkijonoksi mj_2 editointioperaatioita käyttämällä. Yksi tapa on siirtyä merkkijonossa mj_1 askel kerrallaan alusta loppuun ja päättää aina jokaisella askeleella, mikä operaatio siinä kohtaa tarvitaan. Samalla tehdyt operaatiot ja niiden aiheuttamat kustannukset merkitään muistiin. Toteutetaan läpikäynti siten, että aloitetaan merkkiä $mj_1[0]$ edeltävän tyhjän merkin ε kohdalta eli paikasta $mj_1[-1]$. Merkin $mj_1[i]$ kohdalla oltaessa tarkastellaan merkkejä $mj_1[i+1]$ ja $mj_2[i+1]$, ja pyritään editointioperaatioiden avulla saamaan ne keskenään täsmääviksi. Valitaan merkin $mj_1[i]$ kohdalla tehtävä toimenpide seuraavista neljästä vaihtoehdosta:

1) Tyhjä operaatio, jos merkit $mj_1[i+1]$ ja $mj_2[i+1]$ täsmäävät jo valmiiksi.
2) Merkin $mj_1[i+1]$ poisto.
3) Merkin $mj_2[i+1]$ lisääminen merkin $mj_1[i]$ perään.
4) Merkin $mj_1[i+1]$ korvaaminen merkillä $mj_2[i+1]$.

Nämä vaihtoehdot kattavat siis sallitut kolme eri editointioperaatiota ja tyhjän operaation. Merkkijonossa mj_1 siirrytään eteenpäin merkin $mj_1[i+1]$ kohdalle jos ja vain jos toimenpiteeseen sisältyvän operaation jälkeen tiedetään, että $mj_1[i+1] = mj_2[i+1]$, eli tapauksissa 1), 3) ja 4). Mitään muunlaisia siirtymiä ei tehdä, ja näin ollen merkkijonossa mj_1 ei ikinä peruuteta. Kun edellisten lisäksi huomataan, että kohdassa $mj_1[i]$ tehty operaatio voi muuttaa ainoastaan merkkiä $mj_1[i+1]$, voidaan todeta seuraava lause:

Lause 2.2.1:
Edellisiä kohtien 1) – 4) sääntöjä noudatettaessa pätee aina merkin $mj_1[i]$ kohdalla oltaessa, että $mj_1 = mj_1[0\ldots m_1\text{-}1] = mj_2[0\ldots i] \cup mj_1{}^0[m_1{}^0\text{-}(m_1\text{-}(i+1))\ldots m_1{}^0\text{-}1]$.

Todistus:
Lause 2.2.1 seuraa seuraavista kahdesta seikasta:

1) Alkutilanteessa $mj_1[-1] = mj_2[-1] = \varepsilon$, aina merkin $mj_1[i]$ kohdalle siirryttäessä pätee $mj_1[i] = mj_2[i]$, siirrot tapahtuvat vain eteenpäin, ja merkin $mj_1[i]$ kohdalla oltaessa voidaan muuttaa vain merkkiä $mj_1[i+1]$, joten induktiolla nähdään, että aina merkin $mj_1[i]$ kohdalle siirtymisen jälkeen pätee $mj_1[0\ldots i] = mj_2[0\ldots i]$.

2) Siirtymät tapahtuvat aina eteenpäin, ja merkin $mj_1[i]$ kohdalta eteenpäin siirryttäessä voidaan editointioperaatio kohdistaa vain merkkiin $mj_1[i+1]$. Tästä seuraa, että merkin $mj_1[i]$ kohdalle saavuttaessa kaikki siihen mennessä tehdyt operaatiot on tehty merkkijonon $mj_1[0\ldots i\text{-}1]$ merkkien kohdalla ja siten ne ovat voineet kohdistua vain merkkijonon $mj_1[0\ldots i]$ alueelle. Toisaalta ainoa merkin $mj_1[i]$ kohdalla pysyvä toimenpide on merkin $mj_1[i+1]$ poistaminen, ja tämän toimenpiteen jälkeen $m_1 = m_1' - 1$ ja $mj_1[i+1\ldots m_1\text{-}1] = mj_1'[i+2\ldots m_1'\text{-}1]$. Täten merkkijonon mj_1 loppuosa $mj_1[i+1\ldots m_1\text{-}1]$ on siis vielä koskematon eli identtinen merkkijonon mj_1^0 vastaavanpituisen loppuosan kanssa. Kyseisen merkkijonon mj_1 loppuosan pituus on $m_1 - (i + 1)$ merkkiä, jolloin merkkijonon mj_1^0 vastaavanpituinen loppuosa on merkkijono $mj_1^0[m_1^0\text{-}(m_1\text{-}(i+1))\ldots m_1^0\text{-}1]$. Siis tässä tapauksessa $mj_1[i+1\ldots m_1\text{-}1] \equiv mj_1^0[m_1^0\text{-}(m_1\text{-}(i+1))\ldots m_1^0\text{-}1]$.

Lauseesta 2.2.1 nähdään, että jos merkkijonossa mj_1 saavutaan kohtaan $mj_1[m_2\text{-}1]$, pätee $mj_1 = mj_1[0\ldots m_1\text{-}1] = mj_2[0\ldots m_2\text{-}1] \cup mj^0[m_1^0\text{-}(m_1\text{-}m_2)\ldots m_1^0\text{-}1] = mj_2 \cup mj^0[m_1^0\text{-}(m_1\text{-}m_2)\ldots m_1^0\text{-}1]$. Jos tällöin $(m_1 - m_2) > 0$, pitää merkin $mj_1[m_2\text{-}1]$ kohdalla vielä suorittaa $(m_1 - m_2)$ kappaletta poisto-operaatioita haluttuun tulokseen eli täsmäävyyteen $mj_1 = mj_2$ pääsemiseksi.

Kohdissa *X*.1 – *X*.4 on nyt esitetty yksityiskohtaisemmin kohtien 1) – 4) mukaiset vaihtoehdot merkin $mj_1[i]$ kohdalla tehtävälle toimenpiteelle.

X.1 Jos $mj_1[i+1] = mj_2[i+1]$, ei tehdä operaatiota ja siirrytään suoraan seuraavaan kohtaan eli merkkiin $mj_1[i+1]$. Koska tämä toimenpide ei sisällä operaatiota, on sen kustannus 0. Toimenpide ei muuta merkkijonon mj_1 pituutta, joten sen suorittamisen jälkeen $m_1 = m_1'$.

X.2 Poistetaan merkki $mj_1[i+1]$ ja pysytään merkin $mj_1[i]$ kohdalla. Toimenpide sisältää yhden operaation, joka on poisto, joten sen kustannus on $\omega(P)$. Toimenpide vähentää merkkijonon mj_1 pituutta yhdellä, joten sen suorittamisen jälkeen $m_1 = m_1' - 1$.

X.3 Lisätään merkki $mj_2[i+1]$ merkin $mj_1[i]$ perään eli kohtaan $mj_1[i+1]$, ja siirrytään askel eteenpäin tämän lisätyn merkin kohdalle. Toimenpide sisältää yhden operaation, joka on lisäys, joten sen kustannus on $\omega(L)$. Toimenpide kasvattaa merkkijonon mj_1 pituutta yhdellä, joten sen suorittamisen jälkeen $m_1 = m_1' + 1$.

X.4 Korvataan merkki $mj_1[i+1]$ merkillä $mj_2[i+1]$ ja siirrytään askel eteenpäin tämän korvatun merkin kohdalle. Toimenpide sisältää yhden operaation, joka on korvaus, joten sen kustannus on $\omega(K)$. Toimenpide ei muuta merkkijonon mj_1 pituutta, joten sen suorittamisen jälkeen $m_1 = m_1'$.

Editointietäisyyden kannalta on oleellista huomata, että kohtien *X*.1 – *X*.4 mukaisilla askeleilla on aina mahdollista päätyä sellaiseen editointioperaatiojonoon, että kyseiset operaatiot tekevät pienimmän mahdollisen työmäärän, eli $E(mj_1^0, mj_2^0)$, muuttaessaan merkkijonon mj_1 merkkijonoksi mj_2.

Lause 2.2.2:
Kohtien *X*.1 – *X*.4 mukaisella merkkijonon mj_1 käsittelyllä voidaan aina toteuttaa merkkijonojen mj_1 ja mj_2 välistä editointietäisyyttä $E(mj_1, mj_2)$ vastaava editointioperaatiojono.

Todistus:
Olkoon O jokin sellainen operaatiojoukko että se muuttaa merkkijonon mj_1^0 merkkijonoksi mj_2 tehden minimaalisen määrän eli yhteensä luvun $E(mj_1^0, mj_2^0)$ verran työtä. Jaetaan joukko O osajoukkoihin O_P, O_L ja O_K siten, että O_P sisältää kaikki joukon O poisto-operaatiot ja vastaavasti O_L lisäysoperaatiot sekä O_K korvausoperaatiot. Käsitellään jokainen joukko O_P, O_L ja O_K erikseen:

O_P: Koska joukon O operaatiot tekevät minimaalisen määrän työtä, tiedetään, että kaikki sen sisältämät poisto-operaatiot kohdistuvat johonkin merkkijonon mj_1^0 merkkiin. Jos näin ei olisi, poistettaisiin joku merkkijonoon mj_1^0 lisätty merkki, ja silloin tämä aiempi lisäys olisi turha, ja joukko O ei olisi minimaalinen. Kohtien *X*.1 – *X*.4 mukaisesta merkkijonon mj_1 läpikäynnistä on helppo huomata, että siinä jokainen merkkijonon mj_1^0 merkki on jossain vaiheessa tarkastelun alla ja voidaan haluttaessa poistaa. Täten joukon O_P poistamat merkit voidaan poistaa myös läpikäynnin yhteydessä.

O_K: Samalla tavalla kuin edellisessä kohdassa voidaan todeta, että myös kaikki joukon O korvausoperaatiot kohdistuvat johonkin merkkijonon mj_1^0 merkkiin, ja näin ollen joukon O_K korvaamat merkit voidaan korvata myös läpikäynnin yhteydessä.

O_L: Merkin lisäys voidaan tehdä joko jo alunperin merkkijonossa $mj_1{}^0$ olleen merkin perään, korvatun merkin perään tai lisätyn merkin perään. Koska selvästi merkkijonon mj_1 läpikäynnissä käydään jokaisen sellaisen merkkijonon mj_1 merkin kohdalla, jota ei ole poistettu tai korvattu, ja korvauksen yhteydessä siirrytään aina lisätyn tai korvatun merkin kohdalle, voidaan näissä kaikissa tapauksissa joukon O_P lisäämät merkit lisätä myös läpikäynnin yhteydessä.

Edellisen mukaan siis joukon O operaatioita vastaavat editoinnit voidaan tehdä myös merkkijonon mj_1 kohtien $X.1 - X.4$ mukaisen läpikäynnin yhteydessä, ja näin lause 2.2.2 on tosi.

Kohtien $X.1 - X.4$ mukainen merkkijonon mj_1 läpikäynti voidaan kätevästi esittää taulukon avulla. Käytetään näiden sääntöjen mukaisen merkkijonon mj_1 läpikäynnin kuvaamiseen kooltaan $(m_1{}^0 +1)\times(m_2{}^0 + 1)$ olevaa taulukkoa. Tässä tosin merkintöjen $mj_2{}^0$ ja $m_2{}^0$ käyttö on siinä mielessä turhaa, että tehtävät operaatiot kohdistuvat ainoastaan merkkijonoon mj_1, ja siten merkkijono mj_2 sekä sen pituus pysyvät vakioina. Muodostetaan taulukon indeksit siten, että rivit numeroidaan luvusta -1 lukuun $m_1{}^0$ ja sarakkeet luvusta -1 lukuun $m_2{}^0$. Kuvassa 2.2.3 on esimerkkinä merkkijonoja $mj_1{}^0$ = ”antura” ja $mj_2{}^0$ = ”apuraha” vastaava tyhjä taulukko.

	j	-1	0	1	2	3	4	5	6
i		ε	’a’	’p’	’u’	’r’	’a’	’h’	’a’
-1	ε								
0	’a’								
1	’n’								
2	’t’								
3	’u’								
4	’r’								
5	’a’								

Kuva 2.2.3:
Merkkijonoja $mj_1{}^0$ = ”antura” ja $mj_2{}^0$ = ”apuraha” vastaava tyhjä taulukko.

Tulkitaan taulukon indeksit siten, että alkion (i, j) kohdalla oleminen vastaa tilannetta, jossa vertailun kohteena ovat merkit $mj_1^0[i+1]$ ja $mj_2^0[j+1]$, ja alkion (i, j) arvo kertoo siihen mennessä tehtyjen askeleiden yhteydessä tehdyn työn kokonaismäärän. Tällöin taulukon alkio (-1, -1) vastaa alkutilannetta, jossa merkkijonoissa mj_1 ja mj_2 ollaan tyhjien alkumerkkien ε kohdalla ja tarkasteltavina ovat merkit $mj_1[0]$ ja $mj_2[0]$. Koska alussa ei ole vielä tehty työtä, on alkion (-1, -1) arvo 0.

Taulukon indeksointi perustuu alkuperäisiin merkkijonoihin mj_1^0 ja mj_2^0, joten merkkijonoa mj_1 pitkin edettäessä pitää tietää, mitä merkkijonojen mj_1^0 ja mj_2^0 indeksejä aina merkin $mj_1[i]$ kohdalla oltaessa seuraavaksi tarkasteltavat merkit $mj_1[i+1]$ ja $mj_2[i+1]$ vastaavat. Jälkimmäinen ei ole ongelma, sillä koska merkkijonoon mj_2 ei kohdisteta yhtään operaatiota, pätee aina $mj_2[i+1] \equiv mj_2^0[i+1]$. Merkin $mj_1[i+1]$ alkuperäinen sijainti merkkijonossa mj_1^0 saadaan käyttäen apuna lausetta 2.2.1, jonka mukaan aina merkin $mj_1[i]$ kohdalla oltaessa $mj_1 = mj_1[0\ldots m_1\text{-}1] = mj_2[0\ldots i] \cup mj^0[m_1^0\text{-}(m_1\text{-}(i+1))\ldots m_1^0\text{-}1]$. Tästä nähdään, että merkin $mj_1[i]$ kohdalla $mj_1[i+1] \equiv mj_1^0[m_1^0\text{-}(m_1\text{-}(i+1))]$.

Lähdetään tilanteesta, jossa tarkastelun kohteena ovat merkit $mj_1^0[i]$ ja $mj_2^0[j] = mj_2[j]$. Koska mielivaltaisen merkin $mj_1[k]$ kohdalla oltaessa tarkastellaan aina merkkejä $mj_1[k+1]$ ja $mj_2[k+1]$, huomataan, että tässä tilanteessa $mj_1^0[i] \equiv mj_1[j]$, ja ollaan merkin $mj_1[j\text{-}1]$ kohdalla. Koska merkit $mj_1[j]$ ja $mj_1^0[i]$ vastaavat tässä tapauksessa toisiaan ja lauseen 2.2.1 mukaan $mj_1 = mj_1[0\ldots m_1\text{-}1] = mj_2[0\ldots i] \cup mj_1^0[m_1^0\text{-}(m_1\text{-}(i+1))\ldots m_1^0\text{-}1]$, voidaan tässä tapauksessa todeta seuraava lause.

Lause 2.2.4:
Kun vertailuvuorossa olevia merkkijonojen mj_1 ja mj_2 merkkejä vastaavat merkkijonojen mj_1^0 ja mj_2^0 merkit ovat $mj_1^0[i]$ ja $mj_2^0[j]$, ollaan merkkijonossa mj_1 merkin $mj_1[j\text{-}1]$ kohdalla ja $mj_1 = mj_2[0\ldots j\text{-}1] \cup mj_1^0[m_1^0\text{-}(m_1\text{-}(j))\ldots m_1^0\text{-}1] = mj_2[0\ldots j\text{-}1] \cup mj_1^0[i\ldots m_1^0\text{-}1]$.

Todistus:
Lause 2.2.4 seuraa aiemmin jo todetun lisäksi lauseesta 2.2.1, jonka avulla huomataan, että $mj_1[j\ldots m_1\text{-}1] \equiv mj_1^0[m_1^0\text{-}(m_1\text{-}(j))\ldots m_1^0\text{-}1]$ ja $mj_1[j] \equiv mj_1^0[i] \equiv mj_1^0[m_1^0\text{-}(m_1\text{-}(j))]$. Siis $i = m_1^0\text{-}(m_1\text{-}(j))$ ja $mj_1[j\ldots m1\text{-}1] \equiv mj_1^0[i\ldots m_1^0\text{-}1]$.

Tutkitaan nyt, miten kukin askeleista X.1 – X.4 vaikuttaa seuraavaksi tarkasteltavien merkkien indekseihin merkkijonoissa mj_1^0 ja mj_2^0, kun ennen askelta ollaan merkin $mj_1[j\text{-}1]$

kohdalla ja tarkasteltavat merkit ovat $mj_1^0[i]$ ja $mj_2^0[j] = mj_2[j]$. Lauseen 2.2.4 mukaan ennen tällöin tehtävää toimenpidettä pätee $mj_1 = mj_1[0\ldots m_1\text{-}1] = mj_2[0\ldots j\text{-}1] \cup mj_1^0[i\ldots m_1^0\text{-}1]$.

Kohdan X.1 askel (ei operaatiota):

Siirrytään merkkiin $mj_1[j]$ ja $mj_1 = mj_1'$. Seuraavaksi tarkastelun kohteena ovat merkit $mj_1[j+1] = mj_1'[j+1] = mj_1^0[i+1]$ ja $mj_2[j+1] = mj_2^0[j+1]$, joita vastaa taulukon alkio $(i+1, j+1)$.

Kohdan X.2 askel (merkin $mj_1[j] \equiv mj_1^0[i]$ poisto):

Pysytään merkin $mj_1[j\text{-}1]$ kohdalla ja $mj_1 = mj_1'[0\ldots j\text{-}1] \cup mj_1^0[i+1\ldots m_1 0\text{-}1]$. Seuraavaksi tarkastelun kohteena ovat merkit $mj_1[j] = mj_1'[j+1] = mj_1^0[i+1]$ ja $mj_2[j] = mj_2^0[j]$, joita vastaa taulukon alkio $(i+1, j)$.

Kohdan X.3 askel (merkin $mj_2[j]$ lisääminen merkin $mj_1[j\text{-}1]$ perään):

Siirrytään merkkiin $mj_1[j]$ ja $mj_1 = mj_1'[0\ldots j\text{-}1] \cup mj_2[j] \cup mj_1^0[i\ldots m_1^0\text{-}1]$. Seuraavaksi tarkastelun kohteena ovat merkit $mj_1[j+1] = mj_1'[j+1] = mj_1^0[i]$ ja $mj_2[j+1] = mj_2^0[j+1]$, joita vastaa taulukon alkio $(i, j+1)$.

Kohdan X.4 askel (merkin $mj_1[j]$ korvaaminen merkillä $mj_2[j]$):

Siirrytään merkkiin $mj_1[j]$ ja $mj_1 = mj_1'[0...j\text{-}1] \cup mj_2[j] \cup mj_1^0[i+1\ldots m_1^0\text{-}1]$. Seuraavaksi tarkastelun kohteena ovat merkit $mj_1[j+1] = mj_1'[j+1] = mj_1^0[i+1]$ ja $mj_2[j+1] = mj_2^0[j+1]$, joita vastaa taulukon alkio $(i+1, j+1)$.

Edellisten mukaan taulukon alkiosta (i, j) siirrytään seuraavasti, kun samalla alkioihin kirjataan askeleiden aiheuttaman työn kokonaismäärä:

Y.1 Tyhjän operaation tapauksessa siirrytään alkioon $(i+1, j+1)$. Koska toimenpide ei aiheuta työtä, annetaan alkiolle $(i+1, j+1)$ alkion (i, j) arvo.

Y.2 Merkin $mj_1^0[i]$ poiston tapauksessa siirrytään alkioon $(i+1, j)$. Tehdyn työn kirjaamiseksi alkio $(i+1, j)$ saa arvon $(i, j) + \omega(P)$.

Y.3 Lisättäessä merkki $mj_2^0[j]$ merkin $mj_1^0[j\text{-}1]$ perään siirrytään alkioon $(i, j+1)$. Tehdyn työn kirjaamiseksi alkio $(i, j+1)$ saa arvon $(i, j) + \omega(L)$.

Y.4 Korvattaessa merkki $mj_1{}^0[j]$ merkillä $mj_2{}^0[j]$ siirrytään alkioon $(i+1, j+1)$. Tehdyn työn kirjaamiseksi alkio $(i+1, j+1)$ saa arvon $(i, j) + \omega(K)$.

Määritelmä 2.2.5:

Kutsutaan sääntöjen X.1 – X.4 mukaista merkkijonon mj_1 läpikäyntiä vastaavaa, kohtien Y.1 – Y.4 mukaan muodostunutta taulukon alkioiden (-1, -1) ja $(m_1{}^0, m_2{}^0)$ välistä polkua merkkijonojen $mj_1{}^0$ ja $mj_2{}^0$ väliseksi editointipoluksi.

	j	-1	0	1	2	3	4	5	6
i		ε	’a’	’p’	’u’	’r’	’a’	’h’	’a’
-1	ε	0							
0	’a’		0						
1	’n’			1					
2	’t’				2				
3	’u’					3			
4	’r’						4		
5	’a’							5	6

	j	-1	0	1	2	3	4	5	6
i		ε	’a’	’p’	’u’	’r’	’a’	’h’	’a’
-1	ε	0							
0	’a’		0						
1	’n’			1					
2	’t’			2					
3	’u’				2				
4	’r’					2			
5	’a’						2	3	4

Vasemmanpuoleisen taulukon siirtymät ja vastaavat toimenpiteet ovat:

(-1, -1) → (0, 0): Koska $mj_1{}^0[0]$ = ’a’ = $mj_2{}^0[0]$, ei operaatiota tarvittu. Työ = 0.
(0, 0) → (1, 1): Merkki $mj_1{}^0[1]$ = ’n’ korvattiin merkillä $mj_2{}^0[1]$ = ’p’. Työ = 1.
(1, 1) → (2, 2): Merkki $mj_1{}^0[2]$ = ’t’ korvattiin merkillä $mj_2{}^0[2]$ = ’u’. Työ = 2.
(2, 2) → (3, 3): Merkki $mj_1{}^0[3]$ = ’u’ korvattiin merkillä $mj_2{}^0[3]$ = ’r’. Työ = 3
(3, 3) → (4, 4): Merkki $mj_1{}^0[4]$ = ’r’ korvattiin merkillä $mj_2{}^0[4]$ = ’a’. Työ = 4.
(4, 4) → (5, 5): Merkki $mj_1{}^0[5]$ = ’a’ korvattiin merkillä $mj_2{}^0[5]$ = ’h’. Työ = 5.
(5, 5) → (5, 6): Merkin $mj_1{}^0[5]$ perään lisättiin merkki $mj_2{}^0[6]$ = ’a’. Työ = 6.

Ja vastaavasti oikeanpuoleisessa taulukossa:

(-1, -1) → (0, 0): Koska $mj_1{}^0[0]$ = ’a’ = $mj_2{}^0[0]$, ei operaatiota tarvittu. Työ = 0.
(0, 0) → (1, 1): Merkki $mj_1{}^0[1]$ = ’n’ korvattiin merkillä $mj_2{}^0[1]$ = ’p’. Työ = 1.
(1, 1) → (2, 1): Poistettiin merkki $mj10[2]$ = ’t’. Työ = 2.
(2, 1) → (3, 2): Koska $mj_1{}^0[3]$ = ’u’ = $mj_2{}^0[2]$, ei operaatiota tarvittu. Työ = 2.
(3, 2) → (4, 3): Koska $mj_1{}^0[4]$ = ’r’ = $mj_2{}^0[3]$, ei operaatiota tarvittu. Työ = 2.
(4, 3) → (5, 4): Koska $mj_1{}^0[5]$ = ’a’ = $mj_2{}^0[4]$, ei operaatiota tarvittu. Työ = 2.
(5,4) → (5, 5): Merkin $mj_1{}^0[5]$ perään lisättiin merkki $mj_2{}^0[5]$ = ’h’. Työ = 3.
(5,5) → (5, 6): Merkin $mj_1{}^0[5]$ perään lisättiin merkki $mj_2{}^0[5]$ = ’a’. Työ = 4.

Esimerkki 2.2.6:

Kaksi erilaista määritelmän 2.2.5 mukaista merkkijonojen mj_1 = ”antura” ja mj_2 = ”apuraha” välistä editointipolkua, kun oletus $\omega(P) = \omega(L) = \omega(K) = 1$ on voimassa.

Seuraavan lauseen väite on ilmeinen askeleiden Y.1 - Y.4 perusteella, joten esitän sen ilman enempiä perusteluja:

Lause 2.2.7:
Editointipolulla olevan alkion (i, j) arvo kertoo aina kyseisen editointipolun osapolulla (-1, -1),…,(i, j) tehdyn työmäärän.

On hyvä tehdä seuraava huomio:

Lause 2.2.8:
Edellä esitelty taulukkotyyppi sopii jokaisen kohtien $X.1 - X.4$ mukaisen merkkijonon mj_1 läpikäynnin kuvaamiseen.

Todistus:
Lause 2.2.8 seuraa siitä, että kohdista $Y.1 - Y.4$ löytyy kohtia $X.1 - X.4$ vastaavat siirtymät taulukossa, ja toisaalta taulukon reunan yli ei voida mennä. Viimeksi mainittu johtuu siitä, että taulukon reuna-alkion kohdalla oltaessa on aina sellainen tilanne merkkijonojen mj_1^0 ja mj_2^0 suhteen, että sellainen toimenpide, joka astuisi reunan yli, ei ole mielekäs. Esimerkiksi taulukon oikeanpuoleisimman sarakkeen alkioiden kohdalla on merkkijonosta mj_2^0 seuraavaksi tarkastelun alla merkki $mj_2^0[m_2^0]$, jota ei ole olemassa. Näin ollen lisäys, korvaus tai tyhjä siirtymä eivät ole mielekkäitä operaatioita tässä tilanteessa.

Kuten esimerkissä 2.2.6 nähtiin, editointipolkuja on erilaisia ja eri määrän työtä tekeviä. Editointietäisyyttä ajatellen ollaan luonnollisesti kiinnostuneita sellaisista editointipoluista, jotka tekevät pienimmän mahdollisen työmäärän muuttaessaan merkkijonon mj_1 merkkijonoksi mj_2 eli joiden viimeinen alkio (m_1^0, m_2^0) sisältää arvon E(mj_1^0, mj_2^0). Kutsutaan tällaisia polkuja minimaalisiksi editointipoluiksi.

Määritelmä 2.2.9:
Määritelmän 2.2.5 mukainen merkkijonojen editointipolku on minimaalinen jos ja vain jos sen viimeinen alkio (mj_1^0, mj_2^0) sisältää arvon E(mj_1^0, mj_2^0).

Esimerkin 2.2.6 oikeanpuoleisen taulukon polku oli minimaalinen, sillä siinä alkion (mj_1^0, mj_2^0) = (5, 6) arvo vastasi arvoa E(mj_1^0, mj_2^0) = E(”antura”, ”apuraha”) = 4.

Lause 2.2.10:
Jokainen minimaalisella editointipolulla oleva alkio (i, j) sisältää aina arvon $E(mj_1^0[0\ldots i], mj_2^0[0\ldots j])$.

Todistus:
Tarkastellaan kyseisen editointipolun alkuosaa alkiosta (-1, -1) alkioon (i, j). Kun polun askeleita on seurattu alkioon (i, j) asti, tiedetään, että siinä vaiheessa tehty työmäärä on alkion (i, j) arvo ja seuraavaksi tarkasteltavat merkit ovat $mj_1^0[i+1]$ ja $mj_2^0[j+1]$. Lauseen 2.2.4 mukaan tällöin $mj_1 = mj_2[0\ldots j] \cup mj_1^0[i+1\ldots m_1^0-1]$. Koska $mj_2[0\ldots j] = mj_2^0[0..j]$ ja polun alussa alkion (-1, -1) kohdalla $mj_1^0 = mj_1^0[0\ldots m_1^0-1] = mj_1^0[0\ldots i] \cup mj_1^0[i+1\ldots m_1^0-1]$, voidaan todeta, että polulla siihen mennessä suoritetut operaatiot ovat muuttaneet merkkijonon $mj_1^0[0\ldots 1]$ merkkijonoksi $mj_2^0[0\ldots j]$. Olkoon alkion (i, j) arvo c, joka ei selvästi voi olla pienempi kuin $E(mj_1^0[0\ldots i], mj_2^0[0\ldots j])$. Vastaavasti alkiosta (i, j) eteenpäin kuljettaessa lopulta alkion (m_1^0-1, m_2^0-1) kohdalla $mj_1 = mj_2[0\ldots m_2^0-1] = mj_2^0$, joten alkioiden (i, j) ja (m_1^0-1, m_2^0-1) välisen polun askeleiden suorittamat editointioperaatiot muuttavat merkkijonon $mj_1^0[i+1\ldots m_1^0-1]$ merkkijonoksi $mj_2^0[j+1\ldots m_2^0-1]$. Olkoon tällä polun loppuosalla tehty kokonaistyömäärä d, jolloin alkion (m_1^0-1, m_2^0-1) arvo on $c + d$, eli polun minimaalisuudesta johtuen $E(mj_1^0, mj_2^0) = c + d$. Tehdään vastaoletus $c > E(mj_1^0[0\ldots i], mj_2^0[0\ldots j])$. Nyt on siis olemassa editointioperaatiojoukko, joka muuttaa merkkijonon $mj_1^0[0\ldots i]$ merkkijonoksi $mj_2^0[0\ldots j]$ tehden vähemmän kuin luvun c verran työtä. Toisaalta merkkijono $mj_1^0[i+1\ldots m_1^0-1]$ voidaan muuttaa merkkijonoksi $mj_2^0[j+1\ldots m_2^0-1]$ tekemällä luvun d verran työtä, ja näin ollen merkkijono mj_1^0 voidaan muuttaa merkkijonoksi mj_2^0 tekemällä yhteensä vähemmän kuin luvun $c + d$ verran työtä. Koska $c + d = E(mj_1^0, mj_2^0)$, on kyseessä ristiriita, sillä $E(mj_1^0, mj_2^0)$ kertoo pienimmän mahdollisen tällaisen työmäärän arvon. Siis minimaalisella editointipolulla olevan alkion (i, j) arvo on aina $E(mj_1^0[0\ldots i], mj_2^0[0\ldots j])$.

Yksi yleisimmistä editointietäisyyden tutkimisessa käytettyjen algoritmien perusperiaatteista on muodostaa jonkinlaisten dynaamisten sääntöjen mukaan vertailtaville merkkijonoille mj_1 ja mj_2 edellä esitettyjä taulukoita muistuttava ns. ”editointietäisyystaulukko”, jossa aina taulukon alkio (i, j) kertoo merkkijonojen $mj_1^0[0\ldots i]$ ja $mj_2^0[0\ldots j]$ välisen editointietäisyyden $E(mj_1^0[0\ldots i], mj_2^0[0\ldots j])$, kun $i < m_1^0$ ja $j < m_2^0 - 1$. Viitataan tästä lähtien tällaiseen editointietäisyystaulukkoon merkinnällä T_e ja kyseisen taulukon alkioon (i, j) merkinnällä $T_e(i, j)$, jolloin siis $T_e(i, j) = E(mj_1^0[0\ldots i], mj_2^0[0\ldots j])$ aina, kun $0 < i < m_1^0$ ja $0 < j < m_2^0$. Editointietäisyystaulukko voidaan näin ollen tulkita eräänlaiseksi kaikkien muotoa $mj_1^0[0\ldots i]$ ja $mj_2^0[0\ldots j]$ olevien merkkijonojen välisten minimaalisten suunnattujen

editointipolkujen esitystavaksi: Arvo $T_e(i, j)$ kertoo aina kustannukseltaan minimaalisen alkioiden $T_e(0, 0)$ ja $T_e(i, j)$ välillä olevan editointipolun kustannuksen, ja lauseesta 2.2.10 seuraa myös, että kaikki näillä minimaalisilla poluilla olevat alkiot ovat tällaisessa taulukossa merkittyinä. Edelliset yhdistämällä nähdään, että editointietäisyystaulukko sisältää kaikki merkkijonojen $mj_1{}^0[0\ldots i]$ ja $mj_2{}^0[0\ldots j]$ väliset minimaaliset editointipolut kokonaisuudessaan.

	j	-1	0	1	2	3	4	5	6
i		ε	’a’	’p’	’u’	’r’	’a’	’h’	’a’
-1	ε	0	1	2	3	4	5	6	7
0	’a’	1	0	1	2	3	4	5	6
1	’n’	2	1	1	2	3	4	5	6
2	’t’	3	2	2	2	3	4	5	6
3	’u’	4	3	3	2	3	4	5	6
4	’r’	5	4	4	3	2	3	4	5
5	’a’	6	5	5	4	3	2	3	4

Kuva 2.2.11:
Merkkijonojen $mj_1{}^0$ = ”antura” ja $mj_2{}^0$ = ”apuraha” välinen täytetty editointietäisyystaulukko T_e. Taulukosta nähdään esimerkiksi, että koska $T_e(4, 6) = 5$, on merkkijonojen ”antu” ja ”apurah” välinen editointietäisyys $E(mj_1[0\ldots3], mj_2[0\ldots5])$ yhtäsuuri kuin 5.

Jokainen alkioiden $T_e(0, 0)$ ja $T_e(i, j)$ välinen minimaalinen polku esittää yhden sellaisen editointioperaatiojonon, että sen operaatiot toteuttamalla merkkijono mj_1 saadaan identtiseksi merkkijonon mj_2 kanssa tekemällä arvon $T_e(i, j) = E(mj_1[0\ldots i], mj_2[0\ldots j])$ verran työtä. Tällainen minimaalinen polku voidaan etsiä lähtemällä peruuttamaan alkiosta $T_e(i, j)$ siirtyen askel kerrallaan aina joko vasemmalle alkioon $T_e(i, j\text{-}1)$, ylöspäin alkioon $T_e(i\text{-}1, j)$ tai vasempaan yläviistoon alkioon $T_e(i\text{-}1, j\text{-}1)$, sillä kohtien Y.1 – Y.4 mukaisella askeleella voidaan alkioon $T_e(i, j)$ saapua vain jostain näistä kolmesta vaihtoehdosta. Peruutusaskel pitää luonnollisesti valita siten, että se on laillinen, eli jonkin kohtien Y.1 – Y.4 siirtymistä pitää vastata alkioiden arvoja. Käytännössä tämä tarkoittaa, että tutkitaan mikä seuraavista ehdoista on voimassa:

i) $T_e(i\text{-}1, j\text{-}1) = T_e(i, j)$, eli tyhjä operaatio (täsmäys).
ii) $T_e(i, j\text{-}1) = T_e(i, j) - \omega(\mathrm{L})$, eli lisäysoperaatio.
iii) $T_e(i\text{-}1, j) = T_e(i, j) - \omega(\mathrm{P})$, eli poisto-operaatio.
iv) $T_e(i\text{-}1, j\text{-}1) = T_e(i, j) - \omega(\mathrm{K})$, eli korvausoperaatio.

Peruutusaskel ei määräydy aina välttämättä yksikäsitteisesti edellisistä kohdista, sillä useampi kuin yksi edellisistä ehdoista voi päteä. Näissä tapauksissa vaihtoehtoisista pienimpään arvoon siirtyvistä peruutusaskeleista voidaan valita mikä tahansa, jolloin tuloksena on tällöin jokin useista mahdollisista minimaalisista poluista. Koska kuitenkin tiedetään, että koko minimaalinen polku on taulukossa, on aina vähintään yhden edellisistä ehdoista oltava voimassa, ja peruutusaskeleita voidaan jatkaa, kunnes saavutaan alkioon $T_e(-1, -1)$. Tällaisen peruutuksen yhteydessä läpikäydyt alkiot muodostavat jonkin merkkijonojen $mj_1^0[0\ldots i]$ ja $mj_2^0[0\ldots j]$ välisen minimaalisen editointipolun, mutta vain päinvastaisessa järjestyksessä.

Kuvan 2.2.12 editointietäisyystaulukko demonstroi edellä käsiteltyjä minimaalisten polkujen ominaisuuksia käyttäen jälleen esimerkkinä merkkijonoja mj_1^0 = ”antura” ja mj_2^0 = ”apuraha”.

	j	-1	0	1	2	3	4	5	6
i		ε	’a’	’p’	’u’	’r’	’a’	’h’	’a’
-1	ε	0	1	2	3	4	5	6	7
0	’a’	1	0	1	2	3	4	5	6
1	’n’	2	1	1	2	3	4	5	6
2	’t’	3	2	**2**	2	3	4	5	6
3	’u’	4	3	3	**2**	3	4	5	6
4	’r’	5	4	4	3	2	3	4	5
5	’a’	6	5	5	4	3	2	3	4

Kuva 2.2.12:
Merkkijonojen mj_1^0 = ”antura” ja mj_2^0 = ”apuraha” välinen editointietäisyystaulukko. Taulukossa on sellaisten alkioiden tausta väritetty, jotka kuuluvat johonkin alkioiden $T_e(0, 0)$ ja $T_e(5, 6)$ väliseen minimaaliseen polkuun. Alkioiden $T_e(2, 1)$ ja $T_e(3, 2)$ arvot on lisäksi lihavoitu merkiksi siitä, että näistä alkioista on enemmän kuin yksi peruutusvaihtoehto.

Kuvan 2.2.12 taulukossa yksi mahdollinen peruutussekvenssi käy läpi alkiot (5, 6), (5, 5), (5, 4), (4, 3), (3, 2), (2, 1), (1, 0) ja (0, 0), jolloin tätä vastaava minimaalinen polku on (0,0), (1, 0), (2, 1), (3, 2), (4, 3), (5, 4) (5, 5) , (5, 6). Kyseistä minimaalista polkua vastaavat editointioperaatiot saadaan seuraavasti käymällä järjestyksessä läpi polun askeleet:

(0, 0) → (1, 0): Siirtymä alaspäin, eli vastaava operaatio on P(mj_1, 1) = P(”antura”, 1). Operaation jälkeen mj_1 = ”atura”.

(1, 0) → (2, 1): Siirtymä oikealle alaviistoon ja $T_e(2, 1) = T_e(1, 0) + 1$, eli kyseessä on korvaus. Vastaava operaatio on K(mj_1, 1, mj_2[1]) = K(”atura”, 1, ’p’). Tämän jälkeen mj_1 = ”apura”.

(2, 1) → (3, 2): Siirtymä oikealle alaviistoon ja $T_e(3, 2) = T_e(2, 1)$. Kyseessä on siis merkkien mj_1[2] = ’u’ ja mj_2[2] = ’u’ täsmääminen. Mikään ei siis muutu, vaan yhä mj_1 = ”apura”.

(3, 2) → (4, 3): Siirtymä oikealle alaviistoon ja $T_e(4, 3) = T_e(3, 2)$. Kyseessä on siis merkkien mj_1[3] = ’r’ ja mj_2[3] = ’r’ täsmääminen. Mikään ei siis muutu, vaan yhä mj_1 = ”apura”.

(4, 3) → (5, 4): Siirtymä oikealle alaviistoon ja $T_e(5, 4) = T_e(4, 3)$. Kyseessä on siis merkkien mj_1[4] = ’a’ ja mj_2[4] = ’a’ täsmääminen. Mikään ei siis muutu, vaan yhä mj_1 = ”apura”.

(5, 4) → (5, 5): Siirtymä oikealle, siis kyseessä on lisäys. Vastaava operaatio on L(mj_1, 5, mj_2[5]) = L(”apura”, 5, ’h’), jonka jälkeen mj_1 = ”apurah”.

(5, 5) → (5, 6): Siirtymä oikealle, siis kyseessä on lisäys. Vastaava operaatio on L(mj_1, 6, mj_2[6]) = L(”apurah”, 6, ’a’), jonka jälkeen mj_1 = ”apuraha”. Saavuttiin viimeiseen alkioon ja operaatioiden ansiosta merkkijono mj_1 = ”apuraha” = mj_2.

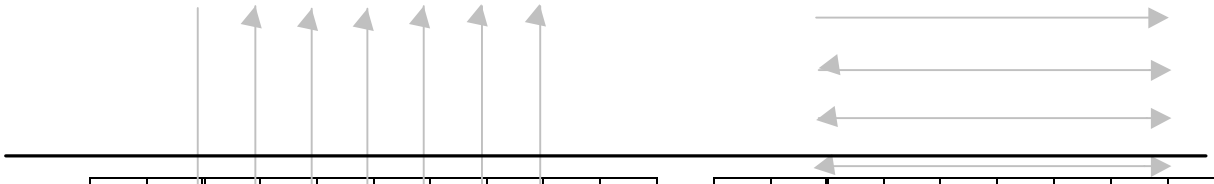
2.2.1 Perusdynaaminen ratkaisu

Editointietäisyys soveltuu hyvin perinteisellä dynaamisella lähestymistavalla laskettavaksi, ja tämä lieneekin syynä siihen, että perus-dynaaminen algoritmi on keksitty moneen eri kertaan eri ihmisten toimesta (esim. [Navarro, 1998]). Algoritmi laskee merkkijonojen mj_1 ja mj_2 yhä pidempien ja pidempien alkuosien välisen editointietäisyyden, kunnes lopulta päädytään haluttuun koko merkkijonojen väliseen editointietäisyyteen. Dynaamisen laskennan periaatteen mukaisesti pidempien alkuosien väliset editointietäisyydet pyritään laskemaan tehokkaasti jo laskettujen lyhyempien alkuosien välisten etäisyyksien pohjalta. Edellisessä kappaleessa esitettyjen editointipolun määrittelevien askeleiden *Y*.1 - *Y*.4 nojalla nähdään, että arvo $T_e(i, j)$ voi olla joko $T_e(i\text{-}1, j) + \omega(\text{P})$, $T_e(i, j\text{-}1) + \omega(\text{L})$, $T_e(i\text{-}1, j\text{-}1) + \omega(\text{K})$, tai $T_e(i\text{-}1, j\text{-}1)$. Nämä vaihtoehdot vastaavat järjestyksessä poisto-, lisäys-, korvaus- ja täsmäysaskelta. Edellinen on helppo todeta oikeaksi, sillä myös minimaalinen editointipolku on editointipolku, eli se ei voi sisältää muita kuin edellisen kaltaisia askeleita, ja siten on alkiosta $T_e(0, 0)$ alkioon $T_e(i,j)$ saapuvan suunnatun editointipolun viimeisen askeleen pakko olla jokin edellämainituista vaihtoehdoista. Koska kyseessä on minimaalinen polku, valitaan luonnollisesti näistä pienimmän arvon alkiolle $T_e(i, j)$ antava askel. Tästä saadaan varsin yksinkertainen laskusääntö alkiolle $T_e(i, j)$:

Lause 2.2.13:
$T_e(i, j) = \min\{T_e(i\text{-}1, j) + \omega(\text{P}), T_e(i, j\text{-}1) + \omega(\text{L}), T_e(i\text{-}1, j\text{-}1) + (mj_1[i] \mathrel{!}= mj_2[j]) \times \omega(\text{K})\}$.

Tässä oletetaan, että erisuuruusoperaattori "!=" toimii kuten *C*-kielessä eli että merkintä $(mj_1[i] \mathrel{!}= mj_2[j])$ saa arvon 1, jos $mj_1[i] \neq mj_2[j]$, ja arvon 0, jos $mj_1[i] = mj_2[j]$. Minimilausekkeen viimeisin vaihtoehto $T_e(i\text{-}1, j\text{-}1) + (mj_1[i] \mathrel{!}= mj_2[j]) \times \omega(\text{K})$ kattaa näin ollen sekä täsmäys- että korvausaskeleen, sillä edellisessä tapauksessa se saa arvon $T_e(i\text{-}1, j\text{-}1) + 0 \times \omega(\text{K}) = T_e(i\text{-}1, j\text{-}1)$ ja jälkimmäisessä arvon $T_e(i\text{-}1, j\text{-}1) + 1 \times \omega(\text{K}) = T_e(i\text{-}1, j\text{-}1) + \omega(\text{K})$.

Editointietäisyystaulukon täyttäminen aloitetaan arvosta $T_e(-1, -1)$, joka on helppo laskea. Selvästi $T_e(-1, -1) = E(mj_1[-1], mj_2[-1]) = 0$, sillä $mj_1[-1] = mj_2[-1] = \varepsilon$. Vastaavasti voidaan laskea suoraan kaikki muotoa $T_e(-1, j)$ ja $T_e(i, -1)$ olevat alkiot, sillä kaikki alkiosta $T_e(-1, -1)$ näihin alkioihin menevät minimaaliset polut koostuvat kokonaan joko lisäys- tai poistoaskeleista, ja näin ollen aina $T_e(i, -1) = i \times \omega(\text{P})$, ja $T_e(-1, j) = j \times \omega(\text{L})$. Tämän jälkeen loput arvot voidaankin jo laskea edellä esitettyä laskusääntöä käyttäen joko riveittäin tai sarakkeittain kuvan 2.2.14 mukaisesti.



	j	-1	0	1	2	3	4	5	6
i		ε	’a’	’p’	’u’	’r’	’a’	’h’	’a’
-1	ε	0	1	2	3	4	5	6	7
0	’a’	1							
1	’n’	2							
2	’t’	3							
3	’u’	4							
4	’r’	5							
5	’a’	6							

	j	-1	0	1	2	3	4	5	6
i		ε	’a’	’p’	’u’	’r’	’a’	’h’	’a’
-1	ε	0	1	2	3	4	5	6	7
0	’a’	1							
1	’n’	2							
2	’t’	3							
3	’u’	4							
4	’r’	5							
5	’a’	6							

Kuva 2.2.14:
Kaksi mahdollista editointietäisyystaulukon alkioiden laskujärjestystä.

Kummassakin kuvan 2.2.14 esittämässä alkioiden arvojen laskujärjestyksessä on aina alkiota $T_e(i, j)$ laskettaessa laskusäännön tarvitsemat alkiot $T_e(i\text{-}1, j)$, $T_e(i, j\text{-}1)$ ja $T_e(i\text{-}1, j\text{-}1)$ jo laskettu aikaisemmin, joten alkion $T_e(i, j)$ arvo saadaan helposti laskettua. Algoritmi 2.2.15 toteuttaa editointietäisyystaulukon täyttämisen käyttäen sarakkeittain etenevää laskujärjestystä (kuvan 2.2.14 vasemmanpuoleinen taulukko).

Algoritmin 2.2.15 aikavaativuudeksi nähdään sisäkkäisten ja rakenteeltaan yksinkertaisten while-silmukoiden pohjalta suoraviivaisesti $O(m_1 \times m_2)$. Algoritmi on siinä mielessä optimaalinen, että koska se täyttää kokoa $m_1 \times m_2$ olevan taulukon, täytyy sen myös tehdä vähintään $m_1 \times m_2$ operaatiota.

```
int i;                                          // Merkkijonoa mj1 läpikäyvä indeksi.
int j;                                          // Merkkijonoa mj2 läpikäyvä indeksi.
int m1;                                         // Merkkijonon mj1 pituus.
int m2;                                         // Merkkijonon mj2 pituus.
int wl, wp, wk;                         // Lisäys-, poisto- ja korvausoperaatioiden kustannukset.
int lisays, poisto, korvaus;            // Laskusääntöä sovellettaessa käytettävät muuttujat.
int taulukko[m1 + 1][m2 + 1];                   // Editointietäisyystaulukko.
int apuri;                                      // Apumuuttuja.

for(apuri = 0; apuri < m1 + 1; apuri++)         // Alustetaan muotoa Te(i, -1) olevat alkiot.
{
    taulukko[apuri][0] = apuri*wp;
}
for(apuri = 0; apuri < m2 + 1; apuri++)         // Alustetaan muotoa Te(-1, j) olevat alkiot.
{
    taulukko[0][apuri] = apuri*wl;
}
j = 1;                          // Alustetaan indeksi j ensimmäisenä täytettävän sarakkeen kohdalle.
while(j < m2 + 1)                       // Edetään sarakkeittain, siis ulompi silmukkamuuttuja on j.
{
    i = 1;                      // Indeksi i alustetaan arvolla 1, eli sarakkeen ylimpään tyhjään alkioon.
    while(i < m1 + 1)                   // Edetään sarakkeen alimpaan riviin asti.
    {
        lisays = taulukko[i][j-1] + wl;         // Lasketaan eri vaihtoehdot laskusäännön
        poisto = taulukko[i-1][j] + wp;         // soveltamista varten.
        korvaus = taulukko[i-1][j-1] + (mj1[i-1] != mj2[j-1])*wk;
        apuri = korvaus;                        // Muuttujaan ”apuri” lasketaan edellisten minimi.
        if(apuri > poisto)                      // Onko poisto pienempi kuin korvaus?
        {
            apuri = poisto;
        }
        if(apuri > lisays)              // Onko lisäys edullisempi kuin min{korvaus, poisto}?
        {
            apuri = lisays;
        }
        taulukko[i][j] = apuri;                 // Taulukko[i][j] saa edellä lasketun minimiarvon.
        i++;
    }
    j++;
}
```

Algoritmi 2.2.15:
Perusdynaaminen kahden merkkijonon välisen editointietäisyystaulukon muodostava algoritmi.

2.2.2 Ukkosen parannettu dynaaminen algoritmi

Esko Ukkonen on esittänyt yhden tavan tehostaa kahden merkkijonon välisen editointietäisyyden laskemista [Ukkonen, 1985a]. Algoritmi on useissa tapauksissa melko tehokas [Stephen, 1994] mutta idealtaan kuitenkin hyvin yksinkertainen. Hänen lähestymistapansa eroaa perusdynaamisesta oleellisesti vain siinä, että algoritmi pyrkii laskemaan arvot vain osalle editointietäisyystaulukon alkioista. Nimittäin, kuten perusdynaamisen algoritmin yhteydessä todettiin, toimii taulukkoon laskettavien alkioiden arvojen lukumäärä alarajana algoritmin aikavaativuudelle.
Tarkastellaan laskettavien alkioiden määrän rajoittamiseksi, minkälaisia alkioita alkioiden (-1, -1) ja (m_1^0-1, m_2^0-1) välisellä minimaalisella polulla voi olla. Tässä keskitytään taulukon T_e diagonaaleihin, joten tutkitaan ensin hieman niitä. Ensin on hyvä aloittaa määrittelemällä diagonaalin käsite.

Määritelmä 2.2.16:
Käytetään nimitystä *k*-diagonaali sellaisesta maksimaalisesta taulukon T_e alkioiden (*i*,*j*) joukosta, että jokaiselle joukon alkiolle pätee ehto $j - i = k$.

---

	j	-1	0	1	2	3	4
i		ε	λ_5	λ_6	λ_7	λ_8	λ_9
-1	ε						
0	λ_1						
1	λ_2						
2	λ_3						
3	λ_4						

Kuva 2.2.17:
Kuvan taulukossa 0-diagonaali on musta, -1-diagonaali on raidallinen, ja 2-diagonaali on harmaa.

Lause 2.2.18:
Editointipolku, joka kulkee *p*-diagonaalilta *q*-diagonaalille, sisältää vähintään $|q - p|$ kappaletta lisäys- tai poistoaskeleita.

Todistus:
Lauseen väite seuraa suoraviivaisesti siitä, että ainoat arvoon $j - i$ vaikuttavat siirtymät vastaavat lisäys- tai poisto-operaatiota ja kumpikin näistä joko kasvattaa tai vähentää kyseisen erotuksen arvoa täsmälleen yhdellä. Koska polun alkuvaiheessa ollaan sellaisen alkion (*i*, *j*) kohdalla, että $j - i = p$, ja loppuvaiheessa sellaisen alkion (*i*, *j*) kohdalla, että $j - i = q$, on ilmeistä, että näiden tilanteiden välillä erotuksen $j - i$ arvo on muuttunut luvun $|q - p|$ verran, ja tämä vaatii vähintään $|q - p|$ kappaletta lisäys- tai poistoaskeleita.

Olkoon (*i*, *j*) jokin alkioiden (-1, -1) ja (m_1^0-1, m_2^0-1) välisen minimaalisen polun alkio. Koska alkio (-1, -1) sijaitsee taulukossa 0-diagonaalilla, ja vastaavasti alkio (*i*, *j*) sijaitsee (*j*-*i*)-diagonaalilla, tiedetään lauseen 2.2.18 perusteella, että näiden alkioiden välinen polku sisältää vähintään $|(j - i) - 0| = |j - i|$ kappaletta lisäys- tai poisto-operaatiota. Vastaavasti voidaan todeta, että polku alkiosta (*i*, *j*) alkioon (m_1^0-1, m_2^0-1) sisältää vähintään $|(m_2^0 - 1 - (m_1^0 - 1)) - (j - i)| = |m_2^0 - m_1^0 - j + i|$ kappaletta lisäyksiä tai poistoja. Yhteensä operaatioiden minimilukumääräksi tulee siis $|m_2^0 - m_1^0 - j + i| + |j - i|$. Tästä saadaan alkion (*i*, *j*) kautta kulkevan alkioiden (-1, -1) ja (m_2^0-1, m_1^0-1) välisen polun operaatioiden tekemän työn yhteismäärälle alarajaksi $(|m_2^0 - m_1^0 - j + i| + |j - i|) \times \min\{\omega(P), \omega(L)\}$. Oletetaan hetkeksi, että merkkijonojen mj_1^0 ja mj_2^0 välinen editointietäisyys $E(mj_1^0, mj_2^0)$ tunnetaan entuudestaan. Tällöin voidaan todeta edellisen polun työmäärän alaraja-arvion perusteella, että jos $E(mj_1^0, mj_2^0) < (|m_2^0 - m_1^0 - j + i| + |j - i|) \times \min\{\omega(P), \omega(L)\}$, ei tämä alkion (*i*, *j*) sisältävä polku voi olla minimaalinen polku, ja siten alkion (*i*, *j*) arvoa ei tarvitse laskea. Tästä saadaan seuraava lause:

Lause 2.2.19:
Editointietäisyystaulukossa riittää laskea arvot vain sellaisille alkioille (*i*, *j*), joiden indekseillä *i* ja *j* pätee $(|m_2^0 - m_1^0 - j + i| + |j - i|) \times \min\{\omega(P), \omega(L)\} \leq E(mj_1^0, mj_2^0)$.

Itseisarvojen takia tämän ehdon indekseille *i* ja *j* asettamien rajojen selvittäminen vaatii seuraavien neljän erillisen tapauksen tutkimisen:

1) $m_2^0 - m_1^0 - j + i \geq 0$ ja $j - i \geq 0$: Nyt $0 \leq |m_2^0 - m_1^0 - j + i| + |j - i| = m_2^0 - m_1^0 - j + i + j - i = m_2^0 - m_1^0$. Nyt lauseen 2.2.19 ehto on muotoa $(m_2^0 - m_1^0) \times \min\{\omega(P), \omega(L)\} \leq E(mj_1^0, mj_2^0)$, joka pätee aina, sillä merkkijonojen mj_1^0 ja mj_2^0 identtisiksi, ja samalla tietysti samanpituisiksi, editoiminen vie vähintään niiden pituuksien eron verran lisäys- tai poisto-operaatioita. Siis alkuehdot yhdistämällä todetaan, että ehdot

$$m_2^0 - m_1^0 \geq j - i \geq 0$$

täyttävät alkiot (i, j) kuuluvat laskettavien alkioiden joukkoon.

2) $m_2^0 - m_1^0 - j + i \geq 0$ ja $j - i < 0$: Nyt $0 \leq |m_2^0 - m_1^0 - j + i| + |j - i| = m_2^0 - m_1^0 - j + i + i - j = m_2^0 - m_1^0 + 2\times(i - j)$. Lauseen 2.2.19 ehto on tässä tapauksessa $(m_2^0 - m_1^0 + 2\times(i - j)) \times \min\{\omega(P), \omega(L)\} \leq E(mj_1^0, mj_2^0)$, mistä nähdään, että myös kaikki ehdot

$$j - i < 0 \text{ ja } m_2^0 - m_1^0 \geq j - i \geq \frac{m_2^0 - m_1^0}{2} - \frac{E(mj_1^0, mj_2^0)}{2 \times \min\{w(P), w(L)\}}$$

täyttävät alkiot (i, j) kuuluvat laskettavien alkioiden joukkoon.

3) $m_2^0 - m_1^0 - j + i < 0$ ja $j - i \geq 0$: Nyt $0 \leq |m_2^0 - m_1^0 - j + i| + |j - i| = m_1^0 - m_2^0 + j - i + j - i = m_1^0 - m_2^0 + 2\times(j - i)$. Lauseen 2.2.19 ehto on tässä tapauksessa $(m_1^0 - m_2^0 + 2\times(j - i)) \times \min\{\omega(P), \omega(L)\} \leq E(mj_1^0, mj_2^0)$, mistä nähdään, että myös kaikki ehdot

$$0 \leq j - i \text{ ja } m_2^0 - m_1^0 < j - i \leq \frac{E(mj_1^0, mj_2^0)}{2 \times \min\{w(P), w(L)\}} + \frac{m_2^0 - m_1^0}{2}$$

täyttävät alkiot (i, j) kuuluvat laskettavien alkioiden joukkoon.

4) $m_2^0 - m_1^0 - j + i < 0$ ja $j - i < 0$: Nyt $0 \leq |m_2^0 - m_1^0 - j + i| + |j - i| = m_1^0 - m_2^0$. Nyt lauseen 2.2.19 ehto on muotoa $(m_1^0 - m_2^0) \times \min\{\omega(P), \omega(L)\} \leq E(mj_1^0, mj_2^0)$, joka todetaan samoin perusteluin kuin kohdassa 1) aina pätevälsi. Yhdistämällä alkuehdot todetaan siis myös, että kaikki ehdot

$$m_2^0 - m_1^0 < j - i < 0$$

täyttävät alkiot (i, j) kuuluvat laskettavien alkioiden joukkoon.

Kohdat 1) – 4) yhdistämällä saadaan siten laskettavien alkioiden (i, j) indekseille seuraavanlainen sääntö:

Seuraus 2.2.20:

Lauseen 2.2.19 asettamat ehdot rajaavat indeksien *i* ja *j* arvot seuraavalle välille:

$$\frac{m_2^{\,0} - m_1^{\,0}}{2} - \frac{E(mj_1^{\,0}, mj_2^{\,0})}{2 \times \min\{w(P), w(L)\}} \leq j - i \leq \frac{E(mj_1^{\,0}, mj_2^{\,0})}{2 \times \min\{w(P), w(L)\}} + \frac{m_2^{\,0} - m_1^{\,0}}{2}.$$

Koska alkio (*i*, *j*) sijaitsee (*j*-*i*)-diagonaalilla, voidaan seuraus 2.2.20 vielä pukea seuraavaan muotoon:

Lause 2.2.21:

Lauseen 2.2.18 asettamien ehtojen mukaan editointietäisyystaulukossa riittää laskea vain niiden alkioiden arvot, jotka kuuluvat sellaisille taulukon *k*-diagonaaleille, että

$$\frac{m_2^{\,0} - m_1^{\,0}}{2} - \frac{E(mj_1^{\,0}, mj_2^{\,0})}{2 \times \min\{w(P), w(L)\}} \leq k \leq \frac{E(mj_1^{\,0}, mj_2^{\,0})}{2 \times \min\{w(P), w(L)\}} + \frac{m_2^{\,0} - m_1^{\,0}}{2}.$$

---

	j	-1	0	1	2	3	4	5	6
i		ε	’a’	’p’	’u’	’r’	’a’	’h’	’a’
-1	ε	0	1	2					
0	’a’	1	0	1	2				
1	’n’		1	1	2	3			
2	’t’			2	2	3	4		
3	’u’				2	3	4	5	
4	’r’					2	3	4	5
5	’a’						2	3	4

Esimerkki 2.2.22:

Käytetään jälleen merkkijonoja $mj_1^{\,0}$ = ”antura” ja $mj_2^{\,0}$ = ”apuraha”, ja oletetaan, että ω(P) = ω(L) = ω(K) = 1. Koska E(”antura”, ”apuraha”) = 4, $m_1^{\,0} = 6$ ja $m_2^{\,0} = 7$, riittää lauseen 2.2.21 mukaan täyttää taulukko vain sellaisten *k*-diagonaalien osalta, missä $(7 - 6)/2 - 4/2 \leq k \leq 4/2 + (7 - 6)/2$ eli $-1{,}5 \leq k \leq 2{,}5$. Koska *k* on kokonaisluku, vastaa tämä diagonaaleja –1, 0, 1 ja 2. Yllä olevassa taulukossa nämä diagonaalit on tummennettu ja vain niiden arvot on laskettu.

Tarpeellisten diagonaalien arvot voidaan laskea melkein samaan tapaan kuin perusdynaamisen ratkaisun yhteydessä eli täyttämällä ensin yksinkertaiset muotoa (*i*, -1) ja (-1, *j*) olevat alkiot ja jatkamalla sitten dynaamisesti. Siinä käytettyä lauseen 2.2.13 laskusääntöä sovellettaessa pitää kuitenkin huolehtia siitä, että laskettavan alueen ulkopuoliset

alkiot jätetään huomioimatta. Tämä onnistuu periaatteessa esimerkiksi asettamalla sääntö $T_e(i, j) = \infty$ kaikille sellaisille alkioille (i, j), jotka eivät sijaitse laskettavilla diagonaaleilla, jolloin lauseen 2.2.13 laskusäännön minimilausekkeessa ei voida valita alueen ulkopuolelta tulevaa askelta vastaavaa vaihtoehtoa.

Kuvan 2.2.23 taulukot havainnollistavat tarpeellisten diagonaalien täyttämistä sekä sarakkeittain että riveittäin.

	j	-1	0	1	2	3	4	5	6
i		ε	’a’	’p’	’u’	’r’	’a’	’h’	’a’
-1	ε	0	1	2	∞	∞	∞	∞	∞
0	’a’	1				∞	∞	∞	∞
1	’n’	∞					∞	∞	∞
2	’t’	∞	∞					∞	∞
3	’u’	∞	∞	∞					∞
4	’r’	∞	∞	∞	∞				
5	’a’	∞	∞	∞	∞	∞			

	j	-1	0	1	2	3	4	5	6
i		ε	’a’	’p’	’u’	’r’	’a’	’h’	’a’
-1	ε	0	1	2	∞	∞	∞	∞	∞
0	’a’	1				∞	∞	∞	∞
1	’n’	∞					∞	∞	∞
2	’t’	∞	∞					∞	∞
3	’u’	∞	∞	∞					∞
4	’r’	∞	∞	∞	∞				
5	’a’	∞	∞	∞	∞	∞			

Kuva 2.2.23:
Kaksi erilaista mahdollista editointietäisyystaulukon alkioiden laskujärjestystä, kun käytetään Ukkosen algoritmia. Täytettävien diagonaalien alkiot on tummennettu, ja muiden alkioiden kohdalla on äärettömän symboli vastaten ajatusta, että näiden alkioiden arvoja ei oteta huomioon määritettäessä minimiarvoa laskettavalle alkiolle.

Nyt ongelmana on vielä se, että arvoa $E(mj_1^0, mj_2^0)$ ei tietenkään voida tietää etukäteen. Ukkosen algoritmissa tämä ongelma ratkaistaan käyttämällä lukua t luvun $E(mj_1^0, mj_2^0)$ arvioimiseen. Aiemman perusteella tiedetään, että $|m_2^0 - m_1^0| \times \min\{\omega(P), \omega(L)\} \leq E(mj_1^0, mj_2^0)$, joten on sopivaa aloittaa arviointi asettamalla esimerkiksi $t = (|m_2^0 - m_1^0| + 1) \times \min\{\omega(P), \omega(L)\}$. Tämän jälkeen algoritmi laskee arvot lauseen 2.2.21 mukaisille diagonaaleille olettaen, että $E(mj_1^0, mj_2^0) = t$. Jos arvio oli liian pieni eli $T_e(m_1^0, m_2^0) > t$, kerrotaan luku t kahdella ja täytetään taulukko kokonaan uudelleen lauseen 2.2.20 mukaisien diagonaalien osalta. Taulukosta täytetään siis vähitellen suurempi ja suurempi osa siten, että aina vanhat arvot pyyhitään yli, sillä uusien laskettavien diagonaalien arvot voivat vaikuttaa myös jo ennestään laskettujen diagonaalien alkioihin. Edellisiä vaiheita jatketaan, kunnes $T_e(m_1^0, m_2^0) \leq t$, sillä tällöin tiedetään, että nyt täytetyn taulukon alueen ulkopuoliset alkiot eivät enää voi pienentää alkion $T_e(m_1^0, m_2^0)$ arvoa, ja siten $T_e(m_1^0, m_2^0) = E(mj_1^0, mj_2^0)$. Tällä

tavalla toimittaessa tehdään pahimmassa tapauksessa enemmän työtä kuin suoraan koko taulukko täyttämällä, sillä taulukon monen alkion arvo voidaan mahdollisesti joutua laskemaan moneen kertaan. Merkkijonojen välisen editointietäisyyden ollessa pieni on algoritmi kuitenkin tehokkaampi kuin perus-dynaaminen ratkaisu. Lisäksi usein täydellisen editointietäisyyden laskemisen sijaan halutaan vain tietää, ovatko merkkijonot mj_1^0 ja mj_2^0 likimäärin samanlaisia eli onko niiden editointietäisyys korkeintaan ennaltamäärätyn suurimman sallitun virheen s_e suuruinen. Tässä tilanteessa Ukkosen algoritmi on vahvimmillaan, sillä tällöin tarvitaan vain yksi täyttökerta, jossa aseteteaan heti editointietäisyyden arvioksi $t = s_e$.

Algoritmi 2.2.24 esittää Ukkosen algoritmin algoritmin 2.2.15 pohjalta muotoillun toteutuksen valittaessa jälleen sarakkeittainen laskujärjestys (kuvan 2.2.23 vasemmanpuoleinen taulukko).

```
int i;                                          // Merkkijonoa mj1 läpikäyvä indeksi.
int j;                                          // Merkkijonoa mj2 läpikäyvä indeksi.
int m1;                                         // Merkkijonon mj1 pituus.
int m2;                                         // Merkkijonon mj2 pituus.
int wl, wp, wk;                          // Lisäys-, poisto- ja korvausoperaatioiden kustannukset.
int min_wp_wl;                    // Muuttuja jonka arvo on min{wp, wl}, oletetaan jo lasketuksi.
int lisays, poisto, korvaus;             // Laskusääntöä sovellettaessa käytettävät muuttujat.
int taulukko[m1 + 1][m2 + 1];                   // Editointietäisyystaulukko.
int apuri;                                      // Apumuuttuja.
int mindiag;                                    // Pienin täytettävä diagonaali.
int maxdiag;                                    // Suurin täytettävä diagonaali.
int t;                        // Merkkijonojen mj1 ja mj2 välisen editointietäisyyden ylärajan arvaus.

t = abs(m1 – m2) + 1;                        // Alustetaan etäisyysarvaus luvulla |m1 – m2| + 1.

for(apuri = 0; apuri < m1 + 1; apuri++)         // Alustetaan muotoa Te(i, -1) olevat alkiot.
{
    taulukko[apuri][0] = apuri*wp;
}
for(apuri = 0; apuri < m2 + 1; apuri++)         // Alustetaan muotoa Te(-1, j) olevat alkiot.
{
    taulukko[0][apuri] = apuri*wl;
}
do                                           // Arvoa t kasvattavan do-while-silmukan alku.
{
    j = 1;                          // Alustetaan indeksi j ensimmäisenä täytettävän sarakkeen kohdalle.
    while(j < m2 + 1)                  // Edetään sarakkeittain, siis ulompi silmukkamuuttuja on j.
    {
        mindiag = (m2 - m1 - t / min_wp_wl) / 2;    // Sovelletaan lausetta 2.2.21
        maxdiag = (m2 - m1 + t / min_wp_wl) / 2;    // diagonaalirajojen laskemiseksi.
        i = 1;                          // Indeksi i alustetaan ensin arvolla 1, eli sarakkeen ylimmän
```

Algoritmi 2.2.24 (jatkuu seuraavalla sivulla):
Ukkosen editointietäisyysalgoritmi.

```
        if(j-i > maxdiag)                    //  alkion kohdalle, ja siirretään sitten tarvittaessa
        {                                    //  maxdiag-diagonaalille.
            i = j -maxdiag;
        }
        while(i <= j-mindiag)                              //  Edetään mindiag-diagonaaliin asti.
        {
            lisays = taulukko[i][j-1] + wl;                //  Lasketaan eri vaihtoehdot laskusäännön
            poisto = taulukko[i-1][j] + wp;                //  soveltamista varten.
            korvaus = taulukko[i-1][j-1] + (mj1[i-1] != mj2[j-1])*wk;
            apuri = korvaus;                         //  Muuttujaan ”apuri” lasketaan edellisten minimi.
            if(apuri > poisto && j - i != maxdiag)         //  Onko poisto pienempi kuin
            {                                              //  korvaus? Poisto-askelta ei oteta
                apuri = poisto;                            //  huomioon jos ollaan maxdiag-
            }                                              //  diagonaalilla.
            if(apuri > lisays && j - i != mindiag)         //  Onko lisäys edullisempi kuin
            {                                              //  min{korvaus, poisto}? Lisäys-
                apuri = lisays;                            //  askelta ei huomioida jos ollaan
            }                                              //  mindiag-diagonaalilla.
            taulukko[i][j] = apuri;                  //  Taulukko[i][j] saa edellä lasketun minimiarvon.
            i++;                                     // Siirrytään sarakkeessa seuraavaan alkioon.
        }
        j++;                                               // Siirrytään seuraavaan sarakkeeseen.
    }
    t = t*2;                                     // Kasvatetaan arviota t kertomalla se kahdella.
}                                                // Silmukan loppu, suorittaminen lopetetaan heti
while(taulukko[m1][m2] > t/2);                   // kun taulukko[m1][m2] £ t (tässä otettu huomioon
                                                 // edeltävä t:n kasvatus), sillä tällöin
                                                 // taulukko[m1][m2] = Te(m1, m2) = E(mj1, mj2).
```

Algoritmi 2.2.24 (jatkoa):
Ukkosen editointietäisyysalgoritmi.

Ukkosen algoritmin aikavaativuus on selvästi suoraan verrannollinen sen täyttämien taulukon T_e alkioiden lukumäärään. Jokaista etäisyysarvion *t* arvoa kohti täytetään lauseen 2.2.21 mukaisesti enintään $t / \min\{\omega(P), \omega(L)\}$ diagonaalia, ja jokaisella taulukon T_e diagonaalilla on korkeintaan $\min\{m_1^0, m_2^0\}$ alkiota. Koska algoritmin suoritus lopetetaan, kun $t \geq E(mj_1^0, mj_2^0)$, ja arvo *t* kasvaa joka kierroksella aina kaksinkertaiseksi, saadaan Ukkosen algoritmin täyttämien alkioiden kokonaismäärälle yläraja $\Sigma(\min\{m_1^0, m_2^0\} * 2 * E(mj_1^0, mj_2^0)/ (2^k * \min\{\omega(P), \omega(L)\}), 0 < i) = \min\{m_1^0, m_2^0\} * 2 * E(mj_1^0, mj_2^0) * \Sigma(1 / 2^k, 0 < i) / \min\{\omega(P), \omega(L)\}$. Koska $\Sigma(1 / 2^k, 0 < i) \leq 2$, seuraa tästä että Ukkosen algoritmin aikavaativuus on $O(\min\{m_1^0, m_2^0\} * E(mj_1^0, mj_2^0) / \min\{\omega(P), \omega(L)\})$.

2.3 Likimääräisen esiintymän etsiminen tekstistä

Tutkitaan nyt aiemman pohjalta kaikkien merkkijonon *hahmo*[0…*m*-1] likimääräisten esiintymien etsimistä tekstistä *teksti*[0…*n*-1] suurimman sallitun editointietäisyyden ollessa *k*. Edellä esitetyt algoritmit eivät sovellu suoraan tähän tarkoitukseen kovinkaan hyvin, sillä ne toimivat sellaisenaan vain aina tiettyjen kahden merkkijonon välisen kokonaiseditointietäisyyden laskemisessa. Koska editointietäisyydellä *k* merkkijonosta *hahmo*[0…*m*-1] oleva tekstin osa voi olla pituudeltaan välillä $m - k$ ja $m + k$ vastaten ääritapauksia, joissa kaikki operaatiot ovat joko lisäyksiä tai poistoja, pitäisi koko tekstin tutkimiseksi tutkia erikseen kaikki tekstin osajonot, joiden pituudet osuvat tälle välille. Tämä on selvästi aivan liian työlästä ja tehotonta. Ongelmaan löytyy onneksi yksinkertainen tehokkaampi ratkaisu tarkastelemalla merkkijonojen *hahmo*[0…*m*-1] ja *teksti*[0…*n*-1] välistä editointietäisyystaulukkoa [Sellers, 1980]. Tämän taulukon alkio $T_e(i, j)$ kertoo aina merkkijonojen *hahmo*[0…*i*-1] ja *teksti*[0…*j*-1] välisen editointietäisyyden, kun hahmon likimääräisen esiintymän kannalta ollaan kiinnostuneita tyyppiä *hahmo*[0…*m*-1] ja *teksti*[*j*-*h*+1…*j*] olevien merkkijonojen välisestä etäisyydestä. Haluttuun tulokseen päästään julistamalla hahmon ensimmäisen merkin eteen tehtävät lisäysoperaatiot ilmaisiksi, mikä ilmenee seuraavan lauseen tuloksesta:

Lause 2.3.1:
Jos merkkijonon mj_1^0 ensimmäisen merkin eteen tehtävät lisäysoperaatiot ovat ilmaisia, pätee $E(mj_1^0[0\ldots i], mj_2^0[0\ldots j]) \leq E(mj_1^0[0\ldots i], mj_2^0[j\text{-}h+1\ldots j])$, missä $0 \leq j < m_2^0$, $0 \leq i < m_1^0$ ja $0 < h \leq j + 1$.

Todistus:
Lauseen väite on ilmeinen, sillä jos $j > 0$, voidaan oletuksen mukaan merkkijonon $mj_2^0[0\ldots j\text{-}h]$ lisääminen merkkijonon mj_1^0 alkuun tehdä ilmaiseksi. Näin ollen merkkijono $mj_1^0[0\ldots i]$ voidaan aina muuttaa merkkijonoksi $mj_2^0[0\ldots j]$ suorittamalla ensin edellämainittu työtä tekemätön lisäys ja tekemällä sen jälkeen arvoa $E(mj_1^0[0\ldots i], mj_2^0[j\text{-}h+1\ldots j])$ vastaavat operaatiot, jolloin kokonaistyö on $E(mj_1^0[0\ldots i], mj_2^0[j\text{-}h+1\ldots j])$.

Lauseen 2.3.1 oleellinen sisältö hahmon likimääräisten esiintymien etsimisen suhteen on se, että alkulisäysten ollessa ilmaisia voidaan keskittyä vertailemaan hahmoa editointietäisyystaulukon sisältämän informaation mukaisesti tyyppiä *teksti*[0…*j*] oleviin merkkijonoihin. Nimittäin jos merkkijonolla mj_1^0 on kohtaan *j* päättyvä likimääräinen esiintymä merkkijonossa mj_2^0, on tällöin määritelmän 2.1.2 mukaan olemassa jokin sellainen

positiiviluku h, että $E(mj_1^0[0\ldots m_1^0\text{-}1], mj_2^0[j\text{-}h\text{+}1\ldots j]) \leq s_e$. Alkulisäysten ollessa ilmaisia pätee tällöin lauseen 2.3.1 nojalla myös ehto $E(mj_1^0[0\ldots m_1^0\text{-}1], mj_2^0[0\ldots j]) \leq s_e$. Toisaalta selvästi huomataan, että jos merkkijonolla mj_1^0 ei ole kohtaan j päättyvää likimääräistä esiintymää merkkijonossa mj_2^0, on pakko päteä $E(mj_1^0[0\ldots m_1^0\text{-}1], mj_2^0[0\ldots j]) > s_e$, sillä muutenhan merkkijono $mj_2^0[0\ldots j]$ olisi kokonaisuudessaan hahmon mj_1^0 likimääräinen esiintymä.

Merkkijonon mj_1^0 alkuun tapahtuvien lisäysten muuttaminen ilmaiseksi onnistuu kätevästi editointietäisyystaulukon kannalta. Ainoa muutos aiempaan arvojen laskemistapaan on se, että nyt kaikki muotoa $T_e(\text{-}1, j)$ olevat alkiot alustetaankin arvolla 0. Käytetään tällä tavalla täytetystä editointietäisyystaulukosta merkintää T_e. Nyt edellisen käsittelyn pohjalta voidaan todeta, että merkkijonolla mj_1^0 on kohtaan j merkkijonossa mj_2^0 päättyvä likimääräinen esiintymä jos ja vain jos $\mathsf{T}_e(m_1^0\text{-}1, j) \leq s_e$. Näin ollen haluttaessa löytää hahmon esiintymät tekstistä riittää täyttää taulukko T_e merkkijonojen *hahmo*$[0\ldots m\text{-}1]$ ja *teksti*$[0\ldots n\text{-}1]$ osalta siten, että aina tyyppiä $\mathsf{T}_e(m\text{-}1, j)$ olevan alkion arvoa laskettaessa merkitään muistiin luku j, mikäli tämä laskettu arvo on pienempi tai yhtäsuuri kuin suurin sallittu virhe s_e. Tällä tavalla tuloksena on lista kaikista sellaisista tekstin kohdista, joihin päättyy jokin hahmon likimääräinen esiintymä. Algoritmissa 2.2.15 esitetty editointitaulukon täyttävä dynaaminen perusratkaisu on helppo muokata toteuttamaan tällainen etsintä esimerkiksi algoritmin 2.3.2 esittämällä tavalla. Muutos ei vaikuta lainkaan aikavaativuuteen, ja siten algoritmin 2.3.2 aikavaativuus on $O(m\times n)$.

Koska taulukko T_e täytetään aivan samanlaisia editointiaskeleita mukaillen kuin taulukko T_e, voidaan jotain kohtaan j päättyvää hahmon likimääräistä esiintymää vastaava editointipolku etsiä samalla tavalla alkiosta $\mathsf{T}_e(m\text{-}1, j)$ peruuttamalla kuin taulukon T_e yhteydessäkin. Erona on, että nyt peruuttaminen voidaan lopettaa, kun saavutaan ensimmäiseen muotoa $\mathsf{T}_e(\text{-}1, j\text{-}h\text{+}1)$ olevaan alkioon. Tällöin $E($*hahmo*$[0\ldots m\text{-}1],$ *teksti*$[j\text{-}h\text{+}2\ldots j]) = \mathsf{T}_e(m\text{-}1, j)$, ja peruutuksen yhteydessä läpikäydyt alkiot muodostavat päinvastaisessa järjestyksessä näitä merkkijonoja *hahmo*$[0\ldots m\text{-}1]$ ja *teksti*$[j\text{-}h\text{+}2\ldots j]$ vastaavan minimaalisen editointipolun.

```
int i;                                          // Hahmoa läpikäyvä indeksi.
int j;                                          // Tekstiä läpikäyvä indeksi.
int m;                                          // Hahmon pituus.
int n;                                          // Tekstin pituus.
int wl, wp, wk;                    // Lisäys-, poisto- ja korvausoperaatioiden kustannukset.
int lisays, poisto, korvaus;       // Laskusääntöä sovellettaessa käytettävät muuttujat.
int taulukko[m + 1][n + 1];                     // Editointietäisyystaulukko T_e.
int se;                  // Suurin sallittu likimääräisen esiintymän etäisyys, oletetaan tunnetuksi.
int apuri;                                      // Apumuuttuja.

for(apuri = 0; apuri < m + 1; apuri++)          // Alustetaan muotoa T_e(i, -1) olevat alkiot.
{
    taulukko[apuri][0] = apuri*wp;
}
for(apuri = 0; apuri < n + 1; apuri++)          // Alustetaan muotoa T_e(-1, j) olevat alkiot.
{                                               // Nyt alkulisäykset ovat ilmaisia, eli kaikki
    taulukko[0][apuri] = 0;                     // kyseiset alkiot alustetaan arvolla 0.
}
j = 1;                       // Alustetaan indeksi j ensimmäisenä täytettävän sarakkeen kohdalle.
while(j < n + 1)                    // Edetään sarakkeittain, siis ulompi silmukkamuuttuja on j.
{
    i = 1;                   // Indeksi i alustetaan arvolla 1, eli sarakkeen ylimpään tyhjään alkioon.
    while(i < m + 1)                            // Edetään sarakkeen alimpaan riviin asti.
    {
        lisays = taulukko[i][j-1] + wl;         // Lasketaan eri vaihtoehdot laskusäännön
        poisto = taulukko[i-1][j] + wp;         // soveltamista varten.
        korvaus = taulukko[i-1][j-1] + (mj1[i-1] != mj2[j-1])*wk;
        apuri = korvaus;                // Muuttujaan ”apuri” lasketaan edellisten minimi.
        if(apuri > poisto)                      // Onko poisto pienempi kuin korvaus?
        {
            apuri = poisto;
        }
        if(apuri > lisays)              // Onko lisäys edullisempi kuin min{korvaus, poisto}?
        {
            apuri = lisays;
        }
        taulukko[i][j] = apuri;         // Taulukko[i][j] saa edellä lasketun minimiarvon.
        i++;                            // Siirrytään sarakkeen seuraavaan alkioon.
    }
    if(taulukko[m][j] <= se)            // Löytyikö merkkiin teksti[j] päättyvä hahmon
    {                                   // likimääräinen esiintymä?
        loytyi(j);                      // Likimääräinen esiintymä löytyi kohdasta j;
    }
    j++;                                // Siirrytään seuraavaan sarakkeeseen.
}
```

Algoritmi 2.3.2:
Perusdynaaminen likimääräisen haun toteuttava algoritmi.

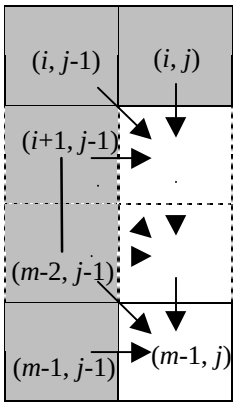
	j	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
i			e	n	t	t	e	n		t	e	n	t	t	e	n		t	e	e	l	i	k	a	m	e	n	t	t	e	n
-1		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	e	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	0	1	1	1	1	1	0	1	1	1	0	1
1	n	2	1	0	1	2	1	0	1	2	1	0	1	2	1	0	1	2	1	1	1	2	2	2	2	1	0	1	2	1	0
2	t	3	2	1	0	1	2	1	1	1	2	1	0	1	2	1	1	1	2	2	2	2	3	3	3	2	1	0	1	2	1
3	t	4	3	2	1	0	1	2	2	1	2	2	1	0	1	2	2	1	2	3	3	3	3	4	4	3	2	1	0	1	2
4	e	5	4	3	2	1	0	1	2	2	1	2	2	1	0	1	2	2	1	2	3	4	4	4	5	4	3	2	1	0	1
5	n	6	5	4	3	2	1	0	1	2	2	1	2	2	1	0	1	2	2	2	3	4	5	5	5	5	4	3	2	1	0

Kuva 2.3.3:
Merkkijonon *hahmo*[0...5] = "tentten" likimääräisten esiintymäkohtien etsiminen merkkijonosta *teksti*[0...28] = "entten tentten teelikamentten" perusdynaamista algoritmia käyttäen, kun suurin sallittu editointietäisyys on 1. Kaikki löydettyjä esiintymiä vastaaville editointipoluille kuuluvat taulukon alkiot on tummennettu.

Tällaista taulukointiin perustuvaa likimääräistä etsintää voidaan yrittää nopeuttaa lähinnä kahdella eri tavalla: tehostamalla taulukon T_e täyttämistä tai pienentämällä tutkittavaa tekstin osaa. Käsittelen seuraavassa hieman molempia lähestymistapoja.

2.3.1 Taulukon T_e tehokkaampi täyttäminen

Tarkastellaan tilannetta, jossa käytetään dynaamista sarakkeittain etenevää alkioiden arvojen laskujärjestystä. Hahmon likimääräisen esiintymän etsimisen tehostamiseksi kustakin sarakkeesta haluttaisiin täyttää aina mahdollisimman pieni määrä alkioita. Hahmon likimääräisen esiintymän löytymisen kannalta oleellisia ovat vain sellaiset alkiot $T_e(i, j)$, joiden arvoilla pätee $T_e(i, j) \leq s_e$. Otetaan tämä lähtökohdaksi laskettavien alkioiden rajoittamiseen siten, että aina kunkin sarakkeen j täyttäminen lopetetaan heti ensimmäisen sellaisen alkion $T_e(i, j)$ kohdalla, että voidaan todeta, että loput tämän sarakkeen alkiot, eli alkiot $T_e(i+1, j)$, $T_e(i+2, j-1)$,…, $T_e(m-1, j)$, saisivat lukua s_e suuremman arvon. Nyt ongelmana on luonnollisesti, kuinka tämä lopetusehto sitten ylipäätään voidaan todeta? Lauseen 2.2.13 laskusäännöstä nähdään, että alkio $T_e(i, j)$ saa aina jonkin arvoista $T_e(i-1, j) + \omega(P)$, $T_e(i, j-1) + \omega(L)$, $T_e(i, j-1) + \omega(K)$ ja $T_e(i-1, j-1)$. Tästä voidaan päätellä, että arvot $T_e(i+1, j)$, $T_e(i+2, j)$,…, $T_e(m-1, j)$ ovat suurempia kuin s_e varmasti ainakin silloin, jos $T_e(i, j) \geq s_e$ ja lisäksi myös sarakkeella $j - 1$ olevien alkioiden $T_e(i, j-1)$, $T_e(i+1, j-1)$,…, $T_e(m-1, j-1)$ arvot ovat suurempia kuin s_e. Tällöin sarakkeen j täyttäminen voidaan siis lopettaa alkioon $T_e(i, j)$. Kuva 2.3.4 havainnollistaa asiaa.



Kuva 2.3.4:
Editointitaulukossa askeleet menevät ainoastaan oikealle ja/tai alas, ja jokainen editointiaskel säilyttää editointipolulla tehdyn työmäärän vähintään askeleen lähtoalkion sisältämän arvon suuruisena. Siten on selvää, että jos alkion (i, j) sekä alkioiden $(i, j-1)$, $(i+1, j-1)$, …, $(m-1, j-1)$ arvot ovat suurempi kuin s_e (nämä alkiot väritetty kuvassa harmaalla), niin silloin pakostakin myös alkioiden $(i+1, j)$, $(i+2, j)$, …, $(m-1, j)$ arvot ovat suurempia kuin s_e, sillä niihin alkiosta (0, 0) johtavat editointipolut sisältävät aiemmassa kohdassa jonkin edellä mainituista alkioista.

Lopetusehdon soveltamiseksi sarakkeen *j* kohdalla tarvitsee aina tietää pienin sellainen luku *i*, jolle pätee, että alkioiden $T_e(i, j\text{-}1)$, $T_e(i\text{+}1, j\text{-}1)$, …, $T_e(m\text{-}1, j\text{-}1)$ arvot ovat suurempia kuin s_e. Tämä tieto on kuitenkin helppo kirjata muistiin samalla, kun sarake $j - 1$ täytetään, ja tämähän tehdään aina juuri ennen sarakkeen *j* täyttämistä.

Käytetty rajoitussääntö on hyvin yksinkertainen ja ensimmäisenä täsmälleen samaan lopputulokseen johtavan menetelmän esitti Esko Ukkonen [Ukkonen, 1985b]. Hän tosin esittää asian laajemmassa yhteydessä (ja mielestäni siksi hieman monimutkaisemmin) pohjautuen taulukon T_e diagonaalien ominaisuuksiin, joten esitin mieluummin oman lyhyen ja yksinkertaisen versioni rajoitussäännön päättelemisestä. Algoritmi 2.3.5 on saatu muokkaamalla algoritmi 2.3.2 toimimaan Ukkosen rajoitussäännön mukaisesti.

```
int i;                                          // Hahmoa läpikäyvä indeksi.
int j;                                          // Tekstiä läpikäyvä indeksi.
int m;                                          // Hahmon pituus.
int n;                                          // Tekstin pituus.
int wl, wp, wk;                        // Lisäys-, poisto- ja korvausoperaatioiden kustannukset.
int lisays, poisto, korvaus;           // Laskusääntöä sovellettaessa käytettävät muuttujat.
int taulukko[m + 1][n + 1];                     // Editointietäisyystaulukko Te.
int se;                  // Suurin sallittu likimääräisen esiintymän etäisyys, oletetaan tunnetuksi.
int apuri;                                      // Apumuuttuja.

int edellinenrivi;                     // Alimman edellisessä sarakkeessa olevan alkion indeksi, jonka
                                       // arvo korkeintaan suurimman sallitun etäisyyden suuruinen.
int alinrivi;                          // Alimman parhaillaan tarkasteltavan sarakkeen alkion
                                       // indeksi, jonka arvo korkeintaan suurin sallittu etäisyys.
for(apuri = 0; apuri < m + 1; apuri++) // Alustetaan muotoa Te(i, -1) olevat alkiot.
{
    taulukko[apuri][0] = apuri * wp;
}
for(apuri = 0; apuri < n + 1; apuri++) // Alustetaan muotoa Te(-1, j) olevat alkiot. Nyt alkulisäykset
{                                      // ovat ilmaisia, joten kaikki kyseiset alkiot alustetaan arvolla 0.
    taulukko[0][apuri] = 0;
}

edellinenrivi = se / wp;               // 0-sarakkeen osalta voidaan laskea suoraan, mikä on sen
if(edellinenrivi > m + 1)              // alimman sellaisen alkion indeksi, jonka arvo on
{                                      // korkeintaan suurin sallittu etäisyys. Tarkistetaan lisäksi,
    edellinenrivi = m + 1;             // että kyseinen arvo ei mene taulukon ulkopuolelle.
}
j = 1;                         // Alustetaan indeksi j ensimmäisenä täytettävän sarakkeen kohdalle.
while(j < n + 1)                       // Edetään sarakkeittain, siis ulompi silmukkamuuttuja on j.
{
    i = 1;                     // Indeksi i alustetaan arvolla 1 sarakkeen ylimpään tyhjään alkioon.
```

Algoritmi 2.3.5 (jatkuu seuraavalla sivulla):
Ukkosen tehostettu likimääräisen haun toteuttava algoritmi.

```
    while(i <= edellinenrivi)                    // Edetään alimman sellaisen edellisessä sarakkeessa olevan
    {                                            // alkion tasolle, jonka arvo korkeintaan suurin sallittu virhe.
        lisays = taulukko[i][j-1] + wl;          // Lasketaan eri vaihtoehdot laskusäännön
        poisto = taulukko[i-1][j] + wp;          // soveltamista varten.
        korvaus = taulukko[i-1][j-1] + (mj1[i-1] == mj2[j-1])*wk;
        apuri = korvaus;                         // Muuttujaan ”apuri” lasketaan edellisten minimi.
        if(apuri > poisto)                       // Onko poisto pienempi kuin korvaus?
        {
            apuri = poisto;
        }
        if(apuri > lisays)                       // Onko lisäys edullisempi kuin min{korvaus, poisto}?
        {
            apuri = lisays;
        }
        taulukko[i][j] = apuri;                  // Taulukko[i][j] saa edellä lasketun minimiarvon.
        if(apuri <= se)                          // Päivitetään alinrivi-muuttujan arvo, jos juuri lasketun
        {                                        // alkion arvo korkeintaan suurimman sallitun virheen
            alinrivi = i;                        // suuruinen.
        }
        i++;                                     // Siirrytään sarakkeen seuraavaan alkioon.
    }
    if(i <= m)                                   // Täytetään vielä se alkio, joka on lopetussäännössä vinottain
    {                                            // edellisen sarakkeen alimman sellaisen alkion kanssa, jonka
                                                 // arvo on korkeintaan suurin sallittu virhe.
        poisto = taulukko[i-1][j] + wp;          // Lasketaan eri vaihtoehdot laskusäännön soveltamista
        korvaus = taulukko[i-1][j-1] + (mj1[i-1] == mj2[j-1])*wk;  // varten, mutta nyt ei oteta huomioon
                                                                   // lisäys-operaatiota, joka tulee
                                                                   // laskettavan alueen ulkopuolelta.
        apuri = korvaus;                         // Muuttujaan ”apuri” lasketaan edellisten minimi.
        if(apuri > poisto)                       // Onko poisto pienempi kuin korvaus?
        {
            apuri = poisto;
        }
        taulukko[i][j] = apuri;                  // Taulukko[i][j] saa edellä lasketun minimiarvon.
        if(apuri <= se)                          // Päivitetään alinrivi-muuttujan arvo, jos juuri lasketun
        {                                        // alkion arvo korkeintaan suurimman sallitun virheen
            alinrivi = i;                        // suuruinen.
        }
    }
    if(j == m && taulukko[m][j] <= se)                   // Löytyikö merkkiin teksti[j] päättyvä
    {                                                    // hahmon likimääräinen esiintymä?
        loytyi(j);                                   // Likimääräinen esiintymä löytyi kohdasta j;
    }
    edellinenrivi = alinrivi;                        // Päivitetään edellinenrivi- ja alinrivi-muuttujat
    alinrivi = 1;                                    // vastaamaan etenemistä seuraavaan sarakkeeseen.
    j++;                                                 // Siirrytään seuraavaan sarakkeeseen.
}
```

Algoritmi 2.3.5 (jatkoa):
Ukkosen tehostettu likimääräisen haun toteuttava algoritmi.

Algoritmin 2.3.5 aikavaativuus on pahimmassa tapauksessa sama kuin perusdynaamisessa ratkaisussa, eli O($m \times n$). Yksi patologinen esimerkki tästä olisi esimerkiksi tilanne, jossa sekä

hahmo että teksti koostuvat kokonaan samoista kirjaimista. Tällöin jokaisessa mahdollisessa tekstin kohdassa löytyisi hahmon eksakti (ja siten samalla myös likimääräinen) esiintymä, ja editointietäisyystaulukko jouduttaisiin täyttämään kokonaan sarakkeesta $\lfloor m - (s_e / \omega(L))\rfloor$ alkaen. Keskimäärin Ukkosen likimääräisen etsintäalgoritmin on kuitenkin todettu toimivan ajassa $s_e \times n$ [Chang and Lampe, 1992].

	j	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
i			e	n	t	t	e	n		t	e	n	t	t	e	n		t	e	e	l	i	k	a	m	e	n	t	t	e	n
-1		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	e	**1**	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	0	1	**1**	**1**	**1**	**1**	0	1	1	1	0	1
1	n	2	**1**	0	1	2	1	0	1	2	1	0	1	2	1	0	1	2	1	**1**	**1**	2	2	2	2	**1**	0	1	2	1	0
2	t	3		**1**	0	1	2	1	1	1	2	1	0	1	2	1	1	1	2	2	2	2					**1**	0	1	2	1
3	t	4			**1**	0	1	2	2	**1**	2	2	**1**	0	1	2	2	**1**	2	3								**1**	0	1	2
4	e	5				**1**	0	1	2	2	**1**	2	2	**1**	0	1	2	2	**1**	2									**1**	0	1
5	n	6					**1**	**0**	**1**	2		**1**	2		**1**	**0**	**1**	2		2										**1**	**0**

Kuva 2.3.6:
Merkkijonon *hahmo*[0...5] = "tentten" likimääräisten esiintymäkohtien etsiminen merkkijonosta *teksti*[0...28] = "entten tentten teelikamentten" Ukkosen parannettua dynaamista algoritmia käyttäen, kun suurin sallittu editointietäisyys on 1. Kaikki löydettyjä esiintymiä vastaaville editointipoluille kuuluvat taulukon alkiot on tummennettu, ja kunkin sarakkeen alin korkeintaan virhemarginaalin suuruinen arvo on lihavoitu.

2.3.2 Tekstin nopea karsinta

Yleensä käytännössä tilanne on sellainen, että etsittävän hahmon likimääräisiä esiintymiä löytyy ainoastaan pienestä osasta tekstiä. Tällöin tehdään paljon ”turhaa” työtä täytettäessä taulukko T_e koko tekstin osalta. Tavoitteena on siksi yrittää karsia nopeasti pois sellaiset osat tekstistä, jotka eivät ainakaan voi sisältää hahmon likimääräistä esiintymää. Karsinnan idea vastaa näin ollen perusajatukseltaan kappaleessa 1.6 esiteltyä Karp-Rabin-algoritmia, eli aluksi kirjataan muistiin joitain hahmon ominaisuuksia, joiden perusteella voidaan sitten nopeasti päätellä, mitkä tekstinosat voivat mahdollisesti olla riittävän samankaltaisia hahmon kanssa. Karsimisen lopputuloksena on näin ollen lista niistä tekstinosista, jotka pitää vielä tutkia tarkemmin, ja tämä viimeinen vaihe voidaan tehdä yksitellen jokaiselle kyseisistä osista käyttäen esimerkiksi taulukon T_e muodostamiseen perustuvaa likimääräistä merkkijonotäsmäysalgoritmia. Jotta karsinnasta olisi jotain käytännön hyötyä, on sen onnistuttava reilusti nopeammin kuin täydellisen taulukon T_e muodostamisen. Tässä esitetään yksi yleinen perusperiaate, jonka mukaan karsinta voidaan suorittaa, sekä kaksi hieman eri tyyppistä tapaa soveltaa kyseistä periaatteetta käytännössä. Tarkastelun aikana oletetaan koko ajan sellainen alkutilanne, että halutaan etsiä hahmon likimääräisiä esiintymiä tekstistä suurimman sallitun editointietäisyyden ollessa s_e.

Yleinen peruskarsintaperiaate

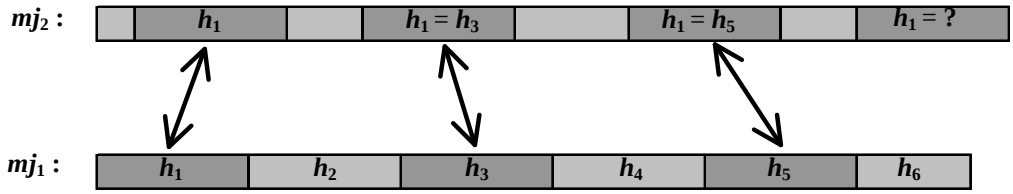
Tarkastellaan tilannetta, jossa merkkijono *hahmo*[0…*m*-1] on jaettu erillisiin osamerkkijonoihin h_1, h_2, …, h_z seuraavasti (tässä $z = 5$):

Yksittäinen operaatio voi kohdistua vain yhteen merkkiin, joten jokainen yksittäinen merkkijonoon *hahmo*[0…*m*-1] kohdistuva operaatio voi luonnollisesti muuttaa aina vain yhtä osamerkkijonoista h_1, h_2, …, h_z. Koska suurin mahdollinen korkeintaan luvun s_e verran työtä tekevä operaatioiden lukumäärä on $\lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\} \rfloor$, tiedetään, että mielivaltainen korkeintaan luvun s_e verran työtä tekevä editointioperaatiojoukko voi kohdistua enintään $\lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\} \rfloor$ eri osamerkkijonoon. Tästä seuraa, että jos osamerkkijonojen lukumäärä z on riittävän suuri, eli tarkemmin ottaen $z > \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\} \rfloor$, on hahmon likimääräisen esiintymän pakko sisältää vähintään $z - \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\} \rfloor$ kappaletta osamerkkijonoista h_1, h_2, ..., h_z sellaisinaan, eksaktisti ja erillisinä. Vaatimus erillisyydestä seuraa suoraan siitä, että merkkijonot h_1, h_2, …, h_z esiintyvät erillisinä

merkkijonossa mj_1. Puetaan tämän yksinkertaisen päättelyn lopputulos seuraavan lauseen muotoon:

Lause 2.3.5:
Olkoot mj_1 ja mj_2 kaksi mielivaltaista ei-tyhjää merkkijonoa, z sellainen kokonaisluku, että $z > \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\}\rfloor$, ja $h_1, h_2, \ldots, h_z$ sellaisia epätyhjiä merkkijonoja, että $mj_1 = h_1 \cup h_1 \cup \ldots \cup h_z$. Tällöin ehto $E(mj_1, mj_2) \leq s_e$ voi olla voimassa ainoastaan siinä tapauksessa, että merkkijono mj_2 sisältää eksaktisti ja erillisinä vähintään $z - \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\}\rfloor$ kappaletta merkkijonoista $h_1, h_2, \ldots, h_z$.

Lauseen 2.3.5 tulkinnassa on oleellista huomata, että jos jotkin merkkijonoista $h_1, h_2, \ldots, h_z$ ovat keskenään identtisiä, eli esimerkiksi $h_a = h_b = h_c$ missä a, b ja c ovat keskenään erisuuria ja kuuluvat joukkoon $\{1, 2, \ldots, z\}$, niin tällöin kyseisen osamerkkijonon esiintymä tekstissä vastaa kuitenkin ainoastaan yhtä osamerkkijonoista h_a, h_b ja h_c, ei kaikkia kolmea. Toisaalta mitään merkkijonoa ei myöskään lasketa kahteen kertaan, eli jos edellisessä tilanteessa olisi löydetty merkkijonosta mj_2 jo kolme merkkijonon $h_a = h_b = h_c$ esiintymää, ja ei ole olemassa sellaista lukua k, että k on erisuuri kuin a, b tai c ja $h_a = h_k$, niin neljättä merkkijonon h_a esiintymää ei enää lasketa, koska merkkijonosta mj_1 ei enää löydy sitä vastaavaa erillistä ja eksaktia osamerkkijonoa (kuva 2.3.6).



Kuva 2.3.6:
Tilanne, jossa merkkijono mj_1 on jaettu kuuteen erilliseen osamerkkijonoon $h_1, \ldots, h_6$ ja pätee $h_1 = h_3 = h_5$ sekä lisäksi $h_1 \neq h_2$, $h_1 \neq h_4$, ja $h_1 \neq h_6$. Jos tällöin merkkijonosta mj_2 löytyy kolme merkkijonon h_1 esiintymää, löytyy niistä jokaiselle oma vastinparinsa merkkijonon mj_1 osituksesta, eli merkkijonot h_1, h_3 ja h_5 (vastinparit merkitty kuvassa nuolilla). Mutta jos merkkijonosta mj_2 löytyy vielä neljäs merkkijonon h_1 esiintymä, ei sille enää löydy vastinparia, koska kaikki jäljellä olevat osajonot h_2, h_4 ja h_6 ovat sen kanssa erilaisia.

Koska suurin mahdollinen hahmon likimääräisen esiintymän pituus on $m + \lfloor s_e / \omega(P) \rfloor$ merkkiä, tiedetään lauseen 2.3.5 pohjalta, että tekstistä riittää tutkia kaikki sellaiset $\lfloor m + s_e / \omega(P) \rfloor$ merkin pituiset kohdat, jotka sisältävät vähintään $z - \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\}\rfloor$ kappaletta näistä osamerkkijonoista. Jotta mitään tekstin osaa ei tutkittaisi useampaan kuin

yhteen kertaan, kannattaa keskenään päällekkäin menevät tutkittavat kohdat aina yhdistää yhdeksi yhtenäiseksi laajemmaksi tutkittavaksi alueeksi.

Teoriassa on mahdollista, että $\lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\}\rfloor > m$, jolloin hahmoa ei ole mahdollista jakaa riittävän moneen erilliseen osamerkkijonoon. Tällöin lauseen 2.3.5 mukaista karsintaa ei voida soveltaa, mutta käytännössä tällaista tilannetta tuskin esiintyy, sillä niin suuren virhemarginaalin soveltaminen, että se sallisi jopa enemmän kuin yhden editointioperaation hahmon merkkiä kohden, ei liene yleensä mielekästä.

Kun tekstistä ei ole käytettävissä kattavaa ennakkotietoa, kannattaa hahmo jakaa mahdollisimman tasapituisiin osamerkkijonoihin, jotta niiden esiintymistodennäköisyydet olisivat keskenään suunnilleen yhtäsuuret. Käytännössä tähän päästään asettamalla jokaisen osamerkkijonon pituudeksi $q = \lfloor m / z \rfloor$, missä z on osamerkkijonojen lukumäärä. Tästä juontuu karsinnan yhteydessä käytetty hienolta kuulostava nimitys ”q-grammi”, joka tarkoittaa yksinkertaisesti q merkkiä pitkää merkkijonoa. Siis esimerkiksi merkkijonot ”abba” ja ”baba” ovat 4-grammeja, ja $h_1, h_2, \ldots, h_z$ ovat $\lfloor m / z \rfloor$-grammeja. Yleensä $z \times \lfloor m / z \rfloor \neq m$, eli jako ei mene tasan. Yli jäävä hahmon loppuosa voidaan yhdistää viimeisen osamerkkijonon h_z loppuun, mutta yksinkertaisuuden vuoksi voi olla järkevämpää pitää kaikki osamerkkijonot tasapituisina ja jättää tämä ylimääräinen hahmon osa huomioimatta.

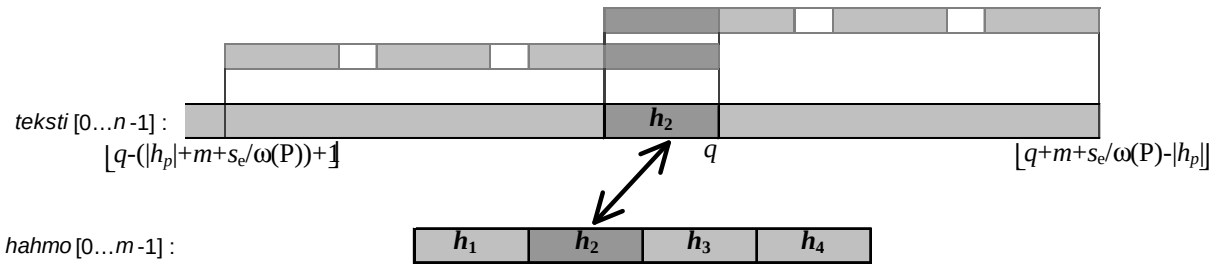
Lause 2.3.5 voidaan melko suoraviivaisesti yleistää koskemaan myös osittain keskenään päällekkäin meneviä q-grammeja [Sutinen, 1998], mutta en käsittele tätä laajempaa karsintakriteeriä tässä.

Ensimmäinen karsintamenetelmä: Baeza-Yates-Perleberg

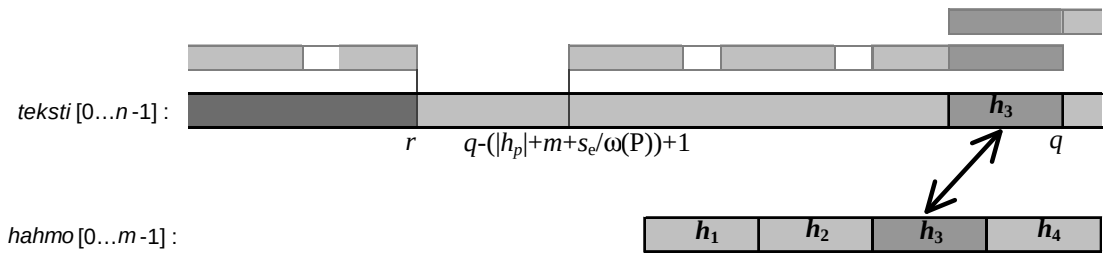
Yksi suoraviivainen ja useissa tapauksissa melko tehokas tapa soveltaa lausetta 2.3.5 karsimisessa on seuraavanlainen Baeza-Yatesin ja Perlebergin [Baeza-Yates and Perleberg, 1996] julkaisema menetelmä. Ajatuksena on valita osamerkkijonojen lukumäärä z pienimmäksi mahdolliseksi, jolloin siis $z = \lfloor s_e / \min\{w(P), w(L), w(K)\} \rfloor + 1$, ja jakaa sen jälkeen hahmo tasaisesti erillisiin osamerkkijonoihin h_1, h_2, …, ja h_z. Aiemman perusteella nämä merkkijonot ovat tällöin $\lfloor m / z \rfloor$-grammeja.

Jos hahmo on jaettu tällä tavalla osamerkkijonoihin, tiedetään, että hahmon jokaisen likimääräisen esiintymän on sisällettävä ainakin yksi osamerkkijonoista h_1, h_2, …, h_z. Koska nyt riittää tutkia ainoastaan sellaiset korkeintaan $m + s_e / \omega(P)$ merkin pituiset kohdat, jotka sisältävät vähintään yhden näistä osamerkkijonoista, voidaan karsinta tehdä seuraavasti: Etsitään kaikki merkkijonojen h_1, h_2, …, h_z eksaktit esiintymät tekstistä, ja aina, kun löytyy jokin merkkijonon h_p esiintymä, joka loppuu kohtaan *teksti*[q] (eli *teksti*[$q-|h_p|+1 \ldots q$] = h_p), lisätään väli *teksti*[$q-(m+|h_p|+\lfloor s_e/\omega(P) \rfloor)+1 \ldots q+m-|h_p|+\lfloor s_e/\omega(P) \rfloor$] tutkittavien kohtien joukkoon. Nimittäin jokainen edellä mainitulla alueella sijaitseva $m + \lfloor s_e / \omega(P) \rfloor$ merkkiä pitkä merkkijono voi lauseen 2.3.5 mukaista karsintaa sovellettaessa sisältää potentiaalisesti hahmon likimääräisen esiintymän ja on siksi tutkittava (kuva 2.3.7).

Osamerkkijonojen etsintävaihe voidaan toteuttaa tehokkaasti käyttäen esimerkiksi kappaleessa 1.7 käsiteltyä Aho-Corasick-algoritmia, ja varsinainen likimääräisellä etsintäalgoritmilla suoritettava tarkistus voidaan tehdä aina sitä mukaa, kun löydetään ”kokonaisia” tekstistä tutkittavia alueita. Tämä viimeksimainittu tarkoittaa sitä, että kun joko saavutaan tekstin loppuun tai löydetään sellainen osamerkkijono, jonka mukaisesti tutkittava alue ei mene enää päällekkäin aiemmin tutkittavaksi todetun alueen kanssa, ei tämä aiempi alue enää tule muuttumaan, ja se voidaan tutkia (kuva 2.3.8).



Kuva 2.3.7:
Tilanne, jossa hahmo on jaettu neljään erilliseen osamerkkijonoon h_1, …, h_4 ja tekstistä löytyy merkkiin *teksti*[q] päättyvä merkkijonon h_2 esiintymä. Tällöin lauseen 2.3.2 periaatetta noudattaen pitää tutkia kaikki sellaiset tekstin osat, jotka sisältävät merkkijonon h_2 ja voivat siten olla hahmon likimääräisiä esiintymiä. Koska tässä ei oteta huomioon osamerkkijonon h_2 sijaintia tekstissä ja hahmon likimääräisen esiintymän suurin mahdollinen pituus on $m + \lfloor s_e / \omega(P) \rfloor$, saadaan tutkittavaksi alueeksi *teksti*[*teksti*[q-(m+$\lfloor s_e/\omega(P) \rfloor$)+$|h_p|$…$q$+($m$+$\lfloor s_e/\omega(P) \rfloor$)], sillä se sisältää täsmälleen kaikki edellisenkaltaiset mahdolliset hahmon esiintymäehdokkaat. Kuvassa on esitetty eräät mahdolliset tämän alueen äärirajatapauksia vastaavat potentiaaliset hahmon esiintymät, joissa tummennettu osa kuvastaa osajonon h_2 täsmäävää kohtaa (vastaten täsmäysaskeleita minimaalisella editointipolulla), valkeat kohdat luvun $\lfloor s_e/\omega(P) \rfloor$ suuruista määrää poisto-operaatioita (vastaten poistoaskeleita minimaalisella editointipolulla) ja vaaleanharmaa tarkoittaa hahmon muiden osien (siis merkkijono h_2 poislukien) kanssa täsmääviä kohtia (vastaten täsmäysaskeleita minimaalisella editointipolulla).



Kuva 2.3.8:
Tilanne, jossa hahmo on jaettu neljään erilliseen osamerkkijonoon h_1, …, h_4 ja tekstistä löytyy merkkiin *teksti*[q] päättyvä merkkijonon h_3 esiintymä. Jos tällöin aiemmin tutkittavaksi todettu alue päättyy merkkiin *teksti*[r] ja $r < q$-($|h_p|$+m+$\lfloor s_e/\omega(P) \rfloor$)+1, niin nykyinen tai mikään sen jälkeen löydettävä hahmon osajono ei enää laajenna tätä aiempaa aluetta. Siten se voidaan jo tutkia kokonaisuudessaan tekemättä päällekkäistä työtä eli tutkimatta mitään tekstinkohtaa useammin kuin kerran.

Algoritmi 2.3.9 on muotoilemani esitys tästä Baeza-Yates-Perlbergin menetelmästä. Se toteuttaa ne toimenpiteet, jotka tulee suorittaa aina kun tekstistä on löydetty osamerkkijono h_p. Tässä ei siis oteta kantaa siihen, miten tämä merkkijono h_p löydettiin. Käytännössä yksi vaihtoehto olisi esimerkiksi korvata merkkijonoja h_1, h_2, …, h_z etsivän Aho-Corasick algoritmin ”loytyi”-funktio, eli jonkin etsittävän merkkijonon esiintymän löytymisen jälkeen tehtävät toimenpiteet, tällä algoritmilla. Kyseessä on siis funktio-tyyppinen toteutus, jota kutsutaan aina, kun tekstistä on löydetty jokin osamerkkijono h_p.

```
int j;                  // Tekstiä läpikäyvä indeksi, jonka oletetaan tässä aina sisältävän sen tekstin
                        // indeksin, johon päättyvä osamerkkijono h_p on löydetty kun tämä
                        // algoritmi suoritetaan.
int m;                                                  // Hahmon pituus.
int n;                                                  // Tekstin pituus.
int q;                  // Karsinnassa käytettyjen tasapituisten q-grammien pituus, eli aina |h_p| = q.
int wl, wp, wk;                         // Lisäys-, poisto- ja korvausoperaatioiden kustannukset.
int alku;                  // Kaksi globaalia muuttujaa, jotka ilmoittavat aiemmin tutkittavaksi
int loppu;                 // todetun, mutta toistaiseksi vielä tutkimattoman tekstinosan alku- ja
                           // loppuindeksit. Oletuksena on , että näiden arvot eivät voi muuttua
                           // tämän algoritmin ulkopuolella, ja että ne on alustettu arvoilla n
                           // ensimmäisen osamerkkijonon löytämistä.
int se;                 // Suurin sallittu likimääräisen esiintymän etäisyys, oletetaan tunnetuksi.

byp(j)                     // Funktio byp, jota kutsutaan parametrilla j, kun merkkiin teksti[j]
{                          // päättyvä osamerkkijono on löydetty.
    if(loppu < j-(q+m+se/wp)+1)                 // Voidaanko aiemmin tutkittavaksi todettu alue
    {                                           // jo tutkia (kuva2.3.8)?
        tutki(alku, loppu);                     // Tutkitaan alue teksti[alku…loppu] jollain
                                                // likimääräisellä etsintäalgoritmilla.
        alku = j-(q+m+se/wp)+1;                 // Tämä osamerkkijono määrittää seuraavaksi
                                                // tutkimisvuorossa olevan alueen alkurajan.
    }
    if(alku > loppu)                    // Onko kyseessä ensimmäinen löydetty osamerkkijono, eli
    {                                   // päteekö yhä alku = n?
        alku = j-(q+m+se/wp)+1;                 // Asetetaan tutkittavan alueen alkuraja
        if(alku < 0)                            // löydetyn osamerkkijonon mukaisesti, ja
        {                                       // tarkistetaan ettei tämä alue menisi tekstin
            alku = 0;                           // ensimmäisen merkin ohi vasemmalle.
        }
    }
   loppu = j+m+se/wp-q;                 // Asetetaan toistaiseksi tutkittavaksi todetun alueen
                                        // loppuraja (kuva 2.3.7).
    if(loppu >= n)                          // Mennäänkö jo tekstin viimeisen merkin yli?
    {
        loppu = n-1;                    // Asetetaan loppu = n - 1, jotta pysytään tekstin rajoissa.
    }
}
```

Algoritmi 2.3.9:
Baeza-Yates-Perlebergin karsintamenetelmä.

Algoritmin 2.3.9 soveltamisessa tulee ottaa huomioon, että jotta viimeisen tutkittavaksi todetun alueen tutkimiseksi pitää koko tekstin karsimisen jälkeen vielä tutkia alue *teksti*[alku…loppu]. Algoritmin 2.3.9 aikavaativuus on O(1), mikäli likimääräisen etsintäalgoritmin työtä ei oteta huomioon, sillä se ei varsinaisesti kuulu itse karsintaan. Siten tämä karsintamenetelmä ei itsessään aiheuta suurta lisätyötä, vaan lopullinen aikavaativuus riippuu siitä, miten osamerkkijonojen h_1, h_2, …, h_z etsiminen toteutetaan ja mitä algoritmia tutkittavaksi todettujen alueiden tarkastamisessa käytetään. Koska likimääräinen etsintä on yleensä näistä toimenpiteistä sekä teoreettiselta että käytännön aikavaativuudeltaan suurempi, riippuu Baeza-Yates-Perlebergin karsinnan käytännön tehokkuus hyvin pitkälti siitä, kuinka suuri osa tekstistä joudutaan vielä tutkimaan karsinnan jälkeen. Tämä taas riippuu tekstistä sekä käytetystä virhemarginaalista s_e siten, että mitä suurempi aakkosto ja pienempi virhemarginaali on käytössä, sitä tehokkaammin karsinta toimii [Baeza-Yates and Perleberg, 1996]. Koska käytettävän q-grammin pituus pienenee hyvin nopeasti virhemarginaalin kasvaessa ja luonnollisesti q-grammien esiintymistodennäköisyys tekstissä on sitä suurempi, mitä lyhyempiä ne ovat, niin karsinnan teho häviää olemattomaksi suurilla virhemarginaaleilla [Sutinen, 1998].

Toinen karsintamenetelmä: Hahmossa esiintyvien merkkien laskenta

Toisen karsintamenetelmän ideana on päätellä tietyssä tekstinosassa esiintyvien merkkien jakauman perusteella, voiko hahmon likimääräinen esiintymä löytyä kyseisen osajonon alueelta virhemarginaalin s_e puitteissa. Ajatus on lauseen 2.3.5 suhteen tietyssä mielessä päinvastainen kuin Baeza-Yatesin ja Perlebergin menetelmässä: nyt jokainen tutkittava hahmon osamerkkijono h_1, h_2, …, h_z on yksi merkki (eli h_1 = *hahmo*[0], h_2 = *hahmo*[1], …, h_z = *hahmo*[m-1]). Tämä vastaa tilannetta, jossa osamerkkijonojen lukumäärä z on maksimaalinen ($z = m$). Tällöin sopivasti toimiva karsintasääntö saadaan jälleen lauseen 2.3.5 avulla, jonka mukaan tiedetään, että jos $h_1 = mj_1[0]$, $h_2 = mj_1[1]$, …, ja $h_{|mj_1|} = mj_1[|mj_1|-1]$, niin $E(mj_1, mj_2) \leq s_e$ vain, jos merkkijono mj_2 sisältää eksaktisti ja erillisinä vähintään $|mj_1| - \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\}\rfloor$ kappaletta merkeistä $mj_1[0]$, $mj_1[1]$, …, $mj_1[|mj_1|-1]$.
Olkoon nyt $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_\alpha\}$ käytössä oleva aakkosto. Otetaan käyttöön merkintä $n(mj, \lambda)$ tarkoittamaan merkin λ esiintymien lukumäärää merkkijonossa mj. Jos tällöin $n(mj_1, \lambda) = x$ ja $n(mj_2, \lambda) = y$, tiedetään, että merkkijono mj_2 sisältää eksaktisti ja erillisinä y kappaletta merkkiä λ. Koska x on merkin λ (erillisten) esiintymien lukumäärä merkkijonossa mj_1, voidaan näin ollen todeta, että merkkijono mj_2 sisältää täsmälleen $\min\{x, y\}$ sellaista merkin λ esiintymää, joiden voidaan katsoa sisältyvän eksaktisti ja erillisinä merkkijonoon mj_1. Koska $\min\{x, y\} = 0$, jos merkki λ ei esiinny lainkaan merkkijonossa mj_1, ja luonnollisesti kaikki merkkijonon mj_1 merkit kuuluvat käytettävään aakkostoon Λ, seuraa edellisestä, että summalauseke $\Sigma(\min\{n(mj_1, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha)$ kertoo, kuinka monta merkkien $mj_1[0]$, $mj_1[1]$, …, $mj_1[m_1-1]$ eksaktia ja erillistä esiintymää merkkijonossa mj_2 on. Näin on päädytty lauseen 2.3.10 tulokseen.

Lause 2.3.10:
Olkoot mj_1 ja mj_2 kaksi mielivaltaista ei-tyhjää merkkijonoa sekä ehto $|mj_1| > \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\}\rfloor$ voimassa. Tällöin ehto $E(mj_1, mj_2) \leq s_e$ voi päteä ainoastaan siinä tapauksessa, että $\sum(\min\{n(mj_1, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) \geq |mj_2| - \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\}\rfloor$.

Lause 2.3.10 on sisällöltään sama kuin Jokisen, Tarhion ja Ukkosen [Jokinen et al., 1996] esittämä alkuperäinen vastineensa, vaikka heidän esityksessään se olikin muodossa $\sum(\max\{(n(mj_1, \lambda_k)-n(mj_2, \lambda_k), 0\}, 1 \leq k \leq \alpha) \leq \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\}\rfloor$. Kyseessä on kuitenkin täsmälleen samalla tavalla toimiva ehto, sillä on yksinkertaista todeta, että $|mj_1| = \sum(\max\{(n(mj_1, \lambda_k)-n(mj_2, \lambda_k), 0\}, 1 \leq k \leq \alpha) + \sum(\min\{n(mj_1, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha)$.

Käytännössä suoraviivaisin tapa soveltaa lausetta 2.3.10 karsintaan lienee keskittyä tutkimaan kiinnitetyn pituisia tekstin osamerkkijonoja. Esimerkiksi Navarron esittämä toteutus [Navarro, 1997] toimii juuri tällä tavoin käyttäen hahmon pituista ”teksti-ikkunaa”. Tämä menettely perustuu lauseen 2.3.11 esittämään yksinkertaiseen havaintoon.

Lause 2.3.11:
Olkoot mj_1, mj_2 ja mj_3 sellaisia mielivaltaisia epätyhjiä merkkijonoja, että $E(mj_1, mj_2) \leq s_e$, $mj_1 \subseteq mj_3$ tai $mj_3 \subseteq mj_1$, ja $|mj_2| = |mj_3|$. Tällöin on voimassa ehto $\sum(\min\{n(mj_3, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) \geq |mj_2| - \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\} \rfloor$.

Todistus:
Kun $E(mj_1, mj_2) \leq s_e$, täytyy lauseen 2.3.10 perusteella ehdon $\sum(\min\{n(mj_1, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) \geq |mj_1| - \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\} \rfloor$ olla voimassa. Jos $mj_1 \subseteq mj_3$, niin selvästi lauseen väite on voimassa, sillä $\sum(\min\{n(mj_3, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) \geq \sum(\min\{n(mj_1, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) \geq |mj_2| - \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\} \rfloor$.
Tarkastellaan nyt tapausta $mj_3 \subseteq mj_1$. Olkoot mj_4 ja mj_5 sellaisia (mahdollisesti tyhjiä) merkkijonoja, että $mj_1 = mj_4 \cup mj_3 \cup mj_5$. Tällöin $\sum(\min\{n(mj_1, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) = \sum(\min\{n(mj_3, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) + \sum(\min\{n(mj_4 \cup mj_5, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha)$ ja $|mj_1| - |mj_3| = |mj_4| + |mj_5|$, joten selvästi $\sum(\min\{n(mj_3, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) = \sum(\min\{n(mj_1, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) - \sum(\min\{n(mj_4 \cup mj_5, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) \geq \sum(\min\{n(mj_1, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) - (|mj1| - |mj3|)$. Ottamalla huomioon yhtäsuuruus $|mj2| = |mj3|$ nähdään yhdistämällä edellinen alussa esitetyn ehdon kanssa, että $\sum(\min\{n(mj_3, \lambda_k), n(mj_2, \lambda_k)\}, 1 \leq k \leq \alpha) \geq |mj_1| - \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\} \rfloor - (|mj1| - |mj3|) = |mj_2| - \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\} \rfloor$. Siis lauseen ehto on voimassa myös tässä tapauksessa, ja lause on todettu päteväksi.

Lauseen 2.3.11 perusteella nähdään, että tekstistä täytyy tutkia likimääräisellä etsintäalgoritmilla tarkemmin ainoastaan sellaiset osajonot, jotka eivät sisällä yhtään sellaista osajonoa *teksti*[*j*-*m*+1…*j*], jolle ehto $\sum(\min\{n(teksti[j\text{-}m\text{+}1\ldots j], \lambda_k), n(hahmo, \lambda_k)\}, 1 \leq k \leq \alpha) < m - \lfloor s_e / \min\{\omega(P), \omega(L), \omega(K)\} \rfloor$ on voimassa. Nimittäin yksikään tällainen osajono *teksti*[*j*-*m*+1…*j*] ei sisälly hahmon likimääräiseen esiintymään eikä myöskään sisällä sellaista.

Käytännössä tämän menetelmän mukainen karsinta voidaan toteuttaa esimerkiksi algoritmin 2.3.12 esityksen mukaisesti, jossa yksittäiset tutkittaviksi todetut alueet yhdistetään yhdeksi suuremmaksi tutkittavaksi alueeksi samaan tapaan kuin Baeza-Yates-Perleberg-algoritmin yhteydessä.

```
int j;                          // Tekstiä läpikäyvä indeksi, jonka oletetaan tässä aina sisältävän sen tekstin
                                // indeksin, johon päättyvä osamerkkijono h_p on löydetty, kun tämä
                                // algoritmi suoritetaan.
int m;                          // Hahmon pituus.
int n;                          // Tekstin pituus, tässä oletetaan n ³ m tarkistamatta erikseen.
int a;                          // Käytössä olevan aakkoston merkkien lukumäärä.
int N(λ);                       // Aakkoston merkit yksikäsitteisesti lukuvälille 0…a-1 kuvaava funktio.
int n_hahmo[a];                 // Hahmon merkkien jakauma, n_hahmo[N( l)] = n(hahmo, l).
int n_teksti[a];                // Tekstinkohdan merkkien jakauma, kullakin indeksin j arvolla
                                // aina n_teksti[N( l)] = n(teksti[j-m+1…j], l).
int merkkisumma = 0;            // Sisältää aina kulloisenkin osan teksti[j-m+1…j] sekä hahmon
                                // välisen lauseen 2.3.11 mukaisen summalausekkeen arvon.
                                // Arvo alustetaan nollaksi.
int alku = n;                   // Kaksi muuttujaa, jotka ilmoittavat aiemmin tutkittavaksi
int loppu = n;                  // todetun mutta toistaiseksi vielä tutkimattoman tekstinosan
                                // alku- ja loppuindeksit. Alustetaan arvoilla n osoittamaan
                                // tyhjää aluetta (menee tekstin takarajan yli).
int apuri;                                          // Indeksoinnissa käytetty apumuuttuja.
int minwplk;                    // Arvo min{w(P), w(L), w(K)}, oletetaan etukäteen lasketuksi.

for(apuri = 0; apuri < a; apuri++)          // Alustetaan merkkijakaumataulukot nollaksi.
{
    n_hahmo[apuri] = 0;
    n_teksti[apuri] = 0;
}
for(apuri = 0; apuri < m; apuri++)          // Lasketaan hahmon ja ensimmäisen tekstinkohdan
{                                           // merkkijakaumat.
    n_hahmo[N(hahmo[apuri])]++;
    n_teksti[N(teksti[apuri])]++;
}
for(apuri = 0; apuri < a; apuri++)          // Lasketaan ensimmäistä tekstinkohtaa
{                                           // vastaava karsintakriteerin summan arvo.
    if(n_teksti[apuri] >= n_hahmo[apuri])
    {
        merkkisumma = merkkisumma + n_hahmo[apuri];
    }
    else
    {
        merkkisumma = merkkisumma + n_teksti[apuri];
    }
}
if(merkkisumma >= m – se / minwplk)         // Toteutuuko karsintakriteeri, eli voiko
{                                           // merkkijono teksti[0…m-1] sisältää
    alku = 0;                               // hahmon likimääräisen esiintymän?
    loppu = m – 1;
}
for(apuri = m; apuri < n; apuri++)          // Jatketaan eteenpäin tekstin loppuun.
{
    n_teksti[N(teksti[apuri – m])]--;       // Päivitetään hahmon pituisen teksti-ikkunan merkkijakauma.
```

Algoritmi 2.3.12 (jatkuu seuraavalla sivulla):
Karsinta käyttäen ”merkkien laskentaa”.

```
    if(n_teksti[N(teksti[apuri – m])] < n_hahmo[N(teksti[apuri – m])])
    {
        merkkisumma--;              // Päivitetään karsintakriteerin summalausekkeen arvoa tarvittaessa.
    }
    n_teksti[N(teksti[apuri])]++;
    if(n_teksti[N(teksti[apuri])] > n_hahmo[N(teksti[apuri])])
    {
        merkkisumma++;
    }
    if(loppu < j – m + 1)                       // Voidaanko aiemmin tutkittavaksi todettu alue
    {                                           // jo tutkia (sama periaate kuin kuvassa 2.3.8)?
        tutki(alku, loppu);                     // Tutkitaan alue teksti[alku…loppu] jollain
                                                // likimääräisellä etsintäalgoritmilla.
        alku = loppu = n;                   // Alustetaan alku ja loppu taas vastaamaan tyhjää.
    }
    if(merkkisumma >= m – minwplk)                      // Onko karsintakriteeri voimassa, eli
    {                                                   // lisätäänkö tämä teksti-ikkuna
        loppu = j;                                      // tutkittavaan alueeseen?
        if(alku == n)                       // Onko tämä tällä hetkellä laajennettavan alueen alku eli
        {                                   // vastaako alku tyhjää aluetta omaamalla arvon n?
            alku = j – pituus + 1;          // Asetetaan alku vastaamaan tämän teksti-ikkunan
        }                                   // alkua.
    }
}
```

Algoritmi 2.3.12 (jatkoa):
Karsinta käyttäen ”merkkien laskentaa”.

Jos aakkoston koko oletetaan suhteellisen pieneksi vakioksi tekstin pituuteen *n* nähden, on algoritmin 2.3.12 aikavaativuus selvästi O(*n*). Tässä pätee luonnollisesti sama kuin algoritmin 2.3.9 yhteydessä (tai yleensäkin karsinta-algoritmien yhteydessä), eli että varsinaisen koko likimääräisen etsinnän kokonaistehokkuus riippuu suuresti siitä, kuinka paljon karsinta pystyy pienentämään likimääräisellä etsintäalgoritmilla tutkittavan tekstin osuutta.

3. Eräs merkkijonotäsmäysalgoritmien sovellus

Käsittelen tässä merkkijonotäsmäyksen käytännön esimerkkinä erästä pientä projektia, jossa käsiteltiin DNA-sekvenssejä. Sovellusalueen kannalta on paikallaan esitellä aluksi lyhyesti aiheeseen liittyviä keskeisiä käsitteitä ja termejä. Seuraavien kuvausten lähteenä on käytetty kirjaa Perinnöllisyyslääketiede [Aula et al., 1998].

Geeni
Periytymisen yksikkö, perintötekijä, joka sijaitsee kromosomissa.

Eksoni
Geenin koodittava osa.

Introni
Geenin ei-koodittava osa.

Genomi
Yksilön kaikkien geenien yhdistelmä, joka jakautuu kromosomeihin.

Kromosomi
Moninkertaisesti kiertynyt DNA-paketti.

Nukleotidi
DNA:n yksittäinen rakenneyksikkö.

DNA
Tärkeä osa genomia koostuu DNA:sta (”DeoxyriboNucleid Acid”). DNA sisältää neljää eri nukleotidityyppiä, jotka ovat adeniini, tymiini, sytosiini ja guamiini. Perinnöllinen informaatio perustuu DNA:n nukleotidijärjestykseen.

DNA-sekvenssi
DNA:n sisältämän yhtenäisen nukleotidiketjun nukleotidijärjestys.

Oligonukleotidi
Lyhyehkö, yleensä 5-60 nukleotidin mittainen DNA-jakso.

3.1 Projektin kuvaus

Projektin päämääränä oli tutkia käytännön tasolla geenien tunnistamisessa apuvälineeksi sopivan oligonukleotidi-indeksin muodostamista kokonaiselle organismin genomille. Varsinainen käyttökohde tällaiselle indeksille tulisi olemaan tulevaisuudessa ihmisen genomi, mutta tässä projektissa käytettiin ikään kuin harjoitusaineistona ensimmäistä kokonaan selvitettyä elävän organismin genomia, joka on leipurin hiivan (Saccharomyces Cerevisiae) genomi. Tavoitteeksi asetettiin sellaisen kaikki genomin geenit kattavan oligonukleotidi-indeksin muodostaminen, että jokainen siihen sisältyvä oligonukleotidi sisältyy yksikäsitteisesti ainoastaan yhteen geeniin. Tällaista oligonukleotidi-indeksiä voidaan hyödyntää, kun halutaan tutkia, sisältääkö jokin biologinen näyte jonkin tietyn geenin. Tämä tapahtuu syntetisomalla kyseistä geeniä vastaavia indeksin oligonukleotideja DNA-sekvensseiksi ja antamalla niiden olla vuorovaikutuksessa biologisen näytteen kanssa. Jos näiden välillä tapahtuu sidosten muodostumista, voidaan epäillä, että näyte todella sisältää kyseisen geenin.

Käytännöllisistä sekä DNA:n biologisiin ominaisuuksiin liittyvistä syistä johtuen asetettiin indeksiin kelpuutettaville oligonukleotideille seuraavat rajoitukset:

1. Jokaisen oligonukleotidin pituus 25 nukleotidia.
2. Sijainti mieluiten geenin alkupuolella.
3. Sisältää korkeintaan 12 adenosiini- tai tymiini-nukleotidia.
4. Sisältää korkeintaan 10 sytosiini- tai guamiini-nukleotidia.
5. Mikään 8 nukleotidin ikkuna ei sisällä enempää kuin 6 adenosiini-nukleotidia.
6. Mikään 8 nukleotidin ikkuna ei sisällä enempää kuin 6 tymiini-nukleotidia.
7. Mikään 8 nukleotidin ikkuna ei sisällä enempää kuin 4 sytosiini-nukleotidia.
8. Mikään 8 nukleotidin ikkuna ei sisällä enempää kuin 4 guamiini-nukleotidia.
9. Sisältää korkeintaan 6 peräkkäistä adenosiini-nukleotidia.
10. Sisältää korkeintaan 6 peräkkäistä tymiini-nukleotidia.
11. Sisältää korkeintaan 5 peräkkäistä sytosiini-nukleotidia.
12. Sisältää korkeintaan 5 peräkkäistä guamiini-nukleotidia.
13. Oligonukleotidin käänteinen komplementtinukleotidi saa täsmätä oligonukleotidin alusta korkeintaan 6 nukleotidia.

Edellä mainitut vaatimukset tulevat lääketieteen puolelta (lähteinä mm. [Wodicka et al., 1997], [Löffert et al., 1998] ja [Lockhart et al., 1996]), eikä niiden taustalla olevia syitä ole oleellista selvittää tässä yhteydessä tämän enempää. Tietojenkäsittelyopin kannalta niitä pidetään tässä ennalta annettuina reunaehtoina.

Tavoitteena oli siis etsiä jokaisesta hiivan geenistä mahdollisimman monta sellaista ehtojen 1-13 mukaista oligonukleotidia, että kunkin tällaisen tiettyä geeniä vastaavan yksittäisen oligonukleotidin esiintymiä ei löydy minkään toisen geenin alueelta.

Projekti jakautui melko selvästi kolmeen erilliseen vaiheeseen, ja seuraan tässä käsittelyssä samaa jakoa. Ensimmäinen osuus oli aineiston keruu ja esiprosessointi sopivaan käsittelymuotoon, ja seuraavat vaiheet vastaavat oikeastaan lukuja 1 ja 2, sillä projektin alkuvaiheessa oligonukleotidin esiintymäksi laskettiin vain eksakti esiintymä, mutta myöhemmin tämä laajennettiin DNA:lle tyypillisten mutaatioiden yms. huomioimiseksi sisältämään myös oligonukleotidin likimääräiset esiintymät.

3.2 Aineisto ja sen esiprosessointi

Kuten aiemmin jo todettiin, oli projektissa käytettävä aineisto leipurin hiivan genomi. Sen sisältämät sekvenssit löytyvät vapaasti internetistä, mutta sellaisenaan aineiston sisältämiä geenejä ei voi vielä ruveta käsittelemään. Nimittäin hiivan genomissa on 16 kromosomia, ja aineisto on jaettu siten, että aina yksi kromosomi on omassa tiedostossaan. Koska geenit sijaitsevat aina melko hajanaisesti kromosomien sisällä, kannattaa ne jatkon kannalta aluksi erotella. Projektissa käytetyt kromosomitiedostot otettiin amerikkalaisen NCBI:n tietokannasta (National Center for Biotechnology Information, [NCBI]). Tiedostot jakautuivat kahteen osaan, otsakkeeseen ja itse sekvenssidataan. Otsakkeessa on eritelty kaikki kyseisen kromosomin sisältämät geenit siten, että jokaisesta kerrotaan muun muassa sen koodaavan nukleotidisekvenssin sijainti (nukleotidien järjestysnumerot) ja geenin koodaama proteiinitranslaatio. Seuraavassa on esimerkkinä pieni osa hiivan ensimmäisen kromosomin sekvenssin sisältävän tiedoston otsakkeesta:

```
CDS     complement(20762..21330)
        /partial
        /note="YM4987.10c, incomplete orf, overlaps YM7056.01c,
        len: > 188, CAI: 0.20, similar to SW:NC5R_YEAST P38626
        putative NADH-cytochrome b5 reductase, (44.0% identity  in
        182 aa overlap)"
        /codon_start=3
        /product="unknown"
        /db_xref="PID:g927537"
        /translation="NPISSKLESGYLDLVVKAYVDGKVSKYFAGLNSGDTVDFKGP
        IGTLNYEPNSSKHLGIVAGGSGITPVLQILNEIITVPEDLTKVSLLYANETENDIL
        LKDELDEMAEKYPHFQVHYVVHYPSDRWTGDVGYITKDQMNRYLPEYSEDNRLLID
        EMAEKYPHFQVHYVVHYPSDRWTGDVGYITKDQMNRYLPEYSEDNRLLICGPDGMN
        NLALQYAKELGWKVNSTRSSGDDQVFVF"
CDS     complement(join(22049..23361,23660..23684))
        /gene="TUB3"
        /note="YM7056.02c, TUB3 gene; len: 445, CAI: 0.25; PS00227
        Tubulin subunits alpha, beta, and gamma signature"
        /codon_start=1
        /product="Tub3p"
        /db_xref="PID:g805018"
        /db_xref="SWISS-PROT:P09734"
        /translation="MREVISINVGQAGCQIGNACWELYSLEHGIKEDGHLEDGLSK
        PKGGEEGFSTFFHETGYGKFVPRAIYVDLEPNVIDEVRTGRFKELFHPEQLINGKE
        DAANNYARGHYTVGREIVDEVEERIRKMADQCDGLQGFLFTHSLGGGTGSGLGSLL
        LENLSYEYGKKSKLEFAVYPAPQLSTSVVEPYNTVLTTHTTLEHADCTFMVDNEAI
        YDICKRNLGISRPSFSNLNGLIAQVISSVTASLRFDGSLNVDLNEFQTNLVPYPRI
        HFPLVSYAPILSKKRATHESNSVSEITNACFEPGNQMVKCDPTKGKYMANCLLYRG
        DVVRDVQRAVEQVKNKKTVQMVDWCPTGFKIGICYEPPSVIPSSELANVDRAVCML
        SNTTAIADAWKRIDQKFDLMYAKRAFVHWYVGEGMEEGEFTEAREDLAALERDYIE
```

VGADSYAEEF"

Lyhenne ”CDS” tarkoittaa, että järjestysnumeroiltaan sen perässä ilmoitettuihin yhteen tai useampaan lukuväliin kuuluvat kromosomin nukleotidit muodostavat koodaavan sekvenssin, jolloin siis yksi koodaava sekvenssi vastaa aina yhden geenin eksoneita. Yksinkertaisuuden vuoksi tämän projektin aikana geenin edustajaksi otettiin vain nämä sen eksonit, sillä tietokantaan on lisätty vasta myöhemmin tiedot myös geenien introneista. Tämä menettely ei aiheuta muutoksia varsinaisiin hakumenetelmiin eikä oleellisesti pienennä käsiteltävän aineiston kokoa. Koska projektin tavoitteena oli tutkia menetelmiä eikä tuottaa varsinaisesti käyttöön otettavaa dataa, ei tällä ole suurta käytännön merkitystä projektin kannalta. Kun jatkossa siis viitataan tämän projektin yhteydessä aineistossa olevaan geeniin, tarkoittaa se tosiasiassa aineistossa olevan geenin eksoneita. Nykyisellä NCBI:n aineistolla olisi helppo ottaa käyttöön kokonaiset geenialueet, sillä siinä kokonaisen geenin alue ilmoitetaan täsmälleen edeltävien sääntöjen mukaisesti, merkkijonon ”CDS” tilalla vain on merkkijono ”gene”.

Koodaavan sekvenssin sijainnin ilmoittavien lukuvälien yhteydessä voi esiintyä toinen tai molemmat lisämääreistä ”join” sekä ”complement”. Näiden määreiden tulkinta on hyvin luonnollinen, edellinen tarkoittaa usean oligonukleotidin yhdistettä ja jälkimmäinen oligonukleotidin komplementtia. Kaikki aineistossa esiintyneet koodaavan sekvenssin määrittävät lukuvälit kuuluivat johonkin seuraavista neljästä päätyypistä:

a) $a..b$
b) join($a_1..b_1$, $a_2..b_2$, ..., $a_r..b_r$)
c) complement(a..b)
d) complement(join($a_1..b_1$, $a_2..b_2$, ..., $a_r..b_r$)).

Käytetään tässä samantapaista taulukkomerkintää kuin merkkijonojenkin yhteydessä, eli sovitaan, että merkintä $kr_i[a]$ tarkoittaa i:nnessä kromosomissa järjestysnumeron a+1 omaavaa nukleotidia ja että merkintä $kr_i[a...b]$ tarkoittaa järjestyksessä nukleotideista $kr_i[a]$, $kr_i[a+1]$,…,$kr_i[b]$ muodostuvaa oligonukleotidia. Nyt siis esimerkiksi hiivan 13. kromosomitiedostossa esiintyvä rivi

```
CDS       577717..578391
```

viittaa oligonukleotidiin kr_{13}[577716…578390], rivi

```
CDS       complement(578950..583920)
```

oligonukleotidin kr_{13}[578949…583919] komplementtiin ja rivi

```
CDS       complement(join(665844..666932,667017..667043))
```

oligonukleotidin $kr_{13}[665843…666931] \cup kr_{13}[667016…667042]$ komplementtiin.

Tiedoston loppuosassa on itse kromosomisekvenssi, joka on yksinkertaisesti ajateltuna hyvin pitkä eri nukleotideja edustavista merkeistä ’a’, ’t’, ’c’ ja ’g’ muodostuva merkkijono. Seuraavassa on esimerkkinä osa hiivan ensimmäisen kromosomin tiedostoa, nukleotidit 147541 – 148440, eli toisin sanoen oligonukleotidi $kr_1[147540…148439]$. Rivin vasemmassa laidassa oleva numero kertoo aina kyseisen rivin ensimmäisen nukleotidin järjestysnumeron koko kromosomin sekvenssissä:

```
147541 tttaatacaa ctttggttac ataaaagtaa aatttataca cctcatttca ttatgtagat
147601 tcatatatag aataccaatt atgattgacc caatagccat caaaatcagt agttatataat
147661 acttgtcttt ctaggagcca tttgcatatt tctgatattt catgaagcga aagtacttca
147721 cgacacctag attgcaatct actcaatgtt atccctggat gaaatattat ttcgttaacg
147781 accatagtaa ctacctgctt ccatatgttt ggcctaatgg aaccagatcc attcaccat
147841 aaacgagaaa atggtttgcc cagtggaact ttgacagcag acttccttgc tgtatcaat
147901 tttgtctgag aattggcata tataatcaga gggggagtta atgttcgtat ttcaaatctc
147961 cttgaagtat acgttaaagg tcgaacattt ctcaccattg gaattacatc catatcaat
148021 agctctcccg aaatcaaatc aattaaaacc caagaggata tatcggacgg ctcttgattg
148081 ataacaatag cgtttccggc ctccaataat tcattaacct tacatctata ctgaaaagct
148141 acaccaaaat ctttataatt tcctctattt tccaaaatgt ctggtaaagt atcagtacat
148201 tcaagttttg agccatggag ataaatttgc ttttccttag ccatatccat gatgacgtta
148261 tctattgatt cgtttccaac gttcttcaac gcctctattt catttctagt ggtcgaagga
148321 ctttctatta atatggaccg gatcactgtg cgaatataat cgtcgctttg actcttgat
148381 aagtccttag tagaagcgga aatctttcta gtgtaagttt tttttaaaga aagatctct
```

Toteutin geenien erottelemisen siten, että genomi käsiteltiin järjestyksessä kromosomitiedosto kerrallaan, ja aina jokaista kromosomia kohti luotiin uusi tiedosto, joka sisältää järjestysnumerolla varustettuina kaikki kyseisen kromosomin sisältämät geenit. Tämä geenien numerointi toteutettiin juoksevasti ja kromosomien välillä nollaamatta, joten jokaisen geenin järjestysnumero identifioi sen koko genomissa. Menetelmän yksinkertaistamiseksi jaoin ensin jokaisen kromosomitiedoston kahdeksi erilliseksi tiedostoksi siten, että otsaketiedot tulivat toiseen ja varsinainen nukleotidisekvenssi toiseen. Geenien erottelu tapahtui tämän jälkeen hakemalla aina ensin otsaketiedostosta tiedot yhden koodaavan sekvenssin sijainnista, minkä jälkeen kyseinen sekvenssi luettiin sekvenssitiedostosta ja kirjoitettiin tulostiedostoon. Tätä menettelyä toistettiin jokaiselle kromosomitiedostolle, kunnes ne kaikki oli käyty kokonaan läpi.

Ainoa ongelma menetelmän toteutuksessa oli sekvenssien sijaintitietojen etsiminen. Tähän syntyi kuitenkin melko nopeasti sääntö, jonka mukaan jokaisen geenin koodaavan sekvenssin sijaintitiedot löytyvät täsmälleen sellaisesta otsaketiedoston kohdasta, jossa sitaattimerkkien ulkopuolella esiintyy merkkijono ”CDS” sekä sen perässä tyyppiä a), b), c) tai d) olevaa

dataa. Vaatimus sitaattien ulkopuolisuudesta syntyi siitä, että esimerkiksi hiivan 13. kromosomin otsakkeessa esiintyy seuraava harhauttava kohta:

```
CDS             complement(join(139063..140090,140184..140214))
                /note="len: 352, CAI: 0.18, possible spliced version
                of the CDS complement(7831..8685)"
```

Tämän jälkeen hahmottelin seuraavien askeleiden mukaisen algoritmin yksittäisten geenien etsimiseksi, missä sisennysten ja vahvennetulla tekstillä kirjoitettujen kommenttien tarkoituksena on selventää askeleiden hierarkiaa ja toimintaa:

Koko geenietsinnän alkupiste:

1. Alusta geenilaskuri nollaan.

Toistetaan, kunnes kaikki kromosomit on käsitelty:

2. Avaa seuraavaksi käsittelyvuorossa olevan kromosomin otsake- ja sekvenssitiedostot sekä luo kromosomille uusi geenitiedosto. Jos kaikki kromosomit on jo käsitelty, lopeta.

Toistetaan kunnes kaikki käsiteltävän kromosomin geenit on haettu:

3. Hae otsaketiedostosta seuraava sellainen merkkijono ”CDS”, joka ei ole heittomerkkien välissä. Jos tällaista ei enää löydy, sulje nykyiset tiedostot ja mene kohtaan 2.
4. Jos seuraava merkki on numero, siirry kohtaan 7. Jos seuraava merkki (välilyöntejä ei huomioida) ei ole numero, tutki löytyykö merkkijono ”complement(” tai ”join(”. Jos ei löytynyt kumpaakaan, on kyseessä virhe, joten lopeta ja ilmoita tapahtunut. Jos löytyi ”complement(”, kirjaa tämä muistiin ja siirry askeleeseen 5, ja jos löytyi ”join(”, kirjaa tämä muistiin ja siirry askeleeseen 6.
5. Jos seuraava merkki on numero, siirry kohtaan 7. Jos seuraava merkki ei ole numero, tutki löytyykö merkkijono ”join(”. Jos ei löytynyt, on kyseessä virhe, joten lopeta ja ilmoita tapahtunut. Jos löytyi, kirjaa tämä muistiin ja siirry kohtaan 6.

Toistetaan, kunnes kaikki kyseisen geenin muodostavat lukuvälit on luettu:

6. Jos seuraava merkki ei ole numero, on kyseessä virhe, joten ilmoita tapahtunut ja lopeta. Muussa tapauksessa siirry kohtaan 7.
7. Lue merkkejä niin kauan, kuin ne ovat numeroita, ja lisää lopuksi luettujen numeroiden muodostama luku muistiin. Siirry kohtaan 8.

8. Jos seuraavat 2 merkkiä eivät muodosta merkkijonoa ”..”, on kaksi mahdollisuutta: joko tämä sekvenssin osa on vain yhden merkin mittainen tai kyseessä on virhe. Koska ensimmäinen vaihtoehto ei ole kovin todennäköinen, ilmoita tapahtunut varmuuden vuoksi, lisää kohdassa 7 luettu luku muistiin toisen kerran peräkkäin ja mene kohtaan 10. Jos seuraavat 2 merkkiä muodostivat merkkijonon ”..”, tutki, onko seuraava merkki numero. Jos on, mene kohtaan 9, ja jos ei, on kyseessä virhe, joten ilmoita tapahtunut ja lopeta.
9. Lue merkkejä niin kauan, kuin ne ovat numeroita, ja lisää lopuksi luettujen numeroiden muodostama luku muistiin. Siirry kohtaan 10.
10. Jos viimeksi kohdassa 4 tai 5 löytyi merkkijono ”join”, tutki onko seuraava merkki ’,’ tai ’)’. Jos se on ’,’, mene kohtaan 6, ja jos se on ’)’, mene kohtaan 11. Jos se ei ole kumpikaan, on kyseessä virhe, joten ilmoita tapahtunut ja lopeta. Jos merkkijonoa ”join” ei oltu viimeksi löydetty kohdassa 4 tai 5, mene kohtaan 11.

Geenin osasekvenssien alku- ja loppukohdat on merkitty muistiin, siis luetaan niitä vastaavat osat sekvenssitiedostosta ja kirjoitetaan tulostiedostoon:

11. Käy läpi pareittain muistiin merkityt luvut siten, että aina jokaista lukuparia vastaava sekvenssi luetaan muistiin kromosomin sekvenssitiedostosta ja tämän jälkeen lukupari poistetaan muistista. Mene kohtaan 12.
12. Kirjoita tulostiedostoon tämänhetkinen geenin järjestysnumero ja kasvata sitä sen jälkeen yhdellä. Jos kohdassa 4 löytyi merkkijono komplementti, kirjoita kohdassa 11 luettujen sekvenssien yhdisteen komplementti tulostiedostoon, ja muussa tapauksessa kirjoita kohdassa 11 luettujen sekvenssien yhdiste sellaisenaan tulostiedostoon. Kirjoita lopuksi tiedostoon geenien erotusmerkiksi merkkijono ” * ” ja mene kohtaan 3.

Askeleissa 1 – 13 tutkitaan hyvin monessa kohtaa, onko kyseessä virhetilanne. Tämä johtui siitä, että halusin varmistaa, että algoritmi toimii oikein ja että aineisto vastaa kaikilta osin sen rakenteelle asetettamiani oletuksia. Tämä johtikin seuraavaa muotoa olevien erikoistapausten löytämiseen:

```
CDS             complement(<131314..131572)
                /note="len: 113, CAI: 0.13, incomplete ORF"
                /codon_start=1
                /db_xref="PID:g558403"
                /translation="MELILNSLISDDLTEEQKRLSLDFLQDILQSNTKDYESY
                FSSRAVPGSITEDIAEIDAELSALDRKIRKTLLDNTSQIIGNILENDDRAQLD
                DIAKSLEQLWELDTNINKAAD"
```

Askel 6 kirjaa tässä tapauksessa virheen lukiessaan lukuväliä edeltävän merkin ’<’. Vastaavien tapausten selvittämiseksi korvasin lopullisessa versiossa askelen 6 askeleella 6’:

6’. Lue merkkejä niin kauan kuin ne eivät ole numeroita, ja siirry kohtaan 7.

Toteutin algoritmissa esiintyvät merkkijonotäsmäykset mukaillen kohtien 1.3 ja 1.7 merkkijonotäsmäysautomaatteja, sillä ne on helppo ja nopea toteuttaa intuition pohjalta if-rakenteella, kun haettavat merkkijonot tunnetaan etukäteen, ne ovat melko lyhyitä ja niiden lukumäärä on pieni. Lähdetiedostojen koko oli yhteensä 21,3 miljoonaa tavua, josta otsaketiedostot veivät 6,1 miljoonaa tavua ja sekvenssitiedostot loput 15,2 miljoonaa tavua. Koska kyseessä oli yksinkertainen lineaarinen operaatio, meni siihen aikaa tavallisella mikrotietokoneella (Pentium II 233Mhz) n. 5 minuuttia. Tulostiedostojen koko oli yhteensä hieman yli 8,7 miljoonaa tavua, ja hiivan geenien lukumääräksi saatiin tämän aineiston perusteella 6154. Seuraavassa on esimerkkinä tulostiedostojen muodosta pieni osa ensimmäisen kromosomin tulostiedostoa, jossa on hiivan toinen geeni kokonaisuudessaan ja hieman kolmannen geenin alkua:

```
0002 atgttatctcttgtaaaaagaagtattcttcattcaataccaattactcgtcacattcttc
caatccaattaatattggttaaaatgaaccatgtgcaaatcagaaacataaaattatatcacttta
tttcatatggtttcatgcttacaaagcttactgtctttctctttaacttatttttctacaggctac
gaattctttgcaggcttactttactcatattatcattacctgtacaaatatatattaaagaaatcc
aaacaaaaatgcttgaaaagcatacagcttccgatacatcatgtatatag * 0003 gaaatgac
aggttacttttaccaccacaaacaagttcttacacgttcaggtttgctaaggtcgatgactctgc
aattctatcagtcggtggcgacgttgcattcggatgctgtgcacaagagcaacctccaa
```

3.3 Indeksiin soveltuvien oligonukleotidien etsiminen

Kun geenit oli eroteltu genomista, voitiin siirtyä varsinaiseen projektin tavoitteeseen eli 25 merkin pituisten, vain yhden geenin alueella esiintyvien oligonukleotidien etsintään. Käytännössä tätä ongelmaa voi lähestyä monella eri tavalla, mutta aineiston valtavasta koosta johtuen sopivan menetelmän valinta on tärkeää. Siksi etenin asiassa siten, että hahmottelin ensin erilaisia ratkaisutapoja paperilla ja vertasin sitten niitä suuntaa antavasti jonkinasteisten pahimpia ja parhaita tapauksia vastaavien analyysien avulla. Tämän pienimuotoisen ennakkoanalyysin jälkeen vertailin lupaavimpien menetelmien toimivuutta käytännössä varsinaisten testiajojen muodossa.

3.3.1 Eksakti vertailu

Seuraavat kolme erilaista lähestymistapaa liittyvät projektin alkuvaiheeseen, jossa tutkittiin vain oligonukleotidien eksakteja esiintymiä eli etsittiin sellaisia 25 merkin mittaisia sekvenssejä, joiden kaikki eksaktit esiintymät sijaitsevat yhden ja saman geenin sisällä.

Ensimmäinen, eikä kovin vakavasti harkittu, vaihtoehto oli seuraavia askeleita noudattava ”brute force”-vivahteinen lähestymistapa:

Menetelmä E.1:

1. Valitse jokin sellainen geeni, josta ei ole vielä löydetty halutunlaista 25 merkin pituista osasekvenssiä ja jonka kaikkia 25 merkin pituisia osasekvenssejä ei ole vielä tutkittu. Jos yhtään tällaista geeniä ei ole enää jäljellä, niin lopeta.
2. Valitse kyseisestä geenistä jokin mielellään sen alkuosassa sijaitseva toistaiseksi tutkimaton 25 merkin pituinen ehdokassekvenssi.
3. Tutki, täyttääkö valittu ehdokas luvun 3 alussa esitetyt ehdot 1 – 13. Jos täyttää, mene kohtaan 4, ja jos ei täytä, merkitse tämä ehdokas testatuksi ja mene kohtaan 2.
4. Tutki, esiintyykö valittu ehdokassekvenssi jossain muussa geenissä, käyttäen jotain eksaktia merkkijonontäsmäysalgoritmia. Jos sekvenssi löytyy jostain toisestakin geenistä, niin merkitse tämä ehdokas testatuksi ja mene kohtaan 2. Jos ei löydy, niin kyseessä on haetunlainen indeksiin sopiva oligonukleotidi, joten merkitse tämä sekvenssi testatuksi ja kirjaa se muistiin hyväksytyksi edustamaan tällä hetkellä tutkittavaa geeniä.
5. Hyppää kohtaan 1.

Tämän menettelyn kannalta pahin tapaus olisi sellainen, että jokaisen geenin kaikki 25 merkin pituiset sekvenssit hyväksyttäisiin askeleessa 3 ja löydettäisiin askeleessa 4 aivan viimeiseksi tutkittavan muun geenin aivan lopusta. Jos oletettaisiin, että kohdassa 4 käytetty merkkijonontäsmäysalgoritmi olisi lineaarinen ja tekisi yhden geenin tutkimisessa täsmälleen geenin sisältämien nukleotidien lukumäärän (= geenin pituuden) verran operaatioita, olisi edellisten askeleiden 1 – 5 tekemä työmäärä pahimmassa tapauksessa suuruusluokkaa (*kaikkien geenien sisältämien 25 merkin pituisten sekvenssien lukumäärä*) × ((*kaikkien geenien yhteispituus*) – (*yhden geenin pituus*)). Tämän tulon ensimmäistä tekijää voidaan arvioida hyvin kaikkien geenien pituuksien summalla, josta on vähennetty 24 × (*geenien lukumäärä*), ja vastaavasti jälkimmäistä tekijää voidaan arvioida vähentämällä geenien keskipituus kaikkien geenien pituuksien summasta. Näin toimittaessa päästään arvioon, että pahimmassa tapauksessa tämän menetelmän tarvitsee suorittaa yhteensä luokkaa $(8{,}55 \times 10^6) \times (8{,}7 \times 10^6) \approx 7{,}4 \times 10^{13}$ oleva määrä operaatiota.

Jos askel 3 jätetään huomioimatta, olisi menetelmän E.1 yhteydessä aikavaativuuden suhteen paras (ja projektin tavoitteen kannalta tuhoisa) tapaus sellainen, että jokainen koepätkä löytyisi jostain toisesta geenistä heti askeleen 4 alkuvaiheessa. Käytännössä tämä ei kuitenkaan tunnu todennäköiseltä, ja käytän mielestäni yhä hyvin optimistisena arviona parhaasta tapauksesta sellaista, jossa kustakin geenistä löytyy täsmälleen yksi askeleessa 4 kelpuutettava oligonukleotidi. Tällöin operaatioiden lukumäärä olisi vähintään luokkaa (*kaikkien geenien lukumäärä*) × ((*kaikkien geenien yhteispituus*) – (*yhden geenin pituus*)) ≈ $6154 \times (8{,}7 \times 10^6) = 5{,}4 \times 10^{10}$.

Toinen harkittu menetelmä oli luonnollinen askel edellisestä eli etsinnän rinnakkaistaminen. Päätin toteuttaa tämän rinnakkaistamisen aina yksittäisen geenin laajuudessa siten, että yksittäisten ehdokassekvenssien sijaan tutkittiin kaikki yksittäisen geenin sisältämät 25 merkin sekvenssit kerralla käyttäen apuna avainpuuta (kappale 1.7). Tämä toinen harkittu vaihtoehto oli seuraavien askeleiden mukainen:

Menetelmä E.2:

1. Valitse jokin sellainen geeni, jota ei ole vielä käsitelty. Jos yhtään tällaista geeniä ei ole enää jäljellä, niin lopeta.
2. Muodosta kaikista valitun geenin sisältämistä luvun 3 alussa esitetyt ehdot 1 – 13 täyttävistä 25 merkin sekvensseistä koostuva avainpuu.

3. Käy avainpuun avulla läpi kaikki muut geenit etsien jokaisesta niistä kaikki avainpuuhun kuuluvien 25 merkin sekvenssien esiintymät, ja merkitse aina jokainen löydetty sekvenssi epäkelvoksi.
4. Kirjaa muistiin tämän geenin osalta hyväksytyiksi kaikki sellaiset avainpuun sekvenssit, joita ei merkitty epäkelvoksi kohdassa 3, sekä merkitse tämä geeni käsitellyksi.
5. Hyppää kohtaan 1.

Tässä lähestymistavassa algoritmin aika-arvio riippuu siitä, millä tavalla avainpuuta sovelletaan askeleessa 3. Käytettäessä algoritmin 1.7.4 kaltaista "brute force"-hakua ja oletettaessa jälleen, että askeleessa ei karsita mitään sekvenssiä pois, saadaan operaatioiden määrän arvioksi pahimmassa tapauksessa (*avainpuiden muodostaminen*) + (*avainpuuhaut*) ≈ $25 \times$ (*geenien yhteispituus*) + 25 ´ (*geenien lukumäärä*) $\times$ ((*geenien pituuksien summa*) – (*geenien keskipituus*)) ≈ $25 \times 8{,}7 \times 10^6 + 25 \times 6154 \times 8{,}7 \times 10^6 \approx 1{,}3 \times 10^{12}$. Koska edellä oleelliset muutokset aikavaativuuteen voisivat tulla ainoastaan askeleen 2 yhteydessä, tyydyin tässä pelkkään pahimman tapauksen analyysiin.

Jos askeleessa 3 käytetäänkin Aho-Corasick-algoritmia, tulee edelliseen analyysiin kaksi muutosta. Toisaalta puun avulla tapahtuva etsintä nopeutuu, kun hakuvaiheessa geeneistä ei enää tarvitse tutkia erikseen jokaista lomittaista sekvenssiä, mutta vastapainona Aho-Corasick-algoritmin korjausfunktion f_{ac} muodostaminen on hieman suuritöisempää. Koska kaikki etsittävät sekvenssit ovat tässä tapauksessa tasamittaisia, ei hahmo-osoittimia tarvita. Kumpaakin näistä muutoksista on vaikea arvioida tarkkaan paperilla, mutta päädyin käyttämään karkeana arvaukseen perustavana arviona, että Aho-Corasick-hakupuun muodostaminen veisi kaksinkertaisen määrän operaatioita verrattuna avainpuun muodostamisessa käytettyyn työmäärään ja että etsintävaihe nopeutuisi suunnilleen viisinkertaisesti. Näin aika-arvioksi tuli (*Aho-Corasick-hakupuiden muodostaminen*) + (*hakupuuhaut*) ≈ $2 \times 25 \times 8{,}7 \times 10^6 + 5 \times 6154 \times 8{,}7 \times 10^6 \approx 2{,}7 \times 10^{11}$.

Edellisten arvioiden mukaan tämä toinen menetelmä olisi pahimmassa tapauksessa jo suhteellisen lähellä ensimmäisen menetelmän parasta käytännössä mahdollista tapausta.

Kolmas hahmottelemani lähestymistapa jatkoi samaa suuntausta eli rinnakkaistamista. Nyt ajatuksena oli laajentaa se koskemaan yhden geenin sijaan kaikkia geenejä kerrallaan. Tämä menetelmä pohjautui myös avainpuun käyttöön mutta eri periaatteella kuin menetelmässä 2. Nyt ajatuksena oli verrata kollektiivisesti kaikkia eri geenien sisältämiä 25 merkin pituisia

sekvenssejä keskenään muodostamalla niistä avainpuu. Tällöin lisättäessä puuhun aina seuraavaa sekvenssiä tiedetään, että jos kyseinen sekvenssi aiheuttaa uuden lehtisolmun luomisen puuhun, on sekvenssi tähänastisten puussa olevien sekvenssien joukossa ainutlaatuinen. Vastaavasti, jos päädytään puussa jo olemassaolevaan lehtisolmuun, voidaan tähän kyseiseen lehtisolmuun päättyvä sekvenssi todeta epäkelvoksi indeksiin. Jos missään vaiheessa sekvenssiä ei todeta tällä tavalla epäkelvoksi, on se koko aineiston suhteen ainutlaatuinen ja siis sopiva indeksioligonukleotidiksi.

Yksi poikkeus tässä vielä tosin on, eli tapaus, jossa jokin sekvenssi esiintyy useaan kertaan yhden geenin sisällä. Mutta jos aina uutta lehtisolmua luotaessa merkitään muistiin, minkä geenin sekvenssin lisäys aiheuttaa sen luomisen, on tämä helppo tarkistaa. Jos lisäksi ylläpidetään listaa kaikista puun lehtisolmuista, voidaan puun muodostamisen jälkeen käydä helposti läpi kaikki lehtisolmut ja tutkia, mitkä niistä edustavat indeksin kannalta kelvollisia oligonukleotideja. Koska jokaiseen lehtisolmuun on merkitty, minkä geenin sekvenssistä on kyse, on nämä edustajasekvenssit helppo kirjata muistiin esimerkiksi jokaiselle geenille erikseen luotavaan tulostiedostoon.

Tässä projektin vaiheessa käyttämässäni tietokoneessa oli 96 megatavua muistia, ja siksi koko aineistolle ei voitu muodostaa avainpuuta muistirajoituksen vuoksi. Mutta soveltamalla Karp-Rabin-algoritmin sormenjälkiperiaatetta voidaan ongelma jakaa pienempiin alitapauksiin: jaetaan ensin kaikki aineiston sisältämät 25 merkin pituiset sekvenssit osiin niiden sormenjälkien mukaan, ja muodostetaan avainpuut jokaista eri sormenjälkeä vastaavalle (ja siis alkuperäistä joukkoa pienemmälle) joukolle erikseen. Myös tämä toimii selvästi oikein etsien kaikki ainutlaatuiset sekvenssit, sillä mikään sekvenssi ei voi omata kahta eri sormenjälkeä, ja siten kaksi keskenään samanlaista sekvenssiä kuuluvat aina samaa sormenjälkeä edustavaan joukkoon.

Menetelmä E.3:

1. Jaa kaikki aineiston geenien sisältämät 25 merkin sekvenssit sopiviin sormenjälkijoukkoihin siten, että mukana on vain luvun 3 alussa esitetyt kriteerien 1 - 13 täyttävät sekvenssit.
2. Valitse jokin toistaiseksi käsittelemätön sormenjälkijoukko. Jos yhtään tällaista ei enää ole jäljellä, niin lopeta.
3. Lisää yksi kerrallaan kaikki kyseisen sormenjälkijoukon sekvenssit avainpuuhun siten, että aina sellainen sekvenssi merkitään epäkelvoksi, joka esiintyy jo aiemmin puussa ja on peräisin eri geenistä. Jos tämän yhteydessä lisätään puuhun uusi lehtisolmu, niin

merkitse kyseisen lisättävän sekvenssin sisältäneen geenin numero sitä vastaavaan lehtisolmuun, ja lisää listaan viite kyseiseen lehtisolmuun.

4. Käy listan avulla läpi kaikki muodostetun avainpuun lehtisolmut siten, että jokaista sellaista lehtisolmua, jota ei ole merkitty epäkelvoksi, vastaava sekvenssi kirjataan muistiin lehtisolmuun merkityn geenin edustajaksi.

Arvioin tämän neljännen menetelmän suorittamien operaatioiden määrääksi (*aineiston jako sormenjälkijoukkoihin*) + (*sormenjälkijoukkojen lukumäärä*) × (*avainpuun luominen keskimäärin*) ≈ 25 × (*kaikkien geenien pituuksien summa*) + (*sormenjälkijoukkojen lukumäärä*) × 25 × ((*kaikkien geenien pituuksien summa*) / (*sormenjälkijoukkojen lukumäärä*)) ≈ $25 \times 8{,}7 \times 10^6 + 25 \times 8{,}7 \times 10^6 \approx 4{,}35 \times 10^8$.
Tämä viimeinen menetelmä vaikutti siis ennakkoanalyysin perusteella kaikkein tehokkaimmalta ja vieläpä melko suurella erolla.

Kaikissa edellä esitetyissa operaatioarvioissa jätettiin löydettyjen indeksiin hyväksyttävien oligonukleotidien muistiin kirjaamisen aiheuttama työ pois. Tämä sen takia, että jokaisessa tapauksessa se on suoraan verrannollinen löydettyjen indeksioligonukleotidien lukumäärään eli lopputuloksen hyvyyteen, ja toisaalta keskenään yhtä hyvään tulokseen päätyessään ei menetelmien 1 – 4 välillä ole kirjaamisessa tehdyn työn välillä käytännössä eroa. Siis menetelmän valinnassa voidaan perustellusti keskittyä itse etsintäprosessin analysointiin.

Eksaktien menetelmien vertailua käytännössä

Ennen menetelmien E.1 - E.3 käytännön vertailua toteutin sellaisen aineistoa tutkivan koeajon, joka tutkii, kuinka moni aineistossa sijaitseva 25 merkin pituinen sekvenssi täyttää luvun 3 alussa esitetyt kriteerit 1 - 13. Ennakkoanalyysissä olin olettanut, että tämä lukumäärä on hyvin suuri, ja se osoittautui oikeaksi olettamukseksi. Nimittäin, kun koko aineistossa oli hieman yli 8,6 miljoonaa 25 merkin pituista sekvenssiä, oli näistä ainoastaan n. 1600, eli hieman alle kaksi promillea sellaisia, jotka eivät täyttäneet ehtoja 1 - 13.

Kaikki seuraavassa esitettävät koeajot suoritettiin 600 Mhz Pentium III prosessorilla ja 512 megatavun keskusmuistilla varustetulla mikrotietokoneella. Koeajoissa pyrittiin keskittymään mahdollisimman tarkasti itse käytettyjen menetelmien suoritusaikojen vaativuuteen, ja sen vuoksi ne kaikki toteutettiin hyvin samantapaisen ohjelmakehyksen sisällä, joka luki ensin koko aineiston muistiin, ja suoritusaikaa alettiin laskea aina vasta, kun tämä pohjatoimenpide oli tehty. Kokeilin aluksi myös sellaisia menettelytapoja, joissa aineistoa luetaan vähän kerrallaan levyltä (esimerkiksi geeni kerrallaan), mutta tämä oli hyvin hidasta verrattuna koko aineiston kerrallaan lukemiseen, sillä jokaiseen yksittäiseen levyasemahakuun menee suhteessa huomattavasti enemmän aikaa kuin suoraan muistihakuun.

Itse projektin aikana en suorittanut käytännön kokeita menetelmän E.1 suoritusajan arvioimiseksi, sillä se tuntui ilmiselvästi liian suurelta. Toteutin kuitenkin jälkikäteen vertailun vuoksi sellaisen koeajojoukon, jossa askeleessa 4 kokeiltiin jokaista tämän tekstin ensimmäisessä osassa esiteltyä eksaktia yksittäisen hahmon etsimisalgoritmia. Koeajoissa tutkitut pätkät valittiin satunnaisesti etukäteen siten, että jokainen eri kokeiltu algoritmi käytti täsmälleen samaa testijoukkoa. Taulukko 3.1 esittää koeajojen tulokset. Sen perusteella voi arvioida, että pahimmassa tapauksessa kyseisistä etsintäalgoritmeista nopeimmalla kaikkien 25 merkin sekvenssien tutkiminen veisi $(8{,}55 \times 10^6) \times (254/1000)$ sekuntia, joka on yli 24 vuorokautta. Siis ennakkoaavistelu liian pitkästä suoritusajasta menetelmän E.1 tapauksessa osui oikeaan.

Etsintäalgoritmi:	Suoritusaika (sekuntia): 1000 sekvenssiä	Aika-arvio koko genomin käsittelemiseksi (tuntia):
"Brute force"	517	1235
Knuth-Morris-Pratt	636	1519
Boyer-Moore	254	607
Boyer-Moore-Horspool	299	714
Quick Search	319	762
Karp-Rabin	1917	4580

Taulukko 3.1:
Koeajo, jossa etsittiin tuhannen leipurin hiivan genomista satunnaisesti valitun 25 merkin pituisen sekvenssin esiintymiä genomin muista geeneistä käyttäen eksaktia etsintäalgoritmia erikseen jokaisen sekvenssin kohdalla.

Taulukossa 3.1 voidaan kiinnittää huomiota kahteen seikkaan: Ensinnäkin Karp-Rabin-algoritmi on tässä tapauksessa yli kolme kertaa hitaampi kuin muut koeajossa mukana olleet algoritmit. Tämä ei ole toisaalta yllättävää, sillä siinä tehdyt sormenjälkilaskut omaavat melko suuren vakiokertoimen vaikka onnistuvatkin asymptoottisesti ajatellen lineaarisessa ajassa. Toinen huomiota herättävä seikka on Knuth-Morris-Pratt-algoritmin hitaus jopa "brute-force"-algoritmiin verrattuna. Ilmeisesti tässä tapauksessa etsittävien merkkijonojen alkuosia täsmättiin keskimäärin niin vähän, että "brute-force"-algoritmi ei tehnyt kovinkaan suurta ylimääräistä työtä, ja Knuth-Morris-Pratt-algoritmi ei saanut tehtyä keskimäärin kovinkaan pitkiä hahmon siirtymiä.

Menetelmän E.2 yhteydessä käytettiin samaa satunnaisesti etukäteen valittua sekvenssijoukkoa kuin menetelmän E.1 yhteydessä, mutta nyt yksittäisen koepätkän sijaan tutkittiin samalla se geeni kokonaan, jossa kyseinen ehdokassekvenssi sijaitsi. Lisäksi suuremman työmäärän takia testiaineistoa sovellettiin pienemmälle ehdokasmäärälle. Tämä testiajo antoi jo huomattavasti järkevämmän koko genomin käsittelemisessä kuluvan aika-arvion, joka oli melko tarkalleen neljä tuntia.

Etsintäalgoritmi:	Suoritusaika (sekuntia): 100 geeniä	Aika-arvio koko genomin käsittelemiseksi (tuntia):
"brute force"-avainpuu	693	12
Aho-Corasick	234	4

Taulukko 3.2:
Koeajo, jossa etsittiin sadan leipurin hiivan genomista satunnaisesti valitun geenin sisältämien 25 merkin pituisten sekvenssien esiintymiä genomin muista geeneistä käyttäen avainpuuta soveltavaa rinnakkaishakua.

Vaikka menetelmä E.3 olikin hieman eriluonteinen, sovelsin edellä käytettyä satunnaisesti muodostettua ehdokassekvenssiluetteloa myös sen yhteydessä. Valitsin nimittäin sormenjälkijoukkojen lukumäärän määrittäväksi alkuluvuksi arvon $p = 6151$, eli siis lähimpänä geenien lukumäärää sijaitsevan alkuluvun, ja käytin näitä koepätkien sijantigeenien numeroita niiden sormenjälkijoukkojen satunnaiseen valitsemiseen, joille koeajot suoritettiin. Tässä tavoitteena oli päästä koko genomin tutkimisen kannalta keskimäärin suhteelliselta aikavaativuudeltaan samankaltaiseen tilanteeseen kuin menetelmän E.2 arvioinnissa. Luonnollisesti tarkistin, että mikään koepätkä ei sijainnut geeneissä 6152, 6153 tai 6154, jotta edellinen menettely onnistuu suoraan. Sinänsä tällä sormenjälkijoukkojen valinnalla tosin ei luultavasti ollut mitään väliä, sillä aineistossa olevat 25 merkin sekvenssit jakautuivat hyvin tasaisesti eri sormenjälkijoukkoihin, eli ne kaikki olivat käytännössä samankokoisia ja siten odotetusti melko tasavertaisia avainpuun muodostamisessa kuluvan suoritusajan suhteen.

Seuraavassa taulukossa 3.3 on esitetty vertailun vuoksi kaksi erilaista menettelytapaa. Ensimmäisessä sormenjälkijoukot luotiin konkreettisesti eri tiedostoihin, jotka sitten käsiteltiin tiedosto kerrallaan, ja toisessa muodostettiin vain jokaista eri sormenjälkeä kohti sellainen lista, joka sisältää jokaisen kyseisen sormenjäljen omaavan 25 merkin pituisen sekvenssin sijaintikohdan koko aineistossa. Tämän jälkimmäisen menettelyn ylivertainen nopeus edelliseen verrattuna demonstroi hyvin sitä, kuinka tärkeää on välttää liiallista tiedostojen käsittelyä, jos on mahdollista pitää aineisto muistissa. Nyt päästiin jo erittäin hyvään suoritusaika-arvioon, jonka mukaan koko aineiston käsittelyyn menisi suunnilleen 10 minuuttia. Siis tämä kolmas menetelmä oli ylivoimaisesti nopein, mikä olikin odotettua, sillä se on tietyssä mielessä optimaalinen. Nimittäin, jos jätetään huomioimatta sinänsä hyvin yksinkertainen ja nopea sormenjälkijoukkojen muodostamisoperaatio (sormenjälkilistoja käytettäessä aikaa meni 10-20 sekuntia), tutkii menetelmä E.3 jokaisen ehdokassekvenssin ainoastaan täsmälleen kerran.

Etsintäalgoritmi:	**Suoritusaika (sekuntia): 100 sormenjälkeä**	**Aika-arvio koko genomin käsittelemiseksi (tuntia):**
Sormenjälki-avainpuut, tiedostot	163	2,8
Sormenjälki-avainpuut, muisti	11	0,19

Taulukko 3.3:

Koeajo, jossa tutkittiin avainpuun avulla sadan leipurin hiivan genomista satunnaisesti valitun sormenjäljen osalta, mitkä aina tietyn sormenjäljen omaavat 25 merkin pituiset sekvenssit esiintyvät useammassa kuin yhdessä geenissä. Sormenjäljet määritettiin alkuluvun 6151 suhteen.

Ilmiselvästi siis sopivin menetelmä oli aineiston jakaminen eri sormenjälkiin, jotka sitten tutkitaan avainpuun avulla. Toteutin koko aineiston tutkimisen tällä menetelmällä siten, että

etsintävaiheessa muodostettiin jokaista geeniä kohden oma lista siitä löytyvien sopivien oligonukleotidien sijaintikohdista, ja sitten lopuksi nämä listat käytiin yksitellen läpi kirjoittaen kyseistä geeniä vastaavaan tulostiedostoon nämä sopivat sekvenssit. Tässä ajatuksena oli jälleen minimoida tarvittavat tiedosto-operaatiot, sillä kuten edellä jo havaittiin, voivat ne yksistään viedä jo hyvinkin merkittävän ajan itse varsinaiseen etsintä-vaiheen suoritusaikaan nähden. Kokonaisuudessaan aineistossa olevista ehdokassekvensseistä todettiin epäkelvoiksi ainoastaan n. 150000 kappaletta eli alle kaksi prosenttia, ja toimenpiteeseen kulunut suoritusaika oli 523 sekuntia eli jopa odotettua hieman pienempi.

3.3.2 Likimääräinen vertailu

Projektin loppuvaiheessa tiukennettiin etsittäville oligonukleotideille asetettuja vaatimuksia siten, että nyt kelpuutettiin vain sellaiset 25 merkin pituiset sekvenssit, joiden kaikki likimääräiset esiintymät sijaitsevat yhden ja saman geenin alueella. Likimääräisen esiintymän mittapuuna käytettiin osassa 2 käsiteltyä editointietäisyyttä suurimman sallitun virheen s_e ollessa 4, ja käytettäessä editointioperaatioiden kustannuksina arvoja $\omega(P) = \omega(K) = \omega(L) = 1$. Motivaatio tähän kriteerin tiukennukseen tuli mikrobiologian puolelta, sillä käytännössä DNA-sekvensseissä esiintyy runsaasti mutaatioita ja muuta vaihtelua, joten tämän huomioimiseksi tulisi tiettyä geeniä indeksissä edustavan sekvenssin olla riittävän erilainen kaikkiin muissa geeneissä sijaitseviin sekvensseihin verrattuna.

Toimin tätä likimääräistä tapausta tutkiessani samalla tavalla kuin aiemminkin, eli aloitin hahmottelemalla ja analysoimalla erilaisia mahdollisia lähestymistapoja ongelman ratkaisemiseksi. Näistä ensimmäinen oli jälleen suoraviivainen ”brute force”-menetelmä, joka on melkein identtinen ensimmäisen eksaktissa tapauksessa hahmotellun menetelmän kanssa.

Menetelmä L.1:

1. Valitse jokin sellainen geeni, josta ei ole vielä löydetty halutunlaista 25 merkin pituista osasekvenssiä ja jonka kaikkia 25 merkin pituisia osasekvenssejä ei ole vielä tutkittu. Jos yhtään tällaista geeniä ei ole enää jäljellä, niin lopeta.
2. Valitse kyseisestä geenistä jokin mielellään sen alkuosassa sijaitseva toistaiseksi tutkimaton 25 merkin ehdokassekvenssi.
3. Tutki, täyttääkö valittu ehdokas luvun 3 alussa esitetyt ehdot 1 – 13. Jos täyttää, mene kohtaan 4, ja jos ei täytä, merkitse tämä ehdokas testatuksi ja mene kohtaan 2.
4. Tutki, esiintyykö valittu ehdokassekvenssi jossain muussa geenissä, käyttäen erikseen jokaisen muun geenin kohdalla jotain likimääräistä merkkijonontäsmäysalgoritmia. Jos sekvenssi löytyy jostain toisestakin geenistä, niin merkitse tämä ehdokas testatuksi ja mene kohtaan 2. Jos ei löydy, niin kyseessä on haetunlainen indeksiin sopiva oligonukleotidi, joten merkitse tämä sekvenssi testatuksi ja kirjaa se muistiin hyväksytyksi edustamaan tällä hetkellä tutkittavaa geeniä.
5. Hyppää kohtaan 1.

Tämä menetelmä poikkeaa eksaktista tapauksesta ainoastaan askeleessa 4 käytetyn merkkijonontäsmäysalgoritmin osalta, joka on siis tässä tapauksessa likimääräinen. Nyt aikavaativuuden arviointi menee vielä hankalammaksi, mutta päädyin ajatukseen, että

likimääräinen haku veisi vähintään 12 kertaa suuremman operaatiomäärän kuin lineaarinen eksakti haku. Arvion pohjana oli se, että editointitaulukossa tehdään kolme eri vertailua jokaista taulukon alkiota kohti ja alkioita joudutaan kokeellisten tulosten mukaan täyttämään keskimäärin lukuun s_e nähden verrannollinen määrä jokaista tekstin merkkiä kohden, mikäli käytetään kappaleessa 2.3.1 esitellyn Ukkosen algoritmin kaltaista etsintäalgoritmia. Siten tässä operaatioiden lukumäärän arvioksi saataisiin paperilla pahimmassa tapauksessa $12 \times 7{,}4 \times 10^{13} \approx 8{,}9 \times 10^{14}$ operaatiota ja parhaassa edes etäisesti realistiselta tuntuvassa tapauksessa $12 \times 5{,}4 \times 10^{10} \approx 6{,}5 \times 10^{11}$ operaatiota.

Eksaktissa tapauksessa luonnollinen askel haun nopeuttamiseksi oli sen rinnakkaistaminen. Likimääräisen vertailun yhteydessä tällainen menettely ei onnistu läheskään yhtä tehokkaasti, sillä, toisin kuin eksaktissa tapauksessa, likimääräisessä tapauksessa ei voida päätellä ehdokassekvenssien aiempien vertailujen perusteella kovinkaan hyvin mitään siitä, kuinka suurelta osin nämä jo tutkitut sekvenssit täsmäävät jonkin muun sekvenssin kanssa. Navarro [Navarro, 1998] esittää aiheesta yhteenvedon, jossa mainitaan kaksi todellisen rinnakkaishaun toteuttavaa algoritmia, mutta kumpikin niistä on liian rajoitettu tähän: Toisessa sallitaan ainoastaan tapaus $s_e = 1$ [Muth and Manber, 1996], ja toisessa kaikkien rinnakkaisesti etsittävien merkkijonojen yhteispituus voi olla korkeintaan käytettävän prosessorin sanan suuruinen [Wu and Manber, 1992] (esimerkiksi Pentium-sarjan prosessoreilla tämä raja on siis 32 merkkiä). Keskityin siksi tehostamaan etsintää kappaleessa 2.3.2 esitellyn karsintamenettelyn avulla. Ainoa muutos menetelmään L.1 nähden oli siten karsintavaiheen ottaminen askeleessa 4.

Menetelmä L.2:

1. Valitse jokin sellainen geeni, josta ei ole vielä löydetty halutunlaista 25 merkin pituista osasekvenssiä ja jonka kaikkia 25 merkin pituisia osasekvenssejä ei ole vielä tutkittu. Jos yhtään tällaista geeniä ei ole enää jäljellä, niin lopeta.
2. Valitse kyseisestä geenistä jokin mielellään sen alkuosassa sijaitseva toistaiseksi tutkimaton 25 merkin ehdokassekvenssi.
3. Tutki, täyttääkö valittu ehdokas luvun 3 alussa esitetyt ehdot 1 – 13. Jos täyttää, mene kohtaan 4, ja jos ei täytä, merkitse tämä ehdokas testatuksi ja mene kohtaan 2.
4. Tutki, esiintyykö valittu ehdokassekvenssi jossain muussa geenissä, käyttäen erikseen jokaisen muun geenin kohdalla karsintaa sekä jotain likimääräistä merkkijonontäsmäysalgoritmia. Jos sekvenssi löytyy jostain toisestakin geenistä, niin merkitse tämä ehdokas testatuksi ja mene kohtaan 2. Jos ei löydy, niin kyseessä on

haetunlainen indeksiin sopiva oligonukleotidi, joten merkitse tämä sekvenssi testatuksi ja kirjaa se muistiin hyväksytyksi edustamaan tällä hetkellä tutkittavaa geeniä.

5. Hyppää kohtaan 1.

Menetelmää L.2 ei voi etukäteen arvioida kovin hyvin, sillä sen suoritusaika riippuu täysin karsinnan tehokkuudesta, eli siitä, kuinka suuri osa aineistosta joudutaan tutkimaan likimääräisellä etsintäalgoritmilla karsinnan jälkeen. Siksi en arvioinut tätä vaihetta sen tarkemmin paperilla, vaan päätin suorittaa suoraan koeajot ja tehdä päätelmät niiden pohjalta. Koska käytännössä ei löytynyt mahdollisuutta suorittaa varsinaista likimääräistä etsintää millään muulla tavalla kuin yksitellen erikseen jokaiselle eri ehdokassekvenssille, oli ainoa tie eteenpäin pyrkiä pienentämään tämä suoran etsinnän tarve minimiin. Siksi hahmottelin tämän toisen vaihtoehdon yhteydessä mahdollisuuksia soveltaa molempia kappaleessa 2.3.2 esitettyjä karsintamenetelmiä samanaikaisesti sekä tiukensin Baeza-Yates-Perlebergin menetelmän karsintakriteeriä ottamaan huomioon myös siinä etsittävän eksaktin osamerkkijonon sijainnin hahmossa. Nimittäin hahmoon voidaan sallitun virheen rajoissa kohdistaa korkeintaan s_e / min{ω(P), ω(L)} sellaista editointioperaatiota, joka vaikuttaa hahmon sisältämien merkkijonojen sijaintikohtiin, ja lisäksi jokainen operaatio voi vaikuttaa sijaintiin korkeintaan yhden merkin verran. Ottamalla huomioon tämä vaatimus huomataan, että aina tekstin indeksiin q päättyvän osamerkkijonon h_p löytymisen seurauksena riittää tutkia alueen *teksti*[q-(m+|h_p|+s_e/ω(P))+1…q+m-|h_p|+s_e/ω(P)] sijaan alue *teksti*[q-(m+|h_p|+s_e/ω(P))+1…q+m-|h_p|+s_e/ω(P)] (kuva 3.4). Koska tämä havainto on hyvin yksinkertainen, ihmettelin hieman, että en ollut löytää kirjallisuudesta mainintaa sen soveltamisesta. Lopulta kuitenkin Navarron [Navarro, 1998] väitöskirjasta löytyi viittaus Holstin ja Sutisen tekstiin [Holsti and Sutinen, 1994], jossa ajatusta hyödynnetään.

Karsinnan tehostamisen ohella keskityin myös karsinnan nopeuttamiseen. Kolmas menetelmä toimikin tällä tavalla, eli ajatuksena oli pyrkiä välttämään aineiston lineaarista hakua jokaisen karsinnan yhteydessä ja käyttää sen sijaan alussa muodostettua sopivaa indeksirakennetta, johon on kerätty karsinnassa tarvittavaa tietoa tekstistä.

Menetelmä L.3:

1. Muodosta tekstistä karsinnassa apuna käytettävä indeksi.
2. Valitse jokin sellainen geeni, josta ei ole vielä löydetty halutunlaista 25 merkin pituista osasekvenssiä ja jonka kaikkia 25 merkin pituisia osasekvenssejä ei ole vielä tutkittu. Jos yhtään tällaista geeniä ei ole enää jäljellä, niin lopeta.

3. Valitse kyseisestä geenistä jokin mielellään sen alkuosassa sijaitseva toistaiseksi tutkimaton 25 merkin ehdokassekvenssi.
4. Tutki, täyttääkö valittu ehdokas luvun 3 alussa esitetyt ehdot 1 – 13. Jos täyttää, mene kohtaan 4, ja jos ei täytä, merkitse tämä ehdokas testatuksi ja mene kohtaan 2.
5. Tutki, esiintyykö valittu ehdokassekvenssi jossain muussa geenissä, käyttäen erikseen jokaisen muun geenin kohdalla indeksin avulla toteutettavaa karsintaa sekä jotain likimääräistä merkkijonontäsmäysalgoritmia. Jos sekvenssi löytyy jostain toisestakin geenistä, niin merkitse tämä ehdokas testatuksi ja mene kohtaan 2. Jos ei löydy, niin kyseessä on haetunlainen indeksiin sopiva oligonukleotidi, joten merkitse tämä sekvenssi testatuksi ja kirjaa se muistiin hyväksytyksi edustamaan tällä hetkellä tutkittavaa geeniä.
6. Hyppää kohtaan 1.

Menetelmässä L.3 ainoa ero menetelmään L.2 nähden on jonkinlaisen indeksin käyttö karsinnassa, joten myös tämän kolmannen menetelmän suoritusaikaa on hyvin vaikea arvioida paperilla. Siis tyydyin tässäkin kokeilemaan menetelmää käytännössä ja tekemään päätelmät vasta niiden pohjalta.

Likimääräisten menetelmien vertailua käytännössä

Käytin likimääräisten menetelmien koeajoissa täsmälleen samoja satunnaisesti valittuja ehdokassekvenssejä kuin eksaktissakin tapauksessa. Esimerkiksi menetelmien E.1 ja L.1 koeajojen ainoa ero olikin itse asiassa hyvin vähäinen muutos kehysohjelmaan, jossa eksakti etsintäalgoritmi korvattiinkin nyt likimääräisellä algoritmilla. Kaikki muut ohjelman osat pysyivät oleellisesti täysin samanlaisina.

Menetelmän L.1 yhteydessä kokeilin molempia tässä tekstissä esiteltyjä likimääräisiä etsintäalgoritmeja eli algoritmeja 2.3.2 ja 2.3.5. Otin lisäksi mukaan aiemmin kappaleessa 2.1 mainitsemani melko tuoreen ja käytännössä lähes kaikissa tilanteissa nopeimmaksi nykyisin tunnetuksi likimääräiseksi etsintäalgoritmiksi sanotun (esim. [Navarro, 1998] ja [Myers, 1998]) Gene Myersin "bitti-vektori"-algoritmin [Myers, 1998], joka noudattaa algoritmin 2.3.5 perusajatusta mutta toteuttaa sen nopeammin rinnakkaistamalla usean editointietäisyystaulukon alkion käsittelemisen soveltamalla sopivasti aritmeettisia operaatioita ja hahmon pohjalta muodostettua bittivektoria. Kuten taulukossa 3.5 esitetyt koeajojen tulokset osoittavat, on kyseinen algoritmi erittäin paljon nopeampi kuin nämä vanhemmat, klassiset menetelmät. Siitä huolimatta koko aineiston kattavassa etsinnässä päädyttiin turhan suureen suoritusaika-arvioon eli yli kolmeen kuukauteen.

Etsintäalgoritmi:	**Suoritusaika (sekuntia):**		**Aika-arvio koko genomin käsittelemiseksi (tuntia):**
	100 sekvenssiä	**1000 sekvenssiä**	
Perusdynaaminen	1818	18517	44235
Ukkonen	726	7272	17372
Myers	106	1098	2623

Taulukko 3.5:
Koeajo, jossa etsittiin ensin sadan ja sitten tuhannen leipurin hiivan genomista satunnaisesti valitun 25 merkin pituisen sekvenssin esiintymiä genomin muista geeneistä käyttäen likimääräistä etsintäalgoritmia ja virhemarginaalia 4. Koko genomin käsittelyä koskeva aika-arvio on muodostettu tuhannen sekvenssin koeajojen perusteella.

Menetelmän L.2 kohdalla suoritin kahdenlaisia koeajoja: Ensinnäkin kokeilin karsintamenetelmien tehoa eli sitä, kuinka monta prosenttia kokeillut karsintamenetelmät pystyivät pienentämään tarkistettavaa aineiston osaa, ja toiseksi suoritin näistä lupaavimpien menetelmien kohdalla kokonaiset koeajot käyttäen menetelmän L.1 kohdalla selvästi nopeimmaksi todettua likimääräistä etsintäalgoritmia eli Myersin algoritmia. Karsintamenetelmiä oli siis kolme: Baeza-Yates-Perleberg, tämän toteutus ottaen huomioon lisäksi osamerkkijonojen sijaintikohdat hahmossa ja laskenta. Kokeilin näitä kolmea erikseen sekä lisäksi laskennan yhdistämistä kahteen edelliseen. Tämä jälkimmäinen tapahtui siten, että ensin sovelletaan jompaakumpaa kahdesta ensimmäisestä karsintamenetelmästä ja sitten jäljelle jäänyt tutkittava alue karsitaan vielä laskentamenetelmällä. Baeza-Yates-Perlebergin sekä vastaavan mutta sijaintikohdat mukaan ottavan karsinta-algoritmin yhteydessä käytin Aho-Corasick-algoritmia osamerkkijonojen etsimiseksi aineistosta. Koska nyt etsittävien merkkijonojen pituus oli 25 merkkiä, suurin sallittu virhe 4 ja jokaisen editointioperaation kustannus 1, jaettiin hahmo Baeza-Yates-Perlebergissä sekä tämän osamerkkijonojen sijainnit huomioivassa versiossa viiteen viiden merkin pituiseen osamerkkijonoon, jolloin laskennassa kriteerinä oli, että tekstistä etsitään sellaisia 25 merkin pituisia osia, joiden merkeistä vähintään 21 voi omata vastinparin kulloinkin etsittävässä ehdokassekvenssissä. Taulukko 3.6 esittää näiden koeajojen tulokset:

Karsintamenetelmä:	**Karsintateho (%) / karsinta-aika (sek.)**	**Kokonaisetsintä aika (sekuntia)**	**Aika-arvio koko genomin käsittelemiseksi (tuntia):**
Baeza-Yates-Perleberg	71% / 81	109	2604
Sijaintikohtien huomiointi	81% / 83	99	2635
Laskenta	26% / 117	189	4515
BYP + laskenta	79% / 107	126	3010
Sijaintikohdat + laskenta	87% / 93	107	2556

Taulukko 3.6:
Koeajo, jossa kokeiltiin karsintamenetelmien tehokkuutta, karsintaan kuluvia aikoja ja kokonaissuoritusaikaa, kun etsitään sadan leipurin hiivan genomista satunnaisesti valitun 25 merkin pituisen sekvenssin esiintymiä genomin muista geeneistä käyttäen Myersin likimääräisen etsintäalgoritmin yhteydessä karsintaa ja virhemarginaalin ollessa 4.

Kuten taulukosta 3.6 voi päätellä, on laskenta tässä tapauksessa melko tehoton karsintatapa. Se tuo alhaisen karsintaprosentin ja on lisäksi suhteellisen hidasta. Taulukon 3.6 perusteella laskennan soveltaminen lisäkarsinnan aikaansaamiseksi jo itse asiassa hidastaa koko etsintäprosessia. Tämän voidaan osittain katsoa johtuvan Myersin algoritmin suuresta nopeudesta, sillä on selvää, että mitä nopeampi itse lopullinen likimääräinen etsintäalgoritmi on, niin sitä nopeampi/tehokkaampi myös karsintamenetelmän on oltava, jotta se säästäisi

suhteessa riittävästi laskenta-aikaa ollakseen hyödyllinen. Sen sijaan huomionarvoista on hahmon osamerkkijonojen sijaintikohtien huomioonottamisesta saatu hyöty, sillä jostain syystä tätä menettelyä ei kuitenkaan ole juurikaan kirjallisuudessa mainittu. Esimerkiksi Navarro [Navarro, 1998] mainitsee tavallisen Baeza-Yates-Perleberg-algoritmin joissain tapauksissa käytännössä parhaana menetelmänä likimääräisessä etsinnässä, vaikka sijaintikohtien huomiointi sen yhteydessä tuottaa esimerkiksi tässä tilanteessa melko suurenkin lisäkarsintatehon vaikuttamatta kuitenkaan merkittävästi karsimisvaiheessa kuluvaan suoritusaikaan. Edellisiä seikkoja painottaa se, että sijaintikohtien huomiointi oli taulukkojen 3.5 ja 3.6 perusteella tässä tapauksessa ainoa karsintatapa, joka oli kannattavaa verrattuna koko aineiston suoraan tutkimiseen Myersin algoritmilla.

Menetelmän L.3 koeajoissa ainoa muutos oli indeksin käyttö. Baeza-Yates-Perlebergin ja osamerkkijonojen sijaintikohtien huomioimisen yhteydessä tämä tapahtui siten, että jokaista erilaista viiden merkin mittaista merkkijonoa kohden muodostettiin lista, joka sisältää kaikki kyseisen merkkijonon esiintymäkohdat aineistossa. Koska tässä tapauksessa aakkoston koko oli vain 4, oli tällaisia pätkiä yhteensä ainoastaan 1024 kappaletta, ja siten nämä listat voitiin muodostaa helposti taulukkoon, jossa aina tiettyä viiden merkin pituista sekvenssiä vastaava lista sijaitsi kyseisen merkkijonon 4-järjestelmäesityksen antamassa taulukon kohdassa. Käytin tässä tulkintaa 'a' = 0, 't' = 1, 'c' = 2 ja 'g' = 3, jolloin siis esimerkiksi merkkijonon "atctg" sijaintikohdat ilmoittava lista sijaitsi taulukon kohdassa $01213_4 = 64 + 2\times16 + 4 + 3\times1 = 101$. Tällöin kaikki tietyn ehdokassekvenssin osamerkkijonojen sijaintikohdat tekstissä saatiin yksinkertaisesti laskemalla niiden 4-järjestelmäesitykset ja yhdistämällä sitten niitä vastaavat taulukosta löytyvät sijaintilistat. Lisänopeuttamisen vuoksi alustin myös sellaisen taulukon, joka kertoo suoraan jokaiseen eri aineiston kohtaan päättyvän viiden merkin pituisen merkkijonon 4-järjestelmäesityksen arvon, jolloin aina kunkin ehdokassekvenssin sijainnin perusteella voitiin nopeasti hakea sen osamerkkijonojen 4-järjestelmäesitysten arvot ja niiden avulla sitten sijaintilistat.

Laskentaan perustuvan karsinnan yhteydessä tyydyin toteuttamaan sellaisen indeksin, joka kertoo sijainnin perusteella jokaisen aineistossa olevan 25 merkin pituisen sekvenssin merkkijakauman. Periaatteessa olisi ollut mahdollista muodostaa lisäksi edellisen indeksin soveltamista merkittävästi tehostava toinen indeksi, joka olisi kertonut suoraan laskennassa käytetyssä karsintakriteerissä esiintyvän summalausekkeen arvon kahden eri merkkijonon välille näiden merkkijakaumien perusteella, mutta käytännössä tämä tuntui sen verran monimutkaiselta toteuttaa käytännössä, että se tuskin kannattaisi ottaen huomioon laskennan heikohkon karsintatehon. Käyttäessäni muodostamaani indeksiä karsinnassa otin huomioon,

että koska nyt jokaisen aineistossa esiintyvän 25 merkin pituisen sekvenssin merkkijakauma saatiin nopeasti indeksistä selville, ei karsintakriteerin summalausekkeen arvon laskemiseksi tarvittu kuin neljä toimenpidettä eli yksi jokaista eri aakkoston merkkiä kohti.

Törmäsin näiden koeajojen yhteydessä odottamattomaan ja sinänsä mielenkiintoiseen ongelmaan. Nimittäin, vaikka laskelmieni mukaan tietokoneen muistin olisi pitänyt riittää reilusti indeksien muodostamiseen, osoittautui, että varsinainen muisti jostain syystä loppui kesken ja käyttöjärjestelmä turvautui kiintolevyaseman käyttöön virtuaalimuistina. Tämän vuoksi indeksointi oli aluksi jopa hitaampaa kuin jokaisen ehdokassekvenssin kohdalla tehtävään lineaariseen aineiston prosessointiin perustuva karsinta. Tein sellaisen johtopäätökseen, että ongelma liittyisi muistinvaraustaulukon toteutukseen liittyvästä pienimmästä varausyksiköstä, sillä indeksitoteutukseni perustui listoihin, joissa oli miljoonia dynaamisesti luotavia alkioita. Oletan, että johtopäätökseni oli oikea, sillä ongelma poistui, kun hoidin muistinhallinnan itse siten, että varasin kerralla riittävän suuren osoitintaulukon, josta varasin aina järjestyksessä yksitellen uusia alkioita listoja varten. Koska listojen alkiot olivat luonteeltaan staattisia sen jälkeen, kun ne oli luotu, ei tämä ongelma tuottanut enää sen suurempia vaikeuksia.

Taulukko 3.7 esittää aiemman perusteella lupaavimpien karsintamenetelmien soveltamiseen kuluvat suoritusajat indeksointia käyttäen. Verrattaessa taulukkoja 3.6 ja 3.7 keskenään havaitaan indeksoinnin hyöty ilmeiseksi. Taulukko 3.8 sisältää samat karsintamenetelmät kuin taulukko 3.7, mutta nyt siten, että lisäksi on suoritettu karsinnan pohjalta yhä tarkistettaviksi todettujen alueiden tutkiminen Myersin likimääräisellä etsintäalgoritmilla.

Karsintamenetelmä:	**Karsinta (sekuntia):**		**Aika-arvio koko genomin**
	100 sekvenssiä	**1000 sekvenssiä**	**käsittelemiseksi (tuntia):**
Indeksoitu BYP	5	54	129
Indeksoitu sijaintikohdat	5	51	122
Ind. sijainnit + laskenta	20	216	516
Ind. sijainnit + ind. lask.	18	177	423

Taulukko 3.7:
Koeajo, jossa kokeiltiin karsintaan kuluvia aikoja käyttäen hyväksi indeksointia, kun etsitään ensin sadan ja sitten tuhannen leipurin hiivan genomista satunnaisesti valitun 25 merkin pituisen sekvenssin esiintymiä genomin muista geeneistä virhemarginaalin ollessa 4. Koko aineiston käsittelemisen aika-arvio perustuu tuhannen sekvenssin koeajoon.

Karsintamenetelmä:	Karsinta ja etsintä Myersillä (sekuntia): 100 sekvenssiä	1000 sekvenssiä	Aika-arvio koko genomin käsittelemiseksi (tuntia):
Indeksoitu BYP	38	392	936
Indeksoitu sijaintikohdat	26	269	643
Ind. sijainnit + laskenta	31	315	753
Ind. sijainnit + ind. lask.	28	292	698

Taulukko 3.8:
Koeajo, jossa kokeiltiin likimääräiseen etsintään kuluvaa kokonaissuortusaikaa käytettäessä Myersin likimääräistä etsintäalgotimia sekä indeksoitua karsintaa, kun etsitään ensin sadan ja sitten tuhannen leipurin hiivan genomista satunnaisesti valitun 25 merkin pituisen sekvenssin esiintymiä genomin muista geeneistä virhemarginaalin ollessa 4. Koko aineiston käsittelemisen aika-arvio perustuu tuhannen sekvenssin koeajoon.

Taulukko 3.8 vahvistaa sen, että laskenta ei tuo tässä yhteydessä riittävän suurta lisäkarsintaa ollakseen kannattavaa kokonaissuoritusajan kannalta. Tämän koeajon perusteella koko aineiston käsitteleminen veisi suunnilleen 27 vuorokautta, joka on yhä turhan pitkä aika käytännössä. Arviota voinee pitää melko hyvänä siltä osin, että laskenta-aika tuntuisi melko tarkalleen kymmenkertaistuvan siirryttäesä 100 ehdokassekvenssistä tuhanteen. Siten laskenta-aika tuntuisi skaalautuvan melko suoraan tutkittavien sekvenssien lukumäärän mukaan.

Pienimuotoista tulosanalyysiä

Projektin likimääräinen osuus epäonnistui siltä osin, että tavoitteena ollutta koko hiivan geeni-aineiston prosessointia ei ainakaan toistaiseksi toteutettu liian suuren laskenta-ajan takia. Mutta, jos käytettävissä on riittävät resurssit, ei kyseisen laskennan suorittaminen kuitenkaan ole ongelma, sillä selvästi eri ehdokassekvenssien tutkimisprosessit voidaan tehdä toisistaan riippumattomasti, ja siten koko aineiston tutkinta voidaan hajauttaa laskettavaksi rinnakkain usealla tietokoneella samanaikaisesti. Muuten projektissa saaavutettua ehdokassekvenssien likimääräisen tarkistamisen tehokkuutta voitaneen pitää sinänsä hyvänä, sillä käytössä ollut likimääräinen etsintäalgoritmi on tällä hetkellä tunnetuista nopein, karsinta saatiin toteutettua suoraan indeksoidun taulukon avulla, mitä nopeampaa tapaa ei liene olemassa, ja käytetty karsintamenetelmä on todettu nykyisin tunnetuista käytännössä tehokkaimmaksi ([Baeza-Yates and Navarro, 1996]).

Lisäksi Baeza-Yates ja Navarro ovat analysoineet käytännössä muodostamani indeksointimenetelmän kanssa lähes identtistä menettelyä ([Baeza-Yates and Navarro, 1998], ja [Baeza-Yates and Navarro, 1999]. He tosin pysyttäytyivät pelkässä Baeza-Yates-Perlebergin karsintamenetelmässä eli eivät ottaneet huomioon osamerkkijonojen sijaintikohtia hahmossa, ja lisäksi he käyttivät paljon hitaampaa likimääräistä etsintäalgoritmia, itsenikin koeajoissa kokeilemaa Ukkosen parannettua dynaamista algoritmia. He vertailivat menetelmän tehoa muihin tunnettuihin indeksointimenetelmiin käyttäen koeaineistona mm. 20 merkin pituisia DNA-sekvenssejä ja virhemarginaalina arvoa $s_e = 4$, ja niiden osalta Baeza-Yates-Perlebergiin perustuva indeksointi oli suunnilleen puolet hitaampi kuin nopein menetelmä. Mutta tässä on otettava huomioon käytettävän likimääräisen etsintäalgoritmin aiheuttama ero, sillä heidän tutkimistaan menetelmistä nopeimmat käyttivät Ukkosen algoritmia hienostuneempia ja nopeampia likimääräisiä etsintämenetelmiä. Toisaalta suorittamieni koeajojen pohjalta tiedetään, että tämän projektin tapauksessa Myersin likimääräisen algoritmin käyttäminen nopeutti etsintää reilusti enemmän kuin kaksinkertaisesti Ukkosen algoritmin käyttämiseen nähden, mikä riittäisi jo hyvinkin kuromaan näiden muiden menetelmien etumatkan kiinni. Tätä päätelmää tukee myös se, että heidän koeajoissaan Baeza-Yates-Perlebergin periaatetta käyttävän menetelmän osalta käytettiin samaa virhemarginaalia kuin tässä, mutta lyhyempien merkkijonojen yhteydessä. Tällöin etsintäalgoritmin nopeus painottuu vielä enemmän, koska tällöin karsintateho jää alhaisemmaksi ja siten likimääräisellä etsintäalgoritmilla joudutaan tutkimaan suurempi alue tekstistä (aineistosta). Ja kun lisäksi otetaan huomioon osamerkkijonojen sijaintien huomioimisesta tuleva nopeutus Baeza-Yates-Perlebergin karsintaperiaatteeseen verrattuna,

voidaan saavutettuja tuloksia pitää mielestäni jo melko hyvinä. Luonnollisesti paras vaihtoehto olisi toteuttaa vertailevat koeajot eri menetelmien välillä, mutta se ei tuntunut tämän projektin kannalta järkevältä, sillä edellä mainitsemistani syistä johtuen en odottanut näiden toisten vaihtoehtojen olevan oleellisesti tehokkaampia. Lisäksi nämä molemmat edellä viittaamistani Baeza-Yatesin ja Navarron testeissä nopeimmista indeksointimenetelmistä ([Myers, 1994] ja [Baeza-Yates and Navarro, 1999]) ovat toteutukseltaan sen verran monimutkaisia, että pelkkä niiden muodostamiseen kuluva työmäärä olisi vienyt käytännön kannalta pieniin odotuksiin nähden turhan paljon projektin aikaa.

Mahdollisen jatkotutkimuksen kannalta ajatuksena olisi kuitenkin toteuttaa eri menetelmien vertailua laajemmin käytännössä ja lisäksi pyrkiä tehostamaan tässä käytettyjä karsintamenetelmiä hieman lisää. Joka tapauksessa mitään huimaa koko aineiston käsittelemisessä kuluvan odotetun suoritusajan parannusta ei ole odotettavissa, sillä käytännössä Myersin algoritmia nopeampaa likimääräistä etsintäalgoritmia tuskin enää saavutetaan ([Navarro, 1998]), ja toisaalta tässä käytetty indeksoitu karsintamenetelmä oli toteutukseltaan jo niin suoraviivaisen nopea käyttäessään suoraa taulukkoa, että on vaikea kuvitella, että sitä merkittävästi paremmin toimiva karsintamenetelmä voisi toimia yhtä nopeasti ainakaan nykyisten karsintamenetelmien puitteissa. Taulukkojen 3.5 - 3.8 perusteella odotettua kokonaissuoritusaikaa voitaisiin arvioida melko tarkkaan kaavalla *genomin prosessointiaika* $\approx$ *karsinta-aika* + ((100 - *karsintaprosentti*) / 100) $\times$ 8,6 $\times$ 10^6 sekuntia. Tämän perusteella, jos karsintatehoa saataisiin nostettua esimerkiksi 95% tienoille, mikä olisi jo erittäin suuri parannus, saisi karsintaan kuluva aika kuitenkin kasvaa korkeintaan n. 20%, jotta odotettu kokonaissuoritusaika puolittuisi. Näin ollen lienee nykymenetelmien valossa melko selvää, että koko genomin prosessoinnin suoritusajan parantamisen kannalta käytännössä tehokkain keino on turvautua aiemmin mainittuun lähestymistapaan eli laskennan hajauttamiseen suoritettavaksi samanaikaisesti usealla tietokoneella.

Viiteluettelo

[Aho and Corasick, 1975] = A. Aho and M. Corasick, Efficient string matching: an aid to bibliographic search. Communications of the ACM 18 (1975), 333-340.

[Aula et al., 1998] = Toim. P. Aula, H. Kääriäinen, ja J. Leisti, Perinnöllisyyslääketiede. Duodecim, Helsinki (1998).

[Baeza-Yates and Navarro, 1996] = R. Baeza-Yates and G. Navarro, A faster algorithm for approximate string matching. Proceedings of CPM´96, Lecture Notes in Computer Science 1075 (1996), 1-23.

[Baeza-Yates and Navarro, 1998] = R. Baeza-Yates and G. Navarro, A practical q-gram index for text retrieval allowing errors. Clei Electronic Journal (www.clei.cl) 1 (1998).

[Baeza-Yates and Navarro, 1999] = R. Baeza-Yates and G. Navarro, A new indexing method for approximate string matching. Proceedings of CPM'99, Lecture Notes in Computer Science 1645 (1999), 163-185.

[Baeza-Yates and Perleberg, 1996] = R. Baeza-Yates and C. Perleberg, Fast and practical approximate pattern searching. Information Processing Letters 59 (1996), 21-27.

[Bentley and Sedgewick, 1997] = J. Bentley and R. Sedgewick, Fast algorithms for sorting and searching strings. Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (1997), 360-369.

[Boyer and Moore, 1977] = R. S. Boyer and J. S. Moore, A fast string searching algorithm. Communications of the ACM 20 (1977), 762-772.

[Chang and Lampe, 1992] = W. Chang and J. Lampe, Theoretical and empirical comparisons of approximate string matching algorithms. Proceedings of CPM'92, Lecture Notes in Computer Science 644 (1992), 172-181.

[Cole, 1994] = R. Cole, Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm. SIAM Journal on Computing 23 (1994), 1075-1091.

[Crochemore and Rytter, 1994] = M. Crochemore and W. Rytter, Text Algorithms. Oxford University Press, New York (1994).

[Gusfield, 1997] = D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997).

[Holsti and Sutinen, 1994] = N. Holsti and E. Sutinen, Approximate string matching using q-gram places. Proceedings of the 7th Finnish Symposium on Computer Science, University of Joensuu (1994), 23-32.

[Horspool, 1980] = N. Horspool, Practical fast searching in strings. Software Practice and Experience 10 (1980), 501-506.

[Jokinen et al., 1996] = P. Jokinen, J. Tarhio, and E. Ukkonen, A comparison of approximate string matching algorithms. Software Practice and Experience 26 (1996), 1439-1458.

[Karp and Rabin, 1987] = R. Karp and M. Rabin, Efficient randomized pattern matching algorithms. IBM Journal on Research Development 31 (1987), 249-260.

[Knuth et al., 1977] = D. E. Knuth, J. H. Morris, and V. R. Pratt, Fast pattern matching in strings. SIAM Journal on Computing 6 (1977), 323-350.

[Landau and Vishkin, 1989] = G. Landau and U. Vishkin, Fast parallel and serial approximate string matching. Journal of Algorithms 10 (1989), 157-169.

[Lecroq, 1995] = T. Lecroq, Experimental results on string matching algorithms. Software Practice and Experience 25 (1995), 728-765.

[Levenshtein, 1966] = V. I. Levenshtein, Binary codes capable of correcting insertions and reversals. Cybernetics and Control Theory 10 (1966), 707-710.

[Lockhart et al., 1996] = D. J. Lockhart, H. Dong, M. C. Byrne, M. T. Follettie, M. V. Gallo, M. S. Chee, M. Mittmann, C. Wang, M. Kobayashi, H. Horton, and E. L. Brown, Expression monitoring by hybridization to high-density oligonucleotide arrays. Nature Biotechnology 14 (1996), 1675-1680

[Löffert et al., 1998] = D. Löffert, N. Seip, S. Karger, and J. Kang, PCR optimization: degenerate primers. Qiagen News issue 2 1998, 3-6.

[Morris and Pratt, 1970] = J. H. Morris and V. R. Pratt, A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley (1970).

[Muth and Manber, 1996] = R. Muth and U. Manber, Approximate multiple string search. Proceedings of CPM'96, Lecture Notes in Computer Science 1075 (1996), 75-86.

[Myers, 1994] = E. Myers, A sublinear algorithm for approximate keyword searching. Algorithmica 12 (1994), 345-374.

[Myers, 1998] = G. Myers, A fast bit-vector algorithm for approximate string matching based on dynamic programming. Proceedings of CPM'98, Lecture Notes in Computer Science 1448 (1998), 1-13.

[Navarro, 1997] = G. Navarro, Multiple approximate string matching by counting. Proceedings of WSP'97 (1997), 125-139.

[Navarro, 1998] = G. Navarro, Approximate Text Searching. PhD Thesis, Dept. of Computer Science, University of Chile (1998).

[NCBI] = National Center for Biotechnology Information, www-sivujen osoite (Toukokuussa 2000): http://www.ncbi.nlm.nih.gov.

[Needleman and Wunsch, 1970] = S. Needleman and C. Wunsch, A general method applicable to the search for similarities in the amino acid sequences of two proteins. Journal of Molecular Biology 48 (1970), 444-453.

[Rytter, 1980] = W. Rytter, A correct preprocessing algorithm for Boyer-Moore string searching. SIAM Journal on Computing 9 (1980), 509-512.

[Sellers, 1980] = P. Sellers, The theory and computation of evolutionary distances: pattern recognition. Journal of Algorithms 1 (1980), 359-373.

[Stephen, 1994] = G. A. Stephen, String Searching Algorithms. Worlds Scientific, Singapore (1994).

[Sunday, 1990] = D. M. Sunday, A very fast substring search algorithm. Communications of the ACM 33 (1990), 132-142.

[Sutinen, 1998] = E. Sutinen, Approximate Pattern Matching with the q-Gram Family. PhD Thesis, Department of Computer Science, University of Helsinki (1998).

[Ukkonen, 1985a] = E. Ukkonen, Algorithms for approximate string matching. Information and Control 64 (1985), 100-118.

[Ukkonen, 1985b] = E. Ukkonen, Finding approximate patterns in strings. Journal of Algorithms 6 (1985), 132-137.

[Wodicka et al., 1997] = L.Wodicka, H. Dong, M. Mittmann, M. Ho, and D. J. Lockhart, Genome-wide expression monitoring in Saccharomyces cerevisiae. Nature Biotechnology 15 (1997), 1359-1367.

[Wu and Manber, 1992] = S. Wu and U. Manber, Fast text searching allowing errors. Communications of the ACM 35 (1992), 83-91.

Sisällys: