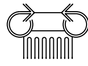
 Tietojenkäsittelytiede **31** Marraskuu 2010 sivut 70–83 © kirjoittaja(t)
Toimittaja: Antti Valmari

# Merkkijonoalgoritmeja monen hahmon hakuun

Leena Salmela
Helsingin yliopisto
Tietojenkäsittelytieteen laitos
Tietotekniikan tutkimuslaitos HIIT
leena.salmela@cs.helsinki.fi

**Tiivistelmä**

Esittelemme algoritmeja merkkijonojoukon hakemiseen tekstistä. Aloitamme esittelemällä trierakennetta käyttäviä algoritmeja kuten klassinen Aho–Corasick-algoritmi ja kuvaamme sitten uusia suodatukseen perustuvia algoritmeja, jotka pystyvät nopeasti suodattamaan monet tekstin paikoista pois potentiaalisten esiintymien joukosta. Lopuksi vertaamme algoritmeja kokeellisesti.

## 1 Johdanto

Merkkijonohakuongelmissa etsitään lyhyttä merkkijonoa eli hahmoa pitkästä merkkijonosta eli tekstistä. Tässä artikkelissa käsitellään erityisesti monen hahmon hakua. Nyt syötteenä on joukko hahmoja ja teksti, ja tehtävänä on etsiä kaikkien hahmojen kaikki esiintymät tekstistä. Keskitymme tässä menetelmiin, jotka esikäsittelevät hahmojoukon. Joissain sovelluksissa on mahdollista esikäsitellä teksti, jolloin tekstistä muodostetaan hakemistorakenne ja hahmoja etsitään yksi kerrallaan hakemistorakenteen avulla.

Asymptoottisesti monen hahmon haku on hyvin ratkaistu sekä pahimmassa että keskimääräisessä tapauksessa. Ongelmaan on esitetty lukuisia hyviä algoritmeja, joista osa on optimaalisia pahimmassa tapauksessa, osa keskimääräisessä tapauksessa – ja osa ei ole optimaalisia pahimmassa eikä keskimääräisessä tapauksessa, mutta kuitenkin käytännössä nopeita. Uusi tutkimus keskittyykin kehittämään käytännössä tehokkaita algoritmeja eri sovellusalueilta nouseviin ongelmiin. Monet vanhoista algoritmeista keskittyvät korkeintaan muutaman sadan hahmon hakuun, mutta uudet sovellukset esimerkiksi bioinformatiikassa vaativat miljoonien hahmojen käsittelyä tehokkaasti.

Monet algoritmeista ovat nopeita, koska ne eivät lue tekstin kaikkia merkkejä. Toisaalta algoritmit voivat lukea montakin merkkiä hahmojen kanssa kohdistetusta tekstistä ennen kuin ne pystyvät päättelemään, että kyseisessä kohdassa ei ole esiintymää. Tällöin tehokas algoritmi pyrkii käyttämään luetuista merkeistä saadun informaation siirtämällä kohdistusta näiden perusteella mahdollisimman paljon eteenpäin. Monet algoritmit käyttävät nopeita heuristiikkoja, jotka tuottavat myös vääriä osumia tai siirtävät hahmoa liian vähän eteenpäin. Tämä ei välttämättä haittaa, jos tästä aiheutuva lisätyö jää keskimäärin pieneksi.

OHJE KIRJAPAINOLLE: B5-arkin vasen- ja yläreuna kohdistetaan A4-arkin vasempaan ja yläreunaan. Näin pitäisi marginaaliksi tulla taitteen puolella noin 33 mm ja muualla noin 22 mm.

Esitetyt menetelmät voidaan jakaa kahteen luokkaan. *Triemenetelmät* rakentavat hahmoista trien ja käyttävät sitä joko sellaisenaan tai jollain tavalla täydennettynä hahmojen etsimiseen tekstistä. *Suodatusmenetelmät* taas perustuvat tekstin paikkojen eli potentiaalisten esiintymien hylkäämiseen jonkin nopeasti laskettavan kriteerin perusteella.

Määrittelemme aluksi tarvittavat käsitteet luvussa 2. Seuraavaksi kuvaamme luvussa 3 yhden hahmon hakuun tarkoitettuja algoritmeja, joita käytetään rakennuspalikoina esitellyissä monen hahmon hakualgoritmeissa. Sitten esittelemme trietä hyödyntäviä menetelmiä luvussa 4 ja suodatusmenetelmiä luvussa 5. Lopuksi esitämme kokeellisia tuloksia luvussa 6.

## 2 Määritelmiä

Jos $S = s_1 s_2 \cdots s_n$ on merkkijono, kaikki jonot $S' = s_i s_{i+1} \cdots s_j$, missä $1 \le i \le j \le n$, ovat merkkijonon $S$ *osajonoja*. Jos $i = 1$, $S'$ on merkkijonon $S$ *alkuosa*, ja jos $j = n$, $S'$ on merkkijonon $S$ *loppuosa*. Kyseessä on *aito alkuosa*, jos $j < n$ eli alkuosa ei ole koko alkuperäinen merkkijono. Vastaavasti loppuosa on *aito loppuosa*, jos $i > 1$. Tyhjä merkkijono $\varepsilon$, jonka pituus on 0, on jokaisen merkkijonon osajono, alkuosa ja loppuosa. Merkitsemme merkkijonon $S$ käännettyä merkkijonoa $S^R = s_n s_{n-1} \cdots s_1$. Kutsumme *q-piirteeksi* $q$:n merkin pituista merkkijonoa.

*Yhden hahmon haussa* syötteenä on kaksi merkkijonoa, *hahmo* $P = p_1 \cdots p_m$ ja *teksti* $T = t_1 \cdots t_n$. Hahmon pituus on $m$ ja tekstin $n$. Ongelmana on löytää kaikki hahmon *esiintymät* tekstistä eli kaikki selläiset paikat $j$ että $t_j \cdots t_{j+m-1} = p_1 \cdots p_m$.

*Monen hahmon hakuongelma* on muuten samanlainen, mutta syötteenä on $r$ hahmoa $P_1, P_2, \ldots, P_r$ ja ongelmana on löytää tekstistä kaikkien hahmojen kaikki esiintymät. Hahmojen pituudet voivat vaihdella. Tällöin merkitsemme $m$:llä lyhyimmän hahmon pituutta.

Osa esitetyistä algoritmeista käyttää bittivektoreita ja bittioperaatioita. Indeksoimme bittivektoreita oikealta vasemmalle eli vähiten merkitsevän bitin indeksi on siis yksi. Käytämme seuraavia C-kielestä tuttuja merkintöjä esittämään bittioperaattoreita: $\ll$ (bittien siirto vasemmalle, alin bitti asetetaan nollaksi), $|$ (biteittäinen or) ja & (biteittäinen and).

## 3 Yhden hahmon algoritmeja

### 3.1 Boyer–Moore–Horspool

Boyer ja Moore [5] esittivät yhden hahmon hakuun menetelmän, jossa hahmon kanssa kohdistettu teksti luetaan takaperin ja hahmoa siirretään tekstissä eteenpäin hahmon lopun kanssa kohdistetun tekstin perusteella. Tällöin voidaan keskimäärin jättää osa tekstin merkeistä lukematta. Boyer–Moore–Horspool [10] on yksinkertainen versio tästä algoritmista. Algoritmi laskee siirtymän yksinomaan hahmon viimeisen paikan kanssa kohdistetun merkin perusteella.

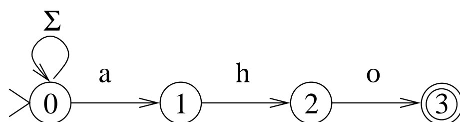
Boyer–Moore–Horspool-algoritmi käyttää haussa apuna *hyppytaulua*, johon on tallennettu siirtymän pituus kullekin aakkoston merkille. Hahmon kanssa kohdistettua tekstiä verrataan merkki merkiltä oikealta vasemmalle, kunnes tekstin merkki ei täsmää hahmoon tai koko hahmo on käsitelty. Tämän jälkeen siirtymä luetaan hyppytaulusta hahmon viimeisen paikan kanssa kohdistetun merkin kohdalta ja kohdistusta siirretään tämän mukaisesti.

Esikäsittelynä algoritmi tallentaa hyppytauluun kullekin aakkoston merkille pisimmän turvallisen hypyn pituuden. Tämä on lyhyin etäisyys hahmon lopusta kyseisen merkin esiintymään hahmon alkuosassa $p_1 \cdots p_{m-1}$. Jos merkki ei esiinny hahmon alkuosassa, hyppytauluun tallennetaan hahmon pituus.

| merkki | siirtymä |
|---|---|
| a | 2 |
| h | 1 |
| muu | 3 |

| o | h | o |   | a | h | o |
|---|---|---|---|---|---|---|
| a | h | o |   |   |   |   |
|   |   |   | a | h | o |   |
|   |   |   |   | a | h | o |

**Kuva 1:** Boyer–Moore-Horspool-algoritmin hyppytaulu ja kohdistukset hahmolle ”aho” ja tekstille ”oho aho”.




**Kuva 2:** Shift-orin simuloima automaatti hahmolle ”aho”. Tilasta 0 on siirtymä itseensä kaikilla aakkoston merkeillä.

Kuvassa 1 on vasemmalla esimerkki Boyer–Moore–Horspool-algoritmin hyppytaulusta hahmolle ”aho” ja oikealla kohdistukset, kun kyseistä hahmoa haetaan tekstistä ”oho aho”. Ensimmäisessä kohdistuksessa luetaan kolme merkkiä tekstistä ja huomataan, että kohdistus ei täsmää hahmoon. Hahmon viimeisen paikan kanssa kohdistettu merkki on ’o’, joten hahmoa siirretään kolme paikkaa oikealle hyppytaulun mukaisesti. Toisessa kohdistuksessa riittää yhden merkin lukeminen toteamaan, että kohdistus ei täsmää hahmoon. Hahmoa siirretään hyppytaulun mukaisesti yksi paikka eteenpäin ja luettuaan koko kohdistuksen algoritmi löytää esiintymän.

### 3.2 Shift-or

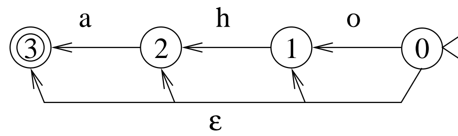
Shift-or-algoritmi [3] simuloi yksinkertaista epädeterminististä äärellistä automaattia. Kuvassa 2 on esimerkki shift-orin simuloimasta automaatista hahmolle ”aho”. Kun tälle automaatille syötetään teksti merkki kerrallaan, automaatin lopputila on aktiivinen hahmon jokaisen esiintymän lukemisen jälkeen. Automaattia simuloidaan bittirinnakkaisesti käyttämällä shift- ja or-komentoja.

Esikäsittelynä shift-or-algoritmi koodaa automaatin siirtymät alustamalla taulun $B$, johon se tallentaa bittivektorin kullekin aakkoston merkille. Bittivektorin paikkaan $i$ tallennetaan nolla, jos kyseinen merkki esiintyy hahmossa paikassa $i$ eli automaatissa on siirtymä kyseisellä merkillä tilasta $i-1$ tilaan $i$. Muuten tallennetaan ykkönen.

Hakuvaiheessa algoritmi lukee tekstin merkki kerrallaan vasemmalta oikealle. Haun aikana algoritmi ylläpitää bittivektoria $E$, jossa on nolla niissä paikoissa, joita vastaavat tilat automaatissa ovat aktiivisia. Huomaa, että tila 0 on aina aktiivinen eikä sitä näin ollen ole tarpeen esittää bittivektorissa $E$. Alussa bittivektori $E$ sisältää pelkkiä ykkösiä. Kun algoritmi lukee tekstistä merkin $c$, bittivektoria $E$ päivitetään seuraavasti:

$$E = (E \ll 1) \mid B[c],$$

missä bittien siirto vasemmalle edustaa siirtymistä automaatin tilasta toiseen ja or-operaatio varmistaa, että siirtyminen mallinnetaan vain, jos luettiin siirtymää vastaava merkki. Bittien siirto vasemalle myös aktivoi automaatin ensimmäisen tilan asettamalla alimman bitin nollaksi. Jos




**Kuva 3:** BNDM-algoritmin simuloima automaatti hahmolle ”aho”.

tämän jälkeen automaatin viimeistä tilaa vastaava bitti vektorissa $E$ on nolla eli viimeinen tila on aktiivinen, tekstistä on löytynyt hahmon esiintymä.

### 3.3 BNDM

BNDM (Backward nondeterministic DAWG matching) [16] on tehokas bittirinnakkainen algoritmi, joka shift-or-algoritmin tapaan simuloi epädeterminististä äärellistä automaattia. BNDM-algoritmin käyttämä automaatti tunnistaa kaikki käännetyn hahmon loppuosat eli alkuperäisen hahmon alkuosat. Kuvassa 3 on esimerkki tällaisesta automaatista hahmolle ”aho”.

BNDM lukee hahmon kanssa kohdistetun tekstin takaperin ja etenee automaatissa siirtymien mukaisesti. Kussakin kohdistuksessa pidetään kirjaa pisimmästä tunnistetusta alkuosasta. Jos pisin alkuosa on yhtä pitkä kuin hahmo, algoritmi on löytänyt esiintymän. Kun mikään automaatin tiloista ei enää ole aktiivinen, luetut merkit kohdistuksesta eivät muodosta mitään hahmon osajonoista ja näin kohdistuksen tutkiminen voidaan lopettaa. Lopuksi kohdistusta siirretään eteenpäin siten, että pisin tunnistettu alkuosa (tai toiseksi pisin, mikäli pisin on hahmon esiintymä) on kohdistettu hahmon alkuosan kanssa.

Shift-or-algoritmin tapaan BNDM simuloi epädeterminististä automaattia bittirinnakkaisesti. Esikäsittelynä alustetaan taulu $B$, johon tallennetaan kullekin aakkoston merkille automaatin siirtymät bittivektorina. Bittivektorin $B[c]$ $i$:s bitti on ykkönen, jos ja vain jos merkki $c$ esiintyy käännetyssä hahmossa paikassa $i$ eli automaatissa on kyseisellä merkillä siirtymä tilasta $i-1$ tilaan $i$.

Shift-or-algoritmista poiketen BNDM merkitsee aktiiviset tilat ykkösillä. Hakuvaiheessa aina, kun aloitetaan uuden kohdistuksen tarkastelu, tilavektori $E$ asetetaan arvoon $B[c]$, missä $c$ on kyseisen kohdistuksen viimeinen merkki. Tilavektorissa ovat tällöin aktiivisena tilat, jotka automaatissa ovat aktiivisia ensimmäisen merkin lukemisen jälkeen. Kun kohdistuksesta luetaan uusi merkki $c$, päivitetään tilavektoria seuraavasti:

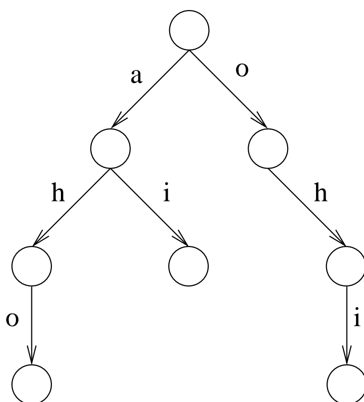
$$E = (E \ll 1) \; \& \; B[c].$$

Kun $m$:s bitti on päällä, luetut merkit kohdistuksesta täsmäävät hahmon alkuun eli on tunnistettu hahmon alkuosa.

## 4 Triemenetelmät

*Trie* on puurakenne, jota käytetään merkkijonojoukon indeksointiin. Kuhunkin puun kaareen liittyy merkki ja kustakin solmusta lapsisolmuihin lähtevissä kaarissa on eri merkit. Jokainen polku puun juuresta lehteen vastaa yhtä indeksoitua merkkijonoa. Kyseinen merkkijono saadaan yhdistämällä polun kaarien merkit juuresta lehteen. Kuvassa 4 on merkkijonoista ”aho”, ”ai” ja ”ohi” muodostettu trie.

Triemenetelmät indeksoivat hahmojoukon trien avulla. Niiden ongelma suuren hahmojoukon hakemisessa on trien vaatima suuri tila. Uusia itseindeksirakenteita [15] voidaan käyttää hahmojoukon




**Kuva 4:** Merkkijonoista ”aho”, ”ai” ja ”ohi” muodostettu trie.

indeksointiin [4, 7, 9, 20], mutta tämän lisäksi täytyy käyttää jonkinlaista suodatusta, jotta algoritmi olisi käytännössä riittävän nopea [7].

### 4.1 Aho–Corasick

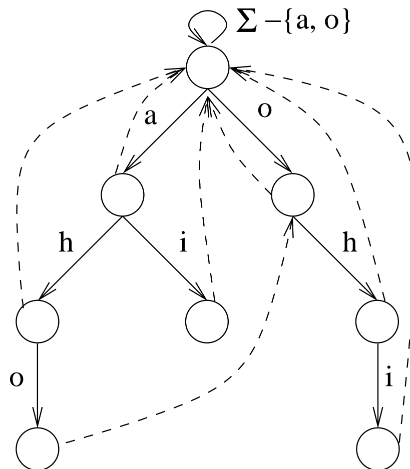
Aho–Corasick-algoritmissa [1] hahmojoukosta muodostetaan automaatti, jota käytetään esiintymien hakemiseen. Automaatti koostuu hahmojoukosta muodostetusta triestä, johon on lisätty ns. korjaussiirtymiä. Algoritmi lukee tekstin merkki kerrallaan vasemmalta oikealle ja etenee automaatissa luettujen merkkien mukaisesti. Korjaussiirtymiä tarvitaan, kun trien perusteella on päädytty tilaan, josta ei ole siirtymää luetulla merkillä.

Olkoon solmun *polkunimi* se merkkijono, joka saadaan yhdistämällä juuresta solmuun kulkevan polun kaarien merkit. Jokaisesta solmusta lisätään *korjaussiirtymä* siihen solmuun, jonka polkunimi on pisin aito loppuosa alkuperäisen solmun polkunimestä. Lisäksi juuresta on siirtymä juureen jokaisella sellaisella merkillä, jolla ei muuten olisi siirtymää juuresta. Lisätään vielä kuhunkin solmuun lista niistä hahmoista, jotka vastaavat solmun polkunimeä. Näin muodostettua automaattia sanotaan *Aho–Corasick-automaatiksi*. Kuvassa 5 on merkkijonoista ”aho”, ”ai” ja ”ohi” muodostettu Aho–Corasick-automaatti.

Aho–Corasick-automaattia käytetään siis seuraavasti esiintymien hakemiseen. Tekstiä luetaan merkki kerrallaan ja edetään siirtymien mukaisesti, mikäli solmusta on siirtymä luetulla merkillä. Jos tällaista siirtymää ei ole, edetään korjaussiirtymiä pitkin, kunnes saavutaan solmuun, josta on siirtymä luetulla merkillä ja edetään tätä siirtymää pitkin. Tällainen solmu löytyy aina, koska juuresta on siirtymä kaikilla merkeillä.

Kun Aho–Corasick-automaatissa ollaan tilassa $v$, tilan $v$ polkunimi on tekstin luetun osan loppuosa. Näin ollen tekstistä on tällöin löytynyt esiintymät niille hahmoille, jotka ovat kyseiseen tilaan liittyvässä listassa.

Aho–Corasick-algoritmia voidaan nopeuttaa lisäämällä kustakin automaatin tilasta siirtymä kaikilla aakkoston merkeillä. Nämä uudet siirtymät saadaan seuraamalla korjaussiirtymiä aivan kuten hakuvaiheessakin. Automaatti vie kuitenkin tällöin enemmän tilaa, mikä voi olla ongelmallista isolla hahmojoukolla.




**Kuva 5:** Merkkijonoista ”aho”, ”ai” ja ”ohi” muodostettu Aho–Corasick-automaatti. Korjaussiirtymät on merkitty katkoviivalla.

### 4.2 Commentz-Walter ja muunnelmat

Commentz-Walter [6] yleisti ensimmäisenä Boyer–Moore-tyyppisen merkkijonohaun monen hahmon hakuun. Käsittelemme tässä Set Horspool -algoritmia [17], joka on yksinkertaistettu versio Commentz-Walter-algoritmista. Algoritmi siis lukee kohdistuksen merkkejä oikealta vasemmalle ja raportoi löydetyt esiintymät. Kohdistuksen tutkiminen lopetetaan, kun on varmaa, että tässä kohdistuksessa ei ole enää esiintymiä. Tämän jälkeen kohdistusta siirretään hyppytaulun perusteella Boyer–Moore–Horspool-algoritmin tapaan.

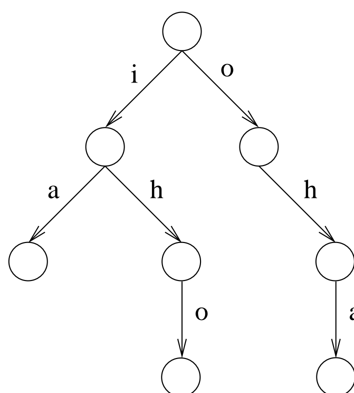
Tunnistaakseen hahmojen esiintymät Set Horspool -algoritmi muodostaa trien käännetyistä hahmoista. Kuvassa 6 on tällainen trie hahmoista ”aho”, ”ai” ja ”ohi”. Tämän lisäksi hahmojen perusteella muodostetaan hyppytaulu, johon tallennetaan kullekin aakkoston merkille minimietäisyys jonkin hahmon lopusta merkin esiintymään hahmon pisimmässä aidossa alkuosassa, tai lyhyimmän hahmon pituus, jos tämä on pienempi. Hyppytauluun tallennettu arvo on siis minimi kaikkien yksittäisten hahmojen Boyer–Moore–Horspool-algoritmin hyppytaulujen arvoista.

Kun nämä rakenteet on muodostettu, etenee varsinainen hakuvaihe siis seuraavasti. Olkoon $m$ lyhyimmän hahmon pituus. Algoritmi aloittaa lukemalla tekstin merkkejä paikasta $m$ taaksepäin ja kulkien triessä näiden perusteella. Kun triessä ei enää voida edetä luetulla merkillä, siirretään lukemisen aloittamispaikkaa eteenpäin hyppytaulun mukaisesti. Suurilla hahmojoukoilla hyppyjä voidaan pidentää muodostamalla hyppytaulu $q$-piirteille.

### 4.3 SBOM

SBOM [2] on lyhenne englanninkielen sanoista Set Backward Oracle Matching. Algoritmi siis käsittelee joukkoa hahmoja, lukee hahmojen kanssa kohdistetun tekstin taaksepäin ja käyttää täsmäämiseen *oraakkeliksi* kutsuttua automaattia.

Esikäsittelynä SBOM muodostaa au-




**Kuva 6:** Käännetyistä hahmoista ”aho”, ”ai” ja ”ohi” muodostettu trie.

tomaatin, joka tunnistaa ainakin kaikkien käännettyjen hahmojen kaikki osajonot. Tätä automaattia voidaan käyttää merkkijonohakuun samaan tapaan kuin BNDM-algoritmin automaattia. Olkoon $m$ taas lyhimmän hahmon pituus. Aloitetaan lukemalla tekstiä paikasta $m$ taaksepäin ja etenemällä automaatissa vastaavasti. Aina, kun automaatti on tilassa, joka vastaa kokonaan luettua hahmoa, tarkastetaan potentiaalinen esiintymä vertaamalla tekstin merkkejä hahmon merkkeihin. Jos tekstistä luetaan merkki, jolla ei ole siirtymää automaatin aktiivisesta tilasta, siirretään tutkittavaa kohtaa eteenpäin $m-j+1$ paikkaa, missä $j$ on luettujen merkkien määrä, tai yksi paikka, jos $m-j+1<1$.

Automaatti muodostetaan kuvan 7 algoritmin mukaisesti. Aluksi muodostetaan trie käännetyistä hahmoista. Tämän jälkeen trietä käydään läpi leveysjärjestyksessä lisäten kaaria siten, että kaikki osajonot tunnistetaan. Määritellään tätä varten kullekin solmulle $v$ *syötesolmu* $S(v)$. Solmun $v$ *nimi* on merkkijono, joka saadaan yhdistämällä lopullisessa automaatissa lyhyimmällä polulla juuresta solmuun $v$ esiintyvät merkit. Syötesolmu $S(v)$ on se solmu, jonka nimi saadaan poistamalla solmun $v$ nimestä ensimmäinen merkki. Trien juurella ei ole syötesolmua. Solmun $v$ läpikäynnin yhteydessä lisättävät kaaret saadaan nyt kulkemalla syötesolmuketjua pitkin, kunnes saavutaan juureen tai solmuun, josta on kaari samalla merkillä $c$ kuin solmuun $v$ tuleva kaari. Muista läpikäydyistä solmuista lisätään kaari solmuun $v$ merkillä $c$.

Näin saatu automaatti saattaa tunnistaa joitakin muitakin merkkijonoja kuin hahmojen osajonoja kuten kuvan 8 automaatti osoittaa. Tästä huolimatta sitä voidaan käyttää täsmäykseen, joskin automaatin raportoimat esiintymät pitää tarkastaa erikseen.

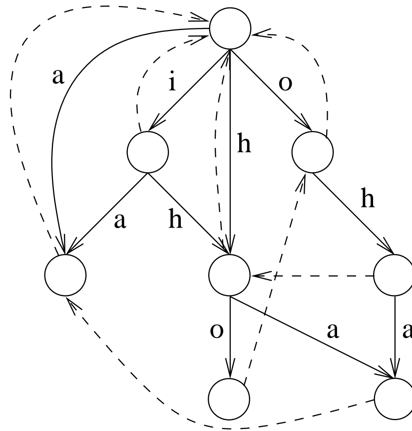
## 5 Suodatusmenetelmät

Monet tehokkaat merkkijonomenetelmät käyttävät jotain nopeaa tapaa suodattaa pois suurin osa tekstin paikoista potentiaalisten esiintymien joukosta. Tässä luvussa esittelemme suodatukseen perustuvia menetelmiä monen hahmon hakuun.

Koska suodatusmenetelmät raportoivat myös sellaisia paikkoja, joissa ei oikeasti ole esiintymää, täytyy näiden menetelmien palauttamat esiintymät tarkastaa. Tähän voidaan käyttää mitä tahansa edellä esitettyä menetelmää. Monet suodatusmenetelmät kuitenkin käyttävät yksinker-

```
1:  rakenna-trie(P_1^R, ..., P_r^R)
2:  S(juuri) ← NULL
3:  kaikille trien solmuille v paitsi juurelle leveysjärjestyksessä do
4:      u ← S(solmun v isäsolmu)
5:      c ← solmun v isäsolmusta solmuun v johtavan kaaren merkki
6:      while u ≠ NULL ja solmusta u ei ole kaarta merkillä c do
7:          lisää kaari solmusta u solmuun v merkillä c
8:          u ← S(u)
9:      if u ≠ NULL then
10:         S(v) ← solmun u lapsi, jonka kaaren merkki on c
11:     else
12:         S(v) ← juuri
```

**Kuva 7:** SBOM-algoritmin esiprosessointi hahmoille $P_1, \ldots, P_r$. Merkinnällä $P_i^R$ tarkoitetaan käännettyä hahmoa.




**Kuva 8:** Hahmoista ”aho”, ”ai” ja ”ohi” muodostettu SBOM-automaatti. Kaaret kunkin solmun syötesolmuun on merkitty katkoviivalla.

taisempia ja tilatehokkaampia ratkaisuja, koska vääriä esiintymiä on usein hyvin vähän. Eräs mahdollinen tarkastusmenetelmä on laskea hahmon merkeistä jokin hajautusarvo ja tallentaa hahmot tämän hajautusarvon mukaisessa järjestyksessä. Tämän jälkeen esiintymät voidaan tarkastaa laskemalla potentiaalisesta esiintymästä sama hajautusarvo ja hakemalla puolitushaulla hahmot, jotka tarkastetaan merkki kerrallaan potentiaalista esiintymää vastaan.

Kaikissa tässä esitetyissä suodatusmenetelmissä oletetaan, että hahmot ovat samanpituisia. Mikäli näin ei ole, voidaan menetelmiä soveltaa käyttämällä suodatusvaiheessa kustakin hahmosta lyhyimmän hahmon pituista alkuosaa, mutta vertaamalla tarkastusvaiheessa koko hahmoa potentiaaliseen esiintymään tekstissä.

### 5.1 Rabin–Karp

Rabin–Karp-algoritmi [12] esitettiin alunperin yhden hahmon hakuun. Ideana on laskea hahmon merkeistä hajautusarvo ja käydä sitten teksti paikka paikalta läpi laskien sama hajautusarvo ja vertaamalla tätä hahmon hajautusarvoon. Löydetyt potentiaaliset esiintymät on tietenkin tarkastettava. Hajautusarvo on mahdollista määritellä siten, että tekstin seuraavan paikan hajautusarvo voidaan laskea vakioajassa edellisen hajautusarvon ja seuraavan merkin perusteella.

Tämä suodatusmenetelmä voidaan yleistää usealle hahmolle [8, 13, 23]. Tällöin luodaan bittitaulu, joka on hajautusfunktion arvojoukon kokoinen. Bittitaulussa on ykkönen, jos kyseinen arvo on jonkin hahmon hajautusarvo, ja nolla muuten. Sitten lasketaan hajautusarvo kullekin tekstin paikalle ja katsotaan taulusta, voisiko jokin hahmo esiintyä kyseisessä paikassa. Jos näin on, kyseinen esiintymä tarkastetaan.

### 5.2 Wu–Manber

Wu–Manber-algoritmi [21] on Boyer–Moore–Horspool-algoritmin yleistys monelle hahmolle. Algoritmi käyttää kahta hajautustaulua $q$-piirteille. Ensimmäistä taulua käytetään siirtymien laskemiseen ja toista kohdistuksessa potentiaalisesti esiintyvien hahmojen tunnistamiseen.

Ensimmäinen taulu on *hyppytaulu*, joka on yleistys Boyer–Moore–Horspool-algoritmin hyppytaulusta monelle hahmolle ja $q$-piirteille. Hyppytauluun tallennetaan siis kullekin $q$-piirteelle lyhyin etäisyys jonkin hahmon lopusta $q$-piirteen esiintymään kyseisen hahmon $m-1$-pituisessa alkuosassa tai $m-q+1$, jos $q$-piirre ei esiinny minkään hahmon aidossa alkuosassa. Usein käytetään $q$-piirteen hajautusarvoa taulun indeksointiin, jolloin useampi $q$-piirre voi osua samaan taulun alkioon.

Algoritmin käyttämä toinen hajautustaulu on *esiintymätaulu*. Sinne tallennetaan kullekin mahdolliselle $q$-piirteen hajautusarvolle lista hahmoista, jonka ensimmäisellä $q$-piirteellä on kyseinen hajautusarvo.

Hakuvaiheessa näitä tauluja käytetään seuraavasti. Algoritmi kohdistaa hahmot ensin tekstin alun kanssa. Kohdistuksen lopusta luetaan $q$-piirre ja tarkastetaan ensin hyppytaulusta, esiintyykö tämä $q$-piirre jossain hahmossa. Jos näin on, luetaan kohdistuksen ensimmäinen $q$-piirre ja haetaan esiintymätaulusta tarkastettavat hahmot. Kohdistusta verrataan merkki kerrallaan kuhunkin tarkastettavaan hahmoon ja raportoidaan esiintymät. Lopuksi siirretään kohdistusta hyppytaulusta saadun arvon mukaan.

Esiintymätaulua siis käytetään tässä esiintymien tarkastukseen. Algoritmia on tietenkin mahdollista muokata käyttämään myös jotain muuta tarkastusmene-

telmää. Agrep-työkalu [22] käyttää Wu–Manber-algoritmia monen hahmon hakuun.

### 5.3 Merkkiluokkamenetelmiä $q$-piirteillä

Merkkiluokkamenetelmät [19] muodostavat koko hahmojoukosta *yleistetyn hahmon*, jossa kuhunkin paikkaan liitetään merkkiluokka. Merkki sallitaan tietyssä paikassa, jos se esiintyy jossain hahmossa tässä paikassa. Esimerkiksi hahmoista ”aho” ja ”ohi” muodostettu yleistetty hahmo on [a,o][h][i,o]. Huomaamme, että jos tekstissä on jomman kumman hahmon esiintymä, myös yleistetty hahmo täsmää. Toisaalta yleistetty hahmo voi täsmätä, vaikka mikään hahmojoukon hahmoista ei täsmäisi. Esimerkiksi ”oho” täsmää esimerkkimme yleistettyyn hahmoon, vaikka se ei ole kumpikaan hahmoista.

Varsinaiseksi hakualgoritmiksi kelpaa mikä tahansa yhden hahmon hakuun tarkoitettu algoritmi, joka osaa käsitellä merkkiluokkia. Esimerkiksi bittirinnakkaiset algoritmit shift-or ja BNDM soveltuvat hyvin merkkiluokkia sisältävien hahmojen hakuun. Ainoa tarvittava muutos on taulun $B$ alustuksessa. Shift-or-algoritmissa taulun alkio $B[c]$ alustetaan nyt siten, että $i$:s bitti on nolla, jos ja vain jos merkki $c$ sisältyy $i$:nnen paikan merkkiluokkaan. Vastaavasti BNDM-algoritmin taulun alkio $B[c]$ alustetaan siten, että $i$:s bitti on ykkönen, jos ja vain jos merkki $c$ sisältyy käännetyn hahmon $i$:nnen paikan merkkiluokkaan.

Jos hahmoja on paljon, yllä kuvattu menetelmä on tällaisenaan tehoton, koska lähes jokainen aakkoston merkki esiintyy jokaisessa paikassa jossain hahmossa. Jos käytetään $q$-piirteitä yksittäisten merkkien sijaan, menetelmä toimii myös suurilla hahmomäärillä. Tehokkainta on käyttää limittäisiä $q$-piirteitä, jolloin hahmosta ”aho” ja ”ohi” saadaan esimerkiksi 2-piirteet ”ah-ho” ja ”oh-hi”. Yleistetyksi hahmoksi tulee nyt [ah,oh][hi,ho].

Jotta yleistetty hahmo olisi riittävän valikoiva, käytännössä riittää valita $q$ siten, että erilaisia $q$-piirteitä on enemmän kuin hahmoja. Toisaalta $q$ ei saa olla liian suuri, jotta algoritmien tilantarve ei kasva liian suureksi.

Näistä aineksista saadaan monen hahmon hakualgoritmi shift-or $q$-piirteillä (SOG). Aluksi siis muodostetaan hahmoista $q$-piirteitä käyttävä yleistetty hahmo, sitten haetaan tätä yleistettyä hahmoa shift-or-algoritmilla ja lopuksi tarkastetaan tämän palauttamat esiintymät. Seuraavan luvun kokeellisia tuloksia varten olemme toteuttaneet tarkastuksen käyttämällä Rabin–Karp-menetelmästä tuttua hajautusta ja puolitushakua.

Vastaavasti muodostetaan monen hahmon hakualgoritmi BNDM $q$-piirteillä (BG). Tässäkin algoritmissa aluksi muodostetaan hahmoista $q$-piirteitä käyttävä yleistetty hahmo, yleistettyä hahmoa haetaan BNDM-algoritmilla ja lopuksi saadut esiintymät tarkastetaan. Myös tähän algoritmiin tarkastus on toteutettu Rabin–Karp-menetelmän hajautuksella ja puolitushaulla.

BG-algoritmia käytetään MPSCAN-työkalussa [18] lyhyiden DNA-jonojen hakemiseen genomisesta datasta. Lisäksi BG-algoritmia on käytetty osavaiheiden toteutuksessa genomin kokoamistyökalussa PMSGA [14] ja paikallisten rinnastusten etsimisessä GAST-työkalussa [11].

Viimeiseksi esittelemme Boyer–Moore–Horspool-algoritmiin perustuvan menetelmän, Horspool $q$-piirteillä (HG). Esikäsittelynä alustetaan kullekin mahdolliselle $q$-piirteelle bittivektori, jossa $i$:s bitti on ykkönen, jos ja vain jos kyseinen $q$-piirre esiintyy jossain hahmossa paikassa $i$ tai sitä ennen. Alla on esitetty nä-

mä bittivektorit hahmoille ”aho” ja ”ohi” käyttäen 2-piirteitä:

| 2-piirre | bittivektori |
|---|---|
| ah | 11 |
| hi | 10 |
| ho | 10 |
| oh | 11 |
| ... | 00 |

Hakuvaiheessa kohdistusta luetaan oikealta vasemmalle $q$-piirre kerrallaan. Olkoon viimeinen luettu $q$-piirre $i$:s kohdistuksen alusta lukien. Jos $i$:s bitti on nolla tätä $q$-piirrettä vastaavassa bittivektorissa, kyseinen $q$-piirre ei esiinny missään hahmossa tässä paikassa tai aikaisemmin. Tällöin mikään hahmoista ei voi esiintyä tässä kohdistuksessa eikä missään sellaisessa kohdistuksessa, joka on oikealle tästä kohdistuksesta, mutta sisältää luetun $q$-piirteen. Näin ollen kohdistusta voidaan siirtää tämän $q$-piirteen yli. Muuten jatketaan lukemalla seuraava $q$-piirre. Jos koko kohdistus on luettu, tarkastetaan, onko löydetty oikea esiintymä, ja tämän jälkeen kohdistusta siirretään yhdellä paikalla eteenpäin.

## 6 Kokeellisia tuloksia

Testasimme edellä esitettyjä algoritmeja kahdella eri tekstillä. Kokeet ajettiin tietokoneella, jossa on 1,0 GHz AMD Athlon prosessori ja 2 GB muistia. Koneen käyttöjärjestelmänä oli Linux 2.6.23. Algoritmit toteutettiin C-kielellä ja käännettiin gcc-kääntäjällä.

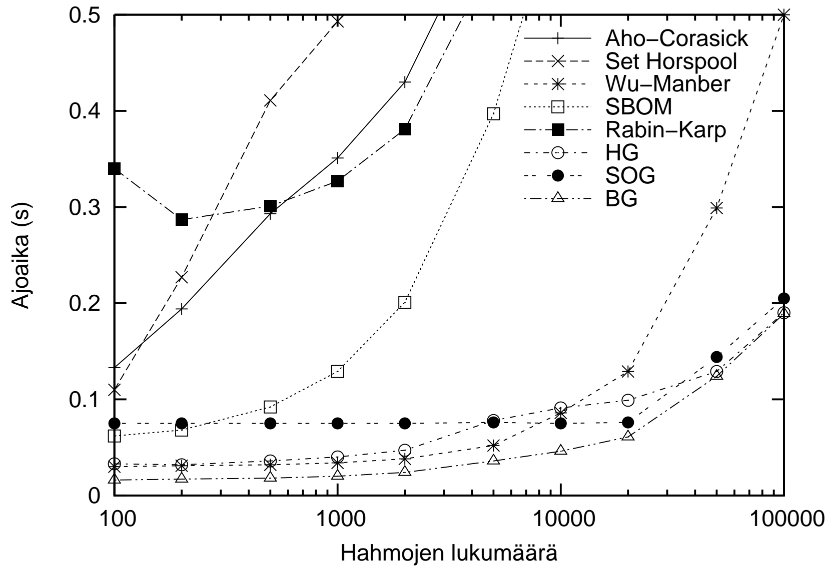
Ensimmäinen teksti on tasaisesti ja riippumattomasti jakautunut merkkijono, jonka aakkoston koko on 256 ja pituus 32 MB. Myös hahmot tuotettiin satunnaisesti samasta aakkostosta. Kunkin hahmon pituus on kahdeksan. Kuvassa 9 on esitetty tämän kokeen tulokset. Ajoajat ovat keskiarvoja kymmenestä toistosta samalla tekstillä ja hahmoilla ja ne eivät sisällä esikäsittelyaikoja. Kokeen tuloksista nähdään, että triemenetelmät hidastuvat nopeasti, kun hahmojoukko kasvaa. Myös Aho–Corasick-algoritmin ajoaika kasvaa, vaikka teoriassa se ei riipu haettavien hahmojen määrästä. Tämä johtunee trien vaatiman tilan kasvusta. Tässä kokeessa BG-algoritmi oli nopein.

Toisena tekstinä käytimme banaanikärpäsen yhtä kromosomia, jonka pituus on 22 MB. Hahmot luotiin taas satunnaisesti käyttäen neljän merkin DNA-aakkostoa. Tässä kokeessa hahmojen pituus on 32. Toteutuksemme Set Horspool -algoritmista ei käytä $q$-piirteitä, joten näin pienellä aakkostolla se ei ole kilpailukykyinen ja se on jätetty vertailusta pois. Käyttämämme Wu–Manber-algoritmin toteutus taas perustuu agrep-työkaluun, jota ei ole tarkoitettu näin pienelle aakkostolle, joten myöskään sitä ei ole testattu tällä aineistolla. Tämän kokeen tulokset on esitetty kuvassa 10. Nämä tulokset ovat hyvin samankaltaisia kuin 256 merkin aakkostolla, mutta nyt HG-algoritmi on nopein.

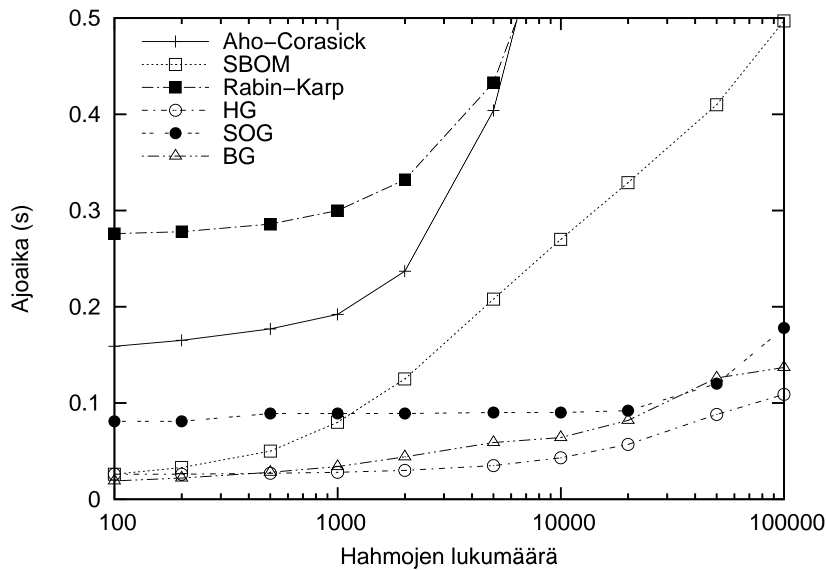
## 7 Yhteenveto

Olemme esitelleet monen hahmon merkkijonohakuun lukuisia algoritmeja klassisesta Aho–Corasick-algoritmista uusiin suodatusmenetelmiin. Esitellyt algoritmit jaettiin kahteen luokkaan, joista ensimmäinen käyttää jonkinlaista trie-tyyppistä tietorakennetta hahmojoukon indeksointiin ja toinen luokka suodattaa nopeasti monet tekstin paikoista pois potentiaalisten esiintymien joukosta.

Koeasetelmassamme suodatusmenetelmät olivat triemenetelmiä tehokkaampia. Kokeissamme kaikki hahmot olivat samanpituisia ja melko pitkiä, mistä kokeissa menestyneet algoritmit selvästi hyötyivät, koska ne pystyivät jättämään suuria osia tekstistä lukematta. Jos hahmojoukossa olisi ollut hyvin lyhyitä




**Kuva 9:** Monen hahmon hakualgoritmien vertailu tekstillä, jonka aakkoston koko on 256.




**Kuva 10:** Monen hahmon hakualgoritmien vertailu DNA-tekstillä.

hahmoja, lineaariaikainen Aho–Corasick-algoritmi olisi ollut huomattavasti kilpailukykyisempi.

## Kiitokset

Kiitokset Jorma Tarhiolle monista parannusehdotuksista.

## Viitteet


1. A.V. Aho ja M.J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.
2. C. Allauzen ja M. Raffinot. Oracle des facteurs d'un ensemble de mots. Tekninen raportti 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.
3. R. Baeza-Yates ja G.H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.
4. D. Belazzougui. Succinct dictionary matching with no slowdown. *Proceedings of the 21st Annual Symposium on Combinatorial Pattern Matching (CPM'10)*, *LNCS* 6129, ss. 88–100. Springer-Verlag, 2010.
5. R.S. Boyer ja J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
6. B. Commentz-Walter. A string matching algorithm fast on the average. *Proceedings of the 6th Colloquium on Automata, Languages and Programming (ICALP'79)*, *LNCS* 71, ss. 118–132. Springer-Verlag, 1979.
7. K. Fredriksson. Succinct backward-DAWG-matching. *ACM Journal of Experimental Algorithmics*, 13(1.8):1–26, 2009.
8. B. Gum ja R. Lipton. Cheaper by the dozen: Batched algorithms. *Proceedings of the 1st SIAM International Conference on Data Mining (SDM'01)*, 2001.
9. W.K. Hon, R. Shah, J.S. Vitter, T.W. Lam ja S.L. Tam. Compressed index for dictionary matching. *Proceedings of the Data Compression Conference*, ss. 23–32. IEEE, 2008.
10. R.N. Horspool. Practical fast searching in strings. *Software – Practise and Experience*, 10(6):501–506, 1980.
11. K. Karhu, J. Mäkinen, J. Rautio, H. Salamon ja J. Tarhio. GAST, a genomic alignment search tool. *Proceedings of 2nd International Conference on Bioinformatics*. INSTICC, 2011.
12. R.M. Karp ja M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
13. R. Muth ja U. Manber. Approximate multiple string search. *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*, *LNCS* 1075, ss. 75–86. Springer-Verlag, 1996.
14. J. Mäkinen, J. Tarhio ja S. Khuri. PMSGA: A fast DNA fragment assembler. *Proceedings of BIOINFORMATICS 2010, First International Conference on Bioinformatics*, ss. 77–82. INSTICC, 2010.
15. V. Mäkinen. Itseindeksit – kun tiivistetty teksti ja sen indeksi ovatkin sama asia. *Tietojenkäsittelytiede*, 25:28–37, 2006.
16. G. Navarro ja M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics* 5(4):1–36, 2000.
17. G. Navarro ja M. Raffinot. *Flexible Pattern Matching in Strings: Practical Online Search Algorithms for Text and Biological Sequences*. Cambridge University Press, 2002.
18. E. Rivals, L. Salmela, P. Kiiskinen, P. Kalsi ja J. Tarhio. MPSCAN: fast localisation of multiple reads in genomes. *Proceedings of the 9th Workshop on Algorithms in Bioinformatics (WABI'09)*, *LNCS* 5724, ss. 246–260. Springer-Verlag, 2009.
19. L. Salmela, J. Tarhio ja J. Kytöjoki. Multipattern string matching with $q$-grams. *ACM Journal of Experimental Algorithmics*, 11(1.1):1–19, 2006.
20. A. Tam, E. Wu, T.-W. Lam ja S.-M. Yiu, Succinct text indexing with wildcards. *Proceedings of the 16th International Symposium on String Processing and In-*

*formation Retrieval*, *LNCS* 5721, ss. 39–50. Springer-Verlag, 2009.

21. S. Wu ja U. Manber. A fast algorithm for multi-pattern searching. Tekninen raportti TR-94-17, Department of Computer Science. University of Arizona, 1994.
22. S. Wu ja U. Manber. Agrep – a fast approximate pattern-matching tool. *Proceedings of the Usenix Winter 1992 Technical Conference*, ss. 153–162. 1992.
23. R.F. Zhu ja T. Takaoka. A technique for two-dimensional pattern matching. *Communications of the ACM*, 32(9):1110–1120, 1989.