

# Itseindeksit — Kun tiivistetty teksti ja sen indeksi ovatkin sama asia

Veli Mäkinen  
Helsingin yliopisto  
Tietojenkäsittelytieteen laitos  
vmakinen@cs.helsinki.fi

## Tiivistelmä

Tekstimuotoisen tiedon talletuksessa käytetään yleisesti kahta tapaa: (1) talletetaan teksti sellaisenaan tai (2) talletetaan teksti pakattuna. Ensimmäisessä vaihtoehdossa etuna on nopea pääsy mielivaltaiseen kohtaan tekstissä, haittana tilan tuhlaus. Toisessa vaihtoehdossa menetetään nopea pääsy tekstin sisälle mutta saavutetaan tilan säästöä. Herää kysymys: olisiko mahdollista saavuttaa molemmat edut yhtä aikaa?

Vastaus on kyllä. *Itseindeksit* yhdistävät tiivistämisen ja nopean pääsyn tekstin osiin. Lisäksi ne tuovat mukanaan kokonaan uuden edun: niiden avulla voidaan tehokkaasti selvittää, esiintyykö annettu hakusana tekstissä. Tällaisiin hakuihin osataan vastata oleellisesti lineaarisessa ajassa *hakusanan* pituuden suhteen. Tämä on huomattava parannus vaihtoehtoon (1), missä vastaavaan kyselyyn vastaaminen vie lineaarisen ajan *tekstin* pituuden suhteen.

Tässä artikkelissa esitellään itseindeksien perusideat ja luodaan katsaus aihealueen viimeisimpiin tuloksiin. Perusideat esitellään kuvaamalla eräs yksinkertainen itseindeksi sillä tasolla, että ohjelmointitaitoinen henkilö voi kuvauksen perusteella sen pienellä vaivalla toteuttaa.

Yleisesti tekstimuotoinen tieto mielletään koostuvaksi jonosta luonnollisen kielen sanoja. Luonnollisen kielen tekstit ovat sekä tiivistämisen että tiedonhaun kannalta erikoisasemassa. Esimerkiksi Internet hakukoneet perustuvat ns. *käänteisindeksien* käyttöön, joissa jokaiselle sanalla löytyy lista dokumentteja, joissa sana esiintyy. Dokumentit itsessään voidaan säilyttää tiivistettynä ja purkaa tarvittaessa.

Laskennallinen biologia on tuonut mukanaan läpäirteiden tekstien käsitte-

lyyn. Biologiset sekvenssit eivät koostu sanoista vaan ovat yhtenäisiä merkkijonoja ilman välimerkkejä. Tiedonhaussa sekvenssien kaikki osajonot pitää ottaa huomioon.

Biologisten sekvenssien määrä kasvaa nopeasti. Nykyään osataan sekvenssoida lähes systemaattisesti eri eliölajien genomeja. Tietokoneiden tallennuskapasiteetti kasvaa kuitenkin vielä nopeammin eikä tuskin tule tämän kehityksen esteeksi. Entäpä jos tulevaisuudessa opitaan sekvenssoimaan eri yksilöiden genomeja? Mihin

talletetaan kuusi miljardia ihmisen genomia?

Jos tallennuskapasiteetti ei nyt tulekaan heti ongelmaksi sekvenssejä käsitellessä muuta kuin epärealistisissa skenaarioissa, niin miten hallitaan kasvavan sekvenssimassan analysointi? Vaikkakin nykyisin käytössä olevat likimääräisiä samankaltaisuuksia etsivät algoritmit ovat entisiä tehokkaampia, ne kuitenkin lukevat koko sekvenssin alusta loppuun suorittaen samalla raskasta laskentaa. Tällainen laskenta saattaa muuttua pullonkaulaksi sekvenssitietokantojen kasvaessa.

Likimääräisessä hahmonsovituksessa on jo pitkään tunnettu tehokas ratkaisu sekvenssien analyysin nopeuttamiseksi: *indeksointi*. Sen sijaan että sekvenssit luetaisiin kerta toisensa jälkeen läpi analysointialgoritmeissa, niille voidaan esiprosessoida tietorakenne, ns. *loppuosaindeksi*, jota käyttäen analysointi tapahtuu käytämällä vain oleelliset osat sekvenssistä. Varjopuolena loppuosaindeksien käytössä on niiden viemä tila. Esimerkiksi ihmisen genomille tällainen rakenne voi viedä 60 gigatavua tilaa. Tämä tila on vieläpä oltava saatavilla keskusmuistissa, sillä muutoin mitään tehokkuushyötyä ei saada aikaan.

Viime vuosien tutkimus indeksirakenteiden saralla on kulminoitunut käänteentekevään keksintöön: on onnistuttu muodostamaan ns. *itseindeksi*, jolla on samat hakuja nopeuttavat ominaisuudet kuin klassisilla loppuosaindekseillä ja joka samalla korvaa alkuperäisen sekvenssin. Vielä merkittävämpää on se, että on onnistuttu muodostamaan tiivistetty versio itseindeksistä, joka vie lähes yhtä vähän tilaa, kuin mihin parhaat tunnetut klassiset tiivistysmenetelmät pystyvät (esim. zip). Toisin sanoen nykyään osataan esittää annettu sekvenssi tiivistettynä siten, että sen sisältöä voidaan hakea purkamatta koko

esitystä.

Tiivistetyt itseindeksit voivat tulevaisuudessa mullistaa sekvenssien talletuksen, sillä ne tarjoavat ilmaiseksi uuden ominaisuuden: tiivistystulos pysyy samana kuin ennen, mutta levyille tallennettu sekvenssi ei olekaan pelkkä bittijono vaan välittömästi hyödynnettävissä oleva monipuolinen indeksi.

Muuttuakseen käyttökelpoisemmiksi tiiviit itseindeksit tarvitsevat vielä lisäominaisuuksia. Nykyään ne tukevat lähinnä vain tarkkaa hakua, mutta likimääräisten samankaltaisuuksien hakuun on jo ehdotettu alustavia ratkaisuja. Käytännön tehokkuus ei ole vielä verrattavissa vastaaviin tiivistämättömiin indekseihin. Monia kysymyksiä on vielä ratkaisematta ennenkuin teoreettinen läpimurto muuttuu teknologiseksi innovaatioksi. Alueen nopea kehitys ennakoi kuitenkin menestystä tälläkin saralla.

Tässä artikkelissa esitellään itseindeksien perusideoiden valottamiseksi eräs yksinkertainen itseindeksi. Alan kirjallisuudessa esiintyvät indeksit ovat tiiviimpiä, tehokkaampia ja monipuolisempia mutta jonkin verran monimutkaisempia kuin nyt esiteltävä. Artikkelin lopussa on katsaus itseindeksien syntyyn johtaneisiin edistysaskeliin. Viitteet kirjallisuuteen on kerätty sinne.

## 1 Burrows-Wheeler-muunnos

Itseindeksien ytimen muodostaa ns. *Burrows-Wheeler-muunnos*. Tämä muunnos on merkkijonon permutaatio, jonka kääntäminen takaisin alkuperäiseksi merkkijonoksi onnistuu helposti. Muunnoksella on myös muita yllättäviä ominaisuuksia, kuten permutoidun jonon hyvä tiivistyvyys, sekä hakujen mahdollistumi-

nen alkuperäisen jonon sisältöön.

Käytetään esimerkkinä merkkijonoa  $T = \text{vesihiisi\#}$ , missä  $\#$  on teknisistä syistä lisättävä lopetusmerkki. Tämä merkki oletetaan aakkosjärjestyksessä pienemmäksi kuin muut merkit. Kuvassa 1 on esitetty Burrows-Wheeler-muunnos,  $T^{bwt}$ , esimerkkipijonolle  $T$ .

Muunnos saadaan järjestämällä syötemerkkijonon  $T = t_1t_2 \cdots t_n$  kaikki *sykliset kierrot*  $\{t_1t_2 \cdots t_n, t_nt_1t_2 \cdots t_{n-1}, t_{n-1}t_nt_1t_2 \cdots t_{n-2}, \dots, t_2t_3 \cdots t_nt_1\}$  aakkosjärjestykseen. (Tässä kukin  $t_i$  on aakkoston  $\Sigma$  merkki,  $T \in \Sigma^*$ .) Kuvassa 1 vasemmalla on visualisoitu esimerkkipijonon kaikki sykliset kierrot, keskellä nämä kierrot aakkosjärjestyksessä. Huomaa, että loppumerkin jälkeiset (harmaalla piirretyt) merkit eivät vaikuta aakkosjärjestykseen. Näin syntyvän matriisin  $M$  viimeinen sarake  $L$  vastaa muunnosta  $T^{bwt}$ . Kuvan tapauksessa  $L = T^{bwt} = \text{ivisshiie\#}$ .

Matriisi  $M$  on muunnoksen muodostuksessa täysin virtuaalinen; olennaista ovat sen ensimmäinen sarake  $F$  ja viimeinen sarake  $L$ . Itse asiassa muunnoksen kääntäminen (alkuperäisen tekstin palauttaminen) onnistuu sarakkeiden  $L$  ja  $F$  välisen yhteyden avulla: huomataan, että sarakkeen  $L$  merkkien järjestäminen tuottaa sarakkeen  $F$ . Talletetaan taulukkoon  $LF[1 \dots n]$  järjestämisen määräämä kuvaus sarakkeelta  $L$  sarakkeelle  $F$ . Tässä on oleellista käyttää stabiilia järjestämistä, eli toissijaisena avaimena järjestämisessä käytetään merkin indeksia sarakkeella. Kuvan 1 tapauksessa kuvaukseksi  $LF$  saadaan

|    |   |    |   |   |   |   |   |   |   |    |
|----|---|----|---|---|---|---|---|---|---|----|
| i  | 1 | 2  | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| LF | 4 | 10 | 5 | 8 | 9 | 3 | 6 | 7 | 2 | 1  |

$LF$ -kuvauksen tulkinta on seuraava: Olkoon matriisin  $M$  rivillä  $i$  syklinen kierto  $fXl$ , missä  $f, l \in \Sigma$  ja  $X \in \Sigma^*$ . Tällöin

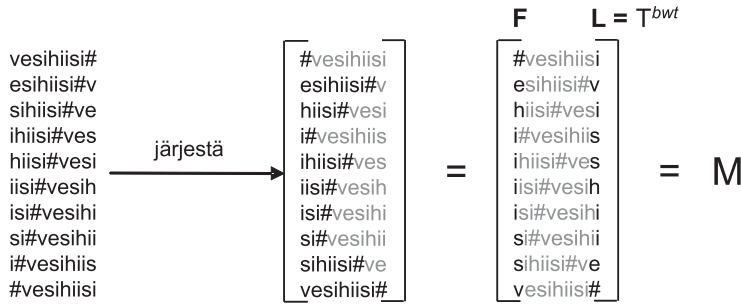
syklinen kierto  $lfX$  esiintyy matriisin rivillä  $LF[i]$ . Esimerkiksi kuvassa 1 syklinen kierto  $iisi\#vesih$  esiintyy rivillä 6 ja  $hiisi\#vesi$  rivillä  $LF[6] = 3$ . Yhteys on helppo todistaa formaalisti.

Alkuperäisen tekstin palauttaminen on nyt helppoa. Esimerkin jono on  $L[10]L[LF[10]]L[LF[LF[10]]] \cdots L[LF^9[10]] = L[10]L[1] \cdots L[2] = \text{\#isiihisev}$  kääntäen, missä  $LF^i$  tarkoittaa kuvauksen  $LF$  soveltamista  $i$  kertaa rekursiivisesti. Tässä aloitusindeksi 10 tiedetään, koska se on ainoa matriisin  $M$  rivi, joka loppuu merkkiin  $\#$ . Algoritmi seuraa peräkkäisiä syklisiä kiertoja paljastamalla kussakin vaiheessa aina merkin, joka kiertyy seuraavan syklisen kierron alkuun; tuloksena on alkuperäinen teksti käännettynä.

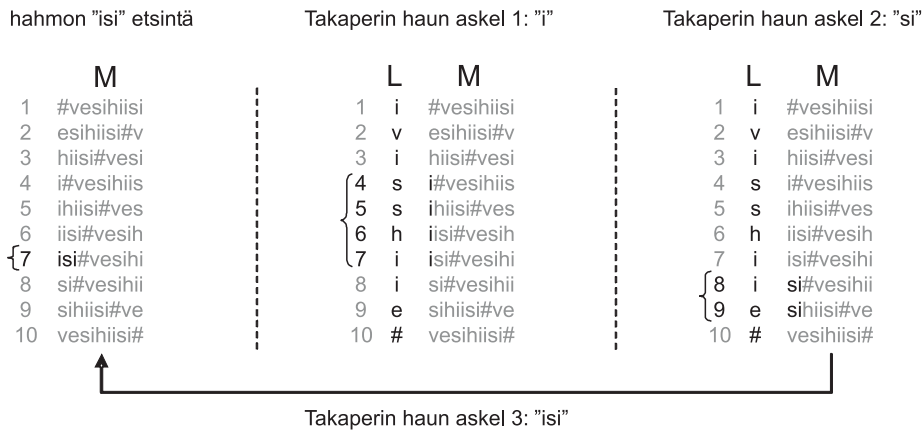
## 2 Takaperinhaku

Edellä esitetty *käänteinen Burrows-Wheeler-muunnos* ei ole ainoa  $LF$ -kuvauksen sovellus. Samaa ideaa voidaan hyödyntää hahmonetsinnässä tekstistä: Yksittäisen syklisen kierron sijasta seurataan kussakin askeleessa joukkoa syklisiä kiertoja. Pidetään yllä invarianttia, että seurattavan joukon viimeiset merkit täsmäävät tarkasteltavaan hahmon merkkiin. Kun tarkasteltavaa hahmon merkkiä askeletaan hahmon lopusta alkuun, saadaan lopulta tulokseksi joukko syklisiä kiertoja, joiden alkuosat täsmäävät koko hahmoon. Kuvassa 2 havainnollistetaan *takaperinhaun* vaiheita etsittäessä hahmon  $i$  esiintymiä tekstissä  $\text{vesihiisi\#}$ .

Kuvan 2 ensimmäinen askel etsii hahmon viimeisen merkin  $i$  esiintymät matriisin  $M$  ensimmäisellä sarakkeella  $F$ . Näin löydetään siis kaikki sykliset kierrot, jotka alkavat merkillä  $i$ . Toisin sanoen, löydetään kaikki neljä merkin  $i$  esiintymää tekstissä  $\text{vesihiisi\#}$ . Tämä askel on helppo toteuttaa tallettamalla tauluk-



Kuva 1: Burrows-Wheeler-muunnoksen laskenta.



Kuva 2: Hahmon isi etsintä tekstistä vesihiisi# käyttäen takaperinhakua.

koon  $C[1 \dots |\Sigma|]$  kullekin aakkoston mer- kille tieto siitä, montako kertaa sitä aak- kosjärjestyksessä pienemmät merkit esiin- tyvät tekstissä. Ensimmäisessä askelees- sa haettava osaväli on silloin  $[C[4] + 1, C[5]] = [4, 7]$ , sillä  $i$  on aakkoston  $\Sigma = \{\#, e, h, i, s, v\}$  neljänneksi pienin mer- ki, sitä pienemmät merkit esiintyvät kukin vain kerran tekstissä ja  $i$  esiintyy 4 kertaa, eli  $C[4] = 3$  ja  $C[5] = C[4] + 3 = 7$ .

Kuvan 2 toinen askel etsii kaik- kia syklisiä kiertoja, jotka alkavat hah- mon kahdella viimeisellä merkillä, eli merkkijonolla  $si$ . Etsintä perustuu seu- raavaan huomioon: Kierrettäessä rivien  $[4, 7]$  merkkijonoja oikealle saadaan sykli-

set kierrot  $si\#vesihii$ ,  $sihiisi\#ve$ ,  $hiisi\#vesi$  ja  $iisi\#vesih$ . Nämä ovat ainoat sykliset kierrot, joiden toinen merkki on  $i$  ensimmäisen askeleen perus- teella. Täten ainoat sykliset kierrot, jotka alkavat merkkijonolla  $si$  ovat näistä  $ne$ , jotka alkavat merkillä  $s$ . Olkoon  $i$  ja  $j$  en- simmäinen ja viimeinen merkin  $s$  esiin- tymä sarakkeen  $L$  riveillä  $[4, 7]$ . Huoma- taan, että  $[LF[i], LF[j]]$  antaa toisessa as- keleessa etsittävän välin:  $[LF[4], LF[5]] = [8, 9]$ . Tämä johtuu siitä, että samalla mer- killä (tässä merkillä  $s$ ) loppuvat merkkijo- not säilyttävät aakkosjärjestyksensä kier- rätessä oikealle; merkin  $j$ :nnes esiinty- mä sarakkeella  $L$  kuvautuu sen  $j$ :nneksi

esiintymäksi sarakkeella  $F$ .

Kuvan 2 kolmas askel toimii samoin kuin toinen. Uudeksi riviväliksi saadaan  $[LF[8], LF[8]]$ , koska hahmon ensimmäinen merkki  $i$  esiintyy vain kerran toisessa askeleessa löydetyllä välillä  $[8, 9]$ .

Takaperinhaun kussakin vaiheessa pitää siis löytää indeksit  $i$  ja  $j$ , missä  $L[i]$  ja  $L[j]$  ovat annetun hahmon merkin  $c$  ensimmäinen ja viimeinen esiintymä edellisessä vaiheessa saavutetussa rivivälissä  $L[sp, ep]$ . Tämä tarvitaan uuden rivivälin  $[LF[i], LF[j]]$  laskentaan. Tämän hankalan laskennan sijasta voidaan hyödyntää yhteyttä

$$LF[i] = C[L[i]] + rank_{L[i]}(L, i), \quad (1)$$

missä funktio  $rank_c(L, i)$  kertoo, montako kertaa merkki  $c$  esiintyy sarakkeella  $L$  kohtaan  $i$  mennessä. Taulukko  $C$  on sama, mitä käytetään haun ensimmäisessä askeleessa. Yhtälön oikeellisuus perustuu aiemmin tehtyyn huomioon syklisten kiertojen käyttäytymisestä aakkosjärjestyksessä. Yhtälön hyöty piilee siinä, että takaperin haun aikana pätee  $L[i] = c$ , missä  $c$  on hahmon vuorossa oleva merkki, sekä  $rank_{L[i]}(L, i) = rank_c(L, sp - 1) + 1$ . Tämä siksi, että jonossa  $L[sp, i - 1]$  ei ole yhtään merkin  $c = L[i]$  esiintymää, koska  $L[i]$  on se ensimmäinen välillä  $[sp, ep]$ . Samoin saadaan  $rank_{L[j]}(L, j) = rank_c(L, ep)$ , koska  $L[j]$  on merkin viimeinen esiintymä välillä  $[sp, ep]$ . Toisin sanoen, indeksejä  $i$  ja  $j$  ei tarvitse tietää.

Saadaan yksinkertainen pseudokoodi takaperinhaululle, ks. Kuva 3.

### 3 Takaperinhausta itseindeksiin

Kuten edellisessä luvussa huomattiin, takaperinhaku ei käytä alkuperäistä tekstiä lainkaan, joten se voidaan unohtaa;

tarvitaan vain taulukko  $C$ , funktio  $rank$  ja Burrows-Wheeler-muunnos  $L$ . Niiden avulla voidaan helposti simuloida käänteistä Burrows-Wheeler-muunnosta, joten alkuperäinen teksti voidaan palauttaa tarvittaessa. Taulukko  $C$  ei vie paljoa tilaa pienillä aakkostoilla ( $|\Sigma| \log n$  bittiä), joten se voidaan tässä unohtaa. Funktion  $rank$  laskentaan on useita vaihtoehtoja. Esimerkiksi  $rank$  ja  $L$  voidaan sulauttaa yhteiseksi tietorakenteeksi.

Hyvin pienillä aakkostoilla varteenotettava vaihtoehto funktion  $rank$  laskemiseksi on seuraava: talletetaan joka  $\alpha$ :s  $rank_c(L, i)$  arvo sellaisenaan kullekin  $c \in \Sigma$ . Tällöin mielivaltaiseen  $rank_c(L, i)$ -kyselyyn voidaan vastata  $O(\alpha)$  ajassa laskemalla naiivisti merkin  $c$  esiintymät lähimmän talletetun arvon ja kyselykohdan välissä. Tilaa absoluuttisen arvojen tallettamiseen käytetään silloin  $(n|\Sigma|/\alpha) \log n$  bittiä. Valitsemalla esimerkiksi  $\alpha = |\Sigma| \log n$ , ylimääräistä tilaa kuluu  $n$  bittiä. Kun itse merkkijonon  $L$  tallennus vie saman tilan kuin alkuperäisen tekstin tallennus eli  $n \log |\Sigma|$  bittiä, on kokonaistilavaatimus  $n \log |\Sigma| + n$  bittiä. Takaperinhaussa tehdään  $2m$   $rank$ -kutsua, missä  $m$  on etsittävän hahmon pituus. Kokonaisaikaavaatimukseksi saadaan  $O(m|\Sigma| \log n)$ .

Edellä kuvattu ratkaisu mahdollistaa vain koko tekstin palauttamisen tehokkaasti, sekä hahmon esiintymien lukumäärän laskemisen. Tähän on kuitenkin helppo lisätä päälle ominaisuuksia, jotta lopputulosta voidaan kutsua itseindeksiksi. Tärkein luvattu ominaisuus on tehokas pääsy tekstin osajonoihin. Tähän päästään tallettamalla joka  $\beta$ :nnelle tekstin kohdalla linkki sitä vastaavaan riviin matriisissa  $M$  (indeksiin sarakkeella  $L$ ). Kun halutaan saada selville osajono  $T[i \dots j]$ , etsitään indeksiä  $j$  lähin sitä seuraava indeksi, jolle on talletettu linkki sarakkeelle  $L$ . Simuloimalla  $LF$ -kuvausta saadaan luettua os-

---

**Algoritmi** Takaperinhaku( $P, m, L, n, C, rank$ )

```

(1)  $sp \leftarrow 1; ep \leftarrow n;$ 
(2) for  $i \leftarrow m$  to 1
(3)    $sp \leftarrow C[p_i] + rank_{p_i}(L, sp - 1) + 1;$ 
(4)    $ep \leftarrow C[p_i] + rank_{p_i}(L, ep);$ 
(5)   if  $sp > ep$  then return  $\emptyset;$ 
(6)    $i \leftarrow i - 1;$ 
(7) return  $[sp, ep];$ 

```

---

Kuva 3: Takaperinhaun algoritmi etsii hahmon  $P$  esiintymät tekstissä käyttäen tekstille laskettua taulukkoa  $C$  ja funktiota  $rank$ . Tuloksena on riviväli matriisissa  $M$ . Käytännössä matriisia  $M$  ei ole talletettuna, joten tuloksesta voidaan päätellä vain esiintymien lukumäärä  $ep - sp + 1$ .

ajono  $T[i \dots j]$  käänteisesti ja korkeintaan  $\beta$  ylimääräistä merkkiä. Käyttämällä nyt  $\beta = \log n$ , osajono  $T[i \dots j]$  saadaan purettua  $O((j - i + 1 + \log n)|\Sigma| \log n)$  ajassa. Ylimääräistä tilaa käytetään  $n$  bittiä, joten kokonaistilavaatimukseksi saadaan  $n \log |\Sigma| + 2n$  bittiä.

### 3.1 Aakkostoriippuvuuden vähentäminen

Saavutetuissa aikavaatimuksissa esiintyvä  $|\Sigma|$  termi on helppo muuntaa  $\log^2 |\Sigma|$  termiksi vaikuttamatta tilavaatimukseen. Tämä onnistuu käyttämällä *hierarkkista aakkoston ryhmittelyä*: Muodostetaan tasapainoinen puu, jossa lehtinä ovat aakkoston merkit. Tällöin kukin sisäsolmu  $v$  vastaa osajoukkoa aakkostosta. Merkitään tätä osajoukkoa  $\Sigma_v$  solmulle  $v$ . Esiitetään merkkijono  $L$  puun avulla seuraavasti. Puun juureen  $v$  talletetaan bittivektori  $B_v[1 \dots n]$ , jossa  $B_v[i] = 0$  jos  $L[i] \in \Sigma_v$ , missä  $\ell$  on juuren vasen lapsi. Muutoin  $B_v[i] = 1$ , eli  $L[i] \in \Sigma_r$ , missä  $r$  on juuren oikea lapsi. Voidaan ajatella, että  $L$  jakautuu kahdeksi alijonoksi: jonoksi  $L_\ell$ , jossa on yhteenliitetty merkit  $L[i]$  joissa  $B_\ell[i] = 0$ , ja jonoksi  $L_r$ , jossa on yhteenliitetty merkit  $L[i]$  joissa  $B_\ell[i] = 1$ . Talletetaan juuren vasempaan lapseen bit-

tivektori  $B_\ell[1 \dots |L_\ell|]$  ja oikeaan lapseen bittivektori  $B_r[1 \dots |L_r|]$ . Nämä täytetään vastaavasti kuin juuren bittivektori katsomalla, mihin suuntaan aakkoston merkit haarautuvat solmussa. Näin menetellään rekursiivisesti, kunnes saavutaan lehtisolmuihin, joihin tarvitsee tallettaa vain tieto aakkoston merkistä. Kuvassa 4 on esitetty hierarkkinen puurakenne esimerkin Burrows-Wheeler-muunnokselle  $L = ivisshie\#$ . Vain bittivektorit ja puun rakenne talletetaan; solmujen merkkijonot tarvitaan vain muodostusvaiheessa.

Yllä esitettyä hierarkkista puurakennetta voidaan käyttää muuntamaan  $rank_c(L, i)$  kysely ( $\log |\Sigma|$ ):ksi binäärijonon  $rank$ -kyselyksi. Katsotaan ensin, miten puurakenteen avulla saadaan selville merkki  $L[i]$  annettuna indeksi  $i$ . Idea selviää tutkimalla kuvan 4 esimerkkijonon kohtaa  $i = 7$ . Juureen liittyvän bittivektorin seitsemäs bitti on 1, joten vastaava merkki on kopioitu juuren oikeaan lapseen. Vastaava kohta juuren oikeassa lapsessa löytyy laskemalla  $rank_1(B, 7) = 6$ , missä  $B$  on juuren bittivektori. Tämä johtuu siitä, että 1-bittien määrä indeksiin 7 mennessä kertoo, kuinka monta merkkiä oikeaan lapseen on kopioitu ennen nykyistä indeksistä. Koska juuren oikean lapsen bittivektorin kohdassa 6 on bitti 0, on

kyseinen merkki kopioitu nykyisen solmun vasempaan lapseen. Tämä lapsisolmu on lehti, ja siihen talletettu merkki on  $i$ . Täten  $L[7] = i$ . Huomataan myös, että laskemalla  $rank_0(B_r, 6) = 3$ , missä  $B_r$  on juuren oikean lapsen bittivektori, saadaan selville  $rank_{\perp}(L, 7) = 3$ . Yleisessä tapauksessa  $rank_c(L, i)$  kyselyyn vastaaminen eroaa merkin  $L[i]$  selvittämisestä, koska  $c$  voi olla mielivaltainen merkki. Katsotaan esimerkkinä kyselyyn  $rank_{\perp}(L, 6)$  vastaamista. Nyt huomataankin, että merkki  $i$  on suurempi kuin aakkoston keskiarvo, joten sen esiintymät löytyvät juuren oikeasta alipuusta. Merkki  $L[6]$  on kuitenkin kopioitu vasemmalle, joten sitä ei voi seurata. Sen sijaan siirtymällä indeksiin  $rank_1(B, 6) = 5$  juuren oikean alipuun bittivektorissa, saavutetaan turvallinen siirto, koska tällöin seurataan viimeistä juuren indeksiä 6 edeltävää merkkiä, joka on siirtynyt oikealle. Tällöin ei siis voida ohittaa yhtään merkin  $i$  esiintymää. Oikeassa alipuussa huomataan, että etsittävä merkki  $i$  kuuluu nykyisen solmun vasempaan alipuuhun. Samalla päättelyllä kuin juuressa voidaan todeta, että  $rank_0(B_r, 5) = 2$  antaa turvallisen siirtymän nykyisen solmun vasempaan lapseen. Koska tämä lapsi onkin lehtisolmu, on näin laskettu tulos sama kuin etsitty  $rank_{\perp}(L, 6) = 2$ .

Hierarkksessa puurakenteessa tarvitaan vain  $rank_1(B, i)$  kyselyitä. (Huomaa, että  $rank_0(B, i) = i - rank_1(B, i)$ .) Rakenteessa on joka tasolla  $n$  bittiä, ja tasoja on  $\log |\Sigma|$ . Tallettamalla joka  $\alpha = ((\log |\Sigma|) \log n)$ :s vastaus  $rank_1$ -kyselyihin sellaisenaan, kokonaistilavaatimus on  $n \log |\Sigma| + n$  bittiä eli sama kuin aikaisemmin. Merkin  $L[i]$  selvittäminen, sekä kyselyyn  $rank_c(L, i)$  vastaaminen vie nyt ajan  $(\log^2 |\Sigma|) \log n$ . Toisin sanoen,

hahmon esiintymien lukumäärän laskenta vie ajan  $O(m(\log^2 |\Sigma|) \log n)$ . Tekstin osajonon  $T[i \dots j]$  tulostaminen vie ajan  $O((j - i + 1 + \log n)(\log^2 |\Sigma|) \log n)$ , kun käytetään  $n$  ylimääräistä bittiä kuten aiemmin on kuvattu.

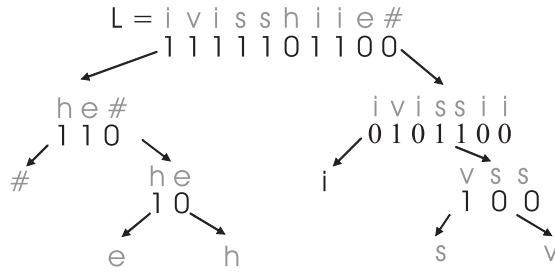
## 4 Kirjallisuusyhteenveto

Edellä kuvattu itseindeksi on yksinkertaistettu versio kirjallisuudessa esiintyvistä ratkaisuksista. Todellisuudessa indeksin ominaisuuksia ja tila- sekä aikavaativuutta voidaan parantaa huomattavasti — vieläpä kaikkia yhtä aikaa. Tässä luvussa kuvataan, miten kirjallisuudessa esitetyt ratkaisut liittyvät edellä esitettyyn ja miten niistä saadaan tehokkaampia ratkaisuja esitettyihin aliongelmiin.

Tietorakenteiden tiivistyksen (toiminnallisuuden säilyttävässä mielessä) voidaan katsoa alkaneen G. Jacobsonin väitöskirjatutkimuksesta [8]. Yhtenä tuloksena hänen työssään on bittivektorin  $rank$ -operaation laskenta vakioajassa käyttäen  $o(n)$  bittiä tilaa bittivektorin itsensä vaatiman  $n:n$  bitin lisäksi.<sup>1</sup> Tulosta käyttäen voidaan välittömästi parantaa edellisessä luvussa esiintyvät  $(\log^2 |\Sigma|) \log n$ -termit  $(\log |\Sigma|)$ -termeiksi, sillä  $rank$ -operaation aikavaatimus paranee näin  $O((\log |\Sigma|) \log n)$ :sta vakioaikaan. Samalla tilavaatimus pienenee luokkaan  $n \log |\Sigma| (1 + o(1))$  bittiä ( $+n$  bittiä, mikäli halutaan tukea tekstin osajonojen tulostamista tehokkaasti).

Takaperinhaun mahdollistavan muunnoksen kehittivät Burrows ja Wheeler [1]. Tästä muunnoksesta tuli pian suosittu, koska sen päälle voitiin rakentaa tehokkaita tekstin tiivistysmenetelmiä. Muunnoksen tärkein ominaisuus onkin teks-

<sup>1</sup>Tulos olettaa RAM-laskennan mallin. Käytännössä riittää olettaa, että enintään  $(\log n) : n$  mittainen bititijono voidaan lukea vakioajassa ja muuttaa vastaavaksi kokonaislukuarvoksi. Tämä (luonnollinen) oletus tehdään kaikissa tässä luvussa esiteltävissä tuloksissa.



Kuva 4: Hierarkkinen puurakenne tekstin vesihiisi# Burrows-Wheeler-muunnokselle  $L = ivisshiie\#$ .

tin kontekstien kerääminen samaan paikkaan (esimerkiksi the sanaa edeltävät välilyönnit kerääntyvät muunnoksessa lähekkäin), jolloin paikalliseen merkkijakaumaan mukautuvat tiivistysmenetelmät tuottavat automaattisesti hyvän tuloksen. Muunnoksen yhteyttä korkeamman asteen entropiaan on tutkittu myös teoreettisesti [10], ja tämän tutkimuksen yhtenä uutena mielenkiintoisena tuloksena on *tiivistyksen kiihdyttäminen*: Burrows-Wheeler-muunnoksen avulla voidaan kiihdyttää mielivaltaisen nolllannen asteen entropiakoodauksen tehoa siten, että saavutetaan korkeamman asteen entropia (entropiat määritellään tulevassa kappaleessa). Tällainen kiihdytys osataan vieläpä tehdä lineaarisessa ajassa [4].

Toinen muunnoksen tärkeä ominaisuus on sen muodostamisen ja muunnoksen palauttamisen tehokkuus. Muunnos voidaan lukea helposti, kun ensin muodostetaan ns. *loppuosataulukko* (tekstin loppuosien järjestetty esitys). Tämän taulukon muodostamiseen on kehitetty useita tehokkaita algoritmeja; läpimurto aiheessa koettiin kesällä 2003, kun yhtä aikaa julkaistiin kolme erilaista lineaariaikaista (optimaalista) muodostusalgoritmia. Näistä Kärkkäisen ja Sanderssin kehittämä algoritmi [9] on noussut suosituimmaksi sen yllättävän yksinkertaisuuden ja käytännön tehokkuuden vuoksi.

Burrows ja Wheeler eivät kuitenkaan vielä kehittäneet takaperinhakua, vaan tämän keksinnön tekivät Ferragina ja Manzini [3, 4]. He esittivät ensimmäisenä myös itseindeksien idean ja myös konkreettisen ratkaisun, ns. *FM-indeksin*. Alkuperäisessä FM-indeksissä oli kuitenkin valitettavia piirteitä, kuten hyvin voimakkaasti kasvava aakkoston suhteen (tuplasti) eksponentiaalinen tilavaatimus. Tulos oli käytännöllinen vain hyvin pienillä aakkostoilla. Kuitenkin yhdistämällä osan heidän ratkaisustaan Jacobsonin *rank*-rakenteeseen, saadaan aikaan jo hyvin käytännöllinen ratkaisu.

Grossi, Gupta ja Vitter [6] tekivät seuraavan tärkeän keksinnön itseindeksien käytännöllisyyden kannalta; he kehittivät edellä kuvatun hierarkkisen aakkoston esityksen. Tätä rakennetta kutsutaan *wavelet*-puuksi. Itse asiassa, he pysyivät hyödyntämään Raman, Raman ja Raon [12] kehittämää, aikaisempaa tilatehokkaampaa, *rank*-ratkaisua wavelet-puun osana. Tuloksena on rakenne, joka vie tilaa suhteessa tekstin kokeelliseen nolllannen asteen entropiaan,  $H_0(T)$  (keskimääräinen yhden merkin koodaukseen tarvittava bittien määrä, kun merkin todennäköisyys on sen frekvenssi tekstissä). Käyttämällä tätä rakennetta takaperinhausassa, saadaan  $nH_0(T) + o(n \log |\Sigma|)$  tilainen itseindeksi, joka tukee muun muas-



sa hahmon esiintymien lukumäärän las-  
kenta ajassa  $O(m \log |\Sigma|)$ . Grossi, Gupta  
ja Vitter käyttivät kuitenkin erilaista haku-  
mekanismia, joten heidän tuloksena poik-  
keaa hieman tästä. Wavelet-puun kehittä-  
misen lisäksi he osoittivat, että itseindek-  
sien tilavaatimusta voidaan edelleen pie-  
nentää viemään tilaa suhteessa korkeam-  
man asteen entropiaan.

Viimeisimpiä tuloksia itseindeksien  
saralla on tiivistyksen kiihdyttämisen ja  
(tehostetun) wavelet-puun yhdistäminen  
takaperinhausassa [5]. Tämä itseindeksi vie  
tilaa  $nH_k(T) + o(n \log |\Sigma|)$  bittiä ja tukee  
takaperinhakua ajassa  $O(m)$ , mikäli aak-  
koston koko on  $O(\text{poly} \log(n))$ . Tässä  
 $H_k(T)$  on tekstin  $T$  kokeellinen  $k$ -nnen as-  
teen entropia (keskimääräinen yhden mer-  
kin koodaukseen tarvittava bittien määrä,  
kun merkin todennäköisyys on sen fre-  
kvenssi sitä edeltävässä  $k$ :n merkin kon-  
tekstissa tekstissä).

Saavutettu teoreettinen tilavaativuus  
 $nH_k(T) + o(n \log |\Sigma|)$  vastaa käytännössä  
tiivistystä, joka on saavutettavissa ylei-  
sesti käytetyillä tekstintiiivistysohjelmilla,  
kuten zip ja bzip2 (niiden tilavaatimus  
ei sisällä alilineaarista termiä  $o(n \log |\Sigma|)$ ,  
mutta niiden entropiasta riippuvassa osas-  
sa on suurempi vakiokerroin). Täten mo-  
dernit itseindeksit pääsevät lähes samaan  
tiivistystulokseen kuin pelkkään tiivistyk-  
seen erikoistuneet menetelmät ja tuottavat  
lisäetuna pääsyn tekstin osiin sekä mah-  
dollisuuden hahmonetsintään suoraan tii-  
vistetystä esityksestä. Näin ollen tiivistet-  
ty teksti ja sen indeksi ovatkin sama asia  
[7].

Ongelmakohtia itseindekseissä ovat  
monimutkaisten tietorakenteiden ja al-  
goritmien tuoma toteutuskyynnys, vakio-  
tekijät aikavaativuuksissa, vakiotekijät  
alilineaarisisissa tilavaativuuustermeissä,  
muodostusaika/-tila, hakujen toteuttami-  
nen toissijaisessa muistissa, soveltuminen

likimääräisiin hakuihin, jne. Osiin näis-  
tä ongelmakohtista löytyy jo ratkaisuja  
kirjallisuudesta.

Lisätietoa alan tämän hetkises-  
tä tilasta sekä tarkempi ja katta-  
vampi otos itseindeksien kuvauksia  
löytyy tuoreesta kooste-artikkelista  
[11]. Itseindekseistä löytyy myös ko-  
koelma valmiita kirjasto-toteutuksia:  
<http://pizzachili.dcc.uchile.cl/>.

## Viitteet

- [1] M. Burrows ja D. Wheeler. A block  
sorting lossless data compression algo-  
rithm. Tekninen raportti 124, Digital  
Equipment Corporation, 1994.
- [2] P. Ferragina, R. Giancarlo, G. Manzini ja  
M. Sciortino. Boosting textual compres-  
sion in optimal linear time. *Journal of  
the ACM*, 52(4):688-713, 2005.
- [3] P. Ferragina ja G. Manzini. Oppor-  
tunistic data structures with applica-  
tions. Teoksessa *FOCS'00*, sivut 390–  
398, 2000.
- [4] P. Ferragina ja G. Manzini. Indexing  
compressed texts. *Journal of the ACM*,  
52(4):552–581, 2005.
- [5] P. Ferragina, G. Manzini, V. Mäkinen ja  
G. Navarro. Compressed representation  
of sequences and full-text indexes. *ACM  
Transactions on Algorithms*, 2006. Jul-  
kaistavana.
- [6] R. Grossi, A. Gupta ja J. Vitter. High-  
order entropy-compressed text indexes.  
Teoksessa *SODA'03*, sivut 841–850,  
2003.
- [7] R. Grossi, A. Gupta ja J. Vitter. When  
indexing equals compression: Experi-  
ments with compressing suffix arrays  
and applications. Teoksessa *SODA'04*,  
sivut 636–645, 2004.

- [8] G. Jacobson. *Succinct Static Data Structures*. Väitöskirja, CMU-CS-89-112, Carnegie Mellon -yliopisto, Yhdysvallat, 1989.
- [9] J. Kärkkäinen ja P.Sanders. Simple linear work suffix array construction. Teoksessa *ICALP '03*. LNCS 2719, Springer, sivut 943–955, 2003.
- [10] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
- [11] G. Navarro ja V. Mäkinen. Compressed Full-Text Indexes (2nd version). Tekninen raportti TR/DCC-2006-6, Tietojenkäsittelytieteen laitos, Chilen yliopisto, Santiago, Huhtikuu 2006. <ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/survcompr2.ps.gz>
- [12] R. Raman, V. Raman ja S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding  $k$ -ary trees and multisets. Teoksessa *SODA'02*, sivut 233–242, 2002.