Tietojenkäsittelytiede **25** Joulukuu 2006 sivut 28–37
Toimittaja: Jorma Tarhio

Itseindeksit — Kun tiivistetty teksti ja sen indeksi ovatkin sama asia

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

Tiivistelmä

Tekstimuotoisen tiedon tallennuksessa käytetään yleisesti kahta tapaa: (1) tallennetaan teksti sellaisenaan tai (2) tallennetaan 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 lisäpiirteen tekstien käsittelyyn. 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 sekvensoida lähes systemaattisesti eri eliölajien genomeja. Tietokoneiden tallennuskapasiteetti kasvaa kuitenkin vielä nopeammin eikä tuskin tule tämän kehityksen esteeksi. Entäpä jos tulevaisuudessa opitaan sekvensoimaan eri yksilöiden genomeja? Mihin

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.

tallennetaan 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 luettaisiin kerta toisensa jälkeen läpi analysointialgoritmeissa, niille voidaan esiprosessoida tietorakenne, ns. *loppuosaindeksi*, jota käyttäen analysointi tapahtuu käyttämällä 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 tallennuksen, sillä ne tarjoavat ilmaiseksi uuden ominaisuuden: tiivistystulos pysyy samana kuin ennen, mutta levylle 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, ennen kuin 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 ja hakujen mahdollistuminen alkuperäisen jonon sisältöön.

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

Muunnos saadaan järjestämällä syötemerkkijonon $T = t_1t_2\cdots t_n$ kaikki *sykliset kierrot* $\{t_1t_2\cdots t_n, t_nt_1t_2\cdots t_{n-1}, t_{n-1}t_nt_1t_2\cdots t_{n-2}, \ldots, t_2t_3\cdots t_nt_1\}$ aakkosjärjestykseen. (Tässä kukin t_i on aakkoston Σ merkki, $T \in \Sigma^*$.) Kuvassa 1 vasemmalla on visualisoitu esimerkkijonon 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 $L = T^{bwt} =$ `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. Tallennetaan taulukkoon $LF[1\ldots n]$ järjestämisen määräämä kuvaus sarakkeelta L sarakkeelle F. Tässä on oleellista käyttää stabiilia järjestämistä, eli toissijaisena avaimena järjestämisessä käytetään merkin indeksiä 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
```

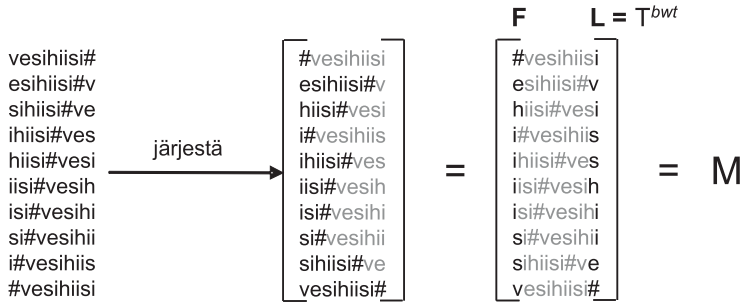
LF-kuvauksen tulkinta on seuraava: Olkoon matriisin M rivillä i syklinen kierto fXl, missä $f,l \in \Sigma$ ja $X \in \Sigma^*$. Tällöin syklinen kierto lfX esiintyy matriisin rivillä $LF[i]$. Esimerkiksi kuvassa 1 syklinen kierto `iisi#vesih` esiintyy rivillä 6 ja `hiisi#vesi` rivillä $LF[6] = 3$. Yhteys on helppo todistaa formaalisti.

Alkuperäisen tekstin palauttaminen on nyt helppoa. Esimerkin jono on $L[10]L[LF[10]]L[LF[LF[10]]]\cdots L[LF^9[10]] = L[10]L[1]\cdots L[2] =$ `#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.

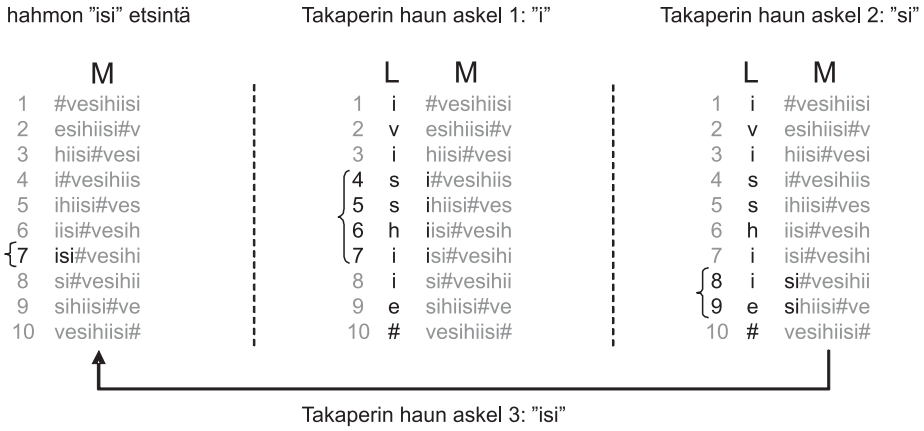
2 Takaperinhaku

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, jossa seurattavan joukon viimeiset merkit täsmäävät tarkasteltavaan hahmon merkkiin. Kun tarkasteltavaa hahmon merkkiä askelletaan hahmon lopusta alkuun, saadaan lopulta tulokseksi joukko syklisiä kiertoja, joiden alkuosat täsmäävät koko hahmoon. Kuvassa 2 havainnollistetaan *takaperinhaun* vaiheita etsittäessä hahmon `isi` esiintymiä tekstissä `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 `vesihiisi#`. Tämä askel on helppo toteuttaa tallentamalla taulukkoon $C[1\ldots|\Sigma|]$ kullekin aakkoston mer-



Kuva 1: Burrows–Wheeler-muunnoksen laskenta.



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

kille tieto siitä, montako kertaa sitä aakkosjärjestyksessä pienemmät merkit esiintyvät tekstissä. Ensimmäisessä askeleessa haettava osaväli on silloin $[C[4]+1, C[5]] = [4,7]$, sillä `i` on aakkoston $\Sigma = \{\#, e, h, i, s, v\}$ neljänneksi pienin merkki, 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 kaikkia syklisiä kiertoja, jotka alkavat hahmon kahdella viimeisellä merkillä eli merkkijonolla `si`. Etsintä perustuu seuraavaan huomioon: kierrettäessä rivien $[4,7]$ merkkijonoja oikealle saadaan sykliset kierrot `si#vesihii`, `sihiisi#ve`, `hiisi#vesi` ja `iisi#vesih`. Nämä ovat ainoat sykliset kierrot, joiden toinen merkki on `i` ensimmäisen askeleen perusteella. Täten ainoat sykliset kierrot, jotka alkavat merkkijonolla `si` ovat näistä ne, jotka alkavat merkillä `s`. Olkoon i ja j ensimmäinen ja viimeinen merkin `s` esiintymä sarakkeen L riveillä $[4,7]$. Huomataan, että $[LF[i], LF[j]]$ antaa toisessa askeleessa etsittävän välin: $[LF[4], LF[5]] = [8,9]$. Tämä johtuu siitä, että samalla merkillä (tässä merkillä `s`) loppuvat merkkijonot säilyttävät aakkosjärjestyksensä kierrättäessä oikealle; merkin j:nnes esiintymä 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

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

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 takaperinhaulle, 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: tallennetaan joka α: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 tallennetun arvon ja kyselykohdan välissä. Tilaa absoluuttisen arvojen tallentamiseen 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. Kokonaisaikavaatimukseksi saadaan $O(m|\Sigma| \log n)$.

Edellä kuvattu ratkaisu mahdollistaa vain koko tekstin palauttamisen tehokkaasti ja 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 tallentamalla joka β:nnelle tekstin kohdalla linkki sitä vastaavaan riviin matriisissa M (indeksiin sarakkeella L). Kun halutaan saada selville osajono $T[i \ldots j]$, etsitään indeksiä j lähin sitä seuraava indeksi, jolle on tallennettu linkki sarakkeelle L. Simuloimalla LF-kuvausta saadaan luettua osajono $T[i \ldots j]$ käänteisesti ja korkein-

Algoritmi Takaperinhaku(P, m, L, n, C, *rank*)
(1) $sp \leftarrow 1$; $ep \leftarrow n$;
(2) **for** $i \leftarrow m$ **to** 1
(3) $\quad sp \leftarrow C[p_i] + rank_{p_i}(L, sp-1) + 1$;
(4) $\quad ep \leftarrow C[p_i] + rank_{p_i}(L, ep)$;
(5) $\quad$ **if** $sp > ep$ **then return** $\emptyset$;
(6) $\quad i \leftarrow i - 1$;
(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 tallennettuna, joten tuloksesta voidaan päätellä vain esiintymien lukumäärä $ep - sp + 1$.

taan β ylimääräistä merkkiä. Käyttämällä nyt $\beta = \log n$ osajono $T[i \ldots j]$ 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 Σ_v solmulle v. Esitetään merkkijono L puun avulla seuraavasti. Puun juureen v tallennetaan bittivektori $B_v[1 \ldots n]$, jossa $B_v[i] = 0$ jos $L[i] \in \Sigma_\ell$, missä ℓ on juuren vasen lapsi. Muutoin pätee $B_v[i] = 1$ eli $L[i] \in \Sigma_r$, missä r on juuren oikea lapsi. Voidaan ajatella, että L jakautuu kahdeksi alijonoksi: jonoksi L_ℓ, jossa on yhteenliitetty merkit $L[i]$ joissa $B_\ell[i] = 0$, ja jonoksi L_r, jossa on yhteenliitetty merkit $L[i]$ joissa $B_\ell[i] = 1$. Tallennetaan juuren vasempaan lapseen bittivektori $B_\ell[1 \ldots |L_\ell|]$ ja oikeaan lapseen bittivektori $B_r[1 \ldots |L_r|]$. Nämä täytetään vastaavasti kuin juuren bittivektori katsomalla, mihin suuntaan aakkoston merkit haarautuvat solmussa. Näin menetellään rekursiivisesti, kunnes saavutaan lehtisolmuihin, joihin tarvitsee tallentaa vain tieto aakkoston merkistä. Kuvassa 4 on esitetty hierarkkinen puurakenne esimerkin Burrows–Wheeler-muunnokselle $L =$ `ivisshiie#`. Vain bittivektorit ja puun rakenne tallennetaan; 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 indeksiä. Koska juuren oikean lapsen bittivektorin kohdassa 6 on bitti 0, on kyseinen merkki kopioitu nykyisen solmun

vasempaan lapseen. Tämä lapsisolmu on lehti, ja siihen tallennettu merkki on `i`. Täten $L[7] = \texttt{i}$. Huomataan myös, että laskemalla $rank_0(B_r, 6) = 3$, missä B_r on juuren oikean lapsen bittivektori, saadaan selville $rank_{\texttt{i}}(L, 7) = 3$. Yleisessä tapauksessa $rank_c(L, i)$ kyselyyn vastaaminen eroaa merkin $L[i]$ selvittämisestä, koska c voi olla mielivaltainen merkki. Katsotaan esimerkkinä kyselyyn $rank_{\texttt{i}}(L, 6)$ 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 $rank_{\texttt{i}}(L, 6) = 2$.

Hierarkkisessa puurakenteessa tarvitaan vain $rank_1(B, i)$ kyselyä. (Huomaa, että $rank_0(B, i) = i - rank_1(B, i)$.) Rakenteessa on joka tasolla n bittiä, ja tasoja on $\log|\Sigma|$. Tallentamalla 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 $T[i \ldots j]$ tulostaminen vie ajan $O((j - i + 1 + \log n)(\log^2|\Sigma|)\log n)$, kun käytetään n ylimääräistä bittiä kuten aiemmin on kuvattu.

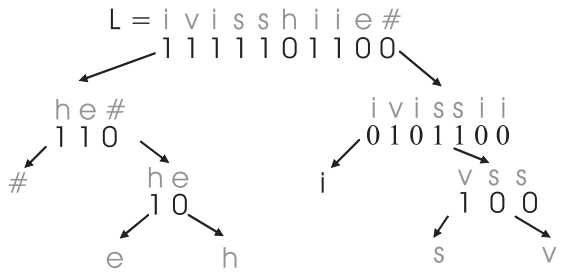
4 Kirjallisuusyhteenveto

Edellä kuvattu itseindeksi on yksinkertaistettu versio kirjallisuudessa esiintyvistä ratkaisuista. 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.[1] 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 tekstin kontekstien kerääminen samaan paik-

[1] Tulos olettaa RAM-laskennan mallin. Käytännössä riittää olettaa, että enintään $(\log n)$:n mittainen bittijono 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#`.

kaan (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 nollannen asteen entropiakoodauksen tehoa siten, että saavutetaan korkeamman asteen entropia (entropiat määritellään myöhemmässä 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 pystyivät hyödyntämään Raman, Raman ja Raon [12] kehittämää, aikaisempaa tilatehokkaampaa, *rank*-ratkaisua wavelet-puun osana. Tuloksena on rakenne, joka vie tilaa suhteessa tekstin kokeelliseen nollannen asteen entropiaan $H_0(T)$ (keskimääräinen yhden merkin koodaukseen tarvittava bittien määrä, kun merkin todennäköisyys on sen frekvenssi tekstissä). Käyttämällä tätä rakennetta takaperinhaussa saadaan $nH_0(T) + o(n \log |\Sigma|)$ tilainen itseindeksi, joka tukee muun muassa hahmon esiintymien lukumäärän lasken-

taa ajassa $O(m \log |\Sigma|)$. Grossi, Gupta ja Vitter käyttivät kuitenkin erilaista hakumekanismia, joten heidän tuloksena poikkeaa hieman tästä. Wavelet-puun kehittämisen lisäksi he osoittivat, että itseindeksien tilavaatimusta voidaan edelleen pienentää viemään tilaa suhteessa korkeamman asteen entropiaan.

Viimeisimpiä tuloksia itseindeksien saralla on tiivistyksen kiihdyttämisen ja (tehostetun) wavelet-puun yhdistäminen takaperinhaussa [5]. Tämä itseindeksi vie tilaa $nH_k(T) + o(n \log |\Sigma|)$ bittiä ja tukee takaperinhakua ajassa $O(m)$, mikäli aakkoston koko on $O(\texttt{polylog}(n))$. Tässä $H_k(T)$ on tekstin T kokeellinen k-nnen asteen entropia (keskimääräinen yhden merkin koodaukseen tarvittava bittien määrä, kun merkin todennäköisyys on sen frekvenssi sitä edeltävässä k:n merkin kontekstissa tekstissä).

Saavutettu teoreettinen tilavaativuus $nH_k(T) + o(n \log |\Sigma|)$ vastaa käytännössä tiivistystä, joka on saavutettavissa yleisesti käytetyillä tekstintiivistysohjelmilla, kuten `zip` ja `bzip2` (niiden tilavaatimus ei sisällä alilineaarista termiä $o(n \log |\Sigma|)$, mutta niiden entropiasta riippuvassa osassa on suurempi vakiokerroin). Täten modernit itseindeksit pääsevät lähes samaan tiivistystulokseen kuin pelkkään tiivistykseen erikoistuneet menetelmät ja tuottavat lisäetuna pääsyn tekstin osiin sekä mahdollisuuden hahmonetsintään suoraan tiivistetystä esityksestä. Näin ollen tiivistetty teksti ja sen indeksi ovatkin sama asia [7].

Ongelmakohtia itseindekseissä ovat monimutkaisten tietorakenteiden ja algoritmien tuoma toteutuskynnys, vakiotekijät aikavaativuuksissa, vakiotekijät alilineaarisissa tilavaativuustermeissä, muodostusaika/-tila, hakujen toteuttaminen toissijaisessa muistissa, soveltuminen likimääräisiin hakuihin, jne. Osiin näistä ongelmakohdista löytyy jo ratkaisuja kirjallisuudesta.

Lisätietoa alan tämän hetkisestä tilasta sekä tarkempi ja kattavampi otos itseindeksien kuvauksia löytyy tuoreesta koosteartikkelista [11]. Itseindekseistä löytyy myös kokoelma valmiita kirjastototeutuksia: `http://pizzachili.dcc.uchile.cl/`.

Viitteet

[1] M. Burrows ja D. Wheeler. A block sorting lossless data compression algorithm. Tekninen raportti 124, Digital Equipment Corporation, 1994.

[2] P. Ferragina, R. Giancarlo, G. Manzini ja M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688-713, 2005.

[3] P. Ferragina ja G. Manzini. Opportunistic data structures with applications. 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. Julkaistavana.

[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: Experiments 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. *ACM Computing Surveys*, hyväksytty. Aiemmin: 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.