Tietorakenteet ja algoritmit -harjoitustyö toteutusdokumentti

Hannes Ihalainen

February 24, 2018

Contents

1 Johdanto

Projektissa on toteutettu Ukkosen algoritmi suffiksipuun muodostamiseen ja toteutettu Ukkosen algoritmilla toimiva ratkaisu Longest common substring -ongelmaan.

Lisäksi työssä on toteutettu itse Vector (vector.h dynaaminen taulukko), ”FastSet” (fastset.h tietorakenne, jossa on nopea elementin lisäys ja haku) ja Bitset (bitset.h) -tietorakenteet.

2 Ukkosen algoritmin toteutus

Ukkosen algoritmi muodostaa suffiksipuun lineaarisessa ajassa merkkijonon kokoon nähden. Implementaationi Ukkosen algoritmin ytimestä seuraa hyvin tarkasti Ukkosen paperissa pseudokoodilla kuvattuja funktioita, ja saavutettu aikavaatimus on ideaalinen, $O(n)$ aakkoston koon ollessa vakio.

Toteutin Ukkosen algoritmin tallentaen edget eri tavoilla. Kaikki toteutukset ovat merkkijonon pituuden suhteen lineaarisia, mutta niiden nopeus vaihtelee aakkoston koon mukaan.

Tässä taulukko työssä toteutetuista edgejen tallennustavoista. n on merkkijonon pituus ja m aakkoston koko.
(Vector on toteuttamani dynaaminen taulukko. ”FastSet” taas on toteuttamani nopeampia lisäys- ja haku -operaatioita tukeva tietorakenne)

	Vector	**Vector järjest.**	**Taulukko**	**FastSet**
Edgen haku	$O(m)$	$O(\log m)$	$O(1)$	$O(\log^2 m)$
Edgen lisäys	$O(1), O(m)*$	$O(m)$	$O(1)$	$O(\log m), O(\log^2 m)*$
Noden lisäys	$O(1)$	$O(1)$	$O(m)$	$O(1)$
Kok. aikavaatimus	$O(n * m)$	$O(n * m)$	$O(n * m)$	$O(n \log^2 m)$
Kok. tilavaatimus	$O(n)$	$O(n)$	$O(n * m)$	$O(n)$

* Edgen lisäykset Vectoriin ja FastSettiin olisivat teoriassa $O(1)$ ja $(O \log m)$, mutta koska edgen lisäyksen yhteydessä tarkistetaan, onko kyseisellä merkillä alkavaa edgeä jo rakenteessa, aikavaatimukseksi tulee sama kuin edgen haussa.

2.1 Implementaatio-rakenteesta...

Ensimmäisenä toteutettu taulukko-implementaatio eroaa muista siinä, että sen UkkonenTree on template-luokka. (muissa implementaatioissa on taas siitä kopioitu suurin osa puun perusrakenteesta) Kaikki muut versiot implementoivat usuffix.h -headerissa määritellyn UkkonenTree -luokan kukin omalla tavallaan, mikä mahdollistaa mm. eri luokkien testausohjelmien compilaamisen samasta lähdekoodista.

Käytännössä usuffix.h:n toteuttavat implementaatiot kuitenkin käyttävät samaa koodia kaikkialla muualla paitsi struct Noden implementaatiossa. Koodin kopioimisen välttämiseksi laitoin yhteisen lähdekoodin tiedostoon (usuffix.cpp). Toteutus on hieman epäelegantti. Jokaiselle eri implementaatiolla on oma usuffixnode*.cpp -tiedoston, josta includataan usuffix.cpp. Tämä ratkaisu kuitenkin todennäköisesti paras. Toinen vaihtoehto olisi tietysti ollut luoda abstracti Node luokka ja eri luokkia, jotka perivät ja toteuttavat sen. Tämä kuitenkin vaatisi, että UkkonenTree -luokan funktiot jollakin tapaa valitsisivat, mitä Node-luokkaa käytetään, mikä vaatisi myös melko paljon säätöä, ja kun lopullisissa ohjelmissa UkkonenTree kuitenkin käyttää aina samaa Node-luokan toteutusta, tuntuisi turhalta lisätä UkkonenTree -luokkaan tietoa mahdollisuudesta, että Noden implementaatio voisi olla vaihdella.

3 "FastSetin" aikavaatimus

"FastSet" on tietorakenne, jossa säilytetään elementtejä järjestetyissä dynaamisissa taulukoissa (käytän niistä nimitystä vektori), joiden koot ovat 2:n potensseja. (en tiedä, mikä on tämän tietorakenteen oikea nimi, jos sillä on sellainen. Opin sen eräältä kaveriltani erään algoritmileirin yhteydessä). Käytän tästä tietorakenteen vektorijoukosta tässä analyysissä nimitystä vektorisetti. Vektorisetin koolla tarkoitan vektorisetin suurimman vektorin järjestyslukua. (Järjestysluvultaan k:nnen vektorin koon ollessa 2^{k-1}). n elementtiä sisältävän vektorisetin koko on siis $\lfloor \log_2 n \rfloor$. Esimerkiksi 10 elementtiä sisältävässä vektorisetissä olisi kaksi täytettyä vektoria, joiden koot ovat 2 ja 8, ja vektorisetin koko olisi tällöin 4. (väliin jäävät puuttuvat vektorit voidaan ajatella tyhjiksi, jolloin vektorisetissä olisivat vektorit 1_{tyhja}, 2_{taysi}, 3_{tyhja} ja 4_{taysi}.)

Pseudokoodissa merkinnällä $A[i]$ tarkoitetaan vektorisetin A vektoria, jonka koko on 2^i. Tämä vektori saattaa olla tyhjä tai täysi.

Hakeminen tietorakenteesta tapahtuu binäärihakemalla jokaisesta vektorista. Elementtien määrän ollessa n vektoreita on enintään $\log_2 n$, jolloin binäärihakujen aikavaatimukseksi tulee yhteensä enintään $\sum_{i=0}^{\lfloor \log_2 n \rfloor} \log_2 2^i \leq \log_2^2 n$. Keskimäärinkin binäärihakuja tarvitsee tehdä n. puolet tästä, joten hakemisen keskimääräinen aikavaatimus on $O(\log_2^2 n)$.

Elementin lisääminen toteutetaan käytännössä luomalla ensin vektorisetti, jossa on vain lisättävä elementti ja yhdistämällä se alkuperäisen vektorisetin kanssa.

Kaksi vektorisettiä yhdistetään käymällä läpi molempien vektorit pienimmästä lähtien ja yhdistämällä samankokoiset vektorit (jos molemmissa sen kokoinen vektori on täysi). Samalla tyhjennetään yhdistettävän vektorisetin vektorit. Aikavaatimus kahden järjestetyn vektorin yhdistämiseen järjestetty vektori tuottaen on $O(k)$, missä k on uuden vektorin koko. Kokonaisaikavaatimus yhdistämisessä voi siis pahimmillaan olla $O(n)$, kun vektorisetissä on yhteensä n elementtiä. Keskimäärin operaatio on kuitenkin selvästi nopeampi. Yhden elementin lisäämisen keskimääräinen aikavaatimus on $O(\log_2 n)$, mikä todistetaan alempana.

Pseudokoodi yhdistämiselle:

```
carryVector:={};
i:=0;
while i<size of B or carryVector is not empty do
    if none of A[i], B[i] and carryVector is empty then
        carryVector:=merge vectors B[i] and carryVector;
        B[i]:={};
    else
        if two of A[i], B[i] and carryVector are not empty then
            carryVector:=merge the two;
            A[i]:={};
            B[i]:={};
        else
            swap(A[i], max_size {B[i], carryVector});
        end
    end
    i:=i+1;
end
```

Algorithm 1: Merge vectorset B to A

Todistus elementin lisäämisen keskimääräiselle aikavaatimukselle Pseudokoodissa esitetty swap-operaation oletetaan toimivan $O(1)$ -ajassa. Näin ollen algoritmin aikavaatimukseksi tulee $O(\log_2 n + M)$, missä M on suoritettujen yhdistämisoperaatioiden aikavaatimus.

Olkoon jokainen elementti i n elementtiä sisältävässä vektorisetissä vektorissa jonka koko on 2^{k_i}. Tällöin elementti i on ollut mukana yhdistämisoperaatiossa k_i kertaa. (Alussa $k_i = 0$ ja jokaisessa yhdistämisessä se kasvaa yhdellä) Kokonaisaikavaatimus vektorisetin rakentamisen yhdistämisoperaatioihin on tällöin $\sum_{i=1}^{n} k_i$. Koska $k_i \leq \log_2 n$, kokonaisaikavaatimus on enintään $n \log_2 n$. Toisaalta yli puolet elementeistä on ollut yhdistämisessä mukana maksimimäärän $\lfloor \log_2 n \rfloor$, joten kokonaisaikavaatimus on $O(n \log_2 n)$ ja keskimääräinen aikavaatimus elementin lisäämiselle on $O(\log_2 n)$.

4 Longest Common Substring

Longest Common Substring on ongelma, jossa halutaan selvittää joukolle merkkijonoja, mikä on pisin kaikille merkkijonoille yhteinen substring.

Longest Common Substring-ongelman voi ratkaista suffiksipuulla $O(N * K)$ -ajassa, missä N on merkkijonojen yhteispituus ja K merkkijonojen määrä. Toteutin osana projektia tällaisen LCS-algoritmin.

Ideana on muodostaa merkkijonoista *generalized suffix tree*, eli muodostaa yksi merkkijono, jossa merkkijonot ovat peräkkäin ja jokaiselle on lisätty loppuun oma erityismerkkinsä ja luoda sille suffiksipuu. Esimerkiksi merkkijonoista *abbabb*, *ccbbabcc* ja *ababc* voitaisiin muodostaa merkkijono $abbabb\$_0ccbbabcc\$_1ababc\$_2$ ja sitten luoda sille suffiksipuu. ($\$_0$, $\$_1$ ja $\$_2$ ovat uniikkeja erityismerkkejä)

Tässä suffiksipuussa jokainen $\$_i$ on varmasti lehteen vievällä edgellä, koska haarautuminen edellyttäisi sitä, että haarautumiskohtaa vastaava merkkijono esiintyy substringinä vähintään kaksi kertaa alkuperäisessä merkkijonossa. Haarautuminen $\$_i$:n jälkeen ($\$_i$:n alipuussa) tarkoittaisi siis sitä, että olisi ainakin kaksi samanlaista substringiä, joissa merkki $\$_i$ olisi mukana. Tämä on mahdotonta, koska jokainen $\$_i$ on uniikki ja esiintyy merkkijonossa vain kerran. Koska merkkijonon viimeinen merkki on erityismerkki, jokaisella lehteen vievällä edgellä on ainakin yksi erityismerkki.

Jokaista nodea, joka ei ole lehti, vastaa siis substring, jossa ei ole mukana erityismerkkejä. Pisimmän yhteisen substringin on oltava jokin näistä nodeja vastaavista substringeistä. (Edgen keskellä olevaa tilaa vastaavaa substringiä voi aina pidentää seuraavaan nodeen, koska edgen keskellä olevaa tilaa ja seuraavaa nodea vastaavilla substringeillä on samat esiintymät alkuperäisessä merkkijonossa. Lisäksi lehdistä ei voi tulla ratkaisuja.)

Nodea vastaava substring on erityismerkkiin $\$_i$ päättyvän merkkijonon substring jos ja vain jos noden alipuussa on lehti, jossa $\$_i$ on ensimmäinen erityismerkki.

LCS:ää vastaava tila puussa on siis sellaisen noden kohdalla, jonka alipuusssa on jokaiselle $\$_i$:lle lehti, jossa $\$_i$ on ensimmäinen erityismerkki, ja jota vastaava merkkijono on mahdollisimman pitkä.

Tällaisen noden etsiminen on helppo toteuttaa $O(N * K)$ -ajassa. Ratkaisussa rekursoin puun läpi ($O(N)$ nodea) ja katson jokaisen noden kohdalla K-kokoisesta bitsetistä, millä $\$_i$:llä alkavia lehtiä sen noden alipuussa ovat. Tämän jälkeen on vain valittava kaikkille merkkijonoille yhteisistä substringeistä pisin/pisimmät, mikä on helppo toteuttaa esim. käymällä kaikki yhteisiä substringejä vastaavat nodet läpi.

Pseudokoodi rekursiiviselle funktiolle findCommonStrings:

```
bitSet:=bitSet of size K initially filled with zeros;
for each child of node do
    if child is leaf then
        bitSet.setBit(i of first $_i on edge);
    else
        bitSet| =(findCommonStrings(child));
    end
end
if every bit in bitSet is 1 then
    commonStrings.append(node);
end
return bitSet
```

Algorithm 2: findCommonStrings

Rekursiivinen funktio lisää siis listaan commonStrings kaikki nodet, joita vastaava merkkijono on yhteinen kaikille merkkijonoille.

Koko puun käsittelyn aikavaatimus on selvästi $O(N * K)$.