Tietorakenteet ja algoritmit -harjoitustyö toteutusdokumentti

Hannes Ihalainen

February 23, 2018

Contents

1 Johdanto

Projektissa on toteutettu Ukkosen algoritmi ja toteutettu Ukkosen algoritmilla toimiva ratkaisu Longest common substring -ongelmaan. (LCS = TODO)

Lisäksi työssä on toteutettu itse Vector (vector.h automaattisesti kasvava taulukko), "FastSet" (fastset.h nopea lisäys ja haku) ja Bitset -tietorakenteet.

2 Ukkosen algoritmin toteutus

Toteutin Ukkosen algoritmin tallentaen edget eri tavoilla. Mittasin eri toteutusten tehokkuutta eri kokoisilla aakkostoilla.

Tässä taulukko työssä toteutetuista edgejen tallennustavoista .

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

2.1 Implementaatio-rakenteesta...

Ensimmäisenä toteutettu taulukko-implementaatio eroaa muista, koska siinä UkkonenTree on template-luokka. (muissa implementaatioissa on 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 vektoreissa, joiden koot ovat 2:n potensseja. (en tiedä, mikä on tämän tietorakenteen oikea nimi, jos sillä on sellainen)

Hakeminen tapahtuu binäärihakemalla jokaisesta vektorista. Vektoreita on enintään $\log_2 n$, jolloin binäärihakujen aikavaatimukseksi tulee yhteensä enintään $\sum_{i=0}^{\lfloor \log_2 n \rfloor} \log_2 2^i \leq \log_2^2 n$. Keskimäärin binäärihakuja tarvitsee tehdä n. puolet tästä, joten hakemisen keskiarvoinen aikavaatimukseksi tulee $O(\log_2^2 n)$.

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

Kaksi settiä voi yhdistää käymällä läpi molempien vektorit ja yhdistämällä samankokoiset vektorit (jos molemmissa sen kokoinen vektori on täysi). Aikavaatimus vektorien yhdistämiseen on $O(k)$, missä k on uuden vektorin koko. Kokonaisaikavaatimus yhdistämisessä on siis enintään $O(\log_2^2 n + log_2^2 m)$. Kesimimäärin yhdistämisen aikavaatimus on kuitenkin $O(log_2 n + log m)$, kun settien koot ovat n ja m.

Pseudokoodi yhdistämiselle:

```
for do
    read current;
    if understand then
        go to next section;
        current section becomes this one;
    else
        go back to the beginning of current section;
    end
end
```

Algorithm 1: How to write algorithms

Todistus elementin lisäämisen keskimääräiselle aikavaatimukselle

Olkoon jokainen elementti i n-kokoisessa setissä vektorissa jonka koko on 2_i^k. Tällöin elementti i on ollut mukana yhdistämisoperaatiossa k_i kertaa. (Alussa $k_i = 0$ ja jokaisessa yhdistämisessä se kasvaa yhdellä) Kokonaisaikavaatimus setin rakentamiseen on tällöin $\sum_{i=1}^{n} k_i$. Koska $k_i \leq \log_2 n$, kokonaisaikavaatimus on enintään $n \log_2 n$, jolloin keskimääräinen aikavaatimus elementin lisäämiselle on $\log_2 n$

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 tällaisen LCS-algoritmin.

Ratkaisu