SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

PROJEKT - BIOINFORMATIKA

Računanje najduljeg zajedničkog prefiksa temeljeno na BWT

Vjekoslav Giacometti, Tomislav Kiš

Voditelj: *doc. dr. sc. Mirjana Domazet-Lošo*

Zagreb, siječanj, 2016.

Sadržaj

1. Uvod

Što je najdulji zajednički prefiks

Računanje niza najduljih zajedničkih prefiksa (LCP – Longest common prefix array) u računarskoj znanosti jedan je od osnovnih algoritama u različitim područjima primjene, primjerice bioinformatici, rudarenju podataka, pretraživanju teksta ili web stranica (Google tražilica). Svrha niza zajedničkih prefiksa jest proširenje podatkovne strukture niza sufiksa. Služi nam kao opisna struktura koja govori kolika je duljina zajedničkog prefiksa između dva uzastopna sufiksna niza. Pošto su sufiksni nizovi abecedno sortirani niz prefiksa ubrzava pretraživanje po stablu sufiksa tako što govori u koliko karaktera se dva susjedna sufiksna niza preklapaju.

Recimo da je ulazni niz *„banana“*. Njemu se dodaje posebni karakter *'EOF'* koji označava kraj stringa. U našem algoritmu to je dekadska vrijednost *13* koja predstavlja *'\t'* karakter (tab).

S = „bananaEOF“, duljine *n = 7*

Sada definiramo niz sufiksa *A*. Prvo moramo zapisati sve nizove sufiksa te ih poredati abecedno.

i	*S[A[i]:n]*	*A*
1	EOF	7
2	aEOF	6
3	anaEOF	4
4	ananaEOF	2
5	bananaEOF	1
6	naEOF	5
7	nanaEOF	3

Računamo niz prefiksa *H = lcp(S[A[i-1] : n], S[A[i] : n])*

i	*H[i]*
1	⊥
2	0
3	1
4	3

5	0
6	0
7	2

Na primjer, *H[3] = 1* nam govori da sufiksni nizovi *S[6:n] = „aEOF“* i *S[4:n] = „anaEOF“* imaju zajednični prefiks duljine 1 karaktera.

Kada bismo željeli pronaći broj pojavljivanja stringa *P* duljine *m* u drugom stringu *T* duljine *n*, pomoću LCP niza smanjujemo složenost sa *O(m*logN)* na *O(m+logN).*

Ako radimo binarno pretraživanje:

L.....M.....R, uspoređujemo duljinu prefiksa *lcp(L,M)* i *lcp(L,P).* Ako *P* ima manje zajedničkih prefiksnih karaktera s *L* nego što ih ima *L* s *M.* Sljedeći korak algoritma se izvodi nad *P[M:R].* U slučaju *da lcp(L,P) > lcp(L,M)* idemo u lijevo podstablo, a u slučaju *lcp(L,P) = lcp(L,M)* moramo provjeriti *lcp(P,M)* počevši od zadnjeg karaktera u kojem se *L* i *P* poklapaju.

Rezultat ovog proširenog binarnog pretraživanja s LCP nizom jest da se niti jedan karakter iz teksta ne uspoređuje s karakterom iz P više od jedanput.

2. Računanje najduljeg zajedničkog prefiksa temeljeno na BWT

2.1 Korišteni algoritmi i strukture

2.1.1 BWT – Burrows – Wheeler transform

Iz ulaznog stringa duljine *n* treba prvo napraviti BWT niz. On se radi tako da se ulazni string proširi 'EOF' karakterom kao oznakom kraja, zatim se rotira *i* karaktera s početka stringa na kraj (iza karaktera 'EOF'), gdje *i* ide od 0 do *n.* Time se dobiva niz stringova duljine *n+1* koji se zatim abecedno sortira te zadnji karakter svakog stringa predstavlja *i*-ti karakter BWT niza. Time se zauzima $(n+1)^2 * 1$ byte memorijskog prostora. Složenost ovog postupka je $O(n^2 log n)$ radi toga što se dobiveni nizovi moraju sortirati te se u tom postupku moraju usporediti svi stringovi koji su duljine *n.*

Primjer za string *„bananaEOF“*:

Rotirani ulazni string	*BWT*
EOFbanan**a**	a
aEOFbana**n**	n
anaEOFba**n**	n
ananaEOF**b**	b
banana**EOF**	EOF
naEOFban**a**	a
nanaEOFb**a**	a

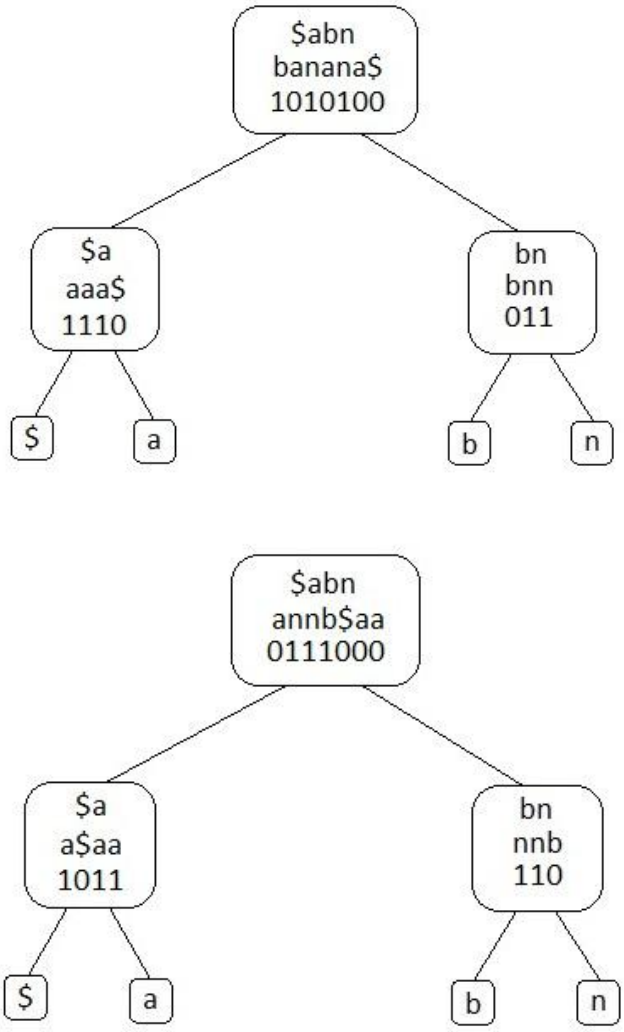
BWT se koristi kako bi se dobili podaci koji se mogu lakše i bolje kompresirati. To je zato što se nakon transfromacije isti karakteri dobivaju u grupama jedan za drugim. Na primjer ako se u tekstu puno puta pojavljuje riječ *„the“*, nakon rotacije i sortiranja svi rotirani nizovi koji počinju s *„he “* na kraju će imati BWT vrijednost *t*. Ovaj postupak je bolji od sličnih postupaka za transformaciju prije sažimanja po tome što se podaci mogu u potpunosti rekonstruirati.

Optimizaciju BW transformacije smo napravili tako što smo u memoriju spremili poduplani ulazni string („bananaEOFbananaEOF“) te samo pamtimo na pokazivač na određeni karakter. Taj pokazivač predstavlja početak rotiranog stringa koji se uspoređuje do duljine jednakoj duljini ulaznog stringa uvećanoj za 1. Time je memorijska složenost smanjena na *(n+1)*2* byte, što je za velike količine podataka iznimno dobra optimizacija.

2.1.2 Wavelet stablo

Generiramo binarno stablo koje u svakom čvoru sadrži dio abecede ulaznog stringa, odnosno cijelu abecedu u korijenu stabla. Na temelju abecede čvora te ulaznog stringa generiramo vektor bitova koji ima bit *0* ako se na tom mjestu u stringu nalazi karakter iz abecede lijevog djeteta odnosno bit *1* u slučaju desnog. Ulazni string za generiranje vektora bitova idućeg čvora jest početni string s izbačenim karakterima koji nisu u abecedi čvora. Abeceda se generira iz početnog stringa te se abecedno sortira, zatim se dijeli na lijevu i desno tako što se uzima sredina, u slučaju neparnog broja karaktera abecede lijevo ide jedan karakter više.

Primjer za ulazni string *„bananaEOF“* i njegovu BWT (*'$'* predstavlja *'EOF'*):



Čvorovi sadrže abecedu, string nad kojim se računa vektor bitova i izračunati vektor. Listovi stabla sadrža samo karakter.

Svrha Wavelet stabla je smanjenje zauzete memorije te brža pretraga prefiksa, odnosno podstringova u ulaznom stringu. Ovaj algoritam smo optimizirali na više načina. Kao prvo vektor bitova spremamo u blokovima *unsigned long long* što je

duljine 64 bita. Blok se inicijalizira te popunjava. Kada je blok popunjen dodajemo novi koji se nadovezuje na prethodni. Druga optimizacija rađena je nad funkcijom $rank_0$ (opisana na str. 6). Ideja je smanjiti složenost prebrojavanja *0* u vektoru bitova tako što za svaki blok od 64 bita spremamo broj *0* od početka do kraja tog bloka. Ovime $rank_0$ jednostavno vrati broj iz prethodnog bloka plus broj *0* u trenutnom bloku do *i-tog* elementa. Složenost je smanjena sa *O(n)* na *O(64) = O(1)* uz malo veće zauzeće memorije.

2.2 Algoritam računanja najduljih zajedničkih prefiksa temeljen na BWT

Za početak potpun primjer za *„bananaEOF“*:

i	*LCP (H[i])*	*BWT*	*S[A[i]:n]*
1	-1	a	EOF
2	0	n	aEOF
3	1	n	anaEOF
4	3	b	ananaEOF
5	0	EOF	bananaEOF
6	0	a	naEOF
7	2	a	nanaEOF
8	-1		

Prvo se za ulazni string izračuna BWT, zatim se izgradi Wavelet stablo te se konačno izvrši sljedeći algoritam.

Algoritam računanja najduljih zajedničkih prefiksa u *O(nlogσ)* vremenu:

```
LCP[i] = -2 za i = 1..n+1
LCP[1] = -1; LCP[n+1] = -1
red_intervala = init()
red_intervala.dodaj([1..n],0)
dok red_intervala nije prazan:
    [i..j], l = red_intervala.uzmiSPocetka()
    lista_intervala = generirajIntervale([i..j])
    za svaki [lb..rb] interval u lista_intervala:
        ako LCP[rb+1] == -2:
            red_intervala.dodaj([lb..rb],l+1)
            LCP[rb+1] = l
```

Ovo je konačni algoritam koji računa niz najduljih zajedničkih prefiksa. Ulazni parametar su mu *n* (duljina ulaznog stringa proširenog s *'EOF'*) i prethodno stvoreno wavelet stablo. *σ* predstavlja duljinu početne abecede. Prilikom računanja koristi se ispod defenirani algoritam koji stvara intervale predznaka.

Algoritam generirajIntervale:

```
generirajIntervale([i..j]):
    lista = []
    generirajIntervale'([i..j], [1..σ], lista)
    vrati lista

generirajIntervale'([i..j], [l..r], lista):
    ako j <= i: vrati 0
    ako l == r:
        c = ∑[l]
        lista.dodaj([C[c]+i .. C[c]+j])
    inače:
        a = rank0(B[l..r], i-1)
        b = rank0(B[l..r], j)
        m = int((l+r)/2)
        generirajIntervale'([a+1..b], [l..m], lista)
        a = i -1 - a
        b = j - b
        generirajIntervale'([a+1..b], [m+1..r], lista)
```

Ovo je općeniti pseudokod navedenih algoritama. Mi u našoj implementaciji koristimo malo drugačiji algoritam po tome što umjesto *[l..r]* člana šaljemo pokazivač na čvor Wavelet stabla koji sadrži *[l..m]* abecedu, *node->left*, odnosno *node->right* za *[m+1..r]* abecedu. Funkcija $rank_0([B,i])$ računa broj pojavljivanja *0* u vektoru bitova *B* do pozicije i (uključujući). *Σ[l]* u slučaju kada je *l == r* predstavlja karakter do kojeg je rekurzija došla, točnije list stabla koji ima abecedu duljine jednog karaktera. Ovdje se pojavljuje još nespomenuta varijabla C. Ona je niz duljine početne abecede koja za svaki karakter abecede na pripadajućem indeksu ima broj karaktera u nizu koji su ispred njega po abecedi.

Primjer računanja *C* za *„bananaEOF“*:

Abeceda	*C*
EOF	0
a	1
b	4
n	6

Uvijek je prvi karakter abecede *'EOF'* te je *C[0] = 0* jer ne postoji niti jedan karakter koji je abecedno prije, a *C[1] = 1* jer jedino *'EOF'* abecedno prethodi, dok se ostali mogu računati kao zbroj vrijednost *C* za sve vrijednosti prije traženog karaktera plus broj pojavljivanja prethodnog karaktera (npr. *C[b] = 0+1+3,* jer se a pojavljuje 3 puta u stringu).

Algoritam se poziva nad cijelom abecedom, dakle s ulaznim parametrom korijenom stabla.

Primjer algoritma za *„bananaEOF“*:

Inicijalizacija:

LCP = [-2, -2, -2, -2, -2, -2, -2, -2]
LCP = [-1, -2, -2, -2, -2, -2, -2, -1]
red_intervala = [([1..7], 0)]

Korak 1:

LCP = [-1, -2, -2, -2, -2, -2, -2, -1]
[i..j], l = [1..7], 0
lista_intervala = [[1..1], [2..4], [5..5], [6..7]]
red_intervala = [([1..1], 1), ([2..4], 1), ([5..5], 1)]
LCP = [-1, 0, -2, -2, 0, 0, -2, -1]

Korak 2:

LCP = [-1, 0, -2, -2, 0, 0, -2, -1]
[i..j], l = [1..1], 1
lista_intervala = [[2..2]]
red_intervala = [([2..4], 1), ([5..5], 1), ([2..2], 2)]
LCP = [-1, 0, 1, -2, 0, 0, -2, -1]

Korak 3:

LCP = [-1, 0, 1, -2, 0, 0, -2, -1]
[i..j], l = [2..4], 1
lista_intervala = [[5..5], [6..7]]
red_intervala = [([5..5], 1), ([2..2], 2)]

LCP = [-1, 0, 1, -2, 0, 0, -2, -1]

Korak 4:

LCP = [-1, 0, 1, -2, 0, 0, -2, -1]
[i..j], l = [5..5], 1
lista_intervala = [[1..1]]
red_intervala = [([2..2], 2)]
LCP = [-1, 0, 1, -2, 0, 0, -2, -1]

Korak 5:

LCP = [-1, 0, 1, -2, 0, 0, -2, -1]
[i..j], l = [2..2], 2
lista_intervala = [[6..6]]
red_intervala = [([6..6], 3)]
LCP = [-1, 0, 1, -2, 0, 0, 2, -1]

Korak 6:

LCP = [-1, 0, 1, -2, 0, 0, -2, -1]
[i..j], l = [6..6], 3
lista_intervala = [[3..3]]
red_intervala = [([3..3], 4)]
LCP = [-1, 0, 1, 3, 0, 0, 2, -1]

Korak 7:

LCP = [-1, 0, 1, -2, 0, 0, -2, -1]
[i..j], l = [3..3], 4
lista_intervala = [[7..7]]
red_intervala = []
LCP = [-1, 0, 1, 3, 0, 0, 2, -1]

Broj poziva funkcije *getIntervals* je 7.

3. Rezulatati testiranja – analiza performansi

Veličina (B)	*BWT (ms)*	*BWT (MB)*	*Wavelet tree construction (ms)*	*Wavelet tree construction (MB)*	*LCP (ms)*	*LCP (MB)*
10000	3	1	0	1	37	1
50000	33	2	7	1	197	2
100000	70	3	17	1	400	3
200000	150	4	37	1	820	5
300000	230	6	53	1	1257	7
400000	313	8	77	2	1700	9
500000	400	10	90	2	2137	13
700000	583	15	130	3	3027	15
1000000	853	21	187	4	4333	22

Rezultati testiranja su očekivani. Vidi se da najviše vremena i memorije odlazi na BWT što doista treba tako biti. BWT zauzme dva puta duljinu ulaznog stringa plus jedna duljina za izlaz te duljina puta 8 bytea za pokazivače na svaki karakter. Za ulazni string duljine *n* zauzeće memoriji je *3*n+8*n = 11n.* Wavelet stablo se konstruira mnogo brže nego BWT te zauzima jako malo memorije jer je korišten vektor bitova za pohranu. Od ukupnog vremena računanja LCP niza veliki dio odlazi na BWT, jako malo na generiranje Wavelet stabla te najveći dio na funkciju generiranja intervala.

4. Zaključak

Računanje najduljeg zajedničkog prefiksa temeljeno na BWT iznimno je dobra optimizacija jer konačna složenost iznosi svega $O(n\log\sigma)$, za abecedu duljine σ i niz duljine n. Uz smanjenje vremena izvođenja još dobivamo i manje zauzeće memorije te iznimno brzu pretragu po stablu koristeći Wavelet strukturu stabla. Svaki čvor u Wavelet stablu je predstavljen vektorom bitova tako da je svaki karakter predstavljen jednim bitom. Ovaj način računanja iznimno je prikladan za korištenje jer se kratkim izračunom dobiva struktura u memoriji male veličine, a ipak prikladna za brzu pretragu. Optimizacija računanja najduljeg zajedničkog prefiksa temeljena na BWT ima bitnu primjenu u području bioinformatike te analize velike količine podataka (rudarenju podataka).

5. Literatura

[1] T. Beller, S. Gog, E. Ohlebusch, T. Schnattiger; Computing the longest common prefix array based on the Burrows – Wheeler transform; 4 August 2012

[2] Burrows–Wheeler transform; Wikipedia; https://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform

[3] LCP array; Wikipedia; https://en.wikipedia.org/wiki/LCP_array

[4] Suffix tree; Wikipedia; https://en.wikipedia.org/wiki/Suffix_tree

[5] Suffic array; Wikipedia; https://en.wikipedia.org/wiki/Suffix_array

[6] Wavelet tree; Wikipedia; https://en.wikipedia.org/wiki/Wavelet_Tree

[7] Succint data structure; Wikipedia; https://en.wikipedia.org/wiki/Succinct_data_structure#Succinct_dictionaries

[8] Compressed suffix array; Wikipedia; https://en.wikipedia.org/wiki/Compressed_suffix_array

[9] FM-Indexes and Backwards Search; http://alexbowe.com/fm-index/