SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Projekt iz Bioinformatike 2016./17. A representation of a compressed de Bruijn graph for pan-genome analysis that enables search

Dino Rakipović
Tena Perak
Katarina Matić

Zagreb, siječanj 2017.

SADRŽAJ

1. Opis projektnog zadatka

De Bruijnov graf niza simbola S duljine n za neki prirodni broj k, usmjereni je graf koji ima čvor za svaki različiti podniz niza S duljine k. Čvorovi u i v su povezani usmjerenim bridom ukoliko se podnizovi u i v pojavljuju uzastopno u nizu S. Susjedni čvorovi u grafu preklapaju se u k-1 simbola te ih može povezivati više od jednog brida, što predstavlja preklapajuća ponavljanja u nizu.

Komprimirani De Bruijnov graf dobiva se spajanjem lanaca čvorova koji se ne granaju u jedan čvor sa duljim podnizom (duljine veće od k). Za sve čvorove u koji imaju samo jednog sljedbenika v, i čvorove v koji imaju samo jednog prethodnika u, moguće je stvoriti jedan zajednički čvor čiji je prethodnik prethodnik od u, a sljedbenik mu je sljedbenik od v.

Takvi grafovi pogodni su za prikaz i pretraživanje pangenoma. Pangenom je skup DNA sekvenci jedinki, najčešće slične ili iste vrste. Prikaz jednog niza koji se sastoji od svih sekvenci pan-genoma u obliku De Bruijnovog grafa daje mogućnost razvoja algoritma za pretraživanje te strukture podataka u svrhu pronalaska ponavljajućeg uzorka među sekvencama.

Projektni zadatak je implementacija konstrukcije implicitnog De Bruijnovog grafa (Algoritam 2) koristeći bit vektore (Algoritam 1). Dodatno je implementirana konstrukcija eksplicitne reprezentacije De Bruijnovog grafa(Algoritam 3).

2. Pomoćne strukture podataka i funkcije

Prije implementacije konstrukcije bit vektora i grafova bilo je potrebno implementirati konstrukciju parametara koje algoritmi primaju, te koristiti prilagođene struktutre podataka za čvorove.

2.1. Format ulazne datoteke i ulazni niz

Ulazne datoteke koje implementacija prima su u FASTA formatu. Taj format sekvence odvaja komentarima koji počinju znakom ">". Pri čitanju datoteke se na kraj svake sekvence stavlja oznaka kraja sekvence ("#"), a nakon što je pročitana zadnja sekvenca u datoteci na kraj niza se stavlja oznaka kraja niza ("$"). Abeceda koju implementacija prihvaća sastoji se od znakova: A, C, G, T, # i $.

2.2. Strukture podataka

2.2.1. Implicitni graf

Implicitna reprezentacija De Bruijnovog grafa opisuje se nizom instanci strukture podataka koja se sastoji od identifikacijskog broja čvora (id), duljine ω niza kojeg čvor predstavlja (len), indeksa lijeve granice ω intervala promatranog niza (lb, prema tome niz se nalazi u ω intervalu [lb ... lb+size-1]), duljine ω intervala (size), indeksa lijeve granice ω intervala (suffix_lb).

```
struct node {

    int len;
    int lb;
    int size;
    int suffix_lb;

    node():
        len(0), lb(0), size(0), suffix_lb(0) {};
};
```

2.2.2. Eksplicitni graf

Eksplicitna reprezentacija De Bruijnovog grafa opisuje se nizom instanci strukture podataka koja opisuje čvor, a sastoji od identifikacijskog broja čvora (id), duljine ω niza koji se veže uz čvor (len), liste pozicija koje označavaju mjesta pojavljivanja ω niza u nizu S (pos_list) i liste sljedbenika promatranog čvora (adj_list).

```
struct enode {
    int len;
    std::vector<int> pos_list;
    std::vector<int> adj_list;

    enode() {};
};
```

2.3. Funkcije

2.3.1. Sufiksno polje (SA)

Sufiksno polje je sortirano polje svih sufiksa stringa (njihovih indeksa). Koristi ga algoritam 1. U algoritmu 1 se sufiksno polje generira koristeći gotovu knjižnicu za izgradnju sufiksnog polja (sais).

2.3.2. Burrows-Wheeler transformacija (BWT)

BWT je reverzibilna transformacija koja mijenja raspored simbola u nizu. Podnizovi koji se ponavljaju dovode do ponavljanja znakova u transformiranom nizu. Transformacija je reverzibilna bez čuvanja dodatnih podataka. Transformirani niz koriste algoritmi 1, 2, i 3.

Za BWT niz vrijedi BWT[i] = s[SA[i]-1].

U ovoj implementaciji je bilo potrebno promijeniti ulazni niz kako bi BWT niz bio ispravno konstruiran. Niz je promijenjen tako što je umjesto znaka "$" u niz stavljen znak "0", te umjesto "#" znak "1" zato što ti znakovi imaju najnižu vrijednost u abecedi.

2.3.3. Wavelet tree

Wavelet tree je struktura podataka koja se koristi za komprimirano spremanje nizova. Stablo rekurzivno dijeli abecedu u parove podskupova gdje listovi odgovaraju znakovima abecede, a u svakom čvoru je spremljen bit vektor koji govori kojem podskupu pripada simbol niza.

Algoritam 2 koristi *wavelet tree* BWT niza.

2.3.4. C mapiranje

Za svaki znak c iz poredane abecede koju prihvaća implementacija vrijedi da je vrijednost C mapiranja za ključ c ukupni broj pojavljivanja znakova koji su strogo manji od znaka c u BWT nizu.

C mapiranje koriste algoritmi 1, 2 i 3.

2.3.5. Longest common prefixes (LCP)

LCP polje sadrži duljine najdužih zajedničkih prefiksa između uzastopnih sufiksa u SA polju.

LCP polje koristi algoritam 1.

2.3.6. Last-to-first-mapping (LF)

Ako je i-ta vrijednost u sufiksnom polju *S_q*, i-ta vrijednost u LF polju sadrži indeks na kojem se može naći sufiks *S_q-1* u sufiksnom polju.

LF vektor se kraće može izračunati uz pomoć C mapiranja i BWT niza, a koristi ga algoritam 3.

3. Algoritam 1: Konstrukcija bit vektora

3.1. Opis algoritma 1

Funkcija kreira dva bit vektora Bl i Br, duljine početnog niza *n*. Ako bit vektor Br na i-toj poziciji ima 1 znači da je $u = S[SA(i), SA(i)+k-1]$ najveći desno ponavljajući niz i $S[SA(i)]$ leksikografski najveći ili najmanji sufiks počenog niza takav da mu je u prefiks. Ako bit vektor Bl na i-toj poziciji ima 1 znači da $u = S[SA(i), SA(i) + k - 1]$ nije najveći desno ponavljajući niz i $S[SA(i)]$ je leksikografski najveći sufiks početnog niza kojem je u prefiks. Funkcija kao parametre prima cijeli broj *k*, koji predstavlja duljinu k-mera, niz BWT, Burrows-Wheeler transformacija i cijeli broj *d*, broj sekvenci početnog niza. Algoritam odmah na početku izračuna sufiksno polje *S* u linearnoj složenosti O(n), ovo sufiksno polje nam je potrebno kako bi izračunali LCP-polje u također linearnoj složenosti O(n). Također računamo i mapu C. Glavni dio algoritma kreće for petljom kroz sve vrijednosti polja LCP. Ako se dva uzastopna sufiksa podudaraju u više od *k* znakova to nam je znak da u idućim koracima možda dobijemo vrijednosti 1 za Br ili Bl te to pamtimo. Ako se vrijednosti podudraju u točno *k* znakova pamtimo indeks $k\_i = i$ jer postoji mogućnost da je $Br(i) = 1$. Ako je podudaranje dva uzastopna sufiksa manje od *k* provjeravamo možemo li dodati jedinice u bit vektore. Ako nam je prije zapamćeni indeks k_i veći od lijeve granice tj. indeksa najmanjeg sufiksa koji se podudara sa svojim sljedbenicima u najmanje *k* znakova, postavljamo vrijednosti Br na 1 na indeksima lijeve i desne granice. U ovom koraku također počinjemo izgradnju implicitnog grafa kojem postavljamo početne čvorove. Dalje pokušavamo postaviti neke vrijednosti Bl na 1, to radimo ako nam je najveći indeks u kojem se razlikuju dva uzastopna BWT znaka veći od lijeve granice. Tada su sve vrijednosti $Bl(C[bwt[j]])$ gdje je *j* od lijeve granice do indeksa trenutnog sufiksa jednake 1. Pošto smo gore pojasnili što predstavlja 1 u Bl, na kraju algoritma brišemo sve 1 iz njega koje su na indeksima najvećih desno ponavljajućih nizova.

3.2. Primjer izvođenja

>1
AGCTTTTAAC
>2
GGGTGGATT
>3
ATCGTGC

Za ulazni primjer stvara se niz S=AGCTTTTAAC#GGGTGGATT#ATCGTGC$. U tablici 1. navedene su vrijednosti svih pomoćnih struktura prije početka samog stvaranja bit vektora.

i	SA	LCP	BWT
1	28	-1	C
2	20	0	T
3	10	0	C
4	7	0	T
5	8	1	A
6	0	1	$
7	21	1	#
8	17	2	G
9	27	0	G
10	9	1	A
11	23	1	T
12	2	1	G
13	16	0	G
14	26	1	T
15	1	2	A
16	15	1	T
17	11	2	#
18	12	2	G
19	24	1	C
20	13	3	G
21	19	0	T
22	6	1	T
23	22	1	A
24	25	1	G
25	14	2	G
26	18	1	A
27	5	2	T
28	4	2	T
29	3	3	C
30		-1	

Tablica 3.1: Stanje pomoćnih struktura

Vrijednosti mape C su sljedeće: C[A] = 3, C[C] = 8, C[G] = 12, C[T] = 20, C[$] = 0, C[#] = 1 Uz činjenicu da je prolaz algoritma rađen za k=3 iz Tablice 3.1. odmah uočavamo da postoje samo dva uzastopna sufiksa koja imaju lcp jednak k. Iz ovoga zaključujemo da ćemo dva puta pokušati osvježiti bit vektore jedinicama. Ostale korake algoritma možemo jednostavno preskočiti jer se ne događa apsolutno ništa, osim što u svakoj iteraciji postavljamo lijevu granicu na trenutni indeks te last_diff ukoliko su bwt[i] i bwt[i-1] različiti te uvećavamo polje C za jedan na znakovima bwt[i-1]. I. Prvi indeks za koji je lcp[i] == k je indeks 20 te k_index = 20 te je postavljena zastavica open. U idućoj iteraciji lcp[i] je manji od k te pošto smo postavili zastavicu open provjeravamo je li nam k_index veći od lijeve granice(lb=19). Uvjet je uspio te postavljamo Br[19] i Br[20] na 1. Zadnji indeks za kojega su bwt[i] i bwt[i-1] različiti je upravo 20 te je last_diff = 20. Uvjet last_diff > lb prolazi te postavljamo dva puta(lb=19, i=21) vrijednost Bl[C[c]] na 1. Za j=19 c = bwt[19] = 'C' te je C['C'] = 11 jer smo prije njega još tri puta naišli na slovo C. Za j=20 c = bwt[20] = 'G' te je C['G'] = 18 iz istog razloga. Stanje bit vektora nakon prvog prolaska možemo vidjeti u prva dva stupca tablice 3.2.

II. Idući indeks je indeks 29, ponavljamo sve provjere, prisjetimo se da je lb=28 te prvi uvjet prolazi, postavljamo Br[28] i Br[29] na 1. Drugi uvjet također prolazi jer je last_diff=29. Istim postupkom kao gore postavljamo Bl[29] i Bl[12] na 1. Stanje bit vektora nakon drugog prolaska možemo vidjeti u trećem i četvrtom stupcu tablice 3.2.

III. Na kraju nam još ostaje provjeriti postoji li lijevi bit vektor Bl koji je jednak 1 tamo gdje je i desni. Odmah vidimo da je to na indeksu 29. Postavljamo Bl[29] = 0. Stanje bit vektora nakon trećeg prolaska možemo vidjeti u zadnja dva stupca tablice 3.2.

i	Br1	Bl1	Br2	Bl2	Br3	B3
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
11	0	1	0	1	0	1
12	0	0	0	1	0	1
13	0	0	0	0	0	0
14	0	0	0	0	0	0
15	0	0	0	0	0	0
16	0	0	0	0	0	0
17	0	0	0	0	0	0
18	0	1	0	1	0	1
19	1	0	1	0	1	0
20	1	0	1	0	1	0
21	0	0	0	0	0	0
22	0	0	0	0	0	0
23	0	0	0	0	0	0
24	0	0	0	0	0	0
25	0	0	0	0	0	0
26	0	0	0	0	0	0
27	0	0	0	0	0	0
28	0	0	1	0	1	0
29	0	0	1	1	1	0

Tablica 3.2: Tijek izvođenja

4. Algoritam 2: Konstrukcija implicitnog grafa

4.1. Opis algoritma 2

Algoritam gradi implicitnu reprezentaciju de Bruijnovog grafa koristeći bit vektore generirane prethodno opisanim algoritmom. Algoritam 1 dodatno inicijalizira čvorove implicitnog grafa. Na početku algoritma potrebno je utvrditi konačni broj čvorova te dodati završne čvorove (engl. *stop nodes*). Svaki čvor u grafu sadrži 4 *integera*: duljinu podniza ω, indeks početka ω-intervala, indeks početka intervala sufiksa od ω duljine k te veličinu oba intervala.

Algoritam računa intervale sufiksnog polja za sve desno-maksimalne podnizove duljine k. Zatim za svaki od tih podnizova ω računa $c\omega$-intervale sufiksnog polja, gdje je c element abecede. Zatim za svaki niz $u = c\omega$ računa cu-intervale itd. Dakle, procedura započinje sa svim desno-maksimalnim podnizovima duljine k i proširuje ih dok god može (na sve moguće načine) znak po znak. [2] Algoritam 1 pronalazi sve desno-maksimalne podnize duljine k, te dodaje čvorove u graf G kao njihove reprezentacije. Svaki dodani čvor ima svoj identifikator koji se dodaje u red Q. Algoritam izgradnje grafa uzima redom te čvorove prema indekima zapamćenim u redu Q te ih proširuje u lijevo. Za pronalazak $c\omega$ intervala korištena je funkcija *intervals_symbols* iz biblioteke sdsl-lite (algoritam opisan u [1]). Za svaki niz $c\omega$ dalje testiramo je li njegov prefiks duljine k desno-maksimalni ponavljajući niz. Provjera je jednostavna korištenjem desnog bitvektora u kojem su označene granice sufiksnih intervala svih desno-maksimalnih ponavljajućih nizova. Ukolko se testom zaključi da prefiks jest desno-maksimalan, računanje staje i algoritam prelazi na idući čvor u redu Q. U suprotnom, ako funkcija *intervals_symbols* pronađe više od jednog znaka c, znamo da je niz ω lijevo-maksimalni podniz (prema dokazu u [2]). U tom slučaju u graf se dodaje novi čvor koji reprezentira prefiks duljine k niza $c\omega$ s kojim se dalje nastavlja izvođenje algoritma. Ako niz ω nije lijevo-maksimalni niz, njegov pripadni čvor se proširuje

(povećavamo vrijednost duljine niza) te se nastavlja istraživati pronađeni $c\omega$ interval.

4.2. Primjer izvođenja

Tablica 4.1 prikazuje početni implicitni graf generiran Algoritmom 1.

id	len	lb	size	suff_lb
0	3	18	2	18
1	3	27	2	27
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	1	0	1	0
6	1	1	1	1
7	1	2	1	2

Tablica 4.1: Primljeni implicitni graf

Algoritam započinje čvorom čiji je identifikator $id = 0$, a odgovara podnizu "GTG". Za ω-interval $[18, 19]$ pronađena su dva intervala, za znak "C" interval [10, 10] te za znak "G" interval [17, 17]. U graf dodajemo dva nova čvora te je novi izgled grafa prikazan u tablici 4.2.

id	len	lb	size	suff_lb
0	3	18	2	18
1	3	27	2	27
2	3	10	1	10
3	0	0	0	0
4	3	17	1	17
5	1	0	1	0
6	1	1	1	1
7	1	2	1	2

Tablica 4.2: Implicitni graf nakon 1. iteracije

Nastavljamo proširivati idući čvor ($id = 1$) te za interval $[27, 28]$ (podniz "TTT") ponovno pronalazimo 2 znaka: "C" i "T". Za znak "T" utvrđujemo da je prefiks duljine

3 desno-maksimalni ponavljajući niz, pa on ne utječe na izgled grafa. Za znak "C" dodajemo novi čvor $(3, 11, 1, 11)$ na poziciju $id = 3$.

Algoritam nastavlja s pretragom čvorova po redoslijedu dodavanja u red Q, pa se idući proširuje čvor $(1, 0, 1, 0)$. Niz dotičnog čvora ("$") nije lijevo-maksimalni, stoga mu povećavamo veličinu za 1 (len = 2). Nastavljamo s jedinim pronađenim intervalom $[8, 8]$ za znak "C". Sada tražimo lijevo proširenje niza "C$". Ponovno pronalazimo samo jedan znak/interval "G"/[13, 13] , pa se veličina trenutnog čvora ponovno povećava za 1 (len = 3), a algoritam nastavlja istraživati pronađeni interval, odnosno niz "GC$". Ponovno, očekivano, nalazimo jedan interval [23, 23] (za znak "T") i inkrementiramo veličinu trenutnog čvora (len = 4). Algoritam dalje traži $c\omega$ intervale niza ω = "TGC$" : pronalazimo interval [18, 18] za znak "G". Pomoću bitvektora sada možemo ustvrditi da smo došli do granice, te pamtimo nove vrijednosti čvora u grafu. Novo stanje grafa prikazano je u tablici 4.3.

id	len	lb	size	suff_lb
0	3	18	2	18
1	3	27	2	27
2	3	10	1	10
3	3	11	1	11
4	3	17	1	17
5	4	23	1	0
6	1	1	1	1
7	1	2	1	2

Tablica 4.3: Implicitni graf nakon 3. iteracije

Obrađujemo sljedeći čvor $(1, 1, 1, 1)$. Niz koji odgovara ovom čvoru je "#". Kako se svaki znak "#" međusobno razlikuje, postoji točno jedan znak koji algoritam sada može pronaći. Metoda *interval_symbols* vraća znak "T" i njemu pripadni sufiksni interval $[20, 20]$. Kako niz "T#" nije lijevo-maksimalni povećavano veličinu čvora (len = 2). Dalje za niz "T#" pronalazimo ponovno jedan znak "T" i njegov sufiksni interval [25, 25] te povećavamo čvor (len = 3). Dalje niz "TT#" proširujemo u lijevo sve dok ne dobijemo da je njegov prefiks duljine k = 3 desno-maksimalni. To vrijedi za niz "GTGGATT#" stoga proširivanje staje jedan korak prije. Vrijednost čvora s $id = 6$ je $(7, 24, 1, 1)$.

Algoitam se dalje nastavlja opisanim postupkom, te se za rezultat dobije konačni graf prikazan u tablici 4.4.

id	len	lb	size	suff_lb
0	3	18	2	18
1	3	27	2	27
2	5	6	1	10
3	5	5	1	11
4	4	16	1	17
5	4	23	1	0
6	7	24	1	1
7	6	26	1	2

Tablica 4.4: Konačni implicitni graf

5. Algoritam 3: Konstrukcija eksplicitnog grafa

5.1. Opis algoritma 3

Algoritam iz implicitne reprezentacije stvara eksplicitnu reprezentaciju grafa. Algoritmu se kao parametri predaju implicitni graf, LF vektor (*last to first mapping*), niz znakova BWT (Burrows-Wheeler transformacija), par bit vektora, broj sekvenci d, veličina ulaznog niza n i veličina k-mera k.

Broj čvorova oba grafa je jednak pa se na početku stvara onoliko praznih čvorova eksplicitnog grafa koliko je čvorova u primljenom implicitnom grafu. Nakon toga se računaju lijevi i desni rank vektori. Vrijednost na određenom indeksu rank vektora označava broj jedinica koji se dotada pojavio u odgovarajućem binarnom vektoru konstruiranim algoritmom 1.

Početna pozicija u ulaznom nizu spremljena je u varijablu pos, a iznos joj odgovara kraju niza. Algoritam iterira po sekvencama. Svaki niz od d sekvenci ($d - 1$ znak "#" i jedan znak "$") ima d zaustavnih čvorova koje koristi algoritam za stvaranje implicitnog grafa. Eksplicitni graf treba d početnih čvorova. Budući da se gradi koristeći implicitni graf, algoritam počinje na zadnjem čvoru implicitnog grafa koji se nalazi na indeksu $id = rightMax + leftMax + s$, gdje su vrijednosti *rightMax* i *leftMax* izračunate uz pomoć rank vektora, a s redni broj trenutne sekvence.

Pozicija se zatim pomiče za duljinu ω niza koji odgovara tom čvoru. Indeks idx dobiva vrijednost lijeve granice ω intervala promatranog niza. Postavlja se duljina čvora na id poziciji i u njegovu pos_list dodaje se trenutna pozicija.

Algoritam nakon toga u *while* petlji provjerava nalazi li se na indeksu lijeve granice ω intervala promatranog niza znak kraja cijelog niza ili znak kraja sekvence, ako da nastavlja dalje u *for* petlji, ako ne ulazi u tijelo *while* petlje. Indeks i dobiva vrijednost indeksa na kojem se nalazi niži sufiks u sufiksnom polju s obzirom na indeks idx ($LF(idx)$). Iznos indeksa novog čvora ovisi o broju jedinica u desnom bit vektoru do

indeksa i, i o vrijednosti bit vektora na tom mjestu.

Nakon što se nađe nova vrijednost indeksa id, prelazi se na novu poziciju umanjivanjem stare za duljinu len i dodavanjem duljine k-mera k. Novom čvoru se postavljaju duljina, dodaje se pozicija u listu pozicija i stari id u listu sljedbenika novog čvora. Za trenutni id se postavlja novi id. Kada dode do izlaska iz *while* petlje zbog uvjeta da lijeva granica promatranog niza ne smije biti znak završetka sekvence ni niza, to znači da se prelazi u novu sekvencu ili se došlo do početnog čvora grafa pa se u listu početnih čvorova dodaje trenutni id. Indeks i se mijenja na isti način kao i pri ulasku u *while* petlju.
Kada algoritam završi u listi čvorova nalazi se eksplicitni graf koji odgovara dobivenom implicitnom grafu.

Sljedeći primjer dobro opisuje rad algoritma za više sekvenci.

5.2. Primjer izvođenja

Primjer ulazne datoteke s tri sekvence.

>1
AGCTTTTAAC
>2
GGGTGGATT
>3
ATCGTGC

Tablica 4.4 prikazuje primljeni implicitni graf za gornji primjer ulazne datoteke u FASTA formatu. Komentari koji počinju sa ">" se zanemaruju i označavaju početak nove sekvence. Tablica 5.1 prikazuje stanje struktura potrebnih za izgradnju grafa .

i	BWT	B_l	B_r	r_l	r_r	LF
0	/	/	/	0	0	8
1	C	0	0	0	0	20
2	T	0	0	0	0	9
3	C	0	0	0	0	21
4	T	0	0	0	0	3
5	A	0	0	0	0	0
6	$	0	0	0	0	1
7	#	0	0	0	0	12
8	G	0	0	0	0	13
9	6	0	0	0	0	4
10	A	0	0	0	0	22
11	T	1	0	1	0	14
12	G	1	0	2	0	15
13	G	0	0	2	0	23
14	T	0	0	2	0	5
15	A	0	0	2	0	24
16	T	0	0	2	0	2
17	#	0	0	2	0	6
18	G	1	0	3	0	10
19	C	0	1	3	1	17
20	G	0	1	3	2	25
21	T	0	0	3	2	26
22	T	0	0	3	2	6
23	A	0	0	3	2	18
24	G	0	0	3	2	19
25	G	0	0	3	2	7
26	A	0	0	3	2	27
27	T	0	0	3	2	28
28	T	0	1	3	3	11
29	C	0	1	3	4	-

Tablica 5.1: parametri koje prima algoritam za konstrukciju eksplicitnog grafa iz implicitnog (u kodu su bwt, bl, br indeksirani od 1, ostali od 0)

n=29; d = 3; k = 3;
i = 0; s = 0; pos = 30;
//1. sekvenca, s = 0
for (s = 0; s<d; s++)
id = 5
pos = 26
idx = 23
while (BWT[idx+1]!={$, #})
//1. ulazak u while petlju za
//BWT[idx+1] = "G"
ones = 1
newId = 0
pos = 25
idx = 18
id = 0
i = 18
while (BWT[idx+1]!={$, #})
//2. ulazak u while petlju
//BWT[idx+1]="C"
ones = 0
newId = 2
pos = 22
idx = 6
id = 2
i = 10
	//2. sekvenca, s = 1
	for (s = 0; s<d; s++)
	id = 6
	pos = 15
	idx = 24
	while (BWT[idx+1]!={$, #})
	//1. ulazak u while petlju za
	//BWT[idx+1] = "G"
	ones = 2
	newId = 0
	pos = 14
	idx = 19
	id = 0
	i = 19
	while (BWT[idx+1]!={$, #})
	//2. ulazak u while petlju
	//BWT[idx+1]="G"
	ones = 0
	newId = 4
	pos = 12
	idx = 16
	id = 4
	i = 17
		//3. sekvenca, s = 2
		for (s = 0; s<d; s++)
		id = 7
		pos = 6
		idx = 26
		while(BWT[idx+1]!={$, #})
		//1. ulazak u while petlju
		//BWT[idx+1] = "T"
		ones = 3
		newId = 1
		pos = 5
		idx = 27
		id = 1
		i = 27
		while(BWT[idx+1]!={$, #})
		//2. ulazak u while petlju
		//BWT[idx+1]="T"
		ones = 4
		newId = 1
		pos = 4
		idx = 28
		id = 1
		i = 28
		while(BWT[idx+1]!={$, #})
		//3. ulazak u while petlju
		//BWT[idx+1]="C"
		ones = 0
		newId = 3
		pos = 1
		idx = 5
		id = 3
		i = 11
		//BWT[idx+1]="$", s = 2
		//algoritam završava

Tablica 5.2: Tijek izvođenja

U tablici 5.2 prikazan je tijek izvođenja algoritma na danom primjeru. U algoritam se ulazi na poziciji $pos = 30$ i na čvoru s $id = 5$. Ulazi se u *for* petlju, mijenjaju se indeksi te se nakon toga ulazi u *while* petlju. U *while* petlji se testira uvjet ulaska, a taj uvjet traži da znak u BWT listi na mjestu $idx + 1$ (+1 zbog 1 indeksiranog BWT) ne bude "$" ni "#". Nakon toga se gleda broj jedinica u desnom rank vektoru na mjestu $i + 1$, te se na temelju toga odlučuje o mijenjanju id-a. Ako je broj jedinica paran i na i-tom mjestu u desnom bit vektoru se nalazi 0, novi id se računa ovako: $noviId = rightMax + lijeviRankVektor[i]$, u protivnom se računa ovako: $noviId = (brojJedinica - 1)/2$. U *while* petlju ulazi dva puta testirajući znakove "G" i "C" dok ne naiđe na "#" znak na uz idx 6 ($BWT(7)$). Unutar petlji dodaje poziciju u listu pozicija te dodaje indekse čvorova sljedbenika novome čvoru. Izlazi iz *while* petlje, dodaje novi početni čvor, povećava s i i brojače *for* petlje i drugi put ulazi u *for* petlju. Unutar *for* petlje dva puta uđe u *while* petlju testirajući znakove "G" i "G" dok uz idx 16 ne naiđe na "#" znak. Zbog toga izlazi iz *while* petlje i kreće na novi podniz. Dodaje novi početni čvor, povećava brojače *for* petlje i ulazi u *for* petlju. Tri puta ulazi u *while* petlju testirajući znakove "T", "T" i "C" dok s idx 5 ne dođe do znaka "$" i izađe iz petlje. Dodaje novi početni čvor (koji je ujedno i prvi početni čvor) i ne ulazi ponovno u *for* petlju jer ne zadovoljava uvjet $s < d$, jer je s sada povećan na 3, što je jednako broju sekvenci. Algoritam završava dodavanjem prvog početnog čvora eksplicitnog grafa. Dobiveni eksplicitni graf je određen svojim čvorovima koji su određeni svojim listama pozicija, duljinom podniza, i listom sljedbenika te odgovara implicitnom grafu koji je proslijeđen algoritmu.

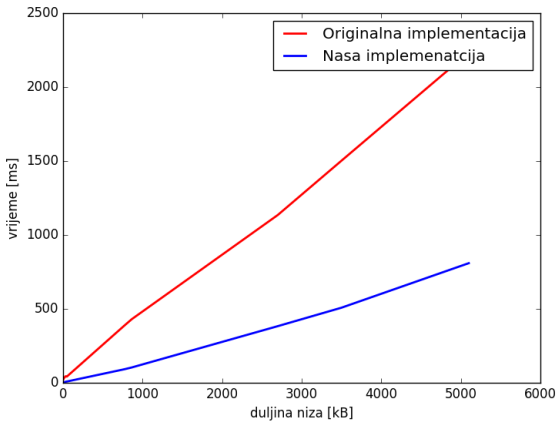
```
node0(len = 3          node1(len = 3          node2(len = 5          node3(len = 5
      pos = [25, 14]         pos = [5, 4]           pos = [22]             pos = [1]
      adj = [5, 6])          adj = [7, 1])          adj = [0])             adj = [1])

node4(len = 4          node5(len = 4          node6(len = 7          node4(len = 6
      pos = [12]             pos = [26]             pos = [15]             pos = [6]
      adj = [0])             adj = [])              adj = [])              adj = [])
```

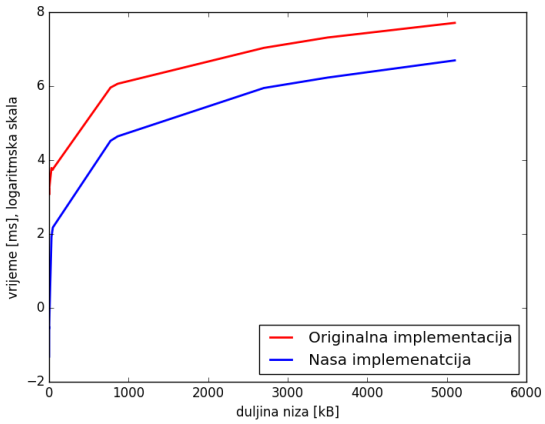
Slika 5.1: Izlaz algoritma za zadani primjer

6. Rezultati

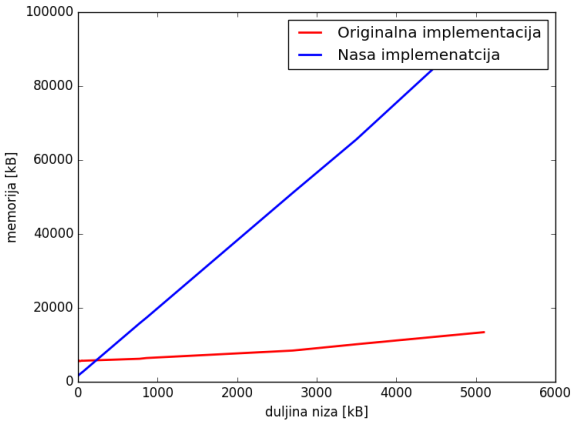
6.1. Algoritam 1



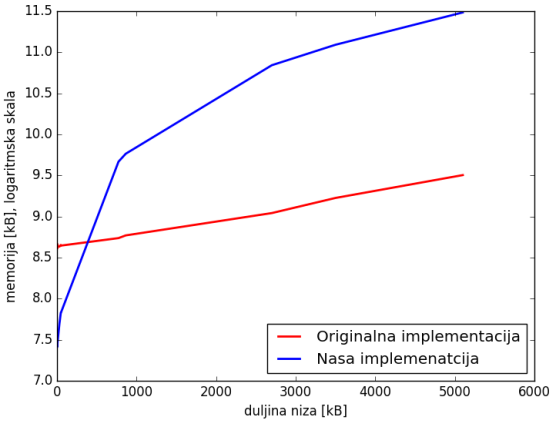
Slika 6.1: Vremenske performanse



Slika 6.2: Vremenske performanse u logaritamskoj skali

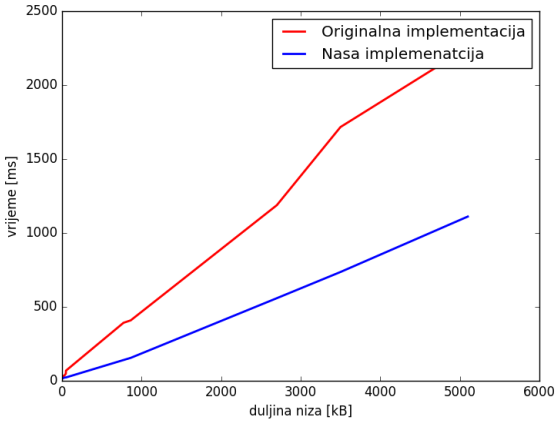


Slika 6.3: Zauzeće memorije

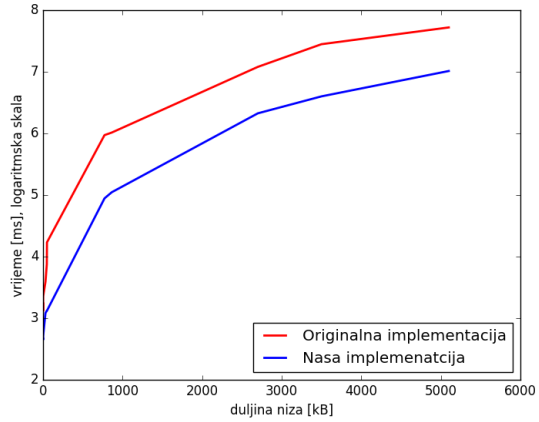


Slika 6.4: Zauzeće memorije u logaritamskoj skali

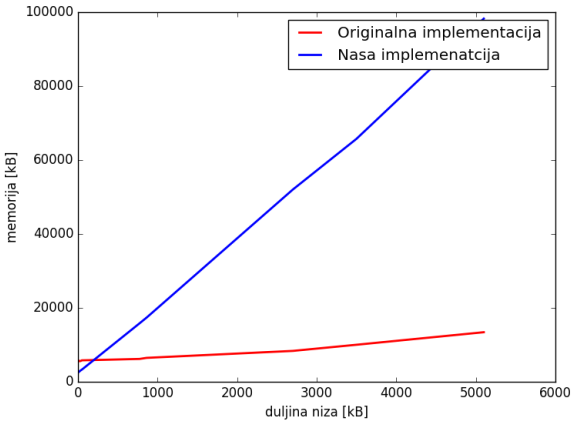
6.2. Algoritam 2



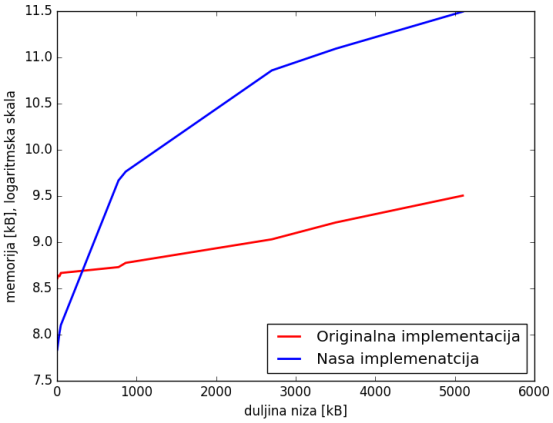
Slika 6.5: Vremenske performanse



Slika 6.6: Vremenske performanse u logaritamskoj skali



Slika 6.7: Zauzeće memorije



Slika 6.8: Zauzeće memorije u logaritamskoj skali

6.3. Algoritam 3

S obzirom da je algoritam 3 dodatno implementiran, na njemu je testirana samo točnost.

7. Rasprava

Mjerenja prikazana u prethodnom poglavlju izvršena su na računalu sljedećih specifikacija: RAM: 8 GB, Procesor: Intel Core i7-6700HQ 2.6Ghz, koristeći programsku knjižnicu Chrono za mjerenje vremena i Unix-ov alat time koji ispisuje i zauzeće memorije.

Mjerenja na oba algoritma pokazuju da je naša implementacija bolja što se tiče vremenskih performansi, dok je originalna implementacija bolja što se tiče zauzeća memorije.

Prilikom mjerenja primijetili smo nagli porast u zauzeću memorije nakon stvaranja sufiksnog polja uz korištenje programske knjižnjice sais. Zauzeće memorije naše implementacije prije stvaranja sufiksnog polja bilo je manje ili jednako zauzeću originalne implementacije do tog koraka u oba algoritma. Međutim, već nakon stvaranja polja, zauzeće memorije naše implementacije četiri puta premašuje maksimalno zauzeće originalne. Primijetili smo da originalna implementacija često koristi standardne ulaze i izlaze i tako smanjuje zauzeće memorije, spremanjem struktura koje trenutačno ne koristi oslobađa prostor u radnoj memoriji.

Vremenske performanse naše implementacije bolje su od performansi originalne implementacije, međutim, moguće je da originalna gubi na vremenu upravo zbog korištenja standardnih izlaza i ulaza u svrhu smanjenja zauzeća memorije.

8. Zaključak

Naša implementacija reprezentacije komprimiranog De Bruijnovog grafa prošla je sve testove točnosti, kako na sintetskim primjerima veličine od 2000 do 100000 znakova, tako i na primjeru Escherichia coli veličine 5140000 znakova. Prema tome, možemo zaključiti da naša implementacija radi ispravno.

Poboljšanja implementacije su moguća, pogotovo poboljšanja u obliku smanjenja zauzeća memorije, koja bi se mogla izvesti na sličan način kao i u originalu, spremanjem dijelova rješenja u priručnu memoriju.

9. Literatura

[1] Timo Beller and Enno Ohlebusch. Efficient construction of a compressed de bruijn graph for pan-genome analysis. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, pages 40–51, 2015.

[2] Timo Beller and Enno Ohlebusch. A representation of a compressed de bruijn graph for pan-genome analysis that enables search. *Algorithms for Molecular Biology*, 11(1):20, 2016.