SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Projekt iz kolegija Bioinformatika 2017./2018.

A representation of a compressed de Bruijn graph for pan-genome analysis that enables search

Bruna Anđelić, 0036478714

Matija Čavrag, 0036478415

Luka Kovačić, 0036479914

Zagreb, siječanj 2018.

Sadržaj

1. Uvod

De Bruijnov graf u originalnoj je definiciji predložio nizozemski matematičar Nicolaas de Bruijn 1946. godine s ciljem pronalaska najkraćeg cirkularnog nadniza koji sadrži sve moguće podnizove duljine k(k-torke) pripadajuće abecede. U ovom je grafu svaka (k-1)-torka određene abecede predstavljena kao vrh, a 2 su vrha povezana bridom ako jedan vrh predstavlja k-1 prefiks, a drugi k-1 sufiks te k-torke. Sam brid između 2 vrha predstavlja k-torku. Možemo tako reći da De Bruijnov graf niza simbola S duljine n za neki prirodni broj k predstavlja usmjereni graf u kojem postoji čvor za svaki različiti podniz niza S duljine k, a bridovi grafa predstavljaju k-torke. Prilikom izrade ovog projektnog zadatka nije se ostvarivao klasični već komprimirani De Bruijnov graf koji se dobiva spajanjem lanaca čvorova koji se ne granaju u jedan čvor s duljim podnizom, odnosno podnizom duljim od konstantne duljine k koja je korištena u klasičnoj implementaciji. Za svaki čvor u komprimiranog De Brujinovog grafa koji ima samo jednog sljedbenika v i čvor v koji ima samo jednog prethodnika u stvara se novi zajednički čvor čiji su prethodnici jednaki prethodnicima čvora u, a sljedbenici jednaki sljedbenicima čvora v. Općenito takvi grafovi pogodni su za prikazivanje i pretraživanje pangenoma, odnosno skupa DNA sekvenci jedinki najčešće iste ili slične vrste. De Bruijnov graf omogućuje prikaz svih sekvenci jednog niza pan-genoma koji daje mogućnost razvoja algoritama za pretraživanje, ali i strukturi podataka korištenih u svrhu pronalaska ponavljajućeg uzorka među sekvencama.

Cilj ovog projektnog zadatka je implementacija algoritma za konstrukciju bit vektora (Algoritam 1) te koristeći taj algoritam implementacija algoritma za konstrukciju implicitnog De Bruijnovog grafa(Algoritam 2). Implementacija algoritama primarno se bazira na pseudokodu originalnog rješenja [1]. U nastavku rada opisat će se i objasniti osnovne strukture i funkcije korištene u implementaciji algoritama, zatim će se pobliže objasniti algoritam jedan zajedno sa primjerom izvođenja, a jednako će se napraviti i za algoritam 2. Nadalje provesti će se analiza rezultata dobivenih usporedbom rješenja dobivenog kroz ovaj projekt s rješenjima originalne implementacije te će se na kraju izvesti zaključak o uspješnosti projekta.

2. Strukture podataka i funkcije

U ovom će se poglavlju dati kratki pregled struktura i funkcija korištenih u ovom programskom rješenju. Za početak bitno je napomenuti formate ulaznih podataka. Ulazne datoteke koje su korištene u implementaciji primaju se u FASTA formatu, odnosno sekvence ulazne datoteke odvojene su komentarima koji počinju oznakom „>“. U ovoj programskoj implementaciji pri čitanju datoteke na kraju svake sekvence stavlja se oznaka za kraj sekvence(u ovoj implementaciji znak „%“), a nakon što se pročita zadnja sekvenca stavlja se oznaka za kraj niza(u ovoj implementaciji znak „\0“). Abeceda korištena u ovoj implementaciji su redom od leksikografski najmanjeg do najvećeg : \0, %, A, C, G, T.

2.1 Struktura podataka

U sklopu ovog projekta implementirana su 2 algoritma, jedan za konstrukciju bit vektora i drugi za konstrukciju implicitnog grafa. Budući da 2. algoritam koristi 1., možemo reći da je konačni rezultat izvođenja oba algoritma implicitni graf. Implicitni graf pisan je vektorom instanci tipa *Node*, koji je struktura definirana u programskom rješenju. Struktura *Node* sastoji se od duljine intervale niza kojeg čvor predstavlja(*len*), indeksa lijeve granice intervale promatranog niza (*lb*, niz se nalazi u intervalu [lb .. lb+size-1]), duljine intervala (*size*) te indeksa lijeve granice intervala (*suffix_lb*).

Prikaz strukture slijedi u nastavku:

```
struct Node {
	uint64_t len;
	uint64_t lb;
	uint64_t size;
	uint64_t suffix_lb;
};
```

2.2 Funkcije

Osnovne funkcije korištene u ovom programskoj implementaciji opisani su u nastavku, a pretežno su ostvarene korištenjem gotovih knjižnica [2] da bi se ostvarila što bolje optimizacija. Sve potrebne dorade nad rješenjima gotovih knjižnica [2] također su opisane u nastavku.

2.2.1 SA (Sufiksno polje)

Generiranje sufiksnog polja u ovoj se programskoj implementaciji ostvaruje korištenjem gotove knjižnice [2] za izradu sufiksnog polja. Sufiksno polje predstavlja sortirano polje indeksa svih sufiksa ulaza. Sufiksi su poredani od leksikografski najmanjeg do leksikografski najvećeg (tako je primjerice uvijek prvi član SA indeksa znaka '\0').

2.2.2. LCP (Longest common prefixes)

Za generiranje LCP polja također se koristi gotova knjižnica [2] za izradu LCP polja, ali gotova implementacija nema jednak oblik kao onaj opisan u pseudokodu osnovne implementacije koji se koristi u ovom projektu. Iz tog se razloga radi dorada rezultata te se na kraj LCP polja dodaje -1. Budući da se rezultat gotove implementacije ne može direktno mijenjati rezultat dodavanja sprema se u novi vektor nazvan *lcpfull*. LCP polje općenito sadrži duljine najdužih zajedničkih prefiksa međusobno uzastopnih sufiksa iz SA polja.

2.2.3. BWT (Burrows-Wheeler transformacija)

Za generiranje BWT također je korištena gotova knjižnica [2], no prvo se ulazni niz trebao promijeniti tako da su, kao što je ranije napomenuto, sekvence odvojene znakom „%“, a kraj naznačen znakom „\0“. Unutar pseudokoda naznačeno je da se indeksi BWT-a kreću od 1, ali gotova knjižnica [2] koristi indekse od 0 pa se umjesto pomicanja cijelog polja udesno korigiralo korištenje indeksa danog pseudokoda. BWT je općenito reverzibilna transformacija koja

mijenja raspored simbola u nizu te ne čuva dodatne podatke. Koristi se u oba algoritma, a općenito vrijedi da je BWT[i] = s[SA[i]-1].

2.2.4. C-mapiranje

Za ostvarivanje C mapiranja nije korištena gotova implementacija već je C polje izgrađeno unutar programske implementacija. Općenito u C polju za svaki znak abecede postoji informacija o tome koliko se ukupno puta u nizu pojavljuju znakovi koji su leksikografski strogo manji od znaka čiju vrijednost gledamo.

2.2.5. WT (Wavelet tree)

Za konstrukciju WT unutar ovog programskog rješenja korištena je gotova knjižnica [2]. *Wavelet tree* općenito je struktura podataka koja se koristi za komprimirano spremanje nizova. Listovima ovog stabla odgovaraju znakovi abecede koja je korištena, a u svakom čvoru spremljen je bit vektor koji govori kojem podskupu pripada znak niza. Općenito stablo rekurzivno dijeli korištenu abecede u parove podskupove.

3. Algoritam1: Konstrukcija bit vektora

3.1 Opis implementacije algoritma

Prvi algoritam koji je ostvaren u ovom projektu koristi se za konstrukciju 2 bit vektora, Bl i Br. Funkcija nazvana *createBitVectors* kao ulazne parametre prima k, BWT, adresu od implicitnog grafa *graph*, adresu reda Q te referencu bit vektora Bl i Br. Ulazni parametar k je cijeli broj koji predstavlja duljinu k-torke koja će se koristiti, BWT je niz znakova koji predstavlja Burrows-Wheeler transformaciju objašnjenu u prethodnom poglavlju. Ostali parametri pokazuju na podatke koji se u algoritmu koriste budući da je funkcija korištena za implementaciju ovog algoritma tipa void. Općenito bit vektor Br koncipiran je tako da je vrijednost njegove i-te pozicije 1 ako je S[SA(i), SA(i)+k-1] najveći desno ponavljajući niz te je S[SA(i)] leksikografski najveći ili najmanji sufiks početnog niza takav da mu je S[SA(i), SA(i)+k-1] sufiks. Kod bit vektora Bl situacija je malo drugačija pa bit vektor Bl na i-toj poziciji ima 1 ako i samo ako S[SA(i), SA(i)+k-1] nije najveći desno ponavljajući niz te je S[SA(i)] leksikografski najveći sufiks početnog niza kojem je S[SA(i), SA(i)+k-1] prefiks. Budući da je ulaz u ovu funkciju BWT, u *main* funkciji izračunava se prvo sufiksno polje SA koje je potrebno za izračunavanje vrijednosti BWT koja se koristi u danjem računu. Također u *main* dijelu programa računa se i BWT koji se koristi u ovom algoritmu. U ovoj programskoj implementaciji sama funkcija koja ostvaruje algoritam 1 poziva se iz funkcije za ostvarivanje algoritma 2. Na početku algoritma izračunavaju se WT te LCP korištenjem gotove knjižnice [2]. Također računa se vektor C za sve znakove te se kreira vektor *lcpFull* iz glavnog vektora kako je opisano u prethodnom poglavlju. U ostatku algoritma pojavljuju se dvije for petlje. Prva for petlja kreće se kroz sve vrijednosti promijenjenog LCP vektora. Ako se 2 uzastopna sufiksa podudaraju u više od k znakova, odnosno ako je vrijednost LCP veća od k to nam je znak da u idućim koracima možemo dobiti vrijednost 1 za Bl ili Br te se taj slučaj prati podizanjem zastavice *open*. Ako je vrijednost LCP-a u nekom koraku točno k, odnosno 2 uzastopna sufiksa podudaraju se u točno k znakova prati se indeks varijablom *kIndex* jer postoji mogućnost da je vrijednost Br(i) jednaka 1. Ako ovo nije slučaj, odnosno 2 uzastopna sufiksa se podudaraju u manje od k znakova, provjeravamo možemo li dodati jedinice u bit vektore. Ako je trenutna vrijednost *kIndex* veća od vrijednosti lijeve granice, odnosno indeksa najmanjeg sufiksa koji se podudara sa svojim sljedbenicima u najmanje k

znakova, postavljamo vrijednost Br na 1 za lijevu i desnu granicu. Ovdje također počinje i izgradnja implicitnog grafa kojem postavljamo početne čvorove. Također u red Q dodajemo indeks čvora. Zatim se provjerava može li se postaviti vrijednost bit vektora Bl na 1. To se radi ako je najveći indeks u kojem se razlikuju 2 uzastopna BWT znaka veći od lijeve granice te tada sve vrijednosti Bl(C[BWT[j]]), gdje se j kreće od lijeve granice pa sve do trenutne vrijednosti i, postavljamo na 1 ako trenutna vrijednost znaka nije znak % ili \0. Na kraju petlje provjerava se je li neki od indeksa za kojeg je vrijednost bit vektora Bl 1 ujedno i indeks najveće desno ponavljajućeg niza te ako je postavljamo vrijednost vektora Bl za taj indeks u 0.

3.2 Primjer rezultata

>1

AGTCATTGCA

>2

GGTTCATT

>3

CGTATGC

Pilikom čitanja datoteke stvara se niz S=AGTCATTGCA%GGTTCATT%CGTATGC\0. Dodatno se, prije izvođenja algoritama A1 i A2, stvaraju vrijednosti pomoćnih struktura koje su navedene u Tablici 3.1.

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

Tablica 3.1 Vrijednosti početnih struktura

Vektor *C* ima sljedeće vrijednosti: C[\0] = 0, C[%] = 1, C[A] = 3, C[C] = 8, C[G] = 13, C[T] = 19. Kako za prolaz ovog algoritma koristimo k = 3, iz gornje tablice možemo uočiti kako postoje četiri uzastopna sufiksa koji imaju LCP veći ili jednak kao k, što znači da ćemo četiri puta pokušati osvježiti jedinice u bit vektorima Bl i Br.

Prva iteracija u kojoj je lcp[i]=k=3 je za indeks i=7 te se tada *kIndex* postavlja na vrijednost 7 i zastavica *open* se postavlja na *true*. U idućoj iteraciji je vrijednost LCP-a manja od k, a kako je zastavica *open* postavljena provjerava se je li *kIndex* veći od lijeve granice (lb = 6). Kako to vrijedi, postavljamo Br[6] = 1 i Br[7] = 1. Indeks 22 koji je pohranjen u *lastDiff* označava slučaj kada je duljina prefiksa veća ili jednaka k, a bwt[i] i bwt[i-1] su različiti. U tom slučaju se ispunjava uvjet kada je *lastDiff* veći od lijeve granice (lb = 21). Bl osvježava jedinice dva puta (lb = 21, lb = 22) u indeksima Bl[C[c] – 1] (moramo oduzimati 1 zbog toga što smo sve pomaknuli ulijevo za jedan). Za j = 21 dobivamo c = BWT[21] = 'T', a C[T] = 27 jer smo prije

njega još osam puta naišli na slovo T. Za j = 22 dobivamo c = BWT[22] = 'G', a C[G] = 18 iz istog razloga.

Na isti način se algoritam izvršava dok se ne prijeđe vrijednost veličine LCP-a. Nakon toga se još provjerava postoji li neki indeks za koji su istovremeno vrijednosti Bl i Br jednaki jedan. U ovom slučaju nisu te vrijednosti vektora ostaju nepromijenjene. U tablici 3.2 mogu se vidjeti vrijednosti bit vektora nakon izvršavanja algoritma 1.

i	Br	Bl
0	0	0
1	0	0
2	0	0
3	0	0
4	0	0
5	0	1
6	1	0
7	1	0
8	0	0
9	0	0
10	0	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0	1
18	0	0
19	0	0
20	0	0
21	0	0
22	0	0
23	1	0
24	1	0
25	0	0
26	0	1
27	0	1

Tablica 3.2 Prikaz bit vektora nakon izvršavanja A1

4. Algoritam 2: Konstrukcija implicitnog grafa

4.1 Opis implementacije algoritma

Rezultat drugog algoritma je konstruirani implicitni graf koji služi za reprezentaciju de Bruijnovog grafa. Za konstrukciju implicitnog grafa koriste se bit vektori dobiveni algoritmom 1 opisanom u prethodnom poglavlju. Kao što je naznačeno algoritam 1 osim konstrukcije bit vektora dodaje i inicijalni čvor u graf. U ovom algoritmu potrebno je najprije odrediti koliko je konačni broj čvorova te dodati završne čvorove u implicitni graf. Graf je kako je već napomenuto vektor elemenata tipa *Node* čija je struktura objašnjena u poglavlju 2.1. Funkcija u kojoj je u ovom projektnom rješenju ostvarena implementacija ovog algoritma naziva se *createCompressedGraph* te prima 3 ulazna parametra k, BWT i zastavica ispisa. Značenje i vrijednosti parametara k i BWT jednake su kao u algoritmu 1 te su opisani u prethodnom poglavlju. Posljednji parametar služi kao zastavica te ako je njena vrijednost jednaka *originalPrint* na ekran se ispisuje prve 2 vrijednosti rezultata, ako je njena vrijednost jednaka *lastYearPrint* na ekran se ispisuje sve 4 vrijednosti rezultata. Ako ne postoji niti jedna od vrijednosti zastavica rezultat se ne ispisuje na ekran već se sve 4 vrijednosti ispisuju u datoteku. Posljednji parametar primarno se koristi u svrhu testiranja programskog koda. Sam algoritam računa intervale sufiksnog polja za sve desno maksimalne podnizove duljine k. Nakon toga za svaki od tih podnizova koje možemo označiti kao *w* računa *cw* intervale sufiksnog polja u kojem *c* predstavlja element abecede korištene u ovom programskom rješenju. Koristi se sada niz *u* koji predstavlja vrijednost *cw* te se računa *cu* interval te tako dalje redom. Cijela procedura tako započinje sa svim desno maksimalnim podnizovima duljine k te ih proširuje koliko god može, na sve moguće načine, znak po znak. Kao što je naznačeno na početku, algoritam 1 pronalazi sve desno maksimalne podnizove duljine k te dodaje početne čvorove u graf G koji predstavljaju njihove reprezentacije, a također se u red Q dodaju identifikatori za pojedini dodani čvor. Algoritam 2 koji gradi implicitni graf redom uzima sve te čvorove prema indeksima pohranjenim u redu Q te ih proširuje ulijevo. Funkcija koja je korištena za pronalazak *cw* intervala naziva se *intervals_symbols* te je to gotova funkcija iz knjižnice sdsl-lite [2]. Zatim se za svaki niz *cw* dalje testira je li njegov prefiks duljine *k* desno-maksimalni ponavljajući niz. Provjera se izvršava korištenjem desnog bit vektora Br u kojem su

označene granice sufiksnih intervala svim desno maksimalno ponavljajućih nizova. Ako test pokaže da taj prefiks je desno maksimalan, račun ovdje staje te algoritam prelazi na idući čvor koji je određen idućom vrijednosti reda Q. Za slučaj kada funkcija *intervals_symbols* pronađe više od jednog znaka c, niz *w* je lijevo maksimalni podniz te se u tom slučaju u graf dodaje novi čvor koji reprezentira prefiks duljine k niza *cw* s kojim se dalje nastavlja izvođenje algoritma. Zadnji slučaj, ako *w* nije lijevo maksimalni podniz, njegov pripadni čvor se proširuje, odnosno povećava se vrijednost duljine niza te se nastavlja istraživanje pronađenog *cw* intervala.

4.2 Primjer rezultata

Tablica 4.1 prikazuje početni implicitni graf generiran s A1.

id	len	lb	size	suffix_lb
0	3	6	2	6
1	3	23	2	23
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	1	0	1	0
7	1	1	1	1
8	1	2	1	2

Tablica 4.1 Prikaz početnog implicitnog grafa

Prije početka izvršavanja algoritma dodaju se završni čvorovi grafa ovisno o broju različitih sekvenci (zadnja 3 stupca). Algoritam započinje s čvorom id = 0, koji predstavlja podniz „ATT“. Za w interval [6..7] pronalazimo samo jedan interval za znak 'C' [10..11] što znači da w interval nije lijevo maksimalan. Kako je *ones* paran broj, a Br[10] = 0, ulazimo u prvi uvjet u kojem se *extendable* postavlja na istinitu vrijednost, a čvor sa id = 0 se mijenja u (4,10,2,6). U idućem koraku se to ponovo događa za znak 'T' [21..22], a čvor sa id = 0 se mijenja u (5 , 21, 2, 6). U trećem koraku su pronađena dva intervala, za znak 'T' [26..26] i za znak 'G' [17..17]. Ispunjava se drugi uvjet i dodajemo dva nova čvora u graf. Dalje ispitujemo čvor id = 1. Za interval w [23..24] ponovno pronalazimo dva intervala te se drugi uvjet ispunjava i dodajemo još dva čvora. Graf se može vidjeti u Tablici 4.2.

id	len	lb	size	suffix_lb
0	5	21	2	6
1	3	23	2	23
2	3	5	1	5
3	3	17	1	17
4	3	26	1	26
5	3	27	1	27
6	1	0	1	0
7	1	1	1	1
8	1	2	1	2

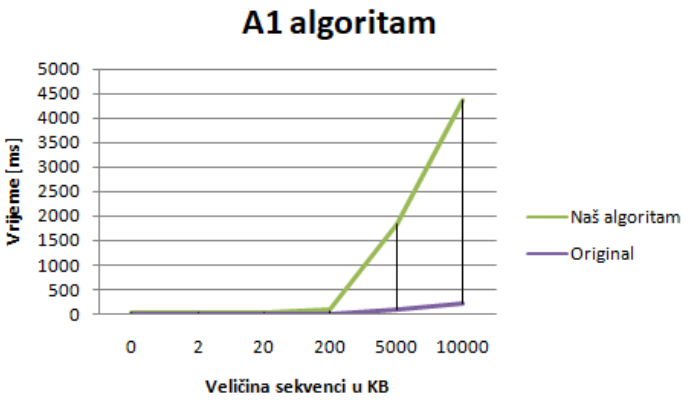
Tablica 4.2 Prikaz implicitnog grafa nakon prolaska čvora 0

Idući čvor je čvor id = 6 koji predstavlja jedan od završnih čvorova (u ovom slučaju '\0'). Kako on nije lijevo maksimalan prvi uvjet se ispunjava i mijenja se u (2, 8, 1, 0). U idućem koraku ponovno mijenjamo čvor u (3, 13, 1, 0). U idućem koraku se pronalazi znak 'T' intervala [23..23], međutim kako je u tom slučaju Br[23] = 1, to znači da smo došli do granice i ništa se ne mijenja, već se prelazi na idući čvor. Ovaj algoritam se izvršava sve dok ima čvorova u redu Q te dobivamo graf kao u Tablici 4.3.

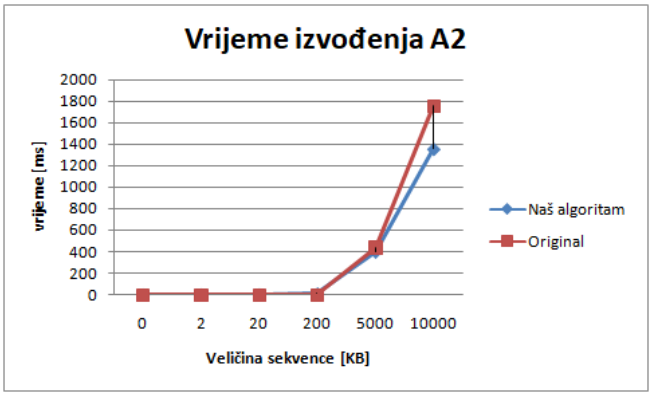
id	len	lb	size	suffix_lb
0	5	21	2	6
1	3	23	2	23
2	6	12	1	5
3	4	4	1	17
4	5	15	1	26
5	3	27	1	27
6	3	13	1	0
7	3	25	1	1
8	4	14	1	2

Tablica 4.3 Prikaz implicitnog grafa nakon završetka algoritma

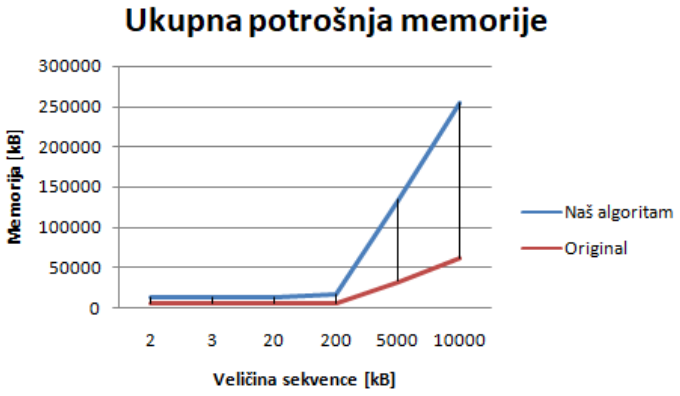
5. Analiza rezultata



Slika 5.1. Vremenske performanse za A1



Slika 5.2. Vremenske performanse za A2



Slika 5.3. Zauzeće memorije

6. Zaključak

Mjerenja iz prethodnog poglavlja izvršena su na virtualnom računalu sa specifikacijama: RAM: 3.5 GB, procesor: Intel Core i7-3632QM, koristeći programsku knjižnicu Chrono [3] za mjerenje vremena i Unix-ov alat *time* sa zastavicom –v koji ispisuje i zauzeće memorije. Moguće pogreške u mjerenju vremena mogu nastati zbog korištenja virtualnog računala.

U usporedbi s originalnom izvedbom može se vidjeti kako naša implementacija troši puno više memorije, a za vrijeme izvođenja dobivamo kako je A1 puno sporiji u našoj izvedbi, dok je A2 brži u našoj izvedbi. U usporedbi sa studentskom izvedbom od prošle godine uočili smo kako naša implementacija troši manje memorije (otprilike 20% manje), a vrijeme izvođenja nismo uspjeli izmjeriti.

Pokušavajući otkriti uzrok velikom trošenju memorije, našli smo kako funkcije za automatsku izgradnju struktura podataka (SA, LCP, WT) iz SDSL knjižnice [2] troše jako puno memorije. Kao primjer možemo uzeti LCP: kod velike duljine sekvence (10,000 KB), generiranje LCP strukture zauzme između 20-30% cjelokupnog zauzeća memorije programa iako je veličina LCP strukture nakon stvaranja očekivane veličine. Zbog toga bi se moglo razmisliti o tome da se koristi neka druga knjižnica za stvaranje struktura ili vlastita implementacija istih.

Naša implementacija algoritama je prošla sve testove točnosti i možemo zaključiti da radi ispravno. Međutim, kao što je već rečeno, moguća su različita poboljšanja kako bi algoritam dobro radio.

7. Literatura

[1] 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.

[2] SDSL - Succinct Data Structure Library, https://github.com/simongog/sdsl-lite, 23.1.2018.

[3] Date and time utilities, http://en.cppreference.com/w/cpp/chrono, 23.1.2018.