

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4446

**ODREĐIVANJE PREKLAPANJA MEĐU
OČITANJIMA U POSTUPKU *DE NOVO*
SASTAVLJANJA GENOMA**

Tin Široki

Zagreb, lipanj 2016.

Zagreb, 16. ožujka 2016.

ZAVRŠNI ZADATAK br. 4446

Pristupnik: **Tin Široki (0036470305)**
Studij: **Računarstvo**
Modul: **Računarska znanost**

Zadatak: **Određivanje preklapanja među očitanjima u postupku de novo sastavljanja genoma**


Opis zadatka:

U ovome radu potrebno je istražiti problem određivanja preklapanja među očitanjima u postupku de novo sastavljanja genoma za podatke dobivene Nanopore MinION tehnologijom. Očitavanja dobivena ovom metodom sekvenciranja su i do dva reda veličine dulja od očitavanja dobivenih tehnologijama prethodnih generacija, ali uz visoku pogrešku koja može biti i do 40%. Ovakva pogreška predstavlja problem u fazi preklapanja koja je tipično vremenski najzahtjevniji dio u postupku de novo sastavljanja genoma. U sklopu ovog rada koristiti sufixno polje za određivanje identičnih preklapanja te dinamičko programiranje za pronalazak preklapanja koja nisu potpuno identična. Algoritamsko rješenje problema potrebno je implementirati u programskim jezicima C i C++. Performanse programa potrebno je usporediti s postojećim rješenjima.

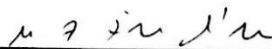
Zadatak uručen pristupniku: 18. ožujka 2016.
Rok za predaju rada: 17. lipnja 2016.

Mentor:

Doc. dr. sc. Mirjana Domazet-Lošo

Djelovoda:

Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za
završni rad modula:


Prof. dr. sc. Siniša Srblić

Zahvala

Zahvaljujem se mentorici doc. dr. sc. Mirjani Domazet-Lošo na ukazanoj podršci.

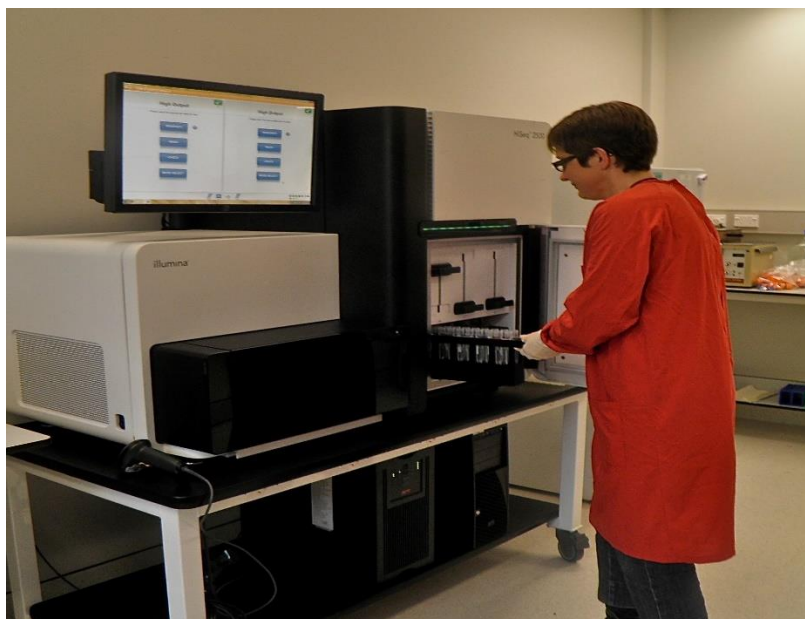
Sadržaj

Uvod	1
1. Notacija.....	4
2. Definicija problema	5
3. Strukture podataka.....	9
3.1. Sufiksno stablo	9
3.2. Sufiksno polje	11
3.3. Inverzno sufiksno polje	12
3.4. Prošireno sufiksno polje	12
4. Needleman-Wunschov algoritam	14
4.1. Globalno poravnanje nizova.....	14
5. Određivanje preklapanja.....	18
5.1. pristupi određivanja preklapanja	18
5.2. Određivanje preklapanja kombiniranim pristupom	20
5.3. Određivanje identičnih preklapanja	22
5.3.1. Izgradnja proširenog sufiksnog polja	22
5.3.2. Određivanje identičnih preklapanja pomoću sufiksnog stabla	24
5.3.3. Koncept sufiksnog stabla implementiran proširenim sufiksnim poljem	25
5.4. Određivanje optimalnih poravnanja	28
6. Analiza programske potpore.....	29
7. Zaključak	35
8. Literatura	36
Naslov, sažetak i ključne riječi	39
Dodatak A: Upute za korištenje programske potpore	40

Uvod

Područje bioinformatike je interdisciplinarno područje koje se bavi razvojem postupaka i programske podrške za interpretiranje bioloških podataka. Jedan problem iz potonje domene je i sastavljanje genoma iz očitavanja dobivenih iz uređaja za sekvenciranje. Sastavljanje genoma se svodi na što točniju rekonstrukciju izvornog genoma iz dobivenih očitavanja („Sequence assembly“, 2016). *De novo* sastavljanje genoma je postupak sastavljanja genoma bez unaprijed poznatog okvira genoma („*De novo* transcriptome assembly“, 2016). Za ilustraciju, zamislimo da imamo 20 knjiga koje smo izrezali na sitne komadiće i koje je potrebno spojiti natrag u jednu izvornu knjigu. U našem slučaju, komadići teksta su očitavanja iz uređaja za sekvenciranje, preciznije nizovi oznaka za neku od nukleotidnih baza. Nukleotidne baze su adenin, gvanin, citozin, timin i uracil, a označavamo ih redom s A, G, C, T i U. Današnji moderni algoritmi sastavljanja genoma su algoritmi temeljeni na grafovima (Šikić & Domazet-Lošo, 2013). Razlikujemo dvije osnovne metode: metoda temeljena na de Bruijn grafovima (de Bruijn, 1946) i preklapanje razmještaj konsenzus metoda (*eng. Overlap-Layout-Consensus, OLC*) (Myers i ostali, 2000). Najpoznatiji alati koji koriste OLC metodu su Celera (Myers i ostali, 2000) i SGA (Simpson & Durbin, 2012). U ovom radu sam obradio traženje preklapanja, prvi korak potonjeg postupka sastavljanja genoma.

Očitavanja, ovisno o generaciji korištene tehnologije, variraju u duljini, točnosti i cijeni (Šikić & Domazet-Lošo, 2013). Trenutno najčešće korištena tehnologija sekvenciranja je tehnologija koju je razvila tvrtka Illumina („Illumina | Sequencing and array-based solutions for genetic research“, 2016). Njihova tehnologija pripada drugoj generaciji tehnologija za sekvenciranje, a kao metodu sekvenciranja koristi se metoda sekvenciranja sintezom. Duljina očitavanja su između 50 i 300 baza, s točnosti od 98 %. Slika 1. prikazuje uređaj Illumina HiSeq 2500 („HiSeq 2500 Ultra-High-Throughput Sequencing System“, 2016). Velika cijena uređaja i njegove dimenzije mogu biti ograničavajući faktori u njegovoj primjenjivosti.



Slika 1. Uređaj Illumina HiSeq 2500.

Nanopore MinION uređaj za sekvenciranje je mali, prenosivi uređaj kojeg je moguće koristiti s običnim osobnim računalom. Slika 2. prikazuje uređaj nanopore MinION („Oxford Nanopore Technologies“, 2016).



Slika 2. Nanopore MinION uređaj.

Uređaj pripada trećoj generaciji tehnologija, a kao metodu sekvenciranja koristi *nanopore* sekvenciranje. Nanopore su mali otvori čije su dimenzije reda veličine nanometra. Kroz nanopore prolazi ionska struja koja se mijenja ovisno o molekulama koje prolaze kroz ili su blizu nanopora. Informacija o promjeni struje koristi se za identifikaciju molekula („Oxford Nanopore Technologies“, 2016). Prednosti ove tehnologije su cijena, prenosivost i duljina očitavanja, a nedostatak je veliki postotak pogreške. Očitavanja dobivena ovom metodom sekvenciranja su i do dva reda veličine dulja od očitavanja dobivenih tehnologijama prethodnih generacija, ali uz visoku pogrešku koja može biti i do 40%. Ovakva pogreška predstavlja problem u fazi preklapanja, koja je tipično vremenski najzahtjevniji dio u postupku *de novo* sastavljanja genoma („*De novo transcriptome assembly*“, 2016).

U ovom radu obradio sam postupak traženja preklapanja među očitavanjima dobivenih korištenjem nanopore minION uređaja. Zbog velikog postotka pogreške dobivenih očitavanja potrebno je ostvariti robusnost u postupku traženja preklapanja. Robusnost je moguće dobiti traženjem optimalnog poravnanja korištenjem Needleman-Wunschovog algoritma (Needleman & Wunsch, 1970). Ovaj algoritam koristi dinamičko programiranje („Dynamic programming“, 2016). Ovim postupkom moguće je pronaći optimalna preklapanja između parova očitavanja. Iako su dobivena preklapanja optimalna, ovakav pristup ima veliko ograničenje zbog vremenske kvadratne složenosti. Zbog veličine ulaznih podataka, koja je čak i za genome jednostavnijih vrsta reda veličine nekoliko 10-taka gigabajta, i ranije spomenute složenosti, za ubrzavanje postupka traženja preklapanja moguće je primijeniti heuristiku. U ovom radu sam obradio pristup koji kombinira traženje identičnih preklapanja s algoritmom optimalnog poravnanja.

1. Notacija

Niz $S = a_0 a_1 a_2 \dots a_{n-1}$ definiramo kao niz znakova duljine n nad abecedom Σ duljine σ . Znak $\$$ koji nije dio abecede Σ koristimo za oznaku kraja niza S , te je leksikografski najveći znak u abecedi. Znak na i -tom mjestu a_i označavamo sa $S[i]$. Podniz niza S označavamo sa $S[i, j] = a_i a_{i+1} \dots a_j$, dodatni uvjet $0 \leq i \leq j \leq n - 1$ određuje da indeksi moraju biti unutar niza S , te da je i uvijek manji. Sufiks niza S definiram kao $S[i, n - 1]$, $0 \leq i \leq n - 1$, odnosno podniz kojemu se kraj poklapa s originalnim nizom, te će u radu biti označen s A_i . Prefiks niza S definiramo kao $S[0, j]$, $0 \leq j \leq n - 1$, odnosno podniz kojemu se početak poklapa s originalnim nizom.

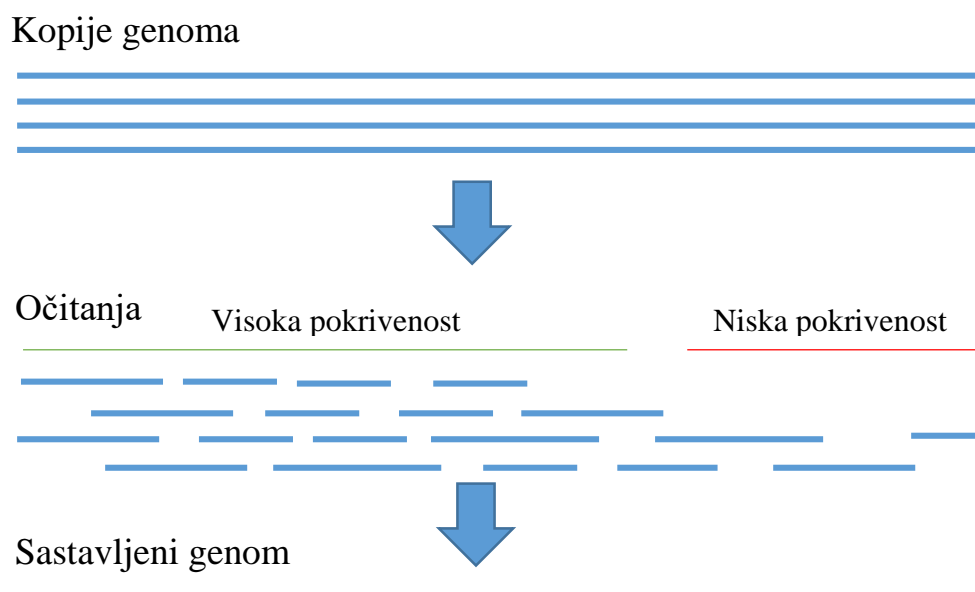
Na primjer, za niz $S = \text{GCATGCU}$. Podniz niza od pozicije 1 do 3 je: $S[1,3] = \text{CAT}$, prefiks niza do pozicije 3 je: $S[0,3] = \text{GCAT}$, i sufiks niza počevši od pozicije 3 je: $A_3 = S[3,6] = \text{TGCU}$.

Identična preklapanja dva niza S_i i S_j definiramo kao identične podnizove nizova S_i i S_j , $S_j[s1, (s1 + l)] = S_j[s2, (s2 + l)]$ koje određuje uređena trojka, te označavamo s $P_{i,j} = (s1, s2, l)$. Gdje su i i j indeksi preklapajućih nizova; $s1$ je indeks početka podniza niza s indeksom i , $s2$ je početak podniza s indeksom j , a l je duljina preklapanja.

Na primjer, za dva niza $S_1 = \text{GCATGCU}$ i $S_2 = \text{GCUTTA}$. Preklapanje nizova počinje na poziciji 4 prvog niza i 0 drugog niza, njegova duljine je 3, te preklapanje zapisujemo s: $P_{1,2} = (4, 0, 3)$. Preklapajući niz je: $P = \text{GCU}$.

2. Definicija problema

Metoda sekvenciranja sačmaricom (*eng.* shotgun) („Shotgun sequencing“, 2016) je najkorištenija metoda sekvenciranja. Naziv je dobilo prema izgledu uzorka nastalog ispaljivanjem sačmarice. Originalne kopije genoma se lome na veliki broj dijelova koji se sekvenciraju čime dobivamo puno kratkih neporedanih očitavanja. Cilj postupka sastavljanja genoma je preklopiti očitavanja te iz preklapanja rekonstruirati traženi genom. Slika 3. prikazuje preklapanje očitavanja na genom duljine G . Uređaj za sekvenciranje na svom izlazu daje N očitavanja duljine L . Prosječna pokrivenost genoma je određena s izrazom: $C = (N * L) / G$ (Šikić & Domazet-Lošo, 2013). Pokrivenost obično nije uniformna, što je vidljivo na slici 3. gdje prepoznajemo dijelove s visokom i niskom pokrivenosti. Ovisno o pokrivenosti pojedinih dijelova ovisi težina postupka sastavljanja. Dijelovi s vrlo niskom pokrivenosti mogu uzrokovati procijepe u sastavljenom genomu.



Slika 3. Preklapanja očitavanja na sastavljeni genom.

Jedna od metoda slaganja genoma je i u uvodu spomenuta preklapanje razmještaj konsenzus metoda (*eng.* *Overlap-Layout-Consensus, OLC*) (Šikić & Domazet-Lošo, 2013). U ovom radu obradio sam njegovu prvu fazu - traženje preklapanja. Cilj ove

faze je pronaći preklapanja između očitavanja. Ulazni skup podataka za postupak traženja preklapanja je skup očitavanja dobivenih iz uređaja za sekvenciranje, pohranjenih u tekstualnoj datoteci koristeći fasta („FASTA format“, 2016) format. Format specificira zapis u obliku koji se sastoji od jednog opisnog retka te proizvoljnog broja podatkovnih redaka. Opisni redak počinje znakom „>“ nakon kojeg slijedi opis podataka. Podatkovni redci sadrže znakovni niz od kojeg svaki znak predstavlja kôd za jednu od nukleinskih baza.

Ulazni skup U je skup od N očitavanja $\{S_0, S_1, S_2 \dots S_{N-1}\}$, gdje svako od očitavanja predstavlja jedan znakovni niz iz abecede $\{A, C, G, T\}$. Pronalaženje preklapanja između N nizova ulaznog skupa U svodi se na traženje preklapanja između svih mogućih kombinacija parova kojih je $N * (N - 1)/2$. Zanimaju nas najbolja preklapanja između dva niza, odnosno preklapanja s najvećom sličnosti po gore danoj definiciji.

S obzirom na velik postotak pogreške u očitanjima dobivenih iz uređaja odabrane tehnologije, značajna su i preklapanja koja nisu u potpunosti identična. Sukladno s tim potrebno je definirati neku mjeru sličnosti između dva niza. Hammingova udaljenost (Hamming, 1950) dva niza jednaka je broju mjesta na kojima se dani nizovi razlikuju. Na primjer, Hammingova udaljenost nizova $S = \text{GCATGCU}$ i $T = \text{GATTACA}$ je 4, zato što se dani nizovi razlikuju na pozicijama 1,2,4,6 kojih je ukupno 4.

Hammingova definicija udaljenosti nizova vrijedi samo za nizove istih duljina. 1965. godine Vladimir Levenshtein je uveo mjeru uređivanja koja predstavlja minimalni broj potrebnih modifikacija kako bi se iz niza S dobio niz T (Levenshtein, 1966) Moguće modifikacije su umetanje, brisanje i izmjena.

Na primjer, mjera uređivanja za niz $S = \text{GCATGCU}$ u niz $T = \text{GATTACA}$ je 4 jer smo nad nizom S napravili 4 modifikacije kako bi dobili niz T , modifikacije su redom (indeksi u nizu na kojima se provodi uređivanje počinju od 0):

1. Brisanje C na 1. mjestu: $\text{GCATGCU} \rightarrow \text{GATGCU}$
2. Izmjena $G \rightarrow T$ na 3. mjestu: $\text{GATGCU} \rightarrow \text{GATTCU}$
3. Dodavanje umetanje A na 4. mjestu: $\text{GATTCU} \rightarrow \text{GATTACU}$
4. Izmjena $U \rightarrow A$ na 6. mjestu: $\text{GATTACU} \rightarrow \text{GATTACA}$

Mjeru sličnosti dva biološka slijeda definirali su Needleman i Wunsch (Needleman & Wunsch, 1970). Pokazano je da su problemi maksimiziranja sličnosti i minimiziranja udaljenosti uređivanja ekvivalentni (Sellers, 1974). Zadavanjem vrijednosti za slaganje, neslaganje i razmak, koji mogu biti različito odabrani s obzirom na biološke razloge, dobivena je mjera sličnosti dva niza.

Na primjer, odaberemo vrijednosti: slaganje = 1, neslaganje = -1, razmak = -1, tražimo sličnost nizova $S = \text{GCATGCU}$ i $T = \text{GATTACA}$. Provođenjem Needleman-Wunsch algoritma (opisan u poglavlju 5.) dobivamo poravnanje nizova, $S' = \text{GCATG} - \text{CU}$ i $T' = \text{G} - \text{ATTACA}$. Znak „-“ u poravnanju nizovima predstavlja razmak. Međusobna sličnost zadanih nizova je 0. Vrijednost je dobivena kao suma vrijednosti za svaku od pozicija u nizovima. Slaganja nizova S' i T' su na pozicijama 0,1,2,6, ukupno 4 slaganja što daje vrijednost sličnosti 4. Neslaganja su na pozicijama 4, 7, ukupno 2 neslaganja što daje vrijednost sličnosti -2. Razmaci su na pozicijama 1,5 što daju vrijednost sličnosti -2. Suma svih vrijednosti je: $4 - 2 - 2 = 0$.

Koristimo li za traženje optimalnih preklapanja Needleman-Wunschov algoritam (Needleman & Wunsch, 1970), koji ima kvadratnu vremensku složenost dolazimo do poteškoća (poglavlje 4.). Na primjer:

Ljudski genom ima približno $3 * 10^9$ parova baza. Kodiramo li svaki par s jednim znakom za kojeg je potreban jedan bajt, dobivamo veličinu genoma od 3 gigabajta. Pretpostavimo da je pokrivenost očitanjima 30 puta, time dobivamo 90 gigabajta očitavanja. Pretpostavimo da je duljina pojedinog očitavanja 10 tisuća, pa iz toga slijedi da radimo s približno $9 * 10^6$ očitavanja. Tražimo parove preklapanja svaki sa svakim kojih ima $4 * 10^{13}$. Zbog spomenute kvadratne složenosti, za poravnanje dva očitavanja od 10000 znakova je potrebno 10000^2 operacija. Dolazimo do broja potrebnih operacija od približno $4 * 10^{21}$. Današnja najbrža računala na svijetu mogu napraviti oko $1 * 10^{15}$ operacija s pomičnim zarezom u sekundi, odnosno FLOPS (*eng.* floating-point operations per second) („FLOPS“, 2016). Čak i na super računalu izračun bi trajao približno 50 dana.

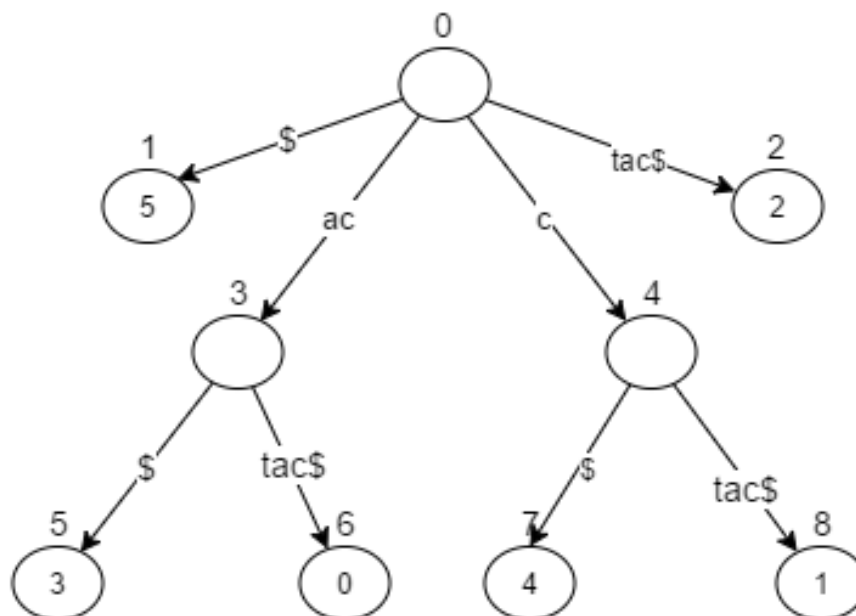
Zbog ovakvog vremena izvođenja, problem je pojednostavljen na način da nije potrebno uvijek odrediti optimalna preklapanja za svaki od parova. Cilj je napraviti rješenje koje radi kompromis između pronalaženja optimalnih preklapanja i brzine izvođenja, odnosno napraviti rješenje koje će koristiti heuristikute na osnovu parametara zadanih od korisnika raditi kompromis između vremena potrebnog za određivanje preklapanje i njihove kvalitete.

3. Strukture podataka

U ovom poglavlju dat ću pregled struktura podataka korištenih u radu a to su: sufiksno stablo, sufiksno polje, inverzno sufiksno polje i prošireno sufiksno polje.

3.1. Sufiksno stablo

Problem pronalaska podniza u nizu možemo usporediti s problemom traženja pojma u knjizi. Ako je zadani niz velik, naivno slijedno pretraživanje nije praktično. Kao rješenje ovog problema moguće je izgraditi indekse koji će nam pokazivati na kojem mjestu u nizu se nalazi pojedini podniz. Pretraživanje niza se svodi na pretraživanje sortiranih podataka u indeksa. Ako smo u listi indeksa pronašli traženi podniz izravno dolazimo i do njegove pozicije u nizu. Jedna takva često korištena struktura je i sufiksno stablo (Gusfield, 1997).



Slika 4. Sufiksno stablo za niz $S=actac\$$

Sufiksno stablo (*eng. suffix tree*) nad nizom S duljine n i abecede Σ je stablo koje sadrži sve sufikse niza S . Znak $\$$ je posebni znak koji se dodaje na kraj niza S i ne postoji u abecedi Σ , a leksički je manji od svakog znaka iz Σ . Znak $\$$ na kraju niza osigurava da svaki sufiks niza S završava u listu stabla. Listovi stabla sadrže brojeve koji predstavljaju poziciju pripadajućeg sufiksa u nizu S . Oznake grana zadržavaju podnizove niza S . Slika 4. prikazuje sufiksno stablo za niz $S=actac\$$; brojevi iznad čvorova su redni brojevi čvorova koji inače nisu dio sufiksnog stabla, ali su ovdje označeni radi lakšeg daljnjeg objašnjavanja. Sve sufikse niza S moguće je rekonstruirati *preorder* obilaskom stabla („Tree traversal“, 2016). *Preorder* obilazak stabla na slici 5. obilazi redom čvorove: 0, 1, 3, 5, 6, 4, 7, 8, 2. Za rekonstrukciju svih sufiksa prolazimo *preorder* stablo i rekonstruiramo jedan niz R . Nizove iz grana koje prolazimo prema listovima dodajemo na kraj niza R , a nizove iz grana koje prolazimo prema korijenu brišemo s kraja niza R . Kada dođemo do lista, niz R je sufiks koji se nalazi na poziciji u nizu S zabilježenu u tom listu. Rekonstrukcija sufiksa iz stabla na slici 4. izgleda ovako:

0. 0, $R=$, niz je na početku prazan i nalazimo se u korijenu (čvor 0).
1. $0 \rightarrow 1$, $R=\$$, došli smo do lista 5 te je sufiks $A_5=\$$.
2. $1 \rightarrow 0$, $R=$, brišemo $\$$ s kraja.
3. $0 \rightarrow 3$, $R=ac$, dodajemo ac na kraj R .
4. $3 \rightarrow 5$, $R=ac\$$, došli smo do lista 3 te je sufiks $A_3=ac\$$.
5. $5 \rightarrow 3$, $R=ac$, brišemo $\$$ s kraja.
6. $3 \rightarrow 6$, $R=actac\$$, došli smo do lista 0 te je sufiks $A_0=actac\$$.
7. $6 \rightarrow 3$, $R=$, brišemo $tac\$$ s kraja.
8. $3 \rightarrow 0$, $R=$, brišemo ac s kraja.
9. $0 \rightarrow 4$, $R=c$, dodajemo c na kraj R .
10. $4 \rightarrow 7$, $R=c\$$, došli smo do lista 4 te je sufiks $A_4=c\$$.
11. $7 \rightarrow 4$, $R=c$, brišemo $\$$ s kraja.
12. $4 \rightarrow 8$, $R=ctac\$$, došli smo do lista 1 te je sufiks $A_1=ctac\$$.
13. $8 \rightarrow 4$, $R=c$, brišemo $tac\$$ s kraja.
14. $4 \rightarrow 0$, $R=$, brišemo c s kraja.
15. $4 \rightarrow 2$, $R=tac\$$, došli smo do lista 2 te je sufiks $A_2=ctac\$$.

Sufiksno stablo nad nizom S duljine n je moguće izgraditi naivnim pristupom u vremenu $O(n^2)$ (Šikić & Domazet-Lošo, 2013). U stablo se dodaje jedan po jedan sufiks. Za svaki sufiks se dodaje jedan list, te ako je potrebno dodaju se unutarnji čvorovi i grane. Ukkonenov algoritam (Ukkonen, 1995) je algoritam koji izgrađuje sufiksno stablo u linearnom vremenu.

Za probleme koji uspoređuju više nizova, npr. traženje preklapanja između nizova ili traženje najduljeg zajedničkog podniza, potrebno je izgraditi sufiksno stablo od nizova koje obrađujemo. Sufiksno stablo izgrađeno od N različitih nizova $\{S_0, S_1, S_2 \dots S_{N-1}\}$ zove se poopćeno sufiksno stablo (*eng. generalized suffix tree*) (Šikić & Domazet-Lošo, 2013).

3.2. Sufiksno polje

Sufiksno polje (Manber & Myers, 1993), kraće SA (*eng. suffix array*), je sortirano polje pozicija svih sufiksa u nekom nizu. Nastalo je kao memorijski manje zahtjevna alternativa sufiksnom stablu. U praksi sufiksno stablo za ulazni niz duljine n zahtjeva minimalno $10n$, obično oko $15n$ okteta, a sufiksno polje oko $4n$ okteta (Kasai, Lee, Arimura, Arikawa, & Park, 2001). Zbog veličine bioloških podataka ova ušteda može biti od značaja, pa sam u svojoj implementaciji koristio ovu strukturu.

Na primjer, sufiksno polje niz $S = \text{actac\$}$ je $SA = [0\ 3\ 1\ 4\ 2\ 5]$. Vrijednosti polja su uvijek nenegativni cijeli brojevi. Vrijednost i -tog broja u polju predstavlja poziciju i -tog sufiksa u originalnom nizu S , gdje se pod i -tim sufiksom smatra i -ti sufiks u listi sortiranih sufiksa, ilustrirano u tablici 1.

i	$SA[i]$ -ti sufiks	$SA[i]$
0	actac\$	0
1	ac\$	3

2	ctac\$	1
3	c\$	4
4	tac\$	2
5	\$	5

Tablica 1. Sufiksno polje za niz actac\$.

Sufiksno polje je moguće dobiti izravno iz sufiksnog stabla preorder obilaskom stabla. Ostvariva je i direktna izgradnja iz niza S , i to u linearnoj vremenskoj složenosti. Jedan od algoritama koji ostvaruje takvu izgradnju je *sais* (eng. *suffix array induced sorting*) algoritma (Nong, Zhang, & Chan, 2009). Algoritam se temelji na induciranom sortiranju. Za izgradnju sufiksnog polja koristio sam implementaciju ovog algoritma, napravljenu od strane grupe studenata na prethodnom akademskom zadatku (Franjić, Mijačika, Sabolić, & Široki, 2016).

3.3. Inverzno sufiksno polje

Inverzno sufiksno polje je inverzno polje pripadajućem sufiksnom polju, koje ću označavati s *rank* (hrv. naziv: rang), čija definicija je određena s izrazom: $SA[k] = i \rightarrow rank[i] = k$ (Kasai i ostali, 2001). *Rank* polje nam zapravo govori na kojoj poziciji u sufiksnom polju se nalazi indeks i .

3.4. Prošireno sufiksno polje

Polje najduljih zajedničkih prefiksa, kraće LCP (eng. *longest common prefix array*) polje, je polje nenegativnih cijelih brojeva. Sufiksno polje uz pripadajuće LCP polje zove se prošireno sufiksno polje (eng. *enhanced suffix array*) (Abouelhoda, Kurtz, & Ohlebusch, 2004). Polje LCP sadrži duljine zajedničkih prefiksa susjednih sufiksa iz liste sortiranih sufiksa. Polje LCP nad nizom S duljine n s pripadajućim sufiksnim poljem SA je polje duljine n . Prvi član je po definiciji 0, $LCP[0] = 0$. *LCP*, na i -toj poziciji je duljina najduljeg prefiksa sufiksa $S[SA[i - 1]]$ i $S[SA[i], n]$. *LCP* polje je moguće izračunati prilikom izgradnje sufiksnog polja ili u linearnom vremenu (Kasai

i ostali, 2001) iz već izgrađenog sufiksnog polja. Primjer proširenog sufiksnog polja za niz $actac\$$ je dan u tablici 2. Na primjer, u polju SA na poziciji 0 je sufiks $A_0=actac\$$, njegov susjed je sufiks na poziciji 1 $A_3=ac\$$. Zajednički prefiks ova dva sufiksa je ac , njegova duljina je 2. Ta vrijednost je zapisana u LCP polju na poziciji koja odgovara sufiksu $A_3=ac\$$, a to je pozicija 1.

i	$SA[i]$ -ti sufiks	$SA[i]$	$LCP[i]$
0	$actac\$$	0	0
1	$ac\$$	3	2
2	$ctac\$$	1	0
3	$c\$$	4	1
4	$tac\$$	2	0
5	$\$$	5	0

Tablica 2. Prošireno sufiksno polje za niz $actac\$$.

4. Needleman-Wunschov algoritam

Needleman-Wunschov algoritam (Needleman & Wunsch, 1970) je algoritam koji izračunava sličnost dva niza S i T , i poravnanje nizova za izračunatu sličnost. Algoritam je temeljen na dinamičkom programiranju („Dynamic programming“, 2016). Dinamičko programiranje je pristup rješavanja složenih problema rastavljanjem problema na manje potprobleme. Rješenja potproblema se zapisuju u memoriju. U daljnjem izračunu se riješeni potproblemi ne moraju ponovno izračunavati, nego se njihova vrijednost dohvati iz memorije.

4.1. Globalno poravnanje nizova

Globalno poravnanje dva niza S i T duljina n i m možemo predstaviti s matricom od dva retka. Prvi redak sadrži niz S' a drugi niz T' , poredak znakova iz originalnih nizova S i T je zadržan, praznine su moguće na bilo kojem mjestu (označavamo ih s „-“) (Šikić & Domazet-Lošo, 2013). Jedno moguće poravnanje nizova $S = \text{GCATGCT}$ i $T = \text{GATTACA}$ je prikazano u tablici 3.

G	C	A	T	G	-	C	T
G	-	A	T	T	A	C	A

Tablica 3. Poravnanje nizova $S = \text{GCATGCT}$ i $T = \text{GATTACA}$

Ovisno o biološkim razlozima zadaju se vrijednosti za slaganje, neslaganje i razmak. Bdujemo preklapanja s obzirom na zadane vrijednosti. Na primjer, odaberemo vrijednosti: slaganje = 1, neslaganje = -1, razmak = -1. Za primjer iz tablice 3. postoje 3 slaganja, 2 neslaganja i 3 razmaka, što daje ukupno vrijednost poravnanja 0. Cilj algoritma je pronaći optimalno poravnanje, odnosno poravnanje koje ima najveću vrijednost.

Sva moguća poravnanja moguće je prikazati pomoću matrice V dimenzija $(n + 1) * (m + 1)$ (tablica 4.). Pojedino poravnanje predstavlja put polja $[0,0]$ do polja $[n-1,m-1]$, uz ograničeno kretanje u tri smjera: desno, dolje i dijagonalno (desno i dolje). Kretanja desno i dolje predstavljaju umetanje znakova, dok kretanje dijagonalno može predstavljati slaganje ili neslaganje. Svaka od akcija se boduje prema zadanim vrijednostima te je potrebno pronaći put s najvećom sumom vrijednosti napravljenih akcija.

Označimo s d vrijednost za razmak. Funkcija $w(A, B)$ prima dva znaka A i B , ako su predani znakovi isti funkcija vraća vrijednost za slaganje, a u situaciji kada su različiti vraća vrijednost za neslaganje. Algoritam popunjava matricu V , dimenzija $(n + 1) * (m + 1)$. Vrijednost matrica V na poziciji i, j označavat ću s $V[i, j]$. Matrica V popunjava se po sljedećim pravilima:

$$V[i, j] = \begin{cases} 0, & i = 0 \text{ i } j = 0 \\ d * i, & i = 0 \\ d * j, & j = 0 \\ \max \begin{cases} V[i - 1, j - 1] + w(S_i, T_j) \\ V[i - 1, j] + d \\ V[i, j - 1] + d \end{cases}, & \text{inače} \end{cases}$$

Algoritam prvo postavlja vrijednosti za redak 0 i stupac 0, po danom izrazu. U sljedećem koraku obilazi matricu na uobičajen način, od polja $[1,1]$ do $[n,m]$ (redak po redak). Vrijednost svakog polja se određuje kao maksimum od:

- vrijednosti polja koje je iznad uvećanom za vrijednost razmaka
- vrijednost polja lijevo uvećanu za vrijednost razmaka
- vrijednost polja dijagonalno (iznad i lijevo) uvećanu za vrijednost slaganja ili neslaganja (ovisno o podudaranju znakova u i -tom retku i j -tom stupcu).

Ako je kasnije potrebno rekonstruirati poravnanje potrebno je pamtit i iz kojeg polja je vrijednost došla. Ovaj algoritam određuje vrijednosti za svako polje jednim prolaskom matrice. Pošto je matrica dimenzija $(n + 1) * (m + 1)$, vremenska složenost algoritma je $O(n*m)$. Ako želimo rekonstruirati optimalno poravnanje, a ne samo koristiti njegovu vrijednost, potrebno je u svakom polju pamtit i od kud je došla vrijednost za to polje, tada je prostorna složenost isto $O(n*m)$. Ako nam je potrebna samo vrijednost poravnanja, za izračun vrijednosti svakog polja dovoljno je znati

vrijednosti iz njegovog retka i retka iznad, što daje složenost od $O(\max(n,m))$ (Šikić & Domazet-Lošo, 2013).

	S		G	C	A	T	G	C	U
T		0	1	2	3	4	5	6	7
	0	0	-1	-2	-3	-4	-5	-6	-7
G	1	-1	1	0	-1	-2	-3	-4	-5
A	2	-2	0	0	1	0	-1	-2	-3
T	3	-3	-1	-1	0	2	1	0	-1
T	4	-4	-2	-2	-1	1	1	0	-1
A	5	-5	-3	-3	-1	0	0	0	-1
C	6	-6	-4	-2	-2	-1	-1	1	0
A	7	-7	-5	-3	-1	-2	-2	0	0

Tablica 4. Matrica poravnanja za nizove $S = \text{GCATGCU}$ i $T = \text{GATTACA}$

Tablica 4. prikazuje matricu poravnanja za nizove $S = \text{GCATGCU}$ i $T = \text{GATTACA}$. Preciznije, matrica poravnanja je dio matrice koji je obojan sivo. Radi lakšeg razumijevanja, u prvi redak i stupac su dodani nizovi koje poravnavamo, a u drugom retku i stupcu su indeksi polja. Vrijednosti za svako polje su dobivene opisanim postupkom. Vrijednost na polju $V[7,7]$ je vrijednost poravnanja. Strelice u polju pokazuju na polje iz kojeg je izračunata njegova vrijednost. Neka polja imaju više strelica zato što je izračunata vrijednost za to polje dolaskom po svakoj od strelica ista. U toj situaciji je moguće da postoji više jednakovrijednih optimalnih poravnanja, kao što je i slučaj u ovom primjeru. Crvene strelice predstavljaju neslaganje, zelene slaganje dok crne predstavljaju razmak. Poravnanje se rekonstruira prateći strelice od polja $V[7,7]$ do polja $V[0,0]$. Žuto su označena sva polja kojima prolaze putevi koji rekonstruiraju optimalna poravnanja. Različitim putevima rekonstrukcije dobivamo različita optimalna poravnanja. Jedan način rekonstrukcije stvara poravnanje na sljedeći način:

1. $V[7,7] \rightarrow V[6,6]$, $S' = U$ i $T' = A$, neslaganje.
2. $V[6,6] \rightarrow V[5,5]$, $S' = CU$ i $T' = CA$, slaganje.
3. $V[5,5] \rightarrow V[4,5]$, $S' = -CU$ i $T' = ACA$, umetanje slova A u niz T .

4. $V[4,5] \rightarrow V[3,4]$, $S' = G - CU$ i $T' = TACA$, neslaganje.
5. $V[3,4] \rightarrow V[2,3]$, $S' = TG - CU$ i $T' = TTACA$, slaganje.
6. $V[2,3] \rightarrow V[1,2]$, $S' = ATGC - CU$ i $T' = ATTACA$, slaganje.
7. $V[1,2] \rightarrow V[1,1]$, $S' = CATGC - CU$ i $T' = -ATTACA$, umetanje slova C u S'
8. $V[1,1] \rightarrow V[0,0]$, $S' = GCATGC - CU$ i $T' = G - ATTACA$, slaganje.

5. Određivanje preklapanja

5.1. pristupi određivanja preklapanja

Razlikujemo tri osnovna pristupa u traženju preklapanja:

- Traženje identičnih preklapanja
- Traženje preklapanja dinamičkim programiranjem (Needleman & Wunsch, 1970)
- Kombinirani pristup (heuristički pristup)

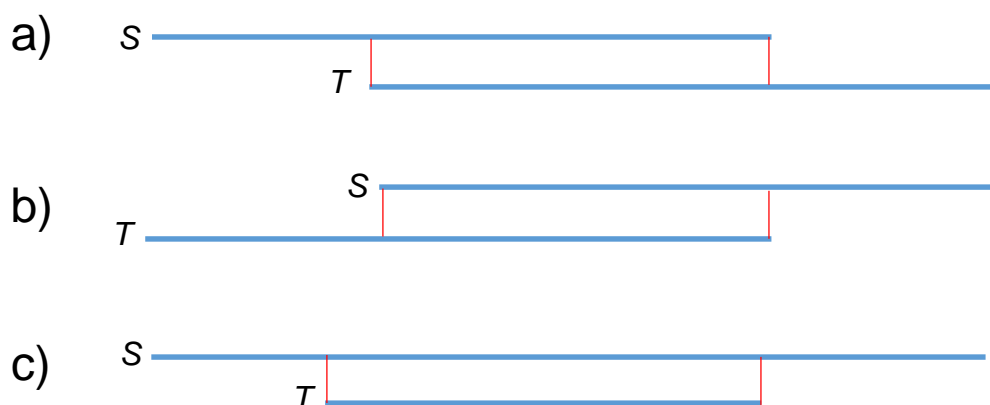
Traženje identičnih preklapanja pronalazi samo identična preklapanja, ovaj postupak je moguće ostvariti korištenjem indeksiranja, npr. sufiksnog stabla (Šikić & Domazet-Lošo, 2013). Ograničenje ovakvog pristup je moguća pogreška u očitanjima između kojih tražimo preklapanja, ako je postotak pogreške velik pronađena preklapanja će biti kratka. S obzirom da u ovom radu koristim očitavanja dobivena iz tehnologije s velikim postotkom pogreške ovaj pristup nije odgovarajući.

Određivanje preklapanja dinamičkim programiranjem ostvarivo je korištenjem Needleman-Wunsch algoritma opisanog u poglavlju 4. uz manje preinake. Ne želimo pronaći globalno poravnanje, nego želimo pronaći poravnanje samo dijela nizova. Ovakva poravnanja zovu se polu-globalna poravnanja (Šikić & Domazet-Lošo, 2013). Slika 5. prikazuje globalno i polu-globalno poravnanje za nizove S = CAGCACTTGGATTCTCGG i T = CAGCGTGG.

Globalno poravnanje:	CAGCACTTGGATTCTCGG CAGC----G--T----GG
Polu-globalno poravnanje:	CAGCA-CTTGGATTCTCGG ---CAGCGTGG-----

Slika 5. Primjer globalnog i polu-globalnog poravnanja (Šikić & Domazet-Lošo, 2013)

Polu-globalnih poravnanja su poravnanja u kojima ne kažnjavamo praznine na početku ili na kraju podniza. Ovakva preklapanja je moguće odrediti Needleman-Wunschovim algoritmom uz napravljene modifikacije ovisno o slučaju preklapanja. Modifikacije ovisno o slučajevima različito inicijaliziraju početni redak i stupac u matrici poravnanja, te traže kraj poravnanja u zadnjem retku ili stupcu. Na slici 6. prikazana su tri slučaja preklapanja nizova S i T : slučaj a) sufiks niza S prefiksom niza T , b) prefiks niza S sa sufiksom niza T i c) niz T se u cijelost preklapa s nizom S .



Slika 6. Različita preklapanja dva niza

Modifikacije Needleman-Wunschovog algoritma za slučajeve sa slike 6. Niz S je stavljen u stupce a niz T u retke matrice poravnanja:

- Inicijaliziramo prvi stupaca s nulama, prvi redak s $j(\text{indeks stupca}) \cdot d$ (vrijednost za razmak), te za kraj poravnanja uzimamo maksimum zadnjeg retka.
- Inicijaliziramo prvi stupac s $d \cdot i(\text{indeks retka})$, prvi redak s nulama, te za kraj poravnanja uzimamo maksimum zadnjeg stupca.
- Inicijaliziramo prvi stupac s $d \cdot i$, prvi redak nulama, te za kraj poravnanja uzimamo maksimum zadnjeg retka.

Ovaj postupak pronalazi optimalna preklapanja, odnosno preklapanja koja imaju maksimalnu sličnost. Nedostatak ovog algoritma je njegova vremenska složenost. Ako s l označimo očitavanje, a s L ukupnu duljinu svih očitavanja. Složenost traženja poravnanja između svih očitavanja je dana izrazom:

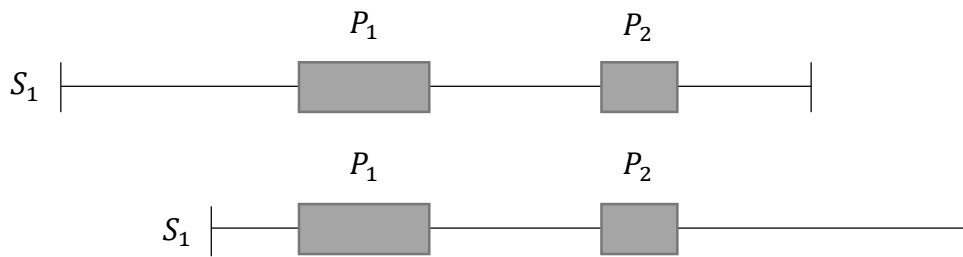
$$O(\sum_{i,j} l_i l_j) = O(\sum_i l_i \sum_j l_j) = O(\sum_i l_i L) = O(L^2) \text{ (Šikić \& Domazet-Lošo, 2013).}$$

Vidimo da je složenost kvadratna s ukupnom duljinom očitavanja. S obzirom na veličinu bioloških podataka ova složenost nije prihvatljiva (primjer dan u 2. poglavlju).

Zbog iznesenih karakteristika traženja identičnih preklapanja i traženja preklapanja dinamičkim programiranjem, u svojoj implementaciji koristio sam kombinirani pristup koji je opisan u sljedećem poglavlju.

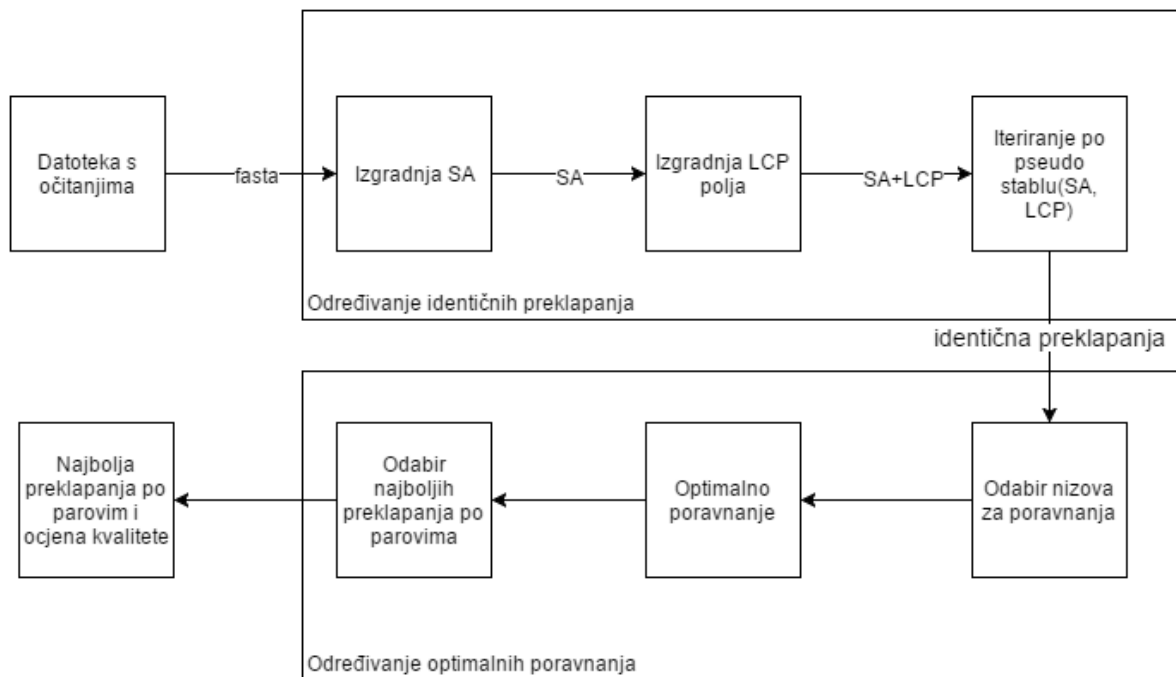
5.2. Određivanje preklapanja kombiniranim pristupom

Za rješavanje definiranog problema odlučio sam se za pristup koji prvo traži identična preklapanja, nakon čega dodatno radi optimalna poravnanja dijelova nizova između identičnih preklapanja. Kao preklapanja dva niza uzima preklapanje s najvećom sličnosti. Definiram dva parametra pretrage, a to su: *duljina* i *udaljenost*. *Duljina* predstavlja minimalnu duljinu traženih identičnih preklapanja, a *udaljenost* predstavlja minimalnu udaljenost između identičnih preklapanja za koje se provodi postupak poravnanja. Ideja je odabirom male vrijednosti za parametar *duljina* dobiti više preklapanja između dva niza, a zatim u drugoj fazi dodatno poravnati dijelove nizova koji su između identičnih preklapanja i udaljeni su manje od parametra *udaljenosti*. Ovakvim postupkom manja preklapanja spajaju u veća. Za konačno preklapanje između dva niza odabire se ono preklapanje koje ima najveću sličnost. Slika 7. prikazuje preklapanja između niza S_1 i S_2 . Pronađena su dva identična preklapanja P_1 i P_2 , koja će se spojiti u jedno ako je njihova udaljenost manja od zadanog parametra *udaljenosti*. Važno je primijetiti da udaljenosti P_1 i P_2 u nizovima S_1 i S_2 obično nisu iste, te obje moraju zadovoljiti uvjet za provedbu postupka spajanja.



Slika 7. Preklapanje između nizova S_1 i S_2

Protočni sustav je podijeljen u dva velika koraka, a to su određivanje identičnih preklapanja te određivanje optimalnih poravnanja. Prvi korak slijedno izgrađuje strukture podataka: sufiksno polje i LCP polje. Iteracijom po izgrađenim strukturama određuju se preklapanja između svih parova očitavanja/nizova odjednom. Ovakvim pristupom je onemogućeno paralelno izvođenje. Drugi korak odabire kandidate nizova za poravnanja te provodi njihova poravnanja. Slika 8. prikazuje cjelokupni cjevovod traženja preklapanja.



Slika 8. Cjevovod traženja preklapanja.

5.3. Određivanje identičnih preklapanja

U postupku određivanja identičnih preklapanja koristio sam prošireno sufiksno polje zbog ranije spomenute uštede u prostornim zahtjevima u odnosu na sufiksno stablo. Postupak određivanja identičnih preklapanja opisat ću kroz tri dijela: izgradnja proširenog sufiksnog polja, traženje identičnih preklapanja pomoću sufiksnog stabla, te koncept sufiksnog stabla implementiran proširenim sufiksnim poljem.

5.3.1. Izgradnja proširenog sufiksnog polja

Program na svoj ulaz dobiva n očitavanja pohranjenih u datoteci, prilikom spremanja tih n očitavanja u memoriju svakom od njih se dodjeljuje identifikator, slijedno od 0 do $n-1$ ($id_0, id_1, \dots, id_{n-1}$). Sva očitavanja slijedno se spremaju u niz S , te se na kraj očitavanja dodaje znak $\$$ kao oznaka kraja niza. U polje cijelih brojeva Poz (duljine n) stavljaju se početne pozicije pojedinih očitavanja. Na primjer, $Poz[0]$ sadrži poziciju početka očitavanja id_0 u nizu S , $Poz[1]$ poziciju početka id_1 itd. Polje Poz se koristi kako bi kasnije mogli odrediti kojem očitanju pripada pojedini niz.

Kao što je spomenuto u poglavlju 3.2 za izgradnju sufiksnog polja koristio sam sais algoritam. Za izgradnju polja LCP iz sufiksnog polju implementirao (slika 9.) sam algoritam koji radi u linearnoj vremenskoj složenosti (Kasai i ostali, 2001).

```

for (i = 0; i < duljina; i++) {
    rank[SA[i]] = i;
}
h = 0;
for (i = 0; i < duljina; i++) {
    if (rank[i] > 1) {
        k = SA[rank[i] - 1];
        while (S[i + h] == S[k + h]) {
            h++;
        }
        LCP[rank[i]] = h;

        if (h > 0) {
            h--;
        }
    }
}

```

Slika 9. Algoritam izgradnje polja LCP

Slika 9. prikazuje isječak kôda s implementacijom algoritma u programskom jeziku C. Inverzno sufiksno polje (poglavlje 3.3.) je u kôdu označeno s `rank[]`. Polje se stvara jednim prolaskom po indeksima od i je 0 do duljine sufiksnog polja, u kôdu na slici 9. ostvareno s prvom for petljom, varijabla *duljina* sadrži vrijednost duljine sufiksnog polja. Prisjetimo se oznake za sufiks s početkom na i -tom mjestu A_i . Dodatno definiramo funkciju $lcp(A, B)$ koja vraća duljinu najduljeg zajedničkog prefiksa od predanih nizova. Efikasnost algoritma se temelji na teoremu kojega je Kasai dokazao u svom radu (Kasai i ostali, 2001). Teorem glasi:

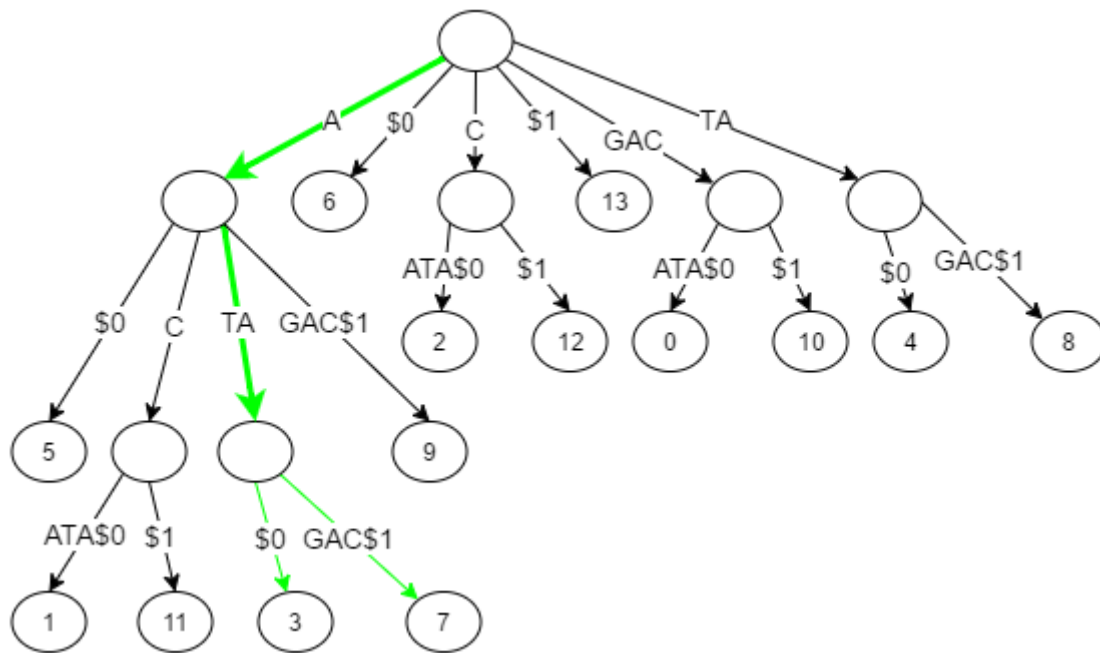
Ako $LCP[p] = lcp(A_{j-1}, A_{i-1}) > 1$ onda $LCP[q] = lcp(A_k, A_i) \geq LCP[p] - 1$.

Ovaj teorem govori da kada je lcp vrijednost sufiksa A_{i-1} i njegovog sljedbenika jednaka h , tada sufiks A_i i njegov sljedbenik imaju lcp vrijednost od najmanje $h - 1$. Ovaj teorem omogućava uspoređivanje sufiksa A_i s njegovi sljedbenikom od h -tog indeksa. Ako je vrijednost h jednaka 1 ili manje, tada se sufiksi uspoređuju od početka.

5.3.2. Određivanje identičnih preklapanja pomoću sufiksnog stabla

Unutarnji čvorovi stabla su svi čvorovi koji nisu listovi. Svaka grana je označena s nekim podnizom originalnog niza. Konkatenacija svih grana od korijena do pojedinog čvora je put do čvora. Putevi do unutarnjih čvorova su podnizovi koji se ponavljaju u originalnom nizu, a takva ponavljanja su upravo ono što tražimo. Postupak ću pojasniti kroz primjer (slika 10.). Tražimo identična preklapanja između nizova $S = GACATA\$0$ i $T = ATAGAC\$1$. Na kraj niza S i T dodali smo oznake za kraj niza $\$0$ i $\$1$. Iz danih nizova izgradimo poopćeno sufiksno stablo (slika 10.). Nazovimo unutarnji čvor zajedničkim čvorom svih originalnih nizova do čijih se oznaka kraja niza (npr. $\$1, \$2..$) može doći iz tog čvora. Određivanje zajedničkih čvorova je moguće ostvariti *postorder* obilaskom stabla („Tree traversal“, 2016). Postorder obilazak prvo posjećuje listove, pamtimmo sve oznake kraja nizova, time je u roditeljskom čvoru moguće odrediti kojim originalnim nizovima je taj čvor zajednički. Putevi do svih zajedničkih čvorova neka dva ulazna niza njihovi su zajednički podnizovi. Postupak pronalaženja zajedničkog podniza ATA (slika 10. označeno debelom zelenom linijom) je:

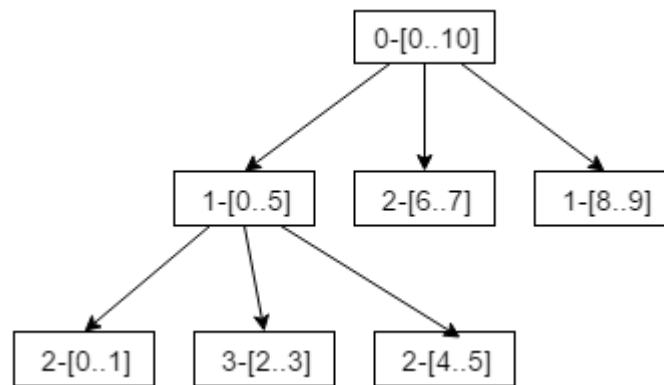
- Obradujemo list 3 (tanka zelena linija), pamtimmo $\$1$.
- Obradujemo list 7 (tanka zelena linija), pamtimmo $\$2$.
- Obradujemo roditeljski čvor (od lista 3 i 7), znamo da je on zajednički originalnim nizovima s oznakom kraja $\$1$ i $\$2$ a to su nizovi S i T .
- Put do ovog čvora je ATA (označeno debelom zelenom linijom), iz čega znamo da je taj put jedan zajednički podniz nizova S i T .



5.3.3. Koncept sufiksnog stabla implementiran proširenim sufiksnim poljem

Koncept sufiksnog stabla implementiran proširenim poljem koristiti lcp intervale kako bi obilazio lcp stablo (Abouelhoda i ostali, 2004). Lcp interval je interval u sufiksnom polju koji sadrži pozicije susjednih sufiksa s istim lcp vrijednostima, označavat ću ga s trojkom $\text{lcp}-[a..b]$, a i b su lijeva i desna granica intervala a lcp je njegova vrijednost. Lcp stablo je konceptualno stablo (nikada se ne izgrađuje) koje nam ilustrira način na koji se obilazi prošireno sufiksno polje.

Na slici 11. prikazano je lcp stablo za niz $S=acaaacatat\$$, u tablici 5. nalaze se *LCP* polje, *SA* i sortirani sufiks za niz *S*. Na primjer, čvor 1-[0-5] nam govori da su sve vrijednosti *LCP* polja u intervalu od 0 do 5 barem 1, što je i vidljivo u *LCP* tablici. Prvih 6 poredanih sufiksa počinje znakom *a*, odnosno imaju zajednički prefiks *a*. Čvor 2-[0..1] nam govori da je na *LCP*[1] vrijednost 2, odnosno sufiksi na poziciji *SA*[0] i *SA*[1] imaju zajednički prefiks *aa*. Ako malo bolje pogledamo ove primjere možemo neformalno zaključiti da su čvorovi lcp stabla ekvivalentni unutarnjim čvorovima sufiksnog stabla.



Slika 11. lcp stablo za niz $S=acaaacatat\$$.

i	$SA[i]$	$LCP[i]$	$SA[i]$ -ti sufiks
0	2	0	aaacatat\$
1	3	2	aacatat\$
2	0	1	acaaacatat\$
3	4	3	acatat\$
4	6	1	atat\$
5	8	2	at\$
6	1	0	caaacatat\$
7	5	2	catat\$
8	7	0	tat\$
9	9	1	t\$
10	10	0	\$

Tablica 5. SA i LCP vrijednosti za niz $S=acaaacatat\$$.

Tražimo identična preklapanja između svih parova od n očitavanja, takvih parova je $n*(n-1)/2$. Značajna su sva preklapanja između dva para, što znači da je za spremanje preklapanja potrebno $n*(n-1)/2$ lista. Većina parova neće uopće imati preklapanja te će time većina lista biti prazne, pa sam u implementaciji stvarao liste tek kad su potrebne.

Preklapanja je moguće odrediti obilaskom konceptualnog lcp stabla pomoću algoritma (Kasai i ostali, 2001) danog na slici 12. Algoritam obilazi stablo koristeći pomoćni stog („Stack (abstract data type)“, 2016). Svaki put kada se posjeti pojedini čvor taj se lcp interval obrađuje, ako je njegova lcp vrijednost veća ili jednaka vrijednosti odabranog parametra duljina. Odnosno, sva preklapanja kraća od

parametra duljina se zanemaruju. Lcp interval nam govori od koje do koje pozicije u SA se nalaze pozicije sufiksa s tom lcp vrijednosti. Vratimo se na primjer sa slike 11. i tablice 5., npr. obrađujemo čvor 3- $[2..3]$ iz kojeg slijedi $SA[3]$ i $SA[2]$ imaju lcp vrijednost 3. $SA[3]=4$ i $SA[2]=0$, sufiksi na pozicijama 4 i 0 su: $A_4=acatat\$$ i $A_0=acaaacatat\$$, njihov zajednički prefiks je *aca*. Odnosno, ovaj lcp interval određuje identično preklapanje podnizova $S[4,6]$ i $S[0,3]$. Recimo da je niz $S=acaaacatat\$$ izgrađen iz dva očitavanja $S_1=aca$ i $S_2=aacatat$, njihovi identifikatori bi bili id0 i id1 te polje Poz (poglavlje 5.3.1.) bi izgledalo ovako: $Poz=[0,3]$. Iz polja Poz odredimo da podniz $S[0,3]$ pripada očitanju id0 (podniz počinje na poziciji 0), podniz $S[4,6]$ pripada očitanju id1 (podniz počinje na poziciji 1). Ovime je dobiveno identično preklapanje između očitavanja id0 i id1: $P_{0,1}=(0,1,3)$.

```
void traverseTree() {
    stack<LcpInterval> stack;
    LcpInterval lastInterval;
    stack.push(LcpInterval(0, 0, EMPTY));

    for (int i = 1; i < length; i++) {
        int lb = i - 1;
        while (LCP[i] < stack.top().lcp) {
            lastInterval = stack.top();
            lastInterval.rb = i - 1;
            stack.pop();
            processor->process(&lastInterval);
            lb = lastInterval.lb;
        }
        if (LCP[i] > stack.top().lcp) {
            stack.push(LcpInterval(LCP[i], lb, EMPTY));
        }
    }
}
```

Slika 12. Algoritam obilaska LCP stabla (Kasai i ostali, 2001).

5.4. Određivanje optimalnih poravnanja

Ova faza kao ulaz dobiva liste identičnih preklapanja po parovima očitavanja. Optimalna poravnanja se provode za nizove između identičnih preklapanja ako je njihova duljina manja od parametra *udaljenost*. Poravnanja se određuju Needleman-Wunsch algoritmom koji je opisan u poglavlju 4.

Kao konačno preklapanje između dva očitavanja se uzima ono preklapanje koje ima najveću vrijednost, vrijednost identičnih dijelova je njihova duljina a poravnati dijelovi se boduju po danoj definiciji sličnosti. Na kraju korisnik određuje koja su mu preklapanja dovoljno dobra (zadaje donju granicu vrijednosti). Sva preklapanja koja imaju vrijednost veću od donje granice se zapisuju u datoteku i za njih se ispisuje statistika.

Kao poboljšanje rješenja moguće je implementirati paralelno izvođenje poravnanja.

6. Analiza programske potpore

Rezultati svih testiranja prikazanih u ovom poglavlju napravljena su na prijenosnom računalu čije su karakteristike navedene na slici 13.

Model računala	HP Probook 4540s
Procesor	Intel® CORE™ i5-2450M CPU; 2.5GHz
RAM	RAM 8GB
Operacijski sustav	64-bit Windows 10

Slika 13. Karakteristike računala.

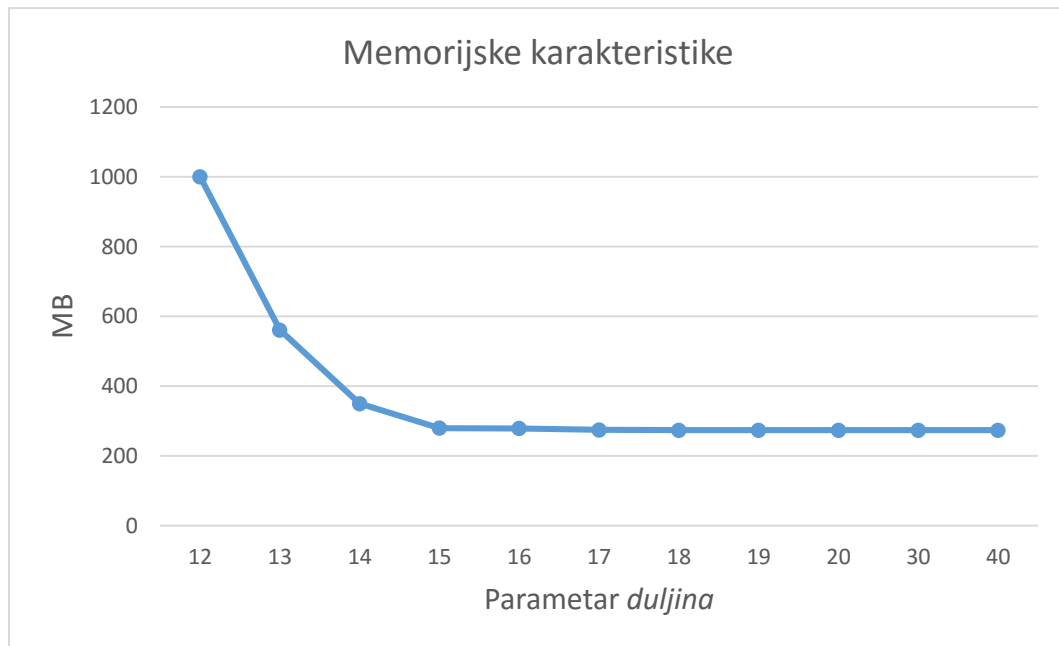
Testiranje sam napravio nad očitanjima genoma bakterije „*Escherichia coli*“ dobivenih nanopore MinION tehnologijom (Quick, Quinlan, & Loman, 2014). Datoteka „*HighQualityTwoDirectionReads.fasta*“ s očitanjima nad kojima je izvršeno testiranje dostupna je u direktoriju „*test_data*“ (priloženo kao dodatak radu zajedno s razvijenom programskom potporom). Ukupan broj očitavanja je 3471, prosječna duljina očitavanja je 6330 nukleotida (nt), a standardna devijacija je 2908. Duljina najduljeg očitavanja je 47422 nt, a duljina najkraćeg je 130 nt.

Određivanje preklapanja ovisi o dva parametra: *duljina* i *udaljenost* (poglavlje 5.2.). Podsjetimo se, parametar *duljina* određuje donju granicu duljine za identična preklapanja, a parametar *udaljenost* gornju granicu razmaka između identičnih preklapanja za koje se provodi poravnanje.

Parametar *udaljenost* je fiksno postavljen na vrijednost 200 nt za testiranja prikazana na slikama 14.-16.

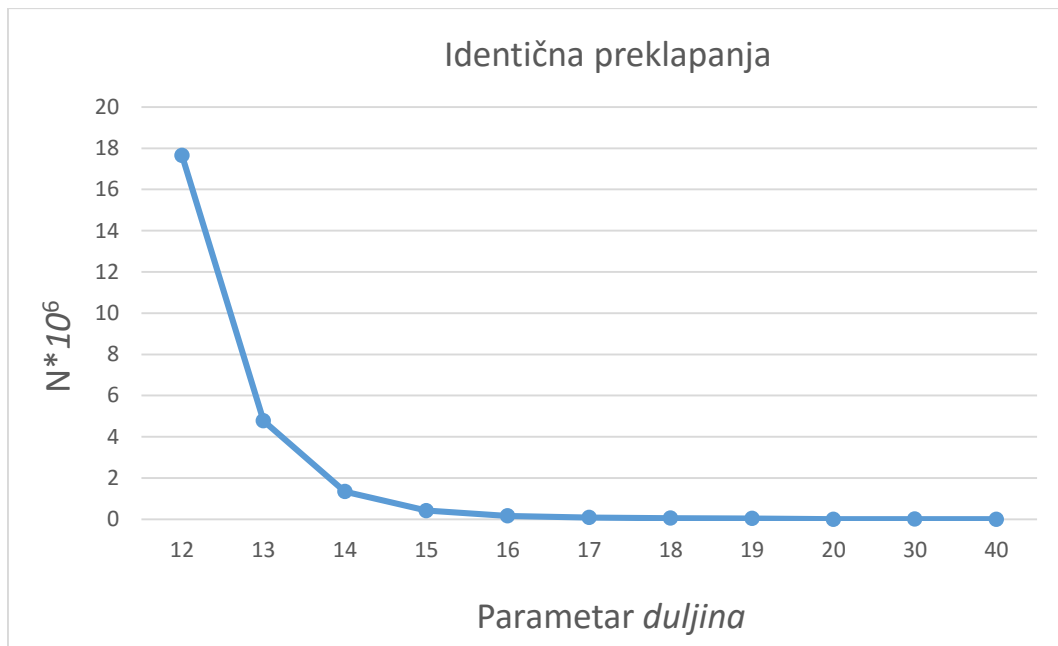
Slika 14. prikazuje maksimalno memorijsko zauzeće određivanja preklapanja u ovisnosti o parametru duljina. Vidimo da memorijsko zauzeće pada s porastom parametra duljina, te se stabilizira na vrijednosti od ~200 MB. Razmotrimo memorijsko zauzeće struktura podataka korištenih prilikom određivanja identičnih preklapanja: sva očitavanja predstavimo kao jedan niz znakova duljine n (svaki n zauzima 1 oktet), sufiksno polje je polje cijelih brojeva duljine n (svaki n zauzima 4 okteta) i *LCP* polje je polje cijelih brojeva (svaki n zauzima 4 okteta). Ukupno $9n$

okteta. Obrađujemo 3471 očitavanja prosječne duljine 6330, iz čega slijeda da je $n = 3471 * 6330 \approx 22 \cdot 10^6$ (što vidimo i iz veličine datoteke od 22 MB). Pomnožimo n s 9 i dobijemo ~ 200 MB, što je vrijednost na kojoj se stabilizira memorijsko zauzeće za parametar duljina veći od 16.



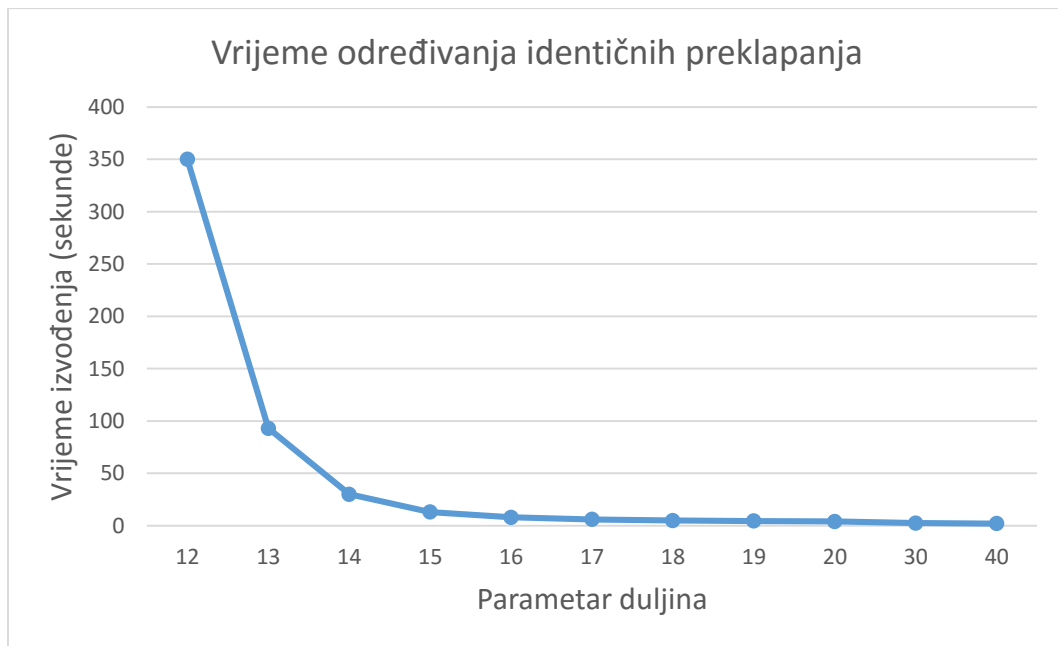
Slika 14. Maksimalno memorijsko zauzeće u MB u ovisnosti o parametru *duljina* (donja granica duljine za tražena identična preklapanja), zauzeće teži u ~ 200 MB, jer su toliko velike strukture podataka korištene za određivanje identičnih preklapanja.

Traženje poravnanja dinamičkim programiranjem koristi matricu cijelih brojeva maksimalnih dimenzija parametar *udaljenost* na kvadrat ($udaljenost^2$). Segmenti se poravnavaju jedan po jedan te se u memoriju sprema samo jedna matrica. Za odabrani parametar 200 matrica sadrži 40000 cijelih brojeva čije je memorijsko zauzeće $40000 \cdot (4 \text{ okteta}) = 0.16$ MB. Čak i za veći parametar npr. 2000 memorijsko zauzeće dinamičkog programiranja će biti 16 MB što je zanemarivo s obzirom na ukupno zauzeće (slika 14.). Iz ovog izračuna i podataka sa slike 14. zaključujemo da većinu memorijskog zauzeća čine pronađena identična preklapanja.



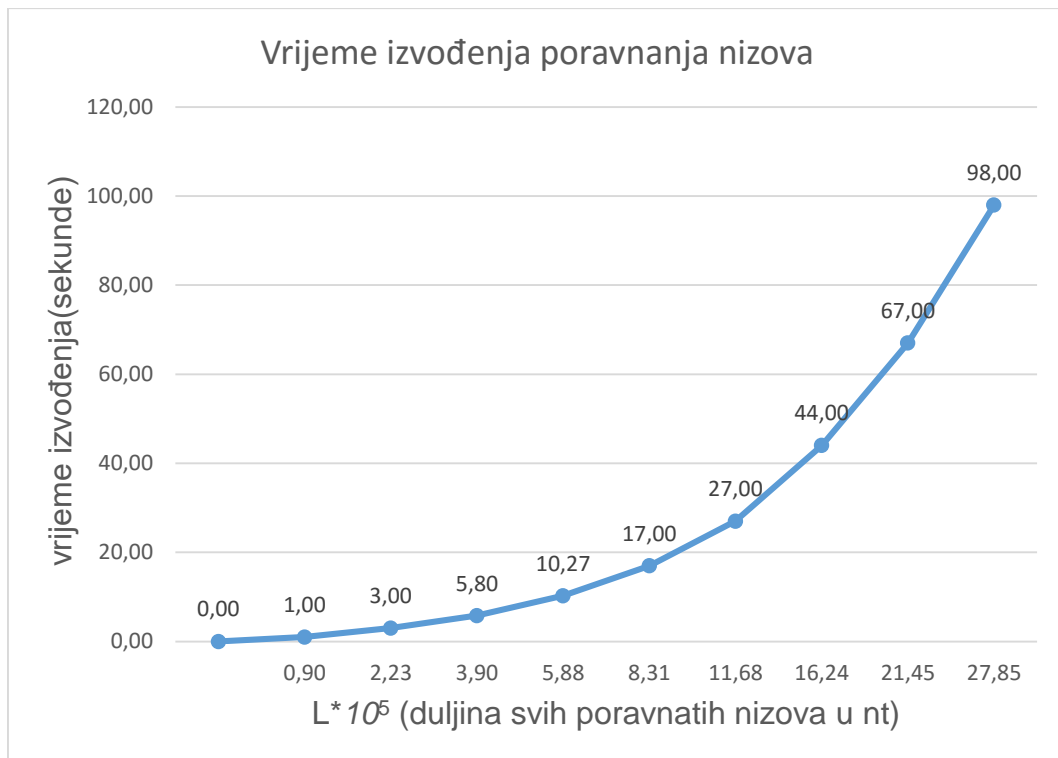
Slika 15. Ovisnost broja identičnih preklapanja (N) i parametra *duljina* (donja granica duljine za tražena identična preklapanja)

Slika 15. prikazuje broj određenih identičnih preklapanja u odnosu na parametar *duljina*. Vidimo kako broj preklapanja pada s porastom parametra *duljina*. Broj preklapanja je jako velik za male vrijednosti parametra *duljina* što stvara i veliku potrošnju memorije, ovime je minimalni mogući parametar *udaljenost* ograničen na dostupne memorijske resurse. U ovom testiranju je računalo za vrijednost parametra *duljina* jednaku 11 ostalo bez memorije.



Slika 16. Vrijeme određivanja identičnih preklapanja u ovisnosti o parametru *duljina* (donja granica duljine za tražena identična preklapanja)

Slika 16. prikazuje vrijeme potrebno za određivanje identičnih preklapanja u ovisnosti o parametru *duljina*. Vidimo kako i vrijeme pada s porastom parametra *duljina*. Iz slika 14., 15. i 16. možemo zaključiti da su broj pronađenih identičnih preklapanja, memorijsko zauzeće te vrijeme potrebno za određivanje identičnih preklapanja u korelaciji s parametrom *duljina* na način da svi padaju s porastom parametra *duljina*.

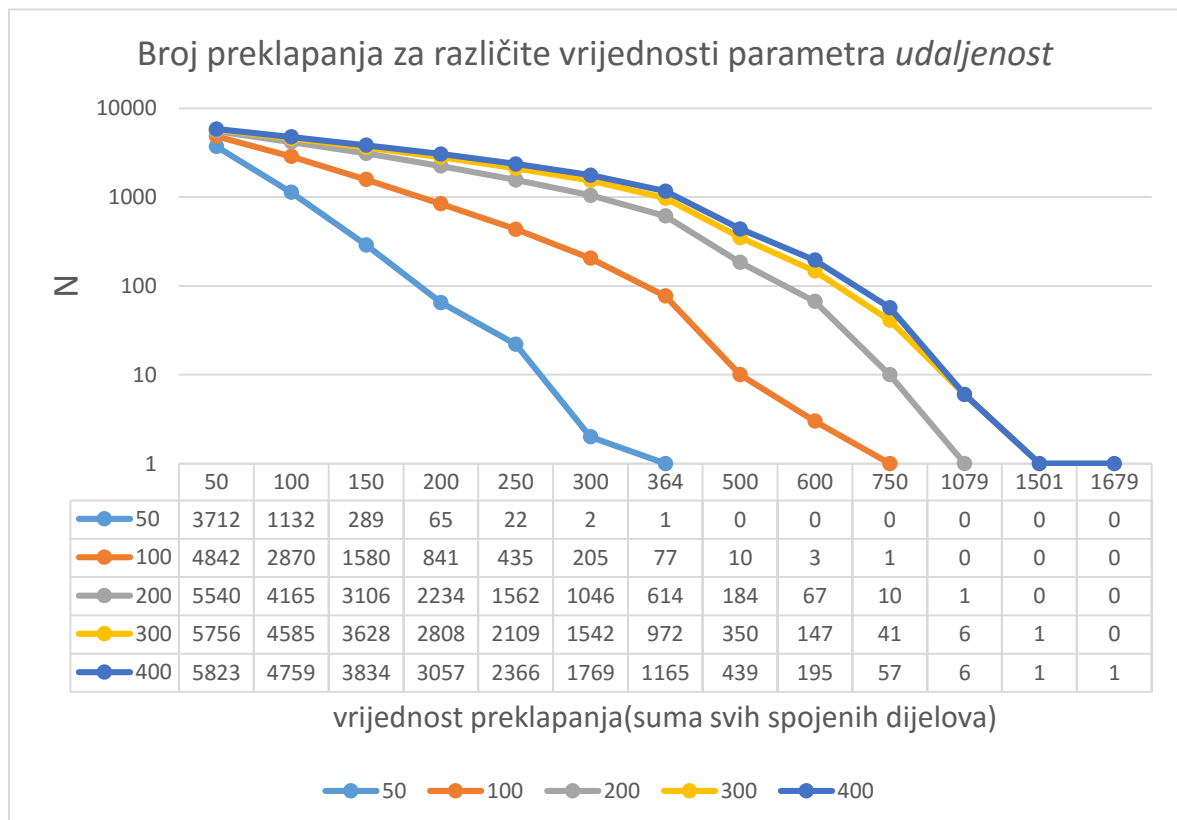


Slika 17. Vrijeme izvođenja poravnanja dinamičkim programiranjem u ovisnosti o ukupnoj duljini poravnatih nizova, rezultat odgovara teoretskoj vremenskoj složenosti od $O(L^2)$

Slika 17. prikazuje trajanje traženja poravnanja korištenjem dinamičkog programiranja u ovisnost na duljinu svih poravnatih dijelova (duljina označena s L). Različite vrijednosti ukupne duljine L su dobivene mijenjanjem parametra udaljenosti kao u prethodnom testu. Vrijednosti L postavljene su na x os, a na osi y se nalazi vrijeme izvođenja u sekundama. Dobivena je kvadratna krivulja u ovisnosti o L , što i odgovara teoretskoj vremenskoj složenosti određivanja poravnanja s dinamičkim programiranjem od $O(L^2)$ (poglavlje 5.1.).

Sljedeća testiranja (slika 18.) provedena su s parametrom *duljina* jednakim 12. Određeno je 17 656 717 identičnih preklapanja za 355 sekundi. Vrijednost pojedinog preklapanja definiramo kao sumu vrijednosti svih preklapanja koja su spojena u to preklapanje. Vrijednost se određuje uspoređivanjem znak po znak po sljedećem načinu bodovanja, slaganje 1 bod, neslaganje -1 bod, i razmak -2 boda. Slika 18. prikazuje broj preklapanja (označeno s N na y osi) koja imaju vrijednost veću od vrijednosti na y osi. Mjerenja su napravljena za 4 parametra *udaljenost* (50, 100,

200, 300, 400) prikazana različitim bojama na grafu. Vrijednosti na y osi su u logaritamskoj skali. Iz slike možemo vidjeti da se povećanjem parametra *udaljenost* povećava i broj pronađenih preklapanja. Brzina porasta broja preklapanja pada s povećanjem parametra udaljenosti te skoro skroz prestaje oko parametra 400 (na grafu se vidi da su krivulje za parametre 300 i 400 skoro jednake).



Slika 18. prikazuje broj preklapanja (označeno s N na y osi) koja imaju vrijednost(suma vrijednosti svih dijelova spojenih u to preklapanje bodovanih s 1 slaganje, -1 neslaganje i -2 razmak) veću od vrijednosti na y osi. Mjerjenja su napravljena za 4 parametra *udaljenost*(50, 100, 200, 300, 400), prikazana različitim bojama. Vrijednosti na y osi su u logaritamskoj skali.

7. Zaključak

Nova tehnologija sekvenciranja nanopore minION generira duga očitavanja uz veliki postotak pogreške. Pri postupku sastavljanja genoma iz dobivenih očitavanja potrebno je uzeti u obzir spomenutu pogrešku. S obzirom da određivanje optimalnih preklapanja s algoritmom dinamičkog programiranja nije ostvarivo u razumnom vremenu moramo napraviti kompromis (*eng. „trade off“*) između vremena izvođenja i kvalitete dobivenih rezultata.

U radu sam demonstrirao kombinirani pristup određivanja preklapanja koje prvo određuje identična preklapanja te ih potom spaja dinamičkim programiranjem. Uz rad sam napravio i pokaznu implementaciju potonjeg pristupa. Implementacija omogućava da korisnik s obzirom na dostupne računalne resurse sam odabere parametre pretraživanja. U rezultatima testiranja razvijene programske potpore sam pokazao kako je moguće mijenjati kvalitetu očitavanja s obzirom na vrijeme izvođenja i utrošak memorije.

Provedenim testiranjima uočio sam problem velike memorijske potrošnje kod pronalaska velikog broja identičnih preklapanja. Problem je moguće riješiti korištenjem memorije s tvrdog diska. Rješavanje potonjeg problema i provođenje dodatnih poboljšanja (npr. paralelizacija dinamičkog programiranja) ostavljam za rad u budućnosti.

8. Literatura

- Abouelhoda, M. I., Kurtz, S., & Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1), 53–86.
[http://doi.org/10.1016/S1570-8667\(03\)00065-0](http://doi.org/10.1016/S1570-8667(03)00065-0)
- de Bruijn, N. G. (1946). A Combinatorial Problem.
- De novo transcriptome assembly. (2016). U *Wikipedia, the free encyclopedia*. Preuzeto od https://en.wikipedia.org/w/index.php?title=De_novo_transcriptome_assembly&oldid=722699320
- Dynamic programming. (2016). U *Wikipedia, the free encyclopedia*. Preuzeto od https://en.wikipedia.org/w/index.php?title=Dynamic_programming&oldid=723853872
- FASTA format. (2016). Preuzeto 31. maj 2016., od <http://zhanglab.ccmb.med.umich.edu/FASTA/>
- FLOPS. (2016). U *Wikipedia, the free encyclopedia*. Preuzeto od <https://en.wikipedia.org/w/index.php?title=FLOPS&oldid=723319802>
- Franjić, M., Mijačika, A., Sabolić, M., & Široki, T. (2016). Izgradnja Fm-indeksa.
- Gusfield, D. (1997). Linear-time construction of suffix trees. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Preuzeto od <http://webdiis.unizar.es/asignaturas/APD/suffixtrees2.pdf>
- Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2), 147–160. <http://doi.org/10.1002/j.1538-7305.1950.tb00463.x>

- HiSeq 2500 Ultra-High-Throughput Sequencing System. (2016). Preuzeto 08. jun 2016., od http://www.illumina.com/systems/hiseq_2500_1500.html
- Illumina | Sequencing and array-based solutions for genetic research. (2016). Preuzeto 08. jun 2016., od <http://www.illumina.com/>
- Kasai, T., Lee, G., Arimura, H., Arikawa, S., & Park, K. (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. U *Combinatorial pattern matching* (str. 181–192). Springer. Preuzeto od http://link.springer.com/chapter/10.1007/3-540-48194-X_17
- Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals.
- Manber, U., & Myers, G. (1993). Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5), 935–948. <http://doi.org/10.1137/0222058>
- Myers, E. W., Sutton, G. G., Delcher, A. L., Dew, I. M., Fasulo, D. P., Flanigan, M. J., ... Venter, J. C. (2000). A whole-genome assembly of *Drosophila*. *Science (New York, N.Y.)*, 287(5461), 2196–2204.
- Needleman, S. B., & Wunsch, C. D. (1970). A General Method Applicable to Search for Similarities in Amino Acid Sequence of 2 Proteins. *Journal of molecular biology*, 48(3), 443–53. [http://doi.org/10.1016/0022-2836\(70\)90057-4](http://doi.org/10.1016/0022-2836(70)90057-4)
- Nong, G., Zhang, S., & Chan, W. H. (2009). Linear Suffix Array Construction by Almost Pure Induced-Sorting. U *Data Compression Conference* (Tom 0, str. 193–202). Los Alamitos, CA, USA: IEEE Computer Society. <http://doi.org/10.1109/DCC.2009.42>

- Oxford Nanopore Technologies. (2016). Preuzeto 29. maj 2016., od <https://www.nanoporetech.com/>
- Quick, J., Quinlan, A. R., & Loman, N. J. (2014). A reference bacterial genome dataset generated on the MinION™ portable single-molecule nanopore sequencer. *GigaScience*, 3, 22. <http://doi.org/10.1186/2047-217X-3-22>
- Sellers, P. (1974). On the Theory and Computation of Evolutionary Distances. *SIAM Journal on Applied Mathematics*, 26(4), 787–793. <http://doi.org/10.1137/0126070>
- Sequence assembly. (2016). U *Wikipedia, the free encyclopedia*. Preuzeto od https://en.wikipedia.org/w/index.php?title=Sequence_assembly&oldid=724059208
- Shotgun sequencing. (2016). U *Wikipedia, the free encyclopedia*. Preuzeto od https://en.wikipedia.org/w/index.php?title=Shotgun_sequencing&oldid=716753514
- Simpson, J. T., & Durbin, R. (2012). Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3), 549–556. <http://doi.org/10.1101/gr.126953.111>
- Stack (abstract data type). (2016). U *Wikipedia, the free encyclopedia*. Preuzeto od [https://en.wikipedia.org/w/index.php?title=Stack_\(abstract_data_type\)&oldid=724015745](https://en.wikipedia.org/w/index.php?title=Stack_(abstract_data_type)&oldid=724015745)
- Šikić, M., & Domazet-Lošo, M. (2013). Bioinformatika.
- Tree traversal. (2016). U *Wikipedia, the free encyclopedia*. Preuzeto od https://en.wikipedia.org/w/index.php?title=Tree_traversal&oldid=724144215
- Ukkonen, E. (1995). *On-Line Construction of Suffix Trees*.

Naslov, sažetak i ključne riječi

Završni rad:

Određivanje preklapanja među očitanjima u postupku de novo sastavljanja genoma

Sažetak

Određivanje preklapanja je prvi korak *de novo* sastavljanja genoma metodom preklapanje-razmještaj-konsenzus . Za određivanje preklapanja koristio sam kombinirani pristup. Pristup koristi prošireno sufiksno polje za pronalazak identičnih preklapanja i dinamičko programiranje za poravnanje dijelova očitavanja između identičnih preklapanja. Ovakav način određivanja preklapanja omogućava kompromis između kvalitete određenih preklapanja i vremena izvođenja.

Ključne riječi: određivanje preklapanja, metoda preklapanje-razmještaj-konsenzus, sastavljanje genoma, prošireno sufiksno polje, dinamičko programiranje

Bachelor thesis:

Finding overlapping reads for de novo genome assembly

Abstract

Finding overlaps is the first step of *de novo genome* assembly using Overlap-Layout-Consensus method. To determine overlaps I used a combined approach. The approach uses enhanced suffix array for finding identical overlaps and dynamic programming to align parts of reads between identical overlaps. This method allows a trade off between the quality of determined overlaps and runtime.

Keywords: finding overlaps, overlap-layout.consensus method, genome assembly, enhanced suffix array, dynamic programming

Dodatak A: Upute za korištenje programske potpore

Uputa za instalaciju:

Za izgradnju izvršne datoteke pozicionirati se u direktorij

Izvorni_kod i izvršiti naredbu make.

Izvršna datoteka će biti spremljena u direktorij Izvrsni_kod.

Uputa za korištenje:

Programu je kao argument naredbenog retka potrebno predati putanju do datoteke koja sadrži očitavanja u fasta formatu.

Pronađena preklapanja će biti spremljena u datoteku out.

U direktoriju test_data se nalaze neki primjeri ulaznih podataka.