UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Filip Miljaković

ALGORITMI ZA EGZAKTNO I PRIBLIŽNO UPARIVANJE ŠABLONA - ELEKTRONSKA LEKCIJA

master rad

Beograd, 2022.

Mentor:

dr Jovana Kovačević, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Mirjana Maljković, docent
Univerzitet u Beogradu, Matematički fakultet

Aleksandar Veljković, asistent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: septembar 2022.

Najmilijima.

Naslov master rada: Algoritmi za egzaktno i približno uparivanje šablona - elektronska lekcija

Rezime: U ovom radu su prikazani algoritmi koji se koriste prilikom uparivanja šablona, od onih koji zahtevaju najviše resursa do onih koji problem rešavaju sa mnogo manjom količinom memorije i vremena. Svi odabrani algoritmi su prikazani uz dodatak primera. Kako je zamišljeno da aplikacija sa grafičkim korisničkim interfejsom koja prati ovaj rad bude korišćena kao elektronska lekcija, korisnička aplikacija je opisana kroz uputstvo za upotrebu sa prikazom svih dostupnih slučajeva upotreba.

Ključne reči: bioinformatika, referentni ljudski genom, Barouz-Viler, BWT, sufiksna stabla, prefiksna stabla, sufiksni niz, *First-Last* svojstvo

Sadržaj

Glava 1

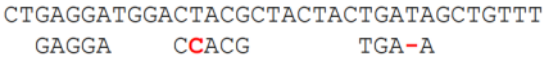
Uvod

Uparivanje šablona se veoma često javlja kao problem u različitim oblastima naučnog istraživanja: biologiji, informatici, samim tim i bioinformatici, medicini itd. Postoje različiti pristupi rešavanju ovog problema. Zbog svoje kompleksnosti i velike količine resursa koje određeni problemi zahtevaju, veoma je važno da algoritmi koji se koriste za uparivanje šablona budu efikasni. Kao jedan od najpoznatijih problema u medicini i bioinformatici koji zahteva uparivanje šablona javlja se problem lociranja mutacija u ljudskom genomu i ranog otkrivanja raznih genetskih poremećaja.

Oko 1% beba se rodi sa nekom vrstom mentalnog poremećaja. Jedan od ovih poremećaja je i Ohdo sindrom, koje kao posledicu ima maskoliki izraz lica. Genetičari su 2011. godine otkrili nekoliko mutacija koje je delilo više pacijenata sa ovim sindromom. Na osnovu tih saznanja otkrivena je mutacija koja skraćuje određeni protein i koja je zapravo odgovorna za nastajanje Ohdo sindroma. Nalaženje uzroka Ohdo sindroma predstavlja samo jedno od mnogo sličnih otkrića kojima su detektovane mutacije odgovorne za različite poremećaje.

Mutacije unutar ćelija jedne osobe se nalaze tako što se njeni segmenti DNK porede sa referentnim ljudskim genomom (slika 1.1). Očitavanje predstavlja sekvencu parova baza koja odgovara nekom delu DNK, dok je referentni ljudski genom sastavljen od genoma više donora i predstavlja šablon u kojem se pomenuti segmenti individualnih ljudskih genoma traže. Ako je sekvenciranje genoma kao sastavljanje slagalice od delova genoma dobijenih iz sekvencera (ove delove zovemo očitavanjima), onda je referentni ljudski genom kao slika gotove slagalice na kutiji i pomaže nam da sastavimo delove. Referentni ljudski genom nije savršen, tokom godina je evoluirao, određene greške su ispravljene i nedostaci popravljeni. Ipak, i dalje, oko 0,1% individualnog ljudskog genoma ne može biti upareno sa referentnim ljudskim

genomom, a delovi koji mogu u proseku imaju ukupno oko 3 miliona mutacija, kao razliku između individualnog i referentnog ljudskog genoma. Ukupan broj nukleotida u ljudskom genomu je oko 3 milijarde. Upravo te mutacije su ono što zavređuje pažnju.



Slika 1.1: Primer mapiranja očitavanja. Gornja niska predstavlja referentni genom, a donje niske očitavanja segmenata individualnih genoma [1]

Razmotrimo ključne izazove iz informatičkog ugla koji se javljaju prilikom rešavanja ovog problema. Dužina ljudskog genoma smeštenog u memoriji je preko 3GB, dok ukupna dužina svih očitavanja može biti veća od 1TB. Zbog toga nam je od izuzetne važnosti da algoritmi kojima radimo mapiranje očitavanja budu efikasni. U sledećim poglavljima krenućemo od iterativnog algoritma koji koristi „grubu silu" i ima gotovo kvadratnu složenost (zahteva najviše resursa - memorije i vremena), pa ćemo nastaviti sa sve efikasnijim algoritmima dok ne dođemo do onog zadovoljavajućeg koji nam omogućava da mapiranje očitavanja uradimo u linearnom vremenu sa malom konstantom.

Glava 2

Algoritmi za egzaktno i približno uparivanje šablona

Kao što je već navedeno u uvodu, postoje mnogi algoritmi različite složenosti koji se bave uparivanjem šablona. U ovom odeljku ćemo prikazati:

- Algoritam uparivanja šablona grubom silom - iterativni algoritam
- Algoritam uparivanja šablona pomoću prefiksnih stabala
- Algoritam uparivanja šablona pomoću sufiksnih stabala
- Algoritam uparivanja šablona pomoću sufiksnog niza
- Algoritam uparivanja šablona pomoću Barouz-Vilerove transformacije.

Biće prikazano još nekoliko algoritama koje zbog svog značaja vredi pomenuti, a to su:

- Algoritam *Aho–Corasick*
- *Ukkonen*-ov algoritam.

Da ne bismo koristili delove genoma kao primere u ovom radu, radi boljeg pregleda i lakšeg razumevanja koristićemo nisku `homomorfizam` kao primer genoma (sačuvana u promenljivoj *Genom* u nastavku teksta).

Možemo razlikovati jednostruko uparivanje šablona (problem 1) i višestruko uparivanje šablona (problem 2) [2].

PROBLEM 1 (PROBLEM JEDNOSTRUKOG UPARIVANJA ŠABLONA)

ulaz: *Niske* ***Patern*** *i* ***Genom****.*

izlaz: *Sve pozicije u niski* ***Genom*** *gde se niska* ***Patern*** *pojavljuje kao podniska.*

PROBLEM 2 (PROBLEM VIŠESTRUKOG UPARIVANJA ŠABLONA)

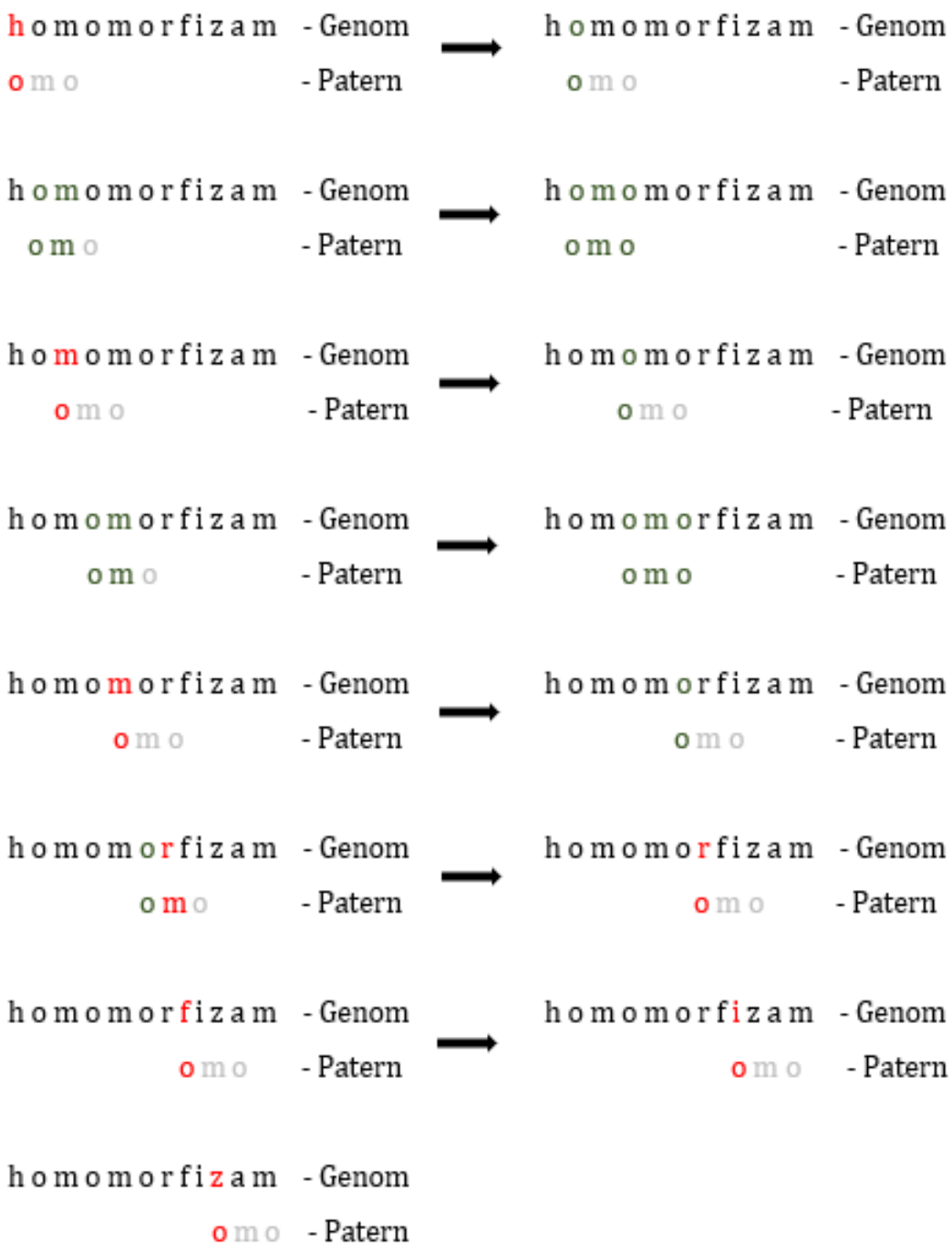
ulaz: *Kolekcija niski* ***PaternLista*** *i niska* ***Genom****.*

izlaz: *Sve pozicije u niski* ***Genom*** *gde se niske iz kolekcije* ***PaternLista*** *pojavljuju kao podniske.*

2.1 Iterativni algoritam za uparivanje šablona

Pristup kojim ćemo prvo pokušati da rešimo problem uparivanja šablona je iterativni pristup gde ćemo linearno da prođemo kroz genom i proverimo da li se dati šablon poklapa sa podniskom genoma iste dužine, a koja počinje na određenoj poziciji. Objasnimo ovaj algoritam na primeru niske `homomorfizam`, dok ćemo za šablon koji tražimo da uzmemo nisku `omo`.

Kao što možemo videti u primeru na slici 2.1, poređenje započinjemo od prvog karaktera genoma, karaktera `h` i poredimo ga sa prvim karakterom šablona, u ovom slučaju karaktera `o`. Pošto vidimo da se ne poklapaju, šablon pomeramo za jedno mesto dalje i poredimo sada drugi karakter genoma (karakter `o`) sa prvim karakterom šablona. U ovom slučaju vidimo da imamo poklapanje i sada ne pomeramo šablon, već poredimo treći karakter genoma sada sa drugim karakterom šablona, karakterom `m`. Vidimo da se i oni poklapaju i nastavljamo dalje. Poredimo četvrti karakter genoma sa trećim karakterom šablona (karakter `o`), vidimo da se i oni poklapaju. Pošto vidimo da smo došli do kraja šablona, zaključujemo da smo pronašli poklapanje. Vraćamo se na početak šablona a poziciju u genomu sa kojom poredimo pomeramo za jedan. Sada poredimo karakter `o` sa karakterom na poziciji 2 (indeksiranje počinjemo od 0), karakterom `m`. Proces nastavljamo sve dok ne stignemo do $|Genom| - |Patern|$ indeksa niske genom.



Slika 2.1: Primer korišćenja iterativnog algoritma na primeru genoma `homomorfizam` i šablona `omo`

U nastavku je prikazan pseudokod algoritma jednostrukog uparivanja šablona.

Algoritam 1 Iterativni algoritam

```
function IterativniAlgoritam(Genom, Patern)
pronađeniIndeksi ← prazan niz
for i = 0 to |Genom| − |Patern| do
    for j = 0 to |Patern| do
        if Genom[i + j] ≠ Patern[j] then
            break
        end if
        if j == |Patern| − 1 then
            dodaj i kao pronađeni indeks u pronađeniIndeksi
        end if
    end for
end for
return pronađeniIndeksi
```

U slučaju višestrukog uparivanja bi kao ulaz umesto jedne niske *Patern* koju pretražujemo imali listu šablona - *PaternLista*. Tada bismo za svaki šablon iz te liste izvršili goreprikazani algoritam.

Složenost jednostrukog uparivanja kod iterativnog algoritma je O($|Genom| \cdot |Patern|$), a kod višestrukog O($|Genom| \cdot |PaternLista|$). $|PaternLista|$ predstavlja sumu dužina svih elemenata liste šablona koja se koristi kao ulaz algoritma. S obzirom da ljudski genom zauzima dosta memorije, te ovaj algoritam može biti vremenski veoma zahtevan, potražićemo efikasnija rešenja.

2.2 Uparivanje šablona pomoću prefiksnih stabala

Kako je gorenavedeni iterativni algoritam veoma zahtevan, potrebno je da nađemo način kako da ceo proces učinimo efikasnijim i smanjimo složenost. Možemo uvideti da u prethodnom algoritmu za višestruko uparivanje prolazimo kroz genom za svaki šablon nezavisno. Način na koji možemo optimizovati prethodno rešenje je da sve šablone smestimo u usmeren aciklični graf koji predstavlja prefiksno stablo (*Trie* u nastavku teksta) i koji ima sledeća svojstva:

- *Trie* ima jedan početni čvor sa ulaznim stepenom 0 koji nazivamo korenim čvorom.
- Svaka grana je označena jednim karakterom alfabeta.

- Sve grane koje izlaze iz istog čvora imaju različite oznake.
- Svaki šablon iz niza šablona koji se traže može da se kreira spajanjem karaktera duž neke putanje od korenog čvora ka listovima (čvorova sa izlaznim stepenom 0).
- Svaka putanja od korenog čvora do lista svojim oznakama može rekonstruisati neku nisku iz liste šablona koji se traže.

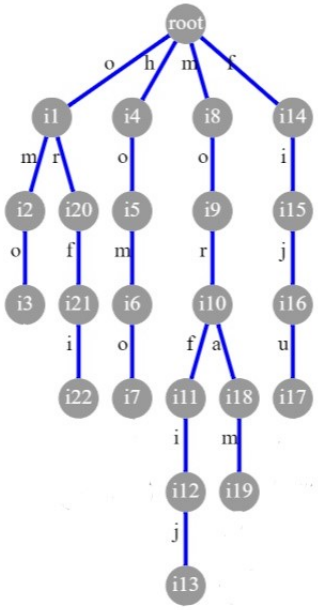
U nastavku teksta možemo videti pseudokod koji objašnjava način kreiranja stabla *Trie*.

Algoritam 2 Konstrukcija stabla *Trie*

```
function KonstrukcijaStabla(PaternLista)
Trie ← stablo od jednog čvora koji zovemo koreniČvor
for svaki Patern iz PaternLista do
    trenutniČvor ← koreniČvor
    for i = 0 to |Patern| do
        trenutniSimbol ← i-ti simbol niske Patern
        if postoji grana iz trenutniČvor sa oznakom trenutniSimbol then
            trenutniČvor ← čvor u kojem se ta grana završava
        else
            dodaj noviČvor u Trie
            nova grana trenutniČvor → noviČvor sa oznakom trenutniSimbol
            trenutniČvor ← noviČvor
        end if
    end for
end for
return Trie
```

Najočigledniji način za njegovo kreiranje je iterativno dodavanje svake niske iz liste šablona koji se traže u stablo idući od korenog čvora. Svaka grana stabla predstavlja karakter šablona. Pomoću ovog stabla lako možemo utvrditi da li je neka niska iz liste šablona prefiks genoma. Primer stabla *Trie* možemo videti na slici 2.2.

Sada kada imamo stablo *Trie* kreirano od liste šablona, objasnićemo kako da pronađemo mesta poklapanja šablona u genomu. To možemo uraditi tako što krenemo sa čitanjem karaktera u genomu i vidimo da li u stablu postoji putanja od korena do lista. Ukoliko smo stigli do lista znamo da je taj šablon prefiks niske genom. Pseudokod objašnjenog algoritma se može videti u nastavku teksta.



Slika 2.2: Primer stabla *Trie* za listu šablona od niski `omo`, `homo`, `morfij`, `fiju`, `moram`, `orfi`

Algoritam 3 Prefiksno stablo - nalaženje prefiksa

```
function NalaženjePrefiksa(Genom, Trie)
trenutniKarakter ← prvi karakter niske Genom
trenutniČvor ← koreni čvor stabla Trie
while 1 do
    if trenutniČvor je list stabla then
        return šablon koji smo našli
    else if postoji grana iz trenutniČvor označena sa trenutniKarakter then
        trenutniKarakter ← sledeći karakter u genomu
        trenutniČvor ← čvor u koji ulazi pomenuta grana
    else
        Ispis da nije pronađeno poklapanje
        return
    end if
end while
```

Da bismo zaključili da li se neka niska iz liste šablona poklapa sa bilo kojom podniskom genoma na nekoj poziciji i, moramo da odsečemo prvih $i-1$ karaktera genoma i na takvu nisku primenimo upravo objašnjeni algoritam za nalaženje prefiksa. Da bismo to postigli, potrebno je pozvati pomenuti algoritam $|Genom|$ puta i u svakoj iteraciji odstraniti prvi karakter tako kreirane podniske genoma.

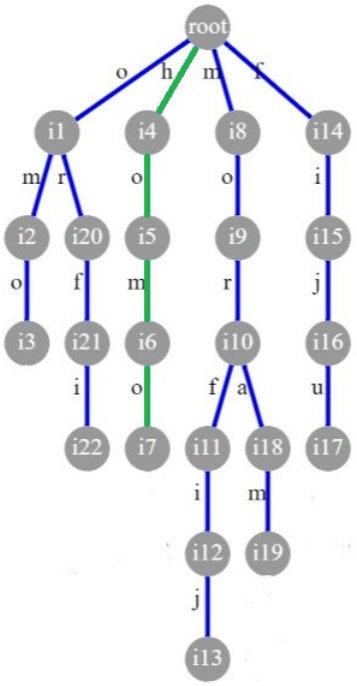
Algoritam 4 Prefiksno stablo - nalaženje poklapanja šablona u genomu

function NalaženjePoklapanjaUGenomu(Genom, Trie)
listaRešenja ← prazna lista
for $i = 0$ to $|Genom|$ **do**
  pronađeniŠablon ← NalaženjePrefiksa(podniska *Genom*-a od i-te pozicije do kraja, Trie)
  if *pronađeniŠablon* nije prazan **then**
    dodaj par (i, *pronađeniŠablon*) u *listaRešenja*
  end if
end for
return *listaRešenja*

Primenimo sada pomenuti algoritam *NalaženjePoklapanjaUGenomu(Genom, Trie)* na primeru genoma `homomorfizam`. Za primer strukture *Trie* možemo uzeti stablo sa slike 2.2. Krenućemo od kompletne niske genoma - `homomorfizam`. Uzimamo prvi karakter `h` i u stablu *Trie* krećući od korena, tražimo njegovo pojavljivanje. Vidimo da od korenog čvora do čvora `i4` postoji grana koja kao oznaku ima karakter `h`. Nastavljamo pretragu dalje od čvora `i4`. Uzimamo sledeći karakter niske `homomorfizam`, karakter `o` i primećujemo da postoji grana sa oznakom `o` koja vodi do čvora `i5`. Nastavljamo dalje sa karakterima `m` i `o` na isti način. Uviđamo da smo došli do lista što znači da je poklapanje uspešno, tj. da niska `homo` iz liste šablona jeste prefiks niske `homomorfizam`. U listu pronađenih indeksa možemo da dodamo indeks 0 jer smo poklapanje našli na početku genoma `homomorfizam`. Objašnjeni slučaj možemo videti na slici 2.3. Dalje nastavljamo tako što za novu nisku genoma uzimamo podnisku niske `homomorfizam` nastalu tako što odbacimo prvi karakter, dakle `omomorfizam`. Vraćamo početak izvršavanja na koreni čvor i postupak nastavljamo dalje na isti način sve dok ne dođemo do kraja genoma. Slučaj nepronalaženja šablona u stablu možemo videti na slici 2.4.

Potrebno nam je $|PaternLista|$ koraka da kreiramo stablo koje će sadržati najviše $|PaternLista|$ čvorova. Svaka iteracija algoritma *NalaženjePrefiksa(Genom, Trie)* može imati najviše onoliko koraka kolika je dužina najdužeg šablona iz liste onih koje tražimo. *NalaženjePoklapanjaUGenomu(Genom, Trie)* algoritam napravi $|Genom|$ poziva algoritma *NalaženjePrefiksa(Genom, Trie)*, što za ukupan broj koraka onda daje $|PaternLista| + |Genom| \cdot |NajdužiPatern|$. Ovo je veliko ubrzanje u odnosu na $|Genom| \cdot |PaternLista|$ koraka koliko je zahtevao iterativni algoritam.

Iako je ovaj algoritam brz, kod iterativnog algoritma samo smo genom čuvali u memoriji, dok ovaj algoritam zahteva dosta memorije za čuvanje stabla *Trie*. To

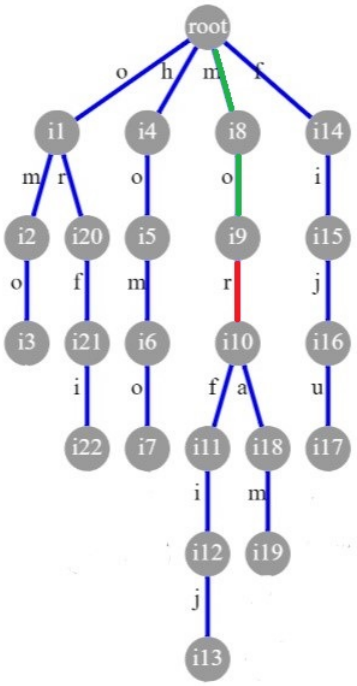


Slika 2.3: Primer poklapanja niske `omo` kao prefiksa jedne od podniski niske `homomorfizam`

je proporcionalno $|PaternLista|$ elemenata, a ako se vratimo na uvod videćemo da ukupna dužina svih očitavanja može biti veća od 1TB, što nije malo.

Možemo primetiti da jedna situacija nije pokrivena ovim algoritmom. To je slučaj ako je neki šablon iz liste šablona prefiks drugog šablona iz liste. U tom slučaju, sa trenutnim rešenjem, samo kraći šablon bi bio uparen, dok bi duži bili preskočeni (na toj lokaciji). Primer slučaja gde bi se greška desila u inicijalnom algoritmu bi bio za genom `homomorfizam` i šablone `omo` i `omomo`. U ovom slučaju, kada dođemo do karaktera `o` na poziciji 2 inicijalnim algoritmom ćemo dobiti poklapanja niske `omo`, ali tada ćemo odbiti prvi karakter genoma i nastaviti sa sledećim sufiksom. Ovde vidimo da će šablon niska `omomo` biti potpuno zanemarena i to poklapanje nećemo naći na ovom mestu, a ono definitivno postoji.

Prethodni problem možemo rešiti tako što ćemo šablone završiti karakterom `$`, koji se ne nalazi u azbuci šablona, pa tako kreirati stablo. U trenutku pretrage ćemo onda kada dođemo do karaktera `$` znati da smo na kraju jednog šablona i dotadašnju nisku staviti u listu rešenja. Ipak, ukoliko pored grane sa oznakom `$`, iz trenutnog čvora postoji još neka grana i ona odgovara sledećem karakteru u genomu (tj. možemo da nastavimo kretanje niz stablo), ići ćemo dalje, sve dok takve grane



Slika 2.4: Primer nepoklapanja niske `morfij` i `moram` kao prefiksa podniske `momorfizam` niske homomorfizam

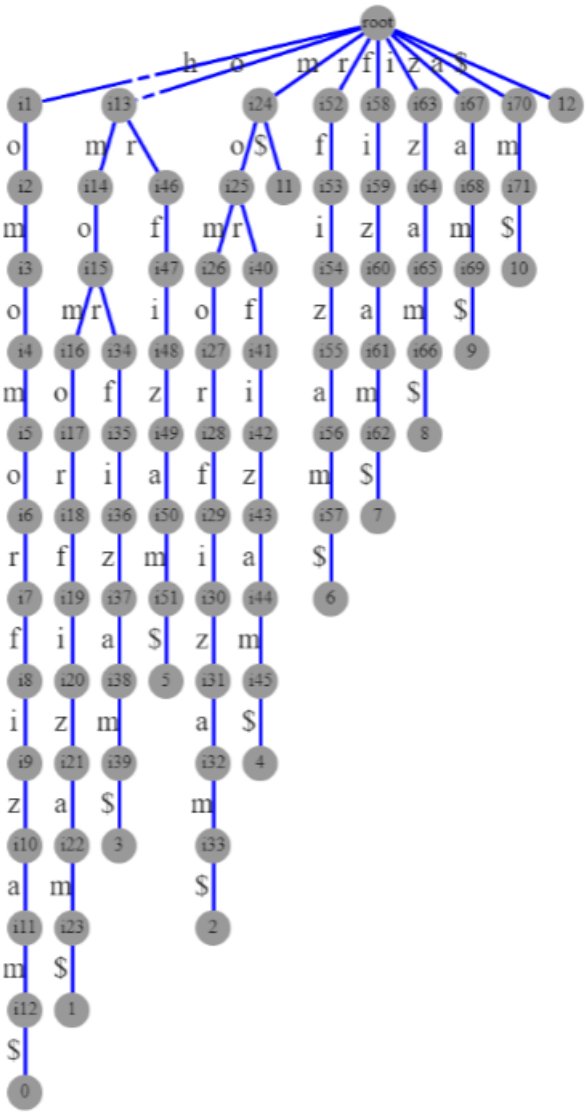
ne bude bilo, ili dok jedina grana iz trenutnog čvora bude grana sa oznakom `$`. Tada odbijamo prvi karakter genoma i nastavljamo dalje kao u inicijalnom algoritmu *NalaženjePoklapanjaUGenomu(Genom, Trie)*.

2.3 Uparivanje šablona pomoću sufiksnih stabala

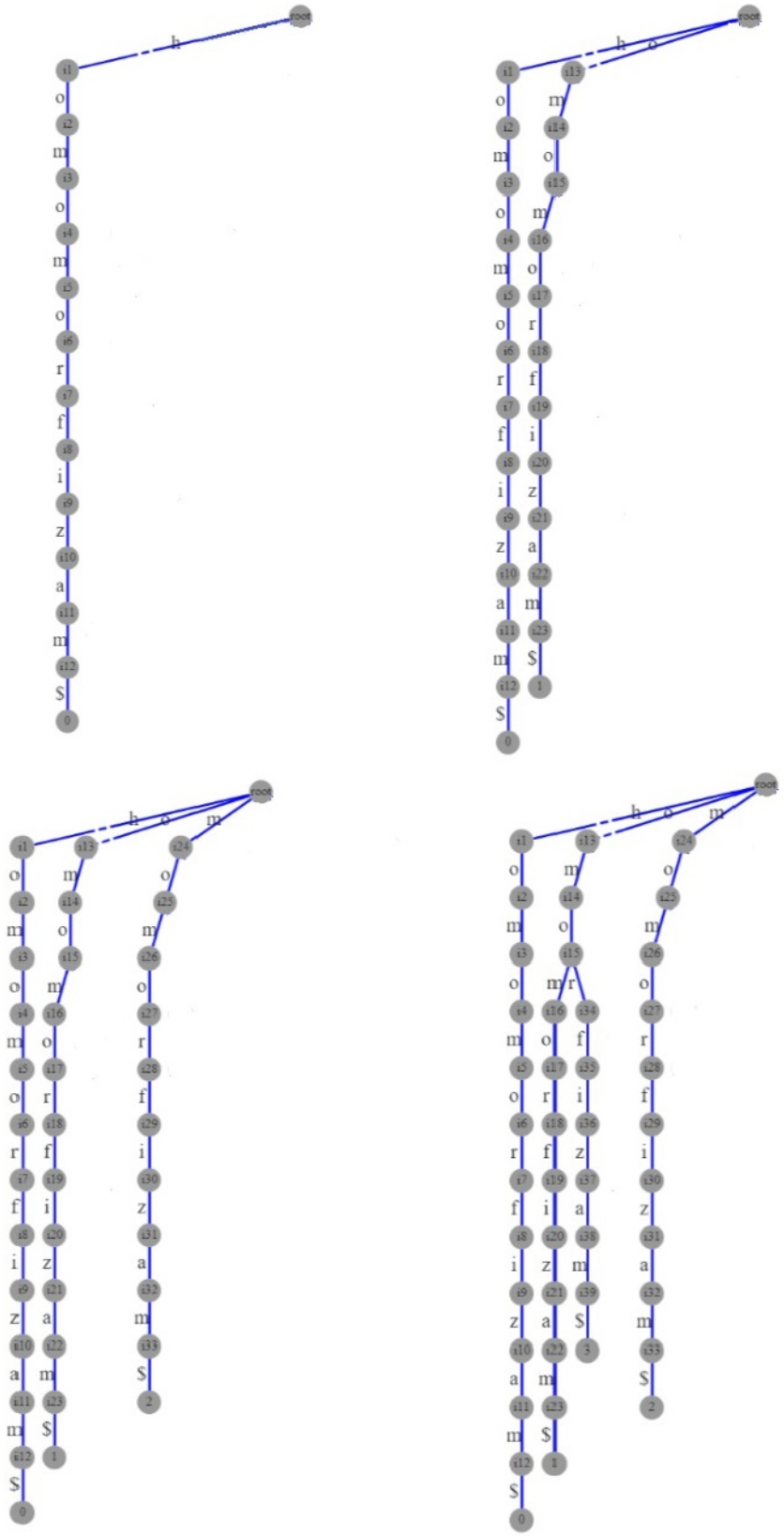
U prethodnom algoritmu čuvali smo celo stablo *Trie(PaternLista)* što može zahtevati mnogo memorije. Sa ciljem umanjenja potrebnih memorijskih resursa, pokušajmo da, umesto od liste šablona, stablo *Trie* sastavimo od svih sufiksa niske *Genom*. Prvo ćemo nadovezati specijalni karakter koji se ne nalazi u sekvenci *Genom* (`$`) na nisku *Genom* kako bismo označili njen kraj. Zatim ćemo svaki list stabla koje ćemo da kreiramo od takve liste sufiksa obeležiti sa indeksom u genomu gde taj sufiks počinje. Na taj način ćemo kada stignemo do lista automatski dobiti indeks koji možemo da vratimo kao mesto poklapanja.

Na slici 2.5 možemo videti sufiksno stablo za nisku genoma `homomorfizam`. Postupno kreiranje stabla se radi prolaženjem kroz sufikse niske *Genom*. Prva četiri

koraka su prikazana na slici 2.6.

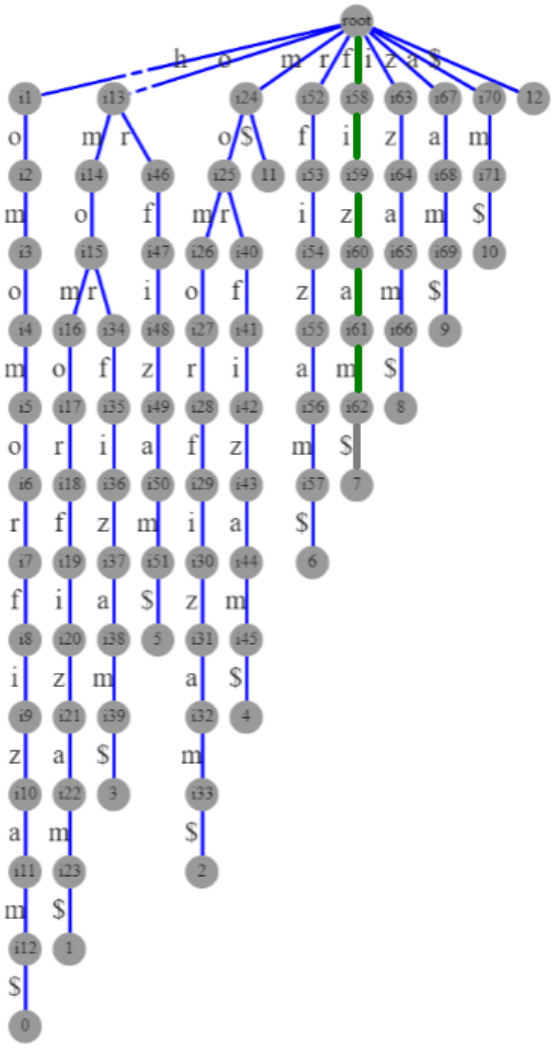


Slika 2.5: Sufiksno stablo za primer genoma `homomorfizam`



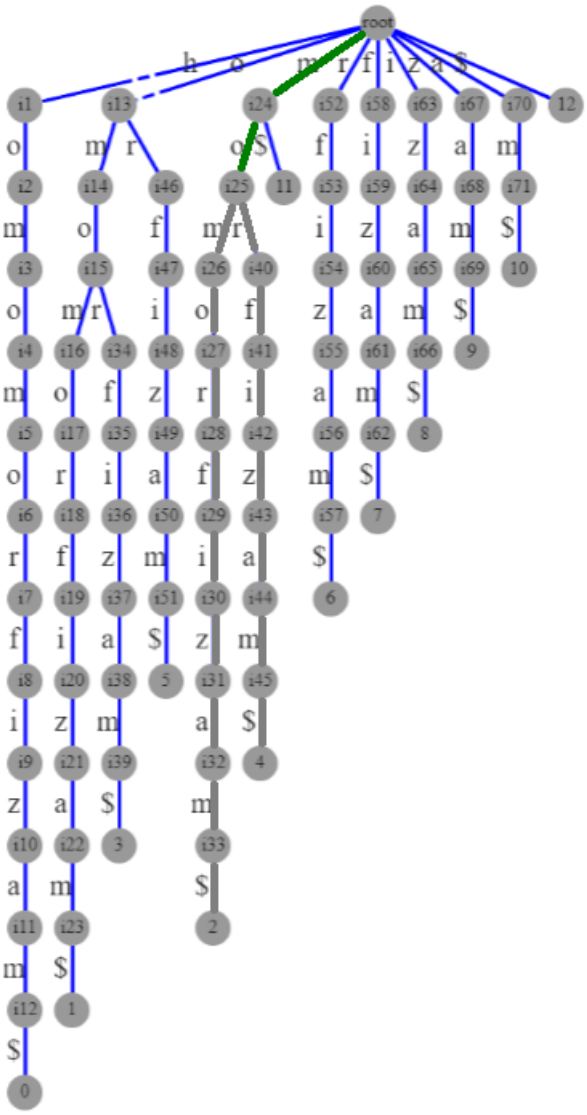
Slika 2.6: Prva 4 koraka kreiranja sufiksnog stabla od genoma `homomorfizam`

Da bismo pronašli da li se određeni šablon nalazi u genomu kao podniska potrebno je da počev od prvog karaktera šablona prođemo kroz stablo *Trie(Genom)* krećući se od korena. Ako možemo da nađemo putanju u stablu pri čemu smo obišli sve karaktere šablona, onda znamo da se on mora pojavljivati u genomu. Ukoliko kretanjem kroz stablo dođemo do lista i poslednji karakter šablona se tu nalazi, odatle možemo zaključiti da je šablon sufiks genoma. U tom slučaju možemo iz lista pročitati indeks na kojem taj sufiks počinje i tu informaciju vratiti kao mesto gde se šablon pojavljuje u genomu (slika 2.7).



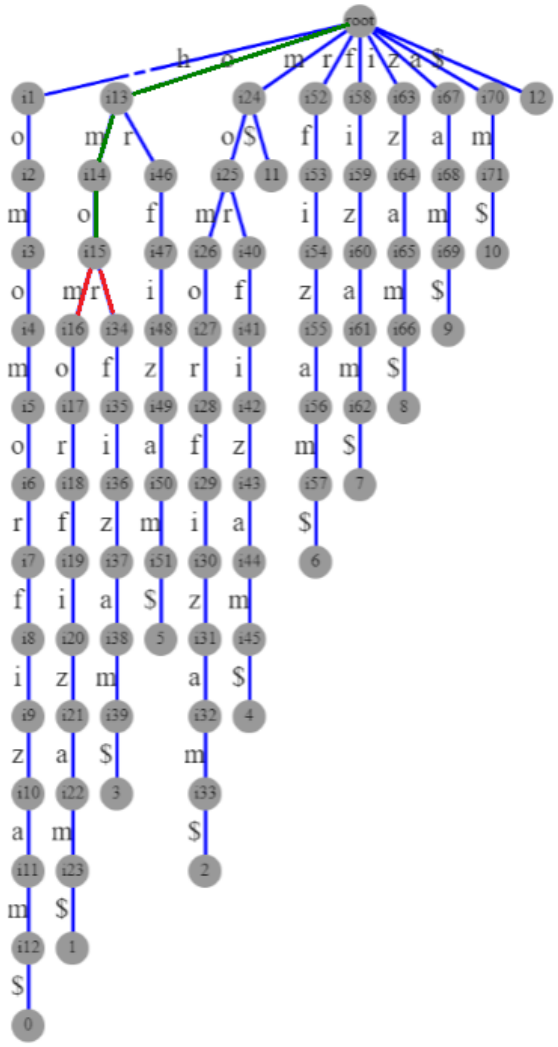
Slika 2.7: Šablon `fizam` je sufiks genoma `homomorfizam`. Imamo jedno poklapanje i ono se nalazi na poziciji 7.

Ako se poslednjim karakterom šablona zaustavimo pre lista u nekom čvoru v, isto imamo poklapanje. U ovom slučaju šablon može, a ne mora, da se pojavljuje više puta u niski genom. Tada, da bismo došli do indeksa gde šablon počinje u genomu, potrebno je da se krećemo svim granama od trenutnog čvora v do listova stabla i dobijemo informaciju o indeksima pozicija pronađenog šablona (slika 2.8).



Slika 2.8: Više pronađenih poklapanja šablona `mo` u niski genoma `homomorfizam`. Poklapanja se nalaze na pozicijama 2 i 4.

Moguć je i slučaj kada nemamo poklapanja. Tada će pretraga stati na nekom čvoru unutar stabla i neće postojati grana iz tog čvora koja će odgovarati narednom karakteru šablona. U tom slučaju zaustavljamo pretragu i zaključujemo da se šablon ne nalazi u genomu kao podniska (slika 2.9).



Slika 2.9: Šablon nije podniska niske genom. Primer genoma `homomorfizam` za primer šablona `omot`

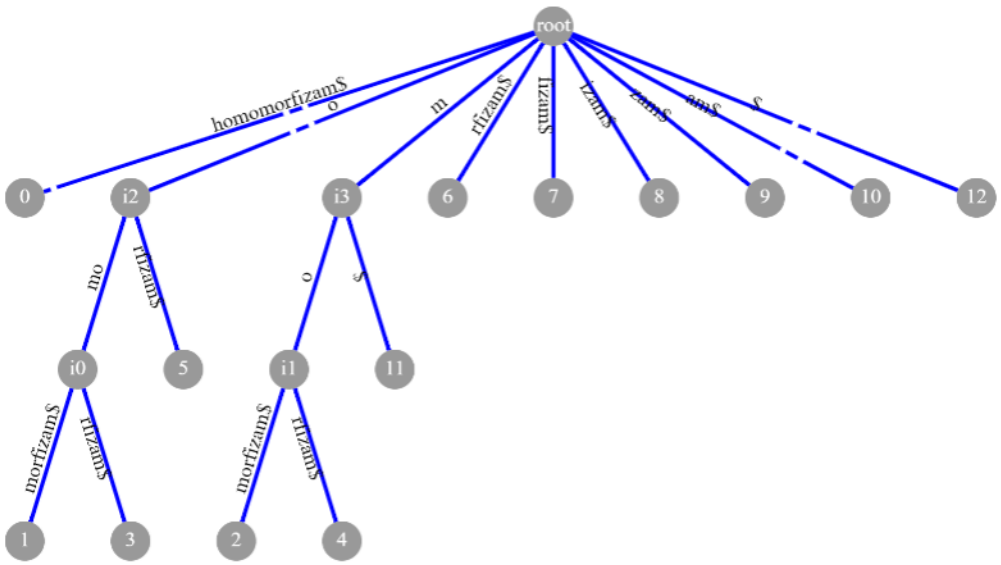
Kod prethodno objašnjenog rešenja stablo konstruišemo od svih sufiksa niske *Genom*, koji su dužine od 1 do $|Genom|$ i imaju ukupnu dužinu $|Genom| \cdot (|Genom| + 1)/2$. Zaključujemo da je složenost približna vrednosti od $O(|Genom|^2)$. Dakle, i

dalje nam je potreban neki način da zahteve za resursima smanjimo.

2.3.1 Kompresovana sufiksna stabla

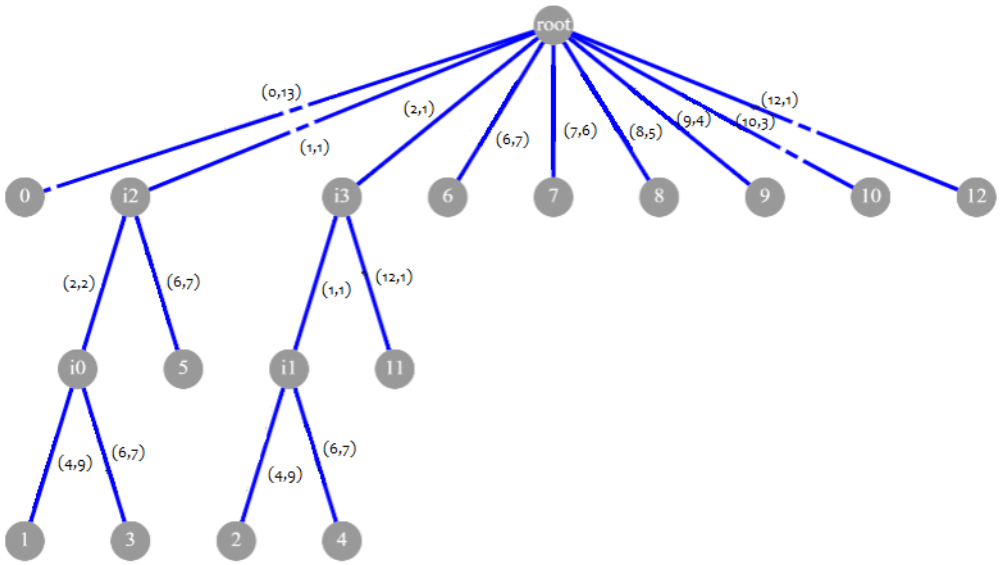
Sufiksna stabla zauzimaju memoriju koja je proporcionalna kvadratu dužine genoma. S obzirom da genomi mogu imati nekoliko milijardi karaktera, to znači da ove strukture traže ogromne prostorne resurse. Razmotrimo da li se oni mogu uštedeti.

Možemo značajno smanjiti broj grana i čvorova u sufiksnom stablu tako što ćemo sve grane u putanjama koje se ne granaju (ulazni i izlazni stepen čvorova je 1) spojiti u jednu granu. Tada će oznake grana biti spojeni karakteri po tim putanjama (slika 2.10).



Slika 2.10: Kompresovano sufiksno stablo za genom `homomorfizam`

Do uštede memorije u ovom slučaju dolazi zato što ne moramo da čuvamo spojene karaktere grana koje se ne granaju, već možemo da čuvamo samo indeks u genomu gde ta niska počinje i njenu dužinu. Npr. za primer sa slike 2.10, za granu `rfizam$` možemo sačuvati indeks 6 i dužinu pomenute niske 7 (slika 2.11).



Slika 2.11: Kompresovano sufiksno stablo za genom `homomorfizam` sa indeksima i dužinama umesto spojenih niski

Iako ovako kompresovano sufiksno stablo značajno smanjuje memorijske zahteve, sa $O(|Genom|^2)$ na $O(|Genom|)$, prosečno nam i dalje treba oko 20 puta $|Genom|$ memorije [2]. U ovom slučaju, gde ljudski genom ima 3GB, 60GB RAM-a ovim pristupom je veliko unapređenje u odnosu na 1TB, ali i dalje možemo smanjiti ovu konstantu. Postoje algoritmi koji ovakvo stablo konstruišu u linearnom vremenu, ali su veoma zahtevni za implementaciju, kao što je *Ukkonen*-ov algoritam pomenut u odeljku 2.6. U narednom odeljku ćemo objasniti jedan od lakših načina za konstruisanje ovog stabla.

2.3.2 Konstrukcija kompresovanog sufiksnog stabla

Da bismo konstruisali kompresovano sufiksno stablo, malo ćemo modifikovati objašnjeni algoritam za konstrukciju sufiksnih stabala. Iako je svaka grana u ovom stablu obeležena jednim karakterom iz genoma, nije najjasnije sa koje pozicije u genomu dolazi taj karakter. Zato ćemo dodati još jednu informaciju na grane stabla koja će prikazivati na kojoj poziciji u genomu se ovaj karakter nalazi. Ako se karakter ponavlja na više mesta u genomu, onda ćemo na svakom pojavljivanju za poziciju uzeti indeks lokacije koji ima najmanju vrednost. U nastavku teksta možemo vi-

deti pseudokod algoritma za ovu modifikaciju. Uzimaćemo sufikse od najdužeg do najkraćeg i idući niz stablo, od korena, dodavati čvorove na dole objašnjeni način:

Algoritam 5 Konstrukcija modifikovanog sufiksnog stabla

```
function KonstrukcijaModifikovanogSufiksnogStabla(Genom)
Trie ← stablo od jednog čvora koji zovemo koreniČvor
for i = 0 to |Genom| − 1 do
    trenutniČvor ← koreniČvor
    for j = i to |Genom| − 1 do
        trenutniSimbol ← j-ti simbol u Genom
        if postoji grana iz trenutniČvor označena karakterom trenutniSimbol then
            trenutniČvor ← čvor u koji ide ta grana
        else
            dodaj novi čvor noviČvor u Trie
            nova grana od trenutniČvor do noviČvor
            dodaj poziciju (j) i simbol (trenutniSimbol) kao informacije na novu
            granu
            trenutniČvor ← noviČvor
        end if
    end for
    if trenutniČvor je list stabla Trie then
        dodaj karakter i kao oznaku u list
    end if
end for
return Trie
```

Na ovako modifikovanom stablu možemo sada primeniti kod objašnjen u sledećem pseudokodu i dobiti konačan izgled stabla koji smo priželjkivali (slika 2.11).

Algoritam 6 Konstrukcija kompresovanog sufiksnog stabla

```
function KonstrukcijaKompresovanogSufiksnogStabla(Genom)
Trie ← KonstrukcijaModifikovanogSufiksnogStabla(Genom)
for svaku putanju Putanja koja se ne račva u stablu Trie do
    zameni putanju Putanja jednom granom koja spaja prvi i poslednji čvor u
    putanji
    pozicija na grani je pozicija prve grane u putanji Putanja
    dužina na grani je broj grana u putanji Putanja
end for
return Trie
```

2.4 Uparivanje šablona pomoću sufiksnog niza

Alternativa kompresovanim stablima koja zahteva manje memorije je sufiksni niz. Da bismo konstruisali sufiksni niz, prvo treba leksikografski sortirati sve sufikse niske *Genom* (uzimamo da je karakter `$`, koji smo dodali, pre svih leksikografski). Sufiksni niz je niz koji se sastoji od svih indeksa u niski *Genom* gde ovako sortirani sufiksi počinju.

početni indeksi	sortirani sufiksi
12	$
10	am$
7	fizam$
0	homomorfizam$
8	izam$
11	m$
2	momorfizam$
4	morfizam$
1	omomorfizam$
3	omorfizam$
5	orfizam$
6	rfizam$
9	zam$

Slika 2.12: Sortirani sufiksi sa početnim indeksima za nisku `homomorfizam`

Sufiksni niz za primer genoma `homomorfizam`:

SufiksniNiz(`homomorfizam`) = [12, 10, 7, 0, 8, 11, 2, 4, 1, 3, 5, 6, 9]

Konstrukcija sufiksnog niza se, kao što smo napomenuli, može izvesti sortiranjem svih sufiksa niske *Genom*. Čak i najbrži algoritmi za sortiranje zahtevaju $O(n \cdot log(n))$ poređenja, tako da je u ovom slučaju potrebno $O(|Genom| \cdot log(|Genom|))$ poređenja. Postoje algoritmi kojima možemo kreirati sufiksni niz u linearnom vremenu i zahtevaju petinu memorije u odnosu na sufiksna stabla, ali o njima ovde neće biti reči [3]. U ovom optimizovanom slučaju nam onda nije potrebno 60GB memorije, već svega 12GB.

Jednom kada imamo konstruisan sufiksni niz, možemo razmotriti problem uparivanja šablona *Patern* sa niskom *Genom*. Kada radimo pronalaženje šablona u

sufiksnom stablu, sva pojavljivanja šablona u sufiksima se nalaze na početku tog sufiksa. Takođe, sortiranjem sufiksa dobijamo da svi sufiksi koji počinju istim sekvencama budu grupisani jedni do drugih (slika 2.12). Sledeći algoritam, baziran na binarnoj pretrazi, nalazi prvi i poslednji indeks gde se niska *Patern* poklapa sa početkom sufiksa.

Algoritam 7 Pronalaženje Paterna u sufiksnom nizu

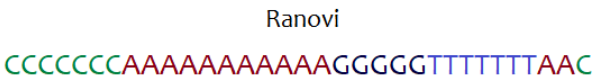
function PronalaženjePaternaSufiksniNiz(Genom, Patern, SufiksniNiz)
$minIndeks \leftarrow 0$
$maxIndeks \leftarrow |Genom|$
while $minIndeks < maxIndeks$ **do**
  $midIndeks \leftarrow (minIndeks + maxIndeks)/2$
  if $Patern >$ sufiks *Genoma* koji počinje na poziciji $SufiksniNiz(midIndeks)$ **then**
    $minIndeks \leftarrow midIndeks + 1$
  else
    $maxIndeks \leftarrow midIndeks$
  end if
end while
$početniIndeks \leftarrow minIndeks$
$maxIndeks \leftarrow |Genom|$
while $minIndeks < maxIndeks$ **do**
  $midIndeks \leftarrow (minIndeks + maxIndeks)/2$
  if $Patern <$ sufiks *Genoma* koji počinje na poziciji $SufiksniNiz(midIndeks)$ **then**
    $maxIndeks \leftarrow midIndeks$
  else
    $minIndeks \leftarrow midIndeks + 1$
  end if
end while
$završniIndeks \leftarrow maxIndeks$
if $početniIndeks > završniIndeks$ **then**
  return Šablon se ne pojavljuje u niski *Genom*
else
  return (*početniIndeks*, *završniIndeks*)
end if

2.5 Uparivanje šablona pomoću Barouz-Vilerove transformacije

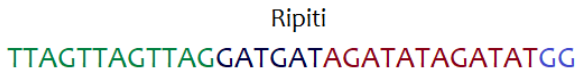
Ideja je dalje smanjiti količinu memorije potrebne za čuvanje niske *Genom*. Zato ćemo istražiti metode za kompresiju niski velikih dužina i njima pokušati da izvršimo kompresiju niske *Genom*.

2.5.1 Kompresija genoma

Za kompresiju genomske sekvence od značaja nam je da utvrdimo da li u genomskoj sekvenci ima nekih pravilnosti koje bi se mogle kodirati. Možemo razlikovati dva slučaja: kada imamo nekoliko uzastopnih ponavljanja jedne aminokiseline (to nazivamo ranovima : *runs*) (slika 2.13) i kada imamo nekoliko uzastopnih ponavljanja niza aminokiselina (to nazivamo ripitima : *repeats*) (slika 2.14)

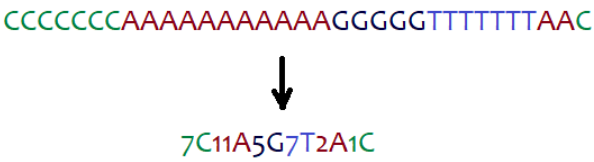


Slika 2.13: Primer ranova u genomu



Slika 2.14: Primer ripita u genomu

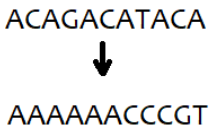
Kod ranova, k uzastopnih ponavljanja jedne aminokiseline možemo kodirati brojem k i oznakom aminokiseline koja se ponavlja (slika 2.15).



Slika 2.15: Primer kodiranja ranova

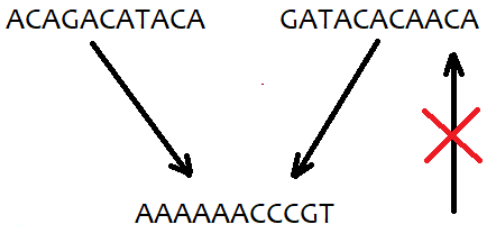
Problem je što kod genoma nemamo mnogo ranova, ali imamo dosta ripita. Zato bi bilo dobro kada bismo imali tehniku kojom bismo pretvorili ripite u ranove i na to primenili prethodno opisanu tehniku za kodiranje ranova.

Naivni pristup bi bio da sortiramo simbole u genomu leksikografski i na taj način ih grupišemo (slika 2.16). Ovom metodom bismo ljudski genom, za koji znamo da u memoriji zauzima oko 3GB, mogli prikazati sa samo 4 broja i 4 oznake aminokiselina ako primenimo i kodiranje ranova posle toga.



Slika 2.16: Sortiranje aminokiselina leksikografski

Međutim, svaka niska sa istim brojem istih aminokiselina će biti sortirana istim redosledom (slika 2.17). Zbog toga ne možemo izvršiti inverznu operaciju da bismo dobili inicijalnu nisku, što ovaj pristup čini neprimenljivim.



Slika 2.17: Više sekvenci sa istim sortiranjem

2.5.2 Barouz-Vilerova transformacija

Algoritam koji rešava pretvaranje ripita u ranove je Barouz-Vilerova transformacija. Za njegovu primenu prvo treba kreirati listu svih cikličnih rotacija niske *Genom*. To ćemo postići tako što ćemo odseći sufiks sa kraja niske *Genom* i dodati takav sufiks na početak niske *Genom* i tako za svaki sufiks. Zatim ćemo takve niske sortirati leksikografski, slično kao kod sufiksnog niza, i dobiti matricu dimenzija $|Genom| \cdot |Genom|$ koju nazivamo *Barouz-Vilerova matrica* (slika 2.18).

Ciklične rotacije	Barouz-Vilerova matrica
homomorfizam$	$ h o m o m o r f i z a m
$homomorfizam	a m $ h o m o m o r f i z
m$homomorfiza	f i z a m $ h o m o m o r
am$homomorfiz	h o m o m o r f i z a m $
zam$homomorfi	i z a m $ h o m o m o r f
izam$homomorf	m $ h o m o m o r f i z a
fizam$homomor	m o m o r f i z a m $ h o
rfizam$homomo	m o r f i z a m $ h o m o
orfizam$homom	o m o m o r f i z a m $ h
morfizam$homo	o m o r f i z a m $ h o m
omorfizam$hom	o r f i z a m $ h o m o m
momorfizam$ho	r f i z a m $ h o m o m o
omomorfizam$h	z a m $ h o m o m o r f i

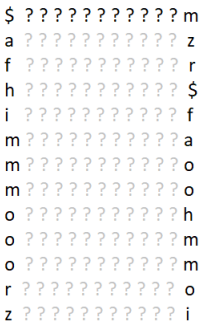
Slika 2.18: Ciklične rotacije niske `homomorfizam` i njena Barouz-Vilerova matrica. Poslednja kolona matrice predstavlja Barouz-Vilerovu transformaciju niske genoma (**BWT**(`homomorfizam`) = `mzr$faoohmmoi`)

Primetimo da je prva kolona matrice dobijena korišćenjem prethodno pomenutog naivnog pristupa koji leksikografski sortira karaktere niske *Genom*. Druga kolona sadrži drugi karakter od svih cikličnih rotacija niske *Genom*, tako da i ona predstavlja njene karaktere raspoređene u nekom redosledu i to važi za svaku kolonu ove matrice. Poslednju kolonu ove matrice nazivamo *Barouz-Vilerovom transformacijom* niske *Genom*, ili skraćeno *BWT(Genom)*. Kao što možemo videti na slici 2.18, za primer genoma `homomorfizam`, BWT glasi **BWT**(`homomorfizam`) = `mzr$faoohmmoi`. Ovu nisku dalje možemo kompresovati gorenavedenom tehnikom brojanja uzastopnih karaktera.

2.5.3 Inverzna Barouz-Vilerova transformacija

Kao što smo već rekli, potrebno je da možemo da uradimo i inverznu operaciju, kako bismo mogli da rekonstruišemo početnu nisku. Razmotrimo već poznati primer za koji znamo da je BWT(*Genom*) = `mzr$faoohmmoi`. Pored ovoga znamo da prvu kolonu Barouz-Vilerove matrice čine leksikografski sortirani karakteri BWT(*Genom*). U ovom slučaju to je niska `$afhimmmooorz`. Radi lakšeg pisanja ove niske ćemo zvati *poslednjaKolona* i *prvaKolona*, respektivno.

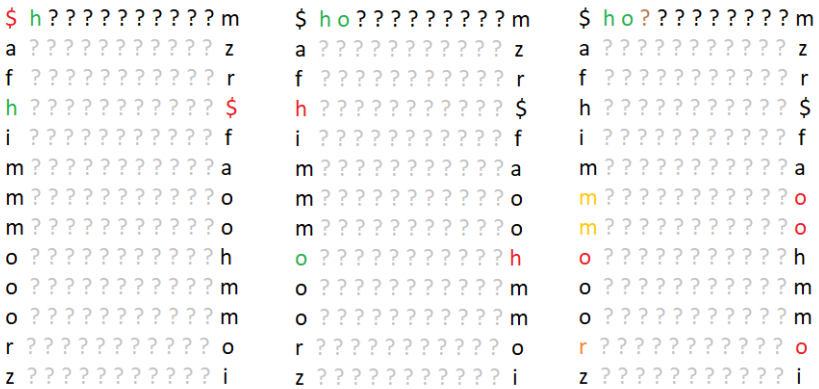
Do sada poznati deo matrice je prikazan na slici 2.19.



Slika 2.19: Prva i poslednja kolona Barouz-Vilerove matrice

Znamo da je prvi red matrice ciklična rotacija niske *Genom* koja počinje karakterom `$`, koji se nalazi na kraju niske. Zbog toga ako utvrdimo karaktere u prvom redu matrice možemo opet cikličnim rotiranjem pomeriti karakter `$` na kraj reda i dobiti početnu nisku *Genom*. Postavlja se pitanje kako da odredimo preostale karaktere u prvom redu ako je sve što znamo *poslednjaKolona* i *prvaKolona*?

Prvi karakter prvog reda je, kao što je rečeno, karakter `$`. Ako potražimo `$` u poslednjoj koloni, videćemo da se on javlja u četvrtom redu matrice. Na osnovu toga znamo da je karakter desno od njega cikličnom rotacijom pomeren na početak tog reda, tj. nalazi se u četvrtom redu u prvoj koloni. U ovom slučaju to je karakter `h`. Prateći istu logiku cikličnih rotacija, možemo nastaviti dalje, ali u jednom trenutku može da nam se desi da imamo više opcija za sledeći karakter. U ovom primeru to se događa na trećem karakteru. Nakon `o` možemo imati karaktere `m` ili `r` (slika 2.20).



Slika 2.20: Primer više mogućnosti za naredni karakter. Posle karaktera `o` možemo imati `m` ili `r`

First-Last svojstvo

Da bismo pronašli ostale simbole početne niske, posmatraćemo Barouz-Vilerovu matricu i indeksiraćemo prvu kolonu i svaki karakter u njoj u redosledu pojavljivanja tog karaktera u prvoj koloni. Za primer `homomorfizam` ćemo prvo posmatrati karakter `m` (slika 2.21).

$	h	o	m	o	m	o	r	f	i	z	a	m
a	m	$	h	o	m	o	m	o	r	f	i	z
f	i	z	a	m	$	h	o	m	o	m	o	r
h	o	m	o	m	o	r	f	i	z	a	m	$
i	z	a	m	$	h	o	m	o	m	o	r	f
m_1	$	h	o	m	o	m	o	r	f	i	z	a
m_2	o	m	o	r	f	i	z	a	m	$	h	o
m_3	o	r	f	i	z	a	m	$	h	o	m	o
o	m	o	m	o	r	f	i	z	a	m	$	h
o	m	o	r	f	i	z	a	m	$	h	o	m
o	r	f	i	z	a	m	$	h	o	m	o	m
r	f	i	z	a	m	$	h	o	m	o	m	o
z	a	m	$	h	o	m	o	m	o	r	f	i

Slika 2.21: Indeksiranje karaktera `m` u prvoj koloni u redosledu pojavljivanja u toj koloni.

Pogledajmo karakter m_1 u prvoj koloni koji se pojavljuje na početku ciklične rotacije m_1`$homomorfiza`. Ako ciklično rotiramo ovu nisku do dobijanja niske `homomorfiza`m_1`$` primetićemo da je ovaj karakter m zapravo tek treće pojavljivanje ovog karaktera u `homomorfizam$`. Na sličan način možemo utvrditi i pojavljivanja ostalih karaktera m u genomu.

$$\mathbf{ho}m_2\mathbf{o}m_3\mathbf{orfiza}m_1\mathbf{\$}$$

Sada, da bismo locirali karakter m_1 u poslednjoj koloni, ciklično ćemo za jedno mesto rotirati šesti red, gde se karakter m_1 nalazi. Time dobijamo nisku `$homomorfiza`m_1 koja se poklapa sa prvim redom Barouz-Vilerove matrice. Istim postupkom možemo dobiti izgled poslednje kolone sa redosledom karaktera m.

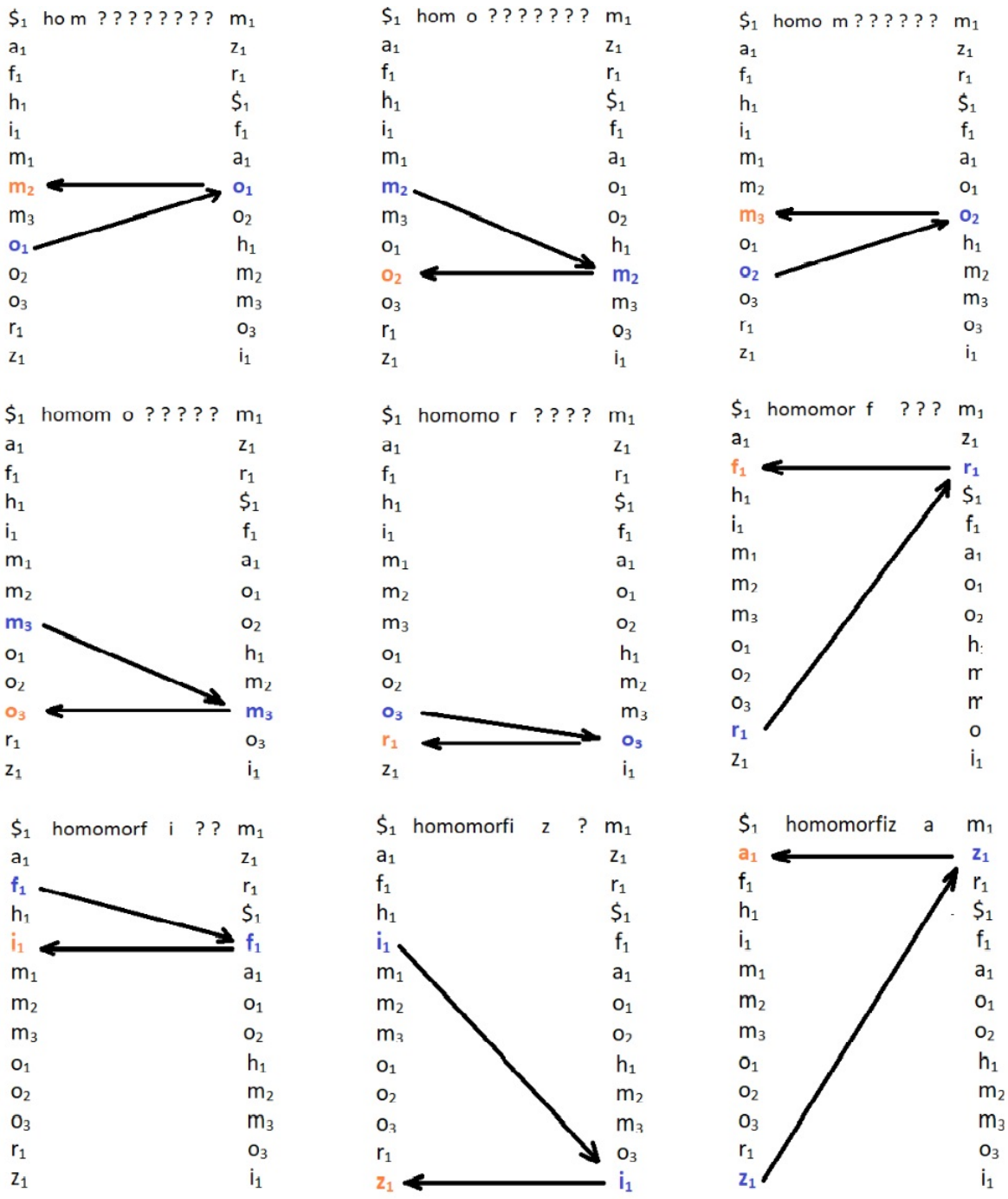
$$m_1\mathbf{zr\$faooh}m_2m_3\mathbf{oi}$$

Ovaj princip važi za svaku nisku *Genom* i za svaki karakter koji odaberemo iz takve niske i naziva se *First-Last svojstvo* (slika 2.22). Formalno, *First-Last* svojstvo podrazumeva da k-to pojavljivanje simbola u prvoj koloni Barouz-Vilerove matrice i k-to pojavljivanje simbola u poslednjoj koloni Barouz-Vilerove matrice odgovaraju istoj poziciji tog simbola u niski *Genom*.

$	h	o	m	o	m	o	r	f	i	z	a	m_1
a	m	$	h	o	m	o	m	o	r	f	i	z
f	i	z	a	m	$	h	o	m	o	m	o	r
h	o	m	o	m	o	r	f	i	z	a	m	$
i	z	a	m	$	h	o	m	o	m	o	r	f
m_1	$	h	o	m	o	m	o	r	f	i	z	a
m_2	o	m	o	r	f	i	z	a	m	$	h	o
m_3	o	r	f	i	z	a	m	$	h	o	m	o
o	m	o	m	o	r	f	i	z	a	m	$	h
o	m	o	r	f	i	z	a	m	$	h	o	m_2
o	r	f	i	z	a	m	$	h	o	m	o	m_3
r	f	i	z	a	m	$	h	o	m	o	m	o
z	a	m	$	h	o	m	o	m	o	r	f	i

Slika 2.22: Pojavljivanje sva tri karaktera `m` u prvoj koloni je u istom redosledu kao u poslednjoj koloni.

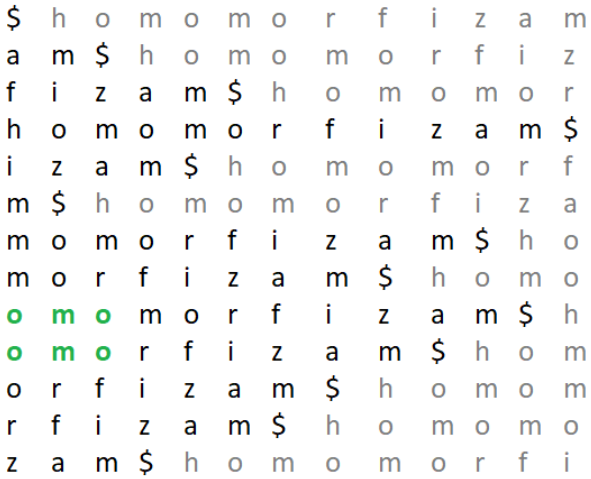
Ovo svojstvo možemo upotrebiti kao alat za izvršavanje inverzne Barouz-Vilerove transformacije. Potrebno je da indeksiramo svaki simbol u prvoj i poslednjoj koloni kao na prethodno demonstriranom primeru za *First-Last* svojstvo. Ponavljajući postupak koji je demonstriran na slici 2.20, samo sada sa indeksiranim kolonama, dobijamo rekonstruisanu početnu nisku *Genom*. Ovaj postupak je moguće izvršiti za svaku nisku koja sadrži jedan karakter `$` u sebi. Postupak za primer genoma `homomorfizam` je prikazan na slici 2.23.



Slika 2.23: Inverzna Barouz-Vilerova transformacija za nisku genoma `homomorfizam` počev od kritične tačke prikazane na slici 2.20

2.5.4 Uparivanje šablona

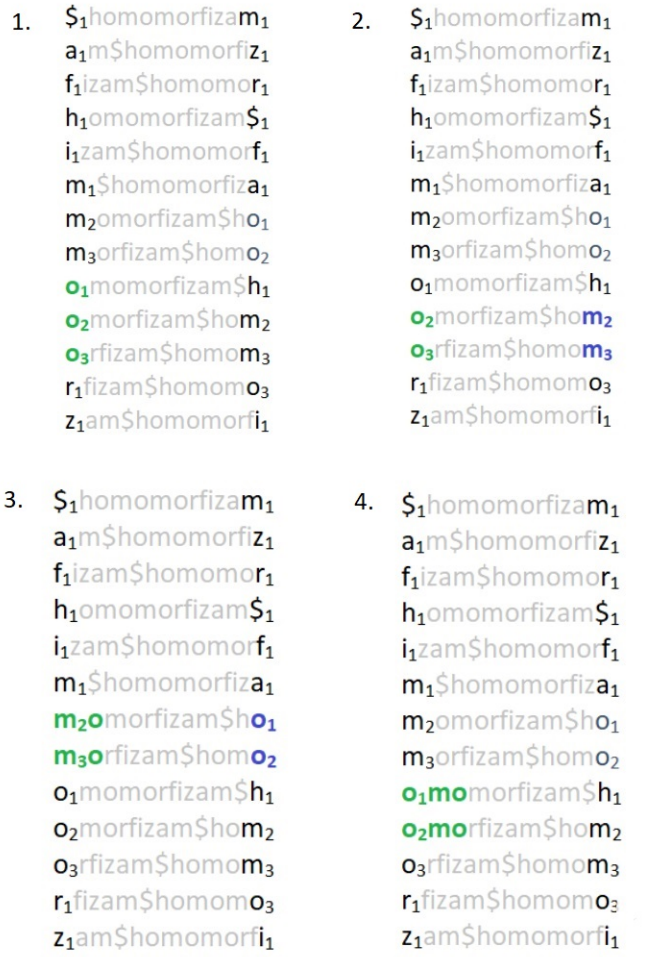
Ideja za korišćenje Barouz-Vilerove transformacije u svrhu uparivanja šablona potiče od činjenice da svaki red Barouz-Vilerove matrice počinje različitim sufiksima niske *Genom*. Pošto su ovi sufiksi već leksikografski sortirani, sva poklapanja niske *Patern* sa niskom *Genom* će se naći na početku uzastopnih redova ove matrice, što je sličan zaključak kao kod sufiksnih nizova (slika 2.24).



Slika 2.24: Poklapanje šablona `omo` u uzastopnim redovima Barouz-Vilerove matrice za genom `homomorfizam`.

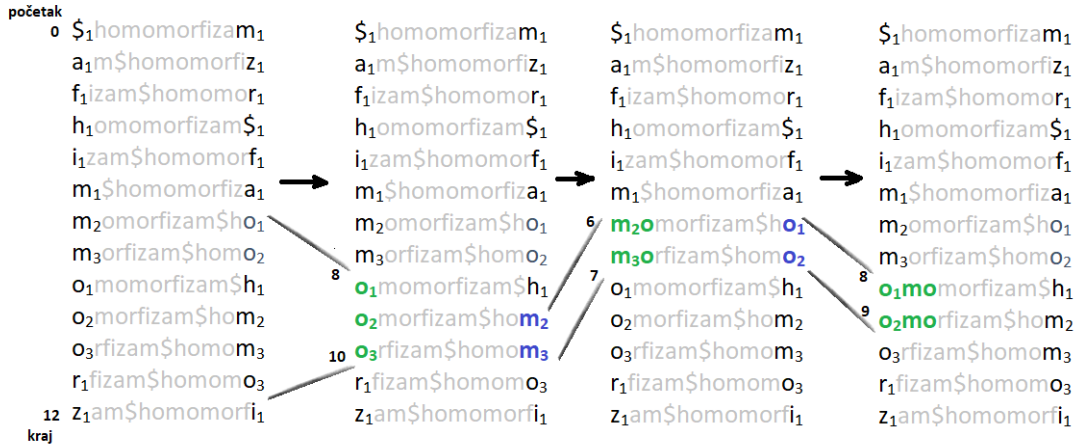
Problem sa ovim pristupom je što ne ide u prilog težnji ka smanjenju korišćene memorije jer čuvanje čitave Barouz-Vilerove matrice za nisku *Genom* zahteva $|Genom|^2$ mesta u memoriji. Primetimo da čuvanje čitave Barouz-Vilerove matrice i nije neophodno i da je moguće da samo na osnovu prve i poslednje kolone matrice uradimo inverziju. Koristeći ove dve kolone naći ćemo poklapanja niske *Patern* u niski *Genom* tako što ćemo krenuti od poslednjeg karaktera niske *Patern*. Na primer, tražićemo pojavljivanja niske `omo` u niski `homomorfizam`. Postupak je detaljno prikazan na slici 2.25.

Kao što smo rekli, krenućemo od poslednjeg karaktera niske `omo`, to je karakter `o` i naći ćemo sva njegova pojavljivanja u prvoj koloni. Kao što vidimo na slici 2.25 karakter `o` se nalazi u devetom, desetom i jedanaestom redu. Kako nastavljamo dalje nailazimo na karakter `m`. Da bismo pronašli njega iskoristićemo svojstvo da se simboli u poslednoj koloni nalaze ispred simbola u istom redu u prvoj koloni. Vidimo da se karakter `m` u poslednjoj koloni nalazi u desetom i jedanaestom redu, to



Slika 2.25: Traženje poklapanja niske `omo` u niski `homomorfizam` pomoću prve i poslednje kolone Barouz-Vilerove matrice

su m_2 i m_3. U prvoj koloni to su karakteri u sedmom i osmom redu. Sledećim korakom završavamo sa prvim karakterom `o`. Tražimo njegovo pojavljivanje u sedmom i osmom redu, ali u poslednjoj koloni. Vidimo da se na oba mesta nalazi karakter `o`, u ovom slučaju o_1 i o_2. Njih u prvoj koloni uočavamo u desetom i jedanaestom redu i tu nalazimo poklapanja datog šablona. Pošto znamo da se poklapanja šablona nalaze u uzastopnim redovima, možemo uvesti pokazivače *početak* i *kraj* koji će pratiti izvršavanje sa slike 2.25 i pokazivati na prvo pojavljivanje poklapanja u prvoj koloni i poslednje pojavljivanje poklapanja u prvoj koloni, respektivno. Na kraju će ovi pokazivači sadržati informaciju o prvom i poslednjem redu prve kolone u kojem imamo poklapanje šablona i na taj način kao povratnu informaciju možemo imati broj poklapanja koji iznosi $kraj - početak + 1$ (slika 2.26).



Slika 2.26: Menjanje pokazivača na početak i kraj redova u prvoj koloni u kojima imamo poklapanje šablona i genoma.

Izvršavanje Barouz-Vilerove transformacije se može opisati sledećim algoritmom.

Algoritam 8 Uparivanje šablona pomoći Barouz-Vilerove transformacije

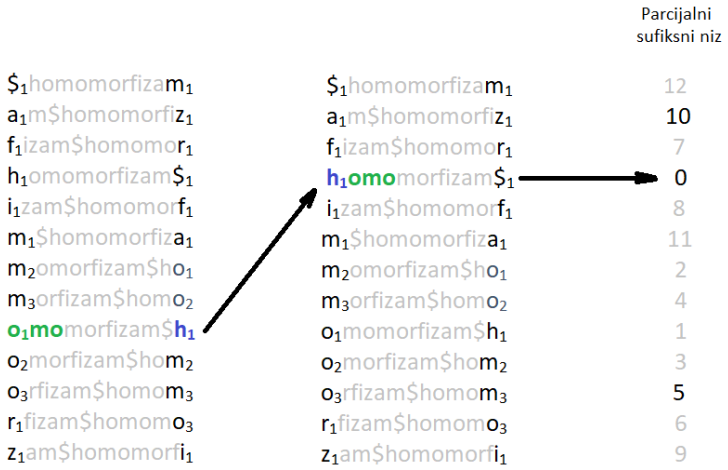
function UparivanjeBWT(PrvaKolona, PoslednjaKolona, Patern)
početak ← 0
kraj ← $|PoslednjaKolona| - 1$
while *početak* ≤ *kraj* **do**
  if *Patern* nije prazan **then**
    karakterZaProveru ← poslednji karakter niske *Patern*
    Ukloniti poslednji karakter niske *Patern*
    if poslednja kolona na pozicijama od *početak* do *kraj* sadrži *karakterZaProveru* **then**
      početniIndeks ← prva pozicija *karakterZaProveru* od pozicije *početak* do *kraj* u *PoslednjaKolona*
      krajnjiIndeks ← poslednja pozicija *karakterZaProveru* od pozicija *početak* do *kraj* u *PoslednjaKolona*
      početak ← pozicija *početniIndeks* u *PrvaKolona*
      kraj ← pozicija *krajnjiIndeks* u *PrvaKolona*
    else
      return 0
    end if
  else
    return *kraj*− *početak* +1
  end if
end while

Primetimo da prethodnim pristupom kao povratnu informaciju dobijamo samo broj pojavljivanja niske *Patern* u niski *Genom*. Želimo da kao povratnu informaciju dobijemo indekse u niski *Genom* na kojima se *Patern* pojavljuje. To je moguće postići korišćenjem sufiksnog niza koji bi pratio Barouz-Vilerovu matricu i pokazivao indekse svih sufiksa koji je čine. Međutim, to ne ide u prilog težnji ka smanjenju korišćene memorije. Jedno rešenje ovog problema je da koristimo parcijalni sufiksni niz niske *Genom*. Da bismo ga konstruisali, od celog sufiksnog niza ćemo ostaviti samo vrednosti koje su deljive sa nekim brojem K. Na slici 2.27 prikazan je parcijalni sufiksni niz za nisku genoma `homomorfizam` i K=5. U realnim slučajevima K je 100, što znači da koristimo 100 puta manje memorije nego sa celim sufiksnim nizom.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
Sufiksni niz	12	10	7	0	8	11	2	4	1	3	5	6	9

Slika 2.27: Parcijalni sufiksni niz

Ovaj niz možemo koristiti za nalaženje indeksa u uparivanju šablona, što možemo prikazati na jednom od pronađenih rešenja za nisku `omo` u niski `homomorfizam` (slika 2.28). Iako smo pronašli rešenje nastavimo dalje postupak dok se ne nađemo u redu gde parcijalni sufiksni niz ima indeks. U ovom slučaju to je sledeći korak i red koji počinje sa `homo`. Parcijalni sufiksni niz ima indeks 0, a nama je bio potreban jedan dodatni korak da do njega dođemo, što znači da je mesto pojavljivanja niske `omo` pozicija $0 + 1 = 1$.



Slika 2.28: Primer korišćenja parcijalnog sufiksnog niza.

2.5.5 Približno uparivanje šablona

U realnim situacijama je vrlo retko egzaktno uparivanje šablona, a mnogo je češća situacija da postoje odstupanja i da je potrebno pretražiti da li se skoro svi karakteri niske *Patern* pojavljuju u niski *Genom* u istom redosledu. Ovaj problem se naziva problem približnog uparivanja šablona i uparivanje može biti jednostruko (problem 3) i višestruko (problem 4).

PROBLEM 3 (PROBLEM PRIBLIŽNOG JEDNOSTRUKOG UPARIVANJA ŠABLONA)

ulaz: *Niska* ***Patern****, niska* ***Genom****, ceo broj* ***d****.*

izlaz: *Sve pozicije u niski* ***Genom*** *gde se niska* ***Patern*** *pojavljuje kao podniska sa najviše* ***d*** *razlika.*

PROBLEM 4 (PROBLEM PRIBLIŽNOG VIŠESTRUKOG UPARIVANJA ŠABLONA)

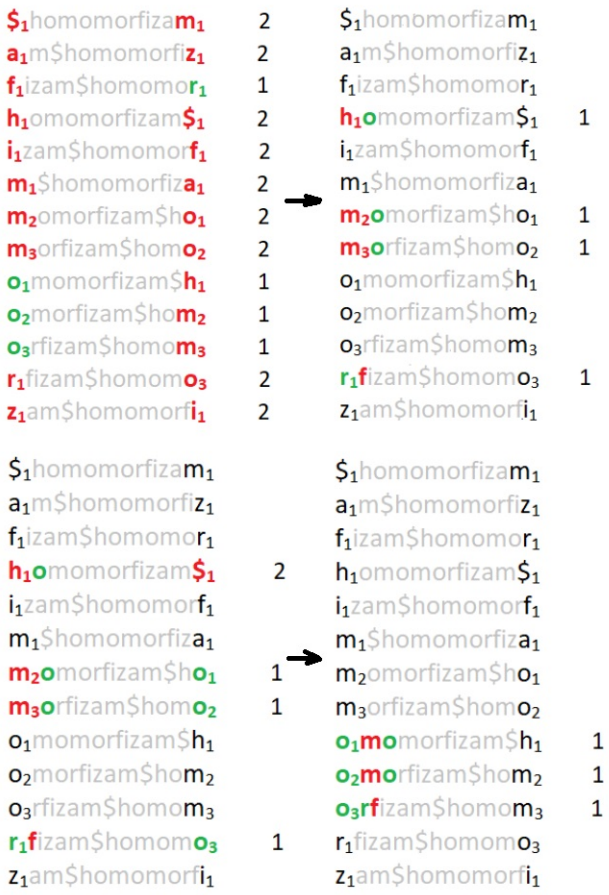
ulaz: *Kolekcija niski* ***PaternLista****, niska* ***Genom****, ceo broj* ***d****.*

izlaz: *Sve pozicije u niski* ***Genom*** *gde se niske iz kolekcije* ***PaternLista*** *pojavljuje kao podniske sa najviše* ***d*** *razlika.*

Kod Barouz-Vilerove transformacije približno uparivanje šablona možemo uraditi tako što ćemo u toku pretrage nastaviti kad naiđemo na nepoklapanje sve dok je broj razlika $\leq d$.

Na slici 2.29 je prikazan slučaj uparivanja niske `oro` u niski `homomorfizam` za $d = 1$. Kao što vidimo, na prvoj slici, uzećemo poslednji karakter šablona, karakter `o` i pogledati da li imamo poklapanja u prvoj koloni. Poklapanja imamo u devetom, desetom i jedanaestom redu, ali još uvek ne odbacujemo ni ostale redove, jer je trenutno za njih broj grešaka 1, što je i dalje $\leq d$. Sada posmatramo drugi karakter šablona otpozadi, karakter `r`. Tražimo poklapanja u poslednjoj koloni i vidimo da samo u trećem redu imamo poklapanje. Povećavamo broj promašaja za ostale kolone. Za većinu kolona je broj promašaja prešao $d = 1$, tako da njih izbacujemo iz daljeg razmatranja. Ostali su redovi tri, devet, deset i jedanaest. Njihove karaktere u poslednjem redu sada tražimo u prvom redu, to su redovi četiri, sedam, osam i dvanaest. Uzimamo sada treći karakter šablona otpozadi, karakter `o` i njega tražimo u poslednjoj koloni gorepomenutih redova. Vidimo da imamo poklapanje u redovima sedam, osam i dvanaest, četvrtom redu povećavamo broj promašaja i on ispada iz

daljeg razmatranja jer je broj promašaja 2. Pronalazimo sada karaktere iz poslednje kolone ostalih redova u prvoj koloni. Oni se nalaze u redovima devet, deset i jedanaest. Nije nam ostao nijedan karakter u šablonu za pretragu, tako da smo stigli do kraja. Pronađena rešenja su: `omo`, `omo` i `orf`, svi sa po jednim pogrešno uparenim karakterom, ali zadovoljavaju to da je broj grešaka $\leq d$.



Slika 2.29: Primer približnog uparivanja šablona `oro` sa niskom genoma `homomorfizam`.

Primetimo da, da u prvom koraku nismo uzeli u obzir sve karaktere prve kolone, ne bismo došli do rezultata u jedanaestom redu (`orf`). Tu je poslednji karakter pogrešan a poređenje kreće upravo od poslednjeg karaktera niske *Patern*.

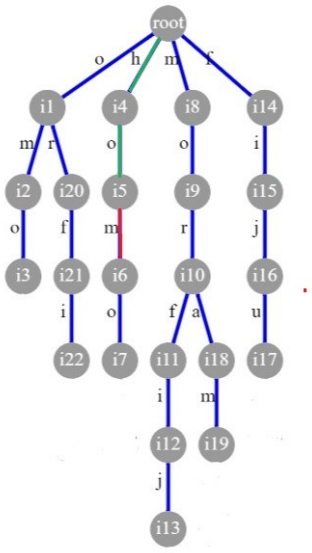
2.6 Još neki algoritmi za uparivanje šablona

U ovom odeljku će biti prikazana još dva algoritma koji neće biti implementirani u elektronskoj lekciji. Prvi se odnosi na uparivanje šablona pomoću prefiksnih stabala i zove se algoritam *Aho-Corasick* i donosi nam unapređenje u pretraživanju prefiksnog stabla sa složenosti O($|Genom|\cdot|NajdužiPatern|$) na O($|Genom|$). Drugi algoritam se odnosi na konstrukciju kompresovanog sufiksnog stabla u linearnom vremenu i konstruisao ga je *Esko Ukkonen*. Oba ova algoritma imaju dosta varijacija i raznih unapređenja, a u ovom radu će biti objašnjene njihove osnovne verzije.

2.6.1 Algoritam *Aho-Corasick* za uparivanje šablona pomoću prefiksnih stabala

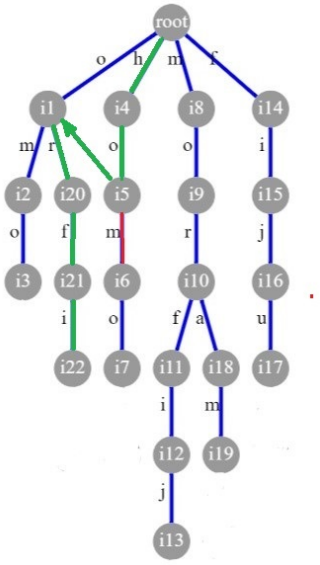
Ovaj algoritam za višestruko uparivanje šablona je razvijen 1975. godine od strane *Alfreda Aho*-a i *Margaret Corasick*. Vreme izvršavanja ovog algoritma je O($|PaternLista| + |Genom| + |m|$), gde je m broj pronađenih poklapanja. Vidimo da ovde složenost algoritma zavisi od broja pronađenih rešenja.

Vratimo se prvo na inicijalno prikazani algoritam uparivanja šablona pomoću prefiksnih stabala. Tako kreirano stablo je sačinjeno od šablona iz *PaternLista* (slika 2.2). Na primer, neka niska *Genom* bude `horfi`. Krenućemo od korenog čvora i cele niske i nakon dva karaktera dolazimo do nepoklapanja (slika 2.30).



Slika 2.30: Nepoklapanje niske `horfi` u stablu sa slike 2.2

Nakon toga krenuli bismo opet od korenog čvora sa niskom `orfi`. Primetimo da smo već pronašli poklapanje za prvi karakter ove niske, karakter `o` kada smo pretraživali nisku `horfi`. S obzirom na to, možemo promeniti pristup i krenuti od čvora u koji grana koja polazi iz korenog čvora sa karakterom `o` ulazi. Da bismo mogli to da uradimo, potrebno je da stablo malo modifikujemo i dodamo takozvane *sufiksne veze* (eng. *suffix links* ili *failure links*). Za prethodno navedeni primer stablo tada izgleda kao na slici 2.31.



Slika 2.31: Sufiksna veza posle čvora `o` u stablu sa slike 2.30, omogućava nam da možemo da skočimo sa putanje koja počinje sa `ho`, na putanju koja počinje sa `o`.

Sufiksne veze povezuju čvor v i čvor w ako je putanja do čvora w najduži sufiks putanje do čvora v koji se pojavljuje u stablu. Algoritam *Aho-Corasick* nam omogućava da posle svakog nepoklapanja izbegnemo ponovno kretanje do korenog čvora i na taj način ubrzamo izvršavanje.

Međutim, ovde može da se desi da propustimo neke prave sufikse i potrebno je u stablo dodati i tzv. *izlazne veze* (eng. *output links*). Izlazna veza čvora koji kompletira putanju w pokazuje na čvor koji kompletira putanju sa najdužim pravim sufiksom koji ujedno imamo i kao šablon koji tražimo ili ga nema ukoliko takav sufiks ne postoji. Detalji algoritma *Aho-Corasick* prevazilaze okvire ovog rada i mogu se pronaći u [4, 5].

2.6.2 Ukkonen-ov algoritam za kreiranje kompresovanog sufiksnog stabla

Ukkonen-ov algoritam za kreiranje kompresovanog sufiksnog stabla je algoritam predložen od strane *Esko Ukkonen*-a 1995. godine [6].

Ukkonen-ov algoritam konstruiše niz implicitnih stabala, od kojih se poslednje konvertuje u kompresovano sufiksno stablo niske *Genom* [7]. Implicitno sufiksno stablo je stablo dobijeno od kompresovanog sufiksnog stabla uklanjanjem svakog pojavljivanja specijalnog karaktera $ sa oznake svake grane, zatim uklanjanjem svake grane koja nema oznaku i na kraju uklanjanjem svakog čvora koji nema makar dva potomka.

Ukkonen-ov algoritam konstruiše implicitno sufiksno stablo za svaki prefiks niske genoma. Prvo kreira stablo samo od prvog karaktera niske *Genom*, nakon toga od prefiksa koji sadrži prva dva karaktera i tako sve do cele niske *Genom*. Implicitno stablo konstruisano od cele niske se onda konvertuje u kompresovano sufiksno stablo i za ceo ovaj postupak je potrebno $O(|Genom|)$ vremena.

Inicijalni algoritam koji obuhvata kreiranje implicitnih stabala ima složenost od $O(|Genom|^3)$, ali se različitim optimizacijama vreme svodi na linearno. U nastavku možemo videti osnovnu strukturu *Ukkonen*-ovog algoritma.

Algoritam 9 Osnovna struktura *Ukkonen*-ovog algoritma

function UkkonenovoStablo(Genom)
for $i = 0$ to $|Genom| - 1$ **do**
  for $j = 0$ to $i + 1$ **do**
    polazeći od korena nalazi se kraj puta označenog sa $Genom[j...i]$ u tekućem stablu
    ako je potrebno taj put se produžava dodavanjem znaka $Genom[i+1]$ i time omogućava da je niska $Genom[j...i+1]$ u stablu
  end for
end for
return $T_{|Genom|}$

Postoje određena pravila koja nam omogućavaju produženje sufiksa pomenuto u prethodnom algoritmu. U produženju sufiksa koji kreće od nekog indeksa j, kada algoritam naiđe na kraj podniske *Genom*[j...i] u tekućem stablu, ono produžuje ovu podnisku da bi osiguralo da je sufiks *Genom*[j...j+1] u stablu.
Ovo radimo po jednom od naredna tri pravila:

- U tekućem stablu, put *Genom*[j...i] se završava u listu. To znači da se put od korena produžava do kraja neke grane sa listom. Da bismo ažurirali ovo stablo znak *Genom*[i+1] se dodaje na kraj oznake na toj grani sa listom.

- Nijedan put od kraja niske *Genom*[j...i] ne počinje sa *Genom*[i+1], ali barem jedan označen put nastavlja od kraja *Genom*[j...i]. U ovom slučaju nova grana sa listom koja počinje od kraja *Genom*[j...i] mora biti napravljena i označena znakom *Genom*[i+1]. Novi čvor će takođe morati da bude napravljen ako se *Genom*[j...i] završava unutar grane. Novonastalom listu se dodeljuje broj j.

- Neka put od kraja niske *Genom*[j...i] počinje znakom *Genom*[i+1]. U ovom slučaju niska *Genom*[j...i+1] je već u tekućem stablu, tako da se dalje ne radi ništa (U implicitnom sufiksnom stablu kraj sufiksa ne mora da bude eksplicitno označen).

Kasnije se optimizacija sprovodi u vidu sufiksnih veza prethodno pomenutih i kod *Aho-Corasick*-ovog algoritma. Takođe se koristi konvertovanje podniski na oznakama grana u uređeni par brojeva koji čine indeks u niski *Genom* gde ta oznaka počinje i dužinu oznake na grani (što je takođe već pominjano u odeljku o kompresovanim sufiksnim stablima).

Konačno implicitno stablo može da se transformiše u kompresovano sufiksno stablo u vremenu O(|*Genom*|). Jednostavno ćemo dodati karakter kraja niske `$` na *Genom* i pustiti da *Ukkonen*-ov algoritam doda ovaj znak. Sada nijedan sufiks nije prefiks bilo kog drugog sufiksa, tako da se ovde izvršavanje završava implicitnim sufiksnim stablom gde se svaki sufiks završava u listu i time je eksplicitno postavljen. Dobijeno stablo je pravo kompresovano sufiksno stablo i vreme ovih promena je O(|*Genom*|).

Glava 3

Uputstvo za korišćenje elektronske lekcije

Pošto je glavna tema ovog rada elektronska lekcija koja sadrži neke od gorenavedenih algoritama, potrebno je objasniti kako tu aplikaciju možemo koristiti. U narednim odeljcima će ukratko biti opisani preduslovi koje određeni sistem treba da ispunjava da bi aplikacija mogla da se instalira i pokrene na njemu, način instalacije i pokretanja i svi algoritmi implementirani unutar aplikacije sa objašnjenim slučajevima upotrebe.

Aplikacija je sačinjena od dve komponente: klijentskog i serverskog dela aplikacije. Serverski deo je implementiran u programskom jeziku *Java* [8], u razvojnom okviru *Spring* [9] i strukturiran je kao *veb API*, dok je klijentski deo implementiran u razvojnom okviru *React* [10] i zavisi od *Node.js*-a [11]. Izvorni kod aplikacije se nalazi na *GitHub*-u, u javnom repozitorijumu [12].

3.1 Priprema okruženja i instalacija

U prvom delu ovog odeljka će biti objašnjeni koraci koji su izvršeni nakon implementacije u svrhu olakšavanja samog pokretanja elektronske lekcije za krajnjeg korisnika koje on neće morati da izvršava. U drugom delu će biti prikazano kako krajnji korisnik treba da koristi prethodno pripremljene komponente da bi pokrenuo elektronsku lekciju na svom okruženju. Alati korišćeni u narednim koracima su *Docker* [13] i *Git* [14].

3.1.1 Dokerizacija

Radi lakšeg korišćenja i instalacije, aplikacija koju ovaj rad pokriva je dokerizovana. *Docker* je *open-source* alat koji služi da aplikaciju pokrenemo sa svim potrebnim bibliotekama, promenljivim, itd. u vidu jednog paketa - *docker* kontejnera. *Docker* kontejner možemo videti kao virtualno okruženje u kojem je naša aplikacija pokrenuta.

Da bi aplikacija mogla biti dokerizovana mora postojati datoteka koja govori u kom okruženju i na koji način izvršiti dokerizaciju. Ta datoteka se obično naziva *Dockerfile*. Pošto postoje dve komponente koje treba pokrenuti u drugačijim okruženjima (klijentski i serverski deo aplikacije), svaka komponenta će imati svoj *Dockerfile*.

Što se serverskog dela tiče, za njega nam je potrebno *Java* okruženje i *Dockerfile* će izgledati kao na slici 3.1.

backend > Dockerfile

```
1  FROM openjdk:11-jre-slim
2
3  ENV SERVER_PORT=5555
4  EXPOSE 5555
5
6  VOLUME /tmp
7  ADD ./target/bwt-0.0.1-SNAPSHOT.jar app.jar
8  ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar" ]
```

Slika 3.1: *Dockerfile* za serverski deo aplikacije.

Za klijentski dao je potreban *Node.js*, tako da *Dockerfile* za ovu komponentu izgleda kao na slici 3.2.

frontend > Dockerfile

```
1  FROM node:alpine
2  WORKDIR /app
3  COPY package.json ./
4  COPY package-lock.json ./
5  COPY ./ ./
6  RUN npm i
7  CMD ["npm", "run", "start"]
```

Slika 3.2: *Dockerfile* za klijentski deo aplikacije.

Kada smo napisali *Dockerfile*, potrebno je izvršiti komandu `build` koja će napraviti *Docker image*. *Docker image* je datoteka koja se sastoji iz više slojeva, a

ti slojevi predstavljaju komande koje treba izvršiti. Pokretanjem *docker image*-a dobićemo *Docker* kontejner, tj. okruženje koje je pripremljeno pomoću komandi iz *Dockerfile*-a i u kojem će naša aplikacija biti pokrenuta. Ideja je da tako napravljen *docker image* okačimo na javni *docker repository*, tako da *docker image* bude dostupan za preuzimanje i pokretanje na računaru korisnika. U svrhu ovog rada je napravljen *Docker repository* pod imenom *filipmiljakovic1994/bioinfo_matf_bg* [12] i na njemu će biti postavljeni svi *docker image*-i od značaja.

Da bismo izvršili komandu `build`, potrebno je da preuzmemo kod sa *Github*-a na lokalnu mašinu, pozicioniramo se u odgovarajući direktorijum (*/frontend* ili */backend*) gde se kod i *Dockerfile* nalaze i izvršimo komandu sa slike 3.3.

```
docker build -t filipmiljakovic1994/bioinfo_matf_bg:backend-1.0.2 .
docker build -t filipmiljakovic1994/bioinfo_matf_bg:frontend-1.0.2 .
```

Slika 3.3: Komande *docker build image* za obe komponente aplikacije

Ovim komandama ćemo napraviti *docker image*-e za obe komponente aplikacije. Nakon opcije `-t` navodimo naziv kojim hoćemo da tagujemo *docker image*. Prvi deo naziva mora da bude naziv *Docker repository*-ja na koji hoćemo da okačimo *docker image*, to je u ovom slučaju *filipmiljakovic1994/bioinfo_matf_bg*, nakon toga navodimo specifičan tag *docker image*-a posle oznake „**:**”. Za komponente ove aplikacije to su **frontend-1.0.2** i **backend-1.0.2**. U ovoj komandi je veoma bitno da se navede „**.**” na kraju, što označava da u paket koji će da se kreira treba da uđu svi direktorijumi i datoteke iz trenutnog direktorijuma.

Da bismo proverili da li je *docker image* uspešno kreiran, možemo izvršiti komandu `docker images` nakon koje će nam biti izlistane svi *docker image*-i na našoj mašini, gde možemo videti da se ove upravo kreirane tu i nalaze.

Nakon ovoga potrebno je da *docker image* okačimo na *docker repository*. To se radi pomoću komande `push`. Ovo jeste javni *docker repository*, ali samo za čitanje, tj. preuzimanje *docker image*-a, dok za komandu `push` ipak moramo da budemo autorizovani. Autorizacija treba biti urađena pomoću komande `docker login`, nakon koje treba navesti kredencijale vlasnika *Docker repository*-ja. Posle uspešno izvedene autorizacije izvršićemo sledeće komande sa slike 3.4.

```
docker push filipmiljakovic1994/bioinfo_matf_bg:backend-1.0.2

docker push filipmiljakovic1994/bioinfo_matf_bg:frontend-1.0.2
```

Slika 3.4: Komande *docker push image* za obe komponente aplikacije

Nakon komande `push`, *docker image*-i se nalaze na *Docker repository*-ju i svako može da ih preuzme i pokrene. Okruženje gde želimo da pokrenemo datu aplikaciju treba samo da poseduje instaliran *Docker* [13]. Pošto instalacija *Docker*-a zavisi od toga koji operativni sistem imamo, više o načinu instalacije *Docker*-a za specifične operativne sisteme može se pronaći u [13].

3.1.2 Pokretanje aplikacije

Da bi se aplikacija pokrenula, najpre je potrebno da preuzmemo *docker image* na lokalnu mašinu pomoću komande `pull`. Za ovde pomenute komponente odgovarajuće komande se nalaze na slici 3.5.

```
docker pull filipmiljakovic1994/bioinfo_matf_bg:backend-1.0.2

docker pull filipmiljakovic1994/bioinfo_matf_bg:frontend-1.0.2
```

Slika 3.5: Komanda *docker pull image* za obe komponente aplikacije

Nakon ove komande *docker image* se nalazi u korisnikovom lokalnom *Docker repository*-ju i on može da je pokrene. Komande za pokretanje gorepomenutih komponenti možemo videti na slici 3.6. Na ovaj način će aplikacija biti dostupna na portu 3000, na IP adresi mašine na kojoj je kontejner pokrenut (za lokalno pokretanje biće `http://localhost:3000`).

```
docker run --name filip_miljakovic_master_backend -p 5555:5555 -d filipmiljakovic1994/bioinfo_matf_bg:backend-1.0.2

docker run --name filip_miljakovic_master_frontend -p 3000:3000 -d filipmiljakovic1994/bioinfo_matf_bg:frontend-1.0.2
```

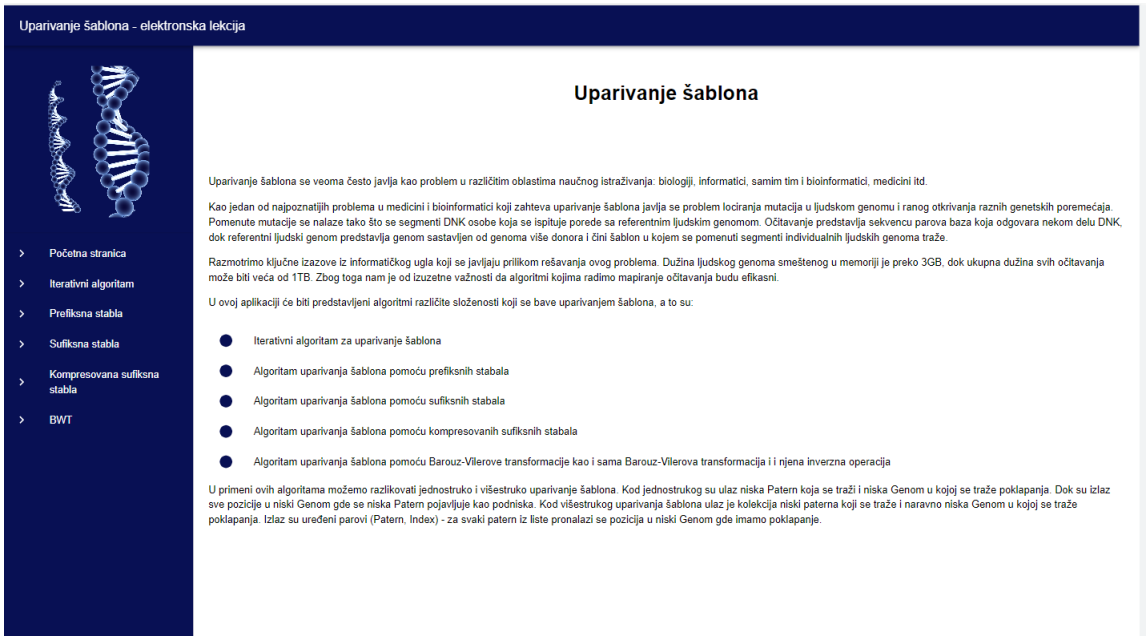
Slika 3.6: Komande *docker run image* za obe komponente aplikacije

U komandi `run` smo naveli ime kontejnera pomoću opcije `-name`. Opcija `-p` nam omogućava da navedemo na kom portu će se aplikacija pokretati. Obe naše komponente imaju portove na kojima se inicijalno pokreću, ali pomoću ove komande

možemo taj port promeniti za kontejner i aplikaciju pokrenuti na nekom drugom. U ovom slučaju portovi su ostali isti. Opcija `-d` nam omogućava da kontejner pokrenemo u pozadini kako bi trenutni terminal ostao funkcionalan nakon čega sledi ime *docker image*-a.

3.2 Početna stranica

Prilikom pristupanja aplikaciji na portu 3000, inicijalno će se otvoriti *Početna stranica*. Na ovoj, kao i na svim ostalim stranicama sa leve strane se nalazi *Side bar* koji sadrži veze koje vode do stranica sa specifičnim algoritmima. Klikom na neku od veza, otvara se stranica za unos ulaznih podataka za svaki od algoritama. Pored ovih veza, na ovoj stranici se može videti kratak teorijski uvod koji objašnjava problem koji želimo da rešimo ovim algoritmima. Početna stranica je prikazana na slici 3.7.

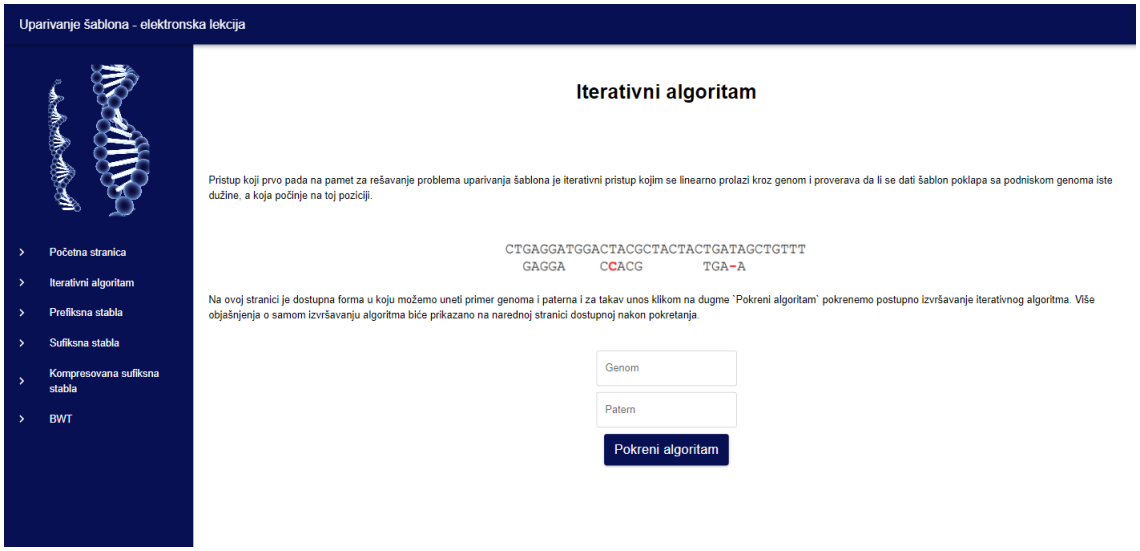


Slika 3.7: Početna stranica aplikacije.

3.3 Iterativno rešenje

Klikom na vezu sa nazivom *Iterativni algoritam* u *Side bar*-u otvara se stranica za unos niske *Genom* i niske *Patern* koju želimo da uparimo sa nekom podniskom

u genomu. Pored polja za unos ulaznih podataka, na stranici se nalazi i kratko teorijsko objašnjenje algoritma, kao i dugme *Pokreni algoritam* (slika 3.8).



Slika 3.8: Unos ulaznih parametara, niski *Genom* i *Patern* za iterativni algoritam.

Klikom na ovo dugme otvara se nova stranica na kojoj se izvršavanje iterativnog algoritma prikazuje korak po korak za unete ulazne parametre.

Za upravljanje izvršavanjem algoritma imamo na raspolaganju *Zaustavi/Pokreni* dugme kojim možemo pauzirati izvršavanje algoritma, ako u nekom stanju hoćemo da proverimo gde je algoritam stao. Pored toga, kada algoritam završi izvršavanje za unesene parametre, a mi iz nekog razloga želimo opet da ga pokrenemo sa istim parametrima, da se ne bismo vraćali na stranicu za unos parametara i opet ih unosili, postoji dugme *Resetuj* kojim se ovo može izvršiti. Pored navedene dugmadi postoji i slajder kojim možemo regulisati brzinu izvršavanja algoritma, tj pauzu između koraka. Brzina se kreće u rasponu od 100ms do 1000ms, tj. jedne sekunde. Incijalna brzina je 500ms.

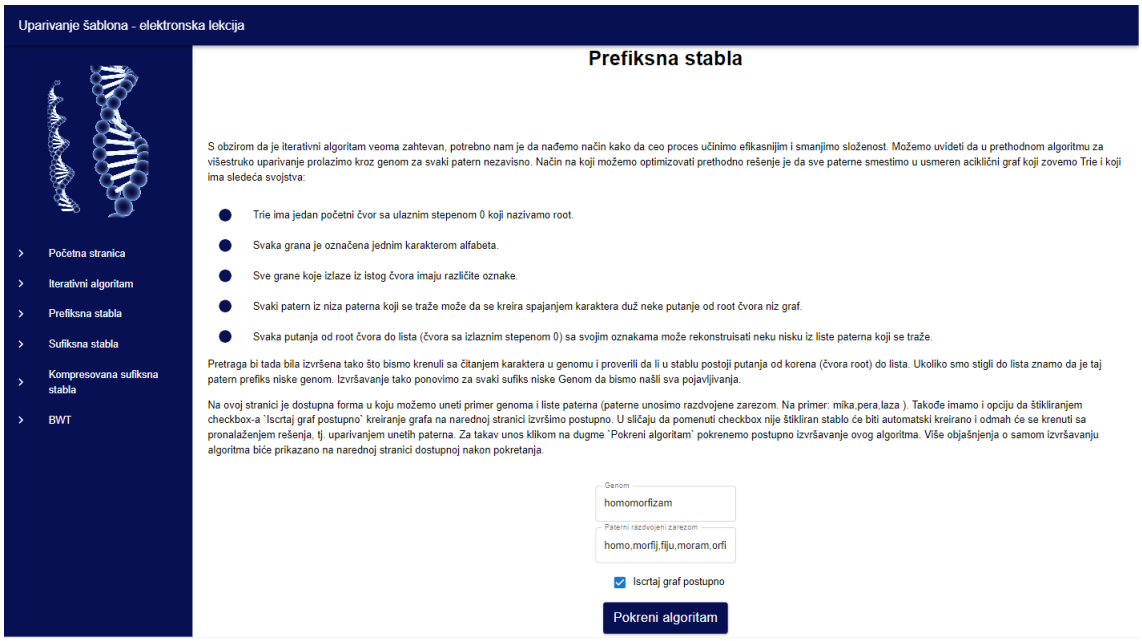
Na stranici se u toku izvršavanja može pratiti koji se trenutno karakter proverava i različitim bojama je naglašeno da li je do poklapanja došlo ili ne (zelena boja za poklapanje i crvena za nepoklapanje). Pored toga, u svakom trenutku možemo videti na kojim indeksima je poklapanje pronađeno do tada, a kada do novog poklapanja dođe lista indeksa će se dopuniti (slika 3.9).

Slika 3.9: Stranica na kojoj je prikazano izvršavanje iterativnog algoritma za ulazne parametre `homomorfizam` i `omo`.

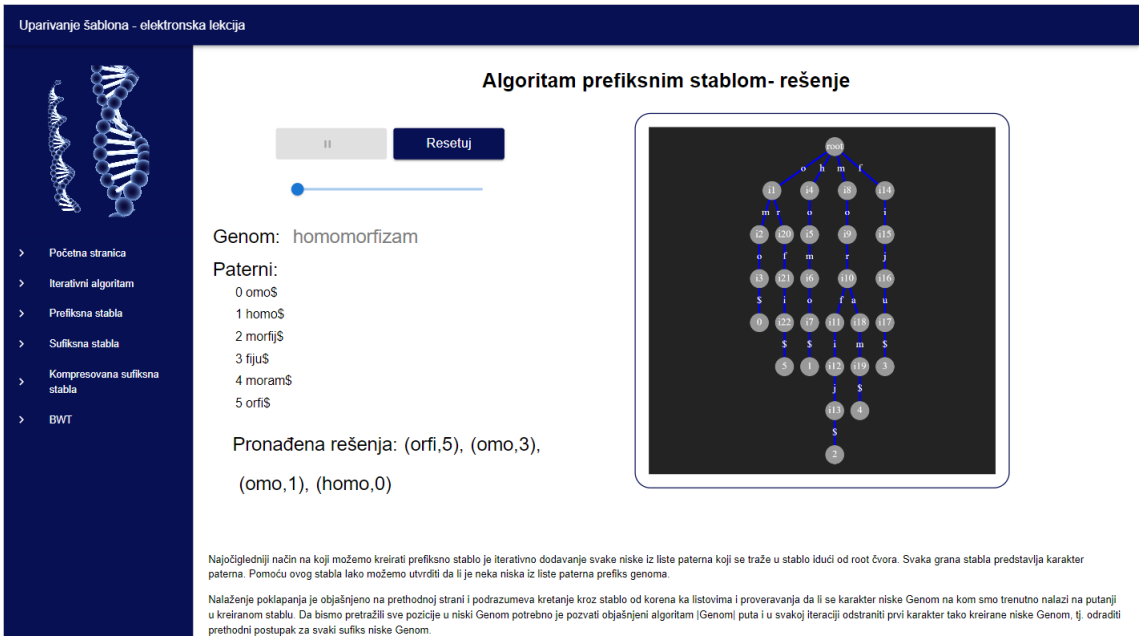
3.4 Rešenje korišćenjem prefiksnog stabla

Klikom na vezu pod nazivom *Prefiksna stabla* u *Side bar*-u otvara se stranica za unos niske *Genom* i liste *PaternLista* koja sadrži listu niski koje želimo da uparimo sa nekom podniskom u genomu. Ovde stranica osim što ima različitu teorijsku pozadinu od prethodne, ima *checkbox* kojim se reguliše da li će kreiranje prefiksnog stabla da se radi postupno ili će se samo prikazati njegov konačan izgled i polja za unos kao za prethodni algoritam. Pošto je potrebno da imamo mogućnost da unesemo i više šablon niski, njih ćemo uneti u polje za unos šablona razdvojene zarezima (slika 3.10).

Klikom na dugme *Pokreni algoritam* otvara se nova stranica koja pomenuti algoritam izvršava korak po korak. Na toj stranici se nalaze isti elementi za upravljanje izvršavanjem algoritma (*Zaustavi/Pokreni* dugme, *Resetuj* dugme i slajder za regulaciju brzine). Pored ovoga tu je i grafički prikaz kreiranja prefiksnog stabla od unesenih šablona. Sami šabloni su izlistani pored radi boljeg pregleda. Kreiranje može biti urađeno postupno (granu po granu) ili ne, u zavisnosti od toga da li je *checkbox* na prethodnoj strani štikliran. Na tom stablu će biti postupno prikazano i uparivanje, tj. nalaženje rešenja.



Slika 3.10: Unos ulaznih parametara, niske *Genom* i niski iz *PaternLista* razdvojene zarezima za algoritam pomoću prefiksnog stabla.

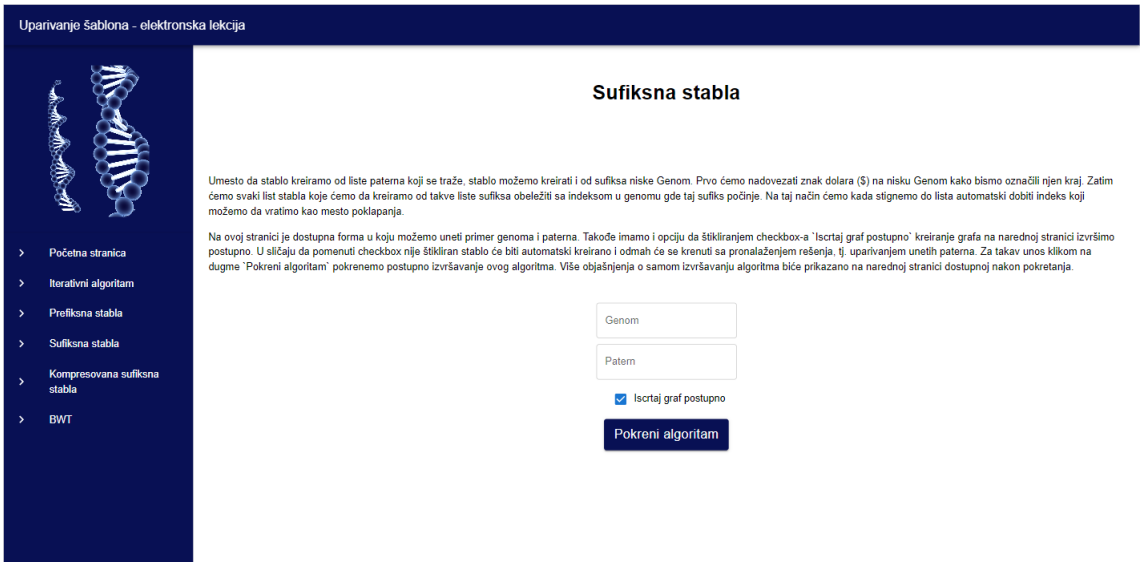


Slika 3.11: Stranica na kojoj je prikazano izvršavanje algoritma pomoću prefiksnog stabla.

Postupak nalaženja rešenja se osim na stablu može pratiti i na niski *Genom* koja je pored prikazana. Slično kao ranije, pozitivna poklapanja u datom trenutku će biti obojena zelenom bojom, a negativna crvenom bojom na granama, odnosno karakterima genoma. Pronađena rešenja su prikazana u vidu uređenih parova *(Šablon, Indeks)* gde je svaki pronađeni šablon prikazan sa mestom u niski *Genom* gde je poklapanje pronađeno (slika 3.11).

3.5 Rešenja korišćenjem sufiksnog stabla

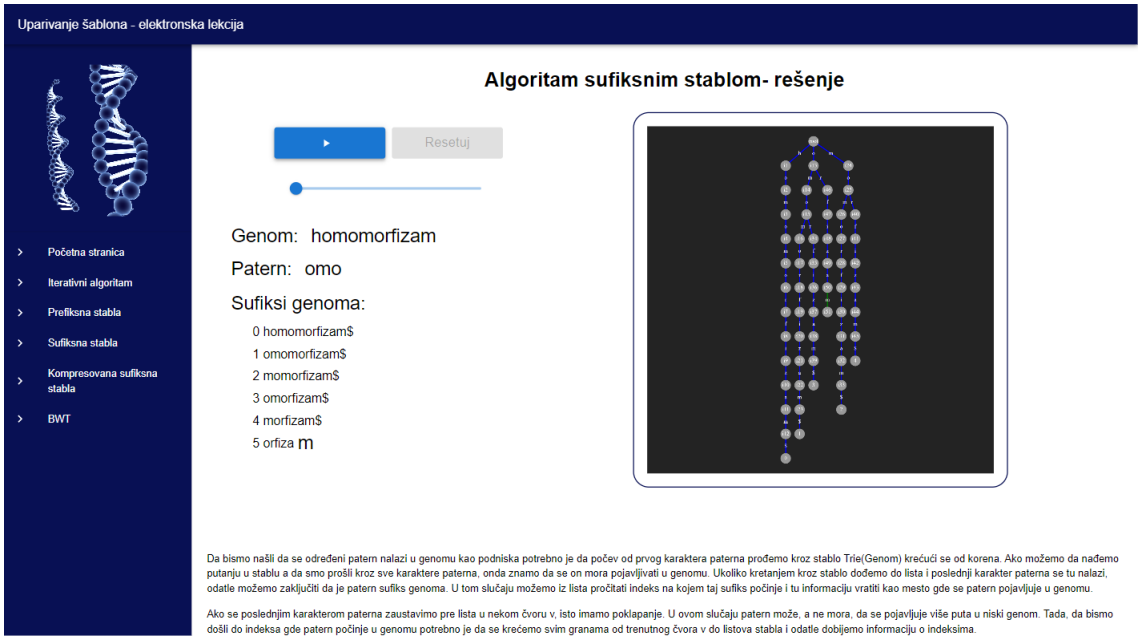
Klikom na vezu pod nazivom *Sufiksna stabla* u *Side bar*-u otvara se stranica za unos niske *Genom* i niske *Patern* koju želimo da uparimo sa nekom podniskom u genomu. Ovde možemo videti kombinaciju prethodne dve stranice za unos. Tu su polja za unos niski *Genom* i *Patern*, ali i *checkbox* pomenut kod prethodnog algoritma koji ima istu funkciju, samo u ovom slučaju za sufikno stablo. Tu je i drugačija teorijska osnova (slika 3.12).



Slika 3.12: Unos ulaznih parametara, niske *Genom* i niske *Patern* za algoritam pomoću sufiksnog stabla.

Klikom na dugme *Pokreni algoritam* otvara se nova stranica koja pomenuti algoritam izvršava korak po korak. Slično kao kod prethodnog algoritma, tu su elementi za upravljanjem izvršavanjem algoritma (*Zaustavi/Pokreni* dugme, *Resetuj* dugme i slajder za regulaciju brzine), grafički prikaz sufiksnog stabla i pored toga je ispisan

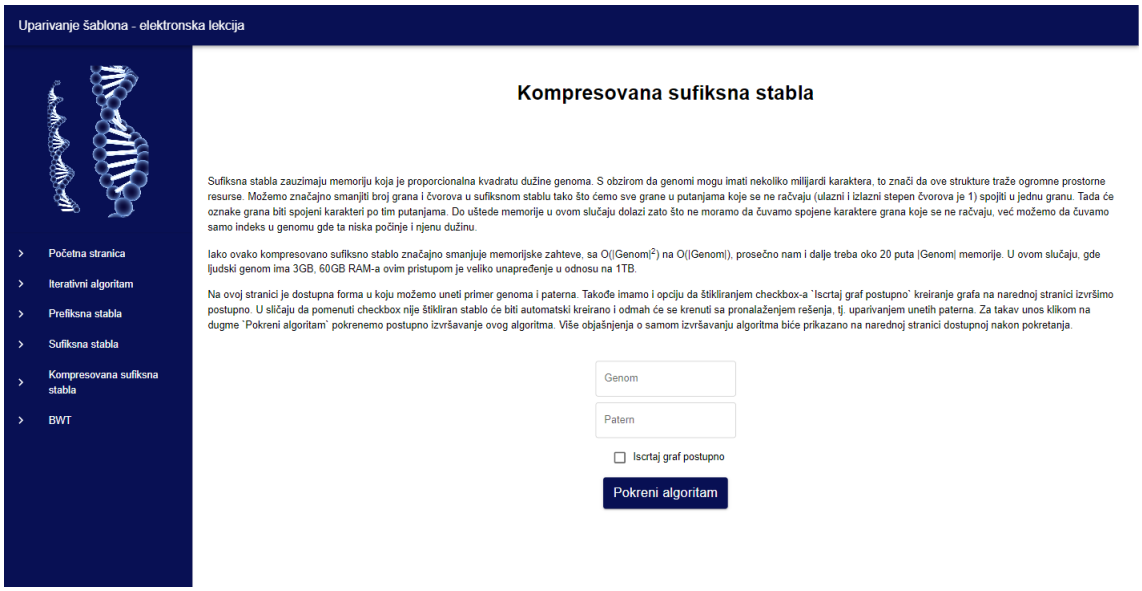
niz sufiksa niske *Genom* od kojeg je stablo kreirano. Način prikazivanja opet zavisi od *checkbox*-a sa prethodne strane za unos podataka. Rešenja, odnosno pronađena poklapanja su prikazana pored labele *Pronađena rešenja:* i tu su izlistani indeksi pronađeni do tog trenutka, a isto kao kod prethodnog algoritma praćenje poklapanja se može videti na prikazu stabla obojeno istim bojama (slika 3.13).



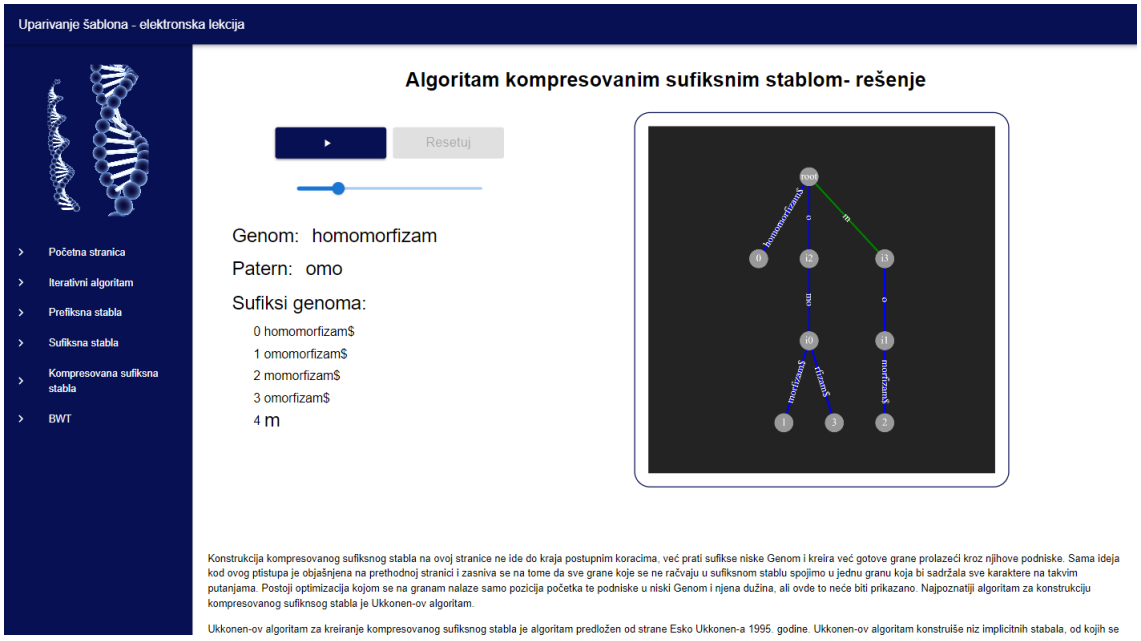
Slika 3.13: Stranica na kojoj je prikazano izvršavanje algoritma pomoću sufiksnog stabla.

Pored ovog, implementiran je i algoritam za prikaz kompresovanog sufiksnog stabla. Njemu ćemo pristupiti klikom na vezu *Kompresovana sufiksna stabla* u *Side bar*-u. Nakon klika, otvara se stranica za unos parametara koja je ista kao za obična sufiksna stabla, samo opet sa drugačijom teorijskom osnovom (slika 3.14).

Klikom na dugme *Pokreni algoritam* otvara se nova stranica na kojoj se pomenuti algoritam izvršava korak po korak. U svim ostalim komponentama ova stranica je ista kao za obična sufiksna stabla, samo će grafički prikaz stabla biti drugačiji. Ovaj algoritam neće prikazati postupno kreiranje kompresovanog sufiksnog stabla (npr. *Ukkonen*-ovim algoritmom, iako jednu njegovu verziju koristi u pozadini), ali će prikazati kreiranje svih grana i čvorova prolazeći kroz sufikse niske *Genom*.



Slika 3.14: Unos ulaznih parametara, niske *Genom* i niske *Patern* za algoritam pomoću kompresovanog sufiksnog stabla.



Slika 3.15: Stranica na kojoj je prikazano izvršavanje algoritma pomoću kompresovanog sufiksnog stabla.

Isto tako, uparivanje šablona će biti urađeno kao u prethodnom slučaju sufiksnih

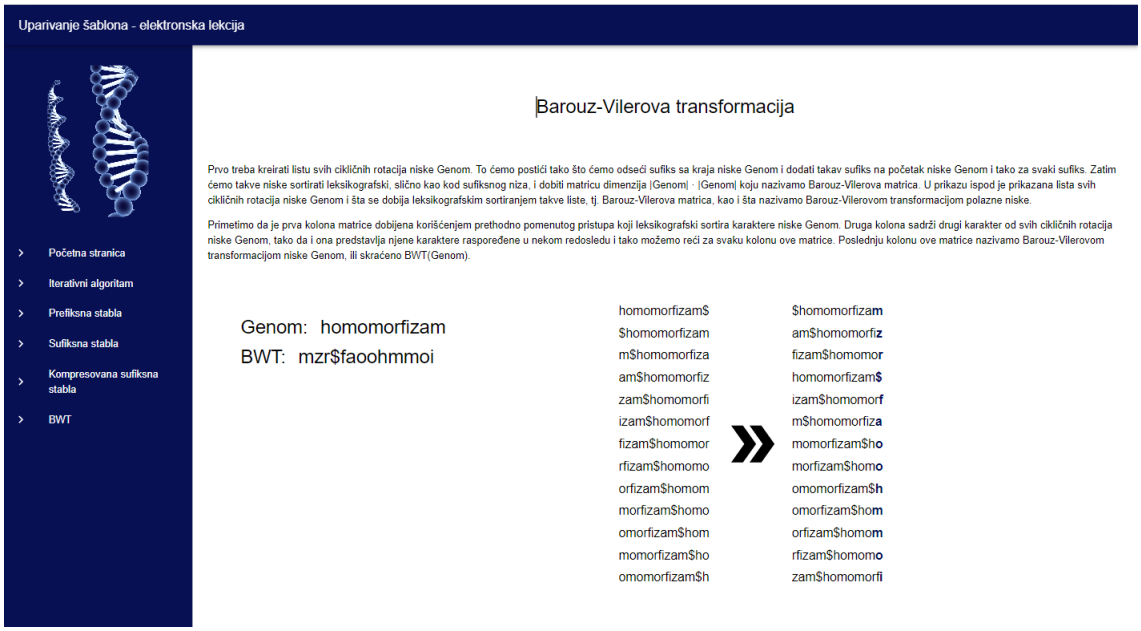
stabala, samo što ćemo i u slučaju da se ne dođe do kraja spojene putanje na jednoj grani, a da dođemo do kraja šablona, smatrati da je poklapanje pronađeno i vratiti indekse do kojih smo došli krećući se do listova (slika 3.15).

3.6 Rešenje korišćenjem Barouz-Vilerovog algoritma

Klikom na vezu pod nazivom *BWT* u *Side bar*-u otvara se stranica za unos podataka. Ovde imamo polja za unos niski *Genom* i *Patern*, ali i polje za unos broja nepoklapanja koja tolerišemo. Zapravo, ovde ćemo prikazati i približno uparivanje šablona. Pored toga, biće prikazana i adekvatna teorijska osnova (slika 3.16).

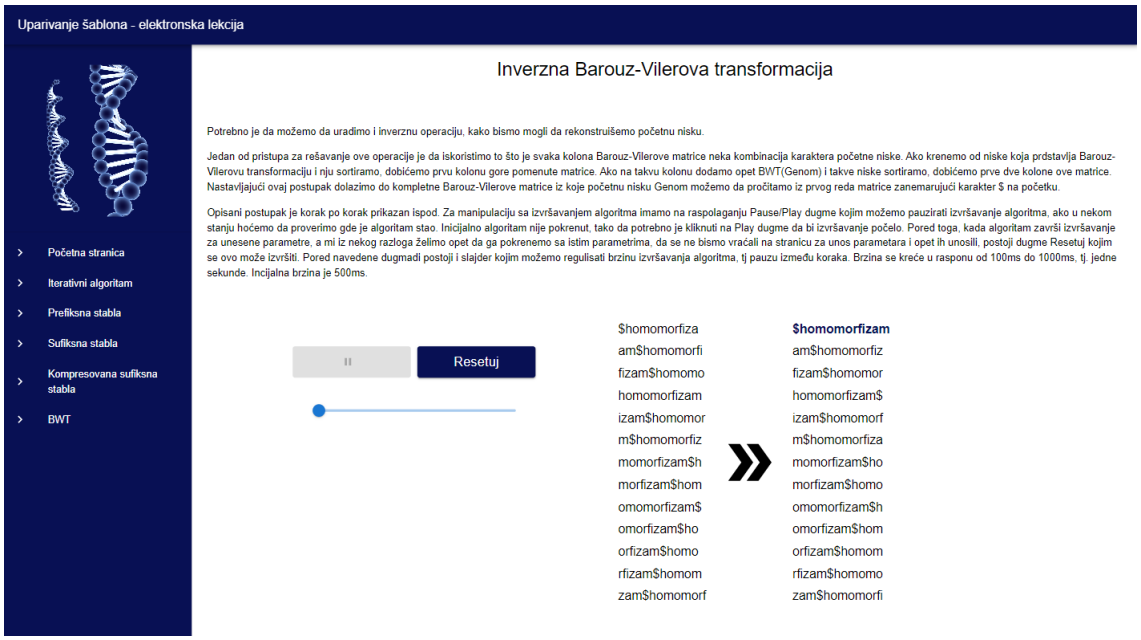
Slika 3.16: Unos ulaznih parametara, niske *Genom*, niske *Patern* i parametra koji označava broj nepoklapanja koja su dozvoljena za algoritam uparivanja pomoću Barouz-Vilerove transformacije.

Klikom na dugme *Pokreni algoritam* otvara se nova stranica na kojoj je prikazano više stvari. Prvi deo stranice prikazuje listu svih cikličnih rotacija niske *Genom* i njenu transformaciju, tj. sortiranje kako bi se dobila Barouz-Vilerova transformacija polazne niske *Genom* u poslednjoj koloni tako dobijene matrice (slika 3.17).



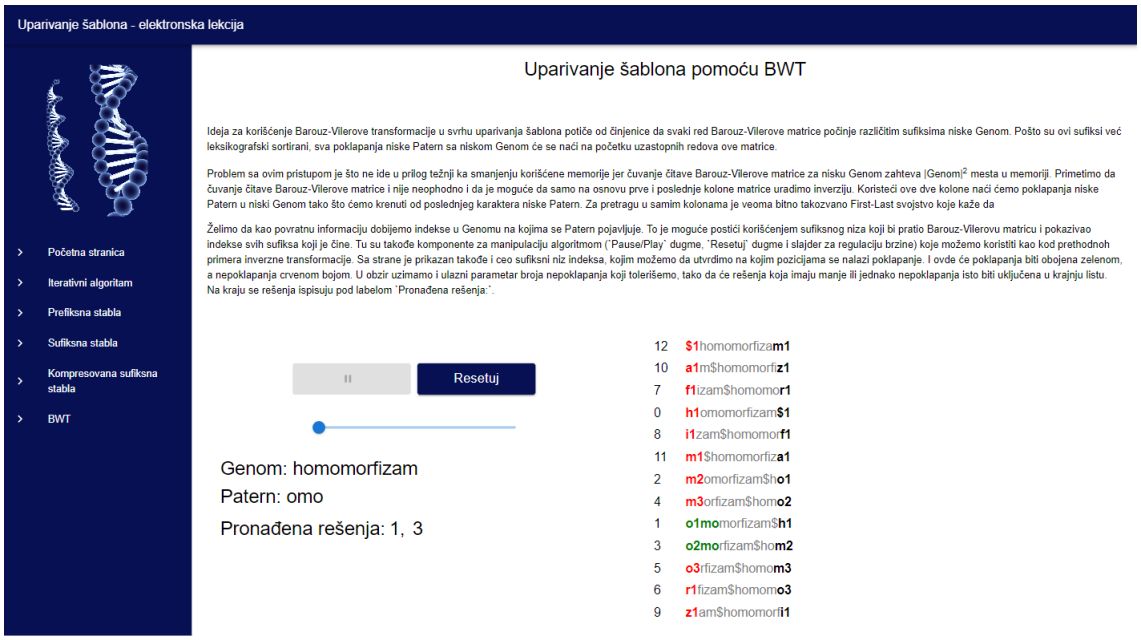
Slika 3.17: Lista cikličnih rotacija ulazne niske *Genom* i generisanje BWT niske.

Kada listamo niže niz stranicu, možemo pokrenuti postupno izvršavanje inverzne BWT za goredobijenu nisku BWT. Ovde imamo komponente za upravljanje izvršavanjem algoritma koje smo pominjali kod stranica sa stablima (*Zaustavi/Pokreni* dugme, *Resetuj* dugme i slajder za regulaciju brzine), sa istom funkcijom. Krajnji proizvod ovog algoritma je polazna niska *Genom* (slika 3.18).



Slika 3.18: Inverzna BWT transformacija goredobijene niske.

Daljim listanjem dolazimo do algoritma uparivanja šablona koji možemo da pokrenemo. Ovde je prikazana Barouz-Vilerova matrica u kojoj su prva i poslednja kolona podaci koje koristimo. Upotrebom *First-Last* svojstva i algoritma za uparivanje šablona pomoću BWT dolazimo do rešenja. Tu su takođe komponente za upravljanje izvršavanjem algoritma (*Zaustavi/Pokreni* dugme, *Resetuj* dugme i slajder za regulaciju brzine) koje možemo koristiti. Sa strane je prikazan i ceo sufiksni niz indeksa, kojim možemo da utvrdimo na kojim pozicijama se nalazi poklapanje. I ovde će poklapanja biti obojena zelenom, a nepoklapanja crvenom bojom. U obzir uzimamo i ulazni parametar broja nepoklapanja koji tolerišemo, tako da će rešenja koja imaju manje ili jednako nepoklapanja isto biti uključena u krajnju listu. Na kraju se rešenja ispisuju pod labelom *Pronađena rešenja:* (slika 3.19).



Slika 3.19: Stranica na kojoj je prikazano nalaženje rešenja pomoću Barouz-Vilerove transformacije.

Glava 4

Zaključak

U ovom radu su izloženi neki od algoritama koji se koriste za uparivanje šablona, tj. za pronalaženje poklapanja šablona kao podniski određene niske, kao i pronalaženje pozicija na kojima se ta poklapanja nalaze.

U uvodu je objašnjen značaj teme i koji su stvarni problemi koji se pomoću ovih algoritama rešavaju. U drugom odeljku su pobrojani neki od algoritama i njihove varijacije. Fokus je zadržan na algoritmima koji se koriste kao nastavni materijal za kurs *Uvod u bioinformatiku* na Matematičkom fakultetu Univerziteta u Beogradu. Redosledom prikazivanja se želeo prvo predstaviti iterativni algoritam koji ima najveću složenost, pa zatim algoritam za višestruko uparivanje šablona gde se rešenje traži pomoću stabla kreiranog od unesenih šablona za pretragu. Složenost se popravlja, ali posle ovog algoritma nailazimo na algoritam gde se za pretragu rešenja koriste sufiksna stabla, tj. stabla kreirana od sufiksa niske *Genom* u kojoj se pretraga vrši. Upoznajemo se sa pojmom kompresovanih sufiksnih stabala i načinima na koje može da se dođe do uštedenja memorije, jer algoritmi sa stablima zahtevaju dosta prostora za čuvanje informacija o stablu. Nakon toga predstavljamo sufiksni niz, koji problem memorije još malo popravlja. Na kraju dolazimo do Barouz-Vilerove transformacije i algoritma za uparivanje šablona koji ovu informaciju može da iskoristi i da smanji memoriju koja se koristi, kao i vreme izvršavanja. Uz osnovne implementacije pomenute su i neke adaptacije tih algoritama koje mogu dovesti do ubrzanja, ali zbog kompleksnosti nije ulaženo u detalje samih implementacija. Pored ovih algoritama, naveli smo i osnovne detalje nekoliko algoritama koji su široko poznati i imaju učestalu primenu u ovoj oblasti. Zbog velikog broja adaptacija i optimizacija, kako i pristupa koji postoje za ove algoritme, oni nisu detaljnije opisivani.

Pošto je osnovni cilj ovog rada elektronska lekcija koja bi kasnije mogla da po-

mogne studentima da lakše nauče pomenute algoritme i eksperimentišu sa njima, glavni akcenat je prebačen na taj deo. Implementirana je korisnička aplikacija koja ima interfejs u vidu *veb* stranice koja za svaki unos korisnika primenjuje pomenute algoritme. U odeljku uputstva za korišćenje je opisano koje funkcionalnosti sama aplikacija nudi i opisani su slučajevi upotrebe za sve implementirane algoritme. Aplikacija je takođe dokerizovana i njeni su *docker image*-i dostupni na javnom *docker repository*-ju, tako da svaki korisnik koji ima instaliran *Docker* alat može da preuzme *docker image*-e i da pokrene aplikaciju na svom okruženju. Kod je takođe javno dostupan.

Postoji dosta prostora za unapređenje trenutno implementirane verzije aplikacije. Kao što smo rekli, opisani algoritmi imaju mnoštvo adaptacija, poboljšanja koja nisu pokrivena ovom aplikacijom. Akcenat je bio, kao što je pomenuto, na algoritmima koji se koriste na predavanjima i vežbama pomenutog kursa. Svako poboljšanje može biti prikazano kroz posebnu stranicu i u tom slučaju bismo imali još bolji postupan uvid kako je pojedini algoritam evoluirao.

Bibliografija

1. Kovačević, J. *Materijali sa predavanja, Uvod u Bioinformatiku, Matematički fakultet Univerziteta u Beogradu* http://www.bioinformatika.matf.bg.ac.rs/predavanja/Chapter_9/Chapter_9_tekst.pdf. (septembar 2022.)
2. Philip Compeau, P. P. *Bioinformatics Algorithms, An Active Learning Approach, Vol. II, Chapter 9: How Do We Locate Disease-Causing Mutations?, Active Learning Publishers, 2015, pp. 120–177*
3. Ječmen, I. *Sufiksno stablo i sufiksni niz, master rad* http://elibrary.matf.bg.ac.rs/bitstream/handle/123456789/1859/Master%20rad.pdf?sequence=1. (septembar 2022.)
4. Schwarz, K. *materijali za kurs Data Structures, Stanford University, Aho-Corasick Automata* http://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/02/Small02.pdf. (septembar 2022.)
5. S. Arudchutha, T. N. & Ragel, R. G. *String matching with multicore CPUs: Performing better with the Aho-Corasick algorithm, 2013 IEEE 8th International Conference on Industrial and Information Systems, 2013, pp. 231-236, doi: 10.1109/ICIInfS.2013.6731987.* https://arxiv.org/ftp/arxiv/papers/1403/1403.1305.pdf. (septembar 2022.)
6. Ukkonen, E. *On-line construction of suffix trees. Algorithmica 14, 249–260 (1995). https://doi.org/10.1007/BF01206331.* https://www.cs.helsinki.fi/u/ukkonen/SuffixT1.pdf. (septembar 2022.)
7. Gusfield, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge: Cambridge University Press. (1997). doi:10.1017/CBO9780511574931.*
8. *Java dokumentacija* https://docs.oracle.com/en/java/. (septembar 2022.)
9. *Spring dokumentacija* https://spring.io/. (septembar 2022.)

10. *React dokumentacija* https://reactjs.org/docs/getting-started.html. (septembar 2022.)

11. *Node.js dokumentacija* https://nodejs.org/en/. (septembar 2022.)

12. *Github repozitorijum sa source code-om* https://github.com/FilipMiljakovic/master_bwt. (septembar 2022.)

13. *Docker dokumentacija* https://docs.docker.com/. (septembar 2022.)

14. *Git download* https://git-scm.com/downloads. (septembar 2022.)

Biografija autora

Filip Miljaković, (*Aleksandrovac Župski, 10. jul 1994.*). Srednju školu „Sveti Trifun" završava u Aleksandrovcu 2013. godine i upisuje Matematički fakultet u Beogradu, smer Računarstvo i informatika. Osnovne studije završava 2017. godine sa prosekom 8.23 i upisuje master na istom smeru. Od 2017. godine je zaposlen kao Software developer (**Enjoy.ing** - decembar 2017.- oktobar 2020. i **Optimal systems d.o.o.** - oktobar 2020. do sada). Projekti na kojima je radio su uglavnom bili zasnovani na veb tehnologijama (*eBanking* industrija i *Document management* sistemi), a osnovni programski jezik u kojem je rađeno je *Java*.