Izračun polja najdužih zajedničkih prefiksa korištenjem Burrows-Wheelerove transformacije

Studentski tim: Daria Bužić

Filip Kozjak

Ivana Vanjak

Nastavnik: Mirjana Domazet-Lošo

Sadržaj

Dokumentacija

1. Opis algoritma

1.1 Pomoćne strukture podataka

1.1.1 Sufiksno polje

Sufiksno polje (*eng. suffix array*) *SA* za niz znakova *S* je polje cijelih brojeva u rangu od *1* do *n*, gdje je n duljina zadanog niza znakova. Polje sadrži početna mjesta (indekse) abecedno sortiranih sufiksa niza *S*. Cijeli brojevi u sufiksnom polju određuju leksikografski poredak *n* sufiksa niza *S*.

Prilikom izgradnje sufiksnog polja, u postojeću abecedu dodajemo znak *$(*sentinel), koji predstavlja najmanji znak abecede, te ga dodajemo na kraj ulaznog niza. Svakom sufiksu novog niza, počevši od najduljeg, pridružujemo indekse od *1* do *n*. Tako najdulji sufiks, odnosno cijeli niz, ima indeks *1*, a *$* ima indeks *n*. Sufiksi niza se zatim abecedno poredaju. Sufiksno polje sadrži početne indekse sufiksa, poredane prema abecednom poretku.

1.1.2 Burrows-Wheelerova transformacija

Burrows-Wheelerova transformacija pretvara ulazni niz *S* u niz $BWT[1..n]$ gdje se svaki element računa kao $BWT[i] = S[SA[i]-1]$ za svaki *i* za koji je $SA[i] \neq 1$, za $SA[i]=1$ $BWT[i]=\$$.

1.1.3 Binarno stablo valića

Za objašnjenje podatkovne strukture stabla valića (*engl. wavelet tree*), potrebna nam je poredana abeceda Σ kao polje veličine σ takvo da su znakovi poredani u uzlaznom redu u polju $\Sigma[1..\sigma]$, tj. $\Sigma[1] = \$ < \Sigma[2] < \cdots < \Sigma[\sigma]$.

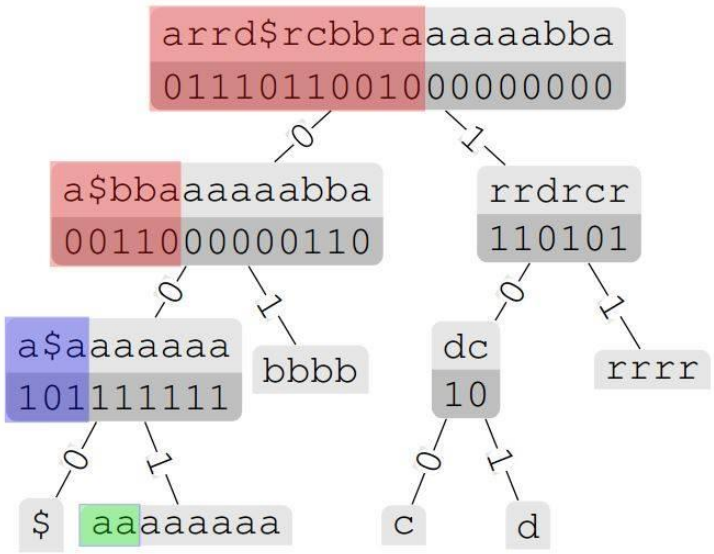
Za interval $[l..r]$ kažemo da je abecedni interval (*alphabet interval*) ako je podinterval od $[1..\sigma]$. Za abecedni interval $[l..r]$ znakovni niz $BWT^{[l..r]}$ dobiven je Burrows-Wheelerovim transformiranim znakovnim nizom (BWT) iz znakovnog niza S, brisanjem svih znakova u BWT koji ne pripadaju podabecedi $\Sigma[l..r]$ od $\Sigma[1..\sigma]$.

Stablo valića od znakovnog niza BWT nad abecedom $\Sigma[1..\sigma]$ je balansirano binarno stablo. Svaki čvor stabla *v* odgovara znakovnom nizu $BWT^{[l..r]}$, gdje je *[l..r]* abecedni interval. Korijen stabla valića odgovara znakovnom nizu $BWT^{[1..\sigma]}$. Ako *l*=*r* onda *v* nema djece, inače *v* ima dvoje djece: lijevo dijete odgovara znakovnom nizu $BWT^{[l..m]}$ a desno dijete odgovara znakovnom nizu $BWT^{[m+1..r]}$, gdje je $m = \lfloor l + r \rfloor / 2$. U ovom slučaju *v* sadrži bit vektor $B^{[l..r]}$ čiji je *i*-ti ulaz *0* ako *i*-ti znak iz $BWT^{[l..r]}$ pripada podabecedi $\Sigma[l..m]$, a *1* ako pripada podabecedi $\Sigma[m + 1..r]$. Odnosno, vrijednost u bit vektoru bit će postavljena na *0* ako znak pripada lijevom podstablu i obrnuto; vrijednost će biti postavljena na *1* ako znak pripada desnom podstablu.
Stablo valića podržava pretraživanje unatrag nad nizom znakova.

- **Pretraživanje unatrag(*eng. backward search*):** Σ predstavlja poredanu abecedu veličine σ čiji je najmanji element *$* (sentinel) postavljen na kraj. Znakovni niz se transformira Burrows Wheelerovom transformacijom u niz *BWT[1..n]* . Neka je $c \in \Sigma$ i ω podniz niza *S*. Za dani ω-interval *[i.. j]* u sufiksnom polju *SA* (ω je prefiks $S_{SA[k]}$ za svaki $i \leq k \leq j$ ali nije prefiks bilo kojem drugom sufiksu od *S*). Pretraga unatrag (*backwardSearch(c, [i.. j]*)vraća $c\omega$-interval $[C[c] + Occ(c, i - 1) + 1 .. C[c] + Occ(c, j)]$, gdje je *C[c]* ukupan broj pojavljivanja znakova u *S* koji su strogo manji od *c*, a *Occ(c,i)* je broj pojavljivanja znaka znaka *c* u *BWT[1..i]*.

Stablo valića efikasno određuje rang nekog znaka do određenog indeks pomoću funkcije *rank.* Ta funkcija određuje broj pojavljivanja znaka prije određenog indeksa. Pomoću bit vektora *B* upiti $rank_0(B, i)$ i $rank_1(B, i)$ mogu biti izračunati u konstantnom vremenu gdje je $rank_b$ broj pojavljivanja bita *b* u *B[1..i]*. Kad se rang funkcije računa direktno iz znakovnog niza to je višestruko složenije i sporije.

- Primjer izračuna ranga za Sliku 1: *rank*(11, a, WT) = *rank*(*rank*(*rank*(11, 0, b_ϵ) = 5, 0, b_0) = 3, 1, b_{00}) = 2



Slika 1 – Izračunavanje funkcije *rank(11, a) = 2*

1.2 Izračun polja najdužih zajedničkih prefiksa

Polje najdužih zajedničkih prefiksa (*engl. Longest common prefix - LCP*) je polje koje sadrži duljinu najduljeg zajedničkog prefiksa između svakog para uzastopnih sufiksa u sufiksnom polju. Formalna definicija LCP polja je: $LCP[1] = -1$, $LCP[n + 1] = -1$ i $LCP[i] = |lcp(S_{SA[i-1]}, S_{SA[i]})|$ za $2 \leq i \leq n$, gdje $lcp(u, v)$ označava najdulji zajednički prefiks između znakovnih nizova u i v.

Postoji više algoritama za konstrukciju LCP polja u linearnom vremenu. Ti algoritmi prvo stvore sufiksno polje i potom iz njega dobivaju LCP polje u linearnom vremenu. Nedostatak takvih algoritama je velika potrošnja memorije i dugo vrijeme izvođenja. Zbog velikog broja podataka koje je potrebno analizirati javlja se potreba za novim pristupima i strukturama koje će ubrzati cjelokupni proces. U daljnjem tekstu bit će opisan algoritam koji radi direktno nad Burrows-Wheelrovim transformiranim nizom znakova za razliku od dosadašnjih algoritama koji koriste sufiksno polje.

Za konstrukciju LCP polja koriste se dva algoritma. Drugi algoritam konstruira LCP-polje pomoću metode *getIntervals* koja je opisana u prvom algoritmu. Prvi algoritam radi nad stablom valića koje je konstruirano iz Burrows-Wheelerovog transformiranog niza znakova. Također, koristi funkciju *rank*, koja rekurzivno prolazi po stablu i tako broji pojavljivanje određenog znaka od početka niza do određenog indeksa.

1.2.1 Algoritam 1

```
getIntervals([i..j])
  list ← [ ]
  getIntervals'([i..j], [1..σ], list)
  return list

getIntervals'([i..j], [l..r], list)
  if l = r then
    c ← Σ[l]
    add(list, [C[c] + i..C[c] + j])
  else
    (a0, b0) ← (rank0(B[l..r], i − 1), rank0(B[l..r], j))
    (a1, b1) ← (i − 1 − a0, j − b0)
    m ← ⌊(l+r)/2⌋
    if b0 > a0 then
      getIntervals'([a0 + 1..b0], [l..m], list)
    if b1 > a1 then
      getIntervals'([a1 + 1..b1], [m + 1..r], list)
```

Za interval *[i..j]* procedura *getIntervals([i.. j])* vraća listu *cω* intervala. Počinje s intervalom *[i..j]* u korijenu stabla i obilazi stablo grananjem u dubinu.
Na trenutnom čvoru *v*, algoritam koristi upite za dobivanje ranga kako bi se dobio broj $b_0 - a_0$ koji predstavlja broj nula u bitvektoru čvora *v* u trenutnom intervalu. Ako je $b_0 > a_0$ onda postoje znakovi u *BWT[i..j]* koji pripadaju lijevom podstablu čvora v i algoritam nastavlja rekurzivnim pozivom nad lijevim djetetom čvora v. Ako je je broj pozitivan, odnosno ako je $b_1 > a_1$, onda se rekurzivni poziv vrši nad desnim djetetom. Algoritam se nastavlja do dna stabla, odnosno do listova koji predstavljaju znakove. Ako se list koji odgovara znaku *c* nalazi u intervalu *[p..q]* tada je interval *[C[c] +p..C[c] +q] cω* interval. Na ovaj način algoritam izračunava listu *cω* intervala.

1.2.2 Algoritam 2

```
initialize the array LCP[1..n + 1] /* i.e., LCP[i] = ⊥ for all 1 ⩽ i ⩽ n + 1 */
LCP[1] ← −1; LCP[n + 1] ← −1
initialize an empty queue
enqueue(⟨[1..n], 0⟩)
while queue is not empty do
  ⟨[i..j], ℓ⟩ ← dequeue()
  list ← getIntervals([i..j])
  for each [lb..rb] in list do
    if LCP[rb + 1] = ⊥ then
      enqueue(⟨[lb..rb], ℓ + 1⟩)
      LCP[rb + 1] ← ℓ
```

Algoritam 2 se bazira na implementaciji algoritma 1. Algoritmu 1 kao ulazni parametar daje se cijela lista a on vraća listu ω intervala. Za svaki interval [*ik*.. *jk*], *LCP*[*jk* +1] je postavljan na 0. jedino je posljednji *LCP[20] = -1*. Dobiveni intervali se stave u podatkovnu strukturu red i svakom se pridruži *l=0* vrijednost. Algoritam skida svaki interval iz reda i poziva metodu *getIntervals* koja za dani interval ponovo vraća listu intervala koji se stavljaju u red. Algoritam nastavlja s radom dok se red ne isprazni te prilikom rada mijenja vrijednost *l* te ju pridružuje konačnom polju *LCP*. Rezultat algoritma je *LCP* polje.

2. Objašnjenje algoritma na primjeru

Algoritam ćemo objasniti na primjeru ulaznog niza S_u = annasanannas.
U našem radu, korištena je pomoćna knjižnica sdsl-light autora Simona Goga.

2.1 Izgradnja sufiksnog polja

Sufiksno polje za ulazni niz *S* izgrađeno je korištenjem sdsl-light knjižnice pomoću strukture za izgradnju sufiksnog polja. Rezultat je je polje SA = [13, 6, 8, 1, 11, 4, 7, 10, 3, 9, 2, 12, 5].
Sufiksno polje gradi se na način opisan u poglavlju 1.1. ovog rada. Svakom sufiksu niza, počevši od najduljeg, pridružujemo indekse od *1* do *n*. Sufiksi niza leksikografski se poredaju. Sufiksno polje sadrži početne indekse sufiksa, poredane prema abecednom poretku. U nastavku ćemo opisati izgradnju sufiksnog polja.

1. Na kraj ulaznog niza *S* dodaje se znak *$* koji predstavlja najmanji znak abecede.

   S = annasanannas$

2. Svakom sufiksu niza S, počevši od najduljeg, pridružujemo indekse od *1* do *n*.

i	**$S_{SA}[i]$**
1	annasanannas$
2	nnasanannas$
3	nasanannas$
4	asanannas$
5	sanannas$
6	anannas$
7	nannas$
8	annas$
9	nnas$
10	nas$
11	as$
12	s$
13	$
14	

Tablica 1 – Svi sufiksi ulaznog niza s pridruženim indeksima

3. Sufiksi niza se leksikografski poredaj, te takav poredak početnih indeksa predstavlja sufiksno polje SA.

i	SA[i]	$S_{SA}[i]$
1	**13**	$
2	**6**	anannas$
3	**8**	annas$
4	**1**	annasanannas$
5	**11**	as$
6	**4**	asanannas$
7	**7**	nannas$
8	**10**	nas$
9	**3**	nasanannas$
10	**9**	nnas$
11	**2**	nnasanannas$
12	**12**	s$
13	**5**	sanannas$
14		

Tablica 2 – Svi sufiksi ulaznog niza s pridruženim indeksima i izračunato sufiksno polje SA

2.2 Izgradnja Burows-Wheelerovog transformiranog niza

Iz izgrađenog sufiksnog polja, dobivamo Burrows-Wheelerov niz BWT[1..n] u kojem se svaki element računa po formuli *BWT[i] = S[SA[i]-1]* za svaki *i* za koji je *SA[i]≠1*, za *SA[i]=1 BWT[i]=$*.

- **Primjer:**
  BWT[10] = S[SA[10]-1] = S[9-1] = S[8] = a
  Rezultat je osmi element u polju S = annasanannas$, a to je znak a.

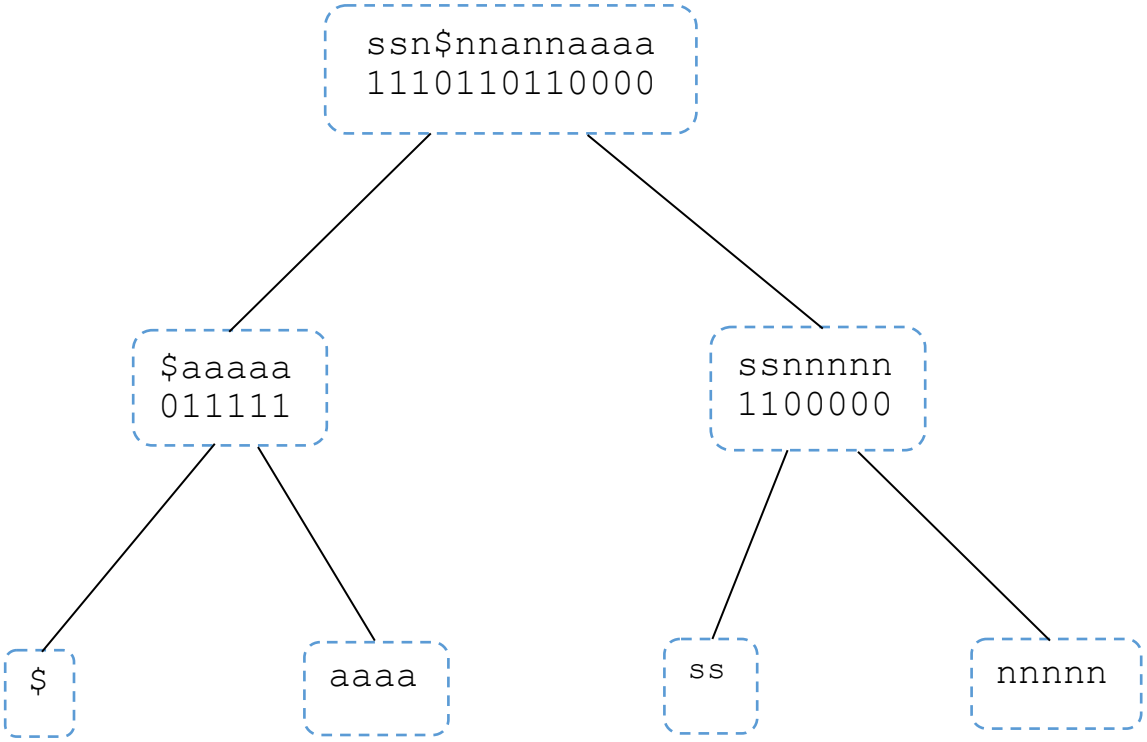
i	SA[i]	BWT[i]	$S_{SA}[i]$
1	**13**	**s**	$
2	**6**	**s**	anannas$
3	**8**	n	annas$
4	**1**	$	annasanannas$
5	**11**	n	as$
6	**4**	n	asanannas$
7	**7**	a	nannas$
8	**10**	n	nas$
9	**3**	n	nasanannas$
10	**9**	a	nnas$
11	**2**	a	nnasanannas$
12	**12**	a	s$
13	**5**	a	sanannas$
14			

Tablica 3 - Svi sufiksi ulaznog niza s pridruženim indeksima, izračunato sufiksno polje SA i BWT

2.3 Izgradnja binarnog stabla valića

Iz Burrows-Wheelerovog transformiranog niza gradi se stablo valića pomoću sdsl-light knjižnice.

Imamo poredanu abecedu Σ[1..4] = $ans. U korijen stabla valića stavlja se bitvektor dobiven iz niza BWT. Bitvektor se dobiva tako da se poredana abeceda podijeli na pola: Σ[1..2] = $a i Σ[3..4] = ns. Znakovima u BWT nizu koji su u prvoj polovini pridružuje se vrijednost 0, a ostalima 1. Stablo se dalje grana tako da svi znakovi koji imaju vrijednost 0 idu u lijevu granu, a ostali u desnu. Takav se postupak ponavlja dok se u čvoru ne nalaze samo isti znakovi. Formalni postupak izgradnje stabla valića opisan je u odlomku 1.3.



Slika 2 – Binarno stablo valića za ulazni niz za BWT = ssn$nnannaaaa

2.4 Implementacija polja najdužih zajedničkih prefiksa

Implementacija započinje inicijalizacijom polja LCP[1 .. n+1]. Sve vrijednosti polja postavljaju se na neku nevažeću vrijednost osim prvog i posljednjeg elementa koji se postavljaju na -1; LCP[1] = -1, LCP[n+1] = -1. Nakon toga inicijalizira se prazan red Q koji radi na principu FIFO (eng. *first in first out*). Na početku u red stavljamo strukturu koja sadrži početni interval i broj *l* = 0. Sve dok red nije prazan, algoritam skidan jednu po jednu strukturu iz reda. Za svaki interval iz strukture računaju se *cω* intervali.

LPC =

-1	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	-1

Q =

<[1..13], 0>

Iz reda skidamo najstariji element i za njega računamo listu *cω* intervala pomoću funkcije *getIntervals*.

Funkcija *getIntervals* prima određeni interval [*i..j*] i vraća listu *cω* intervala. Ona radi nad stablom valića i koristi njegovu funkciju *rank*, koja rekurzivno obilazi stablo računajući broj pojavljivanja znaka prije određenog indeksa, tj. računa rang tog znaka.

Indeks početka intervala dobiva se po formuli *rank(c, i-1) + C[c] + 1.*
Indeks kraja intervala dobiva se po formuli *C[c]+rank(c, j).*
c – određeni znak
C[c] – zbroj rangova svih elemenata iz poredane abecede koji su leksikografski manji od znaka c
i – početak intervala
j – kraj intervala
rank(a, k) – broj pojavljivanja znaka a do indeksa polja k

- **Na primjeru ćemo pokazati kako radi funkcija *getIntervals* za početni interval [*i..j*]=[1..13].**

  Prvo se pronađu svi jedinstveni znakovi iz abecede koji se nalaze u zadanom intervalu, te se poredaju leksikografski. Konačno ćemo imati onoliko *cω* intervala koliko je jedinstvenih znakova.
  Za svaki znak se računa njegov *cω* interval.
  U intervalu [1..13] nalaze se svi znakovi abecede $, a, n, s, što znači da ćemo imati četiri *cω* intervala kao rezultat. Idemo po svakom znaku i računamo njegov interval po formuli navedenoj iznad.

C = [0, 1, 6, 11]

Za znak $ dobiveni interval biti će [*rank(c, i-1) + C[c] + 1 .. C[c]+rank(c, j)*] = [*rank('$', 0)*] + *C['$'] + 1 .. C['$']+rank('$', 13)*] = [0+0+1 .. 0+1] = [1..1].

Za znak *a* dobiveni interval biti će [*rank(c, i-1) + C[c] + 1 .. C[c]+rank(c, j)*] = [*rank('a', 0)*] + *C['a'] + 1 .. C['a']+rank('a', 13)*] = [0+1+1 .. 1+5] = [2..6].

Za znak *n* dobiveni interval biti će [*rank(c, i-1) + C[c] + 1 .. C[c]+rank(c, j)*] = [*rank('n', 0)*] + *C['n'] + 1 .. C['n']+rank('n', 13)*] = [0+6+1 .. 6+5] = [7..11].

Za znak *s* dobiveni interval biti će [*rank(c, i-1) + C[c] + 1 .. C[c]+rank(c, j)*] = [*rank('s', 0)*] + *C['s'] + 1 .. C['s']+rank('s', 13)*] = [0+11+1 .. 11+2] = [12..13].

I tako za svaki znak. Konačno funkcija *getIntervals* vraća *cω* intervale [1..1],[2..6],[7..11],[12..13].

Za svaki interval [*lb .. rb*] iz liste *cω* intervala provjeravamo vrijednost LCP polja na indeksu *rb+1*. Ako je vrijednost tog polja još uvijek nevažeća, u red stavljamo strukturu [<[*lb..rb*]>, *l+1*], te se na isti indeks polja stavlja vrijednost *l*.

U nastavku su prikazat ćemo izgled reda Q i LCP polja za četiri dobivena *cω* intervala.

LPC =	-1	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	-1

Q =
<[1..13], 0>

1) [*lb..rb*] = [1..1], *l=0*

Provjeravamo LCP[*rb*+1] = LCP[2] = ⊥. U red stavljamo strukturu <[*lb..rb*], *l+1*> = <[*1..1*], 1>, te LCP[*rb+1*]= LCP[2] = 0

LPC =	-1	0	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	-1

Q =
<[1..1], 1>

2) [*lb..rb*] = [2..6], *l=0*

Provjeravamo LCP[*rb*+1] = LCP[7] = ⊥. U red stavljamo strukturu <[*lb..rb*], *l+1*> = <[*2..6*], 1>, te LCP[*rb+1*]= LCP[7] = 0

LPC =	-1	0	⊥	⊥	⊥	⊥	0	⊥	⊥	⊥	⊥	⊥	⊥	-1

Q =
<[2..6], 1>
<[1..1], 1>

3) [*lb..rb*] = [7..11], *l=0*

Provjeravamo LCP[*rb*+1] = LCP[12] = ⊥. U red stavljamo strukturu <[*lb..rb*], *l+1*> = <[*7..11*], 1>, te LCP[*rb+1*]= LCP[12] = 0

LPC =

-1	0	⊥	⊥	⊥	⊥	0	⊥	⊥	⊥	⊥	0	⊥	-1

Q =

<[7..11], 1>
<[2..6], 1>
<[1..1], 1>

4) [*lb..rb*] = [12..13], *l=0*

Provjeravamo LCP[*rb*+1] = LCP[14] = -1. U ovom slučaju preskačemo dodavanje u red i LCP polje jer je vrijednost LCP[14] već postavljena na važeću vrijednost.

LPC =

-1	0	⊥	⊥	⊥	⊥	0	⊥	⊥	⊥	⊥	0	⊥	-1

Q=

<[7..11], 1>
<[2..6], 1>
<[1..1], 1>

Algoritam nastavlja s radom sve dok ima elemenata u redu Q. Konačan izgled LCP polja dan je u nastavku.

LPC =

-1	0	2	5	1	2	0	2	3	1	4	0	1	-1

3. Rezultati testiranja

Testiranja smo proveli na različitim sintetskim testnim podacima s različitim duljinama znakova. Podaci su generirani iz bakterije ecoli. Uzeti su različiti podskupovi te bakterije i nad njima provedena mjerenja.

3.1 Analiza točnosti

Točnost podataka provjerili smo usporedbom izlaza vlastitih implementacija i originalne implementacije Simona Goga. Izlazni niz znakova jednak je u obje implementacije. Implementacija Simona Goga ima drugačiji format. Njegov izlaz na početku postavlja dvije nule dok mi na početak i kraj niza stavljamo -1. Ostatak izlaznog niza u potpunosti se podudara.

3.2 Analiza vremena izvođenja

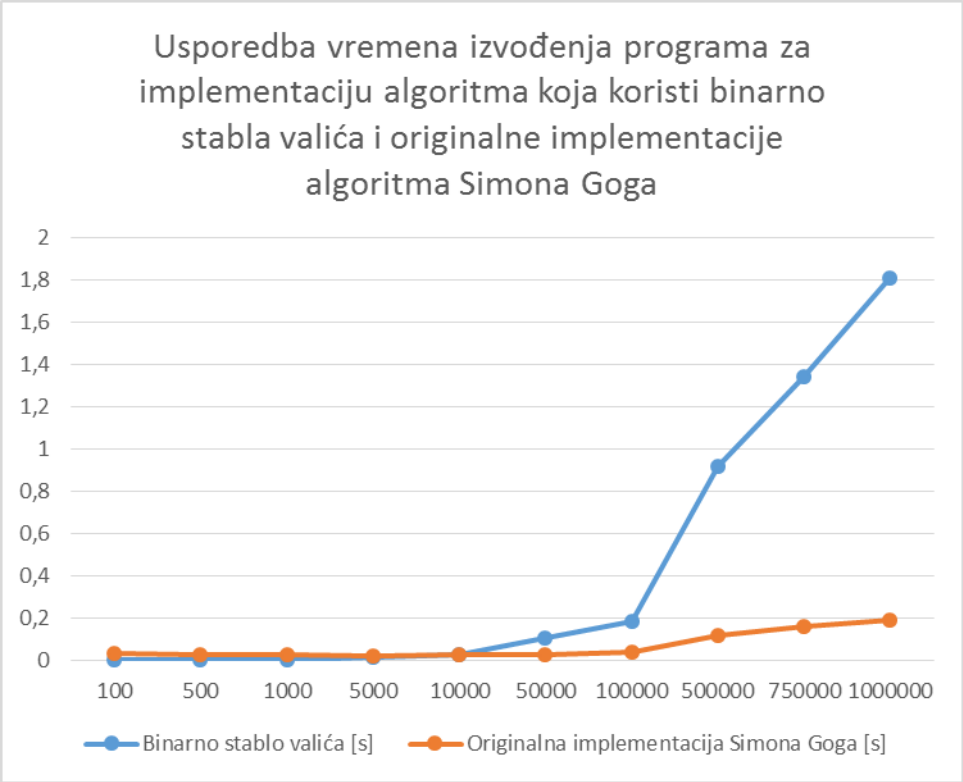
U analizi vremena izvođenja uspoređivali smo:

- vlastitu implementaciju algoritma uz korištenje implementacije binarnog stabla valića Simona Goga
- vlastitu implementaciju algoritma bez stabla valića (vlastita implementacija funkcije *rank*)
- vlastitu implementaciju algoritma uz korištenje n-arnog stabla valića koje su implementirali studenti na predmetu Bioinformatika na FER-u
- originalnu implementaciju Simona Goga.

Broj znakova	Binarno stablo valića [s]	5-arno stablo valića [s]	10-arno stablo valića [s]	20-arno stablo valića [s]	Bez stabla valića [s]	Originalna implementacija Simona Goga [s]
100	0,002105	0,003058	0,001347	0,005476	0,000219	0,031202
500	0,001839	0,013365	0,009989	0,00636	0,0026	0,021912
1000	0,002683	0,057796	0,03267	0,018455	0,009632	0,021881
5000	0,011742	0,800149	0,633978	0,344578	0,197036	0,020889
10000	0,023956	2,61539	1,96101	1,29077	0,735055	0,021431
50000	0,104976	71,8259	50,914	30,8301	18,1147	0,02451
100000	0,182855	367,628	237,741	128,854	75,6483	0,036276
500000	0,914995	>2000	>1900	1821,84	2158,55	0,114948
750000	1,34068	-	-	>2000	4856,42	0,158044
1000000	1,80493	-	-	-	7056,18	0,189965

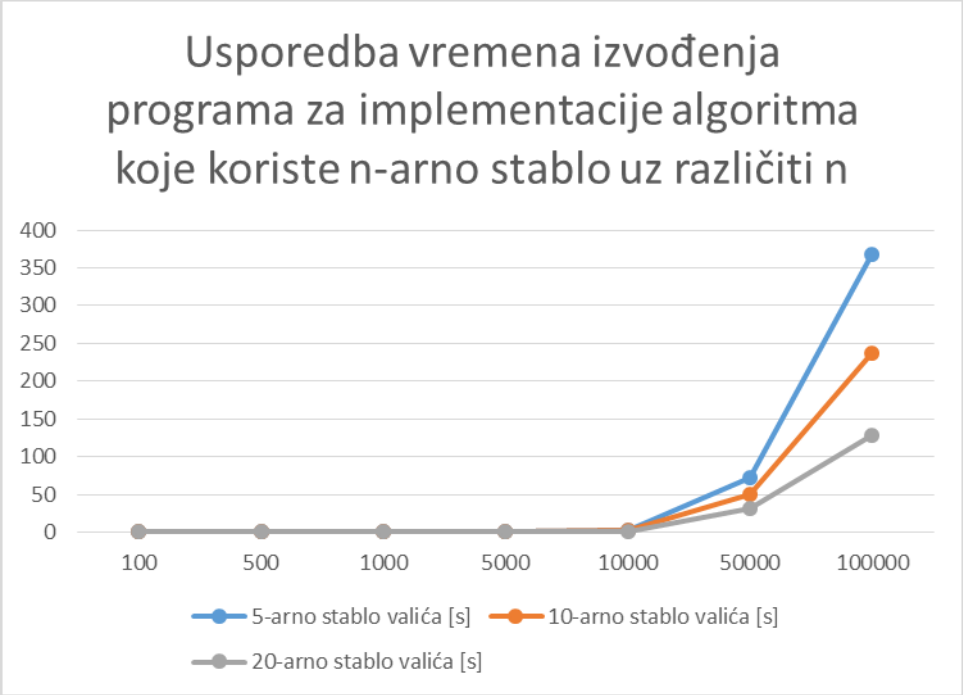
Tablica 4 – Vremena izvođenja

Iz prethodne tablice možemo vidjeti kako se vrijeme izvođenja povećava kako se povećavaju duljine ulaznih nizova. Implementacije se razlikuju u vremenima izvođenja. U našem slučaju je ispalo da je implementacija bez stabla valića brža od implementacije s n-arnim stablom. Očekivano je da bude obrnuto. No, pošto je za n-arno stablo preuzeta studentska implementacija, možemo pretpostaviti da bi se za neku drugu implementaciju n-arnog stabla dobili bolji rezultati. Za najveće ulazne nizove testiranja se nisu provela do kraja jer su izvođenja programa predugo trajala. Zbog toga ćemo u nastavku posebno uspoređivati brzinu izvođenja algoritma za n-arno stablo s obzirom na n-arnost stabla (broj grana koji izlazi iz nekog čvora), a posebno implementaciju s binarnim stablom valića i originalnu implementaciju Simona Goga.



Slika 3 – Grafički prikaz usporedbi vremena izvođenja algoritma uz korištenje binarnog stabla valića i originalne implementacije algoritma

Iz priloženog grafa možemo vidjeti kako se naša i originalna implementacija Simona Goga podudaraju u vremenima izvođenja za relativno male ulazne nizove. Za velike ulazne nizove vrijeme izvođenja kod naše implementacije eksponencijalno raste.



Slika 4 – Grafički prikaz usporedbi vremena izvođenja algoritma za n-arno stablo uz različiti n

Iz priloženog grafa se vidi kako se u početku vremena izvođenja podudaraju, no za veće ulazne nizove razlika u brzini izvođenja se povećava. Brži su programi koji koriste stablo veće n-arnosti.

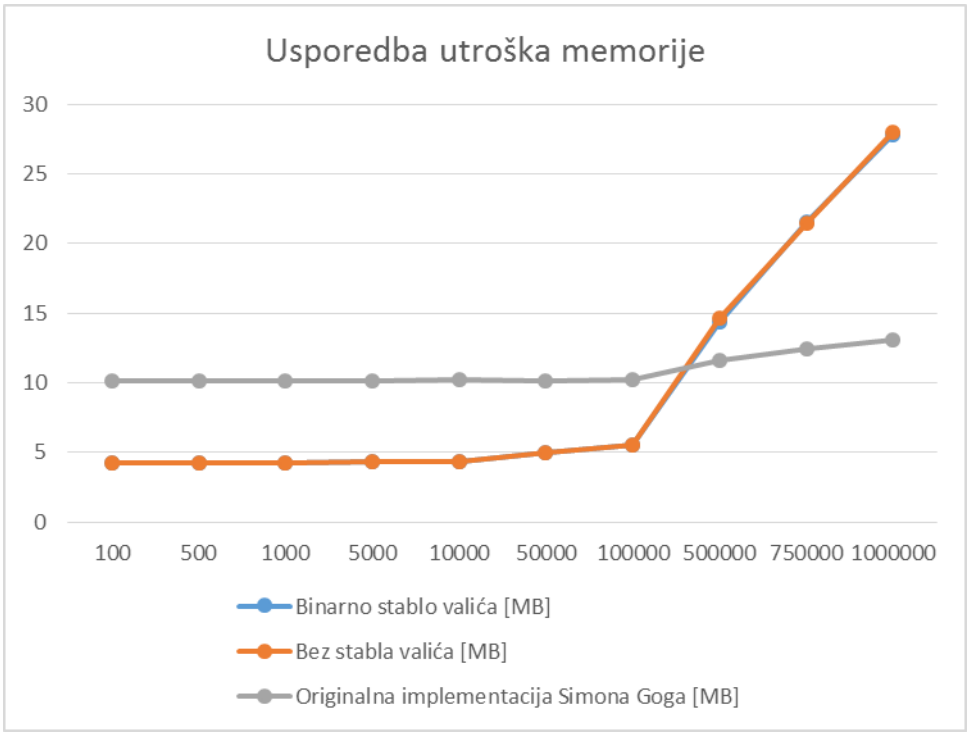
3.3 Analiza zauzeća memorije

Za analizu zauzeća memorije korištene su iste implementacije kao i kod analize vremena izvođenja.

Broj znakova	Binarno stablo valića [MB]	5-arno stablo valića [MB]	10-arno stablo valića [MB]	20-arno stablo valića [MB]	Bez stabla valića [MB]	Originalna implementacija Simona Goga [MB]
100	4,26562	4,27734	4,27734	4,27734	4,26562	10,1133
500	4,26953	4,27344	4,27734	4,27734	4,26562	10,1172
1000	4,26953	4,28125	4,28125	4,28125	4,26953	10,1133
5000	4,32031	4,33203	4,33203	4,33203	4,32031	10,1289
10000	4,37891	4,39062	4,39062	4,39062	4,37891	10,2109
50000	5,00781	9,78516	7,71875	6,42969	5,00781	10,0898
100000	5,53906	17,9766	14,3672	11,5352	5,54297	10,2148
500000	14,3828	-	-	14,3828	14,6914	11,6484
750000	21,5977	-	-	-	21,4414	12,4727
1000000	27,7852	-	-	-	28,0321	13,0586

Tablica 5 – Utrošak memorije

Kod mjerenja utroška memorije možemo vidjeti kako se zauzeće memorije povećava kako se povećava ulazni niz. U svim našim implementacijama zauzeće memorije za male nizove je dosta malo, ali pri većim duljinama niza eksponencijalno raste. Kod originalne implementacije Simona Goga zauzeće memorije polako počinje rasti kod većih duljina niza. Naše implementacije u početku zauzimaju manje memorije od originalne, no kod većih nizove originalna implementacija ima uvjerljivo bolje performanse što se tiče zauzeća memorije. Što se tiče n-arnog stabla, zauzeće memorije se smanjuje s povećanjem n-arnosti.



Slika 5 – Grafički prikaz usporedbi memorijskog zauzeća različitih programa

Možemo vidjeti kako se obje naše implementacije gotovo potpuno podudaraju. Implementacija Simona Goga je puno bolja za veće ulazne nizove.

4. Zaključak

Brojne analize nizova znakova zahtijevaju izračun polja najduljeg zajedničkog prefiksa. Implementacije koje ne uključuju strukturu stabla valića nisu primjenjive za nizove velikih duljina jer su vremenski prezahtjevne. Također, implementacija algoritma uz korištenje studentske implementacije n-arnog stabla valića također treba previše vremena za relativno dugačke nizove. Povećanjem n-arnosti stabla dobivaju se bolji rezultati, no još uvijek ne dovoljno dobri. Uz vlastitu implementaciju izračuna polja najdužih zajedničkih prefiksa korištenjem Burrows-Wheelerove transformacije, te binarnog stabla valića Simona Goga ostvarili smo implementaciju koja daje prihvatljive rezultate i za velike nizove podataka. U usporedbi sa originalnom implementacijom Simona Goga vidi se razlika kako u vremenu izvođenja tako i u utrošku memorije. Naša implementacija algoritma je kod relativno dugačkih nizova još uvijek lošija od implementacije Simona Goga u svim segmentima.

5. Literatura

[1] Simon Gog: sdsl Cheat Sheet, http://simongog.github.io/assets/data/sdsl-cheatsheet.pdf

[2] Simon Gog: Succinct Data Structures: From Theory to Practice, http://www.lirmm.fr/mastodons/talks/Gog-indexing-2014.pdf

[3] Simon Gog: sdsl-lite, https://github.com/simongog/sdsl-lite

[4] Simon Gog: Succinct Data Structures, http://simongog.github.io/assets/data/sdsl-slides/tutorial#2

[5] Timo Beller, Simon Gog, Enno Ohlebusch, Thomas Schnattinger: Computing the longest common prefix array based on the Burrows–Wheeler transform, 2013.

[6] Timo Beller, Simon Gog, Enno Ohlebusch, Thomas Schnattinger: Computing the Longest Common Prefix Array Based on the Burrows-Wheeler Transform – Slides 2013.

[4] Kristijan Ivančić, implementacija n-arnog stabla valića u programskom jeziku C++, https://github.com/marijanaSpajic/multiary_wavelet_tree/tree/master/C%2B%2B