SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DOKUMENTACIJA

Računanje najduljeg zajedničkog prefiksa temeljeno na BWT

Silvestar Badak, Tomislav Gudelj, Domagoj Vukadin
Voditelj: *doc. dr. sc. Mirjana Domazet-Lošo*

Zagreb, siječanj 2017.

SADRŽAJ

1. Uvod

Bioinformatika je grana računarstva koja proučava specifične algoritme i strukture podataka kako bi ih mogli primijeniti u području biologije živih bića. Gen je osnovna jedinica nasljeđivanja u živim organizmima. Sastavni je dio DNA i sastoji se od niza nukleotida (*dušična baza - A,C,G,T, šećera i fosfatne skupine*). Kodirana mRNA dobivena iz gena procesom transkripcije služi za sintezu proteina (*procesom translacije*) koji se sastoje od niza aminokiselina. Proteini obavljaju mnogobrojne funkcije unutar živih bića (*npr. ubrzavaju metaboličke reakcije, služe za transport molekula, umnažanje i prepisivanje DNA, itd.*). Jedan od najpoznatijih proteina je hemoglobin koji je odgovoran za prijenos atoma kisika u krv.[3]

Prilikom procesa razmnožavanja živih bića, jedna mala mutacija, tj. promjena jednog nukleotida može rezultirati ozbiljnim posljedicama i obilježiti cijeli životni vijek nastalog organizma. Neke se bolesti uzrokovane mutacijom gena i naslijeđenim genima mogu predvidjeti iz ljudske DNA, koja je jedinstveni identifikator svakog organizma. Sekvenciranjem genoma, skupa svih gena nekog organizma, dobivamo sveukupnu nasljednu informaciju organizma. Genomi su izrazito veliki i nemoguće ih je analizirati bez korištenja računalnih programa, tj. strojeva za analizu. Algoritmi i strukture podataka koji su proizašli iz bioinformatičke znanosti sastavni su dijelovi tih programa i omogućavaju nam da u razumnom vremenu dobijemo informaciju koju želimo. Jedan od takvih programa koji se koristi u analizi genoma je **računanje najduljeg zajedničkog prefiksa temeljeno na Burrows-Wheelerovoj transformaciji** što je tema ovog projekta. U nastavku slijedi opis korištenih algoritama i struktura podataka.

2. Strukture podataka i algoritmi

Računanje najduljeg zajedničkog prefiksa izračunatih sufiksa ulaznog niza - LCP u razumnom vremenu može se obaviti na sljedeći način:

1. određivanjem **sufiksnog polja** SA ulaznog niza S
2. računanjem **Burrows-Wheelerove transformacije - BWT** koristeći SA
3. izgradnjom **binarnog stabla valića** nad novim nizom $S' = BWT(S)$
4. implementacijom funkcije **rang** nad binarnim stablom valića koja se koristi u **algoritmu 1**
5. konstrukcijom **LCP polja algoritmom 2** koji poziva metodu $getIntervals$ **algoritma 1**

2.1. Sufiksno polje

Sufiksno polje je polje indeksa leksikografski poredanih sufiksa nekog niza S. Na primjer, neka je zadan ulazni niz $S = "el\_anele\_lepanelen"$ koji je kao primjer naveden u [1]. Abeceda ovog niza je leksikografski poredan skup znakova sadržan u S - $\sum = \{\$, \_, a, e, l, n, p\}$. Na kraj niza dodajemo jedinstveni znak \$. Leksikografski poredani sufiksi navedenog niza su:

1. $\$$
2. $\_anele\_lepanelen\$$
3. $\_lepanelen\$$
4. $anele\_lepanelen\$$
5. $anelen\$$
6. $e\_lepanelen\$$
7. $el\_anele\_lepanelen\$$
8. $ele\_lepanelen\$$
9. $elen\$$
10. $en\$$
11. $epanelen\$$
12. $l\_anele\_lepanelen\$$
13. $le\_lepanelen\$$
14. $len\$$

15. $lepanelen\$$
16. $n\$$
17. $nele\_lepanelen\$$
18. $nelen\$$
19. $panelen\$$

Sufiksno polje SA je polje indeksa ulaznog niza kojim započinju gore navedeni sufiksi. Slijedno tome vrijedi $SA = [18, 2, 8, 3, 12, 7, 0, 5, 14, 16, 10, 1, 6, 15, 9, 17, 4, 13, 11]$.[1]

2.2. Burrows-Wheelerova transformacija - BWT

Burrows-Wheelerova transformacija je transformacija ulaznog teksta koja rezultira sažimanjem istoga. Može se odrediti na dva načina definirana u [3]:

1. izravno iz ulaznog niza (rotacijom ulaznog niza S)
2. preko sufiksnog polja $SA(S)$

U ovom projektu koristi se drugi način u kojem za ulaz $S = "el\_anele\_lepanelen"$ kao rezultat dobivamo $S' = BWT(S) = "nle\_pl\$nnlleee\_eaae"$ na sljedeći način:

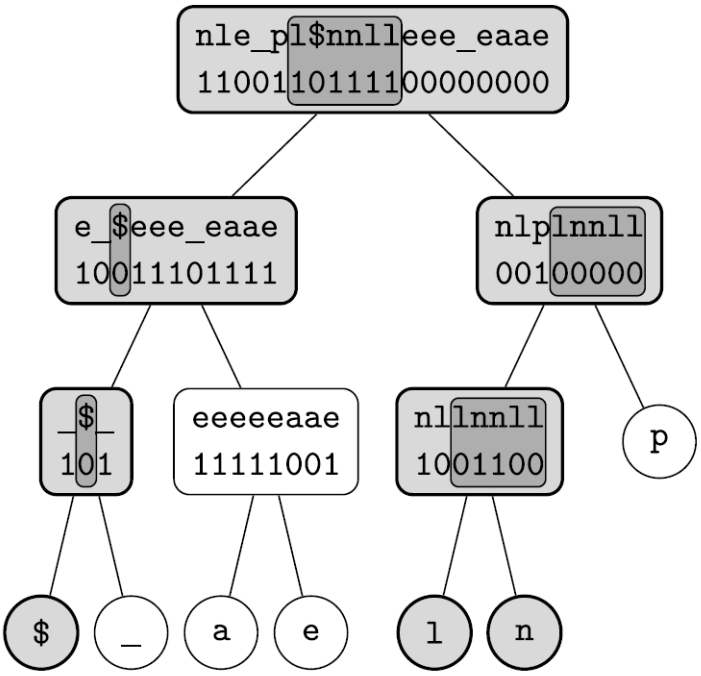
$$BWT[i] = \begin{cases} S[SA[i] - 1], & \text{ako } \forall i(SA[i] \neq 0) \\ \$, & \text{inače} \end{cases} \tag{2.1}$$

2.3. Binarno stablo valića

Binarno stablo valića (*engl. Binary Wavelet Tree*) je struktura podataka novijeg vremena koja nudi rješenja mnogim problemima iz različitih domena kao što su analiza nizova znakova, računalna geometrija i kompresija podataka. Kao ulaz prima niz znakova S koji je sadržan od znakova abecede $\sum$. Omogućava viši stupanj entropije i podržava različite funkcije koje manipuliraju nad ulaznim nizom S značajno brže nego neke klasične funkcije istog izlaza. [4]

Stablo valića podržava tri vrste upita: **pristup** (*engl. access*), **rang** (*engl. rank*) i **odabir** (*engl. select*). Funkcija *access(p)* vraća znak *c* koji se nalazi na poziciji *p*, $rank_p(c)$ broj pojavljivanja znaka *c* do pozicije *p* uključivo, $select_c(o)$ poziciju znaka *c* nakon *o* pojavljivanja. U ovom se projektu koristi samo funkcija rang o kojoj će biti rečeno nešto više u nastavku.[4] Primjer stabla valića dan je na slici 2.1. Za konstrukciju čvorova stabla korišten je pseudokod na slici 2.2.

Svaki čvor sadrži vektor bitova koji predstavlja trenutni niz dobiven iz prethodnog čvora i definiran je abecedom $\sum^{[l..r]}$, gdje l i r predstavljaju indekse početka i završetka podabecede u početnoj abecedi $\sum^{[0..\sigma]}$ koja pripada korijenu stabla i gdje je σ indeks zadnjeg znaka u nizu. Nekom znaku dodjeljujemo vrijednost 0 ako se nalazi u intervalu $[l..m]$ unutar podabecede trenutnog čvora koja sadrži znakove zastupljene samo u tom čvoru, gdje je $m = \frac{\lfloor l+r \rfloor}{2}$, inače 1 ako se pak nalazi u intervalu $[m+1..r]$. Čvorovi se granaju u dva smjera: lijevo - svi znakovi koji su dobili vrijednost 0 i desno - svi znakovi koji su dobili vrijednost 1, zajedno sa svojim podabecedama. Grananje prestaje kad se u čvoru nalazi samo jedan bit (niz istih bitova), odnosno kad je vrijednost tog čvora jednoznačna (samo jedan znak). Taj se čvor naziva list. Listova je onoliko koliko abeceda sadrži znakova.[1]



Slika 2.1: Binarno stablo valića nad nizom $S' = BWT(S) = "nle\_pl\$nnlleee\_eaae"$. Slika preuzeta iz [1].

Algorithm 1 Construction of nodes in the Wavelet Tree

```
function ConstructNode(S, Σ)
    if |Σ| = 1 or |S| = 0 then
        return Self
    end if
    (Σ_left, Σ_right) ← Σ
    SplitChar ← Σ_left[σ_left]
    for all c in S do
        if c > SplitChar then
            S_right.Append(c)
            Self.Bitmap.Append(1)
        else
            S_left.Append(c)
            Self.Bitmap.Append(0)
        end if
    end for
    RightNode ← ConstructNode(S_right, Σ_right)
    LeftNode ← ConstructNode(S_left, Σ_left)
    return Self
end function
```

Slika 2.2: Pseudokod za izgradnju stabla valića. Slika preuzeta iz [4].

Funkcija rang implementirana je koristeći pseudokod na slici 2.3. Rekurzivna je i efikasno računa broj pojavljivanja znaka do nekog indeksa. Uzmemo li za primjer ulazni niz $S' = BWT(S) =$ "$nle\_pl\$nnlleee\_eaae$" te nakon konstrukcije stabla pozovemo *rank$_p$(c)*, vrijednost ćemo dobiti na sljedeći način:

1. provjeravamo kojoj skupini (0 ili 1) pripada dani znak
2. funkcijom BinaryRank računamo broj pojavljivanja dobivene skupine iz 1. do danog indeksa u trenutnom čvoru, odnosno indeks koji tražimo za dijete u koje prelazimo
3. ovisno koju smo skupinu dobili u 1. prelazimo u lijevo, odnosno desno dijete
4. ponavljamo od 1. do 3. sve dok 5. nije zadovoljeno
5. kada dođemo do čvora u kojem postoji samo jedan znak rezultat je novoizračunati indeks u prethodnom čvoru plus jedan - povratak iz rekurzija

Algorithm 2 Rank of character c until position p

```
function Rank(c, p)
    if Self.IsLeaf then
        return p
    end if
    CharBit ← bit representing c in bitmap of current node
    o ← BinaryRank(CharBit, p)
    if CharBit = 1 then
        Rank ← RightChildNode.Rank(c, o)
    else
        Rank ← LeftChildNode.Rank(c, o)
    end if
    return Rank
end function
```

Slika 2.3: Pseudokod funkcije rang. Slika preuzeta iz [4].

Na primjer *rank*$_9$*('n')*. Prvi ulazak rezultira novim indeksom 6 i prelazimo u desno dijete (jer 'n' pripada skupini 1 u prvoj podabecedi). Drugi ulazak rezultira novim indeksom 5 te prelazimo u lijevo dijete. Kod trećeg ulaska novi indeks je 2, prelazimo u desno dijete. Desno dijete je jednoznačno i zadovoljen je uvjet 4. te dobivamo rezultat: $index + 1 = 2 + 1 = 3$.

2.4. Algoritmi

Neke analize nizova zahtijevaju polje najdužih zajedničkih prefiksa (*engl. longest common prefix array* - LCP). Polje sadržava duljine najdužih zajedničkih prefiksa između svih parova koji su uzastopno poredani u sufiksnom polju. Primjer izračunatog LCP polja prikazan je na slici 2.4.

i	SA	LCP	BWT	$S_{\mathsf{SA}[i]}$
1	19	-1	n	$
2	3	0	l	_anele_lepanelen$
3	9	1	e	_lepanelen$
4	4	0	_	anele_lepanelen$
5	13	5	p	anelen$
6	8	0	l	e_lepanelen$
7	1	1	$	el_anele_lepanelen$
8	6	2	n	ele_lepanelen$
9	15	3	n	elen$
10	17	1	l	en$
11	11	1	l	epanelen$
12	2	0	e	l_anele_lepanelen$
13	7	1	e	le_lepanelen$
14	16	2	e	len$
15	10	2	_	lepanelen$
16	18	0	e	n$
17	5	1	a	nele_lepanelen$
18	14	4	a	nelen$
19	12	0	e	panelen$
20		-1		

Slika 2.4: Sufiksno polje (SA), BWT niz i LCP polje. Slika preuzeta iz [1].

U ovom radu obrađuje se implementacija LCP polja temeljena na BWT transformaciji. Prema radu[1] najlošije vrijeme izvođenja algoritma je $O(n log \sigma)$. Algoritmi 1 i 2 za izračunavanje LCP polja objašnjeni su u nastavku.

2.4.1. Algoritam 1

Feragina i Manzini u članku[2] pokazuju kako je moguće pretraživati uzorke u sufiksnom polju unazad znak po znak bez spremanja sufiksnog polja u memoriju. Neka je $\omega \epsilon \Sigma$ i ω podniz od S. Za ω-interval $[i..j]$ u sufiksnom polju SA niza S (ω je prefiks svih sufiksa u sufiksnom polju $S_{SA[k]}$ za sve $i \leq k \leq j$), funkcija *backwardSearch(c, [i..j])* vraća $c\omega$-interval $[C[c] + Occ(c, i-1) + 1..C[c] + Occ(c, j)]$ gdje je $C[c]$ ukupni broj pojavljivanja znakova u S koji su strogo leksikografski manji od c, a $Occ(c, i)$ je broj pojavljivanja znaka c u BWT transformaciji u intervalu $[1..i]$.

Umjesto daljnjeg razmatranja implementacije u nastavku je dan pseudokod funkcije *getIntervals([i..j])*.

```
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)
```

Slika 2.5: Algoritam 1. Preuzeto iz [1].

Funkcija za zadani ω-interval $[i..j]$ vraća listu $c\omega$-intervala. Točnije, kreće s ω-intervalom $[i..j]$ u korijenu stabla valića i prolazi njime pretraživanjem u dubinu. Na svakom čvoru v izračunava se rank za dohvat broja nula $(b_0 - a_0)$ u bit vektoru unutar trenutnog intervala. Ako je $(b_0 < a_0)$, onda postoje znakovi u BWT$[i..j]$ koji pripadaju lijevom podstablu čvora v, i algoritam nastavlja rekurzivnim pozivom lijevo dijete čvora. Ako je $(b_1 > a_1)$, tj. broj jedinica je pozitivan, tada se rekurzivno poziva desno

dijete čvora. Ako se trenutnim intervalom $[p..q]$ može dohvatiti odgovarajući list znaka c, tada je $[C[c] + p \,..\, C[c] + q]$ $c\omega$ interval.

2.4.2. Algoritam 2

Algoritam služi za izračunavanje LCP polja i temelji se na algoritmu 1. U nastavku je prikazan pseudokod.

```
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
```

Slika 2.6: Algoritam 2. Preuzeto iz [1].

Potrebno je inicijalizirati LCP polje veličine $n + 1$ gdje je n broj znakova. Prvi i zadnji element treba postaviti na -1, a ostale na neku neodređenu vrijednost. Zatim se inicijalizira red Q - FIFO struktura koja sadrži strukture $< [i..j], l >$. Algoritam 2 računa $c\omega$ intervale pozivanjem algoritma 1, tj. funkcije $getIntervals([i..j])$.

Određeni interval uzima se iz strukture u redu Q. Za svaki dobiveni $c\omega$ interval $[lb..rb]$ provjerava se vrijednost LCP polja na poziciji $rb + 1$. Ako je neodređena, na tu poziciji se stavlja vrijednost l, a u Q se stavlja nova struktura $< [lb..rb], l + 1 >$ te se algoritam nastavlja sve dok Q nije prazan.

3. Eksperimenti i rezultati

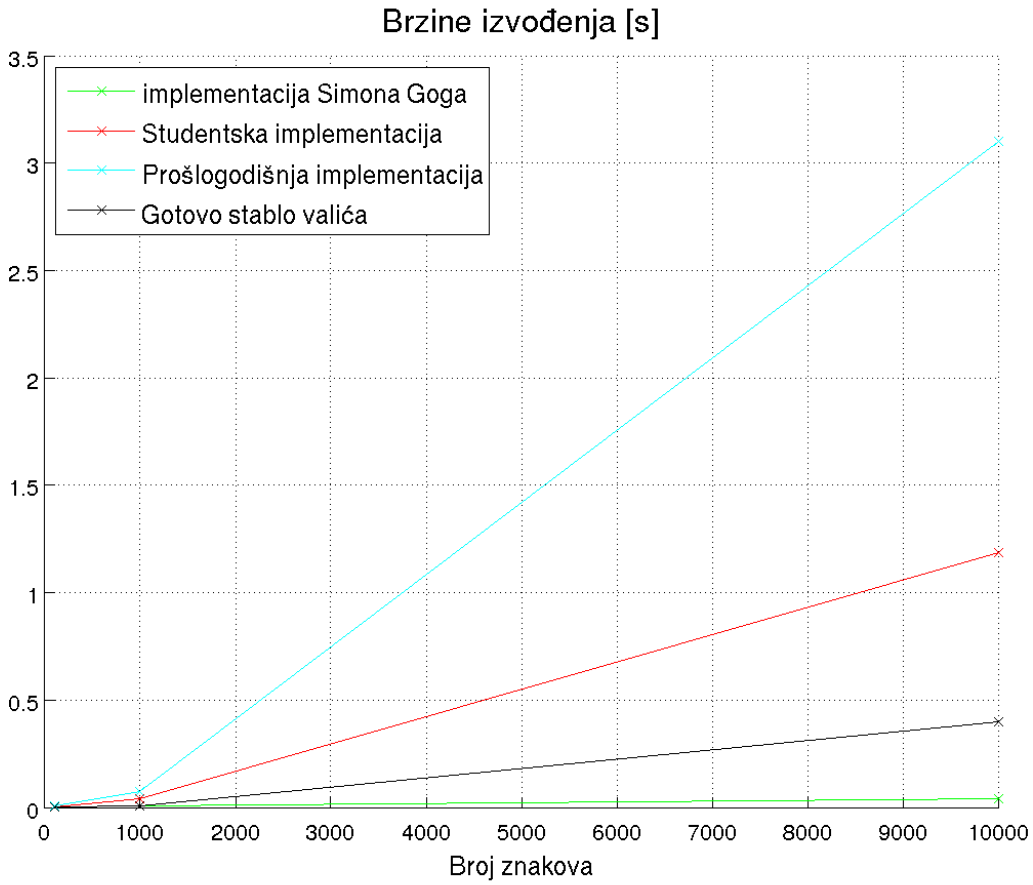
Pri analizi vremena uspoređuje se:

1. Originalna implementacija Simona Goga
2. Studentska implementacija algoritma
3. Studentska implementacija uz gotovu rank funkciju
4. Prošlogodišnja studentska implementacija algoritma

Kao što je vidljivo u tablici 3.1, brzine izvođenja ovise o veličini ulaza. Izvođenje programa najveći utrošak vremena ostvaruje na funkciji *Rank*. Kod studentske implementacije funkcije *Rank* dobiva se eksponencijalni rast vremena izvođenja, a kod gotove implementacije se također dobije eksponencijalni rast, ali uz manji nagib. Grafovi brzine izvođenja su na slici 3.1.

Broj znakova	Implementacija Simona Goga [s]	Binarno stablo valića [s]	Gotova implementacija stabla [s]	Studentska implementacija (prošlogodišnja) [s]
100	0.037231	0.00074	0.00246	0.00479
1,000	0.030089	0.03871	0.00499	0.07335
10,000	0.022991	1.18603	0.04204	3.1015
100,000	0.03929	107.89600	0.39687	-
1,000,000	0.223351	-	5.38922	-

Tablica 3.1: Brzina izvođenja

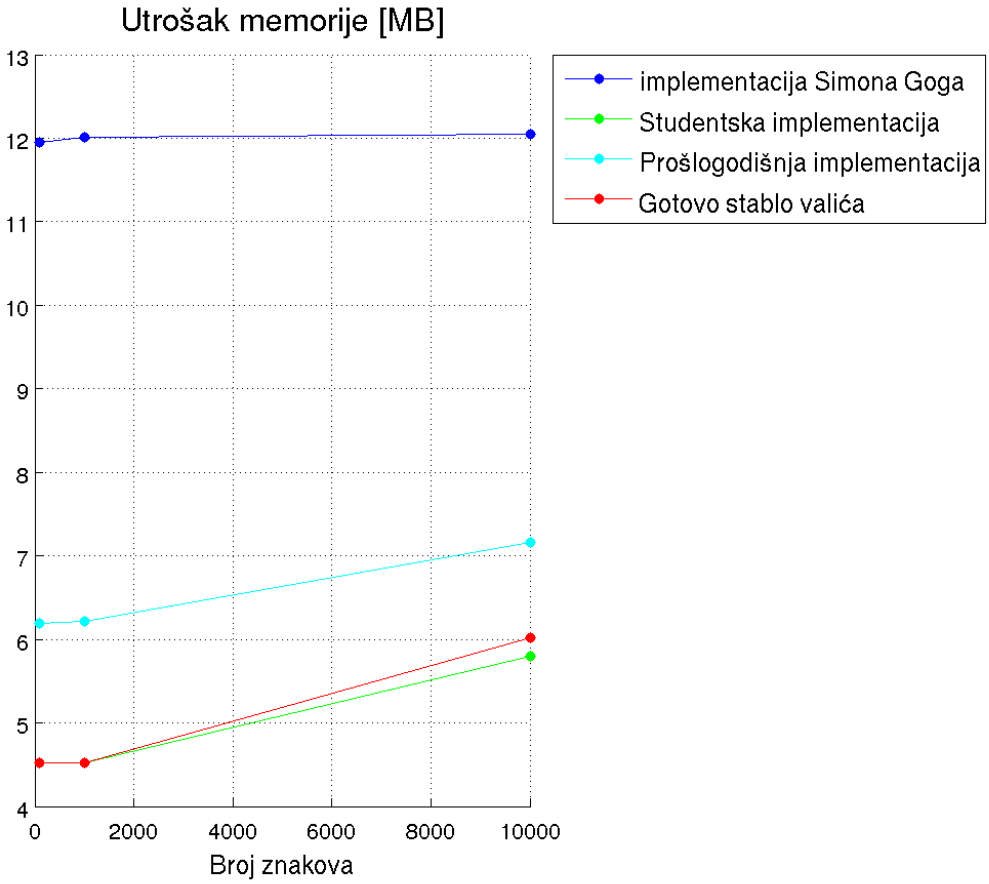


Slika 3.1: Brzina izvođenja

Slijedeća važna komponenta pri ocjeni rada izvođenja je utrošak memorije. Zanimljivo je kako kod implementacije Simona Goga utrošak memorije za nizove do sto tisuća je skoro isti, odnosno kreće se oko 12 MB, a tek za niz od milijun znakova dobiva se nešto znatniji skok na 15 MB. Studentske implementacije, slično kao i za brzinu izvođenja, imaju ekspnencijalan rast utroška memorije. Za male nizove studentska implementacija koristi znatno manje memorije nego implementacija Simona Goga (oko 4 MB), ali za velike nizove dolazi do eksplozije potrošnje memorije. To je uzrokovano cache-iranjem rezultata funkcije *Rank*. Funkcija *Rank* se može za neki podniz i određeno slovo iz abecede pozvati više puta, stoga se radi ubrzavanja rada algoritma izračunate vrijednosti spremaju hash-tablici. Pri svakom slijedećem pozivu funkcije *Rank* prvo se pokuša naći rezultat u *cache*-u, te ako zapis u *cache*-u ne postoji, tek tada se obavlja izračun. Pokušaj dohvata i spremanja rezultata iz *cache* memorije se obavlja u konstantnom vremenu (u veliko O notaciji je $O(1)$), ali kao što je već rečeno, hash-tablica iziskuje relativno velike količne memorije. Podaci potrošnje memorije su vidljivi u tablici 3.1 i grafu 3.2.

Broj znakova	Implementacija Simona Goga [MB]	Binarno stablo valića [MB]	Gotova implementacija stabla [MB]	Studentska implementacija (prošlogodišnja) [MB]
100	11.945	4.527	4.527	6.1856
1,000	12.015.	4.527	4.527	6.2156
10,000	12.043	5.793	6.016	7.1581
100,000	12.336	31.121	31.360	-
1,000,000	15.051	-	316.121	-

Tablica 3.2: Utrošak memorije



Slika 3.2: Utrošak memorije

4. Zaključak

Rezultat ovog rada je uspješno implementirani algoritam traženja najduljeg zajedničkog prefiksa uz korištenje Burrows-Wheelerove transformacije. Prilikom implementacije najveći problem bio je zadovoljiti postavljena resursna ograničenja, brzinu izvođenja i utrošak memorije, čak i ako se na ulazu postave za nizovi s velikim brojem znakova. Najkritičniji dio aplikacije je funkcija *Rank*, tj. preciznije funkcija *BinaryRank* koja se poziva u funkciji *Rank*. Navedena funkcija se može efikasnije napraviti kako bi se dobilo na ubrzanju izvođnja algoritma. Iako aplikacija za velike nizove radi puno lošije nego originalna implementacija Simona Goga, primjetno je kako studentska implementacija za male nizove (do 1000 znakova) troši znatno manje memorije i vremenski je brža. Sve u svemu, ostvaren je cilj projekta, dobivena aplikacija ima prihvatljive brzine izvođenja i utrošak memorije.

5. Literatura

[1] Timo Beller, Simon Gog, Enno Ohlebusch, i Thomas Schnattinger. *Computing the longest common prefix array based on the Burrows–Wheeler transform*, svezak 18, stranice 22 – 31. 2013. doi: http://dx.doi.org/10.1016/j.jda.2012.07.007. URL `http://www.sciencedirect.com/science/article/pii/S1570866712001104`. Selected papers from the 18th International Symposium on String Processing and Information Retrieval (SPIRE 2011).

[2] Manzini G. Ferragina, P. Opportunistic data structures with applications. *IEEE Symposium on Foundations of Computer Science*, stranice 390 – 398, 2000.

[3] Mile Šikić i Mirjana Domazet-Lošo. *Bioinformatika*. 1nd izdanju, 2013.

[4] Jan H. Knudsen i Roland L. Pedersen. *Engineering Rank and Select Queries on Wavelet Trees*. 2015.

6. Sažetak

Analize nizova su računalno zahtjevni problemi koji za velike instance nizova mogu iziskivati velike količine resursa. Efikasne analize se postižu korištenjem sufiksnih polja, a često je potrebno i polje najduljih zajedničkih prefiksa. U ovom radu objašnjeno je računanje najduljeg zajedničkog prefiksa korištenjem Burrows-Wheelerove transformacije sufiksnog polja u indekse podnizova unutar cijelog teksta. Za spremanje Burrows-Wheelerove transformacije korišteno je stablo valića.