SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

PROJEKT IZ BIOINFORMATIKE

Računanje najduljeg zajedničkog prefiksa temeljeno na BWT

Lovre Mrčela, Ana Škaro, Ante Žužul
Voditelj: *doc.dr.sc. Mirjana Domazet-Lošo*

Zagreb, siječanj 2017.

SADRŽAJ

1. Uvod

Bioinformatika se bavi analizom sekvenci genoma. U toj analizi, mnogi algoritmi koriste sufiksna polja i polja najduljeg zajedničkog prefiksa. Cilj ovog projekta bio je implementirati algoritme 1 i 2 iz rada Beller et al. (2013), uz korištenje gotove knjižnice za izgradnju sufiksnog polja. Implementaciju je potom bilo potrebno testirati s gotovom knjižnicom koja sadržava implementaciju stabla valića, a napravljena je i naša implementacija stabla valića, odnosno njegova funkcija rang (engl. *rank*). Konačno rješenje je trebalo biti uspoređeno s rezultatima iz studentskog rada Bužić et al. (2016). Kao ulazni niz koristio se genom Escherichie coli.

2. Algoritmi

U svom radu Beller et al. (2013) navode kako porast informacija koje se u novije doba mogu dobiti analizom sekvenci DNK, zahtijevaju podatkovne strukture koje zauzimaju manje prostora. Jedna od njih je stablo valića niza transformiranog Burrows–Wheelerovom transformacijom (BWT). Opisani postupak odvija se u sljedećim koracima: 1) niz se transformira Burrows–Wheelerovom transformacijom, 2) transformirani niz se pohranjuje u stablo valića, 3) stablo valića omogućuje pretraživanje unatrag na originalnom nizu. Mnogi algoritmi za sekvenciranje koriste najdulji zajednički prefiks (engl. *longest common prefix, LCP*), Beller et al. (2013) u svom radu navode kako većina postojećih postupaka za izračun LCP-polja (engl. *LCP-array*) prvo gradi sufiksno polje, a potom u linearnom vremenu dolazi do LCP-polja. Zbog toga, oni predlažu algoritam koji radi direktno nad BW transformiranim nizom znakova, a ne nad sufiksnim poljem. Taj algoritam ima vremensku složenost $O(\mathrm{n}\log\sigma)$, gdje je σ veličina abecede. U nastavku je opisan ovaj postupak, temeljen na algoritmima 1 i 2 iz navedenog rada te podatkovne strukture potrebne za njegovu implementaciju. Kod koji je rezultat ovog rada dostupan je na poveznici https://github.com/mrlovre/LCPA-BWT/.

2.1. Podatkovne strukture

2.1.1. Stablo valića

Stablo valića (engl. *wavelet tree*) omogućuje pretraživanje unatrag u vremenskoj složenosti od $O(\log\sigma)$ po koraku. Za razumijevanje strukture potrebno je prvo objasniti nekoliko pojmova. Poredana abeceda Σ je polje veličine σ, tako da su znakovi poredani uzlazno u polje $\Sigma[1...\sigma]$, primjerice $\Sigma[1]=\$ < \Sigma[2] < ... < \Sigma[\sigma]$. Interval [l...r] je *abecedni interval*, ako je to podinterval intervala $[1...\sigma]$. Za abecedni interval [l...r], niz znakova $BWT^{[l..r]}$ se dobije iz BW transformiranog niza znakova BWT od S, brisanjem svih znakova u BWT koji ne pripadaju pod-abecedi Σ[l...r] od $\sigma[1...\sigma]$. Stablo valića niza znakova BWT izgrađeno nad abecedom $\Sigma[1...\sigma]$ je balansirano binarno stablo

pretraživanja (engl. *balanced binary search tree*) definirano u nastavku. Svaki čvor v odgovara nizu znakova BWT$^{[l..r]}$, gdje je [l..r] abecedni interval. Korijen stabla odgovara nizu znakova BWT = BWT$^{[1..\sigma]}$. Ako je l=r, onda taj čvor v nema djece. Inače, svaki čvor v ima dvoje djece: **lijevo dijete** odgovara nizu znakova BWT$^{[l..m]}$, dok **desno dijete** odgovara nizu znakova BWT$^{[m+1..r]}$, gdje je m = $\lfloor \frac{l+r}{2} \rfloor$. U ovom slučaju v sadrži bit vektor (engl. *bit vector*) B$^{[l..r]}$, čiji je i-ti element 0 ako je i-ti znak u BWT$^{[l..r]}$ iz pod-abecede Σ[l..m], a 1 ako je iz pod-abecede Σ[m+1..r]. Drukčije rečeno, element bit vektora je 0 ako odgovarajući znak pripada lijevom podstablu, a 1 ako odgovarajući znak pripada desnom podstablu. Izgradnja stabla valića objašnjena je na primjeru u poglavlju 2.2.1.

Svaki vektor B u stablu predprocesira se tako da se upiti *rank$_0$(B,i)* i *rank$_1$(B,i)* mogu odgovoriti u konstantnom vremenu, gdje *rank$_b$(B,i)* predstavlja broj pojavljivanja bita *b* u B[1..i]. Stablo valića ima visinu $O(\log \sigma)$. S obzirom da je u implementaciji dovoljno spremiti samo bit vektore, stablo valića zahtijeva samo n$\log \sigma$ bitova prostora i $O(nlog\sigma)$ bitova za podatkovne strukture koje podržavaju rang upite u konstantnom vremenu.

2.1.2. Sufiksno polje

Sufiksno polje (engl. *suffix array, SA*) niza znakova S je cjelobrojno polje u intervalu od 1 do *n* koje određuje leksikografski poredak *n* sufiksa niza znakova S. Točnije, sufiksno polje SA zadovoljava: S$_{SA[1]}$ < S$_{SA[2]}$ < ... < S$_{SA[n]}$, gdje S$_i$ označava i-ti sufiks S[i..n] niza znakova S.

2.1.3. Burrows–Wheelerova transformacija (BWT)

Burrows–Wheelerova transformacija ili BWT, pretvara niz znakova S u niz znakova BWT[1..n], definiran kao:

$$BWT[i] = \begin{cases} S[SA[i]-1], & \text{ako } SA[i] \neq 1 \\ \$, & \text{inače.} \end{cases}$$

2.2. Izračunavanje LCP na temelju BWT

Autori u radu Beller et al. (2013) koriste dva algoritma kako bi izgradili LCP polje i naš osnovni zadatak unutar ovog projekta bio je upravo implementacija tih dvaju algoritama. Prema prvom algoritmu koji su autori predložili, za jedan ω-interval[i..j],

funkcija *getIntervals([i..j])* vraća listu svih $c\omega$-intervala. To ustvari znači da se unutar intervala [i..j] pronađu svi znakovi abecede Σ, koji se potom poredaju leksikografski te se za svaki od znakova računa njegov ω-interval. Na poslijetku, funkcija vraća onoliko cω intervala koliko ima jedinstvenih znakova u intervalu [i..j]. Postupak započinje ω intervalom [i..j] u korijenu stabla valića te se nastavlja spuštati u dubinu (kao DFS (engl. *depth-first search*) algoritam). Dok se nalazi u trenutnom čvoru, algoritam postavlja rang upit stablu valića (složenost upita je konstantna!), kako bi došao do broja b_0 - a_0. Taj broj predstavlja nule u bit vektoru trenutnog čvora *v* unutar trenutnog intervala. Ako je ta vrijednost pozitivna u BWT[i..j] se nalaze znakovi koji pripadaju lijevom podstablu čvora *v* i algoritam nastavlja rekurzivno pozivanje unutar lijevog djeteta čvora *v*. Isto tako, ako je broj jedinica pozitivan (b_1 - a_1), onda se algoritam nastavlja odvijati u desnoj grani. Algoritam se zaustavlja kad se za trenutni interval [p..q] dosegne list stabla koji odgovara znaku *c*. Tada je cω-interval [C[c]+p..C[c]+q], gdje je C[c] zbroj rangova svih elemenata iz poredane abecede koji su leksikografski manji od znaka c. Vremenska složenost ovog postupka je $O(k\log\sigma)$, gdje je k duljina liste cω-intervala.

Kao što je već rečeno, **algoritam 2** se oslanja na algoritam 1. Svi elementi polja LCP[1..n+1] se inicijalno postavljaju na neku nemoguću vrijednost, npr. $\perp$, osim prvog i zadnjeg elementa koji poprimaju vrijednost -1. Jedan red (engl. *queue*) sadržava parove (interval,*l*). Na početku rada algoritma u redu se nalazi samo interval [1..n], a *l* vrijednost je postavljena na 0. Algoritam potom skida iz reda interval (po principu FIFO, dok god se red ne isprazni), te pozivom metode *getIntervals* prvog algoritma dobiva listu cω-intervala za dani interval. Za svaki interval [lb..rb] iz liste cω-intervala gleda se je li vrijednost polja LCP[rb+1] nepostavljena (iznosi $\perp$) te ako jest, taj interval [lb..rb] se dodaje u red, a njegova *l* vrijednost se povećava za 1 u odnosu na *l* vrijednost intervala koji je posljednji skinut iz reda. Također, vrijednost LCP[rb+1] se postavlja na tu posljednju *l* vrijednost.

Algorithm 1 prema Beller et al. (2013)

function GETINTERVALS([i..j])
  $list \leftarrow []$
  getIntervals'([i..j],[1..σ],list)
  return $list$
end function
function GETINTERVALS'([i..j],[l..r],list)
  if $l = r$ **then**
    $c \leftarrow \Sigma[l]$
    $add(list, [C[c] + i..C[c] + j)$
  else
    $(a_0, b_0) \leftarrow (rank_0(B^{[l..r]}, i - 1), rank_0(B^{[l..r]}, j))$
    $(a_1, b_1) \leftarrow (i - 1 - a_0, j - b_0)$
    $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
    if $b_0 > a_0$ **then**
      getIntervals'([a_0+1..b_0],[l..m],list)
    end if
    if $b_1 > a_1$ **then**
      getIntervals'([a_1+1...b_1],[m+1..r],list)
    end if
  end if
end function

Algorithm 2 prema Beller et al. (2013)

inicijalizacija polja LCP[1..n+1] /* npr. LCP[i]=$\perp$ za sve $1 \leq i \leq n+1$ */
LCP[1] $\leftarrow$ -1; LCP[n+1] $\leftarrow$ -1
inicijaliziraj prazni red
dodaj (<[1..n],0>) u red q
while q nije prazan **do**
  <[i..j],l> $\leftarrow q.pop()$
  list $\leftarrow$ *getIntervals*([i..j])
  for each `[lb..rb] in list` **do**
    if LCP[rb+1]=$\perp$ **then**
      $q.push(< [lb..rb], l+1 >)$
      LCP[rb+1] $\leftarrow l$
    end if
  end for
end while

2.2.1. Primjer

U nastavku je dan primjer rada algoritma na stringu S_u=*annasanannas*. preuzet iz prošlogodišnjeg rada studenata Bužić et al. (2016).

Prvo se iz niza gradi sufiksno polje.

1. Na kraj niza znakova S_u dodaje se znak $ (najmanji znak abecede) Sada imamo niz S = annasanannas$.

2. Svakom sufiksu niza S pridružuju se indeksi od 1 do *n*, počevši od najduljeg. Ovo je prikazano u **tablici 2.1**

3. Sufiksi se potom poredaju leksikografski, čime nastaje sufiksno polje SA = [13, 6, 8, 1, 11, 4, 7, 10, 3, 9, 2, 12, 5]. Ovo je prikazano u **tablici 2.2**.

4. Iz polja SA se prema formuli navedenoj u 2.1.3 izračunava Burrows-Wheelerov niz BWT[1..n]. Rezultat je prikazan u 4. stupcu **tablice 2.2**.
   Npr. BWT[10]=S[SA[10]-1]=S[9-1]=S[8]=a.

5. Iz Burrows-Wheelerove transformacije niza znakova, gradi se stablo valića. Poredana abeceda u ovom primjeru ima 4 znaka Σ[1..4]=$ans. U korijen stabla se stavlja bit vektor dobiven kodiranjem niza BWT prema pravilu opisanom u

poglavlju 2.1.1. Abeceda se podijeli na dva dijela, u ovom slučaju: Σ[1..2]=$a i Σ[3..4]=ns. Znakovi u prvoj polovini se kodiraju vrijednošću 0, a ostali 1. Ovaj postupak se ponavlja sve dok se u čvoru ne nalaze samo jednaki znakovi. Nastalo stablo može se vidjeti na slici 2.1.

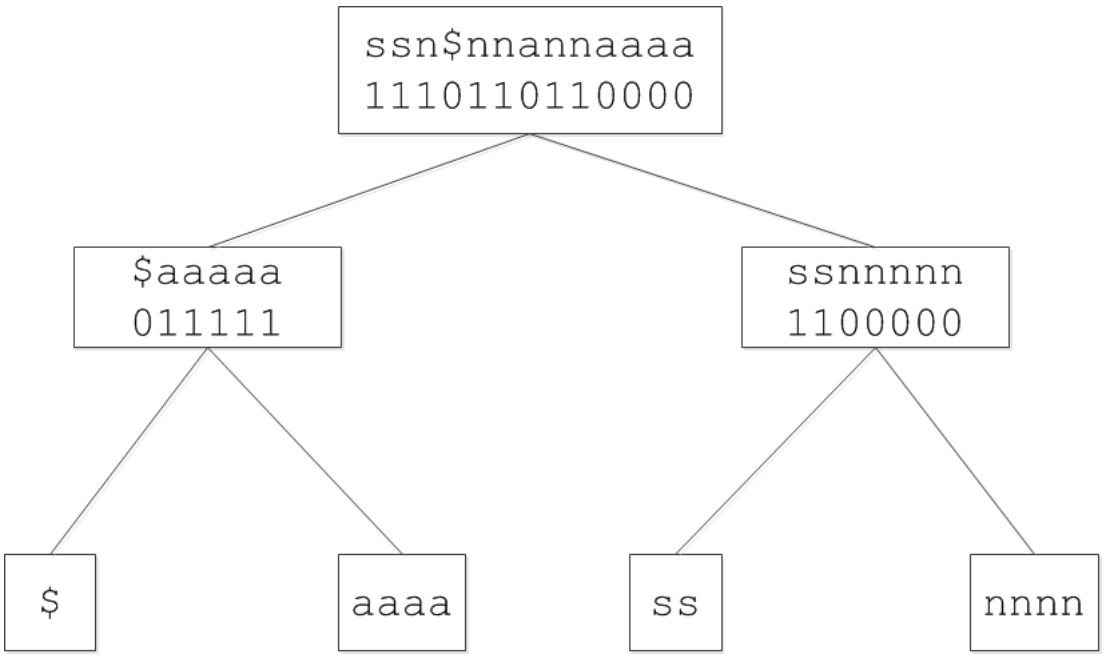
6. U ovom koraku može započeti konstrukcija polja najdužih zajedničkih prefiksa, za što se koriste algoritmi 1 i 2 iz rada Beller et al. (2013) opisani u poglavlju 2.2.

   (a) Inicijalizacija polja LCP i reda Q:
   - LCP = [-1,$\perp$, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$, $\perp$, -1]
   - red Q = [<[1..13],0>]

   (b) Uklanjanje elementa iz reda Q po FIFO principu te izračunavanje cω-intervala za taj interval pomoću funkcije *getIntervals*.
   - Indeks početka intervala dobiva se po formuli *rang*(c,i-1)+C[c]+1. Indeks kraja intervala dobiva se po formuli C[c]+*rang*(c,j).Funkcija *rang(a,k)* vraća broj pojavljivanja znaka *a* do *k*-tog indeksa u polju.
   - Interval I = [i..j]=[1..13].
   - Znakovi abecede iz intervala I se poredaju leksikografski. Ima onoliko cω-intervala koliko je i jedinstvenih znakova.
   - C = [0,1,6,11]
   - U I se nalaze **svi** znakovi abecede ($,a,n,s). Algoritam prolazi svaki od znakova i računa njegov interval prema formuli navedenoj iznad.

   rang('$',0)+C['$' +1..C['$']+*rang*('$',13)]=[0+0+1..+0+1]=[1..1]
   rang('a',0)+C['a' +1..C['a']+*rang*('a',13)]=[0+1+1..+1+5]=[2..6]
   rang('n',0)+C['n' +1..C['n']+*rang*('n',13)]=[0+6+1..+6+5]=[7..11]
   rang('s',0)+C['s' +1..C['s']+*rang*('s',13)]=[0+11+1..+11+2]=[12..13]
   - Konačno, povratna vrijednost *getIntervals* je lista cω-intervala: [[1..1],[2..6],[7..11],[12..13]]. Za svaki od tih intervala se potom provjerava vrijednost LCP[rb+1] te se postupa sukladno koracima u pseudokodu algoritma 2.

   (c) Konačna vrijednost polja LCP, nakon što algoritam završi s radom je: LCP = [-1,0,2,5,1,2,0,2,3,1,4,0,1,-1].



Slika 2.1: Slika prikazuje izgled stabla valića za dani primjer.

Tablica 2.1: Pridruživanje indeksa sufiksima niza S, počevši od najduljeg.

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	$

Tablica 2.2: Sufiksi su poredani leksikografski, a njihovi indeksi čine sufiksno polje SA.

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

3. Rezultati

Nakon implementacije opisanih algoritama zadatak je bio usporediti ih s rezultatima studentskog rada Bužić et al. (2016) [1] te s originalnom implementacijom Simona Goga (prema radu Beller et al. (2013)) na ulaznom nizu koji pripada genomu Escherichie coli. Usporedba je napravljena koristeći alat *cgmemtime*[2] na računalu sa sustavom Arch Linux. Napominjemo da smo implementaciju Simona Goga kojoj smo testirali vrijeme izvođenja preuzeli smo iz rada studenata.

Kako možemo vidjeti u tablici 3.1 i na slici 3.1, naša implementacija se ponašala bolje u odnosu na implementaciju studenata Bužić et al. (2016) i to za čak 8 do 9 puta bolje na duljim nizovima znakova. U odnosu na originalnu implementaciju algoritma, naš je algoritam bio sporiji (što je i u redu za očekivati), ali tek oko 1.5 puta. Dobro je primijetiti da se razlike među algoritmima osjetno primjećuju tek na nizovima iznad deset tisuća (10000) znakova.

Što se tiče potrošnje memorije, prikazane u tablici 3.2 i na slici 3.2, naš algoritam se na nizovima do 250000 znakova ponašao bolje od rada studenata, a kasnije im je bio relativno blizu. Originalna implementacija Simona Goga na kraćim nizovima treba više memorijskog prostora, ali zato pri povećanju ulaznog niza s milijun (1000000) znakova na puni genom Escherichie Coli (4639211) znakova ima porast od tek 2-3 puta, dok naš algoritam zahtjeva gotovo 5 puta više memorije pri takvom skoku.

3.1. Tablice i grafovi

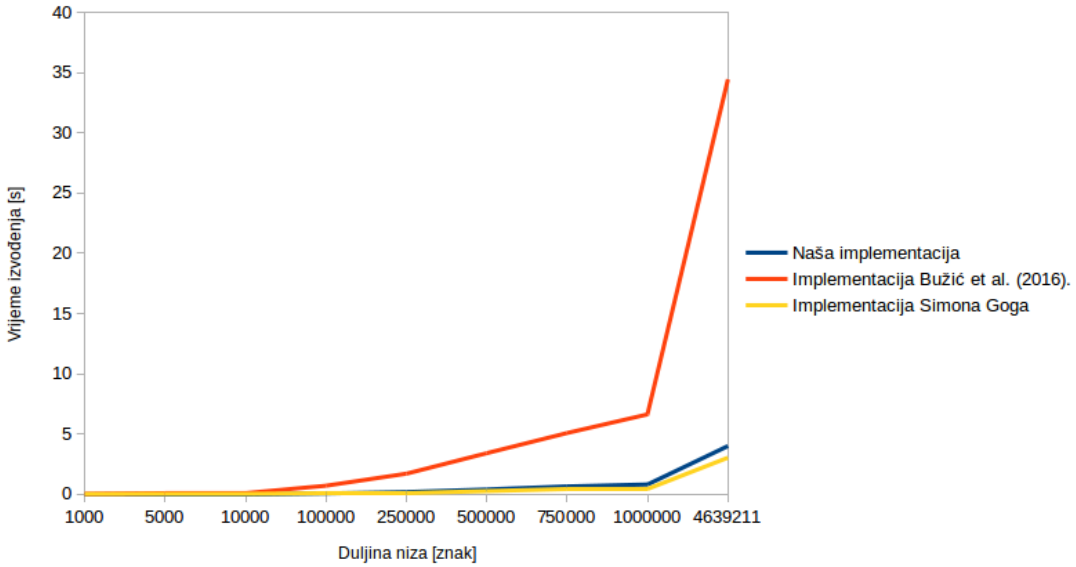
U nastavku su prikazane tablice i grafovi usporedbe vremena izvođenja i zauzeća memorije različitih implementacija algoritma.

[1] Cijeli rad i implementacija su dostupni na *https://github.com/wissil/LCP_BWT*.

[2] Alat *cgmemtime* se može pronaći na *https://github.com/isovic/cgmemtime*.

Tablica 3.1: Rezultati usporedbe vremena izvođenja naše implementacije algoritma i implementacije iz rada Bužić et al. (2016) te originalne implementacije Simona Goga.

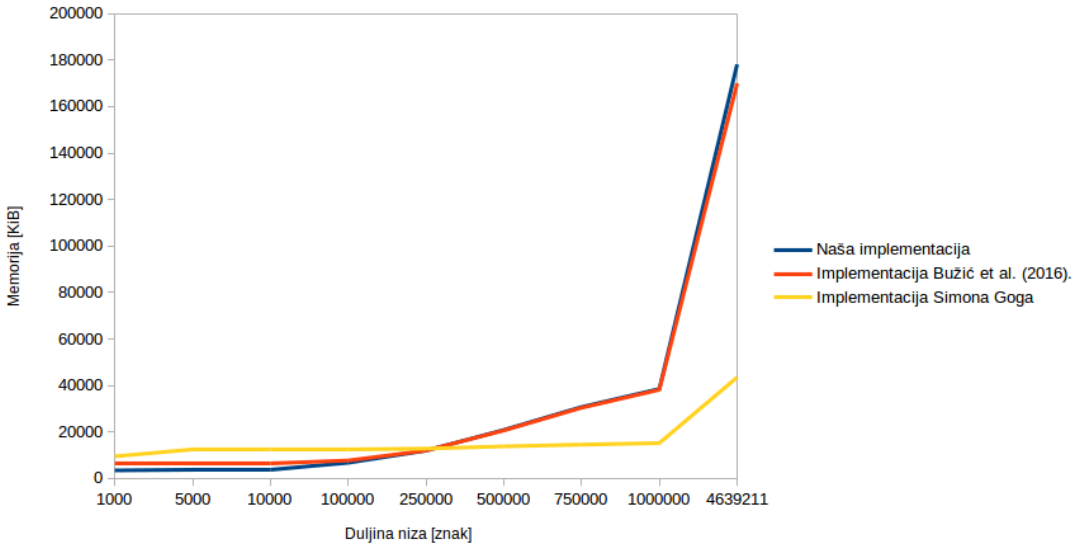
Duljina ulaznog niza [znak]	Naša implementacija [s]	Implementacija Bužić et al. (2016) [s]	Originalna implementacija Simona Goga [s]
1000	0.000	0.023	0.016
5000	0.000	0.039	0.031
10000	0.008	0.070	0.023
100000	0.062	0.703	0.055
250000	0.18	1.703	0.117
500000	0.391	3.398	0.258
750000	0.648	5.078	0.398
1000000	0.805	6.625	0.469
4639211	4.000	34.438	3.031



Slika 3.1: Graf prikazuje rezultate usporedbe vremena izvođenja algoritma prema tablici 3.1

Tablica 3.2: Rezultati usporedbe zauzeća memorije naše implementacije algoritma i implementacije iz rada Bužić et al. (2016) te originalne implementacije Simona Goga.

Duljina ulaznog niza [znak]	Naša implementacija [KiB]	Implementacija Bužić et al. (2016) [KiB]	Originalna implementacija Simona Goga [KiB]
1000	3476	6336	9532
5000	3732	6392	12308
10000	3592	6400	12320
100000	6768	7736	12360
250000	11992	11996	12848
500000	20880	20644	13816
750000	30736	30424	14564
1000000	38532	38172	15216
4639211	178172	170020	43516



Slika 3.2: Graf prikazuje rezultate usporedbe zauzeća memorije prilikom izvođenja algoritma prema tablici 3.2

4. Zaključak

Organske molekule, npr. DNK, proteini, koji su predmeti proučavanja bioinformatike, predstavljeni su nizovima znakova. Ti nizovi znakova su velikih duljina pa njihovo pretraživanje i analiza zahtijeva korištenje optimiziranih algoritama i struktura podataka. Jedna od stvari koja pomaže tim postupcima je polje najdužih sajedničkih prefiksa (LCP-polje). LCP polje može se izgraditi u složenosti $O(\mathrm{n}\log\sigma)$, gdje je σ veličina abecede, kako je pokazano radom Beller et al. (2013). U našem radu je uspješno implementiran predloženi algoritam koji uključuje izgradnju stabla valića nad Burrows–Wheelerovim transformiranim nizom znakova. Naš algoritam se bolje nosi s dugim nizovima od prethodno implemetiranih studentskih algoritama. Ipak, još uvijek je nešto lošiji od implementacije Simona Goga kad se radi o duljim nizovima znakova. Kao najveći nedostatak možemo primijetiti nagli porast zauzeća memorije pri povećanju ulaznog niza s milijun na preko 4.5 milijuna znakova. Originalna implementacija na manjim nizovima troši više memorije od naše, no zato je porast potrošnje znatno stabilniji.

5. Literatura

T. Beller, S. Gog, E. Ohlebusch, i T. Schnattinger. *Computing the longest common prefix array based on the Burrows-Wheeler transform*. Elsevier B.V., 2013.

D. Bužić, F. Kozjak, i I. Vanjak. *Izračun polja najdužih zajedničkih prefiksa korištenjem Burrows-Wheelerove transformacije*. 2016.

6. Sažetak

Ovaj rad je napravljen kao projekt iz predmeta Bioinformatika diplomskog studija računarstva na Fakultetu elektrotehnike i računarstva, sveučilišta u Zagrebu. Cilj projekta bila je implementacija algoritama za izračun polja najdužih zajedničkih prefiksa (LCP-polja) iz rada Beller et al. (2013), uz korištenje gotove knjižnice za izgradnju sufiksnog polja. Implementacija je testirana i uz gotovu i uz vlastitu implementaciju stabla valića. Konačno rješenje je uspoređeno s rezultatima iz studentskog rada Bužić et al. (2016) naspram kojih je dalo bolje rezultate po pitanju vremena izvođenja i zauzeća memorije.