SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

PROJEKT IZ BIOINFORMATIKE

Računanje najduljeg zajedničkog prefiksa temeljeno na BWT

Autori
FRANE DŽAPO, DARKO DAŠIĆ
Voditelj
doc.dr.sc MIRJANA DOMAZET-LOŠO

Zagreb, siječanj 2018.

Sadržaj

1 Uvod

Bioinformatika je grana znanosti koja se bavi analizom sekvenci genoma. U toj analizi, mnogi algoritmi koriste raznovrsne algoritme manipulacije stringova, kao što su sufiksna polja i polja najduljeg zajedničkog prefiksa.

Cilj ovog projekta je bio čim učinkovitije implementirati algoritme 1 i 2 iz rada Beller et al. (2013), koristeći pritom gotovu knjižnicu za izgradnju sufiksnog polja[1], te potom ostvarenu implementaciju usporediti s originalnom te sa rezultatima prošlogodišnjeg studentskog tima, koji su opisani u njihovu radu Mrčela et al. (2017). Za usporedbu rezultata korišteni su sintetski podaci različitih duljina i abeceda (sekvence DNA i slučajno generirana abeceda), te sekvence genoma bakterije E. coli.

[1] sais.hxx

2 Algoritmi

Za potrebe ovog rada korišteni su algoritmi i strukture izgradnje sufiksnog polja, Burrows-Wheelerova transformacija, stablo valića te algoritmi 1 i 2 opisani u radu Beller et al. (2013).

Algoritmi 1 i 2 iz rada Beller et al. (2013) koriste stablo valića, za koje je potrebna BW transformacija ulaza, za koju je pak potrebno sufiksno polje, redosljed računanja je kako slijedi:

1. Izračuna se sufiksno polje originalnog niza
2. Na temelju sufiksnog polja odredi se BW transformacija
3. Nad BW transformacijom izgradi se stablo valića
4. Korištenjem stabla valića te algoritama 1 i 2 iz rada Beller et al. (2013), konstruira se polje najduljih zajedničkih prefiksa

Opisani algoritam izračunava polje najduljih zajedničkih prefiksa u vremenskoj složenosti $O(n \log \sigma)$ (gdje je n duljina ulaznog niza, a σ veličina njegove abecede).

2.1 Sufiksno polje

Sufiksno je polje struktura podataka koja u sebi sadrži sortirani poredak svih sufiksa početnog niza.

Konkretno, sufiksno polje SA_S niza znakova S je polje cijelih brojeva iz intervala $1..N$ koji predstavljaju leksikografski poredak svih sufiksa niza S. Preciznije, svi članovi sufiksnog polja zadovoljavaju izraz $S_{SA[1]} < S_{SA[2]} < ... < S_{SA[n]}$, gdje S_i označava i-ti sufiks niza znakova S, te sadrži znakove $S[i..n]$.

2.2 Burrows-Wheelerova transformacija (BWT)

Burrows-Wheelerova transformacija na reverzibilan način preuređuje ulazni niz znakova S u niz jednake duljine $BWT_S[1..N]$, koji ima svojstvo da se slični znakovi nalaze blizu jedni drugih te služi za kompresiju podataka.

Redoslijed elemenata u preuređenm nizu određuje se korištenjem sufiksnog polja prema formuli:

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

Tablica 1: Svi sufiksi niza $S = ananas$, od najdužeg do najkraćeg:

i	S_{SA}[i]
1	ananas$
2	nanas$
3	anas$
4	nas$
5	as$
6	s$
7	$

Tablica 2: Sufiksno polje ulaznog niza $S = ananas$ te rezultat BW transformacije nad njime

i	SA[i]	S_{SA}[i]	BWT[i]
1	7	$	s
2	1	ananas$	$
3	3	anas$	n
4	5	as$	n
5	2	nanas$	a
6	4	nas$	a
7	6	s$	a

2.3 Stablo valića

Stablo valića je sažeta struktura podataka koja omogućava komprimirano pohranjivanje nizova znakova u obliku stabla, i podržava upite tipa `rank` i `select`, koji omogućavaju učinkovito pretraživanje ulaznog niza unatrag u složenosti $O(\log \sigma)$ po koraku (σ je veličina abecede). Stablo valića konstruira se na sljedeći način:

1. Na temelju svih znakova prisutnih u ulaznom nizu odredi se njegova uzlazno poredana abeceda Σ (veličine $|\Sigma| = \sigma$), i svakom se pripadnom znaku dodijeli njegov indeks u intervalu $[1..\sigma]$.

2. Počevši od cjelokupne abecede i ulaznog niza, stablo valića gradi se tako da se u svakom koraku trenutna abeceda podijeli na dvije

polovice ($\Sigma[l..r] \rightarrow \Sigma_L[l..m] + \Sigma_R[(m+1)..r]$, gdje je $m = \frac{l+r}{2}$), te se trenutni niz S razdijeli na dva, tako da svi znakovi koji pripadaju abecedi Σ_L postanu dio niza S_L, a svi iz Σ_R niza S_R. Sadašnjem se čvoru pridjeljuje niz bitova B duljine jednake duljini trenutnog niza S, gdje je

$$B_i = \begin{cases} 0, & ako\ S_i \in S_L,\ tj.\ ako\ S_i \in \Sigma_L \\ 1, & ako\ S_i \in S_R,\ tj.\ ako\ S_i \in \Sigma_R \end{cases}$$

Konstrukcija se potom rekurzivno nastavlja razmatranjem dvaju novih čvorova L i R koji su djeca trenutnog čvora.

3. Kada se u konstrukciji dosegne čvor čija abeceda je jedno slovo on postaje jednim od listova u stablu valića.

4. Kada više nema niti jednog čvora u stablu koji bi se mogao razdijeliti, konstrukcija je gotova, i stablo valića je izgrađeno.

Dubina ovako konstruiranog stabla valića je $O(\log \sigma)$, a s obzirom da se u svakom čvoru pohranjuju samo bitvektori, prostorna složenost stabla valića je $O(n \log \sigma)$.

2.4 LCP - Algoritmi 1 i 2

Konstrukciju polja najduljih zajedničkih prefiksa iz prethodno izgrađenog stabla valića obavljaju dva algoritma opisana u radu Beller et al. (2013).

2.4.1 Algoritam 1

Prvi od njih, izveden kao funkcija $getIntervals$, za jedan dani ω-interval vraća listu svih odgovarajućih cω-intervala. To se postiže pretražujući stablo valića od korijena prema listovima, u svakom čvoru ispitujući koliko u trenutnom intervalu ima znakova iz lijeve i desne polovice trenutne abecede (tj. iz lijevog ili desnog podstabla), te se potom na odgovarajući način spušta sve do dna stabla. Algoritam se zaustavlja kada dosegne sve dostupne listove stabla (one čiji odgovarajući znakovi abecede postoje u početnom nizu), i za svaki dosegnuti list u listu cω-intervala dodaje novi, $[C[c] + p..C[c] + q]$, gdje je c pripadni znak toga lista, $[p..q]$ interval s kojim je taj list dosegnut, a $C[c]$ kumulativna suma pojavljivanja svih znakova leksikografski manjih od c.

U konačnici, algoritam vraća onoliko novih intervala koliko je jedinstvenih znakova abecede prisutno u početnom ω-intervalu $[i..j]$, a za to mu je potrebno $O(k \log \sigma)$ operacija, gdje je k duljina rezultantne liste cω-intervala.

Algorithm 1 iz rada Beller et al. (2013)

function GETINTERVALS([i..j])
  $list \leftarrow []$
  $getIntervals'([i..j], [1..\sigma], list)$
  return $list$
end function

function GETINTERVALS'([i..j],[l..r],list)
  if $l = r$ **then**
    $c \leftarrow \Sigma[l]$
    $list.add([C[c] + i..C[c] + j)$
  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

2.4.2 Algoritam 2

Drugi algoritam, koji vrši konstrukciju LCP polja koristi prvi algoritam:

1. U početku se svi elementi polja LCP inicijaliziraju na nemoguću vrijednost $\perp$, osim prvog i zadnjeg člana, čija se vrijednost postavlja na -1. Također se stvara i red Q, koji će pohranjivati sve ω-intervale koji trenutno čekaju svoju obradu. U taj se red na početku dodaje interval $[1..n]$, kojemu se pridružuje i vrijednost duljine zajedničkog prefiksa $l = 0$.

2. Sve dok u redu Q postoji još intervala, iz reda se uzima prvi interval i prosljeđuje funkciji $getIntervals$, koja vraća odgovarajuću listu novih cω-intervala.

3. Za svaki interval $[lb..rb]$ iz tako dobivene liste provjerava se vrijednost $LCP[rb+1]$, i ako je ona jednaka $\perp$, taj se interval dodaje u red za daljnju obradu (uz uvećanu vrijednost duljine zajedničkog prefiksa $l' = l + 1$), a $LCP[rb+1]$ se postavlja na trenutnu vrijednost l.

4. Koraci 2 i 3 ponavljaju se sve dok se red Q sasvim ne isprazni čime konstrukcija polja najduljih zajedničkih prefiksa završena.

Algorithm 2 iz rada Beller et al. (2013)

$LCP[1] \leftarrow -1$; $LCP[n+1] \leftarrow -1$; $LCP[i] \leftarrow \perp$, $\forall i \in [2..n]$
$Q = [\varnothing]$ /* prazan red */
$Q.push(\langle [1..n] \rangle, 0)$
while not Q.empty **do**
  $\langle [i..j], l \rangle \leftarrow Q.pop()$
  $list \leftarrow getIntervals([i..j])$
  for each $[lb..rb]$ **in** $list$ **do**
    if $LCP[rb+1] = \perp$ **then**
      $Q.push(\langle [lb..rb], l+1 \rangle)$
      $LCP[rb+1] \leftarrow l$
    end if
  end for
end while

2.5 Primjer rada algoritma

U nastavku je dan primjer rada algoritma na stringu S = ananas.

2.5.1 Izgradnja sufiksnog polja

1. Na kraj ulaznog niza dodaje se znak $ te je sada $S = ananas\$$.

2. Svakom sufiksu niza S pridružuju se indeksi od 1 do n, počevši od najduljeg. Ovo je prikazano u **tablici 3**.

Tablica 3: Pridruživanje indeksa sufiksima niza S

i	S_{SA}[i]
1	ananas$
2	nanas$
3	anas$
4	nas$
5	as$
6	s$
7	$

3. Sufiksno polje SA dobivamo soritiranjem svih sufiksa leksikografski od najmanjeg prema najvećem. To je, skupa sa BW transformacijom, prikazano u **tablici 4**.

2.5.2 Burrows-Wheelerova transformacija

Na temelju izračunatog sufiksnog polja provodi se Burrows-Wheelerova transformacija nad ulaznim nizom prema prethodno navedenoj formuli.

2.5.3 Izgradnja stabla valića

Iz dobivene Burrows-Wheelerove transformacije gradi se stablo valića na sljedeći način:

1. Prvo se stvara sortirana abeceda ulaznog niza koja je jednaka $\Sigma = \$ans$.

Tablica 4: Sufiksno polje ulaznog niza, kao i njegova Burrows-Wheelerova transformacija

i	SA[i]	S_{SA}[i]	BWT[i]
1	7	$	s
2	1	ananas$	$
3	3	anas$	n
4	5	as$	n
5	2	nanas$	a
6	4	nas$	a
7	6	s$	a

2. Pri izgradnji korijena stabla, abeceda se dijeli na dva dijela, $\Sigma_L = \$a$ i $\Sigma_R = ns$. Potom se određuje niz B, koji je jednak 1011000, te se ulazni niz razdjeljuje na $S_L = \$aaa$ i $S_R = snn$.

3. Lijevo dijete korijena, kojemu pripada abeceda $\Sigma = \$a$ i niz $S = \$aaa$, gradi se tako da se abeceda podijeli na $\Sigma_L = \$$ i $\Sigma_R = a$, odredi vektor $B = 0111$, te niz podijeli na $S_L = \$$ i $S_R = a$.

4. Desno dijete korijena, kojemu pripada abeceda $\Sigma = ns$ i niz $S = snn$, gradi se tako da se abeceda podijeli na $\Sigma_L = n$ i $\Sigma_R = s$, pripadni niz na $S_L = nn$ i $S_R = s$, te se čvoru dodijeli niz $B = 100$.

5. Svi preostali čvorovi, budući da imaju abecedu od samo jednog znaka su listovi te je konstrukcija stabla valića završena.

2.5.4 Konstrukcija polja najduljih zajedničkih prefiksa

1. Polje najduljih zajedničkih prefiksa gradi se prema Algoritmima 1 i 2 iz rada Beller et al. (2013):

   (a) Vrijednosti polja LCP postavljaju se na nevažeće($\perp$), osim za $LCP[1]$ i $LCP[n+1]$, koji se postavljaju na -1. U red Q se stavlja početni interval $I = [i..j] = [1..12]$ te duljina zajedničkog prefiksa $l = 0$:

$$LCP = [-1, \perp, \perp, \perp, \perp, \perp, \perp, \perp, -1]$$
$$Q = [\langle [1..12], 0 \rangle]$$

(b) cω-intervali za prvi ω-interval u redu Q izračunavaju se funkcijom *getIntervals* (Algoritam 1 iz Beller et al. (2013)). Za početni slučaj to su

$$[[1..1], [2..4], [5..6], [7..7]]$$

.

(c) Za svaki od dobivenih intervala $[lb..rb]$ se potom provjerava vrijednost $LCP[rb+1]$, i ako je ona jednaka $\perp$, u red se dodaje $\langle [lb..rb], l+1 \rangle$, a na indeks $rb+1$ u polju LCP zapisuje se vrijednost l. Nakon prve iteracije algoritma, stanje je:

$$LCP = [-1, 0, \perp, \perp, 0, \perp, 0, 0, -1]$$

$$Q = [\langle [1..1], 1 \rangle, \langle [2..4], 1 \rangle, \langle [5..6], 1 \rangle, \langle [7..7], 1 \rangle]$$

(d) Ponavljanjem prethodno opisanih koraka konačno se odredi da je vrijednost polja najduljih zajedničkih prefiksa jednaka:

$$LCP = [-1, 0, 3, 1, 0, 2, 0, 0 - 1]$$

2.5.5 Implementacijska napomena

Čitatelju je važno napomenuti da svi opisani algoritmi te primjer koriste indeksiranje počevši od indeksa 1, dok se, zbog korištenja programskog jezika C++, zapravo sve indeksira počevši od 0. Dani algoritmi tada imaju neznatno drugačije uvjete zaustavljanja. Također, iako je stablo valića opisano u cijelosti, u samoj implementaciji upit tipa `select` nije podržan jer nije potreban u implementaciji algoritama iz rada Beller et al. (2013). Sama implementacija može se pronaći na web stranici GitHub[2].

[2]https://github.com/Dzaposlav/bioinf

3 Rezultati

Prethodno opisani algoritam implementiran i ispitan u sklopu ovoga rada, a njegove su performanse uspoređene s originalnom implementacijom autora rada Beller et al. (2013) i prošlogodišnjom studentskom implementacijom opisanom u radu Mrčela et al. (2017).

Točnost algoritma provjerena je na dvije skupine sintetskih podataka različitih veličina abecede – slučajno generiranim sekvencama DNA, slučajno generiranim nasumičnim sekvencama slučajne abecede. Duljine ispitnih podataka bile su u rasponu od 10 do 10000 ulaznih znakova.

Usporedba performansi izvršena je na skupovima sintetskih DNA podataka u rasponu duljina od 1000 do 1.000.000 ulaznih znakova, a vremena izvršavanja i zauzeća memorije određena su korištenjem programa napisanog u programu python.

Performanse ovdje opisane implementacije nisu dovoljno dobre što će biti vidljivo u tablicama 5 i 6. Performanse originalne implementacije nisu izmjerene već su preuzete is prošlogodišnjeg studentsko rada, te je moguće manje odstupanje rezultate, budući da nisu korišteni točno isti ispitni primjeri. Odstupanje je ipak zanemarivo, jer se originalna implementacija pokazala boljom i za nekoliko redova veličine.

Svi izmjereni (i preuzeti) rezultati prikazani su u tablicama 5 i 6.

3.1 Usporedba trajanja izvođenja

Iz prikazanih rezultata trajanja izvođenja na raznim veličinama ulaza vidljivo je da su performanse svih implementacija približno linearne (za istu ulaznu abecedu), što odgovara teoretskoj složenosti algoritma. Prošlogodišnja implementacija, iako dobra, nije dosegla performanse originalne implementacije. Naša implementacija kaska za ostalima, vrlo vjerojatno jer se zbog nedostatka vremena i krive procjene autora, nisu koristili nikakve optimizacije programskog koda.

Tablica 5: Rezultati usporedbe vremena izvođenja naše implementacije algoritma i implementacije iz rada Mrčela et al. (2017) te originalne implementacije dane u radu Beller et al. (2013).

Duljina ulaznog niza [znak]	Naša implementacija [s]	Implementacija Mrčela et al. (2017) [s]	Originalna implementacija implementacija [s]
1000	0.001	0.000	0.016
5000	0.002	0.000	0.031
10000	0.009	0.008	0.023
100000	0.088	0.062	0.055
250000	0.204	0.18	0.117
500000	0.457	0.391	0.258
750000	0.871	0.648	0.398
1000000	1.155	0.805	0.469
4639211	5.179	4.000	3.031

3.2 Usporedba zauzeća memorije

Rezultati mjerenja zauzeća memorije pojedinih implementacija, prikazani u tablici 6, također ukazuju na to da nisu korištene prigodne optimizacije koda te je vidljivo da rezultati potrošnje memorije odskaću od originalne implementacije, dok su rezultati bolji od prošlogodišnje implementacije. To je vrlo vjerojatno posljedica toga što memorijske perfomanse nisu mijenjane za brže vrijeme izvođenja koristeći neku od bržih, ali memorijski složenijih struktura za pohranu stabla valića i bitvektora. Vidi se da na manjim primjerima naša implementacija koristi manje memorije čak i od originalne implementacije što je vjerojatno posljedica nikakvih optimiranja vremena u zamjenu za memoriju.

Tablica 6: Rezultati usporedbe zauzeća memorije naše implementacije algoritma i implementacije iz rada Mrčela et al. (2017) te originalne implementacije dane u radu Beller et al. (2013).

Duljina ulaznog niza [znak]	Naša implementacija [KiB]	Implementacija Mrčela et al. (2017) [KiB]	Originalna implementacija implementacija [KiB]
1000	131	3476	9532
5000	201	3732	12308
10000	367	3592	12320
100000	3342	6768	12360
250000	6905	11992	12848
500000	12048	20880	13816
750000	18095	30736	14564
1000000	23122	38532	15216
4639211	75973	178172	43516

4 Zaključak

Rezultati ovog studentskog rada, odnosno implementacije dvaju algoritama opisanih u radu Beller et al. (2013) pokazali su se točnima, ali ne i uspješnima jer je zbog neodgovornosti autora i kasnog početka propušteno optimirati programske kodove, te poboljšati implementaciju.

Napredak u implementaciji može se postići korištenjem vlastitih struktura za bitvektor i red umjesto onih danih u standardnoj biblioteci jezika C++ te bi se time rezultati poboljšali i do nekoliko redova veličine, ako ne i bili brži od prošlogodišnje implementacije. Originalna implementacija je pak vrlo vjerojatno koristila dodatne optimizacije koda te bi njene performanse bilo teško dostići.

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.

L. Mrčela, A. Škaro, i A. Žužul. *Računanje najduljeg zajedničkog prefiksa temeljeno na BWT*. 2017.