SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Implementacija FM-indeks algoritma

Ivan Borko, Sofia Čolaković, Florijan Stamenković
Voditelj: *doc. dr. sc. Mirjana Domazet Lošo*

Zagreb, siječanj 2015.

SADRŽAJ

1. Uvod i problematika **1**

2. FM-indeks algoritam **2**

2.1. Burrows-Wheeler transformacija (BWT) 2

2.1.1. LF-mapiranje 3

2.1.2. Rekonstrukcija originala 4

2.1.3. Pretraživanje 5

2.2. Složenost pretraživanja 6

2.2.1. Vremenska složenost 6

2.2.2. Memorijska složenost 7

3. Stablo valića **8**

3.1. Definicija stabla valića 8

3.2. Rangiranje korištenjem stabla valića 9

3.3. Vremenska i memorijska složenost stabla valića 9

3.3.1. Vremenski konstantno binarno rangiranje 9

3.3.2. Vremenska složenost rangiranja stablem valića 10

3.3.3. Memorijska složenost stabla valića 10

4. Implementacija i testiranje **11**

4.1. Implementacija 11

4.2. Testiranje 12

4.2.1. Oblici testiranja 12

4.2.2. Rezultati 13

5. Zaključak **17**

6. Literatura **18**

1. Uvod i problematika

Pretraživanje teksta česta je praktična potreba mnogih informacijskih sustava. Pod terminom "pretraživanje teksta" podrazumijevamo pronalazak svih pojavljivanja nekog niza znakova Q (engl. *query*) unutar drugog niza znakova S (engl. *string*). Tipično se rezultat pretraživanja R formulira kao niz indeksa (rednog broja znaka) unutar niza S na kojem počinje pojavljivanje niza Q. Primjerice, za niz S = *"Žuti pas je opasan kad je opasan remenom oko pasa"* i niz Q = *"pas"* rezultati pretraživanja su R = *{6, 14, 28, 46}*.

U području bioinformatike pretraživanje teksta koristi se za pronalazak sekvenci unutar zadanog genoma. Specifičnost bioinformatičkog pretraživanja jest da su nizovi koji se pretražuju iznimno velike duljine. Primjerice, ljudski genom tipično sadrži oko 3.3×10^9 znakova, što bi otisnuto na A4 stranice fontom veličine 10pt rezultiralo s otprilike milijun stranica.

Postoje mnogi algoritmi pretraživanja teksta koji na jednostavan način ispunjavaju definirane zahtjeve. Iz perspektive računalne složenosti algoritama, nije teško implementirati pretraživanje teksta koje radi u linearnom vremenu[1]. Nažalost, za nizove vrlo velike duljine linearno vrijeme znači praktično predugo trajanje pretraživanja. Otud potreba, pogotovo u području bioinformatike, za vremenski sub-linearnim algoritmima pretraživanja. Istovremeno, potrebno je memorijske zahtjeve algoritma pretraživanja zadržati što manjima.

U ovom projektu razmatramo implementaciju pretraživanja teksta koja se bazira na konceptu FM-indeksa. Konkretna implementacija bazira se na binarnim stablima valića (engl. *wavelet-trees*). Teoretsko razmatranje i praktično testiranje pokazuju da ovakva implementacija pretraživanja ima vremenski sub-linearnu složenost, uz prihvatljivu memorijsku složenost.

[1] Ako nije drukčije navedeno pri razmatranju složenosti pretraživanja uvijek govorimo o složenosti s obzirom na duljinu pretraživanog niza S.

2. FM-indeks algoritam

"Indeksiranje" teksta označava generiranje struktura podataka koje su podrška efikasnom pretraživanju. Za velike tekstove poželjno je da indeks bude memorijski efikasan. FM-indeks [1] pristup je indeksiranju koji ispunjava zahtjeve memorijske efikasnosti i sub-linearnog vremena pretraživanja. Prije nego definiramo FM-indeks, potrebno je razmotriti podatkovne strukture i algoritme koji ga sačinjavaju.

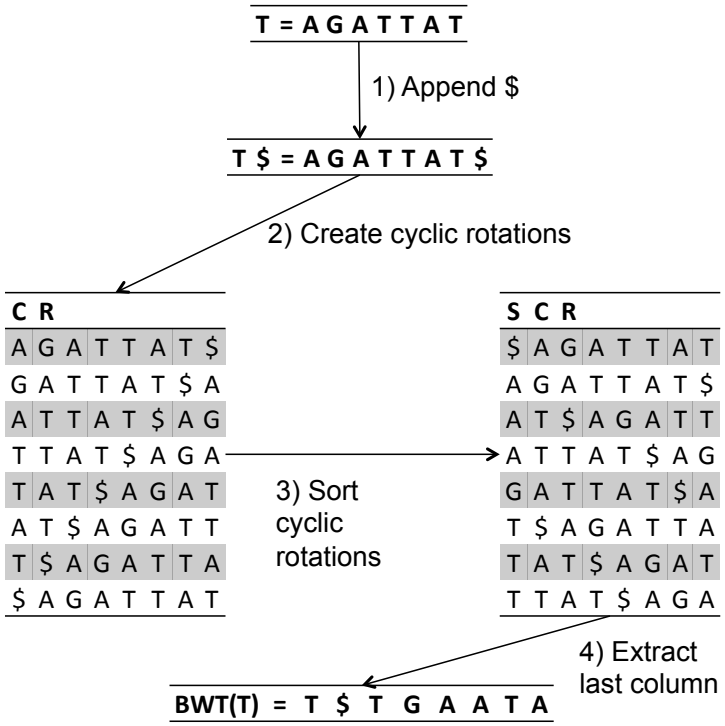
2.1. Burrows-Wheeler transformacija (BWT)

Burrows-Wheeler transformacija [2] transformira niz znakova na način koji će omogućiti efikasnu pohranu i brzo ptreživanje. BWT transformirani niz originalnog teksta T označavati ćemo sa T^{BTW}. Transformacija se provodi sljedećim koracima:

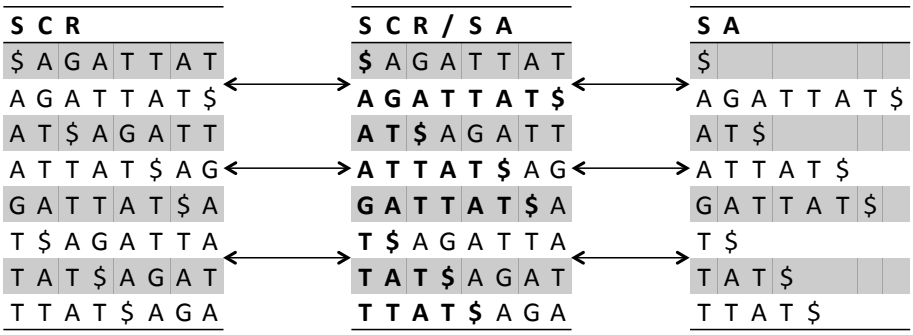
1. Poseban znak '$' koji je leksikografski manji od svih ostalih znakova se dodaje na kraj niza T
2. Cikličkim rotacijama niza T dobiva sa se skup nizova koji čini tablicu CR (engl. *cyclic rotation*)
3. Tablica CR se leksikografski sortira u tablicu SCR
4. Posljednji stupac tablice SCR (čitan odozgo prema dolje) čini rezultat T^{BTW}

Opisani postupak ilustriran je za niz T = *"AGATTAT"* na slici 2.1, preuzetoj iz rada [3].

Bitno je primjetiti kako je BWT transformacija niza srodna sufiksnoj listi SA (engl. *suffix array*). Sufiksna lista je struktura podataka koja se često koristi u algoritmima nad tekstom. Njenu formulaciju nećemo detaljno objašnjavati, materijali na temu su široko dostupni. Sličnost između BWT transformacije i SCR tablice korištene u BWT transformaciji ilustrirana je slikom 2.2.



Slika 2.1: Primjer algoritma BWT transormacije niza $T = AGATTAT$



Slika 2.2: SCR tablica BWT tranfsformacije i sufiksna lista za niz $T = AGATTAT$

2.1.1. LF-mapiranje

LF-mapiranje (engl. *last-to-first mapping*) opisuje relaciju između posljednjeg stupca *SCR* tablice (označenog *L*) i prvog stupca te iste tablice (označenog *F*)[1]. LF-mapiranje postulira da *i*-to pojavljivanje znaka *c* unutar stupca *L* korespondira *i*-tom pojavljivanju tog istog znaka unutar stupca *F*. Pri tome "korespondira" znači da se radi o istom znaku unutar originalnog niza *T*. Jednostavan dokaz ovog iskaza moguće je naći u [3].

Brza implementacija LF-mapiranja (konstantne vremenske složenosti) može se ostvariti korištenjem dviju pomoćnih tablica. Tablica prefiksnih suma *C* (engl. *prefix-sum*

[1] Primjetimo da je *L* isti stupac koji definira BWT transformaciju.

table) niza T za svaki znak c pohranjuje broj znakova u T koji su manji od c. Tablica pojavljivanja Occ (engl. *occurrence table*) pohranjuje informaciju koliko puta se neki znak c pojavio u nizu T do pozicije i (isključujući znak točno na poziciji i). Korištenjem tablica C i Occ LF-mapiranje za T^{BWT} (koji odgovara stupcu L) računa se na sljedeći način, za znak c na poziciji i:

1. Pronađi broj pojavljivanja c u T^{BWT} do pozicije i unutar tablice Occ
2. Pronađi broj znakova manjih od c u T^{BWT} unutar tablice C
3. Zbroj pronađenih vrijednosti je indeks korespondirajućeg znaka u stupcu L

2.1.2. Rekonstrukcija originala

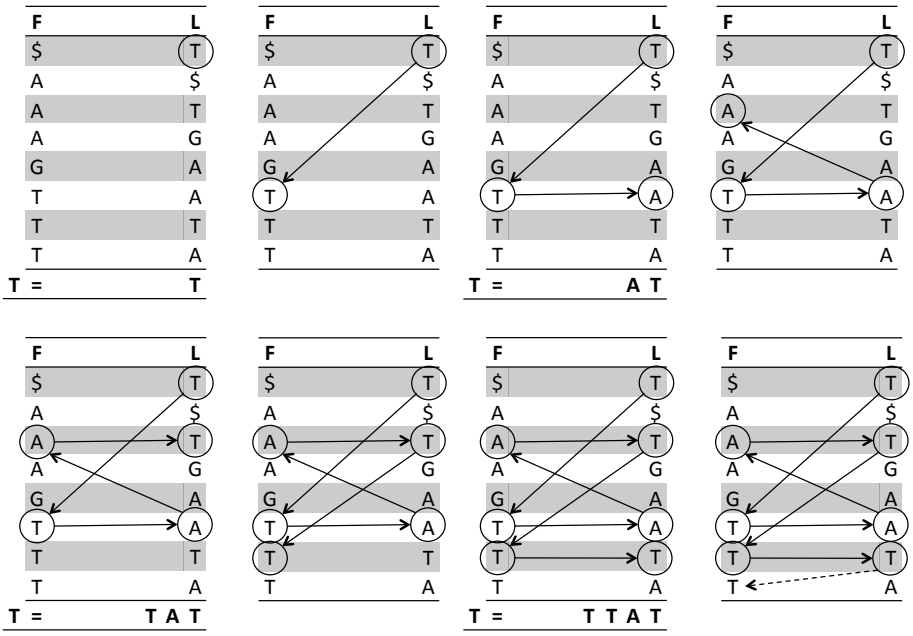
Na temelju transformiranog niza T^{BWT} moguće je rekonstruirati originalni niz T korištenjem LF-transformacije. Postupak je jednostavan, ako imamo na umu definiciju LF-transformacije i činjenicu da je prvi znak u T^{BWT} zasigurno posljednji znak niza T (ovo proizlazi iz činjenice da smo pri postupku BWT transormacije na početak niza umetnuli znak '$').

Rekonstrukcija niza T obavlja se unatrag, od posljednjeg znaka prema prvom. Postupak je sljedeći:

1. Prvi znak iz T^{BWT} je posljednji znak iz T, zabilježimo ga
2. LF-transformacijom pronađimo F-indeks posljednjeg zabilježenog znaka [2]
3. Dobiveni F-indeks u T^{BWT} nizu ukazuje na znak koji u nizu T prethodi posljednjem zabilježenom znaku (posljedica rotacije pri konstrukciji SCR)
4. Zabilježimo znak na F-indeks poziciji niza T^{BWT} u rekonstrukciju
5. Ako posljednji zabilježeni znak nije '$', vraćamo se na korak 2.

Opisani postupak vizualiziran je na slici 2.3. Bitno je primjetiti kako pri rekonstrukciji originala nismo koristili ništa osim transformiranog niza T^{BWT} i LF-mapiranja (koje se ostvaruje tablicama C i Occ).

[2] U ovom trenutku nemamo tablicu SCR niti stupac F, zanima nas samo indeks.



Slika 2.3: Primjer rekonstrukcije originala iz transformiranog niza T^{BWT}

2.1.3. Pretraživanje

Transformirani niz T^{BWT} može se koristiti i za pretraživanje originalnog teksta T. Algoritam pretraživanja vrlo je sličan rekonstrukciji originala. Bazira se zapravo na poznatom načinu pretraživanja sufiksnih polja. Kao što je već spomenuto, BWT transformacija i sufiksno polje nekog niza su skoro ekvivalentni.

Algoritam pretraživanja pojavljivanja niza Q unutar niza T na temelju BWT transforacije T^{BWT} bazira se na sljedećim konceptima:

- Pretraživanje se vrši po znakovima Q unatrag (od zadnjeg prema prvom)
- Prate se početni i krajnji indeks sufiksa (*SCR* tablice) unutar kojih su moguće podudarnosti
- U svakom koraku (procesiranom znaku iz P) se područje mogućih podudarnosti smanjuje

Ako početni indeks označimo *ps* (engl. *pointer start*), a krajnji indeks *pe* (engl. *pointer end*), tada se algoritam izvršava sljedećim koracima:

1. Inicijaliziraj indekse *ps* i *pe* tako da obuhvaćaju cijelu tablicu *SCR*
2. Odaberi prvi neobrađeni znak c iz niza Q, počevši od kraja
3. Unutar područja među indeksima nađi prvo i posljednje pojavljivanje znaka c [3]

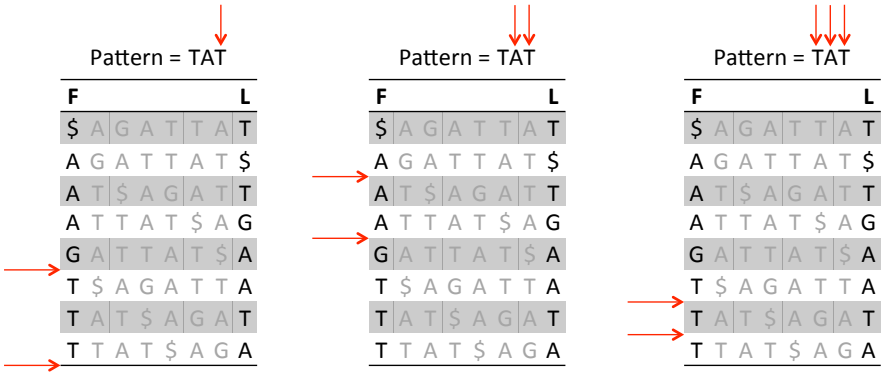
[3] Učinkovita implementacija pretraživanja ne izvršava ovaj korak, navodimo ga samo radi opisa rada

4. Izračunaj *L*-indekse nađenih pojavljivanja koristeći LF-mapiranje

5. Dobiveni *L*-indeksi su nove vrijednost indeksa *ps* i *pe*

6. Ako postoje neobrađeni znakovi u *Q*, vrati se na korak 2.

Navedeni koraci algoritma formulirani su kako bi bili što jasniji. Opis prikladniji za izravnu računalnu implementaciju može se naći u radu [3]. Slika 2.4 prikazuje korake pretraživanja za $Q = TAT$.



Slika 2.4: Primjer pretraživanja teksta korištenjem transformiranog niza T^{BWT}

2.2. Složenost pretraživanja

Glavni razlog za razvoj FM-indeksa je povećanje efikasnosti pretraživanja dugih nizova (poput genoma). U ovom poglavlju će se razmotriti teoretska složenost pretraživanja korištenjem FM-indeksa. Pri tome nas zanima složenost korištenja izgrađenog indeksa za neki niz, a ne složenost stvaranja indeksa.

2.2.1. Vremenska složenost

Razmotrimo vrijeme pronalaska pojavljivanja niza *Q* unutar niza *T*, na način opisan u poglavlju 2.1.3. Iz algoritma je vidljivo da je potreban jedan korak za svaki znak u *Q*, dakle pretraživanje ima linearnu složenost s obzirom na duljinu *Q*. Svaki od tih koraka svodi se na dvije operacije LF-mapiranja, koje ima konstantnu vremensku složenost (ne ovisi o duljinama nizova *T* i *Q*).

algoritma.

Važna napomena je da prolazak kroz znakove niza Q može biti prekinut u slučaju da se utvrdi da nema pojavljivanja Q unutar T, što se može desiti u bilo kojem trenutku prolaska kroz Q.

Dakle, vremenska složenost pretraživanja je generalno linearna s obzirom na duljinu niza Q. Ovaj rezultat je pogodan za pretraživanje dugih nizova T, jer ne ovisi o njihovoj duljini.

2.2.2. Memorijska složenost

Memorijska (prostorna) složenost označava utrošak memorije na potporne strukture podataka FM-indeksa, s obzirom na niz T. Podatkovne strukture koje se koriste u pretraživanju su transformirani niz T^{BWT} (odnosno sufiksno polje) te tablice C i Occ korištene za LF-mapiranje.

Sufiksno polje ima jednak broj elemenata kao i originalni niz T. U tom smislu je memorijska složenost polja linearna s obzirom na duljinu T. Isto vrijedi i za transformirani niz T^{BWT}. Prostorni utrošak sufiksnog polja može se smanjiti na više načina, a niz T^{BWT} je često oblika pogodnog za komprimiranje. Obje tehnike izlaze izvan okvira ovog rada, mi koristimo postojeće implementacije sufiksnog polja koje su memorijski optimizirane.

Tablica prefiksnih suma C pohranjuje po jedan cijeli broj za svaki element abecede niza T. Iako je u teoriji broj znakova abecede ograničen samo duljinom niza T, u praksi je najčešće vrlo malen te tablica C ne predstavlja problem u kontekstu zauzeća memorije.

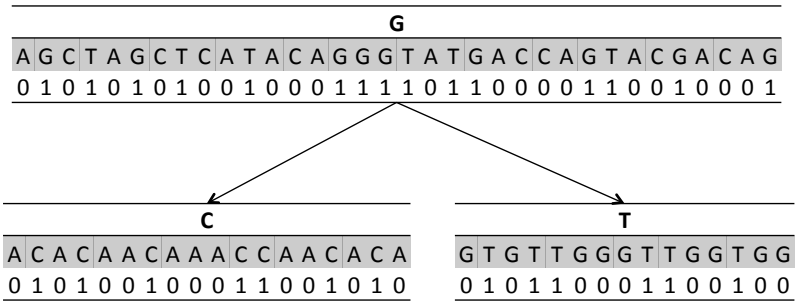
Tablica pojavljivanja Occ (u naivnoj implementaciji) pohranjuje po jedan cijeli broj za svaku poziciju niza T, za svaki element abecede T. Primjećujemo linearnu složenost s obzirom na duljinu niza T. S obzirom na potencijalno ogromne duljine nizova koje želimo moći pretraživati, ovo je problem. Postoji više pristupa "kodiranja" tablice Occ. Unutar ovog rada, zadatak je implementirati FM-indeks korištenjem stabla valića.

3. Stablo valića

Stablo valića (engl. *wavelet tree*), definirano u [4], koristimo za efikasnu implementaciju tablice pojavljivanja *Occ*, koja se koristi za LF-mapiranje unutar FM-indeksa. *Occ* tablica daje informaciju o broju pojavljivanja znaka c unutar niza T do pozicije i (ne uključujući znak na poziciji i). Ova operacija naziva se "rangiranje", možemo reći da tražimo *rank* znaka c u T do pozicije i, odnosno *rank(T, c, i)*.

3.1. Definicija stabla valića

Stablo valića kodira niz T u binarno stablo. Svaki čvor stabla ima pripadajući znak (prijelomnu točku p) i binarni niz (niz sačinjen od nula i jedinica). Nule u binarnom nizu čvora označavaju znakove originalnog niza koji su manji od znaka prijelomne točke p, a jedinice znakove koji su veći ili jednaki (pri tome se u čvoru ne pohranjuje originalni niz, već samo binarni). Znakovi originalnog niza označeni nulama sačinjavaju lijevo dijete čvora, a oni označeni jedinicama desno. Čvorovi se kreiraju samo ako sadrže dva ili više različita znaka (za niz proizvoljne duljine začinjen od samo jednog znaka se ne stvara stablo). Primjer binarnog stabla za niz "AGCTAGCTCATACAGGGTATGACCAGTACGACAG" prikazan je na slici 3.1.



Slika 3.1: Primjer stabla valića za niz "AGCTAGCTCATACAGGGTATGACCAGTACGACAG". Znakovi originalnog niza prikazani u čvorovima stabla služe samo za ilustraciju, konkretna implementacija sadrži samo binarne nizove i znakove koji su prijelomne točke čvorova.

3.2. Rangiranje korištenjem stabla valića

"Rank" znaka *c* unutar niza *T* do pozicije *i* je broj pojavljivanja tog istog znaka u prvih *i* znakova niza *T*. Operacija rangiranja može se efikasno ostvariti korištenjem stabla valića.

Pretpostavimo da imamo vremenski konstantnu implementaciju binarnog rangiranja. Binarno rangiranje ima istu definiciju kao općenito rangiranje, ali se obavlja nad binarnim nizovima, dakle svodi se na prebrojavanje jedinica u binarnom nizu. Implementacija vremenski konstantne složenosti biti će objašnjena u nastavku.

Svaki čvor stabla ima pripadnu prijelomnu točku *p* i binarni niz. Koristeći binarno rangiranje za svaki čvor stabla valića lako je utvrditi broj znakova manjih ili većih od *p*, u prvih *i* znakova čvora. Razmatramo dvije situacije: znak *c* koji rangiramo može biti leksikografski manji od *p* ili veći-ili-jednak. U prvoj situaciji broj nula binarnog ranga čvora po *p* govori koliko znakova lijevog djeteta čvora trebamo pretražiti, u drugoj situaciji broj jedinica govori koliko znakova desnog djeteta trebamo pretražiti. Spuštajući se tako kroz stablo dolazi trenutak kada relevantno dijete trenutnog čvora ne postoji, tada je trenutni broj pretraživanja istovremeno i konačni rezultat. Opširnija objašnjenja rangiranja korištenjem stabla valića široko su dostupna[1].

3.3. Vremenska i memorijska složenost stabla valića

3.3.1. Vremenski konstantno binarno rangiranje

Za ostvarivanje brzog rangiranja korištenjem stabla valića potrebna je sposobnost brzog rangiranja binarnog niza. Problem rangiranja binarnog niza jednostavniji je jer postoje samo dva znaka abecede. Pri tome je potrebno rangirati samo jedan znak (primjerice znak 1) jer se rang drugog znaka dobiva kao *rank(T, 0, i) = i - rank(T, 1, i)*. Konstantna vremenska složenost ovakvog rangiranja ostvaruje se grupiranjem niza u "pretince" duljine *k*. Za svaki pretinac se prati broj jedinica sadržanih u svim prethodnim pretincima. Time se za svaki *i* rangiranje obavlja određivanjem pretinca kojoj *i* pripada, običnim prebrojavanjem jedinica do pozicije *i* samo unutar tog pretinca, te pribrajanjem broja jedinica sadržanih u prethodnim pretincima. Time je prebrojavanje (iteracija po znakovima) ograničeno na veličinu pretinca, ne ovisi više o duljini originalnog niza. Ovakvo rangiranje stoga ima konstantnu vremensku složenost s obzirom na duljinu originalnog niza.

[1] Primjerice na http://alexbowe.com/wavelet-trees/

Konkretna implementacija opisanog rangiranja je malo složenija jer se zbog memorijske učinkovitosti koriste pretinci i "super-pretinci", ali je konceptualno identična i vremenski jednako efikasna.

3.3.2. Vremenska složenost rangiranja stablem valića

Stablo valića može prestavljati proizvoljan niz, proizvoljne abecede. Pri tome se ono razlaže binarno po elementima abecede. Rangiranje se svodi na jedan dubinski prolaz kroz stablo. Dubina binarnog stabla je jednaka $log_2(|\Sigma|)$, gdje je Σ skup znakova abacede, što istovremeno određuje vremensku složenost rangiranja. Pošto je binarno rangiranje u svakom čvoru konstantne složenosti, konačna složenost rangiranja je logaritamska s obzirom na veličinu abecede. Pošto su u većini primjena korištene abecede relativno male (u bioinformatici radi se o 4 ili 5 znakova), rangiranje korištenjem stabla valića ima iznimno dobre performanse.

3.3.3. Memorijska složenost stabla valića

Indeksiranje niza znakova u pravilu povećava memorijsku složenost kako bi se smanjila vremenska. Pri tome je poželjno da memorijsko povećanje nije preveliko, pogotovo kada se radi sa iznimno velikim nizovima.

Stablo valića pohranjuje binarne varijante originalnog niza. Pri tome korijenski čvor pohranjuje binarni niz dulje jednake originalnom nizu. Djeca korijenskog čvora sadrže binarne nizove koji zajedno imaju duljinu jednaku originalnom nizu (može se isčitati iz definicije, ilustrirano na slici 3.1). Ista stvar se dešava sa daljnjim grananjem stabla. Stoga svaka razina stabla (čvorevi jednako udaljeni od korijenskog) sadrži binarne nizove čija ukupna duljina je jednaka originalnom nizu. Stablo ima $log_2(|\Sigma|)$ razina, gdje je Σ skup znakova abacede. Stoga je konačna memorijska složenost stabla $n * log_2(|\Sigma|)$ bitova, gdje je n broj znakova originalnog niza. Sličnu složenost imaju neki algortmi kompresije, radi se dakle o relativno niskoj memorijskoj zahtjevnosti.

4. Implementacija i testiranje

4.1. Implementacija

Zbog brzine izvođenja FM-indeks je implementiran u C++ programskom jeziku, osim skripti za pokretanje generiranja sintetskih podataka i samih skripti za generiranje sintetskih podataka, koje su pisane u *bash*-u, odnosno *pythonu* verzije 2. Pojedini algoritmi su prije toga izvedeni i u Python programskom jeziku, radi jednostavnosti implementacije i testiranja, te potom optimalno prevedeni u C++. C++ kod smo prevodili pomoću `gcc` prevodioca verzije 4.9, koji je praktički standard na *unix* okruženjima. Kod prevođenja je korištena naredba *-O2* prevodica.

Implementacija se sastoji od BW-transformacije i potpornih struktura za LF-mapiranje. BW-transformacija je implementirana kao dokaz koncepta i za testiranje ispravnosti algoritma. Pri testiranju složenosti pretaživanja koristi se postojeća implementacija sufiksnog polja[1]. Tablica prefiksnih suma *C* implementirana je trivijalno (ništa drugo nije potrebno), a tablica pojavljivanja *Occ* je implementirana trivijalno i u obliku stabla valića. Napisani su UnitTestovi pojedinih funkcionalnosti, kao i testovi za cjelokupni FM-indeks, što uključuje evaluaciju vremenske i memorijske složenosti pretraživanja. Pri testiranju složenosti uspoređuju se trivijalna implementacija *Occ* tablica i stablo valića.

Cijeli kod je objavljen na javnom *GitHub* repozitoriju, na adresi `https://github.com/iborko/fmindex`. Cijeli kod se može preuzeti naredbom:

```
# git clone -recursive https://github.com/iborko/fmindex
```

Napisana je i *makefile* datoteka koja ubrzava cijeli proces prevođenja te omogućava zasebnu izgradnju izvršne datoteke s testovima (*test*) i izvršne datoteke sa samom implementacijom FM indeksa (*fmindex*). Automatizirani proces prevođenja koristeći *Makefile* alat se pokreće upisivanjem naredbe:

[1] https://sites.google.com/site/yuta256/sais

```
# make
```

u korijenskom direktoriju repozitorija. Može se pozvati i zasebno prevođenje same implementacije:

```
# make fmindex
```

ili samo testova:

```
# make test
```

Nakon prevođenja izvršne će se datoteke nalaziti u mapi `bin`. Opis korištenja:

```
# fmindex <sequence> <reads> [<occurrence_table> [<bucket_size>]]
```

sequence putanja do sekvence (engl. *sequence*) na kojoj se izvodi pretraživanje, u FASTA[2] formatu

reads putanja do očitanja (engl. *reads*) koji će se tražiti u sekvenci, FASTQ[3] formatu

occurrence_table može biti *0* (trivijalna implementacija tablice pojavljivanja), *1* (implementacija tablice pojavljivanja pomoću stabla valića)

bucket_size veličina pretinca kod implementacije stabla valića

Primjer pokretanja:

```
# fmindex Esch_coli_536.fna Esch_coli_536_reads.fq 1 40
```

Za svako očitanje iz `<reads>` datoteke, program generira dvije linije. Prva linija je header tog očitka iz `<reads>` datoteke, a druga linija je niz indeksa položaja tog očitka u `<sequence>` datoteci. Indeksi su odvojeni razmakom. Izlaz se ispisuje na standardni izlaz (engl. *stdout*).

4.2. Testiranje

4.2.1. Oblici testiranja

UnitTestovi pojedinih funkcionalnosti (primjerice stabla valića i binarnog ranka) pisani su kako bi se utvrdila njihova ispravnost. Pri tome se testiralo nad sintetiziranim, nasumičnim podatcima. Uspoređuju se trivijalne implementacije (jednostavne za napisati, ali vremenski i memorijski neefikasne) sa produkcijskim algoritmima. U trenutku

[2] http://en.wikipedia.org/wiki/FASTA_format
[3] http://en.wikipedia.org/wiki/FASTQ_format

pisanja UnitTestovi indiciraju da su sve produkcijske funkcionalnosti ispravno implementirane.

Unit testovi su napisani koristeći *Catch*[4] modul za C++ jezik koji se nalazi u mapi `include/catch`.

Testiranje cjelokupnog FM-indeksa s obzirom na vremensku i memorijsku složenost pretraživanja izvršeno je nad sintetiziranim podatcima, kao i nad javno dostupnim genom bakterije *Escherichia coli*[5].

Skripta `test_run.sh` se koristi za pokretanje programa na nekoj testnoj sekvenci. Testne sekvence se nalaze u mapi `test_data`. Skripta najprije generira skupove od 1000, 5000, 10000, 50000, 100000, 500000 i 1000000 očitanja (engl. *reads*) duljine 80 znakova. Svako generirano očitanje ima u FASTQ headeru indeks početka u originalnoj sekvenci, tako da se kasnije može usporediti s pripadnom .out datotekom. Nakon toga pokreće *fmindex* program nad svakim skupom i ispisuje vršnu potrošnju radne memorije i proteklo vrijeme. Skripta sprema izlaze u .out datoteke u mapu `test_data`.
Primjer:

```
# test_run.sh test_data/Esch_coli_536.fna
```

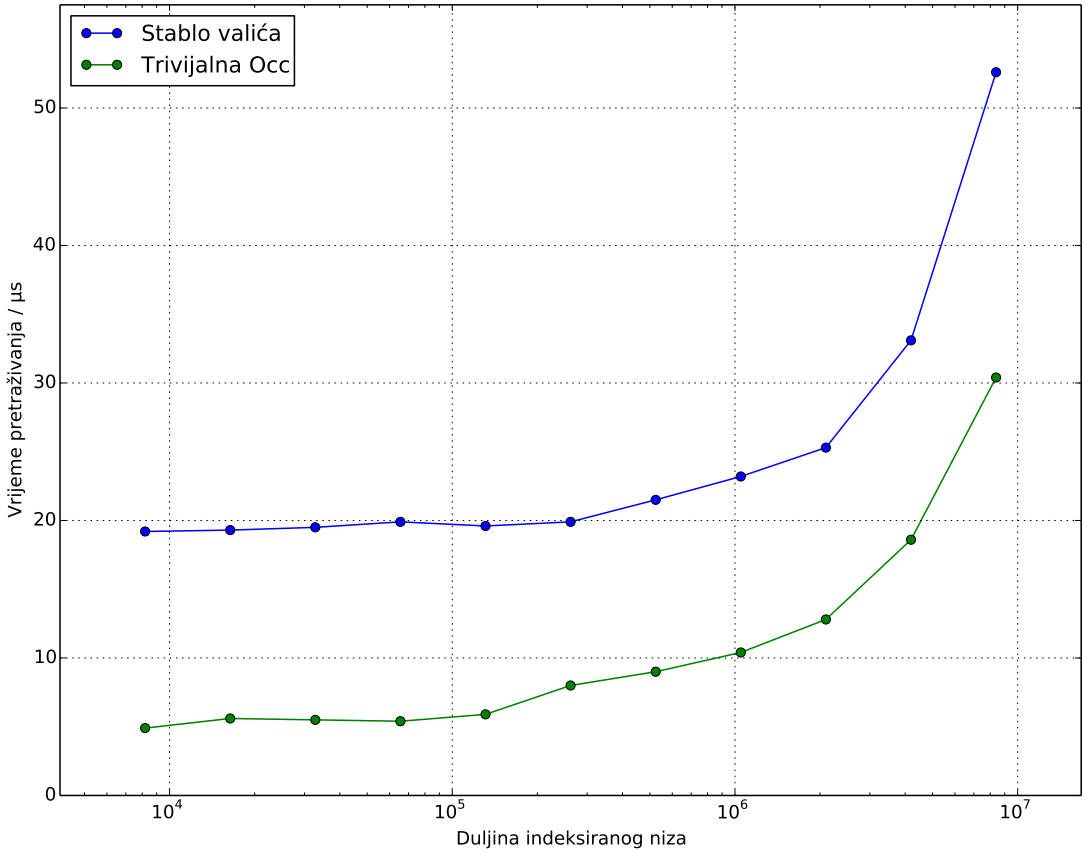
4.2.2. Rezultati

Vremenska složenost

Slika 4.1 prikazuje trajanje pretraživanja u ovisnosti o duljini indeksiranog niza. Testiranje je rađeno nad sintetskim podatcima. Prikazane su brzine pretraživanja za FM-indeks implementran korištenjem trivijalne implementacije *Occ* tablice, kao i brzine za indeks koji koristi stablo valića. U implementaciji sa stablom valića su pretinci korišteni za binarno rangiranje veličine 20. Pretraživana su pojavljivanja očitaka duljine 80 znakova unutar nizova duljina u rasponu $[5 * 10^3, 10^7]$ (svaka potencija broja 2 unutar tog raspona). Rezultirajuće vrijeme je trajanje pojedine pretrage dobiveno kao prosjek od 10^5 pretraga.

Rezultati su pomalo iznenađujući, s obzirom na teorijsko razmatranje konstantne složenosti pretraživanja s obzirom na duljinu indeksiranog niza. U [3] je observiran isti fenomen, a objašnjen je kao posljedica hardverskog baratanja priručnom memorijom. Pri indeksiranju nizova vrlo velike duljine, memorijski zahtjevi strukture indeksa

[4] https://github.com/philsquared/Catch
[5] http://bacteria.ensembl.org/index.html



Slika 4.1: Vrijeme pretraživanja u ovisnosti o duljini indeksiranog niza, pretražuju se očitci duljine 80 znakova. Testiranje FM-indeksa sa trivijalnom *Occ* tablicom, naspram indeksa sa stablom valića. Pretinci za binarno rangiranje u stablu valića su veličine 20.

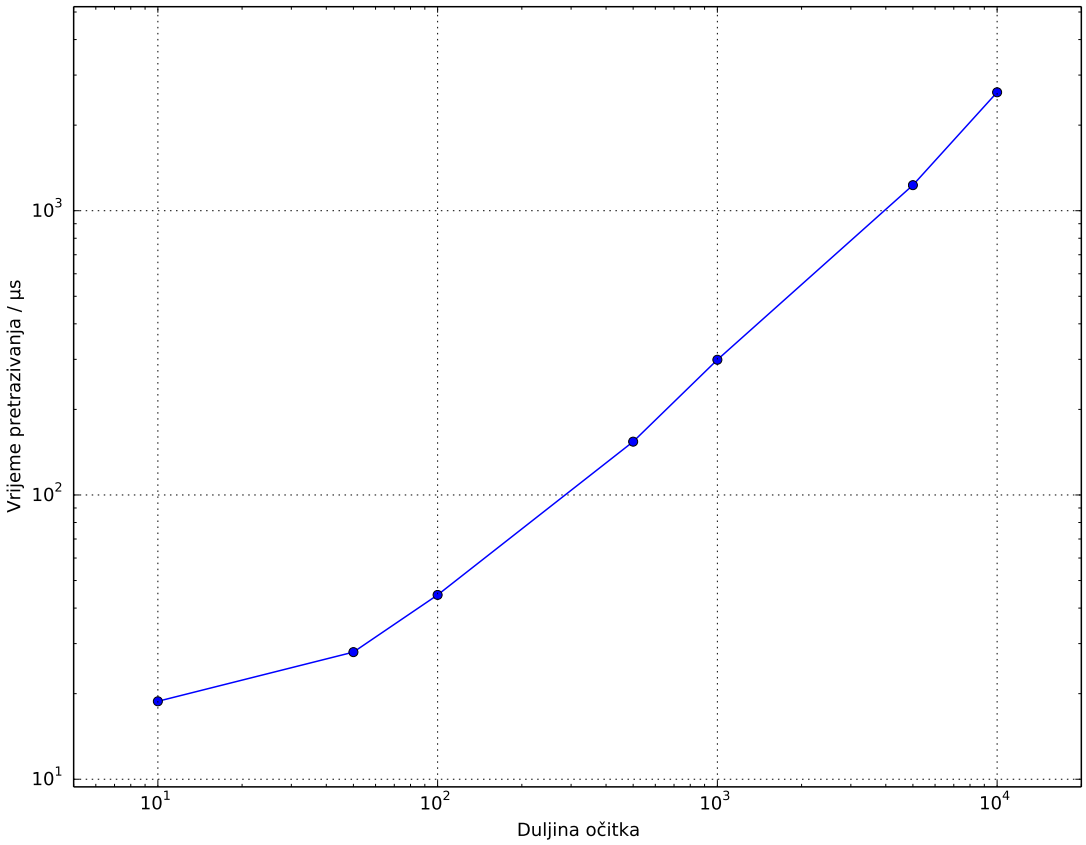
postaju preveliki da bi te strukture stale u priručnu memoriju. Prijenos podataka preko raznih razina priručne memorije vremenski je zahtjevan i stoga snažno utječe na performase pretraživanja.

Slika 4.2 prikazuje trajanje pretraživanja s obzirom na duljinu očitka (engl. *read-length*) koji se traži. Testiranje je rađeno nad genomom bakterije *Escherichia coli*, duljine $4.9 * 10^6$ znakova. FM-indeks koristi stablo valića s pretincima binarnog rangiranja veličine 20. Rezultati su srednje vrijednosti 10^5 pretraga.

Rezultati su u skladu sa teorijskom analizom složenosti. Vidimo linearni porast trajanja pretrage s obzirom na duljinu očitka.

Memorijska složenost

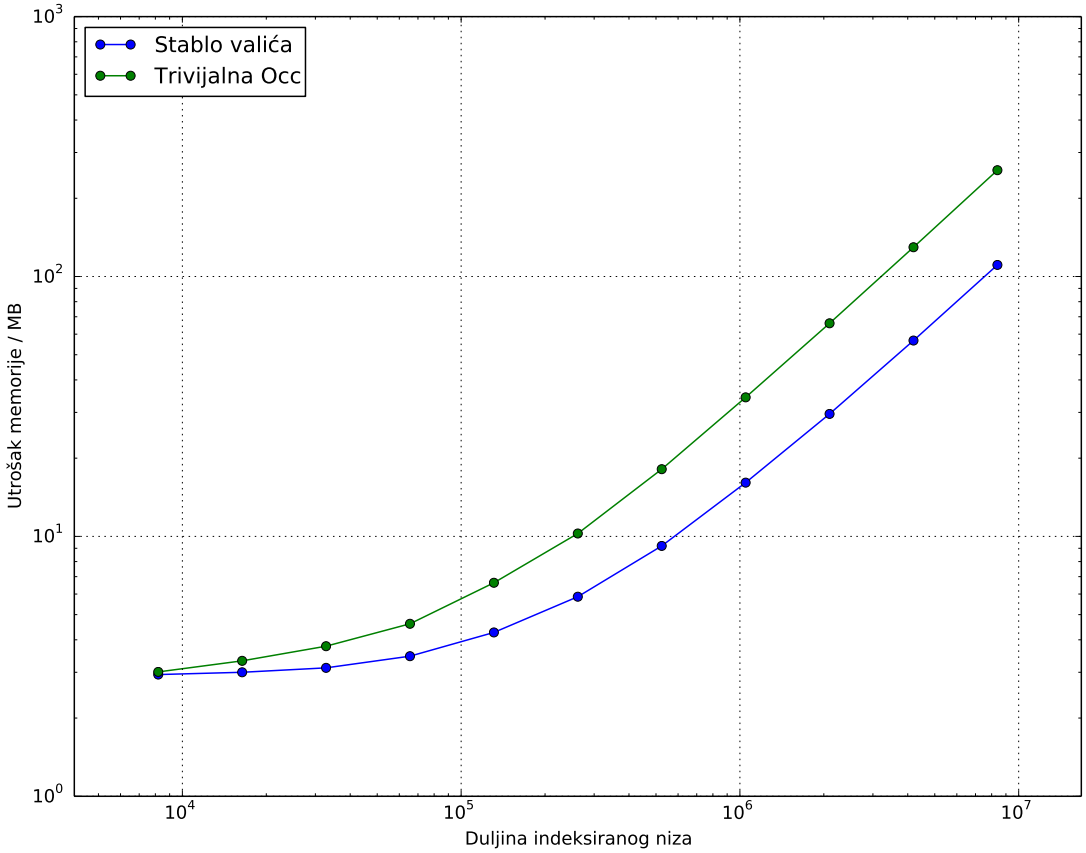
Utrošak memorije na potporne strukture FM-indeksa prikazan je na slici 4.3. Testiranje je rađeno nad sintetskim podatcima. Prikazan je utrošak memorije za FM-indeks implementran korištenjem trivijalne implementacije *Occ* tablice, kao i utrošak za in-



Slika 4.2: Vrijeme pretraživanja u ovisnosti o duljini očitka. Pretražuje se genom bakterije *Escherichia coli*, duljine $4.9 * 10^6$ znakova. Pretinci za binarno rangiranje u stablu valića su veličine 20.

deks koji koristi stablo valića. U implementaciji sa stablom valića su pretinci korišteni za binarno rangiranje veličine 20. Mjerena je potrošnja memorije cijelog programa.

Iz grafa 4.3 vidljivo je kako je utrošak memorije s obzirom na duljinu indeksiranog niza asimptotski linearan (primjetimo da su obje osi grafa logaritamske skale). Vidljivo je kako implementacija FM-indeksa bazirana na stablu valića koristi manje od pola memorije trivijalne implementacije, za dugačke nizove. Rezultati su u skladu sa teorijskim razmatranjem memorijske potrošnje FM-indeks struktura podataka.



Slika 4.3: Utrošak memorije čitavog programa u ovisnosti o duljini indeksiranog niza. Testiranje FM-indeksa sa trivijalnom *Occ* tablicom, naspram indeksa sa stablom valića. Pretinci za binarno rangiranje u stablu valića su veličine 20.

5. Zaključak

Velike količine informacija koje su dio današnjeg svakodnevnog života definiraju potrebu za učinkovitim načinom pretraživanja. Istovremeno, znanost u području genetskog istraživanja radi sa tekstualnim nizovima enormnih duljina, nad kojima je potrebno vršiti velike količine pretraga. Efikasni algoritmi pretraživanja teksta, koji ne ovise o duljini tog teksta, praktički su neophodni.

Unutar ovog rada implementirali smo FM-indeks algoritam i potporne podatkovne strukture. Radi se o indeksu koji omogućava pretraživanje teksta brzinom neovisnom o duljini tog teksta, već samo o duljini niza očitka. Istovremeno je memorijski utrošak indeksa prihvatljive veličine. Implementirali smo BW-transformaciju te stablo valića kao osnove FM-indeksa. Isto tako smo kao alternativu BW-transformaciji koristili jednu od poznatijih implementacija sufiksnog polja. Izvršili smo testiranje vremenske složenosti pretraživanja, kao i memorijskog utroška FM-indeksa, koje je potvrdilo očekivane rezultate.

6. Literatura

[1] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. pages 390–398, 2000.

[2] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[3] J. Singer. A wawelet tree based fm-index for biological sequences in seqan. Master's thesis, 2012.

[4] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.