SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2420

Implementacija FM indeksa

Matija Šošić

Zagreb, srpanj 2012.

Zahvaljujem svojoj obitelji na podršci tijekom izrade rada. Također zahvaljujem mentoru Mili Šikiću na ideji te velikoj volji i trudu.

SADRŽAJ

POPIS SLIKA

POPIS TABLICA

1. Uvod

Jedan od najvažnjih resursa današnjice jest znanje. S razvojem tehnologije moguće je u sve kraćem vremenu prikupiti sve veći broj informacija stoga je prijeko potrebno osigurati sustave koji će podržati njihovu učinkovitu obradu i spremanje. S obzirom na ogromnu količinu podataka koje obrađuju, vrlo je bitno da i sami sustavi budu kako vremenski tako i memorijski učinkoviti. Često se pribjegava spremanju ulaznih podataka u sažetom obliku, za što je do danas razvijeno više obitelji postupaka prikladnih za upotrebu u različitim slučajevima. Također, kako bismo prikupljene podatke mogli iskoristiti nužno je postojanje mogućnosti njihovog pretraživanja. Naravno, algoritam koji će to omogućiti mora se također moći izvršiti u razumnom vremenu uz korištenje ograničene količine memorije.

Problem kojem se obraćamo u ovom radu postavlja oba navedena zahtjeva: potrebno je implementirati strukturu za spremanje i algoritam za pretraživanje zadanog teksta. Po zadavanju uzorka koji može biti riječ, rečenica ili naprosto niz slova bez posebnog značenja želimo znati koliko se puta i na kojem mjestima u tekstu pojavljuje. Sam problem je jednostavno definirati i s njime se većina nas susreće gotovo svakodnevno — tražimo određeni pojam na web-stranici ili u .pdf dokumentu i pri tome koristimo ugrađenu tražilicu koja uglavnom trenutno obrađuje naše zahtjeve. Unatoč maloj količini podataka u obradi, već i u takvim slučajevima se koriste razne metode ubrzavanja. Kako količina podataka i broj upita rastu nasuprot stalnim resursima koji su nam na raspolaganju, potrebno je primijeniti metode koje ih što učinkovitije iskorištavaju.

Rješenje koje smo odlučili implementirati predstavljeno je u radovima autora Ferragine i Manzinia [1]. Strukturu koju koriste nazvali su *FM index* (engl. *Full-text index in Minute space*). Temelji se na korištenju Burrows-Wheeler transformacije [2] kao podloge za sažimanje i sufiksnim poljima [3].

Osim same implementacije *FM indexa*, cilj rada je i primijeniti ga te istestirati na konkretnom problemu sastavljanja genoma. *FM index* je često korišten u tu svrhu i upravo zato smo ga odabrali.

U poglavlju *Metode* problem je detaljnije definiran i predstavljen je način izgradnje strukture *FM index*, kao i algoritmi koji se nakon toga nad njom primjenjuju. U poglavlju *Implementacija* detaljnije opisujemo programsku implementaciju te je priložena gruba skica programskog koda. U poglavlju *Rezultati i diskusija* prezentirani su i provedena je analiza rezultata dobivenih na konkretnom problemu. Zadnje poglavlje *Zaključak* pruža ideje za moguća daljnja istraživanja.

2. Notacija

Tekst koji želimo sažeti i pretraživati označavat ćemo s $T[1, n]$, gdje je n broj znakova u tekstu. Znakovi od kojih je T izgrađen čine abecedu Σ, gdje je Σ skup znakova. $T[i]$ označava i-ti znak u T, a $T[a, b]$ označava podniz od T koji se proteže od a-tog do b-tog znaka, uključivo.
$|A|$ označava kardinalitet skupa A, dok $|w|$ označava duljinu niza w.
Pri spomenu logaritamske funkcije podrazumijeva se da se radi s bazom broja 2 osim ako nije drugačije navedeno.

3. Metode

U ovom poglavlju opisana je ideja algoritma koji se koristi za sažimanje i povrh toga algoritam za brojanje pojavljivanja uzoraka u tekstu. Argumentirano uvodimo nove pojmove i objašnjavamo njihovu ulogu. Kao što je prethodno navedeno, cilj nam je spremiti tekst u sažetom obliku i istovremeno omogućiti određenu funkcionalnost pretraživanja. Također, veličinu strukture koja to omogućava ograničavamo veličinom ulaznog teksta.

3.1. Burrows-Wheeler transformacija

Burrows-Wheeler transformacija (dalje u tekstu i kao BWT) jest reverzibilan postupak transformacije primjenjiv na tekst. Razvili su ga Burrows i Wheeler 1994. godine, a i danas se koristi u mnogim algoritmima za sažimanje, poput algoritma *bzip*. Zanimljiv nam je jer je tako transformirani tekst lakše sažeti, a posjeduje i još neka druga korisna svojstva. Transformacija se provodi u sljedeća četiri koraka:

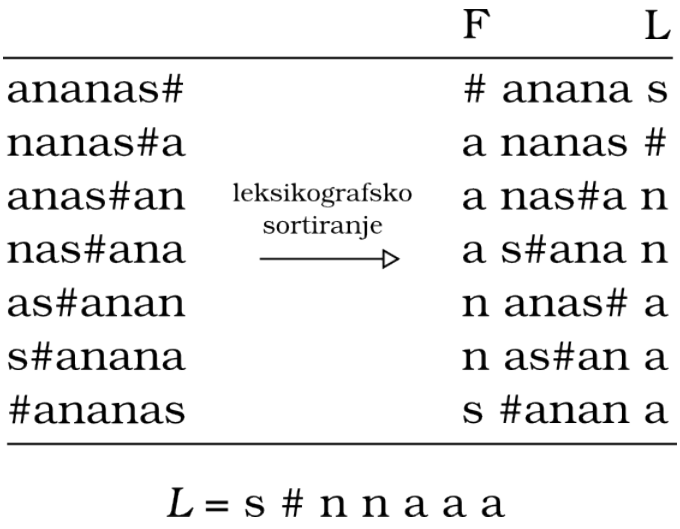
1. Na kraj teksta dodaje se posebni znak # leksikografski manji od svakog znaka iz T.

2. Stvori se konceptualna matrica u kojoj je prvi redak jednak T, a svaki sljedeći je ciklički pomak prethodnog retka za jedno mjesto udesno.

3. Retci tako dobivene matrice se sortiraju u leksikografskom poretku, čime se dolazi do matrice M_T.

4. Transformirani tekst L dobiva se uzimanjem posljednjeg stupca iz M_T.

Prednosti BWT su što je relativno jednostavan – ne uvodi nove znakove u postojeću abecedu (osim znaka iz koraka 1.) Σ već samo vrši permutacije nad T. Ukoliko T sadrži više jednakih podnizova, velika je vjerojatnost da će L sadržavati uzastopne nizove jednakih znakova. Za takve nizove moguće je uvesti kraći način zapisa (spremiti znak



Slika 3.1: Primjer Burrows-Wheeler tranformacije za tekst T = `ananas`.

i broj ponavljanja umjesto eksplicitnog spremanja svakog znaka), i to je osnovna ideja sažimanja.

Također je važno primijetiti vrlo jaku vezu između BWT i sufiksnog polja. Prefiks svakog retka do znaka # matrice M_T je sufiks od T. Zaključujemo da sufiksno polje od T jednoznačno određuje matricu M_T, a time i transformaciju L. To je i uobičajeni način implementacije BWT algoritma, pronaći sufiksno polje od T i na temelju njega jednostavnim postupkom izgraditi L. Ovime smo problem implementacije BWT algoritma sveli na pronalazak i implementaciju odgovarajućeg algoritma za izgradnju sufiksnog polja.

Slika 3.1 ilustrira moć Burrows-Wheeler transformacije. Iako je T jako kratak, namjerno je odabran tako da sadrži ponavljajući podniz, u ovom slučaju `na` koji se ponavlja dva puta. Dobivena transformacija sadrži sve jednake znakove u potpunosti grupirane što je čini idealnom za sažimanje. U stvarnim ulaznim tekstovima koji će biti puno veće duljine takvi podnizovi tipično će se ostvarivati čestom pojavom člana u jeziku, primjerice *the* u engleskom.

Najvažnije svojstvo BWT jest upravo reverzibilnost – obični algoritam za sortiranje može T učiniti idealnim za sažimanje, ali želimo također moći rekonstruirati originalni tekst iz njegovog transformiranog oblika. Uvedimo još nekoliko struktura koje nam pomažu iskoristiti svojstva matrice M_T:

- C[] označava niz duljine $|\Sigma|$ takav da C[c] sadrži broj znakova u L (uključujući i #) leksikografski manjih od znaka c. Ova struktura će nam koristiti u pronalaženju redaka matrice M_T koji započinju određenim znakom.
- Occ(c, q) označava broj pojavljivanja znaka c u u prvih q znakova transformi-

ranog niza *L*. Ovo je središnja struktura i u nastavku rada ćemo joj posvetiti najviše pozornosti.

U primjeru na slici 3.1, vrijedile bi sljedeće jednakosti: C[n] = 2 (od n su manji a i #) i Occ(n, 3) = 1 (n se pojavljuje samo jednom u prva 3 znaka niza *L*).

Burrows i Wheeler dokazali su sljedeća svojstva koja nam omogućuju izvesti algoritam reverzne BWT:

1. U svakom retku matrice M_T, zadnji znak prethodi prvom znaku u istom retku matrice u originalnom tekstu *T*. To je direktno posljedica definicije retka matrice M_T koji se dobiva cikličkim pomakom prethodnog retka u desno.

2. Neka je *LF*[*i*] = C[*L*[*i*]] + Occ(*L*[*i*], *i*). *LF*(*i*) (engl. *Last-to-First column mapping*) vraća indeks retka matrice M_T u kojem se posljednji znak *i*-tog retka pojavljuje u prvom stupcu. Prvi pribrojnik C[*L*[*i*]] označava broj retka neposredno prije prvog primjerka znaka koji tražimo (tj. u M_T "preskočimo" sve retke koji započinju znakom manje leksikografske vrijednosti). Preostaje još preskočiti znakove iste vrijednosti, ali koji se nalaze u retcima do *i*-tog što se postiže pribrajanjem Occ(*L*[*i*], *i*). To je ispravno zato jer retci koji su bili ispred traženog dok im je zadnji znak bio jednak *L*[*i*], bit će ispred njega i kad se svima ciklički udesno pomakne jedan znak.

3. Ako je *T*[*k*] *i*-ti znak od *L*, tada je *T*[*k* - 1] = *L*[*LF*[*i*]]. Ova tvrdnja slijedi direktno iz 1. i 2. tvrdnje, i nju ćemo koristiti kao osnovni korak pri reverznoj transformaciji.

Rekonstrukciju originalnog teksta *T* iz transformacije *L* moguće je ostvariti na sljedeći način: znamo da je posljednji znak *T* upravo # te primjenom tvrdnje 3. možemo lako dobiti pretposljednji znak. Tada opet ovaj korak opet ponovimo, ali ovaj puta za pretposljednji znak koji smo upravo otkrili te dobivamo *T*[*n* - 2]. Ponavljanjem postupka na opisani način dolazimo do *T* počevši od *L*.

3.2. Sažimanje temeljeno na Burrows-Wheeler transformaciji

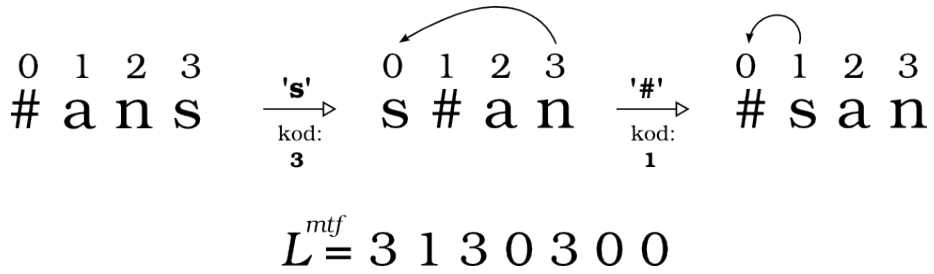
BWT sam po sebi ne vrši sažimanje nad tekstom koji transformira, već kao izlaz daje samo permutaciju ulaznih znakova, čime se zauzeće memorije ne mijenja. Pokazali smo potencijalnu korisnost ovakve transformacije, a u nastavku ćemo objasniti kako je

to svojstvo moguće iskoristiti. Glavna ideja je na dobiveni niz L primijeniti više koraka koji na kraju rezultiraju binarnim kodom. Tako dobiveni kod će biti sažet u odnosu na L i sprema se u memoriju, dok se rezultati svih ostalih međukoraka brišu.

3.2.1. Move-To-Front kodiranje

Move-To-Front (dalje u tekstu MTF) kodiranje je postupak pridjeljivanja dekadskog broja svakom znaku u T. Kao ni BWT, ne vrši sažimanje već služi samo kao pripremni korak za postupke koji slijede. Odvija se na sljedeći način:

- Stvori listu koja sadrži znakove abecede Σ proširene s *#*. Inicijalno su znakovi u listi u leksikografski rastućem poretku. Početno stanje liste ne mora biti zadano nužno ovim pravilom, ali mora biti unaprijed definirano kako bismo znali napraviti dekodiranje. MTF kodna vrijednost znaka definira se kao njegova trenutna pozicija u listi, počevši od nule.
- Uzimamo znak po znak iz L i pridijeljujemo mu vrijednost koja odgovara njegovoj trenutnoj poziciji u listi. Pročitani znak stavljamo na prvo mjesto u listi gdje poprima vrijednost 0. Pri tome redoslijed ostalih znakova u listi ne mijenjamo. Na kraju dobivamo kod L^{mtf} izgrađen nad abecedom {0, 1, 2, ..., |Σ|}.



Slika 3.2: Move-To-Front kodiranje prva dva znaka za L = `s#nnaaa` i krajnji rezultat.

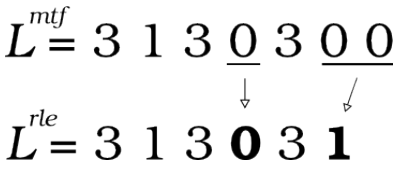
Ovakav algoritam će za niz uzastopnih znakova generirati niz nula. Možemo reći da u svakom trenutku (osim na početku) MTF lista sadrži znakove poredane po neposrednosti njihovog pojavljivanja. Niži brojevi predstavljaju učestalo pojavljivanje jednakih znakova. Zbog opisanih svojstava BWT transformacije očekivano je da će u generiranom MTF kodu prevladavati manji brojevi. Ovakvo kodiranje zahtjeva prolazak kroz sve znakove od T i neprestano osvježavanje MTF liste. Za implementaciju MTF liste uobičajeno je koristiti vezanu listu koja učinkovito može mijenjati poredak svojih elemenata, unatoč linearnoj složenosti dohvaćanja elementa.

3.2.2. Run-length kodiranje

Run-length kodiranje (engl. *Run-length encoding*) (dalje u tekstu i kao RLE) jednostavan je oblik sažimanja u kojem se niz jednakih znakova zamjenjuje drugačijim, tipično kraćim zapisom. RLE ovdje koristimo za kodiranje nizova nula koje generira MTF kodiranje na *L*. Niz nula duljine *m* kodira se na sljedeći način:

- Niz nula 0^m zamijeni se binarnim zapisom broja (*m* + 1) takvim da je najmanje značajan bit na prvom mjestu dok se najznačajniji bit zanemaruje (pošto je uvijek 1). Ovu transformaciju označavat ćemo nadalje s `bin`(*m*). Opisana zamjena opravdana je i prikladna zbog jednostavne reverzne funkcije, a također je poznata pod nazivom 1/2 kod. Ako je `bin`(*m*) = $b_0 b_1 ... b_k$, tada je *m* = $\Sigma_{j=0}^{k}(b_j+1)2^j$. Drugim riječima, znajući vrijednost `bin`(*m*) lako možemo doći do početnog niza 0^m tako da svaki bit b_j zamijenimo s nizom od $(b_j+1)2^j$ nula. Ovo svojstvo koristit ćemo u dekodiranju.
- Pošto ovako dobiven kod još uvijek nije krajnji binarni kod, a kako već u ovoj fazi koristimo binarne znamenke (koje se neće naći u finalnom kodu), kako ne bi došlo do zabune njih označavamo uvodeći nova dva znaka: **0** i **1**. Rezultirajući kod ove faze jest izgrađen nad abecedom {**0**, **1**, 1, 2, ..., |Σ|}, a označavamo ga s L^{rle}. MTF kod 0 se više ne pojavljuje pošto su svi nizovi njegova pojavljivanja zamijenjeni na opisan način.

Primjerice, niz 0^{100} bit će zamijenjen nizom **001001**. Tako je niz od 100 uzastopnih jednakih znakova zapisan pomoću samo 6 znakova. RLE je iako jednostavna, prilično učinkovita i lako reverzibilna metoda sažimanja.



Slika 3.3: Run-length kodiranje primjera sa slike 3.2.

3.2.3. Prefiksni kod promjenjive duljine

Preostalo nam je još samo sažeti i zakodirani L^{rle} (*T*) zapisati u binarnom obliku. Za zapis u binarnom obliku odlučili smo se prvenstveno zbog želje za zauzećem čim manjeg memorijskog prostora, čak i po cijenu kompliciranijeg postupka dekodiranja koji dodatno otežavamo za još jedan "sloj". Prefiksni kod promjenjive duljine

(engl. *Variable-length prefix code*) (dalje u tekstu i kao VLPC) karakterizira što nijedna kodna riječ nije prefiks druge kodne riječi, a duljina kodne riječi nije fiksna. Prvo svojstvo omogućava jednoznačno dekodiranje tako zakodiranog teksta, a drugo svojstvo osigurava učinkovitiji kod pošto svaka kodna riječ poprima optimalnu veličinu. Nepovoljna posljedica promjenjive veličine kodne riječi je nemogućnost jednostavnog određivanja granica kodnih riječi u kodu, već je nužno provesti dekodiranje od samog početka koda. VLPC kod koji koristimo definiramo na sljedeći način:

- Znakovi **0** i **1** kodiraju se s po 2 bita – 10 za **0**, 11 za **1**.
- Znaku *i* iz {1, 2, ..., |Σ|} pridijeli se binarna reprezentacija broja (*i* + 1) kojoj prethodi niz nula duljine za jedan manje od duljine binarnog zapisa od (*i* + 1).

Rezultat ovog koraka je finalni kod u binarnom obliku koji se sprema u memoriju označen s Z. Možemo primijetiti da će kodna riječ koja označava niz nula uvijek započinjati s bitom 1, dok će kodna riječ koja predstavlja MTF kodnu riječ različitu od nule započinjati s bitom 0. Ovo svojstvo, uz stvaranje rječnika za sve kodne riječi koje ne predstavljaju niz nula temelj je algoritma za dekodiranje ovako dobivenog koda.

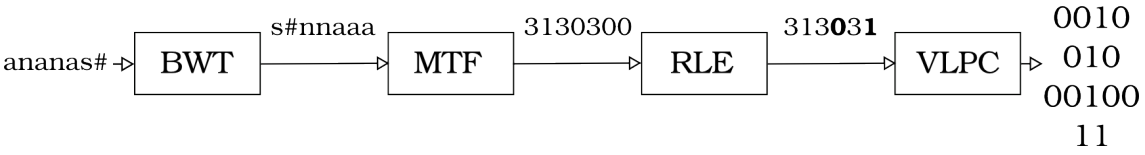
$$Z = \overbrace{00100}^{3}\ \overbrace{010}^{1}\ \overbrace{00100}^{3}\ \overbrace{10}^{0}\ \overbrace{00100}^{3}\ \overbrace{11}^{00}$$

Slika 3.4: Prefiksni kod promjenjive duljine za L^{rle} sa slike 3.3.

Ferragina i Manzini u svom radu [1] dokazali su gornju granicu memorijskog zauzeća ovakvog načina kodiranja:

$$|Z| \leq 5nH_k(T) + O(logn), \forall k \geq 0$$

, gdje je $H_k(T)$ entropija *k*-tog reda od *T*.



Slika 3.5: Dijagram toka kodiranja za *T* = `ananas#` .

3.3. Strukture FM indeksa

Traženje nekog uzorka možemo svesti na dvije faze: prebrojavanje pojavljivanja zadanog uzorka te lociranje njegovih pojavljivanje u *T*-u. FM indeks izrađen je upravo na taj način, faza lociranja koristi fazu prebrojavanja. Također, ovakva podjela na dvije odvojene faze omogućava nam lakše razumijevanja cijeloga procesa i samim time lakšu implementaciju algoritma. U ovom radu opisani su i implementirani samo algoritam za prebrojavanje pojavljivanja uzorka i pripadajuće mu strukture.

3.3.1. Algoritam unazadne pretrage

Želimo moći odgovoriti na pitanja "Pojavljuje li se uzorak *P* u *T*?" i ako je odgovor potvrdan, "Koliko se puta *P* pojavljuje u *T*?" Primijetimo da odgovor na drugo pitanje u sebi sadrži i odgovor na prvo pitanje – moramo samo utvrditi je li broj pojavljivanja veći od nule. Zaključujemo da ostvarenjem algoritma za brojanje pojavljivanje uzorka možemo također vršiti provjeru postoji li traženi uzorak u tekstu.

Algoritam unazadne pretrage (engl. *backward search algorithm*) temelji se na konceptualnoj matrici M_t i strukturama C te Occ. Korisna su nam sljedeća svojstva matrice M_t:

- Svi sufiksi od *T* prefiksirani uzorkom *P* zauzimaju neprekinuti niz redaka u M_t. To je posljedica činjenice da matrica M_t sadrži retke leksikografski sortiranima pa će zato retci prefiksirani istim nizom (u ovom slučaju *P*) nalaziti se jedan do drugoga u M_t.
- Početak takvog niza nazovimo *Prvi*, a kraj *Zadnji*. Zbog prethodnog svojstva broj pojavljivanja uzorka *P* u tekstu *T* bit će $(Zadnji - Prvi + 1)$.

Zaključak je da pošto su svi retci M_t koji su prefiksirani uzorkom *P* u neprekinutom nizu, dovoljno je naći poziciju leksikografski najmanjeg i najvećeg takvog retka iz čega direktno slijedi broj pojavljivanja *P* u *T*. Primjerice, za *T* = `ananas` i *P* = `an` vrijedi *Prvi* = 2 (retke M_t indeksiramo od 1) i *Zadnji* = 3, stoga broj pojavljivanja iznosi $3 - 2 + 1 = 2$ (vidljivo na slici 3.1).

Algoritam 1 izvodi se u *p* koraka, što odgovara duljini traženog uzorka *P*. Nakon postavljanja početnih uvjeta, svaka faza induktivnim korakom izvodi se iz prethodne te vrijedi: na kraju *i*-te faze *Prvi* pokazuje na prvi redak M_t prefiksiran s $P[i, p]$, a *Zadnji* pokazuje pokazuje na posljednji redak M_t prefiksiran s $P[i, p]$. Važno je primijetiti da se rješenje induktivno gradi na sufiksima zadanog prefiksa *P* s pomakom od jednog znaka. Ime algoritma dolazi od činjenice da počinje od posljednjeg znaka te se kreće

Algorithm 1 `unazadna_pretraga`

1: **Ulaz:** $P[1, p]$ - uzorak koji tražimo duljine p.
2: **Izlaz:** Prvi i Zadnji ako postoje.
3: **Korištene strukture:** C[] je niz koji za svaki znak iz Σ pamti koliko je ukupno u tekstu manjih znakova. Occ(c, q) vraća koliko puta se znak c pojavljuje u prvih q znakova L.
4: $i := p$, $c := P[p]$, Prvi := $C[c] + 1$, Zadnji := $C[c + 1]$
5: **while** Prvi $\leq$ Zadnji **and** $i \geq 2$ **do**
6: $c := P[i - 1]$
7: Prvi := $C[c]$ + 1 + Occ(c, Prvi -1)
8: Zadnji := $C[c]$ + Occ(c, Zadnji)
9: $i := i - 1$
10: **end while**
11: **if** Zadnji $<$ Prvi **then**
12: **return** "Ne postoji nijedan redak prefiksiran s P!
13: **else**
14: **return** (Prvi, Zadnji)
15: **end if**

unazad po *P*. Kako bismo dobili rješenje za *P*, prisiljeni smo izračunati i rješenja za sve sufikse od *P*. Tu činjenicu treba u daljnjim fazama iskoristiti i pokušati na neki način iskoristiti ili spremiti dobivena međurješenja kako bi se smanjio broj ponovljenih upita za isti uzorak.

Objasnimo algoritam detaljnije:

- U retku (3) računa se rješenje za prefiks od samo jednog znaka – posljednjeg znaka *P*. *Prvi* "preskače" sve znakove manje od njega te pokazuje na prvi redak koji počinje njime. *Drugi* "preskače" sve znakove manje ili jednake od traženog i pokazuje na posljednji redak koji njime počinje.
- Ako $p = |P|$ iznosi 1, dosad izračunato rješenje odgovara konačnome i algoritam završava. U suprotnom, računa se rješenje za sufiks od *P* duljine 2.
    - U retku (6) $C[c] + 1$ namješta *Prvi* na prvi redak M_t prefiksiran znakom *c* (predzadnji znak u *P*). Vrijednost Occ(c, Prvi -1) govori nam koliko postoji redaka koji počinju znakom *c*, a drugi znak im je leksikografski manji od drugog znaka prefiksa koji u ovom koraku rješavamo. Ti retci se nalaze u M_t prije prvog retka koji tražimo i njih treba "preskočiti".

- U retku (7) postupak zaključivanja jednak je kao u prethodnom retku, jedino što *Zadnji* "preskače" preko redaka koji su prefiksirani traženim prefiksom i namješta se na posljednji takav redak.

– Postupak se ponavlja sve dok nisu obrađeni svi znakovi od *P* te se vraćaju posljednje izračunate vrijednosti $(Prvi, Zadnji)$. U [1] dokazano je ako za rezultat nekog koraka vrijedi $(Zadnji < Prvi)$ znači da se u *T* ne pojavljuje *P* i algoritam se prekida.

Vrijeme izvođenja procedure `unazadna_pretraga` dominantno je određeno vremenom evaluiranja upita Occ(c, q), stoga možemo reći da je vremenska složenost $O(p * O(\text{Occ}(c, q)))$.

3.3.2. Učinkovito računanje upita Occ(c, q)

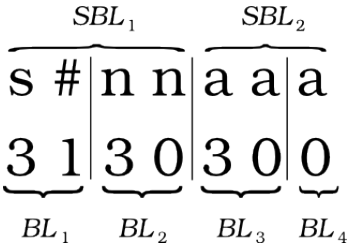
Pokazali smo da brzina evaluacije upita Occ(c, q) određuje brzinu izvođenja procedure `unazadna_pretraga`, a samim time i cjelokupni postupak prebrojavanja pojavljivanja uzorka. Također, faza lociranja uzorka će biti izgrađena na fazi prebrojavanja stoga je vrlo bitno za ukupnu složenost algoritma ovaj upit učiniti čim optimalnijim. Drugim riječima, želimo moći čim brže odgovoriti na pitanje: "Koliko se puta znak c pojavljuje u prvih q znakova *L*?".

Ferragini i Manzina ovaj problem su riješili korištenjem sažetog teksta *Z* i još nekoliko pomoćnih struktura ukupne veličine $o(n)$. Osnovna ideja je podijeliti početni upit na više manjih podupita za koje smo unaprijed izračunali rješenja pri izgradnji indeksa. Postupak je definiran na sljedeći način:

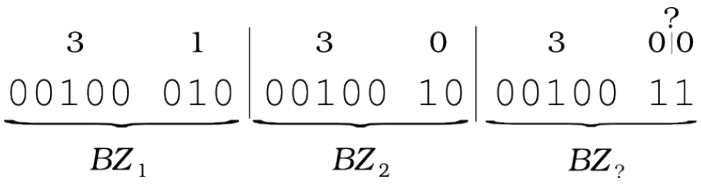
– *L* se logički podijeli u podnizove duljine l. Svaki takav podniz zovemo *odjeljak* (engl. *bucket*) - zamišljamo da znakove iz *L* dijelimo u odjeljke jednakog kapaciteta i označimo ih indeksima 1, ..., n/l. i-ti *odjeljak* označavamo $BL_i = L[(i-1)l + 1, il]$.

– Na isti način *L* se podijeli u podnizove duljine l^2 pod nazivom *nadodjeljak* (engl. *superbucket*). Tako dobiveni podnizovi označavaju se indeksima 1, ..., n/l^2. i-ti *nadodjeljak* označavamo SBL_i.

Podijelimo li *L* na opisan način, podijelili smo i L^{mtf} pošto se MTF kodiranje vrši bez sažimanja i jedna kodna riječ odgovara jednom znaku u *L*. Do problema dolazi ukoliko je niz nula u L^{mtf} sadržan u više od jednog *odjeljka*. Nakon što se primijeni RLE izvršeno je sažimanje i više ne vrijedi činjenica da jedna kodna riječ odgovara jednom znaku u L^{mtf}.



Slika 3.6: Podjela na *odjeljke* i *nadodjeljke* za T = `ananas` i $l = 2$.



Slika 3.7: Situacija u kojoj je niz nula sadržan u više *odjeljaka*.

Opisana situacija prikazana je na slici 3.7. Niz od dvije nule zakodiran je samo jednom kodnom riječi, a svaka nula mora pripasti drugom *odjeljku*. Problem se rješava tako da se lijevom *odjeljku* pridaju kodne riječi koje kodiraju prelijevajući niz nula (kodne riječi koje se koriste za kodiranje niza nula su `11` i `10`) sve dok je broj nula koje sadrži manji od broja nula koje su doista u njemu. U slučaju sa slike 3.7 BZ_3 će glasiti `00100 11` i time sadržavati jednu nulu previše, dok će BZ_4 jedna nula nedostajati. Zato je nužno uvesti dodatnu strukturu koja za svaki *odjeljak* pamti koliko mu nula na početku nedostaje. Nazvat ćemo je MZ (engl. *missing zeroes*), a $MZ[i]$ sadrži broj nula koje i-tom odjeljku nedostaju na njegovom početku u Z. Također, činjenica da jedan odjeljak u svojem binarnom zapisu sadrži previše kodnih riječi ne utječe na točnost algoritma. Poznata nam je točna veličina svakog odjeljka stoga će dekodiranje završiti prije nego što dođe do te kodne riječi.

Kako bismo izračunali Occ(c, q), logički dijelimo $L[1, q]$ na sljedeće podnizove:

- Najduži prefiks od $L[1, q]$ čija duljina je višekratnik od l^2;
- Najduži prefiks preostalog sufiksa čija je duljina višekratnik od l;
- Preostali sufiks, koji je ujedno i prefiks *odjeljka* koji sadrži q-ti znak.

Ono što smo zapravo ovom podjelom postigli je da smo L pridijelili pripadajućem odjeljku i nadodjeljku te nam je eventualno ostao dio odjeljka u kojem se nalazi posljednji dio niza s q-tim znakom. Ideja je unaprijed izračunati za svaki znak koliko se puta pojavljuje u svakom odjeljku i nadodjeljku i po upitu samo očitati te vrijednosti. Traženi rezultat dobivamo zbrajanjem broja pojavljivanja znaka c u svakom od tri navedena podniza. U tu svrhu, uvodimo sljedeće strukture:

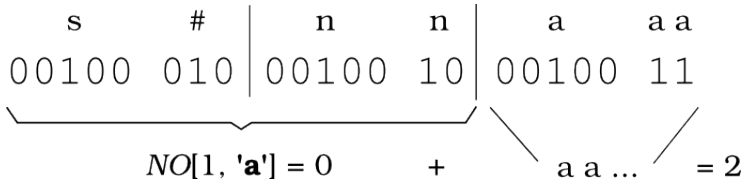
– Za podniz duljine l^2:

- Za $i = 1, ..., n/l^2$ $NO[i, c]$ sadrži broj pojavljivanja znaka c u svim nadodjeljcima do i-tog (uključivo).
- Za $i = 1, ..., n/l^2$ $W[i]$ sadrži zbroj veličina binarnih zapisa odjeljaka u u svim nadodjeljcima do i-tog (uključivo).

– Za podniz duljine l:

- Za $i = 1, ..., n/l$ $NO'[i, c]$ sadrži broj pojavljivanja znaka c u svim odjeljcima do i-tog, počevši od nadodjeljka kojemu i-ti odjeljak pripada.
- Za $i = 1, ..., n/l$ $W'[i]$ sadrži zbroj veličina binarnih zapisa svih odjeljaka do i-tog, počevši od nadodjeljka kojemu i-ti odjeljak pripada.

Nakon podjele na navedena tri podniza, pojavljivanje traženog znaka u prva dva (podnizovi duljine l i l^2) možemo jednostavno očitati iz struktura NO i NO'. Preostaje na još samo odrediti broj pojavljivanja znaka u trećem podnizu, prefiksu jednog od odjeljaka. Taj posljednji rezultat nemamo nigdje unaprijed izračunat, već moramo odjeljak dekodirati iz njegovog binarnog zapisa u početni oblik i prebrojati koliko se puta traženi znak u njemu pojavljuje. Također, ne moramo pretražiti cijeli odjeljak već samo do sveukupno q-tog znaka što znači da je dovoljno iz odjeljka dekodirati samo onaj broj znakova koji nas zanima, a ne odjeljak u potpunosti. Pošto za kodiranje koristimo prefiksni kod promjenjive duljine, strukture W i W' nužne su nam kako bismo znali koji bit u Z odgovara početku odjeljka koji sadrži q-ti znak.



Slika 3.8: Računanje Occ('a', 6) za *T* = `ananas` i $l = 2$.

Primjer na slici 3.8 vizualizira opisani način računanja Occ(c, q). Znak koji nas zanima je drugi po redu u trećem odjeljku. Pošto se nalazi neposredno na početku drugog nadodjeljka, prije početka dekodiranja dovoljno je pogledati zapis u NO za prvi nadodjeljak. Ostaje još dekodirati treći odjeljak i izbrojati pojavljivanje znaka 'a' u njemu. Kod koji koristimo za zapisivanje u binaran oblik je prefiksan, stoga ga je moguće dekodirati u samo jednom prolasku. Po njegovom dekodiranju dobivamo MTF kod. Kako bismo njega mogli dekodirati u znakove od *L*, nužno je znati izgled MTF liste

neposredno prije početka kodiranja ciljanog odjeljka. Zato uvodimo dodatnu strukturu, niz MTFState. Za $i = 1, ..., n/l$ MTFState[i] sadrži stanje MTF liste neposredno prije početka kodiranja i-tog odjeljka.

Koristeći ovakav pristup, Occ(c, q) moguće je izračunati u složenosti $O(l)$. Vrijednost za l se tipično odabire kao $log_2(n)$ pa složenost iznosi $O(log(n))$. Ideja je čuvati cijeli ulazni tekst u sažetom obliku i po svakom upitu dekodirati samo mali, ciljani dio teksta koji odgovora sufiksu jednog od odjeljaka. Zaključujemo da brzina Occ upita dominantno ovisi o brzini dekodiranja dijela teksta, i zato je bitno taj postupak čim učinkovitije implementirati.

3.3.2.1. Struktura Occ

Sukladno opisu u poglavlju 3.3.2 definiramo strukturu Occ koja omogućuje upit $Occ(c, q)$:

Članovi

- $NO[1, ..., n/l^2, |\Sigma|]$ - za svaki nadodjeljak i znak broj pojavljivanja
- $W[1, ..., n/l^2]$ - za svaki nadodjeljak informacija o veličini
- $NO'[1, ..., n/l, |\Sigma|]$ - za svaki odjeljak i znak broj pojavljivanja
- $W'[1, ..., n/l]$ - za svaki odjeljak informacija o veličini
- $C[|\Sigma|]$ - za svaki znak broj manjih znakova u T
- $MTFState[1, ..., n/l]$ - za svaki odjeljak stanje MTF liste prije kodiranja
- $MZ[1, ..., n/l]$ - za svaki odjeljak broj nula koje nedostaju
- Z - binarni kod teksta T
- Σ - abeceda nad kojom je izgrađen T

Posljedična složenost upita $Occ(c, q)$

Ferragina i Manzini za vrijednost parametra l (veličine odjeljka) predložili su $\alpha * \log n$, gdje je α proizvoljan parametar. Zbrojem memorijskog zauzeća svakog pojedinog člana dolazi se do sljedeće memorijske složenosti:

$$O(n * |\Sigma| * (\frac{1}{\log n} + \frac{1}{(\log n)^2}))$$

Naravno, memorijsko zauzeće u implementaciji dosta ovisi o konstantama nevidljivima u ovoj formuli, poput α i zauzeća prikaza cijelog broja u memoriji računala.

Vidljiva je i ovisnost o veličini abecede Σ koja se pokazala dosta važnom u praksi, stoga je istaknuta u formuli složenosti iako je konstantna vrijednost. Postoje određene implementacije FM - indeksa [4] koje u puno manjoj mjeri ovise o veličini abecede.

Vremenska složenosti ovisi isključivo o veličini odjeljka koji je potrebno dekodirati stoga iznosi $O(\log n)$.

4. Implementacija

Glavni cilj ovoga rada bio je implementirati funkcionalni FM indeks koji podržava upite o broju pojavljivanja zadanog uzorka. Implementacija se može koristiti u svrhe istraživanje u različitim područjima i ostvareno je sučelje koje omogućuje laku nadogradnju novih funkcionalnosti poput lociranja uzoraka. Zbog jednostavnosti, interakcija programa i korisnika odvija se putem komandne linije.

Implementacija je napravljena na operacijskom sustavu Linux. Pošto je osnovna ideja FM indeksa postići veću brzinu u odnosu na postojeća rješenja, kao jezik za implementaciju odabran je C++. C++ dijeli brzinu s C-om, ali je obogaćen skupom dodatnih funkcija i struktura koje znatno olakšavaju razvoj programskih rješenja. Također podržava objektno orijentiranu paradigmu koja čini strukturni temelj ove implementacije. U ovom poglavlju skicirat ćemo izgled programskog rješenja, detaljnije objasniti dijelove čije izvedba nije očita iz same metode te ukazati na prostor za moguća poboljšanja i nadogradnju.

4.1. Razredi i metode

Izvorni kod programa podijeljen je na razrede i njihova zaglavlja koja definiraju javne i privatne metode. Svaki razred ostvaruje skup odgovornosti za određeno područje i implementira smislenu cjelinu algoritma. Pošto je rad usmjeren na implementaciju algoritama i brzinu izvođenja, nije se ukazala potreba za polimorfizmom niti složenijim oblikovnim obrascima. Implementirani razredi su sljedeći:

- **Alphabet** - sadrži skup znakova abecede i kroz sučelje omogućuje drugim razredima pristup u prikladnom obliku. Također podržava dekodiranje pojedinog znaka abecede iz binarnog zapisa u MTF kod.
- **BitArray** - predstavlja niz bitova kao rezultat binarnog zapisa dobivenog primjenom prefiksnog koda te kroz sučelje mogućuje klijentu pristup pojedinom bitu. Odvaja logičku izvedbu od tehničke i omogućuje promjenu interne struk-

ture bez promjene ostatka programa u slučaju da se ukaže potreba za učinkovitijom implementacijom.

- **Compressor** - najsloženiji razred u cijeloj implementaciji, sadrži algoritme za kodiranje i dekodiranje ulaznog teksta te stvaranje struktura indeksa. Enkapsulira stvoreni indeks i omogućava klijentu korištenje putem javnog sučelja.
- **Opp** - glavni javni razred koji enkapsulira ostale. Njega korisnik treba uključiti u svoj kod želi li koristiti metode ove strukture FM indexa.
- **OppRows** - predstavlja povratnu vrijednost metode koja implementira proceduru `unazadna_pretraga` opisanu u poglavlju 3.3.1. Sadrži indekse prvog i posljednjeg retka prefiksiranog traženim uzorkom.

4.1.1. Alphabet

Razred **Alphabet** izgrađuje se iz ulaznog niza znakova *T*, a u konstruktor mu se također prosljeđuje i specijalni znak za provođenje Burrows-Wheeler transformacije (poglavlje 3.1). Pri inicijalizaciji prolazi kroz cijeli *T* i gradi skup znakova. Ukoliko se radi na velikim tekstovima bilo bi efikasnije kada bi korisnik sam unaprijed zadao korištenu abecedu u tekstu čime bi se izbjegao jedan prolaz kroz čitavi tekst. Interno se za spremanje skupa znakova koristi standardna C++ implementacija skupa `set`. Važnije metode su:

- `toSortedList()` - vraća skup znakova transformiran u listu, što je optimalna struktura za MTF kodiranje.
- `decodeToMTF()` - prima kodnu riječ u binarnom obliku i vraća odgovarajući MTF kod. Razred ustvari sadrži rječnik koji se koristi pri dekodiranju. Rječnik je ostvaren pomoću strukture `map`.

4.1.2. BitArray

BitArray koristi se za spremanje niza bitova i omogućava pristup pojedinom bitu. U trenutnoj verziji implementacije interno koristimo niz podataka tipa `bool`. Iako je takva struktura memorijski neučinkovita pošto zauzima 7 bitova previše za svaki bit koristili smo je u svrhu lakšeg testiranja programa. Upravo je namjena ovog razreda sakriti internu strukturu i omogućiti laku implementaciju prikladnije strukture u budućnosti.

4.1.3. Compressor

Ovaj razred implementira algoritme za sažimanje, dekodiranje i inicijalizira strukture indeksa. Zanimljive su nam sljedeće metode:

- `getSuffixArray()` - kao što je spomenuto u poglavlju 3.1, brzina BWT u potpunosti ovisi o brzini izgradnje sufiksnog polja. Postoje razni algoritmi koji rješavaju ovaj problem u različitim složenostima te su pogodni za različite vrste tekstova. Mi smo implementirali algoritam koji koristi standardnu C++ implementaciju quicksort algoritma, s tim da smo definirali posebni razred za usporedbu. Time smo omogućili sortiranje korištenjem samo ulaznog niza, što je memorijski vrlo učinkovito. Složenost ovakvog algoritma sortiranja je $O(n^2 * log(n))$, što je posljedica usporedbe dva podniza. Složenost samog quicksort algoritma iznosti $O(n * log(n))$, dok usporedba dva podniza u najgorem slučaju traje $O(n)$ što zajedno daje $O(n^2 * log(n))$. Ipak, pokazalo se da za većinu tekstova usporedba podnizova traje znatno kraće te se vrijeme izvođenja algoritma pokazalo sasvim prihvatljivim.
- `getVarLengthPrefixEncoding()` - uzima MTF kod kao ulaz i vraća niz bitova zakodiran pomoću prefiksnog koda promjenjive duljine. Za razliku od opisa u poglavlju 3.2.2, faza primjene RLE nije eksplicitno izdvojena već se izvodi "u letu" zajedno s primjenom prefiksnog kodiranja. Tu se rješava problem prelijevajućih nizova nula i inicijaliziraju strukture W, W' i MZ.
- `occ()` - ova metoda najvažnija je za cjelokupnu brzinu algoritma. U trenutnoj implementaciji implementirana je na način opisan u poglavlju 3.3.2, a u [5] predložene su sljedeće zanimljive optimizacije s ciljem ubrzanja nauštrb zauzeća memorije.
    - Moguće je uvesti spremanje dijelova *Z* koji su dekodirani. Kada jednom dekodiramo dio kodiranog teksta, možemo ga pospremiti i iskoristiti kada opet dobijemo upit za tim odjeljkom. Dekodiranje je dosta skupa operacija, stoga ova optimizacija izgleda obećavajuće.
    - Moguće je za svaki odjeljak pamtiti za svaki znak abecede pojavljuje li se barem jednom u odjeljku. Time se izbjegava nepotrebno dekodiranje cijelog odjeljka u kojem uopće nema traženog znaka.
    - Smanjenjem veličine odjeljka (parametar l) direktno se ubrzava upit i povećava zauzeće memorije.

5. Rezultati i diskusija

U ovom poglavlju testirat ćemo predstavljenu implementaciju na nekoliko različitih tekstova i prezentirati dobivene rezultate. Ono što nas zanima je vrijeme izgradnje indeksa i zauzeće memorije. Za mjerenje vremena koristili smo ugrađene funkcije iz biblioteke `<ctime>`, dok smo zauzeće memorije mjerili alatom *massif*. Sva testiranja provedena su na računalu s Intel Core2 Quad Q9550@2.83Ghz procesorom i 8GB radne memorije. Korišten je prevoditelj `g++` s zastavicom `-O2`. Tekstovi na kojima vršimo testove preuzeti su sa službenih web-stranica FM indexa [6]:

- *english* - spojeni tekstovi na engleskom jeziku preuzeti iz projekta Gutenberg [7].
- *proteins* - datoteka je dobivena spajanjem proteinskih sekvenci iz baze podataka Swisspros.
- *dna* - sadrži sekvence DNA, preuzete iz projekta Gutenberg [7]. Zapis sadrži čisti DNA kod bez opisa gdje su baze označene znakovima A, G, C, T.

Tablica 5.1: Karakteristike tekstova nad kojima su izvođeni testovi

Tekst(mB)	Veličina(mB)	Veličina abecede
english	25	239
proteins	25	25
dna	25	16

Tablica 5.2: Rezultati za tekst *english*

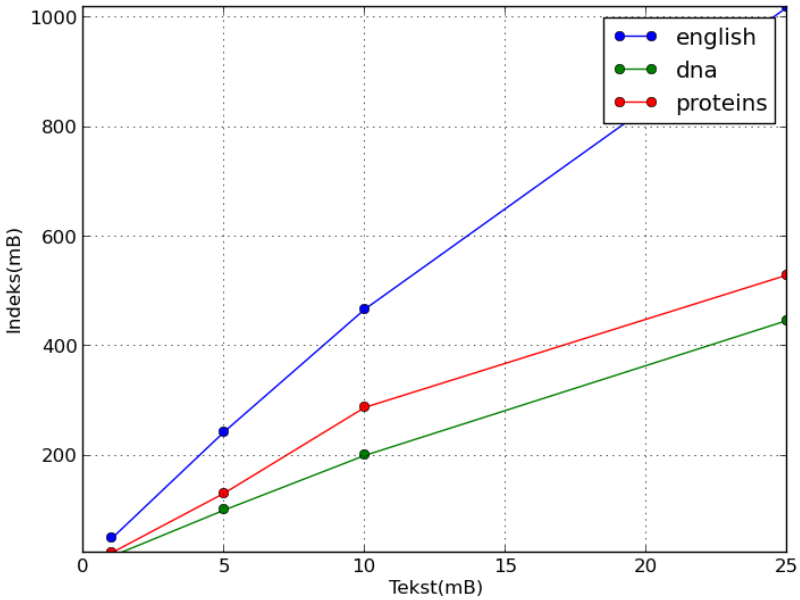
Veličina teksta(mB)	Ukupno vrijeme(s)	Vrijeme izgradnje SA	Maks. mem. (mB)
1	1.85	0.58	49.5
5	11.64	4.87	244.2
10	27.97	14.8	468.1
25	90	56	1019

Tablica 5.3: Rezultati za tekst *dna*

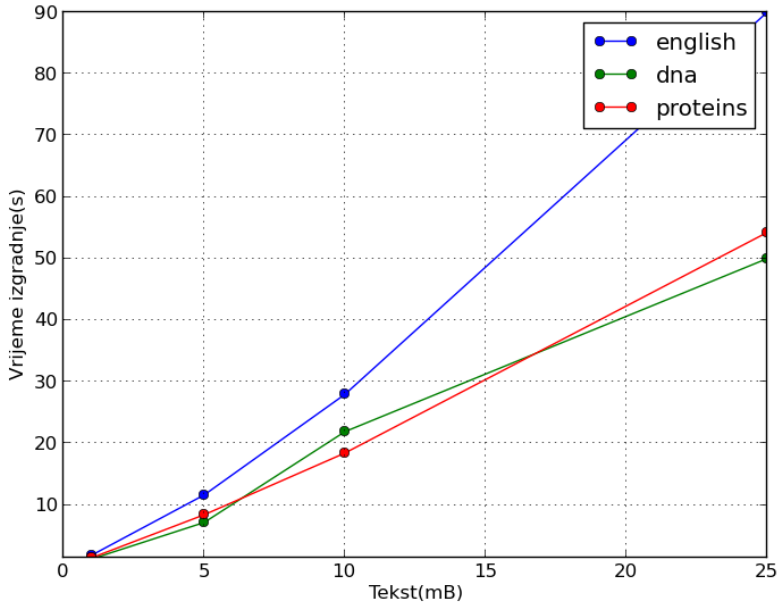
Veličina teksta(mB)	Ukupno vrijeme(s)	Vrijeme izgradnje SA	Maks. mem. (mB)
1	1.23	0.63	16.9
5	7.17	3.9	101.4
10	21.91	12.45	201.1
25	50	30.14	447.3

Tablica 5.4: Rezultati za tekst *proteins*

Veličina teksta(mB)	Ukupno vrijeme(s)	Vrijeme izgradnje SA	Maks. mem. (mB)
1	1.43	0.66	23.4
5	8.42	4.18	132.2
10	18.44	10.56	288.8
25	54.25	34	530.3



Slika 5.1: Ovisnost veličine indeksa o veličini ulaznog teksta.



Slika 5.2: Ovisnost vremena izgradnje indeksa o veličini ulaznog teksta.

Iz rezultata testiranja vidljivo je da zauzeće memorije indeksa i vrijeme njegove izgradnje doista teže linearnoj ovisnosti i veličini teksta kako je i pretpostavljeno. Prosječno faktor koji se pojavljuje uz n za vrijeme izgradnje iznosi 2, a za zauzeće memorije 20. Također je na slikama 5.2 i 5.1 vidljiv utjecaj konstante $|\Sigma|$. Tekst *english*

ima bitno veću abecedu ostalih tekstova stoga mu je potrebno više memorije i duže se izgrađuje. Sukladno tome, tekst *dna* u gotovo svim točkama postiže minimalne vrijednosti od tri teksta.

Možemo primijetiti da u pravilu polovicu vremena izgradnje uzima izgradnja sufiksnog polja. Pošto je za tu svrhu korišten jednostavan algoritam temeljan na uspoređivanju sufiksa i stoga složenosti $O(n^2 * logn)$, smatramo da tu ima mjesta za napredak jer su poznati algoritmi koji isti problem rješavaju u $O(n * logn)$ složenosti.

Zauzeće memorije dominantno je određeno veličinama struktura *NO'* i *W'*, koje pamte velik broj cijelih brojeva. Testovi su provođeni na 64-bitnoj arhitekturi stoga veličina podatkovnog tipa `integer` iznosi 8 bajtova. Algoritam prikazuje cijele brojeve od 1 do *n* (duljina teksta *T*) i u većini slučajeva nije potrebno toliko prostora. Jednostavna mjera smanjenja zauzeća memorije, ukoliko se pokaže potrebnim, bila bi eksplicitno odrediti veličinu cjelobrojnog podatka na 4 bajta, primjerice podatkovnim tipom `uint32_t` iz `<cstdint>` biblioteke.

Posljednji stupac tablica 5.2, 5.3 i 5.4 prikazuje maksimalno zauzeće memorije tijekom izgradnje indeksa. Veličina indeksa nakon izgradnje je manja pošto se iz memorije otpuštaju pomoćne strukture korištene tijekom izgradnje poput sufiksnog polja i MTF koda od *T*.

6. Zaključak

U radu je opisan algoritam za izgradnju FM indeksa i algoritam za podršku operaciji prebrojavanja predstavljen od Ferragine i Manzinia. Definirana je Burrows-Wheeler transformacija te su navedena njena zanimljiva nam svojstva. U nastavku su uvedeni ostali stupnjevi kodiranja, sve do završnog prefiksnog koda promjenjive duljine koji konačno prevodi ulazni tekst u binarni oblik. Navedena je ideja algoritma i struktura koje su mu potrebne, uz analizu složenosti. Objašnjen je problem prelijevajućih nula i predstavljeno rješenje. Opisani algoritam je implementiran i testiran na skupu različitih tekstova.

Ponuđeno je nekoliko ideja za moguću optimizaciju, poput implementacije vremenski učinkovitijeg algoritma i za izgradnju sufiksnog polja i načina za ubrzanje dekodiranja sažetog teksta pomoću pamćenja već izračunatih rezultata. Obje operacije dominiraju vremenom za izgradnju i pretraživanje indeksa stoga vjerujemo da bi predložena poboljšanja mogla imati znatan utjecaj na ubrzanje algoritma te bi ih bilo zanimljivo testirati u nekoj od budućih inačica ove implementacije.

FM indeks je relativno nova ideja u ovom području. Strukturno je sličan sufiksnim stablima i poljima koja su se do sada koristila ali je napredniji po malom zauzeću memorije koja se približava empirijskoj entropiji indeksiranom teksta, stoga ga se često naziva sažetim indeksom. Moguća primjena je pogotovo aktualna u području bionformatike koje se danas bavi obradom golemih količina podataka stoga je presudno koristiti algoritme koji zadovoljavaju postavljena memorijska ograničenja. Autori Ferragina i Manzini vjeruju da bi uz odgovarajući daljnji razvoj sažeti indeksi poput FM indeksa mogli u budućnosti postati bitan dio u razvoju sofisticirane i efikasne programske podrške za rad s velikim količinama podataka.

LITERATURA

[1] P. Ferragina and G. Manzini, "Indexing compressed text," *Journal of the ACM (JACM)*, vol. 52, no. 4, pp. 552–581, 2005.

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

[3] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," in *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pp. 319–327, Society for Industrial and Applied Mathematics, 1990.

[4] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "An alphabet-friendly fm-index," in *String Processing and Information Retrieval*, pp. 228–228, Springer, 2004.

[5] T. Lakatos, "Fm-index implementation," 2008.

[6] *Pizza&Chili Corpus, Compressed Indexes and their Testbeds, The Text Collection*, 2005.

[7] M. Hart, *Project gutenberg*. 2000.

Implementacija FM indeksa

Sažetak

FM indeks relativno je novi algoritam za istovremeno sažimanje i omogućavanje pretraživanja pojavljivanja određenog uzorka u tekstu. Zauzeće memorije ovisi o ulaznom tekstu, a može se regulirati pomoću više parametara i tako mijenjati odnos zauzeća memorije te brzine izgradnje indeksa i njegova pretraživanja. Provedeni su testovi na nekoliko različitih tekstova. Objašnjena je teoretska ideja izgradnje i pretraživanja FM indeksa. Predstavljena je implementacija u jeziku C++ i detaljnije su objašnjeni dijelovi kompliciraniji za implementaciju. Ponuđeno je nekoliko ideja za daljnje poboljšanje performansi FM indeksa u budućnosti.

Ključne riječi: FM indeks, sažeti indeks, pretraživanje teksta, bioinformatika

FM index implementation

Abstract

FM index is relatively new algorithm for data compression that also supports full-text indexing. It uses roughly the space required for storing the text in compressed form. This thesis explains Ferragina and Manzini's proposed solution and presents our own implementation. Implementation was written in C++ according to Ferragina and Manzini's article with certain adjustments. Implementation was tested on several different texts. Data and space complexity and acquired performance is analzyed. Some possible further improvements on current implementation of FM index are presented and left for future work.

Keywords: FM index, compressed index, text search, bioinformatics