SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Projekt iz kolegija bioinformatika

FM-index count

Autori:
Matea Đonlić
Marko Đurasević
Maja Kontrec

Zagreb, 2013.

Sadržaj

1 Uvod **2**

2 Pomoćni algoritmi **3**
2.1 Burrows-Wheeler transformacija 3
2.1.1 Algoritam transformacije 3
2.1.2 Rekonstrukcija originalnog teksta iz BWT transformacije 4
2.1.3 Konstrukcija BWT-a 4
2.1.4 Multikey quicksort 5
2.2 Wavelet stablo 6
2.2.1 Konstruiranje wavelet stabla 6
2.2.2 Upiti nad wavelet stablom 8
2.3 RRR struktura 8
2.3.1 Konstruiranje RRR strukture 9
2.3.2 Izračunavanje broja jedinica u nizu 9

3 FM-indeks **11**
3.1 Originalna implementacija 11
3.2 Naša implementacija 11
3.2.1 Nedostaci implementacije u programskom jeziku Java 14

4 Literatura **15**

Uvod

Porast količine tekstualnih informacija u zadnja dva desetljeća potaknuo je razvoj struktura podataka koje će tu informaciju memorijski efikasno pohraniti, omogućujući pritom vremensku efikasnost pretrage i dohvata. Jedno od područja koje ima veliku korist od razvoja ovakvih struktura je bioinformatika. Radi same prirode bioinformatičkih problema, odnosno, obrade bioloških podataka (npr. analiza dugačkih sekvenci DNK), vrlo je pogodno nad takvim biološkim podacima izgraditi spomenute strukture kako bi se njihova analiza ubrzala. Jedan od pristupa rješavanja ovog problema je upotreba indeksa pretraživanja po cijelom tekstu (eng. full-text search). Ove strukture omogućuju brzo i potpuno pretraživanje teksta nad kojim su izgrađene. Nakon izgradnje, pronalazak proizvoljnog tekstualnog uzorka (zajedno s njegovim brojem ponavljanja) vrlo je učinkovit. No, ovaj pristup, koji stavlja naglasak na brz pronalazak traženih uzoraka, često ne ispunjava memorijske zahtjeve. Budući da korisnost pretrage raste s količinom teksta, memorijska efikasnost vrlo je bitan faktor pri odabiru načina obrade i pohrane teksta kojega je potrebno analizirati. Novije izvedbe indeksa dobiveni tekst prvo sažmu, te tek nakon toga, nad sažetim tekstom grade indekse. Jedan od takvih indeksa je FM-index čija je implementacija i zadatak ovoga projekta.

Pomoćni algoritmi

U ovom poglavlju upisani su algoritmi koji su potrebni za konstruranje i korištenje FM-indeksa u svrhu prebrojavanja broja ponavljanja podniza unutar zadanog niza nad kojim je konstruiran FM-indeks. Opisani su algoritmi Burrows-Wheeler transformacije, wavelet stabla i RRR strukture.

2.1 Burrows-Wheeler transformacija

Burrows-Wheeler transformacija (skraćeno BWT) [1], služi za kompresiju tekstualnog niza na osnovi pronalaska ponavljajućih uzoraka unutar njega. Ova transformacija je reverzibilna i ne iziskuje pohranu dodatnih podataka za rekonstrukciju originalnog niza. Osim ovoga, prilikom BWT transformacije, ni jedan znak originalnog niza se ne mijenja. Dodatna prednost je činjenica da se za transformaciju (osim dodavanja znaka $) ne proširuje ulazna abeceda. Ukoliko ulazni niz sadrži više jednakih podnizova, velika je vjerojatnost da će transformirani niz sadržavati uzastopne nizove jednakih znakova, što kasnije uvelike pogoduje njegovom komprimiranju.

2.1.1 Algoritam transformacije

Algoritam transformacije provodi se u 4 koraka:

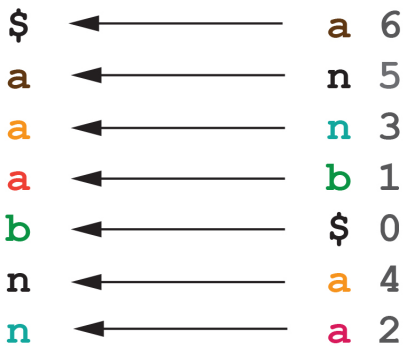
1. Na kraj teksta postavlja se znak $ koji označava kraj te je, leksikografski gledano, najmanji znak ulazne abecede.
2. Stvara se matrica svih različitih cikličkih pomaka niza dobivenog nakon 1. koraka.
3. Matrica iz koraka 2 se sortira u leksikografskom poretku.
4. Dobivena transformacija isčitava se iz poslijednjeg stupca matrice dobivene u koraku 3.

Tijek algoritma za riječ ”banana” prikazan je na slici 2.1. Vidi se kako je BWT transformacija te riječi jednaka annb$aa.

Slika 2.1: Ispunjene vrijednosti u poljima BS i SBS

2.1.2 Rekonstrukcija originalnog teksta iz BWT transformacije

Matrica dobivena u 3. koraku algoritma poslužiti će za rekonstrukciju originalnog teksta iz dane BWT transformacije. Naime, ta matrica ima svojstvo da u svakom njenom retku, zadnji znak prethodi prvom znaku u istom retku matrice u originalnom tekstu. To znači, da je za rekonstrukciju izvornog teksta dovoljan samo prvi i zadnji stupac te matrice. Budući da je poznatno da tu matricu čine sve permutacije ulaznog niza i to sortiranog, jednostavno se može rekonstruirati prvi stupac - jednostavno se uzmu i sortiraju svi znakovi danog transformiranog niza. Budući da znamo da je posljednji znak u nizu znak $, lako je iz prvog i zadnjeg stupca matrice rekonstruirati originalni niz. Na slici 2.2 prikazan je postupak rekonstrukcije originalnog niza (sivi brojevi na desnoj strani označavaju broj koraka).



Slika 2.2: Ispunjene vrijednosti u poljima BS i SBS

2.1.3 Konstrukcija BWT-a

Iako je sam algoritam izgradnje BWT transformacije veoma jednostavan, nažalost nije i efikasan. Za poveće tekstove je jednostavno nemoguće napraviti sve njihove rotacije i onda ih

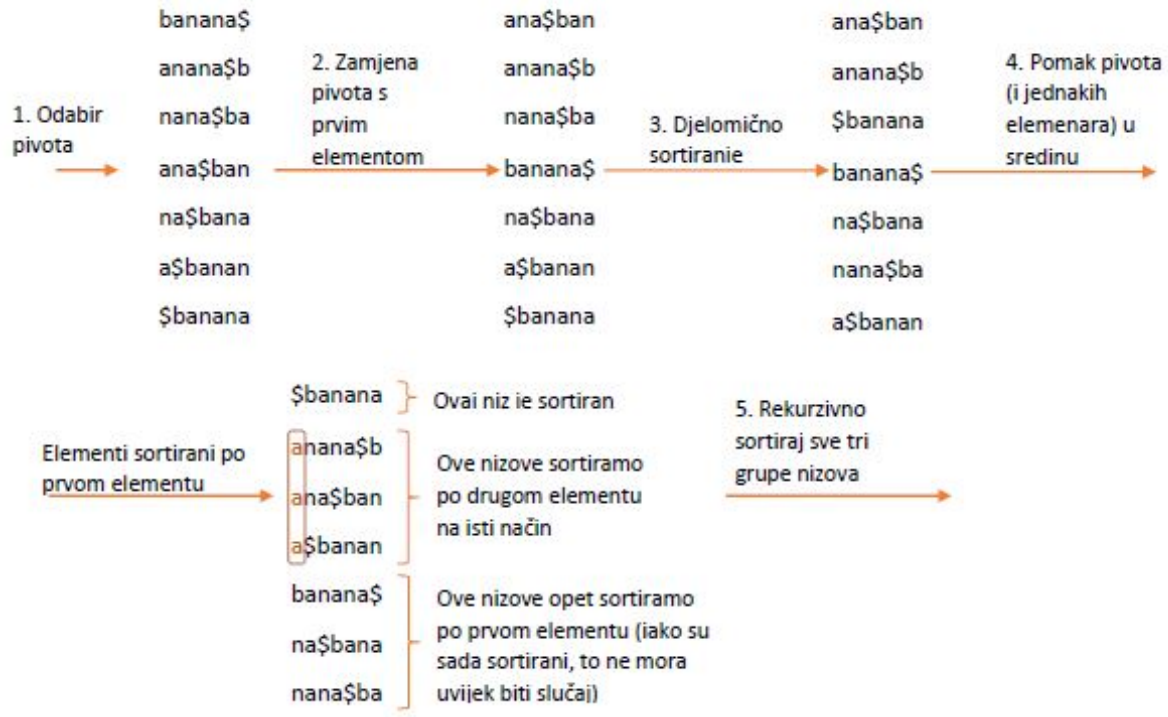
sortirati, jer bi memorijsko zauzeće bilo preveliko. Iz tog razloga potrebno je koristiti algoritme koji BWT transformaciju mogu obaviti memorijski i vremenski efikasno. U nastavku ćemo pogledati dva takva algoritma te vidjeti koje su njihove prednosti i nedostatci.

2.1.4 Multikey quicksort

Prvo ćemo proučiti jedan tip algoritma za sortiranje koji se temelji na quick sort algoritmu predloženom u [referenca]. Prednost ovog algoritma jest u tome da može sortirati sve rotacije jednog niza koristeći samo osnovni niz i dodatno polje pokazivača na taj niz. Na taj način nije potrebno raditi sve rotacije ulaznog niza. Algoritam u svojoj osnovi spaja quick sort i radix sort. Osnovni koraci algoritma prikazani su na slici **??** Prvo se nasumično odabere jedan niz, a prvi element toga niza je pivot. Odabrani niz se nakon toga zamijeni s prvim nizom. Sada djelomično sortiramo sve nizove tako da se nizovi koji počinju s elementom koji je jednak pivotu pozicioniraju na početak ili kraj niza, dok se ostali nizovi poredaju tako da tako da se u prvom dijelu nalaze svi nizovi koji započinju elementom koji je manji od pivota (u primjeru na slici je to niz koji započinje s ”$”), dok se u drugom dijelu nalazenizovii koji započinju s elementom koji je veći od pivotna (u primjeru su to nizovi koji započinju s ”b” i ”n”). Idući korak jest da se svi nizovi koji započinju elementom koji je iznosom jednak pivotu premjeste u sredinu i to tako da se nizovi koji započinju s manjim elementom nalaze ispred njih, a nizovi koji započinju s većim elementom iza njih. Sada su nizovi efektivno podijeljeni u tri dijela, prvi dio koji sadrži nizove koji započinje elementom koji je manji od pivora, drugi dio koji čine nizovi koji počinje elementima koji su jednaki pivotu i treći dio koji sačinjavaju nizovi koji započinju elementom koji je veći od pivota. Sada kada imamo ovu podjelu, svaki od ta tri dijela je potpuno neovisan o drugima i svi se mogu nezavisno sortirati (ovdje je očita i jedna prednost ovog postupka koju ćemo kasnije naglasiti). Sada prvu i treću grupu nizova sortiramo na isti način kao što smo sortirali sve nizove. Drugu grupu ćemo sortirati na isti način, jedina će razlika biti da ćemo sada sortirati po drugom znaku tih nizova (jer je očito da su sortirani po prvom znaku), no ostatak algoritma je u potpunosti jednak. Algoritam se nastavlja dok svi nizovi nisu u potpunosti međusobno sortirani.

Bitno je za napomenuti da je zbog ilustracije rad algoritam prikazan na nezavisnim nizovima (kao da smo prvo napravili sve rotacije osnovnog niza i onda njih sortirali). Naravno u implementaciji to nije tako napravljeno, već postoji niz brojeva koji predstavljaju pokazivače (odnosno indekse pošto pokazivači ne postoje u programskom jeziku Java) na elemente ulaznog polja. Onda umjesto da se zamjenjuju čitavi nizovi jednostavno se zamijene pokazivači u tom polju.

Prednost ovog postupka su prije svega njegova jednostavnost. Sam algoritam je veoma intuitivan i nije ga teško razumjeti.Druga, mnogo očitija prednost jest mogućnost paralelizacije ovog algoritma. Kako je ranije navedeno, nizovi su na kraju podijeljeni u tri nezavisne grupe, od kojih se svaka sortira nezavisno. To znaći da bi teoretski svaku ovu grupu mogli sortirati međusobno paralelno. Nažalost zbog veoma loše implementacije paralelnih mehanizama u programskom jeziku Javi , ovo ipak nije napravljeno. No kada bi implementacija bila napravljena u nekom programskom jeziku koji je više specijaliziran za paralelizaciju to bi bilo dobro napraviti. Najveći nedostataci ovog postupka su svakako nešto veća složenost naspram konstrukcije BWT-a korištenjem sufiksnog polja, ta također rekurzivna priroda algoritma koja



Slika 2.3: Test

zahtijeva puno rekurzivnih poziva i iz tog alokaciju veće količine memorije za stog.

Iako se ovdje radi o veoma zanimljivom algoritmu, nažalost pokazao se kao prespor u usporedbi s izračunavanjem BWT transformacije korištenjem sufiksnog polja.

2.2 Konstrukcija sufiksnog polja

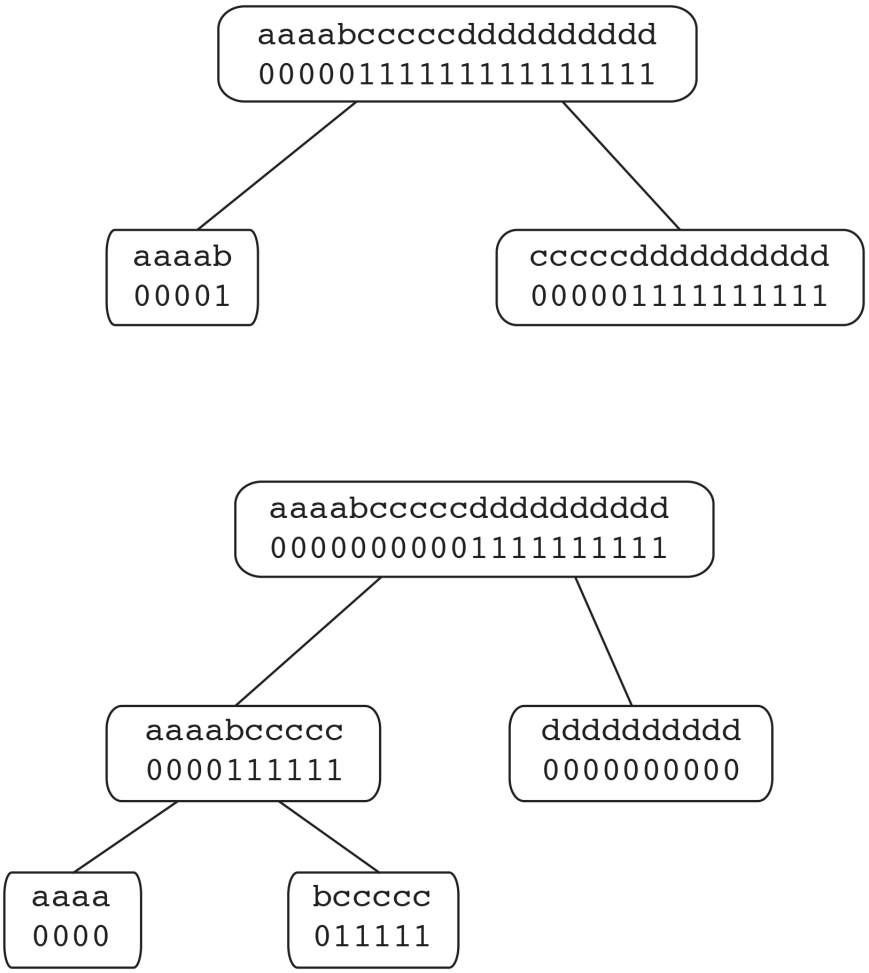
Sufiksno polje jest jednostavna stuktura podataka koje je sastavljena od niza cijelih brojeva koji sadrže početna mjesta abecedno poredanih sufiksa nekog niza.

2.3 Wavelet stablo

Wavelet stablo (engl. *wavelet tree*) je podatkovna struktura koja tekstualni niz organizira u hijerarhiju nizova nula i jedinica - *bit-vektora*. Ova struktura omogućuje pronalazak broja pojavljivanja do nekog indeksa u nizu - *ranga* nekog znaka sa složenošću od $O(logn)$, gdje je n veličina abecede danog tekstualnog niza. Stablo se izgrađuje rekurzivno, razdijeljujući abecedu u parove podabeceda. Nakon izgradnje stabla, svaki list odgovara jednom znaku abecede, dok ostali unutarnji čvorovi označavaju da li pojedini znak teksta pripada prvoj ili drugoj podabecedi. Ukoliko se bit-vektori koji izgrađuju stablo pohrane u RRR strukturu (koja je opisana u idućem poglavlju), moguće je da se memorijsko zauzeće smanji s obzirom na originalno stablo koje ne koristi RRR strukturu.

2.3.1 Konstruiranje wavelet stabla

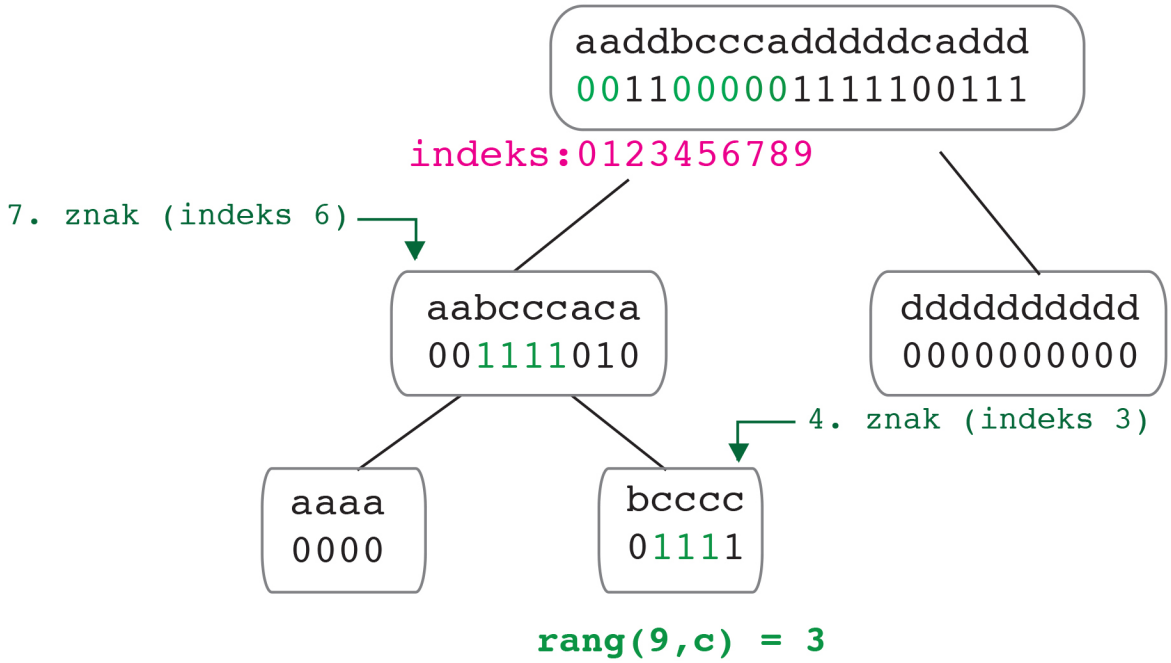
Wavelet stablo pretvara dovedeni niz u balansirano binarno stablo bit-vektora, gdje se dio znakova, koji pripadaju prvoj polovici dane abecede, kodira kao 0, a drugi dio kao 1. Iako se na prvi pogled čini kako ovo unosi nejednoznačnost u dekodiranju sadržaja stabla, to ipak nije tako. Naime, prelaskom u slijedeću razinu, prva polovica abecede iz prethodne razine ponovno se dijeli na dva dijela te se njeni znakovi ponovo kodiraju kao 0 ili 1. Postupak se rekurzivno nastavlja sve dok se abeceda ne sadrži samo dva znaka, koji se tada jednoznačno kodiraju; prvi s 0, a drugi s 1. Originalna podjela ulazne abecede koju su predložili Grossi, Grupta i Vitter [2] abecedu uvijek dijeli na dva jednaka dijela. Podjela abecede koja je implementirana u sklopu ovog projekta razlikuje se od originalne. Naime, kako bi se ujednačila veličina podijeljenih abeceda, a time i memorijsko zauzeće strukture, abecede se dijele na osnovu učestalosti pojavljivanja pojedinog znaka abecede unutar ulaznog niza. U ovoj implementaciji, abeceda se dijeli na prvom znaku čiji broj pojavljivanja u ulaznog nizu, zbrojen s ukupnim brojem pojavljivanja svih znakova koji su u abecedi njegovi prethodnici, veći ili jednak polovici veličine ulaznog niza. Na slici 2.4 prikazana je usporedba izgradnje stabla originalnim (gore) i ovdje implementiranim postupkom (dolje). Abeceda ulaznog niza je a,b,c,d. Broj pojavljivanja znaka a je 4, znaka b 1, znaka c 5, a znaka d 10 puta. Veličina ulaznog niza iznosi 20. U prvom slučaju, ulazna abeceda dijeli se na podabecede a,b i c,d, dok se u drugom slučaju abeceda dijeli na a,b,c i d. Razlog tomu je što je zbroj broja pojavljivanja znakova a, b i c jednak polovici veličine ulaznog niza. Razlog zašto je ovaj način memorijski efikasniji objašnjen je u idućem poglavlju iz razloga što ovisi o RRR strukturi koja pohranjuje bit-vektore stabla.



Slika 2.4: Razlika između originalnog i ovdje implementiranog wavelet stabla

2.3.2 Upiti nad wavelet stablom

Slika 2.5 prikazuje primjer pronalaska ranga unutar wavelet stabla. Točnije, primjer pronalazi rang znaka c na poziciji 9. U prvom koraku, budući da znamo da je slovo c kodirano s 0, brojimo nule do indeksa 9. Nakon što smo pronašli 7 nula, spuštamo se u slijedeću razinu stabla - u lijevu granu, iz razloga što je znak c kodiran s 0. Sada, u ovoj grani, budući da je znak c kodiran s 1, brojimo koliko ima jedinica u prvih 7 znakova. Važno je primjetiti kako u ovom koraku ustvari idemo do indeksa 6, a ne 7. Nakon što su pronađene 4 jedinice, spuštamo se u iduću razinu, u desnu granu, te brojimo jedinice do četvrtog znaka (indeks 3). U ovoj razini, ujedno i posljednjoj razini stabla, do četvrtog znaka prebrojavamo 3 jedinice iz čega zaključujemo da je rang znaka c na poziciji 9 jednak 3.



Slika 2.5: Razlika između originalnog i ovdje implementiranog wavelet stabla

2.4 RRR struktura

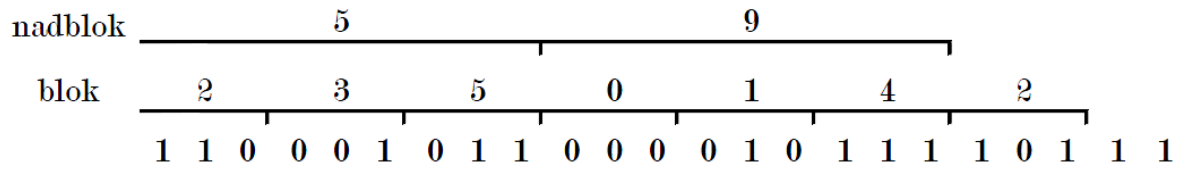
RRR struktura je struktura koja za dani niz nula i jedinica proizvoljne duljine vraća broj postavljenih jedinica do određene pozicije. Općenito, za dobivanje broja jedinica u nizu nula i jedinica, potrebno je proći kroz cijeli niz i provjeravati svaki element niza. Takvo prebrojavanje niza odvija se u linearnoj vremenskoj složenosti $O(n)$ gdje je n duljina niza. RRR strukturom se ukupan broj jedinica do određene pozicije u nizu može izračunati u konstantnom vremenu $O(1)$. Također, korištenjem RRR strukture se postiže i implicitna kompresija.

Ideja RRR strukture je da se inicijalno nad nizom konstruiraju blokovi (engl. *buckets*) i nadblokovi (engl. *superbuckets*) u kojima se pohranjuju brojevi jedinica za određene intervale niza. Na taj način se izbjegava pregledavanje cijelog dijela niza u potrazi za brojem jedinica, već je potrebno pregledati samo mali dio niza i određene vrijednosti pohranjene u blokovima i nadbloku.

RRR struktura implementirana u ovom projektu se djelomično razlikuje od originalne RRR strukture koju su predložili Raman, Raman and Rao [3]. U originalnoj RRR strukturi niz je podijeljen u blokove tako da je svaki blok predstavljen parom (SB,B) te su definirane dodatne tri tablice[4]. Implementacija u ovom projektu je nešto drugačije izvedena, ali je zadržala izračun broja jedinica u nizu u konstantnom vremenu te je opisana u nastavku teksta.

2.4.1 Konstruiranje RRR strukture

U ovoj implementaciji RRR strukture, niz duljine n se dijelio u blokove veličine l i nadblokove veličine l^2, a vrijednost l se pritom izračunavala kao logaritam duljine niza, tj. $l = log_2 n$. Stvorena su polja BS i SBS veličina $\lfloor n/l \rfloor$ i $\lfloor n/l^2 \rfloor$ za pohranjivanje vrijednosti blokova i nadblokova. Za stvaranje RRR strukture potrebno je proći kroz cijeli niz znamenki. Brojač jedinica je inicijalno postavljen na nula i povećava se kada se prolaskom kroz niz naiđe na element koji ima vrijednost 1. Kada se prođe kroz l elemenata niza, vrijednost brojača pohranjuje se u polje BS kao zbroj vrijednosti stvorenih blokova u još neispunjenom nadbloku. To znači da se blokovi u jednom nadploku pune kumulativno. Kada se prođe l^2 elemenata niza, vrijednost novog nadbloka jednaka je zbroju vrijednosti prošlog nadbloka (ako postoji) i vrijednosti brojača, odnosno zadnjeg konstruiranog bloka. Nakon izračuna vrijednosti nadbloka brojač se postavlja ponovo na vrijednost nula. Primjer ispunjenih vrijednosti u poljima BS i SBS dan je 2.6.



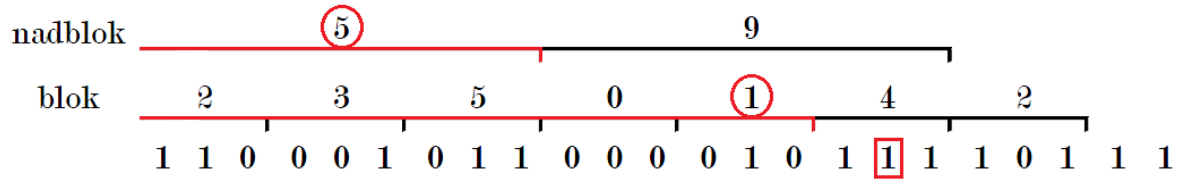
Slika 2.6: Ispunjene vrijednosti u poljima BS i SBS

Ukoliko se osvrnemo na primjer wavelet stabla prikazanim u prethodnom poglavlju (2.4), lako možemo izračunati da, ukoliko stablo izgrađujemo originalnim algoritmom, ukupna veličina izgrađenih polja BS i SBS jednaka je 17. Ako se stablo izgradi dijeljenjem abecede na osnovi broja pojavljivanja znakova unutar ulaznog niza, ukupna veličina izgrađenih BS i SBS polja iznosi 14. Već na ovako malom primjeru zamjećuje se poprilična razlika između dobivenih veličina polja BS i SBS.

2.4.2 Izračunavanje broja jedinica u nizu

Broj jedinica u nizu se izračunava kao zbroj vrijednosti zadnjeg popunjenog nadbloka i bloka do zadane pozicije te broja jedinica u ostatku niza koji nije obuhvaćen blokom. Zbog načina punjenja blokova i nadblokova, kod računanja ukupnog broja pojavljivanja jedinica u nizu do određene pozicije potrebno je pripaziti na određene situacije. Ako se tražena pozicija nalazi na mjestu u nizu jednakom $k \cdot l^2$, onda je broj pojavljivanja jedinica u podnizu jednak samo vrijednosti nadbloka, a ako se pozicija nalazi na mjestu u nizu $k \cdot l$, to znači da je vrijednost broja jedinica u podnizu dan ili samo nadblokom (ranije naveden slučaj) ili zbrojem vrijednosti nadbloka i bloka, te nema dijela niza koji nije obuhvaćen blokom/nadblokom za koji treba dodatno provjeravati vrijednosti znamenki. U ostalim slučajevima se vrijednostima zadnjeg popunjenog nadbloka i bloka pridodaje broj jedinica u dijelu niza od zadnjeg popunjenog bloka do pozicije do koje se traži izračun broja pojavljivanja jedinica u podnizu, i to je jedini dio u računanju kada je potrebno slijedno prolaziti kroz elemente niza prebrajajući pojavljivanje jedinica.

U nastavku je opisan postupak izračunavanja broja jedinica za primjer niza prikazanog slikom 2.7. Neka se želi izračunati broj pojavljivanja jedinica do pozicije 17. elementa niza (uključujući). Zadnji popunjeni nadblok moguće je pronaći formulom $indNadblok = \lfloor pozicija/l^2 \rfloor = \lfloor 17/9 \rfloor = 1$, a zadnji popunjeni blok formulom $indBlok = \lfloor pozicija/l \rfloor = \lfloor 17/3 \rfloor = 5$. Broj pojavljivanja jedinica u ostatku podniza računa se provjeravanjem broja jedinica u dijelu niza od pozicije $indBlok \cdot l + 1$ do tražene pozicije (uključujući). U ovom primjeru potrebno je provjeriti pojavljivanje jednica u još dva elementa niza, 16. i 17. elementu, čime se dobiva vrijednost dodatnih jedinica $dodatno = 2$. Ukupan broj pojavljivanja jedninica u nizu do 17. elementa niza se izračunava kao ukupno = $SBS(indNadblok) + BS(indBlok) + dodatno = 5 + 1 + 2 = 8$.



Slika 2.7: Ispunjene vrijednosti u poljima BS i SBS

FM-indeks

FM-indeks je samostojeći indeks koji su 2000. godine u svom radu opisali i objavili Ferragina i Manzini. Općenito indeks je podatkovna struktura koja omogućuje učinkovito dohvaćanje podataka. Samostojeći indeks (engl. *self-index*) je struktura koja se stvara nad određenim tekstom te ga indeksira na način da je memorijski proporcionalna veličini komprimiranog teksta te omogućava efikasno dohvaćanje i prebrojavanje dijelova tog teksta (bez potrebe za dekompresijom cijelog teksta).

3.1 Originalna implementacija

FM-indeks count je implementacija FM-indeksa koja služi za pronalaženje broja pojavljivanja uzorka P u tekstu S. Općenito se FM-indeks temelji na Burrows-Wheeler transformaciji i sufisknom polju, ali kod izvedbe samo prebrojavanja, ne i dohvaćanja svih pojavljivanja (FM-indeks find), struktura sufiksnog polja se može izostaviti. FM-indeks count je ustvari samo izvedba algoritma pretraživanja unatrag (engl. *backward search*) u kojemu se kreće s pretraživanjem od zadnjeg znaka uzorka P te ako uzorak postoji u tekstu, algoritam vraća koliko se puta on u tekstu ponavlja. Pronalaženje broja pojavljivanja uzorka u tekstu provodi se tako da se nad zadanim tekstom prvo stvori FM-indeks sa svojim strukturama OCC tablicom i C tablicom te se zatim koristeći stvorene tablice provodi prebrojavanje za zadani uzorak. U originalnoj implementaciji FM-indeksa [**?**], algoritam pretraživanja unatrag se provodi nad nizom znakova koji je dobiven iz originalnog teksta na koji su primjenjene različite operacije. Nad originalnim tekstom provedene su redom Burrows-Wheeler transformacija, *Move-To-Front* i *Run-length* kodiranje te stvaranje prefiksnog koda promjenjive duljine. Za provođenje algoritma potrebne su funkcije C(c) i Occ(c,i). C(c) daje broj znakova u (originalnom) tekstu koji su abecedno prije znaka c uključujući i ponavljanje pojedinih znakova. Occ(c,i) daje broj pojavljivanja znaka c u nizu B[1,i], i=1...$|S|$, gdje je S originalni tekst, B niz dobiven transformacijama originalnog teksta, a znak c je bilokoji znak iz abecede originalnog teksta. Pokazano je [neka referenca] da brzina izvođenja funkcije Occ(c,i) određuje brzinu izvođenja cijelog algoritma pretraživanja unatrag. U originalnoj implementaciji funkcija Occ(c,i) se računa korištenjem transformiranog (sažetog) originalnog teksta te nekoliko dodatnih pomoćnih struktura.

(opisat te strukture? ako bude ugrubo bit će nejasno, ali sve objasnit će biti predugačko)

3.2 Naša implementacija

U ovoj implementaciji FM-indeksa koristi se algoritam pretraživanja unatrag i ranije spomenuta struktura C tablice na isti način kao i u originalnoj implementaciji, ali je funkcija Occ(c,i)

izvedena korištenjem wavelet stabla i RRR struktura čiji je princip objašnjen u ranijem poglavlju.

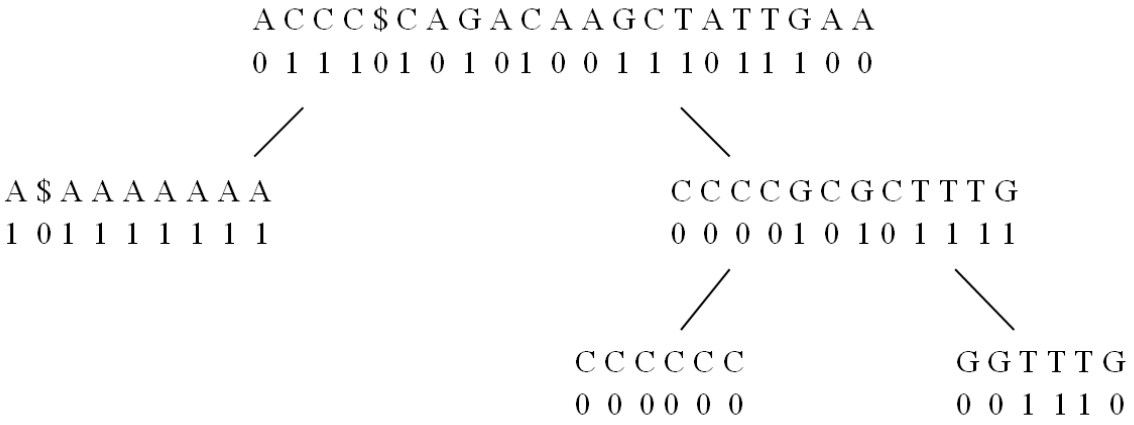
Prvi korak je dodavanje znaka $ na kraj niza te stvaranje C tablice koja za svaki znak iz abecede ulaznog teksta sadrži informacije koliko je znakova abecedno manjih ispred tog znaka abecede, uključujući i ponavljanje znakova. Pretpostavimo da ulazni niz s nadodanim znakom kraja niza glasi *ACAAGATGCACAATGTCCCA$.* Stvorena struktura (C tablica) izgledala bi ovako:

Tablica 3.1: C tablica

c	$	A	C	G	T
C(c)	0	1	9	15	18

Sljedeći korak je provođenje Burrows-Wheeler transformacije nad zadanim tekstom korištenjem *multikey quick* sortiranja čime se iz niza *ACAAGATGCCAATGTCCCA$* dobiva niz *ACCC$CAGACAAGCTATTGAA*. Burrows-Wheeler transformacija i *multikey quick* sortiranje detaljno su opisani u prethodnom poglavlju. Na temelju dobivenog niza stvara se OCC tablica koja je u ovoj implementaciji predstavljena wavelet stablom. Na temelju izgrađenog stabla prikazanog slikom 3.1, uz korištenje RRR struktura koje se stvaraju za svaki list stabla (i služe za računanje broja jedinica u dijelu niza pojedinog lista), lako se računaju vrijednosti Occ(c,i) koje su potrebne za algoritam pretraživanja unatrag. Struktura wavelet stabla i RRR struktura su detaljnije opisane u ranijim potpoglavljima.

SLIKA STABLA



Slika 3.1: Prikaz izgrađenog wavelet stabla za niz *ACCC$CAGACAAGCTATTGAA*

Ovim dijelom implementacije završeno je stvaranje same strukture FM-indeksa te je još samo potrebno implementirati algoritam koji će obuhvatiti prebrojavanje pojavljivanja nekog podniza u nizu nad kojim je napravljen FM-indeks, tj. *Count* dio implementacije projekta ***(? malo čudno zvuči).*** Za završni dio implementacije korišten je algoritam pretraživanja unatrag (engl. *backward search algorithm*). Algoritam je implementiran kao i u originalnoj implementaciji FM-indeksa [**?**], prema sljedećem pseudokodu:

Algorithm 1 Pretraživanje unatrag

```
Ulaz: P[0, p - 1] – zadani podniz
Izlaz: broj pojavljivanja podniza
i = p - 1;
c = P[i];
startPosition = C[c] + 1;
endPosition = C[c + 1];
for (i=i-1; i>=0; i--) do
   c = P[i];
   startPosition = C[c] + Occ(c, startPosition - 2) + 1;
   endPosition = C[c] + Occ(c, endPosition - 1);
end for
if endPosition >= startPosition then
   return endPosition - startPosition + 1;
else
   return 0;
end if
```

Ako se i dalje pretpostavi da se FM-indeks izgradio nad nizom *ACAAGATGCCAATGTCCCA* u kojemu se želi pronaći broj pojavljivanja podniza *ATG*, algoritam pretraživanja unatrag se provodi prema sljedećim koracima:

i = 2;
c = 'G';
startPosition = 15 + 1 = 16;
endPosition = 18;

———

i = 1;
c = 'T';
startPosition = 18 + 1 + 1 = 20;
endPosition = 18 + 3 = 21;

i = 0;
c = 'A';
startPosition = 1 + 6 + 1 = 8;
endPosition = 1 + 8 = 9;

STOP

broj pojavljivanja = 9 - 8 + 1 = 2

3.2.1 Nedostaci implementacije u programskom jeziku Java

section ili subsection??

Literatura

[1] David J. Burrows, Michael; Wheeler. A block sorting lossless data compression algorithm. *Technical Report 124, Digital Equipment Corporation*, 1994.

[2] J. Vitter R. Grossi, A. Gupta. High-order entropy-compressed text indexes. *In Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850, 2003.

[3] S. Srinivasa Rao R. Raman, V. Raman. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *SODA*, pages 233–242, 2002.

[4] Gonzalo Navarro Francisco Claude. Practical rank/select queries over arbitrary sequences.