

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Projekt iz kolegija bioinformatika

FM-indeks count

Autori:

Matea Donlić

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
2.1.5	Konstrukcija sufiksnog polja	6
2.2	Wavelet stablo	13
2.2.1	Konstruiranje wavelet stabla	13
2.2.2	Upiti nad wavelet stablom	14
2.3	RRR struktura	15
2.3.1	Konstruiranje RRR strukture	16
2.3.2	Izračunavanje broja jedinica u nizu - rang niza	16
3	FM-indeks	18
3.1	Originalna implementacija	18
3.2	Implementacija FM-indeksa	19
3.3	Nedostaci implementacije u programskom jeziku Java	21
4	Testiranje	22
4.1	Vrijeme izgradnje FM-indeksa	22
4.2	Vrijeme obavljanja <i>count</i> upita	29
4.3	Memorijsko zauzeće FM-indeksa	31
5	Zaključak	39
6	Literatura	40

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 (engl. *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-indeks čija je implementacija i zadatak ovoga projekta.

Pomoćni algoritmi

U ovom poglavlju opisani su algoritmi koji su potrebni za konstruiranje i korištenje FM-indeksa u svrhu prebrojavanja broja ponavljanja nekog 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 \$ (ili neki drugi znak koji se ne pojavljuje u ulaznom nizu i leksikografski je manji od svih ostalih znakova u nizu) 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 iščitava se iz posljednjeg stupca matrice dobivene u koraku 3.

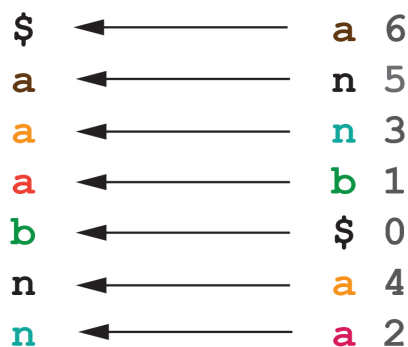
Tijek algoritma za riječ *banana* prikazan je na slici 2.1. Vidi se kako je BWT transformacija te riječi jednaka *annb\$aa*.



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 poznato 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 povećanje tekstove je jednostavno nemoguće napraviti sve njihove rotacije i

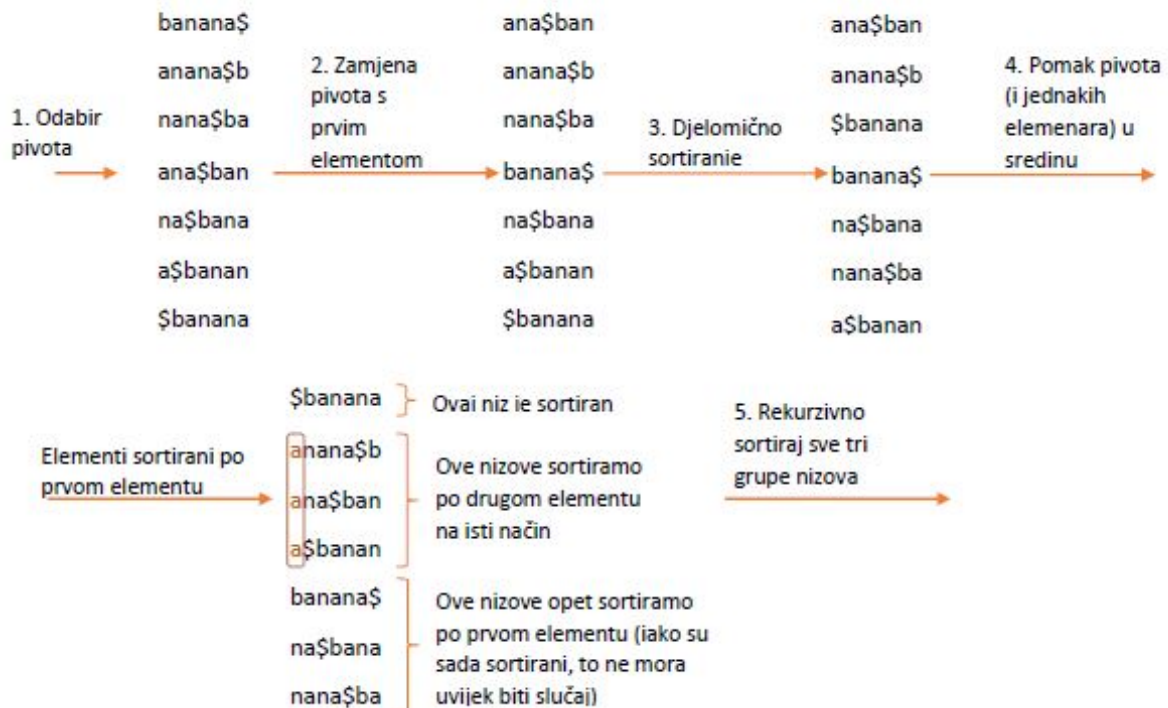
onda ih sortirati, jer bi memorijsko zauzeće bilo preveliko. Iz tog razloga potrebno je koristiti algoritme koji BWT transformaciju mogu obaviti memorijski i vremenski efikasno. U nastavku ćemo pogledati dva takva algoritma te vidjeti koje su njihove prednosti i nedostaci.

2.1.4 Multikey quicksort

Prvo ćemo proučiti jedan tip algoritma za sortiranje koji se temelji na quick sort algoritmu predloženom u [2]. 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 2.3. 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 nalaze nizovi koji započinju s elementom koji je veći od pivota (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 pivota, 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 algoritma 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. Tako se umjesto zamjene čitavih nizova jednostavno zamijene pokazivači u tom polju.

Prednost ovog postupka je 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 znači 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



Slika 2.3: Multikey quicksort algoritam

za paralelizaciju, spomenutu paralelizaciju bi bilo dobro napraviti. Najveći nedostaci ovog postupka su svakako nešto veća složenost naspram konstrukcije BWT-a korištenjem sufiksnog polja, te također rekurzivna priroda algoritma koja zahtijeva puno rekurzivnih poziva i iz tog razloga također i alokaciju veće količine memorije za stog.

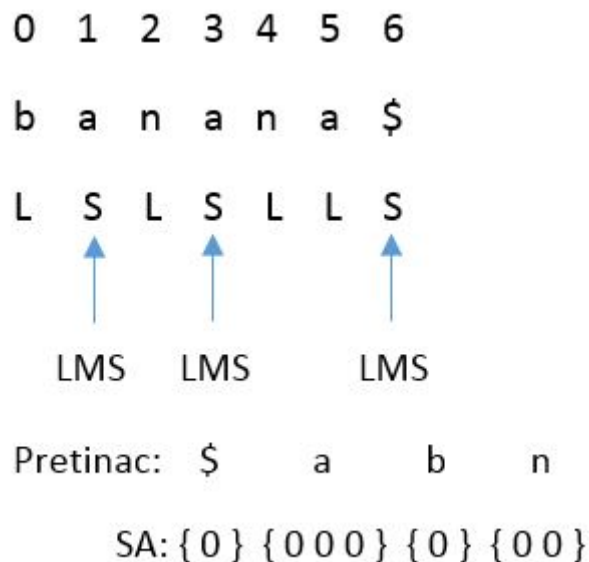
Iako se ovdje radi o veoma zanimljivom algoritmu (algoritmu koji nije naravno primjenjiv samo na pronalazak BWT transformacije), on se nažalost pokazao presporim u usporedbi s izračunavanjem BWT transformacije korištenjem sufiksnog polja te se iz tog razloga dalje neće razmatrati.

2.1.5 Konstrukcija sufiksnog polja

Sufiksno polje jest jednostavna struktura podataka koje je sastavljena od niza cijelih brojeva koji sadrže početna mjesta abecedno poredanih sufiksa nekog niza. Dobra stvar kod sufiksnog polja jest da se iz njega može izračunati BWT transformacija u linearnom vremenu. Još veća prednost ovakvog postupka jest da se sufiksno polje isto tako može konstruirati u linearnom vremenu. Iako postoje mnogi algoritmi za konstrukciju sufiksnih polja, u ovoj implementaciji iskorišten je algoritam predložen u [3] i [4].

Prije nego što se krene na objašnavanje samog algoritma potrebno je uvesti određenu terminologiju koju ćemo koristiti za objašnjenje postupaka ovog algoritma. Ulazni niz znakova ćemo označiti sa S . Neka je ulazni niz znakova duljine n , i neka je oznaka indeksiranja jednog elementa $S[i]$ pri čemu je $i \in [0, n - 1]$. Posljednja stvar koju je bitno napomenuti jest da znakovni niz mora obavezno završavati znakom koji je strogo manji

od svih znakova u ostatku niza. Definirat ćemo dvije različite vrste znakova, L-znakove i S-znakove. S-znak je onaj znak koji je manji od znaka koji dolazi neposredno iza njega (odnosno formalnije, ako gledamo i -ti znak onda mora vrijediti $S[i] < S[i + 1]$) ili je jednak znaku koji se nalazi desno od njega i taj znak je S-znak (formalnije $S[i] = S[i + 1]$ i $S[i + 1]$ je S-znak). L-znak je onaj znak za koji vrijedi da je veći od znaka koji se nalazi desno od tog znaka (odnosno formalnije, ako gledamo i -ti znak onda mora vrijediti $S[i] > S[i + 1]$) ili je jednak znaku koji se nalazi desno od njega i taj znak je L-znak (formalnije $S[i] = S[i + 1]$ i $S[i + 1]$ je L-znak). Iznimka od ovog pravila jest zadnji znak koji će uvijek biti definiran kao S-znak. Ovaj koncept možemo proširiti i na sufikse, pa reći da je S-sufiks sufiks koji počinje S-znakom i L-sufiks sufiks koji počinje L-znakom. Uvest ćemo još pojam LMS-znak i LMS-podniz. LMS-znak je S-znak kojemu se na lijevom mjestu nalazi L-znak, odnosno vrijedi $S[i]$ je S-znak i $S[i - 1]$ je L-znak. LMS-podniz je niz koji započinje LMS-znakom i završava LMS-znakom, te se u njemu ne nalaze drugi LMS-znakovi. Primjer jednog ovako označenog niza može se vidjeti na slici 2.4.



Slika 2.4: Oznake znakova u jednom nizu

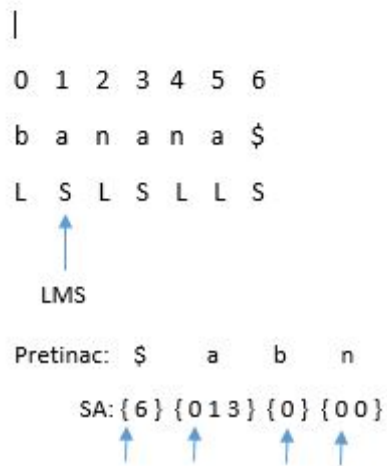
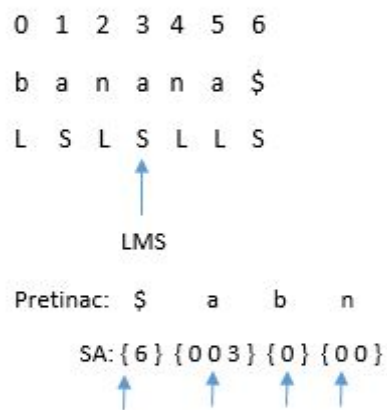
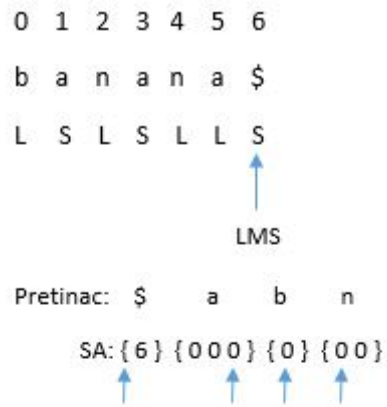
Sada kada su uvedene osnovne oznake, možemo pojasniti samu ideju algoritma. Osnovna ideja algoritma predložena je već u [5]. U svojoj osnovnoj inačici, algoritam se zasniva na svojstvu da ako znamo poredak svih S-sufiksa ona možemo odrediti i poredak svih L-sufiksa. Nažalost, problem je ostao u tome kako zapravo efikasno sortirati S-sufikse. U spomenutom radu je predložen način kako bi se problem mogao riješiti korištenjem S-udaljenosti, što je opet zahtijevalo dodatnog memorijskog prostora. Ta ideja je razrađena i poboljšana te je konačno predložen novi algoritam u [3] koji se zasniva na istoj ideji, ali uvodi neke promjene koje su rezultirale mnogo boljim ponašanjem ovog algoritma. Prvo bitno poboljšanje koje je uvedeno jest da se umjesto sortiranja S-sufiksa mogu sortirati LMS-sufiksi, koji su sasvim dovoljni kako bi se odredio poredak L-sufiksa, pomoću kojih se konačno može odrediti poredak S-sufiksa. Kako su LMS-sufiksi obično dulji, to znači kako ćemo vjerojatno imati manji broj takvih sufiksa koje moramo sortirati nego bi to bio slučaj da sortiramo sve S-sufikse (u najgorem slučaju broj LMS-podnizova je polovica

duljine ulaznog niza). Drugo veoma zanimljivo poboljšanje jest da se za sortiranje LMS-podnizova ne mora koristiti nikakav posebni algoritam za sortiranje nego se može koristiti isti postupak kao i kod sortiranja L-znakova i S-znakova. U nastavku ćemo pogledati detaljnije korake ovog algoritma.

Prvo je potrebno za sve znakove odrediti jesu li S-znakovi, L-znakovi ili LMS-znakovi. Nakon tog koraka slijedi postupak induciranog sortiranja. Definirajmo prvo inducirano sortiranje. To je postupak kojim ćemo iz LMS-sufiksa inducirati poredak L-sufiksa i potom S-sufiksa. Ovo je zapravo centralni i najvažniji dio algoritma. Za rad ovog algoritma potrebno nam je nekoliko struktura: ulazni niz, sufiksno polje, niz s oznakama tipova znakova i pokazivači na pretince. Sve strukture su već ranije spomenute i poznate, osim pokazivača na pretince. Pretince ćemo zapravo definirati kao područja u sufiksnom polju u kojima se nalaze indeksi sufiksa koji počinju s istim početnim znakom. Broj pretinaca će dakle biti jednak veličini abecede (originalne abecede proširene s dodatnim znakom koji je manji od svih ostalih). Kada imamo sve te strukture možemo provesti postupak induciranog sortiranja. Postupak induciranog sortiranja se provodi kroz tri koraka opisanih u nastavku.

Prije početka prvog koraka pretpostavimo da svi pokazivači na pretince pokazuju na krajeve tih pretinaca, i neka su vrijednosti elemenata sufiksnog polja inicijalno postavljene na vrijednost 0. Prvi korak se sastoji od toga da prođemo kroz niz i da sve LMS-znakove stavimo u odgovarajući pretinac (smjer prolaska kroz niz je u potpunosti nebitan, može se prolaziti od početka ili od kraja niza). Kada stavimo LMS-znak u odgovarajući pretinac potrebno je pokazivač na taj pretinac smanjiti tako da pokazuje na iduće prazno mjesto. Ovaj korak algoritma detaljno je prikazan na slici 2.5.

Prije početka drugog koraka potrebno je pokazivače postaviti na početke svih pretinaca. Drugi korak algoritma jest induciranje L-znakova. U ovom koraku prolazimo kroz sufiksno polje s lijeva na desno i za svaki element koji je veći od 0, dakle za elemente za koje vrijedi $SA[i] > 0$, obavljammo postupak indukcije. Svaki taj element zapravo predstavlja indeks nekog znaka u ulaznom nizu. Za svaki element iz sufiksnog polja ispitat ćemo da li je znak iz ulaznog niza koji se nalazi na poziciji koja je određena vrijednošću trenutnog elementa sufiksnog polja umanjenoj za jedan L-znak. Mnogo jednostavnije rečeno ispitujemo da li je znak $S[SA[i]]$ L-znak, pri čemu je i indeks elementa sufiksnog polja na kojem se trenutno nalazimo. Ukoliko je taj znak L-znak, onda u pretinac koji predstavlja taj znak stavimo njegov indeks i uvećamo pokazivač na taj pretinac.



Slika 2.5: Prvi korak u induciranom sortiranju

0	1	2	3	4	5	6
b	a	n	a	n	a	\$
L	S	L	S	L	L	S

i=0

Pretinac: \$ a b n

SA: {6} {5 1 3} {0} {0 0}

0	1	2	3	4	5	6
b	a	n	a	n	a	\$
L	S	L	S	L	L	S

i=1

Pretinac: \$ a b n

SA: {6} {5 1 3} {0} {4 0}

0	1	2	3	4	5	6
b	a	n	a	n	a	\$
L	S	L	S	L	L	S

i=2

Pretinac: \$ a b n

SA: {6} {5 1 3} {0} {4 0}

0	1	2	3	4	5	6
b	a	n	a	n	a	\$
L	S	L	S	L	L	S

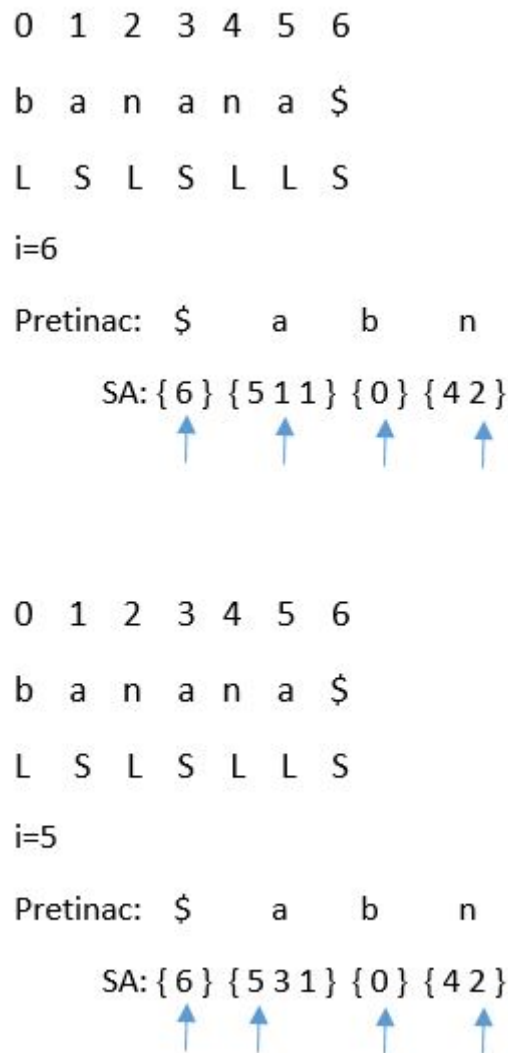
i=3

Pretinac: \$ a b n

SA: {6} {5 1 3} {0} {4 2}

Slika 2.6: Drugi korak u induciranom sortiranju

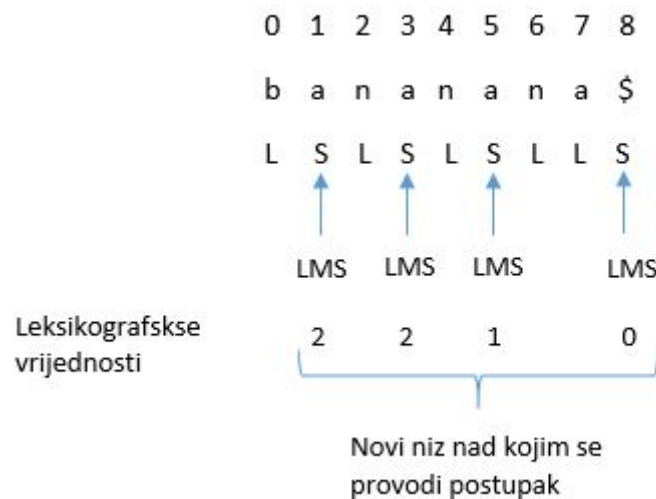
Prije početka treće i posljednje faze induciranog sortiranja moramo pokazivače postaviti na kraj pretinaca. Treći dio induciranog sortiranja je veoma sličan drugom koraku. Kroz sufiksno polje se ovaj puta prolazi od kraja prema početku. Opet izostavljamo elemente koji su jednaki 0. Za svaki element iz sufiksnog polja ispitat ćemo da li je znak iz ulaznog niza, koji se nalazi na poziciji koja je određena vrijednošću trenutnog elementa sufiksnog polja umanjenoj za jedan, S-znak, odnosno ispitujemo da li je znak $S[SA[i]]$ S-znak, pri čemu je i indeks elementa sufiksnog polja na kojem se trenutno nalazimo. Ukoliko je taj znak S-znak, onda u pretinac koji predstavlja taj znak stavimo njegov indeks i smanjimo vrijednost pokazivača na taj pretinac.



Slika 2.7: Treći korak u induciranom sortiranju

S ta tri koraka završen je postupak induciranog sortiranja koji predstavlja središnji dio ovog algoritma za izgradnju sufiksnog polja. Ovim postupkom smo zapravo djelomično sortirali sve LMS-sufikse. Problem predstavljaju LMS-sufiksi koji su jednaki, jer se njih na spomenuti način ne može sortirati. Kako bi sortirali LMS-sufikse koji su jednaki moramo još pogledati znakove, odnosno LMS-podnizove koji dolaze iza tih LMS-podnizova.

Definirajmo prvo što znači da su dva LMS-podniza jednaka. Dva LMS-podniza su jednaka ako su jednake duljine, ako na istim pozicijama imaju jednake znakove i dodatno, ako su znakovi na istim pozicijama u oba niza istog tipa (S-znak ili L-znak). Kako bi odredili njihov poredak moramo vidjeti koji LMS-podniz slijedi nakon tih LMS-podnizova. To veoma lako možemo napraviti na način da svakom sufiksu pridijelimo neku leksičku vrijednost, odnosno neki broj. LMS-sufiksi koji su manji dobit će manju leksičku vrijednost, oni koji su veći dobit će veću leksičku vrijednost, dok će jednaki LMS-podniz dobiti jednake leksičke vrijednosti. Podjela leksičkih vrijednosti pojedinim LMS-podnizovima je prikazana na slici 2.8. Na taj način smo originalni niz zamijenili kompaktnijim nizom gdje je svaki LMS-podniz predložen svojom leksičkom vrijednošću. Sada možemo ponoviti cijeli algoritam i na isti način kao i za originalni niz (jer i znakovi su zapravo predstavljeni brojevima, tako da to ništa ne mijenja na stvari) i tako nastavljamo dok ne uspijemo odrediti poredak svih LMS-podnizova. Kada je on određen, induciranim sortiranjem lako dobijemo poredak svih L-sufiksa i S-sufiksa te je time postupak računanja sufiksnog polja dovršen.



Slika 2.8: Podjela leksičkih vrijednosti LMS-podnizovima

Sada kada smo pogledali sve korake algoritma zasebno, rezimirajmo cjelokupni algoritam još jednom. U prvom koraku je potrebno odrediti tipove svih znakova (da li je pojedini znak L-znak ili S-znak), kao i koji su znakovi LMS-znakovi. Drugi korak algoritma je provođenje inducirano sortiranja nad ulaznim nizom. Time smo dobili međusobno sortirane LMS-podnizove, no bitno je za zapamtiti da su sortirani samo oni koji su međusobno u potpunosti različiti. Ukoliko je dva ili više LMS-podniza u potpunosti jednako (po jednakosti definiranoj ranije), onda ovim korakom nije jednoznačno određen poredak LMS-podnizova, odnosno, oni nisu u potpunosti sortirani. Kako bi odredili poredak LMS-nizova potrebno ih je dodatno sortirati, i to radimo tako da sve LMS-podnizove zamijenimo leksikografskim znakovima po principu da LMS-podnizovi koji se u sortiranom poretku pojavljuju ranije dobiju manje leksikografske znakove, jednakim LMS-podnizovima pridijele se jednaki leksikografski znakovi i onim LMS-podnizovima koji se pojavljuju kasnije pridijele veće leksičke vrijednosti. Time smo stvorili novi, redu-

cirani problem koji možemo riješiti istim postupkom kao i originalni problem. Kada je riješen ovaj reducirani problem, jednoznačno nam je određen poredak LMS-podnizova, i sada ukoliko provedemo ponovno inducirano sortiranje, dobit ćemo sufiksno polje.

Kao što vidimo radi se o vrlo jednostavnom, konciznom i elegantnom algoritmu koji u linearnom vremenu može izgraditi sufiksno polje. Teoretski, memorijsko zauzeće algoritma je $O(5n)$, no u implementaciji je obično nešto veće zbog dodatnih struktura koje su potrebne za rad algoritma (primjerice polje koje sadrži tipove svih znakova kako se ono ne bi moralo ponovo određivati u pojedinim koracima algoritma). Možda se kao jedan nedostatak algoritma može činiti potreba za rekurzivnim pozivim tijekom rješavanja problema, no to obično nije tako veliki problem kao što se može činiti jer je najčešće broj rekurzivnih poziva dosta malen, čak i za veće nizove.

2.2 Wavelet stablo

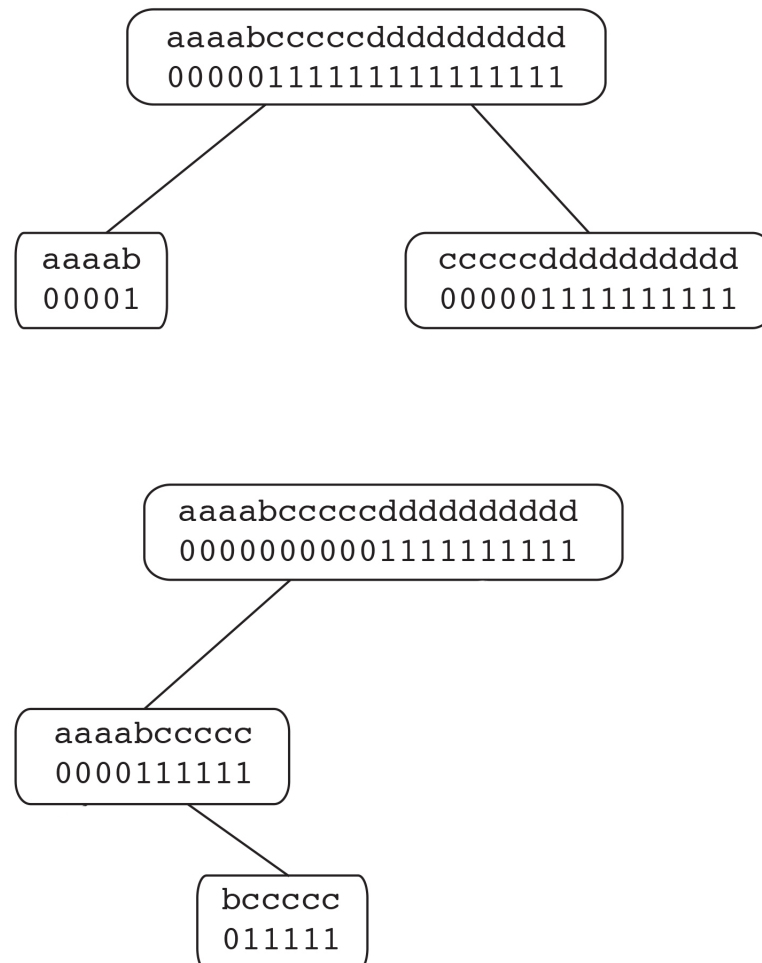
Wavelet stablo (engl. *wavelet tree*) je podatkovna struktura koja tekstualni niz organizira u hijerarhiju nizova nula i jedinica - *bit-vektora*. Ova struktura omogućuje pronalazak broja pojavljivanja do nekog indeksa u nizu - *ranga* nekog znaka sa složenošću od $O(\log \Sigma)$, gdje je Σ veličina abecede danog tekstualnog niza. Velika prednost ove strukture je činjenica da je ona samokomprimirajuća, iz razloga što se znakovi pohranjuju kao bitovi.

Stablo se izgrađuje rekurzivno, razdjeljujući 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.

2.2.1 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 sljedeću 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 predložena je u [6] - abeceda se 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 2.9 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.

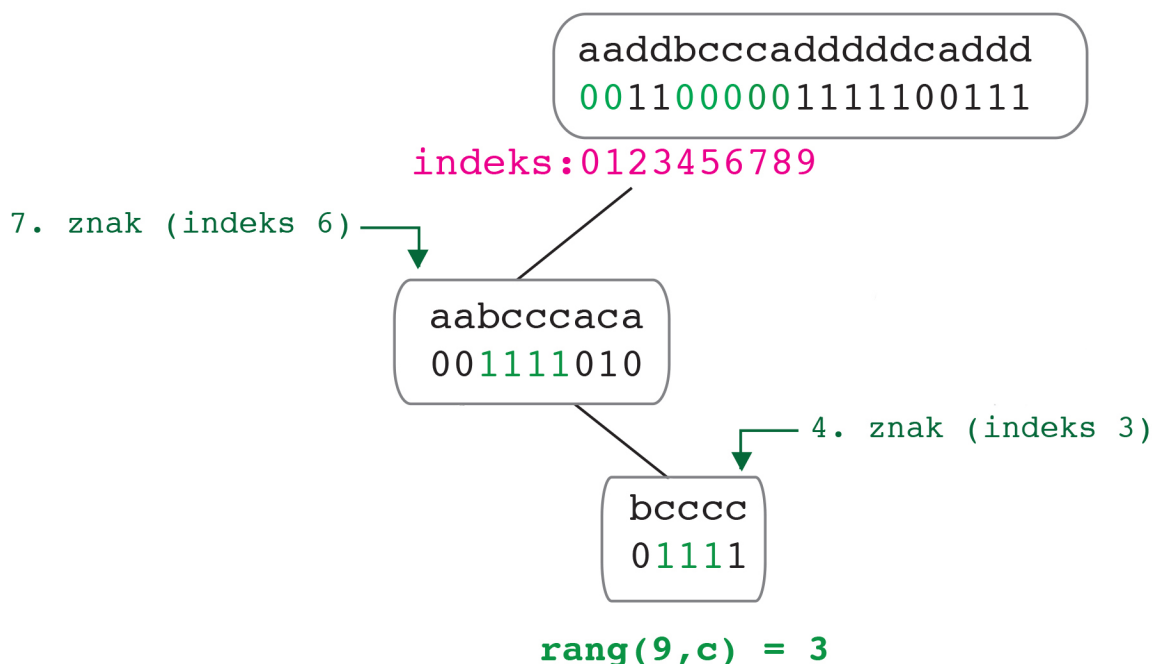


Slika 2.9: Razlika između originalnog i ovdje implementiranog wavelet stabla

2.2.2 Upiti nad wavelet stablom

Slika 2.10 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 sljedeću 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 prвих 7 znakova. Važno je primijetiti 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 2.10: Razlika između originalnog i ovdje implementiranog wavelet stabla

2.3 RRR struktura

RRR struktura je struktura koja za dani niz nula i jedinica proizvoljne duljine vraća broj postavljenih jedinica do određene pozicije - *rang* niza. 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 $O(1)$ [7]. Također, korištenjem RRR strukture se postiže i implicitna kompresija [8].

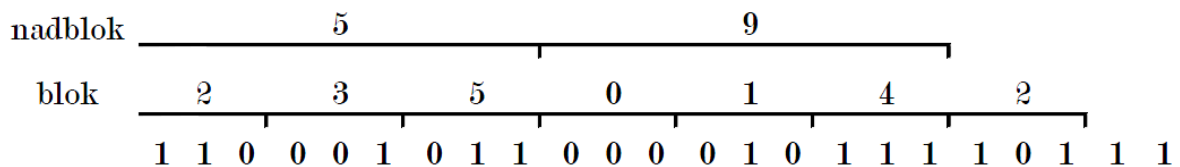
Ideja RRR strukture je da se inicijalno nad nizom konstruiraju blokovi (engl. *buckets*) i nadblokovi (engl. *superbuckets*) u kojima se pohranjuju brojevi jedinica za određene intervale niza. Na taj način se izbjegava pregledavanje cijelog dijela niza u potrazi za brojem jedinica, već je potrebno pregledati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

RRR struktura implementirana u ovom projektu se djelomično razlikuje od originalne strukture opisane u [7]. U originalnoj RRR strukturi niz je podijeljen u blokove tako da je svaki blok predstavljen parom (SB,B) te su definirane dodatne tri tablice [9]. 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.

2.3.1 Konstruiranje RRR strukture

U ovoj implementaciji RRR strukture, niz duljine n se dijelio u blokove veličine l i nadblokove veličine l^2 . U idućem potpoglavlju opisan je način računanja ranga, odnosno broja jedinica u nizu, te se može vidjeti da je za računanje ranga potrebno slijedno proći kroz najviše l elemenata niza, a budući da je veličina bloka l fiksne duljine, određivanje ranga se računa u konstantnom vremenu. Vrijednost veličine bloka je potrebno pažljivo odrediti jer o njoj ovise i vremenska složenost i memorijsko zauzeće. Već je spomenuto da se u postupku računanja ranga maksimalno mora slijedno proći kroz niz duljine l znakova pa je iz tog razloga poželjno što više smanjiti tu vrijednost. S druge strane, ako su blokovi manje veličine, trebat će se pohranjivati veći broj blokova što uzrokuje veće memorijsko zauzeće. Detaljnija razrada problema kako najbolje odabrati vrijednost veličine bloka (što ovisi i o veličini brojeva koji će se pohranjivati u blokove i nadblokove) opisana je u [10]. U ovom radu korištena je veličina bloka l koja se računala kao logaritam duljine niza, tj. $l = \log_2 n$. Također, 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 nadbloku 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 slikom 2.11.



Slika 2.11: Ispunjene vrijednosti u poljima BS i SBS

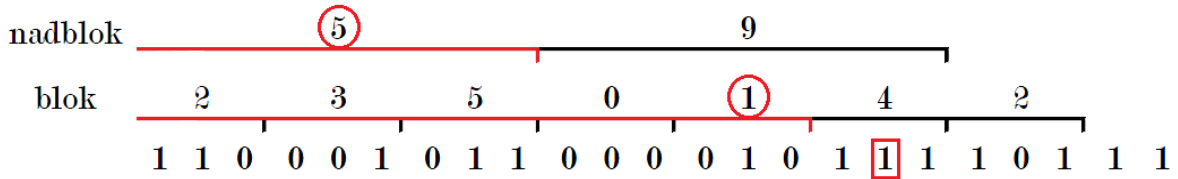
Ukoliko se osvrnemo na primjer wavelet stabla prikazanim u prethodnom poglavlju (slika 2.9), lako možemo izračunati da je, ukoliko stablo izgrađujemo originalnim algoritmom, ukupna veličina izgrađenih polja BS i SBS jednaka 17. Ako se stablo izgradi dijeljenjem abecede na osnovi broja pojavljivanja znakova 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.

2.3.2 Izračunavanje broja jedinica u nizu - rang niza

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 2.12. 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 jedinica u još dva elementa niza, 16. i 17. elementu, čime se dobiva vrijednost dodatnih jedinica $dodatno = 2$. Ukupan broj pojavljivanja jedinica u nizu do 17. elementa niza se izračunava kao ukupno $= SBS(indNadblok) + BS(indBlok) + dodatno = 5 + 1 + 2 = 8$.



Slika 2.12: 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 [11]. 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). Osim originalne implementacije opisane u nastavku, tokom godina napravljene su različite inačice FM-indeksa (npr. *Alphabet-Friendly FM-index* - Ferragina, 2004.).

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 primijenjene različite operacije [11]. 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 bilo koji znak iz abecede originalnog teksta. Pokazano je da brzina izvođenja funkcije $Occ(c,i)$ određuje brzinu izvođenja cijelog algoritma pretraživanja unatrag [12]. U originalnoj implementaciji funkcija $Occ(c,i)$ se računa korištenjem transformiranog (sažetog) originalnog teksta te nekoliko dodatnih pomoćnih struktura.

3.2 Implementacija FM-indeksa

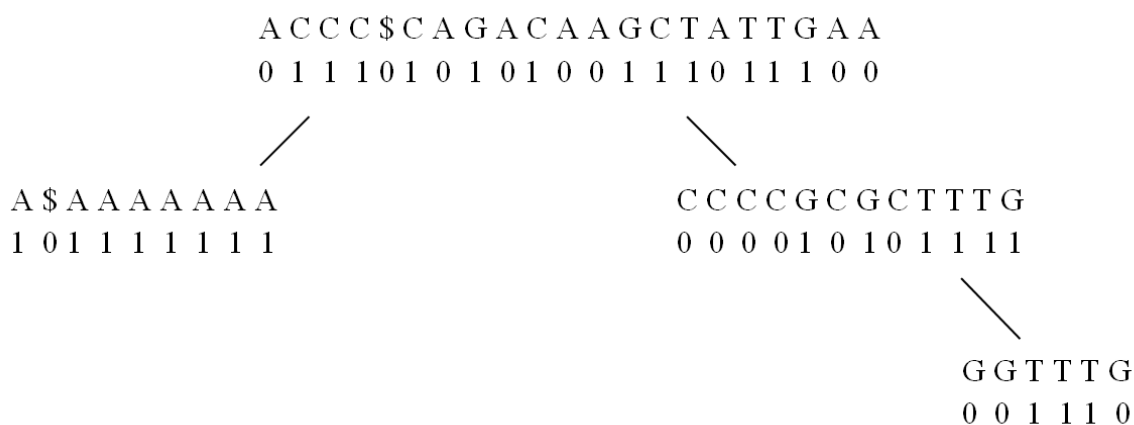
U ovoj implementaciji FM-indeksa koristi se algoritam pretraživanja unatrag i ranije spomenuta struktura C tablice na isti način kao i u originalnoj implementaciji, ali je funkcija $\text{Occ}(c,i)$ izvedena korištenjem wavelet stabla i RRR struktura čiji je princip objašnjen u ranijem poglavlju, a napravljen po uzoru na [10].

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 sufiksnog polja, čime se iz niza *ACAAGATGCCAATGTCCCA\$* dobiva niz *ACCC\$CAGACAAGCTATTGAA*. BWT i algoritam izgradnje sufiksnog polja 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 $\text{Occ}(c,i)$ koje su potrebne za algoritam pretraživanja unatrag. Struktura wavelet stabla i RRR struktura su detaljnije opisane u ranijim potpoglavljima.



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 indeksa.

Za završni dio implementacije korišten je algoritam pretraživanja unatrag (engl. *backward search algorithm*). Algoritam je implementiran kao i u originalnoj implementaciji FM-indeksa [11], 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 ( $i=i-1; i \geq 0; i--$ ) do
     $c = P[i];$ 
     $startPosition = C[c] + Occ(c, startPosition - 2) + 1;$ 
     $endPosition = C[c] + Occ(c, endPosition - 1);$ 
end for
if  $endPosition \geq startPosition$  then
    return  $endPosition - startPosition + 1;$ 
else
    return 0;
end if

```

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

Inicijalizacija:

```

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

```

1. korak petlje:

```

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

```

2. korak petlje:

```

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

```

Izlaz: broj pojavljivanja = $9 - 8 + 1 = 2$

3.3 Nedostaci implementacije u programskom jeziku Java

Implementacija ovog algoritma napravljena je u programskom jeziku Java. Iako Java kao programski jezik nudi mnoge pogodnosti, kao jednostavnu sintaksu, dobre popratne biblioteke i automatsko upravljanje memorijom, ipak postoje i određeni nedostaci ovakvog jezika. Možda najveći nedostatak Java programskog jezika dolazi upravo od jedne prednosti samog jezika, a to je automatsko upravljanje memorijom. To znači da se programski jezik sam brine oko oslobađanja i zauzimanja memorije, što može biti problematično prilikom izvođenja aplikacije koje troše puno memorije. U tim slučajevima potrebno je ručno pozvati *garbage collector* kako bi oslobodio zauzetu memoriju, no to nažalost opet ima utjecaj na brzinu izvođenja, posebno ukoliko je takvih poziva više. Osim toga, Java virtualni stroj često zauzme više memorije nego joj je uistinu potrebno, što se jako dobro može vidjeti korištenjem alata za profiliranje. To svakako predstavlja problem ukoliko program prilikom izvođenja zauzima velike količine memorije, jer se one dodatno povećaju zbog tog praznog prostora kojeg zauzme Java virtualni stroj. Iako postoje naredbe kojima se može pokušati regulirati količina memorije koju će virtualna stroj smjeti zauzeti, kao i naredbe kojima bi se trebalo moći natjerati virtualnu stroj da memoriju koju je zauzela, a ne koristi ju, vrati operacijskom sustavu, uočeno je kako je potrebno dobro podesiti parametre tih naredbi prilikom izvođenja kako bi se dobili što bolji rezultati. Osim toga, Java virtualni stroj sam po sebi zauzima dosta veliku količinu memorije (prilikom testiranja to je otprilike bilo oko 10MB), što svakako predstavlja još jedan nedostatak programskog jezika Java. Možda se memorija od 10MB ne čini kao velika količina memorije, posebice uzevši u obzir raspoloživu količinu memorije u današnjim računalima, no to svejedno nije ni zanemariva količina memorije, posebice ako se indeksiraju kraći znakovni nizovi.

Još jedan nedostatak Java programskog jezika jest što primitivni tip podatka *char* zauzima dva bajta memorije. To znači da zapravo imamo dvostruko veću memorijsku složenost nego što bi htjeli. Iz tog razloga, u implementaciji se koristi tip podatka *byte* kako bi se uštedjelo na memoriji. Time je limitiran broj različitih znakova koje možemo prikazati, no kako je glavni fokus ipak stavljen na DNK sekvence i nizove proteina, ovo ograničenje ne predstavlja veliki problem. Dodatno je postavljeno još jedno ograničenje, a to je da implementacija može raditi nad osnovnim skupom *ASCII* znakova. To je napravljeno iz razloga što Java programski jezik nema *unsigned* inačice tipova podataka (opet jedan veliki nedostatak), te ukoliko bi se htjelo raditi nad proširenim skupom *ASCII* znakova, moralo bi se paziti na preslikavanje negativnih vrijednosti u njihove pozitivne ekvivalente, zadržavajući leksički poredak. Taj postupak bi uveo dodatne operacije koje bi mogle utjecati na performanse izvođenja ovog algoritma, te iz tog razloga nije napravljen (iako doduše implementaciju nije teško proširiti tako da radi i s proširenim skupom *ASCII* znakova).

Konačno, upitna je i brzina Java programskog jezika naspram jezika C/C++. Ovdje se neće ulaziti u detalje oko tog pitanja, jer različiti izvori nude različita mjerenja performansi. Programi koji su napisani u programskom jeziku Java obično postižu nešto lošije rezultate, no to sve ovisi o tome koliko su efikasno napravljene pojedine implementacije, kao i o problemu koji je rješavan.

Testiranje

Testiranja su obavljena na računalu s procesorom Intel Core i7 920 @ 2.66 GHz, s 6 GB RAM memorije i na operacijskom sustavu Windows 7. Razlog zašto testiranja nisu bila obavljena na Biolinuxu (ili nekoj drugoj Linux distribuciji) je taj što se Biolinux, na računalu koje se koristilo za testiranja, pokreće u virtualnom stroju, pa je upitno kako bi to utjecalo na performanse izvođenja samog programa. Za mjerenje vremena izvođenja korištena je metoda *currentTimeMillis* razreda *System*, dok je za mjerenje potrošnje memorije korišten alat *Jprofiler*.

Za obavljanje testiranja korištena su tri tipa tekstualnih datoteka. Prvi tip tekstualnih datoteka se sastojao od nizova nukleotida i dodatnih znakova koji se još mogu pojaviti u takvim nizovima. Drugi tip tekstualnih datoteka sastojao se od nizova znakova koji predstavljaju proteine. Posljednji tip tekstualnih datoteka koje su korištene sastojale su se od nizova slučajno generiranih znakova iz osnovnog ASCII skupa znakova i to od znakova koji se mogu ispisati (dakle ne od specijalnih znakova). Prvi i drugi tip datoteka su preuzeti s [13] te su ponešto prilagođene za obavljanje testiranja. Treći tip datoteka je bio generiran uz pomoć posebnog programa. Spomenute datoteke se razlikuju po tome koliko se različitih znakova može pojaviti u njima, odnosno koje su veličine abeceda iz kojih su te datoteke generirane. Tablica 4.1 prikazuje veličinu abeceda iz kojih su generirane određene datoteke, odnosno koliko najviše različitih znakova mogu sadržavati.

Tablica 4.1: Maksimalna veličina abecede pojedinih datoteka

Vrsta niza	Broj različitih znakova
DNK	15
Proteini	25
ASCII	95

U ovom poglavlju pogledat ćemo performanse koje pruža implementirani FM-indeks i to u tri kategorije: brzina izgradnje, brzina obavljanja *count* upita i memorijska potrošnja.

4.1 Vrijeme izgradnje FM-indeksa

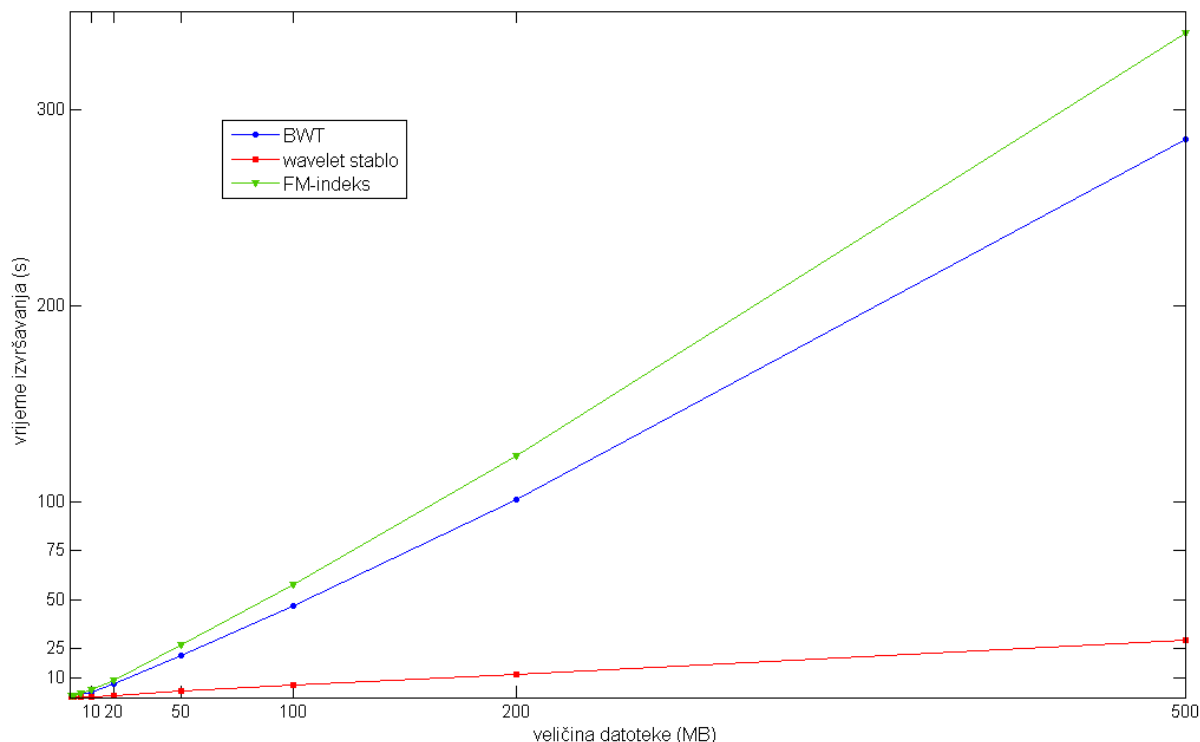
Pogledajmo prvo performanse izgradnje FM-indeksa. Ovo je možda i jedan od najbitnijih faktora prilikom ocjene koliko je neka implementacija dobra, pošto je izgradnja samog indeksa računalno veoma zahtjevana operacija. Kako bi rezultati koji će biti prikazani u ovom poglavlju bili što relevantniji, vrijednosti koje su prikazane u tablicama izračunate su kao prosjek od deset izgradnji FM-indeksa za svaku pojedinu veličinu datoteka. Osim toga, prilikom provođenja ovih eksperimenata nije bio pokrenut alat za profiliranje, jer on

dodatno usporava izvođenje programa, stoga su mjerenja za memorijsku potrošnju obavljena zasebno. Eksperimenti su obavljeni za svaki tip datoteke koji bio ranije spomenut i to za 9 različitih veličina takvih datoteka, od manjih (1MB) pa sve do većih (500MB). Osim toga, posebno su izdvojena vremena izgradnje BWT transformacije i wavelet stabla, koji ipak sačinjavaju glavne građevne blokove FM-indeksa, pa je iz tog razloga dobro vidjeti koliko se mijenja i vrijeme izgradnje ovih struktura s promjenom veličine ulazne datoteke.

Pogledajmo prvo rezultate koji su dobiveni za datoteke koje su sadržavale nizove nukleotida. Tablica 4.2 prikazuje vremena izgradnje FM-indeksa nad tipom datoteka koje sadrže nukleotide. Osim toga iscrtan je graf, prikazan na 4.1, koji predočava kako se mijenjaju vremena potrebna za izgradnju FM-indeksa, za izgradnju wavelet stabla i za izračunavanje BWT transformacije u ovisnosti o veličini ulazne datoteke.

Tablica 4.2: Testiranje na DNK

veličina (MB)	vrijeme izgradnje (s)		
	BWT	wavelet stablo	FM indeks
1	0.4694	0.1431	0.6954
2	0.6427	0.1881	0.9581
5	1.4017	0.3137	1.9724
10	2.9828	0.5214	3.9738
20	6.7284	0.9268	8.5542
50	21.1193	3.5455	27.0914
100	46.4439	6.2703	57.6676
200	101.1712	11.9061	122.9448
500	284.38207	29.0901	338.5938



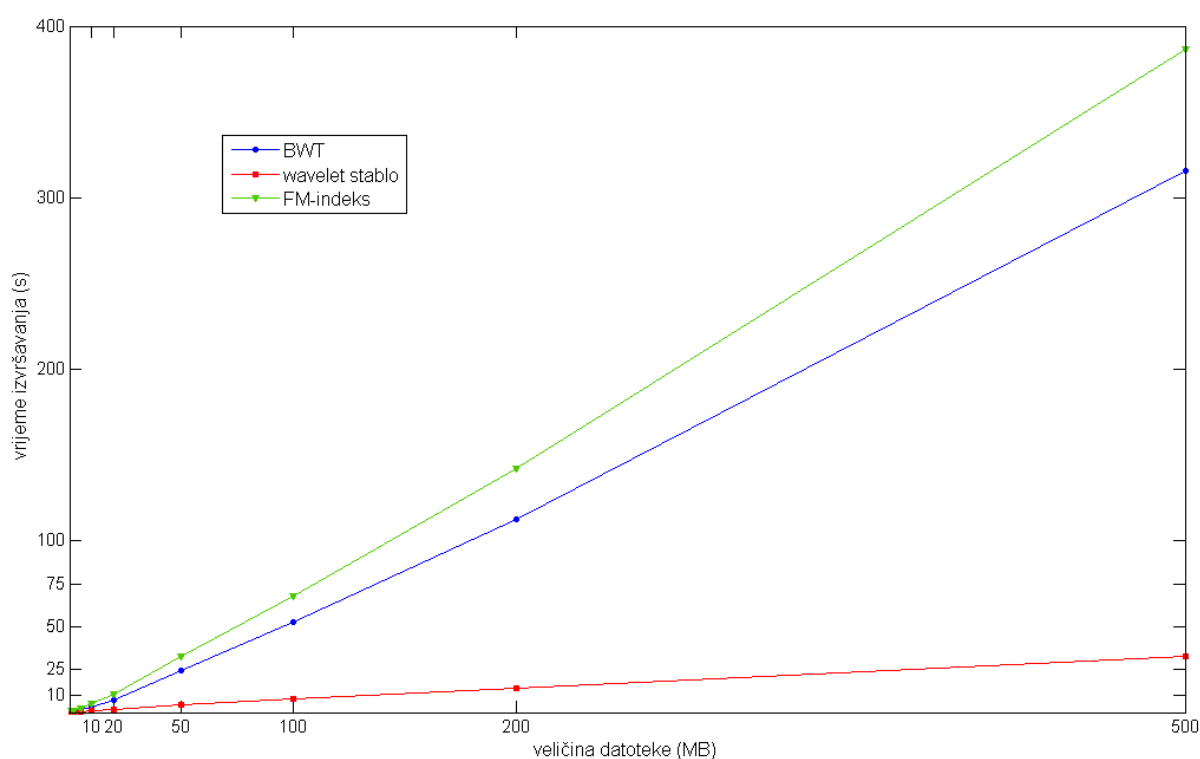
Slika 4.1: Vremenska ovisnost trajanja algoritama o veličini datoteke koja sadrži nukleotide

Ukoliko malo proučimo podatke iz tablice i grafova, možemo vidjeti kako je složenost izgradnje indeksa linearno ovisna o veličini ulaznog niza. To je svojstvo koje smo i željeli postići, odnosno htjeli smo da nam je složenost izgradnje indeksa linearna. Iz priložene tablice i grafa vidimo kako je proces izgradnje FM-indeksa za veće ulazne datoteke veoma zahtjevan i dugotrajan proces, koji može potrajati i do nekoliko minuta. Upravo iz tog razloga javlja se potreba za pronalaskom što efikasnijih algoritama koji bi se mogli koristiti u postupku izgradnje FM-indeksa.

Pogledajmo sada vrijeme potrebno za izgradnju FM-indeksa nad proteinskim nizovima. U tablici 4.3 prikazani su rezultati koji su dobiveni za datoteke koje su sadržavale proteinske nizove. Slika 4.2 prikazuje ovisnost izgradnje pojedinih dijelova FM-indeksa, kao i samog FM-indeksa u ovisnosti o veličini ulaznog niza, odnosno ulazne datoteke.

Tablica 4.3: Testiranje na proteinima

veličina (MB)	vrijeme izgradnje (s)		
	BWT	wavelet stablo	FM indeks
1	0.4547	0.366	0.9295
2	0.6815	0.4119	1.2776
5	1.5011	0.5886	2.5025
10	3.2401	0.8957	4.9289
20	7.392	1.4896	10.4347
50	24.1723	4.61334	32.6025
100	52.6474	7.7732	67.9625
200	112.8949	13.9333	142.1054
500	315.5526	32.8476	386.6631

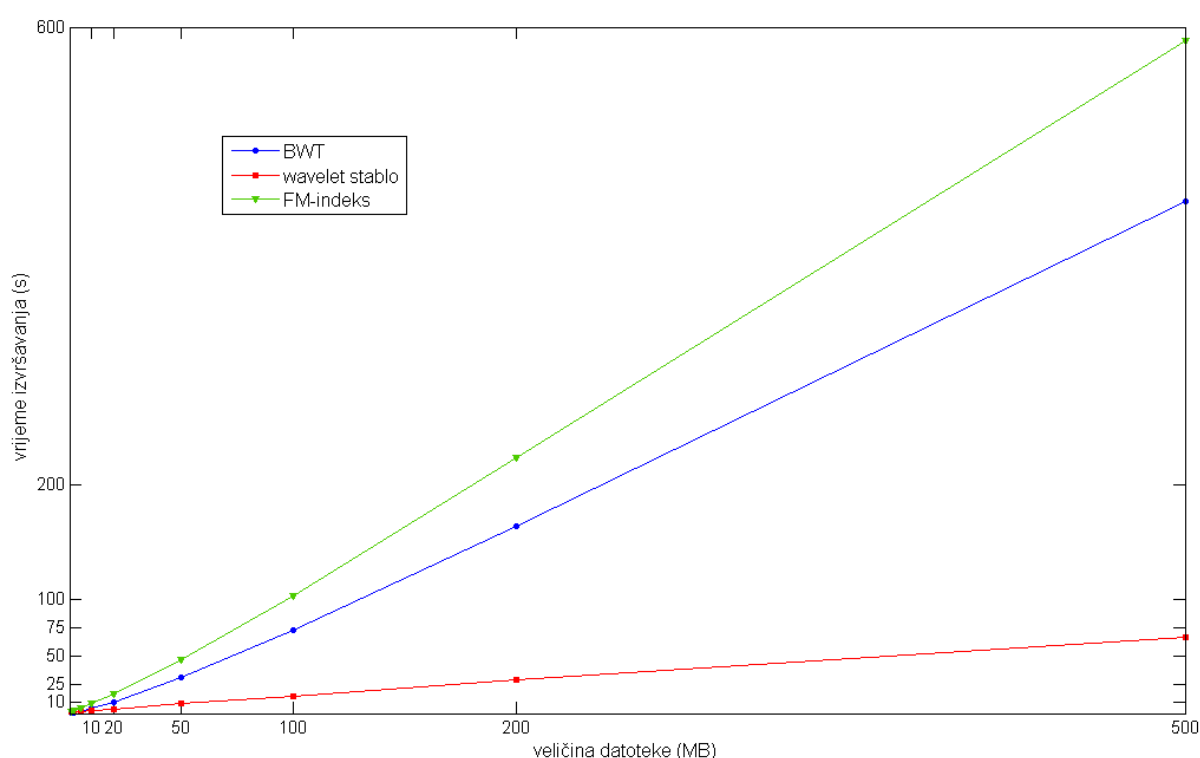


Slika 4.2: Vremenska ovisnost trajanja algoritama o veličini datoteke koja sadrži proteine

Konačno, u tablici 4.4 je prikazana i ovisnost izgradnje FM-indeksa o veličini ulaznog niza za treću vrstu datoteka. Ovisnost izgradnje FM-indeksa, provođenja BWT transformacije, kao i izgradnju wavelet stabla, u odnosu na veličinu ulaznog niza prikazana je na slici 4.3.

Tablica 4.4: Testiranje na ASCII znakovima

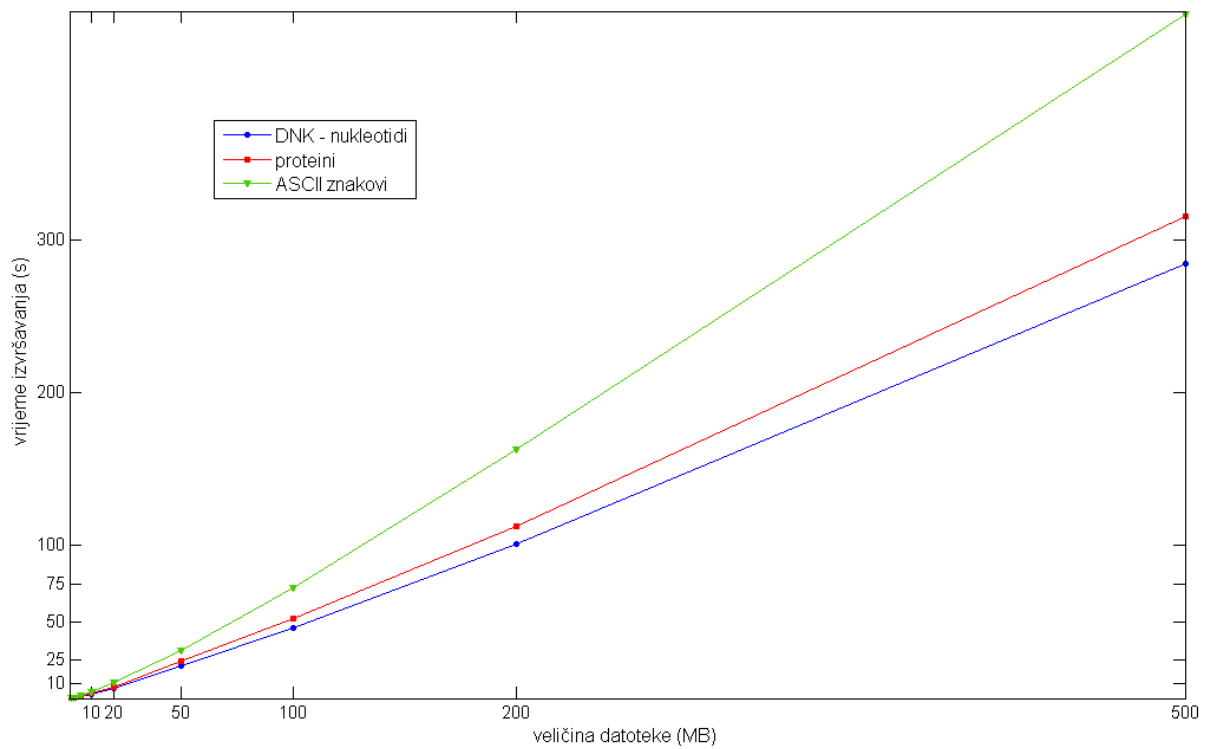
veličina (MB)	vrijeme izgradnje (s)		
	BWT	wavelet stablo	FM indeks
1	0.4795	1.2554	1.9021
2	0.772	1.3382	2.4216
5	1.8199	1.789	4.34387
10	4.2868	2.579	8.3161
20	10.2298	4.061	17.1314
50	31.4606	8.3703	46.9771
100	72.435	15.4384	103.1664
200	163.362	29.5337	223.3576
500	447.88	66.3437	588.4351



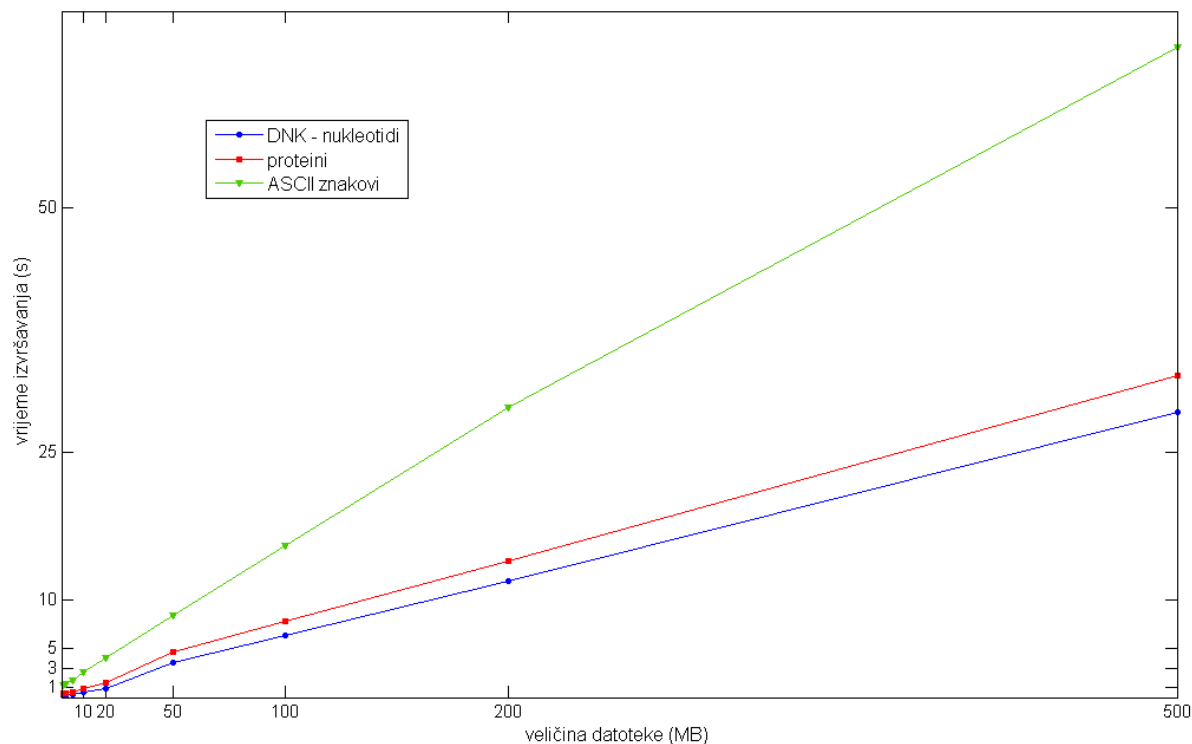
Slika 4.3: Vremenska ovisnost trajanja algoritama o veličini datoteke koja sadrži ASCII znakove

Razmotrimo još za kraj ovog poglavlja vremenske ovisnosti izgradnje pojedinih struktura u ovisnosti o veličini ulazne datoteke, ali i o vrsti ulazne datoteke. Slike 4.4, 4.5, 4.6 prikazuju te ovisnosti za BWT transformaciju, izgradnju wavelet stabla i izgradnju FM-indeksa respektivno. Možemo vidjeti kako su za sve tri vrste datoteka te ovisnosti linearne za izgradnju svake od struktura (kao što je i prethodno bilo spomenuto). Također možemo vidjeti kako trajanje izgradnje indeksa ovisi o vrsti datoteke nad kojom se indeks izgrađuje. Iz slika se vidi kako se trajanje izgradnje indeksa za nizove nukleotida i proteina ne razlikuje jako puno, dok je vrijeme potrebno za izgradnju indeksa nad datotekama s ASCII znakovima značajno veće. Ukoliko se vratimo na tablicu 4.1 možemo vidjeti kako se ti nizovi razlikuju po veličini njihovih abeceda. Odatle možemo zaključiti kako

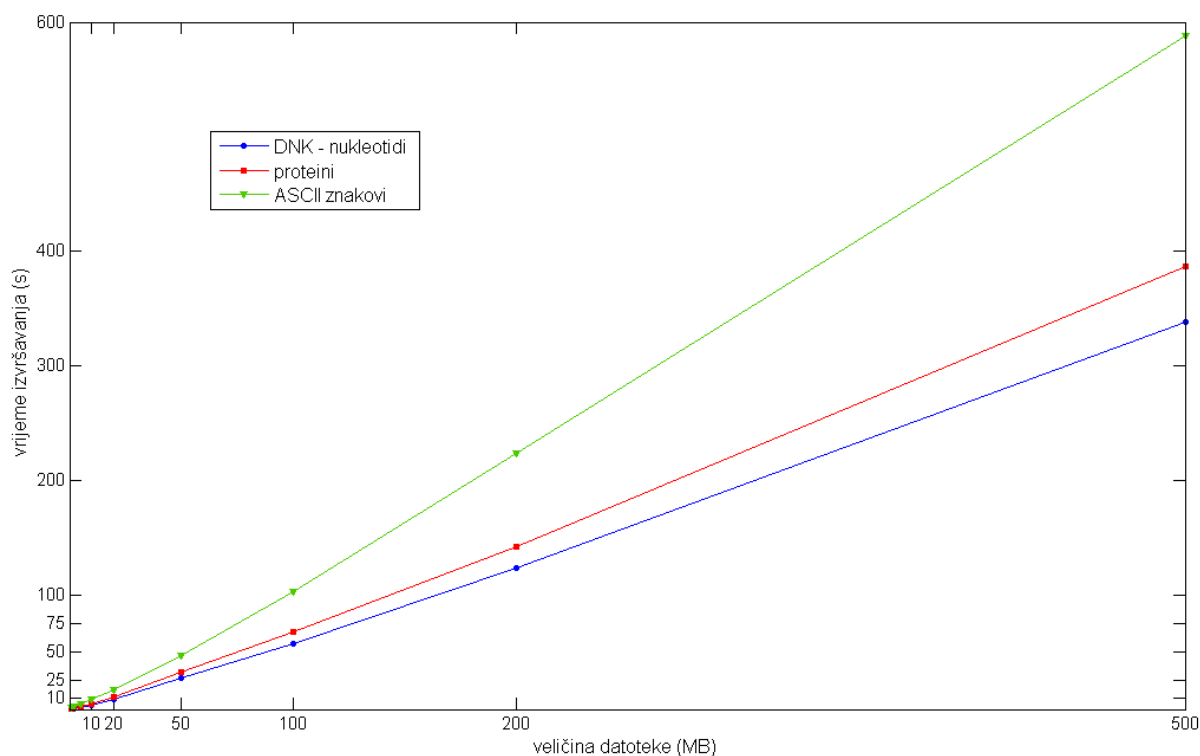
trajanje izgradnje FM-indeksa uvelike ovisi i o veličini abecede nad kojom su ti nizovi izgrađeni, odnosno da će trajanje izgradnje biti veće što je veličina abecede ulaznog niza veća.



Slika 4.4: Vremenska ovisnost trajanja BWT algoritma o veličini datoteke određenog tipa



Slika 4.5: Vremenska ovisnost trajanja stvaranja wavelet stabla o veličini datoteke određenog tipa



Slika 4.6: Vremenska ovisnost trajanja stvaranja FM-indeksa o veličini datoteke određenog tipa

4.2 Vrijeme obavljanja *count* upita

Drugi važan parametar pomoću kojeg se može ocijeniti uspješnost implementacije FM-indeksa jest obavljanje *count* upita nad izgrađenim nizom. Dakle, sada kada je izgrađen FM-indeks želimo vidjeti koliko brzo se može izračunati pojavljivanje nekog podniza u nizu nad kojim je indeks izgrađen. Osim toga, želimo vidjeti i kako veličina niza nad kojim je izgrađen indeks utječe na trajanje prebrojavanja. Krenimo dakle redom. Tablice 4.5, 4.6 i 4.7 prikazuju trajanje prebrojavanja pojavljivanja nekog podniza u indeksu. Iz tablica se mogu izvesti neki veoma važni zaključci. Kao prvo, vidi se da trajanje prebrojavanja raste otprilike linearno s porastom duljine uzorka koji se traži. Nadalje, kroz sve tablice vidi se kako vrijeme prebrojavanja nekog uzorka ne ovisi o veličini ulaznog niza nad kojim je indeks izgrađen (u tablici 4.5 se može uočiti jedna mala anomalija, odnosno skok u trajanju prebrojavanja između datoteke od 20MB i 50MB, no to je iz razloga što je uočeno kako u datoteci od 20MB i manje postoji samo 6 različitih znakova, dok u onima od 50MB naviše postoji 16 različitih znakova). Konačno, uspoređujući tablice možemo vidjeti kako trajanje obavljanja prebrojavanja ovisi o veličini abecede nad kojom je ulazni niz izgeneriran. To je iz razloga što se za obavljanje prebrojavanja koristi wavelet stablo čija veličina ovisi upravo o veličini abecede.

Tablica 4.5: *count* upit nad DNK

veličina (MB)	trajanje prebrojavanja (ms)					
	duljina traženog niza					
	100	200	300	400	500	1000
1	2	3	5	7	8	17
2	2	4	5	7	7	17
5	2	3	5	7	7	16
10	2	3	5	7	8	16
20	1	4	5	7	8	16
50	2	5	7	8	10	22
100	2	4	7	8	10	22
200	2	4	7	9	9	22
500	3	4	7	7	10	23

Tablica 4.6: *count* upit nad proteinima

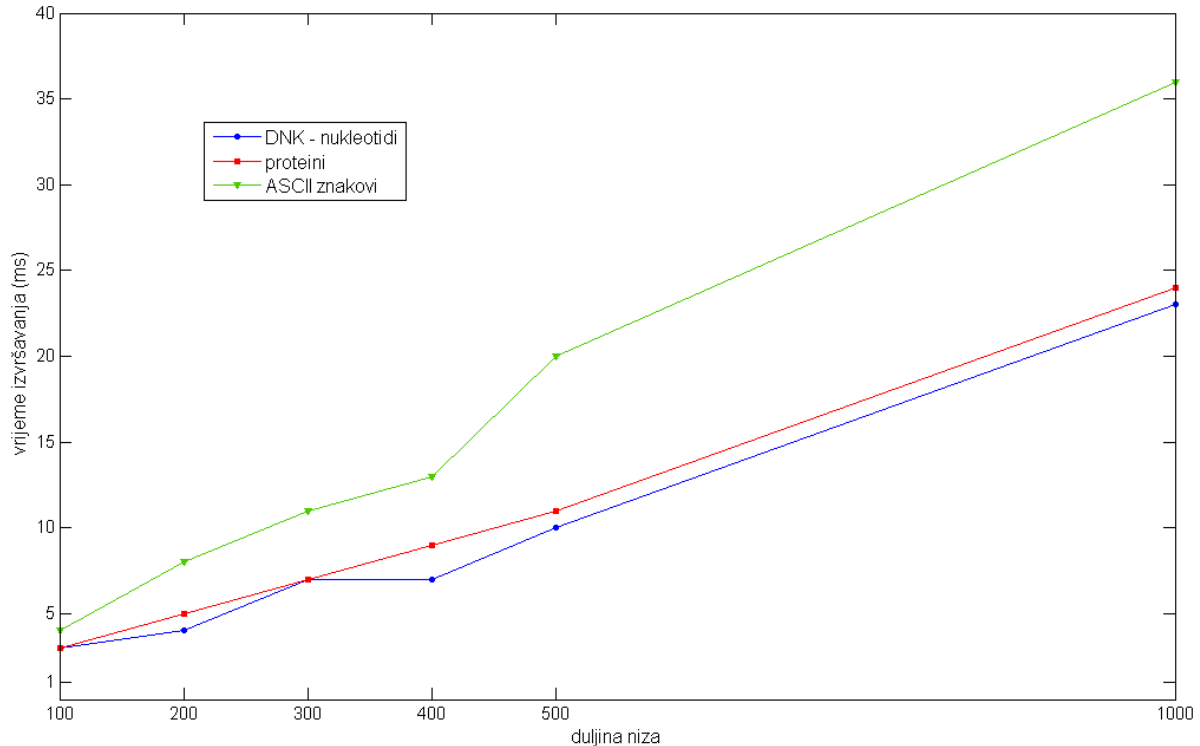
veličina (MB)	trajanje prebrojavanja (ms)					
	duljina traženog niza					
	100	200	300	400	500	1000
1	3	5	8	9	9	25
2	3	5	8	8	10	24
5	3	5	7	9	10	23
10	2	5	7	8	10	23
20	2	5	8	9	11	24
50	2	6	8	9	11	26
100	2	5	9	9	12	24
200	3	5	8	9	11	25
500	3	5	7	9	11	24

Tablica 4.7: *count* upit nad ASCII znakovima

veličina (MB)	trajanje prebrojavanja (ms)					
	duljina traženog niza					
	100	200	300	400	500	1000
1	3	8	11	13	18	34
2	4	7	11	12	18	33
5	4	7	10	12	19	34
10	4	8	10	13	19	35
20	4	8	10	12	19	35
50	4	8	11	13	19	35
100	4	8	11	14	20	36
200	4	8	11	13	19	38
500	4	8	11	13	20	36

Potrebno je napomenuti kako izrada mjerenja u milisekundama naravno nije najbolja praksa, no nažalost ovako su ipak dobiveni najuvjerljiviji rezultati, jer je uočeno da prilikom obavljanja više uzastopnih upita dolazi do anomalije da se vremena obavljanja upita znaju smanjiti. Uzrok tomu je vjerojatno što se određeni podaci spremaju u priručnu memoriju, pa se onda mogu puno brže dohvatiti prilikom ponovnog obavljanja prebrojavanja nekog niza.

Za kraj pogledajmo još sliku 4.7 na kojoj je prikazan graf gdje su isctane ovisnosti obavljanja prebrojavanja nekog uzorka u ovisnosti o duljini tog uzorka i u ovisnosti o veličini abecede niza nad kojim je izgrađen indeks (veličina datoteke nad kojom su se obavljali ovi eksperimenti bila je 500MB). Možemo vidjeti kako vrijeme prebrojavanja nekog uzorka raste linearno s veličinom uzorka kojeg se pretražuje (što je već ranije u tekstu utvrđeno). Također se na ovom grafu lijepo može vidjeti kako je obavljanje prebrojavanja također ovisno i o veličini abecede niza nad kojim je indeks izgrađen.



Slika 4.7: Vremenska ovisnost izvršavanja *count* algoritma o veličini niza koji se prebrojava

4.3 Memorijsko zauzeće FM-indeksa

Posljednji parametar kojeg možemo promatrati tijekom izrade FM-indeksa jest memorijsko zauzeće programa. U ovom poglavlju istaknut ćemo koliko je memorijsko zauzeće tokom izgradnje indeksa, kao i na kraju kada je indeks u potpunosti izgrađen. Osim toga, u ovom poglavlju pokazat će se kako Java virtualni stroj zauzima mnogo više memorije nego što je zaista potrebno za samo izvođenje algoritma, što predstavlja jedan veliki nedostatak. Ovdje će se razmotriti memorijska zauzeća za sve tipove datoteka, no promatrat će se samo veličine datoteka od 50MB nadalje. U tablicama 4.8, 4.9 i 4.10 je prikazana potrošnja memorije FM-indeksa za sve tri vrste datoteka i različite veličine datoteka. Izdvojena su tri stupca u tablicama, od kojih prvi stupac predstavlja maksimalnu memoriju koju je cijeli proces zauzeo. Drugi stupac predstavlja maksimalnu količinu memorije koju je algoritam u nekom trenutku koristio, dok treći stupac predstavlja stvarno zauzeće memorije (dakle količinu memorije koju stvarno zauzimaju izgrađene strukture) od strane izgrađenog indeksa. Iz tablica se vidi kako je maksimalna memorija koja je zauzeta od strane Java virtualnog stroja puno veća nego li je za rad algoritma potrebno. To je ogroman nedostatak, jer ispada da su memorijski zahtjevi mnogo veći nego što oni uistinu jesu.

Slike 4.8, 4.9 i 4.10 prikazuju grafičke prikaze prethodno navedenih tablica radi boljeg pregleda. Iz tablica i slika se može vidjeti kako prosječno memorijsko zauzeće izgradnje FM-indeksa za sve inačice iznosi otprilike $6n$ (ovdje se misli na memoriju koju algoritam

Tablica 4.8: Memorijsko zauzeće - DNK

veličina (MB)	memorijsko zauzeće (MB)		
	maksimalno	pravo	konačno
50	393	314	53
100	803	620	103
200	1553	1230	201
500	3830	3080	494

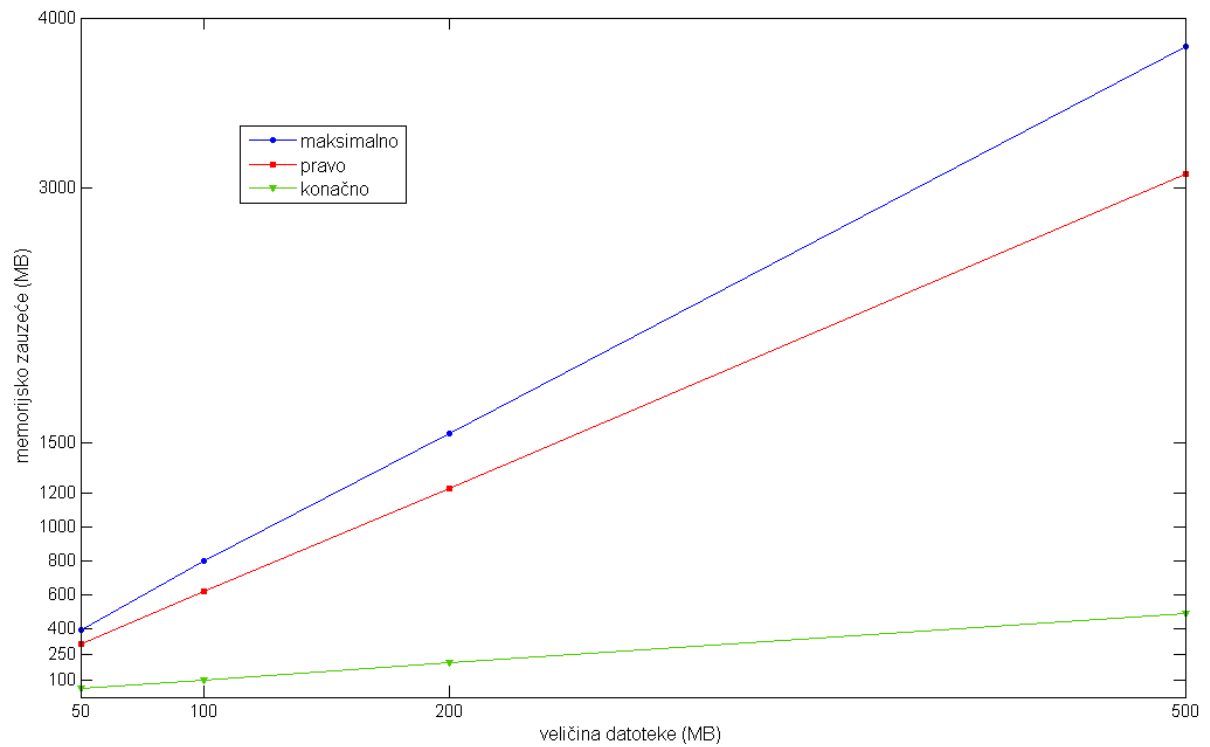
Tablica 4.9: Memorijsko zauzeće - proteini

veličina (MB)	memorijsko zauzeće (MB)		
	maksimalno	pravo	konačno
50	370	292	67
100	800	614	130
200	1550	1230	254
500	4800	3070	680

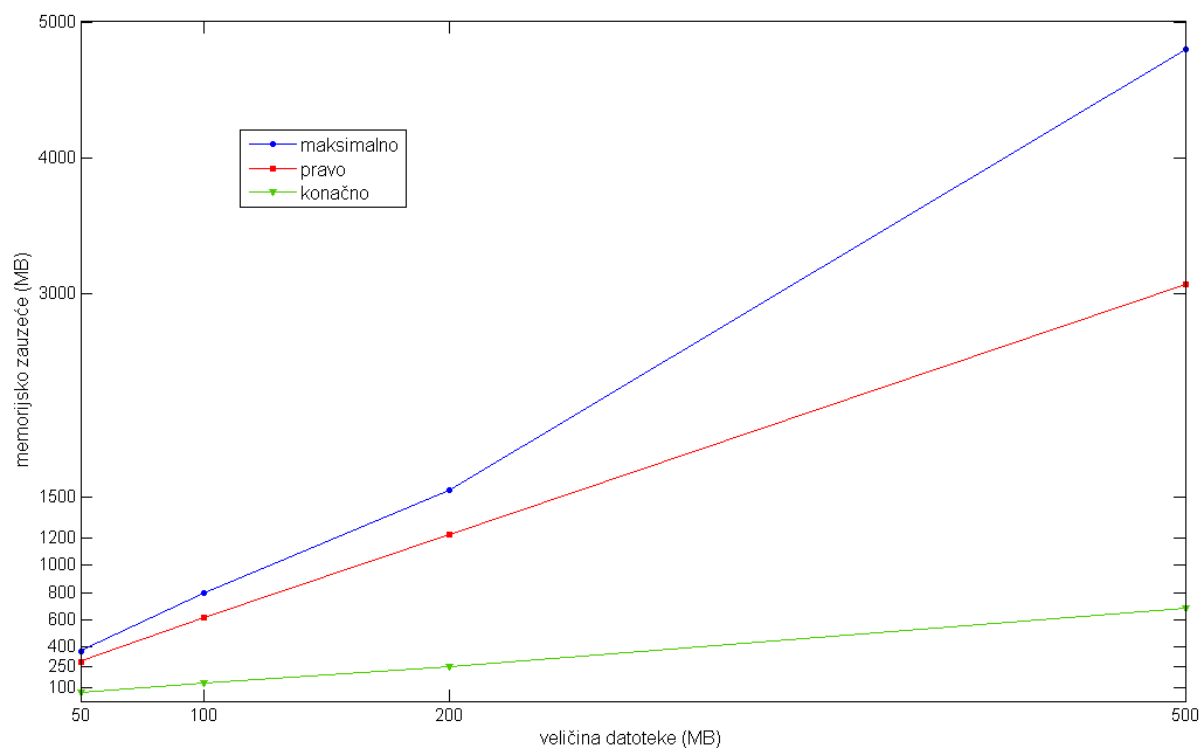
stvarno koristi) pri čemu je n veličina ulaznog niza. U tablici 4.10 se može vidjeti kako je u 2 slučaja bilo potrebno i nešto više memorije od toga. Razlog tome nije direktno poznat, no vjerojatno se radi o većem broju rekurzivnih poziva prilikom konstrukcije sufiksnog polje, jer se u tom postupku koriste dodatne strukture koje mogu povećati memorijsku potrošnju. Osim toga, te su datoteke izgenerirane u potpunosti nasumično, bez ikakvih pravilnosti i nad većim abecedama. Vidimo da se kod ostalih datoteka algoritam ponaša očekivano. Nakon izgradnje vidimo da FM-indeks zauzima mnogo manje memorija nego je bilo potrebno za izgradnju. Koliko točno memorije će zauzimati će opet ovisiti o tome kolika je veličina abecede nad kojom je indeks izgrađen. Vidimo da za nukleotidne nizove FM-indeks zauzima otprilike istu količinu memorije kao i originalni niz. Kako se veličina abecede povećava, tako se i povećava količina memorije koju FM-indeks zauzima nakon izgradnje.

Tablica 4.10: Memorijsko zauzeće - ASCII znakovi

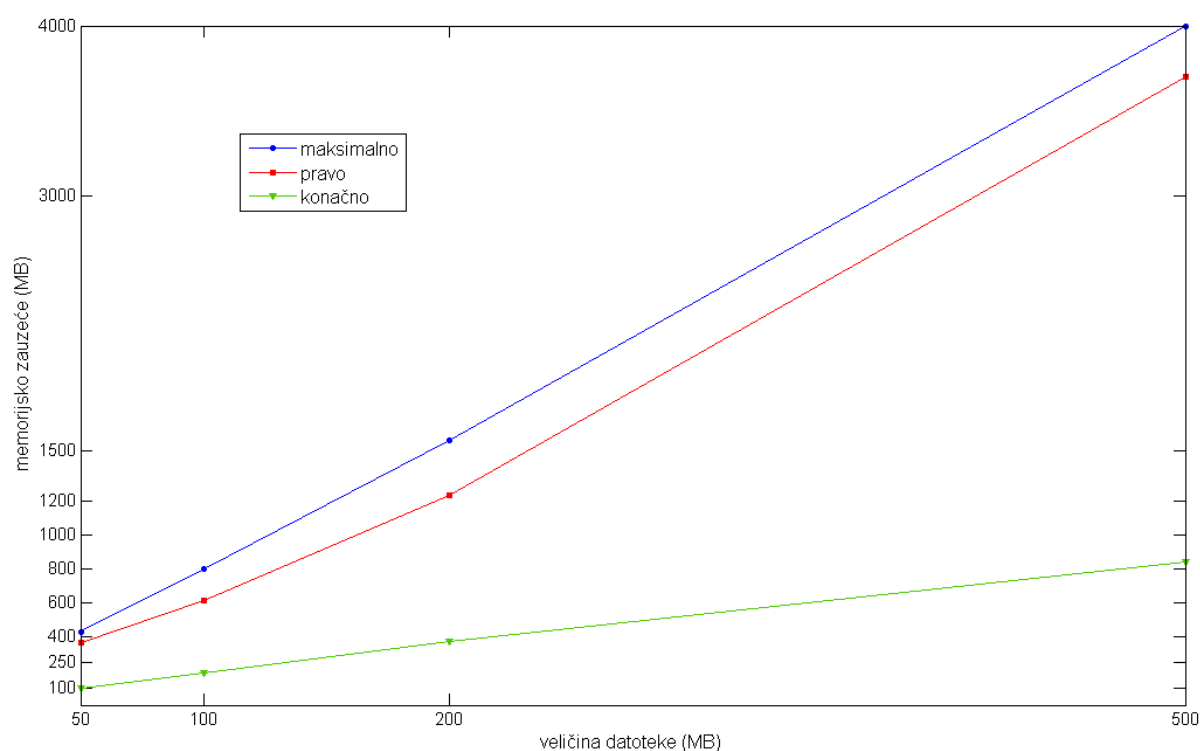
veličina (MB)	memorijsko zauzeće (MB)		
	maksimalno	pravo	konačno
50	432	365	100
100	800	619	191
200	1558	1235	378
500	4000	3700	840



Slika 4.8: Memorijska ovisnost stvaranja FM-indeksa algoritama o veličini datoteke koja sadrži nukleotide

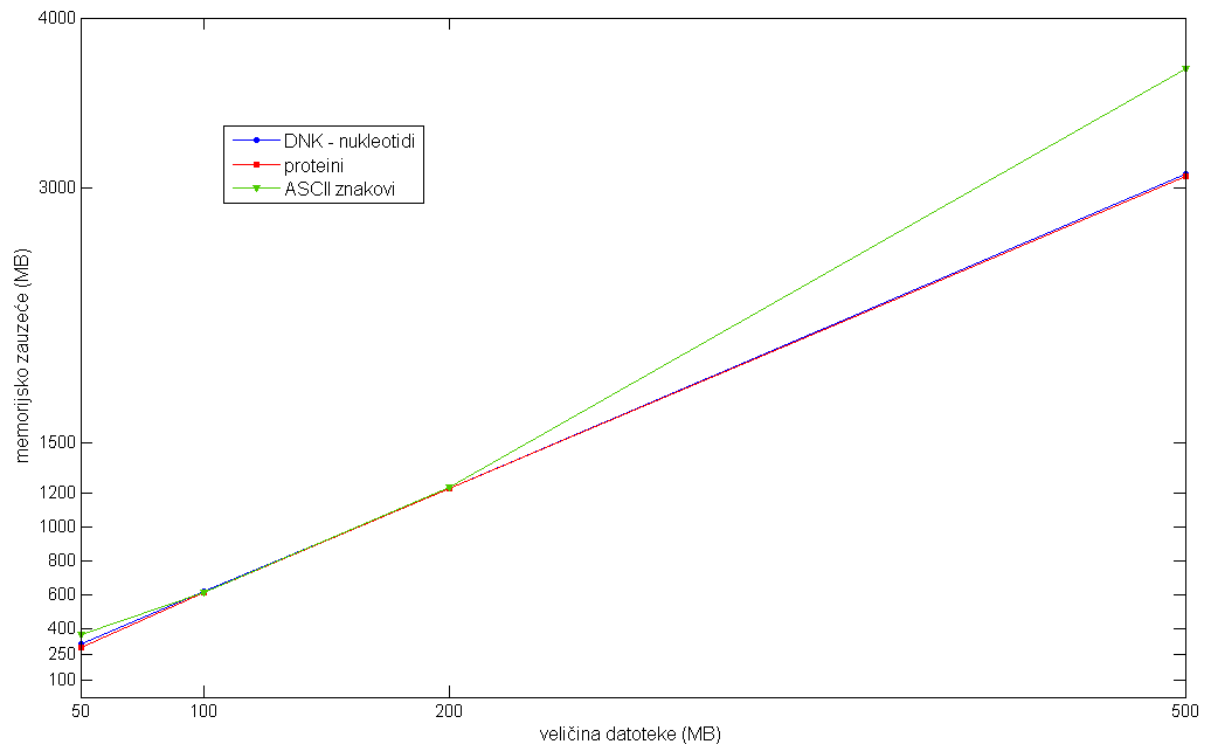


Slika 4.9: Memorijska ovisnost stvaranja FM-indeksa algoritama o veličini datoteke koja sadrži proteine



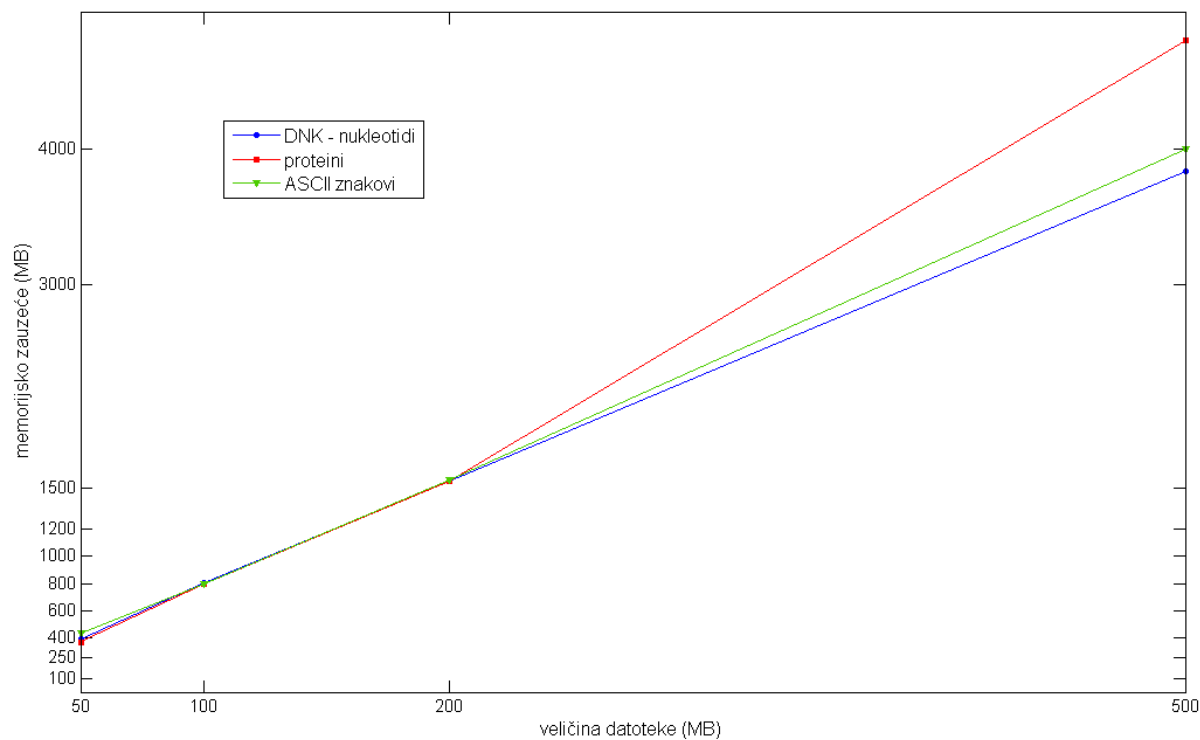
Slika 4.10: Memorijska ovisnost stvaranja FM-indeksa algoritama o veličini datoteke koja sadrži ASCII znakove

Slika 4.12 prikazuje maksimalnu memorijsku potrošnju algoritma. Ovo je zapravo veličina koja nas najviše zanima. Vidimo kako je memorijska potrošnja za sve datoteke u istim veličinama gotovo podjednaka (osim onih dviju datoteka koje ponešto odstupaju). Vidimo kako se algoritam ponaša veoma poželjno i kako otprilike znamo koliku memorijsku potrošnju možemo očekivati s obzirom na ulazni niz.



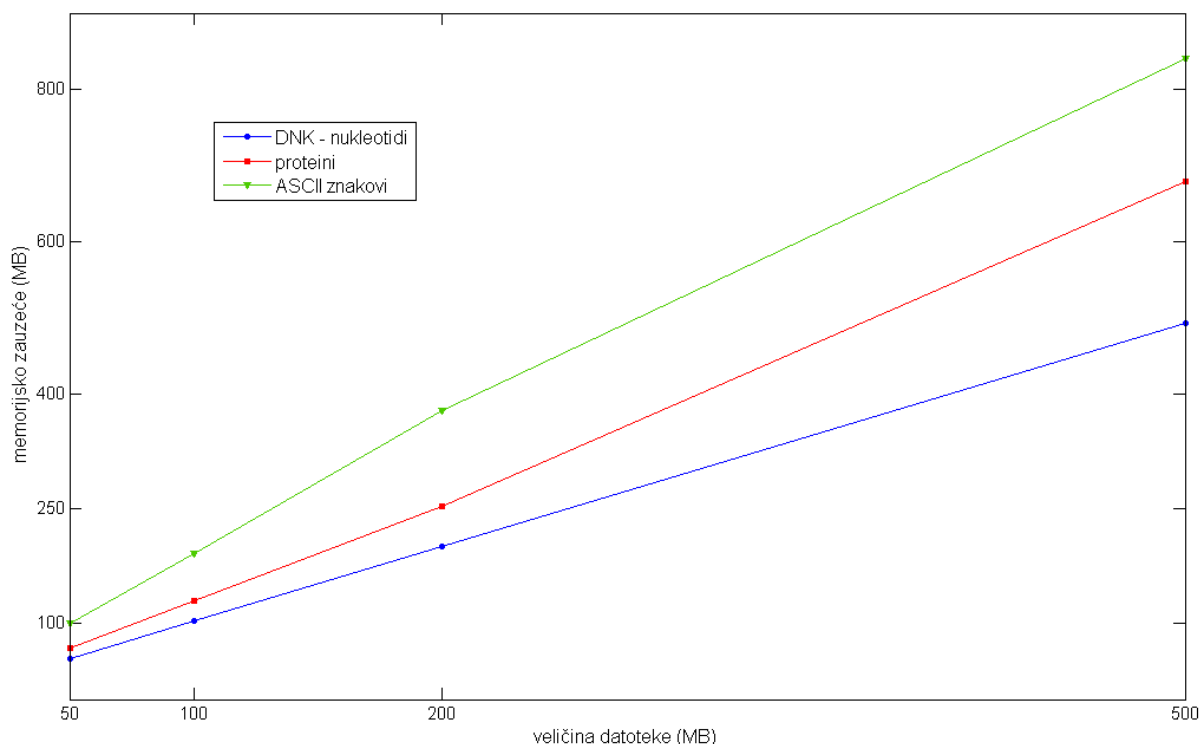
Slika 4.11: Memorijska (maksimalna) ovisnost stvaranja FM-indeksa o veličini datoteke određenog tipa

Slika 4.11 prikazuje maksimalnu memoriju koju je program zauzeo tijekom izgradnje indeksa za svaki tip datoteke po različitim veličinama. Ovaj graf se neće posebno komentirati jer on zapravo ni ne predstavlja pravu memorijsku potrošnju algoritma.



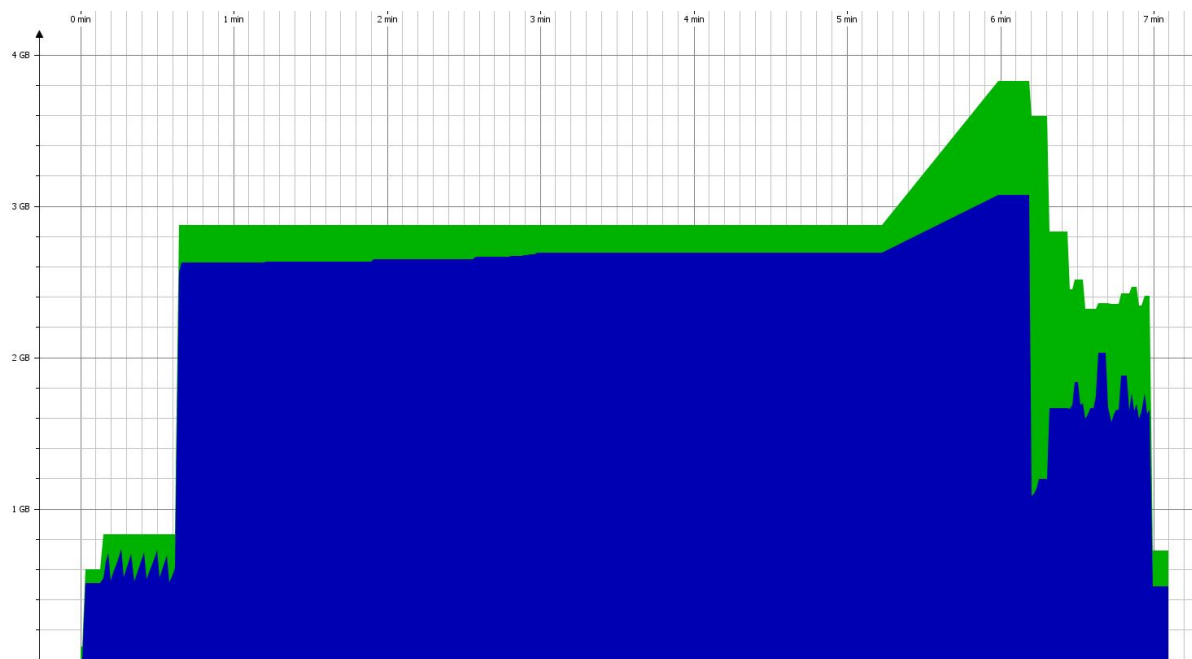
Slika 4.12: Memorijska (realno maksimalna) ovisnost stvaranja FM-indeksa o veličini datoteke određenog tipa

Slika 4.13 prikazuje memorijska zauzeća FM-indeksa nakon što je izgrađen. Vidi se kako to zauzeće ovisi o ulaznom nizu, ali ne samo to, ono ovisi i o veličini abecede ulaznog niza. Što je abeceda manja to će i zauzeće indeksa biti manje.



Slika 4.13: Memorijska (konačna) ovisnost stvaranja FM-indeksa o veličini datoteke određenog tipa

Pogledajmo sada sliku 4.14, preuzetu iz alata Jprofiler, koje predstavlja memorijsko zauzeće algoritma kroz vrijeme. Plavom bojom označena je memorija koju program, odnosno algoritam stvarno koristi za spremanje struktura, dok je zelenom bojom označena slobodna memorija koju je Java virtualni stroj dodatno zauzeo. Vidimo da cijelo vrijeme Java virtualni stroj zauzima mnogo više memorije nego što mu doista treba za rad. To je ogroman nedostatak pri izradi memorijski zahtjevnih algoritama kao što je ovaj, jer Java programski jezik ne omogućuje efikasno rukovanje memorijom. Stoga je prilikom korištenja ovog programa potrebno imati na umu kako će realno zauzeće memorije zapravo biti veće od onog što algoritam zaista koristi.



Slika 4.14: Zauzeće memorije Java virtualnog stroja tijekom konstrukcije indeksa

Zaključak

Iako je izgradnja FM-indeksa za veće ulazne datoteke zahtjevan i dugotrajan proces, prednost takvog pristupa je kasnije efikasno pretraživanje i dohvaćanje željenih tekstualnih uzoraka. Implementacijom opisanom u ovome radu, postignuta je izgradnja indeksa s linearnom složenošću, koja ovisi samo o veličini ulazne datoteke. Pokazalo se da je opisanom implementacijom postignuto to da vrijeme prebrojavanja nekog uzorka ne ovisi o veličini ulaznog niza nad kojim je indeks izgrađen, već o veličini abecede ulaznog niza. Što se tiče memorijske složenosti, prilikom izgradnje indeksa, ona iznosi otprilike $6n$, pri čemu je n veličina ulaznog niza. Nakon izgradnje indeksa, zauzeta memorija se smanjuje, te tada ovisi o veličini abecede ulaznog niza. Unatoč nedostacima programskog jezika Java u kojem je napravljena implementacija, vremenska i memorijska složenost ostvarenog algoritma je zadovoljavajuća.

Literatura

- [1] Michael Burrows; David J. Wheeler. A block sorting lossless data compression algorithm. *Technical Report 124, Digital Equipment Corporation*, 1994.
- [2] Jon L. Bentley; Robert Sedgewick. Fast algorithms for sorting and searching strings. *SODA '97 Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, 1997.
- [3] Ge Nong; Sen Zhang; Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. *Data Compression Conference, 2009. DCC '09*, 2009.
- [4] Ge Nong; Sen Zhang; Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *Computers, IEEE Transactions on*, 2011.
- [5] Pang Ko; Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 2003.
- [6] R. Grossi; A. Gupta; J. Vitter. High-order entropy-compressed text indexes. *In Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850, 2003.
- [7] R. Raman; V. Raman; S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *SODA*, pages 233–242, 2002.
- [8] Alex Bowe. RRR – a succinct rank/select index for bit vectors. <http://alexbowe.com/rrr>, November 2013.
- [9] Francisco Claude; Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. *In Proc. 15th SPIRE, LNCS 5280*, pages 176–187, 2008.
- [10] Jochen Singer. A wavelet tree based FM-index for biological sequences in SeqAn. *Freie Universitat Berlin - Bioinformatik, Master Thesis*, 2012.
- [11] Paolo Ferragina; Giovanni Manzini. Opportunistic data structures with applications. *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000.
- [12] Paolo Ferragina; Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, Volume 52, Issue 4, July 2005, pages 552–581, 2005.
- [13] Paolo Ferragina; Giovanni Manzini. Pizza&Chili Corpus, Compressed indexes and their testbeds. <http://pizzachili.dcc.uchile.cl/texts.html>, December 2013.