SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2325

Implementacija komprimirane podatkovne strukture za pretraživanje teksta temeljene na FMindeksu

Martin Šošić

Zagreb, srpanj 2012.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da biste uklonili ovu stranicu obrišite naredbu `\izvornik`.

Hvala mom mentoru Mili Šikiću na trudu, strpljenju i razumijevanju.

SADRŽAJ

POPIS SLIKA

POPIS TABLICA

1. Uvod

Brzim razvojem informacijske tehnologije došlo je do eksponencijalnog porasta podataka kojima raspolažemo [1]. Kako bismo takve podatke mogli obraditi i iskoristiti, potrebni su novi i brži načini iskorištavanja podataka. Jedan od aktualnih problema je pretraživanje teksta gdje se osim brzine traži i minimalno zauzeće prostora. Dva osnovna pristupa tom problemu su *indeksi pretraživanja po cijelom tekstu* (engl. *Full-text index*) i *indeksi pretraživanja po riječima* (engl. *Word-based index*).

Indeksi pretraživanja po cijelom tekstu su podatkovne strukture koje omogućavaju brzo i potpuno pretraživanje teksta. Problem koji takvi indeksi rješavaju možemo preciznije opisati na sljedeći način: Nad zadanim tekstom T želimo izgraditi podatkovnu strukturu koja podržava učinkovito pronalaženje svih pojavljivanja nekog proizvoljnog teksta P unutar teksta T. Neke od uobičajenih operacija koje takvi indeksi nude su lociranje i prijava broja pojavljivanja teksta P, dok napredniji indeksi nude i operacije kao što je uređivanje teksta T.

Ozbiljan problem takvih indeksa je dugo vremena bilo njihovo veliko prostorno zauzeće [2]. Pošto korisnost indeksa najviše dolazi do izražaja upravo kod velikih tekstova, gdje omogućuje znatno brže pretraživanje nego uobičajene metode, veliko zauzeće prostora predstavlja ozbiljno ograničenje. Zbog tog razloga noviji indeksi se oslanjaju na mogućnost sažimanja teksta čime se postiže zauzeće prostora koje više nije funkcija veličine teksta T već njegove veličine nakon sažimanja. Mi ćemo posebnu pažnju posvetiti radu Ferragine i Manzini-a [3], čiji je indeks trenutno jedan on najpopularnijih.

U poglavlju 2 uvest ćemo pojmove i oznake koje ćemo koristiti kroz ostatak rada. U poglavlju 3 ćemo sažeto objasniti LZ78 algoritam te opisati prvi i drugi indeks iz [3]. Prvi indeks nećemo objasniti detaljno već ćemo se dotaći samo onih tema koje su nam bitne za razumijevanje rada drugog indeksa. Drugi indeks ćemo objasniti u detalje i ujedno ponuditi rješenja za određene probleme koji u [3] nisu razrađeni. U poglavlju 4 detaljno ćemo opisati implementaciju indeksa iz sekcije 3.3. Opisat ćemo probleme na koje smo naišli i kako smo ih riješili. Za razliku od [3], dodatnu ćemo

pažnju posvetiti postupku izgradnje indeksa. U poglavlju 5 opisat ćemo kako smo primijenili indeks za ubrzanje postupka sastavljanja genoma. U poglavlju 6 prikazat ćemo rezultate izgradnje indeksa nad različitim vrstama tekstova i rezultate primjene indeksa u sastavljanju genoma te diskutirati o dobivenom. U poglavlju 7 osvrnut ćemo se na rad i predložiti buduća unaprijeđenja.

2. Notacija

U ovom poglavlju uvest ćemo notaciju koju ćemo koristiti kroz ostatak rada.
Tekst nad kojim gradimo indeks i koji želimo pretraživati označavat ćemo kao *T*. *T* je tekst duljine *n* koji se sastoji od znakova iz abecede Σ.
Oznakom *T*[*i*] ćemo predstaviti *i*-ti znak iz teksta *T*, gdje prvi znak ima indeks 1.
Oznakom *T*[*i*, *j*] ćemo predstaviti podniz teksta *T* koji počinje *i*-tim znakom a završava *j*-tim znakom. Analogno tome cijeli tekst *T* možemo zapisati i kao *T*[1, *n*]. U tekstu ćemo često koristiti *T*[*i*, *n*] kako bismo predstavili sufiks od *T* duljine $n - i + 1$ i *T*[1, *i*] kako bismo predstavili prefiks od *T* duljine i.
Oznakom T^R označavat ćemo obrnuti tekst. Npr. ako je *T* = `abcd` tada je T^R = `dcba`.
Oznakom |*A*| ćemo označavati broj elemenata skupa *A*, dok ćemo oznakom |*w*| označavati duljinu teksta *w*.

Tekst koji tražimo unutar teksta *T* označavat ćemo slovom *P*. *P* je tekst duljine *p* te se također sastoji od znakova iz abecede Σ. Također ćemo reći da se *P* pojavljuje *occ* puta unutar *T*.

Podrazumijevat ćemo da svi logaritmi od sada pa nadalje imaju bazu 2.

Oznakom $H_k(T)$ označavat ćemo empirijsku entropiju *k*-tog reda za tekst *T* [3].

3. Metode

U [3] opisana su dva indeksa koji koriste svojstvo sažmivosti teksta te imaju prostorno zauzeće otprilike kao i sažeti tekst. Prvi indeks se temelji na pažljivoj kombinaciji Burrows-Wheeler transformacije i sufiksnih polja te omogućuje brzo pretraživanje uz minimalno zauzeće prostora. Taj indeks se često naziva i *FM indeks*. Drugi indeks se temelji na kombinaciji Burrows-Wheeler transformacije i LZ78 algoritma [4] te se nastavlja na i koristi prvi indeks. Razlikuje se od prvog indeksa po tome što omogućava veću brzinu pretraživanja. U ovom radu bavimo se implementacijom drugog indeksa. U ovom poglavlju ćemo uvesti neke osnovne pojmove, sažeto objasniti LZ78 algoritam, pojasnit ćemo prvi indeks, te ćemo zatim detaljnije opisati drugi indeks i ponuditi rješenje pojedinih problema koji nisu razrađeni u [3].

3.1. LZ78 algoritam

LZ78 algoritam [4] je kompresijski algoritam koji se temelji na rječniku. Osnovna ideja algoritma je ta da ako se neki podatak više puta ponavlja tada ga možemo zapisati u rječnik i za pamćenje idućih pojavljivanja tog podatka koristiti već zapamćene podatke. Kao rezultat algoritma dobivamo rječnik koji predstavlja podatke koje smo komprimirali.

Zamislimo da komprimiramo neki tekst K. Algoritam započinje s praznim rječnikom, te u svakoj iteraciji dodaje po jednu riječ u rječnik. U jednoj iteraciji algoritam uzima slova iz teksta K i iz njih gradi novu riječ. Prvo slovo uzima od tamo gdje je stao u prošloj iteraciji, a uzima slova dok god ne izgradi riječ koja ne postoji u rječniku. Tada tu riječ, nazovimo ju R, dodaje u rječnik kao par (i, c), gdje je i indeks zapisa u rječniku koji predstavlja riječ R bez zadnjeg slova, a c je zadnje slovo iz riječi R. Ako se riječ sastoji od samo jednog slova onda ju pamtimo kao $(0, c)$. Postupak se ponavlja dok se ne potroše svi znakovi iz teksta K.

Rezultat komprimiranja je rječnik D. Također možemo reći da smo tekst K parsirali u $K_1K_2 \ldots K_n$ gdje su riječi $K_1, K_2, \ldots, K_n$ riječi koje predstavljaju zapisi u rječniku D.

Objašnjeni postupak ćemo pojasniti na primjeru. Uzmimo da komprimiramo tekst *K* = `aabaab`. Počinjemo sa praznim rječnikom *D* = {}. U prvoj iteraciji uzimamo prvi znak iz teksta *K*, pa imamo riječ `a`. U rječniku ne postoji riječ `a` pa u dodajemo u rječnik. Sada imamo *D* = {(0,`a`)}. U idućoj iteraciji uzimamo drugi znak pa imamo riječ `a`. U rječniku postoji riječ `a` pa uzimamo iduće slovo, `b`. U rječniku ne postoji riječ `ab` pa ju dodajemo u rječnik. Sada imamo *D* = {(0,`a`), (1,`b`)}. U idućoj iteraciji uzimamo slovo `a`. U rječniku postoji riječ `a` pa uzimamo iduće slovo, `a`. U rječniku ne postoji riječ `aa` pa ju dodajemo u rječnik. Sada imamo *D* = {(0,`a`), (1,`b`), (1,`a`)}. U idućoj iteraciji uzimamo slovo `b`. U rječniku ne postoji riječ `b` pa ju dodajemo u rječnik. Sada imamo *D* = {(0,`a`), (1,`b`), (1,`a`), (0,`b`)}. Iz rječnika *D* sada možemo lako rekonstruirati tekst *K*. Kažemo da smo *K* parsirali u `a ab aa b`.

3.2. Prvi indeks

Drugi indeks se nastavlja na i koristi prvi indeks. Zbog toga ćemo prvo objasniti neke osnovne koncepte prvog indeksa koji su nam potrebni za razumijevanje drugog indeksa.

Prvi indeks nudi dvije različite operacije: brojanje pojavljivanja teksta *P* i lociranje teksta *P*. Vremenska složenost lociranja je $O(p + occlog^{1+\epsilon}n)$ gdje je $0 < \epsilon < 1$ proizvoljna konstanta odabrana pri izgradnji indeksa, dok je vremenska složenost brojanja pojavljivanja $O(p)$. Lociranje se nastavlja na brojanje pojavljivanja i njega nećemo objašnjavati jer nam ono nije potrebno za izgradnju drugog indeksa, dapače, svrha drugog indeksa jest da ponudi brže lociranje od onog koje nudi prvi indeks. Ono što će nama biti potrebno je prva operacija, tj. odgovor na pitanje "Koliko se puta *P* pojavljuje u *T*?". Prvi indeks se gradi na temelju teksta *T* te mu nakon toga *T* više nije potreban. Pri izgradnji indeksa koristi se Burrows-Wheeler transformacija [5] (nadalje BWT) kako bi se tekst doveo u oblik pogodniji za sažimanje, a zatim se sažima BW_RLX algoritmom [3].

3.2.1. Konceptualna matrica M_T

Za nas bitan pojam je konceptualna matrica M_T koja se koristi u BWT kao centralni pojam. Bitno je primjetiti da se ta matrica nikada zbilja ne stvara već ostaje samo na razini koncepta. Matrica M_T se gradi na sljedeći način:

1. Na kraj teksta *T* se dodaje posebni znak # manji od svih znakova iz abecede Σ.

2. Redovi matrice se dobiju cikličkim rotiranjem teksta *T*#.

3. Redovi matrice se leksikografski sortiraju.

#mississippi
i#mississipp
ippi#mississ
issippi#miss
ississippi#m
mississippi#
pi#mississip
ppi#mississi
sippi#missis
sissippi#mis
ssippi#missi
ssissippi#mi

Slika 3.1: Primjer konceptualne matrice M_T za tekst *T* = mississippi.

Možemo primijetiti da smo sortiranjem redaka matrice M_T ustvari sortirali sufikse teksta *T*. Bitno svojstvo M_T je da su svi retci koji započinju nekim prefiksom *P* uzastopni.

3.2.2. Podatkovne strukture i algoritmi

Navesti ćemo strukture podataka i algoritme prvog indeksa koji nude funkcionalnost potrebnu drugom indeksu, a zatim ćemo navesti još i neke dodatne algoritme kojima ćemo proširiti prvi indeks isključivo za potrebe drugog indeksa.

Opp(*T*)

Stukture podataka koje nastaju izgradnjom prvog indeksa, a koriste se za brojanje pojavljivanja odsada ćemo nazivati zajedničkim imenom Opp(*T*) te ćemo ih promatrati kao jedinstvenu strukturu podataka. Opp(*T*) je temeljni građevni element drugog indeksa. Prostorno zauzeće strukture Opp(*T*) je ograničeno s $5nH_k(T) + O(n\frac{log(logn)}{logn})$ bitova, za bilo koji $k \geq 0$ [3].

findRows(backward_search)

Algoritam koji izvodi operaciju brojanja pojavljivanja koristeći strukturu Opp(T) originalno se zove `backward_search`. U daljnjem tekstu taj ćemo algoritam radi boljeg razumijevanja zvati `findRows`. Algoritam `findRows` prima kao ulaz tekst P, a vraća brojeve redaka konceptualne matrice M_T koji započinju tekstom P. Dovoljno je da vrati brojeve samo prvog i zadnjeg retka zbog svojstva matrice M_T da su retci sa zajedničkim prefiksom uzastopni. Na primjer, ako uzmemo da je T = `mississippi` i da je P = `i`, `findRows` će vratiti dvojku (2, 5) (pogledati 3.1).

Algoritam `findRows` interno radi tako da prvo nađe brojeve redaka za $P[p]$, zatim te podatke koristi kako bi našao brojeve redaka za $P[p\text{-}1, p]$, i tako sve dok ne nađe brojeve redaka za $P[1, p]$. Svaka od tih operacija je vremenske složenosti $O(1)$, a pošto tu operaciju radimo p puta (jednom za svaki sufiks teksta P), ukupna vremenska složenost algoritma `findRows` iznosi $O(p)$.

findRowsForSuffixes

Ako nas zanima za svaki sufiks teksta P koji retci u M_T počinju s njim, možemo to izračunati tako da p puta izvedemo algoritam `findRows`, za svaki sufiks jednom. Ukupna vremenska složenost takvog postupka iznosila bi $O(p^2)$. Ako se prisjetimo da algoritam `findRows` ionako mora izračunati brojeve redaka za sve sufikse teksta P da bi došao do rješenja za P, zaključujemo da pamćenjem međurezultata algoritma `findRows` možemo u složenosti $O(p)$ dobiti brojeve redaka za svaki sufiks teksta P. Zato uvodimo novi algoritam koji nazivamo `findRowsForSuffixes`. On prima tekst P te vraća brojeve redaka za sve sufikse od P u složenosti $O(p)$.

findRowsForSuffixesWithPrefix

Postavljamo sljedeći problem: imamo neki tekst P i zanimaju nas brojevi redaka za sve njegove sufikse, ali prefiksirane sa nekim zadanim znakom \$. Npr. ako je P = abcb i prefiks = \$, tada nas zanimaju brojevi redaka M_T koji započinju sa \$b, \$cb, \$bcb i \$abcb. Da bismo efikasno riješili taj problem modificirat ćemo algoritam `findRowsForSuffixes` na sljedeći način: prije nego što na temelju brojeva redaka za $P[i, p]$ izračunamo brojeve redaka za $P[i\text{-}1, p]$ ubacujemo korak u kojem računamo brojeve redaka za $\$P[i, p]$ i pamtimo taj međurezultat. Vremenska složenost ubačenog koraka je $O(1)$ pa vremenska složenost cijelog algoritma i dalje ostaje $O(p)$. Tako modificirani algoritam nazvat ćemo `findRowsForSuffixesWithPrefix`.

On prima tekst *P* te vraća brojeve redaka za sve sufikse od *P* prefiksirane zadanim znakom \$ u vremenskoj složenosti $O(p)$.

findRowsDoStep

Algoritam `findRowsDoStep` radi na nižoj razini nego ostali ovdje nabrojani algoritmi te se iz njega mogu izvesti svi ostali algoritmi. `findRowsDoStep` prima raspon redaka iz konceptualne matrice i jedan znak, zatim na temelju tog znaka i redaka računa novi raspon redaka matrice. Složenost algoritma je $O(1)$.
Tipična primjena bila bi kada znamo retke matrice koji odgovaraju nekom tekstu *A* a zanimaju nas retci matrice koji odgovaraju tekstu *cA* gdje je *c* neki znak iz abecede Σ.

3.3. Drugi indeks

Detaljno ćemo objasniti drugi indeks iz [3]. Opisat ćemo podatkovne strukture i algoritme te objasniti kako smo odlučili izvesti pojedine korake.

Drugi indeks je indeks pretraživanja cijelog teksta koji locira sva pojavljivanja teksta *P* unutar teksta *T* u vremenu $O(p + occ)$ pri tome koristeći $O(nH_k(T)log^{\epsilon}n) + O(n/(log^{1-\epsilon}n))$ bitova, gdje je $0 < \epsilon < 1$ konstanta proizvoljno odabrana pri izgradnji indeksa. Drugi indeks koristi prvi indeks, a ključna ideja je iskorištavanje pravilnosti teksta ne samo kako bi se smanjilo prostorno zauzeće već i kako bi se ubrzalo lociranje.

3.3.1. LZ78 parsiranje teksta *T*

Nakon što primjenimo LZ78 algoritam opisan u 3.1 na tekst *T* dobit ćemo rječnik *D* = $\{T_1, T_2, \ldots, T_d\}$ i parsirani tekst $T = T_1T_2 \ldots T_d$. Postoji iznimka pri kojoj je *D* = $\{T_1, T_2, \ldots, T_{d-1}\}$ ako se dogodi da je T_d (zadnja riječ iz *T*) jednaka nekoj od riječi iz rječnika *D*. Zbog te iznimke morat ćemo biti dodatno pažljivi te ćemo je još kasnije spominjati. Bitno svojstvo rječnika *D* je *prefiks-potpunost*, što znači da za svaku riječ T_i iz *D* vrijedi da je svaki njen neprazni prefiks također riječ iz *D*.

Reći ćemo da je \$ neki znak koji ne pripada abecedi Σ te ćemo uvesti novi tekst:

$$T_{\$} = T_1\$T_2\$ \ldots \$T_d\$.$$

Znakovi \$ u $T_{\$}$ predstavljaju točke sidrišta (engl. *anchor points*). Za te znakove ćemo zapamtiti njihove pozicije u tekstu *T* (točnije, za \$ koji slijedi iza riječi T_i pamtimo vrijednost $1 + |T_1| + \ldots + |T_i|$). Na taj način će nam da bismo odredili poziciju teksta *P* biti dovoljno odrediti njegovu poziciju s obzirom na neki \$.

Pojavljivanja teksta P u tekstu T možemo podijeliti na dva glavna slučaja:

1. Pojavljivanja u cijelosti sadržana unutar jedne riječi T_i.

2. Pojavljivanja koja se prostiru kroz dvije ili više riječi $T_iT_{i+1}\ldots T_{i+h}$, h ≥ 0.

Kako bismo riješili ta dva slučaja koristimo dva različita algoritma o kojima ćemo detaljno govoriti u sljedećim sekcijama.

3.3.2. Lociranje unutarnjih pojavljivanja

Opisat ćemo algoritam koji locira sva pojavljivanja teksta P unutar teksta T gdje je P sadržan u samo jednoj riječi T_i. Takva pojavljivanja zvati ćemo *unutarnja* pojavljivanja.

Tekst $T_\$$ ćemo predstaviti pomoću sufiks stabla gdje će svaki brid biti označen jednim slovom iz Σ. Zbog prefiks-potpunosti rječnika D svaki će čvor iz sufiks stabla predstavljati jednu riječ iz rječnika D, a možemo tu riječ dobiti tako da prođemo kroz stablo od korijena do čvora predstavnika. U čvorovima stabla pamtit ćemo poziciju prvog slova pripadne riječi T_i u tekstu T.
Ovdje je bitno prisjetiti se da se može dogoditi da je zadnja riječ T_d iz $T_\$$ jednaka nekoj drugoj riječi T_j. Tada više ne vrijedi tvrdnja da svaki čvor u stablu predstavlja jednu riječ već se pojavljuje potreba da jedan od čvorova predstavlja dvije riječi. To smo odlučili riješiti tako da za riječ T_d u stablo dodamo poseban čvor koji je dijete čvora koji predstavlja riječ T_j. Kako bismo mogli taj poseban čvor razlikovati od ostalih čvorova označiti ćemo brid koji povezuje njega i njegovog oca posebnim znakom & koji nije u abecedi Σ. Važno je primjetiti da iako je čvor riječi T_d izveden kao dijete čvora riječi T_j on konceptualno nije njegovo dijete već ćemo ga kasnije gledati kao njegovog susjeda.

Osnovna strategija lociranja unutarnjih pojavljivanja započinje nalaženjem svih riječi T_i kojima je P sufiks. Zbog prefiks-potpunosti rječnika D primjećujemo sljedeće: ako neka riječ T_j u sebi sadrži P tada sigurno postoji riječ T_i koja je prefiks od T_j a kojoj je p sufiks. To znači da je dovoljno obići podstabla čvorova koji predstavljaju riječi kojima je P sufiks kako bismo pronašli sva pojavljivanja.

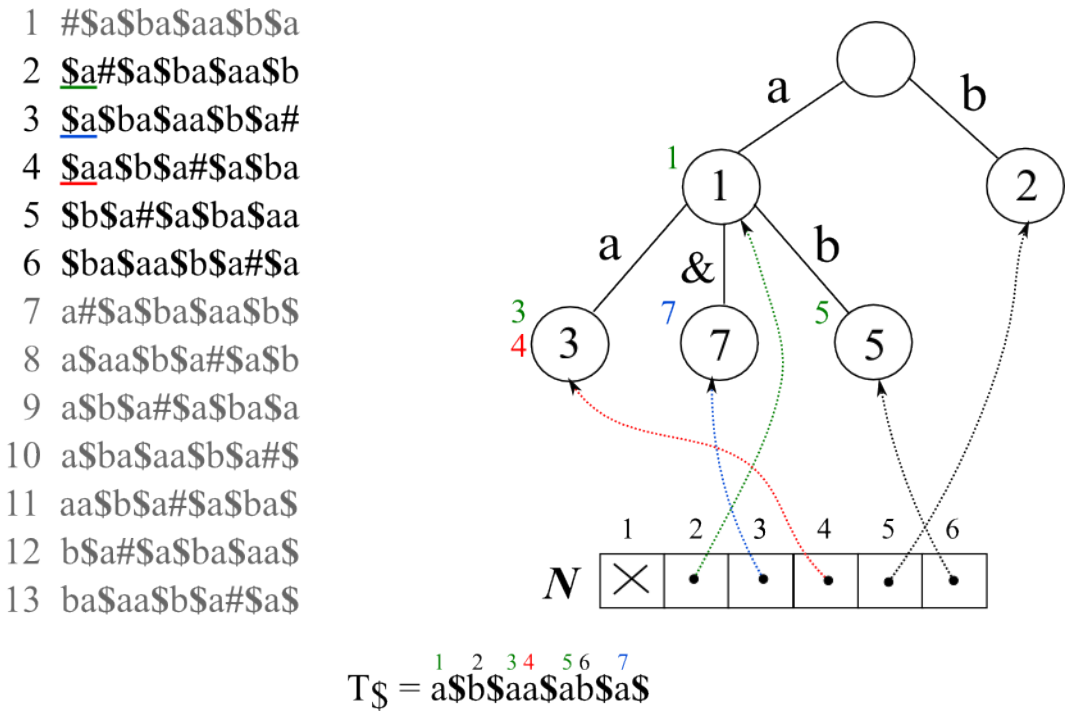
Problem koji nam preostaje rješiti je kako efikasno naći čvorove u stablu koji predstavljaju riječi T_i kojima je P sufiks. U tu svrhu ćemo se poslužiti konceptualnom matricom $M_{T^R_\$}$. Svaka riječ T_i ima svoj odgovarajući redak u matrici, i taj redak počinje sa $\$T_i^R\$$ odnosno $\$T_i^R\#$ za i = 1. Da bismo našli riječi T_i kojima je P sufiks trebamo naći sve retke matrice kojima je $\$P^R$ prefiks. Traženje takvih redaka ćemo ostvariti

strukturom Opp($T^R_\$$) i algoritmom `findRows`.
Nakon što nađemo takve retke preostaje nam odrediti čvorove u stablu koji predstavljaju riječi određene nađenim retcima. Zbog toga ćemo unaprijed u nekom nizu N za svaki čvor zapamtiti koji redak matrice mu odgovara. Na taj način možemo redak preslikati u čvor u vremenskoj složenosti $O(1)$.
Kada obilazimo čvorove u podstablu za svaki čvor koji odgovara riječi T_j ćemo vratiti vrijednost v_j + ($|T_{i_k}|$ - p) gdje je v_j pozicija prvog znaka riječi T_j u T a T_{i_k} je riječ koja odgovara korijenu podstabla. Ta je vrijednost pozicija unutarnjeg pojavljivanja.

Objašnjeni postupak ćemo obuhvatiti algoritmom `get_internal` koji prima tekst P i vraća sve pozicije unutarnjih pojavljivanja tog teksta. Vremenska složenost algoritma `get_internal` će biti $O(p+occ)$ (p zbog složenosti algoritma `findRows` a occ zbog složenosti obilaska podstabala).

Primjer

Na primjeru ćemo pokazati prethodno opisani postupak. Uzeti ćemo da je T = abaaaba. Tada je $T_\$$ = a$b$aaaba$ te je $T^R_\$$ = $a$baaab$a.



Slika 3.2: Primjer sufiks stabla za T = abaaaba i P = a.

Na slici 3.3.2 su prikazani konceptualna matrica $M_{T^R_\$}$ i sufiks stablo opisano u postupku. Također je prikazan niz N koji na poziciji i sadrži pokazivač prema čvoru stabla koji odgovara i-tom retku matrice. N sadrži pokazivače samo za one retke matrice koji

imaju odgovarajući čvor u stablu, a to su oni retci koji počinju znakom \$. Njih smo na slici označili tamnijom bojom. Primjetimo da je zadnja riječ u $T_\$$, T_5 = a, ista kao i prva riječ T_1 = a. Zato smo ju u sufiks stablo dodali kao dijete od čvora koji predstavlja T_1 i brid označili znakom &. Čvorovi stabla sadrže početne pozicije u T od riječi koje predstavljaju.

Uzet ćemo u ovom primjeru da je P = a. Prvi korak nam je odrediti retke matrice koji su prefiksirani sa $\$P^R$. Pošto je $\$P^R$ = \$a, upotrijebit ćemo `findRows`(\$a) i kao rezultat dobiti (2, 4). Sada kada znamo retke upotrijebit ćemo niz N i pogledati na koje čvorove pokazuju redovi matrice 2, 3 i 4. Sada kada smo odredili te čvorove, obilazimo njihova podstabla i za svaki čvor izračunamo vrijednost v_j + ($|T_{i_k}|$ - p). U ovom slučaju moramo biti pažljivi zbog čvora koji odgovara riječi T_5. Iako je taj čvor u podstablu čvora koji odgovara riječi T_1, nećemo vratiti njegovu vrijednost pri obilasku tog podstabla jer kao što smo već prije spomenuli on konceptualno nije dijete čvora pod kojim se nalazi već je njegov susjed. Tako u ovom primjeru obilazimo tri podstabla i na kraju dobivamo vrijednosti 1, 3, 4, 5 i 7.

3.3.3. Lociranje preklapajućih pojavljivanja

Opisati ćemo algoritam koji locira sva pojavljivanja teksta P unutar teksta T gdje se P prostire kroz barem dvije ili više riječi $T_jT_{j+1}\ldots T_{j+h}$, h $\geq$ 0. Takva pojavljivanja zvati ćemo *preklapajuća* pojavljivanja. Prvo ćemo objasniti sporiju ali jednostavniju verziju algoritma kao uvod, a zatim ćemo objasniti nešto kompliciraniji algoritam koji radi u vremenskoj složenosti $O(p + occ)$.

Jednostavni algoritam

Tekst P počinje u riječi T_{j-1} te se prostire kroz riječi $T_j \ldots T_{j+h}$ za neki h > 0 ako i samo ako postoji $m, 1 \leq m < p$, takav da je P[1, m] sufiks od T_{j-1} te da je P[m+1, p] prefiks od $T_j \ldots T_d$. U skladu sa navedenim od sada pa nadalje ćemo govoriti da se tekst P *prelama* na poziciji m. Koristimo riječ *prelamati* jer možemo zamisliti da je P prelomljen na više dijelova znakovima \$.
Ovdje je također bitno primijetiti da iako se tekst P može prostirati kroz više od dvije riječi nama će uvijek biti zanimljiva samo granica između prve i druge riječi tj. prvi znak \$. Na taj način izbjegavamo višestruko lociranje istog pojavljivanja.

Osnovna ideja algoritma je sljedeća. Za svaki m, $1 \leq m < p$ pretpostavimo da se P prelama na toj poziciji te se pitamo da li u $T_\$$ postoji takav \$ da se odmah s njegove lijeve strane nalazi P[1, m] a odmah s desne strane P[m+1, p](uz iznimku da P[m+1,

p] smije biti izlomljen znakovima \$). Ako postoji jedan ili više takvih \$ onda je naša pretpostavka bila točna i *P* se zbilja pojavljuje u $T_\$$ prelomljen na poziciji *m*. Pošto znamo pozicije znakova \$ (rekli smo prije da ih pamtimo) onda je trivijalno doznati početne pozicije pojavljivanja *P* u *T*.

Kako bismo navedenu ideju efikasno ostvarili koristit ćemo strukturu Opp te pripadajuće algoritme `findRowsForSuffixes` i `findRowsForSuffixesWithPrefix`. Nak strukturom Opp($T_\$^R$) pozvat ćemo `findRowsForSuffixesWithPrefix`($P^R$, \$) te tako za svaki *m* doznati koji su to retci konceptualne matrice $M_{T_\$^R}$ prefiksirani sa $\$P[1,m]^R$. Ti retci odgovaraju LZ78-riječima koje imaju *P*[1, *m*] kao sufiks. Reći ćemo da ti retci, tj. njihovi brojevi čine skup X_m.

Na sličan način ćemo nad strukturom Opp(*T*) pozvati `findRowsForSuffixes`(*P*) te tako za svaki *m* doznati koji su to retci konceptualne matrice M_T prefiksirani sa *P*[*m*+1, *p*]. Svaki taj redak je ustvari sufiks od *T*, pa smo tako našli sve sufikse of *T* kojima je *P*[*m*+1, *p*] prefiks. Reći ćemo da ti retci, tj. njihovi brojevi čine skup Y_m.

Sada svakom *m* možemo pridružiti par (X_m, Y_m) koji nam govori koje riječi T_i smiju doći lijevo od \$ (to nam govori X_m) i koji sufiksi od *T* smiju doći desno od \$ (to nam govori Y_m). Npr. ako smo našli 3 riječi koje smiju biti lijevo i 2 sufiksa od *T* koji smiju biti desno tada postoji ukupno 6 kombinacija za koje ćemo se pitati da li zbilja u $T_\$$ odnosno *T* postoje znakovi \$ okruženi tim kombinacijama.

Kako bismo mogli brzo doznati odgovor na to pitanje, unaprijed ćemo pripremiti te podatke. Za *i* = 1,..., d, naći ćemo redak matrice $M_{T_\$^R}$ prefiksiran sa $\$T_{i-1}^R\$$ (uvijek će biti točno jedan takav redak), nazovimo broj tog retka x_i. Na sličan način ćemo naći redak u matrici M_T prefiksiran sa $T_iT_{i+1}\ldots T_d\#$ (uvijek će biti točno jedan takav redak), nazovimo broj tog retka y_i. Tako smo za svaki *i* pronašli par (x_i, y_i) za koji možemo reći da opisuje tekst koji okružuje *i*-ti \$. Definiramo skup točaka $Q = \{(x_1, y_1), (x_2, y_2), \ldots, (x_d, y_d)\}$.

Sada prethodni problem možemo postaviti na sljedeći način: za svaki par (X_m, Y_m) želimo naći sve točke (x_i, y_i) iz *Q* takve da je x_i u X_m i da je y_i u Y_m. Kako bismo brzo obavili tu pretragu koristit ćemo geometrijsku podatkovnu strukturu *RT*(*Q*) opisanu u [6] koja podržava ortogonalnu pretragu raspona (engl. *orthogonal range query*) u vremenskoj složenosti $O(log(log|Q|) + q)$ gdje je *q* broj točaka nađenih pretragom. Svaka točka (x_j, y_j) koju tako nađemo predstavlja jedno pojavljivanje *P* u *T*. Jedino što preostaje je doznati točnu poziciju, a to računamo izrazom v_j - *m* gdje je v_j početna pozicija riječi T_j u *T*.

Algoritam koji objedinjuje gore opisani postupak nazivamo `get_overlapping`. Vremenska složenost tog algoritma je $O(plog(logn) + occ_0)$.

Lociranje preklapajućih pojavljivanja u vremenskoj složenosti $O(p + occ_0)$

Razlikujemo dvije vrste teksta s obzirom na duljinu:

- ako je duljina teksta $\leq$ log(log n) kažemo da je tekst *kratak*.
- ako je duljina teksta > log(log n) kažemo da je tekst *dugačak*.

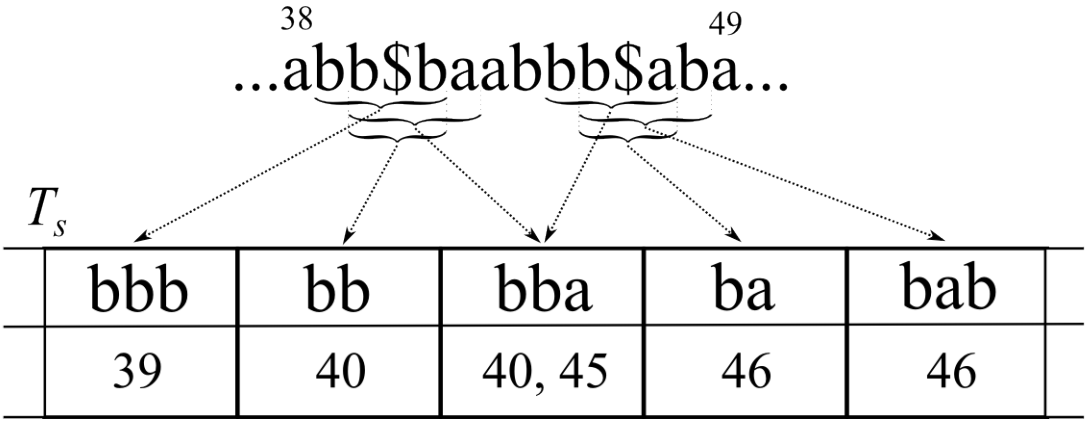
S obzirom na duljinu teksta *P* definirati ćemo dva različita algoritma lociranja, jedan za dugački *P* a drugi za kratki *P*. Oba algoritma imati će vremensku složenost $O(p+occ_0)$.

Algoritam za kratki *P* Najbitnije je primijetiti da preklapajućih pojavljivanja kratkog teksta *P* ne može biti puno. Naime, da bi pojavljivanje bilo preklapajuće mora počinjati lijevo od nekog $ i završavati desno od tog istog $. Pošto tekst *P* nije dulji od log(log n) početak njegovog preklapajućeg pojavljivanja ne može biti od $ koji ga lomi udaljen više od log(log n) pozicija. Isto vrijedi i za kraj. To znači da početak i kraj možemo odabrati na $O((log(logn))^2)$ različitih načina. Pošto u tekstu $T_\$$ ukupno ima *d* znakova $, a oko svakog od njih postoji $O((log(logn))^2)$ različitih kratkih tekstova, ukupno imamo $O(d(log(logn))^2)$ kratkih tekstova prelomljenih znakom $.
Ovdje je bitno napomenuti da dok oko svakog znaka $ gledamo koji kratki tekstovi postoje, nećemo gledati one kratke tekstove koji su desno od znaka $ prelomljeni sa još jednim ili više znakova $. Tako izbjegavamo višestruko uzimanje istog kratkog teksta.

Kao što smo pokazali, takvih preklapanja nema puno, pa ćemo za svako takvo pojavljivanje eksplicitno zapamtiti njegovu poziciju u tekstu *T*. Za to ćemo koristiti tablicu raspršenog adresiranja T_s kojoj je ključ kratki tekst a vrijednost je niz sa pozicijama tog teksta u *T*. Primjer izgradnje tablice prikazan je na slici 3.3. Kada nas zanimaju pozicije preklapajućih pojavljivanja teksta *P* jednostavno pogledamo u tablicu T_s pod ključ *P* i pročitamo pozicije iz dobivenog niza. Takvih pozicija ima *occ* pa je stoga vremenska složenost cijelog algoritma $O(p + occ)$.

Algoritam za dugački *P* Algoritam koji ćemo ovdje opisati konceptualno je vrlo sličan jednostavnom algoritmu iz 3.3.3. Glavna razlika je u tome da ćemo izbaciti faktor log(log n) iz vremenske složenosti, zbog čega ćemo morati pamtiti neke dodatne podatke.

Algoritam se temelji na ideji da više ne pokušavamo prelomiti tekst *P* na svakoj poziciji, već na svakoj (log(log n))-toj poziciji. Na taj način smo ubrzali algoritam za faktor log(log n) ali smo izgubili velik broj točnih rješenja. Kako bismo to popravili a ujedno i zadržali postignuto ubrzanje pamtit ćemo neke dodatne podatke vezane uz



Slika 3.3: Primjer izgradnje tablice T_s za kratke tekstove duljine manje od 4.

tekst *T*. Dok smo prije za svaki \$ iz $T_\$$ našli i zapamtili par (x_i, y_i), sada ćemo istu stvar učiniti još i za svaku od (log(log n)) pozicija ispred \$. Možemo zamisliti da na tim pozicijama postoje "lažni" \$. Dok smo prije u skupu *Q* pamtili samo "prave" \$, sada ćemo u *Q* pamtiti i "lažne" \$.

Kao što smo već rekli, kada probamo prelomiti *P* na nekoj poziciji *h*, iduća pozicija na kojoj ćemo ga probati prelomiti će biti *h*+log(log *n*) te ga nećemo probati prelomiti na pozicijama između. Upravo to je razlog zašto smo uveli "lažne" \$, oni će nam omogućiti da istražimo i te pozicije. Naime, ako prelomimo *P* na nekoj poziciji *h* te u $T_\$$ nađemo odgovarajući \$, a taj \$ je ustvari "lažni" \$ koji je od prvog desnog ne-"lažnog" \$ udaljen *k* pozicija, to nam govori sljedeće: *P* zbilja postoji prelomljen u *T* ali nije prelomljen na poziciji *h* već na poziciji *h* + *k*. Naravno ovdje je potrebno pripaziti da li je pozicija *h* + *k* unutar teksta *P*, ako nije onda nećemo prijaviti to pojavljivanje. Pošto smo za svaki \$ zapamtili log(log *n*) "lažnih" \$ lijevo od njega znači da za svaki *h* možemo otkriti prijelome na idućih log(log *n*) pozicija desno od pozicije *h*. Na taj način pokrili smo sve pozicije u *P* i postigli jednak učinak kao i jednostavni algoritam iz 3.3.3 uz ubrzanje za faktor log(log *n*).

Opisani algoritam za dugački *P* zovemo `get_overlapping_long`. Vremenska složenost tog algoritma je $O(p + occ_0)$, a prostorno zauzeće $O(nH_k(T)log^\epsilon n) + O(n/log^{1-\epsilon}n)$.

4. Implementacija

U ovom poglavlju opisat ćemo detalje implementacije indeksa iz poglavlja 3.3. Spomenut ćemo neke posebnosti implementacije, razlike naspram postupaka opisanih u poglavlju 3.3, opisati strukturu koda i sučelje te navesti podatkovne strukture.

Indeks je implementiran u programskom jeziku C++, razvijan i testiran na operativnom sustavu Linux(Ubuntu).

4.1. Struktura koda i sučelje

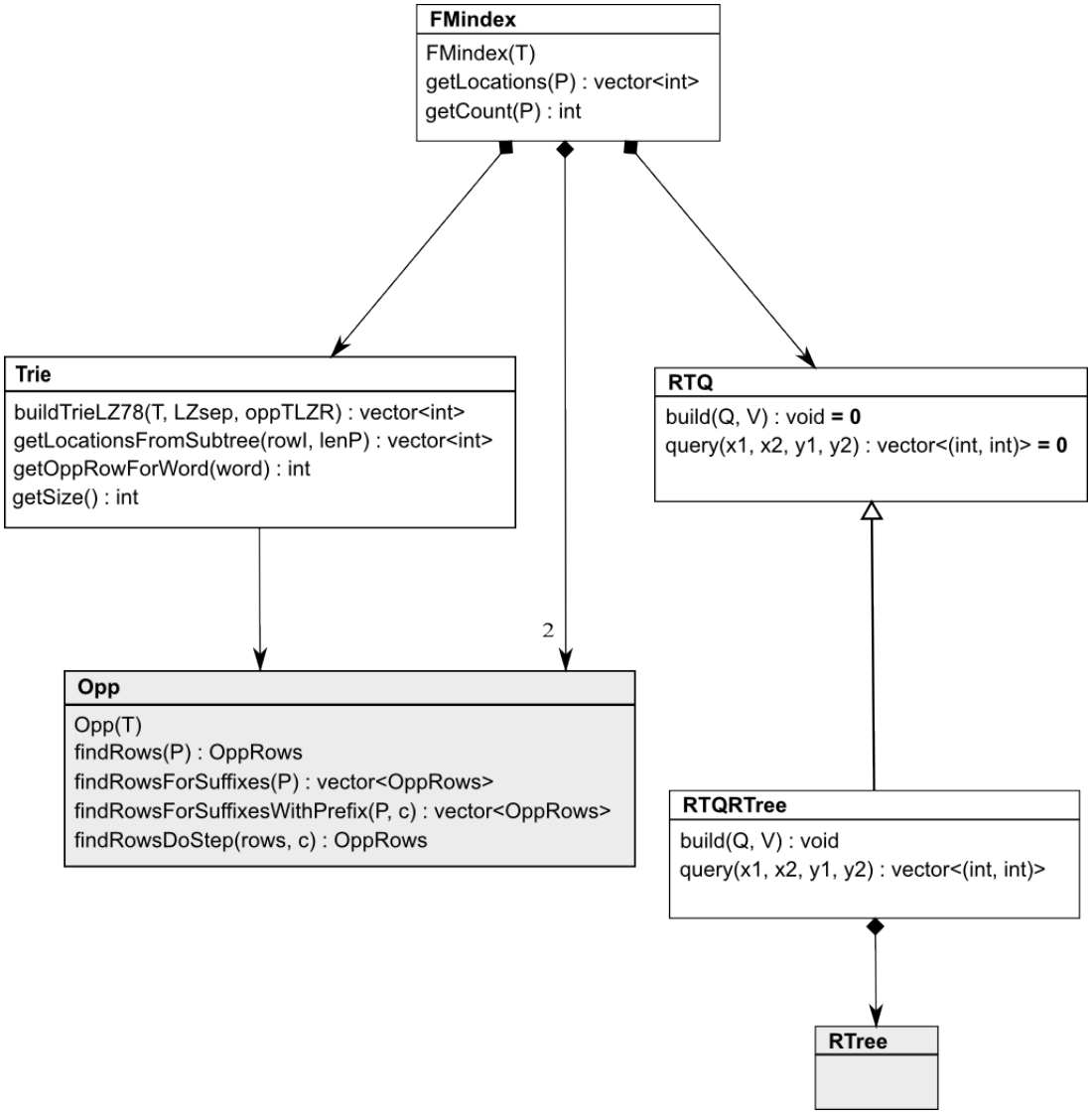
Implementacija indeksa je zasnovana na objektno-orijentiranoj paradigmi pa je tako kod podijeljen u nekoliko razreda:

- **FMindex** – vršni razred, sadrži podatkovne strukture indeksa i pruža sučelje za njegovo korištenje
- **Trie** – sufiks stablo
- **RTQ**(sučelje) – sučelje za podatkovnu strukturu $RT(Q)$
- **RTQRTree** – implementacija podatkovne strukture $RT(Q)$
- **Opp** – podatkovna struktura Opp
- **StringView** – pomoćni razred za brži rad sa c++ stringovima

Razredni dijagram je prikazan na slici 4.1. Razredi koji nisu implementirani u ovom radu već su korištene postojeće implementacije obojani su sivom bojom. Zbog preglednosti nije prikazan pomoćni razred StringView. Također zbog preglednosti nisu prikazani razredi koje koristi razred Opp pošto ionako nisu dio ovoga rada.

Indeks je implementiran sa idejom lakog daljnjeg korištenja. Iz tog razloga sučelje je napisano na jednostavan i razumljiv način, te se sastoji od samo dvije funkcije iz razreda FMindex:

- `getLocations`(P) – vraća pozicije pojavljivanja teksta P u tekstu T
- `getCount`(P) – vraća broj pojavljivanja teksta P u tekstu T



Slika 4.1: Razredni dijagram.

Korištenje indeksa u klijentskom kodu je zamišljeno tako da klijent jednom stvori instancu razreda FMindex za neki tekst T te zatim po želji poziva funkcije `getLocations`(P) i `getCount`(P).

4.2. Izgradnja indeksa

Izgradnja indeksa odvija se odmah pri konstrukciji instance razreda FMindex za dani tekst T. Iako smo se u poglavlju 3.3 ponajviše fokusirali na brzinu i učinkovitost pretraživanja, treba spomenuti da je također bitno učinkovito izvesti i izgradnju indeksa. Naime, brzinu i moć indeksa najbolje ćemo osjetiti na velikim tekstovima te bismo htjeli da izgradnja indeksa za velike tekstove traje neko razumno vrijeme. Bitno je

spomenuti da se kao bitno najsporiji dio izgradnje indeksa pokazalo pozivanje funkcija nad strukturom Opp (`findRows` i njene varijante). Zbog toga smo pri izgradnji izbjegavali pozivanje tih funkcija koliko god je moguće.

Pri izgradnji indeksa grade se i pamte sljedeće podatkovne strukture:

- sufiks stablo za tekst $T_{\$}^{R}$: razred Trie
- Opp(T): razred Opp
- Opp($T_{\$}^{R}$): razred Opp
- $RT(Q)$: razred RTQRTree i sučelje RTQ
- tablica T_s

Više o implementaciji izgradnje pojedinih struktura reći ćemo u sekciji 4.3.

Pri izgradnji računamo i pamtimo *prag duljine*, vrijednost na temelju koje odlučujemo da li je tekst P kratak ili dugačak. Iako smo prije spomenuli da je *prag duljine* jednak $\log(\log n)$, treba biti oprezan kada je $n < 4$ jer tada logaritam postaje negativan broj.

Pri izgradnji je također bitno dobro izabrati dva posebna znaka: znak \$ koji se koristi za razdvajanje LZ riječi i znak # koji se koristi kao znak kraja teksta u konceptualnoj matrici strukture Opp. Da bi indeks radio dobro znak # mora biti manji od znaka \$, a \$ mora biti manji od svih znakova iz abecede Σ. U našoj implementaciji postavili smo znak # na znak s ASCII vrijednošću 0 a znak \$ na znak s ASCII vrijednošću 1.

4.3. Podatkovne strukture

Osnovni sastavni dijelovi indeksa su njegove podatkovne strukture. U ovoj sekciji ćemo opisati implementacije pojedinih struktura.

4.3.1. Sufiks stablo

Implementaciju sufiks stabla opisanog u 3.3.2 ostvarili smo razredom Trie. Sufiks stablo smo ostvarili kao čvorove povezane pokazivačima u samo jednom smjeru, dakle otac sadrži pokazivače na djecu ali ne i ona na njega. Svaki čvor osim korijena stabla predstavlja jednu LZ riječ T_i iz $T_{\$}$.

Bridovi su označeni znakovima iz abecede Σ. Zbog toga svaki čvor pokazivače na svoju djecu drži u tablici raspršenog adresiranja gdje je ključ slovo iz abecede.

U svakom čvoru osim korijena, za razliku od opisa iz 3.3.2, uz v_i (početna pozicija riječi T_i u T) pamtimo još i $|T_i|$). Dodatno pamtimo duljinu kako bismo mogli pri

obilasku stabla u algoritmu `get_internal` izračunati poziciju P u T. Kada ne bismo pamtili duljinu, morali bismo prvi svakom obilasku podstabla prvo prošetati do korijena sufiks stabla da bi doznali duljinu korijena podstabla.

Izgradnja

Iz praktičnih razloga i veće efikasnosti izgradnje napravili smo da sufiks stablo pri svojoj izgradnji ujedno parsira tekst T LZ78 algoritmom i gradi strukturu Opp($T^R_\$$). Gradnja je ostvarena funkcijom `buildTrieLZ78` koja prima tekst T, gradi stablo te vraća tekst $T^R_\$$ i strukturu Opp($T^R_\$$).

Gradnju stabla smo ostvarili tako da se odvija isprepleteno sa LZ78 parsiranjem teksta T, sada ćemo pojasniti zašto. Kada bismo prvo radili LZ78 parsiranje, morali bismo imati rječnik sa LZ rječima koje smo do sada našli te se za svaki znak iz T s kojim pokušavamo izgraditi novu riječ pitati postoji li već takva riječ u rječniku. Nakon toga bismo u nekom kasnijem trenutku izgradili stablo na temelju tih riječi. Kako bismo izbjegli da za svaki znak iz T moramo raditi upit prema rječniku, isprepličemo gradnju stabla i parsiranje. Naime, uobičajeno stablo gradimo tako da za svaku LZ riječ uzimamo znak po znak iz te riječi i krećemo se po stablu, ako nekog znaka nema tada ga dodamo. Ako bismo malo promijenili postupak i uzimali znak po znak iz teksta T a ne iz svake LZ riječi, onda ćemo morati dodavati novo slovo u stablo jedino u slučaju kada naiđemo na novu LZ riječ. Na taj način više ne moramo prvo napraviti LZ78 parsiranje teksta pa onda graditi stablo, već obadvoje radimo paralelno. To je efikasnije jer pri gradnji rječnika treba stalno zapitkivati postoji li neka LZ riječ već u rječniku pa bi nam trebala neka podatkovna struktura koja omogućuje takve brze upite, dok smo ovako u tu svrhu iskoristili sufiks stablo i ujedno ga kroz upite izgradili.

Pri izgradnji stabla trebamo posebnu pozornost posvetiti slučaju spomenutom u 3.3.2, kada imamo dvije iste LZ riječi. Kao što je već opisano, takvu LZ riječ dodajemo kao dijete riječi koja je ista kao ona. Brid označimo posebnim znakom & kako bi prepoznali takvu riječ.

Kao zadnji korak izgradnje radi se preslikavanje redaka konceptualne matrice $M_{T^R_\$}$ u čvorove stabla. Naći ćemo toliko preslikavanja koliko ima čvorova, za svaki čvor po jedno, i pospremiti ih u niz N. Preslikavanja nalazimo tako da za svaki čvor tj. riječ predstavljenu tim čvorom nađemo njen pripadajući redak u matrici. No, to nećemo raditi tako da za svaki čvor zovemo funkciju `findRows` već ćemo za svaki čvor zvati funkciju `findRowsDoStep` (prisjetimo se, vremenska složenost `findRowsDoStep` je $O(1)$ dok je vremenska složenost `findRows` $O(p)$). Koristit ćemo rezultate rodi-

teljskih čvorova kako bi našli brojeve redaka za djecu i tako ćemo drastično ubrzati izgradnju niza *N*.

Još jedna optimizacija koju smo uveli je pamćenje odgovarajućeg retka iz matrice za svaki čvor. Dok u nizu *N* za određene redove matrice pamtimo njima pripadajuće čvorove, sada ćemo također u svakom čvoru zapamtiti i njemu odgovarajući redak. To radimo zato jer će se pri izgradnji podatkovne strukture *RT*(*Q*) pojaviti potreba za nalaženjem odgovarajućeg retka u matrici za svaku LZ riječ. Pošto smo već u sufiks stablu zapamtili za svaki čvor koji je njegov pripadajući redak u matrici, biti će dovoljno samo očitati iz stabla taj redak. Tako ne moramo zvati funkcije strukture Opp, koje su bitno sporije od upita prema sufiks stablu te postižemo znatno ubrzanje pri izgradnji skupa *Q*.

Obilazak

Pri obilasku jednostavno rekurzivno obilazimo neko zadano podstablo. Dok obilazimo čvorove, za svaki čvor izračunamo poziciju kao što je opisano u 3.3.2.

Poseban slučaj na koji moramo pripaziti je kada u stablu imamo čvor koji predstavlja LZ riječ koja već postoji, označimo taj čvor sa *D*. Čvor *D* sa svojim je roditeljem povezan bridom & te ćemo ga tako prepoznati. Kao što je već spomenuto u 3.3.2, čvor *D* konceptualno promatramo kao da je susjed svog roditelja a ne njegovo dijete. Kako bismo pravilno obišli takav čvor, uvodimo slijedeće pravilo: Ako obilazimo podstablo s korijenom *K* i pri tome naiđemo na čvor *D* koji je direktno dijete od *K*, tada preskačemo čvor *D*, inače ne.

4.3.2. *RT*(*Q*)

Podatkovna struktura *RT* podržava ortogonalnu pretragu zadanog raspona što znači da za skup točaka *Q* i raspon pretrage $((x_{min}, x_{max}), (y_{min}, y_{max}))$ možemo efikasno pronaći sve točke (x_i, y_j) iz tog raspona.

U našoj implementaciji podatkovnu strukturu *RT* nad skupom *Q* predstavili smo razredom RTQ. Razred RTQ ne sadrži implementaciju već je samo sučelje koje pruža dvije funkcije: `build` i `query`. Samu implementaciju ostvarili smo razredom RTQRTree, koji ostvaruje sučelje opisano razredom RTQ.

U 3.3.3 smo govorili o *RT*(*Q*) strukturi opisanoj u [6] koja podržava pretragu u $O(log(logn))$ vremenu gdje je *n* broj točaka u skupu *Q*. Nažalost zbog nedostupnosti implementacije navedene strukture u razredu RTQRTree koristili smo implementaciju *RTree* autora Greg Douglas-a. Navedena implementacija nema složenost pretrage

$O(log(logn))$ kao što je prethodno opisano već $O(logn)$. Zbog toga vremenska složenost lociranja pojavljivanja teksta *P* u tekstu *T* ipak nije $O(p + occ)$ kao što smo zamislili već $O(p\frac{logn}{log(logn)} + occ)$. Navedeno je razlog zašto smo uveli sučelje RTQ: zato da u budućnosti lako možemo trenutnu implementaciju zamijeniti sa bržom implementacijom, samo je potrebno ponuditi drugačiju implementaciju sučelja RTQ.

Izgradnja skupa *Q*

Izgradnja skupa *Q* je korak koji prethodi izgradnji same *RT*(*Q*) strukture. Skup *Q* se gradi kao što je opisano u sekciji 3.3.3. Za svaki znak \$ iz $T_\$$ i za "lažne" \$ na svakoj od *k* pozicija ispred znaka \$ tražimo dvojku (x_i, y_i). Prema postupku koji smo opisali u 3.3.3, da nađemo jednu dvojku trebala bi nam dva poziva funkcije `findRows`. To znači da bi nam ukupno trebalo 2*|*Q*| poziva funkcije `findRows`.

Prethodno smo već spomenuli kako su upravo pozivi funkcije `findRows` najsporiji dio izgradnje indeksa, pa bismo htjeli smanjiti broj poziva. Pokazati ćemo da 2*|*Q*| poziva funkcije `findRows` možemo zamijeniti sa samo jednim pozivom funkcije `findRowsForSuffixes` i upitima prema sufiks stablu.
Prvo ćemo primijetiti da dok računamo vrijednost x_i tražimo retke konceptualne matrice $M_{T^R_\$}$ koji odgovaraju nekoj LZ riječi. U 4.3.1 smo rekli kako ćemo za svaku LZ riječ u sufiks stablu zapamtiti njoj odgovarajuće retke matrice. To ćemo iskoristiti ovdje te nećemo zvati funkciju (findRows) već ćemo iz sufiks stabla pročitati retke matrice. Pošto je poziv funkcije (findRows) puno sporiji od pristupa LZ riječi u sufiks stablu, ovime ćemo uštediti znatnu količinu vremena.
Preostaje nam još |*Q*| poziva funkcije `findRows`, koji se koriste pri računanju vrijednosti y_i. Ovdje možemo primijetiti da funkciju `findRows` pozivamo kako bismo za neki sufiks teskta *T* doznali retke konceptualne matrice M_T. Kao što je opisano u 3.2.2, takve pozive možemo zamijeniti samo jednim pozivom funkcije `findRowsForSuffixes` čime se postiže znatno ubrzanje.

4.3.3. Opp

Podatkovna struktura Opp ključna je struktura indeksa, u kôdu je predstavljena istoimenim razredom. Detaljnije smo ju opisali u poglavlju 3.2. Koristimo već gotovu implementaciju autora Matije Šošića [7] koja nudi funkcije opisane u poglavlju 3.2.

4.3.4. Tablica T_s

Tablica T_s se koristi za pamćenje kratkih prelomljenih tekstova iz teksta *T*, kao što je opisano u 3.3.3. T_s smo implementirali kao tablicu raspršenog adresiranja gdje je ključ kratki tekst a sadržaj je niz sa popisom pozicija u *T* na kojima se pojavljuje.

4.4. Operacije

Kratko ćemo opisati operacije koje indeks nudi i funkcije koje ih izvode.

4.4.1. Lociranje

Lociranje svih pojavljivanja nekog teksta *P* u tekstu *T* je implementirano funkcijom `getLocations`. Ulaz funkcije je tekst *P* a izlaz je niz sa pozicijama svih pojavljivanja. U nizu se neće nalaziti dvije iste pozicije i nema garancije da će niz biti sortiran.

Lociranje se odvija u dva koraka, prvo se traže unutarnja pojavljivanja a zatim vanjska pojavljivanja. Traženje vanjskih pojavljivanja se još dijeli na traženje kratkih pojavljivanja i traženje dugih pojavljivanja. Na kraju se pozicije svih pojavljivanja spajaju u zajednički niz. Vremenska složenost je $O(p\frac{logn}{log(logn)} + occ)$.

4.4.2. Brojanje

Brojanje svih pojavljivanja nekog teksta *P* u tekstu *T* je implementirano funkcijom `getCount`. Ulaz funkcije je tekst *P* a izlaz je broj pojavljivanja.

Brojanje se odvija tako da se poziva funkcija `findRows` nad strukturom Opp(*T*) te se vraća vrijednost (*last* - *first* + 1) gdje je *first* vrijednost prvog retka u M_T a *last* vrijednost zadnjeg. Vremenska složenost je $O(p)$.

5. Primjena u sastavljanju genoma

5.1. Algoritmi poravnavanja

Jednu od ključnih uloga u bioinformatici imaju algoritmi poravnavanja slijedova. Zadaća takvih algoritama je poravnanje dvaju ili više slijedova DNK, RNK i proteina te ocjena sličnosti na temelju poravnanja. Sam postupak je vremenski i memorijski vrlo zahtjevan zbog čega je izazov napraviti algoritam koji radi dovoljno precizno i brzo.

Jedan od algoritama koji se koristi za poravnavanje slijedova je *Smith-Waterman* [8] algoritam. Za razliku od nekih drugih algoritama za poravnavanje *Smith-Waterman* algoritam je deterministički algoritam, što znači da uvijek vraća optimalno rješenje. Iz tog razloga je bitno sporiji od nedeterminističkih algoritama ali garantira optimalne rezultate.

5.2. Problem sastavljanja genoma

Genom je predstavljen kao niz slova `A`, `C`, `T` i `G`. Postavimo sljedeći problem (problem sastavljanja genoma): imamo zadan jedan cijeli genom G i malene komadiće (očitanja) genoma. Potrebno je za svako očitanje pronaći odgovarajuću poziciju u genomu G. To je problem koji bismo brzo i efikasno mogli riješiti pomoću našeg indeksa: izgradili bismo indeks nad genomom G te zatim za svako očitanje pronašli lokaciju u G koristeći indeks. No, problem s kojim se mi suočavamo je nešto složeniji: naime očitanja često nisu sasvim jednaka kao njihovi odgovarajući djelići u genomu G. Neka slova u očitanjima su drugačija, neka slova nedostaju, a neka slova su višak. Takve pojave nazivati ćemo pogreškama u očitanju. Zbog toga nam je potreban algoritam za poravnavanje kao što je *Smith-Waterman* algoritam. Za neko očitanje nam *Smith-Waterman* algoritam može reći koja je njegova najbolja pozicija u genomu G, unatoč tome što očitanje nije sasvim jednako kao odgovarajući djelić. Treba napomenuti kako su neka očitanja možda obrnuta i komplementirana, pa treba za svako očitanje pronaći

i moguće pozicije obrnutog komplementa.

5.3. Primjena indeksa

U ovom radu primjenili smo našu implementaciju indeksa u kombinaciji sa *Smith-Waterman* algoritmom kako bismo ubrzali sastavljanje genoma. Koristili smo implementaciju *Smith-Waterman* algoritma na grafičkoj kartici autora Matije Korpara [9] koja paralelnim izvođenjem postiže znatno ubrzanje s obzirom na CPU verziju. Osnovna ideja je smanjiti prostor pretrage *Smith-Waterman* algoritma. Naime *Smith-Waterman* je sam po sebi spor te mu je vremenska složenost *O*(*duljina_genoma * duljina_očitanja*), no ako bismo mu smanjili prostor pretrage (duljinu genoma ili duljinu očitanja) postigli bismo bitno ubrzanje cijelog algoritma sastavljanja genoma.

Prostor pretrage ćemo smanjiti tako da najprije na manje precizan način odredimo moguće pozicije u genomu *G* gdje bi se moglo nalaziti očitanje *P*. Tako *Smith-Waterman* algoritam više neće pretraživati po cijelom genomu *G* već samo dijelove genoma *G* koje smo proglasili kao moguće pozicije. Pronalazak mogućih pozicija možemo učiniti na mnogo raznih načina, pri čemu je najbitnije da za svako očitanje nađemo što manje mogućih pozicija pri tome ne gubeći najbolju (pravu) poziciju. Pri pronalaženju mogućih pozicija koristili smo se našom implementacijom indeksa kako bismo mogli brzo pretraživati po genomu *G*. Nakon što nađemo moguće pozicije za neko očitanje *P*, koristimo *Smith-Waterman* algoritam kako bismo odabrali najbolju poziciju od mogućih, te je ona naše rješenje (za očitanje *P*). Iako ovakav algoritam više nije deterministički (moguće je da niti jedna od mogućih pozicija koje smo odabrali nije najbolja pozicija, tj. moguće je izgubiti najbolje rješenje), ako smo dovoljno dobro birali moguće pozicije očekujemo da će algoritam postići znatno ubrzanje bez velikog gubitka na preciznosti.

5.3.1. Odabir mogućih pozicija

Moguće pozicije u genomu *G* za očitanje *O* biramo na sljedeći način. Nad genomom *G* izgradimo indeks. Zatim iz očitanja *O* izdvojimo prefiks *L* i sufiks *R*, oboje duljine *k*. Također izdvojimo i preostali dio očitanja, sredinu *M*. Koristeći izgrađeni indeks pronađemo sve pozicije pojavljivanja *L* i *R* u genomu *G*. Za svaki par (*l*, *r*) gdje je *l* pozicija pojavljivanja *L* u *G*, a *r* pozicija pojavljivanja *R* u *G* kažemo da je *dobar* ako se udaljenost između *l* i *r* ne razlikuje od udaljenosti između *L* i *R* u *O* za više od nekog proizvoljno odabranog prirodnog broja *e*. Za sve parove (*l*, *r*) koji su *dobri* kažemo

da su moguće pojavljivanje *O* u *G*, pa je tako pozicija određena parom (*l*, *r*) jedna od mogućih pozicija *O* u *G*.

Opisanim postupkom smo omogućili da se u središnjem dijelu *M* može pojaviti pogreška, no nismo omogućili pojavu pogreške u *L* ili *R* dijelovima. Kako bismo i tamo omogućili pogrešku pronaći ćemo pomoću našeg indeksa sva pojavljivanja *M* u *G*. Za svako takvo pojavljivanje ćemo reći da određuje jednu od mogućih pozicija *O* u *G*. Do sada opisani postupak smo u implementaciji ostvarili funkcijom `getCandidatesFast`.

Algorithm 1 `getCandidatesFast`

Ulaz: *P* - očitanje; *index* - index nad genomom *G*.
Izlaz: Niz parova (*first*, *last*) gdje svaki par predstavlja jedno moguće pojavljivanje *P* u *G*. *first* je pozicija prvog slova pojavljivanja, *last* je pozicija zadnjeg slova pojavljivanja.
$mogucaPojavljivanja := []$
$L := P[0, k]$, $M := P[k, p-k]$, $R := P[p-k, p]$
$locsL := index.getLocations(L)$
$locsM := index.getLocations(M)$
$locsR := index.getLocations(R)$
for all $l \in locsL$ **do**
  for all $r \in locsR$ **do**
    if $abs((r-l-1)-(p-2*k)) \leq e$ **then**
      dodaj $(l, r+k)$ **u** $mogucaPojavljivanja$
    end if
  end for
end for
for all $m \in locsM$ **do**
  dodaj $(m-k, m-k+p)$ **u** $mogucaPojavljivanja$
end for
return $mogucaPojavljivanja$

Opisani postupak dodatno ćemo promijeniti kako bismo omogućili raznovrsniji raspored pogrešaka u očitanju *O* i time postigli bolje rezultate. Za razliku od funkcije `getCandidatesFast`, promijeniti ćemo postupak tako da dopušta pogreške na dvije razine a ne na samo jednoj. To ćemo učiniti tako da pozicije pojavljivanja *L* i *R* u *G* ne tražimo koristeći naš indeks(koji ne dopušta pogreške) već koristimo funkciju `getCandidatesFast`. Tako ćemo dobiti moguća pojavljivanja *L* i *R* u *G* koja

mogu u sebi imati pogreške. Na isti ćemo način naći i moguća pojavljivanja *M* u *G*. Ostalo je sve isto kao u funkciji `getCandidatesFast`. Ovaj promijenjeni postupak u implementaciji smo ostvarili funkcijom `getCandidatesFastRecursive` i njega u ovom radu koristimo kako bismo ubrzali sastavljanje genoma.

Algorithm 2 `getCandidatesFastRecursive`

Ulaz: *P* - očitanje; *index* - index nad genomom *G*.
Izlaz: Niz parova (*first*, *last*) gdje svaki par predstavlja jedno moguće pojavljivanje *P* u *G*. *first* je pozicija prvog slova pojavljivanja, *last* je pozicija zadnjeg slova pojavljivanja.
$mogucaPojavljivanja := []$
$L := P[0, k]$, $M := P[k, p-k]$, $R := P[p-k, p]$
$locsL := getCandidatesFast(L, index)$
$locsM := getCandidatesFast(M, index)$
$locsR := getCandidatesFast(R, index)$
for all $l \in locsL$ **do**
    for all $r \in locsR$ **do**
        if $abs((r.first - l.first - 1) - (p - 2 * k)) \leq e$ **then**
            dodaj $(l.first, r.last)$ **u** $mogucaPojavljivanja$
        end if
    end for
end for
for all $m \in locsM$ **do**
    dodaj $(m.first - k, m.last + k)$ **u** $mogucaPojavljivanja$
end for
return $mogucaPojavljivanja$

6. Rezultati i diskusija

U ovom poglavlju prikazat ćemo brzinu izgradnje indeksa, brzinu lociranja i brzinu brojanja pojavljivanja za različite ulaze. Vrijeme izvođenja smo mjerili pomoću standardne C++ funkcije clock() iz knjižnice ctime. Zauzeće memorije smo mjerili alatom *massif* koji je dio alata *valgrind*. Testiranja smo provodili na procesoru Quad-Core@2.43GHz, 8GB RAM, grafička kartica NVIDIA GTX 570.

6.1. Testiranje indeksa

Program smo testirali na velikim tekstovima preuzetim sa stranice `http://pizzachili.di.unipi.it`:

- **dna** – DNA sekvenca koja se sastoji samo od slova A, G, C, T. Omjer kompresije pri entropiji 0-tog reda: 25%
- **proteins** – Proteinske sekvence dobivene iz *Swissprot* baze proteina. Svaka od 20 aminokiselina je kodirana kao jedno veliko slovo. Omjer kompresije pri entropiji 0-tog reda: 52%
- **english** – Više tekstova na engleskom jeziku spojenih zajedno, odabranih iz *Guttenberg* projekta. Omjer kompresije pri entropiji 0-tog reda: 57%

Testove smo provodili tako da na velikom tekstu prvo izgradimo indeks, zatim iz njega uzorkujemo 1000 uzoraka duljine 50 znakova te ih tražimo po zadanom tekstu. Rezultate smo prikazali kroz tri tablice. Svaki redak tablice ima zapis o broju znakova uzetih s početka velikog teksta nad kojima se gradio indeks, zauzeće tih znakova u MB, vrijeme izgradnje indeksa, vrijeme lociranja po uzorku, vrijeme brojanja po uzorku i najveće zauzeće memorije u MB tijekom izgradnje indeksa.

Br. zn.	[MB]	Vr. izgradnje	Vr. loc. po uz.	Vr. br. po uz.
1000000	1	27.70	0.00117	0.00070
5000000	5	144.65	0.00105	0.00097
10000000	10	228.59	0.00136	0.00067
25000000	25	756.23	0.00148	0.00076

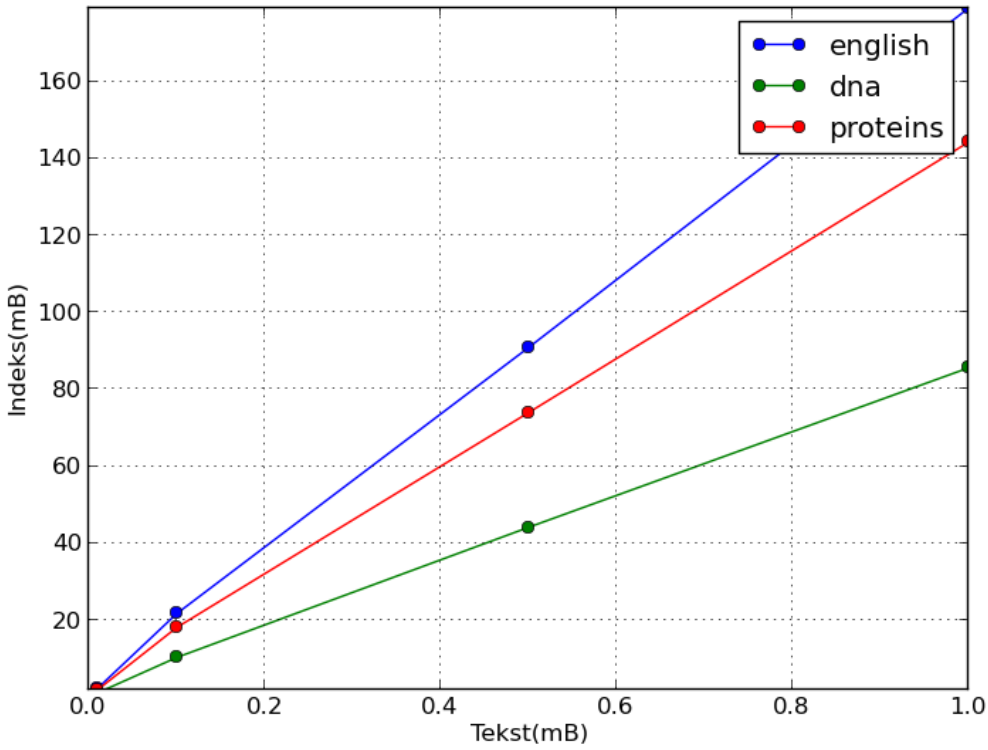
Tablica 6.1: Rezultati za tekst english. Vremena su u sekundama.

Br. zn.	[MB]	Vr. izgradnje	Vr. loc. po uz.	Vr. br. po uz.
1000000	1	14.64	0.00065	0.00038
5000000	5	77.97	0.00074	0.00045
10000000	10	228.86	0.00122	0.00057
25000000	25	516.53	0.00090	0.00071

Tablica 6.2: Rezultati za tekst dna. Vremena su u sekundama.

Br. zn.	[MB]	Vr. izgradnje	Vr. loc. po uz.	Vr. br. po uz.
1000000	1	23.37	0.00073	0.00055
5000000	5	127.82	0.00092	0.00070
10000000	10	231.24	0.00090	0.00067
25000000	25	613.92	0.00109	0.00055

Tablica 6.3: Rezultati za tekst proteins. Vremena su u sekundama.



Slika 6.1: Memorijsko zauzeće indeksa.

Rezultati su pokazali da vrijeme izgradnje indeksa raste linearno sa veličinom teksta nad kojim se gradi indeks. Vrijeme lociranja također raste s veličinom teksta nad kojim se gradi indeks i to zbog logaritamske ovisnosti o n, koju nismo uspjeli izbjeći zbog nedostatka odgovarajuće implementacije *RTQ* strukture. Iako ovisno u n, vrijeme lociranja raste vrlo sporo, pa tako za porast n-a od 25 puta vrijeme lociranja poraste za samo 50%. S druge strane, vrijeme brojanja se također mijenja s n, iako teoretski ne bi trebalo ovisiti o n. Naime zbog praktičnosti i prevelikog zauzeća memorije implementacija *Opp* koju koristimo logaritamski ovisi o n. U prikazanim rezultatima možemo primjetiti kako vrijeme brojanja ne raste neprestano s veličinom teksta već povremeno i pada, a slično ponašanje se nazire i kod vremena lociranja. Iako je teško točno reći zbog čega dolazi do toga, pretpostavljamo da utjecaj ima veće zauzeće memorije koje se događa kod većih tekstova. Naime, neki segmenti zauzeća memorije ovise o broju $log(log(n))$ koji se u implementaciji zaokružuje na cijeli broj. Zbog toga memorije ne raste sasvim jednoliko već se pojavljuju skokovi kad $log(log(n))$ (zaokružen na cijeli broj) poraste. Tada se zauzima više memorije ali dolazi i do ubrzanja algoritma. Na slici 6.1 prikazano je maksimalno zauzeće memorije pri izgradnji indeksa za raz-

ličite tekstove. Vidimo da je manje memorije zauzeto za tekst koji ima bolja svojstva kompresije (tekst dna) i da je porast memorijskog zauzeća linearan.

6.2. Testiranje sastavljanja genoma

Testiranje smo proveli na primjerku ljudske DNA sa stranice *ftp://ftp.ensembl.org/pub/release-67/fasta/homo_sapiens/dna/* od koje je sa početka uzet dio duljine d. Očitanja su simulirana alatom *wgsim* dostupnim na *https://github.com/lh3/wgsim*. Alatom *wgsim* smo napravili m očitanja duljine 70. Na prvom skupu testova koristili smo podrazumijevane postavke *wgsim*-a, dok smo za drugi skup podesili e (vjerojatnost pogreške) i r (vjerojatnost mutacije). Što se tiče algoritama opisanih u 5.3.1, koristili smo sljedeće parametre:
`getCandidatesFastRecursive` -> $k = 20\%p$, $e = 7\%p$ i
`getCandidatesFast` -> $k = 30\%p$, $e = 7\%p$, gdje je p duljina očitanja.

d	m	e	r	Vr. izgr.	Vr. sastav.	Preciznost
1000	100	0.02	0.001	0.83s	9s	99.00%
1000	1000	0.02	0.001	0.83s	1m40s	98.40%
10000	100	0.02	0.001	0.83s	38s	68.00%
10000	1000	0.02	0.001	0.83s	6m40s	72.90%
10000	10000	0.02	0.001	23s	72m30s	71.25%
10000	100	0.01	0.005	23s	39s	71.00%
10000	1000	0.01	0.005	23s	7m47s	70.50%
10000	10000	0.01	0.005	23s	79m21s	70.30%

Tablica 6.4: Rezultati sastavljanja genoma.

Vrijeme izgradnje smo izdvojili od vremena sastavljanja jer je izgradnju dovoljno napraviti samo jednom unaprijed. Rezultati su pokazali kako je i dalje znatno najsporiji dio *Smith-Waterman* algoritam. To znači da bi se daljnja ubrzanja mogla postići boljim usmjeravanjem *Smith-Waterman* algoritma tj. izborom bolje metode za odabir mogućih pozicija pojavljivanja očitanja. Pogreške se pojavljuju iz više razloga, jedan od njih je nemogućnost korištene metode da pokrije sve moguće kombinacije pogrešaka. Moguće je i da neka od očitanja nije moguće točno locirati jer im zbog pogrešaka bolje odgovara neka kriva pozicija u genomu. Također dolazi zbog pogrešaka jer za

svako očitanje moramo gledati i obrnuti komplement, pa je moguće da od to dvoje ono koje je pogrešno bolje pristaje u genom.

7. Zaključak

U radu smo detaljno objasnili ideju i implementaciju drugog indeksa iz [3]. Objasnili smo temeljnu ideju prvog indeksa iz [3] i nadodali na njega funkcije potrebne za izradu drugog indeksa. Zatim smo se detaljnije upustili u objašnjavanje drugog indeksa. Razmotrili smo razne pristupe izgradnji indeksa i lociranju, pokazali određene nedostatke i predstavili rješenja implementacijskih problema.

Pokazali smo da se ispreplitanjem izgradnje sufiks stabla i LZ78 parsiranja može postići puno bolja efikasnost nego odvajanjem ta dva koraka. Također smo predložili i isprobali način za rješavanje problema dvostrukih LZ riječi u sufiks stablu.

Posebnu smo pozornost posvetili uporabi funkcija koje pruža struktura Opp. Naime, iako je u [3] opisana samo jedna funkcija nad tom strukturom, ograničavanje na isključivo njeno korištenje pokazalo se neefikasnim. Zato smo predložili i izveli nove funkcije nad strukturom Opp koje nam daju veću fleksibilnost i omogućuju da puno efikasnije provedemo upite nad njom. Na taj smo način na nekoliko mjesta izbjegli suvišno računanje i bitno ubrzali postupak izgradnje indeksa.

Zbog nedostupnosti implementacije *RT* strukture nismo mogli u praksi ostvariti teoretsku brzinu lociranja. Unatoč tome rezultati testiranja su pokazali izuzetnu stabilnost vremena lociranja i brojanja pojavljivanja. Naime, porastom veličine teksta došlo je tek do malog porasta u vremenu lociranja i brojanja.

Vjerujemo da bi u budućnosti bilo zanimljivo detaljnije proučiti utjecaj karakteristika teksta na brzinu izgradnje i izvođenja operacija indeksa. Naime već kod testiranja na malim tekstovima vidi se velika razlika u brzini izgradnje ovisno o tipu teksta (engleski tekst, dna). Također bi bilo dobro naći bolju implementaciju *RT* strukture, idealno sa vremenskom složenošću $log(logn)$ čime bi se izbjegla ovisnost o duljini teksta pri lociranju. U primjeni indeksa na sastavljanje genoma kao bitno najsporiji dio i dalje se pokazao *Smith-Waterman* algoritam.

LITERATURA

[1] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pp. 390–398, IEEE, 2000.

[2] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Comput. Surv.*, vol. 39, Apr. 2007.

[3] P. Ferragina and G. Manzini, "Indexing compressed text," *J. ACM*, vol. 52, pp. 552–581, July 2005.

[4] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *Information Theory, IEEE Transactions on*, vol. 24, no. 5, pp. 530–536, 1978.

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

[6] S. Alstrup, G. Stolting Brodal, and T. Rauhe, "New data structures for orthogonal range searching," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pp. 198–207, IEEE, 2000.

[7] M. Šošić, "Implementacija FM indeksa," 2012.

[8] T. Smith, M. Waterman, and W. Fitch, "Comparative biosequence metrics," *Journal of Molecular Evolution*, vol. 18, no. 1, pp. 38–46, 1981.

[9] M. Korpar, "Implementacija smith waterman algoritma koristeći grafičke kartice s cuda arhitekturom," 2011.

Implementacija komprimirane podatkovne strukture za pretraživanje teksta temeljene na FMindeksu

Sažetak

FMindeks autora Ferragine i Manzini-a posljednjih je godina jako popularan. FMindeks je komprimirana struktura podataka koja omogućuje brzo i učinkovito pretraživanje teksta uz zauzeće memorije ovisno o veličini sažetog teksta. U ovom radu implementirali smo FMindeks u jeziku C++ i isprobali njegovu uporabu na raznim tipovima teksta. Za razliku od idealne implementacije indeksa koja ima složenost lociranja $O(p)$, naša implementacija radi lociranje u složenosti $O(p(logn)/(loglogn))$. Unatoč tome testovi su pokazali da indeks radi brzo i da vrijeme lociranja s porastom teksta raste vrlo sporo. Pokazali smo i primjenu indeksa u sastavljanju genoma.

Ključne riječi: sažimanje, FMindeks, indeks, implementacija, bioinformatika, genom, sastavljanje

Implementation of compressed data structure for text searching based on FMindex

Abstract

Ferragina and Manzini's FMindex has attracted great attention in last few years. FMindex is compressed data structure that supports fast and efficient substring searches using roughly the space required for storing the text in compressed form. In this thesis we implemented FMindex using programming language C++ and tested it on different types of text. Although theoretical time complexity of locating the occurences is $O(p)$, our implemention achieves complexity of $O(p(logn)/(loglogn))$. Tests have shown that index is working fast and that time needed for locating the occurences does not grow significantly with larger text. We also showed usage of index in genome assembly.

Keywords: compressed, FMindex, index, implementation, bioinformatics, genome, assembly