UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Ana Knežević

PRONALAŽENJE MAKSIMALNIH I SUPERMAKSIMALNIH PONAVLJANJA U DNK SEKVENCAMA PRIMENOM POBOLJŠANIH SUFIKSNIH NIZOVA

master rad

Beograd, 2026.

Mentor:

dr Vesna Marinković, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Jovana Kovačević, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Mirjana Maljković Ružičić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Tati, mami i sestri

Naslov master rada: Pronalaženje maksimalnih i supermaksimalnih ponavljanja u DNK sekvencama primenom poboljšanih sufiksnih nizova

Rezime: Ponavljanja čine značajan deo genomskih sekvenci, a njihovo efikasno pronalaženje predstavlja važan zadatak u bioinformatici. U ovom radu implementiran je poboljšani sufiksni niz i primenjen na traženje maksimalnih i supermaksimalnih egzaktnih ponavljanja u DNK sekvencama. Implementacija je napisana u jeziku C++17 i sadrži tri algoritma za konstrukciju sufiksnog niza: osnovni algoritam zasnovan na udvostručavanju prefiksa, njegovu optimizovanu varijantu i algoritam SA-IS. Nad sufiksnim nizom grade se inverzni sufiksni niz, niz najdužih zajedničkih prefiksa, Barouz–Vilerova transformacija i stablo LCP intervala. Za pronalaženje maksimalnih ponavljanja implementirana su dva pristupa. Osnovni pristup zasniva se na eksplicitnom određivanju maksimalnih ponovljenih parova, dok optimizovani pristup koristi LCP intervale i informacije o levim kontekstima dobijene iz Barouz–Vilerove transformacije. Supermaksimalna ponavljanja određuju se analizom odgovarajućih LCP intervala i njihovih levih konteksta. Ispravnost implementacije proverena je automatskim testovima, a performanse su analizirane na sintetičkim sekvencama dužine od 10 000 do 500 000 baza i na četiri kompletna genoma. Na najvećem analiziranom genomu algoritam SA-IS je bio 3,83 puta brži od optimizovanog algoritma udvostručavanja prefiksa. Optimizovani postupak za pronalaženje maksimalnih ponavljanja na realnim genomima ostvario je ubrzanje od približno 1,8 puta, uz povećanje maksimalne potrošnje memorije do 10,7%.

Ključne reči: poboljšani sufiksni niz, sufiksni niz, LCP niz, Barouz–Vilerova transformacija, SA-IS, maksimalna ponavljanja, supermaksimalna ponavljanja, DNK sekvence, bioinformatika

Sadržaj

Glava 1

Uvod

1.1 Motivacija

Savremene tehnologije sekvenciranja omogućavaju generisanje velikih količina genomskih podataka, zbog čega njihova efikasna analiza predstavlja značajan računarski izazov. Genom bakterije ima nekoliko miliona baza, ljudski genom oko tri milijarde, a projekti koji obuhvataju sekvenciranje velikog broja uzoraka dodatno povećavaju količinu podataka koju je potrebno obraditi. Zbog toga su za analizu genomskih sekvenci potrebni algoritmi koji mogu efikasno da obrađuju veoma duge nizove.

Jedan od osnovnih zadataka u analizi sekvenci jeste pronalaženje ponavljanja (engl. *repeats*). Ponovljeni delovi DNK čine značajan deo mnogih genoma i nastaju različitim mehanizmima, među kojima su duplikacije. Njihov značaj ogleda se i u biološkom i u računarskom smislu. Sa biološkog stanovišta, ponavljanja su povezana sa evolucijom i promenama u organizaciji genoma. Sa računarskog stanovišta, repetitivni regioni predstavljaju jedan od glavnih izazova pri sklapanju genoma iz kratkih očitavanja [14].

Broj različitih ponovljenih podniski jedne sekvence može biti kvadratan u odnosu na njenu dužinu, pa njihovo nabrajanje nije praktično. Zbog toga se razmatraju sažetije klase ponavljanja. Među njima su *maksimalna ponavljanja* (engl. *maximal repeats*), koja se ne mogu produžiti ni ulevo ni udesno, i *supermaksimalna ponavljanja* (engl. *supermaximal repeats*), koja nisu sadržana ni u jednom drugom dužem ponavljanju [3]. Njihov broj je znatno manji, a informacija koju nose dovoljna je za mnoge primene.

1.2 Indeksne strukture nad niskama

Efikasno pronalaženje ponavljanja zahteva indeksnu strukturu nad sekvencom. Klasičan izbor je sufiksno stablo (engl. *suffix tree*), za koje postoje algoritmi konstrukcije u linearnom vremenu [9, 15, 16]. Iako je njegova prostorna složenost linearna, sufiksno stablo u praksi zauzima veliku količinu memorije zbog visokog konstantnog faktora. Na ovoj strukturi zasnovani su i neki od ranih alata za analizu ponavljanja u genomskim sekvencama, kao što je REPuter [6, 7].

Sufiksni niz (engl. *suffix array*), koji su uveli Manber i Majers, čuva permutaciju početnih pozicija sufiksa i zahteva znatno manje memorije [8]. Međutim, ne sadrži sve informacije o strukturi grananja sufiksa. Poboljšani sufiksni niz (engl. *enhanced suffix array*, ESA) dopunjuje osnovni sufiksni niz nizom najdužih zajedničkih prefiksa (engl. *longest common prefix*, LCP) i dodatnim tabelama, čime omogućava efikasno izvođenje operacija koje se inače zasnivaju na sufiksnom stablu [1]. Važan deo ove strukture predstavljaju LCP intervali (engl. *LCP intervals*), koji odgovaraju unutrašnjim čvorovima sufiksnog stabla.

Ovakva reprezentacija je pogodna i za pronalaženje ponavljanja. LCP intervali grupišu sufikse sa zajedničkim prefiksom i daju informacije potrebne za određivanje desnih konteksta (engl. *right contexts*) ponavljanja, dok se levi konteksti (engl. *left contexts*) mogu dobiti iz odgovarajućih delova Barouz–Vilerove transformacije [2]. Kombinovanjem ovih informacija moguće je efikasno odrediti maksimalna i supermaksimalna ponavljanja bez eksplicitnog razmatranja svih parova njihovih pojavljivanja.

1.3 Predmet, cilj i sadržaj rada

Predmet ovog rada je implementacija strukture poboljšanog sufiksnog niza i njena primena na pronalaženje maksimalnih i supermaksimalnih egzaktnih ponavljanja u DNK sekvencama. Cilj rada je dvostruk. Prvi cilj je da se razmatrane strukture i algoritmi implementiraju od nule, bez korišćenja gotovih biblioteka, i da se njihova ispravnost potvrdi automatskim testovima. Drugi cilj je da se kroz kontrolisane eksperimente ispita koliko izbor algoritma utiče na performanse na sintetičkim i realnim genomskim podacima. Cilj nije razvoj konkurentskog softverskog alata za analizu ponavljanja, već povezivanje teorijskih osobina razmatranih struktura sa ponašanjem njihove konkretne implementacije.

Implementacija je napisana u jeziku C++17 i obuhvata tri algoritma za konstrukciju sufiksnog niza: osnovno udvostručavanje prefiksa, optimizovano udvostručavanje prefiksa i algoritam SA-IS (engl. *Suffix Array Induced Sorting*) [10]. Na osnovu sufiksnog niza grade se inverzni sufiksni niz, niz najdužih zajedničkih prefiksa, Barouz–Vilerova transformacija i stablo LCP intervala. Za detekciju maksimalnih ponavljanja implementirana su dva pristupa: osnovni, koji eksplicitno nabraja maksimalne ponovljene parove, i optimizovani, koji prolazi kroz stablo LCP intervala odozdo naviše i koristi bitovske maske levih konteksta. Supermaksimalna ponavljanja traže se proverom dva uslova nad listovima tog stabla.

Opšti princip optimizovanog pristupa, obrada LCP strukture uz korišćenje levih konteksta i izbegavanje eksplicitnog nabrajanja parova, saglasan je sa idejama iz rada o kontekstno osetljivim ponavljanjima u linearnom vremenu [12]. Implementacija, međutim, nije doslovno preslikavanje pseudokoda iz tog rada, već koristi sopstvenu reprezentaciju konteksta za fiksnu bajtovsku azbuku.

Eksperimentalni deo obuhvata dva nezavisna poređenja. U prvom se menja samo algoritam za konstrukciju sufiksnog niza, dok algoritam za pronalaženje ponavljanja ostaje isti, čime se posebno analizira vreme konstrukcije sufiksnog niza. U drugom se porede dva algoritma za pronalaženje ponavljanja nad istim sufiksnim nizom, čime se analizira vreme detekcije ponavljanja. Merenja su izvedena na sintetičkim sekvencama dužina od 10 000 do 500 000 baza i na četiri kompletna genoma, od virusnog sa 29 903 baze do bakterijskog sa 4 641 652 baze.

1.4 Organizacija rada

Glava 2 uvodi teorijske osnove: niske i sufikse, sufiksni niz i prateće nizove, poboljšani sufiksni niz, LCP intervale i njihovo stablo, kao i definicije ponavljanja sa njihovom karakterizacijom preko poboljšanog sufiksnog niza. U glavi 3 opisana su tri algoritma za konstrukciju sufiksnog niza, dok je glava 4 posvećena algoritamskom postupku izgradnje pratećih nizova i stabla LCP intervala. U glavi 5 predstavljena su dva pristupa detekciji ponavljanja i obrazložena je njihova ekvivalentnost. Veza ovih algoritama sa modulima, strukturama i funkcijama programa opisana je u glavi 6, dok glava 7 sadrži analizu njihove vremenske i prostorne složenosti.

Glava 8 opisuje ulazni format, izlaz i arhitekturu programa, a glava 9 testove i način validacije. Eksperimentalna metodologija definisana je u glavi 10, rezultati su prikazani u glavi 11, dok glava 12 sadrži njihovo tumačenje i navodi ograničenja. Na kraju, glava 13 daje pregled najvažnijih zaključaka rada.

Glava 2

Teorijske osnove

Ova glava uvodi pojmove na kojima počiva ostatak rada. Redosled izlaganja prati redosled kojim se strukture grade u implementaciji: prvo niske i sufiksi, zatim sufiksni niz i prateći nizovi, pa struktura poboljšanog sufiksnog niza. Na kraju se uvode tipovi egzaktnih ponavljanja i njihova karakterizacija preko te strukture.

2.1 Niske, sufiksi i notacija

Neka je Σ konačan skup simbola, koji nazivamo azbukom, i neka je $\sigma = |\Sigma|$. Niska S dužine n je konačan niz simbola iz Σ.

U nastavku se koristi indeksiranje pozicija od nule, u skladu sa načinom indeksiranja u implementaciji. Simbol na poziciji i označavamo sa $S[i]$, a podnisku od pozicije i do pozicije j sa $S[i..j]$. Prefiks niske S je podniska oblika $S[0..j]$, a sufiks je podniska oblika $S[i..n-1]$, koju kraće označavamo sa S_i. Niska dužine n ima tačno n nepraznih sufiksa. Kažemo da se niska ω javlja u S na poziciji p ako je $\omega = S[p..p+|\omega|-1]$; skup svih takvih pozicija označavamo sa $\mathrm{pos}(\omega)$.

Poređenje niski je leksikografsko: $u < v$ ako je u pravi prefiks od v, ili ako na prvoj poziciji na kojoj se razlikuju niska u ima manji simbol. Ova konvencija je važna jer implementacija ne dodaje poseban završni simbol na kraj ulazne niske, pa se kraći sufiks koji je prefiks dužeg uvek smatra manjim.

Osnovna azbuka nukleotida je $\Sigma_{\mathrm{DNK}} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Implementacija dodatno prihvata simbol `N`, koji se u FASTA datotekama koristi za nepoznatu bazu; detalji FASTA formata dati su u odeljku 8.1.

2.2 Sufiksni niz i inverzni sufiksni niz

Sufiksni niz niske S dužine n je permutacija SA skupa $\{0, 1, \ldots, n-1\}$ takva da važi

$$S_{SA[0]} < S_{SA[1]} < \cdots < S_{SA[n-1]}.$$

Dakle, $SA[i]$ je početna pozicija i-tog sufiksa u leksikografskom poretku. Sufiksni niz su uveli Manber i Majers kao memorijski efikasniju alternativu sufiksnom stablu [8]. Sam sufiksni niz sastoji se od n celobrojnih vrednosti i u praksi zahteva znatno manje memorije od odgovarajućeg sufiksnog stabla. Pregled različitih algoritama za konstrukciju sufiksnog niza dostupan je u [13].

Inverzni sufiksni niz (engl. *inverse suffix array*, ISA) je inverzna permutacija sufiksnog niza, to jest $ISA[SA[i]] = i$ za svako i. Vrednost $ISA[p]$ je leksikografski rang sufiksa koji počinje na poziciji p. Inverzni sufiksni niz može se izračunati jednim prolazom kroz sufiksni niz, u vremenu $\mathcal{O}(n)$. U implementaciji se konstruiše jednom, nakon čega se koristi pri računanju niza najdužih zajedničkih prefiksa.

Primer 2.1 *Za nisku* $S =$ `ACAGACAT` *sufiksni niz i ostali nizovi koji se uvode u nastavku prikazani su u tabeli 2.1.*

Tabela 2.1: Poboljšani sufiksni niz niske $S =$ `ACAGACAT` dužine $n = 8$. Vrednosti su dobijene pokretanjem implementacije opisane u ovom radu.

i	$SA[i]$	$ISA[i]$	$LCP[i]$	$BWT[i]$	sufiks $S_{SA[i]}$
0	0	0	0	`$`	`ACAGACAT`
1	4	4	3	`G`	`ACAT`
2	2	2	1	`C`	`AGACAT`
3	6	6	1	`C`	`AT`
4	1	1	0	`A`	`CAGACAT`
5	5	5	2	`A`	`CAT`
6	3	3	0	`A`	`GACAT`
7	7	7	0	`A`	`T`

U prethodnom primeru sufiksni niz i inverzni sufiksni niz imaju iste vrednosti, što nije uvek slučaj. Sledeći primer ilustruje razliku između ova dva niza.

Primer 2.2 *Za nisku* $S =$ `ACAC` *sufiksni niz je*

$$SA = [2, 0, 3, 1],$$

jer su sufiksi u leksikografskom poretku $S_2 =$ `AC`, $S_0 =$ `ACAC`, $S_3 =$ `C` *i* $S_1 =$ `CAC`. *Odgovarajući inverzni sufiksni niz je*

$$ISA = [1, 3, 0, 2].$$

Na primer, $ISA[0] = 1$ *znači da se sufiks koji počinje na poziciji* 0 *nalazi na poziciji* 1 *u sufiksnom nizu.*

2.3 Niz najdužih zajedničkih prefiksa

Za dve niske u i v neka $\text{lcp}(u, v)$ označava dužinu njihovog najdužeg zajedničkog prefiksa. Niz najdužih zajedničkih prefiksa (engl. *longest common prefix array*, LCP) definiše se sa $LCP[0] = 0$ i

$$LCP[i] = \text{lcp}\big(S_{SA[i-1]},\, S_{SA[i]}\big)\,, \qquad 1 \leq i \leq n-1.$$

Vrednost $LCP[i]$ meri koliko simbola dele dva leksikografski susedna sufiksa. U primeru 2.1, $LCP[1] = 3$ označava da sufiksi `ACAGACAT` i `ACAT` imaju zajednički prefiks `ACA` dužine 3.

Naivno računanje niza najdužih zajedničkih prefiksa je kvadratne vremenske složenosti u funkciji dužine niske. Kasai i saradnici pokazali su da se, uz pomoć inverznog sufiksnog niza, niz može izračunati u vremenu $\mathcal{O}(n)$ [5]. Ključno zapažanje je da, ako sufiks S_p deli sa svojim leksikografskim prethodnikom prefiks dužine h, tada sufiks S_{p+1} deli sa svojim prethodnikom prefiks dužine bar $h-1$. Zato se pozicije obrađuju redosledom $p = 0, 1, \ldots, n-1$, a brojač zajedničkih simbola se između dva koraka smanjuje najviše za jedan, pa je ukupan broj poređenja linearan.

2.4 Barouz–Vilerova transformacija

Barouz–Vilerova transformacija (engl. *Burrows–Wheeler transform*, BWT) predstavlja reverzibilno preuređenje simbola niske, prvobitno uvedeno radi efikasnije kompresije podataka [2]. Intuitivno, u reprezentaciji zasnovanoj na sufiksnom nizu, BWT beleži *leve kontekste* leksikografski poređanih sufiksa, odnosno simbole koji neposredno prethode njihovim početnim pozicijama u originalnoj niski. Kada je sufiksni niz već izračunat, BWT se dobija u jednom prolazu:

$$BWT[i] = \begin{cases} S[SA[i]-1], & SA[i] > 0, \\ \$, & SA[i] = 0. \end{cases}$$

Simbol $BWT[i]$ predstavlja simbol koji u originalnoj sekvenci neposredno prethodi i-tom sufiksu, to jest levi kontekst tog pojavljivanja. Za sufiks koji počinje na poziciji nula levi kontekst ne postoji, pa implementacija upisuje poseban simbol `$`, koji se ne javlja u ulaznoj sekvenci. Time se takvo pojavljivanje razlikuje od svih ostalih.

Svi sufiksi kojima je neka niska ω prefiks zauzimaju uzastopan interval u sufiksnom nizu. Zbog toga se levi konteksti svih pojavljivanja niske ω nalaze u odgovarajućem uzastopnom segmentu BWT niza.

Veza između segmenta BWT niza i levih konteksta prikazana je u primeru 2.3.

Primer 2.3 *Za nisku* $S =$ `ACAGACAT` *iz primera 2.1, niska* $\omega =$ `ACA` *javlja se na pozicijama* 0 *i* 4. *Sufiksi koji počinju na tim pozicijama su* `ACAGACAT` *i* `ACAT` *i nalaze se na pozicijama* 0 *i* 1 *u sufiksnom nizu, odnosno čine interval* $[0..1]$. *Odgovarajući segment BWT niza je zato* $BWT[0..1] =$ `$G`. *Simbol* `$` *označava da prvo pojavljivanje počinje na početku sekvence, dok simbol* `G` *neposredno prethodi drugom pojavljivanju. Prema tome, skup levih konteksta niske* ω *je* $\{\$, \texttt{G}\}$.

Ovo svojstvo omogućava efikasnu analizu levih konteksta i koristi se pri pronalaženju maksimalnih i supermaksimalnih ponavljanja.

2.5 Poboljšani sufiksni niz

Poboljšani sufiksni niz predstavlja sufiksni niz dopunjen dodatnim strukturama koje omogućavaju efikasno izvođenje operacija tradicionalno zasnovanih na sufiksnom stablu. Abuelhoda, Kurc i Olebuš pokazali su da se svaki algoritam nad sufiksnim stablom može prevesti u algoritam nad sufiksnim nizom i nizom najdužih zajedničkih prefiksa, uz istu asimptotsku složenost i manju potrošnju memorije [1]. Sistematičan prikaz ovih struktura dat je i u monografiji [11].

Implementacija opisana u ovom radu čuva ulaznu nisku, sufiksni niz, inverzni sufiksni niz, niz najdužih zajedničkih prefiksa i Barouz–Vilerovu transformaciju. Pored njih, na osnovu LCP niza se konstruišu LCP intervali i njihova hijerarhijska struktura, koja će biti opisana u narednom odeljku.

Svaki od navedenih nizova i struktura ima određenu ulogu pri pronalaženju ponavljanja. Sufiksni niz određuje leksikografski poredak sufiksa, dok LCP niz i LCP intervali omogućavaju grupisanje sufiksa koji imaju zajedničke prefikse i analizu njihovih desnih ekstenzija. Barouz–Vilerova transformacija daje informacije o simboli-

ma koji prethode tim sufiksima, odnosno o levim kontekstima njihovih pojavljivanja. Kombinovanjem ovih informacija mogu se okarakterisati maksimalna i supermaksimalna ponavljanja.

2.6 LCP intervali i stablo LCP intervala

Struktura grananja sufiksnog stabla se u ESA reprezentaciji izražava pojmom LCP intervala. Naredna definicija preuzeta je iz rada Abuelhode i saradnika [1] i prilagođena indeksiranju od nule.

Definicija 2.1 *Neka je* $0 \leq i < j \leq n-1$. *Interval* $[i..j]$ *je* ℓ-interval *ako važi: (1)* $LCP[i] < \ell$ *ili* $i = 0$*; (2)* $LCP[k] \geq \ell$ *za svako* k *sa* $i < k \leq j$*; (3)* $LCP[k] = \ell$ *za bar jedno takvo* k*; (4)* $LCP[j+1] < \ell$ *ili* $j = n-1$. *Broj* ℓ *je LCP vrednost intervala, a niska* $\omega = S[SA[i]\,..\,SA[i]+\ell-1]$ *je niska pridružena tom intervalu.*

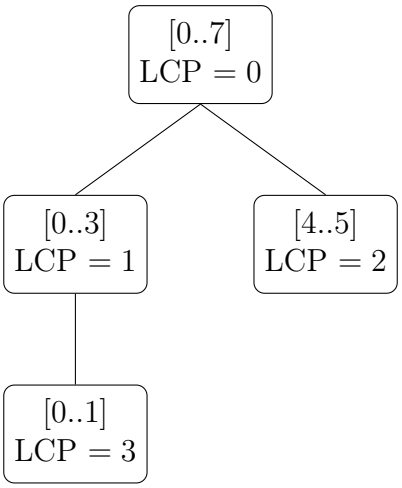
Uslovi (1) i (4) kažu da se interval ne može proširiti bez gubitka zajedničkog prefiksa dužine ℓ. Uslov (2) kaže da svi sufiksi u intervalu dele prefiks ω, a uslov (3) da se interval zaista grana na dubini ℓ. Skup pozicija na kojima se ω javlja je tada $\text{pos}(\omega) = \{\, SA[k] : i \leq k \leq j \,\}$, pa je nabrajanje svih pojavljivanja svedeno na čitanje uzastopnog dela sufiksnog niza. ℓ-intervali odgovaraju unutrašnjim čvorovima sufiksnog stabla, a pojedinačne pozicije $[i..i]$ odgovaraju listovima [1].

Konkretan LCP interval prikazan je u primeru 2.4.

Primer 2.4 *Za nisku* $S =$ `ACAGACAT` *iz primera 2.1, interval* $[0..1]$ *je* 3*-interval. Zaista, sufiksi na tim pozicijama u sufiksnom nizu su* `ACAGACAT` *i* `ACAT`*, a njihov najduži zajednički prefiks je* `ACA` *dužine* 3*. Zato je ovom intervalu pridružena niska* $\omega =$ `ACA`*. Interval se ne može proširiti na poziciju* 2*, jer je* $LCP[2] = 1 < 3$*. Slično, interval* $[0..3]$ *je* 1*-interval kojem je pridružena niska* `A`*, dok je* $[4..5]$ 2*-interval kojem je pridružena niska* `CA`*.*

Interval $[k..l]$ je *ugnežden* u interval $[i..j]$ ako je $i \leq k \leq l \leq j$ i ako je LCP vrednost unutrašnjeg intervala strogo veća. Ako pritom ne postoji treći interval koji leži između njih, kažemo da je $[k..l]$ dete intervala $[i..j]$. Relacija roditelj–dete definiše *stablo LCP intervala* (engl. *LCP-interval tree*), čiji je koren interval $[0..n-1]$ sa LCP vrednošću nula. Stablo LCP intervala predstavlja konceptualnu strukturu koja odgovara unutrašnjim čvorovima sufiksnog stabla, bez potrebe za eksplicitnim čuvanjem pokazivača između čvorova.

Odnosi između LCP intervala iz primera 2.4 mogu se predstaviti stablom prikazanim na slici 2.1. Njegova struktura objašnjena je u primeru 2.5.



Slika 2.1: Stablo LCP intervala za nisku $S =$ `ACAGACAT`.

Primer 2.5 *U stablu sa slike 2.1 korenski interval* [0..7] *obuhvata sve sufikse i ima LCP vrednost* 0*, jer nemaju svi sufiksi zajednički početni simbol. Interval* [0..3] *predstavlja grupu sufiksa koji počinju slovom* `A`*, pa mu je LCP vrednost* 1*. Unutar njega se nalazi uži interval* [0..1]*, koji predstavlja sufikse sa zajedničkim prefiksom* `ACA` *dužine* 3*, pa je zato njegovo dete. Sa druge strane, interval* [4..5] *predstavlja sufikse sa zajedničkim prefiksom* `CA` *dužine* 2 *i neposredno je dete korena. Na taj način, spuštanje kroz stablo odgovara prelasku na sve uže grupe sufiksa sa sve dužim zajedničkim prefiksima.*

Za algoritme za pronalaženje ponavljanja korisna je još jedna podela. Neka interval $[i..j]$ ima decu $[k_1..l_1], \ldots, [k_m..l_m]$. Tada se $[i..j]$ raspada na disjunktne *grupe*: svako dete čini jednu grupu, a svaka pozicija koja ne pripada nijednom detetu formira zasebnu grupu. Dva pojavljivanja iz iste grupe dele zajedničko produženje udesno duže od ω, a dva pojavljivanja iz različitih grupa ga nemaju. Ova podela je direktna posledica definicije ℓ-intervala i biće korišćena u glavi 5.

2.7 Egzaktna ponavljanja

Sledeće definicije prate klasičnu formulaciju iz literature o algoritmima nad niskama [1, 3].

Definicija 2.2 Ponovljeni par *(engl.* repeated pair*) u niski* S *je trojka* (p_1, p_2, ℓ) *takva da je* $p_1 \neq p_2$ *i* $S[p_1..p_1+\ell-1] = S[p_2..p_2+\ell-1]$. *Par je* levo maksimalan *(engl.* left maximal*) ako je* $p_1 = 0$ *ili* $p_2 = 0$ *ili* $S[p_1 - 1] \neq S[p_2 - 1]$, *a* desno maksimalan *(engl.* right maximal*) ako je* $p_1 + \ell = n$ *ili* $p_2 + \ell = n$ *ili* $S[p_1 + \ell] \neq S[p_2 + \ell]$. Maksimalni ponovljeni par *(engl.* maximal repeated pair*) je par koji je i levo i desno maksimalan.*

Definicija 2.3 *Niska* ω *je* maksimalno ponavljanje *(engl.* maximal repeat*) ako postoji maksimalni ponovljeni par* $(p_1, p_2, |\omega|)$ *takav da je* $\omega = S[p_1..p_1 + |\omega| - 1]$. *Maksimalno ponavljanje* ω *je* supermaksimalno *(engl.* supermaximal repeat*) ako* ω *nije podniska nijednog drugog maksimalnog ponavljanja.*

Maksimalna ponavljanja predstavljaju ponovljene podniske koje se ne mogu dalje produžiti uz očuvanje svih svojih pojavljivanja. Supermaksimalna ponavljanja čine strožu klasu maksimalnih ponavljanja, jer nisu sadržana ni u jednom drugom maksimalnom ponavljanju.

Odnos maksimalnih i supermaksimalnih ponavljanja ilustrovan je u primeru 2.6.

Primer 2.6 *Posmatrajmo nisku* $S =$ `ACAGACAT` *iz primera 2.1. Niska* `ACA` *javlja se na pozicijama* 0 *i* 4*; njeni levi konteksti su* `$` *i* `G`*, a desni* `G` *i* `T`*, pa je par* $(0, 4, 3)$ *maksimalni ponovljeni par. Niska* `A` *javlja se na pozicijama* 0, 2, 4 *i* 6 *i takođe je maksimalno ponavljanje, jer se par* $(0, 2, 1)$ *ne može produžiti ni ulevo ni udesno. Ipak,* `A` *jeste podniska niske* `ACA`*, pa nije supermaksimalna. Jedino supermaksimalno ponavljanje u ovoj niski je* `ACA`*.*

2.8 Karakterizacija ponavljanja preko ESA

Prethodne definicije govore o parovima pozicija, pa bi njihova direktna primena zahtevala ispitivanje svih parova pojavljivanja. Poboljšani sufiksni niz omogućava da se ista svojstva provere lokalno, po čvoru stabla LCP intervala.

Neka je $[i..j]$ ℓ-interval kome je pridružena niska ω. Desna maksimalnost je automatski zadovoljena: po definiciji ℓ-intervala postoje bar dva sufiksa koji se granaju na dubini ℓ, pa postoji par pojavljivanja sa različitim desnim kontekstima. Leva maksimalnost svodi se na sadržaj Barouz–Vilerove transformacije. Naredna dva tvrđenja preuzeta su iz rada Abuelhode i saradnika [1] i predstavljaju osnovu optimizovanih algoritama opisanih u glavi 5.

Teorema 2.1 *Niska ω pridružena ℓ-intervalu $[i..j]$ je maksimalno ponavljanje ako i samo ako skup $\{ BWT[k] : i \leq k \leq j \}$ sadrži bar dva različita simbola.*

Uslov iz teoreme 2.1 u literaturi se naziva *različitost levih konteksta* (engl. *left diversity*) odgovarajućeg čvora [3].

Teorema 2.2 *Niska ω pridružena ℓ-intervalu $[i..j]$ je supermaksimalno ponavljanje ako i samo ako je $[i..j]$ list stabla LCP intervala, to jest u njemu nije ugnežden nijedan drugi LCP interval, i ako su simboli $BWT[i], \ldots, BWT[j]$ međusobno različiti.*

Intuicija iza teoreme 2.2 je sledeća. Prvi uslov isključuje postojanje dužeg zajedničkog produženja udesno, jer bi takvo produženje formiralo ugnežden interval. Drugi uslov isključuje zajedničko produženje ulevo za bilo koja dva pojavljivanja. Ako oba važe, nijedno duže ponavljanje ne sadrži ω.

Obe teoreme daju provere koje zavise samo od jednog intervala i od uzastopnog dela niza BWT. Zato se cela klasifikacija može obaviti jednim prolazom kroz stablo LCP intervala, bez nabrajanja parova pojavljivanja. Upravo je to razlika između osnovnog i optimizovanog algoritma iz glave 5.

Glava 3

Algoritmi za konstrukciju sufiksnog niza

Konstrukcija sufiksnog niza predstavlja jedan od vremenski najzahtevnijih koraka u izgradnji poboljšanog sufiksnog niza, pa izbor algoritma za njegovu konstrukciju može značajno uticati na ukupno vreme izvršavanja. U ovoj glavi opisana su tri algoritma: osnovno udvostručavanje prefiksa, optimizovano udvostručavanje prefiksa i SA-IS. Prva dva algoritma koriste istu ideju postepenog određivanja poretka sve dužih prefiksa, dok SA-IS koristi indukovano sortiranje.

3.1 Osnovno udvostručavanje prefiksa

Klasični algoritam udvostručavanja prefiksa (engl. *prefix doubling*) sortira sufikse u više rundi po sve dužim prefiksima, pri čemu se dužina posmatranog prefiksa u svakoj rundi udvostručava.

Na početku se svakom sufiksu dodeljuje klasa ekvivalencije jednaka numeričkoj vrednosti njegovog prvog simbola. U rundi sa parametrom k sufiksi se porede po paru $(c[i], c[i+k])$, gde je c tekući niz klasa, a nepostojeća druga komponenta se tretira kao -1. Na taj način kraći sufiks se pravilno uređuje ispred dužeg sufiksa kada je kraći sufiks njegov prefiks. Sortiranje se obavlja algoritmom sortiranja poređenjem nad tim parovima, posle čega se vrši ponovno dodeljivanje klasa. Postupak se prekida čim sve klase postanu različite.

Broj rundi je $\mathcal{O}(\log n)$, a svaka runda ima složenost $\mathcal{O}(n \log n)$ zbog potrebe za sortiranjem, pa je ukupna složenost $\mathcal{O}(n \log^2 n)$. Zbog jednostavne strukture ovaj algoritam predstavlja prirodnu polaznu tačku za poređenje sa složenijim postupcima.

Primer 3.1 *Za nisku* $S =$ `ACAGACAT` *iz primera 2.1 u nastavku je prikazana konstrukcija sufiksnog niza osnovnim algoritmom udvostručavanja prefiksa. Ista niska koristi se i u primerima naredna dva algoritma radi lakšeg poređenja njihovih postupaka. Na početku se svakom sufiksu dodeljuje klasa prema njegovom prvom simbolu. Radi preglednosti, simbolima* `A`, `C`, `G` *i* `T` *mogu se pridružiti redom klase* 0, 1, 2 *i* 3, *jer je za algoritam važan njihov međusobni poredak. Početni niz klasa je zato*

$$c^{(0)} = [0, 1, 0, 2, 0, 1, 0, 3].$$

U prvoj rundi je $k = 1$, *pa se sufiksi porede prema parovima* $(c^{(0)}[i], c^{(0)}[i+1])$. *Dobijaju se parovi*

i	*početak sufiksa*	$(c^{(0)}[i], c^{(0)}[i+1])$
0	`AC`	$(0, 1)$
1	`CA`	$(1, 0)$
2	`AG`	$(0, 2)$
3	`GA`	$(2, 0)$
4	`AC`	$(0, 1)$
5	`CA`	$(1, 0)$
6	`AT`	$(0, 3)$
7	`T`	$(3, -1)$

Za sufiks koji počinje na poslednjoj poziciji druga komponenta ne postoji, pa se predstavlja vrednošću -1. *Sortiranjem navedenih parova dobija se redosled*

$$SA = [0, 4, 2, 6, 1, 5, 3, 7].$$

Pozicije 0 *i* 4 *pripadaju istoj klasi, jer obe odgovaraju prefiksu* `AC`, *dok pozicije* 1 *i* 5 *pripadaju istoj klasi zbog prefiksa* `CA`. *Nakon ponovnog dodeljivanja klasa dobija se*

$$c^{(1)} = [0, 3, 1, 4, 0, 3, 2, 5].$$

Pošto sve klase još nisu različite, algoritam prelazi na narednu rundu.

U drugoj rundi je $k = 2$. *Klase iz prethodne runde opisuju poredak prefiksa dužine do dva, pa se poređenjem parova* $(c^{(1)}[i], c^{(1)}[i+2])$ *određuje poredak prefiksa dužine do četiri. Dobijaju se parovi*

i	*početak sufiksa*	$(c^{(1)}[i], c^{(1)}[i+2])$
0	`ACAG`	$(0, 1)$
1	`CAGA`	$(3, 4)$
2	`AGAC`	$(1, 0)$
3	`GACA`	$(4, 3)$
4	`ACAT`	$(0, 2)$
5	`CAT`	$(3, 5)$
6	`AT`	$(2, -1)$
7	`T`	$(5, -1)$

U ovoj rundi razlikuju se i sufiksi koji su nakon prve runde pripadali istim klasama. Na primer, za pozicije 0 *i* 4 *važi* $(0, 1) < (0, 2)$*, pa sufiks* `ACAGACAT` *prethodi sufiksu* `ACAT`*. Slično, parovi* $(3, 4)$ *i* $(3, 5)$ *određuju poredak sufiksa* `CAGACAT` *i* `CAT`*.*

Ponovnim sortiranjem dobija se

$$SA = [0, 4, 2, 6, 1, 5, 3, 7],$$

dok su nove klase

$$c^{(2)} = [0, 4, 2, 6, 1, 5, 3, 7].$$

Pošto su sada sve klase različite, postupak se završava. Konačni sufiksni niz je zato

$$SA = [0, 4, 2, 6, 1, 5, 3, 7],$$

što odgovara rezultatu prikazanom u tabeli 2.1.

3.2 Optimizovano udvostručavanje prefiksa

Optimizovana varijanta zadržava istu ideju, ali uklanja opšte sortiranje poređenjem. Početno sortiranje po prvom simbolu obavlja se sortiranjem prebrojavanjem (engl. *counting sort*) nad azbukom od σ simbola. Zatim se u svakoj rundi koristi sortiranje po dva ključa, izvedeno kao stabilno sortiranje prebrojavanjem.

Jedna runda algoritma odvija se na sledeći način. Najpre se formira pomoćni niz pozicija koji treba da bude uređen prema drugoj komponenti para. Na početak tog niza upisuju se pozicije $n - k, \ldots, n - 1$, za koje druga komponenta ne postoji.

Zatim se, redom kojim se pozicije pojavljuju u tekućem sufiksnom nizu, u pomoćni niz dodaju vrednosti $SA[i] - k$ za sve i za koje važi $SA[i] \geq k$. Pošto je tekući sufiksni niz već uređen prema klasama iz prethodne runde, na ovaj način dobijeni pomoćni niz uređen je prema drugoj komponenti para. Zatim se nad njim obavlja stabilno sortiranje prebrojavanjem po prvoj komponenti. Broj klasa je najviše n, pa je jedna runda linearne složenosti. Na kraju runde klase se ponovo dodeljuju poređenjem parova koji odgovaraju susednim sufiksima u novom poretku.

Broj rundi ostaje $\mathcal{O}(\log n)$, ali je svaka runda sada složenosti $\mathcal{O}(n)$, pa je ukupna složenost $\mathcal{O}(n \log n)$. Postupak se prekida i ranije, čim broj klasa dostigne n. Izbegavanjem opšteg sortiranja u svakoj rundi ova varijanta smanjuje asimptotsku vremensku složenost u odnosu na osnovni algoritam.

Primer 3.2 *Posmatrajmo ponovo nisku* $S =$ `ACAGACAT`. *Za razliku od osnovnog algoritma, početno uređivanje sufiksa prema prvom simbolu obavlja se sortiranjem prebrojavanjem. Dobija se početni poredak*

$$SA^{(0)} = [0, 2, 4, 6, 1, 5, 3, 7],$$

jer sufiksi koji počinju simbolom `A` *prethode sufiksima koji počinju simbolima* `C`, `G` *i* `T`. *Odgovarajući niz klasa je*

$$c^{(0)} = [0, 1, 0, 2, 0, 1, 0, 3].$$

U prvoj rundi je $k = 1$. *Najpre se formira pomoćni niz pozicija uređen prema drugoj komponenti para klasa. Na njegov početak postavlja se pozicija* 7, *za koju druga komponenta ne postoji. Zatim se, redosledom iz tekućeg sufiksnog niza, dodaju vrednosti* $SA^{(0)}[i] - 1$ *za sve pozicije za koje je ova vrednost definisana. Dobija se*

$$P = [7, 1, 3, 5, 0, 4, 2, 6].$$

Pošto je pomoćni niz već uređen prema drugoj komponenti, nad njim se obavlja stabilno sortiranje prebrojavanjem prema prvoj komponenti, odnosno prema vrednosti $c^{(0)}[i]$. *Time se dobija*

$$SA^{(1)} = [0, 4, 2, 6, 1, 5, 3, 7].$$

Ponovnim dodeljivanjem klasa dobija se

$$c^{(1)} = [0, 3, 1, 4, 0, 3, 2, 5].$$

Pozicije 0 *i* 4 *i dalje pripadaju istoj klasi, kao i pozicije* 1 *i* 5*, pa je potrebna još jedna runda.*

U drugoj rundi je $k = 2$*. Na početak pomoćnog niza sada se postavljaju pozicije* 6 *i* 7*, za koje druga komponenta para ne postoji. Zatim se na isti način dodaju vrednosti* $SA^{(1)}[i] - 2$*, čime se dobija*

$$P = [6, 7, 2, 0, 4, 3, 1, 5].$$

Stabilnim sortiranjem ovog niza prema prvoj komponenti para dobija se

$$SA^{(2)} = [0, 4, 2, 6, 1, 5, 3, 7].$$

Nakon ponovnog dodeljivanja klasa važi

$$c^{(2)} = [0, 4, 2, 6, 1, 5, 3, 7].$$

Sve klase su sada različite, pa se postupak završava. Konačni sufiksni niz je

$$SA = [0, 4, 2, 6, 1, 5, 3, 7].$$

Dobijeni rezultat jednak je rezultatu osnovnog algoritma, ali se u svakoj rundi umesto opšteg sortiranja poređenjem koristi linearno sortiranje prebrojavanjem.

3.3 Algoritam SA-IS

Algoritam SA-IS, zasnovan na indukovanom sortiranju (engl. *induced sorting*), predložili su Nong, Džang i Čen [10]. Algoritam gradi sufiksni niz u vremenu $\mathcal{O}(n)$ za azbuku čiji su simboli predstavljeni celobrojnim vrednostima koje određuju njihov poredak. Njegova osnovna ideja je da se sortira samo mali podskup sufiksa, a da se poredak svih ostalih *indukuje* iz tog podskupa. SA-IS nije jedini linearan algoritam; među ranijima je i algoritam Kerkejnena i Sandersa [4].

Tipovi pozicija i LMS pozicije

Pozicija i je tipa S ako je $S_i < S_{i+1}$, a tipa L ako je $S_i > S_{i+1}$; poslednja pozicija je po konvenciji tipa S, a ako je $S[i] = S[i+1]$, pozicija nasleđuje tip pozicije $i+1$. Ovo pravilo omogućava da se tipovi izračunaju jednim prolazom zdesna nalevo.

Pozicija $i > 0$ je *LMS pozicija* (engl. *leftmost S-type*) ako je tipa S, a pozicija $i - 1$ je tipa L. LMS pozicije dele nisku na preklapajuće segmente koje nazivamo LMS podniskama (engl. *LMS substrings*); svaka počinje na jednoj LMS poziciji i završava se na sledećoj. Broj LMS pozicija je najviše $n/2$, jer dve susedne pozicije ne mogu obe biti LMS pozicije. Zbog toga je dužina redukovane niske na svakom rekurzivnom nivou najviše polovina dužine prethodne niske.

Kofe i indukovano sortiranje

Sufiksi se grupišu u *kofe* (engl. *buckets*) prema prvom simbolu. Veličine kofa određuju njihove početne i završne pozicije. Indukovano sortiranje izvodi tri prolaza:

1. dati LMS sufiksi se, obrnutim redosledom, upisuju na krajeve svojih kofa;
2. prolaskom sleva nadesno, za svaku popunjenu poziciju p proverava se pozicija $p - 1$; ako je tipa L, upisuje se na početak svoje kofe, čime se sortiraju svi sufiksi tipa L;
3. prolaskom zdesna nalevo isti postupak se ponavlja za pozicije tipa S, koje se upisuju na kraj svoje kofe.

Svaki od ovih prolaza se izvršava u linearnoj vremenskoj složenosti, a pomoćne informacije o kofama čuvaju se u nizovima čija je veličina određena veličinom azbuke. Ako su ulazni LMS sufiksi dati u tačnom leksikografskom poretku, rezultat je potpuno tačan sufiksni niz.

Imenovanje LMS podniski, rekurzija i završno indukovanje

Početnim indukovanim sortiranjem određuje se relativni leksikografski poredak LMS podniski [10]. Zatim se prolazi kroz dobijeni poredak i svakoj LMS podniski dodeljuje ime, to jest ceo broj. Dve uzastopne LMS podniske dobijaju isto ime ako su jednake, što se utvrđuje poređenjem odgovarajućih simbola i LMS granica obe podniske. Imena se nižu po redosledu LMS pozicija u originalnoj niski, čime nastaje *redukovana niska* (engl. *reduced string*) dužine najviše $n/2$.

Ako su sva imena različita, leksikografski poredak sufiksa redukovane niske može se odrediti direktno, bez rekurzivnog poziva. U tom slučaju svako ime jednoznačno određuje rang odgovarajuće LMS pozicije, pa se sufiksni niz redukovane niske formira jednim prolazom kroz njene elemente. U suprotnom se algoritam rekurzivno poziva

nad redukovanom niskom. Rezultat rekurzije određuje poredak LMS sufiksa. Taj poredak se zatim preslikava na odgovarajuće pozicije u originalnoj niski, nakon čega sledi drugo indukovano sortiranje. Ono proizvodi konačan sufiksni niz.

Pošto se veličina problema na svakom nivou rekurzije bar polovi, ukupna vremenska složenost algoritma ostaje linearna, odnosno iznosi $\mathcal{O}(n)$ [10].

Završni simbol

Algoritam SA-IS zahteva jedinstven završni simbol koji je leksikografski manji od svih simbola ulazne niske. Zato se simboli ulazne niske preslikavaju u pozitivne cele brojeve, a na njen kraj dodaje se vrednost 0 kao završni simbol (engl. *sentinel*). Pošto se vrednost 0 ne javlja među preslikanim simbolima, završni simbol je jedinstven i leksikografski najmanji. Sufiks koji sadrži samo završni simbol zato zauzima prvo mesto u sufiksnom nizu. Po završetku algoritma taj sufiks se uklanja, pa sufiksni niz originalne niske sadrži tačno n elemenata.

Primer 3.3 *Posmatrajmo ponovo nisku* $S =$ `ACAGACAT`. *SA-IS najpre dodaje jedinstven završni simbol, pa se algoritam primenjuje na nisku*

$$S' = \texttt{ACAGACAT\$},$$

pri čemu je završni simbol leksikografski manji od svih ostalih simbola.

Prvi korak je određivanje S i L tipova pozicija. Polazeći zdesna nalevo, dobija se

i	0	1	2	3	4	5	6	7	8
$S'[i]$	A	C	A	G	A	C	A	T	$
tip	*S*	*L*	*S*	*L*	*S*	*L*	*S*	*L*	*S*

LMS pozicije su S pozicije kojima neposredno prethodi L pozicija, pa su u ovom primeru

$$[2, 4, 6, 8].$$

Sufiksi se zatim raspoređuju u kofe prema prvom simbolu. Veličine kofa su

simbol	$	A	C	G	T
veličina kofe	1	4	2	1	1

pa se njihove pozicije u sufiksnom nizu nalaze u intervalima $[0]$, $[1, 4]$, $[5, 6]$, $[7]$ *i* $[8]$.

U prvom indukovanom sortiranju LMS pozicije se, obrnutim redosledom, postavljaju na krajeve odgovarajućih kofa. Nakon ovog koraka dobija se

$$SA = [8, -1, 2, 4, 6, -1, -1, -1, -1],$$

gde vrednost -1 *označava još nepopunjenu poziciju.*

Zatim se niz obilazi sleva nadesno i indukuju se sufiksi tipa L. Za svaku već upisanu poziciju p *posmatra se pozicija* $p - 1$ *i, ako je ona tipa L, upisuje se na početak odgovarajuće kofe. Posle ovog prolaza dobija se*

$$SA = [8, -1, 2, 4, 6, 1, 5, 3, 7].$$

Nakon toga se niz obilazi zdesna nalevo i na isti način indukuju se sufiksi tipa S, koji se postavljaju na krajeve svojih kofa. Rezultat prvog indukovanog sortiranja je

$$SA = [8, 0, 4, 2, 6, 1, 5, 3, 7].$$

Iz ovog poretka izdvajaju se LMS sufiksi. Njihov leksikografski poredak je

$$8,\ 4,\ 2,\ 6.$$

Odgovarajuće LMS podniske su

LMS pozicija	*LMS podniska*
2	`AGA`
4	`ACA`
6	`AT$`
8	$

i sve su međusobno različite. Prema njihovom leksikografskom poretku dodeljuju im se imena

$$8 \mapsto 0, \qquad 4 \mapsto 1, \qquad 2 \mapsto 2, \qquad 6 \mapsto 3.$$

Imena se zatim zapisuju redosledom LMS pozicija u originalnoj niski, odnosno za pozicije $2, 4, 6, 8$. *Time se dobija redukovana niska*

$$S_R = [2, 1, 3, 0].$$

Pošto su sva imena različita, nije potreban rekurzivni poziv. Sufiksni niz redukovane niske može se odrediti direktno i iznosi

$$SA_R = [3, 1, 0, 2].$$

Preslikavanjem ovih pozicija na LMS pozicije originalne niske dobija se njihov tačan leksikografski poredak

$$[8, 4, 2, 6].$$

Sa tako uređenim LMS pozicijama izvršava se završno indukovano sortiranje. Nakon postavljanja LMS sufiksa na krajeve njihovih kofa, indukovanja L sufiksa sleva nadesno i zatim S sufiksa zdesna nalevo dobija se

$$SA = [8, 0, 4, 2, 6, 1, 5, 3, 7].$$

Prvi element predstavlja sufiks koji sadrži samo završni simbol. Njegovim uklanjanjem dobija se sufiksni niz originalne niske

$$SA = [0, 4, 2, 6, 1, 5, 3, 7].$$

Dobijeni rezultat jednak je rezultatima prethodna dva algoritma.

3.4 Poređenje algoritama

Osnovno udvostručavanje prefiksa ima složenost $\mathcal{O}(n \log^2 n)$ i predstavlja najjednostavniji od razmatranih postupaka. Optimizovana varijanta zamenjuje sortiranje poređenjem sortiranjem prebrojavanjem i smanjuje složenost na $\mathcal{O}(n \log n)$, uz zadržavanje iste osnovne ideje. SA-IS se suštinski razlikuje od njih, jer se zasniva na indukovanom sortiranju i ima linearnu teorijsku vremensku složenost. Njihovo eksperimentalno poređenje dato je u odeljku 11.4.

Glava 4

Izgradnja poboljšanog sufiksnog niza

U ovoj glavi opisan je postupak izgradnje poboljšanog sufiksnog niza i stabla LCP intervala. Polazi se od sufiksnog niza, a zatim se redom grade inverzni sufiksni niz, LCP niz i Barouz–Vilerova transformacija.

4.1 Postupak izgradnje poboljšanog sufiksnog niza

Prvi korak je konstrukcija sufiksnog niza jednim od algoritama opisanih u glavi 3. Inverzni sufiksni niz zatim se dobija jednim prolazom kroz sufiksni niz, primenom jednakosti

$$ISA[SA[i]] = i, \qquad 0 \leq i < n.$$

Treći korak je konstrukcija niza najdužih zajedničkih prefiksa algoritmom Kasaija i saradnika [5]. Algoritam koristi već izgrađeni inverzni sufiksni niz i svojstvo da se dužina poznatog zajedničkog prefiksa pri prelasku na narednu poziciju može smanjiti najviše za jedan, kao što je objašnjeno u odeljku 2.3. Poslednji korak je konstrukcija Barouz–Vilerove transformacije po formuli iz odeljka 2.4. Sva četiri prateća niza imaju dužinu n, a svaki korak nakon konstrukcije sufiksnog niza zahteva linearno vreme.

Primer 4.1 *Postupak izgradnje poboljšanog sufiksnog niza može se prikazati na niski* $S =$ `ACAC`. *Njen sufiksni niz je*

$$SA = [2, 0, 3, 1].$$

Inverzni sufiksni niz dobija se primenom jednakosti $ISA[SA[i]] = i$. *Pošto je*

$$SA[0] = 2, \qquad SA[1] = 0, \qquad SA[2] = 3, \qquad SA[3] = 1,$$

dobija se

$$ISA[2] = 0, \qquad ISA[0] = 1, \qquad ISA[3] = 2, \qquad ISA[1] = 3,$$

odnosno

$$ISA = [1, 3, 0, 2].$$

Niz najdužih zajedničkih prefiksa određuje se za susedne sufikse u leksikografskom poretku. U ovom primeru važi

$$LCP = [0, 2, 0, 1],$$

jer sufiksi `AC` *i* `ACAC` *imaju zajednički prefiks* `AC` *dužine* 2*, dok sufiksi* `C` *i* `CAC` *imaju zajednički prefiks* `C` *dužine* 1*.*

Na kraju se Barouz–Vilerova transformacija dobija uzimanjem simbola koji neposredno prethodi svakom sufiksu u originalnoj niski, dok se za sufiks koji počinje na poziciji 0 *koristi završni simbol* \$*. Tako se dobija*

$$BWT = \texttt{C\$AA}.$$

Prema tome, za nisku $S =$ `ACAC` *poboljšani sufiksni niz čine*

$$SA = [2, 0, 3, 1], \qquad ISA = [1, 3, 0, 2], \qquad LCP = [0, 2, 0, 1], \qquad BWT = \texttt{C\$AA}.$$

4.2 LCP intervali i stablo LCP intervala

LCP intervali mogu se izdvojiti jednim prolazom kroz LCP niz pomoću steka parova (LCP vrednost, leva granica). Kada se naiđe na manju LCP vrednost, sa steka se skidaju svi intervali koji su time zatvoreni. Svaki skinuti par određuje LCP vrednost i levu granicu intervala, dok je njegova desna granica pozicija neposredno pre tekuće.

Za izgradnju stabla svaki unos na steku, pored LCP vrednosti i leve granice, pamti i do tada zatvorenu decu. Na dnu steka nalazi se unos sa LCP vrednošću nula, koji odgovara korenu i ne skida se. Kada tekuća LCP vrednost padne ispod

vrednosti na vrhu steka, interval na vrhu se zatvara i pridružuje odgovarajućem roditelju. Ako se u istoj iteraciji otvara novi interval, prethodno zatvoreni interval postaje njegovo dete. Na kraju se dodaje koren sa granicama $[0..n-1]$. Pomoćna vrednost $LCP[n] = 0$ obezbeđuje da se pri završetku prolaza zatvore svi preostali intervali.

Poredak odozdo naviše

Ključna posledica opisanog postupka je da se čvor evidentira tek pošto su sva njegova deca već evidentirana. Ako se čvorovi čuvaju redosledom zatvaranja, za svaki čvor važi invarijanta

$$\text{indeks deteta} \;<\; \text{indeks roditelja},$$

pa su čvorovi uređeni odozdo naviše, pri čemu je koren poslednji. Ovaj poredak je važan za optimizovani algoritam pronalaženja maksimalnih ponavljanja iz odeljka 5.2: pri obradi roditelja već su dostupne informacije izračunate za svu njegovu decu, pa nisu potrebni rekurzija niti dodatni obilazak stabla.

Primer 4.2 *Za nisku* `ACGTACGTGATTACANN` *čvorovi stabla LCP intervala, uređeni odozdo naviše, prikazani su u tabeli 4.1.*

Tabela 4.1: Stablo LCP intervala niske `ACGTACGTGATTACANN`. Čvorovi su navedeni u poretku odozdo naviše.

ind.	LCP	interval	ω	deca	ind.	LCP	interval	ω	deca
0	4	[1..2]	`ACGT`	—	6	1	[8..10]	`G`	5
1	2	[0..2]	`AC`	0	7	1	[11..12]	`N`	—
2	1	[0..4]	`A`	1	8	3	[13..14]	`TAC`	—
3	3	[6..7]	`CGT`	—	9	1	[13..16]	`T`	8
4	1	[5..7]	`C`	3	10	0	[0..16]	—	2, 4, 6, 7, 9
5	2	[9..10]	`GT`	—					

Glava 5

Algoritmi za pronalaženje ponavljanja

Razmatraju se dva pristupa detekciji maksimalnih ponavljanja: osnovni, zasnovan na eksplicitnom nabrajanju maksimalnih ponovljenih parova, i optimizovani, zasnovan na karakterizaciji iz teoreme 2.1. Supermaksimalna ponavljanja određuju se na osnovu teoreme 2.2.

5.1 Osnovni pristup

Osnovni pristup direktno prati definiciju maksimalnog ponovljenog para iz odeljka 2.7.

Za svaki čvor stabla LCP intervala čija LCP vrednost zadovoljava zadatu minimalnu dužinu najpre se formiraju grupe opisane u odeljku 2.6. Svaka pozicija koja ne pripada nijednom detetu formira zasebnu grupu, dok sva pojavljivanja obuhvaćena istim detetom čine jednu grupu. Grupe predstavljaju klase desnog konteksta, pa se razmatraju samo parovi iz različitih grupa. Za svaki takav par proveravaju se levi konteksti, odnosno simboli koji prethode njegovim pojavljivanjima, pri čemu se za pojavljivanje na poziciji 0 koristi simbol `$`. Ako su levi konteksti različiti, par je maksimalan i dodaje se u rezultat.

Ako čvor ima q pojavljivanja, broj razmatranih parova pozicija iz različitih grupa u najgorem slučaju je najviše $\binom{q}{2}$, odnosno $\mathcal{O}(q^2)$. Za svaki takav par levi konteksti proveravaju se u konstantnom vremenu. Kada su oni različiti, par se prijavljuje kao maksimalni ponovljeni par. Ako se za svaki prihvaćeni par čuva i odgovarajuća niska, ukupno vreme zavisi i od broja prihvaćenih parova i od dužine tih niski.

Maksimalna ponavljanja dobijaju se iz maksimalnih ponovljenih parova u tri koraka. Najpre se određuju svi maksimalni ponovljeni parovi. Zatim se njihove niske grupišu, čime se zadržavaju samo različita maksimalna ponavljanja. Na kraju se za svako takvo ponavljanje formira puna lista pozicija pojavljivanja.

Ako se različite niske čuvaju u uređenom skupu, memorija zavisi od zbira njihovih dužina, a složenost svake pretrage je $\mathcal{O}(|\omega| \log R)$, gde je R broj različitih maksimalnih ponavljanja.

Pošto neposredno prati formalnu definiciju maksimalnog ponovljenog para, osnovni pristup predstavlja prirodnu polaznu tačku za poređenje sa optimizovanim algoritmom.

5.2 Optimizovana detekcija maksimalnih ponavljanja

Optimizovani pristup izbegava eksplicitno nabrajanje parova tako što maksimalnost proverava pomoću skupova levih konteksta, u skladu sa teoremom 2.1. Pošto je azbuka konačna, skup levih konteksta može se predstaviti bitovskom maskom: bit koji odgovara simbolu c postavljen je ako se c javlja kao levi kontekst bar jednog pojavljivanja. Unija skupova tada se svodi na uniju odgovarajućih maski.

Čvorovi se obrađuju odozdo naviše, u poretku iz odeljka 4.2. Za tekući čvor formiraju se iste grupe kao u osnovnom pristupu, ali se parovi ne nabrajaju. Održavaju se unija levih konteksta prethodno obrađenih grupa i skup levih konteksta celog tekućeg intervala. Za grupu sačinjenu od pojedinačne pozicije k skup sadrži samo $BWT[k]$, dok je za grupu određenu detetom već izračunat pri obradi tog deteta. Ako unija konteksta prethodnih grupa i tekuće grupe sadrži bar dva različita simbola, čvor zadovoljava uslov različitosti levih konteksta. Posle obrade svih grupa skup celog intervala postaje dostupan roditelju.

Ekvivalentnost sa definicijom

U pseudokodu 1 se uslov iz teoreme 2.1 proverava inkrementalno, pri dodavanju svake nove grupe, umesto da se provera vrši nakon formiranja maske celog intervala. Ova dva načina provere su ekvivalentna. Ako je tokom obrade neke grupe utvrđeno da unija njenih levih konteksta i konteksta prethodno obrađenih grupa sadrži bar dva različita simbola, onda ih sadrži i maska celog intervala. Obrnuto, ako maska

Algoritam 1 Optimizovana detekcija maksimalnih ponavljanja

U pseudokodu se sa M označava niz maski levih konteksta čvorova, sa A unija konteksta prethodno obrađenih grupa, sa C maska celog tekućeg intervala, a sa G maska tekuće grupe.

Ulaz: poboljšani sufiksni niz E, čvorovi V stabla LCP intervala u poretku odozdo naviše, minimalna dužina $\ell_{\min}$

Izlaz: lista maksimalnih ponavljanja

1: $M \leftarrow$ niz praznih maski dužine $|V|$; $R \leftarrow \emptyset$
2: **za svako** $v \in V$ redom **radi**
3: $A \leftarrow \emptyset$; $C \leftarrow \emptyset$; $maksimalno \leftarrow$ **netačno**
4: **za svako** grupa g čvora v (pojedinačne pozicije koje ne pripadaju deci i grupe određene decom) **radi**
5: **ako** g je pozicija k **onda**
6: $G \leftarrow \{\, E.BWT[k] \,\}$
7: **inače**
8: $G \leftarrow M[\,\text{indeks deteta}\,]$
9: **ako** $A \neq \emptyset$ **i** $G \neq \emptyset$ **i** $|A \cup G| \geq 2$ **onda**
10: $maksimalno \leftarrow$ **tačno**
11: $A \leftarrow A \cup G$; $C \leftarrow C \cup G$
12: $M[v] \leftarrow C$
13: **ako** $v.lcp \geq \max(1, \ell_{\min})$ **i** $maksimalno$ **onda**
14: dodaj u R nisku ω_v i sve pozicije $E.SA[k]$, $k \in [v.left..v.right]$
15: **vrati** R

celog intervala sadrži najmanje dva različita leva konteksta, onda će tokom obrade grupa u nekom trenutku biti dodat kontekst različit od svih prethodno viđenih. Tada unija maski sadrži najmanje dva različita simbola i provera uspeva.

Isti argument objašnjava zašto optimizovani i osnovni pristup prepoznaju iste maksimalne niske. Osnovni algoritam eksplicitno traži dve pozicije iz različitih grupa sa različitim levim kontekstima, dok optimizovani algoritam postojanje takvog para utvrđuje pomoću unije maski levih konteksta.

Razlike u odnosu na osnovni pristup

Optimizovani algoritam ne konstruiše maksimalne ponovljene parove, čime se izbegava eksplicitno kvadratno nabrajanje parova iz osnovnog pristupa. Umesto toga, za svaki čvor održava skup levih konteksta pomoću bitovskih maski i na osnovu njihove unije proverava uslov maksimalnosti.

Uslov minimalne dužine proverava se tek pri odlučivanju da li će se ponavljanje dodati u rezultat. Čvorovi čija je LCP vrednost manja od zadatog praga ipak se

moraju obraditi, jer se njihove maske levih konteksta koriste pri obradi roditeljskih čvorova.

5.3 Detekcija supermaksimalnih ponavljanja

Detekcija supermaksimalnih ponavljanja direktno primenjuje teoremu 2.2.

Prvi uslov zahteva da posmatrani čvor stabla LCP intervala nema decu. Ako čvor ima dete, postoji duže ponavljanje čiji je prefiks posmatrana niska, pa ona ne može biti supermaksimalna. Umesto eksplicitne provere da li je posmatrana niska sadržana u dužem ponavljanju, dovoljno je proveriti da li je njen LCP interval list stabla.

Drugi uslov zahteva da levi konteksti svih pojavljivanja budu međusobno različiti. Oni se obilaze redom, a postupak se prekida čim se naiđe na već viđen simbol. Ako postoji σ mogućih levih konteksta, interval sa više od σ pojavljivanja biće odbačen najkasnije pri obradi $(\sigma + 1)$-vog elementa.

5.4 Primer detekcije ponavljanja

Postupak je ilustrovan u primeru 5.1.

Primer 5.1 *Posmatrajmo nisku* `ACGTACGTGATTACANN` *i njeno stablo LCP intervala iz primera 4.2, uz minimalnu dužinu ponavljanja* 2. *Čvor* [1..2] *sa* $\omega =$ `ACGT` *nema decu, a njegovi levi konteksti su* $BWT[1] =$ \$ *i* $BWT[2] =$ `T`. *Pošto čvor nema decu, a levi konteksti su međusobno različiti,* ***`ACGT`*** *je i maksimalno i supermaksimalno ponavljanje.*

I čvor [13..14] *sa* $\omega =$ `TAC` *nema decu, a njegovi levi konteksti su* `T` *i* `G`*, pa je niska* ***`TAC`*** *takođe maksimalno i supermaksimalno ponavljanje.*

Čvor [0..2] *sa* $\omega =$ `AC` *ima jedno dete, čvor* [1..2]. *Njegove grupe čine pojedinačna pozicija* 0 *i dete* [1..2]. *Levi kontekst prve grupe je* `T`*, dok grupa određena detetom ima leve kontekste* {\$, `T`}. *Unija njihovih levih konteksta sadrži dva različita simbola, pa je niska* ***`AC`*** *maksimalno ponavljanje. Međutim, pošto čvor ima dete, niska* ***`AC`*** *nije supermaksimalno ponavljanje.*

Čvorovi [0..4] *i* [11..12] *imaju LCP vrednost* 1*, manju od zadate minimalne dužine, pa se odgovarajuća ponavljanja ne dodaju u rezultat.*

Dobijaju se tri maksimalna ponavljanja (`ACGT`, `AC`, `TAC`) i dva supermaksimalna ponavljanja (`ACGT`, `TAC`). Isti rezultati proveravaju se automatskim testom iz odeljka 9.2.

Glava 6

Implementacija

Kompletan izvorni kod implementacije dostupan je u javnom GitHub repozitorijumu: `https://github.com/anaknezeviic/master-rad-esa`.

6.1 Implementacija algoritama za konstrukciju sufiksnog niza

U izvornoj datoteci `suffix_array.cpp` nalaze se tri funkcije koje implementiraju razmatrane algoritme za konstrukciju sufiksnog niza: `build_suffix_array_baseline`, `build_suffix_array_optimized` i `build_suffix_array_sais`. Njihove deklaracije nalaze se u zaglavlju `suffix_array.hpp`. Sve tri funkcije imaju isti interfejs: primaju ulaznu nisku tipa `std::string`, a sufiksni niz vraćaju kao vektor celih brojeva tipa `std::vector<int>`.

Funkcija `build_suffix_array_baseline` implementira osnovno udvostručavanje prefiksa iz odeljka 3.1. U svakoj rundi za sortiranje sufiksa poziva se funkcija standardne biblioteke `std::sort`. Redosled sufiksa određuje lokalna funkcija `compare_suffixes`, koja poredi parove njihovih klasa ekvivalencije. Nepostojeća druga komponenta para predstavlja se vrednošću -1. Pomoćni niz `new_equivalence_class` čuva nove klase ekvivalencije. Postupak se prekida kada poslednji sufiks u uređenom nizu dobije klasu $n - 1$, što znači da svih n sufiksa pripada različitim klasama.

Funkcija `build_suffix_array_optimized` implementira postupak iz odeljka 3.2. Početno sortiranje prebrojavanjem koristi pomoćni niz brojača `count`, koji ima 256 elemenata, po jedan za svaku moguću vrednost bajta. U narednim rundama

pomoćni niz `second_key_order` određuje redosled pozicija prema drugoj komponenti para klasa. U njega se najpre upisuju pozicije za koje druga komponenta ne postoji, a zatim pozicije oblika `suffix_array[i] - k`.

Pomoćni niz `class_size` čuva broj pozicija u svakoj klasi, dok pomoćni niz `class_start` čuva početnu poziciju svake klase u novom sufiksnom nizu. Na osnovu njih obavlja se stabilno sortiranje po prvoj komponenti para, a rezultat se upisuje u pomoćni niz `new_suffix_array`. Nove klase čuvaju se u pomoćnom nizu `new_equivalence_class`. Petlja se završava kada broj klasa dostigne n, odnosno kada svi sufiksi dobiju različite klase.

Funkcija `build_suffix_array_sais` predstavlja spoljni interfejs za algoritam SA-IS iz odeljka 3.3. Svaki bajt ulazne niske pretvara se u ceo broj uvećan za jedan i upisuje u pomoćni niz `encoded_text`. Na kraj tog niza dodaje se završni simbol predstavljen vrednošću 0, pa azbuka kodirane niske ima 257 mogućih vrednosti. Nakon izvršavanja algoritma iz dobijenog sufiksnog niza uklanja se pozicija n, na kojoj se nalazi završni simbol.

Rekurzivni deo algoritma realizuje pomoćna funkcija `sais`. Pomoćna funkcija `classify_suffix_types` određuje S i L tipove pozicija, dok pomoćna funkcija `get_lms_positions` izdvaja LMS pozicije. Provera da li je pozicija LMS obavlja se pomoćnom funkcijom `is_lms_position`.

Pomoćne funkcije `get_bucket_sizes`, `get_bucket_heads` i `get_bucket_tails` određuju veličine, početke i krajeve kofa. Pomoćna funkcija `induced_sort` obavlja postavljanje LMS sufiksa i dva prolaza indukovanog sortiranja. Pomoćna funkcija `build_reduced_string` dodeljuje imena LMS podniskama i gradi redukovanu nisku, dok pomoćna funkcija `lms_substrings_are_equal` proverava jednakost dve LMS podniske. Ako imena LMS podniski nisu jedinstvena, funkcija `sais` rekurzivno se poziva nad redukovanom niskom.

Funkcija `build_suffix_array` predstavlja osnovnu funkciju za konstrukciju sufiksnog niza i podrazumevano poziva funkciju `build_suffix_array_sais`. Tri nezavisne implementacije koriste se i pri testiranju: test iz odeljka 9.1 zahteva da za isti ulaz sve tri vrate jednak sufiksni niz.

6.2 Implementacija poboljšanog sufiksnog niza

Osnovna struktura podataka definisana u modulu `esa` jeste `EnhancedSuffixArray`. Ona objedinjuje ulaznu nisku i četiri prateća niza:

```
struct EnhancedSuffixArray {
  std::string text;
  std::vector<int> suffix_array;
  std::vector<int> inverse_suffix_array;
  std::vector<int> lcp_array;
  std::string bwt;
};
```

Polje `text` omogućava pristup niskama pridruženim LCP intervalima na osnovu njihovih pozicija u sufiksnom nizu. Spoljna reprezentacija ne sadrži veštački završni simbol, pa sva četiri prateća niza imaju dužinu n. Ako se posmatra samo prostor potreban za elemente i pretpostavi da `int` zauzima četiri bajta, struktura zahteva približno $14n$ bajtova: po n bajtova za originalnu sekvencu i BWT i po $4n$ bajtova za svaki od tri celobrojna niza.

Funkcija `build_esa`, definisana u datoteci `esa.cpp`, realizuje redosled iz odeljka 4.1. Parametar `implementation`, čiji je tip nabrajanje `SuffixArrayImplementation`, određuje koja će od tri funkcije za konstrukciju sufiksnog niza biti pozvana. Ako druga vrednost nije zadata, podrazumevano se koristi algoritam SA-IS.

Funkcija zatim jednim prolazom kroz sufiksni niz formira inverzni sufiksni niz. Nakon toga poziva funkciju `build_lcp_array` iz datoteke `lcp.cpp`. Ova funkcija implementira algoritam Kasaija i saradnika [5] i koristi prethodno formiran inverzni sufiksni niz, čime se izbegavaju njegovo ponovno izračunavanje i dodatna privremena alokacija. Na kraju funkcija `build_esa` formira Barouz–Vilerovu transformaciju prema formuli iz odeljka 2.4.

Opcioni parametar `metrics` predstavlja pokazivač na strukturu tipa `ESAConstructionMetrics`. Ako pokazivač nije jednak `nullptr`, u toj strukturi se odvojeno beleže vremena konstrukcije sufiksnog niza, inverznog sufiksnog niza, LCP niza i Barouz–Vilerove transformacije. Za merenje vremena koristi se časovnik standardne biblioteke `std::chrono::steady_clock`. Procena rezervisane memorije računa se na osnovu kapaciteta pet kontejnera strukture `EnhancedSuffixArray`.

Implementacija LCP intervala

Funkcija `build_lcp_intervals` iz datoteke `esa.cpp` realizuje stekovni postupak iz odeljka 4.2 i vraća vektor struktura tipa `LCPInterval`. Svaka struktura sadrži LCP vrednost i levu i desnu granicu intervala. Funkcija je pomoćna i ne koristi se pri izvršavanju algoritama za pronalaženje ponavljanja.

Funkcija `build_lcp_interval_tree` vraća vektor čvorova tipa `LCPIntervalNode`. Svaki čvor sadrži LCP vrednost, levu i desnu granicu intervala i vektor indeksa njegove dece. Pomoćni stek implementiran je vektorom unosa tipa `StackEntry`. Svaki unos čuva LCP vrednost, levu granicu i indekse do tada zatvorene dece intervala. Desna granica određuje se tek prilikom zatvaranja intervala.

Petlja se izvršava do indeksa n. Kada indeks dostigne n, promenljivoj `current_lcp` dodeljuje se pomoćna vrednost 0, čime se zatvaraju svi preostali intervali bez pristupanja nepostojećem elementu $LCP[n]$. Na kraju se u izlazni vektor dodaje korenski čvor sa granicama $[0..n-1]$.

Pošto se svaki čvor dodaje tek nakon svoje dece, za svaki odnos roditelj–dete važi invarijanta iz odeljka 4.2. Ovaj poredak proverava se testom iz odeljka 9.2 i neposredno se koristi u optimizovanoj detekciji maksimalnih ponavljanja.

6.3 Implementacija algoritama za pronalaženje ponavljanja

U izvornoj datoteci `repeats.cpp` implementirani su algoritmi opisani u glavi 5, dok se njihove deklaracije nalaze u datoteci `repeats.hpp`. Rezultat se predstavlja strukturom tipa `Repeat`, koja sadrži nisku, njenu dužinu i vektor pozicija njenih pojavljivanja.

Osnovna detekcija maksimalnih ponavljanja

Funkcija `find_maximal_repeated_pairs` implementira osnovni postupak iz odeljka 5.1. Za svaki čvor stabla LCP intervala formira pomoćni vektor grupa i eksplicitno obilazi parove pozicija iz različitih grupa. Levi kontekst svakog pojavljivanja čita iz originalne niske, dok za pojavljivanje na poziciji nula koristi simbol `$`. Za svaki pronađeni maksimalni ponovljeni par funkcija `substr` izdvaja odgovarajuću nisku.

Funkcija `find_maximal_repeats_baseline` najpre poziva funkciju `find_maximal_repeated_pairs`. Iz dobijenih parova zatim izdvaja različite niske pomoću uređenog skupa tipa `std::set<std::string>`. Nakon toga ponovo gradi stablo LCP intervala i za svaku zadržanu nisku formira potpunu listu pozicija pojavljivanja. Pozicije se sortiraju rastuće pomoću funkcije standardne biblioteke `std::sort`.

Optimizovana detekcija maksimalnih ponavljanja

Funkcija `find_maximal_repeats` implementira pseudokod 1.

Tip `LeftContextMask` predstavlja drugo ime za tip `std::bitset<256>` i koristi se za čuvanje maske levih konteksta. Pomoćni vektor `node_left_contexts` čuva po jednu takvu masku za svaki čvor. Zbog poretka čvorova odozdo naviše, maska deteta već je izračunata kada počne obrada njegovog roditelja.

Za svaki čvor promenljiva `accumulated_contexts` predstavlja uniju levih konteksta prethodno obrađenih grupa, dok promenljiva `complete_node_contexts` predstavlja uniju levih konteksta celog intervala. Pomoćna funkcija `has_distinct_left_contexts` proverava da li su obe prosleđene maske neprazne i da li njihova unija sadrži najmanje dva postavljena bita. Po završetku obrade čvora njegova potpuna maska upisuje se u vektor `node_left_contexts`, bez obzira na zadatu minimalnu dužinu, jer može biti potrebna pri obradi roditeljskog čvora.

Detekcija supermaksimalnih ponavljanja

Funkcija `find_supermaximal_repeats` odbacuje čvorove čija je LCP vrednost nula, čija je LCP vrednost manja od zadate minimalne dužine ili koji imaju decu. Za preostale čvorove promenljiva `left_contexts`, tipa `LeftContextMask`, beleži već pronađene leve kontekste. Obrada intervala prekida se čim se naiđe na levi kontekst čiji je bit već postavljen.

Za validiran FASTA ulaz mogući levi konteksti su `A`, `C`, `G`, `T`, `N` i `$`. Zbog toga interval sa više od šest pojavljivanja ne može zadovoljiti uslov da su svi levi konteksti međusobno različiti.

Ponovna upotreba stabla LCP intervala

Za funkcije `find_maximal_repeats` i `find_supermaximal_repeats` postoje po dve varijante sa različitim brojem parametara. Varijanta koja prima poboljšani su-

fiksni niz i opcije interno gradi stablo LCP intervala. Druga varijanta dodatno prima već izgrađeni vektor čvorova, pa stablo ne gradi ponovo.

Kada je program pokrenut u konfiguraciji `final` sa opcijom `-type both`, funkcija `main` jednom gradi stablo LCP intervala, a zatim isti vektor čvorova prosleđuje funkcijama za pronalaženje maksimalnih i supermaksimalnih ponavljanja. Konfiguracije programa detaljnije su opisane u odeljku 8.3.

U ovoj konfiguraciji stablo se gradi nakon početka merenja vremena pronalaženja maksimalnih ponavljanja, pa je vreme njegove izgradnje uključeno u rezultat tog merenja. Merenje vremena pronalaženja supermaksimalnih ponavljanja počinje nakon toga i koristi već izgrađeno stablo. Ova razlika uzima se u obzir pri tumačenju rezultata u odeljku 10.4.

Glava 7

Analiza složenosti

7.1 Oznake i pregled složenosti

U analizi koristimo sledeće oznake. Sa n označavamo dužinu ulazne sekvence, a sa σ veličinu azbuke. Za FASTA ulaz dozvoljeni su simboli `A`, `C`, `G`, `T` i `N`. U delovima implementacije koji simbole obrađuju kao bajtovske vrednosti koristi se skup od 256 mogućih vrednosti. Algoritam SA-IS interno koristi 257 vrednosti, jer se originalne bajtovske vrednosti pomeraju za jedan, a vrednost 0 rezerviše se za jedinstveni završni simbol. Oznake z i z_s predstavljaju ukupan broj pozicija pojavljivanja pronađenih maksimalnih, odnosno supermaksimalnih ponavljanja. Oznake m i m_s predstavljaju zbir dužina svih pronađenih maksimalnih, odnosno supermaksimalnih ponavljanja. Sa P označavamo broj maksimalnih ponovljenih parova koje generiše osnovni pristup, sa R broj različitih maksimalnih ponavljanja, a sa $\ell_{\max}$ najveću dužinu pronađenog ponavljanja.

Veličine z i z_s zavise od sadržaja ulazne sekvence i od zadate minimalne dužine ponavljanja, a ne samo od njene dužine n. Zbog toga ukupno vreme izvršavanja ne zavisi samo od dužine sekvence, već i od veličine rezultata koje je potrebno formirati. U analizi se zato posebno posmatra vreme potrebno za pronalaženje ponavljanja i vreme potrebno za formiranje njihovih niski i pozicija pojavljivanja.

Tabela 7.1 prikazuje vremensku i prostornu složenost razmatranih algoritama i ostalih koraka obrade.

Tabela 7.1: Vremenska i prostorna složenost razmatranih algoritama i koraka obrade. Oznake su objašnjene u odeljku 7.1.

Algoritam / korak obrade	Vreme	Prostor
Osnovno udvostručavanje prefiksa	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(n)$
Optimizovano udvostručavanje prefiksa	$\mathcal{O}(n \log n)$	$\mathcal{O}(n + \sigma)$
SA-IS	$\mathcal{O}(n)$	$\mathcal{O}(n + \sigma)$
Inverzni sufiksni niz	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Kasaijev LCP niz	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Barouz–Vilerova transformacija	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Stablo LCP intervala	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Maksimalni ponovljeni parovi	$\mathcal{O}(n^2 + P\ell_{\max})$	$\mathcal{O}(n + P\ell_{\max})$
Osnovna detekcija maksimalnih	$\mathcal{O}(n^2 + P\ell_{\max} \log R)$	$\mathcal{O}(n + P\ell_{\max})$
Klasifikacija maksimalnih	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Formiranje rezultata za maksimalna ponavljanja	$\mathcal{O}(z + m)$	proporcionalno veličini rezultata
Maksimalna ponavljanja, ukupno	$\mathcal{O}(n + z + m)$	$\mathcal{O}(n)$ + prostor za rezultat
Klasifikacija supermaksimalnih	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Formiranje rezultata za supermaksimalna ponavljanja	$\mathcal{O}(z_s + m_s)$	proporcionalno veličini rezultata
Supermaksimalna ponavljanja, ukupno	$\mathcal{O}(n + z_s + m_s)$	$\mathcal{O}(n)$ + prostor za rezultat
Čitanje FASTA datoteke	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Upis maksimalnih rezultata u CSV	$\mathcal{O}(z + m)$	$\mathcal{O}(1)$
Upis supermaksimalnih rezultata u CSV	$\mathcal{O}(z_s + m_s)$	$\mathcal{O}(1)$

7.2 Složenost određivanja ponavljanja i formiranja rezultata

Iz tabele 7.1 vidi se da je određivanje maksimalnih ponavljanja optimizovanim pristupom linearne vremenske složenosti u odnosu na dužinu sekvence n. Svaki čvor stabla obrađuje se jednom, svaka veza roditelj–dete razmatra se jednom, a levi konteksti svih pojavljivanja obuhvaćenih jednim detetom predstavljeni su jednom već izračunatom bitovskom maskom.

Vreme potrebno za formiranje rezultata zavisi od ukupnog broja pozicija pojavljivanja pronađenih ponavljanja i od ukupne dužine pronađenih niski. Za svako pronađeno ponavljanje u rezultat se upisuju sve njegove pozicije pojavljivanja, pa ukupan rad za taj deo odgovara veličinama z i z_s. Pored toga, funkcija `substr` kopira samu nisku ω, zbog čega vreme zavisi i od ukupnih dužina m i m_s. Zato ukupna vremenska složenost iznosi $\mathcal{O}(n+z+m)$ za maksimalna ponavljanja i $\mathcal{O}(n+z_s+m_s)$ za supermaksimalna ponavljanja.

Ako bi se umesto samih niski čuvale samo njihove početne pozicije i dužine, ne bi bilo potrebno kopirati sadržaj niski, pa bi ukupne vremenske složenosti iznosile $\mathcal{O}(n + z)$ za maksimalna i $\mathcal{O}(n + z_s)$ za supermaksimalna ponavljanja.

U analiziranim skupovima podataka vrednosti z i m bile su znatno manje od n, što ilustruje primer 7.1.

Primer 7.1 *Za genom* Escherichia coli *sa* $n = 4\,641\,652$ *i minimalnom dužinom ponavljanja* 20 *izmereni su* $z = 9\,889$ *i* $m = 147\,609$. *Član* n *je u ovom slučaju znatno veći od* z *i* m, *pa ima najveći uticaj na ukupnu vremensku složenost.*

7.3 Vremenska i prostorna složenost osnovnog pristupa

Osnovni pristup ima dva izvora dodatne složenosti. Prvi izvor je nabrajanje parova. Za čvor sa q pojavljivanja razmatra se do $\binom{q}{2}$ parova, pa broj razmatranih parova u najgorem slučaju može rasti kvadratno sa veličinom ulaza. Drugi izvor dodatnog vremena izvršavanja je kopiranje niski, jer se za svaki prihvaćeni par poziva funkcija `substr`. Složenost ovog koraka u najgorem slučaju je $\mathcal{O}(P\ell_{\max})$. Uz to, osnovni pristup gradi stablo LCP intervala dva puta. Za čuvanje različitih maksimalnih ponavljanja koristi se uređeni skup niski, pri čemu poređenje dve niske u najgorem slučaju zahteva $\mathcal{O}(\ell_{\max})$ vremena. U eksperimentima iz odeljka 11.5 najgore kvadratno ponašanje osnovnog pristupa nije se jasno ispoljilo. Razlog je to što broj parova koje osnovni pristup razmatra zavisi od broja pojavljivanja pojedinačnog ponavljanja: ako se ono javlja q puta, može se formirati do $\binom{q}{2}$ parova, kao što je prikazano u primeru 7.2.

Primer 7.2 *Za dva pojavljivanja postoji samo jedan par, za tri pojavljivanja tri para, dok bi za sto pojavljivanja bilo potrebno razmotriti čak* 4 950 *parova.*

U analiziranim eksperimentima takav rast broja parova nije doveo do izraženog kvadratnog rasta vremena izvršavanja, pa se izmereno ubrzanje optimizovanog pristupa kreće od približno 1,8 do 2,6 puta i ne pokazuje jasan rast sa povećanjem dužine ulaza.

Osnovni deo zauzete memorije čini struktura poboljšanog sufiksnog niza. Ako se posmatra samo prostor potreban za njene elemente, ona zauzima približno $14n$ bajtova: po n bajtova za ulaznu DNK sekvencu i BWT i po $4n$ bajtova za sufiksni niz, inverzni sufiksni niz i LCP niz. Optimizovana detekcija dodatno čuva po jednu bitovsku masku od 32 bajta za svaki čvor stabla LCP intervala. Zbog toga se dodatna potrošnja memorije povećava proporcionalno broju čvorova stabla, što doprinosi razlikama u ukupnoj potrošnji memorije između osnovnog i optimizovanog pristupa prikazanim u odeljku 11.5. Osnovni pristup ne koristi bitovske maske, ali eksplicitno

čuva sve generisane maksimalne ponovljene parove i skup različitih niski. Zbog toga odnos potrošnje memorije osnovnog i optimizovanog pristupa zavisi od strukture ulazne sekvence.

Glava 8

Ulaz, izlaz i struktura programa

8.1 FASTA ulaz

Ulazne sekvence se čitaju iz datoteka u FASTA formatu, u kojem prvi neprazan red počinje znakom `>` i sadrži zaglavlje, dok naredni redovi sadrže sekvencu.

Modul `fasta.cpp` proverava ispravnost FASTA ulaza. Prihvata se tačno jedan zapis po datoteci, pri čemu zaglavlje mora prethoditi sekvenci. Prazni redovi i beline unutar redova sekvence se preskaču, dok se mala slova prevode u velika. Dozvoljeni simboli ulazne sekvence su `A`, `C`, `G`, `T` i `N`. Greška se prijavljuje ako datoteka sadrži drugo zaglavlje, sekvencu pre zaglavlja, nedozvoljen simbol ili praznu sekvencu.

Simbol `N` u FASTA sekvenci označava nepoznat nukleotid, ali se u ovom radu ne tretira kao simbol koji može da odgovara bilo kom od nukleotida `A`, `C`, `G` ili `T`. Umesto toga, `N` se obrađuje kao običan simbol azbuke, pa se dva pojavljivanja simbola `N` smatraju jednakim samo međusobno. Ovakvo tumačenje simbola `N` usvojeno je u ovom radu, jer bi njegovo tretiranje kao simbola koji može odgovarati bilo kom nukleotidu zahtevalo drugačiju definiciju jednakosti podniski i odgovarajuće izmene algoritama.

Validacija sprečava da se u obradu uključe simboli van dozvoljene azbuke. To obuhvata, na primer, simbole iz proširene IUPAC notacije za zapis nukleotida, kao što su `R`, `Y`, `S` i `W`, za koje u ovom radu nije definisana posebna semantika. Sve greške pri obradi FASTA ulaza prijavljuju se izuzetkom tipa `std::runtime_error`. Funkcija `main` hvata ovaj izuzetak, ispisuje odgovarajuću poruku na standardni izlaz za greške i završava program povratnim kodom 1.

8.2 Izlaz sa ponavljanjima

Kada program nije pokrenut u režimu merenja performansi, koji se uključuje opcijom `-benchmark`, pronađena ponavljanja ispisuju se na standardni izlaz i upisuju u CSV datoteku sa zaglavljem `sequence,length,occurrences,positions`. Kolona `sequence` sadrži pronađenu nisku, `length` njenu dužinu, a `occurrences` broj njenih pojavljivanja. Kolona `positions` sadrži pozicije tih pojavljivanja, zapisane unutar navodnika i međusobno razdvojene jednim razmakom. Konkretan zapis prikazan je u primeru 8.1.

Primer 8.1 *ACGT,4,2,„0 4". U ovom zapisu niska* `ACGT` *dužine* 4 *ima dva pojavljivanja, na pozicijama* 0 *i* 4.

Pozicije navedene u koloni `positions` indeksirane su od nule i odnose se na sekvencu dobijenu nakon uklanjanja belina i prevođenja malih slova u velika.

Direktorijum u koji se upisuje izlazna datoteka određuje se prema direktorijumu ulazne datoteke. Rezultati za realne genome, smeštene u direktorijumu `real`, upisuju se u istoimeni poddirektorijum ispod `data/processed/repeats/`. Rezultati za test primere iz direktorijuma `tests` upisuju se u poddirektorijum `tests`, dok se rezultati za sintetičke sekvence upisuju u poddirektorijum `synthetic`. Ime izlazne datoteke izvodi se iz imena ulazne datoteke, uz oznaku `maximal` ili `supermaximal`, u zavisnosti od vrste pronađenih ponavljanja.

U režimu merenja performansi preskaču se ispis pronađenih ponavljanja na standardni izlaz i upis rezultata u CSV datoteke, kako vreme potrebno za formiranje izlaza ne bi bilo uključeno u vreme detekcije ponavljanja.

8.3 Moduli i konfiguracije

Projekat je podeljen na module sa jasno razdvojenim odgovornostima, koji su prikazani u tabeli 8.1.

Glavni algoritamski moduli organizovani su tako da između njih nema kružnih zavisnosti. Modul `esa` koristi funkcionalnosti modula `suffix_array` i `lcp`. Pri tome je modul `lcp` nezavisan od modula `esa`, dok modul `suffix_array` koristi samo standardnu biblioteku. Modul `repeats` zatim koristi strukture i funkcije modula `esa` za pronalaženje maksimalnih i supermaksimalnih ponavljanja u ulaznoj sekvenci. Ovakva organizacija omogućava da svaki test koristi samo module potrebne za konkretnu

Tabela 8.1: Moduli implementacije i njihove odgovornosti.

Modul	Odgovornost
`suffix_array.*`	tri algoritma za konstrukciju sufiksnog niza
`lcp.*`	Kasaijev algoritam
`esa.*`	struktura poboljšanog sufiksnog niza, LCP intervali i njihovo stablo
`repeats.*`	maksimalni ponovljeni parovi, maksimalna i supermaksimalna ponavljanja
`fasta.*`	čitanje i validacija FASTA zapisa
`output.*`	upis ponavljanja u CSV
`benchmark.*`	čuvanje rezultata merenja i upis u CSV datoteku
`memory.*`	očitavanje maksimalne potrošnje memorije procesa
`main.cpp`	obrada argumenata, upravljanje tokom programa, ispis i merenje

proveru. Projekat se gradi pomoću alata CMake i koristi standard C++17. Datoteka `CMakeLists.txt` definiše glavni program i tri test programa, koji su registrovani u okviru sistema CTest.

Program se pokreće navođenjem putanje do FASTA datoteke, uz opcije komandne linije navedene u tabeli 8.2. Greška se prijavljuje ako je navedena nepoznata opcija, ako nedostaje njena vrednost ili ako prosleđena vrednost nije dozvoljena. Opcija `-benchmark-name` može se koristiti samo zajedno sa opcijom `-benchmark`.

Tabela 8.2: Opcije komandne linije.

Opcija	Vrednosti	Podrazumevano
`-min-length`	ceo broj ≥ 1	1
`-type`	`maximal`, `supermaximal`, `both`	`both`
`-implementation`	`baseline`, `optimized-sa`, `sais`, `final`	`final`
`-benchmark`	—	isključeno
`-benchmark-name`	ime datoteke bez nastavka	automatski određeno

Opcija `-min-length` određuje minimalnu dužinu ponavljanja koja će biti prijavljena, dok `-type` određuje da li se traže maksimalna, supermaksimalna ili obe vrste ponavljanja.

Opcija `-implementation` određuje kombinaciju algoritama za konstrukciju sufiksnog niza i pronalaženje maksimalnih ponavljanja. Četiri dostupne konfiguracije prikazane su u tabeli 8.3. Ovakav izbor konfiguracija omogućava dva nezavisna, kontrolisana poređenja, o čemu govori odeljak 10.3. Konfiguracija `final` dodatno deli stablo LCP intervala između dve vrste ponavljanja, kao što je opisano u odeljku 6.3.

Opcija `-benchmark` uključuje režim merenja performansi, u kojem se rezultati merenja upisuju u CSV datoteku. Opcijom `-benchmark-name` može se zadati ime te datoteke bez nastavka `.csv`; ako ime nije zadato, ono se određuje automatski prema direktorijumu ulazne datoteke.

Tabela 8.3: Konfiguracije implementacije. Supermaksimalna ponavljanja se u svim konfiguracijama traže optimizovanim algoritmom.

Konfiguracija	Sufiksni niz	Maksimalna ponavljanja
`baseline`	osnovno udvostručavanje prefiksa	osnovni algoritam
`optimized-sa`	optimizovano udvostručavanje prefiksa	osnovni algoritam
`sais`	SA-IS	osnovni algoritam
`final`	SA-IS	optimizovani algoritam

8.4 Merenje vremena rada i memorijskog zauzeća

Vremena se mere pomoću `std::chrono::steady_clock`. Posebno se mere vreme čitanja FASTA datoteke, vremena četiri koraka izgradnje ESA, ukupno vreme izgradnje ESA i vremena detekcije maksimalnih i supermaksimalnih ponavljanja. Pored toga, meri se ukupno vreme obrade od pokretanja programa do završetka detekcije ponavljanja i očitavanja potrošnje memorije.

Maksimalna potrošnja memorije procesa meri se u modulu `memory.cpp`. Na sistemu Windows koristi se funkcija `GetProcessMemoryInfo`, iz koje se uzima vrednost `PeakWorkingSetSize`. Ona predstavlja najveću zabeleženu veličinu radnog skupa procesa tokom izvršavanja. Na ostalim platformama ova vrednost se trenutno ne meri, pa svi rezultati potrošnje memorije prikazani u ovom radu potiču sa sistema Windows.

Procena memorije strukture ESA računa se posebno, sabiranjem memorije potrebne za ulaznu sekvencu, sufiksni niz, inverzni sufiksni niz, LCP niz i BWT. Ova procena ne uključuje dodatnu memoriju koja se privremeno koristi tokom kasnijih faza izvršavanja programa. Nasuprot tome, `PeakWorkingSetSize` obuhvata najveću količinu memorije koju je ceo proces koristio tokom izvršavanja, uključujući i memoriju za privremene strukture i druge delove programa.

Glava 9

Testiranje i validacija

Ispravnost implementacije proverava se automatskim testovima koji se pokreću alatom CTest. Postoje tri testna programa: `test_suffix_array`, `test_repeats` i `test_fasta`. Testovi ne koriste dodatnu biblioteku za testiranje. Provere su implementirane pomoću funkcija koje vraćaju logičku vrednost, dok glavna funkcija objedinjuje njihove rezultate i završava program neuspešnim izlaznim kodom ako neka provera ne uspe.

9.1 Testovi sufiksnog niza

Program `test_suffix_array` proverava sve tri implementacije konstrukcije sufiksnog niza na istim ulazima. Pomoćna funkcija `check_suffix_array` zahteva da sve tri implementacije vrate očekivani sufiksni niz i da su njihovi rezultati međusobno jednaki.

Testni primeri obuhvataju granične i klasične slučajeve: praznu nisku, nisku `A`, niske `BANANA` i `MISSISSIPPI`, nisku `AAAAA` i mali DNK primer `ACGTACGTGATTACANN`. Za niske `BANANA` i `MISSISSIPPI` koriste se poznati sufiksni nizovi $5, 3, 1, 0, 4, 2$, odnosno $10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2$. Prazna niska i niska dužine jedan proveravaju granične slučajeve, koji su čest izvor grešaka u indeksiranju. Niske `BANANA` i `MISSISSIPPI` koriste se kao primeri sa više preklapajućih ponavljanja različitih dužina. Niska `AAAAA` proverava ponašanje na izrazito repetitivnom ulazu, kod kojeg veliki broj sufiksa ima duge zajedničke prefikse. DNK primer proverava rad nad dozvoljenom azbukom ulaznih sekvenci, uključujući simbol `N`.

Poređenje tri nezavisne implementacije predstavlja dodatnu proveru ispravnosti, jer se pored poznatog očekivanog rezultata proverava i saglasnost njihovih rezultata.

9.2 Testovi ponavljanja

Program `test_repeats` sadrži tri testa.

Prvi test proverava strukturnu invarijantu iz odeljka 4.2. Za DNK nisku iz odeljka 5.4 gradi se stablo LCP intervala, a zatim se za svaki odnos roditelj–dete proverava da je indeks deteta manji od indeksa roditelja i da je interval deteta sadržan u intervalu roditelja. Ovaj test ne proverava rezultat algoritma za pronalaženje ponavljanja, već strukturnu osobinu stabla na kojoj se optimizovani algoritam zasniva. Ako bi promena u konstrukciji stabla narušila poredak odozdo naviše, test bi takvu promenu neposredno otkrio.

Ulaz i očekivani rezultat drugog testa dati su u primeru 9.1. Primer potiče iz rada o kontekstno osetljivim ponavljanjima [12] i pogodan je jer razdvaja dva pojma koja se lako mešaju.

Primer 9.1 *Za nisku* `NLAREPLNOREPTFCGIREPTLSIG` *minimalna dužina ponavljanja postavljena je na* 3*. Očekuju se dva maksimalna ponavljanja:* `REP` *na pozicijama* $\{3, 9, 17\}$ *i* `REPT` *na pozicijama* $\{9, 17\}$*, kao i jedno supermaksimalno ponavljanje,* `REPT`*. Niska* `REP` *jeste maksimalno ponavljanje, ali nije supermaksimalna jer je sadržana u dužem maksimalnom ponavljanju* `REPT`*. Test proverava i tačan broj pronađenih ponavljanja, čime se otkriva i pojava neispravnih dodatnih rezultata i izostavljanje očekivanih ponavljanja.*

Pošto primer 9.1 ne pripada DNK azbuci, on dodatno potvrđuje da algoritmi za pronalaženje ponavljanja nisu ograničeni isključivo na simbole `A`, `C`, `G`, `T` i `N`.

Treći test je regresioni i koristi nisku `ACGTACGTGATTACANN` sa minimalnom dužinom 2, sa rezultatima datim u odeljku 5.4. Njegova uloga je da proveri da izmene u kodu ne promene očekivane rezultate maksimalnih i supermaksimalnih ponavljanja za ovaj testni slučaj.

Pomoćna funkcija `contains_repeat` sortira očekivane i dobijene pozicije pre poređenja, pa se one porede nezavisno od redosleda. Razlog je to što optimizovana implementacija vraća pozicije redosledom sufiksnog niza, kao što je prikazano u primeru 9.2.

Primer 9.2 *Za nisku* `AC` *u DNK primeru 5.1 optimizovana implementacija vraća redosled pozicija* $12, 0, 4$*, dok osnovna implementacija vraća* $0, 4, 12$*. U oba slučaja skup pozicija je isti.*

Na taj način test ne zahteva numerički uređen redosled pozicija, koji nije deo definicije ponavljanja.

9.3 Testovi FASTA čitača i provere u analizi

Program `test_fasta` koristi četiri male FASTA datoteke iz direktorijuma `data/raw/tests/`. Test uspešnog čitanja proverava da su iz datoteke `test.fasta` pravilno pročitani zaglavlje i sekvenca. Preostala tri testa proveravaju da se greška prijavljuje za nedozvoljen simbol, datoteku sa više FASTA zapisa i datoteku bez zaglavlja. Ovi testovi smatraju se uspešnim samo ako čitanje odgovarajuće datoteke dovede do izuzetka.

Dodatne provere izvode se i u skriptama za analizu rezultata. Skripta `plot_benchmarks.py` proverava da je broj pronađenih maksimalnih i supermaksimalnih ponavljanja jednak u svim ponovljenim merenjima za svaku posmatranu dužinu sekvence. Skripte `analyze_suffix_array.py` i `analyze_repeat_algorithms.py` dodatno proveravaju da konfiguracije koje se međusobno porede daju isti broj ponavljanja na istom skupu podataka. Ako neka od ovih provera ne uspe, skripta prekida izvršavanje i prijavljuje grešku.

Ove provere su posebno korisne na velikim ulazima, uključujući realne genome, za koje nije praktično ručno proveriti kompletan skup očekivanih rezultata. One same po sebi ne potvrđuju ispravnost pronađenih ponavljanja, ali mogu otkriti promenu rezultata između ponovljenih merenja ili između konfiguracija koje treba da daju isti rezultat.

Glava 10

Eksperimentalna metodologija

10.1 Okruženje i ponavljanje merenja

Sva merenja izvedena su na sistemu Windows, korišćenjem prevodioca MSVC i konfiguracije `Release`. Izbor konfiguracije je važan, jer `Debug` konfiguracija može uključivati dodatne provere i druge mehanizme koji povećavaju vreme izvršavanja, pa nije pogodna za merenje performansi optimizovanog koda. Merenje maksimalne potrošnje memorije procesa dostupno je samo na sistemu Windows, kao što je opisano u odeljku 8.4.

Svaka konfiguracija pokrenuta je pet puta za svaki skup podataka, a za obradu rezultata korišćena je medijana. Medijana je manje osetljiva na pojedinačna odstupanja u merenjima od aritmetičke sredine, što je važno kod merenja vremena izvršavanja. Kod sintetičke sekvence dužine 10 000 zabeležena je izražena razlika između prvog i narednih čitanja FASTA datoteke. Prvo čitanje trajalo je 23,31 ms, dok su naredna četiri trajala između 0,90 i 1,41 ms. Medijana od 1,32 ms zbog toga bolje predstavlja tipično vreme čitanja od aritmetičke sredine.

10.2 Skupovi podataka

Sintetičke DNK sekvence generisane su skriptom `generate_dna.py`, pri čemu je svaki simbol iz azbuke `ACGT` biran sa jednakom verovatnoćom. Za sva merenja korišćeno je seme (engl. *seed*) 42, pa su sekvence potpuno reproducibilne. Korišćene su dužine 10 000, 50 000, 100 000, 250 000 i 500 000, što omogućava praćenje ponašanja algoritama sa povećanjem dužine ulazne sekvence.

Prednost sintetičkih podataka je potpuna kontrola nad dužinom sekvence i mogućnost da se isti eksperimenti ponove nad identičnim ulazima. Njihovo ograničenje je to što ravnomerno generisane sekvence ne odražavaju u potpunosti strukturu realnih genoma, pa se broj i raspored ponavljanja, a time i ponašanje algoritama, mogu razlikovati. Zbog toga su u eksperimente uključeni i realni genomi.

Korišćena su četiri kompletna genoma preuzeta iz baze NCBI RefSeq, čiji su podaci dati u tabeli 10.1. Odabrani skup čine jedan virusni i tri bakterijska genoma i izabran je tako da obuhvati sekvence različitih dužina i različitog sastava baza. SARS-CoV-2 predstavlja najmanji skup podataka, dok su *Bacillus subtilis* i *Escherichia coli* najveći i međusobno sličnih dužina. *Mycoplasma genitalium* ima znatno manji genom od ova dva bakterijska genoma. Razlike u GC sadržaju[1] dodatno omogućavaju posmatranje ponašanja algoritama na sekvencama različitog sastava baza.

Tabela 10.1: Realni genomi korišćeni u eksperimentima.

Organizam	Pristupni broj	Dužina (bp)	GC (%)	Min. dužina
SARS-CoV-2 Wuhan-Hu-1	NC_045512.2	29 903	37,97	10
M. genitalium G37	NC_000908.2	580 076	31,69	20
B. subtilis 168	NC_000964.3	4 215 606	43,51	20
E. coli K-12 MG1655	NC_000913.3	4 641 652	50,79	20

Minimalna dužina ponavljanja postavljena je na 10 za SARS-CoV-2 i na 20 za ostale genome. Za veće genome korišćen je viši prag kako bi se ograničio broj kratkih ponavljanja i veličina rezultujućeg izlaza. Za svaki skup podataka isti prag korišćen je u svim poređenim konfiguracijama, čime je obezbeđeno njihovo poređenje pod istim uslovima.

10.3 Kontrolisana poređenja

Merenja su organizovana kao dva odvojena poređenja. U svakom od njih menja se samo jedan algoritamski izbor, dok svi ostali uslovi ostaju isti.

U *poređenju sufiksnih nizova* porede se konfiguracije `baseline`, `optimized-sa` i `sais`. Sve tri konfiguracije koriste isti osnovni algoritam za pronalaženje maksimalnih ponavljanja, pa se razlikuju samo po algoritmu za konstrukciju sufiksnog niza. Za poređenje se koristi vreme iz kolone `suffix_array_time_ms`.

[1]GC sadržaj predstavlja udeo nukleotida guanina (G) i citozina (C) u ukupnoj dužini DNK sekvence, izražen u procentima.

U *poređenju algoritama za pronalaženje ponavljanja* porede se konfiguracije `sais` i `final`. Obe koriste sufiksni niz konstruisan algoritmom SA-IS, pa se razlikuju samo po algoritmu za pronalaženje maksimalnih ponavljanja. Za poređenje se koristi vreme iz kolone `repeat_detection_time_ms`.

Ovakvo razdvajanje omogućava da se uticaj svake optimizacije posmatra nezavisno. Direktno poređenje konfiguracija `baseline` i `final` istovremeno bi obuhvatilo promenu algoritma za konstrukciju sufiksnog niza i promenu algoritma za pronalaženje maksimalnih ponavljanja, pa ne bi bilo moguće utvrditi koliki deo razlike u performansama potiče od svake od te dve promene.

10.4 Metrike i obrada rezultata

Svako pokretanje programa u režimu merenja performansi upisuje jedan ili dva reda u CSV datoteku sa šesnaest kolona: ime skupa podataka, konfiguraciju, dužinu sekvence, minimalnu dužinu ponavljanja, vrstu ponavljanja, vreme čitanja FASTA datoteke, vremena četiri koraka izgradnje ESA, ukupno vreme izgradnje ESA, vreme detekcije ponavljanja, ukupno vreme obrade, procenjenu memoriju ESA, maksimalnu potrošnju memorije procesa i broj pronađenih ponavljanja. Kada je izabrana opcija `-type both`, upisuju se dva reda, po jedan za maksimalna i supermaksimalna ponavljanja, sa istim vrednostima metrika koje se odnose na izgradnju ESA.

Pri tumačenju kolone `repeat_detection_time_ms` treba imati u vidu ponovnu upotrebu stabla LCP intervala opisanu u odeljku 6.3. U konfiguraciji `final` sa opcijom `-type both`, izgradnja stabla ulazi u vreme detekcije maksimalnih ponavljanja, dok detekcija supermaksimalnih ponavljanja koristi već izgrađeno stablo. Zbog toga vreme detekcije supermaksimalnih ponavljanja ne uključuje izgradnju stabla i nije direktno uporedivo sa vremenom detekcije maksimalnih ponavljanja.

Obrada rezultata odvojena je od glavnog programa i izvedena u jeziku Python. Skripta `plot_benchmarks.py` obrađuje merenja skalabilnosti konačne konfiguracije i generiše grafikone `01`–`04`. Skripta `analyze_suffix_array.py` obrađuje poređenje algoritama za konstrukciju sufiksnog niza i generiše grafikone `05`–`08`, dok `analyze_repeat_algorithms.py` obrađuje poređenje algoritama za pronalaženje ponavljanja i generiše grafikone `09`–`12`.

Sve tri skripte koriste medijane ponovljenih merenja i proveravaju konzistentnost broja pronađenih ponavljanja, kao što je opisano u odeljku 9.3. Skripte za poređenje sufiksnih nizova i algoritama za ponavljanja dodatno upisuju zbirne CSV datoteke,

dok `plot_benchmarks.py` generiše samo grafikone.

Glava 11

Eksperimentalni rezultati

Ova glava prikazuje rezultate eksperimentalnih merenja. Rezultati su dobijeni iz CSV datoteka u direktorijumu `data/processed/benchmarks/` i obrađeni u skladu sa metodologijom iz glave 10, pri čemu se za ponovljena merenja koristi medijana. Grafikoni su generisani skriptama opisanim u odeljku 10.4. Tabele prikazuju numeričke vrednosti, dok grafikoni omogućavaju lakše uočavanje trendova.

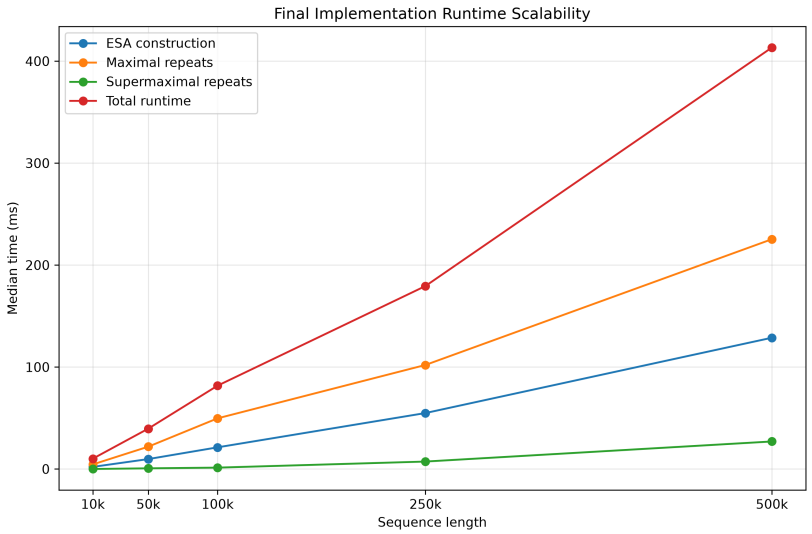
11.1 Skalabilnost konačne konfiguracije

Prvi eksperiment ispituje kako se konačna konfiguracija ponaša sa povećanjem dužine ulazne sekvence, uz minimalnu dužinu ponavljanja 10. Tabela 11.1 prikazuje medijane vremena izgradnje ESA, detekcije maksimalnih i supermaksimalnih ponavljanja i ukupnog vremena obrade, kao i vremena pojedinačnih koraka izgradnje ESA.

Tabela 11.1: Medijane vremena izvršavanja konačne konfiguracije na sintetičkim sekvencama, u milisekundama, i broj pronađenih ponavljanja. Kolone SA-IS, ISA, LCP i BWT prikazuju vremena pojedinačnih koraka izgradnje poboljšanog sufiksnog niza, dok kolona ESA prikazuje ukupno vreme njegove izgradnje.

Dužina	SA-IS	ISA	LCP	BWT	ESA	Maks.	Supermaks.	Ukupno	Br. maks.	Br. supermaks.
10 000	1,67	0,04	0,28	0,03	2,09	4,52	0,03	10,15	24	24
50 000	7,73	0,24	1,53	0,18	9,74	21,90	0,67	39,58	883	874
100 000	15,86	0,38	4,03	0,34	21,26	49,68	1,36	81,72	3 383	3 291
250 000	41,20	1,32	10,82	0,84	54,87	102,03	7,31	179,42	20 389	18 761
500 000	92,12	3,09	27,00	1,89	128,73	225,34	27,00	413,32	74 795	63 667

Slika 11.1 prikazuje rast vremena izvršavanja sa povećanjem dužine ulazne sekvence. Kada se dužina poveća sa 10 000 na 500 000 baza, odnosno 50 puta, vreme detekcije maksimalnih ponavljanja povećava se sa 4,52 na 225,34 ms, što je pribli-



Slika 11.1: Skalabilnost konačne konfiguracije na sintetičkim DNK sekvencama. Prikazane su medijane pet pokretanja za vreme izgradnje ESA, detekcije maksimalnih i supermaksimalnih ponavljanja i ukupno vreme obrade.

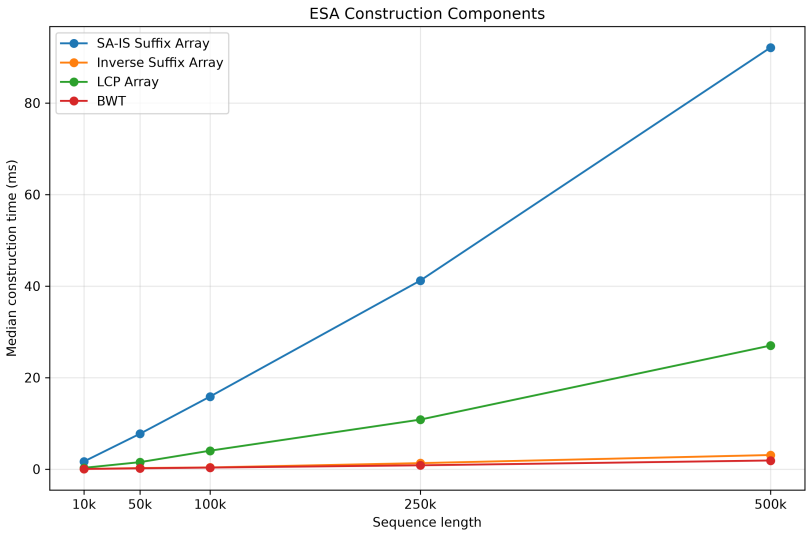
žno 49,9 puta. Vreme izgradnje ESA povećava se približno 61,5 puta, dok se ukupno vreme obrade povećava približno 40,7 puta. Dobijeni rezultati pokazuju da vreme izvršavanja u posmatranom opsegu raste približno linearno sa povećanjem dužine ulazne sekvence, iako rast nije potpuno proporcionalan za sve merene delove obrade.

Vreme detekcije supermaksimalnih ponavljanja povećava se sa približno 0,03 ms za sekvencu dužine 10 000 na 27,00 ms za sekvencu dužine 500 000. Istovremeno, broj pronađenih supermaksimalnih ponavljanja raste sa 24 na 63 667. Ovo pokazuje da se sa povećanjem dužine sekvence znatno povećava i količina rezultata koju algoritam treba da formira, što doprinosi rastu vremena detekcije. U posmatranom skupu sintetičkih sekvenci broj supermaksimalnih ponavljanja povećava se brže od dužine ulazne sekvence.

U apsolutnim vrednostima detekcija maksimalnih ponavljanja ostaje najzahtevniji pojedinačni deo obrade. Na sekvenci dužine 500 000 baza ona čini približno 54,5% ukupnog vremena, dok izgradnja ESA čini približno 31,1%. Detekcija supermaksimalnih ponavljanja traje znatno kraće. Pri tome treba imati u vidu da ona u konfiguraciji `final` koristi već izgrađeno stablo LCP intervala, dok je vreme njegove izgradnje uključeno u vreme detekcije maksimalnih ponavljanja. Zbog toga ova dva vremena nisu direktno uporediva kao vremena dva potpuno nezavisna algoritma.

11.2 Faze izgradnje ESA

Slika 11.2 odvojeno prikazuje vremena četiri faze izgradnje ESA iz tabele 11.1.



Slika 11.2: Medijane vremena izvršavanja pojedinačnih faza izgradnje poboljšanog sufiksnog niza na sintetičkim sekvencama različitih dužina.

Konstrukcija sufiksnog niza zauzima najveći deo vremena izgradnje ESA u svim posmatranim slučajevima, sa udelom između 71,6% i 80,0%. Zbog tako velikog udela, izbor algoritma za konstrukciju sufiksnog niza može značajno uticati na ukupno vreme izgradnje ESA, zbog čega je ova faza zasebno eksperimentalno poređena.

Niz najdužih zajedničkih prefiksa predstavlja drugu vremenski najzahtevniju fazu izgradnje ESA. Njegov udeo u ukupnom vremenu izgradnje povećava se sa približno 13,4% za sekvencu dužine 10 000 na 21,0% za sekvencu dužine 500 000. Iako Kasaijev algoritam ima linearnu vremensku složenost, u posmatranim merenjima vreme njegove izgradnje povećava se nešto brže od vremena izgradnje inverznog sufiksnog niza i BWT-a.

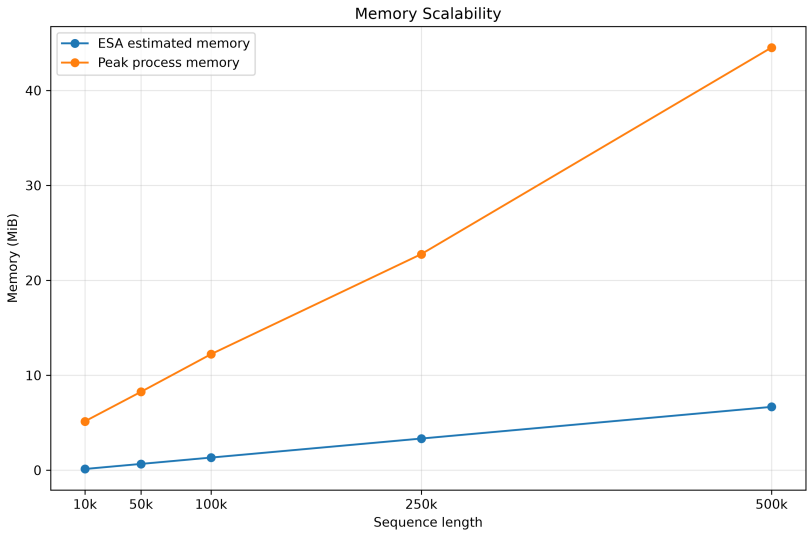
Inverzni sufiksni niz i Barouz–Vilerova transformacija zajedno čine manje od 4,5% ukupnog vremena izgradnje ESA u svim posmatranim slučajevima. Njihova izgradnja zato predstavlja mali deo ukupnog vremena potrebnog za formiranje poboljšanog sufiksnog niza.

11.3 Potrošnja memorije i broj pronađenih ponavljanja

Tabela 11.2 prikazuje potrošnju memorije i broj pronađenih ponavljanja za iste sintetičke sekvence. Promene ovih vrednosti sa povećanjem dužine sekvence prikazane su na slikama 11.3 i 11.4.

Tabela 11.2: Potrošnja memorije i broj pronađenih ponavljanja na sintetičkim sekvencama, uz minimalnu dužinu ponavljanja 10. Poslednja kolona prikazuje odnos broja supermaksimalnih i maksimalnih ponavljanja.

Dužina	ESA (MiB)	Maks. mem. (MiB)	Maks. mem. (B/bazi)	Maks.	Supermaks.	Odnos
10 000	0,13	5,16	540,7	24	24	1,000
50 000	0,67	8,26	173,3	883	874	0,990
100 000	1,34	12,23	128,2	3 383	3 291	0,973
250 000	3,34	22,76	95,5	20 389	18 761	0,920
500 000	6,68	44,52	93,4	74 795	63 667	0,851

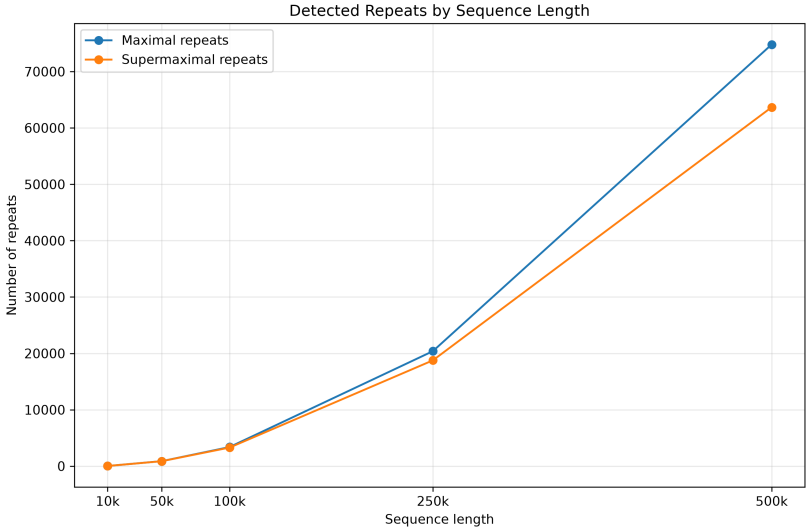


Slika 11.3: Procenjena memorija strukture ESA i maksimalna potrošnja memorije procesa u zavisnosti od dužine sekvence.

Procenjena memorija ESA raste proporcionalno dužini sekvence i iznosi približno 14 bajtova po bazi, u skladu sa procenom iz odeljka 6.2. Maksimalna potrošnja memorije procesa takođe raste sa dužinom ulaza, ali obuhvata i memoriju koju koriste ostali delovi programa i privremene strukture.

Odnos maksimalne potrošnje memorije procesa i procenjene memorije ESA smanjuje se sa približno 40 na najmanjem ulazu na oko 6,7 na najvećem. To pokazuje da kod manjih ulaza značajan deo ukupne potrošnje memorije potiče od delova

programa koji nisu obuhvaćeni procenom same ESA strukture.



Slika 11.4: Broj pronađenih maksimalnih i supermaksimalnih ponavljanja u zavisnosti od dužine sintetičke sekvence, uz minimalnu dužinu 10.

Slika 11.4 pokazuje da se broj pronađenih ponavljanja povećava brže od dužine sekvence u posmatranom opsegu. Pri povećanju dužine sa 100 000 na 250 000 broj maksimalnih ponavljanja povećava se približno 6,03 puta, dok se pri povećanju sa 250 000 na 500 000 povećava približno 3,67 puta. Ovi rezultati ukazuju na nadlinearan rast broja maksimalnih ponavljanja u posmatranim sintetičkim sekvencama.

Jedno moguće objašnjenje ovakvog rasta proizlazi iz načina generisanja sintetičkih sekvenci. Sa povećanjem dužine sekvence povećava se i broj mogućih parova početnih pozicija na kojima može započeti ponavljanje. Za dva nezavisno posmatrana segmenta ravnomerno generisane DNK sekvence, verovatnoća poklapanja prvih 10 simbola iznosi 4^{-10}. Zbog većeg broja mogućih parova pozicija, sa rastom sekvence povećava se i broj prilika za pojavu ponavljanja, što je u skladu sa rezultatima prikazanim na slici 11.4.

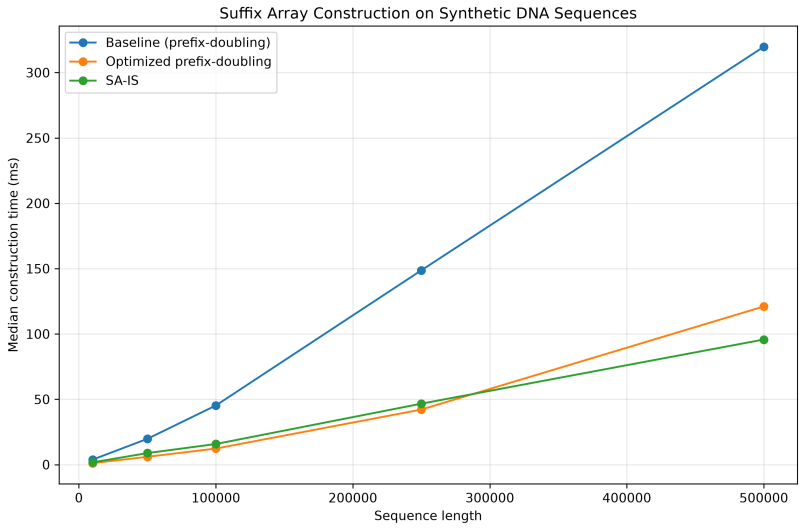
Odnos broja supermaksimalnih i maksimalnih ponavljanja opada sa 1 na 0,851 sa povećanjem dužine sekvence. Na sekvenci dužine 10 000 broj maksimalnih i supermaksimalnih ponavljanja je jednak, dok se na većim ulazima razlika između njihovih brojeva postepeno povećava. Odnos pritom ostaje manji ili jednak 1, što je u skladu sa definicijom supermaksimalnih ponavljanja iz odeljka 2.7: svako supermaksimalno ponavljanje jeste maksimalno, pa supermaksimalna ponavljanja čine podskup maksimalnih.

11.4 Poređenje algoritama za konstrukciju sufiksnog niza

Ovaj eksperiment poredi tri algoritma za konstrukciju sufiksnog niza, dok se osnovni algoritam za pronalaženje maksimalnih ponavljanja ne menja. Zbog toga se za poređenje koristi vreme iz kolone `suffix_array_time_ms`. Rezultati na sintetičkim sekvencama prikazani su u tabeli 11.3 i na slikama 11.5 i 11.6.

Tabela 11.3: Medijane vremena konstrukcije sufiksnog niza na sintetičkim sekvencama, u milisekundama, i ubrzanja algoritma SA-IS.

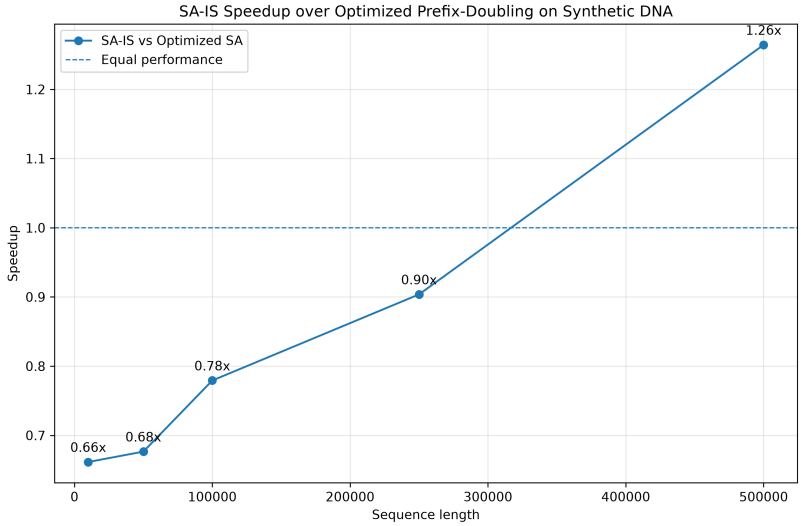
Dužina	Osnovni	Optimizovani	SA-IS	Ubrz. prema osn.	Ubrz. prema opt.
10 000	3,82	1,15	1,73	2,20×	0,66×
50 000	19,74	5,98	8,84	2,23×	0,68×
100 000	45,28	12,31	15,80	2,87×	0,78×
250 000	148,72	42,16	46,66	3,19×	0,90×
500 000	319,79	121,05	95,74	3,34×	1,26×



Slika 11.5: Medijane vremena konstrukcije sufiksnog niza na sintetičkim DNK sekvencama za tri implementacije.

Osnovna implementacija je najsporija na svim posmatranim dužinama, a njeno vreme raste brže od vremena preostale dve implementacije. Optimizovano udvostručavanje prefiksa brže je od algoritma SA-IS na prve četiri dužine, dok na sekvenci od 500 000 baza SA-IS postaje brži.

Ubrzanje algoritma SA-IS u odnosu na optimizovano udvostručavanje raste od 0,66 do 1,26 i prelazi vrednost 1,0 između 250 000 i 500 000 baza. Ovakav rezultat je



Slika 11.6: Ubrzanje algoritma SA-IS u odnosu na optimizovano udvostručavanje prefiksa na sintetičkim sekvencama. Vrednost 1,0 označava jednako vreme izvršavanja oba algoritma.

u skladu sa teorijskim složenostima: SA-IS ima linearnu vremensku složenost, dok optimizovano udvostručavanje ima složenost $\mathcal{O}(n \log n)$. Na manjim ulazima razlika u složenosti još nije dovoljna da nadmaši praktične troškove algoritma SA-IS, dok se njegova prednost pojavljuje na najvećem sintetičkom ulazu.

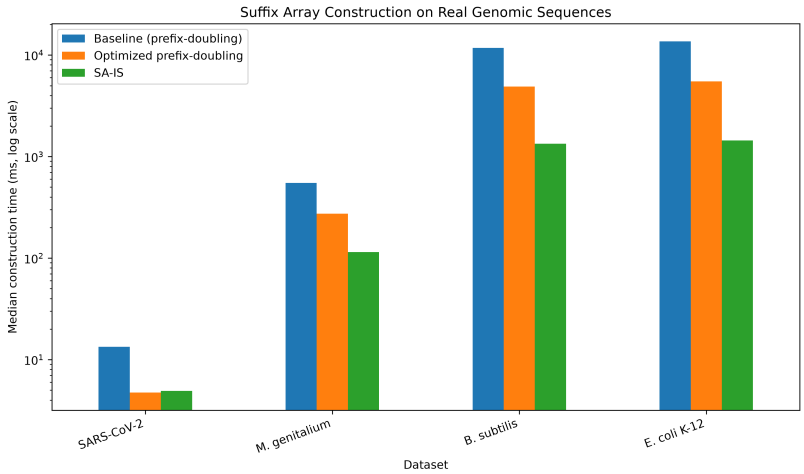
Isto poređenje sprovedeno je i na četiri realna genoma. Rezultati su prikazani u tabeli 11.4 i na slikama 11.7 i 11.8.

Tabela 11.4: Medijane vremena konstrukcije sufiksnog niza na realnim genomima, u milisekundama, i ubrzanja algoritma SA-IS.

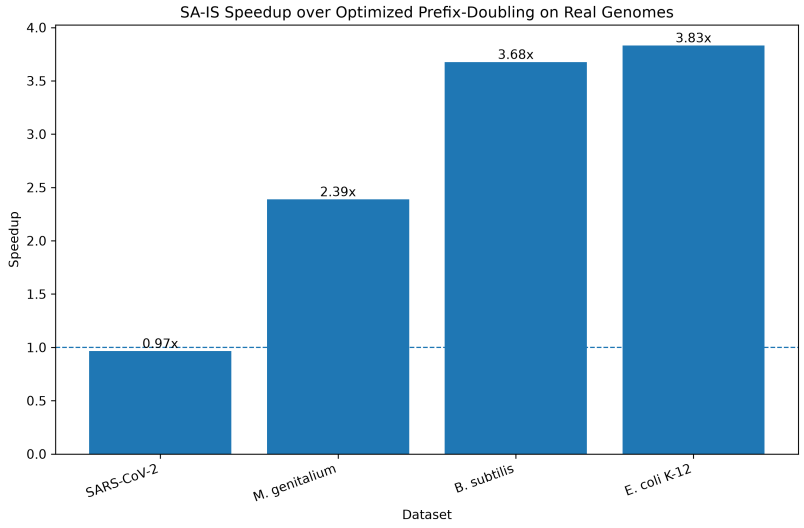
Genom	Dužina	Osnovni	Optimiz.	SA-IS	Ubrz. prema osn.	Ubrz. prema opt.
SARS-CoV-2	29 903	13,34	4,74	4,91	2,72×	0,97×
M. genitalium	580 076	550,31	273,75	114,59	4,80×	2,39×
B. subtilis	4 215 606	11 738,70	4 908,29	1 335,01	8,79×	3,68×
E. coli K-12	4 641 652	13 675,40	5 510,19	1 437,73	9,51×	3,83×

Na najmanjem genomu, SARS-CoV-2, SA-IS je neznatno sporiji od optimizovanog udvostručavanja prefiksa, slično kao na manjim sintetičkim ulazima. Na većim genomima SA-IS postaje znatno brži. Na genomu *M. genitalium*, dužine 580 076 baza, ubrzanje u odnosu na optimizovano udvostručavanje iznosi 2,39 puta. Na genomima *B. subtilis* i *E. coli*, sa više od četiri miliona baza, ubrzanje raste na 3,68, odnosno 3,83 puta. U odnosu na osnovnu implementaciju, najveće izmereno ubrzanje iznosi 9,51 puta.

Rezultati na realnim genomima razlikuju se od rezultata na sintetičkim sekven-



Slika 11.7: Medijane vremena konstrukcije sufiksnog niza na realnim genomima. Vertikalna osa je u logaritamskoj razmeri zbog velikog raspona vrednosti.



Slika 11.8: Ubrzanje algoritma SA-IS u odnosu na optimizovano udvostručavanje prefiksa na realnim genomima.

cama slične dužine. Na sintetičkoj sekvenci od 500 000 baza SA-IS je 1,26 puta brži od optimizovanog udvostručavanja, dok na genomu *M. genitalium*, dužine 580 076 baza, ubrzanje iznosi 2,39 puta. Ova razlika pokazuje da sama dužina ulaza nije dovoljna da objasni praktične performanse algoritama. Mogući uticaj strukture i sastava sekvence razmatra se u odeljku 12.1.

11.5 Poređenje algoritama za pronalaženje maksimalnih ponavljanja

Ovaj eksperiment poredi osnovni i optimizovani algoritam za pronalaženje maksimalnih ponavljanja, pri čemu obe konfiguracije koriste sufiksni niz konstruisan algoritmom SA-IS. Zbog toga se za poređenje koristi vreme iz kolone `repeat_detection_time_ms`. Rezultati na sintetičkim sekvencama prikazani su u tabeli 11.5 i na slikama 11.9 i 11.10.

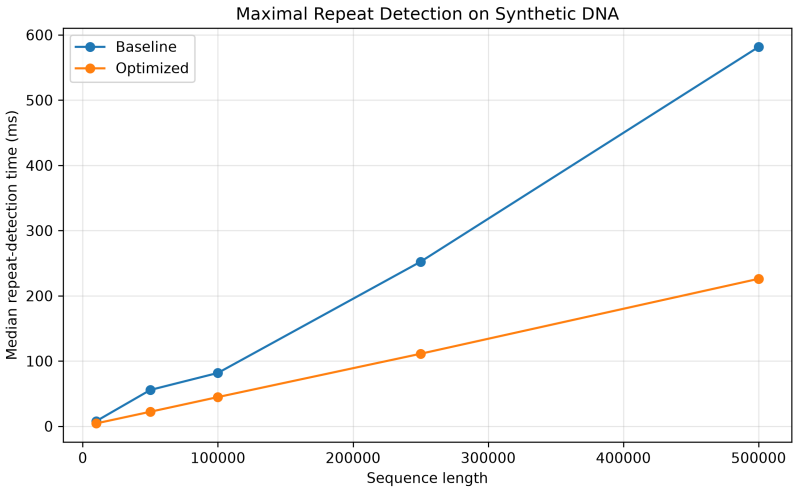
Tabela 11.5: Detekcija maksimalnih ponavljanja na sintetičkim sekvencama. Vremena predstavljaju medijane u milisekundama. Poslednja kolona prikazuje procentualnu razliku u maksimalnoj potrošnji memorije procesa optimizovane u odnosu na osnovnu implementaciju.

Dužina	Broj	Osnovni	Optimizovani	Ubrzanje	Ušteda vremena	Promena memorije
10 000	24	7,95	4,43	1,79×	44,2%	−5,1%
50 000	883	55,71	22,28	2,50×	60,0%	+7,5%
100 000	3 383	81,79	44,77	1,83×	45,3%	+0,6%
250 000	20 389	252,36	111,20	2,27×	55,9%	+1,2%
500 000	74 795	581,55	226,10	2,57×	61,1%	−1,1%

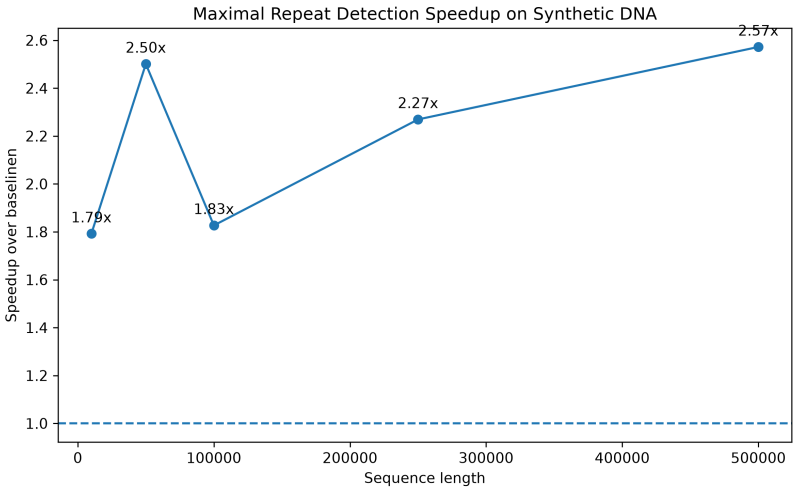
Optimizovani algoritam je brži na svim posmatranim dužinama, sa ubrzanjem od 1,79 do 2,57 puta i smanjenjem vremena izvršavanja od 44,2% do 61,1%. Ubrzanje ne raste monotono sa dužinom sekvence: veće vrednosti dobijene su za 50 000 i 500 000 baza, dok su manje vrednosti dobijene za 10 000 i 100 000 baza. Pošto je za svaku dužinu korišćena jedna sintetička sekvenca, ovaj eksperiment ne omogućava da se uzrok tih odstupanja posebno izdvoji.

Broj pronađenih ponavljanja identičan je za obe implementacije na svakoj dužini. Skripta za analizu to eksplicitno proverava i prekida rad ako se brojevi razlikuju. Ova provera potvrđuje da optimizacija nije promenila broj prijavljenih maksimalnih ponavljanja na posmatranim skupovima podataka.

Razlika u maksimalnoj potrošnji memorije je mala i menja znak, od −5,1% do +7,5%. Optimizovani algoritam koristi dodatne maske levih konteksta, ali istovre-



Slika 11.9: Medijane vremena detekcije maksimalnih ponavljanja na sintetičkim sekvencama za osnovni i optimizovani algoritam.



Slika 11.10: Ubrzanje optimizovanog algoritma za maksimalna ponavljanja u odnosu na osnovni algoritam na sintetičkim sekvencama.

meno izbegava eksplicitno čuvanje maksimalnih ponovljenih parova i skupa različitih niski. Dobijeni rezultati pokazuju da na sintetičkim sekvencama nijedna od dve implementacije nema dosledno manju maksimalnu potrošnju memorije.

Isto poređenje sprovedeno je i na četiri realna genoma. Rezultati su prikazani u tabeli 11.6 i na slikama 11.11 i 11.12.

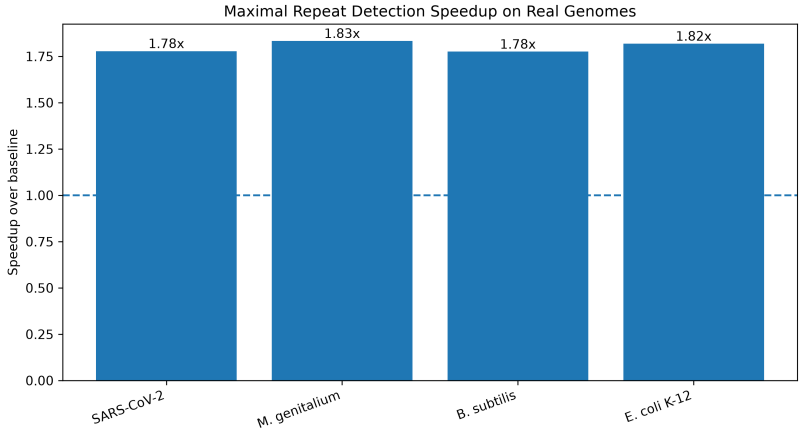
Tabela 11.6: Detekcija maksimalnih ponavljanja na realnim genomima. Vremena predstavljaju medijane u milisekundama. Poslednja kolona prikazuje procentualnu razliku u maksimalnoj potrošnji memorije procesa optimizovane u odnosu na osnovnu implementaciju.

Genom	Broj	Osnovni	Optimizovani	Ubrzanje	Ušteda vremena	Promena memorije
SARS-CoV-2	804	26,68	15,00	1,78×	43,8%	+5,5%
M. genitalium	596	483,72	263,90	1,83×	45,4%	+8,2%
B. subtilis	622	3 403,66	1 917,43	1,78×	43,7%	+2,9%
E. coli K-12	2 048	3 500,23	1 925,06	1,82×	45,0%	+10,7%

Slika 11.11: Medijane vremena detekcije maksimalnih ponavljanja na realnim genomima za osnovni i optimizovani algoritam.

Na realnim genomima ubrzanje je veoma ujednačeno i kreće se u uskom opsegu od 1,78 do 1,83 puta, dok smanjenje vremena izvršavanja iznosi između 43,7% i 45,4%. Slične vrednosti dobijene su na genomima čije se dužine razlikuju za više od dva reda veličine, što pokazuje da je ubrzanje dosledno u svim posmatranim skupovima podataka. Broj pronađenih ponavljanja identičan je za obe implementacije na svakom od četiri genoma.

Razlika u maksimalnoj potrošnji memorije je na svim realnim genomima pozitivna i kreće se između +2,9% i +10,7%. Optimizovani algoritam koristi dodatne maske levih konteksta za čvorove stabla LCP intervala, pa je povećanje memorije u odnosu na osnovni algoritam očekivano. Ipak, tačan doprinos pojedinačnih pomoć-



Slika 11.12: Ubrzanje optimizovanog algoritma za maksimalna ponavljanja u odnosu na osnovni algoritam na realnim genomima.

nih struktura ukupnoj potrošnji memorije nije posebno meren.

11.6 Ponavljanja u realnim genomima

Tabela 11.7 prikazuje osnovne karakteristike ponavljanja pronađenih u četiri realna genoma. Vrednosti su izračunate iz izlaznih CSV datoteka u direktorijumu `data/processed/repeats/real/`.

Tabela 11.7: Ponavljanja pronađena u realnim genomima. Kolona z daje ukupan broj prijavljenih pozicija pojavljivanja maksimalnih ponavljanja.

Genom	Min. dužina ponavljanja	Maks.	Supermaks.	Najduže	Medijana dužine	z
SARS-CoV-2	10	804	749	32	10	1 909
M. genitalium	20	596	274	243	32,5	1 924
B. subtilis	20	622	397	2 983	40	2 018
E. coli K-12	20	2 048	894	2 815	29	9 889

Broj maksimalnih ponavljanja ne prati direktno dužinu genoma. Genom *Bacillus subtilis*, sa više od četiri miliona baza, ima 622 maksimalna ponavljanja dužine najmanje 20, dok genom *Mycoplasma genitalium*, čiji je genom oko 7,3 puta kraći, ima 596. Ovi rezultati pokazuju da broj pronađenih ponavljanja ne zavisi samo od dužine genoma, već i od strukture njegove sekvence.

Genom *Escherichia coli* se izdvaja sa 2 048 maksimalnih ponavljanja i $z = 9\,889$ prijavljenih pozicija pojavljivanja. Broj prijavljenih pozicija približno je pet puta veći nego kod ostala tri genoma.

U posmatranim skupovima podataka najduža ponavljanja pronađena su u većim bakterijskim genomima. U genomu *SARS-CoV-2* najduže pronađeno ponavljanje

ima 32 baze, dok u genomima *B. subtilis* i *E. coli* najduža ponavljanja dostižu 2 983, odnosno 2 815 baza.

Odnos broja supermaksimalnih i maksimalnih ponavljanja iznosi približno 0,93 za *SARS-CoV-2*, 0,46 za *M. genitalium*, 0,64 za *B. subtilis* i 0,44 za *E. coli*. Niži odnos znači da manji deo maksimalnih ponavljanja istovremeno zadovoljava i uslove za supermaksimalno ponavljanje.

Za genom *E. coli* vrednost $z = 9\,889$ predstavlja svega oko 0,21% dužine genoma. To pokazuje da je u ovom primeru broj prijavljenih pozicija pojavljivanja mali u odnosu na dužinu ulazne sekvence.

Glava 12

Diskusija

Prethodna glava prikazala je izmerene rezultate, dok se u ovoj glavi oni tumače i povezuju sa svojstvima korišćenih algoritama i skupova podataka. Pri tome se, gde god je moguće, jasno razlikuju neposredno izmereni rezultati od mogućih objašnjenja uočenog ponašanja.

12.1 Kada SA-IS postaje bolji izbor

Na sintetičkim sekvencama SA-IS je sporiji od optimizovanog udvostručavanja prefiksa do dužine od 250 000 baza, dok na sekvenci od 500 000 baza postaje brži. Na realnim genomima njegova prednost pojavljuje se već pri 580 076 baza, gde je SA-IS 2,39 puta brži od optimizovanog udvostručavanja prefiksa.

Jedno moguće objašnjenje razlike na manjim ulazima proizlazi iz praktičnih troškova algoritma SA-IS. Iako ima linearnu vremensku složenost, njegova implementacija obuhvata više koraka: klasifikaciju L i S tipova, pronalaženje LMS pozicija, imenovanje LMS podniski i, kada je potrebno, rekurzivno rešavanje redukovanog problema. Kod manjih ulaza ti dodatni koraci mogu imati veći uticaj na ukupno vreme izvršavanja, dok optimizovano udvostručavanje koristi iterativni postupak u kojem se sufiksi razvrstavaju prema već izračunatim klasama ekvivalencije.

Drugi deo objašnjenja odnosi se na samu strukturu sekvence. Kod optimizovanog udvostručavanja sufiksi se razdvajaju postepeno: u svakoj rundi posmatra se prefiks dvostruko veće dužine nego u prethodnoj. Ako dva sufiksa imaju dug zajednički prefiks, potrebno je više rundi da bi algoritam mogao da ih razlikuje i svrsta u različite klase. Zbog toga broj rundi ne zavisi samo od dužine sekvence, već i od toga koliko su njeni sufiksi međusobno slični. Kod algoritma SA-IS ovaj postupak

udvostručavanja se ne koristi, pa najduži zajednički prefiks ne određuje broj koraka na isti način.

Dužine pronađenih ponavljanja pokazuju da se sintetičke i realne sekvence razlikuju po strukturi ponavljanja. U sintetičkoj sekvenci dužine 10 000 najduže maksimalno ponavljanje ima 15 baza, a u sekvenci dužine 500 000 baza 18. U realnim genomima najduža ponavljanja mogu biti znatno duža. Na primer, u genomu *M. genitalium* najduže pronađeno maksimalno ponavljanje ima 243 baze, a u genomu *B. subtilis* 2 983 baze. Takvi duži zajednički delovi sufiksa mogu zahtevati više rundi kod algoritma udvostručavanja prefiksa pre nego što svi sufiksi dobiju različite klase.

Kod sintetičke sekvence od 500 000 baza, sa najdužim maksimalnim ponavljanjem dužine 18, potrebno je približno pet rundi udvostručavanja da bi posmatrani prefiks dostigao dužinu 32. Kod genoma *B. subtilis*, gde najduže maksimalno ponavljanje ima 2 983 baze, potrebno je približno dvanaest rundi da bi posmatrani prefiks dostigao dužinu 4 096. SA-IS ne koristi ovakav postupak, pa dužina zajedničkih prefiksa ne utiče na broj njegovih koraka na isti način. Ova razlika može biti jedno od objašnjenja za izraženiju prednost algoritma SA-IS na realnim genomima.

Praktičan zaključak je da optimizovano udvostručavanje prefiksa predstavlja dobar izbor za manje ulaze, gde je u eksperimentima često bilo brže od algoritma SA-IS. Sa povećanjem dužine ulaza prednost algoritma SA-IS postaje izraženija, naročito na većim realnim genomima. Pošto je cilj ovog rada obrada DNK sekvenci koje mogu imati više miliona baza, izbor algoritma SA-IS u konačnoj konfiguraciji je opravdan.

12.2 Uticaj konstantnih faktora na manjim ulazima

Na manjim ulazima pojedinačne razlike u vremenu izvršavanja imaju veći relativni uticaj na rezultat. Za sintetičku sekvencu dužine 10 000 baza ukupno vreme obrade iznosi oko 10 ms, pa i mali dodatni vremenski troškovi mogu činiti primetan deo ukupnog vremena.

Kao što pokazuju tabela 11.3 i slika 11.6, ubrzanje algoritma SA-IS u odnosu na optimizovano udvostručavanje prefiksa raste od 0,66 na 10 000 baza do 1,26 na 500 000 baza. Ovaj rezultat ukazuje da praktični troškovi algoritma SA-IS imaju veći relativni značaj na manjim ulazima, dok se njegova bolja asimptotska složenost izraženije ispoljava sa povećanjem dužine sekvence.

12.3 Ponašanje na velikim realnim genomima

Na genomima od preko četiri miliona baza razlike između algoritama postaju izražene i praktično značajne. Konstrukcija sufiksnog niza za genom *E. coli* traje 13,68 sekundi osnovnom implementacijom, 5,51 sekundi optimizovanim udvostručavanjem prefiksa i 1,44 sekunde algoritmom SA-IS. Razlika između najsporije i najbrže implementacije iznosi više od 12 sekundi, što pokazuje da izbor algoritma za konstrukciju sufiksnog niza ima veliki uticaj na vreme izgradnje poboljšanog sufiksnog niza kod velikih genoma.

Rezultati za *B. subtilis* i *E. coli* veoma su slični, iako se ova dva genoma razlikuju po GC sadržaju, koji iznosi približno 43,5%, odnosno 50,8%. Pošto su njihove dužine i vremena konstrukcije sufiksnog niza slični, iz ovog poređenja se ne vidi da razlika u GC sadržaju ima izražen uticaj na vreme konstrukcije.

Broj pronađenih ponavljanja, međutim, znatno se razlikuje. U genomu *E. coli* pronađeno je 2 048 maksimalnih ponavljanja dužine najmanje 20, dok ih genom *B. subtilis* ima 622. Iako su ova dva genoma sličnih dužina, broj pronađenih ponavljanja razlikuje se više od tri puta. To pokazuje da broj ponavljanja ne zavisi samo od dužine genoma, već i od strukture same sekvence.

12.4 Poboljšanje detekcije maksimalnih ponavljanja

Na svim posmatranim skupovima podataka optimizovani algoritam postiže ubrzanje između 1,78 i 2,57 puta u odnosu na osnovni algoritam. Na realnim genomima vrednosti su posebno ujednačene i kreću se u uskom opsegu od 1,78 do 1,83 puta, iako se dužine ovih genoma znatno razlikuju.

Takva ujednačenost ubrzanja znači da se prednost optimizovanog algoritma ne povećava primetno sa dužinom genoma. Da je kvadratni najgori slučaj osnovnog algoritma imao sve veći uticaj na vreme izvršavanja, očekivalo bi se da ubrzanje optimizovanog algoritma raste sa dužinom sekvence. Takav trend u ovim merenjima nije uočen. Optimizovani pristup izbegava eksplicitno nabrajanje maksimalnih ponovljenih parova i dodatne operacije prisutne u osnovnoj implementaciji, kao što su kopiranje niski, ponovno građenje stabla LCP intervala i rad sa uređenim skupom niski. Ove razlike u implementaciji predstavljaju moguće objašnjenje za uočeno smanjenje vremena izvršavanja.

Jedno moguće objašnjenje za to što ubrzanje ne raste primetno sa dužinom genoma jeste mali broj pojavljivanja pojedinačnih ponavljanja. Osnovni algoritam eksplicitno formira parove njihovih pojavljivanja, pa broj potrebnih operacija može naglo porasti kada se isto ponavljanje javlja na velikom broju pozicija. Na primer, za genom *B. subtilis* 622 maksimalna ponavljanja imaju ukupno 2 018 prijavljenih pozicija, odnosno prosečno oko 3,2 pojavljivanja po ponavljanju. Ovaj rezultat sugeriše da se nepovoljni slučajevi sa veoma velikim brojem pojavljivanja istog ponavljanja u ovom skupu podataka ne javljaju često.

Na osnovu načina rada osnovnog algoritma može se pretpostaviti da bi prednost optimizovanog pristupa bila veća na ulazima kod kojih se ista ponavljanja javljaju na velikom broju pozicija, jer bi tada osnovni algoritam morao da formira znatno više parova pojavljivanja. Takvi ulazi nisu posebno ispitani u ovom radu, pa bi njihova analiza mogla predstavljati pravac daljeg istraživanja.

12.5 Odnos vremena izvršavanja i potrošnje memorije

Prethodni rezultati pokazali su da optimizovani algoritam smanjuje vreme detekcije maksimalnih ponavljanja, ali njegov uticaj na potrošnju memorije nije jednoznačan. Optimizovani algoritam za svaki čvor stabla LCP intervala čuva masku od 256 bitova za predstavljanje levih konteksta. Izmerena razlika u maksimalnoj potrošnji memorije na realnim genomima kreće se između +2,9% i +10,7%, dok je na sintetičkim podacima između −5,1% i +7,5%.

Odnos između vremena izvršavanja i potrošnje memorije zato nije jednostavan. Optimizovani pristup koristi dodatne maske levih konteksta, ali istovremeno izbegava eksplicitno čuvanje maksimalnih ponovljenih parova i skupa kopiranih niski koje koristi osnovna implementacija. Zbog toga razlika u maksimalnoj potrošnji memorije zavisi od konkretnog ulaza. Na sintetičkim sekvencama optimizovana implementacija u nekim slučajevima koristi nešto više, a u nekim nešto manje memorije od osnovne.

Za genom *E. coli* maksimalna potrošnja memorije procesa iznosi oko 306 MiB, dok procenjena memorija glavnih struktura ESA iznosi oko 62 MiB. Razlika pokazuje da značajan deo ukupne memorije procesa potiče i od drugih struktura i privremenih podataka koji nisu obuhvaćeni procenom ESA. Pošto se maksimalna potrošnja

memorije meri na nivou celog procesa, iz tih vrednosti nije moguće precizno izdvojiti doprinos pojedinačnih struktura, kao što su maske levih konteksta.

12.6 Ograničenja eksperimenata

Sva merenja izvedena su na jednoj mašini i korišćenjem jednog prevodioca, pa izmerena vremena zavise i od konkretnog hardverskog i softverskog okruženja. Zbog toga se vremena izvršavanja izmerena u ovom radu ne mogu neposredno porediti sa vremenima dobijenim u drugačijim eksperimentalnim uslovima.

Svaka konfiguracija izvršena je pet puta, a za predstavljanje rezultata korišćena je medijana, koja smanjuje uticaj pojedinačnih odstupanja u merenjima. Ipak, tako mali broj ponavljanja ne omogućava pouzdanu procenu varijabilnosti vremena izvršavanja.

Za svaku dužinu sintetičke sekvence korišćena je samo jedna sekvenca, generisana sa semenom 42. Rezultati zato opisuju ponašanje algoritama na konkretnim generisanim sekvencama, a ne prosečno ponašanje nad većim brojem slučajnih uzoraka. To može biti jedan od razloga za nemonotono kretanje ubrzanja prikazano u tabeli 11.5, ali taj uticaj nije posebno ispitan.

Eksperimentalne serije sprovedene su odvojeno, pa se vremena izvršavanja dobijena u različitim serijama ne mogu neposredno međusobno porediti. Zbog toga se zaključci o relativnim performansama u radu zasnivaju na poređenjima unutar iste eksperimentalne serije.

Skup realnih podataka obuhvata jedan virusni i tri bakterijska genoma, bez eukariotskih genoma. Dobijeni rezultati se zato ne mogu bez dodatnih eksperimenata generalizovati na sve vrste genoma i znatno veće sekvence.

12.7 Ograničenja implementacije

Implementacija pronalazi isključivo egzaktna direktna ponavljanja. Približna ponavljanja, koja mogu sadržati nepoklapanja (engl. *mismatches*), umetanja ili brisanja, nisu podržana, kao ni ponavljanja u obliku obrnutog komplementa, koja su takođe značajna u analizi DNK. Čitač FASTA datoteka prihvata tačno jedan zapis, pa obrada više sekvenci iz iste FASTA datoteke u jednom pokretanju nije podržana.

U FASTA zapisima `N` označava nepoznatu bazu, ali ga implementacija ne tumači kao proizvoljan simbol iz skupa `A`, `C`, `G` i `T`. Simbol `N` u implementaciji se tretira kao

običan simbol i poklapa se samo sa drugim simbolom `N`. U genomima korišćenim u eksperimentima simbol `N` se ne javlja.

Indeksi u implementaciji čuvaju se tipom `int`, pa je maksimalna teorijski podržana dužina sekvence ograničena na približno 2,1 milijardu pozicija. To je dovoljno za genome korišćene u ovom radu, ali nije dovoljno za sekvence dužine ljudskog genoma, koji ima približno tri milijarde baza. Korišćenje 64-bitnih indeksa omogućilo bi obradu znatno dužih sekvenci, ali bi nizovi koji čuvaju indekse zauzimali približno dvostruko više memorije. Merenje maksimalne potrošnje memorije trenutno je implementirano samo za sistem Windows.

Struktura ESA trajno čuva inverzni sufiksni niz, iako se on nakon konstrukcije LCP niza više ne koristi u algoritmima za pronalaženje ponavljanja. Kada bi se nakon konstrukcije LCP niza oslobodila memorija zauzeta inverznim sufiksnim nizom, procenjena memorija glavnih ESA struktura smanjila bi se sa približno $14n$ na $10n$ bajtova.

Pored toga, za svako pronađeno ponavljanje trenutno se čuva i sama niska ponavljanja. Za *B. subtilis* zbir dužina svih pronađenih maksimalnih ponavljanja iznosi 117 514 simbola. Kada bi se umesto cele niske čuvale samo njena početna pozicija i dužina, smanjila bi se memorija potrebna za čuvanje rezultata.

Glava 13

Zaključak

Cilj ovog rada bio je da se ispita primena poboljšanog sufiksnog niza za pronalaženje maksimalnih i supermaksimalnih egzaktnih ponavljanja u DNK sekvencama, sa posebnim osvrtom na praktične performanse različitih algoritama za konstrukciju sufiksnog niza i pronalaženje maksimalnih ponavljanja.

Eksperimentalni rezultati pokazali su da izbor algoritma za konstrukciju sufiksnog niza značajno utiče na vreme njegove izgradnje. Na manjim sintetičkim sekvencama optimizovano udvostručavanje prefiksa bilo je brže od algoritma SA-IS, dok je na najvećoj sintetičkoj sekvenci od 500 000 baza SA-IS postao brži. Na realnim genomima njegova prednost bila je izraženija: SA-IS je bio 2,39 puta brži od optimizovanog udvostručavanja na genomu dužine 580 076 baza, odnosno 3,83 puta brži na najvećem analiziranom genomu od 4 641 652 baze. U odnosu na osnovnu implementaciju udvostručavanja prefiksa, najveće izmereno ubrzanje iznosilo je 9,51 puta.

Rezultati takođe pokazuju da praktične performanse algoritama ne zavise samo od dužine ulaza. Sintetičke i realne sekvence sličnih dužina davale su različite odnose vremena izvršavanja posmatranih algoritama. Jedno moguće objašnjenje jeste struktura same sekvence, jer duži zajednički prefiksi sufiksa mogu zahtevati veći broj koraka kod algoritama zasnovanih na udvostručavanju prefiksa. Zbog toga se procena algoritama za obradu genomskih sekvenci ne može zasnivati samo na dužini ulazne sekvence.

Optimizovani postupak za pronalaženje maksimalnih ponavljanja takođe je pokazao poboljšanje u odnosu na osnovni pristup. Na sintetičkim podacima izmereno ubrzanje kretalo se između 1,79 i 2,57 puta, dok je na realnim genomima iznosilo između 1,78 i 1,83 puta. To pokazuje da za pronalaženje maksimalnih ponavljanja

nije neophodno eksplicitno nabrajati maksimalne ponovljene parove, što je u našoj implementaciji dovelo do kraćeg vremena izvršavanja.

Na realnim genomima optimizovani postupak koristio je nešto više memorije od osnovnog, pri čemu je maksimalna potrošnja memorije procesa bila veća za 2,9% do 10,7%. Procenjena memorija osnovnih struktura poboljšanog sufiksnog niza rasla je linearno sa dužinom ulazne sekvence i iznosila približno 14 bajtova po bazi.

Analiza pronađenih ponavljanja pokazala je da njihov broj ne zavisi samo od dužine genoma. Na primer, *Escherichia coli* ima 2 048 maksimalnih ponavljanja dužine najmanje 20, dok *Bacillus subtilis*, čiji je genom slične dužine, ima 622. Ova razlika pokazuje da na broj pronađenih ponavljanja utiče i struktura same sekvence.

Dobijene rezultate treba posmatrati u okviru ograničenja ovog rada. Razmatrana su isključivo egzaktna direktna ponavljanja, dok približna ponavljanja i ponavljanja u obliku obrnutog komplementa nisu obuhvaćena. Eksperimenti su izvedeni na jednom računarskom okruženju i na ograničenom skupu podataka koji obuhvata jedan virusni i tri bakterijska genoma. Za svaku dužinu sintetičkog ulaza korišćena je jedna generisana sekvenca, pa rezultati ne predstavljaju prosek nad većim brojem slučajnih uzoraka. Zbog toga se zaključci o performansama ne mogu bez dodatnih eksperimenata generalizovati na druge vrste genoma i znatno veće sekvence.

Literatura

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. SRC Research Report 124, Digital Equipment Corporation, 1994.

[3] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[4] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, 2003.

[5] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching (CPM)*, Lecture Notes in Computer Science, 2001.

[6] Stefan Kurtz, Jomuna V. Choudhuri, Enno Ohlebusch, Chris Schleiermacher, Jens Stoye, and Robert Giegerich. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29(22):4633–4642, 2001.

[7] Stefan Kurtz and Chris Schleiermacher. REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5):426–427, 1999.

[8] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[9] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[10] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference (DCC)*, 2009.

[11] Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

[12] Enno Ohlebusch and Timo Beller. Alphabet-independent algorithms for finding context-sensitive repeats in linear time. *Journal of Discrete Algorithms*, 34:23–36, 2015.

[13] Simon J. Puglisi, William F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), 2007.

[14] Todd J. Treangen and Steven L. Salzberg. Repetitive DNA and next-generation sequencing: computational challenges and solutions. *Nature Reviews Genetics*, 13(1):36–46, 2012.

[15] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[16] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory*, 1973.

Upotreba alata zasnovanih na veštačkoj inteligenciji

Tokom izrade rada korišćen je alat ChatGPT. Korišćen je za dodatna objašnjenja pojmova koji nisu bili dovoljno jasni iz korišćenih izvora. Takođe je korišćen kao pomoć pri pojedinim delovima implementacije, kao što su skripte za generisanje i analizu podataka, pisanje testova, čitanje i parsiranje FASTA datoteka, kao i pri otklanjanju pojedinih problema u algoritmima. Alat je korišćen i kao pomoć pri organizaciji i strukturi rada, kao i za proveru i ispravljanje gramatičkih, stilskih i pravopisnih grešaka u tekstu.

Svi predlozi i informacije dobijeni na ovaj način pažljivo su provereni pre uključivanja u rad i prilagođeni konkretnim zahtevima rada i postojećoj implementaciji. Predlozi nisu prihvatani bez provere, već su po potrebi menjani i usklađivani sa korišćenim izvorima, implementacijom i rezultatima testiranja. Za sadržaj rada, tačnost navedenih informacija i konačnu verziju implementacije u potpunosti je odgovorna autorka rada.