SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 5745

Landau-Vishkin-Nussinov algoritam za poravnanje dva niza

Donik Vršnak

Zagreb, lipanj 2018.

SADRŽAJ

1. Uvod

Problem pronalaženja homolognih poklapanja unutar dva niza je problem koji je značajan za mnoga područja moderne znanosti, a pogotovo je važan u području bioinformatike. Zbog brzog razvoja različitih metoda sekvenciranja genoma, koje postaju sve jeftinije i pristupačnije, u bioinformatici se javila potreba za razvojem algoritama koji mogu efikasno rješavati problem pronalaženja poravnanja homolognih preklapanja unutar genoma. Sekvenciranjem genoma dobivamo sekvence koje je moguće jednostavno prikazati u računalu, jer se svaka aminokiselina može preslikati u jedan znak. Drugim riječima, jedno očitanje molekule DNK se može preslikati u niz znakova. Time se problem pronalaženja sličnih dijelova sekvenci dobivenih sekvenciranjem DNK pretvara u problem pronalaženja preklapanja unutar dva niza znakova. Također, vidljivo je da se ovakvi algoritmi mogu primjenjivati na bilo koja dva niza znakova, no u ovom radu fokusirat ćemo se na njihovu primjenu u bioinformatici i preklapanju sekvenci DNK.

Iako se na prvi pogled taj problem može činiti jednostavnim, do problema dolazimo kada u obzir uzmemo moguće duljine sekvenci koje mogu imati od nekoliko desetaka proteinskih baza, pa i do čak nekoliko stotina tisuća proteinskih baza, a da ukupni broj baza u svim sekvencama može biti i do nekoliko desetaka milijardi. Ako bi problem probali riješiti nekim jednostavnim algoritmom koji bi, ako uspoređujemo dvije sekvence od kojih jedna ima duljinu m, a druga duljinu n, njegova složenost bila bi $O(nm)$, vidljivo je da bi vrijeme potrebno za rješavanje bilo preveliko. Zbog toga dolazimo do potrebe za efikasnijim algoritmima poravnanja, koje možemo podijeliti na heurističke i analitičke. Heuristički algoritmi se baziraju na procjeni poravnanja korištenjem različitih heurističkih funkcija te su samim time nešto nepreciznji od analitičkih, ali tu nepreciznost nadoknađuju u brzini pronalaženja rješenja. S druge strane, analitički algoritmi uvode neka ograničenja na rješenja, te si time smanjuju ukupni prostor rješenja koje moraju pretražiti. Algoritam Landau-Vishkin-Nussinov spada u skupinu analitičkih algoritama te će dalje u radu biti razmatrani samo analitički algoritmi.

U suštini, problem traženja poravnanja između dva niza svodi se na traženje naj-

manjeg broja razlika između ta dva niza, pronalaska mjesta na kojima se te razlike pojavljuju te njihove vrste. Najčešća podjela razlika je na tri vrste:

1. brisanje znaka iz niza
2. umetanje znaka u niz
3. izmjena jednog znaka drugim znakom

Drugi naziv za broj razlika između dva niza je Levenshteinova udaljenost (eng. edit distance), koju ćemo dalje u radu, radi jednostavnosti, nazivati samo udaljenost. Udaljenost dva niza možemo promatrati i kao broj transformacija nad pojedinim znakovima koje je potrebno napraviti da bi se prvi niz pretvorio u drugi niz. Također, radi jednostavnosti uvest ćemo nazive za dva niza koja uspoređujemo, tako da ćemo kraći niz prozvati uzorak, a duži tekst. Na primjer, najmanja udaljenost između uzorka "Hello" i teksta "World!" je 5 jer sve znakove osim znaka *l* moramo promijeniti kako bi iz niza "Hello" dobili niz "World!".

1.1. Cilj rada

Cilj ovog rada bio je implementirati algoritam poravnanja dva niza koji su predložili Gad M. Landau, Uzi Vishkin i Ruth Nussinov (Landau et al., 1986). Također nakon implementacije, performanse implementacije algoritma smo uspoređivali s implementacijom Myers-ovog bit-vektor algoritma (Myers, 1999) koja se koristi u alatu za određivanje poravnanja parova nizova edlib (Šošić i Šikić, 2017). Kako je Myersov algoritam sporiji u slučajevima kada se radi o poravnanju kratkih nizova, dodatni cilj rada bio je integrirati implementirani algoritam u alat edlib, te odrediti jednostavnu heurističku funkciju koja će odrediti koju implementaciju je potrebno koristiti, u ovisnosti o duljinama ulaznih nizova.

2. Pregled pojmova

2.1. Metode poravnavanja

Metode poravnanja dijele se u tri glavne skupine: globalne, lokalne i polu-globalne. Ove metode razlikuju se u načinu na koji se penaliziraju transformacija unutar teksta. Kod lokalnog poravnanja tražimo slična područja unutar dva niza, odnosno uspoređujemo sličnost nizova s obzirom na podnizove određenih duljina. Najpoznatiji takav algoritam je Smith-Watermanov algoritam (Smith i Waterman, 1981), koji koristi dinamičko programiranje za pronalaženje optimalnog lokalnog poravnanja. S druge strane, u globalnim metodama provodi se uspoređivanje nizova na temelju pojedinačnih znakova, odnosno određivanju udaljenosti nizova. Kod čistih globalnih algoritama, uzorka svaka transformacija se penalizira, obično s cijenom 1, kao što je prije navedeno. No, kod polu-globalnih metoda, dopuštamo da se transformacije na početku i/ili na kraju nizova ne penaliziraju. Ako cijenu transformacija na početku i na kraju niza zanemarujemo, tada globalni algoritam prelazi u infiksni, a ako zanemarujemo cijenu transformacija samo na kraju niza tada algoritam prelazi u prefiksni. Iz ovoga je vidljivo da su globalni algoritmi primjenjivi na nizove slične duljine, dok su polu-globalni bolji u slučajevima kada je tekst mnogo duži od uzorka. U ovome radu fokusirat ćemo se samo na globalne i polu-globalne algoritme. Osnovni algoritam za globalno poravnanje je Needleman-Wunschov algoritam (Needleman i Wunsch, 1970), koji dinamičkim programiranjem pronalazi poravnanje. Problem ovog algoritma je njegova vremenska složenost, koja iznosi $O(nm)$, gdje je m duljina uzorka, a n duljina teksta. Zbog toga je došlo do razvoja mnogih drugih algoritama koji pokušavaju smanjiti vremensku složenost, kao što su Ukkonen-ov algoritam (Ukkonen, 1985), koji složenost smanjuje pametnim ograničavanjem prostora pretraživanja te Hirschebergov algoritam (Hirschberg, 1975), koji prostornu složenost smanjuje na linearnu u zamjenu za nešto duže vrijeme izvođenja.

2.2. Poravnanje

Poravnanje dva niza, uzorka R i teksta B, je postupak u kojem se u tekstu pokušavaju pronaći područja koja su slična uzorku. Sličnost uzorka i područja unutar teksta definira se pomoću njihove udaljenosti, odnosno broja transformacija nad pojedinim znakovima koje se moraju primijeniti na uzorak kako bi se on u potpunosti podudarao s odabranim područjem u tekstu. Pretpostavimo da je uzorak $R = r_0r_1...r_{i-1}r_ir_{i+1}...r_{m-1}r_m$. Dopuštene transformacije na uzorku su:

1. brisanje znaka na poziciji *i* iz niza. Uzorak tada postaje $R = r_0r_1...r_{i-1}r_{i+1}...r_{m-1}r_m$.
2. umetanje znaka *z* u niz na poziciju *i*. Novi uzorak je $R = r_0r_1...r_{i-1}r_izr_{i+1}...r_{m-1}r_m$.
3. zamjena znaka na poziciji *i* znakom *z*, čime uzorak postaje $R = r_0r_1...r_{i-1}zr_{i+1}...r_{m-1}r_m$.

Prilikom određivanja udaljenosti, svaka od ovih transformacija može se penalizirati te se tada udaljenost uzorka od teksta definira kao zbroj svih vrijednosti penala transformacija (cijena transformacija) potrebnih da se uzorak transformira u tekst. Zbog jednostavnosti, ako nije drugačije navedeno, vrijednosti penala za svaku transformaciju bit će jednake te će iznositi 1, odnosno $C_{CH} = C_{DE} = C_{IN} = 1$, gdje C_{CH} predstavlja cijenu zamjene znaka, C_{DE} cijenu brisanja znaka, a C_{IN} cijenu umetanja znaka.

2.3. CIGAR format

Za prikaz rezultata poravnanja koristi se format CIGAR, koji je jedan od standardnih formata korištenih u bioinformatici. On se sastoji od oznake početne pozicije u tekstu od koje uspoređujemo uzorak i tekst, te od niza znakova koji predstavljaju broj znakova te transformacija koje se odnose na te znakove. Postoje dvije vrste CIGAR formata, standardni i prošireni. U standardnom formatu zamjena znaka i poklapanje znakova unutar uzorka i teksta se označavaju znakom "M", umetanje znaka u tekst označava se znakom "I", a brisanje znaka iz teksta označava se znakom "D". Za razliku od toga, prošireni CIGAR format razlikuje zamjenu znakova, koja se označava znakom "X", i poklapanje znakova, koje se označava znakom "M". Na primjer, za uzorak $R = ACTAGAATGGCT$ i tekst $B = CCATACTGAACTGACTAAC$, rezultat infiks poravnanja prikazan u standardnom CIGAR formatu bio bi:

$POS : 5$
$CIGAR : 3M1I3M1D5M$,
dok bi u proširenom CIGAR formatu to bilo:
$POS : 5$
$CIGAR : 3{=}1I3{=}1D2{=}1X2{=}$.

2.4. Edlib

Edlib (Šošić i Šikić, 2017) je c/c++ implementacija Myersovog bit vektor algoritma (Myers, 1999) za pronalaženje globalnog poravnanja između dva niza. Uz Myersov algoritam, edlib još koristi i Hirschebergov (Hirschberg, 1975) i Ukkonenov (Ukkonen, 1985) algoritam. Iako mu je složenost i dalje kvadratna, edlib je jako brz za dugačke slične nizove, jer se za njih dobiva najveće ubrzanje. No, edlib, zbog algoritama koje koristi, ima problem s kratkim nizovima, do otprilike 500 proteinskih baza, te je zato bilo potrebno ugraditi novi algoritam koji će moći brže rješavati problem za kratke nizove. Edlib podržava traženje globalnog, prefiksnog i infiksnog poravnanja dva niza, te omogućava prikaz rezultata poravnanja u formatu CIGAR.

3. Algoritam Landau-Vishkin-Nussinov

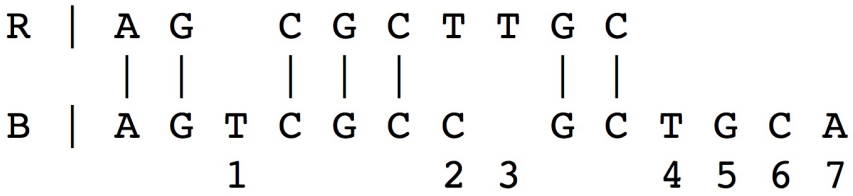
3.1. Definicija problema

Generalno, problem koji je potrebno riješiti je efikasno odrediti poravnanja između dva niza $R = r_0r_1 \ldots r_{m-1}r_m$ i $B = b_0b_1 \ldots b_{n-1}b_n$, gdje u našem slučaju, abecede sadrže samo znakove A, C, T, G. Ovaj problem možemo podijeliti na dva potproblema, a to su pronalaženje najmanje udaljenosti između nizova R i B, te pronalaženja točnih transformacija nad nizovima za koji se taj minimum postiže. Taj problem još se naziva problem traženja optimalnog puta za neko poravnanje.

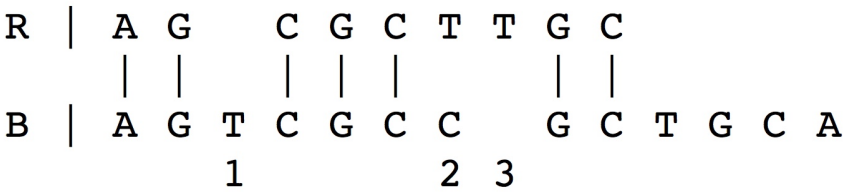
Razlikujemo tri vrste poravnanja, globalno, prefiksno i infiksno. Globalno poravnanje je poravnanje kod kojega se svaka razlika između dva niza ubraja u konačnu udaljenost. Za razliku od toga, kod prefiksnog i infiksnog algoritma, prilikom izračunavanja udaljenosti zanemarujemo razlike uzorka R i teksta B na početku ili na kraju teksta. Tako, kod prefiksnog algoritma u rezultat ulaze samo cijene transformacija koje dobivamo dok nismo došli do kraja uzorka R. Infiksno poravnanje slično je prefiksnom samo što dopuštamo zanemrivanje cijena i s početka teksta B, drugim riječima, dopuštamo da poravnanje uzorka ne počinje od prve pozicije teksta B. Demonstrirajmo to na primjeru uzorka $R = AGCGCTTGCTGC$ i teksta $B = AGTCGCCGCTGCTGCA$. Primjeri za ove tri vrste poravananja dani su na slijedećim slikama (Slika 3.1, Slika 3.2 i Slika 3.3).

3.1.1. Pronalaženje minimalne udaljenosti

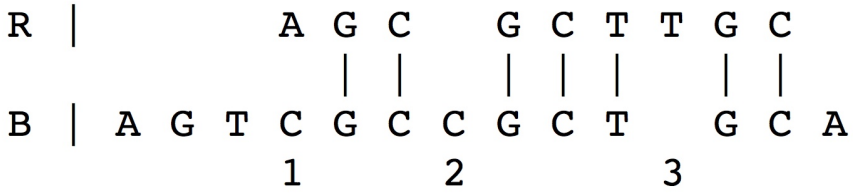
Pretpostavimo, bez smanjenja općenitosti, da se je $m \leq n$. Tada niz R možemo prozvati uzorkom, a niz B tekstom. Nizovi R i B mogu se razlikovati na $0 \leq i \leq m$ mjesta, a te promjene (mutacije) mogu biti brisanje, umetanje i izmjena znaka. Svaku od tih razlika možemo penalizirati s cijenom 1, te se tada naš problem pretvara u traženje minimal-



Slika 3.1: Primjer globalnog poravnanja između uzokra $R = AGCGCTTGCTGC$ i podniza teksta $B = AGTCGCCGCTGCA$



Slika 3.2: Primjer prefiksnog poravnanja između uzokra $R = AGCGCTTGCTGC$ i podniza teksta $B = AGTCGCCGCTGCA$



Slika 3.3: Primjer infiksnog poravnanja između uzokra $R = AGCGCTTGCTGC$ i podniza teksta $B = AGTCGCCGCTGCA$

nog broja transformacija koje uzorak R pretvaraju u tekst B. Ovako definiran problem odgovara globalnom poravnanju nizova. Prefiksno poravnanje možemo lagano dobiti iz globalnog tako da zbrajanje cijena prekinemo kada dođemo do kraja uzorka, a infiksno poravnanje odgovara prefiksnom, s mogućnošću da preklapanje uzorka i teksta počinjemo od proizvoljne pozicije unutar teksta. Matrica D je matrica $m \times n$ koja sadrži vrijednosti udaljenosti uzorka i teksta, dok se uz y os ($i = 1 \ldots m$) matrice proteže uzorak R, a uz x os ($j = 1 \ldots n$) tekst B. Drugim riječima, $D_{i,j}$ predstavlja udaljenost podnizova $r_1 \ldots r_i$ i $b_1 \ldots r_j$. Također, vidljivo je da povećanje vrijednosti $i \rightarrow i + 1$ odgovara pomicanju po uzorku, povećanje vrijednosti $j \rightarrow j + 1$ pomicanju po tekstu, dok je istovremeno pomicanje po uzorku i tekstu predstavljeno kao istovremeno povećanje vrijednosti i i j ($i \rightarrow i + 1, j \rightarrow j + 1$). U biološkom smislu, pomicanje samo

po uzorku ($i \rightarrow i+1$) odgovara brisanju znaka iz uzorka, pomicanje samo po tekstu ($j \rightarrow j+1$)odgovara umetanju znaka u uzorak, a pomicanje i po uzorku i po tekstu ($i \rightarrow i+1, j \rightarrow j+1$) odgovara ili poklapanju u slučaju da je $r_i = b_j$ ili izmjeni ako $r_i \neq b_j$.

Vidljivo je da je kretanje po nekoj dijagonali $d = j - i$ poželjnije nego kretanje po retku ili stupcu, jer svako takvo kretanje sigurno povećava broj transformacija, dok se kretanjem po dijagonali taj broj, u slučaju poklapanja, neće povećavati. Matrica koja nastaje ovakvim kretanjem kroz uzorak $R = ACTAGAATGGCT$ i tekst $B = CCATACTGAACTGACTAAC$ prikazana je u tablici 3.1. Ova matrica može se jednostavno izračunati pomoću algoritma (1) koji je dan u nastavku.

Algoritam 1 Osnovni algoritam za izračunavanje udaljenosti dva niza

$D_{0,0} \leftarrow 0$
$C_{CH} \leftarrow 1$
$C_{DE} \leftarrow 1$
$C_{IN} \leftarrow 1$
for all $i, 1 \leq i \leq m$ **do**
    $D_{i,0} \leftarrow i$
for all $j, 1 \leq j \leq n$ **do**
    $D_{j,0} \leftarrow j$

for $i \leftarrow 1$ to m **do**
    for $j \leftarrow 1$ to n **do**
        $D_{i,j} \leftarrow min(D_{i-1,j} + C_{DE}, D_{i.j-1} + C_{IN}, D_{i-1,j-1}$ if $r_i = b_j$ or $D_{i-1,j-1} + C_{CH})$

3.1.2. Pronalaženje optimalnog puta

Jednom kada pronađemo minimalnu udaljenost između uzorka i niza, još nam ostaje problem pronalaska točnih transformacija za koje se ta udaljenost postiže. Taj problem može se protumačiti i kao pronalazak puta po matrici D od kraja do početka uzorka. Najlakši način za pronalazak puta pri računanju optimalnog rješenja je pamćenje transformacija koje se provode, te to rješenje ne utječe na vremensku složenost algoritma, no zahtjeva da se sve transformacije na trenutnom putu čuvaju u memoriji, te se time povećava prostorna složenost problema. U matrici $D_{i,j}$ 3.1 podebljani elementi nalaze se na jednom od mogućih optimalnih puteva.

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			A	G	T	C	G	C	C	G	C	T	G	C	T	G	C
0		**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	A	1	**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	G	2	1	**0**	1	2	3	4	5	6	7	8	9	10	11	12	13
3	C	3	2	1	**1**	**1**	2	3	4	5	6	7	8	9	10	11	12
4	G	4	3	2	2	2	**1**	2	3	4	5	6	7	8	9	10	11
5	C	5	4	3	3	2	2	**1**	2	3	4	5	6	7	8	9	10
6	T	6	5	4	3	3	3	2	**2**	3	4	4	5	6	7	8	9
7	T	7	6	5	4	4	4	3	3	**3**	4	4	5	6	6	7	8
8	G	8	7	6	5	5	4	4	4	**3**	4	5	4	5	6	6	7
9	C	9	8	7	6	5	5	4	4	4	**3**	4	5	4	5	6	6
10	T	10	9	8	7	6	6	5	5	5	4	**3**	4	5	4	5	6
11	G	11	10	9	8	7	6	6	6	5	5	4	**3**	4	5	4	5
12	C	12	11	10	9	8	7	6	6	6	5	5	4	**3**	4	5	4

Tablica 3.1: Matrica D_{ij} prikazuje udaljenosti izračunate osnovnim algoritmom (1), za nizove $R = ACTAGAATGGCT$ i $B = CCATACTGAACTGACTAAC$. Vremenska složenost ovakvog pristupa je *O*(*nm*). Podebljani elementi prikazuju jedan od mogućih puteva do optimalnog rješenja.

3.2. Prefiksni algoritam

3.2.1. Teorijska pozadina

Vremenska složenost osnovnog algoritma opisanog u poglavlju 3.1.1 je kvadratna, odnosno *O*(*nm*), gdje je *n* duljina teksta, a *m* duljina uzorka. Uvedimo sada konstantu k, koja predstavlja maksimalni broj transformacija koje je dozvoljeno napraviti. Time smo ograničili prostor rješenja na nizove gdje je $n - m \leq k$, odnosno na nizove $R = r_0 r_1 \ldots r_m$ i $B = b_0 b_1 \ldots b_{m+k}$. Ako pogledamo još jednom tablicu 3.1, možemo vidjeti da elementi daleko od glavne dijagonale sigurno neće dati optimalnu udaljenost. Uvođenjem konstante k, za posljedicu ima ograničavanje prostora pretraživanja na sveukupno $2k - 1$ dijagonalu, koje se nalaze oko glavne dijagonale jer za sve dijagonale koje su od glavne dijagonale udaljene za više od k sigurno daju rješenje za koje je udaljenost veća od k.

Također, primijetimo da se, ako se krećemo po dijagonali d u matrici D_{ij}, tada razlika $D_{i,j} - D_{i-1,j-1}$ poprima vrijednost 0 ili 1. Ova informacija omogućava nam da efikasno spremamo podatke o matrici D_{ij}, jer nam je za svaku dijagonalu $d, |d| \leq k$ dovoljno pohraniti samo informacije o mjestima $i, j = i + d$ na kojima se vrijednost $D_{i,j}$ povećava. U svrhu spremanja tog podatka, uvest ćemo oznaku $L_{d,e}$, gdje d predstavlja dijagonalu, a $e \leq k$ predstavlja broj transformacija. Tada $L_{d,e}$ definiramo kao najveći red i takav da je $D_{i,i+d=j} = e$, iz čega možemo zaključiti da je broj razlika između podniza uzorka $r_0 r_1 \ldots r_{L_{d,e}}$ i podniza teksta $b_0 b_1 \ldots b_{L_{d,e}+d=j}$ jednak e. Također, primijetimo da iz definicije $L_{d,e}$ slijedi da $r_{L_{d,e}+1} \neq b_{L_{d,e}+d+1}$, jer bi inače $L_{d,e} + 1$ bio najveći red s e razlika.

Ako za neki $e \leq k$ uspijemo dobiti da je $L_{d,e} = m$, to znači da smo došli do kraja uzorka s manje od k pogrešaka, te iz toga možemo zaključiti da tekst B sadrži uzorak R transformiran e puta. Uvođenjem ograničenja na maksimalnu vrijednost udaljenosti, vremenska složenost algoritma prelazi iz *O*(*nm*) u *O*(*km*). Taj algoritam dan je u nastavku (2). Još nam je preostalo pokazati definirati kako možemo odrediti vrijednost od $L_{d,e}$. $L_{d,e}$ izračunavamo iz vrijednosti njegovih prethodnika, odnosno vrijednosti od $L_{d,e-1}$, $L_{d-1,e-1}$ i $L_{d+1,e-1}$ po formuli:

$$row = max \begin{cases} L_{d,e-1} + 1 \\ L_{d-1,e-1} \\ L_{d+1,e-1} + 1 \end{cases}$$

$$L_{d,e} = row + i, \text{ za } i \text{ td. } \forall a, a \leq i, r_{row+1+a} = b_{row+d+1+a}$$

Varijablu row za dijagonalu d inicijaliziramo kao maksimum od vrijednosti udaljenosti na tri dijagonale s koje smo mogli doći na dijagonalu d, a to su dijagonala $d-1$ (ako se mičemo po istom redu), d (ako se mičemo po dijagonali) i $d+1$ (ako se mičemo po istom stupcu). U slučajevima kada se mičemo po istom stupcu ili po dijagonali, dodajemo 1 na prošlu vrijednost, jer se takvim micanjem trenutni redak povećava za 1, odnosno micanjem po retku nismo napredovali u novi redak, te zato ne dodajemo 1. Ovo objašnjenje algoritma preuzeto je iz orginalnog rada Landau et al. (1986).

Algoritam 2 Algoritam za izračunavanje prefiksne udaljenosti dva niza s najviše k razlika

for $d \leftarrow -(k+1)$ to $(k+1)$ **do**
    $L_{d,|d|-2} \leftarrow -\infty$
    if $d < 0$ **then**
        $L_{d,|d|-1} \leftarrow |d| - 1$ [1]
    else
        $L_{d,|d|-1} \leftarrow -1$

for $e \leftarrow 0$ to k **do**
    for $d \leftarrow -e$ to e **do**

        [2] $row \leftarrow max(L_{d,e-1}+1, L_{d-1,e-1}, L_{d+1,e-1}+1)$
        while $(r_{row+1} = b_{row+d+1})$ **do**
            [3] $row \leftarrow row + 1$
        $L_{d,e} \leftarrow row$

        if $(L_{d,e} = m)$ **then**
            [4] **return** e

[5] **return** -1

3.2.2. Objašnjenje algoritma

1. Inicijalizacija vrijednosti na rubu tablice.

2. Određivanje vrijednosti varijable row na temelju maksimalne vrijednosti svih $L_{d,e}$ za koje smo mogli doći do trenutne pozicije, kao što je opisano ranije u

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			A	G	T	C	G	C	C	G	C	T	G	C	T	G	C
0		0	1	2	3												
1	A	1	0	1	2	3											
2	G	2	1	0	1	2	3										
3	C	3	2	1	1	1	2	3									
4	G		3	2	2	2	1	2	3								
5	C			3	3	2	2	1	2	3							
6	T				3	3	3	2	2	3							
7	T							3	3	3							
8	G									3							
9	C										3						
10	T											3					
11	G												3				
12	C													3			

Tablica 3.2: Matrica $D_{i,j}$ koja se dobije primjenom prefiksnog algoritma s maksimalnom udaljenosti $k = 3$. Vidljivo je da je znatno manji dio matrice istražen s obzirom na matricu dobivenu primjenom osnovnog algoritma (3.1). Važno je napomenuti da se prilikom izvođenja algoritma ovakva matrica ne sprema nigdje, već je dovoljno spremiti vrijednosti $L_{d,e}$.

poglavlju.

3. Iteracija po istoj dijagonali dokle su god vrijednosti uzorka R na mjestu row jednake vrijednostima teksta na B na mjestu $row + d$. Prilikom svake iteracije pomičemo se za jedno mjesto dalje po tekstu i uzorku

4. Algoritam prestaje ako dođemo do kraja uzorka, odnosno $row = m$, s manje od k transformacija.

5. Ako nismo uspjeli doći do kraja uzorka s manje od k transformacija tada je prefiksna udaljenost teksta i uzorka veća od k, te javljamo da nije moguće naći rješenje.

		A	G	C	G	C	T	T	G	C	T	G	C
		0	1	2	3	4	5	6	7	8	9	10	11
A	0	12	0	0	0	0	0	0	0	0	0	0	0
G	1	0	11	0	2	0	0	0	2	0	0	2	0
C	2	0	0	10	0	1	0	0	0	1	0	0	1
G	3	0	2	0	9	0	0	0	3	0	0	2	0
C	4	0	0	1	0	8	0	0	0	2	0	0	1
T	5	0	0	0	0	0	7	1	0	0	1	0	0
T	6	0	0	0	0	0	1	6	0	0	3	0	0
G	7	0	2	0	3	0	0	0	5	0	0	2	0
C	8	0	0	1	0	2	0	0	0	4	0	0	1
T	9	0	0	0	0	0	1	3	0	0	3	0	0
G	10	0	2	0	2	0	0	0	2	0	0	2	0
C	11	0	0	1	0	1	0	0	0	1	0	0	1

Tablica 3.3: MAXLENGTH matrica za uzorak $R = AGCGCTTGCTGC$. Primjetimo da je matrica simetrična, to jest da $MAXLENGTH(i, j) = MAXLENGTH(j, i)$.

3.3. Infiksni algoritam

3.3.1. Teorijska pozadina

Primijetimo da se u algoritmu opisanom u poglavlju 3.2 uvijek mičemo za maksimalno jedan redak u svakoj iteraciji. Tada, ako bi željeli ispitati za koju je poziciju j unutar teksta $B = b_0b_1...b_{j-1}b_jb_{j+1}...b_{n-1}b_n$, udaljenost uzorka $R = r_0r_1...r_{i-1}r_ir_{i+1}...r_{m-1}r_m$ od teksta B najmanja, morali bismo prefiksni algoritam ponoviti $n - m + k + 1$ puta, gdje k predstavlja maksimalnu dozvoljenu udaljenost. Vidljivo je da ovo nije optimalno rješenje, jer svaka nova iteracija ne uzima u obzir informacije dobivene iz prethodne iteracije. Ovaj problem riješit ćemo primjenom novog algoritma, koji na ulaz svake iteracije dobiva i rezultate prošle iteracije, od kojih će nam najvažnija biti najdalja pozicija do koje smo došli u tekstu u prethodnoj iteraciji.

Prvi korak algoritma bit će napraviti analizu uzorka. Na izlazu iz analize dobit ćemo kvadratnu matricu dimenzija m $MAXLENGTH$, gdje je m duljina uzorka R. Vrijednost f u matrici $MAXLENGTH$ na poziciji (i, j), odnosno $MAXLENGTH(i, j) = f$, označava da $r_{i+1}...r_{i+f} = r_{j+1}...r_{j+f}$ i da $r_{i+f+1} \neq r_{j+f+1}$. Vremenska složenost ove analize je $O(m^2)$. U tablici 3.3 prikazana je $MAXLENGTH$ matrica dobivena analizom uzorka $R = ACTAGAATGGCT$.

Nakon što provedemo analizu uzorka počinjemo s glavnim djelom algoritma koji se sastoji od $n - m + k + 1$ iteracija. Pretpostavimo da se nalazimo u i-toj iteraciji. U njoj tražimo poklapanje s najviše k transformacija između uzorka i podniza teksta koji počinje na $i+1$ mjestu (važno je napomenuti da se prvi element niza, kao što je vidljivo u tablici 3.2, nalazi na poziciji 1). Također, pretpostavimo da je b_j najdalja pozicija u tekstu do koje smo došli u nekoj od prethodnih iteracija algoritma (nazovimo ju l-ta iteracija, $0 \leq l < i$). Iz ovoga možemo jednostavno zaključiti da je udaljenost između podniza teksta $b_{l+1} \dots b_j$ i uzorka $\leq k$.

3.3.2. Primjer izvođenja

Uzmimo na primjer niz $R = ACTACTTTCCGAG$ i tekst $B = b_0 \dots b_{17} \dots b_{30} \dots b_m = AGCTACTTGTCCAG$, te iteraciju $l = 16, j = 30$. Vizualna interpretacija poklapanja između ta dva niza dana je na slici 3.4.

```
     1     2  3  4  5  6  7     8  9  10 11 12 13
R |  A     C  T  A  C  T  T     T  C  C  G  A  G
     |     |  |  |  |  |  |     |  |  |     |  |
B |  A  G  C  T  A  C  T  T  G  T  C  C     A  G
     17 18 19 20 21 22 23 24 25 26 27 28    29 30
     bl+1          bi+1                       bj
```

Slika 3.4: Poklapanje između uzokra $R = ACTACTTTCCGAG$ i podniza teksta $B = b_{17} \dots b_{30} = \dots AGCTACTTGTCCAG \dots$

Udaljenost između ta dva niza je $k = 3$. Iz toga slijedi da je udaljenost nekog podniza uzorka R i podniza teksta $b_{i+1} \dots b_j$ sigurno $\leq k$. U prethodnom primjeru, za podniz uzorka $r_4 \dots r_{13}$, postoji barem $30 - 20 - 3$ ($j - i - k$) znakova koji se poklapaju s podnizom teksta $b_{21} \dots b_{30}$. Podniz uzorka možemo podijeliti na najviše $k + 1$ uzastopnih podnizova, takvih da se svaki od tih podnizova u potpunosti poklapa s podnizom teksta, odnosno podniz dobivamo ako niz podijelimo na mjestima gdje se uzorak i tekst razlikuju. Uvedimo sada trojku (p, c, f), gdje p označava poziciju u tekstu, c označava poziciju u uzorku te f označava duljinu poklapanja. Ako se podniz teksta $b_{p+1} \dots b_{p+f}$ poklapa s podnizom uzorka $r_{c+1} \dots r_{c+f}$, te ako je $r_{c+f+1} \neq b_{p+f+1}$, tada ćemo to poklapanje zapisati prethodno definiranom trojkom (p, c, f) (u primjeru, jedna od trojki je $(18, 1, 6)$). Isto tako, znakove podniza teksta $b_{i+1} \dots b_j$ za koje ne postoji poklapanje, možemo označiti trojkom $(h, 0, 0)$, gdje h označava poziciju unutar teksta B, a 0 označava da u uzorku ne postoji znak koji odgovara znaku u tekstu (u primjeru, $(17, 0, 0)$). Primjenom ovakve analize na podniz teksta $b_{i+1} \dots b_j$, podniz možemo po-

dijeliti na $O(k)$ slijednih poklapanja i promašaja. Za podniz u primjeru to prikazujemo podnizom $(20,3,4),(24,0,0),(25,7,3),(28,11,2)$. Ovaj niz trojki označit ćemo sa $S_{20,30}$ (općenito $S_{i,j}$, gdje i označava trenutnu iteraciju, a j označava najdalju poziciju koju smo dosegli u tekstu). Ovaj primjer direktno je preuzet iz orginalnog rada Landau et al. (1986).

3.3.3. Detaljan opis algoritma

Jedna iteracija algoritma bazira se na prefiksnom algoritmu opisanom u poglavlju 3.2 s time da prilikom pomicanja po uzorku koristimo matricu $MAXLENGTH$ (3.3) te niz trojki $S_{i,j}$ dobiven iz prethodne iteracije kako bi postigli ubrzanje. U jednoj iteraciji, koristit ćemo bazu prefiksnog algoritma (2), no uz zamjenu instrukcije [3] s novim nizom instrukcija. Umjesto da varijablu row povećavamo za jedan u svakom prolazu, koristi ćemo se znanjem prethodne iteracije i matricom $MAXLENGTH$ kako bi varijablu row povećali za veće iznose.

Vratimo se sada na iteraciju $i = 20$ opisanu ranije u poglavlju ($j = 30$, $S_{i,j} = (20,3,4),(24,0,0),(25,7,3),(28,11,2)$). Novi skup instrukcija (Algoritam 4) koji zamjenjuje instrukciju [3] prefiksnog algoritma omogućava nam da, dokle god se nalazimo u djelu teksta prije $j = 30$ pozicije, koristimo matricu $MAXLENGTH$ (3.3) i niz trojki $S_{i,j}$ da varijablu row povećavamo za više od 1 te tako ubrzamo algoritam. While petlja traži najduže moguće poklapanje između podnizova teksta $b_{i+row+d+1}$ i uzorka r_{row+1}. To postižemo uz pomoć niza $S_{i,j}$, odnosno tražimo broj c u nekoj od trojki (p,c,f) sadržanih u $S_{i,j}$ takav da $b_{i+row+d+1} \dots b_{i+row+d+f} = r_{c+1} \dots r_{c+f}$.

U slučaju kada u $S_{i,j}$ postoji poklapanje (postoji trojka (p,c,f) takva da je $f \geq 1$), tražimo element matrice $g = MAXLENGTH[c][row]$. Tada razlikujemo dva slučaja:

1. Kada je $f \neq g$: Tada je duljina poklapanja $min(f,g)$, odnosno da je $b_{i+row+d+1} \dots b_{i+row+d+min(f,g)} = r_{row+1} \dots r_{row+min(f,g)}$ i $b_{i+row+d+min(f,g)+1} \neq r_{row+min(f,g)+1}$. To možemo zaključiti iz definicija matrice *MAXLENGTH* i trojke (p,c,f). Vrijednost $f \geq 1$ nam govori da smo u nekoj od prošlih iteracija algoritma već pokrili ovaj dio uzorka i teksta, te da ovdje postoji poklapanje između uzorka i teksta duljine f. Vrijednost g nam omogućava da koristimo i vrijednosti trojke (p,c,f) koje se ne odnose izravno na trenutni podniz uzorka $r_{row+1} \dots r_{row+g}$, nego se odnose na podniz $r_{c+1} \dots r_{c+g}$ identičan trenutnom ($r_{row+1} \dots r_{row+g} = r_{c+1} \dots r_{c+g}$). Informacija o duljini g podniza za koje prethodna jednakost vrijedi zapisana je u

Algoritam 3 Algoritam za izračunavanje infiksne udaljenosti dva niza s najviše k razlika

1: $\# C_{CH} = C_{DE} = C_{IN} = 1$
2: $j \leftarrow 0$
3: $S_{i,j} \leftarrow \emptyset$
4: **for** $i \leftarrow 0$ to $n - m + k$ **do**
5: **for** $d \leftarrow -(k+1)$ to $(k+1)$ **do**
6: $L_{d,|d|-2} \leftarrow -\infty$
7: **if** $d < 0$ **then**
8: $L_{d,|d|-1} \leftarrow |d| - 1$
9: **else**
10: $L_{d,|d|-1} \leftarrow -1$

11: **for** $e \leftarrow 0$ to k **do**
12: **for** $d \leftarrow -e$ to e **do**
13: $row \leftarrow max(L_{d,e-1} + 1, L_{d-1,e-1}, L_{d+1,e-1} + 1)$
14: Izvrši novi skup instrukcija
15: **while** $(r_{row+1} = b_{row+d+1})$ **do**
16: $row \leftarrow row + 1$
17: [5] $L_{d,e} \leftarrow row$
18:
19: **if** $(L_{d,e} = m)$ **then**
20: **goto** 7
21: [7] Ako smo u trenutnoj iteraciji došli do nekih novih simbola, odnosno ako se j povećao, počevši od $L_{d,k}$ (po definiciji $j = L_{d,k} + d + i$) stvorimo novi niz trojki $S_{i,j}$ na temelju analize teksta.
22: **if** $(L_{d,e} = m)$ **then**
23: **return** e
return -1

Algoritam 4 Nove istrukcije koje omogućuju ubrzanje infiksnog algoritma

1: **while** $(i + row + d + 1 \leq j)$ **do**
2: $c \leftarrow 0$
3: $f \leftarrow 0$
4: **for** S in $S_{i,j}$ **do**
5: **if** $S.p + S.f > row + d + i + 1$ and $S.p = row + d + i + 1$ **then**
6: $c \leftarrow S.c$
7: $f \leftarrow S.f$
8: **break**
9:
10: **if** $f \geq 1$ **then**
11: **if** $f \neq MAXLENGTH[c][row]$ **then**
12: $row \leftarrow row + min(f, MAXLENGTH[c][row])$
13: **goto** 5
14: **else**
15: $row \leftarrow row + f$
16: **else**
17: **if** $r_{row+1} \neq b_{i+row+1+d}$ **then**
18: **goto** 5
19: **else**
20: $row \leftarrow row + 1$

$MAXLENGTH[c][row]$.

Varijablu row tada povećavamo za $min(f, g)$ i odlazimo na instrukciju [5]. U prethodnom primjeru, pretpostavimo da je $row = 8$ i $d = -2$. To znači da tražimo preklapanje između $r_9 \ldots$ i $b_{27} \ldots$. Trojka koja pokriva trenutni položaj je $(25, 7, 3)$, što znači da su znakovi na pozicijama 27 i 28 u tekstu jednaki znakovima na mjestu 9 i 10 u uzorku, odnosno $f = 2$ i $c = 8$ te $g = MAXLENGTH[7][7] = 5$. Iz toga slijedi da se možemo pomaknuti za 2 mjesta dalje, odnosno $row = row + 2$.

2. Kada je $f = g$: Tada sigurno vrijedi da je $b_{i+row+d+1} \ldots b_{i+row+d+f} = r_{row+1} \ldots r_{row+f}$, no ne znamo ništa o odnosu sljedećih znakova u uzorku i tekstu, odnosno vrijedi li $b_{i+row+d+f+1} = r_{row+f+1}$. Zato, sada povećavamo row za f te se vraćamo na while petlju i nastavljamo algoritam.

Ako poklapanje koje pokriva $b_{i+row+d+1}$ ne postoji u $S_{i,j}$, tada tražimo poklapanje između $b_{i+row+d+1}$ i r_{row+1}. U prethodnom primjeru to bi odgovaralo stanju kada je $row = 3$ i $d = 1$. Ovdje opet razlikujemo dva slučaja:

1. Kada je $b_{i+row+d+1} \neq r_{row+1}$: Tada ne postoji poklapanje i odmah nastavljamo na instrukciju [5].

2. Kada je $b_{i+row+d+1} \neq r_{row+1}$: Tada varijablu row povećavamo za 1 te se vraćamo na while petlju i nastavljamo algoritam.

U instrukciji [7] stvaramo novi niz trojki $S_{i,j}$ ako smo došli do nekog novog znaka u tekstu. Neka $b_{\bar{j}}$ predstavlja zadnji novi znak u tekstu do kojeg smo došli u iteraciji i. Također, za svaku iteraciju i čuvamo niz trojki za svaki $L_{d,e}$ izračunat u toj iteraciji. Za izgradnju niza trojki za $L_{d,e}$ koristit ćemo se nizom trojki njegovog prethodnika, odnosno niza trojki od ili $L_{d-1,e-1}$ ili $L_{d,e-1}$ ili $L_{d+1,e-1}$, ovisno o tome koji od njih smo odabrali na početku iteracije, na liniji 13 algoritma. Nazovimo taj niz trojki $S_{L_{d,e}}$ Uvedimo još i l_1 koji predstavlja vrijednost varijable row na početku iteracije (on je određen odabirom u liniji 13). Novi niz trojki sada gradimo tako da prvo, ako smo l_1 dobili odabirom ili $L_{d-1,e-1}$ ili $L_{d,e-1}$, dodamo trojku $(i + l_1 + d - 1, 0, 0)$ na kraj $S_{L_{d,e}}$. Nakon što izvršimo instrukciju [5], provjerimo je li $L_{d,e} > l_1$. Ako je, na kraj niza $S_{L_{d,e}}$ dodajemo trojku $(i + l_1 + d, l_1, L_{d,e} - l_1)$.

Na samom kraju iteracije i provjerimo za koju od $2k + 1$ nizova trojki pridruženih svakom $L_{d,e}$, dolazimo najdalje u tekstu. Ako je taj index veći od j, odnosno ako je $L_{d,e} + d + i > j$, tada niz trojki $S_{i,j}$ postavljamo na $S_{L_{d,e}}$.

3.3.4. Vremenska složenost infiksnog algoritma

U najgorem slučaju, vremenska složenost stare instrukcije [3] (instrukcija gdje se varijabla row može povećati samo za 1) prefiksnog algoritma bila je $O(kn)$. Za izračun složenosti novog skupa instrukcija, koristit ćemo se činjenicom da u svakom trenutku održavamo najviše $2k + 1$ dijagonalu te da niz $S_{i,j}$ sadrži najviše $2k + 1$ trojki. Za bilo koju operaciju na dijagonali možemo pretpostaviti da se radi ili o pronalasku razlike (takvih operacija ima najviše k) ili o provjeravanju trojke u nizu $S_{i,j}$. Vidimo da ovo rezultira s $O(k)$ operacija za svaku dijagonalu za svaku iteraciju i. Iz toga slijedi da je ukupna vremenska složenost ovog algoritma $O(k^2n)$.

4. Implementacija

Algoritam Landau-Vishkin-Nussinov implementirali smo u programskom jeziku C++, korištenjem standarda C++11. Odabrali smo jezik C++ iz više razloga, od kojih su dva glavna bila velika brzina izvođenja jezika te zato što je edlib napisan isto napisan u C++, pa smo time olakšali i ubrzali ugradnju.

Programski kod implementacije zajedno s uputstvima za korištenje javno je dostupan na web starnici: https://github.com/donikv/Zavrsni_Rad. Također, programski kod edliba te uputstva za korištenje dostupni su na web stranici: https://github.com/Martinsos/edlib. Važno je napomenuti da se dalje u poglavlju detaljnije opisuje verzija implementacije algoritma koja je postojala u trenutku pisanja rada te da postoji mogućnost da se trenutne implementacije dostupne na webu razlikuje od opisane.

4.1. Prefiksni algoritam

U nastavku poglavlja dane su implementacije 5 osnovnih dijelova prefiksnog algoritma koje smo naveli u poglavlju 3.2. Definirajmo razred $EqualityDefinition$ koji nam omogućava da proširimo relaciju jednakosti nad znakovima abecede od kojih su uzorak i tekst načinjeni. Kako bi mogli koristiti taj razred, prije početka algoritma, uzorak i tekst transformiramo u format koji nam omogućava brzo pronalaženje proširene relacije jednakosti pomoću razreda $EqualityDefinition$ te prilikom te transformacije također formiramo i abecedu s kojom su nizovi napravljeni. Definicija razreda $EqualityDefinition$ preuzeta je iz alata edlib. Inicijalizacija tog razeda bazira se na izgradnji matrice gdje nam element na poziciji a, b govori jesu li znakovi koji se transformiraju u a i b jednaki. Transformacija znakova je jednostavna pretvorba znaka u njegovu poziciju u nizu koji predstavlja abecedu. To nam omogućava da provjera jednakosti ima složenost $O(1)$.

4.1.1. Inicijalizacija vrijednosti

U sljedećem odsječku koda prikazana je inicijalizacija vrijednosti na početku algoritma. Vrijednosti koje se inicijaliziraju su vrijednosti koje se nalaze izvan matrice (3.2) te su njihove vrijednosti izabrane na takav način da se nikada ne može dogoditi da se u koraku [2] algoritma neka od tih vrijednosti odabere kao maksimalna. Vrijednost -5 dana je kao zamjena za $-\infty$, jer je to dovoljno velika negativna vrijednost da će sigurno uvijek biti manja od svih $L_{d,e}$. Također važno je primijetiti da mapa $cigarDict$ sadrži kopije vektora znakova, a razlog za to je jer se vektor cv konstantno nadograđuje na vektore svojih prethodnika kako napredujemo kroz algoritam, no njegovu vrijednost u svakom koraku (za svaki izračunati $L_{d,e}$) moramo pospremiti kako bi kasnije mogli odabrati optimalni put. Vrijednost varijable nk nam omogućava da s donje strane ograničimo udaljenost, odnosno definiramo najmanju moguću udaljenost između nizova. Razredi $Hasher$ i $EqualFn$ su pomoćni razredi koji omogućavaju pohranu strukture L u hash tablicu.

```
unordered_map<L,vector<char>, Hasher, EqualFn> cigarDict;
vector<char> cv;

for (int d = -(k); d<=k; d++){
  if(d>=-nk && d<=nk && d!=0) continue;
   D[L{d,abs(d)-2}] = -5;
   if(d<0) D[L{d, -d-1}] = -d-1;
   else D[L{d, d-1}] = -1;

   if(cigar){
      cigarDict[L{d,abs(d)-2}] = cv;
      cigarDict[L{d,abs(d)-1}] = cv;
   }
}
```

4.1.2. Odabir vrijednosti varijable row

Način odabira nove vrijednosti varijable row detaljno je opisan u poglavlju 3.2. Funkcija max prima vrijednost num, u koju pohranjuje vrijednost koja je od predanih $L_{d,e}$ vrijednosti bila odabrana. Ta informacija nam je potrebna u drugom dijelu odsječka

kako bi mogli odrediti na koju od vrijednost moramo dodati na kraj vektora cv. Također, isto kao i u prošlom odsječku, vrijednost -5 predstavlja zamjenu za vrijednost $-\infty$.

```
if(d==-e){
   row = max(D[L{d,e-1}]+1, D[L{d+1,e-1}]+1, -5, &num);
} else if (d==e) {
   row = max(D[L{d,e-1}]+1, -5, D[L{d-1,e-1}], &num);
} else {
   row = max(D[L{d,e-1}]+1, D[L{d+1,e-1}]+1, D[L{d-1,e-1}],
      &num);
}


if(cigar){
   switch (num){
      case 1:
         cv = cigarDict[L{d,e-1}];
         if(e!=0)
            cv.push_back('I');
         break;
      case 2:
         cv = cigarDict[L{d+1,e-1}];
         cv.push_back('X');
         break;
      case 3:
         cv = cigarDict[L{d-1,e-1}];
         cv.push_back('D');
         break;
   }
}
```

4.1.3. Napredovanje po dijagonali

Nakon odabira trenutne vrijednosti varijable row, dolazimo do glavnog dijela algoritma, odnosno djela gdje iteriramo po dijagonali d sve dok su vrijednosti uzorka R na mjestu row jednake vrijednostima teksta na B na mjestu $row + d + bStart$. Vari-

jabla $bStart$ omogućava nam da na početku algoritma zadamo poziciju unutar teksta B od koje ćemo početi tražiti udaljenost. Na kraju odsječka obavljamo pridruživanje trenutne vrijednosti varijable row trenutno aktivnom $L_{d,e}$. Varijabla $equality$ je primjerak razreda $EqualityDefinition$.

```
while(equality.areEqual(R[row], B[row+d+bStart]) && row<m) {
   if (cigar) cv.push_back('=');
   row++;
}
D[L{d,e}] = row;
if (cigar) cigarDict[L{d,e}] = cv;
```

4.1.4. Kraj algoritma

Prefiksni algoritam završava ako u nekom trenutku vrijednost varijable row dosegne vrijednost m koja predstavlja duljinu uzorka R ili ako isprobamo svaku od $2k + 1$ dijagonalu i ni u jednom slučaju ne dođemo do kraja uzroka.

```
for (int e = nk; e<=k; e++) {
   for(int d = -e; d<=e; d++) {

      //Tijelo algoritma, koje je opisano ranije u poglavlju

      if(row == m) {
         if (cigar) cigarVector = cv;
      }
   }
}
return -1;
```

4.2. Infiksni algoritam

Implementacija infiksnog algoritma bazira se na ponavljanju prefiksnog algoritma $n - m + k + 1$ puta, uz optimizaciju napredovanja po dijagonali prikazanog u potpoglavlju 4.1.3. Infiksni algoritam detaljno je opisan u poglavlju 3.3.

4.2.1. Optimizirano napredovanje po dijagonali

Ubrzanje napredovanja po dijagonali postižemo korištenjem znanja iz prošlih iteracija kako bi mogli varijablu row povećavati za iznose veće od 1. Ovdje se koristimo nizom trojki (p, c, f) $S_{i,j}$. U ispod prikazanom odsječku je while petlja koja nam omogućava da dokle god se nalazimo u već istraženom dijelu teksta ($row + d + i \leq j$) koristimo $S_{i,j}$ da napredujemo brže. Ovaj dio algoritma opisan je pseudokodom te popratnim objašnjenjem u poglavlju 3.3 (Algoritam 4).

```
l1 = row;
while(row+d+i<=j){
   unsigned int c=0;
   unsigned int f=0;
   for(const auto& t: Sij){
      if(t.p+t.f>row+d+i && t.p==row+d+i){
         f=t.f;
         c=t.c;
         break;
      }
   }

   if(f>=1) {
      if(f != MAXLENGTH[c*m+row]) {
         row += std::min(f,MAXLENGTH[c*m+row]);
         if (cigar) {
            for(int a=0;a<std::min(f,MAXLENGTH[c*m+row]);a++)
               cv.push_back('=');
         }
         goto inst5;
      } else {
         if (cigar) {
            for(int a=0;a<f;a++) cv.push_back('=');
         }
         row += f;
      }
   } else {
      if(!(equality.areEqual(R[row], B[row+d+i]))){
         goto inst5;
```

```
        } else {
            if (cigar) cv.push_back('=');
            row++;
        }
    }
}
```

4.2.2. Stvaranje novog niza trojki za trenutni $L_{d,e}$

Svaki puta kada izračunamo novi $L_{d,e}$ potrebno je izračunati i pripadni niz trojki $S_{i,j}$ koji opisuju kako smo došli do pozicije unutar uzorka koja je u njemu zapisana. Varijabla $pickedL$ sadrži informaciju o tome koji $L_{\bar{d},e-1}$ (gdje $\bar{d}$ može biti ista dijagonala ili dijagonala ispod ili iznad trenutne dijagonale d) smo izabrali kao prethodnik prilikom izračunavanja trenutnog $L_{d,e}$. Novi niz trojki S za trenutni L izgradit ćemo na temelju niza S za prethodni L. Ako se prethodni L nalazio na dijagonali ispod ili na istoj dijagonali tada ćemo u niz S na kraj dodati trojku $(i + l_1 + d - 1, 0, 0)$. Zatim, ako smo varijablu row povećali za barem 1 (varijabla l_1 čuva vrijednost varijable row s početka iteracije), dodajemo na kraj niza S trojku $(i + l_1 + d, l_1, row - l_1)$. Ovaj postupak detaljno je opisan i objašnjen kroz primjer u poglavlju 3.3.2.

```
inst5:
D[L{d,e}] = row;
if (cigar) cigarDict[L{d,e}] = cv;

L pickedL; //get L that was picked as the L that gives the
   maximum row
if(num==1) { pickedL.d = d; pickedL.e = e-1; }
else if(num == 2){ pickedL.d = d+1; pickedL.e = e-1; }
else { pickedL.d = d-1; pickedL.e = e-1; }

std::vector<Triple> sequenceForCurrentL = lSeqMap[pickedL];

if((pickedL.d == d-1 || pickedL.d == d) && l1+d>0)
   sequenceForCurrentL.push_back(Triple{i+l1+d-1,0,0});
if(row>l1) sequenceForCurrentL.push_back(Triple{i+l1+d, l1,
   row-l1});
```

```
lSeqMap[L{d,e}] = sequenceForCurrentL;

if(row == m){
   goto inst7;
}
```

4.2.3. Obabir novog niza $S_{i,j}$

U ovom dijelu algoritma provjeravamo jesmo li se pomaknuli dalje u tekstu nego prije. U slučaju kada je $row + d + i \leq j$ tada to nije slučaj te preskačemo ovaj cijeli dio algoritma. U suprotnom, tražimo prvo pojavljivanje niza $S_{i,j}$ za koje je zadnja pozicija zapisana u trojki (p, c, f) veća ili jednaka trenutnom j. Taj niz odabiremo kao novi $S_{i,j}$ te prelazimo u sljedeću iteraciju u slučaju da nismo došli do kraja uzorka. Inače vratimo trenutnu vrijednost e, odnosno udaljenost nizova.

```
inst7:
if(i+row+d<=j) continue;
j = i+row+d;

L current_L;

for(int l = -k; l<=k; l++){
   current_L = L{l,k};
   std::vector<Triple> sequenceForCurrentL =
      lSeqMap[current_L];
   if(sequenceForCurrentL.size()<=0) continue;

   if((sequenceForCurrentL.back().p+sequenceForCurrentL.back().f)>=j)
      {
      Sij = sequenceForCurrentL;
      break;
   }
}
if(row == m) {
   if (cigar) cigarVector = cigarDict[current_L];
   return e;
}
```

4.3. Ugradnja u edlib

U nastavku dan je odsječak koda koji prikazuje najjednostavniju predloženu metodu ugradnje prefiksne implementacije algoritma Landau-Vishkin-Nussinov u edlibovu glavnu metodu za traženje poravnanja. Edlib koristi drugačije nazivlje, tako da varijabla $query$ predstavlja uzorak, a $target$ predstavlja tekst. Alat i biblioteka edlib detaljnije je opisan u radu Šošić i Šikić (2017). Prijedlog za ugradnju prefiksnog algoritma u edlib dostupan je kao *Pull Request* na web stranici: https://github.com/Martinsos/edlib/pull/119.

```
bool useLV = queryLength < 500 && config.mode ==
   EDLIB_MODE_SHW;
if (useLV) {
   vector<unsigned char> R(query, query + queryLength);
      vector<unsigned char> B(target, target + targetLength);
      vector<unsigned char> cigarVector;

   int k = config.k < 0 ? queryLength : config.k;
   result.endLocations = (int *) malloc(sizeof(int) * 1);
   result.editDistance = landauVishkinAlignPrefix(R, B, k,
      equalityDefinition, config.task == EDLIB_TASK_PATH,
      cigarVector, result.endLocations);

   if(result.editDistance>=0) {
      if(config.task == EDLIB_TASK_PATH) {
         result.alignmentLength = cigarVector.size();
         unsigned char* alignment = new unsigned
            char[result.alignmentLength];
         for(int i=0;i<result.alignmentLength;i++)
            alignment[i] = cigarVector[i];
         result.alignment = alignment;
      }

      result.startLocations = (int *) malloc(sizeof(int) * 1);
      result.startLocations[0] = 0;
```

```
        result.numLocations = 1;
    }
    return result;
}
```

5. Rezultati

Algoritam Landau-Vishkin-Nussinov dijeli se na dva algoritma, jedan za određivanje prefiksne udaljenosti nizova, a drugi za određivanje infiksne udaljenosti nizova, dokle god je ta udaljenost manja od k. Uvođenjem tog ograničenja na rezultat, vremenska složenost prefiksnog algoritma postaje $O(km)$, a infiksnog algoritma $O(k^2n)$, gdje je m duljina uzorka, a n duljina teksta. Iz toga možemo zaključiti da će ovakav algoritam imati najveće ubrzanje za kratke nizove, s relativno malim brojem razlika između njih.

5.1. Usporedba s edlibom

Brzina implementacije opisane u ovom radu testirana je u ovisnosti o alatu edlib. Testiranje je provedeno s jednim tekstom i nizom različitih uzoraka. Niz koji je korišten kao tekst bio je zapis genoma bakterije *Escherichie coli*. Duljina teksta n iznosila je oko 5 milijuna proteinskih baza, dok su duljine korištenih uzoraka varirale su od 50 do 500 proteinskih baza. Nizovi su bili zapisani u FASTA formatu (**?**) te je za njihovo učitavanje korišten jednostavni parser koji u linearnom vremenu učitava niz u memoriju. Vrijeme potrebno za učitavanje nizova također je ulazilo u vrijeme koje je testirano.

5.1.1. Prefiksni algoritam

U tablicama 5.1, 5.2, 5.3 i 5.4 prikazani su rezultati testiranja naše implementacije i edliba. Prvi stupac tablice određuje uzorak na kojem je testiranje provedeno, a drugi stupac prikazuje duljinu uzorka. U trećem stupcu nalazi se vrsta algoritma koji je korišten, gdje Edlib označava obično traženje udaljenosti, Edlib (path) označava traženje poravnanja i optimalnog puta, dok LandauVishkin označava korištenje naše implementacije. Sljedeći stupac prikazuje prosječno vrijeme trajanja algoritma u sekundama za 100 ponavljanja, dok se u zadnjem stupcu nalazi prefiksna udaljenost uzorka i teksta. Vidljivo je da je naša implementacija na nekim, a pogotovo na vrlo kratkim primjerima brža i za više od 100 puta od edliba. Takvi rezultati su očekivani s obzirom na

Uzorak	m	Algoritam	Vrijeme	Udaljenost
50bp/mutated_60_perc.fasta	50	Edlib	0.4526	22
50bp/mutated_60_perc.fasta	50	Edlib (path)	0.4529	22
50bp/mutated_60_perc.fasta	50	LandauVishkin	0.0158	22
50bp/mutated_70_perc.fasta	50	Edlib	0.4511	15
50bp/mutated_70_perc.fasta	50	Edlib (path)	0.4585	15
50bp/mutated_70_perc.fasta	50	LandauVishkin	0.0097	15
50bp/mutated_80_perc.fasta	50	Edlib	0.4552	11
50bp/mutated_80_perc.fasta	50	Edlib (path)	0.4506	11
50bp/mutated_80_perc.fasta	50	LandauVishkin	0.0060	11
50bp/mutated_90_perc.fasta	50	Edlib	0.4501	4
50bp/mutated_90_perc.fasta	50	Edlib (path)	0.4520	4
50bp/mutated_90_perc.fasta	50	LandauVishkin	0.0028	4
50bp/mutated_94_perc.fasta	50	Edlib	0.4527	3
50bp/mutated_94_perc.fasta	50	Edlib (path)	0.4529	3
50bp/mutated_94_perc.fasta	50	LandauVishkin	0.0026	3
50bp/mutated_97_perc.fasta	50	Edlib	0.4508	3
50bp/mutated_97_perc.fasta	50	Edlib (path)	0.4523	3
50bp/mutated_97_perc.fasta	50	LandauVishkin	0.0027	3
50bp/mutated_99_perc.fasta	50	Edlib	0.4505	1
50bp/mutated_99_perc.fasta	50	Edlib (path)	0.4540	1
50bp/mutated_99_perc.fasta	50	LandauVishkin	0.0024	1

Tablica 5.1: Tablica koja prikazuje rezultate testiranja implementacije prefiksnog algoritma Landau-Vishkin-Nussinov i alata edlib na uzorcima duljine 50 proteinskih baza.

to da je algoritam Landau-Vishkin-Nussinov pogodan za kratke, slične nizove, dok edlib najveće ubrzanje dobiva kod dugačkih, sličnih nizova. Usporedba edliba s drugim algoritmima dana je u radu Šošić i Šikić (2017).

5.1.2. Infiksni algoritam

U tablici 5.5 dani su rezultati usporedbe implementacije infiksnog algoritma i alata edlib. Prvi stupac tablice određuje uzorak na kojem je testiranje provedeno, a drugi stupac prikazuje duljinu uzorka. U trećem stupcu nalazi se vrsta algoritma koji je korišten, gdje Edlib označava obično traženje udaljenosti, Edlib (path) označava traženje porav-

Uzorak	m	Algoritam	Vrijeme	Udaljenost
100bp/mutated_60_perc.fasta	100	Edlib	0.4523	47
100bp/mutated_60_perc.fasta	100	Edlib (path)	0.4637	47
100bp/mutated_60_perc.fasta	100	LandauVishkin	0.0649	47
100bp/mutated_70_perc.fasta	100	Edlib	0.4543	35
100bp/mutated_70_perc.fasta	100	Edlib (path)	0.4565	35
100bp/mutated_70_perc.fasta	100	LandauVishkin	0.0376	35
100bp/mutated_80_perc.fasta	100	Edlib	0.4520	22
100bp/mutated_80_perc.fasta	100	Edlib (path)	0.4626	22
100bp/mutated_80_perc.fasta	100	LandauVishkin	0.0197	22
100bp/mutated_90_perc.fasta	100	Edlib	0.4520	9
100bp/mutated_90_perc.fasta	100	Edlib (path)	0.4540	9
100bp/mutated_90_perc.fasta	100	LandauVishkin	0.0075	9
100bp/mutated_94_perc.fasta	100	Edlib	0.4531	7
100bp/mutated_94_perc.fasta	100	Edlib (path)	0.4546	7
100bp/mutated_94_perc.fasta	100	LandauVishkin	0.0071	7
100bp/mutated_97_perc.fasta	100	Edlib	0.4541	4
100bp/mutated_97_perc.fasta	100	Edlib (path)	0.4538	4
100bp/mutated_97_perc.fasta	100	LandauVishkin	0.0058	4
100bp/mutated_99_perc.fasta	100	Edlib	0.4534	2
100bp/mutated_99_perc.fasta	100	Edlib (path)	0.4557	2
100bp/mutated_99_perc.fasta	100	LandauVishkin	0.0054	2

Tablica 5.2: Tablica koja prikazuje rezultate testiranja implementacije prefiksnog algoritma Landau-Vishkin-Nussinov i alata edlib na uzorcima duljine 100 proteinskih baza.

Uzorak	m	Algoritam	Vrijeme	Udaljenost
250bp/mutated_60_perc.fasta	250	Edlib	0.4519	111
250bp/mutated_60_perc.fasta	250	Edlib (path)	0.4564	111
250bp/mutated_60_perc.fasta	250	LandauVishkin	0.3381	111
250bp/mutated_70_perc.fasta	250	Edlib	0.4593	92
250bp/mutated_70_perc.fasta	250	Edlib (path)	0.4536	92
250bp/mutated_70_perc.fasta	250	LandauVishkin	0.2510	92
250bp/mutated_80_perc.fasta	250	Edlib	0.4505	55
250bp/mutated_80_perc.fasta	250	Edlib (path)	0.4560	55
250bp/mutated_80_perc.fasta	250	LandauVishkin	0.1032	55
250bp/mutated_90_perc.fasta	250	Edlib	0.4520	32
250bp/mutated_90_perc.fasta	250	Edlib (path)	0.4543	32
250bp/mutated_90_perc.fasta	250	LandauVishkin	0.0456	32
250bp/mutated_94_perc.fasta	250	Edlib	0.4510	16
250bp/mutated_94_perc.fasta	250	Edlib (path)	0.4547	16
250bp/mutated_94_perc.fasta	250	LandauVishkin	0.0224	16
250bp/mutated_97_perc.fasta	250	Edlib	0.4518	10
250bp/mutated_97_perc.fasta	250	Edlib (path)	0.4537	10
250bp/mutated_97_perc.fasta	250	LandauVishkin	0.0168	10
250bp/mutated_99_perc.fasta	250	Edlib	0.4533	5
250bp/mutated_99_perc.fasta	250	Edlib (path)	0.4542	5
250bp/mutated_99_perc.fasta	250	LandauVishkin	0.0145	5

Tablica 5.3: Tablica koja prikazuje rezultate testiranja implementacije prefiksnog algoritma Landau-Vishkin-Nussinov i alata edlib na uzorcima duljine 250 proteinskih baza.

Uzorak	m	Algoritam	Vrijeme	Udaljenost
500bp/mutated_60_perc.fasta	500	Edlib	0.4566	239
500bp/mutated_60_perc.fasta	500	Edlib (path)	0.4609	239
500bp/mutated_60_perc.fasta	**500**	**LandauVishkin**	**3.7646**	**239**
500bp/mutated_70_perc.fasta	500	Edlib	0.4558	180
500bp/mutated_70_perc.fasta	500	Edlib (path)	0.4606	180
500bp/mutated_70_perc.fasta	**500**	**LandauVishkin**	**1.9114**	**180**
500bp/mutated_80_perc.fasta	500	Edlib	0.4532	108
500bp/mutated_80_perc.fasta	500	Edlib (path)	0.4591	108
500bp/mutated_80_perc.fasta	500	LandauVishkin	0.3591	108
500bp/mutated_90_perc.fasta	500	Edlib	0.4519	63
500bp/mutated_90_perc.fasta	500	Edlib (path)	0.4666	63
500bp/mutated_90_perc.fasta	500	LandauVishkin	0.1533	63
500bp/mutated_94_perc.fasta	500	Edlib	0.4560	28
500bp/mutated_94_perc.fasta	500	Edlib (path)	0.4592	28
500bp/mutated_94_perc.fasta	500	LandauVishkin	0.0536	28
500bp/mutated_97_perc.fasta	500	Edlib	0.4611	15
500bp/mutated_97_perc.fasta	500	Edlib (path)	0.4798	15
500bp/mutated_97_perc.fasta	500	LandauVishkin	0.0349	15
500bp/mutated_99_perc.fasta	500	Edlib	0.4529	7
500bp/mutated_99_perc.fasta	500	Edlib (path)	0.4577	7
500bp/mutated_99_perc.fasta	500	LandauVishkin	0.0291	7

Tablica 5.4: Tablica koja prikazuje rezultate testiranja implementacije prefiksnog algoritma Landau-Vishkin-Nussinov i alata edlib na uzorcima duljine 500 proteinskih baza. Podebljano su označeni redci u kojima je algoritam Landau-Vishkin-Nussinov sporiji od edliba.

nanja i optimalnog puta, dok LandauVishkin označava korištenje naše implementacije. Sljedeći stupac prikazuje prosječno vrijeme izvođenja algoritma u sekundama, dok se u zadnjem stupcu nalazi prefiksna udaljenost uzorka i teksta d. Vidljivo je da vrijeme izvođenja algoritma Landau-Vishkin-Nussinov puno duže od edliba. Do toliko loših rezultata dolazi zbog toga što vremenska složenost infiksnog algoritma iznosi $O(k^2n)$, a n za tekst na kojem je algoritam testiran iznosi oko pet milijuna. Još jedan od problema je taj što ne poznajemo vrijednost konstante k te zbog toga moramo isprobati sve moguće vrijednosti $k \leq m$, što nam vremensku složenost pretvara u $O(mk^2n)$. Iz toga je vidljivo da će infiksni algoritam Landau-Viskin-Nussinov raditi brzo za slične, kratke nizove.

Uzorak	m	Algoritam	Vrijeme	d
50bp/e_coli_DH1_illumina_1x50.fasta	50	Edlib	0.0399	0
50bp/e_coli_DH1_illumina_1x50.fasta	50	Edlib (path)	0.0428	0
50bp/e_coli_DH1_illumina_1x50.fasta	50	LandauVishkin	2.0580	0
50bp/mutated_90_perc.fasta	50	Edlib	0.0401	2
50bp/mutated_90_perc.fasta	50	Edlib (path)	0.0427	2
50bp/mutated_90_perc.fasta	50	LandauVishkin	23.4833	2
50bp/mutated_94_perc.fasta	50	Edlib	0.0402	4
50bp/mutated_94_perc.fasta	50	Edlib (path)	0.0426	4
50bp/mutated_94_perc.fasta	50	LandauVishkin	90.2955	4
50bp/mutated_97_perc.fasta	50	Edlib	0.0400	3
50bp/mutated_97_perc.fasta	50	Edlib (path)	0.0429	3
50bp/mutated_97_perc.fasta	50	LandauVishkin	50.1223	3
100bp/e_coli_DH1_illumina_1x100.fasta	100	Edlib	0.0612	0
100bp/e_coli_DH1_illumina_1x100.fasta	100	Edlib (path)	0.0631	0
100bp/e_coli_DH1_illumina_1x100.fasta	100	LandauVishkin	2.0877	0
100bp/mutated_90_perc.fasta	100	Edlib	0.0547	6
100bp/mutated_90_perc.fasta	100	Edlib (path)	0.0565	6
100bp/mutated_90_perc.fasta	100	LandauVishkin	243.8266	6
100bp/mutated_94_perc.fasta	100	Edlib	0.0540	7
100bp/mutated_94_perc.fasta	100	Edlib (path)	0.0565	7
100bp/mutated_94_perc.fasta	100	LandauVishkin	356.0862	7
100bp/mutated_97_perc.fasta	100	Edlib	0.0538	3
100bp/mutated_97_perc.fasta	100	Edlib (path)	0.0564	3
100bp/mutated_97_perc.fasta	100	LandauVishkin	51.1574	3

Tablica 5.5: Tablica koja prikazuje rezultate testiranja implementacije infiksnog algoritma Landau-Vishkin-Nussinov i alata edlib na uzorcima duljine 50 i 100 proteinskih baza.

6. Zaključak

Algoritam koji su 1986. godine predložili Landau, Vishkin i Nussinov (Landau et al., 1986) rješava problem traženja infiksnog poravnanja između dva niza, uzorka duljine m i teksta duljine n ($n \geq m$), s vremenskom složenosti $O(k^2n)$, gdje k predstavlja maksimalni dopušteni broj razlika između nizova. Izvod vremenske složenosti dan je u potpoglavlju 3.3.4. Vidimo da ovdje dobivamo najveće ubrzanje s obzirom na jednostavne algoritme ako radimo sa sličnim, kratkim nizovima. S druge strane, edlib (Šošić i Šikić, 2017) je alat koji također rješava problem traženja poravnanja između dva niza, ali najveće ubrzanje postiže na dugim, sličnim nizovima. Zbog toga, odlučili smo se napraviti implementaciju algoritma Landau-Vishkin-Nussinov koju smo potom ugradili u edlib, kako bi omogućili brže rješavanje problema za kraće nizove. Uz infiksni algoritam, implementirali smo i prefiksnu varijaciju algoritma, jer edlib omogućava traženje prefiksnog, infiksnog i globalnog poravnanja nizova. Osim integracije u edlib, implementacija algoritma dostupna je i kao zasebna biblioteka, te ju je moguće preuzeti s web stranice: https://github.com/donikv/Zavrsni_Rad.

Rezultati usporedbe implementecije algoritma Landau-Vishkin-Nussinov s edlib alatom pokazuju da je edlib u prosjeku 10 puta sporiji prilikom traženja optimalnog prefiksnog poravnanja za kratke nizove. Razlika u brzini je najveća prilikom traženja poravnanja za vrlo kratke uzorke, od oko 50 do 250 proteinskih baza, dok je za nizove od 250 do 500 bp brzina naša implementacija je skoro svaki puta brža od edliba, osim za najduže nizove s velikom udaljenosti između uzorka i teksta ($k > 150$). S druge strane, edlib je brži prilikom traženja globalnog i infiksnog poravnanja, uz nepoznati k, dok je kod infiksnog poravnanja za poznati i mali k sporiji. Detaljnija obrada rezultata dana je u poglavlju 5.

Uz to, implementacija algoritama prilagođena je tako da, uz traženje udaljenosti između nizova, omogućava i određivanje puta za koje je ta udaljenost postignuta. Da bi implementacija bila kompatibilna s edlibom, put je zapisan u formatu CIGAR, koji je opisan u poglavlju 2.3.

Daljnji rad na implementaciji uključuje prilagodbu rada globalnog i infiksnog al-

goritma kako bi dobili bolje rezultate s obzirom na edlib. Uz to, bilo bi dobro osmisliti kvalitetnu heurističku funkciju koja bi na temelju ulaznih nizova i vrste poravnanja dobro određivala koji od algoritama treba primijeniti kako bi dobili što veće ubrzanje.

LITERATURA

D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, Lipanj 1975. ISSN 0001-0782. doi: 10.1145/360825.360861. URL http://doi.acm.org/10.1145/360825.360861.

G. M. Landau, U. Vishkin, i R. Nussinov. An efficient string matching algorithm with k differences for nucleotide and amino acid sequences. *Nucleic Acids Res*, 14(1): 31–46, Jan 1986. ISSN 0305-1048. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC339353/. 3753770[pmid].

Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, Svibanj 1999. ISSN 0004-5411. doi: 10.1145/316542.316550. URL http://doi.acm.org/10.1145/316542.316550.

Saul B. Needleman i Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970. ISSN 0022-2836. doi: https://doi.org/10.1016/0022-2836(70)90057-4. URL http://www.sciencedirect.com/science/article/pii/0022283670900574.

T.F. Smith i M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981. ISSN 0022-2836. doi: https://doi.org/10.1016/0022-2836(81)90087-5. URL http://www.sciencedirect.com/science/article/pii/0022283681900875.

Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1):100 – 118, 1985. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(85)80046-2. URL http://www.sciencedirect.com/science/article/pii/S0019995885800462. International Conference on Foundations of Computation Theory.

Martin Šošić i Mile Šikić. Edlib: a c/c ++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017. doi: 10.1093/bioinformatics/btw753. URL http://dx.doi.org/10.1093/bioinformatics/btw753.

Landau-Vishkin-Nussinov algoritam za poravnanje dva niza

Sažetak

Vremenska i memorijska složenost optimalnog poravnanja dva niza je kvadratna što za dulje nizove rezultira jako dugačkim vremenom izvršavanja. Međutim, često imamo informaciju o tome da su nizovi slični i možemo unaprijed ograničiti kolika će biti razlika među njima. U tom slučaju koristimo algoritme koji u matrici poravnanja računaju samo glavnu dijagonalu i određen broj susjednih. Jedan od najbržih takvih algoritama opisan je u radu "An efficient string matching algorithm with k differences for nucleotide and amino acid sequences". U ovome radu predstavljena je implementacija tog algoritma, te je algoritam prilagođen kako bi se mogao ugraditi u biblioteku edlib, gdje je zadužena za traženje poravnanja između kratkih nizova.

Ključne riječi: bioinformatika, Landau-Viskin-Nussinov, edlib, poravnanje nizova, Levenshteinova udaljenost

Landau-Vishkin-Nussinov Algorithm for Pair-wise Sequence Alignment

Abstract

Time and memory complexity of optimal pair-wise sequence alignment is quadratic, which, for longer sequences, results in very long computation time. However, we usually have the information that the sequences are similar and we can limit the maximal edit distance between them. In that case, we use algorithms which in the alignment matrix calculate only the main diagonal and a fixed number of neighboring diagonals. One of the fastest such algorithms is described in the paper titled "An efficient string matching algorithm with k differences for nucleotide and amino acid sequences". In this paper, we present an implementation of that algorithm, and we show how the algorithm was adapted to be included into edlib library, where it is used to determine pair-wise sequence alignment of shorter sequences.

Keywords: bioinformatics, Landau-Vishkin-Nussinov, edlib, pair-wise sequence alignment, Levenshtein's distance