

BIOINFORMATIKA

8. juni 2018.

Sadržaj

1 Gde u genomu počinje replikacija genoma?	1
1.1 Uvod	1
1.2 Replikacija genoma	2
1.2.1 DNK	2
1.2.2 Replikacija genoma u ćeliji	3
1.2.3 Pronalaženje početnog regiona replikacije	9
1.3 Zadaci sa vežbi	13
1.3.1 FrequentWords	13
1.3.2 Faster FrequentWords	13
1.3.3 Skew Diagram	15
1.3.4 FrequentWords With Mismatches	15
2 Koji DNK šabloni igraju ulogu molekularnog sata?	19
2.1 Biološki problem	19
2.2 Informatički problem	21
2.3 Problem ubačenog motiva	21
2.3.1 Enumeracija motiva	22
2.3.2 Najsličniji k-grami u parovima niski	23
2.3.3 Matrice motiva	24
2.3.4 Problem niske medijane	27
2.3.5 Probabilistički pristup	28
2.3.6 Koji princip odabrati?	35
2.4 Zadaci sa vežbi	35
2.4.1 <i>MedianString</i>	35
2.4.2 <i>GreedyMotifSearch</i>	37
2.4.3 <i>RandomizedMotifSearch</i>	40
2.4.4 <i>GibbsSampler</i>	42
3 Kako složiti genomsku slagalicu od milion delova?	47
3.1 Šta je sekvenciranje genoma?	47
3.1.1 Kratka istorija sekvenciranja genoma	47
3.1.2 Sekvenciranje ličnih genoma	48
3.2 Eksplozija u štampariji	49
3.3 Problem sekvenciranja genoma	50
3.4 Rekonstrukcija niske kao problem Hamiltonove putanje	52
3.4.1 Genom kao putanja	52
3.5 Rekonstrukcija niske kao Ojlerove putanje	53
3.6 De Brojnovi grafovi na osnovu kolekcije k -grama	54
3.7 Ojlerova teorema	55
3.7.1 Dokaz Ojlerove teoreme	55
3.8 Sastavljanje parova očitavanja	57

3.8.1	DNK sekvenciranje sa parovima očitavanja	57
3.9	U realnosti	59
3.10	Zadaci sa vežbi	61
3.10.1	Maximal Non Branching Path	61
3.10.2	All Euler Cycles	63
3.10.3	String Spelled By Gapped Patterns	66
4	Kako sekvenciramo antibiotike?	69
4.1	Otkriće antibiotika	69
4.2	Kako bakterije prave antibiotike?	70
4.3	Sekvenciranje antibiotika razbijanjem na komade	73
4.3.1	Sekvenciranje ciklopeptida grubom silom	74
4.3.2	<i>Branch-and-Bound</i> algoritam za sekvenciranje ciklopeptida	75
4.4	Prilagođavanje sekvenciranja za spekture sa greškama	76
4.4.1	Testiranje na spektru tirocidina B1	78
4.5	Od 20 do više od 100 aminokiselina	78
4.6	Spektralna konvolucija	78
4.7	Spektri u realnosti	80
4.8	Zadaci sa vežbi	81
4.8.1	Linear Spectrum	81
4.8.2	Cyclic Spectrum	81
4.8.3	Cyclopeptide Sequencing	82
4.8.4	Leaderboard Cyclopeptide Sequencing	84
5	Kako poredimo biološke sekvence?	89
5.1	Biološki uvid u poređenje sekvenci	89
5.2	Igra poravnanja i najduža zajednička podsekvencia	89
5.2.1	Najduža zajednička podsekvencia	90
5.3	Problem turiste na Menhetnu	90
5.4	Problem kusura	93
5.4.1	Pohlepni algoritam	93
5.4.2	Rekurzivni algoritam	94
5.4.3	Vraćanje kusura dinamičkim programiranjem	95
5.5	Dinamičko programiranje i putokazi za povratak	96
5.5.1	Rekurentna relacija dinamičkog programiranja kod Menhetn grafa	98
5.6	Od Menhetna do grafa poravnjanja	98
5.6.1	Rekurentna relacija dinamičkog programiranja kod grafa poravnjanja	98
5.6.2	Računanje putokaza za povratak	100
5.6.3	Određivanje najduže zajedničke podsekvence (LCS – longest common subsequence) korišćenjem putokaza za povratak	101
5.6.4	Topološko sortiranje	101
5.7	Od globalnog do lokalnog poravnjanja	102
5.7.1	Globalno poravnanje	103
5.7.2	Lokalno poravnanje	104
5.8	Kažnjavanje insercija i delecija u poravnaju sekvenci	107
5.8.1	Kažnjavanje praznina	107
5.8.2	Adekvatnije kazne za praznine	107
5.8.3	Izgradnja Menhetn grafa sa afnim kaznama za praznine	107
5.9	Prostorno efikasno poravnanje sekvenci	109
5.9.1	Prostorna složenost	111
5.9.2	Vremenska složenost	112
5.10	Višestruko poravnanje sekvenci	113
5.10.1	Od dvostrukog do višestrukog poravnanja	113

5.10.2	Poravnanje tri A-domena	114
5.10.3	Generalizacija dvostrukog na višestruko poravnanje	114
5.10.4	Poravnaj = 3-D putanje	115
5.10.5	2-D poravnanje u odnosu na 3-D poravnanje	115
5.10.6	Rekurentna relacija dinamičkog programiranja za višestruko poravnanje	116
5.10.7	Vremenska složenost dinamičkog algoritma za višestruko poravnanje	116
5.10.8	Da li se višestruko poravnanje može izgraditi iz dvostrukog?	116
5.10.9	Profilna reprezentacija višestrukog poravnanja	117
5.10.10	Višestruko poravnanje: pohlepni pristup	117
5.11	Zadaci sa vežbi	118
5.11.1	Manhattan Tourist	118
5.11.2	LCS Backtrack	119
5.11.3	Global Alignment	120
5.11.4	Local Alignment	121
5.11.5	Edit Distance	122
6	Postoje li osetljivi delovi u ljudskom genomu?	125
6.1	Transformacija čoveka u miša	125
6.2	Sortiranje po promenama	129
6.3	Teorema u prekidnoj tački	132
6.4	Preuređivanje u multihromozomalnim genomima	134
6.4.1	Translokacije, fuzije i fizije	134
6.5	Problem rastojanja 2-prekida	135
6.5.1	Od linearnih do cirkularnih hromozoma	135
6.5.2	Rastojanje 2-prekida	138
6.6	Grafovi prekidnih tačaka	139
6.7	Teorema o rastojanju 2-prekida	144
6.8	Zadaci sa vezbi	147
6.8.1	ChromosomeToCycle	147
6.8.2	CycleToChromosome	147
6.8.3	GreedySorting	148
6.8.4	ShortestRearrangementScenario	149
7	Kako locirati mutacije koje izazivaju bolesti?	153
7.1	Mapiranje očitavanja	153
7.1.1	Egzaktno upativanje šablonu	156
7.1.2	Moguća rešenja	156
7.1.3	Sufiksna stabla	157
7.2	Kompresija niski i Barouz-Vilerova transformacija	162
7.2.1	Kompresija genoma	162
7.3	Inverzna BWT	164
7.3.1	Efikasnija BWT dekompresija	165
7.3.2	Korišćenje BWT za uparivanje šablonu	165
7.3.3	Pronalaženje uparenih šablonu	166
7.3.4	Pronalaženje uparenih šablonu	166
7.4	Približno preklapanje	168
7.5	Zadaci sa vezbi	169
7.5.1	TrieConstruction	169
7.5.2	SufixReconstruction	170
7.5.3	BWT	172

8 Zašto naučnici i dalje nisu razvili vakcinu za HIV	175
8.1 Uvod	175
8.1.1 Klasifikacija HIV fenotipa	175
8.2 Pronalaženje CG ostrva	176
8.3 Nepoštena kockarnica	177
8.3.1 Kockanje sa Jakuzama	177
8.3.2 Dva novčića u krupijeovom rukavu	178
8.4 Skriveni Markovljevi modeli	179
8.4.1 Od bacanja novčića do Skrivenog Markovljevog Modela	179
8.4.2 HMM dijagrami	179
8.5 Problem dekodiranja	181
8.5.1 Viterbi graf	181
8.5.2 Viterbi algoritam	182
8.5.3 Brzina Viterbi algoritma	182
8.6 Računanje najverovatnijeg ishoda HMM-a	183
8.7 Profilni algoritmi za poravnjanje sekvenci	184
8.7.1 Kako su HMM povezani za poravnjanje sekvenci?	184
8.7.2 Građenje profilnog HMM-a	187
8.8 Zadaci sa vežbi	191
8.8.1 HMM	191

Predgovor

Tekst se sastoji od proširenih beleški sa predavanja na osnovu knjige Pavel A. Pevzner, Phillip Compeau: Bioinformatics Algorithms: An Active Learning Approach.

Tekst su sastavili studenti sa kursa održanog u školskoj 2017/2018 godini:

- Una Stanković 1095/2016
- Marina Nikolić 1055/2017
- Strahinja Milojević 1049/2017
- Anja Bukurov 1082/2016
- Nikola Ajzenhamer 1083/2016
- Vojislav Stanković 1080/2016
- Milica Đurić 1084/2016
- Ana Stanković 1096/2016
- Aleksandra Branković 1057/2017
- Ljubica Aćimović 1027/2016
- Jasmina Vasilijević 1067/2017
- Andjela Mijailović 1050/2017
- Milena Dukanac 1020/2017
- Filip Miljaković 1040/2017
- Petar Kulezić 1058/2017

Glava 1

Gde u genomu počinje replikacija genoma?

1.1 Uvod

Na samom početku, želimo da definišemo pojam bioinformatike i da pokušamo da shvatimo koji je njen osnovni cilj. Da bismo to postigli, pogledajmo tri definicije, iz različitih izvora:

- "Bioinformatika je nauka koja se bavi prikupljanjem i analizom kompleksnih bioloških podataka poput genetskih kodova." - Oksfordski rečnik (engl. *Oxford Dictionary*)
- "Bioinformatika predstavlja prikupljanje, klasifikaciju, čuvanje i analizu biohemijskih i bioloških informacija korišćenjem računara, a posebno se primenjuje u molekularnoj genetici i genomici." - Rečnik Merriam-Vebster (engl. *Merriam-Webster Dictionary*)
- "Bioinformatika je interdisciplinarno polje koje radi na razvoju metoda i softverskih alata za razumevanje bioloških podataka." - Vikipedija (engl. *Wikipedia*)

Na osnovu ove tri definicije možemo zaključiti da:

Bioinformatika predstavlja primenu računarskih tehnologija u istraživanjima u oblasti biologije i srodnih nauka.

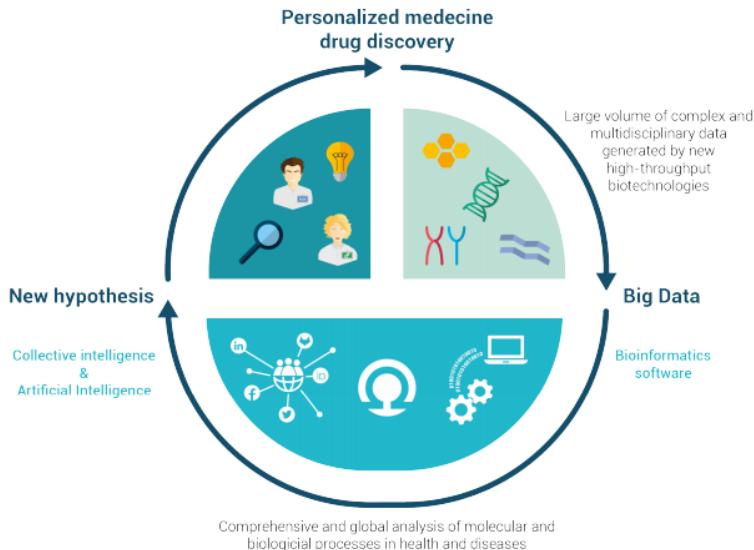
Bioinformatika ima široku primenu i njene primene rastu zajedno sa razvojem discipline. Kao što možemo videti na slici ispod, primena bioinformatike se može sagledati kroz personalizovani medicinu. Naime, na osnovu prikupljene veće količine podataka i njihove analize, uz pomoć različitih računarskih metoda, na primer metoda veštačke inteligencije, možemo doći do informacija potrebnih da na najbolji način lečimo pacijenta ili mu odredimo terapiju koja će mu na najbolji, najbrži i najbezboljniji način pomoći da prevaziđe određene zdravstvene probleme.

Bioinformatika je spoj više različitih disciplina, kao što su:

- Statistika
- Istraživanje podataka
- Računarstvo
- Računarska biologija
- Biologija
- Biostatistika

Prikaz preklapanja ovih disciplina možemo videti na slici 1.2.

Slika 1.1: Primena bioinformatike



1.2 Replikacija genoma

1.2.1 DNK

Dezoksiribonukleinska kiselina (akronimi DNK ili DNA, od engl. *deoxyribonucleic acid*), nukleinska kiselina koja sadrži uputstva za razvoj i pravilno funkcionisanje svih živih organizama. Zajedno sa RNK i proteinima, DNK je jedan od tri glavna tipa makromolekula koji su esencijalni za sve poznate forme života.

Sva živa bića svoj genetički materijal nose u obliku DNK, sa izuzetkom nekih virusa koji imaju ribonukleinsku kiselinu (RNK). DNK ima veoma važnu ulogu ne samo u prenosu genetičkih informacija sa jedne na drugu generaciju, već sadrži i uputstva za građenje neophodnih ćelijskih organela, proteina i RNK molekula. DNK segment koji sadrži ova važna uputstva se naziva gen.

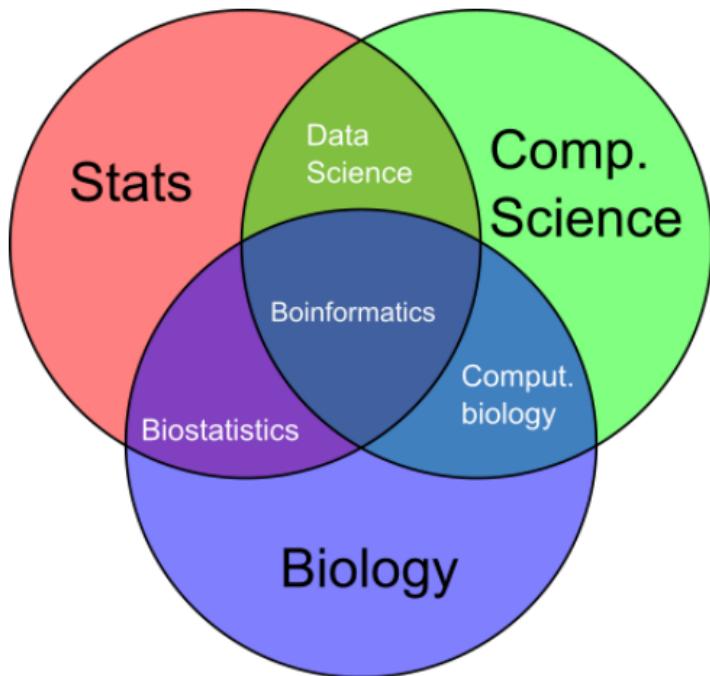
DNK se sastoji iz dva polimerna lanca koji imaju antiparalelnu orientaciju, i svaki od njih je sastavljen od azotnih baza:

- adenin (A)
- timin (T)
- guanin (G)
- citozin (C)

Lanci DNK su međusobno spojeni i to tako da se veze uspostavljaju isključivo između adenina i citozina ili između guanina i timina. Na osnovu toga, ako nam je poznat sastav jednog lanca, lako možemo zaključiti i sastav drugog lanca, zbog čega se kaže da su DNK lanci **međusobno komplementarni**.

Da bismo lakše manipulisali sa informacijama koje DNK nosi i približili sadržaj računarskoj struci, DNK ćemo posmatrati kao nisku nad azbukom *A,C,G,T*.

Slika 1.2: Preklapanjem različitih disciplina dobijamo bioinformatiku.



1.2.2 Replikacija genoma u ćeliji

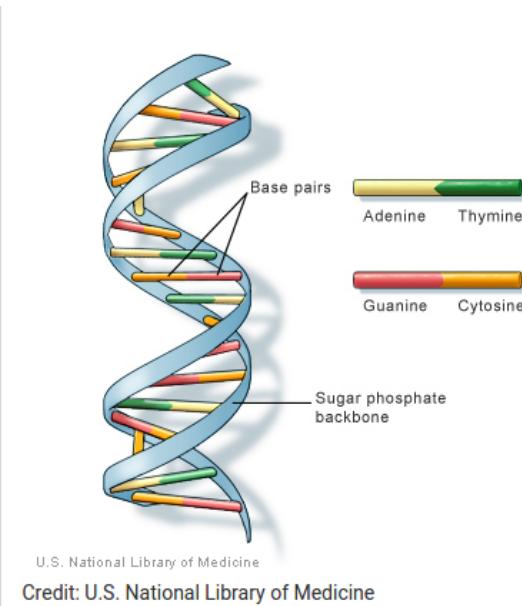
Replikacija genoma je jedan od najvažnijih zadataka ćelije. Pre nego što se podeli, ćelija mora da najpre replicira svoj genom, tako da svaka od čerki ćelija dobije svoju kopiju.

Dzejms Votson (engl. *James Watson*) i Fransis Krik (engl. *Fransis Crick*) su 1953. godine napisali rad u kome su primetili da postoji mehanizam za kopiranje genetskog materijala. Oni su uočili da se lanci roditeljskog DNK molekula odvijaju tokom replikacije i da se, potom, svaki lanac ponaša kao uzorak za sintezu novog lanca (na osnovu toga što se uvek spajaju iste amionokiseline A-C i G-T, rekreiranje lanca je moguće). Kao rezultat ovakvog ponašanja, proces replikacije počinje parom komplementarnih lanaca i završava se sa dva para komplementarnih lanaca, kao što se može videti na slici ispod.

Replikacija počinje u regionu genoma koji se naziva **početni region replikacije** (skraćeno *oriC*), izvode je enzimi koje se nazivaju DNK polimeraze, koje predstavljaju mašine za kopiranje na molekularnom nivou.

Nalaženje početnog regiona replikacije predstavlja veoma važan problem, ne samo za razumevanje funkcionalisanja kako se ćelije repliciraju, već je koristan i u raznim biomedicinskim problemima. Na primer, neki metodi genskih terapija uključuju genetski izmenjene mini genome, koji se zovu virusni vektori, zbog svoje sposobnosti da prodržu kroz ćelijski zid (poput pravih virusa). Virusni vektori u sebi nose veštačke gene koji unapredaju postojeći genom. Genska terapija je prvi put uspešno izvršena 1990. godine na devojčici koja je bila toliko otporna na infekcije da je bila primorana da živi isključivo u sterilnom okruženju.

Osnovna ideja genske terapije je da se pacijent, koji pati od nedostatka nekog bitnog gena, zarazi viralnim vektorom koji sadrži veštački gen koji enkodira terapeutski protein. Jednom kad

Slika 1.3: Prikaz DNK, slika preuzeta sa <https://ghr.nlm.nih.gov/primer/basics/dna>

je unutar ćelije, vektor se replicira, što dovodi do lečenja bolesti pacijenta. Da bi moglo da dodje do ovoga, biolozima je neophodno da znaju gde je *oriC*.

Kako ćelija prepoznaje *oriC*?

Pitamo se kako ćelija prepoznaje *oriC*? Sigurno je da postoji neka niska aminokiselina koja označava *oriC*, ali kako ga prepoznati?

Ograničimo se na bakterijski genom, koji se sastoji od jednog kružnog hromozoma. Istraživanje je pokazalo da je region, koji predstavlja *oriC* kod bakterija, dug svega nekoliko stotina nukleotida.

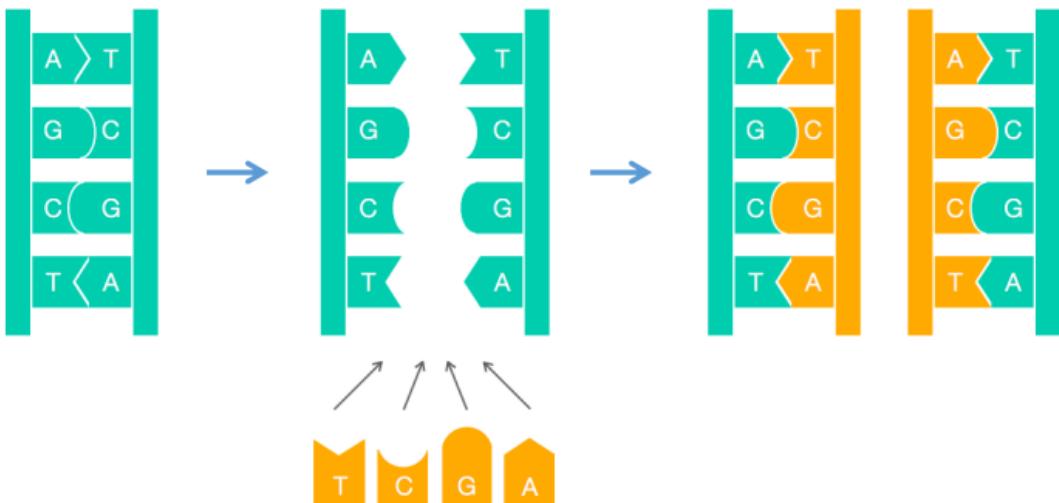
Poznato je da DNKA utiče na početak replikacije. *DNKA* je protein koji se vezuje na kratki segment unutar *oriC*, poznatiji kao **DNKA boks**. Ona predstavlja poruku unutar sekvence DNK koja govori proteinu DNKA da se veže baš tu. Postavlja se pitanje kao pronaći taj region bez prethodnog poznavanja izgleda DNKA boks?

Da bismo bolje razumeli *problem skrivene poruke* uzmimo za primer priču Edgara Alana Poa - "Zlatni jelenak" (engl. "The Gold-Bug"). Naime, u toj priči jedan od likova, Vilijam Legrand (engl. *William Legrand*), treba da dešifruje poruku :

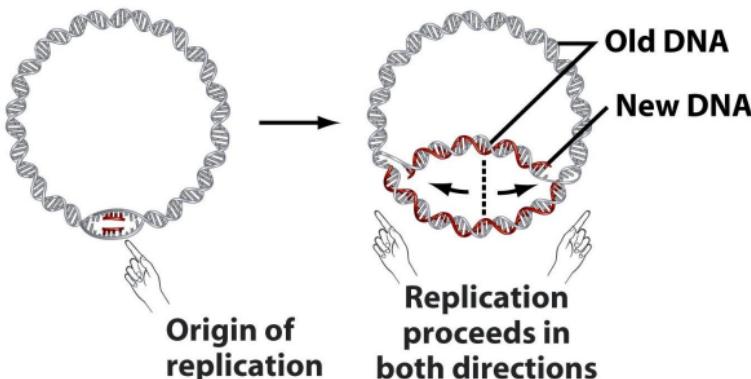
```
53++!305))6*;4826)4+.)4+);806*;48!8'6 0))85;]8*:+*8!83(88)5*!;46(;88*96*?;8
)*+(;485);5*!2:+*(;4956*2(5*4)8'8*;40 69285);)6!8)4++;1(+9;48081;8:8+1;48!8 5;4)
485!528806*81(+9;48;(88;4(+?34;48 )4+;161;:188;+?;
```

On uočava da se ";48" pojavljuje veoma često, i da verovatno predstavlja "THE", najčešću reč u engleskom jeziku. Znajući to, zamenjuje karaktere odgovarajućim slovima i postepeno dešifruje celu poruku.

Slika 1.4: Prikaz replikacije



Slika 1.5: Prikaz početka replikacije kod bakterija



```

53++!305))6*THE26)H+.)H+)806*THE !E'60))E5;]E*:+*E!E3(EE)5*!TH6(T EE*96*?;E)*+
(THE5)T5*!2:+*(TH956 *2(5*H)E'E*TH0692E5)T)6!E)H++T1(+ +9THE0E1TE:E+1
THE!E5T4)HE5!52880 6*E1(+9THE(+?34THE)H+T161T :1EET+?T

```

Želeli bismo da ovaj princip primenimo na naš problem nalaska *oriC*-a. Ideja je da uvidimo da li postoje reči koje se neuobičajeno često pojavljuju. Uvedimo termin k-gram da označimo string dužine k i COUNT(Text, Pattern) da označimo broj puta kojih se k-gram *Pattern* pojavio u tekstu *Text*. Osnovna ideja je da pomeramo prozor, iste dužine kao k-gram *Pattern*, niz tekst, usput proveravajući da li se pojavljuje *Pattern* u nekome od njih.

```

1  PATTERNCOUNT(Text, Pattern)
2      count = 0
3      for i = 0 to |Text| - |Pattern|
4          if Text(i,|Pattern|) = Pattern
5              count = count + 1
6      return count

```

Za neki *Pattern* kažemo da je on *najčešći k-gram* u tekstu *Text*, ako je njegov *COUNT* najveći među svim k-gramima. Na primer, **ACTAT** je najčešći 5-gram u tekstu *Text* = ACAACTATGCAACTATCGGGACAACTATCCT, a **ATA** je najčešći 3-gram u *Text* = CGATATATCCATAG.

Sada, problem pronaletača čestih reči možemo posmatrati kao računarski problem:

Problem čestih reči: Pronaći najčešće k-grame u niski karaktera.

Ulaz: Niska *Text* i ceo broj *k*.

Izlaz: Svi najčešći k-grami u niski *Text*.

Osnovni algoritam za pronaletačak čestih k-grama u stringu *Text* proverava sve k-grame koji se pojavljuju u tom stringu (takvih k-grama ima $|Text| - k + 1$) i potom izračunava koliko puta se svaki k-gram pojavljuje. Da bismo implementirali ovaj algoritam, moramo da izgenerišemo niz *COUNT*, gde je $COUNT(i) = COUNT(Text, Pattern)$ za $Pattern = Text(i, k)$.

```

1 FrequentWords(Text, k)
2     FrequentPatterns <- an empty set
3     for i = 0 to |Text| - k
4         Pattern <- the k-mer Text(i,k)
5         COUNT(i) <- PatternCount(Text, Pattern)
6     maxCount <- max value in array COUNT
7     for i = 0 to |Text| - k
8         if COUNT(i) = maxCount
9             add Text(i,k) to FrequentPatterns
10    remove duplicates from FrequentPatterns
11    return FrequentPatterns

```

Pitamo se, sada, kolika je složenost ovakvog pristupa?

Ovaj algoritam, iako uspešno nalazi ono što se od njega traži, nije najefikasniji. S obzirom na to da svaki k-gram zahteva $|Text| - k + 1$ provera, svaki od njih zahteva i do *k* poređenja, pa je broj koraka izvršavanja funkcije *PatternCount*(*Text*, *Pattern*) zapravo $(|Text| - k + 1) * k$. Osim toga, *FrequentWords* mora pozvati *PatternCount* $|Text| - k + 1$ puta (po jednom za svaki k-gram teksta), tako da je ukupan broj koraka $(|Text| - k + 1) * (|Text| - k + 1) * k$.

Iz navedenog, možemo zaključiti da je ukupna cena izvršavanja algoritma *FrequentWords* $O(|Text|^2 * k)$.

Primer: Pronaletačak čestih reči kod bakterije *Vibrio cholerae*

Posmatrajmo, najpre, tablicu najčešćih k-grama u *oriC* regionu bakterije *Vibrio cholerae*. Da li nam se čini da se neki k-grami pojavljuju neuobičajeno često?

Na primer, 9-gram **ATGATCAAAG** se pojavljuje tri puta u *oriC* regionu, da li nas to iznenađuje?

Označili smo najčešće 9-grame, umesto nekih drugih k-grama, jer je eksperimentima pokazano da su DNKA boksovi kod bakterija dugi 9 nukleotida. Uočimo da postoje četiri različita 9-grama koji se ponavljaju tri ili više puta u ovom regionu, to su: ATGATCAAAG, CTTGAT-CAT, TCTTGATCA i CTCTTGATC.

Slika 1.6: Tablica najčešćih k-grama u *oriC* regionu bakterije *Vibrio cholerae*

k	3	4	5	6	7	8	9
count	25	12	8	8	5	4	3
k -mers	tga	atga	gatca	tgatca	atgatca	atgatcaa	atgatcaag
			tgatc			cttgatcat	
					tcttgatca		
						ctcttgatc	

Slika 1.7: Prikaz 9-grama ATGATCAAG i njegovog komplementa u *oriC* regionu *Vibrio cholerae*

atcaatgatcaacgtaaagcttctaagc **ATGATCAAG** gtgctcacacagtttatccacaacctgagtgg
atgacatcaagataaggcgttgtatctccttcgtactctcatgaccacggaaag **ATGATCAAG**
agaggatgattctggccatatcgcaatgaatacttgtgacttgtgctccaattgacatcttcagc
gccatattgcgtggcaagggtgacggagcgggattacgaaagcatgatcatggcttgttctgttt
atcttggtttgactgagacttgttaggatagacggtttcatcaactgactagccaaagccttactct
gcctgacatcgaccgtaaattgataatgaatttacatgcttcgcgacgatttacct **CTTGATCAT** cg
atccgattgaagatctcaattgttaattcttgcctcgactcatagccatgatgagct **CTTGATCA**
Tqtttccttaacccttattttacgaaag **ATGATCAAG** ctqctqct **CTTGATCAT** cqtttc

Mala verovatnoća da se neki 9-gram toliko puta pojavi u *oriC*-u kolere, govori nam da neki od četiri 9-grama koje smo pronašli može biti potencijalni DNKA boks, koji započinje replikaciju. Ali, koji?

Podsetimo se da nukleotidi A i T, kao i C i G, su komplementarni. Ako imamo jednu stranu lanca DNK i neke slobodne nukleotide, možemo lako zamisliti sintezu komplementarnog lanca, kao što se vidi na slici ispod.

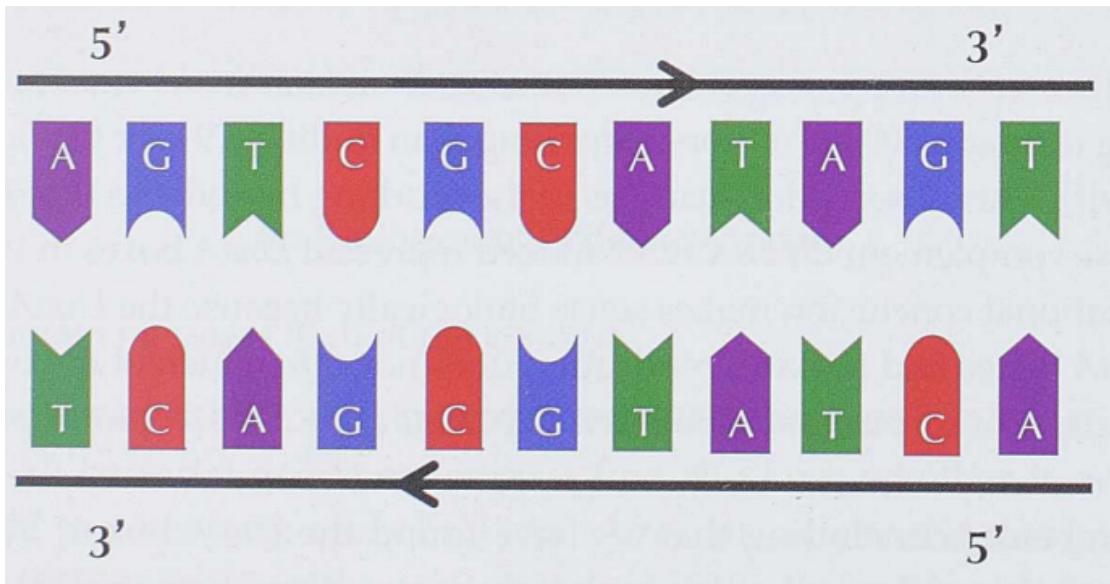
Posmatrajmo ponovo sliku 1.7. Na njoj možemo uočiti 6 pojavljivanja niski ATGATCAAG i CTTGATCAT, koji su zapravo komplementarni. Naći 9-gram koji se pojavljuje 6 puta u DNK nisci dužine 500 nukleotida, je još više iznenadjujuće, nego pronaći 9-gram koji se pojavljuje tri puta. Ovo posmatranje nas dovodi do toga da je ATGATCAAG (zajedno sa svojim komplementom) zaista DNKA boks *Vibrio cholerae*. Ovaj zaključak ima i smisla biološki, jer DNKA proteinu, kojii se vezuje i započinje replikaciju, nije bitno za kojii od dva lanca se vezuje.

Primer: Pronalazak čestih reči kod bakterije *Thermotoga petrophila*

Nakon što smo pronašli skrivenu poruku za *Vibrio cholerae*, ne bi trebalo da odmah zaključimo da je ta poruka ista kod svih bakterija. Najpre bi trebalo da proverimo da li se ona nalazi u *oriC* regionu drugih bakterija, možda različite bakterije, imaju drugačije DNKA boksove. Uzmimo, za primer, *oriC* region bakterije *Thermotoga petrophila*. Ona predstavlja bakteriju koja obitava u izrazito toplim regionima, na primer u vodi ispod rezervi nafte, gde temperature prelaze 80 stepeni Celzijusa. Pogledajmo kako izgleda *oriC* region ove bakterije.

Možemo lako uočiti da se u ovom regionu nigde ne javljaju niske ATGATCAAG ili CTT-GATCAT, iz čega zaključujemo da različite bakterije mogu koristiti različite DNKA boksove, kako bi pružile skrivenu poruku DNKA proteinu. Odnosno, za različite genome imamo različite DNKA boksove.

Slika 1.8: Komplementarni lanci se "kreću" u suprotnim smerovima.

Slika 1.9: Prikaz *oriC* regiona *Thermotoga petrophila*

```
aactctatacctccctttgtcgaaatttgtgtatagagaaaatcttattaactgaaactaa
aatggtaggtttgggttaggtttgtacatttgttagtatctgattttaattacataccgta
tattgtattaaattgacgaacaattgcattgaaattgaatatatgcaaaacaaacctaccaccaaac
tctgtattgaccattttaggacaacttcagggtggtaggttctgaagctctcatcaatagactat
tttagtctttacaaacaatattaccgttcagattcaagattctacaacgctgtttatggcggtt
gcagaaaaacttaccacctaataccgttatccaagccgattcagagaaaacctaccactacctac
cacttacctaccacccgggtggtaagttgcagacattatataaaacctcatcagaagctgttcaa
aaatttcaataactcgaaaccttaccacctgcgtcccattattactactaataatagcagta
taattgatctgaaaagagggtggtaaaaaaa
```

Najčešće reči u ovom *oriC* su:

- AACCTACCA,
- ACCTACCAC,
- GGTAGGTTT,
- TGGTAGGTT,
- AACCTACC,
- CCTACCACC

Pomoću alata koji se zove Ori-Finder, nalazimo CCTACCACC i njegov komplement GGTGG-TAGG kao potencijalne DNKA boksove naše bakterije. Ove dve niske se pojavljuju ukupno 5 puta.

Naučili smo da pronađemo skrivene poruke ako je *oriC* dat, ali ne znamo da pronađemo *oriC* u genomu.

Slika 1.10: Prikaz CCTACCACC i njenog komplementa u *oriC* regionu Thermotoga petrophila

```
aactctatacctccctttgtcgatttgtgatattatagagaaaatcttattaactgaaactaa
aatggtaggttGGTGGTAGGttttgtgtacattttgttagtatctgatttttaattacataccgtat
tattgtattaaattgacgaacaattgcatttgcattttatgcacaaacaaaCCTACCACCaaac
tctgtattgaccattttaggacaacttcagGGTGGTAGGtttctgaagctctcatcaatagactat
tttagtctttacaacaatattaccgttcagattcaagattctacaacgcgtttatggcggtt
gcagaaaacttaccacctaataatccagtatccaagccgatttcagagaaaacctaccactac
cacttaCCTACCACCcggtggtaagttgcagacattataaaaacctcatcagaagcttggtaaa
aaatttcaataactcgaaaCCTACCACCtgcgccccctattttactactaataatagcgtt
taatttgcgaaaagagggtggtaaaaaaa
```

1.2.3 Pronalaženje početnog regiona replikacije

Zamislimo da pokušavamo da nađemo *oriC* u novom sekvenciranom genomu bakterije. Ako bismo tražili niske poput ATGATCAAG/CTTGATCAT ili CCTACCACC/GGTGGTAGG to nam verovatno ne bi bilo puno od pomoći, jer novi genom može koristiti potpuno drugačiju skrivenu poruku. Posmatrajmo, zato, drugačiji problem: umesto da tražimo grupe određenog k-grama, pokušajmo da nađemo svaki k-gram koji formira grupu u genomu. Nadajmo se da će nam lokacije ovih grupa u genomu pomoći da odredimo lokaciju *oriC*-a.

Ideja je da pomeramo prozor fiksirane dužine L kroz genom, tražeći region u kome se k-gram pojavljuje više puta uzastopno. Za L ćemo uzeti vrednost 500, koja predstavlja najčešću dužinu *oriC*-a kod bakterija.

Definisali smo k-gram kao *grupu*, ako se pojavljuje više puta unutar kratkog intervala u genomu. Formalno, k-gram *Pattern* formira (L, t) grupu unutar niske *Genome* ako postoji interval genome, dužine L , u kome se k-gram pojavljuje barem t puta.

Problem pronalaženja grupa. Naći k-grame koji formiraju grupe unutar niske karaktera.

Ulas: Niska Genome i celi brojevi k (dužina podniske), L (dužina prozora) i t (broj podniski u grupi).

Izlaz: Svi k-grami koji formiraju (L, t) -grupe u niski Genome.

U genomu bakterije E.coli postoji 1904 različitih 9- grama koji formiraju $(500, 3)$ -grupe. Koji od njih ukazuje na početni region replikacije?

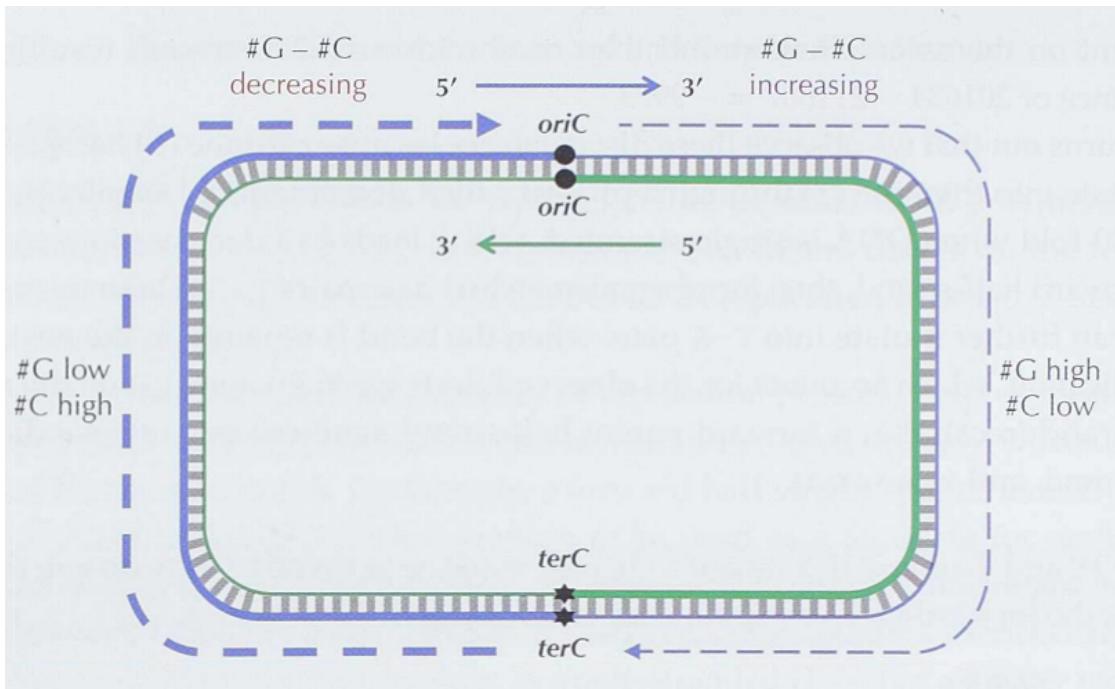
Iskrivljeni dijagrami

S obzirom na to da imamo veliku količinu statističkih podataka, pitamo se kako ih možemo upotrebiti da bismo došli do lokacije *oriC*-a? U tome nam mogu pomoći **iskrivljeni dijagrami**(engl. *skew diagram*). Osnovna ideja je da prođemo kroz genom i da računamo razliku između količine guanina(G) i citozina(C). Ako ova razlika raste, onda možemo prepostaviti da se krećemo niz polulanac koji ide na desno (u nastavku samo polulanac, smer $5' -> 3'$), a ako razlika počne da se smanjuje, onda prepostavljamo da smo na obrnutom polulancu ($3' -> 5'$). Zbog procesa koji se naziva deaminacija (gubljenje aminokiselina), svaki polulanac ima manjak citozina u poređenju sa guaninom, a svaki obrnuti polulanac ima manjak guanina u odnosu na citozin.

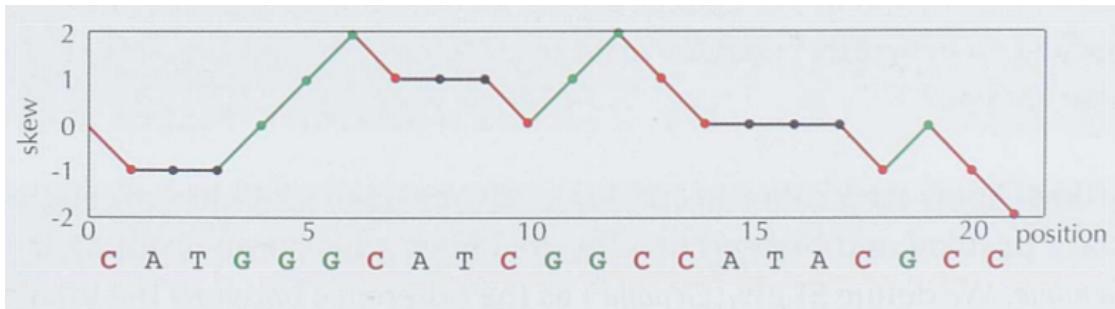
Posmatrajmo iskrivljeni dijagram bakterije Ešerihija Koli. Lako uočavamo minimalnu vrednost skew dijagrama.

Minimalna vrednost iz iskrivljenog dijagrama ukazuje baš na ovaj region:

Slika 1.11: Prikaz kretanja.



Slika 1.12: Iskrivljeni dijagram genoma Genome = CATGGGCATCGGCCATACGCC.



Slika 1.14: Region na koji pokazuje minimalna vrednost iskrivljenog dijagrama Ešerihije koli.

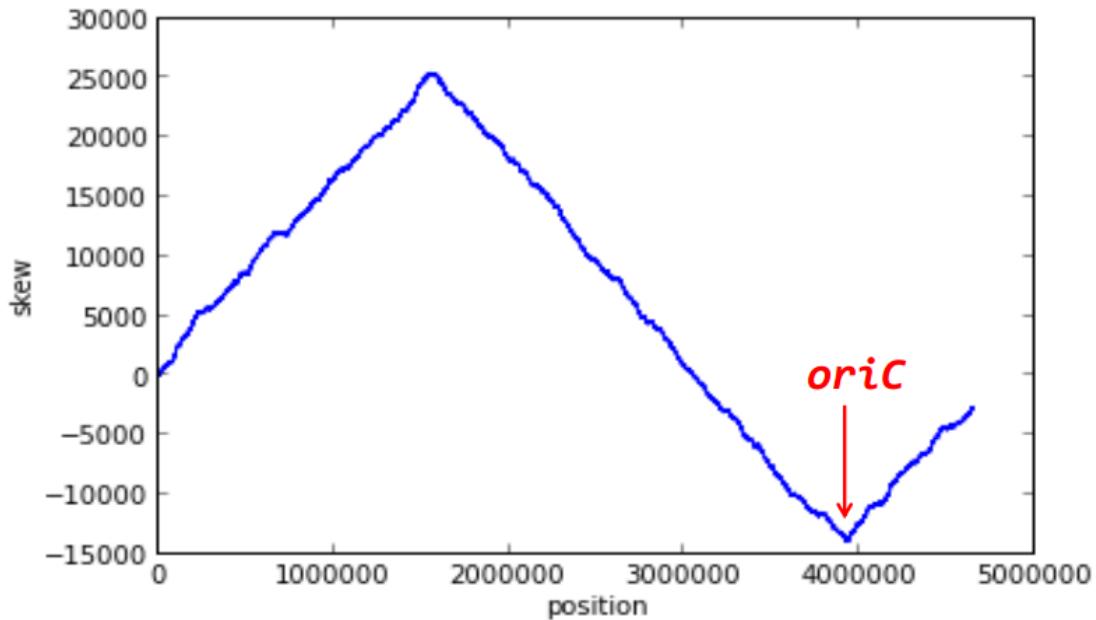
```

aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgataacgcggta
tggaaatggattgaagccggccgtggattctactcaactttgtcggcttgagaaagacc
ttggatcctgggtattaaaaagaagatctattatttagagatctgttctattgtatctc
ttattaggatcgcactgccctgtggataacaaggatccggcttttaagatcaacaacctgg
aaaggatcattaactgtgaatgatcggtatcctggaccgtataagctggatcagaatga
gggttatacacaactaaaaactgaacaacagttttttggataactaccgggtatc
caagcttcctgacagagttatccacagttagatcgcacgtatctgtataacttatttgagtaaa
ttaaccacgatcccagccattctctgccggatcttccggaatgtcgatcaagaatgt
tgatcttcagtg

```

Uočimo da u ovom regionu nema čestih 9-grama (koji se pojavljuju 3 ili više puta). Iz toga zaključujemo da, iako smo uspeli da nađemo *oriC* bakterije Ešerihija koli, nismo uspeli da nađemo

Slika 1.13: Iskrivljeni dijagram Ešerihije koli.



DNKA boksove. Međutim, pre nego što odustanemo od potrage, osmotrimo još jednom *oriC* Vibrio cholerae, kako bismo pokušali da nađemo način da izmenimo naš algoritam i uspemo da lociramo DNKA boksove u Ešerihiji koli. Veoma brzo, može se uvideti da osim tri pojavljivanja ATGATCAAG i tri pojavljivanja CTTGATCAT, *oriC* Vibrio cholerae sadrži i dodatna pojavljivanja ATGATCACAC i CATGATCAT koji se razlikuju samo u jednom nukleotidu od gornjih niski. Ovo još više povećava šanse da smo naišli na prave DNKA boksove, a ima i biološkog smisla. Naime, DNKA se može vezati i za nesavršene DNKA boksove, one koji se razlikuju u nekoliko nukleotida.

Slika 1.15: Prikaz pojavljivanja nesavršenih niski nukleotida.

```
atcaATGATCAACgttaagcttctaaggATGATCAAGgtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtcggttatctccttcctctcgtaactctcatgacca
cgaaaaagATGATCAAGagaggatgattcttgccatattcgaaataacttgtgactt
gtgcttccaatttgacatcttcagcgccatattgcgtggccaagggtgacggagcgggatt
acgaaaagCATGATCATgctgttctgttatctgtttgactgagacttgttagga
tagacggttttcatcactgacttagccaaagccttactctgcctgacatcgaccgtaat
tgataatgaatttacatgcttccgcgacgattacctCTTGATCATcgtccgattgaag
atcttcaattgttaattctttcgactcatagccatgatgagctCTTGATCATgtt
tccttaaccctctattttacggaagaATGATCAAGctgctgctCTTGATCATcgttcc
```

Cilj nam je da sada izmenimo algoritam čestih reči (*FrequentWords*) tako da možemo da pronađemo DNKA boksove koji su predstavljeni čestim k-gramima, sa mogućim izmenama na pojedinim nukleotidima. Ovaj problem nazvaćemo problem čestih reči sa propustima.

Problem čestih reči sa propustima. Pronaći najčešće k-grame sa propustima u niski karaktera.

Ulaz: Niska Text i celi brojevi k i d.

Izlaz: Svi najčešći k-grami sa najviše d propusta u niski Text.

Pokušajmo, još jednom, sa pronalaskom DNKA boksova kod Ešerihije koli, tako što ćemo naći najčešće 9-grame sa propustima i komplementima u regionu *oriC* koji nam je predložen minimalnom vrednošću iskrivljenog dijagrama. Pokušaćemo sa malim prozorom koji ili počinje ili se završava ili je centriran na poziciji najmanje iskrivljenosti. Ovakvim izvođenjem pronalazimo TTATCCACA/TGTGGATAA kao najčešći 9-gram. Međutim, ovo nije jedini 9-gram. Za ostale 9-grame još uvek ne znamo čemu služe, ali znamo da nose skrivene informacije, da se grupišu unutar genoma i da većina njih nema veze sa replikacijom.

Slika 1.16: Prikaz pronađenih niski sa propustima i komplementima u *oriC* regionu Ešerihije koli.

```

aatgatgatgacgtcaaaaggatccggataaaaacatggtgattgcctcgccataacgcgg
tataaaaatggattgaagcccgggcccgtggattctactcaacttgcggcttgagaaa
gacctggatcctgggtattaaaaagaagatctatttttagagatctgttctattgt
gatctcttatttaggatcgcactgccTGTGGATAAcaaggatccggcttttaagatcaa
caacctggaaaggatcattaactgtgaatgatcggtgatcctggaccgtataagctggg
atcagaatgaggggTTATACACAactcaaaaactgaacaacagttgttcTTTGGATAAC
taccgggttcatccaagcttctgacagagTTATCCACAgtagatcgcacgatctgtata
cttatttgagtaaattAACCCACGATCCCAGCCATTCTGCCGGATCTCCGGATG
tcgtgatcaagaatgttcatcttcagtg

```

1.3 Zadaci sa vežbi

U nastavku će biti predstavljeni zadaci sa vežbi na kursu rađeni u programskom jeziku Python.

1.3.1 FrequentWords

```

1 #frequent_words
2 def pattern_count(text, pattern):
3     count = 0
4     k = len(pattern)
5     for i in range(len(text) - k):
6         if text[i:i+k] == pattern:
7             count += 1
8     return count
9
10 def frequent_words(text, k, min_count):
11     frequent_patterns = set([])
12     count = []
13     n = len(text)-k
14     for i in range(n):
15         # Izvuci podnisku koji pocinje na $i$-toj poziciji i ima $k$ karaktera
16         pattern = text[i:i+k]
17         count.append(pattern_count(text, pattern))
18     max_count = max(count)
19     if max_count < min_count:
20         return []
21     for i in range(n):
22         if count[i] == max_count:
23             frequent_patterns.add(text[i:i+k])
24     return frequent_patterns
25
26 def main():
27     print(frequent_words('agcttagatgcttagcttagctgatcgagctgatgcaggcagtgtac', 4,
28                           2))
29
30 if __name__ == "__main__":
31     main()

```

1.3.2 Faster FrequentWords

```

1 #faster frequent_words
2 def pattern_to_number(pattern):
3     if len(pattern) == 0:
4         return 0
5     last = pattern[-1]
6     prefix = pattern[:-1]
7     return 4 * pattern_to_number(prefix) + symbol_to_number(last)
8
9 def number_to_pattern(n, k):
10    if k == 1:
11        return number_to_symbol(n)
12    prefixIndex = n // 4 #celobrojno deljenje
13    r = n % 4

```

```

14     symbol = number_to_symbol(r)
15     prefix = number_to_pattern(prefixIndex, k-1)
16     return prefix + symbol
17
18 def symbol_to_number(c):
19     pairs = {
20         'a' : 0,
21         't' : 1,
22         'c' : 2,
23         'g' : 3
24     }
25     return pairs[c]
26
27 def number_to_symbol(n):
28     pairs = {
29         0 : 'a',
30         1 : 't',
31         2 : 'c',
32         3 : 'g'
33     }
34     return pairs[n]
35
36 def pattern_count(text, pattern):
37     count = 0
38     k = len(pattern)
39     for i in range(len(text) - k):
40         if text[i:i+k] == pattern:
41             count += 1
42     return count
43
44 def computing_frequencies(text, k):
45     frequency_array = [0 for i in range(4**k)] #4 ^ k
46     for i in range(len(text) - k):
47         pattern = text[i:i+k]
48         j = pattern_to_number(pattern)
49         frequency_array[j] += 1
50     return frequency_array
51
52 def faster_frequent_words(text, k, min_count):
53     frequent_patterns = set([])
54     frequency_array = computing_frequencies(text, k)
55     max_count = max(frequency_array)
56     if max_count < min_count:
57         return []
58     for i in range(4**k):
59         if frequency_array[i] == max_count:
60             pattern = number_to_pattern(i, k)
61             frequent_patterns.add(pattern)
62     return frequent_patterns
63
64 def main():
65     print(faster_frequent_words('agcttagatgcttagctgatcgagctgatgcaggcagtgtacg
→ ', 4, 2))

```

```

66     #print(number_to_pattern(pattern_to_number('ta'),2))
67
68 if __name__ == "__main__":
69     main()

```

1.3.3 Skew Diagram

```

1 #GC-skew
2 import matplotlib.pyplot as plt
3
4 def draw_skew(skew):
5     x = [i for i in range(len(skew))]
6     ax = plt.subplot()
7     ax.plot(x, skew)
8     plt.show()
9
10 def calculate_skew(text):
11     skew = [0 for c in text]
12     last = 0
13     for i in range(0, len(text)):
14         if text[i] == 'g':
15             skew[i] = last + 1
16         elif text[i] == 'c':
17             skew[i] = last - 1
18         else:
19             skew[i] = last
20         last = skew[i]
21     return skew
22
23
24 def main():
25     text = "catgggcatcgccatacgcc"
26     print(calculate_skew(text))
27     draw_skew(calculate_skew(text))
28
29 if __name__ == "__main__":
30     main()

```

1.3.4 Frequent Words With Mismatches

```

1 # Prevodjenje nukleotida u brojeve
2 def symbol_to_number(c):
3     pairs = {
4         'a' : 0,
5         't' : 1,
6         'c' : 2,
7         'g' : 3
8     }
9
10    return pairs[c]
11
12 # Prevodjenje nukleotida u brojeve
13 def number_to_symbol(n):
14     pairs = {

```

```

15     0 : 'a',
16     1 : 't',
17     2 : 'c',
18     3 : 'g'
19 }
20
21     return pairs[n]
22
23 # Prevodjenje broja u odgovarajucu nukleotidnu sekvencu
24 def number_to_pattern(n, k):
25     if k == 1:
26         return number_to_symbol(n)
27
28     prefix_index = n // 4
29     r = n % 4
30     c = number_to_symbol(r)
31     prefix_pattern = number_to_pattern(prefix_index, k - 1)
32
33     return prefix_pattern + c
34
35
36
37 # Prevodjenje nukleotidne sekvence u odgovarajuci broj
38 def pattern_to_number(pattern):
39     if len(pattern) == 0:
40         return 0
41
42     last = pattern[-1:]
43     prefix = pattern[:-1]
44
45     return 4 * pattern_to_number(prefix) + symbol_to_number(last)
46
47
48
49 # Hamingova distanca, broj pozicija karaktera na kojima se tekstovi 1 i 2
50 # razlikuju,
51 # podrazumeva se da je duzina obe niske jednaka
52 def hamming_distance(text1, text2):
53     distance = 0
54
55     for i in range(len(text1)):
56         if text1[i] != text2[i]:
57             distance += 1
58
59     return distance
60
61
62 # Brojanje pojavljivanja podsekvenci u tekstu koje se od uzorka razlikuju na
63 # najvise d pozicija
64 def approximate_pattern_count(text, pattern, d):
65     count = 0

```

```

66     for i in range(len(text) - len(pattern)):
67         pattern_p = text[i:i+len(pattern)]
68
69         if hamming_distance(pattern, pattern_p) <= d:
70             count += 1
71
72     return count
73
74
75 # Pronalazenje svih niski susednih zadatom uzorku sa razlikama na najvise d
76 # pozicija
77 def neighbors(pattern, d):
78     if d == 0: # Ako je dozvoljena greska jednaka nuli onda je samo uzorak svoj
79     # sused bez gresaka
80     return set([pattern])
81
82     # Izlaz iz rekurzije:
83     if len(pattern) == 1: # Ako je duzina uzorka jednaka 1, a dozvoljena greska
84     # je veca od nule, onda je moguce iskoristiti bilo koji karakter
85     return set(['a', 't', 'c', 'g'])
86
87     neighborhood = set([])
88
89     suffix_neighbors = neighbors(pattern[1:], d) # Pronalaze se svi susedi
90     # duzine n-1
91
92     for text in suffix_neighbors:
93         if hamming_distance(pattern[1:], text) < d: # Ako se sused razlikuje na
94         # manje od d pozicija od podnische uzorka bez prvog karaktera
95         for x in ['a', 't', 'c', 'g']:
96             neighborhood.add(x + text) # Moguce je dodati bilo koji
97             # karakter na pocetak suseda i time dobiti najvise d razlika
98             else: # U suprotnom, sused se vec razlikuje na d pozicija od uzorka pa
99             # je dozvoljeno samo dodavanje ispravnog karaktera kako se
100            neighborhood.add(pattern[0] + text) # razlika ne bi povecala preko
101            d
102
103     return neighborhood
104
105
106
107 def frequent_words_with_mismatches(text, k, d):
108     frequent_patterns = set([])
109
110     close = [0 for i in range(4**k)] # Kandidati za proveru
111     frequency_array = [0 for i in range(4**k)]
112
113     # Za svaki uzorak duzine k u tekstu evidentiraju se kandidat susedi cija
114     # pojavljivanja treba uzeti u razmatranje (niske koje se razlikuju od
115     # uzorka na najvise d pozicija)
116     for i in range(len(text) - k):
117         neighborhood = neighbors(text[i:i+k], d) # Pronalaze se kandidati

```

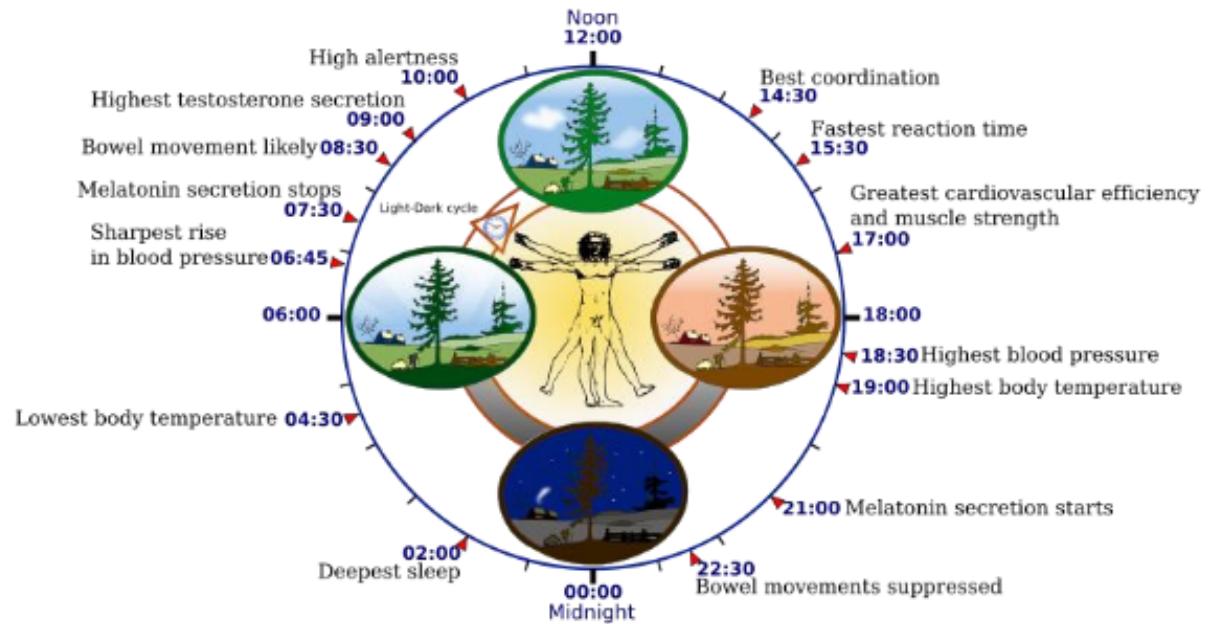

Glava 2

Koji DNK šabloni igraju ulogu molekularnog sata?

2.1 Biološki problem

Bioritam svih živih bića kotorišće "unutrašnji časovnik" koji još zovemo i cirkadijalni. Ljudi koji često putuju avionom na drugi kraj sveta mogu to da osećaju kada pokušaju da zaspnu nakon promene nekoliko vremenskih zona. Kao i svaki sat, i ovaj može da se pokvari, što rezultuje genetskom bolešću pod nazivom sindrom odložene faze spavanja. Njegova osnova je na molekularnom nivou. Mnogi procesi su kontrolisani ovim časovnikom, što ilustruje slika 2.1. Kao što vidimo, postoji tačno određeno vreme u toku dana kada telo ima najnižu temperaturu, kada kreće i prestaje lučenje hormona poput melatoninina (neophodnog za kvalitetan san) i slično.

Slika 2.1: Cirkadijalni ritam



Naučnici su se pitali kako ćelije znaju kada treba da uspore ili ubrzaju proizvodnju određenih proteina. Ranih sedamdesetih, Ron Konopka i Sejmor Benzer su napravili prve korake ka

rešavanju ove misterije. Do danas je otkriveno mnogo cirkadijalnih gena koji koordiniraju ponašanje stotine drugih gena.

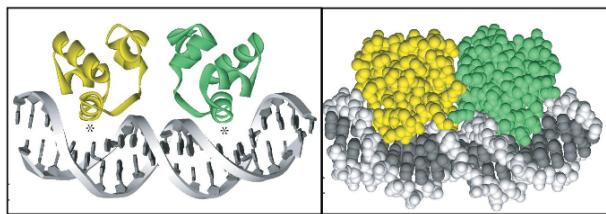
Kod čoveka, cirkadijali ritam je promenljiv, tj. varira od osobe do osobe. Mi ćemo se u daljem tekstu fokusirati na biljke, jer je kod njih cirkadijali ritam pitanje života i smrti, stoga ne sme biti promenljiv. Geni biljaka moraju znati kada sunce izlazi i zalazi kako bi znali kada treba vršiti fotosintezu, jer je ona od krucijalne važnosti za život biljke, a usko povezana sa količinom sunčeve svetlosti. Primeri specifičnih ponašanja nekih biljaka u zavisnosti od cirkadijalnog ritma dati su na slici 2.2.

Slika 2.2: Cirkadijali ritam biljke

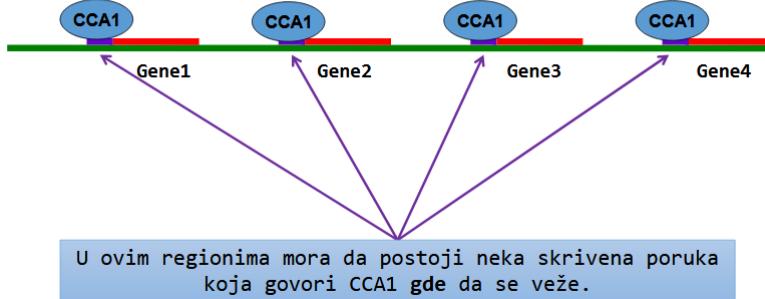


Ispostavlja se da svaka ćelija biljke čuva podatak o tome da li je dan ili noć nezavisno od drugih ćelija, kao i da su samo tri gena odgovorna za upravljanje satom. Oni kodiraju regulatorne proteine (transkripcione faktore)- to su LCY, CCA1 i TOC1. Spoljašnji faktori, kao što je količina sunčeve svetlosti, kontrolisu regulatorne gene i regulatorne proteine kako bi organizmi prilagodili svoju gensku ekspresiju, odnosno da li će se protein sintetisati u to vreme ili ne. Dakle, svaki metabolički proces je regulisan cirkadijalnim časovnikom kojim upravljaju regulatorni proteini, odlučujući kada će se koji protein u ćeliji biljke sintetisati. Regulatorni proteini upravljaju genskom ekspresijom tako što se vežu za DNA u trenucima kada je potrebno sintetisati neki protein važan za cirkadijali ritam, kao što je prikazano na slici 2.3. Naravno, to vezivanje se ne može ostvariti bilo gde u okviru DNA, već postoje posebna mesta, kao što je prikazano na slici 2.4. Ta posebnost je neophodna iz više razloga. Regulatorni protein mora znati gde treba da se veže, a isto tako DNA mora znati kako da to protumači, odnosno, to za nju mora biti signal da je vreme započeti sintezu, kao i pružiti informaciju o kom se proteinu radi. Ta posebna mesta predstavljaju manje skupove nukleotida u DNA koji nazivamo **regulatorni motivi**. Naš zadatak je da ih pronađemo.

Slika 2.3: Regulatorni proteini na delu



Slika 2.4: Mesta vezivanja regulatornih proteina



2.2 Informatički problem

Kako bismo informatički rešili problem pronalaženja regulatornih motiva u DNK, moramo predstaviti sve gorepomenute biološke pojmove na način koji bi informatičar i njegov računar mogli razumeti i obraditi.

- Iz ove tačke gledišta, DNK predstavlja nisku karaktera nad azbukom: A, G, T, C.
- Kodirajuće sekcije DNK (one koje se prepisuju i prevode u proteine) za nas će biti podniske niske DNK.
- Regulatorni motiv je šablon koji se pojavljuje tačno jednom u svakoj kodirajućoj sekciji DNK.

Hajde da damo u računarskom smislu korektnu definiciju problema:

Informatička definicija problema nalaženja regulatornih motiva u DNK

Ulaz: N niski koje predstavljaju kodirajuće sekvene DNK.

Izlaz: Podniska dužine k koja predstavlja regulatorni motiv (skrivenu poruku, mesto vezivanja regulatornih proteina).

Ovaj problem nas podseća na problem pronalaženja *OriC-a*, koji smo rešavali svodeći ga na pronalaženje čestih reči u tekstu koristeći algoritam *FrequentWords*, detaljno opisan u prethodnom poglavljju. Da bismo taj algoritam primenili ovde, neophodno je da od niza niski dobijemo konkatenacijom jednu veliku nisku, na koju ćemo primeniti ovaj algoritam.

Međutim, ovaj algoritam nije primenljiv ako motivi mutiraju. Možemo pokušati da primenimo algoritam *FrequentWordsWithMissmatches*. To bi nas dovelo do tačnog rešenja, ali nakon previše vremena. Naime, kada smo nalazili *OriC*, tražili smo uzorke dužine 9 karaktera (DnaA box je bio te dužine). U ovom slučaju, tražimo motive koji su najčešće dužine 15 karaktera, pa nam ovaj algoritam ne radi dovoljno brzo. Dakle, potrebna nam je nova ideja.

2.3 Problem ubačenog motiva

I dalje pokušavamo da pronađemo skrivenu poruku u DNK koja kodira mesto vezivanja regulatornih proteina za DNK. Dali smo korektnu definiciju problema i, kao pravi matematičari, pokušali najpre da svedemo problem na neki drugi, koji smo već rešili. Međutim, to nas je

odvelo u veliku vremensku složenost, pa samim tim i neupotrebljivost pronađenog algoritma za rešavanje. Zaključili smo da nam je potreban novi pristup, tj. da moramo probati da problem rešimo direktno - bez pomoći algoritama za *OriC*. Za ovako nešto, neophodno je najpre definisati još nekoliko pojmove vezanih za šablove koje pokušavamo da pronađemo:

- Mutirani šablon predstavlja šablon u kome se na nekim mestima može pojaviti mutacija, odnosno odstupanje od početne niske.
- (k, d) motiv je k -gram koji se pojavljuje u svakoj sekvenci sa najviše d razlika.
- Kanonski motiv predstavlja motiv koji tražimo (bez uticaja mutacija).
- Instance su mutirani motivi - oni koji se pojavljuju u niskama sa najviše d grešaka, odnosno razlika u odnosu na kanonski motiv.

Sada, kada smo tačno i precizno definisali pojmove koji će nam igrati uloge regulatornih motiva, izmenićemo i samu definiciju problema, tj. malo promeniti scenario tako da se sve uloge lepo uklope, pazeći naravno da ne izgubimo na tačnosti definicije.

Problem ubačenog motiva: Pronalaženje (k, d) motiva u skupu niski

Ulaz: Skup niski Dna i celi brojevi k (dužina motiva) i d (maksimalni broj razlika).

Izlaz: Svi (k, d) motivi u skupu Dna .

Nova definicija problema izrodiće nekoliko novih ideja za njegovo rešavanje, a te ideje izrodiće nove algoritme. U daljem tekstu prikazaćemo nekoliko rešenja, uz osvrt na ideju koja nas je do njega dovela.

2.3.1 Enumeracija motiva

Početna ideja je zasnovana na gruboj sili - za svaki k -gram ćemo ispitati da li je (k, d) motiv za dati skup niski. Dakle, trebalo bi generisati 4^k kombinacija i za svaku ispitati da li je (k, d) motiv. Postavlja se pitanje da li je potrebno ispitati svih 4^k kandidata.

Pogledajmo na primer sledeći skup niski:

AAATTTAAATTAAATTAA
TTTAAATTAAATTAAA

Da li ima smisla proveravati 8-gram GGAAGGAA? Naravno da nema! Karakter G se ne pojavljuje ni u jednoj niski iz našeg skupa. Zato sigurno ne može biti traženi (k, d) motiv. Dakle, prvo možemo izbaciti kandidate koji se ne pojavljuju ni u jednoj niski iz skupa Dna . Takođe, možemo primetiti da je skup instanci jednog kandidata koje se pojavljuju u ostalim niskama (ceo skup Dna , bez niske u kojoj je dati kandidat podniska) prilično ograničen. Ograničenje nameće broj d . Dakle, jasno je da treba proveravati samo one instance koje se od kandidata razlikuju na najviše d pozicija. Ovako redukovana pravila potrage iznedriće algoritam koji se naziva *MotifEnumeration*, koji bismo u obliku pseudokoda predstavili na sledeći način:

```

1 MotifEnumeration(Dna, k, d)
2   for each k-mer a in Dna
3     generate all possible k-mers a1 differing from a by at most d mutations
4     for each such k-mer a1
5       if a1 is a (k, d)-mer in each sequence in Dna
6         output a1

```

Ovaj algoritam je suviše spor kada su k i/ili d veliki brojevi. Hajde da malo analiziramo složenost ovog algoritma. Možemo videti da najveću (i jedinu) komponentu složenosti čini dvostruka forpetlja, koja zavisi od broja kandidata koje pretražujemo i broja njihovih instanci. Kada su k i/ili d veliki brojevi, instanci ima mnogo, i pretraga ide sporo. Zbog toga, u praksi, ovaj algoritam ne pokazuje dobre rezultate za velike vrednosti k i/ili d . Dakle, potrebna nam je nova ideja.

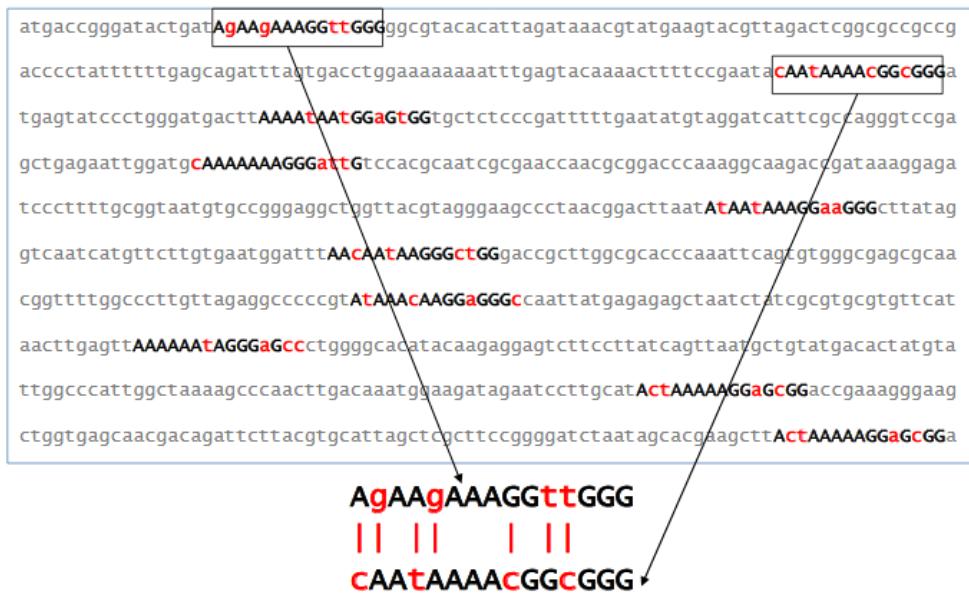
2.3.2 Najsličniji k-grami u parovima niski

Videli smo da pretraga grubom silom ima veliku složenost, usled (iako redukovanih ali i dalje ogromnog) broja kandidata i njihovih instanci. Pokušali smo da redukujemo broj provera i nismo postigli mnogo. To nas dovodi do razmišljanja da ključ rešenja optimalnog algoritma ne leži u broju, već možda u redosledu. Hajde da konstruišemo ovakav primer:

```
atcgtcagAAATTAAAGGGtgtcaactg
ggatcaagctAAATCTAAAGGGcttcag
gatctacccaAAAActTAAGGGgtaaac
```

Velikim slovima predstavljen je (12,1)-motiv koji tražimo. Grubom silom pretraga bi počela na samom početku prve niske. Kako naš motiv počinje na 9. poziciji prve niske, vidimo da bi 8 puta naleteli na čorsokak. To znači da bi se 8 velikih iteracija izvršilo uzalud. Ako bismo nekako mogli početi od devete pozicije, uštedeli bismo vreme. Ali, kako da znamo koja je tajna pozicija od koje treba krenuti? Primećujemo da se podniske atcgtcag i ggatcaag prilično razlikuju (7/8 pozicija razlike). Dakle, to sigurno ne može biti sastavni deo neke instance. Međutim, podniske AAATTAAAGGG i AAATCTAAAGGG su prilično slične (1/12 pozicija razlike). To je svakako dobar kandidat. Ako bismo u prve dve niske pronašli kandidate sa dosta sličnosti, šanse da smo odmah pogodili (k, d) motiv bi bile velike. Sve i da nismo, ovim putem ćemo obilaziti prvo verovatnije kandidate, pa bi ukupan broj pretraga sa velikom verovatnoćom bio dosta manji. Dakle, možemo probati da poredimo parove niski iz *Dna*, da uočimo dva najsličnija k-grama u dve niske iz *Dna*, od njih napravimo kanonski, i za njega proveravamo da li je (k, d) motiv. Malo veći primer možete videti na slici 2.5 i probati sami da nađete ubaćeni motiv.

Slika 2.5: Najsličniji k-grami u parovima niski - primer



Zvuči predobro da bi bilo istinito? Pa možda ste i u pravu. Sličnost para niski je takođe određena brojem d . One se od kanonskog motiva mogu razlikovati na d pozicija, što znači da se među sobom, mogu razlikovati na $2d$ pozicija. Na malom izolovanom primeru, gde je bilo $d = 1$, zaista su slične niske isplivale kao dobri kandidati. Međutim, u praksi, sa (15,4)-motivima, broj dozvoljenih razlika je malo veći, čak 8. Kako izgledaju takvi parovi, prikazano je na slici 2.6, gde se jasno vidi koliko je to zaista mnogo i kako ove dve niske više i nisu tako slične.

Slika 2.6: Najsličniji k-grami u parovima niski - problem



Primer koji pokazuje koliko je ovo zaista loše rešenje je eksperiment rađen nad 10 slučajno generisanih niski iz *Dna* dužine 600, sa ubačenim (15, 4) motivom. Pristupom pronalaženja parova niski iz *Dna*, pronađeno je nekoliko hiljada parova k-grama koji su se razlikovali na manje od 8 pozicija. Ovo sigurno nisu skrivene poruke, jer ih je previše.

Da bismo ih pretražili u nekom smislenom redosledu, potrebno nam je da ih nekako rangiramo. Dakle, potreban nam je novi pristup.

2.3.3 Matrice motiva

Dosadašnje pretrage skupa *Dna*, nisu dale dobre rezultate. Pokušavali smo da ubrzamo pretragu na razne načine, ali bez puno uspeha. Možda je vreme da probamo da potpuno promenimo pristup, da mislimo izvan kutije. Prednost nas informatičara je u tome što ne mislimo uvek linerano. Nekada je ključ u rešenju, a ne u početnim premisama. Zbog toga, pokušajmo da analiziramo jedan skup rešenja, i da malim koracima unazad dođemo do početnih uslova.

Pretpostavićemo da imamo neku kolekciju motiva i stavimo ih u matricu. Recimo da smo uradili jednom grubu pretragu i dobili neke rezultate. Da bismo olakšali komunikaciju, uvešćemo par pojmove koje ćemo koristiti u daljem tekstu.

- Najpopularniji nukleotid u nekoj koloni matrice je onaj koji se pojavljuje najveći broj puta.
- Konsenzus niska predstavlja nisku koja se dobija nadovezivanjem najpopularnijih nukleotida iz svake kolone.
- Skor predstavlja broj nepopularnih simbola (onih koji nisu najpopularniji u svojoj koloni) u matrici.

Za lakše usvajanje, ilustrovani primer ovih pojmove prikazan je na slici 2.7. Najpopularniji nukleotidi u svakoj koloni matrice motiva su napisani velikim slovima i boldovani. Oni su nadovezani u konsenzus nisku. Nepopularni simboli su napisani malim slovima, i njihov ukupan broj predstavlja skor.

Slika 2.7: Matrica motiva, konsenzus niska i skor

Motifs	T	C	G	G	G	G	g	T	T	T	t	t
	c	C	G	G	t	G	A	c	T	T	a	C
	a	C	G	G	G	G	A	T	T	T	t	C
	T	t	G	G	G	G	A	c	T	T	t	t
	a	a	G	G	G	G	A	c	T	T	C	C
	T	t	G	G	G	G	A	c	T	T	C	C
	T	C	G	G	G	G	A	T	T	c	a	t
	T	C	G	G	G	G	A	T	T	c	C	t
	T	a	G	G	G	G	A	a	c	T	a	C
	T	C	G	G	G	t	A	T	a	a	C	C

Consensus(Motifs) T C G G G G A T T T C C

Score(Motifs) 3+ 4+ 0+ 0+ 1+ 1+ 1+ 5+ 2+ 3+ 6+ 4= 30

Analizirajmo malo našu matricu rešenja. Ako bismo pretpostavili da su ovo instance nekog (k, d) motiva, jasno, želeli bismo da one budu što sličnije, tj. da se razlikuju na što manje pozicija. Dakle, želimo da nam broj nepopularnih simbola bude što manji, odnosno želimo mali skor. Dakle, ako ovako postavimo problem, skor je zapravo "mera neuspjeha", pa je ključ dobrog rešenja u minimizovanju ove veličine.

Novi pojmovi, novi pristup i nova ideja, zahtevaju i malo drugačiju definiciju problema:

Problem pronalaženja motiva: Za dati skup niski iz *Dna* naći skup k-grama (po jedan iz svake niske) sa minimalnim skorom među svim mogućim k-gramima iz datog skupa niski.

Ulaz: skup niski *Dna* i ceo broj *k*.

Izlaz: skup k-grama *Motifs*, po jedan iz svake niske *Dna*, tako da je vrednost skora matrice *Motifs* minimalna.

Kada smo definisali problem, vreme je da krenemo da ga rešavamo. Prvo što nam svakako pada na pamet je gruba sila. To je obično najjednostavniji algoritam, sa ne tako dobrom brzinom. Ali, ako on da zadovoljavajuću brzinu, nema potrebe komplikovati dalje, zar ne?

Dakle, krećemo redom. Uzimamo po jedan k-gram iz svake niske skupa *Dna* i računamo skor. Tako redom obiđemo sve kombinacije k-grama, pamteći minimalni skor, i na kraju, proglašimo rešenjem onaj skup motiva sa najmanjim skorom.

Rešenje je dobro, pa hajde da pogledamo složenost. Neka je *t* broj niski i *n* dužina svake niske. Zaključujemo da je $(n - k + 1)$ broj k-grama jedne niske, odnosno, $(n - k + 1)^t$ broj kombinacija koje treba da proverimo. Vreme provere je konstantno (treba samo izračunati skor i ažurirati neku vrednost). Dakle, ukupna složenost je: $(n - k + 1)^t$. Eksponencijalna složenost svakako nije dobra, tako da zaključujemo da je ovaj algoritam prespor. Moramo ga nekako unaprediti.

Vratimo se na našu matricu motiva i pokušajmo da primetimo nešto što bi nam pomoglo da malo ubrzamo stvari. Pogledajmo primer dat na slici 2.8.

Slika 2.8: Matrica motiva, konsenzus niska i skor - detaljnije

<i>Motifs</i>	T	C	G	G	G	G	g	T	T	T	t	t	3
	c	c	G	G	t	G	A	c	T	T	a	C	4
	a	C	G	G	G	G	A	T	T	T	t	C	2
	T	t	G	G	G	G	A	c	T	T	t	t	4
	a	a	G	G	G	G	A	c	T	T	C	C	3
	T	t	G	G	G	G	A	c	T	T	C	C	2
	T	C	G	G	G	G	A	T	T	c	a	t	3
	T	C	G	G	G	G	A	T	T	c	C	t	2
	T	a	G	G	G	G	A	a	c	T	a	C	4
	T	C	G	G	G	t	A	T	a	a	C	C	3

$=30$

Consensus(Motifs) T C G G G G A T T T T C C

Score(Motifs) 3+ 4+ 0+ 0+ 1+ 1+ 1+ 5+ 2+ 3+ 6+ 4=30

Izračunali smo skor po vrstama i primetili da je isti kao skor po kolonama. Da li je to slučajnost? Ispostavlja se da nije. Suština veličine skor leži u tome da broji nepopularne simbole. Nije važno kako ih prebrajamo, ako su svi na broju. Pojam brojanja definisaćemo na malo formalniji način. Definisaćemo veličinu koja se zove Hamingonovo rastojanje.

Hamingovo rastojanje između dve niske istih dužina predstavlja broj pozicija na kojima se one razlikuju. Npr. Hamingonovo rastojanje niski AACCTTGG i ATCCATGG je 2:

AACCTTGG

ATCCATGG

-1-1— = 2

Kada imamo Hamingovo rastojanje, možemo reći da je skor po kolonama suma Hamingovih rastojanja k-grama iz matrice od konsenzus niske.

Vreme je da pokušamo da unapredimo naš algoritam. Želimo nekako da nađemo način da uradimo suprotno - da od konsenzus niske i niza *Dna* dobijemo matricu *Motifs*. Koristićemo sledeće osobine koje smo primetili na matrici *Motifs*:

1. Kako skor predstavlja sumu rastojanja od k-grama iz matrice do konsenzus niske, minimalni skor predstavlja minimizovanu sumu tih rastojanja.
2. Kako je skor po kolonama i vrstama isti, možemo rastojanje računati i po vrstama. Tako dolazimo do reformulacije našeg problema, tj. njegovog ekvivalenta koji koristi naša nova zapažanja.

Iskoristićemo ove osobine kako bismo još jednom predefinisali naš problem tako da odražava sliku iz novog pristupa.

Problem pronalaženja motiva - reformulacija: Naći k-gram *Pattern* i skup k-grama *Motifs* iz skupa niski *Dna* koji minimizuju rastojanje između svih mogućim k-grama *Pattern* i svih mogućih skupova k-grama *Motifs*.

Ulaz: skup niski iz *Dna* i ceo broj *k*.

Izlaz: k-gram *Pattern* i skup k-grama *Motifs* iz skupa niski *Dna* koji minimizuju $d(\text{Pattern}, \text{Motifs})$.

Ovo je ekvivalentno našem prethodnom problemu jer smo primetili da $d(Pattern, Motifs)$ predstavlja skor ukoliko je *Pattern* konsenzus niska. Postavlja se pitanje da li smo ovim otežali naš problem. Ispostaviće se da nismo, jer ne moramo ispitivati sve skupove k-grama *Motifs*, već je dovoljno da iskoristimo problem niske medijane koji ispituje sve k-grame *Pattern*.

2.3.4 Problem niske medijane

Prethodnom definicijom problema, nameće se zaključak da je sledeći prirodni korak proći kroz sve kombinacije skupa k-grama *Motifs* i k-grama *Pattern*, kako bismo našli kombinaciju sa najmanjim rastojanjem. Međutim, videli smo i da algoritmi zasnovani na gruboj sili, tj. iscrpnoj pretrazi kombinacija u potrazi za pravom, vode ka ogromnoj složenosti i lošoj praktičnoj primeni.

Zato ćemo ovaj put preskočiti taj korak i odmah krenuti da modifikujemo algoritam tako da dobijemo manju složenost. Primetićemo da imamo dve dimenzije pretrage (računarski bi to bile dve ugnježdene for petlje). Jedna je po svim k-gramima *Pattern* a druga po skupovima *Motifs*. Jasno je da bi se složenost prepolovila (tj. korenovala) ako bismo imali samo jednu dimenziju. Zvuči neverovatno, ali je moguće, ako primetimo da nam unutrašnja for petlja nije potrebna, jer možemo uzeti cele niske umesto motiva. Tačnije, želimo da računamo rastojanje jednog k-grama od celog skupa *Dna*. Računanjem rastojanja od samih niski iz *Dna* umesto od njihovim podniski dužine k , štedimo dosta vremena, a postižemo isti efekat. Međutim, da bismo to učinili, najpre moramo definisati nekoliko pojmoveva, na prvom mestu jer još uvek nismo rekli kako se računa rastojanje sem ako imamo dve niske i to iste dužine. Dakle, definisaćemo najpre sledeće pojmove:

- Hamingovo rastojanje između dve niske različitih dužina predstavlja minimum Hamingovih rastojanja između kraće niske i svih podniski duže niske odgovarajuće dužine.
- Definišemo rastojanje između k-grama i skupa (dužih) niski kao sumu rastojanja između tog k-grama i svih niski iz skupa.
- Niska medijana za skup niski *Dna* predstavlja onaj k-gram koji minimizuje rastojanje između tog k-grama i skupa *Dna* - $d(k\text{-}gram, Dna)$.

Novi pojmovi i nova ideja, prirodno vode i do nove definicije problema. Moramo dobro definisati problem koji rešavamo ako želimo da nas to dovede do dobrog algoritma za njegovo rešavanje.

Problem niske medijane: pronaći nisku medijanu.

Ulaz: skup niski *Dna*.

Izlaz: k-gram $k\text{-}mer$ koji minimizuje rastojanje $d(k\text{-}mer, Dna)$.

Sada imamo sav potreban alat da rešimo problem. Eliminišući jednu dimenziju, pretragu ćemo vršiti samo po svim mogućim kombinacijama k-grama *Pattern* (njih 4^k) i kao rezultat vratiti onaj k-gram koji ima najmanje rastojanje od skupa *Dna*. Ovaj algoritam se naziva *MedianString* i prikazan je sledećim pseudokodom:

```

1 MedianString(Dna, k)
2   best-k-mer ← AAA...AA
3   for each k-mer from AAA...AA to TTT...TT
4     if d(k-mer, Dna) < d(best-k-mer, Dna)
5       best-k-mer ← k-mer
6   return best-k-mer

```

Hajde da analiziramo složenost i vidimo da li smo zaista napredovali. Neka je t dužina niza *Dna* i n dužina niske iz *Dna*. Tada nam je za izračunavanje rastojanja između jednog k-grama i jedne niske iz skupa *Dna* potrebno $k \cdot (n - k + 1)$ poređenja. Kako imamo t niski, za

izračunavanje rastojanja k-grama od skupa potrebno nam je $t \cdot k \cdot (n - k + 1)$, odnosno približno $t \cdot k \cdot n$, jer je n daleko veće od k pa čini primarnu komponentu složenosti u $(n - k + 1)$. Ceo taj postupak moramo izvršiti za svaki k-gram, odnosno 4^k puta, pa imamo, dakle, da ukupna složenost algoritma iznosi: $4^k \cdot n \cdot t \cdot k$. Primećujemo da je ovo dobar napredak u odnosu na $n^t \cdot k \cdot t$, ali i dalje imamo eksponencijalnu složenost. Dakle, iako je *MedianString* algoritam mnogo brži od grube sile, za velike k je i dalje prespor.

2.3.5 Probabilistički pristup

Uopšteno o pristupu

Analizirali smo nekoliko determinističkih algoritama i videli da, iako uvek daju tačno rešenje, troše preveliku količinu vremena. Čak i najbolji među njima, *MedianString*, imao je eksponencijalnu složenost. To nas navodi na razmišljanje da je možda ponovo došlo vreme da uradimo veliki zaokret u pristupu samom problemu i pokušamo da nađemo kompromis između tačnosti i utroška vremena. Algoritmi koji ovo omogućavaju se nazivaju probabilistički algoritmi. Oni daju tačno rešenje sa određenom verovatnoćom, ali za dosta manje vremena, i oni su naša sledeća stanica u potrazi za ubaćenim motivima. Kao i svaki put kada menjamo pristup, upoznaćemo se najpre sa nekoliko osnovnih pojmovima:

- *Count* matrica neke matrice motiva prikazuje koliko se puta koji nukleotid ponavlja u svakoj koloni matrice motiva.
- Profilna matrica neke matrice motiva prikazuje učestalost pojavljivanja nukleotida u svakoj koloni matrice motiva.

Ilustrovani primer ovih pojmoveva prikazan je na slici 2.9.

Slika 2.9: *Count* i profilna matrica

	T	C	G	G	G	G	g	T	T	T	t	t
Motifs	c	C	G	G	t	G	A	c	T	T	a	C
	a	C	G	G	G	G	A	T	T	T	t	C
	T	t	G	G	G	G	A	c	T	T	t	t
	T	t	G	G	G	G	A	c	T	T	C	C
	T	C	G	G	G	G	A	c	T	T	C	C
	T	C	G	G	G	G	A	T	T	c	a	t
	T	C	G	G	G	G	A	T	T	c	C	t
	T	a	G	G	G	G	A	a	c	T	a	C
	T	C	G	G	G	T	A	T	a	a	C	C

<i>Count(Motifs)</i>	A:	2	2	0	0	0	0	9	1	1	1	3	0
	C:	1	6	0	0	0	0	0	4	1	2	4	6
	G:	0	0	10	10	9	9	1	0	0	0	0	0
	T:	7	2	0	0	1	1	0	5	8	7	3	4

<i>Profile(Motifs)</i>	A:	.2	.2	0	0	0	0	.9	.1	.1	.1	.3	0
	C:	.1	.6	0	0	0	0	0	.4	.1	.2	.4	.6
	G:	0	0	1	1	.9	.9	.1	0	0	0	0	0
	T:	.7	.2	0	0	.1	.1	0	.5	.8	.7	.3	.4

Osnovna ideja ovog pristupa može se ilustrovati bacanjem k četverostranih kockica sa otežanim stranama. Svaka kockica pretstavlja jednu poziciju u motivu dužine k , i na stranama ima simbole A, C, T i G. Strane kockica su otežane, tako da svaka strana pada sa onom verovatnoćom koja je zapisana u profilnoj matrici, u preseku kolone rednog broja pozicije u motivu i reda karaktera koji je na toj strani kockice. Na primer, ukoliko imamo profilnu matricu kao na slici 2.10, baćali bismo 6 četverostranih kockica, gde bi recimo druga kockica bila otežana tako da simbol A pada sa verovatnoćom $7/8$, simbol C sa verovatnoćom 0, simbol T sa verovatnoćom $1/8$ i simbol G sa verovatnoćom 0. Naravno da u praksi ne možemo imati verovatnoću nula, ali sa našim zamišljениm kockicama, to je legitimna situacija. Zašto ovo nije dobro i kako se to popravlja, diskutovaćemo malo kasnije. Za sada ćemo prihvati ovu metaforu takvu kakva jeste i baciti naše zamišljene kockice. Neka su redom pali karakteri: A, T, A, C, A i G. Verovatnoću podnische koju oni čine možemo izračunati na osnovu profilne matrice, pretpostavljajući da su bacanja nezavisni događaji. Dakle, pročitaćemo iz profilne matrice verovatnoću karaktera A na prvoj poziciji ($1/2$), verovatnoću karaktera T na drugoj ($1/8$), ..., i pomnožićemo sve te vrednosti. Proizvod verovatnoća iznosi 0.001602.

Slika 2.10: Računanje verovatnoće k-grama

Neka je data profilna matrica <i>Profile</i> :						
A	1/2	7/8	3/8	0	1/8	0
C	1/8	0	1/2	5/8	3/8	0
T	1/8	1/8	0	0	1/4	7/8
G	1/4	0	1/8	3/8	1/4	1/8

$\Pr(\text{ATACAG} \mid \text{Profile}) =$
 $\frac{1/2}{1/8} \times \frac{1/8}{1/8} \times \frac{3/8}{0} \times \frac{5/8}{1/8} \times$
 $\frac{1/8}{1/8} = 0.001602$



Ako se setimo kako smo definisali matricu motiva, konsenzus nisku i profilnu matricu, lako možemo zaključiti da je ova verovatnoća zapravo verovatnoća da je niska ATACAG konsenzus niska. Sada nam same kockice više nisu potrebne, već možemo uzeti realne k-grame iz niski skupa *Dna* i za njih računati. Na primer, za nisku CTATAAACCTTACAT iz nekog skupa *Dna*, možemo izračunati verovatnoće svih njenih podniski dužine 6 (rezultate možete videti na slici 2.11) i potražiti onaj sa najvećom verovatnoćom (u primeru sa slike 2.11 to je AACCT). On je svakako dobar kandidat za ubačeni motiv. U ovome leži sama suština svih probabilističkih algoritama za traženje ubačenih motiva.

U daljem tekstu, predstavićemo nekoliko probabilističkih algoritama za rešavanje problema ubačenih motiva. Opisaćemo same algoritme, ali i diskutovati o njihovim dobrim i lošim stranama.

Važna napomena: Kao što smo već napomenuli, cenu malog vremena ovi algoritmi plaćaju tačnošću. Dakle, rešenje može malo odstupati od traženog. Zbog toga, za postizanje najbolje tačnosti, probabilističke algoritme treba pokretati više puta i uzeti najbolji rezultat.

Algoritam Greedy motif search

Prvi algoritam koji ćemo predstaviti je pohlepni algoritam. Kao ulaz, on prima skup niski *Dna*, broj njegovih elemenata t i dužinu ubačenog šablonu k , a kao izlaz daje matricu ubačenih motiva *BestMotifs*. Suština algoritma leži u njegovom imenu: grabi k-mere iz jedne niske redom, na osnovu svakog od njih nađi najbolju matricu u koju se on uklapa, i uzmi onu koja ima najmanji skor. Na početku *BestMotifs* se inicijalizuje na t podniski dužine k , od kojih svaka predstavlja prvih k karakera jedne niske iz skupa *Dna*. Iterira se kroz prvu nisku skupa *Dna*, i iteracija ima onoliko koliko ima podniski dužine k u njoj. U svakoj iteraciji, računa se po jedna matrica motiva. Građenje te matrice je jedan vid pohlepe ovog algoritma. Grabimo već izračunate vrednosti, kako bismo što bolje odredili sledeće. Prvi red čini tekuća podniska iteracije

Slika 2.11: Primer: najverovatniji 6-gram

6-mer	Pr (6-mer Profile)	
CTATAAAACCTTACAT	$1/8 \times 1/8 \times 3/8 \times 0 \times 1/8 \times 0$	0
CTATAAAACCTTACAT	$1/2 \times 7/8 \times 0 \times 0 \times 1/8 \times 0$	0
CTATAAAACCTTACAT	$1/2 \times 1/8 \times 3/8 \times 0 \times 1/8 \times 0$	0
CTATAAAACCTTACAT	$1/8 \times 7/8 \times 3/8 \times 0 \times 3/8 \times 0$	0
CTATAAAACCTTACAT	$1/2 \times 7/8 \times 3/8 \times 5/8 \times 3/8 \times 7/8$.0336
CTATAAACCTTACAT	$1/2 \times 7/8 \times 1/2 \times 5/8 \times 1/4 \times 7/8$.0299
CTATAAAACCTTACAT	$1/2 \times 0 \times 1/2 \times 0 \times 1/4 \times 0$	0
CTATAAAACCTTACAT	$1/8 \times 0 \times 0 \times 0 \times 0 \times 1/8 \times 0$	0
CTATAAACCTTACAT	$1/8 \times 1/8 \times 0 \times 0 \times 3/8 \times 0$	0
CTATAAACCTTACAT	$1/8 \times 1/8 \times 3/8 \times 5/8 \times 1/8 \times 7/8$.0004

(podniska prve niske skupa Dna koja počinje od karaktera na poziciji rednog broja iteracije). Za svaki sledeći red se pravi profilna matrica prethodno izračunatih redova i na osnovu nje računa najverovatnija podniska iz one niske skupa Dna , koja ima redni broj isti kao redni broj reda matrice koji računamo. Dakle, svaki sledeći motiv u matrici se sve lepše poklapa sa ostatkom matrice. Sledeći korak u iteraciji je da izračunamo skor i ako je manji, da ažuriramo vrednost $BestMotifs$. Dakle, ona matrica koja ima najmanji skor je tražena matrica $BestMotifs$. Sam algoritam ilustrovan je sledećim pseudokodom:

```

1 GreedyMotifSearch(Dna, k, t)
2     BestMotifs ← motif matrix formed by first k-mers in each string from Dna
3     for each k-mer Motif in the first string from Dna
4         Motif1 ← Motif
5         for i = 2 to t
6             form Profile from motifs Motif1, ..., Motifi-1
7             Motifi ← Profile - most probable k-mer in the i-th string in Dna
8             Motifs ← (Motif1, ..., Motift)
9             if Score(Motifs) < Score(BestMotifs)
10                BestMotifs ← Motifs
11
return BestMotifs

```

GreedyMotifSearch je brži od *Median string* algoritma. Povoljan je i za veće k , ali cena toga je manja tačnost. Tačnost se može poboljšati primenom Laplasovog pravila, o kojem će biti nešto više reči malo kasnije.

Randomized motif search

Posmatrajući algoritam *GreedyMotifSearch*, možemo primetiti da nismo postigli pun potencijal po pitanju prelaska na probabilistički pristup. Broj iteracija je i dalje fiksan (zavisi isključivo od dužine niski iz skupa Dna i dužine ubačenog motiva), a i sam njihov redosled je nametnut deterministički. Kako uvek krećemo od istog skupa motiva, i pratimo iste utabane korake, pokretanjem ovog algoritma više puta nećemo poboljšati ni tačnost ni vreme konvergencije. Hajde da pokušamo da potpuno izbacimo determinizam iz priče i da pustimo da izgled same niske Dna i zakon verovatnoće igraju glavne uloge u konvergenciji algoritma. Prva ideja o poboljšanju probabilizma bi mogla biti vezana za sam početak algoritma. Hajde da umesto kretanja od početnih pozicija, krećemo od nekih nasumično odabranih. To sasvim sigurno donosi malo pro-

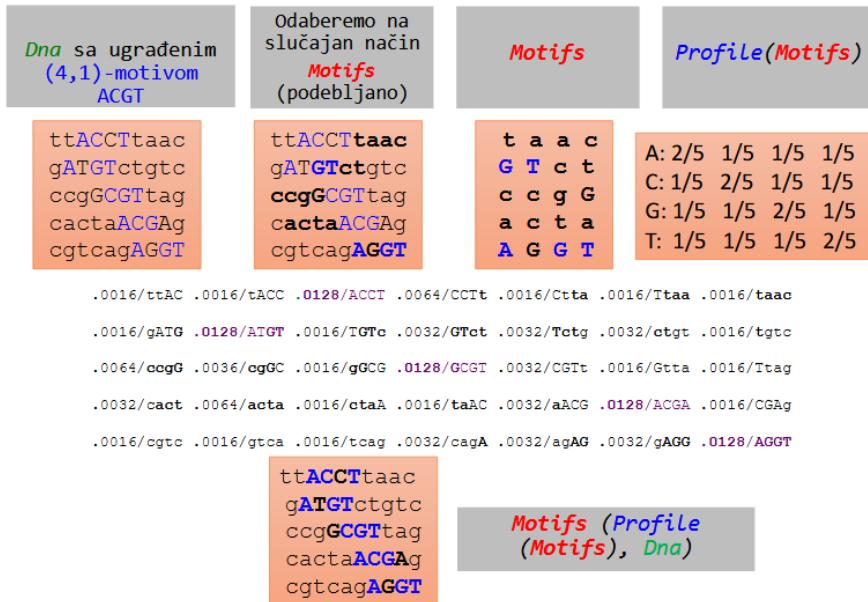
babilizma u našu priču. Međutim, i dalje imamo iteriranje po jednoj od niski, što fiksira i broj iteracija i njihov redosled. To je ono što nam najviše i smeta. Na početku sekcije o probabiličkim algoritmima, naučili smo da pravimo profilnu matricu od matrice motiva, a u algoritmu *GreedyMotifSearch* smo videli da je moguće i obrnuto. To je dobra ideja za osnovu iteracije. Od matrice motiva bismo mogli praviti profilnu matricu, a od profilne matrice ponovo matricu motiva. Ponavljanjem tog postupka, mogli bismo iterirati bez potrebe da se vezujemo za bilo kakav redosled podniski u nekoj od niski iz *Dna*. Ostalo je još da razmislimo kako bismo znali kada da stanemo. Prepostavljujući da krećemo od neke nasumične matrice motiva sa lošim, tj. velikim skorom, prepostavljujući da se krećemo ka dobrom rešenju koje ima mali skor, kao i da to činimo postepeno gradeći sve bolja, tj. verovatnija rešenja pomoću profilnih matrica, prirodno je da pomislimo da će se skor smanjivati iz iteracije u iteraciju. Dakle, iteriraćemo dokle god nam se skor smanjuje. Ova ideja je zapravo osnova algoritma *RandomizedMotifSearch*. Njegova suština leži u tome da u velikom broju iteracija ažuriramo matricu motiva i profilnu matricu tako da dobijamo sve verovatniji skup motiva, zaustavljajući se kada postignemo najniži skor. Ovaj algoritam je ilustrovan sledećim pseudokodom:

```

1 RandomizedMotifSearch(Dna, k, t)
2     randomly select k-mers Motifs = (Motif1, ..., Motift) in each string from
    ↪ Dna
3     bestMotifs ← Motifs
4     while forever
5         Profile ← Profile(Motifs)
6         Motifs ← Motifs(Profile, Dna)
7         if Score(Motifs) < Score(bestMotifs)
8             bestMotifs ← Motifs
9         else
10            return bestMotifs

```

Radi lakšeg razumevanja, hajde da ilustrujemo rad ovog algoritma na jednom primeru. Jedan takav, dosta uprošćen primer možemo videti na slici 2.12. Prikazan je algoritam *RandomizedMotifSearch* koji konvergira u jednoj iteraciji. Neka je dat skup *Dna* od 5 niski dužine 10 karaktera, sa ubačenim (4,1) motivom. Na slici 2.12. u gornjem levom uglu, prikazan je skup *Dna*, tako što su velikim slovima ispisani karakteri instanci ubacenog motiva: ACCT, ATGT, GCGT, ACGA i AGGT, crnom bojom označene mutacije na motivima, a plavom nemutirani karakteri motiva. Prvi korak jeste određivanje početne vrednosti matrice *Motifs*. Odabraćemo slučajno 5 pozicija (u opsegu od 1 do 7) na kojima će početi 4-grami kojima ćemo inicijalizovati matricu *Motifs*, i neka su to pozicije 7, 5, 1, 2 i 7. 4-grami koji počinju na ovim pozicijama: taac, GTct, ccgG, acta i AGGT, čine početnu matricu *Motifs*, koja je prikazana na slici 2.12. u gornjem centralnom delu. Sledеći korak jeste izgradnja profilne matrice na osnovu matrice *Motifs*. Prebranjem svakog karaktera u svakoj koloni matrice *Motifs* možemo odrediti *count* matricu, a zatim deljenjem svakog broja iz *count* matrice sa brojem niski, tj. 5 u našem primeru, dobijamo profilnu matricu. Kako izgleda profilna matrica u našem primeru, može se videti na slici 2.12 u gornjem desnom uglu. Sada kada imamo profilnu matricu, možemo izračunati nove najverovatnije motive u niskama iz *Dna*, i ažurirati matricu *Motifs*. Postupak računanja matrice motiva prikazan je na slici 2.12 u središnjem delu. Za svaku nisku iz skupa *Dna*, odredićemo verovatnoće svih njenih podniski, na osnovu profilne matrice, i iz svake niske odabrati po jednu koja ima najveću verovatnoću (na slici su prikazane ljubičastom bojom). Sledеći korak bi bio da ponovo napravimo profilnu matricu itd. Međutim, u našem primeru, vidimo da je to kraj. Dobili smo traženo rešenje. Treba napomenuti da je ovo veštački kreiran primer i da je u praksi broj iteracija dosta veći.

Slika 2.12: Primer rada algoritma *RandomizedMotifSearch*

Velika prednost ovog algoritma je u nasumičnom izboru početnog skupa motiva, što nam omogućava da ga pokrećemo iznova i izaberemo najbolje rešenje.

Gibsovo sempliranje

Algoritam *RandomizedMotifSearch* je dobar i efikasan, ali ima jednu malu manu. Matrica motiva se potpuno menja iz iteracije u iteraciju, što može dovesti do toga da se izmene i neki dobro izračunati redovi. To svakako ne želimo. Ako bismo u svakoj iteraciji menjali samo po jedan motiv iz matrice, tj. posle manjih izmena pravili novu profilnu matricu i proveravali skor, mogli bismo bolje da kontrolišemo to kvarenje već nameštenih motiva. Ova modifikacija algoritma *RandomizedMotifSearch* se naziva Gibbsovo sempliranje. Na prvi pogled deluje kao da povećava broj iteracija i usporava konvergenciju, ali to što su promene manje, nikako ne znači da su manje i efikasne. Ako idemo pravo sitnjim koracima, stižemo brže i lakše na cilj, nego ako krupnim koracima idemo levo-desno. Osnovna razlika Gibbsovog sempliranja u odnosu na algoritam *RandomizedMotifSearch* se nalazi u sadržaju iterativnog koraka. Umesto da od cele matrice motiva pravimo profilnu matricu i na osnovu nje ažuriramo celu matricu motiva, sada ćemo nasumično odabrati jedan motiv koji ćemo izbaciti, profilnu matricu praviti od ostatka matrice, a ažuriranje matrice motiva vršiti samo nad izbačenim redom u matrici. Postupak kreiranja same profilne matrice i ažuriranja motiva je potpuno isti kao u algoritmu *RandomizedMotifSearch*. Umesto pseudokodom, ovaj put ćemo algoritam prikazati u koracima:

1. Formirati *Motifs* izborom jednog k-grama iz svake sekvene **na slučajan način**
2. **Na slučajan način** odabrat jedan od k-grama i ukloniti ga iz *Motifs*; označimo sekvenu kojoj taj k-gram pripada sa *RemovedSequence*
3. Kreirati profilnu matricu *Profile* od preostalih k-grama u *Motifs*
4. Za svaki k-gram iz *RemovedSequence*, izračunati $Pr(k-mer|Profile)$; na taj način dobijamo $n - k + 1$ verovatnoća: $p_1, p_2, \dots, p_{n-k+1}$
5. **Bacimo kockicu** sa $n - k + 1$ strana kod koje je verovatnoća da će pasti na i-tu stranu proporcionalna verovatnoći p_i

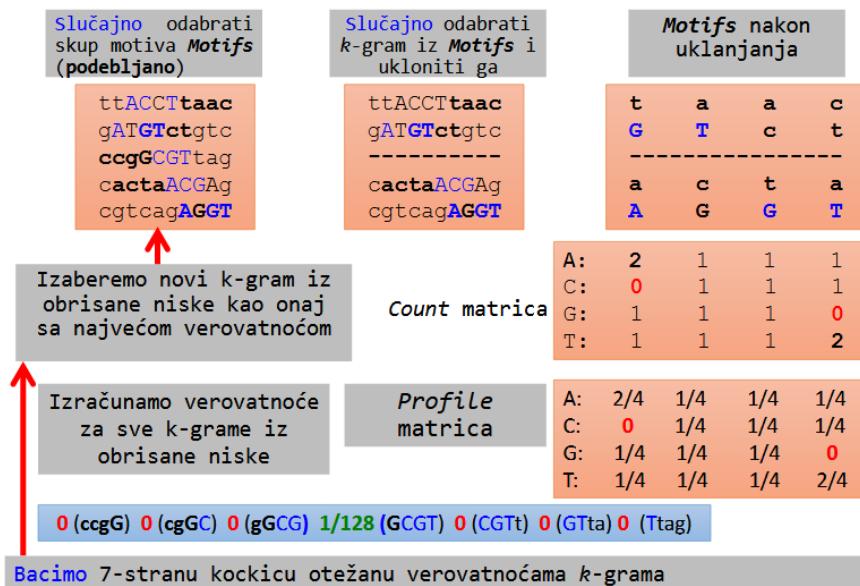
6. Odredimo k-gram iz sekvence *RemovedSequence* kao onaj koji ima najveću verovatnoću i dodamo ga u *Motifs*

7. Ponavljamo korake 2-6

Radi lakšeg razumevanja, prikazaćemo i ovaj algoritam na istom primeru kao i *RandomizedMotifSearch*. Skica rada algoritma je prikazana na slici 2.13.

Neka je ponovo dat isti skup *Dna* od 5 niski dužine 10 karaktera, sa ubačenim (4,1) motivom. Na slici 2.13. u gornjem levom uglu, prikazan je skup *Dna*, tako što su velikim slovima ispisani karakteri instanci ubačenog motiva: ACCT, ATGT, GCGT, ACGA i AGGT, crnom bojom označene mutacije na motivima, a plavom nemutirani karakteri motiva. Prvi korak je isti kao i u algoritmu *RandomizedMotifSearch*: određivanje početne vrednosti matrice *Motifs*. Odabramo slučajno 5 pozicija (u opsegu od 1 do 7) na kojima će početi 4-grami kojima ćemo inicijalizovati matricu *Motifs*, i neka su to pozicije 7, 5, 1, 2 i 7. 4-grami koji počinju na ovim pozicijama: taac, GTct, ccgG, acta i AGGT, čine početnu matricu *Motifs*. Sledeći korak jeste izbacivanje jednog motiva. Nasumičnim izborom jednog broja iz opsega (1,5), odredili smo motiv iz reda 3 da bude izbačen. Matrica *Motifs* sada izgleda kao na slici 2.13 u gornjem desnom uglu. Naredni korak je izgradnja profilne matrice na osnovu matrice *Motifs*. Ovaj postupak je takođe isti kao u algoritmu *RandomizedMotifSearch*, samo je skup nad kojim radimo za jedan 4-gram manji. Prebrajanjem svakog karaktera u svakoj koloni matrice *Motifs* možemo odrediti *count* matricu (slika 2.13, centar-desno), a zatim deljenjem svakog broja iz *count* matrice sa brojem motiva, tj. 4 u našem primeru, dobijamo profilnu matricu. Kako izgleda profilna matrica u našem primeru, može se videti na slici 2.13 u donjem desnom uglu. Sada kada imamo profilnu matricu, možemo izračunati novi najverovatniji motiv u trećoj niski iz *Dna*, i ponovo popuniti treći red u matrici *Motifs*. Vidimo da je podniska GCGT jedina sa pozitivnom verovatnoćom, pa samim tim i najvećom, tako da njome popunjavamo treći red matrice *Motifs*. Sledeći korak bi bio da ponovo izbacimo nasumično odabrani motiv, napravimo profilnu matricu itd. Iterirali bismo sve dok skor ne prestane da se smanjuje.

Slika 2.13: Primer rada algoritma Gibbsovo sempliranje



Međutim, sada ćemo se ovde zaustaviti kako bismo primetili nešto mnogo važno. Kada smo računali verovatnoće podniski kandidata za zamenu trećeg motiva, dobili smo samo jednu ne-nulu vrednost od 7. To nije dobro. Jedna nula u profilnoj matrici može anulirati verovatnoće

koje su sve do množenja sa njom bile prilično velike. Zbog toga one nisu baš poželjne. Ako zastanemo malo i razmislimo, shvatićemo da nisu previše ni realne, tj. da ne odražavaju realnu verovatnoću, već da su u velikom broju slučajeva uzrokovane malim obimom skupa nad kojim računamo. Zbog toga, modifikacija nula u profilnoj matrici predstavlja vrlo dobru i poželjnu modifikaciju ovog algoritma, a ta modifikacija se sprovodi korišćenjem Laplasovog pravila.

Mala napomena: Setimo se da smo već pominjali Laplasovo pravilo kao dobro poboljšanje i *GreedyMotifSearch* i *RandomizedMotifSearch* algoritama. Zapravo, ovo je korisna modifikacija svakog algoritma koji se zasniva na računanju profilne matrice i korišćenju njenih vrednosti.

Pre nego pređemo na modifikaciju, naglašićemo još da se prednost Gibsovog algoritma ogleda u najmanjem skoru, kao i to da je mana ovog algoritma zaglavljivanje u lokalnom minimumu usled pretrage ograničenog skupa rešenja.

Laplasovo pravilo kao poboljšanje Gibsovog semplera

Davne 1650. godine, Oliver Kromvel je napisao: "Molim Vas, tako Vam Hrista, mislite da je moguće da grešite". Ovu izjavu, je uobličio Denis Lindli i nazvao Kromvelovo pravilo: "Traba da ostavimo malu mogućnost da Sunce sutra neće izaći". Nešto novija replika bi se mogla naći u seriji Smolvil i rečima Leksa Lutora: "Uvek ostavljam malu mogućnost da nisam u pravu. Tako me ništa ne može iznenaditi. Šuština Kromvelog pravila leži u tome da, ukoliko potpuno odbacimo mogućnost nekog događaja, sasvim sigurno ostavljamo mogućnost pogreške. Zbog toga, treba izbegavati verovatnoće 0 i 1.

Za poboljšanje Gibsovog semplera (zapravo se može primeniti na bilo koji algoritam koji računa profilnu matricu), izvršićemo malu modifikaciju algoritma primenom Laplasovog pravila. Primenjujemo Laplasovo a ne Kromvolevo pravilo, samo iz razloga notacije i matematičkog zapisa. Laplasovo pravilo je samo preciznija (matematička) formulacija Kromvelovog pravila, sa istom suštinom.

Laplasovo pravilo: U malim skupovima podataka uvek postoji šansa da se događaj koji je moguć ne desi. Slučajni algoritmi uvode pseudovrednosti koje povećavaju verovatnoće retkih događaja i eliminisu frekvencije jednake nuli zabeležene na osnovu iskustva.

Suština Laplasovog pravila leži u tome da, ukoliko znamo da se događaj nekada u prošlosti desio, on ne može imati verovatnoću nula. Stoga u računanje, pored vrednosti dobijenih u našem eksperimentu, ubacujemo i dve pseudovrednosti: događaj se desio i događaj se nije desio.

Računanje uslovne verovatnoće bez primene Laplasovog pravila bi bilo:

Ako su X_1, \dots, X_{n+1} uslovno nezavisne slučajne logičke promenljive (neuspех 0, uspeh 1), tada: $Pr(X_{n+1} = 1 | X_1 + \dots + X_n = s) = s/n$

Računanje uslovne verovatnoće sa primenom Laplasovog pravila bi bilo:

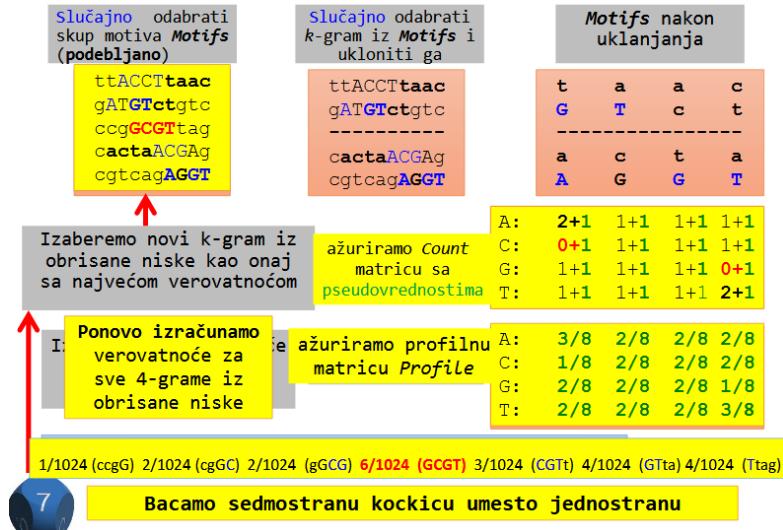
Ako su X_1, \dots, X_{n+1} uslovno nezavisne slučajne logičke promenljive (neuspех 0, uspeh 1), tada: $Pr(X_{n+1} = 1 | X_1 + \dots + X_n = s) = (s + 1)/(n + 2)$

Modifikacija Gibsovog algoritma primenom Laplasovog pravila je sadržana u dva mala koraka. Prva izmena je u okviru računanja *count* matrice, gde se vrednost svakog polja uvećava za 1. To je zapravo pseudovrednost: događaj se desio. Pod događaj, podrazumeva se pojava određenog karaktera na određenoj poziciji u matrici motiva. Druga izmena je u okviru računanja profilne matrice, zato što *count* matrica izgleda nešto drugčije.

Modifikaciju Gibsovog algoritma ćemo prikazati samo na primeru, jer je skup koraka u suštini isti i ne prikazuje najbolje samu modifikaciju. Koristićemo isti primer na kome smo već demonstrirali rad Gibsovog semplera. Na slici 2.14. prikazan je rad modifikovanog Gibsovog semplera.

Možemo primetiti da *count* i profilna matrica izgledaju nešto drugčije (dodata su pseudovrednosti), kao i da je rezultat 7 ne-nula verovatnoća.

Slika 2.14: Primer rada Gibsovog sempliranja sa primenom Laplasovog pravila



2.3.6 Koji princip odabrat?

Odgovor na ovo pitanje je - nema pravila. Nekada će se bolje pokazati jedan, a nekada drugi. Najbolje je da isprobamo više algoritama i vidimo koji se najbolje ponaša u našem slučaju. Možemo koristiti već zapažene prednosti i mane u izboru, kao na primer biranje *Median string* algoritma pri malim vrednostima k , ali isprobavanje je ipak najpouzdaniji pristup.

2.4 Zadaci sa vežbi

U nastavku će biti predstavljeni zadaci sa vežbi na kursu rađeni u programskom jeziku *Python*.

2.4.1 MedianString

```

1 def number_to_symbol(n):
2     pairs = {
3         0: 'A',
4         1: 'T',
5         2: 'C',
6         3: 'G'
7     }
8
9     return pairs[n]
10
11
12 def number_to_pattern(n, k):
13     if k == 1:
14         return number_to_symbol(n)
15

```

```

16     prefix_index = n // 4
17     r = n - n // 4
18     symbol = number_to_symbol(r)
19
20     return number_to_pattern(prefix_index, k - 1) + symbol
21
22
23 def hamming_distance(pattern_p, pattern):
24     k = len(pattern_p)
25
26     distance = 0
27
28     for i in range(k):
29         if pattern_p[i] != pattern[i]:
30             distance += 1
31
32     return distance
33
34
35 # Sumiranje hamingovih rastojanja izmedju pattern niske i svih DNK sekvenci
36 def d(pattern, dna):
37     k = len(pattern)
38     distance = 0
39
40     for dna_string in dna:
41
42         h_dist = float('inf')
43
44         for i in range(len(dna_string) - k):
45
46             pattern_p = dna_string[i:i+k]
47             dist = hamming_distance(pattern_p, pattern)
48
49             if dist < h_dist:
50                 h_dist = dist
51
52         distance += h_dist
53     return distance
54
55 # Pronalazenje median niske
56 def median_string(dna, k):
57     distance = float('inf')
58     median = ''
59
60 # Za svaku od  $4^k$  niski pretpostavimo da je median niska i proverimo kolika
61 # je njena udaljenost od DNK sekvenci
62     for i in range(4**k):
63         pattern = number_to_pattern(i, k)
64
65         current_distance = d(pattern, dna)
66
67         # Ako je tekuća kandidat median niska bolja od dosadašnje najbolje
68         # vrednosti se azuriraju i pamti se najbolje

```

```

69     if distance > current_distance:
70         distance = current_distance
71         median = pattern
72
73     return median
74
75
76 def main():
77     dna = [
78         'GTAGATGTCATTAGCATGCAC',
79         'CCTAGCCACTCTGCCATGTCG',
80         'AACTCGTGCATTCTACGACTG',
81         'AAACTTCCGGATCTTCATAC',
82         'CTACATCATCGAAGGCTACGC'
83     ]
84
85     print(median_string(dna, 4))
86
87
88 if __name__ == "__main__":
89     main()

```

2.4.2 GreedyMotifSearch

```

1 import copy
2
3 def symbol_to_number(n):
4     pairs = {
5         'A': 0,
6         'T': 1,
7         'C': 2,
8         'G': 3
9     }
10
11    return pairs[n]
12
13 # Formiranje profil matrice za zadati skup motiva
14 def profile_from_motifs(motifs, k, t):
15     profile = [[1 for i in range(k)] for x in range(4)]
16
17     for j in range(k):
18         for i in range(t):
19             index = symbol_to_number(motifs[i][j])
20             profile[index][j] += 1
21
22     for j in range(k):
23         for i in range(4):
24             profile[i][j] /= (t+2)
25
26     return profile
27
28
29 # Izracunavanje verovatnoce pojave pattern sekvence u odnosu na zadati profil

```

```

30 def probability(pattern, profile):
31     prob = 1
32
33     for j in range(len(pattern)):
34         c = pattern[j]
35         index = symbol_to_number(c)
36
37         prob *= profile[index][j]
38
39     return prob
40
41 # Pronalazenje najverovatnijeg podstringa duzine k iz zadate DNK sekvence
42 # koji je najverovatniji u odnosu na zadati profil
43 def most_probable_k_mer(dna_string, profile, k):
44
45     best_k_mer = ''
46     best_probability = -1
47
48     for i in range(len(dna_string) - k):
49         pattern = dna_string[i:i+k]
50         pattern_prob = probability(pattern, profile)
51
52         if pattern_prob > best_probability:
53             best_probability = pattern_prob
54             best_k_mer = pattern
55
56     return best_k_mer
57
58 # Izracunavanje ukupnog skora za skup motiva
59 def score(motifs, k):
60     t = len(motifs)
61
62     total_score = 0
63
64     for j in range(k):
65
66         counts = [0, 0, 0, 0]
67
68         for i in range(t):
69             c = motifs[i][j]
70             index = symbol_to_number(c)
71             counts[index] += 1
72
73         max_index = 0
74
75         for i in range(1,4):
76             if counts[i] > counts[max_index]:
77                 max_index = i
78
79         total_score += t - counts[max_index]
80
81     return total_score
82

```

```

83
84 # Pohlepno pronalazenje motiv niski
85 def greedy_motif_search(dna, k, t):
86
87 # Pretpostavimo da najbolji skup motiv sekvenci predstavljuju
88 # prefiksi duzine k svih niski
89     best_motifs = [dna_string[0:k] for dna_string in dna]
90
91 # i izracunamo njihov ukupan skor
92     best_score = score(best_motifs, k)
93
94     first_string = dna[0]
95
96     for i in range(len(first_string) - k):
97
98         motifs = []
99
100    # Za svaku podnisku duzine k iz prve DNK sekvence kazemo da u tekucoj
101    # iteraciji predstavlja prvi motiv
102
103        motifs.append(first_string[i:i+k])
104
105    # Iz svake od preostalih t-1 DNK sekvenci izdvajamo podstring duzine k
106    # koji je najverovatniji u odnosu na profil dobijen od motiva dobijenih iz
107    # prethodnih iteracija. Taj podstring dodajemo u skup motiva kako bi se
108    # koristio u narednoj iteraciji za pronalazenje sledeceg motiva
109
110        for j in range(1, t):
111            profile = profile_from_motifs(motifs, k, j)
112            motifs.append(most_probable_k_mer(dna[j], profile, k))
113
114    # Sa svaki izgenerisami skup motiva proveravamo da li daje bolji skor
115    # u odnosu na do sada najbolji pronadjen
116    current_score = score(motifs, k)
117
118    # Ako je trenutni skup motiva bolji od dosadasnjeg azuriraju se vrednosti
119    if current_score < best_score:
120        best_motifs = copy.deepcopy(motifs)
121        best_score = current_score
122
123    return best_motifs
124
125 def main():
126     dna = [
127         'GTAGATGTCATTAGCATGCAC',
128         'CCTAGCCACTCTGCCATGTCG',
129         'AACTCGTGCATTCTACGACTG',
130         'AAACTTCCGGATCTTCATAC',
131         'CTACATCATCGAAGGCTACGC'
132     ]
133
134     print(greedy_motif_search(dna, 4, len(dna)))
135

```

```
136
137 if __name__ == "__main__":
138     main()
```

2.4.3 RandomizedMotifSearch

```
1 import copy
2 import random
3
4 def symbol_to_number(n):
5     pairs = {
6         'A': 0,
7         'T': 1,
8         'C': 2,
9         'G': 3
10    }
11
12    return pairs[n]
13
14 def probability(pattern, profile):
15     prob = 1
16
17     for j in range(len(pattern)):
18         c = pattern[j]
19         index = symbol_to_number(c)
20
21         prob *= profile[index][j]
22
23     return prob
24
25 # Izdvajanje pseudoslucajno odabranih podniski iz DNK sekvenci u skupu
26 def random_k_mers(dna, k, t):
27     k_mers = []
28
29     for i in range(t):
30         start = random.randrange(0, len(dna[i]) - k)
31         dna_string = dna[i]
32         k_mers.append(dna_string[start:start+k])
33
34     return k_mers
35
36
37 def profile_from_motifs(motifs, k, t):
38     profile = [[1 for i in range(k)] for x in range(4)]
39
40     for j in range(k):
41         for i in range(t):
42             index = symbol_to_number(motifs[i][j])
43             profile[index][j] += 1
44
45     for j in range(k):
46         for i in range(4):
47             profile[i][j] /= (t+2)
```

```
48
49     return profile
50
51 # Pronalazenje podniski DNK sekvenci iz skupa koje su najverovatnije u
52 # odnosu na zadati profil i one zajedno cine skup motiva
53 def motifs_from_profile(profile, dna):
54     motifs = []
55     k = len(profile[0])
56
57     for dna_string in dna:
58         motifs.append(most_probable_k_mer(dna_string, profile, k))
59
60     return motifs
61
62
63 def score(motifs, k):
64     t = len(motifs)
65
66     total_score = 0
67
68     for j in range(k):
69
70         counts = [0, 0, 0, 0]
71
72         for i in range(t):
73             c = motifs[i][j]
74             index = symbol_to_number(c)
75             counts[index] += 1
76
77         max_index = 0
78
79         for i in range(1,4):
80             if counts[i] > counts[max_index]:
81                 max_index = i
82
83         total_score += t - counts[max_index]
84
85     return total_score
86
87 def most_probable_k_mer(dna_string, profile, k):
88
89     best_k_mer = ''
90     best_probability = -1
91
92     for i in range(len(dna_string) - k):
93         pattern = dna_string[i:i+k]
94         pattern_prob = probability(pattern, profile)
95
96         if pattern_prob > best_probability:
97             best_probability = pattern_prob
98             best_k_mer = pattern
99
100    return best_k_mer
```

```

101
102 # Pronalazenje motiva koriscenjem algoritma za pseudoslucajni izbor
103 def randomized_motif_search(dna, k, t):
104
105 # Pretpostavimo da najbolji skup motiva cine slucajno odabrane podniske
106 # iz skupa DNK sekvenci
107     motifs = random_k_mers(dna, k, t)
108     best_motifs = copy.deepcopy(motifs)
109     best_score = score(best_motifs, k)
110
111 # Dok se skor popravlja svakom iteracijom:
112     while True:
113
114         # Formiramo profil od tekucih motiva
115         profile = profile_from_motifs(motifs, k, t)
116
117         # A zatim motive od dobijenog profila
118         motifs = motifs_from_profile(profile, dna)
119
120         current_score = score(motifs, k)
121
122         if current_score < best_score:
123             best_score = current_score
124             best_motifs = copy.deepcopy(motifs)
125         else:
126             return best_motifs
127
128
129 def main():
130     dna = [
131         'GTAGATGTCATTAGCATGCAC',
132         'CCTAGCCACTCTGCCATGTCG',
133         'AACTCGTGCATTCTACGACTG',
134         'AAACTTTCCGGATCTTCATAC',
135         'CTACATCATCGAAGGCTACGC'
136     ]
137
138     print(randomized_motif_search(dna, 4, len(dna)))
139
140
141 if __name__ == "__main__":
142     main()

```

2.4.4 GibbsSampler

```

1 import copy
2 import random
3
4 def symbol_to_number(n):
5     pairs = {
6         'A': 0,
7         'T': 1,
8         'C': 2,

```

```

9         'G': 3
10        }
11
12    return pairs[n]
13
14 def probability(pattern, profile):
15     prob = 1
16
17     for j in range(len(pattern)):
18         c = pattern[j]
19         index = symbol_to_number(c)
20
21         prob *= profile[index][j]
22
23     return prob
24
25 def random_k_mers(dna, k, t):
26     k_mers = []
27
28     for i in range(t):
29         start = random.randrange(0, len(dna[i]) - k)
30         dna_string = dna[i]
31         k_mers.append(dna_string[start:start+k])
32
33     return k_mers
34
35 def profile_from_motifs(motifs, k, t):
36     profile = [[1 for i in range(k)] for x in range(4)]
37
38     for j in range(k):
39         for i in range(t):
40             index = symbol_to_number(motifs[i][j])
41             profile[index][j] += 1
42
43     for j in range(k):
44         for i in range(4):
45             profile[i][j] /= (t+2)
46
47     return profile
48
49 def score(motifs, k):
50     t = len(motifs)
51
52     total_score = 0
53
54     for j in range(k):
55
56         counts = [0, 0, 0, 0]
57
58         for i in range(t):
59             c = motifs[i][j]
60             index = symbol_to_number(c)
61             counts[index] += 1

```

```

62
63     max_index = 0
64
65     for i in range(1,4):
66         if counts[i] > counts[max_index]:
67             max_index = i
68
69         total_score += t - counts[max_index]
70
71     return total_score
72
73 def most_probable_k_mer(dna_string, profile, k):
74
75     best_k_mer = ''
76     best_probability = -1
77
78     for i in range(len(dna_string) - k):
79         pattern = dna_string[i:i+k]
80         pattern_prob = probability(pattern, profile)
81
82         if pattern_prob > best_probability:
83             best_probability = pattern_prob
84             best_k_mer = pattern
85
86     return best_k_mer
87
88 # Pronalazenje skupa motiva nakon N iteracija koriscenjem Gibbs sampler-a
89 def gibbs_sampler(dna, k, t, N):
90     motifs = random_k_mers(dna, k, t)
91     best_motifs = copy.deepcopy(motifs)
92     best_score = score(best_motifs, k)
93
94     for j in range(N):
95
96         # Biramo slucajno i iz skupa [0,t)
97         i = random.randrange(0,t)
98
99         # Formiramo skup motiva koji se sastoji od svih dosadasnjih motiva osim
100        ↪ i-tog
101        selected_motifs = copy.deepcopy(motifs)
102        del selected_motifs[i]
103
104        # Pravimo profil od odabranih motiva (bez i-tog)
105        profile = profile_from_motifs(selected_motifs, k, t-1)
106
107        # Za i-ti motiv postavljamo najverovatniji podstring duzine k iz i-te
108        ↪ DNK sekvence, u odnosu na dobijeni profil
109        motifs[i] = most_probable_k_mer(dna[i], profile, k)
110        del selected_motifs
111
112        # Ako dobijeni motiv ima skor bolji od do sada najboljeg, vrednosti se
113        ↪ azuriraju
114        current_score = score(motifs, k)

```

```
112
113     if current_score < best_score:
114         best_motifs = copy.deepcopy(motifs)
115         best_score = current_score
116
117     return best_motifs
118
119 def main():
120     dna = [
121         'GTAGATGTCATTAGCATGCAC',
122         'CCTAGCCACTCTGCCATGTCG',
123         'AACTCGTGCATTCTACGACTG',
124         'AAACTTCCGGATCTTCATAC',
125         'CTACATCATCGAAGGCTACGC'
126     ]
127
128     print(gibbs_sampler(dna, 4, len(dna), 500))
129
130
131 if __name__ == "__main__":
132     main()
```


Glava 3

Kako složiti genomske slagalice od milion delova?

U ovom poglavlju predstavićemo dati problem i prikazati kako možemo primeniti grafovske algoritme nad problemom slaganja genomske slagalice. Poglavlje će započeti pričom o sekvencirajućem genoma. Do sada smo videli šta za nas znači pojam DNK – da se podsetimo, to je niska karaktera nad abecedama $\Sigma = \{A, C, G, T\}$. Biološki posmatrano, DNK je molekul koji se nalazi u svakoj ćeliji svakog organizma i da je u njemu zapisan način pravilnog razvoja i funkcionalnosti svakog organizma.

3.1 Šta je sekvenčiranje genoma?

Sa biološke strane, genom jednog organizma predstavlja njegov genetski materijal. Pod genetskim materijalom podrazumevamo delove genoma koji se nasleđuju i koji su predstavnici jedne vrste dele u velikoj meri. Kod većine organizama, genetski materijal je sadržan u DNK, odnosno, genom je ekvivalentan sa DNK molekulima. Kod nekih drugih organizama koji su u manjini, kao što su, na primer, virusi, važi da oni ne sadrže DNK i njihov genetski materijal se nalazi u ribonukleinskoj kiselini – RNK – o kojoj će biti reči u nastavku.

Kod čoveka, genom sadrži oko tri milijarde nukleotida. Dakle, sa računarske strane posmatrano genom je niska karaktera nad abecedama $\Sigma = \{A, C, G, T\}$. Kompleksnost organizma nije u relaciji sa veličinom genoma. Genomi nekih organizama su i stotinu puta veći od humanog genoma. Na primer, jedna vrsta amebe ima 670 milijardi nukleotida ili jedna vrsta kaktusa koja raste u Japanu ima 150 milijardi nukleotida.

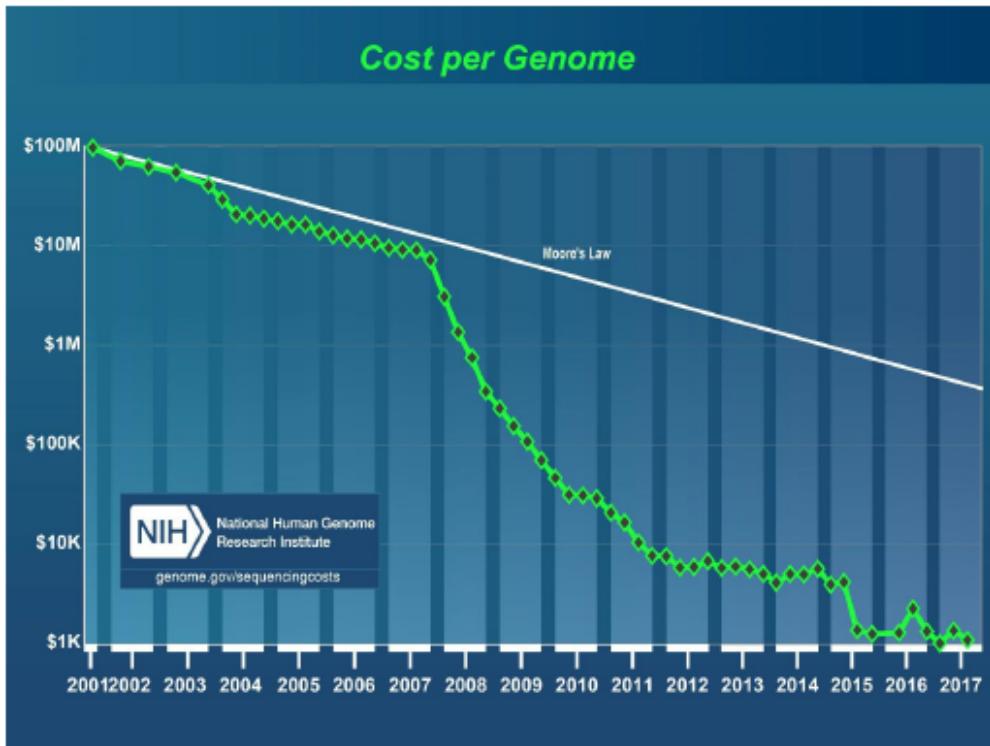
Sekvenčiranje genoma podrazumeva otkrivanje sastava genoma. U pitanju je eksperimentalni proces – da bismo saznali šta se nalazi u sastavu jednog genoma, potreban nam je uzorak tkiva odgovarajuće vrste. U nastavku dajemo kratak pregled razvoja sekvenčiranja genoma.

3.1.1 Kratka istorija sekvenčiranja genoma

Kao što smo videli, sekvenčiranje genoma je eksperimentalni proces, za koji je neophodna veoma sofisticirana tehnologija. Pre svega možemo govoriti o razvoju fizičko-hemijskih tehnika koje bi dovele do mogućnosti saznavanja sastava genoma. Walter Gilbert i Frederick Sanger su 1977. godine razvili nezavisne metode sa sekvenčiranje genoma, za koje su, 1980. godine, podelili Nobelovu nagradu. Međutim, iako su njihovi metodi bili pionirski u ovoj oblasti, njihove metode za sekvenčiranje su bile veoma skupe – za sekvenčiranje humanog genoma je bilo potrebno 3 milijarde dolara.

Krajem 2000-ih Sanger metodom je sekvencioniran veliki broj genoma. Visoka cena je bila ograničavajući faktor i za dalji napredak je bila neophodna nova tehnologija sekvencioniranja.

NGS (skr. *Next Generation Sequencing*) predstavlja metode nove generacije sekvencioniranja, odnosno, novu generaciju mašina sekvencera koji vrše sekvencioniranje. *Illumina*, jedan od proizvođača sekvencera, smanjuje trošak sekvencioniranja humanog genoma sa 3 milijarde na 10 hiljada dolara. Kompanija *Complete Genomics* otvara genomsku fabriku u Silikonskoj dolini koja sekvencionira stotine genoma mesečno. Pekinški genomski institut (*Beijing Genome Institute*, skr. *BGI*) preuzima *Complete Genomics* 2013. godine i postaje najveći svetski centar za sekvencioniranje genoma. Na slici 3.1 prikazano je kako se cena sekvencioniranja menjala godinama.



Slika 3.1: Cena sekvencioniranja kroz istoriju.

3.1.2 Sekvenciranje ličnih genoma

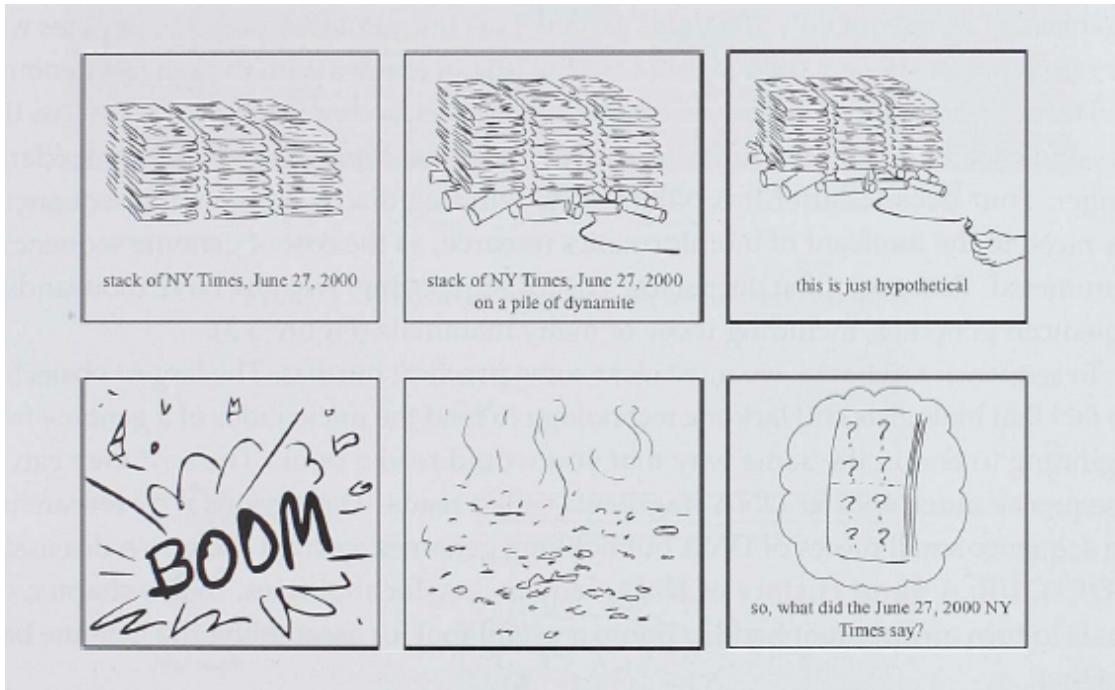
Prirodno je zapitati se o značaju poznavanja sastava genoma. Što je tiče biljaka, neke od primena sekvenciranja su: razvoj novih biljnih vrsta u poljoprivredi, određivanje pogodnog podneblja za neku biljnu vrstu, u farmaciji, i dr. Međutim, najznačajnija primena je u sekvenciranju ličnih genoma.

Genomi se kod različitih ljudi razlikuju na malom broju pozicija (u proseku sadrže jednu mutaciju na hiljadu nukleotida). Ova razlika je odgovorna za, na primer, različite visine kod ljudi, da li će imati sklonost ka visokom holesterolu ili ne, za veliki broj genetskih bolesti, itd.

Godine 2010. Nicholas Volker je postao prvo ljudsko biće čiji je život spašen zahvaljujući genomskom sekvencioniranju. Lekari nisu mogli da postave tačnu dijagnozu i morali su da ga podvrgnu velikom broju operacija pokušavajući da je utvrde. Sekvenciranje je otkrilo retku mutaciju na jednom genu (XIAP) koja je bila povezana sa oštećenjem njegovog imunog sistema. Ovo otkriće je navelo lekare na adekvatnu terapiju koja je rešila problem.

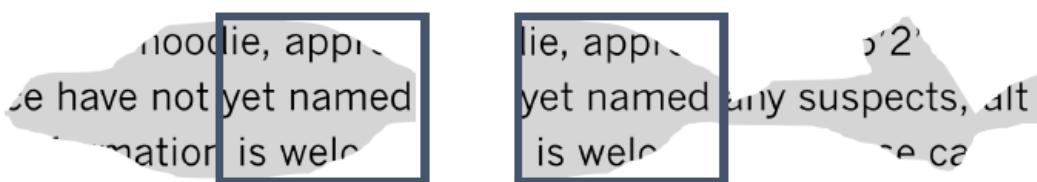
3.2 Eksplozija u štampariji

Razmotrimo sledeći primer. Neka imamo hiljadu kopija istog izdanja novina na jednoj gomili, a ispod njih postavljen je dinamit. Upalimo fitil i zamislimo da nije sve samo izgorelo već da se raspršilo u milione delića papira. Kako možemo da iskoristimo te deliće da bismo saznali koje su bile vesti iz tog izdanja? Ovaj problem nazvaćemo *Problem novina* (videti sliku 3.2).



Slika 3.2: Problem novina poslužiće nam u razumevanju problema slaganja genoma.

Problem novina je mnogo teži nego što izgleda. Kako smo imali više kopija istog izdanja, i kako smo izgubili neki deo informacija prilikom eksplozije, ne možemo samo da prilepimo deliće novina kao da su slagalica. Umesto toga, potrebno je da preklopimo delove različitih novina kako bismo rekonstruisali jedan primerak.



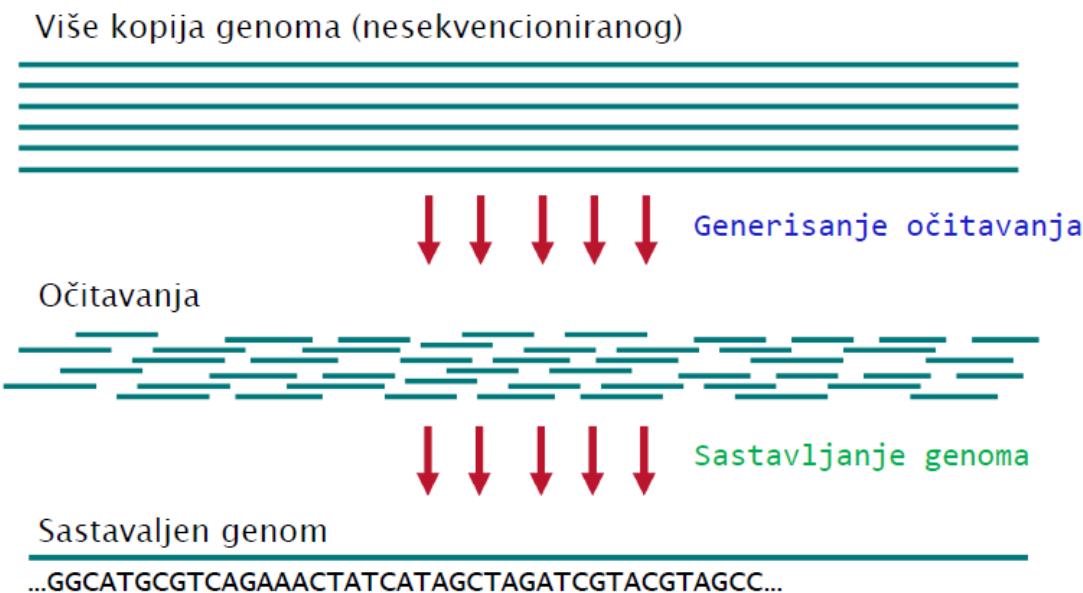
Slika 3.3: Spajanje delova različitih novina koji se jednim delom preklapaju.

Određivanje redosleda nukleotida u genomu, odnosno sekvenciranje genoma, predstavlja bitan problem u bioinformatici. Već smo pomenuli da dužine genoma variraju – humani genom je dugačak oko 3 milijarde nukleotida, dok je genom jednoćelijskog organizma *Amoeba dubia* čak 200 puta duži.

Razmotrimo sada povezanost problema novina i sekvenciranja genoma. Kopije izdanja u problemu novina odgovaraju ono predstavlja ulaz u sekvencerima – uzorak tkiva. Moderne

mašine za sekvenciranje ne mogu da pročitaju ceo genom nukleotid po nukleotid od početka do kraja (kao što bismo pročitali knjigu). Mogu samo da isekaju genom i generišu njegova kratka *očitavanja* (engl. *reads*). Kako to zapravo funkcioniše?

Na slici 3.4 ilustrovan je proces sekvenciranja. Sekvencer dobija milione kopija istog genoma. Zatim vrši očitavanja čime dobijamo delice odnosno kratke podnische. Neki delovi odnosno očitavanja biće izgubljena (kao delići novina u eksploziji, dakle gubimo deo informacije). Očitavanja su izmešana i ono što nam sekvencer daje je zapravo kolekcija podniski koje treba spojiti u jednu. Sastavljanje genoma nije isto kao i slaganje slagalice – moramo da koristimo preklapajuća očitavanja da bismo rekonstruisali genom.



Slika 3.4: Ilustracija problema.

3.3 Problem sekvenciranja genoma

Do sada smo videli šta predstavlja sekvenciranje genoma, koja je njegova biološka podloga i kako se on definiše kao biološki problem. Pređimo sada na formulisanje računarskog problema sekvenciranja genoma kao problem rekonstrukcije niske.

Problem sekvencioniranja genoma: Rekonstruisati genom na osnovu očitavanja.

Ulas: Kolekcija niski *Reads*.

Izlaz: Niska *Genome* rekonstruisana na osnovu *Reads*.

Ovo nije dobro definisan problem. Potrebno je uvesti dodatne pojmove kako bismo uspeli da problem sekvencioniranja genoma predstavimo kao problem rekonstrukcije niske.

Definišemo pojam *k-gramski sastav niske* na sledeći način. *k*-gramska sastav niske *Text*, u označi $Composition_k(Text)$, predstavlja kolekciju podniski dužine *k* niske *Text*, pri čemu su u kolekciju uključeni duplikati. Na primer, neka je *Text* = TAATGCCATGGGATGTT. Njen 3-gramska sastav niske *Text* izgleda:

```

1 Composition_3(TAATGCCATGGGATGTT) =
2           TAA
3           AAT
4           ATG
5           TGC
6           GCC
7           CCA
8           CAT
9           ATG
10          TGG
11          GGG
12          GGA
13          GAT
14          ATG
15          TGT
16          GTT

```

odnosno, ako kolekciju uredimo po leksikografskom poređenju:

```

1 Composition_3(TAATGCCATGGGATGTT) =
2 AAT ATG ATG ATG CAT CCA GAT GCC GGA GGG GTT TAA TGC TGG TGT

```

Sada možemo malo bolje da definišemo problem.

Problem rekonstrukcije niske: Rekonstruisati nisku na osnovu njenog k -gramskog sastava.

Ulaz: Kolekcija k -grama.

Izlaz: Niska *Genome* takva da je $\text{Composition}_k(\text{Genome})$ ekvivalentno kolekciji k -grama.

Započnimo prvo sa naivnim pristupom rešavanju ovog problema. Odaberimo jedan k -gram za početni. Zatim nižemo ostale tako da se sufiks poslednjeg odabranog poklopi sa prefiksom nekog od preostalih k -grama. Pri tome, ako ima više takvih k -grama, biramo proizvoljan jedan. Na ovaj način možemo doći do rešenja, ali je veoma skupo. Pri tome, velika je šansa da ćemo se negde zaglaviti (tj. nijedan od preostalih k -grama neće biti kandidat za nadovezivanje na tekuću nisku) ili zbog izbora početnog k -grama ili zbog izbora nekog od preostalih k -grama kada je postojalo više odgovarajućih. Sledeći primer ilustruje ovaj problem.

Neka nam je dat sledeći 3-gramska sastav: AAT ATG ATG ATG CAT CCA GAT GCC GGA GGG GTT TAA TGC TGG TGT. Treba rekonstruisati nisku koja ima takav sastav. Biramo početni 3-gram, neka to bude na primer TAA. Zatim na njega treba nadovezati 3-gram koji počinje njegovim sufiksom dužine 2, odnosno onaj 3-gram koji ima prefiks AA. U našem slučaju, postoji jedan takav 3-gram i njega nadovezujemo na tekuću nisku, tako da sada imamo TAAT. Zatim biramo 3-gram čiji je prefiks AT. Ovog puta imamo 3 kandidata, ali, na našu sreću, sva tri su isti 3-grami, ATG. U takvom slučaju nije bitno koji smo odabrali, jer su svi jednakci. Nadovezujemo ga na tekuću nisku i dobijamo TAATG. Tražimo 3-grame sa prefiksom TG, koji do sad nisu upotrebljeni. Ponovo pronalazimo 3 kandidata. Međutim, u ovom slučaju, svi kandidati predstavljaju različite 3-grame, a to su TGC, TGG i TGT. Naivni pristup kaže da biramo jedan od njih, i recimo da smo odabrali TGT i dobili nisku TAATGT. Sada nam je potreban 3-gram sa prefiksom GT i tu dolazi do zaglavljivanja. Imamo još 3-grama koji nisu iskorišćeni za rekonstrukciju niske, ali nijedan ne možemo da iskoristimo u ovom trenutku. U takvim situacijama treba se vratiti u nazad do koraka u kom je bilo više kandidata.

3.4 Rekonstrukcija niske kao problem Hamiltonove putanje

Videli smo da nam naivni pristup ne odgovara i moramo smisliti bolje rešenje. Mogli bismo da iskoristimo znanja iz teorije grafova za rešavanje ovakvog problema. U tom slučaju, prvi zadatak je da našu nisku predstavimo u vidu grafa.

3.4.1 Genom kao putanja

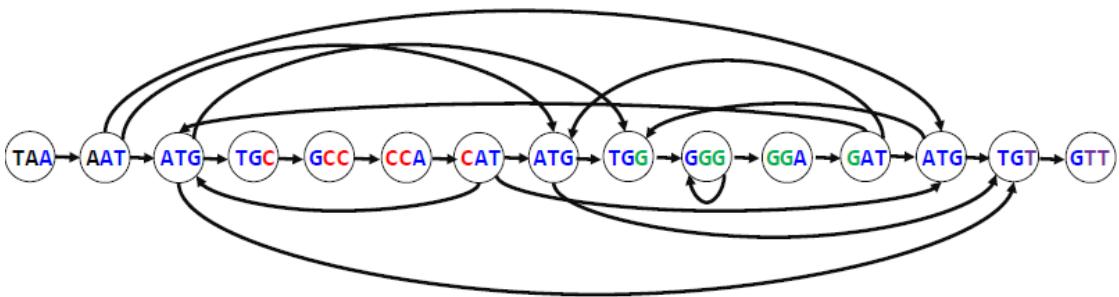
Vratimo se na prethodni primer. Dat nam je naredni 3-gramska sastav:

```

1 Composition_3(TAATGCCATGGGATGTT) =
2           TAA
3           AAT
4           ATG
5           TGC
6           GCC
7           CCA
8           CAT
9           ATG
10          TGG
11          GGG
12          GGA
13          GAT
14          ATG
15          TGT
16          GTT

```

Ovakav 3-gramska sastav možemo predstaviti kao graf na sledeći način. Svakom čvoru u grafu odgovara jedan od k -grama. Zatim, potrebne su nam grane koje će povezati te čvorove. Dva čvora su povezana usmerenom granom ako izlazni čvor ima sufiks koji je jednak prefektu ulaznog čvora te grane, kao što je prikazano na slici 3.5.



Slika 3.5: Graf koji odgovara 3-gramskom sastavu niske $TAATGCCATGGGATGTT$.

Jasno je da postoji više puteva u ovom grafu. Postavlja se pitanje – da li možemo da pronađemo genomsku putanju u ovom grafu, od svih koje postoje?

Podsetimo se šta je Hamiltonova putanja. Hamiltonova putanja je putanja koja posećuje svaki čvor u grafu tačno jednom. To je upravo ono što nam je potrebno za rešavanje problema. Svaki čvor predstavlja jedan k -gram i potrebno nam je da svi k -grami budu uključeni u rekonstruisanu nisku tačno jednom.

Problem Hamiltonove putanje: Naći Hamiltonovu putanju u grafu.

Ulaz: Graf.

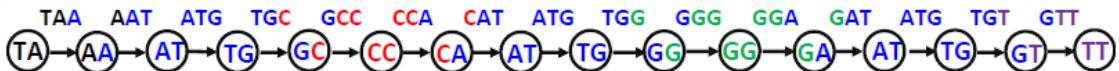
Izlaz: Putanja koja posećuje svaki čvor u grafu tačno jednom.

Iako deluje kao da smo rešili sve probleme, zapravo smo naišli na još jednu veliku prepreku. Naime, pronalaženje Hamiltonovog puta u grafu je NP-kompletan problem, što znači da ne postoji algoritam koji ga rešava u polinomijalnom vremenu. U tom slučaju, moramo da se vratimo na početak, a to je predstavljanje k -gramskog sastava grafom.

3.5 Rekonstrukcija niske kao Ojlerove putanje

U prethodnoj sekciji, k -grame smo predstavili čvorovima u grafu i u njemu tražili Hamiltonov put, odnosno, put koji obilazi svaki čvor tačno jednom. Videli smo da za taj problem još uvek nije poznat efikasan algoritam pa se sada pitamo kako možemo izmeniti graf tako da ne zahteva traženje Hamiltonove putanje.

Ono što se javlja kao ideja jeste obeležavanje grana umesto čvorova. Dakle, svaka grana biće obeležena jednim k -gramom. Izlazni čvor biće obeležen prefiksom k -grama te grane, dok će ulazni čvor biti obeležen sufiksom istog tog k -grama. Slika 3.6 ilustruje ovaj postupak za nisku *TAATGCCATGGGATGTT*.



Slika 3.6: Grafički prikaz 3-gramskog sastava niske *TAATGCCATGGGATGTT*. Grane su obeležene 3-gramima, a čvorovi 2-gramima koji predstavljaju prefikse i sufikse.

Primećujemo da su neki čvorovi obeleženi identično (na primer, imamo tri čvora sa oznakom *AT*). Sve čvorove koji imaju istu oznaku treba spojiti u jedan, pri čemu zadržavamo sve grane koje su ulazile u taj čvor ili su izlazile iz njega. Ponavljamo postupak dokle god imamo čvorove koji imaju istu oznaku i na kraju dobijamo graf koji nazivamo *De Brojnov graf*. Slika 3.7 ilustruje De Brojnov graf dobijen ovom procedurom od polaznog grafa sa slike 3.6.

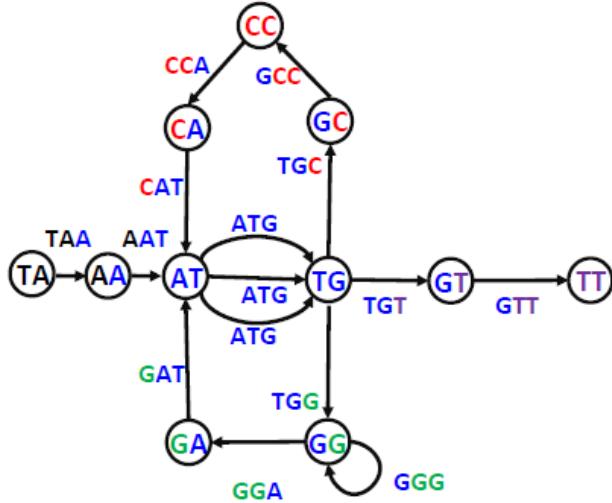
Ovime smo dobili novu reprezentaciju niske pomoću grafa. Prirodno se postavlja naredno pitanje – gde se nalazi niska *Genome* u ovoj reprezentaciji grafa? Kako nam se 3-grami sada nalaze na granama, a ne u čvorovima, potrebno je da pronađemo putanju u grafu koja prolazi sve grane tačno jednom. Takav put nazivamo *Ojlerova putanja*. Srećom, algoritam za pronalaženje Ojlerove putanje u grafu nije NP-kompletan i možemo efikasno da je pronađemo.

Problem Ojlerove putanje: Pronaći Ojlerovu putanju u grafu.

Ulaz: Graf.

Izlaz: Putanja koja posećuje svaku granu u grafu tačno jednom.

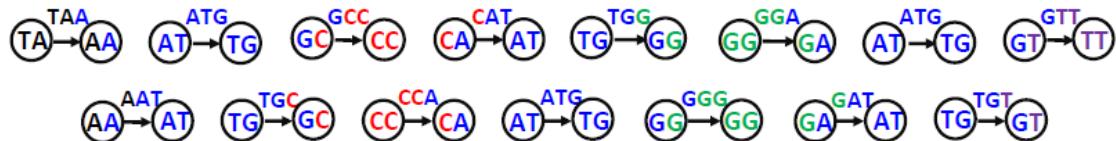
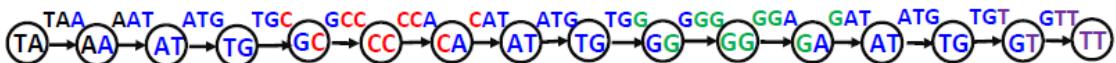
Sada znamo kako možemo da dobijemo nisku kada znamo De Brojnov graf koji odgovara njenom k -gramskom sastavu. Međutim, konstruisali smo De Brojnov graf na osnovu genoma, ali u realnim primenama, genom je nepoznat.

Slika 3.7: De Brojnov graf koji odgovara niski *TAATGCCATGGGATGTT*.

3.6 De Brojnovi grafovi na osnovu kolekcije k -grama

Videli smo kako možemo od zadate niske pronaći De Brojnov graf. Nažalost, u primenama nije nam poznata niska, ali znamo njen k -gramski sastav. Postavlja se pitanje kako možemo konstruisati De Brojnov graf od k -gramskog sastava niske.

Za svaki k -gram pravimo dva čvora i jednu granu – oznaka grane je upravo taj k -gram, oznaka izlaznog čvora je prefiks, a ulaznog čvora je sufiks datog k -grama. Time dobijamo nepovezani graf kao na slici 3.8. Zatim lepimo identične čvorove sve dok ne dobijemo graf čiji svi čvorovi imaju različite oznake. Na slici 3.9 dat je jedan korak ovog postupka, međutim, tu nije kraj jer i dalje postoje čvorovi sa istim oznakama.

Slika 3.8: Svaki k -gram prestavljen je pomoću dva čvora i jedne grane.

Slika 3.9: Prvi korak u postupku lepljenja čvorova. Napomenimo da postupak nije završen jer treba zlepiti čvorove sa identičnim oznakama (na primer, AT).

Po završetku postupka dobijamo de Brojnov graf koji je isti kao onaj koji smo dobili kada smo znali nisku, odnosno, graf na slici 3.7. Svaka grana je označena jednim k -gramom, a svaki čvor je označen prefiksom, odnosno, sufiksom odgovarajuće izlazne, odnosno, ulazne grane, redom. Naravno, čvorovi koji imaju identične oznake su zlepili.

3.7 Ojlerova teorema

Za rešavanje problema Ojlerove putanje koji smo predstavili u prethodnoj sekciji možemo iskoristiti rešenje narednog problema. Ovaj problem je značajan jer postoji teorema koja ga prati, a koja određuje uslove za njegovo rešavanje.

Problem Ojlerovog ciklusa: Pronaći Ojlerov ciklus u grafu.

Ulaz: Graf.

Izlaz: Ciklus koji posećuje svaku granu u grafu tačno jednom.

Kažemo da je graf Ojlerov ako sadrži Ojlerov ciklus. Ispostavlja se da postoje određene karakteristike koje određuju da li je graf Ojlerov. Uvedimo pojmove povezan graf i balansiran graf. Kažemo da je graf *povezan* ako za ma koja dva čvora postoji putanja koja ih povezuje. Graf je *balansiran* ako za svaki čvor važi da mu je izlazni stepen jednak ulaznom. Naredna teorema govori o potrebnim i dovoljnim uslovima da graf bude Ojlerov.

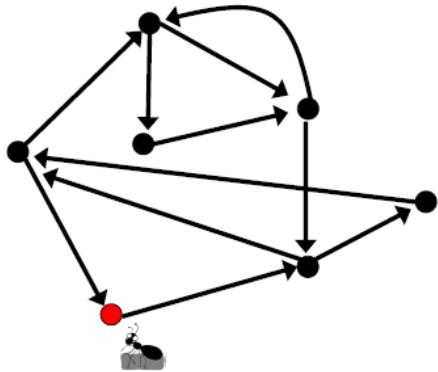
Teorema 3.1 (Ojlerova teorema). *Svaki Ojlerov graf je balansiran. Svaki povezan graf i balansiran graf je Ojlerov.*

3.7.1 Dokaz Ojlerove teoreme

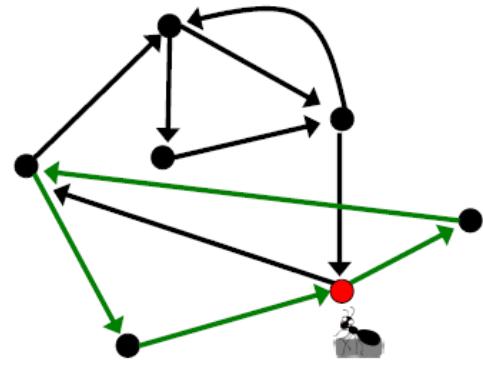
Neka nam je dat povezan balansiran graf. Da bismo pokazali da graf sadrži Ojlerov ciklus, postavićemo mrava na bilo koji od čvorova tog grafa, kao na slici 3.10. Zašto baš mrav? Poznato je da mravi nikada ne idu istim putem dva puta pa smo sigurni da će naš mrav proći svaku granu tačno jednom.

Puštamo mrava da slučajno odabira grane kojima će se kretati. Ako je veoma pametan, običiće svaku granu jednom i vratiće se u početni čvor. Međutim, velike su šanse da nije veoma pametan i da će se u nekom čvoru zaglaviti, odnosno, neće imati granu koju već nije obišao.

Da li mrav može da se zaglavi u bilo kom čvoru? Ispostavlja se da može da se zaglavi samo u početnom čvoru (jer je graf balansiran). U trenutku kada se zaglavio on je napravio ciklus. Samo, taj ciklus nije Ojlerov jer još uvek nije obišao sve grane. Ideja je da odabere drugačiji početni čvor iz kog će krenuti obilazak. Koji čvor će izabrati? Treba da izabere čvor iz ciklusa koji ima izlaznih grana koje još uvek nije obišao (slika 3.11).



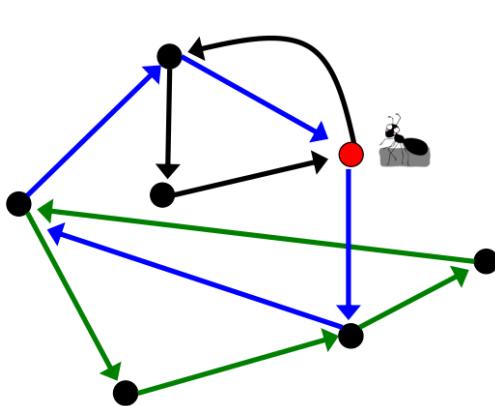
Slika 3.10: Mrav je postavljen u crveni čvor i odatle kreće obilazak povezanog balansiranog grafa.



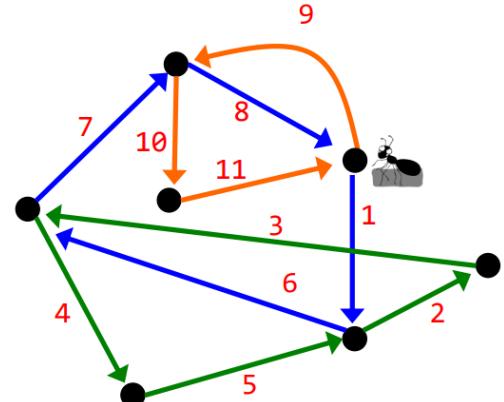
Slika 3.11: Mrav se zaglavio i pokušava ponovo iz drugog čvora koji pripada ciklusu i ima izlazne grane koje nisu posećene.

Sada, mraavlje pokušava ispočetka iz novog čvora. Prvo obilazi ciklus koji je već pronašao u prethodnom pokušaju, a zatim nastavlja obilazak preko neposećenih grana. Na taj način, ciklus se uvećava dok se ne dođe do Ojlerovog. Ukoliko se ponovo zaglavlji (slika 3.12) ponovo bira novi početni čvor, obilazi pronađeni ciklus (koji i dalje nije Ojlerov) i tako sve dok ne uspe da obide sve grane (slika 3.13).

Dokaz Ojlerove teoreme daje primer konstruktivnog dokaza, koji ne dokazuje samo željeni rezultat, već pruža metod za konstrukciju onoga što nam je potrebno. Ukratko, pratili smo kretanje mrava dok nije pronašao Ojlerov ciklus u povezanom balansiranom grafu, što je sumirano u algoritmu `EulerianCycle`.



Slika 3.12: Mrav se ponovo zaglavio i pokušava od novog čvora.



Slika 3.13: Mrav je konačno uspeo da pronađe dobar početni čvor i Ojlerov ciklus. Grane su obeležene redosledom kojim su posećene.

```

1 EulerianCycle(BalancedGraph)
2 begin
3   form a Cycle by randomly walking in BalancedGraph (avoiding already visited
   ↪ edges)
4   while Cycle is not Eulerian
5     select a node newStart in Cycle with still unexplored outgoing edges
6     form a Cycle' by traversing Cycle from newStart and randomly walking
   ↪ afterwards
7     Cycle ← Cycle'
8   return Cycle
9 end

```

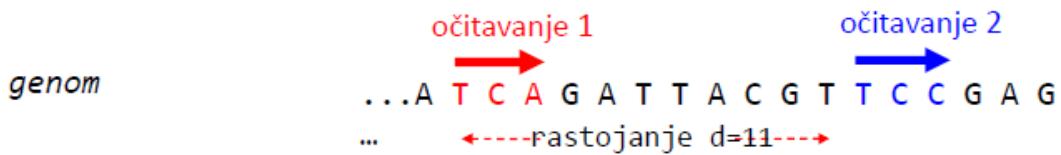
Ovaj algoritam radi u linearном vremenu. Da bi se zaista postigla ta efikasnost, potrebne su efikasne strukture podataka za održavanje ciklusa koje mrav pronalazi kao i za liste neiskorišćenih grana za svaki čvor i lista čvorova u trenutnom ciklusu koji imaju neiskorišćene grane.

3.8 Sastavljanje parova očitavanja

Predstavimo kao da su svi naši problemi rešeni. Međutim, može se javiti više Ojlerovih putanja u grafu. Srećom, i za ovo imamo jednostavno rešenje.

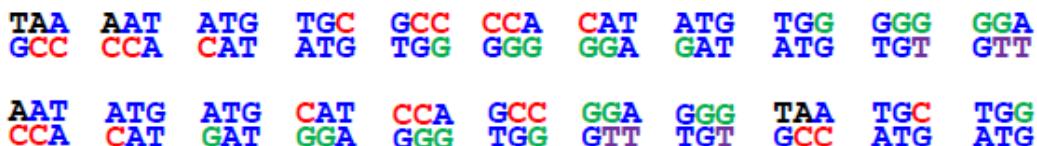
3.8.1 DNK sekvenciranje sa parovima očitavanja

Imamo više identičnih kopija genoma i na slučajnim pozicijama sećemo genom na fragmente iste dužine *InsertLength*. Zatim generišemo *parove očitavanja* – dva očitavanja sa krajeva svakog fragmenta na jednakoj, fiksiranoj udaljenosti. Pod *uparenim k -gramom* podrazumevamo par k -grama na fiksiranom rastojanju d u genomu. *Upareni k -gramske sastav*, u označi $\text{PairedComposition}_k(\text{Text})$, sastoji se od svih k -grama niske Text i njihovih parova.



Slika 3.14: TCA i TCC na rastojanju $d = 11$ čine jedan upareni 3-gram.

Dajmo jedan primer. Neka imamo nisku TAATGCCATGGGATGTT, i upareni 3-gram TAA i GCC. Upareni k -gramske sastav date niske prikazan je na slici 3.15.



Slika 3.15: *PairedComposition* niske TAATGCCATGGATGTT i njegov leksikografski poredak.

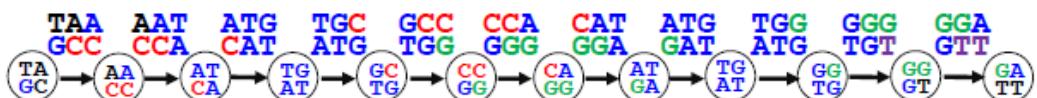
Sada možemo formulisati naredni problem.

Problem rekonstrukcije niske na osnovu parova očitavanja: Rekonstruisati nisku na osnovu njenih uparenih k -grama.

Ulaz: Kolekcija uparenih k -grama.

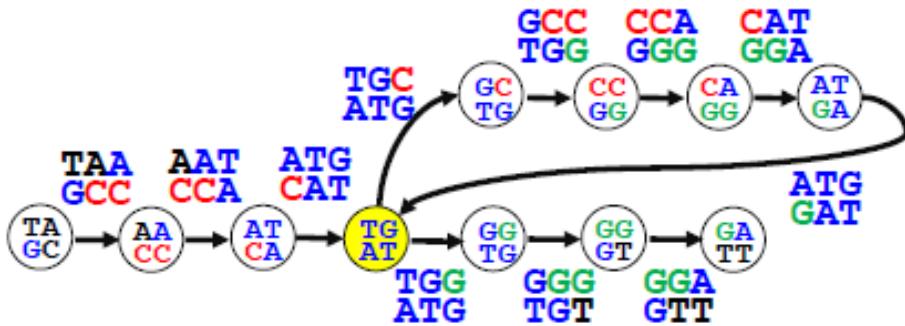
Izlaz: Niska *Text* takva da je $\text{PairedComposition}_k(\text{Text})$ jednak kolekciji uparenih k -grama.

Kako konstruisati upareni De Brojnov graf na osnovu uparenog k -gramskog sastava? Postupak je sličan prethodnom slučaju, kada nismo imali parove. Prepostavimo da je dat genom (niska *Genome*). Posmatrajmo genom kao putanju u grafu obeleženom na osnovu njegovog uparenog k -gramskog sastava (videti sliku 3.16). Svaka grana obeležena je uparenim k -gramom, a svaki čvor uparenim prefiksom, odnosno, sufiksom k -grama.



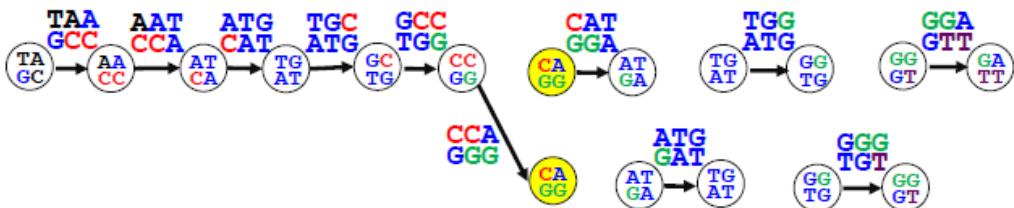
Slika 3.16: Graf koji odgovara uparenom 3-gramskom sastavu niske TAATGCCATGGATGTT.

Potrebito je zlepiti čvorove sa istom oznakom, tako da svi čvorovi budu jedinstveno obeleženi. Postupak je identičan prethodnom slučaju, odnosno, kada nismo imali parove. Primetimo da sada imamo mnogo manje lepljenja jer imamo samo dva čvora sa istom oznakom (**TG AT**), (videti sliku 3.17).



Slika 3.17: Dva čvora sa oznakom (TG AT) spajaju se u jedan pri čemu su sve grane, incidentne sa tim čvorovima, očuvane.

Kao i u prethodnom slučaju, pretpostavili smo da je dat genom (niska *Genome*), što često nije slučaj. Posmatrali smo genom kao putanju u grafu obeleženom na osnovu njegovog uparenog k -gramskog sastava. Sada pretpostavimo da nije dat genom već samo upareni k -gramske sastav. Za svaki upareni k -gram pravimo dva čvora i jednu granu, zatim lepimo identične čvorove (videti 3.18), i na kraju dobijamo upareni De Brojnov graf, kao onaj na slici 3.17.



Slika 3.18: Konstrukcija uparenog De Brojnovog grafa na osnovu uparenih k -grama.

Dakle, upareni De Brojnov graf, na osnovu kolekcije uparenih k -grama, dobijamo tako što svaku granu označavamo jednim uparenim k -gramom. Zatim, svaki čvor označavamo prefiksima, odnosno, sufiksima odgovarajuće izlazne, odnosno, ulazne grane, redom. Na kraju, lepimo čvorove sa identičnim oznakama.

3.9 U realnosti

Ovde smo imali neke nerealne pretpostavke:

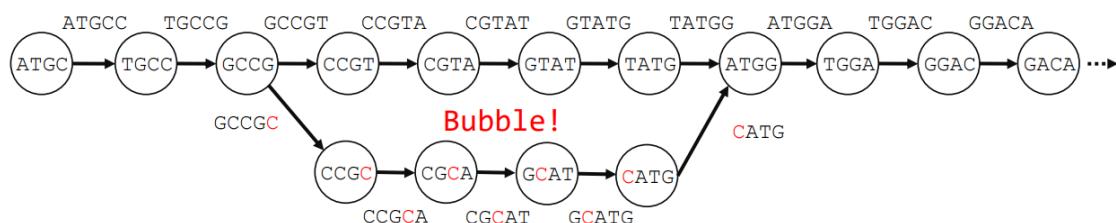
- Savršena pokrivenost genoma očitavanjima (svaki k -gram iz genoma je očitan). Očitavanja dužine 250 nukleotida dobijena Illumina tehnologijom predstavljaju samo mali deo 250-grama unutar genoma. Rešenje je u razbijanju dobijenih očitavanja na kraće k -grame (kao na slici 3.19).
- Očitavanja ne sadrže greške. U ovom slučaju, ako bismo razbili na manje k -grame, onda bismo dobili više niski koje imaju pogrešno očitavanje. Postavljamo pitanje kako se ovakvi

slučajevi manifestuju u konstrukciji DeBrojnovog grafa. Dolazi do stvaranja *balončića* (engl. *bubble*) u grafu (videti sliku 3.20). Jednostavan je slučaj kada govorimo o grešci na jednom očitavanju, međutim, ukoliko postoji više grešaka, onda dolazi do *eksplozije balončića* (videti sliku 3.21).

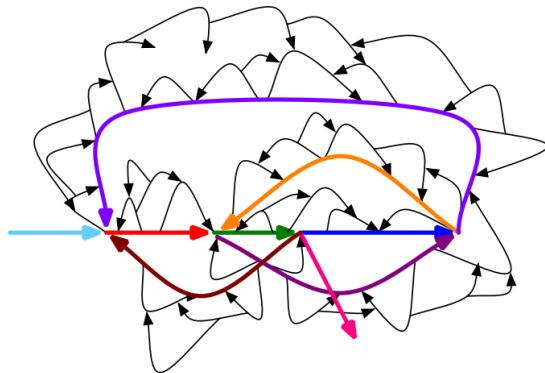
- Rastojanja između očitavanja u okviru parova očitavanja su egzaktna.
- itd.

atgcgttatggacaacgact atgcgtatg gccgtatgga gtatggacaa gacaacgact	atgcgttatggacaacgact atgcc tgccg gccgt ccgta cgtat gtatg tatgg atgga tggac ggaca gacaa acaac caacg aacga acgac cgact
---	--

Slika 3.19: Primer razbijanja dobijenih očitavanja na manje k-grame.



Slika 3.20: Primer pojave balončića u De Brojnovom grafu usled pojave greške u očitavanju nukleotida T nukleotidom C.



Slika 3.21: Eksplozija balončića.

3.10 Zadaci sa vežbi

U nastavku će biti predstavljeni zadaci sa vežbi na kursu rađeni u programskom jeziku Python.

3.10.1 Maximal Non Branching Path

```

1 from collections import deque
2 import copy
3
4 # Secenje DNK niske na k-mere
5 def string_to_k_mers(dna_string, k):
6     k_mers = []
7
8     for i in range(len(dna_string) - (k-1)):
9         k_mer = dna_string[i:i+k]
10        k_mers.append(k_mer)
11
12    return k_mers
13
14 # Konstruisanje Debruijn grafa od k-mera
15 def debruijn_graph_from_k_mers(k_mers):
16     G = {}
17
18     for k_mer in k_mers:
19         u = k_mer[:-1]
20         v = k_mer[1:]
21
22         if u in G:
23             if v not in G[u]:
24                 G[u].append(v)
25         else:
26             G[u] = [v]
27
28         if v not in G:
29             G[v] = []
30
31     return G

```

```

32
33
34 # Izracunavanje ulaznog i izlaznog stepena za zadati cvor
35 def degree(G, v):
36     out_deg = len(G[v])
37     in_deg = 0
38
39     for u in G:
40         if v in G[u]:
41             in_deg += 1
42
43     return (in_deg, out_deg)
44
45 # Pronalazenje izolovanih 1 in 1 out ciklusa u grafu polazeci od zadatog cvora
46 def isolated_cycle(G, v):
47     cycle = []
48
49     (in_deg, out_deg) = degree(G, v)
50
51     while in_deg == 1 and out_deg == 1:
52         u = G[v][0]
53         cycle.append((v,u))
54         if cycle[0][0] == cycle[-1][1]:
55             return cycle
56
57         v = u
58         (in_deg, out_deg) = degree(G, v)
59
60     return None
61
62
63 # Pronalazenje maksimalnih nerazgranatih putanja u grafu
64 def maximal_non_branching_paths(G):
65     paths = []
66     visited = {}
67
68     for v in G:
69
70         (v_in_deg, v_out_deg) = degree(G, v)
71         if v_in_deg != 1 or v_out_deg != 1:
72
73             visited[v] = True
74
75             if v_out_deg > 0:
76
77                 for w in G[v]:
78                     non_branching_path = [(v,w)]
79
80                     visited[w] = True
81                     (w_in_deg, w_out_deg) = degree(G, w)
82
83                     while w_in_deg == 1 and w_out_deg == 1:
84                         u = G[w][0]

```

```

85                     non_branching_path.append((w,u))
86                     w = u
87                     visited[w] = True
88                     (w_in_deg, w_out_deg) = degree(G, w)
89
90             paths.append(non_branching_path)
91
92     for v in G:
93         if v not in visited:
94             c = isolated_cycle(G, v)
95             if c != None:
96                 paths.append(c)
97
98     return paths
99
100
101 # Konstruisanje DNK niske od dobijene putanje
102 def create_string_from_path(path):
103
104     dna_string = path[0][0]
105
106     for i in range(len(path)):
107         dna_string += path[i][1][-1]
108
109     return dna_string
110
111
112 def main():
113     dna_string = "AATCGTGACCTCAACT"
114     #           TCGTGAC
115     #           AATC
116     #           ACT
117     #           ACCT
118     #           AAC
119     #           TCAAC
120     k = 3
121     k_mers = string_to_k_mers(dna_string, k)
122     g = debruijn_graph_from_k_mers(k_mers)
123     paths = maximal_non_branching_paths(g)
124
125     print(paths)
126
127 if __name__ == "__main__":
128     main()

```

3.10.2 All Euler Cycles

```

1 from collections import deque
2 import copy
3
4 # Izracunavanje ulaznog i izlaznog stepena za zadati cvor
5 def degree(G, v):
6     out_deg = len(G[v])

```

```

7     in_deg = 0
8
9     for u in G:
10        if v in G[u]:
11            in_deg += 1
12
13     return (in_deg, out_deg)
14
15 # Pronalazenje izolovanih 1 in 1 out ciklusa u grafu polazeci od zadatog cvora
16 def isolated_cycle(G, v):
17     cycle = []
18
19     (in_deg, out_deg) = degree(G, v)
20
21     while in_deg == 1 and out_deg == 1:
22         u = G[v][0]
23         cycle.append((v,u))
24         if cycle[0][0] == cycle[-1][1]:
25             return cycle
26
27         v = u
28         (in_deg, out_deg) = degree(G, v)
29
30     return None
31
32 # Konstruisanje DNK niske od dobijene putanje
33 def create_string_from_path(path):
34
35     dna_string = path[0][0].replace(">", ",")
36
37     for i in range(len(path)):
38         dna_string += path[i][1].replace(">", ",")[-1]
39
40     return dna_string
41
42
43 # Pronalazenje cvorova od kojih postoje grane ka zadatom cvoru v
44 def incoming(G, v):
45     in_list = []
46
47     for u in G:
48         if v in G[u]:
49             in_list.append(u)
50
51     return in_list
52
53 # Pronalazenje cvorova do kojih postoje grane od zadatog cvora v
54 def outgoing(G, v):
55     return G[v]
56
57
58 # Pravljenje (u,v,w) "zaobilaznice" u zadatom grafu G
59 def bypass(G, u, v, w):

```

```

60     G_p = copy.deepcopy(G)
61     G_p[u].remove(v)
62     G_p[v].remove(w)
63     G_p[u].append(v+">") #v'
64     G_p[v+">"] = [w]
65     return G_p
66
67
68 def DFS(G, v, visited):
69     visited[v] = True
70
71     for w in G[v]:
72         if w not in visited:
73             DFS(G, w, visited)
74
75
76 # Provera da li je graf povezan u odnosu na DFS obilazak iz zadatog cvora
77 def is_connected(G):
78
79     visited = {};
80     for v in G:
81         DFS(G,v,visited)
82         break;
83
84     for v in G:
85         if v not in visited:
86             return False
87
88     return True
89
90 # Pronalazenje svih Ojlerovih ciklusa u zadatom grafu G
91 def all_eulerian_cycles(G):
92     all_graphs = deque([copy.deepcopy(G)])
93     cycles = []
94
95     while len(all_graphs) > 0:
96         G_p = all_graphs.popleft()
97         v_p = None
98         for v in G_p:
99             (in_deg, out_deg) = degree(G_p, v)
100
101            if in_deg > 1:
102                v_p = v
103                break
104
105            if v_p != None:
106                for u in incoming(G_p, v_p):
107                    for w in outgoing(G_p, v_p):
108                        new_graph = bypass(G_p, u, v, w)
109                        if is_connected(new_graph):
110                            all_graphs.append(copy.deepcopy(new_graph))
111            else:
112                for k in G_p:

```

```

113         cycle = isolated_cycle(G_p, k)
114         if cycle != None:
115             path = create_string_from_path(cycle)
116             if path not in cycles:
117                 cycles.append(path);
118
119     return cycles
120
121
122
123
124 def main():
125     G = {'AT' : ['TC'], 'TC' : ['CG'], 'CG': ['GA', 'GG'], 'GA':['AT', 'AC'],
126          'TG': ['CG'], 'GG': ['GA']}
127     print(all_eulerian_cycles(G))
128
129 if __name__ == "__main__":
130     main()

```

3.10.3 String Spelled By Gapped Patterns

```

1 # Sastavljanje DNK niske pomocu k-mera
2 def string_spelled_by_patterns(patterns, k):
3     dna_string = patterns[0] [-1]
4
5     for i in range(0, len(patterns)):
6         dna_string += patterns[i] [-1]
7
8     return dna_string
9
10 # Sastavljanje DNK niske pomocu parova k-mera na udaljenosti d
11 def string_spelled_by_gapped_patterns(gapped_patterns, k, d):
12     first_patterns = [s[0] for s in gapped_patterns]
13     second_patterns = [s[1] for s in gapped_patterns]
14
15     prefix_string = string_spelled_by_patterns(first_patterns, k)
16     suffix_string = string_spelled_by_patterns(second_patterns, k)
17
18     print(prefix_string)
19     print(suffix_string)
20
21     for i in range(k+d, len(prefix_string)):
22         if prefix_string[i] != suffix_string[i-k-d]:
23             print('There is no string spelled by the gapped patterns')
24             return ''
25     return prefix_string + suffix_string[-k-d:]
26
27
28 def main():
29     gapped_patterns = [('CTG', 'CTG'), ('TGA', 'TGA'), ('GAC', 'GAC'), ('ACT', 'ACT')]
30
31     print(string_spelled_by_gapped_patterns(gapped_patterns, 3, 1))

```

```
32
33 if __name__ == "__main__":
34     main()
```


Glava 4

Kako sekvenciramo antibiotike?

U ovom poglavlju i dalje govorimo o sekvenciranju, ali ćemo proširiti pogled i pokazati različite načine za sekvenciranje peptida.

4.1 Otkriće antibiotika

Pre svega, krenućemo sa biološkim uvodom. Šta su to antibiotici? Sama reč antibiotik znači „onaj koji ubija život”, a tačnije, on predstavlja supstancu koja ubija bakterije. Kada ostavimo pomorandžu dugo negde gde je toplo, ona će da razvije čudne osobine kao što je buđ. Šta to znači? Bud jest jedna vrsta antibiotika što znači da se antibiotici nalaze u prirodi i da ih proizvode organizmi iz porodice gljiva (npr. buđi) i bakterija.

Mi ćemo posmatrati antibiotike na molekularnom nivou koji nam govori od čega su oni zapravo izgrađeni. Od svih antibiotika posmatraćemo **tirocidin B1**, antibiotik koji proizvodi bakterija *Bacillus Brevis*. Tirocidin B1 na molekulskom nivou pripada *peptidima*, kratkim niskama aminokiselina, odnosno malim proteinima. Ovo je skok u odnosu na ono što smo do sada posmatrali – nukleotidne niske nad četverostrukom azbukom $\Sigma = \{A, C, G, T\}$, odnosno DNK. Za DNK smo govorili da se pojavljuje u svakoj ćeliji svakog živog bića i da je veoma značajna supstanca jer sadrži recept (tačnije, nosi informaciju) za pravilno funkcionisanje i razvoj svakog živog bića. Da bi se svako živo biće pravilno razvijalo, neophodno je da njegove ćelije proizvode (sintetišu) u tačno određeno vreme određene supstance koje se nazivaju *proteini*. DNK nosi informaciju o tome kako treba neki protein da izgleda, od čega treba da se sastoji. Zašto je to bitno? Na primer, kada je dan, neke biljke treba da vrše fotosintezu, a za vršenje fotosinteze treba u samim ćelijama biljaka da se sintetišu određeni proteini.

Proteini su, nakon nukleinskih kiselina, druga značajna grupa molekula koja sa računarske tačke gledišta takođe predstavlja dugačke niske, ali ne nad azbukom od 4 karaktera, nego nad azbukom od 20 karaktera, a svaki karakter predstavlja molekul koji se naziva *aminokiselina*. Kao i nukleinske kiseline, aminokiseline se predstavljaju velikim latiničnim slovima $\{V, K, L, F, P, W, N, Q, Y, G, A, I, M, D, E, S, T, C, R, H\}$, a pored toga postoje i troslovne oznake $\{Val, Lys, Leu, Phe, Pro, Trp, Asn, Gln, Tyr, Gly, Ala, Ile, Met, Asp, Glu, Ser, Thr, Cys, Arg, His\}$. U prirodi postoji mnogo više od 20 aminokiselina, ali 20 njih najčešće učestvuje u sastav proteina. DNK upravlja time kada će nastati protein u okviru ćelije. Recept za nastajanje svakog proteina je zapisan u DNK. Kako je taj recept zapisan, videćemo u nastavku.

Proteini se još nazivaju i *polipeptidi*. Dužina proteina je obično od 100 aminokiselina do nekoliko hiljada (proteini su kraći od genomske sekvence). Tirocidin B1 je peptid jer se sastoji iz malog broja aminokiselina, svega deset – $V, K, L, F, P, W, F, N, Q, Y$. Problem sekvenciranja antibiotika jeste problem određivanja aminokiselina koje ulaze u sastav tog antibiotika. U prethodnom poglavlju smo sekvencirali genom, ali tehnike iz prethodnog poglavlja nećemo moći da koristimo u sekvenciranju tirocidina B1, što će biti objašnjeno u poglavlju [4.2](#).

4.2 Kako bakterije prave antibiotike?

Pre rešavanja problema sekvenciranja antibiotika, govorićemo o zanimljivoj i kompleksnoj temi, a to je tema – kako se prave proteini? Već je pomenuto da se u okviru DNK nalazi recept za pravljenje proteina. Sada je vreme da se zapitamo kako je sve to zapisano u DNK pomoću A, C, G, T .

Znamo da je DNK dvostruki lanac čiji su krajevi označeni sa 5' i 3' (uvek čitamo lanac od 5' ka 3'). DNK jeste jedna vrsta nukleinskih kiselina koje postoji u ćeliji živih bića. Pored nje, postoje i različite vrste **ribonukleinskih kiselinina**, odnosno **RNK**. Ribonukleinske kiseline nisu predstavljene dvostrukim lancem, već jednostrukim. One se sastoje od nukleotida A, C, G, U . Umesto timina, kod RNK se pojavljuje nukleotid uracil koji se označava sa U .

DNK se **prepisuje** u RNK. Šta to znači? Da bi nastali proteini, neophodno je da se na osnovu dva lanca od DNK konstruiše RNK molekul. Pošto se RNK molekul sastoji od istih nukleotida kao i DNK, osim timina, onda kažemo da formiranje RNK na osnovu DNK predstavlja jednostavno *prepisivanje* nukleotida iz oba lanca DNK, uz zamenu nukleotida T sa nukleotidom U . Drugi naziv za prepisivanje jeste *transkripcija*. Ovo je prvi korak, i dalje nismo došli do aminokiseline, i dalje smo u abzuci nukleotida. RNK predstavlja jedan međukorak između DNK i samog proteina.

Drugi korak jeste *prevodenje*, odnosno *translacija*, prepisanog RNK u proteine. Imamo 4 nukleotida A, C, G, U i treba njih da prevedemo u nisku od 20 mogućih aminokiselina. To znači da mora da postoji neko preslikavanje, nekakav kod koji prevodi neke k -grame nukleotida u aminokiseline. Nad abzukom od 4 nukleotida postoji 16 različitih 2-grama, tj. bigrama. Da li možemo tih 16 bigrama da preslikamo u 20 aminokiselina? Tačnije, da li dva nukleotida možemo da preslikamo u jednu aminokiselinu? Ne možemo, jer moramo za svaku aminokiselinu da znamo koji je bigram označava. Pošto ne možemo to da uradimo sa bigramima, da li možemo sa 3-gramima? Svih mogućih 3-grama nad abzukom od 4 nukleotida ima 64. To znači da će svaka od aminokiselina imati svoj kod, a neke od njih će možda imati i više kodova, tj. više trigrami može da ukazuju na jednu aminokiselinu. To je u redu, bitno je da je naša funkcija „na”, ne mora da bude „1 – 1”. Ali kako napraviti funkciju? Ne možemo svojevoljno da dodelimo trigramima određene aminokiseline. Ta funkcija je unapred utvrđena, odnosno, prirodnom determinisana i dokazana. U nastavku, koristićemo drugaćiji naziv za naše 3-grame.

Definicija 4.1. *Kodon predstavlja jedan 3-gram (triplet) nukleotida.*

Preslikavanje o kojem je do sada bilo reči se naziva *genetski kod* i on je prikazan na slici 4.1.

Definicija 4.2. *Genetski kod predstavlja preslikavanje skupa kodona u skup aminokiselina.*

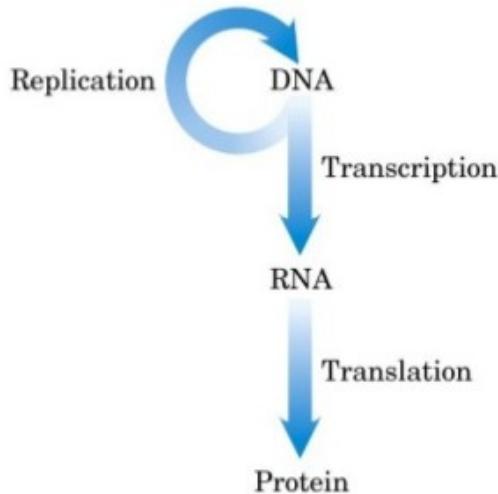
Vidimo da se, na primer, kodon UGC preslikava u aminokiselinu $Trp(W)$, dok se više kodona, $CUA, CUC, CUG, CUU, UUA, UUG$, preslikava u jednu aminokiselinu $Lys(L)$.

Redom, kodone iz RNK preslikavamo u aminokiseline. Ali, kako znamo da je kraj nekog proteina? U genetskom kodu je i tako nešto kodirano. Postoje tzv. **stop kodoni** koji označavaju da iza njih nema više aminokiselina koje čine taj protein. Ti stop kodoni su UAA, UAG, UGA .

0	AAA	K	16	CAA	Q	32	GAA	E	48	UAA	*
1	AAC	N	17	CAC	H	33	GAC	D	49	UAC	Y
2	AAG	K	18	CAG	Q	34	GAG	E	50	UAG	*
3	AAU	N	19	CAU	H	35	GAU	D	51	UAU	Y
4	ACA	T	20	CCA	P	36	GCA	A	52	UCA	S
5	ACC	T	21	CCC	P	37	GCC	A	53	UCC	S
6	ACG	T	22	CCG	P	38	GCG	A	54	UCG	S
7	ACU	T	23	CCU	P	39	GCU	A	55	UCU	S
8	AGA	R	24	CGA	R	40	GGA	G	56	UGA	*
9	AGC	S	25	CGC	R	41	GGC	G	57	UGC	C
10	AGG	R	26	CGG	R	42	GGG	G	58	UGG	W
11	AGU	S	27	CGU	R	43	GGU	G	59	UGU	C
12	AUA	I	28	CUA	L	44	GUA	V	60	UUA	L
13	AUC	I	29	CUC	L	45	GUC	V	61	UUC	F
14	AUG	M	30	CUG	L	46	GUG	V	62	UUG	L
15	AUU	I	31	CUU	L	47	GUU	V	63	UUU	F

Slika 4.1: Genetski kod.

Dolazimo do jednog veoma značajnog biološkog aksioma – **centralne dogme molekularne biologije**. Ona govori da se transkripcijom na osnovu DNK može dobiti RNK, a translacijom se iz RNK, na osnovu genetskog koda, dobijaju proteini. Ovu teoriju je predstavio Fransis Krik (eng. *Francis Crick*) i prikazana je na slici 4.2.



Slika 4.2: Centralna dogma molekularne biologije.

Ono što želimo da saznamo jeste koje aminokiseline i kojim redom ulaze u sastav našeg malog peptida tirocidina B1. Pošto se on sastoji iz 10 aminokiselina, to znači da ga čine 30 nukleotida u genomu bakterije *Bacillus Brevis* koje će da se prepišu u RNK i da se prevedu iz RNK u tirocidin B1. Hiljade različitih 30-grama se može prevesti u tirocidin B1 jer se u genetskom kodu različiti kodoni mogu prevesti u istu aminokiselinu. Na slici 4.3 su prikazani neki od takvih 30-grama. Vidimo da oni nisu previše slični.

Treba uzeti u obzir da translacija može početi na bilo kojoj poziciji u genomu. To znači

GT**TAAATTATTTCCTTGTTTAA**TCAATAT

GT**CAAGCTTTCCTGGTCAA**CCAGTAC

GT**AAAACTATTCCGTGGTCAA**TCAATAT

Slika 4.3: Neki od 30-grama koji se mogu prevesti u tirocidin B1.

da bismo za datu poziciju, ako gledamo 30-gram, mogli da imamo 6 različitih tzv. **čitajućih okvira**, tj. 6 varijanti prepisivanja u RNK i onda prevodenja. Tri čitajuća okvira potiču iz tri nukleotida iz jednog kodona iz jednog prevedenog RNK lanca (ako krenemo da čitamo od prvog nukleotida, to je jedan čitajući okvir, iz drugog nukleotida je drugi čitajući okvir, iz trećeg nukleotida je treći čitajući okvir, a ako pročitamo od četvrtog nukleotida, to je već isti čitajući okvir kao prvi jer tu kreće novi kodon), a isto tako za drugi RNK lanac imamo tri čitajuća okvira sa druge strane.

Naš peptid tirocidin B1 jeste *cikličan*. Tih 10 aminokiselina koje ga čine idu nekim redom, ali su one povezane u krug, tako da imamo ukupno 10 različitih **linearnih reprezentacija** za tirocidin B1 u zavisnosti od toga koja nam je prva aminokiselina bila u samom receptu DNK. Koju god linearnu reprezentaciju pronađemo, rešili smo problem.

Ne odustajemo od pronalaženja 30-grama u genomu bakterije *Bacillus Brevis* koji kodira bar jednu linearnu reprezentaciju od svih 10 koje čine tirocidin B1. Prepostavimo da imamo na raspolaganju veoma moćan računar i neograničeno vreme. Doći ćemo do jednog čudnog rezultata. Nećemo uspeti da pronađemo nijedan 30-gram u genomu bakterije *Bacillus Brevis* koji kodira bar jednu linearnu reprezentaciju proteina tirocidina B1. Zašto? Stvari se komplikuju. Na ovom primeru je pokazano da centralna dogma ne važi uvek, odnosno ne važi da svaki protein u ćeliji nastaje na osnovu recepta koji je zapisan u DNK. Centralna dogma govori da se proces transkripcije izvršava pod uticajem enzima koji se zove *RNK polimeraza*, a translacija RNK u protein se vrši u ćelijskoj organeli koja se naziva *ribozom*. Postoje neki蛋白 koji ne nastaju na ovaj način, nego na specijalan način gde obično sekvenciranje genoma ne može da nam pomogne. Moramo da predložimo novi metod kako možemo da pronađemo odgovarajuću sekvencu aminokiselina.

Edvard Tejtum (eng. *Edward Tatum*), jedan od poznatih američkih genetičara, je 1963. godine inhibirao ribozom bakterije *Bacillus Brevis*. Šta ovo znači? S obzirom da se znalo da se u ribozomu vrši translacija RNK u protein, on je onemogućio da se bilo šta desi u ribozomu, isključio je funkcionisanje te organele u ćeliji i očekivao je da se neće stvoriti nijedan protein, pa ni tirocidin B1. Međutim, suprotno očekivanjima, nastavljena je proizvodnja nekih peptida, uključujući i tirocidine. Ovo je bilo izuzetno iznenađujuće otkriće.

Fric Lipman (eng. *Fritz Lipmann*), američko-nemački biohemičar, je 1969. godine pokazao da tirocidini spadaju u grupu **ne-ribozomalnih peptida (NRP-ova)**. To su peptidi za čiju sintezu nisu odgovorni ribozomi i RNK polimeraza već enzimi poznati pod nazivom **NRP sintetaze**, molekuli koji se takođe nalaze u ćeliji i utiču na različite procese koji se dešavaju u njoj. To znači da se stvaranje tirocidina razlikuje od većeg broja proteina u živim bićima.

Kako izgleda sinteza tirocidina B1 pomoću NRP sintetaze? Postoji veliki broj različitih NRP sintetaza, nije samo jedna odgovorna za stvaranje svih mogućih NRP-ova, nego za svaki ne-ribozomalni peptid postoji odgovarajuća NRP sintetaza. Ona NRP sintetaza koja je odgo-

vorna za stvaranje tirocidina B1 se sastoji od 10 različitih podjedinica koje nazivamo *moduli*. Svaki modul je odgovoran za nadovezivanje jedne aminokiseline na budući molekul tirocidina B1. Svaki od modula privuče jednu aminokiselnu i spoji je sa prethodnom, a poslednji korak jeste cirkularizacija – spajanje aminokiselina nastalih uz pomoć prvog i poslednjeg modula radi kreiranja cikličnog peptida.

4.3 Sekvenciranje antibiotika razbijanjem na komade

Pošto nam sekvenciranje genomske sekvene i pronađenje odgovarajuće podnische koja je zadužena za translaciju u aminokiseline odgovarajućeg peptida ne može pomoći u sekvenciranju tirocidina B1 (jer on ne nastaje na osnovu informacije zapisane u DNK), postavljamo pitanje da li postoji način na koji možemo da sekvenciramo antibiotike. Moramo direktno da sekvenciramo peptid. Jedan od načina jeste **sekvenciranje razbijanjem na komade** i biće predstavljen u ovoj sekciji.

U sekvenciranju antibiotika može nam pomoći mašina koja se naziva **maseni spektrometar** i koju možemo opisati kao skupu molekularnu vagu. Šta on radi? Za početak ćemo da se zapitamo kako možemo da merimo težinu, tačnije masu molekula.

Pošto se molekuli sastoje od atoma, prvo treba da govorimo o masi pojedinačnog atoma. U atomima postoje protoni, neutroni i elektroni. Protoni i neutroni su približno iste mase, dok su elektroni izuzetno mali i gotovo zanemarljive mase. Zato masu jednog atoma možemo svesti na masu protona, odnosno neutrona koji učestvuju u izgradnji konkretnog atoma koji posmatramo, pa se i masa molekula može izračunati kao suma masa atoma koji ga čine. Maseni spektrometar vraća masu molekula izračunatu u **Daltonima**.

$$\begin{aligned} 1 \text{ Dalton(Da)} &\approx \text{masa jednog protona/neutrona} \\ \text{Masa molekula} &\approx \text{suma masa protona/neutrona} \end{aligned}$$

Posmatrajmo masu jedne aminokiseline, recimo glicina. Glicin ima hemijsku formulu C_2H_3ON . Ugljenik ima masu 12, vodonik ima masu 1, kiseonik 16, a azot 14. Na osnovu ovih masa računamo masu celog molekula glicina.

$$\text{masa}(C_2H_3ON) = 12 * 2 + 1 * 3 + 16 * 1 + 14 * 1 \approx 57Da$$

Simbol \approx koristimo jer nam je stvarna masa nešto malo drugačija od celobrojne mase koju dobijamo ovde. Stvarna masa glicina iznosi 57.02Da. Podrazumevaćemo da je masa celog molekula upravo celobrojna masa koju smo dobili.

Tabela masa svih aminokiselina data je u tabeli 4.1.

Tabela 4.1: Tabela masa 20 aminokiselina poredanih rastuće prema celobrojnim masama.

G	A	S	P	V	T	C	I,L	N	D	K,Q	E	M	H	F	R	Y	W
57	71	87	97	99	101	103	113	114	115	128	129	131	137	147	156	163	186

Primećujemo da neke aminokiseline imaju iste mase, npr. I i L, K i Q, pa za 20 aminokiselina imamo 18 celobrojnih masa.

Kada imamo mase aminokiselina, možemo da se zapitamo koja je masa tirocidina B1. Znajući da se tirocidin B1 sastoji iz 10 aminokiselina ovim redom: VKLFPWFNQY, masu računamo koristeći tabelu 4.1.

$$\text{masa(tirocidina B1)} = 99 + 128 + 113 + 147 + 97 + 186 + 147 + 114 + 128 + 163 = 1322$$

Vratimo se na maseni spektrometar o kojem smo govorili ranije. Zamislimo da imamo kratak peptid koji se sastoji od samo 4 aminokiseline *NQEL*. Uzorak ovog peptida ubacimo u maseni spektrometar. Šta se u njemu dešava? U njemu se nekim hemijskim procesima, u čije detalje nećemo ulaziti, generišu svi podpeptidi ulaznog peptida. Sa računarske tačke gledišta, maseni spektrometar generiše od zadate niske *NQEL* sve moguće podniske, podrazumevajući da je ulazni peptid cikličan. Tako se generišu podniske dužine jedan: *N, Q, E, L*, podniske dužine dva: *NQ, QE, EL, LN* i podniske dužine tri: *NQE, QEL, ELN, LNQ*. Maseni spektrometar za svaki od dobijenih podpeptida može da odredi molekulsku masu. Ono što mi dobijemo kao izlaz iz masenog spektrometra nisu podpeptidi, mi ne znamo koje podpeptide je dobio, niti kojim podpeptidima je prudružena koja masa. Izlaz iz masenog spektrometra jeste samo niz masa! Taj izlazni niz može da sadrži i dva ista broja jer neki podpeptidi mogu da imaju istu masu.

Kako bismo formulisali računarski problem, moramo da definišemo šta je to *teorijski spektar*.

Definicija 4.3. *Teorijski spektar peptida predstavlja niz masa svih mogućih podpeptida tog peptida, uključujući nulu kao masu praznog peptida i masu samog peptida.*

Poznajući sastav peptida, lako možemo da izračunamo teorijski spektar. Suprotan smer je težak, odnosno teško je da na osnovu teorijskog spektra zaključimo kako je izgledao peptid. Upravo ovo jeste problem sekvenciranja ciklopeptida.

Problem 1 (Problem sekvenciranja ciklopeptida). *Rekonstruisati ciklični peptid na osnovu njegovog teorijskog spektra.*

4.3.1 Sekvenciranje ciklopeptida grubom silom

Kada dobijemo spektar iz masenog spektrometra, najveća masa će označavati masu celog peptida. Želimo prvo da generišemo sve peptide sa masom jednakom masi celog peptida, zatim da za svaki tako generisan peptid formiramo teorijski spektar i uporedimo ga sa datim spektrom. Algoritam grube sile za problem sekvenciranje ciklopeptida dat je u nastavku.

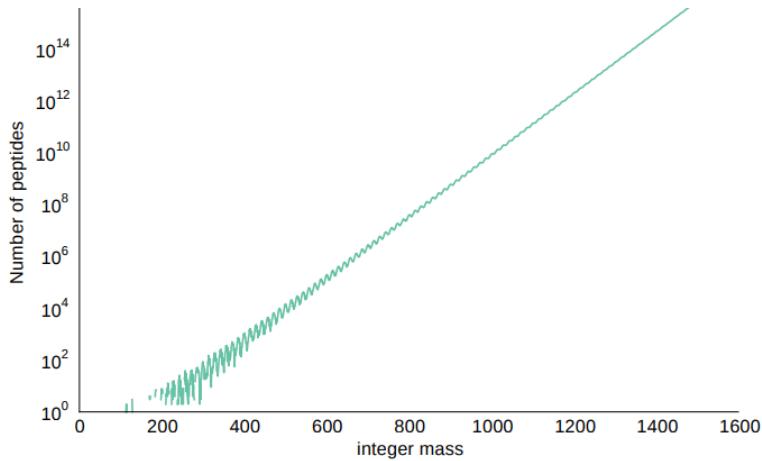
```

1 BFCyclopeptideSequencing(Spectrum)
2 begin
3     // ParentMass(Spectrum) jeste najveća masa u spektru Spectrum
4     for every Peptide with Mass(Peptide) equal to ParentMass(Spectrum)
5         if Spectrum == CycloSpectrum(Peptide)
6             output Peptide
7 end

```

Vidimo da u ovom algoritmu ispitujemo sve kandidate (peptide sa istom masom kao dati peptid). Koliko imamo takvih kandidata? Grafik koji oslikava odgovor na ovo pitanje dat je na slici 4.4.

Vidimo da je ovaj algoritam grube sile eksponencijalne složenosti. Pod uslovom da imamo dovoljno brz računar, ovako nešto bismo i mogli da izračunamo. Ali, šta bi bili nedostaci ovog algoritma grube sile? Možemo da imamo dva peptida sa istom masom, a da su potpuno različiti. Na primer, peptid *NQEL* i peptid *TMDH* imaju masu 484. Kako možemo da isključimo pogrešan peptid? Za oba ova peptida možemo da generišemo teorijski spektar. Ispostavlja se da su njihovi spektri potpuno različiti. Želimo da ovu informaciju iskoristimo u sledećem pristupu rešavanja problema sekvenciranja ciklopeptida. Cilj nam je da ne idemo grubom silom već da ogroman broj kandidata od samog početka odstranimo.



Slika 4.4: Broj peptida sa zadatom masom.

4.3.2 *Branch-and-Bound* algoritam za sekvenciranje ciklopeptida

U ovom pristupu postepeno konstruišemo kandidate za rešenja od manjih linearnih peptida za razliku od prethodnog pristupa u kome smo odmah generisali ceo peptid koji postaje kandidat. Na taj način ćemo smanjiti ukupan broj linearnih peptida koje posmatramo. Ovakav pristup se koristi kod ***Branch-and-Bound*** algoritama koji će biti opisani u nastavku.

Kod *Branch-and-Bound* algoritama u celokupnom prostoru svih mogućih rešenja vršimo nekakva odsecanja. Počnemo od kratkog peptida dužine jedan, pa ga proširimo na sve moguće načine, odnosno, dodamo po jednu aminokiselinu i od toga napravimo sve moguće kandidate dužine dva. To je *branch grana* i predstavljena je na slici 4.5. *Bound grana* bi od postojećih kandidata, nastalih u *branch* granama, isključila neke potencijalne kandidate. *Bound grana* je predstavljena na slici 4.6. Postupak proširivanja i odsecanja ponavljamo sve dok ne dođemo do odgovarajućih vrednosti. Na ovaj način će nam ostati mnogo manje kandidata za rešenja nego u prethodnom pristupu grube sile.

Slika 4.5: *Branch* grane algoritma *Branch-and-Bound*.Slika 4.6: *Bound* grane algoritma *Branch-and-Bound*.

Primenimo ovaj algoritam na konkretan problem. Recimo da nam je dat spektar

0 97 97 99 101 103 196 198 198 200 202 295 297 299 299 301 394 396 398 400 400 497.

Vidimo da se u datom spektru nalaze mase nekih aminokiselina, što nam govori koje aminokiseline ulaze u sastav traženog peptida. Te aminokiseline su *P, V, T, C* sa masama 97, 99, 101, 103. Ovo znači da možemo da počnemo ne sa svih 20 aminokiselina, nego sa 4 unigramama *P, V, T, C*. Ovo je unapred jedna *bound* grana jer smo 20 aminokiselina sveli na 4 kandidata aminokiselina. Zatim idemo na *branch* granu, širimo unigrame u sve moguće bigrame: *PA, PC, PD,..PY, VA, VC, VD,...,VY, TA, TC, TD,..., TY, CA, CC, CD,..., CY*. Proširujemo sa svih 20 aminokiselina jer ćemo kasniji videti da ovako zadati spektar jeste čisto teorijski spektar, a maseni spektrometar skoro nikada u praksi ne vraća teorijski spektar već spektar sa nekakvima greškama. Kako možemo

da skratimo ovu listu, kako možemo da izvršimo korak *bound* u ovom trenutku? Posmatramo da li postoje odgovarajući bigrami koji se takođe pojavljuju u spektru. Zbog toga uvodimo pojam **konzistentnosti**.

Definicija 4.4. Za proizvoljan podpeptid p_1, \dots, p_n kažemo da je **konzistentan** sa spektrom S ukoliko se svaka masa iz teorijskog spektra podpeptida p_1, \dots, p_n nalazi u spektru S .

Na primer, PV je **konzistentno** sa spektrom ukoliko se masa od P , masa od V i masa od PV nalaze u spektru.

Konzistentnost ćemo koristiti u *bound* koraku, tačnije, izbacićemo sve bigrame koji nisu konzistentni sa spektrom. Tako dobijemo listu konzistentnih bigrama PV , PT , PC , VP , VT , VC , TP , TV , CP , CV koju proširujemo u sve moguće 3-grame, a zatim svodimo na listu samo konzistentnih 3-grama. Postupak ponavljamo. Kada dođemo do liste konzistentnih pentagrama $PVCPT$, $PTPVC$, $PTPVC$, $PCVPT$, $VPTPC$, $VCPTP$, $TPVCP$, $TPCVP$, $CPTPV$, $CVPTP$ vidimo da zapravo svi oni pokazuju na jedan isti ciklični peptid.

Pseudokod opisanog algoritma dat je u nastavku.

```

1 CyclopeptideSequencing(Spectrum)
2 begin
3     Peptides = a set containing only the empty peptide
4     while Peptides is non-empty
5         // proširujemo sve peptide u skupu sa svim mogucim aminokiselinama
6         Peptides = Expand(Peptides)
7         for each Peptide in Peptides
8             // ParentMass(Spectrum) jeste najveca masa u spektru
9             if Mass(Peptide) = ParentMass(Spectrum)
10                if Cyclospectrum(Peptide) = Spectrum
11                    output Peptide
12                    remove Peptide from Peptides
13                else if Peptide is not consistent with Spectrum
14                    remove Peptide from Peptides
15    end

```

Podsetimo se da je složenost algoritma grube sile, koji ovde pokušavamo da poboljšamo, eksponencijalna. Ispostavlja se da *Branch-and-Bound* algoritam takođe može biti eksponencijalne složenosti za neke peptide, ali je u praksi veoma brz.

4.4 Prilagođavanje sekvenciranja za spektre sa greškama

Spektar koji smo do sada definisali jeste teorijski spektar. Za razliku od njega, spektar koji izlazi iz masenog spektrometra, **eksperimentalni spektar**, često sadrže greške. O kakvim greškama se govori biće prikazano pomoću slike 4.7.

teorijski:	0	113	114	128	129	227	242	242	257	355	356	370	371	484		
eksperimentalni:	0	99	113	114	128		227			257	299	355	356	370	371	484

Slika 4.7: Primer teorijskog i eksperimentalnog spektra za *NQEL*.

Lažne mase jesu mase koje su na slici 4.7 prikazane zelenom bojom. To su mase koje su prisutne u eksperimentalnom spektru, ali nisu prisutne u teorijskom spektru.

Nedostajuće mase jesu mase koje su na slici 4.7 prikazane plavom bojom. To su mase koje se nalaze u okviru teorijskog spektra, ali ne i u okviru eksperimentalnog spektra.

Zbog pojave ovih otežavajućih okolnosti, tj. grešaka u spektru, neophodan je novi algoritam jer se kod dva predložena algoritma teorijski spektar peptida morao u potpunosti poklapati sa spektrom peptida koji je predstavljao rešenje problema. Sada moramo da olabavimo taj uslov pa uvodimo pojam **skor peptida**.

Definicija 4.5. *Skor peptida pokazuje koliko masa njegov teorijski spektar deli sa eksperimentalnim spektrom.*

Tako, za sliku 4.7, skor iznosi 11. Želimo da skor bude što veći.

S obzirom da imamo nov način upoređivanja, moramo da unapredimo naš *Branch-and-Bound* algoritam, konkretno korak odsecanja.

Uzmimo primer golfa. U golfu, kada igrači prođu prvi krug takmičenja, u sledeći krug prolaze dalje samo igrači koji su konkurentni, oni koji imaju šanse da nešto osvoje. To znači da možemo da kažemo da nam je odsecanje takvo da, na primer, prva tri igrača sa najboljim skorom idu dalje, a ukoliko imamo još neke igrače koji imaju isti skor kao poslednji igrač, onda i oni prolaze dalje. Znači, zadržavaju se tri najbolja igrača „*with ties*”. Ovakav sistem primenjen na *Branch-and-Bound* algoritam prikazan je u sledećem pseudokodu.

```

1 LeaderboardCyclopeptideSequencing(Spectrum, N)
2 begin
3     Leaderboard = set containing only the empty peptide
4     LeaderPeptide = empty peptide
5
6     while Leaderboard is non-empty
7         // proširujemo sve elemente koji se nalaze u okviru skupa Leaderboard
8         Leaderboard = Expand(Leaderboard)
9         for each Peptide in Leaderboard
10            // ParentMass(Spectrum) predstavlja najveću masu u spektru Spectrum
11            if Mass(Peptide) == ParentMass(Spectrum)
12                if Score(Peptide, Spectrum) > Score(LeaderPeptide, Spectrum)
13                    LeaderPeptide = Peptide
14                else if Mass(Peptide) > ParentMass(Spectrum)
15                    remove Peptide from Leaderboard
16            // odsecamo kandidate iz Leaderboard na osnovu njihovog skora
17            Leaderboard = Trim(Leaderboard, Spectrum, N)
18
19        output LeaderPeptide
20    end
21
22 Trim(Leaderboard, Spectrum, N, AminoAcid, AminoAcidMass)
23 begin
24     for j=1 to |Leaderboard|
25         Peptide = j-th peptide in Leaderboard
26         // LinearScore jeste skor nad linearnim spektrom
27         LinearScores[j] = LinearScore(Peptide, Spectrum)
28
29     sort Leaderboard according to the dec order of scores in LinearScores
30     sort LinearScores in dec order
31
32     for j=N+1 to |Leaderboard|
33         if LinearScores[j] < LinearScores[N]
34             remove all peptides starting from the j-th peptide from Leaderboard
35
36     return Leaderboard

```

```
37     return Leaderboard
38 end
```

Leaderboard pristup omogućava da bolje definišemo za eksperimentalni spektar kod *Branch-and-Bound* algoritma onu bound fazu kada treba da izbacimo neke kandidate.

4.4.1 Testiranje na spektru tirocidina B1

U ovom delu razmatraćemo rezultate testiranja na $Spectrum_{10}$, spektru sa 10% lažnih/nedostajućih masa.

Kada primenimo *LeaderboardCyclopeptideSequencing* na spektar sa 10% loših vrednosti, tada zaista dobijemo peptid sa najvišim skorom *VKLFPWFNQY* koji odgovara tirocidinu B1. Međutim, ukoliko uzmemo spektar $Spectrum_{25}$ koji ima 25% lažnih i nedostajućih vrednosti, spektar koji se još više udaljava od teorijskog spektra, onda se peptid sa najvišim skorom *VKLFPADFNQY* razlikuje od peptida *VKLFPWFNQY* koji želimo da dobijemo.

Ovo znači da *LeaderboardCyclopeptideSequencing* algoritam radi dobro kada nam je eksperimentalni spektar malo različit od teorijskog.

4.5 Od 20 do više od 100 aminokiselina

U ovoj sekciji biće reči o poboljšanju našeg algoritma uz uvođenje premlaza koje postoje u stvarnosti, a koje smo do sada zanemarivali da bismo dali neke početne načine za rešavanje.

Kada smo govorili o proteinima, rekli smo da 20 aminokiselina najčešće učestvuje u njihovoj izgradnji i da su za nas, sa računarske tačke gledišta, proteini niske nad abzukom od 20 karaktera i da postoji još veliki broj aminokiselina nezavisno od izgradnje proteina u ćelijama živih bića. U gentskom kodu postoji kodovi samo za tih 20 aminokiselina, i u tabeli celobrojnih masa aminokiselina postoji mase samo za iste te aminokiseline. S obzirom da u ovom poglavlju razmatramo NRP peptide, peptide koji ne nastaju prema pravilima centralne dogme, onda ovi peptidi mogu da sadrže i neke nestandardne aminokiseline, one aminokiseline koje se ne nalaze među standardnih 20 aminokiselina. Na primer, tirocidin B sadrži nestandardnu aminokiselinu *Ornitin* (*Orn*). Za *Ornitin* ne postoji nukleotidni triplet u okviru genetskog koda na osnovu koga se ova aminokiselina dobija i ne postoji celobrojna masa u tabeli celobrojnih masa za aminokiseline. S ozbirom na to, možemo da pretpostavimo da bilo koji ceo broj između 57 i 200 (koliko nam iznosi najmanja i najveća masa standardnih aminokiselina) može biti masa neke nestandardne aminokiseline. Ovako nešto može da izgleda kao grubo ograničenje, ali je eksperimentalno potvrđeno da većina masa svih mogućih aminokiselina pripada ovom intervalu.

Spektar u kome nismo ograničeni na tabelu od samo 18 celobrojnih masa, već uzimamo u obzir da bilo koji ceo broj između 57 i 200 može da označava neku aminokiselinu, nazivamo **prošireni spektar**. Kada primenimo *Leaderboard* algoritam na prošireni spektar sa 10% lažnih i nedostajućih masa, peptid koji dobijemo *VKLFPWFN* – 98 – 65 sadrži neke vrednosti za mase koje ne odgovaraju nijednoj aminokiselini. Pošto *Leaderboard* algoritam ovde ne daje ispravne vrednosti, moramo da primenimo jedan sasvim novi princip.

4.6 Spektralna konvolucija

Kod algoritma sa proširenim spektrom podrazumeva se da svi cevi brojevi između 57 i 200 odgovaraju masama aminokiselina. To znači da razmatramo 144 ili više (znamo da jednoj masi može da odgovara više aminokiselina, a sa druge strane postoje vrednosti kojima ne odgovara nijedna) aminokiselina u koje spadaju i standardne i nestandardne aminokiseline. Želimo da smanjimo broj aminokiselina koje razmatramo.

Posmatrajmo eksperimentalni spektar za *NQEL*

0 99 113 114 128 227 257 299 355 356 370 371 484.

Mi znamo da je $Mass(E) = 129$ i vidimo da u spektru ne postoji ta vrednost. Sa druge strane, u spektru postoji $Mass(QE) = 257$ i $Mass(Q) = 128$. Razlika ove dve mase daje vrednost 129. Ova vrednost već deluje kao dobra vrednost za nedostajuću masu. U spektru postoji još ovakvih slučajeva. Recimo, $Mass(ELN) - Mass(LN) = 356 - 227 = 129$ i $Mass(NQEL) - Mass(LNQ) = 484 - 355 = 129$. Obe ove razlike ukazuju na masu od E koja nedostaje.

Uvodimo tabelu koja se naziva **spektralna konvolucija**.

Definicija 4.6. *Spektralna konvolucija je tabela koja pokazuje absolutnu vrednost razlike između svake dve mase u spektru.*

Primer spektralne konvolucije za spektar čije su lažne vrednosti označene sa „false” prikazan je na slici 4.8.

	""	false	L	N	Q	LN	QE	false	LNQ	ELN	QEL	NQE
0	99	113	114	128	227	257	299	355	356	370	371	
99	99											
113	113	14										
114	114	15	1									
128	128	29	15	14								
227	227	128	114	113	99							
257	257	158	144	143	129	30						
299	299	200	186	185	171	72	42					
355	355	256	242	241	227	128	98	56				
356	356	257	243	242	228	129	99	57	1			
370	370	271	257	256	242	143	113	71	15	14		
371	371	272	258	257	243	144	114	72	16	15	1	
484	484	385	371	370	356	257	227	185	129	128	114	113

Slika 4.8: Primer spektralne konvolucije.

Na preseku svake vrste i kolone u spektralnoj konvoluciji upisana je absolutna vrednost razlike celobrojnih masa. Kako iskoristiti spektralnu konvoluciju? Tražimo vrednosti razlike koje se pojavljuju najveći broj puta, a da se nalaze između 57 i 200. Obojene vrednosti na slici 4.8 se pojavljuju veći broj puta. To su vrednosti 99, 113, 114, 128 i 129. Ove vrednosti odgovaraju masama aminokiselina, redom, V, L, N, Q, E . Od 5 najčešćih aminokiselina u konvoluciji 4 čine peptid $NQEL$.

Kako bi izgledao unapređeni algoritam za sekvenciranje ciklopeptida ukoliko uzmemo u obzir i nestandardne aminokiseline, odnosno proširenu tabelu celobrojnih masa aminokiselina? Pseudokod je dat u nastavku.

```

1 ConvolutionCyclopeptideSequencing(Spectrum, N, M)
2 begin
3   Formirati spektralnu konvoluciju spektra Spectrum.
4   Uzeti M najcescih elemenata u konvoluciji (izmedju 57 i 200).
5   Primeniti LeaderboardCyclopeptideSequencing, formirajuci peptide samo na
     ↪ osnovu ovih M celih brojeva.
6 end

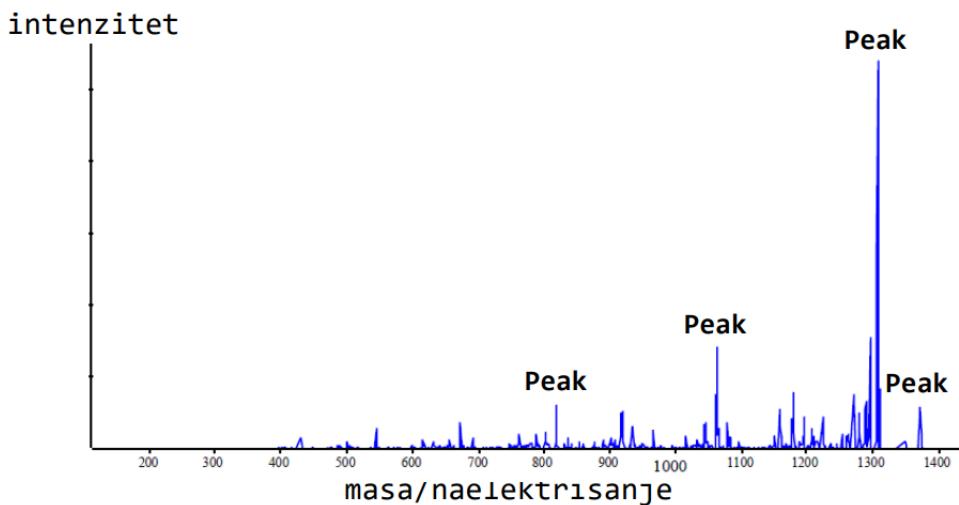
```

Algoritam *ConvolutionCyclopeptideSequencing* daje tačan rezultat i za spektre sa šumom od 10% i za spektre sa šumom od 25%, što pokazuje da je spektralna konvolucija odgovorila na sve izazove koji su postavljeni.

4.7 Spektri u realnosti

Kao što znamo, realnost je obično drugačija. Neke poteškoće iz realnosti smo zanemarivali. Koje?

- *Spectrum₂₅* je mnogo manje šumovit nego spektri dobijeni u praksi iz masenog spektrometra.
- Maseni spektrometar ne meri jednostavno fragmente podpeptida, već su postupci merenja mnogo komplikovaniji. Najpre se zaista vrši razbijanje datog peptida na fragmente. Zatim se oni sortiraju, korišćenjem elektromagnetskog polja, prema svojoj masi. Ono što maseni spektrometar meri jeste zapravo **odnos mase i naelektrisanja** za svaki fragment (znači nije baš masa) i određuje **intenzitet** (kao broj jona) u svakom odnosu mase i naelektrisanja. Šta to znači? To znači da kao izlaz iz masenog spektrometra ne dobijamo eksperimentalni spektar koji smo do sada imali prilike da vidimo, nego grafik intenziteta prema odnosu mase i naelektrisanja sa vrhovima na određenim mestima. Primer ovakvog grafika dat je na slici 4.9. Na osnovu vrhova na grafiku, određivaćemo sam sastav peptida. Ovaj grafik se



Slika 4.9: Primer grafika intenziteta prema odnosu mase i naelektrisanja.

naziva **realni spektar**. Rekonstrukcija peptida na osnovu realnog spektra biće obrađena u poglavljju 11.

4.8 Zadaci sa vežbi

U nastavku će biti predstavljeni zadaci sa vežbi na kursu rađeni u programskom jeziku Python.

4.8.1 Linear Spectrum

```

1  # Formiranje linearog spektra zadatog peptida
2  def linear_spectrum(peptide, amino_acid, amino_acid_mass):
3      prefix_mass = [0]
4      current_mass = 0
5      for i in range(len(peptide)):
6          for j in range(20):
7              if amino_acid[j] == peptide[i]:
8                  prefix_mass.append(current_mass + amino_acid_mass[j])
9                  current_mass += amino_acid_mass[j]
10
11
12     linear_spectrum = [0]
13     for i in range(len(prefix_mass)):
14         for j in range(i+1, len(prefix_mass)):
15             linear_spectrum.append(prefix_mass[j] - prefix_mass[i])
16
17     linear_spectrum.sort()
18     return linear_spectrum
19
20
21 def main():
22
23     # Lista aminokiselina
24     amino_acid = ['G', 'A', 'S', 'P', 'V', 'T', 'C', 'I', 'L', 'N', 'D', 'K',
25     ↪ 'Q', 'E', 'M', 'H', 'F', 'R', 'Y', 'W']
26
27     # Lista masa odgovarajućih aminokiselina
28     amino_acid_mass = [57, 71, 87, 97, 99, 101, 103, 113, 113, 114, 115, 128,
29     ↪ 128, 129, 131, 137, 147, 156, 163, 186]
30
31     # Zadati peptid
32     peptide = "NQEL"
33
34     spectrum = linear_spectrum(peptide, amino_acid, amino_acid_mass)
35     print(spectrum)
36
37 if __name__ == "__main__":
38     main()
```

4.8.2 Cyclic Spectrum

```

1  # Formiranje ciklicnog spektra zadatog peptida
2  def cyclic_spectrum(peptide, amino_acid, amino_acid_mass):
3      prefix_mass = [0]
4      current_mass = 0
5      for i in range(len(peptide)):
```

```

7     for j in range(20):
8         if amino_acid[j] == peptide[i]:
9             prefix_mass.append(current_mass + amino_acid_mass[j])
10            current_mass += amino_acid_mass[j]
11
12    peptide_mass = prefix_mass[-1]
13    cyclic_spectrum = [0]
14    for i in range(len(prefix_mass)):
15        for j in range(i+1, len(prefix_mass)):
16            cyclic_spectrum.append(prefix_mass[j] - prefix_mass[i])
17            if i > 0 and j < len(prefix_mass)-1:
18                cyclic_spectrum.append(peptide_mass - (prefix_mass[j] -
19                  prefix_mass[i]))
20
21    cyclic_spectrum.sort()
22
23
24 def main():
25     amino_acid = ['G', 'A', 'S', 'P', 'V', 'T', 'C', 'I', 'L', 'N', 'D', 'K', 'Q',
26                   'E', 'M', 'H', 'F', 'R', 'Y', 'W']
27     amino_acid_mass = [57, 71, 87, 97, 99, 101, 103, 113, 113, 114, 115, 128,
28                         128, 129, 131, 137, 147, 156, 163, 186]
29
30     peptide = "NQE"
31
32
33 if __name__ == "__main__":
34     main()

```

4.8.3 Cyclopeptide Sequencing

```

1 import copy
2
3 # Formiranje ciklicnog spektra peptida
4 def cyclic_spectrum(peptide, amino_acid, amino_acid_mass):
5     prefix_mass = [0]
6     current_mass = 0
7     for i in range(len(peptide)):
8         for j in range(20):
9             if amino_acid[j] == peptide[i]:
10                 prefix_mass.append(current_mass + amino_acid_mass[j])
11                 current_mass += amino_acid_mass[j]
12
13     peptide_mass = prefix_mass[-1]
14     cyclic_spectrum = [0]
15     for i in range(len(prefix_mass)):
16         for j in range(i+1, len(prefix_mass)):
17             cyclic_spectrum.append(prefix_mass[j] - prefix_mass[i])
18             if i > 0 and j < len(prefix_mass)-1:
19                 cyclic_spectrum.append(peptide_mass - (prefix_mass[j] -

```

```

        ↪ prefix_mass[i]))
20
21     cyclic_spectrum.sort()
22     return cyclic_spectrum
23
24 # Prosirivanje liste peptida dodavanjem svih mogucih amino kiselina na kraj
25     ↪ lanca
25 def expand(peptides, amino_acid):
26     extension = []
27
28     for peptide in peptides:
29         for aa in amino_acid:
30             extension.append(peptide + aa)
31
32     return extension
33
34 # Izracunavanje ukupne mase peptida kao sume svih aminokiselina u lancu
35 def mass(peptide, amino_acid, amino_acid_mass):
36     total_mass = 0
37
38     for i in range(len(peptide)):
39         for j in range(len(amino_acid)):
40             if peptide[i] == amino_acid[j]:
41                 total_mass += amino_acid_mass[j]
42
43     return total_mass
44
45 # Izdvajanje sume celog peptida iz spektra
46 def parent_mass(spectrum):
47     return spectrum[-1]
48
49 # Formiranje linearog spektra
50 def linear_spectrum(peptide, amino_acid, amino_acid_mass):
51     prefix_mass = [0]
52     current_mass = 0
53     for i in range(len(peptide)):
54         for j in range(20):
55             if amino_acid[j] == peptide[i]:
56                 prefix_mass.append(current_mass + amino_acid_mass[j])
57                 current_mass += amino_acid_mass[j]
58
59     linear_spectrum = [0]
60     for i in range(len(prefix_mass)):
61         for j in range(i+1, len(prefix_mass)):
62             linear_spectrum.append(prefix_mass[j] - prefix_mass[i])
63
64     linear_spectrum.sort()
65     return linear_spectrum
66
67
68 # Provera da li je dati peptid konzistentan sa zadatim spektrom
69 def consistent(peptide, target_spectrum, amino_acid, amino_acid_mass):
70     peptide_linear_spectrum = linear_spectrum(peptide, amino_acid,

```

```

    ↪ amino_acid_mass)

71   for aa in peptide_linear_spectrum:
72     found = False
73     for aa_p in target_spectrum:
74       if aa_p == aa:
75         found = True
76     if found == False:
77       return False
78
79   return True
80
81
82
83 # Sekvenciranje ciklopeptida
84 def cyclopeptide_sequencing(spectrum, amino_acid, amino_acid_mass):
85   peptides = [',']
86   i = 1;
87   while len(peptides) > 0:
88     next_peptides = []
89     peptides = expand(peptides, amino_acid)
90     next_peptides = copy.copy(peptides)
91     for peptide in peptides:
92       if mass(peptide, amino_acid, amino_acid_mass) == parent_mass(
93         ↪ spectrum):
94         if cyclic_spectrum(peptide, amino_acid, amino_acid_mass) ==
95           ↪ spectrum:
96           print(peptide)
97           next_peptides.remove(peptide)
98         elif not consistent(peptide, spectrum, amino_acid, amino_acid_mass)
99           ↪ :
100           next_peptides.remove(peptide)
101   peptides = next_peptides
102
103
104
105
106
107
108
109
110
111 if __name__ == "__main__":
112   main()

```

4.8.4 Leaderboard Cyclopeptide Sequencing

```

1 import copy
2

```

```

3 # Formiranje ciklicnog spektra peptida
4 def cyclic_spectrum(peptide, amino_acid, amino_acid_mass):
5     prefix_mass = [0]
6     current_mass = 0
7     for i in range(len(peptide)):
8         for j in range(20):
9             if amino_acid[j] == peptide[i]:
10                 prefix_mass.append(current_mass + amino_acid_mass[j])
11                 current_mass += amino_acid_mass[j]
12
13     peptide_mass = prefix_mass[-1]
14     cyclic_spectrum = [0]
15     for i in range(len(prefix_mass)):
16         for j in range(i+1, len(prefix_mass)):
17             cyclic_spectrum.append(prefix_mass[j] - prefix_mass[i])
18             if i > 0 and j < len(prefix_mass)-1:
19                 cyclic_spectrum.append(peptide_mass - (prefix_mass[j] -
19 → prefix_mass[i]))
20
21     cyclic_spectrum.sort()
22     return cyclic_spectrum
23
24 # Prosirivanje liste peptida dodavanjem svih mogucih amino kiselina na kraj
24 → lanca
25 def expand(peptides, amino_acid):
26     extension = []
27
28     for peptide in peptides:
29         for aa in amino_acid:
30             extension.append(peptide + aa)
31
32     return extension
33
34 # Izracunavanje ukupne mase peptida kao sume svih aminokiselina u lancu
35 def mass(peptide, amino_acid, amino_acid_mass):
36     total_mass = 0
37
38     for i in range(len(peptide)):
39         for j in range(len(amino_acid)):
40             if peptide[i] == amino_acid[j]:
41                 total_mass += amino_acid_mass[j]
42
43     return total_mass
44
45 # Izdvajanje sume celog peptida iz spektra
46 def parent_mass(spectrum):
47     return spectrum[-1]
48
49 # Formiranje linearнog spektra
50 def linear_spectrum(peptide, amino_acid, amino_acid_mass):
51     prefix_mass = [0]
52     current_mass = 0
53     for i in range(len(peptide)):
```

```

54     for j in range(20):
55         if amino_acid[j] == peptide[i]:
56             prefix_mass.append(current_mass + amino_acid_mass[j])
57             current_mass += amino_acid_mass[j]
58
59     linear_spectrum = [0]
60     for i in range(len(prefix_mass)):
61         for j in range(i+1, len(prefix_mass)):
62             linear_spectrum.append(prefix_mass[j] - prefix_mass[i])
63
64     linear_spectrum.sort()
65     return linear_spectrum
66
67
68 # Provera da li je dati peptid konzistentan sa zadatim spektrom
69 def consistent(peptide, target_spectrum, amino_acid, amino_acid_mass):
70     peptide_linear_spectrum = linear_spectrum(peptide, amino_acid,
71                                               amino_acid_mass)
72
73     for aa in peptide_linear_spectrum:
74         found = False
75         for aa_p in target_spectrum:
76             if aa_p == aa:
77                 found = True
78         if found == False:
79             return False
80
81     return True
82
83 def score(peptide, spectrum_2, amino_acid, amino_acid_mass):
84     p1 = 0
85     p2 = 0
86     score = 0
87
88     spectrum_1 = cyclic_spectrum(peptide, amino_acid, amino_acid_mass)
89
90     while p1 < len(spectrum_1) and p2 < len(spectrum_2):
91         if spectrum_1[p1] == spectrum_2[p2]:
92             score += 1
93             p1 += 1
94             p2 += 1
95         elif spectrum_1[p1] < spectrum_2[p2]:
96             p1 += 1
97         else:
98             p2 += 1
99
100    return score
101
102 def linear_score(peptide, spectrum_2, amino_acid, amino_acid_mass):
103     p1 = 0
104     p2 = 0
105     score = 0

```

```

106
107     spectrum_1 = linear_spectrum(peptide, amino_acid, amino_acid_mass)
108
109     while p1 < len(spectrum_1) and p2 < len(spectrum_2):
110         if spectrum_1[p1] == spectrum_2[p2]:
111             score += 1
112             p1 += 1
113             p2 += 1
114         elif spectrum_1[p1] < spectrum_2[p2]:
115             p1 += 1
116         else:
117             p2 += 1
118
119     return score
120
121 # Sekvenciranje ciklopeptida
122 def leaderboard_cyclopeptide_sequencing(spectrum, N, amino_acid,
123                                         amino_acid_mass):
123     leaderboard = [',']
124     leader_peptide = ','
125     while len(leaderboard) > 0:
126         next_peptides = []
127         leaderboard = expand(leaderboard, amino_acid)
128         next_leaderboard = copy.copy(leaderboard)
129         for peptide in leaderboard:
130             if mass(peptide, amino_acid, amino_acid_mass) == parent_mass(
131                 spectrum):
132                 if score(peptide, spectrum, amino_acid, amino_acid_mass) >
133                     score(leader_peptide, spectrum, amino_acid, amino_acid_mass):
134                     leader_peptide = peptide
135                 elif mass(peptide, amino_acid, amino_acid_mass) > parent_mass(
136                     spectrum):
137                     next_leaderboard.remove(peptide)
138         leaderboard = trim(next_leaderboard, spectrum, N, amino_acid,
139                             amino_acid_mass)
140     return leader_peptide
141
142
143 def trim(leaderboard, spectrum, N, amino_acid, amino_acid_mass):
144     linear_scores = []
145     for j in range(len(leaderboard)):
146         peptide = leaderboard[j]
147         linear_scores.append(linear_score(peptide, spectrum, amino_acid,
148                                         amino_acid_mass))
149
150     leaderboard_zipped = list(zip(linear_scores, leaderboard))
151     leaderboard_zipped.sort(reverse=True)
152
153     leaderboard = [el[1] for el in leaderboard_zipped]
154     for j in range(N, len(leaderboard_zipped)):
155         if leaderboard_zipped[j][0] < leaderboard_zipped[N-1][0]:
156             leaderboard = [el[1] for el in leaderboard_zipped[:j]]
157
158     return leaderboard

```

```
153     return leaderboard
154
155
156
157
158 def main():
159     amino_acid = ['G', 'A', 'S', 'P', 'V', 'T', 'C', 'I', 'L', 'N', 'D', 'K',
160                   ↪ 'Q', 'E', 'M', 'H', 'F', 'R', 'Y', 'W']
160     amino_acid_mass = [57, 71, 87, 97, 99, 101, 103, 113, 113, 114, 115, 128,
161                   ↪ 128, 129, 131, 137, 147, 156, 163, 186]
161
162     peptide = "SPQR"
163
164     spectrum = cyclic_spectrum(peptide, amino_acid, amino_acid_mass)
165
166     print(leaderboard_cyclopeptide_sequencing(spectrum, 10, amino_acid,
167                                               ↪ amino_acid_mass))
167
168 if __name__ == "__main__":
169     main()
```

Glava 5

Kako poredimo biološke sekvence?

5.1 Biološki uvid u poređenje sekvenci

Kako su biološke sekvence podložne promeni, umetanju i brisanju, čest je slučaj da i-ti simbol jedne sekvene odgovara simbolu na drugoj poziciji druge sekvene. U tom slučaju, cilj je postići najbolje poklapanje simbola. Na primer, *ATGCATGC* i *TGCATGCA* nemaju delove koji se poklapaju, pa je njihova Hamingova udaljenost 8:

ATGCATGC
TGCATGCA

Ali ako ih malo drugačije poravnamo, ove dve niske imaju 6 poklapajućih pozicija:

A~~TGCATGC~~ –
–~~TGCATGCA~~

Stringovi ATGCTTA i TGCATTAA imaju manje uočljive sličnosti:

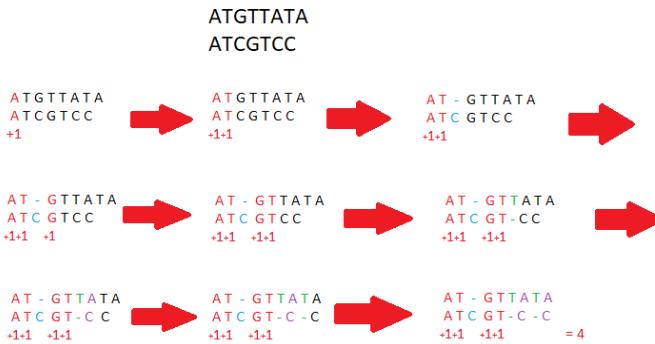
A~~TGC~~ – ~~TTA~~ –
–~~TGCATTAA~~

Ovi primjeri navode nas da definisemo dobro poravnanje kao ono koje ima najveći mogući broj poklapanja. Povećanje broja poklapanja simbola možemo posmatrati kao igricu u kojoj u svakom potezu imamo dva izbora. Možemo da uklonimo oba simbola i osvojimo poen ako su oni isti ili možemo ukloniti simbol iz jedne od niski, ne osvojimo poene, ali omogućimo da u daljem igranju osvojimo više poena. Cilj je da maksimizujemo broj poena.

5.2 Igra poravnanja i najduža zajednička podsekvence

Kod **Igre poravnanja** cilj je ukloniti sve simbole iz sekvenci tako da pritom sakupimo što više poena :

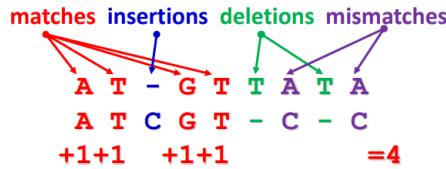
- Uklanjanje prvog simbola iz svake sekvene
- 1 poen ako se simboli poklapaju, 0 ako se simboli ne poklapaju
- Uklanjanje prvog simbola iz jedne sekvene
 - 0 poena



Slika 5.1: Igra poravnjanja

Poravnanje dve sekvene predstavlja matricu koja ima dva reda:

1. red: simboli prve sekvene (redom) eventualno sa ubačenim “-”
2. red: simboli druge sekvene (redom) eventualno sa ubačenim “-”



Slika 5.2: Poravnjanje

5.2.1 Najduža zajednička podsekvence

Poklapanja (matches) u poravnjanju dve sekvene (u primeru 5.2 to je ATGT) formiraju njihovu zajedničku podsekvencu.

Problem 2 (Problem najduže zajedničke podniske). *Naći najdužu zajedničku podsekvencu dve niske.*

Ulaz: Dve niske.

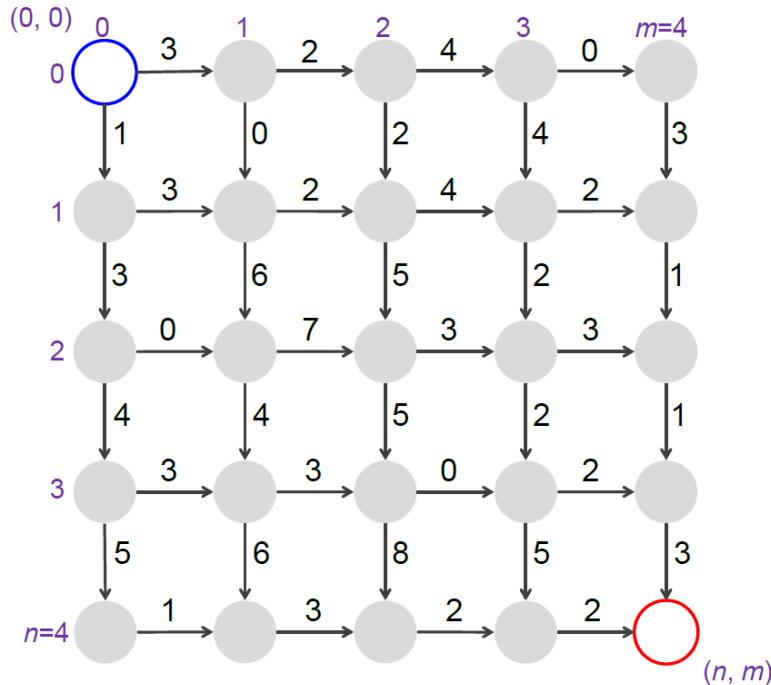
Izlaz: Najduža zajednička podsekvanca ovih niski

5.3 Problem turiste na Menhetnu

Pre svega postavimo problem:

Problem 3 (Problem turiste na Menhetnu). *Naći najdužu putanju u pravougaonoj mreži gradskih ulica.*

Ulaz: Usmeren težinski mrežni graf.
Izlaz: Najduža putanja od početnog (*source*) do krajnjeg čvora (*sink*) u mrežnom grafu.



Slika 5.3: Problem turiste na Menhetnu

Na slici 5.3 grafički je prikazan problem turiste na Menhetnu. Cilj je stići od plavog do crvenog kruga i pri tom sakupiti što više poena. Dozvoljeno kretanje je dole i desno. Možemo koristiti pohlepni algoritam i tako doći do cilja, ali da li smo tako sakupili najviše poena?

Dodatna izmena grafa bi bila da imamo i dijagonalne grane (5.4).

Time dolazimo do sledećeg problema:

Problem 4 (Problem najduže putanje u usmerenom grafu). *Naći najdužu putanju između dva čvora u težinskom usmerenom grafu.*

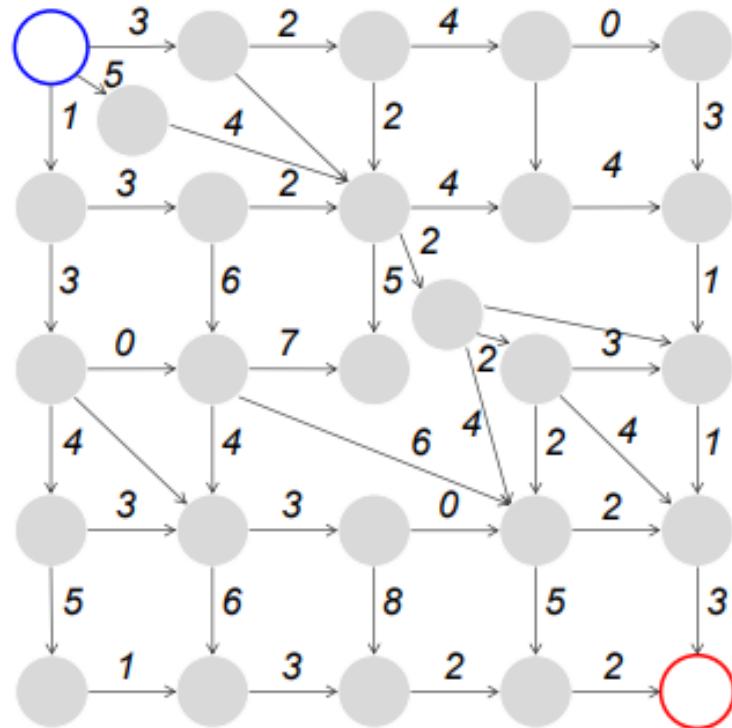
Ulaz: Usmereni težinski graf sa označenim čvorovima *source* i *sink*.

Izlaz: Najduža putanja od čvora *source* do čvora *sink* u usmerenom težinskom grafu.

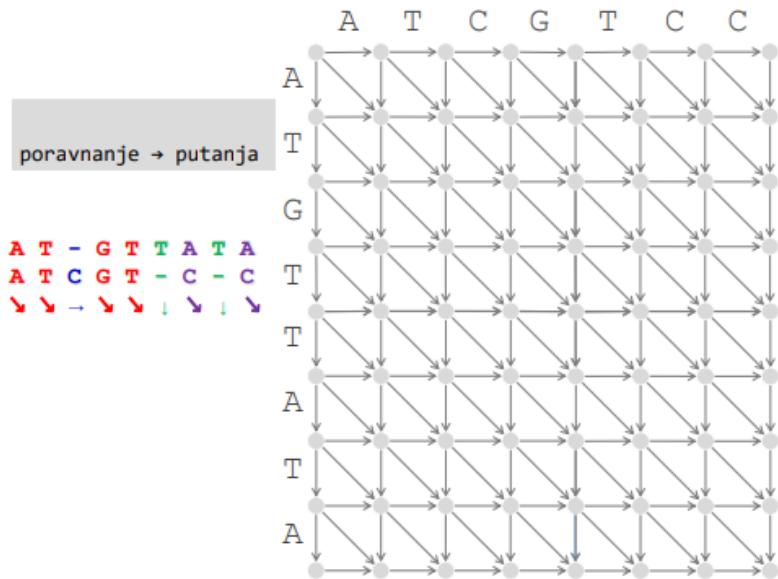
Ako se prisetimo igre poravnjanja, videćemo da postoji veza između ova dva problema (igre).

Pitamo se kako izgraditi graf za igru poravnjanja i za problem najduže podsekvence. To ćemo uraditi na sledeći način:

- Vrste označimo aminokiselinama iz prve niske
- Kolone označimo aminokiselinama iz druge niske
- U svaku presečnu tačku postavimo jedan čvor
- Gde god je moguće, postaviti vertikalne (insercija), horizontalne (delecija) i dijagonalne grane (match ili mismatch)
- Dijagonalne grane otežati koeficijentom 1, ostale koeficijentom 0



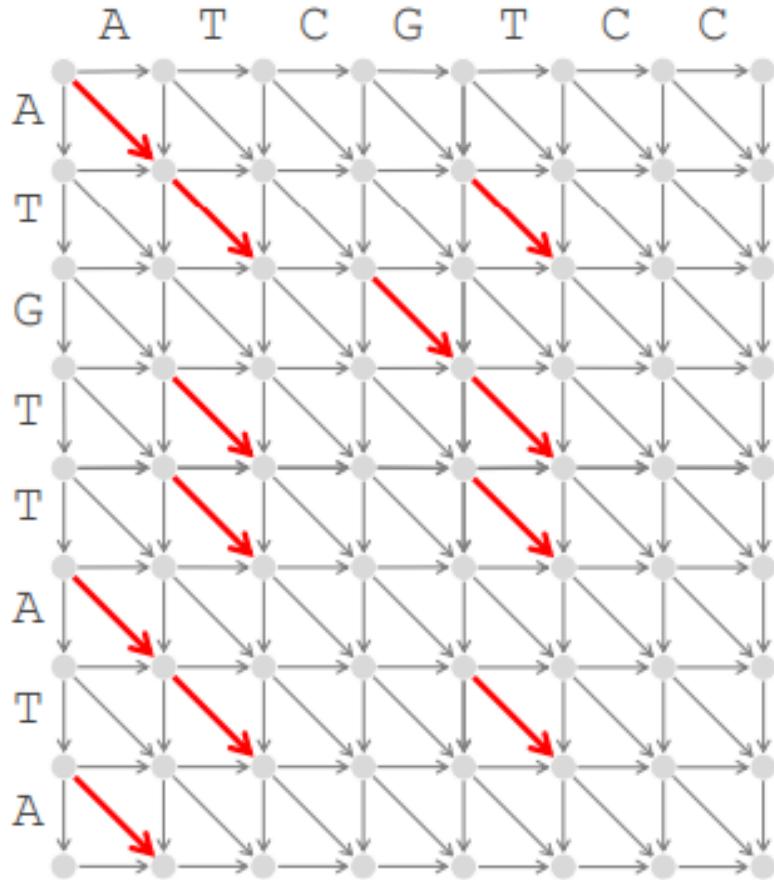
Slika 5.4: Nepravilna mreža



Slika 5.5: Poravnanje → Putanja

- Problem najduže zajedničke podsekvence se svodi na problem nalaženja najduže putanje između dva data čvora u usmerenom grafu

Kada nađemo poravnanje najvišeg skora našli smo i najdužu putanju u mrežnom grafu. Dijagonalne crvene grane odgovaraju poklapajućim simboli i imaju skor 1 (5.6)



Slika 5.6: Poravnanje → Putanja

5.4 Problem kusura

Upoznajmo se sa sledećim problemom:

Problem 5 (Problem vraćanja kusura). *Naći minimalan broj novčića neophodnih za vraćanje kusura.*

Ulaz: Ceo broj $money$ i niz pozitivnih celih brojeva ($coin_1, coin_2, \dots, coin_d$).

Izlaz: Minimalan broj novčića ($coin_1, coin_2, \dots, coin_d$) u apoenima koji rasitnjava sumu $money$.

5.4.1 Pohlepni algoritam

Najzastupljeniji način vraćanja kusura širom sveta podrazumeva iterativno traženje sledećeg najvećeg novčića.

To bi značilo da bismo za kusur od 42 dinara dobili sledeće novčiće: $20 + 10 + 10 + 2$.

Ovakav način vraćanja kusura opisuje takozvani pohlepni algoritam.

```

1 GreedyChange(money)
2 begin
```

```

3   change ← empty collection of coins
4   while money > 0
5       coin ← largest denomination that does not exceed money
6       add coin to change
7       money ← money - coin
8   return change
9 end

```

Međutim, ako malo bolje razmislimo ovo rešenje zapravo nije najbolje. Kusur bismo mogli vratiti i sa manje novčića na sledeći način: $42 = 20 + 20 + 2$

Zaključak: GreedyChange ne daje optimalno rešenje!

5.4.2 Rekursivni algoritam

Pokušajmo sada da problem rešimo na drugačiji način koristeći rekurziju. Za zadate apoene 6, 5, 1, koji je najmanji broj novčića neophodnih za vraćanje kusura od 9 centi?

money	1	2	3	4	5	6	7	8	9	10	11	12
MinNumCoins			?	?				?	?			

Slika 5.7: Vraćanje kusura - rekurzija

Problem resavamo tako sto prvo od 9 oduzmemo 6 i dobijemo 3 kao ostatak kusura. Dakle, 9 se može vratiti od jednog novčića od 6 apoena i jos plus broj novčića koji je potreban za preostali deo kusura od 3 centa.

U istoj iteraciji analogno računamo za preostale apoene.

Na slici 5.7 crvenim znakom pitanja označeno je traženo rešenje koje dobijemo rešavanjem manjih problema za kusure 3, 4 i 8.

$$\text{MinNumCoins}(9) = \min \begin{cases} \text{MinNumCoins}(9 - 6) + 1 = \text{MinNumCoins}(3) + 1 \\ \text{MinNumCoins}(9 - 5) + 1 = \text{MinNumCoins}(4) + 1 \\ \text{MinNumCoins}(9 - 1) + 1 = \text{MinNumCoins}(8) + 1 \end{cases}$$

Na osnovu prethodnog, moguće je izvesti opštu formulu:

$$\text{MinNumCoins}(\text{money}) = \min \begin{cases} \text{MinNumCoins}(\text{money} - \text{coin}_1) + 1 \\ \dots \\ \text{MinNumCoins}(\text{money} - \text{coin}_d) + 1 \end{cases}$$

Hajde sada da vidimo kako bismo to isprogramirali:

```

1 RecursiveChange(money, coins)
2 begin
3     if money = 0
4         return 0

```

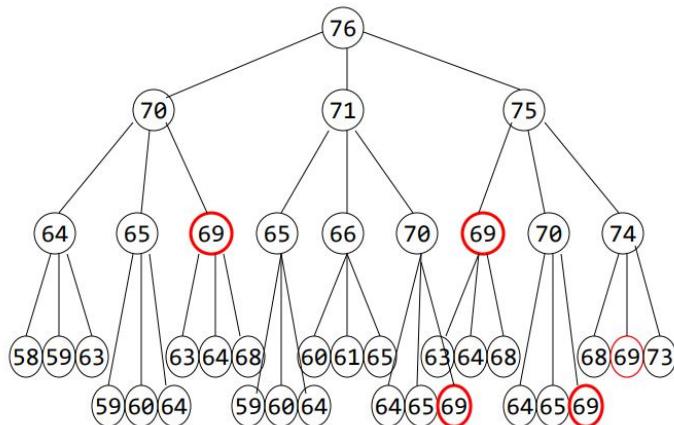
```

5     MinNumCoins ← infinity
6     for i ← 1 to |coins|
7         if money ≥ coini
8             NumCoins ← RecursiveChange(money - coini, coins)
9             if numCoins + 1 < MinNumCoins
10                MinNumCoins ← numCoins + 1
11
12     return MinNumCoins
13 end

```

Reklo bi se da smo sada dobili odgovarajuci algoritam za naš problem, hajde to da proverimo. Postavlja se pitanje, koliko je brz RecursiveChange?

Pokušajmo na konkretnom primeru da dođemo do rešenja. Neka naš problem sada bude vraćanje kusura od 76 centi. Pomoću rekurzivnog stabla demonstrirajmo ponašanje našeg algoritma:



Slika 5.8: Vracanje kusura - ponašanje rekurzivnog algoritma

Ono što se odmah može primetiti jeste višestruko pozivanje algoritma za vrednost od 69 centi, čak 6 puta!

Daljim procenama možemo doći do zaključka da se optimalna kombinacija novčića za 30 centi izračunava milijardama puta!

Sada je očigledno da nam rekurzija ne rešava problem na najbolji mogući način.

5.4.3 Vraćanje kusura dinamičkim programiranjem

Cilj nam je da izbegnemo višestruka izračunavanja vraćanja kusura za istu vrednost, tako da bi ideja bila da imamo objekat koji će pamtitи sva računanja i iz koga ćemo čitati već izračunate vrednosti.

Dakle, umesto vremenski zahtevnih poziva

RecursiveChange(money - coin_i, coins)

jednostavno bismo potražili vrednosti iz unapred izračunate tabele

MinNumCoins($money - coin_i$).

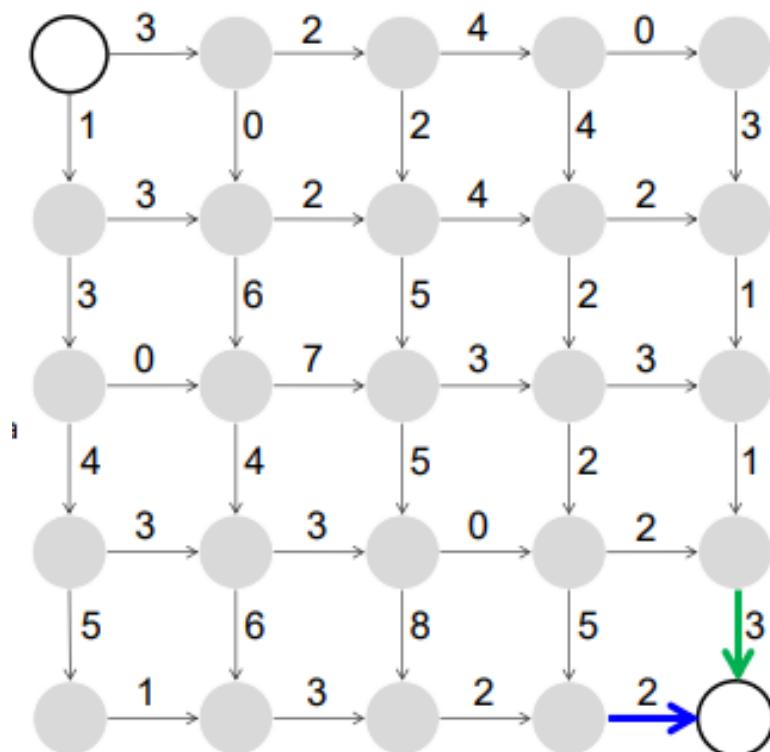
```

1 DPChange(money, coins)
2 begin
3     MinNumCoins(0) ← 0
4     for m ← 1 to money
5         MinNumCoins(m) ← infinity
6         for i ← 1 to |coins|
7             if m ≥ coini
8                 if MinNumCoins(m - coini) + 1 < MinNumCoins(m)
9                     MinNumCoins(m) ← MinNumCoins(m - coini) + 1
10    return MinNumCoins(money)
11 end

```

5.5 Dinamičko programiranje i putokazi za povratak

Posmatramo jednostavniji, Menhetn graf: Pretpostavimo da do čvora sink možemo doći samo na dva načina: kretanjem južno ↓ ili kretanjem istočno →



Slika 5.9: Južno ili istočno?

Prvo probamo da rešimo problem rekurzivno:

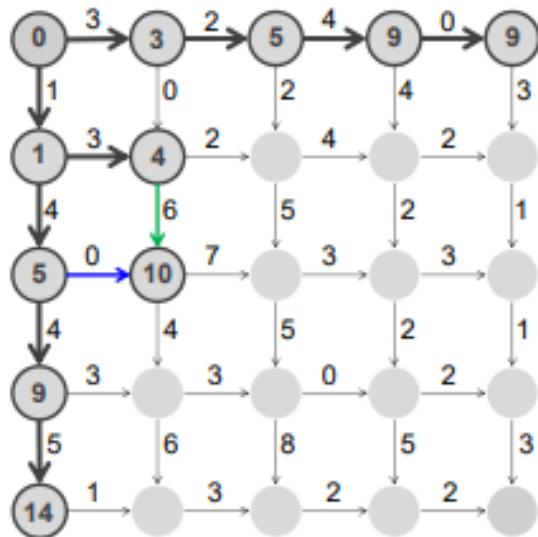
```
1 SouthOrEast(n, m)
2 if n=0 and m=0
3     return 0
4 x ← -infinity, y ← -infinity
```

```

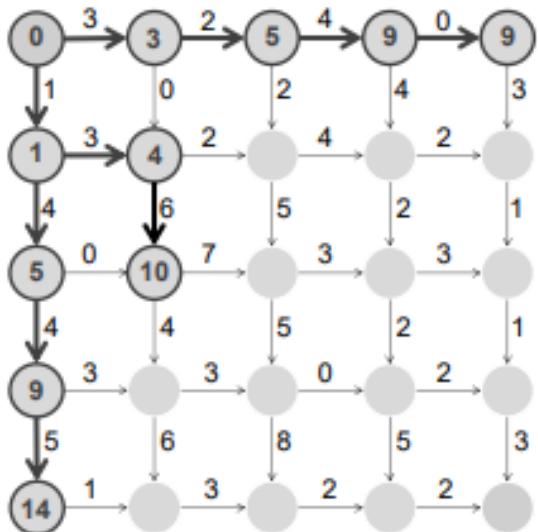
5 if n > 0
6 x ← SouthOrEast(n-1,m)+weight of edge "↓" into (n, m)
7 if m > 0
8 y ← SouthOrEast(n,m-1)+ weight of edge "→" into (n,m)
9 return max{x, y}

```

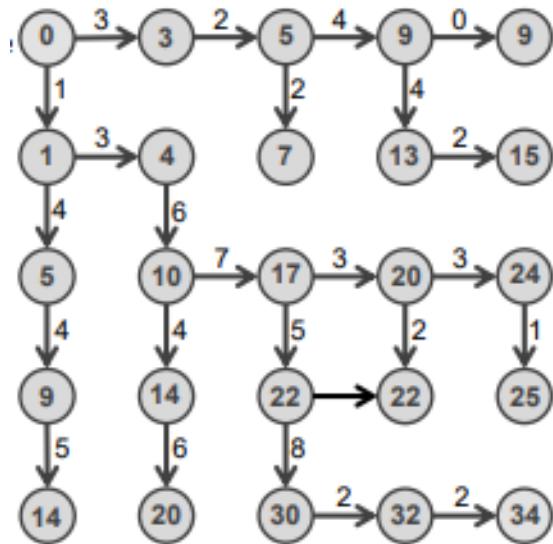
Ovaj algoritam se poziva za svaki čvor u grafu veličine $m \times n$, a pri tom se dešava da za jedan isti čvor računamo više puta. Zbog toga je ovaj pristup previše spor, pa prelazimo na dinamičko programiranje. Krenućemo od početnog čvora. Zatim, u čvor (i, j) upisujemo dužinu maksimalne putanje od $(0,0)$ do (i,j) . Prvo izračunamo za čvorove na obodu grafa a zatim, kolonu po kolonu, za preostale čvorove.



Slika 5.10: Južno ili istočno?



Slika 5.11: Južno ili istočno?



Slika 5.12: Južno ili istočno?

Na slici 5.12 prikazane su podebljane grane koje predstavljaju putokaze za povratak od čvora sink do čvora source.

5.5.1 Rekurentna relacija dinamičkog programiranja kod Menhetn grafa

$s_{i,j}$: the length of a longest path from (0,0) to (i,j)

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight of edge } \downarrow \text{ "into } (i,j) \\ s_{i,j-1} + \text{weight of edge } \rightarrow \text{ "into } (i,j) \end{cases}$$

```

1 ManhattanTourist(n, m, Down, Right)
2  $s_{0,0} \leftarrow 0$ 
3 for i  $\leftarrow 1$  to n
4    $s_{i,0} \leftarrow s_{i-1,0} + down_{i,0}$ 
5 for j  $\leftarrow 1$  to m
6    $s_{0,j} \leftarrow s_{0,j-1} + right_{0,j}$ 
7 for i  $\leftarrow 1$  to n
8   for j  $\leftarrow 1$  to m
9      $s_{i,j} \leftarrow \max \{ s_{i-1,j} + down_{i,j}, s_{i,j-1} + right_{i,j} \}$ 
10 return  $s_{n,m}$ 
```

5.6 Od Menhetna do grafa poravnjanja

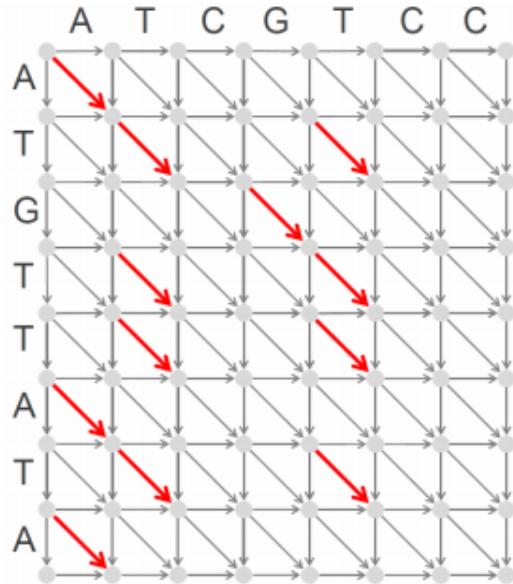
5.6.1 Rekurentna relacija dinamičkog programiranja kod grafa poravnjanja

Najduži put (slika 5.13) od (0,0) do (i,j) se računa:

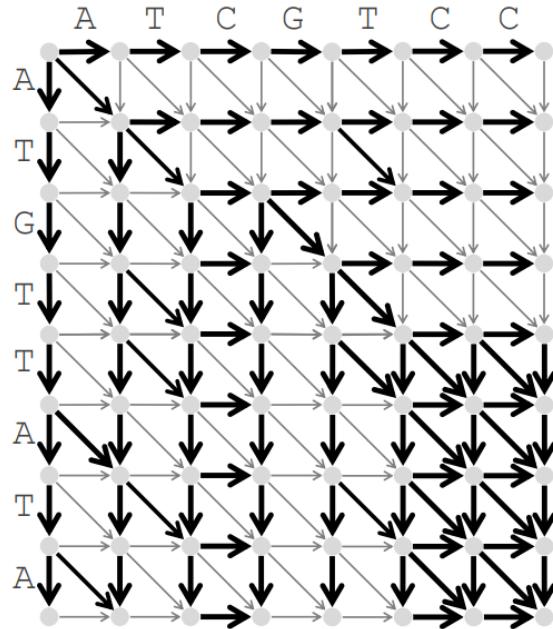
$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight of edge } \downarrow \text{ "into } (i,j) \\ s_{i,j-1} + \text{weight of edge } \rightarrow \text{ "into } (i,j) \\ s_{i-1,j-1} + \text{weight of edge } \searrow \text{ "into } (i,j) \end{cases}$$

Što dalje daje :

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1, v_i = w_j \\ s_{i-1,j-1} + 0, v_i \neq w_j \end{cases}$$

Slika 5.13: Crvene grane težina 1, ostale grane težina 0, v_i i w_j oznake vrste i kolone

U slici 5.14 se vide boldovane grane koje su nastale primenom pravila rekurentne relacije. One predstavljaju putokaze za povratak (backtrack) kod grafa za najdužu zajedničku podsekvencu.



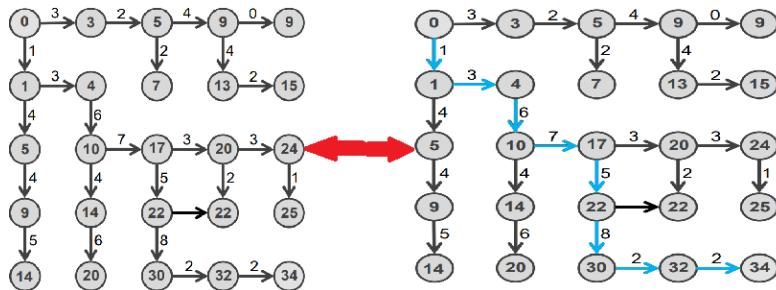
Slika 5.14: Putokazi za povratak (backtrack)

5.6.2 Računanje putokaza za povratak

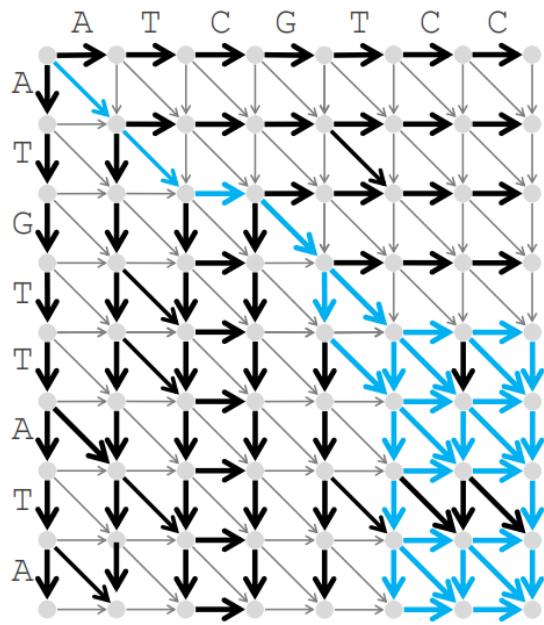
$$s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1, v_i = w_j \\ s_{i-1,j-1} + 0, v_i \neq w_j \end{cases}$$

$$\text{backtrack}_{i,j} \leftarrow \max \begin{cases} " \rightarrow ", s_{i,j} = s_{i,j-1} \\ " \downarrow ", s_{i,j} = s_{i-1,j} \\ " \searrow ", \text{otherwise} \end{cases}$$

Podsetimo se sada kako bismo rekonstruisali putanju preko putokaza kod Menhetn grafa? Krenuli bismo od krajnjeg čvora (sink) i pratili putokaze u obrnutom smeru do početnog čvora (source) 5.15.



Slika 5.15: Rekonstrukcija putanje preko putokaza kod Menhetn grafa



Slika 5.16: Backtracking

Sada na slici 5.16 možemo videti povratak (backtracking) kod grafa za najdužu zajedničku podsekvencu.

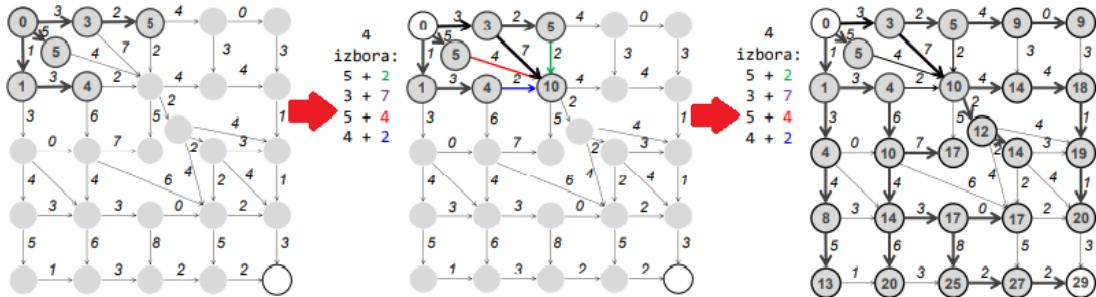
5.6.3 Određivanje najduže zajedničke podsekvence (LCS – longest common subsequence) korišćenjem putokaza za povratak

```

1 OutputLCS (backtrack, v, i, j)
2 if i = 0 or j = 0
3     return
4 if backtracki,j = "→"
5     OutputLCS (backtrack, v, i, j-1)
6 else if backtracki,j = "↓"
7     OutputLCS (backtrack, v, i-1, j)
8 else
9     OutputLCS (backtrack, v, i-1, j-1)
10    output  $v_i$ 
```

Do sada smo prepostavljali da graf u kom tražimo najdužu putanju ima samo tri vrste grana. Da li se OutputLCS može generalizovati tako da važi i za grafove koji nemaju tako specifičnu topologiju?

Kako se rekurentna relacija dinamičkog programiranja menja za ovakav graf? 5.17



Slika 5.17

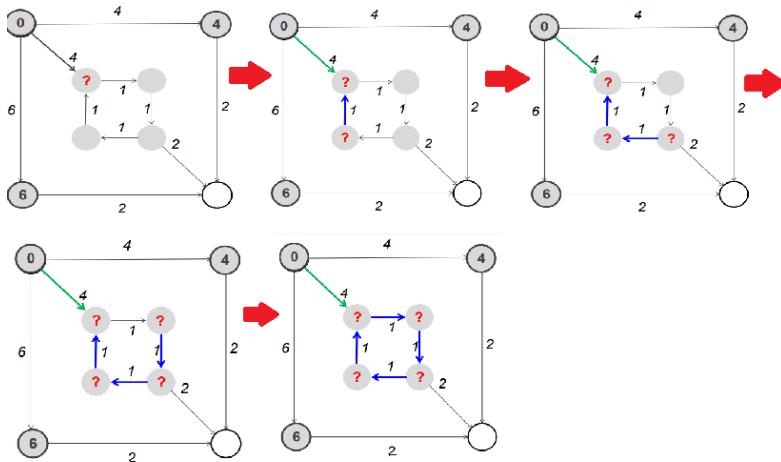
$$s_a = \max_{\text{all predecessors } b \text{ of node } a} \{s_b + \text{weight of edge from } b \text{ to } a\}$$

Računanje skora za SVE prethodnike 5.18.

- Kod ovakve rekurentne relacije, važno je da pri računanju s_a imamo izračunate s_b za sve čvorove prethodnike b (čvorovi za koje postoji grana do čvora a). Da li je to moguće u bilo kom usmerenom težinskom grafu? Odgovor je **nije**. Da bismo kod svakog čvora mogli da mogli da izračunamo skor za sve njegove prethodnike, usmereni težinski graf mora biti acikličan. DAG (Directed Acyclic Graph)
- Ako je dat usmereni aciklični graf, da li njegove čvorove možemo poređati u niz tako da njihov redosled u nizu osigurava uslov da pri računanju s_a imamo izračunate s_b za sve čvorove prethodnike b (čvorovi za koje postoji grana do čvora a)? Odgovor je **da**, moguće je poređati sve čvorove grafa u niz i taj niz topološki sortirati

5.6.4 Topološko sortiranje

- **Topološko sortiranje** : Sortiranje čvorova DAG-a u nizu tako da sve grane u takvom nizu idu s leva na desno.
- **Teorema:** Svaki DAG se može topološki sortirati.
- Topološko sortiranje svakog DAG-a se obavlja za $O(\#edges)$ koraka.



Slika 5.18: Začarani krug

Algoritam za nalaženje najduže putanje u DAG-u :

```

1 LongestPath(Graph, source, sink)
2 for each node a in Graph
3      $s_a \leftarrow -\infty$ 
4  $s_{source} \leftarrow 0$ 
5 topologically order Graph
6 for each node a (from source to sink in topological order)
7  $s_a \leftarrow \max_{all predecessors b of node a} \{s_b + \text{weight of edge from } b \text{ to } a\}$ 
8 return  $s_{sink}$ 
```

- Pošto svaka grana učestvuje tačno jednom, složenost je $O(\#edges)$
- LongestPath vraća dužinu najdužeg zajedničkog podniza ali ne rekonstruiše putanju

5.7 Od globalnog do lokalnog poravnanja

Uvedimo sledeće:

- Skor poravnjanja do sada - #*matches*
- Skor sa mismatch i indel kaznama - #*matches* – $\mu * \#mismatches$ – $\sigma * \#indels$

Primer na slici 5.19.

A T - G T T A T A
 A T C G T - C - C
 $+1+1-2+1+1-2-3-2-3 = -7$

	A	C	G	T	-		A	C	G	T	-	
A	+1	$-\mu$	$-\mu$	$-\mu$	$-\sigma$		A	+1	-3	-5	-1	-3
C	$-\mu$	+1	$-\mu$	$-\mu$	$-\sigma$		C	-4	+1	-3	-2	-3
G	$-\mu$	$-\mu$	+1	$-\mu$	$-\sigma$		G	-9	-7	+1	-1	-3
T	$-\mu$	$-\mu$	$-\mu$	+1	$-\sigma$		T	-3	-5	-8	+1	-4
-	$-\sigma$	$-\sigma$	$-\sigma$	$-\sigma$			-	-4	-2	-2	-1	

Slika 5.19

Navedimo rekurentnu relaciju dinamičkog programiranja kod grafa poravnjanja. Počinjemo od:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight of edge } \downarrow \text{ into } (i,j) \\ s_{i,j-1} + \text{weight of edge } \rightarrow \text{ into } (i,j) \\ s_{i-1,j-1} + \text{weight of edge } \searrow \text{ into } (i,j) \end{cases}$$

Odnosno,

$$s_{i,j} = \max \left\{ \begin{array}{l} s_{i-1,j} - \sigma \\ s_{i,j-1} - \sigma \\ s_{i-1,j-1} + 1, v_i = w_j \\ s_{i-1,j-1} - \mu, v_i \neq w_j \end{array} \right.$$

A uz pomoc funkcije `score()` može se zapisati i kao:

5.7.1 Globalno poravnanje

Problem 6 (Problem globalnog poravnjanja). Naći poravnanje sa najvišim skorom između dve niske za datu matricu skora.

Ulaz: Niske v i w, kao i matrica skora score

Izlaz: Poravnanje niski v i w čiji je skor poravnjanja (prema matrici skora) maksimalan od svih mogućih poravnjanja v i w.

Šta bi bili homeobox geni?

- Dva gena u različitim vrstama mogu biti slična u kratkim, konzervativnim regionima, a različita u ostalim delovima.

	A	C	G	T	-
A	+1	-3	-5	-1	-3
C	-4	+1	-3	-2	-3
G	-9	-7	+1	-1	-3
T	-3	-5	-8	+1	-4
-	-4	-2	-2	-1	

$s_{i,j} = \max \begin{cases} s_{i-1,j} + score(v_i, -) \\ s_{i,j-1} + score(-, w_j) \\ s_{i-1,j-1} + score(v_i, w_j) \end{cases}$

Slika 5.20

- Homeobox geni sadrže kratak region homeodomena koji je čvrsto konzerviran među različitim vrstama.
- Globalno poravnjanje može da propusti nalaženje homeodomena jer pokušava da poravna sekvene u celosti.

Uporedimo sledeća dva poravnanja:

GCC-C-A GT--TATGT-CAGGGGGCACG--A-GCATGCAGA-
GCCGCC-GTCGT-T-TTCAG---CA-GTTATG--T-CAGAT

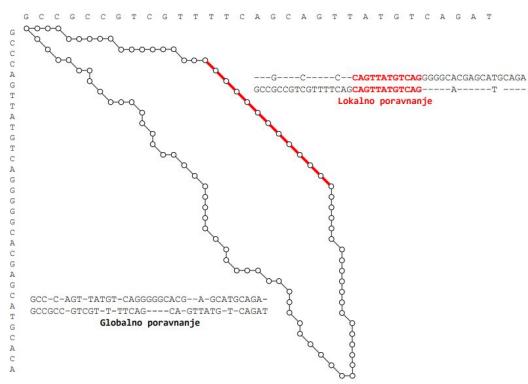
Slika 5.21

$$\text{score} = 22(\text{matches}) - 2(\text{indent}) = 2$$

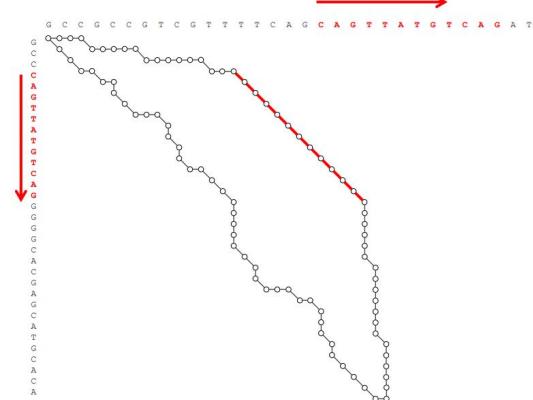
-----G-----C-----C-----CAGTTATGTCA GGGGCA CGAGCATGCAGA
GCCGCCGT CGTTTT CAGCAGTTATGTCAG-----A-----T-----

Slika 5.22

$$\text{score} = 17 \text{ (matches)} - 30 \text{ (indent)} = -13$$



Slika 5.23



Slika 5.24

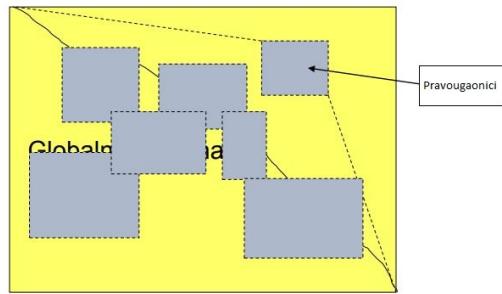
Ako se zapitamo koje od njih je bolje, iz priloženog zaključujemo da je to lokalno poravnjanje.

5.7.2 Lokalno poravnjanje

Lokalno poravnamo kao globalno poravnanje u pravougaoniku, pogledajmo sliku 5.26.



Slika 5.25



Slika 5.26

Da bismo dobili lokalno poravnanje potrebno je da izračunamo globalno poravnanje u okviru svakog pravougaonika.

Algoritam globalnog poravnanja ponovićemo između svaka dva čvora, ne samo između početnog (source) i krajnjeg (sink).

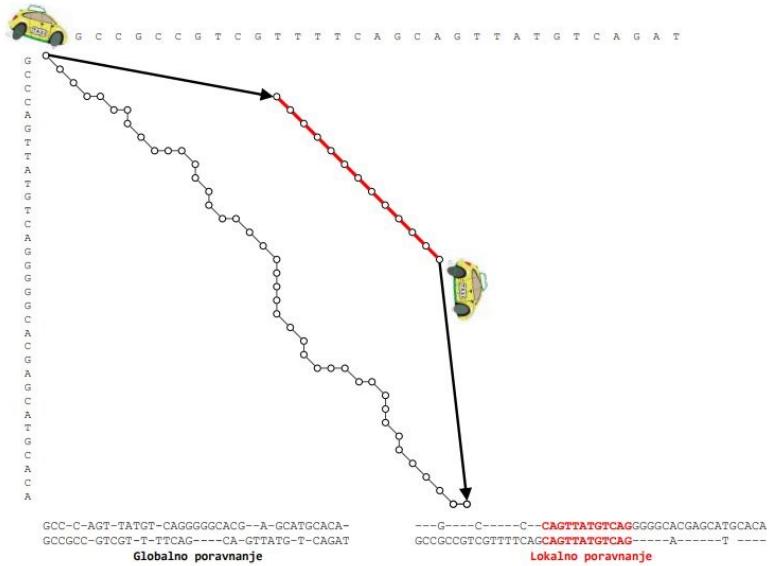
Stoga, broj ponavljanja algoritma će biti $\#nodes^2$ puta.

Problem 7 (Problem lokalnog poravnanja). *Naći lokalno poravnanje najvećeg skora između dve niske.*

Uzorak: Niske v i w , kao i matrica skora score

Izlaz: Podnische niske v i w čije je globalno poravnanje (prema matrici skora) maksimalno među svim globalnim poravnanjima svih podniski niske v i w .

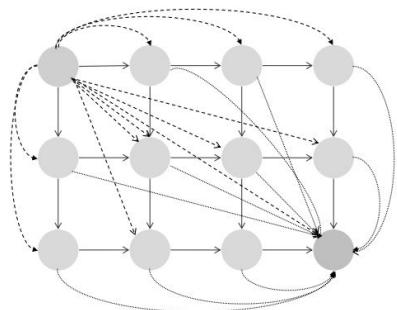
Zamislimo da postoji taksi koji bi nas besplatno vozio do tačke početka lokalnog poravnanja, i od tačke završetka lokalnog poravnanja pa do kraja. Na taj način ne bismo skupili negativne poene, već samo pozitivne. Ovakva vožnja nam daje samo skor lokalnog poravnanja kao što smo i želeli. [5.27](#)



Slika 5.27: Besplatne taksi vožnje

Konstruišimo Menhetn graf za problem lokalnog poravnjanja:

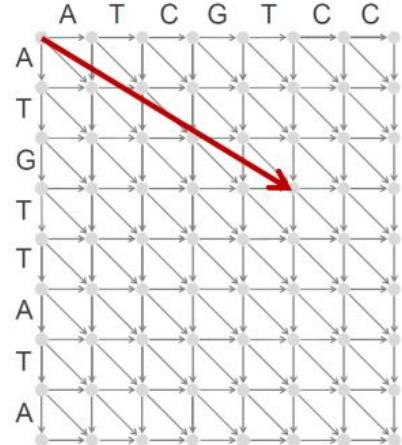
Kako bi izgledao Menhetn graf za nas problem?
 Dodamo grane težine 0 od (0,0) do svakog čvora, i od svakog čvora do (n,m).
 Ukupan broj dodatih grana je $O(|v| + |w|)$, pa algoritam ostaje brz.



Slika 5.28: Menhetn graf za lokalno poravnjanje

Problem rešavamo dinamičkim programiranjem:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight of edge } \downarrow \text{ into } (i,j) \\ s_{i,j-1} + \text{weight of edge } \rightarrow \text{ into } (i,j) \\ s_{i-1,j-1} + \text{weight of edge } \searrow \text{ into } (i,j) \end{cases}$$



Slika 5.29: weight of (0,0) into (i,j) = 0

5.8 Kažnjavanje insercija i delecija u poravnanju sekvenci

5.8.1 Kažnjavanje praznina

- U globalnom poravnanju je fiksna kazna σ bila dodeljena svakom indelu
- Međutim, ova fiksna kazna može biti preoštra kod lokalnog poravnanja kada možemo imati 100 uzastopnih indela.
- Niz od k uzastopnih indela često predstavlja jedan isti evolucioni događaj, ne k različitim, slika 5.30

dve praznine (niži skor)	GATCCAG	GATCCAG jedna praznina (viši skor)
	GA-C-AG	GA--CAG

Slika 5.30

5.8.2 Adekvatnije kazne za praznine

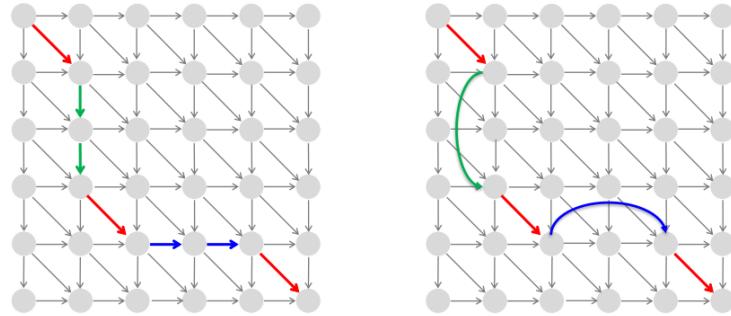
Afina kazna za praznine za prazninu dužine k: $\sigma + \epsilon * (k - 1)$

- σ kazna za otvaranje praznine
- ϵ kazna za proširenje praznine
- $\sigma > \epsilon$, jer otvaranje praznine treba kazniti više nego njeno proširenje

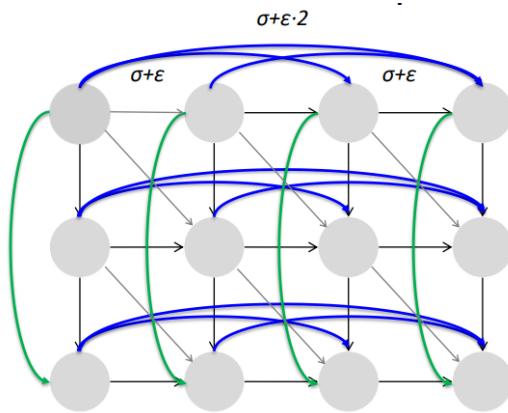
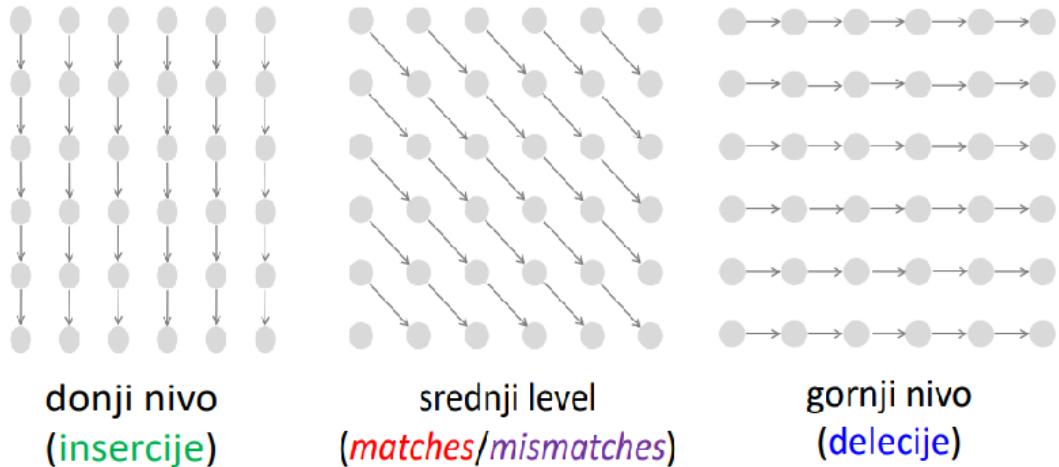
U slici 5.31 prikayano je modelovanje afinih kazni za praznine pomoću dugih grana.

5.8.3 Izgradnja Menhetn grafa sa afnim kaznama za praznine

- Vremenska složenost je direktno proporcionalna broju grana, zbog čega želimo da smanjimo broj grana u grafu (trenutno je $O(n^3)$ 5.32)
- Jedan način za smanjivanje broja grana je povećanje broja čvorova u grafu



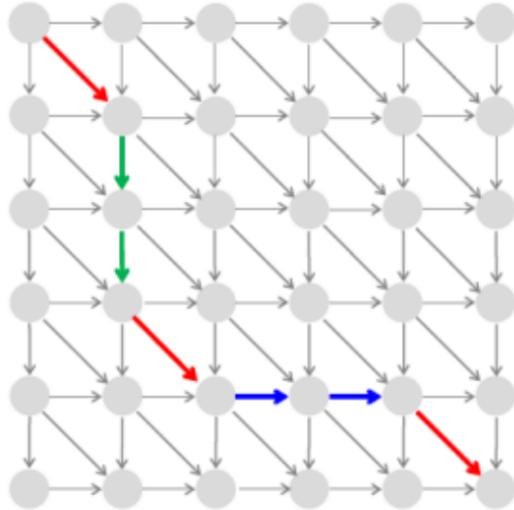
Slika 5.31: Modelovanje afinih kazni za praznine pomoću dugih grana

Slika 5.32: Dodali smo $O(n^3)$ grana

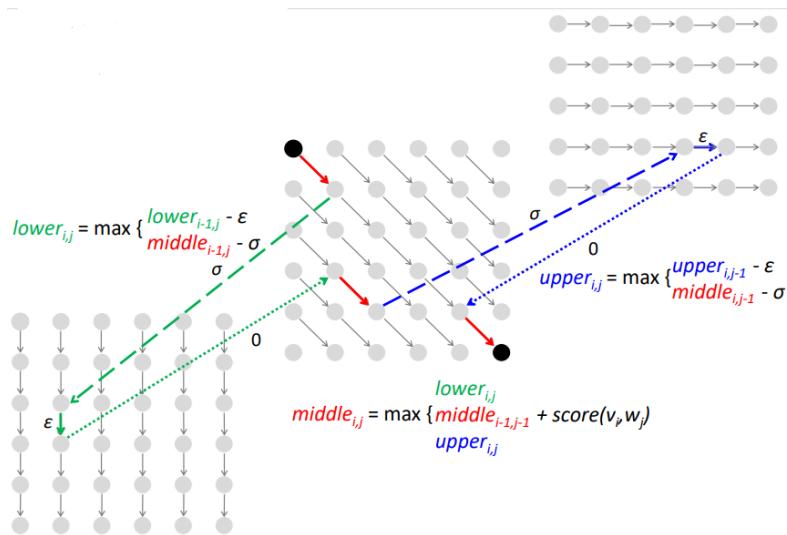
Slika 5.33: Podela Menhetna grafa na 3 nivoa

- Zato delimo Menhetn graf na tri nivoa 5.33

Ako imamo putanju poput one na slici 5.34, kako da je predstavimo pomoću Menhetn grafa na 3 nivoa? Rešenje je prikazano na slici 5.35



Slika 5.34: Kako predstaviti pomoću Menhetn grafa na 3 nivoa?



Slika 5.35: Simulacija Menhetn grafa na 3 nivoa

5.9 Prostorno efikasno poravnjanje sekvenci

Zapitajmo se sledeće:
Da li možemo poravnati NPR sintetaze iz dve različite bakterije?

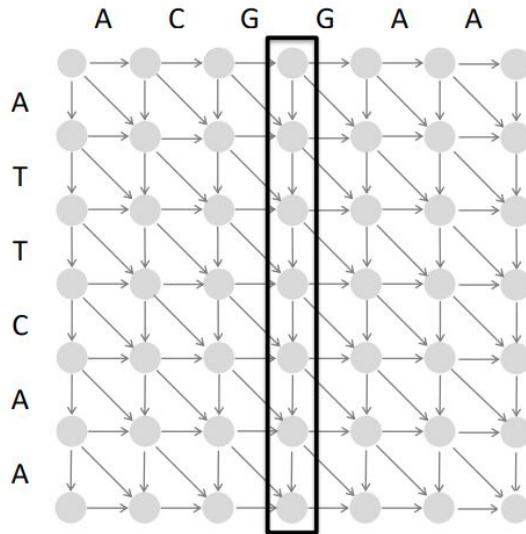
Uzmimo u obzir sledeće činjenice:

- NPR sintetaze su obično veoma dugi proteini, približno 20 000 aminokiselina
- Vremenska složenost poravnjanja je približno jednaka broju ivica (#edges), odnosno kvadratna
- Prostorna složenost poravnjanja je približno jednaka broju čvorova (#nodes), odnosno kvadratna
- **Memorija je često usko grlo pri poređenju dugih sekvenci**

Iz prethodnog zaključujemo da nam prostorna složenost pravi problem i da bi za prosečan racunar ovo bilo nemoguće da izračuna. Stoga, potreban nam je drugi pristup koji će nam rešiti problem. Potreban nam je algoritam koji zahteva linearnu prostornu složenost i udvostručenu vremensku složenost. U te svrhe najčešće se koristi algoritam koji radi po principu podeli-podvladaj.

Uvedimo nove pojmove:

- Srednja kolona poravnanja (middle) = $\#columns/2$, slika 5.36
- Srednji čvor poravnanja - čvor u preseku putanje optimalnog poravnanja i srednje kolone, slika 5.37



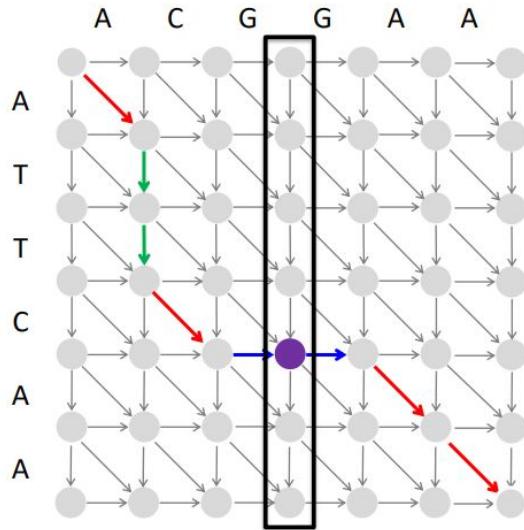
Slika 5.36: Srednja kolona poravnanja

Koristeći navedene pojmove, demonstrirajmo algoritam **Podeli-pa-vladaj** za poravnanje sekvenci, slika 5.38:

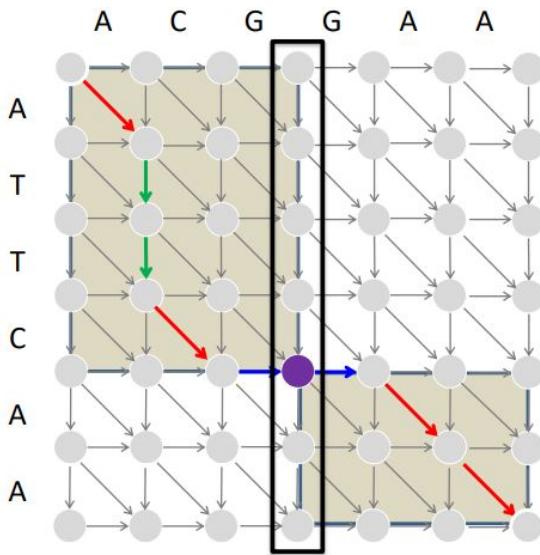
```

1 AlignmentPath(source, sink)
2     find middleNode
3     AlignmentPath(source, middleNode)
4     AlignmentPath(middle, sink)

```



Slika 5.37: Srednji čvor poravnjanja



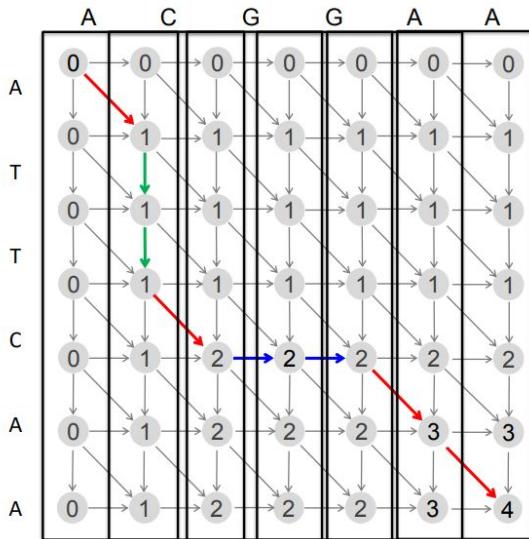
Slika 5.38: Srednja kolona poravnjanja

5.9.1 Prostorna složenost

Za nalaženje najduže putanje u grafu poravnanja traži se čuvanje svih putokaza za - $O(nm)$ prostora.

Međutim, za nalaženje dužine najduže putanje u grafu poravnanja ne traži se čuvanje svih putokaza - $O(n)$ prostora, slika 5.39

Dobijamo da je prostor potreban za algoritam $2 * n \sim O(n)$, odnosno, linearan.

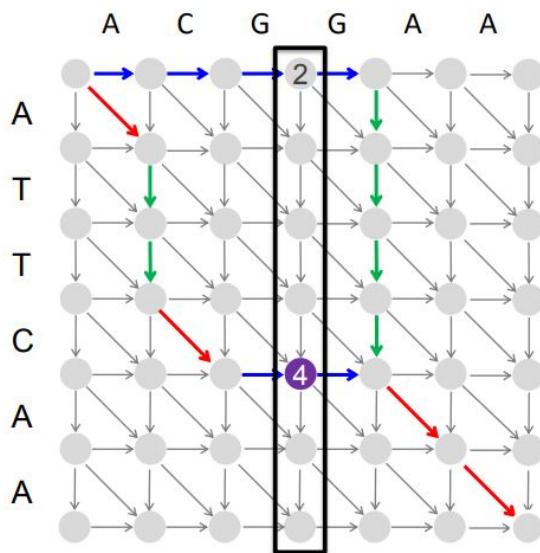


Slika 5.39: Recikliranje prostora za kolone u grafu poravnjanja

5.9.2 Vremenska složenost

Neka je **i-putanja** najduža putanja od svih putanja koja posećuje i-ti čvor u srednjoj koloni i neka je $length(i)$ dužina i-putanije.

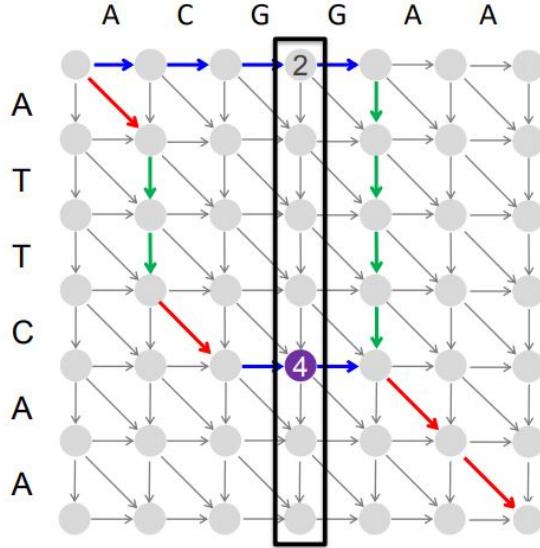
Na slici 5.41 vidimo da je $length(0) = 2$ i $length(4) = 4$.

Slika 5.40: Računanje $length(i)$

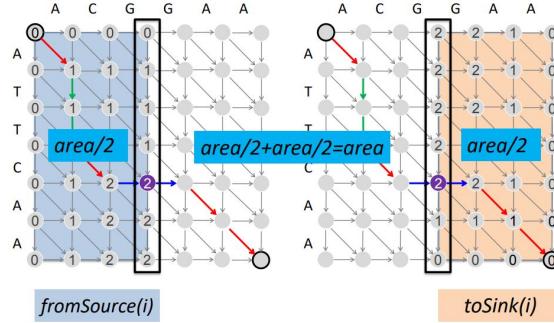
$length(i)$ možemo računati na sledeći način:

$$length(i) = fromSource(i) + toSink(i)$$

Na slici 5.42 vidimo koliko je potrebno vremena za nalaženje srednjeg čvora - čak $O(nm)$

Slika 5.41: Računanje $\text{length}(i)$

vremena za nalaženje samo jednog čvora!



Slika 5.42: Računanje vremena za nalaženje srednjeg čvora

Svaki problem se može rešiti u vremenu proporcionalnom broju grana tj. površini koju zauzima.

Vreme potrebno za rešavanje naredna dva potproblema: $\text{area}/4 + \text{area}/4 = \text{area}/2$, znači $O(nm + nm/2)$ vremena za nalaženje 3 čvora.

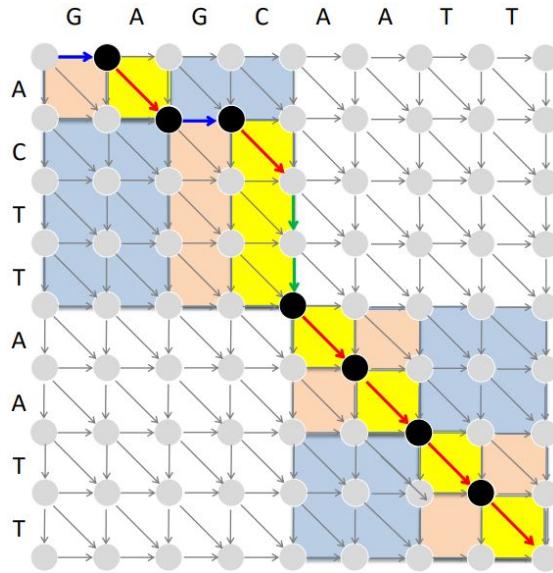
Dakle, vreme potrebno za nalaženje svih čvorova je: $\text{area} + \text{area}/2 + \text{area}/4 + \text{area}/8 + \dots < 2 * \text{area}$! Slika 5.43 vizuelno prikazuje vreme potrebno za nalaženje 7 tačaka.

Pokazali smo da je vremenska složenost $2 * n * m \sim O(n * m)$.

5.10 Višestruko poravnanje sekvenci

5.10.1 Od dvostrukog do višestrukog poravnjanja

- Do sada su u poravnjanju učestvovalo samo dve sekvene.



Slika 5.43: Računanje vremena za nalaženje 7 tačaka

- Slaba sličnost između dve sekvene postaje značajna ako je prisutna i u drugim sekvencama
- Višestruka poravnanja mogu otkriti suptilne sličnosti koje dvostruka poravnanja ignoriraju

5.10.2 Poravnjanje tri A-domena

Na slici 5.44 je prikazano poravnjanje tri A-domena

```

YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGIITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIIDVIAFRKMYGHTE-FINHYGPTEATIGA
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI-YEQKLDISQLQILIVGSDCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNNGGTVVCIDYYTTDIKALEAVFKQHHIRGAMLPPALLKQCLVSA---PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS

```

↓

```

YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGIITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIIDVIAFRKMYGHTE-FINHYGPTEATIGA
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI-YEQKLDISQLQILIVGSDCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNNGGTVVCIDYYTTDIKALEAVFKQHHIRGAMLPPALLKQCLVSA---PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS

```

↓

```

YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGIITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIIDVIAFRKMYGHTE-FINHYGPTEATIGA
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI-YEQKLDISQLQILIVGSDCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNNGGTVVCIDYYTTDIKALEAVFKQHHIRGAMLPPALLKQCLVSA---PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS

```

Slika 5.44: Poravnavanja 3 A-domena

5.10.3 Generalizacija dvostrukog na višestroko poravnjanje

- Poravnanje 2 sekvene je matrica od 2 reda

- Poravnjanje 3 sekvence je matrica od 3 reda (slika 5.45)

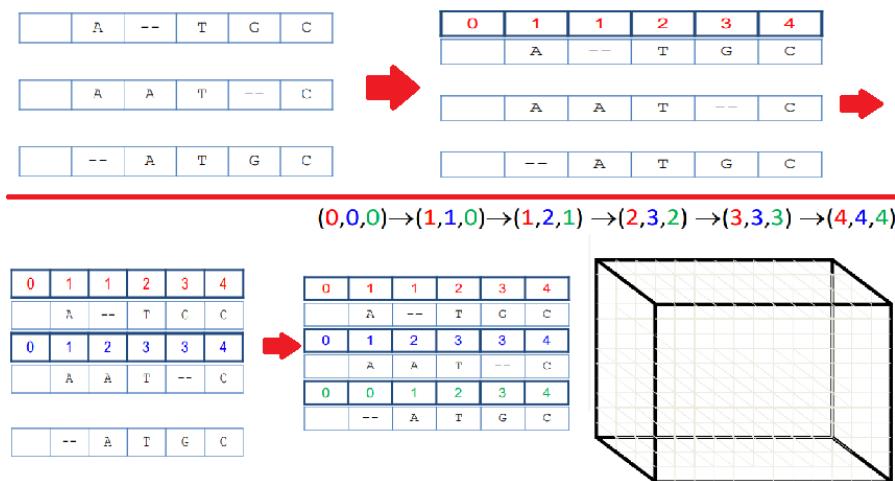
A	T	-	G	C	G	-
A	-	C	G	T	-	A
A	T	C	A	C	-	A

Slika 5.45

- Funkcija skora treba da dodeljuje visok skor poravnajima sa konzerviranim kolonama

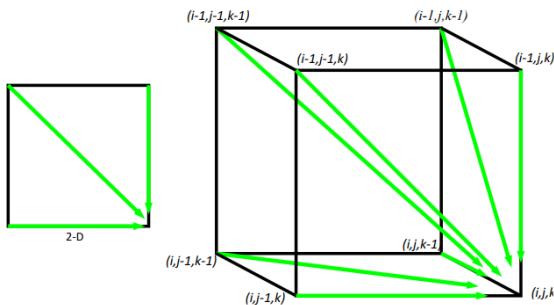
5.10.4 Poravnjanja = 3-D putanje

Poravnjanje sekvenci ATGC, AATC i ATGC 5.46



Slika 5.46: Poravnjanje sekvenci ATGC, AATC i ATGC

5.10.5 2-D poravnjanje u odnosu na 3-D poravnjanje



Slika 5.47

2-D poravnjanje u odnosu na 3-D poravnjanje prikazano na slici 5.47.

5.10.6 Rekurentna relacija dinamičkog programiranja za višestruko poravnanje

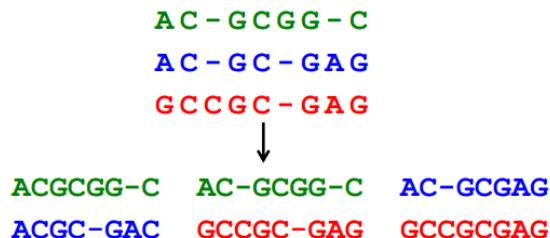
$$s_{i,j,k} = \max \begin{cases} s_{i-1,j-1,k-1} + \delta(v_i, w_j, u_k) \\ s_{i-1,j-1,k} + \delta(v_i, w_j, -) \\ s_{i-1,j,k-1} + \delta(v_i, -, u_k) \\ s_{i,j-1,k-1} + \delta(-, w_j, u_k) \\ s_{i-1,j,k} + \delta(v_i, -, -) \\ s_{i,j-1,k} + \delta(-, w_j, -) \\ s_{i,j,k-1} + \delta(-, -, u_k) \end{cases}$$

$\delta(x, y, z)$ - element 3-D matrice skora

5.10.7 Vremenska složenost dinamičkog algoritma za višestruko poravnanje

- Kao kod dvostrukog poravnanja, vremenska složenost je proporcionalna broju grana $O(\#edges)$
- Za 3 sekvence n , vremenska složenost je proporcionalna $7n^3$
- Za poravnanje k sekvenici, potrebno je izgraditi k -dimenzionalni Menhetn graf sa:
 - n^k čvorova
 - većina čvorova će imati $2^k - 1$ ulaznih grana
 - Vremenska složenost: $O(2^k n^k)$

Višestruko poravnanje uključuje i dvostruko poravnanje, slika 5.48



Slika 5.48

5.10.8 Da li se višestruko poravnanje može izgraditi iz dvostrukog?

Za dati skup proizvoljnih dvostrukih poravnanja, možemo li konstruisati višestruko poravnanje iz kog su izvedeni (slika 5.49)?

AAAATTTT----	----- AAAATTTT	TTTTGGGG----
---- TTTTGGGG	GGGGAAAA ----	---- GGGGAAAA

Slika 5.49

-	A	G	G	C	T	A	T	C	A	C	C	T	G
T	A	G	-	C	T	A	C	C	A	-	-	-	G
C	A	G	-	C	T	A	C	C	A	-	-	-	G
C	A	G	-	C	T	A	T	C	A	C	-	G	GG
C	A	G	-	C	T	A	T	C	G	C	-	G	GG
A	0	1	0	0	0	0	1	0	0	.8	0	0	0
C	.6	0	0	0	1	0	0	.4	1	0	.6	.2	0
G	0	0	1	.2	0	0	0	0	0	.2	0	0	.4
T	.2	0	0	0	0	1	0	.6	0	0	0	0	.2
-	.2	0	0	.8	0	0	0	0	0	.4	.8	.4	0

Slika 5.50

5.10.9 Profilna reprezentacija višestrukog poravnjanja

Na slici 5.50 prikazana profilna reprezentacija višestrukog poravnjanja.

- Do sada smo poravnavali **sekvencu u odnosu na sekvencu**.
 - Možemo li poravnati **sekvencu u odnosu na profil**?
 - Možemo li poravnati **profil u odnosu na profil**?

5.10.10 Višestruko poravnjanje: pohlepni pristup

- Izabratи najsličnije sekvence i kombinovati ih u profil
- Tako bismo smanjili broj sekvenci sa k na k-2 i jedan profil
- Iterirati

Primer pohlepnog pristupa:

- Sekvence: GATTCA, GTCTGA, GATATT, GTCAGC
- 6 dvostrukih poravnjanja (**match+1, indels i mismatches -1**) slika 5.51

<i>s₂</i>	GTCTGA	<i>s₁</i>	GATTCA--
<i>s₄</i>	GTCAGC	(score = 2)	<i>s₄</i> G-T-CAGC (score = 0)
<i>s₁</i>	GAT-TCA	<i>s₂</i>	G-TCTGA
<i>s₂</i>	G-TCTGA	(score = 1)	<i>s₃</i> GATAT-T (score = -1)
<i>s₁</i>	GAT-TCA	<i>s₃</i>	GAT-ATT
<i>s₃</i>	GATAT-T	(score = 1)	<i>s₄</i> G-TCAGC (score = -1)

Slika 5.51

- Pošto su *s₂* i *s₄* najbliže, od njih pravimo profil

$$\left. \begin{array}{l} s_2 \text{ GTCTGA} \\ s_4 \text{ GTCAGC} \end{array} \right\} s_{2,4} = \text{GT Ct/aGa/c}$$

Slika 5.52

- Novi skup od 3 sekvence za poravnjanje:

s₁ GATTCA
s₃ GATATT
s_{2,4} GT Ct/aGa/c

5.11 Zadaci sa vežbi

U nastavku će biti predstavljeni zadaci sa vežbi na kursu rađeni u programskom jeziku Python.

5.11.1 Manhattan Tourist

```

1 def manhattan_tourist(n, m, down, right):
2     s = [[0 for j in range(m)] for i in range(n)]
3
4     backtrack = [[[0,0) for j in range(m)] for i in range(n)]
5
6     backtrack[0][0] = (-1, -1)
7
8     for i in range(1,n):
9         s[i][0] = s[i-1][0] + down[i][0]
10        backtrack[i][0] = (i-1, 0)
11
12    for j in range(1,m):
13        s[0][j] = s[0][j-1] + right[0][j]
14        backtrack[0][j] = (0, j-1)
15
16    for i in range(1, n):
17        for j in range(1, m):
18
19            to_down = s[i-1][j] + down[i][j]
20            to_right = s[i][j-1] + right[i][j]
21
22            if to_down > to_right:
23                backtrack[i][j] = (i-1, j)
24                s[i][j] = to_down
25            else:
26                backtrack[i][j] = (i, j-1)
27                s[i][j] = to_right
28
29    i = n-1
30    j = m-1
31    while backtrack[i][j] != (-1,-1):
32        print(backtrack[i][j])
33        i = backtrack[i][j][0]
34        j = backtrack[i][j][1]
35
36    return s[n-1][m-1]
37
38 def main():
39     down = [[0, 0, 0, 0],
40             [0, 1, 2, 1],
41             [0, 1, 1, 1],
42             [0, 1, 1, 1]]
43
44     right = [[0, 0, 0, 1],
45               [0, 3, 5, 1],
46               [0, 1, 0, 1],
47               [0, 1, 0, 1]]
```

```

48     print(manhattan_tourist(3, 3, down, right))
49
50 if __name__ == "__main__":
51     main()

```

5.11.2 LCS Backtrack

```

1 def LCSBacktrack(v, w):
2     n = len(v)
3     m = len(w)
4     s = [[0 for j in range(m + 1)] for i in range(n + 1)]
5
6     backtrack = [[(-1,-1) for j in range(m + 1)] for i in range(n + 1)]
7
8     for i in range(1, n + 1):
9         backtrack[i][0] = (i-1, 0)
10
11    for j in range(1, m + 1):
12        backtrack[0][j] = (0, j-1)
13
14    for i in range(1, n + 1):
15        for j in range(1, m + 1):
16
17            s[i][j] = max(s[i-1][j], s[i][j-1], s[i-1][j-1] + int(v[i-1] == w[j
18            ↪ -1]))
19
20            if s[i][j] == s[i-1][j]:
21                backtrack[i][j] = (i-1, j)
22            elif s[i][j] == s[i][j-1]:
23                backtrack[i][j] = (i, j-1)
24            else:
25                backtrack[i][j] = (i-1, j-1)
26
27    i = backtrack[n][m][0]
28    j = backtrack[n][m][1]
29
30    lcs = ""
31
32    if i == n-1 and j == m - 1:
33        lcs = v[n-1]
34
35    while not (i == 0 and j == 0):
36        if backtrack[i][j] == (i-1, j-1):
37            lcs = v[i-1] + lcs
38
39        i = backtrack[i][j][0]
40        j = backtrack[i][j][1]
41
42    print(lcs)
43
44    return s[n][m]

```

```

45 def main():
46     v = "abcd"
47     w = "dabe"
48
49     print(LCSBacktrack(v,w))
50
51 if __name__ == "__main__":
52     main()

```

5.11.3 Global Alignment

```

1 GAP_PENALTY = -2
2 MISSMATCH = 0
3 MATCH = 1
4
5 def match_score(c1, c2):
6     if c1 == c2:
7         return MATCH
8     else:
9         return MISSMATCH
10
11 def global_alignment(v, w):
12     n = len(v)
13     m = len(w)
14
15     backtrack = [[(-1, -1) for j in range(m + 1)] for i in range(n + 1)]
16     s = [[0 for j in range(m + 1)] for i in range(n + 1)]
17
18     for i in range(1, n + 1):
19         s[i][0] = s[i-1][0] + GAP_PENALTY
20         backtrack[i][0] = (i-1, 0)
21
22     for j in range(1, m + 1):
23         s[0][j] = s[0][j-1] + GAP_PENALTY
24         backtrack[0][j] = (0, j-1)
25
26     for i in range(1, n + 1):
27         for j in range(1, m + 1):
28             s[i][j] = max(
29                 s[i-1][j] + GAP_PENALTY,
30                 s[i][j-1] + GAP_PENALTY,
31                 s[i-1][j-1] + match_score(v[i-1], w[j-1])
32             )
33
34         if s[i][j] == s[i-1][j] + GAP_PENALTY:
35             backtrack[i][j] = (i-1, j)
36
37         elif s[i][j] == s[i][j-1] + GAP_PENALTY:
38             backtrack[i][j] = (i, j-1)
39         else:
40             backtrack[i][j] = (i-1, j-1)
41
42     v_p = ""

```

```

43     w_p = ""
44
45     i = n
46     j = m
47
48     while (i,j) != (0,0):
49         if backtrack[i][j] == (i-1, j-1):
50             v_p = v[i-1] + v_p
51             w_p = w[j-1] + w_p
52
53         elif backtrack[i][j] == (i-1, j):
54             v_p = v[i-1] + v_p
55             w_p = '-' + w_p
56
57         else:
58             v_p = '-' + v_p
59             w_p = w[j-1] + w_p
60
61     (i,j) = backtrack[i][j]
62
63     print(v_p)
64     print(w_p)
65
66     return s[n][m]
67
68 def main():
69     v = "AAATTTGGGCCCGGGAAATTTCCTT"
70     w = "GGGCCCTT"
71
72     print(global_alignment(v, w))
73
74 if __name__ == "__main__":
75     main()

```

5.11.4 Local Alignment

```

1 def local_alignment(string_1, string_2):
2     DP = [[0 for j in range(len(string_2) + 1)] for i in range(len(string_1) +
3         1)]
4
5     for i in range(len(string_1) + 1):
6         DP[i][0] = 0
7
8     for j in range(len(string_2) + 1):
9         DP[0][j] = 0
10
11    for i in range(1, len(string_1) + 1):
12        for j in range(1, len(string_2) + 1):
13            DP[i][j] = max(0, DP[i-1][j] - 2, DP[i][j-1] - 2, DP[i-1][j-1] +
14                int(string_1[i-1] == string_2[j-1]))
15
16    maximum = 0

```

```

16
17     for i in range(len(DP)):
18         for j in range(len(DP[i])):
19             if DP[i][j] > maximum:
20                 maximum = DP[i][j]
21
22     return maximum
23
24
25
26 def main():
27     string_1 = 'ACGTGCTCG'
28     string_2 = 'AATGCTCT'
29
30     print(local_alignment(string_1, string_2))
31
32 if __name__ == "__main__":
33     main()

```

5.11.5 Edit Distance

```

1 def edit_distance(v, w):
2     n = len(v)
3     m = len(w)
4
5     s = [[0 for j in range(m + 1)] for i in range(n + 1)]
6     backtrack = [[(-1, -1) for j in range(m + 1)] for i in range(n + 1)]
7
8     for i in range(1, n + 1):
9         s[i][0] = i
10        backtrack[i][0] = (i-1, 0)
11
12    for j in range(1, m + 1):
13        s[0][j] = j
14        backtrack[0][j] = (0, j-1)
15
16    for i in range(1, n + 1):
17        for j in range(1, m + 1):
18            s[i][j] = min(s[i-1][j] + 1, s[i][j-1] + 1, s[i-1][j-1] + int(v[i]
19            ↪ -1] != w[j-1]))
20
21            if s[i][j] == s[i-1][j] + 1:
22                backtrack[i][j] = (i-1, j)
23
24            elif s[i][j] == s[i][j-1] + 1:
25                backtrack[i][j] = (i, j-1)
26            else:
27                backtrack[i][j] = (i-1, j-1)
28
29    v_p = ""
30    w_p = ""
31    i = n

```

```
32     j = m
33
34     while (i,j) != (0,0):
35         if backtrack[i][j] == (i-1, j-1):
36             v_p = v[i-1] + v_p
37             w_p = w[j-1] + w_p
38
39         elif backtrack[i][j] == (i-1, j):
40             v_p = v[i-1] + v_p
41             w_p = '-' + w_p
42
43         else:
44             v_p = '-' + v_p
45             w_p = w[j-1] + w_p
46
47         (i,j) = backtrack[i][j]
48
49     print(v_p)
50     print(w_p)
51
52     return s[n][m]
53
54 def main():
55     v = "AAATTTGGGCCCGGGAAATTCCC"
56     w = "AAACCCCTTGGGCCCTTAAACCC"
57
58     print(edit_distance(v, w))
59
60 if __name__ == "__main__":
61     main()
```

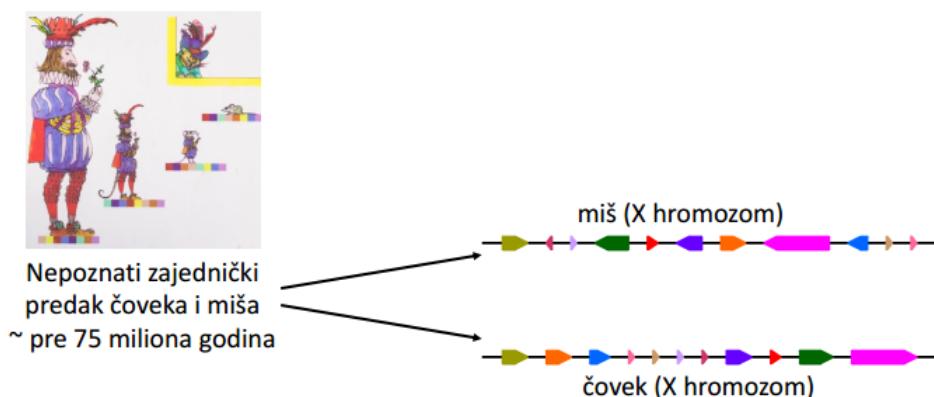

Glava 6

Postoje li osetljivi delovi u ljudskom genomu?

6.1 Transformacija čoveka u miša

Koji blokovi genoma su slični i kako da ih nademo?

Kakav bi bio evolucijski scenario za transformisanje jednog genoma u drugi?

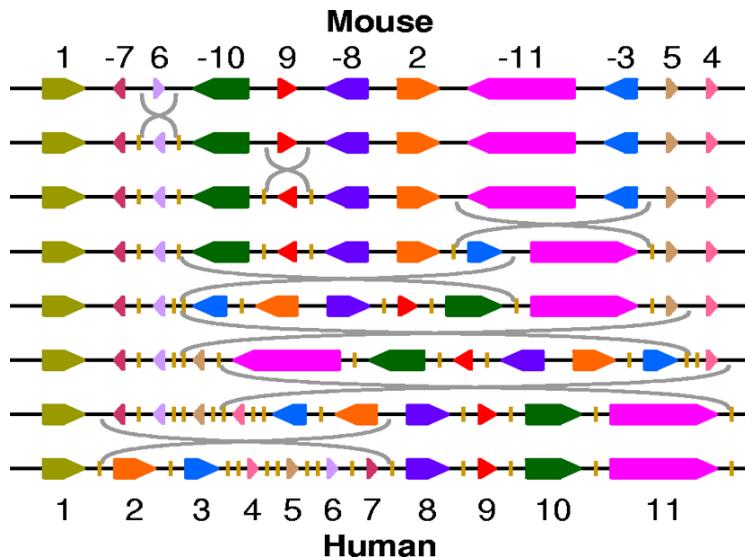


Slika 6.1: X hromozom miša i čoveka

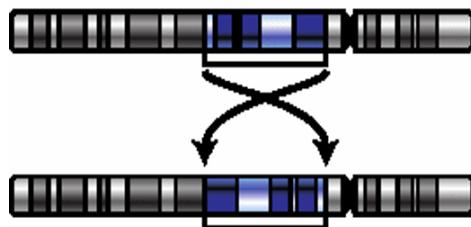
Kada je Šarl Pero opisao transformaciju čoveka u miša u delu "Mačak u čizmama", jedva je mogao očekivati da će 3 veka kasnije istraživanje pokazati da su ljudski i mišiji genom iznenađujuće slični. Na slici 6.1 možemo videti X hromozom miša i čoveka.

Zapravo, ako bismo isekli 23 ljudska hromozoma na 280 delova i pomerali ove fragmente DNK, a zatim zlepili delove zajedno u novom redosledu, formiralo bi se 20 mišijih hromozoma. Međutim, evolucija nije koristila samo operaciju *cut-and-paste*, već manju promenu poznatu kao **preuređenje genoma**, što će biti naš fokus u ovom poglavlju.

Blokovi sintenije



Slika 6.2: Blokovi sintenije



Slika 6.3: Blokovi sintenije

Svaki od jedanaest obojenih segmenata na slici 6.2 predstavlja blok sličnih gena i naziva se **blok sintenije** (pogledati i sliku 6.3). U nastavku će biti objašnjeno kako se izgrađuju blokovi sintenije i šta označavaju levi i desni pravac blokova.

Slika 6.2 prikazuje niz 7 promena koje transformišu mišiji X hromozom u ljudski X hromozom. Nažalost, ovaj niz od 7 promena predstavlja samo jedan od 1.070 različitih scenarija od 7 promena koji transformišu X hromozom miša u X hromozom čoveka.

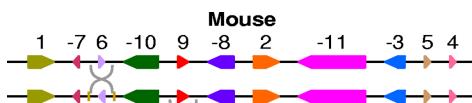
Možemo li pretvoriti X hromozom miša u ljudski X hromozom koristeći samo šest promena?

Bez obzira na to koliko promena razdvaja mišije i ljudske X hromozome, promene moraju biti *retki genomske događaji*. Zapravo, obično preuređeni genomi uzrokuju smrt ili sterilnost mutiranog organizma, čime se sprečava prenošenje

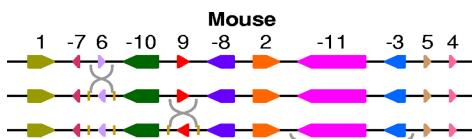
preuređenja na narednu generaciju. Međutim, mali deo preuređenja genoma može imati pozitivan efekat na preživljavanje i propagirati se kroz vrstu kao rezultat prirodne selekcije. Kada stanovništvo postane izolovano od ostatka njene vrste dovoljno dugo, preuređenja mogu čak stvoriti i novu vrstu.

Promenu mozemo zamisliti kao prekid genoma sa obe strane hromozomskog intervala, pomeranje intervala, a zatim lepljenje rezultujućih segmenata u novom redosledu.

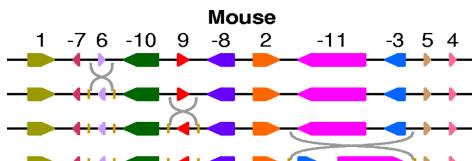
Na slikama 6.4, 6.5, 6.6, 6.7, 6.8 i 6.9 vidimo pojedinačno svaku od promena.



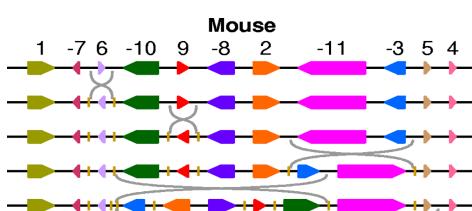
Slika 6.4: Promena 1



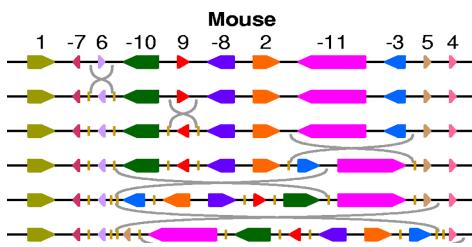
Slika 6.5: Promena 2



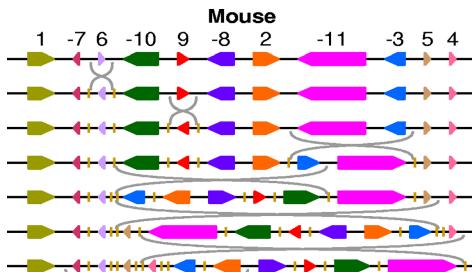
Slika 6.6: Promena 3



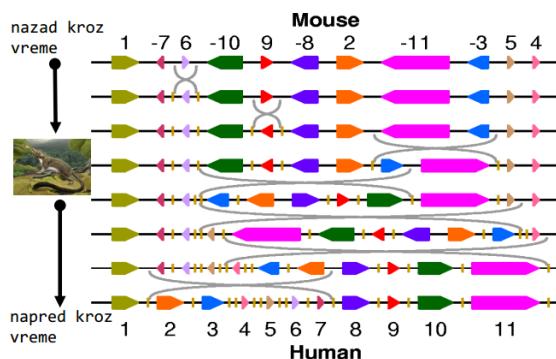
Slika 6.7: Promena 4



Slika 6.8: Promena 5



Slika 6.9: Promena 6



Slika 6.10: Završena transformacija

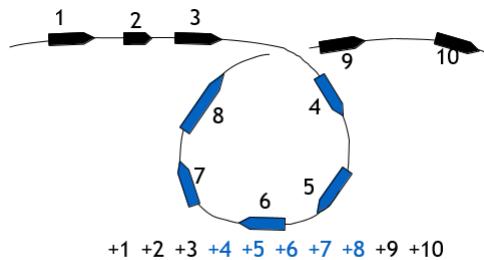
Slika 6.10 predstavlja transformaciju mišijeg X hromozoma u ljudski X hromozom sa sedam promena. Svaki blok sintenije je jedinstveno obojen i označen celim brojem između 1 i 11. Pozitivni ili negativni znak svakog celog broja ukazuje na smer sintenog bloka (pokazivanje desno ili levo, respektivno). Dva kratka vertikalna segmenta obeležavaju krajnje tačke obrnutog intervala u svakom preokretu.

Prepostavimo da je evolucijski scenario tačan i recimo peti blok sintenije od vrha predstavlja uređenje hromozoma pretka. Zatim su se desile prve četiri promene na evolucionom putu od miša do zajedničkog pretka, a poslednje tri promene su se desile na evolucionom putu od zajedničkog pretka ka coveku.

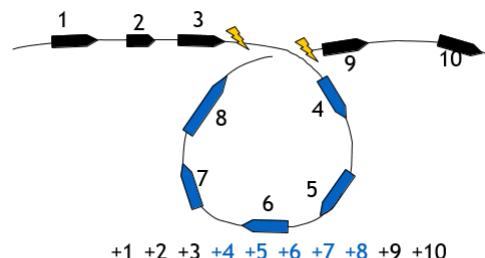
6.2 Sortiranje po promenama

Glavni problem je, kao sto je pomenuto u uvodu, nalaženje minimalnog broja promena koje omogućavaju transformaciju X hromozona miša u X hromozom čoveka.

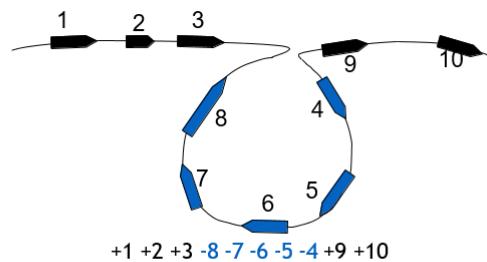
Možemo posmatrati niz blokova sintenije numerisanih kao na slici 6.11.



Slika 6.11: Blokovi sintenije

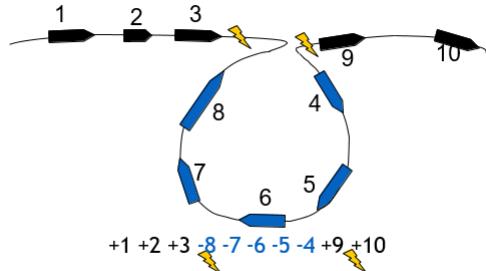


Slika 6.12



Slika 6.13

Nakon izvršene promene, dobijamo preuređen niz blokova sintenije u genomu to vidimo na slikama 6.12 i 6.13.



Slika 6.14: 2 tačke prekida

Promene u genomu su dovele do stvaranja dve tačke prekida koje predstavljaju poremećaj u redosledu gena u genomu (slika 6.14).

Posmatraćemo 2 scenarija sa različitim brojem promena. Na slici 6.15 vidimo scenario sa 5 promena, a na slici 6.16 sa 4 promene.

Step 0:	2	<u>-4</u>	-3	5	-8	-7	-6	1
Step 1:	2	3	4	5	<u>-8</u>	-7	-6	1
Step 2:	2	3	4	5	6	7	8	<u>1</u>
Step 3:	2	3	4	5	6	7	8	-1
Step 4:	<u>-8</u>	-7	-6	-5	-4	-3	-2	-1
Step 5:	1	2	3	4	5	6	7	8

Slika 6.15: Scenario sa 5 promena

Step 0:	2	<u>-4</u>	-3	5	-8	-7	-6	1
Step 1:	<u>2</u>	3	4	5	<u>-8</u>	-7	-6	1
Step 2:	<u>-5</u>	-4	-3	<u>-2</u>	<u>-8</u>	-7	-6	1
Step 3:	<u>-5</u>	-4	-3	<u>-2</u>	<u>-1</u>	6	7	8
Step 4:	1	2	3	4	5	6	7	8

Slika 6.16: Scenario sa 4 promene

Definicija 6.1. *Rastojanje premutacija* je najmanji broj promena potrebnih za transformisanje jedne premutacije u drugu.

Naredni problem koji posmatramo je **problem sortiranja po promenama** koji predstavlja izračunavanje rastojanja između date permutacije i identične permutacije (+1 +2 ... +n)

Ulaz: permutacija P

Izlaz: rastojanje između permutacije P i identične permutacije

Pohlepno sortiranje po promenama

Prva aproksimacija rastojanja između 2 permutacije je **pohlepno sortiranje po promenama**(primer je dat na slici 6.17).

```
(+1 -7 +6 -10 +9 -8 +2 -11 -3 +5 +4)
(+1 -2 +8 -9 +10 -6 +7 -11 -3 +5 +4)
(+1 +2 +8 -9 +10 -6 +7 -11 -3 +5 +4)
(+1 +2 +3 +11 -7 +6 -10 +9 -8 +5 +4)
(+1 +2 +3 -4 -5 +8 -9 +10 -6 +7 -11)
(+1 +2 +3 +4 -5 +8 -9 +10 -6 +7 -11)
(+1 +2 +3 +4 +5 +8 -9 +10 -6 +7 -11)
(+1 +2 +3 +4 +5 +6 -10 +9 -8 +7 -11)
(+1 +2 +3 +4 +5 +6 -7 +8 -9 +10 -11)
(+1 +2 +3 +4 +5 +6 +7 +8 -9 +10 -11)
(+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 -11)
(+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11)
```

Slika 6.17: Pohlepno sortiranje

Prvi korak je da se izvrši promena koji postavlja +1 na pravo mesto (na prvu poziciju), a zatim slede promene koji postavljaju +2 na drugu poziciju, i tako dalje. Na primer, element 1 je već na pravom mestu i ima ispravan znak (+) u X hromozomu miša, ali element 2 nije na ispravnom položaju. Element 1 možemo zadržati fiksirani i prenesti element 2 na pravi položaj jednom promenom. Još jedna promena je potrebna da bi element 2 imao ispravan znak.

Daljim iteriranjem postupka dovodimo sve veće elemente na njihove ispravne pozicije.

Definicija 6.2. Element k u permutaciji $P = (p_1, p_2, \dots, p_n)$ je **sortiran**, ako je $p_k = k$, a u suprotnom je **nesortiran**.

Definicija 6.3. Permutacija P je **k -sortirana**, ako je prvih $k-1$ elemenata sortirano, a k -ti element nesortiran.

Sledeći primer pokazuje da je pohlepno sortiranje loša aproksimacija rastojanja između dve permutacije(slika 6.18) jer je ponekad moguće naći mnogo jednostavniji način (slika 6.19).

```
(-6 +1 +2 +3 +4 +5)
(-1 +6 +2 +3 +4 +5) step 1
(+1 +6 +2 +3 +4 +5) step 2
(+1 -2 -6 +3 +4 +5) step 3
(+1 +2 -6 +3 +4 +5) step 4
...
(+1 +2 +3 +4 + -6 +5) step 9
(+1 +2 +3 +4 + -5 +6) step 10
(+1 +2 +3 +4 + +5 +6)
```

Slika 6.18: Pohlepno sortiranje

```
( -6 +1 +2 +3 +4 +5)
( -5 -4 -3 -2 -1 +6)
( +1 +2 +3 +4 +5 +6)
```

Slika 6.19: Kraći način

6.3 Teorema u prekidnoj tački

Uočimo da su uzastopni elementi (npr. $(+12 +13)$) poželjni, jer se javljaju u istom redosledu kao i u identičnoj permutaciji. Takodje, i $(-11 -10)$ su poželjni, jer se mogu inverzijom postaviti u pravi redosled. Ova dva para elemenata imaju zajedničku osobinu da je drugi element za 1 veci od prvog. Stoga, definišemo pojam suseda.

Definicija 6.4. (p_i, p_{i+1}) u permutaciji $P = (p_1, p_2, \dots, p_n)$ predstavljaju **susede**, ako je $p_{i+1} - p_i = 1$, a u suprotnom čine **prekid** (primere vidimo na slici 6.20).

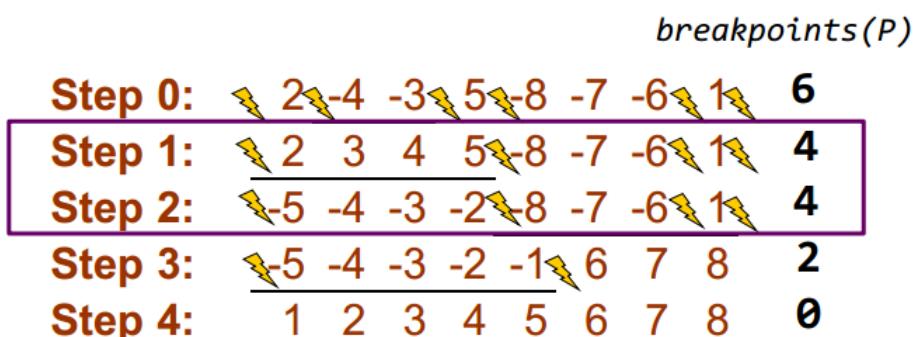
Važi:

$$\text{adjacencies}(P) + \text{breakpoints}(P) = |P| + 1.$$



Slika 6.20: Susedi i prekidi

Sortiranje po promenama eliminacijom prekidnih tacaka



Slika 6.21

Koliko prekidnih tačaka može biti eliminisano u jednoj promeni?

Teorema u prekidnoj tački: Rastojanje između permutacija nije manje od polovine broja prekidnih tačaka.

$$\text{rastojanje između permutacija} \geq \text{breakpoints}(P)/2$$

- Nema garancije da će svaka promena eliminisati 2 prekidne tačke (step 2)
- Najveći broj promena bi bio za permutaciju $(+n + (n - 1) \dots + 1)$ i iznosi $n+1$ (gornja granica)
- Donja granica: $(n + 1)/2$

Velika razlika između donje i gornje granice nam sugerira da moramo preći na drugi način za rešavanje ovog problema.

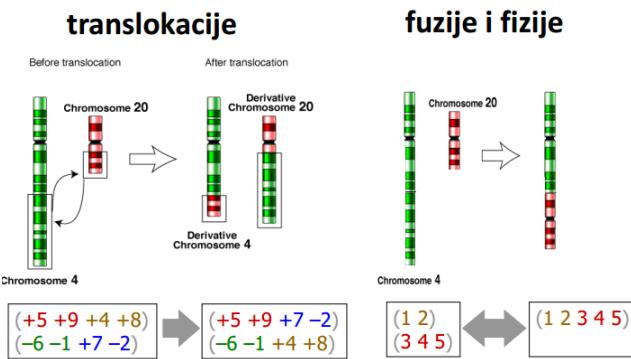
6.4 Preuređivanje u multihromozomnim genomima

Umesto što posmatramo preuređivanje gena u okviru jednog hromozoma (hromozom X kod čoveka i miša), generalizujemo problem i posmatramo sve hromozome genoma.

U ovoj generalizaciji će biti više oblika preuređivanja blokova u genomu (do sada su bila samo obrtanja).

Problem je naizgled komplikovaniji, u nastavku će se ispostaviti da nije tako.

6.4.1 Translokacije, fuzije i fizije



Slika 6.22: Translokacije, fuzije i fizije

Za modeliranje translokacija posmatramo multihromozomni genom sa k hromozoma kao permutaciju koja je podeljena na k delova.

Na primer, genom $(+1 +2 +3 +4 +5 +6) (+7 +8 +9 +10 +11)$ je sastavljen od dva hromozoma $(+1 +2 +3 +4 +5 +6)$ i $(+7 +8 +9 +10 +11)$.

Translokacija razmenjuje segmente različitih hromozoma, npr. translokacija dva hromozoma $(+1 +2 +3 +4 +5 +6) (+7 +8 +9 +10 +11)$ može dovesti do sledeća 2 hromozoma $(+1 +2 +3 +4 +9 +10 +11) (+7 +8 +5 +6)$. Možemo zamišljati translokaciju kao prvo cepanje svakog od hromozoma $(+1 +2 +3 +4 +5 +6) (+7 +8 +9 +10 +11)$ na 2 dela, $(+1 +2 +3 +4) (+5 +6) (+7 +8) (+9 +10 +11)$, a zatim lepljenje rezultujućih segmentata u 2 nova hromozoma, $(+1 +2 +3 +4 +9 +10 +11) (+7 +8 +5 +6)$.

Preuređenja u multihromozomnim genomima nisu ograničena na promene i translokacije. Ona takođe uključuju hromozomske fuzije, koje spajaju 2 hromozoma u 1, kao i fisije, koje dele 1 hromozom na 2 hromozoma (slika 6.22).

Na primer, 2 hromozoma ($+ 1 + 2 + 3 + 4 + 5 + 6$) ($+ 7 + 8 + 9 + 10 + 11$) mogu biti fuzionisani (spojeni) u 1 hromozom ($+ 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11$). Sledeća fisija ovog hromozoma može dovesti do 2 hromozoma ($+ 1 + 2 + 3 + 4$) ($+5+6+7+8+9+10+11$).

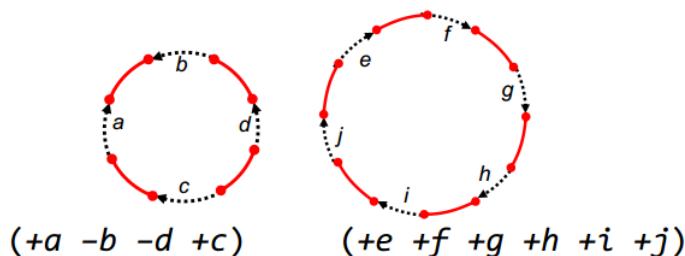
Pre pet miliona godina, ubrzo nakon razdvajanja čoveka i šimpanze, fuzija dva hromozoma (nazvana 2A i 2B) u jednom od naših predaka stvorila je ljudski hromozom 2 i smanjila broj hromozoma sa 24 na 23.

6.5 Problem rastojanja 2-prekida

6.5.1 Od linearnih do cirkularnih hromozoma

Sada se fokusiramo na jedan od hromozoma u multihromozomalmnom genomu i razmotrimo transformacije promene kružnog hromozoma $P = (+ a -b -c + d)$ u $Q = (+ a -b -d + c)$.

Uvodimo pojam crnih usmerenih i crvenih neusmerenih grana. Posmatraćemo hromozome P i Q sa slike 6.23.

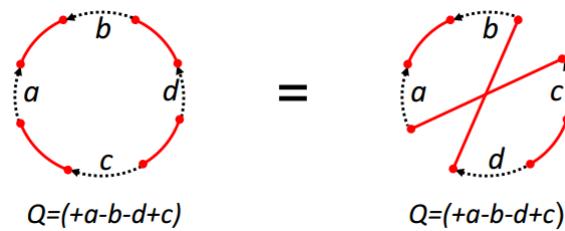


Slika 6.23: Hromozomi P i Q

Crne usmerene grane predstavljaju blokove sintenije.

Crvene neusmerene grane povezuju susedne blokove sintenije.

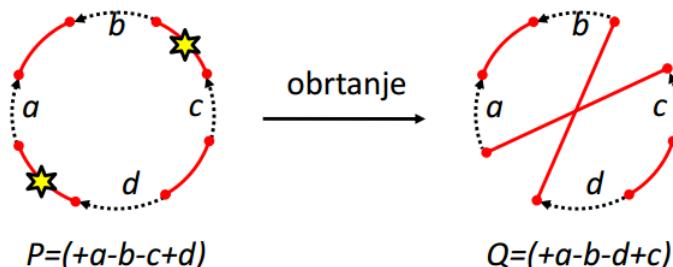
Možemo nacrtati Q na različite načine, zavisno od toga kako se odlučimo da uredimo crne grane. Slika 6.24 prikazuje dve takve ekvivalentne reprezentacije.



Slika 6.24: Ekvivalentne reprezentacije hromozoma Q

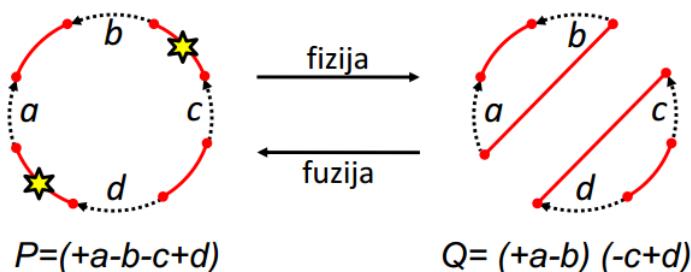
Iako je prvi crtež Q na slici njegova najprirodnija reprezentacija, koristićemo drugu reprezentaciju, jer su joj crne grane raspoređene kružno u potpuno istom redosledu kako se pojavljuju u prirodnoj reprezentaciji $P = (+a -b -c +d)$.

Kao što je prikazano na slici 6.25, fiksiranje crnih grana omogućava nam da vizualizujemo efekat promena. Kao što možemo videti, promena briše dve crvene grane iz P (povezivanje b sa c i d sa a) i zamenjuje ih sa dve nove crvene grane (povezivanje b sa d i c sa a). Ova promena se naziva **obrtanje**.



Slika 6.25: Obrtanje

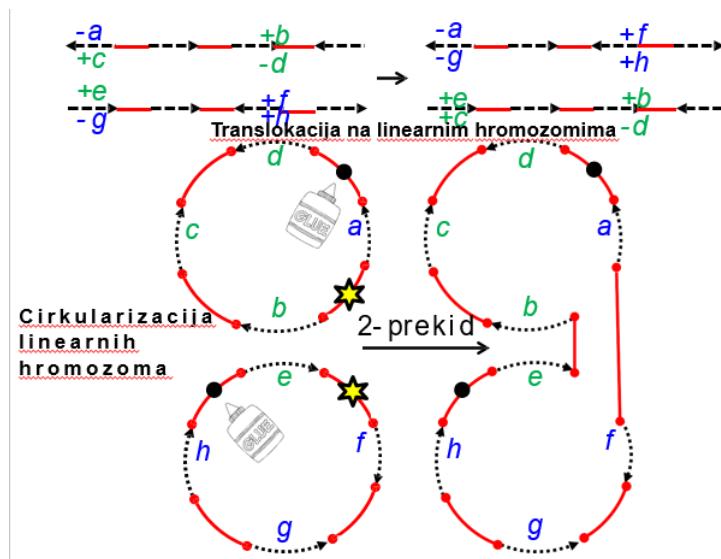
Slika 6.26 ilustruje fiziju $P = (+a -b -c +d)$ u $Q = (+a -b) (-c +d)$. Inverzna operacija fiziji odgovara fuziji dva hromozoma iz Q u hromozom P. Operacije fizije i fizije, kao i promene, odgovaraju brisanju dve grane u jednom genomu i njihovim zamenjivanjem sa 2 nove grane u drugom genomu.



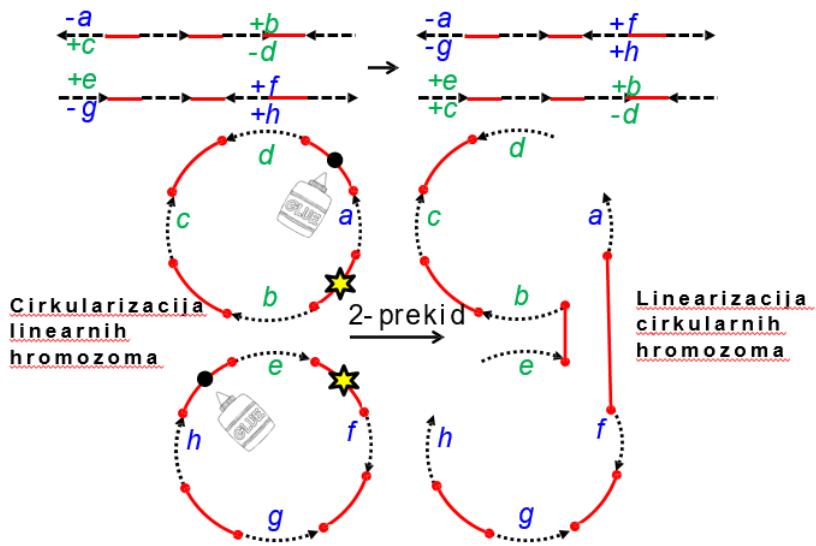
Slika 6.26: Fizija i fuzija

Translokacija koja uključuje dva linearne hromozoma takođe se može simulirati cirkularizacijom ovih hromozoma, a zatim zamenjivanjem dve crvene grane sa dve različite crvene grane, kao što je prikazano na slici 6.27. Zbog toga se mogu objediniti ova 4 različita tipa preuređenja (slika 6.28). Svi oni se mogu posmatrati kao cepanje 2 crvene grane grafa genoma i zamena sa dve nove crvene grane na ista 4 cvora.

Iz tog razloga definišemo opštu operaciju na grafu genoma koja zamenjuje crvenu granu sa dve nove crvene grane pri čemu čvorovi ostaju isti i nazivamo je **2-prekid**.



Slika 6.27: 2-prekid



Slika 6.28: Objedinjavanje sva 4 preuređenja

6.5.2 Rastojanje 2-prekida

Definicija 6.5. *Rastojanje 2-prekida $d(P, Q)$: Minimalni broj 2-prekida koji transformišu genom P u genom Q .*

Definicija 6.6. *Problem rastojanja 2-prekida: Naći rastojanje 2-prekida između dva genoma.*

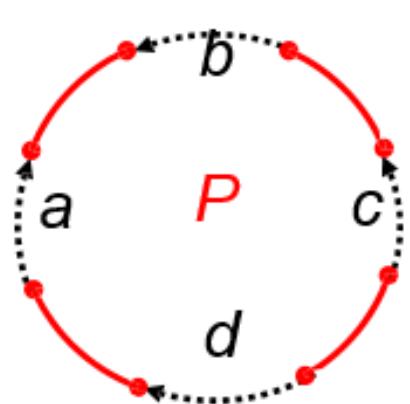
Ulaz. Dva genoma nad istim skupom blokova sintenije.

Izlaz. Rastojanje 2-prekida između ovih genoma.

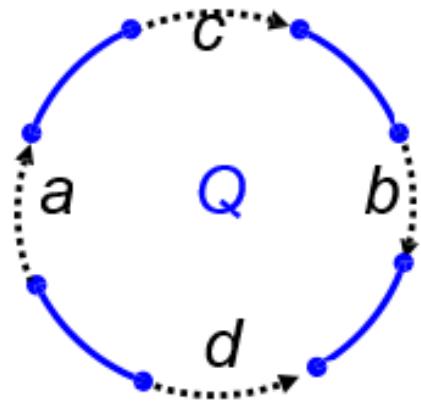
6.6 Grafovi prekidnih tačaka

Za izračunavanje rastojanja 2-prekida konstruisaćemo graf za upoređivanje dva genoma.

Posmatrajmo genome **P** (slika 6.29) i **Q** (slika 6.30).

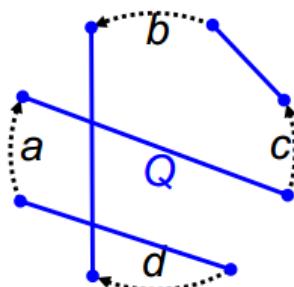


Slika 6.29: Slika 29: Genom P



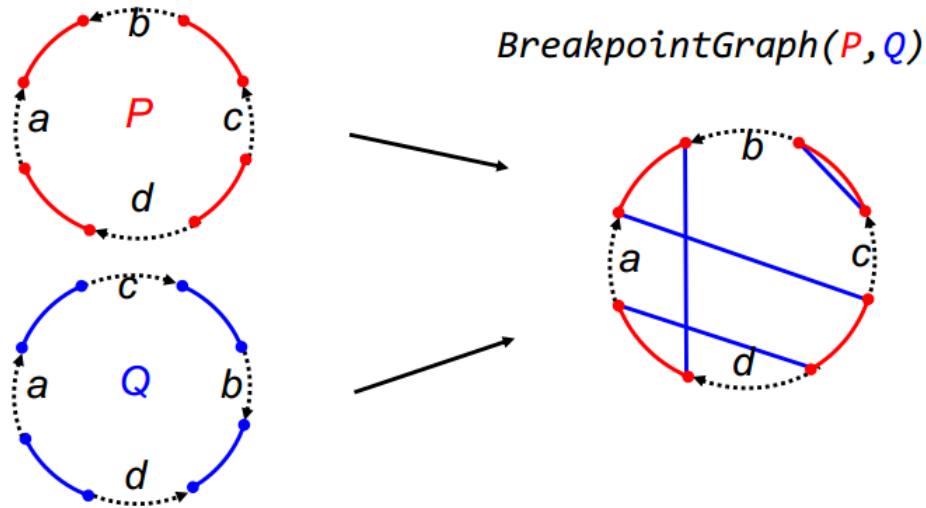
Slika 6.30: Genom Q

Genom **Q** možemo predstaviti i na drugi nacin (slika 6.31).



Slika 6.31: Drugačija reprezentacija genoma Q

Nadgradnjom genoma P i Q dobijamo $\text{BreakpointGraph}(P, Q)$ (slika 6.32).



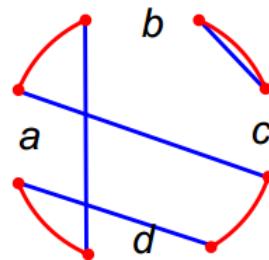
Slika 6.32: Nadgradnja genoma P i Q

Crvene i crne grane u grafu prekidnih tačaka formiraju genom P .

Plave i crne grane u grafu prekidnih tačaka formiraju genom Q .

Crvene i **plave** grane formiraju **alternirajuće crveno-plave cikluse** (slika 6.33).

$\text{BreakpointGraph}(P, Q)$

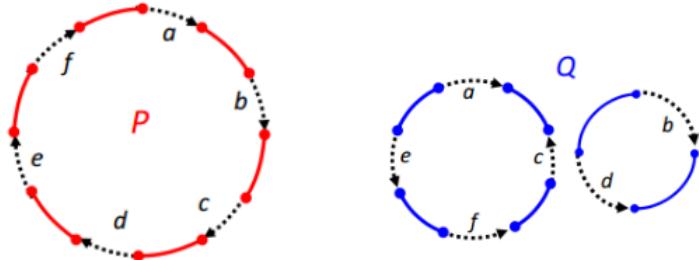


Slika 6.33: Alternirajući crveno-plavi ciklusi

Koristimo oznaku $\text{cycle}(P, Q)$: broj alternirajućih crveno-plavih ciklusa.

Šta predstavlja $\text{cycle}(P, Q)$?

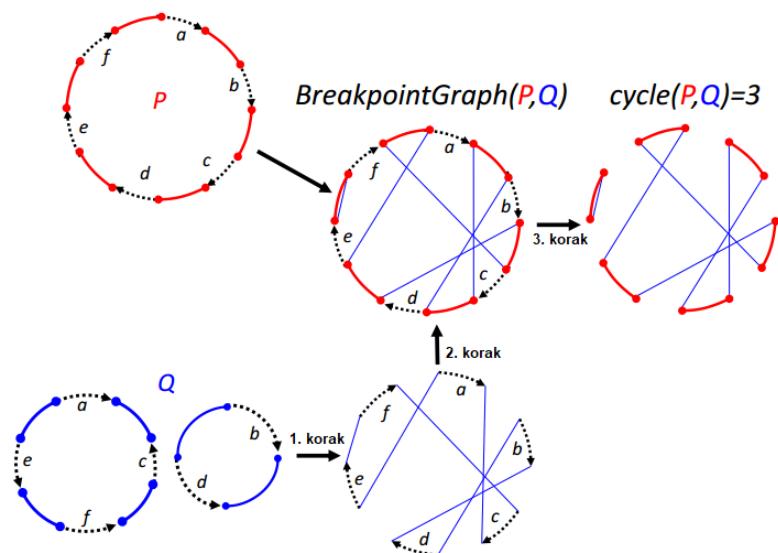
Posmatraćemo grafove genoma P i Q na slici 6.34.



Slika 6.34: Grafovi genoma P i Q

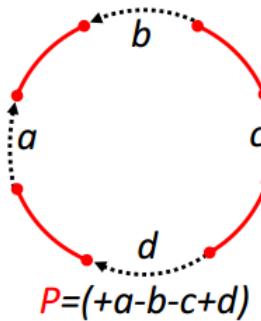
1. korak: Poređamo crne grane genoma Q u isti redosled kao u genomu P
2. korak: Nadgradnja genoma P i Q u jedan
3. korak: Uklanjanje crnih grana

Čitav postupak je prikazan na slici 6.35.



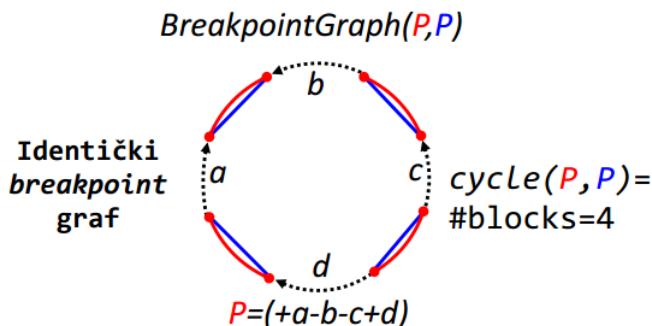
Slika 6.35: $\text{cycle}(P, Q)$

Za dato P (slika 6.36), koje Q maksimizuje $\text{cycle}(P, Q)$?



Slika 6.36: Genom P

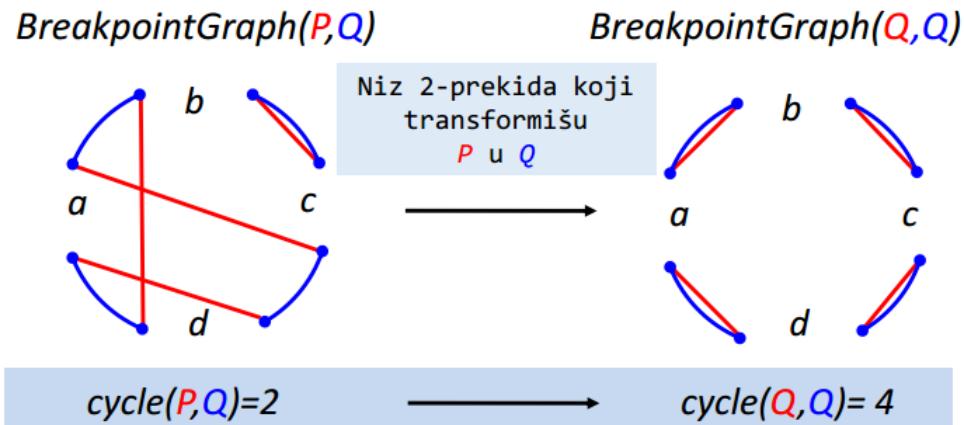
U slučaju da P i Q imaju isti broj blokova sintenije, označimo taj broj sa **Blocks(P, Q)**. Ako su P i Q identični, njihov graf prekidnih tačaka se sastoji od $\text{Blocks}(P, Q)$ ciklusa dužine 2 od kojih svaki sadrži jednu crvenu i jednu plavu granu. Cikluse dužine 2 nazivamo **identičkim ciklusima**, a graf prekidnih tačaka formiran na osnovu identičkih genoma nazivamo **identičkim grafom prekidnih tačaka** (slika 6.37).



Slika 6.37: Identički graf prekidnih tačaka

Preuređenje genoma utiče na crveno-plave cikluse.

Svaka transformacija $P \rightarrow Q$ (slika 6.38) odgovara transformaciji:



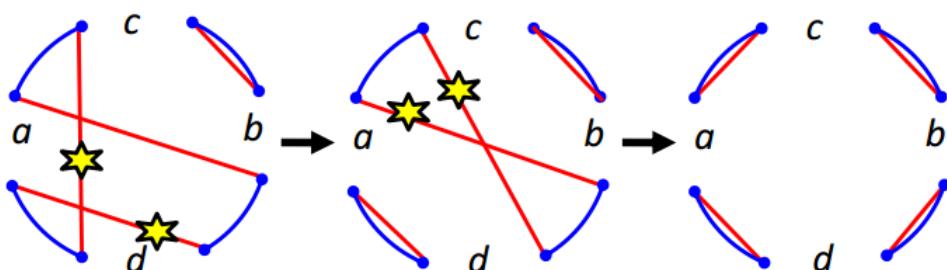
Slika 6.38: Trasnformacija $P \rightarrow Q$

Preuređenja genoma takođe utiču i na $cycle(P, Q)$ (slika 6.39):

$$P = (+a -b -c +d) \rightarrow P' = (+a -b -c -d) \rightarrow P'' = Q = (+a +c +b -d)$$

$BreakpointGraph(P, Q) \rightarrow BreakpointGraph(P', Q) \rightarrow BreakpointGraph(Q, Q)$

$$cycle(P, Q)=2 \rightarrow cycle(P', Q)=3 \rightarrow cycle(Q, Q)=4$$



Slika 6.39: Uticaj preuređenja na $cycle(P, Q)$

6.7 Teorema o rastojanju 2-prekida

Posmatramo problem sortiranja po 2-prekidima (slika 6.40):

$$\begin{array}{c} \text{2-prekidi} \\ P \rightarrow \dots \rightarrow Q \end{array}$$

$$\text{BreakpointGraph}(P, Q) \rightarrow \dots \rightarrow \text{BreakpointGraph}(Q, Q)$$

$$\text{cycle}(P, Q) \rightarrow \dots \rightarrow \text{cycle}(Q, Q) = \text{blocks}(Q, Q)$$

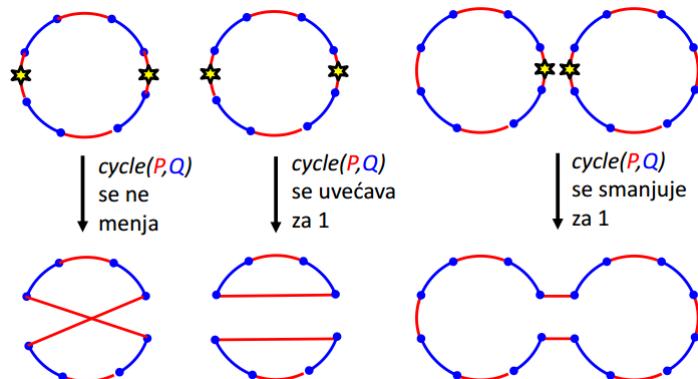
blocks(P, Q) – broj blokova koji učestvuje u izgradnji P i Q

broj crveno-plavih ciklusa se uvećava za
blocks(P, Q) - cycle(P, Q)

Slika 6.40: Sortiranje po 2-prekidima

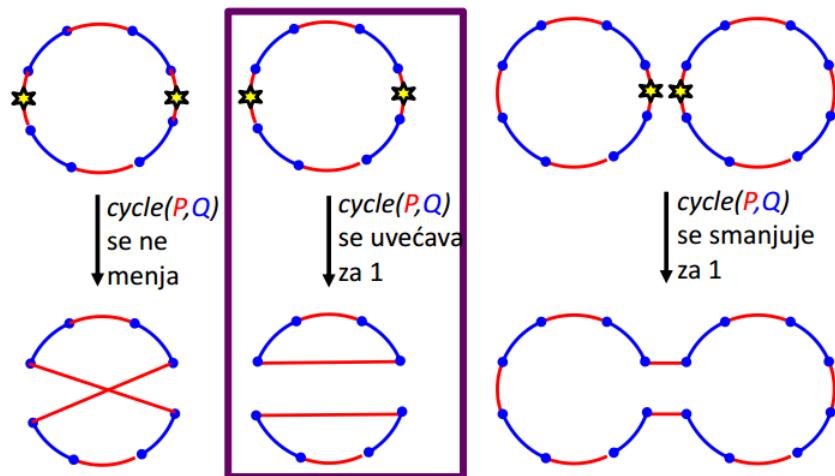
Koliko može svaki 2-prekid da doprinese ovom uvećanju?

2-prekid može izmeniti $\text{cycle}(P, Q)$ za 1. Posmatrajmo sliku 6.41.



Slika 6.41

Postoji 2-prekid povećanje veličine $\text{cycle}(P, Q)$ za 1 (slika 6.42).



Slika 6.42

Teorema o rastojanju 2-prekida

- Svaki 2-prekid povećava broj ciklusa najviše za 1
- Za svaki 2-prekid postoji povećanje broja ciklusa za tačno 1
- Svako sortiranje po 2-prekidima mora povećati broj ciklusa za $\text{blocks}(P, Q) - \text{cycle}(P, Q)$
- 2-prekid rastojanje između genoma P i Q:

$$d(P, Q) = \text{blocks}(P, Q) - \text{cycle}(P, Q)$$

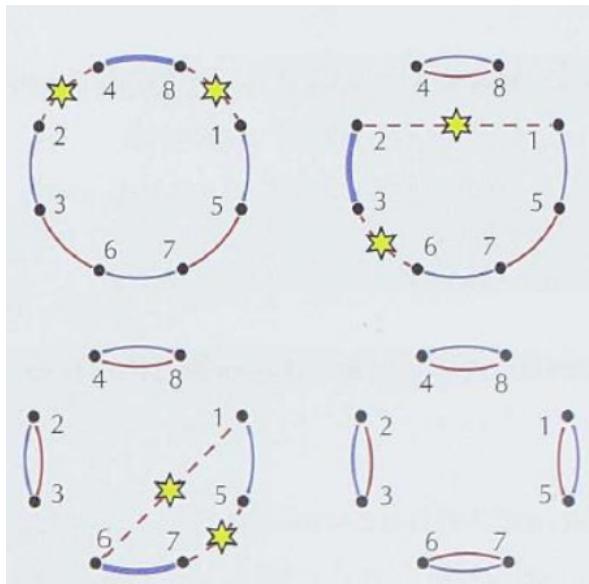
Rastojanje 2-prekida između genoma čoveka i miša

- Genomi čoveka i miša se mogu rastaviti na 280 blokova sintenije (dužine bar pola miliona nukleotida)
- Graf prekidnih tačaka nad ovim blokovima ima ukupno 35 ciklusa
- Na osnovu teoreme o rastojanju 2-prekida:

$$d(H, M) = \text{blocks}(H, M) - \text{cycle}(H, M) = 280 - 35 = 245$$

- Postoje različite verzije scenarija sa 245 koraka.
- Pravi evolutivni scenario je možda imao i više od 245 koraka.

Shortest rearrangement scenario



Slika 6.43: Shortest rearrangement scenario

ShortestRearrangementScenario(P, Q)

output P

$RedEdges \leftarrow ColoredEdges(P)$

$BlueEdges \leftarrow ColoredEdges(Q)$

$BreakpointGraph \leftarrow$ the graph formed by $RedEdges$ and $BlueEdges$

while $BreakpointGraph$ has a non-trivial cycle $Cycle$

$(j, i') \leftarrow$ an arbitrary edge from $BlueEdges$ in a nontrivial red blue cycle

$(i, j) \leftarrow$ an edge from $RedEdges$ originating at node j

$(i', j') \leftarrow$ an edge from $RedEdges$ originating at node i'

$RedEdges \leftarrow RedEdges$ with edges (i, j) and (i', j') removed

$RedEdges \leftarrow RedEdges$ with edges (j, i') and (j', i) added

$BreakpointGraph \leftarrow$ the graph formed by $RedEdges$ and $BlueEdges$

$P \leftarrow 2\text{-BreakOnGenome}(P, i, i', j, j')$

output P

$2\text{-BreakOnGenome}(P, i, i', j, j')$ - uklanja grane (i, i') i dodaje grane (i, j) i (i', j') (genom predstavljen grafom prekidnih tačaka) (pogledati sliku 6.43).

6.8 Zadaci sa vezbi

6.8.1 ChromosomeToCycle

```

1 def chromosome_to_cycle(chromosome):
2
3     nodes = [0 for i in range(2*len(chromosome))]
4
5     for j in range(len(chromosome)):
6         i = chromosome[j]
7         if i > 0:
8             #dodajemo cuvorove
9             nodes[2*j] = 2*i - 1
10            nodes[2*j + 1] = 2*i
11        else:
12            nodes[2*j] = -2*i
13            nodes[2*j + 1] = -2*i - 1
14
15    return nodes
16
17 def main():
18     print(chromosome_to_cycle([1, -2, -3, 4]))
19
20 if __name__ == "__main__":
21     main()

```

6.8.2 CycleToChromosome

```

1 def cycle_to_chromosome(nodes):
2
3     chromosomes = [0 for i in range(len(nodes)//2)]
4
5     for j in range(len(nodes)//2):
6         if nodes[2*j] < nodes[2*j + 1]:
7             chromosomes[j] = nodes[2*j +1] // 2
8         else:
9             chromosomes[j] = -nodes[2*j] // 2
10
11    return chromosomes
12
13 def main():
14     nodes = [1, 2, 4, 3, 6, 5, 7, 8]
15     print(cycle_to_chromosome(nodes))
16
17 if __name__ == "__main__":
18     main()

```

6.8.3 GreedySorting

```

1 def find(P, start, n):
2     for i in range(start, len(P)):
3         if P[i] == n or P[i] == -n:
4             return i
5
6 def reversal(P, start, stop):
7     rev = [-i for i in P[start:stop+1]]
8     rev.reverse()
9     P[start:stop+1] = rev
10
11    return P
12
13 def greedy_sorting(P):
14     approx_reversal_distance = 0
15
16     print(P)
17
18     for k in range(len(P)):
19         if P[k] != k+1:
20             i = find(P, k, k+1)
21             P = reversal(P, k, i)
22             approx_reversal_distance += 1
23
24             print(P)
25
26             if P[k] < 0:
27                 P[k] = -P[k]
28                 approx_reversal_distance += 1
29
30             print(P)
31
32     return approx_reversal_distance
33
34 def main():
35     # P = [+1, -7, +6, -10, +9, -8, +2, -11, -3, +5, +4]
36     # P = [+6, -7, +1, -10, +9, -8, +2, -11, -3, +5, +4]
37     P = [-2, -5, +3, +4, +1]
38
39     print(greedy_sorting(P))
40
41 if __name__ == "__main__":
42     main()

```

6.8.4 ShortestRearrangementScenario

```

1 import copy
2
3 def cycle_to_chromosome(nodes):
4
5     chromosomes = [0 for i in range(len(nodes)//2)]
6
7     for j in range(len(nodes)//2):
8         if nodes[2*j] < nodes[2*j+1]:
9             chromosomes[j] = nodes[2*j+1] // 2
10        else:
11            chromosomes[j] = -nodes[2*j] // 2
12
13    return chromosomes
14
15 def colored_edges(P):
16     edges = []
17     for chromosome in P:
18         nodes = chromosome_to_cycle(chromosome)
19         for j in range(len(chromosome)):
20             # Obojene grane su neusmerene, pa dodajemo oba smera
21             edges.append((nodes[2*j+1], nodes[(2*j+2)
22 len(nodes)]))edges.append((nodes[(2*j+2)
23 len(nodes)],nodes[2*j+1]))
24
25 def chromosome_to_cycle(chromosome):
26
27     nodes = [0 for i in range(2*len(chromosome))]
28
29     for j in range(len(chromosome)):
30         i = chromosome[j]
31         if i > 0:
32             nodes[2*j] = 2*i - 1
33             nodes[2*j + 1] = 2*i
34         else:
35             nodes[2*j] = -2*i
36             nodes[2*j + 1] = -2*i - 1
37     return nodes
38
39
40 def graph_to_genome(genome_graph):
41     P = []

```

```
42     nodes = []
43
44     for (i,j) in genome_graph:
45         nodes.append(i)
46         nodes.append(j)
47
48
49     prvi = [nodes[-1]]
50     ostatak = copy.copy(nodes[:-1])
51     nodes = prvi + ostatak
52
53     chromosome = cycle_to_chromosome(nodes)
54     P.append(chromosome)
55
56     return P
57
58 def two_break_on_genome_graph(genome_graph, i, ip, j, jp):
59
60     new_edges = []
61
62     for edge in genome_graph:
63         if (edge[0] == i and edge[1] == ip) or (edge[0] == j and
64             ↪ edge[1] == jp):
65             continue
66         new_edges.append(edge)
67
68         new_edges.append((i,j))
69         new_edges.append((ip,jp))
70
71     return new_edges
72
73 def black_edges(P):
74     nodes = chromosome_to_cycle(P)
75     edges = []
76     i = 0;
77
78     while i < len(nodes):
79         if nodes[i] < nodes[i+1]:
80             edges.append((nodes[i], nodes[i+1]))
81         else:
82             edges.append((nodes[i+1], nodes[i]))
83
84         i = i + 2
85
86     return edges
```

```

86
87 def two_break_on_genome(P, i, ip, j, jp):
88     genome_graph = black_edges(P) + colored_edges([P])
89     genome_graph = two_break_on_genome_graph(genome_graph, i, ip, j
90     ↪ , jp)
91     P = graph_to_genome(genome_graph)
92
93     return P
94
95 def has_nontrivial_cycle(P, Q):
96     for (v,w) in P:
97         if (v,w) not in Q and (w,v) not in Q:
98             return True
99
100    return False
101
102 def select_edge_from_nontrivial_cycle(P, Q):
103     for (v,w) in Q:
104         if (v,w) not in P and (w,v) not in P:
105             return (v,w)
106
107
108 def shortest_rearrangement_scenario(P, Q):
109     red_edges = colored_edges([P])
110     blue_edges = colored_edges([Q])
111
112     num_of_breaks = 0
113
114     while has_nontrivial_cycle(red_edges, blue_edges):
115         (j,i_p) = select_edge_from_nontrivial_cycle(red_edges,
116         ↪ blue_edges)
117
118         i = -1
119         j_p = -1
120
121         for (v,w) in red_edges:
122             if v == j:
123                 i = w
124             if w == j:
125                 i = v
126             if v == i_p:
127                 j_p = w
128
129         red_edges.remove((j, i))

```

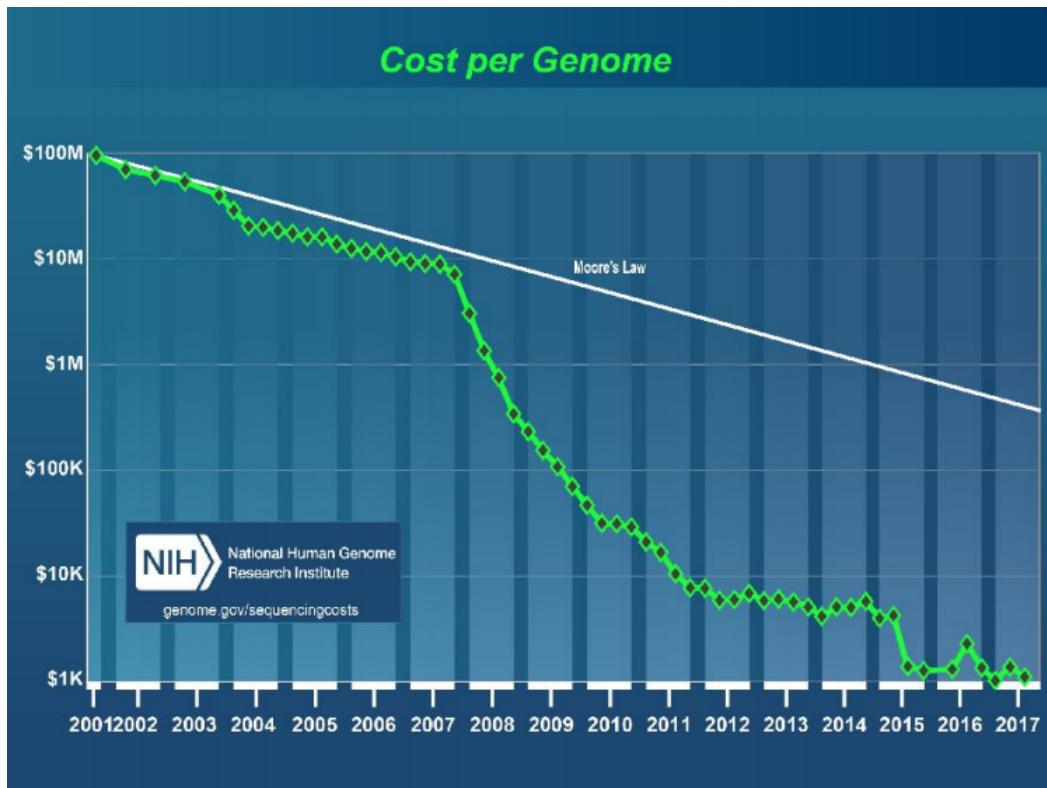
```
129     red_edges.remove((i, j))
130
131     red_edges.remove((i_p, j_p))
132     red_edges.remove((j_p, i_p))
133
134     red_edges.append((j, i_p))
135     red_edges.append((i_p, j))
136
137     red_edges.append((j_p, i))
138     red_edges.append((i, j_p))
139
140     num_of_breaks += 1
141
142     return num_of_breaks
143
144 def main():
145     P = [1,-2,-3,4]
146     Q = [1, 2, 3, -4]
147
148     print(shortest_rearrangement_scenario(P,Q))
149
150 if __name__ == "__main__":
151     main()
```

Glava 7

Kako locirati mutacije koje izazivaju bolesti?

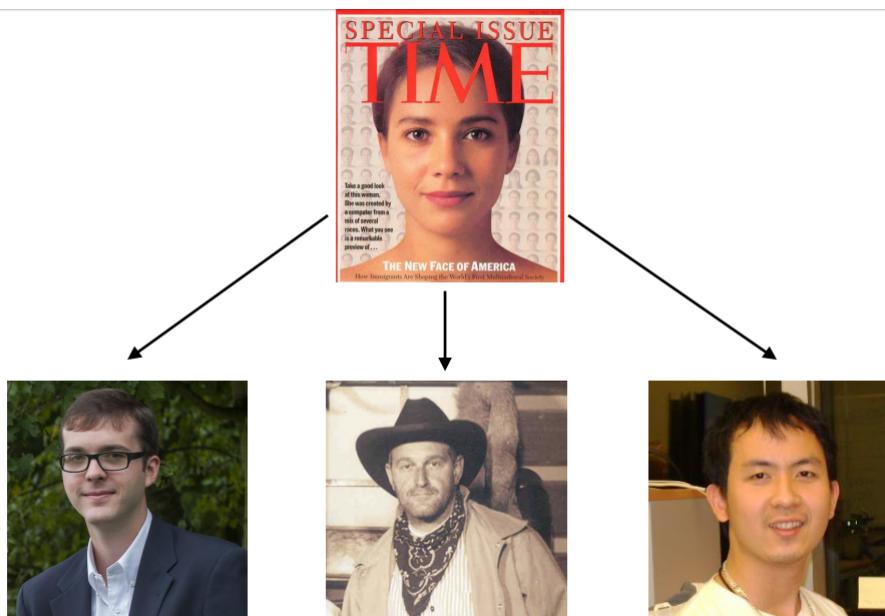
7.1 Mapiranje očitavanja

Cena sekvencionisanja genoma je od 2001. u konstantnom padu, i teži se tome da ono postane potpuno pristupačno običnom čoveku, kao i da bude sastavni deo lekarske usluge.



Slika 7.1: Kretanje cene sekvencioniranja u poslednjih 17 godina

Oko 1% dece rodi se sa mentalnom retardacijom, ali uzroci ove pojave i danas nisu razjašnjeni, jer do nje može dovesti niz različitih genetskih poremećaja. Jedan od njih je i Ohdo sindrom, koji izaziva bezličan, "maskoliki" izraz lica. Biologzi su 2011. godine uspeli da pronađu niz zajedničkih mutacija kod pacijenata, koje su kasnije iskorišćene za identifikovanje jedinstvene mutacije proteina, odgovorne za nastanak ovog sindroma. Razumevanje suštinskog uzroka Ohdo sindroma samo je jedno od otkrića do kojeg se došlo proučavanjem genetskih poremećaja upotrebom **mapiranja očitavanja**. Kod ove metode, porede se sekvencionisana očitavanja DNK uzeta od pojedinaca sa **referentnim ljudskim genomom**. **Referentni ljudski genom** zamišljen je da bude "prosečan" ljudski genom izračunat na određenom broju uzoraka. Trenutni referentni genom baziran je na genomima 13 dobrovoljaca iz SAD-a; i dalje se vrši njegovo usavršavanje ispravljanjem grešaka i popunjavanjem rupa (trenutno ih je preko stotinu). U proseku, razlika između individualnog i referentnog genoma je u oko 3 miliona mutacija.



Slika 7.2: Prikaz granja genoma vrste na više personalnih genoma

Pitanje je kako možemo efikasno sastaviti individualne genome koristeći referentne. Možemo koristiti **asempliranje**, ali konstrukcija de Brojnovog grafa zahteva mnogo memorije. Možemo koristiti postojeću strukturu referentnog genoma kao pomoć u sekvencioniranju genoma pacijenta.

```

CTGATGATGGACTACGCTA
CTACTGCTAGCTGTATTAC
GATCAGCTACCACATCGTA
GCTACGATGCATTAGCAAG
CTATCGATGATCGATCGA
TTATCTACGATCGATCGAT
CGATCACTATACGAGCTAC
TACGTACGTACGATGCCG
GACTATTATCGACTACAGA
TAAAACATGCTAGTACAAC
AGTATACATAGCTGCGGGGA
TACGATTAGCTAATAGCTG
ACGATATCCGAT

```

```

CTGATGATGGACTACGCTA
CTACTGCTAGCTGTATTAC
GATCAGCTACAACATCGTA
GCTACGATGCATTAGCAAG
CTATCGATGATCGATCGA
TTATCTACGATCGATCGAT
CGATCACTATACGAGCTAC
TACGTACGTACGATGCCT
GACTATTATCGACTACAGA
TGAAACATGCTAGTACAAC
AGTATACATAGCTGCGGGGA
TACGATTAGCTAATAGCTG
ACGATATCCGAT

```

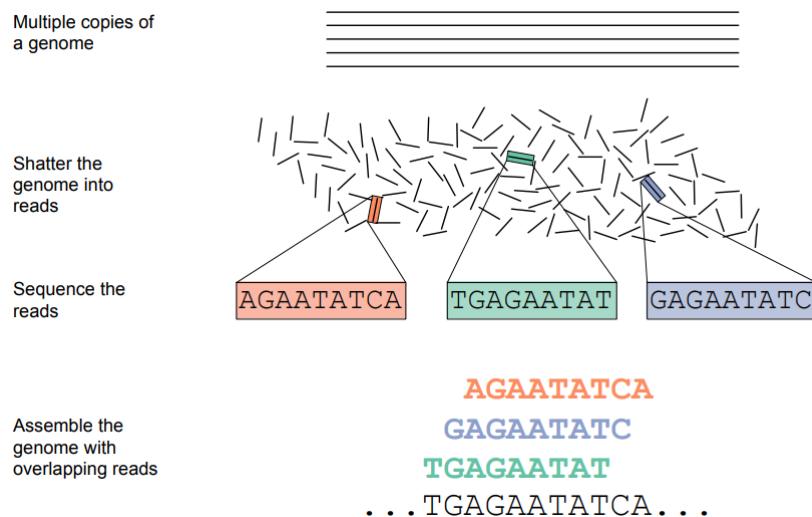
```

CTGATGATGGACTACGCTA
CTACTGCTAGCTGTATTAC
GATCAGCTACTACATCGTA
GCTACGATGCATTAGCAAG
CTATCGATGATCGATCGA
TTATCTACGATCGATCGAT
CGATCACTATACGAGCTAC
TACGTACGTACGATGCCA
GACTATTATCGACTACAGA
TCAAACATGCTAGTACAAC
AGTATACATAGCTGCGGGGA
TACGATTAGCTAATAGCTG
ACGATATCCGAT

```



Slika 7.3: Razlike u personalnim genomima



Slika 7.4: Primer asembliranja

Mapiranje očitavanja predstavlja određivanje pozicije u referentnom genomu sa kojima svako očitavanje ima visoku sličnost.

```

CTGAGGATGGACTACGCTACTACTGATAGCTGTTT
GAGGA      CCACG      TGA-A

```

Slika 7.5: Primer mapiranja očitavanja; gornja niska predstavlja referentni genom, a donje niske očitavanja individualnog genoma

7.1.1 Egzaktno upativanje šablon-a

Potrebno je pronaći gde se očitavanja egzaktno poklapaju sa referentnim genom. Postoji jednostruko i višestruko uparivanje šablon-a.

Problem jednostrukog uparivanja šablon-a:

Ulaz: Niske *Pattern* i *Genome*.

Izlaz: Sve pozicije u niski *Genome* gde se niska *Pattern* pojavljuje kao podniska.

Problem višestrukog uparivanja šablon-a:

Ulaz: Kolekcija niski *Patterns* i *Genome*.

Izlaz: Sve pozicije u niski *Genome* gde se niske iz kolekcije *Patterns* pojavljuju kao podnische.

7.1.2 Moguća rešenja

Rešenje koje nam prvo pada na pamet je rešavanje problema grubom silom. Algoritam se sastoji u tome da se linearno krećemo kroz genom i proveravamo da li se dati šablon poklapa sa podniskom genoma iste dužine, koja počinje na toj poziciji.

```

    p a n a m a b a n a n a s
    n a n a
  
```

Slika 7.6: Uparivanje šablon-a grubom silom - nepoklapanje

```

    p a n a m a b a n a n a s
    n a n a
  
```

Slika 7.7: Uparivanje šablon-a grubom silom - poklapanje

Vreme izvršavanja algoritma u slučaju jednostrukog *Pattern*-a je $O(|Genome| * |Pattern|)$, dok je u slučaju višestrukog *Patterns*-a je $O(|Genome| * |Patterns|)$, gde je $|Patterns|$ — suma dužina elemenata liste *Patterns*.

Međutim, problem je u tome što genomi mogu biti veoma dugi. U slučaju ljudskog genoma (3 GB), ukupna dužina svih očitavanja može biti veća od 1 TB; kao rezultat toga, algoritam složenosti $O(|Genome| * |Patterns|)$ je previše spor.

7.1.3 Sufiksna stabla

Razlog velike neefikasnosti prethodnog algoritma jeste u tome što paterni prolaze kroz genom nezavisno jedan od drugog. Ako nisku *Genome* zamislimo kao put, onda bi izvršavanje algoritma grube sile bilo analogno vožnji svakog paterna po putu u zasebnom automobilu. Ono što želimo jeste da sve paterne smestimo u jedan "autobus", čime bi nam bio dovoljan samo jedan prolazak kroz nisku *Genome*.

Zbog toga paterne organizujemo u strukturu podataka nalik na usmereni aciklički graf, koju nazivamo **Trie** i koja ima sledeće osobine:

- Trie ima jedinstven čvor sa ulaznim stepenom nula, koji nazivamo koren
- Svaka grana Trie je obeležena jednim slovom
- Grane koje izlaze iz jednog čvora obeležene su različitim slovima
- Svaki sufiks neke niske dobija se nadovezivanjem slova duž neke putanje grafa, idući od korena naniže
- Svaka putanja stabla od korena do lista, ili do čvora sa izlaznim stepenom 0, predstavlja jedan element iz liste *Patterns*

Najjednostavniji način konstrukcije strukture Trie jeste iterativno dodavanje niski iz niza *Patterns* u rastuću strukturu.

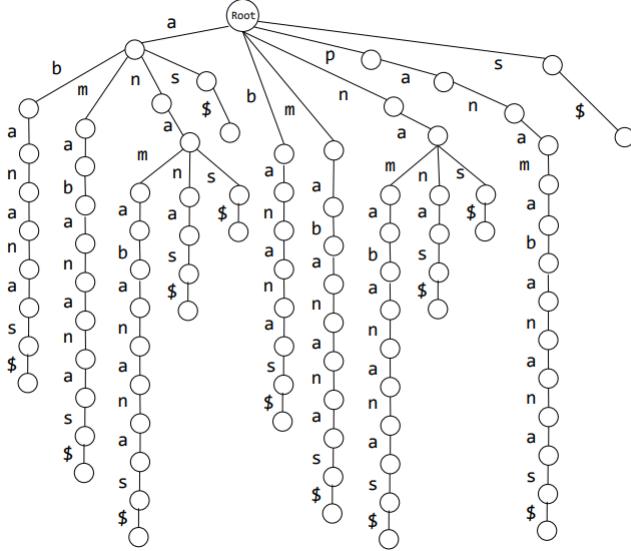
Za date niske Text i Trie(*Patterns*) možemo brzo proveriti da li se neki element niza *Patterns* predstavlja prefiks niske Text. Dovoljno je da krenemo da spelujemo Text i da slovo po slovo prolazimo kroz Trie od korena naniže. Za svako slovo iz Text-a gledamo da li iz trenutnog čvora postoji grana obeležena tim slovom; ukoliko postoji, nastavljamo sa pretragom; u suprotnom obustavljamo pretragu i zaključujemo da nijedan element niza *Patterns* nije prefiks niske Text. Ukoliko stignemo do lista, onda je pretraga bila uspešna.

Da bismo pronašli da li se neki patern nalazi u genomu, potrebno je da u $\|Text\|$ iteracija pokrećemo prethodno opisani algoritam, pri čemu u svakoj iteraciji iz niske Text izbacujemo početni simbol, sve dok ona ne postane prazna.

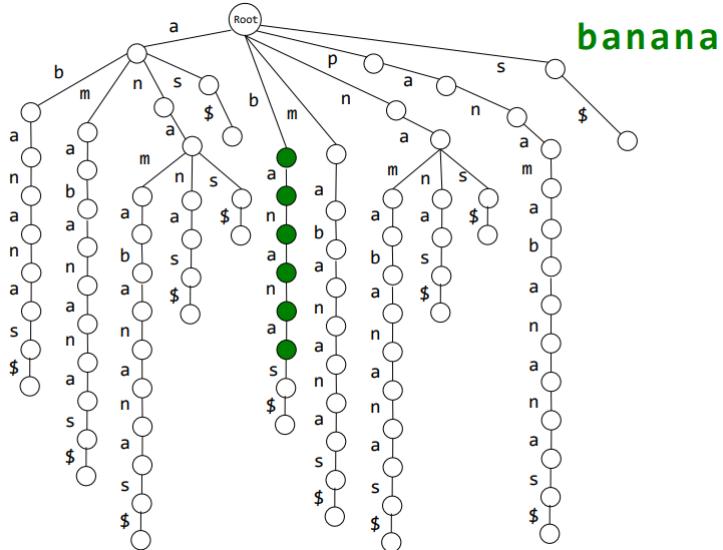
Iako je prethodno opisani postupak vremenski efikasan, njegova mana leži u velikom zauzeću memorije. Naime, veličina strukture Trie proporcionalna je ukupnom broju simbola niza *Patterns*, a posto veličina kolekcije očitavanja ljudskog genoma može dostići 1 TB, memorija potrebna za čuvanje ove strukture je prevelika.

Bolji pristup je da se struktura Trie pravi na osnovu niske Text, tj. genoma. Za ovo će nam biti struktura poznata kao **sufiksni trie**. **Sufiksni trie** date niske predstavlja trie formiran na osnovu svih sufiksa te niske. Pre konstrukcije sufiksnog stabla, na kraj niske dodajemo simbol \$, kako bismo kasnije znali kad smo stigli do kraja.

Proveru da li se dati patern nalazi u tekstu vršimo tako što tražimo put u sufiksnom stablu, speljujući slova paterna do kraja. Ukoliko dođe do nepoklapanja, pretraga je neuspešna. U suprotnom, pretraga je uspešna.

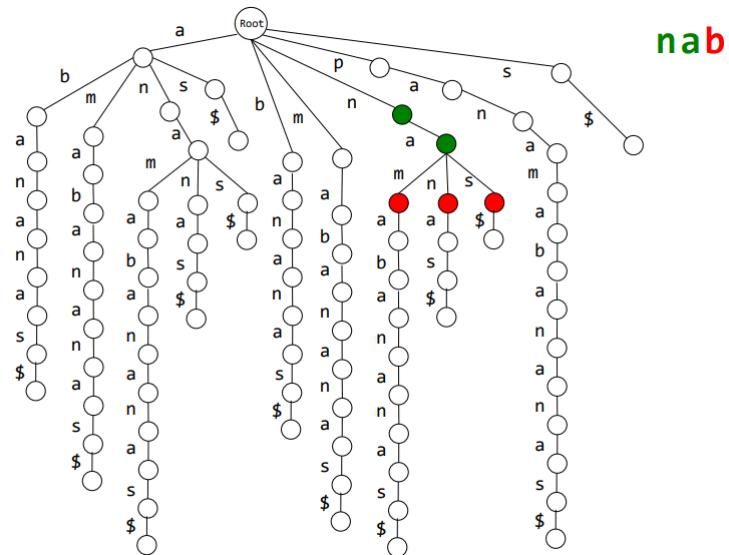


Slika 7.8: Nekompresovano sufiksno stablo

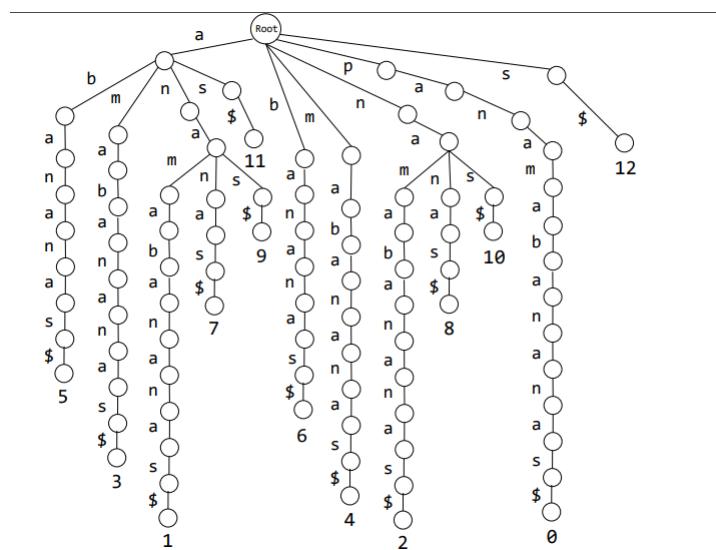


Slika 7.9: Nekompresovano sufiksno stablo - uspešno pronalaženje niske

Ovim postupkom možemo utvrditi da li se pattern pojavljuje u genomu, ali ne i na kojoj poziciji. Za to moramo dodati još informacija u stablu. Na svakom listu dodamo početnu poziciju u niski Genome sufiksa koji se završava u tom listu.

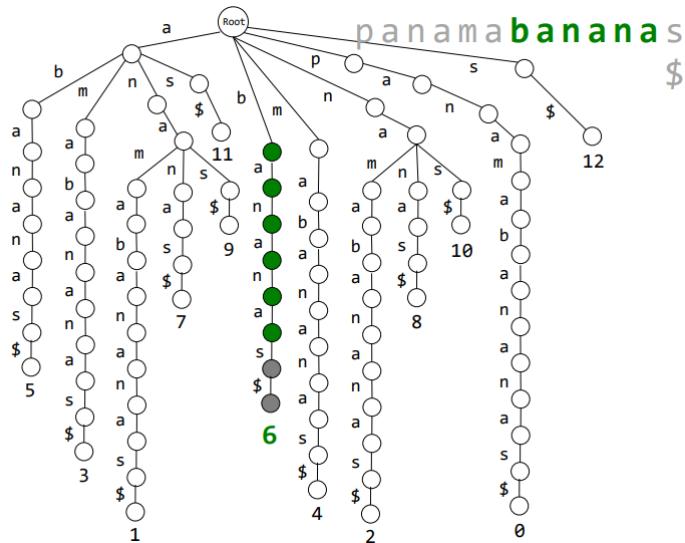


Slika 7.10: Nekompresovano sufiksno stablo - neuspešno pronalaženje niske

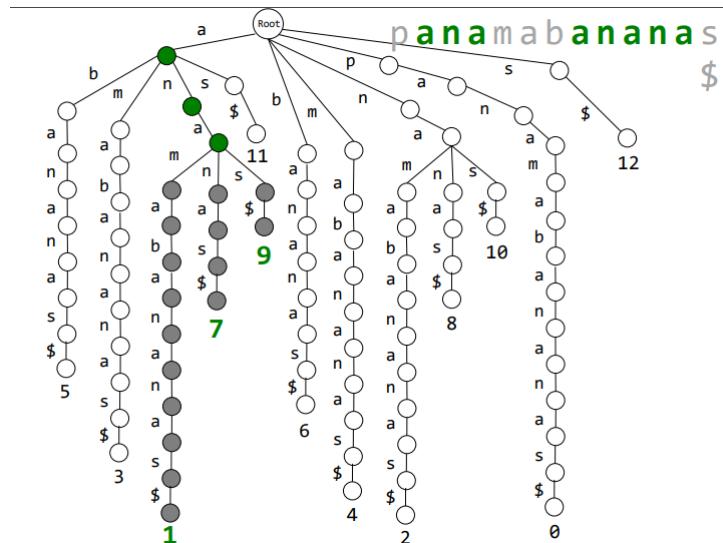


Slika 7.11: Sufiksno stablo sa numerisanim prefiksima

Sad se postavlja pitanje, kada pronađemo uparivanje, kako da znamo na kojoj poziciji se ono nalazi. To je sada lako, kada pronađemo uparivanje, nastavimo sa kretanjem naniže do lista, gde se nalazi pozicija odakle počinje pojavljivanje podniske.



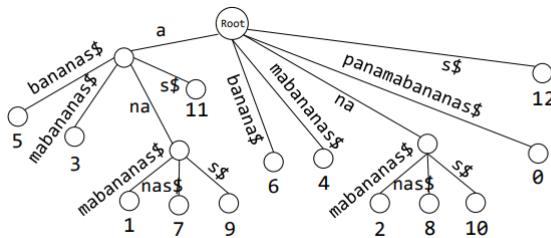
Slika 7.12: Primer nalaženja pozicije nakon uparivanja



Slika 7.13: Primer nalaženja pozicije nakon uparivanja - više poklapanja

Da bismo smanjili prostornu složenost, možemo kompresovati svaku putanju koja se ne grana u jednu granu. Ovakva struktura podataka naziva se **sufiksno stablo**.

Za svaku nisku *Genome* važi da je ukupan broj čvorova manji od dvostrukе dužine niske *Genome*, tj. $\#nodes < 2|Genome|$. Ovo važi na osnovu činjenice da je broj listova jednak dužini genoma, tj. $\#leaves = |Genome|$, odnosno da je broj unutrašnjih čvorova manji od dužine genoma umanjene za jedan, tj. $\#internalnodes < |Genome| - 1$.



Slika 7.14: Kompresovano stablo

Prostorna i vremenska složenost

Vremenska složenost:

- $O(|Genome|^2)$ za konstrukciju sufiksnog stabla tako što se prvo konstruiše nekompresovano sufiksno stablo.
- $O(|Patterns|)$ za nalaženje uparivanja.

Prostorna složenost:

- $O(|Genome|^2)$ za konstrukciju sufiksnog stabla tako što se prvo konstruiše nekompresovano sufiksno stablo.
- $O(|Patterns|)$ za čuvanje sufiksnog stabla.

Postoje algoritmi sa linearnom prostornom i vremenskom složenošću. Vremenska složenost:

- $O(|Genome|)$ za konstrukciju sufiksnog stabla direktno.
- $O(|Patterns|)$ za nalaženje uparivanja.
- Ukupno $O(|Genome| + |Patterns|)$

Prostorna složenost:

- $O(|Genome|^2)$ za konstrukciju sufiksnog stabla direktno.
- $O(|Patterns|)$ za čuvanje sufiksnog stabla.
- Ukupno $O(|Genome|)$

7.2 Kompresija niski i Barouz-Vilerova transformacija

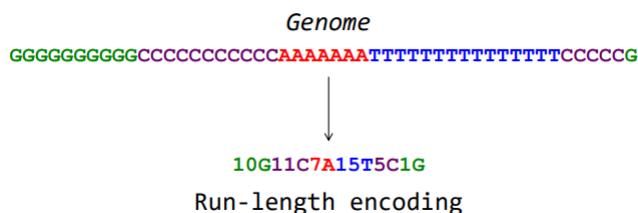
Najveći problem koji se javlja sa prethodnom rešenjem je to što O-notacija ignoriše konstante, a najpoznatija implementacija sufiksnih stabala zahteva 20^* —Genome— (npr. veličina humanog genoma je 3GB = \approx 60 GB; i dalje unapređenje u odnosu na 1TB). Postavlja se pitanje da li možemo smanjiti faktor konstante. Odgovor nam daje kompresija genoma.

7.2.1 Kompresija genoma

Glavna ideja ovog rešenja jeste da se smanji količina memorije potrebna za čuvanje niske Genome. Za ovo su nam potrebne metode za kompresiju niske velikih dužina, što je naizgled sasvim drugačiji problem.

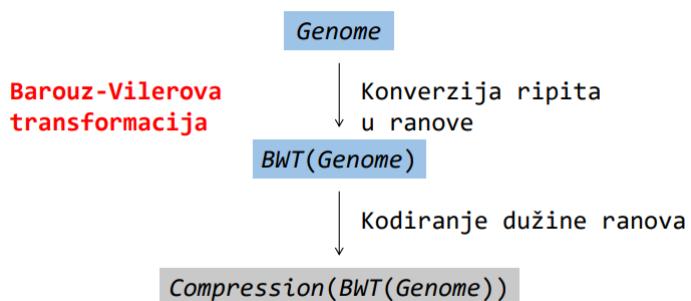
U ovom genomimu imamo nekoliko uzastopnih ponavljanja jedne aminokiseline (ranovi, runs): prvo uzastopna ponavljanja aminokiseline G, pa C i tako dalje), a u nekim imamo uzastopna ponavljanja nizova aminokiselina (ripitsi, repeats): prvo uzastopna ponavljanja GAC, pa CATT i tako dalje.

Prva ideja pri rešavanju ovog problema jeste da kodiramo dužine ranova.



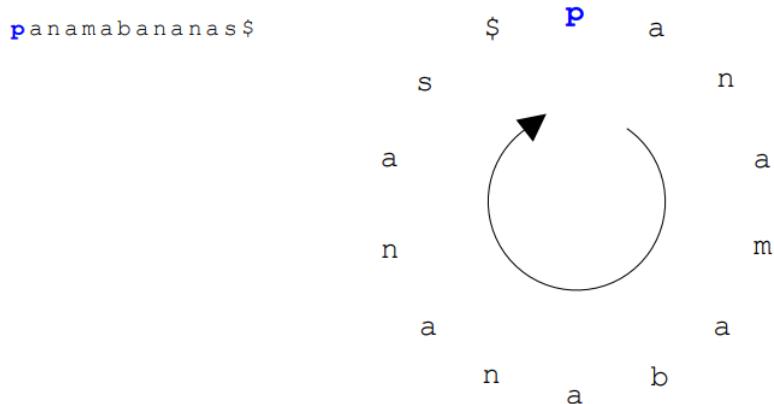
Slika 7.15

Problem kod ovog pristupa jeste to što u genomu nema mnogo ranova. Međutim, ima mnogo ripita. Postavlja se pitanje kako izvesti transformaciju ripita u ranove.



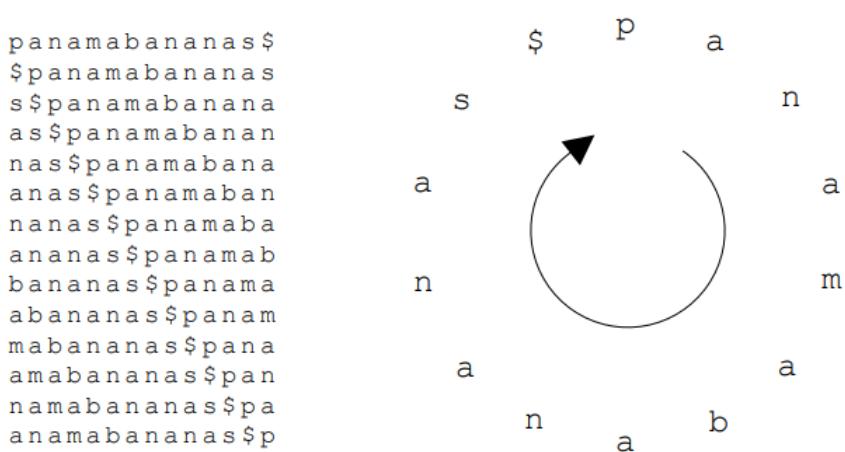
Slika 7.16

Odgovor na ovo pitanje daje nam Varouz-Vilerova transformacija (The Burrows-Wheeler Transform, skraćeno BWT).



Slika 7.17

Ideja kod ovog algoritma je da se na početku formiraju sve ciklične rotacije date niske.



Slika 7.18: Scenario sa 4 promene

Zatim se vrši sortiranje svih dobijenih niski leksikografski (\$ je na početku).

Zatim posmatramo poslednju kolonu. Možemo primetiti da poslednja kolona sadrži veliki broj ranova. Međutim, isti slučaj je i sa prvom kolonom. Prvo ćemo se pozabaviti dekompresijom dobijene niska, pa ćemo se posle vratiti na ovo pitanje.

7.3 Inverzna BWT

Pogledajmo primer BWT-a za nisku *banana*.

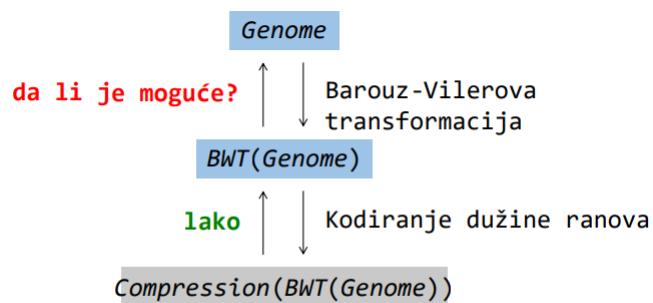
Ako sortiramo karaktere poslednje kolone "annb\$aa", dobićemo prvu kolonu matrice. Na osnovu toga znamo 2-gramske sastav cirkularne niske *banana\$*.

panamabananas\$	\$panamabana
\$panamabananas	s\$panamabana
s\$panamabana	anamabana
as\$panamabana	ananas\$panam
nas\$panamabana	ananas\$panamab
anas\$panamab	anas\$panamaba
anas\$panamaba	as\$panamabana
ananas\$panamab	bananas\$panam
bananas\$panam	mabananas\$pana
abanan\$panam	namabananas\$p
mabananas\$pana	nanas\$panamab
amabananas\$pan	nas\$panamaba
namabananas\$pa	panamabananas\$
anamabananas\$p	s\$panamabana

Formirati sve ciklične
rotacije niske
"panamabananas\$"

Barouz-Vilerova
transformacija:
poslednja kolona =
smnpbnnaaaa\$a

Slika 7.19: Pohlepno sortiranje



Slika 7.20

\$banan	a
a\$bana	n
ana\$ba	n
anana\$b	b
banana\$	
na\$ban	a
nana\$b	a

Slika 7.21

Srtiranjem niski dobijamo prve dve kolone matrice.

Sada imamo dve kolone cikličnih niski. Zatim ponavljamo postupak - dodamo poslednju koju znamo, itd. Na kraju dobijamo rekonstruisanu celu matricu

Nisku banana\$ dobijamo tako što uzmemo sve elemente iz prvog reda posle \$.

\$banana	a \$
a\$banana	na
ana\$ban	na
anana\$b	ba
banana\$	2-mers
na\$banana	\$b
nana\$ba	an
nana\$ba	an

Slika 7.22

\$banana	a \$	\$b
a\$banana	na	a\$
ana\$ban	na	an
anana\$b	ba	an
banana\$	2-mers	\$b
na\$banana	ba	na
nana\$ba	an	na
nana\$ba	an	na

Slika 7.23

Prostorna složenost:

Rekonstrukcija niske Genome na osnovu BWT(Genome) zahteva čuvanje —Genome— kopija niske Genome, što iznosi $O(|Genome|^2)$. Poboljšanje složenosti je moguće ako primetimo nešto.

First-Last svojstvo: k-to pojavljivanje simbola u FirstColumn i k-to pojavljivanje simbola u LastColumn odgovaraju istoj poziciji simbola u niski Genome.

```
$1panamabananas1
a1bananas$panam1
a2mabananas$pan1
a3namabananas$p1
a4nanas$panama1
a5nas$panamaban2
a6s$panamabanan3
b1ananas$panama1
m1abananas$pana2
n1amabananas$pa3
n2anas$panamaba4
n3as$panamabana5
p1anamabananas$1
s1$panamabanan6
```

Slika 7.24: First-Last svojstvo

7.3.1 Efikasnija BWT dekompresija

Krenemo od simbola \$ (prvi u nizu cikličnih niski) u FirstColumn, zatim pogledamo koji simbol je u LastColumn u tom redu, nađemo ga u FirstColumn, onda za taj red nađemo koji simbol je u tom redu u LastColumn, itd. U jednom trenutku ćemo doći do \$ u LastColumn i tada smo okrenuli ceo krug i rekonstruisali celu Genome nisku. Prostorna složenost je $2 - \text{Genome} = O(-\text{Genome})$.

7.3.2 Korišćenje BWT za uparivanje šablonu

Da se podsetimo, uparivanje šablonu korišćenjem sufiksnih stabala zahtevalo je vremensku složenost od $O(-\text{Genome} + -\text{Patterns})$, prostorna $O(-\text{Genome})$. Problem je bio što je sufiksno stablo tražilo $20^* - \text{Genome}$ prostora.

Poboljšanje možemo dobiti ako umesto sufiksnog stabla koristimo BWT(Genome) kao strukturu podataka.

Postupak se sastoji od toga da krenemo od kraja niske koju tražimo i u FirstColumn nađemo taj karakter. Zatim u odgovarajućem redu u LastColumn tražimo drugi od pozadi karakter od tih kojima je u FirstColumn poslednji iz uzorka.

Zatim nađemo u FirstColumn gde su ti iz LastColumn i gledamo naredni karakter. Ako se poklapa sa trećim od pozadi, nastavljamo dalje, ako ne, nema ga. I tako dok ne pređemo ceo uzorak od kraja ka početku.

```
$1panamabananass1
a1bananas$panamm1
a2mabananas$pann1
a3namabananas$pp1
a4anas$panamabb1
a5nas$panamaban2
a6s$panamabana3
b1ananas$panamai1
m1abananas$panaa2
n1a1mabananas$pa3
n2a2nas$panamaba4
n3a3s$panamabana5
p1anamabananas$s1
s1$panamabananaa6
```

Slika 7.25: First-Last svojstvo

7.3.3 Pronalaženje uparenih šablonu

7.3.4 Pronalaženje uparenih šablonu

Problem višestrukog uparivanja šablonu:

Ulaz: Kolekcija niski Patterns i niska Genome.

Izlaz: Sve pozicije u niski Genome gde se niske iz kolekcije Patterns pojavljuju kao podnische.

Treba da nađemo pozicije. BWT ne daje ovaj podatak. Na primer, na gornjem primeru Ana se pojavljuje 3 puta, ali na kojim pozicijama?

Na kojim pozicijama se nalazi odredićemo pomoću sufiksnog niza.

Sufiksni niz je niz koji čuva početnu poziciju za svaki sufiks (niz karaktera u svakom redu matrice do simbola \$).

```

$1panamabananass1
a1bananas$panamm1
a2mabananas$pann1
a3amabananas$pp1
a4anas$panamabb1
a5as$panamabann2
a6s$panamabann3
b1ananas$panama1
m1abananas$panaa2
n1amabananas$paa3
n2anas$panamabaa4
n3as$panamabanaa5
p1anamabananas$s1
s1$panamabananaa6

```

Slika 7.26: First-Last svojstvo

13	\$ ₁ panamabananas _{s₁}
5	a ₁ bananas\$panam _{m₁}
3	a ₂ mabananas\$pan _{n₁}
1	a ₃ a mabananas\$p _{p₁}
7	a ₄ a nas\$panamab b₁
9	a ₅ a s\$panamaban _{n₂}
11	a ₆ s\$panamaban _{n₃}
6	b ₁ ananas\$panama ₁
4	m ₁ abananas\$pana _{a₂}
2	n ₁ amabananas\$pa _{a₃}
8	n ₂ anas\$panamaba _{a₄}
10	n ₃ as\$panamabana _{a₅}
0	p ₁ anamabananas\$ _{s₁}
12	s ₁ \$panamabanana _{a₆}

Slika 7.27: Sufiksni niz

Sa slike vidimo da se **ana** iz prethodnog primera pojavljuje na pozicijama 1, 7 i 9.

Prostorna složenost je $4 * |Genome|$ (ako koristimo 4B za cele brojeve kao elemente niza), što je bolje nego $20 * |Genome|$.

7.4 Približno preklapanje

Ponekad je nophodno pronaći približna uparivanja šablonu.

Ulaz: Niska Pattern, niska Genome, ceo broj d (kod višestrukog uparivanja ulaz je kolekcija niski Patterns). **Izlaz:** Sve pozicije niske Genome, gde se niska Pattern pojavljuje kao podniska sa najviše d razlika.

Traženje preklapanja radimo kao i pre, samo što sada prihvatamo i kad imamo različite karaktere (crvena slova na slici ispod), sve dok je broj razlika $j = d$.

```

# Mismatches
$1panamabananas1
a1ananas$panam1 1
a2abananas$pana1 0
a3namabananas$p1 1
a4anas$panama1 1
a5nas$panamaba2 0
a6s$panamabana3 0
b1ananas$panama1
m1abananas$pana2
n1amabananas$p3
n2anas$panamaba4
n3as$panamabana5
p1anamabananas$1
s1$panamabana6

```

Slika 7.28: Traženje približnog preklapanja za $d = 1$

Na kraju ovog primera pronašli smo 5 3-grama sa najviše jednim nepoklapanjem (prepostavili smo da je $d = 1$).

```

Suffix Array
$1panamabananas1
a1bananas$panam1 5
a2mabananas$pana1 3
a3namabananas$p1 1
a4anas$panamab1 7
a5s$panamabana2 9
a6s$panamabana3
b1ananas$panama1
m1abananas$pana2
n1amabananas$p3
n2anas$panamaba4
n3as$panamabana5
p1anamabananas$1
s1$panamabana6

```

Slika 7.29: Pozicije u genomu gde se javljaju približna preklapanja

7.5 Zadaci sa vezbi

7.5.1 TrieConstruction

```

1
2 def add_to_trie(Trie, pattern,  number_of_nodes, pattern_id):
3
4     current_node = 'root'
5
6     for c in pattern:
7
8         if c in Trie[current_node]:
9             current_node = Trie[current_node][c]
10        else:
11            if c != '$':
12                Trie['i' + str(number_of_nodes)]={}
13                Trie[current_node][c] = 'i' + str(number_of_nodes)

```

```
14         current_node = 'i' + str(number_of_nodes)
15         number_of_nodes += 1
16     else:
17         Trie[current_node][c] = pattern_id
18         Trie['P' + str(pattern_id)]={}
19
20     return (Trie, number_of_nodes)
21
22 def trie_construction(patterns):
23     Trie = {}
24     Trie['root'] = {}
25
26     number_of_nodes = 1
27
28     for i in range(len(patterns)):
29         pattern = patterns[i]
30         (Trie, number_of_nodes) = add_to_trie(Trie, pattern+'$', ,
31         ↪ number_of_nodes, i)
32
33     return Trie
34
35 def prefix_trie_pattern_matching(text, Trie):
36     v = 'root'
37     for c in text:
38         if c not in Trie[v]:
39             return False
40
41         v = Trie[v][c]
42
43         if '$' in Trie[v]:
44             return Trie[v]['$']
45
46     return False
47
48 def trie_matching(text, Trie):
49     found_patterns = []
50     while len(text) > 0:
51         res = prefix_trie_pattern_matching(text, Trie)
52         if res != False:
53             found_patterns.append(res)
54         text = text[1:]
55     return found_patterns
56
57 def main():
```

```

58
59     patterns = ['ananas', 'and', 'antenna', 'banana', 'bandana', '
60     ↪ nab', 'nana', 'pan']
61
62     query = 'bananananaspand'
63     Trie = trie_construction(patterns)
64     #print(prefix_trie_pattern_matching(query, Trie))
65     print(trie_matching(query, Trie))
66 if __name__ == "__main__":
67     main()

```

7.5.2 SufixReconstruction

```

1 def suffix_array_construction(string):
2     suffix_array = []
3     string += '$'
4     for i in range(len(string)):
5         suffix_array.append(string[i:])
6
7     suffix_array.sort()
8     return suffix_array
9
10
11 def find_neighborhood(suffix_array, mid, pattern):
12     up = mid
13     down = mid
14
15     while up >= 0 and len(suffix_array[up]) > len(pattern) and
16     ↪ suffix_array[up][:len(pattern)] == pattern:
17         up -= 1
18
19     while down < len(suffix_array) and len(suffix_array[down]) >
20     ↪ len(pattern) and suffix_array[down][:len(pattern)] == pattern
21     ↪ :
22         down += 1
23
24     positions = []
25
26     for i in range(up+1, down):
27         positions.append(len(suffix_array) - len(suffix_array[i]))
28
29     positions.sort()
30     #print('hello')
31     return positions
32

```

```

31
32 def pattern_matching_with_suffix_array(suffix_array, pattern):
33     top = 0
34     bottom = len(suffix_array) - 1
35
36
37
38     while top <= bottom:
39         mid = (top + bottom)//2
40
41         if len(suffix_array[mid]) > len(pattern):
42             if suffix_array[mid][:len(pattern)] == pattern:
43                 return find_neighborhood(suffix_array, mid, pattern)
44             ↪ )
45         if pattern < suffix_array[mid]:
46             bottom = mid - 1
47         else:
48             top = mid + 1
49
50
51 def main():
52     string = 'ananas'
53
54     suffix_array = suffix_array_construction(string)
55
56     pattern = 'an'
57
58     print(pattern_matching_with_suffix_array(suffix_array, pattern))
59     ↪ )
60
61
62 if __name__ == '__main__':
63     main()

```

7.5.3 BWT

```

1 def BWT(s):
2     matrix = []
3     s += '$'
4
5     for i in range(len(s)):
6         matrix.append(s)
7         s = s[1:] + s[0]
8

```

```
9     matrix.sort()
10    return [row[-1] for row in matrix]
11
12 def last_to_first(first_column, last_column_character,
13     ↪ character_count):
14
15     for i in range(len(first_column)):
16         if first_column[i] == last_column_character:
17             character_count -= 1
18         if character_count == 0:
19             return i
20
21
22
23
24 def bw_matching(first_column, last_column, pattern):
25     top = 0
26     bottom = len(last_column) - 1
27
28     while top <= bottom:
29         if len(pattern) > 0:
30             symbol = pattern[-1]
31             pattern = pattern[:-1]
32
33             subset = last_column[top:bottom+1]
34
35             if subset.index(symbol) != -1:
36
37                 top_index = -1
38                 bottom_index = -1
39
40                 for i in range(top, bottom+1):
41                     if symbol == last_column[i]:
42                         if top_index == -1:
43                             top_index = i
44                             bottom_index = i
45
46                 top_count = 0
47
48                 for i in range(top_index + 1):
49                     if last_column[i] == symbol:
50                         top_count += 1
51
52                 bottom_count = top_count
```

```
53
54     for i in range(top_index + 1, bottom_index + 1):
55         if last_column[i] == symbol:
56             bottom_count += 1
57
58         top = last_to_first(first_column, last_column[
59             ↪ top_index], top_count)
60         bottom = last_to_first(first_column, last_column[
61             ↪ bottom_index], bottom_count)
62
63     else:
64         return 0
65
66     else:
67         return bottom - top + 1
68
69 def main():
70     s = 'panamabananas'
71
72     last_column = BWT(s)
73
74     pattern = 'ana'
75
76     first_column = last_column[:]
77     first_column.sort()
78
79
80     print(bw_matching(first_column, last_column, pattern))
81
82 if __name__ == "__main__":
83     main()
```


Glava 8

Zašto naučnici i dalje nisu razvili vakcinu za HIV

8.1 Uvod

8.1.1 Klasifikacija HIV fenotipa

1984. godine, američka ministarka zdravlja Margaret Hekler je objavila da će vakcina za HIV biti dostupna u narednih 2 godine. 1997. godine, Bil Klinton je otvorio novi centar za istraživanje na Nacionalnom institutu zdravlja, sa ciljem da se razvije vakcina za HIV. Kompanija Merck je 2005. počela kliničko ispitivanje vakcine za HIV, ali je odustala posle 2 godine, jer su rezultati pokazali da je vakcina zapravo povećala rizik od dobijanja HIV-a kod nekih primalaca vakcine.

Danas, uprkos ogromnim investicijama i istraživanjima, daleko smo od razvijanja vakcine za HIV, a 35 miliona ljudi žive s tom bolesti. Naučnici su napravili ogroman napredak u razvoju antiretroviralne terapije, koja predstavlja mešavinu lekova koji stabilizuju simptome zaraženog pacijenta. Međutim, ova terapija ne leči sidu i ne može da zaustavi širenje HIV-a tako da ne predstavlja pravu vakcincu za sidu.

Klasične vakcine protiv virusa su često napravljene od proteina virusa. Ove vakcine stimulišu čovekov imuni sistem da prepozna virusne omotače proteina kao strane, da ih uništi i da sačuva podatke o njima, da bi imuni sistem mogao kasnije da ih identificuje i iskoreni.

Međutim, virusni omotači proteina HIV virusa mogu biti ekstremno promenljivi, zato što virus mora da mutira brzo da bi preživeo. Virus HIV-a kod neke osobe evoluira vrlo brzo da bi izbegao imuni sistem čoveka. Takođe, uzorci HIV-a uzeti od različitih pacijenata su pokazali da oni imaju podtipove koji se veoma razlikuju. Dakle, uspešna vakcina za HIV mora biti dovoljno širokog spektra da pokrije sve ove različitosti.

HIV ima samo devet gena i u ovom poglavlju se fokusiramo na *env* gen koji brzo mutira. Protein koji kodira *env* gen ulazi u **glycoprotein gp120** i **glycoprotein gp41**.

S obzirom da HIV mutira tako brzo, različiti izolati HIV-a mogu imati različite fenotipe, koji onda zahtevaju različite mešavine lekova. HIV virusi se mogu podeliti

na brzo replicirajuće (SI) izolate i sporo replicirajuće NSI izolate. Tokom infekcije, proteini virusa kao što je gp120 koje HIV koristi da uđe u ćelije se prenose do površine ćelije, gde mogu da prouzrokuju da se ta ćelija spoji sa susednom ćelijom. To uzrokuje da desetine ljudskih ćelije spoje svoje ćelijske membrane u jedan veliki, nefunkcionalni syncytium ili u abnormalnu multinukleaturnu ćeliju. Na ovaj način, inficirajući samo jednu ćeliju, biće ubijene mnoge ljudske ćelije.

Ograničenja u poravnanju sekvenci

Pre nego što biologičari uopšte mogu da počnu da proučavaju pitanje predviđanja HIV fenotipa koristeći gp120 sekvene, oni se suočavaju sa problemom konstrukcije preciznog poravnjanja ovih sekvenci. Čak i jedna pogrešno poravnanje, koje postavlja amino kiselinu na poziciju utičući na SI/NSI fenotip, može prouzrokovati pogrešnu klasifikaciju HIV fenotipa. Iz poglavlja 5, već znamo da je konstrukcija višestrukog poravnjanja sekvenci koja divergiraju težak algoritamski problem.

Problem formulacije višestrukog poravnjanja uveden u poglavlju 5 ne pruža adekvatnu translaciju biološkog problema PHV klasifikacije u algoritamski problem. Zbog toga moramo smisliti novu formulaciju problema poravnjanja sekvenci koja će dovesti statistički solidne analize gp120 proteina.

8.2 Pronalaženje CG ostrva

Početkom dvadesetog veka, Phoebus Levene je otkrio četiri nukleotida od kojih se sastoji DNK. U to vreme, vrlo malo se znalo o DNK. Zbog toga, Levene je sumnjao da DNK može da čuva genetske informacije koristeći samo četiri slova i postavio je hipotezu da se u DNK nalazi gotovo jednak broj adenina, citozina, guanina i timina. Jedan vek kasnije, znamo da komplementarni nukleotidi na suprotnim obalama DNK imaju jednaku frekvenciju osnovnog uparivanja, ignorujući ekstremno retke greške osnovnog uparivanja. Međutim, nije tačno da su frekvencije nukleotida približno iste na jednoj obali DNK. Različite vrste imaju različite **CG-sadržaje**, ili procenat citozina i guanina u genomu.

Mogli bismo očekivati da se svaki od dinukleotida CC, CG, GC, GG u ljudskom genomu javlja sa frekvencijom od $0.21 * 0.21 + 4.41\%$. Međutim, frekvencija CG u ljudskom genomu je samo 1%. Ovaj dinukleotid je toliko redak zbog **metilacije**.

Definicija 8.1. *Metilacija je dodavanje metil (CH_3) grupe na citozin (često u okviru CG dinukleotida).*

Rezultujući metilovani citozin ima tendenciju da deaminuje u timin. Kao rezultat metilacije, CG je najređi dinukleotid u mnogim genomima. Metilacija je često izostavljena u genima u regionima pod nazivom **CG-ostrva** (CG se često pojavljuje).

U prvom pokušaju za nalaženje ovakvih gena, kako bismo tražili CG-ostrva? Naivan pristup traženju CG-ostrva bi bio da se pomera prozor kroz genom, proglašavajući prozore sa većom frekvencijom CG, potencijalnim CG-ostrvima. Mane ovog pristupa bi bile iste kao i pri pomeranju prozora da bi se odredio koji novčić

je krupije koristio u kom trenutku. Ne znamo koliki će prozor biti i zato nije jasno kako odabratи veličinu prozora za detekciju CG-ostrva. Takođe, različiti prozori mogu klasifikovati iste pozicije u genomu različito.

8.3 Nepoštena kockarnica

8.3.1 Kockanje sa Jakuzama

Japanska kriminalna organizacija pod nazivom jakuza potiče iz od grupe putujućih kockara iz 18. veka koji su se nazivali bakut ("jakuza" je naziv za gubitničku ruku u Japanskoj kartaškoj igri). Jedna od najpopularnijih igara koju su bakuto organizovali u svojim kazinoima se zvala Čo-Han. U ovoj igri, koja se bukvalno prevodi kao "jednake šanse", krupije baca dve kockice, a igrač se kladi na to da li će suma kockica biti paran ili neparan broj.

Iako je igranje Čo-Han igre u jakuzinim kockarnicama veoma zanimljivo, možemo takođe igrati i igru koja se zove "glava ili pismo", tako što se baca novčić u vazduh i pogoda se ishod. Pretpostavimo da se iz nekog razloga više ljudi kladi na pismo nego na glavu u ovoj igri. U tom slučaju bi nepošten krupije mogao da iskoristi otežani novčić koji ima veću verovatnoću da padne na glavu nego na pismo. Mi ćemo pretpostaviti da otežani novčić ima verovatnoću da padne na glavu 3/4.

Pitanje: Recimo da igramo igru pismo ili glava 100 puta, i novčić padne na glavu 63 puta. Da li možemo da kažemo da krupije vara? Da li je korišćen fer ili otežan novčić? **Nagoveštaj:** 63 je bliže 75 nego 50!

Pitanje nije dobro formulisano, jer bilo koji novčić može da proizvede bilo koji niz bacanja. Da li možemo da utvrdimo koji novčić je verovatnije korišćen? Zapišimo verovatnoću padanja pisma ("T") i glave ("H") za fer novčić (F) kao:

$$Pr_F("H") = 1/2 Pr_F("T") = 1/2 \quad (8.1)$$

i verovatnoća otežanog novčića (B):

$$Pr_B("H") = 3/4 Pr_B("T") = 1/4 \quad (8.2)$$

Kako su bacanja novčića nezavisni događaji, verovatnoća da će n bacanja fer novčića proizvesti niz $x = x_1x_2\dots x_n$ sa k pojavljivanja glave je:

$$Pr(x|F) = \prod_{i=1}^n Pr_F(x_i) = (1/2)^n. \quad (8.3)$$

Verovatnoća da će otežani novčić da proizvede isti niz je:

$$Pr(x|B) = \prod_{i=1}^n Pr_B(x_i) = (1/4)^{(n-k)} \cdot (3/4)^k = 3^k / 4^n. \quad (8.4)$$

Ako je $Pr(x|F) > Pr(x|B)$, onda je veća verovatnoća da je krupije koristio fer novčić, a ako je $Pr(x|F) < Pr(x|B)$, onda je obrnuto. Kada su jednaki $Pr(x|F) = Pr(x|B)$ tada je:

$$(1/2)^n = (1/4)^{(n-k)} \cdot (3/4)^k \rightarrow 2^n = 3^k \rightarrow k = \log_2 3 \cdot n \rightarrow k \approx 0.632 \cdot n \quad (8.5)$$

Dakle ukoliko je mera odnosa $k/n < 1/\log_2 3$, tada je i $\Pr(x|F) > \Pr(x=B)$. Iako je 63 bliže 75 nego 50, fer novčić će sa većom verovatnoćom dati 63 glave u 100 bacanja.

8.3.2 Dva novčića u krupijeovom rukavu

U bakuto kockarnicama, Čo-Han krupije bi skinuo svoju majicu u toku igre, kako bi skinuo sa sebe sumnju da mulja sa kockicama. Mi ćemo ipak pretpostaviti da u igri glava ili pismo, nepošteni krupije nosi majicu i drži oba novčića u svom rukavu i može da ih neprimetno menja u bilo kom trenutku. Pošto ne želi da bude uhvaćen kako zamenjuje novčice, on ih menja samo povremeno, pretpostavimo sa verovatnoćom 0.1 nakon svakog bacanja.

Pitanje: Nakon niza bacanja novčića, možemo li reći kada je krupije koristio fer coin a kada otežani novčić?

Problem 8 (Kazino problem.). Za dati niz bacanja novčića, odrediti kada je krupije koristio fer a kada otežani novčić.

- *Ulaz: Niz $x = x_1x_2 \dots x_n$ bacanja dobijenih od novčića F (fer) i B (otežani).*
- *Izlaz: Niz $\pi = \pi_1\pi_2 \dots \pi_n$, gde je svako π_i jednako ili F ili B što znači da je x_i dobijeno bacanjem fer ili otežanog novčića, redom.*

Nažalost, ni ovo nije dobro definisan problem. Svaki ishod bacanja novčića može biti dobijen bilo kojom permutacijom fer i otežanog novčića! HHHHH je moglo biti dobijeno od BBBBB, FFFFF, FBFBF, itd.

Neophodan je način za ocenjivanje različitih scenarija: BBBBB, FFFFF, FBFBF, itd. zavisno od toga koliko je svaki od njih verovatan.

Kako da ispitamo i ocenimo 2^n mogućih scenarija?

Jedan pristup ovom problemu da pogodimo koji novčić je verovatnije krupije koristio u svakom bacanju je da uzmem "okvir prozora" (dužine $t < n$) duž niza bacanja $x = x_1 \dots x_n$ i da izračunamo meru odnosa fer i otežanog novčića (kao u gornjem primeru) u okviru svakog prozora. Ukoliko je mera odnosa u okviru prozora manja od nule, onda je veća verovatnoća da krupije koristi otežani novčić u okviru prozora, inače je obrnuto.

Postoji dva problema sa metodom "okvira prozora". Prvo je da nemamo vidan način da odaberemo dužinu okvira prozora. Drugo, okviri prozora koji se preklapaju mogu klasifikovati isti ishod uzrokovani i otežanim i fer novčićem. Na primer, ako je $x = "HHHHHTTHHHTTTT"$, onda prozor $x_1 \dots x_{10} = "HHHHHTTHH"$ ima negativnu meru odnosa, a onda prozor $x = x_6 \dots x_{15} = "TTHHHTTTT"$ ima pozitivnu meru odnosa. Koji je novčić krupije koristio na bacanjima $x_6 \dots x_{10}$?

8.4 Skriveni Markovljevi modeli

8.4.1 Od bacanja novčića do Skrivenog Markovljevog Modela

Naš cilj je razvijemo koncept koji modeluje nepoštenog krupijea i potragu za CG-ostrvima u genomu. Krupijea možemo posmatrati kao mašinu koja ima k skrivenih stanja (F i B). U svakom koraku, emituje simbol (H ili T) iz jednog od svojih skrivenih stanja. Dok je u određenom stanju, mašina donosi dve odluke:

- Koji simbol će emitovati?
- U koje skriveno stanje će nakon toga preći?

Mašina odgovara na prvo pitanje tako što izabere proizvoljan broj između stanja F i B , sa verovatnoćom 0.9 da će ostati u trenutnom stanju i verovatnoćom 0.1 da će promeniti stanje. Mašina odgovara na drugo pitanje tako što bira između simbola H i T sa verovatnoćama koje zavise od toga u kom stanju je trenutno. Naš cilj je da zaključimo koja je najverovatnija sekvenca stanja mašine analizirajući sekvence simbola koje emituje.

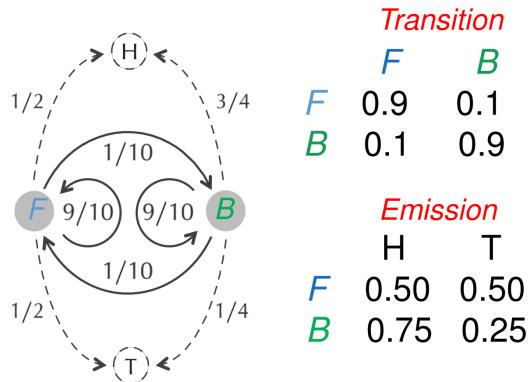
Na ovaj način smo pretvorili krupijea u apstraktnu mašinu koja se zove **Skriveni Markovljev Model** (HMM). Razlika između mašine za bacanje novčića i generalnog koncepta HMM-a je što će HMM kasnije imati proizvoljan broj stanja i može da ima proizvoljnu distribuciju verovatnoća, odlučujući u koje stanje da pređe i koje simbole da emituje.

8.4.2 HMM dijagrami

HMM se definiše kao skup četiri objekta:

- azbuka emitovanih simbola Σ (H i T)
- skup skrivenih stanja (F i B)
- Matrica verovatnoće prelaska: $Transition = (transition_{l,k})$: $|States| \times |States|$ matrica **verovatnoća prelaska** (iz stanja l u stanje k)
- Matrica emisionih verovatnoća: $Emision = (emision_k(b))$: $|States| \times |\Sigma|$ matrica **emisionih verovatnoća** (emitovanje simbola b u stanju k)

Kao što je prikazano na slici 8.1, HMM se može prikazati kao HMM dijagram, graf gde je svako stanje predstavljeno jednim punim čvorom. Usmerene pune grane povezuju svaki par čvorova, kao i svaki čvor sa sobom. Svaka takva grana je obeležena verovatnoćom prelaska iz jednog stanja u drugo. HMM dijagram takođe ima i isprekidane čvorove koji predstavljaju simbole azbuke Σ i isprekidanje grane koje povezuju svako stanje sa svojim isprekidanim čvorom. Svaka takva grana je obeležena verovatnoćom da će HMM emitovati taj simbol dok je u tom stanju.



Slika 8.1: HMM dijagram

Definicija 8.2. *Skrivena putanja* je niz $\pi = \pi_1, \dots, \pi_n$ stanja kroz koje HMM prolazi.

Slika 8.1 prikazuje primer gde nepošteni krupije HMM proizvodi sekvencu $x = \text{"THTHHHTHTTH"}$ sa skrivenom putanjom $n = FFFBBBFFFF$. Fer novčić je korišćen za prva tri bacanja i za poslednja tri bacanja, a otežani novčić se koristi za pet bacanja između.

Uvedimo sledeće jednakosti:

- $Pr(x, \pi)$: zajednička verovatnoća da dati HMM polazi kroz stanja π i emituje nisku $x = x_1x_2\dots x_n$.
- $Pr(x|\pi)$: uslovna verovatnoća da HMM emituje nisku x nakon prolaska kroz skrivenu putanju π .
- $Pr(x, \pi) = Pr(x|\pi) * Pr(\pi)$

Da bi se izračunalo $Pr(x, \pi)$, prvo moramo da izračunamo $Pr(\pi)$. Neka $Pr(\pi_i \rightarrow \pi_{i+1})$ označava verovatnoću prelaska HMM-a iz stanja π_i u stanje π_{i+1} . Verovatnoća za π je jednaka proizvodu verovatnoća prelaska

$$Pr(\pi) = \prod_{i=1}^n Pr(\pi_{i-1} \rightarrow \pi_i) = \prod_{i=1}^n transition_{\pi_{i-1}, \pi_i} \quad (8.6)$$

Problem 9 (Problem verovatnoće skrivene putanje.). Izračunati verovatnoću skrivene putanje HMM-a.

Ulaz: Skrivena putanja π i model HMM ($\Sigma, States, Transition, Emission$).

Izlaz: Verovatnoća date putanje, $Pr(\pi)$.

Da bismo izračunali $Pr(x|\pi)$ za neki HMM, označićemo sa $Pr(x_i|\pi_i)$ verovatnoću emitovanja $emission_{\pi_i}(x_i)$ da je x_i emitovan kada je HMM bio u stanju π_i . Ko rezultat toga, za neku putanju π , HMM emituje string x sa verovatnoćom jednakom proizvodu verovatnoća emitovanja na toj putanji,

$$Pr(x, \pi) = \prod_{i=1}^n Pr(x_i | \pi_i) = \prod_{i=1}^n emission_{\pi_i}(x_i) \quad (8.7)$$

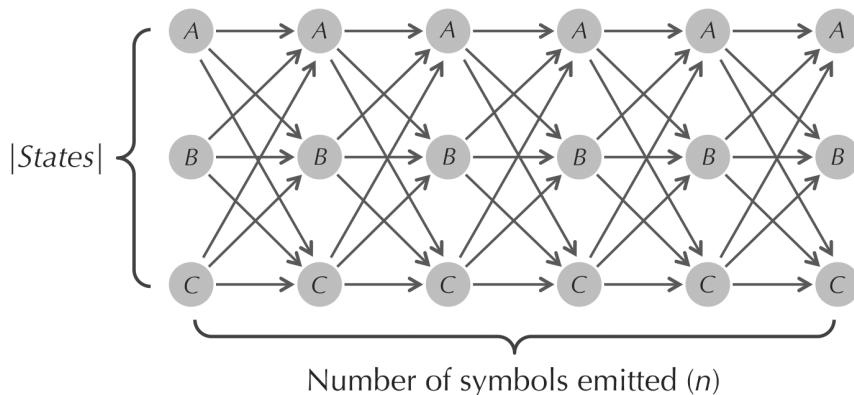
Problem 10 (Problem verovatnoće ishoda za datu skrivenu putanju). Izračunati verovatnoću da dati HMM emituje datu nisku za datu skrivenu putanju.
Ulaz: Niska $x = x_1, \dots, x_n$ koju emituje dati $HMM(\Sigma, States, Transition, Emission)$ i skrivena putanja $\pi = \pi_1, \dots, \pi_n$.
Izlaz: Uslovna verovatnoća $Pr(x|\pi)$ da će dati HMM emitovati nisku x prateći skrivenu putanju π .

8.5 Problem dekodiranja

8.5.1 Viterbi graf

Problem 11 (Problem dekodiranja). Naći optimalnu skrivenu putanju sa kojom je dati HMM emitovao datu nisku.
Ulaz: Niska $x = x_1 \dots x_n$ koju emituje $HMM(\Sigma, States, Transition, Emission)$.
Izlaz: Putanja π koja maksimizuje verovatnoću $Pr(x, \pi)$ po svim mogućim putanjama π za ovaj HMM.

Da bi rešio problem dekodiranja, Andrew Viterbi je koristio Menhetn graf inspirisan HMM-om. Za HMM koji emituje string od n simbola $x = x_1 \dots x_n$, čvorovi HMM-ovog Viterbi grafa se dele na —States— vrsta i n kolona (slika 8.2). Dakle, čvor (k, i) reprezentuje stanje k i i -ti emitovani simbol. Svaki čvor je povezan sa svim čvorovima iz kolone s njegove desne strane; grana koja povezuje $(l, i-1)$ sa (k, i) odgovara prelasku iz stanja l u stanje k (sa verovatnoćom $transition_{l,k}$) i zatim emitovanju simbola x (sa verovatnoćom $emission_k(x_i)$). Kao rezultat toga, sve putanje koja povezuje čvor u prvoj koloni Viterbi grafa sa čvorom u poslednjoj koloni, odgovara skrivenoj putanji $\pi = \pi_1 \dots \pi_n$.



Slika 8.2: Menhetn graf za problem dekodiranja

Težina grane koja povezuje $(l, i - 1)$ i (k, i) u Viterbi grafu je jednaka

$$Weight_i(l, k) = transition_{\pi_i, \pi_{i-1}} * emission_{\pi_i}(x_i) \quad (8.8)$$

Zatim, definišemo **proizvod težina** putanja u Viterbi grafu, kao proizvod težina njegovih grana. Za putanju od najlevlje kolone do najdesnije(da li se ovo kaže ovako ili sam nepismen?) kolone u Viterbi grafu koja odgovara skrivenoj putanji n , ovaj proizvod je jednak proizvodu $n - 1$ članova,

$$\prod_{i=2}^n transition_{\pi_i, \pi_{i-1}} * emission_{\pi_i}(x_i) = \prod_{i=1}^{n-1} Weight_i(l, k). \quad (8.9)$$

8.5.2 Viterbi algoritam

Primeničemo algoritam dinamičkog programiranja da bismo rešili problem dekodiranja. Prvo, definišimo $s_{k,i}$, što predstavlja proizvod težina optimalne putanje (putanje sa najvećom težinom proizvoda) od *izvora* do čvora (k, i) . Viterbi algoritam je zasnovan na činjenici da prvih $i - 1$ grana optimalne putanje od izvora do (k, i) moraju formirati optimalnu putanju od izvora do $(l, i - 1)$ za neko (nepoznato) stanje l . Ovo zapažanje proizvodi sledeću jednačinu:

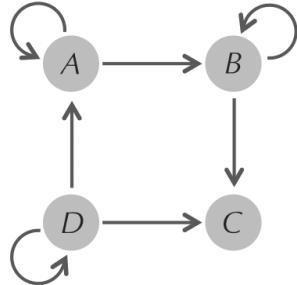
$$\begin{aligned} s_{k,i} &= \max_{all states l} \{s_{l,i-1} \cdot (weight of edge between nodes (l, i - 1) and (k, i))\} \\ &= \max_{all states l} \{s_{l,i-1} \cdot Weight_i(l, k)\} \\ &= \max_{all states l} \{s_{l,i-1} \cdot transition_{\pi_{i-1}, \pi_i} \cdot emission_{\pi_i}(x_i)\} \end{aligned} \quad (8.10)$$

8.5.3 Brzina Viterbi algoritma

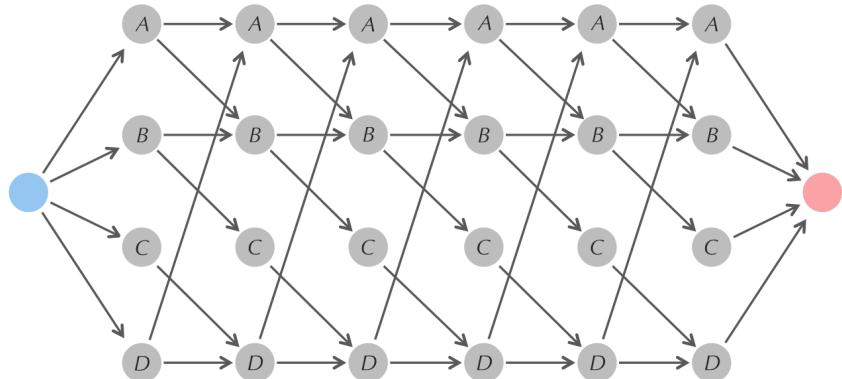
Možemo da posmatramo problem dekodiranja kao još jednu isntancu problema najduže putanje u DAG problemu iz 5. poglavља, zato što putanja π koja maksimizira proizvod težina $\prod_{i=1}^n Weight_i(\pi_{i-1}, \pi_i)$ takođe maksimizira logaritam ovog proizvoda, koji je jednak $\sum_{i=1}^n \log(Weight_i(\pi_{i-1}))$. Prema tome, možemo da zamenimo težine svih grana u Viterbi grafu njihovim logaritmima. Nalaženje najdužeg puta u rezultujućem grafu će odgovarati putanji maksimalnih težina proizvoda u originalnom Viterbi grafu. Iz ovog razloga, vreme izvršavanja Viterbi algoritma je linearno u odnosu na broj grana u Viterbi grafu. Broj ovih grana je $|States|^2 \cdot n$ gde je n broj emitovanih simbola.

U praksi, mnogo HMM-ova ima **zabranjene prelaze** između nekih stanja. Za takve prelaze, možemo da obrišemo odgovarajuće grane iz HMM dijagrama (slika 8.3). Ova operacija dovodi do ređeg Viterbi grafa (slika 8.4), što dovodi do smanjenja vremena izvršavanja Viterbi algoritma, s obzirom da je vreme izvršavanja

algoritma za pronalaženje najduže putanje u DAG-u linearno u odnosu na broj grana u tom DAG-u.



Slika 8.3: HMM dijagram sa nekim zabranjenim stanjima kao npr. od A do D ili od C do samog sebe



Slika 8.4: Viterbi graf za HMM sa prethodne slike koji emituje string dužine 6

8.6 Računanje najverovatnijeg ishoda HMM-a

Dinamičko programiranje nam pomaže da odgovorimo na pitanje proširivanja HMM-a i preko najverovatnije skrivene putanje. Mi možemo da izračunamo verovatnoću $Pr(\pi)$ skrivene putanje π . Ali šta je sa $Pr(x)$, koja je verovatnoća da HMM emituje string x ?

Problem 12 (Problem verovatnoće ishoda). *Izračunati verovatnoću da HMM emituje datu nisku.*

Ulas: Niska $x = x_1 \dots x_n$ koju emituje $HMM(\Sigma, States, Transition, Emission)$.

Izlaz: Verovatnoća $Pr(x)$ da model HMM emituje nisku x .

Već smo zaključili da je $Pr(x)$ jednak sumi $Pr(x, \pi)$ za sve skrivene putanje π . Međutim, broj putanja u Viterbi grafu je eksponencijalan u odnosu na broj

emitovanih stringova x , tako da možemo da koristimo dinamičko programiranje kao brži način da izračunamo $Pr(x)$.

Neka je $forward_{k,i}$ proizvod svih putanja od *izvora* do čvora (k, i) u Viterbi grafu; treba uočiti da je $forward_{sink}$ jednak $Pr(x)$. Da bismo izračunali $forward_{k,i}$, podelićemo sve putanje koje povezuju *izvor* i čvor (k, i) na $|States|$ podskupova, gde svaki podskup sadrži one putanje koje prolaze kroz čvor $(l, i-1)$ (sa težinom proizvoda $forward_{l,i-1}$), dok ne dođemo do (k, i) za neko l između 1 i $|States|$. Dakle, $forward_{k,i}$ je suma $|States|$ članova,

$$\begin{aligned} forward_{k,i} &= \sum_{\text{all states } l} forward_{l,i-1} \cdot \text{težina grane koja povezuje } (l, i-1) \text{ i } (k, i) \\ &= \sum_{\text{all states } l} forward_{l,i-1} \cdot Weight_i(l, k) \end{aligned} \tag{8.11}$$

Treba primetiti da je jedina razlika između ove jednačine i Viterbi jednačine,

$$s_{k,i} = \max_{\text{all states } l} \{s_{l,i-1} \cdot Weight_i(l, k)\}, \tag{8.12}$$

je da se maksimizacija u Viterbi algoritmu menja u sumaciju. Sada možemo rešiti problem verovatnoće ishoda računajući $forward_{sink}$, što je jednako

$$\sum_{\text{all states } k} forward_{k,n} \tag{8.13}$$

Sada možemo da izračunamo $Pr(x)$ za emitovani string x , logično pitanje je pronaći najverovatniji takava string. Za problem nepoštenog krupijea, ovo odgovara pronalasku najverovatnije sekvene bacanja novčića za sve moguće sekvene, za fer i otežani novčić koji krupije može koristiti.

Problem 13 (Problem najverovatnijeg ishoda). *Naći najverovatniji string koji emituje HMM.*

Ulaz: $HMM(\Sigma, States, Transition, Emission)$ i ceo broj n .

Izlaz: *Najverovatniji string $x = x_1 \dots x_n$ koji emituje HMM, odnosno, string koji maksimizira verovatnoću $Pr(x)$ da će HMM emitovati x .*

8.7 Profilni algoritmi za poravnjanje sekvenci

8.7.1 Kako su HMM povezani za poravnjanje sekvenci?

Za datu familiju povezanih proteina, možemo proveriti da li nova sekvenca proteina pripada ovoj familiji, konstruišući parno poravnjanje između novo sekveniranog proteina i svakog člana familije. Ako jedno od rezultujućih poravnjanja da rezultat iznad nekog strogog praga, onda možemo pretpostaviti da novi protein pripada familiji. Međutim, ovaj pristup može neuspšeno da identificuje proteine koji su udaljeno povezani. Ako sekvenca ima slabe povezanosti sa velikim brojem

članova familije, onda ona najverovatnije pripada toj familiji. Problem je poravnati novi protein sa *svim* članovima familije odjednom. Da bismo ovo postigli, moramo da pretpostavimo da već imamo konstruisano višestruko poravnanje familije proteina. Srećom, često će biti očigledno da dva proteina dolaze iz iste familije. Shodno tome, biolozi često počinju konstruišući poravnanje proteina koji su nesumnjivo povezani, koje je obično lako poravnati, cak i koristeći jednostavne metode poravnanja koje smo predstavili u poglavlju 5. Slika 8.5 (prvi deo) prikazuje 5x10 poravnanje *Alignment* koje predstavlja hipotetičku familiju proteina. Primetimo da 6. i 7. kolona ovog poravnanja sadrže mnogo praznih “-“ simbola, i verovatno ne predstavljaju značajne karakteristike familije. Shodno tome, biolozi često ignoriraju kolone za koje je deo ovih praznih “-“ simbola veći ili jednak **pragu brisanja kolone θ** . Brisanje kolona rezultuje semenom poravnanju *Alignment** 5x8 predstavljenom na slici 8.5 (drugi deo). Dato semeno poravnanje *Alignment** predstavlja familiju povezanih proteina i naš cilj je da izgradimo HMM koji realistično modelira sklonosti simbola u *Alignment** koji je predstavljen profilnom matricom PROFILE(*Alignment**) na slici 8.5 (treći deo). Umesto da razmišljamo o poravnavanju postojećeg semenog poravnanja do datog *Text-a* (koji predstavlja novi protein), mi ćemo umesto da razmišljamo o tome da izračunamo verovatnoću

	1	2	3	4	5	6	7	8	
<i>Alignment</i>	A	C	D	E	F	A C	A	D	F
	A	F	D	A	-	- C	C	F	
	A	-	-	E	F D -	F	D	C	
	A	C	A	E	F - -	A	-	C	
	A	D	D	E	F A A	A	D	F	
<i>Alignment*</i>	A	C	D	E	F	A	D	F	
	A	F	D	A	-	C	C	F	
	A	-	-	E	F	F	D	C	
	A	C	A	E	F	A	-	C	
	A	D	D	E	F	A	D	F	
PROFILE(<i>Alignment*</i>)	A	1	0	0	1/5	0	3/5	0	0
	C	0	2/4	0	0	0	1/5	1/4	2/5
	D	0	1/4	3/4	0	0	0	3/4	0
	E	0	0	0	4/5	0	0	0	0
	F	0	1/4	0	0	1	1/5	0	3/5

$M_1 \longrightarrow M_2 \longrightarrow M_3 \longrightarrow M_4 \longrightarrow M_5 \longrightarrow M_6 \longrightarrow M_7 \longrightarrow M_8$

Slika 8.5: 5x10 višestruko poravnjanje (prvi deo). Uklanjamo kolone ukoliko broj praznina “-“ prelazi θ . 5x8 semeno poravnjanje (drugi deo). Izbacili smo kolone. Profilna matrica semenog poravnjanje (treći deo) i jednostavni HMM dijagram koji modelira gornji PROFIL. Semeni algoritam je dobijen od originalnog poravnanja ignorisanjem kolona (osenčenih sivom bojom). U ovom slučaju ignorisemo kolone čiji je deo praznih “-“ simbola veći ili jednak pragu $\theta = 0.35$. Kako bismo jasnije prikazali vezu između poravnanja i semenog poravnjanja, razdvojili smo prvih 5 kolona u semenom poravnjanju od poslednje 3 kolone i numerisali te kolone iznad originalnog poravnjanja. Stanja pogotka MATCH(i) su skraćena kao M_i . HMM ima samo jednu moguću putanju; u svom početnom stanju MATCH(1), verovatnoća prelaska iz stanja MATCH(i) do stanja MATCH(i+1) je jednak 1 za svako i svi drugi prelasci su zabranjeni. Emisione verovatnoće su jednake frekvencijama u profilu, npr. emisione verovatnoće za M_2 su 0 za A, 2/4 za C, 1/4 za D, 0 za E i 1/4 za F.

da HMM emituje *Text*. Ako je HMM dobro dizajniran, onda što je slicciji *Text* nizu u *Alignment**, to će verovatnije biti emitovan od strane HMM-a.

Prvo ćemo konstruisati jednostavan HMM koji tretira kolone *Alignment**-a kao k sekvensijalno povezanih stanja koja ćemo nazvati **stanja pogotka** (slika 8.5 četvrti deo), označeni MATCH(1), ..., MATCH(k). Kada HMM uđe u stanje MATCH(i), tada emituje simbol x_i sa verovatnoćom jednakom frekvenciji ovog simbola u i-toj koloni PROFILE(*Alignment**). HMM se onda prebacuje u stanje MATCH(i+1) sa prelaznom verovatnoćom jednakom 1.

Slicnost pogotka između *Alignment**-a i *Text*-a je verovatnoća $\text{Pr}(\text{Text})$ da HMM za *Alignment** emituje *Text*. Ovaj rezultat je jednak proizvodu frekvencija u PROFILE(*Alignment**) koji odgovaraju svakom simbolu iz *Text*-a. Na primer, verovatnoća da HMM na slici 8.5 emituje ADDAFFDF je:

$$1 \cdot \frac{1}{4} \cdot \frac{3}{4} \cdot \frac{1}{5} \cdot 1 \cdot \frac{1}{5} \cdot \frac{3}{4} \cdot \frac{3}{5} = 0.003375. \quad (8.14)$$

HMM koji smo prikazali rezultuje svaku kolonu na slici 8.5 drugačije i do određenog stepena, što je sličniji *Text Alignment*-u*, to je veći rezultat sličnosti.

Međutim, ovaj HMM ima samo jednu skrivenu putanju, i nudi jednostavan pogled višestrukih poravnanja zato što ne uracunava inserciju i brisanja. Na kraju, *Text* se može “poravnati” u odnosu na *Alignment** ako je dužina *Text*-a tačno jednak broju kolona u *Alignment**-u (slika 8.6). Ipak mi ćemo iskoristiti ovaj ograničeni HMM kao temelj za moćnije HMM-ove.

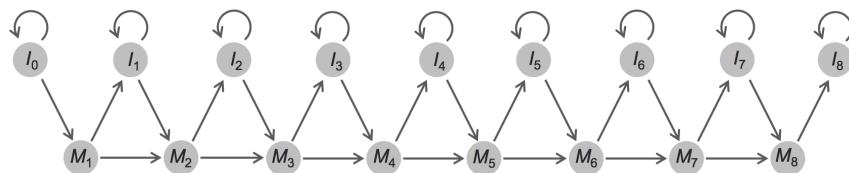
	A	C	D	E	F	A	D	F
	A	F	D	A	—	C	C	F
<i>Alignment*</i>	A	—	—	E	F	F	D	C
	A	C	A	E	F	A	—	C
	A	D	D	E	F	A	D	F
<i>Text</i>	A	D	D	A	F	F	D	F
emission probability	1	1/4	3/4	1/5	1	1/5	3/4	3/5

Slika 8.6: Poravnanje *Text* = ADDAFFDF u odnosu na semeno poravnanje *Alignment** predstavljeno kao jednostavan HMM na slici 8.5. Ovaj HMM je ograničen zato što nismo u mogućnosti da poravnamo nizove dužine različite od 8.

8.7.2 Građenje profilnog HMM-a

Poboljšani HMM koji ćemo predstaviti se zove **profilni HMM**. Sa datim višestrukim poravnanjem *Alignment* i pragom brisanja kolona θ koji koristimo da dođemo do *Alignment**-a, označićemo ovaj profilni HMM kao HMM(*Alignment*, θ). Za dati niz *Text* da se poravna u odnosu na postojeće semeno poravnanje, naš cilj je da pronađemo optimalni skriveni put u profilnom HMM-u tako što ćemo rešiti Problem Dekodiranja za ovaj HMM i emitovani niz *Text*.

Prvo dodajemo $k+1$ **insercionih stanja**, označenih kao $\text{INSERTION}(0), \dots, \text{INSERTION}(k)$ (slika 8.7). Ulaženje u $\text{INSERTION}(i)$ dopušta profilnom HMM-u da emituje dodatni simbol nakon posećivanja i -te kolone $\text{PROFILE}(\text{Alignment}^*)$ -a i pre ulaska u $(i+1)$ -tu kolonu. Povezujemo $\text{MATCH}(i)$ sa $\text{INSERTION}(i)$ i $\text{INSERTION}(i)$ sa $\text{MATCH}(i+1)$. Kako bismo dopustili višestruke insertovane simbole između kolona $\text{PROFILE}(\text{Alignment}^*)$ -a, povezaćemo $\text{INSERTION}(i)$ samu sa sobom.

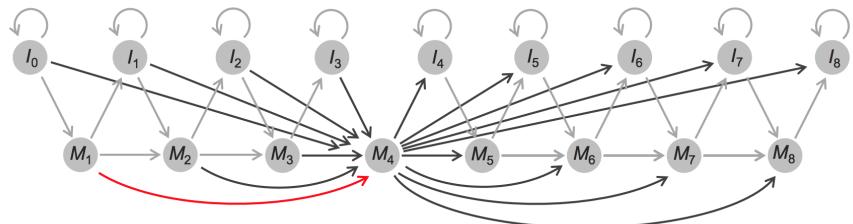


Slika 8.7: HMM dijagram za semeno poravnanje sa slike 8.5 sa pogotcima i insercionim stanjima, skraćeni kao M i I, redom. Stanja I_0 i I_8 modeluju insercije simbola koje se dešavaju pre početka i kraja *Alignment**-a, redom.

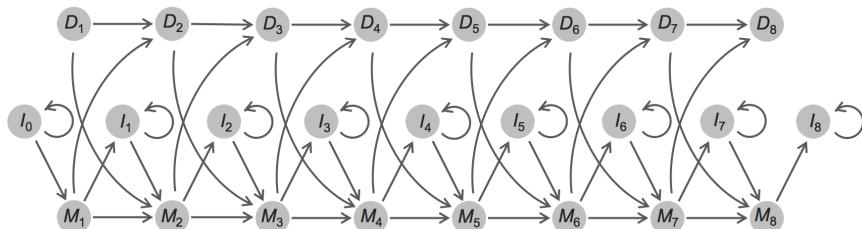
Pitanje: Da li možemo da iskoristimo HMM na slici 8.7 da poravnamo niz *Text*

dužine manje od 8?

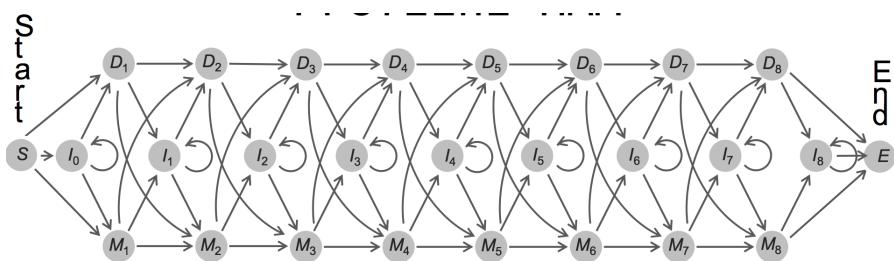
Nakon modelovanja insercija novih simbola u PROFILE(*Alignment*^{*}), treba da modelujemo i “delecije“ koja omogućavaju profilnom HMM-u da preskoči kolone PROFILE(*Alignment*^{*})-a. Jedan način modeliranja ovih delecija je da dodamo ivice koje povezuju svako stanje u profilnom HMM-u sa svakim stanjem desno od njega (slika 8.8).



Slika 8.8: Dodavanjem ivica koje povezuju svako stanje u profilnom HMM-u sa slike 8.7 sa svakim stanjem desno od njega, možemo preskočiti kolone *Alignment*-a kada poredimo *Text* u odnosu na ovo poravnanje. Gornji HMM dijagram označava da sve ivice vode u i iz MATCH(4).



Slika 8.9: Dodavanje stanja deleciji (skraceno kao D_i) profilnom HMM dijagramu.



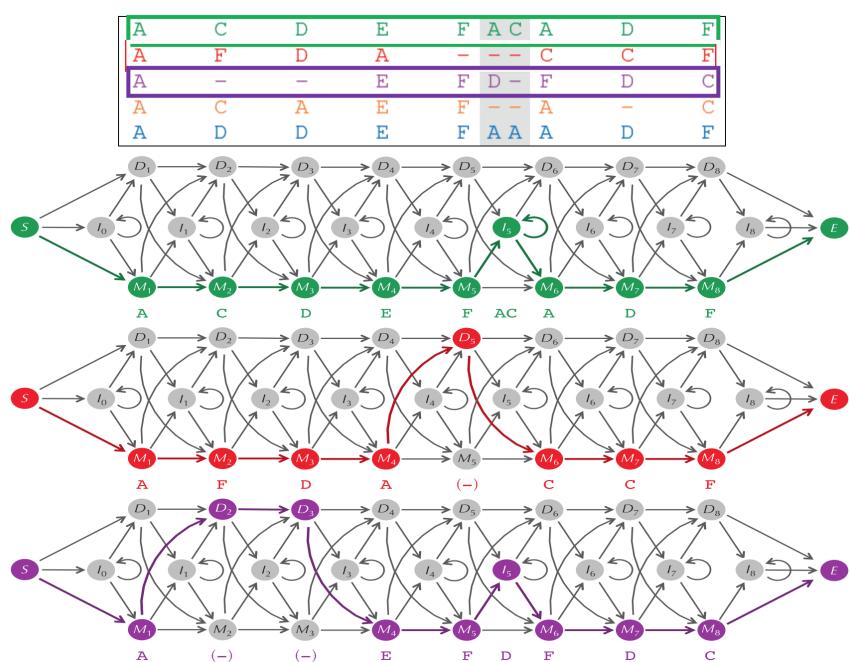
Slika 8.10: Dodavanje prelaza od insercionih stanja do delecionih stanja i obrnuto upotpunjava profilni HMM dijagram za profilnu matricu na slici 8.5. Početno i završno stanje su označeno kao S i E, redom.

Problem 14 (Problem profilnog HMM-a). *Konstruisati profilni HMM na*

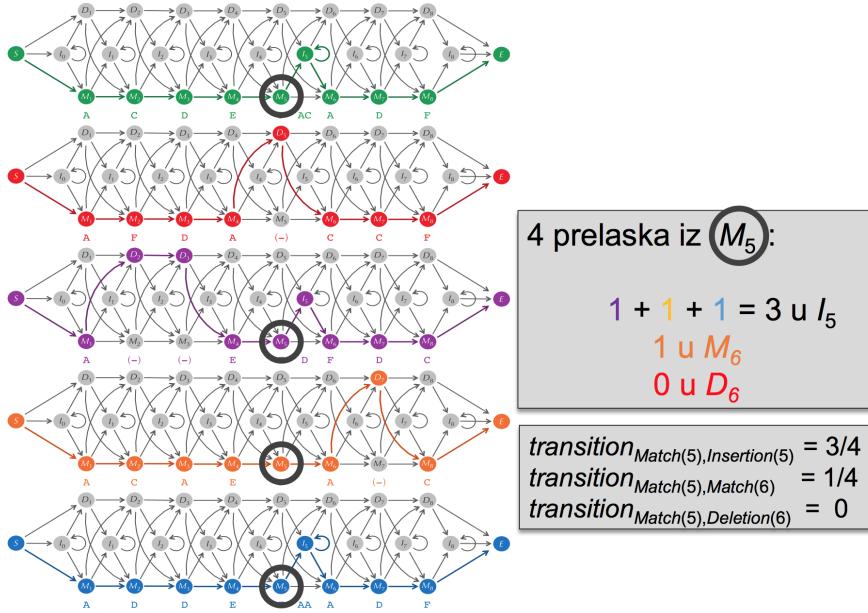
osnovu višestrukog poravnjanja.

Uzorak: Višestruko poravnanje Alignment i parametar θ (maksimalni udeo insercija po koloni).

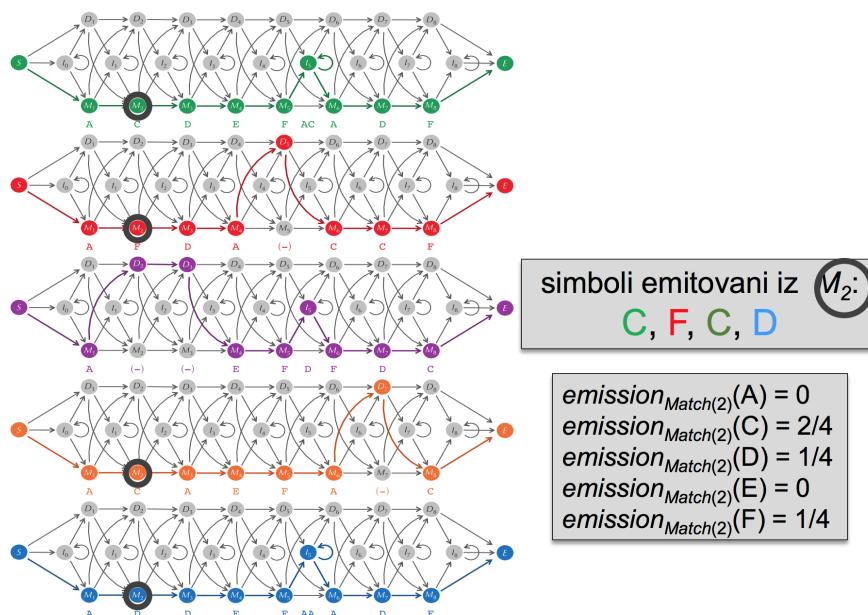
Izlaz: Emisiona i tranziciona matrica profilnog HMM $HMM(Alignment, \theta)$.



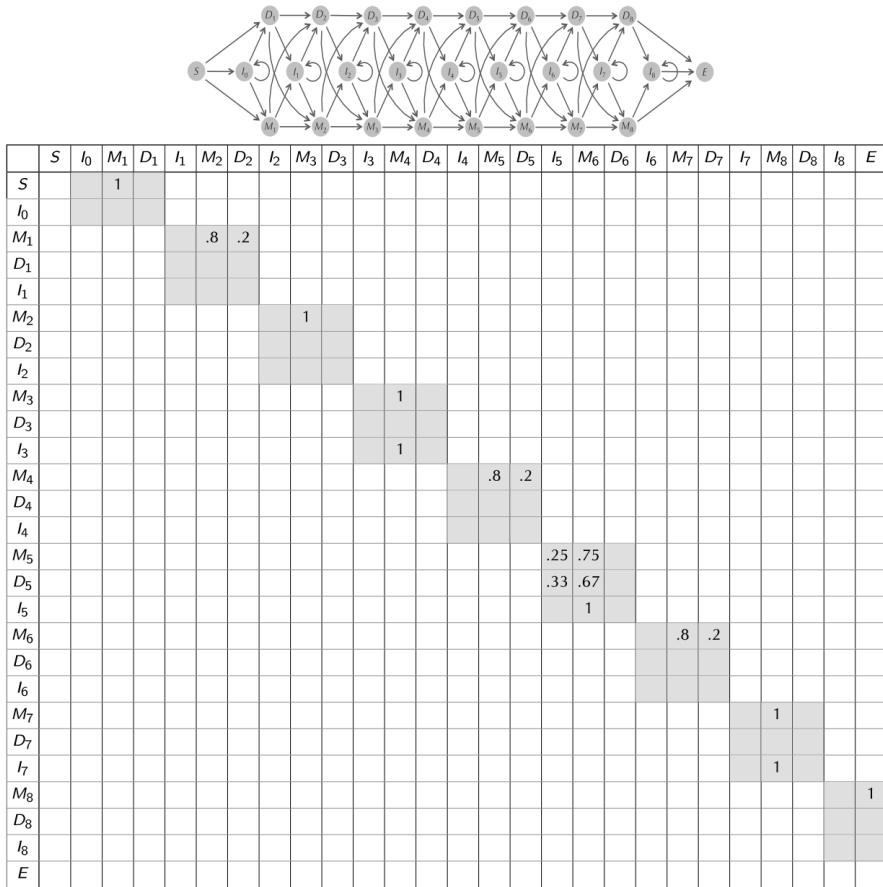
Slika 8.11: Tri putanje kroz profilni HMM koje odgovaraju trim redovima i poravnjanju na slici 8.5. Prazni simboli “-“ ispod HMM dijagrama koji odgovaraju delecionim stanjima su prikazani u zagradama da naznače da se ne emituju od strane HMM-a



Slika 8.12: Verovatnoće prelaska u profilnom HMM



Slika 8.13: Emisione verovatnoće u profilnom HMM



Slika 8.14: Zabranjeni prelasci. **Sive ćelije:** grane u HMM dijagramu. **Prazne ćelije:** zabranjeni prelasci.

8.8 Zadaci sa vežbi

U nastavku će biti predstavljeni zadaci sa vežbi na kursu rađeni u programskom jeziku Python.

8.8.1 HMM

```

1 def HMM(strings_pos, strings_neg):
2
3     HMM = []
4     # HMM['O']['A+'] = 0.125
5     # HMM['O']['T+'] = 0.125
6     # HMM['O']['C+'] = 0.125
7     # HMM['O']['G+'] = 0.125
8     # HMM['O']['A-'] = 0.125
9     # HMM['O']['T-'] = 0.125

```

```
10     # HMM['0']['C-'] = 0.125
11     # HMM['0']['G-'] = 0.125
12
13     for string in strings_pos:
14         for i in range(1, len(string)):
15             c_prev_state = string[i-1] + '+'
16             c_curr = string[i]
17
18             if c_prev_state not in HMM:
19                 HMM[c_prev_state] = {}
20                 HMM[c_prev_state]['state'] = 1
21                 HMM[c_prev_state]['nucleotide'] = string[i-1]
22                 HMM[c_prev_state]['transitions'] = {}
23
24             if c_curr not in HMM[c_prev_state]['transitions']:
25                 HMM[c_prev_state]['transitions'][c_curr] = 0
26
27                 HMM[c_prev_state]['transitions'][c_curr] += 1
28
29     for string in strings_neg:
30         for i in range(1, len(string)):
31             c_prev_state = string[i-1] + '-'
32             c_curr = string[i]
33
34             if c_prev_state not in HMM:
35                 HMM[c_prev_state] = {}
36                 HMM[c_prev_state]['state'] = 0
37                 HMM[c_prev_state]['nucleotide'] = string[i-1]
38                 HMM[c_prev_state]['transitions'] = {}
39
40             if c_curr not in HMM[c_prev_state]['transitions']:
41                 HMM[c_prev_state]['transitions'][c_curr] = 0
42
43                 HMM[c_prev_state]['transitions'][c_curr] += 1
44
45     for source in HMM:
46         output_sum = 0
47         for dest in HMM[source]['transitions']:
48             output_sum += HMM[source]['transitions'][dest]
49
50             for dest in HMM[source]['transitions']:
51                 HMM[source]['transitions'][dest] = (HMM[source][
52                 ↳ 'transitions'][dest] / output_sum) * 0.98
53
54     HMM[source]['0'] = 0.02
```

```
54
55
56     return HMM
57
58 def viterbi(HMM, string):
59     i = 1
60
61     matrix = [{} for i in range(len(string))]
62     path = ""
63
64     for i in range(len(string)):
65         nucleotide = string[i]
66
67         state_number = 0
68
69         if i > 0:
70             path += max_transition_state
71
72             max_transition_prob = -1
73             max_transition_state = ''
74
75         for state in HMM:
76
77             if HMM[state]['nucleotide'] != nucleotide:
78                 matrix[i][state] = 0
79
80             elif i == 0:
81                 if HMM[state]['nucleotide'] == nucleotide:
82                     matrix[i][state] = 0.125
83
84             else:
85                 max_prev = -1
86                 max_prev_state = ''
87
88                 for prev_state in matrix[i-1]:
89                     if HMM[prev_state]['state'] == HMM[state][
90                         'state']:
91                         state_change_prob = 0.99
92                     else:
93                         state_change_prob = 0.01
94
95                     if HMM[state]['nucleotide'] in HMM[prev_state][
96                         'transitions']:
97                         transition_prob = HMM[prev_state][
98                             'transitions'][HMM[state]['nucleotide']]
```

```
96         else:
97             transition_prob = 0
98
99             curr_state = matrix[i-1][prev_state] *
100            → transition_prob * state_change_prob
101            # print('{} -> {}'.format(prev_state, state))
102            # print('{} = {} * {} * {}'.format(curr_state,
103            → matrix[i-1][prev_state], transition_prob, state_change_prob
104            → ))
105
106             if curr_state > max_prev:
107                 max_prev = curr_state
108                 max_prev_state = prev_state
109
110             matrix[i][state] = max_prev
111
112             if max_prev > max_transition_prob:
113                 max_transition_prob = max_prev
114                 max_transition_state = max_prev_state
115
116             # print(matrix)
117             # print('-----')
118
119             max_end = -1
120             max_end_state = ''
121
122             n = len(string)
123             for state in matrix[n-1]:
124                 if matrix[n-1][state] > max_end:
125                     max_end = matrix[n-1][state]
126                     max_end_state = state
127
128             path += max_end_state
129
130
131
132
133     def main():
134         strings_pos = ['ACACAGACGCACA', 'CACATAGACAGGCATACACA', '
135         → AAATACAGTATCTTGCACTCCGGAGTCGG']
136         strings_neg = ['CGAGCGTGTGAGTGAGAGATGAG', '
137         → GTGGAACAGTAGGTAGGAGAGTG', 'AAATACAGTATCTTGCACTCCGGAGTCGG']
```

```
136
137     model = HMM(strings_pos, strings_neg)
138     print(viterbi(model, 'CTCACGAGAGGCCACAC'))
139
140 if __name__ == "__main__":
141     main()
```

