

BIOINFORMATIKA

12. maj 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 Kako složiti genomsку slagalicu od milion delova?	19
2.1 Šta je sekvenciranje genoma?	19
2.1.1 Kratka istorija sekvenciranja genoma	19
2.1.2 Sekvenciranje ličnih genoma	20
2.2 Eksplozija u štampariji	21
2.3 Problem sekvenciranja genoma	22
2.4 Rekonstrukcija niske kao problem Hamiltonove putanje	24
2.4.1 Genom kao putanja	24
2.5 Rekonstrukcija niske kao Ojlerove putanje	25
2.6 De Brojnovi grafovi na osnovu kolekcije k -grama	26
2.7 Ojlerova teorema	26
2.7.1 Dokaz Ojlerove teoreme	27
2.8 Sastavljanje parova očitavanja	28
2.8.1 DNK sekvenciranje sa parovima očitavanja	28
2.9 U realnosti	30
2.10 Zadaci sa vežbi	31
2.10.1 Maximal Non Branching Path	31
2.10.2 All Euler Cycles	34
2.10.3 String Spelled By Gapped Patterns	36
3 Kako sekvenciramo antibiotike?	39
3.1 Otkriće antibiotika	39
3.2 Kako bakterije prave antibiotike?	40
3.3 Sekvenciranje antibiotika razbijanjem na komade	43
3.3.1 Sekvenciranje ciklopeptida grubom silom	44
3.3.2 Branch-and-Bound algoritam za sekvenciranje ciklopeptida	45
3.4 Prilagođavanje sekvenciranja za spektre sa greškama	46
3.4.1 Testiranje na spektru tirocidina B1	48
3.5 Od 20 do više od 100 aminokiselina	48

3.6 Spektralna konvolucija	48
3.7 Spektri u realnosti	50
3.8 Zadaci sa vežbi	51
3.8.1 Linear Spectrum	51
3.8.2 Cyclic Spectrum	51
3.8.3 Cyclopeptide Sequencing	52
3.8.4 Leaderboard Cyclopeptide Sequencing	54

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

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.

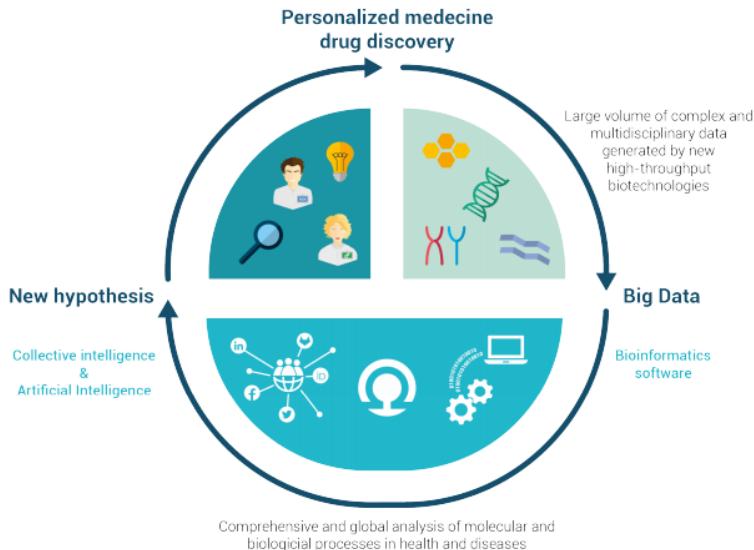
Bioinformatika ima široku primenu i njene primene rastu zajedno sa razvojem discipline. Kao što možemo videti na slici ispod, primena bioinformatike se može sagledati kroz personalizovani medicinu. Naime, na osnovu prikupljene veće količine podataka i njihove analize, uz pomoć različitih računarskih 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.

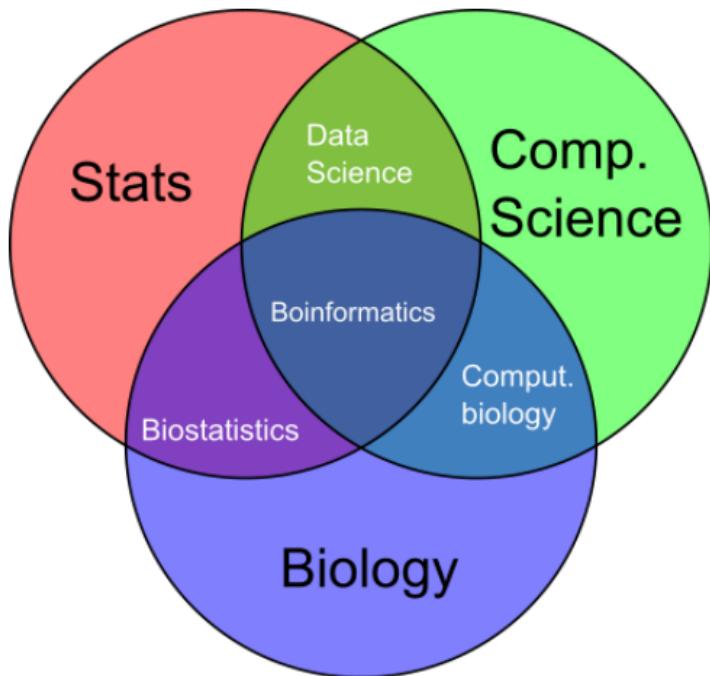
DNK se sastoji iz dva polimerna lanca koji imaju antiparalelnu orientaciju, i svaki od njih je sastavljen od azotnih baza:

- 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

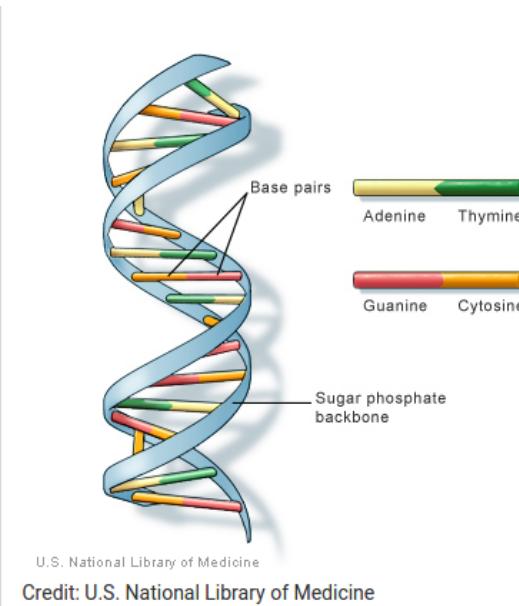
Replikacija genoma je jedan od najvažnijih zadataka ćelije. Pre nego što se podeli, ćelija mora da najpre replicira svoj genom, tako da svaka od čerki ćelija dobije svoju kopiju.

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 prodrže 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.

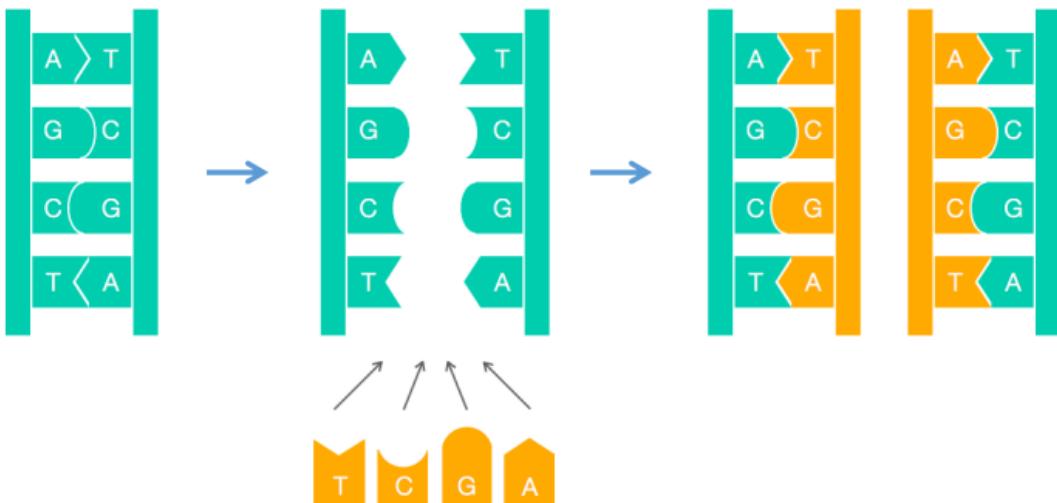
Poznato je da DNKA utiče na početak replikacije. *DNKA* je protein koji se vezuje na kratki segment unutar *oriC*, poznatiji kao **DNKA boks**. Ona predstavlja poruku unutar sekvence DNK koja govori proteinu DNKA da se veže baš tu. Postavlja se pitanje kao pronaći taj region bez prethodnog poznavanja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

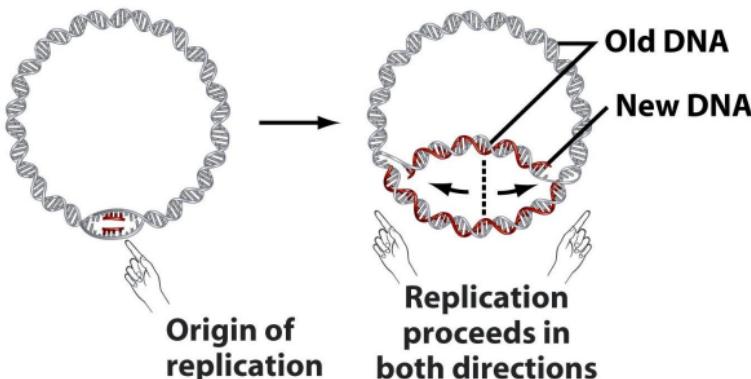
```
53++!305))6*;4826)4+.)4+);806*;48!8'6 0))85;]8*:+*8!83(88)5*!;46(;88*96*?;8
)*+(;485);5*!2:+*(;4956*2(5*4)8'8*;40 69285);)6!8)4++;1(+9;48081;8:8+1;48!8 5;4)
485!528806*81(+9;48;(88;4(+?34;48 )4+;161;:188;+?;
```

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*

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

atcaatgatcaacgtaaagcttctaagc **ATGATCAAG** gtgctcacacagtttatccacaacctgagtgg
atgacatcaagataggctgttatctccttcgtactctcatgaccacggaaag **ATGATCAAG**
agaggatgatttcttgccatatcgcaatgaatacttgtgacttgtgcttccaattgacatcttcgc
ccatattgcgtggccaagggtgacggagcgggatatacgaaagcatgatcatggctgttgttctgttt
atcttggtttactgagacttgttaggatagacggttttcatcaactgactagccaaagccttactct
gcctgacatcgaccgtaaattgataatgaattacatgctccgcacgattacct **CTTGATCAT** cg
atccgattgaagatcttcaattgttaattcttgcctcgactcatagccatgatgagct **CTTGATCA**
Tgtttcctaacccttattttacgaaag **ATGATCAAG** ctgctgct **CTTGATCAT** cgtttc

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.

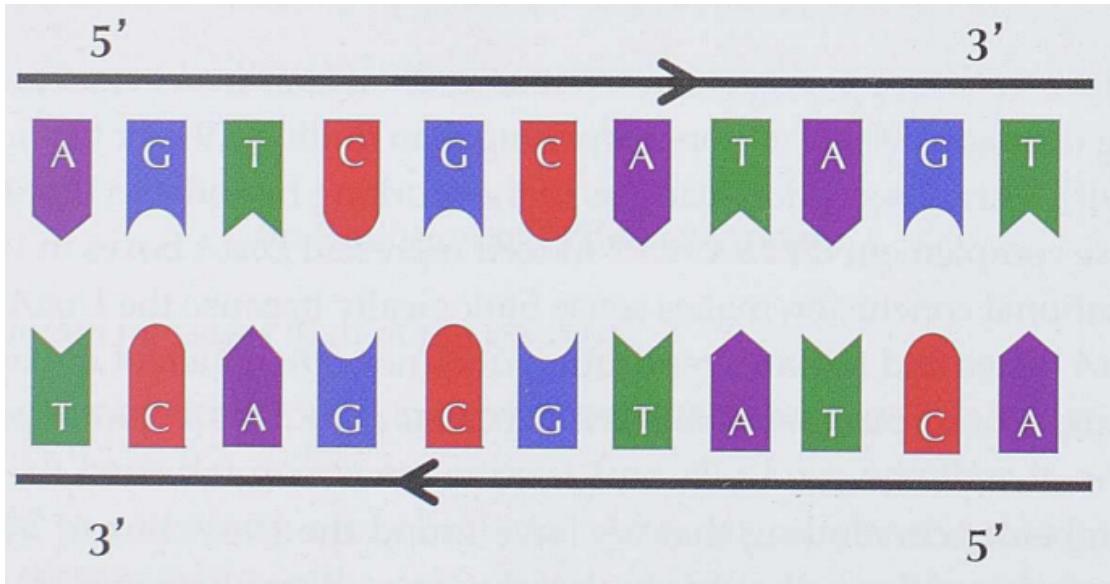
Posmatrajmo ponovo sliku 1.7. Na njoj možemo uočiti 6 pojavljivanja niski ATGATCAAG i CTTGATCAT, koji su zapravo komplementarni. Naći 9-gram koji se pojavljuje 6 puta u DNK nisci dužine 500 nukleotida, je još više iznenadjujuće, nego pronaći 9-gram koji se pojavljuje tri puta. Ovo posmatranje nas dovodi 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
tattgtattaaattgacgaacaattgcatttgcattttatgcacaaaacaaaCCTACCACCaaac
tctgtattgaccattttaggacaacttcagGGTGGTAGGtttctgaagctctcatcaatagactat
tttagtctttacaacaatattaccgttcagattcaagattctacaacgcgtttatggcggttgcagaaaaacttaccacctaattccagtatccaagccgatttcagagaaaacctaccactacactac
cacttaCCTACCACCcggtggtaagttgcagacattataaaacctcatcagaagcttggtaaaatttcaataactcgaaaCCTACCACCtgcgccccctattttactactaataatagcgttaatttgcgatctgaaaagagggtggtaaaaaaa
```

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?

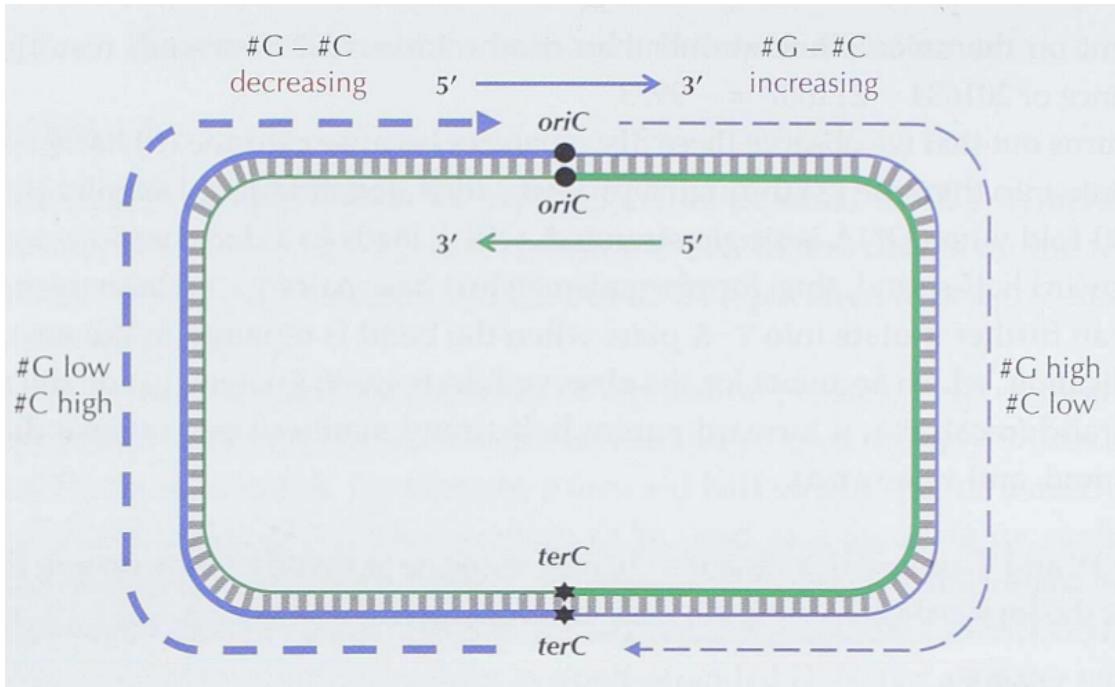
Iskrivljeni dijagrami

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.

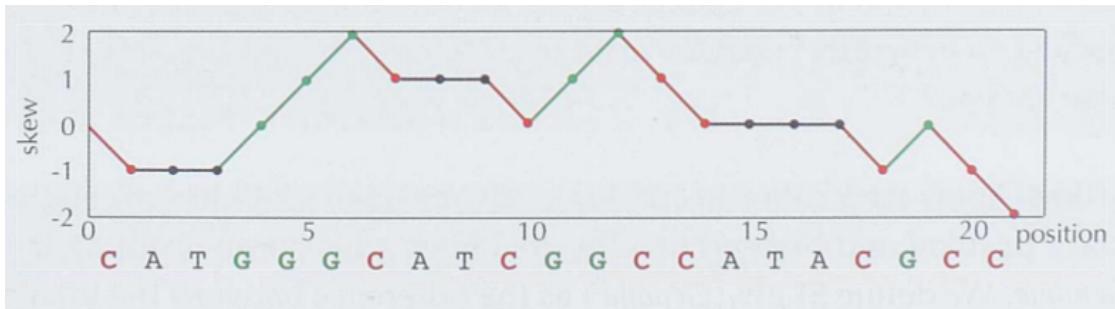
Posmatrajmo iskrivljeni dijagram bakterije Ešerihija Koli. Lako uočavamo minimalnu vrednost skew dijagrama.

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.

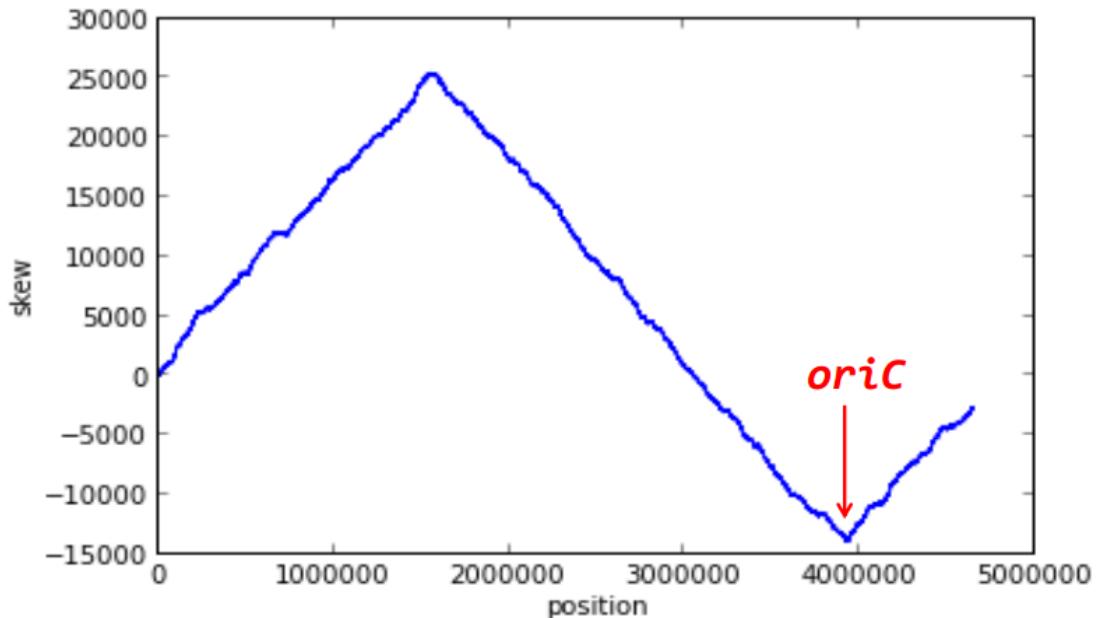
```

aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgataacgcggta
tggaaatggattgaagccggccgtggattctactcaactttgtcggcttgagaaagacc
ttggatcctgggtattaaaaagaagatctattatttagagatctgttctattgtatctc
ttattaggatcgcactgccctgtggataacaaggatccggcttttaagatcaacaacctgg
aaaggatcattaactgtgaatgatcggtatcctggaccgtataagctggatcagaatga
gggttatacacaactcaaaaactgaacaacagttttttggataactaccgggtatc
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 ATGATCAAC 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.

```
atcaATGATCAACgtaagcttctaagcATGATCAAGgtgctcacacagtttatccacaac
ctgagtggatgacatcaagataggtagtcgttatctccttcctcgtaactctcatgacca
cgaaaaagATGATCAAGagaggatgatttcttgccatatcgcaatgaataacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgcgtggccaagggtgacggagcgggatt
acgaaaagCATGATCATggctgttgttctgttatctgttttgactgagacttgttagga
tagacgggttttcatcactgactagccaaagccttactctgcctgacatcgaccgtaaat
tgataatgaatttacatgcttccgcgacgatttacctCTTGATCATcgtatccgattgaag
atcttcaattgttaattctcttgccctcgactcatagccatgatgagctCTTGATCATgtt
tccttaaccctctattttacggaagaATGATCAAGctgctgctCTTGATCATcgtttc
```

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  
tataaaaatggattgaagcccggccgtggattctactcaacttgcggcttgagaaa  
gacctggatcctgggtattaaaaagaagatctatttttagagatctgttctattgt  
gatctcttatttaggatcgcactgccTGTGGATAAcaaggatccggctttaaagatcaa  
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

```


Glava 2

Kako složiti genomsку slagalicu od milion delova?

Kao i do sada, iz naslova ovog poglavlja nam verovatno nije jasno o kakvom problemu je reč. U ovom poglavlju predstavićemo dati problem i prikazati kako možemo primeniti grafovske algoritme nad problemom slaganja genomske slagalice. Poglavlje ćemo započeti pričom o sekvenciraju genoma. Do sada smo videli šta za nas znači pojam DNK – da se podsetimo, to je niska karaktera nad abzukom $\Sigma = \{A, C, G, T\}$. Biološki posmatrano, DNK je molekul koji se nalazi u svakoj ćeliji svakog organizma i da je u njemu zapisan način pravilnog razvoja i funkcionalisanja svakog organizma.

2.1 Šta je sekvenciranje genoma?

Sa biološke strane, genom jednog organizma predstavlja njegov genetski materijal. Pod genetskim materijalom smatramo materijal koji se nasleđuje i koji svi 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.

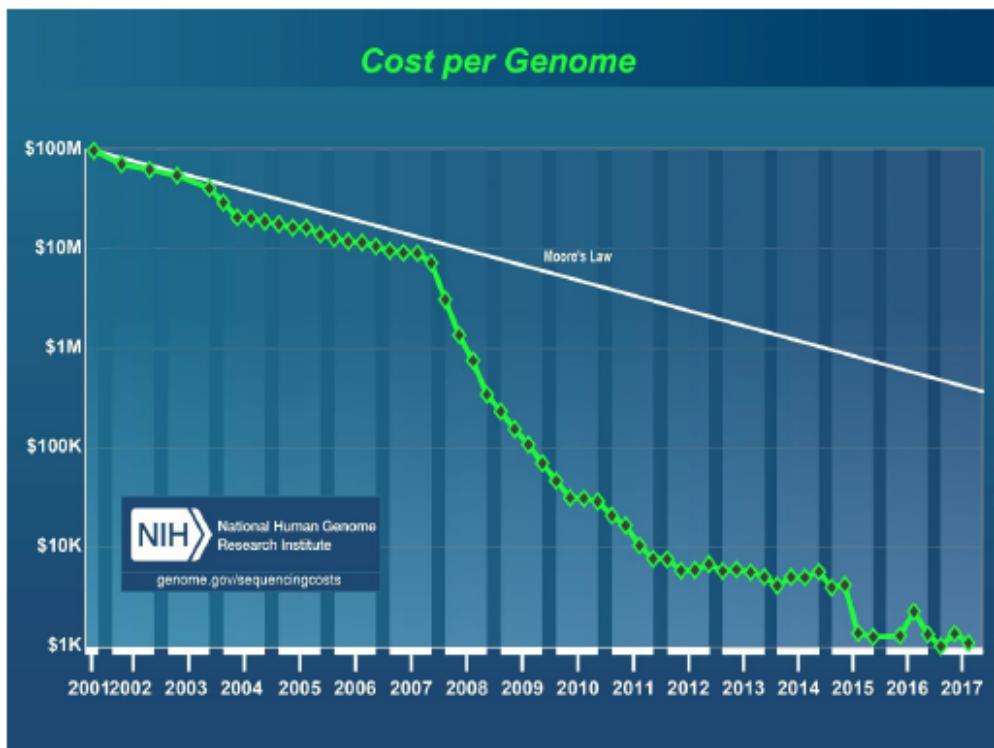
Kod čoveka, genom sadrži oko tri milijarde nukleotida. Dakle, sa računarske strane posmatrano genom je niska karaktera nad abzukom $\Sigma = \{A, C, G, T\}$. Kompleksnost organizma nije u relaciji sa veličinom genoma. Genomi nekih organizama su i stotinu puta veći od humanog genoma. Na primer, jedna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

Sekvenciranje genoma podrazumeva otkrivanje izgleda genoma. U pitanju je eksperimentalan proces – da bismo saznali šta se nalazi u sastavu jednog genoma, potreban nam je uzorak tkiva odgovarajuće vrste. U nastavku dajemo kratak pregled razvoja sekvenciranja genoma.

2.1.1 Kratka istorija sekvenciranja genoma

Kao što smo videli, sekvenciranje genoma je eksperimentalan proces, za koji je neophodna veoma ozbiljna 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 sekvenciranjem genoma, za koje su, 1980. godine, podelili Nobelovu nagradu. Međutim, iako su njihovi metodi bili pionirski u ovoj oblasti, njihove metode za sekvenciranje su bile veoma skupe – za sekvenciranje 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. *Net 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 sekvenciranje genoma. Na slici 2.1 prikazano je kako se cena sekvencioniranja menjala godinama.



Slika 2.1: Cena sekvencioniranja kroz istoriju.

2.1.2 Sekvenciranje ličnih genoma

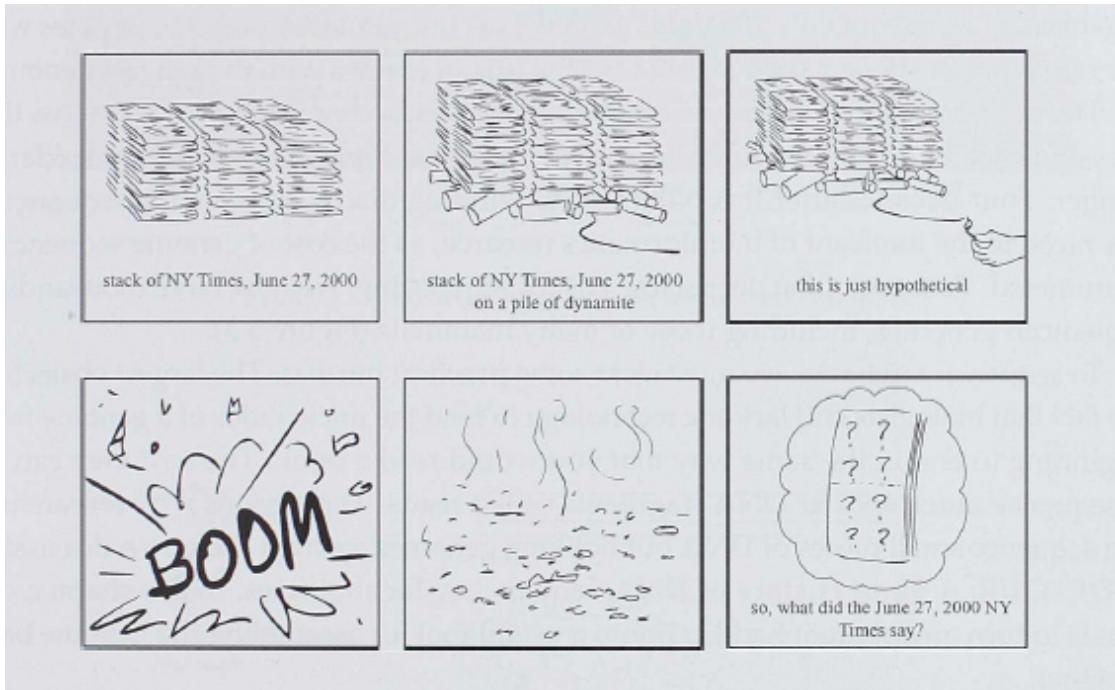
Prirodno je zapatiti se o značaju poznavanja sastava genoma. Što je tiče biljaka, neke od primena sekvenciranja su: razvoj novih biljnih vrsta u poljoprivredi, određivanje pogodnog podneblja za neku biljnu vrstu, u farmaciji, i dr. Međutim, najznačajnija primena je u sekvenciranju ličnih genoma.

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.

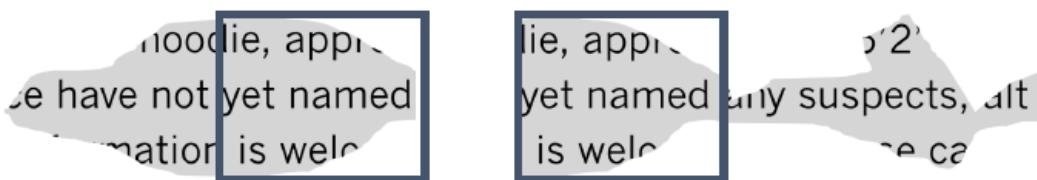
2.2 Eksplozija u štampariji

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 2.2).



Slika 2.2: 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.



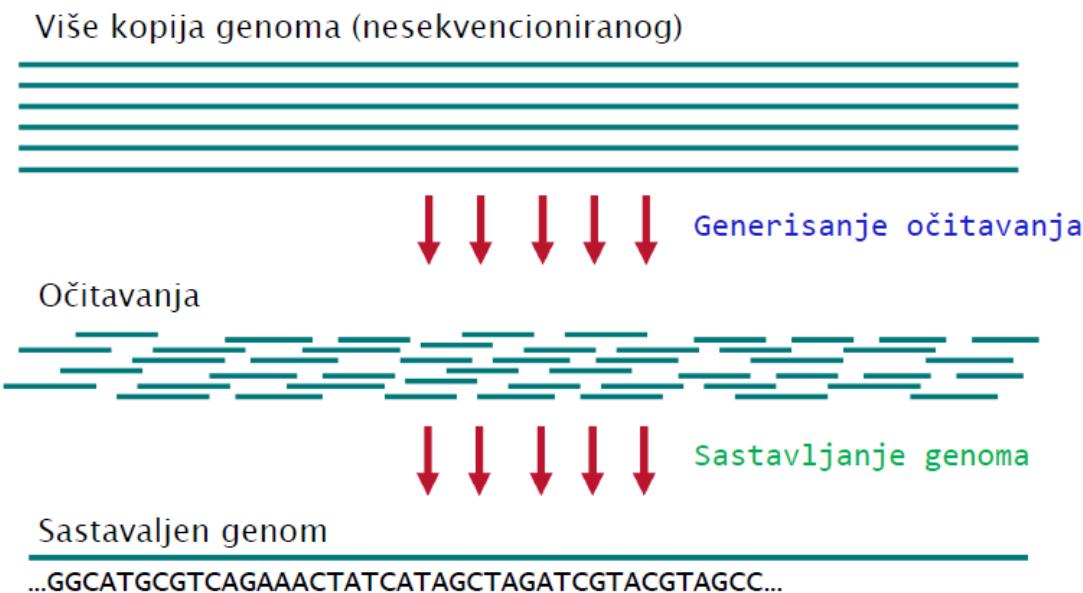
Slika 2.3: Spajanje delova različitih novina koji se jednim delom preklapaju.

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 2.4 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 2.4: Ilustracija problema.

2.3 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 1 (Problem sekvencioniranja genoma). *Rekonstruisati genom na osnovu očitavanja.*
Ulaz: 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-gramsni sastav niske* na sledeći način. *k*-gramsni sastav niske *Text*, u oznaci *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-gramsni 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 2 (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 iskorisćeni 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.

2.4 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.

2.4.1 Genom kao putanja

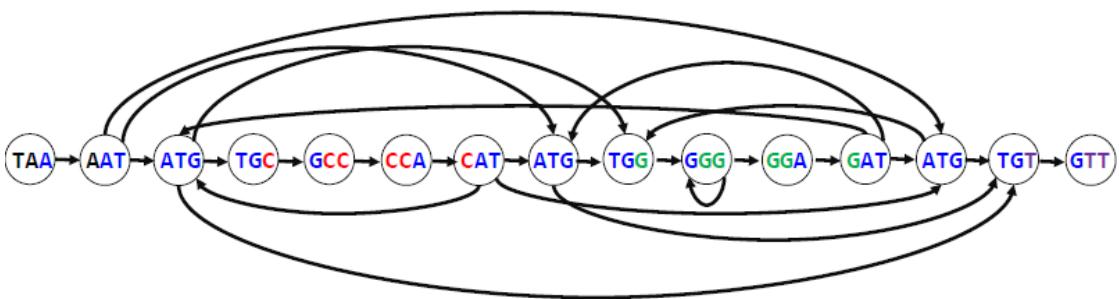
Vratimo se na prethodni primer. Dat nam je naredni 3-gramske sastav:

```

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-gramske 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 prefiksu ulaznog čvora te grane, kao što je prikazano na slici 2.5.



Slika 2.5: 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 3 (Problem Hamiltonove putanje). *Naći Hamiltonovu putanju u grafu.*

Ulaz: Graf.

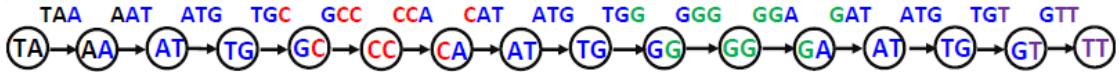
Izlaz: Putanja koja posećuje svaki čvor u grafu tačno jednom.

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 efikasan algoritam koji to radi. U tom slučaju, moramo da se vratimo na početak, a to je predstavljanje k -gramskog sastava grafiom.

2.5 Rekonstrukcija niske kao Ojlerove putanje

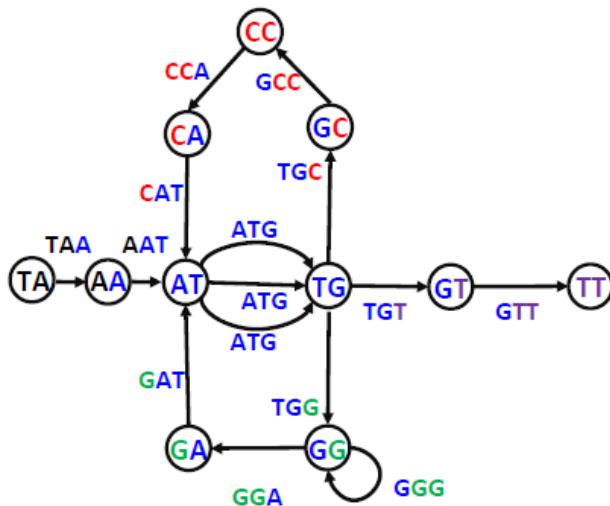
U prethodnoj sekciji, k -grame smo predstavili čvorovima u grafu i u njemu tražili Hamiltonov put, odnosno, put koji obilazi svaki čvor tačno jednom. Videli smo da za taj problem još uvek nije poznat efikasan algoritam pa se sada pitamo kako možemo izmeniti graf tako da ne zahteva traženje Hamiltonove 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 2.6 ilustruje ovaj postupak za nisku *TAATGCCATGGGATGTT*.



Slika 2.6: Grafički prikaz 3-gramske kompozicije niske *TAATGCCATGGGATGTT*. Čvorovi su označeni 2-gramima (prefiksima i sufiksimi), a grane su obeležene 3-gramima.

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 2.7 ilustruje De Brojnov graf dobijen ovom procedurom od polaznog grafa sa slike 2.6.



Slika 2.7: De Brojnov graf koji odgovara niski *TAATGCCATGGGATGTT*.

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 4 (Problem Ojlerove putanje). *Pronaći Ojlerovu putanju u grafu.*

Ulaz: Graf.

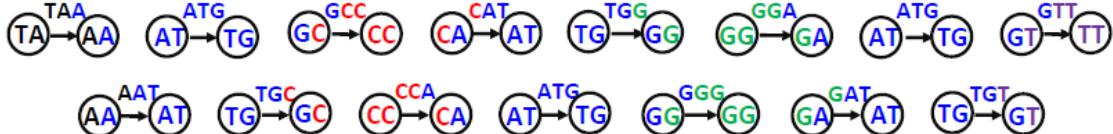
Izlaz: Putanja koja posećuje svaku granu u grafu tačno jednom.

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

2.6 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 -gramsks sastav. Postavlja se pitanje kako možemo konstruisati De Brojnov graf od k -gramsksog 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 2.8. Zatim lepimo identične čvorove sve dok ne dobijemo graf čiji svi čvorovi imaju različite oznake. Na slici 2.9 dat je jedan korak ovog postupka, međutim, tu nije kraj jer i dalje postoje čvorovi sa istim oznakama.



Slika 2.8: Svaki k -gram prestavljen je pomoću dva čvora i jedne grane.



Slika 2.9: Prvi korak u postupku lepljenja čvorova. Napomenimo da postupak nije završen.

Po završetku postupka dobijamo de Brojnov graf koji je isti kao onaj koji smo dobili kada smo znali nisku, odnosno, graf na slici 2.7. 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 zapepljeni.

2.7 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 5 (Problem Ojlerovog ciklusa). *Pronaći ciklus u Ojlerovom 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 mrežu 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.

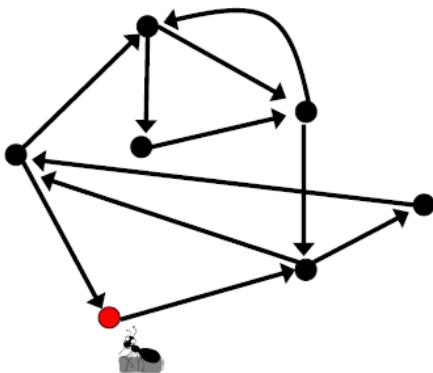
Teorema 2.1 (Ojlerova teorema). *Svaki Ojlerov graf je balansiran. Svaki povezan graf i balansiran graf je Ojlerov.*

2.7.1 Dokaz Ojlerove teoreme

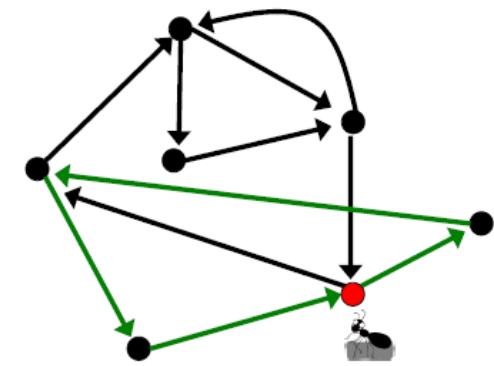
Neka nam je dat povezan balansiran graf. Da bismo pokazali da graf sadrži Ojlerov ciklus, postavićemo mrava na bilo koji od čvorova tog grafa, kao na slici 2.10. Zašto baš mrav? Poznato je da mravi nikada ne idu istim putem dva puta pa smo sigurni da će naš mrav proći svaku granu tačno jednom.

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 2.11).



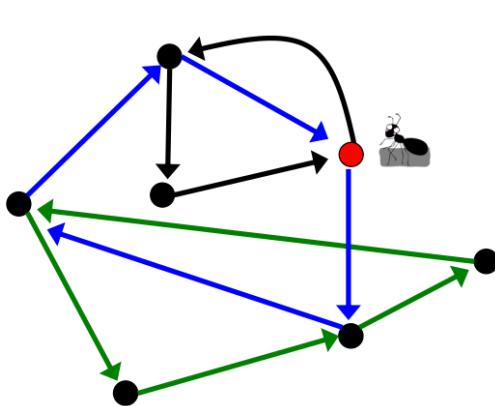
Slika 2.10: Mrav je postavljen u crveni čvor i odatle kreće obilazak povezanog balansiranog grafa.



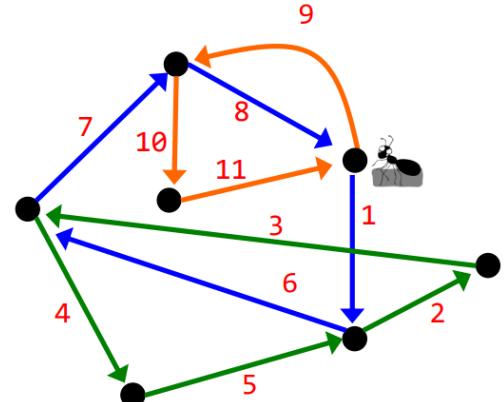
Slika 2.11: Mrav se zaglavio i pokušava ponovo iz drugog čvora koji pripada ciklusu i ima izlazne grane koje nisu posećene.

Sada, mrav 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 zaglavii (slika 2.12) ponovo bira novi početni čvor, obilazi pronađeni ciklus (koji i dalje nije Ojlerov) i tako sve dok ne uspe da obide sve grane (slika 2.13).

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 2.12: Mrav se ponovo zaglavio i pokušava od novog čvora.



Slika 2.13: 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

```

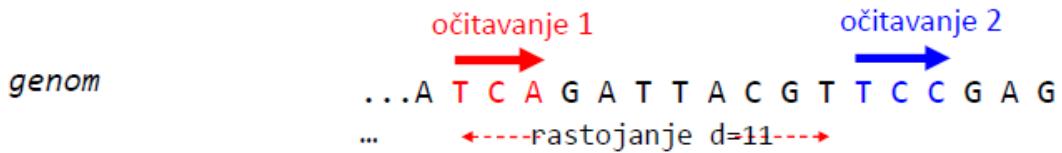
Ovaj algoritam radi u linearном vremenu. Da bi se zaista postigla ta efikasnost, potrebne su efikasne strukture podataka za održavanje ciklusa koje mrav pronalazi kao i za liste neiskorišćenih grana za svaki čvor i lista čvorova u trenutnom ciklusu koji imaju neiskorišćene grane.

2.8 Sastavljanje parova očitavanja

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

2.8.1 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 2.14: 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 sastave date niske prikazan je na slici 2.15.

TAA	AAT	ATG	TGC	GCC	CCA	CAT	ATG	TGG	GGG	GGA
GCC	CCA	CAT	ATG	TGG	GGG	GGA	GAT	ATG	TGT	GTT
AAT	ATG	ATG	CAT	CCA	GCC	GGA	GGG	TAA	TGC	TGG
CCA	CAT	ATG	GAT	GGA	TGG	GTT	TGT	GCC	ATG	ATG

Slika 2.15: *PairedComposition* niske TAATGCCATGGGATGTT i njegov leksikografski poredak.

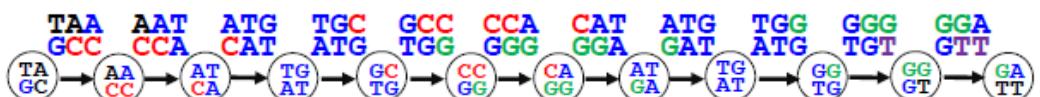
Sada možemo formulisati naredni problem.

Problem 6 (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.

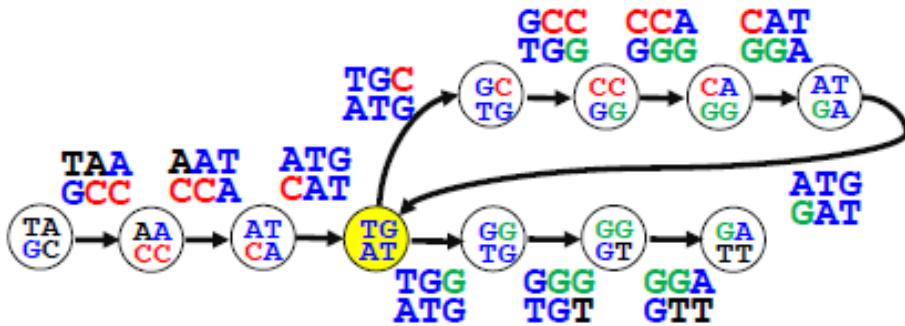
Kako konstruisati upareni De Brojnov graf na osnovu uparenog k -gramske sastave? Postupak je sličan prethodnom slučaju, kada nismo imali parove. Pretpostavimo da je dat genom (niska *Genome*). Posmatrajmo genom kao putanju u grafu obeleženom na osnovu njegovog uparenog k -gramske sastava (videti sliku 2.16). Svaka grana obeležena je uparenim k -gramom, a svaki čvor uparenim prefiksom, odnosno, sufiksom k -grama.



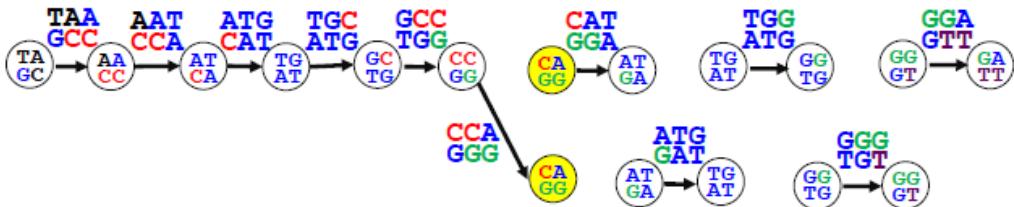
Slika 2.16: Graf koji odgovara uparenom 3-gramscom sastavu niske TAATGCCATGGGATGTT.

Potrebno 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 2.17).

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 -gramske 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 2.18), i na kraju dobijamo upareni De Brojnov graf, kao onaj na slici 2.17.



Slika 2.17: Dva čvora sa oznakom (TG AT) spajaju se u jedan pri čemu su sve grane, incidentne sa tim čvorovima, očuvane.



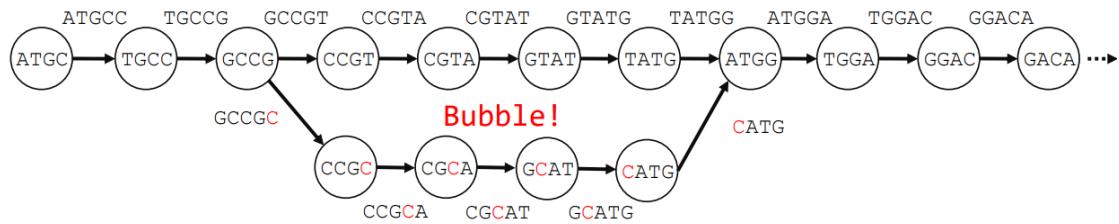
Slika 2.18: Konstrukcija uparenog De Brojnovog grafa na osnovu uparenih k -grama.

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.

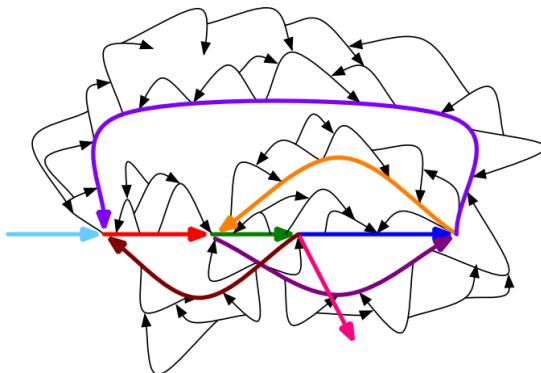
2.9 U realnosti

Ovde smo imali neke nerealne prepostavke:

- 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.
- 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 2.19). Jednostavan je slučaj kada govorimo o grešci na jednom očitavanju, međutim, ukoliko postoji više grešaka, onda dolazi do *eksplozije balončića* (videti sliku 2.20).
- Rastojanja između očitavanja u okviru parova očitavanja su egzaktna.
- itd.



Slika 2.19: Primer pojave balončića u De Brojnovom grafu usled pojave greške u očitavanju nukleotida T nukleotidom C.



Slika 2.20: Eksplozija balončića.

2.10 Zadaci sa vežbi

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

2.10.1 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
88                     visited[w] = True
89                     (w_in_deg, w_out_deg) = degree(G, w)
90
91             paths.append(non_branching_path)
92
93     for v in G:
94         if v not in visited:
95             c = isolated_cycle(G, v)
96             if c != None:
97                 paths.append(c)
98
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     #
115     #
116     #
117     #
118     #
119     #
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()
```

2.10.2 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(">", "A")
36
37     for i in range(len(path)):
38         dna_string += path[i][1].replace(">", "A")[-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          'AT' : ['CG'], 'GG': ['GA']}
127     print(all_eulerian_cycles(G))
128
129 if __name__ == "__main__":
130     main()

```

2.10.3 String Spelled By Gapped Patterns

```

1 # Sastavljanje DNK niske pomoci 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 pomoci 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 3

Kako sekvenciramo antibiotike?

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

3.1 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 [3.2](#).

3.2 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 azbuci 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 azbukom 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 azbukom 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 3.1. *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 3.1.

Definicija 3.2. *Genetski kod predstavlja preslikavanje skupa kodona u skup aminokiselina.*

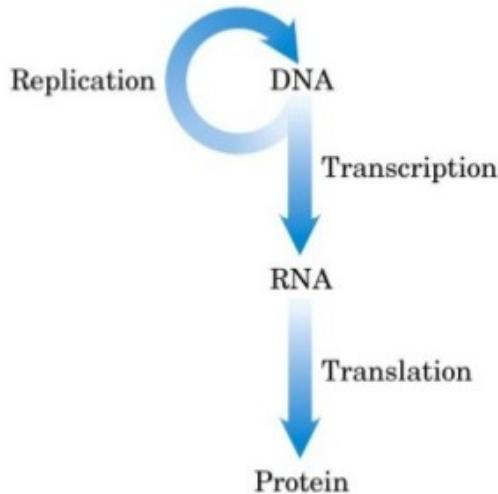
Vidimo da se, na primer, kodon UGC preslikava u aminokiselinu $Trp(W)$, dok se više kodona, $CUA, CUC, CUG, CUU, UUA, UUG$, preslikava u jednu aminokiselinu $Lys(L)$.

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 3.1: 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 3.2.



Slika 3.2: 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 3.3 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 3.3: 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.

3.3 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 3.1.

Tabela 3.1: 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 3.1.

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

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 3.3. *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 7 (Problem sekvenciranja ciklopeptida). *Rekonstruisati ciklični peptid na osnovu njegovog teorijskog spektra.*

3.3.1 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.

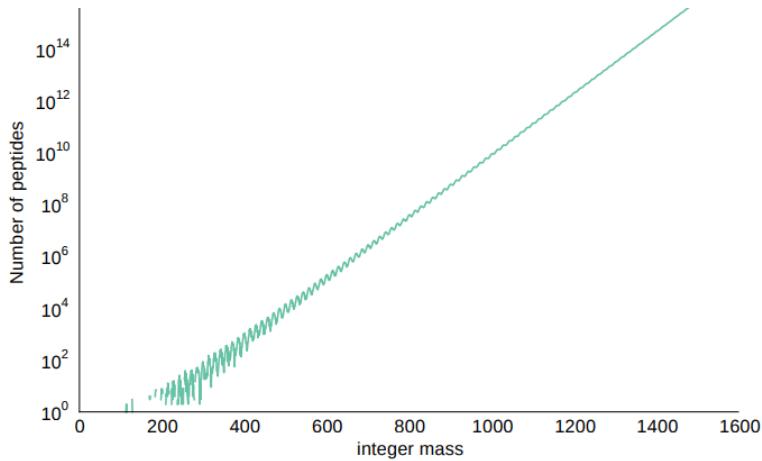
```

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 3.4.

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 3.4: Broj peptida sa zadatom masom.

3.3.2 *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 3.5. *Bound grana* bi od postojećih kandidata, nastalih u *branch* granama, isključila neke potencijalne kandidate. *Bound grana* je predstavljena na slici 3.6. 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 3.5: *Branch* grane algoritma *Branch-and-Bound*.Slika 3.6: *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 unigramma *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 kasnije videti da ovako zadati spektar jeste čisto teorijski spektar, a maseni spektrometar skoro nikada u praksi ne vraća teorijski spektar već spektar sa nekakvим grešкама. 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 3.4. 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.

3.4 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 kakvим greškama se govori biće prikazano pomoću slike 3.7.

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 3.7: Primer teorijskog i eksperimentalnog spektra za *NQEL*.

Lažne mase jesu mase koje su na slici 3.7 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 3.7 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 3.5. *Skor peptida pokazuje koliko masa njegov teorijski spektar deli sa eksperimentalnim spektrom.*

Tako, za sliku 3.7, 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         return Leaderboard
36

```

```

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.

3.4.1 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.

3.5 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 azbukom 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.

3.6 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 3.6. *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 3.8.

	""	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 3.8: 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 3.8 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

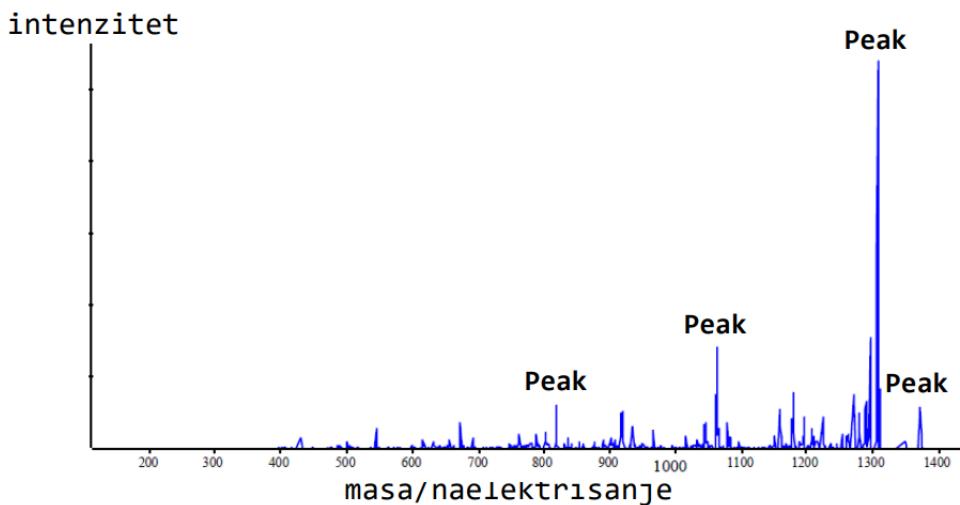
```

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.

3.7 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 3.9. Na osnovu vrhova na grafiku, određivaćemo sam sastav peptida. Ovaj grafik se



Slika 3.9: Primer grafika intenziteta prema odnosu mase i naelektrisanja.

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

3.8 Zadaci sa vežbi

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

3.8.1 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()
```

3.8.2 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()

```

3.8.3 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] -
20             ↪ prefix_mass[i]))
21
22     cyclic_spectrum.sort()
23     return cyclic_spectrum
24
25 # Prosirivanje liste peptida dodavanjem svih mogucih amino kiselina na kraj
26 # lanca
27 def expand(peptides, amino_acid):
28     extension = []
29
30     for peptide in peptides:
31         for aa in amino_acid:
32             extension.append(peptide + aa)
33
34     return extension
35
36 # Izracunavanje ukupne mase peptida kao sume svih aminokiselina u lancu
37 def mass(peptide, amino_acid, amino_acid_mass):
38     total_mass = 0
39
40     for i in range(len(peptide)):
41         for j in range(len(amino_acid)):
42             if peptide[i] == amino_acid[j]:
43                 total_mass += amino_acid_mass[j]
44
45     return total_mass
46
47 # Izdvajanje sume celog peptida iz spektra
48 def parent_mass(spectrum):
49     return spectrum[-1]
50
51 # Formiranje linearog spektra
52 def linear_spectrum(peptide, amino_acid, amino_acid_mass):
53     prefix_mass = [0]
54     current_mass = 0
55
56     for i in range(len(peptide)):
57         for j in range(20):
58             if amino_acid[j] == peptide[i]:
59                 prefix_mass.append(current_mass + amino_acid_mass[j])
60                 current_mass += amino_acid_mass[j]
61
62     linear_spectrum = [0]
63     for i in range(len(prefix_mass)):
64         for j in range(i+1, len(prefix_mass)):
65             linear_spectrum.append(prefix_mass[j] - prefix_mass[i])
66
67     linear_spectrum.sort()
68     return linear_spectrum
69
70 # Provera da li je dati peptid konzistentan sa zadatim spektrom
71 def consistent(peptide, target_spectrum, amino_acid, amino_acid_mass):

```

```

70     peptide_linear_spectrum = linear_spectrum(peptide, amino_acid,
71         ↪ amino_acid_mass)
72     for aa in peptide_linear_spectrum:
73         found = False
74         for aa_p in target_spectrum:
75             if aa_p == aa:
76                 found = True
77         if found == False:
78             return False
79
80     return True
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 def main():
104     amino_acid = ['G', 'A', 'S', 'P', 'V', 'T', 'C', 'I', 'L', 'N', 'D', 'K',
105         ↪ 'Q', 'E', 'M', 'H', 'F', 'R', 'Y', 'W']
106     amino_acid_mass = [57, 71, 87, 97, 99, 101, 103, 113, 113, 114, 115, 128,
107         ↪ 128, 129, 131, 137, 147, 156, 163, 186]
108
109     peptide = "SPQR"
110
111     spectrum = cyclic_spectrum(peptide, amino_acid, amino_acid_mass)
112     cyclopeptide_sequencing(spectrum, amino_acid, amino_acid_mass)
113
114     if __name__ == "__main__":
115         main()

```

3.8.4 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] -
→ prefix_mass[i]))
20
21     cyclic_spectrum.sort()
22     return cyclic_spectrum
23
24 # Prosirivanje liste peptida dodavanjem svih mogucih amino kiselina na kraj
→ 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
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]]

```

```
152         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]
162
163     peptide = "SPQR"
164
165     spectrum = cyclic_spectrum(peptide, amino_acid, amino_acid_mass)
166
167     print(leaderboard_cyclopeptide_sequencing(spectrum, 10, amino_acid,
168                                               ↪ amino_acid_mass))
167
168 if __name__ == "__main__":
169     main()
```

Glava 4

Kako poredimo biološke sekvence?

4.1 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:

$$\begin{array}{c} ATGCATGC \\ TGCATGCA \end{array}$$

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

$$\begin{array}{c} A\color{red}{TGCATGC}- \\ -\color{red}{TGCATGCA} \end{array}$$

Stringovi ATGCTTA i TGCATTAA imaju manje uočljive sličnosti:

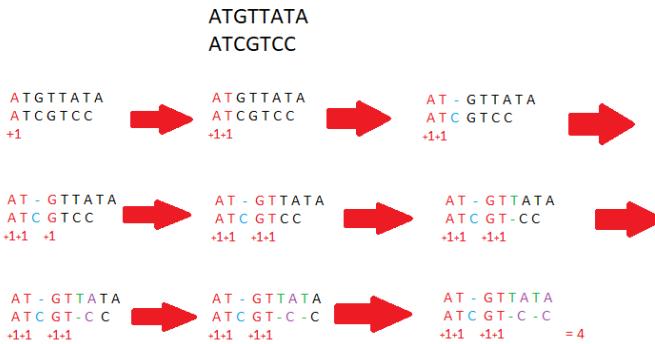
$$\begin{array}{c} A\color{red}{TGC}-\color{red}{TTA}- \\ -\color{red}{TGCATTAA} \end{array}$$

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.

4.2 Igra poravnanja i najduža zajednička podsekvenca

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

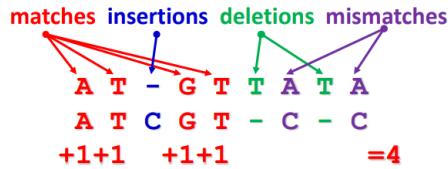
- Uklanjanje prvog simbola iz svake sekvene
- 1 poen ako se simboli poklapaju, 0 ako se simboli ne poklapaju
- Uklanjanje prvog simbola iz jedne sekvene
 - 0 poena



Slika 4.1: 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 4.2: Poravnanje

4.2.1 Najduža zajednička podsekvenca

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

Problem 8 (Problem najduže zajedničke podnizske). *Naći najdužu zajedničku podsekvenku dve niske.*

Ulaz: Dve niske.

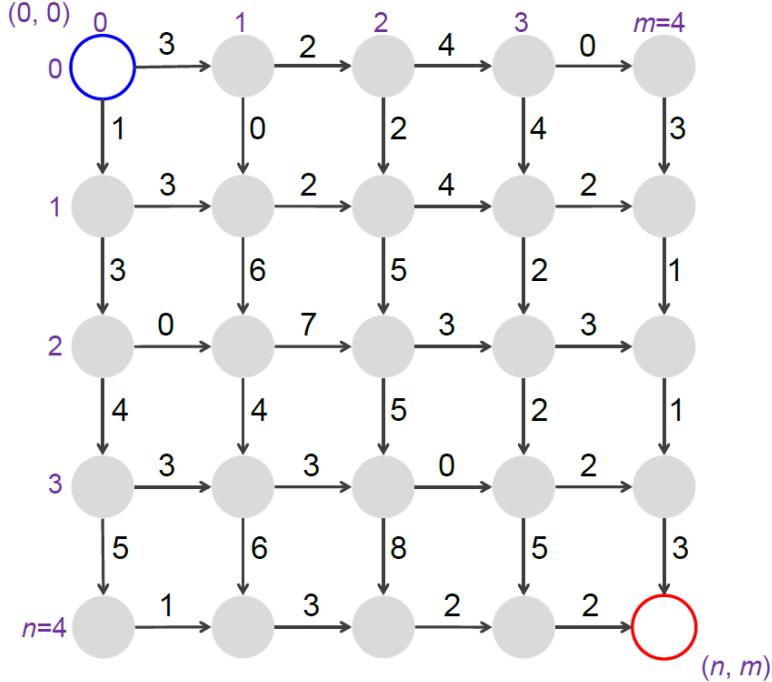
Izlaz: Najduža zajednička podsekvenca ovih niski

4.3 Problem turiste na Menhetnu

Pre svega postavimo problem:

Problem 9 (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 4.3: Problem turiste na Menhetnu

Na slici ?? 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 (??).

Time dolazimo do sledećeg problema:

Problem 10 (Problem najduže putanje u usmerenom grafu). *Naći najdužu putanju između dva čvora u težinskom usmerenom grafu.*

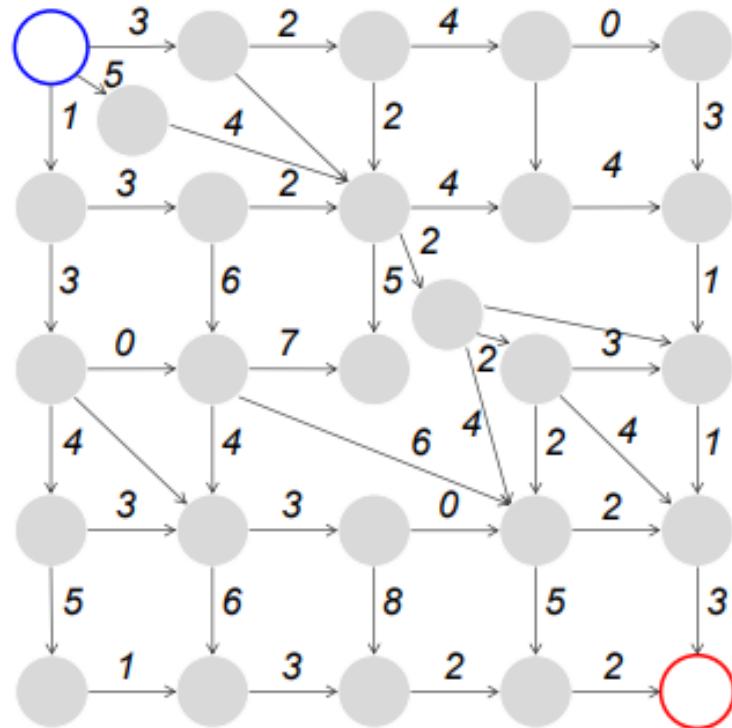
Ulaz: Usmereni težinski graf sa označenim čvorovima *source* i *sink*.

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

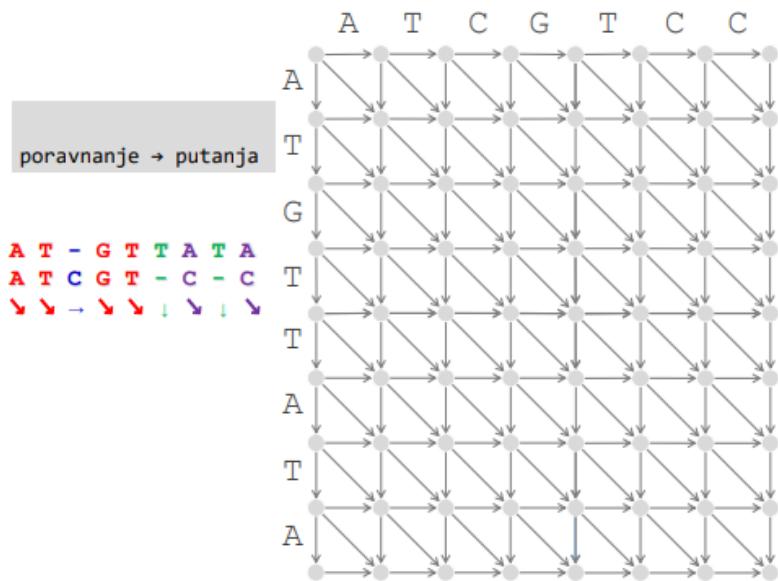
Ako se prisetimo igre poravnjanja, videćemo da postoji veza između ova dva problema (igre).

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



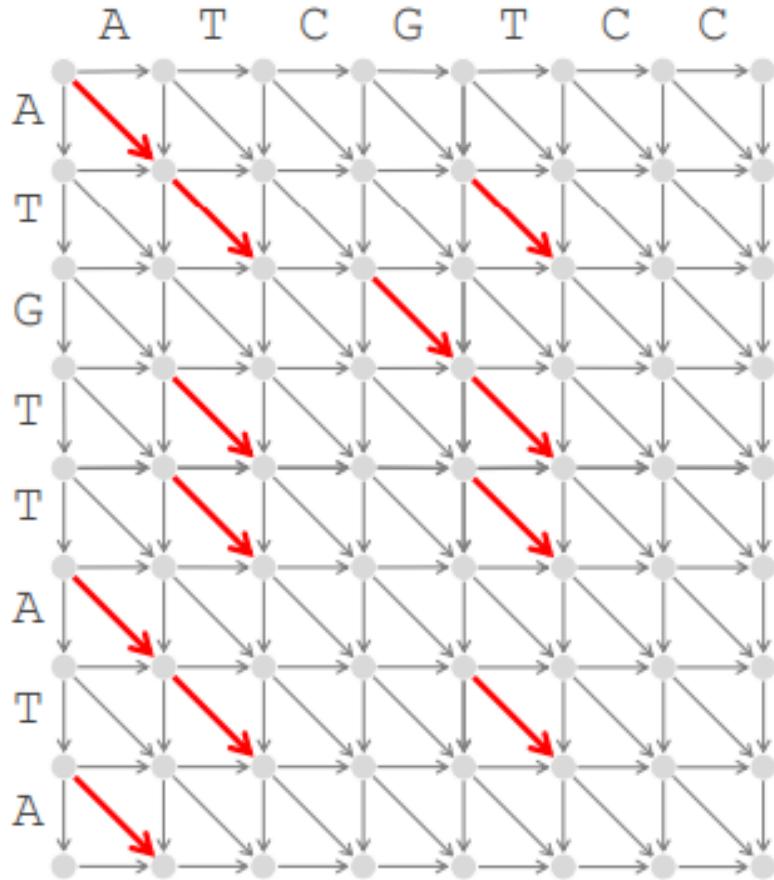
Slika 4.4: Nepravilna mreža



Slika 4.5: Poravnjanje → Putanja

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

Kada nađemo poravnjanje najvišeg skora našli smo i najdužu putanju u mrežnom grafu. Dijagonalne crvene grane odgovaraju poklapanju simbola i imaju skor 1 (??)



Slika 4.6: Poravnanje → Putanja

4.4 Problem kusura

Upoznajmo se sa sledećim problemom:

Problem 11 (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 $money$.

4.4.1 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!

4.4.2 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 4.7: 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 ?? 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

```

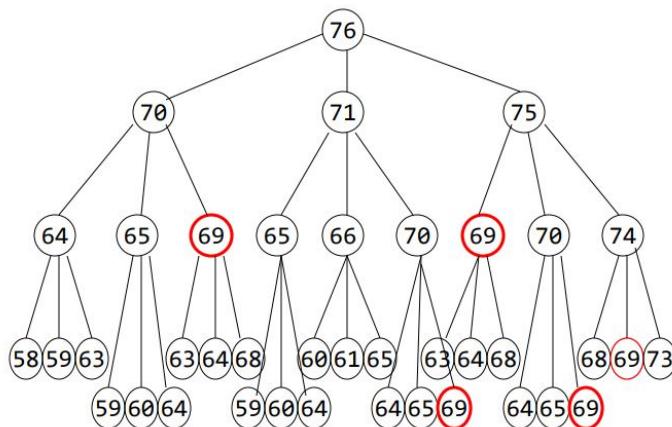
```

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 4.8: 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.

4.4.3 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

$\text{MinNumCoins}(\text{money} - \text{coin}_i)$.

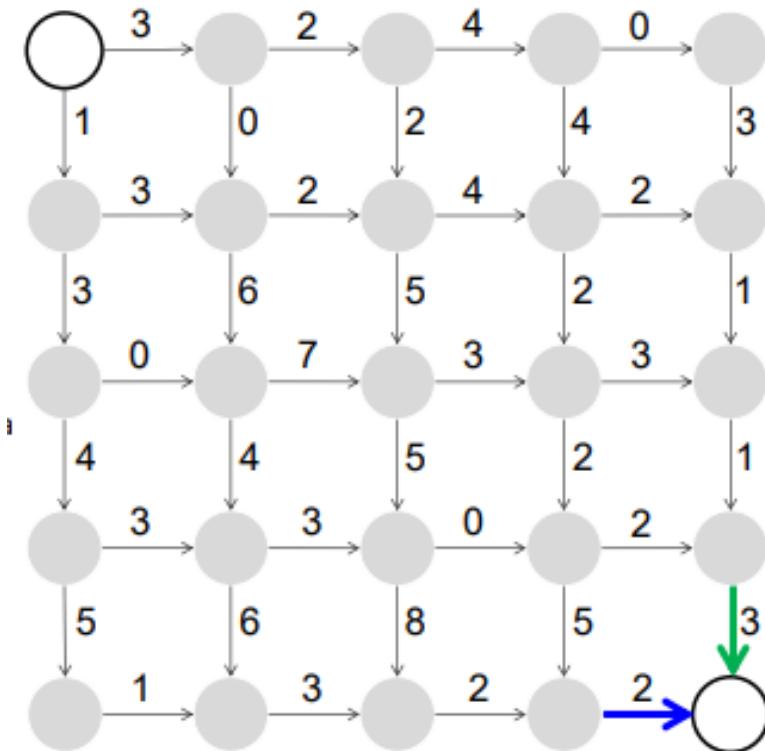
```

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

```

4.5 Dinamičko programiranje i putokazi za povratak

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



Slika 4.9: 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

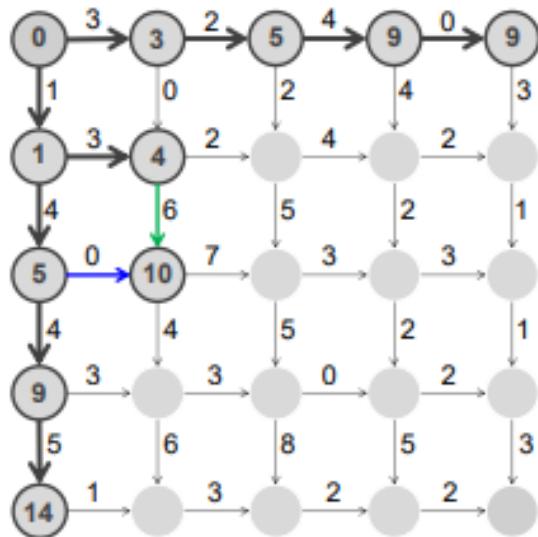
```

```

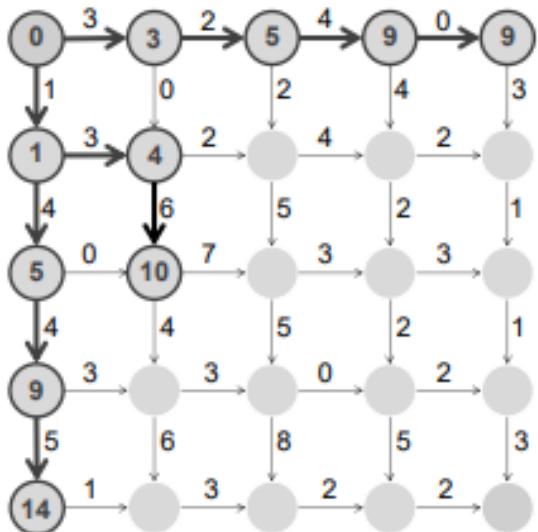
5 if n > 0
6 x ← SouthOrEast(n-1,m)+weight of edge "↓" into (n, m)
7 if m > 0
8 y ← SouthOrEast(n,m-1)+ weight of edge "→" into (n,m)
9 return max{x, y}

```

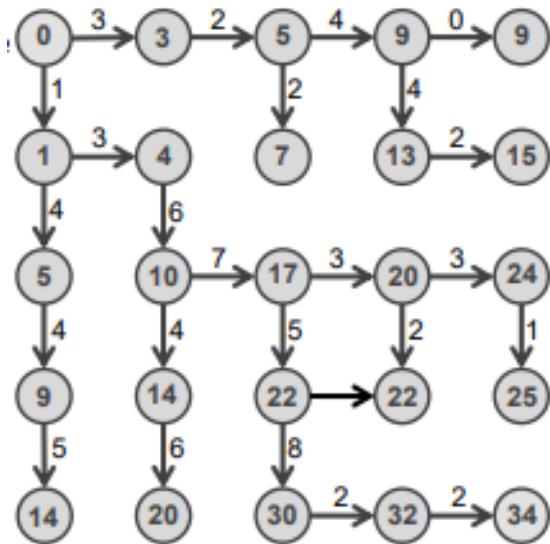
Ovaj algoritam se poziva za svaki čvor u grafu veličine $m \times n$, a pri tom se dešava da za jedan isti čvor računamo više puta. Zbog toga je ovaj pristup previše spor, pa prelazimo na dinamičko programiranje. Krenućemo od početnog čvora. Zatim, u čvor (i, j) upisujemo dužinu maksimalne putanje od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



Slika 4.10: Južno ili istočno?



Slika 4.11: Južno ili istočno?



Slika 4.12: Južno ili istočno?

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

4.5.1 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  $s_{n,m}$ 
```

4.6 Od Menhetna do grafa poravnjanja

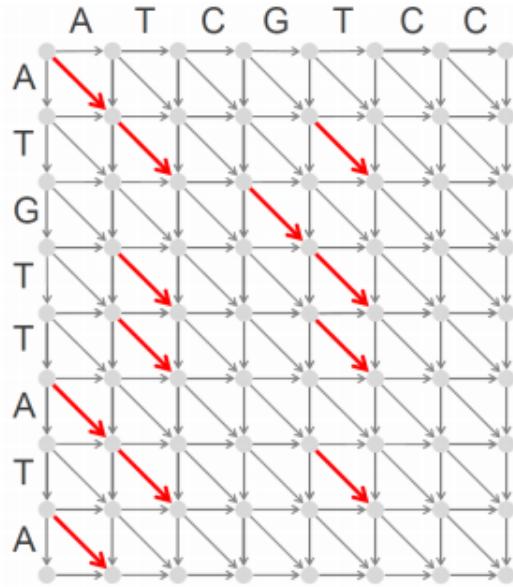
4.6.1 Rekurentna relacija dinamičkog programiranja kod grafa poravnjanja

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

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

Što dalje daje :

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

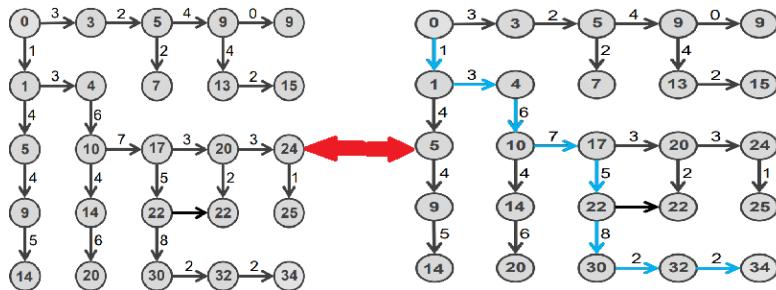


4.6.2 Računanje putokaza za povratak

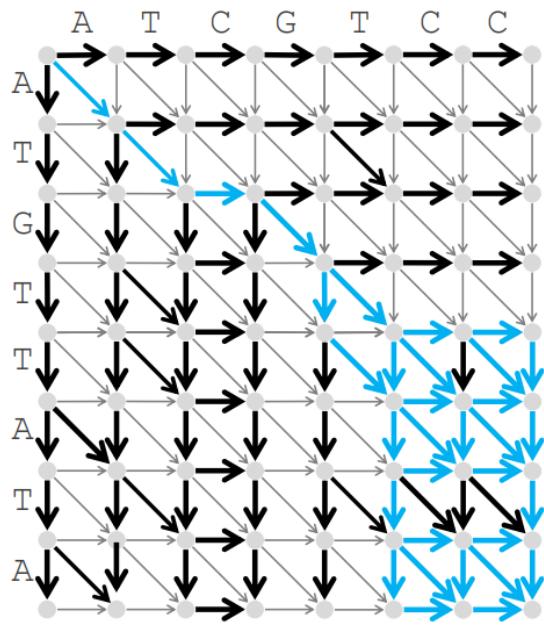
$$s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1, v_i = w_j \\ s_{i-1,j-1} + 0, v_i \neq w_j \end{cases}$$

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

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) ??.



Slika 4.15: Rekonstrukcija putanje preko putokaza kod Menhetn grafa



Slika 4.16: Backtracking

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

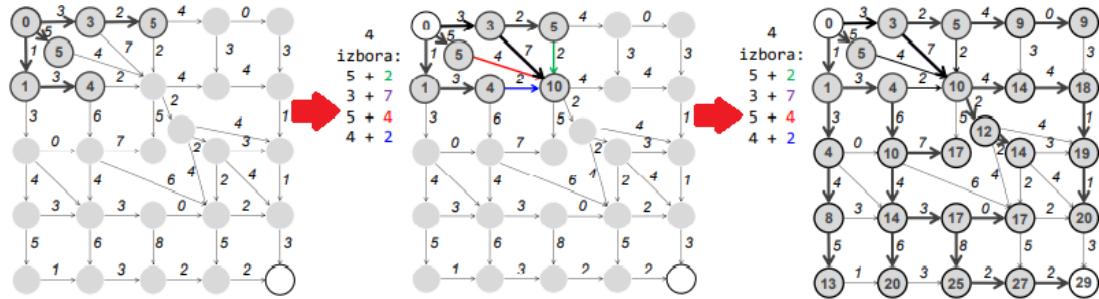
4.6.3 Određivanje najduže zajedničke podsekvence (LCS – longest common subsequence) korišćenjem putokaza za povratak

```

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? ??



Slika 4.17

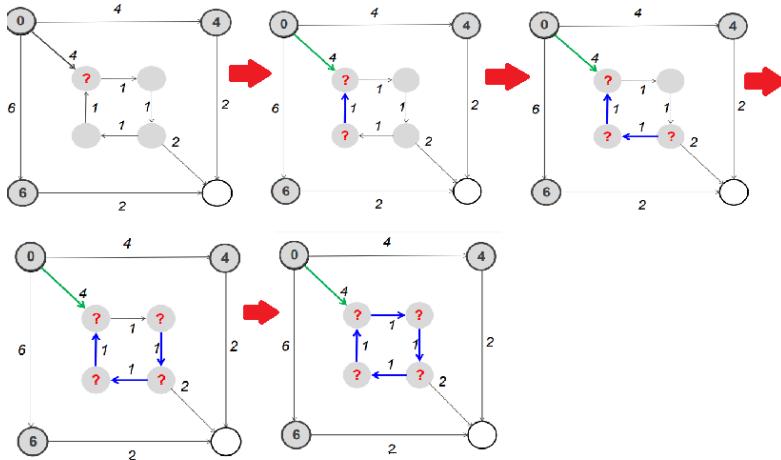
$$s_a = \max_{\text{all predecessors } b \text{ of node } a} \{s_b + \text{weight of edge from } b \text{ to } a\}$$

Računanje skora za SVE prethodnike ??.

- 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

4.6.4 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 4.18: 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

4.7 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 ??.

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	+1	$-\mu$	$-\mu$	$-\mu$	$-\sigma$
C	$-\mu$	+1	$-\mu$	$-\mu$	$-\sigma$
G	$-\mu$	$-\mu$	+1	$-\mu$	$-\sigma$
T	$-\mu$	$-\mu$	$-\mu$	+1	$-\sigma$
-	$-\sigma$	$-\sigma$	$-\sigma$	$-\sigma$	

Matrica skora

	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	

Još generalnija matrica skora

Slika 4.19

Navedimo rekurentnu relaciju dinamičkog programiranja kod grafa poravnjana. 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:

4.7.1 Globalno poravnanje

Problem 12 (Problem globalnog poravnjanja). Naći poravnanje sa najvišim skorom između dve niske za datu matricu skora.

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

Izlaz: Poravnjanje 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	

Slika 4.20

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

Uporedimo sledeća dva poravnanja:

```

GCC-C-AGT--TATGT-CAGGGGGCAGC-A-GCATGCGAGA- ---G---C---C---CAGTTATGTCAGGGGGCA CGAGCATGCAGA
GCCGCC-GTCGT-T-TTCAG---CA-CTTATG-T-CAGAT GCGCCCGTCTGTTTCA GAGTTATGTCAG---A---T

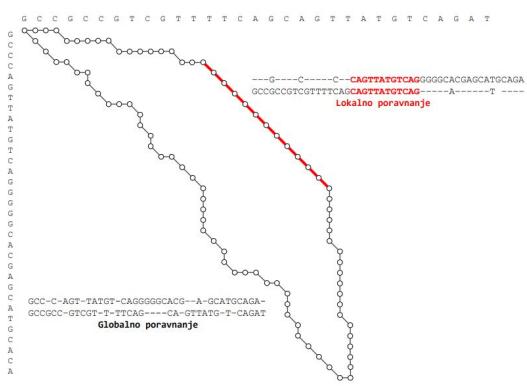
```

Slika 4.21

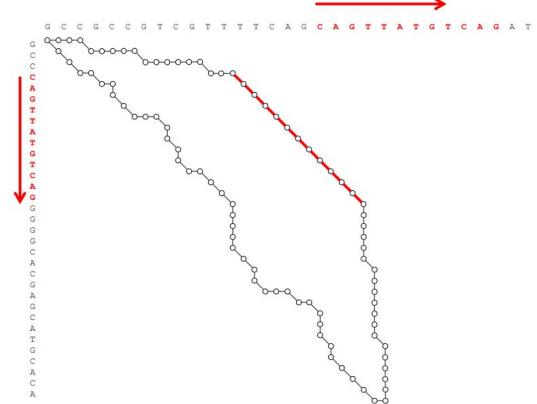
$$\text{score} = 22(\text{matches}) - 2(\text{indent}) = 2$$

Slika 4.22

score = 17 (matches) - 30(indent) = -13



Slika 4.23

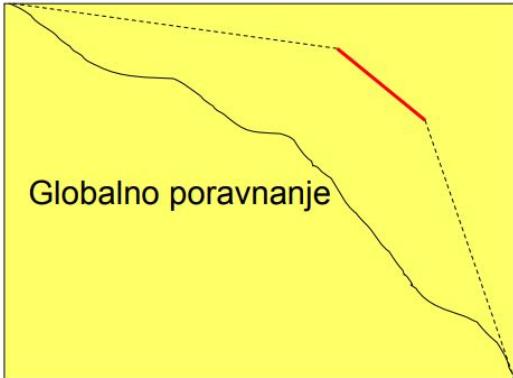


Slika 4.24

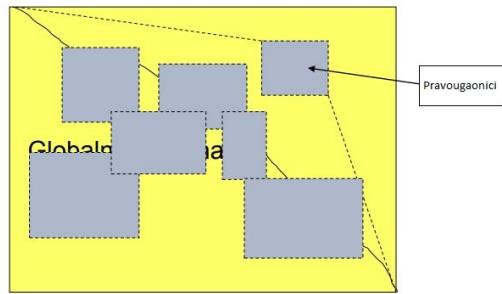
Ako se zapitamo koje od njih je bolje, iz priloženog zaključujemo da je to lokalno poravnanje.

4.7.2 Lokalno poravnanje

Lokalno poravnjanje računamo kao globalno poravnjanje u pravougaoniku, pogledajmo sliku ??.



Slika 4.25



Slika 4.26

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

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

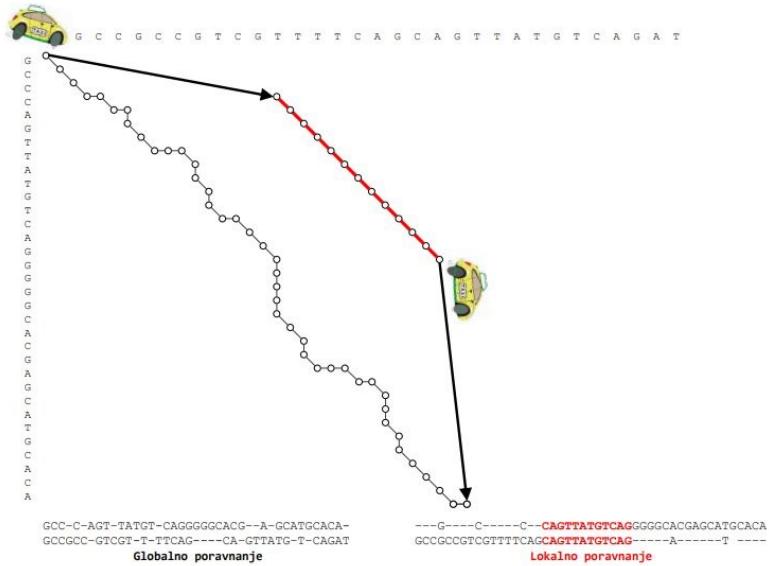
Stoga, broj ponavljanja algoritma će biti $\#nodes^2$ puta.

Problem 13 (Problem lokalnog poravnjanja). *Naći lokalno poravnanje najvećeg skora između dve niske.*

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

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

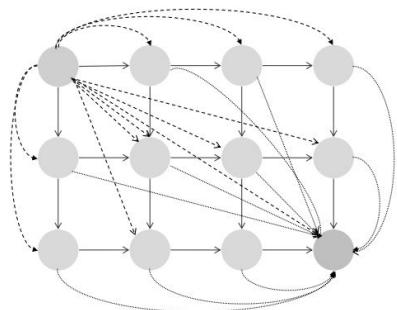
Zamislimo da postoji taksi koji bi nas besplatno vozio do tačke početka lokalnog poravnjanja, i od tačke završetka lokalnog poravnjanja pa do kraja. Na taj način ne bismo skupili negativne poene, već samo pozitivne. Ovakva vožnja nam daje samo skor lokalnog poravnjanja kao što smo i želeli. ??



Slika 4.27: Besplatne taksi vožnje

Konstruišimo Menhetn graf za problem lokalnog poravnanja:

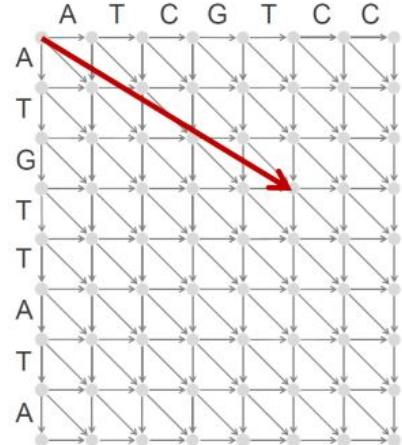
Kako bi izgledao Menhetn graf za nas problem?
 Dodamo grane težine 0 od (0,0) do svakog čvora, i od svakog čvora do (n,m).
 Ukupan broj dodatih grana je $O(|v| + |w|)$, pa algoritam ostaje brz.



Slika 4.28: 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 4.29: weight of (0,0) into (i,j) = 0

4.8 Kažnjavanje insercija i delecija u poravnanju sekvenci

4.8.1 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 ??

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

Slika 4.30

4.8.2 Adekvatnije kazne za praznine

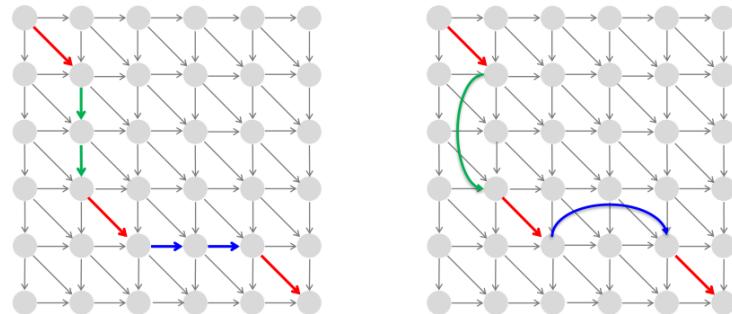
Afina kazna za praznine za prazninu dužine k: $\sigma + \epsilon * (k - 1)$

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

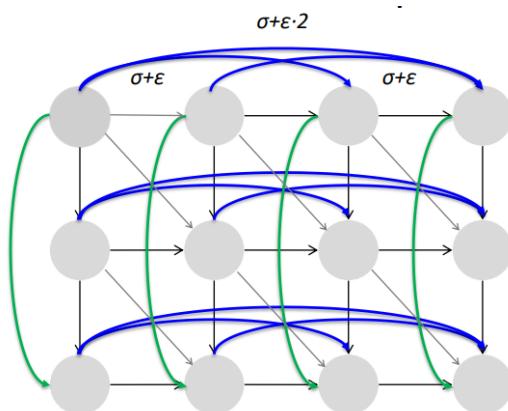
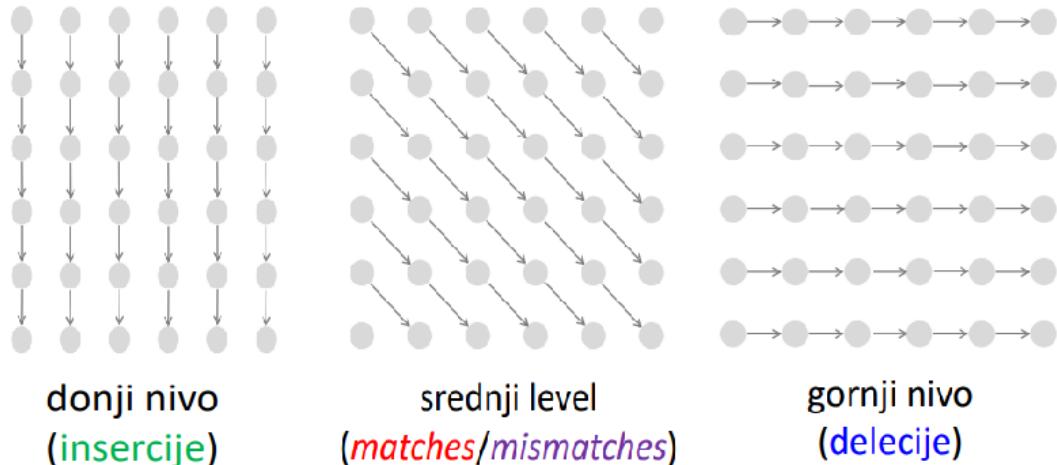
U slici ?? prikayano je modelovanje afinih kazni za praznine pomoću dugih grana.

4.8.3 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 $O(n^3)$??)
- Jedan način za smanjivanje broja grana je povećanje broja čvorova u grafu



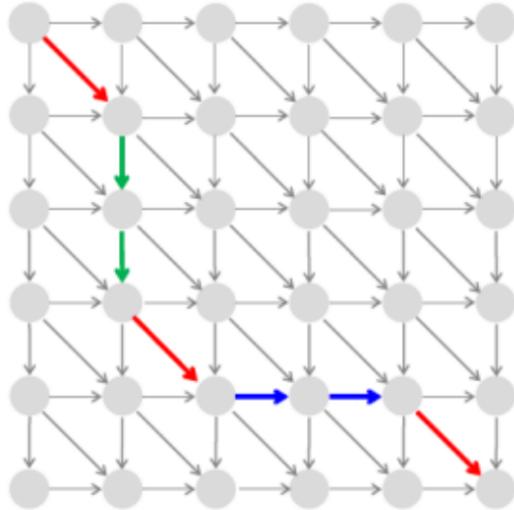
Slika 4.31: Modelovanje afinih kazni za praznine pomoću dugih grana

Slika 4.32: Dodali smo $O(n^3)$ grana

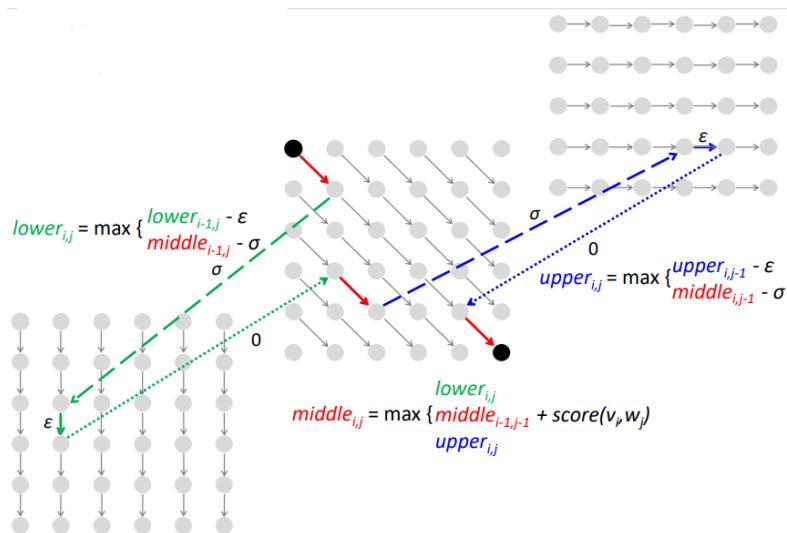
Slika 4.33: Podela Menhetna grafa na 3 nivoa

- Zato delimo Menhetn graf na tri nivoa ??

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



Slika 4.34: Kako predstaviti pomoću Menhetn grafa na 3 nivoa?



Slika 4.35: Simulacija Menhetn grafa na 3 nivoa

4.9 Prostorno efikasno poravnjanje sekvenci

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

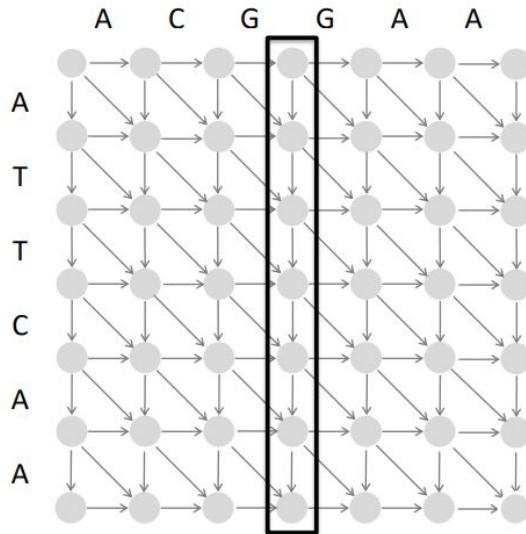
Uzmimo u obzir sledeće činjenice:

- 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-podjedjeljivanja.

Uvedimo nove pojmove:

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



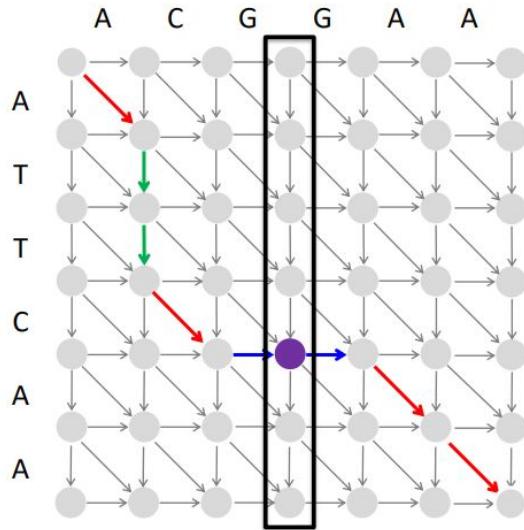
Slika 4.36: Srednja kolona poravnanja

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

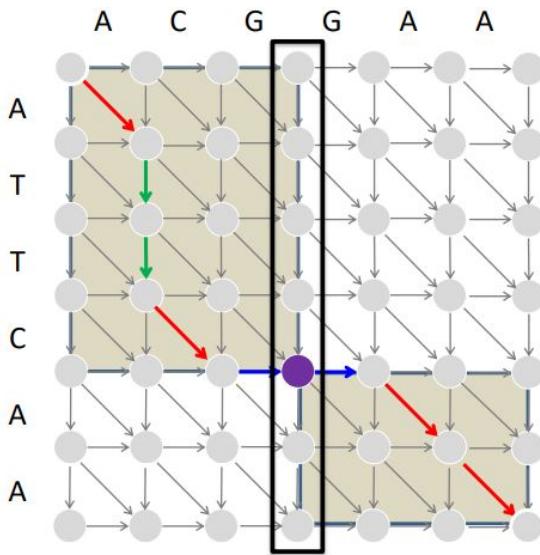
```

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

```



Slika 4.37: Srednji čvor poravnjanja



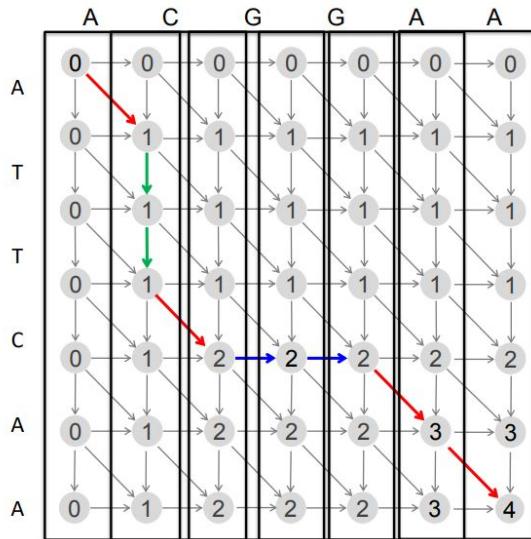
Slika 4.38: Srednja kolona poravnjanja

4.9.1 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 ??

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

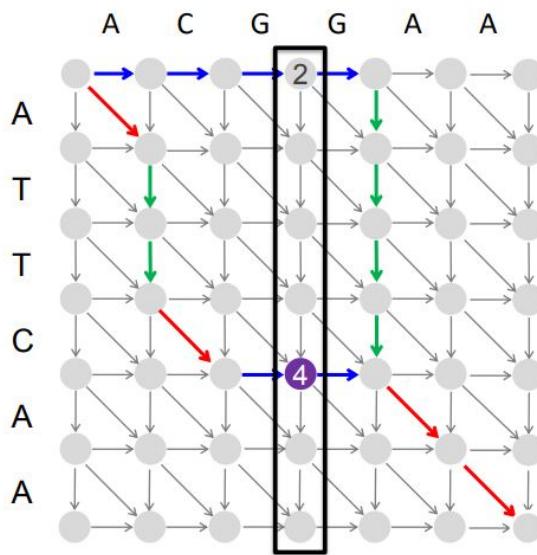


Slika 4.39: Recikliranje prostora za kolone u grafu poravnjanja

4.9.2 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-putanje.

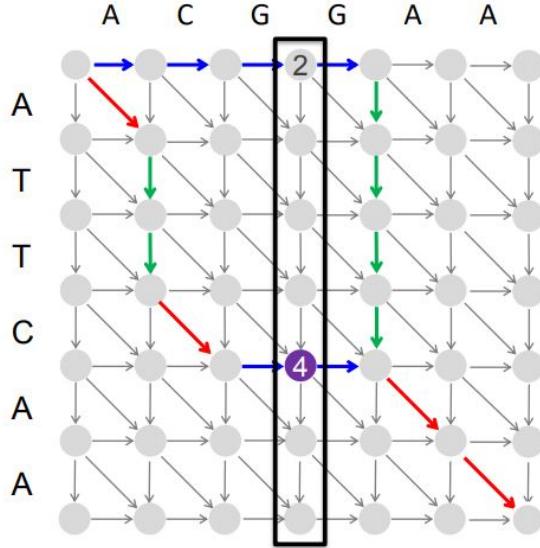
Na slici ?? vidimo da je $length(0) = 2$ i $length(4) = 4$.

Slika 4.40: Računanje $length(i)$

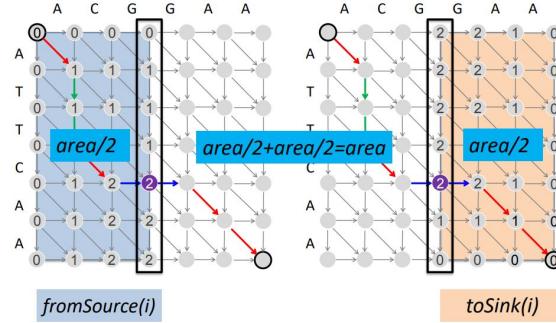
$length(i)$ možemo računati na sledeći način:

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

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

Slika 4.41: Računanje $\text{length}(i)$

vremena za nalaženje samo jednog čvora!



Slika 4.42: 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.

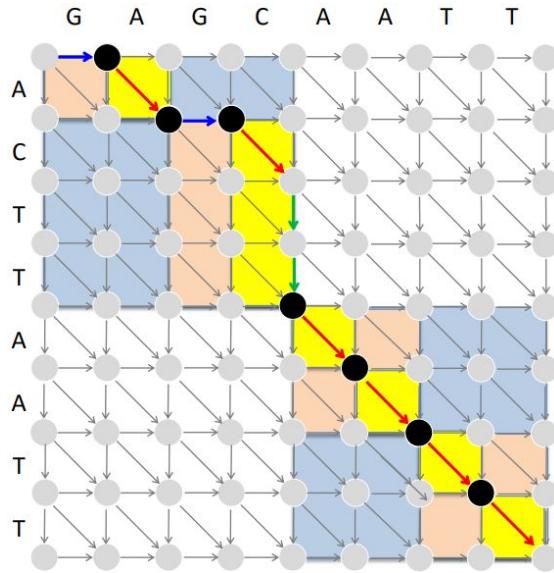
Dakle, vreme potrebno za nalaženje svih čvorova je: $\text{area} + \text{area}/2 + \text{area}/4 + \text{area}/8 + \dots < 2 * \text{area}$! Slika ?? vizuelno prikazuje vreme potrebno za nalaženje 7 tačaka.

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

4.10 Višestruko poravnanje sekvenci

4.10.1 Od dvostrukog do višestrukog poravnjanja

- Do sada su u poravnanju učestvovalo samo dve sekvence.



Slika 4.43: 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

4.10.2 Poravnjanje tri A-domena

Na slici ?? je prikazano poravnjanje tri A-domena

```

YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIIDVIAFRKMYGHTE-FINHYGPTEATIGA
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIKKYDITIFEATPALVIPMEYI-YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA---PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS
↓
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIIDVIAFRKMYGHTE-FINHYGPTEATIGA
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIKKYDITIFEATPALVIPMEYI-YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA---PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS
↓
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIIDVIAFRKMYGHTE-FINHYGPTEATIGA
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIKKYDITIFEATPALVIPMEYI-YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA---PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS

```

Slika 4.44: Poravnavanja 3 A-domena

4.10.3 Generalizacija dvostrukog na višestroko poravnjanje

- Poravnjanje 2 sekvene je matrica od 2 reda

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

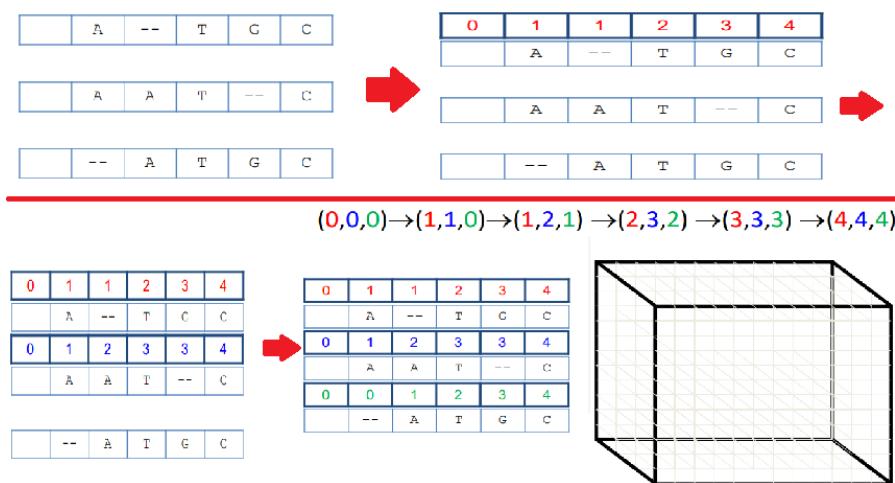
A	T	-	G	C	G	-
A	-	C	G	T	-	A
A	T	C	A	C	-	A

Slika 4.45

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

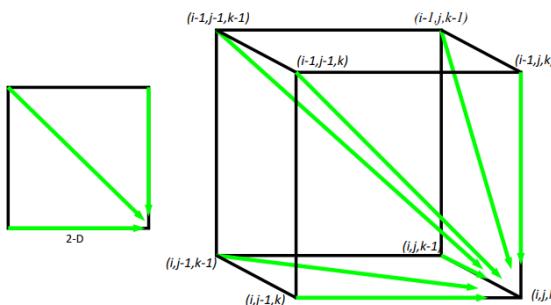
4.10.4 Poravnjanja = 3-D putanje

Poravnjanje sekvenci ATGC, AATC i ATGC ??



Slika 4.46: Poravnjanje sekvenci ATGC, AATC i ATGC

4.10.5 2-D poravnjanje u odnosu na 3-D poravnjanje



Slika 4.47

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

4.10.6 Rekurentna relacija dinamičkog programiranja za višestruko poravnanje

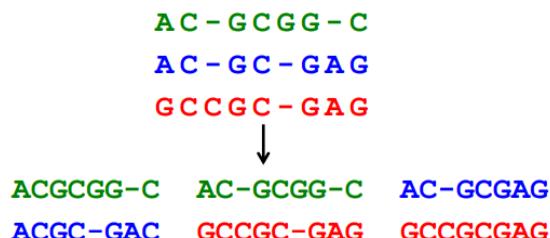
$$s_{i,j,k} = \max \begin{cases} s_{i-1,j-1,k-1} + \delta(v_i, w_j, u_k) \\ s_{i-1,j-1,k} + \delta(v_i, w_j, -) \\ s_{i-1,j,k-1} + \delta(v_i, -, u_k) \\ s_{i,j-1,k-1} + \delta(-, w_j, u_k) \\ s_{i-1,j,k} + \delta(v_i, -, -) \\ s_{i,j-1,k} + \delta(-, w_j, -) \\ s_{i,j,k-1} + \delta(-, -, u_k) \end{cases}$$

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

4.10.7 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 sekvenca, 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 ??



Slika 4.48

4.10.8 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 ??)?

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

Slika 4.49

-	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 4.50

4.10.9 Profilna reprezentacija višestrukog poravnjanja

Na slici ?? 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**?

4.10.10 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 ??

<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 4.51

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

$$\left. \begin{array}{ll} \textcolor{red}{s_2} & \textcolor{green}{\text{GTCTGA}} \\ \textcolor{blue}{s_4} & \textcolor{red}{\text{GTCAGC}} \end{array} \right\} s_{2,4} = \textcolor{red}{\text{GT}}\textcolor{green}{\text{Ct}}/\textcolor{blue}{\text{a}}\textcolor{red}{\text{G}}\textcolor{blue}{\text{a}}/\textcolor{red}{\text{c}}$$

Slika 4.52

- Novi skup od 3 sekvence za poravnjanje:

s₁ GATTCA

s₃ GATATT

s_{2,4} **GT***Ct*/**a***G***a**/**c**

4.11 Zadaci sa vežbi

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

4.11.1 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()

```

4.11.2 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()

```

4.11.3 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()

```

4.11.4 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()

```

4.11.5 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 = "AAATTTGGGCCCGGAAATTCCC"
56     w = "AAACCCCTTGGGCCCTTAAACCC"
57
58     print(edit_distance(v, w))
59
60 if __name__ == "__main__":
61     main()
```

