

BIOINFORMATIKA

31. mart 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 genomsku 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	Eksplzija u štampanju	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
	Literatura	39

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

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 Meriam-Webster (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.

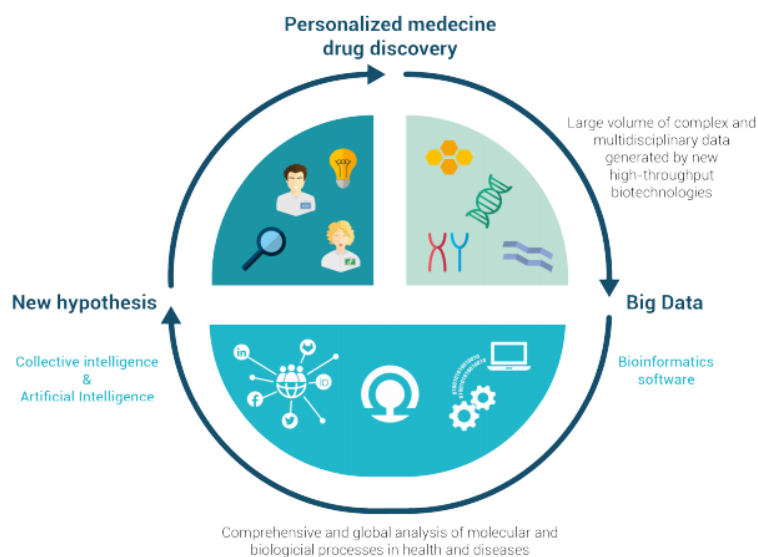
Bioinformatika ima široku primenu i njene primene rastu zajedno sa razvojem discipline. Kao što možemo videti na slici ispod, primena bioinformatike se može sagledati kroz personalizovanu medicinu. Naime, na osnovu prikupljene veće količine podataka i njihove analize, uz pomoć različitih računarskih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

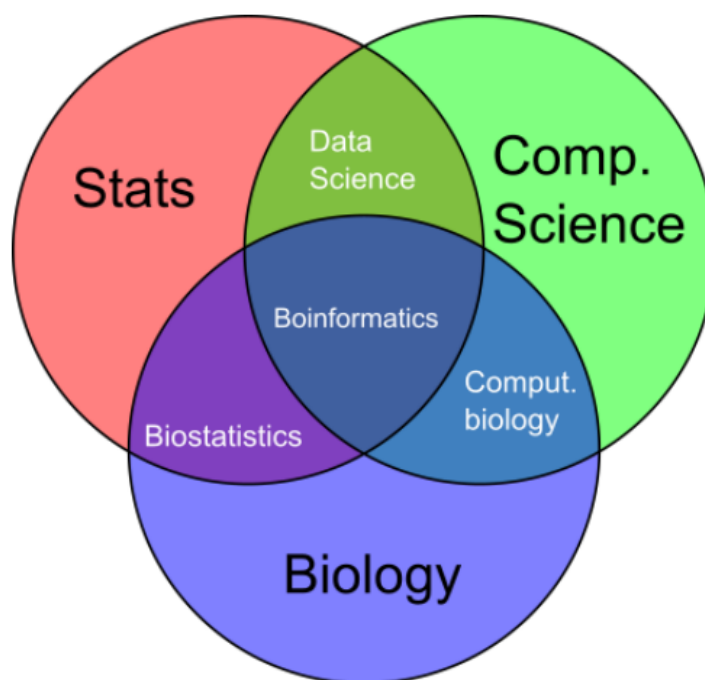
DNK se sastoji iz dva polimerna lanca koji imaju antiparalelnu orijentaciju, i svaki od njih je sastavljen od azotnih baza:

- 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

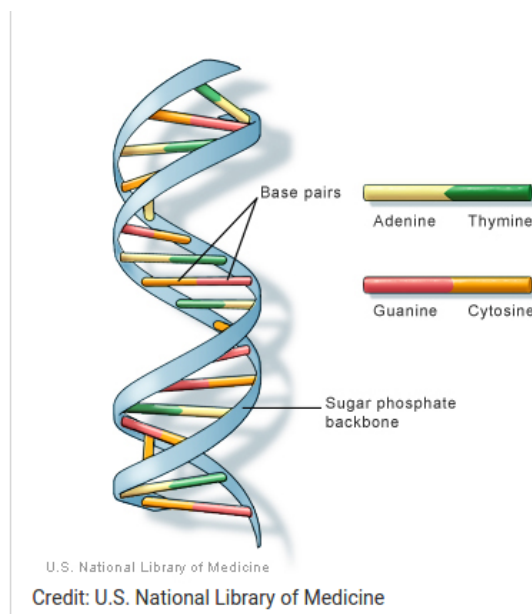
Replikacija genoma je jedan od najvažnijih zadataka ćelije. Pre nego što se podeli, ćelija mora da najpre replicira svoj genom, tako da svaka od ćerki ćelija dobije svoju kopiju.

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 aminokiseline A-C i G-T, rekreiranje lanca je moguće). Kao rezultat ovakvog ponašanja, proces replikacije počinje parom komplementarnih lanca 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 funkcionisanja 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 prodru kroz ćelijski zid (poput pravih virusa). Virusni vektori u sebi nose veštačke gene koji unapređuju 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, zarazí viralnim vektorom koji sadrži veštački gen koji enkodira terapijski 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.

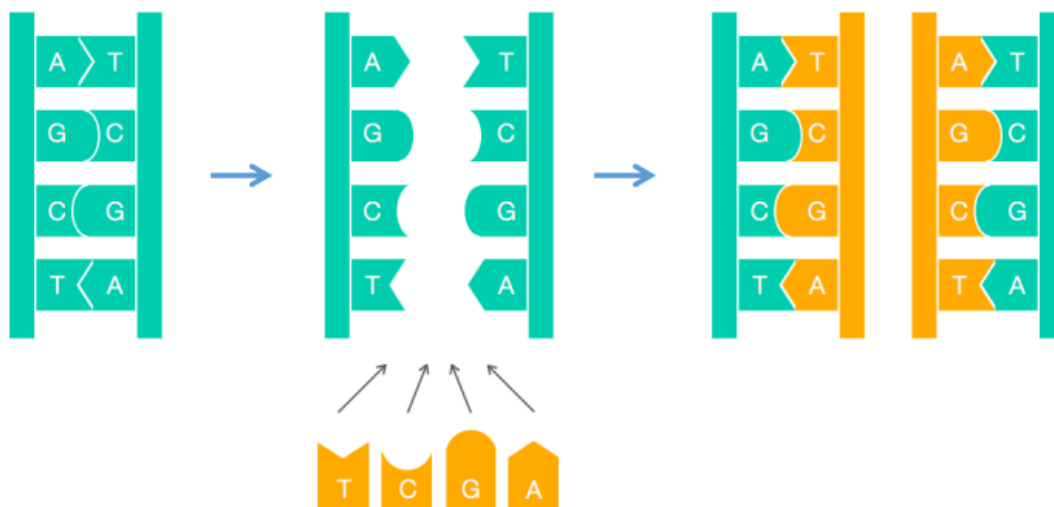
Poznato je da DNKA utiče na početak replikacije. DNKA je protein koji se vezuje na kratki segment unutar *oriC*, poznatiji kao **DNKA boks**. Ona predstavlja poruku unutar sekvence DNK koja govori proteinu DNKA da se veže baš tu. Postavlja se pitanje kao pronaći taj region bez prethodnog poznavanja izgleda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

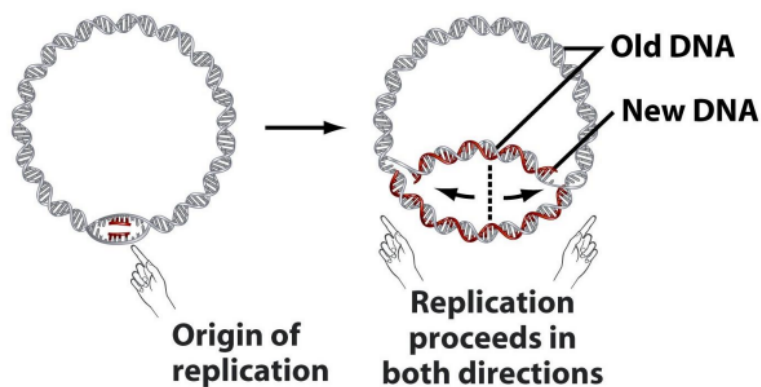
53++!305))6*;4826)4+.)4+);806*;48!8'6 0))85;]8*:*8!83(88)5*!;46(;88*96*?;8
)**(;485);5*!2:*+(;4956*2(5*4)8'8*;40 69285);)6!8)4++;1(+9;48081;8:8+1;48!8 5;4)
485!528806*81(+9;48;(88;4(+?34;48)4+;161;:188;+?;

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(+9THET(EETH(+?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 $\text{COUNT}(\text{Text}, \text{Pattern})$ 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 pronalaska 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 pronalazak 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 $|Text| - k + 1$ proveru, 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: Pronalazak 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 **ATGATCAAG** 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. Verovatnoća da postoji 9-gram koji se pojavljuje 3 ili više puta u proizvoljno generisanom DNK stringu dužine 500 je $1/1300$. Uočimo da postoje četiri različita 9-grama koji se ponavljaju tri ili više puta u ovom regionu, to su: ATGATCAAG, CTTGATCAT, TCTTGATCA i CTCTTGATC.

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

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

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

```

atcaatgatcaacgtaagcttctaagcATGATCAAGgtgctcacacagtttatccacaacctgagtgg
atgacatcaagataggtcgttgatatctccttcctctcgtactctcatgaccacggaaagATGATCAAG
agaggatgatttcttggccatatcgcaatgaatacttgtagcttggtgcttccaattgacatcttcagc
gccatattgctgctggccaaggtgacggagcgggattacgaaagcatgatcatggctggtgttctgttt
atcttgttttgactgagacttgtaggatagacggtttttcatcactgactagccaaagccttactct
gcctgacatcgaccgtaaattgataatgaatttacatgcttcgcgacgatttacctCTTGATCATcg
atccgattgaagatcttcaattgttaattctcttgctcgactcatagccatgatgagctCTTGATCA
TgttttccttaaccctctatttttttacggaagaATGATCAAGctgctgctCTTGATCATcgtttc

```

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.

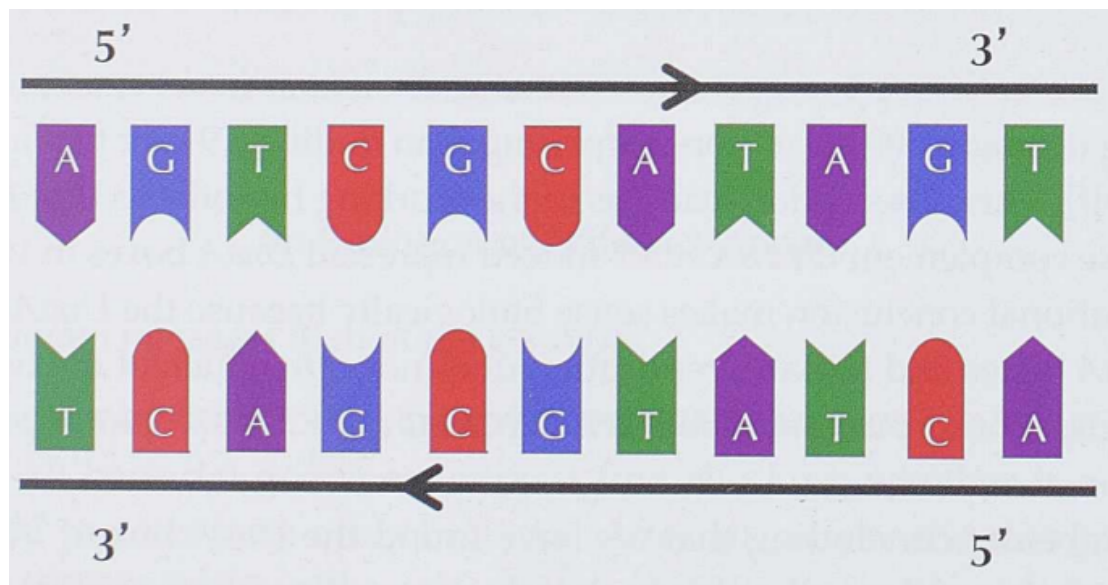
Posmatrajmo ponovo sliku 1.7. Na njoj možemo uočiti 6 pojavljivanja niski ATGATCAAG i CTTGATCAT, koji su zapravo komplementarni. Naći 9-gram koji se pojavljuje 6 puta u DNK nisci dužine 500 nukleotida, je još više iznenađujuće, nego pronaći 9-gram koji se pojavljuje tri puta. Ovo posmatranje nas dovodi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

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

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

```

aactctatacctcctttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaactaa
aatggtaggttttggtggttaggttttgtgtacattttgtagtatctgatttttaattacataccgta
tattgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaacctaccaccaaac
tctgtattgaccatttttaggacaacttcaggggtggttaggtttctgaagctctcatcaatagactat
tttagtctttacaacaatattaccggttcagattcaagattctacaacgctgttttaatgggcgtt
gcagaaaacttaccacctaataatccagtatccaagccgatttcagagaaacctaccacttacctac
cacttacctaccacccgggtggttaagttgcagacattattaaaaacctcatcagaagcttggtcaa
aaatttcaatactcgaacacctaccacctgcgtccctattatttactactactaataatagcagta
taattgatctgaaaagaggtggtataaaaa

```

kako bi pružile skrivenu poruku DNKA proteinu. Odnosno, za različite genome imamo različite DNKA boksove.

Najčešće reči u ovom *oriC* su:

- AACCTACCA,
- ACCTACCAC,
- GGTAGGTTT,
- TGGTAGGTT,
- AAACCTACC,
- CCTACCACC

Pomoću alata koji se zove Ori-Finder, nalazimo CCTACCACC i njegov komplement GGTGGTAGG 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*

```
aactctatacctcctttttgtcgaatttgtgtgatttatagagaaaatcttattaactgaaactaa
aatggtaggtttGGTGGTAGGttttgtgtacattttgtagtatctgatttttaattacataccgta
tattgtattaaattgacgaacaattgcatggaattgaatatatgcaaaacaaaCCTACCACCaaac
tctgtattgaccatttttaggacaacttcagGGTGGTAGGttttctgaagctctcatcaatagactat
tttagtctttacaacaataattaccgttcagattcaagattctacaacgctgttttaaatgggcgtt
gcagaaaacttaccacctaataatccagtatccaagccgatttcagagaaacctaccacttacctac
cacttaCCTACCACCcggggtggttaagttgcagacattattaaaaacctcatcagaagcttggtcaa
aaatttcaataactcgaaaCCTACCACCTgcgtcccctattattttactactactaataatagcagta
taattgatctgaaaagaggtggttaaaaaa
```

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 genoma, dužine L , u kome se k-gram pojavljuje barem t puta.

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

Ulaz: 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?

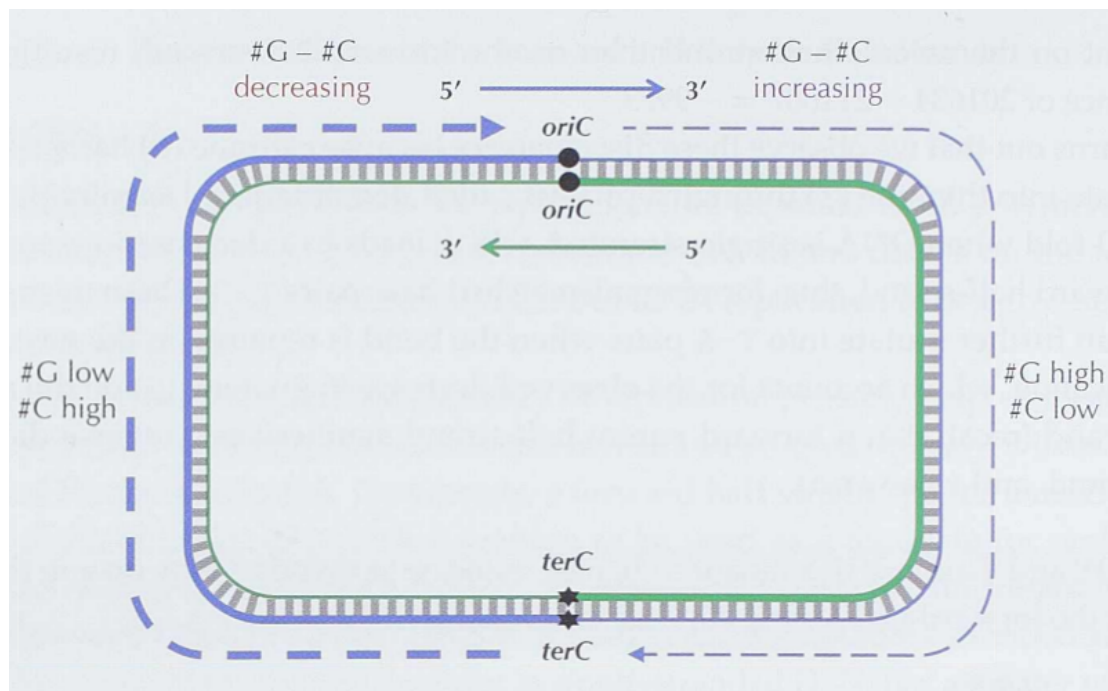
Iskrivljeni dijagrami

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 pretpostaviti da se krećemo niz polulanac koji ide na desno (u nastavku samo polulanac, smer $5' \rightarrow 3'$), a ako razlika počne da se smanjuje, onda pretpostavljamo da smo na obrnutom polulancu ($3' \rightarrow 5'$). 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.

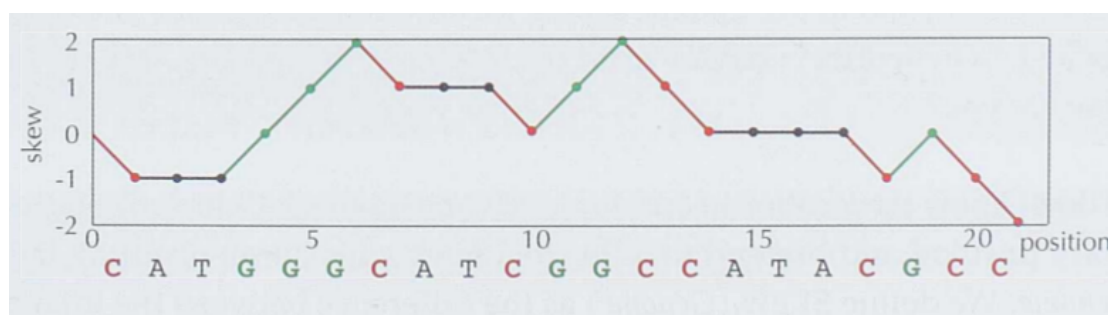
Posmatrajmo iskrivljeni dijagram bakterije *Ešerihija Koli*. Lako uočavamo minimalnu vrednost skew dijagrama.

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.

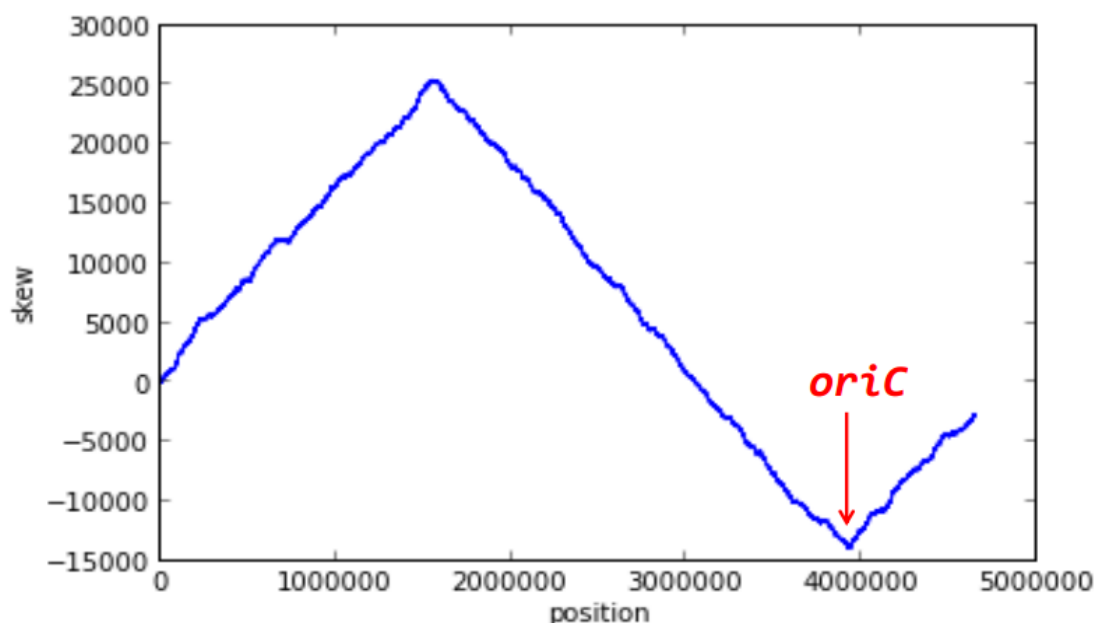
```

aatgatgatgacgtcaaaaggatccggataaaacatgggtgattgcctcgcataacgcggta
tgaaaatggattgaagcccgccggtggattctactcaactttgtcggcttgagaaagacc
tgggatcctgggtattataaaagaagatctatttatttagagatctgttctattgtgatctc
ttattaggatcgactgcctgtggataacaaggatccggcttttaagatcaacaacctgg
aaaggatcattaactgtgaatgatcggtgatcctggaccgtataagctgggatcagaatga
ggggttatacacaactcaaaaactgaacaacagttgttctttggataactaccggttgatc
caagcttcctgacagagttatccacagtagatcgcacgatctgtatacttatttgagtaa
ttaaccacgatcccagccattcttctgcccggatcttccggaatgtcgtgatcaagaatgt
tgatcttcagtg

```

Uočimo da u ovom regionu nema čestih 9-grama (koji se pojavljuju 3 ili više puta). Iz toga zaključujemo da, iako smo uspjeli da nađemo *oriC* bakterije Ešerihije koli, nismo uspjeli 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
ctgagtggatgacatcaagataggtcggttgatatctccttcctctcgtactctcatgacca
cggaaagATGATCAAGagaggatgatttcttggccatatcgcaatgaatacttgtgactt
gtgcttccaattgacatcttcagcgccatattgcgctggccaaggtagcgagcgggatt
acgaaagCATGATCATggctggtgttctgtttatcttggtttgactgagacttgtagga
tagacgggtttttcatcactgactagccaaagccttactctgctgacatcgaccgtaa
tgataatgaatttacatgcttccgcgacgatttacctCTTGATCATcgatccgattgaag
atcttcaattgttaattctcttgctcgactcatagccatgatgagctCTTGATCATggtt
tccttaaccctctattttttacggaagaATGATCAAGctgctgctCTTGATCATcgtttc
```

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.

```
aatgatgatgacgtcaaaaggatccggataaaacatgggtgattgcctcgcataacgcgg
tatgaaaatggattgaagcccgccgtggattctactcaactttgtcggcttgagaaa
gacctgggatcctgggtattaaaaagaagatctattttatttagagatctgttctattgt
gatctcttattaggatcgactgcccTGTGGATAACAaggatccggcttttaagatcaa
caacctggaaaggatcattaactgtgaatgatcggatcctggaccgtataagctggg
atcagaatgaggggTTATACACAactcaaaaactgaacaacagttgttcTTGGATAAC
taccggttgatccaagcttcctgacagagTTATCCACAgtagatcgacgatctgtata
cttatttgagtaaattaaccacgatcccgccattcttctgccggatcttccggaatg
tcgtgatcaagaatggttgatcttcagtg
```

1.3 Zadaci sa vežbi

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

1.3.1 Frequent Words

```

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('agctagatgctagctagctgatcgagctgatgcaggcagtgctagc', 4,
28         ↪ 2))
29
30 if __name__ == "__main__":
31     main()

```

1.3.2 Faster Frequent Words

```

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('agctagatgctagctagctgatcgagctgatgcaggcagtgctagc
    ↪ ', 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 = "catgggcatcggccatacgcc"
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     # Prevođenje broja u odgovarajuću 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     # Prevođenje nukleotidne sekvence u odgovarajući 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 dužina 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     ↪ najviše d pozicija
64     def approximate_pattern_count(text, pattern, d):
65         count = 0
```

```

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

```

```
109     for pattern in neighborhood:
110         index = pattern_to_number(pattern) # Svaka kandidat sekvenca se
        ↪ prevodi u svoj odgovarajući indeks
111         close[index] = 1 # i evidentira
112
113     # Za svaku kandidat sekvenču (koja je do sada pronadjena u tekstu sa
        ↪ greskom d), broji se pojavljivanje njenih d-suseda
114     for i in range(4**k):
115         if close[i] == 1:
116             pattern = number_to_pattern(i, k)
117             frequency_array[i] = approximate_pattern_count(text, pattern, d
        ↪ )
118
119     max_count = max(frequency_array)
120
121     # Pronalaze se one sekvence duzine k cija su pojavljivanja najzastupljenija
122     for i in range(4**k):
123         if frequency_array[i] == max_count:
124             pattern = number_to_pattern(i, k)
125             frequent_patterns.add(pattern)
126
127     return frequent_patterns
128
129
130
131
132 def main():
133     print(frequent_words_with_mismatches('
        ↪ tgactatcatcgtagtatcgatgtgcacacagtgcgcgcgccctgtacatgac', 5, 2))
134
135 if __name__ == "__main__":
136     main()
```


Glava 2

Kako složiti genomsku slagalicu od milion delova?

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

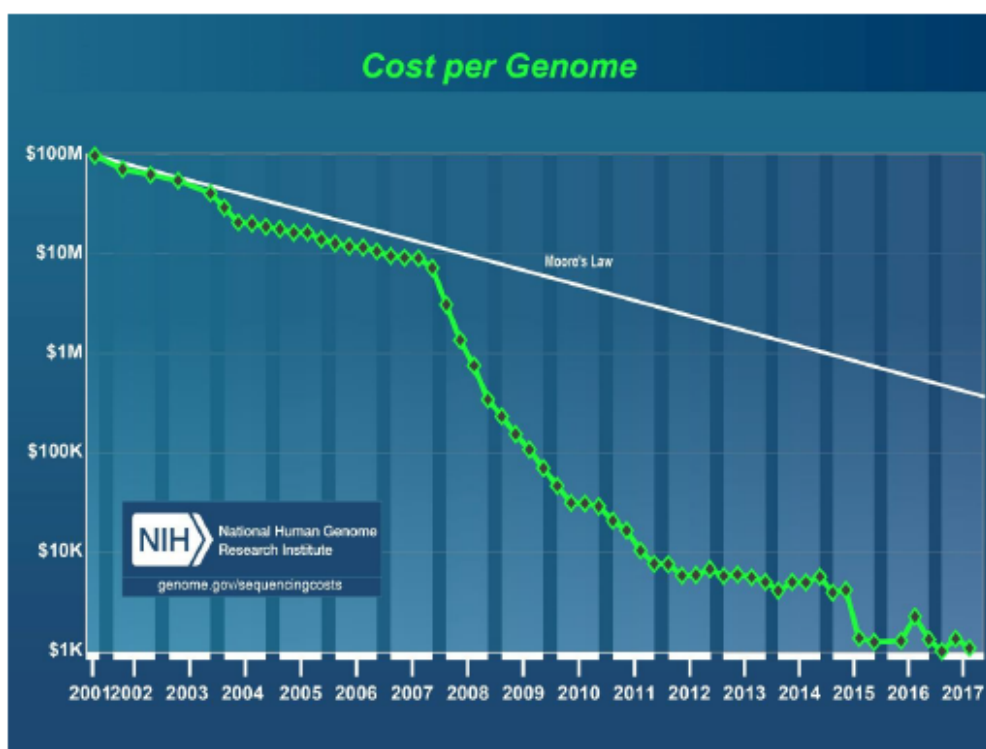
Kod čoveka, genom sadrži oko tri milijarde nukleotida. Dakle, sa računarske strane posmatrano genom je niska karaktera nad azbukom $\Sigma = \{A, C, G, T\}$. Kompleksnost organizma nije u relaciji sa veličinom genoma. Genomi nekih organizama su i stotinu puta veći od humanog genoma. Na primer, jedna 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 razdvoja 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 sekvenciranje 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. *Next Generation Sequencing*) predstavlja metode nove generacije sekvencioniranja, odnosno, novu generaciju mašina sekvencera koji vrše sekvencioniranje. *Illumina*, jedan od proizvođača sekvencera, smanjuje trošak sekvencioniranja humanog genoma sa 3 milijarde na 10 hiljada dolara. Kompanija *Complete Genomics* otvara genomsku fabriku u Silikonskoj dolini koja sekvencionira stotine genoma mesečno. Pekinški genomski institut (*Beijing Genome Institute*, skr. *BGI*) preuzima Complete Genomics 2013. godine i postaje najveći svetski centar za 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

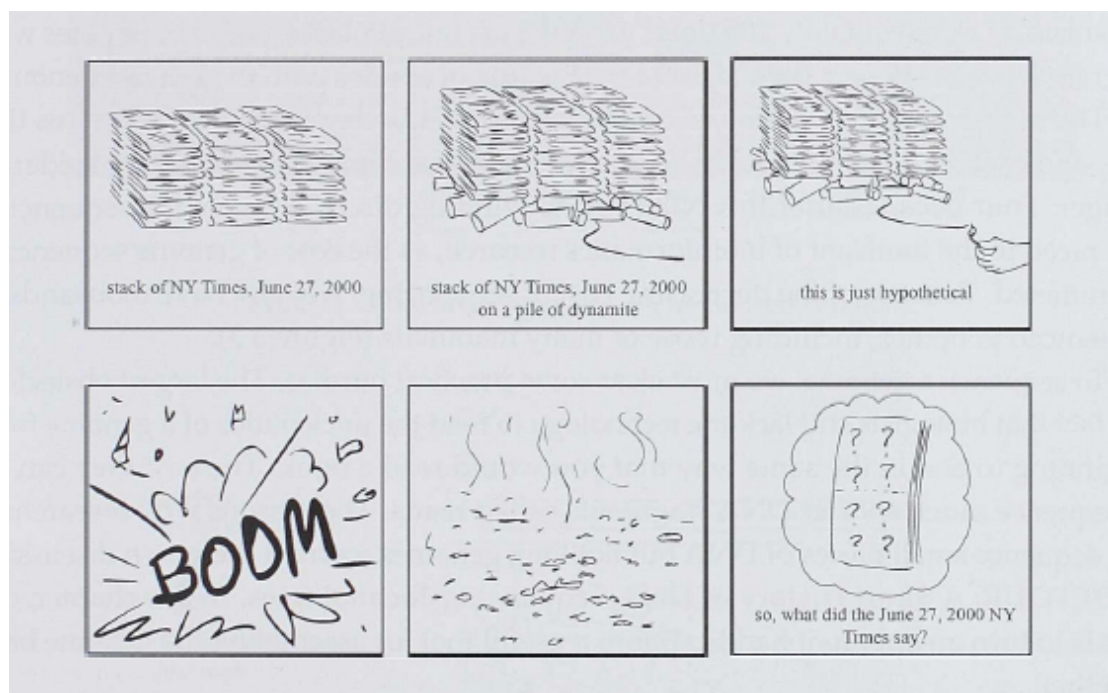
Prirodno je zapitati se vrednosti dobijka poznavanja sastava genoma. Što je tiče biljaka, neke od primena sekvenciranja su: razvoj novih biljnih vrsta u poljoprivredi, određivanje pogodnog podneblja za neku biljnu vrstu, u farmaciji, i dr. Međutim, najznačajnija primena je u sekvenciranju ličnih genoma.

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.

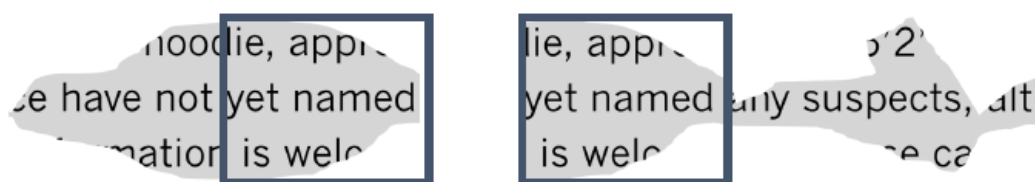
2.2 Eksplozija u štampariji

Razmotrimo sledeći primer. Neka imamo hiljadu kopija istog izdanja novina na jednoj gomili, a ispod njih postavljen je dinamit. Upalimo fitilj 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.



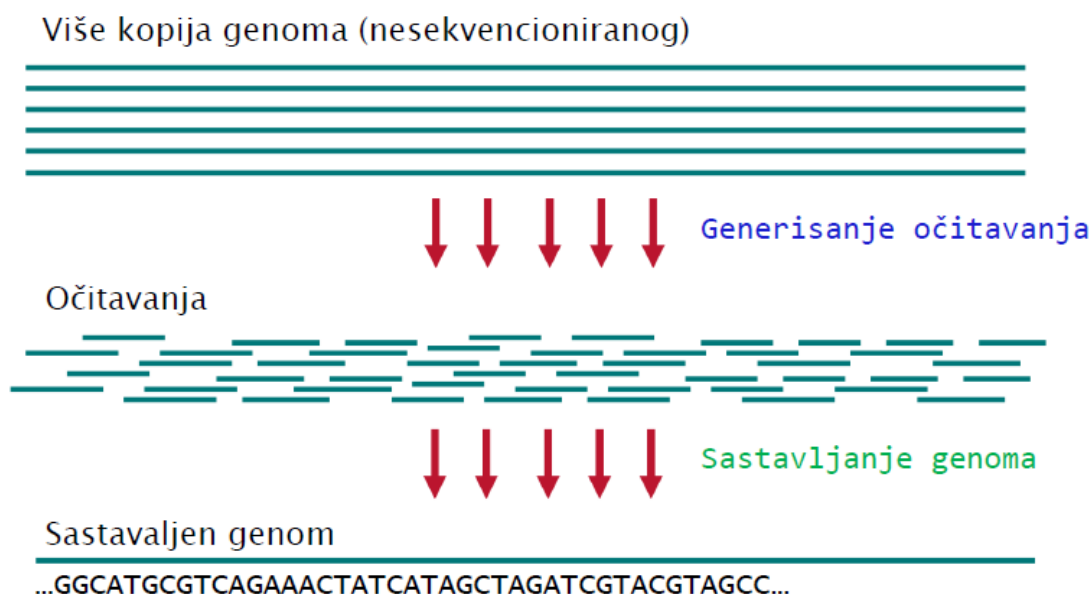
Slika 2.3: Spajanje delova različitih novina koji se jednim delom preklapaju.

Određivanje redosleda nukleotida u genomu, odnosno sekvenciranje genoma, predstavlja bitan problem u bioinformatičari. 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 iseckaju 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 deliće odnosno kratke podniske. Neki delovi odnosno očitavanja biće izgubljena (kao delići novina u eksploziji, dakle gubimo deo informacija). 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 uspeali da problem sekvencioniranja genoma predstavimo kao problem rekonstrukcije niske.

Definišemo pojam *k*-gramski sastav niske na sledeći način. *k*-gramski 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-gramski 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 $Composition_k(Genome)$ 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-gramski 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 jednaki. Nadovezujemo ga na tekuću nisku i dobijamo TAATG. Tražimo 3-gram sa prefiksom TG, koji do sad nisu upotrebljeni. Ponovo pronalazimo 3 kandidata. Međutim, u ovom slučaju, svi kandidati predstavljaju različite 3-gramove, 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 zaglavlivanja. Imamo još 3-grama koji nisu iskorišćeni za rekonstrukciju niske, ali nijedan ne možemo da iskoristimo u ovom trenutku. U takvim situacijama treba se vratiti u nazad do koraka u kom je bilo više kandidata.

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

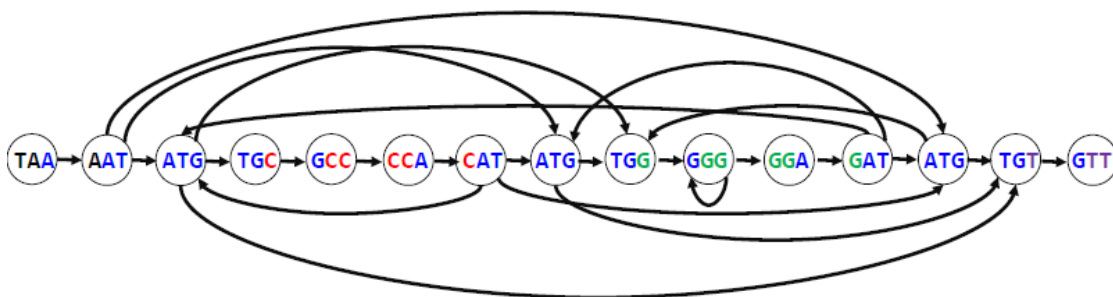
Vratimo se na prethodni primer. Dat nam je naredni 3-gramski sastav:

```

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-gramski 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.

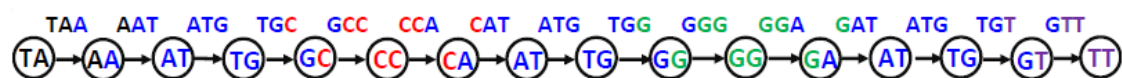
Izlaz: Putanja koja posećuje svaki čvor u grafu tačno jednom.

Iako deluje kao da smo rešili sve probleme, zapravo smo naišli na još jednu veliku prepreku. Naime, pronalaženje Hamiltonovog puta u grafu je NP-kompletni 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 grafom.

2.5 Rekonstrukcija niske kao Ojlerove putanje

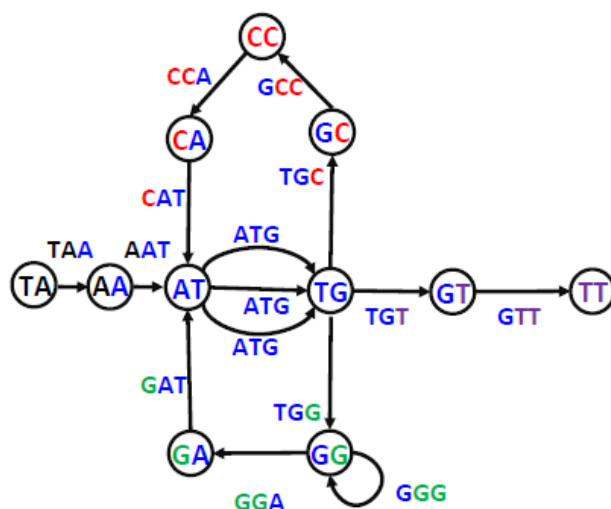
U prethodnoj sekciji k -grame smo predstavili čvorovima u grafu i u njemu tražili Hamiltonov put, odnosno, put koji obilazi svaki čvor tačno jednom. Videli smo da za taj problem još uvek nije poznat efikasan algoritam pa se sada pitamo kako možemo izmeniti graf tako da ne zahteva traženje Hamiltonove 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: Graf koji odgovara 3-gramskom sastavu niske *TAATGCCATGGGATGTT*. Grane su obeležene 3-gramima, a čvorovi 2-gramima koji predstavljaju prefikse i sufikse.

Primećujemo da su neki čvorovi obeleženi identično (na primer, imamo tri čvora sa oznakom *AT*). Sve čvorove koji imaju istu oznaku treba spojiti u jedan, pri čemu zadržavamo sve grane koje su ulazile u taj čvor ili su izlazile iz njega. Ponavljamo postupak dokle god imamo čvorove koji imaju istu oznaku i na kraju dobijamo graf koji nazivamo *De Brojnov graf*. Slika 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. Prirodni 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-kompletna i možemo efikasno da je pronađemo.

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

Ulaz: Graf.

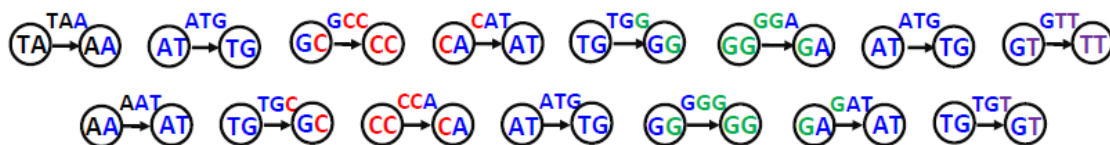
Izlaz: Putanja koja posećuje svaku granu u grafu tačno jednom.

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

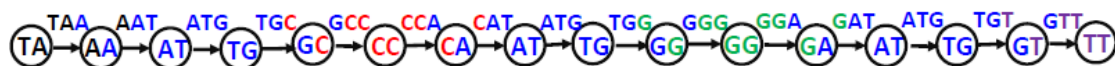
2.6 De Brojnovi grafovi na osnovu kolekcije k -grama

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

Za svaki k -gram pravimo dva čvora i jednu granu – oznaka grane je upravo taj k -gram, oznaka izlaznog čvora je prefiks, a ulaznog čvora je sufiks datog k -grama. Time dobijamo nepovezani graf kao na slici 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 predstavljen 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 zalepljeni.

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 ma koja dva čvora postoji putanja koja ih povezuje. Graf je *balansiran* ako za svaki čvor važi da mu je izlazni stepen jednak ulaznom. Naredna teorema govori o potrebnim i dovoljnim uslovima da graf bude Ojlerov.

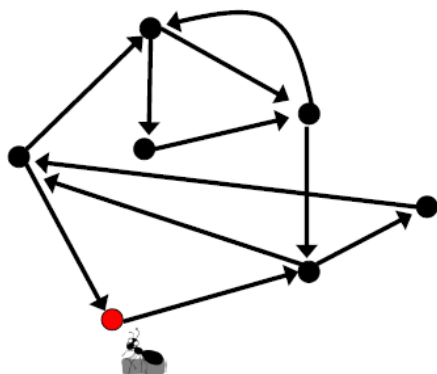
Teorema 2.1 (Ojlerova teorema). *Svaki Ojlerov graf je balansiran. Svaki povezan graf i balansiran graf je Ojlerov.*

2.7.1 Dokaz Ojlerove teoreme

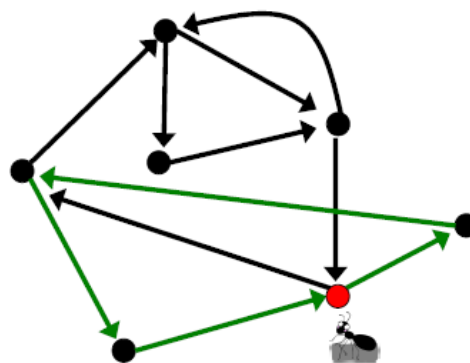
Neka nam je dat povezan balansiran graf. Da bismo pokazali da graf sadrži Ojlerov ciklus, postavimo mrava na bilo koji od čvorova tog grafa, kao na slici 2.10. Zašto baš mrav? Poznato je da mravi nikada ne idu istim putem dva puta pa smo sigurni da će naš mrav proći dvaku granu tačno jednom.

Puštamo mrava da slučajno odabira grane kojima će se kretati. Ako je veoma pametan, obići će 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 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 koji ima izlaznih grana koje još uvek nije obišao (slika 2.11).



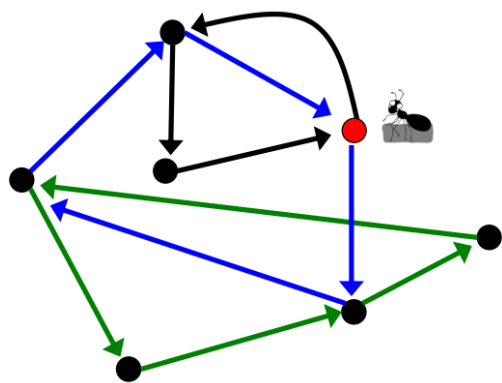
Slika 2.10: Mrav je postavljen u crveni čvor i odatle kreće obilazak povezanog balansiranog grafa.



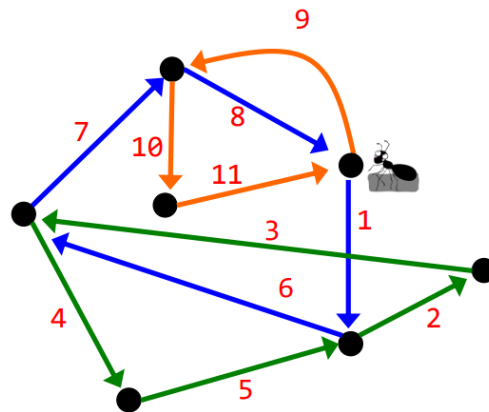
Slika 2.11: Mrav se zaglavio i pokušava ponovo iz drugog čvora koji pripada ciklusu i ima izlazne grane koje nisu posećene.

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 zaglavi (slika 2.12) ponovo bira novi početni čvor, obilazi pronađeni ciklus (koji i dalje nije Ojlerov) i tako sve dok ne uspe da obiđe 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

```

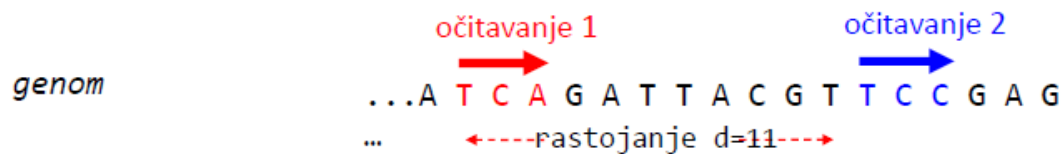
Ovaj algoritam radi u linearnom vremenu. Da bi se zaista postigla ta efikasnost, potrebne su efikasne strukture podataka za održavanje ciklusa koje mrav pronalazi kao i za liste neiskorišćenih grana za svaki čvor i lista čvorova u trenutnom ciklusu koji imaju neiskorišćene grane.

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 sečemo genom na fragmente iste dužine *InsertLength*. Zatim generišemo *parove očitavanja* – dva očitavanja sa krajeva svakog fragmenta na jednako, fiksiranoj udaljenosti. Pod *uparenim k-gramom* podrazumevamo par *k*-grama na fiksiranom rastojanju *d* u genomu. *Upareni k-gramski sastav*, u oznaci $PairedComposition_k(Text)$, 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.

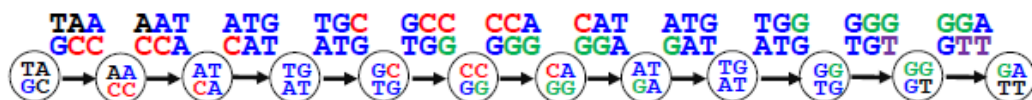
Dajmo jedan primer. Neka imamo nisku TAATGCCATGGGATGTT, i upareni 3-gram TAA i GCC. Upareni k -gramski sastav date niske prikazan je na slici 2.15.

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 $PairedComposition_k(Text)$ jednak kolekciji uparenih k -grama.

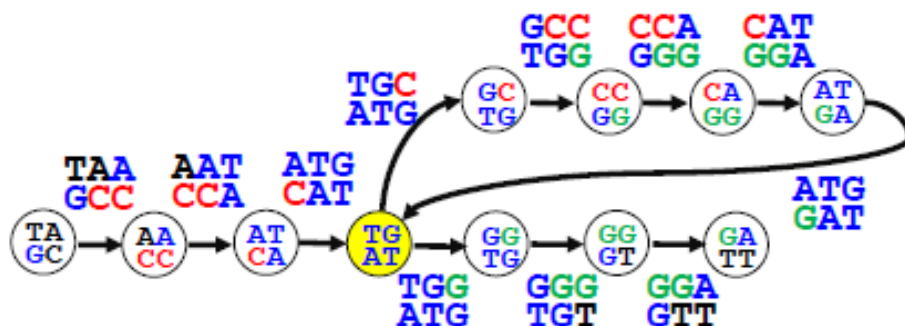
Kako konstruisati upareni De Brojnov graf na osnovu uparenog k -gramskog sastava? Postupak je sličan prethodnom slučaju, kada nismo imali parove. Pretpostavimo da je dat genom (niska *Genome*). Posmatrajmo genom kao putanju u grafu obeleženom na osnovu njegovog uparenog k -gramskog sastava (videti sliku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



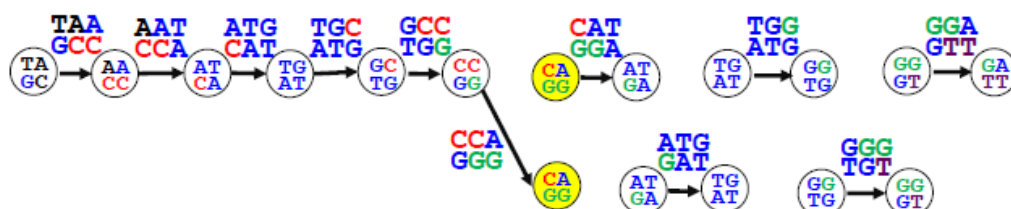
Slika 2.16: Graf koji odgovara uparenom 3-gramskom sastavu niske TAATGCCATGGGATGTT.

Potrebno je zalepiti 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 -gramskog sastava. Sada pretpostavimo da nije dat genom već samo upareni k -gramski 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.



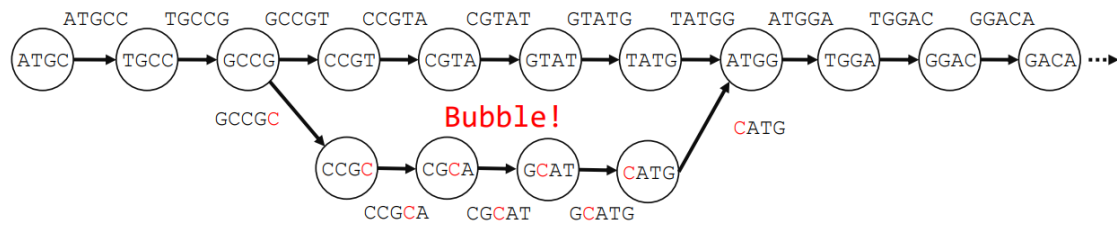
Slika 2.18: Konstrukcija uparenog De Brojnovog grafa na osnovu uparenih k -grama.

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.

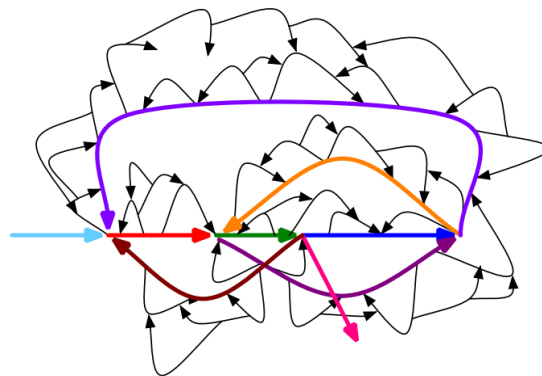
2.9 U realnosti

Ovde smo imali neke nerealne pretpostavke:

- Savršena pokrivenost genoma očitavanjima (svaki k -gram iz genoma je očitao). Očitavanja dužine 250 nukleotida dobijena *Illumina* tehnologijom predstavljaju samo mali deo 250-grama unutar genoma. Rešenje je u razbijanju dobijenih očitavanja na kraće k -game.
- Očitavanja ne sadrže greške. U ovom slučaju, ako bismo razbili na manje k -game, 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 zadanog cvora
46 def isolated_cycle(G, v):
47     cycle = []
48
49     (in_deg, out_deg) = degree(G, v)
50
51     while in_deg == 1 and out_deg == 1:
52         u = G[v][0]
53         cycle.append((v,u))
54         if cycle[0][0] == cycle[-1][1]:
55             return cycle
56
57         v = u
58         (in_deg, out_deg) = degree(G, v)
59
60     return None
61
62
63 # Pronalazenje maksimalnih nerazgranatih putanja u grafu
64 def maximal_non_branching_paths(G):
65     paths = []
66     visited = {}
67
68     for v in G:
69
70         (v_in_deg, v_out_deg) = degree(G, v)
71         if v_in_deg != 1 or v_out_deg != 1:
72
```

```

73         visited[v] = True
74
75         if v_out_deg > 0:
76
77             for w in G[v]:
78                 non_branching_path = [(v,w)]
79
80                 visited[w] = True
81                 (w_in_deg, w_out_deg) = degree(G, w)
82
83                 while w_in_deg == 1 and w_out_deg == 1:
84                     u = G[w][0]
85                     non_branching_path.append((w,u))
86                     w = u
87                     visited[w] = True
88                     (w_in_deg, w_out_deg) = degree(G, w)
89
90                 paths.append(non_branching_path)
91
92     for v in G:
93         if v not in visited:
94             c = isolated_cycle(G, v)
95             if c != None:
96                 paths.append(c)
97
98     return paths
99
100
101 # Konstruisanje DNK niske od dobijene putanje
102 def create_string_from_path(path):
103
104     dna_string = path[0][0]
105
106     for i in range(len(path)):
107         dna_string += path[i][1][-1]
108
109     return dna_string
110
111
112 def main():
113     dna_string = "AATCGTGACCTCAACT"
114     #           TCGTGAC
115     #           AATC
116     #           ACT
117     #           ACCT
118     #           AAC
119     #           TCAAC
120     k = 3
121     k_mers = string_to_k_mers(dna_string, k)
122     g = debruijn_graph_from_k_mers(k_mers)
123     paths = maximal_non_branching_paths(g)
124
125     print(paths)

```

```
126
127 if __name__ == "__main__":
128     main()
```

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 zadanog cvora
16 def isolated_cycle(G, v):
17     cycle = []
18
19     (in_deg, out_deg) = degree(G, v)
20
21     while in_deg == 1 and out_deg == 1:
22         u = G[v][0]
23         cycle.append((v,u))
24         if cycle[0][0] == cycle[-1][1]:
25             return cycle
26
27         v = u
28         (in_deg, out_deg) = degree(G, v)
29
30     return None
31
32 # Konstruisanje DNK niske od dobijene putanje
33 def create_string_from_path(path):
34
35     dna_string = path[0][0].replace("'", '')
36
37     for i in range(len(path)):
38         dna_string += path[i][1].replace("'", '')[-1]
39
40     return dna_string
41
42
43 # Pronalazenje cvorova od kojih postoje grane ka zadanom 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 zadanog cvora v
54 def outgoing(G, v):
55     return G[v]
56
57
58 # Pravljenje (u,v,w) "zaobilaznice" u zadanom 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 zadanog 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 Djlerovih ciklusa u zadanom 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)
```

```

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'], 'AC
126         ↪ ': ['CG'], 'GG': ['GA']}
127
128     print(all_eulerian_cycles(G))
129
130 if __name__ == "__main__":
131     main()

```

2.10.3 String Spelled By Gapped Patterns

```

1  # Sastavljanje DNK niske pomocu k-mera
2  def string_spelled_by_patterns(patterns, k):
3      dna_string = patterns[0][:-1]
4
5      for i in range(0, len(patterns)):
6          dna_string += patterns[i][-1]
7
8      return dna_string
9
10 # Sastavljanje DNK niske pomocu parova k-mera na udaljenosti d
11 def string_spelled_by_gapped_patterns(gapped_patterns, k, d):
12     first_patterns = [s[0] for s in gapped_patterns]
13     second_patterns = [s[1] for s in gapped_patterns]
14
15     prefix_string = string_spelled_by_patterns(first_patterns, k)
16     suffix_string = string_spelled_by_patterns(second_patterns, k)
17
18     print(prefix_string)
19     print(suffix_string)

```

```
20
21     for i in range(k+d, len(prefix_string)):
22         if prefix_string[i] != suffix_string[i-k-d]:
23             print('There is no string spelled by the gapped patterns')
24             return ''
25     return prefix_string + suffix_string[-k-d:]
26
27
28 def main():
29     gapped_patterns = [('CTG', 'CTG'), ('TGA', 'TGA'), ('GAC', 'GAC'), ('ACT', 'ACT')]
30
31     print(string_spelled_by_gapped_patterns(gapped_patterns, 3, 1))
32
33 if __name__ == "__main__":
34     main()
```


Literatura