1 Frequent Words

Funkcija ima dva parametra, `text` i `k`. Parametar `text` predstavlja nisku u kojoj tražimo podnisku dužine `k`. Povratna vrednost funkcije je skup podniski dužine `k` koje se najčešće pojavljuju.

Polazimo od praznog skupa `frequent_patterns` i praznog niza `count`. Polazeći od nulte pozicije određujemo broj pojavljivanja (funkcijom `pattern_count`) svake podniske dužine `k` i broj dodajemo u niz `count`. Zatim, određujemo najveći broj iz niza (ugrađenom funkcijom `max`). Na kraju, u skup `frequent_patterns` dodajemo sve podniske čiji je broj pojavljivanja jednak najvećem.

```
def frequent_words(text, k):
    frequent_patterns = set([])
    count = []

    for i in range(len(text) - k):
        pattern = text[i:i+k]
        count.append(pattern_count(text, pattern))

    max_count = max(count)

    for i in range(len(text) - k):
        if count[i] == max_count:
            frequent_patterns.add(text[i:i+k])

    return frequent_patterns
```

1.1 Pattern Count

Funkcija ima dva parametra, `text` i `pattern`. Prebrojava pojavljivanja zadate sekvence `pattern` u tekstu `text`.

```
def pattern_count(text, pattern):
    count = 0
    for i in range(len(text) - len(pattern)):
        if text[i:i+len(pattern)] == pattern:
            count += 1
    return count
```

1.1.1 Test primer

`text` = 'atgcgctagtcactagtgtcagttcgatgtcgat'
`k` = 3
`frequent_patterns` = ...

2 Faster Frequent Words

Funkcija ima dva parametra, `text` i `k`. Parametar `text` predstavlja nisku u kojoj tražimo podnisku dužine `k`. Povratna vrednost funkcije je par - skup podniski dužine `k` koje se najčešće pojavljuju i broj pojavljivanja sekvenci iz skupa u tekstu.

Polazi se od praznog skupa sekvenci. Prvo, određujemo niz frekvencija pojavljivanja svih podniski dužine `k` korišćenjem funkcije `computing_frequencies`. Indeks niza jedinstveno određuje nisku, i obrnuto (u tu svrhu koriste se funkcije `nuber_to_pattern` i `pattern_to_number`).

Zatim, određujemo najveću frekvenciju. Na kraju, prolazimo sve kombinacije niski nad azbukom $\{A, T, C, G\}$ i u skup dodajemo sekvence sa maksimalnom frekvencijom.

```
def faster_frequent_words(text, k):
    frequent_patterns = set([])

    frequency_array = computing_frequencies(text, k)

    max_count = max(frequency_array)

    for i in range(4**k):
        if frequency_array[i] == max_count:
            pattern = number_to_pattern(i, k)
            frequent_patterns.add(pattern)

    return (frequent_patterns, max_count)
```

2.1 Computing Frequencies

Funkcija ima dva parametra, `text` i `k`. Funkcija formira niz frekvencija pojavljivanja za sve moguće kombinacije niski nad azbukom $\{A, T, C, G\}$. Polazimo od niza nula dimenzije 4^k. Za svaku podnisku dužine `k` određujemo njen broj (funkcijom `pattern_to_number`) odnosno, indeks u nizu frekvencija, i odgovarajući element uvećamo za jedan.

```
def computing_frequencies(text, k):
    frequency_array = [0 for i in range(4**k)]

    for i in range(len(text) - k):
        pattern = text[i:i+k]
        j = pattern_to_number(pattern)
        frequency_array[j] += 1

    return frequency_array
```

2.2 Number To Pattern

Funkcija ima dva parametra, `n` - broj koji treba pretvoriti u sekvencu, i `k` - dužinu sekvence. Implementacija je rekurzivna. Izlazimo iz rekurzije kada je dužina sekvence jednaka 1, pri čemu vraćamo karakter koji odgovara trenutnoj vrednosti broja `n`. Računa se u osnovi 4. Prefiksni indeks predstavlja količnik broja `n` i broja 4. Određujemo karakter `c`, koji odgovara ostatku koji se dobija pri tom deljenju. Takođe, treba odrediti prefiksnu sekvencu koja odgovara prefiksnom indeksu, rekurzivnim pozivom, pri čemu je `k` umanjeno za 1. Vraćamo nisku koja se dobija nadovezivanjem karaktera `c` na prefiksnu sekvencu.

```
def number_to_pattern(n, k):
    if k == 1:
        return number_to_symbol(n)

    prefix_index = n // 4
    r = n % 4
    c = number_to_symbol(r)
    prefix_pattern = number_to_pattern(prefix_index, k - 1)
```

```
    return prefix_pattern + c
```

2.2.1 Pattern To Number

Funkcija ima jedan parametar, `pattern`, koji treba pretvoriti u broj. Implementacija je rekurzivna. Izlaz iz rekurzije je sekvenca dužine 0, kojoj odgovara broj 0. Broj se računa u osnovi 4, korišćenjem Hornerove sheme. Rekurzivno računamo broj prefiksa (podniska bez poslednjeg karaktera), množimo sa 4 i dodajemo broj koji odgovara poslednjem karakteru (korišćenjem funkcije `symbol_to_number`).

```
def pattern_to_number(pattern):
    if len(pattern) == 0:
        return 0

    last = pattern[-1:]
    prefix = pattern[:-1]

    return 4 * pattern_to_number(prefix) + symbol_to_number(last)
```

2.2.2 Symbol To Number

Funckija prima jedan karakter i vraća odgovarajući broj. Broj se čita iz mape koja preslikava karaktere A, T, C, G u brojeve 0, 1, 2, 3.

```
# Prevodjenje nukleotida u brojeve
def symbol_to_number(c):
    pairs = {
        'a' : 0,
        't' : 1,
        'c' : 2,
        'g' : 3
    }

    return pairs[c]
```

2.2.3 Number To Symbol

Funkcija prima jednu cifru i vraća odgovarajući karakter. Karakter se čita iz mape koja preslikava cifre 0, 1, 2, 3 u karatere A, T, C, G.

```
# Prevodjenje brojeva u nukleotide
def number_to_symbol(n):
    pairs = {
        0 : 'a',
        1 : 't',
        2 : 'c',
        3 : 'g'
    }

    return pairs[n]
```

3 Frequent Words With Mismatches

Funkcija ima tri parametra, `text`, `k` i `d` - broj dozvoljenih promašaja. Povratna vrednost je skup čestih sekvenci sa najviše `d` promašaja.

Polazimo od praznog skupa čestih sekvenci. Pravimo dva niza nula dimenzije 4^k, `close` - kanadidati za proveru i `frequency_array` - frekvencije kandidata. Za svaki uzorak dužine `k` određujemo susede - sekvence koje se od uzorka razlikuju na najviše `d` pozicija (korišćenjem funkcije `neighbors`). Za svakog suseda određujemo indeks i evidentiramo ga u nizu kandidata (niz `close`) - postavljamo mu vrednost na 1.

Prolazimo elemente niza `close` i za sve koji su evidentirani određujemo koja je sekvenca u pitanju, na osnovu indeksa (funkcijom `number_to_pattern`) i za njih se određuje broj pojavljivanja koji se pamti u nizu frekvencija (funkcijom `approximate_pattern_count`).

Zatim, određujemo najveću frekvenciju. Na kraju, u skup čestih sekvenci dodaje se svaka sekvenca čiji je frekvencija jednaka najvećoj.

```
def frequent_words_with_mismatches(text, k, d):
    frequent_patterns = set([])
    close = [0 for i in range(4**k)]
    frequency_array = [0 for i in range(4**k)]

    for i in range(len(text) - k):
        neighborhood = neighbors(text[i:i+k], d)

        for pattern in neighborhood:
            index = pattern_to_number(pattern)
            close[index] = 1

    for i in range(4**k):
        if close[i] == 1:
            pattern = number_to_pattern(i, k)
            frequency_array[i] = approximate_pattern_count(text,
                                                           pattern, d)
    max_count = max(frequency_array)

    for i in range(4**k):
        if frequency_array[i] == max_count:
            pattern = number_to_pattern(i, k)
            frequent_patterns.add(pattern)

    return frequent_patterns
```

3.1 Neighbors

Funkcija ima dva parametra, `pattern` i `d`. Povratna vrednost je skup `neighborhood`. Implementacija je rekurzivna. Prvo se proverava da li je `d` jednako 0. U tom slučaju vraćamo jednočlani skup koji sadrži samo `pattern`. To omogućava da funkciju koristimo i u slučaju kad ne želimo promašaje bez ikakvih modifikacija.

Izlazimo iz rekurzije kada je dužina uzorka jednaka 1. U tom slučaju vraćamo skup koji sadrži listu svih slova azbuke.

Pravimo skup `neighborhood` koji inicijalno sadrži praznu listu. Zatim, određujemo susede sufiksa, odnosno, sekvence bez prvog karaktera niske `pattern` i smeštamo u `suffix_neighbors`. Za svakog suseda sufiksa, koji se od sufiksa razlikuje na manje od `d` mesta, u `neighborhood` dodajemo 4 sekvence, po jedna za svako slovo azbuke, pri čemu se slovo nadovezuje na početak suseda.

Za susede koji se razlikuju na više od `d` pozicija, u `neighborhood` dodajemo suseda na čiji je početak nadovezan prvi karakter niske `pattern`, kako se razlika ne bi povećala i premašila dozvoljeni broj `d`. Kao mera za razliku koristi se Hamingovo rastojanje (funckija `hamming_distance`).

```
def neighbors(pattern, d):
    if d == 0:
        return set([pattern])

    if len(pattern) == 1:
        return set(['a', 't', 'c', 'g'])

    neighborhood = set([])

    suffix_neighbors = neighbors(pattern[1:], d)

    for text in suffix_neighbors:
        if hamming_distance(pattern[1:], text) < d:
            for x in ['a','t','c','g']:
                neighborhood.add(x + text)
        else:
            neighborhood.add(pattern[0] + text)

    return neighborhood
```

3.2 Approximate Pattern Count

Funckija ima tri parametra, `tekst`, `pattern` i `d`. Povratna vrednost je broj pojavljivanja podsekvenci u tekstu koje se od uzorka razlikuju na najvise `d` pozicija. Kao i u prethodnoj funkciji, koristi se Hamingovo rastojanje.

```
def approximate_pattern_count(text, pattern, d):
    count = 0

    for i in range(len(text) - len(pattern)):
        pattern_p = text[i:i+len(pattern)]

        if hamming_distance(pattern, pattern_p) <= d:
            count += 1

    return count
```

3.2.1 Hamming distance

Funkcija ima dva parametra, `text1` i `text2`. Povratna vrednost je brj pozicija na kojima se tekstovi razlikuju. Podrazumeva se da je dužina niski jednaka.

```
def hamming_distance(text1, text2):
    distance = 0

    for i in range(len(text1)):
        if text1[i] != text2[i]:
            distance += 1
    return distance
```

3.2.2 Test primer

`text` = 'tgactatcatcgtagtatcgatgtgcacacacgtgcgcgcgcgcgccctgtacatgatc'
`k` = 5
`d` = 2
`frequent_patterns` = ...

4 GC-Skew

Otvaramo fajl u FASTA formatu koji sadrži nukleotidnu sekvencu. Zanemarujemo prvi red datoteke. Dodatno, potrebno je da uklonimo beline, odnosno sve nove redove, kao i da pretvorimo slova iz velikih u mala. Nakon toga, računamo skew na osnovu nukleotida od milionitog do kraja (korišćenjem funkcije `calculate_skew`). Na kraju, crtamo skew (pomoću funkcije `draw_skew`).

```
def main():

    fajl = open('ecoli.fna', 'r')

    fajl.readline() # Zanemaruje se prvi red datoteke u FASTA formatu
    sadrzaj = ""

    sadrzaj = fajl.readlines()
    sadrzaj = "".join(sadrzaj)
    sadrzaj = sadrzaj.replace('\n', '')
    sadrzaj = sadrzaj.lower()

    skew = calculate_skew(sadrzaj[1000000:])
    draw_skew(skew)
```

4.1 Calculate Skew

Funkcija ima jedan parametar, `text`, koji predstavlja nukleotidnu sekvencu za koju računamo skew. Povratna vrednost je `skew`.

Skew je inicijalno niz nula, dimenzije datog teksta. Prolazimo sve karaktere teksta i u zavisnosti od nukleotida vršimo odgovarajuću akciju:

- ako je nukleotid `G` - prethodnu vrednost uvećavamo za jedan
- ako je nukleotid `C` - prethodnu vrednost smanjujemo za jedan
- u ostalim slučajevima ne menjamo prethodnu vrednost.

Dobijenu vrednost smeštamo u `skew` na odgovarajuću poziciju.

4.2 Draw Skew

Funkcija ima jedan parametar, `skew` - skew dijagram koji treba vizuelizovati. U tu svrhu, neophodno je uključiti pyplot iz biblioteke matplotlib.

Vrednosti na x-osi su brojevi iz intervala $[0, len(skew)]$. Pravimo `ax` koja sadrži subplot. U njoj iscrtavamo `plot` za `x` i `skew`. Možemo označiti ose (xlabel, ylabel) i postaviti naslov, pozivom funkcije `set` nad `ax`. Takođe, možemo dodati mrežu pozivom funkcije `grid` nad `ax`. Na kraju prikazujemo dijagram, pozivom funkcije `show` iz biblioteke pyplot.

```
def calculate_skew(text):
    skew = [0 for c in text]

    last = 0

    for i in range(0,len(text)):

        if text[i] == 'g':
            skew[i] = last + 1

        elif text[i] == 'c':
            skew[i] = last - 1

        else:
            skew[i] = last

        last = skew[i]

    return skew
```

```
import matplotlib.pyplot as plt

def draw_skew(skew):
    x = [i for i in range(len(skew))]

    ax = plt.subplot()
    ax.plot(x, skew)

    ax.set(xlabel='G-C', ylabel='Nucleotide',
        title='Genome GC Skew')
    ax.grid()

    plt.show()
```

5 Median String

Funkcija ima dva parametra, `dna` - niz niski koje čine nukleotidnu sekvencu, i `k` - željena dužina . Povratna vrednost je niska `median`.

Tražimo nisku dužine `k` koja je najmanje udaljena od svih iz `dna`. Uzimamo u obzir svih 4^k kombinacija i proveravamo koja najmanje udaljenda od svih. Za određivanje niske koristi se funkcija `number_to_pattern` (2.2). Zatim, za dobijenu nisku određujemo ukupno rastojanje od svih sekvenci iz `dna` (funkcijom `d`). Potrebno je još proveriti da li je to rastojanje trenutni minimum i, ako jeste, ažurirati minimalno rastojanje i nisku `meadian`.

```
def median_string(dna, k):
    distance = float('inf')
    median = ''

    for i in range(4**k):
        pattern = number_to_pattern(i, k)

        current_distance = d(pattern, dna)

        if distance > current_distance:
            distance = current_distance
            median = pattern

    return median
```

5.1 D

Funkcija ima dva parametra, `pattern` - kandidat za median, i `dna` - niz niski koje čine nukleotidnu sekvencu. Povratna vrednost je suma Hamingovih rastojanja između niske `pattern` i svih sekvenci iz `dna`.

Prolazimo jednu po jednu sekvencu iz `dna` i za svaku određujemo najmanje Hamingovo rastojanje u odnosu na `pattern`. Ideja je da u toj sekvenci tražimo podsekvencu, dužine niske `pattern` (u kodu, to je vrednost `k`), koja se najbolje poklapa sa niskom `pattern` odnosno, koja ima najmanje hamingovo rastojanje (3.2.1) u odnosu na nisku `pattern`. Nakon što je određeno najmanje rastojanje za jednu sekvencu, ono se dodaje na ukupan zbir rastojanja, što je povratna vrednost ove funkcije.

```
def d(pattern, dna):
    k = len(pattern)
    distance = 0

    for dna_string in dna:
        h_dist = float('inf')

        for i in range(len(dna_string) - k + 1):
            pattern_p = dna_string[i:i+k]
            dist = hamming_distance(pattern_p, pattern)

            if dist < h_dist:
                h_dist = dist

        distance += h_dist
    return distance
```

5.1.1 Test primer

`dna` = [
'GTAGATGTCATTAGCATGCAC',
'CCTAGCCACTCTGCCATGTCG',
'AACTCGTGCATTCTACGACTG',
'AAACTTTCCGGATCTTCATAC',
'CTACATCATCGAAGGCTACGC'
]
`k` = 4
`median` =

6 Greedy Motif Search

Funkcija ima tri parametra, `dna` - niz sekvenci, `k` - dužina motiva, i `t` - broj sekvenci u `dna`. Povratna vrednost je skup najboljih motiva.

Inicijalno, `best_motifs` sadrži prefikse svih niski iz `dna` dužine `k`. Pored najboljih motiva, pamtimo i trenutno nabolji (najmanji) skor, koji se računa korišćenjem funkcije `score`.

Polazimo od prve sekvence u nizu `dna` iz koje ćemo izdvajati podniske dužine `k`. Čuvamo skup motiva, a prvi motiv biće podniska dužine `k` u tekućoj iteraciji. Dakle, indeksiramo prvu nisku i u svakoj iteraciji izdvajamo podniske dužine `k`. Pored toga, u svakoj iteraciji iz preostalih `t` - 1 sekvenci izdvajamo podniske dužine `k` koje su najverovatnije (to određuje funkcija `most_probable_kmer`) u odnosu na matricu `profile`. Određujemo profil na osnovu motiva iz prethodnih iteracija (a dobijamo ga kao povratnu vrednost funkcije `profile`), a onda i najverovatniju podnisku. Najverovatniju podnisku dodajemo u skup motiva i on se koristi u narednoj iteraciji za pronalaženje sledećeg motiva. Kada su određeni svi motivi u tekućoj iteraciji, računa se trenutni skor i, ukoliko je bolji od trenutnog najboljeg, ažuriramo najbolji skor i motive.

```
def greedy_motif_search(dna, k, t):
    best_motifs = [dna_string[0:k] for dna_string in dna]
    best_score = score(best_motifs, k)
    first_string = dna[0]

    for i in range(len(first_string) - k):
        motifs = []
        motifs.append(first_string[i:i+k])

        for j in range(1, t):
            profile = profile_from_motifs(motifs, k, j)
            motifs.append(most_probable_k_mer(dna[j], profile, k))

        current_score = score(motifs, k)

        if current_score < best_score:
            best_motifs = copy.deepcopy(motifs)
            best_score = current_score

    return best_motifs
```

6.1 Score

Funkcija ima dva parametra, `motifs` - skup motiva, i `k` - dužina svakog od motiva. Povratna vrednost je ukupan skor. Da se podsetimo, skor predstavlja ukupan broj nepopularnih nukleotida

po kolonama.

Promenljiva `t` pamti broj motiva, a povratna vrednost biće smeštena u promenljivu `total_score`. Dakle, krećemo se po kolonama, tako da će granica za izvršavanje petlje biti vrednost `k`, što je dužina svakog od motiva. Za svaku kolonu pamtimo koliko se koji nukleotid pojavio u nizu `counts` dimenzije 4. Zatim prolazimo redom karaktere motiva iz odgovarajuće kolone i ažuriramo niz. Koristimo funkciju `symbol_to_number` (2.2.2) za dobijanje indeksa, a potom sledi jednostavna inkrementacija elementa niza na odgovarajućoj poziciji. Kada su obrađeni svi motivi, potrebno je odrediti indeks maksimuma. Na kraju, uvećati `total_score` za razliku ukupnog broja nukleotida i broj pojavljivanja najpopularnijeg nukleotida.

```
# Izracunavanje ukupnog skora za skup motiva
def score(motifs, k):
    t = len(motifs)

    total_score = 0

    for j in range(k):

        counts = [0, 0, 0, 0]

        for i in range(t):
            c = motifs[i][j]
            index = symbol_to_number(c)
            counts[index] += 1

        max_index = 0

        for i in range(1,4):
            if counts[i] > counts[max_index]:
                max_index = i

        total_score += t - counts[max_index]

    return total_score
```

6.2 Profile From Motifs

Funkcija ima dva parametra, `motifs` - skup motiva, i `k` - dužina svakog motiva. Povratna vrednost funkcije je matrica profila dimenzija $4 \times k$. Podsetimo se da vrednosti u matrici profila predstavljaju verovatnoću da se pojavi odgovarajući nukleotid na odgovarajućem mestu (ako bacamo četvorostranu pristrasnu kockicu, ove vrednosti bi bile verovatnoće da padne odgovarajući nukleotid). U ovoj implementaciji primenjeno je i Laplasovo pravilo, kako bi se izbegla vreovatnoća jednaka nuli.

Zbog Laplasovog pravila, matrica profila inicijalno je popunjena jedinicama. U prvom prolazu dvostruke petlje, koja se prvo kreće po kolonama a onda po vrstama, određujemo indeks nukleotida, a onda uvećamo odgovarajuće polje u matrici profila. U drugom prolazu dvostruke petlje, delimo svaki element matrice brojem motiva uvećanog za 2.

```
def profile_from_motifs(motifs, k, t):
    profile = [[1 for i in range(k)] for x in range(4)]

    for j in range(k):
        for i in range(t):
```

```
            index = symbol_to_number(motifs[i][j])
            profile[index][j] += 1

    for j in range(k):
        for i in range(4):
            profile[i][j] /= (t+2)

    return profile
```

6.3 Most Probable Kmer

Funkcija ima tri parametra, `dna_string`, `profile` i `k`. Povratna vrednost funkcije je najverovatnija podniska niske `dna_string` dužine `k` u odnosu na amtricu profila.

Iniciijalno, `best_kmer` je prazna niska, a `best_probability` je negativna vrednost. Prolazimo sve podniske dužine `k` i za svaku računamo verovatnoću (funkcijom `probability`). Ukoliko je ta verovatnoća veća od trenutno najbolje, ažuriramo najbolju podnisku i verovatnoću.

```
def most_probable_k_mer(dna_string, profile, k):

    best_k_mer = ''
    best_probability = -1

    for i in range(len(dna_string) - k + 1):
        pattern = dna_string[i:i+k]
        pattern_prob = probability(pattern, profile)

        if pattern_prob > best_probability:
            best_probability = pattern_prob
            best_k_mer = pattern

    return best_k_mer
```

6.3.1 Probability

Funkcija ima dva parametra, `pattern` i `profile`. Funkcija vraća verovatnoću pojave `pattern` sekvence u odnosu na zadati profil. Ukupna verovanoća dobija se kao proizvod odgovarajućih vrednosti iz matrice profil.

Inicijalno, verovatnoća je jednaka 1. Za svaki karakter iz sekvence `pattern`, određujemo njegov indeks funkcijom `symbol_to_number` (2.2.2). Zatim, trenutnu verovatnoću množimo vrednošću iz matrice profil koja odgovara datom karakteru i poziciji na kojoj se nalazi u niski.

```
def probability(pattern, profile):
    prob = 1

    for j in range(len(pattern)):
        c = pattern[j]
        index = symbol_to_number(c)

        prob *= profile[index][j]

    return prob
```

6.3.2 Test primer

```
dna = [
'GTAGATGTCATTAGCATGCAC',
'CCTAGCCACTCTGCCATGTCG',
'AACTCGTGCATTCTACGACTG',
'AAACTTTCCGGATCTTCATAC',
'CTACATCATCGAAGGCTACGC'
]
k = 4
t = len(dna)
best_motifs =
```

7 Randomized Motif Search

Funkcija ima tri parametra, `dna`, `k` i `t`. Povratna vrednost je skup najboljih motiva.

Polazimo od slučajno odabranih motiva (dobijamo ih funkcijom `random_k_mers`). Pamtimo trenutno najbolje motive (na početku su to ovi slučajno odabrani) i njihov skor (koji računamo korišćenjem funkcije `score`, 6.1).

Vrtimo petlju dok se skor popravlja. Prvo, formiramo profil od tekućih motiva (funkcijom `profile_from_motifs`, 6.2), a onda obrnuto, određujemo motive od dobijenog profila (funkcijom `motifs_from_profile`). Računamo skor novih motiva, i ako je skor manji od najboljeg, ažuriramo najbolje motive i njihov skor. Ukoliko je novi skor lošiji, tada prekidamo petlju i vraćamo najbolje motive.

```
import copy

def randomized_motif_search(dna, k, t):

    motifs = random_k_mers(dna, k, t)
    best_motifs = copy.deepcopy(motifs)
    best_score = score(best_motifs, k)

    while True:
        profile = profile_from_motifs(motifs, k, t)
        motifs = motifs_from_profile(profile, dna)
        current_score = score(motifs, k)

        if current_score < best_score:
            best_score = current_score
            best_motifs = copy.deepcopy(motifs)
        else:
            return best_motifs
```

7.1 Random Kmers

Funkcija ima tri parametra, `dna`, `k` i `t`. Povratna vrednost je skup slučajno odabranih motiva.

Skup motiva na početku je prazan. Za svaku sekvencu iz `dna` generišemo jedan sluačajan broj koji predstavlja poziciju od koje počinje motiv, a biće dužine `k`. Odabranu podsekvencu dodajemo u skup motiva.

```
def random_k_mers(dna, k, t):
    k_mers = []
```

```
    for i in range(t):
        start = random.randrange(0, len(dna[i]) - k + 1)
        dna_string = dna[i]
        k_mers.append(dna_string[start:start+k])

    return k_mers
```

7.2 Motifs From Profile

Funkcija ima dva parametra, `profile` i `dna`. Povratna vrednost je skup motiva.

Polazi se od praznog skupa motiva. Za svaku sekvencu iz `dna` određujemo najverovatniju podnisku (korišćenjem funkcije `most_probable_kmer`, 6.3) i dodajemo je u skup motiva.

```
def motifs_from_profile(profile, dna):
    motifs = []
    k = len(profile[0])

    for dna_string in dna:
        motifs.append(most_probable_k_mer(dna_string, profile, k))

    return motifs
```

7.2.1 Test primer

`dna` = [
'GTAGATGTCATTAGCATGCAC',
'CCTAGCCACTCTGCCATGTCG',
'AACTCGTGCATTCTACGACTG',
'AAACTTTCCGGATCTTCATAC',
'CTACATCATCGAAGGCTACGC'
]
`k` = 4
`t` = len(dna)
`best_motifs` =

8 Gibbs Sampler

Funkcija ima četiri parametra, `dna`, `k`, `t` i `N` - broj iteracija. Povratna vrednost je skup najboljih motiva.

Polazimo od slučajno odabranih motiva (dobijamo ih funkcijom `random_k_mers`). Pamtimo trenutno najbolje motive (na početku su to ovi slučajno odabrani) i njihov skor (koji računamo korišćenjem funkcije `score`, 6.1).

Funkcija se izvršava u `N` iteracija, a u svakoj prvo biramo slučajan broj `i` iz skupa $[0, t)$ (to je slučajno odabrani motiv koji brišemo). Koristimo pomoćnu promenljivu `selected_motifs` u koju kopiramo elemente iz `motifs`, a onda brišemo onaj na poziciji `i`, koja je slučano odabrana. Zatim, pravimo matricu profila (`profile_from_motifs`, 6.2). Na osnovu dobijenog profila, određujemo najverovatniji motiv (funkcijom `most_probable_kmer`, 6.3) i smeštamo u `motifs` na prehodno odabranu poziciju `i`, a brišemo ceo `selected_motifs`.

Nakon toga, određuje se skor novog skupa motiva i, ukoliko je manji, ažuriramo najbolje motive i njihov skor.

```
# Pronalazenje skupa motiva nakon N iteracija koriscenjem Gibs sampler-a
def gibbs_sampler(dna, k, t, N):
    motifs = random_k_mers(dna, k, t)
    best_motifs = copy.deepcopy(motifs)
    best_score = score(best_motifs, k)

    for j in range(N):
        i = random.randrange(0,t)

        selected_motifs = copy.deepcopy(motifs)
        del selected_motifs[i]

        profile = profile_from_motifs(selected_motifs, k, t-1)

        motifs[i] = most_probable_k_mer(dna[i], profile, k)
        del selected_motifs

        current_score = score(motifs, k)

        if current_score < best_score:
            best_motifs = copy.deepcopy(motifs)
            best_score = current_score

    return best_motifs
```

8.0.1 Test primer

```
dna = [
'GTAGATGTCATTAGCATGCAC',
'CCTAGCCACTCTGCCATGTCG',
'AACTCGTGCATTCTACGACTG',
'AAACTTTCCGGATCTTCATAC',
'CTACATCATCGAAGGCTACGC'
]
k = 4
t = len(dna)
N = 500
best_motifs =
```

9 Maximal Non Branching Path

Funkcija ima jedan parametar, `G` - de Bruijn graf. Funkcija vraća maksimalne nerazgranate putanje u grafu.

Putanja je inicijalno prazna lista, a dodatno održavamo mapu posećenih čvorova, koja je inicijalno prazna.

Za svaki čvor `v` u grafu `G` računamo ulazni i izlazni stepen (funkcijom `degree`). Ukoliko je jedan od ta dva različit od 1, obeležavamo da je čvor posećen. Dodatno, ukoliko je izlazni stepen veći od 0, želimo da obiđemo sve susede. Za svaki čvor `w` iz liste suseda čvora `v`, pravimo putanju koja na početku sadrži samo granu (v, w). Zatim, obeležimo da je čvor `w` posećen i računamo njegov ulazni izlazni stepen.

Pošto nam je potreban nerazgranati put, pratimo čvorove koji imaju tačno jednu ulaznu i jednu izlaznu granu. Tako, dokle god su i ulazni i izlazni stepeni čvora `w` jednaki 1, određujemo čvor `u`, čvor koji se nalazi na drugom kraju izlazne grane iz čvora `w`. Tu granu treba dodati u trenutni put. Zatim, čvor `w` dobija vrednost čvora `u` i potrebno je ponovo označiti čvor `w` kao posećen i ažurirate stepene. Kada nema više čvorova koji ispunjavaju uslov, u listu putanja dodajemo trenutni put.

Još jednom prolazimo sve čvorove grafa `G`, i za svaki čvor `v` koji nije posećen, odredićemo izlovoani ciklus (funkcijom `isloated_cycle`). Ukoliko takav ciklus postoji, dodajemo ga u listu putanja.

```
# Pronalazenje maksimalnih nerazgranatih putanja u grafu
def maximal_non_branching_paths(G):
    paths = []
    visited = {}

    for v in G:
        (v_in_deg, v_out_deg) = degree(G, v)
        if v_in_deg != 1 or v_out_deg != 1:
            visited[v] = True

            if v_out_deg > 0:
                for w in G[v]:
                    non_branching_path = [(v,w)]
                    visited[w] = True
                    (w_in_deg, w_out_deg) = degree(G, w)

                    while w_in_deg == 1 and w_out_deg == 1:
                        u = G[w][0]
                        non_branching_path.append((w,u))
                        w = u
                        visited[w] = True
                        (w_in_deg, w_out_deg) = degree(G, w)

                    paths.append(non_branching_path)

    for v in G:
        if v not in visited:
            c = isolated_cycle(G, v)
            if c != None:
                paths.append(c)

    return paths
```

9.1 Degree

Funkcija ima dva parametra, `G` - de Bruijn graf, i `v` - čvor čiji ulazni i izlazni stepen treba odrediti. Povratna vrednost je par ulazni i izlazni stepen čvora `v`.

Pošto je graf predstavljen kao mapa, koja preslikava čvorove u listu čvorova sa kojim je povezan, izlazni stepen biće dužina liste čvora `v` u mapi `G`. Ulazni stepen je broj čvorova koji u svojoj listi sadrže čvor `v`.

```
def degree(G, v):
    out_deg = len(G[v])
    in_deg = 0

    for u in G:
        if v in G[u]:
            in_deg += 1

    return (in_deg, out_deg)
```

9.2 Isolated Cycle

Funkcija ima dva parametra, `G` - de Bruijn graf, i `v` - izolovani čvor za koji želimo da nađemo ciklus. Povratna vrednost je ciklus, ako postoji, a u suprotnom `None`.

Ciklus je inicijalno prazna lista grana (odnoso, parova ulaznih i izlaznih čvora). Pošto tražimo nerazgranate puteve, i ovde je bitno da svi čvorovi imaju ulazne i izlazne stepene jednake 1. Tako da, prvi korak je određivanje stepena čvora `v` (funkcijom `degree`, 9.1). Petlja se vrti dok su ulazni i izlazni stepen jednaki 1. Biramo čvor `u` koji se nalazi na drugom kraju jedine izlazne grane čvora `v` i dodajemo tu granu u ciklus.

Ukoliko su ulazni čvor prve grane i izlazni čvor poslednje grane ciklusa jednaki, znači da smo napravili ciklus i, pritom, obišli sve čvorove u toj komponenti povezanosti grafa `G`, pa vraćamo ciklus. U suprotnom, čvor `v` dobija vrednost čvora `u`. Ponovo računamo stepen čvora `v`. Ukoliko se naiđe na čvor koji ne ispunjava uslove petlje, vratiti `None`.

```
def isolated_cycle(G, v):
        cycle = []

        (in_deg, out_deg) = degree(G, v)

        while in_deg == 1 and out_deg == 1:
                u = G[v][0]
                cycle.append((v,u))
                if cycle[0][0] == cycle[-1][1]:
                        return cycle

                v = u
                (in_deg, out_deg) = degree(G, v)

        return None
```

U nastavku su funkcije neophodne da bismo došli do reprezentacije niske u obliku de Bruijn grafa.

9.3 String To Kmers

Funkcija ima dva parametra, `dna_string` - nisku koju želimo da rasparčamo da delove, i `k` - dužina delova.

Polazimo od prazne liste k-grama. Polazeći od pozicije 0, uzimamo podniske dužine `k` i smeštamo u listu.

```
def string_to_k_mers(dna_string, k):
        k_mers = []

        for i in range(len(dna_string) - (k-1)):
                k_mer = dna_string[i:i+k]
                k_mers.append(k_mer)

        return k_mers
```

9.4 De Bruijn

Funkcija ima jedan parametar, `kmers` - lista k-grama. Povratna vrednost je graf predstavljen mapom.

Prolazimo listu, element po element. Za svaki k-gram izdvajamo prefiks `u`, bez poslednjeg karaktera i sufiks `v`, bez prvog karaktera. Ukoliko se `u` nalazi u grafu `G`, a čvor `v` nije u njegovoj listi, dodajemo čvor `v` u listu čvora `u`. Ako se `u` ne nalazi u grafu, onda ga dodajemo u graf, a lista inicijlano sadrži samo `v`. Ako čvor `v` nije u grafu, dodajemo ga sa praznom listom.

```
def debruijn_graph_from_k_mers(k_mers):
    G = {}

    for k_mer in k_mers:
        u = k_mer[:-1]
        v = k_mer[1:]

        if u in G:
            if v not in G[u]:
                G[u].append(v)
        else:
            G[u] = [v]

        if v not in G:
            G[v] = []

    return G
```

9.4.1 Test primer

`dna_string` = "AATCGTGACCTCAACT"
`k` = 3
`k_mers` = string_to_k_mers(dna_string, k)
`g` = debruijn_graph_from_k_mers(k_mers)
`paths` =

10 All Euler Cycles

Funkcija prima jedan parametra, graf `G`. Povratna vrednost je lista ciklusa.

Lista ciklusa na početku je prazna. Koristimo pomoćnu promenljivu, `all_graphs`, koja čuva kopiju grafa `G` u strukturi *deque*. Petlja se vrti dok ima grafova u `all_graphs`, odnosno, dok je njegova dužina veća od nula.

Izdvajamo jedan graf, `G_p` pozivom funkcije `popleft()`. U promenljivu `v_p` želimo da smestimo čvor sa ulaznim stepenom većim od jedan u grafu `G_p`. Inicijalizujemo ga sa `None` i onda pokušavamo da ga pronađemo.

Prolazimo sve čvorove grafa `G_p` i računamo njihov ulazni i izlazni stepen (funkcija `degree`, 9.1). Prvo čvor na koji naiđemo, sa ulaznim stepenom većim od 1, dodeljujemo promenljivoj `v_p` i prekidamo potragu.

Ukoliko smo našli takav čvor, odnosno, ukoliko njegova vrednost nije jednaka `None`, želimo da napravimo jednostavniji (u, v, w) bajpas graf. Da se podsetimo, potrebno je da iz grafa uklonimo grane (u, v) i (v, w) i da dodamo novi čvor x sa granama (u, x) i (x, v).

Prolazimo sve čvorove `u` od kojih postoje grane ka čvoru `v_p` u grafu `G_p`, i sve čvorove `w` do kojih postoji grana od čvora `v_p` u grafu `G_p`. Zatim, pravimo bajpas graf (funkcijom `bypass`). Ukoliko je novi graf povezan (što možemo proveriti funkcijom `is_connected`), dodajemo njegovu kopiju u `all_graphs`.

Ukoliko nismo našli odgovarajući čvor `v_p`, onda prolazimo čvorove `k` grafa `G_p` i određujemo isolovani ciklus iz svakog (funkcija `isloated_cycle`, 9.2). Ukoliko takav ciklus postoji, želimo da napravimo nisku od ciklusa (funkcijom `create_string_from_path`) i da je dodamo u listu ciklusa, ukoliko se već ne nalazi tamo.

```
def all_eulerian_cycles(G):
    all_graphs = deque([copy.deepcopy(G)])
    cycles = []

    while len(all_graphs) > 0:
        G_p = all_graphs.popleft()
        v_p = None
        for v in G_p:
            (in_deg, out_deg) = degree(G_p, v)

            if in_deg > 1:
                v_p = v
                break

        if v_p != None:
            for u in incoming(G_p, v_p):
                for w in outgoing(G_p, v_p):
                    new_graph = bypass(G_p, u, v, w)
                    if is_connected(new_graph):
                        all_graphs.append(copy.deepcopy(new_graph))
        else:
            for k in G_p:
                cycle = isolated_cycle(G_p, k)
                if cycle != None:
                    path = create_string_from_path(cycle)
                    if path not in cycles:
                        cycles.append(path);

    return cycles
```

10.1 Incoming

Funkcija ima dva parametra, graf `G` i čvor `v`. Funkcija vraća listu čvorova grafa `G` od kojih postoji grana do čvora `v`.

Dovoljno je da jednom petljom prođemo sve čvorove grafa `G` i sve koji sadrže čvor `v` u svojoj listi dodamo u listu.

```
def incoming(G, v):
    in_list = []

    for u in G:
        if v in G[u]:
            in_list.append(u)

    return in_list
```

10.2 Outgoing

Funkcija ima dva parametra, graf `G` i čvor `v`. Funkcija vraća listu čvorova do kojih postoje grane iz čvora `v` u grafu `G`, odnosno, vraća listu čvora `v`.

```
def outgoing(G, v):
    return G[v]
```

10.3 Bypass

Funkcija ima četiri parametra, graf `G` i čvorove `u`, `v` i `w`. Povratna vrednost je novi graf sa izmenjenim granama. Još jednom, treba izbaciti grane (u, v) i (v, w), dodati novi čvor x (u kodu je `v'`, kako bi implementacija funkcije za pravljenje niske bila olakšana) povezati ga sa čvorom `u` i čvorom `w`.

Prvo, kopiramo graf `G` u `G_p`, koji ćemo dalje menjati. Zatim, iz liste čvora `u` brišemo čvor `v`, a iz liste čvora `v` brišemo čvor `w`. Onda dodajemo novi čvor, `v'` u listu čvora `u`, a lista čvora `v'` sadržaće samo čvor `w`. Na kraju vraćamo izmenjeni graf.

```
def bypass(G, u, v, w):
    G_p = copy.deepcopy(G)
    G_p[u].remove(v)
    G_p[v].remove(w)
    G_p[u].append(v+"'")   #v'
    G_p[v+"'"] = [w]
    return G_p
```

10.4 Is Connected

Funkcija ima jedan parametar, graf `G`. Povratna vrednost je tipa boolean.

Održavamo mapu posećenosti. Za svaki čvor `v` u grafu `G` pozivamo pomoćnu proceduru koja će izvršiti obilazak grafa u dubinu (`DFS`) počevši iz čvora `v` nakon čega prekidamo petlju.

Za svaki čvor vu grafu `G` proveravamo da li se nalazi u mapi posećenih čvorova. Ukoliko bar jedan čvor nije u mapi, graf nije povezan i vraćamo `False`. Inače, vraćamo `True`.

```
def is_connected(G):

    visited = {};
```

```
    for v in G:
        DFS(G,v,visited)
        break

    for v in G:
        if v not in visited:
            return False

    return True
```

10.5 Create String From Path

Funkcija ima jedan parametar, `path` - putanja na osnovu koje pravimo nisku.

Niska na početku sadrži karaktere ulaznog čvora prve grane. Ostale karaktere dobijamo proslaskom kroz sve grane na putanji, i izdvajanjem poslednjeg karaktera izlaznog čvora te grane.

```
def create_string_from_path(path):

    dna_string = path[0][0].replace("'",'')

    for i in range(len(path)):
        dna_string += path[i][1].replace("'",'')[-1]

    return dna_string
```

10.5.1 DFS

Funkcija ima tri parametra, graf `G`, čvor `v` i mapu posećenosti `visited`. Nema povratne vrednosti.

Čvor `v`, iz kog kreće obilazak grafa, obeležimo da je posećen. Zatim, prolazimo sve čvorove `w`, koji se nalaze u listi čvora `v`. Za svaki čvor koji do tad nije posećen, pozivamo istu proceduru.

```
def DFS(G, v, visited):
    visited[v] = True

    for w in G[v]:
        if w not in visited:
            DFS(G, w, visited)
```

10.5.2 Test primer

```
G = {
   'AT' : ['TC'],
   'TC' : ['CG'],
   'CG': ['GA','GG'],
   'GA':['AT','AC'],
   'AC':['CG'],
   'GG':['GA']
}
cycles =
```

11 String Spelled By Gapped Patterns

Funkcija ima tri parametra, `gapped_patterns` - skup parova k-grama, `k` i `d` - rastojanje dva susedna para k-grama. Povratna vrednost je niska sačinjena od zadatih k-grama.

Razdvajamo parove u posebne liste, `first_patterns` i `second_patterns`. Zatim, formiramo prefiksne (od `first_patterns`) i sufiksne (od `second_patterns`) niske (funkcijom `string_spelled_by_patterns`).

Prefiksna i sufiksna niska imaju prefiks, odnoso, sufiks dužine `k+d`, a ostatak niski treba da im se poklopi. Tako da, prolazimo ove niske karakter po karakter. Prefiksu prolazimo počevši od pozicije `k+d` do kraja, a sufiksnu prolazimo od 0 do `len(prefiks_string)-k-d`. Ukoliko naiđemo na nepoklapanje nukleotida na nekoj poziciji, ispisujemo odgovarajuću poruku i vraćamo praznu nisku. Ako su svi nukleotidi u redu, vraćamo nisku koja se dobije nadovezivanjem sufiksa iz sufiksne niske na prefiksnu nisku.

```
# Sastavljanje DNK niske pomocu parova k-mera na udaljenosti d
def string_spelled_by_gapped_patterns(gapped_patterns, k, d):
    first_patterns = [s[0] for s in gapped_patterns]
    second_patterns = [s[1] for s in gapped_patterns]

    prefix_string = string_spelled_by_patterns(first_patterns, k)
    suffix_string = string_spelled_by_patterns(second_patterns, k)

    print(prefix_string)
    print(suffix_string)

    for i in range(k+d, len(prefix_string)):
        if prefix_string[i] != suffix_string[i-k-d]:
            print('There is no string spelled by the gapped patterns')
            return ''
    return prefix_string + suffix_string[-k-d:]
```

11.1 String Spelled By Patterns

Funkcija ima dva parametra, `patterns` - niske koje treba spojiti, i `k` - dužina svake niske. Povratna vrednost je DNK niska sastavljena od datih k-grama.

Početna vrednost niske `dna_string` je k-gram bez poslednjeg karaktera. Prolaskom svih k-grama iz liste `patterns`, gradimo nisku nadovezivanjem poslednjeg karaktera iz svakog.

```
# Sastavljanje DNK niske pomocu k-mera
def string_spelled_by_patterns(patterns, k):
    dna_string = patterns[0][:-1]

    for i in range(0, len(patterns)):
        dna_string += patterns[i][-1]

    return dna_string
```

11.1.1 Test primer

`gapped_patterns` = [('CTG','CTG'),('TGA','TGA'),('GAC','GAC'),('ACT','ACT')]
`k` = 3
`d` = 1
`string` =