CSE4065
Introduction to Computational Genomics

Assignment II

Aybüke Özkan - 150115005
Celal Bayrak-150114044

1. Introduction

In this assignment, we wrote two algorithms for finding the motifs in DNA strings, which are Randomized Motif Search and Gibbs Sampler.

At the beginning of Randomized Motif Search algorithm, we created our input file. Input file contains 10 DNA strings consist of 500 bases each. We inserted 10-mers with 4 mutations in different positions to each string.

Because the input file is created in Randomized Motif Search algorithm, RMS should run first, then Gibbs Sampler algorithm will use that input file when it runs.

2. Randomized Motif Search

Firstly, we created input file. **line_generator()** and **txt_generator()** functions generates input.txt file which includes 10 DNA strings and 10 mutated motifs. **k_mer_generator()** function generates a k-mer randomly.

```
import random
import collections
nucleotids=["A","T","G","C"]
#generates line of text, by using mutated k-mer. It inserts k-mer to random position of line.
def line_generator(mutated_k_mer):
    global nucleotids
    line=""
    for i in range(0,490):
        rand=random.randrange(4)
        line=line+nucleotids[rand]
    rand2=random.randrange(0,490)
    line=line[0:rand2]+mutated_k_mer+line[rand2:]
    return line

#generates input.txt file by using lines, returned from line_generator() function.
def txt_generator(line_num,mutated_k_mers):
    lines=[]
    f = open("input.txt", "w")
    for i in range(0,line_num):
        line=line_generator(mutated_k_mers[i])
        lines.append(line)
    for l in lines:
        f.write("%s\n" %l)

#generates k-mer randomly.
def k_mer_generator(k):
    global nucleotids
    k_mer=""
    for i in range(0,k):
        rand=random.randrange(4)
        k_mer=k_mer+nucleotids[rand]
    return k_mer
```

k_mer_mutator() function generates 10 mutated k-mer from given k-mer. Each generated k-mer has 4 mutations.

```
def k_mer_mutator(k_mer):
    global nucleotids
    mutateds=[]
    for i in range(0,10):
        rands1=[]
        while len(rands1)!=4:
            rand1=random.randrange(10)
            if rand1 not in rands1:
                rands1.append(rand1)
    for i in range(0,10):
        temp = k_mer
        for rand in rands1:
            rand2=random.randrange(4)
            temp_arr=list(temp)
            if temp_arr[rand] != nucleotids[rand2]: #checking process which mentioned above.
                temp_arr[rand]=nucleotids[rand2]
            elif rand2 == 3:
                temp_arr[rand]=nucleotids[rand2 - 1] # changing the mutated nucleotide.
            else:
                temp_arr[rand]=nucleotids[rand2 +1]  # changing the mutated nucleotide.
            temp="".join(temp_arr)
        mutateds.append(temp)
    return mutateds
```

randomized_motif_search() function applies Randomized Motif Search algorithm to given lines of DNA strings and given k value of k-mer. The comments are include the details of implementation.

```
#randomized motif search algorithm. Inputs are lines of .txt file and k of k-mer.
def randomized_motif_search(lines,k):
    motifs=[]
    random_indexes=[]
    #randomly generating the initial 10 motifs. 1 motif from each line.
    for i in range(0,10):
        rand=random.randrange(491)
        random_indexes.append(rand)
        motif=lines[i][rand:rand+k]
        motifs.append(motif)
    prev_score=1000     #initial unreachable score.
    while True:         #infinite loop. if score does not improve in any iteration, loop ends.
        counters=[]
        score=0
        for i in range(0,k):    #these nested loops take the columns of motif matrix.
            arr=[]
            for motif in motifs:
                arr.append(motif[i]) # append the columns to a list.
            counter=collections.Counter(arr) # for each column counter object counts each element of column
                                             #and keeps those variables in dictionary format.
            counters.append(counter)    # for each column there is 1 counter object.
            score=score+(10-counter.most_common(1)[0][1]) #score calculation. The score is summation of (element number of each column -
        if score < prev_score: # if score improves continue the loop.
            prev_score=score   # keeps the previous score.
            motifs=[]
            for line in lines:  #these nested loops for probability calculation of each k-mer in .txt file.
                max_prob=0      #initial maximum probability of each line
                best_word=""    #initial best k-mer of each line
                for index in range(0,500-k):
                    word=line[index:index+k] #each k-mer of line
                    probability=1           #initial probability is 1. Because 1 is ineffective element for multiplication
                    for j in range(0,k):
                        letter=word[j]      #each letter of k-mer
                        frequency=float(counters[j][letter])/float(k) #frequency of each letter of k-mer in profile matrix.
                        probability=frequency*probability        #probability is multiplication of each letters frequency
                    if probability > max_prob:        #for each line max_prob keeps the highest probability
                        max_prob=probability
                        best_word=word               #for each line best_word keeps the best k-mer
                motifs.append(best_word)            #motifs keeps the best k-mer of each line and loop returns back to top.
        else:
            return motifs                # if score does not improve function returns the best motifs.
```

consensus() function returns the consensus string of given motifs.

```
#returns the consensus of motifs
def consensus(motifs,k):
    counters=[]
    consensus=""
    for i in range(0,k):
        arr=[]
        for motif in motifs:
            arr.append(motif[i])
        counter=collections.Counter(arr)
        counters.append(counter)
    for counter in counters:
        consensus=consensus+counter.most_common(1)[0][0]
    return consensus
```

3. Gibbs Sampler

motif_selector() function generates a random k-mer motif for each DNA string and store them in a motifs list.

```
def motif_selector(lines, k):
    motifs = []

    for line in lines:
        rand = random.randrange(0,500-k)
        motif = line[rand:rand+k]
        motifs.append(motif)

    return motifs
```

count_matrix_generator() generates count matrix for each base counts in each column. It first removes the selected motif from the matrix and assumes that all the bases should have their counts at least 1 in each column. So it initializes them as 1 at the beginning, then counts every base in each column and stores them in a matrix.

```
def count_matrix_generator(motifs, k, del_motif):
    motif_matrix = motifs.copy()
    # remove the selected motif from the matrix
    del motif_matrix[del_motif]
    column_matrix = []
    # we assume that all the bases should have their counts at least 1 in each column
    count_matrix = {'A':[1]*k , 'C':[1]*k , 'G':[1]*k , 'T':[1]*k}

    # create a column list from the bases in the same location at the motifs
    for j in range(k):
        column = []
        for motif in motif_matrix:
            column.append(motif[j])
        # count every base at that column and add the numbers to the corresponding column of count matrix
        count_column = Counter(column)
        column_matrix.append(count_column)
        count_matrix['A'][j] += column_matrix[j]['A']
        count_matrix['C'][j] += column_matrix[j]['C']
        count_matrix['G'][j] += column_matrix[j]['G']
        count_matrix['T'][j] += column_matrix[j]['T']

    return count_matrix
```

profile_matrix_generator() generates a profile matrix with the probabilities of the bases in each column.

```
def profile_matrix_generator(motifs, k, del_motif):
    count_matrix = count_matrix_generator(motifs, k, del_motif)
    profile_matrix = {'A':[0]*k , 'C':[0]*k , 'G':[0]*k , 'T':[0]*k}
    col_sum = count_matrix['A'][0] + count_matrix['C'][0] + count_matrix['G'][0] + count_matrix['T'][0]

    for j in range(k):
        profile_matrix['A'][j] = count_matrix['A'][j] / col_sum
        profile_matrix['C'][j] = count_matrix['C'][j] / col_sum
        profile_matrix['G'][j] = count_matrix['G'][j] / col_sum
        profile_matrix['T'][j] = count_matrix['T'][j] / col_sum

    return profile_matrix
```

profile_random_generator() is a weighted random generator. It selects a k-mer motif profile-randomly by using every line's bias values as weights. It computes every possible k-mer in selected line and computes the probability of that k-mer based on the values at given profile matrix. It selects and return a profile-random k-mer.

```
def profile_random_generator(profile, dna_line, k):
    prob_list = [0]*(500-k)
    probs_sum = 0

    # compute every possible k-mer in selected line
    for x in range(500-k):
        probability = 1
        k_mer = dna_line[x:x+k]
        # compute the probability of that k-mer based on the values at profile matrix
        for y in range(k):
            probability *= profile[k_mer[y]][y]
        prob_list[x] = probability
        probs_sum += probability

    # select a profile-random k-mer based on the bias values
    rand = random.uniform(0, probs_sum)
    num = 0
    for x in range(500-k):
        bias = prob_list[x]
        num += bias
        if rand <= num:
            return dna_line[x:x+k]
```

find_consensus() finds the consensus string for a given list of motifs.

```
def find_consensus(motifs, k):
    consensus = []
    # create a column list from the bases in the same location at the motifs
    for y in range(k):
        column = []
        for x in range(1, 10):
            column.append(motifs[x][y])
        # get the most common base in that column and append it to consensus string
        consensus.append(Counter(column).most_common(1)[0][0])
    return consensus
```

score() computes the score for a given list of motifs.

```
def score(motifs, consensus, k):
    score = 0
    for y in range(k):
        column_score = 0
        for x in range(10):
            if not consensus[y] is motifs[x][y]:
                column_score += 1
        score += column_score
    return score
```

gibbs_sampler() generates the list of k-mer motifs for each DNA string, finds the consensus string, and computes the score for the list of motifs. First, it generates motifs list and initializes the best_motifs as the first motifs list.

It selects a random motif for exception. It generates the profile matrix for given list of motifs except the selected motif and generates a new motif profile-randomly. Then it replaces the excepted motif with the new motif and finds the consensus for the new motifs list. It also finds the consensus for the best motifs list. It compares the scores of new motifs list and best motifs list, if the new motifs list gives a better score, makes it the best motifs list.

It iterates those steps 1000 times and at the end it returns the best motifs list, score and consensus string of that list.

```
def gibbs_sampler(dna_lines, k):
    # generate motifs
    motifs = motif_selector(dna_lines, k)
    # initialize the best_motifs as the first motifs list
    best_motifs = motifs.copy()
    final_consensus = ""

    # iterate the algorithm 1000 times
    for j in range(1000):
        # select a random motif for exception
        excepted_motif = random.randrange(0,10)
        line = lines[excepted_motif]
        # generate the profile matrix for given list of motifs except the selected motif
        profile = profile_matrix_generator(motifs, k, excepted_motif)
        # generate a new motif profile-randomly
        new_motif = profile_random_generator(profile, line, k)
        # replace the excepted motif with the new motif
        motifs[excepted_motif] = new_motif
        # find the consensus for the new motifs list
        consensus = find_consensus(motifs, k)
        # find the consensus for the best motifs list
        best_consensus = find_consensus(best_motifs, k)

        # compare the scores of new motifs list and best motifs list
        if score(motifs, consensus, k) < score(best_motifs, best_consensus, k):
            # if the new motifs list gives a better score, make it the best motifs list
            best_motifs = motifs.copy()

    # return the consensus as a string
    for base in best_consensus:
        final_consensus = final_consensus + base

    return best_motifs, score(best_motifs, best_consensus, k), final_consensus
```

4. Conclusion

In general, we run many times both of the algorithms. Then, we observed that the Gibbs Sampler algorithm gives better results compared to the Randomized Motif Search algorithm. But the runtime of the Gibbs Sampler algorithm is higher than Randomized Motif Search. So we can say there is a tradeoff between time and accuracy.

The results for an example run are given below. Here is an example 10-mer and the mutated versions of it:

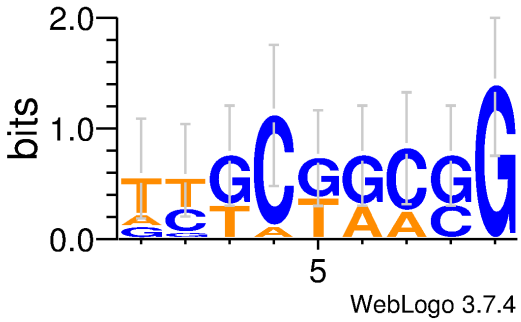
```
In [72]: print(ten_mer)
TCTTCGCCAC

In [73]: for i in ten_mers:
    ...:     print(i)
    ...:
AATTAGGCAC
AGTTGGGCAC
GTTTAGTCAC
AGTTTGGCAC
ATTTAGACAC
GGTTGGACAC
GGTTAGGCAC
GGTTGGTCAC
GGTTAGTCAC
GGTTGGGCAC
```

We run our algorithms three times; for 9-mer, 10-mer, and 11-mer motifs.

9-mer motifs:

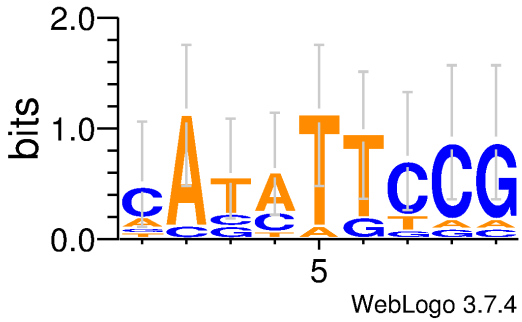
Randomized Motif Search algorithm:

```
9-mer motifs:
GCGATGCGG
TTGCGACGG
GTGCTACCG
ATGCTGACG
TCTCGGCCG
TTGCTGCCG
ACTCGGAGG
TCTCGAAGG
TGTCTACGG
TTGCGGCGG
Consensus: TTGCTGCGG
Score: 30
```



Gibbs Sampler algorithm:

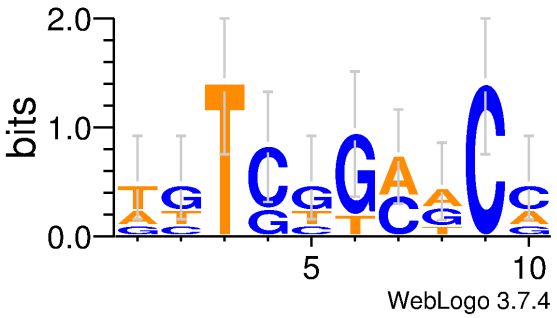
```
9-mer motifs:
CACCATCCG
CCGATTCGG
CATATTTCG
AATATTCCA
CAGCTTTCG
CATATGCCC
TATATGCAG
AACATTCCG
GATCTTCCG
CATTTTGCG
Consensus: CATATTCCG
Score: 23
```



10-mer motifs:

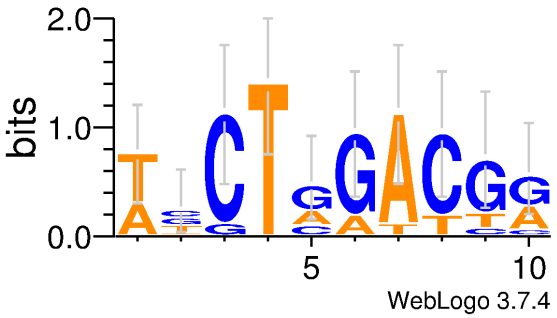
Randomized Motif Search algorithm:

```
10-mer motifs:
TGTCGTCGCC
GCTCGGAACG
ATTCTGCGCC
AGTCTGAACA
TGTCTGCGCG
TGTCCGAGCA
ACTGGGCTCC
GTTGGTCACC
TGTGGGAACA
TTTCCGATCC
Consensus: TGTCGGCGCC
Score: 36
```



Gibbs Sampler algorithm:

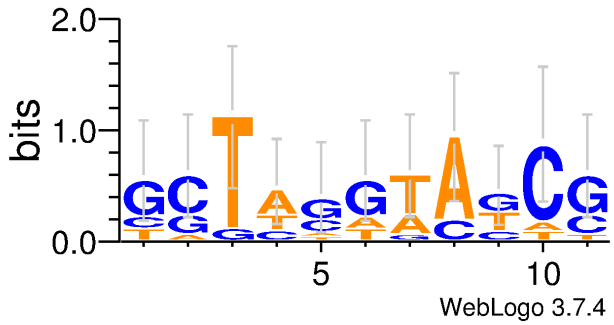
```
10-mer motifs:
TCCTGGACGA
ATGTCGACGG
ATCTAAACGG
AGCTCGTCGG
TCCTAGACGA
TGCTGGATTA
TACTGGACGC
ATCTAGACCG
TCCTGAACGG
TGCTGGATTA
Consensus: TTCTGGACGG
Score: 30
```



11-mer motifs:

Randomized Motif Search algorithm:

```
11-mer motifs:
TCTAGGAAGAG
GGGAGGAACCG
GCTCGAAATCC
GCTTGATCGCG
CATTCGGATCG
TGTAGGTCGCC
GCTCCGTAGTG
CCTATGTACCG
GGTTATTATCC
GCTACTTATCT
Consensus: GCTAGGTAGCG
Score: 41
```



Gibbs Sampler algorithm:

```
11-mer motifs:
GAGGTTACAAG
GCGACAAAAAC
GGAGCTACAAT
GGGTTTACTAT
GGGGTTATACC
GGGATTAAAAT
GGGCCTACTAC
GAGCTTAAAAT
GGGGCTATTAG
GGGGTTGCGTC
Consensus: GGGGTTACAAC
Score: 32
```