CSE4065 Introduction to Computational Genomics

Pattern Search Algorithms

April 14, 2021

150115037 - Can Berk Durmuş

150117069 - Elif Balcı

Overview

Random Pattern Search and Gibbs Sampling algorithms have been implemented by us to compare the results of those two algorithms.

The Input File

The input file has been created using a simple Python function which firstly creates a 10-mer and then places it into 10 DNA strings with 500 nucleotides length by mutating 4 random nucleotides of each one. Details of the input file can be seen below. The input file is being saved as dna.txt and the information file for the input file is being saved as dna_info.txt. The information file includes the original 10-mer that has been placed into the DNA and the mutant variants of it.

Original	tctacttaca
Mutant-1	cctactaagt
Mutant-2	tctatctaag
Mutant-3	tccacgacca
Mutant-4	tctgtatacg
Mutant-5	acttcattca
Mutant-6	tcaactacga
Mutant-7	tcgacttcgt
Mutant-8	tccggttact
Mutant-9	tacgcttgca
Mutant-10	tgtactaccg

Implementation Details

1. Random Motif Search

Random Motif Search has been implemented as below. In the first step, the variables that are needed to be used across the loop are initialized. At the beginning of the first loop, the algorithm creates a profile for the initial motifs and after that, it creates a new set of motifs to be used. After the first loop, every loop the profile calculation will be made according to the latest created motifs. The algorithm keeps track of the best score and the motifs that belong to that score. There is a temporary score buffer to keep track of the improvements on the score, so the algorithm can stop when it reaches a point that it won't be improving the score.

```
def randomized_motif_search(self):
    motifs = self.initial_motifs
    iterations = 0
    best_score = None
    best_motifs = None
    while True:
        iterations += 1
        self.set_new_profile(motifs)
        motifs = self.create_new_motifs()
        score = self.calculate_score(motifs)
        if best_score is None:
            best_score = score
            best_motifs = motifs

        if score < best_score:
            best_score = score
            best_motifs = motifs

        self.temp_score_buffer.append(score)

        if len(self.temp_score_buffer) >= self.check_period:
            mean = statistics.mean(self.temp_score_buffer)
            if mean <= score:
                # Terminate random search
                self.best_motifs = best_motifs
                self.iterations = iterations
                self.final_score = best_score
                self.consensus_string = self.get_consensus_string()
                return
            else:
                self.temp_score_buffer.clear()
```

2. Gibbs Sampler

Gibbs Sampling algorithm is implemented as follows. The algorithm selects one random motif to change in every loop. After one motif changes, the algorithm profiles the new motif set, after that it calculates the score of the motif set. The algorithm keeps track of the best score and the meat motif set that belongs to that score. At the end of every loop, the algorithm checks the temp score buffer and if it is full, it checks the overall improvement to terminate the iterations. If the improvement stopped, the algorithm will decide to terminate itself.

```
def gibbs_sampler(self):
    motifs = self.initial_motifs
    iterations = 0
    best_score = None
    best_motifs = None
    while True:
        iterations += 1
        i = random.randint(0, len(self.dna) - 1)
        copy_motifs = motifs.copy()
        copy_motifs.pop(i)
        self.set_new_profile(copy_motifs)
        motifs[i] = self.get_motif_from_string(self.dna[i], prob_dist=True)
        score = self.calculate_score(motifs)

        if best_score is None:
            best_score = score
            best_motifs = motifs
        if score < best_score:
            best_score = score
            best_motifs = motifs

        self.temp_score_buffer.append(score)

        if len(self.temp_score_buffer) >= self.check_period:
            mean = statistics.mean(self.temp_score_buffer)
            if mean <= score:
                # Terminate random search
                self.best_motifs = best_motifs
                self.iterations = iterations
                self.final_score = best_score
                self.consensus_string = self.get_consensus_string()
                return
            else:
                self.temp_score_buffer.clear()
```

Results

The main.py file creates a pair of instances of MotifSearch class for every k=9, k=10, and k=11 numbers. The results can be seen by the output:

Run	k	Gibbs Sampling Score	Gibbs Iterations	RPS Score	RPS Iterations
1	9	34	100	33	50
1	10	34	150	34	100
1	11	38	150	43	50
2	9	25	1500	28	1000
2	10	32	1000	38	1000
2	11	32	500	39	500
3	9	25	1500	32	1000
3	10	32	100	32	1000
3	11	38	500	41	1000

As we can see that if we can be sure that the algorithms iterate enough (with setting the buffer check size ~500) the Gibbs Sampling works significantly better. The first run is done with buffer check size 50 and the superiority of the Gibbs Sampling is not very clear.