Bioinformatics Homework 1.2

Implementing Gibbs Sampler

Problem Statement: Given a set of *t* strings "*DNA"*, and integers *k*, and *N*, find the hidden k-mer motif in *DNA* where the motif occurs once in each of the strings in *DNA* with some differences. Return a set of motifs that are the most probable representation of the hidden motif in each of the strings in *DNA*.

Algorithm Description:

```
GibbsSample(DNA, k, t, N)
     {
randomly select k-mer Motifs = (M1, M2... Mt) from DNA
BestMotifs <- Motifs
for (j <- 1 to N)
i <- Random(t)
// Stores frequency of each letter for each position              // of
each motif
Profile <- profile matrix made from all the current
motifs except for the motif in string i
// Reduces chaotic answers
Impliment Laplace smoothing on the matrix
// Random selection weighted by profile probabilities
Motif[i] = new randomly chosen motif from profile
if score(motifs) < score(BestMotifs)
BestMotifs = motifs

return BestMotifs
     }
```

First, select random k-mer from each of the strings in DNA. Create a set that will store the best motifs that we have found so far and set it to the randomly selected k-mers we just selected. Then, we will run the following process *N* times: Choose a random string from the *DNA*. Then, create a frequency profile for all of the strings except for the one we just chose. Find the probabilities for each k-mer in the chosen string from the frequency profile. Randomly choose a new motif for that string using the weights from the frequency profile. If the score for this new set of motifs is better than the score we have for our best motifs, we set our best motifs to our current motifs. At the end of this algorithm, the set contains a set of string that most probably contain a hidden k-mer in DNA

Time Analysis:

This algorithm is interesting because the actual runtime may differ drastically depending on what the variables input into the function are. The Gibbs sampler runs the

program for a set number of times. In each iteration of the function, the program finds the frequency of letters in motifs and compares the scores between the current set of motifs and the best set of motifs. These functions run in O(t*k) since it needs to check t motifs and k letters for each motif to update both the profile and the current score. Additionally, when finding the probability for the k-mer we will update in the “removed” DNA string, we need to do O(DNA-String-Length) operations to update the probabilities using a sliding window. This means that the runtime is N * ((DNA-String-Length) + (t * k)) or when simplified, O (N * max(k*t, DNA-String-Length)).

Implementation:

Submission Number: **#1072706**

```
#include <bits/stdc++.h>
using namespace std;
#define all(x) (x).begin(), (x).end()
#define sz(x) (int)(x).size()
#define rep(i, a, b) for (int i = a; i < (int)b; ++i)
typedef long double ld;


vector<string> GibbsSampler(vector<string> DNA, int k, int t, int N)
{
    // Setting up a random number generator
    random_device rd;
    mt19937 gen(rd());

    // The program had difficulty finding the global minimum under a small N
    N = 1000000;

    // Initialization
    map<char, int> mp = {{'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}};
    vector<int> motifs(t), bestMotifs;
    int bestScore = 100000;

    // Adding our initial motifs
    rep (i, 0, t)
    {
        uniform_int_distribution<> dist(0, sz(DNA[i])-k);
        motifs[i] = dist(gen);
    }

    bestMotifs = motifs;

    // The iterative process to find better motifs
    rep (x, 0, N)
    {
        // Randomly picking a removal index
        uniform_int_distribution<> dist(0, t-1);
        int removalIndex = dist(gen);

        // Creating a profile based on the motifs - not including motif 'removalIndex'
        vector<vector<ld>> profile(4, vector<ld>(k, 0.0));

        rep (i, 0, t) if (i != removalIndex)
            rep (j, 0, k) profile[mp[DNA[i][j+motifs[i]]]][j]++;
```

```
        // Adding pseudocounts
        rep (i, 0, 4) rep (j, 0, k)
            profile[i][j] += 1.0;

        // Finding the probabilities for each k-mer in the removed string
        vector<ld> prob;
        rep (start, 0, sz(DNA[removalIndex]) - k + 1)
        {
            ld p = 1.0;
            rep (j, 0, k) p *= profile[mp[DNA[removalIndex][start + j]]][j];
            prob.push_back(p);
        }

        // Picking a random motif based on the probabilities we calculated
        discrete_distribution<> newMotifDist(all(prob));
        motifs[removalIndex] = newMotifDist(gen);

        // Finding the score for the new set of motifs
        int currentScore = 0;
        rep (j, 0, k)
        {
            vector<int> colCount(4, 0);
            rep (i, 0, t) colCount[mp[DNA[i][motifs[i] + j]]]++;
            currentScore += t - *max_element(all(colCount));
        }

        // Updating bestMotifs if we improved score
        if (currentScore < bestScore)
        {
            bestScore = currentScore;
            bestMotifs = motifs;
        }
    }

    vector<string> ans;
    rep (i, 0, t)
        ans.push_back(DNA[i].substr(bestMotifs[i], k));

    return ans;
}
```

Discussion:

This algorithm is interesting in the sense that a large portion of the runtime comes from the number of iterations it performs. Because of its random nature, these iterations are a crucial part of this algorithm. In addition to this, string character comparison and addition for the profile matrix for the probability calculation for our “7-sided die” is costly for the time complexity of this program. Although doing these comparisons is costly and the number of iterations can be quite large, I believe that this algorithm solves the problem in a much better way than most alternatives. Frankly, this problem is so difficult to solve in an efficient manner, that one of the best solutions is to use randomness to find a most probable global motif.

For the implementation, I attempted to minimize double computations and string comparisons. Generally, finding optimizations for this algorithm is difficult- other than small optimizations, I could not find as many potentially impactful optimizations for this algorithm.