CS 466 Final Project:
A Comparison of Motif-Finding Algorithms

Sidney Oderberg, NetID: sidneyo2

1. Abstract

Motif-finding is a bioinformatics problem that aims to find **transcription factor binding sites** in DNA. I compared three algorithms: Greedy Search, Beam Search, and Gibbs sampling. These methods aim to maximize the **information content** of probability distribution sets for the bases of sections of DNA. I ran several experiments to see how information content and program running time vary with different factors of the algorithms and found similar trends across all three methods. Gibbs sampling appears to be the fastest way to achieve a high information content.

2. Introduction

DNA consists of a sequence of nucleotides. The four bases (components of nucleotides) in DNA are adenine (A), cytosine (C), guanine (G), and thymine (T). Certain sections of DNA, called **genes**, encode the information needed to construct **proteins**. Proteins are an essential molecule in the functioning of an organism. Cells of different types produce different combinations of proteins to perform different functions. This differentiation allowed for the evolution of complex multi-cellular organisms. But, since each cell in an organism has the same DNA, an important question to ask is how cells know which genes to use for protein production so that they can differentiate.

A single gene can be used to produce multiple different proteins. This is possible because genes have multiple sub-sections call exons. Different combinations and orderings of the exons will lead to different proteins. The base sequences between exons are called introns and are not read in the protein synthesis process. Introns are an example of a part of the DNA for which their function (if there is one) is unknown. In fact, it is widely believed that genes only account for 3% of DNA. The function of the remaining 97% is largely still a mystery. However, some of that mystery is being unraveled with the discovery of **transcription factor binding sites**. These non-gene sections of DNA are the answer to the question posed above regarding cell differentiation.

Gene regulation is a process that determines which proteins and how much of them get produced from a gene. **Transcription factors**, a kind of protein, play an essential role in this process. By binding (or not binding) to a binding site, transcription factors determine the regulation of a specific gene. This regulation is what causes cells to differentiate by determining which proteins they produce. Finding these binding sites so that they can be studied is a common challenge in the field of bioinformatics today and was the goal of my project.

In the next section I will discuss the computational approach to solving this problem. In section 4 I will describe the three algorithms I compared (Greedy Search, Beam Search, and Gibbs Sampling), in section 5 will show the results of my experiments, in section 6 I will discuss the results, and in section 7 I will conclude.

3. Approach

Transitioning into the computational side of the problem, this becomes a string search task. Given a series of DNA sequences (strings), find a substring pattern that occurs in all (or enough times to be significant) the given sequences. This pattern is also called a **motif**. Motifs do not have to be exact strings, meaning the characters (from “A”, “C”, “G”, and “T”) are not all the same from one instance of the motif to another. However, the approach to finding motifs assumes that they look like each other and look different from the rest of the sequence. For example, two instances of a motif could be "CCTTATG" and "CCTTCTG." The fifth character differs, but it is clear that these two strings resemble each other. Each character from a motif is drawn from some probability distribution that differs from a uniform distribution. If this distribution can be found, then instances of a motif can be differentiated from non-motif regions by determining the likelihood that any substring is from the specific motif distribution.

Before the motifs can be found, I will describe the steps used to evaluate whether a set of probability distributions for motif characters is significant or not. The first step is to assume that all the starting positions of the motifs in the set of strings have been found and that we know the length of the motif. Then, assemble the motif instances into a matrix where each row is a different instance and each column is the same position in the motif. Construct a **Profile Matrix** by counting the number of bases at each position. This results in a matrix of four rows (one for each base) and columns equal to the length of the motif.

The next step is to turn the profile matrix into a **Position Weight Matrix** (PWM). This turns frequencies of appearance into probabilities by dividing each count by the number of substrings. Then an equation can be used to evaluate the position weight matrix:

$$\sum_{k=1}^{L} \sum_{B \in [A,C,G,T]} W_{Bk} * log \frac{W_{Bk}}{q_B}$$

where L is the motif length, B is a base (matrix row), k is a matrix column, W is the PWM, and q is the probability of seeing a base anywhere in the sequences (generally this will be one-fourth). Evaluating this equation yields the **Information Content.** A higher information content means the found motif is more distinguishable from the rest of the sequence.

Now that a motif and its found instances can be evaluated, we return to the problem of finding the starting positions within a set of DNA sequences. The naïve approach is to check all combinations of substrings across all the sequences in the set. This, however, is intractable since the time complexity is $O(N^s)$, where 's' is the number of sequences being studied and 'N' is the length of a sequence. Accordingly, there are several approaches to this problem that aim to find the best possible motifs while maintaining a reasonable time complexity. I will discuss the three algorithms that I used in the next section.

4. Implementation

For my project, I decided to implement and compare the three motif-finding algorithms that we discussed in class. These are the **Greedy Search**, **Beam Search**, and **Gibbs Sampling**. I implemented all of these in Python and ran them on my Windows machine.

I. Greedy Search

The greedy search approach sacrifices finding the optimal solution for improved running time. Start with the first two sequences. Compare all combinations of starting positions between these two to get an initial guess on the motif. Keep the two substrings that result in the highest information content. Then, iterate through the remaining sequences. At each sequence, find the substring that, when added to the list of substrings found so far, produces the PWM with the highest information content. At the end of the iterations, the greedy search is complete, and a list of motif instances have been found. I then compute the final PWM and information content and find the **consensus motif**. The consensus motif is just the most likely substring given the probability distributions on the characters at each position of the motif. My program returns the consensus motif, final information content, and time elapsed for the algorithm.

Most of the time that the program spends is finding the initial guess from the first two sequences. The time complexity of this algorithm is $O(N^2 + (s-2)*N)$. The quadratic term outweighs the linear term, so the final complexity is just $O(N^2)$. 'N' again represents the length of the sequences and 's' is the number of sequences.

II. Beam Search

One problem with the greedy search is that the initial guess could have been bad and that influences the rest of the search. A bad initial guess could result in a bad final approximation for the motif. To combat this, beam search keeps track of the best 'k' guesses at each iteration. In my program, I created a class called 'K_Best' to handle this. It is similar to a KD-Tree. The 'K_Best' class stores a dictionary of tuples, a worst score, and 'k'. Each tuple has a value and a list (the data). The dictionary keeps the tuples sorted by value (with higher being sooner in the dictionary). Order is maintained by having the dictionary keys be the numbers from 0 to (k-1). This class has one method: adding tuples. If a tuple's value is lower that the worst score so far, it will be rejected and not added. Otherwise, it will be placed in the correct place to maintain the sorted order. In my program, the scores corresponded to the information content, and the data corresponded to a list of substrings that were instances of a possible motif.

With this new data structure at hand, my implementation of the beam search was very similar to the greedy search. Instead of finding the best initial guess, I find and keep track of the best 'k' initial guesses. Then, in each iteration step, I find the 'k' new best lists of substrings by testing the current sequence's starting positions with the previous best 'k' substring lists. At the end of the iterations, I take the best substring list and again find the final PWM, information content, and consensus motif. I report those as well as the elapsed time.

The time complexity for beam search ends up being the same as greedy search. It's $O(N^2 + (s-2) * k * N)$. The 'k' is added because each position at each of the sequences is tested with 'k' guesses. However, this also just simplifies to $O(N^2)$. The space complexity, on the other hand, increases k-fold because of all the extra information that is kept throughout the execution of the program.

III. Gibbs Sampling

The final algorithm that I implemented was Gibbs Sampling. The aim of this method is to further reduce the chance that a local optimum for the information content is found and instead aim for the global optimum. It is a probabilistic approach to finding motif starting positions in the sequences. In my program, it can be run in one of two ways: by starting with a solution from the greedy algorithm or starting with a random set of starting positions.

For either way that the starting positions are initialized, the rest of the program runs the same. For random number generation I used NumPy's 'random' module. Out of a complete set of starting positions, pick one sequence (at random) to leave out. Using the remaining sequences, compute the PWM to get the character (base) probability distributions at each position of the motif. These distributions will be used to find a new starting position from the sequence that was left out.

To move towards the global optimum information content, the new position must be chosen with probability proportional to the motif probability distributions. In my program, I ensured this as follows. For each potential starting position x, I computed the likelihood of the motif starting there using the product of the probabilities for each character in the previously calculated PWM. I then divide these likelihoods by the sum of all likelihoods so that I have a list of likelihoods that sum to one. Then I get a random float value between zero and one. I use this to determine the starting position by considering the list of likelihoods to represent buckets that the random value could fall into. A bigger likelihood is a bigger bucket, meaning it is more likely to get picked by the random float. Once I have the new starting position, I get a new full list of substrings and compute another PWM and information content. If it's higher than what I've seen so far, I keep track of the substring list as the best list so far.

This sampling process of removing a sequence and finding a new starting position is repeated some number of times. At the end, I use the best seen list of substrings to get the final PWM, information content, and consensus motif.

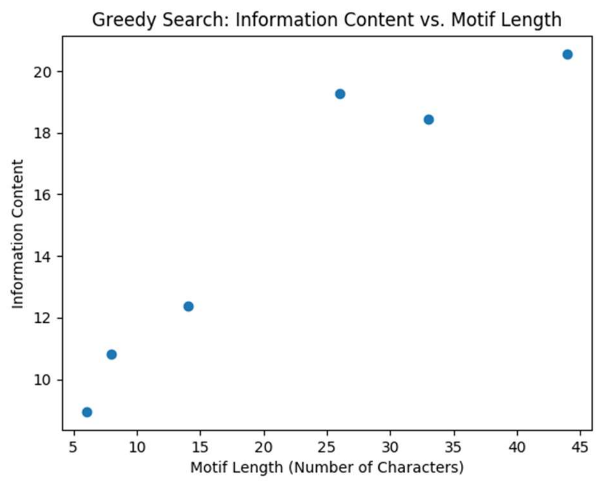
When using the greedy search solution as the starting point, the time complexity is again $O(N^2)$, because most of the execution time is spent finding the initial solution. This can be improved drastically by starting with random positions. The time complexity is then $O(TN)$, where 'T' is the chosen number of times to sample a new starting position. This is the complexity because each iteration of sampling just looks through a single sequence, so there is no quadratic term.

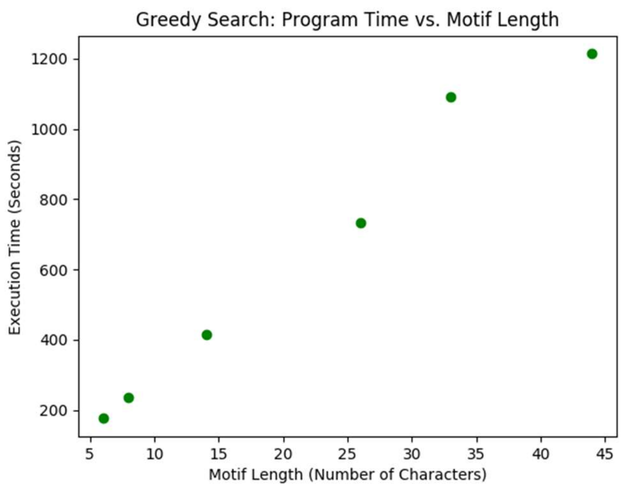
5. Results

In this section I will show several plots of information content/execution time (in seconds) vs. motif length for the three algorithms. For beam search I also experimented with different values for 'k' and for Gibbs sampling I experimented with different numbers of samples and whether I started with the greedy search solution or not.

I used data from a University of Washington study [1]. They had DNA sequences from four organisms: humans, mice, *D. melanogaster*, and *S. cerevisiae*. For my experiments I used one of the files containing a list of human DNA sequences: *"hm01r.fasta"*.

Experiment 1: Greedy Search
This tests the greedy search for the following values of motif length: [6, 8, 14, 26, 33, 44]. The plots show how information content and program execution time vary with motif length.



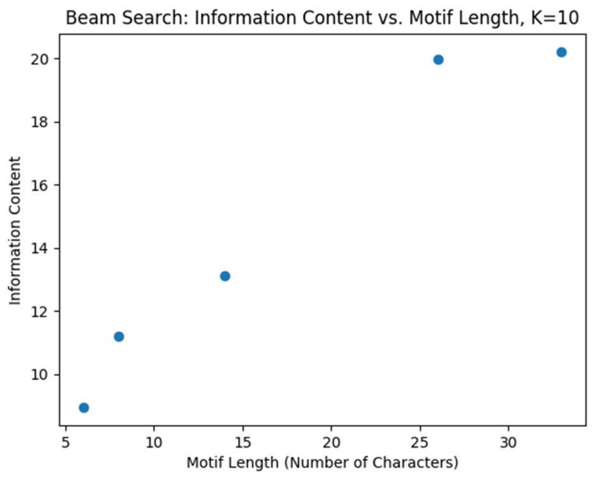


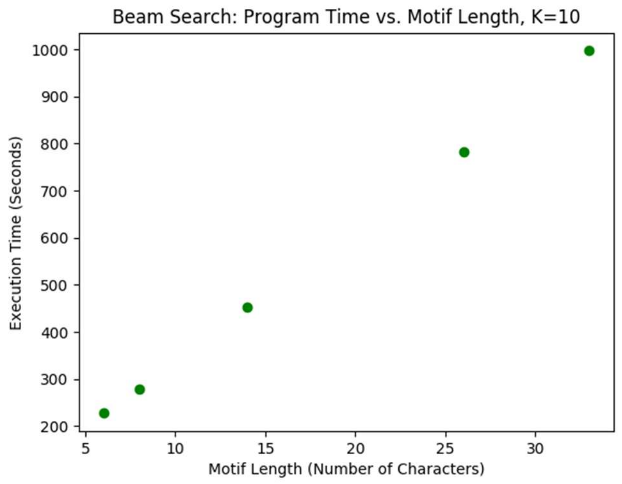
Motifs Found:

- TGGGGG
- CCCTGGGA
- CCTTTCCCTGTTGC
- TCTCTCCCCCTTCTTCTTTTCTTCCT
- TATTATCTCTCTCTTTTCCCCGTTGCCTCCTTC
- CTTCTCTGTTTTTTTCCTCTTTTCCTTCAGAGATTGCGACTCAC

Experiment 2: Beam Search, varying length

This tests the beam search for the following values of motif length: [6, 8, 14, 26, 33]. The plots show how information content and program execution time vary with motif length.



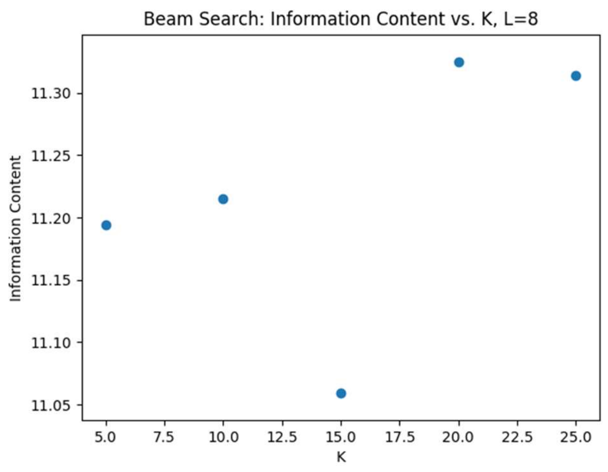


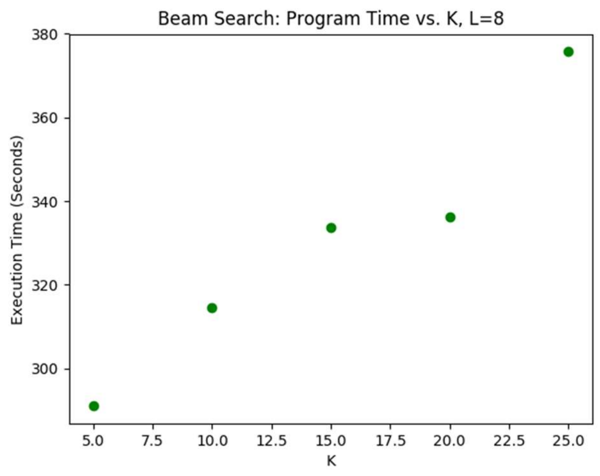
Motifs Found:

- TGGGGG
- CCTGGCTC
- CTCTCCCTGTTCCC
- TCCCAGCTCACTGCAGCCTCCACCTC
- CTGCCTTCCCTCCCTCTCCTCCCTCCCCACCTC

Experiment 3: Beam Search, varying K

This tests the beam search for the following values of K: [5, 10, 15, 20, 25]. The plots show how information content and program execution time vary with K.



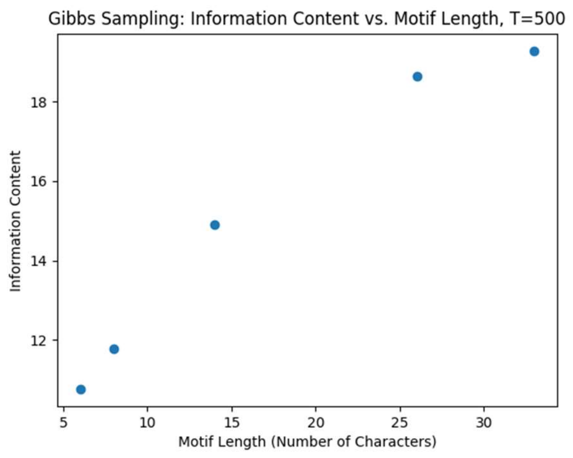


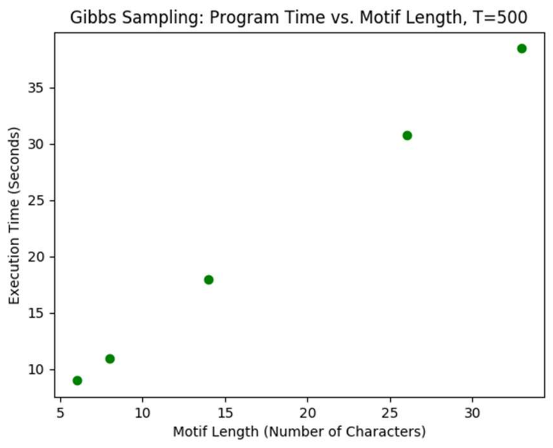
Motifs Found:

- GCCCTGGG
- CCTGGCTC
- CCTGGCTC
- CCTGGCTC
- CCTGGCTC

Experiment 4: Gibbs Sampling, varying length

This tests the Gibbs sampling method for the following values of motif length: [6, 8, 14, 26, 33]. The plots show how information content and program execution time vary with motif length.



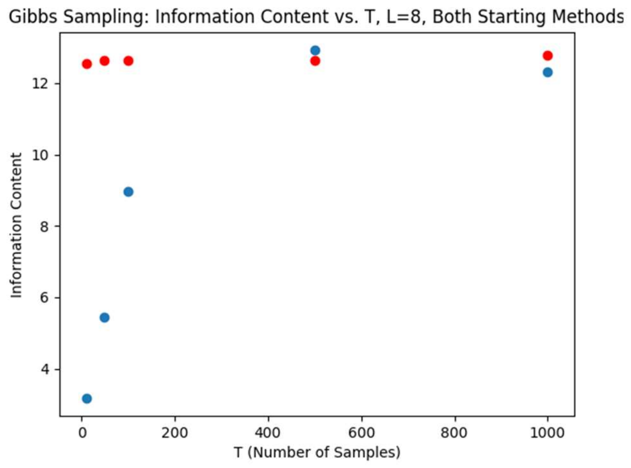


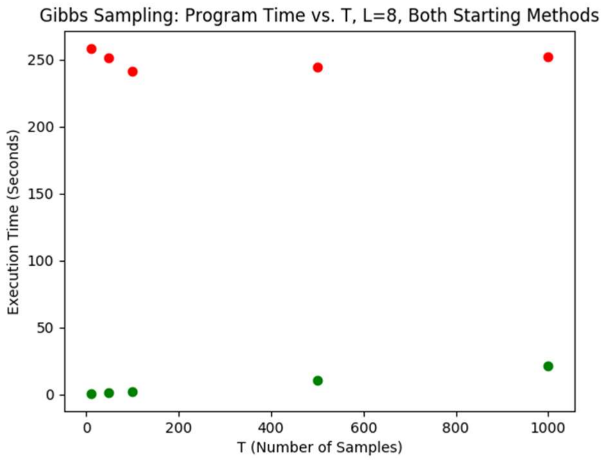
Motifs Found:

- TGTTTT
- GGAAATGG
- AAAAAAAAAAAAAA
- ACAGAAAAAAAAAAAAAAATAAAGAAC
- CCAAGGTGCGGGGATTACAGGCGTGAGGCCCGG

Experiment 5: Gibbs Sampling, varying T and starting with Greedy solution vs. random solution

This tests the Gibbs sampling for the following values of T (number of samples): [10, 50, 100, 500, 1000]. The plots show how information content and program execution time vary with T. They also compare starting with a solution to the greedy search (red dots) compare to starting with random starting positions.





Motifs Found:

- Random Starting: AAAAATTC, Greedy Starting: CCCTGGGA
- Random Starting: CAATGCAG, Greedy Starting: CCCTGGGA
- Random Starting: GTGGGTGG, Greedy Starting: CCCTGGGC
- Random Starting: CCAGCCTG, Greedy Starting: GCCTCGGC
- Random Starting: CAGCACAG, Greedy Starting: CCCTCAGC

6. Discussion

The results were for the most part what I expected. Experiment 1 shows that both the information content and program execution time increase as the desired motif length increases. The information content increase may not be the most valuable information since it could just be increasing because the PWM has more values to sum over. It could also be that all the lengths I chose are reasonable motif lengths, and if I were to increase the length even more the information content could go down. At a certain length it could be that binding sites aren't that long, so any motifs found would not be distinguishable from the rest of the sequence. More experiments would need to be run to investigate that further. However, the running time increased linearly with the motif length increases. Note that this differs from the quadratic time that I calculated in section 4. This is because it is quadratic in terms of the entire sequence length, but I was varying the motif length. That's why there's no quadratic increase in time, and the linear increase is just from having to do more operations for longer motifs.

The results from experiment 2 seem to be about the same as those from experiment 1. Both information content and time increased with motif length. Upon close inspection the information content appears to be slightly higher for the trials in experiment 2 than 1, but the increase is not very significant. An explanation for this is that the greedy search is getting lucky with finding a good initial guess. It could also just be a property of the data I used.

Experiment 3 yielded an unexpected result. Running beam search with k=15 resulted in the lowest information content. I expected an increasing k to result in an always increasing information content. It seems that beam search is not as consistent as I had thought. In that one trial there must have been some bad guesses that fit within the top k and then somehow got ahead of the others while still being worse than the guesses of the previous trial. The time increased as expected.

I was surprised at how quickly Gibbs sampling (using random starting positions) could reach information contents comparable to the other methods. Experiment 4 shows a similar trend in information content vs. motif length, but the time plot shows that the trials ran in 10-40 seconds, while the trials of the other experiments were 200-1000 seconds. These time ranges align with the fact that I had calculated linear time complexity for Gibbs sampling (compared to the quadratic time complexities of the other methods).

I found the results of experiment 5 to be interesting as well. This experiment shows the importance of using enough samples. The first plot shows that Gibbs sampling, when starting from a greedy search solution, will be consistently high, whereas starting with random starting positions will yield a low information content if too few samples are used. The second plot shows that the running time generally increases with the number of samples but using a greedy search first will take up most of the program's time. A result that isn't shown here is that Gibbs sampling could increase the information content of a motif found in the greedy search while keeping the motif the same. This means that the sampling found better instances of the same motif to back it up as a good solution. Other times the sampling would result in a slightly different motif, but either way the information content would increase from Gibbs sampling.

7. Conclusion

From my results, Gibbs sampling appears to be the best algorithm to use for motif-finding. From method to method, the information contents that I saw per motif length were similar. However, the time benefit from Gibbs sampling (using random starting positions) was very significant.

However, it is important to note that all these methods are just approximations and are all at risk of getting stuck in a local optimum.

I found the motif-finding problem to be very interesting and I wonder if these techniques could be used to unlock the mysteries of the remaining unknown sections of DNA.

References

[1] University of Washington, Department of Computer Science and Engineering. http://bio.cs.washington.edu/assessment/download.html

Libraries Used

1.) Biopython. V1.71. http://biopython.org/
2.) NumPy. v1.14. http://www.numpy.org/