Three Approaches to Solving the Motif-Finding Problem

Zachariah Huebener
Computer Science Department
Simpson College
Indianola, IA 50125
zach.huebener@my.simpson.edu

Kylie Van Houten
Computer Science Department
Simpson College
Indianola, IA 50125
kylie.vanhouten@my.simpson.edu

Abstract

The purpose of this paper is to analyze three methods for solving the Motif-Finding problem. The Motif-Finding problem is the problem of finding patterns in sequences of DNA. The methods we analyze compare many DNA strands of equal length and find the most closely-matching sequences of a certain length in each strand. These patterns are of great scientific interest to those doing research in genetics because they correspond to sequences of DNA that control the activation of specific genes.

In this paper we compare the brute force method, the branch and bound method, and the greedy method as ways of solving the problem. Then we analyze the complexity and running time of the methods. Our experimental results show that the greedy algorithm can find an approximate solution to large files in a reasonable amount of time. The branch and bound method is more efficient than the brute force method, but neither method is able to find solutions for files with more than 10 strands of DNA in a reasonable amount of time.

1 Introduction

Motif finding is described as “the problem of ...discovering of motifs without any prior knowledge of how the motifs look” [1]. But what is a motif? *An Introduction to Bioinformatics Algorithms* by Neil C. Jones and Pavel A. Pevzner describes regulatory motifs as binding sites in strands of DNA that activate certain genes, such as immunity genes, at the proper time. Proteins bind to these binding sites and encourage RNA polymerase to make an exact copy of the genes necessary for fighting infections.

For instance, when a fruit fly becomes infected, the immunity genes that are normally dormant in the fly genome become active to start producing proteins that will destroy the pathogen. The regulatory motifs tell the genes to activate, so it is of great importance for biologists to know what these motifs are and how they function. Unfortunately, it is no easy task to determine what these motifs are. One cannot simply look at a fly under a microscope and see what each part of the DNA strand is doing. The best approach that is currently known is to analyze many strands of DNA and search for patterns because the patterns might be the motifs in question.

The strands of DNA are represented as strings with a, t, c, and g denoting the four possible nucleotide bases adenine, thymine, cytosine, and guanine. It is a simple process, though by no means a quick process, to find all possible string patterns and then determine which pattern has the most identical matches in other DNA strings. However, this process fails to find motifs with small mutations that do not change their function but cause them to be distinct from one another in regards to nucleotide sequence. So we cannot simply search for identical patterns in the strings because the motifs will generally look similar, but not identical.

Assuming we can find a string in each sequence that very closely matches the others, a “profile matrix” can be constructed to total the number of each nucleotide base in each position of the strings [1]. The most common letter in each column of the matrix is concatenated together to form a new string called the “consensus string” [1]. This could be thought of as the “perfect” motif, and all the strings that were found are slight mutations of it.

The method for finding all of those matches is the Median String Method. It uses a metrics known as the Hamming distance, which gives a numerical answer for the number of positions that differ between two strings [1]. For example, the string ATGC and the string AGGA would have a Hamming distance of 2 because two of the positions are different. For a string of length l the method finds the pattern with the same length that is the best match (meaning they have the smallest Hamming distance) in each DNA sequence and records the positions within each sequence at which those patterns begin. The total Hamming distance is the sum of the Hamming distance between the original string and each closest match. Then this process is repeated for every possible distinct string of length l (which is 4^l strings) to find the string that has the minimum total Hamming Distance.

The string with the minimum total Hamming distance is the "Median String," which, interestingly enough, is the same as the consensus string [1]! So once the Median String has been found, the original method using the profile matrix can be run backwards to find the closest matching string in each DNA sequence-the closest match being the one with the smallest Hamming distance as compared to the Median String. So the best way to find a motif is to search all possible strings of a certain length to find the string that has the closest matches in all the DNA strands and those closest matches are the best possibilities of being the motif that is being searched for. This is an exhaustive search method that is very inefficient even though it delivers an exact solution.

In the sections below we discuss three algorithms used to solve the motif-finding problem based on the brute force method, the branch and bound method, and the greedy method. First, we briefly introduce three types of tree traversals used in the algorithms and a method to evaluate a string – the Score method. Next, we discuss the details of the algorithms. Finally, we present our experimental results.

2 Tree Traversals and Score Method

The method for finding the consensus string that is described in *An Introduction to Bioinformatics Algorithms* generates all possible sets of starting positions as arrays and then determines the set that produces the best consensus string. A tree is used to generate all possible sets of starting positions, and three tree traversal methods are used in the brute force and the branch and bound algorithms for solving the motif-finding problem.

2.1 NextVertex Method

The first tree traversal is called NextVertex. It sequentially generates every vertex of the tree from the root to the leaves. The leaves of this tree are used as the sets of starting positions to search in each DNA sequence for the best motif.

2.2 NextLeaf Method

The NextLeaf method is very similar to the NextVertex method, but it only produces the leaves of the tree and not the other vertices.

2.3 Bypass method

The Bypass method is used in conjunction with the NextVertex method in one of our approaches to solving the motif-finding problem. It is used to skip certain branches in the tree generated by the NextVertex method if it is determined that they will not produce a desired result. When we no longer wish to continue moving down a certain branch of the tree to check those vertices the Bypass method jumps up and over in the tree to the next branch to check those vertices.

2.4 Score Method

The DNA score is basically the opposite of the Hamming distance. The Hamming distance totals the number of positions that differ between two strings, but the DNA score totals the number of positions between two strings that are the same. Thus, a higher score means that the strings are more identical to one another.

The Score method returns the total score of each consensus string. It first creates the profile matrix for the DNA, which is an integer array with length four. The profile matrix compares the motifs found at each position in the current starting position set, and totals the number of times each letter (a,t,g, and c) occurs in each column of the matrix. The letters with the most occurrences in each column are concatenated together to create the consensus string. The total DNA score for the consensus string is the sum of the number of times that each of those letters occurs in their respective columns.

3 Three Algorithms to Solving the Motif-Finding Problem

We implemented three algorithms to solve the motif-finding problem: a brute force algorithm, a branch and bound algorithm, and a greedy algorithm.

3.1 Brute Force Algorithm

The brute force algorithm determines the consensus string by determining which set of starting positions produces the best DNA score. It is an exhaustive search because it checks every possible set of starting positions produced by the NextLeaf method.

```
ALGORITHM BruteForce ($DNA, t, n, l$)
//Determines the consensus string by finding the set of starting positions that produces the best DNA score
//Input: Two-dimensional Array $DNA$ contains all the strands of DNA to be checked
          Integer $t$ is the number of DNA strands to be checked
          Integer $n$ is the length of each DNA strand (they all have the same length)
          Integer $l$ is the length of the consensus string (the motif to be found)
//Output: A set of starting positions for the DNA sequences that produce the best consensus string. This consensus string is printed to the screen.
1       $\mathbf{s} \leftarrow (1, 1, \ldots, 1)$
2       $bestScore \leftarrow Score(\mathbf{s}, DNA)$
```

```
3    while forever
4        s ← NEXTLEAF(s, $t$, $n – l + 1$)
5        if Score(s, DNA) > bestScore
6            bestScore ← Score(s, DNA)
7            bestMotif ← ($s_1, s_2, …, s_t$)
8        if $s = (1, 1, …, 1)$
9            return bestMotif
```

3.1.1 Complexity

The complexity of the exhaustive search is $O(ln^t)$ where *l* is the length of the motif, *n* is the length of the DNA samples, and *t* is the number of DNA samples. So the method takes exponentially longer to solve as more DNA strands are checked.

3.2 Branch and Bound Algorithm

The branch and bound algorithm uses a combination of the NextVertex method and the Bypass method to skip branches of the tree if it is determined that they cannot produce a better score than the score that has already been obtained.

ALGORITHM BranchAndBoundMotifSearch (*DNA*, *t*, *n*, *l*)
//Faster brute force method that skips branches of the tree while determining the consensus string by finding the set of starting positions that produces the best DNA score
//Input: Two-dimensional Array *DNA* contains all the strands of DNA to be checked
Integer *t* is the number of DNA strands to be checked
Integer *n* is the length of each DNA strand (they all have the same length)
Integer *l* is the length of the consensus string (the motif to be found)
//Output: A set of starting positions for the DNA sequences that produce the best consensus string. This consensus string is printed to the screen.

```
1    s ← (1,…,1)
2    bestScore ← 0
3    $i$ ← 1
4    while $i > 0$
5        if $i < t$
6            optimisticScore ← Score(s, $i$, DNA) + $(t – i) \cdot l$
7            if optimisticScore < bestScore
8                (s, $i$) ← BYPASS(s, $i$, $t$, $n – l + 1$)
9            else
10               (s, $i$) ← NEXTVERTEX(s, $i$, $t$, $n – l + 1$)
11       else
12           if Score(s, DNA) > bestScore
13               bestScore ← Score(s)
14               bestMotif ← ($s_1, s_2, …, s_t$)
```

```
15                (s, i) ← NEXTVERTEX(s, i, t, n – l + 1)
16    return bestMotif
```

3.3 Greedy Algorithm

The greedy algorithm does not use any of the aforementioned tree traversals because it is not an exhaustive search method. However, the greedy method does do an exhaustive search on the first two strands of DNA to determine the best motif in these two strands. This motif is called the seed. The method then sequentially searches the remaining DNA strands for the motif in each strand that best matches the seed and the motifs that have already been found.

ALGORITHM GreedyMotifSearch (*DNA*,*t*,*n*,*l*)
//Finds the seed in the first two DNA strands through an exhaustive search and then linearly finds the motif in the remaining lines that matches the seed.
//Input: Two-dimensional Array *DNA* contains all the strands of DNA to be checked
Integer *t* is the number of DNA strands to be checked
Integer *n* is the length of each DNA strand (they all have the same length)
Integer *l* is the length of the consensus string (the motif to be found)
//Output: A set of starting positions for the DNA sequences that produce the best consensus string by finding the set of starting positions that produces the best DNA score

```
1     bestMotif ← (1,1,…,1)
2     s ← (1,1,…,1)
3     for $s_1$ ← 1 to $n – l + 1$
4           for $s_2$ ← 1 to $n – l + 1$
5           if Score(s,2,DNA) > Score(bestMotif, 2, DNA)
6                 $BestMotif_1$ ← $s_1$
7                 $BestMotif_2$ ← $s_2$
8     $s_1$ ← $BestMotif_1$
9     $s_2$ ← $BestMotif_2$
10    for i ← 3 to t
11          for $s_1$ ← 1 to $n – l + 1$
12                if Score(s, i, DNA) > Score(bestMotif, i, DNA)
13                      $bestMotif_i$ ← $s_1$
14          $s_i$ ← $bestMotif_i$
15    return bestMotif
```

3.3.1 Complexity

The complexity of the greedy algorithm is $O(ln^2 + nlt)$ where l is the length of the motif, *n* is the length of the DNA samples, and *t* is the number of DNA samples. So this method has a squared term for the exhaustive search of the first two DNA strands, and then the rest of the program is a linear search. This is much faster than the exponential brute force algorithm and branch and bound algorithm.

4 Experiments and Analysis of Results

To test the brute force and the branch and bound algorithms we created files with varying numbers of DNA strands and varying numbers of lengths. All of the DNA strands are actual DNA sequences from a source that our professor found [2]. We varied the number of rows and the length of the rows in each file, but within each file every row had the same length. Our program does not compare rows with different lengths. For every trial that we ran we recorded the time in milliseconds for the program to produce a result.

First, we started with a small number of rows and successively increased the length of these rows. Then we increased the number of rows and ran tests with those same lengths for this increased number of rows. This shows how the efficiencies of the algorithms change with respect to the length of the rows and the number of rows. Both algorithms have a linear efficiency with respect to the length of the rows, but have an exponential efficiency with respect to the number of rows. Therefore, the time they take to compute a result increases exponentially with the number of rows being tested.

For the branch and bound algorithm we repeated this set of tests with files that had different sets of DNA with the same numbers of rows and row lengths. This is interesting because the efficiency of the branch and bound algorithm varies significantly depending on how many branches of the tree it bypasses. The efficiency of the brute force algorithm does not change with different DNA sequences like the branch and bound algorithm. It is the same for any file with the same number of rows and the same row length because it checks all possibilities every time.

The branch and bound algorithm can be much more efficient than the brute force algorithm if many bypasses are made, but it can also be less efficient if very few bypasses are made. The efficiency of the branch and bound algorithm depends on the sequence of letters in the DNA strands, so testing on different sets of DNA sequences can sometimes yield very different results as shown by Table 1.

We did not run the programs with more than 10 rows of DNA because they took far too long to be of any value. Both the brute force and the branch and bound methods ran for more than 60 minutes with a test file of 15 rows.

Rows	Row Length	Brute Force	Branch and Bound	Branch and Bound, Second Run
5	10	46 milliseconds	62 milliseconds	47 milliseconds
5	20	592 milliseconds	515 milliseconds	390 milliseconds
5	30	4,992 milliseconds	3,853 milliseconds	1,919 milliseconds
5	40	24,758 milliseconds	10,811 milliseconds	3,058 milliseconds
5	60	204,882 milliseconds	27,317 milliseconds	20,031 milliseconds
10	10	111,154 milliseconds	14,883 milliseconds	18,877 milliseconds

Table 1: Runtimes for Brute Force and Branch and Bound Methods

As seen in Table 1, the branch and bound algorithm was always faster than the brute force algorithm. Also, the gap between the running times of the brute force and the branch and bound algorithms increases as the number of rows and the row lengths increase.

Another interesting observation from Table 1 is the change in running times resulting from increasing the length of the rows as compared to increasing the number of rows. Increasing the length of the rows increased the running time fairly linearly as the complexity shows. Increasing the length of the rows by ten increased the running time by a factor of 5 to 10. However, increasing the number of rows drastically increased the running time. Increasing the number of rows by 5 increased the running time of the brute force method by over 2000 times and the branch and bound method by over 200 times. This shows the exponential nature of increasing the number of rows, and this is why it would not have been profitable to test the algorithms with more than 10 rows of DNA. They would have taken far too long to yield any meaningful results.

We also ran the greedy algorithm on test files ranging from 5 rows each containing 10 elements up to 400 rows each containing 120 elements. The results are shown in the table below.

rows	Row length	Run Time (milliseconds)
5	10	0
5	20	15
5	30	32
5	40	31
5	50	32
5	60	47
10	10	16
10	20	0
10	30	32
10	40	32
10	50	31
10	60	47
15	60	78
20	60	62
25	60	63
30	60	47
35	60	46
100	120	63
200	120	116
400	120	296

Table 2: Runtimes for Greedy Method

As seen in Table 2, the greedy algorithm is extremely fast. Its running time is only comparable to the brute force and the branch and bound algorithms for files of 5 rows containing 10 elements. For larger files the greedy algorithm is far more efficient. In fact, the algorithm's runtime does not even change significantly for files as large as 35 rows containing 60 elements-it still runs in 46 milliseconds. There does not seem to be a noticeable pattern in the change in the runtime as rows are added or row length is increased. Even though these file sizes were far too large for the brute force and the branch and bound algorithms to find a solution, these file sizes seem too small for the greedy algorithm to show any significant change in runtime. So we ran the greedy algorithm with much larger files of more than 100 rows each containing more than 120 elements and the runtime was still only in the hundreds of milliseconds, which is still quite efficient.

5 Conclusion

The comparison of the brute force algorithm to the branch and bound algorithm in solving the motif-finding problem showed that while the branch and bound algorithm is more efficient than the brute force algorithm, neither are feasible for even moderate study sizes. An exhaustive search such as these has the advantage of yielding an exact solution, but it cannot produce this solution in a time-efficient way. The greedy algorithm is much more efficient than the other two algorithms, but it gives an approximate solution that may not always be a good solution.

Even though the branch and bound algorithm is faster than the brute force algorithm, it becomes inefficient at roughly the same point as the brute force algorithm. This means that the branch and bound algorithm will not help researchers find solutions to files that are too big for the brute force algorithm to handle, which is what they would like to do. The greedy algorithm can find solutions for files that are much too large for the other two algorithms to handle. This makes it very desirable for researchers who would like to determine solutions for very large files that contain actual DNA information from living organisms. However, the fact that it produces an approximate solution, not an exact solution, presents a problem that would need to be addressed. In short, the brute force and the branch and bound algorithms are inefficient for solving the motif-finding problem, and the greedy algorithm can efficiently produce an approximate solution for very large files.

For future work we would like to investigate ways to improve the approximation generated by the greedy algorithm. The result that the greedy algorithm produces is highly dependent on the seed found in the first two DNA strands, and the approximation will not be good if the seed is not a good match to the rest of the strands. This problem can be addressed by running the greedy algorithm on many different pairs of strands to generate approximations from many different seeds. The best motif found from all of these approximations should be much more accurate than an approximation generated by running the greedy algorithm only one time.

Acknowledgements

We would like to thank Dr. Lydia Sinapova for her assistance in conducting our research and preparing this paper. This research project was conducted as part of the Introduction to Algorithms course at Simpson College.

References

[1] Jones, Neil C., and Pavel A. Pevzner. *An Introduction to Bioinformatics Algorithms*. Cambridge, MA: MIT, 2004. Print.

[2] http://www.ncbi.nlm.nih.gov/nuccore/XM_004915.2?report=genbank