ECE 573 - Data Structures and Algorithms
Spring 2018

DNA Sequence Detector

Project Report - Project# 11
May 6, 2018

Divyaprakash Dhurandhar	Tina Drew	Anirudh Kulkarni	Juhi Tripathi
18007458	035006375	179004206	179001757
dd839	tld95	ak1512	jt866

1 Problem Statement

DNA is complex set of bases that are arranged in string sequences forming a double helix. An organism's full set of DNA including their genes and regulatory sites are included in its genome. Genomics is the scientific field that allows us to study genome mapping, function, structure and evolution. Because genomics focuses on genes collectively and quantitatively, it can be used for applications that differ from traditional genetic studies. The broadened perspective of genes provided by genomics enables the examination of protein production and enzyme operation. Applications for genomics include developing medicine, modeling diseases, and modeling brain functionality. [1,2] Additionally effective DNA sequencing algorithms can be used for DNA tests and hereditary matching.

The abundant possibility of applications of DNA sequencing has lead to a demand for efficient DNA and genome modeling sequences. In our project we intend to implement an effective algorithm to test matches in DNA sequence patterns.

2 Data Structures and Algorithms employed

DNA sequences are comprised of four basics bases, a, c, g, and t, arranged in varying string configurations with sizes that could be billions of characters in length. We can abstractly model DNA sequences by using long sets of strings as a substitute. So if we consider two strings, s1 and s2, we can compare them by examining the set of k characters that they have in common. In order to compare the DNA sequences, we are implementing three algorithms, namely : Brute Force (Naive implementation), Goldman method and Goldman-Bloom method. We implement the mentioned algorithms to facilitate the search of locations of similar pattern within the large sequence of characters (such as that of a DNA). We study and analyse the speed and performance of the three discussed algorithms.

Naively, we can test every k set in s1 against every k set in s2. The basic idea to compare two strings, (using strings as the data structure) is less complex in implementation, but it results in an increased run time cost, which would be extremely problematic for strings that are billions of characters long.

Instead we propose using an algorithm discussed in "*A Practical Guide to Data Structures and Algorithms Using Java*" by Sally and Kenneth Goldman. For the duration of this paper we will refer to this algorithm as the Goldman method. [1] The Goldman method implements the comparison, such that every set of k, is hashed to an index in the hashing table, and we use separate chaining hash table as the data structure. The implementation of Goldman method is discussed further in the report in next subsection.

While the Goldman method serves the purpose of a faster algorithm in comparison to the brute force implementation, we still find a scope for faster execution, using the Bloom Filter as the data structure. While bloom filters do provide a faster result, but it comes at the cost of a probabilistic

result. There is a significant chance of false positives in using Bloom filters, which increases as the data set increases. The implementation is elaborated and discussed in the following subsections.

2.1 Brute Force Algorithm

2.1.1 Description

Generally, the first approach to attempting to solve a problem is the "easy- to-implement" solution. The object of this approach is not efficiency; instead the goal is to find a solution to the problem presented in the simplest way possible. When developing an algorithm this approach is known as the "Brute Force" or "Naive" method.

The most easy implementation to compare strings, is to traverse through both the strings for every possible combination of required set. Sequentially we compare the strings for every possible length of k- characters. But the brute force implementation increases the runtime cost to such an extent, that for a large dataset (like that of a DNA string) it becomes an impossible solution.

Our Brute for implementation of find patterns in the DNA strings involves the following steps:

1. Generate a two random sets of strings s1, and s2 to represent DNA
2. Loop through the length of one string (corpus string : say, s1) and execute the following steps -
    a. Form every possible combination of k-length string in string s1 (in each iteration of the loop) - say, ks1
    (where, k is the user entered sequence matching pattern length)
    b. Loop through the length of other string (pattern string : s2) and execute the following steps -
        i. Form every possible combination of k-length string in string s2 - say ks2 (in each iteration of the loop)
        ii. Check if the k-length string from s2 is equal to k-length string from s1
            if (ks1 == ks2) : increase the count by 1
            Else : continue

2.1.2 Analysis

As evident from the implementation of the brute force algorithm, entire string length of both the strings (corpus and pattern) are traversed to compute the count of matching sequence in strings. Thus, the number of string access dominates the order of growth for these algorithms. The"Brute Force" algorithm requires checking each k-length string in both the strings with each iteration of the loop. Let's consider s1 as the size of string 1 (corpus) and s2 as the size of string 2 (pattern). We are attempting to find sets of k set elements that are the same in both arrays. So we start with an initial set of k elements from s1. For the first loop, we iterate through the s2, s1-k times. Then for the next k set, again, we search through s2, s1-k times. This is repeated for the length of s1 (or s2-k) times which leads to the following equation:

$$O(G(N)) = (s1-k)*(s2-k)$$

As the sizes of the array become larger the value of k is negligible. So the order of growth can be reduced to:

$$O(G(N)) = (s1)*(s2)$$

Thus, the runtime complexity for Brute force, turns out to be quadratic, given by : $O(n^2)$ .This growth rate is not ideal because the algorithm would take a massive amount of time for DNA sequences that are billions of characters in length. The next sections discuss algorithms with more efficient growth rates.

2.2 Goldman Method

2.2.1 Description

The Goldman Method offers an improvement over the Brute Force approach by creating a separate chaining hash table. Initially, for a given reference string, we form a separate chaining hash table using k-set of strings, each hashed to an index, given by the hash function. Instead of comparing the entire reference string with the input string, we use the same hash function and lookup for the k-set of strings in the index that is output by the hash function

Our implementation of the Goldman Method involves the following steps:

1. Generate a two random sets of strings s1, and s2 to represent DNA sequences strands
2. Separate chaining hash table formation :
    a. Create a hash table of length ‘V’ (= length of corpus string / k : in our implementation)
    b. Using one of the strings, (corpus string, say s1) we form every possible set of k-length strings, and for every k-length string, we loop through the following steps :
    c. Compute the hash function of k-length string.

        We use a constant hash function throughout the implementation, which is -

        "*weight of string % number of buckets*"

        Where, weight of string is the sum of ASCII numbers of all the characters in the string

        number of buckets is ‘V’ (length of hash table, or length of corpus string/k)
    d. Push the k-length string into the hash table, at the index computed using the hash function.
3. Comparison :
   Using the other string, s2, we form every possible set of k-length strings, and for every k-length string, we loop through the following steps :

- Compute the hash function of k-length string.
  We use the same constant hash function - "*weight of string % number of buckets*"
- Lookup for the string only at the index (and the chain associated with the index) computed through the hash function.
  - If the string is found; increase the count by 1.
  - Else, continue for the other strings.

2.2.2 Analysis

Using the Goldman method, we are decreasing the search time, by looking at only the index (and the separate chain associated with that index) that the string is hashed to. If s1 is the string used to form the hash table, the runtime cost would include the length of s1 (since, the entire s1 string is traversed in the formation of hash table) and the length of s2 (search through the length of s2). Provide that the search of S is in constant time, the expected time for the Goldmann Method is -

$$O(s1 + s2 + x)$$

(where, x is the substring matches and s1,s2 are the sequence strings)

As compared to the Brute force implementation, the Goldman method provides a much faster execution time. We have reduced the runtime cost from quadratic to linear using the Goldman method over brute force. This method, much like the rapid biosequence matching algorithms of today, involves rapid and efficient hashing implementation. [1]

2.3 Goldman-Bloom Method

2.3.1 Description

One way to improve run time of the Goldman method is to implement Bloom filter. A Bloom filter is “space efficient probabilistic data structure” that involves testing elements and inserting them into the dataset. For this data structure, elements cannot be removed, they can only be added. [1,3]

Using the conventional Goldman method, we store all the k-length strings in the separate chaining hash table. Implementation of bloom filter eliminates the need to store the strings in data structure. Instead of the separate chaining hash table, we use a bit array that stores 0 or 1. For every hash mapping, the corresponding bit at the index is changed (to 1).

The steps involved in our implementation of bloom filter are :

1. Create a bit array of size ‘V’. (where, V = number of buckets)

[NOTE : V will always be less than the length if string : We have taken V=length of corpus string/k]

2. Initialize the array with 0.
3. Array Formation :

Using one of the strings, (say s1) we form every possible set of k-length strings, and for every k-length string, we loop through the following steps :

- Compute the hash function of k-length string.
  We use a constant hash function throughout the implementation, which is -

  "*weight of string* % *number of buckets*"

  where, Weight of string is the sum of ASCII numbers of all the characters in the string

  Number of buckets is the number of buckets in hash table (length of hash table)
- At the array index (computed using the hash function), set the bit to '1'

4. Comparison :
   Using the other string, s2, we form every possible set of k-length strings, and for every k-length string, we loop through the following steps :
   - Compute the hash function of k-length string.
     We use the same constant hash function - "*weight of string* % *number of buckets*"
   - Lookup for the bit at the index (computed using the hash function) in array.
     - If the bit is set to 1; increase the count by 1.
     - Else, continue for the other strings.

2.3.2 Analysis

Bloom filter implementation provides a constant time array access, which is the fastest execution time, we observed during the implementation. For a convenient and simpler implementation, we have used a single hash function ("*weight of string* % *number of buckets*"). Thus, we have reduced the runtime complexity to a constant time, using the bloom filter.

Time complexity for Goldman-Bloom implementation : O(1)

Also, the implementation uses an array of size 'V' (number of buckets), therefore -

Space complexity : O(V)

where, V=length of corpus string / k

But, the reduced run time comes at the cost of a probabilistic result. The major disadvantage of using Bloom filter is that false positives are possible. This means that the filter could indicate that an element or element is in a set when it not. As the data set is increased, the number of false positive results increases. For a lesser number of buckets, the false positive increases quickly, and for a larger number of buckets, there would be a lot of 0-bits. Hence, there should be an optimum number of buckets of the data size to avoid false positives and use the space efficiently. The false positive error rate is given by the following equation:

$$E_{FP} = Y/(X+Y)$$

Here, E_{FP} is the probability of a false positive, Y is the number of substrings classified incorrectly, and X is the number of strings classified correctly. As more elements are added to the filter the probability of false positives increases. Fortunately, the filter will not skip over or miss elements that are part of the set. [1]

There are several benefits to using a Bloom filter including: Bloom filters by design are space efficient. They require constant time lookup regardless of the element already in the set. This filter can be implement in parallel with other processes. They provide tradeoff between memory or space and error rate. [1]

3 Experiment Configuration and Details.

3.1 Code languages:

- All algorithms were implemented in C++.
- The test file generator code was written in Python 3.6.5.

3.2 System Configuration

The system configuration is as follows-

Operating System: Windows 7
Processor: Intel Core i3, 1.80 GHz
RAM: 5 GB
Compiler: Microsoft Visual Studio 2017

3.3 Dataset

In order to test the algorithms we ran each algorithm with the same DNA data strings. A python script was used to create some of our data sets. In the script, we randomly selected characters from the set - {a, c, g, t } and insert them into a file. We have created data strings of varying sizes ranging from 12 to 10,000 characters in increments of 1,000 characters. This datasets are titled DNA_String_**size**.txt and DNA_String_**size**_corpus.txt, where **size** refers to the number of characters in the sequence.

Other data files were download from: http://goldman.cse.wustl.edu/crc2007/projects/. These datasets are smaller in size and were used in the some of the tests from the Goldman Project. [1] These files are titled case**X**-corpus.txt and case**X**-pattern.txt where **X** refers to the case number (total 3 cases)[1]. Due to a very small size, we were able to track the implementation and assert the

results, but we had a very little scope to study the variation in runtime differences, which led us to create our data sets of large string length.

3.4 Test Cases

3.4.1 Case 1 - Order of Growth Classification

Our experiments contained a "corpus" DNA string and a "pattern" DNA string. The corpus string is the reference string that undergoes comparison, and the pattern string is used to determine the k-length pattern sets. The pattern string was implemented to be compared with the corpus string, and the matching sequences in pattern string with reference to the corpus string, are reported. For this test, both the corpus and pattern strings were the same size. "N" is designated as the number of characters in, or size of, our DNA string. For our experiments the the set size or k was equal to 10. The results section of the report, includes all the observations and analysis based on the obtained results.

3.4.2 Case 2 - Accuracy Verification

For this experiment we wanted to observe the accuracy level of the bloom filter when compared to the other algorithms. In order to test the accuracy, we wanted to use a smaller data file, that were downloaded from http://goldman.cse.wustl.edu/crc2007/projects/. The corpus and pattern files were changed each time to match with the case under test. So for case 1 : the case1-corpus.txt and case1-pattern.txt files were used to test the accuracy of the algorithms implemented.

4 Results

4.1 Case 1: Order of Growth Classification

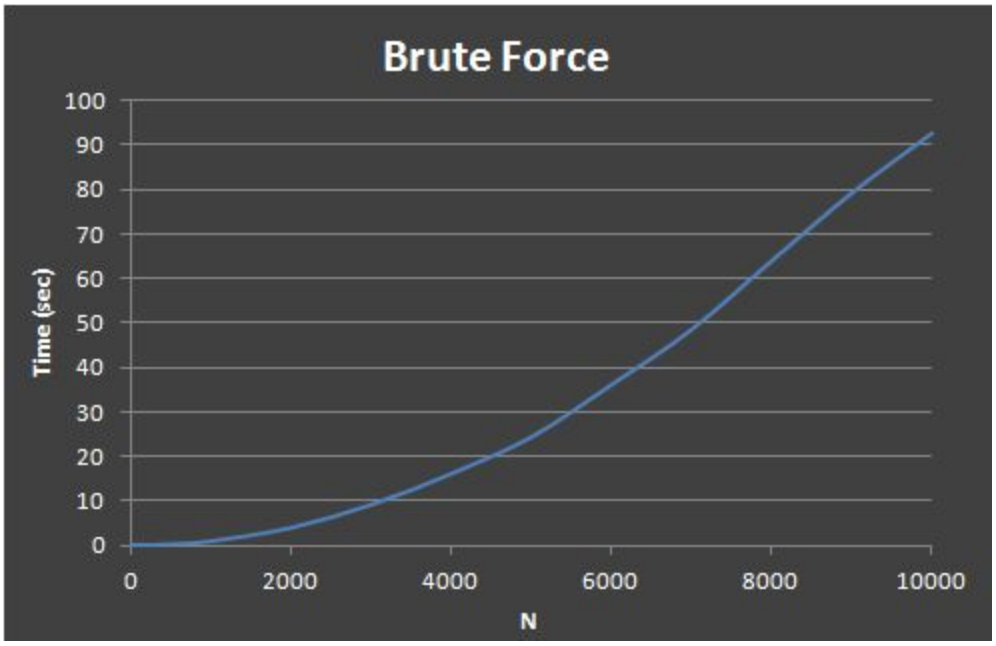
4.1.1 Brute Force

As expected, the brute force algorithm took the longest time to run. When the datasets are 1,000 characters in length the algorithm completes in about 1 second. When both data sets were 10,000 characters in length the algorithm's execution time was almost 100 seconds. So, when both data sets increase by a power of 10 the complete time increases by a power of 100 or N^2. This supports our analysis of worst case order of growth being $O(N^2)$ for this implementation.

Since this algorithm observes each dataset and does not implement hash tables (and thus no chances of collision) it calculates matches with almost 100% accuracy. When the pattern dataset in consideration have multiple occurrences of the k-length string, the count of matches includes the sum of all the corresponding matches in corpus string. This causes an increase in the number of matches more than the string length. This increase in number of matches could be considered as

the result of taking the two strings and traversing the search sequence through both the string lengths, without maintaining an explicit memory space to store the exclusive count of matches.

Brute Force		
N	**time (ms)**	**Matches**
12	0.002	0
500	0.232	0
1000	0.952	2
2000	3.956	7
3000	9.134	11
4000	16.097	29
5000	24.354	28
6000	36.077	37
7000	48.479	32
8000	63.914	52
9000	79.14	92
10000	92.486	97

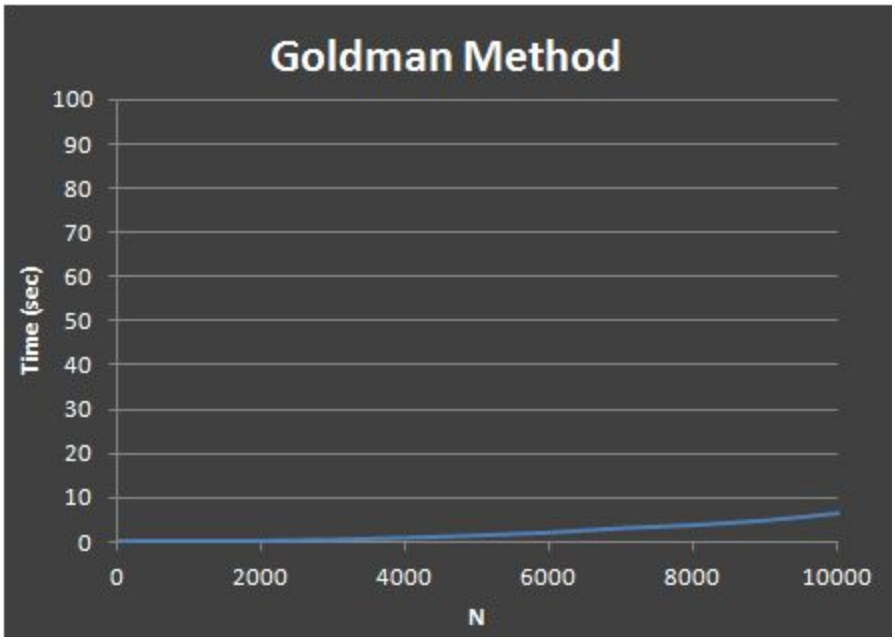


4.1.2 Goldman Method

The Goldman Method executes in a shorter amount of time than the Brute Force technique. For data sizes of 12 and 10000 the execution time is approximately 4 ms. When both data set sets are 10,000 characters long, the execution is just over 8 seconds. This is an order of magnitude smaller than that of brute force.

The Goldman Method shows the same number of matches as the Brute Force technique which means that determines matches with 100% accuracy.

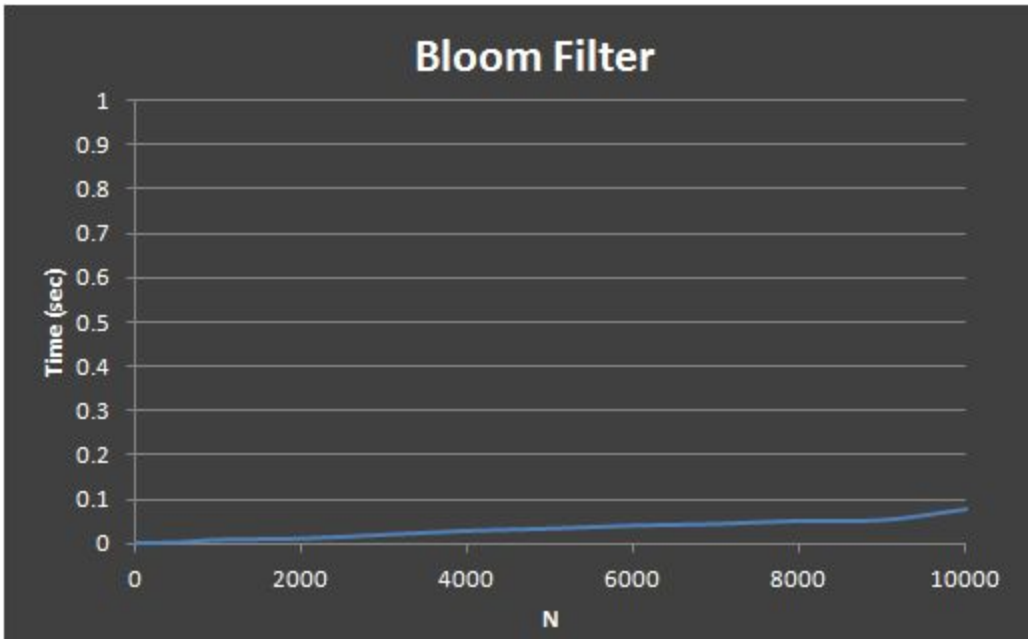
Goldman		
N	**time (ms)**	**Matches**
12	0.004	0
500	0.039	0
1000	0.109	2
2000	0.36	7
3000	0.789	11
4000	1.342	29
5000	2.008	28
6000	3.015	37
7000	3.742	32
8000	4.829	52
9000	6.464	92
10000	8.269	97



4.1.3 Bloom Filter

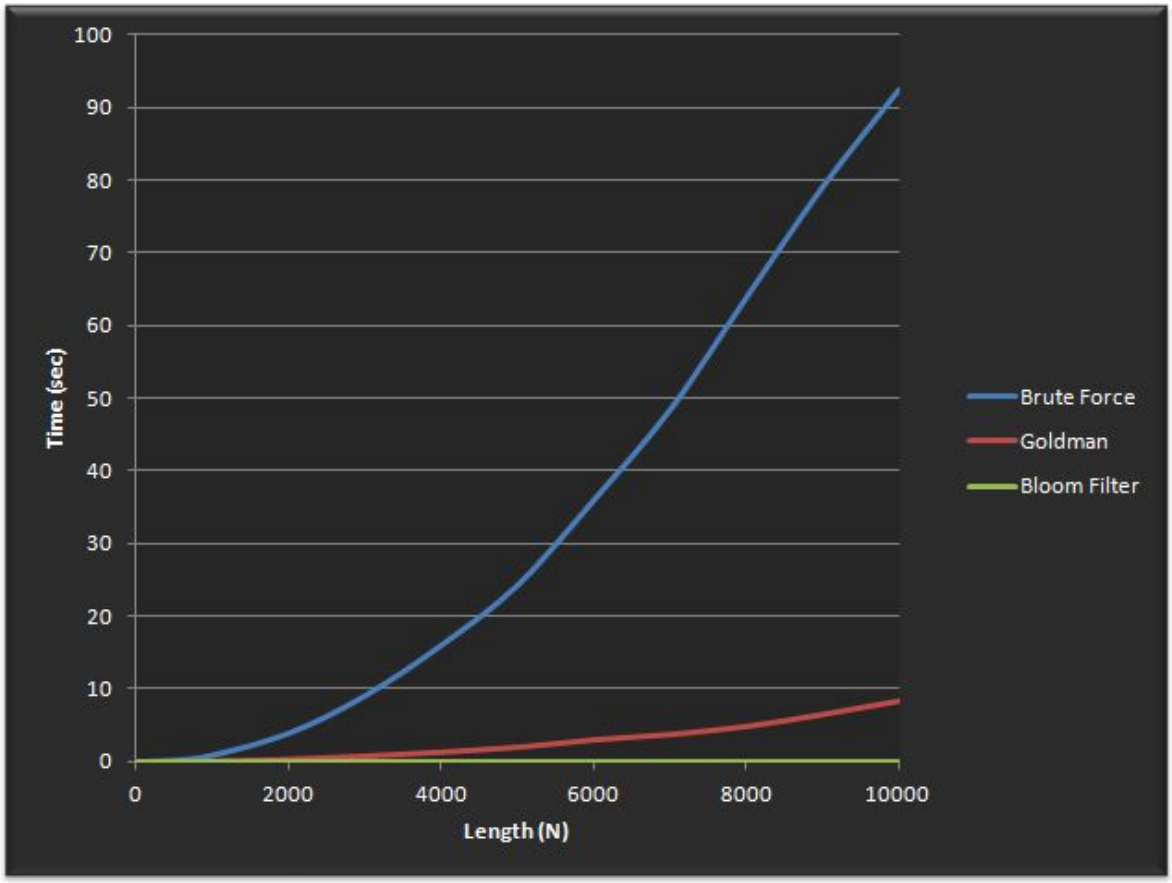
The Bloom Filter has the shortest execution times. Even with both data sets at 10,000 character in length the algorithm only takes 68 ms to run. Unfortunately, as the data set increases, the number of false positive increases; thus the technique is not accurate for large datasets. The large dataset resulted in a high percentage of false positives. In fact, the program showed almost a 100% of the dataset matched despite the fact that the datasets were not equivalent.

Bloom		
N	**time (ms)**	**Matches**
12	0.002	2
500	0.008	490
1000	0.011	969
2000	0.02	1978
3000	0.029	2978
4000	0.034	3967
5000	0.041	4983
6000	0.045	5978
7000	0.052	6990
8000	0.054	7976
9000	0.079	8983
10000	0.068	9986



4.1.4 Algorithm Comparison

Below are the combined results. This graph below shows a comparison between the three algorithms in terms of execution time.



4.2 Case 2: Accuracy Comparison

As stated previously, for the Bloom Filter algorithm, large datasets comprised of 10,000 characters resulted in large false positive rates. The table below show the pattern matches results for smaller corpus and pattern datasets (for k = 3).

	CASE 1	**CASE 2**	**CASE 3**
Brute Force	24	72	40
Goldman	24	72	40
Bloom Filter	24	74	42

The number of pattern matches are same (correct) for brute force and Goldman, method for all the three cases, but for the bloom filter implementation, the number of matches increases with the dataset (length of dataset : case 1 < case 2 < case 3), that asserts an important property of bloom filter; the number of false positive increases with the insertion of larger data set in the bloom filter.

5 Conclusion

After analyzing the results, we can conclude that bloom filter is the fastest lookup implementation, we have come across. Goldman method solves the problem of faster search over brute force, but implementing bloom filter saves both time and space, at the cost of probabilistic results. To summarize our inference in the experiments we have observed -

- Brute Force remains the slowest and most un-feasible solution to implement the sequence detector.
- Goldman method using separate chaining hash table produces faster result as compared to the brute force.
- The implementation complexity could be further refined using bloom filter in conjunction with the Goldman method - which reduces both the space and time complexities, but at the cost of a probabilistic result.

The key factor to be considered in the implementation of the Goldman method and the Goldman - Bloom implementation is the optimum size of data structure (number of buckets in hash table for Goldman method and array size in Goldman - Bloom implementation).

For our implementation, we believe that the Goldman Method is the best of the three approaches, as the Bloom filter is producing a large number of false positives. The cons of using bloom filter could be mitigated using multiple hash functions and collision avoidance methods. As of now, we have used a single hash function, and for the given configurations, we concluded that the Goldman Method offers accuracy as well as efficiency.

6 Applications

- The applications of the discussed algorithms are not limited to genomics or DNA sequencing. They can also be applied as follows:
- Brute Force -
    - To minimise energy costs and train delays for differing railway train control systems.
    - Brute force algorithm on district artery net
- Bloom Filter -
    - Synchronization of Data : Calendar and email synchronization.
    - Chemical Structure Similarity
    - Skip Lists
- Hash Table -
    - Dictionary and fast lookups

7 References

1) Goldman, Sally A. and Goldman, Kenneth J. *A Practical Guide to Data Structures and Algorithms Using Java*. http://goldman.cse.wustl.edu/crc2007/projects/
2) Genomics. Retrieved, March 6, 2018 https://en.wikipedia.org/wiki/Genomics#Applications_of_genomics
3) Bloom Filter. Wikipedia. Retrieved, May, 1, 2018 https://en.wikipedia.org/wiki/Bloom_filte

8 Contributions

Tina Drew

- Designed and implemented of Alpha Code - Brute Filter.
- Designed and implemented test code to create random DNA Strings
- Contributed in documentation and reports.
- Modified Beta Test code for Brute Force, Goldman Method, and Bloom Filter Applications
- Ran alpha and beta test on code versions
- Debugged Brute Force, Goldman, and Bloom Filter Beta Code

Juhi Tripathi

- Designed and implemented of Goldman

- Designed and implemented of Goldman - Bloom method.
- Debugged of Goldman and Goldman - Bloom method
- Contributed in documentation and reports.
- Debugged BruteForce, Goldman, and Bloom Filter codes for alpha testing
- Performed testing on Brute Force, Goldman Method, and Bloom Filter implementations

Divyaprakash Dhurandhar

- Design and implementation of Beta test code for Brute force algorithm.
- Generating results based on brute force substring matching.
- Used practical data sets obtained from Kaggle.com to obtain time complexity for brute force.
- Contributed in documentation and reports.

Anirudh Kulkarni

- With the design implementation of the Brute force Algorithm and Application Related to the The Algorithms used.
- Ran alpha testing on Brute Force implementation
- Debugged Brute Force Code
- Contributed in documentation and reports.