DNA Trace Reconstruction

Udvas Basak, Rishi Roy, Hrishi Narayanan

1 Introduction

The primary objective is to reconstruct the original string from m random traces(subsequences) of the original string, where each trace is obtained by independently deleting elements of the original string. The algorithm needs to be optimized so that the reconstruction takes place in lower order time.

We investigate a number of algorithms for the abovementioned problem, and arrive at conclusions regarding their feasibility and applicability in different situations and values of parameters. The most preliminary algorithm for this is the Bitwise Majority Algorithm(BMA) which we investigated, and then compared the performance of the modified algorithms with BMA. All the investigations include simulations, followed by analysis of the results.

Even though the objective is to work with DNA, which is a quaternary alphabet, our simulations start with a binary alphabet, which we can very easily transport into a quaternary one.

2 Terminologies

The original string that has to be reconstructed is referred to as the ***transmittedString***.

A substring of consecutive 0's or 1's is referred to as a ***run***. A substring with only 0's is a ***0-run*** and with only 1's is a ***1-run***.

Every string that is obtained after independently deleting each element with a predefined ***deletion probability***, is referred to as a ***trace*** or a ***transmission***.

The array, each of whose rows is one trace obtained from the same transmittedString, is referred to as the ***DataSet*** or the ***Binary Dataset Matrix***. The matrix has l rows, where l is the number of traces, and c columns, where c is the length of the longest trace generated(sometimes padded with extra characters).

Whenever percentage error is being calculated, 1000 independent samples are generated, and the percentage of samples that return a wrong answer is being calculated.

The Levenshtein distance is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number

of single-character edits (insertions, deletions or substitutions) required to change one word into the other. In this report, it is used interchangeably with ***Edit Distance***.

3 Bitwise Majority Algorithm

3.1 Main Algorithm

With a thorough understanding of the algorithm, the BMA is coded.

The pseudo code is as follows:

```
Take the binary dataset matrix with size (l,c) as input

Create an array of pointers #p of length l, initialized to zero.
Create a binary array #transmittedString of length c.

Loop k from 0 through c-1:
    Take a majority vote #mv along all members[i, p[i]] ##i runs from 0 to
    l.
    transmittedString[k] <- mv
    If p[i] equals mv
        Increment p[i] by 1

Return transmittedString
```

Listing 1: BMA Algorithm PseudoCode

3.2 DataSet Generation

For simulating the results and also to test whether the algorithm can successfully reconstruct the trace, we need to generate a random dataset, from a randomly generated arbitrary binary string. A code is generated for the same, which takes the parameters *length of the original string(c), no. of traces to generate(l), and the deletion probability(p).*

The pseudocode is as follows:

```
Generate a random c-length binary string #rb.
transmittedString <- rb

Generate an empty matrix of size(l,c) #dataset.

For each trace:
    Generate the trace by independently deleting elements from
   transmittedString with probability p.
    If length of trace is less than c, pad with zeros at the end.
    dataset[i] <- trace
```

```
10
11 Return transmittedString, dataset
```

Listing 2: DataSet Generation for BMA

3.3 Results of the Analysis

For the analysis, we shall run the algorithm for a range of lengths, a range of the number of traces, a range of deletion probabilities, and investigate the percentage error and the time taken for the algorithm to work.

The length ranges from 10 to 100, in steps of 5.
The number of traces range from 3 to 51 in steps of 3.
The deletion probabilities are (0.1, 0.3, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 10.0).

The results are analyzed and the plots are presented.

3.4 Important Features and Results

- Since the algorithm can work on random strings, we can input a string of any kind, and can expect a correct result when *the conditions are favourable.*
- The algorithm has a self-correcting property.

4 Approximate Trace Reconstruction

The original BMA algorithm has needs a large number of traces to accurately predict the transmitted string. In fact, the accuracy drops drastically if the number of traces provided is low. For that, we need to devise algorithms that may compromise a little on the accuracy, but will take lesser number of traces to find the original string.

To investigate this, we studied two algorithms that can work on a *limited number of strings*[1]. The algorithms have been referred to as Warmup Algorithms in the relevant literature, and hence, we shall also use the references '**Warmup Algorithm #1**' and '**Warmup Algorithm #2**' for the same.

[1]These strings must follow some prerequisites for the algorithms to work.

5 Warmup Algorithm #1

5.1 Prerequisites and Assumptions

Proposition: Let X be a string on n bits such that all of its runs have length at least $log(n^5)$. Then X can be ϵn-approximately reconstructed with $O(log(n)/\epsilon^2)$ traces.

Claim: The first algorithm ϵn-approximately reconstructs a string with long runs using $O(log(n)/\epsilon^2)$ traces by scaling an average of the run length across all traces.

5.2 Main Algorithm

With a thorough understanding of the algorithm, the BMA is coded.

The pseudo code is as follows:

```
Take the binary dataset matrix with size (l,c) as input
Take the deletion probability #p

Check if all the traces have same number of runs.
k <- common number of runs

Loop i from 0 through k:
    Compute the average of the lengths of the i-th runs in all the traces
    b <- ceiling(average/(1-p))
    transmittedString <-+- ((bit in i-th run) x b)

Return transmittedString
```

Listing 3: Warmup Algorithm #1

5.3 DataSet Generation

The dataset generation in this is very involved. The first requirement is that all the runs must be of length at least $log(n^5)$, so we have to generate a pseudorandom sequence with that property. For that, to create the original string, we randomly choose a bit to start with, then keep appending alternating runs of random lengths between $log(n^5)$ and $2 \times log(n^5)$.Also, to make sure that the minimum number of runs is 5, the minimum length of the original string is 40(See Fig 1). The pseudocode for the same is as follows:

```
Take length of transmittedString c and probability p as input

Initialize empty string #transmittedString
Choose a random bit -> #bit

```

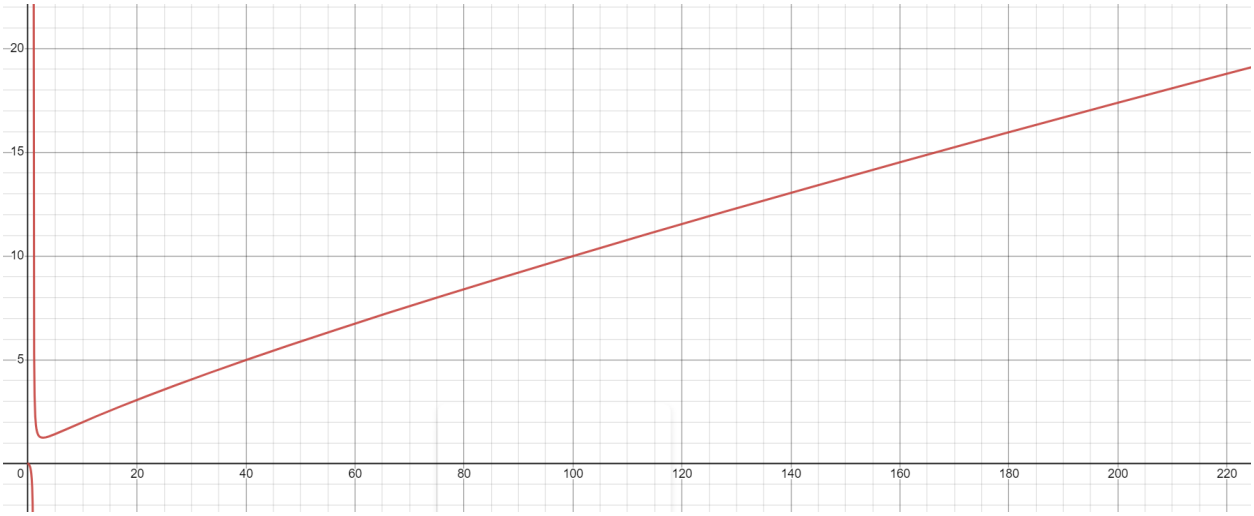


Figure 1: A plot of $x/log(x^5)$. This will approximately give the number of runs for different values of length x.

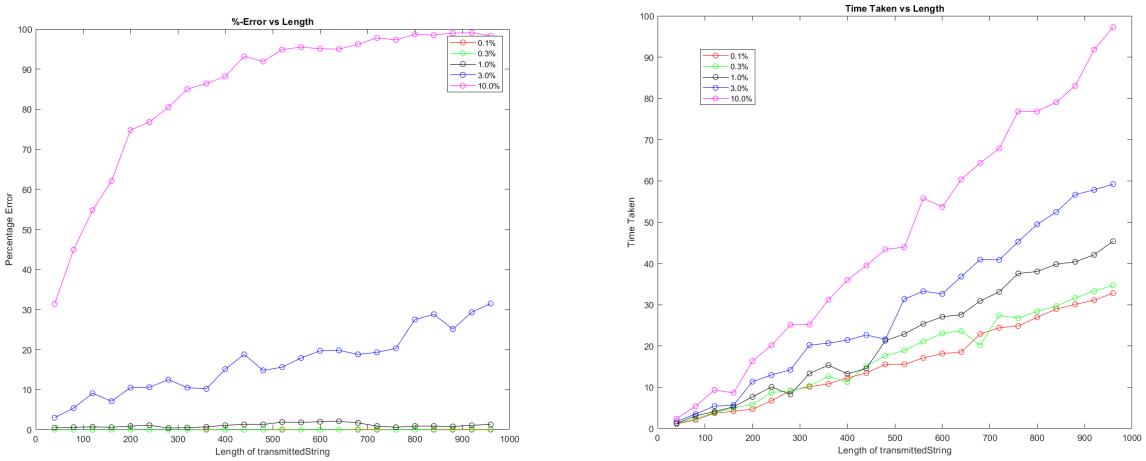
```
Infinite Loop:
    Invert #bit
    Choose a random length #rl between a = log(n^5) and 2a.
    If length(transmittedString)+rl less than c:
        Append rl number of bits to transmittedString
    Break and append bits otherwise

Return transmittedString
```

Listing 4: String Generation for WA#1

For making the whole dataset, we just need to do the deletions and make sure that the number of runs in all the traces is same. The pseudocode is as follows: The pseudocode is as follows:

```
Take transmittedString and probability p as input
transmittedString <- rb

l is calculated from the formula in Fig 1.

Generate an empty matrix of size(l,c) #dataset.

For each trace:
    Generate the trace by independently deleting elements from
   transmittedString with probability p.
    Check if trace has same number of runs as 0th trace.
    If Yes:
        dataset[i] <- trace

Return
```

Listing 5: DataSet Generation for WA#1



(a) % error vs Length

(b) Time Taken vs Length

Figure 2: Independent Simulation of Warmup Algorithm #1

5.4 Results of the Analysis

5.4.1 Properties of WA#1

For the analysis, the algorithm is run independently to capture its crude behaviour. For this, the algorithm is fed with strings of different lengths with different deletion probabilities and the % error and the time taken for the algorithm to run is calculated.

The length ranges from 40 to 1000, in steps of 40.
The deletion probabilities are (0.1, 0.3, 1.0, 3.0, 10.0).
The value of epsilon is taken to be 0.5, so by the proposition, X can be 0.5n-approximately reconstructed.

The plots are presented in Fig 2.

Considering Fig 2a, it is seen that %-error keeps increasing almost linearly as the length of the transmittedString increases, and for high deletion probabilities, it saturates around the 100% mark at about 440. For a deletion probability of 3% also, the plot is almost linear, and it is on and around the 30% mark at even 1000 length. Notably, the test is considered correct only if the original and the recalculated string match completely, so it is still possible that the approximate reconstruction is achieved with much higher precision in these cases.

In Fig 2b, all plots are linear, with different slopes for different deletion probabilities. This is indirectly because for higher deletion probability, the algorithm has to work on higher number of traces(number of traces sampled depended on the base of the log, which is lower for higher deletion rates, hence, more traces are sampled for higher deletion rates).

5.4.2 Comparison of WA#1 and BMA

A second part of the analysis compares the behaviour of BMA and WA#1. The input string will follow the prerequisites of WA#1, and the same number of traces would be passed into both the algorithms. The situation is plotted for a large number of cases.

The length ranges from 40 to 1000, in steps of 20.
The deletion probabilities are (0.1,0.3,0.5,1.0,1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 10.0).
The values of epsilon are (0.1,0.3,0.5,1.0).

The time taken, % error, and average edit distance from the correct string is calculated, and the performances are plotted.