ISLAMIC UNIVERSITY OF TECHNOLOGY

Candidate Gene Prioritization Using Unique Pattern Indexing and Mapping Techniques

Authors:

Kazi Mahbub Mutakabbir (104433)
Shah S Mahin (104440)

Supervisor:

Md. Abid Hasan

Lecturer

Department of Computer Science and Engineering

Islamic University of Technology

A thesis submitted in partial fulfilment of the requirements

for the degree of Bachelor of Science in Computer Science and Engineering

Academic Year: 2013-2014

Department of Computer Science and Engineering
Islamic University of Technology.

A Subsidiary Organ of the Organization of Islamic Cooperation.

Dhaka, Bangladesh.

October, 2014

Declaration of Authorship

This is to certify that the work presented in this thesis is the outcome of the analysis and investigation carried out by Kazi Mahbub Mutakabbir and Shah S Mahin under the supervision of Md. Abid Hasan in the Department of Computer Science and Engineering (CSE), IUT, Dhaka, Bangladesh. It is also declared that neither of this thesis or any part of this thesis has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of reference is given.

Authors:

____

Kazi Mahbub Mutakabbir (104433)

____

Shah S Mahin (104440)

Supervisor:

____

Md. Abid Hasan
Lecturer
Department of Computer Science and Engineering
Islamic University of Technology (IUT)

Abstract

"Prioritizing the candidate gene is amongst the notable work in bioinformatics. Techniques have been applied to reduce the number of promising genes for a certain disease. Previous works were done by using PageRank and HITS algorithm on graph based network. However using frequent pattern mining this prioritizing can be made more efficient. In this paper, we propose four algorithms. The first one indexes the unique sequences of length four using an integer value. The second algorithm finds the frequency of the frequent patterns of various lengths by searching through the integer values instead of the patterns themselves. Third one weights the candidate gene in compare with the genes of database. Fourth algorithm creates the graph network and ranks the candidate gene. All this is done highly efficiently by the use of mapping techniques e.g. HashMap. Due to its highly frugal nature, the proposed algorithm can reduce typical memory usage by 37.5% at the very minimum."

Table of Contents

Abstract 2
Table of Contents 3
List of Figures 4
List of Tables 5
Chapter 1: Introduction 6
Chapter 2: Literature Reviews 9
2.1 Apriori Algorithm: 9
2.2 GSP (Generalized Sequential Pattern): 11
2.3 Prefix Span: 12
2.4 PageRank Algorithm: 12
2.5 HITS (Hyper Induced Topic Search): 14
2.6 Directed Graph: 14
2.7 Dempster-Shafer Theory 15
2.8 Motivation: 16
Chapter 3: Proposed Algorithm 17
3.1 Unique Pattern Indexing 17
3.2 Searching Frequent Pattern: 19
3.3 Displacement Based Weighting Algorithm: 20
3.4 Graph Generating and Ranking Algorithm: 23
Chapter 4: Materials and Methods 24
4.1 Materials: 24
4.2 Methods: 25
Chapter 5: Performance Evaluation 28
5.1 Experimental Results (Mining Frequent Pattern): 28
5.2 Graph Generation and Weight Assignment: 30
5.3 Simulation Environment: 33
Chapter 6: Discussion & Conclusion 34
6.1 Discussion: 34
6.2 Conclusion: 34
References 36

List of Figures

1 – *GSP Algorithm* ……………………………………………………………………………..…..……….. *11*

2 – *PageRank Scenario* …………………….…………………………………………………………….….. *13*

3 – *A Directed Graph* ……......………………………………………………………………………….......… *14*

4 – *Flowchart of Dempster-Shafer Theory* ……...……………………………………………….……….. *15*

5 – *Graph Generation Code* ………………………......…………………………………………….... *23*

6 – *Sample Dataset* ….……………………………………………………………………………………….. *27*

7 – *Output Graph Networks* ….………………………………………….……………………….….. *32*

List of Tables

I – *Item Set* …………………………………………………………………………………………… *9*

II – *Items and Corresponding Support Values* ……………………………………………………… *10*

III – *Pair of frequent Items* ……………………………………………………………………………. *10*

IV – *Support Value* …………………………………………………………………………………. *10*

V – *Unique Pattern Index* …………………………………………………………………………. *17*

VI – *Sample Dataset* ………………………………………………………………………………. *24*

VII – *Disease Gene Database* ……………………………………………………………………. *25*

VIII – *Simulation Results Showing Frequent Patterns of Length 4, 8, 12 and 16* …………… *29*

IX – *Relative Frequency for ACTR3BP6 (hiv_1)* ……………………………………………………. *30*

X – *Weights of the tested Disease Genes* ……………………………………………………………. *31*

XI – *Weights of Genes when compared with an unrelated disease* ………………………………. *31*

Chapter 1: Introduction

Bioinformatics is the field of science in which biology, computer science and information technology merges to form a single discipline. It is the emerging field that deals with the application of computers to the collection, organization, analysis, manipulation, and sharing of data to solve biological problems on molecular level. In comparison with other areas of science, Bio-informatics represents a new area of growth. In recent years, there has been a significant change in the field of bioinformatics due to the advance methods applied in computation. Applying these methods we came to know the answers of some of the ever-alluding biological questions which were previously unknown.

Still many things are there that are unanswered and researchers are trying to find a solution [9]. Answering these questions requires that investigators will deal with large, complex data sets (both public and private) in a rigorous fashion to reach valid, biological conclusions. While working with any sector of genomic sequences we must handle millions of data sets. For certain disease from the database we can see that many genes are associated with that particular disease, challenging task is to find out the most prominent gene that is responsible for that disease and rank the rest of the genes associated with the disease, this is the basic concept of gene prioritization and before ranking genes we need to find the solution from the dataset of a gene that is by mapping or finding the frequent pattern that is dominating the gene. This also needs to be analyzed in an efficient way [1] this is also a major challenges in this field. We can give many examples of the importance of gene ranking or prioritizing, data mining and finding frequent pattern from the given sequences, such as in case of disease diagnosis and other important sectors these play a vital role. For a particular disease if we can find that it is occurring for a certain gene (there are many genes but this one is prominent) and of that gene which pattern in the gene then we can work only on that to eradicate that disease, this particular pattern can be found through data mining, again in case of agriculture if we can find that if a pattern frequently occurs in the sequence then it leads to maximum growth of the crops. Research for finding patterns is on process for a long period of time and many came up with many solutions and the work is still going on. We can naturally say that efficient solution of data handling will lead the world to a new dimension.

The scientific world could not reach to a certain stage where they can declare they have reached the ultimate level of efficiency in Candidate gene prioritization. So ranking these genes for finding which gene is most prominent is still considered as one of the remarkable works of this field. Normally every now and then new genes are being discovered and researchers are finding new relation between gene and diseases. In normal cases similarity between genes is derived from one or more types of known information about genes such as functional annotations [11],

but due to lack of information researchers moved to another approach that is of building networks, still we cannot considered it as the most efficient one. Again we should have to consider this that each gene consists of large size and in case of DNA sequence only A,C,T,G makes a nucleotide and so it is very natural that numerous combinations and permutations of A,C,T and G's will be repeated many times[3]. Thus, the importance of candidate gene ranking and then recognizing the correct DNA sequence pattern is easily understandable. There are a lot of aspects of gene that needs to be discovered for proper analysis. Reasons and causes of all diseases can be found if the data are properly analyzed and sorted. Analyzing the gene and ranking it then from the sequence we can find the maximum weighted rank.

Now we will go through some of the methods that have been implemented in this two fields (gene prioritization and pattern mining/data analysis). Here we will discuss about Hyperlink-Induced Topic Search (HITS), PageRank, GSP (Generalized Sequential Pattern), MacosVspan.

Initial works in the field of gene prioritization has been done by HITS [16] and PageRank [15]. PageRank is an algorithm that is being used by Google to rank their page according to the weight provided on the page, this concept is also used in case of gene ranking this method is normally used for building and getting result from the network. Another method that was used to be applied is HITS which used to work on the number of clicks and visits on that page, this method is not very efficient so later PageRank got the priority.

Early works in the field of finding frequent pattern and data mining were mainly based on Apriori algorithm [5, 6]. This algorithm is quite straightforward in its approach. It generally finds all the repeating frequent patterns in the sequence. It is better in the sense that it picks out all the repeating elements and places it in the database. The disadvantage is that it takes a lot of computational time to complete the data transformation of the repeating sequences. Summary of the apriori algorithm is: super pattern of a non-frequent pattern is not frequent [2]. Apriori algorithm was easy but not efficient, to introduce the efficiency factor need for another method arise. So after apriori algorithm came GSP (Generalized Sequential Pattern) with a view to removing some of the disadvantages of the algorithm [6]. It doesn't directly list the information in database. It first scans the general candidate solutions and this scanning happens in a continuous manner. After the scanning then it is tested. Still it has the limitations as it needs to scan large number of database and so it affects the memory.

Then, a more efficient algorithm was introduced which was prefix span [6]. The concept is that it examines only the postfix subsequence after it's completed or taken into account of the prefix subsequence. It follows the recursion pattern which can be used once at a time so it was still not enough to fulfill our goal. Later MacosVspan and improved version of MacosVspan was

introduced which helped to access and manipulate the database faster using fixed length contiguous sequence [7].

Now to make this gene prioritization efficient we used frequent pattern finding. We develop an algorithm where we will get a gene (unknown) as the input, then we will analyze that particular gene sequence the subsequent nucleotide sequences of a given size can be identified within a particular DNA sequence using an assigned numerical value. We'll also store their repetition count in a memory efficient manner by using ASCII byte-encoding, and the most repeating sequence(s) of the given length will be provided as output. Now for that candidate gene we will find the frequency of the sequence of particular length and then we will find the weight by comparing it with the displacement and then dividing it with the number of sequences. We will have same information about the other genes so after getting weight of the candidate gene we can put this gene into different networks (graph). This network will be a directed graph where direct all gene nodes will point towards a disease node and in a network there will be only one disease node gene nodes will be the gene that are associated with the disease. We will compare it with different networks and try to rank the candidate gene with their weights in different network. For larger weight we will say that this candidate gene is most prominent and if we get a low weight value then we can say that for that particular disease that gene is not that much prominent. To find a responsible gene for a disease is a hectic job and there is still scope for developing the procedures, our proposed method is one of them and it will pave the way for further development in this field.

Chapter 2: Literature Reviews

We have studied few algorithms and methods that are related to the candidate gene prioritization and frequent pattern finding and data mining. In the following we tried to describe the concepts of some of the algorithms, along with their perceived shortcomings/limitations.

2.1 Apriori Algorithm:

This algorithm is quite straightforward in its approach. It generally finds all the repeating frequent patterns in the sequence. Apriori uses a "bottom up" approach, where frequent subsets are extended one item at a time (a step known as *candidate generation*), and groups of candidates are tested against the data. The algorithm terminates when no further successful extensions are found. Assume that a large supermarket tracks sales data by stock-keeping unit (SKU) for each item: each item, such as "butter" or "bread", is identified by a numerical SKU. The supermarket has a database of transactions where each transaction is a set of SKUs that were bought together.

Let the database of transactions consist of following item sets:

Table I: Item Set

Item sets
{1,2,3,4}
{1,2,4}
{1,2}
{2,3,4}
{2,3}
{3,4}
{2,4}

We will use Apriori to determine the frequent item sets of this database. To do so, we will say that an item set is frequent if it appears in at least 3 transactions of the database: the value 3 is the *support threshold*.

The first step of Apriori is to count up the number of occurrences, called the support, of each member item separately, by scanning the database a first time. We obtain the following result:

Table II: Items and Corresponding Support Values.

Item	Support
{1}	3
{2}	6
{3}	4
{4}	5

All the item sets of size 1 have a support of at least 3, so they are all frequent.

The next step is to generate a list of all pairs of the frequent items:

Table III: Pair of frequent Items.

{1,2}	3
{1,3}	1
{1,4}	2
{2,3}	3
{2,4}	4
{3,4}	5

The pairs {1, 2}, {2, 3}, {2, 4} and {3, 4} all meet or exceed the minimum support of 3, so they are frequent. The pairs {1, 3} and {1, 4} are not. Now, because {1, 3} and {1, 4} are not frequent, any larger set which contains {1, 3} or {1, 4} cannot be frequent. In this way, we can *prune* sets: we will now look for frequent triples in the database, but we can already exclude all the triples that contain one of these two pairs:

Table IV: Support Value

Item	Support
{2,3,4}	2

In the example, there are no frequent triplets -- {2, 3, 4} is below the minimal threshold, and the other triplets were excluded because they were super sets of pairs that were already below the threshold.

We have thus determined the frequent sets of items in the database, and illustrated how some items were not counted because one of their subsets was already known to be below the threshold.

2.2 GSP (Generalized Sequential Pattern):

The basic structure of the GSP algorithm for finding sequential patterns is as follows:

The algorithm makes multiple passes over the data. The first pass determines the support of each item, that is, the number of data-sequences that include the item. At the end of the first pass, the algorithm knows which items are frequent, that is, have minimum support. Each such item yields a 1-element frequent sequence consisting of that item. Each subsequent pass starts with a seed set: the frequent sequences found in the previous pass. The seed set is used to generate new potentially frequent sequences, called candidate sequences. Each candidate sequence has one more item than a seed sequence; so all the candidate sequences in a pass will have the same number of items. The support for these candidate sequences is found during the pass over the data. At the end of the pass, the algorithm determines which of the candidate sequences are actually frequent.

The main difference between Apriori and GSP is the generation of candidate sets. Let us assume that **A → B** and **A → C** are two frequent 2-sequences. The items involved in these sequences are (A, B) and (A, C) respectively. Under normal circumstances, the candidate generation in a usual Apriori style would give (A, B, C) as a 3-itemset, but in the present context we get the following 3-sequences as a result of joining the above 2- sequences:

A → B → C, **A → C → B** and **A → BC.**

Figure 1: *GSP Algorithm*

The candidate–generation phase takes this into account. The GSP algorithm discovers frequent sequences, allowing for time constraints such as maximum gap and minimum gap among the sequence elements. Moreover, it supports the notion of a sliding window, i.e. of a time interval

within which items are observed as belonging to the same event, even if they originate from different events.

2.3 Prefix Span:

The concept of prefix span is that it examines only the postfix subsequence after it's completed or taken into account of the prefix subsequence.

Given two sequences $\alpha = a_1 a_2 a_3 \ldots a_n$ and $\beta = b_1 b_2 b_3 \ldots b_m$, $m < n$, sequence β is called a *prefix* of α if and only if:

$$b_i = a_i, for\ i \leq m - 1;$$

$$b_m \subseteq a_m.$$

Example:

$$\alpha\ = < a(abc)(ac)d(cf) >$$

$$\beta\ = < a(abc)a >$$

This algorithm needs to be re-run for every sequence, which is why there still remains the problem. Thus later MacosVspan was introduced to make this access faster using fixed length contiguous sequence [7].

Going through the algorithms we understood that GSP succeeded Apriori algorithm, still it was not sufficient, and so Prefix Span and MacosVspan was introduced. Later two algorithms has the benefit of accessing the sequence in a fast manner. But the goal of memory efficiency and fast access both were not meet together in those algorithms. In our work we tried to come up with the solution of making it memory efficient as well as fast accessing.

2.4 PageRank Algorithm:

PageRank is an algorithm that is being used by Google to rank their page according to the weight provided on the page.

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites [17].

PageRank is a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. PageRank can be calculated for collections of documents of any size. It is assumed in several research papers that the distribution is evenly divided among all documents in the collection at the beginning of the computational process. The PageRank computations require several passes, called "iterations", through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value. A probability is expressed as a numeric value between 0 and 1. A 0.5 probability is commonly expressed as a "50% chance" of something happening. Hence, a PageRank of 0.5 means there is a 50% chance that a person clicking on a random link will be directed to the document with the 0.5 PageRank.

In the following we explain the general equation for the PageRank algorithm:

EQUATION:

P(1) = P(2) + P(3)+ P(4)

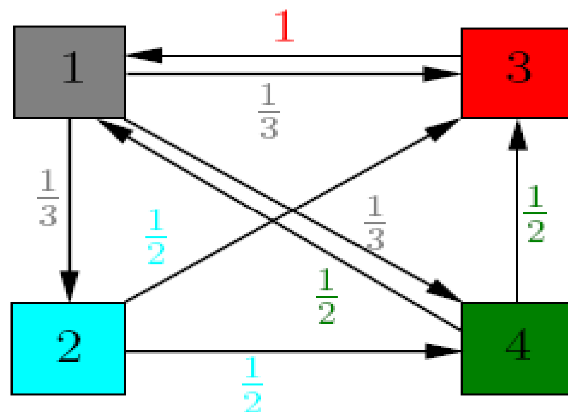

Figure 2: *PageRank Scenario*

Let us denote by *A* the transition matrix of the graph,

$$A = \begin{bmatrix} 0 & 0 & 1 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{bmatrix}$$

.

2.5 HITS (Hyper Induced Topic Search):

HITS is an algorithm that generally ranks the number of pages in the web by counting the number of clicks that has been hit on that page.

In the HITS algorithm, the first step is to retrieve the most relevant pages to the search query. This set is called the *root set* and can be obtained by taking the top n pages returned by a text-based search algorithm. A *base set* is generated by augmenting the root set with all the web pages that are linked from it and some of the pages that link to it. The web pages in the base set and all hyperlinks among those pages form a focused subgraph. The HITS computation is performed only on this *focused subgraph*. According to Kleinberg the reason for constructing a base set is to ensure that most (or many) of the strongest authorities are included.

2.6 Directed Graph:

In graph theory a graph is considered to be as a directed graph if a set of objects (called vertices or nodes) are connected together, where all the edges are directed from one vertex to another.A directed graph is sometimes called a digraph or a directed network.

One can formally define a directed graph as G = (N, E), consisting of the set N of nodes and the set E of edges, which are ordered pairs of elements of N.

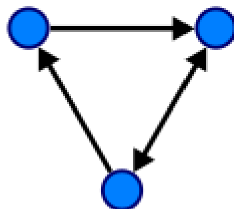

Figure 3: *A Directed Graph.*

2.7 Dempster-Shafer Theory

This theory is related with probability, in case of probability we generalize the whole sample space and assume that it is known to us. By normalization we make it as 1(one), but in this theory a belief works that is we are not aware about the whole sample space and in case of normalization there is a chance that total probability might be less or greater than one. The **Dempster–Shafer** theory (DST) is a mathematical theory of evidence. It allows one to combine evidence from different sources and arrive at a degree of belief (represented by a belief function) that takes into account all the available evidence [20].

The belief value can depend on the type of objects we are dealing with. We can also find out the conditional probability with the Dempster-Shafer theory given that the sample space is that we are dealing with is considered as a complete one.

A flow-chart representing the activities that we need to perform to find the belief function and corresponding belief-value is given in the following:

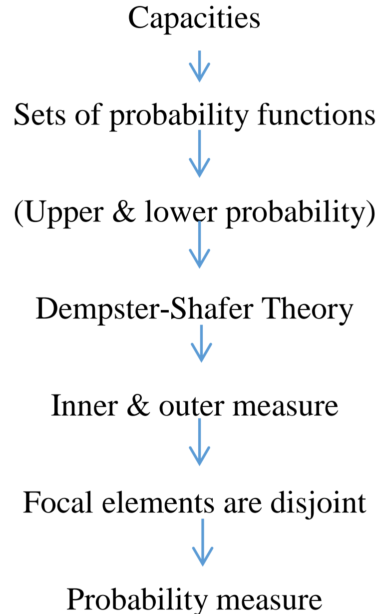

Figure 4: *Flowchart of Dempster-Shafer Theory*

2.8 Motivation:

Candidate Gene Prioritization and Frequent pattern finding are still considered as one of the areas that the researchers are willing to work. There are many aspects in that field that has the scope for development. We tried to combine this two aspects and making the ranking system of the gene a better one. An algorithm to find the frequent pattern has already been developed by us and by realizing that this can be implemented on finding the candidate gene we tried to develop a new algorithm.

Recent years many high throughput technologies that survey a large number of genes have been developed for elucidating the genetic factor of common disease, challenge is that they produce a large number of data set , that is why we tried to combine this two aspects, to handle this large dataset efficiently we used mapping technique and with gene prioritization we reduced the number of promising gene of a disease.

Mapping has always been a favorite tool for database designers to make data access much more efficient and if the data handling mostly consists of unique data, Mapping is one of the most invaluable techniques.

In computing, a hash table (also hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found. [10]

Thus, taking the concept of HashMaps (Hash tables that uses one-to-one mapping) we wanted to devise an algorithm that will search frequent patterns occurring within a single genetic sequence and do the computation in such a manner that the size of the database doesn't increase the time complexity.

Methods of Prioritizing gene are still need to be developed, using graph network many have given methods, we saw an opportunity that in the existing network of this system we can make a new method by implementing the existing algorithm of frequent pattern finding. Little modification in the network can give us a way better scope to implement our method.

Chapter 3: Proposed Algorithm

In order to reduce computational complexity as well as space complexity, we've put two algorithms in place. The first algorithm will take a DNA database file (collected from GenBank, NCBI etc.) of a sample species and then encode it using numerical values. We refer to this algorithm as "Unique Pattern Indexing".

The second algorithm will then find the frequent patterns themselves and map them to appropriate values. We refer to this algorithm as "Searching Frequent Patterns" These two algorithms are detailed further in the following.

3.1 Unique Pattern Indexing

This algorithm takes a text file containing the DNA sequence of the target species (referred to as 'Database' from hereon) and encodes it with numerical values for further processing. The encoding process is dynamic. The idea behind the encoding is that since the unique sequences within a Database will be nothing more than various permutations and combinations of A, G, C, T (in case of DNA) of length four, we will only need to assign a numerical value to each unique sequence and later replace that sequence whenever encountered within the complete DNA sequence. Table 1 represents a sample of such a database:

Table V: Unique Pattern Index

ID	Sequence
1	AGCT
2	ACGT
3	AACT
4	TGAA
5	TAGC
6	CATA
⋮	⋮
255	CCCT
256	AAAA

Since at most 256 (4^4) combinations of A, C, G, T (of length 4) is possible in case of DNA, the ID value will only go up to 256 at worst case scenario. Once the ID value is generated, we will use these values to replace the corresponding sequence (e.g. TGAA will be replaced with 4) and thus save the final output as a text file.

As we divide the whole DNA sequence in sub-sequences of length four, around the end of the sequence there may be a stray sequence of length one, two, or three. For example, for the sequence AAATAGCTTATAGC, the program will extract and id the sub-sequences AAAT (1), AGCT (2), TATA (3). Since the last sequence encountered is GC which is of length two, the program will simply put it into the file as GC and id it to 4.

In summary, *Unique Pattern Indexing* Algorithm indexes each unique pattern and puts it into a HashMap. The workflow is described below:

Algorithm: Unique Pattern Indexing (Step 1)

Input: A text file containing the DNA sequence (D) of the target species.

Output: A dynamically populated text file containing every (length <= 4) unique DNA sequence encountered in the target sequence.

Pseudocode:

```
//read from the database file
F = read database
//put the sequences within database into a HashMap h
h.key = sequence
h.value = id
//read the subsequence of length 4 from the target sequence
length = target_sequence.length
beginning = 0
i = 0
While segment_size < length
        segment_size = i * 4
        end = segment_size
        if segment_size < length
                temp = target_sequence.substring.beginning to target_sequence.substring.end
                i++
                beginning = end
                if h contains key = temp //if there's a match
                      put corresponding h.value into output
                else //if no match is found, put the new sequence into hashmap and write to database
                        h.set_key_value_pair = temp, h.size+1 //h.size gives the total length of the hashmap
```

```
            length++
        endif
    endif
endwhile
```

Algorithm 1: Unique Pattern Indexing

3.2 Searching Frequent Pattern:

Upon processing the given input DNA sequence by replacing each unique sequence (primarily of length four) with its corresponding index, the file is then checked for frequent patterns of length 4, 8, 12, and 16 via *Searching Frequent Pattern* Algorithm.

Algorithm: Searching Frequent Pattern (Step 2)

Input: The output text file from Step 1.

Output: Four different text files, each containing the repeating sequences of length 4, 8, 12, and 16 respectively, along with their repeat counted, and sorted in an ascending manner**.**

Pseudocode:

```
//read from the processed_sequence file
F = readfile processed_sequence
//put the sequences within F into a StringBuilder sb
sb.append = f.readline
//put each id within sb (separated by a space) into a String Array
HashMap map
String splitted []
for i = 0; i < sb.length; i++
    splitted [i] = sb.unique_id .split_by_spaces //the ids are separated by spaces in the original
file
endfor
//process segment of size 4 and check for repetition
for I = 0; i < splitted.length; i++
    temp = splitted[i]
    if map.contains temp
        flag = map.value_for_temp
        flag++
        map.put_value = flag
    else
        map.put_key = temp
        map.put_value = 1
    endif
endfor
//sort map
map.sort_by_ascending_order = true
//write to an appropriately titled text file
File output = repetition_count_4
```

```
output.write map
//clear map to save memory
map.clear
//similarly process segments of size 8, 12 and 16 while adjusting the loop termination condition accordingly
```

Algorithm 2: Searching Frequent Pattern

3.3 Displacement Based Weighting Algorithm:

This algorithm will provide the weight on the gene which will be used for generating the graph. Creating the network graph and analyzing the weights we will be able to rank the gene. We will first find the most frequent pattern (of length 4) of the candidate gene and will find its relative frequency then we will calculate the displacement by comparing this frequency with the frequencies from known database. This will be for one pattern, we will have to find this for all the genes in the database, from this cumulative result we will subtract 1 and will get weight for that gene.

Calculate Relative Frequency:

I. Find out the frequent patterns of length 4 in ascending/descending order (ordering method is not important since we'll take the most frequent pattern first and then the second most one and so on).
II. Each sequence will have a unique ID (as the sequences themselves are unique as well). We'll need this ID value in the later parts of the procedure.

Then we calculate the relative frequency using the following equation:

$$\textbf{Relative Frequency} = \frac{length\ of\ the\ sequence * repeat\ count}{total\ length\ of\ the\ gene\ sample}$$

Here, length of the sequence is 4. We won't be handling sequences of length 8/12/16 for now since that will need manipulating thousands of data.

III. Then we will construct the sorted table for each gene sample. In our case, we have 20 gene samples for 2 diseases, 10 samples per disease. So there will be 20 different tables.
IV. Now, we will take the target gene sample (the sample that needs to be ranked) and construct the Relative Frequency table in the similar manner.

Calculate Displacement:

In this step, we need to compare the ranks of each ID sequence in the following manner:

There are two diseases in the database for now: Alzheimer's and Prostate Cancer. Each disease has 10 samples (collected from NCBI). Accordingly, there are 10 tables sorted by the Relative Frequencies of the unique sequences (of length 4).

I. We take the most frequent sequence of the test gene sample. Let the ID of the sequence be 55. One thing to note here is: the ID values are consistent across samples. That is, if ID 55 represents the sequence ***AATA***, then it does the same across all the samples in the current database as well as samples that might be added in the future.

II. Then we take one sample from the database for a particular disease. Say, we want to find out how the sample gene ranks in terms of inducing Prostate Cancer. So, we take one sample gene (that induces Prostate Cancer) from the database.

III. Now, we calculate the *Displacement* value Δd given that the selected sequence of length 4 (in this case, the sequence with ID 55) exists in the sample taken from the database using the following equation:

$$\Delta d_i = |r_t - R_i| \cdots \cdots \cdots \text{(i)}$$

Where,
Δd_i = Displacement for sequence i (i = ID value for the particular sequence),
r_t = Rank of the sequence in the test sample,
R_i = Rank of the sequence in the ith database sample.

IV. In case the sequence in the test sample that needs to be weighted doesn't exist in the database sample, then we simply assign: $R_i = 0$.

Assign Weights:

Weight of sequence with ID i,

$$w_i = 1 - \frac{\Delta d_i}{\text{n}} \cdots \cdots \cdots (ii)$$

If i exists in the sample from the database.

Otherwise (in case i doesn't exist),

$$w_i = -\frac{\Delta d_i}{\text{n}} \cdots \cdots \cdots (iii)$$

Where $\Delta d_i = r_t$ (since $R_i = 0$) and n = total number of unique sequences of length 4.

Then we add up the weights of all *n* unique sequences and calculate the cumulative weight for the target sample ***with respect to the first sample in the database.***

$$\overline{w_1} = \frac{\sum_{i=1}^{n} w_i}{n} \dots \dots \dots (iv)$$

Similarly, we find $\overline{w_k}$ for all *k* samples in the database. Since we have 10 sequences for Prostate Cancer, we will have weights from $\overline{w_1}$ up to $\overline{w_{10}}$.

Now, we find the cumulative weight for the test sample with respect to all the sequences present in the database:

$$\overline{W_d} = \frac{\sum_{k=1}^{m} w_k}{m} \dots \dots \dots (v)$$

Where,

$$\overline{W_d} = cumulative\ weight\ for\ the\ particular\ disease\ \boldsymbol{d}$$

$$m\ =\ number\ of\ gene\ samples\ for\ disease\ \boldsymbol{d}$$

Example:

Let us imagine that the gene we need to rank is named XYZ.

Say, sequence with ID 99 is the most frequent one and repeats 59 times and the length of the sample is 16767. So, the Relative Frequency, R.F. = (59*4)/16767 = 0.014075. In this way, we build the table with every sequence involved.

Now, we want to rank XYZ in terms of Prostate Cancer. So, we take the first gene sample from Prostate Cancer, which is ABCB7.

In case of ABCB7, let's say that seq. ID 99 ranks 3rd. So, the displacement will be:

$$\Delta di\ =\ |1-3|\ =\ 2.$$

In this way, we find the consecutive displacements for all the sequences involved and then calculate the weight.

3.4 Graph Generating and Ranking Algorithm:

This algorithm will take the weighted genes as input and will create a directed graph where gene nodes will point towards the disease node. In a single network there will be one disease node and many gene nodes. From the weight of each node assigned on the edge we rank the genes. We use JUNG2 [19] for graph generation.

Step 4: Generating Weighted Graph Network

I. We take the corresponding nodes and their edge weight (calculated via the algorithm described in section 4.3 and put it in a three-dimensional array in the following order: (gene_node, disease_node, weight).
II. We then generate the graph so that the gene nodes are connected to the disease nodes via a directed edge.
III. The direction will be such that for the disease nodes, out-degree = 0 and for the gene nodes, in-degree = 0.
IV. In this manner, we keep repeating the process for each new sample in the database.

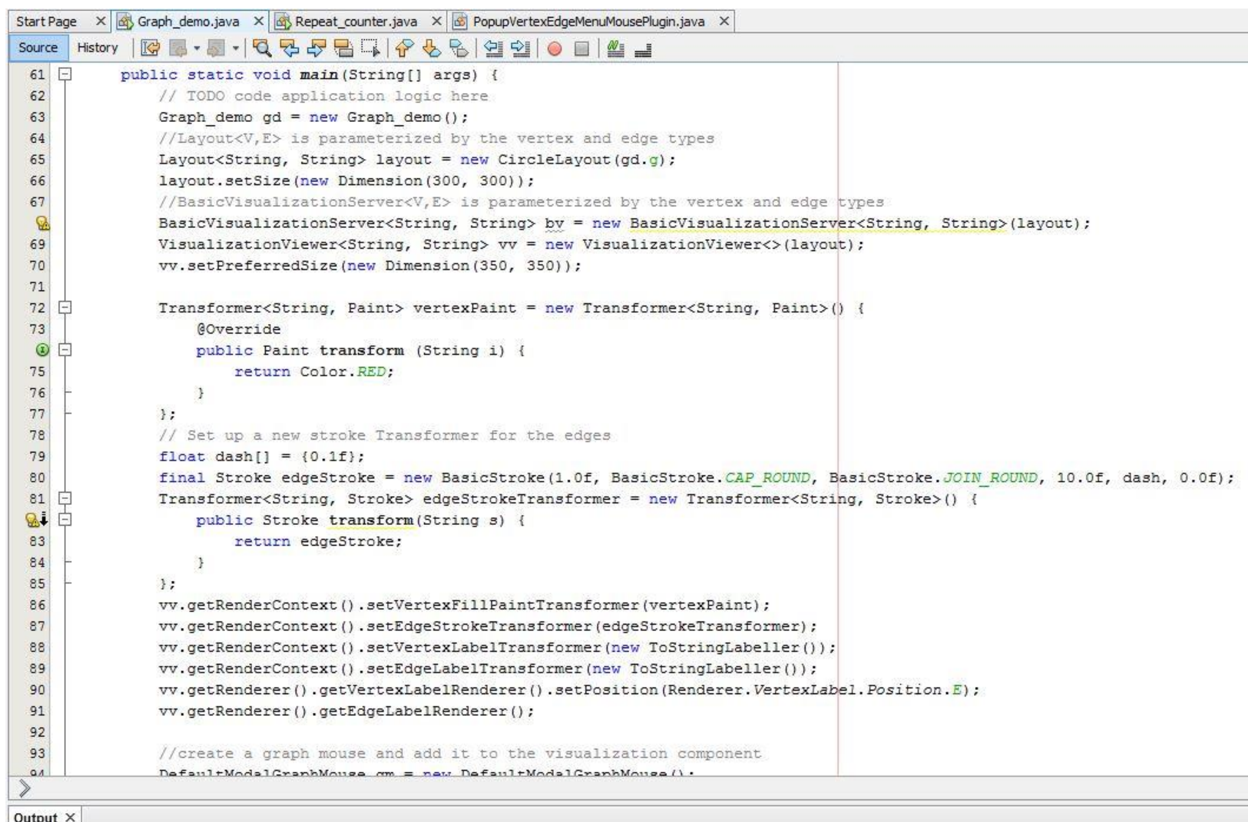

Figure 5: *Graph Generation Code.*

Chapter 4: Materials and Methods

4.1 Materials:

In order to build the initial database, we need some sample DNA sequences. The sample dataset are obtained from NCBI database and are of different lengths. This gives us a realistic simulation given the dataset are all obtained from different species of bacteria and viruses. Table II shows our sample dataset, along with their NCBI reference.

Table VI: Sample Dataset

Experimental Dataset	NCBI Reference Sequence	Data
***Acaryochloris marina* MBIC11017 chromosome, complete genome**	NC_009925.1	AATAAATA…ACCAC
***Acidithiobacillus caldus*, SM-1 chromosome, complete genome**	NC_015850.1	ATGAGTAG…TCATC
***Achromobacter xylosoxidans* A8 chromosome, complete genome**	NC_014640.1	ATGAAAGA…GCGAC
***Acetobacter pasteurianus* 386B, complete genome**	NC_021991.1	AATGGGTA…GCTAG
***Acetobacter pasteurianus* IFO 3283-01, complete genome**	NC_013209.1	ACTGCAGG…TAGAA

Each of these sample files are of varying sizes and of different species, thus making the simulation environment closer to a real-life test case scenario.

As for the disease gene prioritization, we have used the following 6 genes for primary analysis. The final dataset will have 20 genes initially, and more will be added with the progression of time.

Table VII: Disease Gene Database

Gene	Name	Induced Disease(s)
ABCB7	ATP-binding cassette, sub-family B (MDR/TAP), member 7	myocardial infarction, pearson syndrome, x-linked sideroblastic anemia with ataxia, x-linked sideroblastic anemia and 12 others.
ALM	alpha-2-macroglobulin	alzheimers disease, argyrophilic grain disease
ADAM10	ADAM metallopeptidase domain 10	alzheimers disease, degos disease
ACOX3	acyl-CoA oxidase 3, pristanoyl	Mutism
ABCA1	ATP-binding cassette, sub-family A (ABC1), member 1	coronary artery disease, scott syndrome, syringomyelia, Alzheimer's, xfamilial hypercholesterolemia, chediak-higashi syndrome
ACTR3BP6	ACTR3B pseudogene 6	HIV-1

4.2 Methods:

To continue with the simulation, at first we applied the Unique Pattern Indexing algorithm on the sample data. To do that, we first load the (unique) DNA sequences within the database into a HashMap and store them as a key. Then we number them sequentially and put the indices into the value field of the HashMap. This was for frequent Pattern finding then starts the process of gene prioritizing. We first find the relative frequencies of candidate gene and known genes and then comparing there displacement we find the weight and after that we generate the graph network. Then we do the ranking of the genes from their assigned weight. The whole process can be summarized as below:

- At first load the input database file.
- Create a HashMap and initialize it as empty.
- Check for each substring of length from the beginning of the sequence, and check if the substring exists within the Map as a 'key'. If it does, ignore it and move on to the next substring. Otherwise assign an 'incremented' value.
- Load this indexing file as input to the *Displacement Based Weighting Algorithm* this will give repeat count and relative frequency.

- We take two HashMap to compare the candidate gene and the known gene this comparison will help to give weight and produce the graph network and it will help to maintain the complexity to O(1).

At first we extract subsequences of length 4 from the beginning of the target sequence and see if it exists as a key within the HashMap. If it exists, we just retrieve the value for that key and put it into the output file. Otherwise we first put the subsequence into the HashMap, assign it a new value to it, and then put that value into the output. To exemplify, we consider the input string AAGTACTTTATAACTTTATA. Now, the algorithm will automatically assign the index value 1 to AAGT, 2 to ACTT and 3 to TATA etc. Since ACTT already got a value assigned, it will just be replaced with 2. So, the output will be 1 2 3 2 3. In this manner, we iterate through the sequence up until the very end. If we encounter a sequence of length 3 or less and it does not exist in the HashMap, we treat it as a new unique sequence and put it into the DNA database.

At this point we employed the *Frequent Pattern Searching* algorithm. At first, we load the output file from *Unique Pattern Indexing* algorithm and treat the whole sequence as a single, continuous string.

In this manner we split the string into segments of length 4(we are not considering length 8/12/16). We process the segments of length 4 at first to check repetition. The first value is put into a HashMap as key and the corresponding value field is set to zero (0). Whenever a repetition is encountered, the value field is incremented by 1.

Upon completion of the whole sequence for repeating patterns of length 4, we sort the HashMap by values and write the output to an appropriate output file. After going through all the values, we clear the HashMap.

Now we got a table with the values of key(id) and their unique pattern(total 256) . Now for the *Displacement Based Weighting Algorithm* we will have to give to input sequence this will be two genes. One is the candidate gene and another will be a gene from the database whose information is known to us. The algorithm will provide two outputs on the two input sequence for each of the sequence the number of repeated pattern will be shown and also the relative frequency (4 * repeat count/total length). Then we will calculate the displacement, we will take the modulus value.

The comparison will be done by putting the information of the two genes in two HashMap table. For first table unique id will be Rank list (most frequent pattern will be ranked 1) and in the second column in will the id which is unique for all. In second HashMap table we will also have same columns here key will be that unique id and second column will hold the ranking. In two

tables taking the common values we can compare the displacement. This was for one candidate gene and one gene from database we have to do this for the candidate gene and all the genes of the database.

Finally we will generate a Graph Network with the disease nodes and corresponding gene nodes, with the gene nodes connected to the disease ones via an outbound directed edge.

```
1  GCACACAGAGCAGCATAAAGCCCAGTTGCTTTGGGAAGTGTTTGGGACCAGATGGATTGTAGGGAGTAGG
2  GTACAATACAGTCTGTTCTCCTCCAGCTCCTTCTTTCTGCAACATGGGGAAGAACAAACTCCTTCATCCA
3  AGTCTGGTTCTTCTCCTCTTGGTCCTCCTGCCCACAGACGCCTCAGTCTCTGGAAAACCGTGAGTTCCAC
4  ACAGAGAGCGTGAAGCATGAACCTAGAGTCCTTCATTTATTGCAGATTTTTCTTTATATCATTCCTTTTT
5  CTTTCCTATGATACTGTCATCTTCTTATCTCTAAGATTCCTTCCAGATTTTACAAATCTAGTTTACTCAT
6  TACTTGCTTACTTTTAATCATTCTTCCCCAACTCTCTGAAGCTCTAATATGCAAAGCCTTCCTAAGGGGT
7  GTCAGAAATTTTTAGCTTTTTAAAAGAATAAATTTTAGATATTCACATTCATATTGATCTACTTGAGACC
8  ATGCTATTTATCTTTTCTTATTTCCTCTTTCTCAAGGGTCCATTTTCTATTTTATAAAAATAAAGACAAT
9  TCTCTCCCACAACCAAACATGGAACAATGCCCTGGAGTATAAAAATCTATAGAGTGCCAAATAAAGGAAC
10 AATTTGAAATACTGGTGTTGATATTGAAAAAGCAAGGGACTCTAATGTCAGAAGAGAAATCCTTTTGCAG
11 ATGAGGTGGTGATGAATTCTTTGTTTCAACACAACTGAAGGAGGAACTGAAGGAAATACCAGCTGATGAG
12 TGATGAGAAGGGATTCTTGATAATAGAGTACTAGGTGATTTTTGGCATGTAATGCAGAAGTTGCAAGAAG
13 TGGTAACAATGATGCAATTGTTTTACCTGCCATTTATTTACTTTTATGTGAGCCATTCTTCTTAGCACTT
14 ATAGCTACACAAAACAAAAATAGTAACAGAATTAATGTTGTTTAATTCTTGCAATCCATGGATGCATAAA
15 TTCACTGGGGGAAAAAACAGCTCATCATTCTCATTAAAGATGTGCTTCAAAAGTATTTTAATTTTATATC
16 TAATATGTATGAATCATACTTTGTATTTATTTTGTTTTGATCAGTTATATACAAGTATTTTTGAACATAG
17 CTCAGTCAGAAGGAAATGTTTAATATTTATAAATTTATGGTTACATTCTATTTAAAAGAGGAGTTAAAGT
18 TAAATTTACCTACCCACATATGTTACATATATATGTATTTATGTATATGTATTCATATATGTATATATGT
19 GAACATAAGTATACATACGTATATGTATAGATGCTTGACAATAAAGAAGTAAGAATAATTCACAACATTT
20 TTTGAAATATAAAAATTTAGGATAAATTTCTGTATGGTAATTGGCATGGAAATTCAAATTCAAAAAGGAA
21 AAAAGAAGAGAAAGATATTAAATATCAGACCATTAAAAGAATTTTTTAATGTACTTTTAAATAGTGATAG
22 TAGGTATCTTATACTACAGTGTTTATTATTCATGAGAAAATTGTAAAAGTAATCTAAGTATTAATTTAAA
23 ATATCATCAAAAATAATATCTTTTGCTATTACTTAAAATCATGATAAAAATATGTTTACTTGAAAATATG
24 TAAGGAGTGCACAGAGTCCAAAAATTATTTTAGGAGTTCTGTGAGCAAAAATGTATAAAAACTACAGGGT
```

Figure 6: *Sample Dataset.*

Chapter 5: Performance Evaluation

We applied the algorithms over the experimental dataset described at Table II to generate the frequent pattern of length 4. We have taken the results of the first five species from Table II in order to keep the resultset more manageable. We have displayed only the last three values (i.e. the three most frequent values) from the ascending ordered sorted list of frequent patterns. However, the other values can be easily retrieved from the output file if such need ever arises. The results are illustrated in Table III.

5.1 Experimental Results (Mining Frequent Pattern):

Our goal is to find the number of repeating sequences of nucleotides of length 4, 8, 12 and 16. The algorithm automatically sorts the results in an ascending manner, so the three most repeating sequences were easy to find.

For example for the whole sequence of reference "NC_009925.1" we got "AAAA" has been repeated 11829 times "TTTT" has been repeated 11588 times "CAAA" has been repeated 11539 times. We got this result for string length of 4. For length 8, 12 and 16 and for rest of the datasets we got the result in similar fashion. For example, string "GCGATCGC" (length = 8) repeated 488 times, TGCGATCGCAAC (length = 12) repeated 52 times and finally the pattern GATCGGCTGAAGTCAG (length = 16) repeated 10 times.

Searching this frequency of patterns helps us in many ways, if we did not find the repeating sequence, it would be hard to find the proper cause that has altered the million character long gene sequence. If we know that which patterns are most frequent in a sequence, then without dealing the whole sequence we can emphasize on those subsequence only. For disease diagnosis it is often the case that we do not have to search the whole sequence. Instead searching for the frequent patterns are often enough to find valuable related data.

Table VIII: Simulation Results Showing Frequent Patterns of Length 4, 8, 12 and 16

Length	Reference	Sequence	Frequency
4	NC_009925.1	AAAA	11829
		TTTT	11588
		CAAA	11539
	NC_015850.1	GGCG	8158
		CGCC	8144
		GCGC	8138
	NC_014640.1	GCGC	38449
		CGGC	31313
		GCCG	31117
	NC_021991.1	TGGC	6290
		GCCA	6223
		CAGC	6099
	NC_013209.1	TGGC	6526
		GCCA	6448
		TGCC	6338
8	NC_009925.1	GCGATCGC	488
		CGAT CGCA	447
		TGCGATCG	418
	NC_015850.1	CCGCCGCC	160
		CGGCGGCG	152
		GGCGGCGG	143
	NC_014640.1	GCGCGGCG	870
		GCGCGCGC	869
		CGCCGCCG	807
	NC_021991.1	TTTCTGGC	106
		GCCT GCGC	102
		CATCCAGC	98
	NC_013209.1	TTTCTGGC	113
		TCTGGCAG	109
12	NC_009925.1	TGCGATCGCAAC	52
		GTTGCGATCGCA	44
		TGCGATCGCATC	44
	NC_015850.1	CCCT GGGCGCGG	10
		CCAGTTCCGCCT	10
		TCGAGAGCCAGA	9
	NC_014640.1	GGCGCTGGCCGC	34
		GAAAGCGCCGCC	32
	NC_021991.1	TCTGGCTCTGGC	16
		GCTCTGGCTCTG	15
		TGGCTCTGGCTC	15
	NC_013209.1	CTGGCTCTGGCT	12
		GGCTCTGGCTCT	12

		CTCTGGCTCTGG	11
16	NC_009925.1	GATCGGCTGAAGTCAG	10
		CTTGGAGTGCATCATC	9
		AATGCTTGGAGTGCAT	9
	NC_015850.1	ACCAGAACCTCAAGAC	9
		GGGCGCGGGAAGGCTC	9
	NC_014640.1	CTCCGGACCGCAATCG	7
		TGCTGCTGGACGAACC	6
		CCAACGGCGCGGGCAA	6
	NC_021991.1s	GCTCTGGCTCTGGCTC	15
		TGGCTCTGGCTCTGGC	15
		TCTGGCTCTGGCTCTG	14
	NC_013209.1	GGCTCTGGCTCTGGCT	12
		CTGGCTCTGGCTCTGG	11
		CTCTGGCTCTGGCTCT	11

5.2 Graph Generation and Weight Assignment:

In order to generate the graph, at first we need to find out the weights associated with each gene sample so that the candidate gene(s) can be prioritized alongside them.

By using the method described in **Algorithm 3.3** we find out the relative frequency of the unique IDs in a particular gene sample. For example, the following table shows the relative frequency for ACTR3BP6 (hiv_1), a gene that is responsible for HIV-AIDS type-1.

Table IX: Relative Frequency for ACTR3BP6 (hiv_1)

Sequence ID	Relative Frequency
99	0.01952277
49	0.01301518
55	0.01301518
59	0.01301518
131	0.01084598
88	0.00867678
159	0.00650759
. .	. .
68	0.00433839

Now, we need to calculate the weights of various target sequence by comparing them with the sequences in the known database and assigning weight as per the method explained in **Algorithm 3.4.**

A few of the weights generated after applying **Algorithm 3.4** is given below in order to gain a perspective:

Table X: Weights of the tested Disease Genes

Disease Gene	Disease Induced	Weight
ABCB7	Alzheimer’s Disease	0.8957366
A2M	Alzheimer's Disease	0.8637768
ADAM10	Alzheimer's Disease	0.9152687
ACOX3	Mutism	0.8969674
ABCA1	Alzheimer's Disease	0.8765102

One thing to note here is: we compared the genes of Alzheimer’s with those of Alzheimer’s in Table V, not with the known genes of other diseases. In case that we compare the gene of one disease with another one which is not a related disease, the weight value decidedly goes down a lot, as cam be seem from Table VI:

Table XI: Weights of Genes when compared with an unrelated disease

Disease Gene	Disease Induced	Compared With	Weight
ACTR3BP6	HIV/AIDS	Alzheimer’s Disease	0.43917594
ADAM10	Alzheimer’s Disease	HIV/AIDS	0.41317346

In this way, we can see that the unrelated diseases need not to be compared since the weight will be below an acceptable level anyway.

Finally, with all the weights assigned to the candidate genes and prioritizing them according to the weight generated, we draw a Graph to visualize the system so that the linkages can be referred to at a glance. We create disease nodes, gene nodes and create an outbound link from a gene node to corresponding disease node.

We also put the calculated weight along the edge of a linkage to show the connection between the disease and the candidate gene. In this way we can build a bigger and more accurate network as we put in more sample and calculate more weights to normalize the final value.

As can be seen from the graph network below, the nodes are of two types: disease and gene nodes and they can be told apart quite easily since only the disease nodes have incoming edges while the gene nodes got only outgoing edges.

The weight of each gene is shown alongside the edge. This graph is “pannable” and “zoomable” in the actual application, so the values can be magnified and analyzed separately if required.

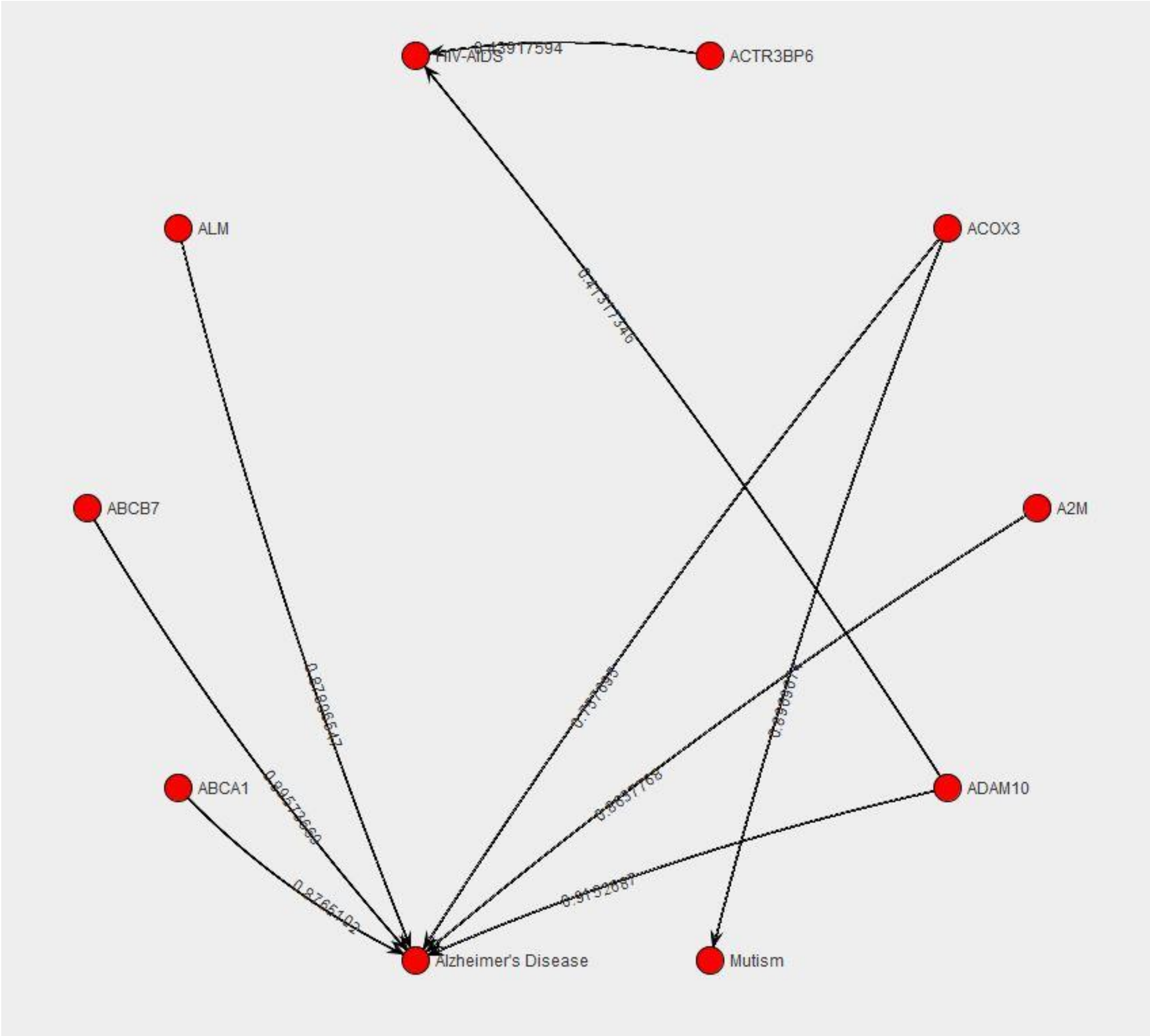

Figure 7: *Output Graph Networks*

5.3 Simulation Environment:

The simulation was done on a desktop running Windows 7 SP1 64 bit. The hardware includes a core i5-3570 processor running at 3.8GHz (overclocked) and 8GB of system RAM running at 2133MHz. The system also has a discrete GPU (Nvidia GTX 760 with 2GB of GDDR5 VRAM).

The programs implementing the algorithms were written in Java and the IDE was Netbeans 7.4 running Java version 7 update 51. We also used JUNG (Java Universal Network and Graph Utility) version 2.0 for Graph Network generation.

The source codes for the algorithm can be downloaded for free from:

https://copy.com/rr7uZC1sQR0m

https://copy.com/i4IXP4jJC4ql

Chapter 6: Discussion & Conclusion

6.1 Discussion:

While working with the gene database due to restrictions we could not access the whole data base. Around 10000 samples were there among them 5000 were accessible. According to the Dempster-Shafer theory or belief that the normalization of the total sample space can be done though we don't have full information of the system. According to this belief the system will work similarly for both 5000 data and 10000 data.

Saving memory and reducing time complexity is our prime concern in this proposed method. In normal case if we want to represent a nucleotide of length four we need 4 characters that occupy 8 bytes (4×2 byte) in the memory. If we want to deal with a pattern 16 characters long then we need 256 (16×16 bits) bits in the memory. In our algorithm we are representing a nucleotide with a numerical value, this can range from one digit to maximum three digit. So for representing nucleotides of length 16, in the best case scenario we need only 4 bytes of memory and for the worst case scenario if we represent all nucleotides using three digit integer then we need maximum 12 bytes of memory that is 96 (12×8) bits of memory so minimum memory efficiency is 37.5%.

Since our algorithm does a lot of comparison within a large dataset, the computational complexity is of a big concern. However, by using Mapping Techniques, we have reduced the time complexity for comparison to $O(1)$. As per the concept of big-oh notation, this means that the comparison takes a constant amount of time regardless of how large the dataset might be. This makes it perfect for comparing large amounts of genetic data without any additional computational overhead.

6.2 Conclusion:

Using our proposed method will help to reduce the amount of accessing data for finding particular pattern as well as it will be memory efficient. Memory efficiency and fast access of data are not only one of the prime targets in the field of bioinformatics but also in other sectors. Sectors where huge amount of data need to be analyzed and where repetitions of patterns are important, our method will prove helpful there [8].

For instance evaluating patterns of telephone use, if a company wants to check the number of calls in a particular time for a particular user then it can use the pattern matching algorithm. If we need this for thousands of users then our algorithm will prove useful.

Again in business identifying fraud insurance claims can be done efficiently or human resources (HR) departments want to identify the characteristics of their most successful employees. Information obtained – such as universities attended by highly successful employees – can help HR focus recruiting efforts accordingly. Also in the field of medical diagnosis where sometimes it is needed to find out the sample or required pattern in less time consuming manner our algorithm can be of great use in this types of scenario. Sales forecasting database marketing, balancing stock, call roaming percentage are also some of the sectors where our method can be proved helpful.

References

[1] Shuang Bai, Si-Xue Bai, “The Maximal Frequent Pattern Mining of DNA Sequence”, GrC, pp 23-26, 2009.

[2] A Fast Contiguous Sequential Pattern Mining Technique in DNA Data Sequences Using Position Information Syeda Farzana Zerin, Byeong-Soo Jeong Department of Computer Engineering, Kyung Hee University, 1 Seocheon-dong, Giheung-gu, Yongin-si, Gyeonggi-do, 446-701, Korea Date of Web Publication 12-Dec-2011

[3] Mining Maximal Adjacent Frequent Patterns from DNA Sequences using Location Information Moin Mahmud Tanvee, Shaikh Jeeshan Kabeer, Tareque Mohmud Chowdhury, Asif Ahmed Sarja, Md. Tayeb Hasan Shuvo, Department of CSE.

[4] T.H Kang, J.S Yoo and H, Y Kim, “Mining frequent contiguous sequence patterns in biological sequences”, in proceeding of the 7th IEEE International Conference on Bioinformatics and Bioengineering, pp 723-8, 2007.

[5] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules.” In Proc. 1994 Int. Conf. Very Large Databases (VLDB?94), pages 487–499, Santiago, Chile, Sept. 1994.

[6] R. Srikant, and R. Agrwal, "Mining sequential patterns: generalizations and performance improvements", in Proceedings of 5 th International Conference on Extending Database Technology (EDBT'96), Avignon, France, pp. 3-17, Mar. 1996.

[7] J. Pan, P. Wang, W. Wang, B. Shi, and G. Yang, "Efficient algotithms for mining maximal frequent concatenate sequences in biological datasets", in Proceedings of the Fifth International Conference on Computer and Information Technology(CIT), pp. 98-104, 2005.

[8] Notable uses of Data mining- Wikipedia.

[9] Bioinformatics and the Internet-Andreas D. Baxevanis.

[10] Charles E. Leiserson, Amortized Algorithms, Table Doubling, Potential Method Lecture 13, course MIT 6.046J/18.410J Introduction to Algorithms – Fall 2005.

[11] Shuguang Wang , Milos Hauskrecht Gene Prioritization Using a Probabilistic Knowledge Model.

[12] E. A. Adie, R. R. Adams, K. L. Evans, D. J. Porte-ous, and B. S. Pickard. Suspects: enabling fast and effective prioritization of positional candidates. BMC Bioinformatics, 22(6):773–774, 2006.

[13] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. The Journal of ACM, 46(5):604–632, 1999

[14] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report, 1998.

[15] Page L, Brin S, Motwani R, Winograd T (1999) The PageRank Citation Ranking: Bringing Order to the Web. Technical Report.

[16] Kleinberg JM (1999) Authoritative sources in a hyperlinked environment. Journal of the ACM 46: 604–632. doi: 10.1145/324133.324140

[17] An Algorithm for Network-Based Gene Prioritization That Encodes Knowledge Both in Nodes and in Links. Chad Kimmel mail, Shyam Visweswaran Published: November 19, 2013 DOI: 10.1371/journal.pone.0079564

[18] "Facts about Google and Competition". Archived from the original on 4 November 2011. Retrieved 12 July 2014.

[19] Fisher D., Smyth P., Boey Y. B. and White S.; "Analysis and Visualization of Network Data using JUNG"; Journal of Statistical Software, pp 136-170, Issue II.

[20] Dempster-Shafer Theory for Intrusion Detection- Thomas M Chen Varadharajan.