Department of Electrical Engineering

# FINAL YEAR PROJECT REPORT

**BENGEG2-INFE-2019/20-YNS-05**

**Develop an efficient path finding algorithm for very large graphs**

Student Name: Hui Ka Ho
Student ID: 54841734
Supervisor: Dr. SUN, Yanni
Assessor: Prof. CHEN, Guanrong

Bachelor of Engineering (Honours) in
Computer and Data Engineering

# Student Final Year Project Declaration

I have read the student handbook and I understand the meaning of academic dishonesty, in particular plagiarism and collusion. I declare that the work submitted for the final year project does not involve academic dishonesty. I give permission for my final year project work to be electronically scanned and if found to involve academic dishonesty, I am aware of the consequences as stated in the Student Handbook.

Project Title: Develop an efficient path finding algorithm for very large graphs

Student Name: Hui Ka Ho

Student ID: 54841734

Signature: 

Date: 10th November, 2020

**Turnitin Originality Report**

# Abstract

Graphs are a data structure made of edges and nodes and both of them can be labelled by sequence, so that it is commonly used to represent a set of sequences, especially for some complex sequence like genome which makes up with a sequence of different proteins. Different sets of genome sequences may have some similar parts, by making use of the properties of the graph, a large graph can represent many combinations of genome sequences and each path in the graph spells a concatenated sequence, which reduces the redundancy part between each sequence. Besides, having an efficient way to accomplish align sequencing reads is very important for analysing genome assembly.

In the project. I compared different data structures and algorithms related to graphs and familiar with different kinds of graph structures such as directed acyclic graphs and cyclic graphs. After that, I implemented a data structure to have a sequence analysis on a large graph such as retrieving the sequence from the graph by using a path. I have also developed two algorithms that can accomplish the align sequencing reads which including is sequence-to-graph matching and semi-global sequence-to-graph alignment by considering the time complexity. For the sequence-to-Graph Matching, the algorithm that I developed provide a time complexity O(|M||E|) to solve the problem, and for the sequence-to-Graph Alignment, the second algorithm I developed provide a time complexity O((|V|+|M|)|E|) to retrieve a path. (where |M| is the length of the searching sequence, |E| and |V| are the number of edges and nodes in the graph respectively.

# Acknowledgements

For the final year project, I would like to express great appreciation to all those people who me the assistance for me to complete project.

Firstly, I would like to express my gratitude to Dr. SUN, Yanni, my final year project supervisor, who provide me patient guidance and encouragement, and good direction for the project and the research work. With the help of my supervisor, I understand the aim of my project quickly.

Secondly, I want to extend my thanks to my project accessor, Prof. CHEN, Guanrong, who gave me a lot of useful suggestion on the project and motivate me in the research.

Thirdly, I want to give a special thanks to the technician staff in the laboratory of Electrical Engineering Department, with their help, I have a powerful computer to run my program.

Last but not least, I am very grateful to get the help of City University of Hong Kong, which provide me a lot of resource for researching. Such as the Run Run Shaw Library website provide me a lot of research paper related to my project topic.

# Contents

# List of Figures

# List of Tables

# Introduction

Graph is a kind of non-primitive data structure (also known as the user-defined data structure) which are not pre-define in the programming language, the data that can hold by the non-primitive data structure are decided by the developer. Non-primitive data structures are always more complicated and derived from primitive data structures. Unlikely to other non-primitive data structure, graphs are non-linear, which mean graphs can express not only the data that arranged sequentially, but also some data that are irregular and having multiple connections.

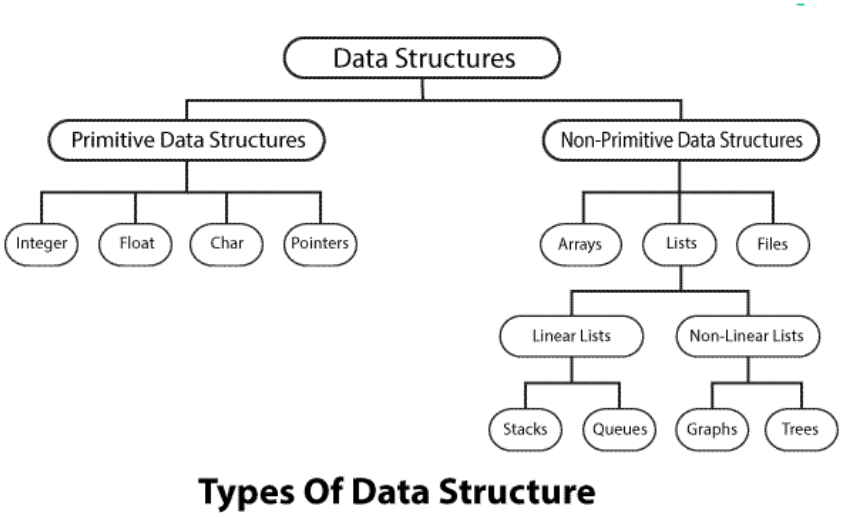



Figure 1 Types of Data Structure

The complicated structure of graph does not allow us to traverse all the elements in a single run. In order to have the traversal regularly and make sure all routes in the graph have been taken, a searching algorithm is needed.

Besides, the special structure of graphs allows us to label data on either on nodes or edges, as a result, a graph is able to represent a set of sequence, especially for some complex sequence like genome. Compare a genome sequence to a graph (also known as graph alignment) is a frequency work for genetic engineer to compare different kinds of DNA and living beings, if there is a efficient algorithm for them to search, the effectiveness of biological research will increase a lot, it is also the objective of the project.

There are many searching algorithm specific design for graph, such as Breadth-first Search (BFS) and Depth-first Search, which having a time complexity of O(|V|+|E|) (where |V| stands for number of nodes in the graph and |E| stands for the number of edges in the graph) for scanning all possible paths in a graph. However, BFS and DFS only report all the possible path but do not handle the comparation between the paths with the searching sequence. If we combine the compare process with the searching algorithm, the algorithms will has a unacceptable time complexity.

```
Define: Graph G, Queue Q, List Inputs, List OutFlows, Parameter
MinFraction

Initialize: G for system with nodes containing tuples defined
(child = node, weight = fraction of emergy flowing to child)
Require: child weights sum to 1.0

Initialize: Inputs for system with tuples (node, flow)

Initialize: MinFraction as minimum fraction of each input to pass
to child process nodes

Initialize: Q as empty FIFO queue, Outflows as empty List

For input (node, flow) in Inputs:
  Define: Set P
  Add: node to P
  Define: MinFlow = MinFraction * flow
  Define: Task T = (node, flow, P, MinFlow)
  Push: T to Q

While Q has elements:
  Pop: Task U = (node, flow, path, minflow)
  If node has no children:
    Define: Output O = (node, flow)
    Add: O to OutFlows

  For child in node (child, weight):
    Define: childflow = weight * flow
    If childflow >= minflow AND child NOT IN path:
      Add: child to path
      Add: Task (child, childflow, path, minflow) to Q

Collect: Sum outputs (node, flow) in Outflows by node
```

Figure 2 Pseudo Code of BFS algorithm

In the project, I am developed two algorithms for compare sequences and graph, one is for searching a path that exactly matches with the searching sequence with the time complexity of **O(|V|*|E|)** where |E| stands for the length of the sequence. Another one is for searching a path that has a minimum edit distance with the searching sequence with the time complexity of **O((|V| + |M|)*|E|)**.

# Background

## Graph

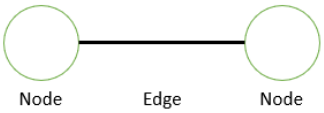



Figure 3 Graph with Two Nodes and One Edge

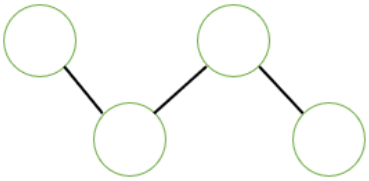


Figure 4 Undirect Graph

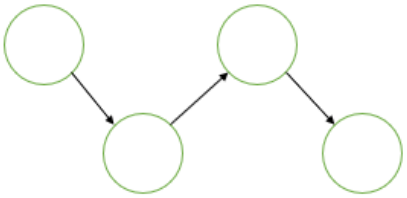


Figure 5 Directed Graph

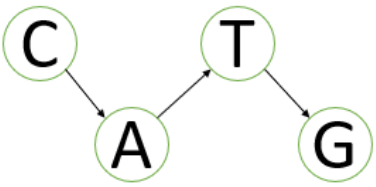



Figure 6 Directed Graph with Assigned Values

Graph is a data structure make of set of nodes and edges. Between nodes, there can be an edge connect them (shown in **Figure 3**). We can also assign different kind of value on each node, and for edge, it can be directed or undirected (shown in **Figure 4 & 5)**. By combining those properties, a directed graph with assigned values (shown in **Figure 6**) can represent sets of sequences.

In the project, there are many specific terms about graph, the following are the terminology that will be mentioned frequently.

### Predecessor Nodes and Successor Nodes

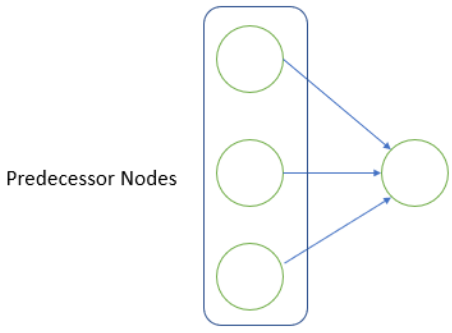



Figure 7 Example of Predecessor Nodes

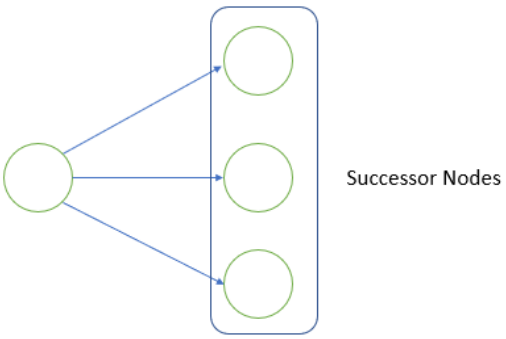



Figure 8 Example of Successor Nodes

**Predecessor node** are the node has a directed edge pointing to the current node, In **Figure 5**, it shows a set of predecessor nodes.

**Successor node** are the node that the current node has a directed edge pointing to, In **Figure 6**, it shows a set of successor nodes.

### In-degree and Out-degree

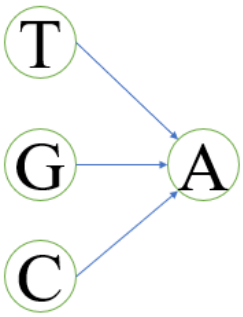



Figure 9 Example of In-degree

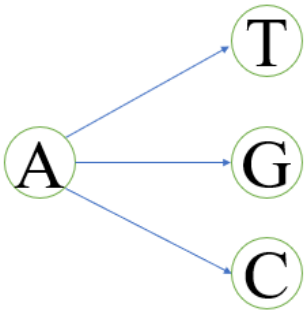



Figure 10 Example of Out-degree

**In-degree** are the number of directed edge point to the current node. In **Figure 9**, There are three directed edge pointing to node A, therefore, the in-degree of node A is 3.

**Out-degree** are the number of directed edges that the current node points out. In **Figure 10**, node A point out three directed edges, therefore, the out-degree of node A is 3.

### Direct Acyclic Graph

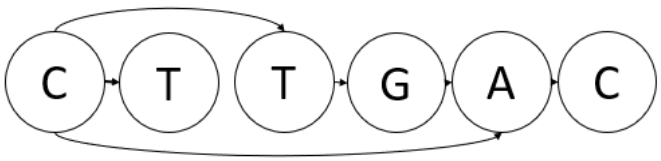



Figure 11 Direct Acyclic Graph

**Direct Acyclic Graph** is a directed graph without any directed cycle, which mean we cannot find any loop in the graph. and from any node in the graph, we can always reach a end point through the directed edge. In the project, the algorithm that I develop will only support the directed acyclic graph.

## Genome sequence

Genome Sequence is the canonical structure of DNA, which is make of the combination of four kinds of nucleobases which is Thymine (denote by ‘T’), Adenine (denote by ‘A’), Cytosine (denote by ‘C’) and Guanine (denote by ‘G’). Genome Sequence are usually very long, for example, the length of the human’s DNA genome sequence is over 3 billion. However, many genome sequences are similar, which are sharing a common part ( also known as pan-genomic, a strains of genome sequence share by the same specie).

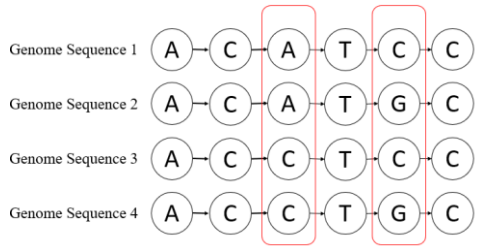



Figure 12 Four Genome Sequences with Similar Structure

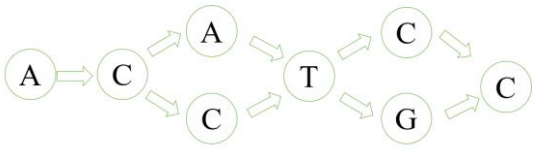



Figure 13 Graph for the Genome Sequences

In **Figure 12**, it shows the four genome sequences with similar structure, they share many common nucleobases and only two of them are variance. If we store the genome sequence by this format, it will waste a lot of memory and cause a redundancy problem, which mean when we are evaluating those sequences, we will have many repeating works. By make use of the properties of graph, we can create a directed acyclic graph for the set of genome sequences (shown in **Figure 13**). Each possible path in the graph spell a unique genome sequence.

| Storing method | Number of nodes | Number of edges |
|---|---|---|
| Graph of genome sequences | 8 | 9 |
| List of genome sequences | 24 | 20 |

Table 1 Comparation of a List of Genome Sequences Against the Graph of Genome Sequences

From **Table 1**, we can see that, the graph of genome sequences, help save up a lot of redundancy part and memory. As the number of nodes decrease, the data that the program need to process decrease too, which mean it will also enhance the efficiency.

## Bitap Algorithm

Bitap Algorithm (as known as Shift-and Algorithm, introduce by Bálint Dömölki in 1964) is a algorithm for finding the exact match index for a sequence against another sequence. Bitap algorithm makes use of the bitwise techniques, it creates a bit array with the size of the searching sequence, and use '0' to indicate match character and use '1' to indicate mismatch character. The condition for a flag being match is that, the current character of the sequence should be the same with the current character of the graph. In addition, the flag before it should also be '0'. Once the last bit in the array turn into '0', it indicates that the path is found. And if all the searching process ended and the last bit of the array is still '1', it means the path is not found.

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | G | C | A | T | C | G | C | A | G | A | G | A | G | T | A | T | A | C | A | G | T | A | C | G |
| 0 | G | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | C | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | A | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | G | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | A | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | G | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6 | A | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | G | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 14 Example of Bitap Algorithm

In **Figure 14,** it shows an example of the Bitap Algorithm. The column is the character in a big sequence and the row is the character in the sequence need to be searched. At index 12, the array[7] turn into 0, and it means that the path is found. As the length of the sequence is 8, therefore the starting position in the graph is 12 – 8 + 1 = 5.

## Standard Dynamic Programming

Standard Dynamic Program algorithm (introduced by Richard Bellman in 1950s) for finding the minimum edit distance between two sequence. The definition of edit distance is the editing operations including the insertion, deletion and substitution.

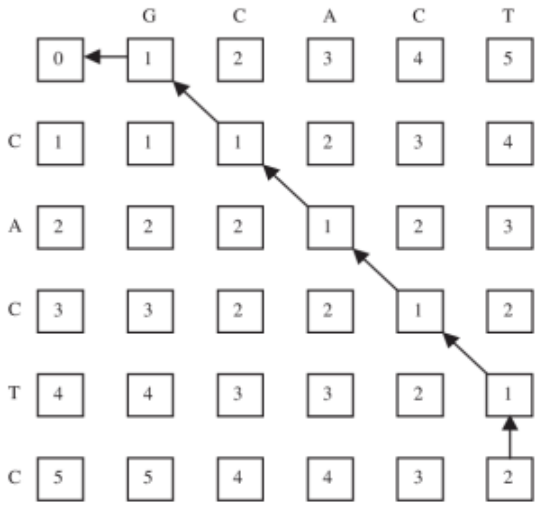



Figure 15 Example of Dynamic Programming

$$\begin{cases} D\ (i-1,\ j)+1 \\ D\ (1, j-1)+1 \\ D(i-1, j-1)+(1\ if\ x[i]=y[j], 0\ if\ x[i]\neq y[j] \end{cases}$$

Figure 16 Reference Formula for DP

With the use of the Dynamic programming, we put the edit distance cost in a 2D-array form (shown in **Figure 15**) such that we can reuse the result of other overlap subproblem.
For each index inside the arrays, its edit distance is the minimum value from the formula shown in **Figure 16.** After the process, the button right index of the array will show the minimum edit distance.

# Objectives

In the project, I am going to develop two algorithms, each algorithm aims to solve one of the following graph-searching problems.

## Sequence-to-Graph Matching Problem (SGM)

Sequence-to-Graph Matching Problem means that for a directed acyclic graph with assigned values (each node contains one alphabet value) and a searching sequence. The algorithm should find a path (a strain of nodes, where the $i^{th}$ node must point to the $(i+1)^{th}$ node) in the graph such that the values inside the path is exactly match the searching sequence. The following is an example to explain what the expected output for the algorithm is to solve the Sequence-to-Graph Match Problem.

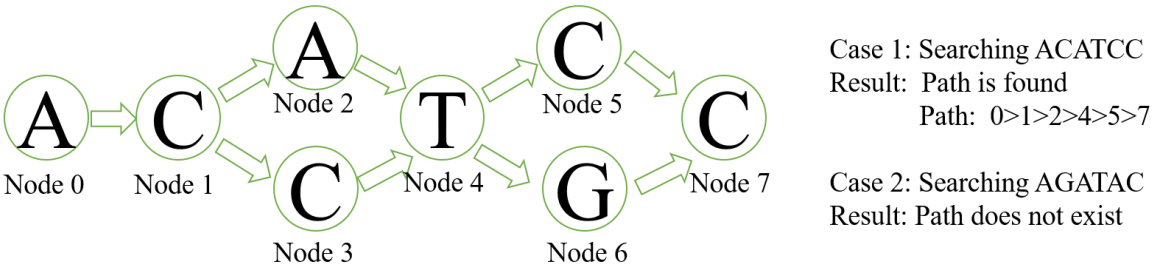



Figure 17 Expected Output of the Algorithm for the SGM Problem

In **Figure 17**, it shows a sample graph with two examples for the expected output for the algorithm. In case 1, the searching sequence is ACATCC, the algorithm should report that the path is found and return the path to the user, which is a set of nodes with the index 0, 1, 2, 4, 5, 7. In case 2, the searching sequence is AGATAC, it is impossible to find a path in the graph that exactly match the searching sequence, therefore, the algorithm needs to report that the path does not exist and does not need to provide any path.

## Sequence-to-Graph Alignment Problem (SGA)

Sequence-to-Graph Alignment problem means that for a directed acyclic graph with assigned values (each node contains one alphabet value) and a searching sequence. The algorithm should find a path (a strain of nodes, where the $i^{th}$ node must point to the $(i+1)^{th}$ node) in the graph such that the values inside the path has a minimum edit distance with the searching

sequence compare to other possible path. The following is an example to explain what the expected output for the algorithm is to solve the Sequence-to-Graph Alignment problem.

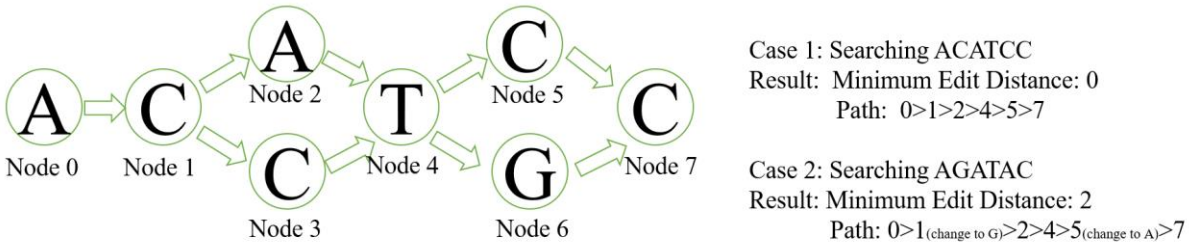



Figure 18 Expected Output of the Algorithm for the SGA Problem

In **Figure 18**, it shows a sample graph with two examples for the expected output for the algorithm. In case 1, the searching sequence is ACATCC, the algorithm should report that the minimum edit distance is 0 and return the path to the user, which is a set of nodes with the index 0, 1, 2, 4, 5, 7. In case 2, the searching sequence is AGATAC, the algorithm should report that the minimum edit distance is 2 and return the path to the user, which is a set of nodes with the index 0, 1, 2, 4, 5, 7. For node 1 and 5, the algorithm should also tell the user about the edit process.

Besides, extract matching is a subset of minimum edit distance because a path that exact match the searching sequence is identical to the 0-edit distance. Which mean the algorithm that can solve the SGA problem must be also able to solve the SGM problem. However, if I develop a algorithm that focus on solving the SGM problem, it can have a better time complexity, therefore, I decided to divide the problem into SGM and SGA.

# Methodology

In the project, I develop two algorithms for solving the problem mention in the objective section, one for the Sequence-to-Graph Matching problem, and another for the Sequence-to-Graph Alignment problem. In the methodology section, I will introduce how the algorithm works.

## Exact Matching Algorithm

This algorithm is developed for solving for the sequence-to-graph matching problem mentioned in the objection section. This algorithm based on the Bitap Algorithm, during the development of the algorithm, I have made some adjustment on the algorithm, so that the algorithm can support the direct acyclic graph.

One of the advantages of the algorithm is it does not need to consider all the possible path in the graph but only need to have one process one each node in the graph, which save a lot of time in searching.

In order to have a fully explain on the algorithm, I am going to explain the algorithm by illustration.

### Algorithm illustration

In the illustration, I will explain the process of the algorithm with a simple example. In **Figure 19**, it shows out the graph that I will use in my explanation.

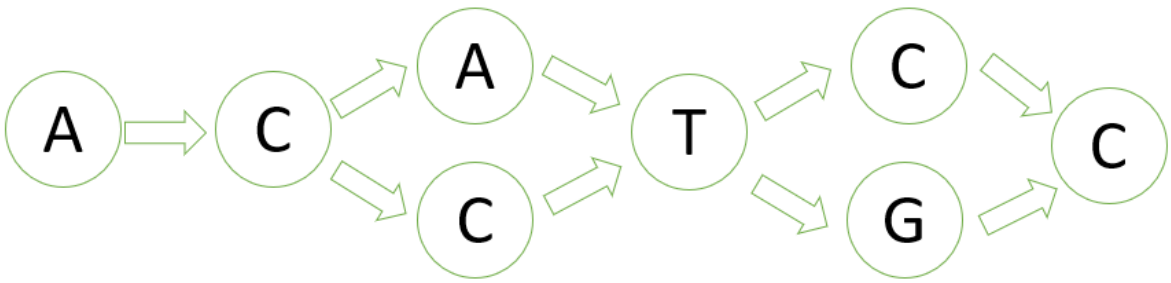



Figure 19 A Graph for the Illustration of Exact Matching Algorithm

Besides, in the illustration, I will try to search a sequence “CATG” in the graph

**Step 1: Initialize the Pattern Mask for Each Alphabet in the Searching Sequence**

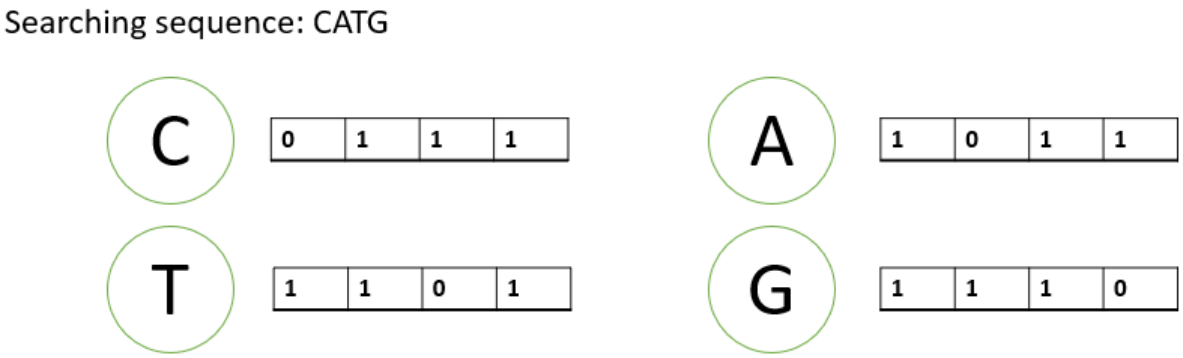



Figure 20 Pattern Masks for Each Alphabet in the Sequence

The first step of the algorithm is to initialize the pattern mask for each possible alphabet in the searching sequence, the size of the pattern mask is equal to the length of the searching sequence. and for each box in the pattern mask means that the matching condition of the alphabet with the corresponding character in the searching sequence, where 0 indicate a match and 1 indicate an unmatched. For example, C appears only in searchSequence[3], therefore, its pattern mask will be [0, 1, 1, 1].

**Step 2: Initialize an NFA State Array for Each Node in the Graph**

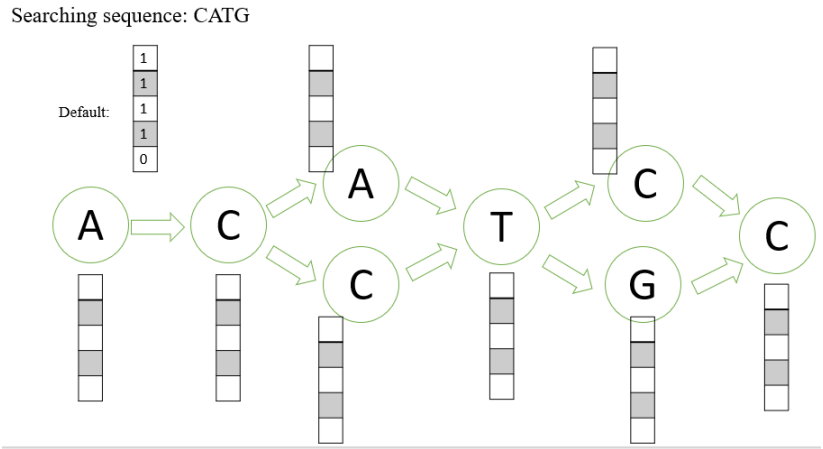



Figure 21 Initializing the NFA State Array for Each Node in the Graph

NFA stand for nondeterministic finite automaton, which is used to define which successor node is the next node according to different conditions (details of the usage of NFA state will be mention in back-tracking step). The size of the NFA state array for each node is m + 1 where m is the length of the searching sequence. The default NFA states is 0 for the last index in the array and 1 for the rest.

**Step 3: Calculate the NFA State for Each Node**

The second step is to calculate the NFA state for every node in the graph, However, different types of nodes may have different handle process. In **Figure 22**, show that there are four type of nodes in a graph. The nodes with red circle are the node with in-degree is 0, which also known as one of the starting points in the graph. The nodes with green circle are the nodes with in-degree is 1. The nodes with blue circle are the nodes that share the same predecessor node with other nodes. The nodes with purple circle are the nodes with in-degree more than 1. In the following part, I will describe the handle process base on those four cases.

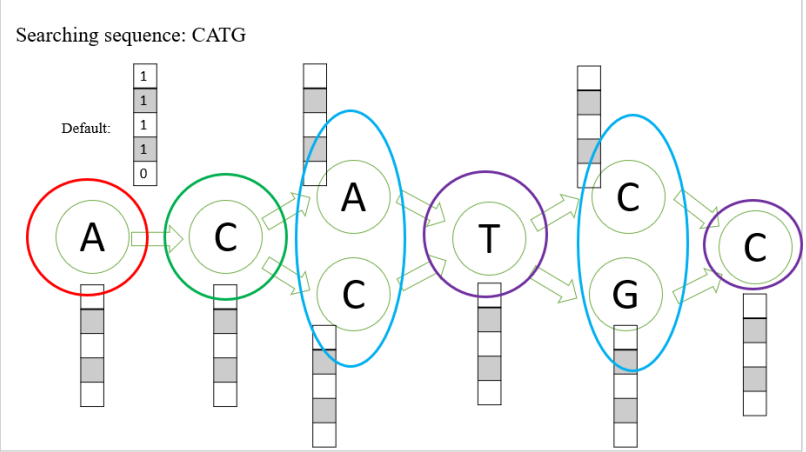



Figure 22 Different Types of Node in the Graph

**Step 3a: NFA Calculation for the Nodes with In-degree is 0**

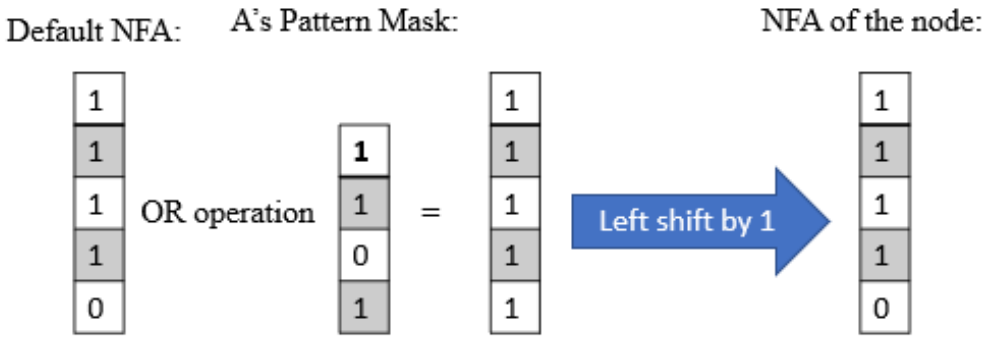



Figure 23 NFA Calculation Process for the Nodes with In-degree is 0

For the nodes with in-degree is 0, we first have an OR operation with the Default NFA array with the pattern mask of the alphabet value of the current node, after the operation, we left-shift it by 1 bit. The result of the shifting is the NFA of the node.

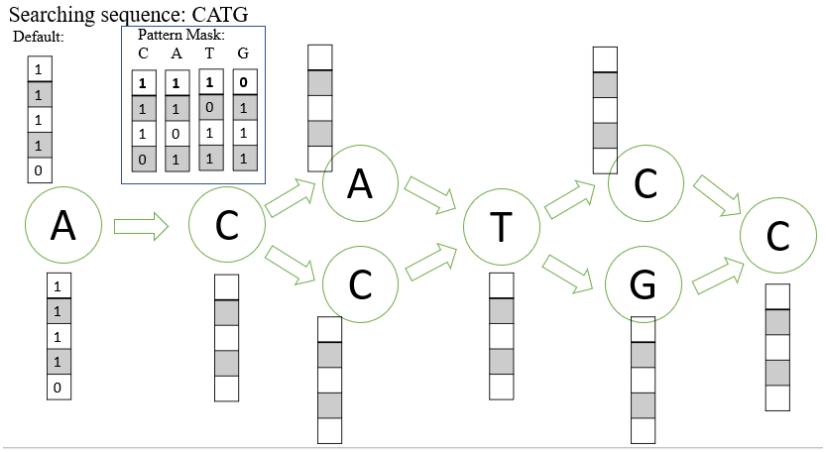



Figure 24 Result of the NFA Calculation Process for the Node with In-degree is 0

**Step 3b: NFA Calculation Process for the Node with In-degree is 1**

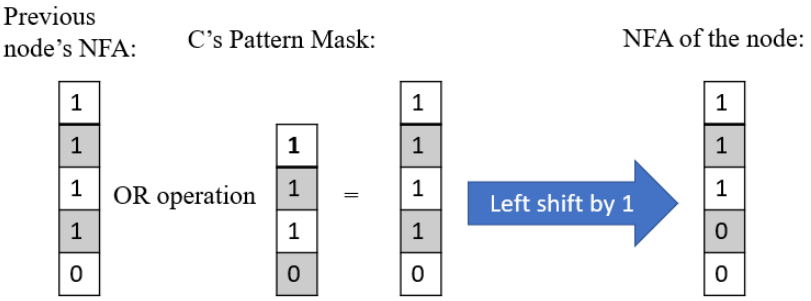



Figure 25 NFA Calculation Process for the Nodes with In-degree is 1

For the nodes with in-degree is 1, we first have an OR operation with the predecessor nodes' NFA array with the pattern mask of the alphabet value of the current node, after the operation, we left- shift it by 1 bit. The result of the shifting is the NFA of the node.

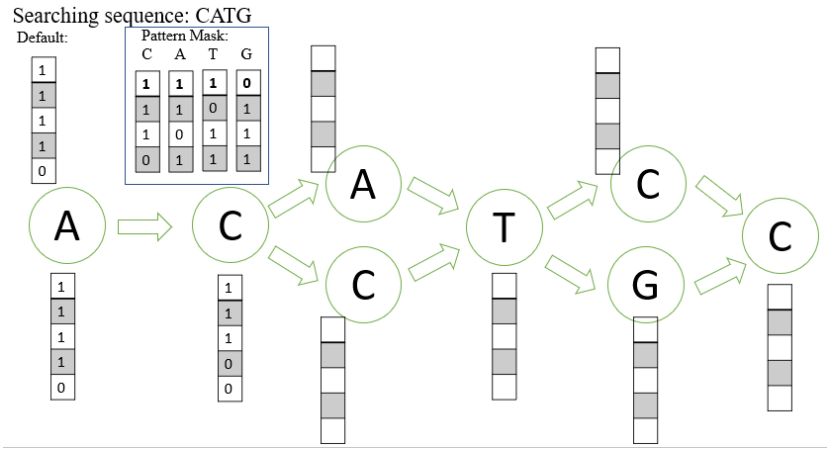



Figure 26 Result of the NFA Calculation Process for the Node with In-degree is 1

### Step 3c: NFA Calculation Process for the Nodes Share the Same Predecessor Node

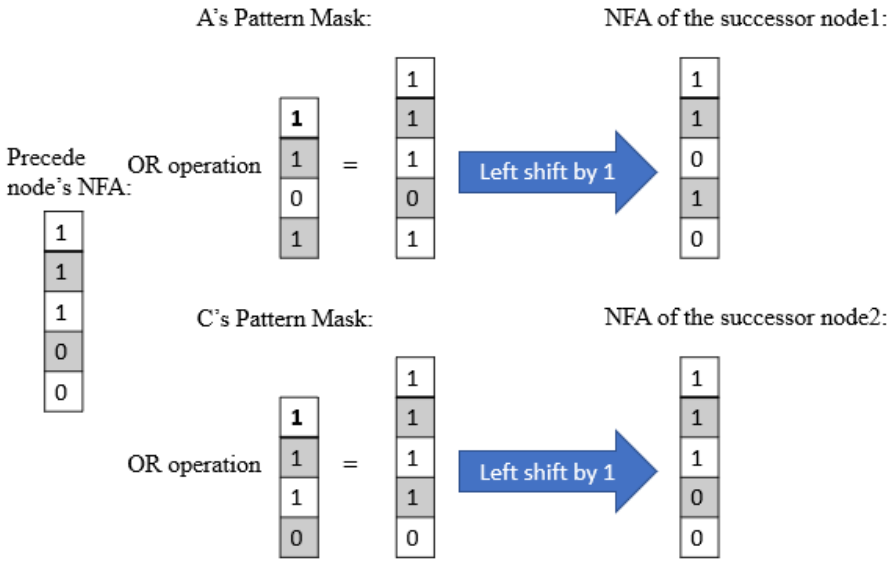



Figure 27 NFA Calculation Process for the Nodes Share the Same Predecessor Node

For the nodes that share the same predecessor node, all the successor node need to make used of the NFA array of their predecessor nodes and have an OR operation with the with the pattern mask of the alphabet value of the their node, after the operation, we left-shift them by 1 bit respectively. The results of the shifting is the NFA of the successor nodes respectively.

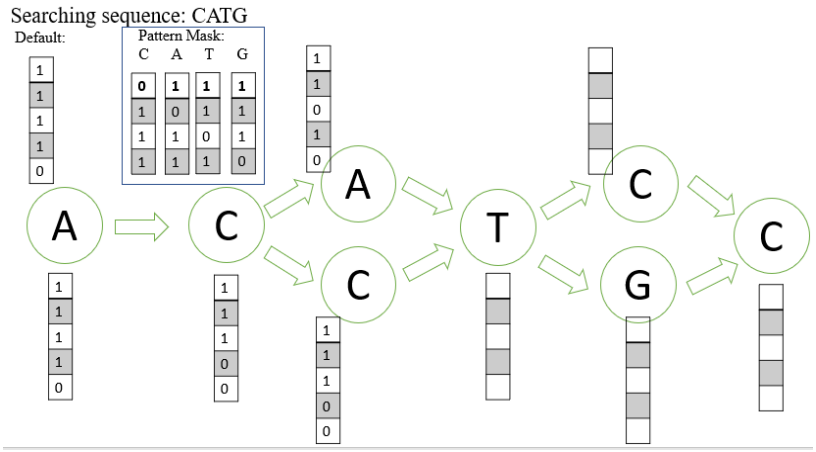



Figure 28 Result of the NFA Calculation Process for the Nodes Share the Same Predecessor Nodes

**Step 3d: NFA Calculation Process for the Nodes with In-degree More than 1**

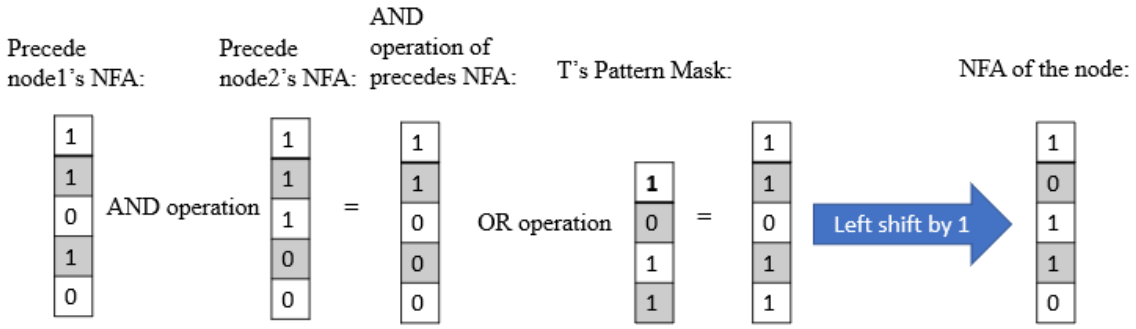



Figure 29 NFA Calculation Process for the Nodes with In-degree More than 1

For the nodes with in-degree more than 1, which mean that those nodes have more than one predecessor node, we need to first have the AND operation with all the predecessor nodes of the current node. After the AND operation, we treat the result NFA as the only one predecessor node's NFA of the current node and take an OR operation with the pattern mask of the alphabet value of the current node, after the operation, we left-shift the result by 1 bit. The results of the shifting is the NFA of the current node.

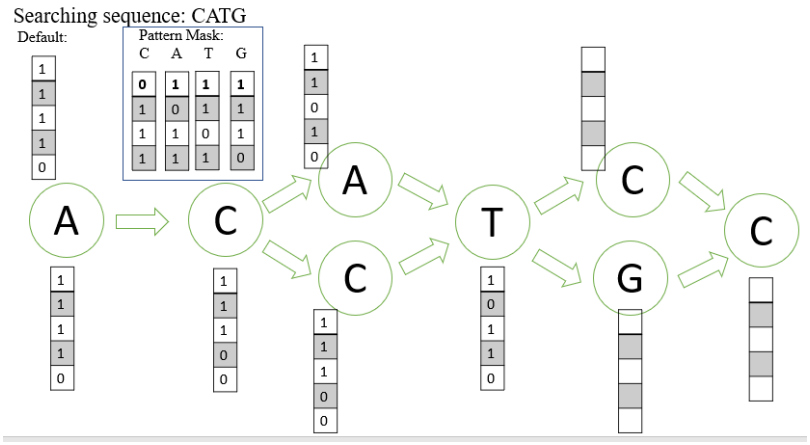



Figure 30 Result of the NFA Calculation Process for the Nodes Have More than One Predecessor Nodes

### Step 3e: Finish of the Algorithm

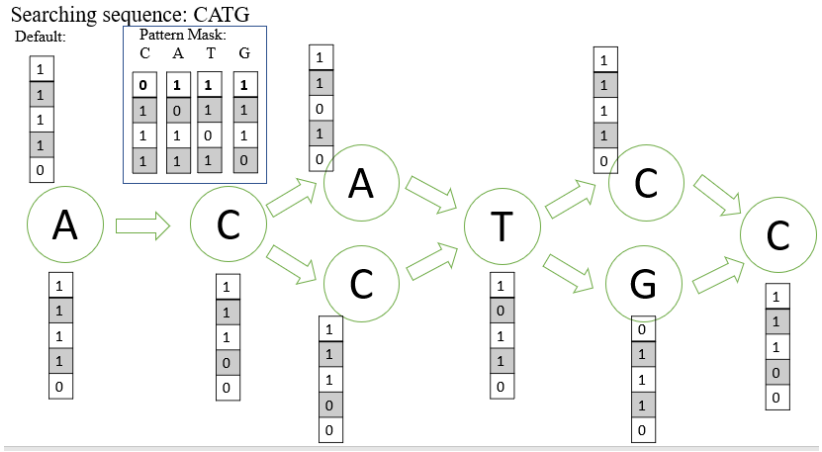



Figure 31 Finish of the Algorithm

After NFA calculation process for every nodes in the graph, we finally get the result, in the reality case, the algorithm will stop once it discovers there is a NFA turn it first bit into zero (which is the NFA of the node G in the bottom right). However, to illustrate the whole NFA calculation process, I show all the NFAs in **Figure 31**.

**Step 4: Back-tracking to Find the Path**

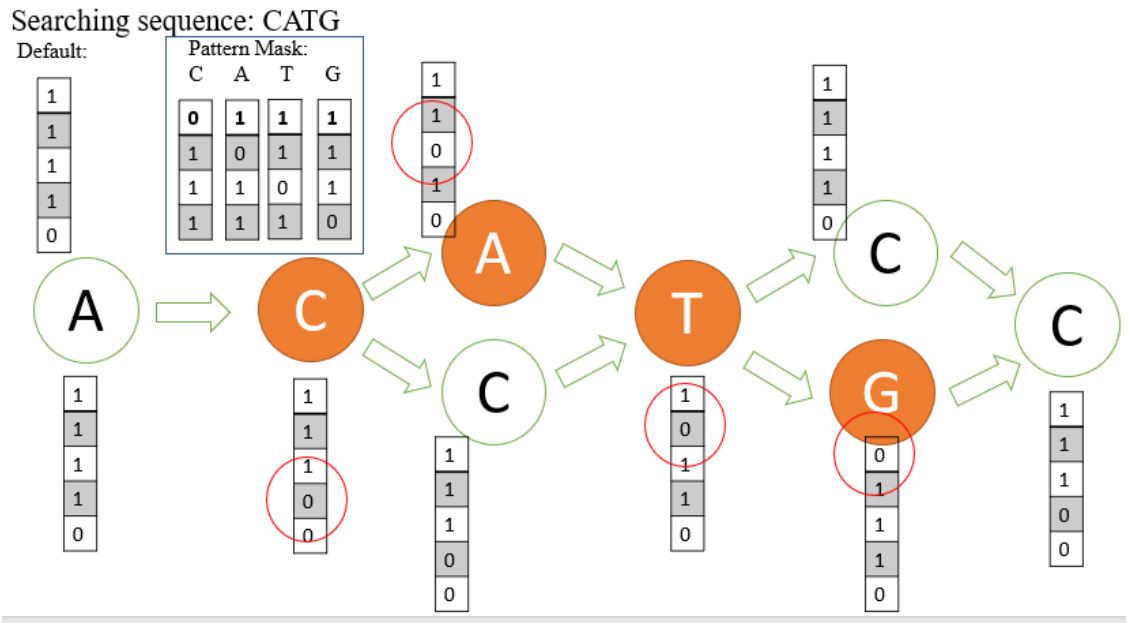



Figure 32 Result of the Back-tracking process

The back-tracking process will start at the node that its NFA turn it first bit into 0, the node must be the ending node of the path, therefore, back-tracking process will push back that node into the path vector. After that, the back-tracking will start the searching by m - 1 times (where m is the length of the searching sequence), in each searching process, the algorithm will check all the predecessor nodes of the current node, if the m + 1 – t (where t is the searching time) bit in the predecessor node turn into 0, it indicate that the node is also a node in the path, the algorithm will also push back it into the path vector and set it to be the current node.

## Pseudo Code of the Exact Matching Algorithm

Exact Matching Algorithm

***Input:***

**G** ← the input graph

**M** ← a searching sequence

***Output:***

Index of the last node of the path

NFA states for each node in the graph

***Process:***

```
P ← an array of pattern mask which initialize at all 1 bit
M ← length of the searching sequence

for i < m do
    P[M[i]] = P[M[i]] and (1 << i)
For each node V ∈ G do
    if Indegree(V) == 0 then
        V.NFA = V.NFA OR P[V.value]
    else
            for each predecessor node j of V do
                V.NFA = V.NFA AND J.NFA
            V.NFA = V.NFA OR P.NFA
    If V.NFA & (1<<m) then
        return V.index
return -1
```

Figure 33 Pseudo Code of the Exact Matching Algorithm

## Pseudo Code of the Backtracking for the Exact Matching Algorithm

```
Back tracking for the Exact Matching Algorithm
Input:
Index ← Index of the last node of the path
G ← the input graph
m ← length of the searching sequence
NFA states for each node in the graph
Output:
A path that contain the matching nodes in the graph
Process:
P.endIndex ← I
V ← G[Index]
Path.push_back(V)
while m > 0 do
    for each predecessor nodes i of V do
```

```
        if j.NFA AND (1 << m) == 0 then
            Path.push_back(j)
            V = j
            m = m – 1
            break
return Path
```

Figure 34 Pseudo Code of the Backtracking for the Exact Matching Algorithm

## Minimum Edit Distance Algorithm

This algorithm is developed for solving for the sequence-to-graph alignment problem mentioned in the objection section. This algorithm based on the dynamic programming, during the development of the algorithm, I have made some adjustment on the algorithm, so that the algorithm can support the direct acyclic graph.

One of the advantages of the algorithm is the algorithm is much faster than recursive function. In recursive function, it makes use of divide and conquer approach and creates numbers of overlapped sub-problem, which mean the algorithm require us to calculate the same problem for many times, which waste a lot of time. By make use of the dynamic programming approach, we can calculate the result of the subproblem once and save the result into the array. If we encounter the sub-problem next time, we can make use of the value in the array and do not need to calculate the subproblem again.

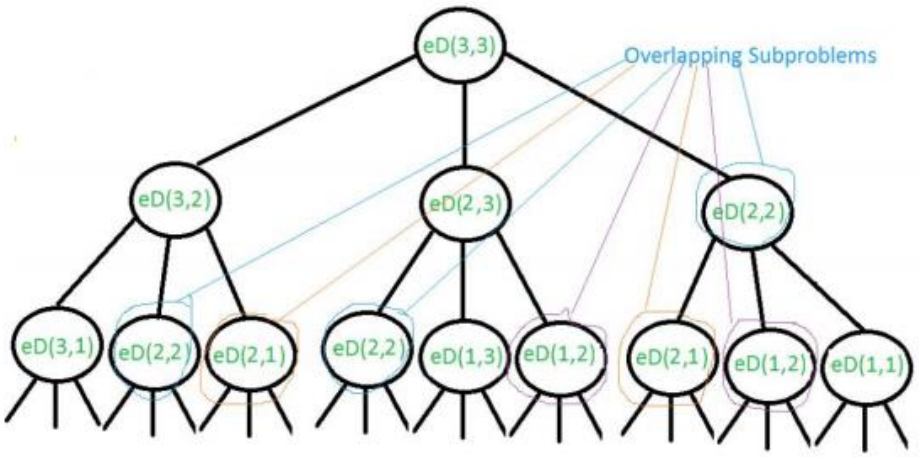



Figure 35 Major Problem of Recursive Function

In order to have a fully explain on the algorithm, I am going to explain the algorithm by illustration.

**Illustration of the Minimum Edit Distance Algorithm**

In the illustration, I will explain the process of the Minimum Edit Distance Algorithm with a simple example. In **Figure 36**, it shows out the graph that I will use in my explanation.

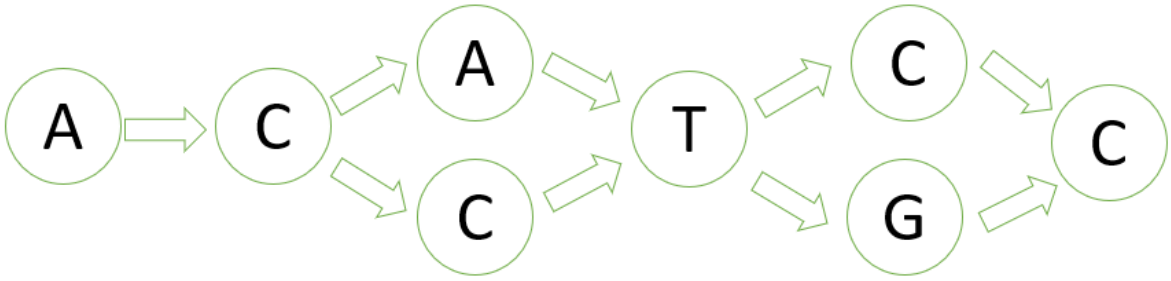



Figure 36 Graph Used in the Illustration of Minimum Edit Distance Algorithm

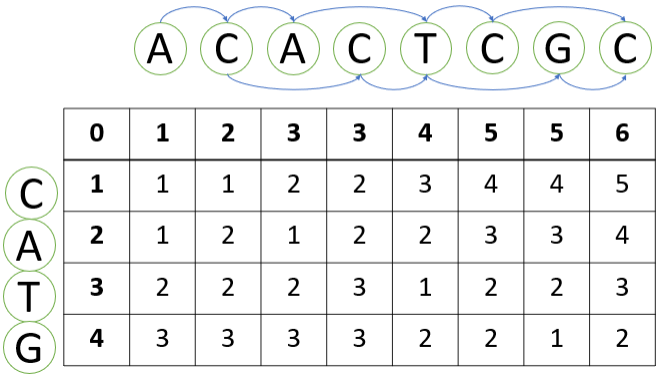
Besides, in the illustration, I will try to search a sequence “CATG” in the graph

**Step 1: Initialize the Dynamic Programming Array**

The first step of the algorithm is to initialize the dynamic programming array. The dynamic programming array is a 3d array with the size of (the length of the searching sequence + 1)*(the number of nodes in the graph + 1)*3 layers.

In the dynamic array, there are 3 layers, the first layer **ED** represent the edit distance layer, the value inside the array indicate the edit distance for the search sequence turn into the graph. The second layer **ARROW** represent the arrow pointing layer, which show the node that the current node pointing to, ARROW layer is important for the back-tracking process (detail will be in back-tracking step). The third layer **STATE** represent the edit state layer, which indicate that the edit process for each box in the array, S for same and keep unchanged, C for not same and change the alphabet, I for insert an alphabet from the node in the graph and R for removing an alphabet from the searching sequence.

ED --- Edit Distance Layer




| | 0 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|
| C | 1 | 1 | 1 | 2 | 2 | 3 | 4 | 4 | 5 |
| A | 2 | 1 | 2 | 1 | 2 | 2 | 3 | 3 | 4 |
| T | 3 | 2 | 2 | 2 | 3 | 1 | 2 | 2 | 3 |
| G | 4 | 3 | 3 | 3 | 3 | 2 | 2 | 1 | 2 |

Figure 37 Edit Distance Layer in the Dynamic Programming Table

ARROW --- Arrow Pointing Layer

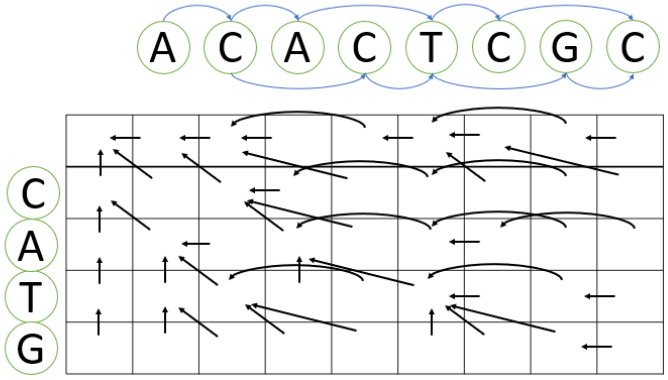
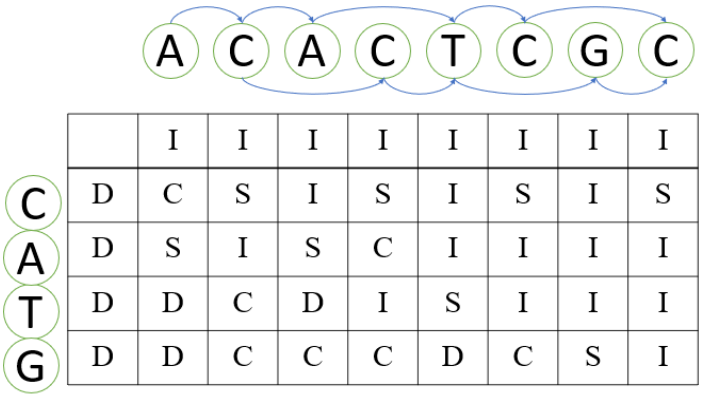



Figure 38 Arrow Pointing layer in the Dynamic Programming Table

STATE – Edit State Layer




| | | I | I | I | I | I | I | I | I |
|---|---|---|---|---|---|---|---|---|---|
| C | D | C | S | I | S | I | S | I | S |
| A | D | S | I | S | C | I | I | I | I |
| T | D | D | C | D | I | S | I | I | I |
| G | D | D | C | C | C | D | C | S | I |

Figure 39 Edit State Layer in the Dynamic Programming Table

**Step 2: Process for the Upper Left Corner**

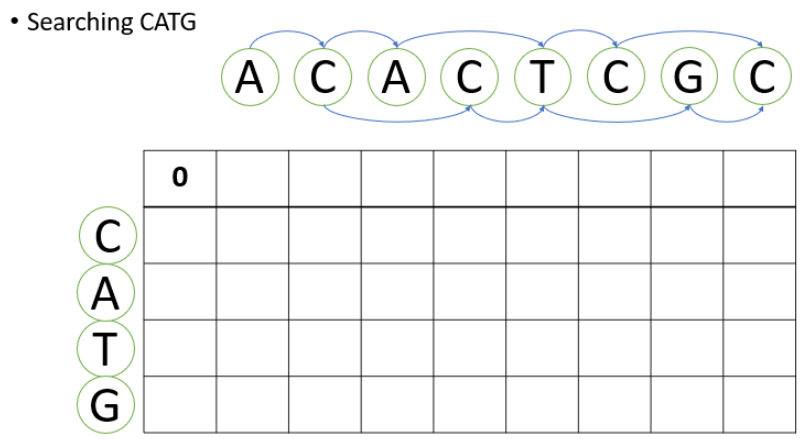



Figure 40 Dynamic Programming Table After the Process for the Upper Left Corner

The second step is the process for the upper left corner in the dynmaic programming table, the edit distance for the upper left corner must be 0, and it does not have any edit state and any poting arrow.

**Step 2: Process for the First Column**

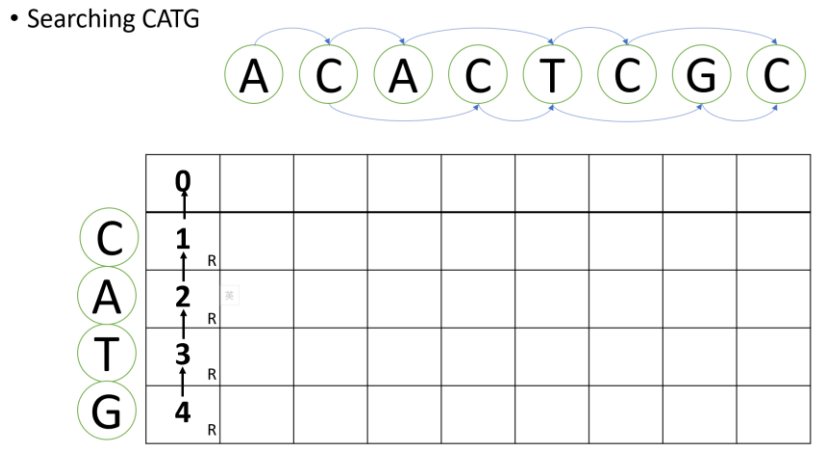



Figure 41 Dynamic Programming Table after the Process for the First Column

The second step is to process for the first column in the dynamic programming table, for all elements in the first column, their edit distance is the edit distance of the up-neighbour +1 and arrow always point to their up-neighbour. Besides, their edit state must be R, which indicate that the edit process is removing the current alphabet in the search sequence.

**Step 3: Process for the first row**

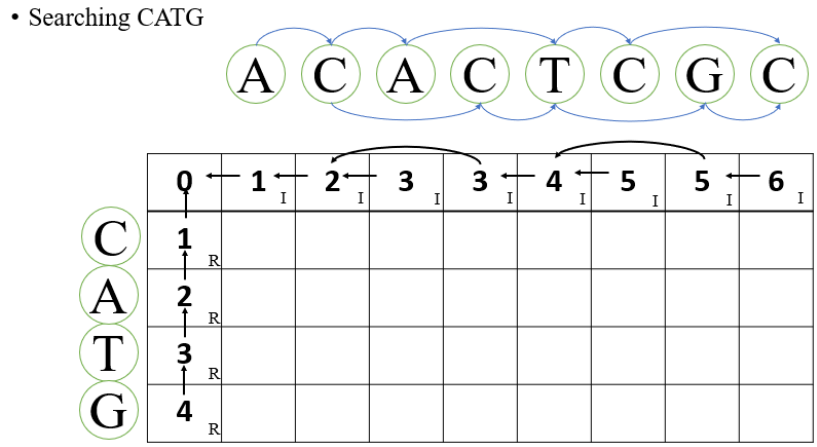



Figure 42 Dynamic Programming Table after the Process for the First Row

The third step is to process for the third column in the dynamic programming table, for the elements in the first row, there are three types of node and different type of nodes have different process. The following are the handle process for those three types of node.

**Node with in-degree is 0**

For the node with in-degree is 0, their edit distance must be 1. In the arrow pointing layer, it should point to the upper left corner. In the edit state layer, its edit state must be an insert state. In **Figure 42**, the first node in the graph (with the value 'A') is an example of a node with in-degree is 0, its edit distance is 0, edit state is insert, and its arrow is pointing to the upper left corner.

**Node with in-degree is 1**

For the nodes with in-degree is 1, their edit distance is the edit distance of its predecessor node's + 1. In the arrow pointing layer, it should point to the side neighbour of its predecessor node. In the edit state layer, its edit state must be an insert state. In **Figure 42**, the second node in the graph(with the value 'C') is an example of a node with in-degree is 1, its edit distance is 2, which is its predecessor node's edit distance + 1 (1 + 1 = 2), its edit state is insert and its arrow is pointing to the side neighbour of its predecessor node.

**Node with in-degree more than 1:**

For the nodes with in-degree more than 1, it should select one of its predecessor nodes with the minimum edit distance, its edit distance is the edit distance of the predecessor node that it selected + 1. In the arrow pointing layer, it should point to the side neighbour of the selected predecessor node. In the edit state layer, its edit state must be an insert state. In **Figure 42**, the fifth node in the graph (with the value 'T') is an example of a node with in-degree more than 1, its edit distance is 4, which is its predecessor node with the minimum edit distance + 1 (its predecessor nodes are the third node and fourth node in the graph, however, both of their edit distance are 3, in this case, we select the node with the minimum index, which is the third node), its edit state is insert and its arrow is pointing to the side neighbour of the selected predecessor node.

**Step 4: Process for the rest of the table**

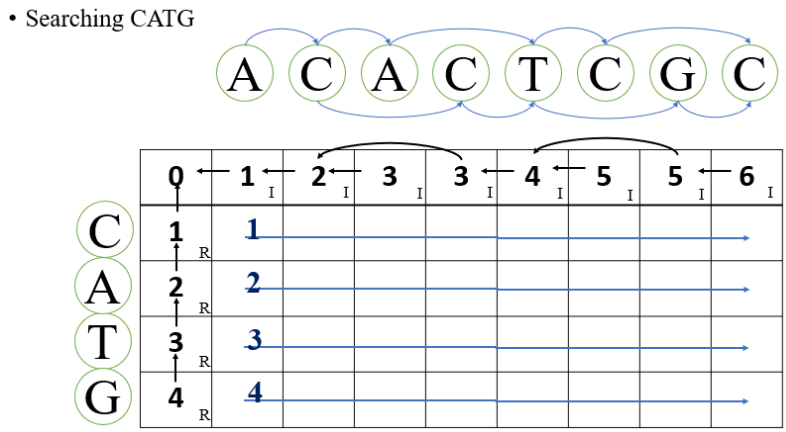



Figure 43 Order for the Process for the Rest in the Dynamic Programming Table

In the fourth step, we need to process the rest of the table, the order of the process is from up to down, left to right. In **Figure 43**, it shows the order of the process for the rest of the table. During the process, the process need to consider the matching condition (either match or not match) and the indegree for the current node (0, 1, or more than 1), and the combination of the consideration create 6 possible cases (2 * 3 = 6). The following are the flow chart of the handle processes for different cases and some example for illustrating the handle process for different case.

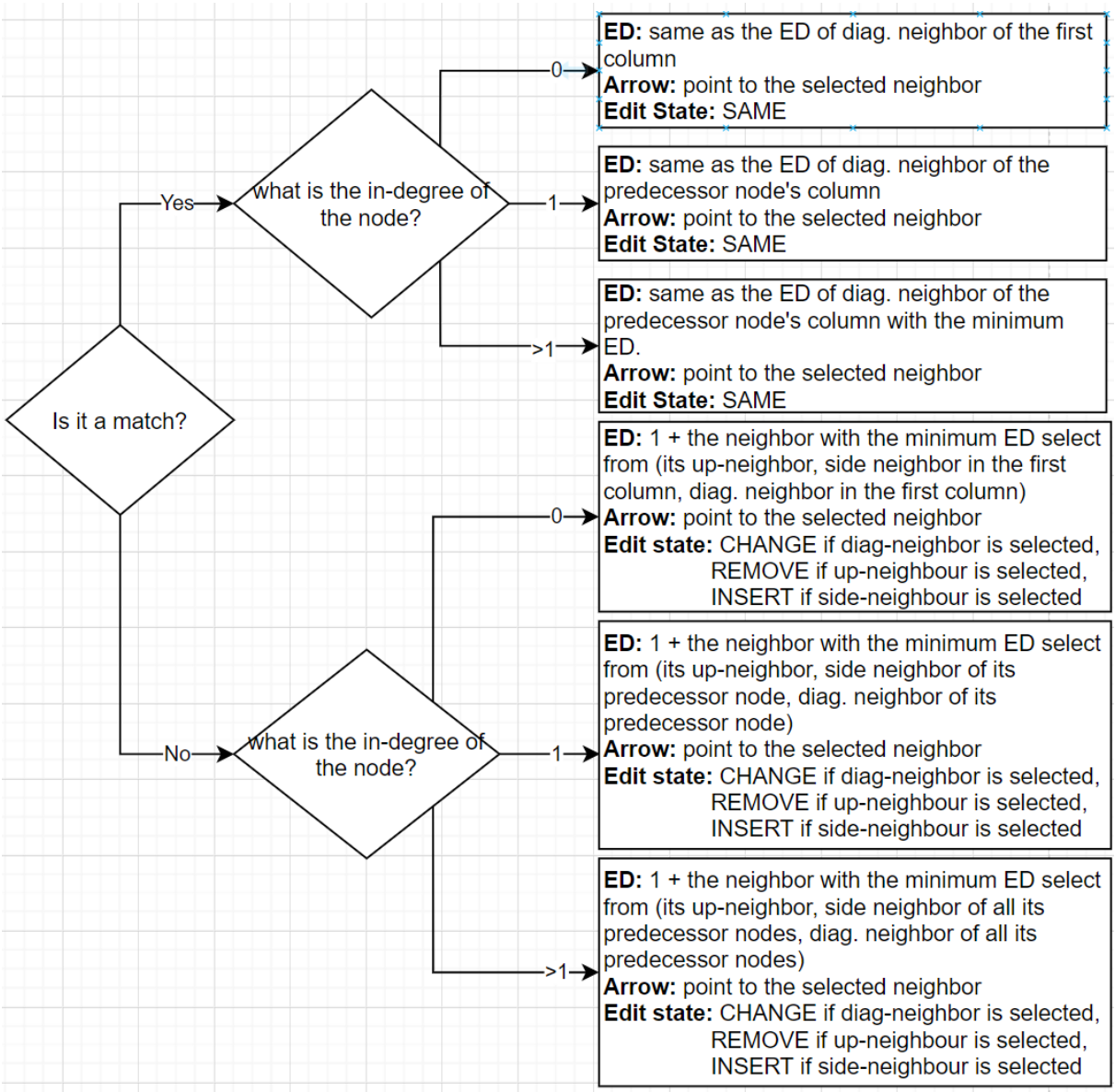



Figure 44 Flowchart for Processing the Rest of the Table

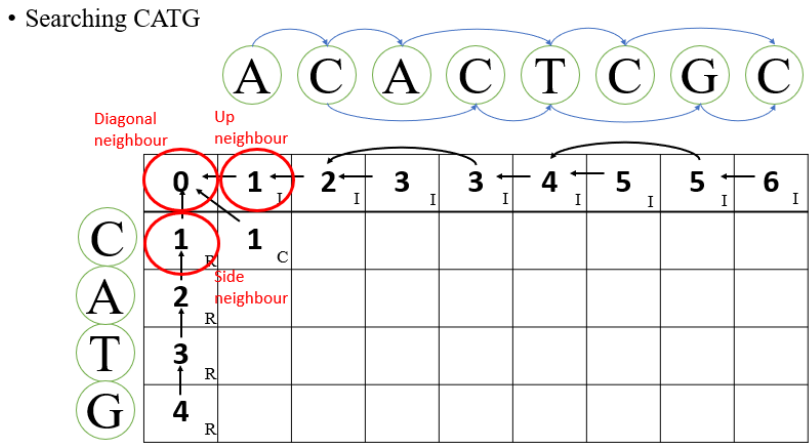



Figure 45 Dynamic Programming Table after the Process for not Match Case with In-degree is 0

For the not match case (‘C’ in the search sequence and ‘A’ in the graph is not a match) with in-degree is 0, the edit distance is selected from the diagonal neighbor in the first column, up neighbor and side neighbor in the first column, the process should choose the neighbor with the minimum edit distance and + 1 for edit penalty. In this case, the neighbor with the minimum value is the diagonal neighbor in the first column, therefore, the edit distance for this case will be 1 (0 + 1 = 1), the arrow is point to the selected neighbor, and the edit state is CHANGE because the diagonal neighbor is selected.

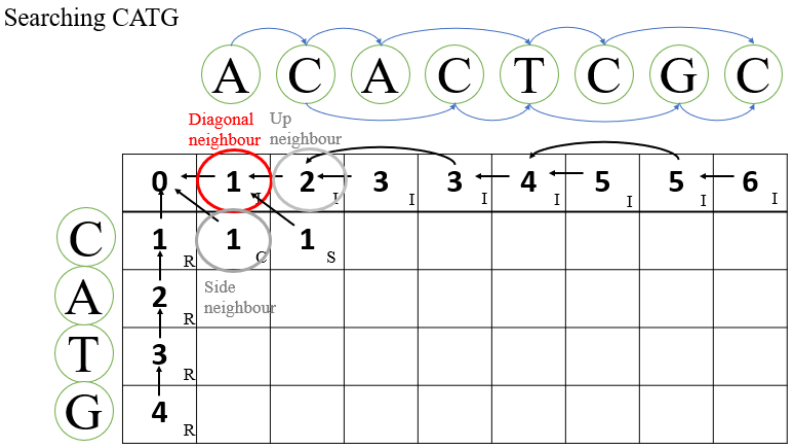


Figure 46 DP Table after the Process for Match Case with In-degree is 1

For the match case (‘C’ in the search sequence and ‘C’ in the graph is a match) with in-degree is 1, the edit distance is selected directly from the diagonal neighbor of its predecessor node and no edit penalty is needed. In this case, the edit distance will be 1 (take from the diagonal neighbor), the arrow is point to the selected neighbor, and the edit state is SAME for the match case.

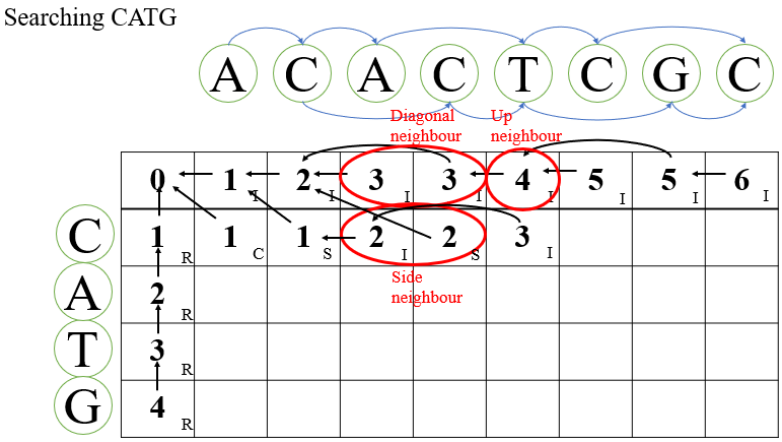


Figure 47 Dynamic Programming Table after the Process for Not Match Case with In-degree More than 1

For the not match case (‘C’ in the search sequence and ‘T’ in the graph is not a match) with in-degree more than 1, the edit distance is selected from the diagonal neighbor in the all its up neighbor, predecessor’s node and side neighbor of all its predecessor’s node, the process should choose the neighbor with the minimum edit distance and + 1 for edit penalty. In this case, the neighbor with the minimum value is the side neighbor in the fourth column, therefore, the edit distance for this case will be 3 (2 + 1 = 13), the arrow is point to the selected neighbor, and the edit state is INSERT because the side neighbor is selected.

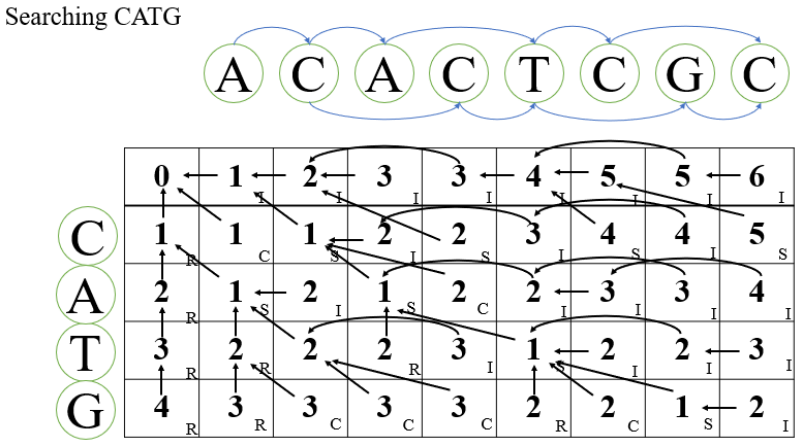


Figure 48 Dynamic Programming Table after Finishing All Process

After finishing the process for all columns and rows in the dynamic programming array, step 4 is ended.

**Step 5: Back-tracking for the path**

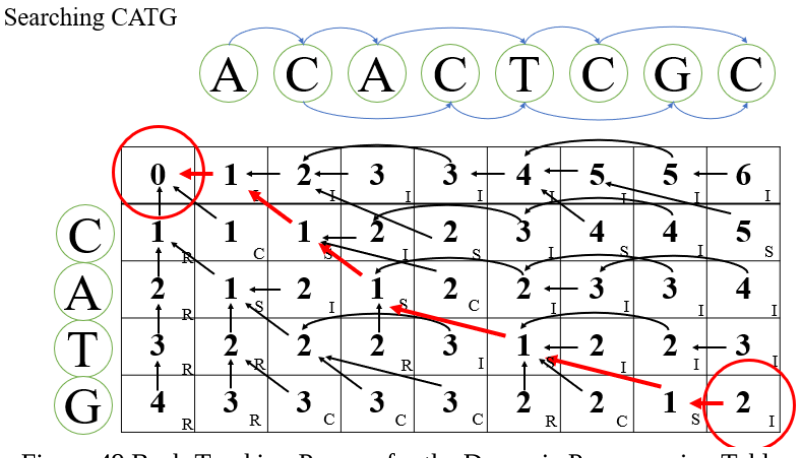


Figure 49 Back-Tracking Process for the Dynamic Programming Table

In the back-tracking process, it will start from the buttom right corner to the upper left corner of the dynamic programming table. The back-tracking process follows the arrow of the current position and the arrows will finally bring us to the upper left corner.

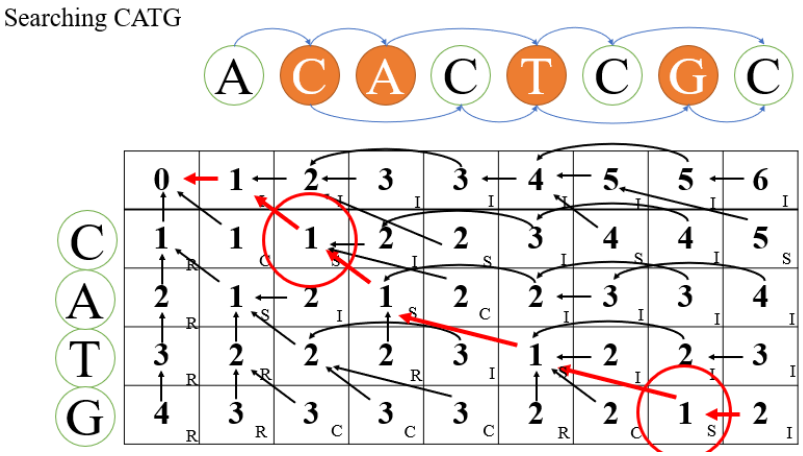


Figure 50 Retrieving the Path from the Back-tracking Result

After the back-tracking process, we need to retrive the path from the result, the ending node of the path is the node that the red arrow starts lift up (as the red circle in the button right in **Figure 50**), and the starting node of the path is the node before the red arrow reach the first row (as the red circle in the upper lift in **Figure 50**). In the example, the path is starting from the second node and end in the seventh node, the full path is CATG (node path: 2>3>5>7).

**Step 6: Calculate the edit distance**

The edit distance is the edit distance of position that the red arrow starts lift up (as the red circle in the button right in **Figure 50**) minus the edit distance of the position before the red arrow reach the first row (as the red circle in the upper lift in **Figure 50**). In the example the edit distance for the path is 0 (1 – 1 = 0).

**Step 7: Retrieving other paths**

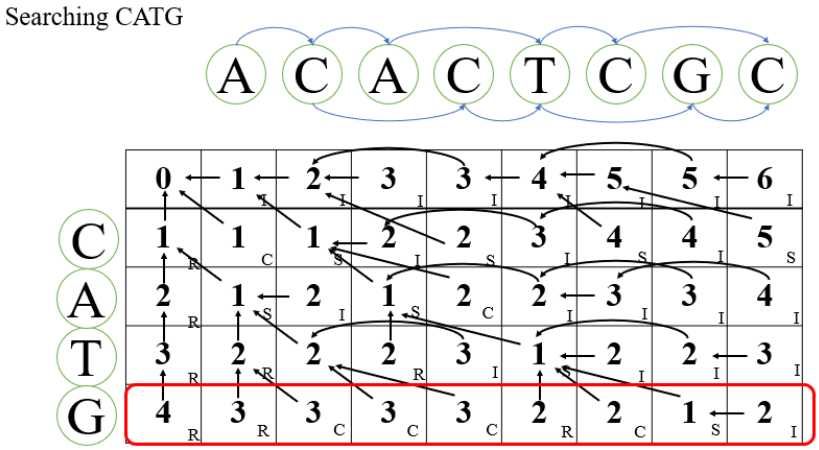



Figure 51 Retrieving Other Paths from the Bottom Row of the Dynamic Programming Table

In the last row of the dynamic programming array, there are v + 1 elements where v is the number of nodes in the graph. and each element in the last row has its own arrow, if we treat each element in the last row as the starting point, and have the back-tracking process for them, we can retrieve v + 1 paths, by ranking the edit distance of the paths, we can finally sort out the top five path with least edit distance.

## Pseudo Code of the Minimum Edit Distance Algorithm

Minimum Edit Distance Algorithm

***Input:***

**G** ← the input graph

**M** ←a searching sequence

***Output:***

DP ← a dynamic programming array of the searching sequence against the graph

***Process:***

DP ← 3d array: [size of G +1][size of M +1][3]

//DP[][][E] = DP[][][0] edit distance score

// DP[][][A] = DP[][][1]  pointing arrow

// DP[][][P] = DP[][][2] edit process

DP[0][0][E] = 0

DP[0][0][P] = 0

```
DP[0][0][A] = 0
for i <= size of M do
    DP[i][0][E] = i
    DP[i][0][A] = 0
    DP[i][0][P] = REMOVE
for i <= size of G do
    V ← G[i]
    for each predecessor node j of V do
        if MIN.editDistance > DP[j][0][E] then
            MIN = DP[j][0]
    DP[j][0][E] = MIN.editDistance
    DP[j][0][A] = MIN.nodeIndex
    DP[j][0][P] = INSERT
for i <= size of M do
    for j <= size of G do
        if G[j]. value == M[i].value then
            for each predecessor node k of V do
                MIN = min(MIN.editDistance, DP[i-1][k][E])
            DP[i][j][E] = MIN.editDistance
            DP[i][j][A] = MIN.nodeindex
            DP[i][j][P] = SAME
        else
            for each predecessor node k of V do
                MIN = min(MIN.editDistance, DP[i-1][j][E], DP[i-1][k][E], DP[i][k][E])
            DP[i][j][E] = MIN.editDistance + 1
            DP[i][j][A] = MIN.nodeIndex
            DP[i][j][P] = { CHANGE if DP[i − 1][k][E] is chosen
                          { INSERT if dp[i][k][E] is chosen
                          { REMOVE if dp[i − 1][j][E] is chosen
return DP
```

Figure 52 Pseudo Code of the Minimum Edit Distance Algorithm

## Pseudo Code of the Backtracking for Minimum Edit Distance Algorithm

```
Backtracking for Minimum Edit Distance Algorithm
Input:
G ← the input graph
DP ← the dynamic programming table
//DP[][][E] = DP[][][0] edit distance score
// DP[][][A] = DP[][][1]  pointing arrow
// DP[][][P] = DP[][][2] edit process
m ← the size of the searching sequence
Output:
PATH ← A path that contain the matching nodes in the graph
STATE← A state path that contain the edit process in the path
Process:
j ← m
i <-G.size
while i>0 do
    if DP[j][i][P] == SAME then
        PATH.push_back(i-1)
        STATE.push_back(SAME)
        j = DP[i][j][A]
        i = i -1
    else if DP[j][i][P] == INSERT then
        PATH.push_back(i-1)
        STATE.push_back(INSERT)
        j = DP[i][j][A]
    else if DP[j][i][P] == REMOVE then
        PATH.push_back(i-1)
        STATE.push_back(REMOVE)
        j = DP[i][j][A]
        i = i -1
    else if DP[j][i][P] == CHANGE then
```

```
        PATH.push_back(i-1)
        STATE.push_back(CHANGE)
        j = DP[i][j][A]
        i = i -1
return Path
```

Figure 53 Pseudo Code of the Backtracking for Minimum Edit Distance Algorithm

## Import Graph File

In my program, it allows user to import graph file, the file format that my program accept is .gfa file format.

### Graphical Fragment Assembly File format

Graphical Fragment Assembly (as known as .gfa) is a standard file format for genome sequence graph. .gfa file format is a tab-delimited text format where the first field of the line decide the type of the input data, S stand for Segment and L stand for Link. The following are the line structure of the .gfa file.

### Line Structure of .gfa File

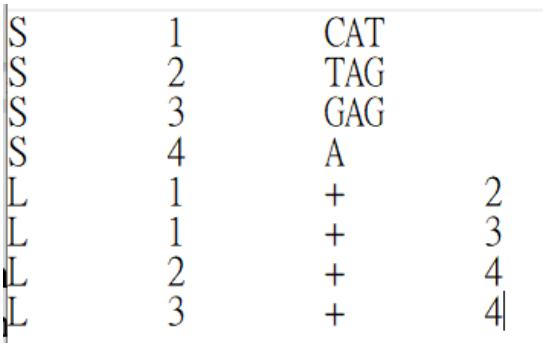



Figure 54 A Sample .gfa File's Content

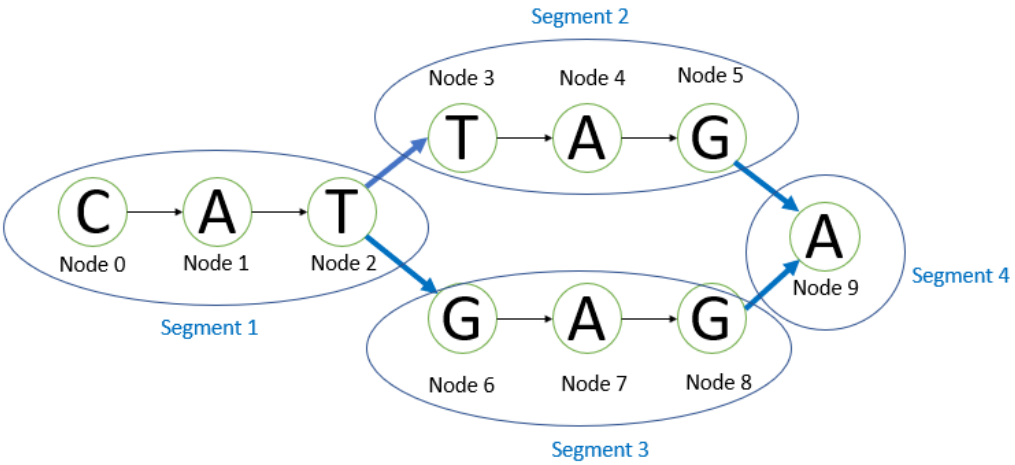



Figure 55 The represent graph for the .gfa file

Segment: first field of the line is denoting by S, the second field defines the segment index, and the third field of the line is the segment. Segment is a continuous sequence of nodes, and there are edges connecting them. The nodes that are not the starting node and the ending node of the segment are all define as middle nodes of the segment, middle nodes in the segment always have both in-degree and out-degree equal to 1, the middle nodes in the segment are not allow to have extra edges point to other nodes or point by other nodes. A single node is also known as a segment, where the starting node and ending node of the segment are also the node itself. In **Figure 55**, segment 4 is a segment with only one node.

Link: first field of the line is denoting by 'L', the second filed defines the link index, and the third field and fifth field are the segments that the link connect to, the ending node of the third field's segment is pointing to the starting node of the fifth field's segment. Link is also a subset of edge, which mean the link that connect the segment together is also known as an edge.

## Class Definition

There are four classes for the algorithms, which is Node, Segment, Path and Graph, Graph is the biggest class which contains a set of node objects and segments objects. When user import the graph file into the program, there will be a Graph object for the imported graph. User can use the algorithms provide by the class (details in the graph class' definition) to retrieve the path by their searching sequence. Path is a class that contain a vector of nodes, path object will be created when user using the searching function, a path object will be

initialize and the algorithm will push the nodes inside the path into the path object. The following are the definition of the four classes.

**Node's Class**

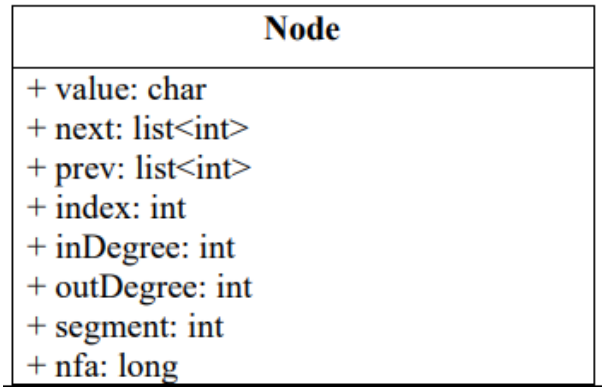



Figure 56 UML Diagram of Node's Class

```
class Node
{
public:
    char value;
    list<int> next;
    list<int> prev;
    int index;
    int inDegree = 0;
    int outDegree = 0;
    int segment;
    long nfa;
};
```

Figure 57 Node Class' Header

The following are the description of the class member of node class.

**Class Variables**

1. **value:** the alphabet value stores inside the node, there are only four possible values for each node, which are 'C','A', 'T' and 'G'.
2. **next:** list of successor nodes of the current node, the default value of the list is empty, which indicate that the node does not have any successor node.
3. **prev:** list of predecessor nodes of the current node, the default value of the list is empty, which indicates that the node does not have any predecessor node.
4. **Index:** the index of the node inside the graph.
5. **inDegree:** number of predecessor nodes of the current node, the default value of inDegree is 0, which mean no predecessor node for the current node by default.
6. **outDegree:** number of successor nodes of the current node, the default value of outDegree is 0, which mean no successor node for the current node by default.
7. **segment:** the segment that the current node is belong to.
8. **nfa:** the NFA state of the current node (exclusive for the extract match algorithm).

### Segment Class

| Segment |
| --- |
| + startNode: int<br>+ endNode: int |

Figure 58 UML Diagram of Segment Class

```
class Segment
{
public:
    int startNode;
    int endNode;
};
```

Figure 59 Header of Segment Class

The following are the description of the class member of segment class.

### Class Variables

1. **startNode:** the index of the starting node of the segment in the graph
2. **endNode:** the index of the ending node of the segment in the graph

### Path's Class

| Path |
| --- |
| + path: vector<int><br>+ state: vector<char><br>+ startSegment: int<br>+ endSegment: int<br>+ startIndex int<br>+ endIndex int<br>+ editDistance int |
| + Path()<br>+ setStartSegment(int seg) void<br>+ setEndSegment(int seg) void<br>+ setStartIndex(int index) void<br>+ setEndIndex(int index) void<br>+ appendNode(int index) void<br>+ appendState(int state) void<br>+ getStartSegment() int<br>+ getEndSegment() int<br>+ getStartIndex() int<br>+ getEndIndex() int<br>+ getPath() vector<int><br>+ getState() vector<char><br>+ ~Path() |

Figure 60 UML Diagram of Path Class

```
class Path
{
public:
    vector<int> path;
    vector<char> state;
    int startSegment;
    int endSegment;
    int startIndex;
    int endIndex;
    int editDistance;

    //Default constructor
    Path();

    //mutator
    void setStartSegment(int segment);
    void setEndSegment(int segment);
    void setStartIndex(int index);
    void setEndIndex(int index);
    void appendNode(int index);
    void appendState(int state);

    //accessor
    int getStartSegment();
    int getEndSegment();
    int getStartIndex();
    int getEndIndex();
    vector<int> getPath();
    vector<char> getState();

    //destructor
    ~Path() { ... }
};
```

Figure 61 Header of Path Class

The following are the description of the class member inside path's class

**Class Variables**

1. **path:** a vector that store the index of the nodes inside the path.
2. **state:** a vector that store the edit state of the nodes inside the path, the possible value of state is 1 for Same, 2 for Change, 3 for Insert and 4 for Remove.
3. **startSegment:** the index of the segment that the path starting in.
4. **endSegment:** the index of the segment that the path ending in.
5. **startIndex:** the index of the node that the path starting in.
6. **endIndex:** the index of the node that the path ending in.
7. **editDistance:** the edit distance of the nodes in the path against the searching sequence. (exclusive for minimum edit distance algorithm)

**Class Functions**

1. **Path():** a default constructor for initializing the path's object, the function pre-sets startSegment, endSegment, startIndex, endIndex into -1, which indicate that the starting and ending position of the path are undefine by default.
2. **setStartSegment():** a mutator function for setting the value of the startSegment.
3. **setEndSegment():** a mutator function for setting the value of the endSegment.
4. **setStartIndex():** a mutator function for setting the value of the startIndex.
5. **setEndIndex():** a mutator function for setting the value of the endIndex.
6. **appendNode():** a function for appending the index of a node into the path.
7. **appendState():** a function for appending the state of the appended node into the state vector of the path.
8. **getStartSegment():** an accessor function that return the value of startSegment.
9. **getEndSegment():** an accessor function that return the value of endSegment.
10. **getStartIndex():** an accessor function that return the value of startIndex.
11. **getEndIndex():** an accessor function that return the value of endIndex.
12. **getPath():** an accessor function that return the path's vector.
13. **getState():** an accessor function that return the path's state vector. (exclusive for the minimum edit distance algorithm)
14. **~Path(): a** destructor function for deleting all the class variable in order to save up memory.

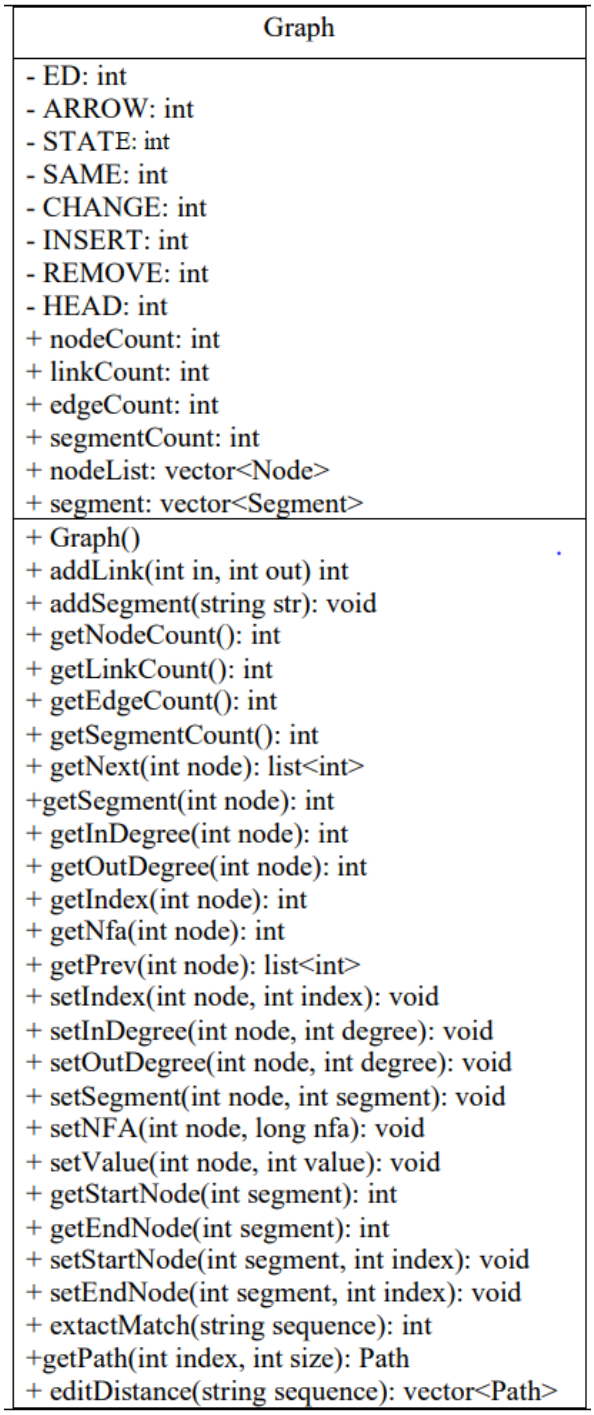



Figure 62 UML Diagram of Graph Class

```
class Graph
{
private:
    int ED = 0;
    int ARROW = 1;
    int STATE = 2;
    int SAME = 1;
    int CHANGE = 2;
    int INSERT = 3;
    int REMOVE = 4;
    int HEAD = 0;

public:
    int nodeCount;
    int linkCount;
    int edgeCount;
    int segmentCount;
    vector<Node> nodeList;
    vector<Segment> segmentList;

    Graph();
    bool addLink(int inSegment, int outSegment);
    void addSegment(string segment);
    int getNodeCount();
    int getLinkCount();
    int getEdgeCount();
    int getsegmentCount();

    //node related getter
    list<int> getNext(int node);
    char getValue(int node);
    int getSegment(int node);
    int getInDegree(int node);
    int getOutDegree(int node);
    int getIndex(int node);
    long getNfa(int node);
    list<int> getPrev(int node);

    //node relate setter
    void setIndex(int node, int index);
    void setIndegree(int node, int degree);
    void setOutdegree(int node, int degree);
    void setSegment(int node, int segment);
    void setNFA(int node, long nfa);
    void setValue(int node, int value);

    //segment relate setter
    int getStartNode(int segment);
    int getEndNode(int segment);
    void setStartNode(int segment, int index);
    void setEndNode(int segment, int index);

    //search function
    int extactMatch(string sequence);
    Path getPath(int index, int size);
    vector<Path> editDistance(string sequence);
    ~Graph() { ... }
};
```

Figure 63 Header of Graph Class

The following are the description of the class member of graph class

1. **ED, ARROW, STATE:** the constant variables for dynamic program to define which layer that the DP array is locate in, ED is define as 0 for the edit distance layer, ARROW is define as 1 for the arrow layer and STATE is define as 2 for the edit state layer.

2. **SAME, CHANGE, INSERT, REMOVE:** a set of constant variables for the edit state layer to define which state that the corresponding node is in, SAME is define as 1 for the match case, CHANGE is define as 2 for not match case and point to the diagonal value of one of its predecessor node, INSERT is define as 3 for not match case and point to the side-neighbour of one of its predecessor node and REMOVE is define as 3 for not match case and point to the up-neighbour.
3. HEAD: a constant variable set as 0, which indicate that the node is one of the starting points of the graph that do not have any predecessor node. a node point to HEAD mean that it does not point to any neighbour.
4. **nodeCount:** a variable that store the number of nodes inside the graph.
5. **linkCount:** a variable that store the number of links inside the graph.
6. **edgeCount:** a variable that store the number of edges inside the graph
7. **segmentCount:** a variable that store the number of segments inside the graph
8. **nodeList:** a variable that store the vector of nodes object inside the graph
9. **segment:** a variable that store the vector of segments object inside in the graph
10. Class member functions
11. **Graph():** the default constructor for initializing the Graph class object, the function will initialize the value of nodeCount, linkCount, edgeCount and segmentCount into 0, which mean the graph do not have any nodes and edges at the beginning.
12. **addLink():** a function that add the link into the graph.
13. **addSegment():** a function that add the segment into the graph.
14. **getNodeCount():** an accessor function that return the number of nodes in the graph.
15. **getLinkCount():** an accessor function that return the number of links in the graph.
16. **getEdgeCount():** an accessor function that return the number of edges in the graph.
17. **getSegmentCount():** an accessor function that return the number of edges in the graph.
18. **getNext():** a function that return the list that contain the successor nodes of the given node.
19. **getSegment():** a function that return the segment of the given node.
20. **getInDegree():** a function that return the in-degree of the given node.
21. **getOutDegree():** a function that return the out-degree of the given node.
22. **getIndex():** a function that return the index of the given node.

23. **getNfa():** a function that return the NFA state of the given node. (for extract match algorithm only)
24. **getPrev():** a function that return the list of predecessor nodes of the given node.
25. **setIndex():** a function that set the index of the given node.
26. **setInDegree():** a function that set the in-degree of the given node.
27. **setOutDegree():** a function that set the out-degree of the given node.
28. **setSegment():** a function that set the segment of the given node.
29. **setNFA():** a function that set the NFA state of the given node. (for extract match algorithm only)
30. **setValue():** a function that set the value of the given node.
31. **getStartNode():** a function that return the starting node of the given segment.
32. **getEndNode():** a function that return the ending node of the given segment.
33. **setStartNode():** a function that set the starting node of the given segment.
34. **setEndNode():** a function that set the ending node of the given segment.
35. **exactMatch():** a function for finding the path that have an exact match with the given searching sequence inside the graph. The function will return the index of the last node in the path and return -1 for no path is found.
36. **exactMatchPath():** a function that return the path object that contain a set of nodes that matching with the searching sequence. The function takes the index and the length of the searching sequence as the input and having a back-tracking process to get the nodes inside the path.
37. **editDistance():** a function for finding the path that have the least edit distance with the given searching sequence inside the graph. The function will return a set of Path object which have the top five least edit distance path inside the graph.

## Software program

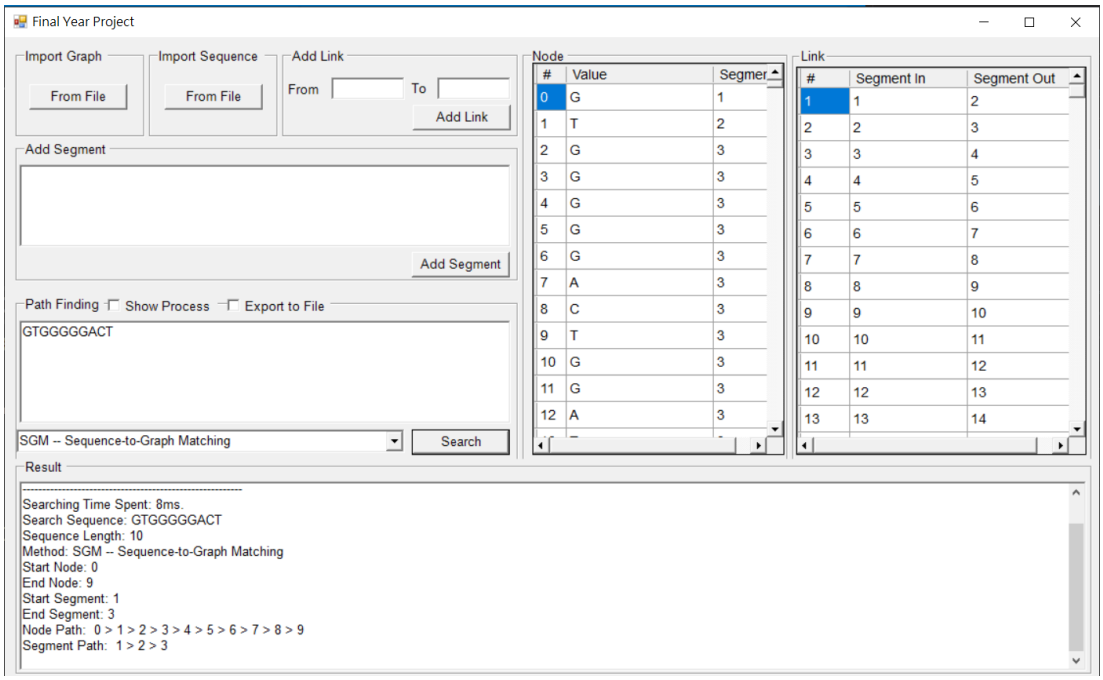



Figure 64 Screenshot of the Software Program

In addition to the algorithm development, I also developed software program for the algorithm, the software program provides a graphic user interface by using the C++/CLI library, which is a library create by Microsoft, it is a management extension for C++.


Figure 65 Logo of C++/CLI Library

The software program provides the following functions:

**Import Graph from file:** allow user to import graph file (.gfa).

**Import Sequence from file:** allow user to import sequence file (.txt).

**Add link:** allow user to add extra link into the graph.

**Add segment:** allow user to add extra segment into the graph.

**View Nodes and Links:** allow user to view the nodes and links in the graph through the table.

**Path Finding:** allows user to input the sequence and select the method for searching.

**View Result:** allows user to view the searching result through the result textbox.

# Results

In the project, I have developed two algorithms, the Exact Match Algorithm solved the Sequence-to-Graph Matching problem and the Minimum Edit Distance Algorithm solved the Sequence-to-Graph Alignment problem. The following are the detail of the result of the two algorithms respectively.

## Exact Matching algorithm

The Exact Match Algorithm successfully solved the Sequence-to-Graph Matching problem, the following are the description of the time complexity, space complexity and the searching time of the algorithm.

### Time complexity

The time complexity of the algorithm is O(|M|*|E|), where M is the length of the searching sequence and E is the number of edges in the graph. The reason for O(|M|*|E|) time complexity is, the more the length of the searching sequence, the more compare operation for the process of each node. Besides, the more the edges in the graph, the more possible to have a node with in-degree more than 1,  and for one extra predecessor nodes, the algorithm need to take an extra AND operation, and if the number of edges in the graph + 1, there will be a node in the graph has in-degree + 1. Therefore, the time complexity of the algorithm must be O( |M|*|E|).

### Searching Time

In the searching time testing, I tried to experiment the algorithm with different length of searching sequence and the graph with different number of nodes. In addition, the specification of the device for testing is a computer with 32 Gigabytes random access memories.

**Searching Time for the searching sequence's length is 5**

| Number of edges in the graph | Searching Time Spent |
|---|---|
| 10 | 0 ms |
| 100 | 0 ms |
| 1000 | 0 ms |
| 2000 | 1 ms |
| 5000 | 1 ms |
| 10000 | 2 ms |
| 20000 | 3 ms |

Table 2 Table of Searching Time Spent for the Searching Sequence with Length is 5

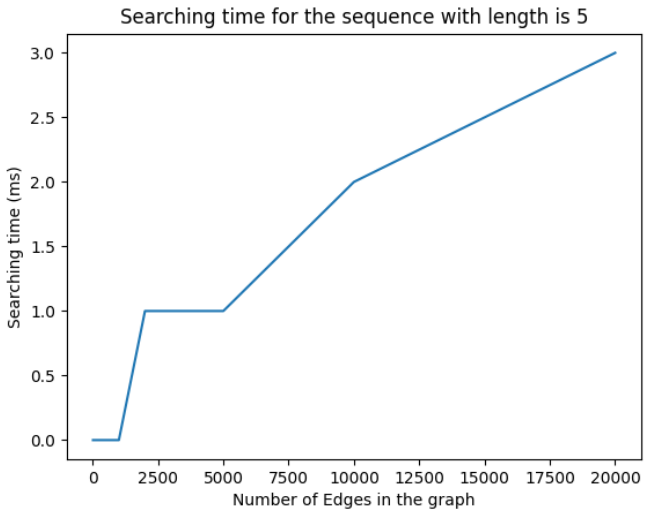



Figure 66 Chart of Searching Time Spends for the Searching Sequence with Length is 5

In **Table 2**, it shows the time spends for the searching sequence with length is 5, and in **Figure 66**, it shows that the searching time spent against the number of edges in the graph increase in linear even though the searching time spent when the number of edges in the graph is smaller than 5000 is not linear, the possible reason is the initialization time for the algorithm is fixed, if the number of edges in the graph is small, they do not affect the total time spent too much compare with the time spend on initialization.

**Searching Time for the searching sequence's length is 10**

| Number of edges in the graph | Searching Time Spent |
|---|---|
| 10 | 0 ms |
| 100 | 0 ms |
| 1000 | 1 ms |
| 2000 | 1 ms |
| 5000 | 1 ms |
| 10000 | 2 ms |
| 20000 | 4 ms |

Table 3 Table of Searching Time Spent for the Searching Sequence with Length is 10

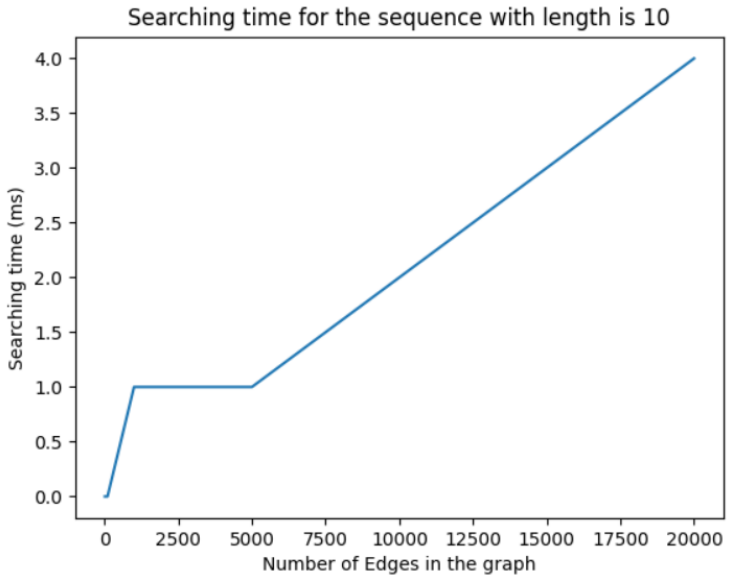



Figure 67 Chart of Searching Time Spends for the Searching Sequence with Length is 10

In **Table 3**, it shows the time spends for the searching sequence with length is 10, and in **Figure 67**, it shows that the searching time spent against the number of edges in the graph increase in linear.

**Searching Time for the searching sequence's length is 20**

| Number of edges in the graph | Searching Time Spent |
|---|---|
| 10 | 1 ms |
| 100 | 1 ms |
| 1000 | 1 ms |
| 2000 | 2 ms |
| 5000 | 2 ms |
| 10000 | 3 ms |
| 20000 | 5 ms |

Table 4 Table of Searching Time Spent for the Searching Sequence with Length is 20

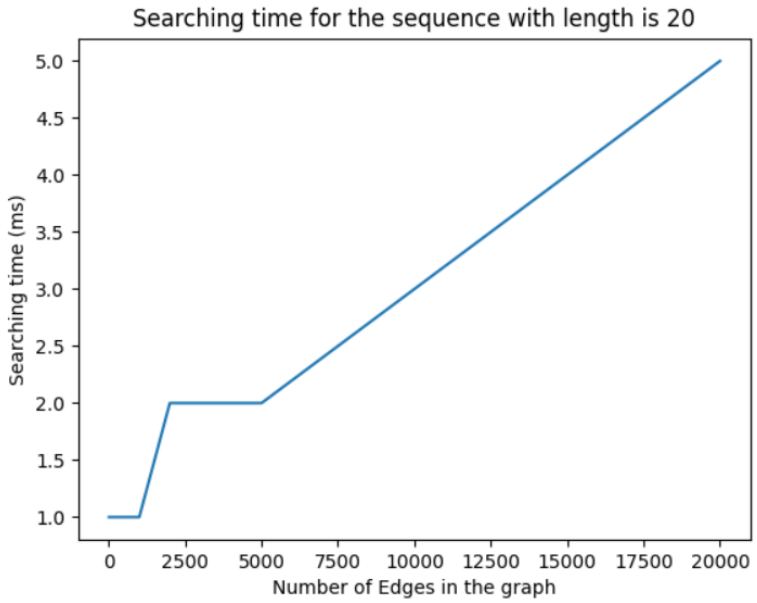



Figure 68 Chart of Searching Time Spends for the Searching Sequence with Length is 20

In **Table 4**, it shows the time spends for the searching sequence with length is 20, and in **Figure 68**, it shows that the searching time spent against the number of edges in the graph increase in linear.

**Searching Time for the searching sequence's length is 30**

| Number of edges in the graph | Searching Time Spent |
|---|---|
| 10 | 1 ms |
| 100 | 1 ms |
| 1000 | 1 ms |
| 2000 | 2 ms |
| 5000 | 3 ms |
| 10000 | 3 ms |
| 20000 | 6 ms |

Table 5 Table of Searching Time Spent for the Searching Sequence with Length is 30

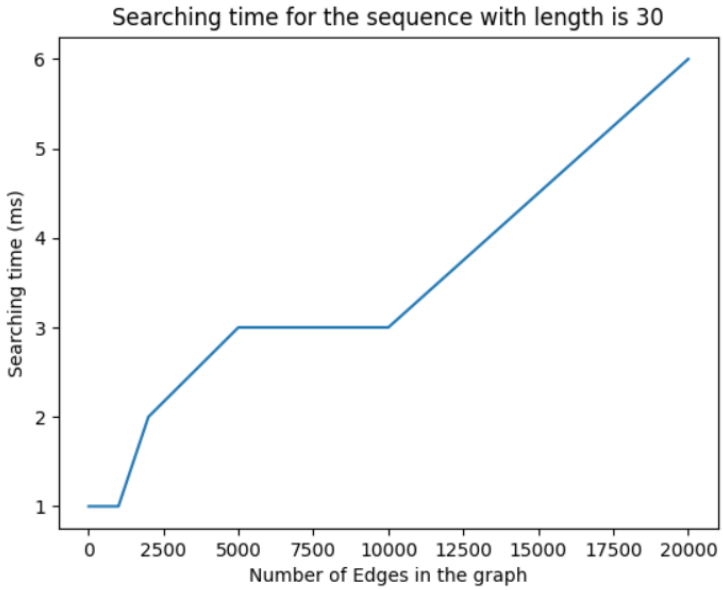



Figure 69 Chart of Searching Time Spends for the Searching Sequence with Length is 30

In **Table 5**, it shows the time spends for the searching sequence with length is 30, and in **Figure 69**, it shows that the searching time spent against the number of edges in the graph increase in linear.

## Minimum Edit Distance Algorithm

The Minimum Edit Distance Algorithm successfully solved the Sequence-to-Graph Alignment problem, the following are the description of the time complexity, space complexity and the searching time of the algorithm.

### Time complexity

The time complexity of the algorithm is O((|V| + |M|)*|E|), where V is the number of nodes in the graph, where M is the length of the searching sequence and E is the number of edges in the graph. The reason for O((|V| + |M|)*|E|) time complexity is, the more the length of the searching sequence, the more rows in the dynamic programming table process. Besides, the more the edges in the graph, the more possible to have a node with in-degree more than 1, and for one extra predecessor nodes, the algorithm need to take two extra comparation to get the neighbour with minimum edit distance, and if the number of edges in the graph + 1, there will be a node in the graph has in-degree + 1. And the more the nodes in the graph, the more columns in the dynamic programming table process, we need to have a extra matching condition for every rows. Therefore, the time complexity of the algorithm must be O((|V| + |M|)*|E|).

### Searching Time

In the searching time testing, I tried to experiment the algorithm with different length of searching sequence and the graph with different number of nodes. In addition, the specification of the device for testing is a computer with 32 Gigabytes random access memories.

**Searching Time for the searching sequence's length is 5**

| Number of edges in the graph | Searching Time Spent |
|---|---|
| 10 | 19 ms |
| 100 | 23 ms |
| 1000 | 53 ms |
| 2000 | 88 ms |
| 5000 | 169 ms |
| 10000 | 319 ms |
| 20000 | 654 ms |

Table 6 Table of Searching Time Spent for the Searching Sequence with Length is 5

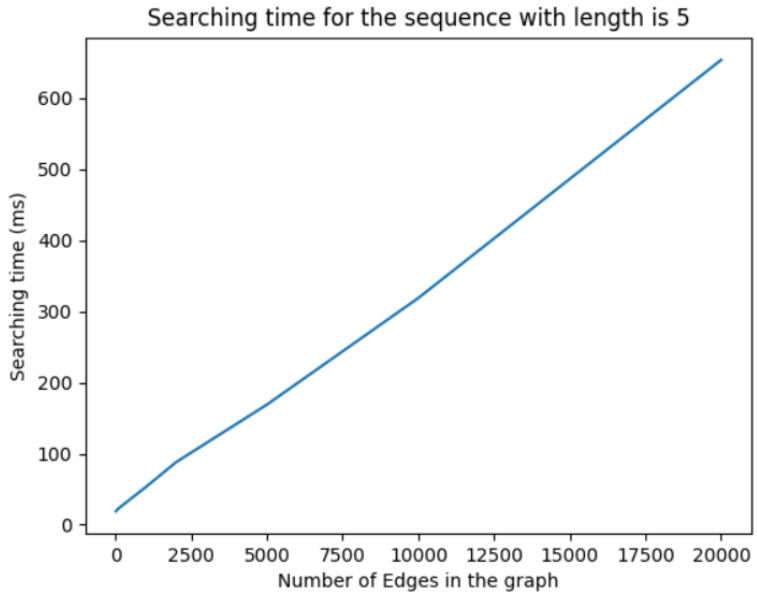



Figure 70 Chart of Searching Time Spends for the Searching Sequence with Length is 5

In **Table 6**, it shows the time spends for the searching sequence with length is 5, and in **Figure 70**, it shows that the searching time spent against the number of edges in the graph increase in linear.

**Searching Time for the searching sequence's length is 10**

| Number of edges in the graph | Searching Time Spent |
|---|---|
| 10 | 21 ms |
| 100 | 25 ms |
| 1000 | 80 ms |
| 2000 | 137 ms |
| 5000 | 290 ms |
| 10000 | 653 ms |
| 20000 | 1145 ms |

Table 7 Table of Searching Time Spent for the Searching Sequence with Length is 10

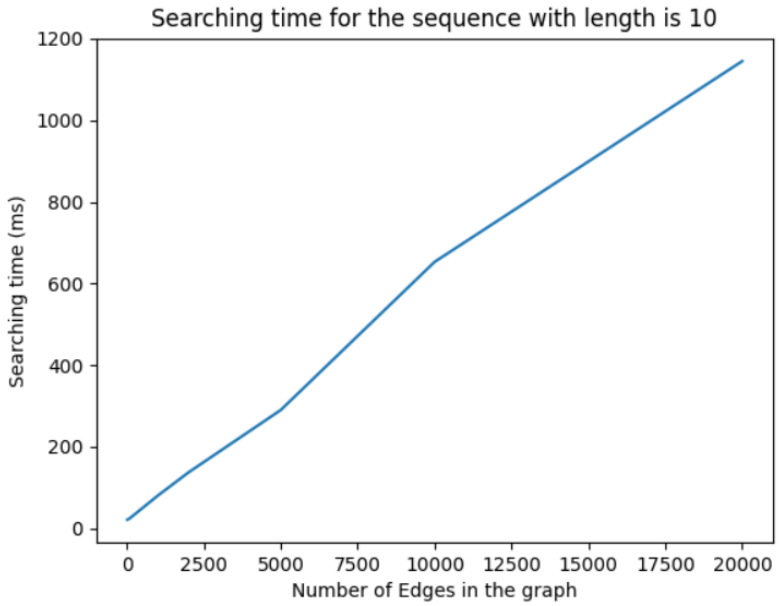



Figure 71 Chart of Searching Time Spends for the Searching Sequence with Length is 10

In **Table 7**, it shows the time spends for the searching sequence with length is 10, and in **Figure 71**, it shows that the searching time spent against the number of edges in the graph increase in linear.

**Searching Time for the searching sequence's length is 20**

| Number of edges in the graph | Searching Time Spent |
|---|---|
| 10 | 21 ms |
| 100 | 30 ms |
| 1000 | 126 ms |
| 2000 | 220 ms |
| 5000 | 520 ms |
| 10000 | 1013 ms |
| 20000 | 2069 ms |

Table 8 Table of Searching Time Spent for the Searching Sequence with Length is 20

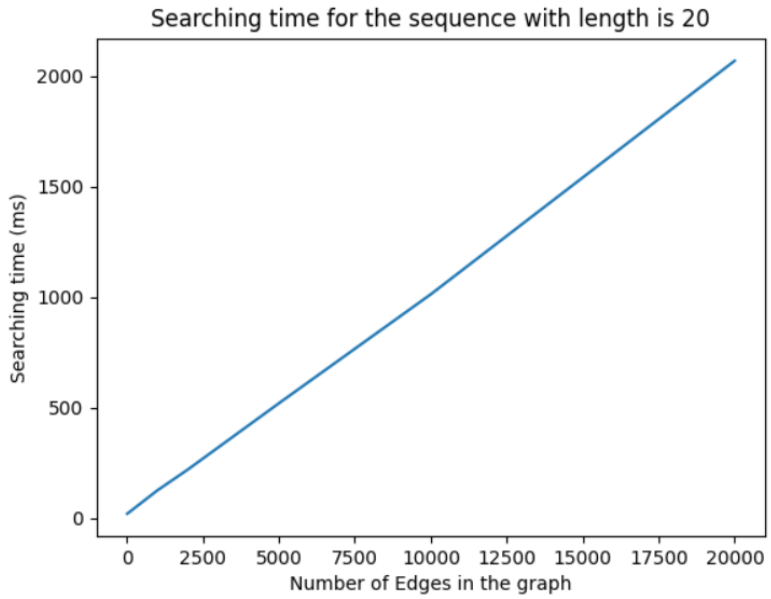



Figure 72 Chart of Searching Time Spends for the Searching Sequence with Length is 20

In **Table 8**, it shows the time spends for the searching sequence with length is 10, and in **Figure 72**, it shows that the searching time spent against the number of edges in the graph increase in linear.

**Searching Time for the searching sequence's length is 30**

| Number of edges in the graph | Searching Time Spent |
|---|---|
| 10 | 22 ms |
| 100 | 35 ms |
| 1000 | 171 ms |
| 2000 | 320 ms |
| 5000 | 760 ms |
| 10000 | 1502 ms |
| 20000 | 3059 ms |

Table 9 Table of Searching Time Spent for the Searching Sequence with Length is 30

**Table 9. Table of searching time spends for the searching sequence length is 30**

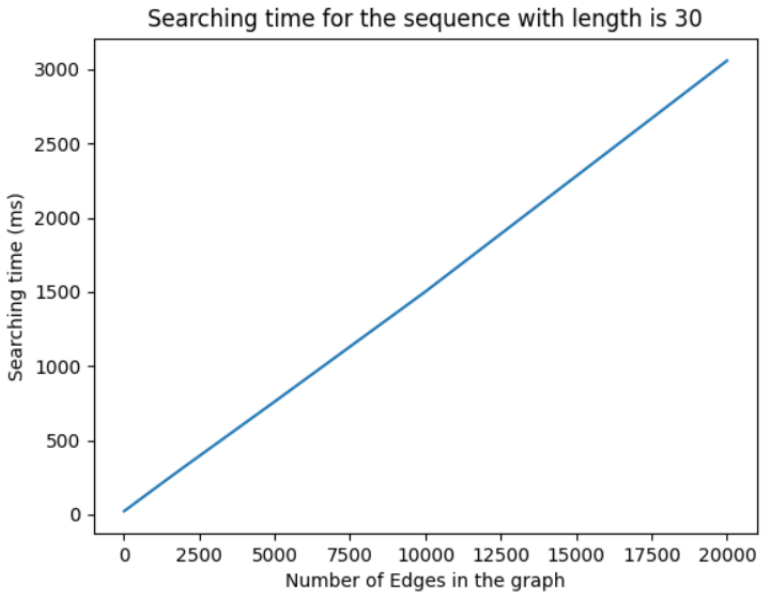



Figure 73 Chart of Searching Time Spends for the Searching Sequence with Length is 30

In **Table 9.**, it shows the time spends for the searching sequence with length is 10, and in **Figure 73**, it shows that the searching time spent against the number of edges in the graph increase in linear.

By compare the searching time spend on the two algorithm, it is obvious that the Edit Distance Algorithm consumes more time than the Exact Match Algorithm, especially for the graph with a lot of edges, it is because in the Exact Match Algorithm, it only need to calculate the NFA states for each node, and in the Minimum Edit Distance Algorithm, it needs to compare the node value with the sequence and also find the minimum edit distance from the predecessor nodes of each node. Therefore, if the user want to find a path that exact match the searching sequence, it is better for he/she to use the Exact Match Algorithm rather than the Minimum Edit Distance Algorithm even though the Minimum Edit Distance Algorithm can also solve his/her problem.

## Software program

For the software program, there are serval feature function that I finally developed, the following are the description of the feature function.

### Display the Searching Result

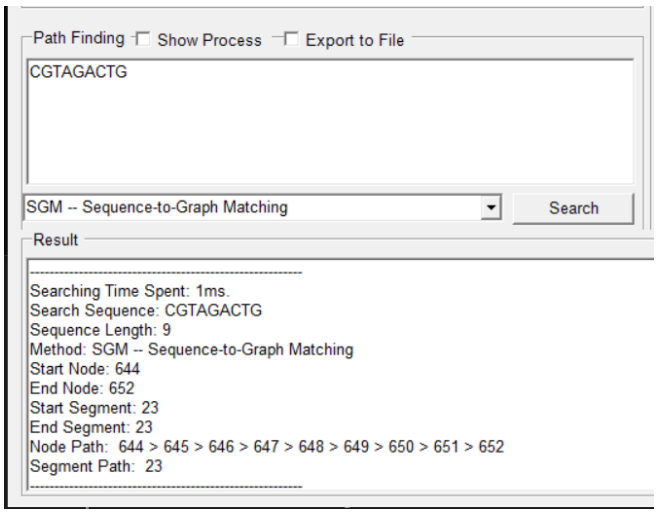



Figure 74 Screenshot of the Software Program Displaying the Searching Result

After the user typed in the sequence that they want to search in the graph and selected the searching method, the searching result will be display in the result textbox. The result provides a lot of information about the searching. The content of the searching result including the searching time spent in searching (in millisecond), the searching sequence, the length of the searching sequence, the searching method that they choose, the starting node

and ending node of the path, the starting segment and end segment of the path, the nodes in the path and the segments involve in the path.

**Display the Top 5 paths**

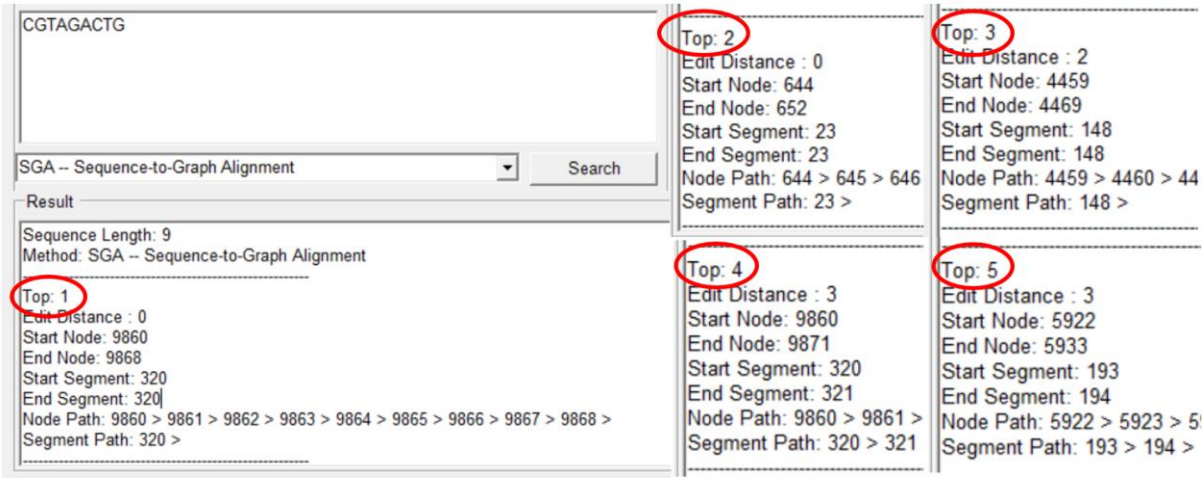



Figure 75 Screenshot of Display the Top 5 Paths with Least Edit Distance

If the user chooses the Minimum Edit Distance Algorithm for their path finding algorithm, the result will display the top 5 paths with the least edit distance.

**Shows Edit Process**

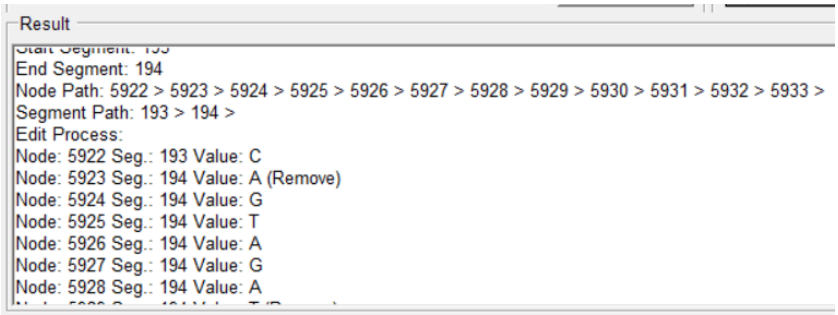



Figure 76 Screenshot of Display the Edit Process in Detail

If the user chooses the Minimum Edit Distance Algorithm for their path finding algorithm, they are optional to select the “Show Process” checkbox. If they select the checkbox, the searching result will display the edit process in detail, including which nodes are keeping, changing, removing and inserting to the sequence.

## Export the searching result

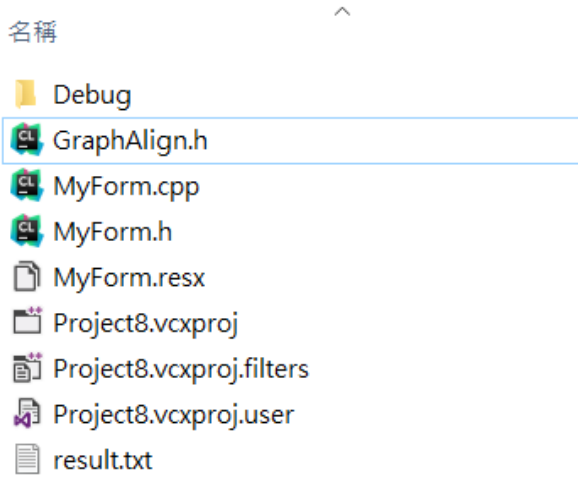



Figure 77 Store Location of the Result Text File

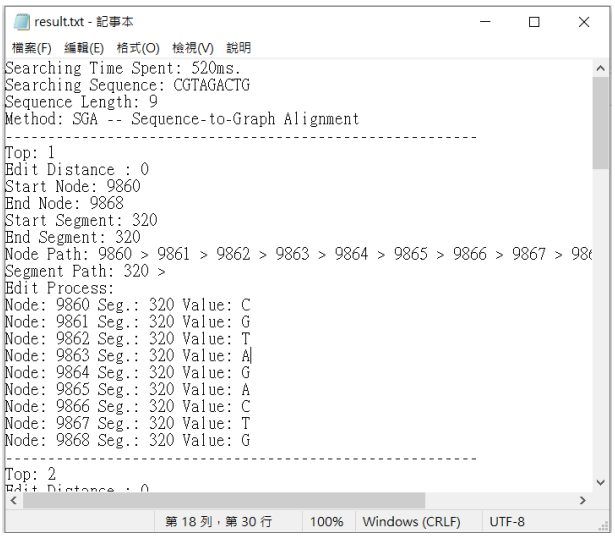



Figure 78 Screenshot of the Result Text File

User can select the “Export to File” checkbox optionally, once they select this checkbox, the result will be export as a text file automatically to the program file (as shown in **Figure 77**) in the result text file, it displays all the searching information which is identical to the information show in the result textbox.

# Discussion:

## Reason for using vector for graph

In the algorithms, I used vector as the data type to store the nodes inside the graph, it is because the time complexity for accessing the data in the vector is O(1), which access by using var[n], where n is the index of the target data in the vector. Besides, the time complexity of insertion and deletion are also O(1) which implement by using push_back(value) and erase(index) respectively. The following are the table of the comparation for different possible data types for the graph in different aspect.

| **Operation** | **Vector** | **Link list** | **Array** |
|---|---|---|---|
| Data access for a specific index k | O(1), by using var[k]. | O(n), where n is the number of elements in the link list, it needs to loop from the head of link list for k times to reach the target data. | O(1), by using var[k]. |
| Insert a node k | O(1), by using push_back(k) for inserting the value in the last of the vector, and<br>O(1) by using insert(index, value) for inserting in a specify index. | O(1), by insert the node in the start of the link list.<br><br>O(n), for inserting in a specific index. | O(n), the size of the array is fixed once it is initialized, any insertion needs to create a new array with the new size and copy all elements from the old array. |
| Delete a node k | O(1), by using erase(index) | O(n), it needs to loop from the head of the linklist for k times to reach the target data position to delete the data. | O(n), the size of the array is fixed once it is initialized, any deletion needs to create a new array with the new size and |

| | | | |
|---|---|---|---|
| | | | copy all elements from the old array. |
| Size | Big, many headers are needed. | Medium, only pointers to other nodes are needed. | Small |

Table 10 Table of the Comparation of Different Possible Data Structure for the Graph

As a result, vector has the best performance in time complexity even it has a worst performance in space complexity. I choose vector by taking account into the efficiency of the algorithm.

## Reason for using list for the predecessor and successor nodes

In the algorithms, I used list as the data type to store the index of the predecessor and successor nodes of each node, it is because the time complexity for inserting for list is O(1), Although the data access time complexity of list is O(n), this is not a big problem because when we need to access the predecessor or successor nodes of a node, it always need to access all of them, for example when we are having dynamic programming process, if a node have in-degree more than one, we need to compare all its predecessor nodes, there does not have any chance that the algorithm only need to access one of the node in the list.

## Defect of the Edit Distance Algorithms

In the minimum edit distance algorithm, I provided the top 5 path with the least edit distance, the method for me to retrieve other possible path is make use of the last rows in the dynamic programming table, and treat every elements in the last row are the start point of its induvial path. However, this approach is not perfect enough because in my approach, the other path is retrieving by taking out the last node in the graph and produce another path. This approach ignored all the possible combination of path in the graph. In the future improvement of the algorithm, the dynamic programming table should be modified in order to consider all possible paths in the graph.

## Reason for not supporting cyclic graph

In the algorithms, I need to take account into the time complexity. As a result, I give up the cyclic graph. It is because cyclic graph involves looping of nodes, an algorithm that support cyclic graph cannot provide a fast time complexity. In the further improvement of the algorithm, I think it should be able to support cyclic graph.

## Space Complexity of the algorithms

Space complexity is also a problem of my algorithm, because I use vector to be the data type for graph to store the nodes inside it because of it fast time complexity. However, the drawback of vector is its heavy header. Besides, the dynamic programming require to create a 2d array with the size of the nodes in the graph and the length of the searching sequence, if the graph and the searching sequence are too large, the size of the table will be also large. In the future improvement of the algorithm, I think it can consider other data type rather than vector to save up the memory.

# Conclusion

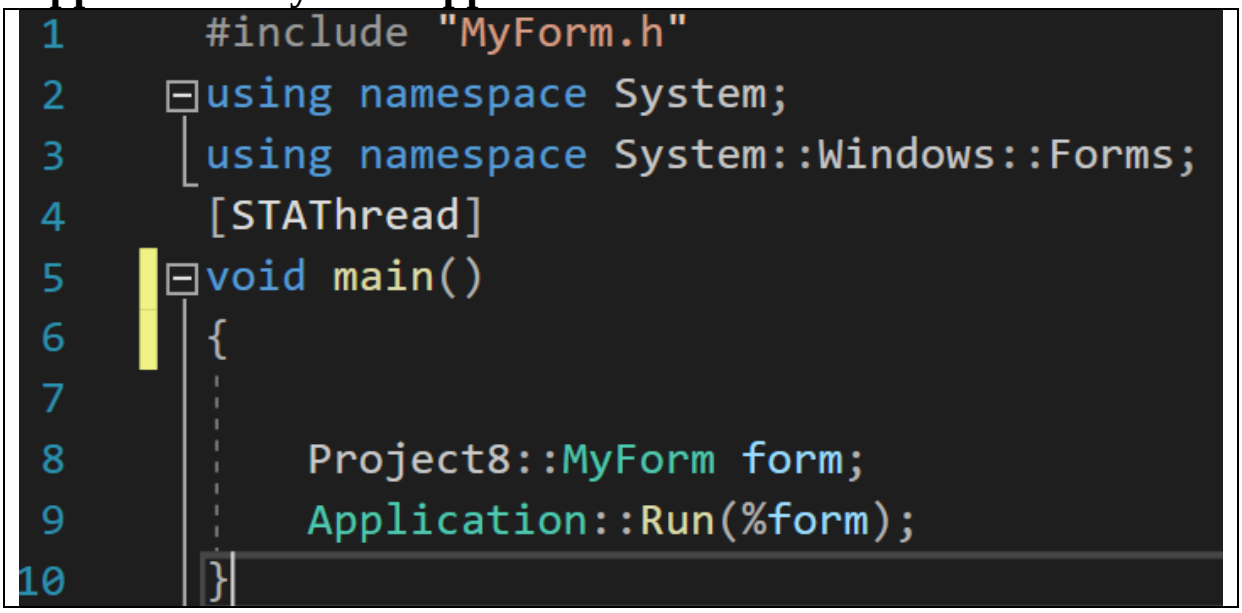
The project aims to develop an algorithm for finding a path in a very large graph by taking account into the time complexity. An efficient path finding algorithm is important for genetic engineers to study on different kinds of genome sequence and reduce the time needed to compare on the genome sequence.

In the start of the project, it required me to take a lot of research on compare different kinds of algorithms for graph searching. And many algorithms are very complicate and difficult to understand. Some of them can solve the problem correctly but with a unacceptable time complexity. For example, breadth first search and depth first search can solve the problem for listing all possible paths in the graph. However, those algorithms required me to compare every path independently, which are time consuming.

The most difficult part of the project is the implementation of the algorithm, especially for the minimum edit distance algorithm, it is easy to make mistake and create a wrong result. Also, the mechanism of dynamic programming is very tricky, which is much different from the approach that I have learnt such as divide and conquer, recursive function and greedy approach. Even though the theory of dynamic programming is hard to understand and the rules of the algorithm is cumbersome, I finally overcome it and implemented the algorithm.

Finally, I successfully developed the algorithms for finding a path in the graph that exactly match the searching sequence or has a minimum edit distance with the searching sequence. And the algorithms can run in a considerable time complexity. I also developed a software program for user to have search on their own graphs and sequences. Besides, it allows export the result of searching into text file for further usage.

## Appendix I: MyForm.cpp



```
#include "MyForm.h"
using namespace System;
using namespace System::Windows::Forms;
[STAThread]
void main()
{

    Project8::MyForm form;
    Application::Run(%form);
}
```

# Appendix II: MyForm.h

```
#pragma once
#include "GraphAlign.h"
#include <msclr\marshal_cppstd.h>
#include <string>
#include <queue>
#include <stdio.h>
#include <time.h>
#include <fstream>
#include <stack>
int ED = 0;
int ARROW = 1;
int STATE = 2;
int SAME = 1;
int CHANGE = 2;
int INSERT = 3;
int REMOVE = 4;
int HEAD = -1;
namespace Project8
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    Graph myGraph;

    public ref class MyForm : public System::Windows::Forms::Form
    {
    public:
        msclr::interop::marshal_context context;
        MyForm(void)
        {
            InitializeComponent();
        }

    protected:
        ~MyForm()
        {
            if (components)
            {
                delete components;
            }
        }

    private: System::Windows::Forms::RichTextBox^ resultTextbox;
    private: System::Windows::Forms::Button^ addSegmentButton;
    private: System::Windows::Forms::Button^ addLinkButton;
    private: System::Windows::Forms::Button^ searchButton;
```

```
50      private: System::Windows::Forms::Button^ searchButton;
51      private: System::Windows::Forms::Button^ importGraphButton;
52      private: System::Windows::Forms::DataGridView^ nodeTable;
53      private: System::Windows::Forms::DataGridView^ linkTable;
54      private: System::Windows::Forms::DataGridViewTextBoxColumn^ nodeNumColumn;
55      private: System::Windows::Forms::DataGridViewTextBoxColumn^ nodeValueColumn;
56      private: System::Windows::Forms::DataGridViewTextBoxColumn^ nodeSegmentColumn;
57      private: System::Windows::Forms::DataGridViewTextBoxColumn^ linkNumColumn;
58      private: System::Windows::Forms::DataGridViewTextBoxColumn^ linkSegmentInColumn;
59      private: System::Windows::Forms::DataGridViewTextBoxColumn^ linkSegmentOutColumn;
60      private: System::Windows::Forms::GroupBox^ nodeGroupbox;
61      private: System::Windows::Forms::GroupBox^ linkGroupbox;
62      private: System::Windows::Forms::GroupBox^ pathFindingGroupbox;
63      private: System::Windows::Forms::GroupBox^ importGraphGroupbox;
64      private: System::Windows::Forms::GroupBox^ addSegmentGroupbox;
65      private: System::Windows::Forms::GroupBox^ importSequenceGroupbox;
66      private: System::Windows::Forms::Button^ importSequenceButton;
67      private: System::Windows::Forms::GroupBox^ addLinkGroupbox;
68      private: System::Windows::Forms::Label^ toLabel;
69      private: System::Windows::Forms::Label^ fromLebel;
70      private: System::Windows::Forms::GroupBox^ resultGroupbox;
71      private: System::Windows::Forms::TextBox^ toTextbox;
72      private: System::Windows::Forms::TextBox^ fromTextbox;
73      private: System::Windows::Forms::RichTextBox^ addSegmentTextbox;
74      private: System::Windows::Forms::RichTextBox^ searchSegmentTextbox;
75      private: System::Windows::Forms::ComboBox^ searchMethodCombobox;
76      private: System::Windows::Forms::CheckBox^ checkBox1;
77      private: System::Windows::Forms::CheckBox^ checkBox2;
78      private: System::Windows::Forms::Button^ button1;
79
80      private:
81          System::ComponentModel::Container ^components;
82
83  #pragma region Windows Form Designer generated code
84          void InitializeComponent(void)
85          {
86              this->resultTextbox = (gcnew System::Windows::Forms::RichTextBox());
87              this->addSegmentButton = (gcnew System::Windows::Forms::Button());
88              this->addLinkButton = (gcnew System::Windows::Forms::Button());
89              this->searchButton = (gcnew System::Windows::Forms::Button());
90              this->importGraphButton = (gcnew System::Windows::Forms::Button());
91              this->nodeTable = (gcnew System::Windows::Forms::DataGridView());
92              this->nodeNumColumn = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());
93              this->nodeValueColumn = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());
94              this->nodeSegmentColumn = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());
95              this->linkTable = (gcnew System::Windows::Forms::DataGridView());
96              this->linkNumColumn = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());
97              this->linkSegmentInColumn = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());
98              this->linkSegmentOutColumn = (gcnew System::Windows::Forms::DataGridViewTextBoxColumn());
99              this->nodeGroupbox = (gcnew System::Windows::Forms::GroupBox());
100             this->linkGroupbox = (gcnew System::Windows::Forms::GroupBox());
```

```
this->pathFindingGroupbox = (gcnew System::Windows::Forms::GroupBox());
this->checkBox2 = (gcnew System::Windows::Forms::CheckBox());
this->checkBox1 = (gcnew System::Windows::Forms::CheckBox());
this->searchMethodCombobox = (gcnew System::Windows::Forms::ComboBox());
this->searchSegmentTextbox = (gcnew System::Windows::Forms::RichTextBox());
this->importGraphGroupbox = (gcnew System::Windows::Forms::GroupBox());
this->addSegmentGroupbox = (gcnew System::Windows::Forms::GroupBox());
this->addSegmentTextbox = (gcnew System::Windows::Forms::RichTextBox());
this->importSequenceGroupbox = (gcnew System::Windows::Forms::GroupBox());
this->importSequenceButton = (gcnew System::Windows::Forms::Button());
this->addLinkGroupbox = (gcnew System::Windows::Forms::GroupBox());
this->toTextbox = (gcnew System::Windows::Forms::TextBox());
this->fromTextbox = (gcnew System::Windows::Forms::TextBox());
this->toLabel = (gcnew System::Windows::Forms::Label());
this->fromLebel = (gcnew System::Windows::Forms::Label());
this->resultGroupbox = (gcnew System::Windows::Forms::GroupBox());
this->button1 = (gcnew System::Windows::Forms::Button());
(cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this->nodeTable))->BeginInit();
(cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this->linkTable))->BeginInit();
this->nodeGroupbox->SuspendLayout();
this->linkGroupbox->SuspendLayout();
this->pathFindingGroupbox->SuspendLayout();
this->importGraphGroupbox->SuspendLayout();
this->addSegmentGroupbox->SuspendLayout();
this->importSequenceGroupbox->SuspendLayout();
this->addLinkGroupbox->SuspendLayout();
this->resultGroupbox->SuspendLayout();
this->SuspendLayout();
//
// resultTextbox
//
this->resultTextbox->Location = System::Drawing::Point(5, 29);
this->resultTextbox->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
this->resultTextbox->Name = L"resultTextbox";
this->resultTextbox->Size = System::Drawing::Size(1449, 240);
this->resultTextbox->TabIndex = 1;
this->resultTextbox->Text = L"";
//
// addSegmentButton
//
this->addSegmentButton->Font = (gcnew System::Drawing::Font(L"Arial", 10, System::Drawing::FontStyle::Regular,
    System::Drawing::GraphicsUnit::Point, static_cast<System::Byte>(0)));
this->addSegmentButton->Location = System::Drawing::Point(539, 141);
this->addSegmentButton->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
this->addSegmentButton->Name = L"addSegmentButton";
this->addSegmentButton->Size = System::Drawing::Size(134, 32);
this->addSegmentButton->TabIndex = 4;
this->addSegmentButton->Text = L"Add Segment";
this->addSegmentButton->UseVisualStyleBackColor = true;
this->addSegmentButton->Click += gcnew System::EventHandler(this, &MyForm::addSegmentButton_Click);
```

```
151             //
152             // addLinkButton
153             //
154             this->addLinkButton->Location = System::Drawing::Point(177, 72);
155             this->addLinkButton->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
156             this->addLinkButton->Name = L"addLinkButton";
157             this->addLinkButton->Size = System::Drawing::Size(133, 32);
158             this->addLinkButton->TabIndex = 8;
159             this->addLinkButton->Text = L"Add Link";
160             this->addLinkButton->UseVisualStyleBackColor = true;
161             this->addLinkButton->Click += gcnew System::EventHandler(this, &MyForm::addLinkButton_Click);
162             //
163             // searchButton
164             //
165             this->searchButton->Location = System::Drawing::Point(539, 170);
166             this->searchButton->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
167             this->searchButton->Name = L"searchButton";
168             this->searchButton->Size = System::Drawing::Size(134, 32);
169             this->searchButton->TabIndex = 9;
170             this->searchButton->Text = L"Search";
171             this->searchButton->UseVisualStyleBackColor = true;
172             this->searchButton->Click += gcnew System::EventHandler(this, &MyForm::searchButton_Click);
173             //
174             // importGraphButton
175             //
176             this->importGraphButton->Font = (gcnew System::Drawing::Font(L"Arial", 10));
177             this->importGraphButton->Location = System::Drawing::Point(19, 46);
178             this->importGraphButton->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
179             this->importGraphButton->Name = L"importGraphButton";
180             this->importGraphButton->Size = System::Drawing::Size(133, 32);
181             this->importGraphButton->TabIndex = 11;
182             this->importGraphButton->Text = L"From File";
183             this->importGraphButton->UseVisualStyleBackColor = true;
184             this->importGraphButton->Click += gcnew System::EventHandler(this, &MyForm::importGraphButton_Click);
185             //
186             // nodeTable
187             //
188             this->nodeTable->ColumnHeadersHeightSizeMode = System::Windows::Forms::DataGridViewColumnHeadersHeightSizeMode::AutoSize;
189             this->nodeTable->Columns->AddRange(gcnew cli::array< System::Windows::Forms::DataGridViewColumn^  >(3) {
190                 this->nodeNumColumn,
191                     this->nodeValueColumn, this->nodeSegmentColumn
192             });
193             this->nodeTable->Font = (gcnew System::Drawing::Font(L"Arial", 10));
194             this->nodeTable->Location = System::Drawing::Point(13, 19);
195             this->nodeTable->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
196             this->nodeTable->Name = L"nodeTable";
197             this->nodeTable->RowHeadersWidth = 4;
198             this->nodeTable->RowTemplate->Height = 27;
199             this->nodeTable->Size = System::Drawing::Size(343, 500);
200             this->nodeTable->TabIndex = 12;
```

```
            //
            // nodeNumColumn
            //
            this->nodeNumColumn->HeaderText = L"#";
            this->nodeNumColumn->MinimumWidth = 6;
            this->nodeNumColumn->Name = L"nodeNumColumn";
            this->nodeNumColumn->Width = 30;
            //
            // nodeValueColumn
            //
            this->nodeValueColumn->HeaderText = L"Value";
            this->nodeValueColumn->MinimumWidth = 6;
            this->nodeValueColumn->Name = L"nodeValueColumn";
            this->nodeValueColumn->Width = 150;
            //
            // nodeSegmentColumn
            //
            this->nodeSegmentColumn->HeaderText = L"Segment";
            this->nodeSegmentColumn->MinimumWidth = 6;
            this->nodeSegmentColumn->Name = L"nodeSegmentColumn";
            this->nodeSegmentColumn->Width = 70;
            //
            // linkTable
            //
            this->linkTable->ColumnHeadersHeightSizeMode = System::Windows::Forms::DataGridViewColumnHeadersHeightSizeMode::AutoSize;
            this->linkTable->Columns->AddRange(gcnew cli::array< System::Windows::Forms::DataGridViewColumn^  >(3) {
                this->linkNumColumn,
                    this->linkSegmentInColumn, this->linkSegmentOutColumn
            });
            this->linkTable->Location = System::Drawing::Point(5, 24);
            this->linkTable->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
            this->linkTable->Name = L"linkTable";
            this->linkTable->RowHeadersWidth = 4;
            this->linkTable->RowTemplate->Height = 27;
            this->linkTable->Size = System::Drawing::Size(395, 495);
            this->linkTable->TabIndex = 13;
            //
            // linkNumColumn
            //
            this->linkNumColumn->HeaderText = L"#";
            this->linkNumColumn->MinimumWidth = 50;
            this->linkNumColumn->Name = L"linkNumColumn";
            this->linkNumColumn->Width = 50;
            //
            // linkSegmentInColumn
            //
            this->linkSegmentInColumn->HeaderText = L"Segment In";
            this->linkSegmentInColumn->MinimumWidth = 120;
            this->linkSegmentInColumn->Name = L"linkSegmentInColumn";
            this->linkSegmentInColumn->Width = 120;
```

```
251             //
252             // linkSegmentOutColumn
253             //
254             this->linkSegmentOutColumn->HeaderText = L"Segment Out";
255             this->linkSegmentOutColumn->MinimumWidth = 120;
256             this->linkSegmentOutColumn->Name = L"linkSegmentOutColumn";
257             this->linkSegmentOutColumn->Width = 120;
258             //
259             // nodeGroupbox
260             //
261             this->nodeGroupbox->Controls->Add(this->nodeTable);
262             this->nodeGroupbox->Font = (gcnew System::Drawing::Font(L"Arial", 10));
263             this->nodeGroupbox->Location = System::Drawing::Point(704, 14);
264             this->nodeGroupbox->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
265             this->nodeGroupbox->Name = L"nodeGroupbox";
266             this->nodeGroupbox->Padding = System::Windows::Forms::Padding(3, 2, 3, 2);
267             this->nodeGroupbox->Size = System::Drawing::Size(361, 530);
268             this->nodeGroupbox->TabIndex = 14;
269             this->nodeGroupbox->TabStop = false;
270             this->nodeGroupbox->Text = L"Node";
271             //
272             // linkGroupbox
273             //
274             this->linkGroupbox->Controls->Add(this->linkTable);
275             this->linkGroupbox->FlatStyle = System::Windows::Forms::FlatStyle::System;
276             this->linkGroupbox->Font = (gcnew System::Drawing::Font(L"Arial", 10));
277             this->linkGroupbox->Location = System::Drawing::Point(1071, 14);
278             this->linkGroupbox->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
279             this->linkGroupbox->Name = L"linkGroupbox";
280             this->linkGroupbox->Padding = System::Windows::Forms::Padding(3, 2, 3, 2);
281             this->linkGroupbox->Size = System::Drawing::Size(408, 530);
282             this->linkGroupbox->TabIndex = 15;
283             this->linkGroupbox->TabStop = false;
284             this->linkGroupbox->Text = L"Link";
285             //
286             // pathFindingGroupbox
287             //
288             this->pathFindingGroupbox->Controls->Add(this->checkBox2);
289             this->pathFindingGroupbox->Controls->Add(this->checkBox1);
290             this->pathFindingGroupbox->Controls->Add(this->searchMethodCombobox);
291             this->pathFindingGroupbox->Controls->Add(this->searchSegmentTextbox);
292             this->pathFindingGroupbox->Controls->Add(this->searchButton);
293             this->pathFindingGroupbox->Font = (gcnew System::Drawing::Font(L"Arial", 10));
294             this->pathFindingGroupbox->ImeMode = System::Windows::Forms::ImeMode::On;
295             this->pathFindingGroupbox->Location = System::Drawing::Point(15, 330);
296             this->pathFindingGroupbox->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
297             this->pathFindingGroupbox->Name = L"pathFindingGroupbox";
298             this->pathFindingGroupbox->Padding = System::Windows::Forms::Padding(3, 2, 3, 2);
299             this->pathFindingGroupbox->Size = System::Drawing::Size(684, 214);
300             this->pathFindingGroupbox->TabIndex = 16;
```

```
301             this->pathFindingGroupbox->TabStop = false;
302             this->pathFindingGroupbox->Text = L"Path Finding";
303             //
304             // checkBox2
305             //
306             this->checkBox2->AutoSize = true;
307             this->checkBox2->Location = System::Drawing::Point(288, 0);
308             this->checkBox2->Name = L"checkBox2";
309             this->checkBox2->Size = System::Drawing::Size(126, 23);
310             this->checkBox2->TabIndex = 17;
311             this->checkBox2->Text = L"Export to File";
312             this->checkBox2->UseVisualStyleBackColor = true;
313             //
314             // checkBox1
315             //
316             this->checkBox1->AutoSize = true;
317             this->checkBox1->Location = System::Drawing::Point(126, 0);
318             this->checkBox1->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
319             this->checkBox1->Name = L"checkBox1";
320             this->checkBox1->Size = System::Drawing::Size(136, 23);
321             this->checkBox1->TabIndex = 16;
322             this->checkBox1->Text = L"Show Process";
323             this->checkBox1->UseVisualStyleBackColor = true;
324             this->checkBox1->CheckedChanged += gcnew System::EventHandler(this, &MyForm::checkBox1_CheckedChanged);
325             //
326             // searchMethodCombobox
327             //
328             this->searchMethodCombobox->FlatStyle = System::Windows::Forms::FlatStyle::System;
329             this->searchMethodCombobox->FormattingEnabled = true;
330             this->searchMethodCombobox->Items->AddRange(gcnew cli::array< System::Object^  >(2) {
331                 L"SGM -- Sequence-to-Graph Matching",
332                     L"SGA -- Sequence-to-Graph Alignment"
333             });
334             this->searchMethodCombobox->Location = System::Drawing::Point(1, 170);
335             this->searchMethodCombobox->Margin = System::Windows::Forms::Padding(4);
336             this->searchMethodCombobox->Name = L"searchMethodCombobox";
337             this->searchMethodCombobox->Size = System::Drawing::Size(527, 27);
338             this->searchMethodCombobox->TabIndex = 14;
339             this->searchMethodCombobox->Text = L"--- Please Select Searching Method ---";
340             //
341             // searchSegmentTextbox
342             //
343             this->searchSegmentTextbox->Location = System::Drawing::Point(5, 31);
344             this->searchSegmentTextbox->Margin = System::Windows::Forms::Padding(4);
345             this->searchSegmentTextbox->Name = L"searchSegmentTextbox";
346             this->searchSegmentTextbox->Size = System::Drawing::Size(667, 130);
347             this->searchSegmentTextbox->TabIndex = 5;
348             this->searchSegmentTextbox->Text = L"";
349             //
350             // importGraphGroupbox
```

```
351             //
352             this->importGraphGroupbox->Controls->Add(this->importGraphButton);
353             this->importGraphGroupbox->Font = (gcnew System::Drawing::Font(L"Arial", 10));
354             this->importGraphGroupbox->Location = System::Drawing::Point(15, 14);
355             this->importGraphGroupbox->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
356             this->importGraphGroupbox->Name = L"importGraphGroupbox";
357             this->importGraphGroupbox->Padding = System::Windows::Forms::Padding(3, 2, 3, 2);
358             this->importGraphGroupbox->Size = System::Drawing::Size(176, 114);
359             this->importGraphGroupbox->TabIndex = 17;
360             this->importGraphGroupbox->TabStop = false;
361             this->importGraphGroupbox->Text = L"Import Graph";
362             //
363             // addSegmentGroupbox
364             //
365             this->addSegmentGroupbox->Controls->Add(this->button1);
366             this->addSegmentGroupbox->Controls->Add(this->addSegmentTextbox);
367             this->addSegmentGroupbox->Controls->Add(this->addSegmentButton);
368             this->addSegmentGroupbox->Font = (gcnew System::Drawing::Font(L"Arial", 10));
369             this->addSegmentGroupbox->Location = System::Drawing::Point(15, 132);
370             this->addSegmentGroupbox->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
371             this->addSegmentGroupbox->Name = L"addSegmentGroupbox";
372             this->addSegmentGroupbox->Padding = System::Windows::Forms::Padding(3, 2, 3, 2);
373             this->addSegmentGroupbox->Size = System::Drawing::Size(684, 179);
374             this->addSegmentGroupbox->TabIndex = 18;
375             this->addSegmentGroupbox->TabStop = false;
376             this->addSegmentGroupbox->Text = L"Add Segment";
377             //
378             // addSegmentTextbox
379             //
380             this->addSegmentTextbox->Location = System::Drawing::Point(5, 30);
381             this->addSegmentTextbox->Margin = System::Windows::Forms::Padding(4);
382             this->addSegmentTextbox->Name = L"addSegmentTextbox";
383             this->addSegmentTextbox->Size = System::Drawing::Size(667, 104);
384             this->addSegmentTextbox->TabIndex = 13;
385             this->addSegmentTextbox->Text = L"";
386             //
387             // importSequenceGroupbox
388             //
389             this->importSequenceGroupbox->Controls->Add(this->importSequenceButton);
390             this->importSequenceGroupbox->Font = (gcnew System::Drawing::Font(L"Arial", 10));
391             this->importSequenceGroupbox->Location = System::Drawing::Point(196, 14);
392             this->importSequenceGroupbox->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
393             this->importSequenceGroupbox->Name = L"importSequenceGroupbox";
394             this->importSequenceGroupbox->Padding = System::Windows::Forms::Padding(3, 2, 3, 2);
395             this->importSequenceGroupbox->Size = System::Drawing::Size(176, 114);
396             this->importSequenceGroupbox->TabIndex = 19;
397             this->importSequenceGroupbox->TabStop = false;
398             this->importSequenceGroupbox->Text = L"Import Sequence";
399             //
400             // importSequenceButton
```

```
401             //
402             this->importSequenceButton->Font = (gcnew System::Drawing::Font(L"Arial", 10));
403             this->importSequenceButton->Location = System::Drawing::Point(22, 46);
404             this->importSequenceButton->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
405             this->importSequenceButton->Name = L"importSequenceButton";
406             this->importSequenceButton->Size = System::Drawing::Size(133, 32);
407             this->importSequenceButton->TabIndex = 12;
408             this->importSequenceButton->Text = L"From File";
409             this->importSequenceButton->UseVisualStyleBackColor = true;
410             this->importSequenceButton->Click += gcnew System::EventHandler(this, &MyForm::ImportSequenceButton_Click);
411             //
412             // addLinkGroupbox
413             //
414             this->addLinkGroupbox->Controls->Add(this->toTextbox);
415             this->addLinkGroupbox->Controls->Add(this->addLinkButton);
416             this->addLinkGroupbox->Controls->Add(this->fromTextbox);
417             this->addLinkGroupbox->Controls->Add(this->toLabel);
418             this->addLinkGroupbox->Controls->Add(this->fromLebel);
419             this->addLinkGroupbox->Font = (gcnew System::Drawing::Font(L"Arial", 10));
420             this->addLinkGroupbox->Location = System::Drawing::Point(377, 14);
421             this->addLinkGroupbox->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
422             this->addLinkGroupbox->Name = L"addLinkGroupbox";
423             this->addLinkGroupbox->Padding = System::Windows::Forms::Padding(3, 2, 3, 2);
424             this->addLinkGroupbox->Size = System::Drawing::Size(321, 114);
425             this->addLinkGroupbox->TabIndex = 20;
426             this->addLinkGroupbox->TabStop = false;
427             this->addLinkGroupbox->Text = L"Add Link";
428             //
429             // toTextbox
430             //
431             this->toTextbox->Location = System::Drawing::Point(211, 38);
432             this->toTextbox->Margin = System::Windows::Forms::Padding(4);
433             this->toTextbox->Name = L"toTextbox";
434             this->toTextbox->Size = System::Drawing::Size(99, 27);
435             this->toTextbox->TabIndex = 12;
436             //
437             // fromTextbox
438             //
439             this->fromTextbox->Location = System::Drawing::Point(67, 38);
440             this->fromTextbox->Margin = System::Windows::Forms::Padding(4);
441             this->fromTextbox->Name = L"fromTextbox";
442             this->fromTextbox->Size = System::Drawing::Size(99, 27);
443             this->fromTextbox->TabIndex = 11;
444             //
445             // toLabel
446             //
447             this->toLabel->AutoSize = true;
448             this->toLabel->Location = System::Drawing::Point(173, 41);
449             this->toLabel->Name = L"toLabel";
450             this->toLabel->Size = System::Drawing::Size(25, 19);
```

```
451             this->toLabel->TabIndex = 10;
452             this->toLabel->Text = L"To";
453             //
454             // fromLebel
455             //
456             this->fromLebel->AutoSize = true;
457             this->fromLebel->Location = System::Drawing::Point(5, 42);
458             this->fromLebel->Name = L"fromLebel";
459             this->fromLebel->Size = System::Drawing::Size(47, 19);
460             this->fromLebel->TabIndex = 9;
461             this->fromLebel->Text = L"From";
462             //
463             // resultGroupbox
464             //
465             this->resultGroupbox->Controls->Add(this->resultTextbox);
466             this->resultGroupbox->Font = (gcnew System::Drawing::Font(L"Arial", 10));
467             this->resultGroupbox->Location = System::Drawing::Point(15, 538);
468             this->resultGroupbox->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
469             this->resultGroupbox->Name = L"resultGroupbox";
470             this->resultGroupbox->Padding = System::Windows::Forms::Padding(3, 2, 3, 2);
471             this->resultGroupbox->Size = System::Drawing::Size(1464, 275);
472             this->resultGroupbox->TabIndex = 21;
473             this->resultGroupbox->TabStop = false;
474             this->resultGroupbox->Text = L"Result";
475             //
476             // button1
477             //
478             this->button1->Location = System::Drawing::Point(453, 141);
479             this->button1->Name = L"button1";
480             this->button1->Size = System::Drawing::Size(75, 23);
481             this->button1->TabIndex = 14;
482             this->button1->Text = L"button1";
483             this->button1->UseVisualStyleBackColor = true;
484             this->button1->Click += gcnew System::EventHandler(this, &MyForm::button1_Click_1);
485             //
486             // MyForm
487             //
488             this->AutoScaleDimensions = System::Drawing::SizeF(8, 15);
489             this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
490             this->ClientSize = System::Drawing::Size(1488, 822);
491             this->Controls->Add(this->resultGroupbox);
492             this->Controls->Add(this->addLinkGroupbox);
493             this->Controls->Add(this->importSequenceGroupbox);
494             this->Controls->Add(this->addSegmentGroupbox);
495             this->Controls->Add(this->importGraphGroupbox);
496             this->Controls->Add(this->pathFindingGroupbox);
497             this->Controls->Add(this->linkGroupbox);
498             this->Controls->Add(this->nodeGroupbox);
499             this->Margin = System::Windows::Forms::Padding(3, 2, 3, 2);
500             this->Name = L"MyForm";
```

```
501         this->Text = L"Final Year Project";
502         this->Load += gcnew System::EventHandler(this, &MyForm::MyForm_Load);
503         (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this->nodeTable))->EndInit();
504         (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this->linkTable))->EndInit();
505         this->nodeGroupbox->ResumeLayout(false);
506         this->linkGroupbox->ResumeLayout(false);
507         this->pathFindingGroupbox->ResumeLayout(false);
508         this->pathFindingGroupbox->PerformLayout();
509         this->importGraphGroupbox->ResumeLayout(false);
510         this->addSegmentGroupbox->ResumeLayout(false);
511         this->importSequenceGroupbox->ResumeLayout(false);
512         this->addLinkGroupbox->ResumeLayout(false);
513         this->addLinkGroupbox->PerformLayout();
514         this->resultGroupbox->ResumeLayout(false);
515         this->ResumeLayout(false);
516     }
517
518     string stdStr(String^ str)
519     {
520         return context.marshal_as<std::string>(str);
521     }
522
523     String^ sysStr(string str)
524     {
525         return gcnew String(str.c_str());
526     }
527
528     void print(string str)
529     {
530         resultTextbox->AppendText(sysStr(str));
531     }
532
533     void refreshVertex()
534     {
535         nodeTable->Rows->Clear();
536
537         for (int i = 0; i < myGraph.getNodeCount(); i++)
538         {
539             string s = "";
540             s += myGraph.nodeList[i].value;
541             nodeTable->Rows->Add(i, sysStr(s), myGraph.nodeList[i].segment);
542         }
543     }
544
545     void refreshLink()
546     {
547         int count = 0;
548         linkTable->Rows->Clear();
549
550         for (int i = 0; i < myGraph.segmentCount; i++)
```

```
551             {
552                 for (int j : myGraph.nodeList[myGraph.segmentList[i].endNode].next)
553                 {
554                     linkTable->Rows->Add(++count, i + 1, myGraph.nodeList[j].segment);
555                 }
556 
557             }
558         }
559 
560     void addSegmentButton_Click(System::Object^ sender, System::EventArgs^ e)
561     {
562             string segment = stdStr(addSegmentTextbox->Text);
563             if (addSegmentTextbox->Text != "")
564             {
565                 for (int i = 0; i < segment.length(); i++)
566                 {
567                     string a = "";
568                     a += segment[i];
569                     nodeTable->Rows->Add(myGraph.nodeCount + i, sysStr(a), myGraph.segmentCount + 1);
570                 }
571                 myGraph.addSegment(stdStr(addSegmentTextbox->Text));
572                 addSegmentTextbox->Text = "";
573             }
574         }
575 
576     private: System::Void MyForm_Load(System::Object^ sender, System::EventArgs^ e)
577     {
578     }
579 
580     private: System::Void addLinkButton_Click(System::Object^ sender, System::EventArgs^ e)
581     {
582         if (fromTextbox->Text != "" && toTextbox->Text != "")
583         {
584             int count = myGraph.segmentCount;
585             int from = int::Parse(fromTextbox->Text);
586             int to = int::Parse(toTextbox->Text);
587 
588             if (to > from && to <= count)
589             {
590                 if (myGraph.addLink(from, to))
591                 {
592                     linkTable->Rows->Add(myGraph.getLinkCount(), from, to);
593                     fromTextbox->Text = "";
594                     toTextbox->Text = "";
595                 }
596             }
597         }
598     }
599 
600     private: System::Void importGraphButton_Click(System::Object^ sender, System::EventArgs^ e)
```

```
601     {
602         String^ Filename;
603         Graph newGraph;
604         OpenFileDialog^ SelectFileDialog = gcnew OpenFileDialog();
605         SelectFileDialog->Filter = "gfa files (*.gfa)|*.gfa";
606         SelectFileDialog->FilterIndex = 1;
607         SelectFileDialog->InitialDirectory = Environment::GetFolderPath(Environment::SpecialFolder::Desktop);
608         if (SelectFileDialog->ShowDialog() == System::Windows::Forms::DialogResult::OK)
609         {
610
611             Filename = SelectFileDialog->FileName;
612             ifstream file(stdStr(Filename));
613             string str;
614             int index;
615             string type;
616             string seq;
617             string op;
618             int from, to;
619             unsigned long times = clock();
620             while (getline(file, str))
621             {
622                 istringstream ss(str);
623                 ss >> type;
624                 if (str.length() > 0)
625                 {
626                     if (type == "S")
627                     {
628                         ss >> index >> seq;
629                         newGraph.addSegment(seq);
630                     }
631                     else if (type == "L")
632                     {
633                         ss >> from >> op >> to;
634                         newGraph.addLink(from, to);
635                     }
636                 }
637
638             }
639             unsigned long timed = clock();
640             print("Graph Imported, time spent: " + to_string(timed - times) + "ms.\n");
641             myGraph = newGraph;
642             refreshVertex();
643             refreshLink();
644         }
645     }
646
647     private: System::Void ImportSequenceButton_Click(System::Object^ sender, System::EventArgs^ e)
648     {
649         String^ Filename;
650         OpenFileDialog^ SelectFileDialog = gcnew OpenFileDialog();
```

```
651         SelectFileDialog->Filter = "txt files (*.txt)|*.txt";
652         SelectFileDialog->FilterIndex = 1;
653         SelectFileDialog->InitialDirectory = Environment::GetFolderPath(Environment::SpecialFolder::Desktop);
654         if (SelectFileDialog->ShowDialog() == System::Windows::Forms::DialogResult::OK)
655         {
656             Filename = SelectFileDialog->FileName;
657             ifstream file(stdStr(Filename));
658             string str;
659             while (getline(file, str))
660             {
661                 searchSegmentTextbox->Text = sysStr(str);
662             }
663         }
664     }
665     int doesFileExist(char* filename)
666     {
667         FILE* file;
668         if (file = fopen(filename, "r"))
669         {
670             fclose(file);
671             return true;
672         }
673         return false;
674     }
675     private: System::Void searchButton_Click(System::Object^ sender, System::EventArgs^ e)
676     {
677         String^ method = searchMethodCombobox->Text;
678         string seq = stdStr(searchSegmentTextbox->Text);
679         if (method == "SGM -- Sequence-to-Graph Matching")
680         {
681             unsigned long times = clock();
682             int index = myGraph.extactMatch(seq);
683             if (index == -1)
684             {
685                 unsigned long timed = clock();
686                 print("-------------------------------------------------------\n");
687                 print("Searching Time Spent: " + to_string(timed - times) + "ms\n");
688                 print("Sequence: " + seq + "\n");
689                 print("Method: SGM -- Sequence-to-Graph Matching\n");
690                 print("Path not found!\n");
691             }
692             else
693             {
694                 Path path = myGraph.getPath(index, seq.length());
695                 unsigned long timed = clock();
696                 print("-------------------------------------------------------\n");
697                 print("Searching Time Spent: " + to_string(timed - times) + "ms.\n");
698                 print("Search Sequence: " + seq + "\n");
699                 print("Sequence Length: " + to_string(seq.length()) + "\n");
700                 print("Method: SGM -- Sequence-to-Graph Matching\n");
```

```
701                 print("Start Node: " + to_string(path.startIndex) + "\n");
702                 print("End Node: " + to_string(path.endIndex) + "\n");
703                 print("Start Segment: " + to_string(path.startSegment) + "\n");
704                 print("End Segment: " + to_string(path.endSegment) + "\n");
705                 print("Node Path:  ");
706                 for (int i = path.path.size() - 1; i >= 0; i--)
707                 {
708                     print(to_string(path.path[i]));
709                     if (i != 0)
710                     {
711                         print(" > ");
712                     }
713                 }
714                 print("\n");
715                 print("Segment Path:  ");
716                 int segment = -1;
717                 int seg;
718                 for (int i = path.path.size() - 1; i >= 0; i--)
719                 {
720                     seg = myGraph.nodeList[path.path[i]].segment;
721                     if (seg != segment)
722                     {
723                         if (segment != -1)
724                         {
725                             print(" > ");
726                         }
727                         print(to_string(seg));
728                         segment = seg;
729                     }
730                 }
731                 print("\n");
732             }
733             if (checkBox2->Checked == true)
734             {
735                 string filename = "result.txt";
736                 char* fname = new char[filename.length() + 1];
737                 strcpy(fname, filename.c_str());
738                 int i = 1;
739                 while (doesFileExist(fname) == 1)
740                 {
741                     filename = "result(" + to_string(i++) + ").txt";
742                     fname = new char[filename.length() + 1];
743                     strcpy(fname, filename.c_str());
744                 }
745                 ofstream myfile(filename);
746 
747                 if (myfile.is_open())
748                 {
749                     if (index == -1)
750                     {
```

```
751                     unsigned long timed = clock();
752                     myfile << "Searching Time Spent: " + to_string(timed - times) + "ms\n";
753                     myfile << "Sequence: " + seq + "\n";
754                     myfile << "Method: SGM -- Sequence-to-Graph Matching\n";
755                     myfile << "Path not found!\n";
756                 }
757                 else
758                 {
759                     Path path = myGraph.getPath(index, seq.length());
760                     unsigned long timed = clock();
761                     myfile << "Searching Time Spent: " + to_string(timed - times) + "ms.\n";
762                     myfile << "Search Sequence: " + seq + "\n";
763                     myfile << "Sequence Length: " + to_string(seq.length()) + "\n";
764                     myfile << "Method: SGM -- Sequence-to-Graph Matching\n";
765                     myfile << "Start Node: " + to_string(path.startIndex) + "\n";
766                     myfile << "End Node: " + to_string(path.endIndex) + "\n";
767                     myfile << "Start Segment: " + to_string(path.startSegment) + "\n";
768                     myfile << "End Segment: " + to_string(path.endSegment) + "\n";
769                     myfile << "Node Path:  ";
770                     for (int i = path.path.size() - 1; i >= 0; i--)
771                     {
772                         myfile << to_string(path.path[i]);
773                         if (i != 0)
774                         {
775                             myfile << " > ";
776                         }
777                     }
778                     myfile << "\n";
779                     myfile << "Segment Path:  ";
780                     int segment = -1;
781                     int seg;
782                     for (int i = path.path.size() - 1; i >= 0; i--)
783                     {
784                         seg = myGraph.nodeList[path.path[i]].segment;
785                         if (seg != segment)
786                         {
787                             if (segment != -1)
788                             {
789                                 myfile << " > ";
790                             }
791                             myfile << to_string(seg);
792                             segment = seg;
793                         }
794                     }
795                     myfile << "\n";
796                 }
797                 myfile.close();
798             }
799         }
800
```

```
801         }
802         else if (method == "SGA -- Sequence-to-Graph Alignment")
803         {
804             unsigned long times = clock();
805             vector<Path> path = myGraph.minimumEditDistance(seq);
806             unsigned long timed = clock();
807             print("-------------------------------------------------------\n");
808             print("Searching Time Spent: " + to_string(timed - times) + "ms.\n");
809             print("Searching Sequence: " + seq + "\n");
810             print("Sequence Length: " + to_string(seq.length()) + "\n");
811             print("Method: SGA -- Sequence-to-Graph Alignment\n");
812             for (int i = 0; i < path.size(); i++)
813             {
814                 print("-------------------------------------------------------\n");
815                 print("Top: " + to_string(i + 1) + "\n");
816                 print("Edit Distance : " + to_string(path[i].editDistance) + "\n");
817                 print("Start Node: " + to_string(path[i].startIndex) + "\n");
818                 print("End Node: " + to_string(path[i].endIndex) + "\n");
819                 print("Start Segment: " + to_string(path[i].startSegment) + "\n");
820                 print("End Segment: " + to_string(path[i].endSegment) + "\n");
821                 print("Node Path: ");
822                 int node = -1;
823                 for (int pathNode = path[i].path.size() - 1; pathNode >= 0; pathNode--)
824                 {
825                     if (!(path[i].path[pathNode] == -1 || path[i].path[pathNode] == node))
826                     {
827                         print(to_string(path[i].path[pathNode]) + " > ");
828                         node = path[i].path[pathNode];
829                     }
830                 }
831                 print("\n");
832                 print("Segment Path: ");
833                 int segment = -1;
834                 for (int pathNode = path[i].path.size() - 1; pathNode >= 0; pathNode--)
835                 {
836                     if (!(path[i].path[pathNode] == -1 || myGraph.nodeList[path[i].path[pathNode]].segment == segment))
837                     {
838                         print(to_string(myGraph.nodeList[path[i].path[pathNode]].segment) + " > ");
839                         segment = myGraph.nodeList[path[i].path[pathNode]].segment;
840                     }
841                 }
842                 print("\n");
843                 if (checkBox1->Checked == true)
844                 {
845                     print("Edit Process:\n");
846                     int index = 0;
847                     int a = path[i].path.size();
848                     for (int pathNode = path[i].path.size() - 1; pathNode >= 0; pathNode--)
849                     {
850                         if (path[i].state[pathNode] == SAME)
```

```
851                     {
852                         index++;
853                         string message = "Node: " + to_string(path[i].path[pathNode]);
854                         message = message + " Seg.: " + to_string(myGraph.nodeList[path[i].path[pathNode]].segment);
855                         message = message + " Value: " + myGraph.nodeList[path[i].path[pathNode]].value;
856                         print(message + "\n");
857                     }
858                     else if (path[i].state[pathNode] == CHANGE)
859                     {
860                         string message = "Node: " + to_string(path[i].path[pathNode]);
861                         message = message + " Seg.: " + to_string(myGraph.nodeList[path[i].path[pathNode]].segment);
862                         message = message + " Value: " + seq[index];
863                         message = message + " (Change From: " + myGraph.nodeList[path[i].path[pathNode]].value + ")";
864                         print(message + "\n");
865                         index++;
866                     }
867                     else if (path[i].state[pathNode] == INSERT)
868                     {
869                         string message = "Node: " + to_string(path[i].path[pathNode]);
870                         message = message + " Seg.: " + to_string(myGraph.nodeList[path[i].path[pathNode]].segment);
871                         message = message + " Value: " + myGraph.nodeList[path[i].path[pathNode]].value + " (Remove)";
872                         print(message + "\n");
873                     }
874                     else if (path[i].state[pathNode] == REMOVE)
875                     {
876                         string message = "Insert: ";
877                         message = message + seq[index];
878                         index++;
879                         print(message + "\n");
880                     }
881                 }
882             }
883         }
884         if (checkBox2->Checked == true)
885         {
886             string filename = "result.txt";
887             char* fname = new char[filename.length() + 1];
888             strcpy(fname, filename.c_str());
889             int i = 1;
890             while (doesFileExist(fname) == 1)
891             {
892                 filename = "result(" + to_string(i++) + ").txt";
893                 fname = new char[filename.length() + 1];
894                 strcpy(fname, filename.c_str());
895             }
896
897             ofstream myfile(filename);
898             if (myfile.is_open())
899             {
900                 myfile << "Searching Time Spent: " + to_string(timed - times) + "ms.\n";
```

```
901                 myfile << "Searching Sequence: " + seq + "\n";
902                 myfile << "Sequence Length: " + to_string(seq.length()) + "\n";
903                 myfile << "Method: SGA -- Sequence-to-Graph Alignment\n";
904                 for (int i = 0; i < path.size(); i++)
905                 {
906                     myfile << "-------------------------------------------------------\n";
907                     myfile << "Top: " + to_string(i + 1) + "\n";
908                     myfile << "Edit Distance : " + to_string(path[i].editDistance) + "\n";
909                     myfile << "Start Node: " + to_string(path[i].startIndex) + "\n";
910                     myfile << "End Node: " + to_string(path[i].endIndex) + "\n";
911                     myfile << "Start Segment: " + to_string(path[i].startSegment) + "\n";
912                     myfile << "End Segment: " + to_string(path[i].endSegment) + "\n";
913                     myfile << "Node Path: ";
914                     int node = -1;
915                     for (int pathNode = path[i].path.size() - 1; pathNode >= 0; pathNode--)
916                     {
917                         if (!(path[i].path[pathNode] == -1 || path[i].path[pathNode] == node))
918                         {
919                             myfile << to_string(path[i].path[pathNode]) + " > ";
920                             node = path[i].path[pathNode];
921                         }
922                     }
923                     myfile << "\n";
924                     myfile << "Segment Path: ";
925                     int segment = -1;
926                     for (int pathNode = path[i].path.size() - 1; pathNode >= 0; pathNode--)
927                     {
928                         if (!(path[i].path[pathNode] == -1 || myGraph.nodeList[path[i].path[pathNode]].segment == segment))
929                         {
930                             myfile << to_string(myGraph.nodeList[path[i].path[pathNode]].segment) + " > ";
931                             segment = myGraph.nodeList[path[i].path[pathNode]].segment;
932                         }
933                     }
934                     myfile << "\n";
935                     myfile << "Edit Process:\n";
936                     int index = 0;
937                     int a = path[i].path.size();
938                     for (int pathNode = path[i].path.size() - 1; pathNode >= 0; pathNode--)
939                     {
940                         if (path[i].state[pathNode] == SAME)
941                         {
942                             index++;
943                             string message = "Node: " + to_string(path[i].path[pathNode]);
944                             message = message + " Seg.: " + to_string(myGraph.nodeList[path[i].path[pathNode]].segment);
945                             message = message + " Value: " + myGraph.nodeList[path[i].path[pathNode]].value;
946                             myfile << message + "\n";
947                         }
948                         else if (path[i].state[pathNode] == CHANGE)
949                         {
950                             string message = "Node: " + to_string(path[i].path[pathNode]);
```

```
951                                 message = message + " Seg.: " + to_string(myGraph.nodeList[path[i].path[pathNode]].segment);
952                                 message = message + " Value: " + seq[index];
953                                 message = message + " (Change From: " + myGraph.nodeList[path[i].path[pathNode]].value + ")";
954                                 myfile << message + "\n";
955                                 index++;
956                             }
957                             else if (path[i].state[pathNode] == INSERT)
958                             {
959                                 string message = "Node: " + to_string(path[i].path[pathNode]);
960                                 message = message + " Seg.: " + to_string(myGraph.nodeList[path[i].path[pathNode]].segment);
961                                 message = message + " Value: " + myGraph.nodeList[path[i].path[pathNode]].value + " (Remove)";
962                                 myfile << message + "\n";
963                             }
964                             else if (path[i].state[pathNode] == REMOVE)
965                             {
966                                 string message = "Insert: ";
967                                 message = message + seq[index];
968                                 index++;
969                                 myfile << message + "\n";
970                             }
971                         }
972                     }
973                     myfile.close();
974                 }
975             }
976         }
977 
978     }
979 
980 private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
981 {
982     for (int i = 0; i < myGraph.getNodeCount(); i++)
983     {
984         print(to_string(myGraph.nodeList[i].segment) + "\n");
985     }
986 }
987 
988 private: System::Void checkBox1_CheckedChanged(System::Object^ sender, System::EventArgs^ e)
989 {
990 }
991 
992 private: System::Void button1_Click_1(System::Object^ sender, System::EventArgs^ e) {
993     print("NodeCount: " + to_string(myGraph.nodeCount) + "\n");
994     print("EdgeCount: " + to_string(myGraph.edgeCount) + "\n");
995 }
996 };
997 }
998 
```

# Appendix III: GraphAlign.h

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
#include <list>
#include <math.h>
#include <stack>
using namespace std;
class Node
{
public:
    char value;
    list<int> next;
    list<int> prev;
    int index;
    int inDegree = 0;
    int outDegree = 0;
    int segment;
    long nfa;
};

class Segment
{
public:
    int startNode;
    int endNode;
};

class Path
{
public:
    vector<int> path;
    vector<char> state;
    int startSegment;
    int endSegment;
    int startIndex;
    int endIndex;
    int editDistance;

    //Default constructor
    Path();

    //mutator
    void setStartSegment(int segment);
    void setEndSegment(int segment);
    void setStartIndex(int index);
    void setEndIndex(int index);
    void appendNode(int index);
    void appendState(int state);
```

```
51
52          //accessor
53          int getStartSegment();
54          int getEndSegment();
55          int getStartIndex();
56          int getEndIndex();
57          vector<int> getPath();
58          vector<char> getState();
59
60          //destructor
61          ~Path()
62          {
63          }
64      };
65
66      class Graph
67      {
68      private:
69          int ED = 0;
70          int ARROW = 1;
71          int STATE = 2;
72          int SAME = 1;
73          int CHANGE = 2;
74          int INSERT = 3;
75          int REMOVE = 4;
76          int HEAD = 0;
77
78      public:
79          int nodeCount;
80          int linkCount;
81          int edgeCount;
82          int segmentCount;
83          vector<Node> nodeList;
84          vector<Segment> segmentList;
85
86          Graph();
87          bool addLink(int inSegment, int outSegment);
88          void addSegment(string segment);
89          int getNodeCount();
90          int getLinkCount();
91          int getEdgeCount();
92          int getsegmentCount();
93
94          //node related getter
95          list<int> getNext(int node);
96          char getValue(int node);
97          int getSegment(int node);
98          int getInDegree(int node);
99          int getOutDegree(int node);
100         int getIndex(int node);
```

```
101         long getNfa(int node);
102         list<int> getPrev(int node);
103
104         //node relate setter
105         void setIndex(int node, int index);
106         void setIndegree(int node, int degree);
107         void setOutdegree(int node, int degree);
108         void setSegment(int node, int segment);
109         void setNFA(int node, long nfa);
110         void setValue(int node, int value);
111
112         //segment relate setter
113         int getStartNode(int segment);
114         int getEndNode(int segment);
115         void setStartNode(int segment, int index);
116         void setEndNode(int segment, int index);
117
118         //search function
119         int extactMatch(string sequence);
120         Path getPath(int index, int size);
121         vector<Path> minimumEditDistance(string sequence);
122         ~Graph()
123         {
124
125         }
126     };
127
128     Path::Path()
129     {
130         startIndex = -1;
131         endIndex = -1;
132         startSegment = -1;
133         endSegment = -1;
134     }
135
136     void Path::setStartSegment(int seg)
137     {
138         this->startSegment = seg;
139     }
140
141     void Path::setEndSegment(int seg)
142     {
143         this->endSegment = seg;
144     }
145
146     void Path::setStartIndex(int index)
147     {
148         this->startIndex = index;
149     }
150
```

```
151 void Path::setEndIndex(int index)
152 {
153     this->endIndex = index;
154 }
155
156 void Path::appendNode(int index)
157 {
158     this->path.push_back(index);
159 }
160
161 void Path::appendState(int state)
162 {
163     this->state.push_back(state);
164 }
165
166 int Path::getStartSegment()
167 {
168     return this->startSegment;
169 }
170
171 int Path::getEndSegment()
172 {
173     return this->endSegment;
174 }
175
176 int Path::getStartIndex()
177 {
178     return this->startIndex;
179 }
180
181 int Path::getEndIndex()
182 {
183     return this->endIndex;
184 }
185
186 vector<int> Path::getPath()
187 {
188     return this->path;
189 }
190
191 vector<char> Path::getState()
192 {
193     return this->state;
194 }
195
196 Graph::Graph()
197 {
198     nodeCount = 0;
199     linkCount = 0;
200     edgeCount = 0;
```

```
201     segmentCount = 0;
202 }
203
204 bool Graph::addLink(int in, int out)
205 {
206     int inNode = segmentList[in - 1].endNode;
207     int outNode = segmentList[out - 1].startNode;
208     for (auto v : nodeList[inNode].next)
209     {
210         if (v == outNode)
211         {
212             return false;
213         }
214     }
215     linkCount++;
216     edgeCount++;
217     nodeList[inNode].outDegree++;
218     nodeList[outNode].inDegree++;
219     nodeList[outNode].prev.push_back(inNode);
220     nodeList[inNode].next.push_back(outNode);
221     return true;
222 }
223
224 void Graph::addSegment(string str)
225 {
226     Segment newSegment;
227     segmentCount++;
228     newSegment.startNode = nodeCount;
229     newSegment.endNode = nodeCount + str.length()-1;
230     segmentList.push_back(newSegment);
231     Node newNode;
232     newNode.value = str[0];
233     newNode.segment = segmentCount;
234     newNode.index = nodeCount;
235     nodeList.push_back(newNode);
236     nodeCount++;
237     if (str.length() > 1)
238     {
239         for (int i = 1; i < str.length(); i++)
240         {
241             Node nNode;
242             nNode.value = str[i];
243             nNode.index = nodeCount;
244             nNode.segment = segmentCount;
245             nNode.inDegree++;
246             nNode.prev.push_back(nodeCount-1);
247             nodeList.push_back(nNode);
248             nodeList[nodeCount-1].outDegree ++;
249             nodeList[nodeCount-1].next.push_back(nodeCount);
250             edgeCount++;
```

```
251                     nodeCount++;
252                 }
253             }
254         }
255
256     list<int> Graph::getNext(int node)
257         {
258             return this->nodeList[node].next;
259         }
260
261     char Graph::getValue(int node)
262         {
263             return this->nodeList[node].value;
264         }
265
266     int Graph::getSegment(int node)
267         {
268             return this->nodeList[node].segment;
269         }
270
271     int Graph::getLinkCount()
272         {
273             return this->linkCount;
274         }
275
276     int Graph::getNodeCount()
277         {
278             return this->nodeCount;
279         }
280
281     int Graph::getEdgeCount()
282         {
283             return this->edgeCount;
284         }
285
286     int Graph::getsegmentCount()
287         {
288             return this->segmentCount;
289         }
290
291     int Graph::getInDegree(int node)
292         {
293             return nodeList[node].inDegree;
294         }
295
296     int Graph::getOutDegree(int node)
297         {
298             return this->nodeList[node].outDegree;
299         }
300
```

```
301     int Graph::getIndex(int node)
302     {
303         return this->nodeList[node].index;
304     }
305
306     long Graph::getNfa(int node)
307     {
308         return this->nodeList[node].nfa;
309     }
310
311     list<int> Graph::getPrev(int node)
312     {
313         return this->nodeList[node].prev;
314     }
315
316      //node relate setter
317     void Graph::setIndex(int node, int index)
318     {
319         this->nodeList[node].index = index;
320     }
321
322     void Graph::setIndegree(int node, int deg)
323     {
324         this->nodeList[node].inDegree = deg;
325     }
326
327     void Graph::setOutdegree(int node, int deg)
328     {
329         this->nodeList[node].outDegree = deg;
330     }
331
332     void Graph::setSegment(int node, int seg)
333     {
334         this->nodeList[node].segment = seg;
335     }
336
337     void Graph::setNFA(int node, long nfa)
338     {
339         this->nodeList[node].nfa = nfa;
340     }
341
342     void Graph::setValue(int node, int value)
343     {
344         this->nodeList[node].value = value;
345     }
346
347     int Graph::getStartNode(int segment)
348     {
349         return segmentList[segment].endNode;
350     }
```

```
351
352     int Graph::getEndNode(int segment)
353     {
354         return segmentList[segment].startNode;
355     }
356
357     void Graph::setStartNode(int seg, int index)
358     {
359         this->segmentList[seg].startNode = index;
360     }
361
362     void Graph::setEndNode(int seg, int index)
363     {
364         this->segmentList[seg].endNode = index;
365     }
366
367     int Graph::extactMatch(string sequence)
368     {
369         int m = sequence.length();
370         long pattern_mask[256];
371         long R = ~1;
372         for (int i = 0; i <= 255; ++i)
373             pattern_mask[i] = ~0;
374         for (int i = 0; i < m; ++i)
375             pattern_mask[sequence[i]] &= ~(1L << i);
376
377
378         for (int i = 0; i < nodeCount; i++)
379         {
380             if(nodeList[i].inDegree == 0)
381             {
382                 nodeList[i].nfa = R | pattern_mask[nodeList[i].value];
383                 nodeList[i].nfa <<= 1;
384             }
385             if (nodeList[i].inDegree > 0)
386             {
387                 nodeList[i].nfa = nodeList[nodeList[i].prev.front()].nfa;
388                 if (nodeList[i].inDegree > 1)
389                 {
390                     for (int j : nodeList[i].prev)
391                     {
392                         nodeList[i].nfa &= nodeList[j].nfa;
393                     }
394                 }
395                 nodeList[i].nfa |= pattern_mask[nodeList[i].value];
396                 nodeList[i].nfa <<= 1;
397             }
398             if ((nodeList[i].nfa & (1L << m)) == 0)
399                 return nodeList[i].index;
400         }
```

```
401
402         return -1;
403     }
404
405     Path Graph::getPath(int index,int size)
406     {
407         Path path;
408         Node currNode = nodeList[index];
409         path.endIndex = index;
410         path.endSegment = nodeList[index].segment;
411         path.path.push_back(index);
412         if (size > 1)
413         {
414             for (int i = size-1; i >= 1; i--)
415             {
416                 if (currNode.inDegree == 1)
417                 {
418                     Node prevNode = nodeList[currNode.prev.front()];
419                     path.path.push_back(prevNode.index);
420                     currNode = prevNode;
421                     if (i == 1)
422                     {
423                         path.startIndex = prevNode.index;
424                         path.startSegment = prevNode.segment;
425                     }
426                 }
427                 else
428                 {
429                     for (int j : currNode.prev)
430                     {
431                         int mask = 1L << i;
432                         if (!(nodeList[j].nfa & (1L << i)))
433                         {
434                             path.path.push_back(j);
435                             currNode = nodeList[j];
436                             if (i == 1)
437                             {
438                                 path.startIndex = j;
439                                 path.startSegment = nodeList[j].segment;
440                             }
441                             break;
442                         }
443                     }
444                 }
445             }
446         }
447         else
448         {
449             path.startIndex = index;
450             path.startSegment = nodeList[index].segment;
```

```
451     }
452     return path;
453 }
454
455 vector<Path> Graph::minimumEditDistance(string pattern)
456 {
457     int len = pattern.length();
458
459     //3d array initialization
460     int*** dp = new int** [len + 1];
461     for (int i = 0; i <= len; i++)
462     {
463         dp[i] = new int* [nodeCount + 1];
464         for (int j = 0; j <= nodeCount; j++)
465         {
466             dp[i][j] = new int[3];
467         }
468     }
469     /*
470     * state table
471     * 1: Same
472     * 2: Change
473     * 3: Insert
474     * 4: Drop
475     */
476     dp[0][0][ED] = 0; //start of the table must be 0
477     dp[0][0][STATE] = 0; //start of the table must be 0
478     dp[0][0][ARROW] = 0; //start of the table must be 0
479
480     for (int Y = 1; Y <= len; Y++)
481     {
482         dp[Y][0][ED] = Y; //for the first column, its score always prev. row + 1
483         dp[Y][0][ARROW] = 0; //for the first column, it always point to the same column
484         dp[Y][0][STATE] = 4; //for the first column, it always is a delete state
485     }
486
487     for (int X = 1; X <= nodeCount; X++)
488     {
489         Node currNode = nodeList[X - 1];
490         if (currNode.inDegree == 0)
491         {
492             dp[0][X][ED] = 1; //score must be 1
493             dp[0][X][ARROW] = 0; //Point to the first column
494             dp[0][X][STATE] = INSERT; //the states for the start of the graph must be insert
495         }
496         else
497         {
498             dp[0][X][STATE] = INSERT; //first row must be insert state
499             dp[0][X][ED] = 1 + dp[0][currNode.prev.front() + 1][ED];
500             dp[0][X][ARROW] = currNode.prev.front() + 1;
```

```
501             if (currNode.inDegree > 1)
502             {
503                 for (int i : currNode.prev)
504                 {
505                     if (dp[0][X][ED] > 1 + dp[0][i + 1][ED])
506                     {
507                         dp[0][X][ED] = 1 + dp[0][i + 1][ED];
508                         dp[0][X][ARROW] = i + 1;
509                     }
510                 }
511             }
512         }
513     }
514
515     //loop for each row and col
516     for (int y = 1; y <= len; y++)
517     {
518         for (int x = 1; x <= nodeCount; x++)
519         {
520             Node currNode = nodeList[x - 1];
521             if (currNode.inDegree == 0)
522             {
523                 if (pattern[y - 1] == currNode.value)
524                 {
525                     dp[y][x][STATE] = SAME;
526                     dp[y][x][ED] = dp[y - 1][0][ED];
527                     dp[y][x][ARROW] = HEAD;
528                 }
529                 else
530                 {
531                     dp[y][x][STATE] = CHANGE;
532                     dp[y][x][ED] = dp[y - 1][0][ED];
533                     dp[y][x][ARROW] = HEAD;
534                     if (dp[y][x][ED] > dp[y - 1][x][ED])
535                     {
536                         dp[y][x][ED] = dp[y - 1][x][ED];
537                         dp[y][x][ARROW] = x;
538                         dp[y][x][STATE] = REMOVE;
539                     }
540                     dp[y][x][ED]++;
541                 }
542             }
543             else
544             {
545                 if (pattern[y - 1] == currNode.value)//Match case
546                 {
547                     dp[y][x][ED] = dp[y - 1][currNode.prev.front() + 1][ED];
548                     dp[y][x][ARROW] = currNode.prev.front() + 1;
549                     dp[y][x][STATE] = SAME;
550                     if (currNode.inDegree > 1)
```

```
551                     {
552                         for (int i : currNode.prev)
553                         {
554                             if (dp[y][x][ED] > dp[y - 1][i + 1][ED])
555                             {
556                                 dp[y][x][ED] = dp[y - 1][i + 1][ED];
557                                 dp[y][x][ARROW] = i + 1;
558                             }
559                         }
560                     }
561                 }
562                 else
563                 {
564                     dp[y][x][ED] = dp[y][currNode.prev.front() + 1][ED];
565                     dp[y][x][ARROW] = currNode.prev.front() + 1;
566                     dp[y][x][STATE] = INSERT;
567 
568                     if (dp[y][x][ED] > dp[y - 1][currNode.prev.front() + 1][ED])
569                     {
570                         dp[y][x][ED] = dp[y - 1][currNode.prev.front() + 1][ED];
571                         dp[y][x][ARROW] = currNode.prev.front() + 1;
572                         dp[y][x][STATE] = CHANGE;
573                     }
574                     if (dp[y][x][ED] > dp[y - 1][x][ED])
575                     {
576                         dp[y][x][ED] = dp[y - 1][x][ED];
577                         dp[y][x][ARROW] = x;
578                         dp[y][x][STATE] = REMOVE;
579                     }
580                     if (currNode.inDegree > 1)
581                     {
582                         for (int i : currNode.prev)
583                         {
584                             if (dp[y][x][ED] > dp[y - 1][i + 1][ED])
585                             {
586                                 dp[y][x][ED] = dp[y - 1][i + 1][ED];
587                                 dp[y][x][ARROW] = i + 1;
588                                 dp[y][x][STATE] = CHANGE;
589                             }
590                             if (dp[y][x][ED] > dp[y][i + 1][ED])
591                             {
592                                 dp[y][x][ED] = dp[y][i + 1][ED];
593                                 dp[y][x][ARROW] = i + 1;
594                                 dp[y][x][STATE] = INSERT;
595                             }
596                         }
597                     }
598                     dp[y][x][ED]++;
599                 }
600             }
```

```
601             }
602         }
603     }
604 
605     vector<Path> cheapPath;
606     int pointer = nodeCount;
607     int currRow,currCol;
608     while (pointer > 0)
609     {
610         currRow = len;
611         currCol = pointer;
612         if (dp[currRow][currCol][STATE] == INSERT)
613         {
614             pointer--;
615         }
616         else
617         {
618             Path myPath;
619             myPath.editDistance = -1;
620             while (currRow > 0)
621             {
622                 if (dp[currRow][currCol][STATE] == SAME)
623                 {
624                     myPath.path.push_back(currCol - 1);
625                     myPath.state.push_back(SAME);
626                     currCol = dp[currRow][currCol][ARROW];
627                     currRow--;
628                 }
629                 else if (dp[currRow][currCol][STATE] == INSERT)
630                 {
631                     myPath.path.push_back(currCol - 1);
632                     myPath.state.push_back(INSERT);
633                     currCol = dp[currRow][currCol][ARROW];
634                 }
635                 else if (dp[currRow][currCol][STATE] == REMOVE)
636                 {
637                     myPath.path.push_back(currCol - 1);
638                     myPath.state.push_back(REMOVE);
639                     currCol = dp[currRow][currCol][ARROW];
640                     currRow--;
641                 }
642                 else if (dp[currRow][currCol][STATE] == CHANGE)
643                 {
644                     myPath.path.push_back(currCol - 1);
645                     myPath.state.push_back(CHANGE);
646                     currCol = dp[currRow][currCol][ARROW];
647                     currRow--;
648                 }
649             }
650             myPath.endIndex = myPath.path[0];
```

```
651                 myPath.endSegment = nodeList[myPath.endIndex].segment;
652                 int si = 1;
653                 do {
654                     myPath.startIndex = myPath.path[myPath.path.size() - si];
655                     si++;
656                 } while ((myPath.startIndex == -1));
657                 myPath.startSegment = nodeList[myPath.startIndex].segment;
658                 myPath.editDistance = dp[len][pointer][ED] - dp[currRow][currCol][ED];
659                 int test = dp[1][myPath.path[myPath.path.size() - 1] + 1][ED];
660
661                 pointer--;
662                 for (int i = 0; i < cheapPath.size(); i++)
663                 {
664                     if (cheapPath[i].editDistance > myPath.editDistance)
665                     {
666                         cheapPath.insert(cheapPath.begin()+i, myPath);
667                         goto here;
668                     }
669                 }
670                 cheapPath.push_back(myPath);
671                 here:;
672                 if (cheapPath.size() > 5)
673                 {
674                     cheapPath.pop_back();
675                 }
676             }
677         }
678         delete[] dp;
679         return cheapPath;
680     }
```

# References:


[1] Rajesh, Prasad and S., Agarwal, *Optimal Shift-Or String Matching Algorithm for Multiple Patterns,* Lecture Notes in Engineering and Computer Science, Oct 01, 2020.

[2] Mikko Rautiainen, Veli Mäkinen, Tobias Marschall, *Bit-parallel sequence-to-graph alignment*, Bioinformatics, Volume 35, Issue 19, 1 October 2019, Pages 3599–3607, https://doi.org/10.1093/bioinformatics/btz162

[3] Robert Giegerich, *A systematic approach to dynamic programming in bioinformatics, Bioinformatics,* Volume 16, Issue 8, August 2000, Pages 665–677, https://doi.org/10.1093/bioinformatics/16.8.665

[4] Tutorial ink, *Types Of Data Structures*, October, 2017, https://tutorialink.com/ds/types-of-data-structures.ds

[5] GeeksforGeeks, *Difference between Linear and Non-linear Data Structures*, 22 April, 2020, https://www.geeksforgeeks.org/difference-between-linear-and-non-linear-data-structures/

[6] Revue de Métallurgie, *Using graph search algorithms for a rigorous application of emergy algebra rules,* Advances in Computer, Communication and Computational Sciences, Proceedings of IC4S 2019 (pp.417-428), January 2013.

[7] GeeksforGeeks, *Edit Distance | DP-5,* 13 January, 2020, https://www.geeksforgeeks.org/edit-distance-dp-5/

[8] Giuseppe Lancia, *Speeding-Up the Dynamic Programming Procedure for the Edit Distance of Two Strings,* Database and Expert Systems Applications (pp.59-66), August 2019.

,

[9] Ricardo Baeza-Yates, *Gaston H. Gonnet, A new approach to text searching, ACM SIGIR Forum 23(SI):168-175*, January 1988