CSC2103 Data Structures and Algorithms

2023 April Semester

Assignment 1

Prepared by:

Cheryl Toh Qiao Rou (20042206)

Yap Wei Xiang (21067939)

Sharifah Hannah Zahra (21066337)

Fong Jun Yang (21075304)

Due Date: 26 July 2023

Table of Contents

1.0 Introduction

1.1 Problem 1: Dijkstra Algorithm

The Dijkstra algorithm is a popular graph traversal algorithm that identifies the shortest path between two vertices in a graph. It operates by iteratively selecting the node with the minimum tentative distance and updating the distances of its adjacent nodes. This procedure persists until the shortest route to the target node has been determined. This assignment concentrates on implementing the Dijkstra algorithm in Java, where we will utilise specific data structures and tools such as IntelliJ, Visual Studio Code, and GitHub. Using these specific tools and data structures, the objective is to comprehend and implement the Dijkstra algorithm within the Java programming language efficiently.

1.1.1 Overview of the Problem

The problem is to implement Dijkstra's algorithm in Java for finding the shortest path between two vertices in a weighted graph. It selects nodes based on distances, updates distances for neighbours, and uses a priority queue and adjacency list for efficiency. The algorithm continues until the target node is reached, or all nodes are visited. The implementation includes input handling and displaying the shortest paths. The goal is to accurately implement Dijkstra's algorithm in Java using provided tools and data structures.

1.1.2 Choice of data structure

We chose a priority queue as the primary data structure for efficient node selection based on distances in Dijkstra's algorithm. A priority queue allows quick access to the element with the smallest distance, optimising the algorithm's runtime. It also supports dynamic updates, crucial for decreasing node distances when shorter paths are discovered.

For the graph representation, we selected an adjacency list. This representation efficiently stores the sparse graph structure, reducing memory usage. It provides fast access to a node's neighbours, facilitating exploration and distance updates. The adjacency list accurately represents the connectivity between locations in the map and enhances visualisation and simulation.

Dijkstra's algorithm has several benefits compared to other alternatives. It guarantees the best outcome by finding the shortest path from a source to all other nodes. It is adaptable, handling both weighted and unweighted graphs, making it useful in various applications. The algorithm's simplicity aids implementation, understanding, and educational purposes. Moreover, Dijkstra's algorithm is widely used and serves as a foundation for other graph algorithms, making it a valuable tool in graph theory.

In conclusion, the choice of a priority queue and an adjacency list as the data structures, along with Dijkstra's algorithm, ensures accurate and efficient visualisation and simulation of the shortest path problem on the given map.

1.1.3 Tools

To implement Dijkstra's Algorithm in a straightforward manner, we chose IntelliJ as our Integrated Development Environment on the recommendation of our professor. In addition, version control is utilised on GitHub to ensure that all codes are regularly updated. GitHub provides a potent collaborative platform for Java development, allowing for simple version control, seamless team collaboration, and extensive code sharing. It enables effective code review, bug monitoring, and integration with common development workflows. IntelliJ IDEA, a robust Java IDE, improves developer efficiency with intelligent code completion, sophisticated debugging, and refactoring tools. It provides seamless integration with Git and GitHub, which facilitates streamlined development and effective project management. Together, GitHub and IntelliJ provide a comprehensive ecosystem for Java programmers, enabling them to easily compose, manage, and collaborate on code.

1.1.4 Programming language feature used

1. **Priority Queue**: Java provides a built-in PriorityQueue class that can be used to efficiently select nodes with the minimum distance. This is useful for implementing the priority queue required in Dijkstra's algorithm.
2. **Comparator Interface**: Java's Comparator interface allows custom comparison logic to be defined for objects. In the provided code, the Node class implements the Comparator interface to compare nodes based on their respective costs. This is utilised in the priority queue to order nodes by their costs.
3. **Collections**: Java's Collections framework provides various data structures, such as lists and sets, which are used in the implementation. The adjacency list is represented as a List of Lists to store the graph structure. The visited set is implemented using Java's HashSet class.
4. **Input Handling**: Java's Scanner class is used to handle user input for selecting the source node. It allows reading input from the console and performing validation checks.
5. **Object-Oriented Programming**: Java's object-oriented programming features, such as classes and objects, are used to structure the code and encapsulate related functionalities. The Node class represents a node in the graph, and the ShortestPathFinding class encapsulates the implementation of Dijkstra's algorithm.

Overall, Java provides the necessary features and tools to implement Dijkstra's algorithm efficiently and maintain a clean, structured codebase.

1.2 Problem 2: Boyer-Moore String Matching

Boyer-Moore is a string-matching algorithm that searches for occurrences of a pattern within a text. It employs two potent heuristics, the Bad Character Rule and the Good Suffix Rule, to optimise the search process by skipping irrelevant portions of the text and achieving quicker search times. In this assignment, we will investigate the Boyer-Moore algorithm's implementation in Java, utilising specific data structures and tools such as IntelliJ and GitHub.

1.2.1 Overview of the Problem

The objective, given a text T and a pattern P, is to locate all instances of P within T. Boyer-Moore is a significant enhancement over simplistic string matching algorithms. It employs effective heuristics to direct the search procedure. By incorporating the Bad Character Rule and the Good Suffix Rule, the algorithm is able to intelligently bypass comparisons and reduce unnecessary shifts, resulting in quicker pattern matching. The following is an illustration of string matching:

```
T: {I,L,O,V,E,D,S,A}
P: {D,S,A}
```

I	L	O	V	E	D	S	A	!

1.2.2 Choice of data structure

To effectively implement the Boyer-Moore algorithm, we will employ data structures that facilitate effective search operations. A 2D array is an essential data structure that is used specifically for the Bad Character Rule. Using the alphabet size (|Σ|) as the number of rows and the pattern length (m) as the number of columns, the 2D array known as the bad character shift table is constructed. When a mismatch between a character in the text and the pattern occurs, each entry in the bad character shift table indicates the number of characters to omit in the

pattern. This enables us to efficiently align the pattern with the text by shifting it to the right when poor characters are encountered.

Additionally, for the Good Suffix Rule, we use a 1D array known as the Good Suffix shift table. The Good Suffix shift table stores precomputed shift values based on the longest border of each suffix of the pattern and the longest suffixes that match a suffix of the pattern. These values determine the optimal shift distances when encountering Good Suffixes during the search process.

The reason for using arrays to compute shift tables instead of other data structures like Hashmaps and LinkedList is that arrays provide efficient access to elements based on their corresponding indices. Arrays also offer a compact way of storing values in a contiguous block of memory, which helps in minimising storage usage. Another reason for using arrays is that arrays are fundamental and are supported in most programming languages, making the implementation accessible and compatible with different platforms and languages. Compared to the other data structures, arrays provide a balance between performance, simplicity and compatibility, making them the best choice for implementing shift tables in the Boyer-Moore algorithm.

1.2.3 Tools

For the implementation of the Boyer-Moore algorithm, we have chosen IntelliJ as the Integrated Development Environment (IDE) and GitHub for version control and collaborative development. IntelliJ provides a user-friendly and feature-rich environment for coding, debugging, and testing our Java implementation. With its intelligent code assistance and debugging capabilities, IntelliJ will aid in the development process, ensuring code correctness and efficiency. On the other hand, GitHub enables us to collaborate efficiently, manage project versions, and receive feedback from peers and instructors throughout the development lifecycle.

1.2.4 Programming language feature used

For the implementation of the Boyer-Moore algorithm, we have chosen Java as the programming language. Java offers several features and tools that align well with the requirements of the algorithm:

1. **Arrays**: Java provides robust support for arrays, allowing efficient storage and access to elements in contiguous memory locations. We will utilise arrays to represent the pattern, text, and various tables required by the algorithm. Specifically, arrays will be used to construct the shift tables for both the Bad Character Rule and the Good Suffix Rule.
2. **Object-Oriented Design**: Java's object-oriented nature enables us to encapsulate related functionalities within classes and methods. We can design classes to represent the Boyer-Moore algorithm, encapsulating the required data structures and algorithms for easy reuse and modularity. This approach promotes code organisation and enhances code readability.
3. **Generics**: Java's generics feature allows us to create flexible and type-safe data structures. We can leverage generics to create reusable data structures that can handle different types of patterns and texts. This flexibility enhances the versatility of the algorithm implementation.
4. **Standard Libraries**: Java provides a rich set of standard libraries that offer various utilities and functionalities. We can leverage these libraries to simplify tasks such as I/O operations, array manipulation, and string handling. These libraries can significantly aid in the efficient implementation of the Boyer-Moore algorithm.

By utilising these features and tools of the Java programming language, we can implement the Boyer-Moore algorithm efficiently and create a well-organised and readable codebase. The algorithm's reliance on appropriate data structures and the programming language features of Java ensure that it meets the criteria of correctness, efficiency, and code readability outlined in the assignment's marking rubric.

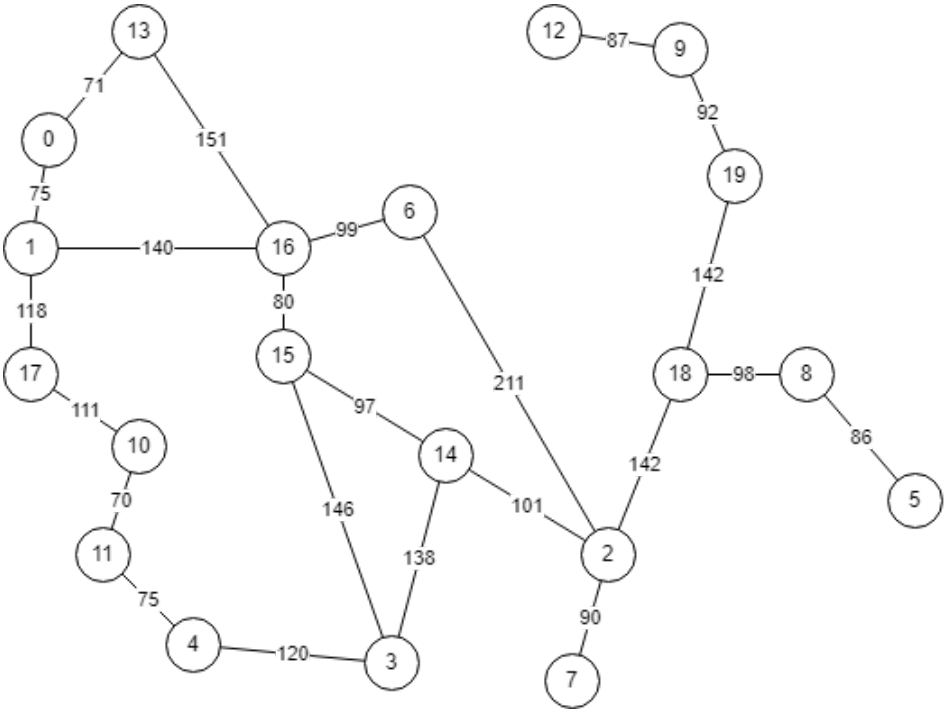
In summary, the Boyer-Moore algorithm is a powerful string-matching technique that uses the Bad Character Rule and the Good Suffix Rule.

2.0 Implementation

2.1 Problem 1: Dijkstra Algorithm

2.1.1 Explanation

The algorithm that we selected for our shortest path problem is Dijkstra's algorithm. We implemented this algorithm fully in Java, utilising a comparator interface, and representing the map below with adjacency lists.



Undirected weighted graph, inspired by the Romania Problem of A.I. Nodes are labelled with a number and connected with edges. The value that lies on the edges signifies the path cost between two nodes.

Our implementation has 3 classes: a *Driver* class for the application, a *Node* class, and finally our *ShortestPathFinding* class.

Class 1: *Node*

```
class Node implements Comparator<Node>{
public int node;
public int cost;

public Node(){} // Empty Constructor

public Node(int node, int cost){
this.node = node;
this.cost = cost;
}
public int compare(Node node1, Node node2){ return
Integer.compare(node1.cost,node2.cost);}
}
```

Our *Node* class represents an actual node of a graph. It contains two instance variables:

1. *node*: an integer value that represents the node identifier.
2. *cost*: an integer value that represents the cost associated with the node.

The *Node* class also implements the Comparator interface, which allows different instances of the *Node* class to be compared based on their respective costs. We use this later on in our program for our *priority queue*.

Method: *compare*

The *compare* method is a simplified version of the code below.

```
public int compare(Node node1, Node node2){
if (node1.cost < node2.cost)
return -1;
if (node1.cost > node2.cost)
return 1;
return 0;
}
```

Essentially, the method takes in two node objects: *node1* and *node2* and returns either −1, 1, or 0. In the event that *node1* has a lower cost than *node2*, the method returns -1. When the opposite is true, the method returns 1. Finally, if both nodes have equal costs, the method returns 0.

Class 2: *ShortestPathFinding*

This class is responsible for the actual implementation of Dijkstra's algorithm.

```
class ShortestPathFinding{

int[] dist,pred;
Set<Integer> visited;
PriorityQueue<Node> pqueue;
int V; // Number of Vertices
List<List<Node>> adj_list;

// class constructor
ShortestPathFinding(int V){
this.V = V;
dist = new int[V];
pred = new int[V];
visited = new HashSet<>();
pqueue = new PriorityQueue<>(V,new Node());
}
```

This class contains five instance variables:

1. *dist*: an array of integers representing the distances from the source node to each node.
2. *pred*: an array of integers representing the predecessor to a node.
3. *visited*: A set of integers to keep track of visited nodes throughout the algorithm.
4. *pqueue*: A priority queue of `Node` objects.
5. *V*: An integer representing the number of vertices (or nodes) in the graph.
6. *adj_list*: A 2D list representing the adjacency list of the graph.

Method 1: *Dijkstra*

```
for (int i = 0; i < V; i++) {
dist[i] = Integer.MAX_VALUE;
pred[i] = -1;
}

// Add Source Vertex into Priority Queue
pqueue.add(new Node(src,0));
// Distance from source to source is 0
dist[src] = 0;
```

This for loop will loop through the length of the *dist* array and change all the values to "infinity". It also changes all the values in the *pred* array to -1 (This will come in handy later in our *printPath* method). We later change the distance of the source node to 0. We then add the source node to the priority queue with a distance cost of 0.

```
while (visited.size() != V){ // While All Nodes are not Visited,
int u = pqueue.remove().node; // Put node with minimum distance into the visited
set
visited.add(u);
graph_adjacentNodes(u);

}
```

The algorithm continues until all the nodes have been visited. In each iteration, the node with the minimum cost is removed from the priority queue and put into the visited set. We then call the *graph_adjacentNodes* function for the current node.

```
System.out.println("Source\t" + "Node#\t\t" + "Distance\t\t" + "Path");
for (int i = 0; i < V; i++){
System.out.print("\t" + src + "\t\t" + i + "\t\t" + dist[i] + "\t\t\t\t");
printPath(i);
System.out.println();
}
```

After the algorithm finishes, the distances from the source node to all the nodes are printed, as well as the paths to the nodes from the selected source node.

Method 2: *graph_adjacentNodes*

```
for (int i = 0; i < adj_list.get(u).size(); i++){

Node v = adj_list.get(u).get(i);

// Proceed Only if Current Node is Not in Visited
if (!visited.contains(v.node)){
edgeDistance = v.cost;
newDistance = dist[u] + edgeDistance;

// Compare Distances
if (newDistance < dist[v.node]){
dist[v.node] = newDistance;
pred[v.node] = u;
}
pqueue.add(new Node(v.node, dist[v.node]));
}
}
```

For each neighbouring node *v*, it checks if *v* has not been visited. If that is the case, it calculates the new distances from the source to *v* through *u*. If the new distance calculated is smaller than the existing distance in the *dist* array, we replace the one in the array with our new distance, and update our *pred* array for *v* to be *u*.

Method 3: printPath

```
for (int i = 0; i < adj_list.get(u).size(); i++){

Node v = adj_list.get(u).get(i);

// Proceed Only if Current Node is Not in Visited
if (!visited.contains(v.node)){
edgeDistance = v.cost;
newDistance = dist[u] + edgeDistance;

// Compare Distances
if (newDistance < dist[v.node]){
dist[v.node] = newDistance;
pred[v.node] = u;
}
pqueue.add(new Node(v.node, dist[v.node]));
}
}
private void printPath(int node){
if (node == -1)
return;

printPath(pred[node]);
System.out.print(node+" ");
}
```

This method is a recursive function that prints the paths of the graph. The method first checks whether the node is -1. It then calls itself to find the predecessor to the current node. The method then prints the node value followed by a space character. This step prints the current node as part of the path.

Class 3: ***Driver***

Method: ***main***

The main method can effectively be split into two parts: **Graph Description** and **Algorithm Execution**.

Graph Description

As mentioned before, we describe our graphs with adjacency lists. Since our example has 20 nodes in total, the description of the graph took quite a bit of time.

```
// Describe Graph
adj_list.get(0).add(new Node(1,75));
adj_list.get(1).add(new Node(0,75));
adj_list.get(0).add(new Node(13,71));
adj_list.get(13).add(new Node(1,71));
adj_list.get(1).add(new Node(16,140));
adj_list.get(16).add(new Node(1,140));
adj_list.get(1).add(new Node(17,118));
adj_list.get(17).add(new Node(1,118));
adj_list.get(2).add(new Node(6, 211));
adj_list.get(6).add(new Node(2,211));
adj_list.get(2).add(new Node(7,90));
```

A small snippet of the code used to describe the graph.

Essentially, we add nodes and their respective distance costs to the adjacency lists of other nodes. That is how we describe our graph.

Algorithm Execution

```
// Call Algorithm
ShortestPathFinding algo = new ShortestPathFinding(V);
Scanner input = new Scanner(System.in);
System.out.println("Which node should be the source node?");
source = input.nextInt();
while (source < 0 || source > 19){
System.out.println("That is beyond the range of our graph!\nWhich node should be
the source node?");
source = input.nextInt();
}
algo.dijkstra(adj_list,source);
```

An instance of the *ShortestPathFinding* class is created, and the user is prompted to enter the source node for which the shortest paths should be calculated. The algorithm is then executed.

In summary, the *Driver* class initialises the graph by creating and populating the adjacency list, prompts the user to enter the source node, and then executes Dijkstra's algorithm to find the shortest paths using the *ShortestPathFinding* class.

Console Output:

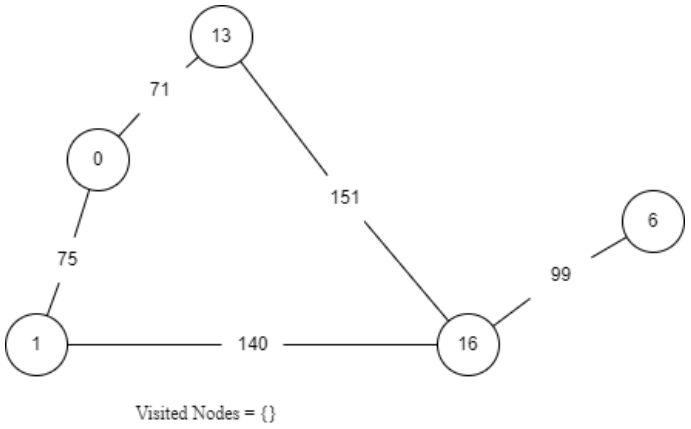
```
Which node should be the source node?
0
Source     Node#      Distance        Path
     0          0          0          0
     0          1          75         0 1
     0          2          493        0 1 16 15 14 2
     0          3          441        0 1 16 15 3
     0          4          449        0 1 17 10 11 4
     0          5          819        0 1 16 15 14 2 18 8 5
     0          6          314        0 1 16 6
     0          7          583        0 1 16 15 14 2 7
     0          8          733        0 1 16 15 14 2 18 8
     0          9          869        0 1 16 15 14 2 18 19 9
     0          10         304        0 1 17 10
     0          11         374        0 1 17 10 11
     0          12         956        0 1 16 15 14 2 18 19 9 12
     0          13         71         0 13
     0          14         392        0 1 16 15 14
     0          15         295        0 1 16 15
     0          16         215        0 1 16
     0          17         193        0 1 17
     0          18         635        0 1 16 15 14 2 18
     0          19         777        0 1 16 15 14 2 18 19
```

Output when source node is 0.

Output Breakdown:

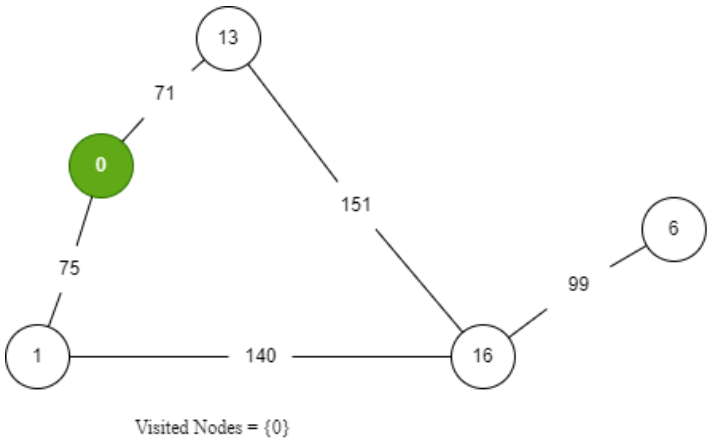
For the purposes of this breakdown, we will use a subgraph that includes nodes = *{0,1,6,13,16}*

Initial State



None of the nodes have been visited yet, in this state, we have to first pick our source node. The list at the bottom of the diagram shows all the unvisited nodes.

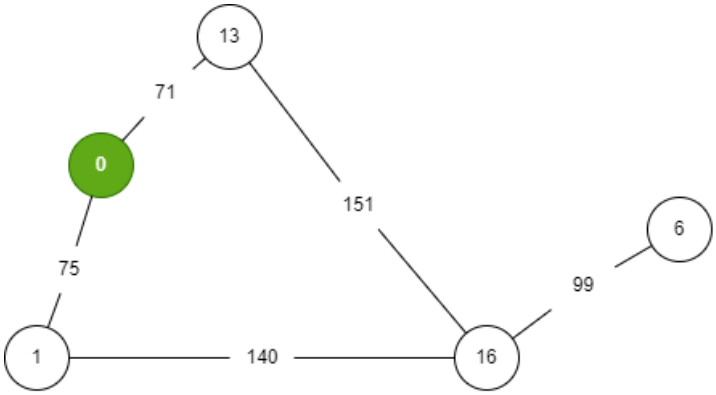
The Algorithm



Node	Distance
0	0
1	∞
6	∞
13	∞
16	∞

We choose *0* to be our source node. The table on the right is used to keep track of distances. To move on from this step, we have to examine the adjacent nodes to *0*. In this case, we have two adjacent nodes – *1* and *13*.

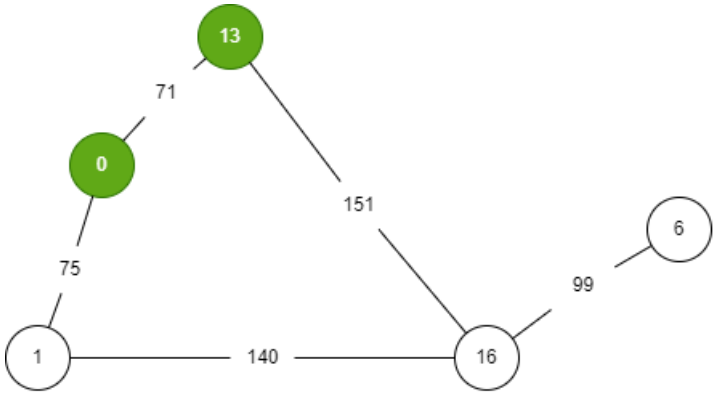
We now have access to the two nodes, and we can add it to our distance table.



Visited Nodes = {0}

Node	Distance
0	0
1	75
6	∞
13	71
16	∞

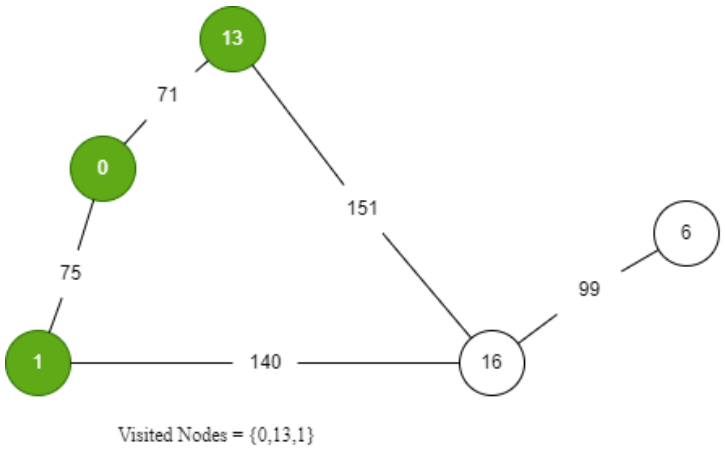
Now, we expand towards the node that has the shortest distance. In this case, it's node 13.



Visited Nodes = {0,13}

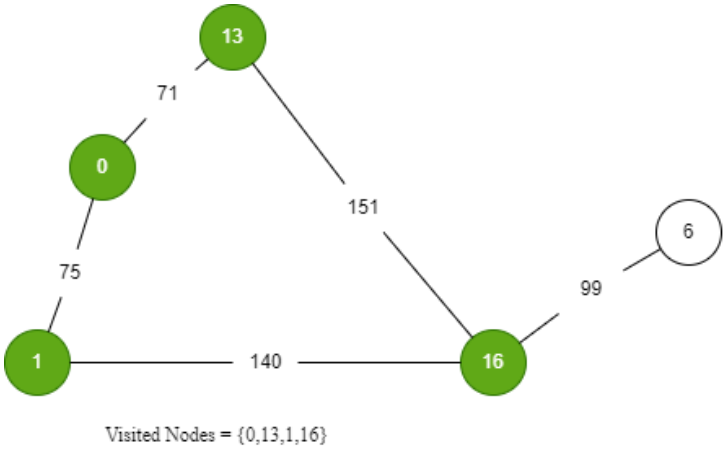
Node	Distance
0	0
1	75
6	∞
13	71
16	222

Here, we see that the distance value on our table for node *16* is also updated. Because we now have access to that node. The number 222 comes from adding the path costs in order to reach node *16* from node *0*. From here, we choose our next node by looking at all the nodes that are not in the visited set, and selecting the one with the shortest distance from the source node. Right now, *1*, *6*, and *16* are not in our visited set, but among the three, node *1* has the shortest distance. We, therefore, decide to visit node *1*.



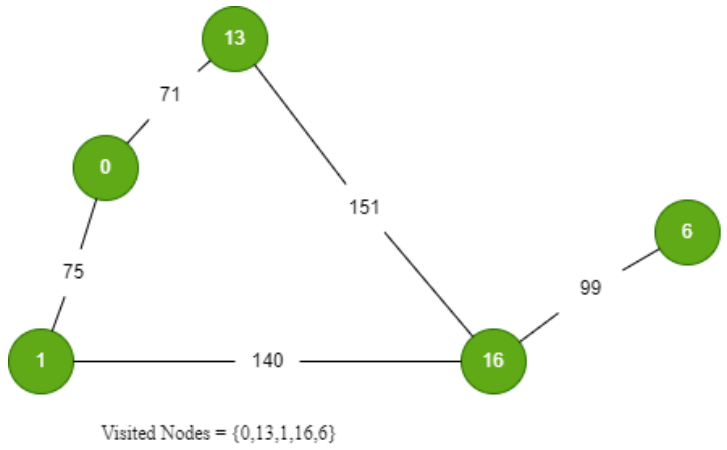
Node	Distance
0	0
1	75
6	∞
13	71
16	**215**

After visiting *1*, we see that *16* is also adjacent to *1*. We then calculate the cost it would take to reach *16* from *0* through *1* and compare the distance calculated with the distance in the table. We see that we have a shorter distance, and so we replace the distance in node *16* with the new value – 215. Now, the only node that we can visit is 16.



Node	Distance
0	0
1	75
6	314
13	71
16	215

Node 16 is visited. Now the only node that has not been visited is node 6.



Node	Distance
0	0
1	75
6	314
13	71
16	215

All the nodes have been visited. The algorithm is complete.

2.1.2 Best and worst case

Best case scenario:

For Dijkstra' Algorithm, the best case scenario occurs when the graph has a sparse structure, and the shortest path between the source node and all other nodes is relatively short compared to the total number of edges in the graph.

Worst case scenario:

The worst case scenario occurs when the graph is dense and negative edge weights exist. In this case, the algorithm's performance degrades significantly, and it may not produce correct results or may enter an infinite loop.

If we assume that a graph has no negative edge weights, both scenarios share a similar time complexity. As shown below,

$$O((V + E) \log V$$

$$V\text{: } Number\ of\ Vertices$$

$$E\text{: } Number\ of\ Edges$$

2.1.3 Limitations of our Solution

1. Limited to Positive Edge Weights

   Dijkstra's algorithm is designed to deal with only positive edge weights. If a graph has negative edge weight, the algorithm may not produce correct results and might enter into an infinite loop.

2. Single Source Shortest Path

   Unlike Prim's or Kruskal's algorithm which builds a minimum spanning tree from any graph, Dijkstra's algorithm finds the shortest path for each node for only one source node at a time, which takes up more processing. This is a double-edged sword, because although one could argue that a minimum spanning tree algorithm would be more efficient, Dijkstra's algorithm is guaranteed to **always** give us the shortest path.

3. Inefficient for Dense Graphs

   As mentioned before, our implementation uses adjacency lists to represent the graph, which is good enough for our implementation. However, in the event of an extremely dense graph, putting in each node and their neighbouring nodes would be highly inefficient.

2.1.4 Future Improvement

1. Explore Parallelism Techniques:
   Implementing parallelism would involve executing certain parts of the code concurrently. This can potentially speed up the execution of the algorithm.

2. Better Graph Representation:
   Choosing a more efficient way of representing the graph would be beneficial. Utilising adjacency matrices, for example, could be taken into consideration.

3. Implement Graph Input from File:
   Being able to read a graph from a file would make the programme more diverse, and more users with little to no programming experience would be able to run the programme.

2.2 Problem 2: Boyer-Moore String Matching

2.2.1 Explanation

Our team worked on implementing the full Boyer-Moore string matching algorithm in Java, utilising both Bad Character Rule and Good Suffix Rule. In our implementation, we created 2 classes. One for the preprocessing and search function and one for demonstration of the algorithm. See methods defined in the algorithm class:

Method	**Explanation**
badCharacterShiftTable	To preprocess patterns and compute shift tables for bad character heuristic
goodSuffixShiftTable	To preprocess pattern and compute shift table for Good Suffix heuristic
search	To implement the full Boyer-Moore algorithm, searching occurrence of a pattern in a given text

The next section will be explaining each code snippet in detail.

Class 1: *BoyerMooreAlgorithm* Class

Method 1: *badCharacterShiftTable*

```
public int[] badCharacterShiftTable(char[] pattern){

//initialise shift table and pattern length
int[] shiftTable = new int[256];
int m = pattern.length;
```

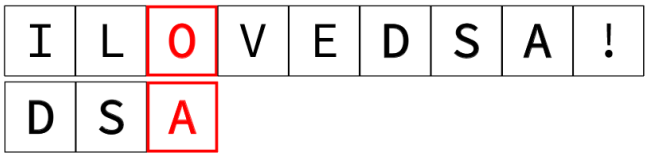
This method takes in a list of characters as a pattern to be preprocessed. We modified the shift table computation explained in *clause 1.2.2* from using a 2D array to a 1D array as a shift table for our Bad Character Rule. By using a 1D array instead of a 2D array, it will highly reduce the storage complexity and speed up the search process. Therefore, in the code snippet above, we have initialised a 1D array with the size of ASCII characters for the shift table and m as the number of characters in the pattern.

```
//initialise shift table values to the pattern length
for (int i = 0; i < 256; i++) {
shiftTable[i] = m;
}
```

Next, the for loop above iterates through the shift table and sets all values in the shift table as the number of characters in the pattern. This assumes that if the character in the shift table does not match the pattern, the pattern can be shifted by full length in case of a mismatch.

```
//update shift table with correct shift value
for (int i = 0; i < m - 1; i++) {
char c = pattern[i];
shiftTable[c] = m - i - 1;
}
```

Then, a for loop is called to iterate over the characters in the pattern. In each iteration of the for loop, the character at index *i* is retrieved as *c*. The shift value for character *c* is calculated by subtracting the index value from the pattern length (*m - i - 1*). This represents the number of characters to shift when a mismatch occurs at character *c*. After the shift value is obtained, the shift table at index *c* is updated with the shift value. See below a visualisation of a scenario:




```
i: 2
m = 3
m - i - 1 = 0
shiftTable:
```

Character	shift value
A	0

```
return shiftTable;}
```

After computing all shift values, the constructed shift table is returned.

Method 2: *goodSuffixShiftTable*

```
public int[] goodSuffixShiftTable(char[] pattern){

//initialise tables and pattern length
int m = pattern.length;
int[] borderTable = new int[m+1];
int[] shiftTable = new int[m+1];

//initialise values in the shift table to be 0
for(int i=0; i<m + 1; i++){
shiftTable[i] = 0;
```

This method takes in a list of characters as a pattern to be preprocessed. We initialised one integer array for the border table, another integer array for the shift table, and the number of characters in the pattern as m. Both arrays have the size of *m+1*. Then, every index of the shift table is set to a value of *0* as the default value. A border table is a data structure to store information about borders in a pattern. By searching the border table, we can easily know the starting position of the longest borders that end at each index in a pattern.

To continue from here, it is important to know that preprocessing of a Good Suffix shift table requires 2 processing rounds to fill out the shift table. We will discuss each preprocessing round in detail in the following section of this document.

Preprocessing round 1

```
int i = m;
int j = m + 1;

borderTable[i] = j;
```

To start our first preprocessing round, we initialised variables *i* and *j* with values *m* and *m+1* respectively, where *m* is the length of the pattern. Then, the last character of the pattern will have no known border, so we set the value of the last index of the border table to *j*.

```
while(i > 0){
while(j <= m && pattern[i - 1] != pattern[j - 1]){

//if a mismatch is found, stop expanding the border and update the shift table
if (shiftTable[j] == 0) {
shiftTable[j] = j - i;
}

//Update the position of the next border
j = borderTable[j];
}
```

Now, we are in the border expansion phase. A while loop is started to search for borders in the pattern by expanding from right to left. While *i* is greater than 0, check if the character in the pattern with index *i-1* matches the character in the pattern with index *j-1*. If they do not match and *j* is less than the length of the pattern, we will need to continue searching for a border. Therefore, we enter the inner loop of the above code snippet. In the inner loop, if the value of the shift table at index *j* is 0, it means that this position has not yet been encountered before. So, we update its value to *j-i*, representing the distance between *j* and *i*. Then, *j* is set to the value of the border table at index *j* to update the position of the next border, allowing a search to the left of the current position for a border.

```
i--;
j--;
borderTable[i] = j;
}
```

Still inside the outer while loop, if the condition of the inner is not matched, means a border is found. The beginning position of the found border will be stored in the border table at index *i*. Then, both *i* and *j* will be decremented to prepare for the next iteration, where we search for the borders of the next character in the pattern. The loop continues until borders are found for all characters in the pattern.

After this processing round, there are some cases where the shift table might not be fully filled out with shift values even if we have found all borders for each character in the pattern. Therefore, a second round of preprocessing is required.

Preprocessing round 2

```
i--;
j--;
borderTable[i] = j;
}
```

Similar to the previous preprocessing round, this round starts by initialising variables *x* and *y* that will be used in the iteration of this preprocessing round. We set the initial value of *y* to the value stored in the first index of the border table, pointing to the widest border of the pattern.

```
for(x = 0; x <= m; x++) {
//if the shift table has any value that has not been filled, update with the widest
border value
if(shiftTable[x] == 0)
shiftTable[x] = y;

if (x == y)
y = borderTable[y];
}
```

After initialization, a for loop is started from $x = 0$ to $x <= m$, where m is the length of the pattern. In each iteration round, the algorithm checks if the value of the shift table at index x is filled out or not. If the value is 0 means the position is not filled out yet. Therefore, it will be set to the value of y, the widest border of the pattern. Then, the algorithm checks if x equals y. If yes, it means that the length of the suffix (substring) x is already smaller than the widest border y. In that case, y will be updated to become the next widest border of the pattern, setting it to the value of the border table at index y. Iteration continues until all indexes of the shift table are filled out. When the algorithm jumps out of the for loop means the shift table of the Good Suffix Rule has been fully constructed.

```
return shiftTable;
}
```

After the 2 rounds of preprocessing of the pattern, the constructed shift table is returned.

Method 3: *search*

```
public void search(char[] text, char[] pattern){

//initialise shift values and length of pattern and text
int s = 0;
int j;
int n = text.length;
int m = pattern.length;
```

This method takes in 2 char arrays, one that stores characters in the text and the other that stores characters in a pattern that needs to be matched in the text. Before the search starts, we first initialised *s* as the current position and *j* which we will be using in the search algorithm. We also initialised *n* and *m* for the length of text and length of the pattern respectively.

```
//prepublic void search(char[] text, char[] pattern){

//initialise shift values and length of pattern and text
int s = 0;
int j;
int n = text.length;
int m = pattern.length;
```

Then, the pattern is preprocessed for both Bad Character Rule and Good Suffix Rule by calling the *badCharacterShiftTable* method and *goodSuffixShiftTable* method. The returned shift tables for both heuristics are stored in their respective array, namely, *badCharacterShift* and *goodSuffixShift*.

```
//start search
while(s <= n-m) {
j = m - 1;

//if the pattern character matches the text character, decrement j until a mismatch
is found or the entire pattern matches
while (j >= 0 && pattern[j] == text[s + j]) {
j--;
}
```

The search phase begins by starting a while loop that iterates through the text. While ensuring the shift value will always be lesser than or equal to the remaining length of the text that has not yet been explored or matched. In the loop, variable *j* is set to *m-1*, the right-most character of the pattern, as the Boyer-Moore algorithm will always search from right to left. In the inner while loop, *j--* indicates that the search iterates through all characters in the pattern, matching each of the characters to the corresponding character in the text. If a mismatch is found or if the entire pattern matches the substring in the text, the algorithm breaks out of the inner loop.

```
//if the entire pattern matches, print shift value
if (j < 0) {
System.out.println("pattern occurs at shift " + s);
//shift the entire pattern to the right to continue searching
s += goodSuffixShift[0];
}
```

After breaking out the inner while loop, the algorithm checks if the value of *j* is lesser than 0. If yes, this means that all the characters in the pattern match their corresponding substring in the text. Therefore, the algorithm prints out a message to notify users that a match of the pattern in the text occurred. Then, the pattern will be shifted entirely to the right by using *goodSuffixShift[0]* to continue searching for further occurrences of the pattern in the text.

```
else {
//if a mismatch is found, compute shift values for both heuristics
int badCharShiftValue = badCharacterShift[text[s+j]-m+1+j];
int goodSuffixShiftValue = goodSuffixShift[j+1];

//use shift value with the largest shift
s += Math.max(badCharShiftValue, goodSuffixShiftValue);
}
```

If *j* is not lesser than 0, this indicates that a mismatch is found at index *j* of the pattern. Therefore, the algorithm will look up for shift value at the said index in the shift tables of both heuristics. The looked-up shift value is then stored in their corresponding variable. Then, the maximum of the two shift values will be chosen to be the final shift value of this iteration. This value is then added into s, moving the pattern to a new position in the text to search for further potential occurrences. This will ensure the pattern shifts to the right by the maximum amount possible. Iteration continues until either the entire text is searched or all occurrences of the pattern are found.

Class 2: *BoyerMooreExample* Class

Method 1: *Main*

```
public class BoyerMooreExample {
public static void main(String[] args) {

//initialise text and pattern
char[] text = "ABAAAABAACD".toCharArray();
char[] pattern = "ABA".toCharArray();

//print information
System.out.println("Text: " + Arrays.toString(text));
System.out.println("Pattern: " + Arrays.toString(pattern));
System.out.println();

//start search
BoyerMooreAlgorithm BoyerMoore = new BoyerMooreAlgorithm();
BoyerMoore.search(text, pattern);
}}
```

In our demonstration, we used “ABAAABAACD” as the text and “ABA” as the pattern to be matched using the Boyer-Moore algorithm. Both strings are stored as char arrays. Then, the text and pattern array are printed out. After that, we initialised our Boyer-Moore algorithm by creating an instance of the *BoyerMooreAlgorithm* class. We then call the search method of the newly instantiated Boyer-Moore algorithm and parse in our text and pattern as parameters of the search method.

Console Output:

```
Text: [A, B, A, A, A, A, B, A, A, C, D]
Pattern: [A, B, A]

pattern occurs at shift 0
pattern occurs at shift 5
```

The output when pattern is “ABA” and text is “ABAAAABAACD”.

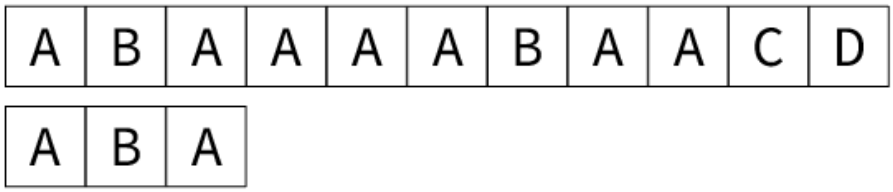
To understand how the string matching algorithm works, the next section shows the detailed breakdown of steps in order to obtain the output as shown above.

Output breakdown:

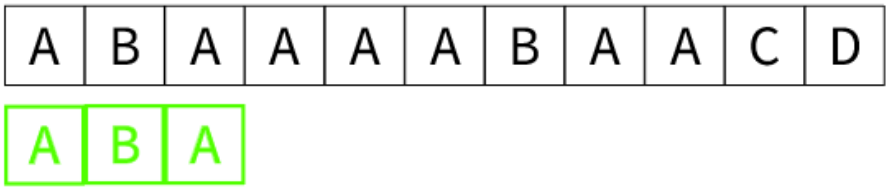
Initial values:

```
s = 0
m = 3
n = 11
badCharacterShift = [3, 3, 3, ... 3, 2, 1, 3, ..., 3]
goodSuffixShift = [2, 2, 2, 1]
```

Iteration 1:



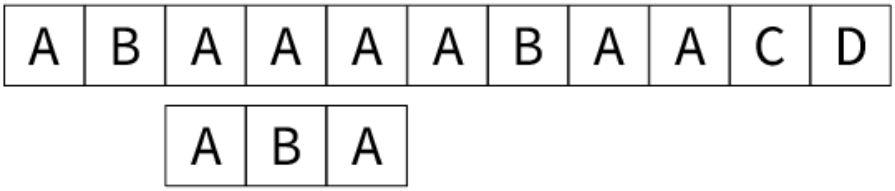
Initially, the pattern will be placed at the start of the text (*s = 0*). The algorithm iterates through the pattern from right to left, matching each character to the corresponding character in text.



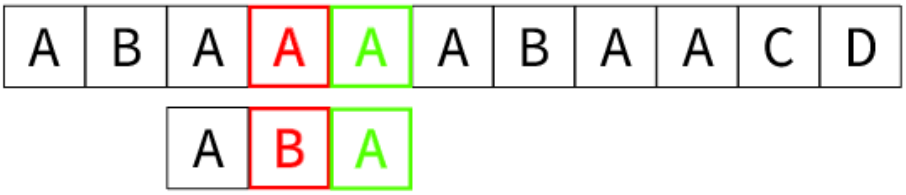

```
Good suffix shift value: 2
s: 0 + 2 = 2
```

At *shift[0]*, the pattern matches the text entirely. A message is printed to notify that a match is found. The algorithm then computes a shift value according to the Good Suffix Rule and updates the variable *s*.

Iteration 2:



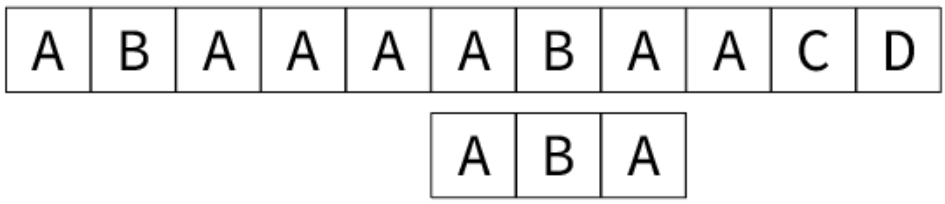
The pattern is shifted to position 2 (*s = 2*). The algorithm iterates through the pattern from right to left, matching each character to the corresponding character in text.



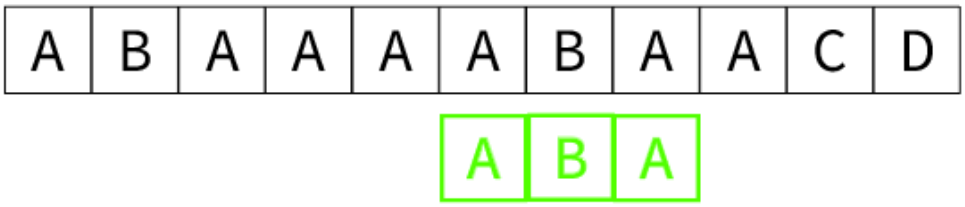

```
Bad Character Shift Value: 3
Good suffix shift value: 2
s: 2 + 3 = 5
```

At *shift[2]*, the pattern mismatched at character *A* of the text. The algorithm then computes a shift value of both rules and chooses the larger shift value to update the variable *s*.

Iteration 3:



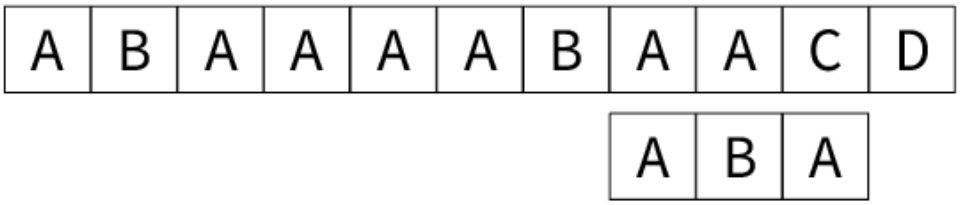
The pattern is shifted to position 5 (*s = 5*). The algorithm iterates through the pattern from right to left, matching each character to the corresponding character in text.



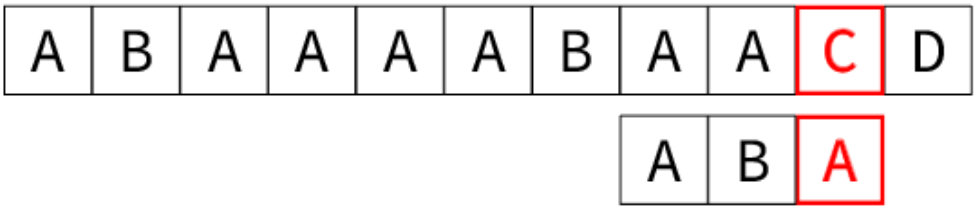

```
Good suffix shift value: 2
s: 5 + 2 = 7
```

At *shift[5]*, the pattern matches the text entirely. A message is printed to notify that a match is found. The algorithm then computes a shift value according to the Good Suffix Rule and updates the variable *s*.

Iteration 4:

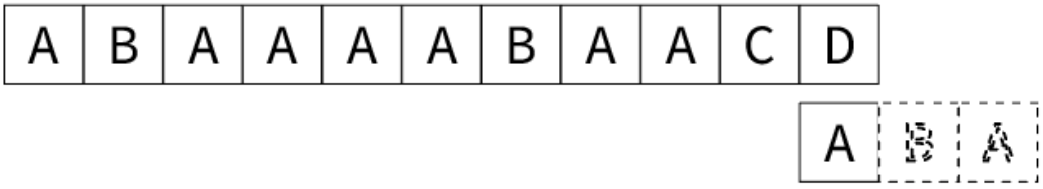


The pattern is shifted to position 7 (*s = 7*). The algorithm iterates through the pattern from right to left, matching each character to the corresponding character in text.




```
Bad Character Shift Value: 3
Good suffix shift value: 1
s: 7 + 3 = 10
```

At *shift[7]*, the pattern mismatched the character *C* of the text. The algorithm then computes a shift value of both rules and chooses the larger shift value to update the variable *s*.



If the pattern is further shifted to position 10 (*s = 10*), the pattern will shift past the size limit of the text. Therefore, iteration will stop, and the algorithm is then terminated.

2.2.2 Best and worst case scenario

Best case scenario:

In our implementation of the Boyer-Moore algorithm, the best case scenario occurs when there are no mismatches between the pattern and text. The time complexity of the best case scenario is shown below:

$$O(n/m)$$

n: $text\ length$

m: $pattern\ length$

Worst case scenario:

In our implementation of the Boyer-Moore algorithm, the worst case scenario occurs when a high number of full comparisons of the pattern to the text is required due to mismatch at every offset in the text. The time complexity of the worst case scenario is shown below:

$$O(nm)$$

n: $text\ length$

m: $pattern\ length$

2.2.3 Limitations of our solution

1. Less efficient when dealing with small alphabet size patterns:

   The Boyer-Moore technique might be less effective when dealing with patterns when the alphabet is smaller than the text. This may be because there aren't many ways to skip under the bad character criteria, which is based on the size of the alphabet.

2. Preprocessing overhead:

   Our Boyer-Moore algorithm needs additional preprocessing operations to produce the bad character shift table and the Good Suffix shift table. For tiny patterns, this preprocessing step can increase overhead. If the pattern size is minimal, the costs associated with creating and using shift tables could outweigh the advantages of the algorithm.

3. Memory requirement:

   Our implementation of the Boyer-Moore algorithm may demand more memory than other string matching algorithms. This is due to the fact that it must save the shift tables for both the Bad Character Rule and the Good Suffix Rule. The size of the shift tables is determined by the size of the alphabet and the length of the pattern, which might result in increased memory use.

2.2.4 Future Improvement

1. Incorporating additional heuristics:

   While the bad character heuristic and Good Suffix heuristic are already strong heuristics built into our Boyer-Moore method, adding other heuristics like the Galil-Giancarlo optimisation can help the system perform even better.

2. Optimise for handling specific patterns or text characteristics:

   Our implementation can be optimised further to better manage patterns and texts with specific characteristics. If a pattern contains numerous repetitions, for instance, the algorithm can be optimised to include run-length encoding or pattern compression to reduce the number of comparisons, thereby enhancing the search performance.

3. Explore parallelization techniques:

   By incorporating parallelization techniques, the search for large texts or patterns can be sped up. Parallelization is achieved by distributing tasks across multiple threads or processors, thereby decreasing the total time complexity.

3.0 Individual member contribution

Name	Student ID	Contribution
Cheryl Toh Qiao Rou (Group Leader)	20042206	- Arranging project timeline - Implementation of Boyer-Moore algorithm - Documentation of implementation explanation
Yap Wei Xiang	21067939	- Implementation of Dijkstra's algorithm - Documentation of implementation explanation, best and worst case scenarios, limitations and future improvements.
Sharifah Hannah Zahra	21066337	- Documentation of Boyer-Moore algorithm's Introduction, best and worst case scenarios and limitations - Proofread and formatting of documentation
Fong Jun Yang	21075304	- Documentation of Dijkstra's algorithm's introduction.

4.0 Challenges and solution

1. Delayed starting of implementation
    a. Time constraint
    b. Amount of piling up assignments
    c. Solution: clear timeline, online meeting to complete documentation
2. False committing of GitHub
    a. Code has been overridden and lost
    b. Unfamiliar use of GitHub
    c. Solution: Creating new branches for each algorithm and merging everything once everyone is done implementing both algorithms

5.0 Reference

Boyer Moore algorithm with Bad Character Heuristic. Top Website Designers, Developers, Freelancers for Your Next Project. (n.d.). https://www.topcoder.com/thrive/articles/boyer-moore-algorithm-with-bad-character-heuristic

Boyer-Moore - Department of Computer Science. (n.d.-a). https://www.cs.jhu.edu/~langmea/resources/lecture_notes/boyer_moore.pdf

GeeksforGeeks. (2019, October 31). *Boyer Moore Algorithm: Good suffix heuristic*. GeeksforGeeks. https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic/

GeeksforGeeks. (2023, June 1). *Dijkstra's algorithm for adjacency list representation: Greedy Algo-8*. GeeksforGeeks. https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/?ref=lbp

Sambol, M. (2014, September 15). *Dijkstra's algorithm in 3 minutes*. YouTube. https://www.youtube.com/watch?v=_lHSawdgXpI&

String algorithms and Data Structures boyer-moore. (n.d.-b). https://courses.engr.illinois.edu/cs225/fa2022/assets/honors/slides/cs199fa22-4-bmoore-slides.pdf