COMP1002

Data Structures & Algorithms

Report

Authors:

Naseh Rizvi 20167671

Contents

1 Introduction **1**

2 User Guide **1**
2.1 Installation . . . 1
2.2 Terminology/Abbreviations . . . 1
2.3 Walkthrough . . . 1
2.3.1 Interactive Mode . . . 2
2.3.2 Silent Mode . . . 3
2.4 Future Work . . . 4

3 Traceability Matrix **5**

4 UML CLass Diagram **6**

5 Classes **7**
5.1 keyMeUp . . . 7
5.2 Modes . . . 7
5.3 DSAGraph . . . 7
5.3.1 DSAGraphVertex . . . 7
5.3.2 DSAGraphEdge . . . 7
5.3.3 DijkstrasStack . . . 8
5.4 DSALinkedList . . . 8
5.4.1 DSAListNode . . . 8
5.4.2 DSALinkedListIterator . . . 8
5.5 DSAQueue . . . 8
5.6 DSAStack . . . 9
5.7 FileIO . . . 9
5.8 UserInputs . . . 9

6 Justification of Decisions **10**
6.1 Generics . . . 10
6.2 Representing Keyboard . . . 10
6.2.1 Adjacency Matrix . . . 10
6.2.2 Adjacency List . . . 10
6.2.3 Edge List . . . 10
6.3 Data Storage . . . 10
6.3.1 Abstract Data Types . . . 10
6.3.2 Storing the paths . . . 11
6.3.3 Serialisation . . . 11
6.4 File reading . . . 11
6.5 Path Finding Algorithms . . . 11

7 Results **13**
7.1 Introduction . . . 13
7.2 Scenario 1: Impact of Wrapping . . . 13
7.3 Scenario 2: Sting Cases . . . 13

8 References **14**

1 Introduction

This program is designed to analyse the shortest path to complete a string. This is achieved by implementing Dijkstra's algorithm of the shortest path. The program will read a keyboard file in the format of adjacency list and perform Dijkstra's algorithm on the strings.

2 User Guide

2.1 Installation

The program `keyMeUp`, is installed by compiling the files under the `src` directory, it has very few dependencies.

1. To Run the program in interactive mode:
   (a) make && java keyMeUp -i
   (b) if a message is printed saying "make: Nothing to be done for 'default'." try running "make clean && make && java keyMeUp -i"
2. To Run the program in silent mode:
   (a) make && java keyMeUp -s <file 1> <file 2> <file 3>
   (b) if a message is printed saying "make: Nothing to be done for 'default'." try running "make clean && make && java keyMeUp -s <file 1> <file 2> <file 3>"
   (c) file1: is the keyboards layout file.
   (d) file2: contains the string(s) that the program would search for.
   (e) file3: saves the output to the file.
3. To Run the test cases:
   (a) make test && java <test file>
   (b) Where <test file> can be: UnitTestLinkedList, UnitTestLinkedListIterator, UnitTestStack, UnitTestQueue, UnitTestGraph

2.2 Terminology/Abbreviations

The report will contain words that mean something completely different when it is taken outside the scope of the program.

- ADT: Abbreviation of Abstract Data Types, are objects where a set of value and operations determine its behaviour. Common examples are: Stack, Queue, Linked List and many more.
- Graph: Refers to the ADT that stores nodes and pairs of links between vertices.
- Node: A single unit in a data structure such as a graph or linked list which contain data and links to other nodes
- Vertex: A node in a graph
- Edge: A directed or undirected link between two vertices in a graph
- Path: A series of connections across multiple vertices in a graph

2.3 Walkthrough

All the functionality of the code is working,

2.3.1 Interactive Mode

As shown in figure 1, the program has a menu when first run in interactive mode. The menu can then be used to select different function all of the functions are self explanatory, however each function will be explore in depth.

```
Interactive mode

Options:
        (1) Load keyboard file
        (2) Node operations
        (3) Edge operations
        (4) Display graph
        (5) Display graph information
        (6) Enter string for finding path
        (7) Generate paths
        (8) Display path
        (9) Save keyboard
        (0) Exit
```

Figure 1: Interactive Menu Options

2.3.1.1 Load keyboard file

As shown in figure 2, the user has an option to either load a file or load a serialised file. NOTE: the file format should be like the adjacency list see section 6.2.2.

```
Options:
        (1) Read a File
        (2) Read a SerialIzed File
```

Figure 2: Load Keyboard File Options

2.3.1.2 Node operations

As shown in figure 3, the user has an option to either add a vertex, delete a vertex or find a vertex. The process for all three option is the same, the program will as the user the label of the vertex depending on the option selected.

```
Options:
        (1) Add A Node
        (2) Delete A Node
        (3) Find A Node
        (0) Exit
```

Figure 3: Node Operations Options

2.3.1.3 Edge operations

As shown in figure 4, the user has an option to either add an edge, delete an edge or find an edge. The process for all three option is the same, the program will as the user the labels of the origin vertex and the destination vertex and depending on the option selected will execute the option.

```
Options:
        (1) Add An Edge
        (2) Delete An Edge
        (3) Find An Edge
        (0) Exit
```

Figure 4: Edge Operations Options

2.3.1.4 Display graph

If the user enters 4 on the terminal the program will display the graph as an adjacency list. For more information refer to section 6.2.2.

2.3.1.5 Display graph information

If the user enters <4> on the terminal the program will display information about the the graph. Information displayed will be:

- The number of Vertices
- The number of Edges
- The number of Edges per Vertex

2.3.1.6 Enter string for finding path

If 6 is entered into the terminal, the user then has to enter a string to find the shortest path for.

2.3.1.7 Generate paths

This section does not display on the terminal however it checks if each character of the string is in the graph.

2.3.1.8 Display paths

As shown in figure 5, the user is given the option to either save the path of the string to a file called output.csv.

```
Options:
        (1) Save the paths (output.csv)
        (0) Do not save
```

Figure 5: Display paths Options

2.3.1.9 Save keyboard

If the user want to save the keyboard after making change to the graph. A simple prompt will show asking for the file name that it should save as.

2.3.1.10 Exit

If at any time the user want to exit the progam, the user has to enter 0 and the program is closed.

2.3.2 Silent Mode

To run the program in silent mode the user has to enter the following command into the terminal:

```
java keyMeUp -s <file 1> <file 2> <file 3>
```

1. file1: is the keyboards layout file.
2. file2: contains the string(s) that the program would search for.
3. file3: saves the output to the file.

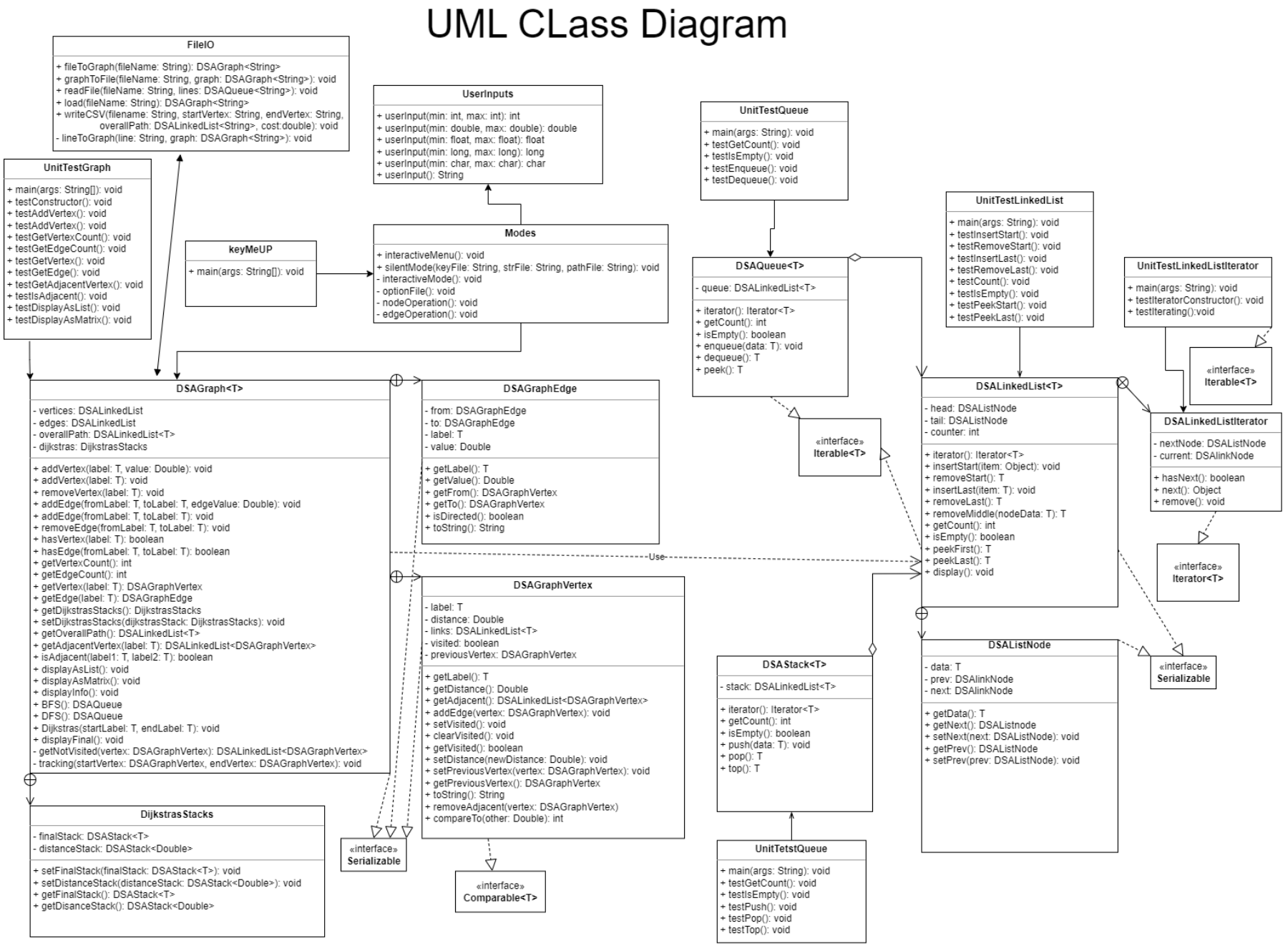
2.4 Future Work

There is future work to be done on the program. This includes a graphical user interface, which will display the graph as Dijkstra's algorithm is performed on it to give a better understanding of how each key is connected with one another. In order to achieve this, the code will need to be significantly modified to ensure it is able to be event driven with the GUI and to ensure that functions return string that can be used to display inside the GUI. Another modification that could be done is the use of a build tool such as Gradle. This is not a hard task to achieve considering all that is required is to set up gradle and follow the structure provided.

3 Traceability Matrix

	Requirements	Code/Design	Test	More Information
Modes and menu	If called without any arguments, system displays usage	keyMeUp.main()	User Testing	Section
	If called with "-i" as the first argument, system displays the interactive menu	modes.interactiveMenu()	User Testing	Section
	If called with "-s" as the first argument and the number of arguements are 4, system runs in silent mode	modes.silentMenu()	User Testing and UnitTestKeyMeUp	Section
Load Data	Menu to load a keyboard file	modes.optionFile()	User Testing	Section
	Load a serialised keyboard file	FileIO.load()	User Testing	Section
	Load a keyboard file	FileIO.fileToGraph()	User Testing	Section
Node Operations	Menu to different types of operations	modes.nodeOperation()	User Testing	Section
	Operation to add a node	DSAGraph.addVertex()	User Testing	Section
	Operation to delete a node	DSAGraph.removeVertex()	User Testing	Section
	Operation to find a node	DSAGraph.hasVertex()	User Testing	Section
Edge Operations	Menu to different types of operations	modes.edgeOperation()	User Testing	Section
	Operation to add an edge	DSAGraph.addEdge()	User Testing	Section
	Operation to delete an edge	DSAGraph.removeEdge()	User Testing	Section
	Operation to find an edge	DSAGraph.hasEdge()	User Testing	Section
Display Graph	Display the graph as an adjacency list	DSAGraph.displayAsList()	User Testing	Section
Display Graph Information	Display the graph information	DSAGraph.displayInfo()	User Testing	Section
Saving keyboard file (Serialised)	Save the keyboard file as a serialised file	FileIO.save()	User Testing	Section
Exit	Exits the program	keyMeUp.main()	User Testing	Section

4 UML CLass Diagram



5 Classes

5.1 keyMeUp

The keyMeUp class only has a main method. The main method checks for the valid command line arguments and number of command line arguments. Depending on the number of arguments and the first argument the program will either run (1) interactive mode, (2) silent mode or (3) print Invalid number of arguments.

5.2 Modes

The Modes class is comprised of static method and depends on the keyMeUp class. depend on the command line argument different methods are ran.

5.3 DSAGraph

The DSAGraph class is an implementation of the graph data type. The graph class does not depend on any other classes and implements the Serializable interface. The class uses generics to allow for custom type for the label of vertices and edges. It has three private inner classes called DSAGraphVertex, DSAGraphEdge and DijkstrasStack. The class fields for DSAGraph are:

- `DSALinkedList<DSAGraphVertex> vertices`
- `DSALinkedList<DSAGraphEdge> edges`
- `DSALinkedList<T> overallPath`
- `DijkstrasStacks dijkstras`

`vertices` and `edges` linked list represent all the vertices and edges in an instance of the DSAGraph class. The `overallPath` stores the label of all the vertices in order from starting to ending vertex. `dijkstras` is an instance of the DijkstrasStack class that contains stack for path and distance.

5.3.1 DSAGraphVertex

The DSAGraphVertex class represents each vertex inside the graph. The vertex class implements the Comparable<Double>, Serializable interfaces. The reason as to why DSAGraphVertex is a private class is because only a graph class should have access to vertex. The class fields for DSAGraphVertex are:

- `T label`
- `Double distance`
- `DSALinkedList<DSAGraphVertex> links`
- `boolean visited`
- `DSAGraphVertex previousVertex`

`label` is used to uniquely identify a vertex inside the graph. `links` is a linked list that stores all the adjacent vertex of a certain vertex. `distance` and `previousVertex` are both used in dijkstras method in the graph. `distance` stores the distance from the starting node and `previousVertex` store the vertex it was reached from.

5.3.2 DSAGraphEdge

The DSAGraphEdge class represents each vertex inside the graph. The vertex class implements the Serializable interfaces. The reason as to why DSAGraphEdge is a private class is because only a graph class should have access to vertex. The class fields for DSAGraphEdge are:

- `DSAGraphVertex from`
- `DSAGraphVertex to`
- `T label`
- `Double value`

`label` is used to uniquely identify a vertex inside the graph. `from` and `to` are used to determine if an edge in the graph is directed or undirected. `value` is used in the implementation of Dijkstra's algorithm.

5.3.3 DijkstrasStack

The DijkstrasStack class represents the path from a starting vertex to an ending vertex. The reason as to why DijkstrasStack is a private class is because only a Dijkstras method should have access to DijkstrasStack class inside DSAGraph class. The class fields for DijkstrasStack are:

- `DSAStack<T> finalStack`
- `DSAStack<Double> distanceStack`

Both `finalStack` and `distanceStack` serve the same purpose of when backtracking from the end vertex all the label for the vertices are stored `finalStack`, in the whereas all the distance is stored in the `distanceStack`.

5.4 DSALinkedList

The DSALinkedList class is an implementation of a doubly ended double linked list. The use of a linked list is idea when the number of objects to be stored if unknown. DSALinkedList class has two is comprised of two private inner classes DSAListNode and DSAListIterator. Hence, the class DSALinkedList has been created to allow for a container of multiple data with no specified max capacity. The class fields for DSALinkedList are:

- `int counter`
- `DSAListNode head`
- `DSAListNode tail`

`counter` increments and decrements each time a node is either inserted or deleted from the linked list. `head` and `tail` store where the head and the tail of the list are at all times.

5.4.1 DSAListNode

The list node represents each node inside the linked list. the reason why DSAListNode is a private inner class is because only the linked list can access the node. The class fields for are:

- `T data`
- `DSAListNode prev`
- `DSAListNode next`

`data` store the data in a new node of the linked list. when a node is created and inserted into the linked list `prev` and `next` store the previous and the next vertex for each node.

5.4.2 DSALinkedListIterator

The iterator is there to allow iterating of the linked list, this is achived by implementing the Iterator interface. This iterating also makes the linked list very useful as data can be stored in it then displayed later on. The class fields for DSALinkedListIterator are:

- `DSAListNode nextNode`
- `DSAListNode current`

`nextNode` and `current` are used to iterate through the linkedlist.

5.5 DSAQueue

The DSAQueue class is an implementation of the queue data type. The class depends on the DSALinkedList class to represent its queue and implements the Iterable interface. By using a linked list, the queue can now dynamically change its size. Queues are used in this program to provide an explicit order when it comes to inserting and removing from the linked list. The class fields for DSAQueue are:

- `DSALinkedList<T> queue`

`queue` is used to represent the queue. Methods such as getting the count are extended through the DSALinkedList class.

5.6 DSAStack

The DSAStack class is an implementation of the stack data type. The class depends on the DSALinkedList class to represent its stack. By using a linked list, the stack can now dynamically change its size. Stacks are used in this program to provide an explicit order when it comes to inserting and removing from the linked list. The class fields for DSAStack are:

- `DSALinkedList<T> stack`

`stack` is used to represent the stack. Methods such as getting the count are extended through the DSALinkedList class.

5.7 FileIO

The FileIO class is comprised of methods related to input and output of files the class depends on DSAGraph, DSAQueue and DSALinkedList. Each of these methods have certain functionality The methods in FileIO are :

- `fileToGraph`
- `lineToGraph`
- `graphToFile`
- `readFile`
- `load`
- `writeCSV`

`fileToGraph` calls `lineToGraph` to convert each line from the file to a graph. `graphToFile` serialises the graph and save the file.`readFile` and `writeCSV` are mean to read a file and save a file respectively. `load` is used to load a serialised graph file.

5.8 UserInputs

The FileIO class is comprised of methods related to user input. Each of these methods have certain functionality The methods in UserInputs are :

- `userInput`
- `userInput`
- `userInput`
- `userInput`
- `userInput`
- `userInput`

`userInput` is method overloading, all of the methods have different parameter. Five of `userInput` methods are used to make sure the user enters a value that is inside the bound. One of `userInput` converts the user input and return a string.

6 Justification of Decisions

6.1 Generics

All the data structure in the program use generics. Generics offer the ability to have one class that is flexible for different types. However, the use of generics did did cause some issues when it came to searching, creating or deleting an edge in the DSAGraph class, due to the edge label <= `originVertex + "-" + destinationVertex`. Since a string and a generic operation was taken place it had to type casted as well as suppressed warning. However, in every other scenario generics made the code a lot simpler and easier to understand, due to knowing exactly what was being stored in each instance of the data structures, as well as removing the need to type cast when ever accessing data stored in these data structures.

6.2 Representing Keyboard

There are several ways of representing a graph and depending on the representation it can have severe impact on performance.

6.2.1 Adjacency Matrix

Representing the graph as a adjacency matrix is the simplest in terms of data representation. It is represented in a 2D matrix of size(V^2), where V is the number of vertices. Each element in the matrix is the weight of the edge from vertex on the left to the vertex on the top of the matrix. Another benefit of using an adjacency matrix is that space efficient for representing graphs which is dense. The time complexity of looking up an edge weight in an adjacency matrix is $\mathcal{O}(1)$, this means that it takes a constant time no matter the amount of data in the matrix. As mentioned before the size of the matrix is V^2, therefore it require $\mathcal{O}(V^2)$ space. Another disadvantage of adjacency matrix is that to iterate over all the edges it have a time complexity of $\mathcal{O}(V^2)$.

6.2.2 Adjacency List

An adjacency list is a method to represent the graph as a map of from vertex to list of edges. The benefits of using an adjacency list is the space efficiency when representing a graph with sparse vertices. it is also efficient when iterating through all the edges in the graph. However, it is slightly more complex graph representation and the edge lookup is $\mathcal{O}(E)$.

6.2.3 Edge List

Edge list is a slight derivation of the adjacency list, it has all the advantages and disadvantages of am adjacency list with the addition of a very simple structure ad the advantage.

The graph representation chosen in this program is the adjacency list. The justification behind using an adjacency list is since all the edges have a weighting of 1 the time complexity of finding the weight is $\mathcal{O}(1)$. Another reasoning as to why an adjacency list was chosen over an adjacency matrix was the space complexity of the keyboard layout. Even though adjacency list are less space efficient with denser graphs, the space complexity for an adjacency matrix is $\mathcal{O}(V^2)$. In section 6.5, it is discussed that this program implements Dijkstra's algorithm for the shortest path and therefore an adjacency list was implemented since it is used in Dijkstra's algorithm.

6.3 Data Storage

6.3.1 Abstract Data Types

It was decided that a graph data structure would make the most sense to store all the key data. This graph is a directed graph, with each edge being one directional, as each character on the keyboard only goes one way with the weight of 1. This is as a movement can be used to represent an edge as it is going from one character, to another. From practical6 the DSAGraph Class from was modified to use generics in its structure. A* was the initially planned to be used, however,

calculating a heuristic in a graph raised an issue, either to (1) brute force the heuristic values for each vertices or to (2) either implement a recursive or iterative algorithm to calculate the heuristic of each vertices relative to other vertices. In either case it would consume time to complete these calculations. Therefore Dijkstra's algorithm is implemented in this program. For more information see Section 6.5.

6.3.2 Storing the paths

6.3.3 Serialisation

Serialisation is useful when their is a need to recreate an object into its original state in a different machine. Serialisation converts the objects into bytes of stream in so that they can easily shift from one JVM to another. Another benefit of using serialisation is that it saves the state of the object for recreation of these objects without having to reprocess them. In this program, serialisation is mainly used to store the keyboard file and output.//

6.4 File reading

6.5 Path Finding Algorithms

Depending on the structure of the graph different types of path finding algorithms are optimal. When focusing on path-finding algorithms on graphs there are different scenario to be considered, such as is the graph directed or undirected? Are the edges of the graph weighted? Is the Graph likely to be sparse or dense with node?

One of the most common problem in graph theory is the shortest path problem. The problem is given in a weighted graph what is the shortest path of edges from node A to node B. There are different algorithms that are optimal in different scenarios. When it comes to graph theory connectivity is a big issue, at its core the problem is does there exists a path from node A to node B. Detecting negative cycles is yet another problem, this is because if there exists a negatively weighted edge it can corrupt an algorithm. In this program the keyboard edges do not have any negative weights. For this program, research was conducted on different path finding algorithms. Some of the algorithms researched were:

- Breath First Search (BFS)
- Dijkstra's Algorithm
- A*
- Bellman Ford

All of these algorithms could have been implemented however, it would not necessary be optimal to implement some of these algorithms.

Breath First Search (BFS) is a fundamental graph traversal algorithm that is used to explore vertices and edges in a graph. The time complexity of Breath First Search is $\mathcal{O}(V + E)$ and is optimal when the graph is unweighted. Breath First Search is often used as foundations to other algorithms. Breath First Search start at an arbitrary vertex of the graph and explores all the neighbours then moving on to the next level neighbours. Breath First Search uses a queue to store all the neighbours that have been visited, when at the starting vertex, it is en-queued to the queue and once all of its neighbours have been en-queued to the queue the vertex if dequeued. This process is either done recursively or iteratively until all the vertices have been visited. This program does not implement Breath First Search because starts at and arbitrary vertex and does not exit if the end vertex has been found. Another reason as to why the program has opt out of Breath First Search is that on a weighted graph, the fastest path may not be the shortest path.

Dijstra's algorithm is a Single Source Shortest Path (SSSP) algorithm for graphs with non-negative edge weights this means that when implementing Dijstra's algorithm the starting vertex is explicitly provided. The time complexity of Dijstra's algorithm is typically $\mathcal{O}(E \log (V))$. In its simplest form Dijkstra's algorithm starts at the starting vertex with the distance 0 and stores the neighbouring vertices in a priority queue with updating the value of each vertex by the previous vertex value + the edge value. In the priority queue the queue is sorted from the next most promising vertex to the least promising vertex. It iterates through the priority queue while it is not empty to find the shortest path to all the vertices. There are many variation of Dijkstra's implementation:

1. Lazy implementation
2. Finding shortest path + stopping early
3. Using indexed priority queue + decrease key
4. Eager Dijkstra's
5. Heap optimization with D-array heap

This program implements the an optimised version of finding the shortest path and stopping early version of Dijkstra's algorithm. The main idea behind this implementation is that since Dijkstra's algorithm process the next most promising vertex and the end vertex has been visited. The algorithm won't find path that is shorter than the path it has already found. In this program the DSAGraphVertex has a class field called `DSAGraphVertex previousVertex` that stores the previous vertex. When the program runs the algorithm, the starting vertex has a `previousVertex = null` as it is the starting vertex and `distance = 0` as it cost 0 moves to go from the starting vertex to itself and then it is stored in a priority queue. the program iterates through its adjacent vertices and setting their `previousVertex` equal the vertex it was reach from and the `distance` equal to the vertex distance + the edge cost and adding these vertices to the priority queue. This process to repeated until the end vertex is found. Once the end vertex is found, a simple backtracking algorithm is used in which starting from the end vertex it iterates through the vertex to find the starting vertex and all the vertices are stored in a stack to make it easier to display the path.

As discussed in section 6.3.1, the A* algorithm is an extension of Dijkstra's algorithm, it is also prominently used on graphs that are either weighted or unweighted. A* functions similar to the Dijkstra's algorithm with the addition of heuristic values. Heuristic values provide a sense on direction relative from the destination. This is so that instead of prioritising vertices with the lowest distance from the current vertex the algorithm prioritises vertices that have the lowest heuristic value. In theory this in not hard to visualise however, as mentioned in section 6.3.1 the issue is calculating the heuristic value for each vertex. This would end up decreasing the performance of algorithm where at each vertex it would have to calculate the heuristic value.

As mentioned before one of the researched algorithms was Bellman Ford. Compared to all the other algorithms, Bellman Ford is the simplest. Just like Dijkstra's algorithm, Bellman Ford is also a Single Source Shortest Path (SSSP) algorithm for graphs with negative edge weights. however, Bellman Ford is not ideal for Source Shortest Path because the time complexity $\mathcal{O}(EV)$ is much worse compare to Dijkstra's algorithm. The only reason to implement Bellman Ford is if Dijkstra's algorithm does not work and it only breaks when there any negative edge weights in the graph. This is because it keeps finding better paths with a lower cost. For the reasons provided, the program opted not to implement Bellman Ford.

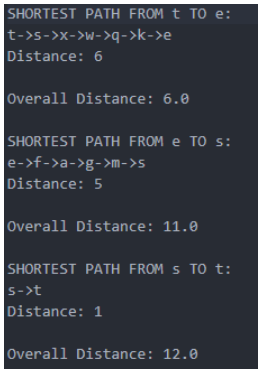
7 Results

7.1 Introduction

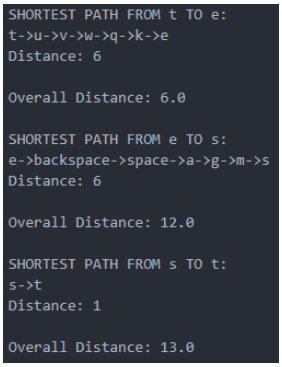
In this program as the user enters a string the program tries to find if all the character exist in the graph, if a lowercase of an alphabet exists, the program will replace the user enter capital character with the lowercase other wise if it cannot find the vertex it will ignore the character. The 3 different scenario that will be discussed are: the impact of wrapping and different types of string cases

7.2 Scenario 1: Impact of Wrapping

In this scenario, keyboard layout iview, stan and netflix will be explored. iview allows wrapping to be done up, down, left and right, where stan allows wrapping only left and right and netflix does not allow wrapping at all. For the same sting `test`, on Dijkstra's algorithm only travelled a total distance of 12 step to complete the string, on both iview and stan, figure 6a this is because both of the keyboards allow left and right wrapping however, for netflix the total distance was 13, figure 6b. This is because it does not allow for wrapping of any sorts.



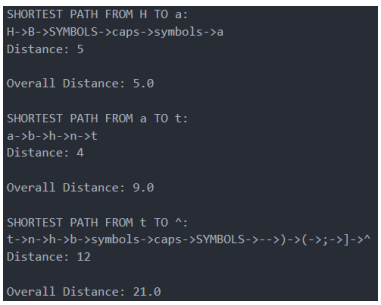
(a) Dijkstra's Algorithm on iview and stan keyboards



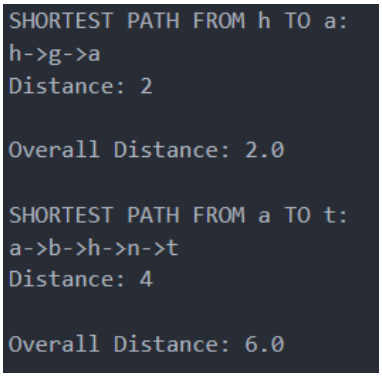
(b) Dijkstra's Algorithm on netflix keyboards

7.3 Scenario 2: Sting Cases

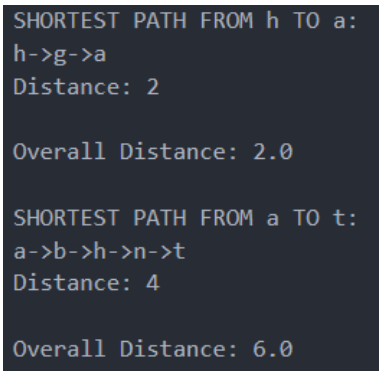
In this scenario, keyboard layout iview, stan and netflix will be explored. Keyboards netflix and stan do not implement symbols nor capital alphabets in their layout. However, iview keyboard layout implements both symbols and capital alphabets in its layout. For the the string `Hat^` with be used. Referring to figure 7, it can be observed the for iview keyboard layout, figure 7a, the graph was able to find all of the vertices with label from each character of `Hat^`. However, this is not the case for stan and netflix keyboards, From figures 7b and 7c, it is observed that both netflix and stan were not able to find vertices for H and , therefore they used the vertex with label h and ignored the character as it is not in the keyboard and the graph.



(a) iview keyboards



(b) stan keyboards



(c) netflix keyboards

Figure 7: Dijkstra's Algorithm for Hat

8 References

1. Practicals 1 to 6, blackboard, Rizvi, HN.
2. Lecture Slides 1 to 6, blackboard, Maxwell, Valerie.