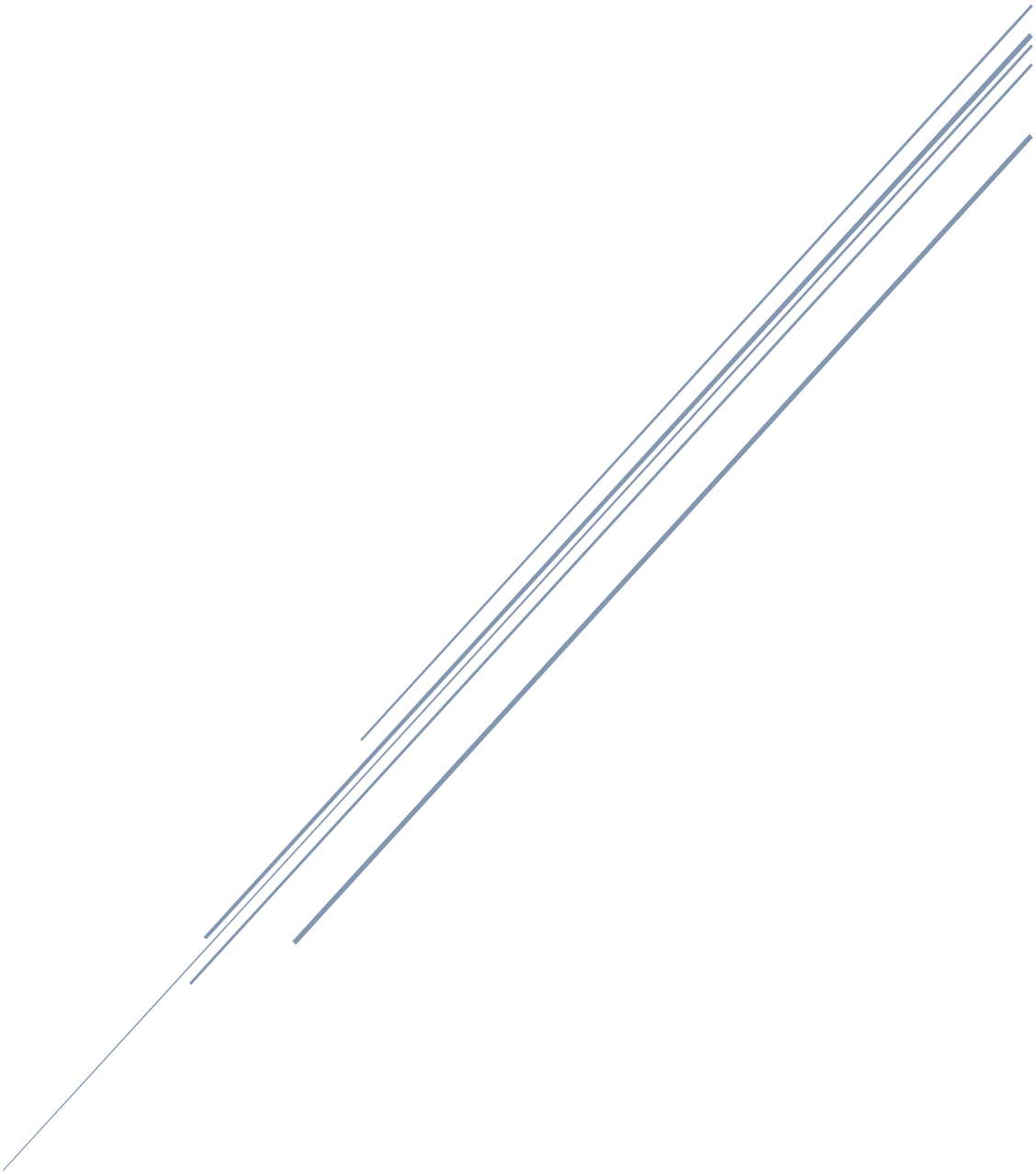
DATA STRUCTURE & ALGORITHMS ASSIGNMENT REPORT

Sohail Bakhshi Student ID :20605126



Curtin University
COMP1002

Table of Contents

User Guide

The purpose of this assignment was to build a representation of a keyboard and to find the best steps for a given string. Within my keyMeUp.py program I have created an interactive menu that has 2 options to run, these include silent mode and interactive mode.

Interactive Mode:

To run interactive mode the terminal only requires one command line argument. For example, to access the interactive mode you would type:

- python3 keyMeUp.py -i

This mode displays a menu that gives users the option to select from 10 options. Which include:

- Load keyboard file
  - Reads a keyboard file and inserts nodes and edges into the graph
- Node operations
  - Gives users the option to insert, delete, find, or update a node
- Edge operations
  - Gives users the option to insert, delete, find, or update an edge
- Display graph
  - Displays an adjacency list and matrix
- Display graph information
  - Displays the graph information such as the number of edges and vertices and the direction and weight of edges
- Enter string for finding path
  - Allows users to enter a string to find the shortest path
- Generate paths
  - Generates a few paths into a file called paths.txt
- Display paths
  - Displays the generated paths in order
- Save keyboard
  - Saves the keyboard as a serialised file so it can be reopened
- Exit
  - Allows users to quit the menu/breaks the loop

Silent Mode

To run silent mode, you will require to input a keyboard file, string file and an output file for the sorted paths. For example, to use silent mode you would type some like this:

- python3 keyMeUp.py -s keyboard.txt stringfile.txt output.txt

This mode will allow users to easily get paths of any keyboard and any strings in a file and outputs the paths ranked in order in the output file.
This mode has been tested with a few string files and keyboard files but feel free to try with your own however they must be in the same format as the files I have provided.

I would also like to note to make sure you are in the correct directory by checking with the command 'ls' in order to load up any files.

Class Descriptions

Keyboard: This class contains all the functions for the interactive mode. This includes all functions for reading a keyboard file, deleting, finding, inserting, updating nodes and edges, path functions and saving and loading a keyboard file. This class calls a lot of the DSAGraph class functions as this is where the functionality comes from. If I were to redo this assignment, I would try to have separate classes for the node and edge operations however I found it a lot simpler to have all the functions in one class.

Menu: This class is important as it runs the interactive menu and allows for users to interact and select from a list of options. It calls the keyboard object so that it can call the functions that are required for each option in the interactive menu. Within the class I have creare a function called userInterface() which displays the interactive menu through an input function . My other function called interactive() will then call this function the user does not exit the program. This class is essentially just one function that contains a long while loop and a bunch of if/elif statements which allows for user interaction. If I had more time to improve my code I would implement it so that there isnt so much code in the if/elif statements but instead they would call functions from a separate class.

DSAGraph: This class is the most important class as it is where the main components of the keyboard class come from. This class is responsible for creating a graph which is important especially with this assignment. Within this class I use two outer classes for the edge and node which help develop the graph class. The class includes a lot of methods such as find, add, delete, and update which is used in the keyboard class. The class uses linked lists to store edges and vertices which makes it a lot easier to represent a graph and also uses stacks and queues for path finding algorithms. Although another way of creating the graph could have been through hash tables however, I found it easier to use a linked list. This was an extension of the graph class created in practical 6.

DSAGraphEdge: This class was made to extend my graph class. Previously in the practical I only had a node class however I added this class as I believed that it would help with finding the shortest path by adding weights to the edges however, I found it quite

difficult and by the end of my assignment I realised it probably wasn't necessary to add this class but I left it in anyway.

DSAGraphNode: This class originally was used to store my adjacent vertices. However, after adjusting it with my edge class, it now stores the edges of each vertex. As stated previously I don't think it was necessary in the end for me to change my graph from the practical as I didn't end up using it to benefit me in anyway.

DSALinkedList: This is a data structure that is used to insert and remove data. This is particularly handy when it comes to handling lots of data which is useful for graphs. My linked list is a doubly linked list which allows me to retrieve the head and tail values along with the next and previous values. This class was re used from practical 4.

DSAListNode: This class is responsible for holding the node values which is used within the linked list class. It also allows me to get the next and previous nodes. This class was re used from practical 4.

DSAStack: This class is a stack which has functions like push, pop, peek. This helped with path algorithms like depth first search. This class was re used from practical 3.

DSAQueue: This class is a queue which has functions like enqueue, dequeue, peek. This helped with path algorithms like breadth first search and depth first search. This class was re used from practical 4 where I used the linked list to replace the functions.

myException,valueError,AlreadyEdge, NotFoundError: These classes were used as my custom exceptions. Although they weren't necessary, I thought that I should try and implement it for my own benefit.

keyMeUp: This is the class that will have all the functionalities that were implement through the other classes. It allows keyMeUp.py to run with all the functionalities implemented. This class allows for command line arguments and allows users to choose between interactive or silent mode. Originally, I had this in a main function but since it was better practice to have my code in classes I decided to also add this to a class.

Justifications

- For my shortest path algorithm, I chose to use a breadth first search algorithm where it takes in parameters from a start node to a destination node. There wasn't really a specific reason for why I chose breadth first search over depth first search for the shortest path it just came down to what I could get working first and it happened to be breadth first search. The code I implement was inspired off a code I found on the internet (Python in Wonderland. 2017.) however I had to change it, so it adapted with my code and made it, so it didn't use built in functions like queues and lists which I substituted for my own queue and linked list classes.

- I had another two other path functions which used breadth first search (BFSPath()) and depth first search (path()) which just gave me a path from a start node to end node however the paths returned were very long. As I wanted there to be more than one path this was my last resort as I couldn't get anything else to work.

- My sorting paths algorithm wasn't really necessary as I could have just put my paths in order manually. However, I purposely put the paths in random order in the findStringPath function so I could sort the paths based on path length. I used one of the sort algorithms from practical 1 also known as selection sort.

- I used selection sort because I felt that the implementation of it was suitable for my code. The best case and worst-case stime complexity is O(n^2) and although insertion sort has a best case of O(n), when sorting larger amounts of data selection sort will work better than insertion sort.

- My findStringPath function writes the unordered paths into a file. This is because this was the easiest way, I could think to sort the paths was to put it in a file then read it a sort it.

- I used my DSAGraph as it was the best suited data structure for this assignment .It contains nodes and edges that connect to each other which is helpful for finding the shortest path a string can take in a keyboard. Compared to other data structures I don't really see how else I would do the assignment without the graph.

- My DSALinkedList has to be the most important data structure used due to the fact that without it my graph would not work. Although and alternative to linked lists could have been a hash table however, I felt as though I would rather use a data structure, I'm more confident with as I really didn't have much working time with hash tables. If I were to do the assignment again I would definitely try and implement it with hash tables as the complexity in most cases is O(1) compared to linked lists which is mostly O(n).

- I used my DSAStack and DSAQueues class because it helped me with my path algorithms such as breadth first search and depth first search.

Showcase

Introduction

My program has two modes. Interactive and silent mode. The main mode we will be testing is through silent mode. This mode will allow me to test different files and keyboards to see what the output will be from the command line. Through this mode I can test different keyboards that have either capitals, lowercase, special characters, or numbers with a file the contains strings. The way I have setup my code to work is that I read the file with the strings and insert the strings into a function that gets each individual word, and it gets the path between each letter for example, the word 'cat' would find the path between c to a then a to t. I then return the final path into a file called paths.txt which is then read and sorted in order using selection sort into an output file of choice. The only issue I found was that in my paths.txt file if a path is unable to be produced it will display the previous paths that were generated as it wouldn't have overwritten the file as there were no paths generated which means that for testing purposes it is best that I manually remove the paths from the paths.txt file after every test. Furthermore, to test my scenarios I have decided to have three different keyboards and tested them with different string files which to see what output they will give.

Scenario 1

For this scenario I will be using three different keyboards with only one string file. The first keyboard is inspired by a Netflix keyboard. The second keyboard is inspired by a switch keyboard. The third keyboard is the capital mode of a switch keyboard. The first-string file I will testing is using contains animal names which is within stringfile.txt. To run these tests, I will be using silent mode. The following command is typed into the terminal:

python3 keyMeUp.py -s keyboard.txt stringfile.txt output.txt

python3 keyMeUp.py -s keyboard2.txt stringfile.txt output2.txt

python3 keyMeUp.py -s keyboard.3txt stringfile.txt output3.txt

The results are outputted into a file called output/2/3.txt which contains 30 paths (3 for each string) ranked in order. Output3 has no results as keyboard3.txt contains all capital letters and the strings in the string file are lowercase.

The top five paths in output.txt and output2.txt are displayed below:

Output.txt
1: p-->o-->ii-->h-->g
2: c-->i-->oo-->p-->q-->w
3: l-->k-->j-->ii-->oo-->n
4: c-->b-->aa-->b-->h-->n-->t
5: d-->c-->i-->oo-->n-->m-->g

Output2.txt
1: p-->o-->ii-->u-->y-->t-->g
2: c-->x-->z-->aa-->s-->d-->f-->g-->t
3: l-->k-->ii-->oo-->i-->u-->y-->h-->n
4: r-->e-->w-->q-->aa-->s-->d-->f-->g-->t
5: d-->f-->g-->h-->j-->k-->l-->oo-->i-->u-->y-->t-->g

For clarification the words in output.txt are:

1. pig
2. cow
3. lion
4. cat
5. dog

For clarification the words in output2.txt are:

1. pig
2. cat
3. lion
4. rat
5. dog

The following examples display the first five paths of each output file that gave back a result. Ouput2.txt has a different output compared to output.txt as a result of the keyboard2.txt being a 4X11 structure which a lot longer in terms of length compared to keyboard1.txt which is 6x6. This means that the route it takes would be longer as it would have to travel further to find each node. It is also apparent that the string 'cow' is now no longer in the top 5 in output2.txt

Scenario 2

In scenario 2 we will be testing all three keyboards again but with a different string file. This file contains strings with special characters, and numbers. To run these tests, I will be using silent mode. The following command is typed into the terminal:

python3 keyMeUp.py -s keyboard.txt stringfile2.txt output.txt

python3 keyMeUp.py -s keyboard2.txt stringfile2.txt output2.txt

python3 keyMeUp.py -s keyboard.3txt stringfile2.txt output3.txt

Due to keyboard1 not having special characters and keyboard3 being in capitals, there was no output in ouput.txt and output3.txt.

The top five paths in output2.txt are displayed below:

```
1: t-->r-->ee-->w-->ss-->d-->f-->g-->tt-->y-->u-->i-->o-->p-->:
2: c-->v-->b-->n-->m-->j-->uu-->y-->t-->rr-->tt-->y-->u-->ii-->u-->y-->h-->n
3: h-->g-->f-->d-->ee-->r-->t-->y-->u-->i-->o-->ll-->k-->ll-->oo-->p-->/-->'-->!
4: b-->v-->c-->x-->z-->aa-->s-->d-->f-->g-->h-->j-->kk-->j-->hh-->g-->f-->d-->ss-->d-->f-->g-->hh-->j-->k-->i
5: s-->d-->f-->g-->h-->j-->k-->l-->oo-->i-->u-->y-->hh-->g-->f-->d-->s-->aa-->s-->d-->f-->g-->h-->j-->k-->ii-->o-->l
```

For clarification the words are:

1. test:
2. curtin
3. hello!
4. bakhshi
5. sohail

As previously stated, there was only one output file which is because my other keyboard files didn't return the required nodes in order for there to be a path. In order for there to be a path I would have to insert all 3 keyboards at once which will insert every node and edges. However, this is not possible in silent mode but can be done in interactive mode

Scenario 3

For scenario 3 I will be testing all three keyboards again with a different string file. The string file will contain strings in capital letters. To run these tests, I will be using silent mode. The following command is typed into the terminal:

python3 keyMeUp.py -s keyboard.txt stringfile3.txt output.txt

python3 keyMeUp.py -s keyboard2.txt stringfile3.txt output2.txt

python3 keyMeUp.py -s keyboard.3txt stringfile3.txt output3.txt

Due to keyboard1 and keyboard2 being in lowercase, there was no output for these keyboards .

The top five paths in output3.txt are displayed below:

```
1: P-->OO-->LL-->O
2: -->P-->O-->O-->I-->P-->9-->L-->L-->K-->&-->O
3: N-->M-->*-->K-->II-->KK-->J-->H-->G-->F-->D-->E
4: P-->O-->I-->UU-->J-->MM-->N-->B-->V-->C-->X-->Z-->A
5: A-->S-->DD-->F-->G-->H-->J-->K-->II-->U-->Y-->T-->R-->E-->DD-->S-->AA-->S
```

For clarification the words are:

1. POLO
2. POLO
3. NIKE
4. PUMA
5. ADIDAS

The results are similar to scenario 2 as there is only one result due some of the keyboards not having the required nodes to find the path.

Scenario 4

I have decided to add an extra scenario in order test all three of my keyboards in interactive mode. As previously discussed in scenario 1,2 and 3 when I test strings with special characters, lowercase and uppercase it results in some output files not giving results. In this scenario I will be loading all three of my keyboards and testing if I can get a path no matter the string type.

After loading all the keyboards, I will type a list of random words into option 6 of the interactive menu to return the shortest path:

Results

- Dog : Shortest Path for string = d-->c-->i-->oo-->n-->m-->g
- Cat: Shortest Path for string = c-->b-->aa-->g-->t
- curtin! : Shortest Path for string = c-->i-->uu-->t-->rr-->tt-->u-->ii-->h-->nn-->m-->,-->.-->?-->!
- university! : Shortest Path for string = u-->t-->nn-->o-->ii-->c-->vv-->w-->ee-->rr-->x-->ss-->t-->u-->ii-->u-->tt-->yy-->s-->m-->,-->.-->?-->!
- NIKE: Shortest Path for string = N-->M-->*-->K-->II-->KK-->L-->O-->9-->3-->E
- ADIDAS: Shortest Path for string = A-->S-->DD-->S-->W-->2-->8-->II-->O-->9-->3-->E-->DD-->S-->AA-->S

Furthermore, these results display that I can indeed find the paths of any characters whether its lowercase, uppercase or contains special characters as long as I have loaded in all my keyboards.

Conclusion and future work:

To conclude my implementation of this assignment was not perfect however I have tried my best to meet every requirement for the task. I believe that the data structures I utilised were reasonable and allowed me to get the main functionalities working. When implementing an algorithm to find the best path I had a lot of difficulties trying to get one to work. I had originally thought of giving every edge a weight which then I could achieve the shortest path by taking the route with the shortest distance based on the edge weight

however I later realised that using breadth first search was a lot easier. Throughout the assignment I had unintentionally used lists which resulted in a lot of changes to code which I am glad I noticed or else I could have lost a lot of marks. If I had more time to work on the assignment, I would have added visualization which could have given me potential bonus marks. I also wish that I had better paths, although the paths that I have now do work, however, I wish that I could have made it so there was more than one shortest path rather than the same path displaying for the same string. In the future I wish to make my code a bit more concise and neater as I feel like I have too many files and too many functions that are a bit redundant. Overall, I am happy that I have managed to create something that I thought would be unmanageable when reading the assignment specs.

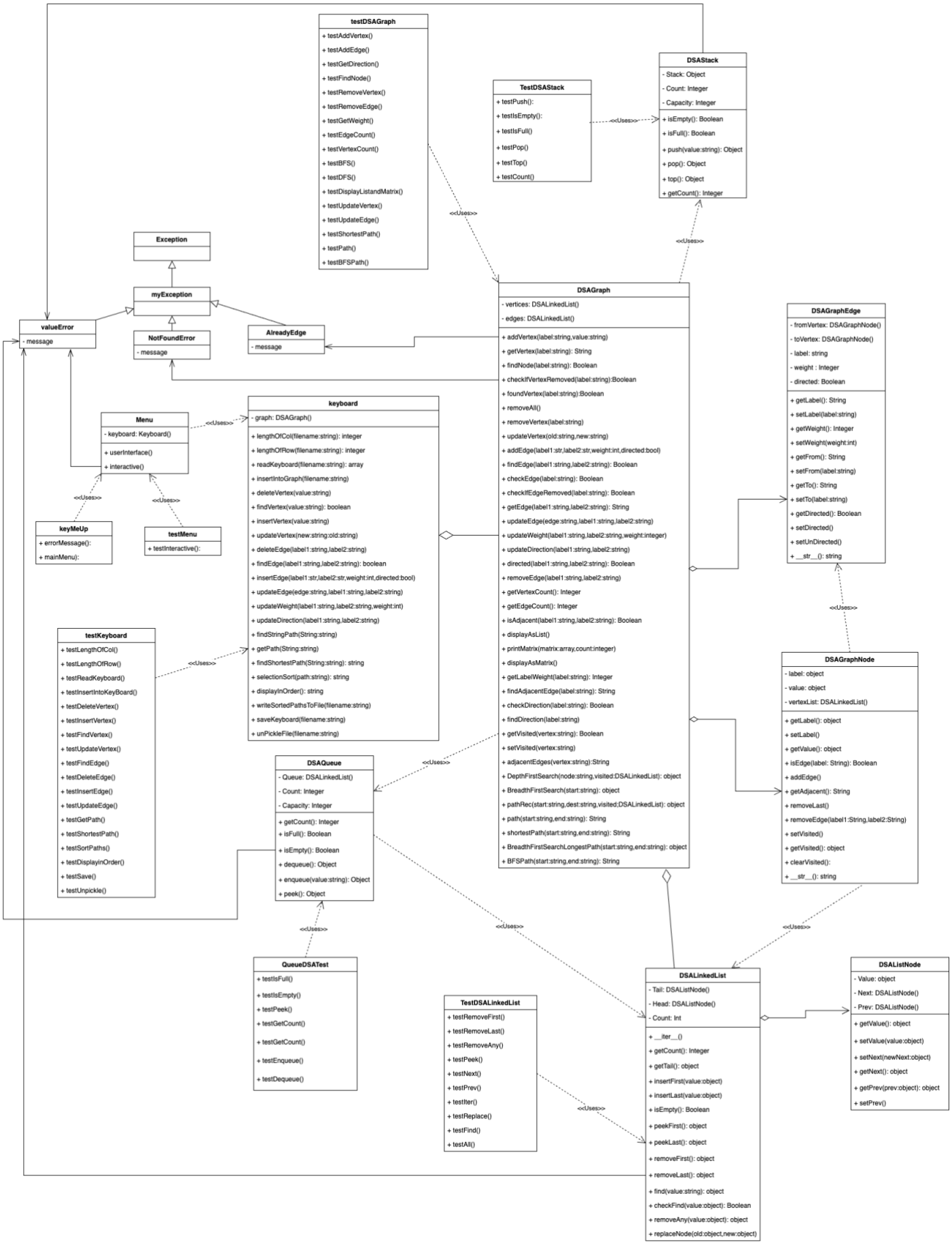
Reference List:

Curtin University. “COMP1002 Data Structures and Algorithms” 2022. Lectures 1-6

FavTutor. 2020. “Depth First Search in Python.” 2020. https://favtutor.com/blogs/depth-first-search-python.

Python in Wonderland. 2017. “How to Implement Breadth-First Search in Python.” Python in Wonderland. March 18, 2017. https://pythoninwonderland.wordpress.com/2017/03/18/how-to-implement-breadth-first-search-in-python/.

UML (also in file)



Traceability Matrix

	Feature	Requirements	Design/Code	Test
1	Modes and Menu			
		1.1 System displays usage if called without arguments	keyMeUp.py | mainMenu() | lines 23-24	testKeyMeUp.sh [passed]
		1.2 System displays interactive menu with "-i" argument	keyMeUp.py | mainMenu() | lines 25-27	testKeyMeUp.sh [passed]
		1.3 In interactive mode, user enters command and system responds.	Menu.py | interactive() | lines 25-181	MenuUnitTest.py [passed]
		1.4 System enters simulation mode with "-s" argument	keyMeUp.py | mainMenu() | lines 31-41	testKeyMeUp.sh [passed]
		1.5 System returns appropriate error from wrong user input	keyMeUp.py | mainMenu() | lines 29,44-45,47-48	testKeyMeUp.sh [passed]
2	File I/O			
		2.1 Load keyboard file	Keyboard.py | insertIntoGraph () | lines 60 -80	keyboardUnitTest.py [passed]
		2.2 Load serialised file	Keyboard.py | unpickleFile() | lines 265 -272	keyboardUnitTest.py [passed]
		2.3 check for invalids	Keyboard.py | readKeyboard() | lines 54 - 55	keyboardUnitTest.py [passed]
		2.4 write unsorted paths to file	Keyboard.py | findStringPath() | lines 165 - 177	keyboardUnitTest.py [passed]
		2.5 read unsorted paths and sort	Keyboard.py | displayInOrder() | lines 236 -241	keyboardUnitTest.py [passed]
		2.5 save serialised keyboard	Keyboard.py | saveKeyboard() | lines 256-261	keyboardUnitTest.py [passed]
		2.6 save sorted paths to file	Keyboard.py | writeSortedPathsToFile() | lines 245-252	keyboardUnitTest.py [passed]
		2.7 read string file	Keyboard.py | getPath() | lines 181-204	keyboardUnitTest.py [passed]
3	Node Operations			
		3.1 can delete vertex from graph	Keyboard.py | deleteVertex() | lines 83-88	keyboardUnitTest.py [passed]
		3.2 can insert vertex into graph	Keyboard.py | insertVertex() | lines 104 - 109	keyboardUnitTest.py [passed]
		3.3 can find vertex from graph	Keyboard.py | findVertex() | lines 92 -99	keyboardUnitTest.py [passed]
		3.4 can update vertex from graph	Keyboard.py | updateVertex() | lines 113 - 118	keyboardUnitTest.py [passed]
4	Edge Operations			

		4.1 can delete edge from graph	Keyboard.py | deleteEdge() | lines 122 -124	keyboardUnitTest.py [passed]
		4.2 can insert edge into graph	Keyboard.py | insertEdge() | lines 141 - 146	keyboardUnitTest.py [passed]
		4.3 can find edge from graph	Keyboard.py | findEdge() | lines 128 - 137	keyboardUnitTest.py [passed]
		4.4 can update edge	Keyboard.py | updateEdge() | lines 150-151	keyboardUnitTest.py [passed]
		4.5 can update edge weight	Keyboard.py | updateWeight() | lines 155-156	keyboardUnitTest.py [passed]
		4.6 can update/change the direction of an edge	Keyboard.py | updateDirection() | lines 160 -161	keyboardUnitTest.py [passed]
5	Path generation			
		5.1 can generate a random path for string	Keyboard.py | findStringPath() | lines 165-177	keyboardUnitTest.py [passed]
		5.2 can generate shortest path for string	Keyboard.py | findShortestPath() | lines 209 -216	keyboardUnitTest.py [passed]
		5.3 gives error message if no vertices or edges in graph	Menu.py | line 133-136,146-148,169-171	keyboardUnitTest.py [passed]
		5.4 can generate paths after reading from a text file	Keyboard.py | getPath() | lines 181-197	keyboardUnitTest.py [passed]
6	Sort Paths			
		6.1 can sort paths in order	Keyboard.py | selectionSort() | lines 223-232	keyboardUnitTest.py [passed]
7	Display Graph			
		7.1 displays adjacency list	DSAGraph.py | displayAsList() | lines 411-416	GraphUnitTest.py [passed]
		7.2 displays matrix	DSAGraph.py | displayAsMatrix() | lines 428-444	GraphUnitTest.py [passed]
		7.3 displays edge count	DSAGraph.py | getEdgeCount() | lines 391-398	GraphUnitTest.py [passed]
		7.4 displays vertex count	DSAGraph.py | getVertexCount() | lines 386-387	GraphUnitTest.py [passed]