Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Sim	Student ID:	12894732
Other name(s):	Gabriel		
Unit name:	Data Structures and Algorithms	Unit ID:	COMP5008
Lecturer / unit coordinator:	Valerie Maxville		
Assessment:	Assignment		

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature:	GABRIEL SIM	Date of signature:	11/10/2022

(By submitting this form, you indicate that you agree with all the above text.)

User Guide

How to use

1) Loading a virtual keyboard layout requires the user to select from the keyboard layout files from the root location. The user is required to type the keyboard file name (including the suffix .al)
2) Vertex and Edge Operations allow the user to add (new), modify, remove and find existing vertices/edges to our virtual keyboard
3) Display of the graph is shown in an Adjacency List format
4) To showcase the navigation feature of the virtual keyboard, the user is required to type in an input string (includes most keyboard characters). Note: if a character entered does not exist in the virtual keyboard an appropriate message will be displayed
5) Once the input string is entered, an option to generate the paths can be selected.
6) After the paths are generated, another option to display the navigation path and save it (as .csv file) can also be selected.
7) The user will also have the ability to save the keyboard layout (as a .al file) if they have modified the initial loaded virtual keyboard.

Overview

The purpose of this program is to demonstrate the extent of Object Oriented Programming only using self-built Abstract Data Structures/Types (ADTs). With a firm understanding of Stacks, Queues, Linked Lists and Graph Data Structures, this also utilises the class relationships and inheritances from one another to create a simulation that is fully run on self-built ADTs. The simulation showcases the navigation of a virtual keyboard stored in a Graph Data Structure based on the input of strings by the user. It explores how objects stored in memory can be efficiently navigated to and allows for ease of modification of data.

Description of Classes

DSALinkedList

The Linked List Data Structure implemented here is a doubly linked list with a head and a tail pointer. The nodes it points to are made from a different class called DSAListNode. The methods in this class are vital to not only navigating through our Linked List but also adding and removing nodes from any position in the Linked List. As you are unable to jump to a specific node the class methods intricately maintain the link between each and all nodes to ensure integrity

DSAListNode

DSAListNode not only contains a value as its attribute but also two pointers to the previous and next node. With every node pointing to its neighbours(points to none if it is the head/tail) in the Linked List chain, it ensures the integrity of the Linked List Data Structure

DSAStack

DSAStack is a composition of DSALinkedList that only requires a top end pointer. Its goal of Last In First Out can be accomplished with the inherited methods from DSALinkedList.

DSAQueue

Our Queue is also a composition of DSALinkedList that has a head and tail pointer just much like a list. It's functionality of enqueuing to the tail and dequeuing from the head is completely inherited from the methods of DSALinkedList

DSAGraph

The Graph class is a Graph Data Structure that consists of vertices and edges in a graph. The constructor only contains a private Linked List of vertices represented by instances of DSAGraph Node. In order to access these vertices the accessor methods uses the DSALinkedList functionality

DSAGraphNode

The Graph node class is a Graph Vertex that contains the attributes label and links. Label is simply a string data that is used as a key for access therefore each label has to be unique from one another. The links are simply the edges that connect to other vertices. It is represented in a Linked List only storing the labels of other vertices it is connected to.

Possible Classes that could have been implemented

A Graph Edge class would have been useful to keep track of all existing edges in our Graph instance. However when using an Adjacency List implementation it was more convenient to acquire edges that can be stored within the GraphNode class. It resulted in fewer class interdependence which also means less memory usage and overall quicker computing execution speed (as we would load fewer classes in memory; As all classes except DSALinkedList are already partly composed of DSALinkedList having fewer classes is definitely beneficial). Another disadvantage of the GraphEdge class is that it won't be able to be accessed/executed directly without the functionality of the parent class Graph. An additional class that would also have been useful is a class that keeps track of all paths between each and every vertex in the graph. It would reduce the need for continuously calling a path searching function each time the program was executed. However it also comes with similar drawbacks in that it cannot be directly called without the presence of a parent class instance.

Justification of Decisions

Use of DSAGraph to create a Virtual Keyboard

The visual layout of a virtual keyboard is essentially the same as a 2D grid. Just like practical test 3 we have done in COMP5005 Fundamentals of Programming, movement of objects within a space and interaction with surrounding neighbours. It makes sense that a graph data structure can be used to mimic location of objects and its surrounding neighbours. Each object in space can be represented with a graph vertex and the links to surrounding neighbours can be represented with a graph edge.

Use of DSALinkedList to store a list of vertices and edges

As opposed to using an array, our Linked List Data Structure has the advantage of being flexible with list size. It does not require a fixed size to begin with and there is no size limit hindering it from creating a great deal of nodes. WIth our vertices stored in a Linked List we may face time complexity issues as we need to iterate through our list from the head node everytime we search through it. But our worst time complexity is only Big-O(n) which is decent enough for a virtual keyboard application. Also each vertex can have a maximum of 4 neighbours (with the exception of our space bar key) there is very little impact on time complexity.

Use of Breadth First Search Algorithm for optimal pathfinding

I **reworked my BFS Search algorithm** from my COMP5005 Assignment to **work with my self-created ADTS**. My original BFS Search code was used in my simulation assignment featuring Pac-Man and the Ghost gang. The code works out the shortest path Pac-man takes to the cherry (as well as the shortest path for the ghosts to get to Pac-Man) by avoiding obstacles. **I achieved this using a Graph Data Structure (inbuilt Python ADT) for the FOP Assignment.** I saw the potential to reuse this algorithm in determining the shortest path between keys on a keyboard therefore I decided to reuse the algorithm but made drastic changes to only work with my self-bulit ADTs.

Breadth First Search in a graph application travels to all its immediate surrounding neighbours before moving onto the next vertex. Therefore when the target key is reached, the number of steps taken is equal to the number of linking edges traversed. With Depth FIrst Search, although it does not waste time by traversing every neighbour each time it moves to a new key, it will keep traversing each edge until the path ends or until the target key is reached. In other words the path it returns will be deemed the shortest path in terms of number of edges traversed.

Use of Depth First Search Algorithm for non-optimal pathfinding

Depth First Search keeps travelling in a single path until it finds its target (or stops when all nodes have been traversed if target doesn't exist). In a complex graph data structure such as our virtual keyboard, it would result in multiple paths of varying length if it chooses edges in no particular order. Therefore It will return a path of any given length as long as it contains its target key.

Returning empty DSALinkedList if no traverse is needed or if target is not found

To keep track of the number of edges traversed in our virtual keyboard, each pathway is stored in a LinkedList which contains the linked vertices(from start to target). If the start and target are the same vertex no traverse is needed which is represented with an empty pathway. If the target is not found it means that it doesn't exist in the graph therefore it can also be represented with an empty pathway.

Loading Keyboard file

The .al file will allow for more than 2 vertices to be stored per line as opposed to the provided .al files from prac06.
Whatever number of vertices on any given line it just means that the first vertex is adjacent to all subsequent vertices

Saving keyboard

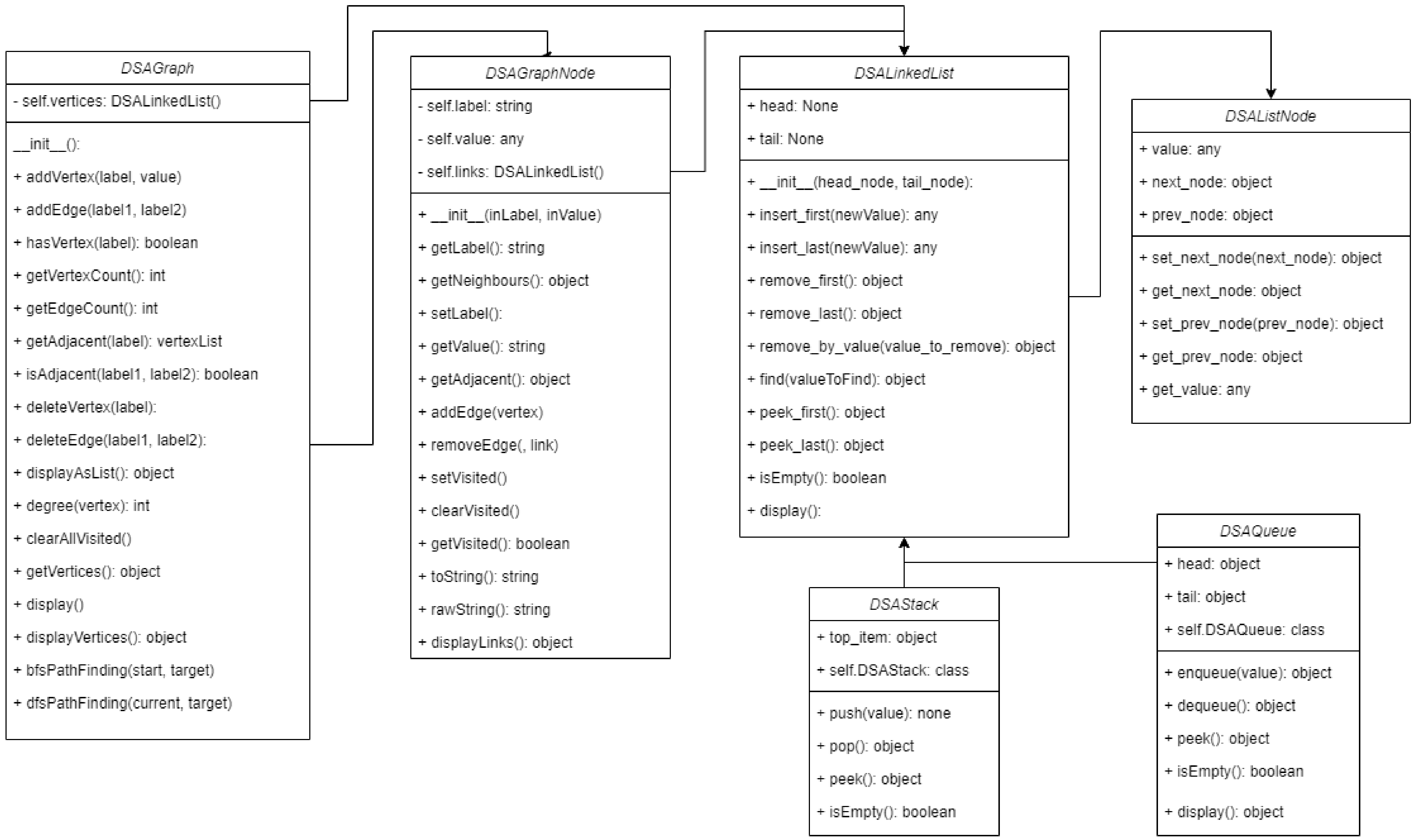
Initially the idea was to save the virtual keyboard with file serialisation so that the memory object can be loaded for immediate use again in its most recent state of use. However I chose to save the keyboard's layout to an abstract Adjacency List format instead. This approach encourages more modularity and the ease of sharing/exporting the keyboard layout for further use and extension in other applications.

Modified Switch Keyboard layout

WIth the provided images it raised the issue of completely replicating the keyboard layout key for key (as well as position for position). As the Graph implementation I am using does not have a value attribute I am using the labels as not just their key but also value.

Hence there would be the issue of having duplicate labels. As a Graph Data Structure relies on their keys being unique (which I'm representing with labels) I modified the two layouts to be distinct, one being alphanumeric and the other symbols. I also had a workaround using uppercase letters which was simply making our pathway go to the uppercase key each time it needed to capitalise a letter much like pressing the shift key when typing. This meant I did not have to create vertices to include the uppercase alphanumeric keyboard which helped to lessen the structural complexity and size of the graph. (Scenario 3 includes a visual of how I perceived the graph to be.

UML Class diagram



Traceability Matrix

Feature	Code	Testing
0. Modes and Menu		
0.1 Interactive Mode	keyMeUp.py | main() | lines 239 - 245	[PASSED] Run program with interactive mode options
0.2 Silent Mode	keyMeUp.py | lines 246-282	[PASSED] Run program with silent mode options and input file on command line
0.3 Usage	keyMeUp.py	[PASSED] Run program with no options
0.4 Menu Operation	keyMeUp.py | main() | lines 170-237	[PASSED] Run program with navigation by input
1. Load Data		
1.1 Load keyboard file	keyMeUp.py | readGraphFromFile() | lines 56-68	[PASSED] opens .al file, reading in lines, assigns all labels read in as vertices. Connects all labels (except the first) to the first label with edges
1.2 Load string file	keyMeUp.py | readStrFromFile() | lines 70-74	[PASSED] opens csv file reading a single line as a string
2. Graph Operations		
2.1 Add Vertex	GraphClasses.py | DSAGraph() |	[PASSED] Inserts an instance of DSAGraphNode into

	addVertex( ) | lines 77-80	the Vertices Linked List
2.2 Add Edge	GraphClasses.py | DSAGraph() | addEdge( ) | lines 105-112	[PASSED] Retrieves both vertices then adds its label to each other's Linked List
2.3 Delete Vertex	GraphClasses.py | DSAGraph() | deleteVertex( ) | lines 85-86	[PASSED] Deletes the labelled key from the Linked List stored in the graph class Then accesses its neighbours and deletes itself from its neighbour's Linked Lists
2.4 Delete Edge	GraphClasses.py | DSAGraph() | deleteEdge() | lines 114-117	[PASSED] Retrieves both vertices that are connected by an edge then deletes its label from each other's Linked List
2.5 Find Vertex	keyMeUp.py | vertexOperations() | lines 78-80	[PASSED] Retrieves the vertex by searching through the Inherited Linked List ADT by vertex label
2.6 Find Edge	keyMeUp.py | edgeOperations() | lines 98-103	[PASSED] Retrieves both vertices and checks if their labels exist in each other's links
2.7 Update Vertex	keyMeUp.py | vertexOperations() | lines 88-94	[PASSED] Relabel a vertex by deleting setting its label to a user defined one, then removes the old label from its neighbours links and adds the new label
2.8 Update Edge	keyMeUp.py | edgeOperations() | lines 118-126	[PASSED] Relinks a vertex's edge to another vertex. Retrieve the vertex, select edges to relink from/to, deletes and adds their labels to the links of the desired vertices

2.9 Display Adjacency List	GraphClasses.py | DSAGraph() | displayAsList() | lines 102-103	[PASSED] Accesses vertices using a loop and prints out their label and links
2.10 Breadth First Search Pathfinding	GraphClasses.py | DSAGraph() | bfsPathFinding() | lines 154-176	[PASSED] Returns the shortest path from start vertex to target vertex using a Breadth First Search approach
2.11 Depth First Search Pathfinding	GraphClasses.py | DSAGraph() | dfsPathFinding() | lines 178-193	[FAILED] Maximum Recursion depth exceeded error raised
3. Save Data		
3.1 Save pathway	keyMeUp.py | savePathway() | lines 165-170	[PASSED] Save the pathway taken into a .csv file
3.2 Save Keyboard	keyMeUp.py | saveKeyboardAsFile() | lines 172-178	[PASSED] Save the graph used for the keyboard as an adjacency list that can be reread

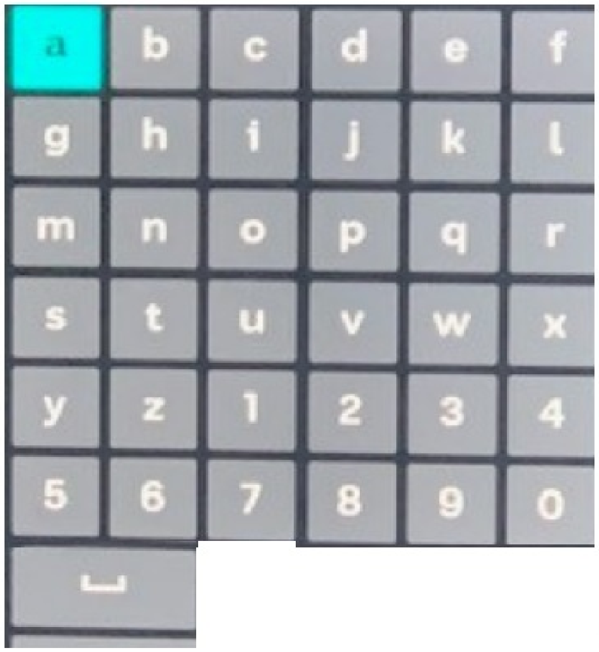
Showcase

Introduction

As movement is limited to the Von Neumann neighbourhood scheme, we have at most 4 edges to a vertex (with the exception of the space bar). This simulation is an attempt to show how graph pathfinding algorithms can be used to navigate a virtual keyboard. The three scenarios include edged keyboards that are bounded, wrap around keyboard that effectively “teleport” to the other opposite end of the grid and a more complex graph build that simulates how to “access” another keyboard layout (other layouts stored in the same graph).

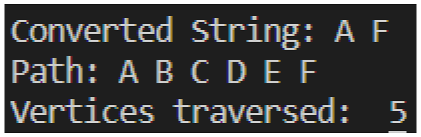
In order to reduce the occurrence of error raising and runtime errors with string input. Although all three keyboards showcased below are in lowercase, all keyboard layouts store letters in uppercase when being read into the graph. Scenario 1 and 2 are only meant to showcase basic graph movement on edged vs wrap keyboards therefore an uppercase implementation was not necessary. Therefore if an uppercase letter was input the validator function would notify the user that it “technically” doesn’t exist in the graph. Scenarios 3 however include the ability to input uppercase letters through a clever input converter to add the uppercase key as a necessary vertex to travel to before traversing to the desired key.

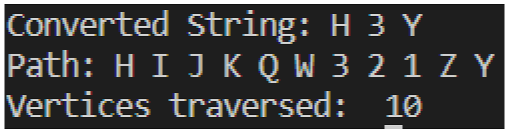
Scenario 1 - Edged Keyboard



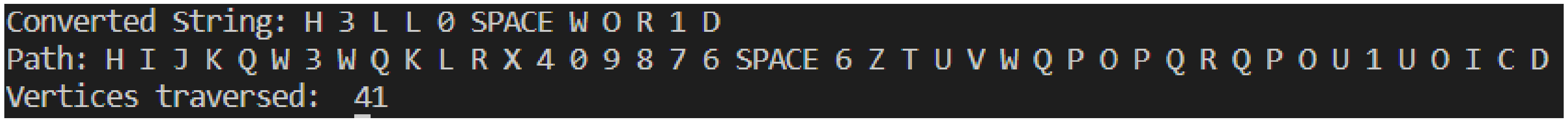
Using the abc layout keyboard limited to A-Z and 0-9 only with a (mainly) 6x6 grid type graph. Vertices at the edge will only contain 2-3 edges whilst the middle ones contain 4 edges. The output produced shows that paths will travel within the boundaries of the graph.

Input (-i, edged.al)	Output

af	 `Converted String: A F`
	`Path: A B C D E F`
	`Vertices traversed:  5`

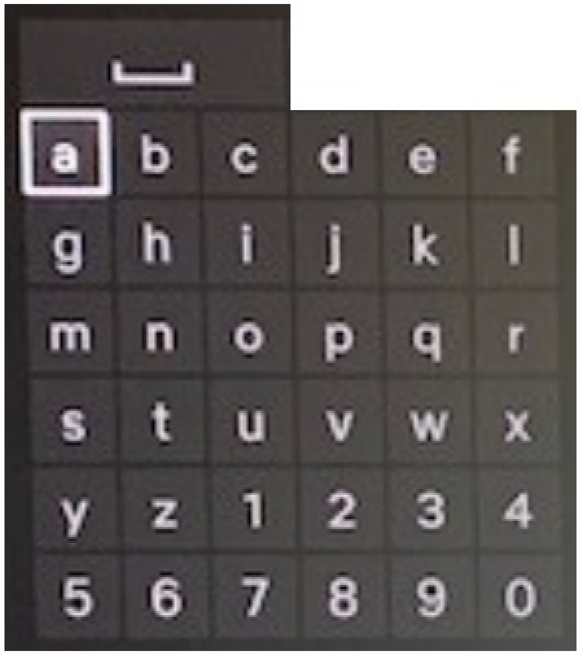
h3y	 `Converted String: H 3 Y`
	`Path: H I J K Q W 3 2 1 Z Y`
	`Vertices traversed:  10`

The above inputs are vertices at the/close to the edge of the keyboard. As you can see in the output the path way is within the boundaries of the keyboard/map. So with a two word combination below notice the converted input has converted a whitespace to the label 'SPACE' (This was to avoid any issues with reading keyboard files into our graph). The highlight here is how the path takes into account traversing to the space bar between words.

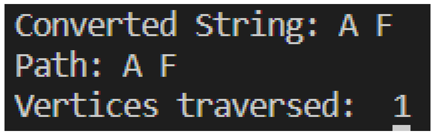
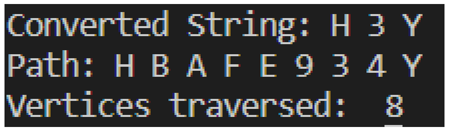
h3ll0 wor1d	 `Converted String: H 3 L L 0 SPACE W O R 1 D`
	`Path: H I J K Q W 3 W Q K L R X 4 0 9 8 7 6 SPACE 6 Z T U V W Q P O P Q R Q P O U 1 U O I C D`
	`Vertices traversed:  41`

Note that although the input is lowercase I have implemented a function to convert to uppercase to be compatible with the keyboard layout (that is stored in uppercase in the graph).

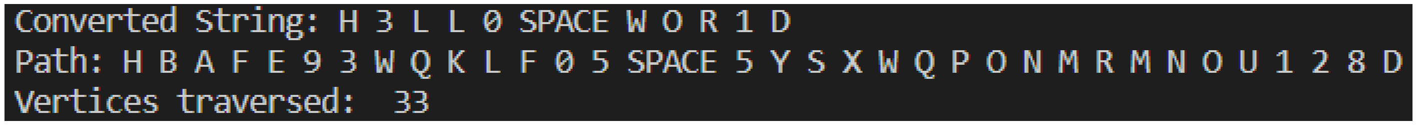
Scenario 2 - Wrap Around Keyboard



The layout of this keyboard is much the same as above with the only difference being the space bar located at the top of the keyboard. We will repeat the same inputs as above to highlight how the generated paths account for 'teleporting' to vertices on the opposite side of the keyboard/map.

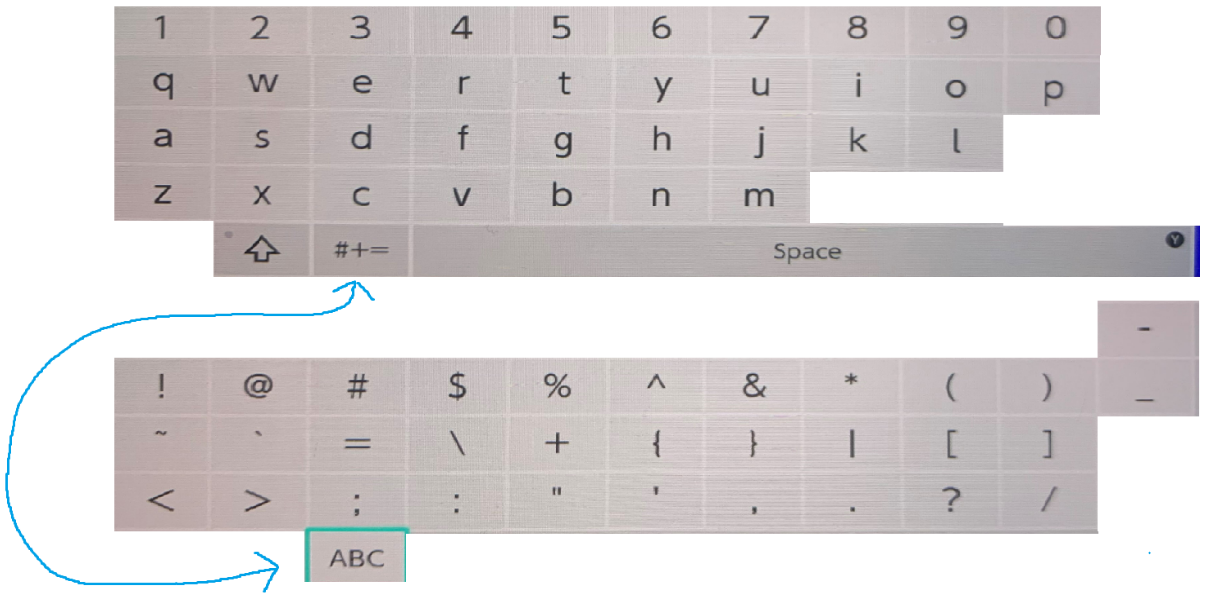
Input(-i, wrap.al)	Output
af	 `Converted String: A F` `Path: A F` `Vertices traversed:  1`
h3y	 `Converted String: H 3 Y` `Path: H B A F E 9 3 4 Y` `Vertices traversed:  8`

As you can see in the output the path taken from 'a' to 'f' simply meant a left move to 'teleport' to 'f' as 'f' is listed as a neighbour to 'a' in our graph. With the second input you can also see two 'teleport' actions taken, first in the upper left corner and then the bottom right corner.

h3ll0 wor1d	 `Converted String: H 3 L L 0 SPACE W O R 1 D` `Path: H B A F E 9 3 W Q K L F 0 5 SPACE 5 Y S X W Q P O N M R M N O U 1 2 8 D` `Vertices traversed:  33`

With the final input our wrap keyboard demonstrates that the total number of vertices traversed is lower overall lower to an edged keyboard.

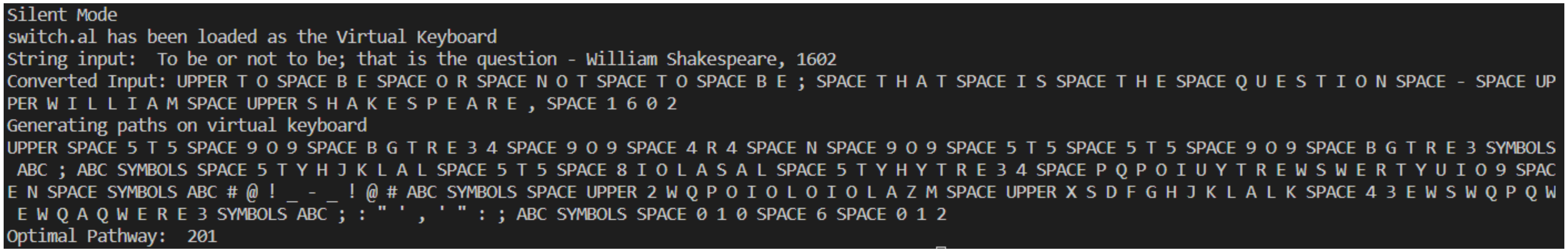
Scenario 3 - QWERTY Keyboard that deals with Uppercase Letters and Symbols



This is the familiar QWERTY layout used in the Nintendo Switch (images provided on Blackboard). I have made adjustments where labels are unique and do not repeat when switching from SYMBOLS to ABC keyboard (and vice versa). Although two keyboards suggest implementing two graphs, it is not the case here as I have simply read everything into the same graph. The way to access the alternate keyboard is connected by a single edge between the SYMBOLS and ABC key.

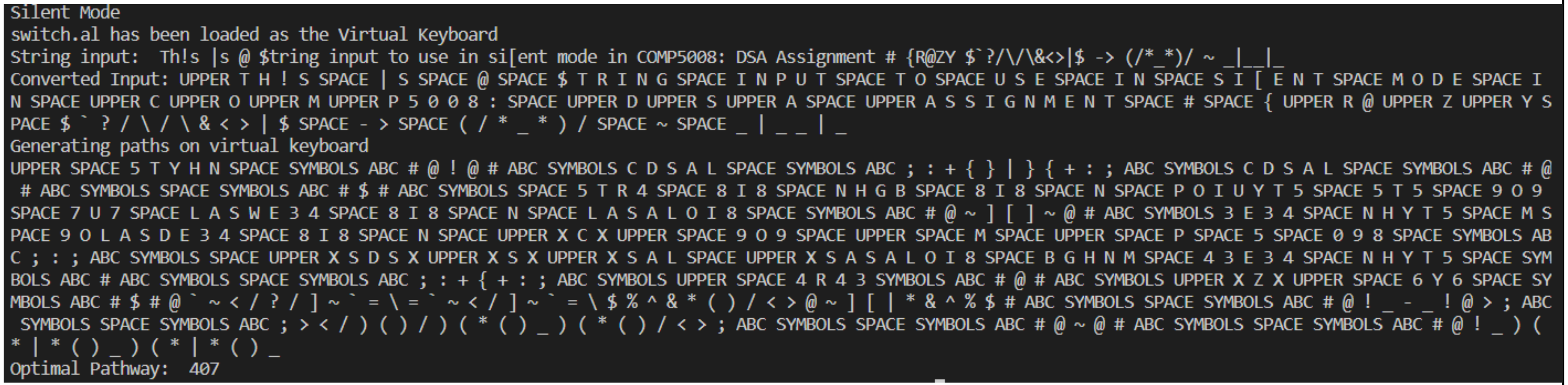
With the 'UPPER' and 'SYMBOLS'- 'ABC' keys being a requirement to traverse to before accessing the desired keys, it results in much greater number of total vertices traversed especially if the input string is long and contains many uppercase letters and symbols. As displayed below the pathways are significantly longer and although quite indiscernible the pathway taken is indeed the shortest pathway generated.

Input on command line: -s switch string1 bill



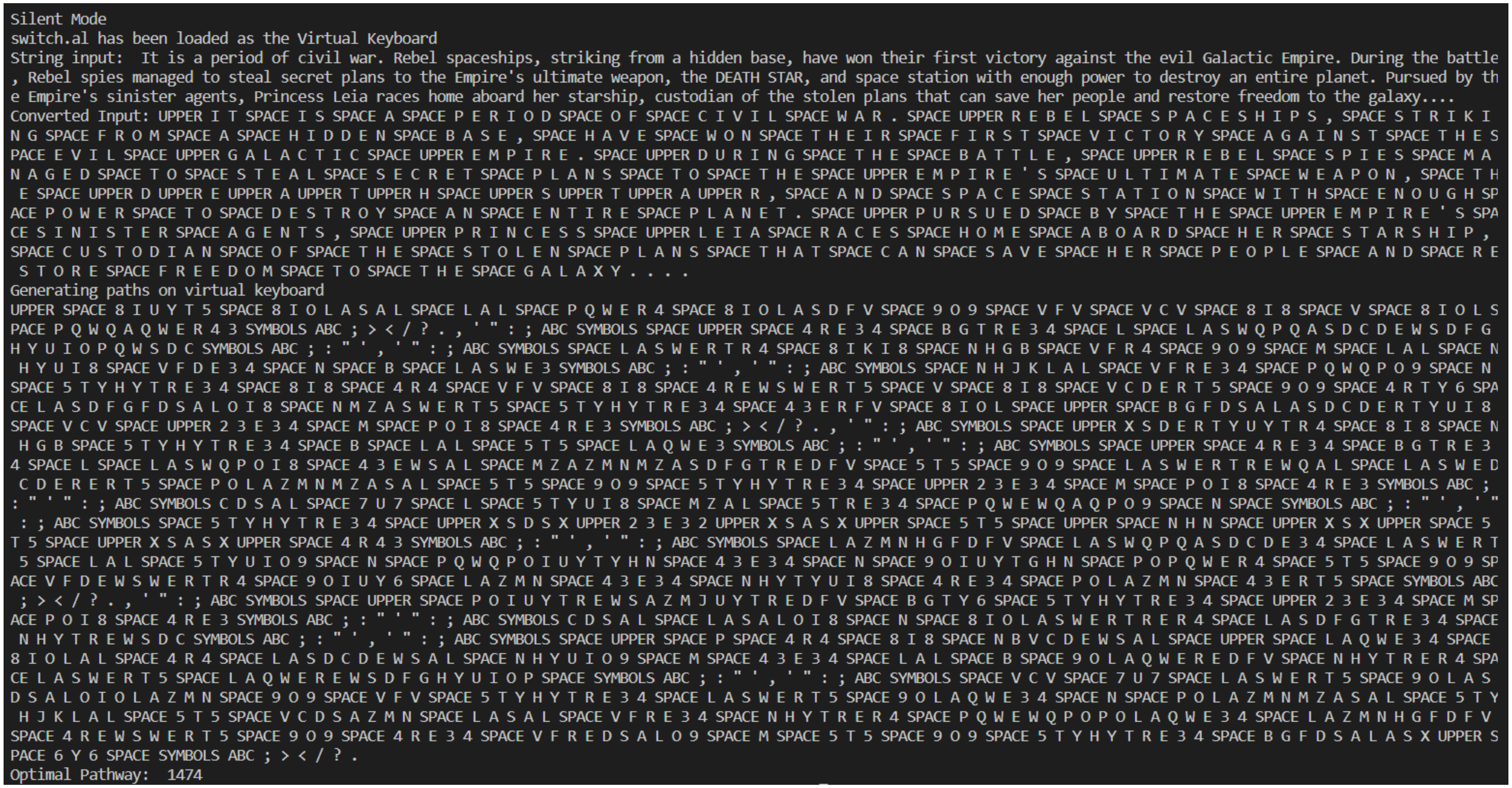
```
Silent Mode
switch.al has been loaded as the Virtual Keyboard
String input:  To be or not to be; that is the question - William Shakespeare, 1602
Converted Input: UPPER T O SPACE B E SPACE O R SPACE N O T SPACE T O SPACE B E ; SPACE T H A T SPACE I S SPACE T H E SPACE Q U E S T I O N SPACE - SPACE UP
PER W I L L I A M SPACE UPPER S H A K E S P E A R E , SPACE 1 6 0 2
Generating paths on virtual keyboard
UPPER SPACE 5 T 5 SPACE 9 O 9 SPACE B G T R E 3 4 SPACE 9 O 9 SPACE 4 R 4 SPACE N SPACE 9 O 9 SPACE 5 T 5 SPACE 5 T 5 SPACE 9 O 9 SPACE B G T R E 3 SYMBOLS
 ABC ; ABC SYMBOLS SPACE 5 T Y H J K L A L SPACE 5 T 5 SPACE 8 I O L A S A L SPACE 5 T Y H Y T R E 3 4 SPACE P Q P O I U Y T R E W S W E R T Y U I O 9 SPAC
E N SPACE SYMBOLS ABC # @ ! _ - _ ! @ # ABC SYMBOLS SPACE UPPER 2 W Q P O I O L O I O L A Z M SPACE UPPER X S D F G H J K L A L K SPACE 4 3 E W S W Q P Q W
 E W Q A Q W E R E 3 SYMBOLS ABC ; : " ' , ' " : ; ABC SYMBOLS SPACE 0 1 0 SPACE 6 SPACE 0 1 2
Optimal Pathway:  201
```

Input on command line: -s switch string2 emoticon



```
Silent Mode
switch.al has been loaded as the Virtual Keyboard
String input:  Th!s |s @ $tring input to use in si[ent mode in COMP5008: DSA Assignment # {R@ZY $`?/\/\&<>|$ -> (/*_*)/ ~ _|__|_
Converted Input: UPPER T H ! S SPACE | S SPACE @ SPACE $ T R I N G SPACE I N P U T SPACE T O SPACE U S E SPACE I N SPACE S I [ E N T SPACE M O D E SPACE I
N SPACE UPPER C UPPER O UPPER M UPPER P 5 0 0 8 : SPACE UPPER D UPPER S UPPER A SPACE UPPER A S S I G N M E N T SPACE # SPACE { UPPER R @ UPPER Z UPPER Y S
PACE $ ` ? / \ / \ & < > | $ SPACE - > SPACE ( / * _ * ) / SPACE ~ SPACE _ | _ _ | _
Generating paths on virtual keyboard
UPPER SPACE 5 T Y H N SPACE SYMBOLS ABC # @ ! @ # ABC SYMBOLS C D S A L SPACE SYMBOLS ABC ; : + { } | } { + : ; ABC SYMBOLS C D S A L SPACE SYMBOLS ABC # @
 # ABC SYMBOLS SPACE SYMBOLS ABC # $ # ABC SYMBOLS SPACE 5 T R 4 SPACE 8 I 8 SPACE N H G B SPACE 8 I 8 SPACE N SPACE P O I U Y T 5 SPACE 5 T 5 SPACE 9 O 9
SPACE 7 U 7 SPACE L A S W E 3 4 SPACE 8 I 8 SPACE N SPACE L A S A L O I 8 SPACE SYMBOLS ABC # @ ~ ] [ ] ~ @ # ABC SYMBOLS 3 E 3 4 SPACE N H Y T 5 SPACE M S
PACE 9 O L A S D E 3 4 SPACE 8 I 8 SPACE N SPACE UPPER X C X UPPER SPACE 9 O 9 SPACE UPPER SPACE M SPACE UPPER SPACE P SPACE 5 SPACE 0 9 8 SPACE SYMBOLS AB
C ; : ; ABC SYMBOLS SPACE UPPER X S D S X UPPER X S X UPPER X S A L SPACE UPPER X S A S A L O I 8 SPACE B G H N M SPACE 4 3 E 3 4 SPACE N H Y T 5 SPACE SYM
BOLS ABC # ABC SYMBOLS SPACE SYMBOLS ABC ; : + { + : ; ABC SYMBOLS UPPER SPACE 4 R 4 3 SYMBOLS ABC # @ # ABC SYMBOLS UPPER X Z X UPPER SPACE 6 Y 6 SPACE SY
MBOLS ABC # $ # @ ` ~ < / ? / ] ~ ` = \ = ` ~ < / ] ~ ` = \ $ % ^ & * ( ) / < > @ ~ ] [ | * & ^ % $ # ABC SYMBOLS SPACE SYMBOLS ABC # @ ! _ - _ ! @ > ; ABC
 SYMBOLS SPACE SYMBOLS ABC ; > < / ) ( ) / ) ( * ( ) _ ) ( * ( ) / < > ; ABC SYMBOLS SPACE SYMBOLS ABC # @ ~ @ # ABC SYMBOLS SPACE SYMBOLS ABC # @ ! _ ) (
* | * ( ) _ ) ( * | * ( ) _
Optimal Pathway:  407
```

Input on command line: -s switch string3 crawl



```
Silent Mode
switch.al has been loaded as the Virtual Keyboard
String input:  It is a period of civil war. Rebel spaceships, striking from a hidden base, have won their first victory against the evil Galactic Empire. During the battle
, Rebel spies managed to steal secret plans to the Empire's ultimate weapon, the DEATH STAR, and space station with enough power to destroy an entire planet. Pursued by th
e Empire's sinister agents, Princess Leia races home aboard her starship, custodian of the stolen plans that can save her people and restore freedom to the galaxy....
Converted Input: UPPER I T SPACE I S SPACE A SPACE P E R I O D SPACE O F SPACE C I V I L SPACE W A R . SPACE UPPER R E B E L SPACE S P A C E S H I P S , SPACE S T R I K I
N G SPACE F R O M SPACE A SPACE H I D D E N SPACE B A S E , SPACE H A V E SPACE W O N SPACE T H E I R SPACE F I R S T SPACE V I C T O R Y SPACE A G A I N S T SPACE T H E S
PACE E V I L SPACE UPPER G A L A C T I C SPACE UPPER E M P I R E . SPACE UPPER D U R I N G SPACE T H E SPACE B A T T L E , SPACE UPPER R E B E L SPACE S P I E S SPACE M A
N A G E D SPACE T O SPACE S T E A L SPACE S E C R E T SPACE P L A N S SPACE T O SPACE T H E SPACE UPPER E M P I R E ' S SPACE U L T I M A T E SPACE W E A P O N , SPACE T H
 E SPACE UPPER D UPPER E UPPER A UPPER T UPPER H SPACE UPPER S UPPER T UPPER A UPPER R , SPACE A N D SPACE S P A C E SPACE S T A T I O N SPACE W I T H SPACE E N O U G H SP
ACE P O W E R SPACE T O SPACE D E S T R O Y SPACE A N SPACE E N T I R E SPACE P L A N E T . SPACE UPPER P U R S U E D SPACE B Y SPACE T H E SPACE UPPER E M P I R E ' S SPA
CE S I N I S T E R SPACE A G E N T S , SPACE UPPER P R I N C E S S SPACE UPPER L E I A SPACE R A C E S SPACE H O M E SPACE A B O A R D SPACE H E R SPACE S T A R S H I P ,
SPACE C U S T O D I A N SPACE O F SPACE T H E SPACE S T O L E N SPACE P L A N S SPACE T H A T SPACE C A N SPACE S A V E SPACE H E R SPACE P E O P L E SPACE A N D SPACE R E
 S T O R E SPACE F R E E D O M SPACE T O SPACE T H E SPACE G A L A X Y . . . .
Generating paths on virtual keyboard
UPPER SPACE 8 I U Y T 5 SPACE 8 I O L A S A L SPACE L A L SPACE P Q W E R 4 SPACE 8 I O L A S D F V SPACE 9 O 9 SPACE V F V SPACE V C V SPACE 8 I 8 SPACE V SPACE 8 I O L S
PACE P Q W Q A Q W E R 4 3 SYMBOLS ABC ; > < / ? . , ' " : ; ABC SYMBOLS SPACE UPPER SPACE 4 R E 3 4 SPACE B G T R E 3 4 SPACE L SPACE L A S W Q P Q A S D C D E W S D F G
H Y U I O P Q W S D C SYMBOLS ABC ; : " ' , ' " : ; ABC SYMBOLS SPACE L A S W E R T R 4 SPACE 8 I K I 8 SPACE N H G B SPACE V F R 4 SPACE 9 O 9 SPACE M SPACE L A L SPACE N
 H Y U I 8 SPACE V F D E 3 4 SPACE N SPACE B SPACE L A S W E 3 SYMBOLS ABC ; : " ' , ' " : ; ABC SYMBOLS SPACE N H J K L A L SPACE V F R E 3 4 SPACE P Q W Q P O 9 SPACE N
SPACE 5 T Y H Y T R E 3 4 SPACE 8 I 8 SPACE 4 R 4 SPACE V F V SPACE 8 I 8 SPACE 4 R E W S W E R T 5 SPACE V SPACE 8 I 8 SPACE V C D E R T 5 SPACE 9 O 9 SPACE 4 R T Y 6 SPA
CE L A S D F G F D S A L O I 8 SPACE N M Z A S W E R T 5 SPACE 5 T Y H Y T R E 3 4 SPACE 4 3 E R F V SPACE 8 I O L SPACE UPPER SPACE B G F D S A L A S D C D E R T Y U I 8
SPACE V C V SPACE UPPER 2 3 E 3 4 SPACE M SPACE P O I 8 SPACE 4 R E 3 SYMBOLS ABC ; > < / ? . , ' " : ; ABC SYMBOLS SPACE UPPER X S D E R T Y U Y T R 4 SPACE 8 I 8 SPACE N
 H G B SPACE 5 T Y H Y T R E 3 4 SPACE B SPACE L A L SPACE 5 T 5 SPACE L A Q W E 3 SYMBOLS ABC ; : " ' , ' " : ; ABC SYMBOLS SPACE UPPER SPACE 4 R E 3 4 SPACE B G T R E 3
4 SPACE L SPACE L A S W Q P O I 8 SPACE 4 3 E W S A L SPACE M Z A Z M N M Z A S D F G T R E D F V SPACE 5 T 5 SPACE 9 O 9 SPACE L A S W E R T R E W Q A L SPACE L A S W E D
 C D E R E R T 5 SPACE P O L A Z M N M Z A S A L SPACE 5 T 5 SPACE 9 O 9 SPACE 5 T Y H Y T R E 3 4 SPACE UPPER 2 3 E 3 4 SPACE M SPACE P O I 8 SPACE 4 R E 3 SYMBOLS ABC ;
: " ' " : ; ABC SYMBOLS C D S A L SPACE 7 U 7 SPACE L SPACE 5 T Y U I 8 SPACE M Z A L SPACE 5 T R E 3 4 SPACE P Q W E W Q A Q P O 9 SPACE N SPACE SYMBOLS ABC ; : " ' , ' "
 : ; ABC SYMBOLS SPACE 5 T Y H Y T R E 3 4 SPACE UPPER X S D S X UPPER 2 3 E 3 2 UPPER X S A S X UPPER SPACE 5 T 5 SPACE UPPER SPACE N H N SPACE UPPER X S X UPPER SPACE 5
T 5 SPACE UPPER X S A S X UPPER SPACE 4 R 4 3 SYMBOLS ABC ; : " ' , ' " : ; ABC SYMBOLS SPACE L A Z M N H G F D F V SPACE L A S W Q P Q A S D C D E 3 4 SPACE L A S W E R T
 5 SPACE L A L SPACE 5 T Y U I O 9 SPACE N SPACE P Q W Q P O I U Y T Y H N SPACE 4 3 E 3 4 SPACE N SPACE 9 O I U Y T G H N SPACE P O P Q W E R 4 SPACE 5 T 5 SPACE 9 O 9 SP
ACE V F D E W S W E R T R 4 SPACE 9 O I U Y 6 SPACE L A Z M N SPACE 4 3 E 3 4 SPACE N H Y T Y U I 8 SPACE 4 R E 3 4 SPACE P O L A Z M N SPACE 4 3 E R T 5 SPACE SYMBOLS ABC
 ; > < / ? . , ' " : ; ABC SYMBOLS SPACE UPPER SPACE P O I U Y T R E W S A Z M J U Y T R E D F V SPACE B G T Y 6 SPACE 5 T Y H Y T R E 3 4 SPACE UPPER 2 3 E 3 4 SPACE M SP
ACE P O I 8 SPACE 4 R E 3 SYMBOLS ABC ; : " ' " : ; ABC SYMBOLS C D S A L SPACE L A S A L O I 8 SPACE N SPACE 8 I O L A S W E R T R E R 4 SPACE L A S D F G T R E 3 4 SPACE
 N H Y T R E W S D C SYMBOLS ABC ; : " ' , ' " : ; ABC SYMBOLS SPACE UPPER SPACE P SPACE 4 R 4 SPACE 8 I 8 SPACE N B V C D E W S A L SPACE UPPER SPACE L A Q W E 3 4 SPACE
8 I O L A L SPACE 4 R 4 SPACE L A S D C D E W S A L SPACE N H Y U I O 9 SPACE M SPACE 4 3 E 3 4 SPACE L A L SPACE B SPACE 9 O L A Q W E R E D F V SPACE N H Y T R E R 4 SPA
CE L A S W E R T 5 SPACE L A Q W E R E W S D F G H Y U I O P SPACE SYMBOLS ABC ; : " ' , ' " : ; ABC SYMBOLS SPACE V C V SPACE 7 U 7 SPACE L A S W E R T 5 SPACE 9 O L A S
D S A L O I O L A Z M N SPACE 9 O 9 SPACE V F V SPACE 5 T Y H Y T R E 3 4 SPACE L A S W E R T 5 SPACE 9 O L A Q W E 3 4 SPACE N SPACE P O L A Z M N M Z A S A L SPACE 5 T Y
 H J K L A L SPACE 5 T 5 SPACE V C D S A Z M N SPACE L A S A L SPACE V F R E 3 4 SPACE N H Y T R E R 4 SPACE P Q W E W Q P O P O L A Q W E 3 4 SPACE L A Z M N H G F D F V
SPACE 4 R E W S W E R T 5 SPACE 9 O 9 SPACE 4 R E 3 4 SPACE V F R E D S A L O 9 SPACE M SPACE 5 T 5 SPACE 9 O 9 SPACE 5 T Y H Y T R E 3 4 SPACE B G F D S A L A S X UPPER S
PACE 6 Y 6 SPACE SYMBOLS ABC ; > < / ? .
Optimal Pathway:  1474
```

Conclusion and Future Work

In hindsight it definitely would have been more beneficial to have a value attribute for the vertex to avoid the label being used both as a key and value. This became an issue when implementing the three Nintendo Switch keyboards into one graph as each keyboard would contain some of the same keys from another. As a keyboard layout looks very similar to a 2D grid, perhaps the label/key could be a coordinate and multiple value attributes could have been added to each coordinate/key. For example, the key representing the letter q could have self.valueLower = ‘q’, self.valueUpper = ‘Q’ and self.valueSymbol = ‘!’. This would greatly reduce the size and number of vertices in our graph and increase its flexibility. To access the uppercase characters or symbols, the simulation would only need to divert its path to the uppercase/symbol key and resume normal traversing in the keyboard again. However this implementation would hinder the functionality of the pathfinding algorithm which depends on knowing which vertices/keys it has already visited. For example if the simulation was currently on the lowercase keyboard and needed to go to the uppercase value of the letter ‘c’ but it had already visited that key whilst travelling to the upper case key. It would result in the pathfinding algorithm wasting time by exploring every single key before eventually returning a null value.

A key reflection point was how robust and modular object oriented programming is when it comes to the data structures we built in the lead up to the assignment. It really helped to solidify the understanding of class relationships. When initially creating DSALinkedList I was pleased to learn that the previous Stack and Queue data structures was also essentially a Linked List with its methods only composing of the specific Linked Lists methods to maintain its structural usage. And with Graphs I got to experience firsthand how polymorphism and class aggregation works; since the graph vertices are stored in a Linked List the vertices will still exist in memory even if the graph gets deleted. Furthermore the Linked List contains instances of DSAGraphNode which absolutely blew my mind; objects that contain other objects. With the assignment essentially using three types of data structures with inheritance from each other, it allowed for code from one another to be reused.

With the exposure to using self built ADTs i can see potential in the future for creating multiple variations of graph based networks. The combination of Graphs and Linked Lists to:

1. create a social network application that connects users based on similar interests.

   a. The graph can connect the user to a network of other users that all like a particular music artist which then suggests a friendship if there are also other similar interests (attends music festivals frequently, lives in the same metro area etc.)
   b. DSAGraphNode can represent each user and their attributes can implement other ADTs like a LinkedList to contain their Friends list

2. a fitness running app that provides real time pathway suggestions based on the user’s real time conditions and preferences
   a. A short run but with intensity could provide the user directions in taking uphill paths utilising weighted edges to represent the cost of traversing from one vertex to another (location/landmarks represented as nodes on a map)
   b. If the user’s heart rate is high for a prolonged period the algorithm can suggest directions to a location with a water fountain or fitness equipment for stretching
3. A collaboration using machine learning to build models that can predict the identity of the object given an image such as guessing what the animal in the picture is
   a. Each picture would be tagged with features to identify the classification of the animal (type of fur/skin, relative size, colour)
   b. The model would then search an extensive network of graphs to gather all the information it needs to classify a relatively small animal with short pointy ears, long bendy tail and patchy fur.
   c. A more refined graph searching algorithm like Djikstra’s, Kruskal’s or Prim’s algorithm would help in navigating directed, weighted graph networks to improve search efficiency and reduce time complexity.
4. A reverse dictionary application where the algorithm searches for a word with the input being the definition of the word.
   a. Users can type out as much as they know what the definition means (for e.g. very difficult to see, visualise or understand would result in the words: incomprehensible, concealed, disguised, confusing etc.)
   b. Words are the vertices and their definition the value, synonyms of words can belong in a highly connected graph/network
   c. The number of edges between vertices can be linked by how similar their definition is which can represent how weak/strong a connection to other words are.
   d. With this logic the algorithm can efficiently produce relevant synonyms/words that match the definition provided

Overall this assignment has taught me that alternative or more challenging problems can be tackled with having a strong foundation of Data Structures and Algorithms.