

AI AND MACHINE LEARNING BSC SOFTWARE ENGINEERING

ACCA6011

University of Wales Trinity Saint David

Siobhan Stephens - 1800709

Contents

1. Task 1.....	3
1.1. Design	4
1.1.1. Visual Representation of Map Paths.....	4
1.1.1. Reading Text File into a 2D Array.....	5
1.1.2. Adjacency Matrix for Shortest Path Algorithms.....	6
1.1.3. Reading 2D Array to find Vertices	10
1.1.4. Building an Adjacency Matrix.....	11
1.1.5. Finding the Shortest Path using Dijkstra's Algorithm	12
1.1.6. Depth First Search.....	12
1.1.7. Menu Options.....	13
1.2. User Guide	13
1.3. Test Schedule	13
1.1. Working Solution.....	13
1.2. Program Listing	13
2. Task 2 – K-Means Clustering Algorithm.....	14
2.1. Introduction.....	14
2.2. Algorithm.....	14
2.3. Approaches to Selecting Optimum K	15
2.3.1. Silhouette Analysis	15
2.3.2. Elbow Method Curve	16
2.4. Distance Measures	17
2.4.1. City Block (Manhattan)	17
2.4.2. Euclidean.....	17
2.4.3. Cosine	18
2.4.4. Correlation Distance	18
2.4.5. Performance	18
2.5. Evaluation	18
3. Task 3.....	19
3.1. K-Means Clustering Data.....	19
3.1.1. Given Centroids and Data Points	19
3.1.2. K-Means Algorithm	20
4. References.....	26
5. Appendix A – User Guide.....	28
5.1. Introduction.....	28

5.2.	Definitions.....	28
5.3.	Load a File	28
5.4.	Use The Menu.....	29
5.5.	Print Map (Option 1).....	30
5.6.	Open Map Walk (Option 2)	31
5.7.	Depth First Search (Option 3).....	32
5.8.	Dijkstra’s Shortest Path (Option 4).....	32
5.9.	Load a New File (Option 5)	35
5.10.	Exit (Option 6)	35
6.	Appendix B – Test Schedule.....	36
7.	Appendix C – Screenshots of Working Application.....	39
8.	Appendix D – Program Listing.....	47
8.1.	DepthFirstSearch.h.....	48
8.2.	DepthFirstSearch.cpp.....	49
8.3.	Dijkstra.h.....	51
8.4.	Dijkstra.cpp.....	52
8.5.	Print.h.....	54
8.6.	Print.cpp.....	55
8.7.	SearchUtility.h	56
8.8.	SearchUtility.cpp	57
8.9.	Main.cpp	79
8.10.	floor0.txt.....	83
8.11.	floor1.txt.....	84
8.12.	floor2.txt.....	85

1. Task 1

The task is to create a program that will systematically search a map, which is read in from a text (txt) file, represented with a fixed set of symbols.

The map is represented as follows:

- Wall – ‘*’ (asterisk)
- Sign – alpha character eg ‘A’, ‘B’, ‘C’ etc
- Corridor – ‘ ’ (blank space)
- Start – ‘S’

The given test map is shown below.

```
*****
S      A      *
*****  *****
*C*                      *
* *****  *
* *                      *
* *****  *****
*   B*          *D*
* *****  *****
* *          * *
* *****  *****
* *****          *
* *****  *****
* *****  *****
*E                      F*
*****
```

Figure 1. floor0.txt test map

1.1.Design

1.1.1. Visual Representation of Map Paths

Below is a visual representation of the paths, walls and signs within the map floor0.txt – the dead ends, intersections, corners and signs are easier to visualize for the design process in this format.

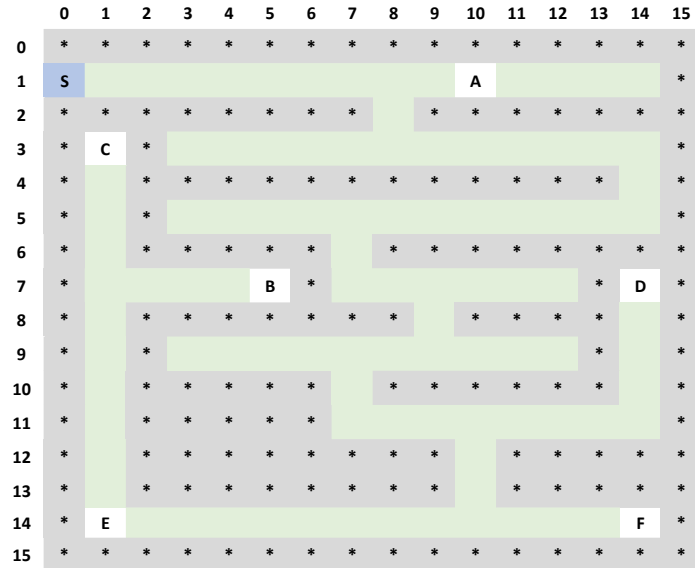


Figure 2. Visual representation of floor0.txt map

1.1.1. Reading Text File into a 2D Array

One of first problems within the solution is to read the map, represented as a multi-line string in a text file, into a 2D array so that it can be manipulated by the program.

The file size must be gathered (width and height of map) In order to declare a 2d array to store the map.

There is a requirement for map sizes to be greater than 16 x 16, and therefore, a while loop is used to continually ask the user for files that do not meet this criteria. In addition, the author decided to add a requirement than maps may not be larger than 9999 for the current application, this may be increased in future.

Therefore, the following pseudocode has been produced to outline the process.

```
array[][] read2DArrayFromFile():
    while (height < 16 || width < 16 || height > 9999 || width > 9999):
        fileName = "" //Clear filename in case of previous loop iteration
        output: Please enter a file name containing a map, including file extension, eg floor0.txt: //Ask user for file name
        input: fileName //Store user input in variable

        Open file
        if myfile.is_open():
            //First, loop through the file to gather the height and width of the map.
            //Height is calculated by incrementing a loop variable for each line of the file &
            //Width is calculated by gathering the length of the line, using string.length() method.
            while (getline(myfile, line)):
                width = line.length()
                height++

            map[height][width] //Declare new array map to hold the contents of the map file. Initialised to the height & width gathered above.

            count = 0;
            while (getline(myfile, line)):
                map[count] = line;
                count++ //Increase loop variable used above

            printMap(map)
            myfile.close() //Close file

            //Check file is correct size
            if (width > 16 || height < 16):
                Output: "Map must be greater than 16 in height and 16 in width. Please try another file."

            else if (height > 9999 || width > 9999):
                Output: "Map must not be greater than 9999 in height and 9999 in width. Please try another file."
            else:
                Output: "File Loaded! Map Size is " + width + " Wide by " + height + " High."

            return map //Return map containing map contents, to be used within calling function
        else return {} //Return empty - will trigger some exception handling inside the calling function
```

Figure 3. Pseudocode for reading file into a 2D array

1.1.2. Adjacency Matrix for Shortest Path Algorithms

In order to build an adjacency matrix to be used within a shortest path algorithm such as Dijkstra, the edges, edge weights and vertices must be discovered from within the map.

For the given task, it can be said that a vertex/node is one of the following:

- Sign – represented by an alpha character (S is the designated start node)
- Intersection - see definition below
- Dead End - see definition below
- Corner - see definition below

And an edge is a corridor, represented by a space ' '.

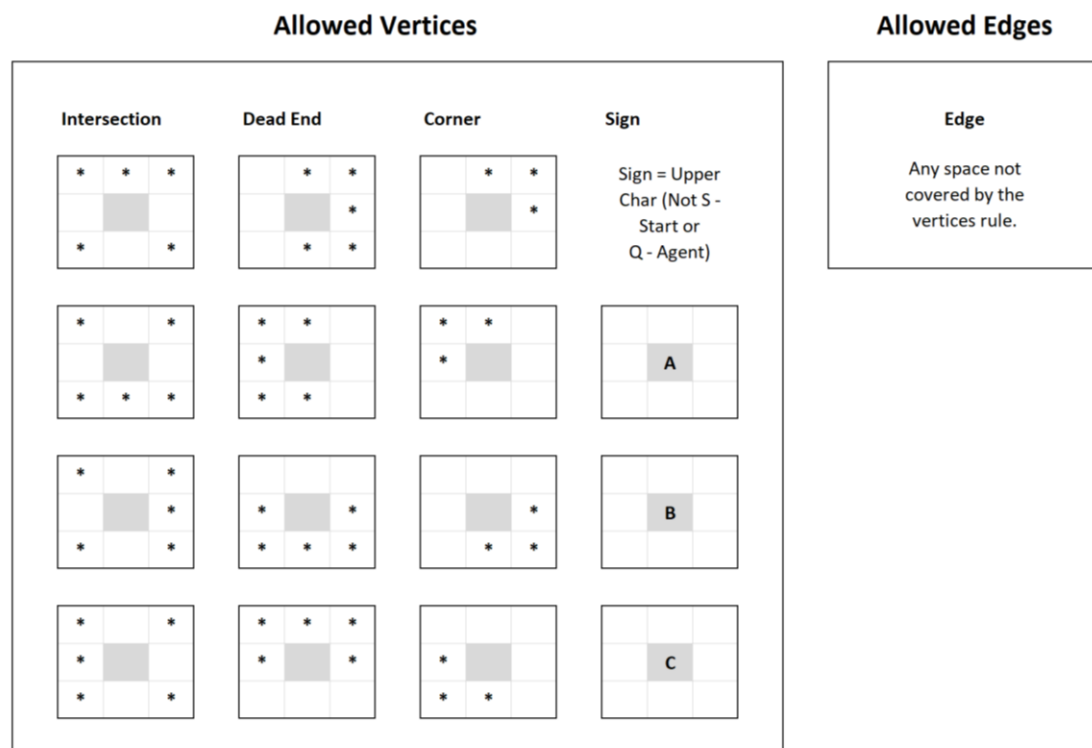


Figure 4. Rules for allowed Vertices and Edges – grey square represents current position being tested

Given the rules outlined, the visual representation in the figure below has been produced, labelled with an 'ID' for each node. The nodes are to be discovered within the application using a nested for-loop, and will be discovered from left to right and top to bottom. Using the map floor0.txt provided, 27 vertices have been identified.

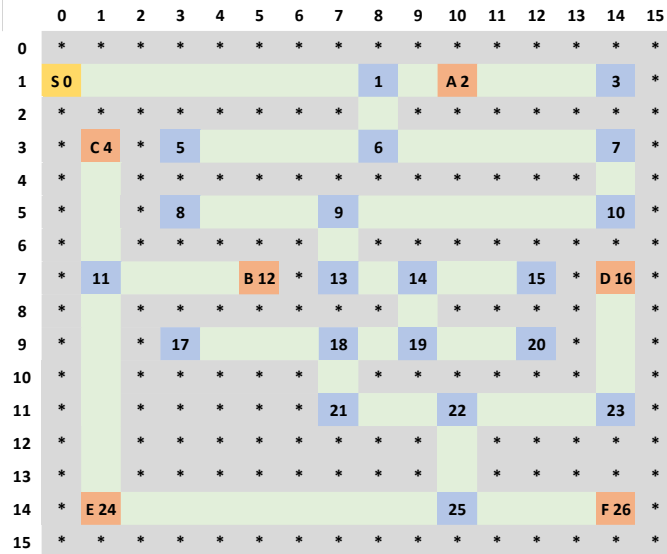


Figure 5. Map with Vertices (Blue & Orange – Signs Highlighted) and Edges (Green), including start node S (Yellow)

Next, a weighted graph has been produced to outline the adjacent vertices and edge costs between each, show in the figure below.

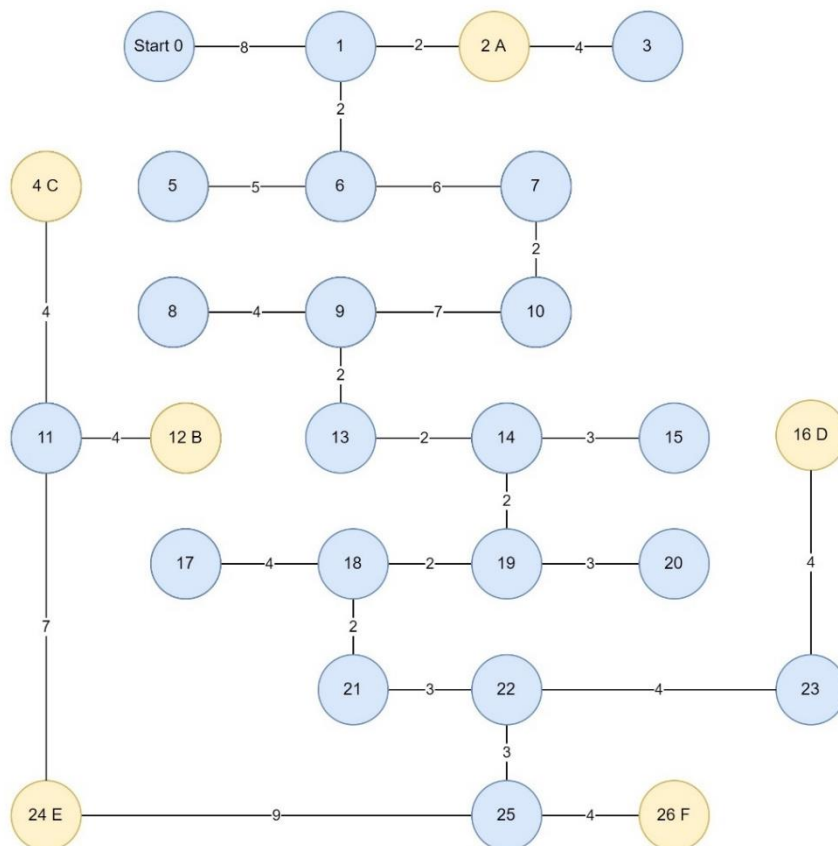


Figure 6. Undirected Weighted Graph for the Given Map

From the graph, the following adjacency matrix was created which can be used by shortest path algorithms.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
0	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	8	0	2	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	2	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	2	0	0	0	5	0	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	6	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	4	0	7	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	2	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	4	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	7	0	0
12	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	2	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	3	0	0	0	2	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	2	0	2	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	2	0	3	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	3	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	4	0	3	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	4	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	9	0	4	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0

Figure 7. Adjacency matrix for the given graph

	X	Y	S
0	0	1	S
1	8	1	
2	10	1	A
3	14	1	
4	1	3	C
5	3	3	
6	8	3	
7	14	3	
8	3	5	
9	7	5	
10	14	5	
11	1	7	
12	5	7	B
13	7	7	
14	9	7	
15	12	7	
16	14	7	D
17	3	9	
18	7	9	
19	9	9	
20	12	9	
21	7	11	
22	10	11	
23	14	11	
24	1	14	E
25	10	14	
26	14	14	F

Figure 8. X & Y Co-ordinates for each vertex in floor0.txt map including highlighted Sign nodes

1.1.3. Reading 2D Array to find Vertices

In order to build an adjacency matrix, vertices must first be listed by the program.

Vertices can be defined as: intersections; dead-ends; corners and signs, and edges can be defined as: corridors.

Vertices are to be stored in an array of x/y co-ordinates, for example, vertices[0] would contain {0,1}, which would be accessed in a manner such as vertices[0].x and vertices[0].y.

Following the rules outlined in a previous section, the following pseudocode has been created to represent checking for each of the rule cases.

```
2 wall = ''
3 corridor = ' '
4
5 upperL = current.x - 1, current.y - 1
6 upperC = current.x, current.y - 1
7 upperR = current.x + 1, current.y - 1
8 centreL = current.x - 1, current.y
9 centre = current.x, current.y
10 centreR = current.x + 1, current.y
11 lowerL = current.x - 1, current.y + 1
12 lowerC = current.x, current.y + 1
13 lowerR = current.x + 1, current.y + 1
14
15 //Test Vertice Rules beginning with the Start Node
16 if(centre == 'S'):
17     Add current position to list of vertices
18     //1
19     if (upperL == wall && upperC == wall && upperR == wall && centreL != wall && centre != wall && centreR != wall && lowerL == wall && lowerC != wall && lowerR == wall):
20         Add current position to list of vertices
21     //2
22     if (upperL == wall && upperC != wall && upperR == wall && centreL != wall && centre != wall && centreR != wall && lowerL == wall && lowerC == wall && lowerR == wall):
23         Add current position to list of vertices
24     //3
25     if(upperL == wall && upperC != wall && upperR == wall && centreL != wall && centre != wall && centreR == wall && lowerL == wall && lowerC != wall && lowerR == wall):
26         Add current position to list of vertices
27     //4
28     if(upperL == wall && upperC != wall && upperR == wall && centreL == wall && centre != wall && centreR != wall && lowerL == wall && lowerC != wall && lowerR == wall):
29         Add current position to list of vertices
30     //5
31     if(upperL != wall && upperC == wall && upperR == wall && centreL != wall && centre != wall && centreR == wall && lowerL != wall && lowerC == wall && lowerR == wall):
32         Add current position to list of vertices
33     //6
34     if(upperL == wall && upperC == wall && upperR != wall && centreL == wall && centre != wall && centreR != wall && lowerL == wall && lowerC == wall && lowerR != wall):
35         Add current position to list of vertices
36     //7
37     if(centreL == wall && centre != wall && centreR == wall && lowerL == wall && lowerC == wall && lowerR == wall):
38         Add current position to list of vertices
39     //8
40     if(upperL == wall && upperC == wall && upperR == wall && centreL == wall && centre != wall && centreR == wall && lowerL != wall && lowerC != wall && lowerR != wall):
41         Add current position to list of vertices
42     //9
43     if(upperC == wall && upperR == wall && centre != wall && centreR == wall):
44         Add current position to list of vertices
45     //10
46     if(upperL == wall && upperC == wall && centreL == wall && centre != wall):
47         Add current position to list of vertices
48     //11
49     if(centre != wall && centreR == wall && lowerC == wall && lowerR == wall):
50         Add current position to list of vertices
51     //12
52     if(centreL == wall && centre != wall && lowerL == wall && lowerC == wall):
53         Add current position to list of vertices
54     //13
55     if(isalpha(centre)):
56         Add current position to list of vertices
```

Figure 9. Pseudocode for detecting vertices within the 2D array representation of the map

1.1.4. Building an Adjacency Matrix

In order for the program to build an adjacency matrix, which will be represented as a 2d array with the rows and columns of equal size relating to the number of vertices, it must calculate the edge cost between vertices, by counting the number of steps taken between each vertex, as outlined in the weighted graph in a previous section.

The pseudocode for building an adjacency matrix is below.

```
2 int[][] buildAdjacencyMatrix(map[][], vertices[]):
3
4 //Declare an array to hold the adjacency matrix
5 int adjacencyMatrix[vertices.size()][vertices.size()]
6
7 //Loop through each vertex in the vertices array
8 for (currVertex = 0; currVertex < verticesArr.size(); currVertex):
9
10     edgeWeight = 0
11     currentPosition = currVertex.x, currVertex.y
12     adjacentNodeFound = false
13
14     //Look Right
15     while(!adjacentNodeFound and currentPosition.x < map.width - 1 and map[y][x] != wall:
16         edgeWeight ++
17         currentPosition.x = currentPosition.x + 1
18         //Check each item in the vertices array to see if the current co-ordinates exist
19         for (vertex = 0; vertex < vertices.size(); vertex++):
20             if (currentPosition.x == vertices[vertex].x and currentPosition.y == vertices[vertex].y):
21                 //Add the vertex ID (current vertex) and edge weight relative to the vertex found in the list (vertex)
22                 adjacencyMatrix[currentVertex][vertex] = edgeWeight
23                 //Set loop variable to be true so that it does not continue searching to the right
24                 adjacentNodeFound = true
25
26     edgeWeight = 0
27     currentPosition = currVertex.x, currVertex.y
28     adjacentNodeFound = false
29
30     //Look Left
31     while(!adjacentNodeFound and currentPosition.x > 0 and map[y][x] != wall:
32         edgeWeight ++
33         currentPosition.x = currentPosition.x - 1
34         //Check each item in the vertices array to see if the current co-ordinates exist
35         for (vertex = 0; vertex < vertices.size(); vertex++):
36             if (currentPosition.x == vertices[vertex].x and currentPosition.y == vertices[vertex].y):
37                 //Add the vertex ID (current vertex) and edge weight relative to the vertex found in the list (vertex)
38                 adjacencyMatrix[currentVertex][vertex] = edgeWeight
39                 //Set loop variable to be true so that it does not continue searching to the left
40                 adjacentNodeFound = true
41
42     edgeWeight = 0
43     currentPosition = currVertex.x, currVertex.y
44     adjacentNodeFound = false
45
46     //Look Upwards
47     while(!adjacentNodeFound and currentPosition.y > 0 and map[y][x] != wall:
48         edgeWeight ++
49         currentPosition.y = currentPosition.y - 1
50         //Check each item in the vertices array to see if the current co-ordinates exist
51         for (vertex = 0; vertex < vertices.size(); vertex++):
52             if (currentPosition.x == vertices[vertex].x and currentPosition.y == vertices[vertex].y):
53                 //Add the vertex ID (current vertex) and edge weight relative to the vertex found in the list (vertex)
54                 adjacencyMatrix[currentVertex][vertex] = edgeWeight
55                 //Set loop variable to be true so that it does not continue searching upwards
56                 adjacentNodeFound = true
57
58     edgeWeight = 0
59     currentPosition = currVertex.x, currVertex.y
60     adjacentNodeFound = false
61
62     //Look Downwards
63     while(!adjacentNodeFound and currentPosition.y < map.height - 1 and map[y][x] != wall:
64         edgeWeight ++
65         currentPosition.y = currentPosition.y + 1
66         //Check each item in the vertices array to see if the current co-ordinates exist
67         for (vertex = 0; vertex < vertices.size(); vertex++):
68             if (currentPosition.x == vertices[vertex].x and currentPosition.y == vertices[vertex].y):
69                 //Add the vertex ID (current vertex) and edge weight relative to the vertex found in the list (vertex)
70                 adjacencyMatrix[currentVertex][vertex] = edgeWeight
71                 //Set loop variable to be true so that it does not continue searching downwards
72                 adjacentNodeFound = true
73
74     return adjacencyMatrix
75
76
```

Figure 10. Pseudocode for building adjacency matrix

1.1.5. Finding the Shortest Path using Dijkstra's Algorithm

Dijkstra's algorithm was published by E. Dijkstra in 1956, and is used for finding the shortest path between a source vertex in a graph and a destination vertex [1]. Dijkstra's algorithm makes it possible to find the shortest path from a given source to all points in a graph at the same time and is therefore sometimes called the single-source shortest path [2].

The pseudocode for Dijkstra's algorithm is as follows [3]:

```
void dijkstra(Vertex s){
    Vertex v,w;

    Initialize s.dist = 0 and set dist of all other vertices to infinity

    while (there exist unknown vertices, find the one b with the smallest
    distance)

        b.known = true;

    for each a adjacent to b

        if (!a.known)

            if (b.dist + Cost_ba < a.dist)

                decrease(a.dist to= b.dist + Cost_ba);

                a.path = b;
```

Dijkstra's algorithm will be used within the solution to find the shortest route between the start node specified as 'S' and all the signs within the map, allowing the agent to reach every sign within the map using the least number of steps possible.

The algorithm will make use of an adjacency matrix representation of the map.

1.1.6. Depth First Search

Depth first search will also be included as part of the solution, the pseudocode for which is as follows:

```
Initialize stack.
Push the first element position (element at (0,0), row=0, column=0) to
- the stack
Now until the stack is not empty
    pop the position from the stack.
    Split it by "," to get the row index and column index and check
    if indexes are within the range of given matrix and have not vert
    been visited, if not then ignore it and pop the next position
    from the stack. If indexes are valid and not visited then print
    the element.
        • Mark the element as visited
        • Add the element positions from left, right, down and up
          from the current element into the stack.
```

Positions will be marked as visited by leaving a breadcrumb ‘.’.

On each iteration, the current position being searched will be marked as a ‘Q’ representing the agent searching the map, and the map will be printed for the user to see.

1.1.7. Menu Options

The program must allow the user to choose from a range of options, outlined below:

- Load New Map File
- Open Map Walk (traverse map at free will)
- Depth First Search
- Shortest Path Search (Dijkstra)

1.2. User Guide

See Appendix A for user guide.

1.3. Test Schedule

See Appendix B for full test schedule.

1.1. Working Solution

See Appendix C for screenshots of working solution.

1.2. Program Listing

See Appendix D for full code listing.

2. Task 2 – K-Means Clustering Algorithm

2.1.Introduction

Clustering is the task of segregating groups of similar and dissimilar features and assigning them into clusters. Clustering is widely used in Machine Learning, with numerous clustering algorithms available, including Mean Shift, Spectral Clustering, Mini-Batch and K-means.

K-means is an unsupervised algorithm often used to form clusters of large datasets due to its scalability, efficiency and simplicity [4]. As stated by Arora et al, “clustering plays a very vital role in exploring data, creating predictions and to overcome the anomalies in the data. As the data in real world is growing day by day so very large datasets with little or no background knowledge can be identified into interesting patterns with clustering” [5]. However, one of its drawbacks is the requirement for the number of clusters, K, to be specified before the algorithm is applied [6].

2.2.Algorithm

The K-Means algorithm aims to minimize the mean squared distance between each instance and its closest centroid, defined as inertia.

A generalized algorithm for K-means clustering can be outlined as follows [7]:

- Step 1: Accept the number of clusters K and the dataset to cluster as input values
- Step 2: Initialize the first K clusters that represent the centroid
- Take first k instances or
 - Take Random sampling of k elements
- Step 3: Calculate the arithmetic means of each cluster formed in the dataset.
- Step 4: K-means assigns each record in the dataset to only one of the initial clusters
- Each record is assigned to the nearest cluster using a measure of distance (e.g. Euclidean distance).
- Step 5: K-means re-assigns each record in the dataset to the most similar cluster and re-calculates the arithmetic mean of all the clusters in the dataset.
- Step 6: Once centroid values remain the same, stop.

2.3.Approaches to Selecting Optimum K

Finding the appropriate number of clusters for a given data set is generally a trial-and-error process [8]. A report by Steorts outlines that “reported studies on K-means clustering and its applications usually do not contain any explanation or justification for selecting particular values for K” [9].

2.3.1. Silhouette Analysis

Silhouette analysis quantifies the separation distance between clusters and is used to visualize which objects lie well within their cluster and which could be improved upon, allowing a view of the relative quality of the clusters and data [10]. Silhouette width may be used to select an appropriate number of clusters K [10].

It uses a measure range of [-1,1], where coefficients near +1 indicate that a sample is far away from neighboring clusters, a value of 0 indicates that the sample is on or very close to the decision boundary between two neighboring clusters and, negative values indicate that those samples might have been assigned to the wrong cluster [11].

A higher silhouette coefficient score relates to a model with better defined clusters. The score is calculated as follows:

For each sample, two scores are calculated:

- a: The mean distance between a sample and all other points in the same class.
- b: The mean distance between a sample and all other points in the next nearest cluster.

The silhouette coefficient for a single sample is then given as:

$$s = \frac{b - a}{\max(a, b)}$$

The silhouette score for K-clusters is the average of all these values [10].

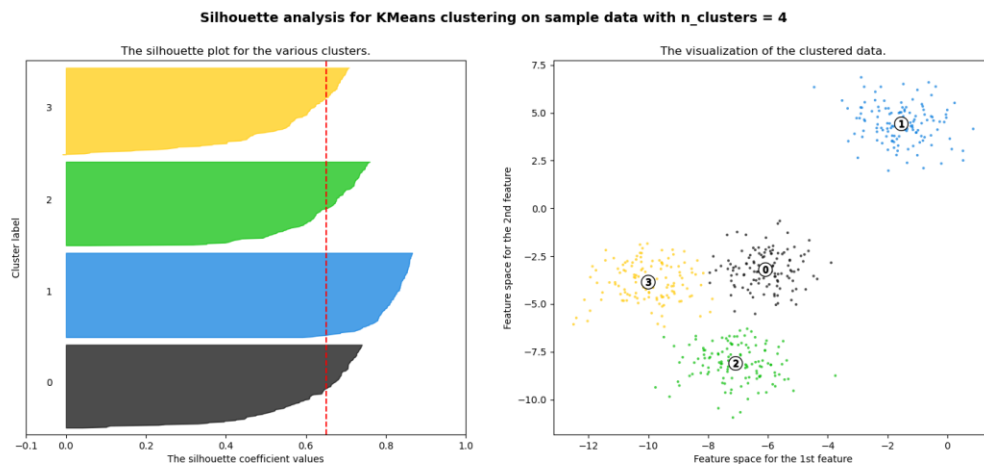


Figure 11. Example Silhouette Analysis Graphical Representation [12]

2.3.2. Elbow Method Curve

The elbow method is a visual method for identifying the optimal number of clusters based on the Within-Cluster Sum of Square (WCSS) or sum of squared errors (SSE) [13].

An example method is to start with $K=2$ and increase K by 1, calculating the new WCSS at each iteration. At some value for K , the WCSS will drop and reach a plateau when increased further. It is at this point the elbow can be identified. Further iterations and increments of K can cause the model to over-train and fit too closely to the dataset [13], therefore we are looking for the minimum number of clusters that provides us with reasonable inertia. An example elbow curve plot is shown below, along with the elbow point. The optimum number of K for this example is $K=5$.

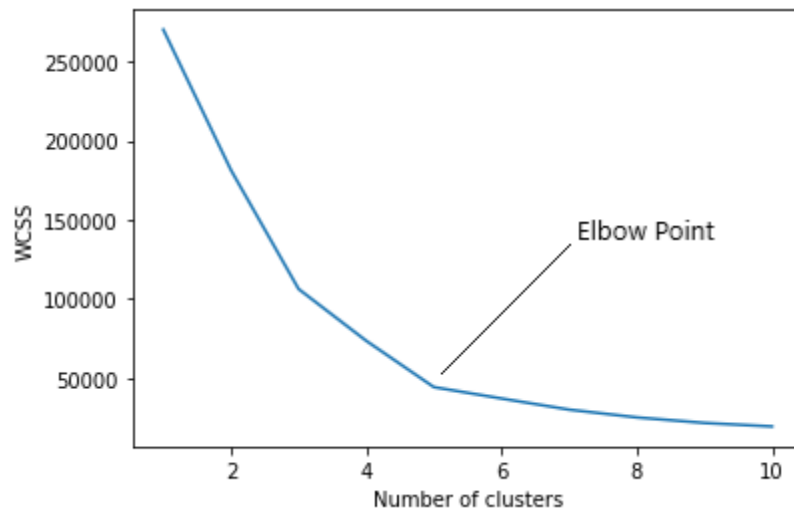


Figure 12. An example Elbow Curve plot

The main disadvantage with the elbow curve method is that the optimum point is subjective, as often many 'elbows' can be seen within the data, and therefore a less optimum K may be selected.

2.4.Distance Measures

Distance, or dissimilarity, and similarity are the basis of clustering algorithms. Distance is used for quantitative data, whereas similarity is preferred when dealing with qualitative data features [1].

The K-means algorithm relies on the calculation of distance between each point of the dataset and each centroid, and based on these values, datapoints are assigned to the closest centroid. Therefore, the distance calculation is crucial to the clustering algorithm.

Distance between two points can be calculated in various different ways, including the most common - Euclidean, Cosine, City block and Correlation. These commonly used distance functions for quantitative data feature are summarized in Table 1 below provided by Madkour et al [1].

Table 1. Dissimilarity (distance) functions for quantitative data feature [1]

Name	Formula	Explanation
Minkowski distance	$\left(\sum_{l=1}^d x_{il} - x_{jl} ^n \right)^{1/n}$	A set of definitions for distance:
		1. City-block distance when n = 1
		2. Euclidean distance when n = 2
Standardized Euclidean distance	$\left(\sum_{l=1}^d \left \frac{x_{il} - x_{jl}}{s_l} \right ^2 \right)^{1/2}$	1. S stands for the standard deviation
		2. A weighted Euclidean distance based on the deviation
Cosine distance	$1 - \cos \alpha = \frac{x_i^T x_j}{\ x_i\ \ x_j\ }$	1. Stay the same in face of the rotation change of data
		2. The most commonly used distance in document area
Pearson correlation distance	$1 - \frac{Cov(x_i, x_j)}{\sqrt{D(x_i)} \sqrt{D(x_j)}}$	1. Cov stands for the covariance and D stands for the variance
		2. Measure the distance based on linear correlation

2.4.1. City Block (Manhattan)

The city block or Manhattan distance is calculated as the distance in x plus the distance in y, similarly to how we move in a city where we go around buildings instead of going straight through [14].

2.4.2. Euclidean

The Euclidean distance between two points is the length of a line segment between the two points. Euclidean distances are not affected by the addition of outliers to the analysis, although difference in measurement scales such as millimeters to centimeters has great consequence to the cluster analysis [14].

2.4.3. Cosine

The cosine distance between two points is one minus the cosine of the included angle between points (treated as vectors) [14].

2.4.4. Correlation Distance

Distance correlation is a measure of dependence between random vectors [14].

2.4.5. Performance

According to a report by Jyoti Bora et al using two datasets, the city block distance was shown to have better performance in terms of less computation time. Cosine was found to take more computation time in comparison to other distance measures [14]. Silhouette analysis was used to plot the results, which showed that correlation distance measures have a better interpretation of the clustered data [14].

Therefore, each distance has benefits based on the use case required.

2.5. Evaluation

The K-means algorithm, although widely used, has many drawbacks which may limit its capability to effectively cluster data.

The main disadvantage of K-means is its reliance on the user to provide a value for the number of clusters, K. When running the algorithm against low K values, the user can test all or most of the possible iterations of K to find the best result, however, as K increases, the margin of error increases and more advanced versions of K-means such as K-means++ may be required for k-means seeding to find better initial centroid values [15]. Although, by using an analysis technique, the optimum value for K can be identified with good results. Both elbow curve and silhouette analysis techniques can be used to identify the optimum number of clusters K, with the silhouette analysis providing more accurate and reliable results over the elbow curve method [13].

Clusters can be negatively affected by outliers in the data, causing centroids to be thrown off or clusters created specifically for these pieces of outlying data causing a skewed segregation of the data against the training model, and therefore data should be cleaned before use within the algorithm [10]. However, this is not always possible, and sometimes outliers still remain. To handle these, in terms of distance measures, Manhattan (City Block) is able to give more robust results as it is influenced by the closeness of other variables. Euclidean distance will exaggerate a discrepancy and is more likely to be influenced by outliers within the dataset [4]. It could be said, that in a dataset with a looser clustering pattern, Manhattan is likely to produce better results than Euclidean.

3. Task 3

3.1.K-Means Clustering Data

3.1.1. Given Centroids and Data Points

The following tables T3.A and T3.B represent the data points and centroids provided for the task, respectively.

Data Points	
X	Y
5	5
0	5
2	3
2	2
5	1
3	3
1	4
5	3
1	2

T3.A – Data Points

Centroids			
1		2	
X	Y	X	Y
0.41	2.66	3.40	0.58

T3.B – Centroids

3.1.2. K-Means Algorithm

An Excel spreadsheet was used to plot the K Means Clustering algorithm and to visualize the data points.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1																						
2																						
3																						
4																						
5																						
6																						
7																						
8																						
9																						
10																						
11																						
12																						
13																						
14																						
15																						

Task 3- K-Means Clustering

Steps for K-Means Clustering Algorithm

1. Select a number of k clusters - for the given task K was provided as K=2.
2. Randomly select k points that represent the centroid – cluster centres.
3. Repeat:-
 - A. Assign each datapoint to their closest centroid.
 - B. Recalculate the centroid of each cluster. It is the mean of the points within that cluster.
 - C. Stop when the centroids no longer change.
4. Calculate Sum of Squared Errors. SSE is defined as the sum of the squared distance between centroid and each member of the cluster.

Figure 13. Outline of steps required for K-Means Clustering Algorithm

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
15																						
16																						
17																						
18																						
19																						
20																						
21																						
22																						
23																						
24																						
25																						
26																						
27																						
28																						
29																						
30																						
31																						
32																						
33																						
34																						
35																						

#1

1. Start by assigning each data point to a cluster randomly. I chose to do this alternating each data point to cluster 1 / cluster 2.
2. Calculate the squared distance of each of the ten data elements to each centroid using formula SUMXMY2 in column N and O (Returns the sum of squares of differences of corresponding values in two arrays).
3. Assign each data point to the nearest cluster as outlined in column M, 'Min Sq Dist'. The results are in the 'Cluster' column R.
4. Recalculate the centroid of each cluster using the mean of the points within the given cluster. This step will be completed in #2 below.

Data Points		k = 2
X	Y	Cluster
5	5	1
0	5	2
2	3	1
2	2	2
5	1	1
3	3	2
1	4	1
5	3	2
1	2	1

Given Centroids		
Centroid	1	2
X	0.41	3.4
Y	2.66	0.58

SSE	53.4441
------------	---------

Not Converged		
----------------------	--	--

Min Sq Dist	1	2
22.0964	26.5437	22.0964
5.6437	5.6437	31.0964
2.6437	2.6437	7.8164
2.9637	2.9637	3.9764
2.7364	23.8237	2.7364
6.0164	6.8237	6.0164
2.1437	2.1437	17.4564
8.4164	21.1837	8.4164
0.7837	0.7837	7.7764

Cluster
2
1
1
1
2
2
1
2
1

Figure 14. Calculations for first iteration of the algorithm

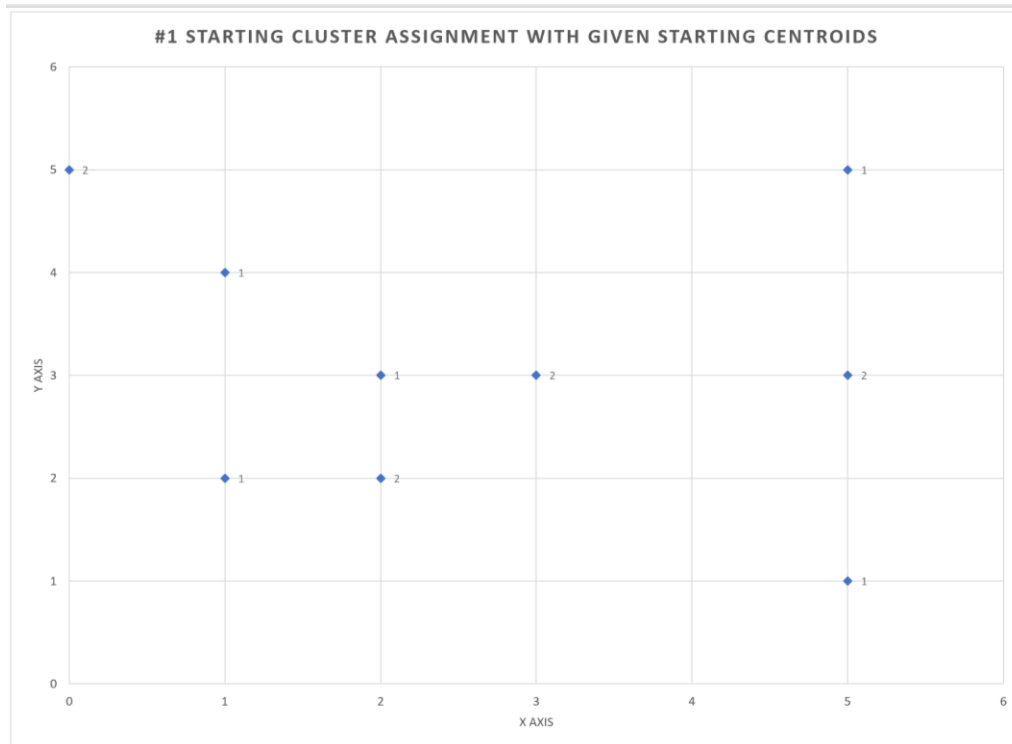


Figure 15. Plot of labelled data points using given starting centroids

</

Figure 16. Calculations including the mean of the points within each cluster to find new centroids

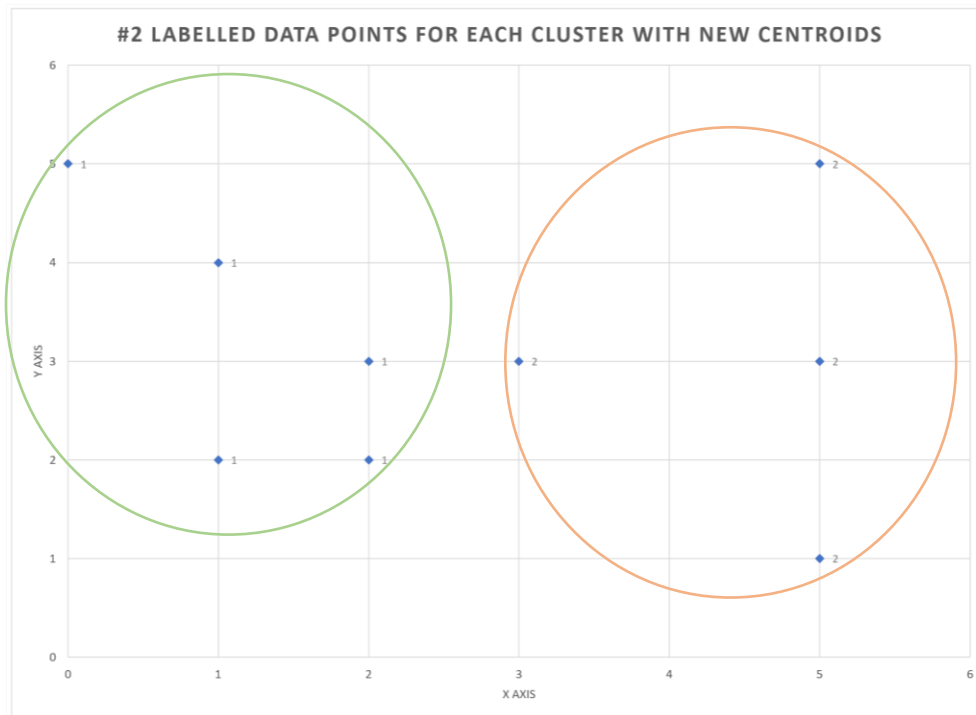


Figure 17. Plot of labelled data points using new centroids and cluster assignment from squared distance

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

#3

1. Using the cluster assignments as chosen above (#2) in column R, set the centroids (column I,J) to be the mean of the points within each cluster (1 or 2).

2. The centroids do not change from the previous step - the algorithm has converged.

Data Points	
X	Y
5	5
0	5
2	3
2	2
5	1
3	3
1	4
5	3
1	2

k=2
Cluster
2
1
1
2
2
1
2
1

Mean Centroids Based on Clusters		
Centroid	1	2
X	1.2	4.5
Y	3.2	3

SSE	20.6
Converged - Centroids Same	

Min Sq Dist	1	2
4.25	17.68	4.25
4.68	4.68	24.25
0.68	0.68	6.25
2.08	2.08	7.25
4.25	19.28	4.25
2.25	3.28	2.25
0.68	0.68	13.25
0.25	14.48	0.25
1.48	1.48	13.25

Cluster
2
1
1
1
2
2
1
2
1

Figure 18. Calculations showing the centroids remain the same - algorithm has converged after 1 iteration

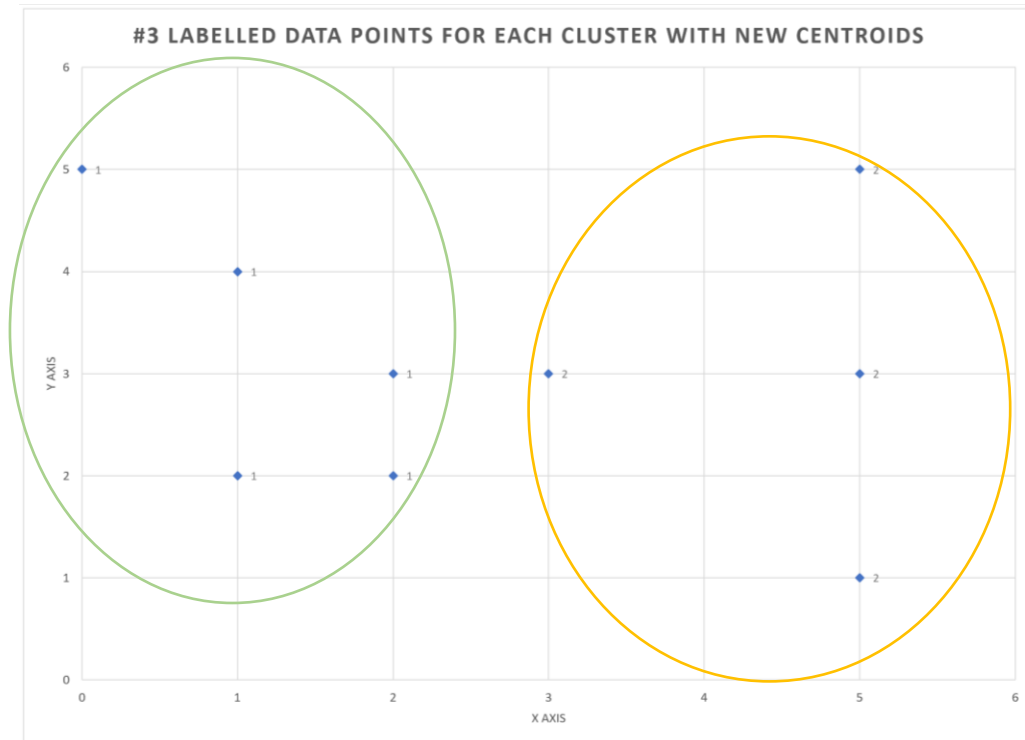


Figure 19. Plot of final cluster assignment

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

#4

1. For experiments sake, test the SSE for when K=3 (3 Clusters)

Follow steps as listed above.

Data Points		k=3 Cluster	Mean Centroids Based on Clusters				Min Sq Dist	1	2	3	Cluster
X	Y		Centroid	1	2	3					
5	5	1	X	2.666667	3.333333	2	6.777777778	7.222222	6.777778	14.444444	2
0	5	2	Y	3.666667	3	2.66667	8.888888889	8.88889	15.11111	9.4444444	1
2	3	3					0.111111111	0.888889	1.777778	0.1111111	3
2	2	1	SSE	30.11111111			0.444444444	3.222222	2.777778	0.4444444	3
5	1	2					6.777777778	12.55556	6.777778	11.777778	2
3	3	3					0.111111111	0.555556	0.111111	1.1111111	2
1	4	1					2.777777778	2.888889	6.444444	2.777778	3
5	3	2					2.777777778	5.888889	2.777778	9.1111111	2
1	2	3	Not Converged				1.444444444	5.555556	6.444444	1.4444444	3

Figure 20. Testing for when K=3 to check for improvement in SSE (First)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
92																						
93																						
94																						
95																						
96																						
97																						
98																						
99																						
100																						
101																						
102																						
103																						
104																						
105																						
106																						
107																						
108																						
109																						

#5

1. Use cluster assignments listed above to calculate new centroids.

Data Points		k=3	Mean Centroids Based on Clusters				Min Sq Dist				Cluster
X	Y	Cluster	Centroid	1	2	3		1	2	3	
5	5	2	X	0	4.5	1.5	6.77777778	7.222222	6.777778	14.444444	2
0	5	1	Y	5	3	2.75	8.88888889	8.888889	15.11111	9.4444444	1
2	3	3					0.11111111	0.888889	1.777778	0.1111111	3
2	2	3	SSE	30.11111111							
5	1	2					0.44444444	3.222222	2.777778	0.4444444	3
3	3	2					6.77777778	12.55556	6.777778	11.777778	2
1	4	3					0.11111111	0.555556	0.111111	1.1111111	2
5	3	2					2.77777778	2.888889	6.444444	2.7777778	3
1	2	3					2.77777778	5.888889	2.777778	9.1111111	2
							1.44444444	5.555556	6.444444	1.4444444	3

Not Converged

Figure 21. Testing for when K=3 to check for improvement in SSE (Second)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
109																						
110																						
111																						
112																						
113																						
114																						
115																						
116																						
117																						
118																						
119																						
120																						
121																						
122																						
123																						
124																						
125																						
126																						
127																						
128																						

#6

1. The centroids do not change from the previous step - the algorithm has converged.

Data Points		k=3	Mean Centroids Based on Clusters				Min Sq Dist				Cluster
X	Y	Cluster	Centroid	1	2	3		1	2	3	
5	5	2	X	0	4.5	1.5	6.77777778	7.222222	6.777778	14.444444	2
0	5	1	Y	5	3	2.75	8.88888889	8.888889	15.11111	9.4444444	1
2	3	3					0.11111111	0.888889	1.777778	0.1111111	3
2	2	3	SSE	30.11111111							
5	1	2					0.44444444	3.222222	2.777778	0.4444444	3
3	3	2					6.77777778	12.55556	6.777778	11.777778	2
1	4	3					0.11111111	0.555556	0.111111	1.1111111	2
5	3	2					2.77777778	2.888889	6.444444	2.7777778	3
1	2	3					2.77777778	5.888889	2.777778	9.1111111	2
							1.44444444	5.555556	6.444444	1.4444444	3

Converged

You can see the SSE is actually higher than when K=2.

Figure 22. Testing for when K=3 to check for improvement in SSE (Final, converged)

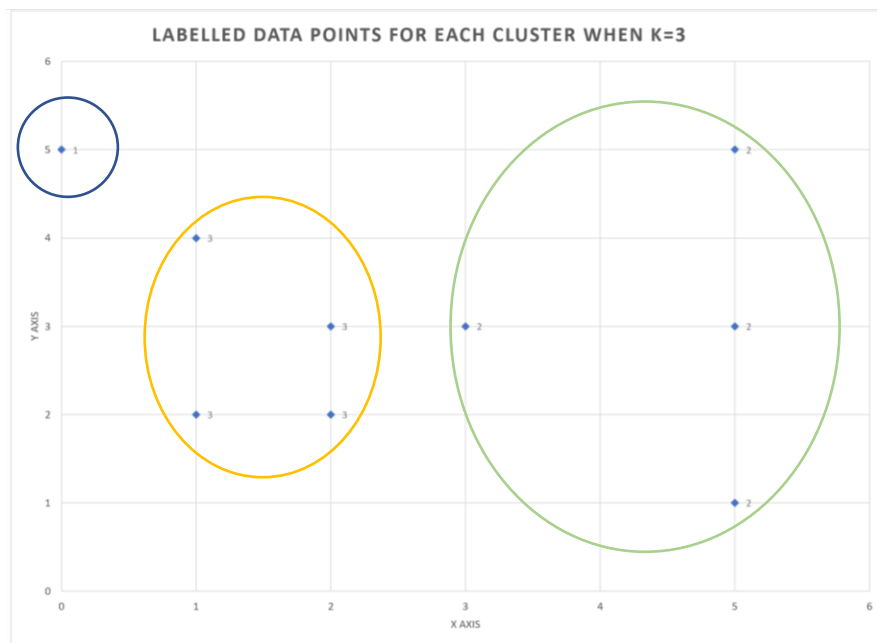


Figure 23. Cluster Assignment for $K=3$

4. References

- [1] A. Madkour, W. G. Aref, F. U. Rehman, M. A. Rahman, and S. Basalamah, “A Survey of Shortest-Path Algorithms,” May 2017, Accessed: Dec. 19, 2021. [Online]. Available: <https://arxiv.org/abs/1705.02044v1>
- [2] M. A. Javaid, “Understanding Dijkstra’s Algorithm,” *SSRN Electronic Journal*, Apr. 2013, doi: 10.2139/SSRN.2340905.
- [3] E. W. Dijkstra, “Dijkstra’s Algorithm,” *Washington Edu*, 2006.
- [4] S. Kapil and M. Chawla, “Performance evaluation of K-means clustering algorithm with various distance metrics,” *1st IEEE International Conference on Power Electronics, Intelligent Control and Energy Systems, ICPEICES 2016*, Feb. 2017, doi: 10.1109/ICPEICES.2016.7853264.
- [5] P. Arora, Deepali, and S. Varshney, “Analysis of K-Means and K-Medoids Algorithm For Big Data,” *Procedia Computer Science*, vol. 78, pp. 507–512, Jan. 2016, doi: 10.1016/J.PROCS.2016.02.095.
- [6] D. T. Pham, S. S. Dimov, and C. D. Nguyen, “Selection of K in K-means clustering:,” <http://dx.doi.org/10.1243/095440605X8298>, vol. 219, no. 1, pp. 103–119, Aug. 2016, doi: 10.1243/095440605X8298.
- [7] O. J. Oyelade, O. O. Oladipupo, and I. C. Obagbuwa, “Application of k-Means Clustering algorithm for prediction of Students’ Academic Performance,” *IJCSIS) International Journal of Computer Science and Information Security*, vol. 7, no. 1, 2010, Accessed: Dec. 19, 2021. [Online]. Available: <http://sites.google.com/site/ijcsis/>
- [8] Han J, Kamber M, and Pei J, *Data Mining Concepts and Techniques*, Third. Morgan Kaufmann, 2012. Accessed: Dec. 19, 2021. [Online]. Available: <http://myweb.sabanciuniv.edu/rdehkharghani/files/2016/02/The-Morgan-Kaufmann-Series-in-Data-Management-Systems-Jiawei-Han-Micheline-Kamber-Jian-Pei-Data-Mining.-Concepts-and-Techniques-3rd-Edition-Morgan-Kaufmann-2011.pdf>
- [9] R. C. Steorts, “How to choose the number of clusters K ?”.
- [10] P. J. Rousseeuw, “Silhouettes: A graphical aid to the interpretation and validation of cluster analysis,” *Journal of Computational and Applied Mathematics*, vol. 20, no. C, pp. 53–65, Nov. 1987, doi: 10.1016/0377-0427(87)90125-7.
- [11] “Selecting the number of clusters with silhouette analysis on KMeans clustering — scikit-learn 1.0.2 documentation.” https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html (accessed Dec. 19, 2021).
- [12] “Selecting the number of clusters with silhouette analysis on KMeans clustering — scikit-learn 1.0.2 documentation.” https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html (accessed Dec. 19, 2021).

- [13] P. Makwana, T. M. Kodinariya, and P. R. Makwana, "Review on Determining of Cluster in K-means Clustering Review on determining number of Cluster in K-Means Clustering," *International Journal of Advance Research in Computer Science and Management Studies*, vol. 1, no. 6, 2013, Accessed: Dec. 19, 2021. [Online]. Available: <https://www.researchgate.net/publication/313554124>
- [14] D. Jyoti Bora and A. Kumar Gupta, "Effect of Different Distance Measures on the Performance of K-Means Algorithm: An Experimental Study in Matlab", Accessed: Dec. 19, 2021. [Online]. Available: www.ijcsit.com
- [15] M. E. Celebi, H. A. Kingravi, and P. A. Vela, "A comparative study of efficient initialization methods for the k-means clustering algorithm," *Expert Systems with Applications*, vol. 40, no. 1, pp. 200–210, Jan. 2013, doi: 10.1016/J.ESWA.2012.07.021.

5. Appendix A – User Guide

5.1.Introduction

The following program allows for a map of size between 16 x 16 and 9999 x 9999 to be searched using Depth First Search and Dijkstra's shortest path algorithm, along with an open map walk.

5.2.Definitions

A *wall* is defined as an asterisk '*'

A *corridor* is defined as an empty space

A *sign* is defined as an uppercase character A-Z

The *start position* is defined by an 'S'

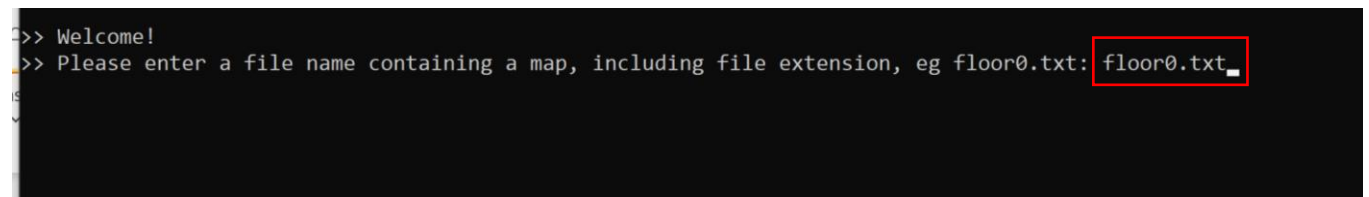
5.3.Load a File

The program will prompt you to enter a file name, please ensure that the file is included within the file directory and that you match the file name exactly including file extensions.

Three files have been provided for test purposes, namely:

- floor0.txt
- floor1.txt
- floor2.txt

If you load an invalid file, eg incorrect size, the program will prompt you to enter the name of a valid map file.



```
>>> Welcome!
>>> Please enter a file name containing a map, including file extension, eg floor0.txt: floor0.txt_
```

Figure 24. Where to enter file name - example floor0.txt entered

If a file has been loaded successfully, the map will be printed, and an alert will inform you the load was successful. You will then be taken to the main menu.

```
*****
S      A  *
*****
*C*
* *****
* *
* *****
*      B*   *D*
* *****
* *
* *****
* *****
* *****
*E      F*
*****

>> File Loaded! Map Size is 16 Wide by 16 High.
```

Figure 25. Successful file load notification

5.4. Use The Menu

Once a valid file has been loaded, the main menu will appear like as below.

```
----- Menu -----

>> Please select from the following options:
>> 1: Print Map
>> 2: Open Map Walk
>> 3: Depth First Search
>> 4: Dijkstra's Shortest Path Algorithm
>> 5: Load A New Map File
>> 6: Exit

>> Please enter a choice (1 to 5): _
```

Figure 26. Main menu options

Please enter for one of the following options:

Enter '1' to Print Map

Enter '2' to go to the Open Map Walk

Enter '3' to run a Depth First Search

Enter '4' to run Dijkstra's Shortest Path Algorithm

Enter '5' to Load a New Map File

Enter '6' to Exit the program

5.5. Print Map (Option 1)

The map will be printed, and then you will be taken back to the menu to select another option.

```
>> Please enter a choice (1 to 5): 1

>> Option Selected: Print Map

*****
S      A      *
*****
*C*      *
* ***** *
* *      *
* *      *
* ***** *
*      B*      *D*
* ***** *
* *      * *
* ***** *
* ***** *
* ***** *
* ***** *
*E      F*
*****
```

Figure 27. Print Map option

5.6.Open Map Walk (Option 2)

To move around the map press:

- A to move left
- D to move right
- W to move up
- S to move down

You will not be able to move into a wall.

If you walk over a sign, it will be turned into a lowercase letter, and the list of signs will be updated – see screenshot.

```
>> A letter on the map represents a sign except 'S' which is the start. If a letter is lowercase, it has been visited.
>> An asterisk '*' represents a wall, and a space ' ' represents a corridor.
>> You are represented by a 'Q' on the map.
>> Press 'a' to move left, 'd' to move right, 'w' to move up, and 's' to move down.
>> Press the 'z' key at any point to quit the map walk.

>> Found Signs at the following Positions:
>> 'A' at X:10,Y:1

*****
      a Q *
*****
*C*      *
* *****
* *      *
* *****
*  B*    *D*
* *****
* *      *
* *****
* *****
* *****
*E      F*
*****
```

Figure 28. Open Map Walk

Enter the z key 'z' at any time to exit the map walk and return to the menu.

5.7.Depth First Search (Option 3)

Option 3 will automatically run a depth first search. Watch as the agent 'Q' searches through the map, leaving a breadcrumb '.' behind.

If found, a list of signs and their co-ordinates will be printed.

[illegible]

Figure 29. DFS

Once completed, you will be returned to the main menu.

5.8.Dijkstra's Shortest Path (Option 4)

Option 4 will automatically run Dijkstra's shortest path algorithm. The adjacency matrix for the graph will be printed to view to the edge weights between each vertex.

If signs are included, the shortest path to each sign from the starting vertex ('S' position) and cost for the path will be printed, as below.


```

Distance of node1=32
Path=1<-2<-7<-8<-11<-12<-19<-18<-27<-29
Distance of node2=25
Path=2<-7<-8<-11<-12<-19<-18<-27<-29
Distance of node3=33
Path=3<-2<-7<-8<-11<-12<-19<-18<-27<-29
Distance of node4=32
Path=4<-9<-10<-11<-12<-19<-18<-27<-29
Distance of node5=15
Path=5<-12<-19<-18<-27<-29
Distance of node6=29
Path=6<-7<-8<-11<-12<-19<-18<-27<-29
Distance of node7=22
Path=7<-8<-11<-12<-19<-18<-27<-29
Distance of node8=16
Path=8<-11<-12<-19<-18<-27<-29
Distance of node9=28
Path=9<-10<-11<-12<-19<-18<-27<-29
Distance of node10=26
Path=10<-11<-12<-19<-18<-27<-29
Distance of node11=13
Path=11<-12<-19<-18<-27<-29
Distance of node12=11
Path=12<-19<-18<-27<-29
Distance of node13=10
Path=13<-14<-17<-18<-27<-29
Distance of node14=8
Path=14<-17<-18<-27<-29
Distance of node15=21
Path=15<-16<-22<-23<-13<-14<-17<-18<-27<-29
Distance of node16=18
Path=16<-22<-23<-13<-14<-17<-18<-27<-29
Distance of node17=6
Path=17<-18<-27<-29
Distance of node18=4
Path=18<-27<-29
Distance of node19=7
Path=19<-18<-27<-29
Distance of node20=31
Path=20<-24<-25<-15<-16<-22<-23<-13<-14<-17<-18<-27<-29
Distance of node21=31
Path=21<-10<-11<-12<-19<-18<-27<-29
Distance of node22=16
Path=22<-23<-13<-14<-17<-18<-27<-29
Distance of node23=13
Path=23<-13<-14<-17<-18<-27<-29
Distance of node24=29
Path=24<-25<-15<-16<-22<-23<-13<-14<-17<-18<-27<-29
Distance of node25=25
Path=25<-15<-16<-22<-23<-13<-14<-17<-18<-27<-29

```

Figure 31. Dijkstra's Shortest Path Option Pt2

Once completed, you will be returned to the main menu.

5.9. Load a New File (Option 5)

Enter file name for new file to be loaded as shown in previous section.

```
----- Menu -----
>> Please select from the following options:
>> 1: Print Map
>> 2: Open Map Walk
>> 3: Depth First Search
>> 4: Dijkstra's Shortest Path Algorithm
>> 5: Load A New Map File
>> 6: Exit

>> Please enter a choice (1 to 5): 5

>> Option Selected: Load a New Map File

>> Please enter a file name containing a map, including file extension, eg floor0.txt: _
```

Figure 32. Load new file option

5.10. Exit (Option 6)

A goodbye prompt will be shown, and the program will end.

```
----- Menu -----
>> Please select from the following options:
>> 1: Print Map
>> 2: Open Map Walk
>> 3: Depth First Search
>> 4: Dijkstra's Shortest Path Algorithm
>> 5: Load A New Map File
>> 6: Exit

>> Please enter a choice (1 to 5): 6

>> Program Exiting! Goodbye.
```

Figure 33. Exit program

6. Appendix B – Test Schedule

A - Menu

#	Description	Input	Test Type	Expected Result	Actual Result	Result
1	Test invalid menu option	a	Invalid	Asks user for input again	Asks user for input again	Pass
2	Test invalid menu option	@	Invalid	Asks user for input again	Asks user for input again	Pass
3	Test out of range menu option	0	Invalid Boundary	Asks user for input again	Asks user for input again	Pass
4	Test out of range menu option	7	Invalid Boundary	Asks user for input again	Asks user for input again	Pass
5	Test lower boundary menu option	1	Valid Boundary	Menu option accepted (Print Map)	Menu option accepted (Print Map)	Pass
6	Test upper boundary menu option	6	Valid Boundary	Menu option accepted (Quit)	Menu option accepted (Quit)	Pass
7	Test valid mid menu option	3	Valid	Menu option accepted (DFS)	Menu option accepted (DFS)	Pass

Figure 34. Test Class A-Menu

B - File Load

#	Description	Input	Test Type	Expected Result	Actual Result	Result
1	Test map width too small	map width = 11	Invalid	Ask for new file name	Ask for new file name	Pass
2	Test map height too small	map height = 11	Invalid	Ask for new file name	Ask for new file name	Pass
3	Test map width too big	map width = 10000	Invalid Boundary	Ask for new file name	Ask for new file name	Pass
4	Test map height too big	map width = 10000	Invalid Boundary	Ask for new file name	Ask for new file name	Pass
5	Test map valid lower boundary width	map width = 16	Valid Boundary	Map loaded	Map loaded	Pass
6	Test map valid lower boundary height	map height = 16	Valid Boundary	Map loaded	Map loaded	Pass
7	Test map valid upper boundary width	map width = 9999	Valid Boundary	Map loaded	Map loaded	Pass
8	Test map valid upper boundary height	map height = 9999	Valid Boundary	Map loaded	Map loaded	Pass
9	Test map valid width and map width different to map height	map width = 21	Valid	Map loaded	Map loaded	Pass

Figure 35. Test Class B - File Loading

C - Map Walk

#	Description	Input	Test Type	Expected Result	Actual Result	Result
1	Test move left where left is a valid move (x,y = ' ')	a	Valid	Q moves left 1 position (new x = old x - 1)	Q moves left 1 position (new x = old x - 1)	Pass
2	Test move right where right is a valid move (x,y = ' ')	d	Valid	Q moves right 1 position (new x = old x + 1)	Q moves right 1 position (new x = old x + 1)	Pass
3	Test move up where up is a valid move (x,y = ' ')	w	Valid	Q moves up 1 position (new y = old y - 1)	Q moves up 1 position (new y = old y - 1)	Pass
4	Test move down where down is a valid move (x,y = ' ')	s	Valid	Q remains in the same position	Q moves down 1 position (new y = old y + 1)	Pass
5	Test move left where left is an invalid move (x,y = '*' wall)	a	Invalid	Q remains in the same position	Q remains in the same position	Pass
6	Test move right where right is an invalid move (x,y = '*' wall)	d	Invalid	Q remains in the same position	Q remains in the same position	Pass
7	Test move up where up is an invalid move (x,y = '*' wall)	w	Invalid	Q remains in the same position	Q remains in the same position	Pass
8	Test move down where down is an invalid move (x,y = '*' wall)	s	Invalid	Q remains in the same position	Q remains in the same position	Pass
9	Test sign is added to sign list when walked over by the agent	using floor0.txt. walk to 8,1 (A)	Valid	>> Found Signs at the following Positions:>> 'A' at X:10,Y:1	>> Found Signs at the following Positions:>> 'A' at X:10,Y:1	Pass
10	Test sign remains visible but lowercase once it has been visited by the agent	using floor0.txt. walk to 8,1 (A)	Valid	A > a	A > a	Pass
11	Test that user can exit the free walk	z	Valid	Returns to main menu	Returns to main menu	Pass

Figure 36. Test Class C - Map Walk

D - Depth First Search

#	Description	Input	Test Type	Expected Result	Actual Result	Result
1	Ensure DFS begins from the correct start position (x,y = 'S') (top left)	floor0.txt 'S' = 0,1	Valid	Begins at 0,1	Begins at 0,1	Pass
2	Ensure DFS begins from the correct start position (x,y = 'S') (bottom right)	floor1.txt 'S' = 15,15	Valid	Begins at 15,15	Begins at 15,15	Pass
3	Ensure every valid space (' ' or char) is visited by the algorithm	floor0.txt	Valid	All valid spaces are marked with a breadcrumb '.'	All valid spaces are marked with a breadcrumb '.'	Pass
4	Ensure every valid space (' ' or char) is visited by the algorithm	floor1.txt	Valid	All valid spaces are marked with a breadcrumb '.'	All valid spaces are marked with a breadcrumb '.'	Pass
5	Ensure that all available signs are discovered by the algorithm	floor0.txt	Valid	A,B,C,D,E,F	A,B,C,D,E,F	Pass
6	Ensure that all available signs are discovered by the algorithm	floor1.txt	Valid	A,B,C,D,E,F	A,B,C,D,E,F	Pass
7	Ensure if no signs are found the user is notified	floor3.txt	Valid	>>> No signs found!	>>> No signs found!	Pass

Figure 37. Test Class D – DFS

E - Dijkstra

#	Description	Input	Test Type	Expected Result	Actual Result	Result
1	Ensure adjacency matrix is accurately created	floor0.txt	Valid	See matrix in document labelled E.1, size 27 x 27	As per E.1	Pass
2	Ensure paths to each sign are correctly identified following the minimum cost	floor0.txt	Valid	Paths are correctly identified	Paths are correctly identified	Pass
3	Ensure adjacency matrix is accurately created	floor1.txt	Valid	Matrix produced correctly with accurate edge costs	Matrix produced correctly with accurate edge costs	Pass
4	Ensure paths to each sign are correctly identified following the minimum cost	floor1.txt	Valid	Paths are correctly identified	Paths are correctly identified	Pass

Figure 38. Test Class E - Dijkstra

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
0	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	8	0	2	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	2	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	2	0	0	0	5	0	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	6	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	4	0	7	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	2	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	4	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	7	0	0
12	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	2	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	3	0	0	0	2	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	2	0	2	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	2	0	3	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	3	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	4	0	3	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	4	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	9	0	4	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0

Figure 39. E.1

7. Appendix C – Screenshots of Working Application

```
>> Welcome!
>> Please enter a file name containing a map, including file extension, eg floor0.txt: floor0.txt

*****
S      A  *
*****
*C*      *
* ***** *
* *      *
* **** *****
*   B*   *D*
* ***** *
* *      * *
* ***** *
* ***** *
* ***** *
* ***** *
*E          F*
*****

>> File Loaded! Map Size is 16 Wide by 16 High.

----- Menu -----

>> Please select from the following options:
>> 1: Print Map
>> 2: Open Map Walk
>> 3: Depth First Search
>> 4: Dijkstra's Shortest Path Algorithm
>> 5: Load A New Map File
>> 6: Exit

>> Please enter a choice (1 to 5): 1
```

Figure 40. Start Menu with floor0.txt file loaded

```

*****
S              A      *
*****
*C*              *
* *****
* * * * *
* * * * *
*          B*      *D*
* *****
* * * * *
* * * * *
* *****
*E              F*
*****

```

```
>> Please enter a choice (1 to 5): _
```

[illegible]

```
>> Found Signs at the following Positions:
>> 'A' at X:10,Y:1
```

```

*****
*****      a Q      *
*****
* C *
*****
* *      *
* *****
*      *
* *****
*      B*      *D*
*****
* *      *
* *****
* *****
*****
* E      F *
*****

```

```
>> A letter on the map represents a sign except 'S' which is the start. If a letter is lowercase, it has been visited.
>> An asterisk '*' represents a wall, and a space ' ' represents a corridor.
>> You are represented by a 'Q' on the map.
>> Press 'a' to move left, 'd' to move right, 'w' to move up, and 's' to move down.
>> Press the 'z' key at any point to quit the map walk.
```

```
>> Found Signs at the following Positions:
>> 'A' at X:10,Y:1
```

```

*****
*****      a Q      *
*****
* C*                *
*****
*      *            *
*****
*      B*          * D*
*****
*      *            *
*****
*      *            *
*****
*      *            *
*****
*      *            *
*****
*      *            *
*****
* E                F*
*****

```

----- Menu -----

```
>> Please select from the following options:
>> 1: Print Map
>> 2: Open Map Walk
>> 3: Depth First Search
>> 4: Dijkstra's Shortest Path Algorithm
>> 5: Load A New Map File
>> 6: Exit
```

```
>> Please enter a choice (1 to 5): _
```

Page | 41


```

Path1<-0
Distance of node2=10
Path2<-1<-0
Distance of node3=14
Path3<-2<-1<-0
Distance of node4=61
Path4<-11<-24<-25<-22<-21<-18<-19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node5=15
Path5<-6<-1<-0
Distance of node6=10
Path6<-1<-0
Distance of node7=16
Path7<-6<-1<-0
Distance of node8=29
Path8<-9<-10<-7<-6<-1<-0
Distance of node9=25
Path9<-10<-7<-6<-1<-0
Distance of node10=18
Path10<-7<-6<-1<-0
Distance of node11=57
Path11<-24<-25<-22<-21<-18<-19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node12=61
Path12<-11<-24<-25<-22<-21<-18<-19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node13=27
Path13<-9<-10<-7<-6<-1<-0
Distance of node14=29
Path14<-13<-9<-10<-7<-6<-1<-0
Distance of node15=32
Path15<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node16=46
Path16<-23<-22<-21<-18<-19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node17=37
Path17<-18<-19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node18=33
Path18<-19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node19=31
Path19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node20=34
Path20<-19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node21=35
Path21<-18<-19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node22=38
Path22<-21<-18<-19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node23=42
Path23<-22<-21<-18<-19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node24=50
Path24<-25<-22<-21<-18<-19<-14<-13<-9<-10<-7<-6<-1<-0
Distance of node25=41
Path25<-22<-21<-18<-19<-14<-13<-9<-10<-7<-6<-1<-0

```

Figure 48. Option 4: Dijkstra's shortest path Continued


```
----- Menu -----  
  
>> Please select from the following options:  
>> 1: Print Map  
>> 2: Open Map Walk  
>> 3: Depth First Search  
>> 4: Dijkstra's Shortest Path Algorithm  
>> 5: Load A New Map File  
>> 6: Exit  
  
>> Please enter a choice (1 to 5): 6  
  
>> Program Exiting! Goodbye.
```

Figure 50. Option 6: exit

8. Appendix D – Program Listing

Continued on the following pages.

File List:

1. DepthFirstSearch.h
2. DepthFirstSearch.cpp
3. Dijkstra.h
4. Dijkstra.cpp
5. Print.h
6. Print.cpp
7. SearchUtility.h
8. SearchUtility.cpp
9. Main.cpp
10. floor0.txt
11. floor1.txt
12. floor2.txt

8.1.DepthFirstSearch.h

```
#pragma once

#include "SearchUtility.h"
#include "Print.h"
#include<iostream>
#include<stdio.h>
#include<vector>
#include <stack>
#include <fstream>
#include <string>
#include <utility>

using namespace std;

class DepthFirstSearch
{
public:
    void DFS(vector<vector<char>>);
};
```

8.2.DepthFirstSearch.cpp

```
#include "DepthFirstSearch.h"

using namespace std;

void DepthFirstSearch::DFS(vector<vector<char>> map) {

    SearchUtility su;
    Print pr;
    stack<SearchUtility::position> dfsStack;
    vector<pair<char, SearchUtility::position>> signs;
    SearchUtility::position current = { 1,1 }; //Initialise values
    SearchUtility::position adjPositions = { 1,1 };
    int i = 0;

    current = su.findStartPosition(map);

    cout << ">> Start position is: (" << current.x << " , " << current.y << ")" <<
endl;

    //Run dfs adding postions onto the stack
    dfsStack.push(current);
    int j = 0; //Used to count number of signs

    while (!dfsStack.empty())
    {
        system("cls"); //Clear console to view map progress
        cout << "\n" << endl;

        pr.print2DVector(map); //Print map to show progress to user

        map[current.y][current.x] = '.'; //Mark visited nodes so that they are not
visited more than once

        current = dfsStack.top(); //Set current to the top of the stack
        dfsStack.pop(); //Pop these positions off the stack as they will be searched
now

        //Retain an array of signs - Assume a sign is anything but a full-stop '.'
asterisk '*' or space ' '
        if (map[current.y][current.x] != ' ' && map[current.y][current.x] != '*' &&
map[current.y][current.x] != '.' && map[current.y][current.x] != 'Q') {
            SearchUtility::position signPosition = { current.x,current.y };
            char sign = map[current.y][current.x];
            signs.push_back(make_pair(sign, signPosition));
        }

        map[current.y][current.x] = 'Q'; //Mark agent's current position

        if (current.x > 0) { //Check to avoid out of bounds error
            //If LEFT position is not an obstacle then stack.push(LEFT position)
            if ((map[current.y][current.x - 1] != '*') &&
(map[current.y][current.x - 1] != '.'))
            {
```

```

        adjPositions.x = current.x - 1;
        adjPositions.y = current.y;
        dfsStack.push(adjPositions);
    }
}
if (current.x < map[0].size() - 1) { //Check to avoid out of bounds error
    //If RIGHT position is not an obstacle then stack.push(RIGHT
position)
    if ((map[current.y][current.x + 1] != '*') &&
(map[current.y][current.x + 1] != '.'))
    {
        adjPositions.x = current.x + 1;
        adjPositions.y = current.y;
        dfsStack.push(adjPositions);
    }
}

if (current.y > 0) { //Check to avoid out of bounds error
    //If UP position is not an obstacle then stack.push(UP position)
    if ((map[current.y - 1][current.x] != '*') && (map[current.y -
1][current.x] != '.'))
    {
        adjPositions.x = current.x;
        adjPositions.y = current.y - 1;
        dfsStack.push(adjPositions);
    }
}

if (current.y < map.size() - 1) { //Check to avoid out of bounds error
    //If DOWN position is not an obstacle then stack.push(DOWN position)
    if ((map[current.y + 1][current.x] != '*') && (map[current.y +
1][current.x] != '.'))
    {
        adjPositions.x = current.x;
        adjPositions.y = current.y + 1;
        dfsStack.push(adjPositions);
    }
}
}

if (signs.size() < 1) { //If no signs were found, inform the user
    cout << "\n>> No Signs were found for the current map!\n" << endl;
}
else { //If signs are found
    //Print the list of signs found during the Depth-First search by iterating
through the signs array
    cout << "\n>> Found Signs at the following Positions:\n" << endl;
    for (i = 0; i < signs.size(); i++) {
        cout << ">> " << signs[i].first << " at X:" << signs[i].second.x
<< ",Y:" << signs[i].second.y << endl;
    }
    cout << "\n" << endl;
}
}
}

```

8.3.Dijkstra.h

```
#pragma once
#include<iostream>
#include<stdio.h>
#include<vector>
#include <stack>
#include <fstream>
#include <string>
#include <utility>

using namespace std;

#define INFINITY 9999

class Dijkstra
{
public:
    void dijkstra(vector<vector<int>>, int, vector<pair<char, int>>);
};
```

8.4.Dijkstra.cpp

```
#include "Dijkstra.h"

using namespace std;

/*
 * Function to run Dijkstra's shortest path algorithm on an adjacency matrix
 * Takes the adjacency matrix G, start node ID startnode, and vector pair signsVertices as
 * parameters
 * The code used below originated from https://www.tutorialspoint.com/cplusplus-program-for-dijkstra-s-shortest-path-algorithm with limited modifications.
 */
/* Please see accompanying documentation for psuedocode on shortest path algorithm.
 */
void Dijkstra::dijkstra(vector<vector<int>> G, int startnode, vector<pair<char, int>>
signsVertices) {
    int n = G[0].size();
    int max = G[0].size();

    vector<vector<int>> > cost(max, vector<int>(max));
    vector<int> distance(max);
    vector<int> pred(max);
    vector<int> visited(max);
    int count, mindistance, nextnode, i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (G[i][j] == 0)
                cost[i][j] = INFINITY;
            else
                cost[i][j] = G[i][j];
    for (i = 0; i < n; i++) {
        distance[i] = cost[startnode][i];
        pred[i] = startnode;
        visited[i] = 0;
    }
    distance[startnode] = 0;
    visited[startnode] = 1;
    count = 1;
    while (count < n - 1) {
        mindistance = INFINITY;
        for (i = 0; i < n; i++)
            if (distance[i] < mindistance && !visited[i]) {
                mindistance = distance[i];
                nextnode = i;
            }
        visited[nextnode] = 1;
        for (i = 0; i < n; i++)
            if (!visited[i])
                if (mindistance + cost[nextnode][i] < distance[i]) {
                    distance[i] = mindistance + cost[nextnode][i];
                    pred[i] = nextnode;
                }
        count++;
    }

    cout << "\n" << endl;
    cout << "List of signs and their node IDs!\n" << endl;
```

```

    for (i = 0; i < signsVertices.size(); i++) {
        cout << "Sign '" << signsVertices[i].first << "' is Node '" <<
signsVertices[i].second << "'!" << endl;
    }

    for (i = 0; i < n; i++) {
        if (i != startnode) {

            cout << "\nDistance of node" << i << "=" << distance[i];
            cout << "\nPath=" << i;
            j = i;
            do {
                j = pred[j];
                cout << "<-" << j;
            } while (j != startnode);
        }
    }
}

```

8.5.Print.h

```
#pragma once
#include<iostream>
#include<stdio.h>
#include<vector>
#include <stack>
#include <fstream>
#include <string>
#include <utility>

using namespace std;

class Print
{
public:
    void print2DVector(vector<vector<char>>);
    void print2DVector(vector<vector<int>>);
};
```

8.6.Print.cpp

```
#include "Print.h"

using namespace std;

void Print::print2DVector(vector<vector<char>> vec) {
    cout << "\n" << endl;
    for (int i = 0; i < vec.size(); i++)
    {
        for (int j = 0; j < vec[i].size(); j++)
        {
            cout << vec[i][j];
        }
        cout << endl;
    }
}

void Print::print2DVector(vector<vector<int>> vec) {
    cout << "\n" << endl;
    for (int i = 0; i < vec.size(); i++)
    {
        for (int j = 0; j < vec[i].size(); j++)
        {
            cout << vec[i][j] << ",";
        }
        cout << endl;
    }
}
```

8.7.SearchUtility.h

```
#pragma once
#include<iostream>
#include<stdio.h>
#include<vector>
#include <stack>
#include <fstream>
#include <string>
#include <utility>

using namespace std;

#define INFINITY 9999

class SearchUtility
{
public:

    typedef struct {
        int x;
        int y;
    } position;

    vector<vector<char>>> loadFileToVector2D();
    void mapWalk(vector<vector<char>>>);
    int findStartNode(vector<vector<char>>>, vector<position>);
    position findStartPosition(vector<vector<char>>>);
    vector<position> findVertices(vector<vector<char>>>);
    bool vertexExists(vector<position>, position);
    vector<vector<int>>> buildAdjacencyMatrix(vector<vector<char>>>, vector<position>);
    vector<pair<char, int>>> signsNodeIndex(vector<vector<char>>>, vector<position>);

};
```

8.8.SearchUtility.cpp

```
#include "SearchUtility.h"
```

```
#include "Print.h"
```

```
using namespace std;
```

```
/*
```

```
* loadFileToVector2D
```

```
* Returns a vector of vectors (similar to 2d array) containing the map
```

```
*/
```

```
vector<vector<char>> SearchUtility::loadFileToVector2D() {
```

```
    Print p;
```

```
    string line;//Used to store each line read from the map file
```

```
    int count = 0;//Loop variable used to increment index of vector
```

```
    string fileName = "";//Variable to store the file name
```

```
    //Reset Values incase of previously loaded file
```

```
    int height = 0;
```

```
    int width = 0;
```

```
    try {
```

```
        //While file does not meet size validation rules, ask for new file
```

```
        while (height < 16 || width < 16 || height > 9999 || width > 9999) {
```

```
            fileName = "";//Clear filename in case of previous loop iteration
```

```
            cout << ">> Please enter a file name containing a map, including file extension,
```

```
            eg floor0.txt: ";
```

```

cin >> fileName;//Store user input in variable
cout << "\n" << endl;

ifstream myfile(fileName);//Open file

if (myfile.is_open()) {

    //First, loop through the file to gather the height and width of the map.
    //Set the global variables height and width to the result.
    //Height is calculated by incrementing a loop variable for each line of the
file.

    //Width is calculated by gathering the length of the line, using
string.length() method.

    while (getline(myfile, line)) {
        width = line.length();
        height++;
    }

    //Reset the position of ifstream so that we read from the beginning of the
file.

    myfile.clear();
    myfile.seekg(0, ios::beg);

    //Declare new vector map to hold the contents of the map file. Initialised
to the height*width gathered above.

    //This allows for files (maps) of varying sizes to be used without
limitation.

    vector<vector<char>>> map(height);
    vector<char> mapRow(width);

    count = 0;

```

```

while (getline(myfile, line))
{
    //cout << line << endl; //Print for testing purposes
    line.copy(&mapRow[0], width, 0); //Copy each line of the file as
a string to the vector

    map[count] = mapRow;
    count++; //Increase loop variable used for vector index above
}

p.print2DVector(map); //Print the loaded file for the user

myfile.close(); //Close file

//Print specific error messages to the user based on the condition that has
not been met

if (width < 16 || height < 16) {
    cout << "\n>> Map must be greater than 16 in height and 16 in
width. Please try another file.\n" << endl;
}
else if (width > 9999 || height > 9999) {
    cout << "\n>> Map must not be greater than 9999 in height or
9999 in width. Please try another file.\n" << endl;
}
else {
    cout << "\n>> File Loaded! Map Size is " << width << " Wide
by " << height << " High.\n" << endl;

    return map; //Return vector map containing map contents, to be
used within calling function
}
}

```

```

        }
        return {};//Return empty - will trigger some exception handling inside the calling
function
    }
    catch (...) { //Catch exceptions
        cout << "\n>> An error has occurred, please try again." << endl;
        return {};
    }
}

/*
* mapWalk
* Method that allows user to freely walk around the map using the aswd keys to move left/right/up/down
respectively.
*/

void SearchUtility::mapWalk(vector<vector <char>> map) {
    Print pr;
    char key = 'q';
    int i = 0;
    SearchUtility::position current = { 1,1 };
    SearchUtility::position adjPositions = { 1,1 };
    vector<pair<char, SearchUtility::position> > signs;

    current = findStartPosition(map);

    while (key != 'z')
    {
        system("cls");

        cout << "\n>> A letter on the map represents a sign except 'S' which is the start. If a letter
is lowercase, it has been visited." << endl;

```

```

        cout << ">> An asterisk '*' represents a wall, and a space ' ' represents a corridor." <<
endl;

        cout << ">> You are represented by a 'Q' on the map." << endl;

        cout << ">> Press 'a' to move left, 'd' to move right, 'w' to move up, and 's' to move
down." << endl;

        cout << ">> Press the 'z' key at any point to quit the map walk.\n" << endl;

//Print the list of signs found during the Depth-First search by iterating through the signs
array
        cout << "\n>> Found Signs at the following Positions:\n" << endl;
        for (i = 0; i < signs.size(); i++) {
                cout << ">> " << signs[i].first << " at X:" << signs[i].second.x << ",Y:" <<
signs[i].second.y << endl;
        }
        cout << "\n" << endl;

        pr.print2DVector(map);
        cin >> key;

        switch (key) {
        case 'd':
                if (current.x < map[0].size() - 1) {
                        if (map[current.y][current.x + 1] != '*') {
                                if (!islower(map[current.y][current.x])) map[current.y][current.x]
= ' '; //overwrite my avatar with a space as I am leaving this position, but do not overwrite sign
                                current.x = current.x + 1; // move
                                if (islower(map[current.y][current.x])) {
                                        break;
                                }
                                else if (map[current.y][current.x] != ' ' &&
map[current.y][current.x] != '*' && map[current.y][current.x] != '.' && map[current.y][current.x] != 'Q'
&& !islower(map[current.y][current.x])) {

```

```

current.x,current.y };

SearchUtility::position signPosition = {

char sign = map[current.y][current.x];
signs.push_back(make_pair(sign, signPosition));
map[current.y][current.x] = tolower(sign);

}

else {

map[current.y][current.x] = 'Q';

}

}

}

break;

case 'a':

if (current.x > 0) {

if (map[current.y][current.x - 1] != '*') {

if (!islower(map[current.y][current.x])) map[current.y][current.x]
= ' '; //overwrite my avatar with a space as I am leaving this position, but do not overwrite sign

current.x = current.x - 1; // move

if (islower(map[current.y][current.x])) {

break;

}

else if (map[current.y][current.x] != ' ' &&
map[current.y][current.x] != '*' && map[current.y][current.x] != '.' && map[current.y][current.x] != 'Q'
&& !islower(map[current.y][current.x])) {

SearchUtility::position signPosition = {

current.x,current.y };

char sign = map[current.y][current.x];

signs.push_back(make_pair(sign, signPosition));

map[current.y][current.x] = tolower(sign);

}

else {

```

```

        map[current.y][current.x] = 'Q';
    }
}
}
break;
case 'w':
    if (current.y > 0) {
        if (map[current.y - 1][current.x] != '*') {
            if (!islower(map[current.y][current.x])) map[current.y][current.x]
= ' '; //overwrite my avatar with a space as I am leaving this position, but do not overwrite sign
            current.y = current.y - 1; // move
            if (islower(map[current.y][current.x])) {
                break;
            }
            else if (map[current.y][current.x] != ' ' &&
map[current.y][current.x] != '*' && map[current.y][current.x] != '.' && map[current.y][current.x] != 'Q'
&& !islower(map[current.y][current.x])) {
                SearchUtility::position signPosition = {
current.x, current.y };

                char sign = map[current.y][current.x];
                signs.push_back(make_pair(sign, signPosition));
                map[current.y][current.x] = tolower(sign);
            }
            else {
                map[current.y][current.x] = 'Q';
            }
        }
    }
}
break;
case 's':
    if (current.y < map.size() - 1) {

```


* a corner, dead end, intersection or sign, and if it is, adds the x,y co-ordinates to a vector vertices

*

* It takes the map vector of vectors (2d arr) as input.

*/

```
vector<SearchUtility::position> SearchUtility::findVertices(vector<vector<char>> vec) {
    cout << "\n" << endl;
    vector<SearchUtility::position> vertices(INFINITY);
    char upperL, upperC, upperR, centreL, centre, centreR, lowerL, lowerC, lowerR;
    char wall = '*';
    char corridor = ' ';
    char current;
    int i = 0;
    position v;
    int width = vec[0].size();
    int height = vec.size();

    for (int y = 0; y < vec.size(); y++)
    {
        //Loop through rows (y)
        for (int x = 0; x < vec[y].size(); x++)
        {
            //Loop through columns (x)
            //Usually values i and j would be used for nested for loops, here i = y, j = x

            centre = vec[y][x];

            if (y > 0) {
                upperC = vec[y - 1][x];
                if (x > 0) {
                    upperL = vec[y - 1][x - 1];
```

```

    }
    if (x < (width - 1)) {
        upperR = vec[y - 1][x + 1];
    }
}

if (y < (height - 1)) {
    lowerC = vec[y + 1][x];
    if (x > 0) {
        lowerL = vec[y + 1][x - 1];
    }
    if (x < (width - 1)) {
        lowerR = vec[y + 1][x + 1];
    }
}

if (x > 0) {
    centreL = vec[y][x - 1];
}

if (x < (width - 1)) {
    centreR = vec[y][x + 1];
}

//Test Vertice Rules

//Start Node

if (centre == 'S')
{
    //Add current position to list of vertices
    v = { x,y };//Create variable to store x y co-ordinates
    vertices[i] = v;//Add position to list of vertices
    i++;//Increment loop var

```

```

//cout << "\nFound Vertex!! Position is x:" << x << ", y:" << y << "\n"
<< endl;//Testing Statement

        continue;

    }

    if (x > 0 && x < (width - 1) && y > 0 && y < (height - 1)) {
        if (
            // #1
            (upperL == wall && upperC == wall && upperR == wall &&
            centreL != wall && centre != wall && centreR != wall && lowerL == wall && lowerC != wall &&
            lowerR == wall) ||

            // #2
            (upperL == wall && upperC != wall && upperR == wall &&
            centreL != wall && centre != wall && centreR != wall && lowerL == wall && lowerC == wall &&
            lowerR == wall) ||

            // #3
            (upperL == wall && upperC != wall && upperR == wall &&
            centreL != wall && centre != wall && centreR == wall && lowerL == wall && lowerC != wall &&
            lowerR == wall) ||

            // #4
            (upperL == wall && upperC != wall && upperR == wall &&
            centreL == wall && centre != wall && centreR != wall && lowerL == wall && lowerC != wall &&
            lowerR == wall)

        ) {
            //Add current position to list of vertices
            v = { x,y }; //Create variable to store x y co-ordinates
            vertices[i] = v; //Add position to list of vertices
            i++; //Increment loop var
            continue;
        }
    }

    if (x < (width - 1) && y > 0 && y < (height - 1)) {

```

```

        // #5
        if (upperC == wall && upperR == wall && centre != wall && centreR
== wall && lowerC == wall && lowerR == wall)
        {
            //Add current position to list of vertices
            v = { x,y }; //Create variable to store x y co-ordinates
            vertices[i] = v; //Add position to list of vertices
            i++; //Increment loop var
            continue;
        }
    }

    if (x > 0 && y > 0 && y < (height - 1)) {
        // #6
        if (upperL == wall && upperC == wall && centreL == wall && centre
!= wall && lowerL == wall && lowerC == wall)
        {
            //Add current position to list of vertices
            v = { x,y }; //Create variable to store x y co-ordinates
            vertices[i] = v; //Add position to list of vertices
            i++; //Increment loop var
            continue;
        }
    }

    if (x > 0 && x < (width - 1) && y < (height - 1)) {
        // #7
        if (centreL == wall && centre != wall && centreR == wall && lowerL
== wall && lowerC == wall && lowerR == wall)
        {
            //Add current position to list of vertices
            v = { x,y }; //Create variable to store x y co-ordinates

```

```

        vertices[i] = v;//Add position to list of vertices
        i++;//Increment loop var
        continue;
    }
}
if (x > 0 && x < (width - 1) && y > 0) {
    //8
    if (upperL == wall && upperC == wall && upperR == wall && centreL
== wall && centre != wall && centreR == wall)
    {
        //Add current position to list of vertices
        v = { x,y };//Create variable to store x y co-ordinates
        vertices[i] = v;//Add position to list of vertices
        i++;//Increment loop var
        continue;
    }
}

if (x < (width - 1) && y > 0) {
    //9
    if (upperC == wall && upperR == wall && centre != wall && centreR
== wall)
    {
        //Add current position to list of vertices
        v = { x,y };//Create variable to store x y co-ordinates
        vertices[i] = v;//Add position to list of vertices
        i++;//Increment loop var
        continue;
    }
}

```

```

if (x > 0 && y > 0) {
    // #10
    if (upperL == wall && upperC == wall && centreL == wall && centre
!= wall)
    {
        // Add current position to list of vertices
        v = { x,y }; // Create variable to store x y co-ordinates
        vertices[i] = v; // Add position to list of vertices
        i++; // Increment loop var
        continue;
    }
}

if (x < (width - 1) && y < (height - 1)) {
    // #11
    if (centre != wall && centreR == wall && lowerC == wall && lowerR
== wall)
    {
        // Add current position to list of vertices
        v = { x,y }; // Create variable to store x y co-ordinates
        vertices[i] = v; // Add position to list of vertices
        i++; // Increment loop var
        continue;
    }
}

if (x > 0 && y < (height - 1)) {
    // #12
    if (centreL == wall && centre != wall && lowerL == wall && lowerC
== wall)
    {

```

```

        //Add current position to list of vertices
        v = { x,y };//Create variable to store x y co-ordinates
        vertices[i] = v;//Add position to list of vertices
        i++;//Increment loop var
        continue;
    }
}

//#13
if (isalpha(centre))
{
    //Add current position to list of vertices
    v = { x,y };//Create variable to store x y co-ordinates
    vertices[i] = v;//Add position to list of vertices
    i++;//Increment loop var
    continue;
}

}

}

vertices.resize(i);//Resize vertices vector to loop variable responsible for populating vector

return vertices;
}

```

/*

* vertexExists is a function which takes two input parameters,

* vertices (vector containing x,y co-ordinates of vertices) and current (current x, y position)

```

* and returns true if the vertice exists, and false if it does not
*/

bool SearchUtility::vertexExists(vector<position> vertices, position current) {
    for (int i = 0; i < vertices.size(); i++) { //Loop through the array and run comparison
        if (vertices[i].x == current.x && vertices[i].y == current.y) {
            return true; //If exists, return true
        }
    }
    return false;
}

/*
* findStartNode returns an int representing the index of the start node within the vertices vector if it
exists, if not it defaults to 0
*/

int SearchUtility::findStartNode(vector<vector<char>> map, vector<SearchUtility::position> vertices)
{
    position startNode = { 1,1 };

    for (int y = 0; y < map.size(); y++) //Loop rows
    {
        for (int x = 0; x < map[0].size(); x++) //Loop columns
        {
            if (map[y][x] == 'S') { //if current position is start marker
                //Loop through the vertices array and find the index of the vertex
                matching this position
                for (int vertex = 0; vertex < vertices.size(); vertex++) {
                    if (x == vertices[vertex].x && y == vertices[vertex].y) {
                        return vertex; //return the index of the vertex
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}

return 0;//Default if S is not present
}

/*
* signsNodeIndex
* Purpose of function is to work out which vertices are 'Signs',
* and to return a vector of pairs representing each sign and its corresponding node number
*/
vector<pair<char, int>> SearchUtility::signsNodeIndex(vector<vector<char>> map,
vector<SearchUtility::position> vertices) {

    vector<pair<char, int>> signsVertices;
    char sign;
    int signNode;

    //Loop through the 2D vector representing the map
    for (int y = 0; y < map.size(); y++) {
        for (int x = 0; x < map[0].size(); x++) {
            if (isalpha(map[y][x])) { //Check if the current index is an alpha character (a Sign)
                for (int vertex = 0; vertex < vertices.size(); vertex++) { //Loop through
the vertices array
                    if (x == vertices[vertex].x && y == vertices[vertex].y)
{ //Compare the vertices x,y coordinates with those of the sign
                        sign = map[y][x]; //Make a note of the sign's name
                        signNode = vertex; //Save the current vertex/node 'ID' to
be used in the pair

```



```

/*
* buildAdjacencyMatrix
* Used to build a 2d array of edge costs between vertices
* Takes the vertices vector and map vector as inputs and returns the new adjacency matrix to be used by
searching algorithms.
*/

vector<vector<int>> SearchUtility::buildAdjacencyMatrix(vector<vector<char>> vec, vector<position>
vertices) {

    Print p;

    vector<vector<int>> adjacencyMatrix(vertices.size(), vector<int>(vertices.size())); // Initialise all
indexes of adjacency matrix to 0s

    int col = 0;
    int row = 0;
    int currNode = 0;
    int vertex = 0;
    int edgeWeight = 0;
    bool adjNodeFound = false;
    position currPos;
    int width = vec[0].size();
    int height = vec.size();

    //for each vertex
    for (currNode = 0; currNode < vertices.size(); currNode++) {

        currPos = { vertices[currNode].x, vertices[currNode].y };
        edgeWeight = 0;
        adjNodeFound = false;
        //1. look right

```

```

while (!adjNodeFound && currPos.x < width - 1 && vec[currPos.y][currPos.x] != '*') {
    edgeWeight++;
    currPos.x = currPos.x + 1;

    //Loop through each vertex in the list and see if the current position is within the
list, if so, add it to the adj matrix.
    for (vertex = 0; vertex < vertices.size(); vertex++) {
        if (currPos.x == vertices[vertex].x && currPos.y == vertices[vertex].y) {
            adjacencyMatrix[currNode][vertex] = edgeWeight;
            adjNodeFound = true; //Set to true so that the loop does not
continue to search to the right.
            break;
        }
    }
}

//Reset values for next check

//Could be made more efficient in future by turning this into a method with a parameter
for left/right/up/down required checking
currPos = { vertices[currNode].x, vertices[currNode].y };
edgeWeight = 0;
adjNodeFound = false;

//2. look left
while (!adjNodeFound && currPos.x > 0 && vec[currPos.y][currPos.x] != '*') {
    edgeWeight++;
    currPos.x = currPos.x - 1;
    for (vertex = 0; vertex < vertices.size(); vertex++) {
        if (currPos.x == vertices[vertex].x && currPos.y == vertices[vertex].y) {
            adjacencyMatrix[currNode][vertex] = edgeWeight;
            adjNodeFound = true;
            break;
        }
    }
}

```

```

    }
}

currPos = { vertices[currNode].x, vertices[currNode].y };
edgeWeight = 0;
adjNodeFound = false;
//3. look upwards
while (!adjNodeFound && currPos.y > 0 && vec[currPos.y][currPos.x] != '*') {
    edgeWeight++;
    currPos.y = currPos.y - 1;
    for (vertex = 0; vertex < vertices.size(); vertex++) {
        if (currPos.x == vertices[vertex].x && currPos.y == vertices[vertex].y) {
            adjacencyMatrix[currNode][vertex] = edgeWeight;
            adjNodeFound = true;
            break;
        }
    }
}

```

```

currPos = { vertices[currNode].x, vertices[currNode].y };
edgeWeight = 0;
adjNodeFound = false;
//4. look downwards
while (!adjNodeFound && currPos.y < height - 1 && vec[currPos.y][currPos.x] != '*') {
    edgeWeight++;
    currPos.y = currPos.y + 1;
    for (vertex = 0; vertex < vertices.size(); vertex++) {
        if (currPos.x == vertices[vertex].x && currPos.y == vertices[vertex].y) {
            adjacencyMatrix[currNode][vertex] = edgeWeight;
            adjNodeFound = true;
        }
    }
}

```

```
                break;
            }
        }
    }
}

return adjacencyMatrix;//Return the matrix to the calling function
}
```

8.9.Main.cpp

```
#include "Dijkstra.h"
#include "Print.h"
#include "DepthFirstSearch.h"
#include "SearchUtility.h"

#include<iostream>
#include<stdio.h>
#include<vector>
#include <stack>
#include <fstream>
#include <string>
#include <utility>

using namespace std;

int main() {

    try {

        //Declare required vars
        int x = 0;
        char input;
        int fileLoaded = 0;

        cout << endl;
        cout << ">> Welcome!" << endl;

        Print pr; //Create new instance of Print class
        SearchUtility su; //Create new instance of SearchUtility class
        Dijkstra d; //Create new instance of Dijkstra class
        DepthFirstSearch dfs; //Create new instance of DepthFirstSearch class

        //Firstly, gather input and load a text file to a 2d vector to be used as
the map going forward
        vector<vector<char>> map = su.loadFileToVector2D();

        //Check if the file loaded correctly - the vector should not be empty.
        //If it is, ask for another file.
        while (map.empty()) {
            cout << ">> File did not load, please try again." << endl;
            map = su.loadFileToVector2D();
        }

        fileLoaded = 1; //Use for checking within option statements

        //Use the 2D array size to calculate the height and width of the map
        int height = map.size();
        int width = map[0].size();

        /*
        * Call the findVertices method to create a vector of positions(x, y co -
ordinates) for each vertex.
        * A vertex can be defined within this application as a:
        * - Corner
        */
    }
```

```

*           - Intersection
*           - Dead End
*           - Sign
*       An edge can be defined as a:
*           - Corridor
*/

vector<SearchUtility::position> vertices = su.findVertices(map);

/*
* Once the vertices have been listed, the start node can be identified by
comparing the 2D map vector and vertices to find the 'S' chracter representing the start
position.
*/
int startNode = su.findStartNode(map, vertices);

//cout << "\nStart node is: " << startNode << "'\n" << endl; //Output used
for application testing

/*
* Call build adjacency matrix function to produce an adjacency matrix to be
used by both DFS and shortest path algorithms.
* The adjacency matrix is used as a way to outline the paths between
vertices, and the edge costs for each path [1] T. Mo, H. Zhao and W. Mo, "Design and
Implementation of Shortest Travel Path Searching Based on Improved Dijkstra Algorithm",
Applied Mechanics and Materials, vol. 157-158, pp. 390-394, 2012.
*/
vector<vector<int>>> adjMatrix = su.buildAdjacencyMatrix(map, vertices);

vector<pair<char, int>> signsVertices = su.signsNodeIndex(map, vertices);

// - - - - -
while (x < 6)
{
    x = 0;
    cout << "\n\n----- Menu -----" << endl;

    cout << "\n>> Please select from the following options:" << endl;
    cout << ">> 1: Print Map" << endl;
    cout << ">> 2: Open Map Walk" << endl;
    cout << ">> 3: Depth First Search" << endl;
    cout << ">> 4: Dijkstra's Shortest Path Algorithm" << endl;
    cout << ">> 5: Load A New Map File" << endl;
    cout << ">> 6: Exit" << endl;

    while (x < 1 || x > 6) {
        cout << "\n>> Please enter a choice (1 to 5): ";
        cin >> input;
        cout << "\n" << endl;
        switch (input) {
            case '1':
                x = 1;
                break;
            case '2':
                x = 2;
                break;
            case '3':

```

```

        x = 3;
        break;
    case '4':
        x = 4;
        break;
    case '5':
        x = 5;
        break;
    case '6':
        x = 6;
        break;
    }
}

if (x == 1 && fileLoaded)
{
    cout << ">> Option Selected: Print Map\n" << endl;
    pr.print2DVector(map);
}
else if (x == 2 && fileLoaded)
{
    cout << ">> Option Selected: Random Walk\n" << endl;
    su.mapWalk(map);
}
else if (x == 3 && fileLoaded) {
    cout << ">> Option Selected: Depth First Search\n" << endl;
    dfs.DFS(map);
}
else if (x == 4 && fileLoaded) {
    cout << ">> Option Selected: Shortest Path\n" << endl;
    /*
    * Firstly, print out the adjacency matrix for the user
    */
    cout << ">> Adjacency Matrix for the given map is as follows:"
<< endl;

    pr.print2DVector(adjMatrix);

    /*
    * Call Dijkstra's Shortest Path Algorithm using the adjacency
matrix created above - along with other necessary variables.
    */
    d.dijkstra(adjMatrix, startNode, signsVertices); // Call
Dijkstra Shortest Path Method
}
else if (x == 5) {
    cout << ">> Option Selected: Load a New Map File\n" << endl;

    fileLoaded = 0; //Reset fileLoaded variable
    //Firstly, gather input and load a text file to a 2d vector to
be used as the map going forward
    map = su.loadFileToVector2D();
    //Check if the file loaded correctly - the vector should not
be empty.

    //If it is, ask for another file.
    while (map.empty()) {

```

```

        cout << ">> File did not load, please try again." <<
endl;
        map = su.loadFileToVector2D();
    }
    fileLoaded = 1; //Use for checking within option statements
    //Use the 2D array size to calculate the height and width of
the map
    height = map.size();
    width = map[0].size();
    }
    else if (x == 6) {
        cout << ">> Program Exiting! Goodbye." << endl;
        return 0;
    }
}

return 1;

}
catch (...) {
    cout << "Error! Please try again later." << endl;
}
return 0;
}

```

8.10. floor0.txt

```
*****
S          A      *
*****
*C*          *
* ***** *
* *          *
* *****
*          B*      *D*
* ***** *
* *          * *
* ***** *
* ***** *
* ***** *
* ***** *
*E          F*
*****
```

8.11. floor1.txt

```
*****
*A                      B*
*** *****
F* *****C*
* *                * *
* ***** * *
* ***** * *
*                *
* * ***** *
* * ***** **** *
* * * ** * **** *
* * * ** * *      *
* D* **      *** ** *
* *** ***** ** *
*      **          **E*
*****S*****
```

8.12. floor2.txt

```
*****
*S      A  *
*****
*C*      *
* ***** *
* *      *
* ** *****
* *B      D*
* ***** *
*E      F*
*****
```