Algorithms & Data structures – CMPU2001

Assignment 1 Report

Name: Stephen Thompson

Student Number: C21394693

Course Code: TU858/2

Code Collaborators:

- Oisin Kiely - C2150436
- Cesar Hannin – C21415142
- Liam Tuite – C21447352

Contents:

1. Introduction

Within the Kruskal source file we have four different classes Kruskal, edge, heap, and set

- Within Kruskal we have the main method as well as functionality for reading in the data from our text file to create a graph object from the data.
- The edge class Is used to represent the edges of our graph by storing values for vertexes and the weight of the graph.
- Our heap class like the heap in GraphLists defines a heap data structure used to manipulate the data provided to implement kruskals algorithm efficiently, it takes in the array of edge and uses sift up sift down and remove methods to maintain the integrity of our data structure.
- Our set class is used to represent the disjoint set data structure that kruskals algorithm uses to keep track of the vertices that have been merged.

Within the GraphLists source file we have the classes Graph Heap and Node.

- Within Graph we define a graph using linked lists to store the graphs adjacent vertices of a corresponding vertex. We also use tail nodes to mark the end of the linked list. Within graph we have constructors defined to read in data from the graph file provided and reads the edges from each line of the file and assigns them values for their corresponding vertexes and weight using a linked list to store these values. This class also contains two methods of traversal being breadth first and depth first both of which contain an array to keep track of visited vertexes. The class also keeps track of adjacent vertexes of each visited vertex.
- Within Heap we define our heap data structure which is used in our implementation of Dijkstras algorithm. This class contains corresponding sift up and sift down methods as well as an is empty method to ensure the structure of our heap is not compromised.
- Within Node we define the Nodes to be used within the program by assigning them values such as weight, vertex and next which helps us keep track of our graph within the linked list format.

2. Adjacency Lists diagrams

For the adjacency list I will be using the sample graph to show the codes functionality.

```
13   22
1   2   1
1   6   2
1   7   6
2   3   1
2   4   2
2   5   4
3   5   4
4   5   2
4   6   1
5   6   2
5   7   1
5   12   4
6   12   2
7   8   3
7   10   1
7   12   5
8   9   2
9   11   1
10   11   1
10   12   3
10   13   2
12   13   1
```

This is the graphs format with line one being the number of vertices and edges and the subsequent lines showing the source -> destination (weight).

Meaning our graphs adjacency list should resemble this:

1 -> 2 (1) -> 6 (2) -> 7 (6)

2 -> 1 (1) -> 3 (1) -> 4 (2) -> 5 (4)

3 -> 2 (1) -> 5 (4)

4 -> 2 (2) -> 5 (2) -> 6 (1)

5 -> 2 (4) -> 3 (4) -> 4 (2) -> 6 (2) -> 7 (1) -> 12 (4)

6 -> 1 (2) -> 4 (1) -> 12 (2)

7 -> 1 (6) -> 5 (1) -> 8 (3) -> 10 (1) -> 12 (5)

8 -> 7 (3) -> 9 (2)

9 -> 8 (2) -> 11 (1)

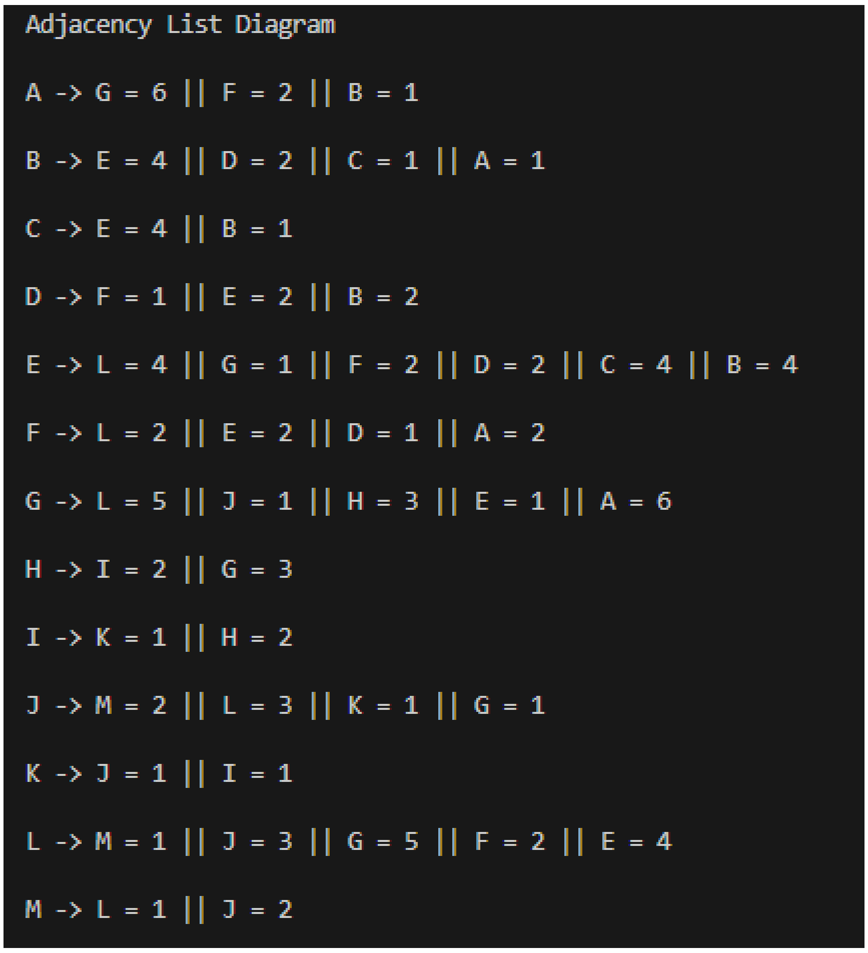
10 -> 7 (1) -> 11 (1) -> 12 (3) -> 13 (2)

11 -> 9 (1) -> 10 (1)

12 -> 5 (4) -> 6 (2) -> 7 (5) -> 10 (3) -> 13 (1)

13 -> 10 (2) -> 12 (1)

Screenshot from executed code:



```
Adjacency List Diagram

A -> G = 6 || F = 2 || B = 1

B -> E = 4 || D = 2 || C = 1 || A = 1

C -> E = 4 || B = 1

D -> F = 1 || E = 2 || B = 2

E -> L = 4 || G = 1 || F = 2 || D = 2 || C = 4 || B = 4

F -> L = 2 || E = 2 || D = 1 || A = 2

G -> L = 5 || J = 1 || H = 3 || E = 1 || A = 6

H -> I = 2 || G = 3

I -> K = 1 || H = 2

J -> M = 2 || L = 3 || K = 1 || G = 1

K -> J = 1 || I = 1

L -> M = 1 || J = 3 || G = 5 || F = 2 || E = 4

M -> L = 1 || J = 2
```

3. Construction of MST using Prim’s

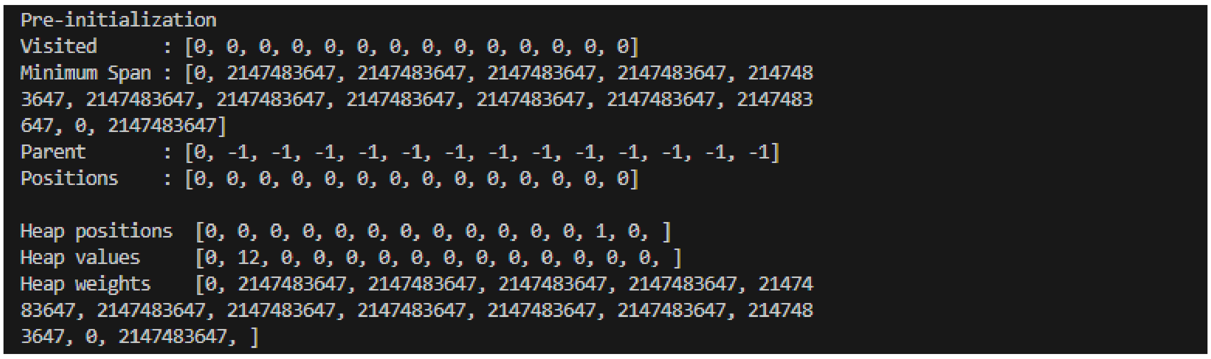
Below are our pre-initialisation values for prims algorithm to create these we create an instance of heap which will store our vertices in order of their MST weights, and we use visited, min_span, parent, and positions to keep track of our vertices that have been visited the minimum weight of the edges in our mst, the parent of every vertex and positional data of our vertices.

Positions is initialised to all 0s to show no values have been added to the heap

Visited is initialised to all 0s to show nothing has been visited

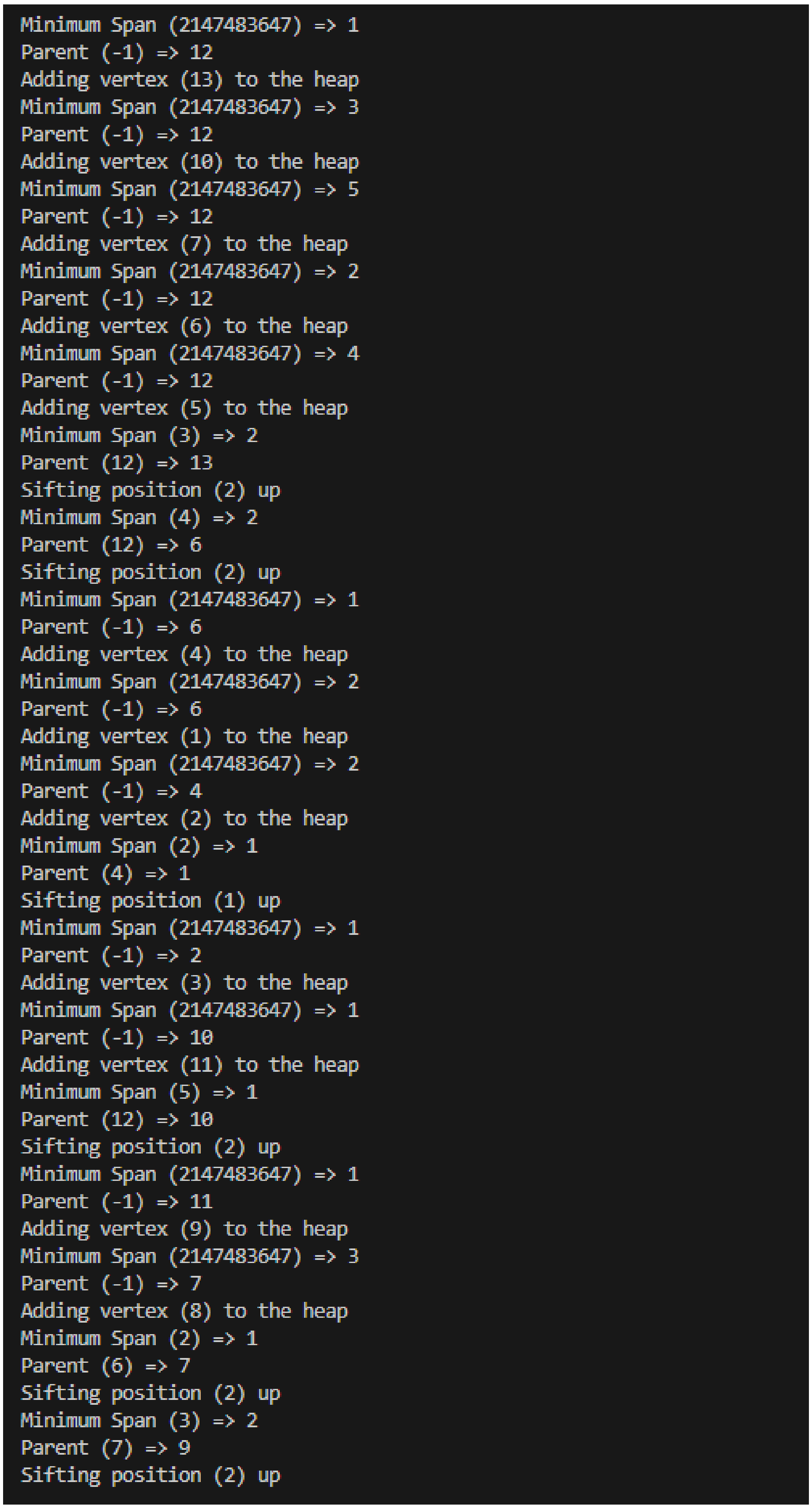
Parent is all -1s as no vertices have been added to the mst

Min_span is set to integer.Max_value for everything except the starting vertex



```
Pre-initialization
Visited      : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Minimum Span : [0, 2147483647, 2147483647, 2147483647, 2147483647, 214748
3647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483
647, 0, 2147483647]
Parent       : [0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
Positions    : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Heap positions  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, ]
Heap values     [0, 12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ]
Heap weights    [0, 2147483647, 2147483647, 2147483647, 2147483647, 21474
83647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 214748
3647, 0, 2147483647, ]
```

Each step of Prims printed in terminal window:



```
Minimum Span (2147483647) => 1
Parent (-1) => 12
Adding vertex (13) to the heap
Minimum Span (2147483647) => 3
Parent (-1) => 12
Adding vertex (10) to the heap
Minimum Span (2147483647) => 5
Parent (-1) => 12
Adding vertex (7) to the heap
Minimum Span (2147483647) => 2
Parent (-1) => 12
Adding vertex (6) to the heap
Minimum Span (2147483647) => 4
Parent (-1) => 12
Adding vertex (5) to the heap
Minimum Span (3) => 2
Parent (12) => 13
Sifting position (2) up
Minimum Span (4) => 2
Parent (12) => 6
Sifting position (2) up
Minimum Span (2147483647) => 1
Parent (-1) => 6
Adding vertex (4) to the heap
Minimum Span (2147483647) => 2
Parent (-1) => 6
Adding vertex (1) to the heap
Minimum Span (2147483647) => 2
Parent (-1) => 4
Adding vertex (2) to the heap
Minimum Span (2) => 1
Parent (4) => 1
Sifting position (1) up
Minimum Span (2147483647) => 1
Parent (-1) => 2
Adding vertex (3) to the heap
Minimum Span (2147483647) => 1
Parent (-1) => 10
Adding vertex (11) to the heap
Minimum Span (5) => 1
Parent (12) => 10
Sifting position (2) up
Minimum Span (2147483647) => 1
Parent (-1) => 11
Adding vertex (9) to the heap
Minimum Span (2147483647) => 3
Parent (-1) => 7
Adding vertex (8) to the heap
Minimum Span (2) => 1
Parent (6) => 7
Sifting position (2) up
Minimum Span (3) => 2
Parent (7) => 9
Sifting position (2) up
```

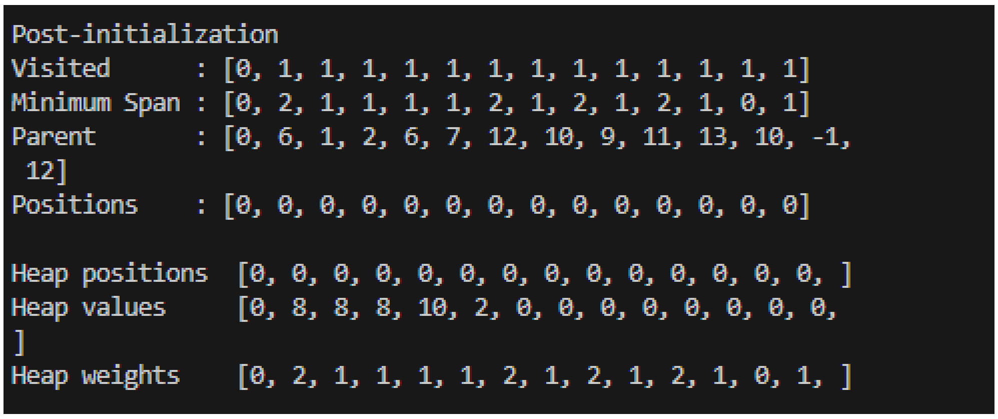
Explaining each step:

This screenshot shows the execution of the while loop for Prims. We start by initialising our data structures and assigning our starting vertex, we then enter the loop and the loop continues until our heap is empty (indicating all our vertices have been visited).

Within our loop Prims chooses the vertex with the smallest weight from our heap designates it visited and then checks every adjacent vertex. With each adjacent vertex that hasn't been visited our algorithm gets the weight of the edge between our current vertex and the next vertex if it decided the weight is less than our current minimum it updates the values for the parent. If our vertex is already in the heap, it sifts it to the right position.

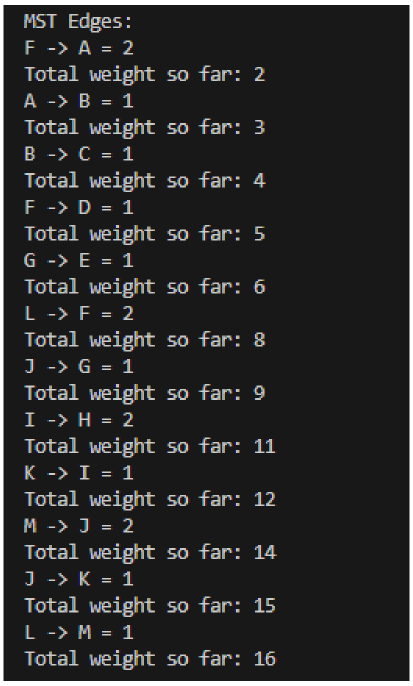
This loop ends when all our vertices are visited.

Below is our post-initialisation which operates the same as our pre-initialisation to show our now processed arrays and data structures within Prims.



```
Post-initialization
Visited      : [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Minimum Span : [0, 2, 1, 1, 1, 1, 2, 1, 2, 1, 2, 1, 0, 1]
Parent       : [0, 6, 1, 2, 6, 7, 12, 10, 9, 11, 13, 10, -1,
 12]
Positions    : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Heap positions  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ]
Heap values     [0, 8, 8, 8, 10, 2, 0, 0, 0, 0, 0, 0, 0, 0,
]
Heap weights    [0, 2, 1, 1, 1, 1, 2, 1, 2, 1, 2, 1, 0, 1, ]
```

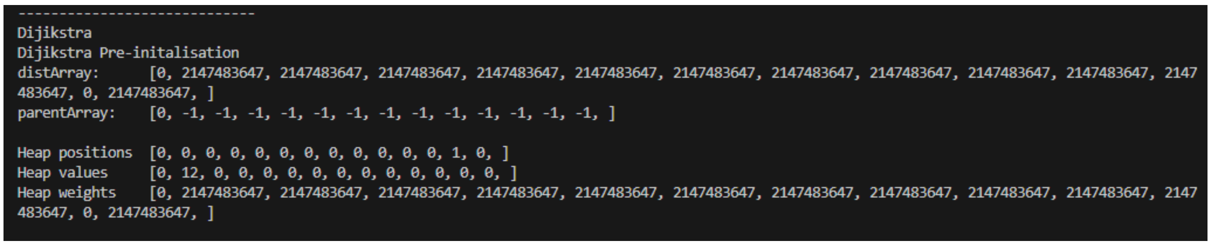
Our MST from L:



```
MST Edges:
F -> A = 2
Total weight so far: 2
A -> B = 1
Total weight so far: 3
B -> C = 1
Total weight so far: 4
F -> D = 1
Total weight so far: 5
G -> E = 1
Total weight so far: 6
L -> F = 2
Total weight so far: 8
J -> G = 1
Total weight so far: 9
I -> H = 2
Total weight so far: 11
K -> I = 1
Total weight so far: 12
M -> J = 2
Total weight so far: 14
J -> K = 1
Total weight so far: 15
L -> M = 1
Total weight so far: 16
```

4. Step by step construction of the SST using Dijkstra's algorithm

Pre-initialisation values + parent and distArray initialisation similar to our Prims pre-initialisation:



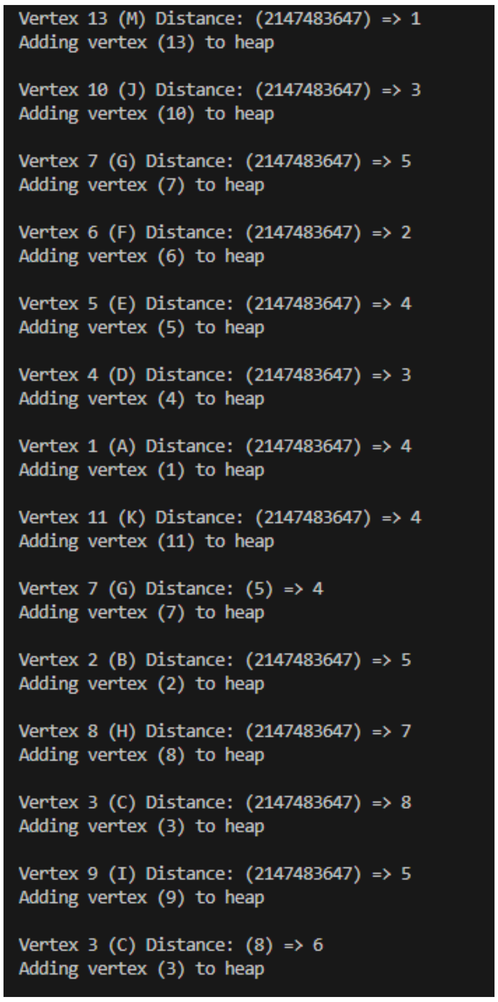
```
-----------------------------
Dijikstra
Dijikstra Pre-initalisation
distArray:      [0, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147
483647, 0, 2147483647, ]
parentArray:    [0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, ]

Heap positions  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, ]
Heap values     [0, 12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ]
Heap weights    [0, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147
483647, 0, 2147483647, ]
```

Traversal of the heap in Dijkstra's algorithm:

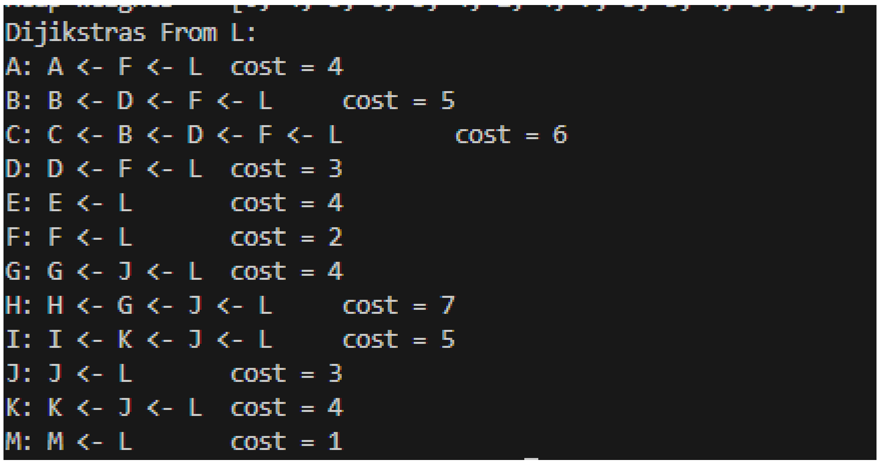
The purpose of Dijkstra's algorithm is to find the shortest path between our starting vertex and all other vertexes in the graph. We use a heap to do this and initialise all our values of distance to be infinite (our max value) except the starting node which is initialised to 0 we then add the starting vertex to the heap and while our heap isn't empty we take our vertexes with the smallest distance from the heap.

We then examine all neighbouring nodes of the extracted node and if our new distance is smaller than our previous distance we update the distance and the parent of the adjacent vertex. If it hasn't been visited we add it to the heap and if it has been visited we siftUp.

This continues until we are left with the shortest path from the starting vertex to all other vertexes.

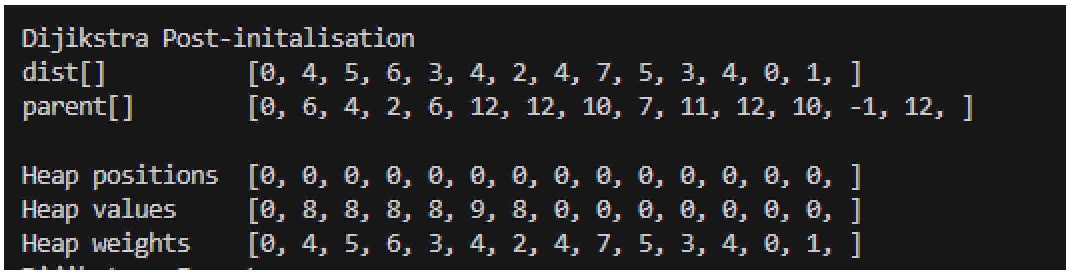


```
Vertex 13 (M) Distance: (2147483647) => 1
Adding vertex (13) to heap

Vertex 10 (J) Distance: (2147483647) => 3
Adding vertex (10) to heap

Vertex 7 (G) Distance: (2147483647) => 5
Adding vertex (7) to heap

Vertex 6 (F) Distance: (2147483647) => 2
Adding vertex (6) to heap

Vertex 5 (E) Distance: (2147483647) => 4
Adding vertex (5) to heap

Vertex 4 (D) Distance: (2147483647) => 3
Adding vertex (4) to heap

Vertex 1 (A) Distance: (2147483647) => 4
Adding vertex (1) to heap

Vertex 11 (K) Distance: (2147483647) => 4
Adding vertex (11) to heap

Vertex 7 (G) Distance: (5) => 4
Adding vertex (7) to heap

Vertex 2 (B) Distance: (2147483647) => 5
Adding vertex (2) to heap

Vertex 8 (H) Distance: (2147483647) => 7
Adding vertex (8) to heap

Vertex 3 (C) Distance: (2147483647) => 8
Adding vertex (3) to heap

Vertex 9 (I) Distance: (2147483647) => 5
Adding vertex (9) to heap

Vertex 3 (C) Distance: (8) => 6
Adding vertex (3) to heap
```



```
Dijikstras From L:
A: A <- F <- L  cost = 4
B: B <- D <- F <- L     cost = 5
C: C <- B <- D <- F <- L        cost = 6
D: D <- F <- L  cost = 3
E: E <- L       cost = 4
F: F <- L       cost = 2
G: G <- J <- L  cost = 4
H: H <- G <- J <- L     cost = 7
I: I <- K <- J <- L     cost = 5
J: J <- L       cost = 3
K: K <- J <- L  cost = 4
M: M <- L       cost = 1
```

Dijkstra's Post-initialisation:



```
Dijikstra Post-initalisation
dist[]          [0, 4, 5, 6, 3, 4, 2, 4, 7, 5, 3, 4, 0, 1, ]
parent[]        [0, 6, 4, 2, 6, 12, 12, 10, 7, 11, 12, 10, -1, 12, ]

Heap positions  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ]
Heap values     [0, 8, 8, 8, 8, 9, 8, 0, 0, 0, 0, 0, 0, 0, ]
Heap weights    [0, 4, 5, 6, 3, 4, 2, 4, 7, 5, 3, 4, 0, 1, ]
```

5. Kruskal's MST

Within our Kruskal's class we operate much the same as we did for Dijkstra's and Prims this is done by printing the graph in its entirety as well as an adjacency list.

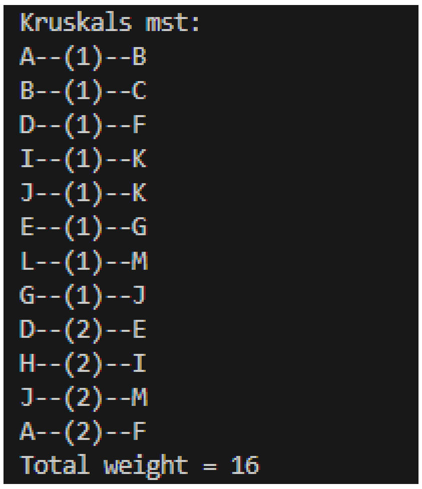
But the implementation of Kruskal's is where things differ.

Within the Kruskal's method we create an array of edges using get_edges then create a heap using Heap class.

Set will then create our disjoint set of all vertices. We traverse through all our edges smallest to largest gradually removing edges from the heap, getting of the sets with their endpoints and then using the get method of Set to check if they're the same. If they're not they get added to the MST and if they are the same, we ignore the edge.

Traversal only stops when all vertices are in the MST then the total weight is calculated, and our edges of the MST are printed.

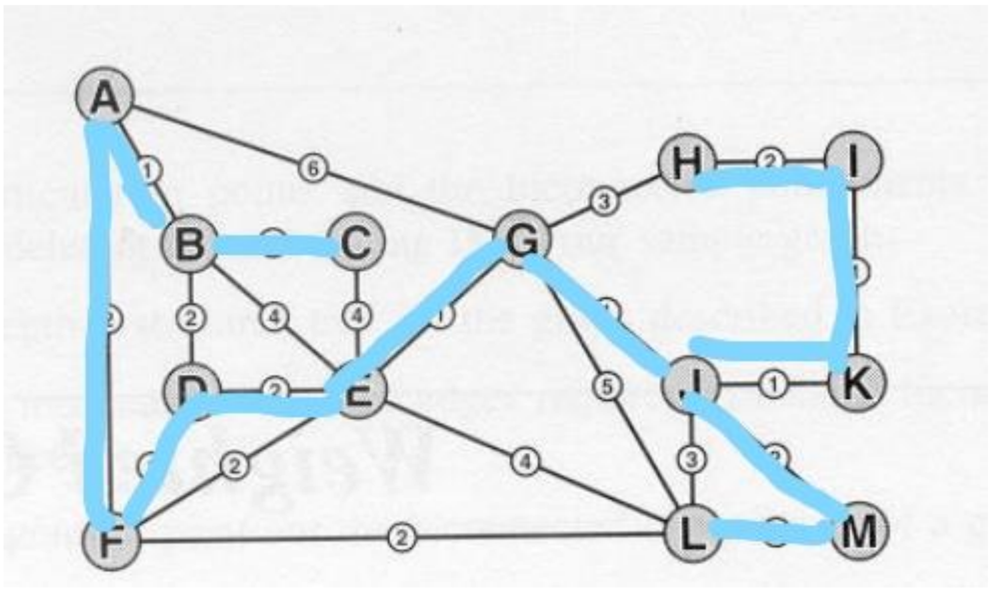
Printed MST with total weight:



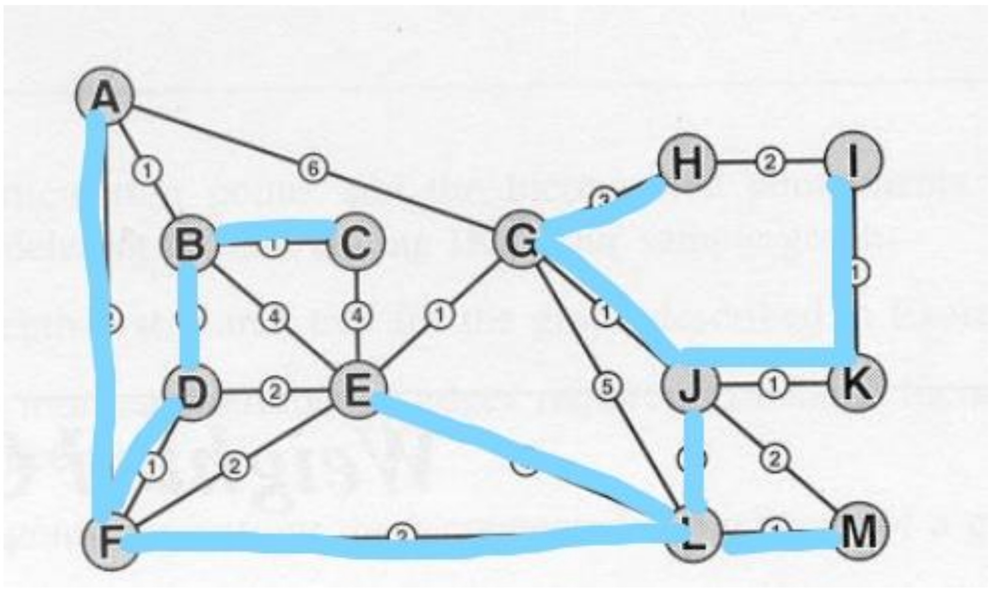
```
Kruskals mst:
A--(1)--B
B--(1)--C
D--(1)--F
I--(1)--K
J--(1)--K
E--(1)--G
L--(1)--M
G--(1)--J
D--(2)--E
H--(2)--I
J--(2)--M
A--(2)--F
Total weight = 16
```

<u>MST and SST on graph graphic</u>:

MST:

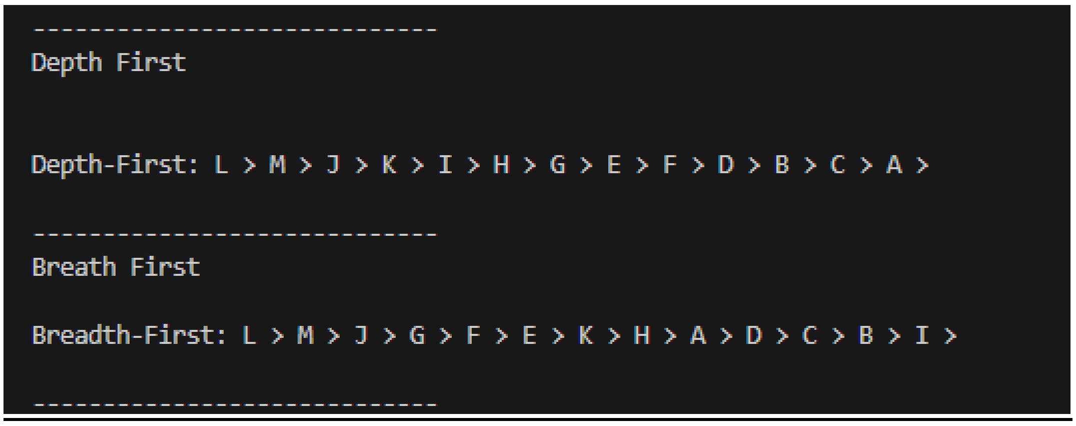


SPT From L:

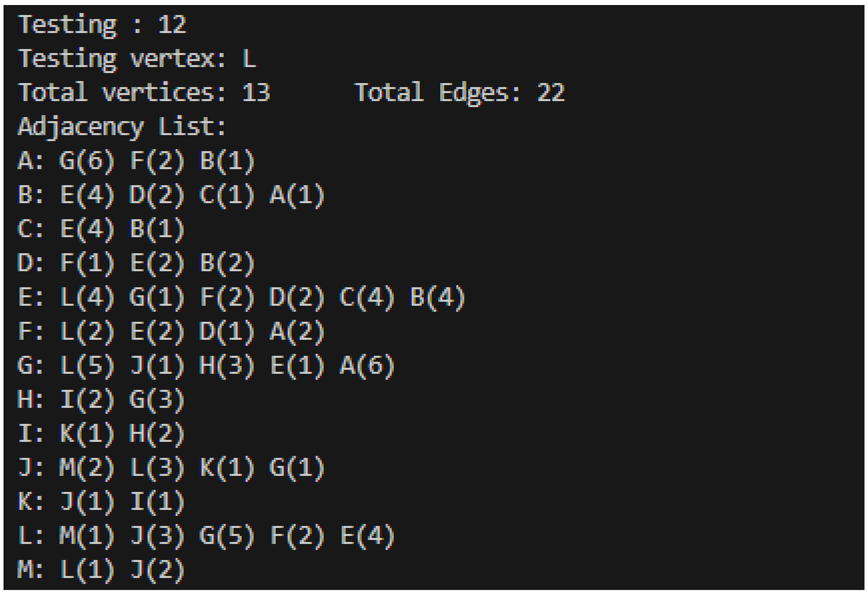


8.(Screenshots of execution shown throughout the report)

- Depth and breadth first search


```
-----------------------------
Depth First


Depth-First: L > M > J > K > I > H > G > E > F > D > B > C > A >

-----------------------------
Breath First

Breadth-First: L > M > J > G > F > E > K > H > A > D > C > B > I >

-----------------------------
```

- Adjacency List printed


```
Testing : 12
Testing vertex: L
Total vertices: 13      Total Edges: 22
Adjacency List:
A: G(6) F(2) B(1)
B: E(4) D(2) C(1) A(1)
C: E(4) B(1)
D: F(1) E(2) B(2)
E: L(4) G(1) F(2) D(2) C(4) B(4)
F: L(2) E(2) D(1) A(2)
G: L(5) J(1) H(3) E(1) A(6)
H: I(2) G(3)
I: K(1) H(2)
J: M(2) L(3) K(1) G(1)
K: J(1) I(1)
L: M(1) J(3) G(5) F(2) E(4)
M: L(1) J(2)
```

9.Discussion of assignment:

I think this assignment has really helped me with my understanding of this module. I was lost going in but having collaborated with my classmates and with help from them along the way as well as some self-directed learning I think I have a far better understanding of these algorithms and they're not as daunting as I once assumed. I think my report reflects what I have learned as I would not have been able to produce accurate descriptions and analysis of the code prior.

Altogether I think this was a fantastic learning experience that I think will be of great benefit to me come exam time.