Graph Traversal, MST and SPT Algorithm Assignment

Student Name: Ian Miller
Student No.: D23124620
Lecturer: Mr. Richard Lawlor
Module: CMPU2001 - Algorithms & Data Structures

1. Introduction

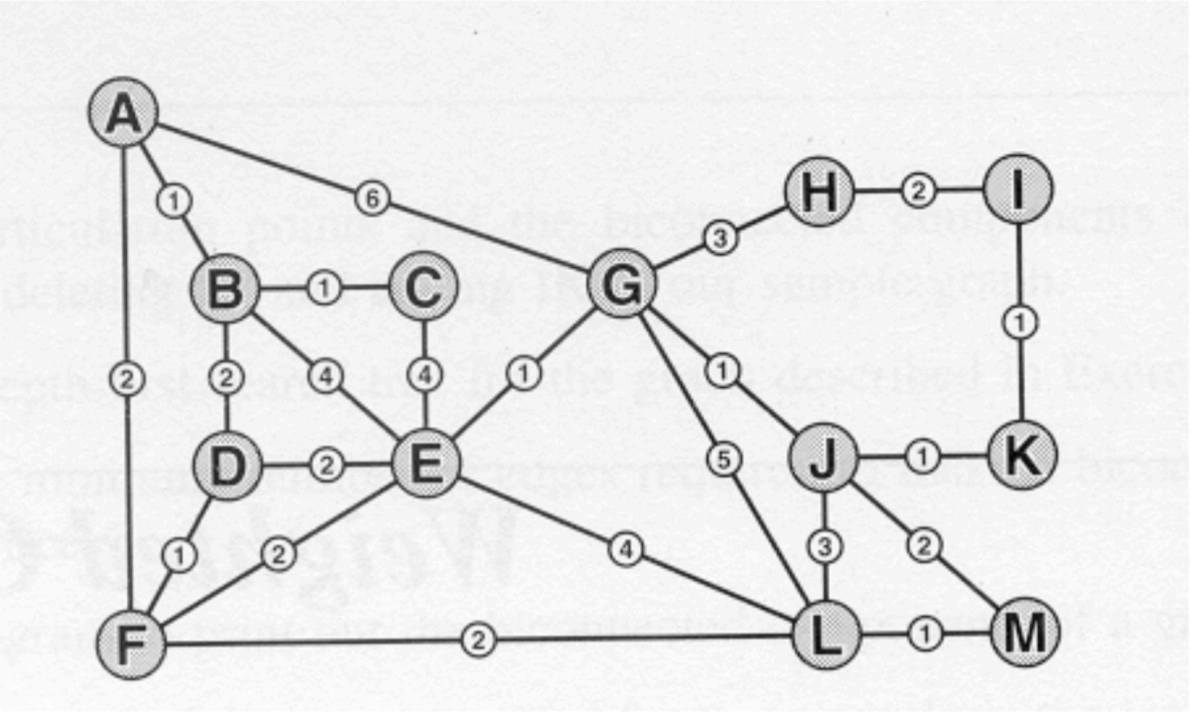
In this assignment I have managed to implement the following algorithms: Depth First Search (DFS) traversal Cormen's version, Breadth First Search (BFS) traversal Cormen's version, MST Prim's algorithm and SPT Dijkstra's algorithm. I have successfully applied them on the input graph given by Mr. Richard. Then, I have found a large roads network graph for a small European country called Gibraltar and ran Dijkstra's algorithm and constructed Shortest Path Tree from source vertex. In

this report, I will provide all the necessary explanations with details and discuss time and space complexity of all algorithms I was dealing with.

2. Adjacency lists diagram for graph from wGraph1.txt

During the code execution in Graph class in Graph constructor we read the lines of the text file and construct an adjacency list where each index points to the vertex's neighbours that are placed into the linked list. Hence, adjacency list is a list of linked lists that shows the graph representation of the sample graph. In this approach every vertex has a linked list associated with it which only records those vertices connected to it with an edge. The edge weights are also recorded. So in undirected graph, each edge is represented by two linked list nodes.

I was given this input graph:



For an undirected graph with V vertices, the maximum number of edges is:

$$\frac{V(V-1)}{2}$$

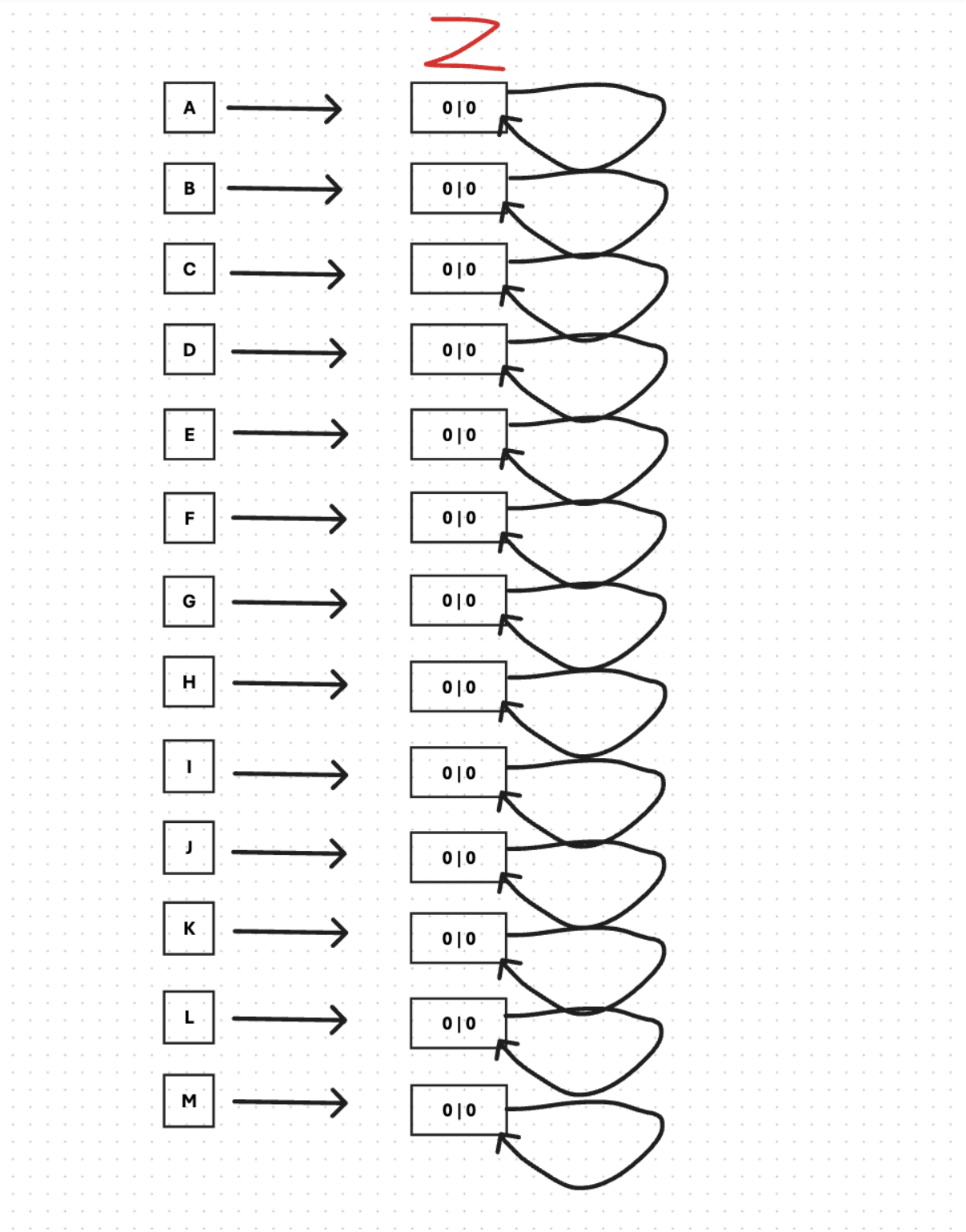
Maximum possible edges:

$$\frac{13 \times 12}{2} = 78$$

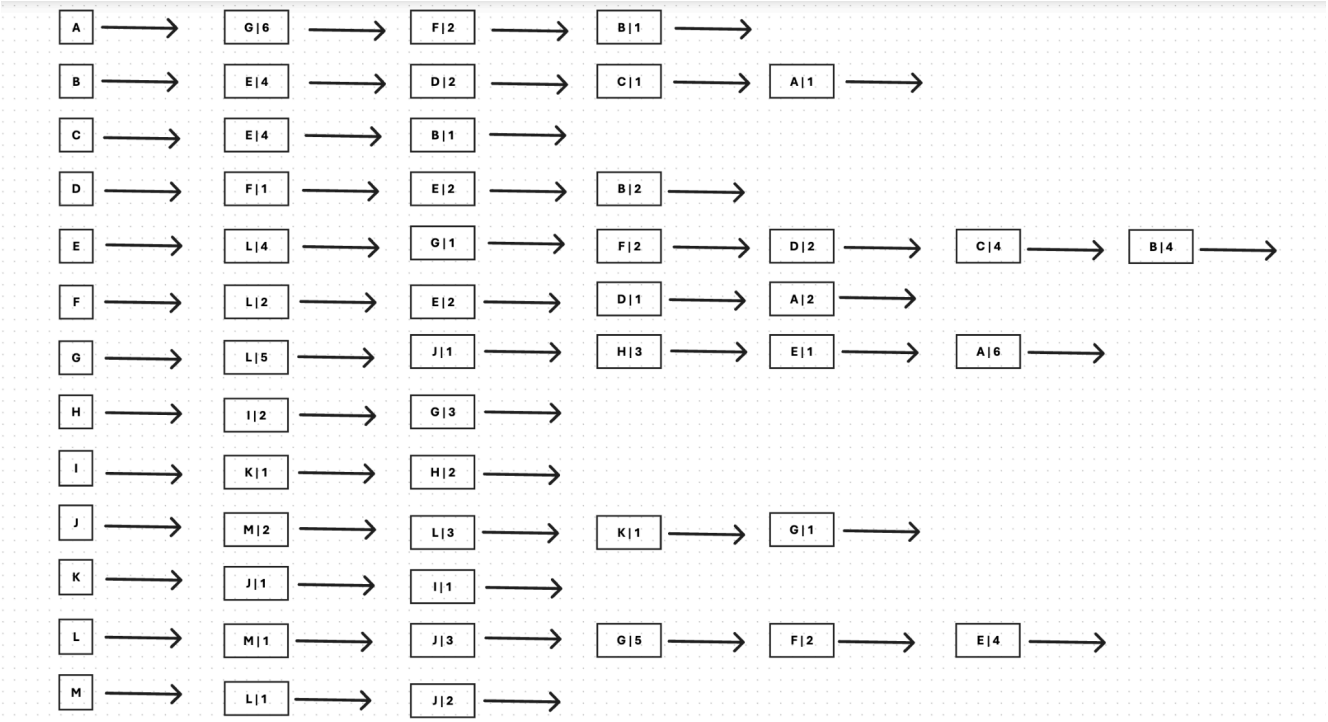
There are only 22 edges in the input graph hence the graph above is a sparse graph because the number of edges is much less than the maximum possible number of edges. Therefore, the usage of Adjacency Linked Lists is reasonable because it's highly efficient for sparse graphs.

Initially, before running all algorithms I created adjacency lists, initialised to sentinel node z

```
// create sentinel node

        z = new Node(0, 0, null);

        z.next = z;



        // create adjacency lists, initialised to sentinel node
z

        adj = new Node[V+1];

        for(v = 1; v <= V; ++v)

            adj[v] = z;
```



After constructing the graph from text file, adjacency lists diagram looks this way:



Where the node in the linked list contains the vertex itself, the weight of the edge to reach the current vertex and the link to the next node in the linked list.

Inside the class Graph I implemented class Node where each node has vertex integer value, weight integer value representing the cost to reach this node and pointer to the next node in the linked list.

```
class Node {

        public int vertex;

        public int wgt;

        public Node next;


        public Node(int vertex, int wgt, Node next) {

            this.vertex = vertex;

            this.wgt = wgt;

            this.next = next;

        }

    }
```

Logs during building the graph, start from vertex L that is 12:

```
└─ java GraphSolution.java
Student name: Ian Miller
Student number: D23124620

Enter graph file name (eg. wGraph.txt): wGraph.txt
Enter starting vertex (as a number, e.g., 1 for A): 12
Parts[] = 13 22
Reading edges from text file
Edge A--(1)--B
Edge A--(2)--F
Edge A--(6)--G
Edge B--(1)--C
Edge B--(2)--D
Edge B--(4)--E
Edge C--(4)--E
Edge D--(2)--E
Edge D--(1)--F
Edge E--(2)--F
Edge E--(1)--G
Edge E--(4)--L
Edge F--(2)--L
Edge G--(3)--H
Edge G--(1)--J
Edge G--(5)--L
Edge H--(2)--I
Edge I--(1)--K
Edge J--(1)--K
Edge J--(3)--L
Edge J--(2)--M
Edge L--(1)--M

Building adjacency list representation, which is very efficient for sparse graphs.

adj[A] -> |G | 6| -> |F | 2| -> |B | 1| ->
adj[B] -> |E | 4| -> |D | 2| -> |C | 1| -> |A | 1| ->
adj[C] -> |E | 4| -> |B | 1| ->
adj[D] -> |F | 1| -> |E | 2| -> |B | 2| ->
adj[E] -> |L | 4| -> |G | 1| -> |F | 2| -> |D | 2| -> |C | 4| -> |B | 4| ->
adj[F] -> |L | 2| -> |E | 2| -> |D | 1| -> |A | 2| ->
adj[G] -> |L | 5| -> |J | 1| -> |H | 3| -> |E | 1| -> |A | 6| ->
adj[H] -> |I | 2| -> |G | 3| ->
adj[I] -> |K | 1| -> |H | 2| ->
adj[J] -> |M | 2| -> |L | 3| -> |K | 1| -> |G | 1| ->
adj[K] -> |J | 1| -> |I | 1| ->
adj[L] -> |M | 1| -> |J | 3| -> |G | 5| -> |F | 2| -> |E | 4| ->
adj[M] -> |L | 1| -> |J | 2| ->
```

3. Depth First Search on graph from wGraph1.txt

I implemented DFS relying on Cormen’s pseudocode from his book:

Cormen's Alg

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1          // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]    // explore edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK          // blacken u; it is finished
9  time = time + 1
10 u.f = time
```

- Uses recursive traversal
- Colors: White -> Grey -> Black
- Parent array is maintained
- Discovery and Finish timestamps are maintained
- Total time is kept tract

Next, I will provide screen captures showing logs during DFS code execution step by step:

- At the beginning each vertex in graph is marked as White (not fully processed) and parent for each vertex is assigned to 0 since 0 is dummy value (in graph we use vertices starting from 1 and so on)

```
1) Preparing for DFS Traversal Cormen's version with colouring

Vertex A is marked as White
Parent of Vertex A is assigned to 0

Vertex B is marked as White
Parent of Vertex B is assigned to 0

Vertex C is marked as White
Parent of Vertex C is assigned to 0

Vertex D is marked as White
Parent of Vertex D is assigned to 0

Vertex E is marked as White
Parent of Vertex E is assigned to 0

Vertex F is marked as White
Parent of Vertex F is assigned to 0

Vertex G is marked as White
Parent of Vertex G is assigned to 0

Vertex H is marked as White
Parent of Vertex H is assigned to 0

Vertex I is marked as White
Parent of Vertex I is assigned to 0

Vertex J is marked as White
Parent of Vertex J is assigned to 0

Vertex K is marked as White
Parent of Vertex K is assigned to 0

Vertex L is marked as White
Parent of Vertex L is assigned to 0

Vertex M is marked as White
Parent of Vertex M is assigned to 0

Starting Depth First Graph Traversal Cormen's version
Starting with Vertex L prompted by user
```

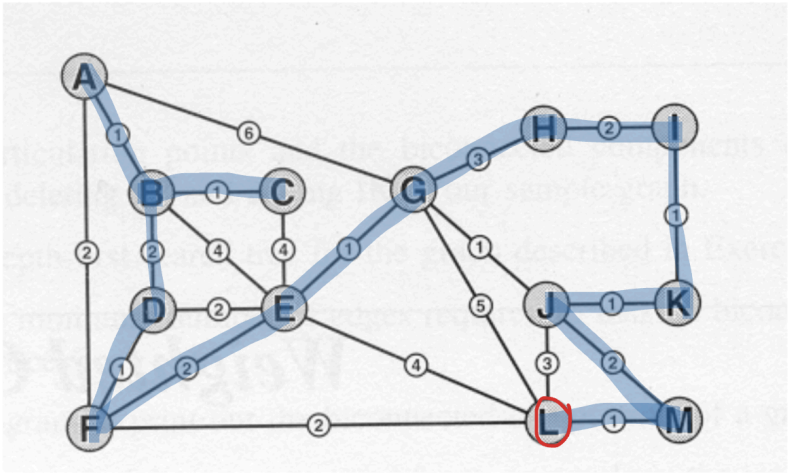
- Then run Depth First from source vertex L that is 12:

```
Starting Depth First Graph Traversal Cormen's version
Starting with Vertex L prompted by user

DF just visited starting vertex L | Discovery time: 1 | Vertex L is marked as Grey
Visited vertex M from L | Discovery time: 2 | Vertex M is marked as Grey
Visited vertex J from M | Discovery time: 3 | Vertex J is marked as Grey
Visited vertex K from J | Discovery time: 4 | Vertex K is marked as Grey
Visited vertex I from K | Discovery time: 5 | Vertex I is marked as Grey
Visited vertex H from I | Discovery time: 6 | Vertex H is marked as Grey
Visited vertex G from H | Discovery time: 7 | Vertex G is marked as Grey
Visited vertex E from G | Discovery time: 8 | Vertex E is marked as Grey
Visited vertex F from E | Discovery time: 9 | Vertex F is marked as Grey
Visited vertex D from F | Discovery time: 10 | Vertex D is marked as Grey
Visited vertex B from D | Discovery time: 11 | Vertex B is marked as Grey
Visited vertex C from B | Discovery time: 12 | Vertex C is marked as Grey
Finished vertex C | Finish time: 13 | Vertex C is marked as Black
Visited vertex A from B | Discovery time: 14 | Vertex A is marked as Grey
Finished vertex A | Finish time: 15 | Vertex A is marked as Black
Finished vertex B | Finish time: 16 | Vertex B is marked as Black
Finished vertex D | Finish time: 17 | Vertex D is marked as Black
Finished vertex F | Finish time: 18 | Vertex F is marked as Black
Finished vertex E | Finish time: 19 | Vertex E is marked as Black
Finished vertex G | Finish time: 20 | Vertex G is marked as Black
Finished vertex H | Finish time: 21 | Vertex H is marked as Black
Finished vertex I | Finish time: 22 | Vertex I is marked as Black
Finished vertex K | Finish time: 23 | Vertex K is marked as Black
Finished vertex J | Finish time: 24 | Vertex J is marked as Black
Finished vertex M | Finish time: 25 | Vertex M is marked as Black
Finished vertex L | Finish time: 26 | Vertex L is marked as Black

Traversal Tree (parent array):
A ← B
B ← D
C ← B
D ← F
E ← G
F ← E
G ← H
H ← I
I ← K
J ← M
K ← J
L ← Root of tree
M ← L

Time complexity: O(V + E), Space complexity: O(V)
```

We record the discovery time and finish time for each vertex. When we firstly visit a vertex, it is marked as Grey and discovery time is recorded, then we need to visit all the neighbours that are marked as White (unvisited) and when all the neighbours of current vertex are visited, calls begin to pop from call stack and when recursive call returns back we mark vertex as Black that means the vertex is fully finished and finished time for vertex is recorded.

Finally, I created the diagram showing the DFS traversal superimposed on the graph:

DFS



Source: L

Analyzing time and space complexity for DFS:

```
Time complexity: O(V + E), Space complexity: O(V)
```

As for time complexity, we process the total number of vertices and edges. As for space complexity, we maintain an array with adjacency linked lists, an array for keeping parent for each vertex, an array d[u] that stores the discovery time of vertex u when mark node u as Grey, an array f[u] that timestamps when mark node as Black and record finished time. All these arrays have fixed size of V + 1 (because start from 1, 0 is dummy).

3. Breadth First Search on graph from wGraph1.txt

I implemented BFS relying on Cormen's pseudocode from his book:

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

- Uses a Queue data structure (FIFO)
- Colors: White -> Grey -> Black
- Distance array d[u] is maintained to store the distance from source vertex s to vertex u (number of edges)
- Parent array is maintained as well

Next, I will provide screen captures showing logs during BFS code execution step by step:

- At the beginning each vertex in graph is marked as White (not fully processed) and parent for each vertex is assigned to 0 since 0 is dummy value (in graph we use vertices starting from 1 and so on)

```
2) Preparing for BFS Traversal Cormen's version with colouring

Vertex A is marked as White

Vertex B is marked as White

Vertex C is marked as White

Vertex D is marked as White

Vertex E is marked as White

Vertex F is marked as White

Vertex G is marked as White

Vertex H is marked as White

Vertex I is marked as White

Vertex J is marked as White

Vertex K is marked as White

Vertex L is marked as White

Vertex M is marked as White
```

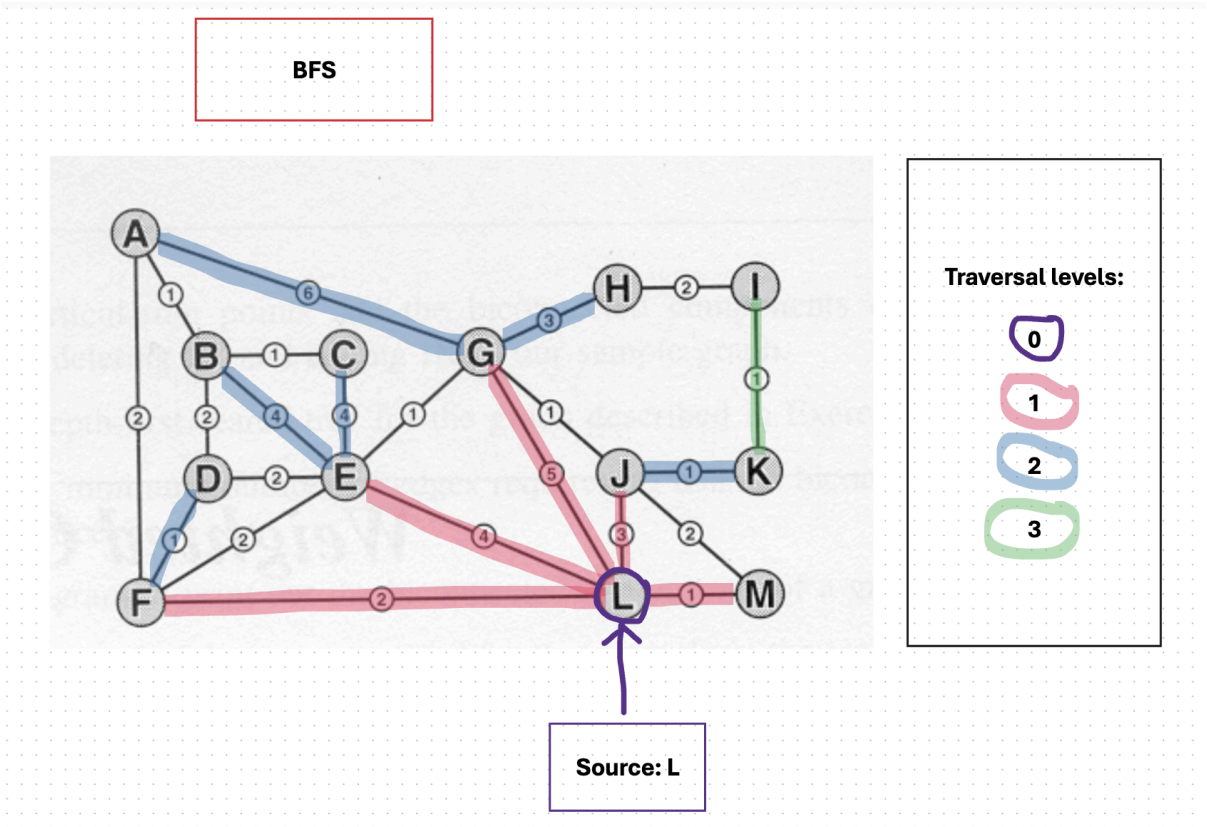
- Then run Breadth First from source vertex L that is 12:

  I also have added a small improvement to Cormen's implementation. I keep track of current traversal level to easily print how BFS traverses the graph level by level and how Queue is changing during traversal. Traversal level starts from 0 because it takes 0 edges to reach the source vertex.

```
Starting Breadth First Traversal Cormen's version

Starting with Vertex L prompted by user
Queue now: [ L ]
Current traversal level: 0

BF just visited starting vertex L | Distance from source L to vertex L is 0 edges | Vertex L was marked as Grey
Vertex M is enqued to the queue |
Queue now: [ M ]
Vertex J is enqued to the queue |
Queue now: [ M J ]
Vertex G is enqued to the queue |
Queue now: [ M J G ]
Vertex F is enqued to the queue |
Queue now: [ M J G F ]
Vertex E is enqued to the queue |
Queue now: [ M J G F E ]
Vertex L is marked as Black

Current traversal level: 1
Visited vertex M from L | Distance from source L to vertex M is 1 edge | Vertex M was marked as Grey
Vertex M is marked as Black

Visited vertex J from L | Distance from source L to vertex J is 1 edge | Vertex J was marked as Grey
Vertex K is enqued to the queue |
Queue now: [ G F E K ]
Vertex J is marked as Black

Visited vertex G from L | Distance from source L to vertex G is 1 edge | Vertex G was marked as Grey
Vertex H is enqued to the queue |
Queue now: [ F E K H ]
Vertex A is enqued to the queue |
Queue now: [ F E K H A ]
Vertex G is marked as Black

Visited vertex F from L | Distance from source L to vertex F is 1 edge | Vertex F was marked as Grey
Vertex D is enqued to the queue |
Queue now: [ E K H A D ]
Vertex F is marked as Black

Visited vertex E from L | Distance from source L to vertex E is 1 edge | Vertex E was marked as Grey
Vertex C is enqued to the queue |
Queue now: [ K H A D C ]
Vertex B is enqued to the queue |
Queue now: [ K H A D C B ]
Vertex E is marked as Black

Current traversal level: 2
Visited vertex K from J | Distance from source L to vertex K is 2 edges | Vertex K was marked as Grey
Vertex I is enqued to the queue |
Queue now: [ H A D C B I ]
Vertex K is marked as Black

Visited vertex H from G | Distance from source L to vertex H is 2 edges | Vertex H was marked as Grey
Vertex H is marked as Black

Visited vertex A from G | Distance from source L to vertex A is 2 edges | Vertex A was marked as Grey
Vertex A is marked as Black

Visited vertex D from F | Distance from source L to vertex D is 2 edges | Vertex D was marked as Grey
Vertex D is marked as Black

Visited vertex C from E | Distance from source L to vertex C is 2 edges | Vertex C was marked as Grey
Vertex C is marked as Black

Visited vertex B from E | Distance from source L to vertex B is 2 edges | Vertex B was marked as Grey
Vertex B is marked as Black

Current traversal level: 3
Visited vertex I from K | Distance from source L to vertex I is 3 edges | Vertex I was marked as Grey
Vertex I is marked as Black

Total traversal levels in graph processed: 3
```

```
Traversal Tree (parent array):
A ← G
B ← E
C ← E
D ← F
E ← L
F ← L
G ← L
H ← G
I ← K
J ← L
K ← J
L ← Root of tree
M ← L

Time complexity: O(V + E), Space complexity: O(V)
```

Finally, I created the diagram showing the BFS traversal superimposed on the graph:



Analyzing time and space complexity for BFS:

```
Time complexity: O(V + E), Space complexity: O(V)
```

Time and Space complexity is pretty much the same as for DFS. We still process all the vertices and edges in the graph and maintain adjacency linked lists, colours array, parent array and distance array for every vertex.

4. Construction of the MST using Prim's algorithm for the graph from wGraph1.txt

Minimum Spanning Tree is such a tree thats is connected, weighted and undirected graph where:

- all vertices are connected
- there are no cycles
- number of edges in MST equals to V - 1, where V is total number of vertices
- the total sum of edges weights is minimised

I implemented MST Prim's algorithm relying on this pseudocode:

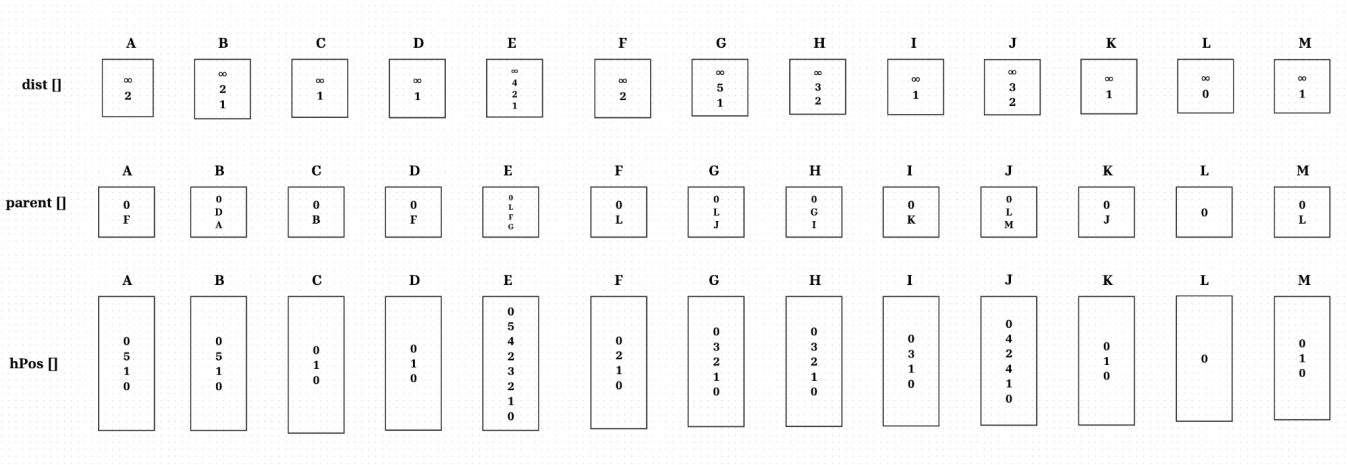
Prim's MST Algorithm on Adjacency Lists

```
Prim_Lists( Vertex s )
Begin
    // G = (V, E)
    foreach v ∈ V
        dist[v] := ∞
        parent[v] := 0                  // treat 0 as a special null vertex
        hPos[v] := 0                    // indicates that v ∉ heap

    h = new Heap(|V|, hPos, dist)       // priority queue (heap) initially empty
    h.insert(s)                         // s will be the root of the MST

    while (not h.isEmpty() )            // should repeat  |V|-1  times
        v := h.remove()                 // add v to the MST
        dist[v] := -dist[v]             // marks v as now in the MST
        foreach u ∈ adj(v)              // examine each neighbour u of v
            if wgt(v, u) < dist[u]
                dist[u] := wgt(v, u)
                parent[u] := v
                if u ∉ h
                    h.insert( u)
                else
                    h.siftUp( hPos[u])

            end if
        end for
    end while
    return parent
End
```

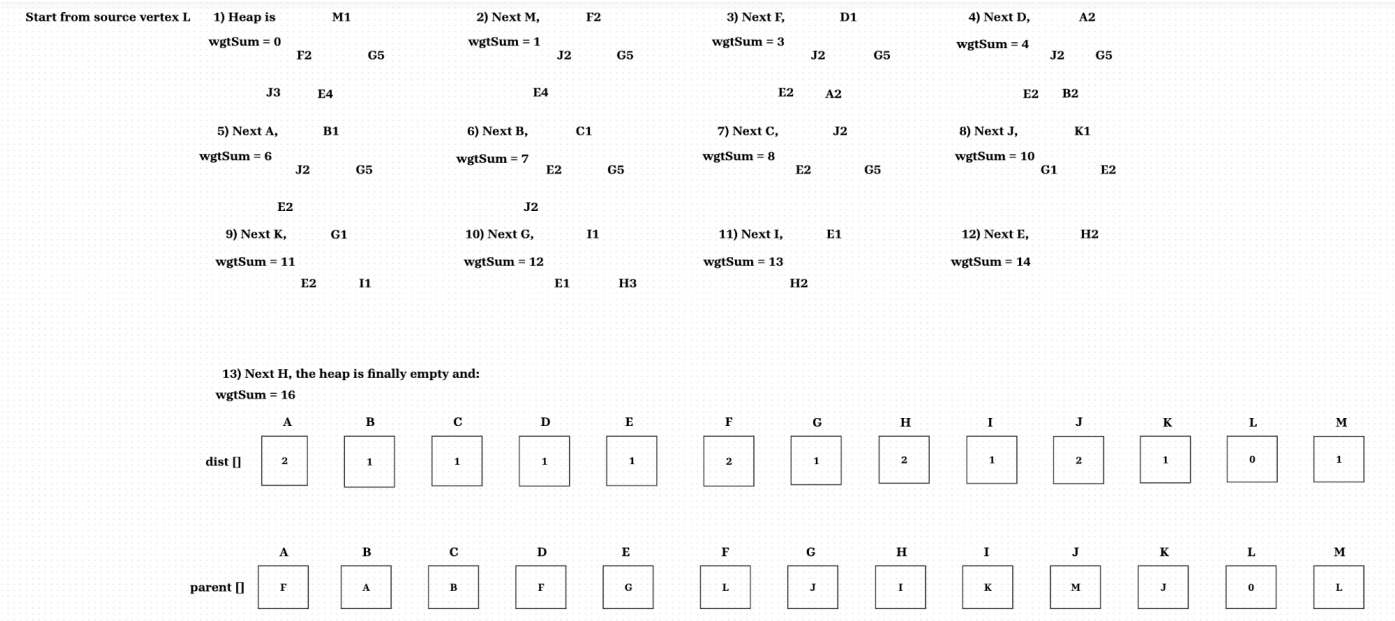
A step by step construction of the MST using Prim's algorithm starting from vertex L that is 12:

- The contents of dist[], parent[] and hPos[] arrays for each step in Prim (the content in cells on my drawings should be read from up to bottom)

  Initially, the priority for each vertex in dist[] array is ∞ and for each vertex the parent is assigned to 0 and the position in the heap is 0 as well. And then Prim's algorithm works and changes the contents of the arrays.



- The contents of the heap for each step in Prim and final results



Next, I will provide screen captures showing logs during MST Prim's code execution step by step:

```
3) Preparing for running Prim's MST Algorithm on Adjacency Lists

Starting MST Prim's algorithm:

Start from source vertex: L, dist = 0, hPos[s] = 1, current total MST weight = 0
Removed from heap: vertex L, dist = 0 (if priority is negative, it means vertex is already in MST), current total MST weight = 0
dist[]: A=∞  B=∞  C=∞  D=∞  E=4  F=2  G=5  H=∞  I=∞  J=3  K=∞  L=0  M=1
hPos[]: A=0  B=0  C=0  D=0  E=5  F=2  G=3  H=0  I=0  J=4  K=0  L=0  M=1
parent[]: A=@  B=@  C=@  D=@  E=L  F=L  G=L  H=@  I=@  J=L  K=@  L=@  M=L

Removed from heap: vertex M, dist = -1 (if priority is negative, it means vertex is already in MST), current total MST weight = 1
Called siftUp() on vertex: J
dist[]: A=∞  B=∞  C=∞  D=∞  E=4  F=2  G=5  H=∞  I=∞  J=2  K=∞  L=0  M=-1
hPos[]: A=0  B=0  C=0  D=0  E=4  F=1  G=3  H=0  I=0  J=2  K=0  L=0  M=0
parent[]: A=@  B=@  C=@  D=@  E=L  F=L  G=L  H=@  I=@  J=M  K=@  L=@  M=L

Removed from heap: vertex F, dist = -2 (if priority is negative, it means vertex is already in MST), current total MST weight = 3
Called siftUp() on vertex: E
dist[]: A=2  B=∞  C=∞  D=1  E=2  F=-2  G=5  H=∞  I=∞  J=2  K=∞  L=0  M=-1
hPos[]: A=5  B=0  C=0  D=1  E=4  F=0  G=3  H=0  I=0  J=2  K=0  L=0  M=0
parent[]: A=F  B=@  C=@  D=F  E=F  F=L  G=L  H=@  I=@  J=M  K=@  L=@  M=L

Removed from heap: vertex D, dist = -1 (if priority is negative, it means vertex is already in MST), current total MST weight = 4
dist[]: A=2  B=2  C=∞  D=-1  E=2  F=-2  G=5  H=∞  I=∞  J=2  K=∞  L=0  M=-1
hPos[]: A=1  B=5  C=0  D=0  E=4  F=0  G=3  H=0  I=0  J=2  K=0  L=0  M=0
parent[]: A=F  B=D  C=@  D=F  E=F  F=L  G=L  H=@  I=@  J=M  K=@  L=@  M=L

Removed from heap: vertex A, dist = -2 (if priority is negative, it means vertex is already in MST), current total MST weight = 6
Called siftUp() on vertex: B
dist[]: A=-2  B=1  C=∞  D=-1  E=2  F=-2  G=5  H=∞  I=∞  J=2  K=∞  L=0  M=-1
hPos[]: A=0  B=1  C=0  D=0  E=4  F=0  G=3  H=0  I=0  J=2  K=0  L=0  M=0
parent[]: A=F  B=A  C=@  D=F  E=F  F=L  G=L  H=@  I=@  J=M  K=@  L=@  M=L

Removed from heap: vertex B, dist = -1 (if priority is negative, it means vertex is already in MST), current total MST weight = 7
dist[]: A=-2  B=-1  C=1  D=-1  E=2  F=-2  G=5  H=∞  I=∞  J=2  K=∞  L=0  M=-1
hPos[]: A=0  B=0  C=1  D=0  E=2  F=0  G=3  H=0  I=0  J=4  K=0  L=0  M=0
parent[]: A=F  B=A  C=B  D=F  E=F  F=L  G=L  H=@  I=@  J=M  K=@  L=@  M=L

Removed from heap: vertex C, dist = -1 (if priority is negative, it means vertex is already in MST), current total MST weight = 8
dist[]: A=-2  B=-1  C=-1  D=-1  E=2  F=-2  G=5  H=∞  I=∞  J=2  K=∞  L=0  M=-1
hPos[]: A=0  B=0  C=0  D=0  E=2  F=0  G=3  H=0  I=0  J=1  K=0  L=0  M=0
parent[]: A=F  B=A  C=B  D=F  E=F  F=L  G=L  H=@  I=@  J=M  K=@  L=@  M=L

Removed from heap: vertex J, dist = -2 (if priority is negative, it means vertex is already in MST), current total MST weight = 10
Called siftUp() on vertex: G
dist[]: A=-2  B=-1  C=-1  D=-1  E=2  F=-2  G=1  H=∞  I=∞  J=-2  K=1  L=0  M=-1
hPos[]: A=0  B=0  C=0  D=0  E=3  F=0  G=2  H=0  I=0  J=0  K=1  L=0  M=0
parent[]: A=F  B=A  C=B  D=F  E=F  F=L  G=J  H=@  I=@  J=M  K=J  L=@  M=L

Removed from heap: vertex K, dist = -1 (if priority is negative, it means vertex is already in MST), current total MST weight = 11
dist[]: A=-2  B=-1  C=-1  D=-1  E=2  F=-2  G=1  H=∞  I=1  J=-2  K=-1  L=0  M=-1
hPos[]: A=0  B=0  C=0  D=0  E=2  F=0  G=1  H=0  I=3  J=0  K=0  L=0  M=0
parent[]: A=F  B=A  C=B  D=F  E=F  F=L  G=J  H=@  I=K  J=M  K=J  L=@  M=L

Removed from heap: vertex G, dist = -1 (if priority is negative, it means vertex is already in MST), current total MST weight = 12
Called siftUp() on vertex: E
dist[]: A=-2  B=-1  C=-1  D=-1  E=1  F=-2  G=-1  H=3  I=1  J=-2  K=-1  L=0  M=-1
hPos[]: A=0  B=0  C=0  D=0  E=2  F=0  G=0  H=3  I=1  J=0  K=0  L=0  M=0
parent[]: A=F  B=A  C=B  D=F  E=G  F=L  G=J  H=G  I=K  J=M  K=J  L=@  M=L

Removed from heap: vertex I, dist = -1 (if priority is negative, it means vertex is already in MST), current total MST weight = 13
Called siftUp() on vertex: H
dist[]: A=-2  B=-1  C=-1  D=-1  E=1  F=-2  G=-1  H=2  I=-1  J=-2  K=-1  L=0  M=-1
hPos[]: A=0  B=0  C=0  D=0  E=1  F=0  G=0  H=2  I=0  J=0  K=0  L=0  M=0
parent[]: A=F  B=A  C=B  D=F  E=G  F=L  G=J  H=I  I=K  J=M  K=J  L=@  M=L

Removed from heap: vertex E, dist = -1 (if priority is negative, it means vertex is already in MST), current total MST weight = 14
dist[]: A=-2  B=-1  C=-1  D=-1  E=-1  F=-2  G=-1  H=2  I=-1  J=-2  K=-1  L=0  M=-1
hPos[]: A=0  B=0  C=0  D=0  E=0  F=0  G=0  H=1  I=0  J=0  K=0  L=0  M=0
parent[]: A=F  B=A  C=B  D=F  E=G  F=L  G=J  H=I  I=K  J=M  K=J  L=@  M=L

Removed from heap: vertex H, dist = -2 (if priority is negative, it means vertex is already in MST), current total MST weight = 16
dist[]: A=-2  B=-1  C=-1  D=-1  E=-1  F=-2  G=-1  H=-2  I=-1  J=-2  K=-1  L=0  M=-1
hPos[]: A=0  B=0  C=0  D=0  E=0  F=0  G=0  H=0  I=0  J=0  K=0  L=0  M=0
parent[]: A=F  B=A  C=B  D=F  E=G  F=L  G=J  H=I  I=K  J=M  K=J  L=@  M=L


There are 13 vertices and 22 edges in the input graph
After running Prim's MST Algorithm on Adjacency Lists:
Weight of MST = 16
Number of vertices connected in MST = 13
Number of edges in MST = 12 (should be equal to V - 1)

Edges in Minimum Spanning Tree (parent -> child):
F --(2)--> A
A --(1)--> B
B --(1)--> C
F --(1)--> D
G --(1)--> E
L --(2)--> F
J --(1)--> G
I --(2)--> H
K --(1)--> I
M --(2)--> J
J --(1)--> K
L --(1)--> M

Time complexity: O(E log V), Space complexity: O(V + E)
```

Analyzing time and space complexity for MST Prim's algorithm for Adjacency Lists Graph with Heap:

```
Time complexity: O(E log V), Space complexity: O(V + E)
```

Heap contains up to V vertices. So a heap operation requires a maximum of logV steps.

Initialisation for loop – requires V steps

while loop:

- each vertex is removed from heap once – so requires at most V logV steps
- foreach loop within the while loop causes all the linked list nodes of the graph representation to be iterated thru once and since each list node represents an edge, the total number of iterations is E
- so each edge is examined once and may involve an insert() or a siftUp() . In total this means at most E logV steps.

Therefore, time complexity: O(V + V logV +E logV) = O(E logV)

Space complexity: O(V + E) because:

- maintain adjacency linked lists that gives us O(V + E)
- dist[], parent[], hPos[] and heap cost O(V)

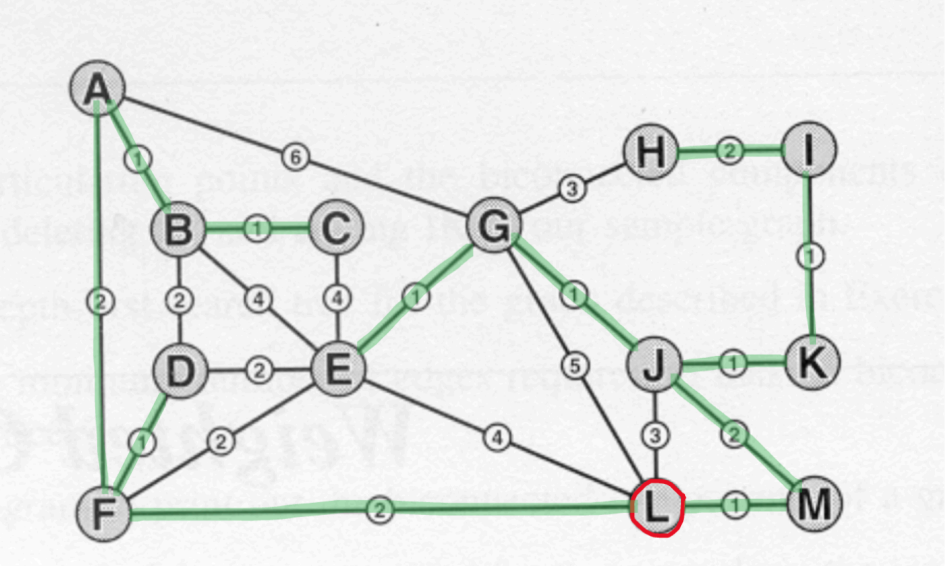
5. Diagram showing the MST superimposed on the graph from wGraph1.txt

V in MST = 13

E in MST = 12 (V - 1)

Weight of MST = 16

MST using Prim's algorithm



V - vertices

E - edges

L is source vertex

6. Construction of the SPT using Dijkstra's algorithm for the graph from wGraph1.txt

The Shortest Path Tree algorithm is very identical to MST but the goal here is to find the shortest path from source to each vertex in the graph. Furthemore, SPT doesn't care about connecting all vertices with minimal total weight like MST.

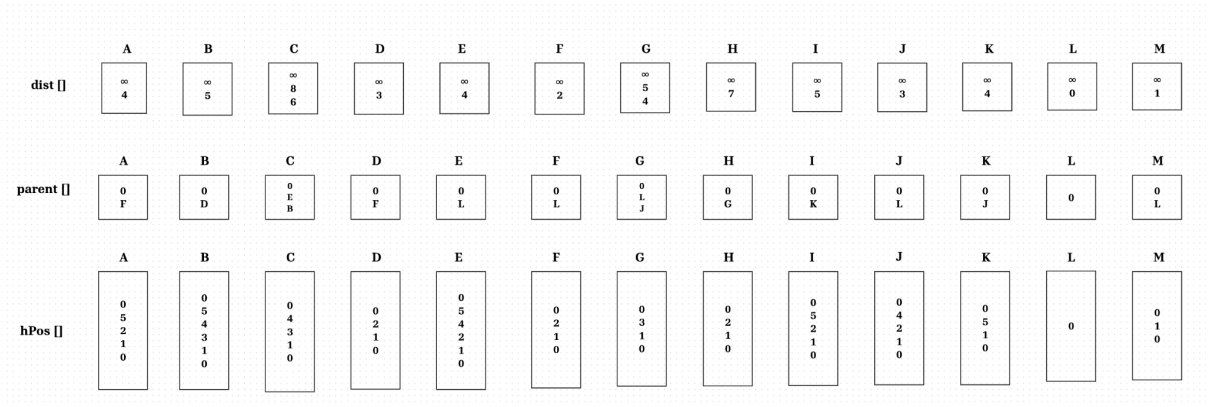
I implemented SPT Dijkstra's algorithm relying on this pseudocode:

```
Graph::SPT_Dijkstra( vertex s )
Begin
    // G = (V, E)
    for each v ∈ V
        dist[v] = ∞
        parent[v] = null        // have a special null vertex
        hpos[v] = 0             // indicates that v ∉ heap
    pq = new Heap               // priority queue (heap) initially empty
    v = s                       // s will be the root of the shortest path tree
    dist[s] = 0

    while v ≠ null                      // Loop should repeat  |V|-1  times

        for each u ∈ adj(v)                 // Examine each neighbour u of v
            d = wgt(v,u)
            if dist[v] + d < dist[u]
                dist[u] = dist[v] + d       // After changing dist[u] either insert
                if u ∉ pq                   // it or sift it up in the heap and record
                    pq.insert( u)           // its new parent vertex.
                else
                    pq.siftUp( hpos[u])
                parent[u] = v
            end if
        end for
        v = pq.remove()

    while end
    return parent
End
```

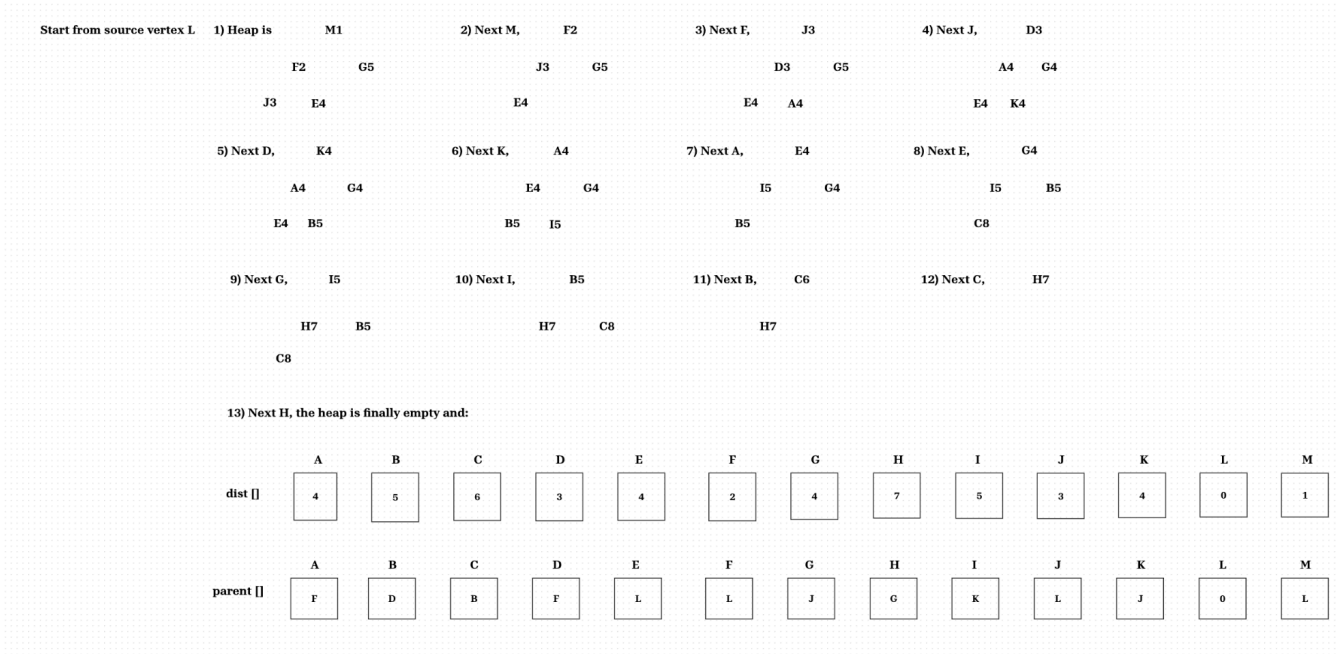
A step by step construction of the SPT using Dijkstra's algorithm starting from vertex L that is 12:

- The contents of dist[], parent[] and hPos[] arrays for each step in Dijkstra (the content in cells on my drawings should be read from up to bottom)

  Initially, the distance for each vertex in dist[] array is ∞ and for each vertex the parent is assigned to 0 and the position in the heap is 0 as well. And then Dikstra's algorithm works and changes the contents of the arrays and finds the shortest path from the source L to each vertex in graph



- The contents of the heap for each step in Dijkstra and final results



Next, I will provide screen captures showing logs during MST Prim's code execution step by step:

```
4) Preparing for running Dijkstra's Shortest Path Tree Algorithm on Adjacency Lists

Starting SPT Dijkstra's algorithm:

Start from source vertex: L, dist = 0, hPos[s] = 1
Removed from heap: vertex L, dist = 0
dist[]: A=∞  B=∞  C=∞  D=∞  E=4  F=2  G=5  H=∞  I=∞  J=3  K=∞  L=0  M=1
hPos[]: A=0  B=0  C=0  D=0  E=5  F=2  G=3  H=0  I=0  J=4  K=0  L=0  M=1
parent[]: A=@  B=@  C=@  D=@  E=L  F=L  G=L  H=@  I=@  J=L  K=@  L=@  M=L

Removed from heap: vertex M, dist = 1
dist[]: A=∞  B=∞  C=∞  D=∞  E=4  F=2  G=5  H=∞  I=∞  J=3  K=∞  L=0  M=1
hPos[]: A=0  B=0  C=0  D=0  E=4  F=1  G=3  H=0  I=0  J=2  K=0  L=0  M=0
parent[]: A=@  B=@  C=@  D=@  E=L  F=L  G=L  H=@  I=@  J=L  K=@  L=@  M=L

Removed from heap: vertex F, dist = 2
dist[]: A=4  B=∞  C=∞  D=3  E=4  F=2  G=5  H=∞  I=∞  J=3  K=∞  L=0  M=1
hPos[]: A=5  B=0  C=0  D=2  E=4  F=0  G=3  H=0  I=0  J=1  K=0  L=0  M=0
parent[]: A=F  B=@  C=@  D=F  E=L  F=L  G=L  H=@  I=@  J=L  K=@  L=@  M=L

Removed from heap: vertex J, dist = 3
Called siftUp() on vertex: G
dist[]: A=4  B=∞  C=∞  D=3  E=4  F=2  G=4  H=∞  I=∞  J=3  K=4  L=0  M=1
hPos[]: A=2  B=0  C=0  D=1  E=4  F=0  G=3  H=0  I=0  J=0  K=5  L=0  M=0
parent[]: A=F  B=@  C=@  D=F  E=L  F=L  G=J  H=@  I=@  J=L  K=J  L=@  M=L

Removed from heap: vertex D, dist = 3
dist[]: A=4  B=5  C=∞  D=3  E=4  F=2  G=4  H=∞  I=∞  J=3  K=4  L=0  M=1
hPos[]: A=2  B=5  C=0  D=0  E=4  F=0  G=3  H=0  I=0  J=0  K=1  L=0  M=0
parent[]: A=F  B=D  C=@  D=F  E=L  F=L  G=J  H=@  I=@  J=L  K=J  L=@  M=L

Removed from heap: vertex K, dist = 4
dist[]: A=4  B=5  C=∞  D=3  E=4  F=2  G=4  H=∞  I=5  J=3  K=4  L=0  M=1
hPos[]: A=1  B=4  C=0  D=0  E=2  F=0  G=3  H=0  I=5  J=0  K=0  L=0  M=0
parent[]: A=F  B=D  C=@  D=F  E=L  F=L  G=J  H=@  I=K  J=L  K=J  L=@  M=L

Removed from heap: vertex A, dist = 4
dist[]: A=4  B=5  C=∞  D=3  E=4  F=2  G=4  H=∞  I=5  J=3  K=4  L=0  M=1
hPos[]: A=0  B=4  C=0  D=0  E=1  F=0  G=3  H=0  I=2  J=0  K=0  L=0  M=0
parent[]: A=F  B=D  C=@  D=F  E=L  F=L  G=J  H=@  I=K  J=L  K=J  L=@  M=L

Removed from heap: vertex E, dist = 4
dist[]: A=4  B=5  C=8  D=3  E=4  F=2  G=4  H=∞  I=5  J=3  K=4  L=0  M=1
hPos[]: A=0  B=3  C=4  D=0  E=0  F=0  G=1  H=0  I=2  J=0  K=0  L=0  M=0
parent[]: A=F  B=D  C=E  D=F  E=L  F=L  G=J  H=@  I=K  J=L  K=J  L=@  M=L

Removed from heap: vertex G, dist = 4
dist[]: A=4  B=5  C=8  D=3  E=4  F=2  G=4  H=7  I=5  J=3  K=4  L=0  M=1
hPos[]: A=0  B=3  C=4  D=0  E=0  F=0  G=0  H=2  I=1  J=0  K=0  L=0  M=0
parent[]: A=F  B=D  C=E  D=F  E=L  F=L  G=J  H=G  I=K  J=L  K=J  L=@  M=L

Removed from heap: vertex I, dist = 5
dist[]: A=4  B=5  C=8  D=3  E=4  F=2  G=4  H=7  I=5  J=3  K=4  L=0  M=1
hPos[]: A=0  B=1  C=3  D=0  E=0  F=0  G=0  H=2  I=0  J=0  K=0  L=0  M=0
parent[]: A=F  B=D  C=E  D=F  E=L  F=L  G=J  H=G  I=K  J=L  K=J  L=@  M=L

Removed from heap: vertex B, dist = 5
Called siftUp() on vertex: C
dist[]: A=4  B=5  C=6  D=3  E=4  F=2  G=4  H=7  I=5  J=3  K=4  L=0  M=1
hPos[]: A=0  B=0  C=1  D=0  E=0  F=0  G=0  H=2  I=0  J=0  K=0  L=0  M=0
parent[]: A=F  B=D  C=B  D=F  E=L  F=L  G=J  H=G  I=K  J=L  K=J  L=@  M=L

Removed from heap: vertex C, dist = 6
dist[]: A=4  B=5  C=6  D=3  E=4  F=2  G=4  H=7  I=5  J=3  K=4  L=0  M=1
hPos[]: A=0  B=0  C=0  D=0  E=0  F=0  G=0  H=1  I=0  J=0  K=0  L=0  M=0
parent[]: A=F  B=D  C=B  D=F  E=L  F=L  G=J  H=G  I=K  J=L  K=J  L=@  M=L

Removed from heap: vertex H, dist = 7
dist[]: A=4  B=5  C=6  D=3  E=4  F=2  G=4  H=7  I=5  J=3  K=4  L=0  M=1
hPos[]: A=0  B=0  C=0  D=0  E=0  F=0  G=0  H=0  I=0  J=0  K=0  L=0  M=0
parent[]: A=F  B=D  C=B  D=F  E=L  F=L  G=J  H=G  I=K  J=L  K=J  L=@  M=L

After running Dijkstra's SPT Algorithm on Adjacency Lists:
Number of vertices connected in SPT = 13
Number of edges in SPT = 12 (should be equal to V - 1)

Shortest Path Tree as it is built is:

Vertex   Parent   Distance from L
A        F        4
B        D        5
C        B        6
D        F        3
E        L        4
F        L        2
G        J        4
H        G        7
I        K        5
J        L        3
K        J        4
L        -        0
M        L        1

Time complexity: O(V + E log V), Space complexity: O(V + E)
```

Analyzing time and space complexity for SPT Dijkstra's algorithm for Adjacency Lists Graph with Heap:

```
Time complexity: O(V + E log V), Space complexity: O(V + E)
```

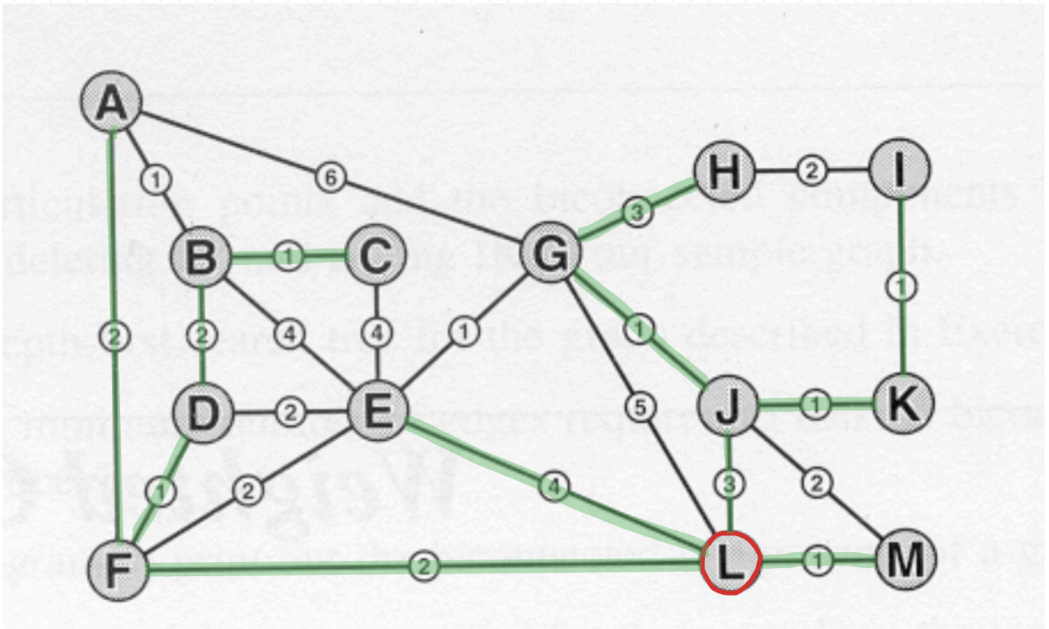
Heap contains vertices. So a heap operation requires a maximum of log V steps.
• Initialization for loop – requires V steps
• Each node bar the first is inserted into heap once - requires at most V log V steps
• Each edge is examined once and may modify a vertex's position on the heap, a siftUP() which requires at most E log V

Therefore, time complexity: O(V + V log V + E log V) ≈ O((V + E) log V)
Space complexity: O(V + E), because store dist[], parent[], hPos[] arrays and adjacency linked lists.

7. Diagram showing the SPT superimposed on the graph from wGraph1.txt

SPT using Dijkstra's algorithm

V in SPT = 13
E in SPT = 12 (V - 1)



V - vertices

E - edges

L is source vertex

8. World Graph - Gibraltar’s roads network

Personally, I have found the graph that represents Gibraltar’s roads network. Gibraltar is a small European country hence the graph is much bigger than the sample graph. The network is small yet complex territory with intricate street layouts and dense road connectivity. Next, I converted the graph into a text file.

Graph properties:

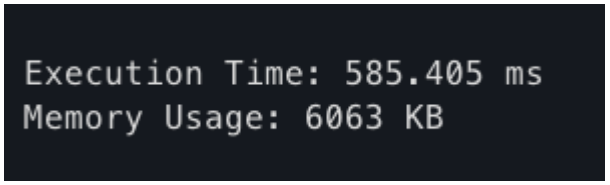
- Vertices (V): 429
- Edges (E): 578
- Edge weight: Length of road segment in meters

As a result, I have got 429 vertices and 578 edges. Therefore, it’s a sparse graph because maximum possible amount of edges is calculated by the formula: (V * (V - 1)) / 2. Potentially, there can be (429 * 428) / 2 = 91,806 maximum edges in the graph but I have only 578. So, I used adjacency linked lists that are a suitable way of storing this sparse graph. This sparse graph is typical for real-world road systems.

SPT Dijkstra’s Algorithm on Gibraltar Road Network

- **Goal**: To construct the Shortest Path Tree (SPT) from a given starting node (in my case, vertex 1), minimizing the total travel distance from this source to all reachable vertices.

- **Performance Summary and complexities:**
  - **Connected vertices:** 429
  - **SPT edges**: 428
  - **Execution time and Memory Usage:** 585.405 ms and 6063 KB



  - **Time complexity:** O((V + E) log V) because each vertex can be inserted/removed to a min heap that costs O(log V) and all edges are examined where each edge might trigger a siftUp if a shorter path is found and this update also takes O(log V) time.
  - **Space Complexity:** O(V + E) because arrays like dist[], parent[], hPos[], and adjacency list representation store all nodes and edges.
  - **Observations:** The algorithm efficiently handled 429 vertices and 578 edges within under a second. All nodes were connected, and Dijkstra correctly calculated the shortest distances in meters. The parent[] array correctly built the traversal tree - each node has a unique parent, ensuring no cycles.

```
After running Dijkstra's SPT Algorithm on Gibraltar's Roads Network:
Number of vertices connected in SPT = 429
Number of edges in SPT = 428 (should be equal to V - 1)

Shortest Path Tree as it is built is:

Vertex   Parent   Distance from source 1 (m)
1        -        0m
2        76       250m
3        2        339m
4        199      619m
5        198      639m
6        7        644m
7        4        635m
8        278      991m
9        11       2024m
10       11       2024m
11       214      2012m
12       344      799m
13       206      1348m
14       12       826m
15       18       562m
16       15       571m
17       18       574m
18       320      557m
19       16       608m
20       21       560m
21       23       473m
22       23       390m
23       421      358m
24       192      895m
25       24       901m
26       126      915m
27       28       916m
28       24       903m
29       26       1157m
30       196      1264m
31       197      1249m
32       176      1207m
33       341      1303m
34       35       1282m
35       307      1260m
36       71       1924m
37       50       2194m
38       37       2565m
39       38       2646m
40       39       2671m
41       51       3652m
42       52       4464m
43       42       4512m
44       185      4923m
45       1        16m
46       174      305m
47       106      421m
48       251      1350m
49       36       1959m
50       118      2049m
51       161      3550m
52       311      4430m
53       56       4817m
54       354      4841m
55       241      4815m
56       241      4809m
57       13       4844m
58       242      3584m
59       139      2698m
60       59       3027m
61       60       3073m
62       392      3396m
63       62       3454m
64       63       3688m
65       165      3364m
66       154      3261m
67       321      3036m
68       203      2970m
69       412      3013m
70       39       2671m
71       222      1833m
72       93       1605m
73       32       1222m
74       293      1392m
```

```
75          79          422m
76          202         59m
77          75          451m
78          76          110m
79          78          276m
80          81          1487m
81          301         1397m
82          124         746m
83          110         4509m
84          83          4781m
85          139         3035m
86          217         2445m
87          333         2347m
88          72          1662m
89          90          1427m
90          96          1356m
91          266         1394m
92          387         1533m
93          386         1498m
94          128         1384m
95          190         1208m
96          95          1239m
97          133         1056m
98          102         915m
99          91          1445m
100         99          1502m
101         404         616m
102         101         849m
103         102         967m
104         145         595m
105         106         324m
106         46          316m
107         17          608m
108         64          4820m
109         108         4872m
110         52          4466m
111         43          4680m
112         261         1934m
113         262         2035m
114         113         2185m
115         252         1792m
116         117         1987m
117         49          1978m
118         117         1995m
119         281         811m
120         121         1132m
121         279         1022m
122         210         1160m
123         124         831m
124         359         700m
125         359         671m
126         25          912m
127         25          962m
128         190         1348m
129         94          1442m
130         94          1414m
131         128         1398m
132         129         1462m
133         384         1022m
134         99          1486m
135         266         1362m
136         286         1653m
137         72          1690m
138         188         2585m
139         207         2660m
140         139         2691m
141         276         1918m
142         31          1267m
143         51          3844m
144         161         3553m
145         244         579m
146         166         1898m
147         49          2000m
148         216         1977m
149         267         1634m
150         153         364m
151         265         1855m
152         256         257m
153         181         323m
154         393         3242m
155         160         300m
```

156	61	3136m
157	225	129m
158	172	304m
159	263	2014m
160	173	250m
161	394	3329m
162	89	1449m
163	228	224m
164	152	303m
165	66	3335m
166	258	1890m
167	146	1918m
168	170	3340m
169	157	214m
170	66	3321m
171	155	408m
172	163	262m
173	348	108m
174	2	273m
175	184	1167m
176	175	1196m
177	3	372m
178	183	1247m
179	176	1248m
180	174	281m
181	46	314m
182	240	385m
183	184	1092m
184	123	972m
185	395	4864m
186	357	4867m
187	357	4866m
188	327	2567m
189	376	1232m
190	133	1106m
191	58	5227m
192	5	696m
193	192	862m
194	35	1274m
195	73	1275m
196	197	1256m
197	32	1238m
198	199	620m
199	77	600m
200	373	754m
201	251	1185m
202	1	31m
203	138	2806m
204	363	2026m
205	92	1665m
206	14	843m
207	327	2560m
208	288	1460m
209	234	1720m
210	27	1003m
211	221	1903m
212	213	1919m
213	214	1910m
214	220	1902m
215	213	1931m
216	236	1970m
217	216	1987m
218	219	1982m
219	236	1971m
220	222	1887m
221	220	1901m
222	205	1819m
223	226	654m
224	226	660m
225	353	117m
226	20	633m
227	223	893m
228	225	215m
229	224	700m
230	233	1501m
231	208	1484m
232	408	1501m
233	231	1485m
234	291	1524m
235	208	1480m
236	215	1957m

237	228	271m
238	122	1168m
239	14	839m
240	78	293m
241	43	4776m
242	38	2603m
243	244	436m
244	105	368m
245	289	2051m
246	427	2557m
247	248	1185m
248	250	1179m
249	250	1182m
250	29	1166m
251	238	1175m
252	253	1638m
253	162	1548m
254	253	1628m
255	135	1401m
256	160	255m
257	143	4520m
258	151	1879m
259	260	2117m
260	112	2019m
261	271	1931m
262	112	1947m
263	151	1885m
264	265	1898m
265	115	1840m
266	96	1358m
267	391	1580m
268	397	1507m
269	159	2069m
270	271	1839m
271	252	1675m
272	149	1673m
273	94	1432m
274	205	1737m
275	274	1742m
276	277	1803m
277	137	1742m
278	280	770m
279	281	802m
280	282	767m
281	278	780m
282	125	743m
283	119	835m
284	283	910m
285	286	1623m
286	287	1577m
287	89	1518m
288	406	1325m
289	233	1503m
290	230	1505m
291	232	1506m
292	293	1396m
293	294	1321m
294	382	1314m
295	296	1275m
296	178	1269m
297	295	1329m
298	297	1351m
299	301	1387m
300	238	1224m
301	352	1270m
302	245	2277m
303	92	1537m
304	307	1221m
305	306	1018m
306	8	1000m
307	305	1140m
308	47	622m
309	67	3164m
310	311	3765m
311	41	3709m
312	394	3515m
313	13	1616m
314	316	502m
315	317	387m
316	390	469m
317	237	353m

318	21	505m
319	22	444m
320	22	455m
321	68	2976m
322	323	2755m
323	59	2722m
324	138	2894m
325	324	2996m
326	412	2910m
327	86	2530m
328	10	2756m
329	331	2492m
330	331	2497m
331	87	2470m
332	333	2322m
333	334	2185m
334	218	1992m
335	79	321m
336	335	437m
337	246	2571m
338	44	5007m
339	63	3683m
340	156	3153m
341	194	1289m
342	53	4835m
343	108	4861m
344	200	777m
345	409	2305m
346	204	2044m
347	45	19m
348	202	50m
349	229	705m
350	5	647m
351	350	651m
352	247	1204m
353	347	41m
354	55	4826m
355	53	4834m
356	335	396m
357	83	4855m
358	144	3815m
359	350	652m
360	242	2930m
361	368	760m
362	368	766m
363	362	783m
364	363	809m
365	364	825m
366	367	803m
367	414	789m
368	349	748m
369	374	714m
370	374	752m
371	378	762m
372	316	707m
373	372	709m
374	349	713m
375	369	747m
376	365	868m
377	12	836m
378	370	755m
379	380	1562m
380	287	1546m
381	396	1487m
382	30	1299m
383	384	690m
384	429	654m
385	47	459m
386	388	1425m
387	131	1410m
388	130	1421m
389	127	1060m
390	23	365m
391	162	1576m
392	340	3364m
393	156	3149m
394	40	3282m
395	342	4842m
396	30	1277m
397	288	1461m
398	204	2063m

```
363       362       783m
364       363       809m
365       364       825m
366       367       803m
367       414       789m
368       349       748m
369       374       714m
370       374       752m
371       378       762m
372       316       707m
373       372       709m
374       349       713m
375       369       747m
376       365       868m
377       12        836m
378       370       755m
379       380       1562m
380       287       1546m
381       396       1487m
382       30        1299m
383       384       690m
384       429       654m
385       47        459m
386       388       1425m
387       131       1410m
388       130       1421m
389       127       1060m
390       23        365m
391       162       1576m
392       340       3364m
393       156       3149m
394       40        3282m
395       342       4842m
396       30        1277m
397       288       1461m
398       204       2063m
399       400       607m
400       104       601m
401       403       619m
402       405       607m
403       402       613m
404       399       613m
405       145       602m
406       103       1298m
407       306       1027m
408       235       1488m
409       346       2080m
410       109       5144m
411       80        1816m
412       70        2823m
413       81        1796m
414       371       775m
415       414       785m
416       377       840m
417       426       2338m
418       398       2302m
419       418       2306m
420       419       2316m
421       347       351m
422       421       360m
423       424       2392m
424       420       2341m
425       345       2307m
426       427       2335m
427       419       2328m
428       426       2337m
429       308       640m

Execution Time: 585.405 ms
Memory Usage: 6063 KB

Time complexity: O(V + E log V), Space complexity: O(V + E)
```

- **Challenges of the World Graph:**
    - Real-world noise: The data included very short road segments (e.g., 1–3 meters) and long edges (e.g., 1000+ meters), testing the algorithm's ability to deal with a wide range of edge weights.
    - Dense local neighborhoods: Some intersections were connected to many roads, causing complex branching.

Conclusion: Dijkstra's algorithm performed robustly and efficiently on the challenging Gibraltar road network. The combination of Java's custom binary heap and careful array handling made it scalable and accurate. This demonstrates how graph algorithms can scale to real-world networks with hundreds of nodes and still produce reliable pathfinding results. In sparse graphs (where $E \approx V$), this is much faster than algorithms with $O(V^2)$ time. That's why Dijkstra + heap works so well on real-world networks like roads, which are usually sparse.

9. Reflection on the Assignment

This assignment offered a highly valuable and practical learning experience. By implementing graph traversal and optimization algorithms such as DFS, BFS, Prim's Minimum Spanning Tree, and Dijkstra's Shortest Path Tree, I gained deeper insights into how real-world problems can be solved using fundamental data structures.

Key learnings:

- **Adjacency List Representation**: Implementing graphs using adjacency lists emphasized how efficient this structure is for sparse graphs like real road networks, minimizing space usage and improving performance when compared to adjacency matrices.

- **Understanding the Difference between MST and SPT**:

    - Prim's MST algorithm minimizes total edge weight while connecting all vertices with no cycles.

    - Dijkstra's algorithm finds the shortest paths from a source to all other vertices, optimizing the individual distances from the start, not the overall sum.

    - Seeing how both result in trees with V-1 edges but with very different construction strategies clarified their conceptual differences.

- **Heap Usage in Graph Algorithms**:

    - I learned how min heaps (priority queues) are used in both Prim and Dijkstra to efficiently extract the next best vertex.

    - Implementing and debugging heap operations such as siftUp and siftDown reinforced my understanding of maintaining heap invariants.

- **Measuring Performance on Real-World Graphs**:

    - The Gibraltar road network allowed me to apply Dijkstra's algorithm on a realistic dataset with 429 nodes and 578 edges.

    - Measuring execution time and memory usage helped connect theoretical complexity with actual runtime behavior.

- **Time and Space Complexity**: I developed a stronger intuition for why Dijkstra's algorithm is O((V + E) log V) and why Prim's is O(E log V), particularly in the context of using heaps and adjacency lists. Dijkstra has to look at all edges, because it builds shortest paths from a single source — it doesn't stop early like Prim might because Prim's algorithm doesn't care about reaching every vertex from one source, just connecting the entire graph minimally.

Practical benefits:

- Working with real input data (e.g., roads with distances in meters) added realism and relevance to the algorithms
- The debugging output (showing changes in dist[], hPos[], and parent[] after each step) helped me deeply trace and validate each stage of the algorithms' execution.
- Writing a clear report with annotated logs and visualizations clarified the storytelling side of algorithm analysis—how to explain what's happening to others.

Final thoughts: Overall, the assignment deepened both my theoretical knowledge and practical coding skills in graph algorithms. It was especially rewarding to go from textbook examples to applying algorithms on a real-world road network. The process gave me confidence to approach more advanced algorithmic challenges in future projects.