Algorithms 2 Assignment report:

Student name: Justin McGarr

Student ID: C23421344

Collaborated with: **Susanna Perkins (C23389191)**

Introduction/Explanation:

The assigned task is based on manipulating and analyzing graphs in Java. We are given a graph and heap class and must implement various functions to test the graph in different ways. The graph mut be represented using an adjacency list data structure and once that is done, as well as completing the code for the Heap, we must implement a depth first search using recursion and a breadth first search using a queue, in order to search the graph fully. As well as this we must implement Prims Minimum Spanning Tree Algorithm and Dijkstras Shortest Path Tree Algorithm. All the above are to be run on an example graph which the user can search for using I/O and Dijkstras must also be ran separately on a dense , real-world graph example like a road network.

Heap:

The first part of the task was to implement the siftUp and siftDown functions in the given heap. I started with siftUp, which is used to traverse the graph from the bottom to its parent nodes.

SiftUp:

```java
public void siftUp( int k)
{
    int v = a[k];

    // code yourself
    // must use hPos[] and dist[] arrays

    //priority is the distance of our initial node
    int priority = dist[v];

    //first index of array is set to 0 as a dummy value which protects from edge cases
    a[0] = 0;
    //by setting the distance to maximum we ensure the dummy value is never compared with a node
    dist[0] = Integer.MAX_VALUE;

    //while the distance of our current node is less than the distance of its parent, swap
    while (k > 1 && priority < dist[a[k/2]]) {
        //perform swap of the nodes
        int parent = a[k / 2];
        a[k] = parent;
        hPos[parent] = k;
        //update the new position to be the parent
        k = k / 2;
    }
    //place the original node back in its original place
    a[k] = v;
    //add the node to hPos to keep track of the positions of nodes in our tree
    hPos[v] = k;
}
```

In my above implementation, we are moving a node upward if its priority (shortest distance from the source node) is less than that of its parent. v represents the current node we are moving (or not) from, and priority is set to dist[v]. The while checks if its weight is less than the parent (parent of a child is always a[k/2]). In this while i also check if k > 1, this is because I was experiencing problems with my Prims later where the MST was being incorrectly calculated due to the dummy value being included and so checking k > 1 made sure only values inside excluding the dummy value were included. As long as the child is smaller than the parent, a basic swap is performed and the new current node becomes k/2, or the parent node. Outside the while we move the original node back to where it was after the swap and our hPos array which tracks the position of nodes, adds the first node to its array, and we continue.

SiftDown:

```java
public void siftDown( int k)
{
    int v, j;

    v = a[k];

    // code yourself
    // must use hPos[] and dist[] arrays
    //setting the priority as the distance of the current node
    int priority = dist[v];

    //while loop continues until we reach a leaf node (end of the tree)
    while (2*k <=N)
    {
        //initialising j to the left child of the current node
        j = 2* k;
        //condition checks that children are in the heap bounds, and compares the priority of the left and right child
        if(j< N && dist[a[j]] > dist[a[j+1]])
        {
            //if the distance of the left child is bigger than right child (right child has higher priority) we now point at the right ch
            ++j;
        }
        //if the distance of the current node is smaller than the child, then we dont need to swap and break out of the loop
        if( priority <= dist[a[j]])
        {
            break;
        }
        //swapping the parent node and the child node
        a[k] = a[j];
        hPos[j] = k;
        k = j;
    }
    //returning the original node to its position
    a[k] = v;
    //adding the node to the hPos to reflect the position of the nodes in the tree
    hPos[v] = k;
}
```

The general logic for SiftDown was simlar to siftUp, except now we are working with comparing a parent with its children. Therefore, the only difference was the while and if clauses. While (2*k <=N) ensures we continue to the very end of the tree, where its saying while the left child is less than or equal the size of the Heap. J is initialized to the left child of the current node and is used to compare priority (smallest possible distance). We check if j is in the Heap, and if its priority is less than the right child, then j is incremented, and we move to the right child. An if (priority <=dist[a[j]]) is also there to ensure that we dont bother sifting if the parent is highest priority already. Once on the right child, a basic swap is performed and our current node and hPos[] array is updated same as siftUp.

Adjacency list representation:

```
    // write code to put edge into adjacency list
    //making a new node that represents destination being edge V
    t = new Node();
    //vertex is v
    t.vert = v;
    t.wgt = wgt;
    //v is pointing to the next node in the adjacency list (current head)
    t.next = adj[u];
    //link new node to existing list (updating head)
    adj[u] = t;

    //the same as previously but for destination
    t = new Node();
    t.vert = u;
    t.wgt = wgt;
    t.next = adj[v];
    adj[v] = t;

}
```

I had to write some simple code to put each edge of the graph into adjacency list format. We create two new Nodes to represent the edges. The first line represents the edge FROM u TO v and the second node represents the edge FROM v TO u. t.next = { } ensures that each time an edge is added that we point at the current head, so it is added at the front of the list.

Shortest Path Tree/Minimum Spanning Tree:

Prims MST Algorithm:

```java
public void MST_Prim(int s)
{
    int v, u;
    int wgt, wgt_sum = 0;
    int[]  dist, parent, hPos;
    Node t;

    //code here
    //initialising all the integer arrays
    dist = new int[V + 1];
    hPos = new int[V +1];
    parent = new int[V + 1];

    for(v =1; v <= V; v++)
    {
        //dist set to infinity from source to ensure dummy value never compares with node
        dist[v] = Integer.MAX_VALUE;
        parent[v] = 0;
        hPos[v] = 0;
    }
    //distance from the source to itself is 0
    dist[s] = 0;

    //create new min heap
    Heap h =  new Heap(V, dist, hPos);

    //insert the source node into our min-heap
    h.insert(s);

    //while the heap is not empty
    while (!h.isEmpty())
    {
        //extract the weight of the minimum distance
        v = h.remove();

        //incrementing the weight sum to = weight + every new distance that is added to the heap
        wgt_sum += dist[v];

        //this is used to flag that dist of v is added to the MST
        dist[v] = -dist[v];

        //iterating or checking every neighbouring node u of v
        for(t = adj[v]; t!= z ; t = t.next)
        {
            //u is the vertex between u->v
            u = t.vert;

            //if the weight of the neighbouring node is less than the distance of dist[u] aka smallest edge linking u
            if(t.wgt < dist[u])
            {
                //if true, update the new dist
                dist[u] = t.wgt;
                //assign parent of u now to v (the edge we're connecting current node via)
                parent[u] = v;
                //if u is not in the heap
                if(hPos[u] == 0){
                    //insert u
                    h.insert(u);
                }
                else{
                    //if it is already, the position is updated
                    h.siftUp(hPos[u]);
                }
            }
        }
```

Next i had to implement Prims' Minimum Spanning Tree algorithm to find the minimum spanning tree. Prims finds a subset of the edges that forms a tree that includes every vertex, with the smallest total weight possible. It is a greedy algorithm meaning it will always choose the shortest edge it can at each possible turn. In the code it uses a min-priority heap and traverses the adjacency list to find the smallest possible distance and updates the new smallest distance (or highest priority) using our dist[] array. We make use of the hPos array and our previously implemented siftUp also updating the position of each node. The wgt_sum variable keeps track of the total value of the MST.

Dijkstras SPT Algorithm:

```
class Graph {
    public void SPT_Dijkstra(int s)
    {
        //initialise variables
        int v, u, d;
        int wgt, wgt_sum = 0;
        int[]  dist, parent, hPos;
        Node t;

        //initialise new integer arrays
        dist = new int[V + 1];
        hPos = new int[V +1];
        parent = new int[V + 1];

        for(v =1; v <= V; v++)
        {
            //dist set to infinity to avoid nodes being compared with the dummy value
            dist[v] = Integer.MAX_VALUE;
            parent[v] = 0;
            hPos[v] = 0;
        }
        //initialising a new priority queue
        Heap pq = new Heap(V, dist, hPos);

        //adding the source node to the priority queue(the node we are starting from)
        pq.insert(s);

        //distance from source node to source is 0
        dist[s] = 0;

        //while the priority queue isnt empty
        while(!pq.isEmpty())
        {
            //v is set to the minimum distance which is extracted using remove
            v = pq.remove();

            //weight sum incremented (adds each new weight every time)
            wgt_sum += dist[v];
            //for examining each neighbour of v
            for(t = adj[v]; t!=z; t= t.next)
            {
                //d is the weight of the neighbouring node
                d = t.wgt;
                //u is the vertex of the neighbouring node
                u = t.vert;

                //if the total distance from the source -> v -> u (current shortest distance + weight of v -> u) is less than the distance from source straight to u
                if(dist[v] + d < dist[u])
                {
                    //update the current shortest distance to now be distance of source -> v -> u
                    dist[u] = dist[v] + d;
                    //if u is not in the heap
                    if(hPos[u] == 0)
                    {
                        //insert u
                        pq.insert(u);
                    }

                    else
                    {
                        //update the position of u in the heap
                        pq.siftUp(hPos[u]);
                    }
                    //the parent of u is now v
                    parent[u] = v;

                }
            }
        }
```

Next i had to implement Dijkstra's Shortest Path Tree algorithm. Dijkstras finds the shortest path from a starting node to all other nodes in the graph. It uses a priority queue min-heap and is very similar to prims. However instead of comparing the distance of each edge in Prims, here we are trying to find the shortest distance from a source node and so we check if the distance from the source through nodes v and u is less than the distance straight from the source to u. Then after this check it is the same as prims where we update the priority queue and hPos array as well as the parent of u.

DepthFirst/BreadthFirst Search:

```java
//method to perform a depth first search on graph
public void DF(int s)
{
    System.out.println("\n\nDepth First Search vertex visits");
    int v;
    for(v=1; v <= V; ++v)
    {
        colour[v] = C.White;
        parent[v] = 0;
    }
    time = 0;
    dfVisit(s);
    for(v = 1; v<=V; ++v)
    {
        if(colour[v] == C.White)
        {
            //if the vertex is white, we call the recursive method to visit it
            dfVisit(v);
        }
    }
}

//method that depth is recursively called by DF to depth first search
public void dfVisit(int v)
{

    Node t;

    ++time;
    d[v] = time;
    colour[v] = C.Grey;

    //source node
    if(parent[v] ==0)
    {
        System.out.println("\nvisited vertex " + toChar(v));
    }
    //following nodes
    else
    {
        //print statement to display the vertex being visited and the edge it was reached by
        System.out.println("visited vertex " + toChar(v) + " along edge " + toChar(parent[v]) + "--" + toChar(v));
    }

    //visiting every neighbour of v
    for(t = adj[v]; t != z; t = t.next)
    {
        //u is the vertex between u->v
        int u = t.vert;

        //if the vertext hasnt been visited, recursively call dfVisit, following a whole path until reaching the end (deep search)
        if(colour[u] == C.White)
        {
            parent[u] = v;
            dfVisit(u);
        }
    }
    colour[v] = C.Black;
    time++;
    f[v] = time;
}
```

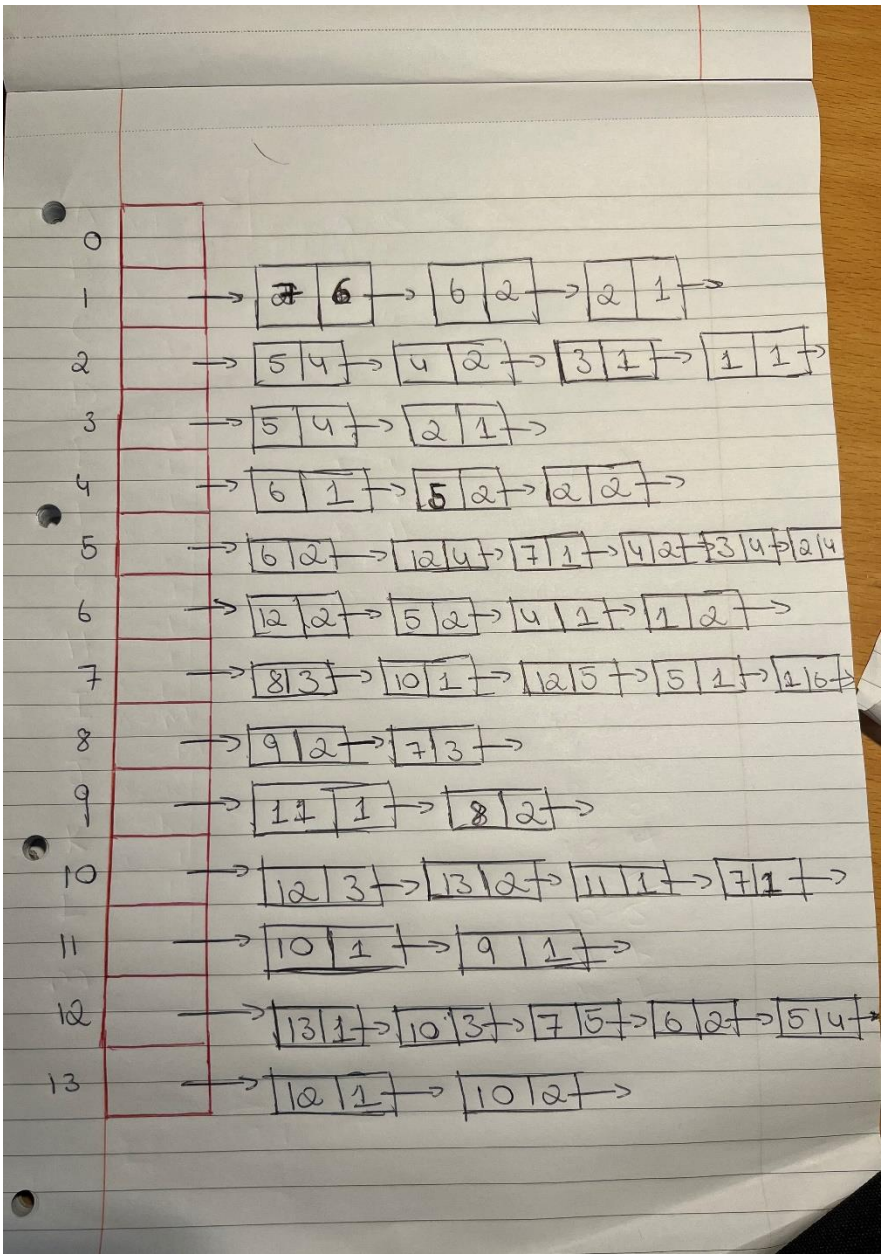
Finally, i had to implement searches on the given graph. Depth first search is a method of exploring graphs where you start at a node and go as deep as possible down a path until you reach a dead end. I had to implement a recursive DFS which can be seen above. The DF function is used to track what nodes have been visited and then recursively search. The actual DepthFirst traverses the adjacency list and updates the vertex each time by recursively calling itself each time it hits a node that has not been visited yet. This implementation uses Cormen's approach where colors are used to mark nodes as either not visited (White), currently being visited (Grey) and visited (Black) so we can track the nodes dynamically.

BreadthFirst

```
public void breadthFirst(int s)
{
    System.out.println("\n\nBreadth First Search vertex visits");
    Node t;

    //initialising an array to keep track of vertexes that have been visited
    visited = new int[V +1];

    //creating a queue which will hold what vertices will be visited when
    Queue<Integer> q = new LinkedList<>();

    //id flags visitors
    id = 0;
    for(int v =1; v <= V; v++)
    {
        //no vertex visited yet
        visited[v] = 0;
    }
    //starting from the source node or adding source node to the queue
    q.add(s);

    //whilst the queue is not empty
    while(!q.isEmpty())
    {
        //remove the next vertex to be processed (source)
        int v = q.remove();
        //if the vertex just extracted has not been visited yet
        if(visited[v] ==0)
        {
            //use the id to mark it as visited
            visited[v] = ++id;

            //print statement to display vertexes being visited
            System.out.println("visited vertex " + toChar(v)  +" along edge " + v + "---" + toChar(v));

            //visiting each neighbour of v
            for(t = adj[v]; t!=z; t = t.next)
            {
                //u is the vertex between u->v
                int u = t.vert;
                //if the neighbour of v has not been visited
                if(visited[u] == 0)
                {
                    //add it to the queue to be processedZ\ aka visited
                    q.add(u);
                }
```

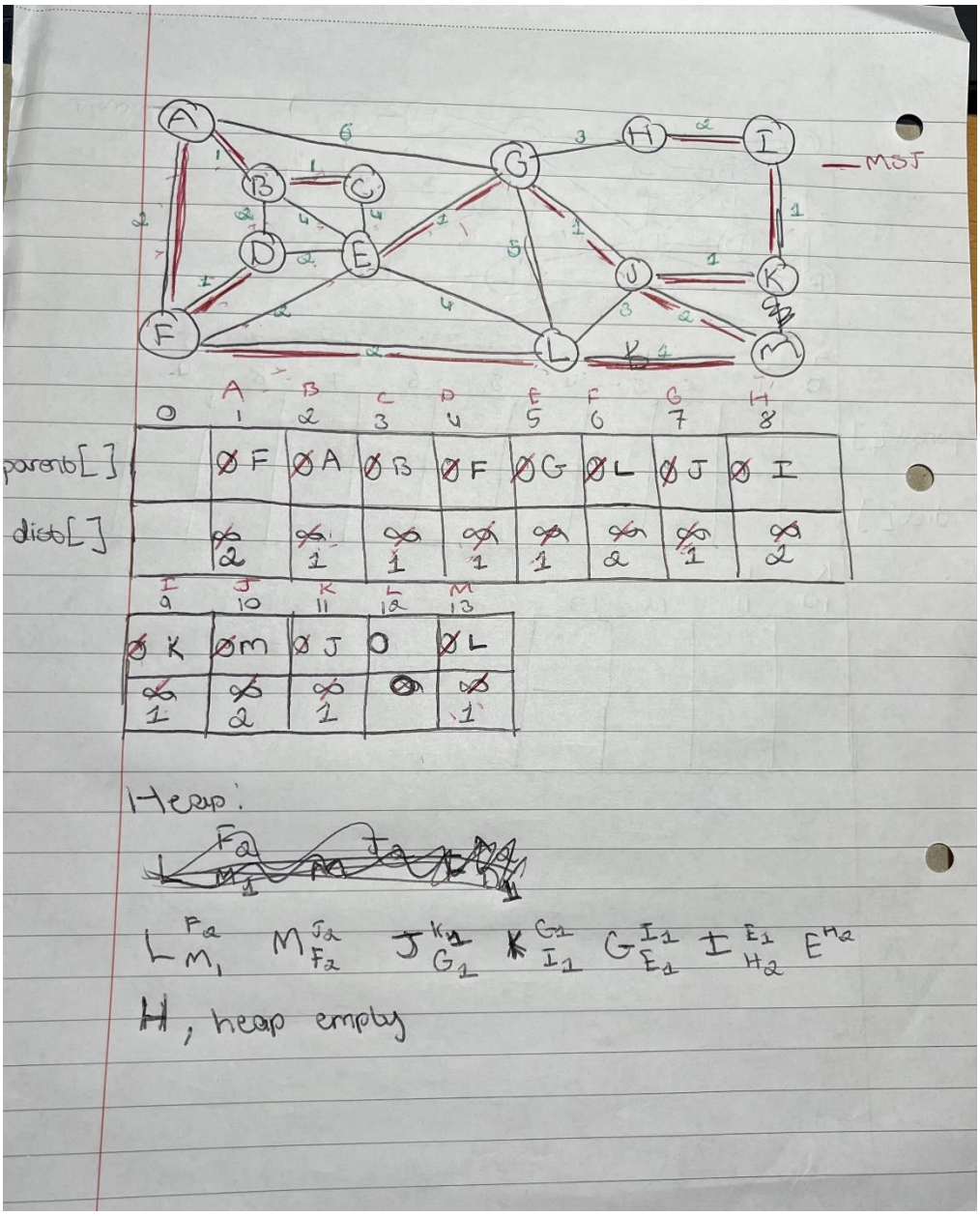
The final step was implementing a BreadthFirst search. BreadthFirst yields the same results as a depth first search but gets there differently. Breadth First Search explores graphs level by level. Starting at a source node and visiting all its neighbours, then their neighbours until all nodes have been visited. A priority queue is used in its implementation. The source node is added to the queue and then we traverse the list, adding each node that has not been visited to the queue and at the start of each loop, removing the tail of the queue until each node had been added and then the queue is emptied, meaning every node has been visited. The same kind of approach adopted by Cormen was implemented as shown previously where colors were used to track the statuses of each node.

2. **Adjacency list diagram:**

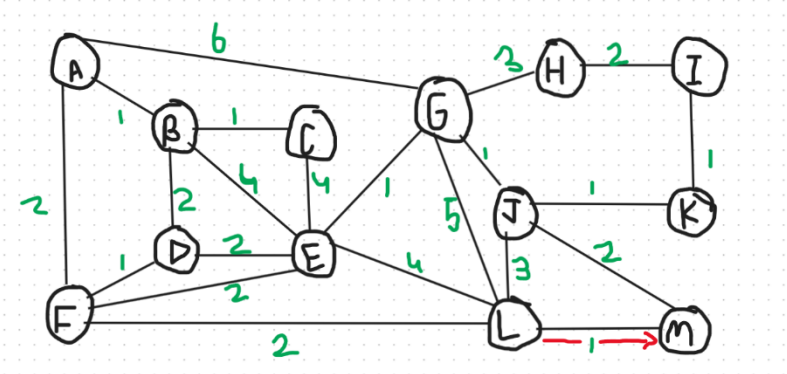


The diagram represents the sample diagram in an adjacency list format

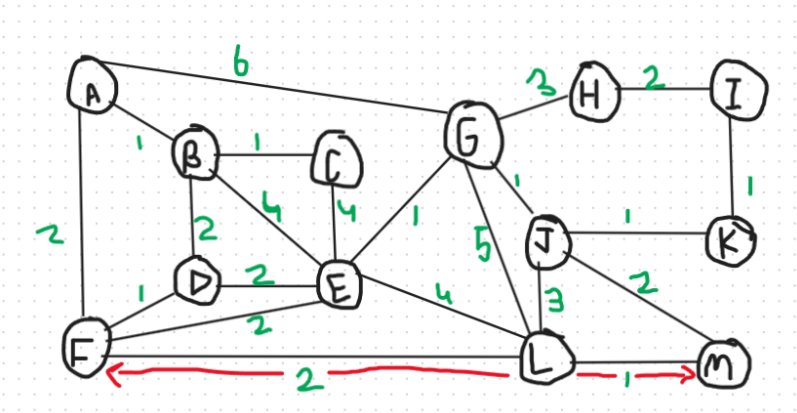
3. MST Construction (Prims):



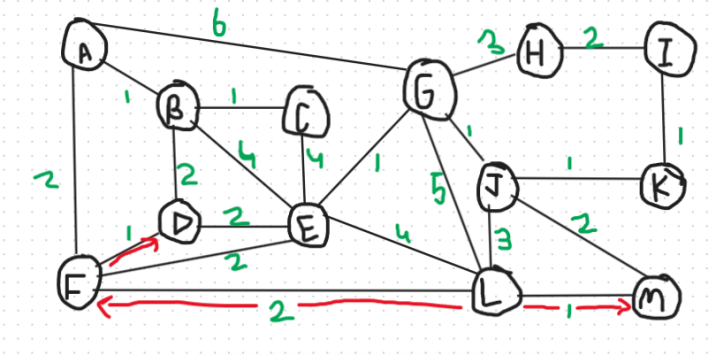
Move 1:



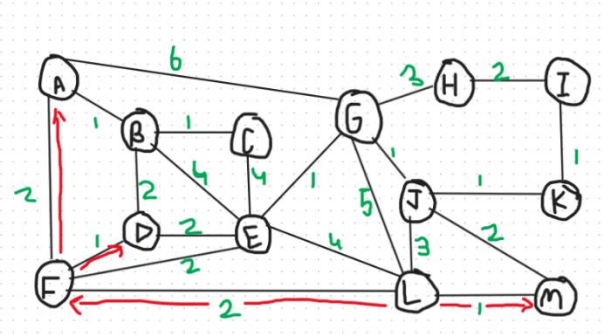
Move 2:



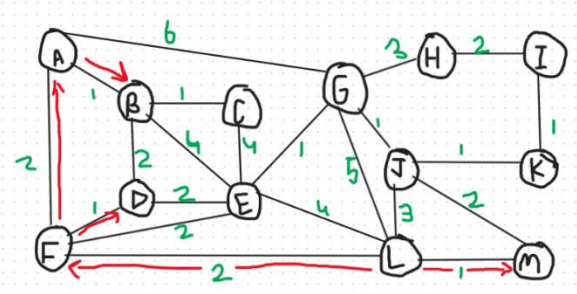
Move 3:



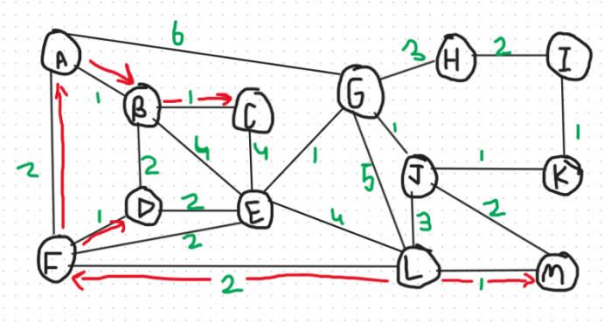
Move 4:



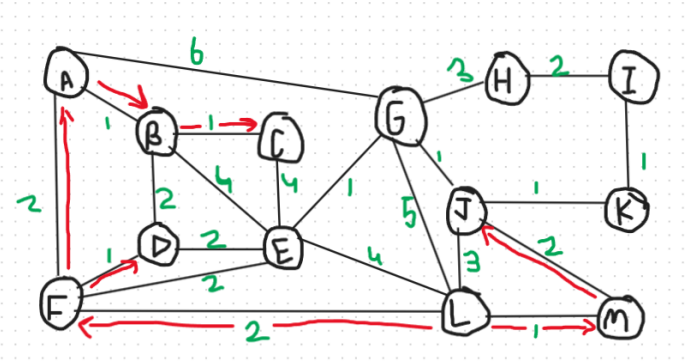
Move 5:



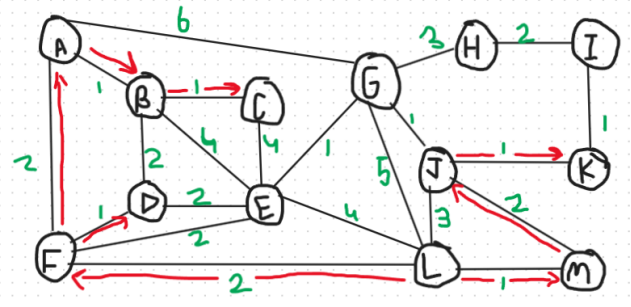
Move 6:



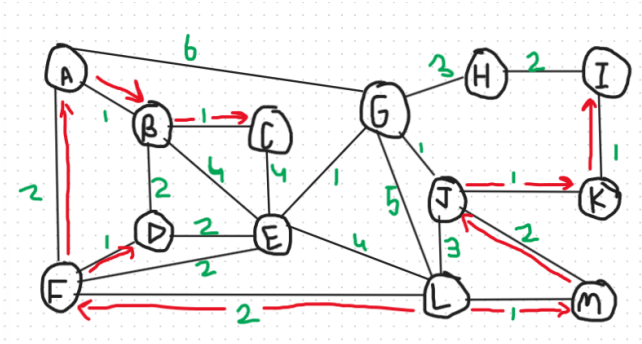
Move 7:



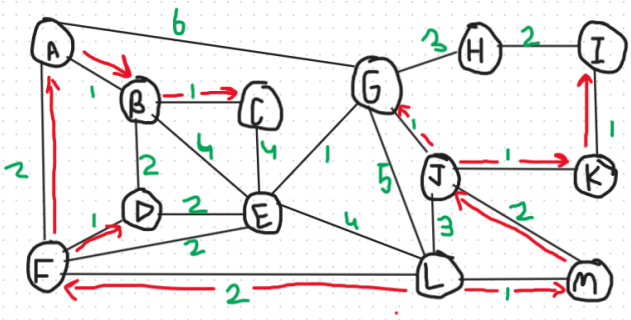
Move 8:



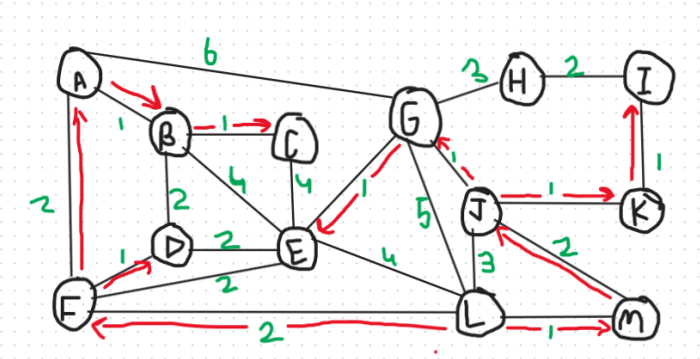
Move 9:



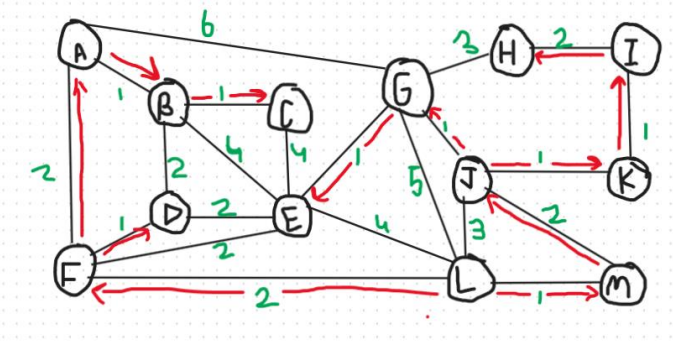
Move 10:



Move 11:

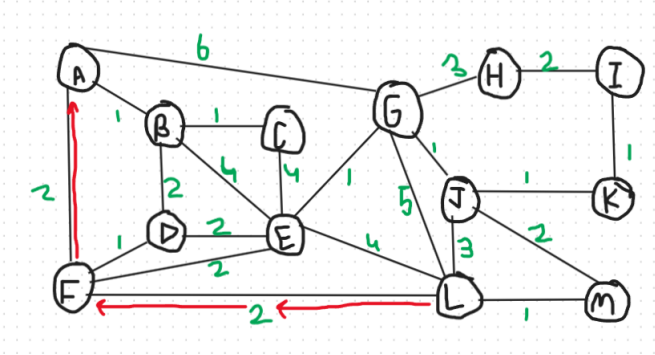


Move 12 (Final move):

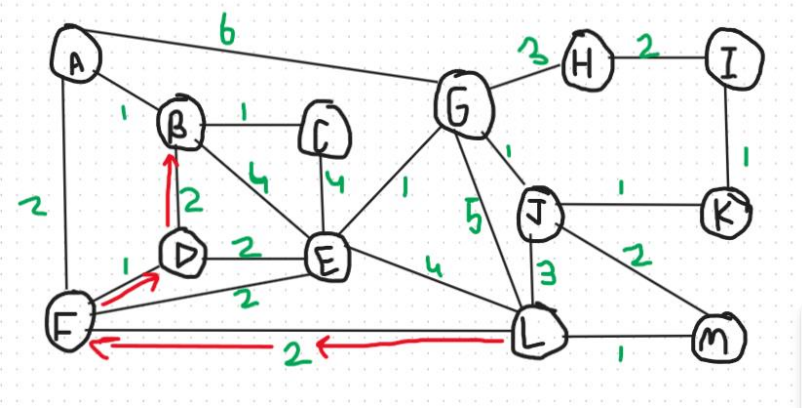


4. SPT Construction (Dijkstras):

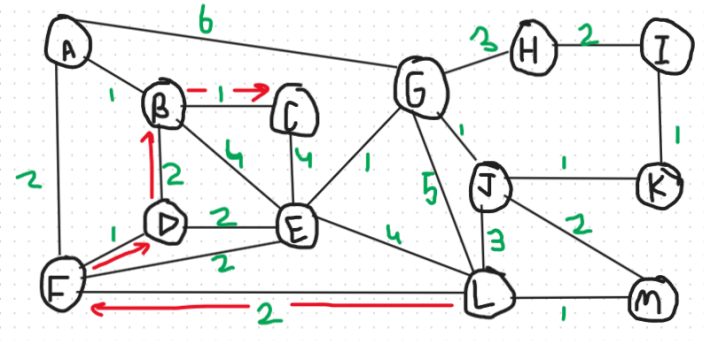
L->A:



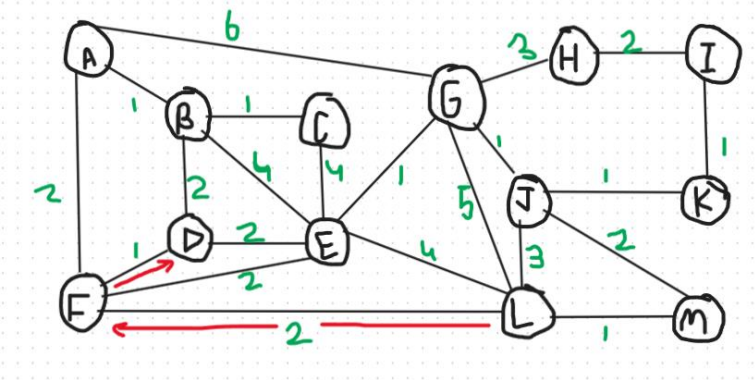
L->B:



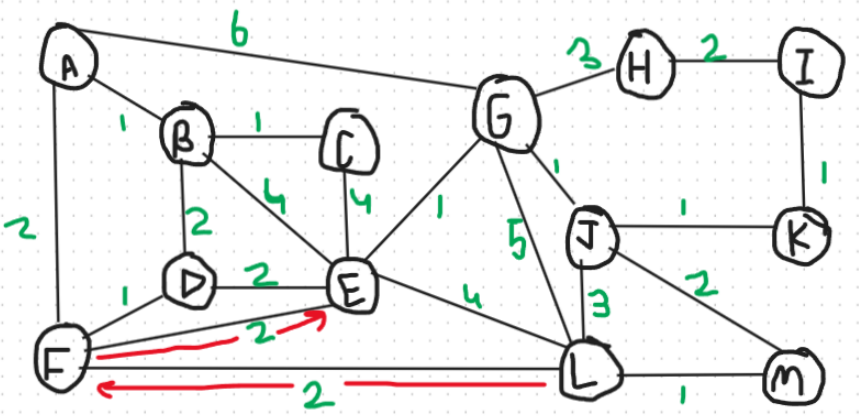
L->C:



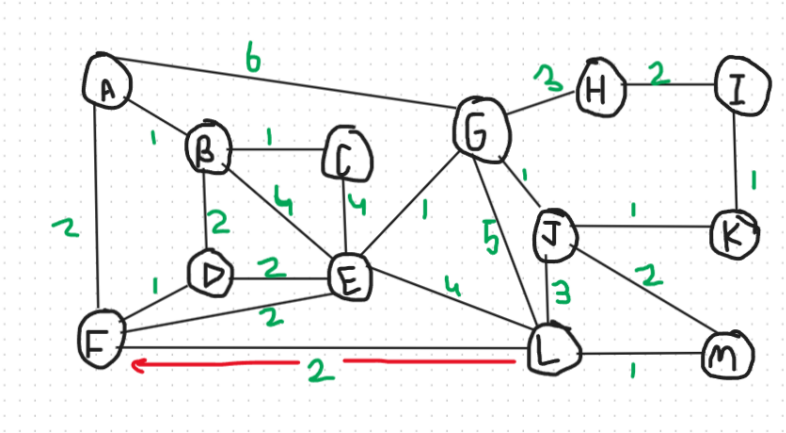
L->D:


A
B
C
D
E
F
G
H
I
J
K
L
M
6
3
2
1
1
1
2
4
4
1
5
1
1
2
2
1
2
4
3
2
2
1

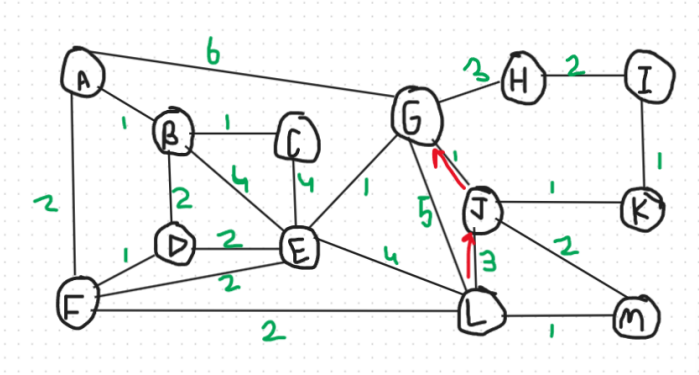
L->E:


A
B
C
D
E
F
G
H
I
J
K
L
M
6
3
2
1
1
1
2
4
4
1
5
1
1
2
2
1
2
4
3
2
2
1

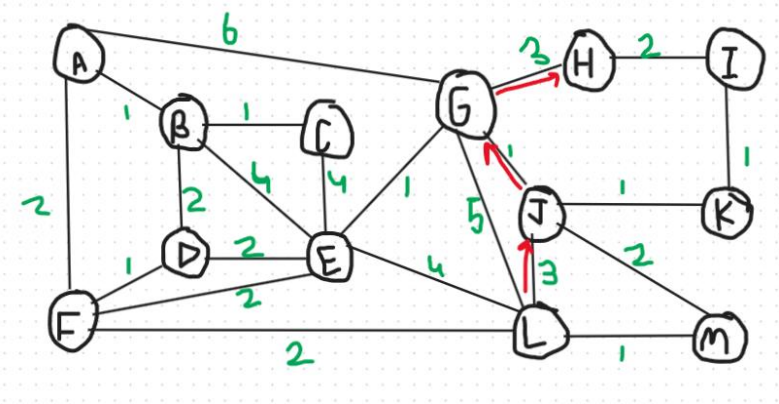
L->F:


A
B
C
D
E
F
G
H
I
J
K
L
M
6
3
2
1
1
1
2
4
4
1
5
1
1
2
2
1
2
4
3
2
2
1

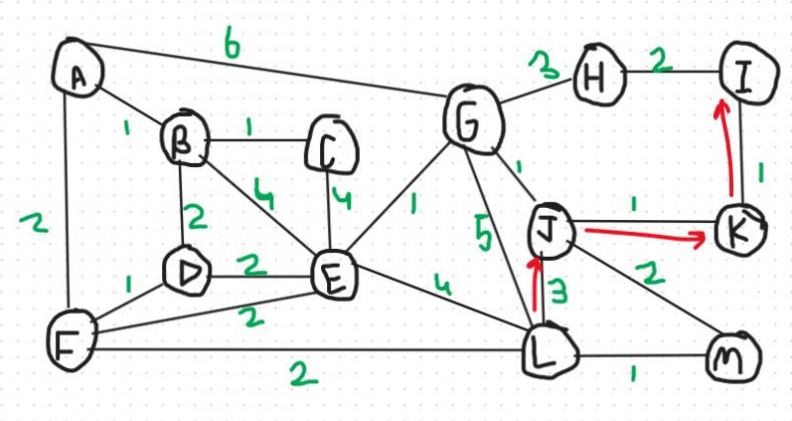
L->G:



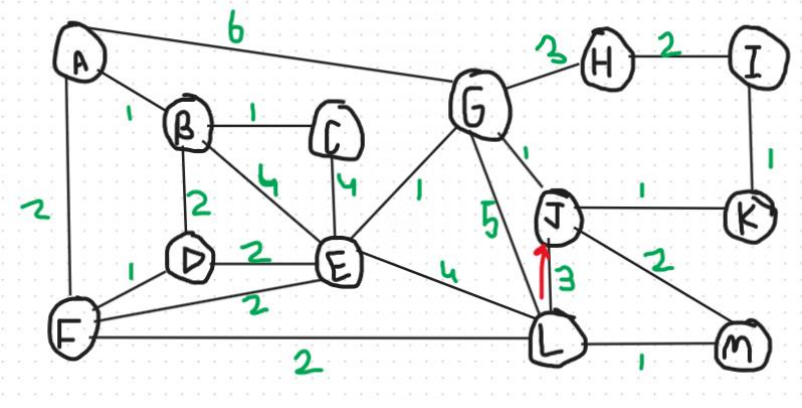
L->H:



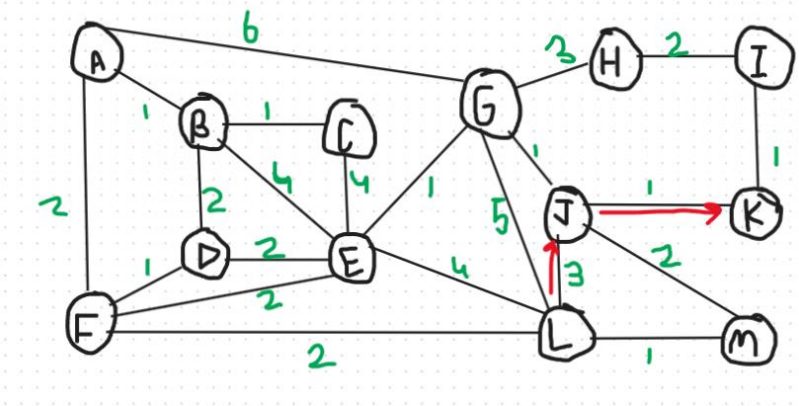
L->I:



L-J:

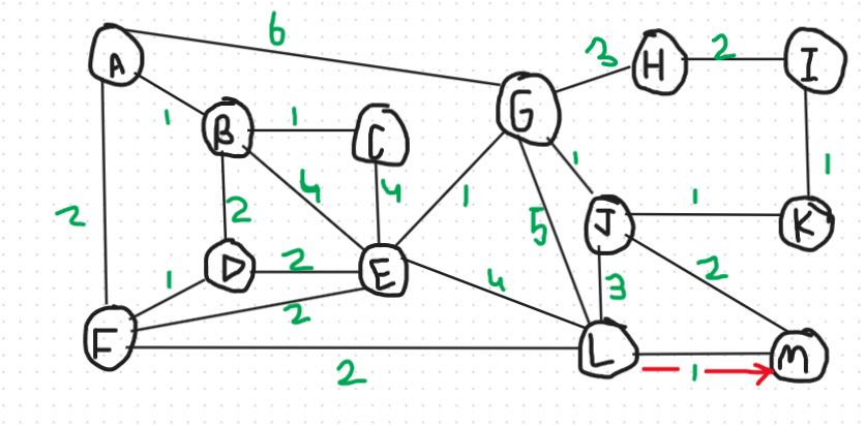


L->K:

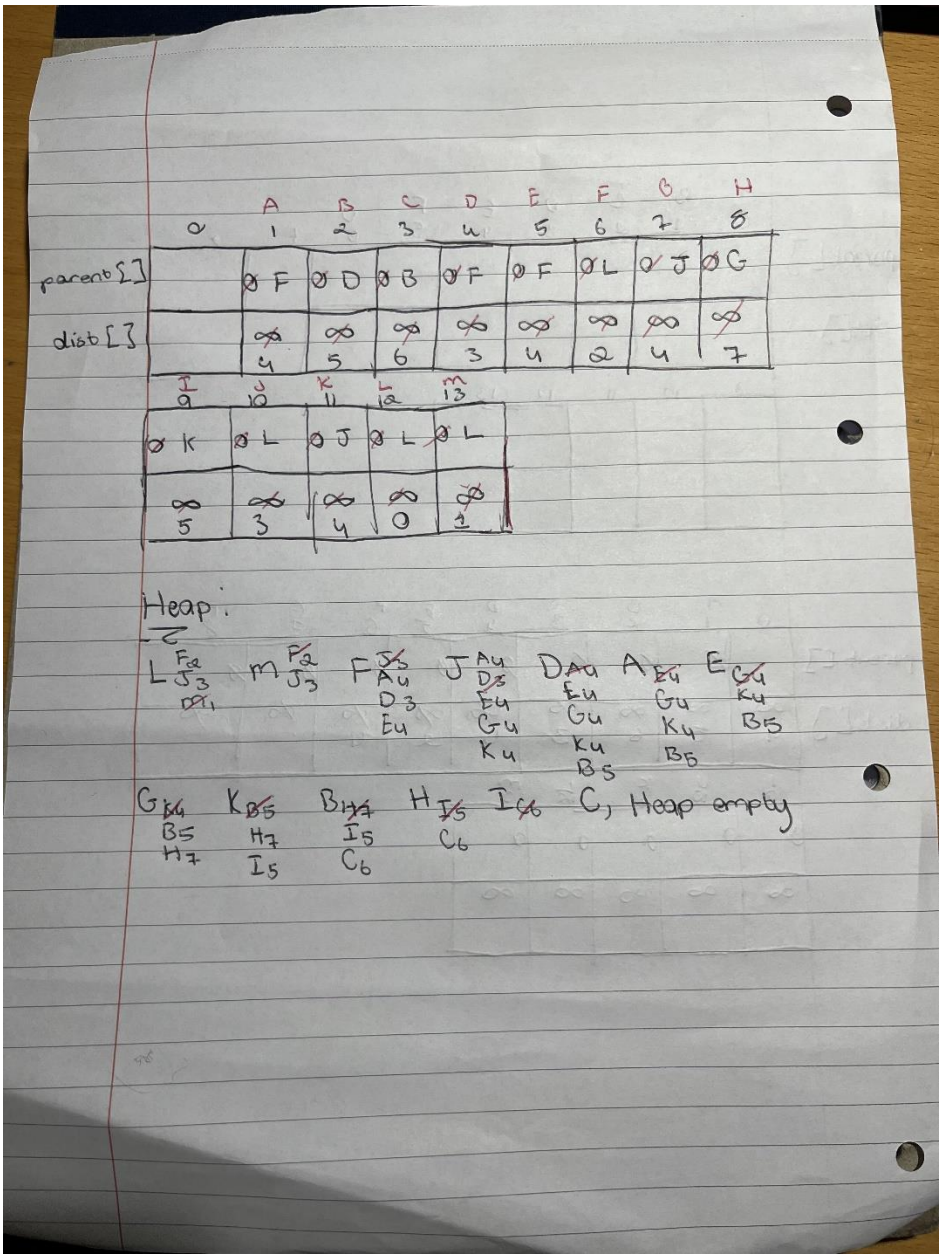


L->L = 0:

L->M:



heap, parent, dist arrays:



	0	A 1	B 2	C 3	D 4	E 5	F 6	G 7	H 8
parent[]		∅ F	∅ D	∅ B	∅ F	∅ F	∅ L	∅ J	∅ G
dist[]		∞ 4	∞ 5	∞ 6	∞ 3	∞ 4	∞ 2	∞ 4	∞ 7

I 9	J 10	K 11	L 12	M 13
∅ K	∅ L	∅ J	∅ L	∅ L
∞ 5	∞ 3	∞ 4	∞ 0	∞ 1

Heap:

L F2 J3 D1

M F2 J3

F ~~J5~~ A4 D3 E4

J A4 ~~D3~~ E4 G4 K4

D ~~A4~~ E4 G4 K4 B5

A ~~E4~~ G4 K4 B5

E ~~G4~~ K4 B5

G ~~K4~~ B5 H7

K ~~B5~~ H7 I5

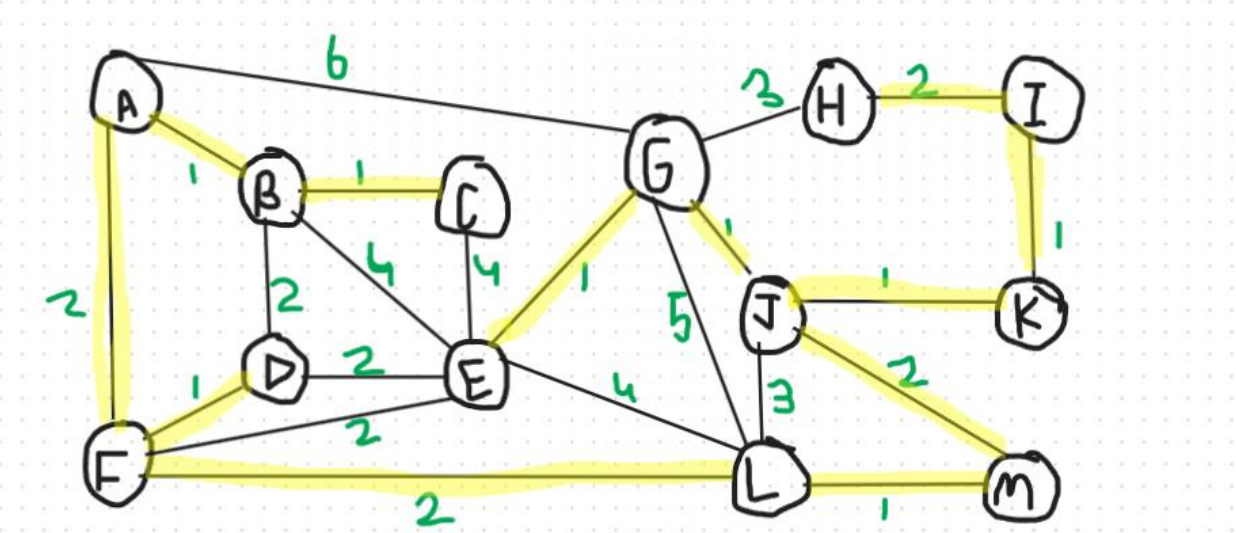
B ~~H7~~ I5 C6

H ~~I5~~ C6

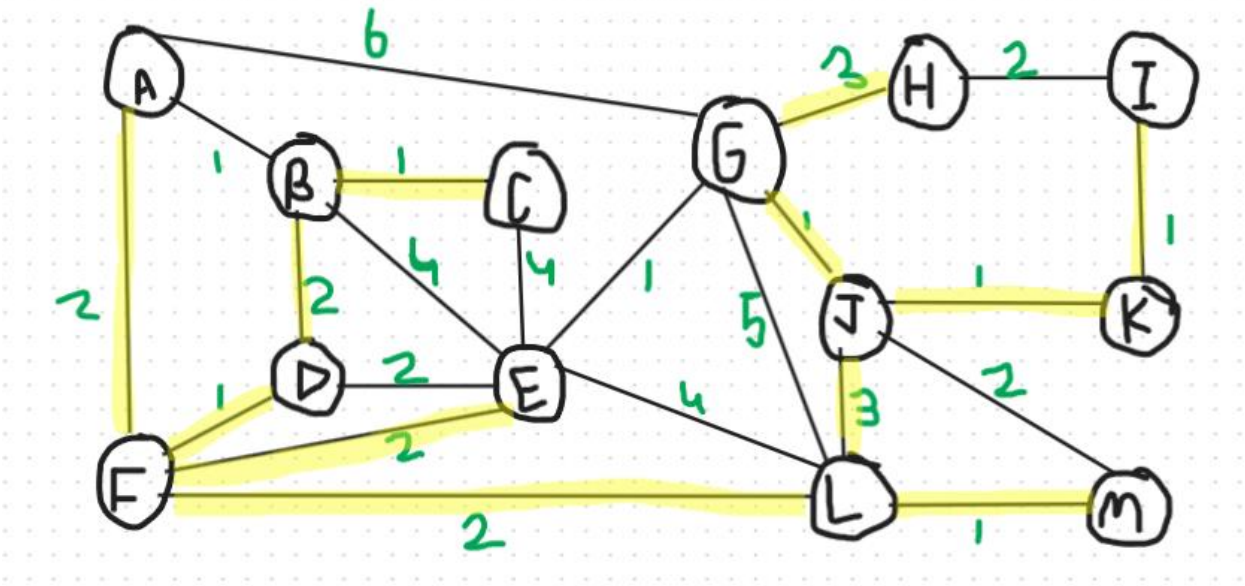
I ~~C6~~

C, Heap empty

5. MST Superimposed on Graph:



6. SPT Superimposed on Graph:



7. Program Output:

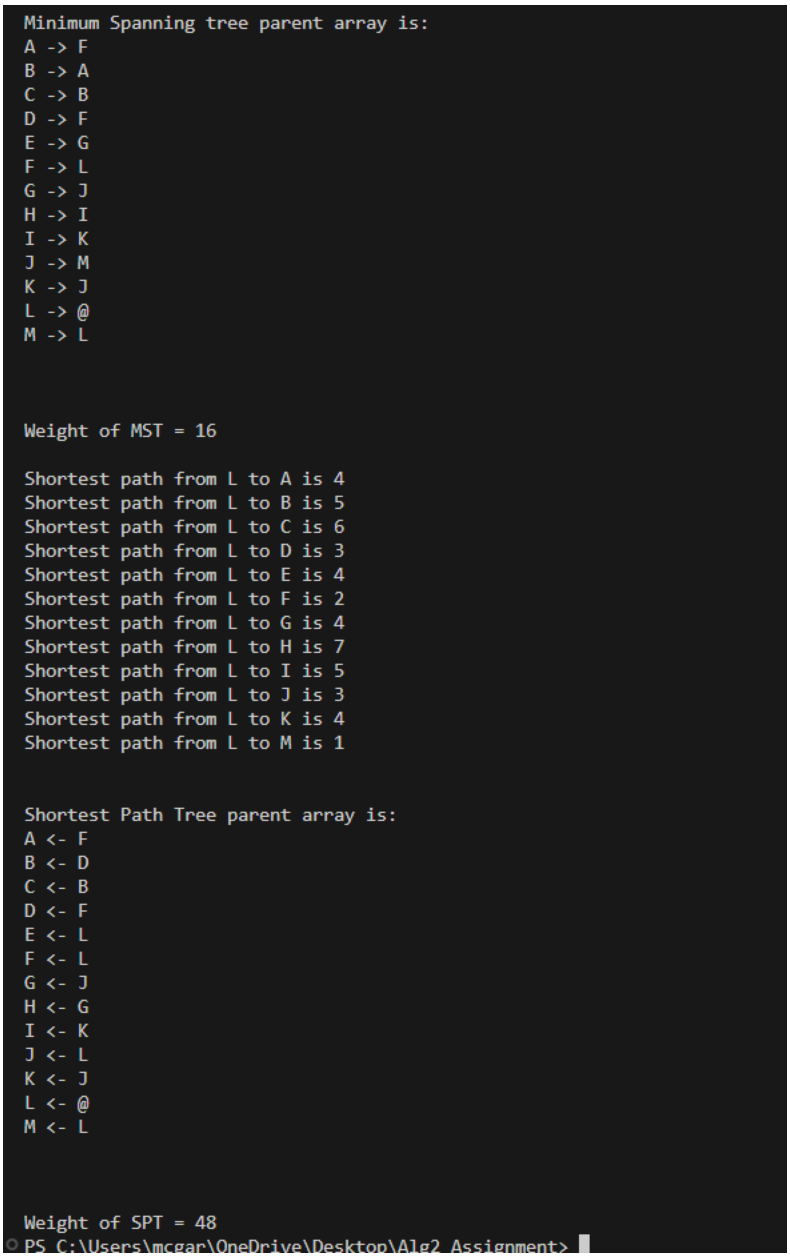
User input, reading all the edges from the file:

```
Enter the name of the graph file:
wGraph1.txt
Enter the starting vertex:
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

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| ->
```

Depth First/Breadth First searches on graph starting from vertex L:

```
Depth First Search vertex visits

visited vertex L
visited vertex M along edge L--M
visited vertex J along edge M--J
visited vertex K along edge J--K
visited vertex I along edge K--I
visited vertex H along edge I--H
visited vertex G along edge H--G
visited vertex E along edge G--E
visited vertex F along edge E--F
visited vertex D along edge F--D
visited vertex B along edge D--B
visited vertex C along edge B--C
visited vertex A along edge B--A


Breadth First Search vertex visits

visited vertex M along edge L--1--M
visited vertex J along edge L--3--J
visited vertex G along edge L--5--G
visited vertex F along edge L--2--F
visited vertex E along edge L--4--E
visited vertex K along edge J--1--K
visited vertex H along edge G--3--H
visited vertex A along edge G--6--A
visited vertex D along edge F--1--D
visited vertex C along edge E--4--C
visited vertex B along edge E--4--B
visited vertex I along edge K--1--I
```

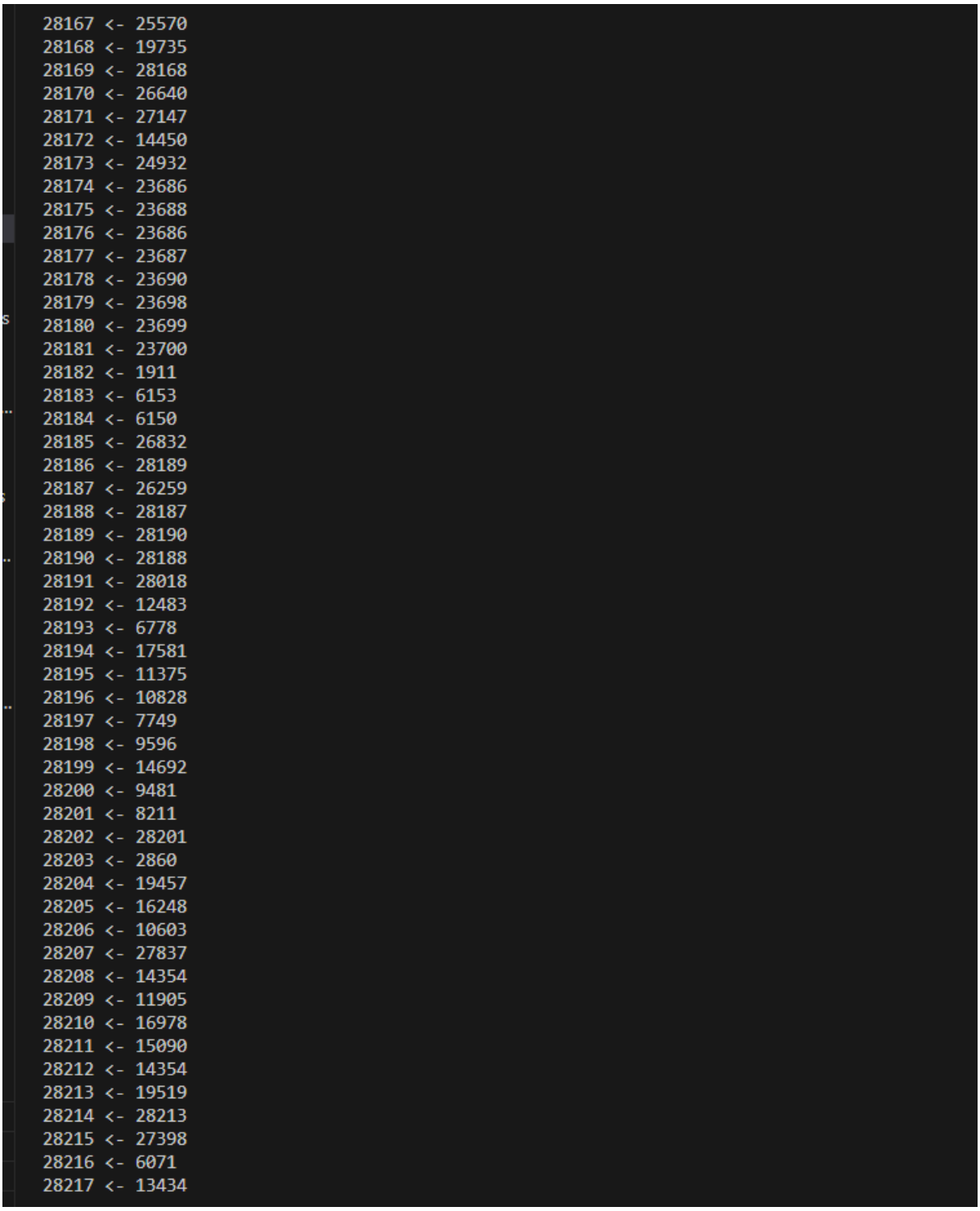
Prims MST Algorithm and Dijkstras SPT Algorithm:



Dijkstras SPT Algorithm running on a large real world road network graph:

SPT:

(most of array doesnt fit in screenshot):



8. Description of real world graph and discussion on Dijkstra performance:

For this part of the assignment, I had to separately run my implementation of Dijkstra's algorithm on a much larger scale graph that represents a real-world road network. I found a graph online that represents the road network of Berlin, which had 28,217 vertices and 42,296 edges. The implementation was easy as i had already done it in GraphLists so i just had to copy that into a different file along with the necessary classes, no changes had to be made. Dijkstras performed very efficiently on the road network, with it successfully running in a couple milliseconds. Dijkstra's performed so well because it perfectly suited for sparse graphs such as the one i used in this example. The time complexity for Dijkstra's on sparse graphs = O(Vlog2V). Since our graphs are represented using adjacency lists and Dijkstra's only visits the relevant neighbours, this is optimal as it doesn't need to run any unnecessary scans of rows like it would have to do with a matrix, therefore Dijkstra's performs optimally without problems with the real-world graph.

9. Discussion on what i learned & found useful:

In general, the assignment gave me a much deeper understanding of working with graphs in code than i ever had before. While I had an elementary understanding of working with graphs in code, this assignment helped me to understand the complexities of graph traversal. Working with the adjacency list implementation was also helpful because I had more of an understanding of the matrix before. The implementation of both Prims and Dijkstras required me to have a more complete, full understanding of how they actually work, not just how to code them. The same goes for Depth First Search and Breadth First Search. At first I implemented Cormens version of BFS/DFS using the visited array and ids to flag visitation, but then ended up changing it to use his implementation which made use of an enumerated colour array. The colour array did prove more difficult but also helped me understand how the actual logic behind BFS/DFS is because the way this implementation works is easier to understand on a more human understanding.

In terms of code, I also improved my ability to manipulate and work with linked lists, especially the adjacency list implementation required me to understand how working with linked lists when traversing graphs.

The real world graph/road network helped me understand how SPT can be useful in real life implementation, I was able to see the actual running time of a calculation of the shortest path tree of the Berlin road network and it opened my eyes to how these can be implemented in real situation.

At the end of this assignment, my java coding ability significantly improved, with my traversal and manipulation of linked lists being far greater than when I began, and my

understanding and ability to use different algorithms that I didn't understand so well before such as the path tree algorithms is far more in depth, thus proving that this assignment was very important to my level of coding.