DF Traversal & Prim’s MST Algorithms Assignment

Cian Doyle

C18430304

DT-228/2

Algorithms and Data Structures

May 8, 2020

Introduction

The purpose of this assignment is to explore algorithms and data structures related to graphs and the traversal of the data in graphs. I will be discussing the concept of a graph, in particular, a weighted, bi-directional (also known as undirected) graph and how we can represent graphs in memory. I will furthermore be exploring the idea of a minimum spanning tree, which can be found with several algorithms. I will be using Prim's algorithm for this assignment. I will be discussing the history of this algorithm further on in this report. I will be implementing these algorithms in Java, as per the assignment's requirements.

I would first like to clarify what is meant by a graph. A graph can be defined as a set of vertices, or nodes, connected via edges or arcs. A graph can be weighted, meaning each edge between vertices has a number associated with it. Graphs can also be directed, meaning an edge points from one vertex to another.

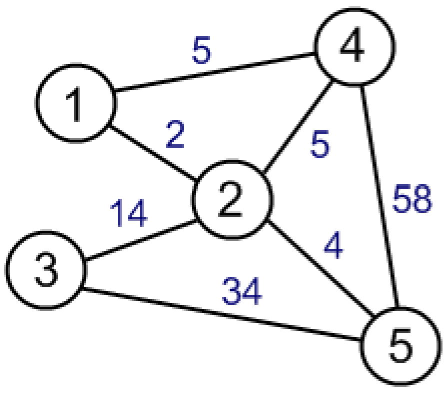


Fig 1.1: Example of a weighted undirected graph.

Shown above is the type of graph that will be used for part (i) and (iii) of the assignment. A disconnected graph, essentially a set of connected graphs, will be used for part (ii). To get started, I created my own graph, as well as the graph provided in the assignment to test code on.

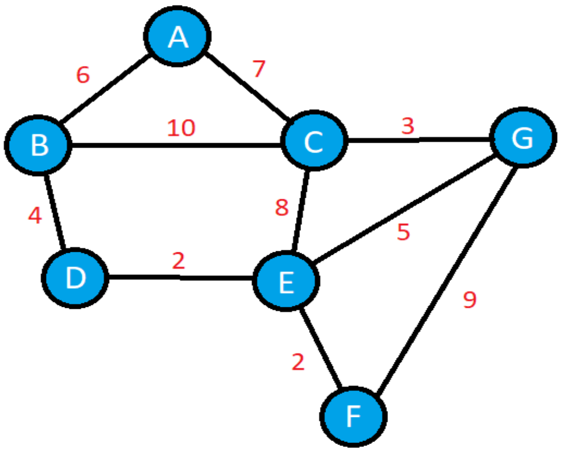


Fig 1.2: My graph to use to test code.

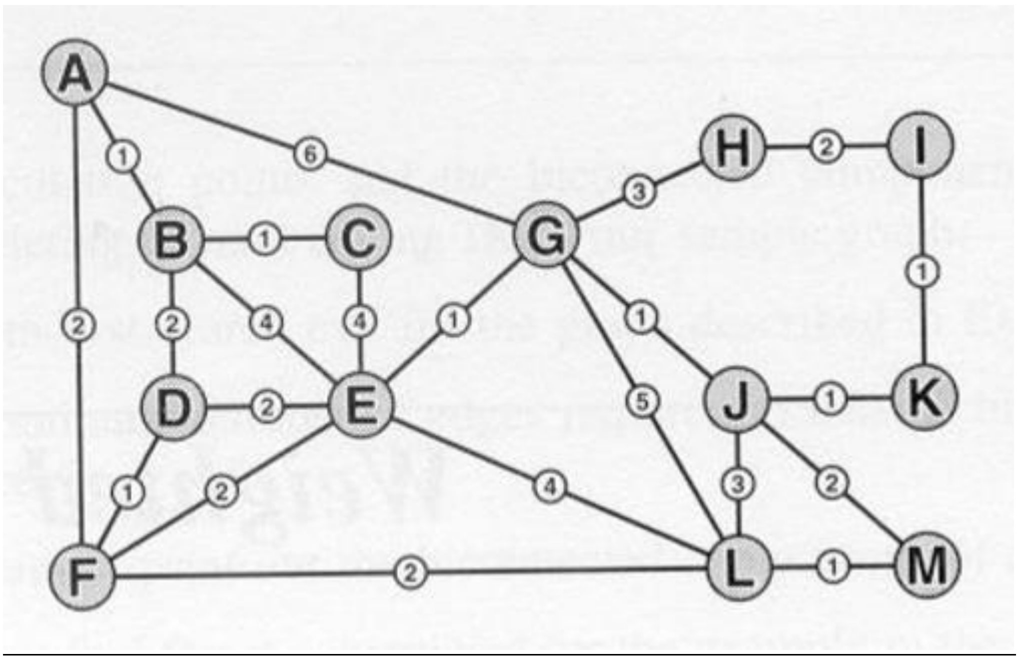
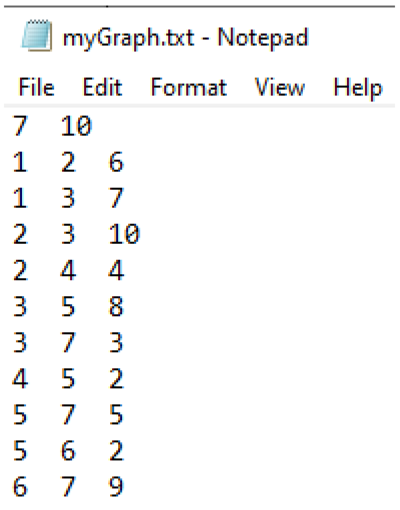


Fig 1.3: Graph given in assignment to test code on, taken from textbook by Sedgewick.

To store these graphs in memory, I used the method discussed in class, a text file. The first line of the text file lists the number of a vertices and number of edges, separated by whitespace. The remaining lines in the text file lists the edges, with their respective weights. Column 1 and 2 are the vertices/nodes. Column 3 is the weight. Below is the text file for my graph shown in Fig 1.2:



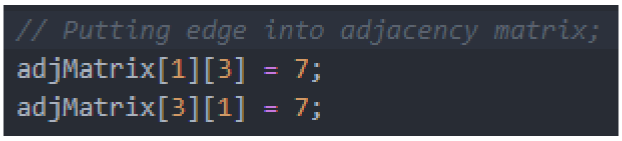
```
7  10
1  2  6
1  3  7
2  3  10
2  4  4
3  5  8
3  7  3
4  5  2
5  7  5
5  6  2
6  7  9
```

Fig 1.4: Text file of my graph.

Java implementation of graph

To even begin any work on implementing Depth First or Prim's algorithm, I needed a way to store a graph in a java program. There are two main approaches to achieving this; An adjacency matrix or an adjacency list.

An adjacency matrix is simply a 2-D array. This array keeps track of all the edges between vertices and the weights. So, suppose we have a graph with 5 vertices and 5 edges. We create a 2-D array, adjMatrix[6][6]. The rows and columns are defined as the number of vertices + 1. if we have an edge between A – C with weight 7, we would enter this into the array as follows:



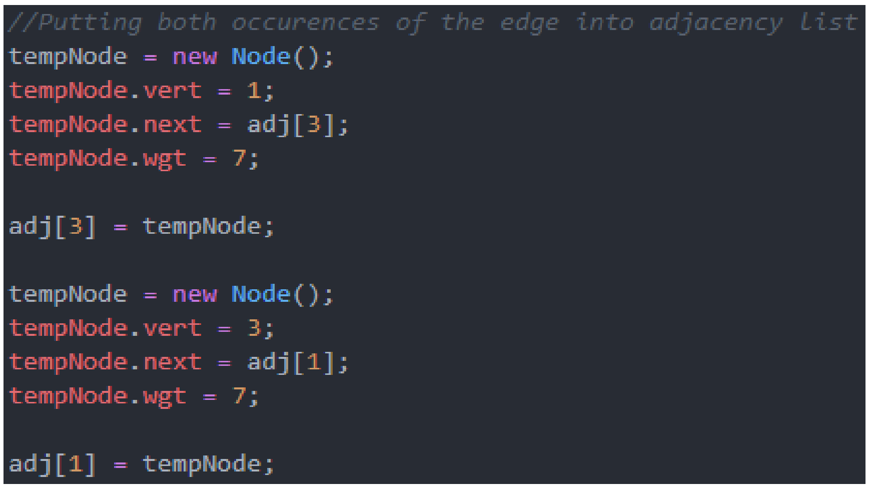
```
// Putting edge into adjacency matrix;
adjMatrix[1][3] = 7;
adjMatrix[3][1] = 7;
```

So, if we were to look at the matrix and go along row 4, we would now see the number 7 at column 2, indicating an edge of weight 7 between vertices 3 and 1 (C and A).

The main issue with this approach, is that for sparse graphs, most of the matrix is filled with 0's. This can be considered a waste of memory and the problem only worsens as more vertices are added to the graph.

Adjacency lists are a much more efficient way of representing a graph in memory. Each vertex has an associated linked list in memory. Each vertex is initialized in the list with a sentinel node. This is used to mark the end of the list, useful for navigation. This is a more efficient approach, as nodes can be dynamically added or removed and there are no empty slots in memory not being used.

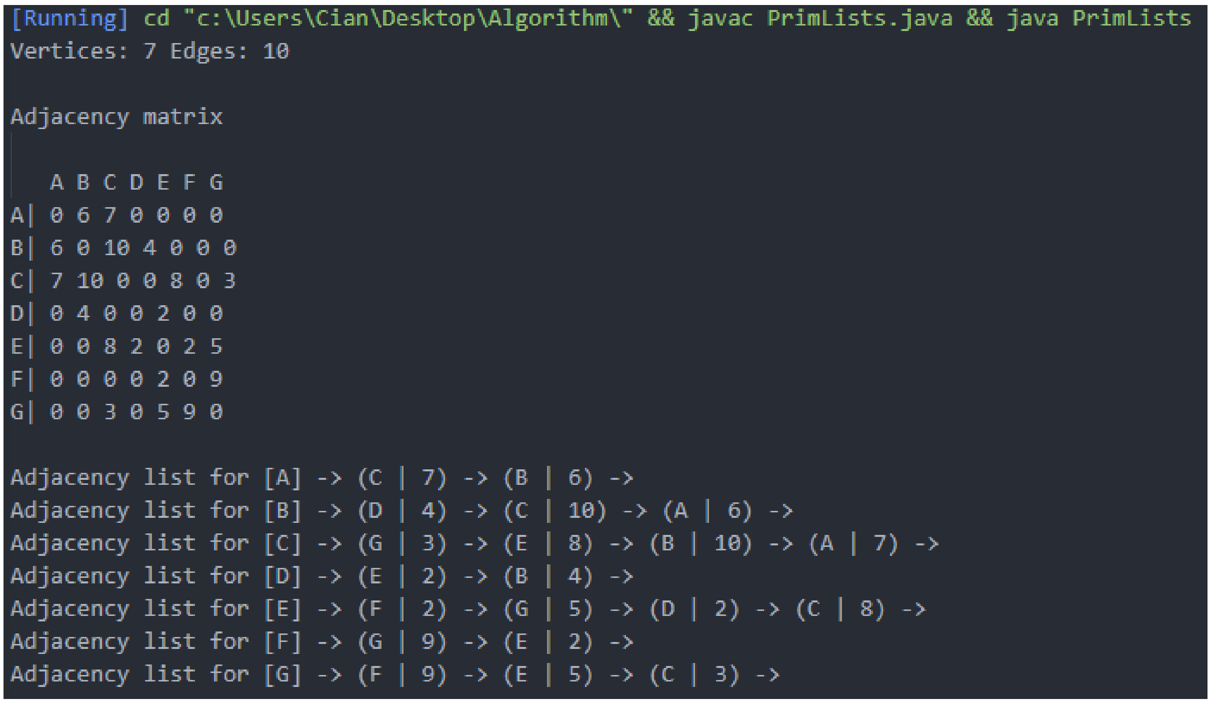
Let’s take the above example once again, an edge between A – C with weight 7. Suppose an array adj[]. With this approach we must create 2 nodes and enter them into the list. This is how it would look in java:



```java
//Putting both occurences of the edge into adjacency list
tempNode = new Node();
tempNode.vert = 1;
tempNode.next = adj[3];
tempNode.wgt = 7;

adj[3] = tempNode;

tempNode = new Node();
tempNode.vert = 3;
tempNode.next = adj[1];
tempNode.wgt = 7;

adj[1] = tempNode;
```

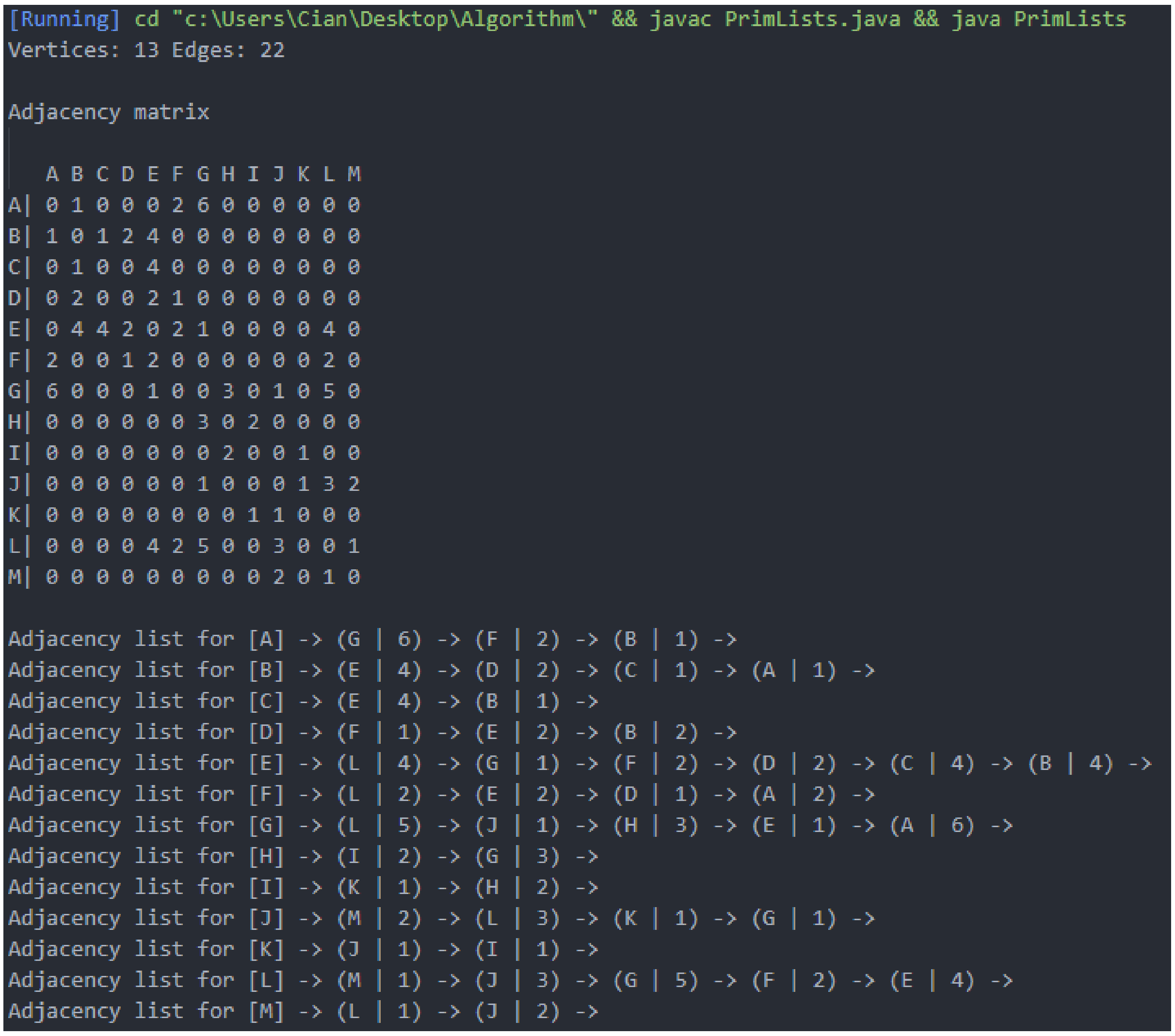
It goes without saying that these numbers are hardcoded for the sake of this example. In my actual implementation they are assigned dynamically based on the values being read from the text file.

I have implemented both methods in my program to demonstrate my understanding of them. Below is the output of my program when using my created graph (Fig 1.2):



```
[Running] cd "c:\Users\Cian\Desktop\Algorithm\" && javac PrimLists.java && java PrimLists
Vertices: 7 Edges: 10

Adjacency matrix

   A B C D E F G
A| 0 6 7 0 0 0 0
B| 6 0 10 4 0 0 0
C| 7 10 0 0 8 0 3
D| 0 4 0 0 2 0 0
E| 0 0 8 2 0 2 5
F| 0 0 0 0 2 0 9
G| 0 0 3 0 5 9 0

Adjacency list for [A] -> (C | 7) -> (B | 6) ->
Adjacency list for [B] -> (D | 4) -> (C | 10) -> (A | 6) ->
Adjacency list for [C] -> (G | 3) -> (E | 8) -> (B | 10) -> (A | 7) ->
Adjacency list for [D] -> (E | 2) -> (B | 4) ->
Adjacency list for [E] -> (F | 2) -> (G | 5) -> (D | 2) -> (C | 8) ->
Adjacency list for [F] -> (G | 9) -> (E | 2) ->
Adjacency list for [G] -> (F | 9) -> (E | 5) -> (C | 3) ->
```

Fig 1.5a: Program outputting adjacency matrix and lists of my graph.



```
[Running] cd "c:\Users\Cian\Desktop\Algorithm\" && javac PrimLists.java && java PrimLists
Vertices: 13 Edges: 22

Adjacency matrix

   A B C D E F G H I J K L M
A| 0 1 0 0 0 2 6 0 0 0 0 0 0
B| 1 0 1 2 4 0 0 0 0 0 0 0 0
C| 0 1 0 0 4 0 0 0 0 0 0 0 0
D| 0 2 0 0 2 1 0 0 0 0 0 0 0
E| 0 4 4 2 0 2 1 0 0 0 0 4 0
F| 2 0 0 1 2 0 0 0 0 0 0 2 0
G| 6 0 0 0 1 0 0 3 0 1 0 5 0
H| 0 0 0 0 0 0 3 0 2 0 0 0 0
I| 0 0 0 0 0 0 0 2 0 0 1 0 0
J| 0 0 0 0 0 0 1 0 0 0 1 3 2
K| 0 0 0 0 0 0 0 0 1 1 0 0 0
L| 0 0 0 0 4 2 5 0 0 3 0 0 1
M| 0 0 0 0 0 0 0 0 0 2 0 1 0

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

Fig 1.5b: Program outputting adjacency matrix and lists of provided graph.

In my opinion, visualising the two methods like this is important, as it highlights the inefficiency of the matrix approach. There are more 0's than any other number in the matrix, leading to a lot of wasted space in memory. This is a graph with only 13 vertices, one can quickly begin to realise how this problem scales when dealing with hundreds or thousands of vertices.

Now that we have our graphs in a java program, I would like to discuss Depth first search/traversal and Prim's algorithm in relation to these graphs.

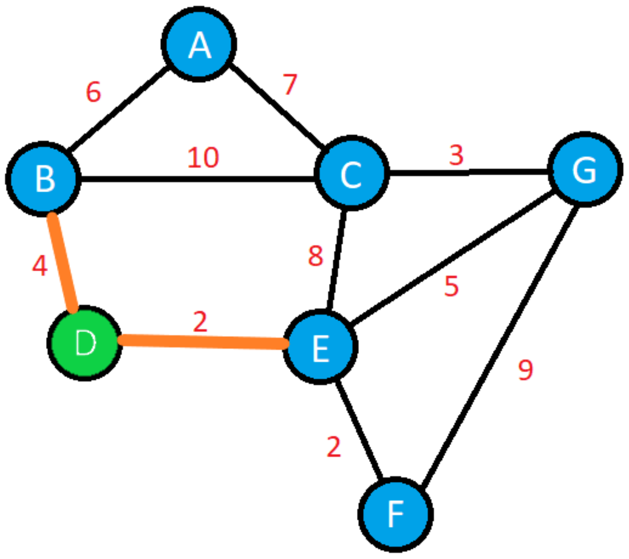
Prim's algorithm

This is one of several algorithms designed to calculate the minimum spanning tree (MST) of a graph. The MST is a subset of the edges in a graph which connect all vertices/nodes together without any cycles (visiting a node twice) with the least possible total edge weight. The real-world application of an MST could be finding the most time-efficient path between several locations, wherein the locations are nodes and the weights could be calculated as the time needed to travel from two locations.

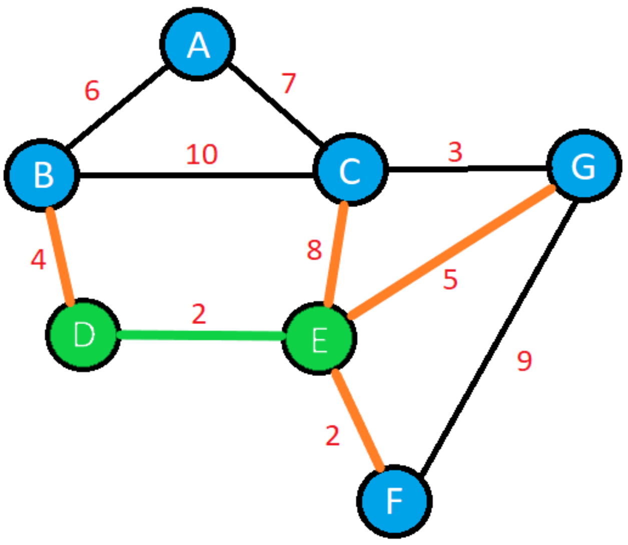
This algorithm was published in 1930 by a Czech mathematician Vojtěch Jarník. The reason it is more commonly referred to as Prim's algorithm is because it was later republished by Robert C. Prim in 1957. It is said that Jarník is the first Czechoslovak mathematician whose work had an impact on a global scale.

The algorithm can best be described with the use of an example. I will be using my connected graph to demonstrate how it works. The process for a disconnected graph is the same, just repeated for each sub-graph.

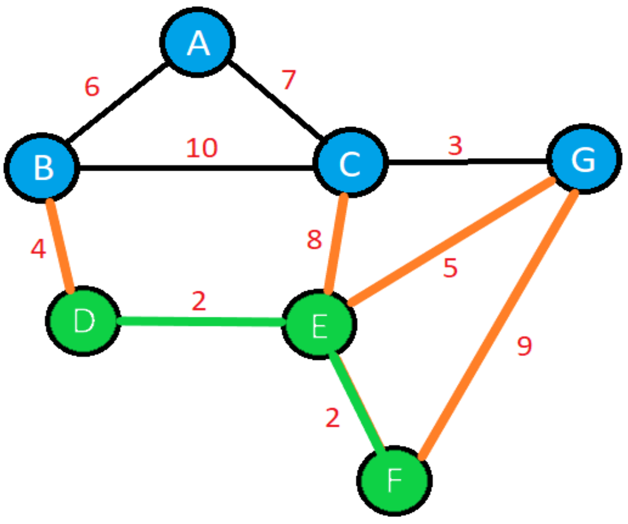
First, we pick an arbitrary starting vertex, let's take D. Our goal is to reach all nodes, without cycling, with the least amount of weight. We can consider D as being visited (green).

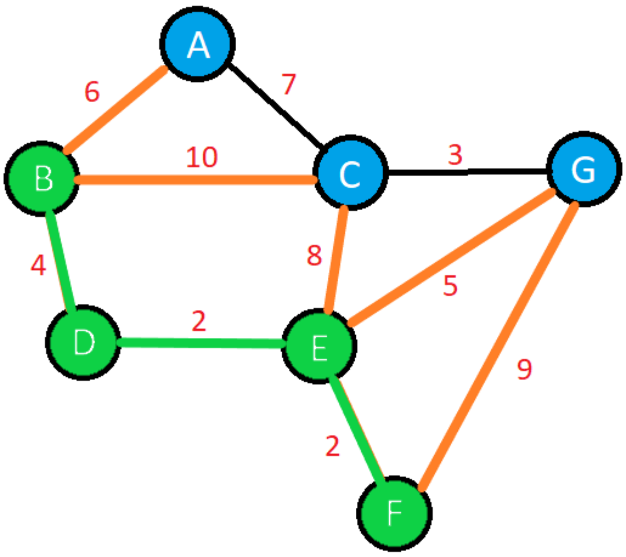


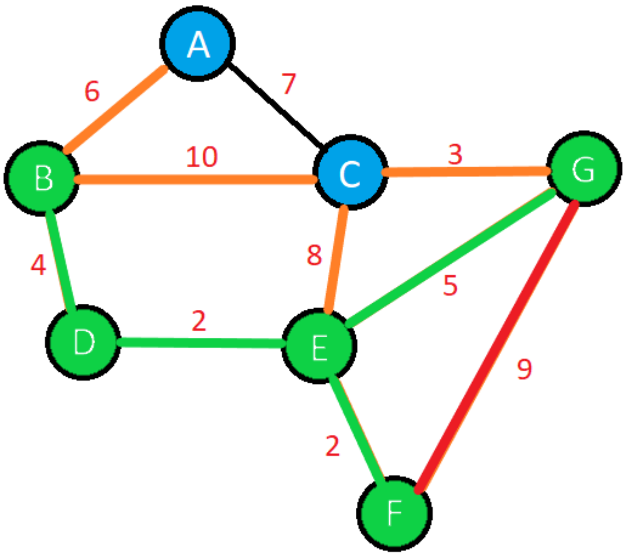
The edges in orange are the edges which we consider for the next node in the spanning tree. We take the one with the least weight, since it is a greedy algorithm. So, we now have nodes D and E in our MST. Now we must consider all edges from both D <u>AND</u> E.



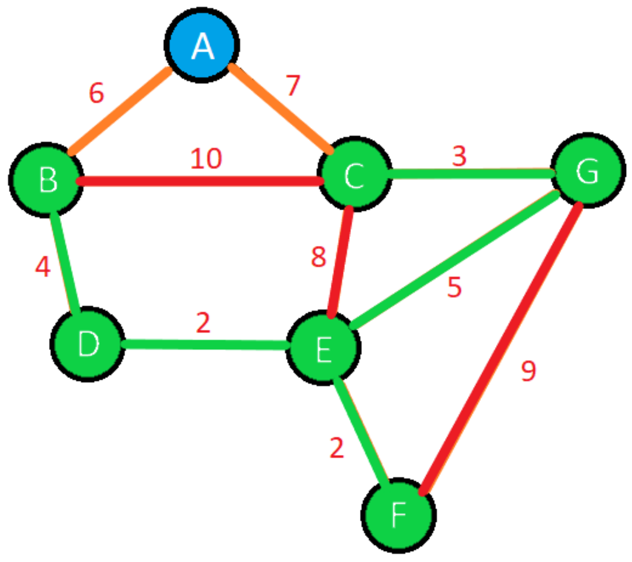
Out of the orange edges, E to F is the least weight, so we will choose that. We will repeat this process until all nodes have been visited, as such:



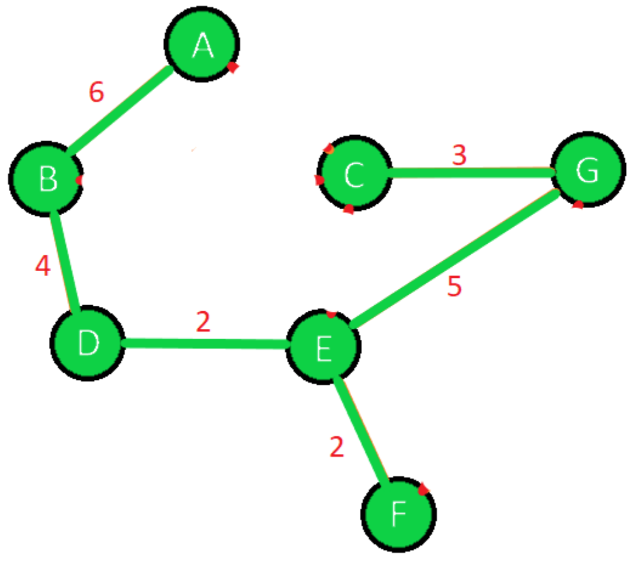

A
B
C
D
E
F
G
6
7
10
3
4
8
5
2
9
2


A
B
C
D
E
F
G
6
7
10
3
4
8
5
2
9
2

Note: Edges in red are taken out of consideration as they connect 2 visited nodes and are not in our tree.

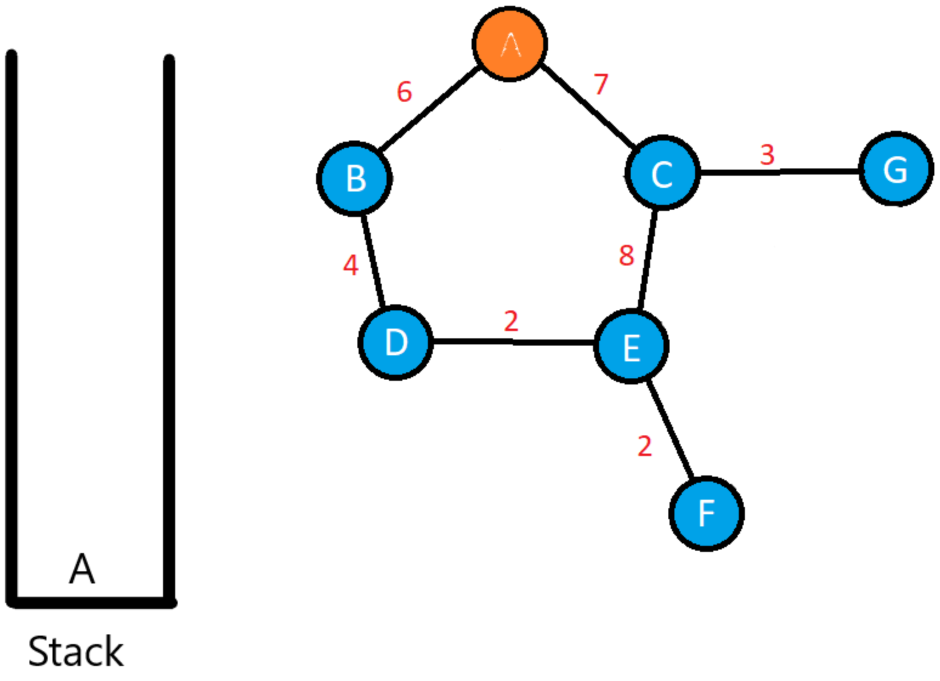


In the end, we are left with the following Minimum Spanning Tree with a weight of 22:

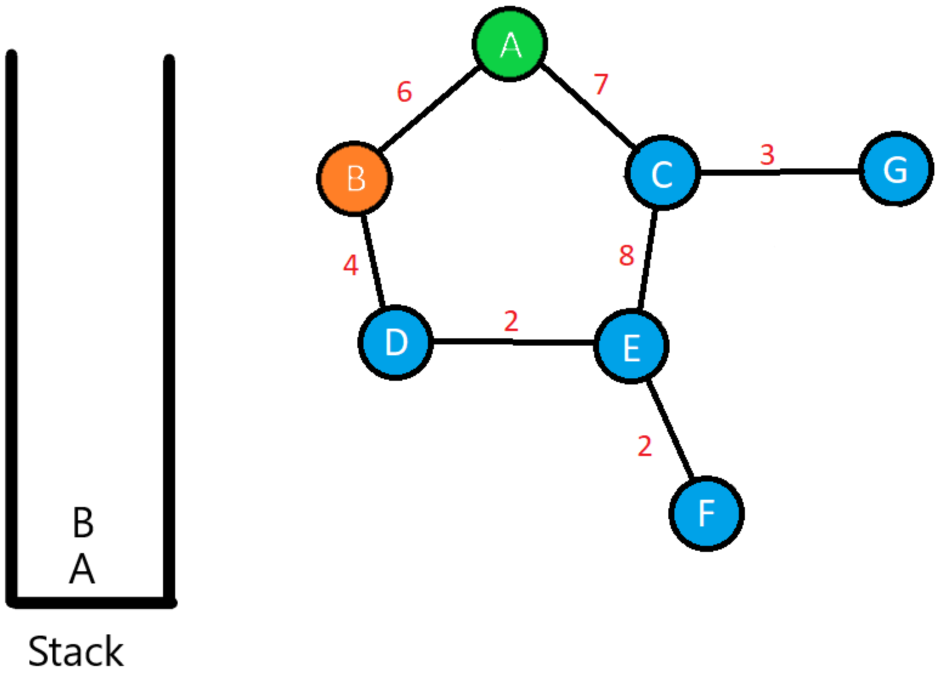


Depth first search

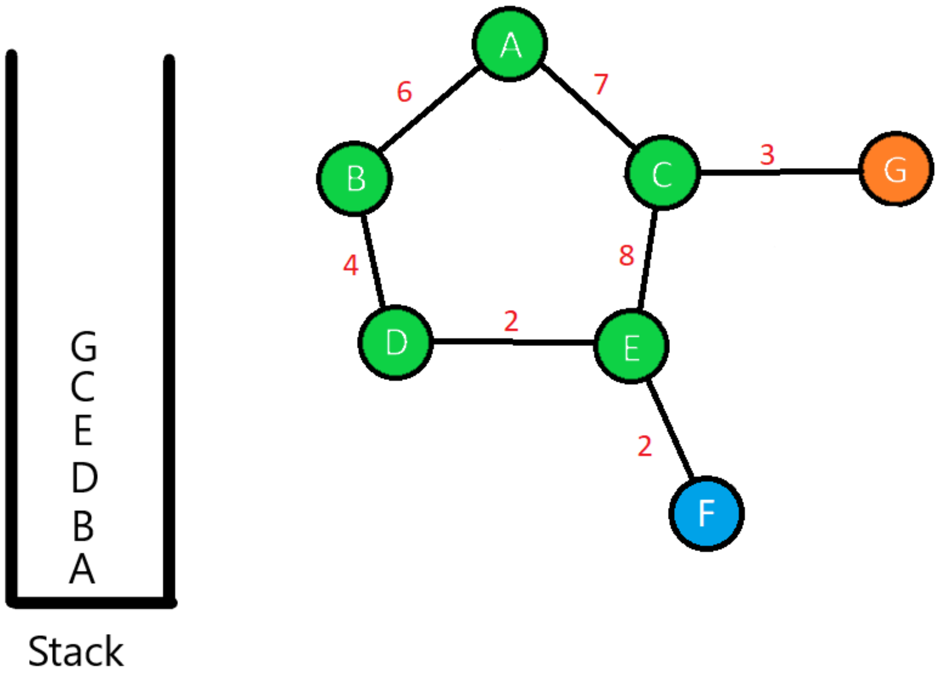
This algorithm is used to traverse all nodes in a graph. Much like prim's algorithm, we start off at an arbitrarily chosen root node. However, we are not trying to create a spanning tree, simply to visit all nodes. It can be implemented as a recursive or iterative function. In this case, let's use A as our root node. I have modified my graph slightly for this example, as it is more suitable for this algorithm. Green nodes are visited, orange node is the node currently on top of stack. We begin by pushing node A onto the stack.



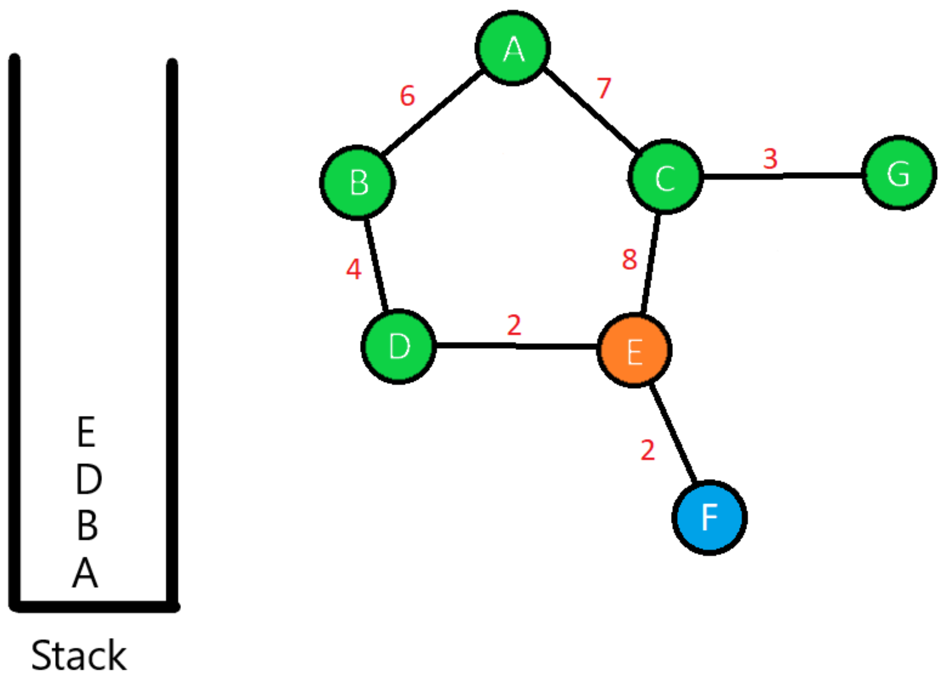
We check for adjacent nodes from the node on top of the stack. Weight is not considered in this algorithm. If the node has an unvisited neighbour, push this node onto the stack and mark it as visited. If there are multiple unvisited neighbours, choose one. I will be going on alphabetical order in such cases for this example. If a node has no unvisited neighbours, we pop it off the stack. This is repeated until the stack is empty, indicating we have visited all nodes.



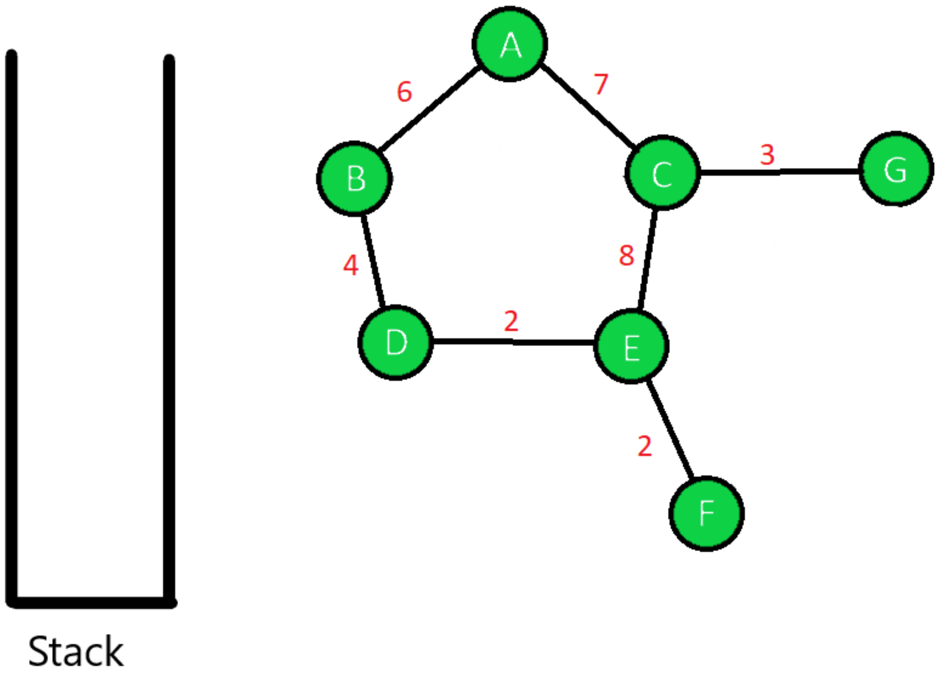
B has only one neighbour, so we will push D onto the stack. Let’s skip forward and examine what happens when we reach a node with no unvisited neighbours.



We are currently at node G. It has no neighbours which are unvisited. This is when we pop G off the stack. Node C is now on top of the stack. Its neighbours are visited, so it is also popped off. Node E is now on top of the stack and we can see it has an unvisited neighbour.

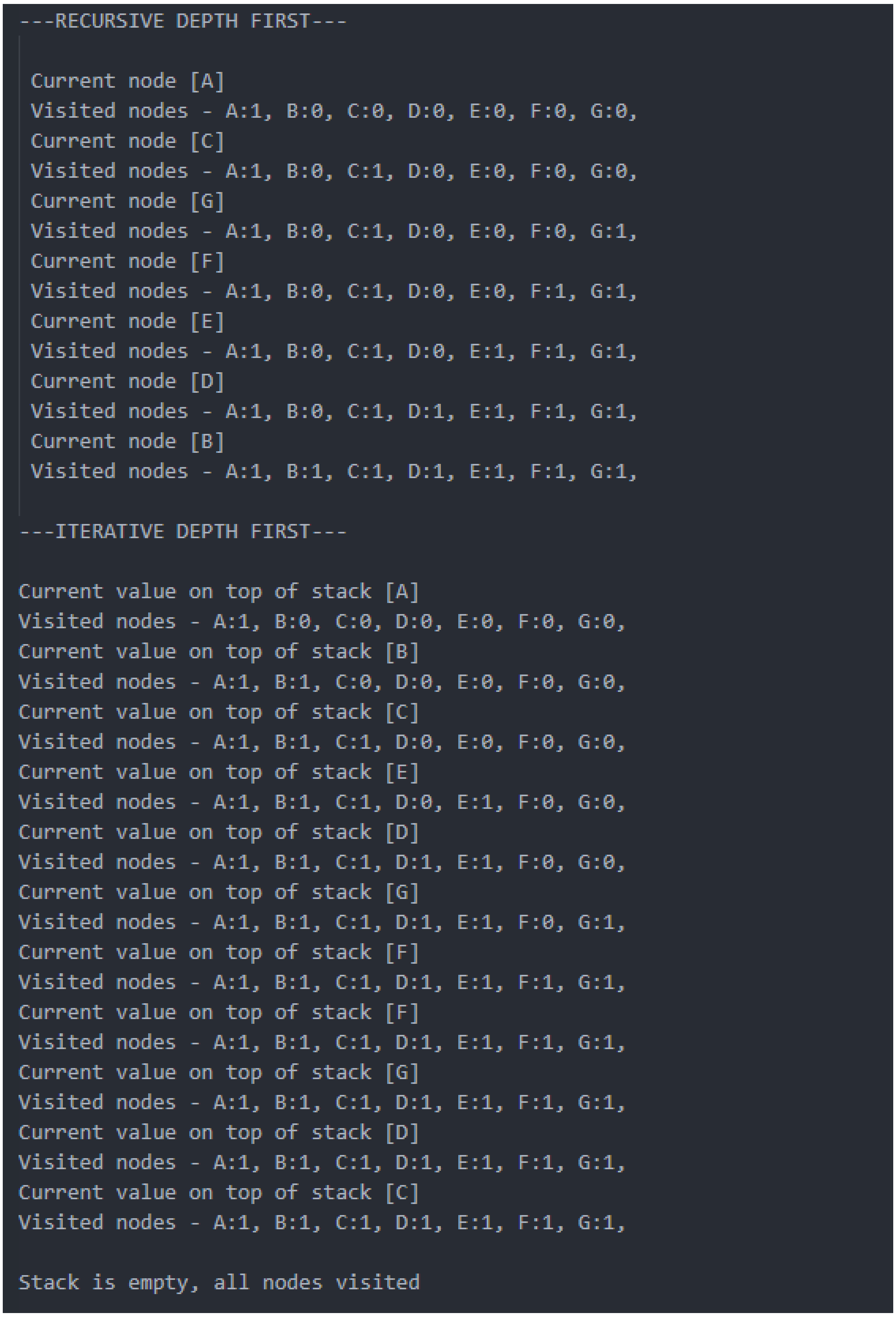


We will do the same thing we did when we reached node G. F will be popped off the stack and eventually the stack will be emptied, indicating that all nodes have been visited, as below:



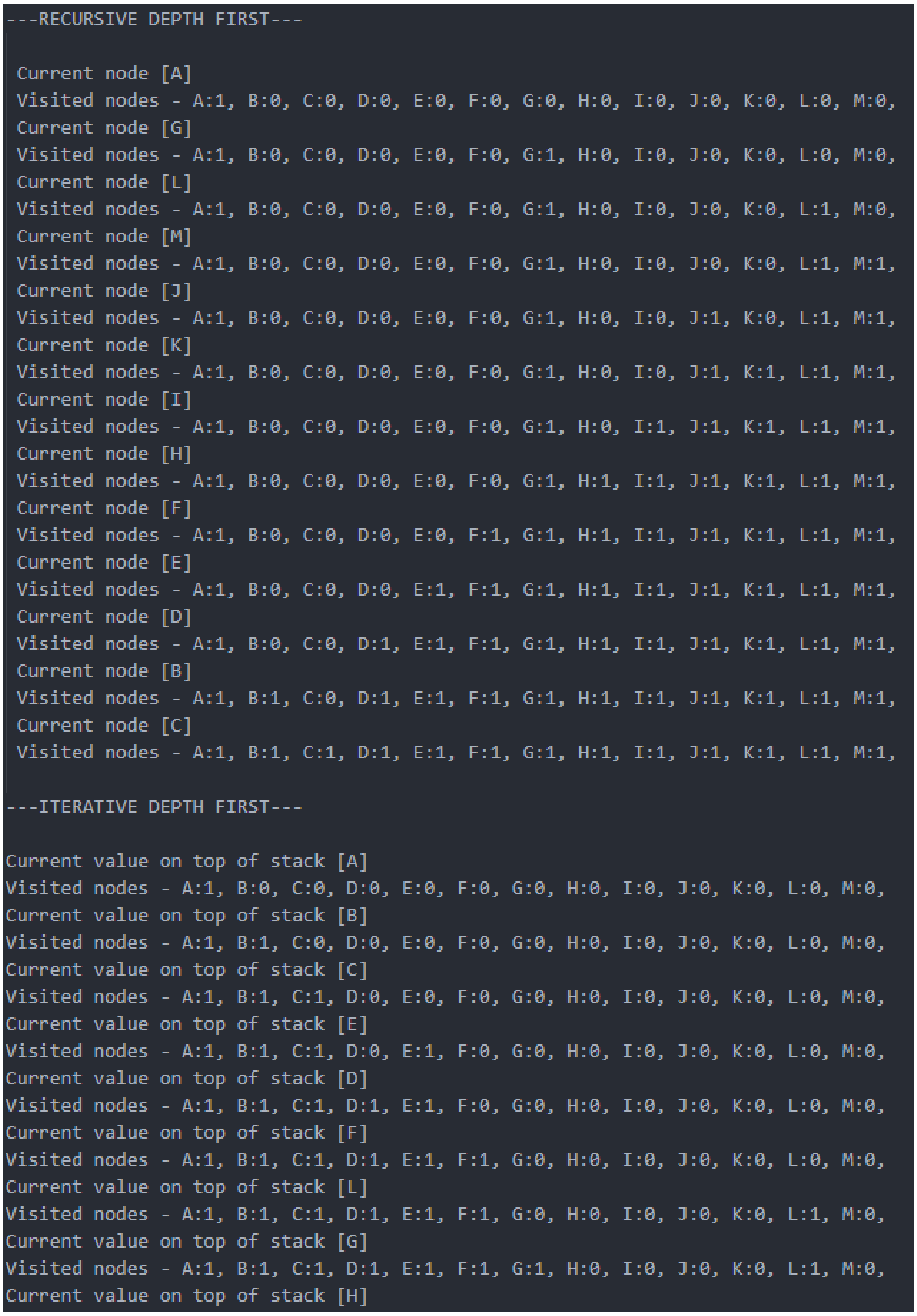
I have implemented this function both recursively and iteratively in my java program. I have made sure that the program outputs a lot of information to the console in relation to the algorithms, for example the current value at the top of the stack, the value in the visited array, etc.

Below are snippets from the console when running the program on each graph. The number of lines printed to the console is quite large for anything but very small graphs, alas, it is designed to show exactly what the program is doing step by step, to show my understanding of the algorithm. The output can be verified by running the program on your own system.

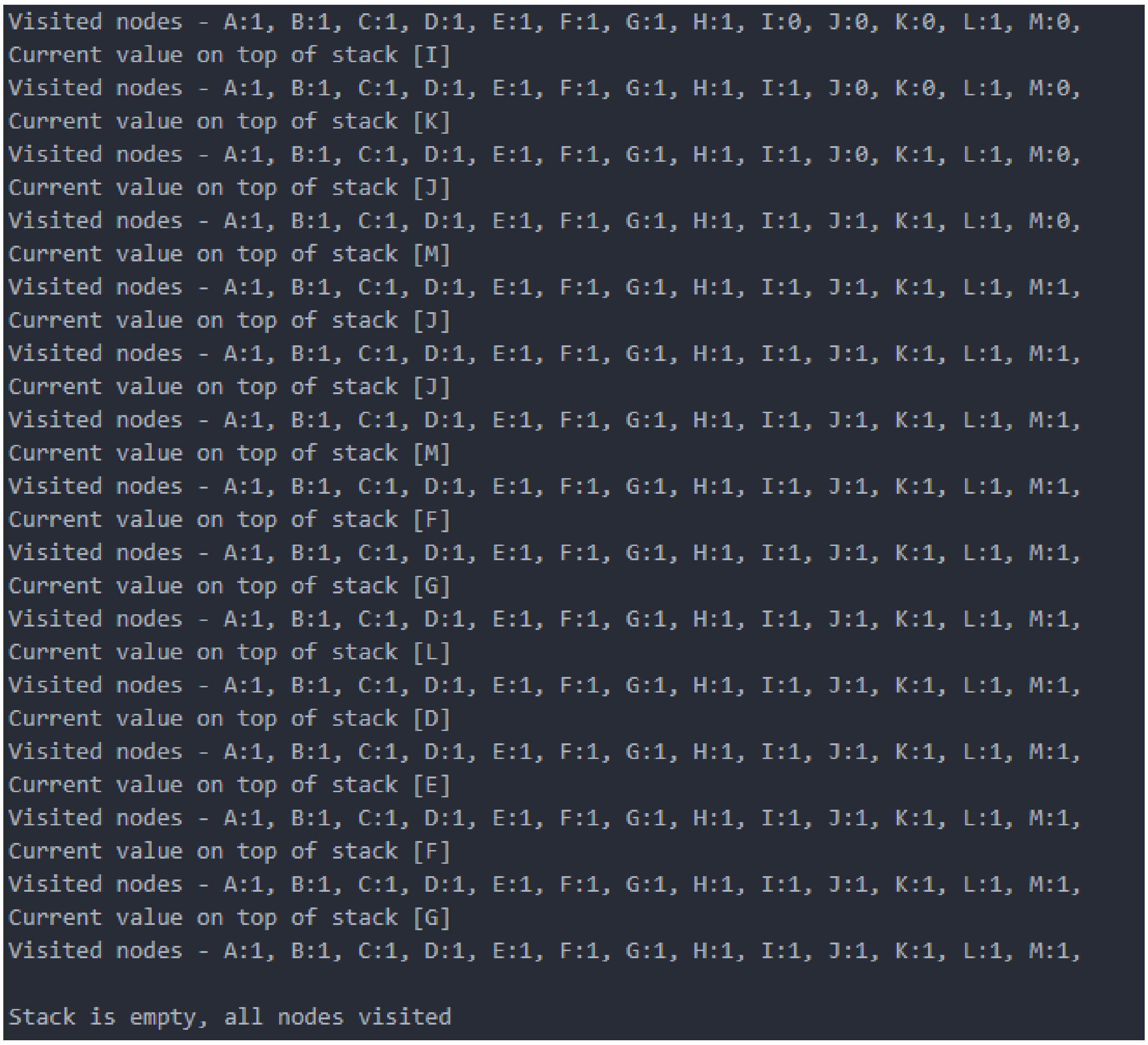


```
---RECURSIVE DEPTH FIRST---

 Current node [A]
 Visited nodes - A:1, B:0, C:0, D:0, E:0, F:0, G:0,
 Current node [C]
 Visited nodes - A:1, B:0, C:1, D:0, E:0, F:0, G:0,
 Current node [G]
 Visited nodes - A:1, B:0, C:1, D:0, E:0, F:0, G:1,
 Current node [F]
 Visited nodes - A:1, B:0, C:1, D:0, E:0, F:1, G:1,
 Current node [E]
 Visited nodes - A:1, B:0, C:1, D:0, E:1, F:1, G:1,
 Current node [D]
 Visited nodes - A:1, B:0, C:1, D:1, E:1, F:1, G:1,
 Current node [B]
 Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1,

---ITERATIVE DEPTH FIRST---

Current value on top of stack [A]
Visited nodes - A:1, B:0, C:0, D:0, E:0, F:0, G:0,
Current value on top of stack [B]
Visited nodes - A:1, B:1, C:0, D:0, E:0, F:0, G:0,
Current value on top of stack [C]
Visited nodes - A:1, B:1, C:1, D:0, E:0, F:0, G:0,
Current value on top of stack [E]
Visited nodes - A:1, B:1, C:1, D:0, E:1, F:0, G:0,
Current value on top of stack [D]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:0, G:0,
Current value on top of stack [G]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:0, G:1,
Current value on top of stack [F]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1,
Current value on top of stack [F]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1,
Current value on top of stack [G]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1,
Current value on top of stack [D]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1,
Current value on top of stack [C]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1,

Stack is empty, all nodes visited
```

Fig 1.6a: Program outputting workings of depth first on my graph



```
---RECURSIVE DEPTH FIRST---

 Current node [A]
 Visited nodes - A:1, B:0, C:0, D:0, E:0, F:0, G:0, H:0, I:0, J:0, K:0, L:0, M:0,
 Current node [G]
 Visited nodes - A:1, B:0, C:0, D:0, E:0, F:0, G:1, H:0, I:0, J:0, K:0, L:0, M:0,
 Current node [L]
 Visited nodes - A:1, B:0, C:0, D:0, E:0, F:0, G:1, H:0, I:0, J:0, K:0, L:1, M:0,
 Current node [M]
 Visited nodes - A:1, B:0, C:0, D:0, E:0, F:0, G:1, H:0, I:0, J:0, K:0, L:1, M:1,
 Current node [J]
 Visited nodes - A:1, B:0, C:0, D:0, E:0, F:0, G:1, H:0, I:0, J:1, K:0, L:1, M:1,
 Current node [K]
 Visited nodes - A:1, B:0, C:0, D:0, E:0, F:0, G:1, H:0, I:0, J:1, K:1, L:1, M:1,
 Current node [I]
 Visited nodes - A:1, B:0, C:0, D:0, E:0, F:0, G:1, H:0, I:1, J:1, K:1, L:1, M:1,
 Current node [H]
 Visited nodes - A:1, B:0, C:0, D:0, E:0, F:0, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
 Current node [F]
 Visited nodes - A:1, B:0, C:0, D:0, E:0, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
 Current node [E]
 Visited nodes - A:1, B:0, C:0, D:0, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
 Current node [D]
 Visited nodes - A:1, B:0, C:0, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
 Current node [B]
 Visited nodes - A:1, B:1, C:0, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
 Current node [C]
 Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,

---ITERATIVE DEPTH FIRST---

Current value on top of stack [A]
Visited nodes - A:1, B:0, C:0, D:0, E:0, F:0, G:0, H:0, I:0, J:0, K:0, L:0, M:0,
Current value on top of stack [B]
Visited nodes - A:1, B:1, C:0, D:0, E:0, F:0, G:0, H:0, I:0, J:0, K:0, L:0, M:0,
Current value on top of stack [C]
Visited nodes - A:1, B:1, C:1, D:0, E:0, F:0, G:0, H:0, I:0, J:0, K:0, L:0, M:0,
Current value on top of stack [E]
Visited nodes - A:1, B:1, C:1, D:0, E:1, F:0, G:0, H:0, I:0, J:0, K:0, L:0, M:0,
Current value on top of stack [D]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:0, G:0, H:0, I:0, J:0, K:0, L:0, M:0,
Current value on top of stack [F]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:0, H:0, I:0, J:0, K:0, L:0, M:0,
Current value on top of stack [L]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:0, H:0, I:0, J:0, K:0, L:1, M:0,
Current value on top of stack [G]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:0, I:0, J:0, K:0, L:1, M:0,
Current value on top of stack [H]
```



```
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:0, J:0, K:0, L:1, M:0,
Current value on top of stack [I]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:0, K:0, L:1, M:0,
Current value on top of stack [K]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:0, K:1, L:1, M:0,
Current value on top of stack [J]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:0,
Current value on top of stack [M]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
Current value on top of stack [J]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
Current value on top of stack [J]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
Current value on top of stack [M]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
Current value on top of stack [F]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
Current value on top of stack [G]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
Current value on top of stack [L]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
Current value on top of stack [D]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
Current value on top of stack [E]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
Current value on top of stack [F]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,
Current value on top of stack [G]
Visited nodes - A:1, B:1, C:1, D:1, E:1, F:1, G:1, H:1, I:1, J:1, K:1, L:1, M:1,

Stack is empty, all nodes visited
```

Fig 1.6b: Program outputting workings of depth first on Sedgewick graph

Reflection

I'll admit, this assignment was not initially very appealing. In the beginning it was a bit overwhelming trying to wrap my head around the concept of representing a graph using the adjacency list method.

However, after some time researching the topic, I eventually managed to achieve a better understanding of graphs in general. I found it quite satisfying to solve problems that would puzzle me for sometimes hours. I also quite enjoyed visually explaining the algorithms with the use of simple diagrams made in MS Paint. It made me realise that some problems, especially graphs, benefit greatly from visual explanations. While writing up this report, I have had some ideas stir up in my head about the real-world applications of this material. I will most likely be starting a personal project this Summer, putting to use some of the things I learned while completing this assignment. I believe a project which involves graphs, stacks, minimum spanning trees etc. would look good on a portfolio.