CS320 Homework 5

Dustin Randall

November 14, 2025

1 Show the BFS and DFS traversal of the provided graph, starting from vertex F.

1.1 BFS Traversal

f h d b i n m

1.2 DFS Traversal

f h d b n m i

2 Give the pseudocode for converting a binary tree into an adjacentcy list.

2.1 Pseudocode

I would approach this problem using a BFS traversal of the tree. Each time we visit a node, we add it's children to the adjacentcy list, and to the queue. If we know that we're given a complete binary tree, we can early out when we reach a node with no children.

```
function toAdjList(root) {
    adjList = new List<List<Node>>()
    if(root is null) return

    queue = new Queue(root)
    while(!queue.isEmpty()) {
        current = queue.dequeue()
        adjlist[current] = List<Node>()
        if(current.left is not null) {
            adjList[current].add(current.left)
            queue.enqueue(current.left)
        }
        if(current.right is not null) {
```

```
                adjList[current].add(current.right)
                queue.enqueue(current.right)
            }
            // optionally early out here if right is null
        }
        return adjList
    }
```

2.2 Runtime analysis

The runtime of this algorithm is $O(n)$ because each node is visited exactly once, and the work done is constant for each node.

3 Design a BFS search using an adjacentcy matrix.

3.1 Pseudocode

Given an adjacentcy matrix, we take the starting point, and find all of it's connected neighbors. Unfortunately, with an adjacentcy matrix, we have to check every other vertex to see if it's connected or not. Each connected neighbor gets added to the queue, and the algorithm continues as normal.

```
function BFS(adjMatrix, start) {
    visited = new Set()
    queue = new Queue(start)
    while(!queue.isEmpty()) {
        current = queue.dequeue()
        print(current) // or whatever you're doing in the search
        foreach neighbor in adjMatrix[current] {
            if(neighbor is not null and neighbor not in visited) {
                visited.add(neighbor)
                queue.enqueue(neighbor)
            }
        }
    }
}
```

3.2 Runtime Analysis

The runtime of this algorithm is now $O(V^2)$ because for each vertex, we need to check every other vertex to see if it's connected or not. In an adjacentcy list implementation, the runtime is $O(V + E)$ which could be $O(V^2)$ in a dense graph, but is often much better.

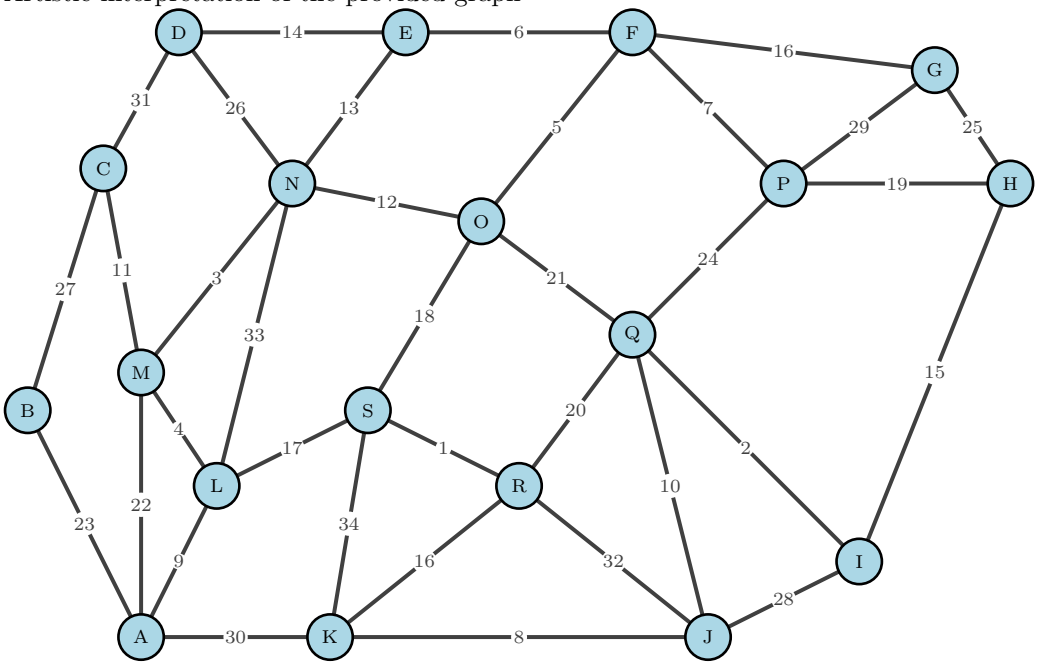
4 Provide a non-recursion based DFS

I've seen someone change from BFS to DFS by changing just a single character, but I'll simply use a stack instead of a queue.

```
function DFS(adjList, start) {
    visited = new Set()
    stack = new Stack(start)
    while(!stack.isEmpty()) {
        current = stack.pop()
        DoStuff(current)
        foreach neighbor in adjList[current] {
            if(neighbor not in visited) {
                visited.add(neighbor)
                stack.push(neighbor)
            }
        }
    }
}
```

5 Build a minimum spanning tree for the provided graph.

Artistic interpretation of the provided graph



Minimum Spanning Tree