Ian Kaiserman 00867173 **Homework 6** CSCD 320 November 21, 2022

1 Problem 1 (60 points).

Print the BFS and DFS that starts from the vertex f of the following graph. If a node has multiple next-hops, then search the next-hops in the order of their vertical coordinates from the lower ones to the higher ones. For example, node d has two next-hop neighbors b and i. Say you are now visiting d, and you have not visited b and i yet. Next you will first visit b because its vertical coordinate is lower. Note that the textbook's DFS algorithm tries all vertices as the starting node, but here only need to show the print of the DFS that starts from the vertex f. You do not have to show the trace of the procedure, but instead you only need to print the output of the BFS and DFS. For example: fh...

DFS: fdbminh

BFS: fhdbinm

2 Problem 2 (40 points).

Tree can be viewed as a special type of graph and thus can be represented by an adjacency list or an adjacency matrix. You can view the binary tree as a direct graph where the direction of each edge is pointing from the parent node to the child node. Now you are given the root node of a binary tree and also the tree's size n. You are also guaranteed that each node has a distinct key drawn from 1,2,...,n. (1) Design an efficient algorithm to create the adjacency list representation of the given binary tree, where each vertex can be represented by the key stored at the corresponding tree node. Give your algorithmic idea and its pseudocode; (2) Analyze the time complexity of your algorithm in the big-oh notation and make the bound as tight as possible.

The idea for this algorithm would be to create a 2D adjacency matrix representing the adjacency being true or false for the nodes, and recursively search the left and right sides of the nodes, assigning a 1 to the index at the node and its child to show adjacency. If the child on that side doesn't exist the algorithm simply skips over it. Recursion might not be the most memory efficient, but it is time efficient as it only has to do a multiple of n instructions.

```
new 2D int array adj[n][n]
treeAdjacency(u) {
  if u is leaf, return
  else if u.left exists {
    adj[u][u.left] = 1
    treeAdjacency(u.left)
  }
  if u.right exists {
    adj[u][u.right] = 1
    treeAdjacency(u.right)
  }
}
```

Since this is an example of recursion similar to other binary tree related problems, the pattern shows that each recursive call happens n/2 times, twice. Once for left and once for right. So the constant time operations within the function happen a multiple of n times, therefore the time complexity of the algorithm is $O(n)$

3 Problem 3 (20 points).

The graph BFS algorithm that we have discussed in class uses the adjacency list representation for the graph. Now you are asked to (1) give the same graph BFS algorithm but use the adjacency matrix representation of the graph; (2) analyze the time complexity of your algorithm and compare it with the time complexity of the one that uses the adjacency list representation.

```
    BFS(G,s) {
      // All pseudocode the same up until the while loop
      while(FIFO.size > 0) {
        u = FIFO.dequeue()
        for(i = 0; i < G.Adj[u].length; i++) // i checked for length of
                                                adjacency list for u
          v = G.Adj[u][i] // adjacency value
          if(v == 1) { // Check if adjacency to u is true
            if(G.V[i].color == WHITE) { // G.V[i] is the location in the
                                           vertices of the one being checked
              G.V[i].color = GRAY
              G.V[i].d = u.d
              G.V[i].p = u
              FIFO.enqueue(G.V[i])
            }
          }
        print(u);
        u.color = BLACK
      }
    }
```

The total time complexity using the adjacency matrix is $O(|V|+n)$, which is slower than the adjacency list version. This is because instead of getting to use a pre-created list of edges stored in G.Adj[], we have to check each value of u's adjacency array to check for values of 1, and then do the steps from there. Other than that, everything else is the same, but the adjacency list is still a bit more efficient, especially for larger data sets.

4 Problem 4 (20 points).

The graph DFS algorithm that we discussed in class uses the recursion-based structure. Now you are asked to give a non-recursion based graph DFS algorithm. Analyze the time cost of your algorithm. Hint: use and maintain your own stack.

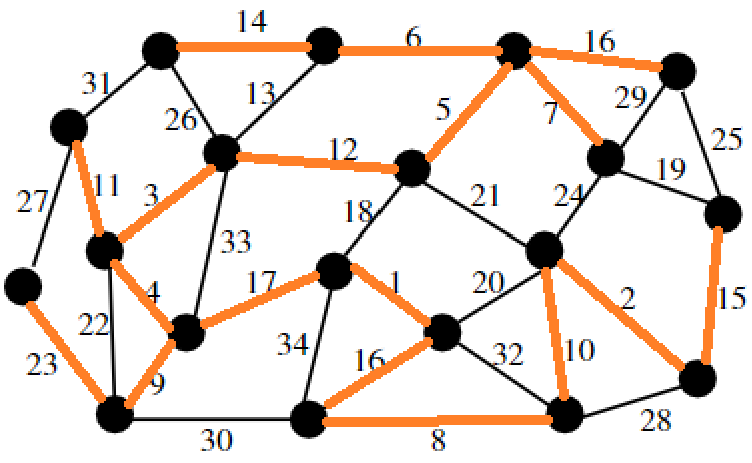
For the non-recursive algorithm, a stack can be used to simultaneously keep track of nodes that are needed, while allowing iteration to deeper parts of the graph to keep the concept of depth-first search. As a node is searched, its adjacencies should be pushed onto the stack, then one of those adjacencies should be popped and printed/stored, then the same process is done on THAT node's adjacencies as well. This will allow the traversal to go deeper, while also making sure that the deeper nodes are always printed AFTER the node it came from.

```
DFS(G,s) { // s is simply a starting point for simplicity
  stack.push(s)
  while stack not empty {
    v = stack.pop()
    if discovered[v] == true, continue
    discovered[v] = true
    add v to dfs
    for each vertex u in G.Adj[v] {
      if u not discovered, stack.push(u)
    }
  }
}
```

The total time complexity of this is the same as the recursive version $O(|V| + |E|)$ since it iterates through all the vertices and edges involved, however since it is not recursive and simply iterative in one function call, it is much more memory efficient especially for larger data sets.

5 Problem 5 (20 points).

Show a minimum spanning tree (MST) of the following connected undirected graph by using any method. You don't have to trace the algorithm that you use, but instead you can just show the MST by making the tree edges in a different color. Give the total weights of the MST that you find.



The total MST weight is 179.