CS180-Spring20 Homework2

Gurbir Singh Arora

TOTAL POINTS

95 / 100

QUESTION 1

1 Binary tree 10 / 10

✓ **- 0 pts Correct**

- 2 pts Inaccurate expression

- 4 pts not stating the base case

- 4 pts inductive case not complete or accurate

- 1 pts not clear handwriting or scanning

QUESTION 2

2 BFS 20 / 25

- 0 pts Correct

- 10 pts Incorrect code

- 15 pts Incorrect or missing explanation

✓ **- 5 pts Incomplete proof of correctness**

- 5 pts No pseudocode

QUESTION 3

3 Diameter 30 / 30

✓ **- 0 pts Correct**

- 5 pts a) incomplete

- 15 pts b) incorrect

- 5 pts b) we dont assume it's binary tree

- 5 pts b) incomplete

- 10 pts b) time complexity not O(n)

- 1 pts unclear handwriting or scanning

- 2 pts wrong definition of distance

- 15 pts a) incorrect

- 2 pts minor mistake

QUESTION 4

4 Semi-connected 35 / 35

✓ **- 0 pts Correct**

- 5 pts (a) 1st step is not "run topological sort"

- 5 pts (a) 2nd step is not "if edges (vi, vi+1) exists, then semiconnected"

- 2.5 pts (a) Incomplete proof

- 5 pts (a) Incorrect proof or no proof

- 2.5 pts (b) incomplete proof

- 5 pts (b) incorrect proof or no proof

- 3 pts (c) Did not Decompose the graph into (maximal) SCCs using Kosaraju's algorithm

- 3 pts (c) Did not Construct a pseudo-graph which will be a DAG based on (b).

- 4 pts (c) Did not run (a) to detect whether the DAG is semi-connected. If yes, then the original graph G is semiconnected;
otherwise G is not semi-connected.

- 2.5 pts (c) incomplete proof

- 5 pts (c) Incorrect proof or no proof

- 35 pts Completely blank

- 2 pts (c) Did not mention how in 2nd step of the algorithm.

Gurmbir Arora
105178554

CS 180 HW #2

* node, nodes, & vertex, vertices use interchangably in proof

1) Want to show: A binary tree T w/ n nodes where $n_0(T)$, number of leaves of binary tree T, $n_2(T)$, number of nodes in T w/ 2 children satisfy $n_1(T) - 1 = n_2(T)$

Base case: A binary tree with 1 nodes has 1 leaf and no nodes w/ 2 children. So, $n_1(T) = 1$ & $1 - 1 = n_2(T) = 0$, thus the condition holds for this case.

Induction Hypothesis: A binary tree T' with n nodes, satisfies $n_1(T') - 1 = n_2(T')$

Inductive Process: Assume this condition holds for a binary tree w/ n nodes; and prove this condition to be true for a binary tree with n+1 nodes.

Let T be an arbitrary binary tree w/ n+1 nodes. Let x be a leaf of the tree. Since the tree has more than 1 node, x isn't the root so it must have parent y. Let T' obtained by deleting the leaf x.

Case 1: If parent y has 2 children
Number of leaves in tree reduced by 1, so $n_1(T') = n_1(T) - 1$
Number of nodes w/ 2 children reduced by 1, $n_2(T') = n_2(T) - 1$
By the induction hypothesis, we see that $n_1(T') = n_2(T') + 1$

$n_1(T') = n_2(T') + 1 \Rightarrow n_1(T) = n_2(T) + 1$

This proves the condition holds for T w/ n+1 nodes.

Case 2: If parent-y has 1 child, then y becomes a leaf. Number of leaves & nodes w/ 2 children remains unchanged.

$n_1(T') = n_1(T)$

$n_2(T') = n_2(T)$

$n_1(T') = n_2(T') + 1 \Rightarrow n_1(T) = n_2(T) + 1$

Therefore, this condition holds for T w/ n+1 nodes.

☑ proved by induction

1 Binary tree 10 / 10

✓ **- 0 pts** **Correct**

- 2 pts Inaccurate expression

- 4 pts not stating the base case

- 4 pts inductive case not complete or accurate

- 1 pts not clear handwriting or scanning

2) Modify the BFS algorithm to also output # of shortest paths from source node s to other nodes.

2)

```
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private Node[] nodes;
    private int V;   // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists
    //mark visited based on what has been traversed
    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
    }

    // prints BFS traversal from a given source s
    void BFS(int startId)
    {
        // Mark all the vertices as not visited(By default
        // set as false)
        boolean[] visited = new boolean[nodes.length];
        // Create a queue for BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();
        private static inc EDGE_DISTANCE =6;
        // equal weight for each edge needed for BFS and to create a sense of distance within
problem.
        // Mark the current node as visited and enqueue it
        visited[startId]=true;
        //turns out visited isn't necessary
        int[] distances = new int[nodes.length];
        Arrays.fill(distances, -1);
```

```
    // fill distances with -1 to mark as unvisited
    distances[startId] = 0;
    //mark distance to source node a zero as distance to self is 0
    while(!queue.isEmpty()) {
            int node = queue.poll();
            for (int neighbor : nodes[node].neighbors) {
                //walk through each neighbor and add to queue
                if (distances[neighbor] == -1){
                    //if node hasn't been visited,
                    distances[neighbor] = distances[node] + EDGE_DISTANCE;
                    //distance to neighbord is current position + distance for each edge
                    queue.add(neighbor);
                    //if hasn't been visited then add to queue
                }
            }
        }
        return distances;
        //return the distances from source node
}
```

2 BFS 20 / 25

- **0 pts** Correct
- **10 pts** Incorrect code
- **15 pts** Incorrect or missing explanation

✓ **- 5 pts** **Incomplete proof of correctness**

- **5 pts** No pseudocode

3)

a. Design an algorithm that runs in time $O(nm)$ & finds diameter of G

Apply BFS on each node of graph & select/store greatest length from these paths (since DFS is $O(m)$ & each node is done on, time complexity should be $O(mn)$.

Do DFS from each node to find max length. While traversing, look for total distance to reach current node, if the adjacent nodes are visited already, then update value of max length if prev max length is less than current value of total distance.

Time Complexity = $O(n(n+m)) = O(nm)$ since it's connected.

3a)

```
#include<bits/stdc++.h>
using namespace std;
// visited[] array to make nodes visited
// src is starting node for DFS traversal
// prev_len is sum of cable length till current node
// max_len is pointer which stores the maximum length after DFS traversal
void DFS(vector< pair<int,int> > graph[], int src,
         int prev_len, int *max_len,
         vector <bool> &visited) {
    // Mark the src node as visited
    visited[src] = 1;

    // curr_len is for length from src
    // node to its adjacent node
    int curr_len = 0;

    // Adjacent is pair type which stores
    //destination node and total length
    pair < int, int > adjacent;

    // Traverse all adjacent
    for (int i=0; i<graph[src].size(); i++)  {
        // Adjacent element
        adjacent = graph[src][i];

        // If node is not visited
        if (!visited[adjacent.first])  {
            // Total length from src node
            // to its adjacent
            curr_len = prev_len + adjacent.second;

            // Call DFS for adjacent node
            DFS(graph, adjacent.first, curr_len,
                max_len,  visited);
        }

        // If total length till now greater than
        // previous length then update it
        if ((*max_len) < curr_len)
            *max_len = curr_len;

        // make curr_len = 0 for next adjacent
```

```
        curr_len = 0;
    }
}

// n is number of nodes in graph
// cable_lines is total cable_lines among the cities
// or edges in graph
int diameter(vector<pair<int,int> > graph[], int n) {
    // maximum distance among the connected
    // nodes
    int max_len = INT_MIN;

    // call DFS for each node to find maximum
    // distance
    for (int i=1; i<=n; i++) {
        // initialize visited array with 0
        vector< bool > visited(n+1, false);

        // Call DFS for src vertex i
        DFS(graph, i, 0, &max_len, visited);
    }

    return max_len;
}
int main() {
    // n = number of nodes
    int n = 6;

    vector< pair<int,int> > graph[n+1];

    // create undirected graph
    // first edge
    graph[1].push_back(make_pair(2, 3));
    graph[2].push_back(make_pair(1, 3));

    // second edge
    graph[2].push_back(make_pair(3, 4));
    graph[3].push_back(make_pair(2, 4));

    // third edge
    graph[2].push_back(make_pair(6, 2));
    graph[6].push_back(make_pair(2, 2));
```

```
    // fourth edge
    graph[4].push_back(make_pair(6, 6));
    graph[6].push_back(make_pair(4, 6));

    // fifth edge
    graph[5].push_back(make_pair(6, 5));
    graph[6].push_back(make_pair(5, 5));

    cout << "Diameter = " << diameter(graph, n);

    return   0;
}
```

b) If graph is a tree, $T=(V,E)$, diameter can be computed efficiently. Give an $O(n)$ time algorithm to find T.

Algorithm:
maxDepth(){
1) If tree is empty return 0
2) Else
a) get max depth of left subtree recursively by calling maxDepth(tree→left-subtree)
b) get max depth of right subtree recursively by calling maxDepth(tree→right-subtree)
c) get max of max depth of left & right subtrees and add 1 to it for the current node
max_depth = max(max depth of left sub, max depth of right sub) + 1
d) return max_depth

Time Complexity: $O(n)$ as the algorithm traverses each node exactly once
↑ this only works for binary tree
for n-ary tree,

1) run bfs to find farthest node from rooted tree, called A
2) run bfs from A to find farthest node from A let B
3) Distance btwn node A & B is diameter
$O(n)$ time complexity.

Wasn't sure if prob was asking for binary tree or n ary trees, so I did both.

3b1)

```
#include <bits/stdc++.h>
using namespace std;

class node
{
    public:
    int data;
    node* left;
    node* right;
};

/* Compute the Diameter of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int diameter(node* node)
{
    if (node == NULL)
        return 0;
    else
    {
        /* compute the diameter of each subtree */
        int lDia = diameter(node->left);
        int rDia = diameter(node->right);

        /* use the larger one */
        if (lDia> rDia)
            return(lDia + 1);
        else return(rDia + 1);
    }
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
node* newNode(int data)
{
    node* Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;

    return(Node);
```

```
}

int main()
{
   node *root = newNode(1);

   root->left = newNode(2);
   root->right = newNode(3);
   root->left->left = newNode(4);
   root->left->right = newNode(5);

   cout << "Diameter of tree is " << diameter(root);
   return 0;
}
```

3b2)

```
#include <bits/stdc++.h>
using namespace std;

// Here 10000 is maximum number of nodes in
// given tree.
int diameter[10001];

// The Function to do bfs traversal.
// It uses iterative approach to do bfs
// bfsUtil()
int bfs(int init, vector<int> arr[], int n)
{
    // Initializing queue
    queue<int> q;
    q.push(init);

    int visited[n + 1];
    for (int i = 0; i <= n; i++) {
        visited[i] = 0;
        diameter[i] = 0;
    }

    // Pushing each node in queue
    q.push(init);

    // Mark the traversed node visited
    visited[init] = 1;
```

```
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int i = 0; i < arr[u].size(); i++) {
            if (visited[arr[u][i]] == 0) {
                visited[arr[u][i]] = 1;

                // Considering weight of edges equal to 1
                diameter[arr[u][i]] += diameter[u] + 1;
                q.push(arr[u][i]);
            }
        }
    }

    // return index of max value in diameter
    return int(max_element(diameter + 1,
                           diameter + n + 1)
               - diameter);
}

int findDiameter(vector<int> arr[], int n)
{
    int init = bfs(1, arr, n);
    int val = bfs(init, arr, n);
    return diameter[val];
}

// Driver Code
int main()
{
    // Input number of nodes
    int n = 6;

    vector<int> arr[n + 1];

    // Input nodes in adjacency list
    arr[1].push_back(2);
    arr[1].push_back(3);
    arr[1].push_back(6);
    arr[2].push_back(4);
    arr[2].push_back(1);
    arr[2].push_back(5);
    arr[3].push_back(1);
```

```
    arr[4].push_back(2);
    arr[5].push_back(2);
    arr[6].push_back(1);

    printf("Diameter of n-ary tree is %d\n",
           findDiameter(arr, n));

    return 0;
}
```

3 Diameter 30 / 30

✓ **- 0 pts** **Correct**

- 5 pts a) incomplete

- 15 pts b) incorrect

- 5 pts b) we dont assume it's binary tree

- 5 pts b) incomplete

- 10 pts b) time complexity not O(n)

- 1 pts unclear handwriting or scanning

- 2 pts wrong definition of distance

- 15 pts a) incorrect

- 2 pts minor mistake

4)

a. Consider a simplified case where G is a DAG. Design an O(m) time algorithm to test whether a DAG is semi-connected

Basic Idea: Compute a topological sort B check if there's an edge between each consecutive pair of nodes in the topological order

according to Piazza, we are allowed to assume we have topological sort B decompose a graph into SCCs functions, so

Initial thought: Graph dag(X) ← create a graph containing X elements

dag.topologicalSort(); ← perform a topological sort on the graph

Claim: A DAG is semi connected in top. sort if for each i, there's an edge (v_i, v_{i+1})

⇒ Now, if there's no edge (v_i, v_{i+1}) for some i, then there must not be a path (v_{i+1}, v_i) because the dag is topologically sorted, thus the graph isn't semi-connected.

If for every i, there's an edge (v_i, v_{i+1}), then for each i, j $(i < j)$ there's a path $v_i \to v_{i+1} \to \ldots \to v_{j-1} \to v_j$, and thus the graph is semi connected.

From here, we use the algorithm described on the next page.

s.t. ⇒ such that

1. Find maximal SCCs in the graph
2. Build the SCC graph $G' = (V', E')$ s.t. V' is a set of SCC & $E' = \{(V_1, V_2)$ s.t. there's a v_1 in V_1 & v_2 in V_2 s.t. (v_1, v_2) is in $E\}$.
3. Do topological sort on G'
4. Check if for each i, there's an edge (V_i, V_{i+1})

Why this is correct: If the DAG is semi connected, for a pair (v_1, v_2) s.t. there's a path $v_1 \Rightarrow \dots \Rightarrow v_2$, let V_1, V_2 be their SCCs. There's a path from $V_1 \Rightarrow V_2$ & thus from v_1 to v_2 since all nodes in V_1 & V_2 are strongly connected. If the algorithm outputs true, then for any two given nodes v_1, v_2 they must be in SCC in V_1 & V_2. Thus for all pairs $u, v \in V$ there's a path from u to v or from v to u.

G is semi connected iff there's an edge (v_i, v_{i+1}) for all $i = 1, 2, \dots, n-1$, where n = # of nodes

i) If there's no edge from v_i to v_{i+1}, G isn't semi connected bc there's no path from $v_i \Rightarrow v_{i+1}$

& vice versa. ii) If there's an edge (v_i, v_{i+1}) for all i, G is semi-connected bc there's a path $(v_1, v_2, \dots v_n)$ so that for each pair of vertices v_i & v_j, they are connected. Thus G is semi connected.

So, perform topological sort on G & then check

This results in an $O(m)$ time complexity as the sort is linear time & the edge check is also linear time $O(m)$.

4b) Prove that for any directed graph G, its pseudo-graph G' constructed this way will be a DAG

$G \Rightarrow$ SCCs

SCCs $\rightarrow G'$ where $G' = (V', E')$ where V' has k nodes & $(v_i, v_j) \in E'$ iff there's an edge pointing from a node in S_i to a node in S_j in the original graph

we know that if G' has a topological ordering, then G' is a DAG

By the algorithm presented previously, the SCC graph is built such that V' is a set of SCCs & $E' = \{(V_1, V_2) \mid$ there's $v_1 \in V_1$ & $v_2 \in V_2$ s.t. (v_1, v_2) is in $E\}$ aka $(v_i', v_j') \in E'$

This means we are able to do a topological sort on G', meaning G' has a topological ordering.

Proof: Sps by contradiction G' has topological ordering $v_1, v_2, v_3 \ldots v_n$ & a cycle C. Let v_i be the lowest indexed node on C, & v_j be the node on C just before v_i, thus (v_j, v_i) is an edge. By our choice of i, $j > i$, which contradicts the assumption that $v_1, v_2, v_3 \ldots v_n$ was a topological ordering.

4 c) Design an algorithm to test semi connectivity of a directed graph in $O(m)$ time.

Algorithm set-up:

1) Find maximal SCCs in the graph
2) Build SCC graph $G' = (V', E')$ s.t. V' is a set of SCCs s.t. $E' = \{(V_1, V_2)$ s.t. there's v_1 in V_1 & v_2 in V_2 s.t. (v_1, v_2) is in $E\}$
3) Do topological sort on G'
4) Check if for every i, there's an edge V_i, V_{i+1}

This works b/c)

-If the graph is semi connected, for a pair (v_1, v_2) s.t. there's a path $v_1 \to \ldots \to v_2$ Let V_1, V_2 be their SCCs. There's a path from V_1 to V_2 and thus also from $v_1 \to v_2$ since all node in V_1 & V_2 are strongly connected.

-If the algorithm yields true, then for any two given nodes v_1, v_2, we know they're SCC in V_1 & V_2. There's a path from $V_1 \to V_2$ and thus also from $V_1 \to v_2$.

This time complexity is $O(V+E) \to O(m+m) \to O(m)$ as all steps require at most $O(V+E)$. so, $O(m)$

Using the maximal SCC creator provided in the problem along with the topological sorter, we completed steps #1-3 and are left with G'

Now, in order to check for semi connectivity, we must complete step 4. We must check for each i, where the ordering is $v_1, v_2, \ldots v_n$, $1 \leq i \leq n$, that there's an edge V_i, V_{i+1}.

this can be done with

```
for(int i=0; i< # of nodes-1; i++){
    if(edge@P DNE)
        return false
    if(edge@P+1 DNE)
        return false
}
return true;
```

$O(V+E) \Rightarrow O(n+m) \rightarrow O(n)$ time complexity, as each step takes at most $O(V+E)$ time.

4 Semi-connected 35 / 35

✓ **- 0 pts Correct**

- 5 pts (a) 1st step is not "run topological sort"

- 5 pts (a) 2nd step is not "if edges (vi, vi+1) exists, then semiconnected"

- 2.5 pts (a) Incomplete proof

- 5 pts (a) Incorrect proof or no proof

- 2.5 pts (b) incomplete proof

- 5 pts (b) incorrect proof or no proof

- 3 pts (c) Did not Decompose the graph into (maximal) SCCs using Kosaraju's algorithm

- 3 pts (c) Did not Construct a pseudo-graph which will be a DAG based on (b).

- 4 pts (c) Did not run (a) to detect whether the DAG is semi-connected. If yes, then the original graph G is semiconnected;
otherwise G is not semi-connected.

- 2.5 pts (c) incomplete proof

- 5 pts (c) Incorrect proof or no proof

- 35 pts Completely blank

- 2 pts (c) Did not mention how in 2nd step of the algorithm.