Homework 2

Raymond Lin, 304937942

May 3, 2020

Problem 1

Claim: Any binary tree has one fewer full nodes (nodes with two children) than leaf nodes.

Proof by induction:

Let n denote the number of full nodes and l denote the number of leaf nodes.

Base Case: Consider the single node binary tree, T. T has $n = 0$ and $l = 1$. Then, it is true that

$$n = l - 1$$
$$0 = 1 - 1$$

Induction Hypothesis: Assume the following is true: $l = n + 1$

Inductive Statement: Want to show $l' = n' + 1 for \;\; n' = n + 1$
For any binary tree, we can expand it by adding a node to an existing leaf node or to an internal node that only has one child. Then, there are two cases.

1. If we add a node, a, to the leaf node, b, then a becomes the new leaf node, while b becomes an additional internal node. However, the number of full nodes (nodes with two children) does not change, because the new internal node b still only has 1 child. Furthermore, the number of leaf nodes also does not change, because a becomes the new leaf node, but b is no longer a leaf node.

   We have:

   $$n' = n$$
   $$l' = l$$

   By the inductive hypothesis:

   $$l = n + 1$$

   Substituting, we have:

   $$l' = n' + 1$$

2. If we add a node, a, to the internal node, b, then a becomes a new leaf node, and b becomes a full node. In other words, both the number of leaf nodes and the number of full nodes increases by 1.

   We have:

   $$n' = n + 1$$
   $$l' = l + 1$$

By the inductive hypothesis:

$$l = n + 1$$

Substituting, we have:

$$\begin{aligned} l' - 1 &= n' - 1 + 1 \\ l' &= n' + 1 \end{aligned}$$

□

Problem 2

Given a graph, G, the BFS algorithm traverses G by starting at a particular node and defining the next level as the nodes that are adjacent to the starting node. Then the newly discovered nodes become the next level to be searched from. In other words, at iteration i, the BFS algorithm finds all the nodes that are i steps away from the starting node. As a result, it searches through G and can find the shortest path to all the reachable nodes.

We can modify the BFS algorithm to find the number of shortest paths to any node in G. The idea is to keep track of how many paths can reach a particular node within the node's minimum level, or in other words, we only count the shortest paths to the node. For example, if we are at level j, denoted L_j, and there are x paths, each of value 1, that reach node $n \in L_j$, then in the next iteration, when we search for nodes that n is adjacent to, we propagate the value x along each of these paths. When another node in the next level receives the value x, its value becomes the sum of x and all the other paths' values that reaches that node. Then in the next iteration, it propagates this value to its neighboring nodes in the next level. We must be careful when we mark the neighboring nodes as visited, because we cannot do so until all of the nodes in a particular level are processed. This algorithm runs in $O(m)$ time, because it only visits each edge at most twice, if the edge connects two nodes in the same level. We are simply running the regular BFS algorithm, but we maintain an additional data structure to keep track of the number of shortest paths to each node.

Algorithm:

```
L[0] = s
numShortestPaths[i] = 0 ∀ i ≠ s
numShortestPaths[s] = 1
visited[i] = false ∀ i ≠ s
visited[s] = true
i = 0
while L[i] ≠ ∅
   foreach u ∈ L[i]
      foreach v such that (u,v) is an edge (ie.  v is a neighbor of u)
         if visited[v] = false
            numShortestPaths[v] += numShortestPaths[u]
            add v to L[i+1]

      // This is where we mark the nodes in the current level as visited
      foreach v such that (u,v) is an edge (ie.  v is a neighbor of u)
         visited[v] = true
   i = i + 1
```

Problem 3

a)

Idea: This algorithm to find the diameter of a graph is based on BFS. The idea of this algorithm is this: for each node in the given graph, G, run BFS on the node. Then we record the distance of the nodes the last level of the BFS search. This is our current diameter. If at any point we find a distance that is greater than our current diameter, we update our current diameter. This algorithm is correct, because it finds the longest shortest path between any two nodes in G. The algorithm runs in $O(mn)$, because we traverse through all of the edges of G, for each node in G.

Algorithm:

```
currDiam = 0
for each u ∈ V
   L[0] = u
   i = 0
   while L[i] ≠ ∅
      visited[i] = false ∀ i ≠ u
      visited[u] = true
      for each v ∈ L[i]
         for each w such that (v,w) is an edge (ie.  w is a neighbor of v)
         if visited[w] = false
            L[i + 1] = L[i + 1] ∪ {w}
            visited[w] = true
      i = i + 1
   if ∃ j such that visited[j] = false (ie.  BFS was not able to search entire graph,
meaning there are multiple separated components in the graph)
      return infinity
   if i > currDiam
      currDiam = i
return currDiam
```

b)

Idea: This algorithm is based on DFS. The idea is to use recursion to search each subtree in the tree. The search should return both the maximum depth and the diameter of the particular subtree. We can do this in the following way. At each node, recursively call the function on all of its child nodes, thus searching the subtrees rooted at the children nodes. Then, return the maximum diameter and depth between all of the recursive calls. At the parent node, once the recursive function returns, we return the maximum between the maximum diameter of its children and the sum of the two maximum depths of its children. This algorithm runs in $O(n)$, because we are using DFS to search the tree, which visits every node in the tree once.

Algorithm:

```
// Helper function to find the maximum depth and diameter of a particular tree
def maxDepthAndDiam(node):
   max1 = 0
   max2 = 0
   maxDiam = 0
   for each c such that c is a child of node
      temp = maxDepthAndDiam(c)
      currDepth = temp[0] + 1
      if currDepth > max1
         max1 = currDepth
      else if currDepth > max2
         max2 = currDepth
```

```
      currDiam = temp[1]
      if currDiam > maxDiam
        maxDiam = currDiam

   // Return a two element tuple containing the maximum depth, maximum diameter
for the subtree rooted at node
   return (max(max1, max2), max(max1 + max2, currDiam))

root = root node of our tree
return maxDepthAndDiam(root)[1]
```

Problem 4

a)

Idea: The idea for this algorithm is to identify the nodes in the DAG with indegree = 0. We know that if there are more than 1 node with indegree = 0 in the DAG, then it is not semiconnected, because there is no way for either of them to reach the other. If the DAG has a single node with indegree = 0, we can remove it, and we get a new DAG. Again, we can find the number of nodes with indegree = 0. We can do this until there are no nodes left in the DAG, whcih means the DAG is semiconnected, or until at any point, there exists more than 1 node with indegree = 0, which means the DAG is not semiconnected. This algorithm runs in $O(m)$ time, because we only traverse each edge of the DAG once.

Algorithm:

```
def isDAGSemiconnected(G):
   indegrees[i] = 0 ∀ i
   for each (u,v) ∈ G (ie.  each edge in the DAG, G)
      indegrees[v]++

   if number of nodes with indegree = 0 is > 1
      return false

   u = node in G with indegree = 0

   while ∃ i such that indegrees[i] ≠ 0
      countZeros = 0
      for each v such that (u,v) is an edge (ie.  neighbors of u)
         indegrees[v]--
         if indegrees[v] = 0
            countZeros++
            u = v
         if countZeros > 1
            return false
   return true
```

b)

Claim: Given a directed graph, G, we can construct G' by running Kosaraju's algorithm on G, and replacing each SCC (strongly connected component) with a single node. Then, G' is a DAG.

Proof (by contradiction):

- Suppose the number of nodes in G' is K
- Suppose G' is not a DAG
- Then G' contains at least 1 cycle

Intermediary Claim: Any cycle is an SCC

Proof (by contradiction):

- Let C be a cycle $v_1, ..., v_{k-1}, v_k, v_1$
- Suppose C is not an SCC
- Then, it must be that $\exists\ v_i, v_j \in$ C such that there is no path from $v_i \rightarrow v_j$

- For each pair of nodes $v_i, v_j \in$ C, $\exists$ either
  $v_i \to ... \to v_{j-1} \to v_j$ or
  $v_i \to ... \to v_{k-1} \to v_k \to v_1 \to ... \to v_{j-1} \to v_j$
- Thus, there is a path from v_i to v_j for all pairs of nodes in C
- $\implies$ Contradiction, since we said that there $\exists$ v_i, v_j such that there is no path between them

- Let K' be the number of SCC's found in G after running Kosaraju's algorithm on it
- From the problem, it must be that $K' = K$
- Because G' contains at least 1 cycle, and since any cycle is an SCC, then it must have been found by Kosaraju's algorithm
- We would have replaced that SCC with a single node in G', so K' must be $< K$
- $\implies$ Contradiction, since we assumed that $K' = K$

□

c)

Idea: As in question b), we can use Kosaraju's algorithm to decompose a graph G into SCCs. Then, we can construct G', by replacing each SCC in G with a single node. In b), we proved that G' is DAG. Then, we can use the algorithm from problem a), to determine if G' is semiconnected. If it is, then G is also semiconnected if we can show that an SCC is semiconnected (since, G' is just G with the SCCs replaced). If not, then G is not semiconnected.

Algorithm:

```
def isSemiconnected(G):
   G' = G but with its SCCs replaced with a single node

   // Run the algorithm, isDAGSemiconncted(), from part a) on G'
   if not isDAGSemiconnected(G')
      return false
   return true
```

Claim: Given any graph G, the above algorithm will determine whether or not G is semiconnected.

Proof of correctness:

- Let G' be G, but with its SCCs replaced with a single node
- G' must be a DAG (proven in part b))
- Run the algorithm in part a) on G'
- There are two cases

1. **Intermediary claim:** If G' is not semiconnected, then G is also not semiconnected

   Proof:
   - Let $v_1, ..., v_m$ be the nodes of G', and $u_1, ..., u_n$ be the nodes of G
   - Since G' is not semiconnected, $\exists$ v_i, v_j such that there is no path between them
   - Since v_i, v_j represent an SCC, pick any $u_k \in v_i$ and any $u_l \in v_j$
   - There is no path from u_k to u_l and vice versa

- Thus, G is not semiconnected

2. **Intermediary claim:** If G' is semiconnected, then G is semiconnected

   Proof:
   - Let $v_1, ..., v_m$ be the nodes of G', and $u_1, ..., u_n$ be the nodes of G
   - Since G' is semiconnected, $\exists\ v_i, v_j$ such that there is either a path from $v_i \to v_j$ or $v_j \to v_i$
   - Since v_i, v_j represent an SCC, pick any $u_k \in v_i$ and any $u_l \in v_j$
   - Let $u_s \in v_i$ and $u_t \in v_j$ be the edge nodes
   - By definition of an SCC, $\exists$ paths $u_k \to u_s$ and $u_l \to u_t$ (ie. in an SCC, there exists a path from any node to any other node)
   - Thus, if v_i can reach v_j, then u_k can also reach u_l, using the path $u_k \to ... \to u_s \to ... \to u_t \to ... \to u_l$, or vice versa
   - It must be true that $\forall\ u_k \in v_i$, $u_l \in v_j$, $\exists$ path from u_k to u_l or vice versa
   - Thus, G is semiconnected

   □