CS 180 - Algorithms and Complexity
Homework 2

Jayant Mehra
Discussion 1D

October 18, 2018

Solution 1. The following is the proof to exercise 5 on page 108 of the textbook.

Proof. Let the base case be a tree with 3 nodes where there is one parent node which has two children. Thus, the number of leaves is 2 and there is 1 node with 2 children. Our base case holds.

Let us assume that this is true for a binary tree with n nodes.

Now, lets add another node to the binary tree already containing n nodes. There are three possible locations as to where the n+1 node can be inserted:

1. The new node can be the parent of the previous root (the root of the n node tree). This does not increase the number of leaves and this does not increase the number of two-children parents. Thus, by our inductive hypothesis, the number of leaves is still exactly one more than the number of two-children parents.

2. The new node can be the child of a previously one-child parent. This increases the number of leaves by 1 and the number of two-children parents by one. Since both increase by 1 and, by our inductive hypothesis, the condition was true for n nodes, the condition still holds for n+1 nodes.

3. The new node can be the child of a leaf (i.e. node with 0 children). This does not change the number of two-children parents. This does not change the number of leaves (however, the new node becomes a leaf and the leaf node to which the new node was added does not remain a leaf). Since there is no change in the number of two-children parents and leaves, by our inductive hypothesis, the condition is true for n+1 nodes.

Thus, by induction, in a binary tree the number of nodes with two children is exactly one less than the number of leaves.

□

Solution 2. The following is the proof to exercise 6 on page 108 of the textbook

Proof. Suppose by way of contradiction that G and T are not not equal.
$\implies$ For two vertices, x and y, belonging to G such that the distance of x from u is less than or equal to the distance of y from u, there is an edge between them in G that is not present in T.

1. Suppose d(u, x) = d(u, y).

   - When we hit node x in BFS, the edge (x, y) will not be added to T because the parent of x will already be connected to y.
   - When we hit node x and we haven't hit node y (without loss of generality) in DFS, we add the edge (x, y) to T and the parent of x will not be connected to y.

   Thus, the BFS tree and DFS tree are different. This is a contradiction. Therefore, our assumption that G $\neq$ T was wrong.

2. Suppose d(u, x) < d(u, y)

   - When we hit node x in BFS, we will add the edge (x, y) as y hasn't been explored already and it is a child of x.
   - When we hit node x in DFS, there are two cases. (1) y is connected to some other node which we explore(d) before exploring y. In this case, we do not add edge (x, y) to our tree. (2) We add edge (x, y) and then recursively explore all other connections. Since our original graph was connected, y is surely connected to some other node which we will then add to our tree. That node would then not be connected to its parent but it would be in the BFS tree.

   Thus, the BFS tree and DFS tree are different. This is a contradiction and our assumption was wrong to begin with it.

By way of contradiction, we have proved that T = G for the conditions provided.

□

Solution 3. The following is the proof to exercise 7 on page 108 of the textbook.

Proof. The claim that if every node of G has degree at least $n/2$ then G is connected is **true**.

Lets suppose by way of contradiction that even if every node of G has degree at least $n/2$ it is still not connected. This assumption implies that there are at least 2 connected components in G.

$\implies$ The number of nodes in the smaller component is $\leq n/2$. Therefore, in that component (the smaller one) the degree of each vertex cannot be greater than $n/2$ - 1.
But this contradicts the fact that each node has a degree of at least $n/2$.
Therefore, our assumption is wrong and, indeed, G is connected.

□

Solution 4. The following is the solution to exercise 9 on page 110 of the textbook.

Proof. Suppose by way of contradiction each level between s and t contains at least 2 nodes. Since the distance between them is strictly greater than $n/2$, there will be at least $n/2$ levels between s and t.
Therefore, the number of nodes in the minimum case of $n/2$ levels and each level having 2 nodes will be $n/2 \times 2 = n$ nodes. But there are n nodes in total. Thus, this is a contradiction and our assumption was wrong.
$\implies$ There will be at least one level between s and t which will just contain 1 node, v which when deleted will destroy all s-t paths.

□

.
The algorithm to find that vertex is as follows:

```
1.  findNode(s)
2.      Queue Q
3.      Q.enqueue(s)
4.      levelSize ← 1
5.
6.      discovered[n]
7.      discovered[s] ← True
8.      discovered[v] ← False for all other nodes
9.
10.     while (Q ≠ ∅ )
11.         while (levelSize ≠ 0)
12.             node n ← Q.top()
13.             Q.dequeue()
14.
15.             for neighbor u of node n
16.                 if discovered[u] ← False
17.                     discovered[u] ← True
18.                     Q.enqueue(u)
19.
20.             levelSize ← levelSize - 1
21.
22.         levelSize = Q.size()
23.         if levelSize = 1
24.             return Q.top()
25.
26.     return ERROR
```

Algorithm Explanation The findNode function implements a BFS starting at s. It uses a queue based implementation. While the queue is not empty, it adds the neighbors of all elements of the queue. It then calculates the size of the queue which corresponds to the size of the level. If it detects that the size of a particular level is 1, it returns that node as that

is the breaking point of the graph.

Time Complexity Analysis Since it is a BFS with no modifications, it runs in O(m+n) time as required.

Solution 5. The following is the solution to exercise 12 on page 112 of the textbook.

The intuition of the algorithm to produce the dates of birth and death of the n people or find an inconsistency in the data is as follows:

Let the births and deaths of $P_1, P_2, P_3, ..., P_n$ be the nodes of a graph. There exists a directed edge between the birth and death nodes of P_i. There exists a directed edge between the death node of P_i and the birth node of P_j if P_i died before P_j. There is a directed edge between the birth node of P_i and the death node of P_j and one between the birth node of P_j and death node of P_i if there is an overlap in lifespan.
If there is a cycle in the graph, the data is inconsistent, otherwise it is plausible.

```
1.  buildGraph()
2.      Graph G
3.      for p in {P_1, P_2, ..., P_n}
4.          add birth and death nodes and a directed edge between them
5.      for p in {P_1, P_2, ..., P_n}
6.          for u in {P_1, P_2, ..., P_n} - p
7.              if p died before u
8.                  add directed edge between death node of p and birth node of u
9.              else if there was an overlap between p and u
10.                 add directed edge between birth node of p and death node of u
11.                 add directed edge between birth node of u and death node of p
12.
13.     return G
14.
15. findOrder()
16.     Graph G ← buildGraph()
17.     LinkedList L
18.     discovered[n]
19.     active[n]
20.
21.     discovered[v] ← False for all nodes
22.     active[v] ← False for all nodes
23.
24.     for v in nodes of G
25.         if discovered[v] = False
26.             if helper(G, L, discovered, active, v) = False
27.                 return INCONSISTENT DATA
28.     return L
```

```
29.
30. helper(G, L, discovered, active, v)
31.     active[v] ← True
32.     for neighbour u of v
33.         if active[u] = True
34.             return False
35.         if discovered[u] = False
36.             if helper(G, L, discovered, active, u) = False
37.                 return False
38.     add in front of L
39.     active[v] ← False
40.     discovered[v] ← True
```

Algorithm Explanation In lines 1-11, we build a directed graph using the rules established above.
In lines 15-40, we run a variation of topological sort using a Linked List. We maintain two boolean arrays, discovered and active, to keep track of all nodes who have been discovered and added (or errored out) to the list by the algorithm and nodes who are currently active (in the DFS) respectively.
In lines 24-27, we run over all undiscovered nodes, and run a DFS on them. We make the current node active (line 31) and run DFS on its neighbors. If a node's neighbor is currently active, we have hit a cycle and we return False (lines 33-34) and our main function returns Inconsistent Data (lines 26-27). Otherwise, we keep adding nodes to the front of the linked list and change the discovered and active states appropriately (lines 38-40). This step takes O(n + e) time.
The linked List L will have the birth and death dates of people in the correct order and can be obtained by running from the head to the tail (null) of the linked list.

Solution 6. I make the following assumptions in solving the problem:

- The entry point is the top left cell and the exit point is the bottom right cell.
- Thus, the only sensible moves are right, bottom, and **diagonal**.
- The isAlarm function runs in O(1) time and takes the coordinates of the grid as inputs and returns true if it is within the radius of an alarm and false otherwise.
- The grid is a 2D 0-indexed array.

The algorithm is as follows:

```
1. IsCrossable(Grid g[n][n])
2.     bool cache[n][n]
3.     cache[n-1][n-1] ← True
4.
5.     for i ← n-2 to 0
6.         cache[i][n-1] ← ¬ isAlarm(i, n-1) ∧ cache[i+1][n-1]
7.
8.     for i ← n-2 to 0
```

```
 9.          cache[n-1][i] ← ¬ isAlarm(n-1, i) ∧ cache[n-1][i+1]
10.
11.      for j ← n-2 to 0
12.          for i ← n-2 to 0
13.              cache[i][j] ← ¬ isAlarm(i, j) ∧ ( cache[i+1][j] ∨ cache[i][j+1] ∨ cache[i+1][j+1])
14.
15.      return cache[0][0]
```

Algorithm Explanation The lines 5-9 set the last column and last row values. For the last column, the value in each cell depends on what the value is in the cell to the bottom and whether or not it is within an alarm. Similarly, for the right column, the value depends in each cell depends on what the value is in the cell to the right of it and whether or not it is within an alarm.
In lines 11-13, the loop moves column major wise from cache[n-2][n-2] to cache[0][0]. The value in each cell depends on whether it is in an alarm vicinity and if its neighbors (bottom, right, and diagonal) are reachable to the exit or not.
Line 15 returns the value of cache[0][0]. In other words, whether the start is reachable to the end.

Time Complexity Analysis We visit each cell of the grid once to fill its boolean value in the cache. We have assumed that the isAlarm method runs in constant time. Therefore, the time complexity is $O(n^2)$.