CS 180 Homework 2

Name: Wang, Zheng
UID: 404855295

CS 180 Dis 1E
Time: F 10:00-11:50
TA: Orpaz Goldstein

Question 1 (Exercise 5, Page 108):

Here, the base case is a tree with only one vertex. In this case, there is only one leaf; also, there is no node with two children. Thus, the number of nodes with two children, zero, is one less than the number of leaves.

Now, assume that for a binary tree with k nodes, it has n number of nodes with 2 children and $n-1$ leaves. If we want to add a node v to this tree, then we connect v to a previous-existing node u on the tree.

There are two cases of u for which v can be attached.

- Firstly, if u already has one child, then add v to u will then create one new node with two children and one new leaf.
  Thus, for any binary tree with $k+1$ nodes and fall in to this case, the number of nodes with two children is $n+1$, and number of leaves is $n+1-1=n$.

- Secondly, if u is a leaf, then add v to u will eliminate one leaf and create a new leaf. Notice that in this case, we won't impact the number of nodes with two children.
  Thus, for any binary tree with $k+1$ nodes and fall into this case, the number of nodes with two children is n, and the number of leaves is $n-1-1+1=n$.

Overall, for any binary tree with $k+1$ nodes, the number of nodes with two children is exactly one less than the number of leaves.

Thus, by induction, for all binary trees, the number of nodes with two children is exactly one less than the number of leaves.

Question 2 (Exercise 6, Page 108):

Suppose towards contradiction that there exists some (x,y) that is an edge of G but (x,y) is not an edge of T.

Now, since T is obtained by doing DFS on G, to T must be a DFS tree, and therefore, one of x and y must be the ancestor of the other. Without loss of generality, let's assume that x is an ancestor of y.

Since T is also a BFS tree, as (x,y) is an edge of G, thus if x belongs to layer L_i and y belongs to layer L_j, i and j cannot be differ more than 1.

Thus, x is an ancestor of y only one layer above, which means that x is y's parent. Thus, (x,y) is an edge of T, a contradiction.

Question 3 (Exercise 7, Page 108):

Suppose towards contradiction that there exist vertices x and y in G such that they are not connected.

Thus, x and y must each connect with a **disjoint** set of $n/2$ nodes. However, this is not possible since there are only $n-2$ nodes left for x and y to connect to but the two disjoint set mentioned above will need n nodes.

Thus, by pigeon-hole principle, there exist some nodes a such that a is connected with x and at the same time, a is connected with y. So, the claim made is true.

Question 4 (Exercise 9, Page 110):

Proof.

Since the distance between s and t is strictly greater than $n/2$ thus, if we run BFS with s as a root, then it is necessary that the resulting three has the last layer L_d, where $d > n/2$.

Claim: If we run a BFS starting at s, then there exists some layer $L_i : 1 \leq i \leq d-1$ in the resulting BFS-tree such that there is only one node on L_i.

Proof of the claim: suppose that for all layer L_i, there are larger or equal to 2 nodes. Then, we have two nodes, s and t, and $2 \times (d-1) = 2 \cdot d - 2$ nodes in layer L_1 to L_{d-1}. This gives us totally $2d > n$ nodes, which is a contradiction. Therefore, there must exist some layer L_i of the BFS-tree where there is only one node.

Claim: Let the node on L_i mentioned above v, then deleting v will result in destruction of all path from s to t.

Proof of the claim: Since all nodes in layers L_0 to L_{i-1} must either be connect with another node in layers L_0 to L_{i-1} or connect with the v in L_i (since if (x, y) is an edge of G, then x and y must result in layers that are in maximum one layer apart). But since t is not on layers L_0 to L_{i-1}, so for s, which is in layer L_0, to be connected with t, some node in layers L_0 to L_{i-1} must be connected to L_i first. But since v is the only node on L_i and it has been removed, so no nodes in layers L_0 to L_{i-1} can be connected with L_i. Thus, there is no path from s to t after v is removed.

Algorithm:

- Run BFS with root s on G to obtain a BFS tree.
- Initialize a queue. First push s to the queue and count the number of its children.
- Then pop s from the queue and push all s's children to the queue. Then count the total number of children of all elements in the queue.

- Repeat this process until we obtain that the total number of children of all elements in the queue is 1
- Then go to that child and remove it

Since run BFS will take $O(n+m)$ (m is the number of edges and n is the number of nodes), then to look for the layer with only one node, we will visit each node once (when we pop the nodes from the queue) and visit each edge twice (when we count the number of children and push children to the queue), so that will take $O(n+m)$ as well. Overall, this algorithm will run with $O(n+m)$.

Question 5 (Exercise 12, Page 112):

The algorithm goes as follows:

- For each person P_i, create node P_{i_b} and P_{i_d}, where the first one represents the event of birth of P_i and the second represents the death of P_i.
- For all i, add a directed edge from P_{i_b} to P_{i_d}. i.e. add an edge (P_{i_b}, P_{i_d}).
- Then go through the fact list.
  - If we encounter the event of the form "For some i and j, person P_i died before person P_j was born.", draw a directed edge from P_{i_d} to P_{j_b}. i.e. add edge (P_{i_d}, P_{j_b}).
  - If we encounter the event of the form "For some i and j, the life spans of P_i and P_j overlapped at least partially.", draw a directed edge from P_{i_b} to P_{j_d}; also draw a directed edge from P_{j_b} to P_{i_d}. i.e. add edge (P_{i_b}, P_{j_d}) and (P_{j_b}, P_{i_d}).

This will then give us a directed graph containing all information of the birth and death of all people in the set. Denote the graph G.

Then, run topological sort on this graph. Suppose that all information is consistent, then topological sorting will produce an ordering of all nodes of G such that they are consistent with the information collected. Since there will be $2n$ nodes in the G, we take $\frac{200}{2n}$ to be the difference of the time between two consecutively ordered nodes. We then calculate the date of all events in the sorted list of nodes by adding $0, \frac{200}{2n}, \frac{200}{n}, \frac{400}{n} \cdots$ to the beginning year the 200-year interval.

Suppose then that the information is not internally consistent. Then when we go down the time line, we must have encountered some event that has already happened in

the past. In this case, we will end up with a cycle in G and topological sort will stop before expiring all the nodes. Then, if the topological sort fails, then we have information that is not internally consistent.

Before we analyze the algorithm, assume that e is the number of edges of G and n is the number of people in the set. Thus, construct P_{i_b} to P_{i_d} for all i will take $2n$ steps and adding all the edges will take e steps. Then construction of the graph will run with order $O(n+e)$. Since the topological sort will also run at $O(n+e)$ time. Thus, the overall algorithm will run with order $O(n+e)$ time.

Question 6:

Introduction of idea:

The goal of this question is to find if there is a way from the start cell to the end. We regard the maze as a graph with each cell in the grid representing a node and each edge connecting a pair of touching cells where neither is in the range of alarm. Given this graph representation of the maze, we can run a DFS rooted on the start cell to see if it is connected with the end.

Algorithm:

This algorithm uses a stack to do DFS search on a "imaginary" graph of the maze. (The graph is "imaginary" since there will be no explicit steps used to construct the graph.)

So, the DFS algorithm here will be slightly modified: instead of directly following an edge (which we haven't explicitly made), check three condition to see if there is an edge between a pair of touching cells:

1. Does not trigger an alarm
2. It is actually inside the $n \times n$ grid.

For the following explanation of the algorithm, assume a cell's location is (x, y), and returning TRUE means there is a solution, and returning FALSE means there is no solution.

Then the exact algorithm goes like this:

- Initialize an $n \times n$ matrix M and fill every place with '*'
- Initialize a stack
- Push the Start cell into it mark the Start cell (in M) 'V'
- While the stack is not empty
    - Pop the cell with location (x, y) from the top of stack
    - If it is the End cell
        - Return **TRUE**
    - If the cell $(x-1, y)$ does not trigger alarm, and is inside the $n \times n$ grid, and is not marked 'V'

        - Push cell $(x-1, y)$ to the stack and mark it as 'V' in M
    - If the cell $(x+1, y)$ does not trigger alarm, and is inside the $n \times n$ grid, and is not marked 'V'
        - Push cell $(x+1, y)$ to the stack and mark it as 'V' in M
    - If the gird $(x, y-1)$ does not trigger alarm, and is inside the $n \times n$ grid, and is not marked 'V'
        - Push cell $(x, y-1)$ to the stack and mark it as 'V' in M
    - If the gird $(x, y+1)$ does not trigger alarm, and is inside the $n \times n$ grid, and is not marked 'V'
        - Push cell $(x, y+1)$ to the stack and mark it as 'V' in M
- Endwhile
- Return **FALSE**

This algorithm will run with time complexity $O(n^2)$. The reasons are:

1. Initialize M will take n^2 steps
2. Initialize a stack will run with $O(1)$
3. Mark the Start cell takes 1 step
4. The while loop will run at $O(n^2)$ time nodes. This is because we are going through all of the nodes in the $n \times n$ grid and inside each loop, we are doing a constant number of operations

Therefore, the overall complexity is $O(n^2)$.