CS 180 Homework 2

Nevin Liang, Nhat Ho, Ty Koslowski

January 19, 2021

Problem 1

You are given an undirected tree T = (V;E). In this problem we will find the diameter of the tree in 3 different ways.

a)

1. Using recursive algorithm to find the diameter of the graph, based on the leaf pruning method discussed in lecture.

Answer:

With the leaf pruning, firstly, we use the has_leaves function to find all the leaf nodes in the original graph. In the main get_diameter function, we continue working on the adjacent nodes of each leaf node. Then remove the leaf node out of the list of vertices and keep recursive function to remove until the there is no leaf nodes and find out the diameter of the graph.

For both has_leaves and get_diameter function, we always need to go through all the adjacent nodes of graph, so the time complexity is O(N), N is the number of edges of original graph.

```
vars:
    adj_list: Vertex -> List of Vertices
funcs:
    func has_leaves: -> list of vertices
        ret_val = []
        for vertex v in adj_list:
            if size of adj_list[v] is 1:
                add v to ret_val
        return ret_val
```

```
func get_diameter: -> int diameter
    leaves = has_leaves()
    if size of leaves is 0:
        return 0
    else:
        for vertex v in leaves:
            parent = adj_list[v]
            remove v from adj_list
            remove v from adj_list[parent]
        if size of adj_list is 2:
            return 1 + get_diameter()
        else:
            return 2 + get_diameter()
```

2. Using the iterative algorithm to find the diameter of the graph, based on the recursive version.

Answer:

Instead of using recursive function to find out the diameter, the iterative algorithm tries to check the size of adjacent list. If size is equal 1, it means there is only 1 vertex left, and we can get the value of diameter. However, if the size is equal 2 ⇔ there are 2 vertices left, and we have to add 1 to the total diameter.

Because we also need to loop all the adjacent node, the time complexity is still O(N), with N = |E|.

funcs:

```
func get_diameter: -> int diameter
    diameter = 0
    while size of adj_list > 2:
        for vertex v in adj_list:
            if size of adj_list[v] is 1:
                parent = adj_list[v][0]
            remove v from adj_list[parent]
            remove v from adj_list
        diameter += 2
```

```
if size of adj_list = 1:
        return diameter
else if size of adj_list = 2:
        return diameter + 1
```

b) Show we can find the diameter of the graph in O(|E|) time by running BFS twice.

All we must do is run BFS once to find a single leaf node. Then if we run BFS again you can flood fill the graph from one starting leaf node to all other leaf nodes. take the maximum distance and that is the diameter of the graph.

c) Find the diameter of the graph by rooting the tree, and a recursive call from the root asking for the diameter of each subtree hanging from each of its child. funcs:

Answer:

We find the diameter based on the subtree hanging from each of its children, so we need to use the recursive function to go deeper to find the diameter for each subtree created by all current's adjacent nodes.

Because for each adjacent node, we do not want to go back to its parent. So we also need to check whether the adjacent node has been visited or not. If the adjacent node is visited, it must be a parent node of the current, so we do not do anything and skip to check the other adjacent node by continuing using the recursive function.

For each subtree hanging from the current node, the diameter should be the maximum value of all its adjacent nodes. Keep doing the same process until all nodes are visited, we have done the algorithm and can get the diameter.

Because in this algorithm, we loop all adjacent nodes based on the edges incident to current node ➔ this checking is at most O(m), with m =|E|. Finally, time complexity for this algorithm is O(m).

```
func recursive(vertex v, &visited) -> int diameter
        diameter = 0
        if(visited[v] == true) return 0
        visited[v] = true
        for vertex v2 in adj_list[v]:
                diameter = max(diameter, recurse(v2))
        return diameter + 1;

func get_diameter: -> int diameter
        vertex v = 0 # pick a random root
        bool visited[N] = false;
        return recurse(v, visited)
```

Problem 2

a.

Given an undirected graph G = (V;E) and a subgraph T = (V',E') where V'= V; E' E, that is a tree. Design an algorithm to find whether T could have been the output of running DFS on the original graph. from some starting vertex.

b. As the above. But now, design an algorithm to find whether this subgraph T could have been the output of running BFS on the original graph.

Answer:

Because we do not know exactly the starting point when we apply the DFS or BFS, so to check if T could have been the output of running DFS/BFS, we need to find all the trees that can be created by running these algorithms for each node in the original graph. If one of these trees is equal to the T, we can conclude that T can be formed by the original graph and vice versa.

Applying the same idea for both DFS and BFS but the data structure. Specifically, using the stack for DFS and queue for BFS. For each node, we tried to check all its adjacent nodes whether it is visited or not.

If the adjacent node is visited, we have already processed it before and do not care about this case. However, if the adjacent node has not been visited, it means we have found the new edge for the tree, then add it into the list of edges for the tree and need to push this node into the stack or queue for the next checking.

Whenever all adjacent nodes of the current node are visited, that means we have finished finding all edges created from the current node; just pop it out stack or queue. Continue the process till no nodes exist in the stack or queue; we have a tree used to compare with the original tree T.

Generally, we have to check all nodes in original graph, so we need O(n) for checking the node with n is number vertices in original graph.
For each checking, we go through all its adjacent node based on the edges incident to current node ➔ this checking is at most O(m), with m =|E|. Finally, time complexity for this algorithm is O(nm) for both DFS and BFS.

vars:

vector<pair<int, int>> original.

gets_orig_edges(): -> the function goes through the given T to get all its edges

dfs(root):

```
bool visited[N] = false;
vector<pair<int, int>> edges;
stack<int> q;
q.push(root);
visited[root] = true
while (!q.empty()) {
    int front = q.pop();
    for (int nbor : adj[front]) {
        if visited[nbor] == true
            continue;
        visited[nbor] = true;
        edges.add(make_pair(front, nbor));
        q.push(nbor);
    }
}
```

bfs(root):

```
bool visited[N] = false;
vector<pair<int, int>> edges;
```

```
        queue<int> q;
        q.push(root);
        visited[root] = true
        while (!q.empty()) {
                int front = q.pop();
                for (int nbor : adj[front]) {
                        if visited[nbor] == true
                                continue
                        visited[nbor] = true;
                        edges.add(make_pair(front, nbor));
                        q.push(nbor);
                }
        }
```

// Main function for DFS

```
main():
        get_orig_edges();
        loop thru vertices v:
                dfs(v);
                for (pair in edges) {
                        if pair is not in  orig:
                                return false;
                }
                return true.
```

// Main function for BFS

```
main():
        get_orig_edges();
        loop thru vertices v:
                bfs(v);
                for (pair in edges) {
                        if pair is not in  orig:
                                return false;
                }
                return true
```

Problem 3

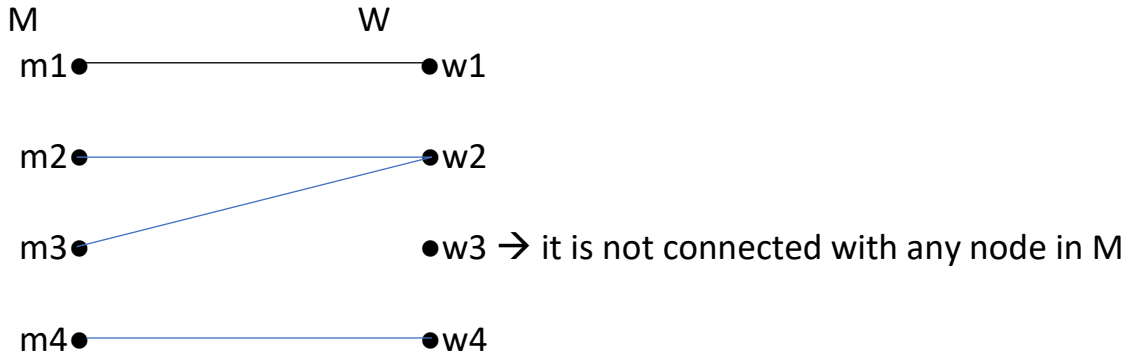
1. If in G each node in W has degree 1 or more, show that the graph is a perfect match.

For each woman W, the degree of w is >= 1. In addition, no two women can share an edge with the same man. Thus, the degree of a subset of women with size K must be greater than or equal to K. Note that this is true for all subsets of women. Therefore, there is a perfect matching by Hall's theorem.

2. If the graph is not a perfect match, we want to find the largest subset L from {1, 2, …, n} such that the subgraph induced by the Men in L and the Women in L constitutes a perfect match in the induced subgraph. Your algorithm should be linear time.

Because, in the original graph, each node in M has degree 1, and whole graph contains exactly n edges. Therefore, it the graph is not a perfect match, there is at least one node in W is not connected with another node in M. So, to find the subgraph, which is perfect match, we need to loop through all the women. If there is an edge coming out of it (or more), pick a man at random and match the woman with the man, and if there is no edge coming out of the woman, then skip her. Finally, when we reach the bottom, all the matches we have before is the maximum subset we need to find.

For example:



→ L = {1, 2, 4}

Because the algorithm will loop all node in W from 1 to N, so the time complexity is O(N) – linear time.