COMS 311: Homework 3
Due: Oct 12, 11:59pm
Total Points: 100

Late submission policy. If you submit by Oct 13, 11:59PM, there will be 20% penalty. That is, if your score is x points, then your final score for this homework after the penalty will be $0.8 \times x$. Submission after Oct 13, 11:59PM will not be graded without explicit permission from the instructors.

Submission format. Your submission should be in pdf format. Name your submission file: <Your-net-id>-311-hw3.pdf. For instance, if your netid is asterix, then your submission file will be named asterix-311-hw3.pdf.

If you are planning to write your solution and scan+upload, please make sure the submission is readable (sometimes scanned versions are very difficult to read - poor scan quality to blame).

If you plan to snap pictures of your work and upload, please make sure the generated pdf is readable - in most cases, such submissions are difficult to read and more importantly, you may end up submitting parts of your work.

If you would like to type your work, then one of the options you can explore is latex (Overleaf).

Rules for Algorithm Design Problems.

1. Unless otherwise mentioned, for all algorithm design problems, part of the grade depends on runtime. Better runtime will lead to higher grade.
2. Unless otherwise mentioned, for all algorithm design problems, you are required to write some justification of why your algorithm correctly addresses the given problem.
3. You will write pseudo-code for your algorithm. It should not be some code in a specific programming language (e.g., Java, C/C++, Python).
4. You can use any operation already covered in class-lectures in your solution without explicitly writing the algorithm. However, you need to use them appropriately (i.e., indicate the inputs to and outputs from these algorithms). For instance, you can use heapifyUp with input as the index of a heap element and you do not need to write the code for heapifyUp.

Learning outcomes.

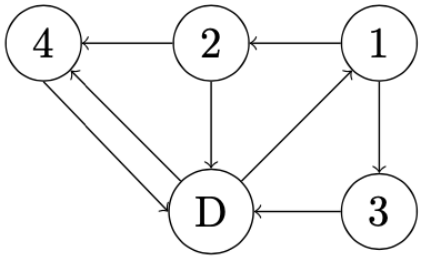
1. Runtime analysis
2. Graph Algorithms

Problems 1–4 are 25pts each. Extra Credit Problem is 20pt.

1. In the indomitable gaulish village, druid Getafix had to make several trips to all village houses everyday. There are walkways between houses in this village such that Getafix can take several walkways to any house and also from any house Getafix can take several walkways to come back to his house. All walkways in this village are *directed*, i.e., one may not be able to take the same walkway to go back and forth between two houses. Everyday Getafix has to do the following:

   for every house h
   Getafix walks from his house to h
   Getafix delivers magic potion to the house members
   Getafix walks from h to his own house

   Write an algorithm to determine the shortest distance Getafix walks everyday. You can assume that every walkway is of length *L*. The input to your algorithm is the village map with houses and walkways between them. Here is an example scenario:



   In the above, druid Getafix (his house is denoted by *D*) would walk

$$\underbrace{(L+(L+L))}_{\text{house 1}}+\underbrace{(L+L+(L))}_{\text{house 2}}+\underbrace{(L+L+(L))}_{\text{house 3}}+\underbrace{(L+(L))}_{\text{house 4}}$$

```
String DeterminSortestDistance(root){
        //put nodes in to 2d list to trach which node is in which layer
        To maintain the node in each layer Li, create L[i] for each i=0,1,2…
        totalDis = 0;
        root.visited= true;
        for all other vertex , visited = false;
        Initialize L[0] with root;
        layerCounter i = 0;
        while(!L[i].isEmpty()){
                Initialise emptyList L[i+1];
                for each node v ∈ L[i]{
                        for each vertex u adjacent to v {
                                If(!u.visited){
                                        u.visited = true;
                                        totalDis+= i+1;
                                        L[i+1].add(u);
                                }
                        }
                }
                i++;
        }
        For all of house, v{
                totalDis += findDistanceFrom(v);
        }
        return totalDis + “L”;
}

int findDistanceFrom(root){
        totalDis = 0;
        root.visited= true;
        for all other vertex , visited = false;
        Initialize L[0] with root;
        layerCounter i = 0;
        While(!L[i].isEmpty()){
                Initialise emptyList L[i+1];
                for each node v ∈ L[i]{
                        for each vertex u adjacent to v {
                                If(!u.visited){
                                        If(u is D){
                                                return i + 1;
                                        }
                                        u.visited = true;
                                        L[i+1].add(u);
                                }
                        }
                }
                i++;
        }
}
```

Run time:
DeterminSortestDistance(root) compute bfs to find the shortest path from root node(D) to each house and track the total distance, and make a list of houses. After that for each house, call findDistancefrom(v) to find the shortest path to come back to D and sum up to the total distance.
∴ O (|V|(|V|+|E|))

2. Given a directed graph $G = (V, E)$, a vertex is $k(< |V|)$ strong if k vertices can reach the vertex (excluding itself). Write an algorithm to verify the existence of a $|V| - 1$ strong vertex in a given graph. The input to your algorithm is a directed graph.

```
Boolean kStrongVertexExist(G){
        Initialize list as |V| length array of integer with 0;
        For each vertex v in G{
                list = BFS(v,list);
        }
        for(int i=0;i<|V|;i++){
                if(list[i]==|V|-1){
                        return true;
                }
        }
        Return false;
}
Int[] BFS(v,list){
        v.visited= true;
        for all other vertex , visited = false;
        Initialize queue with v;
        while(!queue.isEmpty()){
                cur = queue.poll();
                for each vertex u adjacent to cur{
                        If(!u.visited){
                                u.visited = true;
                                //u is visible from v
                                list[u]++;
                                queue.add(u);
                        }
                }
        }
        Return list;
}
```

Run time:
This algorithm compute BFS from each node to find whether |V|-1 strong vertex exist in the given graph
∴ O (|V|(|V|+|E|))

3. Given a connected undirected graph, write an algorithm for verifying the existence of simple cycle in the graph. A simple cycle is defined as a sequence of vertices $v_1, \ldots, v_k$, where all vertices are distinct and $\forall i \in [1, k-1], (v_i, v_{i+1}) \in E$ and $(v_k, v_1) \in E$. The input to your algorithm is a connected undirected graph.

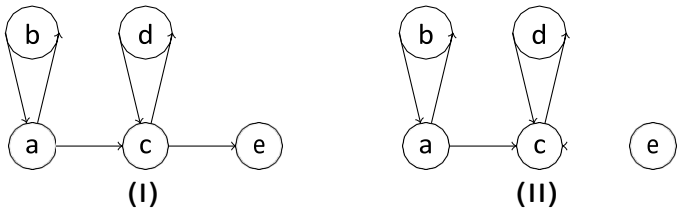
```
List cycle;

Int head, node;

Boolean SimpleCycleExist(G){
        if(detectCycle(G.Node.parent, G.Node.child)){
                simpleCycle(head, tail);
                if (!cycle.isEmpty()){
                        return true;
                }
        }
        else{
                return false;
        }
}

boolean detectCycle(v, node){
        for each vertex u adjacent to node{
                if(!u.visited){
                        if(detectCycle(u,node){
                                return true;
                        }
                }
                //check back edge
                Else if(u!=v){
                        head = u;
                        tail = node;
                        return true;
                }
        }
        Return false;
}

void isSimpleCycle(int u, int v){
        int[] parents;
        Initialize queue with u
        boolean flag = true;
        while (!queue.empty()) {
                int cur = queue.poll();
                cur.visited = true;
                for each vertex u adjacent to cur{
                        if(cur==head){
                                if(u==tail){
                                        continue;
                                }
                        }
                        If(!u.visited){
                                Parents[u] = cur;
                                If(u==tail){
                                        Flag= false;
                                        Break;
                                }
                                Queue.add(u)
```

```
                        u.visited = true;
                }
        }
        If(!flag){
                Break;
        }
        Cycle.add(u);
        Int temp = v;
        While(temp!=u){
                Cycle.add(temp);
                Temp = parents[temp];
        }
    }
}
```

Runtime:
Firstly, find a cycle by simply calling dfs. If the graph contains a cycle, store the head and tail. Then compute bfs between head and tail to find the shortest path which is the simple cycle.
∴The run time is run time of dfs + bfs
∴ O (|V|+|E|)

4. Given a directed graph $G = (V, E)$, we say that it is at least one-way connected if for every $u_1, u_2 \in V$, there exists a path from u_1 to u_2 or a path from u_2 to u_1. Write an algorithm to determine whether G is at least one-way connected. The input to your algorithm is a directed graph.

   Here are couple of scenarios. Graph I is at least one-way connected and Graph II is not at least one-way connected (in Graph II for pair of vertices *a* and *e*, there is no path from *a* to *e* and from *e* to *a*).




```
Boolean kStrongVertexExist(G){
        Initialize |V|*|V| array of Boolean map with false;
        For each vertex v in G{
                map = BFS(v,map);
        }
        for(int i=0;i<|V|;i++){
                int j=i+1;j<|V|;j++){
                        if(!map[i][j]&&!map[j][i]){
                                return false;
                        }
                }
        }
        Return true;
}
Int[] BFS(v,map){
        v.visited= true;
        for all other vertex , visited = false;
        Initialize queue with v;
        while(!queue.isEmpty()){
                cur = queue.poll();
                for each adjacent of cur, u {
                        If(!u.visited){
                                u.visited = true;
                                //u is visible from v
                                map[v.index][u.index]=true;
                                queue.add(u);
                        }
                }
        }
        Return map;
}
```

Run time:
This algorithm compute BFS from each node to find the visible nodes and after that check if it is at least one way connected or not.
$\therefore O(|V|(|V|+|E|))$

5. **Extra Credit** Given an undirected graph $G = (V, E)$, we say that it is k-colorable if each vertex v can be colored with one of the k colors ($c(v)$ is the color of vertex v) such that $\forall (u_1, u_2) \in E : c(u_1) \neq c(u_2)$. Write an algorithm to verify that a given graph is 2-colorable and if it is, then determine a coloring scheme (i.e., determine the color for each vertex). The input to your algorithm is an undirected graph and 2 colors.

```
String [] color = String[|V|];
String[] twoColor(G,color1, color2){
	Set color[1 to |V|] null;
	For each vertex v, 1 to |V|{
		If(color[v]==null){
			If(!dfs(u,color1,color2)){
				Return false;
			}
		}
	}
	Return color;
}
dfs(v,color1,color2){
	color[v]=color1;
	for each vertex u adjacent to v{
		if(color[u]==color1){
			return false;
		}
		Else if(color[u]==null&&!dfs(u,color2,color1)){
			Return false;
		}
		Else{
			Return true;
		}
	}
}
```

Runtime:
This algorithm goes through each edge and vertex at most once.
$\therefore O(|V|+|E|)$