HW3 - Coms 311

Joseph Schmidt

1.

```
//If a row i has a 1 in it then it can't be a trustworthy person so you move onto next person
//If you go down through the entire column and they have all 0s then potential to be Trustworthy
//Person
//We know we don't have to continue are search because if the rest of column is 0s then
//That means that none of the future ones are trustworthy people because otherwise there
//Would be a 1 in this column relating to the future row as a trustworthy person has everyone
//else trusting them
//Once we have our candidate we look in the row and column to see if they have the respective
//1s and 0s. We jump over T[i][i] in the second loop as it should be a 0 not 1

int AlgorithmOne(Matrix T)
{
        i = 1
        j = 1

        while i < n and j < n do
                if T[i][j] == 1 then
                        i++
                else
                        j++

        for k = 1 to n do
                if T[i][k] != 0 then
                        return -1

        for k = 1 to n do
                if k != i and T[k][i] != 1
                        return -1

        return i
}

//It goes across each of the nodes so it is O(n)
```

2.

```
//Performing Bfs on each vertex, if it is already explored no need. While doing this we are
//Keeping a set of the number of islands and the starting node for that respective set
//We continue to go through performing bfs and if we find a set of islands that has a higher
```

```
//count than our max set of islands we set the new start vertex and the new set of max islands

global: set setMaxBridges, set setBridges,

bfsExploration( set L, matrix B) )
{
        StartingIsland = random start island from set L
        setMaxBridges = ∅

        for all l ∈ L do
                l.explored = false

        for all l ∈ L do
                if l.explored == false then
                        if(setMaxBridges.count > setBridges.count)
                                setMaxBridges = setBridges
                                startingIsland = l
        return G = (StartingIsland, maxBridges)

}

bfs(island l)
{
        add l to queue Q
        setBridges.clear
        l.explored = true
        while Q is not empty do
                u = dequeue(Q)
                for all w ∈ Neighbor(u) do
                        if w.explored == false then
                                add w to queue Q
                                w.explored = true
                                setBridges.add(w)
}

//O(|V| + |E|) runtime, this is because bfs is only called on non visited nodes

3.

//Idea, create a graph where you combine coreqs into one node adding all prereqs of classes
//inside of node as directed edges to the node
//You then go through and have to add the prereqs for the other classes and make sure that
//If one of the prereqs was a coreq class it has to map to the new group node
```

```
//Then execute modified DFS to check if you come across a cycle then return false else return
//true

Algo3(set C, set Pre, set of sets S)
{
	G = (V, E)
	for each s in S
		V.add(Bi)		//Where Bi represents the ith set of coreqs
		for each p in s		//for all class in the set of coreqs add prereqs to node
			if(p.added = false)
				p.parent = i		//Same as bi number
				p.added = true
				Et = Pre(p)
				for each e in Et
					E.add(e, Bi)	//Add edge from prereq p to new Bi
							//combined node
	for each c in C
		if(c.added = false)
			c.added = true
			Et = Pre(c)
			for each e in Et
				if(c.parent != null)	//Class is a coreq add to the grouped node
					E.add(e, Bi where i is c.parent value)
				else
					E.add(e,c)

	return DFS(G, first c in C);
}

//Checking if there is a cycle
boolean DFS(Graph G, vertex s)
{
	s.state = discovered
	foreach(v ∈ s.neighbors)
		if(v.state == undiscovered)
			if(!DFS(G, v))
				return FALSE
		else
			return FALSE
	s.state = processed
	return true
}

//Runtime is O(V + E), V is twice the number of classes and E is the number of prereqs
```

4.
//Assume you have valid DAG from question 3, also assuming that part of runtime is not from
//Finding the DAG, also assuming once you have DAG you can get the starting node from the
//topo sort

Using dynamic programming technique where you can start from the bottom (no incoming edges) and find the distance (going to be 0 for no incoming edges) and work your way up, always taking the max length from the predecessor vertices and adding + 1 for the next edge you took to get to your current vertex
.

```
//Assuming we are keeping L(v) in some global set
SPDAG(DAG(V,E), s)
{

        foreach(v ∈ V)
                L(v) = ∞
        L(s) = 0
        foreach(v ∈ V - {s} in topological order)
                if(Pre(v) == ∅)                         //No incoming edges
                        L(v) = 0
                else
                        //v' denotes the vertex with the max length out of predecessors of v
                        L(v) = max {L(v')} + 1 where v' ∈ V s.t. (v', v) ∈ E
        return L(the v last in topological order)
}
```

The runtime is going to be O(|V|(|V| + |E|) this is because for every vertex you are going through all the past vertices to find the max length