COMS 311: Homework 4
Due: Oct 18, 11:59pm
Total Points: 30

Late submission policy. If you submit by Oct 19, 11:59PM (and after the specified deadline), there will be 10% penalty. That is, if your score is x points, then your final score for this homework after the penalty will be $0.9 \times x$. Similarly, if you submit by Oct 20, 11:59PM (after Oct 19, 11:59PM), there will be 20% penalty.

Submission after Oct 20, 11:59PM will not be graded without (prior) explicit permission from the instructors.

Submission format. Your submission file should be in pdf format. Name your submission file: `<Your-net-id>-311-hw4.pdf`. For instance, if your netid is `asterix`, then your submission file will be named `asterix-311-hw4.pdf`.

You are required to type-set your solution. You can use word, latex and any other type-setting tool to type your solution; you need to make sure you export/save it in pdf before submission.

For questions where you are asked to write an algorithm, you are required to write pseudocode. Do not write code in any specific language such as Java, Python, C/C++, etc. You are also required to present the runtime of your algorithm. Part of the grade will depend on the efficiency of your algorithm.

Algorithm Design Problems

- **When asked to design/write an algorithm, write pseudo code, not code of any specific programming language and not library operations of any specific programming language library.**
- For any algorithm design problem, present some justification for correctness of the algorithm you are presenting. For instance, checking whether there exists a subarray whose sum is 0 depends on computing prefix sums. This is because if there are two prefix sums that are equal then there exists a subarray whose sum is 0.
- For any algorithm design problem, you will present the runtime of your algorithm. The derivation does not need to be detailed but sufficient for explaining the final runtime.
- For all algorithm design problems, part of the grade depends on its runtime efficiency.

Learning outcomes.

1. Graph Algorithms
2. Application of Basic Graph Algorithms

---

1. In a dystopian future, Agent Smith has been able to set up a communication network where he controls a computing node, *replicator* that can reach every other node in one hop while none of the other nodes can reach it. Trinity has obtained the information of this communication network and now she needs to quickly identify the replicator. Write an efficient algorithm that Trinity can use to realize her objective.

```
ReplicatorFinder(G){
    for(all v in V){
        v.visited = false
    }
    for(all v in V){
        if(v.visited == false){
            replicator = v
            DFSFrom(v)
        }
    }
    return replicator
}

DFSFrom(v){
    v.visited = true
    for(all u in N(v)){
        if(u.visited == false){
            DFSFrom(u)
        }
    }
}
```

Justification: We know that the vertex with the largest end time will not depend on any other vertices. Therefore, if we update the replicator to be the current vertex we are calling DFSFrom(), then that final vertex before all vertices become visited will be our replicator since no other vertices can reach it.

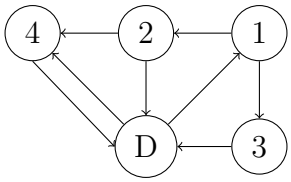
Runtime: O(V+E) $\rightarrow$ This is due to the fact that our algorithm's runtime solely depends on DFSFrom(), which has a runtime of O(V+E).

2. 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

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

For instance, Getafix can go from his house to house 1 in one step (of length L) and from house 1, he can come back to his house either via house 2 or via house 3 (both will be of length 2L).

```
ShortestDailyDistance(G){
    distance = 0
    for(all v in V){
        for(all u in V){
            u.visited = false
        }
        BFSFrom(v)
        if(v == D){
            for(all w in V){
                distance += w.layer - D.layer
            }
        }
        else{
            distance += D.layer - v.layer
        }
    }
    return (distance * L)
}

BFSFrom(v){
```

```
    v.visitied = true
    v.layer = 0
    add v to queue Q
    while(Q != empty){
        u = dequeue(Q)
        for(all w in N(u)){
            if(w.visited == false){
                w.visited = true
                w.layer = u.layer + 1
                add w to Q
            }
        }
    }
}
```

Justification: Running BFSFrom(D) gives us the shortest paths to each house by finding the difference between the layer values. Since Getafix has to return back in the shortest path, running BFSFrom() for each of the other vertices gives us the shortest paths from those vertices and we can find the difference between D's layer and the current vertices layer.

Runtime: O(V*(V+E)) → This is due to the fact that our algorithm's runtime depends on BFSFrom(), which has a runtime of O(V+E), which runs V times, giving us O(V*(V+E)).

3. In a city, the road network connects intersections; in other words, if there is a road between two intersections, then one can go from one intersection to the connected intersection following the direction of the road (some roads can be one-way). Furthermore, the city is well-planned and hence, each road has exactly the same length.

   The city planning commission is trying to determine whether electric buses will be a viable option for public transportation in the city. Each bus is routed to travel along the shortest path from the starting intersection to its final destination intersection. To reduce the range anxiety of the electric bus drivers, the commission wants to determine the longest such bus route in the city. Write an algorithm that outputs the length of the longest bus route.

```
LongestBusRoute(G){
    longest_route = 0
    for(all v in V){
        for(all u in V){
            u.visited = false
            u.distance = -1
        }
```

```
        // If the vertex is the start of a bus route
        if(v.busNumberStart){
            BFS_distance = BFSFrom(v)
            if(BFS_distance > longest_route){
                longest_route = BFS_distance
            }
        }
    }
    return longest_route
}

BFSFrom(v){
    v.layer = 0
    add v to queue Q
    while(Q != empty){
        u = dequeue(Q)
        for(all w in N(u)){
            if(w.visited == false){
                w.visited = true
                w.layer = u.layer + 1
                add w to Q
            }
            // For the bus number that starts at node v...
            // -> Is the current node the end?
            if(v.busNumberStart == w.busNumberEnd){
                // The difference in the layers is the shortest distance
                return (w.layer - v.layer)
            }
        }
    }
}
```

Justification: Using BFSFrom() for each vertex finds the shortest path for each bus. Using that knowledge, we just check if the shortest path that BFSFrom() returns is larger than the current longest bus route, if so, update it.

Runtime: O(V*(V+E)) → This is due to the fact that our algorithm's runtime depends on BFSFrom(), which has a runtime of O(V+E), which runs a maximum of V times, giving us O(V*(V+E)).

4. **Extra Credit** You are helping a group of ethnographers analyze some oral history data. The ethnographers interviewed members of a village and learned about the birth and death a set of n people: $P_1, P_2, \ldots, P_n$. The information is categorized as follows:

- For some i and j, person P_i died before person P_j was born
- For some i and j, the life spans of P_i and P_j overlapped at least partially.

As these are word of mouth information, it becomes important to validate the data. You are tasked with writing an algorithm which takes as input the learned information (about relative ordering of birth and death of n people) and verify its validity.

```
Validation(information){
    graph G = (V, E) = MakeGraph(information)
    for(all v in V){
        if(v.visited == false){
            add v to stack S
            while(S != empty){
                u = pop(S)
                if(u.visited == false){
                    u.visited = true
                    for(all w in N(u)){
                        if(w.visited == false){
                            add w to S
                        }
                        // Already visited = cycle
                        else{
                            return invalid
                        }
                    }
                }
            }
        }
    }
    return valid
}

MakeGraph(information){
    graph G = (V, E)
    for(all P){
        // Establish birth and death nodes.
        add vertex P_ib and P_id to G
        // Person has to be born before they die.
        add edge from P_ib to P_id to G
        // May as well instantiate visited nodes here.
        P_ib.visited = false
        P_id.visited = false
        if(P_i died before P_j was born){
```

```
            // Person i has to die before person j is born.
            add edge from P_id to P_jb to G
        }
        if(P_i and P_j overlapped at least partially){
            // Person i has to born before person j dies (and vice versa).
            add edge from P_ib to P_jd to G
            add edge from P_jb to P_id to G
        }
    }
    return G
}
```

Justification: Since we are trying to determine the validity of the order, we want to use topological ordering. Therefore, since we just have information, I want to make a graph and run DFS() on it. We build the graph by having a having the directed arrows point from the node that occurs first to the node that happens after it. When running DFS(), if we visit a node that's already been visited before, then we have a cycle and that breaks a property of the topological ordering. Therefore, the information is invalid.

Runtime: O(V+E) $\rightarrow$ This is due to the fact that our algorithm's runtime depends on DFSFrom(), which has a runtime of O(V+E). Also, the construction of the graph takes O(V+E) since we have to insert |V| vertices and |E| edges.