Hannah Kwon
705182275
Professor Hsieh

Homework #2

Computer Science 180 - Introduction to Algorithms and Complexity

Question 1

```
Data
V : Number of nodes in the graph
G : The graph stored as an adjacency list
S : The index of the source node
D : The index of the destination node
Returns the number of shortest paths between S and D

visited[n] #false
distance[n] #infinity
paths[] #0
queue.add(S) #adds source node into the queue
distance[S] #0
paths[S] #1

while queue not empty() :
  current = queue.front()
  queue.pop()

  for each child in current : #for each vertex in the adjacency list
    if visited[child] == False : #if not discovered
      queue.add(child) #add to queue
      visited[child] == True

    if distance[child] > distance[current] + 1 :
    #check if distance of prev layer + 1 < distance of current vertex
    #since BFS works layer by layer
        distance[child] = distance[current] + 1 #if yes, shorter path found so replace
        paths[child] = paths[current] #and set the number of paths of current vertex
    else if distace[child] == distance[current] + 1 :
    #if yes, another short path found
        paths[child] = paths[child] + paths[current] #so update the paths

return paths[D]
```

This modified BFS function keeps 3 arrays of size n (number of nodes) : visited, distance, paths as above.

The distance array stores the smallest distance from the source node to the other nodes and tracks nodes using BFS. If a node is not explored the distance is set to infinity and if it starts visiting other nodes, it is set to the distance of its previous node + 1 since BFS explores layer by layer. The number of paths are also equivalent to the previous node paths since it only reached that node until then. If the node is already visited, then the distance and the number of paths are compared. Depending on the distance, the distance of the shortest path is either replaced or remained and the number of the paths are updated.

Running the original BFS algorithm takes O(m) time and. The only added time would be O(n) which is used to set up an additional array and accessing certain elements of the array and doing a comparison which takes O(1) time. So overall, this modified BFS can be done in O(m + n). However, we can assume the graph is connected and it's mostly $m \geq n$, the overall time-complexity would not change and still be O(m).

Question 2

(a) In any DFS graph, parent[u] stores the parent of vertex u. And a vertex u can be shown as an articulation point with the following two conditions :

1. u is a root of DFS tree and it has at least two children
2. u is not a root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge of one of the ancestors

Using the L[v] is by finding it for the second condition. In order to check using L[v], we maintain a D[u] to store discovery time as shown in 1. and for every node u, we need to find out the vertex with minimum discovery time that can be reached from subtree rooted with u. So we use L[v] to track whether an ancestor of u has a back edge from a descendant of u to a. And whether L[v] ≥ D[u], so if this condition is true, then it means that the lowest time to reach v other than the DFS path is greater than or equal to discovery time of node u, so it means that if node u is removed, then there is no way to reach node v, hence v is a articulation point here.

(b) A recursive function that finds articulation points using DFS traversal.

```
u : the vertex to be visited next
visited [] : keeps track of visited vertices
D[] : stores discovery times of visited vertices
parent : stores parent vertices in DFS tree
ap [] : store articulation points

def DFS(self, u, visited, ap, parent, low, D):

    #recur for all the vertices adjacent to this vertex
    for v in graph[u] :
        #If v is not visited yet,
        #then make it a child of u
        #in DFS tree and recur for it
        if D[v] == -1 :
            parent[v] = u
            children += 1
            DFS(v, visited, ap, parent, low, D)

            #check if the subtree rooted with v
            #has a connection to
            #one of the ancestors of u
            L[u] = min(L[u], L[v])

            #u is an articulation point in following cases
            #(1) u is root of DFS tree and has two or more children
            if parent[u] == -1 and children >= 2 :
                ap[u] = True #u is an articualtion point


            #(2) If u is not root and low value of one of its child is more than
            #than discovery value of u
            if parent[u] != -1 and L[v] >= D[u]:
                ap[u] = True

    print out all the points u that makes ap[u] = True
```

Question 3

For this algorithm, we are constructing a directed graph G, such that if a topological ordering on the graph is found, then we have a possible ordering of events, consistent with the data. If one is not found, it must be because the graph has a cycle, it is not a topological ordering anymore and therefore the data is inconsistent.

The below algorithm explains how to create a node of each person and their degree and life and death, birth situations. For every person Pi we have two nodes : birth of P and death of P. If Pi died before person Pj was born, then the following reflects : Pi, Pj doesn't overlap. However, if Pi and Pj overlap at least partially, then the following directed graph reflects : 1. Pj was born after Pi was born and Pj died after Pi died. 2. Pj was born after P was born and Q died before P died. So create a node of an event that has an empty adjacency list and an indegree set to 0. Also, create an edge start and push it to the list and add the indegree by 1. For every p in p_i make a node of birth and death and connect the two nodes creating an edge.

In this algorithm we put p for P_i and q for P_j for simple use. If p, q doesn't overlap, it means p died before q's birth thus, creating a node of death[p] to birth[q]. And if p, q overlaps which means they coexisted for a certain period, create an edge of birth p to death of q and birth of q to death of p. If the graph does not contain a cycle then it's DAG that has a topological ordering. Given this topological ordering, we can assign a list of births and deaths that is consistent with the data, by checking the ordering to dates with the 200 year interval : dates[event] = START_DATE + (200yrs/2*n)(topological-order[event])

So the problem is asking to find a topological ordering on the graph of events : if yes, then we have a possible ordering of the events, consistent with the data. If not, it must be a graph with a cycle thus inconsistent.

```
def add-node(event) :
#initialize the node
out-adjacentlist[event] = [] #empty
indegree[event] = 0

def add-edge (start, end) :
    push(out-adjacentlist[start], end)
    indegree[end] += 1 #adds a degree

#make a node for each birth and death
#connect the nodes by an edge
for p in p_i :
    add-node(birth[p])
    add-node(death[p])
    add-edge(birth[p], death[p])

#doesn't overlap
for (p,q) in diedbefore :
    add-edge(death[p], birth[q])

#overlaps
for (p,q) in overlap:
    add-edge(birth[p], death[q])
    add-edge(birth[q], death[p])

newlist = []
#start with vertex whose in-degree is 0 and put it into the newlist
for p in p_i :
    if indegree[birth[p]] = 0 #checking topological order
      push(newlist, birth[p])

in200yrs = 200 / 2*|{P_i}|)
order = 0
```

```
while newlist is not empty :
    event = pop(newlist)
    order += 1

    #checking when the event started and how long it lasted
    dates[event] = START_DATE + order * in200yrs

    for newevent in newlist :
        indegree[newevent] -= 1
        if indegree[newevent] == 0:
            push(newlist, newevent)

if order < 2 * |{Pi}| :
    return Inconsistent data #has a cycle, not a DAG
else :
    return consistent data , dates #have ordering of events
```

Question 4

This is a problem using greedy algorithms. We have to eventually shoot all the flying saucers, so for each flying saucer there should be a shoot between interval[0] and interval[1] (say interval is a List[int] inside List[List[int]]).
In order to shoot the flying saucers the most, we should shoot as to the right as possible, because it is sorted, it gives a better chance to take down more flying saucers. Therefore, the position should always be interval[i][1].

Have a list of lists with the starting time and end time for each flying saucers.

Count valid intervals we need, and skip overlapping intervals return the shoots needed to destroy all the flying saucers.

1. sort the flying saucers by ending time Ri.
2. set shots = 0, end = negative infinity number.
3. Initialize an end variable which ends the first, end = points[0][1]
4. for interval in points :
    1. If interval[0] starts after end : #interval[0] > end
        i. Add shots by 1
        ii. Set end = interval[1] #interval[1] = Ri
5. return shots

This algorithm has an O(nlogn) time complexity since it's sorted first which is an O(nlogn) and then you iterate through the for loop which is O(n). Thus O(nlogn) + O(n) would be just an O(nlogn) algorithm.

This algorithm can be proved by contradiction. For the example in figure 2, if we use the above algorithm, it returns a shot of 3. However, if we use 2 shots, smaller than the 3, only 5 flying saucers are destroyed the max which means it can't destroy all the flying saucers. Thus, the above algorithm is optimal to find the minimum number of shots.