CS 180 HW 2

Narek Daduryan

November 1, 2021

1

For an undirected and unweighted graph, the BFS algorithm will output the shortest path from source (node s) to other nodes. Modify the BFS algorithm to also output the number of shortest paths from s to other node.

Solution. We can modify the BFS algorithm to not stop at the first occurrence of the target node, and instead run until all nodes in this level are exhausted.

```
BFS(start, target):
    discovered[u] = 0;
    discovered[start] = 1;
    i = 0;
    levels[i] = {start};
    while (!levels[i].empty) {
        levels[i+1] = {};
        for each node n in levels[i] {
            for each edge incident to n: (n,m) {
                if discovered[m] == 0 {
                    discovered[m] = discovered[n];
                    levels[i+1].add(m);
                } else {
                    if (levels[i+1].contains(m)) {
                        discovered[m] += discovered[n];
                    }
                }
            }
        }
        i++;
    }
    return discovered;
```

In this algorithm, we maintain a collection for each level of the graph, where the level corresponds to the distance from the start node.

For any node u whose level is l, a shortest path from s to u must go through level $l-1$. For any shortest path, the 2nd-to-last node in the path must be in level $l-1$. The number of paths from a node u in level l to a node v in level $l+1$ is the number of incoming edges into node v from level l. Thus, we can use a simple BFS which keeps track of the nodes in each level as it iterates. This BFS will iterate one level at a time; iterating through every node in level 1 before going on to level 2. Thus, by populating all the shortest paths to level 1, we can be rest assured that there are no other shortest paths existing in the graph to nodes in level 1, since their distance must be greater than paths in level 1. Likewise, for level 2, the number of shortest paths to a node u is simply the sum of shortest paths to nodes v in level 1 which connect to level 2 (where for each node v we have n shortest paths to v).

The time complexity of this algorithm is $O(m)$ where m is the number of edges in the graph. Notice that similar to the original algorithm (in the textbook), we will iterate over all n nodes in order to initialize our 'discovered' list. We then will iterate over each node n's incident edges (the degree of n). Thus, this step will take $O(\sum_{v \in V} degree(v)) = O(2m) = O(m)$. The only additional code in this implementation, compared to an original BFS, is the use of an integer for the discovered, rather than a boolean, which does not add time complexity. Additionally, we perform a lookup on each level to determine whether a node is part of that level. We can simply use an array of hash-maps for our levels container. This way, a lookup to see if a node is contained in a certain level is a constant time. Thus, the whole algorithm will run in $O(m + n)$ time. Notice, however, that we can assume the graph is connected by the problem statement. Thus, for a connected graph, we know that $m \geq n - 1$. Thus, we have that $m + m \geq m + (n - 1)$ and thus $O(m + n) = O(m + m) = O(2m) = O(m)$. Thus, our algorithm runs in $O(m)$.

$\square$

2

Given an undirected and connected graph, we define a node as an "articulation point" if and only if removing it disconnects the graph. Our goal is to develop an efficient algorithm for finding all the articulation points in a given graph using DFS.

2 (a)

Consider the following DFS algorithm introduced in the class where $D[u]$ stores the "discover time" of node u. Let's 'define another value $L[u]$ for each note, denoting the lowest D value of any node that is either in the DFS subtree rooted at u or connected to a node in that subtree by a non-tree edge. For a node u, how can we determine whether it is an articulation point using the L value of its children?

Solution. For a node u, if $degree(u) \leq 1$ then u is not an articulation point. This is true, because intuitively, for a point to be an articulation point, there must be a path passing through u, such that removing u 'cuts' that path. If u has degree 1 or 0, then there are no paths that go through u. Thus, removing u cannot possibly disconnect other nodes from the rest of the graph. Otherwise, check the following: (1) If u is the root of our DFS tree, (if $D[u] == 1$), then it is an articulation point. (2) If u has a child v such that $L[v] >= D[u]$, then u is an articulation point.

The intuition here is that a node u's L value will be equal to its D value, unless u has a path *through lower nodes* to an earlier-discovered node. Thus, if we remove node u, if it is *not* an articulation point, this implies there is some other path to get to its children; which implies there is some non-tree edge to its children lower in the tree; which implies its children's L values are all equal to that non-tree edge's D value. □

2 (b)

Design an algorithm to compute L values of each node during DFS and using it find all the articulation points of a graph. The algorithm should take $O(m)$ time.

Solution. The algorithm is pretty much the same as the given DFS algorithm. However, we also maintain an array of L values for each node. If we ever discover a non-tree edge (in the last line of the given algorithm), then we update the L value. I also store the parent of each node in the P array.

```
D[u] = -1, P[u] = -1, L[u] = Inf for all node u
AP = [];
counter = 1
DFS(s)
```

```
Function DFS(u):
    D[u] = counter++;
    L[u] = D[u];
    for each edge (u,v) in E:
        If (D[v] == -1):
            // (u,v) is a tree edge
            P[v] = i;

            DFS(v);

            L[u] = min(L[u], L[v]);

            if (len(E[u]) > 1):
                if (D[u] == 1 && len(E[u]) >= 2):
                    AP.push(u);
                else if (L[v] >= D[u]):
                    AP.push(u);
        Else:
            // (u,v) is a non-tree edge
            if (v != P[u]):
                L[u] = min(L[u], D[v])
```

We compute the L value of each node since the algorithm will iterate depth-first, and will get to the deepest-nested child v until it finds a non-tree edge. If it does, then v's L value is updated accordingly, and when we go up the stack, each parent also updates its L value accordingly to the subtree by looking at its children. If no non-tree edges are found, then each node's L value correctly remains equal to its D value.

Throughout the iteration, we check for the two conditions for identifying an articulation point listed in part (a); these articulation points are stored in array AP.

We can also prove the time complexity of this algorithm. The basic algorithm is still a DFS, and our additions do not cause the algorithm to visit any additional nodes that a traditional DFS would not. Thus, the basic algorithm is still $O(m)$. In addition, by using a traditional array for L and P, we have constant-time lookups for the L value and P parent of every node. In addition, by assuming an adjacency-list representation of the graph, we can lookup the array representing all edges of a node in constant time, and thus lookup the length of this array (degree of u) in constant time. The min function is also a constant-time function. Lastly, appending values to a traditional array AP is a constant-time operation. Thus, every operation that we have added to the basic DFS runs in constant-time, and thus the entire algorithm maintains the time-complexity of a traditional DFS: $O(m)$.

$\square$

3

From the interviews, they've learned about a set of n people, whom we'll denote $P_1, P_2, ..., P_n$. They've collected the following facts:

(1) For some i and j, person P_i died before person P_j was born,

(2) For some i and j, the life spans of P_i and P_j overlapped at least partially.

Give an efficient algorithm to either produce proposed dates of birth and death for each other the n people so that the facts are all true, or it should report that no such dates can exist.

Solution. If we think of each person P_i as a node within a graph, we can draw directed edges between the people to denote the constraints of relative chronology. More precisely, for each person P_i, we can define two nodes in our graph: B_i, which denotes the event of the "Birth" of P_i, and D_i, which denotes the event of the "Death" of P_i.

Recognize that the two forms of facts give us the following chronologies:

(1) If P_i died before P_j, this implies D_j occurred before B_i.

(2) If P_i and P_j lives overlapped a bit, then this implies both of the following: that B_i occurred before D_j, and B_j occurred before D_i. (A was born before B died, and B was born before A died)

Thus, given the input of people, we can construct two nodes for each person - a birth and death "event". Then, given the input of facts, we can construct a set of directed edges between these events, determining a chronological ordering constraint. Lastly, for each person P_i, let's define the following edge: B_i occurred before D_i (P_i was born before they died). With this information, we can create a directed graph - and we can attempt to create a topological sorting of these nodes. The topological sort will output an ordering of the events (i.e. when each person was born and died relative to each other). If the given facts are contradictory, then we will not be able to output a topological sort (the generated graph will contain cycle(s)) and we can detect this and output an error.

```
Function validateData(facts, numPeople) {
    Nodes = [];
    AdjacencyList = {};

    for i = 0 to numPeople:
        Nodes.add('B_i')
        Nodes.add('D_i')
        AdjacencyList['B_i'] = ['D_i']

    for each fact in facts:
        if (i died before j born):
            AdjacencyList['D_i'].add('B_j');
        else if (i and j coexisted):
            AdjacencyList['B_i'].add('D_j');
            AdjacencyList['B_j'].add('D_i');

    ordering = TopologicalSort(Nodes, AdjacencyList);

    if (ordering == null) return "Contradictory␣Data";
    else return ordering;
}
```

For the topological sort, we can use the following implementation learned about in class: Maintain a queue of nodes representing all nodes with zero in-degree. (Initialize this by iterating through the adjacency lists). Also maintain an array of the in-degree for each node. While the queue is not empty, place the head node into the output ordering; remove the node from the graph, and for that node, for each outgoing edge (u, v), decrease the value of the in-degree for node v. If node v's indegree is zero, add it to the queue. If the algorithm finishes prematurely (no nodes in the queue, even though not all nodes are outputted), this means there is a cycle and the graph is not valid (return null ordering).

This will output an ordering of birth and death events. If we want to be exact, the question asked for proposed dates of birth and death. So, we can initialize d_0 to be the starting date. Then, we can iterate through the output ordering array, with j the index of the array, and for each ordering[j], we can say that event occurred at date d_0+j. For example, for the ordering $[B_1, B_0, D_1, D_0]$, we can say P_0 was born on $d_0 + 1$, and died on $d_0 + 3$. P_1 was born on d_0 and died on $d_0 + 2$.

We can examine the time complexity of the whole algorithm. We know that a topological sort has time complexity of $O(|E|+|V|)$ where E is the edge set and V is the vertex set (using a queue to maintain zero-in-degree vertices). In our algorithm, we have two nodes for each person (one signifying their birth event, and one their death). Thus, $|V| = 2n$ where n is the number of people. Also notice, that we generate an edge for each node (birth before death), and that we generate an edge (or two) for each fact. (Let's assume we generate one edge per fact, since the constant will not be important later). Thus, we have $|E| = m+n$. Thus, our topological sort will run in $O(|E| + |V|) = O(m + 2n) = O(m + n)$.

Aside from the topological sort, we also have to set up our graph data using our input data. We have one loop over each person, where we initialize the nodes and adjacency lists. This loop will run n times since it iterates through the number of people. Then, we have a loop over each fact, which will append to an adjacency list (if using lists, appending is constant time). Thus, this second loop is $O(m)$. Thus, the initialization is $O(m + n)$. Thus, our entire algorithm time is the addition of the topological sort and the graph initialization code. This is $O(m + n + m + n) = O(2m + 2n) = O(m + n)$.

□

4

We know there are n flying saucers and each other them occupies the open interval (L_i, R_i) (assume L_i, R_i are integers). If the laser cannon is fired at position x, it will destroy all the flying saucers with $x \in (L_i, R_i)$. Mathematically, given n intervals $\{(L_i, R_i)|i = 1, ..., n\}$, our goal is to find a minimum set of numbers $X = \{x_1, ..., X_k\}$ such that for every interval i, there is at least one x_j in X contained in the interval $(L_i < x_j < R_i)$. Given an $O(n \log n)$ time algorithm to solve this problem, and prove the correctness.

Solution. Let's create a greedy algorithm that 'chooses' an x to shoot at based on an ascending order of the *end* time of each saucer interval. More concretely, sort each each interval according to R_i, and shoot at each R_i assuming the ith saucer is not yet shot at.

```
saucers.sort(x -> x.end);
shots = [];
for each i from 0 to n:
    if (shots.empty() ||
        shots.last() <= saucers[i].start):
        shots.push(saucers.end - 0.5);
```

Since we assume all intervals are bounded by integers, we can use an arbitrary offset of 0.5 in order to 'hit' any saucer between $(a, a+1)$ with a shot at $a+0.5$.

We can prove the correctness of this algorithm by showing that the greedy algorithm always makes a choice that is optimal. For any given saucer that is not yet shot, S_i, with begin and end times (L_i, R_i), we shoot at the edge of R_i ($R_i - 0.5$ in our example, since we assume integer bounds). If there is another saucer S_k with (L_k, R_k), if $L_k < R_i$, then we know it has already been shot at by our algorithm. If $L_k >= R_i$, then there is no such x where a shot at x will shoot both S_i and S_k in one shot. For each shot, moving it left would not improve the outcome since everything to the left has already been shot. Moving it right would not improve the outcome since moving the shot to the right would omit the current saucer S_i. Thus, by shooting at the earliest-ending saucers first, we know there is no better choice that we could have made to shoot that saucer.

We can prove the time complexity of this algorithm runs in $O(n \log n) time$. Sorting the saucers takes $O(n \log n)$ time. Initializing an empty 'shots' array is constant time. We then enter a loop n times; for each loop, we check if the 'shots' array is empty, then retrieve its last value; and conditionally append a value to the array. If we use a traditional array (with a length property), then checking if the array is empty is $O(1)$, retrieving any value from the array is $O(1)$, and adding an item to the array is also $O(1)$. Thus, the loop body runs in constant time, and thus the entire loop is $O(n)$. Thus, the sorting at the start of the algorithm is the most significant portion of the algorithm, and the entire algorithm runs in $O(n \log n)$ time.

□