CS 180 Midterm

Narek Daduryan

November 4, 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.

(a)

Any woman w remains engaged from the point at which she receives her first proposal, and the sequence of partners to which she is engaged get better and better.

Solution. True. We learned this theorem in class; at no point in the algorithm does a woman become non-engaged. The only time her matching is (m, w) replaced in the solution set is when a new m' proposes, whom w prefers more. Thus, she will always remain engaged and her partners can only increase. □

(b)

If a man m is free at the end of the algorithm, then he must have proposed to every non-forbidden woman.

Solution. True. The algorithm runs until there are no men that haven't proposed to every non-forbidden woman. Thus, if the algorithm ends, this implies that there does not exist such a man, who is free and hasn't proposed to every non-forbidden woman. □

(c)

If a woman w is free at the end of the algorithm, then it must be that no man ever proposed to w.

Solution. True. By (a) we showed that if a woman is proposed to, she will never become free again. Thus, the only way a woman can be free at the end of the algorithm is if she was never proposed to in the first place. □

(d)

At the end of the algorithm, there can be a man m and a woman w, such that $(m, w) \notin F$, but neither of which is part of any pair in the matching S.

Solution. False. By contradiction. Let's assume there is some man m, and w is not forbidden. Assume m is free; by (b) we know that m must have proposed to every woman not forbidden; thus, we know m must have proposed to w at some point. By (a), we know since w was proposed to, she cannot be free at the end of the algorithm. Thus, the contradiction. □

(e)

At the end of the algorithm, there can be a pair $(m, w) \in S$ and a man m' that is free, $(m', w) \notin F$, but such that w prefers m' to m.

Solution. False. By contradiction. Lets assume that there is a man m' that is free. Thus, by (b), we know that m' has proposed to every woman that is not forbidden. Thus, m' has proposed to w at some point. We know by (a) that w's partners will only increase, not decrease. Thus, w has only been engaged to people whom are less than or equal to m. Thus, if m' proposed at any time in the sequence, w would have chosen m', and by the end of the algorithm, w's partner would be greater than or equal to m'. Since m is less than m', this is a contradiction. $\square$

2

Solution. Notice that the given graph G with non-adjacency list is the complement of the graph G with an adjacency list. (If we treat the non-adjacency list as a normal adjacency list). Let's call the given (input) graph G, and its complement $\bar{G}$. In order to find connectedness, we can do a BFS or DFS. However, we need to do the BFS/DFS on $\bar{G}$ using the information in G. The following algorithm does that:

```
unreached = V;
queue = [];

choose any element v0 as the root
queue.push(v0);
unreached.remove(v0);

while(!queue.empty())
    node s = queue.pop();
    for (node u in unreached) {
        if (!edges[s].includes(u)) {
            queue.push(u);
            unreached.remove(u);
        }
    }
}

return unreachable.length == 0;
```

The algorithm is correct since given a starting node, it will try to reach all other nodes, but is only able to traverse using the non-edges of $\bar{G}$ (which are the actual edges of the graph G. This is a breadth-first search, since we are using a queue; for a given node u, we will evaluate all of u's neighbors before going to a deeper level. Thus, at the end of the algorithm, if G was connected, then we should be able to successfully visit every node using the non-edges of $\bar{G}$ (actual edges of G).

Examining the time complexity: We initialize "unreached" to every node, which is $O(|V|)$. The rest of the initialization is constant time, until the actual BFS. The BFS will visit every node. There are $|V|$ unreached nodes, and we will end up reaching at most $|V|$ of these nodes; or we will end up failing to reach at most $|V|$ nodes. Thus, in this BFS, we will evaluate reached/unreached nodes at $O(|V|)$ time. Thus, the whole algorithm runs in $O(2|V|) = O(|V|)$. Since we are given that $|E| > |V|$, we can conclude that the algorithm runs in $O(|E|)$. □

3

Given a Directed Acyclic Graph (DAG) $G = (V, E)$, design an algorithm to determine whether there exists a path that can visit every node. The algorithm should have time complexity of $O(|E| + |V|)$. Prove why your algorithm is correct.

Solution. Since we have a DAG, we know that all DAG's will have at least one valid topological sort. A topological sort is a sorting of the nodes in the DAG such that all the edges between nodes go from 'left to right' (For every edge $e = (v, u)$, v occurs in the topological sort before u).

Thus, to find a path in the DAG that goes through all nodes, we simply need to construct a left-to-right path in a given topological sorting of the graph G. If the path from left-to-right in the topological sort is not valid, then there doesn't exist such a path.

```
find_all_path(G = (V,E)) {
    sorted = topological_sort(G);

    for (i from 0 to V.length - 1):
        if (there is no edge between sorted[i] and sorted[i+1]):
            return false;

    return true;
}
```

We can verify that this is correct. Every DAG has at least one topological sort. Let's verify that if we have a topologically sorted node sequence, choosing any path other than simply left-to-right would not be a valid path. Intuitively, in the topological sort, every edge goes from left to right. Thus, if we construct a path such that we do not simply go left-to-right, then at some point, the path will need to go right-to-left in order to visit all the nodes, and this is impossible since there do not exist any edges left-to-right.

More concretely, the topological sort returns a sequence of nodes $(v_0, v_1, ..., v_i)$. If we begin constructing our path at some node v_j where $j \neq 0$ (i.e. start the path not at the beginning of the sort), then we will never be able to visit not v_0, since there does not exist any edge $e = (v_k, v_0)$ where $k > 0$ (due to the definition of topological sort). Thus, we have one constraint: we must start our path at v_0 (the beginning of the topological sort) in order to reach all nodes. Next, let's prove by similar reasoning that we cannot "jump" any nodes in the sort. If somewhere in our path we are at v_k, let's assume that we choose as the next node in our path v_{k+n} where $n > 1$ (i.e. we "jump" over v_{k+1}. Then, by a similar reasoning, there does not exist any path from v_j to v_{k+1} where $j > k+n$. Thus, we cannot reach node v_{k+1}, and this is not a valid path according to our spec.

Thus, we have proved that for a topologically-sorted sequence of nodes, we must start our path at the first node (v_0), and we cannot "jump" over any nodes. We also know we have to travel to the right (since there does not exist any edge $e = (v_j, v_k)$ such that $j < k$). Thus, by these constraints, we know that given a topological sort, the only possible valid path that touches all nodes must be the sequence of nodes $[v_0, v_1, ..., v_n]$. Thus, this is the only possible path that can touch all nodes in G. We cannot guarantee that this path does exist in G, so as long as there are edges from each v_k to v_{k+1}, then this path does exist in G, and there exists a path in G that touches all nodes. If there is an edge missing in this theoretical path, then, since we know this would be the only possible path that could work, then there does not exist a path in G such that the path touches all nodes.

The time complexity of this algorithm is based only on the topological sort, and the additional for-loop. The topological sort itself can run in $O(|V|+|E|)$ as shown in class. Then, we iterate through each vertex, which has $|V|$ iterations, and for each iteration, we have to iterate through the entire adjacency list of v to see that an edge does not exist. Thus, we iterate $|V|$ times, and for each iteration of vertex u, we iterate over $degree(u)$ items. Thus, the total for-loop runs in $O(\sum_{v \in V} degree(v)) = O(|E|)$.

Thus, we have the topological sort $(O(|E| + |V|))$ and for-loop $(O(|E|))$. Thus, the whole algorithm is $O(|V| + 2|E|) = O(|V| + |E|)$.

$\square$

4

Solution. Given the array of length n, we can iterate through the array, keeping track of the unique partitions that we have seen so far. We keep track of two pieces of storage: "part_idx" and "partitions".

"part_idx" is a 1-D array representing the index of the first element that is a unique occurrence. "part_idx" contains indices into the input array, all of which are distinct elements. For example, if the input array is $[a, b, a, c, b]$, then the "part_idx" array would be: $[0, 1, 3]$ since the unique elements in this array are a at index 0, b at index 1, and c at index 3.

"partitions" is the output that we are generating. It is a 2D array, where each top-level index represents a unique partition. For each partition, there is a 1D array containing the indices that belong to that partition. For example, if the input array is $[a, b, a, c, b]$, then "partitions" is $[[0, 2], [1, 4], [3]]$. Notice that when the algorithm is complete, "partitions" is the requested return value.

Let us define an algorithm that will iterate through each element in the array. For each element, we can use a binary-search to determine which existing partition that element should be a part of, or if it creates a new partition. Intuitively, we iterate over n elements in the array, and if we can create a binary search which runs in $O(logn)$, then we can have an algorithm that works in $O(nlogn)$.

```
let part_idx = [];
let partitions = [];

for i in [0, n):
    partition_index = binary_search(part_idx, i);
    if (partition_index exists) {
        partitions[partition_index].add(i);
    } else {
        part_idx.add(i);
        partitions.add([i]);
    }

return partitions
```

Let's assume "binary_search" is an $O(logn)$ function which, given the "part_idx" and a new index i, returns the index in "part_idx" which i belongs to, or determines that i does not belong to any previous partition.

Then, we see that the given algorithm is correct. Since, for each element, it will either add it to an existing partition if a match is found, or it will create a new partition for it if it is unique. Then, we can see that the "partitions" array will contain all the indices partitioned into unique values.

We can also see that the given algorithm will run in $O(nlogn)$. We initialize some arrays to zero ($O(n)$), and then enter the top-level for-loop. This loop runs n times. For each iteration, we will run "binary_search", which we assumed runs in $O(logn)$ time. When adding to the end of arrays, this is constant time. Thus, with "binary_search" for $O(logn)$ and the n iterations, the entire algorithm is $O(nlogn)$.

Now, let's try to implement a "binary_search" algorithm which will run in $O(logn)$ and only use the query() interface given.

```
function binary_search(part_idx, i):
    if (part_idx.length == 0) return -1;

    numDistinctElements = arr.query(part_idx + {i});

    // if there are more distinct elements than we have previously seen,
    // then index i is itself a distinct element
    if (numDistinctElements > part_idx.length) return -1;

    // if part_idx is one long
    // then index i's partition must be that first element
    if (part_idx.length == 1) return 0;

    half_index = part_idx.length / 2;
    leftHalf = part_idx from [0, half_index)
    rightHalf = part_idx from [half_index, end)

    numDistinctInLeftHalf = arr.query(leftHalf + {i});
    numDistinctInRightHalf = arr.query(rightHalf + {i});

    if (numDistinctInLeftHalf == leftHalf.length)
        return binary_search(leftHalf, i);
    if (numDistinctInRightHalf == rightHalf.length)
        return binary_search(rightHalf, i);
```

This is correct, since we use the fact that given a a set of n distinct indices, if query$(a+\{i\})$ returns $n+1$, then i is a distinct element. Otherwise, we can do the same trick on the left and right half of the array. If the left-half query returns n', n' being the length of the left-half array, then we know i's partition exists in the left half, and we keep going... This will run with $logn$ steps, because at each step, we get rid of half of the "part_idx" array, which has a maximum length of n, where each element is distinct. In the body, the only non-trivial computation that we do is slicing and appending arrays. Traditionally, slicing an array would be $O(n)$. However, we can use some more advanced data representation, where instead of slicing and making a copy, we simply take a "view" of the existing array in a limited range. This is possible in many programming languages. Then, we can get a view of a subset of an array in $O(1)$ time. Thus, the body of the binary_search is $O(1)$, and itself runs $logn$ times, thus we have successfully made a binary search using the query() function that runs in $O(logn)$.

We conclude that given the two code snippets, the total algorithm is $O(nlogn)$ and is correct! $\square$