601.433/633 Introduction to Algorithms **Fall 2024**
Homework #6 **Due:** October 31, 2024, 9am

Bella Lu
Partners: Jecia Mao, Luna Liu

1 Reachable nodes (33 points)

Let G be a directed graph represented by an adjacency list. Suppose each node u has a weight $w(u)$, which might be different for each node. Give an algorithm that computes, for every node u, the maximum weight of any node that is reachable from u. So, for example, if G is strongly connected then every node can reach every other node, so for every node the maximum reachable weight is the same (the largest weight of any node in the graph).
More formally, for each vertex u let $R(u)$ denote the vertices reachable from u. Then when your algorithm is run on G, it should return an array of values where the value for node u is $\max_{v \in R(u)} w(v)$. Your algorithm should run in $O(m+n)$ time. Prove correctness and running time.

Original graph:

Vertex	Adjacency List
u_1	$R(u_1)$
u_2	$R(u_2)$
...	...

Step 1: Kosaraju's algorithm

After running Kosaraju's algorithm on the graph G, we will get a list of all the strongly connected components

$$\begin{array}{ll} \text{SCCs after algorithm:} & \underbrace{[u_i, u_{i+1}, ...]}, \underbrace{[u_j, u_{j+1}, ...]}, \underbrace{...} \\ \text{New vertices:} & [v_1] \quad [v_2] \quad ... \end{array}$$

We can name each group of SCC to be $v_1, v_2, ...$

Correctness
Kosaraju's algorithm is correct, and will find/return the SCCs in G.

Runtime
As proved by Kosaraju's algorithm, the runtime is $O(m+n)$.

Step 2: Create mappings

We can create a mapping, or a dictionary, between all old vertices to the new vertices.

$$[u_i] \to [v_1]$$
$$[u_{i+1}] \to [v_1]$$

$$[...] \rightarrow [v_1]$$

$$[u_j] \rightarrow [v_2]$$
$$[u_{j+1}] \rightarrow [v_2]$$
$$[...] \rightarrow [v_2]$$

...

Correctness
We are simply creating a mapping from each old node to new node, which makes later looking faster.

Runtime
This takes O(n) because we traverse every vertex in the old graph.

Step 3: Create new graph
New graph:

Vertex	Adjacency List
v_1	$R(v_1)$
v_2	$R(v_2)$
...	...

Correctness
The value of each new vertex (v) would be the maximum of all old vertices (u) that was mapped to it. This is correct because if a graph is strongly connected then every node's max reachable weight is the same as every other node in this graph, which is the largest weight of any node.

Runtime
Finding the new value of every node in the new graph takes $O(n)$, where n is the maximum number of nodes possible in this new graph.

To find the edges of this new graph, we would need to traverse all the edges in the old graph and figure out the new edges this new graph.
Each checking is O(1), and we traverse $O(1) \cdot O(m+n) = O(m+n)$

Overall, this step takes $O(n) + O(m+n) = O(m+n)$.

Step 4: Topological sort
After running topological sort on the new graph, a list from nodes with no incoming edges (ancestor) to nodes with no outgoing edges (children) will be returned.

Correctness
Topological sorts are correct, and will return the correct list.

Runtime

Topological sort takes $O(m+n)$.

Step 5: Bottom-up DP
We can start with vertices with no outgoing edges (at the end of the list returned by the topological sort), and build our OPT table up to vertices with no incoming edges. This follows exactly the order (from back to front) we found from the topological sort.

$$OPT(v_i) \begin{cases} v_{i\_value} & \text{if } v_i \text{ has no outgoing edges,} \\ max[OPT(v_{\text{nodes with incoming edges from } v_i}), v_{i\_value}] & \text{otherwise} \end{cases}$$

Correctness (By induction)
Base case: When a node has no outgoing edges, the only node reachable is itself. Thus, its OPT must be its value.

For the sake of induction, assume that for $i < i' < n$, $OPT(v_{i'})$ is correct.

OPT(v_i) is the maximum between the OPT of all its neighboring edges, with indices larger (which by the induction hypothesis, is correct) and its value. This gives the correct OPT(v_i) because OPT(v_i's children) will store the maximum reachable weight at that node. If v_i can reach this node, it means that v_i can also reach any nodes that its children can reach.

Thus, OPT(v_i) is correct.

Runtime
We have to calculate the OPT at every node, which is at most n. The total number of edges is at most m. To calculate the OPT at each node, we need to look at all the neighboring edges that has indices larger (from topological sort). However, once we look at an edge, we will never look at it again in our DP. So the total number of edges we looked at after computing all the OPTs is m. Thus, this takes $O(n) + O(m) = O(m+n)$.

Step 6: Output
Build an empty array of size n, and let the value for each node u be the corresponding v's OPT value. Since checking is $O(1)$, this is simply $O(n)$.

Proof of Correctness for Entire Algorithm
Each step is correct, proven in each step above. The final output will be an array of values where the value for node u is $\max_{v \in R(u)} w(v)$.

Proof of Runtime for Entire Algorithm
Step 1: $O(m+n)$
Step 2: $O(n)$
Step 3: $O(m+n)$
Step 4: $O(m+n)$
Step 5: $O(m+n)$

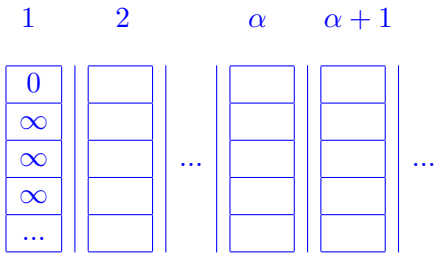
Step 6: $O(n)$

Each runtime is proven in each step of the algorithm.
The maximum runtime of the steps is $O(m+n)$, therefore the total runtime of this entire algorithm is $O(m+n)$.

2 Faster Shortest Paths (33 points)

Let $G = (V, E)$ be a directed graph with lengths $\ell : E \to \mathbb{R}$ with no negative-length cycles. Let $v \in V$, and let α denote the maximum, over all $u \in V$, of the number of edges on a shortest path from v to u (where the shortest-path is defined with respect to the lengths). Given G, ℓ, and v (but not α), give an algorithm that computes shortest path distances from v to all other nodes in $O(m\alpha)$ time. You may assume that $m = \Omega(n)$. Prove correctness and running time.

Iteration:



Algorithm
Run Bellman-Ford on the graph G, terminate when OPT for every node stays the same from one iteration to the next. This means terminating at the $(\alpha + 1)$th iteration.

Proof of Correctness
Without loss of generality, we can assume that there exists a node (we can call x) where it needs α iterations before it finds its optimal path. All other nodes will have founds its optimal at iteration α or before.

For iterations $k \leq \alpha$
Assume for the sake of contradiction that it is able to terminate early, which means that at a point from iteration k-1 to k, all OPTs don't change. This means that $OPT(x_{k-1}) = OPT(x_k)$. This cannot be true because this means that the number of iterations needed is no longer α.

For iterations $k > \alpha$
Assume for the sake of contradiction that it will find its optimal at some iteration larger than α, i.e. $OPT(x_k) \neq OPT(x_{k+1})$. This cannot be true because this means that it wasn't the final optimal for x at iteration α, and that it needed longer. However, in the premise we said that it only needs α iterations before it finds its optimal path.

Thus, we have proven that there is at least one OPT changes from iterations before α, and nothing changes after iteration α.

Proof of Runtime

We can terminate when we reach the $(\alpha + 1)$th iteration. Although we don't know what α is, we can check when the array stays the same after one iteration. This would take at most $\alpha + 1$ iterations. The time per iteration is $O(m)$, as proved by Bellman-Ford. Thus, the total run time is $O((\alpha + 1)m) = O(\alpha m)$.

3 Traveling With a Coffee Constraint (34 points)

Let $G = (V, E)$ be a connected, undirected graph with positive edge lengths $\ell : E \to \mathbb{R}^+$. This graph represents a road network. You are trying to get from $s \in V$ to $t \in V$. Unfortunately, you are a coffee addict: getting caffeine often enough is crucial. There is a coffee shop at every vertex $v \in V$ (including s and t) where you can get coffee.

(a) (17 points) Since you care most about getting coffee regularly, what you really want to do is minimize the amount of time you are on the road between coffee stops. More formally, you want to know the minimum L so that there is a path from s to t in which every edge has length at most L. Design an $O(m \log n)$-time algorithm for this problem, and (as always) prove running time and correctness.

Algorithm

Mergesort on the edge lengths (i.e. least to greatest), then we can go through the list, extracting the minimum until we create a path that connects s to t. To know when we have obtained a valid path, we can use a union-find structure. More specifically, make-set on each extracted min edge, union with current tree, and check if find(s) = find(t). We keep extracting min edge until s and t have the same root, which by then we have found a path that connects the two vertices.

Proof of Correctness

Mergesort is correct, and will correctly sort the edges by length.

Union-find (with union by rank and path compression) is correct, and by checking find(s) = find(t) we should be comparing their roots. This means that if its equal, the nodes in our current tree creates a path that connects them.

For the algorithm, we can assume by contradiction that the final path we found has an edge length greater than L. This cannot be true because by the order of extracting edges, we should have visited an edge before that created a valid path of maximum length L, and the process should have terminated. In other words, this edge of length larger than L shouldn't have been added to our path.

Proof of Runtime

Mergesort: $O(m \log m)$

m can be at most n^2, so $O(m \log m)$ can become $O(m \log n^2) = O(2m \log n) = O(m \log n)$.

Union-Find (union by rank and path compression):

For each new minimum edges, we do the following operations:

Make-Set of the new edge: takes O(1).

Union with current path: Two finds plus a link, which is $O(\log^* n)$

Check Find(s) and Find(t): $O(\log^* n)$

The maximum edges that could be added is m, so we could do the above operations m times. Thus, finding a path that connects s and t takes $O(m \log^* n)$.

Total runtime: $O(m \log n) + O(m \log^* n) = O(m \log n)$

The mergesort dominates, so the final runtime for the entire algorithm is $O(m \log n)$.

(b) (17 points) Now let's consider a variant where you're an addict and a snob: while you want coffee, you *hate* Starbucks. Unfortunately, you're even more of an addict than in the previous problem: there is some number $L \in \mathbb{R}^+$ so that if you go more that L distance without getting coffee, you will get a caffeine migraine and crash your car. So you simply *cannot* go more than L distance without stopping for coffee. So now you have the following problem: given a subset $S \subseteq V$ of vertices where the only coffee shop is a Starbucks, find the path from s to t which uses only edges of length at most L, and has the *fewest* vertices in S. You may assume that $s, t \notin S$. Design an $O(m + n \log n)$-time algorithm for this problem, and prove running time and correctness.

Algorithm

We can first traverse the entire graph and create a new graph with all edges with length greater than L removed. Then, we can run Dijkstra's algorithm with all edge weights equal to 1 for edges going to a starbucks vertex, and 0 otherwise, so that we are minimizing edges going to starbucks.

Proof of Correctness

After removing all edges with length larger than L, we are only with valid paths, so we won't go down invalid edges at all.

There are no negative edge lengths in this problem, so we can use Dijkstra's algorithm. Dijkstra's algorithm returns the shortest path from a source node (s) to all other nodes in the graph.

If all the edges not going to Starbucks have weight 0, then they are basically free. Indeed, we are only minimizing the number of Starbucks vertices, and don't care about edges not going to Starbucks.

If all the edges going to Starbucks has weight 1, then we are simply minimizing the number of vertices because they all have the same weight.

The final output of this algorithm would be the path with the least Starbucks vertices between s and t.

Proof of Runtime

Create new graph: $O(m+n)$

Dijkstra's: If we use a Fibonacci Heap, the running time is $O(m+n\log n)$, as proven in class

Dijkstra's runtime dominates, so the final runtime of the algorithm is $O(m+n\log n)$.