CSE 101 Homework 2

Winter 2023

Question 1 (The Easy Way Down, 30 points)**.** *Dave was already at the top of the ski slope by the time he realized that he wasn't prepared for it. Fortunately, the slope has many branching paths (given by a DAG G) that can be taken to the bottom, and Dave is determined to find the easiest way down. Two vertices of G are labelled TOP (Dave's current location) and $BOTTOM$ (the place he is trying to reach). Furthermore, each edge has been assigned a difficulty rating. Dave is trying to find a path to the bottom so that the highest difficulty edge he needs to use is as small as possible.*

Give a linear time algorithm that given G, TOP, $BOTTOM$ and the difficulty ratings finds Dave's best path.

Hint: For each vertex v, compute the maximum difficulty of the best path from TOP to v. If you do this for each v in the correct order, it is relatively straightforward.

We begin by getting the topological ordering of our graph $G = (V, E)$ by running the topological sort algorithm. Let the difficulty of an edge, $(u, v) \in E$ be given by $d(u, v)$. We define the cost of a path to be the highest difficulty of any edge on that path. We say that a path from u to v is the best $u - v$ path, if it has the least cost among all paths from u to v. Let us define a function, $f(v)$ which gives us the cost of the best path from TOP to v. If there is an edge (u, v) incident on v, then $f(v)$ depends on $f(u)$ and $d(u, v)$, where $f(u)$ is the least cost of the best path from TOP to u and $d(u, v)$ is the difficulty of the edge (u, v). As we're finding the highest difficulty edge of the path, this would be $\max(f(u), d(u, v))$. We need to calculate the least cost among all the paths from TOP to v. Thus, $f(v)$ is defined as follows:

$$f(v) = \min_{(u,v)\in E}(\max(f(u), d(u, v)))$$

We define $f(TOP) = 0$. We calculate $f(v)$ for each vertex v in the order of the topological order beginning from TOP. Along with $f(v)$ for each v, we also store $g(v) = u$ which corresponds to the vertex u that gives us the value $f(v)$. We break ties arbitrarily. To return the path that we need, we start from $g(BOTTOM)$ and trace back the path to TOP.

Runtime Analysis: The topological sort algorithm runs in $O(|V| + |E|)$ time. When we calculate $f(v)$ and $g(v)$, for each v, we check all edges incident on v. Thus, every vertex and every edge is used at most once. Thus, calculating $f(v)$ for all vertices takes $O(|V| + |E|)$. Thus, the algorithm is linear in the number of vertices and edges of G.

Proof of Correctness: We prove the correctness of this algorithm by showing that whenever the algorithm assigns a value to $f(v)$ that the value is the cost of the best path from TOP to v. We prove this by induction on the index of v in the topological order. We use k to define the index of the vertex v in our topological order.

Base case: $k = 0$

This is when the vertex $v = TOP$, thus we return $f(TOP) = 0$.

Inductive Step:

Assume that for an index $k < n$, $f(v_k)$ is the correct value for the cost of the best path from TOP to the kth index in our topological sort. Since, we are going in topological ordering, for all vertices that come before v_k, $f(v)$ is computed correctly. Now, to calculate $f(v_n)$, say there are $u_1, u_2, ..u_m$ vertices that have edges incident to v_n. For v_n, to find the cost of the best path from TOP through u_i, we need to consider two terms. Either the cost of the best path is the difficulty of the edge (u_i, v_n), thus we need to consider $d(u_i, v_n)$. Or, the cost of the best path is from one of the edges in the path from TOP to u_i. This is already calculated in $f(u_i)$. And as we're looking for the highest difficulty of an edge on that path from TOP to v_n, $\max(f(u_i), d(u_i, v_n))$ gives us the cost (highest difficulty) of the best path from TOP to v_n via u_i. To find the least cost among all paths, i.e. to find the best path, we need to find the minimum over all such u_i. Thus, we calculate $f(v_n) = \min_{1 \leq i \leq m}(\max(f(u_i), d(u_i, v))$. Since, we know that $f(u_1), f(u_2), ..., f(u_i)$ are all computed correctly, $f(v_n)$ is also computed correctly. Thus, using the correct $f(v)$ values for all vertices our algorithm is able to return the correct path from TOP to $BOTTOM$.

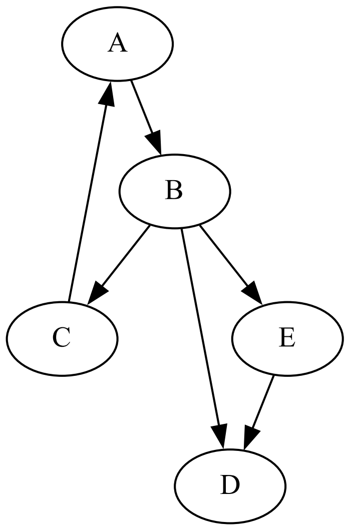
Node	Preorder	Postorder
A	1	10
B	2	9
C	3	4
D	5	6
E	7	8

(a) Pre and post order for Fig. 1a

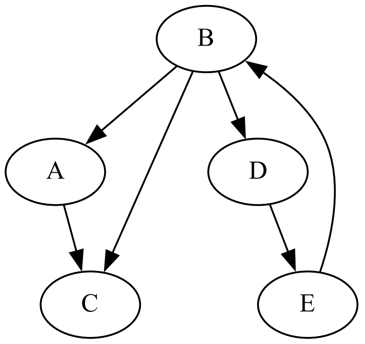
Node	Preorder	Postorder
A	1	4
B	5	10
C	2	3
D	6	9
E	7	8

(b) For Figure 1b

Table 1: Pre and post order tables for Figure 1



(a) Q2 (a) and (b)



(b) Q2 (c)

Figure 1: Counter examples for Q2

Question 2 (Other Attempts to Find Source and Sink Components, 15 points)**.** *In class we showed that in a directed graph G the vertex with the largest postorder number is in a source SCC. Provide counter-examples to disprove the following statements:*

(a) The vertex with the smallest postorder number is always in a sink SCC. [5 points]

(b) The vertex with the largest preorder number is always in a sink SCC. [5 points]

(c) The vertex with the smallest preorder number is always in a source SCC. [5 points]

The counterexample graphs are provided in Figure 1. The pre and post order number tables are in Table 1.

(a) In Figure 1a, we see that the sink SCC is D. The postorder number for D is 6. From Table 1a, we can see that the smallest post order number is for vertex C, which is not a sink SCC

(b) In Figure 1a, we see that the sink SCC is D. The preorder number for D is 5. From Table 1a, we can see that the largest pre order number is for vertex E, which is not a sink SCC.

(c) In Figure 1b, we see that the source SCC is the connected component consisting of $\{B, D, E\}$. From Table 1b, we can see that the smallest pre-order number is at A, which is not a source SCC.

Question 3 (Max Reachable, 25 points)**.** *Let G be a directed graph where every vertex is assigned a real number. We wish to compute for each vertex v of G the largest number assigned to any vertex reachable from v.*

(a) Give a linear time algorithm for this problem if G is strongly connected. [5 points]

(b) Give a linear time algorithm for this problem if G is a DAG. [10 points]

(c) Give a linear time algorithm for this problem for a general directed graph G. (Hint: you will want to find a way to combine the previous algorithm ideas.) [10 points]

(a) If G is strongly connected, every vertex is reachable from every other vertex. Thus, we can keep track of the largest number assigned to any vertex in the graph and compute the maximum among them, say m. Finally, assign m to every vertex and return that for every vertex.

The time complexity of the algorithm is linear and O(V), where V is the number of vertices in graph G since we just need to find the maximum values among the numbered vertices.

(b) If G is a DAG, we can run topological sort on the vertices to find the order of vertices from source to sink. We reverse this order. We define a function f(v) to give us the largest number among the vertices reachable from a vertex v. Let $N(v)$ be the number assigned to a vertex v. For the sink vertices s, $f(s) = N(s)$, otherwise

$$f(v) = \max(N(v), \max_{(v,w)\in E} f(w))$$

We recursively calculate the function $f(v)$ for each vertex v in reverse topological order and update the $N(v)$ for the vertices. This ensures that we are first calculating the answer for a vertex with no outgoing edges and using those answers to calculate the answer for the parent vertices pointing to those vertices whose answer we have already calculated. We ultimately end up calculating for each vertex v the largest number for all the reachable vertices from v.

Proof of correctness: Proof by strong induction:
Inductive hypothesis: We prove by strong induction that each assigned value f(v) is correct.
Assume that $f(v)$ has been correctly assigned to every index with index greater than i i.e all vertices downstream from vertex v_i. Thus, we prove that for index i, $f(v_i)$ gives the correct value for the largest number reachable from vertex v_i.
Inductive step: Every vertex reachable from v_i is either v_i or is reachable from some w with $(v_i, w) \in E$. The maximum number reachable from w is f(w), so the largest number overall is $\max(N(v_{i-1}), \max_{(v_{i-1},w)\in E} f(w))$. Thus, computing this maximum will find the largest number reachable from vertex v_i. Hence, by recursively computing $f(v)$ for all vertices, the algorithm can correctly determine the largest number reachable from each vertex and then assigns it to the vertex.

Thus, by strong induction we complete the proof.

Time complexity: The topological sort algorithm takes $O(|V| + |E|)$, and reversing the order takes $O(|V|)$ time. The function f(v) is calculated for each vertex which can take $O(|V| + |E|)$ time leading to an overall linear $O(|V| + |E|)$ time complexity.

(c) For a general directed graph, we can have multiple strongly connected components (SCC). Compute a metagraph G' of graph G with each vertex being an SCC creating a DAG G'. Assign to each vertex w of G' (corresponding to a SCC C of G) the largest number of a vertex in C. Then run the algorithm from part (b) on G' to calculate the answer for each vertex(SCC). Finally, assign to each vertex v of G the number assigned to its SCC in G'.

Our pseudocode is as follows:

1. Compute a metagraph of graph G, say graph G'
2. For each SCC C in G, assign the largest number of a vertex in C to each vertex in C.
3. Run algorithm from part (b) on G' to compute the largest number reachable from every vertex C in G'.
4. Assign to each vertex v of G the number assigned to its SCC in G'.

Proof of correctness

If v is in SCC C, then v can reach all vertices w in any SCC reachable from C in G'. The algorithm from part (b) correctly computes the maximum over C' reachable from C of the number assigned to C' (which is the largest number of any element in C'). Thus, the output assigned to v is the largest number assigned to any w reachable from v.

Time complexity The algorithm to compute metagraph G' takes $O(|V| + |E|)$. As shown above the algorithms of part (a) and (b) take linear time leading to an overall time complexity of $O(|V|+|E|)$.

Question 4 (Line Switching, 30 points)**.** *The subway system of Graphopolis is given by an undirected graph G with the vertices representing stations and the edges representing tracks. Furthermore, the edges are partitioned into lines. Each line consists of some contiguous collection of edges and a traveller can travel from any station on a given line to any other without changing trains.*

Give an algorithm that given the graph G, a description of the lines and two stations v and w, determines the fewest number of times that a traveler would need to change trains to get from v to w. For full credit, your algorithm should be linear time.

We think of a new problem. First, starting at the source station, $s = s_0$, we get on some line ℓ_1. We travel along this line and get off at station s_1. From there we transfer to line ℓ_2, which we take to station s_2 and so on. Eventually, we end up at station s_k, which is our destination. We want to minimize k, the number of lines used in such a route. Note though that the number of steps on this path of transferring between stations and lines has length $2k$. Therefore, we can write this problem as that of finding the shortest paths in a different graph G'. The shortest path in this new graph G' has a length twice the number of lines needed to reach the destination. Our pseudocode is as follows:

1. Create a new graph G'
2. The vertices of G' are given by either:
   - A color, or line on our subway map.
   - A station on our subway map.
3. For each edge in G, add an edge to G' between the line the edge is a part of and the stations on either end of it.

4. Let v be our starting vertex and w our destination vertex, both correspond to station-vertices of G'.

5. Run BFS(G', v).

6. Return dist$(w)/2 - 1$. (We want the number of times we *change* lines, which is one less than the total number of lines used)

Proof of correctness

Given a path P in G, we build a path in G' that uses all of lines used in P and all of the stations at which line changes take place. We note that since this station connects to the lines on each end, it corresponds to a path in G', and it is not hard to see that any path in G' similarly corresponds to a path in G. If the path P uses n different lines, the path in G' will use those n lines, the vertices v and w, and n-1 line change stations and thus have 2n edges. Thus the length of the path in G' will be 2(#line changes + 1). When there is a line change, an extra traversal of edges to and from a line vertex needs to be traversed thus guaranteeing the shortest path to be between station vertices having the least number of line changes. Since, BFS guarantees to find the shortest path between vertices, running BFS on graph G' can find the path with minimum train changes between the source and destination vertices.

Time complexity

To analyze the runtime of this algorithm, we note that the number of vertices of G' is the number of vertices of G plus the number of lines, which is at most $|V| + |E|$. Furthermore, we can produce G' by creating a vertex for each station and line, and then for each edge of G creating an edge between the appropriate line and the two stations on either side. Thus, we can produce the graph G' in $O(|V| + |E|)$ time. Finally, the Breadth First Search runs in time $O(|V'| + |E'|) = O(|V| + |E|)$. Thus, our entire algorithm runs in time $O(|V| + |E|)$.

Question 5 (Extra credit, 1 point)**.** *Approximately how much time did you spend working on this homework?*