CSE 101 Homework 2 Solutions

Winter 2021

This homework is due on gradescope Friday January 29th at 11:59pm pacific time. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in LaTeXis recommend though not required.

Question 1 (Marble Run, 35 points)**.** *Connie is building a marble run. She has built a complicated downhill, branching track for marbles to run along represented by a DAG G. When a marble hits a vertex of this graph that it not a leaf, it will travel along a random one of the outgoing edges. With some experimentation, Connie has computed for each vertex the probability that the marble will leave through each of its outgoing edges.*

Give an algorithm that given G, these probabilities and the vertex s where the marble starts, computes for each vertex v of G, the probability $p(v)$ that a marble running through this course will at some point pass through v. For full credit, your algorithm should run in linear time. Hint: Compute the values of $p(v)$ one at a time in an appropriate order.

Solution 1.

Let $P_{u,v}$ represent that the probability that the marble leaves vertex u through edge (u, v), and set $p(s) = 1$. We first compute the topological order of G (specifically the subset of G reachable from s) by running a DFS starting from s and returning the vertices in reverse postorder (algorithm described in lecture, but only with vertices that s reaches). During our DFS, we add every edge that we traverse to a new set E' (representing the edges reachable from s). We then iterate through the vertices in topological order, and for each vertex v, we compute $p(v) = \sum_{(u,v)\in E'} p(u) * P_{u,v}$.

Finding the topological order of G takes linear time, as shown in lecture. We then perform a constant-time operation for each incoming edge for each vertex in V, which is clearly $O(|V| + |E|)$. Thus, our algorithm is linear time, as required.

Since our algorithm computes $p(v)$ in topological order, we know that for every incoming edge (u, v), our algorithm has already computed $p(u)$, so our algorithm at the very least runs without error. We approach a proof of correctness through an inductive argument. For our basis step, note that clearly $p(s) = 1$ holds. Now, for the inductive step, assume that we have computed $p(w)$ correctly for every vertex w before vertex v in the topological ordering of G; that is, $p(w) = P(\text{marble passes through } w)$. Then, for vertex v, note that the probability that a marble will pass through v must be equal to the sum over all incoming edges of the probability that the marble will pass through an incoming edge to v. Let (u, v) represent such an edge. Then $P(\text{marble passes through } (u, v)) = P(\text{marble passes through } u) * P(\text{marble leaves } u \text{ through } (u, v)) = P(\text{marble passes through } u) * P_{u,v} = p(u) * P_{u,v}$ (by the inductive hypothesis). Summing over all incoming edges (u, v) reachable from s, we get $p(v) = \sum_{(u,v)\in E'} p(u) * P_{u,v}$, which is what our algorithm returns.

Question 2 (Contracting Cycles, 20 points)**.** *Consider the following method of computing the metagraph of a graph G. If G is a DAG, return G. Otherwise, find some cycle C in G and let G' be the graph obtained from G by replacing all of the vertices of C with a single vertex with edges to/from the same vertices that have edges from/to some vertex of C. Then recursively compute the metagraph of G'. Prove that this algorithm correctly computes the metagraph.*

Solution 2.

In order to prove that this algorithm correctly computes the metagraph, for each iteration we need to check:

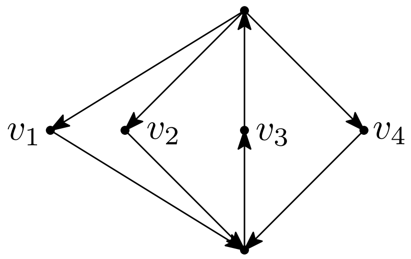

Figure 1: Visiting vertices

(1) For each iteration of this algorithm, the original graph G and the updated graph G' have the same metagraph.

(2) Eventually the algorithm will end up with a DAG.

(3) The metagraph of a DAG is the DAG itself.

To prove (1), the first step is to show that all the vertices in the cycle C are in the same SCC. For an arbitrary vertex in the cycle C, there exists a path from this vertex to any other vertices along the cycle and finally the path will lead back to the vertex itself. This proves that all the vertices in cycle C are strongly connected, thus they belong to the same SCC. More genrally, we need to show that for any vertices u and v there are paths from u to v and back in G if and only if there are such paths in G'. If there is a path in G that does not pass through any vertex of C, we can use the same path in G'. If the path does pass through C, then we can replace every string of consecutive vertices of C in this path with the corresponding vertex in G'. Since for any edge entering leaving C in G will have a corresponding edge in G', this should work. If on the other hand we have a path in G', we also need a path in G. If the path doesn't pass through C, we are done. If it does, suppose it goes to this vertex from some vertex a and leaves to some vertex b. By the definition of G', there must be edges (a, a') and (b', b) in G with a' and b' in C. We can now create a path in G that replaces the edges $(a, C), (C, b)$ with (a, a'), the path from a' to b' along C and (b', b). This completes the proof of (1).

To prove (2), we can refer to the algorithm itself. During each iteration, if the remaining graph is DAG, then the algorithm will stop. Otherwise, the updated graph will have one less cycle with a decreasing number of vertices. The algorithm will not stop until all the cycles in the directed graph are replaced with singles vertices, thus the graph will end up with a DAG.

To prove (3), we need to refer to the definition of DAG. For any pairs of vertices (u, v) in a DAG G, if there exists a path from u to v, there can't be a path from v to u otherwise there will exist a directed cycle in the directed graph, which contradicts with the definition of DAG. This means that for any pairs of vertices (u, v) in a DAG, u and v are not strongly connected. As a result the metagraph of a DAG is the DAG itself.

From (1) and (2) and (3) we can ensure that the metagraph of G will never change during this iterative procedure. The algorithm will end up with a DAG, which is indeed a metagraph of the DAG itself and this metagraph is the same as the the metagraph of G. Thus we can conclude that this algorithm is correct.

Question 3 (Graph Cycle, 15 points)**.** *Is it the case that for every finite, strongly connected directed graph G that there is a cycle that visits each vertex of G at least once, but uses no edge more than once? Prove or provide a counter-example.*

Solution 3.

It may be verified that the graph shown in Figure 1 is strongly connected. Observe that any path that passes through v_1, v_2 and v_4 must pass through v_3 at least twice. The two edges pointing up are repeated in such a path, giving us a counter example.

Question 4 (Dijkstra at Small Distances, 30 points)**.** *Suppose that you are given a graph G with positive integer edge weights, a vertex s and an integer L. Give an algorithm that determines which other vertices w in G have paths from s to w of length at most L. For full credit your algorithm should run in time $O(|V| + |E| + L)$. Hint: You will want to devise some appropriate modification of Dijkstra's algorithm that takes advantage of the fact that you only need to keep track of distances that are less than L.*

Solution 4.

The key idea is to implement a version of Dijkstra using a new implementation of a priority queue. We will do this by having an array of size $L + 1$, where the i^{th} bucket will store a list of vertices currently at distance i. We can perform insert and decrease key in $O(1)$ time by placing the element in question in the right bucket or moving it to the new bucket. To delete min, we will need to scan through buckets to find the first non-empty one and return some vertex in that bucket. If done naively, this will take $O(L)$ time per delete min, which is too slow. We can improve this by noting that Dijkstra only finds new vertices in increasing order of distance, so we only need to start scanning from whatever the distance to the most recently discovered vertex was.

In the following pseudocode of the algorithm described above $d(v)$ represents the distance of node v from s and $w(u, v)$ represents the weight of edge (u, v).

```
DistanceAtMostL($s$):
    Initialize buckets $B_i$ corresponding to distances $i = 0, \dots, L$
    Initialize $d(v)$ of all nodes $v$ except $s$ as INF and distance of node $s$ as $0$ and add $s$ to $B_0$
    For $l = 0 \longrightarrow L$:
        For each node $u$ in $B_l$:
            For each neighbour $v$ of $u$:
                If $d(v) > l$ and $d(v) <= L$:
                    Update $d(v)$ to $l_v = min(d(v), l + w(u, v))$
                    Add/move node $v$ to bucket $B_{l_v}$
    return the union of the set of nodes in all the buckets $B_0, \dots, B_L$
```

Since every distance value in $0, \dots L$ is processed once and each node and edge is visited at most once in the above algorithm the time complexity is $O(L + |V| + |E|)$

For the proof of correctness, note that this approach is basically a modified version of the Djikstra's algorithm. Instead of using a priority queue to extract the node with the smallest distance to the source in each iteration, we are exploring nodes at increasing values of the distance from the source node. Therefore, when exploring nodes at distance l, it is guaranteed that all the nodes at distances lesser than l have been correctly identified and placed in the appropriate buckets.

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