ESO207A: Data Structures and Algorithms

Practice Set *5: Shortest Paths* –

Problem 1. Give a linear-time algorithm for the following problem. Given an undirected graph $G = (V, E)$ with unit edge weights and a distinct source vertex s, find the *number* of shortest paths from s to v, for every vertex $v \in V - \{s\}$.

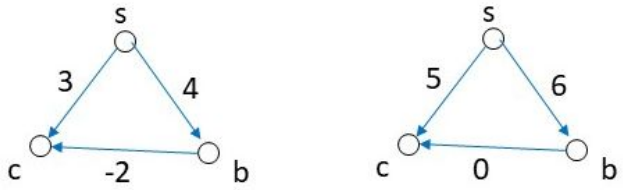
Solution. Modify BFS from s to associate a variable $v.count$ with every vertex v that counts the number of shortest paths from s to v. Initialize $s.count = 1$ and $v.count$ to 0 for every vertex $v \in V - \{s\}$. Also keep a variable $u.level$ which is the index i of the set L_i in which u is placed. Initially, $s.level = 0$ and $u.level = \infty$, for all $u \in V - \{s\}$. Now run BFS. $L_0 = \{s\}$. As we compute L_i, assume that $v.count$ is found for all vertices $v \in L_0 \cup \ldots \cup L_i$. Also set the variable $u.discovered = false$ initially for all vertices v, except for s–this is used by BFS to ensure vertices are not repeatedly explored. The pseudo-code is something like this.

```
level = 0
s.level = 0
s.count = 1
for every vertex u ∈ V − {s}
    v.count = 0
L = {s}
while L ≠ φ
    L' = {}
    for every vertex u ∈ L
        for vertex v adjacent to u
            if v.discovered == FALSE
                v.level = level + 1
                v.count = u.count
                v.discovered = TRUE
                v.parent = u
                L' = L ∪ {v}
            elseif v.level == level+1 // discovered at this level
                v.count = v.count + u.count // another shortest routes
    L = L'
    level = level + 1
```

Problem 2. Given a graph with possible negative weight edges, add a large enough positive constant to each edge weight so that it becomes positive, and now run Dijkstra's algorithm to compute shortest paths. Give a counterexample and also explain briefly.

Solution.

Say all edge weights are increased by $\Delta > 0$. Then all paths are not increased by the same amount. A path of length k is increased by $k\Delta$. Hence, if there is a path of k hops with cost c, its cost is now $c + k\Delta$, which for different k, is different. The minimum therefore may not be preserved in all cases.



(i) Original graph.

(ii) Graph obtained by increasing edge weights by 2 to make all weights non-negative.

Figure 1: The shortest path in the original graph from s to c is the path $s - b - c$ of weight 1. In figure (ii) the shortest path from s to c is the edge $s - c$ of weight 5, which reflects the wrong path in the original graph.

Problem 3. Given a directed graph $G = (V, E)$ with (possibly negative) edge weights along with a specific node $s \in V$ and a tree $T = (V, E')$, give a (linear-time) algorithm that checks whether T is the shortest path tree for G with starting point s.

Solution. Let's get obvious checks out of the way. 1. Check if $s.cost = 0$, if not, then there is a problem and the tree cannot be a shortest path tree. 2. The tree T should be a valid cost tree, that is, $E' \subset E$ and for every vertex $v \in V - \{s\}$, if $u = v.parent$, then, $v.cost$ must equal $u.cost + w(u, v)$. This also implies that $v.cost$ equals the weight of the tree path from s to v for all vertices v. If this fails for any vertex v, it is not a valid tree. All this is checked in $O(|V|)$ time. The main part of the check is the following. Relax all the edges of E in some order and for each edge relaxed, check if there is any reduction in the cost. If after one relaxation pass, there is no change, then conclude that T is a correct shortest path tree, and otherwise, conclude that T is not a shortest path tree. Pseudo-code is as follows.

SPTreeCheck(T, s)
1. **if** $s.cost \neq 0$
2. **return** "Not Shortest Path Tree"
3. Check if the tree T is a valid tree
4. **for** every edge $(u, v) \in E$
5. **if** $v.cost < u.cost + w(u, v)$
6. **return** "Not Shortest Path Tree"
7. **return** "Possible Shortest Path Tree"

If the relaxation of some edge (u, v) reduces the cost, then clearly, we have a possibly shorter path to v, which means T could not have been the shortest path tree. Now suppose the algorithm returns "Possible Shortest Path Tree". We show that T is indeed a shortest path tree.

We will show by induction on the length (number of hops) of the path from s to a vertex v that $\delta(s, v) = v.cost$. The base case occurs for path length of 0, where, $0 = \delta(s, s) = s.cost$. *Induction case.* Suppose there is a shortest path from s to v consisting of $k + 1$ hops and assume that the induction hypothesis is true for shortest paths from s to all vertices u consisting of k hops. Consider

a shortest path of $k+1$ hops from s to v. Let u be the penultimate vertex in this shortest weight path from s to v. Then, the part of this path from s to u is the shortest path from s to u and hence, by the induction hypothesis, $u.cost = \delta(s,u)$. When the edge (s,u) is relaxed, by the check, we have,

$$v.cost \leq u.cost + w(u,v) = \delta(s,u) + w(u,v) = \delta(s,v)$$

since the shortest path from s to v has penultimate vertex u. Since $v.cost$ is a valid cost of some path in G, therefore, $v.cost = \delta(s,v)$, which is what we wanted to prove.

Thus, the assertion is proved by induction, that is, if the algorithm returns "YES", the shortest path tree claimed is indeed correct.

Problem 4. Given a directed graph $G = (V,E)$ with (possibly negative) edge weights and an additional guarantee that between any two vertices there exists a shortest path of length k (no of hops), give an $O(k|E|)$ time algorithm to compute the single source shortest path problem.

Solution. Instead of running the Bellman Ford algorithm as $|V|-1$ passes of all edge relaxations, just run k passes, since shortest paths are of length no larger than k.

Problem 5. You are given a directed graph with positively weighted edges. Give an efficient algorithm that returns the length of the shortest cycle in the graph. (Can you do this in time $O(|V|^3)$?)

Solution. Given the graph in the adjacency list format, also make a copy of it in the adjacency matrix format. Let $w(u,v)$ double as the representation in the matrix format, where, $w(u,v) = \infty$ means that there is no edge from u to v. This takes time $O(|V|^2)$. Let $s = u_0, u_1, \ldots, u_k, u_0 = s$ be the shortest weight cycle in G. Then, the path $s, u_1, \ldots, u_k$ is the shortest path from s to u_k. So one strategy could be: find the single source shortest paths from s to every vertex v and check the cycle weight $\delta(s,v) + w(v,s)$. Keep the minimum over all $v \in V - \{s\}$. Since we are not sure what s is, we iterate over all vertices $s \in V$, and keep the global minimum cycle weight. We can use Dijkstra's algorithm, from all vertices s, takes $O(|V|\,(|V|+|E|)\log|V|)$ time. For each s, the check for the smallest cycle involving s takes $O(|V|)$ time, and so the total takes time $O(|V|^2)$. Overall time complexity is $O(|V|^2 \log|V| + |V|\,|E|\log|V|)$. The algorithm appears to be improvable.

The problem essentially uses Dijkstra's algorithm to solve the all pairs shortest path problem, that is find the shortest path and length from any source to any destination. This dominates the cost to be $|V|\,|E|\log|V|$ time. There is a general Floyd's algorithm that finds this is time $O(|V|^3)$. For some dense graphs, this is an improvement, and Floyd's works for graphs with negative edge weights as well (without negative cycles as usual).

The algorithm is classified as dynamic programming. It is based on the following logic. It orders the vertices as $v_1, v_2, \ldots, v_n$. Let P_{ij}^k denote the length (weight) of the shortest path from v_i to v_j passing only through intermediate vertices $v_1, v_2, \ldots, v_k$. Initially $P_{ij}^0 = w(i,j)$. The observation of the algorithm is the following recurrence relation. Consider the shortest path from v_i to v_j passing only through intermediate vertices $v_1, \ldots, v_{k+1}$. Now there are two possibilities. Either this shortest path passes through v_{k+1} or it does not. If it does not, then, $P_{ij}^{k+1} = P_{ij}^k$. If it does, then any such path passes through v_{k+1} exactly once. Because, if it passes twice, you get a cycle from v_{k+1} to v_{k+1} on the path, which can be cut out, and this can only reduce (or in case of zero length cycle, not increase) the cost of the original path. Thus, we get two segments of this path, the initial portion of the path from v_i to v_{k+1} passing only through $v_1, \ldots, v_k$ and the latter portion of the path from v_{k+1} to v_j passing only through $v_1, \ldots, v_k$. This gives the following recurrence

equation:

$$P_{ij}^{k+1} = \min(P_{ij}^{k}, P_{i,k+1}^{k} + P_{k+1,j}^{k}) \ .$$

This gives the following (simple) pseudo code.

AllPairsSP(G, w)
1. Set P_{ij} to $w(i,j)$ whenever j is adjacent to i
2. Set diagonal entries P_{ii} to 0
3. All other entries of P_{ij} are set to ∞
4. **for** $k = 1$ **to** n
5. **for** $i = 1$ **to** n
6. **for** $j = 1$ **to** n
7. $P_{ij} = \min(P_{ij}, P_{i,k+1} + P_{k+1,j})$

After computing all pairs shortest path, for every vertex $s \in V$, and $v \in V - \{s\}$, check the cycle length $P_{sv} + P_{sv}$ and keep the minimum. This runs in time $O(|V|^2)$ time and does not dominate. The time is dominated by Floyd's all pairs shortest path algorithm $O(|V|^3)$.

Problem 6. You are given an undirected graph $G = (V, E)$ with positive edge weights and an edge $e \in E$. Find the length of the shortest cycle containing the edge e.

Solution. Let G' be the graph obtained from G by deleting e. Let $e = \{u, v\}$. In G', use Dijkstra's algorithm to obtain the shortest path from u to v. Add e to this path–this gives the shortest cycle containing e.

Correctness: Consider the shortest cycle including $e = \{u, v\}$. Start the cycle from u and traverse e first and then onwards along the edges of the cycle back to u. Denote the cycle C as $u = u_0, v, v_1, v_2, \ldots, v_k$. The (reverse) path $u = u_0, v_k, \ldots, v_1, v$ is a shortest path from $u = u_0$ to v. Because if it weren't, there would be a shorter path and adding e to it would give a shorter weight cycle.