CMPSC 465 Data Structures & Algorithms
Fall 2025 Ke Chen and Yana Safonova

HW 6 Solution

1. (10 pts.) **Dijkstra with Reinsertion.** Let $G = (V, E)$ be a directed graph with weights $\ell : E \to \mathbb{R}$ and let $s \in V$ be the starting vertex. Consider the variant of Dijkstra that, after a vertex has been extracted, allows it to be reinserted into the priority queue if a later Update call reduces its distance from s.

(a) Does this algorithm correctly compute single-source shortest paths from s when negative edge weights are allowed but there are no negative cycles?

(b) Analyze the runtime complexity of the algorithm.

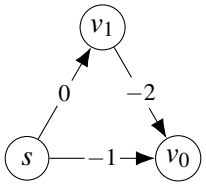
Solution.

(a) Dijkstra's usual correctness hinges on the invariant: when a vertex u is extracted, its tentative distance $d[u]$ equals the true shortest distance $\text{dist}(s,u)$; nonnegativity of edge weights guarantees that any other path to u found later cannot be shorter. Allowing reinsertion removes the invariant as an extracted vertex may have its tentative distance decreased later through a path that includes negative-weight edges, so extraction no longer implies finality.

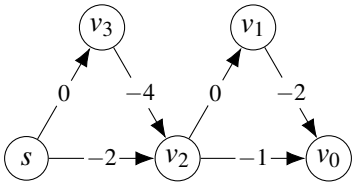
If the algorithm is implemented as described (allowing reinserts), every time an Update call decreases $d[v]$ we push v into the priority queue again and continue until no further Updates are possible. In the absence of negative cycles reachable from s this process does converge to the correct shortest distances because when the algorithm terminates no edge can be used for Update and the recorded vector d satisfies the triangle inequalities $d[v] \le d[u] + \ell(u,v)$ for all $u, v \in V$. (Otherwise, it is possible to call Update on the edge (u,v) to improve $d[v]$.)

For the sake of contradiction, assume there is some vertex $u \in V$ whose shortest distance from s is not correctly computed, namely $d[u] > dist(s,u)$. Consider a shortest path from s to u, let x be the first vertex on this path whose shortest distance from s is incorrect, i.e., $d[x] > dist(s,x)$. Note that since $d[s] = dist(s,s) = 0$ and $d[u] > dist(s,u)$, such a vertex x must exist and it's not the first node on this path. Let y be the predecessor of x on this path. By our choice of x, $d[y]$ must have been correctly computed, namely $d[y] = dist(s,y)$. Now we have $d[x] > dist(s,x) = dist(s,y) + \ell(y,x) = d[y] + \ell(y,x) \ge d[x]$, which simplifies to $d[x] > d[x]$, a contradiction.

(b) In the worst case, this version of Dijkstra's algorithm can take exponential time. Consider the following graph:

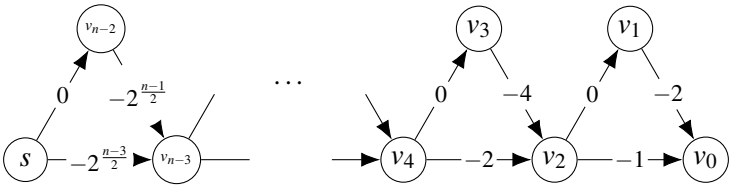


Starting from node s, we would first extract v_0, and then v_1 which causes a shorter path to v_0. So in total, we extract v_1 once and extract v_0 twice from the priority queue. Now consider expanding the graph as follows:



Again starting from node s, we would first extract v_2, followed by extracting v_0 twice and v_1 once, just like in the previous graph; then we extract v_3, which causes a shorter path to v_2, so we must again extract v_0 twice and v_1 once. In total, we extracted v_3 once, v_2 twice, v_1 twice, and v_0 four times.

It is not hard to see (or proof formally with induction), attaching another triangle by adding two more vertices would double the number of priority queue extractions for each node in the previous graph (plus the two new nodes will be extracted once and twice, respectively). So for such a graph with n nodes (n is odd and ≥ 3, see figure below, due to Douglas R Shier and Christoph Witzgall[1]), Dijkstra's algorithm needs to delete the min from the priority queue $3(2^{\frac{n-1}{2}} - 1) = \Omega(2^{\frac{n}{2}})$ times.



2. (20 pts.) **Budget Bellman-Ford** Given a directed graph $G = (V, E)$, a weight function of the edges $\ell : E \to \mathbb{R}$ (you can assume there is no negative cycles in G), a starting vertex s, and an integer budget $k \geq 0$, we would like to compute for every vertex $v \in V$ the shortest path from s to v that **uses at most k edges**. Give an algorithm for this problem, argue for its correctness, and analyze its time complexity.

Solution. To find shortest paths that use at most k edges, we can modify the Bellman-Ford algorithm to run for exactly k rounds.

However, a subtle issue arises if we update $dist[v]$, the current shortest distance from s to v, using edges that themselves were just updated in the same round. For example, suppose in the first round we call Update on edge (s,u) and obtain a smaller value for $dist[u]$. If we then call Update on edge (u,v) in the same round, dist$[v]$ may be further decreased using the path $s \to u \to v$, which already uses two edges. As a result, after k rounds of Bellman-Ford, the shortest paths recorded in $dist$ may actually use more than k edges.

To correctly restrict paths to at most k edges, we must ensure that each round only uses edges whose starting vertices' distances were computed in the previous round. This can be done by maintaining a separate copy of the distance array from the previous iteration, denoted as old_dist. In each round, we use old_dist to perform all edge Updates and store the updated values in $dist$. Continuing the previous example, even if $dist[u]$ is improved by calling Update on (s,u), when Update (u,v), we still refer to $old\_dist[u]$. This ensures that after k rounds, the resulting $dist[v]$ values correspond to the shortest paths that use at most k edges.

The correctness of this algorithm follows from the standard Bellman-Ford argument. For any vertex $v \in V$, suppose the shortest path from s to v that uses at most k edges consists of edges $e_1, e_2, \ldots, e_p \in E$ in this order, where $p \leq k$. As we discussed in class, this shortest path can be correctly computed if the Update procedure is applied to these edges in this order (possibly interleaved with other Update calls).

Since in each round of Bellman-Ford we perform Update on every edge, the edge e_1 will be used for Updated in the first round, e_2 in the second round, and so on. Because $p \leq k$, after k rounds we are guaranteed to have

[1] `https://pmc.ncbi.nlm.nih.gov/articles/PMC6756282/#S5`

called Update on all edges $e_1, \ldots, e_p$ in order, and thus correctly compute the shortest distance to v using at most k edges.

Furthermore, by our modification of maintaining a separate *old_dist* array, each round can only use distance values from the previous round. This ensures that the shortest paths discovered in round i use at most i edges. Consequently, after k rounds, the algorithm correctly computes the shortest distances corresponding to paths that use at most k edges.

Each round of the algorithm performs one Update call on every edge, which takes $O(|E|)$ time. Additionally, copying the *dist* array into *old_dist* takes $O(|V|)$ time per round. Since the algorithm runs for k rounds in total, the overall running time is $O(k(|V|+|E|))$.

3. (30 pts.) **Ordered Bellman-Ford** Bellman–Ford performs $|V|-1$ rounds of Update calls on all edges; a practical improvement is to stop early when no Update takes effect in the previous round.

The following variant aims to reduce the number of rounds required: in each round, call Update on the edges in nondecreasing order of their weights, so that edges with smaller weights are updated earlier.

(a) Does this variant still correctly compute the shortest distances from the starting vertex s?

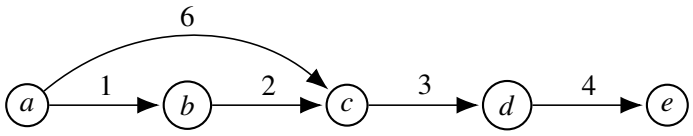
(b) Give an example where this variant reduces the number of rounds required (namely, there is some order of updates that needs $|V|-1$ rounds but this method requires fewer), or argue that no such instance exists.

(c) Give an instance where this variant does not reduce the number of rounds required (namely, this method needs $|V|-1$ updates), or argue that no such instance exists.

Solution.

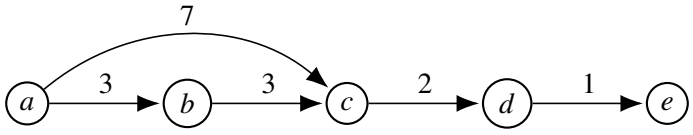
(a) Yes, as we saw in the lecture, the order of Update calls in each round does not matter.

(b) Lets consider the following graph:



The standard Bellman-Ford algorithm needs up to $|V|-1=4$ rounds for guaranteed convergence in this graph. However, by processing the edges in non-decreasing order of weights: (a,b) (1), then (b,c) (2), (c,d) (3), (d,e) (4), and finally (a,c) (6), the shortest-path distances propagate efficiently. Specifically, the processing of (a,b) sets $dist(b)$ to 1. Then, immediately within the same round, the processing of (b,c) uses this new, shorter $dist(b)$ to correctly set $dist(c)$ to 3. This pattern continues along the path $a \to b \to c \to d \to e$. Because the distance to every node is correctly finalized within this single pass, the algorithm will detect no further updates in the next round and terminate after just 1 round of effective updates, drastically outperforming the theoretical 4-round limit.

(c) Lets consider the following graph:



The shortest path from a to e is the simple chain $a \to b \to c \to d \to e$. The sorted order of edges by non-decreasing weight is (d,e) (1), (c,d) (2), (a,b) (3), (b,c) (3), and (a,c) (7). In Round 1, the shortest

path begins with the edges (d,e) and (c,d) being processed first, but they have no effect because their source nodes (d and c) have not yet been reached ($dist(d) = \infty, dist(c) = \infty$). The propagation must start at a. The distance to node e requires $|V| - 1 = 4$ sequential updates (one for each edge in the four-hop path). Since Bellman-Ford can only propagate the distance one hop further in each full round, $|V| - 1 = 4$ full rounds are required to correctly set $dist(e)$, forcing the variant to match the theoretical worst-case bound.

4. (20 pts.) **Pinned Nodes** Given a directed graph $G = (V,E)$, with n vertices (namely, $n = |V|$), a weight function on edges $\ell : E \to \mathbb{R}$ (you can assume the graph has no negative cycles), and a subset $S \subseteq V$ of special vertices, the task is to compute $d(i,j)$, the length of the shortest path from i to j that **includes at least one vertex from** S, for all pairs of vertices $(i,j) \in V \times V$. If no such path exists, the length is defined to be ∞.

(a) Design an algorithm, based on the Floyd–Warshall method (or using it as a subroutine), that computes $d(i,j)$ as defined above for all pairs (i,j).

(b) Prove the correctness of your algorithm and analyze its time and space complexity.

Solution.

(a) To compute the shortest path from i to j that passes through at least one vertex in S, we can first execute the standard Floyd-Warshall algorithm to obtain the all-pairs shortest path distances $dist[i,j]$. Then, we use a post-processing step to compute the desired distances. For each pair (i,j), the shortest path that passes through at least one vertex in S can be expressed as:

$$d(i,j) = \min_{s \in S}(dist[i,s] + dist[s,j])$$

The expression considers all possible special vertices $s \in S$ and selects the one that yields the minimum total path length. If $dist[i,s] = \infty$ or $dist[s,j] = \infty$ for a given s, then that vertex s does not contribute a valid path. If no valid path exists through any $s \in S$, then $d(i,j) = \infty$.

The algorithm is presented in pseudocode below:

(b) The correctness of the algorithm follows from the fact that the standard Floyd-Warshall algorithm computes the shortest paths $dist[i,s]$ and $dist[s,j]$ for all vertices i, j, s. The post-processing step ensures that for each pair (i,j), we consider all possible special vertices $s \in S$ to find the minimum path length that includes at least one vertex in S. If no such path exists (i.e., $dist[i,s] + dist[s,j] = \infty$ for all $s \in S$), the distance $d[i,j]$ remains ∞, as required.

The standard Floyd-Warshall algorithm runs in $O(n^3)$ time to compute $dist[i,j]$ for all pairs. The post-processing step requires iterating over all pairs (i,j) and, for each pair, computing the minimum over all $s \in S$, which takes $O(n^2|S|)$ time. Thus, the total time complexity is $O(n^3 + n^2|S|) = O(n^3)$. The space complexity is $O(n^2)$, as both the distance matrix $dist$ and the output matrix d are both of size $n \times n$.

5. (20 pts.) **Bottleneck Path** Consider a directed weighted graph $G = (V,E)$, where V is a set of n vertices, and $E \subseteq V \times V$ is a set of directed edges, each associated with a non-negative weight $w : E \to \mathbb{R}_{\geq 0}$. The bottleneck weight of a path from vertex i to vertex j is defined as the maximum weight of any edge in the path. The objective is to compute, for each pair of vertices $(i,j) \in V \times V$, the path from i to j with the minimum bottleneck weight. If no path exists from i to j, the bottleneck weight is defined to be ∞. Formally, for each pair (i,j), we seek the value $b(i,j)$, defined as:

Algorithm 1 Floyd-Warshall for Paths via Special Vertices

Input: Directed graph $G = (V, E)$, weights $\ell : E \to \mathbb{R}$, special vertices $S \subseteq V$, number of vertices n.
Output: Matrix d where $d[i, j]$ is the shortest path from i to j passing through at least one vertex in S.
Initialize $dist[i, j] \leftarrow \ell(i, j)$ if $(i, j) \in E$, 0 if $i = j$, ∞ otherwise.
for $k = 1$ to n **do**
    for $i = 1$ to n **do**
        for $j = 1$ to n **do**
            $dist[i, j] \leftarrow \min(dist[i, j], dist[i, k] + dist[k, j])$
        end for
    end for
end for
Initialize $d[i, j] \leftarrow \infty$ for all i, j.
for $i = 1$ to n **do**
    for $j = 1$ to n **do**
        for $s \in S$ **do**
            $d[i, j] \leftarrow \min(d[i, j], dist[i, s] + dist[s, j])$
        end for
    end for
end for
Return d

$$b(i, j) = \min_{\substack{\text{path} \\ p: i \to j}} \left(\max_{(u,v) \in p} w(u, v) \right),$$

where the minimum is taken over all simple paths p from i to j, and $\max_{(u,v)\in p} w(u, v)$ is the bottleneck weight of a path p. Design an algorithm based on the Floyd-Warshall algorithm to find minimum bottleneck weight for all pairs of vertices in the graph. Argue for its correctness and analyze its time complexity.

Solution

To solve the bottleneck path problem, we modify the Floyd-Warshall algorithm to compute the minimum bottleneck weight instead of the minimum sum of edge weights. Define a matrix $B^{(k)}[i, j]$, which represents the minimum bottleneck weight of any path from i to j using only vertices $\{1, 2, \ldots, k\}$ as intermediate vertices. The bottleneck weight of a path is the maximum weight of any edge in the path, and we aim to minimize this value over all possible paths.

The recurrence relation for $B^{(k)}[i, j]$ is based on the observation that a path from i to j either does not use vertex k as an intermediate vertex, in which case the bottleneck weight is $B^{(k-1)}[i, j]$, or it uses vertex k, forming a path $i \to k \to j$. In the latter case, the bottleneck weight of the path $i \to k \to j$ is the maximum of the bottleneck weight from i to k and from k to j. Thus, we take the minimum of these two options:

$$B^{(k)}[i, j] = \min(B^{(k-1)}[i, j], \max(B^{(k-1)}[i, k], B^{(k-1)}[k, j])).$$

The initialization is similar to the standard Floyd-Warshall algorithm, but it reflects the bottleneck objective:

$$B^{(0)}[i, j] = \begin{cases} w(i, j) & \text{if } (i, j) \in E, \\ 0 & \text{if } i = j, \\ \infty & \text{if } (i, j) \notin E \text{ and } i \neq j. \end{cases}$$

This ensures that direct edges have their weights as the initial bottleneck values, self-loops have a bottleneck weight of 0 (as they involve no edges), and nonexistent edges have an infinite bottleneck weight. After n iterations, $B^{(n)}[i,j]$ gives the minimum bottleneck weight $b(i,j)$ for the path from i to j.

The algorithm is presented in pseudocode below:

Algorithm 2 Modified Floyd-Warshall for Bottleneck Paths

Input: Directed graph $G = (V,E)$, weights $w : E \to \mathbb{R}_{\geq 0}$, number of vertices n.
Output: Matrix b where $b[i,j]$ is the minimum bottleneck weight of any path from i to j.
Initialize $B^{(0)}[i,j] \leftarrow w(i,j)$ if $(i,j) \in E$, 0 if $i = j$, ∞ otherwise.
for $k = 1$ to n **do**
    for $i = 1$ to n **do**
        for $j = 1$ to n **do**
            $B^{(k)}[i,j] \leftarrow \min(B^{(k-1)}[i,j], \max(B^{(k-1)}[i,k], B^{(k-1)}[k,j]))$
        end for
    end for
end for
Return $B^{(n)}$

The correctness of the algorithm follows from the fact that it considers all possible intermediate vertices and selects the path with the minimum bottleneck weight. For each k, the matrix $B^{(k)}[i,j]$ represents the optimal bottleneck weight using vertices up to k. The use of the max operator in the recurrence ensures that the bottleneck weight of the path through k is correctly computed, and the min operator ensures that we select the path with the smallest bottleneck weight.

The time complexity is $O(n^3)$, as the algorithm iterates over n vertices and updates an $n \times n$ matrix with constant-time operations for each entry. The space complexity is $O(n^3)$, if we store all the matrices $B^{(k)}$. But just like for the original Floyd-Warshall, the space usage can be reduced to $O(n^2)$ by storing a single $n \times n$ matrix and update it in place. This is safe as the value of $B^{(k)}[i,j]$ depends only on values from $B^{(k-1)}$.

Rubric:

Problem 1, ? pts

?

Problem 2, ? pts

?

Problem 3, ? pts

?

Problem 4, ? pts

?

Problem 5, ? pts

?