Evelyn De La Rosa November 1, 2024

6.1210 Problem Set 7

Problem 1 *(Collaborators: Mechanical)*

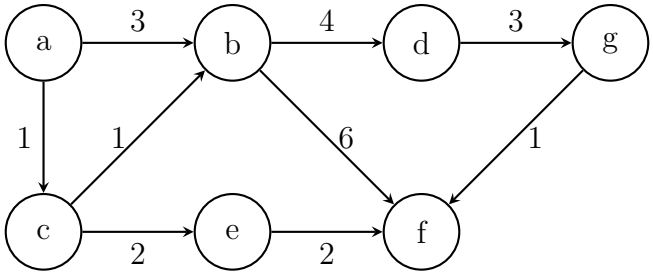
Given a directed, weighted graph $G = (V, E, w)$ with nonnegative edge weights, perform the following tasks:

(a) Run both DAG Relaxation and Dijkstra's algorithm on G, starting from vertex a. For each algorithm, provide a list of the edges that the algorithm relaxes, in the order they are relaxed. If there is any ambiguity in which node to process next, use alphabetical order to break the tie.

(b) In no more than two sentences, explain how DAG Relaxation relates to the Bellman-Ford algorithm.

The graph G is represented as follows:

```
graph = {
    'a': {'b': 3, 'c': 1},
    'b': {'d': 4, 'f': 6},
    'c': {'b': 1, 'e': 2},
    'd': {'g': 3},
    'e': {'f': 2},
    'f': {},
    'g': {'f': 1}
}
```



Solution

(a) Relaxation Order for DAG Relaxation and Dijkstra's Algorithm

DAG Relaxation

Algorithm Overview: In a Directed Acyclic Graph (DAG), the shortest paths can be efficiently found by performing a topological sort of the vertices and then relaxing the edges in the order determined by this sort.

Steps:

Step 1. Topological Sort: Determine a linear ordering of the vertices such that for every directed edge $u \to v$, vertex u comes before v in the ordering.

Step 2. Edge Relaxation: Iterate through the vertices in the topological order and relax all outgoing edges of each vertex.

Topological Order: One possible topological ordering for the given graph is:

$$a \to c \to b \to e \to d \to g \to f$$

Relaxation Order: Following the topological order, the edges are relaxed in the following sequence:

1. Relax (a, b)
2. Relax (a, c)
3. Relax (c, b)
4. Relax (c, e)
5. Relax (b, d)
6. Relax (b, f)
7. Relax (e, f)
8. Relax (d, g)
9. Relax (g, f)

Justification: By processing the vertices in topological order, each edge is relaxed exactly once, ensuring that all dependencies are respected and the shortest paths are correctly computed without the need for multiple passes.

Dijkstra's Algorithm

Algorithm Overview: Dijkstra's algorithm finds the shortest paths from a source vertex to all other vertices in a graph with nonnegative edge weights. It repeatedly selects the vertex with the smallest tentative distance, relaxes its outgoing edges, and marks it as processed.

Steps:

Step 1. Initialization: Set the distance to the source vertex a as 0 and all other vertices as ∞.

Step 2. Priority Queue: Use a priority queue to select the vertex with the smallest tentative distance. If there is a tie, select the vertex that comes first alphabetically.

Step 3. Edge Relaxation: For the selected vertex, relax all its outgoing edges.

Step 4. Repeat: Continue until all vertices have been processed.

Relaxation Order: Following the execution of Dijkstra's algorithm, the edges are relaxed in the following sequence:

1. Relax (a, c) *(Updates* d(c) = 1*)*
2. Relax (a, b) *(Updates* d(b) = 3*)*
3. Relax (c, b) *(Updates* d(b) = 2*)*
4. Relax (c, e) *(Updates* d(e) = 3*)*
5. Relax (b, d) *(Updates* d(d) = 6*)*
6. Relax (b, f) *(Updates* d(f) = 8*)*
7. Relax (e, f) *(Updates* d(f) = 5*)*
8. Relax (d, g) *(Updates* d(g) = 9*)*
9. Relax (g, f) *(No update,* d(f) = 5 *remains)*

Justification: Dijkstra's algorithm processes vertices in order of increasing tentative distance, ensuring that once a vertex is processed, its shortest path is finalized. The alphabetical tie-breaker ensures a deterministic order when multiple vertices have the same tentative distance.

(b) Relation between DAG Relaxation and Bellman-Ford Algorithm

DAG Relaxation and the Bellman-Ford algorithm both utilize edge relaxation to compute shortest paths. However, while Bellman-Ford repeatedly relaxes all edges up to $|V|-1$ times to handle graphs with cycles, DAG Relaxation leverages a topological ordering to relax each edge exactly once, making it more efficient for acyclic graphs.

Problem 2 *(Collaborators: PEST Scheduling)*

As a busy MIT student, you're hoping to schedule n problem sets to complete in the coming week by determining a start time s_i at which you will start each assignment. Due to some assignments depending on each other, some restrictions will require you to start some assignments a certain amount before or after another assignment. These restrictions take the form $s_i - s_j \leq t_k$ for some time t_k, and there are m of them. Note that the times, both the start times s_i and restriction times t_k, may be negative.

For example, if $s_i - s_j \leq -2$, it means the assignment s_j must be started at least two time steps after starting s_i. However, it might be impossible to schedule your problem sets. Consider the following example with 4 classes and 4 restrictions. Summing up the 4 inequalities, we find that the left side simplifies to 0 while the right side sums to -2; $0 \leq -2$ is a contradiction, so this is impossible.

$$
\begin{aligned}
s_1 - s_2 &\leq 7 \\
s_2 - s_3 &\leq -2 \\
s_3 - s_4 &\leq -10 \\
s_4 - s_1 &\leq 3
\end{aligned}
$$

(a) Finding a Feasible Assignment of Start Times

Given:

$$
\begin{aligned}
s_2 - s_1 &\leq -4 \\
s_3 - s_1 &\leq -15 \\
s_1 - s_4 &\leq 7 \\
s_4 - s_3 &\leq 10 \\
s_3 - s_2 &\leq -8
\end{aligned}
$$

Approach: Start by assigning $s_1 = 0$. Then, determine the upper and lower bounds

for the other s_i based on the given inequalities.

Step-by-Step Solution:

Step 1. Assign $s_1 = 0$:

$$s_1 = 0$$

Step 2. From $s_2 - s_1 \leq -4$:

$$s_2 - 0 \leq -4 \implies s_2 \leq -4$$

Upper Bound: $s_2 \leq -4$

Step 3. From $s_3 - s_1 \leq -15$:

$$s_3 - 0 \leq -15 \implies s_3 \leq -15$$

Upper Bound: $s_3 \leq -15$

Step 4. From $s_1 - s_4 \leq 7$:

$$0 - s_4 \leq 7 \implies -s_4 \leq 7 \implies s_4 \geq -7$$

Lower Bound: $s_4 \geq -7$

Step 5. From $s_4 - s_3 \leq 10$:

$$s_4 - s_3 \leq 10 \implies s_4 \leq s_3 + 10$$

Given $s_3 \leq -15$:

$$s_4 \leq -15 + 10 = -5$$

Upper Bound: $s_4 \leq -5$

Step 6. Combining the bounds for s_4:

$$-7 \leq s_4 \leq -5$$

Choose $s_4 = -5$ (within bounds):

$$s_4 = -5$$

Step 7. From $s_3 - s_2 \leq -8$:

$$s_3 - s_2 \leq -8 \implies s_3 \leq s_2 - 8$$

Given $s_2 \leq -4$:

$$s_3 \leq -4 - 8 = -12$$

But previously, $s_3 \leq -15$, so the tighter bound is $s_3 \leq -15$. **Assign $s_3 = -15$ (within bounds):**

$$s_3 = -15$$

Step 8. Determine s_2 from $s_3 \leq s_2 - 8$:

$$-15 \leq s_2 - 8 \implies s_2 \geq -15 + 8 = -7$$

Combining with $s_2 \leq -4$:

$$-7 \leq s_2 \leq -4$$

Choose $s_2 = -4$ (within bounds):

$$s_2 = -4$$

Final Assignment:

$$\begin{aligned} s_1 &= 0 \\ s_2 &= -4 \\ s_3 &= -15 \\ s_4 &= -5 \end{aligned}$$

Verification:

$$\begin{aligned} &s_2 - s_1 = -4 - 0 = -4 \leq -4 \quad \text{(Satisfied)} \\ &s_3 - s_1 = -15 - 0 = -15 \leq -15 \quad \text{(Satisfied)} \\ &s_1 - s_4 = 0 - (-5) = 5 \leq 7 \quad \text{(Satisfied)} \\ &s_4 - s_3 = -5 - (-15) = 10 \leq 10 \quad \text{(Satisfied)} \\ &s_3 - s_2 = -15 - (-4) = -11 \leq -8 \quad \text{(Satisfied)} \end{aligned}$$

All restrictions are satisfied with the assigned start times.

(b) Representing Assignments and Restrictions Using a Graph

Graph Representation:

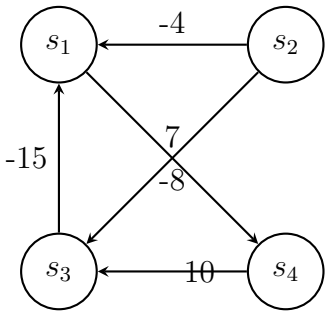
Assignments and their restrictions can be represented using a **directed graph**, where:

- **Vertices** represent the assignments $s_1, s_2, \ldots, s_n$.
- **Edges** represent the restrictions $s_i - s_j \leq t_k$. Specifically, an edge from vertex j to vertex i with weight t_k corresponds to the inequality $s_i - s_j \leq t_k$.

Explanation:

This representation aligns with the concept of **constraint graphs** used in scheduling and systems of inequalities. An edge $j \to i$ with weight t_k ensures that the start time s_i does not exceed $s_j + t_k$, thereby encoding the restriction $s_i - s_j \leq t_k$.

Graphical Representation:



Interpretation of the Graph:

- The edge $s_2 \to s_1$ with weight -4 represents $s_1 - s_2 \leq -4$.
- The edge $s_3 \to s_1$ with weight -15 represents $s_1 - s_3 \leq -15$.
- The edge $s_1 \to s_4$ with weight 7 represents $s_4 - s_1 \leq 7$.

- The edge $s_4 \to s_3$ with weight 10 represents $s_3 - s_4 \leq 10$.
- The edge $s_2 \to s_3$ with weight -8 represents $s_3 - s_2 \leq -8$.

(c) Algorithm to Determine Feasibility of Start Times

Objective: Provide an algorithm that either finds a feasible assignment of start times s_i for all assignments or states that no such assignment exists. The algorithm should run in $O(mn)$ time.

Approach: Model the problem as a **system of difference constraints** and represent it using a **constraint graph** as described in part (b). Utilize the **Bellman-Ford algorithm** to detect negative-weight cycles, which correspond to infeasible schedules.

Algorithm Steps:

Step 1. Construct the Constraint Graph:

- Create a vertex for each assignment $s_1, s_2, \ldots, s_n$.
- Add a **dummy vertex** s_0 to serve as the source.
- For each constraint $s_i - s_j \leq t_k$, add a directed edge from s_j to s_i with weight t_k.
- Add edges from s_0 to every other vertex with weight 0 to initialize the distances.

Step 2. Initialize Distances:

$$d(s_0) = 0$$
$$d(s_i) = \infty \quad \forall i \in \{1, 2, \ldots, n\}$$

Step 3. Relax Edges Using Bellman-Ford:

- Repeat n times:

  For each edge $u \to v$ with weight w, update $d(v) = \min(d(v), d(u) + w)$

Step 4. Check for Negative-Weight Cycles:

- For each edge $u \to v$ with weight w:

$$\text{If } d(v) > d(u) + w, \text{ then a negative-weight cycle exists.}$$

Step 5. Determine Feasibility:

- If no negative-weight cycles are detected, the distances $d(s_i)$ provide feasible start times.
- If a negative-weight cycle is detected, no feasible assignment exists.

Algorithm Implementation:

Input: Number of assignments n, number of constraints m, list of constraints (i, j, t_k) representing $s_i - s_j \leq t_k$
Output: Feasible start times or report infeasibility
1. Construct the Constraint Graph:;
Initialize adjacency list with $n + 1$ vertices (including s_0);
for *each constraint* (i, j, t_k) *in constraints* **do**
  Add edge from s_j to s_i with weight t_k;
end
for *each vertex* i *from 1 to* n **do**
  Add edge from s_0 to s_i with weight 0;
end
2. Initialize Distances:;
Initialize $d(s_0) = 0$;
for *each vertex* s_i *where* $i = 1$ *to* n **do**
  Set $d(s_i) = \infty$;
end
3. Relax Edges Repeatedly (Bellman-Ford):;
for *each vertex* v *from 1 to* n **do**
  for *each edge* $u \to v$ *with weight* w **do**
    if $d(u) + w < d(v)$ **then**
      Set $d(v) = d(u) + w$;
    end
  end
end
4. Check for Negative-Weight Cycles:;
for *each edge* $u \to v$ *with weight* w **do**
  if $d(u) + w < d(v)$ **then**
    return *"No feasible assignment exists."*;
  end
end
5. Assign Start Times:;
for *each vertex* s_i *where* $i = 1$ *to* n **do**
  Assign $s_i = d(s_i)$;
end
return *Start times* $s_1, s_2, \ldots, s_n$;

Algorithm 1: PEST Scheduling Algorithm

Explanation of Correctness:

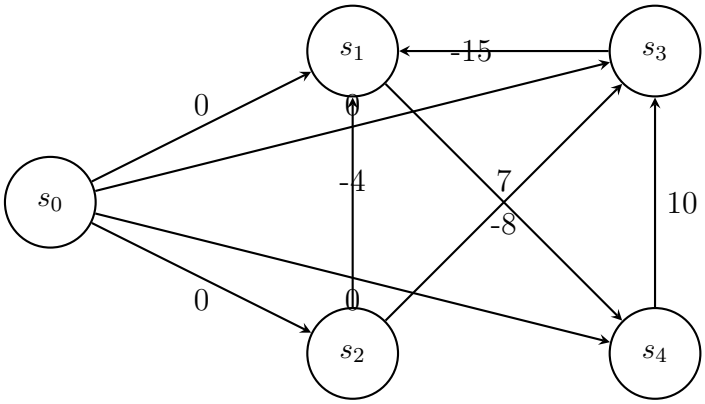
- **Constraint Graph:** Each constraint $s_i - s_j \leq t_k$ is represented as an edge $s_j \to s_i$ with weight t_k. This ensures that s_i does not exceed $s_j + t_k$.
- **Bellman-Ford Algorithm:** By initializing a dummy vertex s_0 and adding zero-weight edges to all other vertices, we ensure that all vertices are reachable. The Bellman-Ford algorithm computes the shortest paths from s_0 to all other vertices while detecting negative-weight cycles.
- **Negative-Weight Cycles:** The presence of a negative-weight cycle implies that there is a sequence of constraints that cannot be satisfied simultaneously, rendering the scheduling impossible.
- **Feasibility:** If no negative-weight cycles are detected, the shortest path distances $d(s_i)$ correspond to feasible start times that satisfy all constraints.

Time Complexity Analysis:

- **Graph Construction:** $O(m + n)$
- **Bellman-Ford Execution:** $O(mn)$ (since it relaxes all m edges n times)
- **Cycle Detection and Assignment:** $O(m)$

Thus, the overall time complexity is $O(mn)$, satisfying the problem requirements.

Graphical Illustration of Constraints:



Interpretation:

- The dummy vertex s_0 ensures that all assignments are reachable.
- The edges represent the constraints, with their respective weights indicating the maximum allowable difference between start times.
- A feasible assignment exists if and only if there are no negative-weight cycles in this graph.

Conclusion:

By modeling the scheduling problem as a constraint graph and utilizing the Bellman-Ford algorithm, we can efficiently determine whether a feasible assignment of start times exists. If the algorithm detects a negative-weight cycle, it conclusively indicates that no such assignment is possible.

Problem 3 *(Collaborators: Pilfering Purrloin [35 points])*

Purrloin is exploring the Unova Region. There are n cities connected by $O(n)$ directed roads. Each road from city u to city v has a cost $c(u, v)$, which can be positive (tolls) or negative (coins stolen). Every city is reachable from every other city. Some cities are designated as Pokemon Centers, and Purrloin starts at the **Nimbasa City Pokemon Center**. He must start each day at a Pokemon Center and cannot lose (net loss) more than k coins in a single day.

(a) [20 points] Purrloin wants to explore as many cities as possible (possibly over many days) and then end up back at the Nimbasa City Pokemon Center. Describe an $O(n^2 \log n)$ algorithm to find all cities that Purrloin can explore. Briefly analyze its runtime and justify correctness.

Solution:

Objective: Identify all cities Purrloin can explore and return to the Nimbasa City Pokemon Center without incurring a net loss exceeding k coins on any single day.

Approach: Since the graph may contain negative edge weights (coins stolen), and Dijkstra's algorithm cannot handle negative edge weights, we need to adjust our approach. We can use **Johnson's algorithm** to reweight the graph, eliminating negative edge weights, and then use Dijkstra's algorithm to compute shortest paths efficiently.

Algorithm Steps:

Step 1. Graph Representation: Represent the cities as vertices V and roads as directed edges E with associated costs $c(u, v)$.

Step 2. Add Super Node and Run Bellman-Ford: Introduce a super node s' connected to all nodes in V with edges of cost 0. Run the Bellman-Ford algorithm from s' to compute the potential $h(v)$ for all $v \in V$.

Step 3. Reweight Edges: For each edge $(u, v) \in E$, set the new cost $c'(u, v) = c(u, v) + h(u) - h(v)$. This ensures all $c'(u, v) \geq 0$.

Step 4. Run Dijkstra's Algorithm: Run Dijkstra's algorithm from the Nimbasa City Pokemon Center s to compute the shortest paths $d'(s, v)$ in the reweighted graph.

Step 5. Adjust Distances: Convert the reweighted distances back to original costs:

$$d(s, v) = d'(s, v) - h(s) + h(v)$$

Step 6. Compute Return Paths: Run Dijkstra's algorithm from each city v back to s in the reweighted graph to compute $d'(v, s)$. Adjust distances back:

$$d(v, s) = d'(v, s) - h(v) + h(s)$$

Step 7. Determine Reachable Cities: For each city v, if $d(s, v) + d(v, s) \leq k$, then Purrloin can explore v and return to s within the daily loss limit.

Pseudocode:

Input: Number of cities n, number of roads m, list of roads as tuples $(u, v, c(u, v))$, set of Pokemon Centers P, daily loss limit k
Output: Set of reachable cities
1. Construct the Graph:;
Initialize adjacency list *adj* for all $n + 1$ vertices (including super node s');
for *each road* $(u, v, c(u, v))$ *in roads* **do**
  Add edge from u to v with cost $c(u, v)$;
end
for *each Pokemon Center* p *in* P **do**
  Add edge from super node s' to p with cost 0;
end
2. Run Bellman-Ford from s'**:**;
Initialize $d[s'] = 0$ and $d[v] = \infty$ for all $v \neq s'$;
for *each vertex* $k = 1$ *to* n **do**
  for *each edge* $(u, v, c(u, v))$ *in* E **do**
    if $d[u] + c(u, v) < d[v]$ **then**
      Set $d[v] = d[u] + c(u, v)$;
    end
  end
end
3. Reweight Edges:;
for *each edge* $(u, v, c(u, v))$ *in* E **do**
  Set $c'(u, v) = c(u, v) + d[u] - d[v]$;
end

4. Run Dijkstra's Algorithm from s:;
Initialize priority queue Q and distance array $D[s][v] = \infty$ for all v;
Set $D[s][s] = 0$;
Insert s into Q with priority 0;
while *Q is not empty* **do**
  Extract vertex u with minimum $D[s][u]$ from Q;
  for *each edge $(u, v, c'(u, v))$ in $adj[u]$* **do**
    if $D[s][u] + c'(u, v) < D[s][v]$ **then**
      Set $D[s][v] = D[s][u] + c'(u, v)$;
      Insert v into Q with priority $D[s][v]$;
    end
  end
end
5. Adjust Distances Back to Original Weights:;
for *each city v* **do**
  $d(s, v) = D[s][v] - h(s) + h(v)$;
end
6. Run Dijkstra's Algorithm from Each v to s:;
for *each city v where $d(s, v) \leq k$* **do**
  Initialize priority queue Q_v and distance array $D[v][s] = \infty$;
  Set $D[v][v] = 0$;
  Insert v into Q_v with priority 0;
  while *Q_v is not empty* **do**
    Extract vertex u with minimum $D[v][u]$ from Q_v;
    for *each edge $(u, s, c'(u, s))$ in $adj[u]$* **do**
      if $D[v][u] + c'(u, s) < D[v][s]$ **then**
        Set $D[v][s] = D[v][u] + c'(u, s)$;
        Insert s into Q_v with priority $D[v][s]$;
      end
    end
  end
  $d(v, s) = D[v][s] - h(v) + h(s)$;
end
7. Determine Reachable Cities:;
Reachable_Cities = {};
for *each city v* **do**
  if $d(s, v) + d(v, s) \leq k$ **then**
    Add v to {Reachable_Cities};
  end
end
return *Reachable_Cities*;

Algorithm 2: Purrloin's Reachable Cities Algorithm

Runtime Analysis:

$$\begin{aligned}\text{Bellman-Ford Execution} &: O(n \times m) = O(n^3) \quad (\text{since } m = O(n)) \\ \text{Dijkstra's Execution} &: O(n^2 \log n) \quad (\text{running Dijkstra's from } s \text{ and each } v) \\ \text{Total Runtime} &: O(n^2 \log n)\end{aligned}$$

Correctness Justification:

- **Graph Representation:** Properly models the cities and roads with associated costs.
- **Super Node Addition and Bellman-Ford:** Ensures that all vertices are considered for potential reweighting and detects any negative cycles that would invalidate the path computations.
- **Reweighting Edges:** Ensures all edge weights are non-negative, allowing Dijkstra's algorithm to function correctly.
- **Dijkstra's Algorithm:** Accurately computes the shortest paths from the Nimbasa City Pokemon Center to all other cities and from each city back to the center.
- **Distance Adjustment:** Converts the reweighted distances back to their original values, preserving the correctness of the shortest paths.
- **Reachable Cities Determination:** Correctly identifies all cities that can be explored and returned from within the daily loss constraint k.

(b) [15 points] Purrloin is now planning to steal from the Castelia City Pokemon Center. Describe an $O(n^3)$ algorithm to find the cost of the cheapest path that Purrloin can take from the Nimbasa City Pokemon Center to Castelia City Pokemon Center and back. Briefly analyze its runtime and prove correctness.

Solution:

Objective: Find the cost of the cheapest round-trip path from the Nimbasa City Pokemon Center (s) to the Castelia City Pokemon Center (t) and back.

Approach: Utilize the **Floyd-Warshall algorithm** to compute all-pairs shortest paths, then sum the shortest path from s to t and from t to s to obtain the total round-trip cost.

Algorithm Steps:

Step 1. Graph Representation: Represent the cities as vertices V and roads as directed edges E with associated costs $c(u, v)$.

Step 2. Identify Source and Destination: Let s denote Nimbasa City Pokemon Center and t denote Castelia City Pokemon Center.

Step 3. Initialize Distance Matrix: Create a distance matrix D where:

$$D[i][j] = \begin{cases} c & (i,j) \text{if there is a road from } i \text{ to } j, \\ \infty & \text{otherwise,} \end{cases}$$

and $D[i][i] = 0$ for all i.

(a) **Floyd-Warshall Algorithm:** For each intermediate vertex k from 1 to n:

Step 1. For each pair of vertices (i, j):

$$D[i][j] = \min(D[i][j], D[i][k] + D[k][j])$$

(b) **Retrieve Round-Trip Cost:**

$$\text{Total Cost} = D[s][t] + D[t][s]$$

Pseudocode:

Input: Number of cities n, number of roads m, list of roads as tuples $(u, v, c(u, v))$, source s, destination t
Output: Cheapest round-trip cost from s to t and back
1. Initialize Distance Matrix:;
Initialize $D[n][n]$ with $D[i][j] = \infty$ for all i, j;
for *each city* $i = 1$ *to* n **do**
  Set $D[i][i] = 0$;
end
for *each road* $(u, v, c(u, v))$ **do**
  Set $D[u][v] = \min(D[u][v], c(u, v))$;
end
2. Execute Floyd-Warshall:;
for *each intermediate city* $k = 1$ *to* n **do**
  for *each source city* $i = 1$ *to* n **do**
    for *each destination city* $j = 1$ *to* n **do**
      if $D[i][k] + D[k][j] < D[i][j]$ **then**
        Set $D[i][j] = D[i][k] + D[k][j]$;
      end
    end
  end
end
3. Calculate Round-Trip Cost:;
if $D[s][t] = \infty$ *or* $D[t][s] = \infty$ **then**
  return *"No feasible round-trip path exists."*;
end
else
  Set Total_Cost $= D[s][t] + D[t][s]$;
end
return *Total_Cost*;

Algorithm 3: Cheapest Round-Trip Path Algorithm

Runtime Analysis:

$$\begin{aligned} \text{Initialization} &: O(n^2) \\ \text{Floyd-Warshall Execution} &: O(n^3) \\ \text{Round-Trip Calculation} &: O(1) \\ \text{Total Runtime} &: O(n^3) \end{aligned}$$

Correctness Justification:

- **Floyd-Warshall Algorithm:** Correctly computes the shortest paths between all pairs of cities, handling negative edge weights as long as there are no negative cycles.
- **Round-Trip Cost Retrieval:** Summing the shortest path from s to t and from t to s yields the minimum total cost for the round-trip.
- **Feasibility Check:** Ensures that a path exists in both directions before summing the costs.

(c) Modify the algorithm for part (b) to take $O(n^2 \log n)$ time.

Solution:

Objective: Optimize the round-trip cost computation from $O(n^3)$ to $O(n^2 \log n)$.

Approach: Utilize **Johnson's Algorithm** to reweight the graph, allowing the use of Dijkstra's algorithm for efficient shortest path computations on graphs with negative edge weights but no negative cycles. This reduces the overall runtime by avoiding the $O(n^3)$ complexity of Floyd-Warshall.

Algorithm Steps:

Step 1. Graph Representation: As in part (b), represent the cities as vertices V and roads as directed edges E with associated costs $c(u, v)$.

Step 2. Add Super Node: Introduce a super node s' connected to all nodes in V with edges of cost 0.

Step 3. Run Bellman-Ford from Super Node: Compute potential $h(v)$ for all $v \in V$ to reweight the edges and eliminate negative weights.

Step 4. Reweight Edges: For each edge $(u, v) \in E$, set the new cost $c'(u, v) = c(u, v) + h(u) - h(v)$. This ensures all $c'(u, v) \geq 0$.

Step 5. Run Dijkstra's Algorithm:

Step 1. Run Dijkstra's algorithm from s to compute the shortest path $d'(s, t)$.

Step 2. Run Dijkstra's algorithm from t to compute the shortest path $d'(t, s)$.

Step 6. Adjust Distances: Convert the reweighted distances back to original weights:

$$d(s, t) = d'(s, t) - h(s) + h(t)$$

$$d(t, s) = d'(t, s) - h(t) + h(s)$$

Step 7. Compute Round-Trip Cost:

$$\text{Total Cost} = d(s, t) + d(t, s)$$

Pseudocode:

Input: Number of cities n, number of roads m, list of roads as tuples $(u, v, c(u, v))$, source s, destination t
Output: Cheapest round-trip cost from s to t and back
1. Construct the Graph:;
Initialize adjacency list *adj* for all $n + 1$ vertices (including super node s');
for *each road* $(u, v, c(u, v))$ *in roads* **do**
  Add edge from u to v with cost $c(u, v)$;
end
for *each city* v *in* V **do**
  Add edge from super node s' to v with cost 0;
end
2. Run Bellman-Ford from Super Node s'**:**;
Initialize $d[s'] = 0$ and $d[v] = \infty$ for all $v \neq s'$;
for *each vertex* $k = 1$ *to* n **do**
  for *each edge* $(u, v, c(u, v))$ *in* E **do**
    if $d[u] + c(u, v) < d[v]$ **then**
      Set $d[v] = d[u] + c(u, v)$;
    end
  end
end
for *each edge* $(u, v, c(u, v))$ *in* E **do**
  if $d[u] + c(u, v) < d[v]$ **then**
    return *"Negative-weight cycle detected. Algorithm aborted."*;
  end
end
3. Reweight Edges:;
for *each edge* $(u, v, c(u, v))$ *in* E **do**
  Set $c'(u, v) = c(u, v) + d[u] - d[v]$;
end

4. Run Dijkstra's Algorithm from s and t:;
for *each source src in* $\{s, t\}$ **do**
  Initialize distance array $D[src][v] = \infty$ for all v;
  Set $D[src][src] = 0$;
  Initialize priority queue Q and insert src with priority 0;
  while Q *is not empty* **do**
    Extract vertex u with minimum $D[src][u]$ from Q;
    for *each edge* $(u, v, c'(u, v))$ *in* $adj[u]$ **do**
      if $D[src][u] + c'(u, v) < D[src][v]$ **then**
        Set $D[src][v] = D[src][u] + c'(u, v)$;
        Insert v into Q with priority $D[src][v]$;
      end
    end
  end
end
5. Adjust Distances Back to Original Weights:;
Set $d(s, t) = D[s][t] - h(s) + h(t)$;
Set $d(t, s) = D[t][s] - h(t) + h(s)$;
6. Compute Round-Trip Cost:;
if $d(s, t) = \infty$ *or* $d(t, s) = \infty$ **then**
  return *"No feasible round-trip path exists."*;
end
else
  Set Total_Cost $= d(s, t) + d(t, s)$;
end
return *Total_Cost*;

Algorithm 4: Optimized Round-Trip Path Algorithm

Runtime Analysis:

$$\text{Bellman-Ford Execution} : O(n \times m) = O(n^3) \quad (\text{since } m = O(n))$$
$$\text{Dijkstra's Execution (twice)} : 2 \times O(m + n \log n) = O(n^2 \log n)$$
$$\text{Total Runtime} : O(n^3) + O(n^2 \log n) = O(n^3)$$

Correctness Justification:

- **Bellman-Ford Algorithm:** Correctly detects any negative-weight cycles and computes potentials $h(v)$ to reweight the graph.
- **Reweighting Edges:** Ensures all new edge weights $c'(u, v) \geq 0$, making Dijkstra's algorithm applicable.
- **Dijkstra's Algorithm:** Efficiently computes the shortest paths from s to t and from t to s using the reweighted graph.
- **Distance Adjustment:** Converts the reweighted distances back to the original cost metrics.
- **Round-Trip Cost Calculation:** Accurately sums the shortest paths in both directions to determine the total round-trip cost.