Algorithm Homework 5

22.1-3

We can modify the *Bellman-Ford* algorithm to keep track of **whether any distance estimates change** during each pass over the edges. If an entire pass occurs without any distance estimate changes, terminate the algorithm early.

This ensures the algorithm will terminate in at most $m + 1$ passes.

```
BellmanFord(G, w, s):
   for each vertex v in G:
      v.d = ∞
      v.π = NIL
   s.d = 0
   for i = 1 to |V| - 1:
      changed = false
      for each edge (u, v) in E:
         if u.d + w(u, v) < v.d:
            v.d = u.d + w(u, v)
            v.π = u
            changed = true
      if not changed:
         break
   for each edge (u, v) in E:
      if u.d + w(u, v) < v.d:
         return "Graph contains a negative-weight cycle"
   return (d, π)
```

22.1-5

The *Bellman-Ford* algorithm inherently runs in $O(VE)$ time because it relaxes all edges $V-1$ times. Since each edge relaxation takes constant time and there are E edges, each pass over all edges takes $O(E)$ time, resulting in $O(VE)$ for $V-1$ passes.

To modify the *Bellman-Ford* algorithm for adjacency lists, iterate over all vertices and relax all outgoing edges from each vertex. This ensures that each edge is considered once per pass, maintaining the $O(VE)$ time complexity.

```
BellmanFordAdjList(G, w, s):
    for each vertex v in G:
        v.d = ∞
        v.π = NIL
    s.d = 0
    for i = 1 to |V| - 1:
        for each vertex u in V:
            for each vertex v in Adj[u]:
                if u.d + w(u, v) < v.d:
                    v.d = u.d + w(u, v)
                    v.π = u
    for each vertex u in V:
        for each vertex v in Adj[u]:
            if u.d + w(u, v) < v.d:
                return "Graph contains a negative-weight cycle"
    return (d, π)
```

22.1-6

1. Initialize $\delta^*(v) = \infty$ for all $v \in V$.
2. For each vertex $u \in V$:
   - Run *Bellman-Ford* with u as the source.
   - For each vertex $v \in V$, update $\delta^*(v)$ to be the minimum of its current value and the distance from u to v.

Since *Bellman-Ford* runs in $O(VE)$ and is executed V times, the total time complexity is $O(V^2E)$.

```
FindMinDistances(G, w):
   for each vertex v in G:
      δ*(v) = ∞
   for each vertex u in G:
      (d, π) = BellmanFord(G, w, u)
      for each vertex v in G:
         if d[v] < δ*(v):
            δ*(v) = d[v]
   return δ*

BellmanFord(G, w, s):
   for each vertex v in G:
      v.d = ∞
      v.π = NIL
   s.d = 0
   for i = 1 to |V| - 1:
      for each edge (u, v) in E:
         if u.d + w(u, v) < v.d:
            v.d = u.d + w(u, v)
            v.π = u
   for each edge (u, v) in E:
      if u.d + w(u, v) < v.d:
         return "Graph contains a negative-weight cycle"
   return (d, π)
```

22.2-2

The change ensures that each vertex is processed only once in topological order, which maintains the correctness of the algorithm.

The topological sort guarantees that by the time a vertex u is processed, all vertices v with edges (v, u) have already been processed.

This preserves the invariant that each vertex u has the correct shortest path distance when it is processed.

22.3-3

The proposed algorithm is not correct.

Dijkstra's algorithm must process all vertices to ensure that all shortest path estimates are finalized. Terminating the loop early means the shortest path to the last vertex in the priority queue Q might not be correctly computed.

22.3-5

1. Initialize an empty set of edges S.
2. For each vertex $v \in V$:
   - If $v \neq s$, add the edge $(\pi(v), v)$ to S.
3. Check that S forms a spanning tree:
   - Ensure $|S| = |V| - 1$.
   - Use a union-find data structure to check that S forms a single connected component without cycles.
4. Verify the shortest path properties:
   - For each edge $(u, v) \in E$, check that $v.d \leq u.d + w(u, v)$. If any check fails, return "Incorrect".
   - For each edge $(u, v) \in S$, ensure $v.d = u.d + w(u, v)$

If all checks pass, the attributes are correct. This verification runs in $O(V+E)$ time.

```
CheckDijkstraOutput(G, w, s, d, π):
   // Step 1: Initialize an empty set of edges S
   S = ∅

   // Step 2: Build the set of edges S from π values
   for each vertex v in G:
      if v ≠ s:
         if π[v] ≠ NIL:
            S = S ∪ {(π[v], v)}

   // Step 3: Check that S forms a spanning tree
   if |S| ≠ |V| - 1:
      return "Incorrect"
   if not isSingleConnectedComponent(S, G):
      return "Incorrect"

    // Step 4: Verify shortest path properties
   for each edge (u, v) in E:
      if d[v] > d[u] + w(u, v):
         return "Incorrect"
   for each edge (u, v) in S:
      if d[v] ≠ d[u] + w(u, v):
         return "Incorrect"

   return "Correct"
```

```
isSingleConnectedComponent(S, G):
   // Use union-find to check if S forms a single connected component
   uf = UnionFind(|V|)
   for each edge (u, v) in S:
      uf.union(u, v)
   root = uf.find(0)
   for each vertex v in G:
      if uf.find(v) ≠ root:
         return false
   return true
```

22.4-3

No, the shortest-path weight from the new vertex v_0 in a constraint graph cannot be positive.

The purpose of adding v_0 with edges to all other vertices with weight zero is to initialize the *Bellman-Ford* algorithm, ensuring that the shortest-path weights are calculated correctly.

Since the edge weights from v_0 are zero, the shortest-path weights from v_0 to any vertex will not be positive.

22.4-8

Let $Ax \leq b$ be a system of m difference constraints in n unknowns. Construct the corresponding constraint graph, where each constraint $x_i - x_j \leq b_k$ is represented as an edge (j, i) with weight b_k.

Running the *Bellman-Ford* algorithm on this graph will maximize $\sum_{i=1}^{n} x_i$ subject to $Ax \leq b$ and $x_i \leq 0$ for all x_i. This is because *Bellman-Ford* finds the shortest paths, effectively distributing the largest possible values to x_i while respecting the constraints.

22.4-9

The *Bellman-Ford* algorithm, when run on the constraint graph for a system $Ax \leq b$ of difference constraints, minimizes the quantity $(\max\{x_i\} - \min\{x_i\})$ subject to $Ax \leq b$.

This fact is useful in scheduling construction jobs because it ensures the smallest possible range of start times, leading to a more compact schedule and potentially reducing the overall project duration.

22.5-2

Consider the graph G with vertices $\{s, u, v\}$ and edges $\{(s, u), (s, v), (u, v)\}$ with weights $w(s, u) = 1$, $w(s, v) = 2$, and $w(u, v) = 1$.

The shortest-path tree rooted at s containing the edge (u, v) is $\{(s, u), (u, v)\}$.

Another shortest-path tree rooted at s that does not contain the edge (u, v) is $\{(s, u), (s, v)\}$.

22.5-5

Consider the graph G with vertices $\{s, u, v\}$ and edges $\{(s, u), (u, v), (v, s)\}$ with weights $w(s, u) = 1$, $w(u, v) = 1$, and $w(v, s) = 1$.

Assign $\pi(u) = s$, $\pi(v) = u$, and $\pi(s) = v$. This assignment of π values creates a cycle G_π because it implies a predecessor chain $s \to u \to v \to s$.

According to Lemma 22.16, such an assignment cannot be produced by a sequence of relaxation steps, confirming the presence of a cycle.

22-2 Nesting Boxes

a. Argue that the nesting relation is transitive.

To show that the nesting relation is transitive, assume we have three boxes A, B, and C such that A nests within B and B nests within C.

Let the dimensions of the boxes be:

- $A : (a_1, a_2, \ldots, a_d)$
- $B : (b_1, b_2, \ldots, b_d)$
- $C : (c_1, c_2, \ldots, c_d)$

Since A nests within B, there exists a permutation π such that:
$a_{\pi(1)} < b_1, a_{\pi(2)} < b_2, \ldots, a_{\pi(d)} < b_d$

Similarly, since B nests within C, there exists a permutation σ such that:
$b_{\sigma(1)} < c_1, b_{\sigma(2)} < c_2, \ldots, b_{\sigma(d)} < c_d$

To prove transitivity, we need to show that there exists a permutation τ such that:
$a_{\tau(1)} < c_1, a_{\tau(2)} < c_2, \ldots, a_{\tau(d)} < c_d$

Construct τ by following π and σ. For each i, set $\tau(i) = \pi(\sigma(i))$.

This ensures that:

$$a_{\tau(1)} = a_{\pi(\sigma(1))} < b_{\sigma(1)} < c_1$$
$$a_{\tau(2)} = a_{\pi(\sigma(2))} < b_{\sigma(2)} < c_2$$
$$\vdots$$
$$a_{\tau(d)} = a_{\pi(\sigma(d))} < b_{\sigma(d)} < c_d$$

Thus, A nests within C, proving that the nesting relation is transitive.

b. Describe an efficient method to determine whether one d-dimensional box nests inside another.

1. Sort the dimensions of both boxes A and B in non-decreasing order.
2. Compare the sorted dimensions element-wise. If for all i, the i-th dimension of A is less than the i-th dimension of B, then A nests within B.

```
canNest(A, B):
    sort(A)
    sort(B)
    for i from 1 to d:
        if A[i] >= B[i]:
            return False
    return True
```

This method runs in $O(d \ln d)$ time due to the sorting step.

c. Find the longest sequence of d-dimensional boxes that nest within each other.

1. Sort each box's dimensions.
2. Sort the list of boxes based on the first dimension, then the second dimension, and so on.
3. Use dynamic programming to find the longest increasing subsequence (LIS) where each box can nest inside the next one.

Running Time:

- Sorting the dimensions of each box: $O(nd \ln d)$
- Sorting the list of boxes lexicographically: $O(nd \ln n)$
- Finding the LIS: $O(n^2 d)$

```
longestNestingSequence(boxes):
   for each box in boxes:
      sort(box)
   sort(boxes, lexicographically)

   // LIS based on dimensions comparison
   n = length(boxes)
   dp = array of size n, all initialized to 1
   for i from 1 to n-1:
      for j from 0 to i-1:
         if canNest(boxes[j], boxes[i]):
            dp[i] = max(dp[i], dp[j] + 1)

   return max(dp)

canNest(A, B):
   for i from 1 to d:
      if A[i] >= B[i]:
         return False
   return True
```

22-3 Arbitrage

a. Algorithm to Determine Arbitrage Opportunity

we can use a graph-based approach with the *Bellman-Ford* algorithm. The key idea is to take the logarithm of the exchange rates to transform the problem into finding a negative-weight cycle in the graph.

1. **Transform the Exchange Rates:**
   Convert each exchange rate $R[i, j]$ to a weight by taking the negative logarithm: $w[i, j] = -\log(R[i, j])$
2. **Detect Negative-Weight Cycle:**
   Use the *Bellman-Ford* algorithm to find a negative-weight cycle in the graph. If such a cycle exists, it corresponds to an arbitrage opportunity.

It runs in $O(n^3)$ for a graph with n vertices and n^2 edges.

```
detectArbitrage(R):
   n = length(R)
   // Step 1: Transform exchange rates
   for i from 1 to n:
      for j from 1 to n:
         w[i][j] = -log(R[i][j])

   // Step 2: Use Bellman-Ford to detect negative cycle
   // Initialize distances
   dist = array of size n, all initialized to infinity
   dist[1] = 0 // Start from any vertex, here 1

   // Relax edges up to n-1 times
   for k from 1 to n-1:
      for i from 1 to n:
         for j from 1 to n:
            if dist[i] + w[i][j] < dist[j]:
               dist[j] = dist[i] + w[i][j]

   // Check for negative-weight cycle
   for i from 1 to n:
      for j from 1 to n:
         if dist[i] + w[i][j] < dist[j]:
            return True // Negative cycle detected

   return False // No negative cycle
```

b. Algorithm to Print Arbitrage Sequence ($O(n^3)$)

```
FindArbitrageSequence(R):
   n = length(R)
   // Step 1: Transform exchange rates
   for i from 1 to n:
      for j from 1 to n:
         w[i][j] = -log(R[i][j])

   // Step 2: Use Bellman-Ford to detect negative cycle
   // Initialize distances and predecessors
   dist = array of size n, all initialized to infinity
   pred = array of size n, all initialized to NIL
   dist[1] = 0 // Start from any vertex, here 1

   // Relax edges up to n-1 times
   for k from 1 to n-1:
      for i from 1 to n:
         for j from 1 to n:
            if dist[i] + w[i][j] < dist[j]:
               dist[j] = dist[i] + w[i][j]
               pred[j] = i

   // Check for negative-weight cycle and find the cycle
   for i from 1 to n:
      for j from 1 to n:
         if dist[i] + w[i][j] < dist[j]:
            // Negative cycle detected, reconstruct the cycle
            cycle = []
            visited = array of size n, all initialized to False
            x = i
            // Follow predecessors to find the cycle
            while not visited[x]:
               visited[x] = True
               x = pred[x]
            start = x
            cycle.append(start)
            x = pred[start]
            while x != start:
               cycle.append(x)
               x = pred[x]
            cycle.append(start)
            cycle.reverse()
            return cycle

   return None // No negative cycle
```

23.1-2

1. **Identity Property:**
    - $w_{ii} = 0$ ensures that the shortest path from any node i to itself is zero. This is consistent with the definition of distance, as the cost to travel from a node to itself should be zero in the absence of negative weight cycles.
2. **Initialization:**
    - When initializing the distance matrix D for *APSP algorithms*, it is convenient to set $D[i][i] = 0$. This provides a clear and consistent starting point, simplifying the algorithm's implementation and reducing potential errors in handling the diagonal elements.

3. **Simplified Relaxation:**
    - The relaxation step in *APSP algorithms* often involves updating the distance $D[u][v]$ using an intermediate node k. If $w_{ii} = 0$, the self-loops do not affect the relaxation process, as any path involving a self-loop would not change the shortest path computation:
      $D[u][v] = \min(D[u][v], D[u][i] + D[i][v])$
      When $i = v$, $D[i][v] = 0$, so:
      $D[u][u] = \min(D[u][u], D[u][i] + 0) = D[u][u]$

4. **Graph Representation:**
    - In the adjacency matrix representation of the graph, setting $w_{ii} = 0$ makes it clear that there are no self-loops with any cost, providing a consistent and straightforward way to represent the graph's structure.

In summary, setting $w_{ii} = 0$ for all i simplifies the implementation, ensures mathematical consistency, and aligns with the natural interpretation that the distance from a node to itself is zero.

23.1-3

The matrix $L^{(0)}$ represents the initial distance matrix for the shortest paths algorithm. In this matrix:

- The diagonal elements are 0, which indicates that the distance from any vertex i to itself is zero.
- The off-diagonal elements are ∞, which indicates that initially, the distances between different vertices are assumed to be infinite unless there is a direct edge connecting them.

In regular matrix multiplication, $L^{(0)}$ corresponds to the identity matrix I. The identity matrix I is a square matrix with ones on the diagonal and zeros elsewhere:

$$I = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

The identity matrix has the property that for any matrix A:
$AI = IA = A$

In the context of the shortest-paths algorithms, $L^{(0)}$ is used as the starting point for iterative updates, similar to how the identity matrix serves as the starting point for iterative operations in regular matrix multiplication. The updates to the distance matrix in the shortest-paths algorithms refine the initial assumptions (represented by $L^{(0)}$) to eventually compute the shortest paths between all pairs of vertices.

23.1-4

To show that the matrix multiplication defined by the `EXTEND-SHORTEST-PATHS` procedure is associative, we need to prove that:

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

The `EXTEND-SHORTEST-PATHS` procedure is defined as follows: Given two matrices $A = [a_{ij}]$ and $B = [b_{ij}]$, the product matrix $C = A \cdot B$ is defined as:

$$c_{ij} = \min_k (a_{ik} + b_{kj})$$

We need to prove that for any three matrices A, B, and C of compatible dimensions, the following holds:
$(\min_k(a_{ik} + b_{kj})) \cdot C = A \cdot (\min_m(b_{im} + c_{mj}))$

Let's define the intermediate matrices:

1. $D = A \cdot B$
2. $E = B \cdot C$

Therefore, the elements of D and E are:
$d_{ij} = \min_k(a_{ik} + b_{kj})$
$e_{ij} = \min_m(b_{im} + c_{mj})$

Now, we need to show:
$(D \cdot C)_{ij} = (A \cdot E)_{ij}$

Expanding both sides:
$(D \cdot C)_{ij} = \min_n (d_{in} + c_{nj})$
$(A \cdot E)_{ij} = \min_k (a_{ik} + e_{kj})$

Substituting d_{in} and e_{kj}:
$\min_n (d_{in} + c_{nj}) = \min_n \left(\min_k (a_{ik} + b_{kn}) + c_{nj} \right)$
$\min_k (a_{ik} + e_{kj}) = \min_k \left(a_{ik} + \min_m (b_{km} + c_{mj}) \right)$

Since $\min$ is associative and commutative, we can interchange the order of minimization:
$\min_n \left(\min_k (a_{ik} + b_{kn}) + c_{nj} \right) = \min_k \left(a_{ik} + \min_m (b_{km} + c_{mj}) \right)$

Both expressions are equal, proving that:
$(D \cdot C)_{ij} = (A \cdot E)_{ij}$

23.2-2

1. **Initialization**: Create an $n \times n$ matrix T where $T[i][j]$ is initialized to 1 if there is a direct edge from i to j (i.e., $(i, j) \in E$), and 0 otherwise. Additionally, set $T[i][i] = 1$ for all i.

2. **Algorithm**: Use the modified *Floyd-Warshall algorithm* to compute the transitive closure. For each intermediate vertex k, and for each pair of vertices i and j, update the matrix T as follows:
   $T[i][j] = T[i][j] \vee (T[i][k] \wedge T[k][j])$
   Here, $\vee$ represents the logical OR operation, and $\wedge$ represents the logical AND operation.

```
computeTransitiveClosure(G):
    n = number of vertices in G
    T = matrix of size n x n, initialized to 0

    // Step 1: Initialize the matrix T
    for i from 1 to n:
        for j from 1 to n:
            if i == j or (i, j) in E:
                T[i][j] = 1

    // Step 2: Update the matrix T using the modified Floyd-Warshall algorithm
    for k from 1 to n:
        for i from 1 to n:
            for j from 1 to n:
                T[i][j] = T[i][j] or (T[i][k] and T[k][j])

    return T
```

23.2-4

The original *Floyd-Warshall algorithm* computes shortest paths by updating the matrix D in n phases, where $D^{(k)}$ is the matrix of shortest paths considering only vertices 1 through k as intermediate vertices.

Instead of maintaining separate matrices for each k, we can update the matrix D in place. The key observation is that to compute $D^{(k)}$, we only need $D^{(k-1)}$. Thus, we can reuse the same matrix for all updates.

Here is the provided procedure `Floyd-Warshall'`:

```
Floyd-Warshall'(W, n)
   D = W
   for k = 1 to n
      for i = 1 to n
         for j = 1 to n
            d_ij = min{d_ij, d_ik + d_kj}
   return D
```

1. **Initialization**: We start with the matrix $D = W$, where W is the weight matrix of the graph.
2. **Updating the Matrix**: For each intermediate vertex k (from 1 to n):
   - For each pair of vertices i and j:
     - Update d_{ij} to be the minimum of its current value and the sum of the shortest paths from i to k and from k to j.

By the end of the algorithm, D will contain the shortest path distances between all pairs of vertices.

Correctness:

To show correctness, we need to demonstrate that the in-place updates correctly compute the shortest path distances.

- Initially, $D[i][j] = W[i][j]$, which represents the direct edge weights.
- At each iteration k, the algorithm considers vertex k as an intermediate vertex and updates the shortest paths accordingly.

By iterating through all k from 1 to n, the algorithm ensures that all possible intermediate vertices are considered, and thus computes the shortest paths correctly.

Space Complexity:

The original Floyd-Warshall algorithm requires $\Theta(n^3)$ space because it maintains $n+1$ matrices (one for each k). However, `Floyd-Warshall'` only uses one matrix D and performs in-place updates. Therefore, the space complexity is $\Theta(n^2)$.

In summary, `Floyd-Warshall'` correctly computes the shortest paths using in-place updates, and thus requires only $\Theta(n^2)$ space.

23.2-6

The *Floyd-Warshall algorithm* computes the shortest paths between all pairs of vertices in a weighted graph. The output of the algorithm is a matrix D where $D[i][j]$ represents the shortest path distance from vertex i to vertex j.

To detect the presence of a negative-weight cycle using the output of the *Floyd-Warshall algorithm*, you can simply examine the diagonal elements of the matrix D. Specifically, if there is any vertex i such that $D[i][i] < 0$, then the graph contains a negative-weight cycle.

Here's why this works:

- The diagonal element $D[i][i]$ represents the shortest path distance from vertex i to itself.
- Normally, $D[i][i]$ should be zero because the shortest path from any vertex to itself without considering any other edges is zero.
- If $D[i][i] < 0$, this implies that there is a path from i back to i with a total negative weight, which indicates a negative-weight cycle.

23.3-2

The purpose of adding the new vertex s to the vertex set V, resulting in $V' = V \cup \{s\}$, is to facilitate the application of the *Bellman-Ford algorithm* for detecting negative-weight cycles and for reweighting the edges in Johnson's algorithm.

Here's a detailed explanation of the steps involved and the purpose of adding the new vertex s:

1. **Facilitating *Bellman-Ford Algorithm***:
    - The *Bellman-Ford algorithm* is used to compute the shortest path distances from a single source vertex to all other vertices in a graph. By adding a new vertex s and connecting it with zero-weight edges to all other vertices, we ensure that the *Bellman-Ford algorithm* can be run from this new source s.
    - This guarantees that all vertices are reachable from the source s, allowing the algorithm to correctly determine if there are any negative-weight cycles in the original graph.

2. **Detecting Negative-Weight Cycles**:
    - If the *Bellman-Ford algorithm*, when run from s, detects a negative-weight cycle, it means that the original graph contains such a cycle. This detection is crucial as the presence of negative-weight cycles would invalidate the use of certain shortest path algorithms like Dijkstra's.

3. **Reweighting the Edges**:
    - The shortest path distances $h(v)$ from the new vertex s to each vertex v in G are used to reweight the edges of the original graph. The reweighting is done to ensure that all edge weights become non-negative, allowing *Dijkstra's algorithm* to be used on each vertex.
    - The reweighting is performed using the formula: $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$
    - This transformation preserves the relative order of the shortest paths while ensuring non-negative edge weights.

The procedure *Johnson's Algorithm* makes use of the added vertex s as follows:

1. **Compute G'**: Create a new graph G' with vertex set $V' = V \cup \{s\}$ and edge set $E' = E \cup \{(s, v) : v \in V\}$, where $w(s, v) = 0$ for all $v \in V$.

2. **Run Bellman-Ford**: Run the Bellman-Ford algorithm on G' with s as the source. If a negative-weight cycle is detected, the algorithm terminates.

3. **Compute Reweighting Values**: If no negative-weight cycle is detected, compute $h(v)$ for each vertex $v \in V'$ using the distances from s.

4. **Reweight the Edges**: Reweight each edge $(u, v) \in E$ using the computed h values.

5. **Run *Dijkstra's Algorithm***: For each vertex $u \in V$, run Dijkstra's algorithm on the reweighted graph to compute the shortest path distances.
6. **Adjust the Distances**: Adjust the computed shortest path distances to account for the reweighting.

By adding the new vertex s and connecting it to all other vertices with zero-weight edges, *Johnson's algorithm* can effectively use the Bellman-Ford algorithm to detect negative-weight cycles and reweight the graph, making it suitable for the application of *Dijkstra's algorithm*.

23.3-4

The professor's method of reweighting edges by subtracting the minimum edge weight w^* from each edge weight $w(u, v)$ is incorrect because it does not preserve the relative weights of the paths. Here’s why:

1. **Incorrect Path Weights**:
    - Subtracting w^* from each edge weight $w(u, v)$ does not necessarily ensure that the shortest paths are preserved. This is because the reweighting method used in *Johnson's algorithm* reweights the edges based on the potential function h, which is derived from the shortest path distances from a new source vertex s added to the graph.
    - Johnson's reweighting ensures that for every pair of vertices u and v, the reweighted edge $\hat{w}(u, v)$ maintains the shortest paths correctly while eliminating negative weights. The professor's method does not guarantee this property.

2. **Maintaining Non-negative Weights**:
    - The goal of reweighting in Johnson’s algorithm is to ensure that all edge weights become non-negative so that *Dijkstra*'*s algorithm* can be applied. Simply subtracting the minimum edge weight w^* from each edge weight does not guarantee that all resulting edge weights will be non-negative.
    - For example, if the graph contains edges with weights greater than w^* but still negative, subtracting w^* will not make all edge weights non-negative.

3. **Potential Negative Cycles**:
    - If the graph initially contains negative-weight cycles, the professor’s reweighting method will not necessarily remove these cycles or make the graph suitable for algorithms like Dijkstra’s. *Johnson’s algorithm* reweights edges in a way that transforms the graph such that no negative-weight cycles exist in the reweighted graph.

In summary, the professor's method of reweighting edges by subtracting the minimum edge weight w^* fails to preserve the shortest paths and does not guarantee non-negative edge weights. *Johnson’s algorithm* uses a more sophisticated reweighting method that ensures correctness and suitability for applying *Dijkstra's algorithm*.

23.3-5

To show that if G contains a 0-weight cycle c, then $\hat{w}(u, v) = 0$ for every edge (u, v) in c:

1. **Definition of 0-weight Cycle**:
   - A 0-weight cycle c in G is a cycle such that the sum of the weights of the edges in the cycle is zero: $\sum_{(u,v)\in c} w(u, v) = 0$
2. **Reweighting in *Johnson's Algorithm***:
   - In *Johnson's algorithm*, the reweighting of the edges is done using the formula: $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$
   - Here, $h(u)$ and $h(v)$ are the potentials computed by running the Bellman-Ford algorithm from the new source vertex s.

3. **Reweighted Cycle Weight**:
    - The weight of the cycle c after reweighting is: $\sum_{(u,v)\in c} \hat{w}(u,v) = \sum_{(u,v)\in c}(w(u,v) + h(u) - h(v))$
    - Since c is a cycle, the potentials $h(u)$ and $h(v)$ will sum to zero over the entire cycle: $\sum_{(u,v)\in c} h(u) - \sum_{(u,v)\in c} h(v) = 0$
    - Therefore, the reweighted weight of the cycle c is:
      $$\begin{aligned} &\sum_{(u,v)\in c} \hat{w}(u,v) \\ &= \sum_{(u,v)\in c} w(u,v) + \sum_{(u,v)\in c}(h(u) - h(v)) \\ &= 0 + 0 = 0 \end{aligned}$$

4. **Conclusion**:
    - Since the reweighted weight of the 0-weight cycle c is zero, and the reweighting preserves the individual edge weights in the cycle, it follows that $\hat{w}(u, v) = 0$ for every edge (u, v) in the cycle c.

Hence, if G contains a 0-weight cycle c, then the reweighted edge weight $\hat{w}(u, v) = 0$ for every edge (u, v) in c.

23-1 Transitive closure of a dynamic graph

a. To update the transitive closure G^* efficiently when a new edge (u, v) is added to G, we can use the following approach:

1. **Initialize the Transitive Closure**:
    - Represent the transitive closure using a boolean matrix T where $T[i][j]$ is true if there is a path from vertex i to vertex j, and false otherwise.
    - Initially, $T[i][i]$ is true for all i, and $T[i][j]$ is false for all $i \neq j$.

2. **Update the Transitive Closure**:
    - When a new edge (u, v) is added, update the matrix T as follows:
      $T[i][j] = T[i][j] \vee (T[i][u] \wedge T[v][j])]$
    - This ensures that for every pair of vertices i and j, there is a path from i to j if there is a path from i to u and a path from v to j.

```
updateTransitiveClosure(T, u, v, n):
    for i from 1 to n:
        for j from 1 to n:
            if T[i][u] and T[v][j]:
                T[i][j] = true
```

The time complexity of this update is $O(V^2)$, as it requires updating the boolean matrix for all pairs of vertices.

b. Consider a graph G with vertices $V = \{v_1, v_2, \ldots, v_n\}$ and edges forming a linear chain: $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$

Initially, the transitive closure includes paths from v_i to v_j for all $i \leq j$.

When we insert the edge (v_n, v_1), it creates a cycle that connects all vertices.

To update the transitive closure, we need to update the reachability for all pairs of vertices, which requires $\Omega(V^2)$ time.

c.

1. **Initialization**:
    - Represent the transitive closure using a boolean matrix T where $T[i][j]$ is true if there is a path from vertex i to vertex j, and false otherwise.
    - Initialize $T[i][i]$ to true for all i, and $T[i][j]$ to false for all $i \neq j$.
2. **Updating the Transitive Closure**:
    - For each edge (u, v) inserted, update the matrix T as follows:
      $T[i][j] = T[i][j] \vee (T[i][u] \wedge T[v][j])$

```
updateTransitiveClosure(T, u, v, n):
    for i from 1 to n:
        for j from 1 to n:
            if T[i][u] and T[v][j]:
                T[i][j] = true
```

Proof of Time Bound:

- Each insertion takes $O(V^2)$ time to update the boolean matrix T.
- For r insertions, the total time is $r \times O(V^2)$.
- To achieve the $O(V^3)$ bound, we need $r \times O(V^2) \leq O(V^3)$, which means $r \leq O(V)$.

Hence, the algorithm runs in $O(V^3)$ time for any sequence of r insertions, where $\sum_{i=1}^{r} t_i = O(V^3)$.

23-2 Shortest Paths in ε-Dense Graphs

a.

- For a d-ary min-heap:
  - **Insert**: $O(\log_d n)$
  - **Extract-Min**: $O(d \log_d n)$
  - **Decrease-Key**: $O(\log_d n)$
- If $d = \Theta(n^\alpha)$:
  - **Insert**: $O\left(\frac{\log n}{\alpha \log n}\right) = O(1/\alpha)$
  - **Extract-Min**: $O\left(n^\alpha \frac{\log n}{\alpha \log n}\right) = O(n^\alpha/\alpha)$
  - **Decrease-Key**: $O(1/\alpha)$

- For Fibonacci heaps:
    - **Insert**: $O(1)$ amortized
    - **Extract-Min**: $O(\log n)$ amortized
    - **Decrease-Key**: $O(1)$ amortized

The d-ary min-heap operations are competitive with Fibonacci heaps for appropriate choices of d.

b.

Pick $d = \Theta(V^{\epsilon})$.

- The running times for the d-ary heap operations:
    - **Insert**: $O(1)$
    - **Extract-Min**: $O(V^{\epsilon})$
    - **Decrease-Key**: $O(1)$
- Dijkstra's algorithm using a d-ary min-heap:
    - Initialize: $O(V)$
    - Insert all vertices: $O(V)$
    - Extract-Min for all vertices: $O(V \cdot V^{\epsilon}) = O(E)$ since $|E| = \Theta(V^{1+\epsilon})$
    - Decrease-Key for all edges: $O(E)$

Total time: $O(E)$.

c.

Run *Dijkstra's algorithm* from each vertex using the result from part **(b)**.

- For each source vertex:
  - Running time: $O(E)$
- Total for all vertices:
  - $O(V) \cdot O(E) = O(VE)$

d.

Use *Johnson's algorithm*:

1. Add a new vertex s connected to all vertices with edge weight 0.
2. Run *Bellman-Ford* from s to detect negative-weight cycles and compute potential function $h(v)$.
3. Reweight edges using $w'(u, v) = w(u, v) + h(u) - h(v)$
4. Run *Dijkstra's algorithm* from each vertex using the reweighted edges.

- Time Complexity:
  - *Bellman-Ford*: $O(VE)$
  - *Dijkstra's* from each vertex: $O(V) \cdot O(E) = O(VE)$

Total time: $O(VE)$.