CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2024**

Homework #5

Name(s): Raashi Maheshwari, Tanvi Yamarthy

Homework Policy

- If you leave a question completely blank, you will receive 20% of the grade for that question. This however does not apply to the extra credit questions.
- You may also consult all the materials used in this course (lecture notes, textbook, slides, etc.) while writing your solution, but no other resources are allowed.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "sort the array in $\Theta(n \log n)$ time using merge sort".
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question). See "Practice Homework" for an example.
- The extra credit problems are generally more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.
- **Groups:** You are allowed to form groups of size two or three students for solving each homework (you can also opt to do it alone if you prefer). The policy regarding groups is as follows:
  - You can pick different partners for different assignments (e.g., from HW1 to HW2) but for any single assignment (e.g., HW1), you have to use the same partners for all questions.
  - The members of each group only need to write down and submit a single assignment between them, and all of them will receive the same grade.
  - For submissions, only one member of the group submits the full solutions on Canvas and lists the name of their partners in the group. The other members of the group also need to submit a PDF on Canvas that contains only a single line, stating the name of their partner who has submitted the full solution on Canvas (Example: Say A, B, and C are in one group; A submits the whole assignment and writes down the names A, B, and C. B and C only submit a one-page PDF with a single line that says "See the solution of A").
  - You are allowed to discuss the questions with any of your classmates even if they are not in your group. **But each group must write their solutions independently.**

Problem 1. A graph $G = (V, E)$ is dense if $|E| = \Theta(V^2)$. Using binary heaps, Prim's algorithm for finding a minimum spanning tree runs in $O(E \log V)$. With Fibonacci heaps, the runtime improves to $O(E + V \log V)$, which achieves linear time $O(E)$ for dense graphs.

Describe an alternative algorithm that:

- Uses a simple data structure (avoiding Fibonacci heaps).
- Runs in linear time $O(E)$for dense graphs.

(30 points)

Solution. The pseudocode for the problem is as follows:

```
function mstDenseGraphs(Graph G, num_vertices) {
    infinity = float('inf')

    min_weight = [infinity] * n
    in_mst = [false] * n
    parent = [-1] * n

    min_weight[0] = 0

    for i in range (0, n - 1) {
        u = -1
        for v in range (0, n - 1) {
            if not in_mst[v] and (u == -1 or min_weight[v] < min_weight[u]) {
                u = v
            }
        }

        in_mst[u] = True

        for v in range(n) {
            if graph[u][v] > 0 and not in_mst[v] and graph[u][v] < min_weight[v] {
                min_weight[v] = graph[u][v]
                parent[v] = u
            }
        }
    }

    return parent
}
```

Proof of Correctness:

1. Initialization: min_weight is set to infinity for every vertex except the starting vertex, which is initialized to 0. This makes it so that the starting vertex is chosen first. in_mst[v] is used to keep track of whether a vertex v is already in the minimum spanning tree. The tree starts off with no vertices in there.
2. The algorithm selects the vertex u with the smallest edge weight among all the vertices that are not already in the minimum spanning tree at each step. This ensures the algorithm follows the greedy choice property.

3. The algorithm works to maintain an optimal substructure. This is done by examining all edges from u to its adjacent vertices v whenever vertex u is added to the minimum spanning tree. If the edge (u,v) examined provides a smaller connection to v than the one previously stored, the value of min_weight[v] s updated.

4. The algorithm will terminate once all the vertices are added to the minimum spanning tree (after n iterations). At this point, parent contains all the edges of the minimum spanning tree and the sum of min_weight is the total weight of the minimum spanning tree

5. We know the minimum spanning tree will be correct because the algorithm only adds edges that connect the minimum spanning tree to a new vertex using the minimum possible weight. Since all the vertices are added exactly once, and no cycles are created, the outcome graph will be a valid minimum spanning tree.

Runtime Analysis:

1. The adjacency matrix allows constant $O(1)$ time to access the edge weights

2. The algorithm iterates V times to add all the vertices to the minimum spanning tree

3. During each iteration, the algorithm scans the vertex indexed element from the min_weight array to find the vertex with the smallest weight. Per iteration, this takes $O(V)$ time for a total time (for all iterations) of $O(V^2)$

4. The algorithm updates the array min_weight for its adjacent vertices for each newly added vertex u. Checking all V vertices in the adjacency matrix takes $O(V)$ time per iteration, for a total runtime of $O(V^2)$.

5. Combining all those elements, we get that the algorithm performs $O(V^2)$. For dense graphs where $|E| = \Theta(V^2)$, the runtime would simplify to $O(E)$

Problem 2. Demonstrate failure of the following algorithm on a directed graph G with edge weights in $\{-1, 0, 1\}$. The algorithm works by transforming G into a new graph G', where all edge weights are increased by 1, i.e., for every edge (x, y) in G, the new weight in G' is $w'(x, y) = w(x, y) + 1$. The algorithm then runs Dijkstra's algorithm on G' starting at source s to compute the shortest s-t path. Finally, the algorithm outputs the weight of this path in the original graph G.

To show that this algorithm fails, provide the following details:

- The graph G, including its vertices, edges, and the original edge weights $w(x, y)$. Ensure G contains at most 7 vertices.
- The source vertex s and the target vertex t.
- The transformation to G', including all edge weights $w'(x, y) = w(x, y) + 1$.
- The priority queue values during the execution of Dijkstra's algorithm on G', showing the shortest path computation step by step.
- The actual shortest distance from s to t in the original graph G.
- The incorrect shortest distance from s to t returned by the algorithm, demonstrating where and why it fails.

(30 points)

Solution. To show that the algorithm fails, we can work towards showing its failure using a counter-example.

1. Construct a Graph: Let us construct a graph G with four vertices: s, A, B, t where s is the source vertex and t is the target vertex.
2. Edges with Weights: The original edges in the graph G are: (s, A) with a weight of 1, (s, B) with a weight of -1, (A, t) with a weight of -1, and (B, t) with a weight of 1.
3. Transforming to G': If we add 1 to each edge weight, G' will end up with the following edges and their weights: (s, A) with a weight of 2, (s, B) with a weight of 0, (A, t) with a weight of 0, and (B, t) with a weight of 2.

Original Graph G: Let $G = (V, E)$ where:

- $V = \{s, A, B, t\}$
- E with weights $w : V \times V \to \{-1, 0, 1\}$:

$$\begin{aligned} w(s, A) &= 1 \\ w(s, B) &= -1 \\ w(A, t) &= -1 \\ w(B, t) &= 1 \end{aligned}$$

Transformed Graph G' Define $w'(u, v) = w(u, v) + 1$ for all $(u, v) \in E$:

$$\begin{aligned} w'(s, A) &= 2 \\ w'(s, B) &= 0 \\ w'(A, t) &= 0 \\ w'(B, t) &= 2 \end{aligned}$$

1 Dijkstra's Algorithm Execution on G'

Priority Queue (PQ) Iterations

Graph G', Source Vertex s

Shortest Path from s to t

Initialize Initial States:

- Distances s: 0, A: ∞, B: ∞, t: ∞
- Priority queue: [(0,s)]

Iteration 1: Explore from s

- Update A: $\delta(A) = 2$ (via $s \to A$)
- Update B: $\delta(B) = 0$ (via $s \to B$)
- PQ: $\{(B, 0), (A, 2)\}$

Iteration 2: Explore B

- Update t: $\delta(t) = 2$ (via $B \to t$)
- PQ: $\{(A, 2), (t, 2)\}$

Iteration 3: Explore A

- Update t: $\delta(t) = 0$ (via $A \to t$)
- PQ: $\{(t, 2), (t, 2)\}$

Final Path: $s \to B \to t$, total cost $= 2$

2 Path Analysis

Paths in Original Graph G

- $s \to A \to t$: Total Weight $= 1 + (-1) = 0$
- $s \to B \to t$: Total Weight $= -1 + 1 = 0$

3 Edge Selection Failure in Transformed Graph

3.1 Graph Representation

Consider the directed graph G' with vertices $\{s, A, B, t\}$ and transformed edge weights:

- $w'(s, A) = 2$
- $w'(s, B) = 0$
- $w'(A, t) = 0$
- $w'(B, t) = 2$

3.2 Dijkstra's Algorithm Edge Selection

Transformed Graph G', Source Vertex s Shortest Path Computation

Initialization:

- $\delta(s) = 0$
- $\delta(A) = \infty$
- $\delta(B) = \infty$
- $\delta(t) = \infty$

First Iteration:

- Edge Candidates from s:

$$(s, A) : \text{Weight } 2$$
$$(s, B) : \text{Weight } 0$$

- Algorithm Selects: (s, B) with weight 0

Critical Edge Selection:

- From vertex B, selects edge (B, t)
- Edge (B, t) has weight 2

4 Failure Analysis

4.1 Incorrect Path Selection

The algorithm fails by selecting the edge (B, t) with weight 2, overlooking the zero-weight path through A.

Proof. Path selection breakdown:

$$\text{Selected Path: } s \to B \to t$$
$$\text{Path Weight: } 0 + 2 = 2$$

$$\text{Optimal Path: } s \to A \to t$$
$$\text{Actual Weight: } 0 + 0 = 0$$

□

4.2 Failure Mechanism

The algorithm fails due to:

- Preferential selection of edge (B, t) with weight 2
- Overlooking zero-weight alternative path through A
- Transformation distorting original path optimality

Actual Shortest Distance: 0

Algorithm's Returned Distance: 2

Failure Explanation: The algorithm fails because the relative path costs become distorted by adding 1 to the edge weights. In this example, the algorithm incorrectly reports a path with a weight of 2 when there are two paths with zero total weight.

The issue is that the transformation doesn't preserve the relative optimality of the paths. Paths that were previously equivalent in terms of total weight become distinguished by uniformly adding 1 to all edges, leading to an incorrect shortest path selection.

4.3 Time Complexity

Let G be a graph with:

- $|V|$ vertices
- $|E|$ edges
- $w_{\max}$ maximum edge weight

[Time Complexity] The overall algorithm time complexity is $O((|V| + |E|) \log |V|)$, dominated by Dijkstra's algorithm.

Proof. Breakdown of computational steps:

1. Graph transformation: $O(|E|)$
2. Dijkstra's algorithm: $O((|V| + |E|) \log |V|)$ using binary heap
3. Path reconstruction: $O(|V|)$

Dominant term: Dijkstra's algorithm $O((|V| + |E|) \log |V|)$ □

Problem 3. You are given a weighted undirected graph $G = (V, E)$ with integer weights $w_e \in \{1, 2, \ldots, W\}$ on each edge e, where $W \leq 25$. Given two vertices $s, t \in V$, the goal is to find the minimum weight path (or shortest path) from s to t. Recall that Dijkstra's algorithm solves this problem in $O(n + m \log m)$ time even if we do not have the condition that $W \leq 25$. However, we now want to use this extra condition to design an even faster algorithm.

Design and analyze an algorithm to find the minimum weight (shortest) $s - t$ path on these restricted weighted graphs in $O(V + E)$ time.

(40 points)

Solution. The pseudocode for the algorithm is as follows:

```
def shortest_path(graph G, s, t) {
    num_vertices = len(G)
    weight_limit = 25

    buckets = new array[n * W + 1]
    for i = 0 to (n * W) {
        buckets[i] = [] //empty list
    }

    distance = [float('inf')] * n
    distance[s] = 0
    buckets[0].append(s)

    curr_distance = 0

    while curr_distance < len(buckets) {
        while buckets[curr_distance] {
            u = buckets[curr_distance].pop()

            if u == t {
                return curr_distance
            }

            for v, weight in graph[u] {
                new_distance = curr_distance + weight

                if new_distance < distance[v] {
                    if distance[v] != float('infinity') {
                        buckets[distance[v]].remove(v)
                    }

                    distance[v] = new_distance
                    buckets[new_distance].append(v)
                }
            }
        }

        curr_distance ++
    }

    return float('infinity')
}
```

Proof of Correctness:

1. Initialization: distance[v] is the shortest path from s to v. Initially distance[s] = 0, distance[v] = infinity for all values where v is not equal to s. s is placed in bucket 0.
2. For each edge(u,v) with weight w, if distance[u] + w ¡ distance[v], then distance[v] is updated. The vertex is then moved to a new bucket corresponding to the updated distance.
3. Distance Processing: The vertices are processed in a non-decreasing order of total path distance. Each vertex is processed at most once per unique distance. At each iteration, we process the vertices at the current minimum distance. This guarantees that each vertex will first be visited through its shortest path.
4. Maintaining the shortest path: For each vertex v, the distance[v] always contains the shortest path distance from the source. If a shorter path is found, the vertex moves to that shorter path's corresponding bucket. This process allows for continuous updates that maintain the shortest path property.
5. First Path is the Shortest: When the target is first processed, distance[t] is guaranteed to be the minimum. No other subsequent processing can work to reduce this distance. This allows us to prove that the first path to the target is the shortest.

Runtime Analysis

1. Initialization: $O(V + E)$
2. Bucket Processing: Each vertex is visited at most once and each edge is processed at most once leading to a bucket processing runtime of $O(W * (V + E))$ which simplifies to $O(V + E)$
3. Total time complexity: $O(V + E)$

Problem 4. We consider two types of graphs:

- **Vertex-Weighted Graph** $H = (V, E)$: Each vertex v has a weight $c(v)$. The weight of a path is the sum of vertex weights along the path. Let $\text{dist}_H(s, v)$ represent the shortest path from s to v.
- **Edge-Weighted Graph** $G = (V, E)$: Each edge (u, v) has a weight $w(u, v)$. Let $\text{dist}_G(s, v)$ represent the shortest path from s to v.

a) Suppose you have an algorithm EdgeWeightedSP(G, s) that solves the edge-weighted shortest path problem in $O(|E(G)|)$. Write pseudocode for VertexWeightedSP(H, s), which solves the vertex-weighted shortest path problem in $O(|E(H)|)$. You can use EdgeWeightedSP(G, s) as a blackbox.

b) Suppose you have an algorithm VertexWeightedSP(H, s) that solves the vertex-weighted shortest path problem in $O(|E(H)|)$. Write pseudocode for EdgeWeightedSP(G, s), which solves the edge-weighted shortest path problem in $O(|E(G)|)$. You can use VertexWeightedSP(H, s) as a blackbox.

(40 points)

Solution. Part A Pseudocode:

```
function vertextWeightedSP(H = (V, E), s):
    G' = new Graph(V, E')

    for v  V {
        G'.addVertex(v)
    }

    for (u, v)  E(H){
        G'.addEdge(u, v, c(v))
    }

    return EdgeWeightedSP(G',s)
```

Proof of Correctness: The algorithm computes the shortest path distances in a vertex-weighted graph by leveraging an edge-weighted shortest path algorithm. The process involves transforming the path weights in a way that preserves the correctness of the final shortest path computation, followed by an adjustment step to ensure accurate distance calculation.

Graph Transformation Correctness

1. The weight of an original path is defined as the sum of the vertex weights along the path.
2. After transformation, the path weight is represented as the sum of edge weights derived from the vertex-weight pairs.
3. The total path weight remains invariant during the transformation process, preserving correctness.

Distance Calculation in the Transformed Graph

1. Consider a path $P = (v_0, v_1, \ldots, v_k)$ in the transformed graph H.
2. The original path weight is given by:
$$\sum_{i=0}^{k} c(v_i)$$
where $c(v_i)$ represents the weight of vertex v_i.

3. In the transformed graph G', the path weight is computed as:

$$\sum_{i=0}^{k-1} [c(v_i) + c(v_{i+1})]$$

Vertex Weight Adjustment

1. To normalize the distances, the weight of the source vertex is subtracted after the shortest path computation.

Time Complexity: The total time complexity is determined by the following steps:

- **Graph Transformation:**
  - Adding vertex self-loops requires $O(|V|)$.
  - Adding edges from the original graph requires $O(|E|)$.

  Together, the graph transformation takes $O(|V| + |E|)$.
- **Shortest Path Algorithm:** The Edge-Weighted Shortest Path algorithm operates on the transformed graph G', requiring $O(|E(G')|)$. Since $|E(G')| = |E(H)|$, this step takes $O(|E(H)|)$.

Total Time Complexity: Combining these steps, the total time complexity is:

$$O(|V| + |E| + |E(H)|)$$

For most practical graphs, where $|E(H)| \geq |V|$, the complexity simplifies to:

$$O(|E(H)|)$$

Part B: The pseudocode for the algorithm is as follows:

```
function EdgeWeighted(G = (V, E), s):
    H' = new Graph()

    for each vertex in V(G){
        H'.addVertex(v, 0)
    }

    for (u, v)  E(G){
        H'.addEdge(u,v)
        H'.setVertexWeight(v, w(u, v))
    }

    return VertexWeightedSP(H', s)
```

Proof of Correctness: This algorithm calculates the shortest path distances in an edge-weighted graph by utilizing a vertex-weighted shortest path algorithm. As with the previous part, the correctness must be demonstrated for three key components of the algorithm:

Graph Transformation

1. The weight of the original path is defined as the sum of edge weights along the path.
2. After transformation, the path weight is expressed as the sum of vertex weights that correspond to the original edge weights.
3. The total path weight remains unchanged, as the transformation simply assigns the edge weights to the vertices, preserving the original weight values.

Distance Calculation

1. Consider a path $P = (v_0, v_1, \ldots, v_k)$ in the original graph G.
2. The weight of the original path is:
$$\sum_{i=0}^{k-1} w(v_i, v_{i+1})$$
where $w(v_i, v_{i+1})$ represents the weight of the edge between v_i and v_{i+1}.
3. In the transformed graph G', the path weight is computed as:
$$\sum_{i=1}^{k} c(v_i)$$
where $c(v_i)$ is the vertex weight derived from the edge weights in G.

Overall Consistency

1. The edge weights are directly translated into vertex weights, ensuring no changes to the actual weight values.
2. The shortest path properties are preserved throughout the transformation process.

Time Complexity Analysis

We will now break down the time complexity of the algorithm in Part B.

1. **Vertex Creation:**
   - We iterate over all vertices in G and add them to H'. Since there are $|V|$ vertices in G, this step takes $O(|V|)$ time.
2. **Vertex Weight Assignments:**
   - Each edge (u, v) in G is transformed into a vertex weight assignment for v in H'. This step involves iterating over all edges in G, which takes $O(|E|)$ time because there are $|E|$ edges.
3. **Vertex-Weighted Shortest Path Computation:**
   - We now run the vertex-weighted shortest path algorithm on the transformed graph H', which is equivalent to running the edge-weighted shortest path algorithm on G. Since the number of edges in H' is the same as in G, the shortest path computation takes $O(|E|)$ time.
4. **Total Time Complexity:** The overall time complexity is the sum of the individual steps:

   $O(|V|)$ (vertex creation)+$O(|E|)$ (vertex weight assignments)+$O(|E|)$ (vertex-weighted shortest path computation).

   Therefore, the total time complexity is:
   $$O(|V| + |E|).$$
   For sparse graphs where $|E|$ dominates $|V|$, this simplifies to:
   $$O(|E|).$$

Problem 5. Given a directed graph $G = (V, E)$ with positive integer edge weights, let s be the source vertex. Assume all shortest path distances satisfy $\text{dist}(s, v) \leq B$ for some parameter B. Design a data structure D such that running Dijkstra's algorithm using D results in a runtime of $O(|E| + B)$.

- Describe the data structure D and its three operations:
  - **Insert:** Add a vertex with its priority.
  - **Extract-Min:** Remove and return the vertex with the smallest priority.
  - **Decrease-Key:** Update the priority of a vertex.
- Write pseudocode for these operations to show how D achieves $O(1)$ or $O(B)$ time per operation as appropriate.
- Justify why the runtime of Dijkstra's algorithm becomes $O(|E| + B)$ using D.

(40 points)

Solution. This question asks us to design a data structure, D, such that if we were to use D to run Dijkstra's algorithm, then our runtime would end up being $O(E+B)$. The data structured that we designed to allow us to achieve the required functionality and perform the three operations (insert, extract-min, and decrease key) is a priority queue that is bucket based. Using a bucket based priority queue allows use to ensure that we can account for positive edge weights. Our bucket based priority queue, D, will be an array that has "buckets" (about B+1 number of buckets). Each bucket, will have a priority associated with it, and the priorities are ranked from 0 to level B. A bucket itself is a doubly linked list that stores vertices with associated distance values from the source vertex. So for example, if we are looking at a bucket at index i, it will contain the vertices such that the shortest path from the source is indeed i, itself. It is very similar to how we typically use adjacency lists to represent graphs. And, in order to access any specific bucket is this array of buckets, it will take a 0(1) runtime, because it is just constant look up. Now, let us describe the three operations that the question asks us. If we want to insert a vertex with priority, then we are trying to add a vertex v, into the bucket that has the right priority that is also in our input parameter. If a vertex has a priority of 2, then our vertex will be placed into bucket[2]. Now, let us understand the extract min operation which removes and returns the vertex with the smallest priority. To do this, we want to iterate through all our buckets in our array from index 0. Our lowest priority is represented by our smallest non-empty bucket, and so once we reach that point, we remove the vertex from it and return it. And our last operation is decrease key which is used to update the priority of a vertex If a vertex' priority decreases, then we move a vertex from that bucket to another. So we remove from bucket[old priority] and add the same vertex, v, to bucket [new priority]. We described each of these operations and the data structures itself, but now lets look at the pseudo code and corresponding runtime for them:

```
Insert: adding a vertex with priority

function Insert(vertex, priorityAmount):
   bIndex=priorityAmount;
   bucket[bIndex].add (vertex);
   //we are adding the vertex to a bucket of the new input priority


   RUNTIME: $O(1)$. Inserting into a bucket array, just requires us to
   index directly,
   using the priorityAmount index in the parameter.
   Traversal is not
   needed to add a vertex to the bucket.
```

```
Extract Min Operation PsuedoCode

function ExtractMin():
   for int i=0 ; i++; i<B+1 :
       if bucket[i] isn't empty:
           vertex=bucket[i].remove
           //removes a vertex from least priority bucket
           return vertex;
   else:
        return null

   RUNTIME: $O(B)$ because we need to traverse through all B+1 buckets
   from index 0 to index B,
   and thats why this
   operation runs in linear time.



Decrease Key Operation psuedocode

function decreaseKey(vertex, newPriority, oldPriority):
   bucket[oldPriority].remove(vertex);
   bucket[newPriority].add(vertex);

RUNTIME: $O(1)$ because we are direct indexing into the buckets without need for
traversal, similar to our insert into operation  that we previously just did.
```

The last part of the question asks us to justify why the **runtime of Dijkstra's algorithm** becomes $O(|E| + B)$, when we are using this data structure D. To understand this, we can examine the 3 phases of Dijkstra's algorithm and the corresponding run times of each plus when we put them all together. A key part of Dijkstra's algorithm is bucket scanning, we need to go through all the buckets because there are the vertices and the corresponding distances/priorities ar stored. Since we are iterating through our array of buckets, the runtime for this step will be $O(B)$, because there are B+1 iterations that need to occur. This runs in linear time, matching up with our extract min operation. Then, we also need to perform our edge relaxation the occurs in Dijkstra's algorithm. Edge relaxation typically relies or calls our Decrease-Key operation, and as we went over in the pseudo code, this costs us $O(1)$ runtime. But, we need to account for the fact that our Edge relaxation requires us to actually traverse through our edges as we "relax" each one, and that costs us a runtime of $O(|E|)$, which will dominate the constant time for this step. As we relax each of the edges, we are also adding the vertices into their corresponding buckets, but the $O(E)$ runtime dominates that. So, our total runtime for Dijkstra's will be $O(|E| + B)$, accounting for the scanning of all the buckets, and relaxing the edges in our graph.

The last thing that we can look at for this question is the **proof of correctness**. Let us look at the proof of correctness for each of the three operations that we went through.

1. **Insert Operation**: This is dependent on the priority of the vertex itself, and so we need to directly index the buckets. Because we are using direct indexing, we can guarantee that all the vertices will be indexed into their appropriate buckets, as this is basic array operations.
2. **Extract Min Operation**: This operation works by finding the smallest priority as we iterate through the bucket list and find the first bucket that is not empty. Again, we are using simple array traversal,

ensuring that we are correctly identifying which bucket is not empty with our if statement, and removing a vertex from that bucket. This achieves the goal of identifying the smallest priority vertex.

3. **Decrease Key Operation**: This operation works by removing a vertex given its old priority, and adding it to a bucket that has its new priority. The code is straight forward, as once again it relies on direct indexing given the bucket priority. We are using a lot of the same logic as the insert function, ensuring that the operation does what is intended to do.

Problem 6. Describe and analyze a modification of the Bellman–Ford algorithm that achieves the following:

- If a negative cycle is reachable from s:
  - Return the negative cycle.
  - Set $\text{dist}(v) = -\infty$ for any vertex v reachable through the cycle.
- If no negative cycle is reachable:
  - Return the shortest-path tree.
  - Compute the correct shortest-path distances $\text{dist}(v)$ from s to every vertex v.
- The algorithm should run in $O(VE)$ time.

(50 points)

Solution. The question asks us to describe a modification of the Bellman-Ford algorithm which typically detects cycles with negative weights in graphs, and then finds the minimal path distance in a graph. However, this question is asking us to modify the Bellman-Ford algorithm such that we can return negative cycles if they are found, else return the shortest path tree, all in the required $O(VE)$ runtime. In my head, the first way that I can go about this algorithm is by setting the distance(u) of a vertex u to infinity, except the source index distance(source), which will have a distance of 0 to itself. Then we want to implement our regular Bellman Ford algorithm, traversing through all the edges in the graph, and then for every edge e (u, v) and every edges' weight w (u, v), we want to see whether the distance of u plus the weight of the edge will be less than the distance of the vertex v that u is connected to. If it is, that is how we know that the predecessor of that vertex v, is indeed u. In other words, u will come before v in a potential cycle. Then, we want to immediately see if a negative cycle exists, and to do that we should take a look at it. Start at the vertex v, and see if distance of u plus the weight of the edge will be less than the distance of the vertex v. In other words if dist(u)+weight(u,v)¡dist(v). if yes, keep backtracking using your previously identified predecessor. For each one you backtrack to that validates this point, mark it as yes, and belong to your negative cycle. Now that we have detected our negative cycle, perform BFS on it, so that we can do what the question asks us and set dist(v) to -infinity.

Now, let us go through the psuedo code that can break down this algorithm.

```
function newBellmanFord:
   for v in graph G,

    s=source
    distance[source]=0
    distance[v]=infinity
    //exactly as mentioned in the description above

    predecessor[v]=null
    //we don't yet know what comes before v in our cycle

    for i=1 to V-1:
        for every edge (u,V)
            if distance[u]+ weight(u,v) < distance [v]:
                predecessor[v]=u
                distance[v]=distance[u]+w

    negativeCyle=[] //list to keep track of vertices in our negative cycle
    for e(u,v) and weight(u,v) in Graph g:
```

```
        if distance[u]+ weight(u,v) < distance [v]:
            //that means negative cycle exists, and this vertex is apart of it
            vertexToAdd=v
            for i=1 to V:
                vertexToAdd=predeseccor[vertexToAdd]
                startPoint=vertexToAdd
                /// go through each predecessor, and all the vertices in the cycle and
                add them to our
                negativeCycle list.


                until vertextoAdd=startPoint
                break;


    if negativeCycle != empty:
        distance[v]=-infinity //as determined by the question
        //call bfs on the negative cycle's vertices
        BFS(Graph G, negativeCycle, distance)

    else:
        return "There is no negative cycle that exists", distances list
```

Now, let us analyze **the runtime for this algorithm.** The runtime of this algorithm is indeed $O(VE)$. We can break our algorithm into three steps.

1. The first step is where we initialize all our distances list values. We set our distances[v] to infinity, our distances[source] to 0, and our predecessors to null. This gives us a runtime of $O(V)$, as we need to iterate through all our vertices to do this.
2. The next part of our algorithm is edge relaxation which has a runtime of $O(VE)$, as we have to relax e, edges for v-1 iterations/times.
3. The next part of our algorithm is when we actually have to detect our negative cycle, and this will get us a runtime of O(E), since we need to iterate over E, edges to do this, to perform edge relaxation. Our backtracking with predecessors takes us $O(V)$ times, but our E dominates, as it is greater than V in our graph.
4. The last part of our algorithm uses BFS, and that takes us $O(V+E)$, as the standards BFS algorithm runtime.
5. Taking into account the most dominating terms from each step, our total runtime for this algorithm will indeed be $O(VE)$.

Lastly, let us analyze the **proof of correctness** for this algorithm.

1. The first component of our proof is to show that we can correctly detect if a negative cycle is reachable from our source vertex, s. This is true because we are relying on a standard Bellman Ford algorithm for this question, which iterates through v-1 iteration. If an edge can be relaxed in the v-th iteration, it shows us that there is a possible negative weight cycle that could be coming into play. Thus, in order to keep track of this cycle, we identify the predecessor and backtrack to keep track of the vertices in this cycle.

2. Once we found the negative cycle, we use a standard BFS algorithm that goes to all the vertices that are reachable from the cycle and sets their corresponding distance to distance[v]=infinity. We can guarantee that no mistake is made in this step, because BFS only ensures to go to the neighboring vertices that are connected to the ones in the cycle and nothing more.

Problem 7. Let $G = (V, E)$ be a directed graph with weighted edges, edge weights can be positive, negative, or zero. Suppose vertices of G are partitioned into k disjoint subsets $V_1, V_2, \ldots, V_k$; that is, every vertex of G belongs to exactly one subset V_i. For each i and j, let $\delta(i, j)$ denote the minimum shortest-path distance between vertices in V_i and vertices in V_j, that is

$$\delta(i, j) = \min \{\text{dist}(u, v) \mid u \in V_i \text{ and } v \in V_j\}$$

Describe an algorithm to compute $\delta(i, j)$ for all i and j that runs in $O(VE+kV \log V)$ time. $O(V^3)$ algorithm will get partial credit.

(40 points)

Solution. To solve this question, there are three main steps that we can take. The first step is that since we are given all subsets (V1, V2....Vk), we can start by finding the the shortest path for all the pairs in each of the subsets, Vi, using the famous Bellman-Ford Algorithm that we learned in class. When we use the Bellman Ford Algorithm, we can make sure that we are using not only the positive and zero weights in the graph, but also accounting for the negative weights as well. Now that we found the shortest path pairs for the subset pairs themselves, we can make sure that we find the distances (shortest path) between the subsets themselves (Vi to Vj), by traversing through the edges (u, v) the vertice' pairs, to find the shortest path between the subsets. After doing that we can use Dijkstra's algorithm to compute the shortest path for each subset to every other subset that exists. That means, computing the distance from Vi to Va, Vb, Vc...Vk. This allows us to achieve our final goal of finding the minimum shortest path distance from any singular vertex in Subset V (i) , to another singular vertex in subset V(j), giving us our return output of (i,j). Now, let us examine the pseudo code that will help break down this algorithm:

```
find minVertexPath(Graph G, Vertices V, 2D array of edges, list of disjoint subsets):


int[][]diff=new int [subsets.size][subsets.size]

m=subsets.size
int [][]distanceArray=new int [V][V]
//start off by intializing ur two 2d arrays
to keep track of vertex
distances and subset differences

for(int i=0; i<m;i++):
    Arrays.fill(diff[i], INF) //max distance placeholder
    }

for (int i=0; i<V; i++){
    Arrays.fill(distanceArray[i], INF
    distanceArray[i][i]=0 //distance from vertex to itself will be 0
}

* now use the blackbox Bellman Ford Algorithm to compute the shortest paths
for every vertex in a
subset itself
    for (subset: subsets):
        for(vertex u in subset):
            bellmanFord(u, V, edges,distanceArray)

//traverse through every edge in the edges list and find the distances between
```

```
the different subsets in the
subset list themselves, by using
an assumed helper method we have
to see if a subset exists given an edge.
for (int[]edge:edges){
    a=edge[0]
    b=edge[1]
    c=edge[2]
    int dis1=subsetFinderHelperMethod(subsets,a)
    int dis2=subsetFinderHelperMethod(subsets,b)
    if (dis1 and dis2 are not -1){
        diff[dis1][dis2]=Math.min(diff[dis1][dis2], distanceArray[a][b])

    }
}

*last step is to use the Dijkstra's algorithm that we learned in class, a black box
algorithm that can help us find the
shortest path inter-subset
(between subsets), and return the
vertices (i,j) that
will get us this minimal subset path. Run it on our m*m graph.
```

Now that we described the algorithm step by step, let us **analyze the runtime of this algorithm.** We learned in class that the Bellman Ford Algorithm has a runtime of O(V E) as we are iterating over all V-1 vertices, where each iteration takes $O(E)$ time, where e represents the edges of course. That covers the first step of our algorithm. Next, to cover the second step of our algorithm, we want to find the distances between the subsets themselves. To do this, we need to iterate over all the edges, which once again takes an $O(E)$ runtime. And lastly, we want to find the shortest paths between the inter subsets, and for this we use Dijkstra's algorithm. We want to run Dijkstra's algorithm on our m*m graph, where m represents the number of subsets we are dealing with. In the case of this question, this will account to a runtime of $O(k^2 logk)$ for the Dijkstra's part of the algorithm. K is synonymous with our "m here". Our final algorithmic runtime for this question is the required $O(VE + kVlogV)$, because V dominates the k term as k is less than /equal to V.

Now, let us examine the **proof of correctness** for this algorithm:

1. We are using a preexisting black box algorithm (Bellman Ford) to find the shortest path between a source vertex , and all the other vertices in our graph, and this gets us our minimal path (u,v) for all vertices per subset in our graph.

2. Then we want to find the minimal distance between the subsets themselves, and thus, we iterate over all the edges, and update our value with the minimum distance from any vertex in subset Vi and any singular vertex in subset Vj. Since we are indeed iterating over every single vertex, our algorithm stands on its correctness because it guarantees that we are checking for the minimal distance between every vertex in the first subset and the the second subset.

3. Lastly, we are using Dijkstra's algorithm, which is again, a proven black box algorithm in order to find the shortest path between the different vertices in the subset graphs, returning our final output. These will find the 2 vertices (i, j) that we are looking for.

Problem 8. Assume you have an efficient algorithm to solve the maximum flow problem in a given flow network. Use this algorithm to reduce and solve the following problems:

a). Minimum Vertex-Disjoint Paths in a Directed Acyclic Graph (DAG): Given a directed acyclic graph $G = (V, E)$, determine the minimum number of vertex-disjoint paths needed to cover all vertices in G.

b). Cycle Cover in a Directed Graph: Given a directed graph $G = (V, E)$, find a cycle cover, which is a collection of vertex-disjoint cycles that cover every vertex in G, or correctly report that no cycle cover exists.

Reduce the problems to a network flow problems. Construct the corresponding flow network and explain how the solution to the maximum flow problem determines the answer. **(40 points)**

Solution. ***PART A*** Part A of this question asks us to return the minimal number of vertex-disjoint paths that are required to "cover all the vertices in G". We want to use a flow network approach to answer this question. The way that we can do this is by taking all the vertices and then continuing to split the graph's vertices into categories of "into" vertices and "outside" vertices. Then we want to add the directed edge that belongs in the into vertices to the corresponding outside vertex. If we have an edge that already belongs in our graph, then we want to ensure that we can change the direction of the edge so that it now goes from the "outside" vertex to the "into" vertex. To keep track of these edges that we are changing, we can create a SourceVertex tracker and a SinkVertex tracker (vertices). The way we want to add edges is from the SourceVertex to the "into" vertex in the graph, if it has no current incoming edges. Then we also want to add edges from the "outside" vertex, to the SourceVertex, given that there are already no outgoing edges present in the "outside" vertex. Now that we have done this, we want to calculate the maximum flow. In order to do this, we want to use the max-flow algorithm on our flow network. Our max flow value itself corresponds directly to the number of vertex-disjoint paths in our graph. The whole point of getting a max flow value is because it is able to represent the minimal number of paths that are required to hit all the vertices in our graph, G. Every path has a corresponding set of vertices that are used in this flow network. Let us look at the psuedo code that breaks down this algorithm step by step:

```
function flowA(Graph G):

//start by initalizing a flowNetwork
    flowNetwork(intoVertex, outsideVertex, capacity)=(null, null, 0);

    for every vertex in Graph G:
        flowNetwork.addVertex(inVertex)
        flowNetwork.addVertex(outVertex)
        flowNetwork.addEdge(inVertex, outVertex,1)
//here, I am splitting each vertex in the graph into either an "into" our "outside vertex.


for edge(u,v) in Graph G:
    flowNetwork.addEdge(outU, inVertex, infinity);
    //add our edges in the Graph to our flow network

flowNetwork.addVertex(SourceVertex)
flowNetwork.addVertex(SinkVertex)
for Vertex V in Graph g:
    if V.incomingEdge ==0 ://if there are no incoming edges,
    then redirect edge by
    adding to flow network
        flowNetwork.addEdge(SourceVertex,inVertex, infinity)
    if V.outgoingEdge==0: //same for outgoing edges
        flowNetwork.addEdge(outVertex, SinkVertex, infinity)

maxFlow=maxFlow(flowNetwork, SourceVertex, SinkVertex)
```

```
return maxFlow;
```

Now that we've listed the psuedo code for the algorithm, **let us analyze the proof of correctness** for this code. We can start with our base case which deals with the case of when the graph only has one singular vertex. If the graph has only one singular vertex, then the flowNetwork itself will start at that vertex (the source vertex), and then we will have an edge that connect from the inVertex and another to the outVertex. Since there is only one vertex in the graph itself, the maxFlow that can be returned is just 1. Now, for our inductive step we want to prove that the algorithm works for all k+1 cases, or k+1 vertices. The reason why this algorithm works for all k+1 cases is because of the splitting that occurs in the beginning of the algorithm. We are able to split all the vertices into either inVertex or outVertex, and that ensures that we have edges that connect from the source vertex to the in vertex, and then to the out vertex, and then back to the sink vertex. When we are trying to return the max flow, our parameters ensure we are examining all the vertices that connect to edges.

Now, lastly, let us look at the runtime for Part A. The runtime for this algorithm will be $O(VE)$. This is because we are relying on the maximum flow algorithm. When we are trying to construct the flow network, it takes of a runtime of $O(V + E)$. We also need to iterate through all our edges and vertices, which takes $O(VE)$, and this is what dominates the overall runtime for the algorithm.

PART B Now, let us look at part B for the algorithm. The question asks us to find the set of vertex-disjoint cycles that hit all the vertices in Graph G. If there are no cycle covers to be found, then we need to return -1. We want to once again, use the maximum flow approach via a flow network. Because we need to be able to split our vertices, we can take an approach that leverages a structure that is closely related to a bipartite graph, to keep track of our disjoint sets. We want to use the same logic of splitting our vertices in our graph into inVertex and outVertex, and then, we want to clearly add an edge between our in and out vertices, so that we can ensure that no repeating cycle occurs. We want our edge to have a capacity of 1. We also want to make sure that we can add a directed edge every time that we encounter an edge. Similar to part A, we will need to keep track of our SourceVertex and SinkVertex. We also would like to use a helper function that actually checks if a cycle cover is present by seeing if the maximum flow is equal to the total number of vertices in the graph, or not. Let us examine the psuedo code for this algorithm:

```
function flowCycle(Graph G):

//start by initalizing a flowNetwork
    flowNetwork(intoVertex, outsideVertex, capacity)=(null, null, 0);

    for every vertex in Graph G:
        flowNetwork.addVertex(inVertex)
        flowNetwork.addVertex(outVertex)
        flowNetwork.addEdge(inVertex, outVertex,1)
//here, I am splitting each vertex in the graph into either an "into" our "outside vertex.


for edge(u,v) in Graph G:
    flowNetwork.addEdge(outU, inVertex, infinity);
    //add our edges in the Graph to our flow network

flowNetwork.addVertex(SourceVertex)
flowNetwork.addVertex(SinkVertex)
for Vertex V in Graph g:
    if V.incomingEdge ==0 ://if there are no incoming edges,
    then redirect edge by
    adding to flow network
```

```
            flowNetwork.addEdge(SourceVertex,inVertex, infinity)
        if V.outgoingEdge==0: //same for outgoing edges
            flowNetwork.addEdge(outVertex, SinkVertex, infinity)

    isCycle(flowNetwork);



    function isCycle(flowNetwork):
        if maxFlow==V
            return flowCycle(flowNetwork)
            //cycle does exist and cover is returned
        else
            return -1;
            //shows that the cycle doesn't exist
```

Our **runtime analysis**, is the exact same as Part A because we are relying on the main code from Part A with a helper function that simply has an if statement. The runtime of $O(VE)$ from part A dominates.

Let's look at the **proof of correctness** for this algorithm:

1. Base Case: Occurs when there is 1 vertex and O edges. In this case, the network starts at the source vertex, has an edge to inVertex, then to outVertex, and then goes to SinkVertex. Since the graph will only have one vertex where the cycle is formed, the capacity will be 1, and a cycle will be correctly identified.
2. Inductive Step: For the inductive step, we want to prove that we can correctly identify the cycle for all k+1 vertices in our graph. The main reason this works is because we are able to split all our vertices into either categories of in or out, and then add directed edges between the in, out, SourceVertex, and SinkVertex. After we properly construct our flowNetwork, our isCycle function uses simple if checks to see if a cycle actually exists by comparing the vertices to the maxFlow number.
