Q1. (10 points)

Given an undirected graph G, describe a linear time algorithm to find the number of distinct shortest paths between two given vertices u and v. Note that two shortest paths are distinct if they have at least one edge that is different.

The algorithm is a variation of the breadth-first search algorithm we covered in class. Let d be the layer of nodes d hops away from the starting point u that have currently been explored by the algorithm (assume that d = 0 refers to starting vertex u). Let d+1 be the layer of nodes d+1 hops away from the starting point u (basically the non visited neighbors of d) that is currently being explored.

If v is found in layer d+1, then find all the other vertices in d that are neighbors of v, catalogue them, and end the algorithm.

The algorithm is built off BFS so the worst case (a scenario in which the two vertices u and v are on opposite sides of the graph) running time is O(E+V) because every vertex and edge will have to be explored to eventually find v.

Q2. (10 points)

Given a weighted directed graph with positive weights, given a O(|V|^3) algorithm to find the length of the shortest cycle or report that the graph is acyclic.

The algorithm calls a modified version of Dijkstra's on every vertex in the graph.

The algorithm is modified such that when it backtracks to update distances after extracting the current minimum, it will no longer consider 0 to be "small". That way if there's a cycle that runs back to the node the algorithm was initially called on, the 0 placeholding value will be replaced. The code could be modified like so:

Normally Dijkstra's replaces larger paths with shorter ones like so:

```
if dist(v) > dist(u) + l(u, v):
        dist(v) = dist(u) + l(u, v)
```

We'd change that to exclude 0s as "small" values:

```
if dist(v) > dist(u) + l(u, v) OR dist(u)  == 0:
        dist(v) = dist(u) + l(u, v)
```

The other modification has to do with the storage of shortest paths. Imagine a 2D array – paths[x][y] -- that contains shortest distances between all pairs of vertexes. Each call of Dijkstra's will update a row of the matrix – calling on the source vertex x1 will update the row with shortest distances between x1 and y1, y2, y3, etc… where yn represents every vertex in the graph.

When the algorithm is done, traverse the matrix structure along the left to right diagonal checking the shortest distance between every vertex and itself. This gets us the shortest cycle between every vertex if one exists. Keep track of the shortest cycle of them all and output it. If every cell checked has the value 0, then we know the graph is acyclic.

With an array based implementation, Dijkstra's runs in V^2. If we call Dijkstra's on every vertex once, the total runtime is therefore v^3.

Q3. (10 points)

Given a directed weighted graph G, with positive weights on the edges, let us also add positive weights on the nodes. Let l(x,y) denote the weight of an edge (x,y), and let w(x) denote the weight of a vertex x. Define the cost of a path as the sum of the weights of all the edges and vertices on that path. Give an efficient algorithm to find all the smallest cost path (as defined above) from a source vertex to all other vertices. Analyze and report the running time of your algorithm.

For this problem we can use another slightly modified version of Dijkstra's algorithm. When calculating the length / cost of a path, we just have to factor in the weight of the vertex at the end of a corresponding edge. Something along the lines of changing:

```
u = deletemin(H)
for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
                dist(v) = dist(u) + l(u, v)
```

to

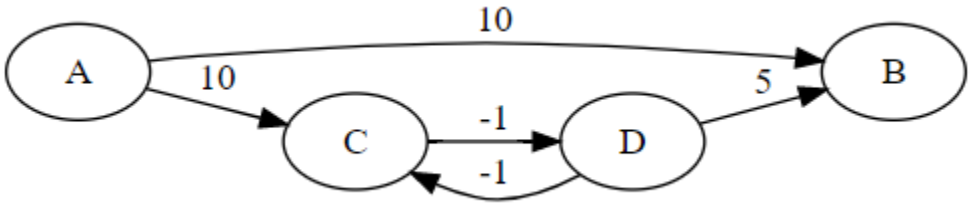
```
u = deletemin(H)
for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v) + w(u):
                dist(v) = dist(u) + l(u, v) + w(u)
```

This doesn't add any meaningful computation to the original Dijkstra's algorithm (the extra addition is O(1)), so it still has the runtime of (V + E)log(V) with an optimal implementation.

Q4. (10 points)

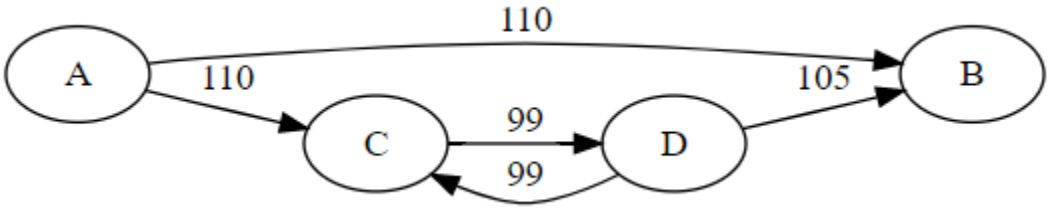
Given a directed graph G with possibly negative edge weights. Consider the following algorithm to find the shortest paths from a source vertex to all other vertices. Pick some large constant c, and add it to the weight of each edge, so that there are no negative weights. Now, just run Dijkstra's algorithm to find all the shortest paths. Is this method correct? If yes, reason why. If not, give a counterexample.

The alteration does not work. A simple counter example is the case of negative cycles. Negative cycles allow a path to become indefinitely short – one can always go through the negative cycle again to get a smaller total distance travelled. Here is a visual example:



In this example, it is clear the shortest path from A to B involves going from A to C, then C to D and back to D to C a potentially infinite number of times (but at least 5), then D to B.

If we remove all negative numbers from the graph by adding a large positive number, say 100, here is what the graph would look like:



With only positive numbers it's clear that the shortest path from A to B is straight across the 110 weighted path. Because the two paths differ, it is clear that the proposed adjustment cannot properly compute shortest paths in the presence of negative weights.