Intro to Algorithms Homework 6
Anooj Pai

Q1) 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.

We can solve this using BFS because they have a linear time of O(v+e), where v is the number of vertices and e is the number of edges.

Start by initializing two arrays, one to hold the number of shortest paths, called paths, and a second one to hold the shortest distances from u to v and call this distances. Initially, only the source vertex will have a value that will be 0 for paths and 0 for distances and the rest will be infinity. Then we iterate through the graph:

```
For neighbor to v:
    If distances[v] > (distances[u] + 1):
        distances[v] = distances[u] + 1
        paths[u] = paths[x]
    else if distances[v] == (distances[u] + 1):
        paths[v] = paths[v] + paths[u]
```

Q2) 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.

This algorithm would be a modified version of Dijskra's algorithm to fit the runtime. The runtime is (|V|^3) because it uses a BFS where the vertices can be the start and end vertex which makes it run for V^2 on top of the first iteration.

```
for all u in V:
    for all v in V:
        path[u][v] = infinity

for all s in V:
    path[s][s] = 0
    H = makequeue (V)

    while H is not empty:
        u = deletemin(H)

        for all edges (u,v) in E:
            if path[s][v] > path[s][u] + l(u, v) or path[s][s] == 0:
            path[s][v] = path[s][u] + l(u,v)
            decreaseKey(H, v)

lenMin = MAX

for all v in V:
    if path[v][v] < lenMin & path[v][v] != 0 :
        lenMin = path[v][v]
        if lenMin == MAX:
                print("The graph is acyclic.")

        else:
                Min length is MAX
```

Q3) 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.

To solve this we can use Dijkstra's algorithm, the code from lab 7 is similar but we need to make some changes. Here is the algorithm that will find the distance between the starting node and all of the other nodes accounting for weight:

File format: startNode endNode weight
Runtime: O(log(|v|)

```
def dijkstras(startNode, edges):
        distances = { v: infinity for v in edges.keys() }
        distances[startNode] = vertex(startNode)
        priorityQueue = h.heapdict(distances)
        while priorityQueue is not empty:
                u, d = priorityQuere.pop()
                If u in edges:
                        for v,w in edges[u].items():
                                tempDistances = distances[u] + w + vertex[v]
                                If tempDistances < distances.get(v, infinity):
                                        Distances[v] = tempDistances
                                        priorityQueue[v] = tempDistances

        return Distances

edges = {}
vertices = {}
file = open(data.txt)
for line in file:
        If line.split()[0] not in edges:
                edges[int(line.split()[0])][int(line.split()[1])] = int(line.split()[2])

        vertices[int(line.split()[0])] = int(line.split()[1])

file.close()

distances = dijkstras(1, edges)
for i in range(1, max(edges.keys()+1)):
        print(str(i) + ' shortest: ' + str(distances[i]))
```

Q4) 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.

This method would be incorrect, below I have given a counter-example:

Let us look at a graph with 3 vertices: A, B, and C with the edge from A to B costing 10, A to C costing 30, and C to B costing -20. The shortest path from A to B has a cost of 10 with the path being ACB. If we added 20 to each edge to make none of them 0, the same path ACB would then cost 50 instead of 10. Because the new shortest distance is 50 it is not the same as the original graph meaning that the method is wrong.