Elad Ohayon

My solution models the problem through an undirected edge-weighted graph where vertices model the cities, edges model the highways, and edge weights model the duration of the highway. The algorithm then conducts a modified bfs on the graph to obtain all possible paths and returns the shortest duration path along with the number of such paths.

Algorithm: FindMinyan

Input: V = # of cities

*G*raph G= (V, E) constructed from calls to addHighway which create an undirected edge-weighted graph in the form of an adjacency list. No negative edge weights.

Output: shortestDuration: shortest duration trip from 1 to n, *numberOfShortestTrips*: number of trips with duration== *shortestDuration*

Solution (solveIt): This algorithm works by conducting a modified bfs starting with vertex 1. This “new” bfs is identical to a regular BFS but it cycles vertices once (to allow for cases where backtracking is necessary), creates units to store the weight accumulated up until the bfs' current position, and records all instances which reach n. To obtain O (1) amortized time with *shortestDuration* and *numberOfShortestTrips* the algorithm updates the shortest path each time a unit reaches vertex n (rather than searching once the BFS terminates) and tracks the frequency of each duration. The frequency of shortestDuration is the *numberOfShortestTrips.* The algorithm uses a sentinel value so it can detect if case S-T have no path.

Claim: My algorithm finds the shortest duration path and returns the number of unique paths if such a path exists or 0 if a path does not exist.

Axiom:

BFS will visit every vertex and edge connected to the source vertex (1)

Proof: Suppose that G=(V,E) is an undirected edge-weighted graph with vertices V and edges E where there are no negative edges weights.

Let **Z**= The shortest path in **G** and **Y**= the number of paths with length **Z**.

1. Let S=1 be our start vertex and T=|V| be our destination. BFS will be able to **find Z and Y if they exist and tell us if they do not.**
2. We can split problem into 3 cases shown below:
    a. **Case 1:** ∃ a path between S and T
        i. **Case 1.1:** ∃ N paths of the same length where N= total # of paths from S to T
            1. If there are N paths with the same length, then all paths are the shortest path since there is no smaller path.

2. **Corollary 1.1:** There can be more than one shortest path.

ii. **Case 1.2**: ∃ [N-1, 0] paths with the same length.

1. If there are 1 or more paths that aren't identical to each other ∃ a shortest path since their weights aren't equal. **Corollary 1.1** proves there can be more than 1.

b. **Case 2**: ∄ a path from S to T

i. **Case 2.1**: As there is no path from S to T then there also isn't a shortest path between the 2.

3. Using the property of Axiom 1 and recording every path BFS takes then if ∃ a path we will have recorded it.
4. The algorithm then uses arithmetical compares to find the shortest duration path and returns its count or if ∄ a path from S-T then ∄$(\mathbf{Z},\mathbf{Y})$ and the algorithm returns 0 accordingly.

The 2 dominating factors in determining the algorithm's runtime is the cost to add an edge when building the graph and the runtime of solveIt(). The cost to add an edge is min(degree[v], degree[w]) as we must check if there are any parallel edges when adding an edge. Due to the properties of an undirected graph, we only check the vertex with less edges. The algorithm *solveIt* consists of a modified bfs that adds each visited vertex back on the queue once (to solve a specific edge case). This means it visits all vertices and their edges twice, making the runtime O(2(E+V)). We make this algorithm faster by not adding paths which have a longer duration than the current shortestDuration(). However, this becomes less effective as shortestDuration becomes longer than more paths or all paths grow at a slow rate, which in this case, they would only be discarded at the end of the BFS.