Code Jam 2017 - Round 1B

Analysis: Pony Express

Pony Express: Analysis

Small dataset

In the Small dataset, we are given a line of cities and the distances between them, and we want to go from the first to the last city in the line. However, we need to minimize the time and not the distance, and the obvious way to transform between the two (divide by speed) does not work directly, because the horses we can use have different speeds. The solution, as in many minimization algorithms on a path that only moves forward, is to use dynamic programming.

Let us define f(i) as the quickest way to get from city i to the finish line, assuming we start using the horse at city i. Of course, f(1) is the answer to the only query.

We do not know where we need to change horses, but we can try every possible intermediate city j to change horses, thus completely defining f(i) by:

- f(i) = min { distance(i, j) / speed(i) + f(j) : for all j such that distance(i, j) ≤ endurance(i) }, for i < **N**.
- f(**N**) = 0.

This solves the problem in O($\mathbf{N}^2$) because the domain of f is of size **N** and the iteration to calculate each value of the domain takes time O(**N**) if we memoize the results. There are other dynamic programming approaches that also work in the same amount of time.

Large dataset

In the Large dataset, we are given a graph G of cities with distances between them instead of just a line, but the setup of the problem is otherwise the same.

As mentioned above, one relatively unusual feature of this graph problem is that the desired result (time) is not in the same units as the weights of the graph's edges (distance). Moreover, the obvious way to transform between the two (speed) is not a constant. This observation leads to a key insight: we should construct a new graph with weights of time instead of distance. That way, we can apply a shortest path algorithm to get the result we want.

We can do this by defining a graph G' that has the same nodes as G, but in which the weight of edge (i, j) is the time that it takes to go from city i to city j. As we said above, there is no fixed speed, but we can fix it. The edge (i, j) then represents the time it takes to go from city i to city j using a single horse, and the obvious choice is to use the horse at the departure city i. Let us call that horse h. In this way, the edge (i, j) on G' represents a path on G, traversed using a single horse. That means the edge (i, j) exists if and only if there is a path between i and j in G with distance less than or equal to h's endurance. Moreover, the weight (i, j) is a time now defined by a single speed, h's speed, and thus it is just the distance between i and j in G, divided by h's speed. You can see that a minimum path from a to b in G' represents a succession of edges in G', that is, a succession of single-horse paths in G, which is exactly what a solution looks like!

Since the limits are low enough, and we need all shortest paths in G to construct G', and many shortest paths in G' to answer many queries, the best option is to use an all-pairs shortest path algorithm. Floyd-Warshall is the easiest and fastest to implement, but others would work too. To

also check for existence of paths, you can use the trick of setting weights to "infinity", that is, a distance too large for an actual shortest path (larger than maximum distance × number of total edges). Thus, an infinity distance simply means the edge or path does not exist in the graph.

To summarize, this pseudocode solves the problem:

1. Apply Floyd-Warshall to input G getting distances between all pairs of nodes.
2. Create G' by adding all edges (i, j) such that the distance between i and j in G is less than or equal to the horse starting at city i's endurance, and set their weights to that same distance divided by that horse's speed.
3. Apply Floyd-Warshall to G' to get minimum times between all pairs of nodes.
4. Read queries and answer immediately from the output of the last step.

This solution takes time $O(\mathbf{N}^3)$. Both uses of Floyd-Warshall take time $O(\mathbf{N}^3)$, and creating G' as an adjacency matrix only takes time $O(\mathbf{N}^2)$.