1. Describe and analyze an algorithm to compute the shortest path from vertex s to vertex t in a directed graph with weighted edges, where exactly *one* edge $u \to v$ has negative weight. First check whether G has a negative length cycle. Then, find the shortest path length from s to t. *[Hint: Modify the input graph and run Dijkstra's algorithm.]*

Solution: Let G denote the input graph, let $w(x \to y)$ denote the weight of edge $x \to y$, and let $u \to v$ denote the unique edge in G with negative weight.

Remove edge $u \to v$ from G and let G' denote the resulting graph. Note that G' has no negative length edges. For any nodes x and y, let $dist(x, y)$ and $dist'(x, y)$ denote the distances from x to y in G and G', respectively.

If G has a negative length cycle then it must contain the edge $x \to y$: the shortest length cycle containing this arc can be seen to consist of a shortest path P from y to x in G' together with the arc $x \to y$. The length of this cycle is $dist'(y, x) + w(x \to y)$. G has a negative length cycle iff this quantity is negative. Thus, we can check if G has a negative length cycle by computing $dist'(y, x)$ in G' via Dijkstra's algorithm.

Suppose G does not have a negative length cycle. The shortest path in G from s to t either traverses the edge $u \to v$ or it doesn't; we consider each case separately. Then we have

$$dist(s, t) = \min \left\{ \begin{array}{c} dist'(s, t) \\ dist'(s, u) + w(u \to v) + dist'(v, t) \end{array} \right\}$$

Thus, we can compute $dist(s, t)$ by running Dijkstra *twice* in G': once starting at s to compute both $dist'(s, t)$ and $dist'(s, u)$, and once starting from v to compute $dist'(v, t)$. The algorithm runs in $\boldsymbol{O(E \log V)}$ ***time***. ■

2. You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving toward each other. But where *exactly* should you meet?

   You are given a weighted graph $G = (V, E)$, where the vertices V represent cities and the edges E represent roads that directly connect cities. Each edge e has a weight $w(e)$ equal to the time required to travel between the two cities. You are also given a vertex p, representing your starting location, and a vertex q, representing your friend's starting location.

   Describe and analyze an algorithm to find the target vertex t that allows you and your friend to meet as soon as possible, assuming both of you leave home *right now*.

Solution: Let $dist(x, y)$ denote the shortest-path distance from vertex x to vertex y. If my friend and I decide to meet at some vertex t, and we leave our respective vertices at time 0, then we will meet at time $\max\{dist(p,t), dist(q,t)\}$. We can compute this meeting time for *all* vertices t by running Dijkstra's algorithm twice—once from p and once from q—and then scanning through the vertices. The overall running time is $O(E \log V)$.

```
WhereToMeet(G, p, q):
  distp ← Dijkstra(G, p)      ⟨⟨distp(v) ← shortest-path distance from p to v, for all v⟩⟩
  distq ← Dijkstra(G, q)      ⟨⟨distq(v) ← shortest-path distance from q to v, for all v⟩⟩
  time ← ∞
  for all vertices t
      if time > max{distp(t), distq(t)}
          time ← max{distp(t), distq(t)}
          best ← t
  return best
```

■

Jeff gave this problem on an exam. Several students proposed the following alternate solutions, all of which are incorrect:

- ***Find the vertex t that minimizes $\mathbf{dist}(p,t) + \mathbf{dist}(q,t)$.*** This function is actually minimized for any vertex t on the shortest path from p to q—in particular, when $t = p$ or $t = q$.
- ***Find the midpoint of the shortest path from p to q.*** See the counterexample below. The best meeting point is the center vertex (at time 5), but the shortest path from p to q is a single edge; the midpoint of this path is not even a vertex.
- ***Find the vertex t that minimizes $|\mathbf{dist}(p,t) - \mathbf{dist}(q,t)|$.*** This function could be minimized on the moon if we can get there at *exactly* the same time. In the counterexample below, the difference in travel times is minimized at the top middle vertex (at time 9).
- ***Use the unique path from p to q in the minimum spanning tree.*** Shortest paths and minimum spanning trees don't have anything to do with each other.

To think about later:

3. Let $G = (V, E)$ be a directed graph with edge length $\ell : E \to \mathbb{R}^+$. A subset of the edges $E' \subseteq E$ are considered risky. Describe an algorithm that given $G = (V, E)$, the edge lengths ℓ, the risky subset E', a node s and an integer h finds for each node $v \in V$ the shortest path distance from s to v among all paths that contain at most h risky edges. *Hint:* Apply the dynamic programming idea behind Bellman-Ford algorithm.

Solution: We give two solutions. The first is explicitly based on dynamic programming while the other uses it implicitly.

Solution 1

Let $d(v, i, j)$ be the minimum distance from s to v among all paths containing at most i total edges and at most j risky edges. Let G' be the graph obtained by removing all the risky edges in G. Then $d(v, i, 0)$ is the shortest path distance from s to v using paths with at most i edges in G'. Using Bellman-Ford algorithm on G' we can compute $d(v, i, 0)$ for all $v \in V$ and all $0 \leq i \leq n-1$ in $O(mn)$ time. For $j > 0$,

$$d(v, i, j) = \min \begin{cases} d(v, i-1, j) \\ d(v, i, j-1) \\ \min_{(u,v) \in E'} \; d(u, i-1, j-1) + \ell(u, v) \\ \min_{(u,v) \in E - E'} \; d(u, i-1, j) + \ell(u, v) \end{cases}$$

And what we wish to compute is $d(v, n-1, k)$ for all $v \in V$. There are at most $n^2 k$ subproblems. To compute $d(v, i, j)$ via the recursive definition above we need time that is proportional to the in-degree of v. Thus the total time to compute $d(v, i, j)$ for all v is $O(m)$, and hence the overall time is $O(mnk)$

Note: It may be tempting to try to define $d(v, j)$ as the shortest path distance from s to v using at most j risky edges. Why does the recursion not work with this definition?

Solution 2

Let $G' = (V, E - E')$, i.e., the original graph with all risky edges removed.

An alternative approach that can use Dijkstra's algorithm is to create $h + 1$ copies of G': $G_0, G_1, \ldots, G_h$. Then, include a directed edge from vertex u in G_i to vertex v in G_{i+1} if (u, v) is a risky edge in G. The idea is that the only way a path can move from one copy of G' to the next is by traversing a risky edge. Now run Dijkstra's algorithm on this new graph, from vertex s_0, the copy of s in G_0. Suppose v is a vertex in G, and let $v_0, \ldots, v_h$ be the corresponding vertices in copies $G_0, \ldots, G_h$. Then $d(s_0, v_i)$ is just the shortest path from s to v in the original graph G that uses exactly i risky edges. Thus, the distance from s to v in the original graph that uses at most h risky edges is just $\min_{0 \leq i \leq h} d(s_0, v_i)$.

To evaluate the running time of this algorithm we notice that the new graph has $O(nk)$ nodes and $O(mk)$ edges and we run Dijkstra's algorithm on this graph and hence the run-time is $O(mk + nk \log(nk))$ which is $O(k(m + n \log n))$; this is equivalent to running Dijkstra's algorithm k times.

Note: One may wonder why the reduction approach in the second solution results in a faster running time. The reason for this is because the DP based solution can in fact solve a more general problem, namely when the edge have negative lengths. ■