Homework 8

CS/ECE 374 B

Due 8 p.m. Tuesday, November 19

1. (This question is the same as the textbook question 6(a)(b).)

(5) (a) Describe and analyze a modification of Bellman–Ford that actually returns a negative cycle if any such cycle is reachable from s, or a shortest-path tree if there is no such cycle. The modified algorithm should still run in $O(VE)$ time.

(5) (b) Describe and analyze a modification of Bellman-Ford that computes the correct shortest path distances from s to every other vertex of the input graph, even if the graph contains negative cycles. Specifically, if any walk from s to v contains a negative cycle, your algorithm should end with $\text{dist}(v) = \infty$; otherwise, $\text{dist}(v)$ should contain the length of the shortest path from s to v. The modified algorithm should still run in $O(VE)$ time.

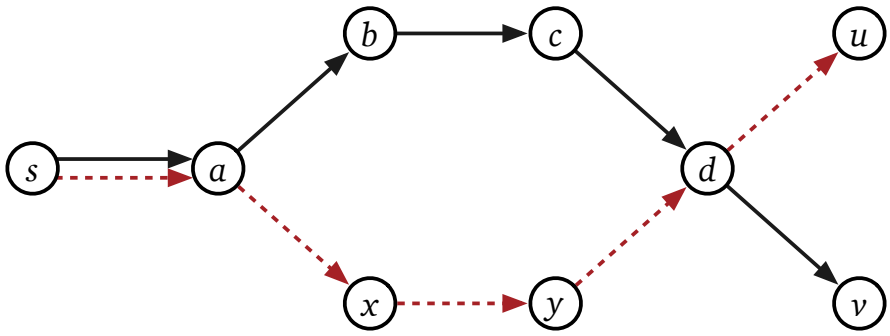
2. (This is question 24 in the textbook.)

You are given a directed graph G that represents bus travel in a city. For an edge $u \to v$, $\ell(u,v)$ is the amount of time it takes for the bus to travel from bus stop u to bus stop v. You are also given two numbers for each stop to determine the schedule: $f(u,v)$ is the first time (in minutes past 12 noon) that the bus leaves from stop u towards stop v, and $\Delta(u,v)$ is the time between such buses. In other words, a bus leaves from stop u to stop v at $f(u,v) + k \cdot \Delta(u,v)$ for $k \geq 0$, and arrives at stop v at times $f(u,v) + k \cdot \Delta(u,v) + \ell(u,v)$. Assume that $\ell(u,v), f(u,v)$, and $\Delta(u,v)$ are all positive numbers.

You are given a graph G and the values ℓ, f, and δ for every edge in G. You are also given the starting stop s and the destination stop t, and the starting time t_0. You need to design and analyze an efficient algorithm to determine the earliest time you can arrive from s to t starting at time t_0. Note that you can change buses at a stop if you arrive at a time that is less than or equal to the departure time of the bus. That is:

$$f(u,v) + k_1 \cdot \Delta(u,v) + \ell(u,v) \leq f(v,w) + k_2 \cdot \Delta(v,w)$$

3. On the Internet, routing paths follow a local preference rule: each Internet service provider (ISP) makes a local decision between possible routes based on its preferences, usually guided by complex business decisions. These preferences may depend not just on the previous hop but the entire path to the destination. For example, in the graph below, there are two paths from s to d. The path that is chosen is based on d's local preference, which can be computed by a function $\text{pref}(v,p)$ where v is the identity of the graph node where the path terminates, p is a path from s to v. The function returns a number such that $\text{pref}(v,p_a) > \text{pref}(v,p_b)$ means that path p_a is preferred to path p_b. In the graph below, to decide which path to use from s to d you would need to compare $\text{pref}(d,(s,a,b,c,d))$ and $\text{pref}(d,(s,a,x,y,d))$.



Note that each ISP (i.e., nodes in the graph) chooses only which of the incoming edges to use for its path. Therefore, if d chooses the red path (s, a, x, y, d) then the only path available to u is (s, a, x, y, d, u). In other words, the only decision to be made in this graph is at d, and therefore the set of paths taken through the graph can be made into a preferred path tree.

(6) (a) Design and analyze an efficient algorithm for computing the preferred path tree according to a given pref function. You are given a directed, unweighted graph $G = (V, E)$ as input and a source node s. You also have access to a local preference function pref. In this part, you should only use the pref function to decide between paths that are of equal length. You should assume that preftakes time proportional to the size of its arguments (i.e., $O(|p|)$, the length of the path).

Your result should be a tree T that satisfies the following properties:

- The path from s to v in the preferred path tree T, path(T, s, v), is a shortest path from s to v
- If $v \to u$ is an edge in T and $w \to u$ is another incoming edge to u, then either:
  - path(T, s, w) is longer than path(T, s, v), or
  - pref$(u,$ path$(T, s, u))$ is higher than pref$(u,$ path$(T, s, w) + w \to u)$

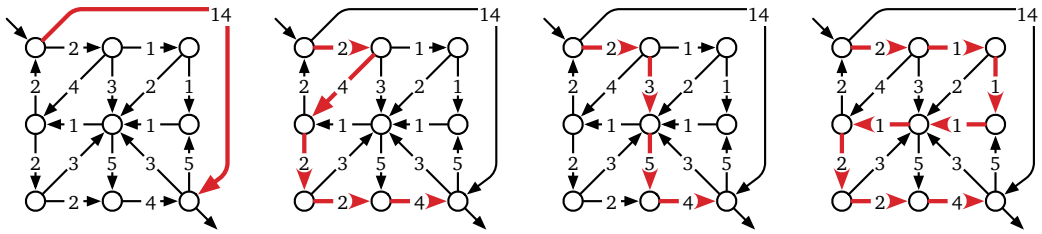
(4) (b) Design an algorithm for computing the preferred path tree where longer paths may be preferred. Again, you are given a directed, unweighted graph G, a source node s, and access to the pref function. The only constraint is that you cannot select paths that would result in a loop.) Your algorithm should return a preferred path tree T with the following property: if $v \to u$ is an edge in T and $w \to u$ is another incoming edge to u then either:

- pref$(u,$ path$(T, s, u)) >$ pref$(u,$ path$(T, s, w) + w \to u)$, or
- path(T, s, w) traverses u (i.e., adding $w \to u$ to the tree would create a loop)

You do not have to analyze the runtime complexity of this algorithm.

Solved Problem

4. Although we typically speak of "the" shortest path between two nodes, a single graph could contain several minimum-length paths with the same endpoints.



Four (of many) equal-length shortest paths.

Describe and analyze an algorithm to determine the number of shortest paths from a source vertex s to a target vertex t in an arbitrary directed graph G with weighted edges. You may assume that all edge weights are positive and that all necessary arithmetic operations can be performed in $O(1)$ time.

[Hint: Compute shortest path distances from s to every other vertex. Throw away all edges that cannot be part of a shortest path from s to another vertex. What's left?]

Solution: We start by computing shortest-path distances $dist(v)$ from s to v, for every vertex v, using Dijkstra's algorithm. Call an edge $u \to v$ **tight** if $dist(u) + w(u \to v) = dist(v)$. Every edge in a shortest path from s to t must be tight. Conversely, every path from s to t that uses only tight edges has total length $dist(t)$ and is therefore a shortest path!

Let H be the subgraph of all tight edges in G. We can easily construct H in $O(V + E)$ time. Because all edge weights are positive, H is a directed acyclic graph. It remains only to count the number of paths from s to t in H.

For any vertex v, let $\underline{PathsToT}(v)$ denote the number of paths in H from v to t; we need to compute $\underline{PathsToT}(s)$. This function satisfies the following simple recurrence:

$$\underline{PathsToT}(v) = \begin{cases} 1 & \text{if } v = t \\ \displaystyle\sum_{v \to w} \underline{PathsToT}(w) & \text{otherwise} \end{cases}$$

In particular, if v is a sink but $v \neq t$ (and thus there are no paths from v to t), this recurrence correctly gives us $\underline{PathsToT}(v) = \sum \varnothing = 0$.

We can memoize this function into the graph itself, storing each value $\underline{PathsToT}(v)$ at the corresponding vertex v. Since each subproblem depends only on its successors in H, we can compute $\underline{PathsToT}(v)$ for all vertices v by considering the vertices in reverse topological order, or equivalently, by performing a depth-first search of H starting at s. The resulting algorithm runs in $O(V + E)$ time.

The overall running time of the algorithm is dominated by Dijkstra's algorithm in the preprocessing phase, which runs in $\underline{\boldsymbol{O(E \log V)}}$ **time**. ■

Rubric: 10 points = 5 points for reduction to counting paths in a dag + 5 points for the path-counting algorithm (standard dynamic programming rubric)