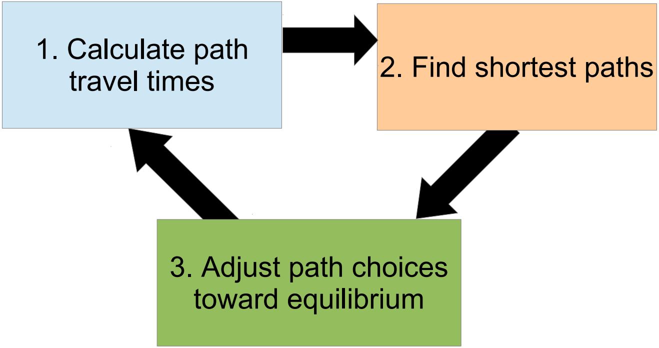
Shortest Paths on a Network

CE 392C



We already know how to calculate path travel times from path flows (step 1); let's now focus on step 2.

SHORTEST PATH CONCEPTS

This is commonly known as the *shortest path* problem. With modern computers, it's possible to find shortest paths in a fraction of a second, even for large networks.

In a shortest path problem, we are given a network $G = (N, A)$ in which each link has a *fixed* cost t_{ij}, an origin r, and a destination s. The goal is to find the path in G from r to s with minimum travel time.

To find this path efficiently, we need to avoid enumerating every possible path.

One odd twist of shortest path problems: it's not much harder to find the shortest path from r to s than to find many shortest paths at the same time. Two broad approaches:

One-to-all: Find the shortest paths from node r to *all* destination nodes.

All-to-one: Find the shortest paths from *all* origin nodes to node s.

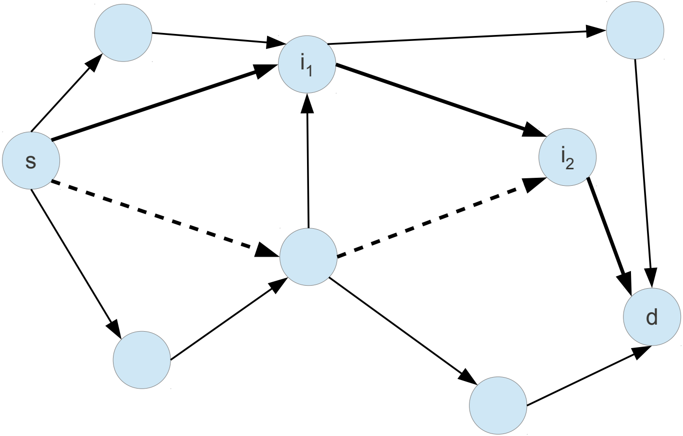
For the purposes of this course, either will work. For clarity, we'll stick with one-to-all shortest paths.

One-to-all shortest path relies on **Bellman's Principle**, which lets us re-use information between different origins and destinations:

If $\pi^* = [r, i_1, i_2, \ldots, i_n, s]$ is a shortest path from r to s, then the subpath $[r, i_1, \ldots, i_k]$ is a shortest path from r to i_k

The upshot: we don't have to consider the *entire* route from s to d at once. Instead, we can break it up into smaller, easier problems. (This is why the "one-to-all" problem is no harder than the "one-to-one" problem.)

Why does Bellman's principle hold?



If there is a shorter path from r to i_k, I could "splice" that into π^* and obtain a shorter path from r to s.

A compact way to store all of the shortest paths from r to every other node is to maintain two labels L_i^r and q_i^r for each node.

- L_i^r is the *cost label*, giving the travel time on the shortest known path from r to i.
- q_i^r is the *backnode label*, which specifies the previous node on the shortest known path from r to i.

By convention, $L_r^r = 0$ and $q_r^r = -1$; $L_i^r = \infty$ and $q_i^r = -1$ if we haven't yet found any path from r to i

SHORTEST PATHS IN ACYCLIC NETWORKS

In acyclic networks, Bellman's principle leads directly to an easy solution method.

Why acyclic networks? First, they're simpler and faster, and make a good first illustration. Second, many advanced traffic assignment algorithms operate by splitting a network into acyclic portions and using the easy method.

A defining characteristic of acyclic networks is the existence of a *topological order* — the nodes can be labeled from 1 to n in a way that every link connects a lower-label node to a higher-label one.

Theorem. A network has a topological order iff it is acyclic.
Proof. An exercise for you.

To find the shortest path from the origin r to all nodes, simply proceed in topological order and apply Bellman's principle:

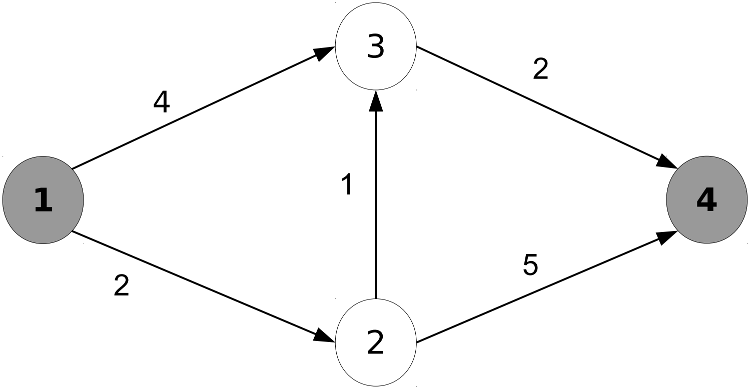
1. Initialize by setting $L_i^r = \infty$ and q_i^r $\forall i \in N$, and set $L_r^r = 0$
2. Let i be the node topologically following r.
3. (*By this point, we have found the shortest path from r to all nodes topologically before i*)
4. Find the best path from r to i by looking at each of the tail nodes one could arrive from:

$$L_i^r = \min_{(h,i) \in A} \{L_h^r + t_{hi}\}$$

$$q_i^r = \arg \min_{(h,i) \in A} \{L_h^r + t_{hi}\}$$

5. Is i the last node topologically? If so, stop. Otherwise, let i be the next node topologically and return to step 3.

Example



SHORTEST PATHS ON NETWORKS WITH CYCLES

If the network has cycles, there is no topological order and a different approach is needed.

Instead of scanning nodes in a predetermined order, fan out from the origin one node at a time.

Because of cycles, a node may be scanned more than once.

We maintain a *scan eligible list SEL* of nodes which still need to be scanned before we are sure all shortest paths have been found.

1. Initialize by setting $L_i^r = \infty$ and q_i^r $\forall i \in N$, and set $L_r^r = 0$
2. Initialize SEL to contain all nodes adjacent to the origin: $SEL \leftarrow \{i : (r, i) \in A\}$
3. Choose a node $i \in SEL$ and remove it from that list.
4. Scan node i as before:

$$L_i^r = \min_{(h,i) \in A} \{L_h^r + t_{hi}\}$$

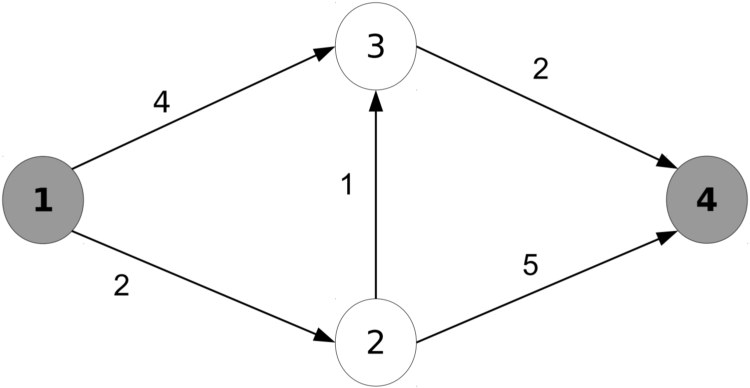
$$q_i^r = \arg \min_{(h,i) \in A} \{L_h^r + t_{hi}\}$$

5. If the previous step changed the value of L_i^r, then add all nodes immediately downstream of i to SEL:

$$SEL \leftarrow SEL \cup \{j : (i, j) \in A\}$$

6. If SEL is empty, then terminate. Otherwise, return to step 3.

Example



All-or-nothing assignment

An **all-or-nothing assignment** is a feasible path flow vector $\mathbf{h}^*$ which has positive flow only for paths with minimum travel time between their OD pair.

The difference between an all-or-nothing assignment and an equilibrium is that the path travel times $\mathbf{c}$ *do not* need to correspond to the path flows $\mathbf{h}^*$. (Think about this in the iterative framework: the all-or-nothing assignment corresponds to the path travel times from the *current* path flows.)

You can think of this as a "target" path flow vector indicating how people would choose paths if the travel times were fixed at their current value.

If there is a tie for an OD pair, you can assign vehicles to any or all of the shortest paths arbitrarily.

The all-or-nothing assignment can also be written in terms of the corresponding link flows $\mathbf{x}^*$. (This saves memory, in large networks there are many more paths than links.)

Given $\mathbf{h}^*$, how can we find $\mathbf{x}*$?

Slow way: directly calculate the sum $x_{ij}^* = \sum_{r \in Z} \sum_{s \in Z} \sum_{\pi \in \Pi^{rs}} \delta_{ij}^{\pi} h_{\pi}^*$

Medium way: don't sum over all paths, just the one path in h^* for each OD pair.

Fast way: use backnodes to avoid having to "sum" over δ_{ij}^{π} terms which are zero.

Really fast way: ????? (see Exercise 4.17; think Bellman's principle and acyclic subnetworks)