Shortest paths: label setting

CE 377K

February 19, 2015

REVIEW

HW 2 posted, due in 2 weeks

Basic search algorithm

Prim's algorithm

Algorithm

(Assumes that the network is connected.)

1. Arbitrarily choose some root note s.
2. Initialize $N_T \leftarrow \{s\}$, $A_T \leftarrow \emptyset$
3. Identify all of the admissible links; if there are none, terminate.
4. Choose an admissible link (u, v) with minimum cost. (Assume u is in N_T, but not v.)
5. Add this link to the tree: $N_T \leftarrow N_T \cup \{v\}$, $A_T \leftarrow A_T \cup (u, v)$
6. Return to step 3.

Complexity

There are $O(n)$ iterations (technically $n-1$).

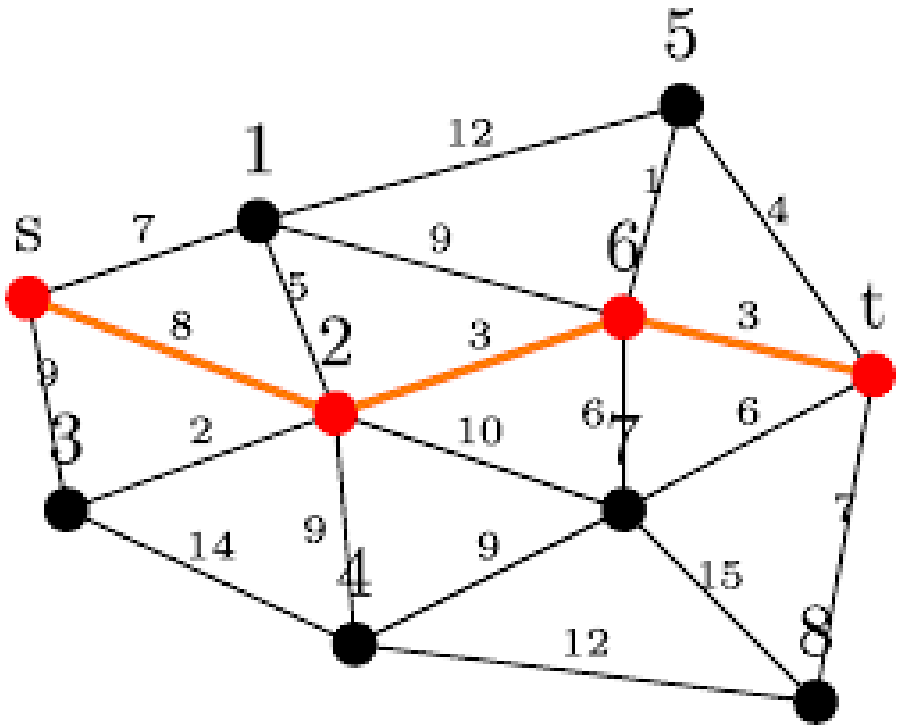
At each iteration, we must identify all admissible links (of which there are at most m), and identify one with minimum cost (which again takes m steps).

So, Prim's algorithm is $O(nm)$.

There are more clever ways of identifying admissible links and finding one with minimum cost, which can reduce the running time to $O(m \log n)$ or $O(m+n \log n)$. These do so by avoiding "duplication of effort" in subsequent iterations.

SHORTEST PATH PROBLEM

Shortest Path Problem



Identify a path connecting a given origin and destination, where the total cost of the links in the path is minimized.

Applications

- Vehicle routing
- Critical path analysis in project management
- Six degrees of Kevin Bacon

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.

Unlike the minimum spanning tree problem, the direction of links is important in the shortest path problem.

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.

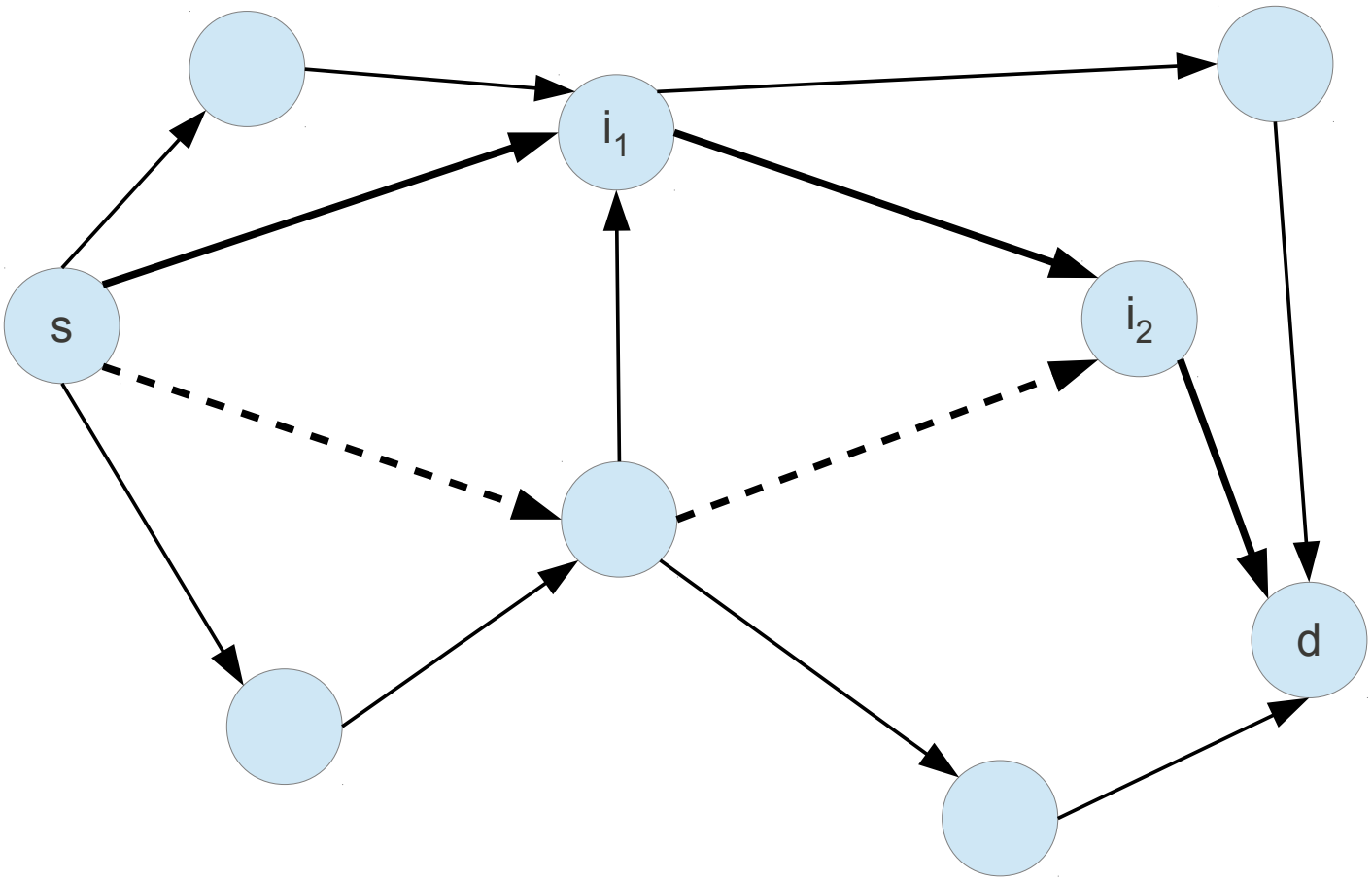
For the purposes of this course, either will work. For clarity, we'll stick with one-to-all shortest paths.

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 *path 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

$$\min_{\mathbf{x}} \quad \sum_{(i,j)\in A} c_{ij} x_{ij}$$

$$\text{s.t.} \quad \sum_{(i,j)\in A(i)} x_{ij} - \sum_{(h,i)\in B(i)} x_{ij} = \begin{cases} 1 & \text{if } i = r \\ -1 & \text{if } i = s \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in \{1, \ldots, I\}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i,j) \in A$$

DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is a **label-setting** shortest path algorithm.

That is, once we scan a node, its labels are set permanently and never changed again.

Notation

Dijkstra's algorithm maintains a set of *finalized* nodes F, which we have already found the shortest path to.

An arc is *admissible* (or *eligible*) if its tail node is in F, but not its head node; E is the set of admissible arcs.

Dijkstra's Algorithm

(Assume there is at least one path from r to all other nodes in the network, and that $c_{ij} \geq 0$ for all links.)

1. Initialize all labels $L_i \leftarrow \infty$, except for the origin $L_r \leftarrow 0$
2. Initialize $F \leftarrow \{r\}$, and the path vector $\mathbf{q} \leftarrow -\mathbf{1}$
3. Find the set of admissible arcs E.
4. For each admissible arc, calculate a temporary label $L_{ij}^{temp} = L_i + c_{ij}$
5. Find the arc (i^*, j^*) for which L_{ij}^{temp} is minimal.
6. Set $L_j \leftarrow L_{ij}^{temp}$, add j^* to F, and set $q_{j^*} = i^*$
7. If $F = N$, terminate. Otherwise, return to step 3.

Example

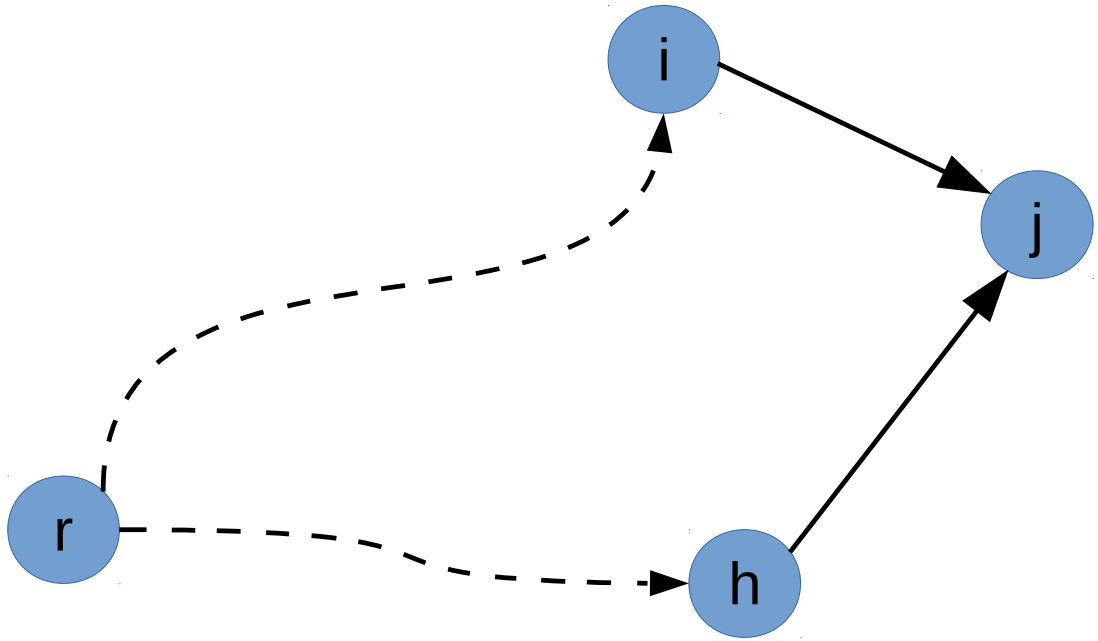
Correctness

Each iteration adds one more node to F. Eventually it must include all nodes in N.

When it terminates, do the path labels represent shortest paths from r?

By contradiction, assume that this is not the case.

What if there were a shorter path through another node (say h)? Consider what happened when Dijkstra's algorithm chose (i, j).



If h had already been finalized, then $L_h + c_{hj} = L_{hj}^{temp}$ would have been less than $L_i + c_{ij} = L_{ij}^{temp}$.

If h had not yet been finalized, then $L_h > L_{ij}^{temp}$ since Dijkstra's algorithm finalizes nodes in order of their L value.

Since $c_{hj} \geq 0$, if $L_h > L_{ij}^{temp}$ then $L_{hj}^{temp} > L_{ij}^{temp}$, again a contradiction.

Complexity

There are $O(n)$ iterations (one for each node except the origin).

At each iteration, we must perform $O(m)$ work: finding the set of admissible arcs, calculating temporary labels, and finding the arc with minimal temporary label.

So, this implementation of Dijkstra's algorithm requires $O(nm)$ steps.

It is not hard to do better and reduce the complexity to $O(n^2)$.

Fancier Dijkstra

(Assume there is at least one path from r to all other nodes in the network, and that $c_{ij} \geq 0$ for all links.)

1. Initialize all labels $L_i \leftarrow \infty$, except for the origin $L_r \leftarrow 0$
2. Initialize $F \leftarrow \emptyset$, and the path vector $\mathbf{q} \leftarrow -\mathbf{1}$
3. Find the node i *not* in F with minimum L_r value.
4. For each arc $(i, j) \in A(i)$, repeat these steps:
5. Calculate $L_{ij}^{temp} \leftarrow L_i + c_{ij}$
6. If $L_{ij}^{temp} < L_j$, then update $L_j \leftarrow L_{ij}^{temp}$ and set $q_j = i$.
7. Add i to F.
8. If $F = N$, terminate. Otherwise, return to step 3.

This implementation of Dijkstra's algorithm only requires $O(n^2)$ steps. Why?

The bottleneck is finding the node with minimum L_r value.

There are even more efficient versions of Dijkstra's, targeted at this step, which can reduce the running time to $O(n \log n)$ steps.