Shortest Paths (15)

1 MST WRAP UP

1.1 Review

Given an undirected weighted graph, find a set of edges so that all nodes are connected (spanning tree) and the total edge weight is minimized.
Kruskal's algorithm works as follows:

- Sort the list of edges.
- Mark each node with a different color.
- Set up a mapping from color to a list of nodes.
- Initialize the count of nodes in each color to 1.
- Repeat $V - 1$ times:
  - Take the smallest weight edge, discard if both endpoints have the same color.
  - If not, figure out which color has the smaller number of nodes, and recolor them to the color of the other list.

1.2 Correctness Analysis

We showed that the "greedy grow lemma" tells us that we can always add the smallest weight edge that *doesn't* make a cycle and we'll end up with an MST.

1.3 Running Time Analysis

Informal potential function analysis:

- All the work is in relabeling nodes. How many times can a node u be relabeled? In particular, if u is in a set of size k, how many times could it have been relabeled?

  Each time it is relabeled, it joins a set at least twice as big as its old set. Thus, if u is in a set of size k, it can only have been relabeled $\log_2 k$ times!

- The final set size is $|V|$, so how many times can a node be relabeled?

  $\log_2(|V|)$.

- Thus, $\Theta(|V| \log |V|)$ time for all calls to JOIN, since all $|V|$ nodes end up in sets of size $|V|$. This is dominated by the time to sort, so no further improvements to JOIN and CONNECTED will help!

Total: SORT plus JOIN is $O(|E| \log |E|) = O(|E| \log |V|)$.

2 SHORTEST PATH PROBLEM

2.1 Route Finding

A number of companies on the web make their money by advertising on cites that produce driving directions for any two points in the US.
Simple version of the problem: map is a weighted directed graph $G = (V, E)$, w.
Nodes are places and intersections, edges are roads, weights are driving times (factoring in distance, road size, expected traffic).

2.2 Single-Source Shortest-Path Problem

Given a source node s and a destination t, we want to find the shortest (minimum weight) path from s to t.
En route, we will find the shortest path to all nodes $u \in V$.
Solution is a sequence of nodes $v_1, \ldots, v_l$ such that $v_1 = s$, $v_l = t$, and $\sum_{i=1}^{l-1} w((v_i, v_{i+1}))$ is minimized.
Definition: $\delta(u, v)$ is the length of the shortest path from u to v. So, we're looking for a path from s to t whose length is $\delta(s, t)$.
Example graph...

2.3 Variations

Some simple variations:

- How find *any* path from s to t?

  Depth-first search ought to do it!

- How find shortest path if $w(u) = 1$ for all $u \in V$?

  Breadth-first search... try short paths before long ones.

- How find shortest path if G is acyclic?

  Like the "makespan" algorithm on the homework.

More complex variations that we'd look at if we had time:

- All-pairs shortest path: How can you compute shortest paths for all pairs s, t in less time than it takes for $|V|$ single-source shortest-path runs?
- Stochastic shortest path: What if there is a probability that you'll leave u headed for v but end up at r instead?
- Negative edge weights: What if traversing some edges actually *improves* your driving time?

2.4 Some Properties of Shortest Paths

Here's an example s to t shortest path in the example graph...

- What can we say about the length of any shortest path to a node u on the path from s to t ($\delta(s, u)$)?

  All subpaths along the shortest path are themselves shortest paths.

- What can we say about the length of any shortest path to nodes v *not* on the path from s to t ($\delta(s, v)$)?

  Length of path from s to v plus the path from v to t is no shorter than the shortest path from s to t: $\delta(s, v) + \delta(v, t) \geq \delta(s, t)$. This is also known as the triangle inequality.

- What can we say about the sequence $\delta(s, v_i)$ along a shortest path $v_1, \ldots, v_l$?

  Increasing.

2.5 Optimal Substructure Property

Because a shortest path is made up of other shortest paths (how many on a path of length n?), we say that the shortest-path problem exhibits the *optimal substructure property*.

$n(n - 1)/2$.

How would you prove it?

If there is a shortcut, it would reduce the overall path length.

Although it might seem backwardsly useful, we actually depend on this property quite a bit when finding shortest paths.

2.6 Shortest Path Tree

The optimal substructure property can be used to prove the following interesting property of the solution to single-source shortest-path problems.
We can always arrange the solution in a tree, so the shortest path to v involves following a shortest path to u and then taking the edge (u, v). The set of such edges forms a *shortest path tree*.
Example...
Proof?

> Like the MST proof, we can take any set of shortest paths and make them into a tree without increasing the overall length of any path. Induction on path length.

3 DIJKSTRA'S ALGORITHM

3.1 Idea

We will list out all the nodes of G in order of their distance from s. This will insure that we won't miss any short cuts as we go.

- Consider shortest edge out of s (to v). No other path to v can be shorter. Why? Moreover, v is a closest node to s. Why?

  > Edge weights are non-negative, so it would have to pass through a one-step away node and then additional weight will be added to get back to v. Going to v directly is no longer.

- Therefore, we can include (s, v) in the shortest-path tree.
- Mark v with its distance from s, $w((s, v))$, and now look for the next closest node. Which will it be?
- Basically, keep a set of "inside" and "outside" nodes and move the outside nodes inside one at a time in order of their distance.

3.2 Bookkeeping

How can we keep track of which nodes are the closest?

- Keep an array d to hold distances: $d[v]$ is the best known distance to node v.
- When a node is added to the tree, see if any new shortcuts have been discovered, thus shortening the best known distance to some node v.
- What data structure ought to hold this kind of information?

3.3 Initialization and Improvement

```
Initialize-Single-Source(G, s)
for each v ∈ V[G]
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

```
Relax(u, v, w)
if d[u] + w(u, v) < d[v]
   then d[v] ← d[u] + w(u, v)
        π[v] ← u
```

3.4 Algorithm

```
Dijkstra(G, w, s)
Initialize-Single-Source(G, s)
S ← ∅
Q ← V[G]
while Q ≠ ∅
    do u ← Extract-Min(Q)
       S ← S ∪ {u}
       for each v ∈ Adj[u]
           do Relax(u, v, w)
```

Example run...

3.5 Connection to Prim's

Nearly identical to Prim's MST algorithm:

```
Relax-Prim(u, v, w)
if d[v] > w(u, v)
   then d[v] ← w(u, v)
        π[v] ← u
```

In Dijkstra, distance to "outside" node is the total distance from source instead of just the minimum length edge from the current tree.

3.6 Correctness

Some facts:

- Algorithm maintains the invariant that for all "outside" nodes v, $d[v]$ is the length of the shortest path from s to v using only inside nodes. Invariant maintained by looking at all possible extensions of "inside" paths.

- The distances of nodes along a shortest path from s to t is monotonically non-decreasing, so it is reasonable to grow the path in increasing order of distance.

- Let u_i be the ith node brought to the inside. The sequence $d[u_i]$ is non decreasing: Dijkstra's sorts the nodes by distance.

- For any given node u, $d[u]$ starts at infinity and decreases until u is brought inside.

- Induction: When u is brought inside, $d[u] = \delta(s, u)$.

3.7 Running Time

Just like Prim's:

- Priority queue contains $|V|$ entries, so queue operations take $O(\log |V|)$.

- Call RELAX at most once per edge, each might require an adjustment to the priority queue: $O(|E| \log |V|)$.

- Call EXTRACT-MIN once per vertex: $O(|V| \log |V|)$.

- Total: $O(|E| \log |V|)$, assuming all edges reachable from source.

- Can also implement the priority queue with a simple array and get $O(|V|^2)$ (better for dense graphs).

4 A*

4.1 Obvious Waste

In a general, asymptotic, worst-case setting, there are no algorithms that are known to find a shortest (s, t) path any faster than computing the entire shortest-path tree.
Nonetheless, there are obvious inefficiencies in this approach.
Consider route finding in 2d in which t is on one side of s, and all other nodes are closer and in the other direction. Dijkstra's "looks where the light is good" in some sense.
Note: This is not a typical "algorithms" topic, but it's pretty cool, pretty useful, and an AI thing.

4.2 Admissible Heuristic

If path lengths can be easily lower bounded, we can use this information to guide the search. An *admissible* heuristic h is a function that gives a guaranteed lower bound on the distance from any node u to the destination t. Natural example: straight-line distance (at maximum speed) to t.

4.3 Important Properties

For the nice formal properties to hold, we need h to satisfy a few properties:

- Admissibility: For any node u, $0 \leq h(u) \leq \delta(u, t)$.
- Monotonicity (triangle inequality): For any pair of nodes u and v, $h(u) \leq h(v) + w((u, v))$.
- Destination: $h(t) = 0$ (follows from admissibility).

Here are some useful definitions.

- $f^*(u) = \delta(s, u) + \delta(u, t)$: shortest diverted path.
- $f(u) = h(u) + \delta(u, t)$: estimated distance.

4.4 Observations

Useful observations:

- $f^*(u)$ is minimized for u on a shortest (s, t) path, with $f^*(u) = \delta(s, t)$.
- For all u, $f(u) \leq f^*(u)$ because of admissibility.
- $f(t) = \delta(s, t)$.
- Consider two consecutive nodes u and v along a shortest path (to anywhere). Claim: $f(u) \leq f(v)$. Proof: By monotonicity, $h(u) \leq h(v) + w((u, v))$. Because u and v are on a shortest path, $w((u, v)) = \delta(s, v) - \delta(s, u)$. Combining, we get $\delta(s, u) + h(u) \leq h(v) + \delta(s, v)$ or $f(u) \leq f(v)$.

4.5 New Algorithm

Change EXTRACT-MIN in Dijkstra's to pick out the node u with the smallest value of $d[u] + h(u)$. [Note, at the smallest value of $d[u] + h(u)$, $f(u) = \delta(s, u) + h(u) = d[u] + h(u)$ because of monotonicity.]
Idea:

- The quantity $f(u)$ is an underestimate of the total distance from s to t through u (true distance from s to u plus an underestimate of the distance from u to t).
- By focussing the search on the nodes with the smallest value of $f(u)$, we are likely to be doing work along the true shortest path.

This is called A*.

4.6 Basic Concepts

Search proceeds in rings of increasing f cost. Note that if h is a perfect heuristic, the source and destination have the same f cost.

- A* brings inside all nodes with $f(v) \leq f(t) = \delta(s,t)$.
- A* may bring inside some nodes with $f(v) = \delta(s,t)$ before bringing inside t and terminating.

Correctness: Solution found is true shortest path because all subsequent contours have higher f cost, and therefore, higher distance.

4.7 Running Time and Heuristic Quality

Higher quality heuristic implies no more work.

- We say a heuristic h' is *higher quality* than another h if $h'(v) \leq h(v)$ for all v (and admissibility and monotonicty hold).
- Therefore, the f values under h' will be smaller than those under h.
- Recall that A* brings inside all nodes with f values below that of the destination.
- Therefore, running with h' will bring inside no more nodes than running with h.

4.8 Running Time

The lowest quality heuristic is $h(v) = 0$. What algorithm is this?

> Dijkstra's.

Therefore, we get at least $O(|V|^2)$ or $O(|E| \log |V|)$, maybe better.

4.9 Generating Heuristics

Some useful ways to think about good heuristics:

- If h_1 and h_2 are admissible heuristics, then $h(v) = \max(h_1(v), h_2(v))$ is admissible (and of no lesser quality).
- "Relaxations" often produce good heuristics: if cost is a function of some constraints on nodes, then constraint-free distance is admissible (Manhattan distance in 8 puzzle).

4.10 Varying Quality

What happens as we vary heuristic quality?

- As a thought experiment, let's consider a grid of points and s and t on the same line.
- Let all nodes be completely connected and w be Euclidean distance.
- Let $h_c(v) = c\delta(v, t)$, so $c = 0$ is Dijkstra and $c = 1$ is the perfect heuristic.
- Which nodes are searched as a function of c? Try 0, 1, 1/2.