CS61B Lecture 34

Friday, April 17, 2020

Shortest Paths

Dijkstra's algorithm gives you shortest paths from a particular given vertex to all others in a graph. However, what if we were only interested in getting to a particular vertex?

Well, one idea would be to use Dijkstra's algorithm anyway, because the algorithm finds paths in order of length, and stop it when we get to the vertex we want. This can, however, be very wasteful.

An example would be finding the shortest paths from Denver to Fifth Avenue in NYC, which is about 1750 miles by road. However, Dijkstra's algorithm would explore everything within a 1749 mile radius, including Berkeley, even though it is in the total wrong direction. This gets worse when the graph is generated on the fly, making it infinite!

A* Search

Looking for a path from Denver to NYC, suppose we had a heuristic guess,

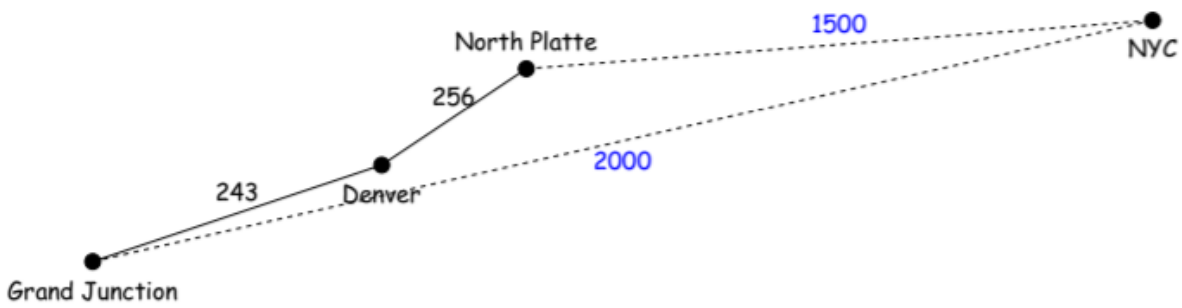
$$h(V)$$

of the length of a path from any vertex `v` to NYC. And suppose that instead of visiting vertices in the fringe in order of their shortest known path to Denver, we order by the sum of that distance plus a **heuristic estimate** of the remaining distance to NYC:

$$d(Denver, V) + h(V)$$

In other words, we look at places that are reachable from places where we already know the shortest path to Denver and choose those that look like they will result in the shortest trip to NYC, guessing at the remaining distance.

If the estimate is good, we will not look at, say, Grand Junction, Colorado (250 miles west by road) because it's in the wrong direction. This algorithm is **A* search**, but for it to work, we must be careful about the heuristic.



If the solid lines indicate distances from Denver we have determined so far, Dijkstra's algorithm would have us look at Grand Junction next. But if we add the heuristic remaining distance to NYC (our goal), we choose North Platte instead.

Admissible Heuristics for A* Search

If our heuristic estimate for the distance to NYC is too high, then we may get to NYC without ever examining points along the shortest route.

For example, if our heuristic decided the midwest was literally in the middle of nowhere, and

$$h(C) = 2000$$

for `c` any city in Michigan or Indiana, we'd only find a path detouring south through Kentucky. To be admissible, the heuristic must never overestimate

$$d(C, NYC)$$

the minimum path distance from `c` to NYC. On the other hand,

$$h(C) = 0$$

will work, but yield a non-optimal algorithm.

Consistency

Suppose we estimate

$$h(Chicago) = 700$$
$$h(Springfield) = 200$$
$$d(Chicago, Springfield) = 200$$

By driving 200 miles to Springfield, we would guess we are suddenly 500 miles closer to NYC. This passes our definition of an admissible heuristic, since both estimates are low, but it will mess up our algorithm.

Specifically, it will require that we put processed nodes back into the fringe, in case our estimate was wrong.

So, for the purposes of 61B, we will also require **consistent heuristics**:

$$h(A) \leq h(B) + d(A, B)$$

as per the triangle inequality.

All consistent heuristics, if you really think about it, are admissible.

A demo of A* search is in `cs61b-software` and on the instructional machines as `graph-demo`.

Summary of Shortest Paths

Dijkstra's algorithm finds a shortest-path tree computing giving (backwards) shortest paths in a weighted graph from a given start node to all other nodes. It takes:

- time to remove `v` nodes from the priority queue.
- time to update all neighbors of each of these nodes and add or reorder them in the queue. (E lg E)
- $$\in \Theta(VlgV + ElgV) = \Theta((V + E)lgV)$$

A* search searches for a shortest-path from a node to a particular target node. It is mostly the same as Dijkstra's algorithm, except:

- we stop when we take the target from queue.
- we order the queue by estimated distance to start plus some heuristic guess of the remaining distance.

$$h(v) = d(v, target)$$

- we must make sure said heuristic does not overestimate distance and is consistent by obeying triangle inequality.

Minimum Spanning Trees

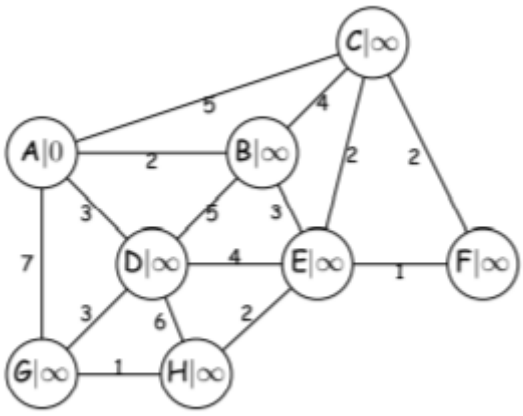
Let's imagine you own a multimillion dollar company with multiple offices. You'd like to build a physical wired network connecting all of the offices using copper wires. We'd like to connect these offices while minimizing the length of copper wire used. This isn't shortest paths, but we will instead end up with a tree.

It's easy to see that such a set of connecting roads and places must form a tree, because removing one road in a cycle stil allows all to be reached.

Prim's Algorithm

MSTs can be done by Prim's Algorithm. The idea is to grow a tree starting from an arbitrary node, and at each step, add the shortest edge connecting some node already in the tree to one that isn't yet. Because of the tree property, this must work:

```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
    Vertex v = fringe.removeFirst();
    For each edge(v, w) {
        if (w in fringe && weight(v, w) < w.dist())
            { w.dist() = weight(v, w); w.parent() = v; }
        }
}
```



This is going to iterate once for every vertex, then every edge connected to it. The time complexity is the product of the number of edges (every vertex should theoretically be connected to an edge) by the time required to check.

Kruskal's Algorithm

There is another approach to doing MSTs, which is Kruskal's algorithm.

We observe that the shortest edge in a graph can always be a part of a minimum spanning tree. In fact, we have a bunch of subtrees of an MST, then the shortest edge that connects two of them can be part of an MST, combining two subtrees into a bigger one.

So,

```
/* Create one (trivial) subtree for each node in the graph. */

MST = {};
for (edge(v, w) in increasing order of weight) {
    if ( v.connects two different subtrees ) {
        MST.add(v);
        (v, w).subtree1.combine((v, w).subtree2);
    }
}
```

Since the edge did not previous connect a node in the subtree to itself, this does not introduce any cycles and preserves the tree property.

Union Find

Kruskal's algorithm required that we have a set of sets of nodes (yes, a set of sets) with two operations:

- **find** which of the sets a given node belongs to
- replace two sets with their **union**, reassigning all nodes in the two original sets to this union.

The obvious thing to do is to store a set number in each node, making finds fast. Union requires changing the set number in one of the two sets being merged; the smaller one is the better choice.

This means an individual union can take

$$\Theta(N)$$

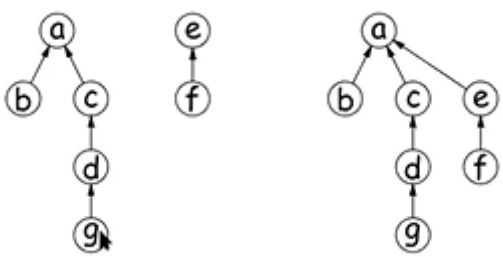
time.

The whole thing is called the union find algorithm.

One clever trick we can use is to represent a set of nodes by one arbitrary representative node in that set, and let every node contain a pointer to another node in the same set. Arrange for each pointer to represent the parent of a node in a tree that has the representative node as its root.

To find what set a node is in, follow parent pointers.

To union two such trees, make one root point to the other, choosing the root of the larger tree as the union representative.



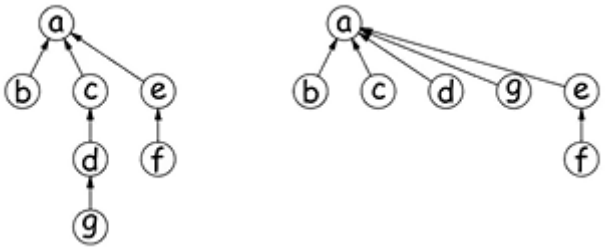
So now, union operations now take whatever the find operation takes, plus the actual process of unionizing, which is a constant time operation.

Our only concern is that the a-c-d-g chain above is quite long, and we could perhaps do something about it. We can, and it's called path compression, which makes unionizing very fast. This is to avoid the long chains that can slow down the find operation to

$$\Omega(lgN)$$

Using the following trick can help mitigate this: whenever we do a find operation, compress the path to the root, so that subsequent finds will be faster.

We compress the path by making each of the nodes in the path point directly to the root, making union very fast, and sequences of unions and finds each have very, very nearly constant, amortized time. For example, finding `g` in the last tree results in the following:



The time bounds on this follow something called an Ackermann function, specifically its inverse, but that is outside the scope of this class (mostly).