union-find.

from a particular given vertex to all others in a graph.

- But suppose you're only interested in getting to a particular vertex?
- Because the algorithm finds paths in order of length, you *could* simply run it and stop when you get to the vertex you want.
- But, this can be really wasteful.
- For example, to travel by road from Denver to a destination on lower Fifth Avenue in New York City is about 1750 miles (says Google).
- But traveling from Denver to the Gourmet Ghetto in Berkeley is about 1650 miles.
- So, we'd explore much of California, Nevada, Arizona, etc. before we found our destination, even though these are all in the wrong

ver to the desired NYC vertex.

- Suppose that 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: $d(\text{Denver}, V) + h(V)$.
- 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, then we don't look at, say, Grand Junction (250 miles west by

the heuristic.

to NYC is too high (i.e., larger than the actual path by road), then we may get to NYC without ever examining points along the shortest route.

- For example, if our heuristic decided that the midwest was literally the middle of nowhere, and $h(C) = 2000$ for C any city in Michigan or Indiana, we'd only find a path that detoured south through Kentucky.
- So to be *admissible,* $h(C)$ must never overestimate $d(C, \text{NYC})$, the minimum path distance from C to NYC.
- On the other hand, $h(C) = 0$ will work (what is the result?), but yield a non-optimal algorithm.

and h(Springfield, IL) $= 200$, where d(Chicago, Springfield) $=$ 200.

- So by driving 200 miles to Springfield, we guess that we are suddenly 500 miles closer to NYC.
- This is admissible, since both estimates are low, but it will mess up our algorithm.
- Specifically, will require that we put processed nodes back into the fringe, in case our estimate was wrong.
- So (in this course, anyway) we also require *consistent heuristics:* $h(A) \leq h(B) + d(A, B)$, as for the triangle inequality.
- All consistent heuristics are admissible (why?).
- For project 3, distance "as the crow flies" is a good $h(\cdot)$ in the trip application.

Tree computing giving (backwards) shortest paths in a weighted graph from a given starting node to all other nodes.

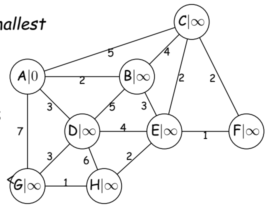
- Time required =
  - Time to remove V nodes from priority queue +
  - Time to update all neighbors of each of these nodes and add or reorder them in queue ($E \lg E$)
  - $\in \Theta(V \lg V + E \lg V) = \Theta((V + E) \lg V)$
- A* search searches for a shortest path to a *particular* target node.
- Same as Dijkstra's algorithm, except:
  - Stop when we take target from queue.
  - Order queue by estimated distance to start + heuristic guess of remaining distance ($h(v) = d(v, \text{target})$)

between them (assume always positive), find a set of connecting roads of minimum total length that allows travel between any two.

- The routes you get will not necessarily be shortest paths.
- Easy to see that such a set of connecting roads and places must form a tree, because removing one road in a cycle still allows all to be reached.

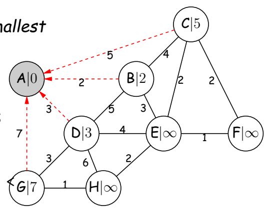
- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

```
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 ∈ fringe && weight(v,w) <
w.dist())
      { w.dist() = weight(v, w);
w.parent() = v; }
  }
}
```



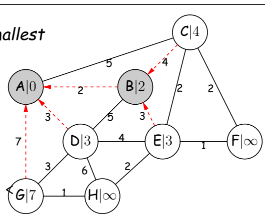
- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?


```
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 ∈ fringe && weight(v,w) <
w.dist())
      { w.dist() = weight(v, w);
w.parent() = v; }
  }
}
```



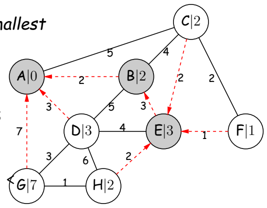
- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?


```
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 ∈ fringe && weight(v,w) <
w.dist())
      { w.dist() = weight(v, w);
w.parent() = v; }
  }
}
```



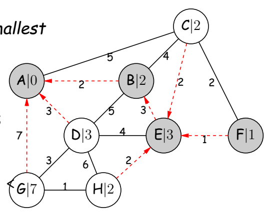
- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

```
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 ∈ fringe && weight(v,w) <
w.dist())
      { w.dist() = weight(v, w);
w.parent() = v; }
  }
}
```



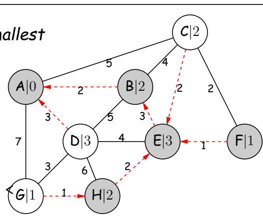
- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?


```
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 ∈ fringe && weight(v,w) <
w.dist())
      { w.dist() = weight(v, w);
w.parent() = v; }
  }
}
```



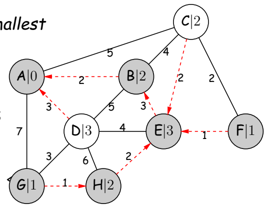
- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?


```
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 ∈ fringe && weight(v,w) <
w.dist())
      { w.dist() = weight(v, w);
w.parent() = v; }
  }
}
```



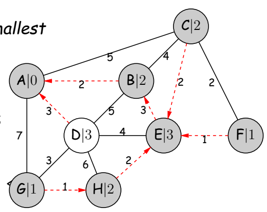
- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

```
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 ∈ fringe && weight(v,w) <
w.dist())
      { w.dist() = weight(v, w);
w.parent() = v; }
  }
}
```



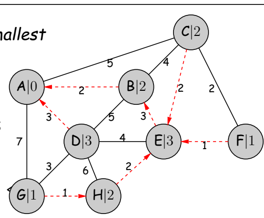
- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?


```
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 ∈ fringe && weight(v,w) <
w.dist())
      { w.dist() = weight(v, w);
w.parent() = v; }
  }
}
```



- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?


```
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 ∈ fringe && weight(v,w) <
w.dist())
      { w.dist() = weight(v, w);
w.parent() = v; }
  }
}
```



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

```
   Create one (trivial) subtree for each node in the
graph;
   MST = {};

   for each edge(v,w), in increasing order of weight
{
      if ( (v,w) connects two different subtrees
) {
         Add (v,w) to MST;
         Combine the two subtrees into one;
      }
```

set of sets of nodes with two operations:

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

- 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 is better choice.
- This means an individual union can take $\Theta(N)$ time.
- Can union be fast?

one arbitrary representative node in that set.

- 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 (choose the root of the larger tree as the union representative).

operation potentially slow ($\Omega(\lg N)$).

- So use the following trick: whenever we do a *find* operation, *compress* the path to the root, so that subsequent finds will be faster.
- That is, make each of the nodes in the path point directly to the root.
- Now union is very fast, and sequence of unions and finds each have very, *very* nearly constant amortized time.
- Example: find 'g' in last tree (result of compression on right):