CS61B Lecture #34

- **Today:** A* search, Minimum spanning trees, union-find.

Point-to-Point Shortest Path

- Dijkstra's algorithm gives you shortest paths from a particul vertex to all others in a graph.
- But suppose you're only interested in getting to a particular
- Because the algorithm finds paths in order of length, you *co* ply 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 Fifth Avenue in New York City is about 1750 miles (says Goo
- But traveling from Denver to the Gourmet Ghetto in Ber about 1650 miles.
- So, we'd explore much of California, Nevada, Arizona, etc. we found our destination, even though these are all in th direction!
- Situation even worse when graph is infinite, generated on th

A* Search

- We're looking for a path from vertex Denver to the desir vertex.
- Suppose that we had a *heuristic guess,* $h(V)$, of the length o from any vertex V to NYC.
- And suppose that instead of visiting vertices in the fringe of their shortest known path to Denver, we order by the that distance plus a *heuristic estimate* of the remaining dis NYC: $d(\text{Denver}, V) + h(V)$.
- In other words, we look at places that are reachable fron where we already know the shortest path to Denver and those that look like they will result in the shortest trip guessing at the remaining distance.
- If the estimate is good, then we don't look at, say, Grand J (250 miles west by road), because it's in the wrong directior
- The resulting algorithm is A* search.
- But for it to work, we must be careful about the heuristic.

Admissible Heuristics for A* Search

- If our heuristic estimate for the distance to NYC is too h larger than the actual path by road), then we may get to NYC ever examining points along the shortest route.
- For example, if our heuristic decided that the midwest was the middle of nowhere, and $h(C) = 2000$ for C any city in Mic Indiana, we'd only find a path that detoured south through Ke
- So to be *admissible,* $h(C)$ must never overestimate $d(C, \text{NY}$ minimum path distance from C to NYC.
- On the other hand, $h(C) = 0$ will work (what is the result?), b a non-optimal algorithm.

Consistency

- Suppose that we estimate $h(\text{Chicago}) = 700$, and $h(\text{Springfiel}$
200, where $d(\text{Chicago, Springfield}) = 200$.
- So by driving 200 miles to Springfield, we guess that we c
denly 500 miles closer to NYC.
- This is admissible, since both estimates are low, but it will
our algorithm.
- Specifically, will require that we put processed nodes back
fringe, in case our estimate was wrong.
- So (in this course, anyway) we also require *consistent he*
$h(A) \leq h(B) + d(A, B)$, 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
application.
- Demo of A* search (and others) is in `cs61b-software` anc
instructional machines as `graph-demo`.

Summary of Shortest Paths

- Dijkstra's algorithm finds a *shortest-path tree* computin (backwards) shortest paths in a weighted graph from a give ing 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 anc 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* targ
- Same as Dijkstra's algorithm, except:
  - Stop when we take target from queue.
  - Order queue by estimated distance to start + heuristic g remaining distance ($h(v) = d(v, \text{target})$)
  - Heuristic must not overestimate distance and obey tria equality ($d(a, b) + d(b, c) \geq d(a, c)$).

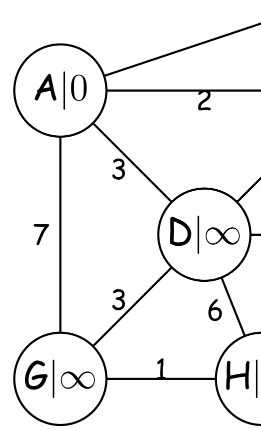
Minimum Spanning Trees

- **Problem:** Given a set of places and distances between them always positive), find a set of connecting roads of minimu 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 plac form a tree, because removing one road in a cycle still allow be reached.

Minimum Spanning Trees by Prim's Algorithm

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

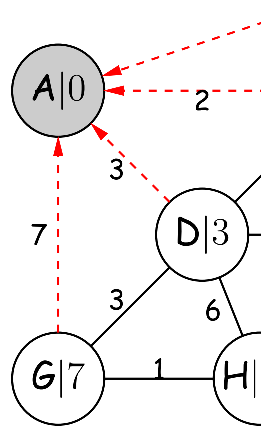
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

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



Minimum Spanning Trees by Prim's Algorithm

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

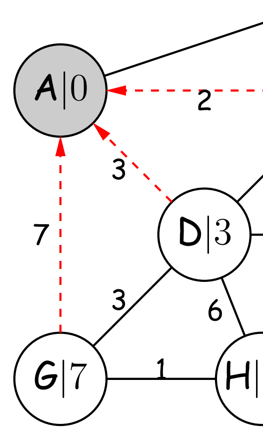
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

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



Minimum Spanning Trees by Prim's Algorithm

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

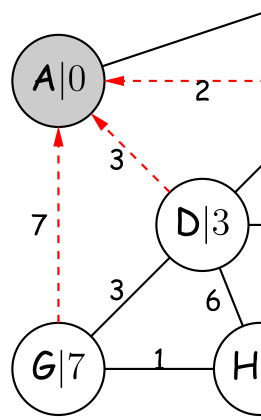
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

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



Minimum Spanning Trees by Prim's Algorithm

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

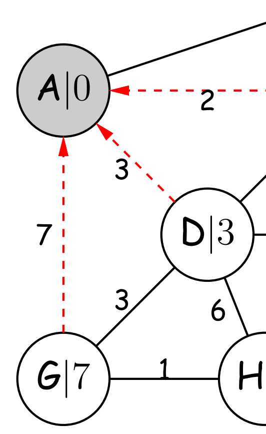
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

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



Minimum Spanning Trees by Prim's Algorithm

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

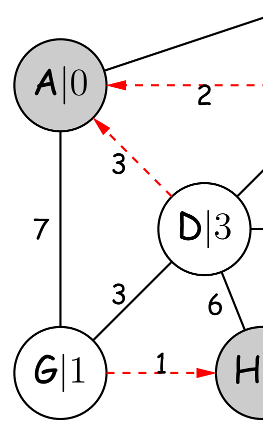
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

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



Minimum Spanning Trees by Prim's Algorithm

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

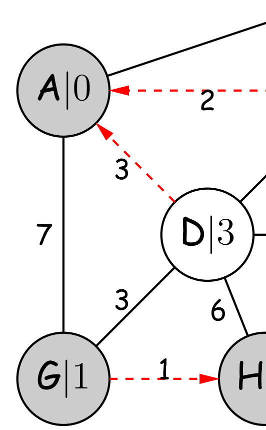
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

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



Minimum Spanning Trees by Prim's Algorithm

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

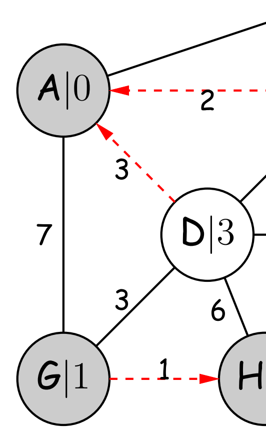
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

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



Minimum Spanning Trees by Prim's Algorithm

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

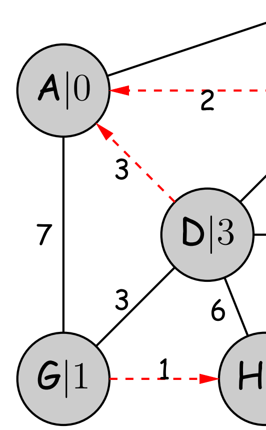
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

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



Minimum Spanning Trees by Prim's Algorithm

- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node in the tree to one that isn't yet.
- Why must this 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 ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```



Minimum Spanning Trees by Kruskal's Algorith

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

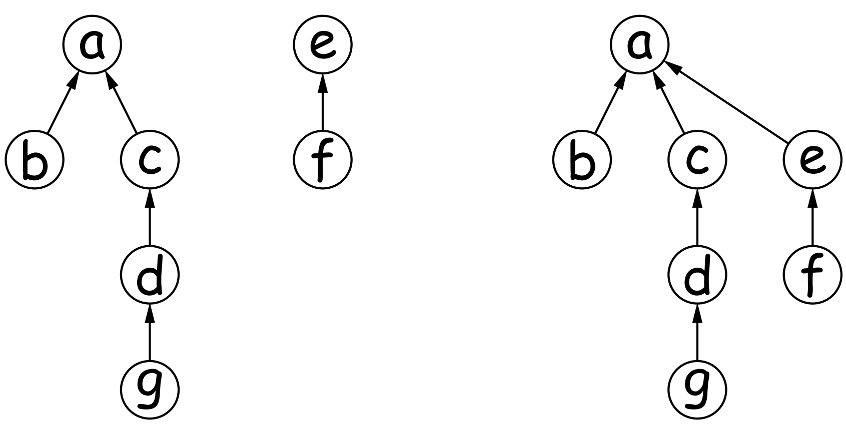
```
Create one (trivial) subtree for each node in the graph;
MST = {};

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;
   }
}
```

Union Find

- Kruskal's algorithm required that we have a set of sets of noc
  two operations:
  - *Find* which of the sets a given node belongs to.
  - Replace two sets with their *union,* reassigning all the node
    two original sets to this union.
- Obvious thing to do is to store a set number in each node,
  finds fast.
- Union requires changing the set number in one of the two se
  merged; the smaller is better choice.
- This means an individual union can take $\Theta(N)$ time.
- Can union be fast?

A Clever Trick

- Let's choose to represent a set of nodes by *one* arbitrary re
  tative node in that set.
- Let every node contain a pointer to another node in the sam
- Arrange for each pointer to represent the *parent* of a node i
  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
  the root of the larger tree as the union representative).



Path Compression

- This makes unioning really fast, but the find operation pot slow ($\Omega(\lg N)$).
- So use the following trick: whenever we do a *find* operatio *press* the path to the root, so that subsequent finds will be
- That is, make each of the nodes in the path point directly root.
- Now union is very fast, and sequence of unions and finds ea very, *very* nearly constant amortized time.
- Example: find 'g' in last tree (result of compression on right