CS61B Lecture #39

- **Today**: Minimum spanning trees, recursive graph algorithms, union-find.
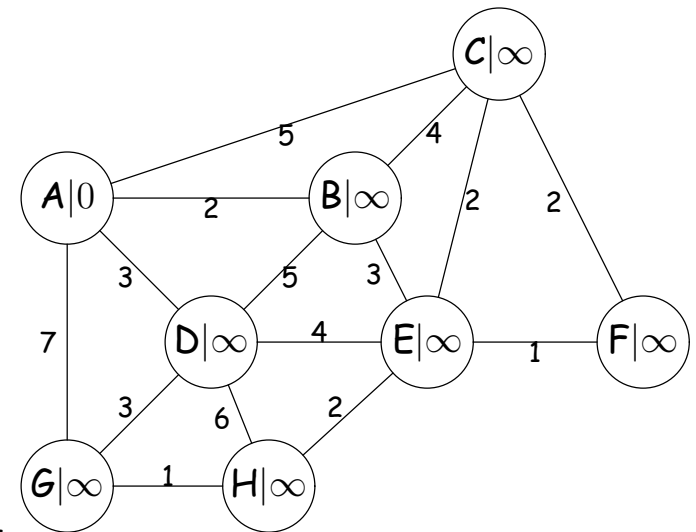

Minimum Spanning Trees

- **Problem**: Given a set of places and distances 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.

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 already in the tree to one that isn't yet.
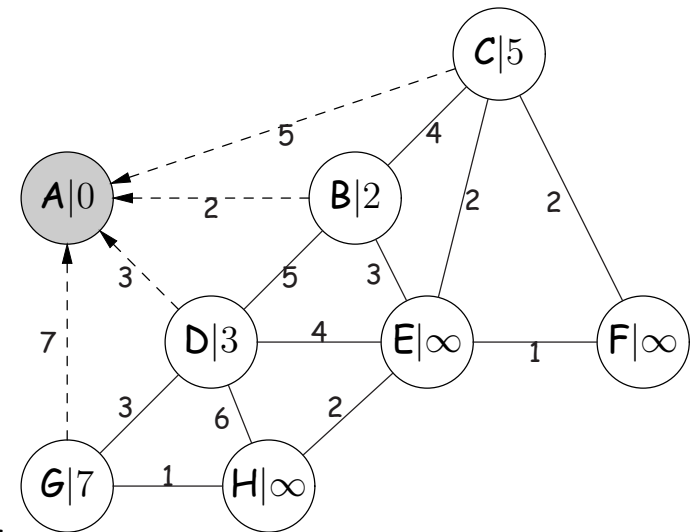
- Why must this work?

```
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 already in the tree to one that isn't yet.
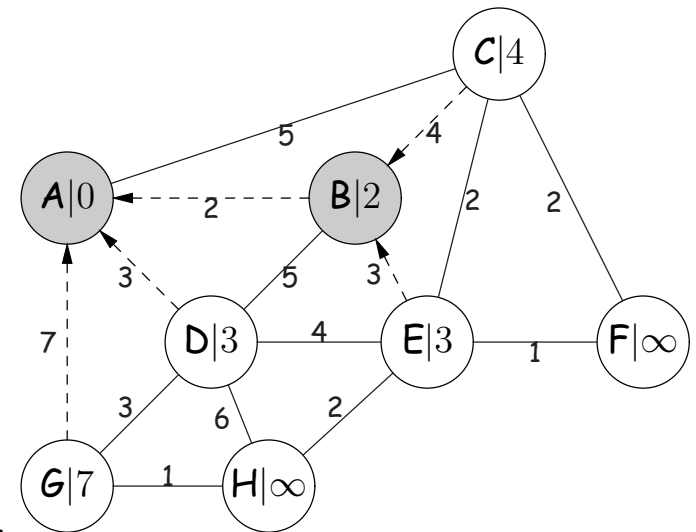
- Why must this work?

```
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 already 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 already in the tree to one that isn't yet.
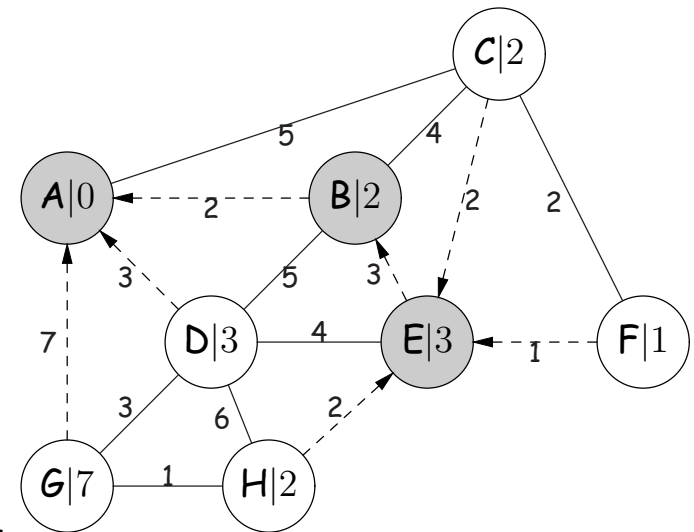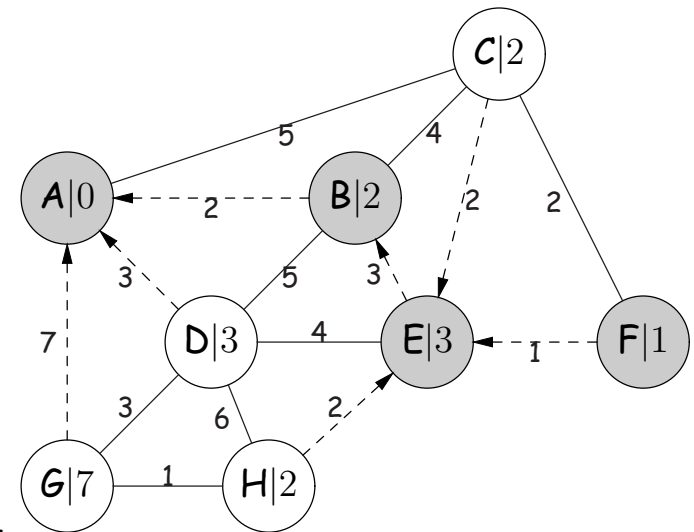
- Why must this work?

```
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 already in the tree to one that isn't yet.
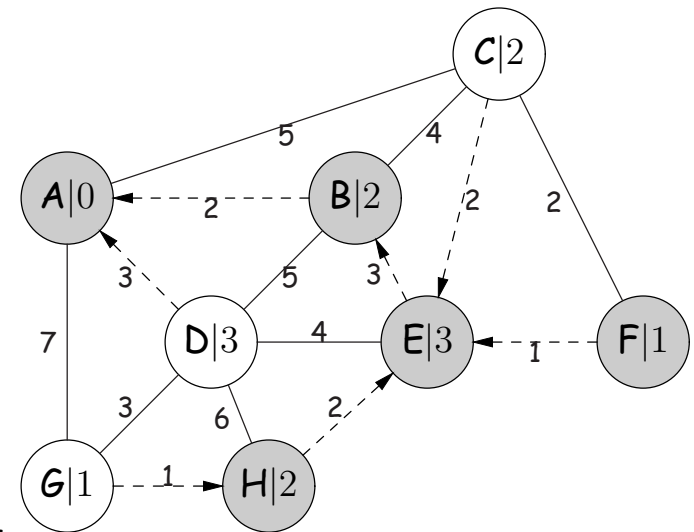
- Why must this work?

```
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 already 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 already in the tree to one that isn't yet.
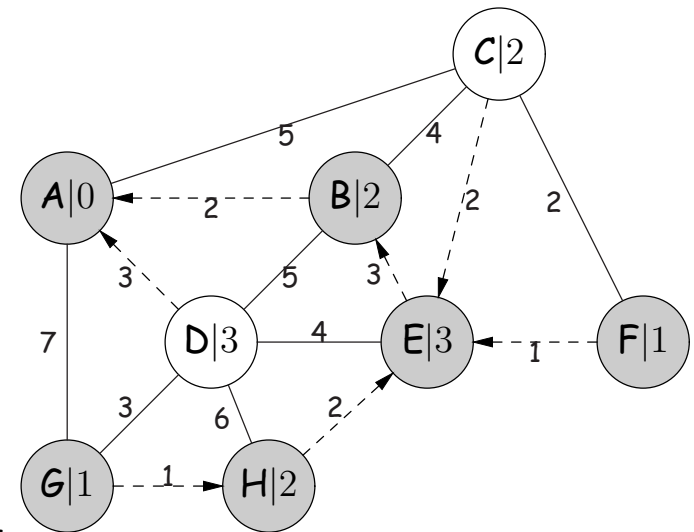
- Why must this work?

```
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 already in the tree to one that isn't yet.
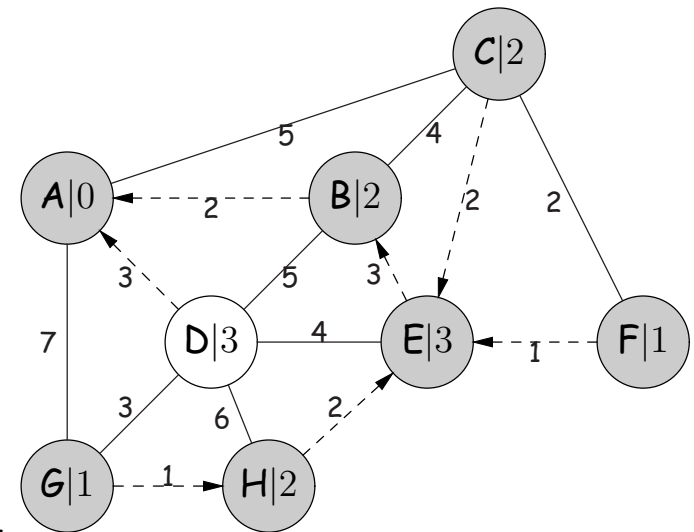
- Why must this work?

```
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; }
  }
}
```
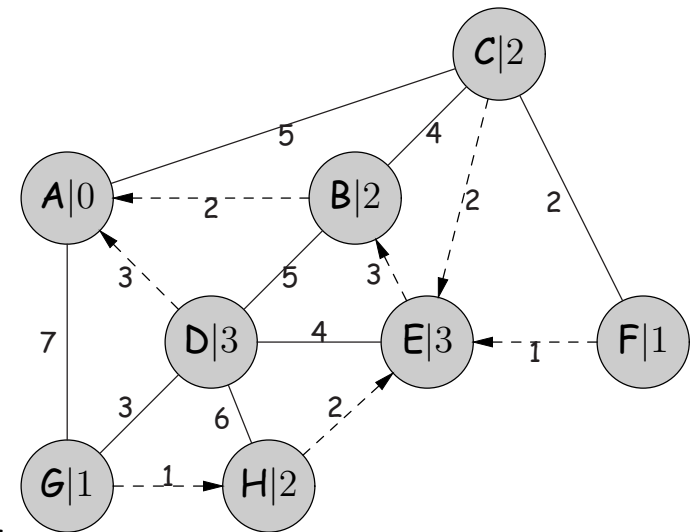
Minimum Spanning Trees by Prim's Algorithm

- Idea is to grow a tree starting from an arbitrary node.

- At each step, add the shortest edge connecting some node already 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 Algorithm

- 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;
    }
}
```

Recursive Depth-First Traversal

- Previously, we saw an iterative way to do depth-first traversal of a graph from a particular node.

- We are often interested in traversing all nodes of a graph, so we can repeat the procedure as long as there are unmarked nodes.

- Recursive solution is also simple:

```
void traverse (Graph G) {
    for (v ∈ nodes of G) {
        traverse (G, v);
    }
}

void traverse (Graph G, Node v) {
    if (v is unmarked) {
        mark (v);
        visit v;
        for (Edge (v, w) ∈ G)
            traverse (G, w);
    }
}
```

Another Take on Topological Sort

- Observation: if we do a depth-first traversal on a DAG whose edges are reversed, and execute the recursive `traverse` procedure, we finish executing `traverse(G,v)` in proper topologically sorted order.

```
void topologicalSort (Graph G) {
    for (v ∈ nodes of G) {
        traverse (G, v);
    }
}

void traverse (Graph G, Node v) {
    if (v is unmarked) {
        mark (v);
        for (Edge (w, v) ∈ G)
            traverse (G, w);
        add v to the result list;
    }
}
```

Union Find

- Kruskal's algorithm required that we have a 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?

A Clever Trick

- Let's choose to represent a set of nodes by *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 higher tree as the union representative).
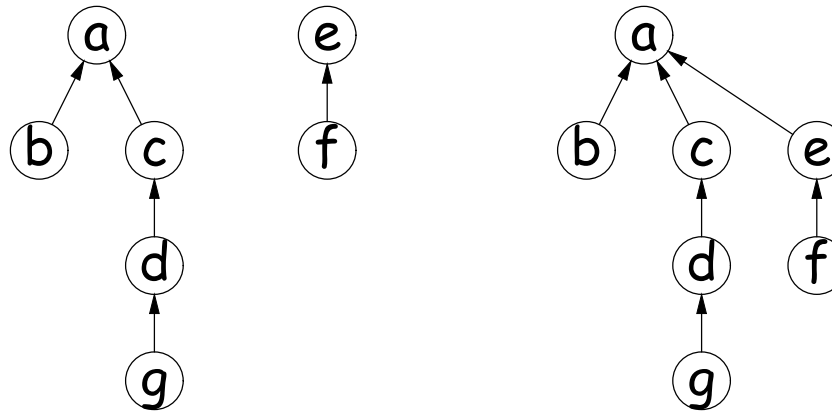
Path Compression

- This makes unioning really fast, but the find 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):