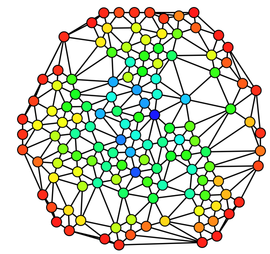
Adam Blank | Lecture 22 | Summer 2015

CSE 332

Data Abstractions

CSE 332: Data Abstractions

Graphs 4: Minimum Spanning Trees



Final Dijkstra's Algorithm


```
dijkstra(G, source) {
   dist = new Dictionary();
   worklist = [];
   for (v : V) {
      if (v == source) { dist[v] = 0; }
      else             { dist[v] = ∞; }
      worklist.add((v, dist[v]));
   }

   while (worklist.hasWork()) {
      v = next();
      for (u : v.neighbors()) {
         dist[u] = min(dist[u], dist[v] + w(v, u));
         worklist.decreaseKey(v, dist[u]);
      }
   }

   return dist;
}
```

Final Dijkstra's Algorithm


```
~~dijkstra~~(G, source) {
   dist = new Dictionary();
   worklist = [];
   for (v : V) {
      if (v == source) { dist[v] = 0; }
      else             { dist[v] = ∞; }
      worklist.add((v, dist[v]));
   }

   while (worklist.hasWork()) {
      v = next();
      for (u : v.neighbors()) {
         dist[u] = min(dist[u], ~~dist[v]~~ + w(v, u));
         worklist.decreaseKey(v, dist[u]);
      }
   }

   return dist;
}
```

What Does ~~Dijkstra's Algorithm~~ Do Now?

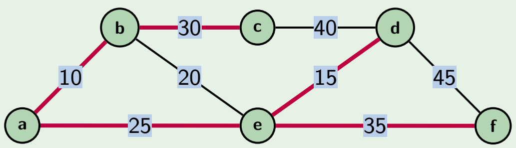
Minimum Spanning Trees

Definition (Minimum Spanning Tree)

Given a graph $G = (V, E)$, find a **subgraph** $G' = (V', E')$ such that

- G' is a **tree**.
- $V = V'$ (G' is **spanning**.)
- $\sum_{e \in E'} w(e)$ is **minimized**.

Example



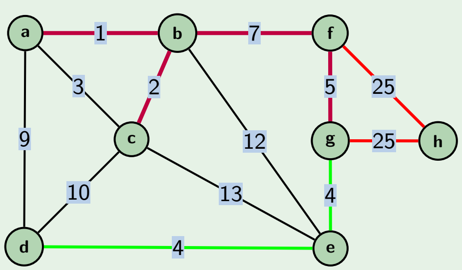
What For?

- Given a layout of houses, where should we place the phone lines to minimize cost?
- How can we design circuits to minimize the amount of wire?
- Implementing efficient multiple constant multiplications
- Minimizing the number of packets transmitted across a network
- Machine learning (e.g., real-time face verification)
- Graphics (e.g., image segmentation)

MST Example

MST Example

- Find a Minimum Spanning Tree of this graph
- Are there any others?
- Come up with a simple algorithm to find MSTs

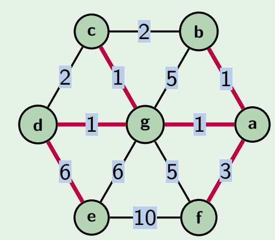


MST Uniqueness

If a graph has all unique edges, there is a unique MST. Otherwise, there might be multiple MSTs.

Back To ~~Dijkstra's~~ Prim's Algorithm


```
prim(G) {
   conns = new Dictionary();
   worklist = [];
   for (v : V) {
      conns[v] = null;
      worklist.add((v, ∞));
   }
   while (worklist.hasWork()) {
      v = next();
      for (u : v.neighbors()) {
         if (w(v, u) < w(conns[u], u)) {
            conns[u] = v;
            worklist.decreaseKey(
               v, w(v, u)
            );
         }
      }
   }
   return conns;
}
```



This really is almost identical to Dijkstra's Algorithm! We build up an MST by **adding vertices** to a "done set" and keeping track of what edge got us there.

Do we have to use vertices? Can we use edges instead?

A Simple Algorithm to Find MSTs

Simple MST

```
findMST(G) {
   mst = {};
   for ((v, w) ∈ sorted(E)) {
      foundV = foundW = false;
      for ((a, b) ∈ mst) {
         foundV |= (a == v) || (b == v);
         foundW |= (a == w) || (b == w);
      }
      if (!foundW || !foundW) {
         mst.add((v, w));
      }
   }
   return mst;
}
```

Some Questions!

- How many edges is the MST?
  Every MST will have $|V|-1$ edges; one edge to include each vertex
- What is the runtime of this algorithm? $\mathcal{O}(|E|\lg(|E|)+|E||V|)$, because sorting takes $\mathcal{O}(|E|\lg(|E|))$, the MST has at worst $\mathcal{O}(|V|)$ edges, and we have to iterate through the MST $|E|$ times.
- What is the slow operation of this algorithm? Checking if a vertex is already in our MST is very slow here. Can we do better?

Disjoint Sets ADT

A **disjoint sets** data structure keeps track of multiple sets which do not share any elements. Here's the ADT:

UnionFind ADT

`find(x)`	Returns a number representing the set that **x** is in.
`union(x, y)`	Updates the sets so whatever sets **x** and **y** were in are now considered the same sets.

Example

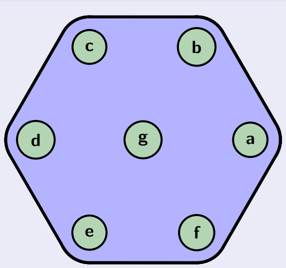
```
list = [1, 2, 3, 4, 5, 6];
UF uf = new UF(list); // State: {1}, {2}, {3}, {4}, {5}, {6}
uf.find(1);           // Returns 1
uf.find(2);           // Returns 2
uf.union(1, 2);       // State: {1, 2}, {3}, {4}, {5}, {6}
uf.find(1);           // Returns 1
uf.find(2);           // Returns 1
uf.union(3, 5);       // State: {1, 2}, {3, 5}, {4}, {6}
uf.union(1, 3);       // State: {1, 2, 3, 5}, {4}, {6}
uf.find(3);           // Returns 1
uf.find(6);           // Returns 6
```

Kruskal's Algorithm

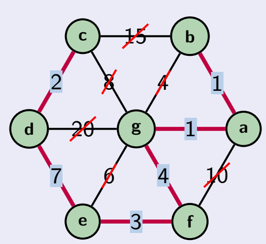
Simple MST

```
kruskal(G) {
   mst = {};
   forest = new UnionFind(V);
   for ((v, w) ∈ sorted(E)) {
      if (forest.find(v) != forest.find(w)) {
         mst.add((v, w));
         forest.union(v, w);
      }
   }
   return mst;
}
```

Forest



Graph



Kruskal's Algorithm Correctness

Proving Correctness

To prove that Kruskal's Algorithm is correct, we must prove:

1. The output is some spanning tree **The output is some spanning tree**
2. The output has minimum weight

Kruskal's Algorithm Outputs **SOME** Spanning Tree

We must show that the output, G' is spanning, connected, and acyclic.

- The algorithm adds an edge whenever one of its ends is not already in the tree. This means that every vertex has an edge in the tree.
- It's acyclic because we check before adding an edge.
- Connected?
  - The original graph is connected; there exists a path between u and v
  - Consider the **first** edge that we look at which is on **some path** between u and v.
  - Since we haven't previously considered **any** edge on **any** path between u and v,it must be the case that u and v are in distinct sets in the disjoint sets data structure. So, we add that edge.

  Since there is a path between every u and v in the graph in G', G' is connected by definition.

Kruskal's Algorithm Correctness

Proving Correctness

To prove that Kruskal's Algorithm is correct, we must prove:

1. The output is some spanning tree
2. **The output has minimum weight**

So, now, we know that G' **is a spanning tree**!

Kruskal's Algorithm Outputs Some **MINIMUM** Spanning Tree

Let the edges we add to G' be, in order, $e_1, e_2, \ldots e_k$.
Claim: For all $0 \le i \le k$, $\{e_1, e_2, \ldots e_i\} \subseteq T_i$ for **some** MST T_i.
Proof: We go by induction.
Base Case. $\emptyset \subseteq G$ for every graph G.
Induction Hypothesis. Suppose the claim is true for iteration i.
Induction Step. By our IH, we know that $\{e_1, \ldots, e_i\} \subseteq T_i$, where T_i is some MST of G.
We consider two cases:

- If $e_{i+1} \in T_i$,then we choose $T_{i+1} = T_i$, and we're done.
- Otherwise...

Kruskal's Algorithm Correctness

So far, we know...

- T_i is a spanning tree of G. (earlier proof)
- that $\{e_1, \ldots, e_i\} \subseteq T_i$, where T_i is some MST of G. (induction hypothesis)
- $e_{i+1} \notin T_i$. (handled that case)

Kruskal's Algorithm Outputs Some **MINIMUM** Spanning Tree (cont.)

Claim: For all $0 \le i \le k$, $\{e_1, e_2, \ldots e_i\} \subseteq T_i$ for **some** MST T_i.

- Since T is a spanning tree, it must have some other edge (call it e') which was added in place of e_{i+1}.
- It follows that $T_i + e_{i+1}$ must have a cycle!
- Note that $w(T - e' + e_{i+1}) = w(T) - w(e') + w(e)$.
- Since we considered e_{i+1} before e', and the edges were sorted by weight, we know $w(e) \le w(e') \iff w(e) - w(e') \le 0$.
- So,
$$w(T - e' + e_{i+1}) = w(T) - w(e') + w(e) \le w(T)$$

This means that $T - e' + e_{i+1}$ has no more than the weight of any MST!

Almost There...

So far, we know...

- T_i is a spanning tree of G. (earlier proof)
- that $\{e_1, \ldots, e_i\} \subseteq T_i$, where T_i is some MST of G. (induction hypothesis)
- $e_{i+1} \notin T_i$. (handled that case)
- $w(T - e' + e_{i+1}) \le w(T)$

Kruskal's Algorithm Outputs Some **MINIMUM** Spanning Tree (cont.)

Claim: For all $0 \le i \le k$, $\{e_1, e_2, \ldots e_i\} \subseteq T_i$ for **some** MST T_i.
Finally, choose $T_{i+1} = T - e' + e_{i+1}$.

- We already know it has the weight of an MST.
- Note that e connects the same nodes as e'; so, it's also a spanning tree.

That's it! For each i, we found an MST that extends the previous one. So, the last one must also be an MST!

Kruskal's Algorithm Runtime

- Sort takes $\mathcal{O}(n \lg n)$
- We don't know how `UnionFind` works, but if we know...
  - `find` is $\mathcal{O}(\lg n)$
  - `union` takes $\mathcal{O}(1)$ time

The runtime is $\mathcal{O}(|E| \lg(|E|) + |E| \lg(|V|))$

Due to time constraints, we won't explore how union-find works, but you can do some very interesting amortized analysis with it...