Minimum Spanning Tree

- Prim-Jarnik algorithm
- Kruskal algorithm

That’s a very nice hat.

That’s not a hat! That’s my head! I’m Tree Head!
Weighted Graphs

(weight of subgraph $G'$) =
(sum of weights of edges of $G'$)

$$\text{weight}(G') = \sum \text{weight}(e)$$

($e \in G'$)

$$\text{weight}(G') = 800 + 400 + 1200 = 2400$$
Minimum Spanning Tree

- spanning tree of minimum total weight
- e.g., connect all the computers in a building with the least amount of cable
- example

```
<table>
<thead>
<tr>
<th></th>
<th>MIA</th>
<th>SEA</th>
<th>SFO</th>
<th>LAX</th>
<th>DFW</th>
<th>STL</th>
<th>LGA</th>
<th>PVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEA</td>
<td>800</td>
<td></td>
<td></td>
<td>1500</td>
<td></td>
<td></td>
<td></td>
<td>200</td>
</tr>
<tr>
<td>SFO</td>
<td>400</td>
<td>800</td>
<td>1200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAX</td>
<td>1800</td>
<td>1500</td>
<td>1500</td>
<td></td>
<td>400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFW</td>
<td>1800</td>
<td>1500</td>
<td>1500</td>
<td></td>
<td></td>
<td>1500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STL</td>
<td>1000</td>
<td>800</td>
<td></td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LGA</td>
<td>1000</td>
<td>1200</td>
<td></td>
<td></td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PVD</td>
<td>200</td>
<td>1500</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- not unique in general
Minimum Spanning Tree Property

Let \((V', V'')\) be a partition of the vertices of \(G\)
Let \(e = (v', v'')\), be an edge of minimum weight across the partition, i.e., \(v' \in V'\) and \(v'' \in V''\).

*There is a MST containing edge \(e\).*
Proof of Property

If the MST does not contain a minimum weight edge \( e \), then we can find a better or equal MST by exchanging \( e \) for some edge.
Prim-Jarnik Algorithm for finding an MST

- grows the MST $T$ one vertex at a time
- *cloud* covering the portion of $T$ already computed
- labels $D[u]$ and $E[u]$ associated with each vertex $u$
  - $E[u]$ is the best (lowest weight) edge connecting $u$ to $T$
  - $D[u]$ (distance to the cloud) is the weight of $E[u]$
Differences between Prim’s and Dijkstra’s

• For any vertex u, $D[u]$ represents the weight of the current best edge for joining u to the rest of the tree (as opposed to the total sum of edge weights on a path from start vertex to u).

• Use a priority queue Q whose keys are D labels, and whose elements are vertex-edge pairs.

• Any vertex v can be the starting vertex.

• We still initialize all the D[u] values to INFINITE, but we also initialize E[u] (the edge associated with u) to null.

• Return the minimum-spanning tree T.

We can reuse code from Dijkstra’s, and we only have to change a few things. Let’s look at the pseudocode....
Algorithm PrimJarnik($G$):

- **Input:** A weighted graph $G$.
- **Output:** A minimum spanning tree $T$ for $G$.

pick any vertex $v$ of $G$
{grow the tree starting with vertex $v$}

$T \leftarrow \{v\}$

$D[u] \leftarrow 0$

$E[u] \leftarrow \emptyset$

for each vertex $u \neq v$ do

$D[u] \leftarrow +\infty$

let $Q$ be a priority queue that contains
vertices, using the $D$ labels as keys

while $Q \neq \emptyset$ do

{pull $u$ into the cloud C}

$u \leftarrow Q$.removeMinElement()

add vertex $u$ and edge $E[u]$ to $T$

for each vertex $z$ adjacent to $u$ do

if $z$ is in $Q$

{perform the relaxation operation on edge $(u, z)$ }\)

if weight($u, z$) < $D[z]$ then

$D[z] \leftarrow$ weight($u, z$)

$E[z] \leftarrow (u, z)$

change the key of $z$ in $Q$ to $D[z]$

return tree $T$
Let’s go through it

Minimum Spanning Tree

<table>
<thead>
<tr>
<th>neighbor</th>
<th>D[u]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFW</td>
<td>STL</td>
</tr>
<tr>
<td>LAX</td>
<td>STL</td>
</tr>
<tr>
<td>LGA</td>
<td>STL</td>
</tr>
<tr>
<td>MIA</td>
<td>STL</td>
</tr>
<tr>
<td>MSN</td>
<td>STL</td>
</tr>
<tr>
<td>PVD</td>
<td></td>
</tr>
<tr>
<td>SEA</td>
<td></td>
</tr>
<tr>
<td>SFO</td>
<td></td>
</tr>
<tr>
<td>STL</td>
<td></td>
</tr>
</tbody>
</table>
Minimum Spanning Tree
Minimum Spanning Tree
Minimum Spanning Tree

neighbor

<table>
<thead>
<tr>
<th>Neighbor</th>
<th>D[u]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFW</td>
<td>1500</td>
</tr>
<tr>
<td>LAX</td>
<td>DFW</td>
</tr>
<tr>
<td>LGA</td>
<td>1000</td>
</tr>
<tr>
<td>MIA</td>
<td>DFW</td>
</tr>
<tr>
<td>MSN</td>
<td>1000</td>
</tr>
<tr>
<td>PVD</td>
<td>LGA</td>
</tr>
<tr>
<td>SEA</td>
<td>MSN</td>
</tr>
<tr>
<td>SFO</td>
<td></td>
</tr>
<tr>
<td>STL</td>
<td></td>
</tr>
</tbody>
</table>
Running Time

T ← \{v\}

\[ D[u] ← 0 \]

\[ E[u] ← \emptyset \]

for each vertex \( u \neq v \) do

\[ D[u] ← +\infty \]

let \( Q \) be a priority queue that contains all the vertices using the \( D \) labels as keys

while \( Q \neq \emptyset \) do

\[ u ← Q.\text{removeMinElement()} \]

add vertex \( u \) and edge \( E[u] \) to \( T \)

for each vertex \( z \) adjacent to \( u \) do

if \( z \) is in \( Q \)

if \( \text{weight}(u, z) < D[z] \) then

\[ D[z] ← \text{weight}(u, z) \]

\[ E[z] ← (u, z) \]

change the key of \( z \) in \( Q \) to \( D[z] \)

return tree \( T \)

\[ O((n+m) \log n) \]

where \( n = \text{num vertices} \), \( m = \text{num edges} \),
and \( Q \) is implemented with a heap.
Kruskal Algorithm

• add the edges one at a time, by increasing weight
• accept an edge if it does not create a cycle
Data Structure for Kruskal Algorithm

• the algorithm maintains a forest of trees

• an edge is accepted if it connects vertices of distinct trees

• we need a data structure that maintains a partition, i.e., a collection of disjoint sets, with the following operations
  - find(u): return the set storing u
  - union(u,v): replace the sets storing u and v with their union
Representation of a Partition

- each set is stored in a sequence
- each element has a reference back to the set

![Diagram of a partition representation]

- operation $\text{find}(u)$ takes $O(1)$ time, and returns the set of which $u$ is a member.

- in operation $\text{union}(u,v)$, we move the elements of the smaller set to the sequence of the larger set and update their references

- the time for operation $\text{union}(u,v)$ is $\min(n_u,n_v)$, where $n_u$ and $n_v$ are the sizes of the sets storing $u$ and $v$

- whenever an element is processed, it goes into a set of size at least double

- hence, each element is processed at most $\log n$ times
#### Algorithm Kruskal(G):

**Input:** A weighted graph G.

**Output:** A minimum spanning tree T for G.

let P be a partition of the vertices of G, where each vertex forms a separate set

let Q be a priority queue storing the edges of G, sorted by their weights

T ← ∅

while Q ≠ ∅ do

    (u,v) ← Q.removeMinElement()

    if P.find(u) ≠ P.find(v) then

        add edge (u,v) to T

        P.union(u,v)

return T

Running time: O((n+m) log n)
Let’s go through it

Minimum Spanning Tree
Minimum Spanning Tree
Minimum Spanning Tree
Minimum Spanning Tree
Now examine LGA-MIA, but don’t add it to T cause LGA and MIA are in the same set.

Now examine LAX-STL, but don’t add it to T cause LAX and STL are in the same set. And we’re done.