Minimal Spanning Trees
Introduction

Start with a connected, undirected graph, and add real-valued weights to each edge.

The weights could indicate time, distance, cost, capacity, etc.
Definitions

A *spanning tree* of a graph $G$ is a tree that contains every vertex of $G$.

The *weight* of a tree is the sum of its edges’ weights.

A *minimal spanning tree* is a spanning tree with lowest weight. (The left tree is not minimal. The right one is, as we will see.)
An Application of MSTs

Wire up a network among several computers so that every computer can reach (directly or indirectly) every other. Use the minimum amount of cable.

Vertices = computers

Weights = distance between computers

Another example: think about inter-connecting a set of n pins, using n−1 wires, using the minimal amount of wire.
The MST Property: Intro

Divide the vertices of a graph into two sets (this is a cut of the graph).

Consider an edge of lowest weight on the cut (e.g. 4, above).

This edge is in some MST of the graph.
Proving the MST Property: 1

Recall: A tree is an acyclic, connected, undirected graph.

**Lemma 1**: Adding an edge to a tree results in a cycle.

**Proof**: Say the edge is from $u$ to $v$. There was already a path from $v$ to $u$ (since the tree is connected); now there is an edge from $u$ to $v$, forming a cycle.
Proving the MST Property: 2

**Lemma 2**: Adding an edge to a tree, then removing a different edge along the resulting cycle, still results in a tree.

![Diagram showing the process of adding an edge, forming a cycle, and removing a different edge to form a tree again.]

**Proof**: Omitted.
The MST Property

**Theorem**: Given a cut of a graph, a lowest-weight edge crossing the cut will be in some MST for the graph.

**Proof**: By contradiction. Assume an MST on the graph containing no lowest-weight edge crossing the cut. Then some other, higher-weight edge on the cut is in the MST (since it is a spanning tree). (continued)
If we now add the lowest-weight edge to the supposed MST, we have a cycle. Removing the higher-weight one still results in a spanning tree, and it has a lower total weight than the alleged MST. Thus the assumption is false: some lowest-weight edge is in the MST. ■
MST Algorithms

Both of the MST algorithms we will study exploit the MST property.

- Kruskal’s: repeatedly add the lowest-weight legal edge to a growing MST.
- Prim’s (really Prim-Jarvik): keep track of a cut, and add the lowest-weight edge across the cut to the growing MST.
Kruskal’s Algorithm

Choose the lowest-weight edge in the graph. This is certainly the lowest weight edge for some cut, so it must be in an MST.

Continue taking the lowest-weight edge and adding it to the MST, unless it would result in a cycle.
Kruskal’s Example

\[\text{Diagram of a graph with weights on edges.} \]
MST - Kruskal\((G, w)\)

1. \(A \leftarrow 0\)  \(\triangleright\)  \(A\) will be the set of edges that is the MST
2. \textbf{for} each vertex \(v \in V[G]\)
3. \hspace{1em} \textbf{do} \text{Make - Set}(v)
4. \hspace{1em} sort the edges of \(E\) by nondecreasing weight \(w\)
5. \textbf{for} each edge \((u, v) \in E\), in order by nondecreasing weight
6. \hspace{1em} \textbf{do if} Find - Set\((u) \neq\) Find - Set\((v)\)
7. \hspace{2em} \textbf{then} \(A \leftarrow A \cup \{(u, v)\}\)  \(\triangleright\) add \((u, v)\) to the MST
8. \hspace{2em} \text{Union}(u, v)
9. \textbf{return} \(A\)
Aside: The Union-Find Problem

(This relates to the running time of Kruskal...)

Given several items, in disjoint sets, support three operations:

- Make-Set(x): create a set containing x
- Find-Set(x): return the set to which x belongs
- Union(x, y): take the union of sets of x and y

The “disjoint-set forests” implementation of union-find, with certain heuristics, yields outstanding performance. The running time of these operations is (for all practical purposes) linear in m (the total number of operations). See section 21.3 for more details...
Fast Union-Find

Choose an item as the representative of its set.
  – Find-Set(x): return x’s representative
  – Union(x, y): make x’s representative point to y

With path compression:
  – Union(x, y): make x and everything on the path to its representative point to y

Time for $n$ Union-finds: $\Theta(n\lg^*n)$.

$\lg^*n$ is the iterated logarithm function: the number of times you must take $\lg n$ in order to get $\leq 1$.

For all practical purposes, $\lg^*n < 4$. 
Running Time of Kruskal’s

MST - Kruskal(G, w)
1  \( A \leftarrow 0 \)  \( \triangleright \) A will be the set of edges that is the MST
2 \textbf{for} each vertex \( v \in V[G] \)
3   \textbf{do} Make/Set(\( v \))
4  sort the edges of \( E \) by nondecreasing weight \( w \)
5 \textbf{for} each edge \( (u, v) \in E \), in order by nondecreasing weight
6   \textbf{do} if Find/Set(\( u \)) \( \neq \) Find/Set(\( v \))
7      \textbf{then} \( A \leftarrow A \{(u, v)\} \)  \( \triangleright \) add \( (u, v) \) to the MST
8       Union(\( u, v \))
9 \textbf{return} \( A \)

Total: \( \Theta(ElgE) \) time.

\( \Theta(ElgE) \)

\( \Theta(Elg*E) \)

\( \Theta(ElgE) \) time.

7-MST
Prim’s Algorithm

Begin with any vertex. Choose the lowest weight edge connected to that vertex.

Add the vertex on the other side of that edge to the “active set.”

Again, choose the lowest-weight edge of any vertex in the active set that connects to a vertex outside the set, and repeat.
Prim’s Example
Why Prim’s Works

It comes right out of the MST property.

There are always two sets, the active set and everything else, making a cut. The MST property says that the lowest-weight edge crossing the cut is in an MST. This is just the edge Prim’s chooses.
High-level Code for Prim’s

MST - Prim(G)

\[ S \leftarrow \{ \text{any vertex in } V[G] \} \uparrow \text{active set} \]

\[ T \leftarrow \emptyset \uparrow \text{set of edges in MST (} \emptyset = \text{empty set)} \]

while \( S \neq V[G] \) do

choose \((u, v)\), a lowest - weight edge such that

\[ u \in S, \text{ and } v \in V[G] - S \]

\[ T \leftarrow T \cup \{(u, v)\} \]

\[ S \leftarrow S \cup \{v\} \]

return \( T \)

7-MST
High-Level Prim (Cont.)

How can we choose an edge without looking through every edge in the graph?

Use a priority queue. But make it a queue of vertices, not edges.
A Sophisticated Implementation of Prim’s

1. Build a minimizing priority queue from the graph’s vertices, where the key of a vertex (the value minimized) is the weight of the lowest-weight edge from that vertex to a vertex in the MST.

2. Repeatedly do the following, until the queue is empty:
   a. Extract the minimum vertex from the queue. Call it \( u \).
   b. For each vertex \( v \) adjacent to \( u \) that is still in the queue, if the weight of the edge between \( u \) and \( v \) is smaller than \( key[v] \), then update \( key[v] \) to that weight, and set the parent of \( v \) to be \( u \).
Prim’s à la CLRS

MST - Prim( G, w, r)

1. \( Q \leftarrow V[G] \)
2. for each \( v \in Q \)
3. \( \text{do } key[v] \leftarrow \infty \)
4. \( key[r] \leftarrow 0 \)
5. \( \pi [r] \leftarrow \text{NIL} \)
6. while \( Q \neq 0 \)
7. \( \text{do } u \leftarrow \text{EXTRACT-MIN}(Q) \)
8. for each \( v \in Adj[u] \)
9. \( \text{do if } v \in Q \text{ and } w(u, v) < key[v] \)
10. \( \text{then } \pi[v] \leftarrow u \)
11. \( key[v] \leftarrow w(u, v) \)

\( Q \) contains all the vertices that are not in the MST.

\( key[v] \) is the weight of the lowest-weight edge from vertex \( v \) to a vertex in the MST.
Choose A to be the root.

<table>
<thead>
<tr>
<th>key</th>
<th>$\pi$</th>
<th>$u$ (min vertex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>F</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

7-MST
Running Time of Prim

MST - Prim( G, w, r)

1 \( Q \leftarrow V[ G ] \)
2 \textbf{for each} \( v \in Q \) \( \theta(v) \) (BUILD - HEAP)
3 \( \textbf{do} \) \( \text{key}[v] \leftarrow \infty \)
4 \( \text{key}[r] \leftarrow 0 \)
5 \( \pi[r] \leftarrow \text{NIL} \)
6 \textbf{while} \( Q \neq 0 \) \( \theta(v) \) iterations
7 \( \textbf{do} u \leftarrow \text{EXTRACT-MIN}(Q) \) \( \text{O}(\lg V) \) per iteration
8 \textbf{for each} \( v \in \text{Adj}[u] \) \( \theta(E) \) total
9 \textbf{do if} \( v \in Q \) and \( w(u,v) < \text{key}[v] \) \( \theta(1) \) per iteration
10 \textbf{then} \( \pi[r] \leftarrow u \)
11 \( \text{key}[v] \leftarrow w(u,v) \) \( \text{O}(\lg V) \) per iteration
Notes:
- The $v \in Q$ test can be done in constant time by associating a bit with each vertex indicating whether or not it is in the queue.
- Line 11 is a call to DECREASE-KEY. That can be done in $\theta(\ln n)$ time worst case from a heap of $n$ items.

Result: $O(V + V\ln V + E\ln V) = O(V\ln V + E\ln V) = O(E\ln V)$
Same as Kruskal's algorithm.
Fibonacci Heaps

It is possible to do better. A Fibonacci Heap is a priority queue implementation that can do

- EXTRACT-MIN in $\Theta(lgn)$
- DECREASE-KEY in $\Theta(1)$

amortized time. (See CLRS, Chapter 20).

Improves the time from $\Theta(ElgV)$ to $\Theta(VlgV+E)$. (We had $\Theta(VlgV+ElgV)$, but the second $lgV$ becomes 1.)

This is better for dense graphs ($E \approx V^2$).

Theoretical interest only — Fibonacci heaps have a large constant.