Lecture 16:
Minimum Spanning Trees

(Reading: AM&O Chapter 13)
The Minimum Spanning Tree Problem

Given: Undirected connected network \( G = (N, A) \) with positive weights \( c_{ij} \) associated with each arc \((i, j)\) of \( G \).

Find: A subset \( T \) of arcs of \( G \) of minimum weight which connects all of the nodes of \( G \).

Fact: \( T \) will always be a spanning tree on \( G \), that is, the subgraph \( G_T = (N, T) \) will have the properties:

- \( G_T \) is connected (a spanning graph)
- \( G_T \) contains no cycles (a forest)

We will usually use \( T \) to refer to \( G_T \).
Applications

Communications Networks: Suppose AT&T or GTE wishes to upgrade a region of telephone users with a fiber-optic network. The cost of upgrading a link between user \( i \) and user \( j \) is \( c_{ij} \). How can they connect this set of users at minimum cost?

Electronic Circuitry: A system with a number of electronic components needs to be connected with power source \( s \), either directly or through some set of intermediate components. It takes amount \( c_{ij} \) of wire to connect component \( i \) to component \( j \). How can the set of components be given electrical power using the minimum amount of wire?
Minimax Path Problem: The quality of an $(s,t)$-path is often measured by the quality of its worst arc. Examples would be if you wanted to maintain a temperature/humidity threshold all along a particular route, or if you wanted a point-to-point communication link with a given minimum capacity. The **max-value** of an $(s,t)$-path $\Gamma$ on a network with arc costs is the cost of the **maximum-cost arc** on $\Gamma$. The **minimax path problem** is that of finding the $(s,t)$-path with the minimum max value. (The **maximin path problem** is defined similarly, and can be solved by solving the minimax path problem on the network with all costs **negated**.)

It will be shown that the minimum (maximum) spanning tree $T$ has the property that the (unique) path between any two points in $T$ is the minimax (maximin) path between those points.
**Well-mixing:** Two alloys are mixed, and then X-ray testing is performed on the surface of the mixture to identify the relative amount of each alloy which appears at chosen points throughout the mixture. One appropriate measure for this might be the maximum “jump” in the proportion of the amounts of alloys that might occur when moving from one part of the surface to the other. The minimax path problem could model this, and the sum of all edges in the associated minimum spanning tree would be the measure of how “well-mixed” the alloy is.
**Cluster Analysis:** It is desired to group a set of points on the plane into “clusters” of points “close” to each other. For example, the points might represent locations to be serviced by a small number of servicepersons, and the clusters would represent those locations assigned to one serviceperson. How can the clusters be grouped so as to give the servicepeople “compact” areas?

It turns out that the construction of the minimum spanning tree using Kruskal’s algorithm gives intermediate forests having successively $n, n - 1, \ldots, 2, 1$ components. Depending on how many clusters are desired, the components could serve as the clusters.
Related Problems

**Steiner tree problem:** Here we are interested in connecting only a *subset* \( K \) of the nodes of the network, and we can perform the connections through non-\( K \)-nodes — called *Steiner nodes* — of the network.

**Traveling salesman problem:** Here we must connect all of the nodes using a *single circuit* of minimum length.

**Survivability problems:** Here we must connect all of the nodes by a set that remains connected after removal of any one arc — *i.e.*, a 2-connected set.

All three of these problems are computationally hard, but good heuristics can be obtained by using minimum spanning tree algorithms as subroutines.
Properties of Spanning Trees

For spanning tree $T$

- $T$ has $n - 1$ edges.
- $T$ is a minimal spanning graph
- $T$ is a maximal forest
- the addition of a nontree edge of $G$ to $T$ will always create a unique cycle.
- the deletion of a tree edge of $G$ to $T$ will always create a unique cut $(X, \bar{X})$ with $(X, \bar{X}) = \emptyset$. 
Matroids and the Basis Exchange Property

The key property that is the basis for our spanning tree algorithms is the **basis exchange property**. Let \( \mathcal{T} \) be a collection of subsets, called **bases**, of a **ground set** \( A \) having the following two properties:

1. All sets in \( \mathcal{T} \) have the same cardinality.

2. **Basis Exchange Property**: For any two sets \( T_1, T_2 \in \mathcal{T} \) and any element \( e \in T_2 \setminus T_1 \), there is an element \( f \in T_1 \setminus T_2 \) such that \( T_1 \cup \{e\} \setminus \{f\} \in \mathcal{T} \).

Any such collection \( \mathcal{T} \) forms a **matroid** on \( A \).
Spanning Trees and Matroids

**Theorem:** The collection of spanning trees of a graph forms a matroid on the set of edges.

**Proof:** Let $T_1$ and $T_2$ be spanning trees, and let $e \in T_2 \setminus T_1$. Then adding $e$ to $T_1$ creates a unique cycle $C$ in $T_1$, and because $T_2$ contains no cycles, then there must be at least one arc $f$ in $C$ that is not in $T_2$. Removing $f$ from $T_1 \cup \{e\}$ will break this cycle and again form a spanning tree, and so the basis exchange property is satisfied.
Some Classes of Matroids

**Graphic matroids:** $A$ is the set of arcs of a connected network, and $\mathcal{T}$ is the collection of **spanning trees**.

**Linear Matroids:** $A$ is a set of vectors in $\mathbb{R}^n$, and $\mathcal{T}$ is the collection of **bases (maximal linearly independent sets)** of the span of $A$.

**Transversal matroids:** $A$ is the set of nodes on one side of a bipartite graph, and $\mathcal{T}$ is the collection of nodes of $A$ adjacent to **maximum cardinality matchings**.
The Greedy Algorithm

**Given:** Ground set $A$ whose elements have weights $w_e, e \in A$, and collection $\mathcal{T}$ of subsets $A$ of equal cardinality.

**Find:** The set $T^* \in \mathcal{T}$ with **minimum total weight**.

**Greedy Algorithm:** Build $T$ by adding elements one-by-one, always adding the **minimum weight element** $e$ to the current set $\tilde{T}$ so that the set $\tilde{T} \cup \{e\}$ is still a subset of some element in $\mathcal{T}$.

**Theorem:** If $\mathcal{T}$ satisfies the basis exchange property, then the greedy algorithm will always give a minimum weight element of $\mathcal{T}$.
Proof

Let $T^g$ be the basis produced by the Greedy Algorithm, and let $T^*$ be a minimum weight basis having the maximum number of elements in common with $T^g$. Suppose $T^* \neq T^g$. Order the elements in $T^*$ and $T^g$ $e^*_1, \ldots, e^*_{n-1}$ and $e^g_1, \ldots, e^g_{n-1}$, respectively, in order of increasing weight. Let $i$ be the first index for which $e^g_i$ is not in $T^*$. Since $T$ satisfies the basis exchange property, then if we add $e^g_i$ to the tree $T^*$ there must be at least one element $e^*_l \in T^* \setminus T^g$ such that $T^* \cup \{e^g_i\} \setminus \{e^*_l\}$ is still a basis. But the greedy algorithm chose $e^g_i$ as the minimum weight element that can be added at that point, and so its weight must be less than that of $e^*_l$. Thus $T^* \cup \{e^g_i\} \setminus \{e^*_l\}$ has weight no greater than $T^*$ and more edges in common with $T^g$ than did $T^*$, contradicting the choice of $T^*$. Thus $T^g = T^*$, and so $T^g$ is a minimum spanning tree for $G$. 
Applying the Greedy Algorithm to the Spanning Tree Problem: Two Approaches

- **Kruskal’s Algorithm**: The current solution $T'$ is always a *forest*, and the arc added is the cheapest arc which does not form a cycle with $T'$.

- **Prim’s Algorithm**: Starting at arbitrary node $s$, the current solution $T'$ is always a *tree rooted at* $s$, and the arc added is the cheapest arc which connects $s$ to a new node of $G$ (through $T'$).
Kruskal’s Algorithm

Assume arcs are ordered $e_1, \ldots, e_n$ in order of nondecreasing cost.

The Algorithm

**Initialize:** $T' = \emptyset$

**while** $T'$ is not spanning **do**
  Add to $T'$ the next arc in the ordering connecting two components of $T'$ (equivalently, not forming a cycle with $T'$)
Example
Order of arclengths:

<table>
<thead>
<tr>
<th>arc</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6,10)</td>
<td>2</td>
</tr>
<tr>
<td>(1,4)</td>
<td>4</td>
</tr>
<tr>
<td>(4,8)</td>
<td>5</td>
</tr>
<tr>
<td>(5,9)</td>
<td>5</td>
</tr>
<tr>
<td>(4,5)</td>
<td>6</td>
</tr>
<tr>
<td>(6,7)</td>
<td>7</td>
</tr>
<tr>
<td>(8,9)</td>
<td>8</td>
</tr>
<tr>
<td>(1,5)</td>
<td>9</td>
</tr>
<tr>
<td>(2,6)</td>
<td>9</td>
</tr>
<tr>
<td>(7,10)</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>arc</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5,6)</td>
<td>10</td>
</tr>
<tr>
<td>(1,2)</td>
<td>11</td>
</tr>
<tr>
<td>(9,10)</td>
<td>11</td>
</tr>
<tr>
<td>(2,5)</td>
<td>12</td>
</tr>
<tr>
<td>(6,9)</td>
<td>12</td>
</tr>
<tr>
<td>(3,7)</td>
<td>13</td>
</tr>
<tr>
<td>(5,8)</td>
<td>13</td>
</tr>
<tr>
<td>(3,6)</td>
<td>14</td>
</tr>
<tr>
<td>(2,3)</td>
<td>15</td>
</tr>
</tbody>
</table>

Diagram of the network with arclengths and costs.
**Complexity:** Need to perform one pass through the list of arcs (in increasing order of costs), and for each arc test for connection (or cycle).

A naive test involves application of FINDPATH on the forest ($O(n)$ — *why?*) per edge tested, for a total of $O(mn)$ for the algorithm.

**Improved Implementation:** Give each component of $T'$ a number, and then just test the current edge $e = (i, j)$ to see if $i$ and $j$ have the same component number.

**Operations:**

- $\text{find}(i)$: determine what component $i$ is in
- $\text{merge}(k, l)$: combine (disjoint) components $k$ and $l$
Implementation of Operations:

Give each vertex $i$ a number $c(i)$ representing the component it is in (initially each vertex is in its own component). Give each component $k$ a number $size(k)$ representing the number of vertices in that component (initially $size(i) = 1$).

$find(i)$ now just returns $c(i)$.

$merge(l, k)$: Take the smaller of the two components, according to $size$, and change the component numbers in each vertex of the smaller component to be that of the larger component. (This requires storing the components in a linked list). Set $size$ of the new component to be the sum of the sizes of the two merged components.
Complexity of operations:

*find*: $O(1)$

*merge*: $O(n \log n)$ for all operations (each vertex can be relabeled at most $\log n$ times).

*Complexity of this implementation of Kruskal’s Algorithm*: $O(m + n \log n)$ (+ time to sort arcs initially).
A faster implementation:

Keep the vertices of the components in trees, with the number of the component stored in the root. Now merge involves just hooking up the roots of the two component trees ($O(1)$), and find involves going up the tree from the argument vertex to find the root. By merging “shorter” trees into “taller” trees, and “dragging” up vertices during a find, the find operation requires amortized time $O(\alpha(m, n))$ per find, where $\alpha(m, n)$ is, for any practical problem, never greater than 6. Thus this implementation of Kruskal’s algorithm is essentially linear, assuming we can do the initial sorting of edges by cost.
Dijkstra’s Algorithm

set labels \( d_v = \begin{cases} 
0 & v = s \\
\infty & \text{otherwise}
\end{cases} \)

set \( X = \{s\}, \ NT = \emptyset \)

Comment: \( NT\) = nodes of curr. partial shortest path tree, \( X\) = nodes of \( E \setminus NT\) reachable from \( NT\) via a single arc, \( d(i)\) = shortest \((s,i)\)-path all of whose arcs except possibly \( i\) are in \( NT\).

while \( NT \neq N\)
begin
select \( x \in X\) with smallest value of \( d_v\)
for \( y \in A(x)\) do
begin
if \( y \notin X\) then add \( y\) to \( X\)
if \( d_y > d_x + c_{xy}\) then
\{set \( d_y = d_x + c_{xy}\), \( \text{Pred}(y) = x\}\}
remove \( x\) from \( X\) and add it to \( NT\)
end
end
Prim’s Algorithm

set labels \( d_v = \begin{cases} 
0 & v = s \\
\infty & \text{otherwise}
\end{cases} \)

set \( X = \{s\} \), \( N_T = \emptyset \)

Comment: \( N_T \) = nodes of curr. partial \( s \)-rooted MST, 
\( X \) = nodes of \( E \setminus N_T \) reachable from \( N_T \) via a single arc, \( d(i) \) = min weight of the min-cost arc connecting \( i \) to \( N_T \).

while \( N_T \neq N \)
begin
select \( x \in X \) with smallest value of \( d_v \)
for \( y \in A(x) \) do
begin
if \( y \notin X \) then add \( y \) to \( X \)
if \( d_y > c_{xy} \) then
\{set \( d_y = c_{xy}, \ \text{Pred}(y) = x \}\}
end
remove \( x \) from \( X \) and add it to \( N_T \)
end
<table>
<thead>
<tr>
<th>stage</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>edge added</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>∞</td>
<td>4</td>
<td>9</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>(1,4)</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>11</td>
<td>∞</td>
<td>4</td>
<td>6</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>(4,8)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>11</td>
<td>∞</td>
<td>4</td>
<td>6</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
<td>8</td>
<td>∞</td>
<td>(4,5)</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>11</td>
<td>∞</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>∞</td>
<td>5</td>
<td>5</td>
<td>∞</td>
<td>(5,9)</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>11</td>
<td>∞</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>∞</td>
<td>5</td>
<td>5</td>
<td>11</td>
<td>(5,6)</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>9</td>
<td>14</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>(6,10)</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>9</td>
<td>14</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>(6,7)</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>9</td>
<td>13</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>(2,6)</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>9</td>
<td>13</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>(7,3)</td>
</tr>
</tbody>
</table>
Complexity of Prim’s Algorithm

Need same data structures as Dijkstra to determine the choice of min \( d(i) \) vertex.

- Dial implementation not applicable, since \( d(i) \)'s are not chosen in increasing order of value throughout the algorithm, although Johnson's data structure(?) will give an \( O(m \log \log C) \) implementation.

- \( d \)-heaps and Fibonacci heaps work here, to provide complexities of \( O(m \log \frac{m}{n} n) \) and \( O(m + n \log n) \), respectively.

- Note that the arcs are not required to be presorted in any of these implementations.