Chapter 23: Minimal Spanning Trees.

Context: Weighted, connected, undirected graph, $G = (V, E)$, with $w : E \to \mathcal{R}$.

Definition: A selection of edges from $T \subseteq E$ such that $(V, T)$ is a tree is called a spanning tree for $G$.

Goal: $T \subseteq E$ with the following properties.

(a) $(V, T)$ is a spanning tree. That is, given two distinct vertices $x, y \in V$, there exists a unique path between $x$ and $y$.

(b) $w(T) = \sum_{e \in T} w(e)$ is minimal over all spanning trees. That is, $T$ is a minimal weight spanning tree.

Observations.

1. We use the term MST, or minimal spanning tree, to mean a minimal weight spanning tree.

2. Note similarity to the $G_{\pi}$-tree established by the breadth-first search algorithm, when $w(e) = 1$ for each $e \in E$.

3. Property (a) above is equivalent to saying that $T$ is acyclic. That is, $T$ contains no simple cycles.
Naive algorithm:

MST (G) {
    $A = \emptyset$;
    while there exists $e \in E \setminus A$ such that $A \cup \{e\}$ is acyclic
        $A = A \cup \{e\}$;
}

$G$ connected implies algorithm returns a spanning tree.

Minimal? Not likely.
Definition: Suppose $A \subseteq E$ is a set of edges and $A \subseteq T$, where $T$ is a minimal spanning tree for $[G = (V, E), w : E \rightarrow \mathcal{R}]$. If $e \in E$ satisfies $A \cup \{e\} \subseteq T'$, where $T'$ is a minimal spanning tree, then $e$ is safe for $A$.

Generic algorithm:

Generic-MST (G) {
    $A = \emptyset$;
    while $|A| < |V| - 1$ and there exists $e \in E \setminus A$ that is safe for $A$
    $A = A \cup \{e\}$;
}

Observations.

1. A tree with $V$ vertices contains exactly $V - 1$ edges. Any further edges must introduce a cycle.

2. Initialization and update maintain the loop invariant: $A \subseteq$ some MST. Consequently, the returned $A$ is a MST.

3. Algorithm is generic in the sense that the mechanism for selecting a safe edge is not specified.
**Definitions:** Assume \([G = (V, E), w : E \rightarrow \mathbb{R}]\) is a weighted, connected, undirected graph.

1. A **cut** is a partition \((S, V\setminus S)\)

2. An edge \((u, v)\) **crosses a cut** \((S, V\setminus S)\) if either \(u \in S, v \in V\setminus S\) or \(u \in V\setminus S, v \in S\).

3. A cut \((S, V\setminus S)\) **respects** \(A \subseteq E\) if \((u, v) \in A\) implies \((u, v)\) does not cross the cut.

4. \((u, v) \in E\) is a **light edge crossing a cut** if (a) \((u, v)\) crosses the cut, and (b) \(w(u, v) \leq w(x, y)\) for all edges \((x, y)\) that cross the cut.
Theorem 23.1: Let \( G = (V, E), w : E \to \mathbb{R} \) be a weighted, connected, undirected graph. Suppose \( A \subseteq E \) is strictly contained in some MST. Suppose also that \((S, V \setminus S)\) is a cut that respects \( A \) and that \((u, v)\) is a light edge crossing that cut. Then \((u, v)\) is safe for \( A \).

Proof:

\( A \): dark edges

\( T \), a minimal spanning tree, dark and light edges, excluding \((u, v)\). Note \( A \subseteq T \).

Cut respects \( A \)

\((u, v)\) is a light edge crossing the cut.

\( u \) on one side of cut, \( v \) on the other means that \( T \) must contain an edge crossing the cut, say \((x, y)\).

Can get new tree, \( T' \), by replacing \((x, y)\) with \((u, v)\). Removing \((x, y)\) severs \( T \) into two components; adding \((u, v)\) then rejoins those components, so cannot introduce a cycle.

Since \((u, v)\) is a light edge crossing the cut, we have \( w(u, v) \leq w(x, y) \). So, \( w(T') \leq w(T) \). Since \( T \) is a minimal spanning tree, \( w(T') = w(T) \) and \( T' \) is also a minimal spanning tree. \( A' = A \cup (u, v) \subseteq T' \), so \((u, v)\) is safe for \( A \).
Observations.

Generic-MST (G) {
    \[ A = \emptyset; \]
    while \(|A| < |V| - 1 \) and there exists \( e \in E \setminus A \) that is safe for \( A \)
    \[ A = A \cup \{e\}; \]
}

1. As generic algorithm adds safe edges to an initially empty set \( A \), the graph \( G_A(V,A) \) is at all times a forest. Since the growing \( A \) remains inside some MST at all times, it never contains a cycle.

2. Each safe edge chosen expands \( A \) by one edge, connects two of the forest trees, and thereby reduces the number of trees by one.

3. After choosing \( V - 1 \) safe edges, the algorithm terminates with minimum spanning tree.

**Corollary 23.2:** Let \([G = (V,E), w : E \rightarrow \mathcal{R}]\) be a weighted, connected, undirected graph. Suppose \( A \subseteq E \) is strictly contained in some MST. Let \( C = (V_c, E_c) \) be one of the trees in forest \( G_A = (V,A) \), and let \((u,v)\) be a light edge connecting \( C \) to another tree in \( G_A \). Then \((u,v)\) is safe for \( A \).

Proof: The cut \((V_c, V \setminus V_c)\) respects \( A \), and \((u,v)\) is a light edge crossing this cut. By Theorem 23.1, \((u,v)\) is safe for \( A \).
Implementations:

1. **Kruskal**: Add the lightest edge that does not introduce a cycle. Necessarily that edge connects two components in the current forest. A becomes a single tree as each iteration reduces the number of trees by one.

2. **Prim**: Start a tree with a source vertex and no edges. Add the lightest edge from the evolving tree that does not introduce a cycle. A remains a single tree at all times.

**Kruskal analysis** requires an excerpt from Data Structures for Disjoint Sets (Chapter 21).

Set representation as a linked list.

![Diagram of set representation as a linked list]

Operations:

1. **Make-Set(x)**: create a linked-list structure containing the singleton element x. Make-Set is a $\Theta(1)$ operation.

2. **Find-Set(x)**: find the set containing element x. The set’s identifier is the first element on its linked list. Find-Set is a $\Theta(1)$ operation.

3. **Union(x, y)**: create a new set containing the union of sets containing elements x and y.
Algorithms for Union(x, y).

**Naive algorithm.**

1. Add linked list of second argument to that of the first.
2. Update the size value of the first argument.
3. Update the end pointer of the first argument.
4. Update the head pointers of links originally in the second argument.
5. Destroy the second header.

All of the above are $\Theta(1)$ operations, except the head pointer updates.
Consider the following scenario.

```
Make-Set(x_1);
for i = 2 to n {
    Make-Set(x_i);
    Union(x_i, x_1);
}
```

The \( i^{\text{th}} \) iteration updates head pointer for all elements of the \( x_1 \)-list, which have been transferred to the end of the singleton list \( x_i \).

For \( i = 2 \): 1 head-pointer is updated.

For \( i = 3 \): 2 head-pointers are updated.

For \( i = n \): \((n-1)\) head-pointers are updated.

Total updates: \( \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = n(n-1)/2 \), a \( \Theta(n^2) \) operation that amortizes to \( \Theta(n) \) per union operation.
**Weighted union heuristic algorithm:** append the shorter list to the end of the longer.

**Theorem 21.1:** Using the weighted union heuristic algorithm, a sequence of \( m \) operations, of which \( n \) are Make-Set(), requires \( O(m + n \lg n) \) total time.

**Proof:** There are at most \( n \) elements in the union of all sets, say \( \{x_1, \ldots, x_n\} \). For a given element \( x \), we count the number of times that its head pointer is updated. In each such case, \( x \) must lie in the argument of shorter length in a Union\((x, y)\) call.

In the first such call, the length of the list containing \( x \) is at least one, which implies that the result list, containing \( x \), has length at least 2.

In the second call, the length of the list containing \( x \) is at least 2, which implies that the result list, containing \( x \), has length at least 4.

In call \( k \), the length of the list containing \( x \) is at least \( 2^{k-1} \) and the result list has length at least \( 2^k \).

Since \( 2^k \leq n \), we conclude \( k \leq \lg n \). That is, element \( x \) has its head-pointer updated at most \( \lg n \) times.

Consequently, the total number of head-pointer updates is at most \( n \lg n \).

As Make-Set, Find-Set, and the activity of Union beyond the head-pointer updates are all \( \Theta(1) \) operations, we compute a total time complexity of \( O(m + n \lg n) \). ★
Kruskal’s Algorithm

\[
\text{MST-Kruskal}(G = (V, E), w : E \rightarrow \mathcal{R}) \{ \\
(1) \quad A = \emptyset; \\
(V + 1) \quad \text{for } v \in V \quad \text{MakeSet}(v); \\
(O(E \lg E)) \quad \text{sort } E \text{ on increasing } w(e) \text{ obtaining } \{e_1, e_2, \ldots, e_{|E|}\}; \\
(E + 1) \quad \text{for } i = 1 \text{ to } |E| \{ \\
(E) \quad (u, v) = e_i; \\
(2E) \quad \text{if } \text{Find-Set}(u) \neq \text{Find-Set}(v) \{ \\
(V - 1) \quad A = A \cup \{(u, v)\}; \\
(V - 1) \quad \text{Union}(u, v); \\
\} \\
\} \\
\text{return } A; \\
\}
\]

Correctness: By Corollary 23.2, each \((u, v)\) added to \(A\) is safe for \(A\).

Complexity:

Set operations: \( V \) Make-set, \( 2E \) Find-set, \((V - 1)\) Union imply \( n = V, m = V + 2E + V - 1 \) in Theorem 21.1. The total count for set operations is then bounded above by \( V + 2E + (V - 1) + V \lg V < 2V + 2E + V \lg V \), which implies \( \Theta(V + E + V \lg V) \). Since a connected graph must have \( E \geq V - 1 \), we have \( E = \Omega(V) \) and the operation count for set operations is then \( O(E + V \lg V) \).

The setup prior to the sort requires \( 2V + 2 \) operations, and the sort itself contributes an additional \( O(E \lg E) \) operations.

Excluding the set operations, the edge loop adds \( 3E + 2V - 1 \) operations.

The total is then \( O(E + V \lg V + E \lg E + 3E + 4V + 1) = O(E \lg E) \), since, again, \( E = \Omega(V) \).

Since \( E \leq \binom{V}{2} < V^2 \), we have \( \lg E \leq 2 \lg V \), which finalizes the total at \( O(E \lg V) \).
Prim’s Algorithm

Prim’s algorithm uses a minHeap, $Q$, to hold vertices not yet added to the evolving MST. The key is the minimal distance from the vertex to the tree.

\[
\text{MST-Prim}(G = (V, E), w : E \rightarrow \mathbb{R}, r \in V) \{
\]

\[
(V + 1) \quad \text{for } u \in V \{ \\
(V) \quad u.\text{key} = \infty; \\
(V) \quad u.\pi = \text{null}; \\
(V) \quad u.\text{inQueue} = \text{true}; \\
\}
\]

\[
(1) \quad r.\text{key} = 0; \\
(O(V)) \quad Q \leftarrow V; \\
(V + 1) \quad \text{while } Q \neq \phi \{ \\
(V \cdot O(\log V)) \quad u = Q.\text{extractMin}(); \\
(V) \quad u.\text{inQueue} = \text{false}; \\
(V + E) \quad \text{for } v \in u.\text{adj} \\
(E) \quad \text{if } v.\text{inQueue} \text{ and } w(u, v) < v.\text{key} \{ \\
(O(E)) \quad v.\text{key} = w(u, v); \\
(O(E)) \quad v.\pi = u; \\
(O(E \log V)) \quad Q.\text{decreaseKey}(v); \\
\}
\}
\]

Complexity:

Initialization (Setup and heap creation): $4V + 2 + O(V) = O(V)$.

While-loop: $O(V \log V + E \log E)$, broken out as follows.

Loop-control: $V$ successful tests plus 1 failure.

Extraction: $V$ extractions at $O(\log V)$ each.
Embedded for-loop test: $E$ successful tests, one for each edge, and $V$ failures, one for each distinct adjacency list.

If-test: $E$ tests, which implies at most $E$ executions of the body, in which each decrease key operation requires $O(lg V)$.

$O(V lg V + E lg V)$.

Total: $O(V) + O(V lg V + E lg V) = O(V lg V + E lg V)$.

Connected graph implies $E \geq V - 1$, implying $E = \Omega(V)$.

The total then simplifies to $O(E lg V)$ – same as Kruskal.

**Correctness:**

During execution, $Q$ contains those nodes not yet added to the growing tree rooted at $r$. The attribute $u$.key maintains the distance of $u$ from the growing tree. This quantity is initially infinite for all nodes except the root $r$.

The only opportunity for the distance from the tree to node $v \in Q$ to change occurs when $v$ appears on the adjacency list of the node most recently added to the tree. That is, assuming that $v$.key is correct before $u$ joins the tree, the only possibility of a yet shorter path uses edge $(u, v)$. Thus the code correctly maintains the correct distances, as well as the correct $v.\pi$ entries reflecting the edge $(u, v) = (v.\pi, v)$ in the evolving tree.