Minimum Spanning Trees

Reading: CLR Sections 5.4 -- 5.5; Chapter 24

General Graph Terminology

A graph is a pair \((V,E)\) of vertices \(V\) and edges \(E \subseteq V \times V\) (all pairs of vertices). We only consider graphs where there is at most one edge between any two vertices. For any vertex \(v\), a possible edge is \((v,v)\) (a self-edge).

The adjacency list \(\text{Adj}[v]\) of a vertex \(v\) in \(G\) is the set of all edges of the form \((v,w)\) in \(G\).

A graph is directed if the each edge \((a,b)\) is interpreted as going from \(a\) to \(b\). It is undirected if \((a,b)\) and \((b,a)\) are considered equivalent edges.

A path in a graph \((V,E)\) is a sequence of vertices \(<v_0, v_1, ..., v_n>\) such that each \(v_i \in V\) and each \((v_{i-1}, v_i) \in E\). Such a path has length \(n\). The singleton sequence \((v_0)\) is a length 0 path. The sequence \(,<v,v>\) is only a path if \(E\) contains the self-edge \((v,v)\).

A path \(<v_0, v_1, ..., v_n>\) is a cycle if \(n \geq 1\) and \(v_0 = v_n\). A cycle is simple if \(v_1, ..., v_n\) are distinct and no edge is repeated. (The latter condition prevents \(<a,b,a>\) from being considered a cycle in an undirected graph.) A graph is acyclic if it contains no simple cycles.

An undirected graph is connected if there is a path between any two vertices.

A directed acyclic graph is a DAG. A connected acyclic undirected graph is a tree. An acyclic undirected graph is a forest.

A subgraph of \(G = (V,E)\) is a graph \(G' = (V',E')\) where \(V' \subseteq V\) and \(E' \subseteq E\).
Minimum Spanning Trees

A **weighted graph** is a graph \((V, E)\) together with a weighting function \(w: E \rightarrow \text{Real}\).

The **weight** of a weighted graph is \(\sum_{e \in E} w(e)\).

A **spanning tree** of a graph \((V, E)\) is a tree \((V, E')\) where \(E' \subseteq E\).

A **sub-spanning tree** (my terminology) of \((V, E)\) is a tree \((V', E')\) where \(V' \subseteq V\) and \(E' \subseteq E\).

A **minimum (weight) spanning tree (MST)** of a connected, undirected graph \(G\) is a spanning tree of \(G\) with minimal weight. (It may not be unique.)

**Skeleton of Greedy MST Algorithm**

The following is the skeleton of a greedy MST algorithm. The skeleton can be instantiated to both Prim's algorithm and Kruskal's algorithm, discussed later.

**Idea:** Grow a set of edges \(ST\) that is a subset of a spanning tree of \(G\). At each step, extend \(ST\) by the "best" safe edge -- i.e., the "best" edge that maintains the invariant that \(ST\) is a subset of a minimum spanning tree of \(G\).

\[
\begin{align*}
\text{MST}(G, w) & \quad ST \leftarrow \{} \\
D & \leftarrow \text{Init-Data}(G) \quad \text{\{}Initialize auxiliary data structure \(D\).\} \\
\textbf{while not} & \text{ Is-Spanning-Tree?}(ST, D) \quad \text{\{}Invariant: \(ST\) is the subset of a spanning tree.\} \\
& \quad (a,b) \leftarrow \text{Find-Safe-Edge}(ST, w, D) \\
& \quad ST \leftarrow ST \cup \{(a, b)\} \\
\textbf{return} & \quad ST
\end{align*}
\]

**Note:** The spanning tree is represented by the edge set \(ST\), from which the vertices can be unambiguously derived.
Prim's Algorithm

Idea: Grow a single sub-spanning tree of G. At each step, add the least weight edge connecting a vertex not in the tree with a vertex in the tree. In this case, the auxiliary data structure Dis a priority queue of vertices ordered by the weight of their minimal edge to a vertex in the growing tree ST (or \(\infty\) if there is no such edge).

**Init-Data(G)**

\[
\begin{align*}
\text{root} &\leftarrow \text{Choose-Root(vertices(G))} \quad \{\text{Choose an arbitrary vertex as root.}\} \\
\text{for } v \text{ in vertices(G) do} & \\
\text{min-weight}[v] &\leftarrow \infty \\
\text{min-parent}[v] &\leftarrow \text{nil} \\
\text{in-tree?}[v] &\leftarrow \text{false} \\
\text{min-weight[root]} &\leftarrow 0 \\
Q &\leftarrow \text{Build-PQ(vertices(G))} \quad \{\text{Priority queue ordered by min-weight.}\} \\
\text{Find-Min-Vertex}([], w, Q) &\{\text{Removes root from Q and establishes ordering on remaining vertices.}\} \\
\text{return}\ Q
\end{align*}
\]

**Is-Spanning-Tree?(ST, Q)**

\[\text{return PQ-Empty?(Q)}\]

**Find-Safe-Edge(ST, w, Q)**

\[
\begin{align*}
v &\leftarrow \text{Find-Min-Vertex(ST,w,Q)} \\
\text{return } (\text{min-parent}[v], v)
\end{align*}
\]

{\text{Returns the vertex with the minimum weight edge to an element of ST. This function does not reference ST directly; instead, it uses the info about ST cached in the fields in-tree, min-weight, and min-parent.)}

**Find-Min-Vertex(ST, w, Q)**

\[
\begin{align*}
a &\leftarrow \text{PQ-Extract-Min(Q)} \\
\text{in-tree?}[a] &\leftarrow \text{true} \\
\text{for } b \text{ in Adj}[a] \text{ do} & \\
\text{if not in-tree?}[b] \text{ and } w(a,b) < \text{min-weight}[b] \text{ then} & \\
\text{min-weight}[b] &\leftarrow w(a,b) \\
\text{min-parent}[b] &\leftarrow a \\
\text{PQ-Decrease-Key}(Q, b, w(a,b)) &\{\text{"Bubble up" b in heap by new weight.}\}
\end{align*}
\]

**Analysis:**

- Build-PQ called on \(|V|\) vertices
- PQ-Extract-Min called once for each of \(|V|\) vertices
- PQ-Decrease-Key called at most once for each of \(|E|\) edges

| Priority Queue implementation | Build-PQ | \(|V|\)-PQ-Extract-Min | \(|E|\)-PQ-Decrease-Key | Total |
|--------------------------------|----------|------------------------|------------------------|-------|
| unsorted array/list            | \(O(V)\) | \(O(V^2)\)             | \(O(E)\)               | \(O(V^2)\) |
| binary heap                     | \(O(V)\) | \(O(V \cdot \lg(V))\) | \(O(E \cdot \lg(V))\) | \(O(E \cdot \lg(V))\) |
| Fibonacci heap                  | \(O(V)\) | \(O(V \cdot \lg(V))\) | \(O(E)\)               | \(O(V \cdot \lg(V) + E)\) |

**Note:** A binary heap is the heap we know and love from CLR Chapter 7. A Fibonacci heap (CLR Chapter 21) is a heap-like data structure in which PQ-Extract-Min takes \(O(\lg(n))\) amortized cost for \(n\) nodes, and PQ-Decrease-Key takes \(O(1)\) amortized cost for \(n\) nodes.
### Kruskal's Algorithm

**Idea:** Grow a spanning forest --- a set of sub-spanning trees whose vertex sets are disjoint. Initially, each vertex is a trivial sub-spanning tree. At each step, add the minimum-weight edge between two distinct sub-spanning trees. This "glues" the two trees into a single tree. Eventually there will be a single spanning tree.

In this case, the auxiliary data structure \( D \) is a pair of (1) as-yet unprocessed edges of \( G \), sorted by increasing weight; and (2) a partition of the vertices in \( G \), where each set in the partition has the vertices in one tree of the current forest.

```plaintext
Init-Data(G)
    sorted-edges ← sort(edges[G])) {sorted by increasing weight}
    partitions ← {}
    for v in vertices(G) do
        partitions ← partitions U Singleton-Partition(v)
    return <sorted-edges, partitions> {returns a pair}

Is-Spanning-Tree?(ST, <edges, partitions>)
    return Is-Singleton?(partitions)

Find-Safe-Edge(ST, w, <edges, partitions>)
    (a, b) ← Least(sorted-edges) {first element of sorted edges}
    if not Same-Partition?(partition(a), partition(b)) then
        partitions ← Union-Partitions(a, b, partitions)
        return (a, b)
    else
        return Find-Safe-Edge(ST, w, <edges - (a,b), partitions>)
```

**Analysis:**

- **Initialization:** (1) sorting edges: \( O(E \lg(E)) \) (2) initializing partitions: \( O(V) \). In a connected graph, \( O(V) \leq O(E) < O(E \lg(E)) \).

- At most \(|E|\) calls to Same-Partition and Union, each of which costs \( O(\lg(E)) \). This gives a total time of \( O(E \lg(E)) \). (Actually, the time per operation is the inverse Ackerman function of \( E \) and \( V \), which grows far more slowly than a logarithm.)

- **Total:** \( O(E \lg(E)) \), which = \( O(E \lg(V)) \) since \( E = O(V^2) \) and \( O(\lg(V^2)) = O(\lg(V)) \)