Greedy Algorithms: Minimum Spanning Trees (cont.)

Last Time:
Kruskal's Algorithm

Initialize \( T = (V, \emptyset) \)

Examine edges of \( E \) in increasing weight order:
  - If edge connects two unconnected components then add edge to \( T \)
  - Else discard edge and continue (forms cycle)

---

Pseudocode:

\[
\text{MST- Kruskal} \ (G, w) \\
T \leftarrow \emptyset \\
\text{for each vertex } v \in V[G] \\
\quad \text{do } \text{Make-Set}(v) \\
\text{sort edges } E \text{ into non-decreasing order by } w \\
\]

\[\rightarrow O(1)\]

\[\rightarrow V \cdot T_{\text{Make-Set}}\]

\[\rightarrow O(E \log E)\]

\[
\text{for each edge } (u, v) \in E \text{ in non-decreasing order} \\
\quad \text{do if } \text{Find-Set}(u) \neq \text{Find-Set}(v) \\
\quad \quad \text{then } T \leftarrow T \cup \{(u, v)\} \\
\text{return } T
\]

\[\rightarrow (2E^2 + O(E) + E^2) \text{ Union} \]

\[\text{Find-Set} \]
Disjoint Set Data Structures

Want to maintain a dynamic collection of pairwise disjoint sets \( S = \{ S_1, S_2, S_3, \ldots, S_r \} \) in which each set \( S_i \) has one representative element, \( \text{rep}[S_i] \).

Supported Operations

Make-Set (\( u \)): Create new set containing single element \( u \)
- \( u \) must not be a member of any pre-existing set
- \( u \) is the representative

Union (\( u, v \)): Replace \( S_u \) and \( S_v \) with \( S_u \cup S_v \) in \( S \)
for any \( u, v \) in distinct sets \( S_u, S_v \). Update representative.

Find-Set (\( u \)): Return representative \( \text{rep}[S_u] \) of
set \( S_u \) containing \( u \).

Note: \( S_u \) means the set containing element \( u \), independently of whether \( u \) is the representative, here.
Solution 1: Doubly-linked list (unordered)

\[ S_i: \]

Ops
- Make-Set \((u)\) — initializes as lone node: \(\Theta(1)\)
- Find-Set \((u)\) — walks 'left' from \(u\) to rep. at head: \(\Theta(n)\)
- Union \((u,v)\) — concatenates lists containing \(u\) and \(v\), leaving front as rep.: \(\Theta(n)\)

Walk to tail of one and head of other; insert joining pointers.

Note that the representative is automatically updated to the set whose head wasn't changed.

Solution 2: Simple balanced tree \(\rightarrow\) forest

\[ S_i \]

Ops
- Make-Set \((u)\) — init. new tree with root node \(u\): \(\Theta(1)\)
- Find-Set \((u)\) — walk up tree from \(u\) to root:
- Union \((u,v)\) — concatenates trees containing \(u\) and \(v\) with overall root as rep.

\(\Theta(h, h + 1)\)

Because we only need these data structures to support so few ops, improvements can be made leading to substantially better running times, approaching \(\Theta(1)\), but not quite constant.
Running Time Analysis for Kruskal

Initialize: $O(1) + V \times \text{Make-Set} + O(E \log E)$

Loop:

$2E \times \text{Find-Set} + O(E) + E \times \text{Tunion}$

$O(E \log E)$

Note that there are 3 $O(E \log E)$ terms. Speeding up the $\text{Find-Set}$ and $\text{Tunion}$ ops wouldn't improve asymptotic runtime behavior.

$O(E \log E) + 2O(E \log V)$

$L \leq N/2 \Rightarrow \log L \leq \log N/2$  \[ L \leq N/2 \Rightarrow \log L \leq \log N/2 \]

Running Time: $O(E \log V)$

This algorithm, Kruskal, progressively adds edges without regard to whether they connect to previously accepted edges. Now will look at a different algorithm that progressively adds to a single tree.

From last time:  \[ \text{THE MST PROPERTY} \]

A light edge crossing a cut on a set of vertices is contained in an MST.

A proper subset is not the full set.  \[ \text{Partition.} \]

Jim away next week.
**Theorem 23.1**

- $G = (V, E)$ is a connected, undirected graph with a real-valued weight function $w$ on $E$.
- $T$ is a subset of $E$ included in some MST for $G$.
- $(U, V-U)$ is a cut of $G$ that respects $T$.
- $(u,v)$ is a light edge crossing the cut.

$\Rightarrow$ Edge $(u,v)$ is "safe" for $T$.

\[ \text{meaning the edge can be added to } T \text{ and the new set of edges is still a subset of some MST of } G \]

\[ \text{If } (u,v) \text{ is part of } T^* \text{ we are done.} \]

Assume $(u,v)$ is not part of $T$, an MST that includes $T$.

By cut-and-paste, construct new MST that contains $(u,v)$ and has weight at least as low as $T$ (call it $T'$).

- $P$ is path from $u$ to $v$ on MST $T^*$.
- Adding $(u,v)$ creates a cycle with $P$.
- At least one edge in $T^*$ is in cycle and crosses the cut because $u$ and $v$ are on opposite sides. Call one such edge $(x,y)$.

Thus, $T'$ is also an MST.

- $w(T') = w(T^*) - w(x,y) + w(u,v) \leq w(T^*)$.
CLRS Corollary 23.2

Same setup as Theorem 23.1

Plus

- \( C = (V_C, E_C) \) is connected component (tree) in \( G_T(V, T) \)
- \((u, v)\) is light edge connecting \( C \) to some other component in \( G_T \), then \((u, v)\) is safe for \( T \).

- Make the cut \((V_C, V - V_C)\)
  - Cut respects \( T \)
  - \((u, v)\) is light edge for this cut

\[ \Rightarrow (u, v) \text{ is safe for } T \text{ by Theorem 23.1} \]

**Prim's Algorithm**

Select a vertex to start, \( s \)

\[ T \leftarrow s \]

Until done

- Select light edge connecting \( T \) to an isolated vertex, \((u, v)\)

\[ T \leftarrow T \cup (u, v) \]
Proof by Loop Invariant, as for Kruskal

Prior to each iteration, \( T \) is a subset of an MST

Initiation: \( T \) has no edges; trivially satisfied

Maintenance: The new edge added to \( T \) in the loop is a light edge on a cut respecting \( T \) and connecting it to another component, so by Corollary 23.2, it is safe for \( T \).
Add edge \((u,v)\) to \( T \) maintains a subset of an MST.

Termination: All vertices have been added to create a single connected component (tree) at termination, which is an MST.

The interesting part of the algorithm is "select light edge connecting \( T \) to an isolated vertex, \((u,v)\)"

Typically done with data structure called min-priority queue that keeps track of unattached vertices and lightest weight connecting each to current tree (weight attributes are "keys")

MIN-PRIORITY QUEUE: Supported Operations

\[
\text{INSERT} \ (S, x) \quad \text{INSERTS element } x \quad \text{into set of elements } S \\
\text{MINIMUM} \ (S) \quad \text{RETURNS element of } S \text{ with smallest key} \\
\text{EXTRACT-MIN} \ (S) \quad \text{REMOVES and returns element w/ smallest key} \\
\text{DECREASE-KEY} \ (S, x, k) \quad \text{DECREASE value of element } x \text{'s key to new value } k
\]
Implementation

**Algorithm:** MST-Prim (G, w, r)

1. Initialize:
   - S = φ
   - \( u \rightarrow \infty \)
   - \( u.\pi \rightarrow NIL \)
   - \( r.\text{key} = 0 \)
   - \( Q = G.\text{V} \)

2. **Initialize**:
   - \( V \) times
   - \( T_{\text{Extract-Min}} \) times total

3. **Extract-Min**:
   - While \( Q \neq \phi \)
     - \( u = \text{Extract-Min}(Q) \)
     - \( T_{\text{Decrease-Key}} \) times
     - For each \( v \in G.\text{Adj}[u] \)
       - If \( u.u.\pi \neq NIL \)
         - If \( v \in Q \) and \( w(u,v) < v.\text{key} \)
           - \( v.\pi = u \)
           - \( v.\text{key} = w(u,v) \)

4. **Decrease-Key**:

Running Time Analysis

\[ \text{Time} = |V| \cdot T_{\text{Extract-Min}} + O(E) \cdot T_{\text{Decrease-Key}} \]

Min-Priority Queue Data Structures

<table>
<thead>
<tr>
<th>Q</th>
<th>( T_{\text{Extract-Min}} )</th>
<th>( T_{\text{Decrease-Key}} )</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>( O(V) )</td>
<td>( O(1) )</td>
<td>( O(V^2) )</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>( O(lg \ V) )</td>
<td>( O(lg \ V) )</td>
<td>( O(E(lg \ V)) )</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>( O(lg \ V) )</td>
<td>( O(1) )</td>
<td>( O(E + V \cdot lg \ V) )</td>
</tr>
</tbody>
</table>

Best MST to date: 1993 Karger, Klein, Tarjan — \( O(V) \) time randomized
Greedy Strategies (CLRS 16.2)

General Approach

1) Structure the problem so that we make a choice and are left with one subproblem to solve.

2) Prove exists optimal soln to org problem that makes same greedy choice ("safe" choice)

3) Demonstrate optimal substructure.
   - After greedy choice, combine optimal solution of remaining subproblem, to give optimal soln to org problem.

Note that this sounds a lot like Dynamic Programming. In fact, there is an interesting relationship between the two.

Key Properties for Greedy Algorithm

1) Greedy-Choice Property — Locally optimal choice leads to globally optimal soln.

   At each local choice, we make the choice independently of the soln to the subproblem (Greedy). In dynamic programming, the local choice we make depends on the soln to the subproblem—solve bottom up.

   Kruskal and Prim can choose a safe edge without having examined the full problem.
Optimal Substructure

The optimal solution to a problem contains optimal solutions to subproblems

Both greedy and dynamic programming algorithms exploit (rely on) optimal substructure

Prim produces (optimal) MSTs on subsets of \( V \) on the way to finding the full MST.