Greedy Algorithms

Part Three

Announcements

- Problem Set Four due right now.
  - Due on Wednesday with a late day.
- Problem Set Five out, due Monday, August 5.
  - Explore greedy algorithms, exchange arguments, “greedy stays ahead,” and more!
  - ***Start early***. Greedy algorithms are tricky to design and the correctness proofs are challenging.
- Handout: “Guide to Greedy Algorithms” also available.
- Problem Set Three graded; will be returned at the end of lecture.
  - Sorry for the mixup from last time!

Outline for Today

- **Implementing Prim's Algorithm**
  - Efficiently finding MSTs.
- **Kruskal's Algorithm**
  - A different algorithm for finding MSTs.
- **Disjoint-Set Forests**
  - A specialized data structure for speeding up Kruskal's algorithm.

Recap: **Prim's Algorithm**

Prim's Algorithm

- **Prim's Algorithm** is the following:
  - Choose some $v \in V$ and let $S = \{v\}$.
  - Let $T = \emptyset$.
  - While $S \neq V$:
    - Choose a least-cost edge e with one endpoint in S and one endpoint in $V - S$.
    - Add e to T.
    - Add both endpoints of e to S.
- Naive implementation takes time O(mn).

A Faster Implementation

- Can speed up using binary heaps:
  - Create a priority queue initially holding all edges incident to v.
  - At each step, dequeue edges from the priority queue until we find an edge (x, y) where $x \in S$ and $y \notin S$.
  - Add (x, y) to T.
  - Add to the queue all edges incident to y whose endpoints aren't in S.
- Each edge is enqueued and dequeued at most once. *(Why?)*
- Total runtime: **$O(m \log m)$**.

A Note on Runtimes

- In any graph, $m = O(n^2)$.
- Therefore:

$$\begin{aligned} O(m \log m) &= O(m \log (n^2)) \\ &= \mathbf{O(\boldsymbol{m} \log \boldsymbol{n})} \end{aligned}$$

- This version is more common and we will use it going forward.

A Different Approach:
Kruskal's Algorithm

Kruskal's Algorithm

- **Kruskal's Algorithm** is the following:
  - Let $T = \emptyset$.
  - For each edge (u, v) sorted by cost:
    - If u and v are not already connected in T, add (u, v) to T.
- Can prove by induction that the result is a spanning tree by showing that
  - Exactly $n - 1$ edges are added.
  - No edges are added that close a cycle.

Showing Correctness

- The correctness proof for Kruskal's algorithm uses an exchange argument similar to that for Prim's algorithm.
- **Recall:** Prove Prim's algorithm is correct by looking at cuts in the graph:
  - Can swap an edge added by Prim's for a specially-chosen edge crossing some cut.
  - Since that edge is the lowest-cost edge crossing the cut, this cannot increase the cost.

Correctness Proof Intuition

- **Claim:** Every edge added by Kruskal's algorithm is a least-cost edge crossing some cut (S, V – S).
  - When the edge was chosen, it did not close a cycle.
  - Choose S to be the CC of nodes on one end of the edge to get cut (S, V – S).
  - Edge must be cheapest edge crossing this cut, since otherwise we would have selected a different edge.

Theorem: Kruskal's algorithm always produces an MST.

Proof: Let T be the tree produced by Kruskal's algorithm and T^* be an MST. We will prove $c(T) = c(T^*)$. If $T = T^*$, we are done. Otherwise $T \neq T^*$, so $T - T^* \neq \emptyset$. Let (u, v) be an edge in $T - T^*$.

Let S be the CC containing u at the time (u, v) was added to T. We claim (u, v) is a least-cost edge crossing cut $(S, V - S)$. First, (u, v) crosses the cut, since u and v were not connected when Kruskal's algorithm selected (u, v). Next, if there were a lower-cost edge e crossing the cut, e would connect two nodes that were not connected. Thus, Kruskal's algorithm would have selected e instead of (u, v), a contradiction.

Since T^* is an MST, there is a path from u to v in T^*. The path begins in S and ends in $V - S$, so it contains an edge (x, y) crossing the cut. Then $T^{*\prime} = T^* \cup \{(u, v)\} - \{(x, y)\}$ is an ST of G and $c(T^{*\prime}) = c(T^*) + c(u, v) - c(x, y)$. Since $c(x, y) \geq c(u, v)$, we have $c(T^{*\prime}) \leq c(T^*)$. Since T^* is an MST, $c(T^{*\prime}) = c(T^*)$.

Note that $|T - T^{*\prime}| = |T - T^*| - 1$. Therefore, if we repeat this process once for each edge in $T - T^*$, we will have converted T^* into T while preserving $c(T^*)$. Thus $c(T) = c(T^*)$. ■

Implementing Kruskal's Algorithm

Kruskal's Algorithm

- High-level overview of Kruskal's algorithm:
  - Let $T = \emptyset$.
  - For each edge (u, v) sorted by cost:
    - If u and v are not connected by T, add (u, v) to T.
- Can visit edges in order by sorting them in time $O(m \log n)$.
- Can check whether u and v are connected in time $O(n)$ by doing DFS. *(Why?)*
- Total time required: $\mathbf{O(\boldsymbol{mn})}$.

Speeding up Kruskal's

- The “bottleneck” in Kruskal's algorithm is checking whether a pair of nodes are connected to one another.
- **Goal:** Optimize queries of the form “are *x* and *y* connected?”
- To do this, we will introduce a new data structure called the *disjoint-set forest*.

Set Partitions

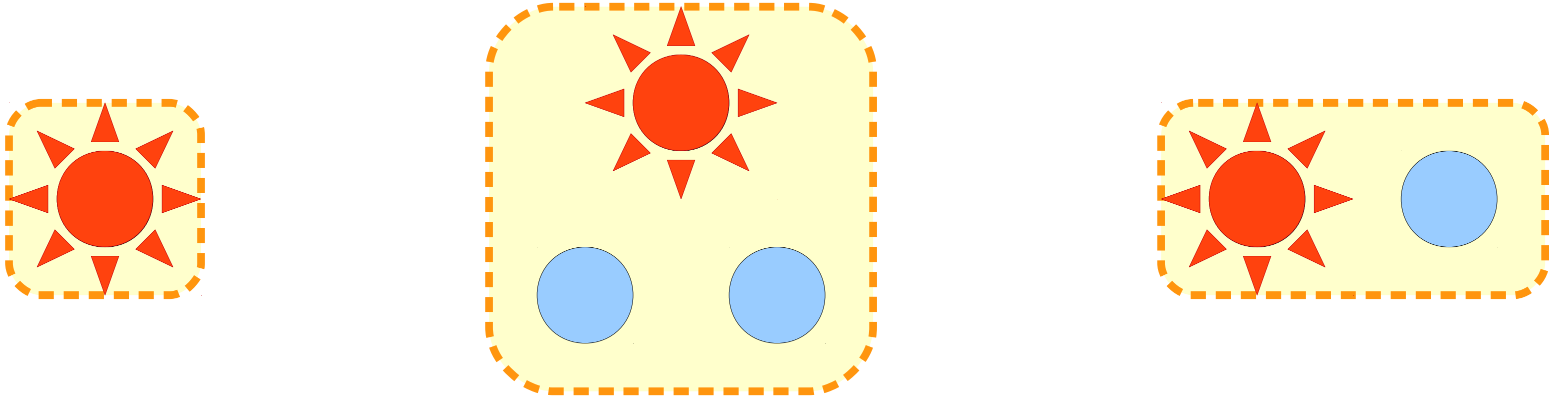
- A **partition** of a set S is a family X of nonempty sets where each element of S belongs to exactly one set in X.
- **Goal:** Build a data structure (called a *disjoint-set data structure*) that efficiently supports three operations:
  - **make-set(*v*)**, which places v into its own set,
  - **union(*u*, *v*)**, which combines the sets containing u and v into one set, and
  - **in-same(*u*, *v*)**, which returns whether u and v belong to the same set.

Kruskal's Algorithm

- Using our new data structure:
  - Let $T = \emptyset$.
  - Let S be a disjoint-set data structure.
  - For each $v \in V$:
    - Call S.make-set(v)
  - For each edge (u, v) sorted by cost:
    - If S.in-same(u, v) is false:
      - Add (u, v) to T.
      - Call S.union(u, v).

Representatives

- Given a partition of a set *S*, we can choose one **representative** from each of the sets in the partition.
- Representatives give a simple proxy for which set an element belongs to: two elements are in the same set in the partition iff their set has the same representative.



Data Structure Idea

- **Idea:** Associate each element in a set with a representative from that set.
- To determine if two nodes are in the same set, check if they have the same representative.
- To link two sets together, change all elements of the two sets so they reference a single representative.

Using Representatives

- If there are *n* total elements, what is the cost of looking up a representative?
  - **O(1)**
- If there are *n* total elements, what is the cost of merging two sets together?
  - **O(*n*)**
- Can we improve this?

Hierarchical Representatives

- If there are n total elements, what is the cost of looking up a representative?
  - **O(n)**
- If there are n total elements, what is the cost of merging two sets together?
  - **O(n)**
- The inefficiency arises because the path from any node to its representative can be very large.
- Can we fix that?

Union by Size

- **Idea:** Store in each node the number of nodes that count it as a representative.
- To merge the sets containing two nodes together:
  - Find the representatives of each.
  - Choose one of the representatives with the least number of nodes below it.
  - Set its representative to the other node.
  - Update the total number of nodes below the other node.

Analyzing Union by Size

- The runtime of these operations depends on the height of the trees formed this way.
- **Claim:** A tree with height k contains at least 2^k nodes.
- **Proof Idea:** Use induction.
  - Trees with height 0 start with $2^0 = 1$ nodes.
  - Merging two trees of unequal heights always results in a single tree of the height of the larger of the two.
  - Merging two trees of height k into a tree of height $k + 1$ results in a tree with at least $2 \cdot 2^k = 2^{k+1}$ nodes.
- **Corollary:** If there are n total nodes, all operations take $O(\log n)$ time.

Kruskal's Algorithm

- Using our new data structure:
  - Let $T = \emptyset$.
  - Let S be a disjoint-set data structure.
  - For each $v \in V$:
    - Call S.make-set(v)
  - For each edge (u, v) sorted by cost:
    - If S.in-same(u, v) is false:
      - Add (u, v) to T.
      - Call S.union(u, v).
- Total runtime: **$O(m \log n)$**.

Looking Forward

- It is possible to speed up our data structure by using two modifications:
  - **Path Compression:** After looking up a representative, change the pointers of all visited nodes to directly point to the representative.
  - **Union-by-Rank:** Link trees based on *height* rather than number of nodes.
- New runtime: m total operations takes time $O(m\ \alpha(m))$, where $\alpha(m)$ is a *ridiculously* slowly-growing function.

Next Time

- Dynamic Programming
- Purchasing Cell Towers
- A Different Approach to Recursion