Minimum Spanning Trees
- Kruskal’s Algorithm
- Prim’s Algorithm

Union-Find Algorithm

or

A Quick Tour of Chapters 21, and 23,
with hints of Chapters 17 and 20.

Minimum Spanning Tree Problem

Given a weighted connected graph \( G = (V,E) \), ...
   - For each edge \((u,v) \in E\), \(w(u,v)\) is its “weight”.

... find a spanning tree \((V,T)\) ...
   - \( T \) includes each node of \( V \).
   - So \( T \) has exactly \( |V| - 1 \) edges.

of minimum weight.
   - Weight \( w(T) \) of \( T \) is sum of weights of its edges.
   - Minimum means no spanning tree has lower weight.

“Size” of instance of MST includes both \(|V|\) and \(|E|\).

Note that \(|V| \leq |E| \leq |V|^2\).

Greedy approach works for MST

If we build up a spanning tree by repeatedly adding a lightest legal edge, it will be a MST.
- “Legal” means you don’t introduce any cycles.

Several ways “lightest” can be interpreted:

1. Lightest among all remaining edges.
   While building, you have a forest (a set of trees).
   This is Kruskal’s algorithm.

2. Lightest among all edges connected to what you have already.
   You keep adding to a single tree.
   This is Prim’s algorithm.

Why greedy approaches work

Thm: Given \( G = (V,E) \) and \( A \subseteq E \) such that there exists a MST \( T \) of \( G = (V,E) \) with \( A \subseteq T \).

“A is a subset of a MST”

Suppose also that \( V = V_1 \cup V_2 \) and \( V_1 \cap V_2 = \emptyset \),

“V is partitioned into \( V_1 \) and \( V_2 \)”

that no edge of \( A \) goes from \( V_1 \) to \( V_2 \),

“the partition respects \( A \)”

and that \( e \) is a lightest edge from \( V_1 \) to \( V_2 \).

Then \( A \cup \{e\} \) is a subset of some MST.

“Though not necessarily of \( T \).”

“...adding light edges to a minimum spanning forest is OK.”
Why greedy approach works

Proof: (That $A \cup \{e\}$ is a subset of some MST)

If $e \in T$, we're done. (Recall: $T$ was MST containing $A$.)

Otherwise, add $e$ to $T$. This creates a cycle including $e$.

(A tree on $|V|$ nodes can only have $|V|-1$ edges.)

The cycle must have another edge $e'$ going from $V_1$ to $V_2$.

Note that weight($e$) $\leq$ weight($e'$). ($e$ was a lightest edge)

Let $S = T \cup \{e\} - \{e'\}$. All nodes are still connected.

(If you needed $e'$ to go from $u$ to $v$ in $T$, now you can take the other way around the cycle.)

So $S$ is a spanning tree that’s not heavier than $T$.

Thus, $A \cup \{e\}$ is a subset of the MST $S$.

Kruskal’s algorithm

So named because Boruvka invented it in 1926

Sort edges by weight ($e_1 \leq e_2 \leq \ldots \leq e_{|E|}$);

$T = \phi$;

For $i = 1$ to $|E|$

If ($T \cup \{e_i\}$ is acyclic) $T = T \cup \{e_i\}$;

Takes $\Theta(|E| \lg |E|)$ time for sort, plus time for $|E|$ tests and $|V|$ “$T \cup \{e\}$” operations.

- If tests and unions take $\lg |E|$ time apiece or less, total time will be $\Theta(|E| \lg |E|)$
- Aside: $\Theta(|E| \lg |E|) = \Theta(|E| \lg |V|)$ (why??)
Digression: **Union-Find Problem**

For Kruskal’s algorithm, we need fast way to test if adding $e_i$ to $T$ creates a cycle.

At $i$th iteration, $T$ is a set of trees.

(Initially, each tree contains one node).

$e_i = (u,v)$ is OK unless $u$ and $v$ are in the same tree.

It would suffice if we could process requests:

- **Make-Set($u$)** - creates set containing $u$ (for initialization)
- **Find-Set($u$)** - returns representative element of $u$’s set
  
  If Find-Set($u$) = Find-Set($v$), we can’t add ($u$, $v$).
- **Union($u,v$)** - combine sets containing $u$ and $v$.
  
  Choose new representative.

---

**Algorithms for Union-Find**

**Approach 1:** Define $A[u] =$ representative of $u$.

- **Find-Set($u$):** return $A[u]$. Takes $O(1)$ time.
- **Union($u,v$):** Let $x = A[u]$, $y = A[v]$; change all $x$’s to $y$’s in $A$
  
  Takes $\Omega(|V|)$ time: too slow!

**Approach 2:** Above plus, for each set, a list of members.

- **Find-Set($u$):** return $A[u]$        Takes $O(1)$ time.
- **Union($u,v$):** for each member $z$ of $u$’s list, add it to $v$’s list and set $A[z] = y$. What is worst case?

Magic Bullet: move elements of smaller list to larger.

No element will be moved more than $\lg |V|$ times.

Example of amortized analysis: Even though Union may take $O(|V|)$ time, doing $|V|$ unions takes $O(|V| \lg |V|)$ (assuming we start from one-element sets).
Algorithms for Union-Find

  - Each set is a tree; each node points to parent.
  - Root has null pointer.
- Find-Set(u): follow pointers to tree’s root.
- Union(u,v): make u’s root point to v’s root (or vice versa)

Balancing: make smaller tree point to larger.
  - Worst case $O(\log |V|)$ per request

Path Compression: whenever you follow path to root, reassign all pointers to go directly to root.
  - Amortized cost $O(\log |V|)$ per request.

Balancing + Path Compression: Amortized inverse-Ackerman’s(\(|V|\)) per request (“almost constant”)

---

Prim’s algorithm

So named because Jarnik invented it in 1930.

Grow (single) tree from start node by adding lightest edge from tree node to non-tree node.

How do we find the lightest edge quickly?

“Obvious” method:
  - Keep a min-priority queue of all edges connected to tree (key is weight of edge).
  - When we add node to tree, add all its edges to the queue.
  - When we Extract an edge from the queue, check that only one endpoint is in tree; if so, add other node to tree.

Requires $|E|$ Insert’s and $|E|$ Extract-Min’s

Complexity is $\Theta( |E| \log |E| )$ (which is also $\Theta( |E| \log |V| )$)
**Prim's algorithm**

Better method to find the lightest edge quickly:
- Keep priority queue of *nodes*, with key being the weight of the lightest edge from the node to the tree.
  - Initialize the queue to all nodes with key $\infty$
- When we add node to tree, for each of its edges, do a Decrease-Key operation to other endpoint.
- Extract-Min tells node to add to tree.

$|V|$ Insert's, $|V|$ Extract-Min's, and $|E|$ Decrease-Key's.

Complexity is still $\Theta(|E| \lg |V|)$

So what?? Fibonacci Heaps take amortized $O(1)$ time for Decrease-Key and $O(\lg |V|)$ time for other operations.

So time is $O(E + |V| \lg |V|)$ (An improvement for dense graphs).