Goals

- Introduce the minimum spanning tree problem

Network Design Problem

- Given: an undirected graph \( G = (V, E) \) with edge costs (weights) \( c_e > 0 \). Assume for now that all edge weights are distinct.
- Find: subset of edges \( T \subseteq E \) such that \( (V, T) \) is connected and the total cost of edges in \( T \) is as small as possible.

Minimum Spanning Tree Problem

- Call \( T \subseteq E \) a spanning tree if \( (V, T) \) is a tree (connected, no cycles).
- Claim: in a minimum-cost solution, \( T \) is a spanning tree.
- Therefore, we call this the minimum spanning tree (MST) problem.

Goals

- State and prove the cut property for minimum spanning trees
Chapter 4 Greedy Algorithms

**Cuts**

- A key to understanding MSTs is a concept called a cut.

**Definition:** A cut is a partition of the nodes into two nonempty subsets \((S, V \setminus S)\).

**Definition:** Edge \(e = (v, w)\) crosses the cut if \(v \in S\) and \(w \in V \setminus S\).

**Proof of Cut Property**

- Let \(e = (v, w)\) be the min-wt edge across cut \((S, V \setminus S)\) and suppose for contradiction that \(T\) is MST but does not include \(e\).

**Cut Property**

- **Theorem (cut property):** Let \(e = (v, w)\) be the minimum-weight edge crossing cut \((S, V \setminus S)\) in \(G\). Then \(e\) belongs to every minimum spanning tree of \(G\).

  - **Terminology:**
    - \(e\) is the cheapest or lightest edge across the cut
    - It is safe to add \(e\) to a MST
  - We will see two different greedy algorithms based on the cut property: Kruskal’s algorithm and Prim’s algorithm.

**Proof of Cut Property**

- Let \(e = (v, w)\) be the min-wt edge across cut \((S, V \setminus S)\) and suppose for contradiction that \(T\) is MST but does not include \(e\).

  - There is a path from \(v\) to \(w\) in \(T\).
  - Let \(e' = (v', w')\) be the first edge on this path that crosses the cut.
  - Let \(T' = T + \{e\} - \{e'\}\).
  - \(T'\) is still a spanning tree:
    - Connected: any path in \(T\) that needed \(e'\) can be routed via \(e\) instead.
    - No cycles: adding \(e\) creates one cycle, removing \(e'\) destroys it.
  - But since \(e\) is the lightest edge from \(S\) to \(V \setminus S\),
    \[
    w(T') = w(T) - w(e') + w(e) < w(T)
    \]
    
    \(T\) is not MST.
Goals

- Use cut property to derive Kruskal's algorithm and prove it is correct

Kruskal's algorithm

Assume edges are numbered $e = 1, \ldots, m$
Sort edges by weight so $c_1 \leq c_2 \leq \ldots \leq c_m$
Initialize $T = \{}$
for $e = 1$ to $m$ do
  if adding $e$ to $T$ does not form a cycle then
    $T = T \cup \{e\}$
Exercise: argue correctness (use cut property)

Kruskal's algorithm proof

- Let $T$ be partial spanning tree just before adding $e = (u, v)$
- What cut can we use to prove that $e$ belongs to MST?
Kruskal’s algorithm proof
- Let $T$ be partial spanning tree just before adding $e = (u, v)$
- Let $S$ be the connected component of $T$ that contains $u$
- $e$ crosses $(S, V \setminus S)$, otherwise adding $e$ would create cycle
- No other edge crossing $(S, V \setminus S)$ has been considered yet; it could have been added without creating a cycle
- $e$ is the cheapest edge across $(S, V \setminus S)$
- $e$ belongs to every MST (cut property)
- Every edge added belongs to the MST
- By design, the algorithm creates no cycles and doesn’t stop until $(V, T)$ is connected
- $T$ is MST

Goals
- Use cut property to derive Prim’s algorithm and prove it is correct

Prim’s Algorithm
- What if we want to grow a tree as a single connected component starting from some vertex $s$?

Prim’s algorithm: Let $S$ be the component containing $s$. Add cheapest edge from $S$ to $V \setminus S.$

Prim’s Algorithm
- Initialize $T = \{\}$
- Initialize $S = \{s\}$
- while $|S| < n$
  - Let $e = (u, v)$ be the minimum-cost edge from $S$ to $V - S$
  - $T = T \cup \{e\}$
  - $S = S \cup \{v\}$
- Correctness? use cut property
Prim’s algorithm proof

- Let $T$ be the partial spanning tree just before adding edge $e$
  - Let $S$ be the connected component containing $s$
  - By construction, $e$ is the cheapest edge across the cut $(S, V - S)$
  - Therefore, $e$ belongs to every MST
- So, every edge added belongs to the MST.
- The algorithm creates no cycles and does not stop until the graph is connected, therefore, the final output is a spanning tree.
- The final output is a minimum-spanning tree.

Goals

- Remove distinctness assumption
- Give implementation of Prim’s algorithm and analyze its running-time

Remove Distinctness Assumption?

- Hack: break ties by perturbing each edge weight by a tiny unique amount.
- Implementation: break ties in an arbitrary but consistent way (e.g., lexicographic)
- (There is a more "elegant" way that requires a stronger cut property.)

Implementation of Prim’s algorithm

Initialize $T = \{\}$
Initialize $S = \{s\}$

while $T$ is not a spanning tree do
  Let $e = (u, v)$ be the minimum-cost edge from $S$ to $V - S$
  $T = T \cup \{e\}$
  $S = S \cup \{v\}$
  mark $v$ "attached"

What does this remind you of?
**Prim Implementation**

Set $A = V$  
Set $a(v) = \infty$ for all nodes  
Set $a(s) = 0$  
Set $\text{edgeTo}(s) = \text{null}$  

while $A$ not empty do

extract node $v \in A$ with smallest $a(v)$ value

Set $T = T \cup \text{edgeTo}(v)$

for all edges $(v, w)$ where $w \in A$ do

if $c(v, w) < a(w)$ then

$a(w) = c(v, w)$

$\text{edgeTo}(w) = (v, w)$

end if

end for

end while

Nearly identical to Dijkstra. Priority queue for $A \Rightarrow O(m \log n)$

**Goals**

- Describe implementation of Kruskal's algorithm

**Kruskal Implementation?**

Sort edges by weight so $c_1 \leq c_2 \leq \ldots \leq c_m$

Initialize $T = \{}$

for $e = 1$ to $m$ do

if adding $e = (u, v)$ to $T$ does not form a cycle then

$T = T \cup \{e\}$

end if

end for

Ideas? use BFS

$O(mn)$

$O(\text{edges of } T + \text{nodes of } T)$

$\Rightarrow O(n)$

**Kruskal Implementation: Union-Find**

Idea: use clever data structure to maintain connected components of growing spanning tree. Should support:

- $\text{find}(v)$: return name of set containing $v$
- $\text{Union}(A, B)$: merge two sets

for $e = 1$ to $m$ do

Let $u$ and $v$ be endpoints of $e$

if $\text{find}(u) \neq \text{find}(v)$ then

$T = T \cup \{e\}$

$\Rightarrow O(1)$

if $\text{find}(u) = \text{find}(v)$ then

$\text{Union}(\text{find}(u), \text{find}(v))$

$\Rightarrow O(1)$

end if

end if

end for

Goal: union = $O(1)$, find = $O(\log n)$ $\Rightarrow O(m \log n)$ overall
Goals

- Describe union-find data structure

Union-Find Data Structure

- Each set elects a representative to act as the “name” of the set
- Nodes point to their representative
- Initially, all nodes point to themselves

Union(e, f)
Union(c, d)
Union(d, f)
Union(b, f)
Union(a, f)

Time for union?

- Union(a, f): which pointer should be updated?
- **Convention**: smaller set updates its pointer
- Time for find? proportional to depth of tree
**Union-Find Data Structure**

**Claim:** let \( d = \text{depth} \) and \( k = \# \text{ nodes in set} \). Then \( d \leq \log_2(k) \).

\[ \implies \text{find is } O(\log n) \]

**Equivalent claim:** \( k \geq 2^d \)

**Proof:** by induction.

Base case: \( d = 0, k = 1 \)

**Induction Step**

Consider union of sets of size \( k_L < k_R \) with depths \( d_L \) and \( d_R \)

1. Case 1: depth same as right subtree; size of set increases
   \[ d = d_R \]
   \[ k = k_L + k_R \geq 2^{d_R} = 2^d \]

2. Case 2: depth one more than left subtree; size of set at least doubles
   \[ d = d_L + 1 \]
   \[ k = k_L + k_R \geq 2k_L \geq 2 \cdot 2^{d_L} = 2^{d_L+1} = 2^d \]

In both cases, \( k \geq 2^d \)

\[ d \leq \log_2(k) \implies \text{find is } O(\log n) \]

**Union-Find Wrap-Up**

- Union is \( O(1) \): update one pointer
- Find is \( O(\log n) \): follow at most \( \log_2(n) \) pointers to find representative of set
- \( m \) union/find operations takes \( O(m \log n) \) time
- Better: path compression \( \Rightarrow \) find in nearly constant time