COMPSCI 311: Introduction to Algorithms

Lecture 9: Minimum Spanning Trees

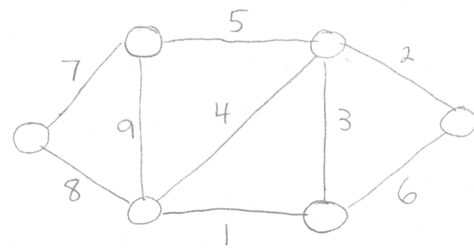
Marius Minea

University of Massachusetts Amherst

slides credit: Dan Sheldon

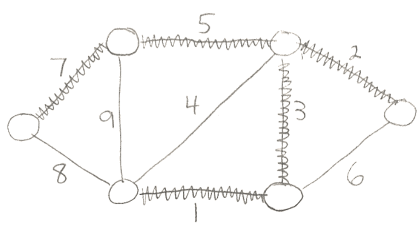
20 February 2019

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 the *minimum* possible

Minimum Spanning Tree Problem



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

Cuts

- A key to understanding MSTs is a concept called a cut.
- **Definition**: A cut in G is a partition of the nodes into two nonempty subsets $(S, V \setminus S)$.
- **Definition**: Edge $e = (v, w)$ crosses cut $(S, V \setminus S)$ if $v \in S$ and $w \in V \setminus S$.
  The cutset of a cut is the set of edges that cross the cut.

Note (see definition) *cut* means the partition (two node sets), *not* the edges cut by dividing the nodes (that's the *cutset*).

Minimum spanning trees: quiz 1

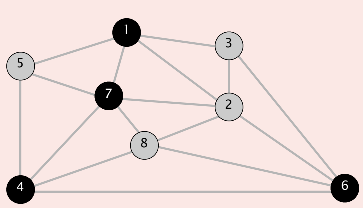
Consider the cut S = { 1, 4, 6, 7 }. Which edge is in the cutset of S?

A. S is not a cut (not connected)

B. 1–7

C. 5–7

D. 2–3



24

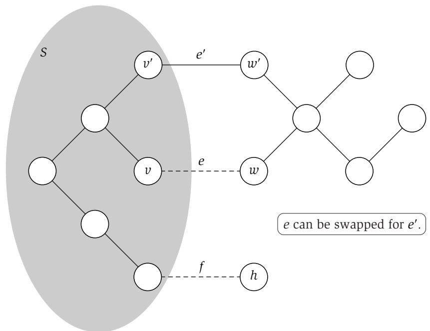
slide credit: Kevin Wayne / Pearson

Cut Property (IMPORTANT)

- **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
- 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



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 an 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 needs e' can now be routed via e
  - No cycles: adding e creates one cycle, removing e' destroys it
- But since e is the lightest edge from S to $V \setminus S$,

$$w(T') = w(T) - w(e') + w(e) < w(T)$$

What's wrong with the following proof ?

- Let T be a spanning tree so the cheapest edge e is not in T
- T must contain an edge f that links S to $V - S$
- e is the cheapest such edge, so $T - \{f\} \cup \{e\}$ is a cheaper tree.

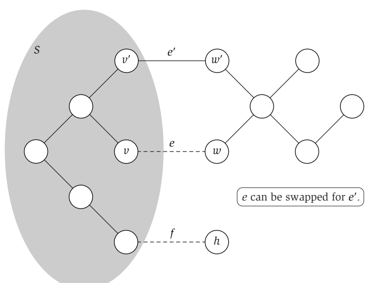


Figure: Kleinberg & Tardos

- $T - \{f\} \cup \{e\}$ may not be a tree!

Clicker Question: Properties of Spanning Trees

Which of the following is not true ?

A. A spanning tree contains at least one edge crossing any cut

B. The union of two different spanning trees produces a cycle

C. Swapping a tree edge with another edge from the same cutset may produce a cycle

D. Adding an edge to a spanning tree produces at least two cycles

Kruskal's algorithm

- Armed with the cut property, how can we find a MST?
  - Starting no edges, which edge do we add first? How can we prove it is safe to add?
  - What edge do we add next? How do we prove it is safe?
  - Next?
  - Where do you get stuck? How can you fix it?
- **Kruskal's algorithm**: add edges in order of *increasing weight*, as long as they *don't cause a cycle*.

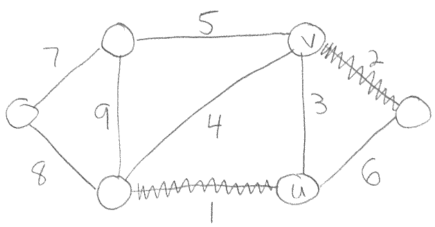
Kruskal's algorithm

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**
        $T = T \cup \{e\}$
    end if
end for

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
- $\implies e$ is the cheapest edge across $(S, V \setminus S)$
- $\implies e$ 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
- $\implies T$ is MST

Could we have chosen a different cut to use in the proof?
$S =$ the connected component of T containing v

Prim's Algorithm

- What if we want to grow a tree as a *single* connected component starting from some vertex s?
  - Which edge should we add first? How can we prove it is safe?
  - Which edge should we add next? How can we prove it is safe?

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

Prim's Algorithm

Initialize $T = \{\}$
Initialize $S = \{s\}$
while $|S| < n$ **do**
  Let $e = (u, v)$ be the minimum-cost edge from S to $V - S$
  $T = T \cup \{e\}$
  $S = S \cup \{v\}$
end while

Exercise: prove correctness

Clicker Question

Which of the following is always true?

A. Kruskal's algorithm creates disconnected trees and links them

B. Prim's and Kruskal's algorithm choose edges in different order

C. Prim's and Kruskal's algorithm choose the same set of edges

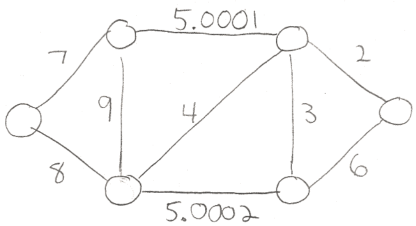
D. Only one of the algorithms is greedy

Exercise: Prove that the minimum spanning tree is unique (C).

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, so the final output is a spanning tree
- The final output is a minimum-spanning tree

Remove Distinctness Assumption?



- Hack: break ties in weights by perturbing each edge weight by a tiny unique amount.
- Implementation: break ties in an arbitrary but consistent way (e.g., lexicographic order)
- This is correct. There is a slightly more principled 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 $V - S$
  $T = T \cup \{e\}$
  $S = S \cup \{s\}$
end while

What does this remind you of?

Prim Implementation

Set $A = V$ ▷ Unattached nodes
Set $a(v) = \infty$ for all nodes ▷ Attachment cost
Set $a(s) = 0$
Set edgeTo(s) = null ▷ Attachment edge
while A not empty **do** ▷ Nodes left to attach
  Extract node $v \in A$ with smallest $a(v)$ value
  Set $T = T \cup$ edgeTo(v)
  for all edges (v, w) where $w \in A$ **do**
    if $c(v, w) < a(w)$ **then** ▷ Cheaper edge to w?
      $a(w) = c(v, w)$
      edgeTo(w) = (v, w)
    end if
  end for
end while

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

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?

BFS to check if u and v in same connected component: $O(mn)$.

(Each BFS is $O(n)$: why?)

Can we do better?

Kruskal Implementation: Union-Find

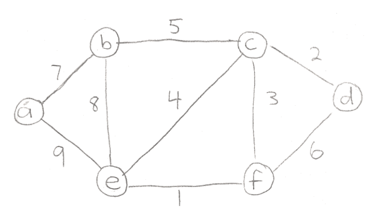
Idea: use clever data structure to maintain connected components of growing spanning tree. Should support:

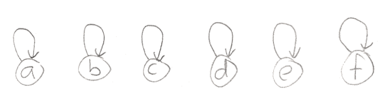
- find(v): return name of set containing v
- Union(A, B): merge two sets

for $e = 1$ to m **do**
  Let u and v be endpoints of e
  if find(u) != find(v) **then** ▷ Not in same component?
    $T = T \cup \{e\}$
    Union(find(u), find(v)) ▷ Merge components
  end if
end for

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

Union-Find Data Structure



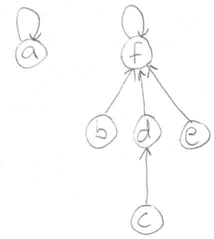


- Each set elects a representative to act as the "name" of the set
- Nodes point to their representative
- Initially, all nodes point to themself

Union-Find Data Structure

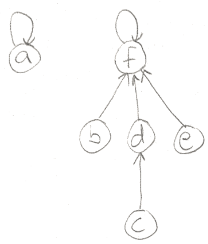
- Union(e, f)
- Union(c, d)
- Union(d, f)
- Union(b, f)
- Union(a, f)
- Time for union? $O(1)$: update one pointer

Union-Find Data Structure



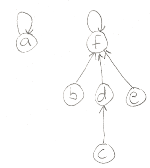
- Union(a, f): which pointer should be updated?
- **Convention**: smaller set changes its name
- Time for find? Equal to depth of tree

Union-Find Data Structure



- **Claim**: let $d =$ depth and $k = \#$ nodes in set.
- Then $d \leq \log_2(k) \implies$ find is $O(\log n)$
- Proof by induction

Union-Find Data Structure



- **Invariant**: let $d =$ depth and $k = \#$ nodes in a given set. Then $k \geq 2^d$
- 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
- New depth is $d = \max\{d_L + 1, d_R\}$
  - $k = k_L + k_R \geq 2k_L \geq 2 \cdot 2^{d_L} \geq 2^{d_L+1}$
  - $k = k_L + k_R \geq k_R \geq 2^{d_R}$
- Therefore $k \geq 2^d \implies d \leq \log_2(k)$

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: *Find* links all traversed nodes to root Essentially constant time.