CSE 421
Introduction to Algorithms

Lecture 7: Minimum Spanning Trees
Prim, Kruskal and more

Greedy Analysis Strategies

Greedy algorithm stays ahead: Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's

Structural: Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Exchange argument: Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Minimum Spanning Trees (Forests)

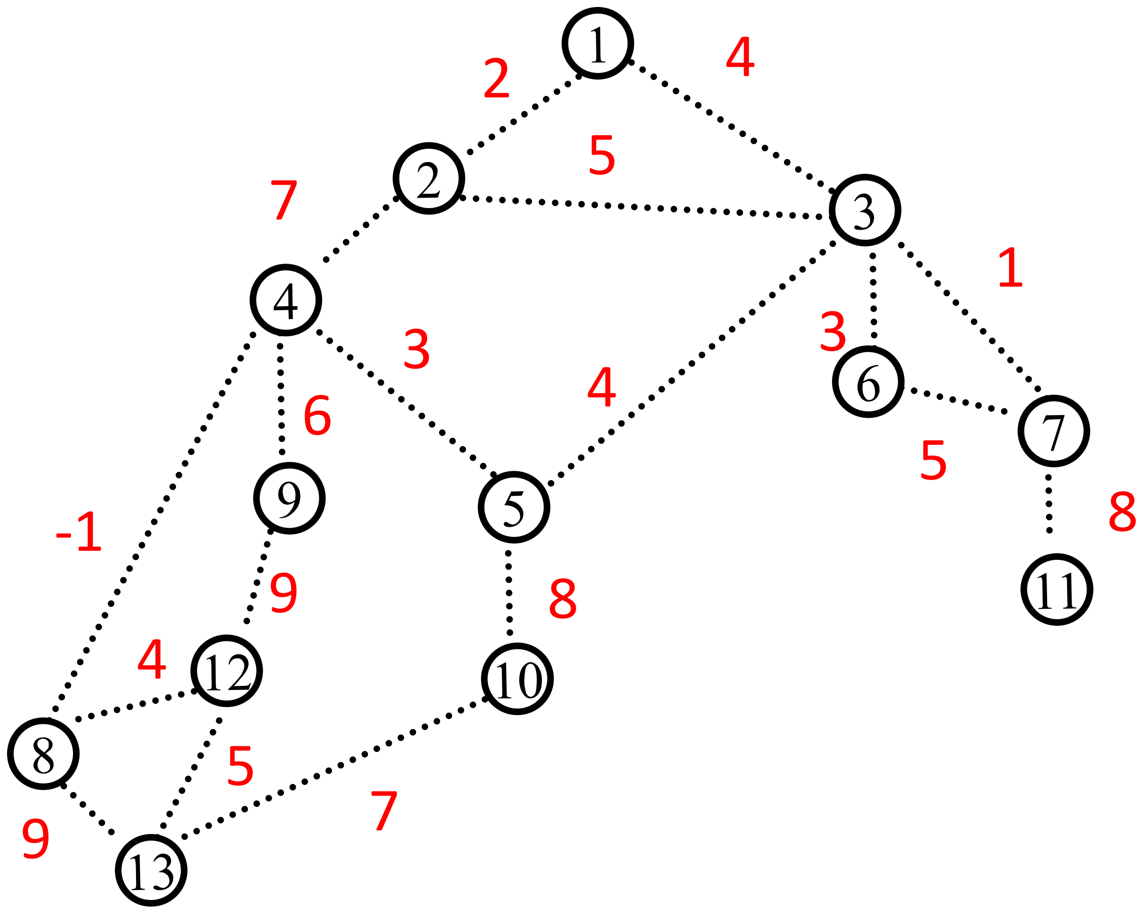
Given: an undirected graph $G = (V, E)$ with each edge e having a weight $w(e)$

Find: a subgraph T of G of minimum total weight s.t.
every pair of vertices connected in G are also connected in T

If G is connected then T is a tree

- Otherwise, T is still a forest

Weighted Undirected Graph



Greedy Algorithm

Prim's Algorithm:

- start at a vertex s
- add the cheapest edge adjacent to s
- repeatedly add the cheapest edge that joins the vertices explored so far to the rest of the graph

Exactly like Dijsktra's Algorithm but with a different objective

Dijsktra's Algorithm

Dijkstra(G,w,s)

$S \leftarrow \{s\}$

$d[s] \leftarrow 0$

while $S \neq V$ {

among all edges $e = (u, v)$ s.t. $v \notin S$ and $u \in S$ select* one with the minimum value of $d[u] + w(e)$

$S \leftarrow S \cup \{v\}$

$d[v] \leftarrow d[u] + w(e)$

$pred[v] \leftarrow u$

}

*For each $v \notin S$ maintain $d'[v]$ = minimum value of $d[u] + w(e)$
over all vertices $u \in S$ s.t. $e = (u, v)$ is in G

Prim's Algorithm

Prim(G,w,s)

$S \leftarrow \{s\}$

while $S \neq V$ {

among all edges $e = (u, v)$ s.t. $v \notin S$ and $u \in S$ select* one with the minimum value of $w(e)$

$S \leftarrow S \cup \{v\}$

$pred[v] \leftarrow u$

}

*For each $v \notin S$ maintain $small[v]$ = minimum value of $w(e)$
over all vertices $u \in S$ s.t. $e = (u, v)$ is in G

Second Greedy Algorithm

Kruskal's Algorithm:

- Start with the vertices and no edges
- Repeatedly add the cheapest edge that joins two different components.
  - i.e. cheapest edge that doesn't create a cycle

Proving Greedy MST Algorithms Correct

Instead of specialized proofs for each one we'll have one unified argument ...

Cuts

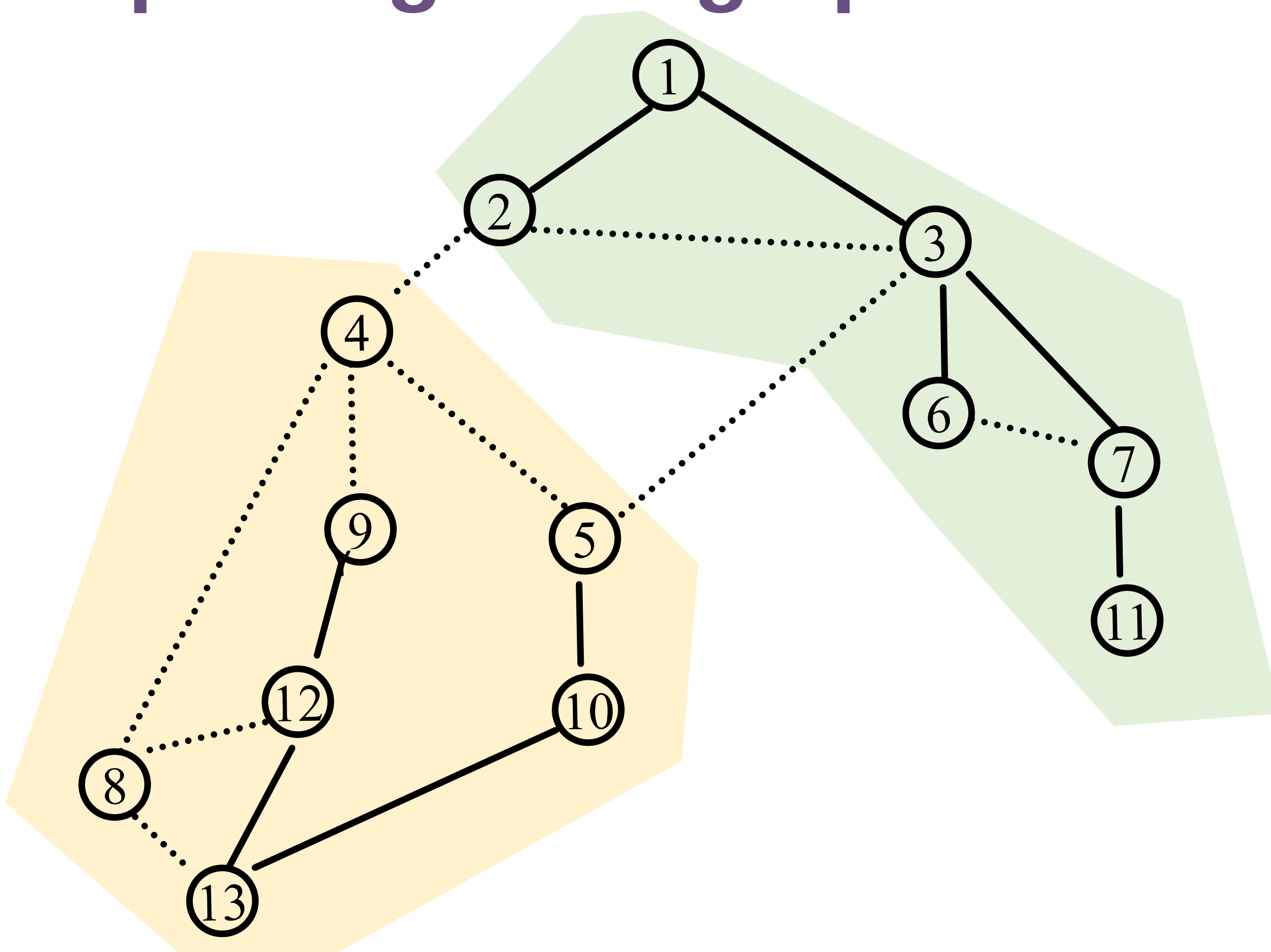
Defn: Given a graph $G = (V, E)$, a cut of G is a partition of V into two non-empty pieces, S and $V \setminus S$.

We write this cut as $(S, V \setminus S)$.

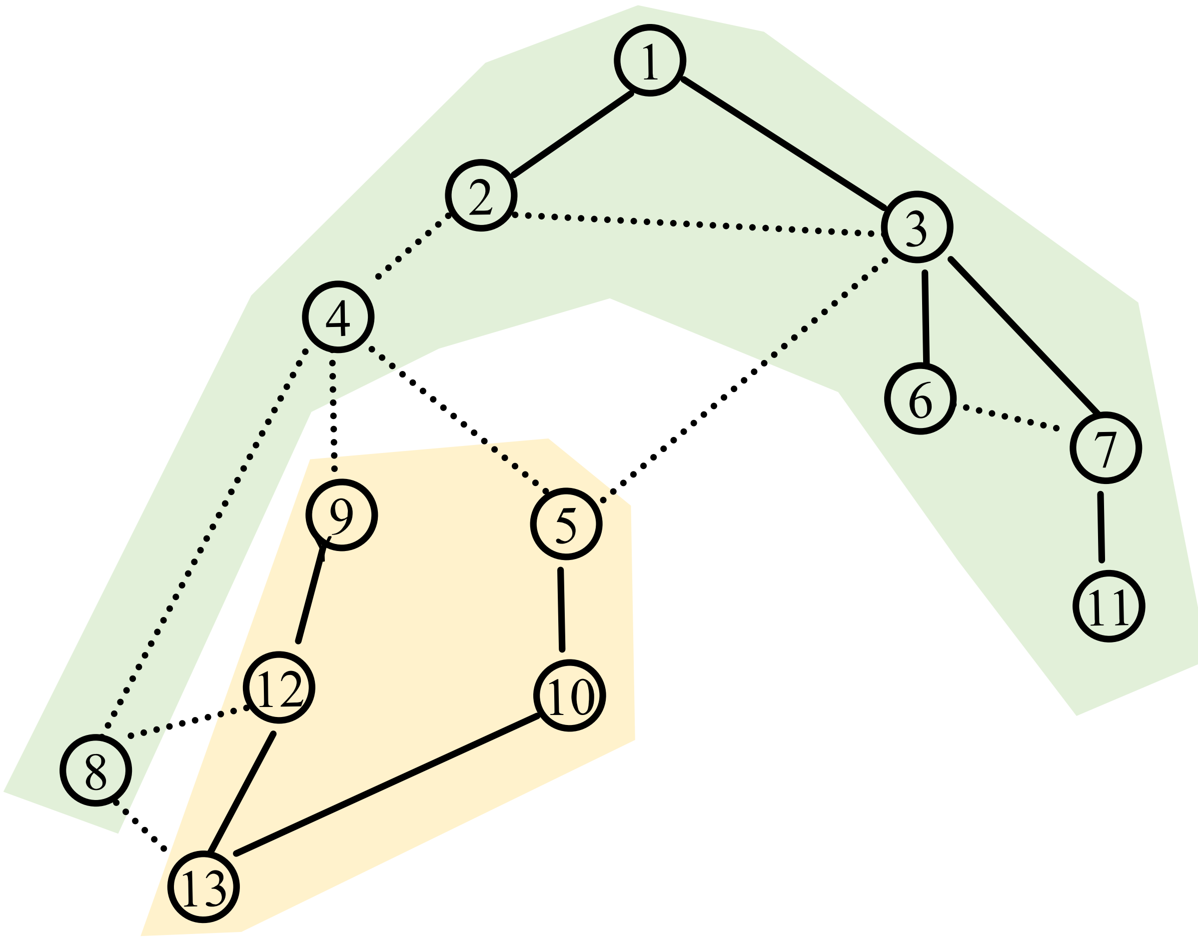
Defn: Edge e crosses cut $(S, V \setminus S)$ iff one endpoint of e is in S and the other is in $V \setminus S$

Defn: Given a graph $G = (V, E)$, and a subgraph G' of G we say that a cut $(S, V \setminus S)$ respects G' iff no edge of G' crosses $(S, V \setminus S)$

A cut respecting a subgraph



Another cut respecting the subgraph



Generic Greedy MST Algorithms and Safe Edges

Greedy algorithms for MST build up the tree/forest edge-by-edge as follows:

$T \leftarrow \emptyset$

while (T isn't spanning)

    choose* some "best" edge e (that won't create a cycle)

    $T \leftarrow T \cup \{e\}$

Defn: An edge e of G is called **safe** for T
iff there is *some* cut $(S, V \setminus S)$ that respects T
s.t e is a *cheapest* edge crossing $(S, V \setminus S)$

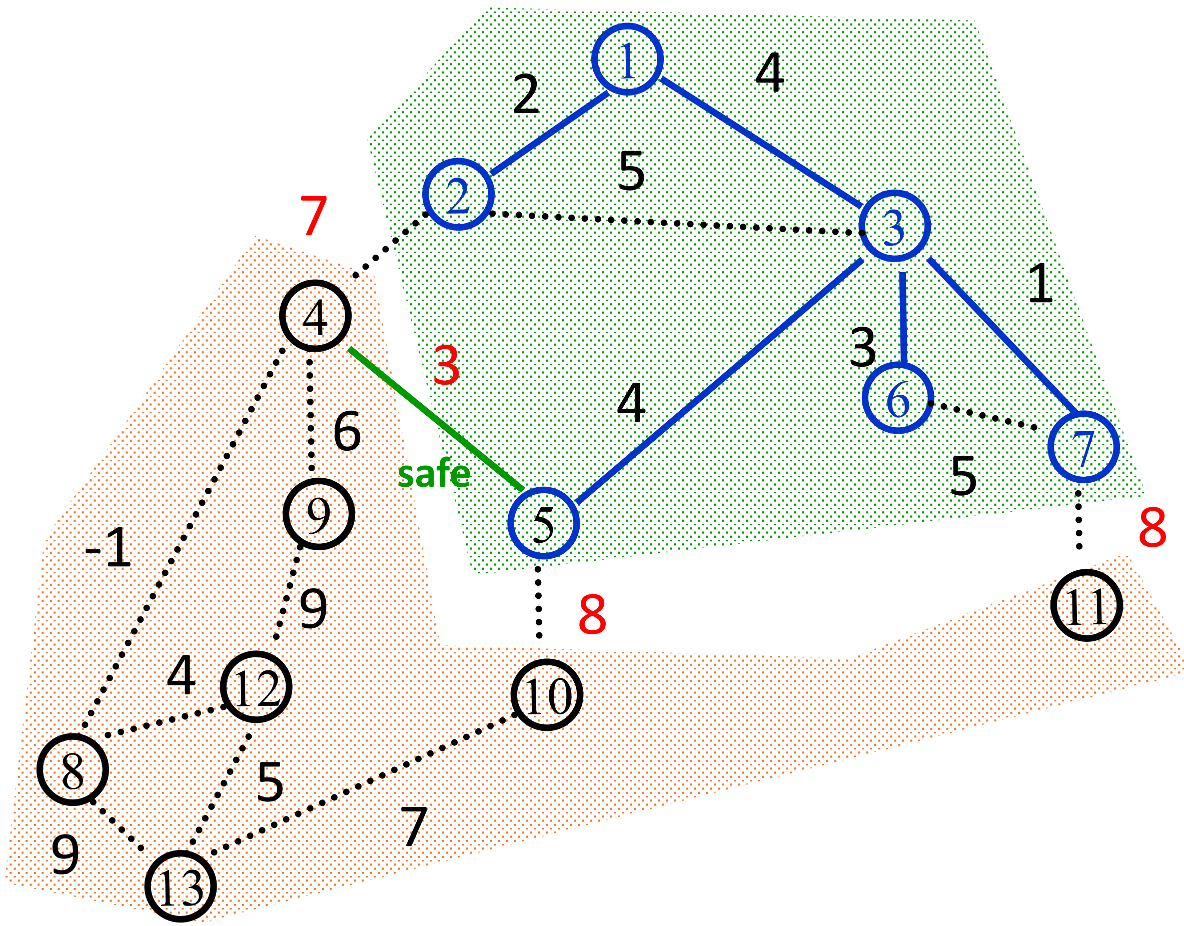
Theorem: Any greedy algorithm that always chooses* an edge e that is safe for T correctly computes an MST

Greedy algorithms: Choose safe edges that don't create cycles

Prim's Algorithm:

- Always chooses cheapest edge from current tree to rest of the graph
- This is cheapest edge across a cut that has all the vertices of current tree on one side.

Prim's Algorithm



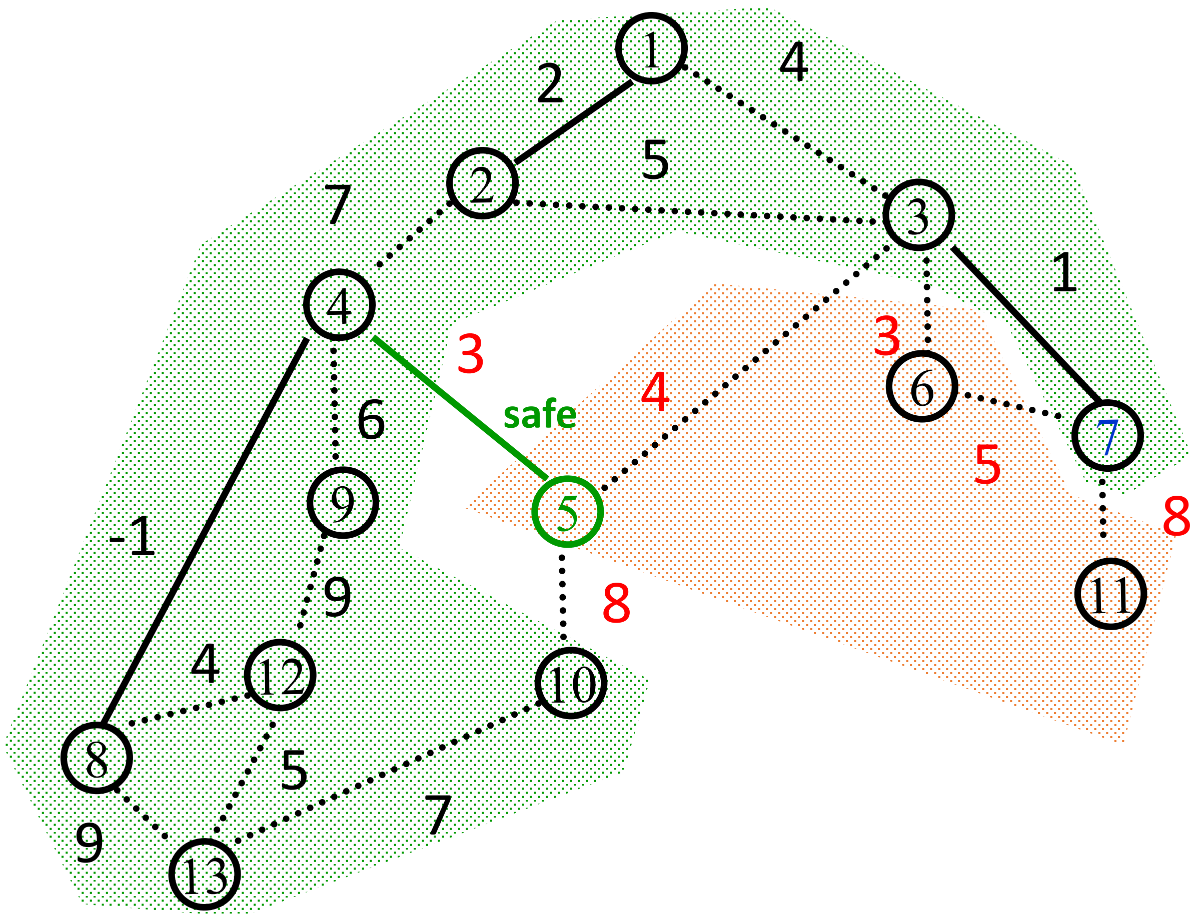
Greedy algorithms: Choose safe edges that don't create cycles

Kruskal's Algorithm:

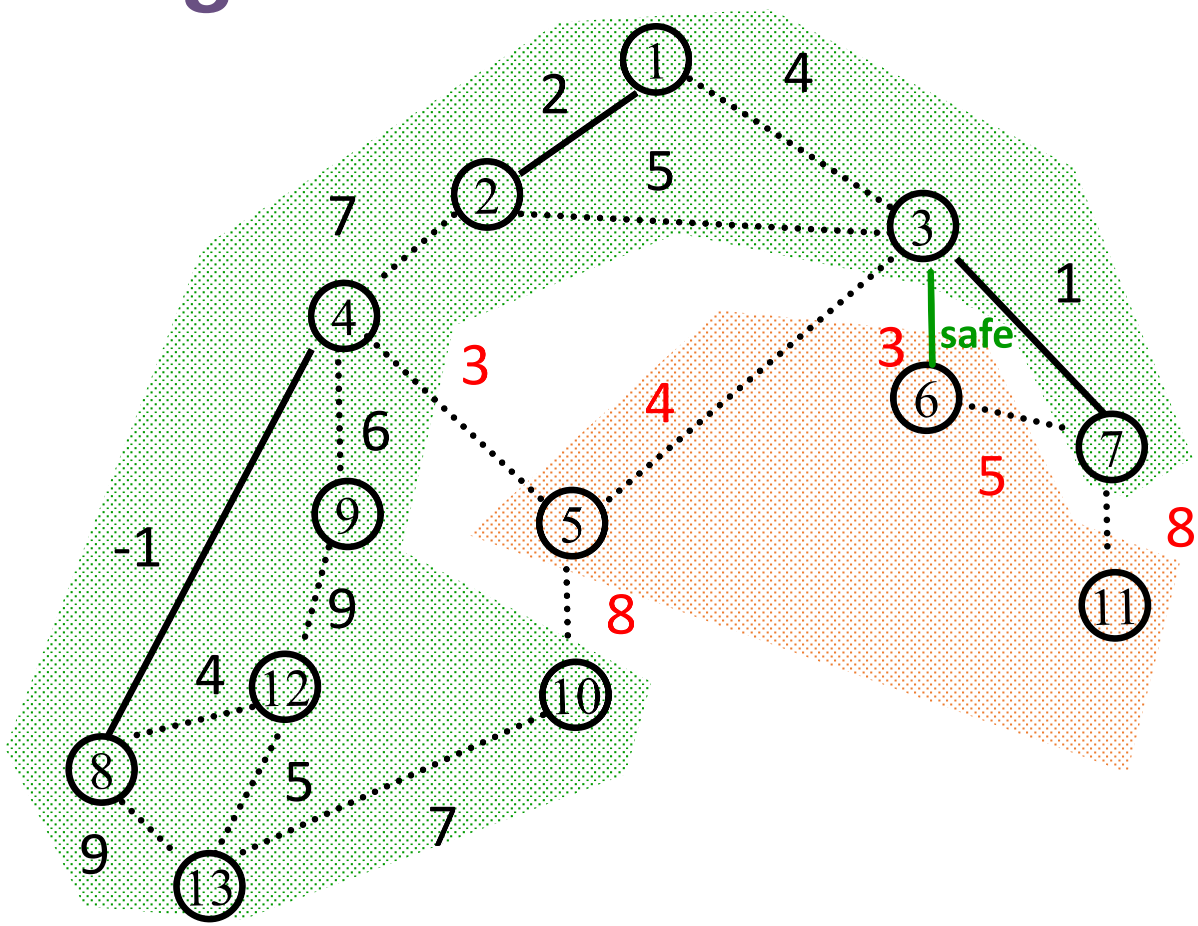
- Always choose cheapest edge connecting two pieces of the graph that aren't yet connected

- This is the cheapest edge across any cut that has those two pieces on different sides and doesn't split any other current pieces (respects the cut).

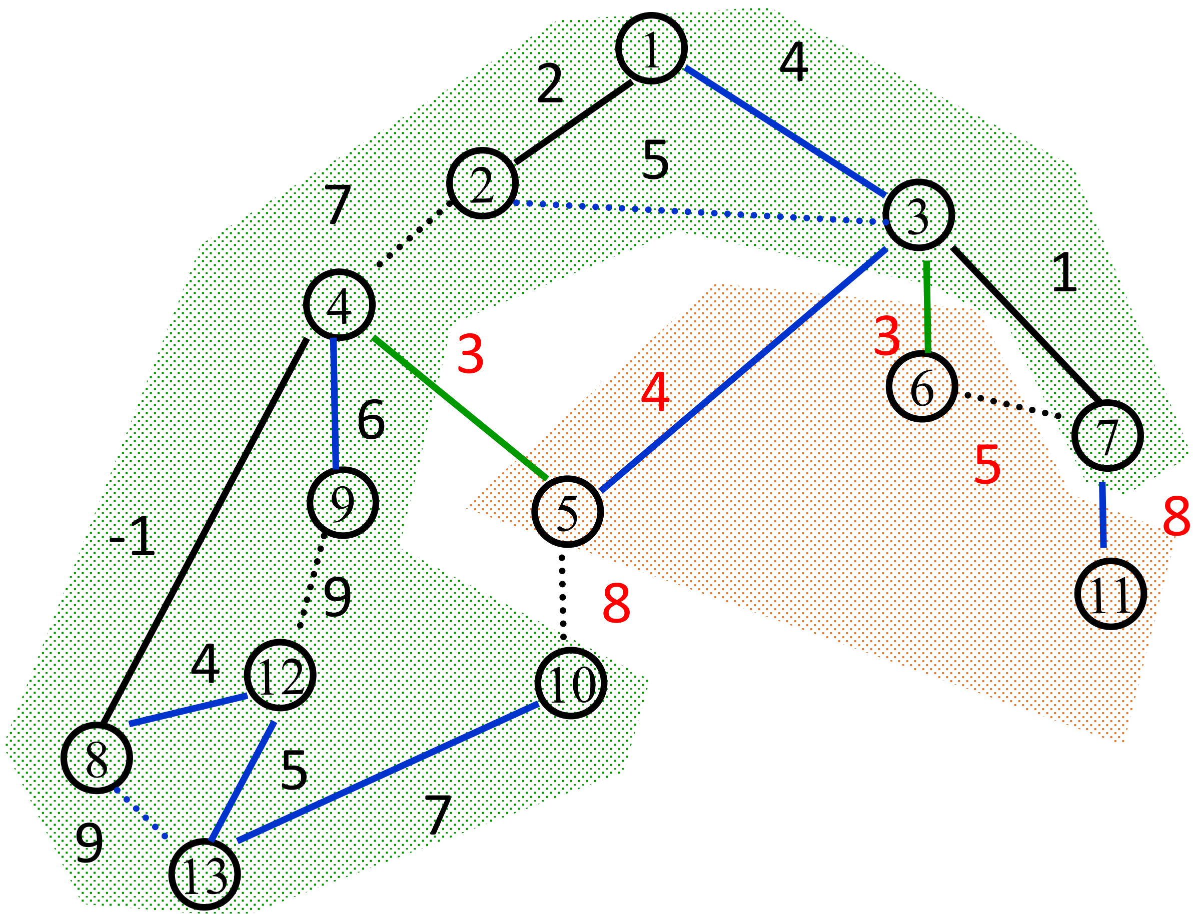
Kruskal's Algorithm



Kruskal's Algorithm



Kruskal’s Algorithm



Generic Greedy MST Algorithms and Safe Edges

Defn: An edge e of G is called **safe** for T
iff there is *some* cut $(S, V \setminus S)$ that respects T
s.t e is a *cheapest* edge crossing $(S, V \setminus S)$

Theorem: Any greedy algorithm that always chooses* an edge e that is safe for T
correctly computes an MST

Proof: We prove via induction and an exchange argument that at every step,
the subgraph T is contained in some MST of G.

Base Case: $T = \emptyset$. This is trivially true since $\emptyset$ is contained in every set.

IH: Suppose that T is contained in some MST of G.

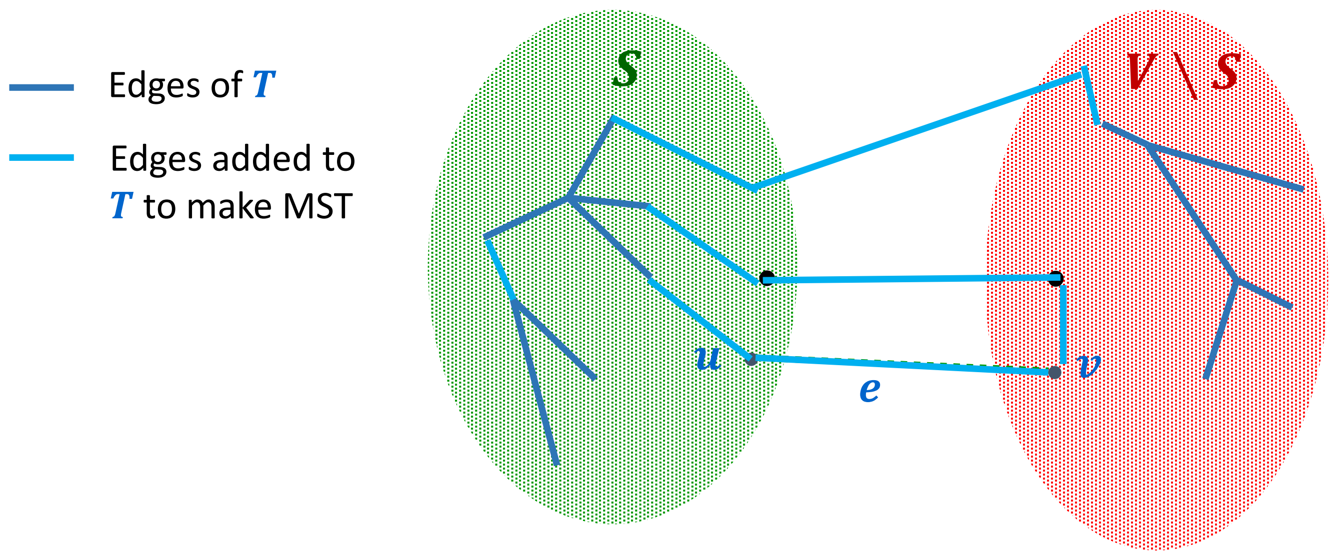
IS: We need to show that if e is safe for T then $T \cup \{e\}$ is contained
in an MST of G.

Proof of Lemma: An Exchange Argument

IS: e is a safe edge for T so e must be a cheapest edge crossing some cut $(S, V \setminus S)$ respecting T

By IH, T is contained in an MST. If this MST contains $e = (u, v)$ we're done.

Otherwise, this MST must contain a path from u to v.



All the same points are connected by the new tree

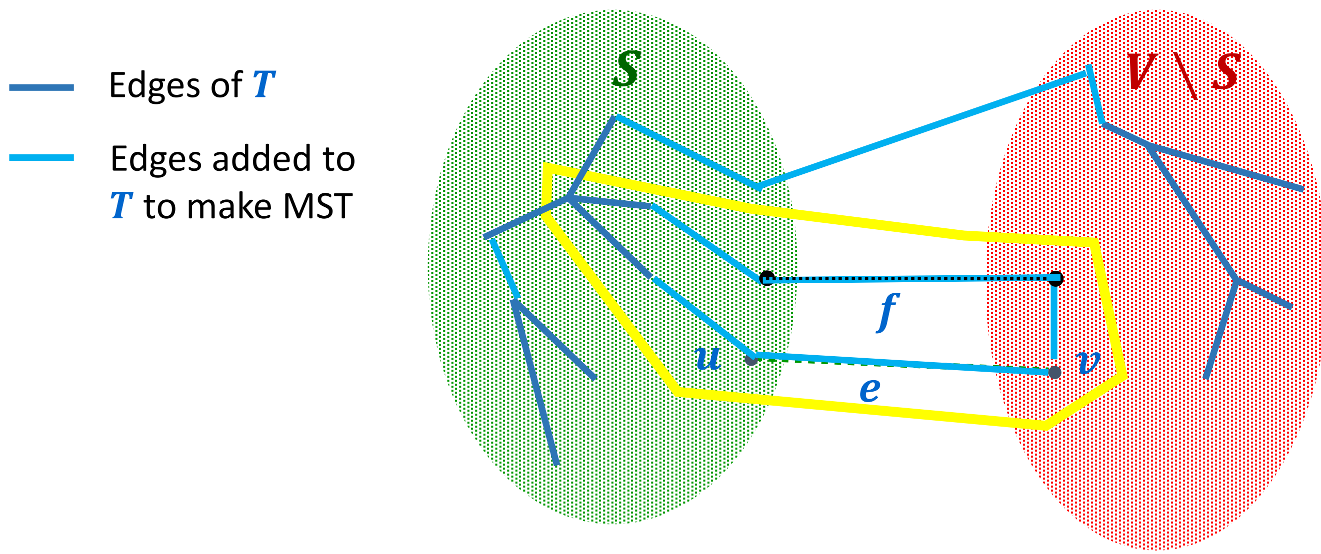
Proof of Lemma: An Exchange Argument

<u>IS</u>: $\boldsymbol{e}$ is a safe edge for $\boldsymbol{T}$ so $\boldsymbol{e}$ must be a cheapest edge crossing some cut $(\boldsymbol{S}, \boldsymbol{V} \setminus \boldsymbol{S})$ respecting $\boldsymbol{T}$

By IH, $\boldsymbol{T}$ is contained in an MST. If this MST contains $\boldsymbol{e} = (\boldsymbol{u}, \boldsymbol{v})$ we're done.

Otherwise, this MST must contain a path from $\boldsymbol{u}$ to $\boldsymbol{v}$.

This must contain some edge $\boldsymbol{f}$ crossing the cut.



Since $\boldsymbol{e}$ was cheapest $\boldsymbol{w(e) \le w(f)}$

Exchange $\boldsymbol{e}$ for $\boldsymbol{f}$ to get a new spanning subgraph that is at least as cheap and contains $\boldsymbol{T} \cup \{\boldsymbol{e}\}$.

All the same points are connected by the new tree

Kruskal's Algorithm: Implementation & Analysis

- First sort the edges by weight $O(\boldsymbol{m} \log \boldsymbol{m})$
- Go through edges from smallest to largest
  - if endpoints of edge $\boldsymbol{e}$ are currently in different components
    - then add to the graph
    - else skip

Union-Find data structure handles test for different components

- Total cost of union find: $O(\boldsymbol{m} \cdot \boldsymbol{\alpha}(\boldsymbol{n}))$ where $\boldsymbol{\alpha}(\boldsymbol{n}) \ll \log \boldsymbol{m}$

Overall $O(\boldsymbol{m} \log \boldsymbol{m})$ which is $O(\boldsymbol{m} \log \boldsymbol{n})$

Union-Find disjoint sets data structure

Maintaining components

- start with n different components
  - one per vertex
- find components of the two endpoints of e
  - $2m$ finds
- union two components when edge connecting them is added
  - $n - 1$ unions

Prim's Algorithm with Priority Queues

- For each vertex $\boldsymbol{u}$ not in tree maintain current cheapest edge from tree to $\boldsymbol{u}$
  - Store $\boldsymbol{u}$ in priority queue with key = weight of this edge
- Operations:
  - $\boldsymbol{n-1}$ insertions (each vertex added once)
  - $\boldsymbol{n-1}$ delete-mins (each vertex deleted once)
    - pick the vertex of smallest key, remove it from the p.q. and add its edge to the graph
  - $< \boldsymbol{m}$ decrease-keys (each edge updates one vertex)

Prim's Algorithm with Priority Queues

Priority queue implementations: same complexity as Dijkstra

- Array
  - insert $O(\mathbf{1})$, delete-min $O(\boldsymbol{n})$, decrease-key $O(\mathbf{1})$
  - total $O(\boldsymbol{n} + \boldsymbol{n}^2 + \boldsymbol{m}) = O(\boldsymbol{n}^2)$
- Heap
  - insert, delete-min, decrease-key all $O(\log \boldsymbol{n})$
  - total $O(\boldsymbol{m} \log \boldsymbol{n})$

Worse if $\boldsymbol{m} = \boldsymbol{\Theta}(\boldsymbol{n}^2)$

- $\boldsymbol{d}$-Heap ($\boldsymbol{d} = \boldsymbol{m}/\boldsymbol{n}$)

$\boldsymbol{m}$
  - insert, decrease-key $O(\log_{\boldsymbol{m}/\boldsymbol{n}} \boldsymbol{n})$

$\boldsymbol{n} - \mathbf{1}$
  - delete-min $O((\boldsymbol{m}/\boldsymbol{n}) \log_{\boldsymbol{m}/\boldsymbol{n}} \boldsymbol{n})$
  - total $O(\boldsymbol{m} \log_{\boldsymbol{m}/\boldsymbol{n}} \boldsymbol{n})$

Better for all values of $\boldsymbol{m}$

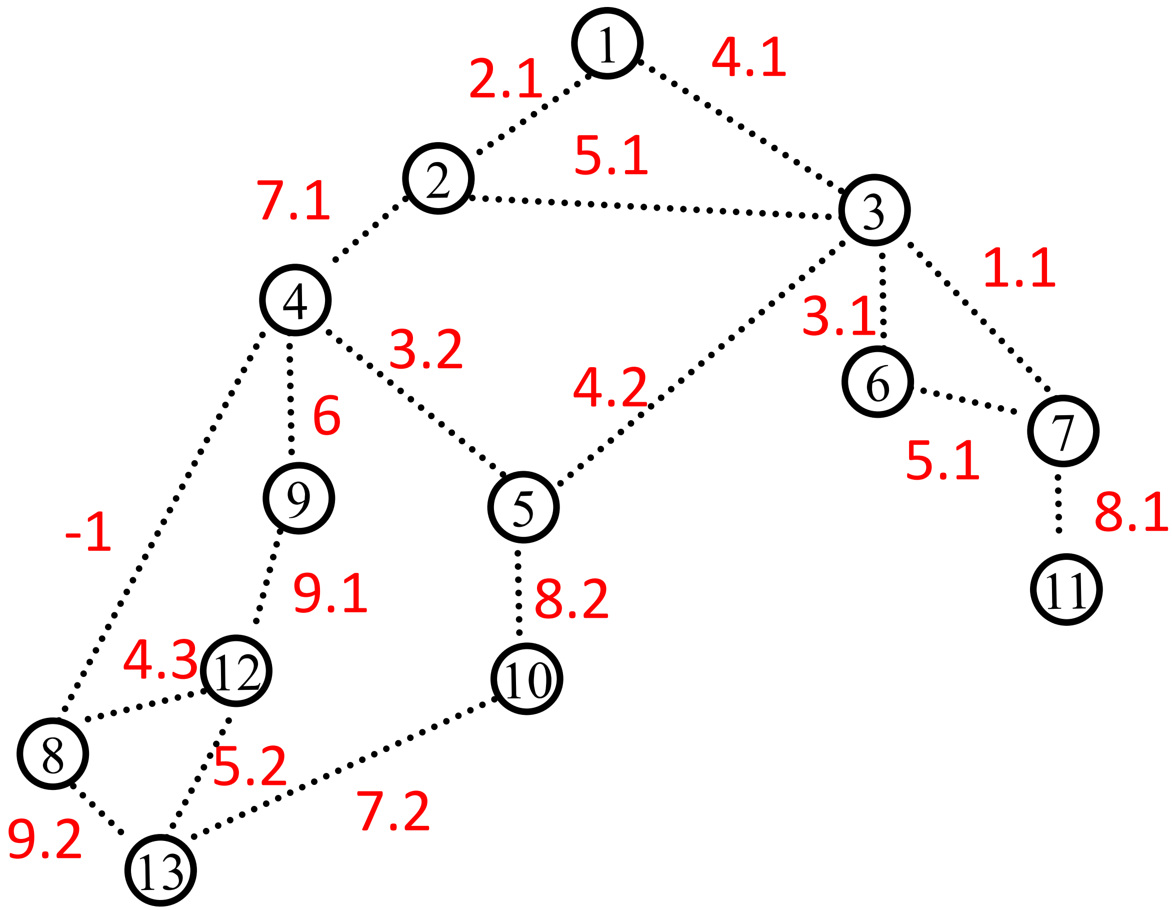
Boruvka's Algorithm (1927)

A bit like Kruskal's Algorithm

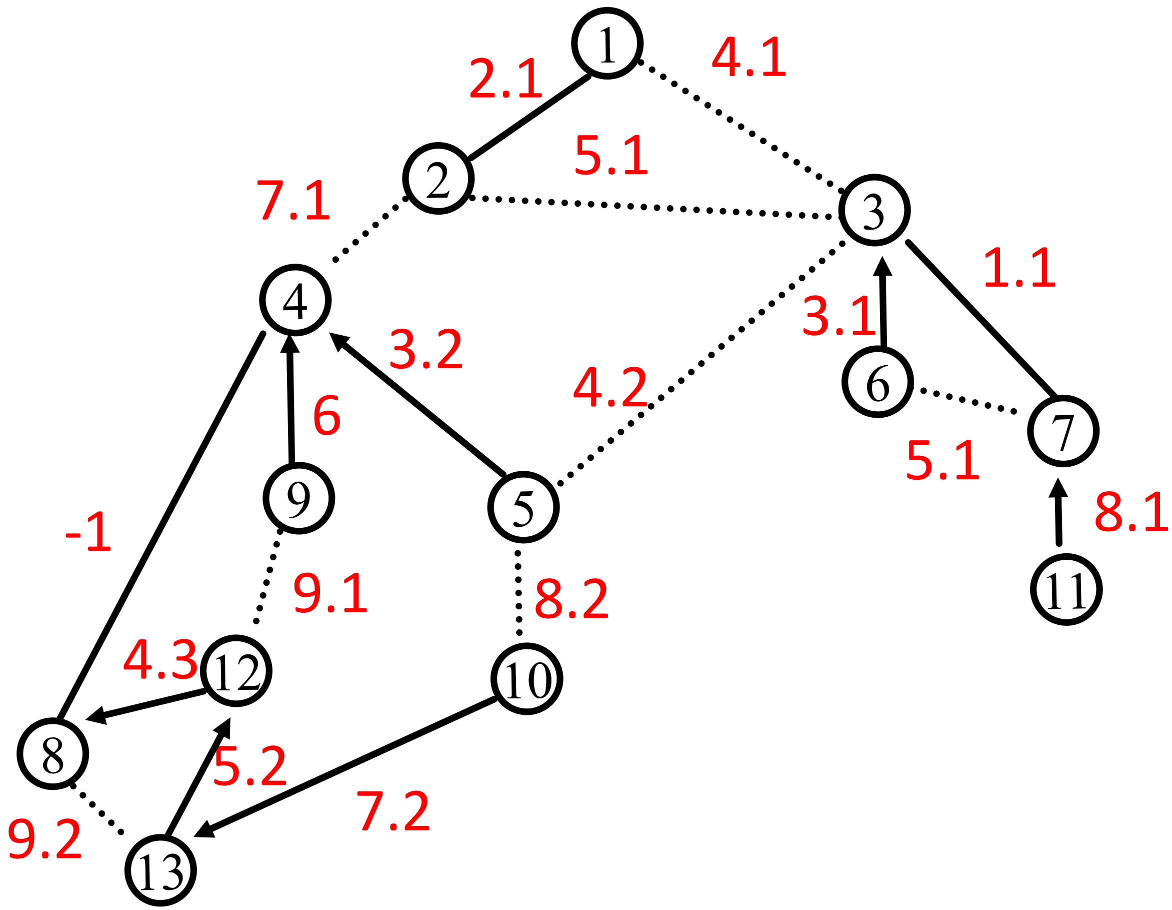
- Start with n components consisting of a single vertex each
- At each step:
  - Each component chooses to add its cheapest outgoing edge
  - Two components may choose to add the same edge
  - Need to add a tiebreaker on edge weights (no equal weights) to avoid cycles

Useful for parallel algorithms since components may be processed (almost) independently

Boruvka

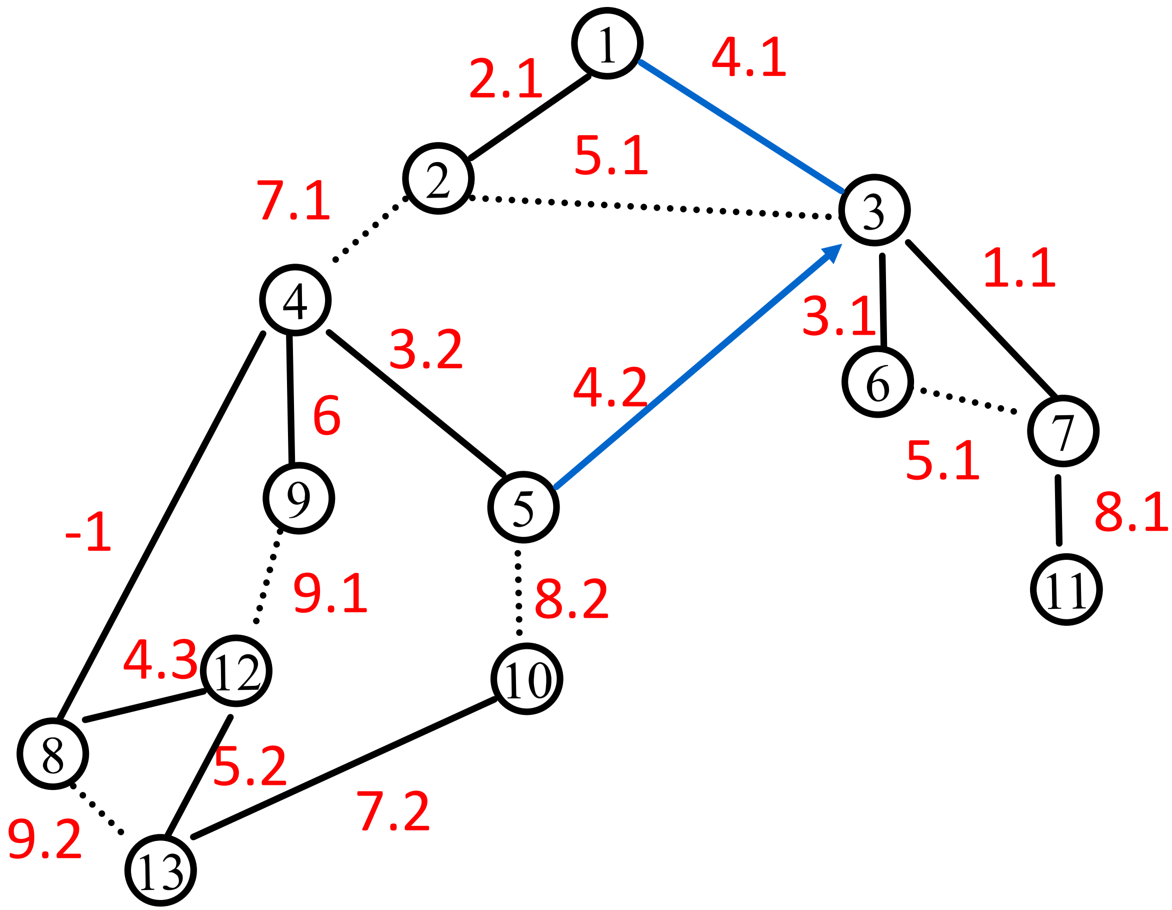

1
2
3
4
5
6
7
8
9
10
11
12
13
2.1
4.1
5.1
7.1
1.1
3.1
3.2
4.2
6
5.1
8.1
-1
9.1
8.2
4.3
5.2
7.2
9.2

Boruvka



1
2
3
4
5
6
7
8
9
10
11
12
13
2.1
4.1
5.1
7.1
1.1
3.1
3.2
6
4.2
5.1
8.1
-1
9.1
8.2
4.3
5.2
7.2
9.2

Boruvka



1
2
3
4
5
6
7
8
9
10
11
12
13
2.1
4.1
5.1
7.1
1.1
3.1
3.2
4.2
6
5.1
8.1
-1
9.1
8.2
4.3
5.2
7.2
9.2

Many other minimum spanning tree algorithms, most of them greedy

Cheriton & Tarjan

- Use a queue of components
  - Component at head chooses cheapest outgoing edge
  - New merged component goes to tail of the queue.
- $O(m \operatorname{loglog} n)$ time

Chazelle

- $O(m \cdot \alpha(m) \cdot \log(\alpha(m)))$ time
  - Incredibly hairy algorithm

Karger, Klein & Tarjan

- $O(m + n)$ time randomized algorithm that works most of the time

Applications of Minimum Spanning Tree Algorithms

MST is a fundamental problem with diverse applications

- Network design
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms
  - travelling salesperson problem, Steiner tree
- Indirect applications
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - reducing data storage in sequencing amino acids
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid network cycles
- Clustering

Applications of Minimum Spanning Tree Algorithms

Minimum cost network design:

- Build a network to connect all locations $\{v_1, \ldots, v_n\}$
- Cost of connecting v_i to v_j is $w(v_i, v_j) > 0$.
- Choose a collection of links to create that will be as cheap as possible
- Any minimum cost solution is an MST
  - If there is a solution containing a cycle then we can remove any edge and get a cheaper solution

Applications of Minimum Spanning Tree Algorithms

Maximum Spacing Clustering:

Given:

- Collection U of n points $\{p_1, \dots, p_n\}$
- Distance measure $d(p_i, p_j)$ satisfying
  - Zero base: $d(p_i, p_j) = 0$
  - Nonnegativity: $d(p_i, p_j) > 0$ for $i \neq j$
  - Symmetry: $d(p_i, p_j) = d(p_j, p_i)$
- Positive integer $k \leq n$

Find: a k-clustering, i.e. partition of U into k clusters $C_1, \dots, C_k$, s.t.
the spacing between the clusters is as large possible where
spacing = min$\{d(p_i, p_j)$: p_i and p_j are in different clusters$\}$

Greedy Algorithm for Maximum Spacing Clustering

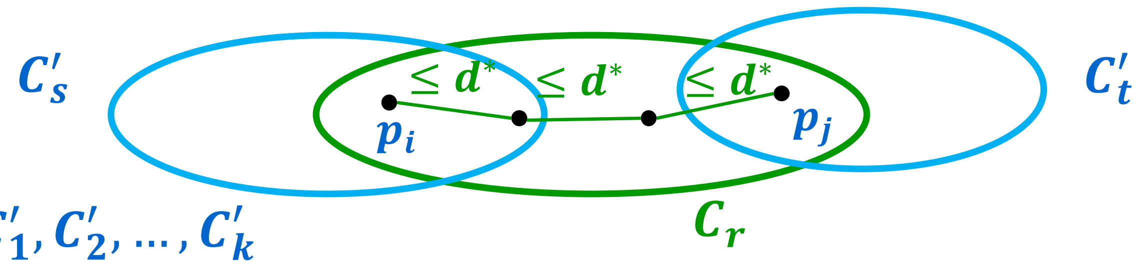
- Start with n clusters each consisting of a single point
- Repeat until only k clusters remain
  - find the closest pair of points in different clusters under distance d
  - merge their clusters

Gets the same components as Kruskal's Algorithm does if we stop early!
- The sequence of closest pairs is exactly the MST

- Alternatively...
  - we could run any MST algorithm once and for any k we could get the maximum spacing k-clustering by deleting the $k - 1$ most expensive edges in the MST

Proof that this works

- Removing the $k-1$ most expensive edges from an MST yields k components $C_1, \ldots, C_k$ and the spacing for them is precisely the cost d^* of the $k-1$st most expensive edge in the tree



- Consider any other k-clustering $C'_1, C'_2, \ldots, C'_k$
  - There is some pair of points p_i, p_j s.t. p_i, p_j are in some cluster C_r but p_i, p_j are in different clusters C'_s and C'_t
  - Since both are in C_r, points p_i and p_j are joined by a path with each hop of distance at most d^*
  - This path must have some *adjacent* pair in different clusters of $C'_1, C'_2, \ldots, C'_k$ so the spacing of $C'_1, C'_2, \ldots, C'_k$ must be at most d^* ∎