CS3000: Algorithms & Data
Jonathan Ullman

Lecture 13:

- Minimum Spanning Trees

Mar 9, 2020

Midterm II

- **In Class Wednesday March 25th**
  - Working on a backup plan

- **Exactly the same format/rules as Midterm I**
- **Topics: Graph Algorithms**
  - Key definitions, properties
  - Representing graphs
  - DFS and topological sort
  - Shortest Paths: BFS, Dijkstra, Bellman-Ford
  - Minimum spanning trees
  - Network flow

} this week

Minimum Spanning Trees

Network Design

- **Build a cheap, well connected network**
- We are given
  - a set of nodes $V = \{v_1, \dots, v_n\}$
  - a set of potential edges $E \subseteq V \times V$
- Want to build a network to connect these locations
  - Every v_i, v_j must be well connected
  - Must be as cheap as possible

- Many variants of network design
  - Recall the bus routes problem from HW2

Minimum Spanning Trees (MST)

nodes potential edges

- **Input:** a weighted graph $G = (V, E, \{w_e\})$ edge costs
  - Undirected, connected, weights may be negative
  - All edge weights are distinct (makes life simpler)

connected and acyclic and has n-1 edges

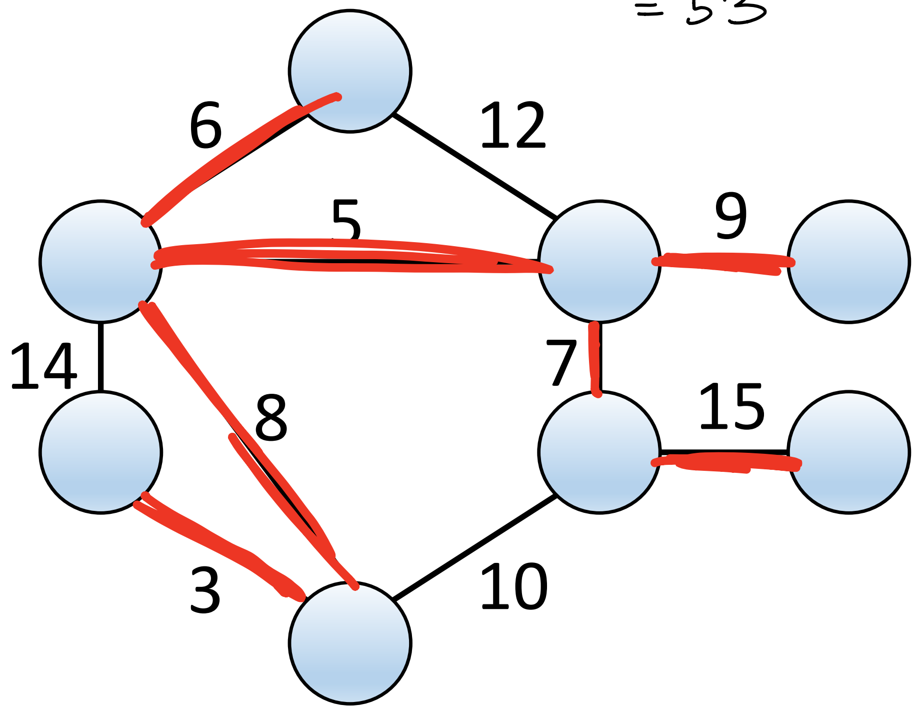
- **Output:** a minimum weight spanning tree $T \subseteq E$
  - A spanning tree of G is a subset of $T \subseteq E$ of the edges such that (V, T) forms a tree
  - Weight of a tree T is the sum of the edge weights $\sum_{e \in T} w_e$
  - We'll use T^* to denote "the" minimum spanning tree

$$\min_{\text{trees } T \subseteq E} \sum_{e \in T} w_e$$

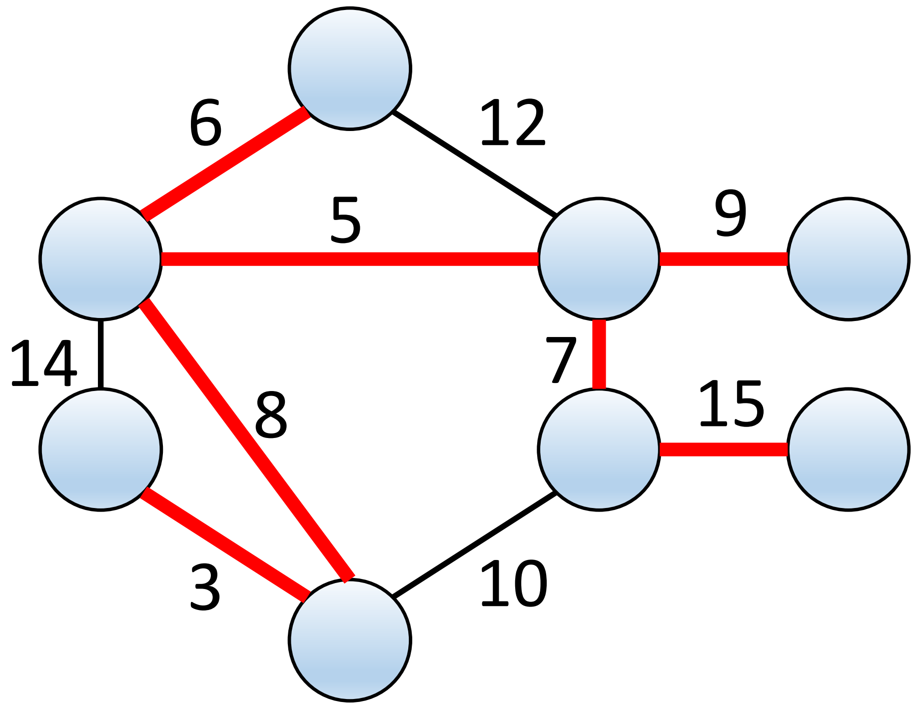
Minimum Spanning Trees (MST)

3+ 5+ 6+ 7+ 8+ 9+15

= 53



Minimum Spanning Trees (MST)

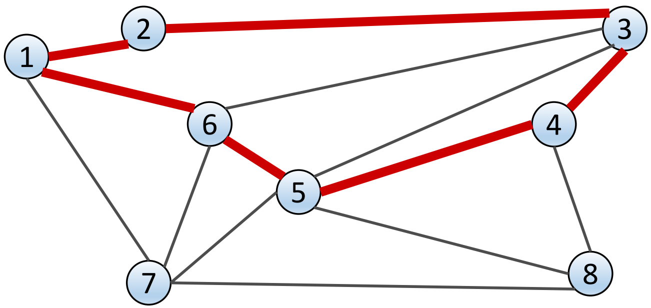


MST Algorithms

- There are at least four reasonable MST algorithms
  - Borůvka's Algorithm: start with $T = \emptyset$, in each round add cheapest edge out of each connected component
  - Prim's Algorithm: start with some s, at each step add cheapest edge that grows the connected component
  - Kruskal's Algorithm: start with $T = \emptyset$, consider edges in ascending order, adding edges unless they create a cycle
  - Reverse-Kruskal: start with $T = E$, consider edges in descending order, deleting edges unless it disconnects

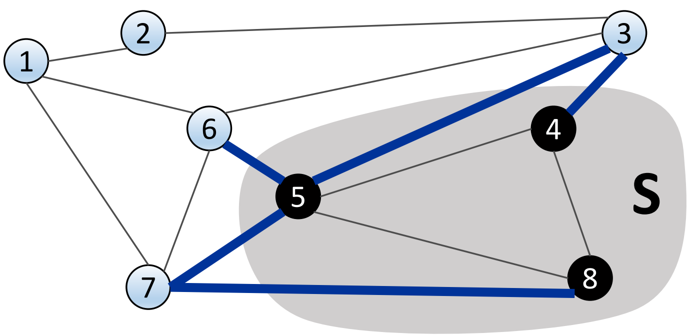
Cycles and Cuts

- **Cycle:** a set of edges $(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_1)$



Cycle C = (1,2),(2,3),(3,4),(4,5),(5,6),(6,1)

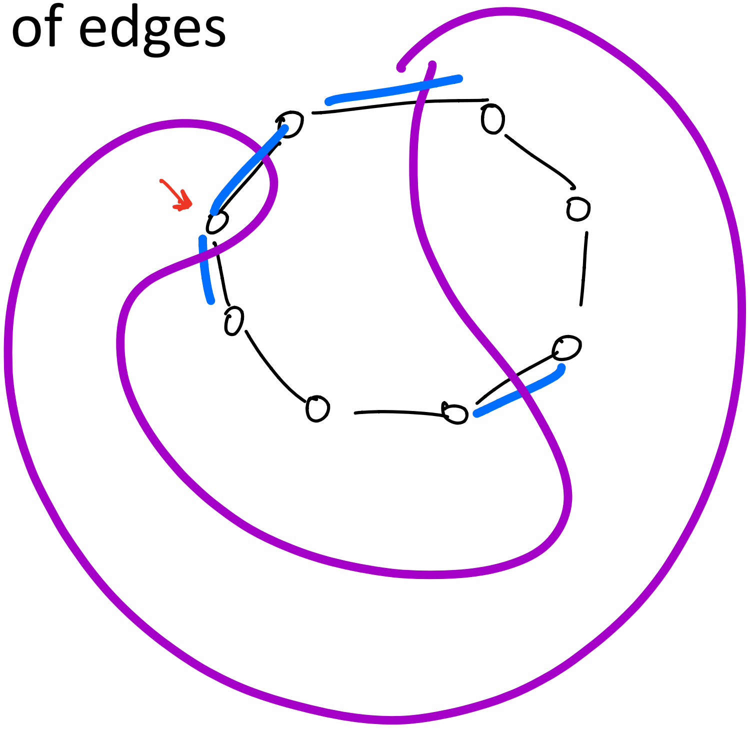
- **Cut:** a partition of the nodes into $S, \bar{S}$



Cut S = {4, 5, 8}
Cutset = (5,6), (5,7), (3,4), (3,5), (7,8)

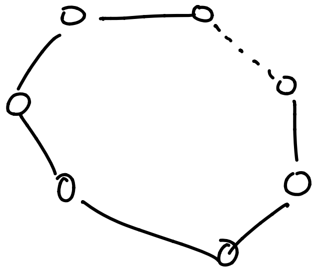
Cycles and Cuts

- **Fact:** a cycle and a cutset intersect in an even number of edges

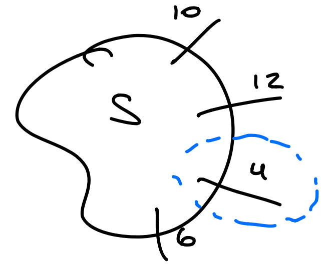


Cycles and Cuts

- **Fact:** removing an edge from a cycle doesn't disconnect any nodes

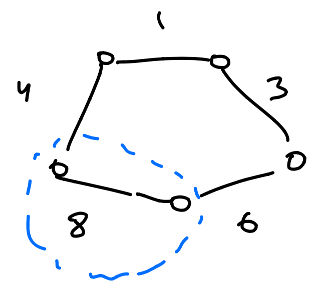


Properties of MSTs



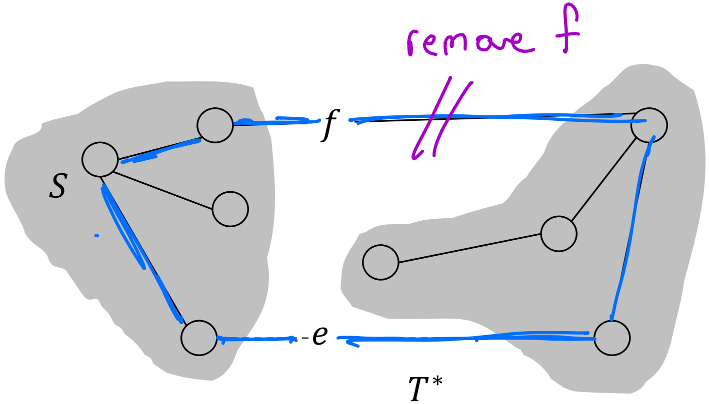
- **Cut Property:** Let S be a cut. Let e be the minimum weight edge cut by S. Then the MST T^* contains e
  - We call such an e a safe edge

- **Cycle Property:** Let C be a cycle. Let f be the maximum weight edge in C. Then the MST T^* does not contain f.
  - We call such an f a useless edge



Proof of Cut Property

- **Cut Property:** Let S be a cut. Let e be the minimum weight edge cut by S. Then the MST T^* contains e



Proof by contradiction:

Assume T^* is the MST and it doesnt contain e.

If we add e to T^* there must be a cycle C. C contains ≥ 2 edges crossing the cut $\{e, f\}$, $w(f) > w(e)$

If we remove f from $T^* \cup \{e\}$ the the total wt is lower than T^*

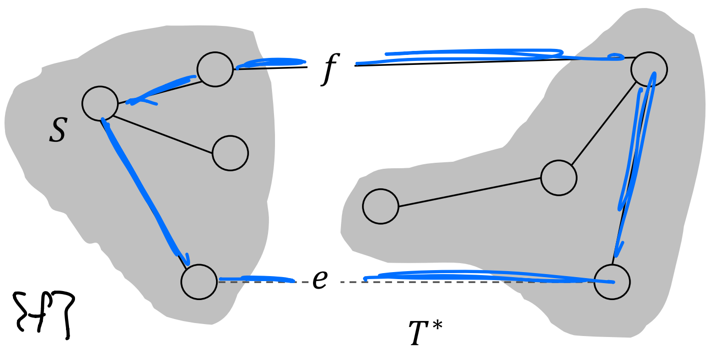
$T^* \cup \{e\} \setminus \{f\}$ is still a tree $\square$

Proof of Cycle Property

- **Cycle Property:** Let C be a cycle. Let f be the max weight edge in C. The MST T^* does not contain f.

Proof by Contradiction:

Assume T^* is the MST and contains f.



If we remove f, the graph $T^* \setminus \{f\}$ has two components $S, \bar{S}$. There is some edge $e \in C$ cut by S. $wt(e) < wt(f)$

Thus $T^* \setminus \{f\} \cup \{e\}$ is a spanning tree w/ lower weight □

Ask the Audience

- Assume G has distinct edge weights
- **True/False?** If e is the edge with the smallest weight, then e is always in the MST T^*
- **True/False?** If f is the edge with the largest weight, then f is never in the MST T^*

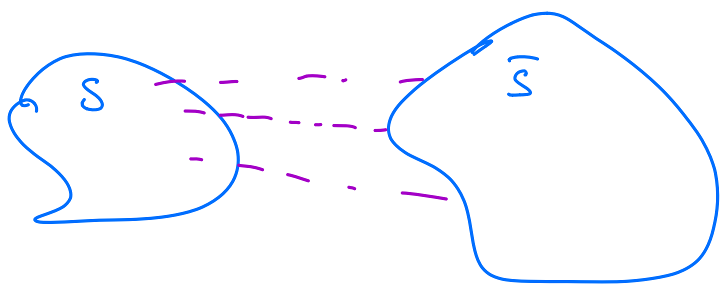
The "Only" MST Algorithm

- **GenericMST:**
  - Let $T = \emptyset$
  - Repeat until T is connected:
    - Find one or more safe edges not in T
    - Add safe edges to T

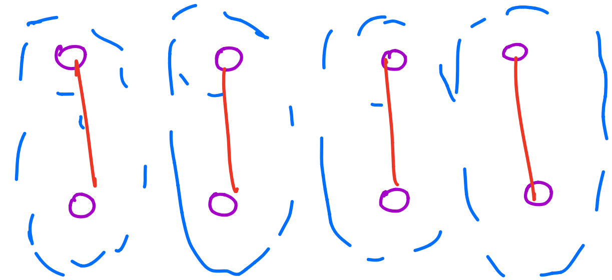
- **Theorem: GenericMST** outputs an MST

Suppose T is not connected. Then it has multiple connected components.

One of the potential edges crossing the cut is a safe edge



Borůvka's Algorithm



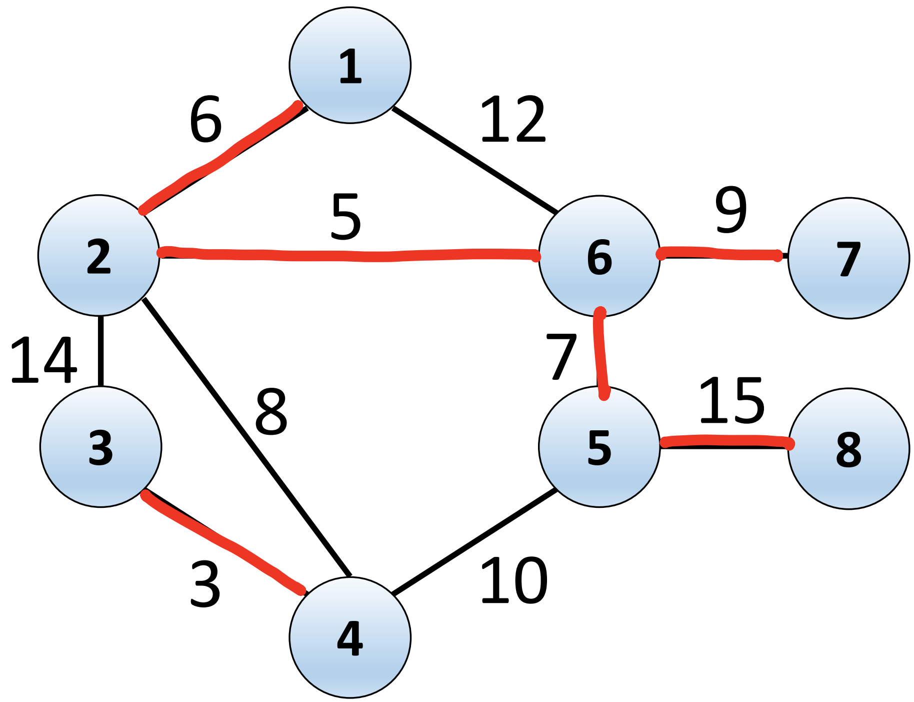
- **Borůvka:**
  - Let $T = \emptyset$
  - Repeat until T is connected:
    - Let $C_1, \dots, C_k$ be the connected components of (V, T)
    - Let $e_1, \dots, e_k$ be the safe edge for the cuts $C_1, \dots, \cancel{C_m}$ C_k
    - Add $e_1, \dots, e_k$ to T

Will contain duplicates

- **Correctness:** every edge we add is safe

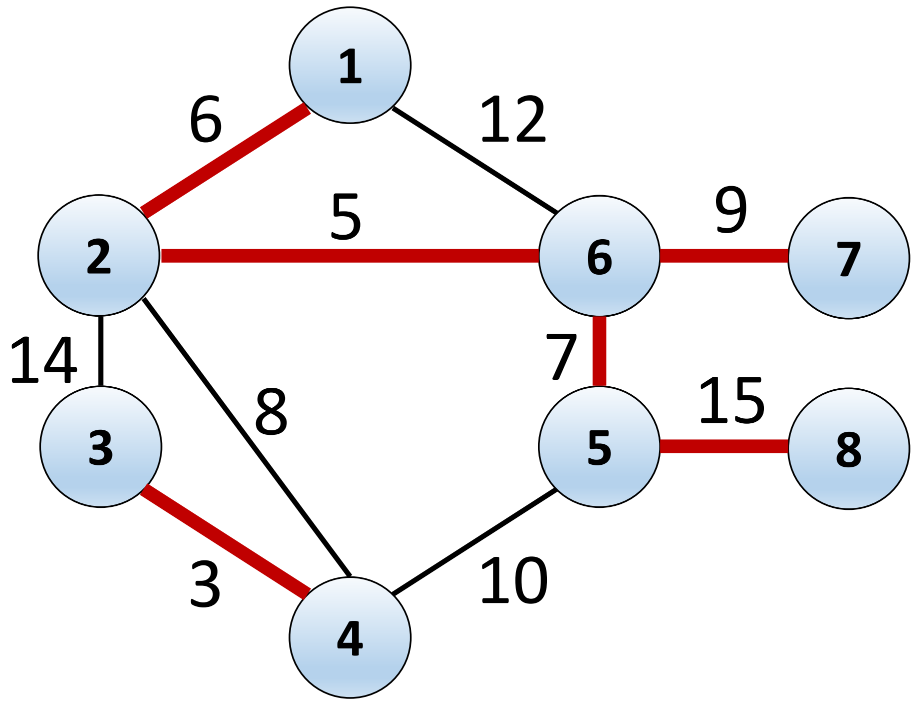
Borůvka's Algorithm

Label Connected Components



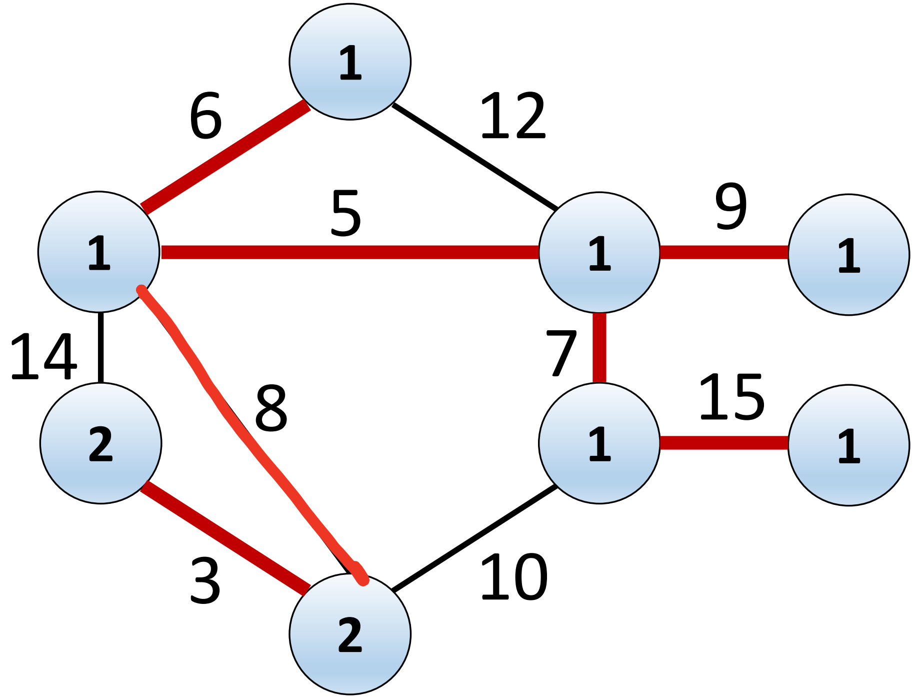
Borůvka's Algorithm

Add Safe Edges



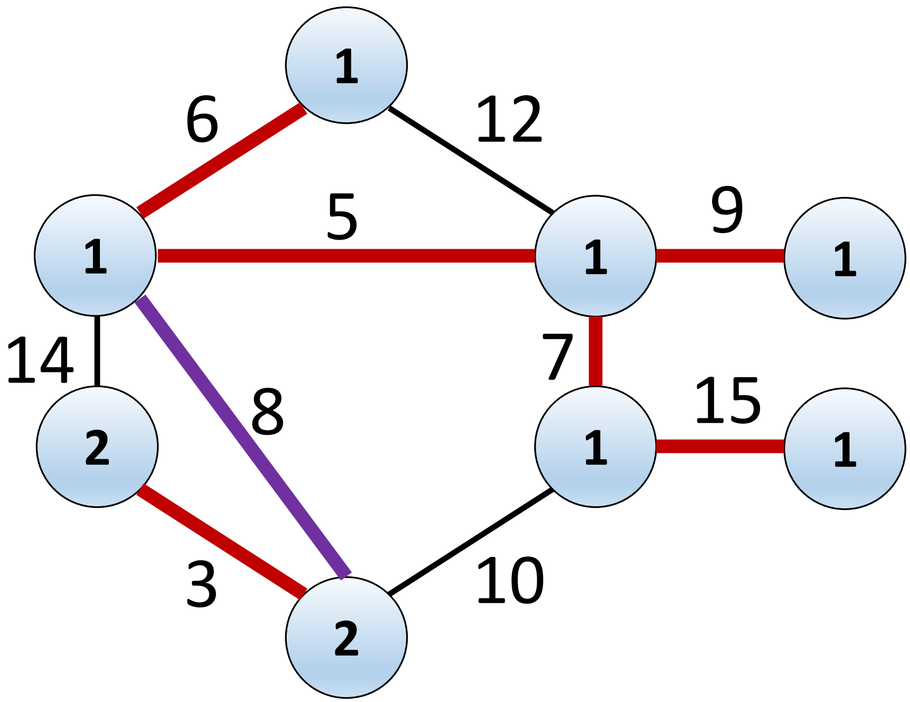
Borůvka's Algorithm

Label Connected Components



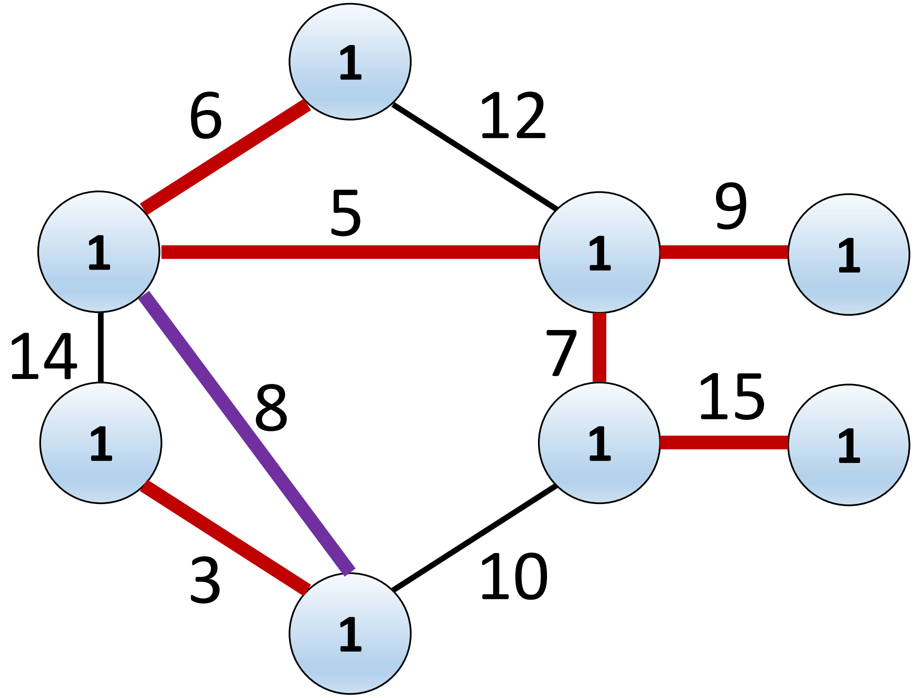
Borůvka’s Algorithm

Add Safe Edges



Borůvka's Algorithm

Done!



Borůvka's Algorithm (Running Time)

- **Borůvka**
  - Let $T = \emptyset$
  - Repeat until T is connected:
    - Let $C_1, \ldots, C_k$ be the connected components of (V, T) $O(n+m)$
    - Let $e_1, \ldots, e_k$ be the safe edge for the cuts $C_1, \ldots, C_m$
    - Add $e_1, \ldots, e_k$ to T

- Running time
  - How long to find safe edges?
  - How many times through the main loop?

BFS the graph to find components

Loop through edges keep track of min cut edge for each component

Borůvka's Algorithm (Running Time)

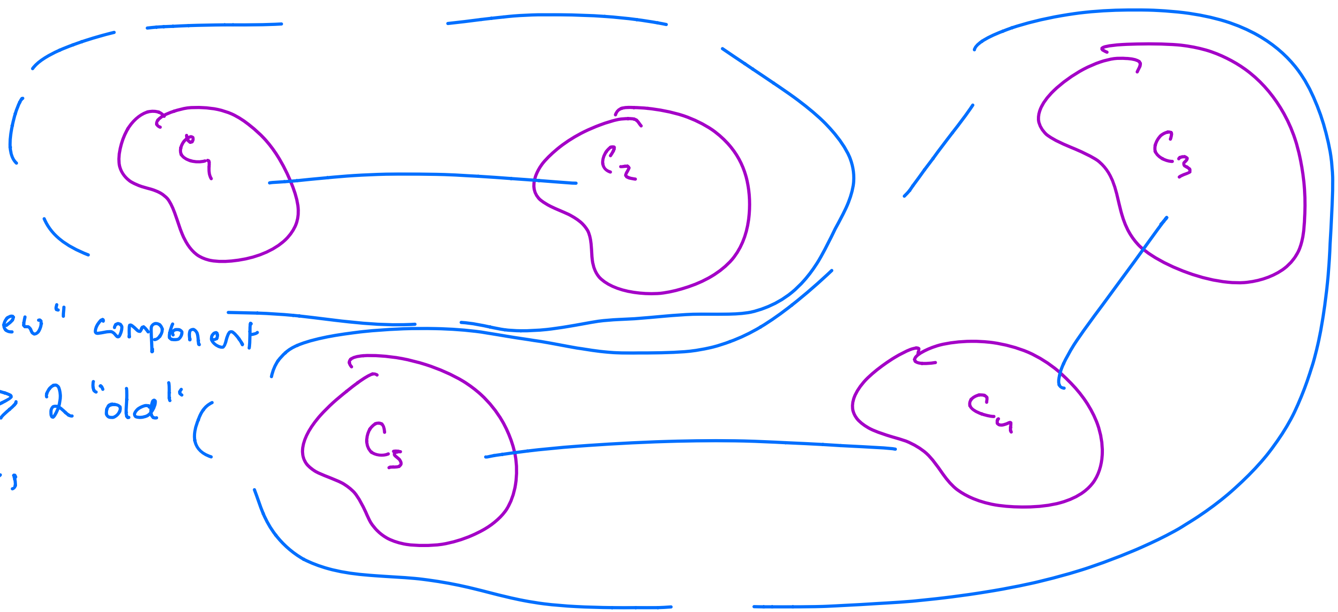
```
FindSafeEdges(G,T):
    find connected components C_1,…,C_k    // Using BFS/DFS
    let L[v] be the component of node v
    Let S[i] be the safe edge of C_i    // Initially ∅
    for each edge (u,v) in E:
        If L[u] ≠ L[v]:
            If w(u,v) < w(S[L[u]]):
                S[L[u]] = (u,v)
            If w(u,v) < w(S[L[v]]):
                S[L[v]] = (u,v)
    Return {S[1],…,S[k]}
```

May have duplicates

Borůvka's Algorithm (Running Time)

- **Claim:** every iteration of the main loop at least halves the number of connected components.

- If the claim is true, then # of iterations $\leq \lfloor \log_2(n) \rfloor$



Every "new" component contains ≥ 2 "old" components

Borůvka's Algorithm (Running Time)

- **Borůvka**
  - Let $T = \emptyset$
  - Repeat until T is connected:
    - Let $C_1, \ldots, C_k$ be the connected components of (V, T)
    - Let $e_1, \ldots, e_k$ be the safe edge for the cuts $C_1, \ldots, C_m$
    - Add $e_1, \ldots, e_k$ to T

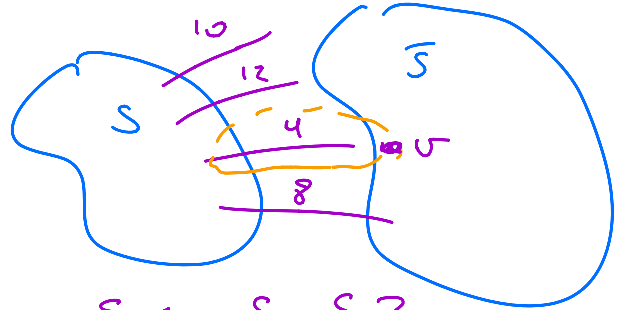
(V, E) is connected, so $m \geq n-1$

$n+m \leq 3m+1 = O(m)$

- Running Time:
  - How long to find safe edges? $O(n+m)$ per iteration
  - How many times through the main loop? $O(\log(n))$

Time: $\underline{O(m \log(n))}$

Prim's Algorithm

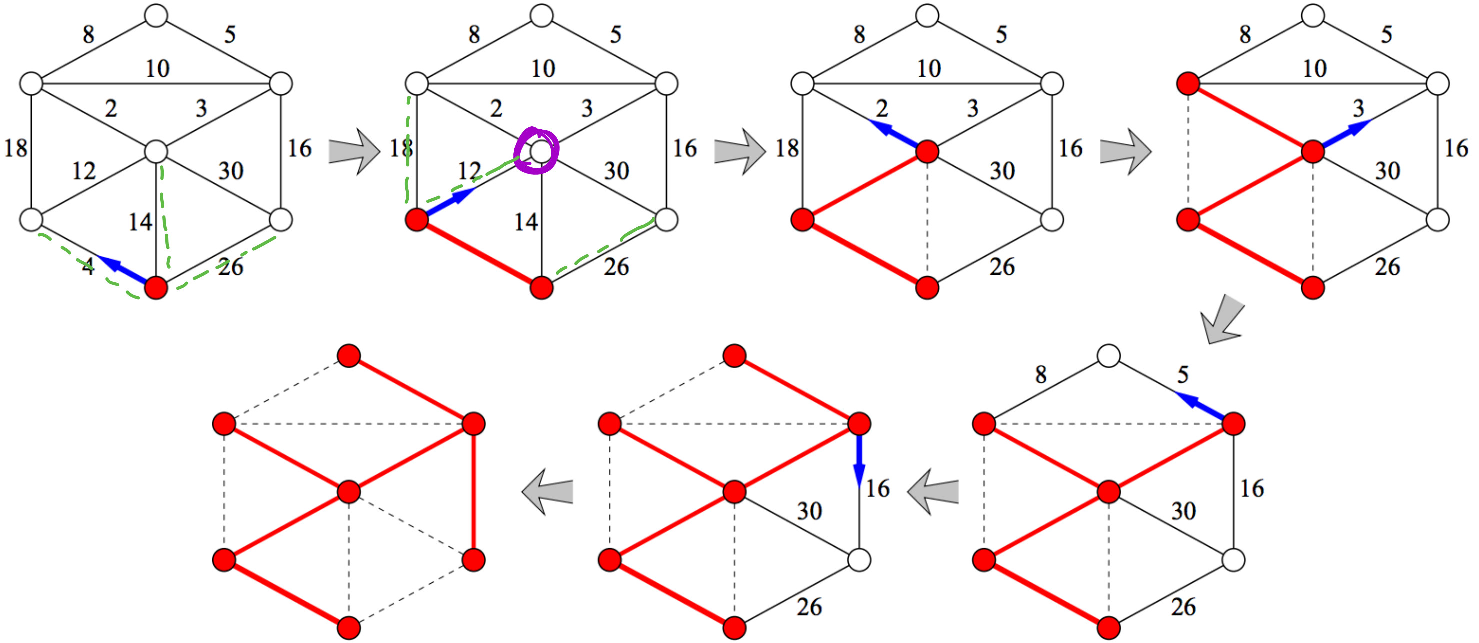


$S \leftarrow S \cup \{v\}$

- **Prim Informal**
  - Let $T = \emptyset$
  - Let s be some arbitrary node and $S = \{s\}$
  - Repeat until $S = V$
    - Find the cheapest edge $e = (u, v)$ cut by S. Add e to T and add v to S

- **Correctness:** every edge we add is safe

Prim's Algorithm



Prim's Algorithm

Time : $O((n+m)\log(n))$

$= O(m \log(n))$

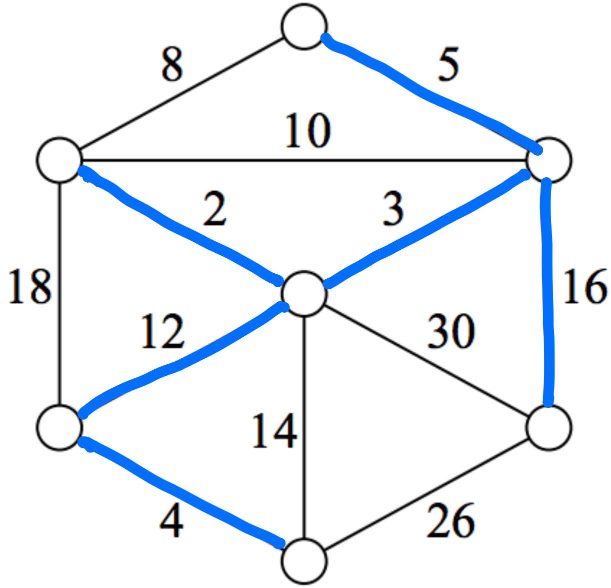
```
Prim(G=(V,E))
    let Q be a priority queue storing V
        value[v] ← ∞, last[v] ←⊥
        value[s] ← 0 for some arbitrary s
    while (Q ≠ ∅):
        u ← ExtractMin(Q)   ← n ExtractMin
        for each edge (u,v) in E:
            if v ∈ Q and w(u,v) < value[v]:
                DecreaseKey(v,w(u,v))   ← m DecreaseKey
                last[v] ← u

    T = {(1,last[1]),…,(n,last[n])} (excluding s)
    return T
```

Kruskal's Algorithm

- **Kruskal's Informal**
  - Let $T = \emptyset$
  - For each edge e in ascending order of weight:
    - If adding e would decrease the number of connected components add e to T

- **Correctness:** every edge we add is safe

Kruskal’s Algorithm



Implementing Kruskal's Algorithm

- **Union-Find**: group items into components so that we can efficiently perform two operations:
  - Find(u): lookup which component contains u
  - Union(u,v): merge connected components of u,v

- Can implement **Union-Find** so that
  - Find takes $O(1)$ time
  - Any k Union operations takes $O(k \log k)$ time

Kruskal's Algorithm (Running Time)

- **Kruskal's Informal**
  - Let $T = \emptyset$
  - For each edge e in ascending order of weight:
    - If adding e would decrease the number of connected components add e to T

- Time to sort:
- Time to test edges:
- Time to add edges:

Comparison

- **Can compute MST in time $\boldsymbol{O(m \log m)}$**

- **Boruvka's Algorithm:**
  - Only algorithm worth implementing
  - Low overhead, can be easily parallelized
  - Each iteration takes $O(m)$, very few iterations in practice

- **Prim's/Kruskal's Algorithms:**
  - Reveal useful structure of MSTs
  - Templates for other algorithms