CS3000: Algorithms & Data
Jonathan Ullman

Lecture 16:
- Minimum Spanning Trees

Nov 2, 2018

Minimum Spanning Trees

Network Design

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

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

Minimum Spanning Trees (MST)

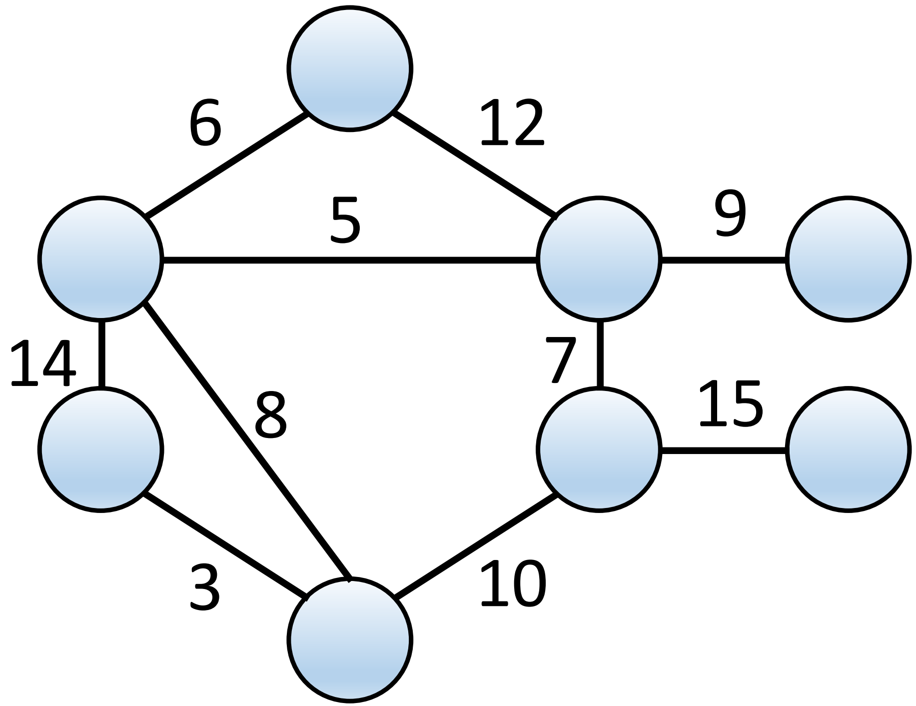
- **Input:** a weighted graph $G = (V, E, \{w_e\})$
  - Undirected, connected, weights may be negative
  - All edge weights are distinct (makes life simpler)

- **Output:** a spanning tree T of minimum cost
  - A spanning tree of G is a subset of $T \subseteq E$ of the edges such that (V, T) forms a tree (connected, no cycles)
  - Cost of a spanning tree T is the sum of the edge weights

$$\text{cost}(T) = \sum_{e \in T} w(e)$$

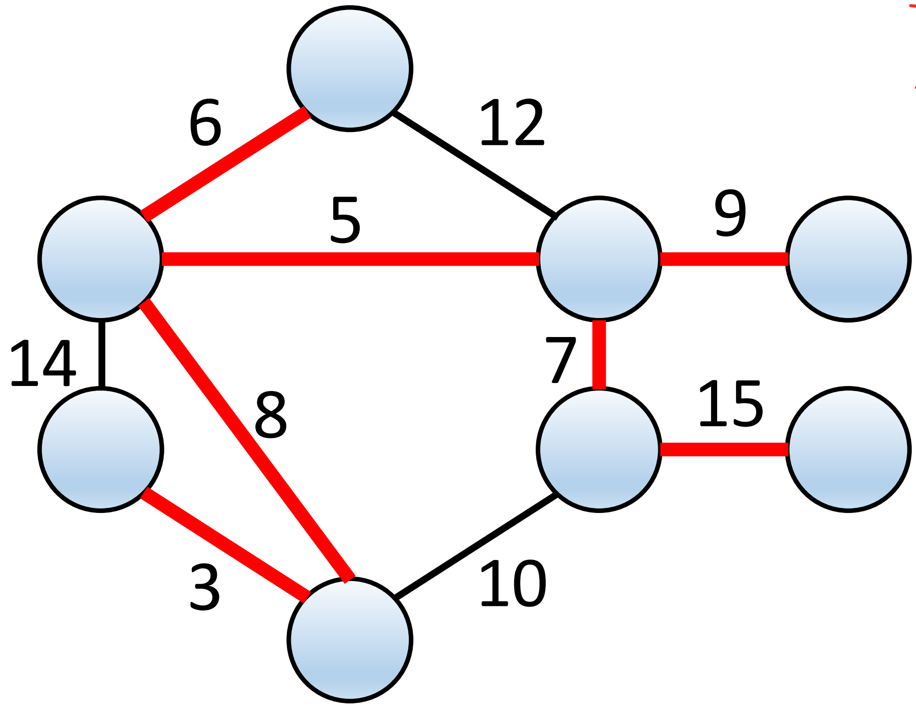
- MST: $T^* \in \underset{\text{trees } T}{\operatorname{argmin}} \ \text{cost}(T)$

Minimum Spanning Trees (MST)



Minimum Spanning Trees (MST)

cost (T) = 3 + 5 + 6 + 7
+ 8 + 9 + 15
= ???

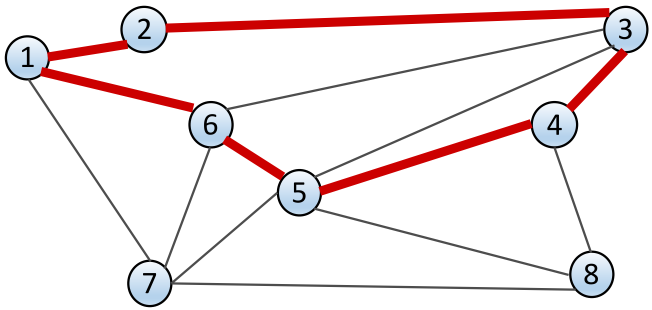


MST Algorithms

- 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

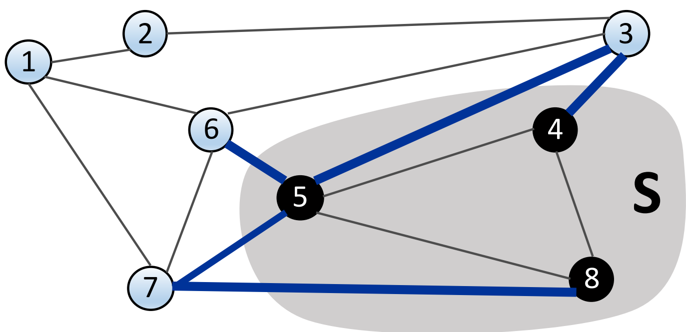
Cycles and Cuts

- **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 subset of nodes S



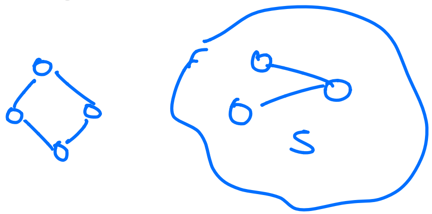
Cut S = {4, 5, 8}
Cutset = (5,6), (5,7), (3,4), (3,5), (7,8)

$\text{Cutset}(S) = \{ (u,v) \in E : u \in S, v \notin S \}$

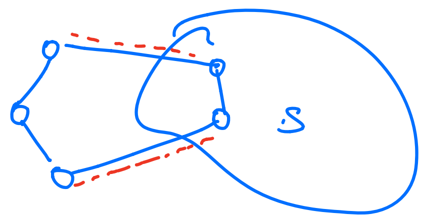
Cycles and Cuts

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

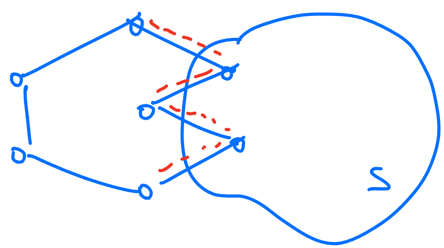
$|C \cap S| = 0$



$|C \cap S| = 2$



$|C \cap S| = 4$



If I walk around a cycle, I cross the cut an even number of times.

Properties of MSTs

Recall edge wts are distinct

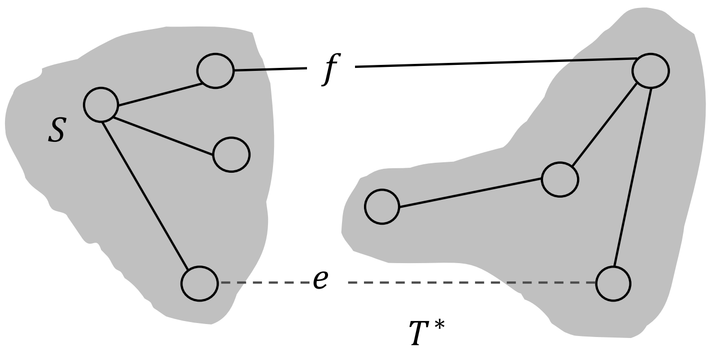
- **Cut Property:** Let S be a cut. Let e be the minimum weight edge cut by S. Then the MST T^* contains e
  - We call such an e a safe edge

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

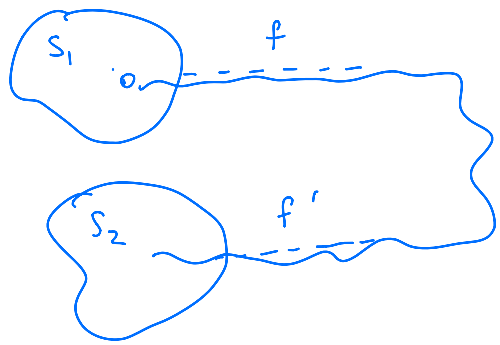
Proof: (By contradiction)



- Suppose T^* is an MST that does not contain e
- There is some edge f in both the cutset S and in T^* (or else T^* is not connected)
- By assumption $w(f) > w(e)$ (all wts distinct)
- $cost(T^* - \{f\} + \{e\}) < cost(T^*)$

- But $T^* - \{f\} + \{e\}$ is a spanning tree
  - S is connected
  - S^c is connected
  - e connects S to S^c
- But then T^* was <u>not</u> an MST, contradiction. $\square$

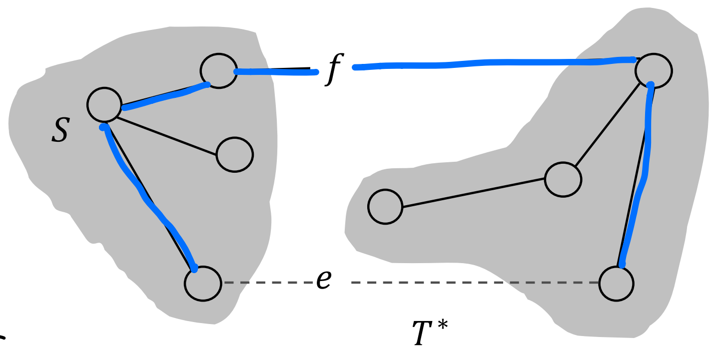
Why is S connected by $T^* - \{f\}$?



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)

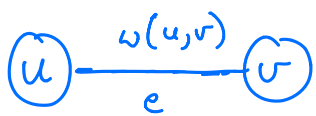


- Suppose T^* is an MST
- Removing f disconnects the graph into two components, S and S^c
- $f \in \text{Cutset}(S)$, $|\text{Cutset}(S)|$ is even $\therefore$ there is some edge e in both $\text{Cutset}(S)$ and in C
- $wt(e) < wt(f)$ (edge wts are distinct)

- $\text{cost}(T^* - \{f\} + \{e\}) < \text{cost}(T^*)$
- $T^* - \{f\} + \{e\}$ is a spanning tree (see cut property)
- But then T^* is not an MST, contradiction $\square$

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 e is the edge with the largest weight, then e is never in the MST T^*



e is the min wt edge for the cut $S = \{u\}$

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 e is the edge with the largest weight, then e is never in the MST T^*

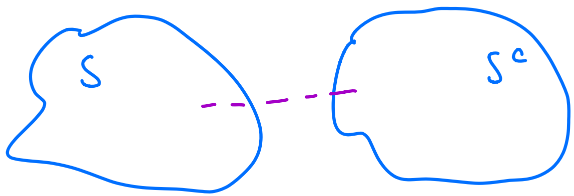
The max wt edge may not lie on any cycle.

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

If T is not connected then it has $\geq$ two connected components



Cutset(S) contains ≥ 1 edge in E

$\Rightarrow$ exists $\geq$ safe edge in the graph

Borůvka's Algorithm

- **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_k
        - Add $e_1, \ldots, e_k$ to T

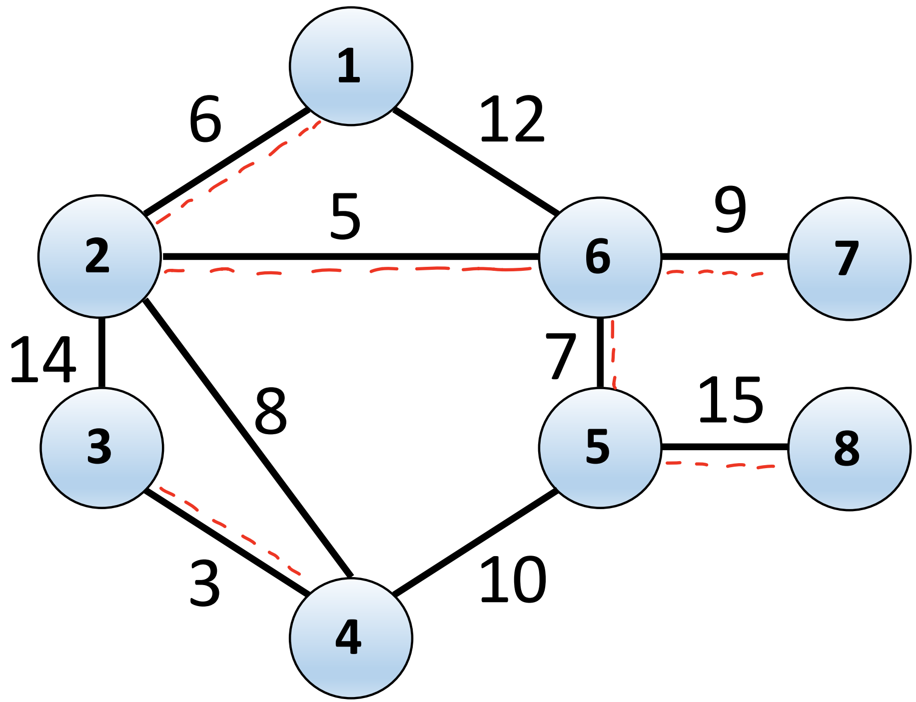
- **Correctness:** every edge we add is safe

Borůvka's Algorithm

Label Connected Components

(for the graph (V, T))

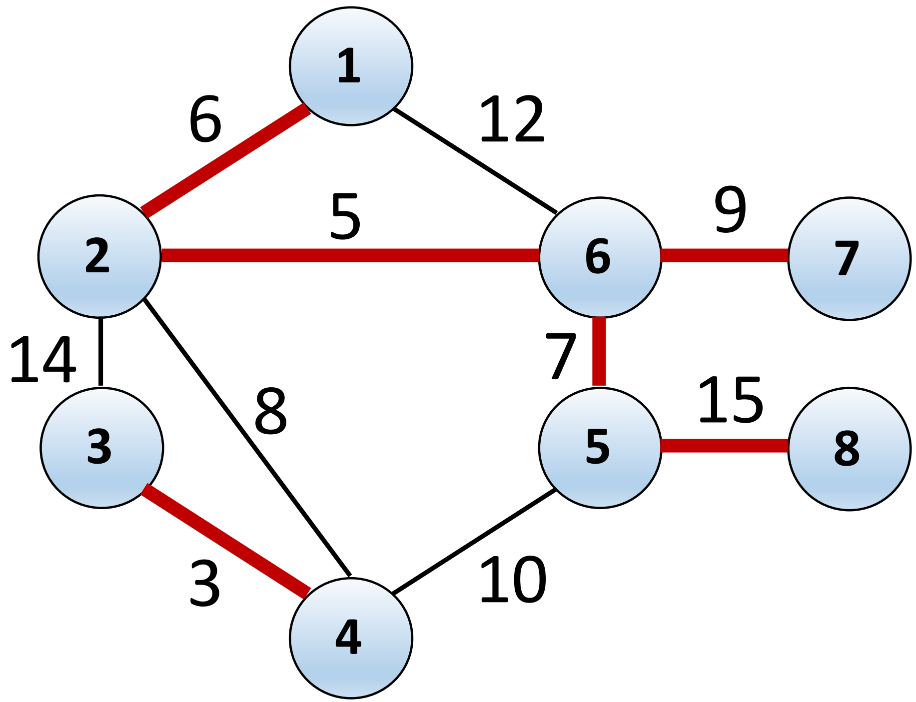
Initially $T = \emptyset$



Borůvka's Algorithm

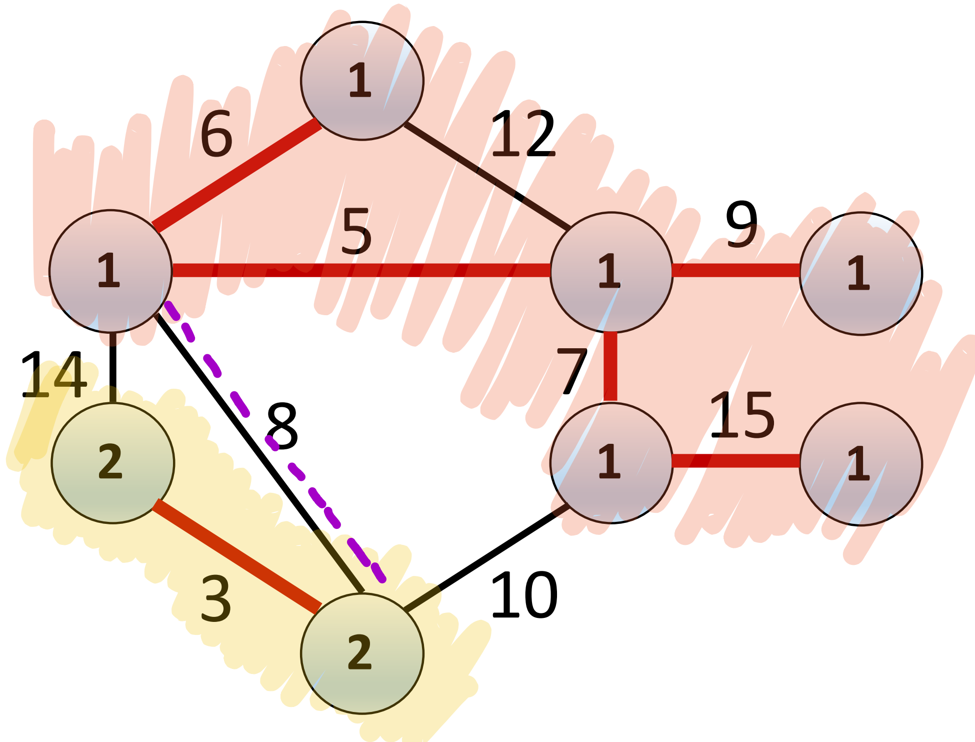
Add Safe Edges

T = { red edges }



Borůvka's Algorithm

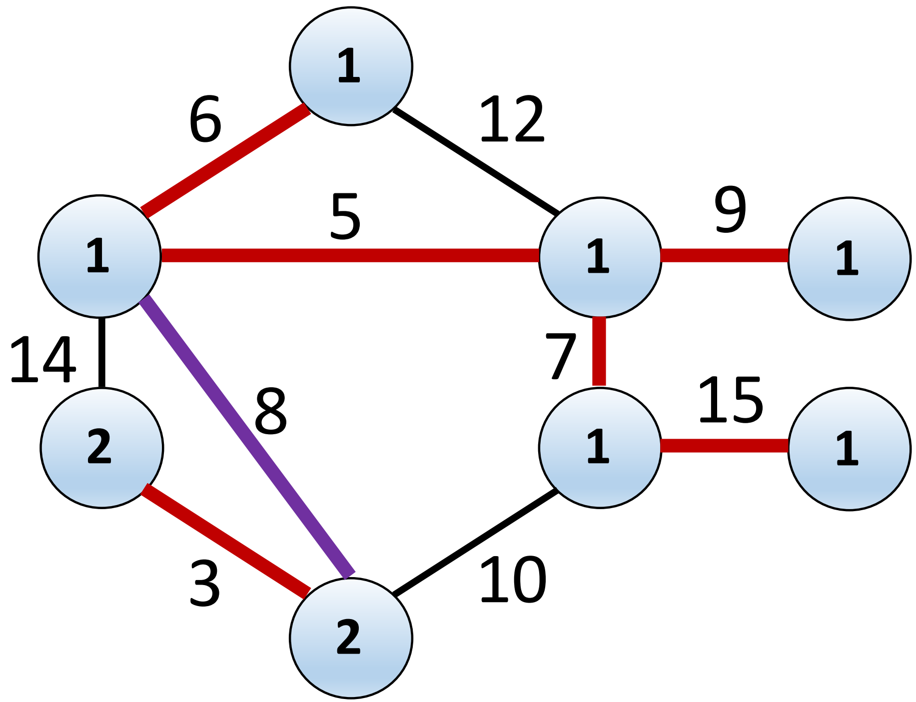
Label Connected Components



Borůvka's Algorithm

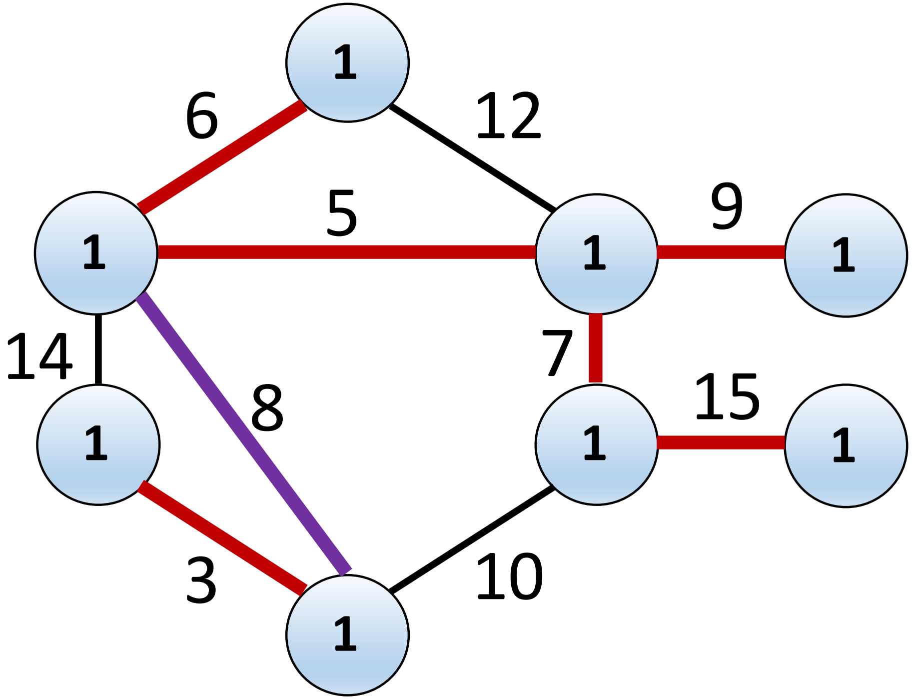
Add Safe Edges

T= { red edges + purple edge }



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)
    - 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? $O(m)$ time per iteration
  - How many times through the main loop? $O(\log n)$ iterations

- Running time $O(m \log n)$

Borůvka's Algorithm (Running Time)

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

Fact: Can find all safe edges in time $O(m)$

Borůvka's Algorithm (Running Time)

- **Claim:** every iteration of the main loop halves the number of connected components.
- $\Longrightarrow$ We do at most $\lceil \log_2 n \rceil$ iterations of the main loop.
- Proof: In iteration $i+1$, every component contains at least 2 of the components from iteration i

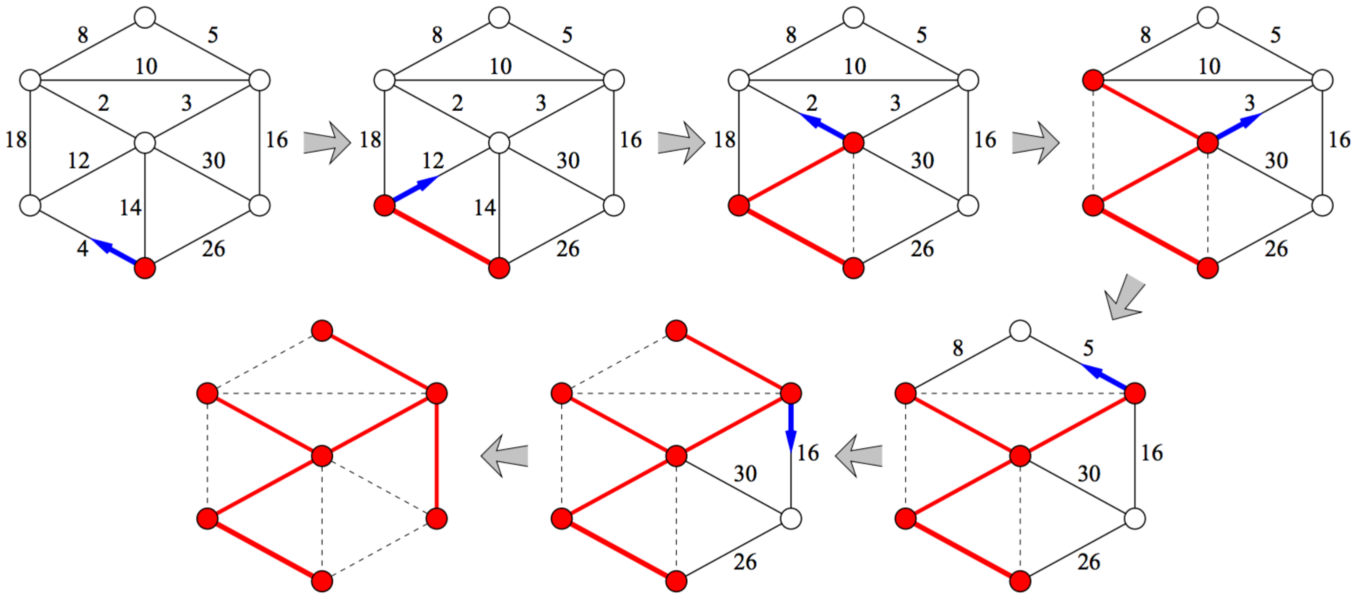
$$\Longrightarrow (\#\text{ of components in } i+1) \leq \frac{1}{2}(\#\text{ of components in } i) \qquad \square$$

Prim's Algorithm

- **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

value[u] = minimum wt of an edge from u to S

```
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 O(n log n)
        for each edge (u,v) in E:
            if v ∈ Q and w(u,v) < value[v]:    m DecreaseKey
                DecreaseKey(v,w(u,v))          O(m log n)
                last[v] ← u

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

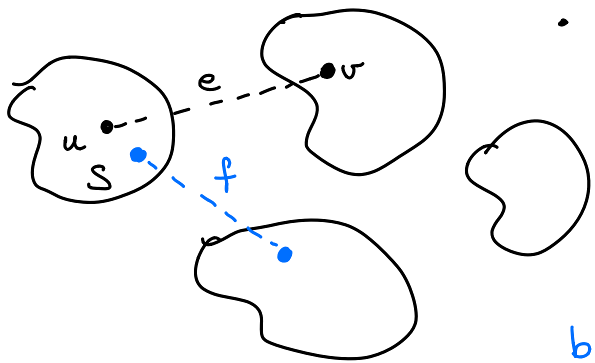
Running Time: $O(m \log n)$

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

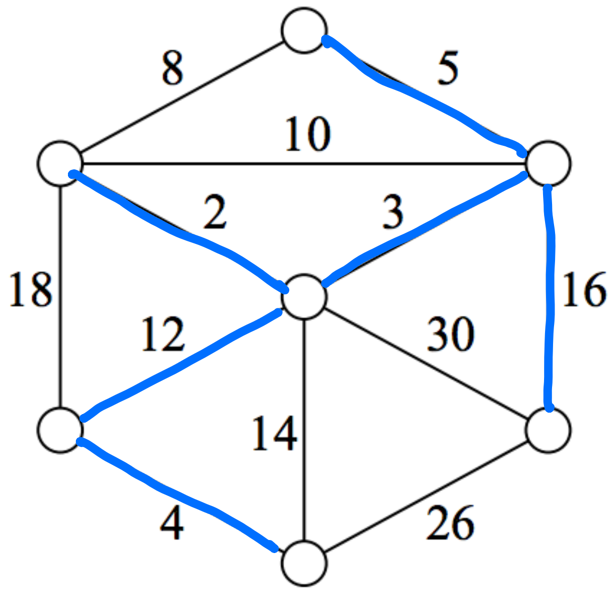
- **Correctness:** every edge we add is safe

Consider the graph when we add e



- $e \in$ Cutset(S)
- We've already considered all f s.t. $wt(f) < wt(e)$
- If $f \in$ Cutset(S), then it bridges two components
- But, we didn't add f, contradiction!

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 } amoitized running time

- Naive Implementation: find taks $O(1)$, union takes $O(n)$

Implementing Union Find:

① Maintain an array comp[1:n] for the component of each i

1	2	3	4	5	6	7

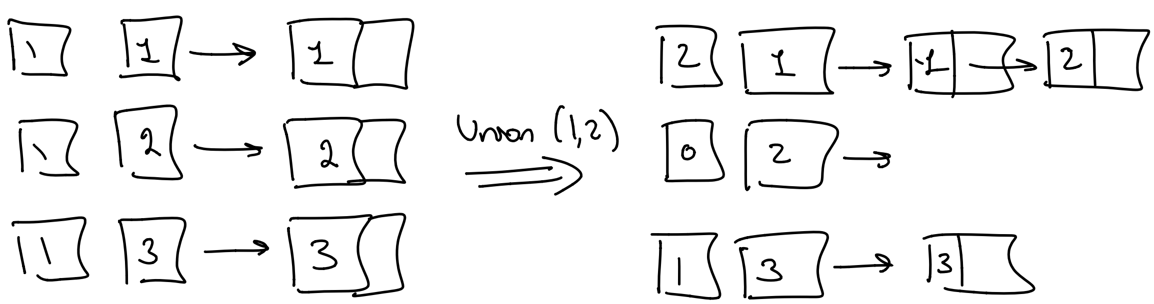
Find = $O(1)$, Union = $O(n)$

1	1	1	2	3	2	2

$\Downarrow$ Union (1,2)

1	1	1	1	3	1	1

② Maintain a list of each component's items, and sizes



Find = $O(1)$ Union = $O(n)$

Implementing Union Find:

(3) Always merge the smaller component into the bigger component. (Minimizes the # of updates needed)

Thm: For any k union operations, running time is $O(k \log k)$.

Pf:

- If we do k unions, only $2k$ total elts have to be "touched."
- How many times does each elt change component?
  - Each merge moves it to a component that is $\geq$ twice as large.
  - Max component size is $\leq 2k$

  $\Longrightarrow \log_2(2k)$ changes
- $(2k \text{ elements}) \times (\log_2(2k) \text{ changes per element})$

  $= O(k \log k)$ 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: $O(m \log m)$
- Time to test edges: $2m$ Find operations $= O(m)$ time
- Time to add edges: $n-1$ Union operations $= O(n \log n)$ time

Total Time $= O(m \log m + m + n \log n)$

$= O(m \log m)$

Comparison

- **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
  - Running time dominated by a single sort