Graph Optimization

a. Shortest Paths
    a. Dijkstra’s Algorithm
    b. Bellman-Ford
b. Minimum Spanning Trees

Network Design

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

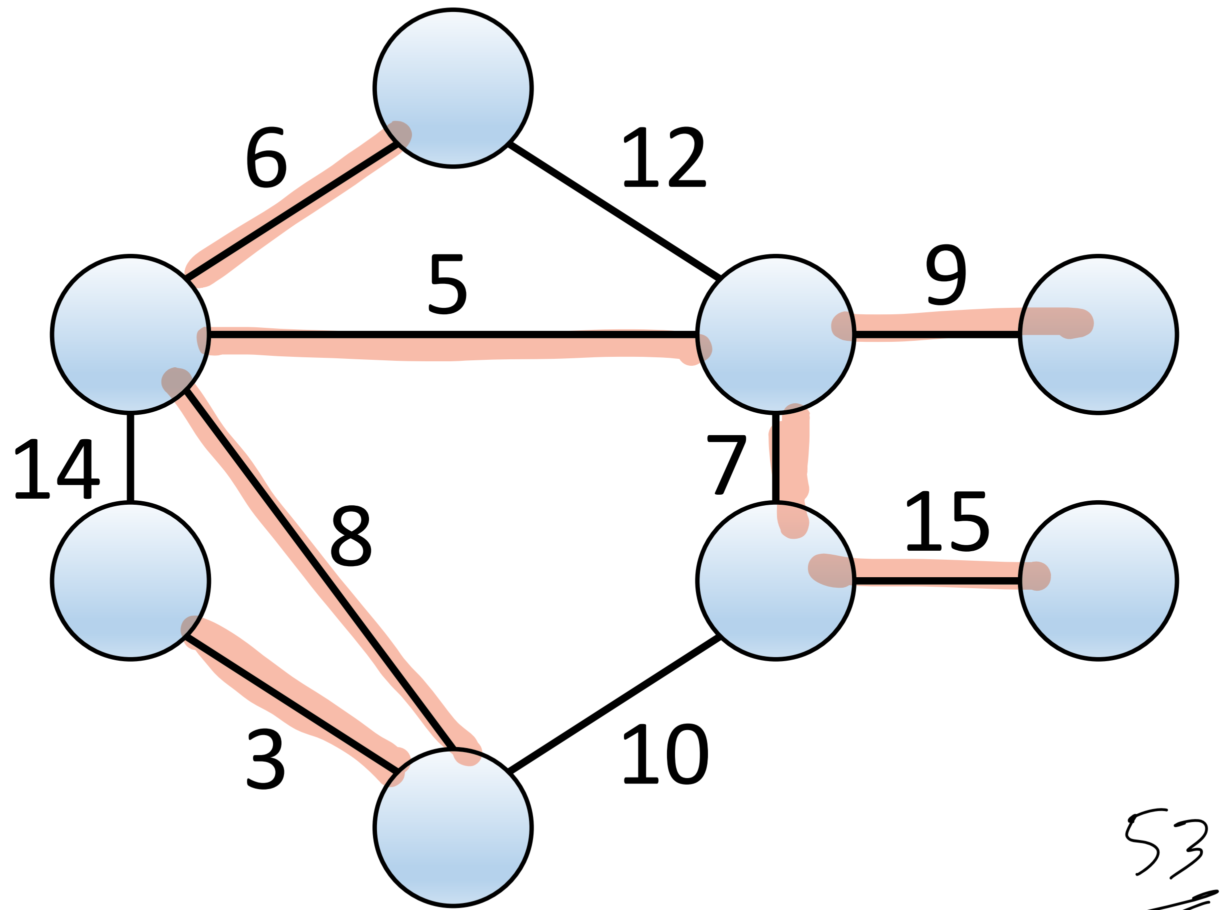
- Many variants of network design

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)

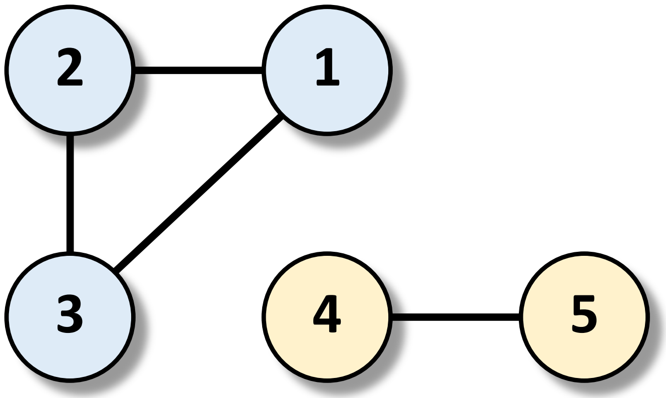
- **Output:** a spanning tree T of minimum cost
  - A spanning tree of G is a subset of $T \subseteq E$ of the edges such that (V, T) forms a tree (*what's a tree?* no cycles, connected)
  - Cost of a spanning tree T is the sum of the edge weights
    - Cost(T) = $\sum_{e \in T} w_e$
    - MST: $\operatorname{argmin}_{\text{spanning tree } T} \text{cost}(T)$

Minimum Spanning Trees (MST)



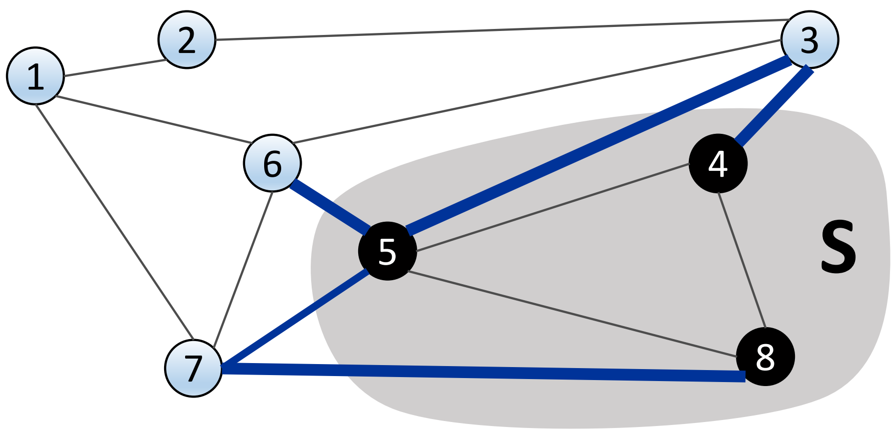
Connected Components

- **Connected component:** a maximal subset of vertices which are all connected in G



Cuts

- **Cut:** a subset of nodes S **Cutset:** edges w/ 1 endpoint in cut



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

an edge (u,v) is cut by S. $\longleftrightarrow$ (u,v) is in the cutset defined by S

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

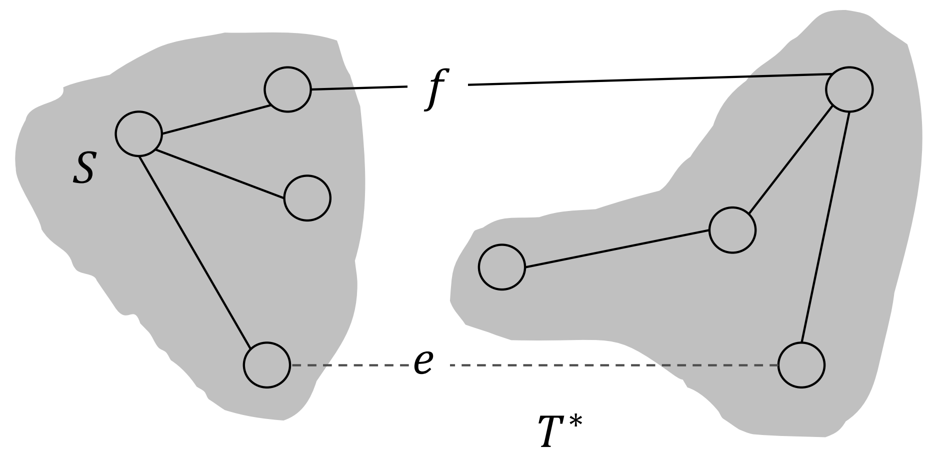
Proof of Cut Property

$cost(T) \to cost(T) - w_f + w_e$

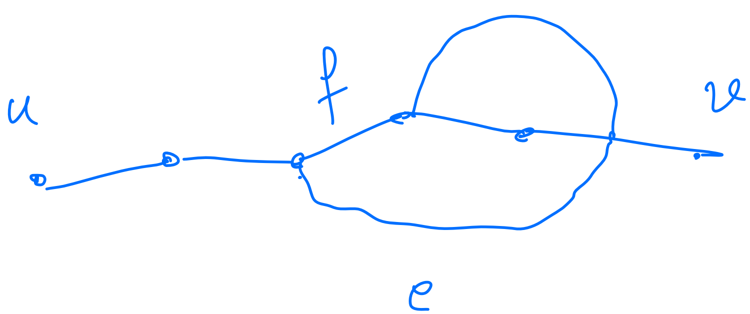
$w_e < w_f$

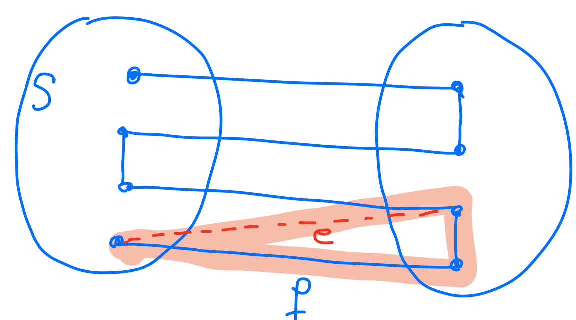
- **Cut Property:** Let S be a cut. Let e be the minimum weight edge cut by S. Then the MST T^* contains e

Proof by contradiction:



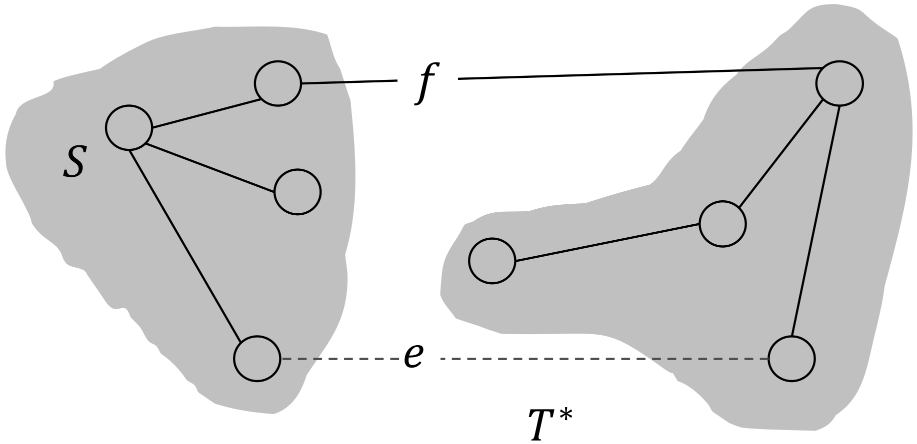
Assume e is not in the MST. Adding it to the MST creates a cycle C with at least one other edge f in the cut set. Replacing f with e in this MST gives us a smaller spanning tree hence the contradiction.





Proof of Cut Property

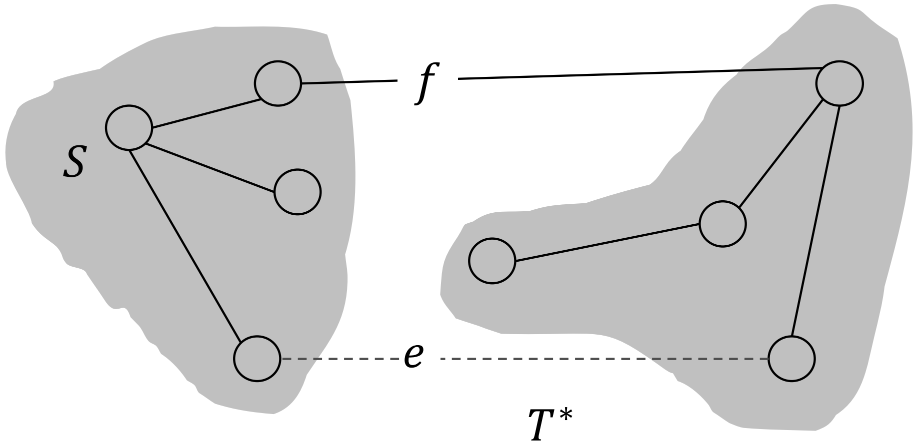
Why does f exist?



Why doesn't replacing f with e create new cycle?

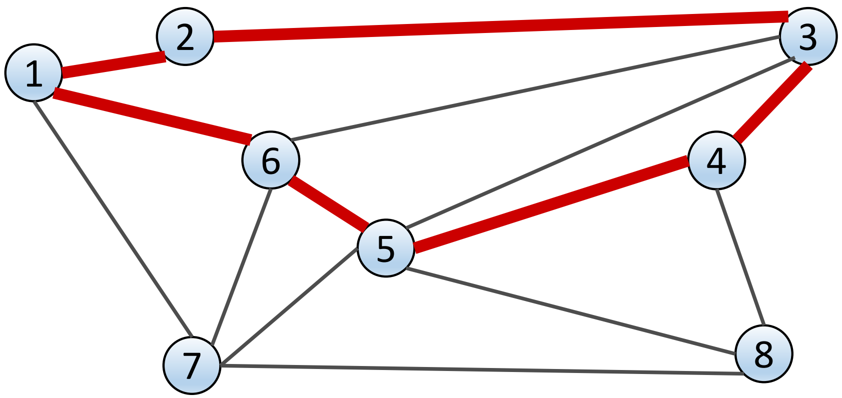
Proof of Cut Property

Why does replacing f with e keep the graph connected?



Cycles

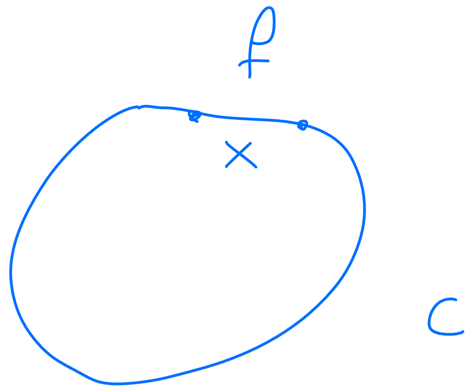
- **Cycle:** a set of edges $(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_1)$



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

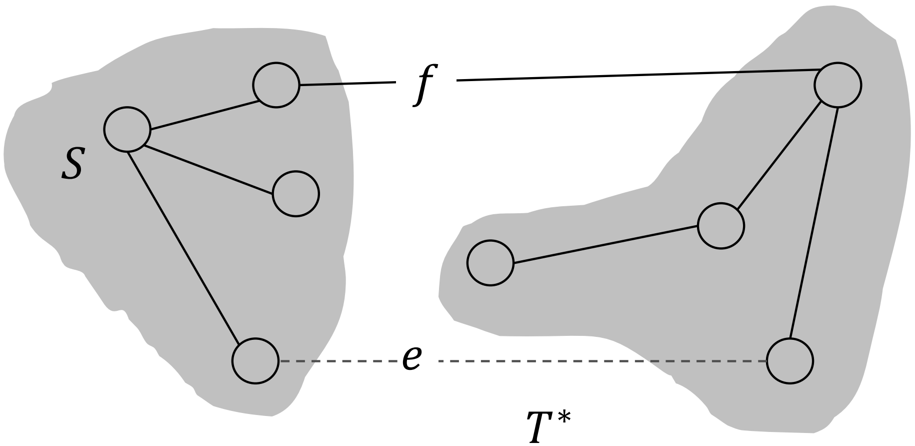
Cycle Property

- **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



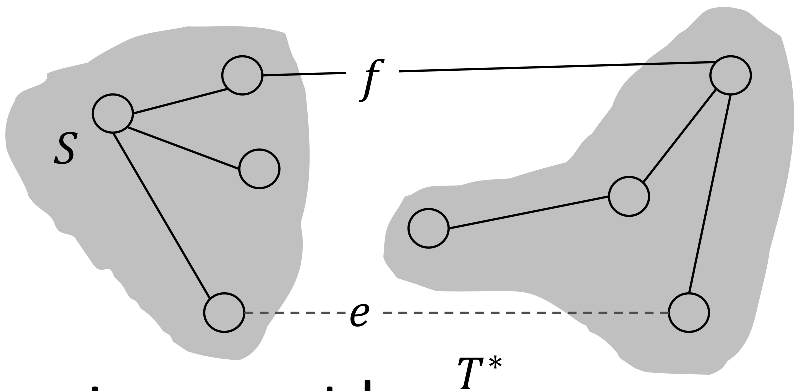
Proof of Cycle Property

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



Proof of Cycle Property

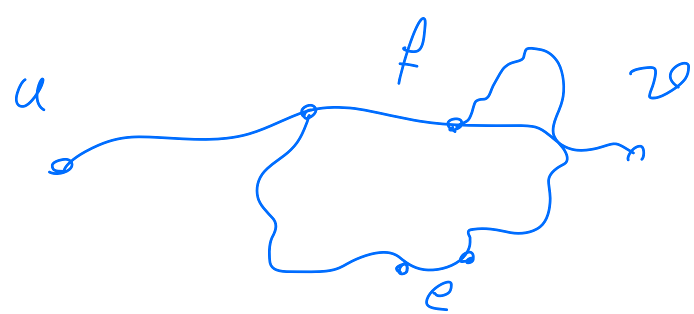
- **Cycle Property:** Let C be a cycle. Let f be the max weight edge in C. The MST T^* does not contain f.



Proof by contradiction:

Assume f is in the MST.

Let S be one of the connected components we get by removing f from this MST. There is at least one other edge e from cycle C in cutset of S. Replacing f with e in this MST gives us a smaller spanning tree hence the contradiction.



Ask the Audience

- Assume G has distinct edge weights
- **True/False?** If $e \underset{}{=} (u,v)$ is the edge with the smallest weight, then e is always in the MST T^*

True. Let $S = \{u\}$. By the cut property, e must belong to MST.

- **True/False?** If e is the edge with the largest weight, then e is never in the MST T^*

False.

MST Algorithms

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

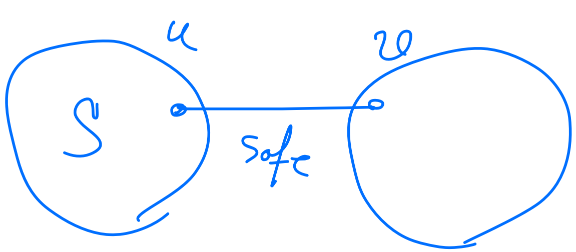
Graph Optimization

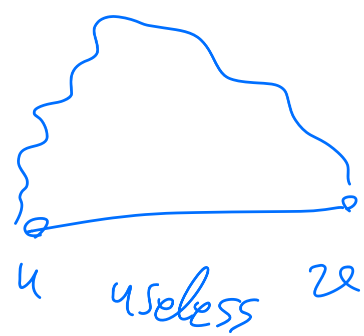
a. Shortest Paths
   a. Dijkstra's Algorithm
   b. Bellman-Ford
b. Minimum Spanning Trees
   a. Kruskal's

Kruskal's Algorithm

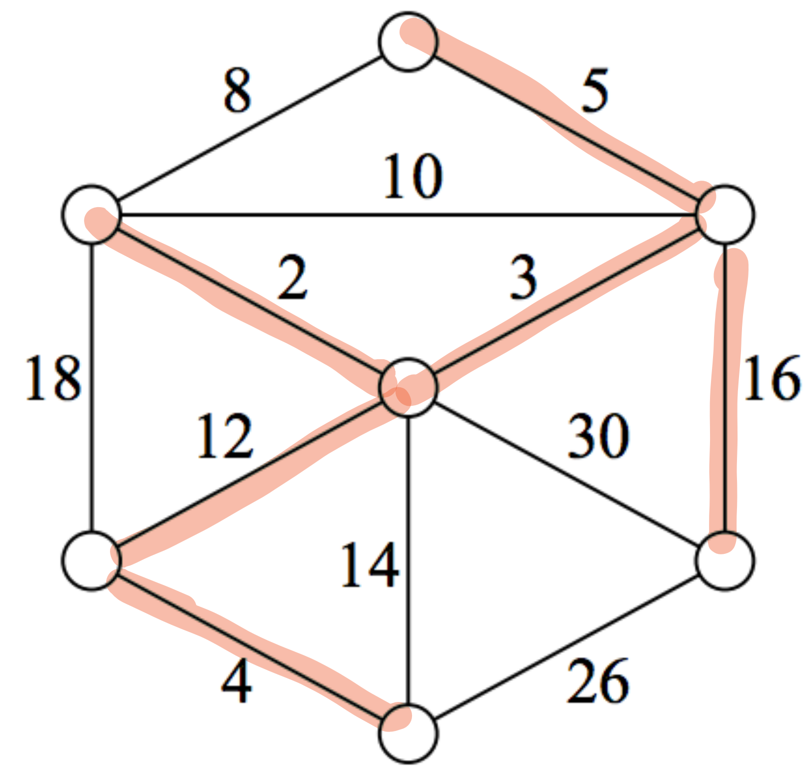
- **Kruskal's Informal**
  - Let $T = \emptyset$
  - For each edge e in ascending order of weight: $m \log n$ time
    - If adding e would decrease the number of connected components add e to T

- **Correctness:** every edge we add is safe and every edge we don't add is useless





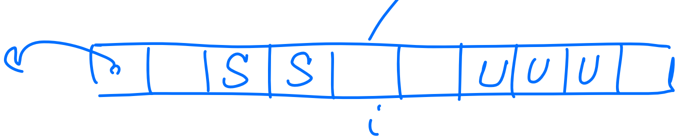
Practice Kruskal's Algorithm

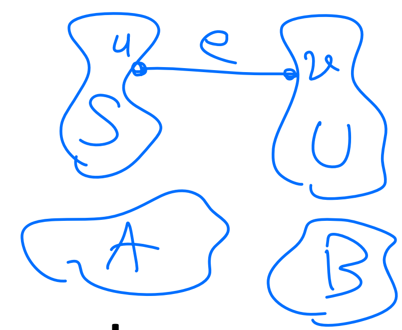


Implementing Kruskal's Algorithm

CC of vertex i

CC of vertex 1





- **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
- Naïve **Union-Find:** Kruskal's RT using naive Union Find $\underbrace{O(m\log n)}_{\text{sorting}} + \underbrace{O(m)}_{\text{calling Find}} + \underbrace{O(n^2)}_{\text{calling Union } n-1 \text{ times}}$

  Take an array mapping each vertex u to the ID of its CC.

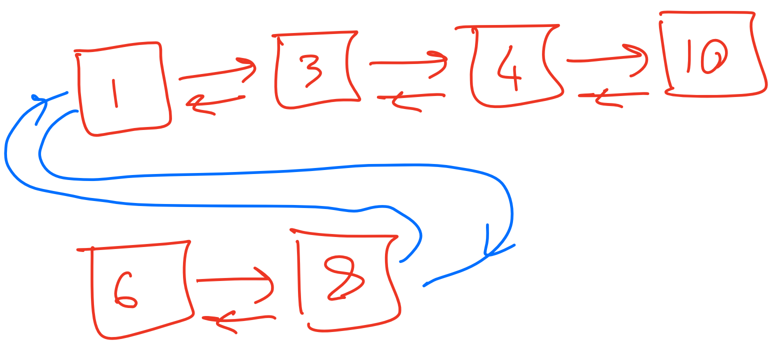
  Find takes $O(1)$ time. Union takes $O(n)$ time.
- Can implement **Union-Find** so that
  - Find takes $O(1)$ time
  - Any k Union operations takes $O(k\log k)$ time

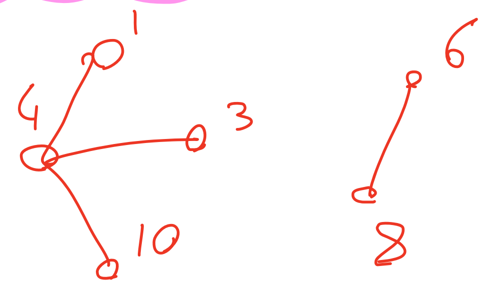
This would improve Kruskal's RT to $O(m\log n) + O(m) + O(n\log n)$
$= O(m\log n)$.

as in the naive implementation

Fast Union-Find

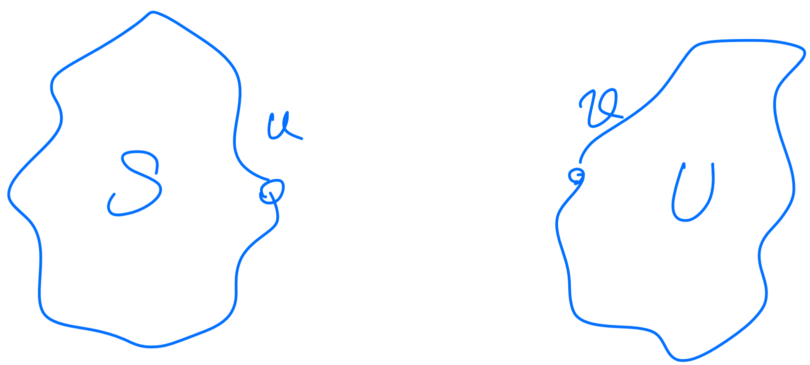
- Use an *array* for current component of each vertex and a *linked list* for items in each component, and keep size of each component (always union smaller into larger)





Fast Union-Find

- Use an *array* for current component of each vertex and a *linked list* for items in each component, and keep size of each component (always union smaller into larger)



It takes $|S|$ time to change the labels in S to U.

Because the # of vertices on which union is called is $\leq 2K$.

- **1.** Largest component has size $O(K)$
- **2.** Every time an item changes component, its new component is twice the size of its old component
- **3.** No item changed components more than $O(\lg K)$ times
- **Total time:** $O(K \lg K)$.

Kruskal's Algorithm (Running Time)

- **Kruskal's:**
  - 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 ("test e")

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

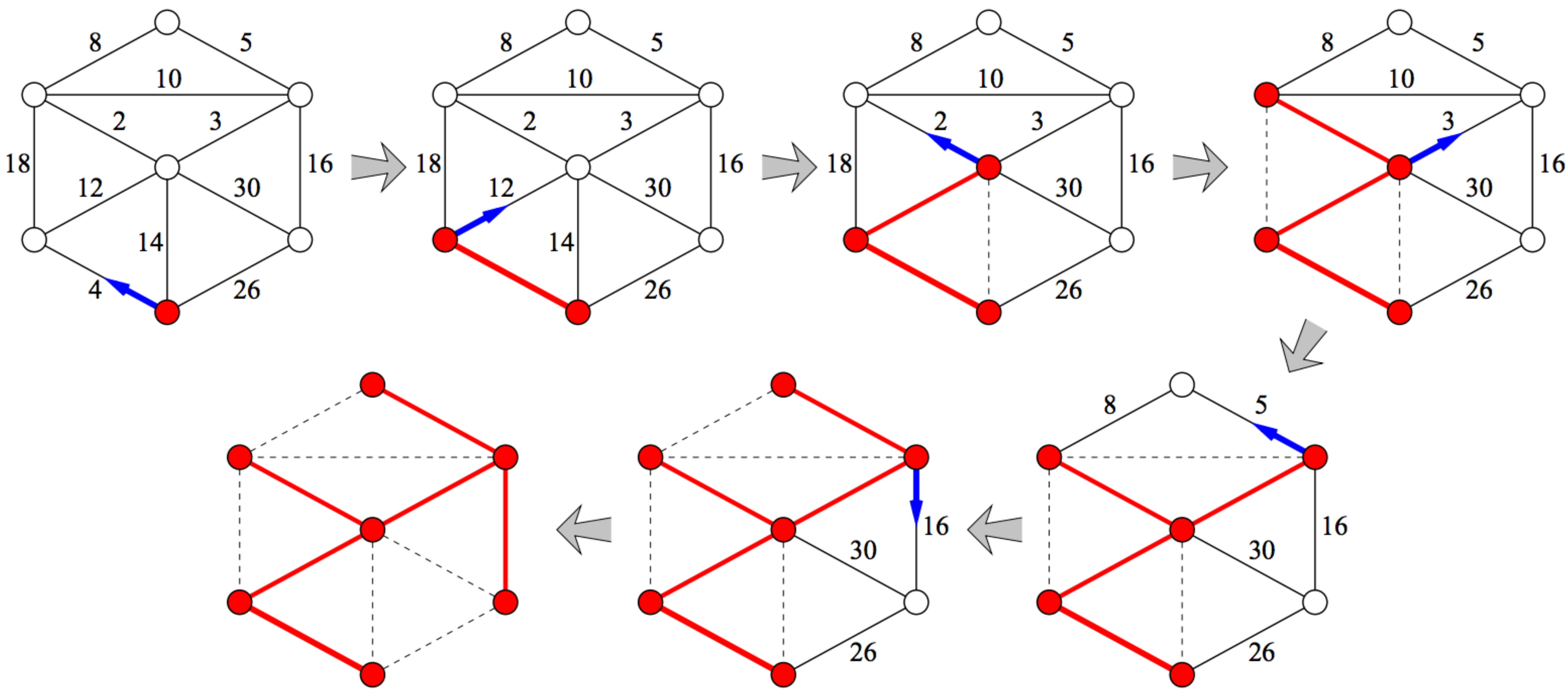
Graph Optimization

a. Shortest Paths
    a. Dijkstra’s Algorithm
    b. Bellman-Ford
b. Minimum Spanning Trees
    a. Kruskal’s Algorithm
    b. Prim’s Algortithm

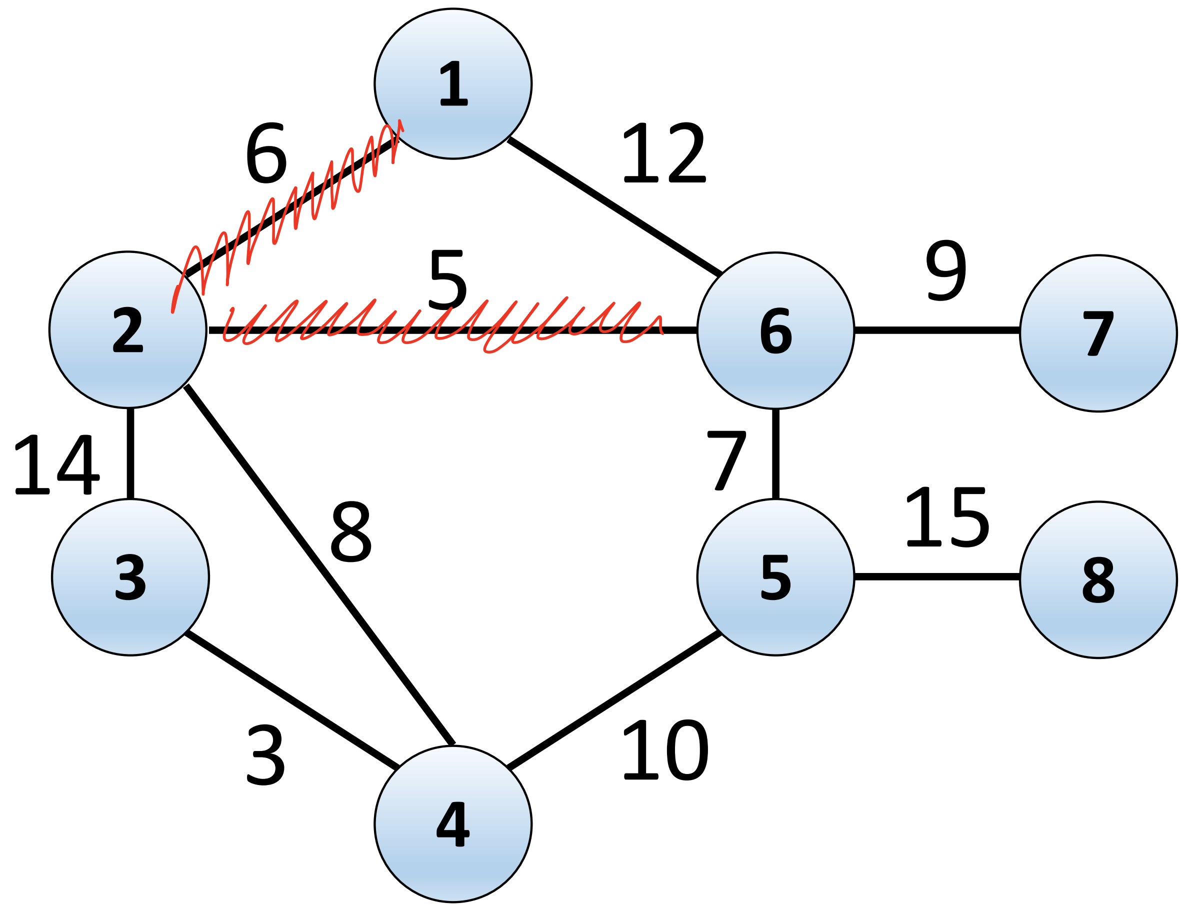
Prim's Algorithm

- **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 and *T* is spanning & connected (S is always connected)

Prim's Algorithm



Practice Prim's Algorithm



Prim’s Algorithm

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

$O(m \log n)$

Prim's vs Kruskal's

- **Prim's Algorithm:**
  - $O(m\log(n))$
  - Iteratively builds one connected component
  - Faster in practice on dense graphs

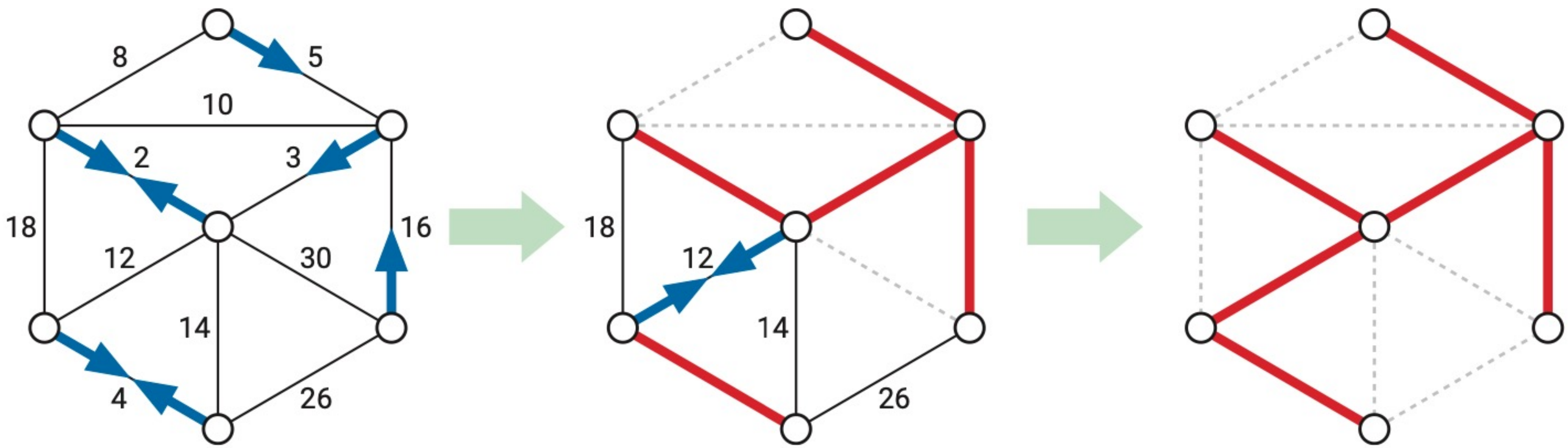
$O(m \lg^* n)$

- **Kruskal's Algorithm:**
  - $O(m\log(n))$
  - Maintains multiple connected components simultaneously
  - Faster in practice on sparse graphs

Borůvka's Algorithm

- **Borůvka's Algorithm (Informal)**

  Add **ALL** the safe edges and recurse.



1) Every iteration can be implemented in $O(m)$ time.

2) # of iterations is $O(\lg n)$. ← each step at least doubles size of smallest component

Borůvka's Algorithm

```
BORŮVKA(V, E):
  F = (V, ∅)
  count ← COUNTANDLABEL(F)
  while count > 1
      ADDALLSAFEEDGES(E, F, count)
      count ← COUNTANDLABEL(F)
  return F
```

```
ADDALLSAFEEDGES(E, F, count):
  for i ← 1 to count
      safe[i] ← NULL
  for each edge uv ∈ E
      if comp(u) ≠ comp(v)
          if safe[comp(u)] = NULL  or  w(uv) < w(safe[comp(u)])
              safe[comp(u)] ← uv
          if safe[comp(v)] = NULL  or  w(uv) < w(safe[comp(v)])
              safe[comp(v)] ← uv
  for i ← 1 to count
      add safe[i] to F
```