

# CS 473: Algorithms

Chandra Chekuri  
chekuri@cs.illinois.edu  
3228 Siebel Center

University of Illinois, Urbana-Champaign

Fall 2009

## Part I

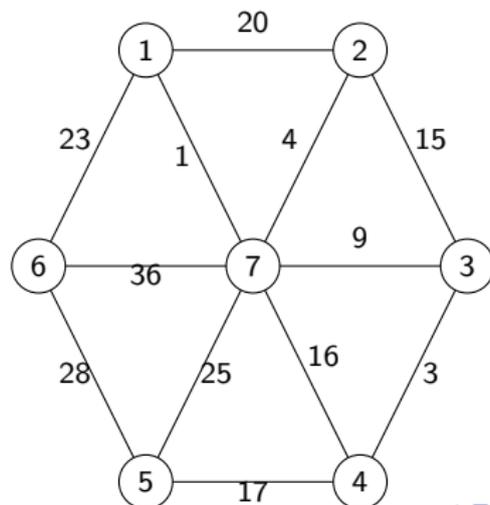
# Greedy Algorithms: Minimum Spanning Tree

# Minimum Spanning Tree

**Input** Connected graph  $G = (V, E)$  with edge costs

**Goal** Find  $T \subseteq E$  such that  $(V, T)$  is connected and total cost of all edges in  $T$  is smallest

- $T$  is the **minimum spanning tree** (MST) of  $G$

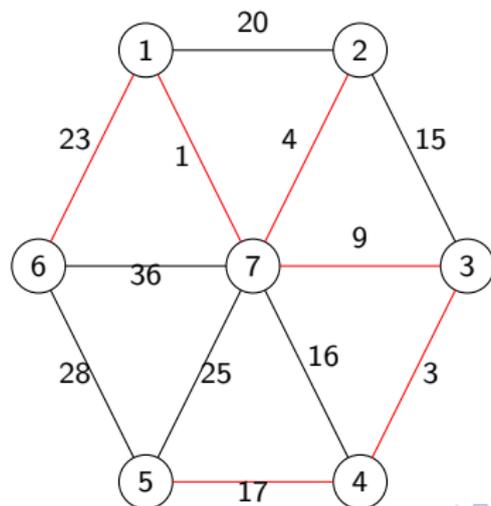


# Minimum Spanning Tree

**Input** Connected graph  $G = (V, E)$  with edge costs

**Goal** Find  $T \subseteq E$  such that  $(V, T)$  is connected and total cost of all edges in  $T$  is smallest

- $T$  is the **minimum spanning tree** (MST) of  $G$



# Applications

- Network Design
  - Designing networks with minimum cost but maximum connectivity
- Approximation algorithms
  - Can be used to bound the optimality of algorithms to approximate Travelling Salesman Problem, Steiner Trees, etc.
- Cluster Analysis

# Greedy Template

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty
    choose  $i \in E$ 
    if (i satisfies condition)
        add i to T
return the set T
```

**Main Task:** In what order should edges be processed? When should we add edge to spanning tree?

KA

PA

RD

# Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to  $T$  as long as they don't form a cycle.

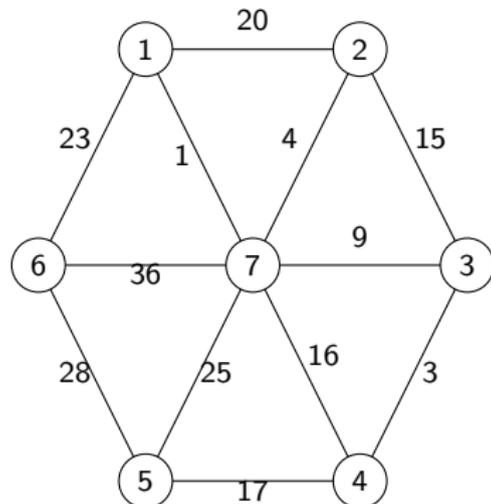


Figure: Graph  $G$

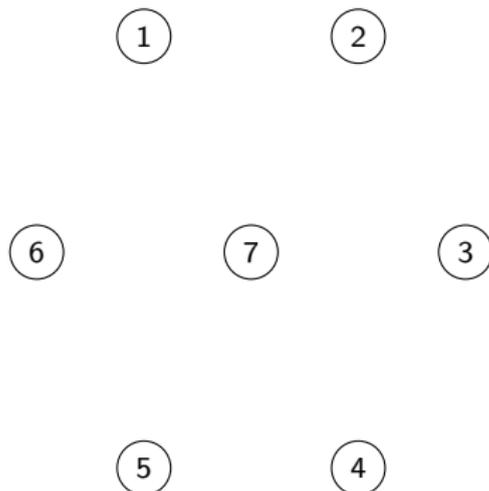


Figure: MST of  $G$

# Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to  $T$  as long as they don't form a cycle.

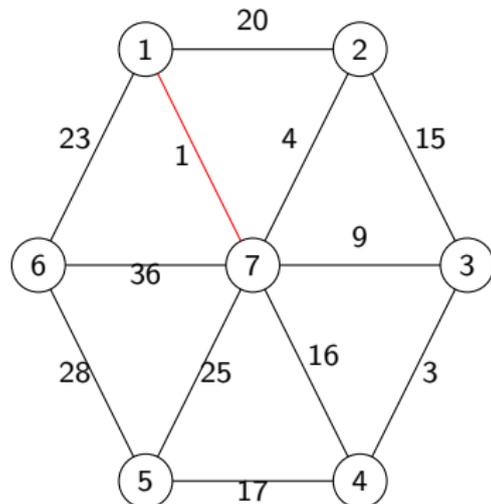


Figure: Graph  $G$

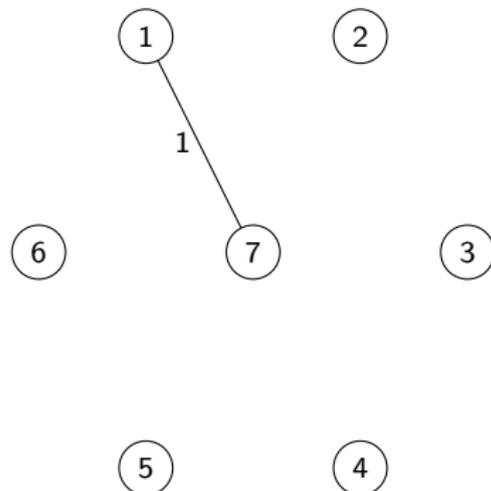


Figure: MST of  $G$

# Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to  $T$  as long as they don't form a cycle.

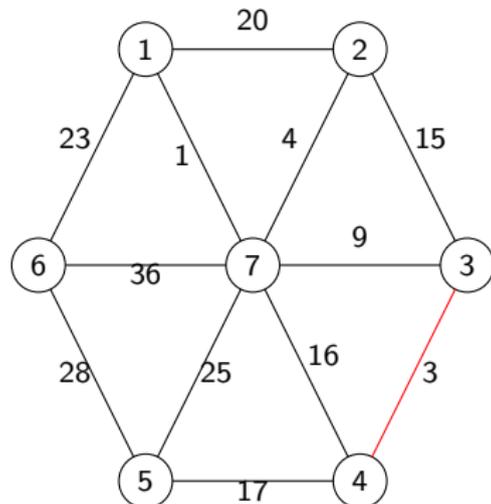


Figure: Graph  $G$

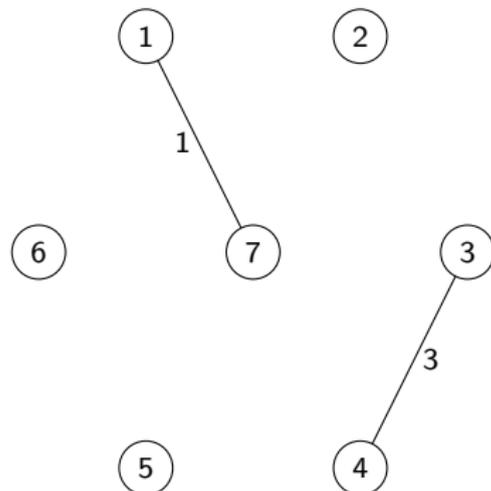


Figure: MST of  $G$

# Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to  $T$  as long as they don't form a cycle.

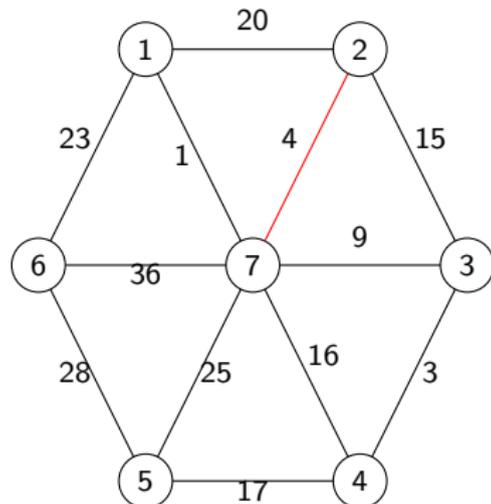


Figure: Graph  $G$

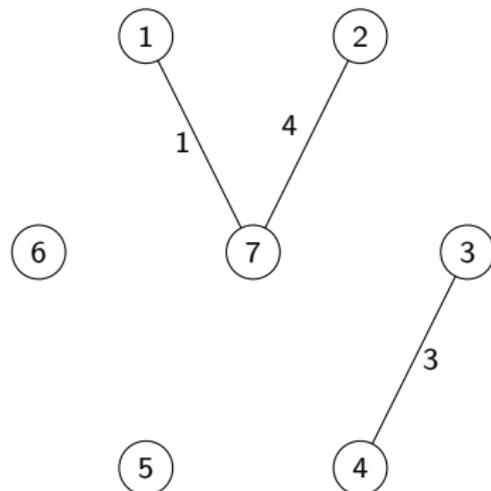


Figure: MST of  $G$

# Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to  $T$  as long as they don't form a cycle.

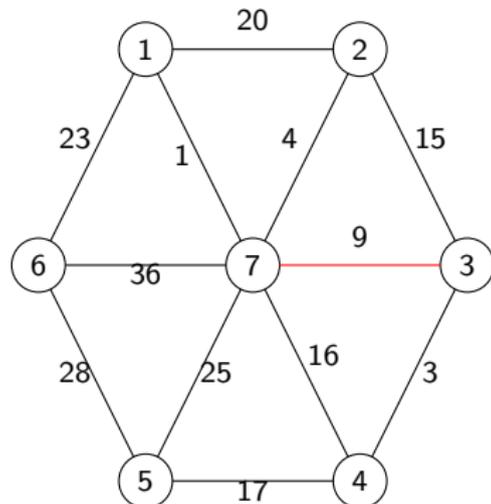


Figure: Graph  $G$

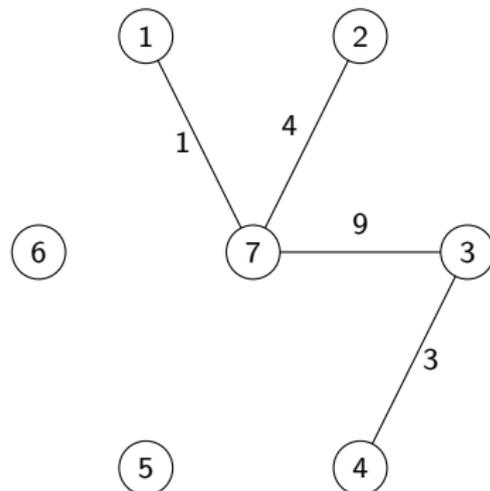


Figure: MST of  $G$

# Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to  $T$  as long as they don't form a cycle.

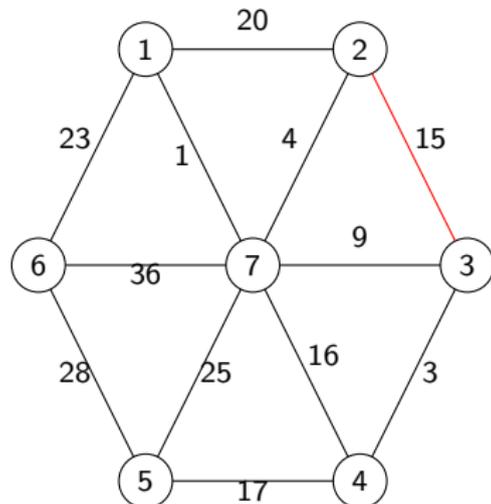


Figure: Graph  $G$

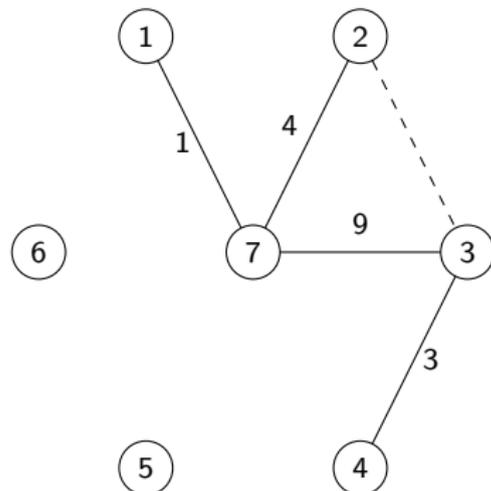


Figure: MST of  $G$

# Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to  $T$  as long as they don't form a cycle.

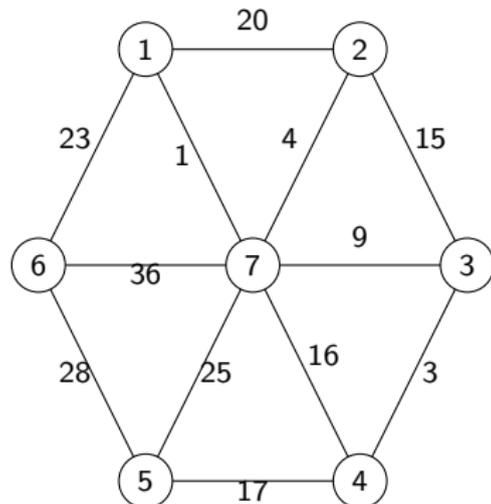


Figure: Graph  $G$

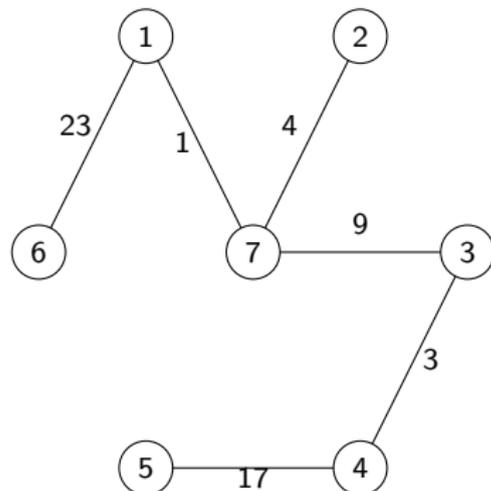


Figure: MST of  $G$

# Prim's Algorithm

$T$  maintained by algorithm will be a tree. Start with a node in  $T$ . In each iteration, pick edge with least attachment cost to  $T$ .

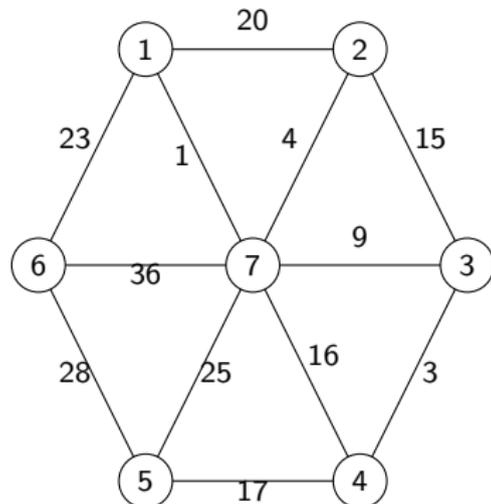


Figure: Graph  $G$

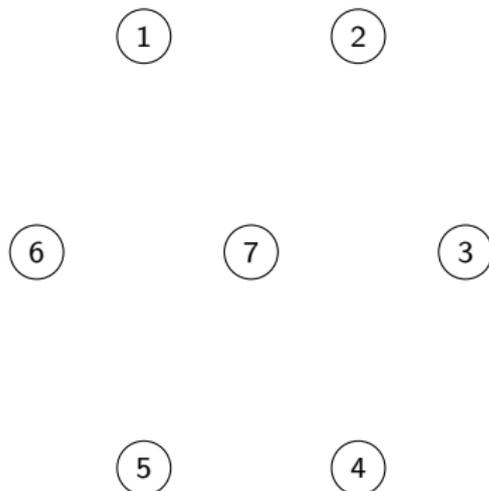


Figure: MST of  $G$

# Prim's Algorithm

$T$  maintained by algorithm will be a tree. Start with a node in  $T$ . In each iteration, pick edge with least attachment cost to  $T$ .

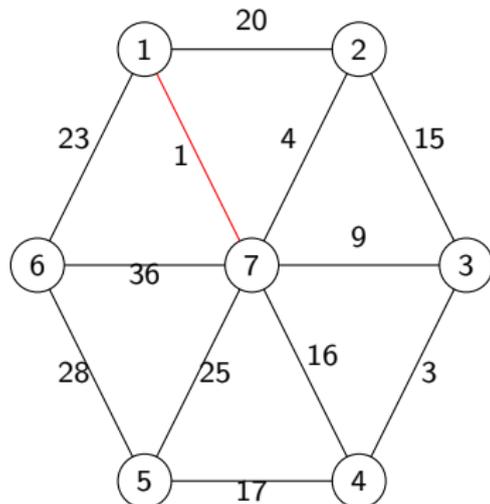


Figure: Graph  $G$

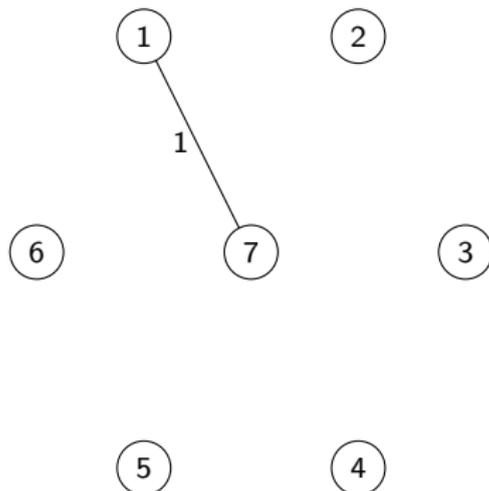


Figure: MST of  $G$

# Prim's Algorithm

$T$  maintained by algorithm will be a tree. Start with a node in  $T$ . In each iteration, pick edge with least attachment cost to  $T$ .

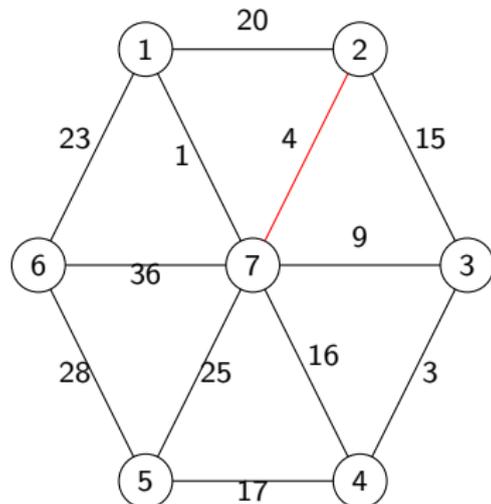


Figure: Graph  $G$

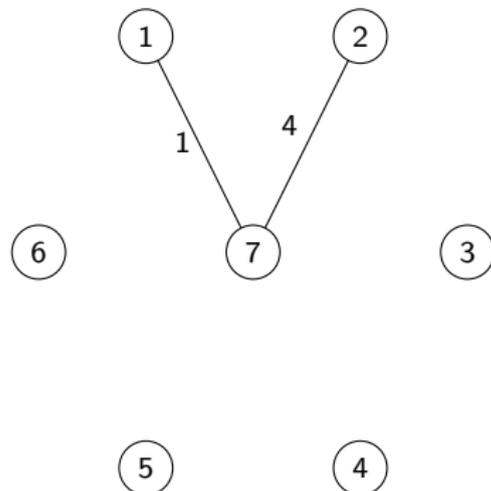


Figure: MST of  $G$

# Prim's Algorithm

$T$  maintained by algorithm will be a tree. Start with a node in  $T$ . In each iteration, pick edge with least attachment cost to  $T$ .

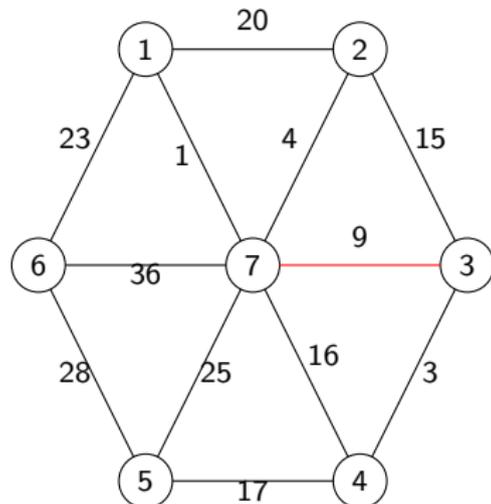


Figure: Graph  $G$

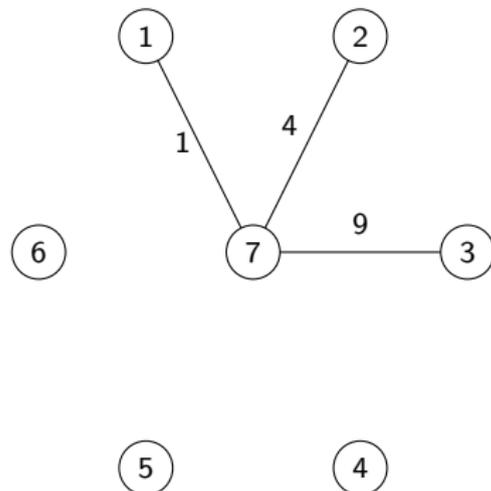


Figure: MST of  $G$

# Prim's Algorithm

$T$  maintained by algorithm will be a tree. Start with a node in  $T$ . In each iteration, pick edge with least attachment cost to  $T$ .

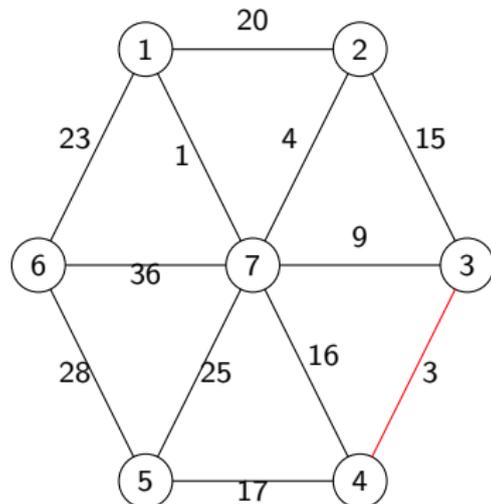


Figure: Graph  $G$

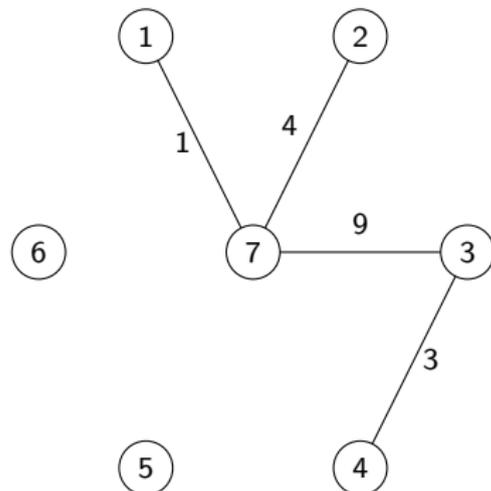


Figure: MST of  $G$

# Prim's Algorithm

$T$  maintained by algorithm will be a tree. Start with a node in  $T$ . In each iteration, pick edge with least attachment cost to  $T$ .

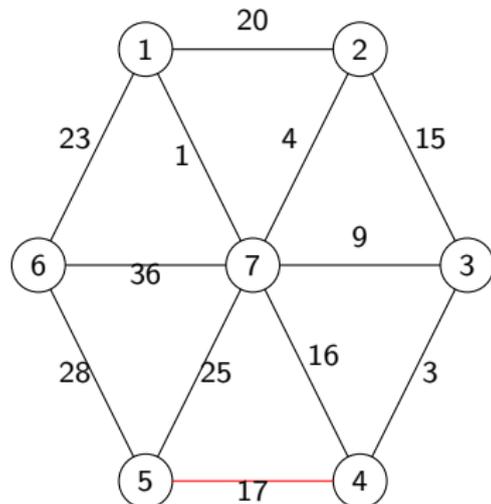


Figure: Graph  $G$

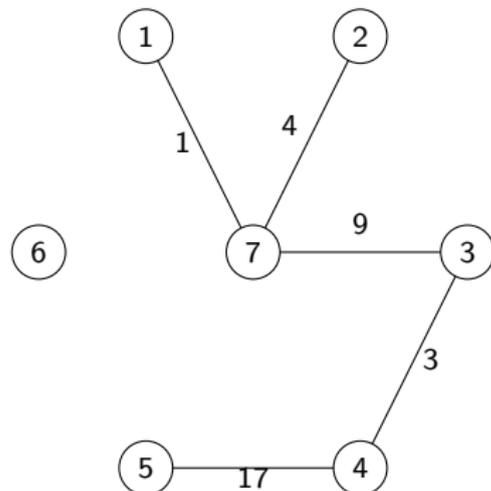


Figure: MST of  $G$

# Prim's Algorithm

$T$  maintained by algorithm will be a tree. Start with a node in  $T$ . In each iteration, pick edge with least attachment cost to  $T$ .

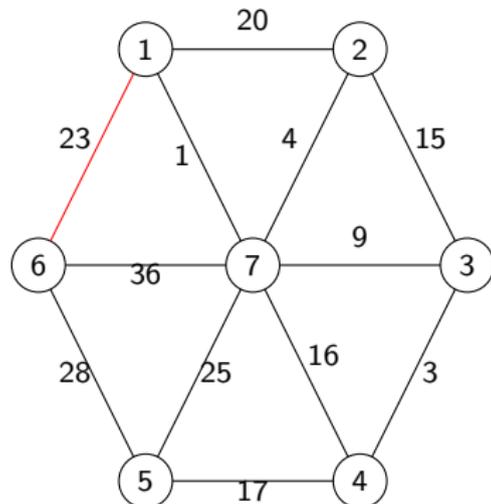


Figure: Graph  $G$

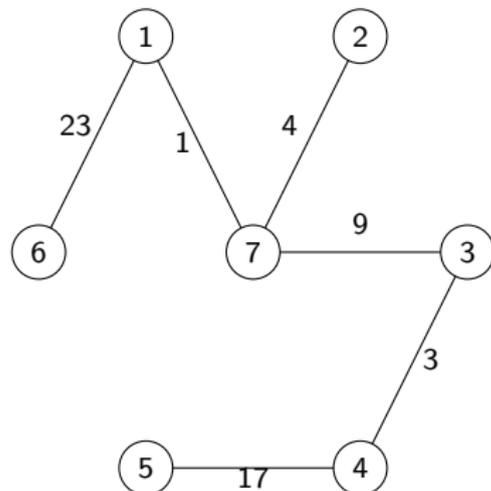


Figure: MST of  $G$

# Reverse Delete Algorithm

```
Initially E is the set of all edges in G
T is E (* T will store edges of a MST *)
while E is not empty
    choose  $i \in E$  of largest cost
    if removing  $i$  does not disconnect T
        remove  $i$  from T
return the set T
```

Returns a minimum spanning tree.

[Back](#)

# Correctness of MST Algorithms

- Many different MST algorithms
- All of them rely on some basic properties of MSTs, in particular the *Cut Property* to be seen shortly.

And for now ...

### Assumption

*Edge costs are distinct, that is no two edge costs are equal.*

# Safe and Unsafe Edges

## Definition

An edge  $e = (u, v)$  is a **safe** edge if there is some partition of  $V$  into  $S$  and  $V \setminus S$  and  $e$  is the unique minimum cost edge crossing  $S$  (one end in  $S$  and the other in  $V \setminus S$ ).

# Safe and Unsafe Edges

## Definition

An edge  $e = (u, v)$  is a **safe** edge if there is some partition of  $V$  into  $S$  and  $V \setminus S$  and  $e$  is the unique minimum cost edge crossing  $S$  (one end in  $S$  and the other in  $V \setminus S$ ).

## Definition

An edge  $e = (u, v)$  is an **unsafe** edge if there is some cycle  $C$  such that  $e$  is the unique maximum cost edge in  $C$ .

# Safe and Unsafe Edges

## Definition

An edge  $e = (u, v)$  is a **safe** edge if there is some partition of  $V$  into  $S$  and  $V \setminus S$  and  $e$  is the unique minimum cost edge crossing  $S$  (one end in  $S$  and the other in  $V \setminus S$ ).

## Definition

An edge  $e = (u, v)$  is an **unsafe** edge if there is some cycle  $C$  such that  $e$  is the unique maximum cost edge in  $C$ .

## Proposition

*If edge costs are distinct then every edge is either safe or unsafe.*

## Proof.

Exercise. □

## Example

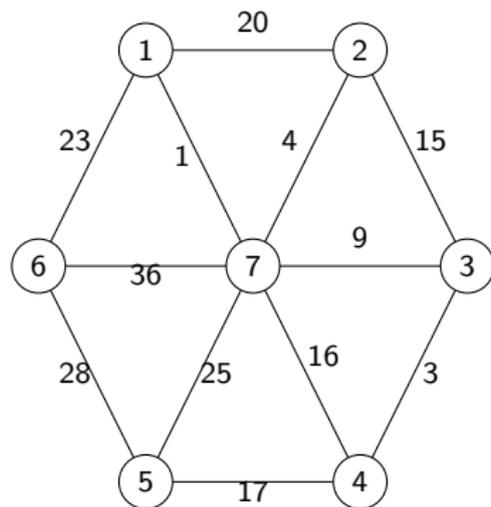


Figure: Graph with unique edge costs.

## Example

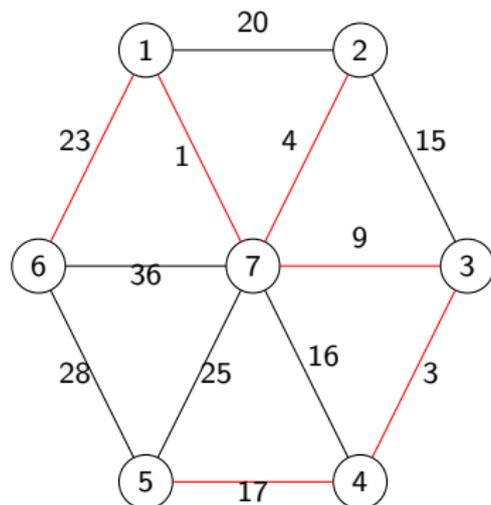


Figure: Graph with unique edge costs. Safe edges are red, rest are unsafe.

# Key Observation: Cut Property

## Lemma

*If  $e$  is a safe edge then every minimum spanning tree contains  $e$ .*

# Key Observation: Cut Property

## Lemma

*If  $e$  is a safe edge then every minimum spanning tree contains  $e$ .*

## Proof.



# Key Observation: Cut Property

## Lemma

*If  $e$  is a safe edge then every minimum spanning tree contains  $e$ .*

## Proof.

- Suppose (for contradiction)  $e$  is not in MST  $T$ .



# Key Observation: Cut Property

## Lemma

*If  $e$  is a safe edge then every minimum spanning tree contains  $e$ .*

## Proof.

- Suppose (for contradiction)  $e$  is not in MST  $T$ .
- Since  $e$  is safe there is an  $S \subset V$  such that  $e$  is the unique min cost edge crossing  $S$ .



# Key Observation: Cut Property

## Lemma

*If  $e$  is a safe edge then every minimum spanning tree contains  $e$ .*

## Proof.

- Suppose (for contradiction)  $e$  is not in MST  $T$ .
- Since  $e$  is safe there is an  $S \subset V$  such that  $e$  is the unique min cost edge crossing  $S$ .
- Since  $T$  is connected, there must be some edge  $f$  with one end in  $S$  and the other in  $V \setminus S$



# Key Observation: Cut Property

## Lemma

*If  $e$  is a safe edge then every minimum spanning tree contains  $e$ .*

## Proof.

- Suppose (for contradiction)  $e$  is not in MST  $T$ .
- Since  $e$  is safe there is an  $S \subset V$  such that  $e$  is the unique min cost edge crossing  $S$ .
- Since  $T$  is connected, there must be some edge  $f$  with one end in  $S$  and the other in  $V \setminus S$
- Since  $c_f > c_e$ ,  $T' = (T \setminus \{f\}) \cup \{e\}$  is a spanning tree of lower cost!



# Key Observation: Cut Property

## Lemma

*If  $e$  is a safe edge then every minimum spanning tree contains  $e$ .*

## Proof.

- Suppose (for contradiction)  $e$  is not in MST  $T$ .
- Since  $e$  is safe there is an  $S \subset V$  such that  $e$  is the unique min cost edge crossing  $S$ .
- Since  $T$  is connected, there must be some edge  $f$  with one end in  $S$  and the other in  $V \setminus S$
- Since  $c_f > c_e$ ,  $T' = (T \setminus \{f\}) \cup \{e\}$  is a spanning tree of lower cost! **Error:  $T'$  may not be a spanning tree!!**



## Error in Proof: Example

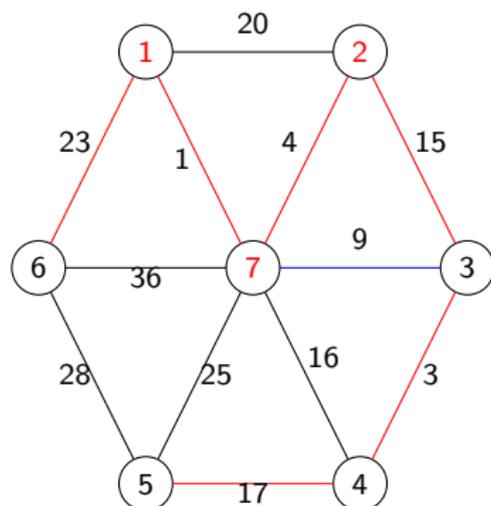
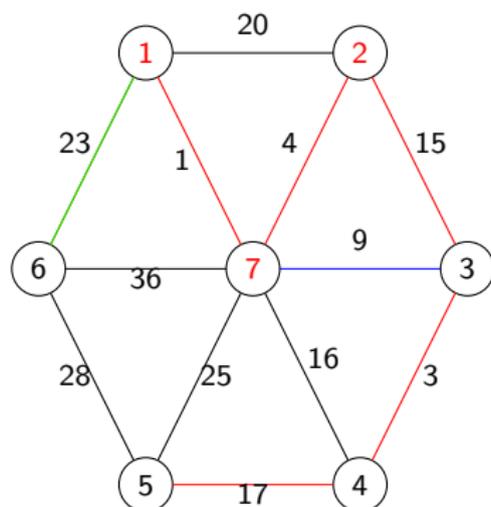


Figure: Problematic example.  $S = \{1, 2, 7\}$ ,  $e = (7, 3)$ ,  $f = (1, 6)$ .  
 $T - f + e$  is not a spanning tree.

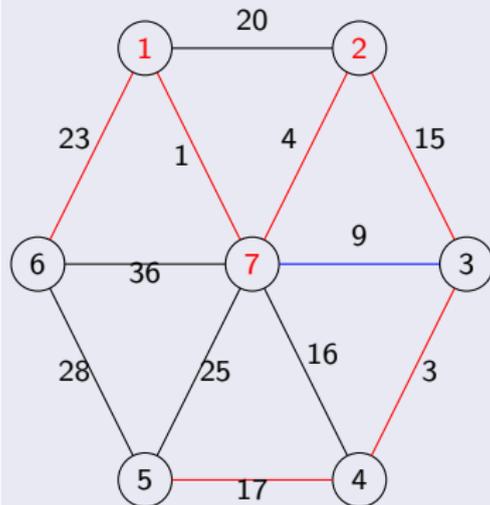
## Error in Proof: Example



**Figure:** Problematic example.  $S = \{1, 2, 7\}$ ,  $e = (7, 3)$ ,  $f = (1, 6)$ .  
 $T - f + e$  is not a spanning tree.

## Proof of Cut Property

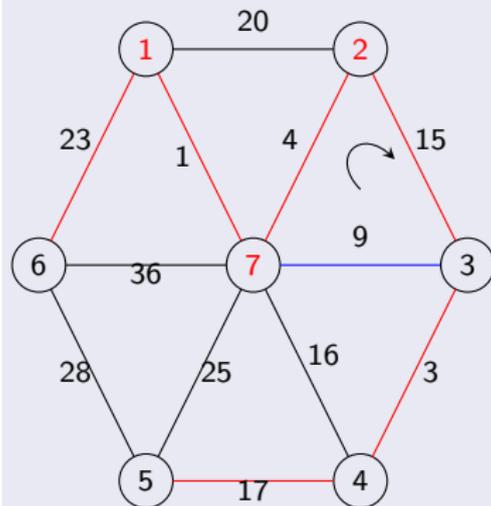
Proof.



- Suppose  $e = (v, w)$  is not in MST  $T$  and  $e$  is min weight edge in cut  $(S, V \setminus S)$ . Wlog  $v \in S$ .

## Proof of Cut Property

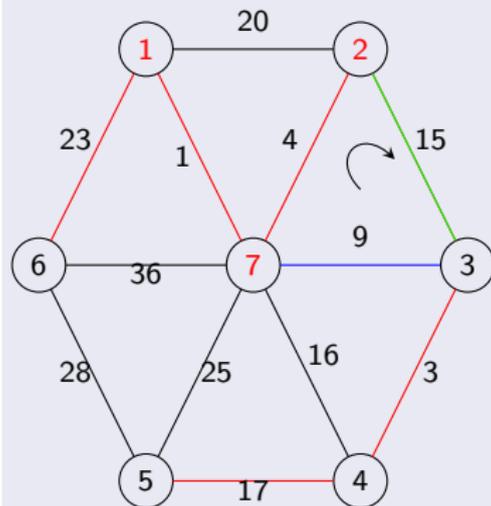
Proof.



- Suppose  $e = (v, w)$  is not in MST  $T$  and  $e$  is min weight edge in cut  $(S, V \setminus S)$ . Wlog  $v \in S$ .
- $T$  is spanning tree: there is a unique path  $P$  from  $v$  to  $w$  in  $T$

## Proof of Cut Property

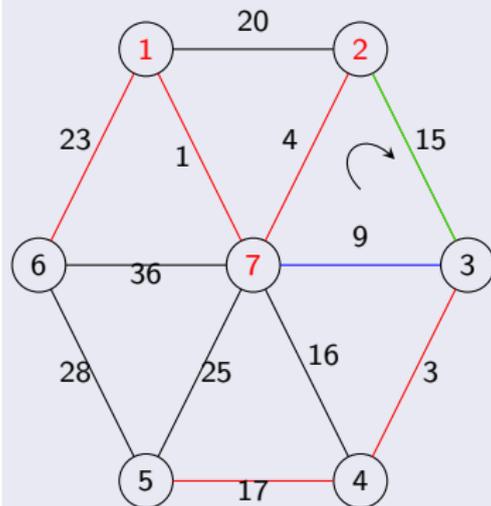
Proof.



- Suppose  $e = (v, w)$  is not in MST  $T$  and  $e$  is min weight edge in cut  $(S, V \setminus S)$ . Wlog  $v \in S$ .
- $T$  is spanning tree: there is a unique path  $P$  from  $v$  to  $w$  in  $T$
- Let  $w'$  be the first vertex in  $P$  belonging to  $V \setminus S$ ; let  $v'$  be the vertex just before it on  $P$ , and let  $e' = (v', w')$

## Proof of Cut Property

Proof.



- Suppose  $e = (v, w)$  is not in MST  $T$  and  $e$  is min weight edge in cut  $(S, V \setminus S)$ . Wlog  $v \in S$ .
- $T$  is spanning tree: there is a unique path  $P$  from  $v$  to  $w$  in  $T$
- Let  $w'$  be the first vertex in  $P$  belonging to  $V \setminus S$ ; let  $v'$  be the vertex just before it on  $P$ , and let  $e' = (v', w')$
- $T' = (T \setminus \{e'\}) \cup \{e\}$  is spanning tree of lower cost. (Why?)  $\square$

# Proof of Cut Property (contd)

## Observation

$T' = (T \setminus \{e'\}) \cup \{e\}$  is a spanning tree.

## Proof.

$T'$  is connected.

$T'$  is a tree



# Proof of Cut Property (contd)

## Observation

$T' = (T \setminus \{e'\}) \cup \{e\}$  is a spanning tree.

## Proof.

$T'$  is connected.

Removed  $e' = (v', w')$  from  $T$  but  $v'$  and  $w'$  are connected by the path  $P - f + e$  in  $T'$ . Hence  $T'$  is connected if  $T$  is.

$T'$  is a tree



# Proof of Cut Property (contd)

## Observation

$T' = (T \setminus \{e'\}) \cup \{e\}$  is a spanning tree.

## Proof.

$T'$  is connected.

Removed  $e' = (v', w')$  from  $T$  but  $v'$  and  $w'$  are connected by the path  $P - f + e$  in  $T'$ . Hence  $T'$  is connected if  $T$  is.

$T'$  is a tree

$T'$  is connected and has  $n - 1$  edges (since  $T$  had  $n - 1$  edges) and hence  $T'$  is a tree



# Safe Edges form a Tree

## Lemma

*Let  $G$  be a connected graph with distinct edge costs, then the set of safe edges form a connected graph.*

## Proof.

- Suppose not. Let  $S$  be a connected component in the graph induced by the safe edges.
- Consider the edges crossing  $S$ , there must be a safe edge among them since edge costs are distinct and so we must have picked it.



# Safe Edges form an MST

## Corollary

*Let  $G$  be a connected graph with distinct edge costs, then set of safe edges form the **unique** MST of  $G$ .*

# Safe Edges form an MST

## Corollary

*Let  $G$  be a connected graph with distinct edge costs, then set of safe edges form the **unique** MST of  $G$ .*

**Consequence:** Every correct MST algorithm when  $G$  has unique edge costs includes exactly the safe edges.

# Cycle Property

## Lemma

*If  $e$  is an unsafe edge then no MST of  $G$  contains  $e$ .*

## Proof.

Exercise. See text book.

**Note:** Cut and Cycle properties hold even when edge costs are not distinct. Safe and unsafe definitions do not rely on distinct cost assumption.

# Correctness of Prim's Algorithm

## Prim's Algorithm

Pick edge with minimum attachment cost to current tree, and add to current tree.

## Proof of correctness.

- If  $e$  is added to tree, then  $e$  is safe and belongs to every MST.
- Set of edges output is a spanning tree

# Correctness of Prim's Algorithm

## Prim's Algorithm

Pick edge with minimum attachment cost to current tree, and add to current tree.

## Proof of correctness.

- If  $e$  is added to tree, then  $e$  is safe and belongs to every MST.
  - Let  $S$  be the vertices connected by edges in  $T$  when  $e$  is added.
- Set of edges output is a spanning tree

# Correctness of Prim's Algorithm

## Prim's Algorithm

Pick edge with minimum attachment cost to current tree, and add to current tree.

## Proof of correctness.

- If  $e$  is added to tree, then  $e$  is safe and belongs to every MST.
  - Let  $S$  be the vertices connected by edges in  $T$  when  $e$  is added.
  - $e$  is edge of lowest cost with one end in  $S$  and the other in  $V \setminus S$  and hence  $e$  is safe.
- Set of edges output is a spanning tree

# Correctness of Prim's Algorithm

## Prim's Algorithm

Pick edge with minimum attachment cost to current tree, and add to current tree.

## Proof of correctness.

- If  $e$  is added to tree, then  $e$  is safe and belongs to every MST.
  - Let  $S$  be the vertices connected by edges in  $T$  when  $e$  is added.
  - $e$  is edge of lowest cost with one end in  $S$  and the other in  $V \setminus S$  and hence  $e$  is safe.
- Set of edges output is a spanning tree
  - Set of edges output forms a connected graph: by induction,  $S$  is connected in each iteration and eventually  $S = V$ .

# Correctness of Prim's Algorithm

## Prim's Algorithm

Pick edge with minimum attachment cost to current tree, and add to current tree.

## Proof of correctness.

- If  $e$  is added to tree, then  $e$  is safe and belongs to every MST.
  - Let  $S$  be the vertices connected by edges in  $T$  when  $e$  is added.
  - $e$  is edge of lowest cost with one end in  $S$  and the other in  $V \setminus S$  and hence  $e$  is safe.
- Set of edges output is a spanning tree
  - Set of edges output forms a connected graph: by induction,  $S$  is connected in each iteration and eventually  $S = V$ .
  - Only safe edges added and they do not have a cycle □



# Correctness of Kruskal's Algorithm

## Kruskal's Algorithm

Pick edge of lowest cost and add if it does not form a cycle with existing edges.

## Proof of correctness.

- If  $e = (u, v)$  is added to tree, then  $e$  is safe
  - When algorithm adds  $e$  let  $S$  and  $S'$  be the connected components containing  $u$  and  $v$  respectively
  
- Set of edges output is a spanning tree : exercise



# Correctness of Kruskal's Algorithm

## Kruskal's Algorithm

Pick edge of lowest cost and add if it does not form a cycle with existing edges.

## Proof of correctness.

- If  $e = (u, v)$  is added to tree, then  $e$  is safe
  - When algorithm adds  $e$  let  $S$  and  $S'$  be the connected components containing  $u$  and  $v$  respectively
  - $e$  is the lowest cost edge crossing  $S$  (and also  $S'$ ).
  
- Set of edges output is a spanning tree : exercise



# Correctness of Kruskal's Algorithm

## Kruskal's Algorithm

Pick edge of lowest cost and add if it does not form a cycle with existing edges.

## Proof of correctness.

- If  $e = (u, v)$  is added to tree, then  $e$  is safe
  - When algorithm adds  $e$  let  $S$  and  $S'$  be the connected components containing  $u$  and  $v$  respectively
  - $e$  is the lowest cost edge crossing  $S$  (and also  $S'$ ).
  - If there is an edge  $e'$  crossing  $S$  and has lower cost than  $e$ , then  $e'$  would come before  $e$  in the sorted order and would be added by the algorithm to  $T$
- Set of edges output is a spanning tree : exercise



# Correctness of Reverse Delete Algorithm

## Reverse Delete Algorithm

Consider edges in decreasing cost and remove an edge if it does not disconnect the graph

## Proof of correctness.

Argue that only unsafe edges are removed (see text book).

# When edge costs are not distinct

**Heuristic argument:** Make edge costs distinct by adding a small tiny and different cost to each edge

# When edge costs are not distinct

**Heuristic argument:** Make edge costs distinct by adding a small tiny and different cost to each edge

**Formal argument:** Order edges lexicographically to break ties

- $e_i \prec e_j$  if either  $c(e_i) < c(e_j)$  or  $(c(e_i) = c(e_j)$  and  $i < j)$

# When edge costs are not distinct

**Heuristic argument:** Make edge costs distinct by adding a small tiny and different cost to each edge

**Formal argument:** Order edges lexicographically to break ties

- $e_i \prec e_j$  if either  $c(e_i) < c(e_j)$  or  $(c(e_i) = c(e_j)$  and  $i < j)$
- Lexicographic ordering extends to sets of edges. If  $A, B \subseteq E$ ,  $A \neq B$  then  $A \prec B$  if either  $c(A) < c(B)$  or  $(c(A) = c(B)$  and  $A \setminus B$  has a lower indexed edge than  $B \setminus A)$

# When edge costs are not distinct

**Heuristic argument:** Make edge costs distinct by adding a small tiny and different cost to each edge

**Formal argument:** Order edges lexicographically to break ties

- $e_i \prec e_j$  if either  $c(e_i) < c(e_j)$  or  $(c(e_i) = c(e_j)$  and  $i < j)$
- Lexicographic ordering extends to sets of edges. If  $A, B \subseteq E$ ,  $A \neq B$  then  $A \prec B$  if either  $c(A) < c(B)$  or  $(c(A) = c(B)$  and  $A \setminus B$  has a lower indexed edge than  $B \setminus A)$
- Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

Prim's, Kruskal, and Reverse Delete Algorithms are optimal with respect to lexicographic ordering.

# Edge Costs: Positive and Negative

- Algorithms and proofs don't assume that edge costs are non-negative! MST algorithms work for arbitrary edge costs.

# Edge Costs: Positive and Negative

- Algorithms and proofs don't assume that edge costs are non-negative! MST algorithms work for arbitrary edge costs.
- Another way to see this: make edge costs non-negative by adding to each edge a large enough positive number. Why does this work for MSTs but not for shortest paths?

## Edge Costs: Positive and Negative

- Algorithms and proofs don't assume that edge costs are non-negative! MST algorithms work for arbitrary edge costs.
- Another way to see this: make edge costs non-negative by adding to each edge a large enough positive number. Why does this work for MSTs but not for shortest paths?
- Can compute *maximum* weight spanning tree by negating edge costs and then computing an MST.

## Edge Costs: Positive and Negative

- Algorithms and proofs don't assume that edge costs are non-negative! MST algorithms work for arbitrary edge costs.
- Another way to see this: make edge costs non-negative by adding to each edge a large enough positive number. Why does this work for MSTs but not for shortest paths?
- Can compute *maximum* weight spanning tree by negating edge costs and then computing an MST.

## Part II

# Data Structures for MST: Priority Queues and Union-Find

# Implementing Prim's Algorithm

```
• E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
while S != V
    pick e = (v,w) in E such that
        v ∈ S and w ∈ V - S
        e has minimum cost
    T = T U e
    S = S U w
return the set T
```

## Analysis

# Implementing Prim's Algorithm

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
while S != V
    pick e = (v,w) in E such that
        v ∈ S and w ∈ V - S
        e has minimum cost
    T = T U e
    S = S U w
return the set T
```

## Analysis

- Number of iterations =  $O(n)$ , where  $n$  is number of vertices

# Implementing Prim's Algorithm

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
while S  $\neq$  V
    pick e = (v,w) in E such that
        v  $\in$  S and w  $\in$  V - S
        e has minimum cost
    T = T U e
    S = S U w
return the set T
```

## Analysis

- Number of iterations =  $O(n)$ , where  $n$  is number of vertices
- Picking  $e$  is  $O(m)$  where  $m$  is the number of edges

# Implementing Prim's Algorithm

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
while S  $\neq$  V
    pick e = (v,w) in E such that
        v  $\in$  S and w  $\in$  V - S
        e has minimum cost
    T = T U e
    S = S U w
return the set T
```

## Analysis

- Number of iterations =  $O(n)$ , where  $n$  is number of vertices
- Picking  $e$  is  $O(m)$  where  $m$  is the number of edges
- Total time  $O(nm)$

# More Efficient Implementation

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v  $\notin$  S, a(v) =  $\min_{w \in S} c(w,v)$ 
for v  $\notin$  S, e(v) = w such that w  $\in$  S and c(w,v) is minimum
while S  $\neq$  V
    pick v with minimum a(v)
    T = T U (e(v),v)
    S = S U v
    update arrays a and e
return the set T
```

# More Efficient Implementation

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v  $\notin$  S, a(v) =  $\min_{w \in S} c(w,v)$ 
for v  $\notin$  S, e(v) = w such that w  $\in$  S and c(w,v) is minimum
while S  $\neq$  V
    pick v with minimum a(v)
    T = T U (e(v),v)
    S = S U v
    update arrays a and e
return the set T
```

# More Efficient Implementation

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v  $\notin$  S, a(v) =  $\min_{w \in S} c(w,v)$ 
for v  $\notin$  S, e(v) = w such that w  $\in$  S and c(w,v) is minimum
while S  $\neq$  V
    pick v with minimum a(v)
    T = T U (e(v),v)
    S = S U v
    update arrays a and e
return the set T
```

Maintain vertices in  $V \setminus S$  in a priority queue with key  $a(v)$

# Priority Queues

Data structure to store a set  $S$  of  $n$  elements where each element  $v \in S$  has an associated real/integer key  $k(v)$  such that the following operations

- `makeQ`: create an empty queue
- `findMin`: find the minimum key in  $S$
- `extractMin`: Remove  $v \in S$  with smallest key and return it
- `add(v, k(v))`: Add new element  $v$  with key  $k(v)$  to  $S$
- `delete(v)`: Remove element  $v$  from  $S$
- `decreaseKey(v, k'(v))`: *decrease* key of  $v$  from  $k(v)$  (current key) to  $k'(v)$  (new key). Assumption:  $k'(v) \leq k(v)$
- `meld`: merge two separate priority queues into one

# Prim's using priority queues

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v  $\notin$  S, a(v) =  $\min_{w \in S} c(w,v)$ 
for v  $\notin$  S, e(v) = w such that w  $\in$  S and c(w,v) is minimum
while S  $\neq$  V
    pick v with minimum a(v)
    T = T U (e(v),v)
    S = S U v
    update arrays a and e
return the set T
```

Maintain vertices in  $V \setminus S$  in a priority queue with key  $a(v)$

# Prim's using priority queues

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v  $\notin$  S, a(v) =  $\min_{w \in S} c(w,v)$ 
for v  $\notin$  S, e(v) = w such that w  $\in$  S and c(w,v) is minimum
while S  $\neq$  V
    pick v with minimum a(v)
    T = T U (e(v),v)
    S = S U v
    update arrays a and e
return the set T
```

Maintain vertices in  $V \setminus S$  in a priority queue with key  $a(v)$

- Requires  $O(n)$  extractMin operations

# Prim's using priority queues

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v  $\notin$  S, a(v) =  $\min_{w \in S} c(w,v)$ 
for v  $\notin$  S, e(v) = w such that w  $\in$  S and c(w,v) is minimum
while S  $\neq$  V
    pick v with minimum a(v)
    T = T U (e(v),v)
    S = S U v
    update arrays a and e
return the set T
```

Maintain vertices in  $V \setminus S$  in a priority queue with key  $a(v)$

- Requires  $O(n)$  extractMin operations
- Requires  $O(m)$  decreaseKey operations

# Running time of Prim's Algorithm

$O(n)$  extractMin operations and  $O(m)$  decreaseKey operations

- Using standard Heaps, extractMin and decreaseKey take  $O(\log n)$  time. Total:  $O((m + n) \log n)$
- Using Fibonacci Heaps,  $O(\log n)$  for extractMin and  $O(1)$  (amortized) for decreaseKey. Total:  $O(n \log n + m)$ .

# Running time of Prim's Algorithm

$O(n)$  extractMin operations and  $O(m)$  decreaseKey operations

- Using standard Heaps, extractMin and decreaseKey take  $O(\log n)$  time. Total:  $O((m + n) \log n)$
- Using Fibonacci Heaps,  $O(\log n)$  for extractMin and  $O(1)$  (amortized) for decreaseKey. Total:  $O(n \log n + m)$ .

Prim's algorithm and Dijkstra's algorithms are similar. Where is the difference?

# Kruskal's Algorithm

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty
    choose e ∈ E of minimum cost
    if (T ∪ {e} does not have cycles)
        add e to T
return the set T
```

# Kruskal's Algorithm

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty
    choose e ∈ E of minimum cost
    if (T ∪ {e} does not have cycles)
        add e to T
return the set T
```

# Kruskal's Algorithm

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty
    choose e ∈ E of minimum cost
    if (T ∪ {e} does not have cycles)
        add e to T
return the set T
```

- Pre-sort edges based on cost. Choosing minimum can be done in  $O(1)$  time

# Kruskal's Algorithm

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty
    choose e ∈ E of minimum cost
    if (T ∪ {e} does not have cycles)
        add e to T
return the set T
```

- Pre-sort edges based on cost. Choosing minimum can be done in  $O(1)$  time

# Kruskal's Algorithm

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty
    choose e ∈ E of minimum cost
    if (T ∪ {e} does not have cycles)
        add e to T
return the set T
```

- Pre-sort edges based on cost. Choosing minimum can be done in  $O(1)$  time
- Do BFS/DFS on  $T \cup \{e\}$ . Takes  $O(n)$  time

# Kruskal's Algorithm

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty
    choose e ∈ E of minimum cost
    if (T ∪ {e} does not have cycles)
        add e to T
return the set T
```

- Pre-sort edges based on cost. Choosing minimum can be done in  $O(1)$  time
- Do BFS/DFS on  $T \cup \{e\}$ . Takes  $O(n)$  time
- Total time  $O(m \log m) + O(mn) = O(mn)$

# Implementing Kruskal's Algorithm Efficiently

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

# Implementing Kruskal's Algorithm Efficiently

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

# Implementing Kruskal's Algorithm Efficiently

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty
    pick e = (u,v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

Need a data structure to check if two elements belong to same set and to merge two sets.

# Union-Find Data Structure

## Data Structure

Store disjoint sets of elements that supports the following operations

- `makeUnionFind( $S$ )` returns a data structure where each element of  $S$  is in a separate set

# Union-Find Data Structure

## Data Structure

Store disjoint sets of elements that supports the following operations

- `makeUnionFind( $S$ )` returns a data structure where each element of  $S$  is in a separate set
- `find( $u$ )` returns the name of set containing element  $u$ . Thus,  $u$  and  $v$  belong to the same set iff `find( $u$ ) = find( $v$ )`

# Union-Find Data Structure

## Data Structure

Store disjoint sets of elements that supports the following operations

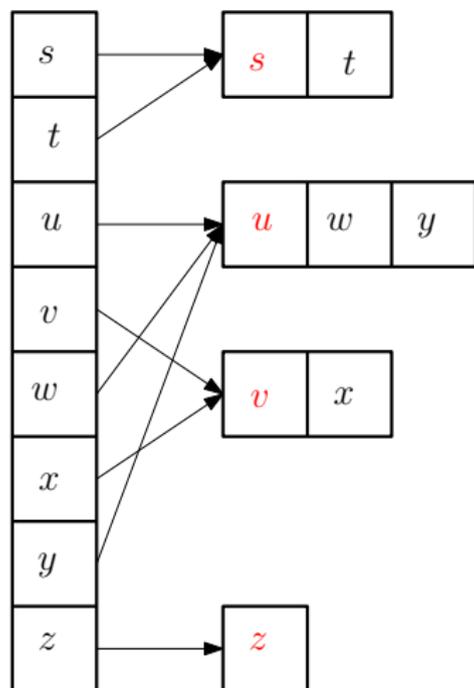
- `makeUnionFind( $S$ )` returns a data structure where each element of  $S$  is in a separate set
- `find( $u$ )` returns the name of set containing element  $u$ . Thus,  $u$  and  $v$  belong to the same set iff `find( $u$ ) = find( $v$ )`
- `union( $A, B$ )` merges two sets  $A$  and  $B$ .  
Typically: `union(find( $u$ ), find( $v$ ))`

# Implementing Union-Find using Arrays and Lists

## Using lists

- Each set stored as list with a name associated with the list.
- For each element  $u \in S$  a pointer to the its set. Array for pointers: `component[u]` is pointer for  $u$ .
- `makeUnionFind(S)` takes  $O(n)$  time and space.

# Example



# Implementing Union-Find using Arrays and Lists

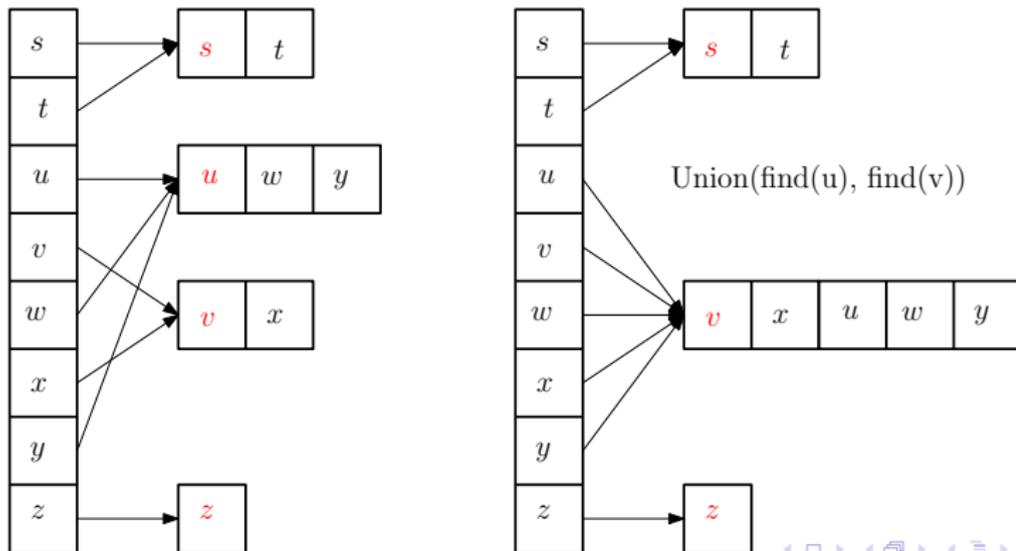
- `find(u)` reads the entry component `[u]`:  $O(1)$  time

# Implementing Union-Find using Arrays and Lists

- `find(u)` reads the entry component `[u]`:  $O(1)$  time
- `union(A,B)`

# Implementing Union-Find using Arrays and Lists

- $\text{find}(u)$  reads the entry component  $[u]$ :  $O(1)$  time
- $\text{union}(A, B)$  involves updating the entries component  $[u]$  for all elements  $u$  in  $A$  and  $B$ :  $O(|A| + |B|)$  which is  $O(n)$



# Improving the List Implementation for Union

## New Implementation

As before use `component[u]` to store set of  $u$ .

Change to `union(A,B)`:

- with each set, keep track of its size
- assume  $|A| \leq |B|$  for now
- Merge the list of  $A$  into that of  $B$ :  $O(1)$  time (linked lists)
- Update `component[u]` only for elements in the smaller set  $A$
- Total  $O(|A|)$  time.

# Improving the List Implementation for Union

## New Implementation

As before use `component[u]` to store set of  $u$ .

Change to `union(A,B)`:

- with each set, keep track of its size
- assume  $|A| \leq |B|$  for now
- Merge the list of  $A$  into that of  $B$ :  $O(1)$  time (linked lists)
- Update `component[u]` only for elements in the smaller set  $A$
- Total  $O(|A|)$  time. Worst case is still  $O(n)$ .

# Improving the List Implementation for Union

## New Implementation

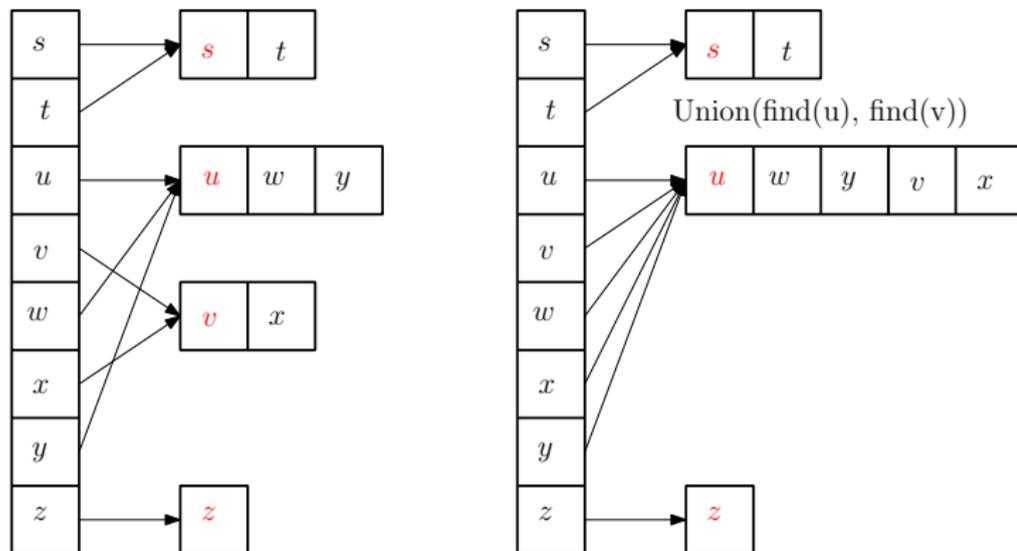
As before use `component[u]` to store set of  $u$ .

Change to `union(A,B)`:

- with each set, keep track of its size
- assume  $|A| \leq |B|$  for now
- Merge the list of  $A$  into that of  $B$ :  $O(1)$  time (linked lists)
- Update `component[u]` only for elements in the smaller set  $A$
- Total  $O(|A|)$  time. Worst case is still  $O(n)$ .

`find` still takes  $O(1)$  time

## Example



The smaller set (list) is appended to the largest set (list)

# Improving the List Implementation for Union

## Question

Is the improved implementation provably better or is it simply a nice heuristic?

# Improving the List Implementation for Union

## Question

Is the improved implementation provably better or is it simply a nice heuristic?

## Theorem

*Any sequence of  $k$  union operations, starting from `makeUnionFind( $S$ )` on set  $S$  of size  $n$ , takes at most  $O(k \log k)$ .*

# Improving the List Implementation for Union

## Question

Is the improved implementation provably better or is it simply a nice heuristic?

## Theorem

*Any sequence of  $k$  union operations, starting from `makeUnionFind(S)` on set  $S$  of size  $n$ , takes at most  $O(k \log k)$ .*

## Corollary

*Kruskal's algorithm can be implemented in  $O(m \log m)$  time.*

Sorting takes  $O(m \log m)$  time,  $O(m)$  finds take  $O(m)$  time and  $O(n)$  unions take  $O(n \log n)$  time.

# Amortized Analysis

Why does theorem work?

## Key Observation

`union(A,B)` takes  $O(|A|)$  time where  $|A| \leq |B|$ . Size of new set is  $\geq 2|A|$ . Cannot double too many times.

# Proof of Theorem

Proof.

# Proof of Theorem

## Proof.

- Any union operation involves at most 2 of the original one-element sets; thus at least  $n - 2k$  elements have never been involved in a union

# Proof of Theorem

## Proof.

- Any union operation involves at most 2 of the original one-element sets; thus at least  $n - 2k$  elements have never been involved in a union
- Also, maximum size of any set (after  $k$  unions) is  $2k$

# Proof of Theorem

## Proof.

- Any union operation involves at most 2 of the original one-element sets; thus at least  $n - 2k$  elements have never been involved in a union
- Also, maximum size of any set (after  $k$  unions) is  $2k$
- `union(A,B)` takes  $O(|A|)$  time where  $|A| \leq |B|$ .

# Proof of Theorem

## Proof.

- Any union operation involves at most 2 of the original one-element sets; thus at least  $n - 2k$  elements have never been involved in a union
- Also, maximum size of any set (after  $k$  unions) is  $2k$
- `union(A,B)` takes  $O(|A|)$  time where  $|A| \leq |B|$ .
- *Charge* each element in  $A$  constant time to *pay* for  $O(|A|)$  time.

# Proof of Theorem

## Proof.

- Any union operation involves at most 2 of the original one-element sets; thus at least  $n - 2k$  elements have never been involved in a union
- Also, maximum size of any set (after  $k$  unions) is  $2k$
- `union(A,B)` takes  $O(|A|)$  time where  $|A| \leq |B|$ .
- *Charge* each element in  $A$  constant time to *pay* for  $O(|A|)$  time.
- How much does any element get charged?

# Proof of Theorem

## Proof.

- Any union operation involves at most 2 of the original one-element sets; thus at least  $n - 2k$  elements have never been involved in a union
- Also, maximum size of any set (after  $k$  unions) is  $2k$
- `union(A,B)` takes  $O(|A|)$  time where  $|A| \leq |B|$ .
- *Charge* each element in  $A$  constant time to *pay* for  $O(|A|)$  time.
- How much does any element get charged?
- If `component[v]` is updated, set containing  $v$  *doubles* in size

# Proof of Theorem

## Proof.

- Any union operation involves at most 2 of the original one-element sets; thus at least  $n - 2k$  elements have never been involved in a union
- Also, maximum size of any set (after  $k$  unions) is  $2k$
- `union(A,B)` takes  $O(|A|)$  time where  $|A| \leq |B|$ .
- *Charge* each element in  $A$  constant time to *pay* for  $O(|A|)$  time.
- How much does any element get charged?
- If `component[v]` is updated, set containing  $v$  *doubles* in size
- `component[v]` is updated at most  $\log 2k$  times

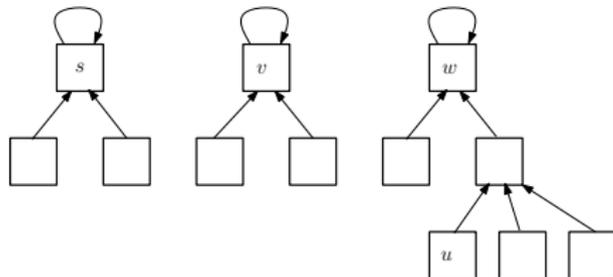
# Proof of Theorem

## Proof.

- Any union operation involves at most 2 of the original one-element sets; thus at least  $n - 2k$  elements have never been involved in a union
- Also, maximum size of any set (after  $k$  unions) is  $2k$
- `union(A,B)` takes  $O(|A|)$  time where  $|A| \leq |B|$ .
- *Charge* each element in  $A$  constant time to *pay* for  $O(|A|)$  time.
- How much does any element get charged?
- If `component[v]` is updated, set containing  $v$  *doubles* in size
- `component[v]` is updated at most  $\log 2k$  times
- Total number of updates is  $2k \log 2k = O(k \log k)$



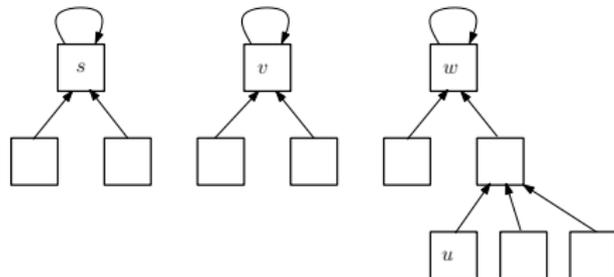
# Improving Worst Case Time



## Better Data Structure

Maintain elements in a forest of *in-trees*; all elements in one tree belong to a set with root's name.

# Improving Worst Case Time

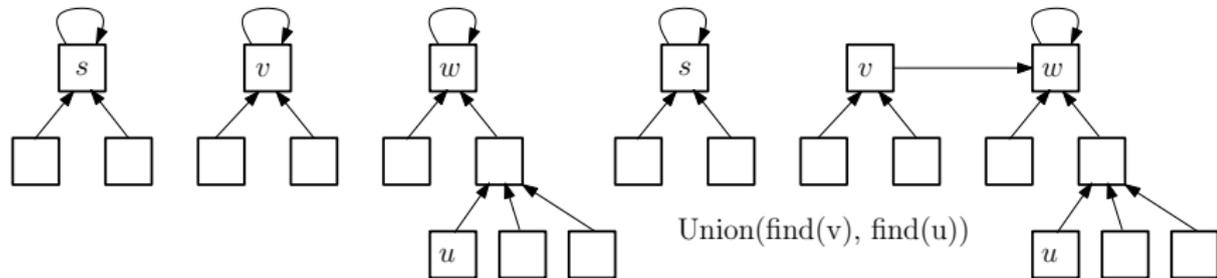


## Better Data Structure

Maintain elements in a forest of *in-trees*; all elements in one tree belong to a set with root's name.

- $\text{find}(u)$ : Traverse from  $u$  to the root

## Improving Worst Case Time



## Better Data Structure

Maintain elements in a forest of *in-trees*; all elements in one tree belong to a set with root's name.

- $\text{find}(u)$ : Traverse from  $u$  to the root
- $\text{union}(A, B)$ : Make root of  $A$  (smaller set) point to root of  $B$ . Takes  $O(1)$  time.

## Details of Implementation

Each element  $u \in S$  has a pointer  $\text{parent}(u)$  to its ancestor.

## Details of Implementation

Each element  $u \in S$  has a pointer  $\text{parent}(u)$  to its ancestor.

```
makeUnionFind(S)
  for each u in S
    parent(u) = u
```

## Details of Implementation

Each element  $u \in S$  has a pointer  $\text{parent}(u)$  to its ancestor.

```
makeUnionFind(S)
  for each u in S
    parent(u) = u

find(u)
  while (parent(u)  $\neq$  u)
    u = parent(u)
  return u
```

## Details of Implementation

Each element  $u \in S$  has a pointer  $\text{parent}(u)$  to its ancestor.

```
makeUnionFind(S)
```

```
    for each u in S  
        parent(u) = u
```

```
find(u)
```

```
    while (parent(u)  $\neq$  u)  
        u = parent(u)  
    return u
```

```
union(component(u), component(v)) (* parent(u) = u & parent(v) = v *)
```

```
    if (|component(u)|  $\leq$  |component(v)|)  
        parent(u) = v
```

```
    else
```

```
        parent(v) = u
```

```
    update new component size to be |component(u)| + |component(v)|
```

# Analysis

## Theorem

*The forest based implementation for a set of size  $n$ , has the following complexity for the various operations: `makeUnionFind` takes  $O(n)$ , `union` takes  $O(1)$ , and `find` takes  $O(\log n)$ .*

# Analysis

## Theorem

*The forest based implementation for a set of size  $n$ , has the following complexity for the various operations: `makeUnionFind` takes  $O(n)$ , `union` takes  $O(1)$ , and `find` takes  $O(\log n)$ .*

## Proof.

# Analysis

## Theorem

*The forest based implementation for a set of size  $n$ , has the following complexity for the various operations: `makeUnionFind` takes  $O(n)$ , `union` takes  $O(1)$ , and `find` takes  $O(\log n)$ .*

## Proof.

- `find( $u$ )` depends on the height of tree containing  $u$

# Analysis

## Theorem

*The forest based implementation for a set of size  $n$ , has the following complexity for the various operations: `makeUnionFind` takes  $O(n)$ , `union` takes  $O(1)$ , and `find` takes  $O(\log n)$ .*

## Proof.

- `find( $u$ )` depends on the height of tree containing  $u$
- Height of  $u$  increases by at most 1 only when the set containing  $u$  changes its name

# Analysis

## Theorem

*The forest based implementation for a set of size  $n$ , has the following complexity for the various operations: `makeUnionFind` takes  $O(n)$ , `union` takes  $O(1)$ , and `find` takes  $O(\log n)$ .*

## Proof.

- `find( $u$ )` depends on the height of tree containing  $u$
- Height of  $u$  increases by at most 1 only when the set containing  $u$  changes its name
- If height of  $u$  increases then size of the set containing  $u$  (at least) doubles

# Analysis

## Theorem

*The forest based implementation for a set of size  $n$ , has the following complexity for the various operations: `makeUnionFind` takes  $O(n)$ , `union` takes  $O(1)$ , and `find` takes  $O(\log n)$ .*

## Proof.

- `find( $u$ )` depends on the height of tree containing  $u$
- Height of  $u$  increases by at most 1 only when the set containing  $u$  changes its name
- If height of  $u$  increases then size of the set containing  $u$  (at least) doubles
- Maximum set size is  $n$ ; so height of any tree is at most  $O(\log n)$



# Further Improvements: Path Compression

## Observation

*Consecutive calls of  $\text{find}(u)$  take  $O(\log n)$  time each, but they traverse the same sequence of pointers.*

# Further Improvements: Path Compression

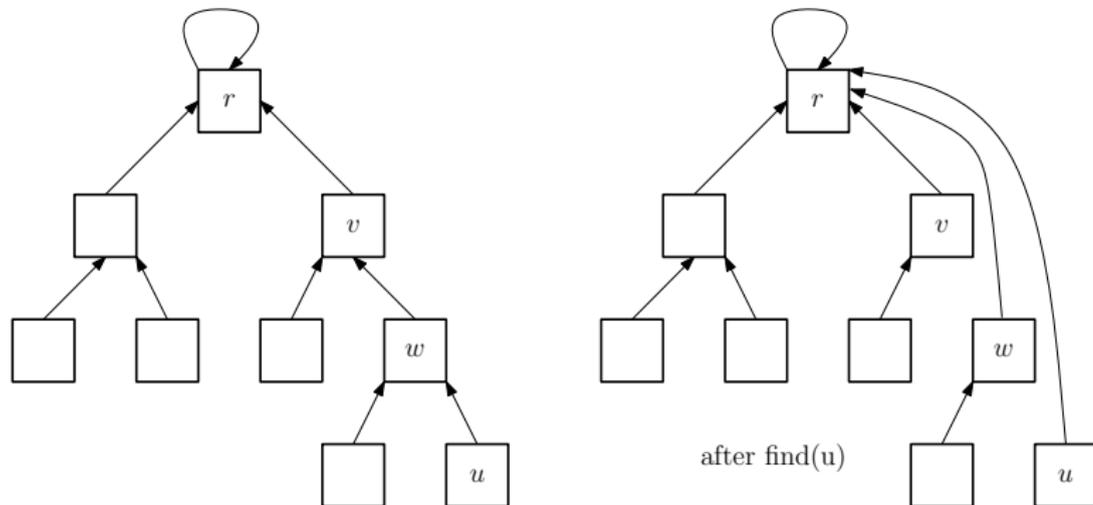
## Observation

*Consecutive calls of  $\text{find}(u)$  take  $O(\log n)$  time each, but they traverse the same sequence of pointers.*

## Idea: Path Compression

Make all nodes encountered in the  $\text{find}(u)$  point to root.

# Path Compression: Example



# Path Compression

```
find(u):  
    if (parent(u)  $\neq$  u)  
        parent(u) = find(parent(u))  
    return parent(u)
```

# Path Compression

```
find(u):  
    if (parent(u)  $\neq$  u)  
        parent(u) = find(parent(u))  
    return parent(u)
```

## Question

Does Path Compression help?

# Path Compression

```
find(u):  
    if (parent(u)  $\neq$  u)  
        parent(u) = find(parent(u))  
    return parent(u)
```

## Question

Does Path Compression help?

Yes!

## Theorem

*With Path Compression,  $k$  operations (find and/or union) take  $O(k\alpha(k, \min\{k, n\}))$  time where  $\alpha$  is the inverse Ackermann function.*

# Ackerman and Inverse Ackerman Functions

Ackerman function  $A(m, n)$  defined for  $m, n \geq 0$  recursively

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

# Ackerman and Inverse Ackerman Functions

Ackerman function  $A(m, n)$  defined for  $m, n \geq 0$  recursively

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

$$A(3, n) = 2^{n+3} - 3$$

$$A(4, 3) = 2^{65536} - 3$$

$\alpha(m, n)$  is inverse Ackerman function defined as

$$\alpha(m, n) = \min\{i \mid A(i, \lfloor m/n \rfloor) \geq \log_2 n\}$$

# Ackerman and Inverse Ackerman Functions

Ackerman function  $A(m, n)$  defined for  $m, n \geq 0$  recursively

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

$$A(3, n) = 2^{n+3} - 3$$

$$A(4, 3) = 2^{65536} - 3$$

$\alpha(m, n)$  is inverse Ackerman function defined as

$$\alpha(m, n) = \min\{i \mid A(i, \lfloor m/n \rfloor) \geq \log_2 n\}$$

For **all practical** purposes  $\alpha(m, n) \leq 5$

# Lower Bound for Union-Find Data Structure

Amazing result:

## Theorem (Tarjan)

*For UnionFind, **any** data structure in the pointer model requires  $O(m\alpha(m, n))$  time for  $m$  operations.*

# Running time of Kruskal's Algorithm

Using Union-Find data structure:

- $O(m)$  find operations (two for each edge)
- $O(n)$  union operations (one for each edge added to  $T$ )
- Total time:  $O(m \log m)$  for sorting plus  $O(m\alpha(n))$  for union-find operations. Thus  $O(m \log m)$  time despite the improved Union-Find data structure.

# Best Known Asymptotic Running Times for MST

Prim's algorithm using Fibonacci heaps:  $O(n \log n + m)$ .  
If  $m$  is  $O(n)$  then running time is  $\Omega(n \log n)$ .

# Best Known Asymptotic Running Times for MST

Prim's algorithm using Fibonacci heaps:  $O(n \log n + m)$ .  
If  $m$  is  $O(n)$  then running time is  $\Omega(n \log n)$ .

## Question

Is there a linear time ( $O(m + n)$  time) algorithm for MST?

# Best Known Asymptotic Running Times for MST

Prim's algorithm using Fibonacci heaps:  $O(n \log n + m)$ .  
If  $m$  is  $O(n)$  then running time is  $\Omega(n \log n)$ .

## Question

Is there a linear time ( $O(m + n)$  time) algorithm for MST?

- $O(m \log^* m)$  time [Fredman and Tarjan '1986]
- $O(m)$  time using bit operations in RAM model [Fredman and Willard 1993]
- $O(m)$  expected time (randomized algorithm) [Karger, Klein and Tarjan '1985]
- $O(m\alpha(m, n))$  time [Chazelle '97]
- Still open: is there an  $O(m)$  time deterministic algorithm in the comparison model?