

Greedy Algorithms for Minimum Spanning Trees

Lecture 18
March 31, 2015

Part I

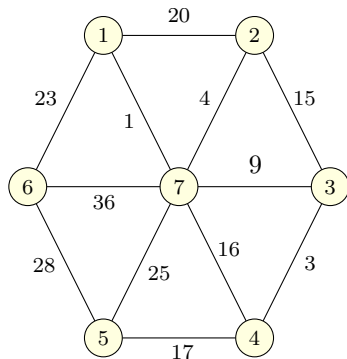
Greedy Algorithms: Minimum Spanning Tree

Minimum Spanning Tree

Input Connected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ with edge costs

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

① \mathbf{T} is the **minimum spanning tree (MST)** of \mathbf{G}

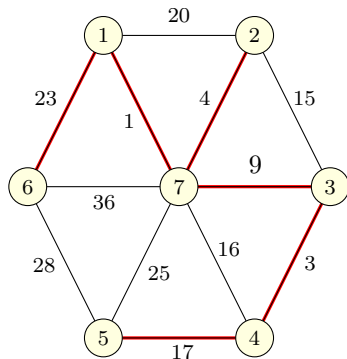


Minimum Spanning Tree

Input Connected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ with edge costs

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

① \mathbf{T} is the **minimum spanning tree** (MST) of \mathbf{G}



Applications

- 1 Network Design
 - 1 Designing networks with minimum cost but maximum connectivity
- 2 Approximation algorithms
 - 1 Can be used to bound the optimality of algorithms to approximate Traveling Salesman Problem, Steiner Trees, etc.
- 3 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 do  
    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.

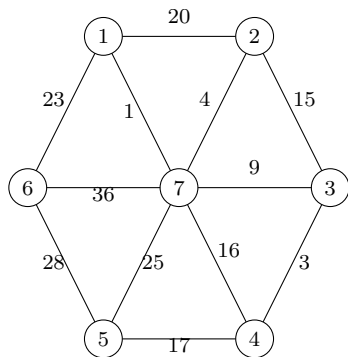


Figure : Graph **G**

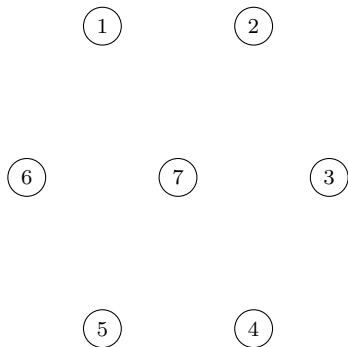


Figure : **MST** of **G**

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.

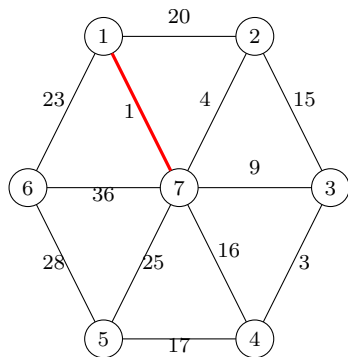


Figure : Graph **G**

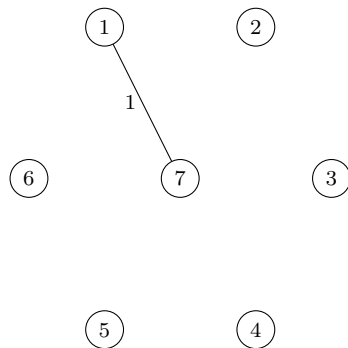


Figure : **MST** of **G**

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.

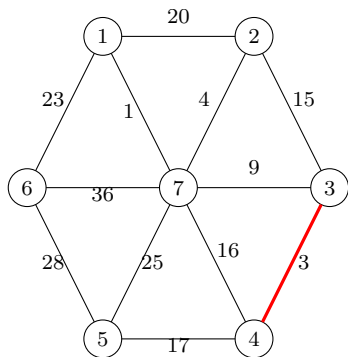


Figure : Graph **G**

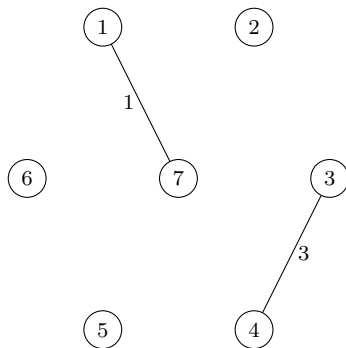


Figure : **MST** of **G**

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.

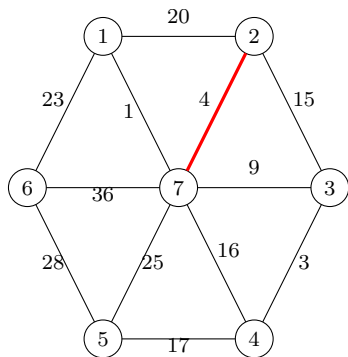


Figure : Graph **G**

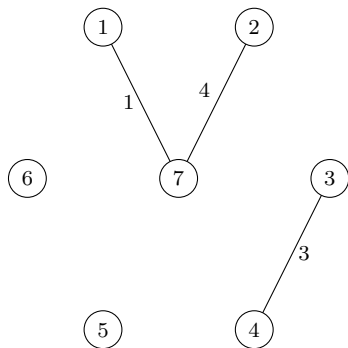


Figure : **MST** of **G**

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.

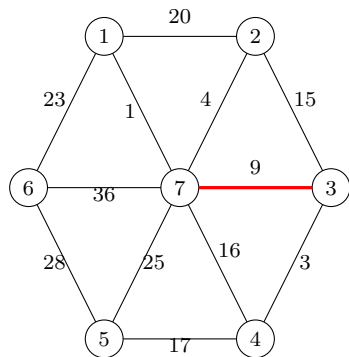


Figure : Graph **G**

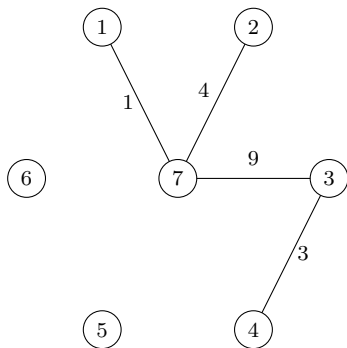


Figure : **MST** of **G**

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.

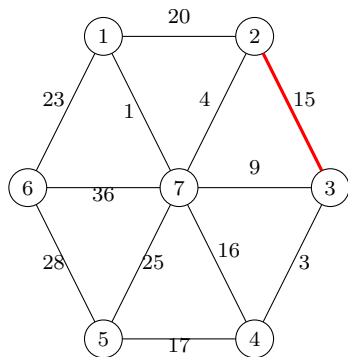


Figure : Graph **G**

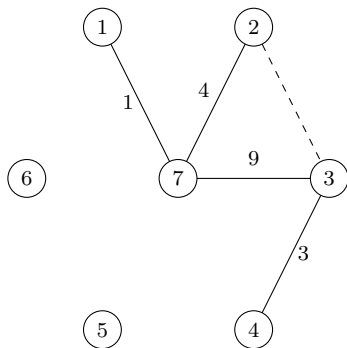


Figure : **MST** of **G**

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.

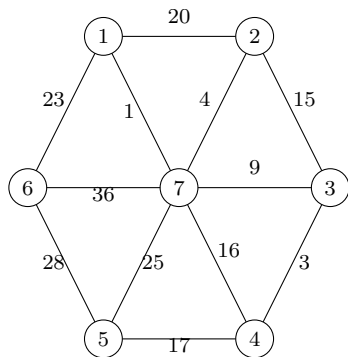


Figure : Graph **G**

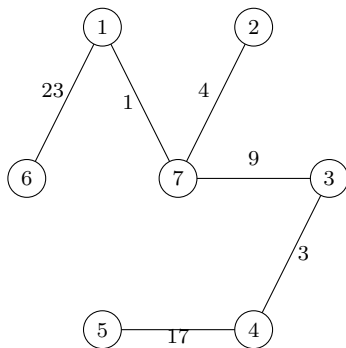


Figure : **MST** of **G**

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**.

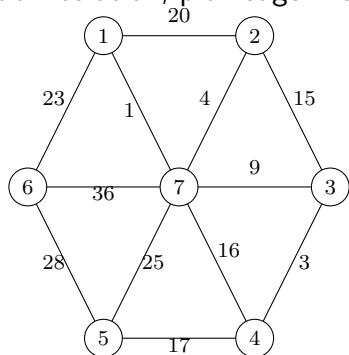


Figure : Graph **G**

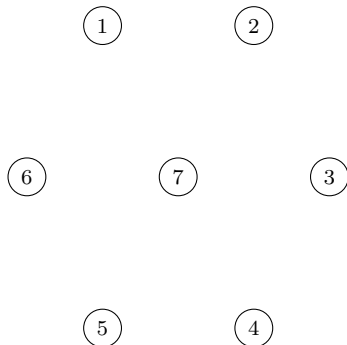
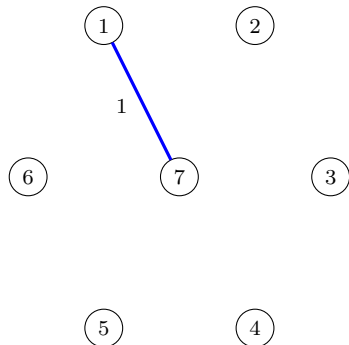
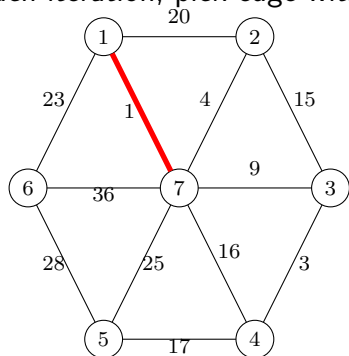


Figure : **MST** of **G**

Back

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 .



Back

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**.

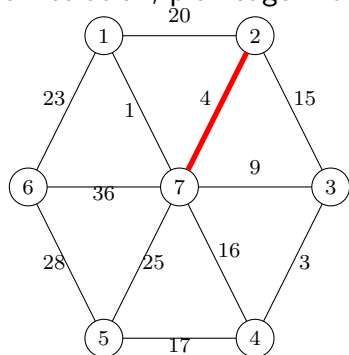


Figure : Graph **G**

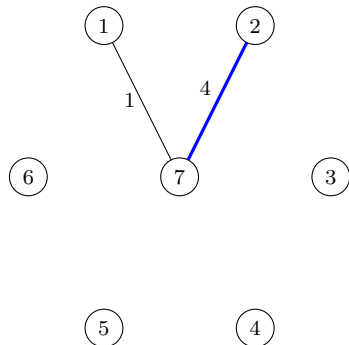


Figure : MST of **G**

Back

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**.

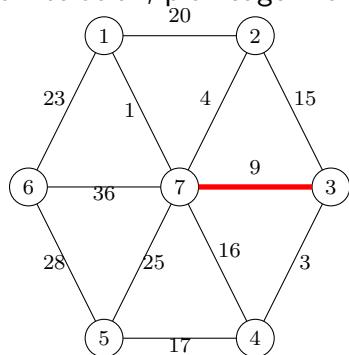


Figure : Graph **G**

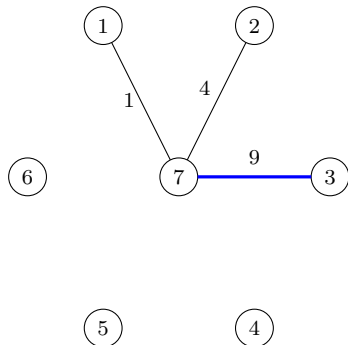


Figure : MST of **G**

Back

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**.

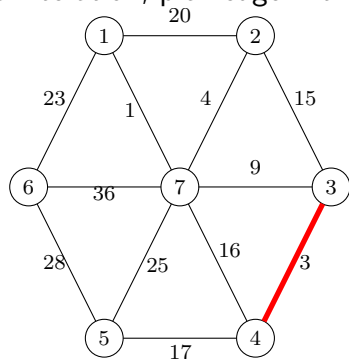


Figure : Graph **G**

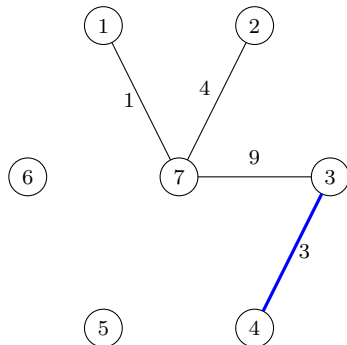


Figure : **MST** of **G**

Back

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**.

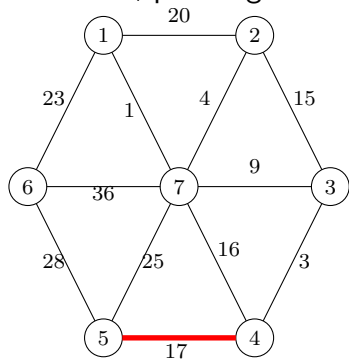


Figure : Graph **G**

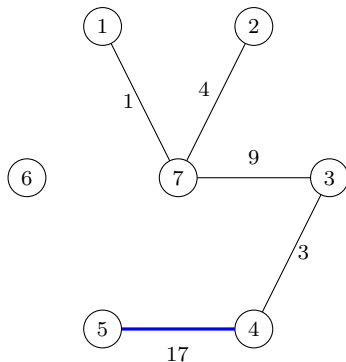


Figure : **MST** of **G**

Back

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**.

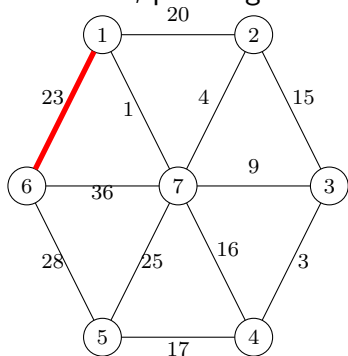


Figure : Graph **G**

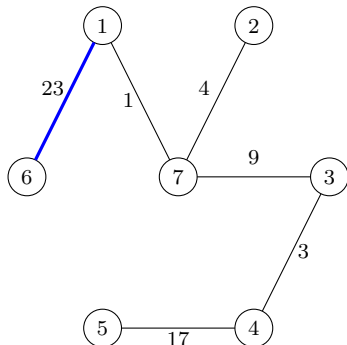


Figure : **MST** of **G**

Back

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 do  
    choose  $i \in E$  of largest cost  
    if removing i does not disconnect T then  
        remove i from T  
return the set T
```

Returns a minimum spanning tree.

Back

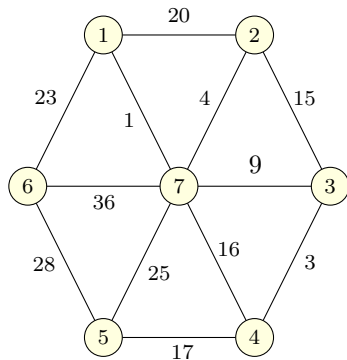
Borůvka's Algorithm

Simplest to implement. See notes.

Assume G is a connected graph.

```
T is  $\emptyset$  (* T will store edges of a MST *)  
while T is not spanning do  
    X  $\leftarrow \emptyset$   
    for each connected component S of T do  
        add to X the cheapest edge between S and  $V \setminus S$   
    Add edges in X to T  
return the set T
```

Borůvka's Algorithm



Correctness of MST Algorithms

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

Assumption

And for now . . .

Assumption

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

Definition

Given a graph $G = (V, E)$, a **cut** is a partition of the vertices of the graph into two sets $(S, V \setminus S)$.

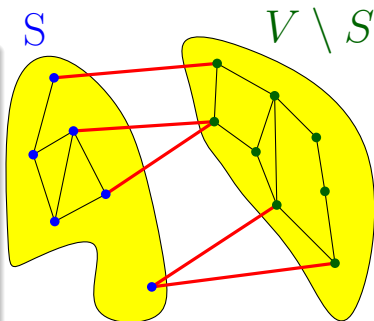
Cuts

Definition

Given a graph $G = (V, E)$, a **cut** is a partition of the vertices of the graph into two sets $(S, V \setminus S)$.

Edges having an endpoint on both sides are the **edges of the cut**.

A cut edge is **crossing** the cut.



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.

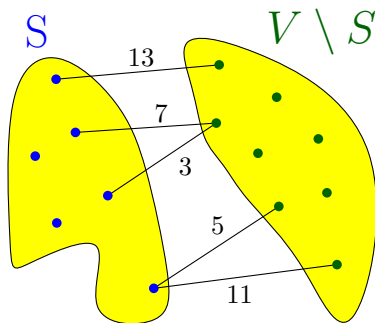
Proof.

Exercise. □

Safe edge

Example...

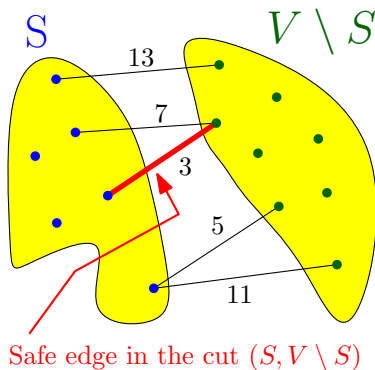
Every cut identifies one safe edge...



Safe edge

Example...

Every cut identifies one safe edge...



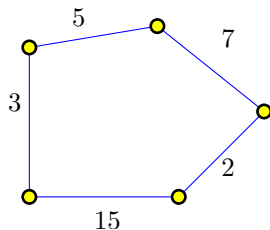
...the cheapest edge in the cut.

Note: An edge e may be a safe edge for *many* cuts!

Unsafe edge

Example...

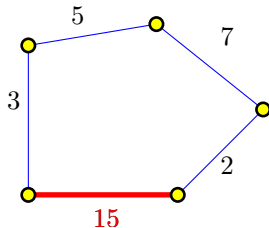
Every cycle identifies one **unsafe** edge...



Unsafe edge

Example...

Every cycle identifies one **unsafe** edge...



...the most expensive edge in the cycle.

Example

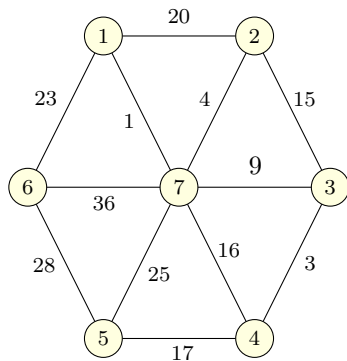


Figure : Graph with unique edge costs. Safe edges are red, rest are unsafe.

Example

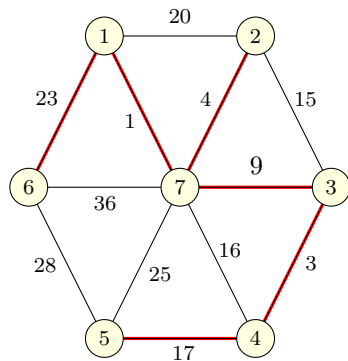


Figure : Graph with unique edge costs. Safe edges are red, rest are unsafe.

Example

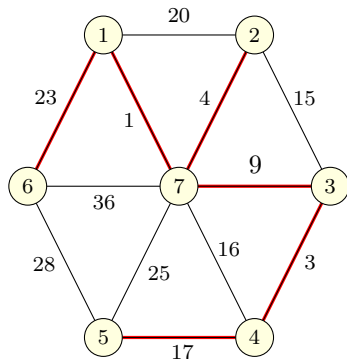


Figure : Graph with unique edge costs. Safe edges are red, rest are unsafe.

And all safe edges are in the **MST** in this case...

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.

- 1 Suppose (for contradiction) e is not in **MST** T .
- 2 Since e is safe there is an $S \subset V$ such that e is the unique minimum cost edge crossing S .
- 3 Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$.
- 4 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 .

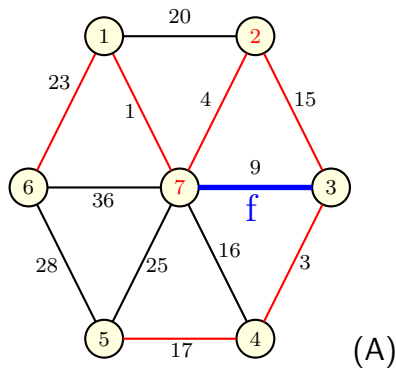
Proof.

- 1 Suppose (for contradiction) e is not in **MST T** .
- 2 Since e is safe there is an $S \subset V$ such that e is the unique minimum cost edge crossing S .
- 3 Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$.
- 4 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

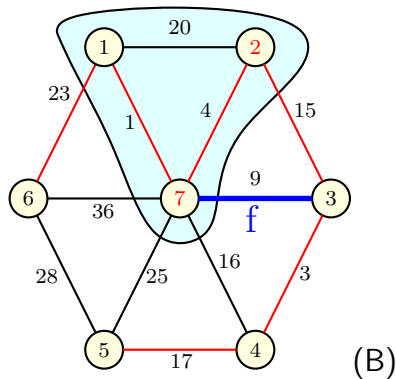
Problematic example. $S = \{1, 2, 7\}$, $e = (7, 3)$, $f = (1, 6)$. $T - f + e$ is not a spanning tree.



- 1 (A) Consider adding the edge f .

Error in Proof: Example

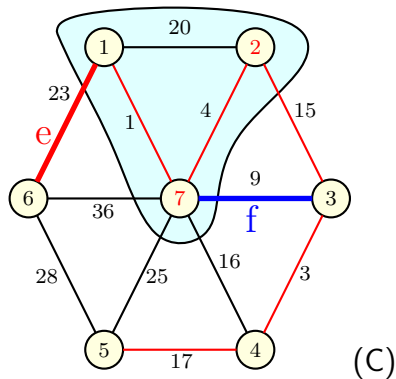
Problematic example. $S = \{1, 2, 7\}$, $e = (7, 3)$, $f = (1, 6)$. $T - f + e$ is not a spanning tree.



- 1 (A) Consider adding the edge f .
- 2 (B) It is safe because it is the cheapest edge in the cut.

Error in Proof: Example

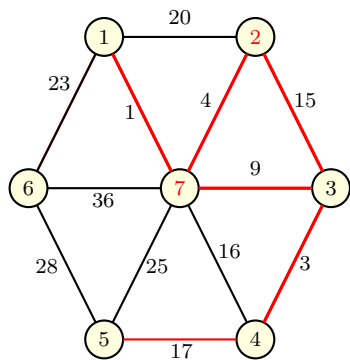
Problematic example. $S = \{1, 2, 7\}$, $e = (7, 3)$, $f = (1, 6)$. $T - f + e$ is not a spanning tree.



- 1 (A) Consider adding the edge **f**.
- 2 (B) It is safe because it is the cheapest edge in the cut.
- 3 (C) Lets throw out the edge **e** currently in the spanning tree which is more expensive than **f** and is in the same cut. Put it **f** instead...

Error in Proof: Example

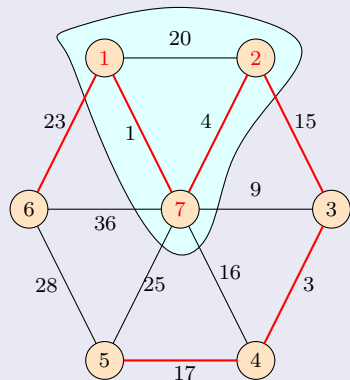
Problematic example. $S = \{1, 2, 7\}$, $e = (7, 3)$, $f = (1, 6)$. $T - f + e$ is not a spanning tree.



- 1 (A) Consider adding the edge f .
- 2 (B) It is safe because it is the cheapest edge in the cut.
- 3 (C) Lets throw out the edge e currently in the spanning tree which is more expensive than f and is in the same cut. Put it f instead...
- 4 (D) New graph of selected edges is not a tree anymore. BUG.

Proof of Cut Property

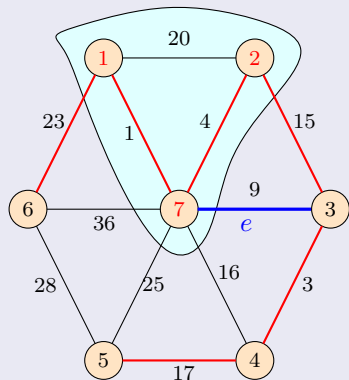
Proof.



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

Proof of Cut Property

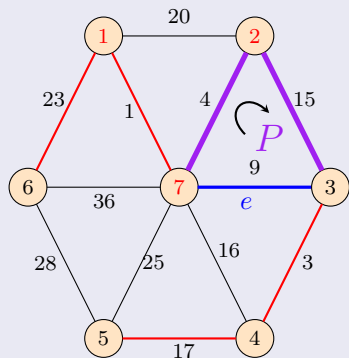
Proof.



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

Proof of Cut Property

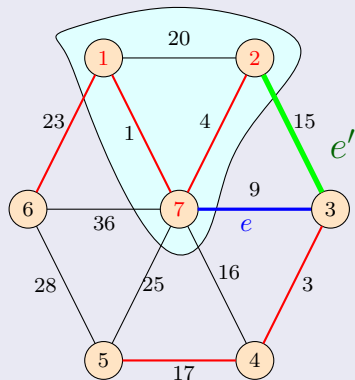
Proof.



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

Proof of Cut Property

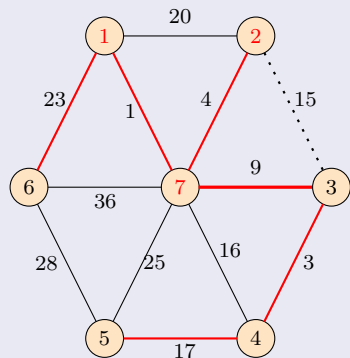
Proof.



- 1 Suppose $e = (v, w)$ is not in **MST T** and e is min weight edge in cut $(S, V \setminus S)$. Assume $v \in S$.
- 2 **T** is spanning tree: there is a unique path **P** from v to w in **T**
- 3 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.



- 1 Suppose $e = (v, w)$ is not in **MST** T and e is min weight edge in cut $(S, V \setminus S)$. Assume $v \in S$.
- 2 T is spanning tree: there is a unique path P from v to w in T
- 3 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 $e' = (v', w')$
- 4 $T' = (T \setminus \{e'\}) \cup \{e\}$ 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.

- 1 Suppose not. Let S be a connected component in the graph induced by the safe edges.
- 2 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.

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.

- 1 If e is added to tree, then e is safe and belongs to every **MST**.
- 2 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.

- 1 If e is added to tree, then e is safe and belongs to every **MST**.
 - 1 Let S be the vertices connected by edges in T when e is added.
- 2 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.

- 1 If e is added to tree, then e is safe and belongs to every **MST**.
 - 1 Let S be the vertices connected by edges in T when e is added.
 - 2 e is edge of lowest cost with one end in S and the other in $V \setminus S$ and hence e is safe.
- 2 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.

- ① If e is added to tree, then e is safe and belongs to every **MST**.
 - ① Let S be the vertices connected by edges in T when e is added.
 - ② e is edge of lowest cost with one end in S and the other in $V \setminus S$ and hence e is safe.
- ② Set of edges output is a spanning tree
 - ① 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.

- ① If e is added to tree, then e is safe and belongs to every **MST**.
 - ① Let S be the vertices connected by edges in T when e is added.
 - ② e is edge of lowest cost with one end in S and the other in $V \setminus S$ and hence e is safe.
- ② Set of edges output is a spanning tree
 - ① Set of edges output forms a connected graph: by induction, S is connected in each iteration and eventually $S = V$.
 - ② 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.

- 1 If $e = (u, v)$ is added to tree, then e is safe
- 2 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.

- 1 If $e = (u, v)$ is added to tree, then e is safe
 - 1 When algorithm adds e let S and S' be the connected components containing u and v respectively

- 2 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.

- 1 If $e = (u, v)$ is added to tree, then e is safe
 - 1 When algorithm adds e let S and S' be the connected components containing u and v respectively
 - 2 e is the lowest cost edge crossing S (and also S').

- 2 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.

- 1 If $e = (u, v)$ is added to tree, then e is safe
 - 1 When algorithm adds e let S and S' be the connected components containing u and v respectively
 - 2 e is the lowest cost edge crossing S (and also S').
 - 3 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
- 2 Set of edges output is a spanning tree : exercise



Correctness of Borůvka's Algorithm

Proof of correctness.

Argue that only safe edges are added.

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.

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

- 1 $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or $(c(e_i) = c(e_j)$ and $i < j)$
- 2 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 $B \setminus A)$
- 3 Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique **MST**.

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

- ① $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or $(c(e_i) = c(e_j)$ and $i < j)$
- ② 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 $B \setminus A)$
- ③ Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique **MST**.

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

- 1 $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or $(c(e_i) = c(e_j)$ and $i < j)$
- 2 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 $B \setminus A)$
- 3 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

- 1 Algorithms and proofs don't assume that edge costs are non-negative! **MST** algorithms work for arbitrary edge costs.
- 2 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 **MST**s but not for shortest paths?
- 3 Can compute *maximum* weight spanning tree by negating edge costs and then computing an MST.

Edge Costs: Positive and Negative

- 1 Algorithms and proofs don't assume that edge costs are non-negative! **MST** algorithms work for arbitrary edge costs.
- 2 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 **MST**s but not for shortest paths?
- 3 Can compute *maximum* weight spanning tree by negating edge costs and then computing an MST.

Question: Why does this not work for shortest paths?

Part II

Data Structures for MST: Priority Queues and Union-Find

Implementing Borůvka's Algorithm

No complex data structure needed.

```
T is  $\emptyset$  (* T will store edges of a MST *)  
while T is not spanning do  
    X  $\leftarrow \emptyset$   
    for each connected component S of T do  
        add to X the cheapest edge between S and V \ S  
    Add edges in X to T  
return the set T
```

- $O(\log n)$ iterations of while loop. Why?

Implementing Borůvka's Algorithm

No complex data structure needed.

```
T is  $\emptyset$  (* T will store edges of a MST *)  
while T is not spanning do  
    X  $\leftarrow \emptyset$   
    for each connected component S of T do  
        add to X the cheapest edge between S and V \ S  
    Add edges in X to T  
return the set T
```

- $O(\log n)$ iterations of while loop. Why? Number of connected components shrink by at least half since each component merges with one or more other components.
- Each iteration can be implemented in $O(m)$ time.

Implementing Borůvka's Algorithm

No complex data structure needed.

```
T is  $\emptyset$  (* T will store edges of a MST *)  
while T is not spanning do  
    X  $\leftarrow \emptyset$   
    for each connected component S of T do  
        add to X the cheapest edge between S and V \ S  
    Add edges in X to T  
return the set T
```

- $O(\log n)$ iterations of while loop. Why? Number of connected components shrink by at least half since each component merges with one or more other components.
- Each iteration can be implemented in $O(m)$ time.

Running time: $O(m \log n)$ time.

Implementing Prim's Algorithm

Implementing Prim's Algorithm

Prim_ComputeMST

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** **do**

 pick **e** = (**v**, **w**) \in **E** such that

v \in **S** and **w** \in **V** - **S**

e has minimum cost

T = **T** \cup **e**

S = **S** \cup **w**

return the set **T**

Analysis

Implementing Prim's Algorithm

Implementing Prim's Algorithm

Prim_ComputeMST

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** **do**

 pick **e** = (**v**, **w**) \in **E** such that

v \in **S** and **w** \in **V** - **S**

e has minimum cost

T = **T** \cup **e**

S = **S** \cup **w**

return the set **T**

Analysis

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

Implementing Prim's Algorithm

Implementing Prim's Algorithm

Prim_ComputeMST

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** **do**

pick **e** = (**v**, **w**) \in **E** such that

v \in **S** and **w** \in **V** - **S**

e has minimum cost

T = **T** \cup **e**

S = **S** \cup **w**

return the set **T**

Analysis

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

Implementing Prim's Algorithm

Implementing Prim's Algorithm

Prim_ComputeMST

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** **do**

 pick **e** = (**v**, **w**) \in **E** such that

v \in **S** and **w** \in **V** - **S**

e has minimum cost

T = **T** \cup **e**

S = **S** \cup **w**

return the set **T**

Analysis

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

Implementing Prim's Algorithm

More Efficient Implementation

Prim_ComputeMST

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 $S \neq V$ **do**

 pick v with minimum $a(v)$

$T = T \cup \{(e(v), v)\}$

$S = S \cup \{v\}$

 update arrays a and e

return the set **T**

Implementing Prim's Algorithm

More Efficient Implementation

Prim_ComputeMST

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 $S \neq V$ **do**

 pick v with minimum $a(v)$

$T = T \cup \{(e(v), v)\}$

$S = S \cup \{v\}$

 update arrays a and e

return the set **T**

Implementing Prim's Algorithm

More Efficient Implementation

Prim_ComputeMST

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 $S \neq V$ **do**

 pick v with minimum $a(v)$

$T = T \cup \{(e(v), v)\}$

$S = S \cup \{v\}$

 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

- 1 **makeQ**: create an empty queue
- 2 **findMin**: find the minimum key in S
- 3 **extractMin**: Remove $v \in S$ with smallest key and return it
- 4 **add**($v, k(v)$): Add new element v with key $k(v)$ to S
- 5 **Delete**(v): Remove element v from S
- 6 **decreaseKey** ($v, k'(v)$): decrease key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$
- 7 **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 $S \neq V$ **do**

 pick **v** with minimum $a(v)$

$T = T \cup \{(e(v), v)\}$

$S = S \cup \{v\}$

 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 $S \neq V$ **do**

 pick v with minimum $a(v)$

$T = T \cup \{(e(v), v)\}$

$S = S \cup \{v\}$

 update arrays a and e

return the set **T**

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

- 1 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 $S \neq V$ **do**

 pick v with minimum $a(v)$

$T = T \cup \{(e(v), v)\}$

$S = S \cup \{v\}$

 update arrays a and e

return the set T

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

- 1 Requires $O(n)$ **extractMin** operations
- 2 Requires $O(m)$ **decreaseKey** operations

Running time of Prim's Algorithm

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

- ① Using standard Heaps, **extractMin** and **decreaseKey** take $O(\log n)$ time. Total: $O((m + n) \log n)$
- ② 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

- ① Using standard Heaps, **extractMin** and **decreaseKey** take $O(\log n)$ time. Total: $O((m + n) \log n)$
- ② 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

Kruskal_ComputeMST

```
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 do  
    choose  $e \in E$  of minimum cost  
    if ( $T \cup \{e\}$  does not have cycles)  
        add e to T  
return the set T
```

Kruskal's Algorithm

Kruskal_ComputeMST

```
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 do  
    choose  $e \in E$  of minimum cost  
    if ( $T \cup \{e\}$  does not have cycles)  
        add e to T  
return the set T
```

Kruskal's Algorithm

Kruskal_ComputeMST

```
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 do  
    choose  $e \in E$  of minimum cost  
    if ( $T \cup \{e\}$  does not have cycles)  
        add e to T  
return the set T
```

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

Kruskal's Algorithm

Kruskal_ComputeMST

```
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 do  
    choose  $e \in E$  of minimum cost  
    if ( $T \cup \{e\}$  does not have cycles)  
        add e to T  
return the set T
```

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

Kruskal's Algorithm

Kruskal_ComputeMST

```
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 do  
    choose  $e \in E$  of minimum cost  
    if ( $T \cup \{e\}$  does not have cycles)  
        add e to T  
return the set T
```

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

Kruskal's Algorithm

Kruskal_ComputeMST

```
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 do  
    choose  $e \in E$  of minimum cost  
    if ( $T \cup \{e\}$  does not have cycles)  
        add e to T  
return the set T
```

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

Implementing Kruskal's Algorithm Efficiently

Kruskal_ComputeMST

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 **do**

 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

Kruskal_ComputeMST

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 **do**

 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

Kruskal_ComputeMST

```
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 do
    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.

MST for really sparse graphs?

Given a graph G with n vertices, and $n + 20$ edges, its **MST** can be computed in

- (A) $O(n^2)$.
- (B) $O(n \log n)$.
- (C) $O(n \log \log n)$.
- (D) $O(n \log^* n)$.
- (E) $O(n)$.

Union-Find Data Structure

Data Structure

Store disjoint sets of elements that supports the following operations

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

Assumption: **S** is indexed by integers **1** to **|S|**.

Union-Find Data Structure

Data Structure

Store disjoint sets of elements that supports the following operations

- 1 **makeUnionFind(S)** returns a data structure where each element of **S** is in a separate set
- 2 **find(u)** returns the *name* of set containing element **u**. Thus, **u** and **v** belong to the same set if and only if **find(u) = find(v)**

Assumption: **S** is indexed by integers **1** to **|S|**.

Union-Find Data Structure

Data Structure

Store disjoint sets of elements that supports the following operations

- 1 **makeUnionFind(S)** returns a data structure where each element of **S** is in a separate set
- 2 **find(u)** returns the *name* of set containing element **u**. Thus, **u** and **v** belong to the same set if and only if **find(u) = find(v)**
- 3 **union(A, B)** merges two sets **A** and **B**. Here **A** and **B** are the names of the sets. Typically the name of a set is some element in the set.

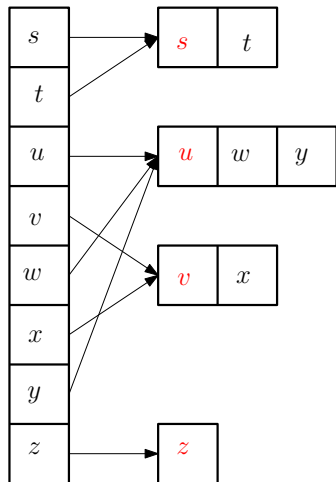
Assumption: **S** is indexed by integers **1** to **|S|**.

Implementing Union-Find using Arrays and Lists

Using lists

- ① Each set stored as list with a name associated with the list.
- ② For each element $u \in S$ a pointer to the its set. Array for pointers: `component[u]` is pointer for u .
- ③ **makeUnionFind** (S) takes $O(n)$ time and space.

Example



Implementing Union-Find using Arrays and Lists

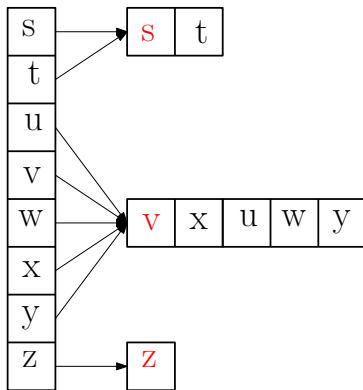
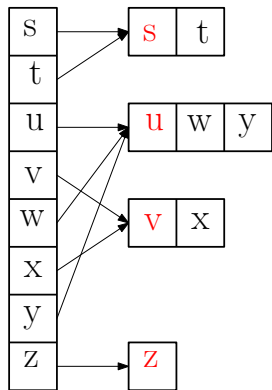
- 1 **find(u)** reads the entry `component[u]`: **$O(1)$** time

Implementing Union-Find using Arrays and Lists

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

Implementing Union-Find using Arrays and Lists

- 1 **find**(**u**) reads the entry component[**u**]: **O(1)** time
- 2 **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)`:

- 1 with each set, keep track of its size
- 2 assume $|A| \leq |B|$ for now
- 3 Merge the list of `A` into that of `B`: $O(1)$ time (linked lists)
- 4 Update `component[u]` only for elements in the smaller set `A`
- 5 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)`:

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

Improving the List Implementation for Union

New Implementation

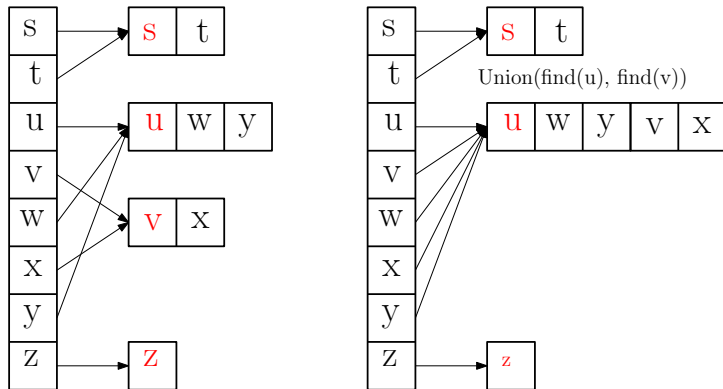
As before use `component[u]` to store set of `u`.

Change to `union(A,B)`:

- 1 with each set, keep track of its size
- 2 assume $|A| \leq |B|$ for now
- 3 Merge the list of `A` into that of `B`: $O(1)$ time (linked lists)
- 4 Update `component[u]` only for elements in the smaller set `A`
- 5 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)

Mergers

Consider an element x . Assume x is in a set X , and let Y be a bigger set. After $\text{union}(X, Y)$ the size of the set containing x is at least:

- (A) At least double what it was.
- (B) Same.
- (C) Maybe bigger, maybe the same size.
- (D) $|X| * |Y|$.
- (E) $|X|(|Y| - |X|)$.

Mergers

Consider starting with n singletons. Consider an element x . The element x can participate in at most

- (A) $\Theta(1)$.
- (B) $\Theta(\log n)$.
- (C) $\Theta(\sqrt{n})$.
- (D) $\Theta(n)$.

mergers where it belongs to the smaller set, throughout the execution of **Union-Find**.

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 $\text{makeUnionFind}(S)$ 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 $\text{makeUnionFind}(S)$ 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

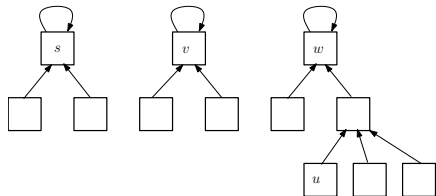
union(**A**,**B**) takes $O(|\mathbf{A}|)$ time where $|\mathbf{A}| \leq |\mathbf{B}|$. Size of new set is $\geq 2|\mathbf{A}|$. Cannot double too many times.

Proof of Theorem

Proof.

- 1 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
- 2 Also, maximum size of any set (after k unions) is $2k$
- 3 **union**(A, B) takes $O(|A|)$ time where $|A| \leq |B|$.
- 4 *Charge* each element in A constant time to *pay* for $O(|A|)$ time.
- 5 How much does any element get charged?
- 6 If component[v] is updated, set containing v *doubles* in size
- 7 component[v] is updated at most $\log 2k$ times
- 8 Total number of updates is $2k \log 2k = O(k \log k)$ □

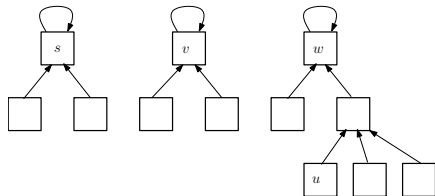
Improving Worst Case Time



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

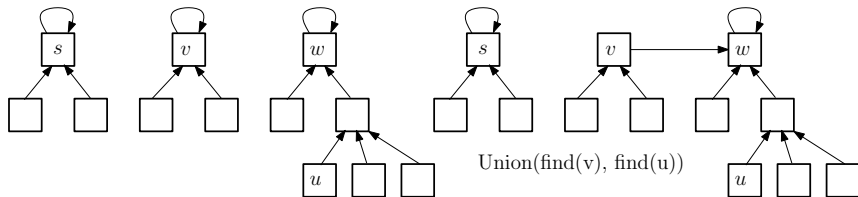


Better data structure

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

- 1 **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.

- 1 **find(u)**: Traverse from u to the root
- 2 **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  $S$  do
     $\text{parent}(u) = u$ 
```

Details of Implementation

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

```
makeUnionFind(S)  
  for each  $u$  in  $S$  do  
     $\text{parent}(u) = u$ 
```

```
find( $u$ )  
  while ( $\text{parent}(u) \neq u$ ) do  
     $u = \text{parent}(u)$   
  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  $S$  do
     $\text{parent}(u) = u$ 
```

```
find(u)
  while ( $\text{parent}(u) \neq u$ ) do
     $u = \text{parent}(u)$ 
  return  $u$ 
```

```
union(component(u), component(v))
  (*  $\text{parent}(u) = u$  &  $\text{parent}(v) = v$  *)
  if ( $|\text{component}(u)| \leq |\text{component}(v)|$ ) then
     $\text{parent}(u) = v$ 
  else
     $\text{parent}(v) = u$ 
  set new component size to  $|\text{component}(u)| + |\text{component}(v)|$ 
```

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 $O(\log n)$.

Proof.

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

Further Improvements: Path Compression

Observation

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

Further Improvements: Path Compression

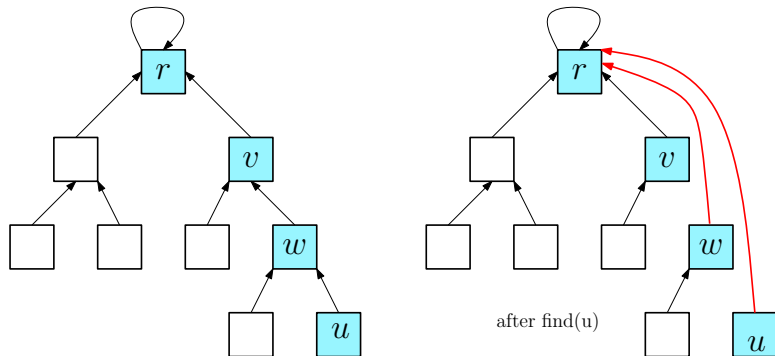
Observation

*Consecutive calls of **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 **find(u)** point to root.

Path Compression: Example



Path Compression

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

Path Compression

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

Question

Does Path Compression help?

Path Compression

```
find(u) :  
  if (parent(u) ≠ u) then  
    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 α is the **inverse Ackermann function**.*

Ackermann and Inverse Ackermann Functions

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

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

Ackermann and Inverse Ackermann Functions

Ackermann 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 Ackermann function defined as

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

Ackermann and Inverse Ackermann Functions

Ackermann 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 Ackermann 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 **Union-Find**, *any* data structure in the pointer model requires $\Omega(m\alpha(m, n))$ time for m operations.

Running time of Kruskal's Algorithm

Using Union-Find data structure:

- 1 $O(m)$ **find** operations (two for each edge)
- 2 $O(n)$ **union** operations (one for each edge added to **T**)
- 3 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?

- 1 $O(m \log^* m)$ time ?.
- 2 $O(m + n)$ time using bit operations in RAM model ?.
- 3 $O(m + n)$ expected time (randomized algorithm) ?.
- 4 $O((n + m)\alpha(m, n))$ time ?.
- 5 Still open: Is there an $O(n + m)$ time deterministic algorithm in the comparison model?