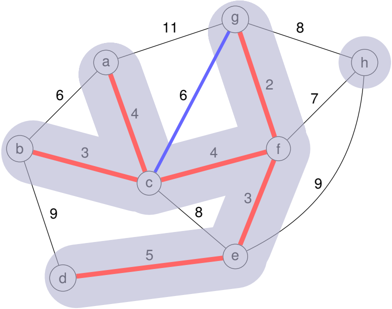



# 6.3: Minimum Spanning Tree

Frank Stajano Thomas Sauerwald

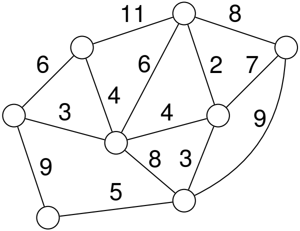
Lent 2015

UNIVERSITY OF CAMBRIDGE

# Minimum Spanning Tree Problem

**Minimum Spanning Tree Problem**

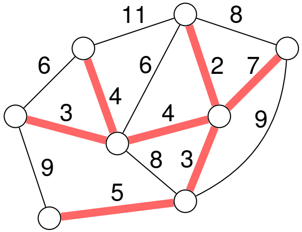
- Given: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights



# Minimum Spanning Tree Problem

**Minimum Spanning Tree Problem**

- Given: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights
- Goal: Find a subgraph $\subseteq E$ of minimum total weight that links all vertices

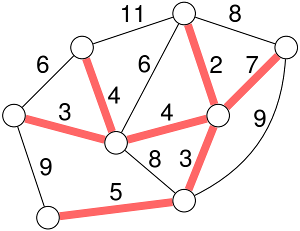


# Minimum Spanning Tree Problem

**Minimum Spanning Tree Problem**

- Given: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights
- Goal: Find a subgraph $\subseteq E$ of minimum total weight that links all vertices

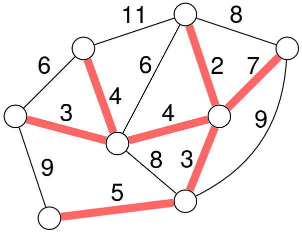
Must be necessarily a tree!



# Minimum Spanning Tree Problem

**Minimum Spanning Tree Problem**

- Given: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights
- Goal: Find a subgraph $\subseteq E$ of minimum total weight that links all vertices




**Applications**

- Street Networks, Wiring Electronic Components, Laying Pipes
- Weights may represent distances, costs, travel times, capacities, resistance etc.

# Generic Algorithm

```
def minimum spanningTree(G)
  A = empty set of edges
  while A does not span all vertices yet:
    add a safe edge to A
```

# Generic Algorithm

```
def minimum spanningTree(G)
  A = empty set of edges
  while A does not span all vertices yet:
    add a safe edge to A
```

Definition

An edge of $G$ is safe if by adding the edge to $A$, the resulting subgraph is still a subset of a minimum spanning tree.

## Generic Algorithm

```
def minimum spanningTree(G)
  A = empty set of edges
  while A does not span all vertices yet:
    add a safe edge to A
```

**Definition**

An edge of $G$ is safe if by adding the edge to $A$, the resulting subgraph is still a subset of a minimum spanning tree.

**How to find a safe edge?**

# Finding safe edges

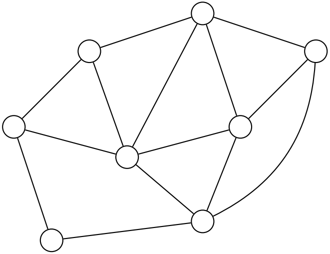
Definitions

- a cut is a partition of $V$ into at least two disjoint sets

# Finding safe edges

Definitions

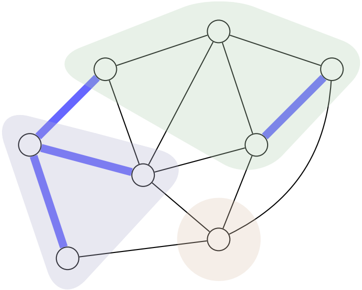
- a cut is a partition of $V$ into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of $A$ goes across the cut



# Finding safe edges

Definitions

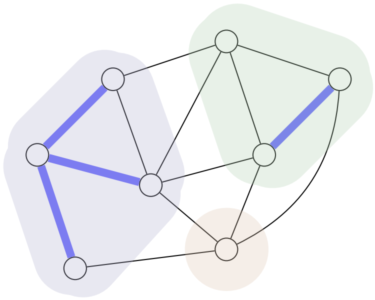
- a cut is a partition of $V$ into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of $A$ goes across the cut



# Finding safe edges

Definitions

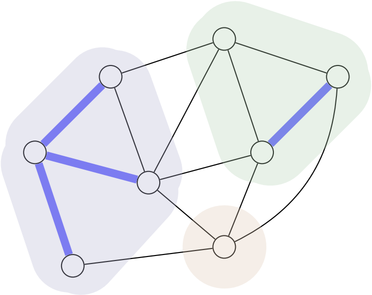
- a cut is a partition of $V$ into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of $A$ goes across the cut



# Finding safe edges

**Definitions**

- a cut is a partition of $V$ into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of $A$ goes across the cut



**Theorem**

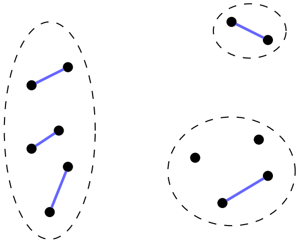
Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

## Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:



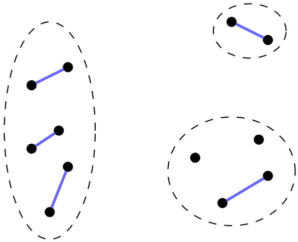
## Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$



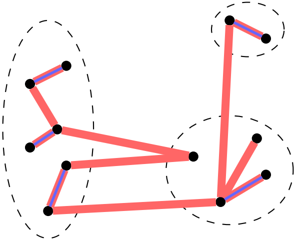
# Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$



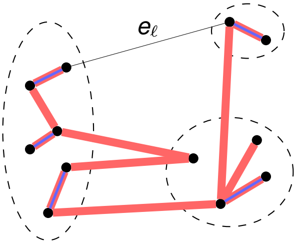
# Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut



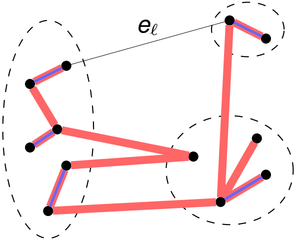
# Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done



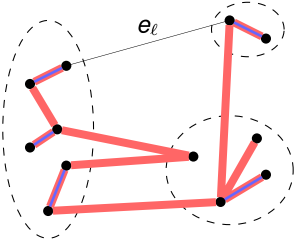
# Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$,



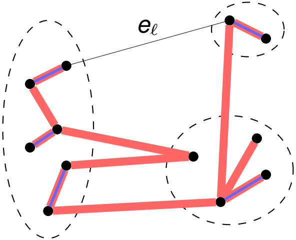
# Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \not\in T$, then adding $e_\ell$ to $T$ introduces cycle



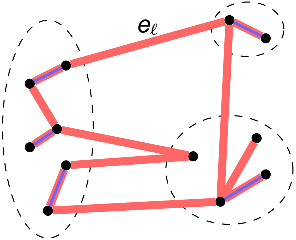
# Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle



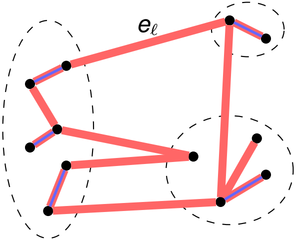
# Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$



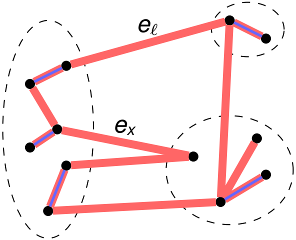
## Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \not\in T$, then adding $e_\ell$ to $T$ introduces cycle
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$



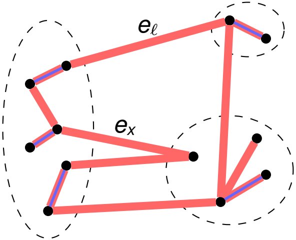
# Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$
- Consider now the tree $T \cup e_\ell \setminus e_x$:



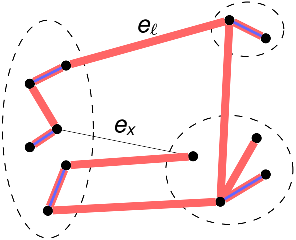
# Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$
- Consider now the tree $T \cup e_\ell \setminus e_x$:



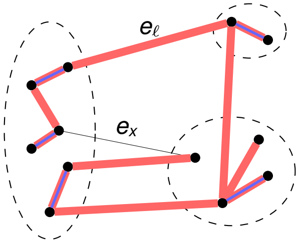
# Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$
- Consider now the tree $T \cup e_\ell \setminus e_x$:
  - This tree must be a spanning tree



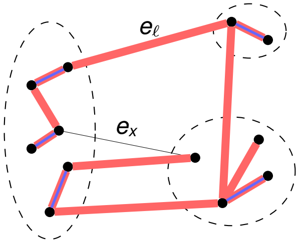
# Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$
- Consider now the tree $T \cup e_\ell \setminus e_x$:
  - This tree must be a spanning tree
  - If $w(e_\ell) < w(e_x)$, then this spanning tree has smaller cost than $T$ (can't happen!)



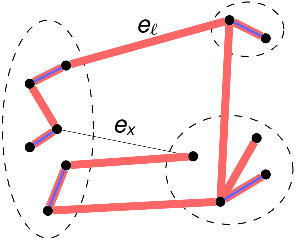
## Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

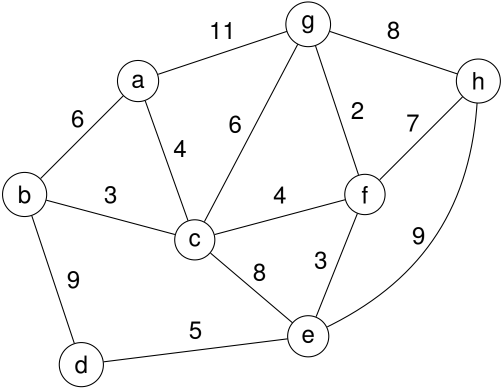
Proof:

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$
- Consider now the tree $T \cup e_\ell \setminus e_x$:
    - This tree must be a spanning tree
    - If $w(e_\ell) < w(e_x)$, then this spanning tree has smaller cost than $T$ (can't happen!)
    - If $w(e_\ell) = w(e_x)$, then $T \cup e_\ell \setminus e_x$ is a MST. $\square$



## Glimpse at Kruskal's Algorithm

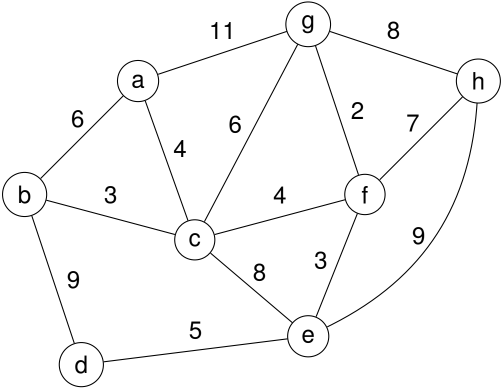
Basic Strategy



# Glimpse at Kruskal's Algorithm

Basic Strategy

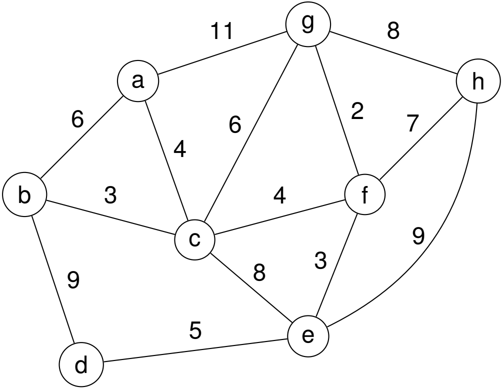
- Let $A \subseteq E$ be a forest, intially empty



# Glimpse at Kruskal's Algorithm

Basic Strategy

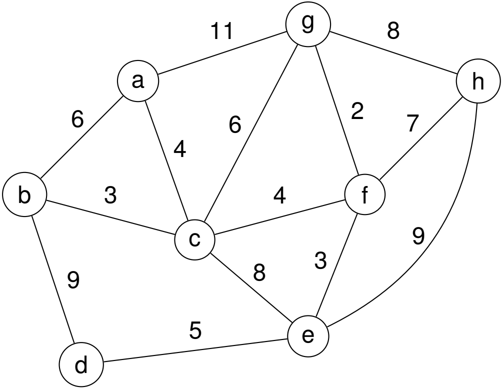
- Let $A \subseteq E$ be a forest, intially empty
- At every step,



# Glimpse at Kruskal's Algorithm

Basic Strategy

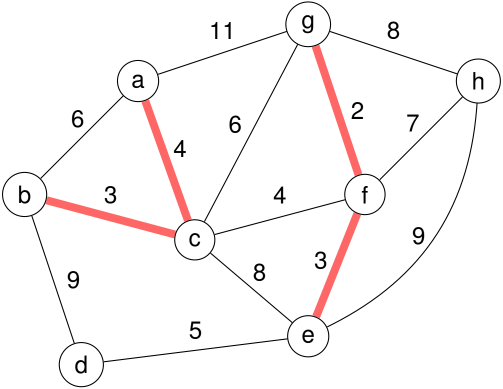
- Let $A \subseteq E$ be a forest, intially empty
- At every step, given $A$, perform:



# Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given $A$, perform:

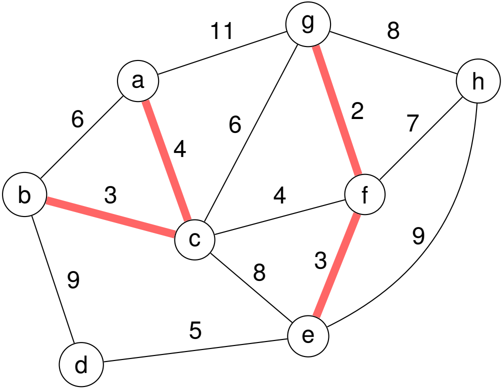


# Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given $A$, perform:

  Add lightest edge to $A$ that does not introduce a cycle

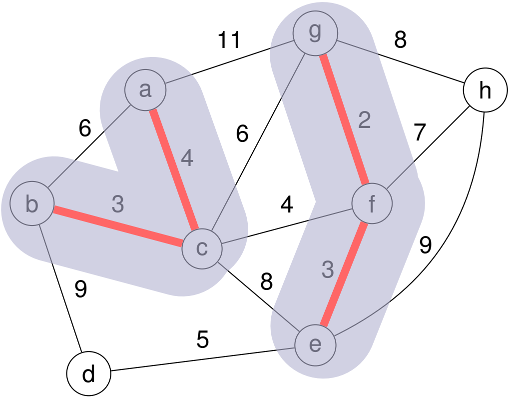


## Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given $A$, perform:

  Add lightest edge to $A$ that does not introduce a cycle

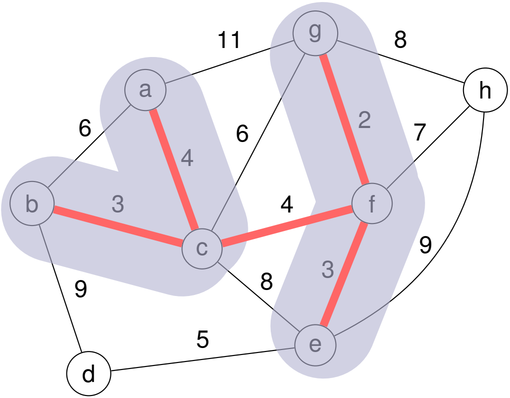


## Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given $A$, perform:

  Add lightest edge to $A$ that does not introduce a cycle

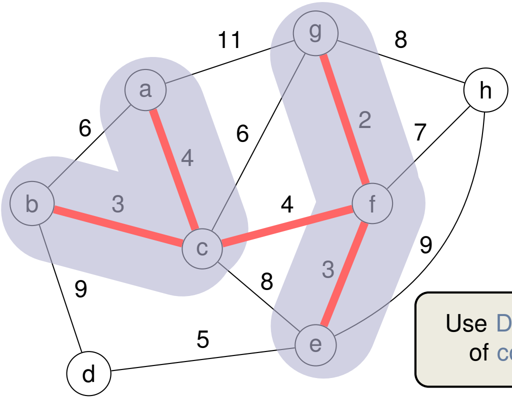


# Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given $A$, perform:

  Add lightest edge to $A$ that does not introduce a cycle



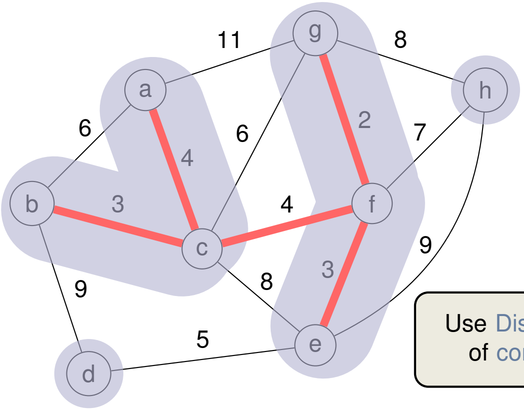

Use Disjoint Sets to keep track of connected components!

## Glimpse at Kruskal's Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, intially empty
- At every step, given $A$, perform:

  Add lightest edge to $A$ that does not introduce a cycle



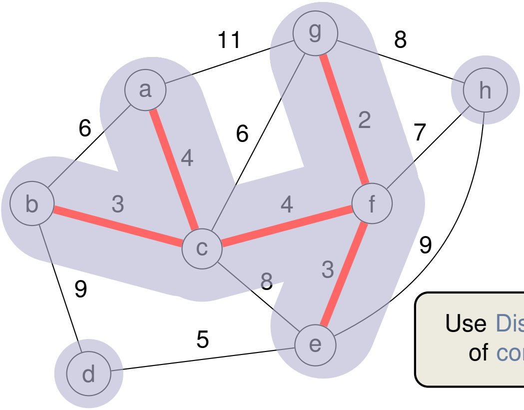

Use Disjoint Sets to keep track of connected components!

# Glimpse at Kruskal's Algorithm

**Basic Strategy**

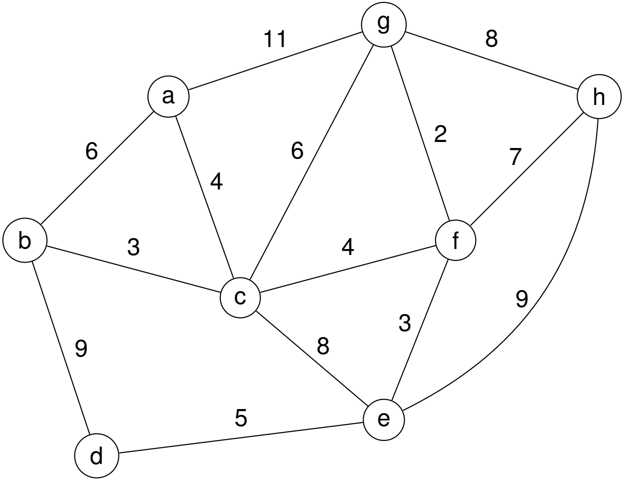
- Let $A \subseteq E$ be a forest, intially empty
- At every step, given $A$, perform:

  Add lightest edge to $A$ that does not introduce a cycle

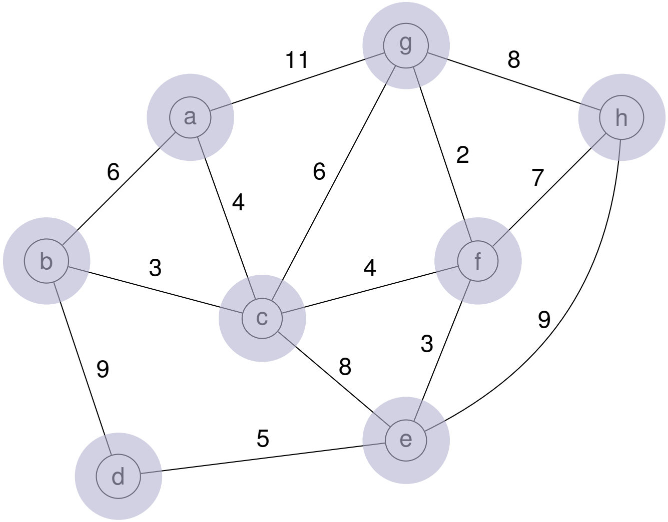



Use Disjoint Sets to keep track of connected components!

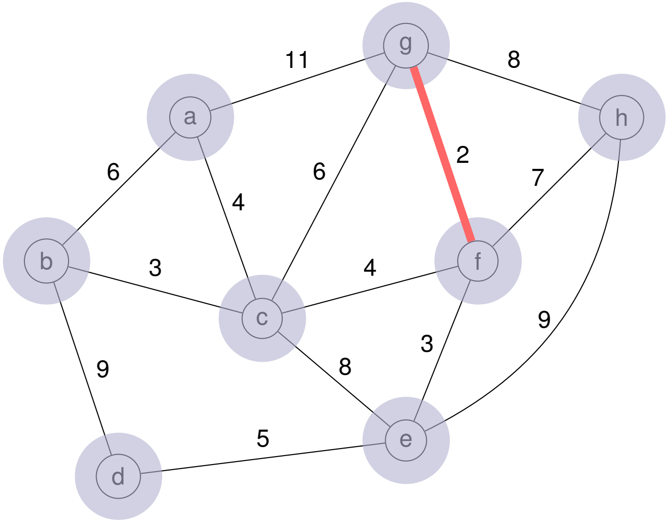
# Complete Run of Kruskal's Algorithm



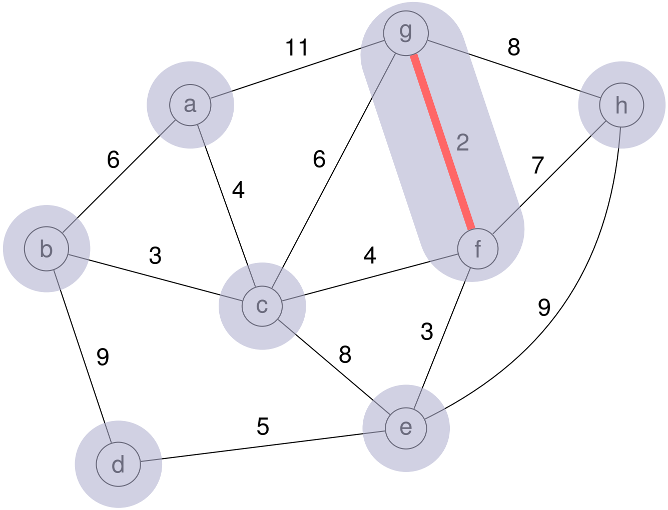
# Complete Run of Kruskal's Algorithm



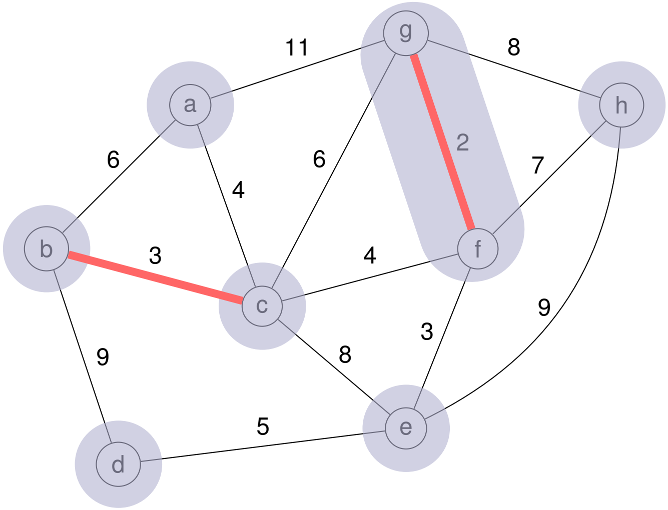
# Complete Run of Kruskal's Algorithm



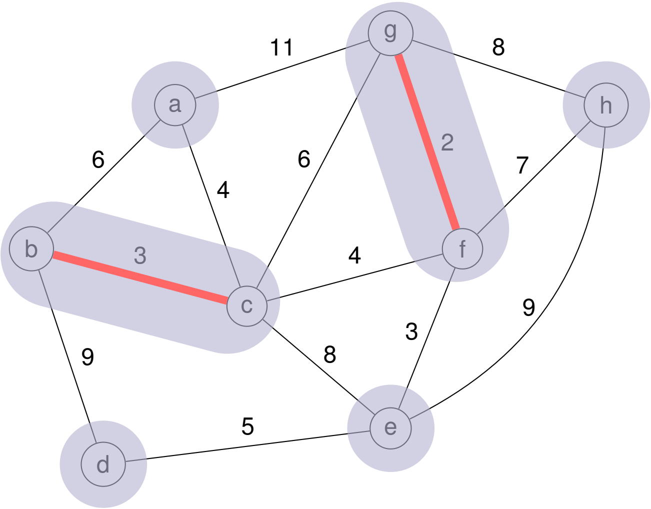
# Complete Run of Kruskal's Algorithm



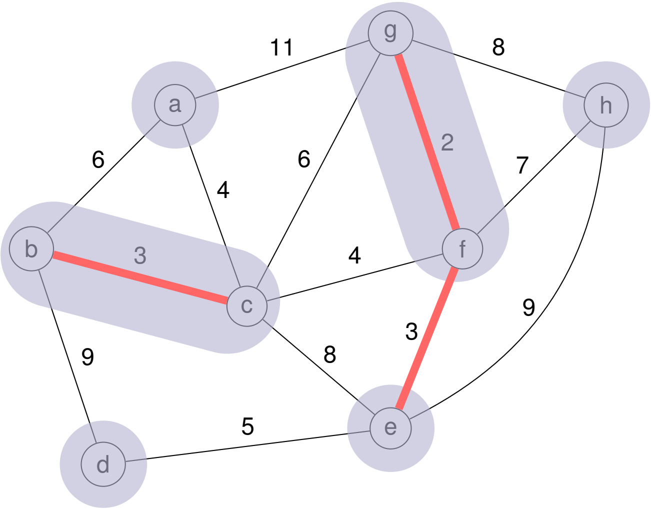
# Complete Run of Kruskal's Algorithm



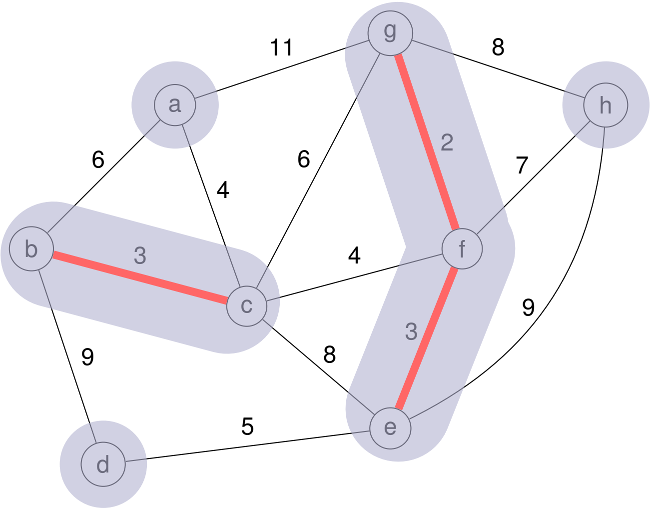
# Complete Run of Kruskal's Algorithm



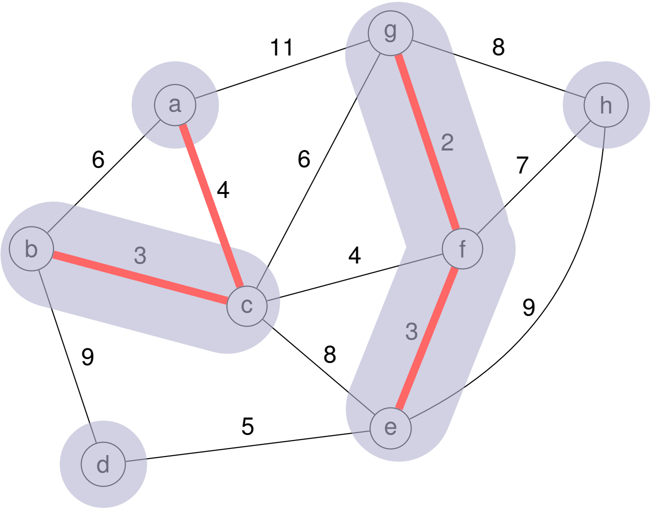
# Complete Run of Kruskal's Algorithm



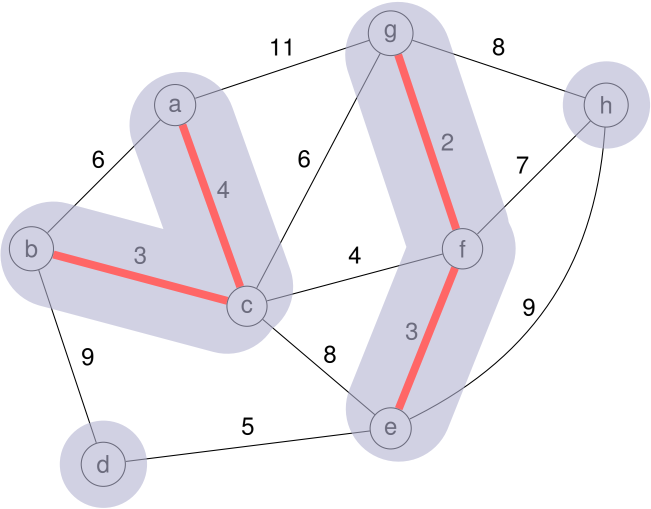
# Complete Run of Kruskal's Algorithm



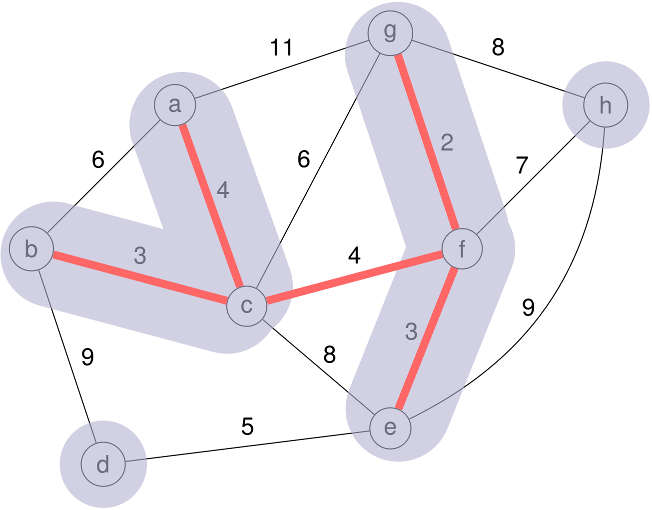
# Complete Run of Kruskal's Algorithm



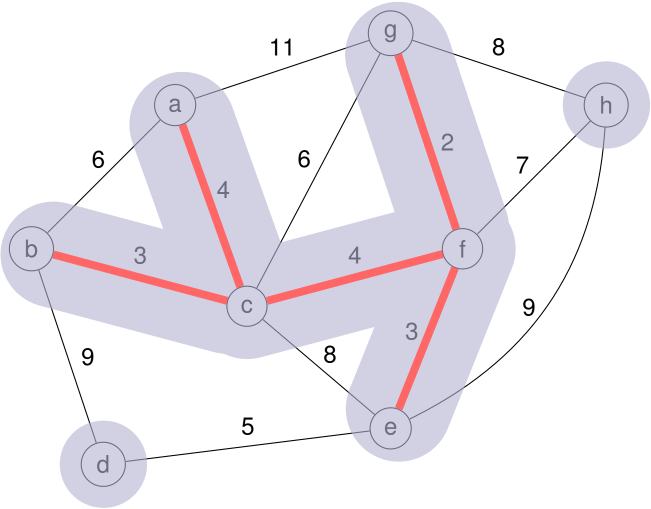
# Complete Run of Kruskal's Algorithm



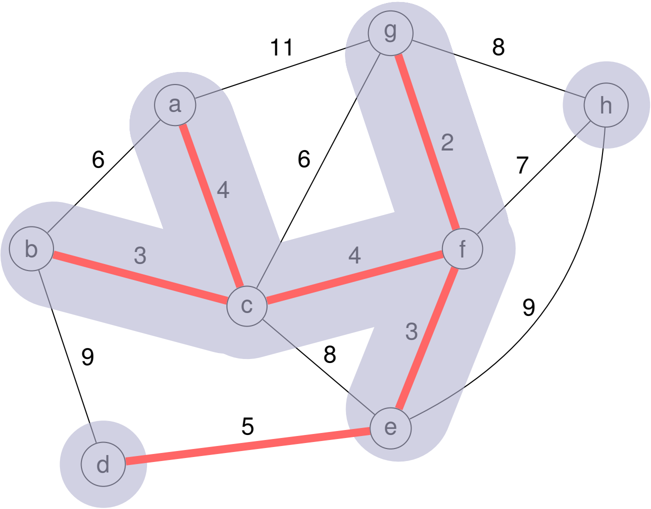
# Complete Run of Kruskal's Algorithm



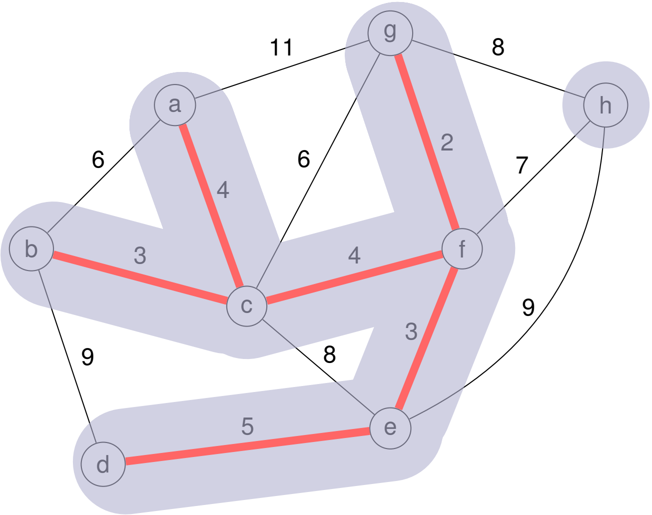
# Complete Run of Kruskal's Algorithm



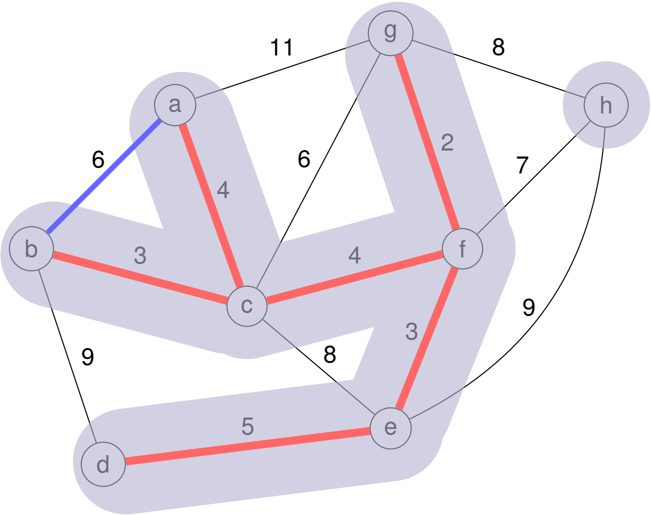
# Complete Run of Kruskal's Algorithm



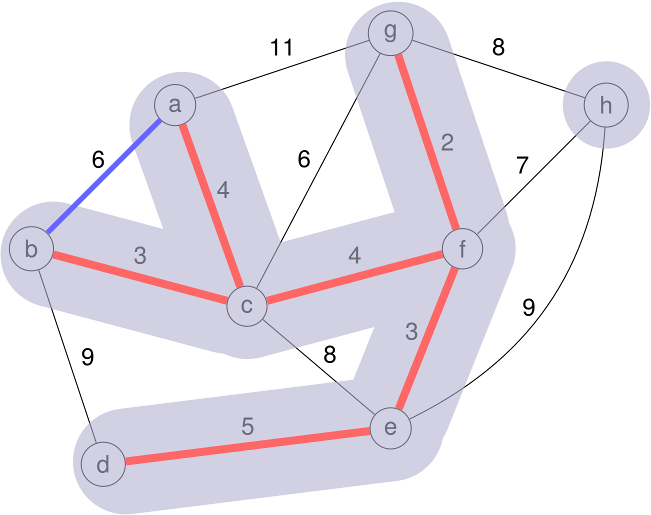
# Complete Run of Kruskal's Algorithm



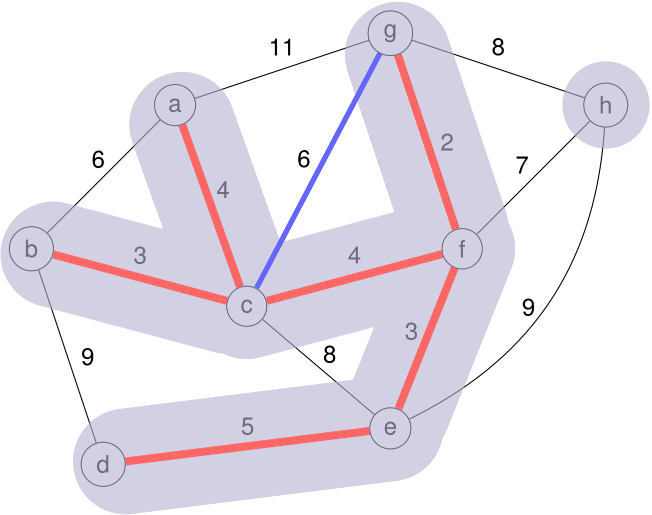
# Complete Run of Kruskal's Algorithm



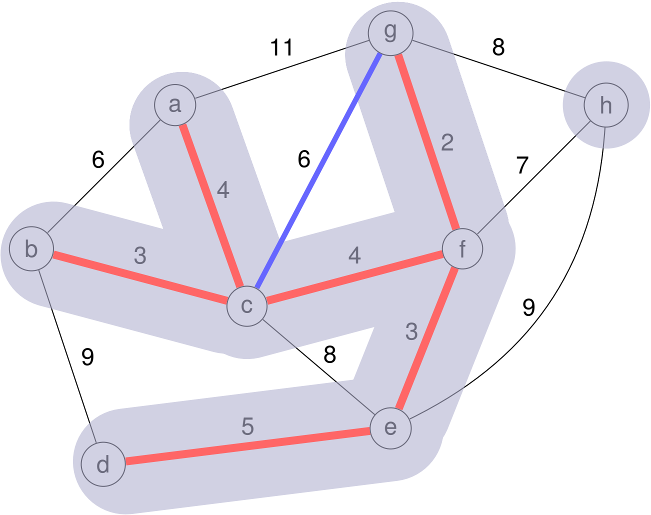
# Complete Run of Kruskal's Algorithm



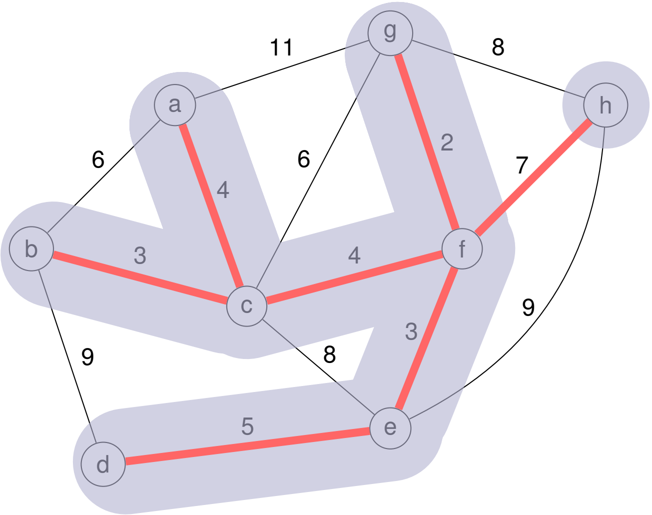
# Complete Run of Kruskal's Algorithm



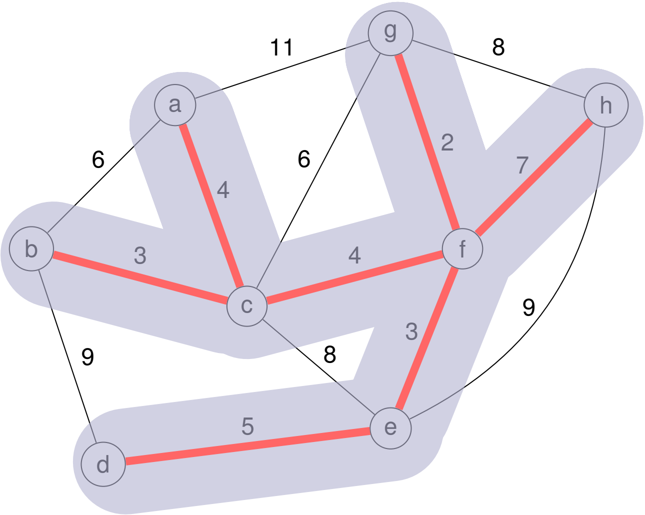
# Complete Run of Kruskal's Algorithm



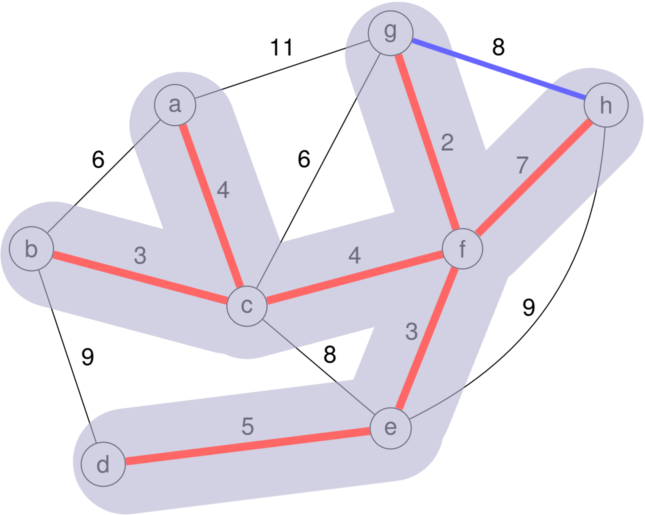
# Complete Run of Kruskal's Algorithm



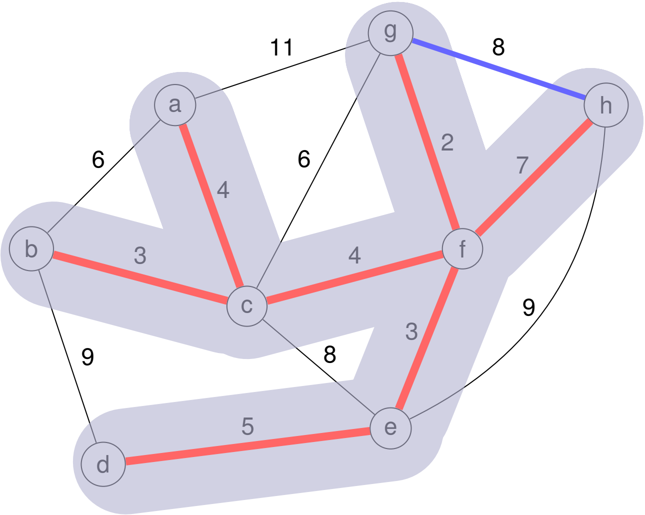
# Complete Run of Kruskal's Algorithm



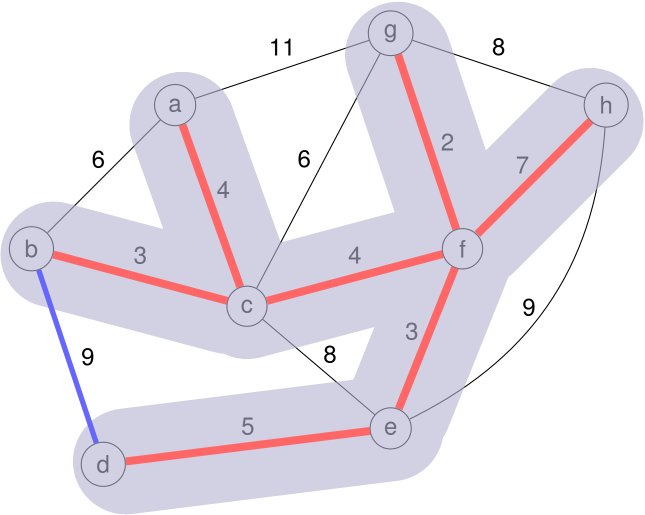
# Complete Run of Kruskal's Algorithm



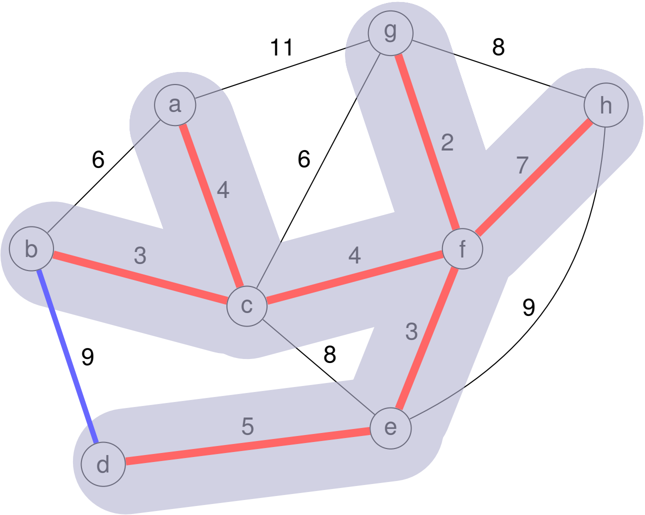
# Complete Run of Kruskal's Algorithm



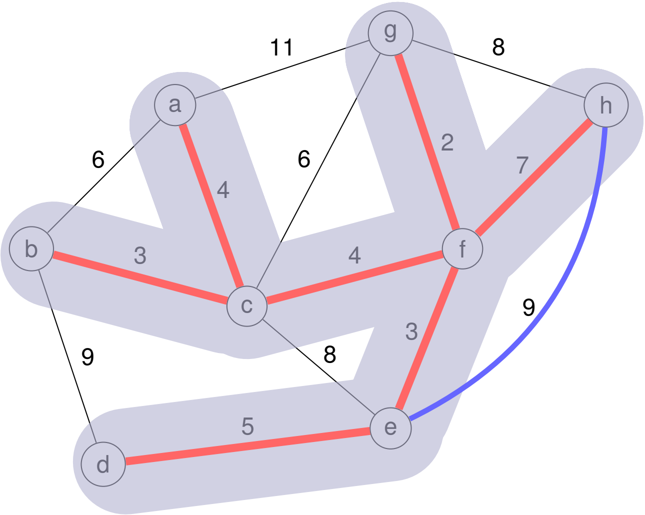
# Complete Run of Kruskal's Algorithm



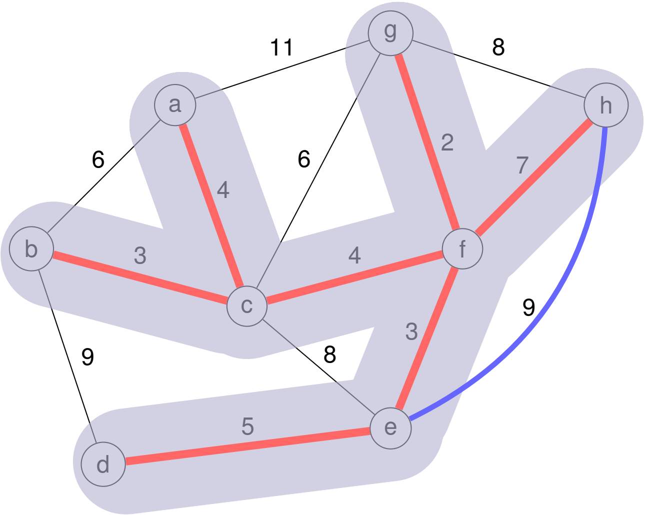
# Complete Run of Kruskal's Algorithm



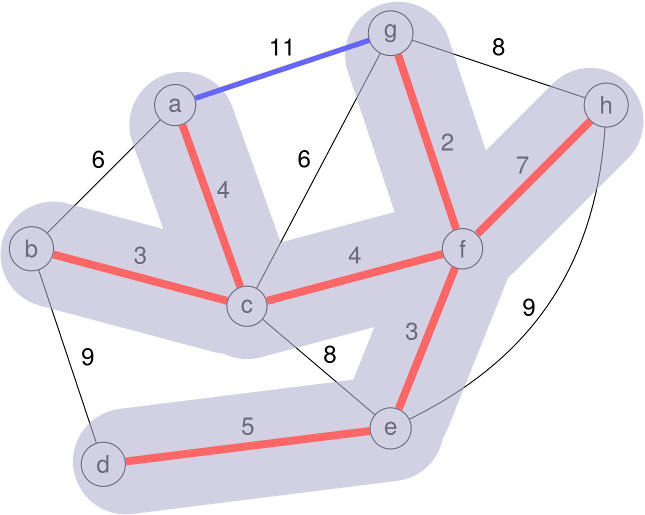
# Complete Run of Kruskal's Algorithm



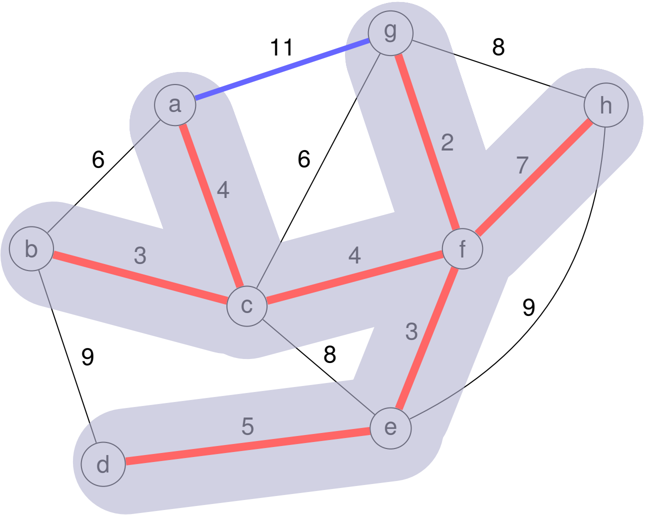
# Complete Run of Kruskal's Algorithm



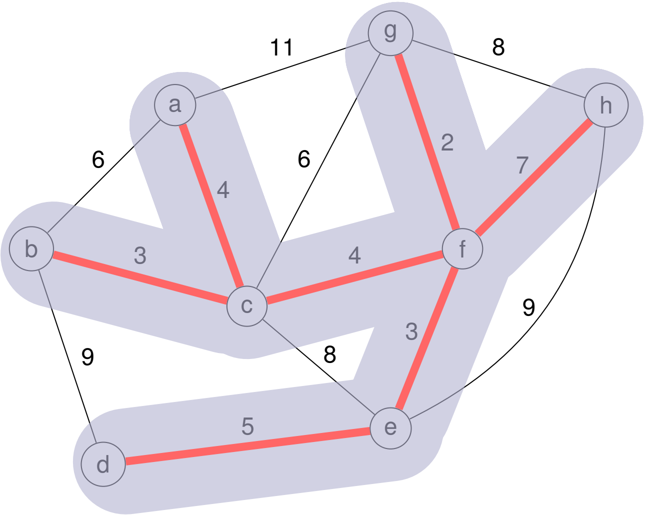
# Complete Run of Kruskal's Algorithm



# Complete Run of Kruskal's Algorithm



# Complete Run of Kruskal's Algorithm



# Details of Kruskal’s Algorithm

```
def kruskal(G)
    Apply Kruskal's algorithm to graph G
    Return set of edges that form a MST

A = Set() # Set of edges of MST
D = DisjointSet()
for v in G.vertices():
   D.makeSet(v)
E = G.edges()
E.sort(key=weight, direction=ascending)

for edge in E:
    startSet = D.findSet(edge.start)
    endSet = D.findSet(edge.end)
    if startSet != endSet:
       A.append(edge)
       D.union(startSet,endSet)
return A
```

# Details of Kruskal’s Algorithm

```
def kruskal(G)
    Apply Kruskal’s algorithm to graph G
    Return set of edges that form a MST

A = Set() # Set of edges of MST
D = DisjointSet()
for v in G.vertices():
   D.makeSet(v)
E = G.edges()
E.sort(key=weight, direction=ascending)

for edge in E:
   startSet = D.findSet(edge.start)
   endSet = D.findSet(edge.end)
   if startSet != endSet:
      A.append(edge)
      D.union(startSet,endSet)
return A
```

**Time Complexity**

# Details of Kruskal’s Algorithm

```
def kruskal(G)
    Apply Kruskal’s algorithm to graph G
    Return set of edges that form a MST

A = Set() # Set of edges of MST
D = DisjointSet()
for v in G.vertices():
   D.makeSet(v)
E = G.edges()
E.sort(key=weight, direction=ascending)

for edge in E:
   startSet = D.findSet(edge.start)
   endSet = D.findSet(edge.end)
   if startSet != endSet:
      A.append(edge)
      D.union(startSet,endSet)
return A
```

**Time Complexity**

# Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:       A.append(edge)
16:       D.union(startSet,endSet)
17: return A
```

**Time Complexity**

- Initialisation (l. 4-9): $\mathcal{O}(V + E \log E)$

## Details of Kruskal’s Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal’s algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:       A.append(edge)
16:       D.union(startSet,endSet)
17: return A
```

**Time Complexity**

- Initialisation (l. 4-9): $\mathcal{O}(V + E \log E)$

# Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:     startSet = D.findSet(edge.start)
13:     endSet = D.findSet(edge.end)
14:     if startSet != endSet:
15:        A.append(edge)
16:        D.union(startSet,endSet)
17: return A
```

**Time Complexity**

- Initialisation (l. 4-9): $\mathcal{O}(V + E \log E)$
- Main Loop (l. 11-16): $\mathcal{O}(E \cdot \alpha(n))$

# Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:       A.append(edge)
16:       D.union(startSet,endSet)
17: return A
```

**Time Complexity**

- Initialisation (l. 4-9): $\mathcal{O}(V + E \log E)$
- Main Loop (l. 11-16): $\mathcal{O}(E \cdot \alpha(n))$

$\Rightarrow$ Overall: $\mathcal{O}(E \log E) = \mathcal{O}(E \log V)$

# Details of Kruskal’s Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:       A.append(edge)
16:       D.union(startSet,endSet)
17: return A
```

**Time Complexity**

- Initialisation (l. 4-9): $\mathcal{O}(V + E \log E)$
- Main Loop (l. 11-16): $\mathcal{O}(E \cdot \alpha(n))$

$\Rightarrow$ Overall: $\mathcal{O}(E \log E) = \mathcal{O}(E \log V)$

If edges are already sorted, runtime becomes $O(E \cdot \alpha(n))$!

# Details of Kruskal’s Algorithm

```
def kruskal(G)
    Apply Kruskal’s algorithm to graph G
    Return set of edges that form a MST

A = Set() # Set of edges of MST
D = DisjointSet()
for v in G.vertices():
   D.makeSet(v)
E = G.edges()
E.sort(key=weight, direction=ascending)

for edge in E:
   startSet = D.findSet(edge.start)
   endSet = D.findSet(edge.end)
   if startSet != endSet:
      A.append(edge)
      D.union(startSet,endSet)
return A
```

**Correctness**

# Details of Kruskal's Algorithm

```
def kruskal(G)
    Apply Kruskal's algorithm to graph G
    Return set of edges that form a MST

A = Set() # Set of edges of MST
D = DisjointSet()
for v in G.vertices():
   D.makeSet(v)
E = G.edges()
E.sort(key=weight, direction=ascending)

for edge in E:
   startSet = D.findSet(edge.start)
   endSet = D.findSet(edge.end)
   if startSet != endSet:
      A.append(edge)
      D.union(startSet,endSet)
return A
```

**Correctness**

- Consider the cut of all connected components (disjoint sets)

# Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:       A.append(edge)
16:       D.union(startSet,endSet)
17: return A
```

**Correctness**

- Consider the cut of all connected components (disjoint sets)
- L. 14 ensures that we extend $A$ by an edge that goes across the cut

# Details of Kruskal's Algorithm

```
0: def kruskal(G)
1:     Apply Kruskal's algorithm to graph G
2:     Return set of edges that form a MST
3:
4: A = Set() # Set of edges of MST
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10:
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:       A.append(edge)
16:       D.union(startSet,endSet)
17: return A
```

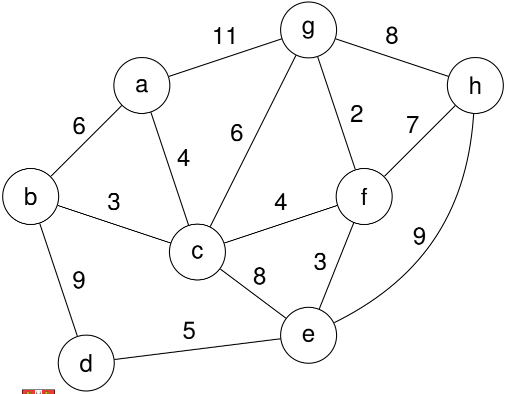
**Correctness**

- Consider the cut of all connected components (disjoint sets)
- L. 14 ensures that we extend $A$ by an edge that goes across the cut
- This edge is also the lightest edge crossing the cut (otherwise, we would have included a lighter edge before)

# Prim's Algorithm

Basic Strategy

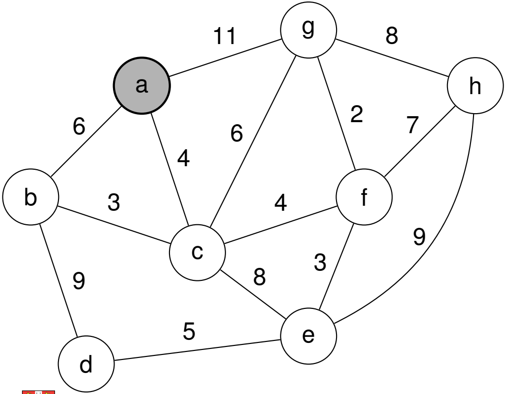
- Start growing a tree from a designated root vertex



## Prim's Algorithm

Basic Strategy

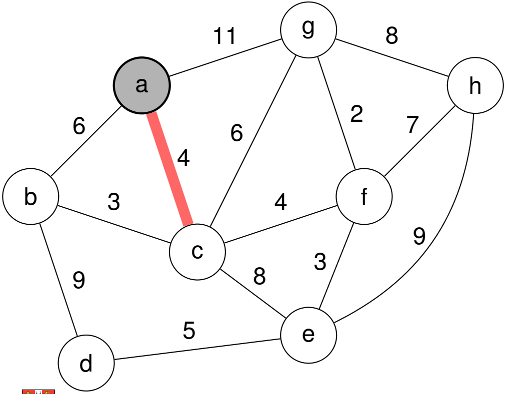
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle



# Prim's Algorithm

Basic Strategy

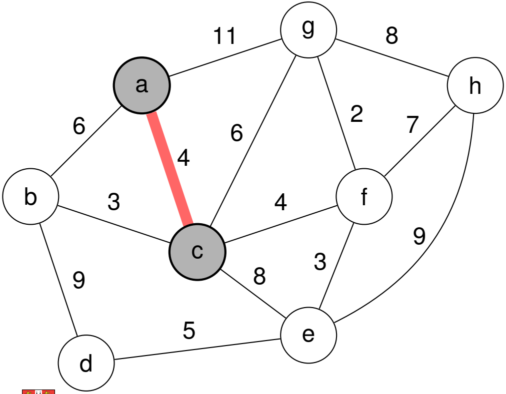
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle



# Prim's Algorithm

Basic Strategy

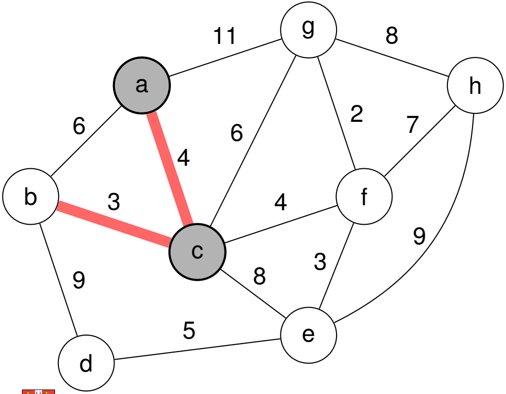
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle



# Prim's Algorithm

Basic Strategy

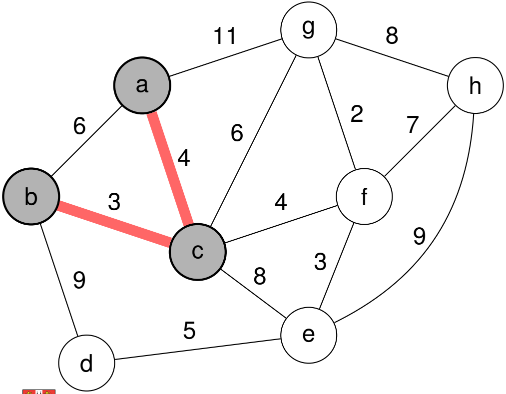
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle



## Prim's Algorithm

Basic Strategy

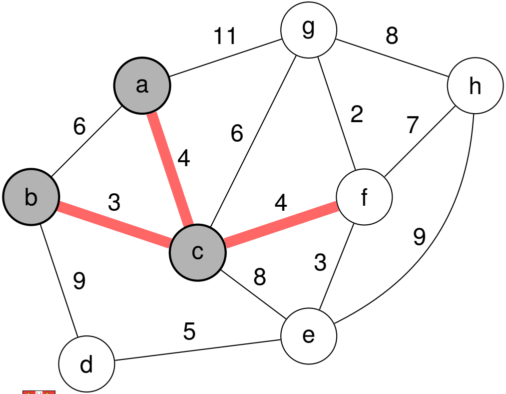
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



# Prim's Algorithm

Basic Strategy

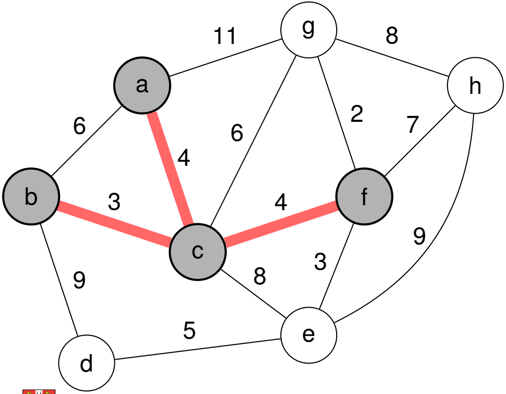
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



## Prim's Algorithm

Basic Strategy

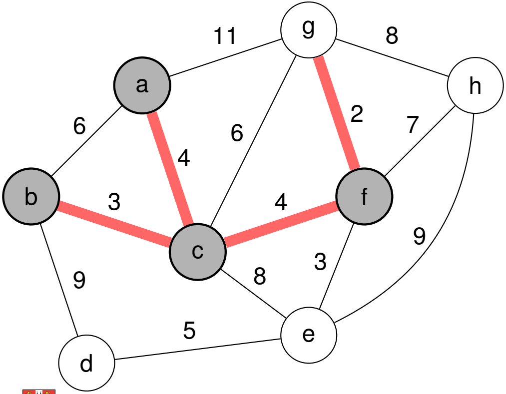
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



## Prim's Algorithm

Basic Strategy

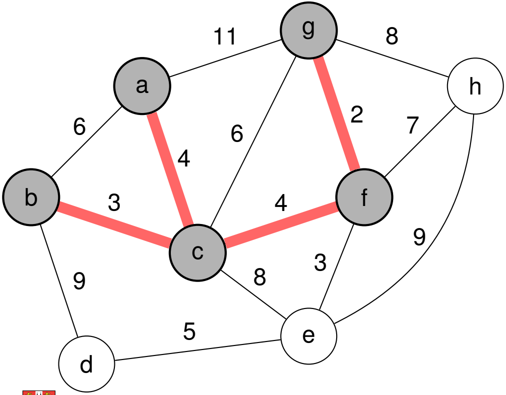
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle



## Prim's Algorithm

Basic Strategy

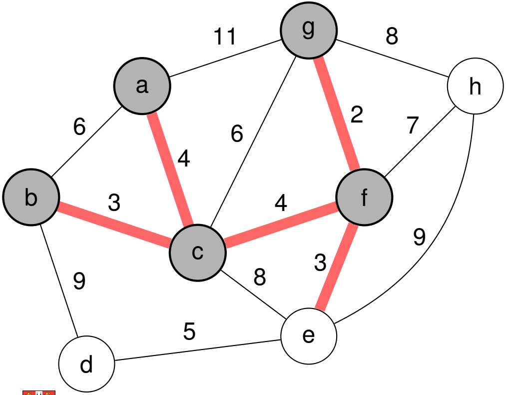
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle



## Prim's Algorithm

Basic Strategy

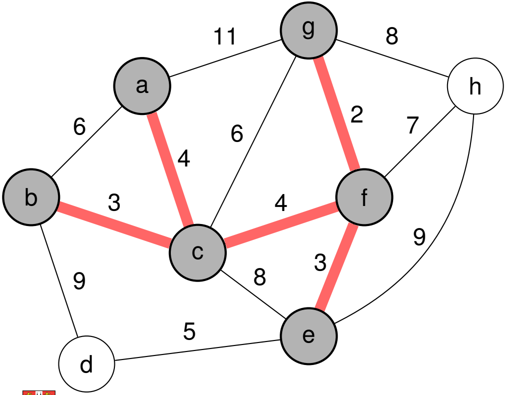
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle



# Prim's Algorithm

Basic Strategy

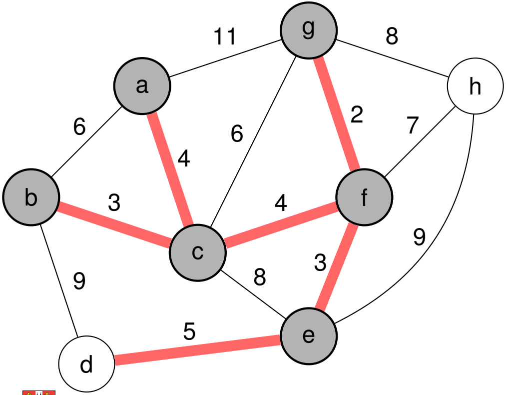
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle



## Prim's Algorithm

Basic Strategy

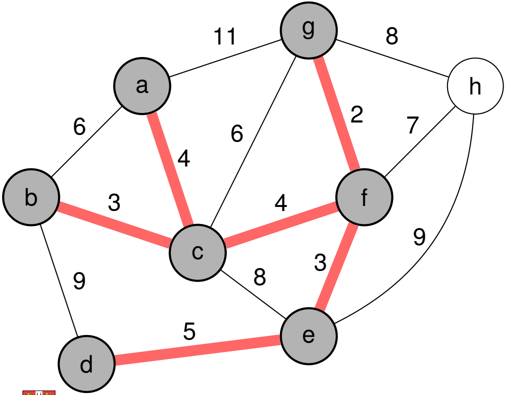
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



## Prim's Algorithm

Basic Strategy

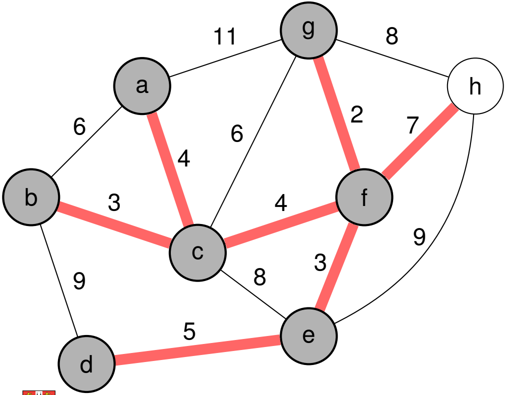
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



## Prim's Algorithm

Basic Strategy

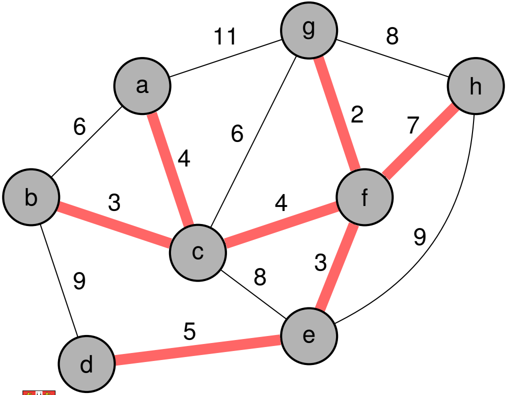
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle



# Prim's Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle

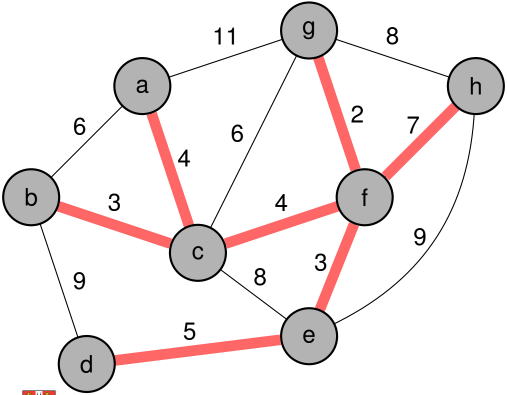


## Prim's Algorithm

**Basic Strategy**

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle

Implementation will be based on vertices!

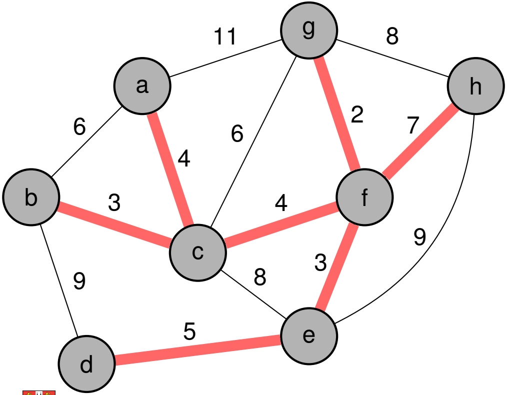


## Prim's Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle

Assign every vertex not in $A$ a key which is at all stages equal to the smallest weight of an edge connecting to $A$



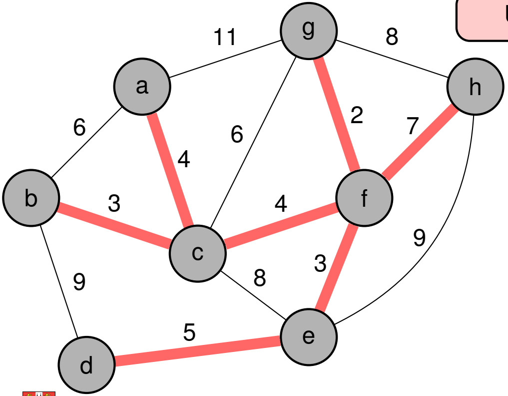
# Prim's Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle

Assign every vertex not in $A$ a key which is at all stages equal to the smallest weight of an edge connecting to $A$

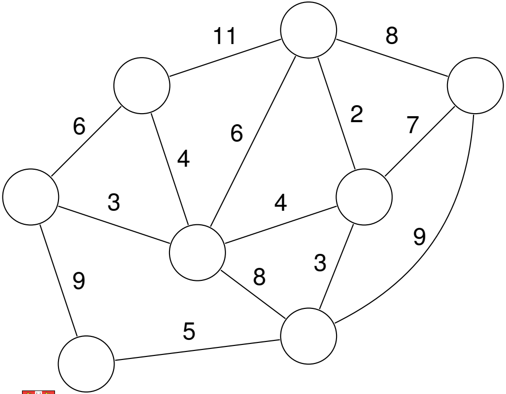
Use a Priority Queue!



# Prim's Algorithm

**Implementation**

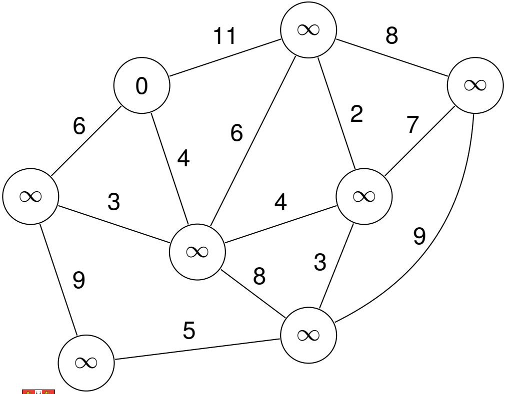
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

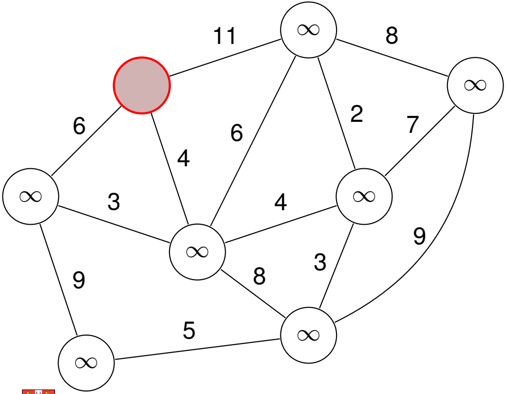
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

**Implementation**

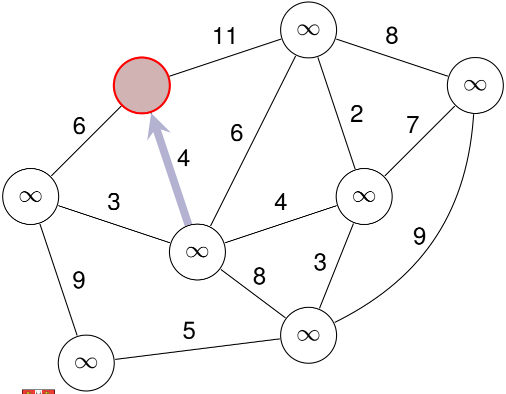
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

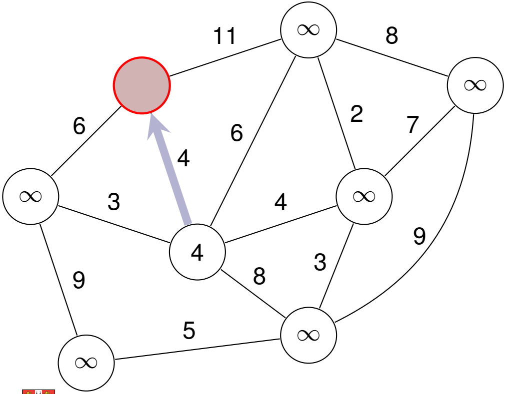
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

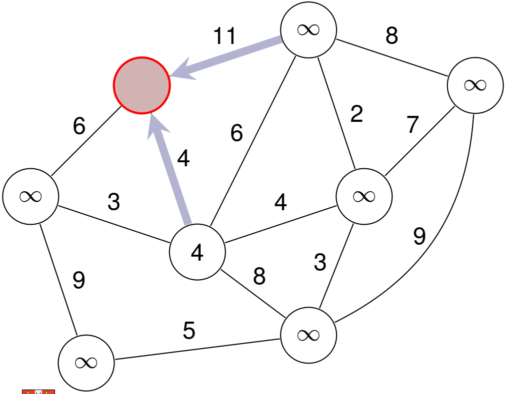
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

Implementation

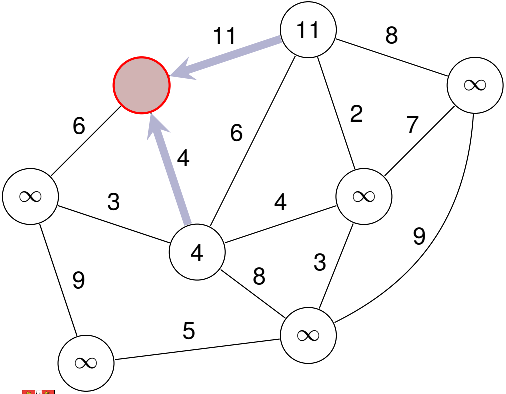
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

**Implementation**

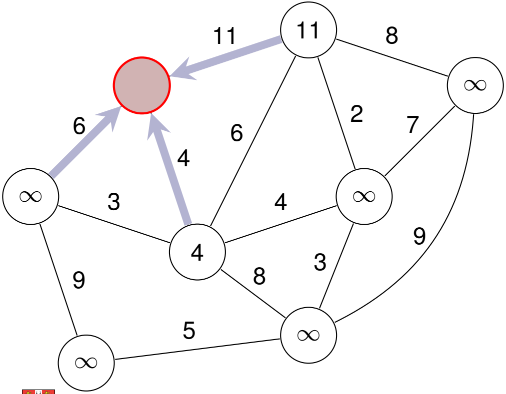
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

**Implementation**

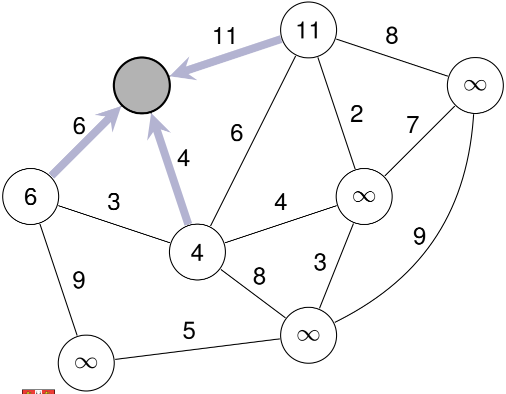
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim’s Algorithm

**Implementation**

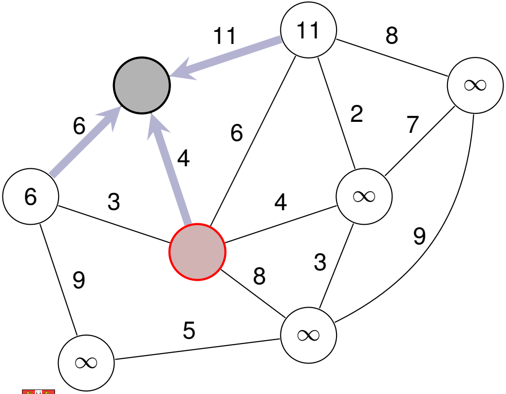
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

**Implementation**

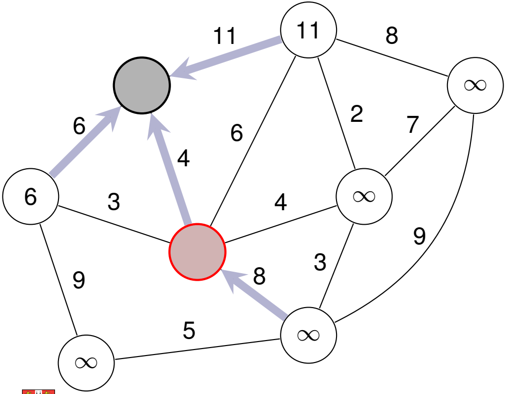
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

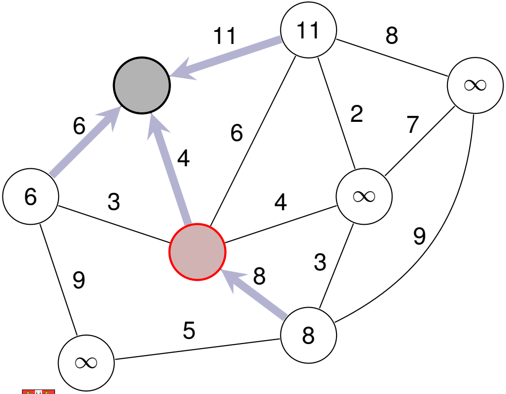
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

**Implementation**

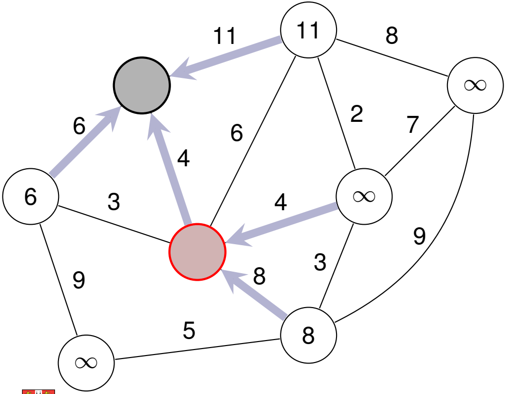
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

Implementation

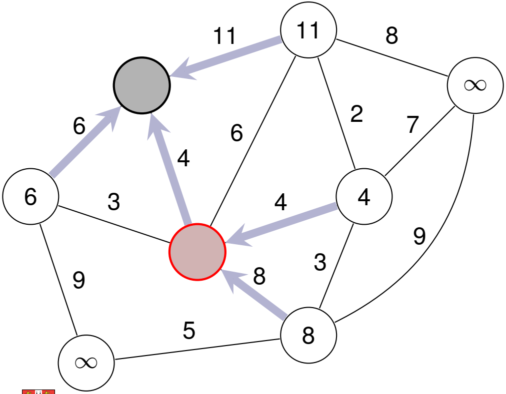
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

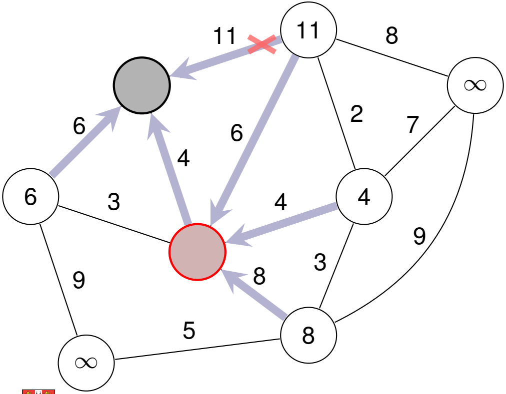
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

Implementation

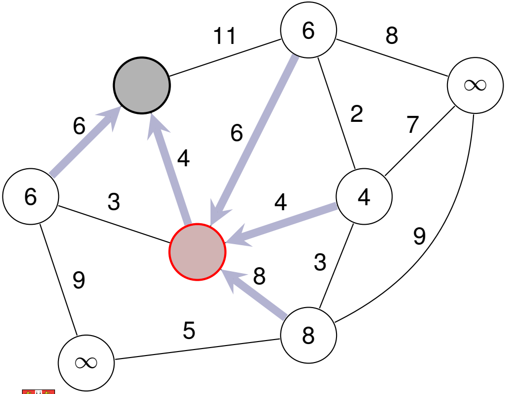
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

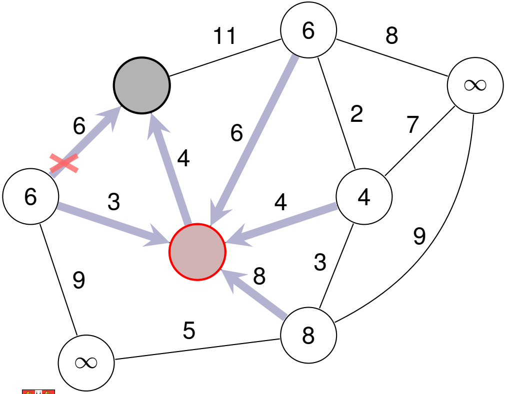
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

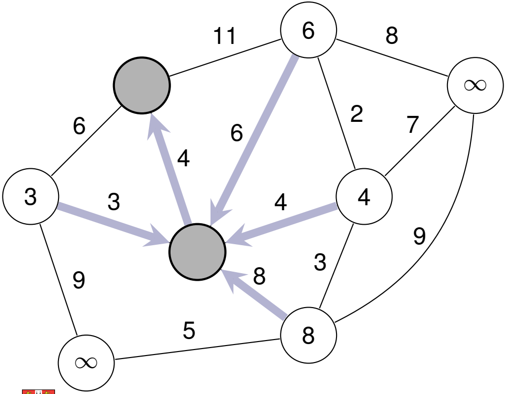
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

Implementation

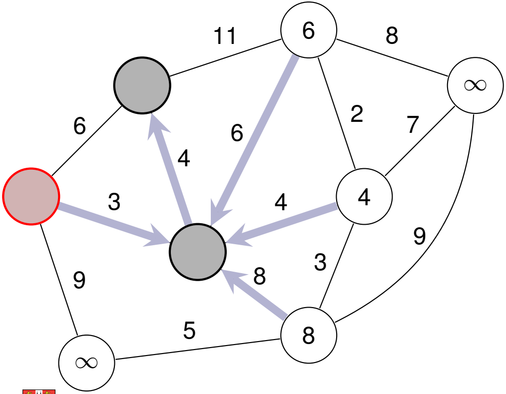
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

**Implementation**

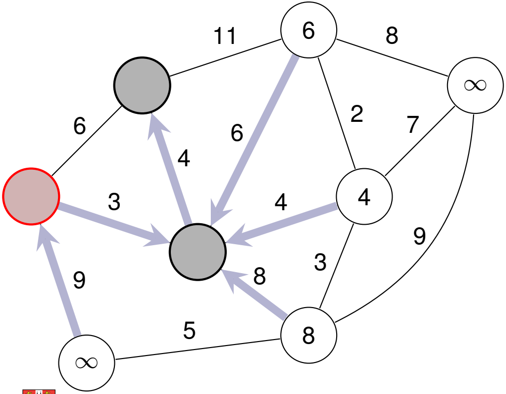
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

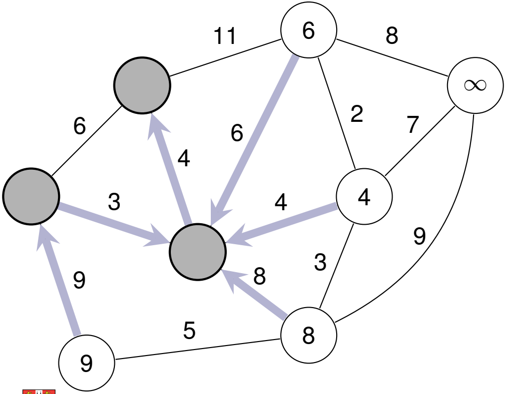
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

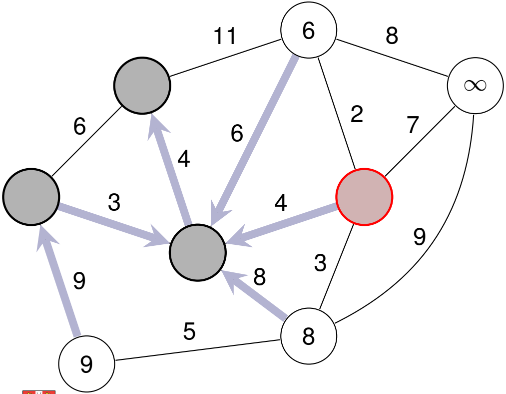
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

Implementation

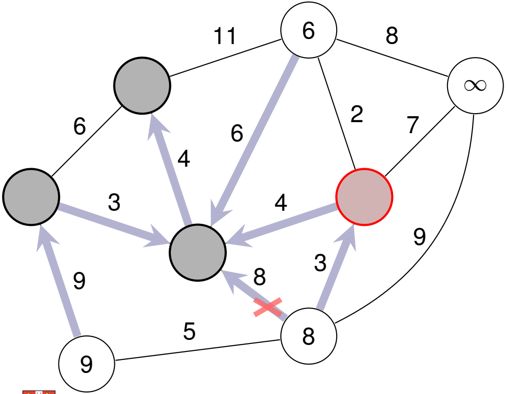
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

**Implementation**

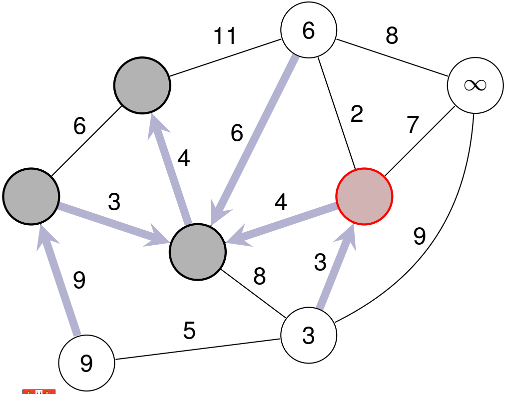
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

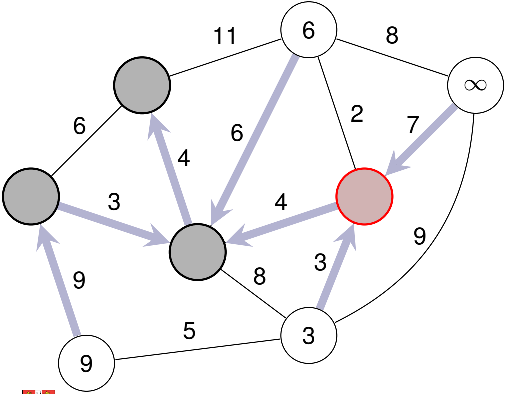
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

**Implementation**

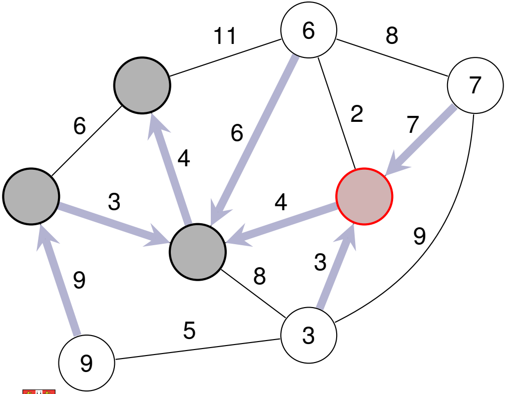
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

**Implementation**

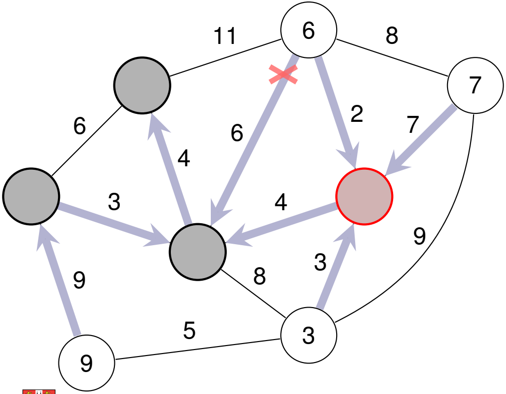
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

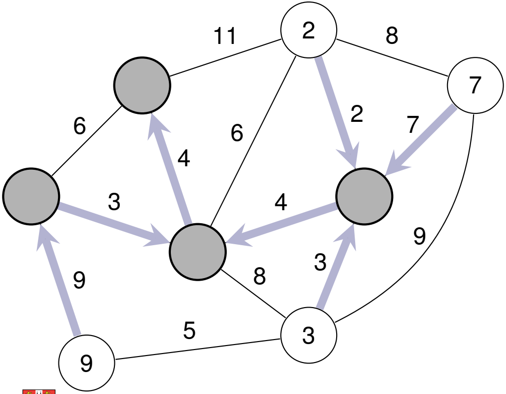
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

**Implementation**

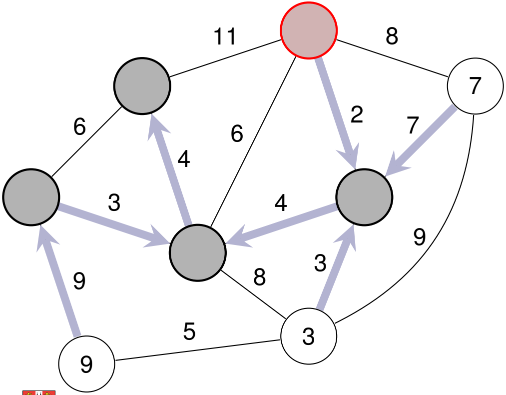
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

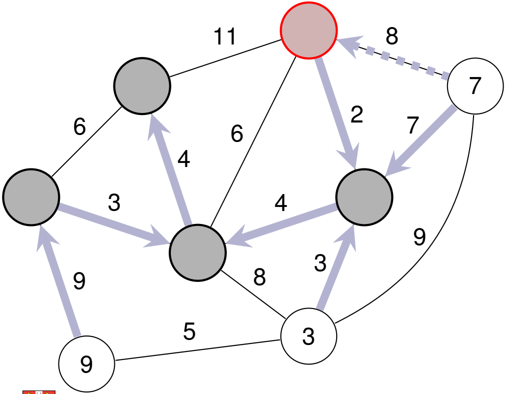
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

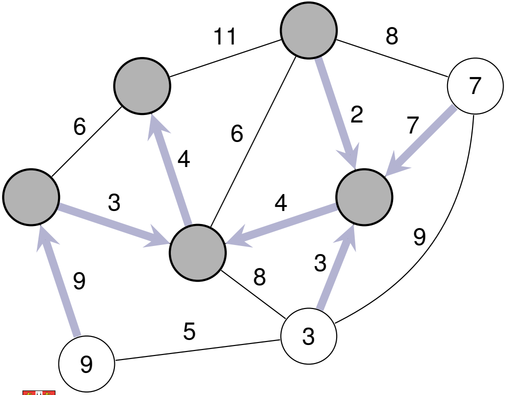
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

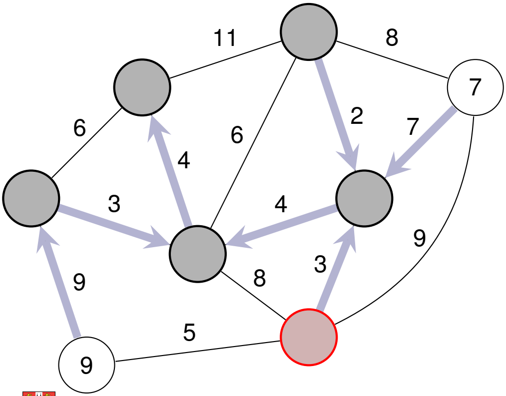
Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



**Implementation**

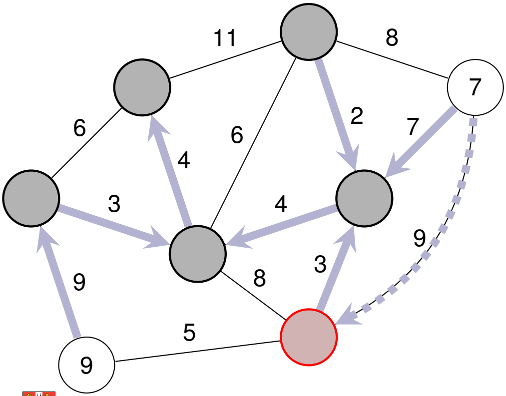
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

**Implementation**

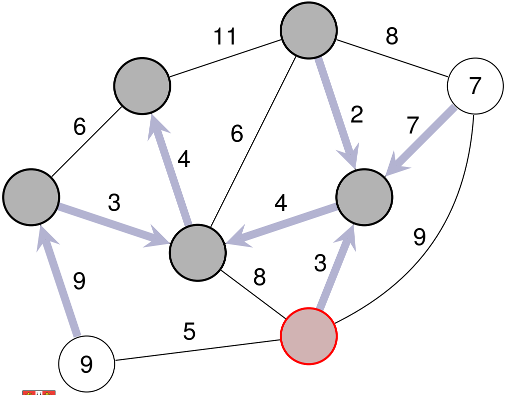
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

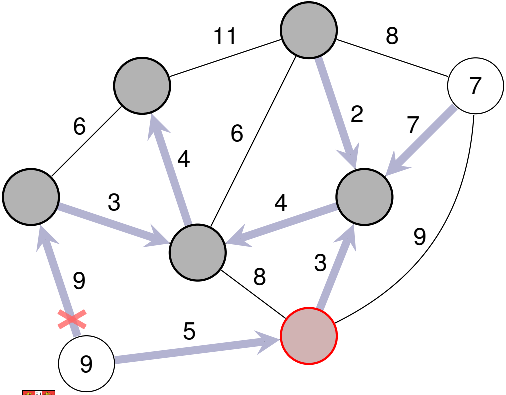
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



## Prim's Algorithm

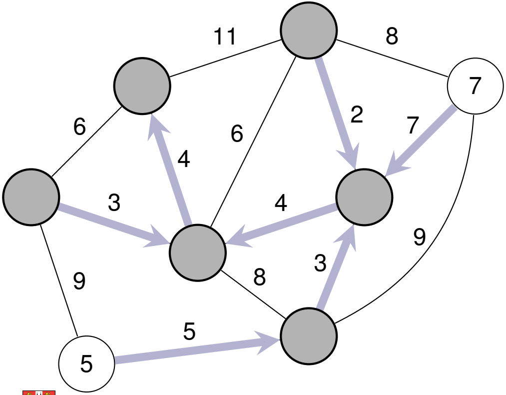
**Implementation**

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



**Implementation**

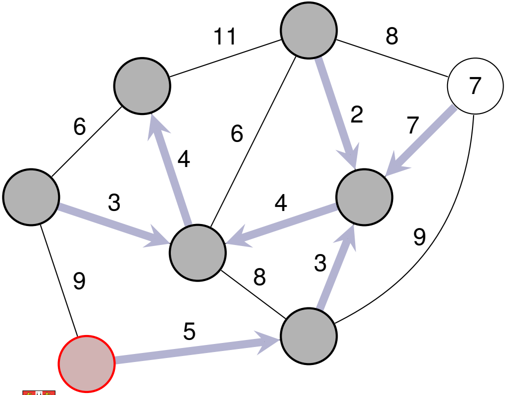
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

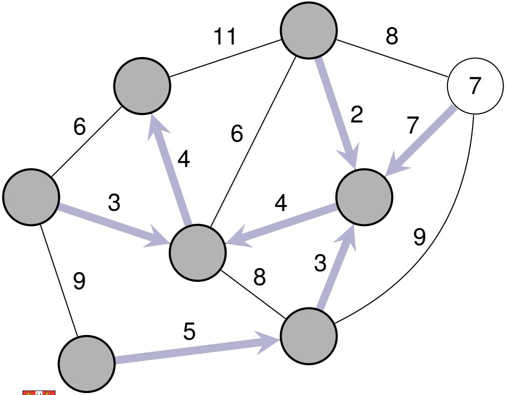
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

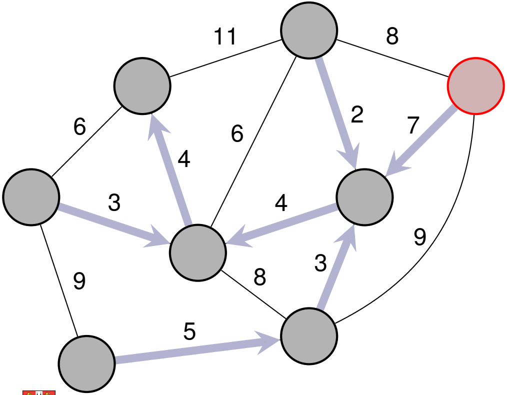
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

**Implementation**

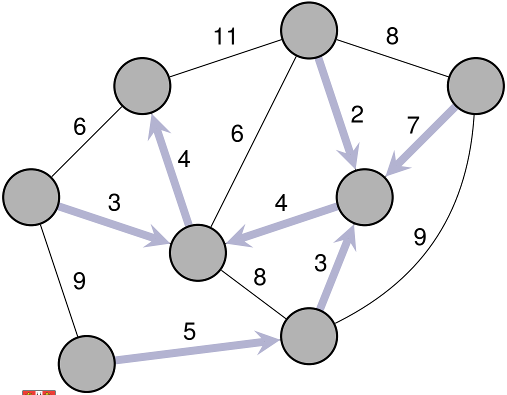
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

## Implementation

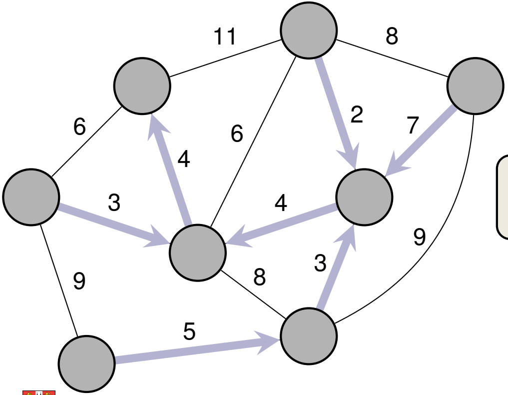
- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$



# Prim's Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

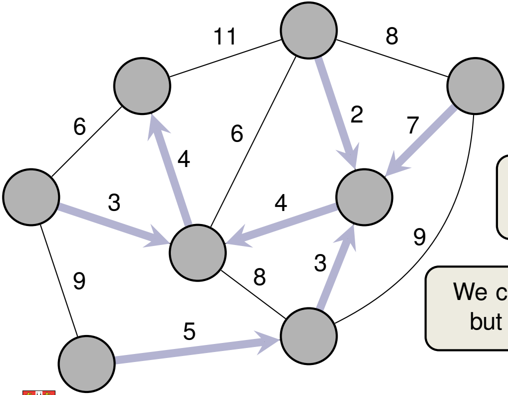



Final MST is given (implicitly) by the pointers!

# Prim's Algorithm

**Implementation**

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$




Final MST is given (implicitly) by the pointers!

We computed same MST as Kruskal, but in a completely different order!

# Details of Prim's Algorithm

```
def prim(G,r)
    Apply Prim's Algorithm to graph G and root r
    Return result implicitly by modifying G:
    MST induced by the .predecessor fields

Q = MinPriorityQueue()
for v in G.vertices():
    v.predecessor = None
    if v == r:
        v.key = 0
    else:
        v.key = Infinity
    Q.insert(v)

while not Q.isEmpty():
    u = Q.extractMin()
    for v in u.adjacent():
        w = G.weightOfEdge(u,v)
        if Q.hasItem(v) and w < v.key:
            v.predecessor = u
            Q.decreaseKey(item=v, newKey=w)
```

# Details of Prim's Algorithm

```
def prim(G,r)
    Apply Prim's Algorithm to graph G and root r
    Return result implicitly by modifying G:
    MST induced by the .predecessor fields

Q = MinPriorityQueue()
for v in G.vertices():
    v.predecessor = None
    if v == r:
        v.key = 0
    else:
        v.key = Infinity
    Q.insert(v)

while not Q.isEmpty():
    u = Q.extractMin()
    for v in u.adjacent():
        w = G.weightOfEdge(u,v)
        if Q.hasItem(v) and w < v.key:
           v.predecessor = u
           Q.decreaseKey(item=v, newKey=w)
```

## Time Complexity

# Details of Prim's Algorithm

```
def prim(G,r)
    Apply Prim's Algorithm to graph G and root r
    Return result implicitly by modifying G:
    MST induced by the .predecessor fields

Q = MinPriorityQueue()
for v in G.vertices():
    v.predecessor = None
    if v == r:
        v.key = 0
    else:
        v.key = Infinity
    Q.insert(v)

while not Q.isEmpty():
    u = Q.extractMin()
    for v in u.adjacent():
        w = G.weightOfEdge(u,v)
        if Q.hasItem(v) and w < v.key:
            v.predecessor = u
            Q.decreaseKey(item=v, newKey=w)
```

## Time Complexity

- Fibonacci Heaps:

# Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:            v.predecessor = u
20:            Q.decreaseKey(item=v, newKey=w)
```

## Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$,

# Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:            v.predecessor = u
20:            Q.decreaseKey(item=v, newKey=w)
```

## Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$,

# Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:             v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

## Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$,

# Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:             v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

## Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$,

# Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:             v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

## Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot 1)$

# Details of Prim's Algorithm

```
 0: def prim(G,r)
 1:     Apply Prim's Algorithm to graph G and root r
 2:     Return result implicitly by modifying G:
 3:     MST induced by the .predecessor fields
 4:
 5: Q = MinPriorityQueue()
 6: for v in G.vertices():
 7:     v.predecessor = None
 8:     if v == r:
 9:         v.key = 0
10:     else:
11:         v.key = Infinity
12:     Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:             v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

## Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot 1)$

  Amortized Cost

  Amortized Cost

# Details of Prim's Algorithm

```
0: def prim(G,r)
1:     Apply Prim's Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:
5: Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:     if v == r:
9:         v.key = 0
10:    else:
11:        v.key = Infinity
12:    Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:            v.predecessor = u
20:            Q.decreaseKey(item=v, newKey=w)
```

## Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot 1)$
  $\Rightarrow$ Overall: $\mathcal{O}(V \log V + E)$

# Details of Prim's Algorithm

```
0:  def prim(G,r)
1:      Apply Prim's Algorithm to graph G and root r
2:      Return result implicitly by modifying G:
3:      MST induced by the .predecessor fields
4:
5:  Q = MinPriorityQueue()
6:  for v in G.vertices():
7:      v.predecessor = None
8:      if v == r:
9:          v.key = 0
10:     else:
11:         v.key = Infinity
12:     Q.insert(v)
13:
14: while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:         w = G.weightOfEdge(u,v)
18:         if Q.hasItem(v) and w < v.key:
19:             v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

## Time Complexity

- Fibonacci Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot 1)$
  $\Rightarrow$ Overall: $\mathcal{O}(V \log V + E)$
- Binary/Binomial Heaps:
  Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot \log V)$
  $\Rightarrow$ Overall: $\mathcal{O}(V \log V + E \log V)$

## Summary (Kruskal and Prim)

Generic Idea

- Add safe edge to the current MST as long as possible
- Theorem: An edge is safe if it is the lightest of a cut respecting $A$

# Summary (Kruskal and Prim)

Generic Idea

- Add safe edge to the current MST as long as possible
- Theorem: An edge is safe if it is the lightest of a cut respecting $A$

Kruskal's Algorithm

- Gradually transforms a forest into a MST by merging trees
- invokes disjoint set data structure
- Runtime $\mathcal{O}(V + E \log V)$

# Summary (Kruskal and Prim)

**Generic Idea**

- Add safe edge to the current MST as long as possible
- Theorem: An edge is safe if it is the lightest of a cut respecting $A$

**Kruskal's Algorithm**

- Gradually transforms a forest into a MST by merging trees
- invokes disjoint set data structure
- Runtime $\mathcal{O}(V + E \log V)$

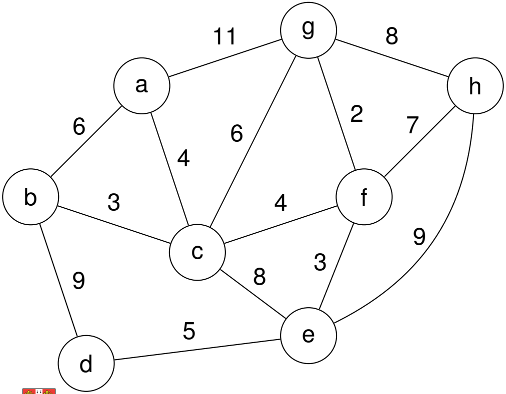
**Prim's Algorithm**

- Gradually extends a tree into a MST by adding incident edges
- invokes Fibonacci heaps (priority queue)
- Runtime $\mathcal{O}(V \log V + E)$

## Reverse-Delete Algorithm

Basic Idea

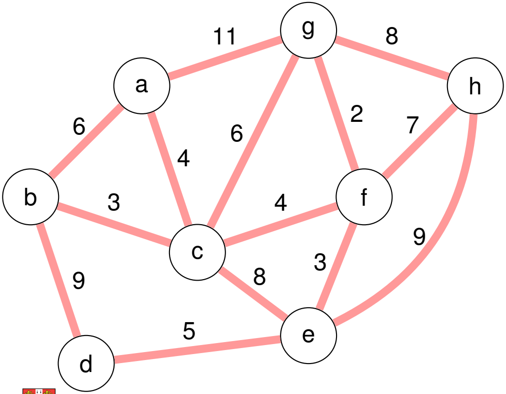
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

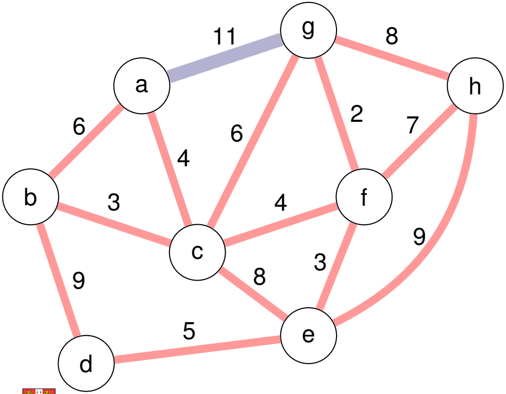
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

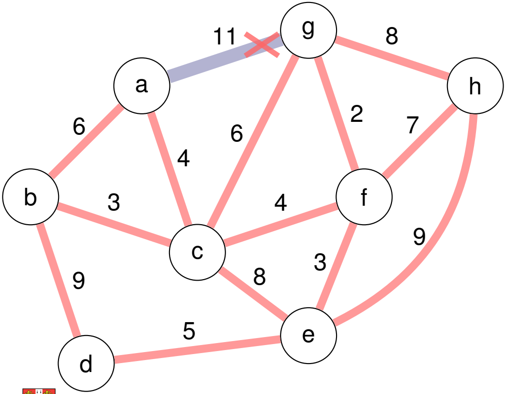
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

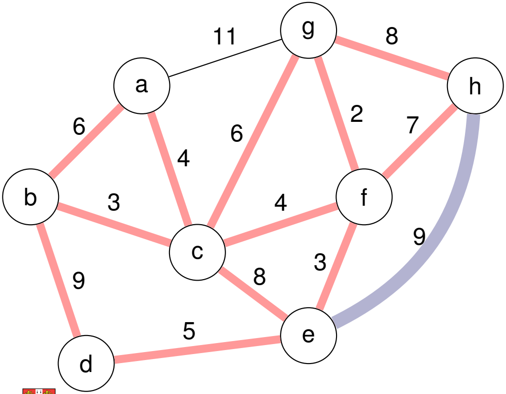
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

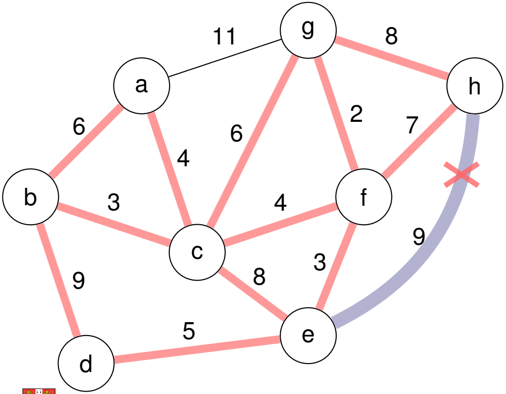
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



## Reverse-Delete Algorithm

Basic Idea

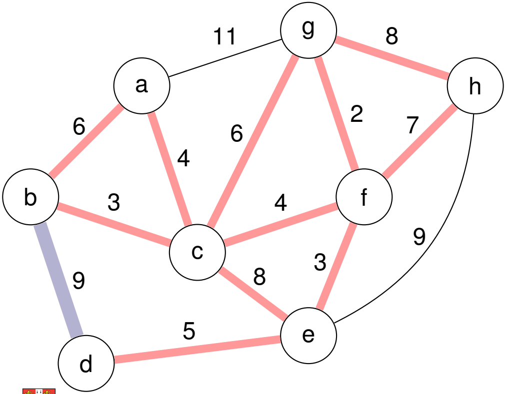
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

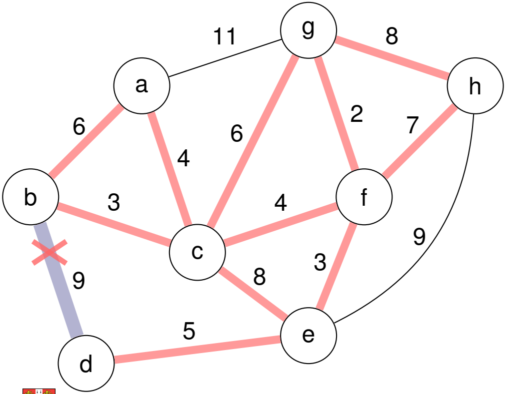
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

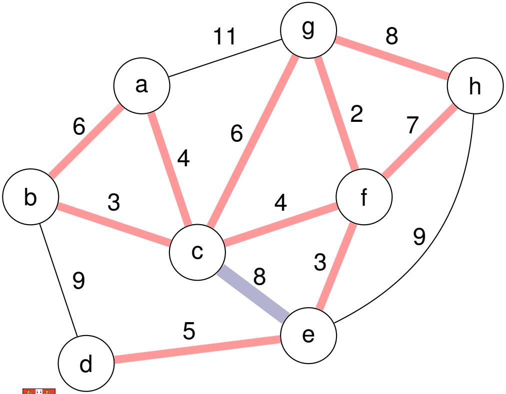
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

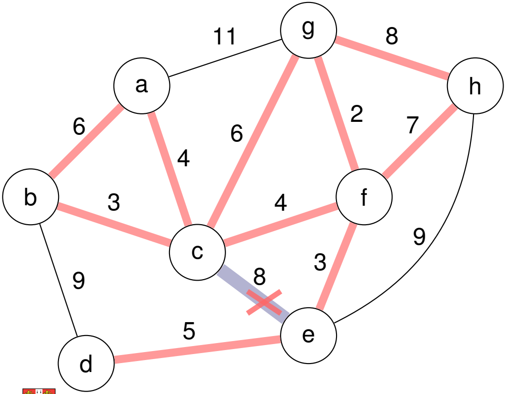
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

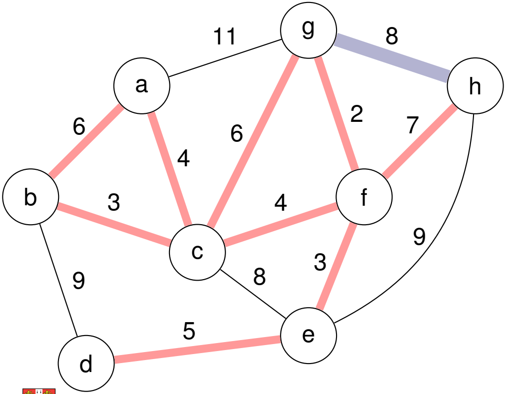
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

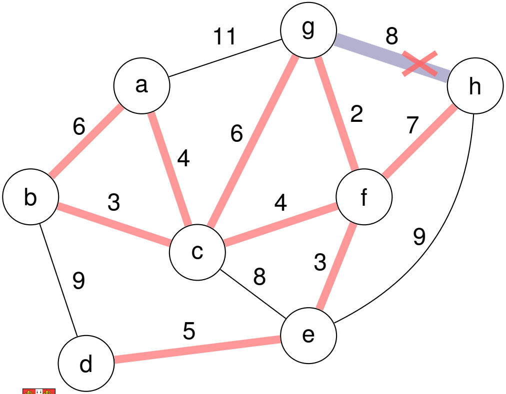
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

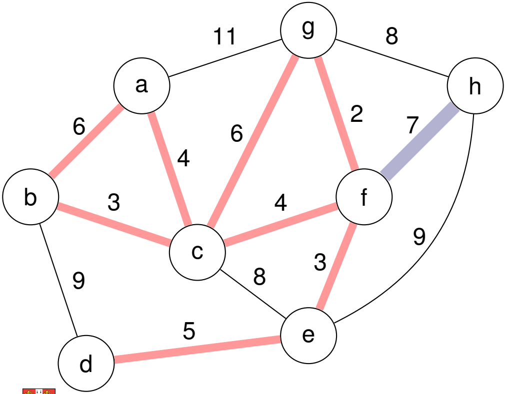
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



## Reverse-Delete Algorithm

Basic Idea

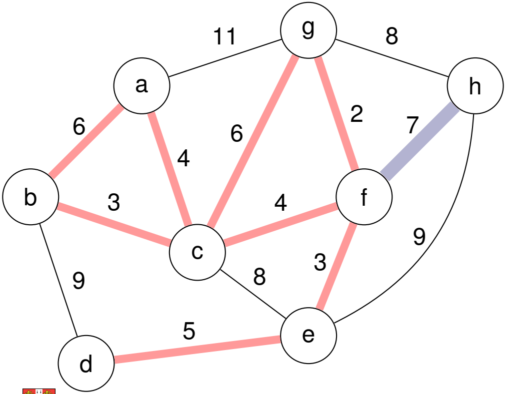
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

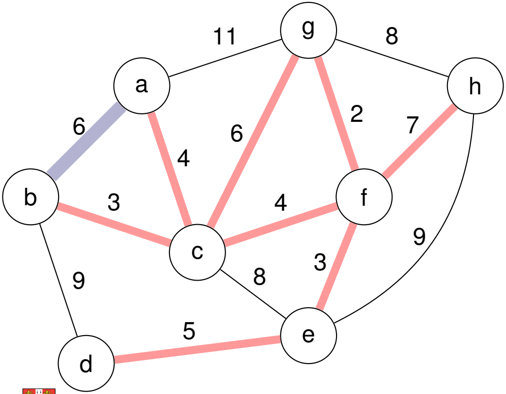
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



## Reverse-Delete Algorithm

Basic Idea

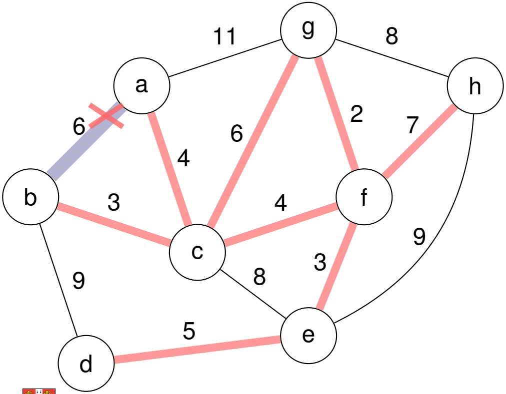
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

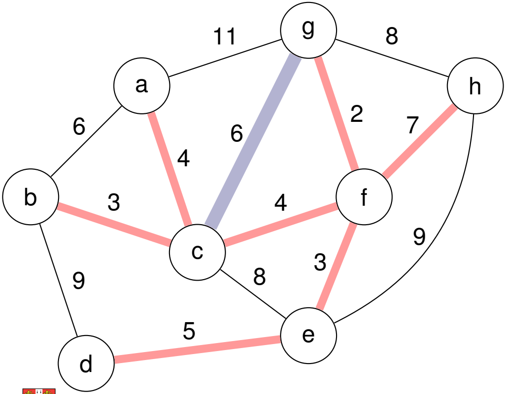
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



## Reverse-Delete Algorithm

Basic Idea

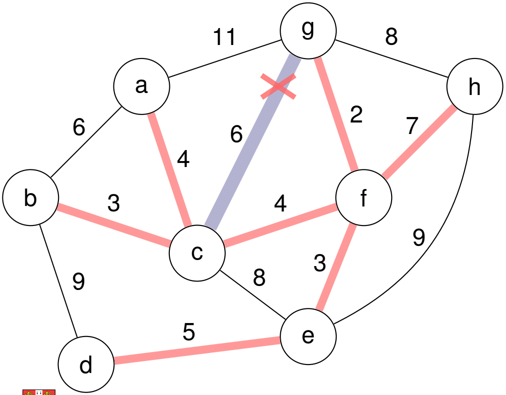
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

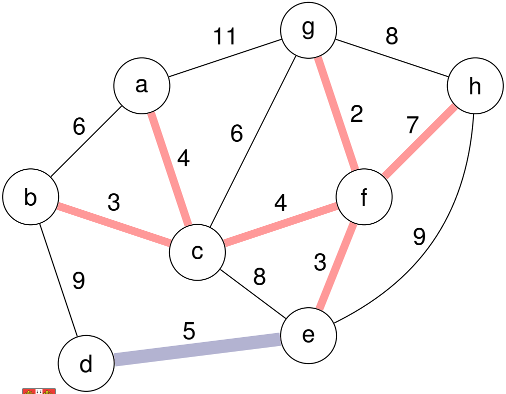
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

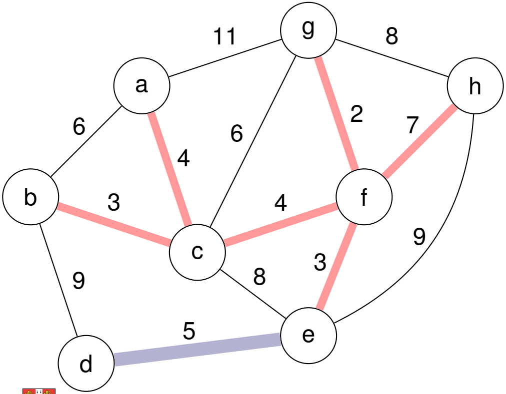
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

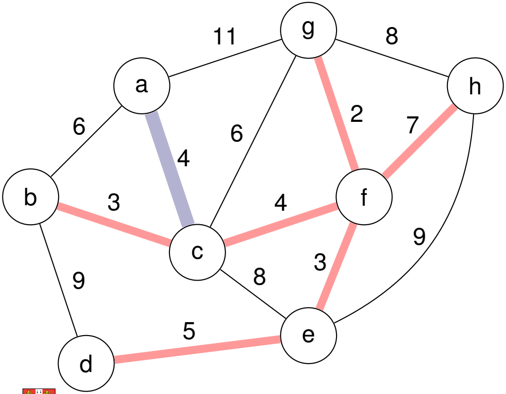
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

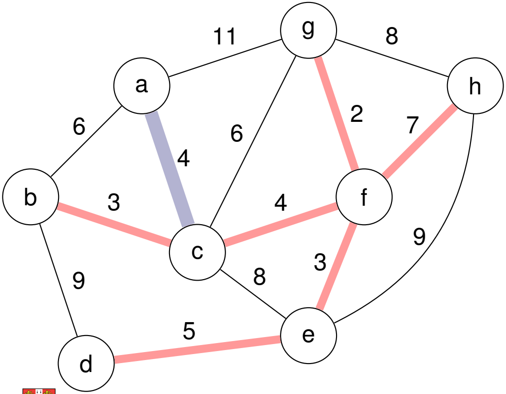
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

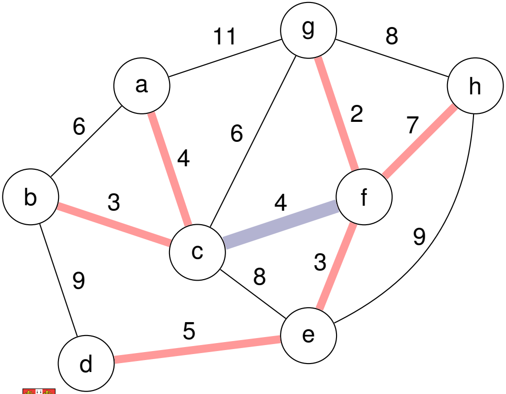
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

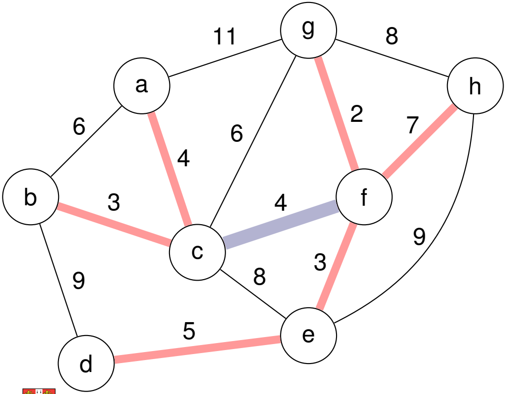
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

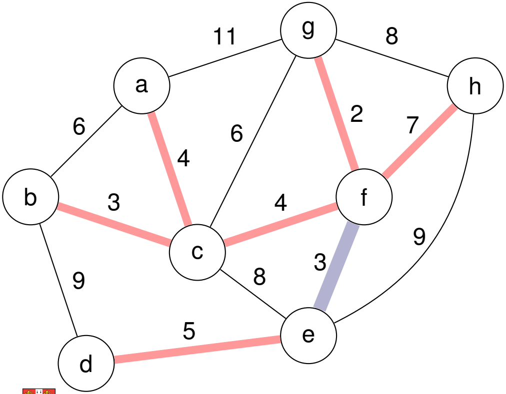
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



## Reverse-Delete Algorithm

Basic Idea

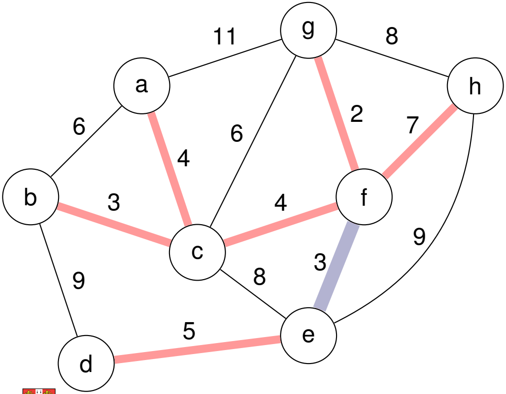
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

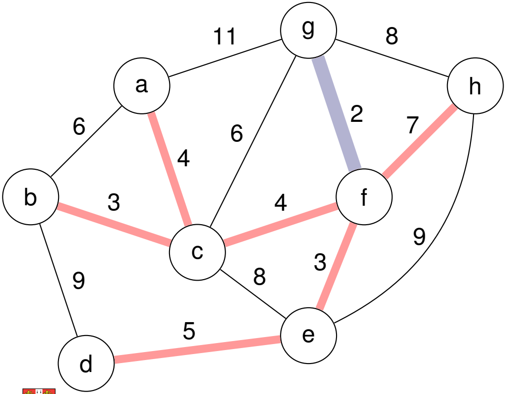
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

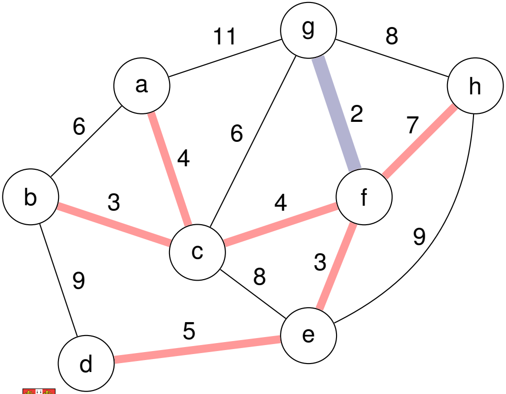
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



## Reverse-Delete Algorithm

Basic Idea

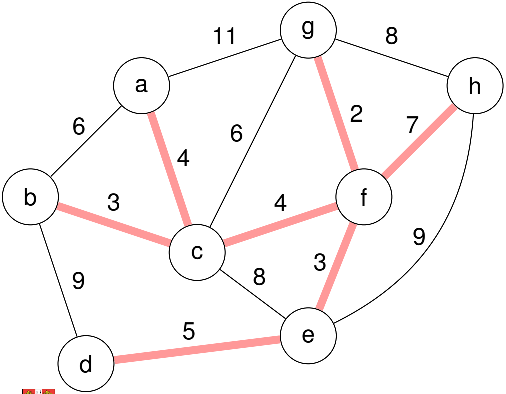
- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$



# Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

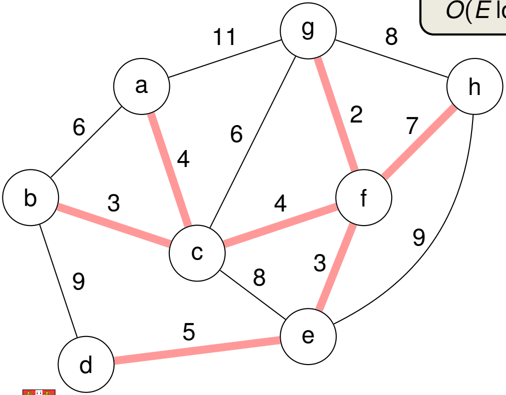


# Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]



# Current State-of-the-Art

Does a linear-time MST algorithm exist?

## Current State-of-the-Art

**Does a linear-time MST algorithm exist?**

Karger, Klein, Tarjan, JACM'1995

- randomised MST algorithm with expected runtime $O(E)$
- based on Boruvka's algorithm (from 1926)

# Current State-of-the-Art

**Does a linear-time MST algorithm exist?**

Karger, Klein, Tarjan, JACM'1995

- randomised MST algorithm with expected runtime $O(E)$
- based on Boruvka's algorithm (from 1926)

Chazelle, JACM'2000

- deterministic MST algorithm with runtime $O(E \cdot \alpha(n))$

# Current State-of-the-Art

**Does a linear-time MST algorithm exist?**

Karger, Klein, Tarjan, JACM'1995

- randomised MST algorithm with expected runtime $O(E)$
- based on Boruvka's algorithm (from 1926)

Chazelle, JACM'2000

- deterministic MST algorithm with runtime $O(E \cdot \alpha(n))$

Pettie, Ramachandran, JACM'2002

- deterministic MST algorithm with asymptotically optimal runtime
- however, the runtime itself is not known...