

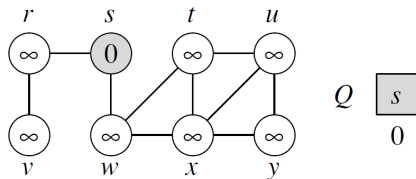
# Algorithms

Shortest paths

Emanuele Rodolà  
[rodola@di.uniroma1.it](mailto:rodola@di.uniroma1.it)

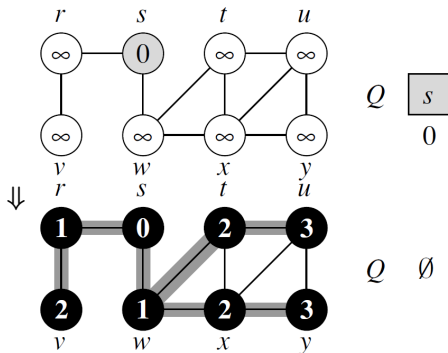


# Breadth-first search (BFS)



We start from a **source**  $s$ , and discover all the reachable vertices.

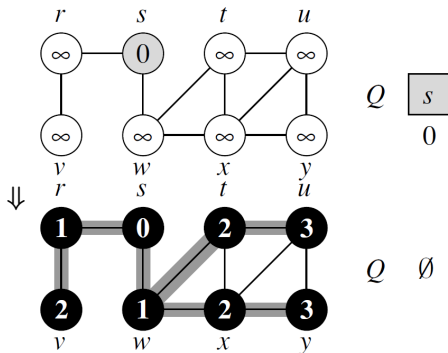
## Breadth-first search (BFS)



We start from a **source**  $s$ , and discover all the reachable vertices.

Each vertex has its **distance** to  $s$  ( $\#$  edges) computed **incrementally**.

# Breadth-first search (BFS)

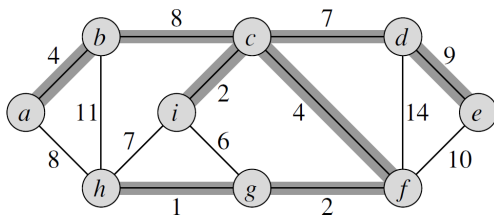


We start from a **source**  $s$ , and discover all the reachable vertices.

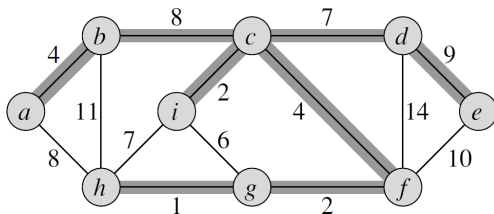
Each vertex has its **distance** to  $s$  ( $\#$  edges) computed **incrementally**.

A **breadth-first tree** is obtained as a side-product.

# Minimum spanning tree (MST)



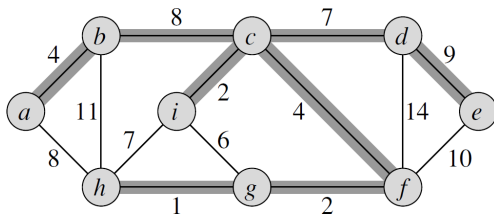
# Minimum spanning tree (MST)



For a **weighted** graph  $G = (V, E)$ , find a subset of edges  $T \subseteq E$  s.t.:

- The set is **acyclic**.
- The set **covers** all the vertices.
- The sum of edge weights is **minimized**.

# Minimum spanning tree (MST)



For a **weighted** graph  $G = (V, E)$ , find a subset of edges  $T \subseteq E$  s.t.:

- The set is **acyclic**.
- The set **covers** all the vertices.
- The sum of edge weights is **minimized**.

The MST is not unique: you can replace  $(b, c)$  with  $(a, h)$ .

# Overall idea

GENERIC-MST( $G, w$ )

```
1   $A \leftarrow \emptyset$ 
2  while  $A$  does not form a spanning tree
3      do find an edge  $(u, v)$  that is safe for  $A$ 
4       $A \leftarrow A \cup \{(u, v)\}$ 
5  return  $A$ 
```

A **safe edge**  $(u, v)$  ensures that  $A \cup \{(u, v)\}$  is a subset of some MST.

At the end, the set  $A$  must then be a MST.



# Overall idea

GENERIC-MST( $G, w$ )

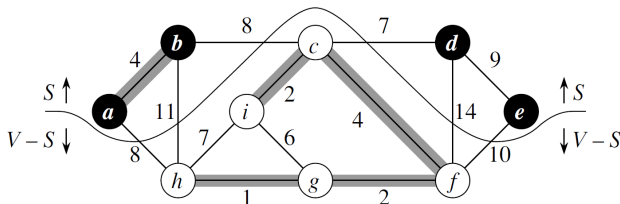
```
1   $A \leftarrow \emptyset$   
2  while  $A$  does not form a spanning tree  
3      do find an edge  $(u, v)$  that is safe for  $A$   
4       $A \leftarrow A \cup \{(u, v)\}$   
5  return  $A$ 
```

A **safe edge**  $(u, v)$  ensures that  $A \cup \{(u, v)\}$  is a subset of some MST.

At the end, the set  $A$  must then be a MST.

How to find safe edges?

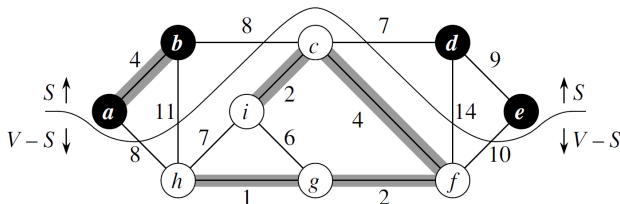
# Definitions and theorem



A **cut**  $(S, V - S)$  partitions the vertices  $V$ .

The edge  $(a, h)$  **crosses** the cut.

# Definitions and theorem

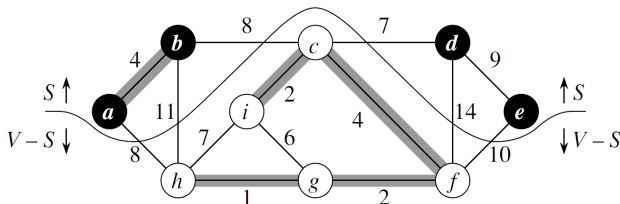


A **cut**  $(S, V - S)$  partitions the vertices  $V$ .

The edge  $(a, h)$  **crosses** the cut.

The cut **respects** **A** since none of the edges in **A** crosses the cut.

# Definitions and theorem



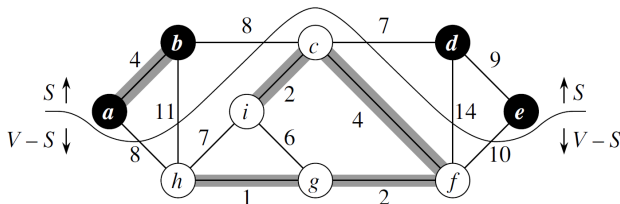
A **cut**  $(S, V - S)$  partitions the vertices  $V$ .

The edge  $(a, h)$  **crosses** the cut.

The cut **respects** **A** since none of the edges in **A** crosses the cut.

The edge  $(d, c)$  is a **light edge** since it is the minimal one crossing the cut.

# Definitions and theorem



A **cut**  $(S, V - S)$  partitions the vertices  $V$ .

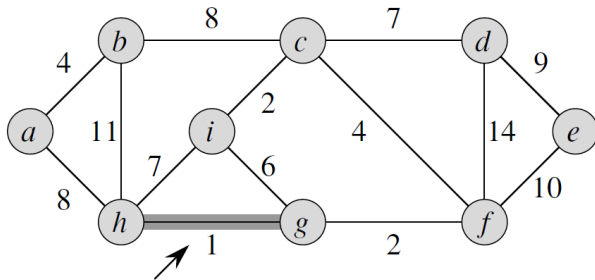
The edge  $(a, h)$  **crosses** the cut.

The cut **respects**  $A$  since none of the edges in  $A$  crosses the cut.

The edge  $(d, c)$  is a **light edge** since it is the minimal one crossing the cut.

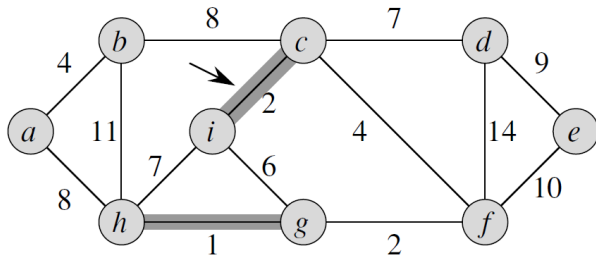
If  $A$  is included in a MST, then each light edge is safe for  $A$ .

# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

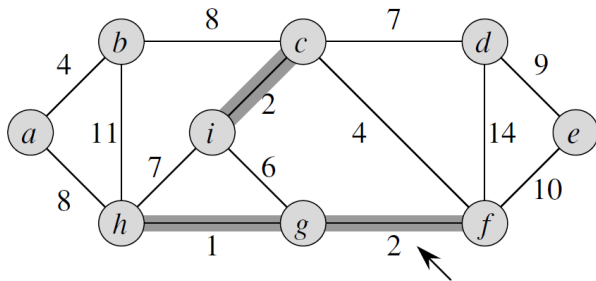
# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

Each edge generates its own tree

# Kruskal's algorithm

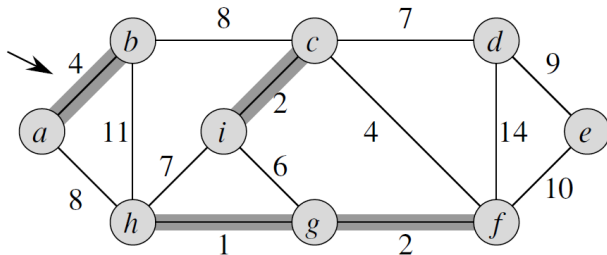


Start exploring each edge by **increasing weight**.

Each edge generates its own tree, or enters an existing tree.



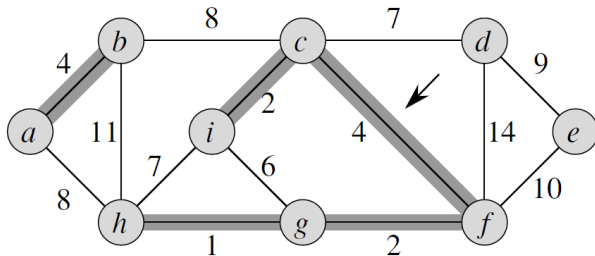
# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

Each edge generates its own tree, or enters an existing tree.

# Kruskal's algorithm

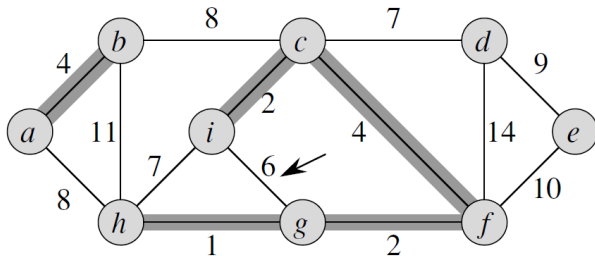


Start exploring each edge by **increasing weight**.

Each edge generates its own tree, or enters an existing tree.

A new edge may **merge** two existing trees.

# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

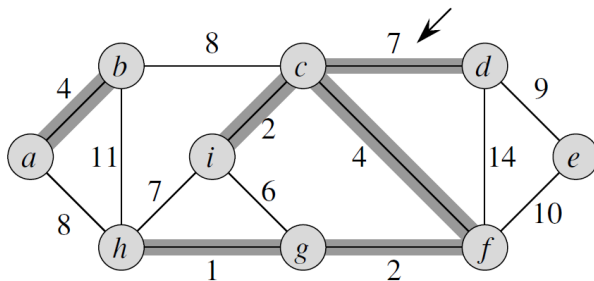
Each edge generates its own tree, or enters an existing tree.

A new edge may **merge** two existing trees.

Edge  $(i, g)$  is **not safe** since it does not cross **any** cut.

Edge  $(c, f)$  was safe since it was the minimal one crossing the cut.

# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

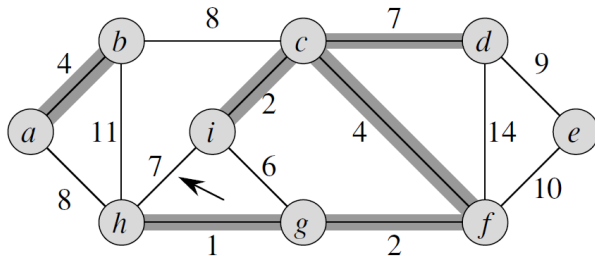
Each edge generates its own tree, or enters an existing tree.

A new edge may **merge** two existing trees.

Edge  $(i, g)$  is **not safe** since it does not cross **any** cut.

Edge  $(c, f)$  was safe since it was the minimal one crossing the cut.

# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

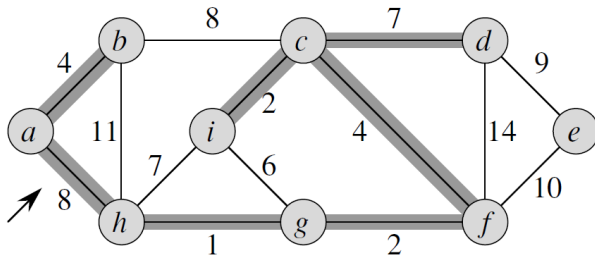
Each edge generates its own tree, or enters an existing tree.

A new edge may **merge** two existing trees.

Edge  $(i, g)$  is **not safe** since it does not cross **any** cut.

Edge  $(c, f)$  was safe since it was the minimal one crossing the cut.

# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

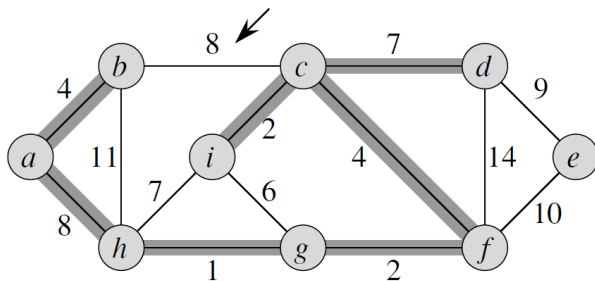
Each edge generates its own tree, or enters an existing tree.

A new edge may **merge** two existing trees.

Edge  $(i, g)$  is **not safe** since it does not cross **any** cut.

Edge  $(c, f)$  was safe since it was the minimal one crossing the cut.

# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

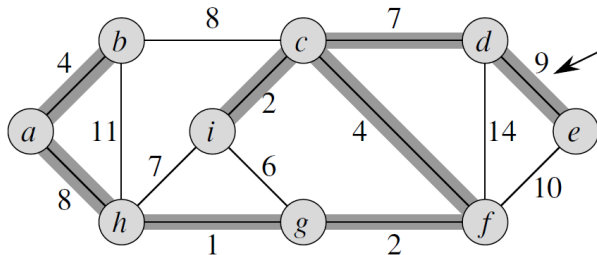
Each edge generates its own tree, or enters an existing tree.

A new edge may **merge** two existing trees.

Edge  $(i, g)$  is **not safe** since it does not cross **any** cut.

Edge  $(c, f)$  was safe since it was the minimal one crossing the cut.

# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

Each edge generates its own tree, or enters an existing tree.

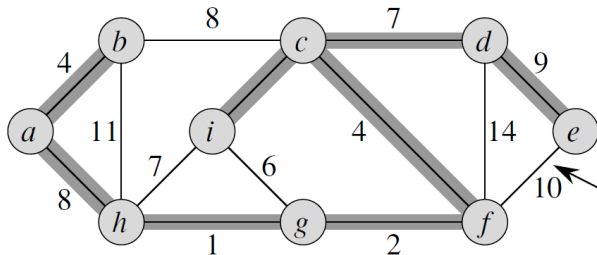
A new edge may **merge** two existing trees.

Edge  $(i, g)$  is **not safe** since it does not cross **any** cut.

Edge  $(c, f)$  was safe since it was the minimal one crossing the cut.



# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

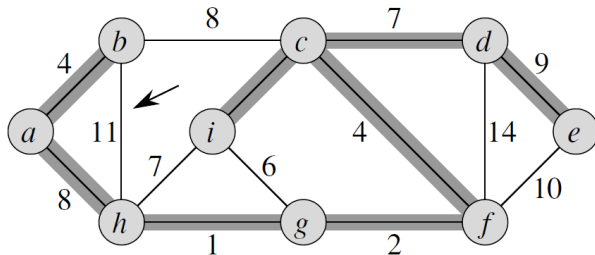
Each edge generates its own tree, or enters an existing tree.

A new edge may **merge** two existing trees.

Edge  $(i, g)$  is **not safe** since it does not cross **any** cut.

Edge  $(c, f)$  was safe since it was the minimal one crossing the cut.

# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

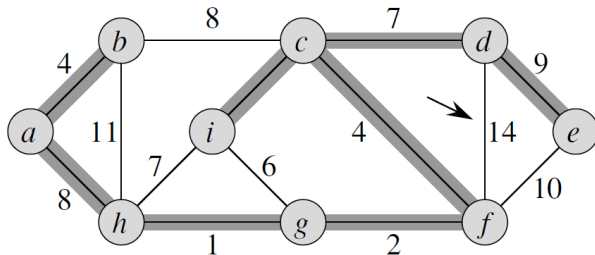
Each edge generates its own tree, or enters an existing tree.

A new edge may **merge** two existing trees.

Edge  $(i, g)$  is **not safe** since it does not cross **any** cut.

Edge  $(c, f)$  was safe since it was the minimal one crossing the cut.

# Kruskal's algorithm



Start exploring each edge by **increasing weight**.

Each edge generates its own tree, or enters an existing tree.

A new edge may **merge** two existing trees.

Edge  $(i, g)$  is **not safe** since it does not cross **any** cut.

Edge  $(c, f)$  was safe since it was the minimal one crossing the cut.

# Kruskal's algorithm

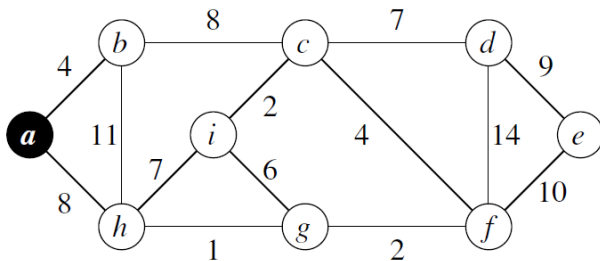
MST-KRUSKAL( $G, w$ )

```
1   $A \leftarrow \emptyset$ 
2  for each vertex  $v \in V[G]$ 
3      do MAKE-SET( $v$ )
4  sort the edges of  $E$  into nondecreasing order by weight  $w$ 
5  for each edge  $(u, v) \in E$ , taken in nondecreasing order by weight
6      do if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7          then  $A \leftarrow A \cup \{(u, v)\}$ 
8              UNION( $u, v$ )
9  return  $A$ 
```

FIND-SET( $u$ ) returns the tree to which  $u$  belongs.

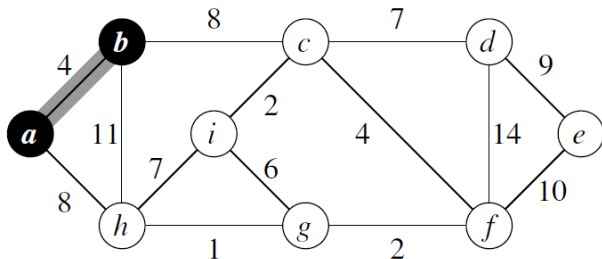
The check at line 6 avoids the creation of **cycles**.

# Prim's algorithm



Start from an arbitrary root.

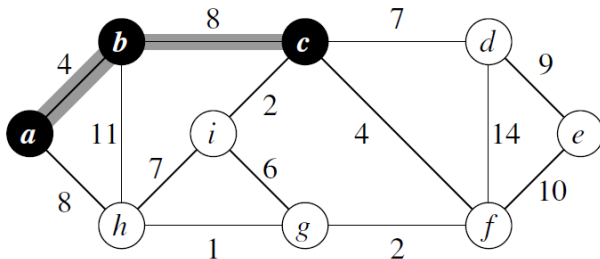
# Prim's algorithm



Start from an arbitrary root.

Grow the tree by adding a safe edge at each step greedily.

# Prim's algorithm

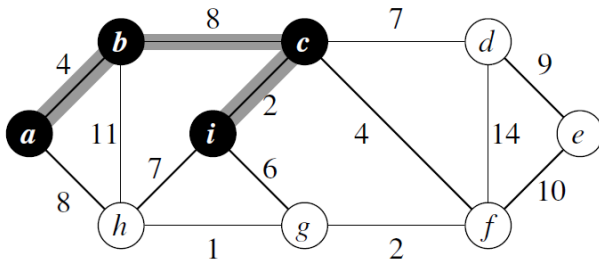


Start from an arbitrary root.

Grow the tree by adding a safe edge at each step greedily.

We can choose between edge  $(b, c)$  or edge  $(a, h)$ , since they both are at the frontier of the current tree and have equal weight.

# Prim's algorithm



Start from an arbitrary root.

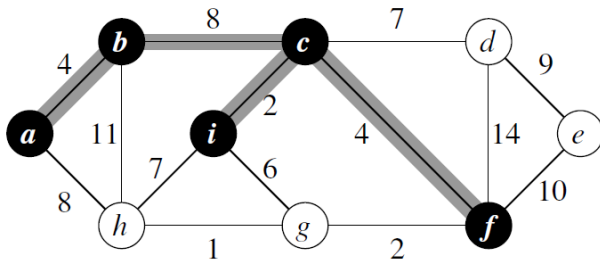
Grow the tree by adding a safe edge at each step greedily.

We can choose between edge  $(b, c)$  or edge  $(a, h)$ , since they both are at the frontier of the current tree and have equal weight.

At each step the selected vertices determine a cut, and a light edge crossing the cut is added to the tree.



# Prim's algorithm



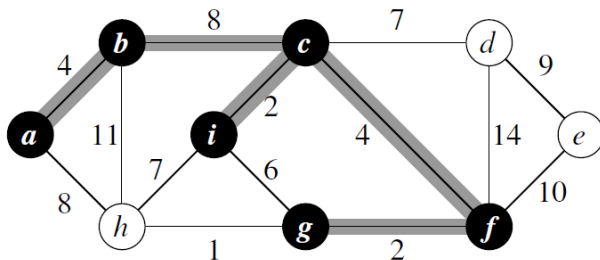
Start from an arbitrary root.

Grow the tree by adding a safe edge at each step greedily.

We can choose between edge  $(b, c)$  or edge  $(a, h)$ , since they both are at the frontier of the current tree and have equal weight.

At each step the selected vertices determine a cut, and a light edge crossing the cut is added to the tree.

# Prim's algorithm



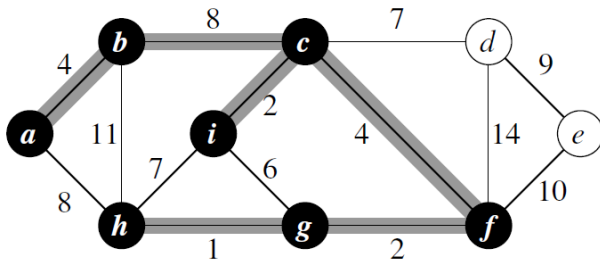
Start from an arbitrary root.

Grow the tree by adding a safe edge at each step greedily.

We can choose between edge  $(b, c)$  or edge  $(a, h)$ , since they both are at the frontier of the current tree and have equal weight.

At each step the selected vertices determine a cut, and a light edge crossing the cut is added to the tree.

# Prim's algorithm



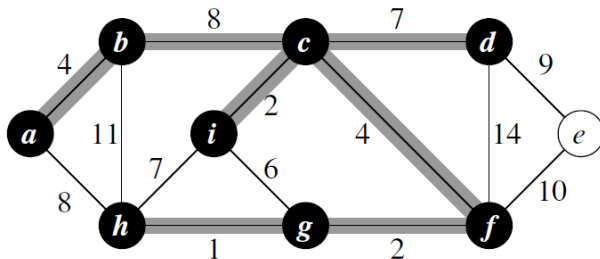
Start from an arbitrary root.

Grow the tree by adding a safe edge at each step greedily.

We can choose between edge  $(b, c)$  or edge  $(a, h)$ , since they both are at the frontier of the current tree and have equal weight.

At each step the selected vertices determine a cut, and a light edge crossing the cut is added to the tree.

# Prim's algorithm



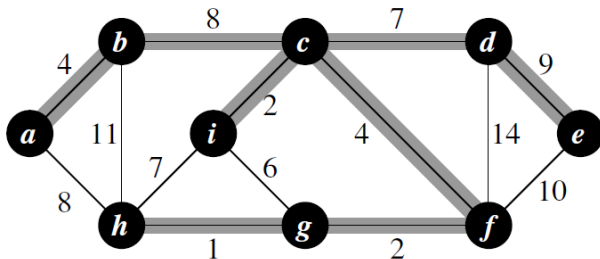
Start from an arbitrary root.

Grow the tree by adding a safe edge at each step greedily.

We can choose between edge  $(b, c)$  or edge  $(a, h)$ , since they both are at the frontier of the current tree and have equal weight.

At each step the selected vertices determine a cut, and a light edge crossing the cut is added to the tree.

# Prim's algorithm



Start from an arbitrary root.

Grow the tree by adding a safe edge at each step greedily.

We can choose between edge  $(b, c)$  or edge  $(a, h)$ , since they both are at the **frontier** of the current tree and have equal weight.

At each step the selected vertices determine a **cut**, and a light edge crossing the cut is added to the tree.

# Prim's algorithm

MST-PRIM( $G, w, r$ )

```
1  for each  $u \in V[G]$ 
2      do  $key[u] \leftarrow \infty$ 
3       $\pi[u] \leftarrow \text{NIL}$ 
```

Each vertex will have a **parent** (in the tree) and a **key**.

# Prim's algorithm

MST-PRIM( $G, w, r$ )

```
1  for each  $u \in V[G]$ 
2      do  $key[u] \leftarrow \infty$ 
3       $\pi[u] \leftarrow \text{NIL}$ 
4   $key[r] \leftarrow 0$             $r$  is the chosen root
5   $Q \leftarrow V[G]$            init. a priority queue with all vertices
```

Each vertex will have a **parent** (in the tree) and a **key**.

The **priority queue** is based on the value of the key.

# Prim's algorithm

```
MST-PRIM( $G, w, r$ )
1  for each  $u \in V[G]$ 
2      do  $key[u] \leftarrow \infty$ 
3       $\pi[u] \leftarrow \text{NIL}$ 
4   $key[r] \leftarrow 0$ 
5   $Q \leftarrow V[G]$ 
6  while  $Q \neq \emptyset$ 
7      do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
8          for each  $v \in \text{Adj}[u]$ 
9              do if  $v \in Q$  and  $w(u, v) < key[v]$ 
10                  then  $\pi[v] \leftarrow u$ 
11                       $key[v] \leftarrow w(u, v)$ 
```

Each vertex will have a **parent** (in the tree) and a **key**.

The **priority queue** is based on the value of the key.

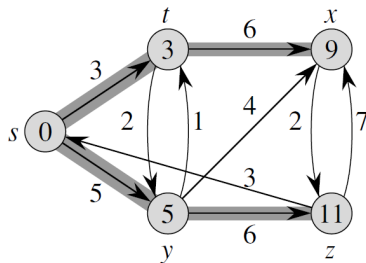
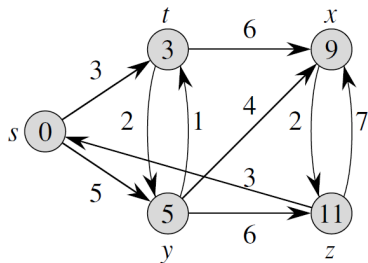
The **key** of a vertex  $v$  will hold the minimum weight connecting  $v$  to a vertex that is **already in the tree**.



# Shortest paths

We now want to compute the **shortest path** (i.e. the sequence of vertices) from a single source to all the other vertices.

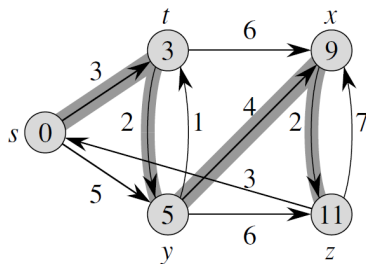
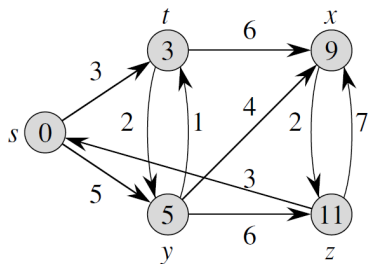
To keep track of the path, we store a **predecessor** for each vertex (similarly to what we did with the BFS tree).



# Shortest paths

We now want to compute the **shortest path** (i.e. the sequence of vertices) from a single source to all the other vertices.

To keep track of the path, we store a **predecessor** for each vertex (similarly to what we did with the BFS tree).



The shortest path is not necessarily **unique**.

# Relaxation

Instead of exact distances, we compute **upper-bound** estimates.

These will be updated to be exact by a sequence of **relaxation** steps.

# Relaxation

Instead of exact distances, we compute **upper-bound** estimates.

These will be updated to be exact by a sequence of **relaxation** steps.

INITIALIZE-SINGLE-SOURCE( $G, s$ )

```
1  for each vertex  $v \in V[G]$ 
2      do  $d[v] \leftarrow \infty$    distance estimate
3       $\pi[v] \leftarrow \text{NIL}$    predecessor
4   $d[s] \leftarrow 0$ 
```

# Relaxation

Instead of exact distances, we compute **upper-bound** estimates.

These will be updated to be exact by a sequence of **relaxation** steps.

INITIALIZE-SINGLE-SOURCE( $G, s$ )

```
1  for each vertex  $v \in V[G]$ 
2      do  $d[v] \leftarrow \infty$    distance estimate
3       $\pi[v] \leftarrow \text{NIL}$    predecessor
4   $d[s] \leftarrow 0$ 
```

An edge  $(u, v)$  with weight  $w$  is relaxed as:

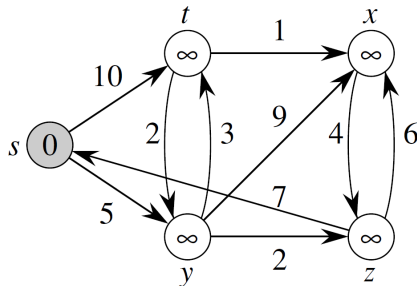
RELAX( $u, v, w$ )

```
1  if  $d[v] > d[u] + w(u, v)$    “can we improve the path to  $v$ ?”
2      then  $d[v] \leftarrow d[u] + w(u, v)$ 
3       $\pi[v] \leftarrow u$ 
```

It tests if we can improve the shortest path to  $v$  by going through  $u$  and, if so, **decreases**  $d[v]$  and **updates**  $\pi[v]$ .

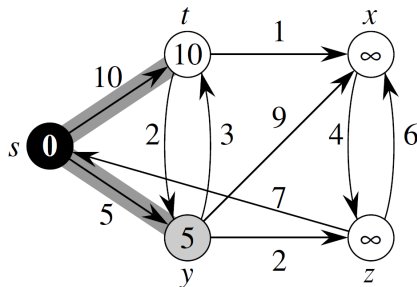
# Dijkstra's algorithm

```
DIJKSTRA( $G, w, s$ )  
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )  
2   $S \leftarrow \emptyset$   
3   $Q \leftarrow V[G]$   
4  while  $Q \neq \emptyset$   
   shaded vertex  $\rightarrow$  5      do  $u \leftarrow \text{EXTRACT-MIN}(Q)$   
6       $S \leftarrow S \cup \{u\}$   
7      for each vertex  $v \in \text{Adj}[u]$   
8          do RELAX( $u, v, w$ )
```



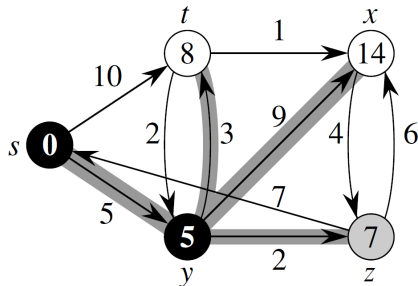
# Dijkstra's algorithm

```
DIJKSTRA( $G, w, s$ )  
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )  
2   $S \leftarrow \emptyset$   
3   $Q \leftarrow V[G]$   
4  while  $Q \neq \emptyset$   
   shaded vertex  $\rightarrow$  5      do  $u \leftarrow \text{EXTRACT-MIN}(Q)$   
6       $S \leftarrow S \cup \{u\}$   
7      for each vertex  $v \in \text{Adj}[u]$   
8          do RELAX( $u, v, w$ )
```



# Dijkstra's algorithm

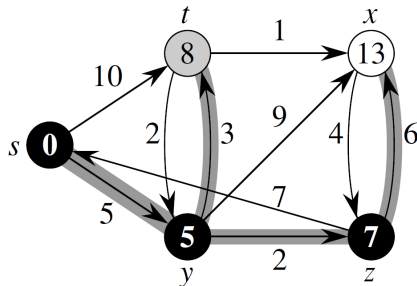
```
DIJKSTRA( $G, w, s$ )  
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )  
2   $S \leftarrow \emptyset$   
3   $Q \leftarrow V[G]$   
4  while  $Q \neq \emptyset$   
   shaded vertex  $\rightarrow$  5      do  $u \leftarrow \text{EXTRACT-MIN}(Q)$   
6       $S \leftarrow S \cup \{u\}$   
7      for each vertex  $v \in \text{Adj}[u]$   
8          do RELAX( $u, v, w$ )
```





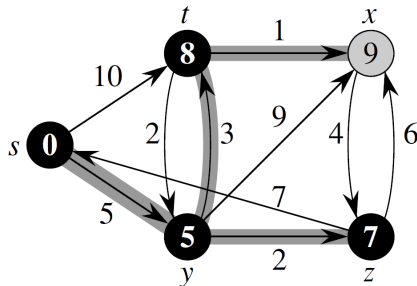
# Dijkstra's algorithm

```
DIJKSTRA( $G, w, s$ )  
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )  
2   $S \leftarrow \emptyset$   
3   $Q \leftarrow V[G]$   
4  while  $Q \neq \emptyset$   
   shaded vertex  $\rightarrow$  5      do  $u \leftarrow \text{EXTRACT-MIN}(Q)$   
6       $S \leftarrow S \cup \{u\}$   
7      for each vertex  $v \in \text{Adj}[u]$   
8          do RELAX( $u, v, w$ )
```



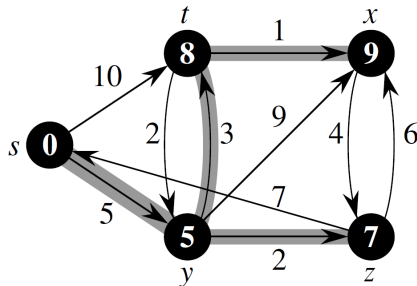
# Dijkstra's algorithm

```
DIJKSTRA( $G, w, s$ )  
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )  
2   $S \leftarrow \emptyset$   
3   $Q \leftarrow V[G]$   
4  while  $Q \neq \emptyset$   
   shaded vertex  $\rightarrow$  5      do  $u \leftarrow \text{EXTRACT-MIN}(Q)$   
6       $S \leftarrow S \cup \{u\}$   
7      for each vertex  $v \in \text{Adj}[u]$   
8          do RELAX( $u, v, w$ )
```

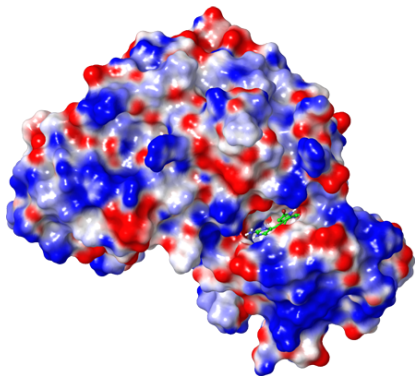


# Dijkstra's algorithm

```
DIJKSTRA( $G, w, s$ )  
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )  
2   $S \leftarrow \emptyset$   
3   $Q \leftarrow V[G]$   
4  while  $Q \neq \emptyset$   
   shaded vertex  $\rightarrow$  5      do  $u \leftarrow \text{EXTRACT-MIN}(Q)$   
6       $S \leftarrow S \cup \{u\}$   
7      for each vertex  $v \in \text{Adj}[u]$   
8          do RELAX( $u, v, w$ )
```



# Demo



# Exercises

Implement [Dijkstra's algorithm](#).

Test it on a random, undirected, weighted graphs generated by yourself.

## Suggested reading

Chapters 23.1, 23.2, 24 (“Representing shortest paths” and “Relaxation” paragraphs) and 24.3 of:

“Introduction to Algorithms – 2nd Ed.”, Cormen et al.