Algorithms and Data Structures (II)

Gabriel Istrate

May 13, 2020

Where are we:

Graphs: Representations and Algorithms

- Graph traversals: BFS, DFS, Topological Sorting.
- Continue with Minimum Spanning Tree.

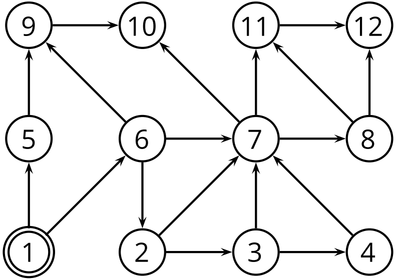
- One of the simplest but fundamental algorithms

- One of the simplest but fundamental algorithms

- *Input:* $G = (V, E)$ and a vertex $s \in V$
  - explores the graph, touching all vertices that are reachable from s
  - iterates through the vertices at increasing distance (edge distance)
  - computes the distance of each vertex from s
  - produces a ***breadth-first tree*** rooted at s
  - works on both *directed* and *undirected* graphs

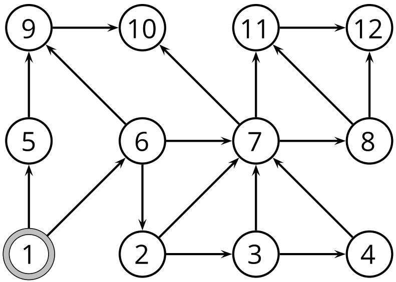
BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```



BFS Algorithm

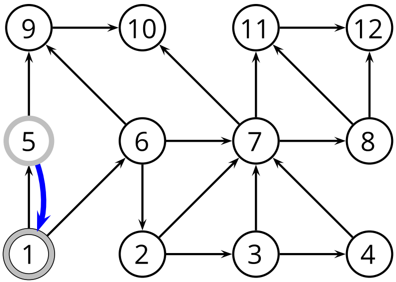
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```



$u = 1$

$Q = \emptyset$

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

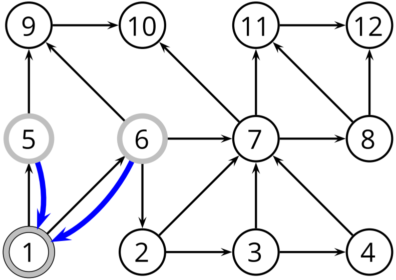


$u = 1$

$Q = \{5\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

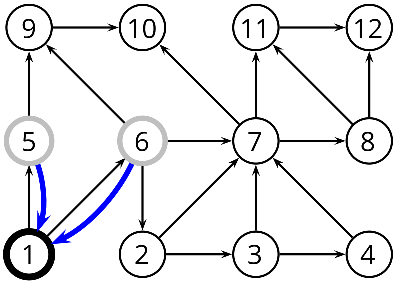


$u = 1$

$Q = \{5, 6\}$

BFS Algorithm

```
BFS(G, s)
    for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
    color[s] = GRAY
    d[s] = 0
    π[s] = NIL
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
            if color[v] == WHITE
                color[v] = GRAY
                d[v] = d[u] + 1
                π[v] = u
                ENQUEUE(Q, v)
        color[u] = BLACK
```

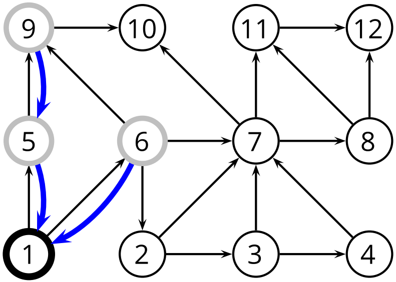


$u = 5$

$Q = \{6\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

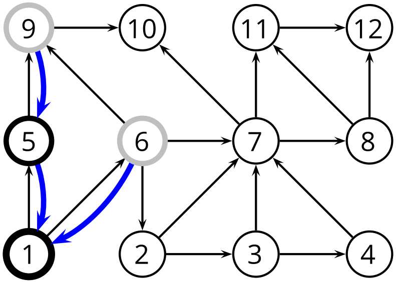


$u = 5$

$Q = \{6, 9\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

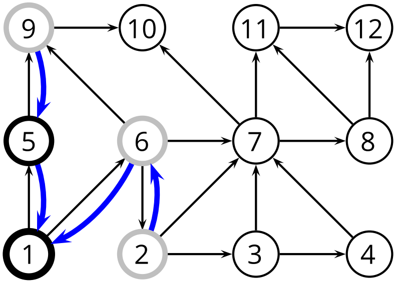


$u = 6$

$Q = \{9\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

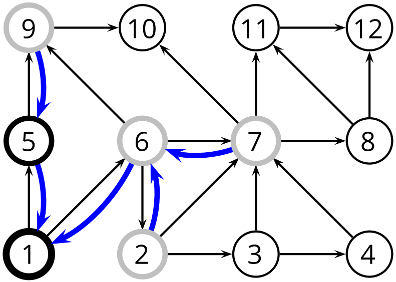


$u = 6$

$Q = \{9, 2, 7\}$

BFS Algorithm

```
BFS(G, s)
    for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
    color[s] = GRAY
    d[s] = 0
    π[s] = NIL
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
            if color[v] == WHITE
                color[v] = GRAY
                d[v] = d[u] + 1
                π[v] = u
                ENQUEUE(Q, v)
        color[u] = BLACK
```

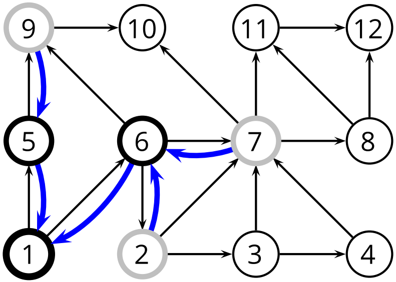


$u = 6$

$Q = \{9, 2, 7\}$

BFS Algorithm

```
BFS(G, s)
    for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
    color[s] = GRAY
    d[s] = 0
    π[s] = NIL
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
            if color[v] == WHITE
                color[v] = GRAY
                d[v] = d[u] + 1
                π[v] = u
                ENQUEUE(Q, v)
        color[u] = BLACK
```

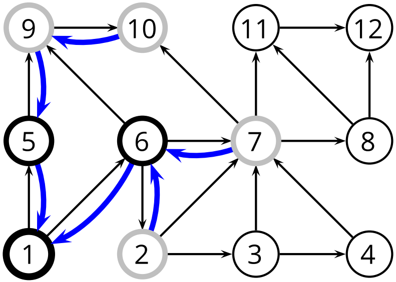


$u = 9$

$Q = \{2, 7\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

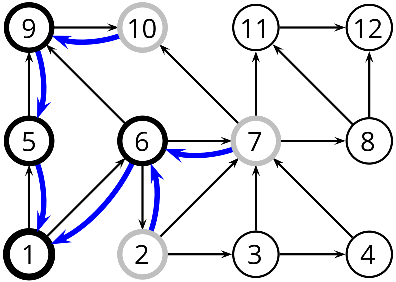


$u = 9$

$Q = \{2, 7, 10\}$

BFS Algorithm

```
BFS(G, s)
    for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
    color[s] = GRAY
    d[s] = 0
    π[s] = NIL
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
            if color[v] == WHITE
                color[v] = GRAY
                d[v] = d[u] + 1
                π[v] = u
                ENQUEUE(Q, v)
        color[u] = BLACK
```

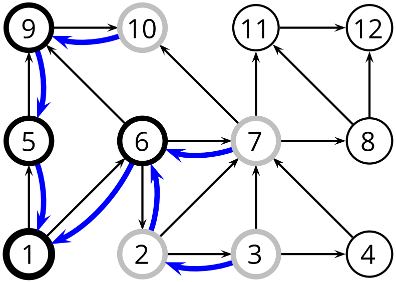


$u = 2$

$Q = \{7, 10\}$

BFS Algorithm

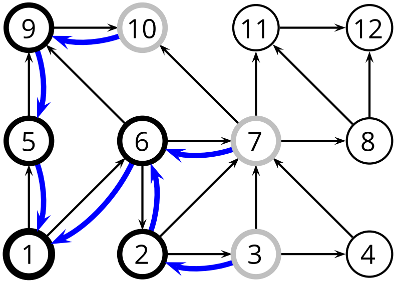
```
BFS(G, s)
    for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
    color[s] = GRAY
    d[s] = 0
    π[s] = NIL
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
            if color[v] == WHITE
                color[v] = GRAY
                d[v] = d[u] + 1
                π[v] = u
                ENQUEUE(Q, v)
        color[u] = BLACK
```



$u = 2$

$Q = \{7, 10, 3\}$

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

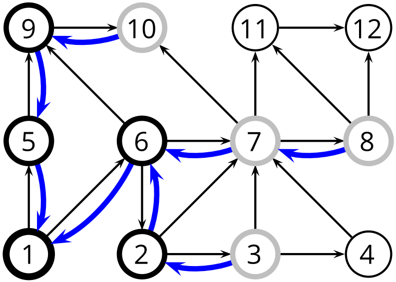


$u = 7$

$Q = \{10, 3\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

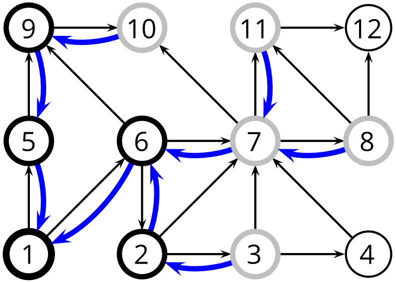


$u = 7$

$Q = \{10, 3, 8\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

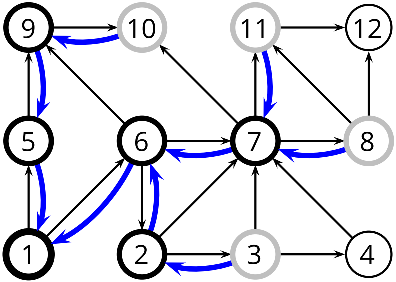


$u = 7$

$Q = \{10, 3, 8, 11\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

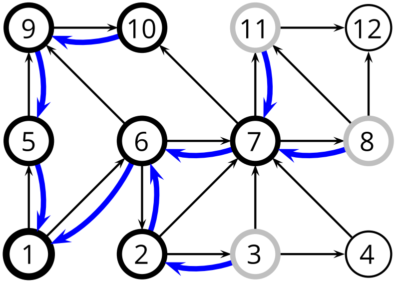


$u = 10$

$Q = \{3, 8, 11\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

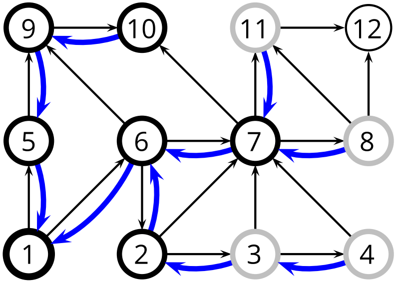


$u = 3$

$Q = \{8, 11\}$

BFS Algorithm

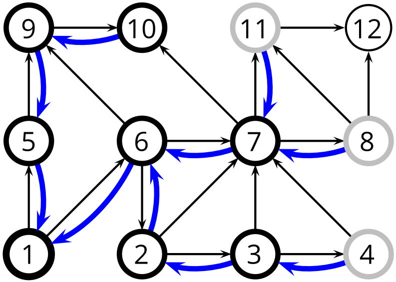
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```



$u = 3$

$Q = \{8, 11, 4\}$

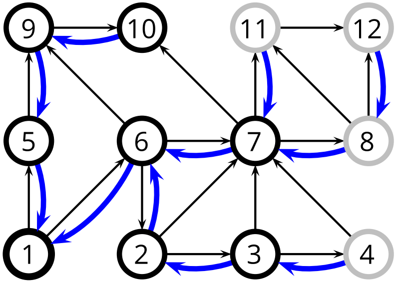
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```



$u = 8$

$Q = \{11, 4\}$

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

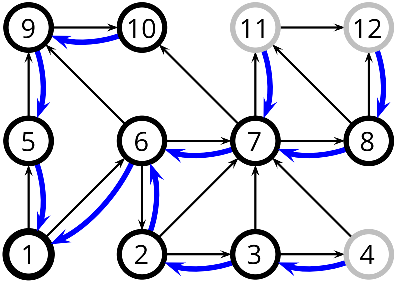


$u = 8$

$Q = \{11, 4, 12\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

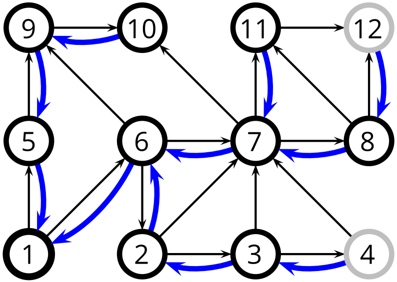


$u = 11$

$Q = \{4, 12\}$

BFS Algorithm

```
BFS(G, s)
    for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
    color[s] = GRAY
    d[s] = 0
    π[s] = NIL
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
            if color[v] == WHITE
                color[v] = GRAY
                d[v] = d[u] + 1
                π[v] = u
                ENQUEUE(Q, v)
        color[u] = BLACK
```

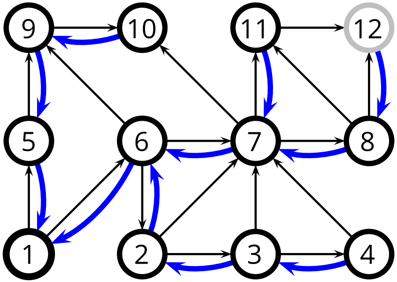


$u = 4$

$Q = \{12\}$

BFS Algorithm

```
BFS(G, s)
  for each vertex u ∈ V(G) \ {s}
      color[u] = WHITE
      d[u] = ∞
      π[u] = NIL
  color[s] = GRAY
  d[s] = 0
  π[s] = NIL
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
      u = DEQUEUE(Q)
      for each v ∈ Adj[u]
          if color[v] == WHITE
              color[v] = GRAY
              d[v] = d[u] + 1
              π[v] = u
              ENQUEUE(Q, v)
      color[u] = BLACK
```

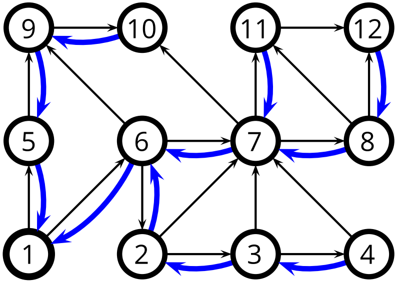


$u = 12$

$Q = \emptyset$

BFS Algorithm

```
BFS(G, s)
    for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
    color[s] = GRAY
    d[s] = 0
    π[s] = NIL
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
            if color[v] == WHITE
                color[v] = GRAY
                d[v] = d[u] + 1
                π[v] = u
                ENQUEUE(Q, v)
        color[u] = BLACK
```



Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*

Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times

Complexity of BFS

```
BFS(G, s)
  for each vertex u ∈ V(G) \ {s}
      color[u] = WHITE
      d[u] = ∞
      π[u] = NIL
  color[s] = GRAY
  d[s] = 0
  π[s] = NIL
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
      u = DEQUEUE(Q)
      for each v ∈ Adj[u]
          if color[v] == WHITE
              color[v] = GRAY
              d[v] = d[u] + 1
              π[v] = u
              ENQUEUE(Q, v)
      color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times
- For each vertex u, the inner loop executes $\Theta(|E_u|)$, for a total of $O(|E|)$ steps

Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times
- For each vertex u, the inner loop executes $\Theta(|E_u|)$, for a total of $O(|E|)$ steps
- So, $O(|V| + |E|)$

Depth-First Search

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
  - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
  - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited
- *Input:* $G = (V, E)$
  - explores the graph, touching *all vertices*

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
  - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited

- *Input:* $G = (V, E)$
  - explores the graph, touching *all vertices*
  - produces a ***depth-first forest***, consisting of all the ***depth-first trees*** defined by the DFS exploration

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
  - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited

- *Input:* $G = (V, E)$
  - explores the graph, touching *all vertices*
  - produces a ***depth-first forest***, consisting of all the ***depth-first trees*** defined by the DFS exploration
  - associates ***two time-stamps*** to each vertex
    - $d[u]$ records when u is first discovered
    - $f[u]$ records when DFS finishes examining u's edges, and therefore backtracks from u

DFS Algorithm

```
DFS(G)
    for each vertex u ∈ V(G)
        color[u] = WHITE
        π[u] = NIL
    time = 0 // "global" variable
    for each vertex u ∈ V(G)
        if color[u] == WHITE
            DFS-VISIT(u)
```

```
DFS-VISIT(u)
    color[u] = GREY
    time = time + 1
    d[u] = time
    for each v ∈ Adj[u]
        if color[v] == WHITE
            π[v] = u
            DFS-VISIT(v)
    color[u] = BLACK
    time = time + 1
    f[u] = time
```

Complexity of DFS

- The loop in **DFS-VISIT**(u) (lines 4–7) accounts for $\Theta(|E_u|)$

- The loop in **DFS-VISIT**(u) (lines 4–7) accounts for $\Theta(|E_u|)$
- We call **DFS-VISIT**(u) *once* for each vertex u
  - either in **DFS**, or recursively in **DFS-VISIT**
  - because we call it only if $color[u] = $ WHITE, but then we immediately set $color[u] = $ GREY

- The loop in **DFS-VISIT**(u) (lines 4–7) accounts for $\Theta(|E_u|)$

- We call **DFS-VISIT**(u) *once* for each vertex u
  - either in **DFS**, or recursively in **DFS-VISIT**
  - because we call it only if $color[u] = $ WHITE, but then we immediately set $color[u] = $ GREY

- So, the overall complexity is $\Theta(|V| + |E|)$

Applications of DFS: Topological Sort

Applications of DFS: Topological Sort

- **Problem:** (topological sort)

  Given a *directed acyclic graph* (DAG)

  - find an ordering of vertices such that you only end up with *forward links*

Applications of DFS: Topological Sort

- **Problem:** (topological sort)

  Given a *directed acyclic graph* (DAG)

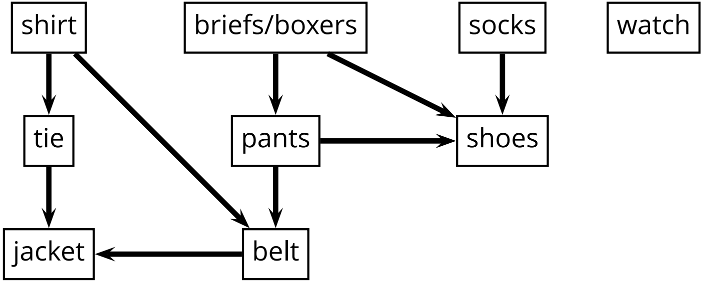
  - find an ordering of vertices such that you only end up with *forward links*

- **Example:** dependencies in software packages

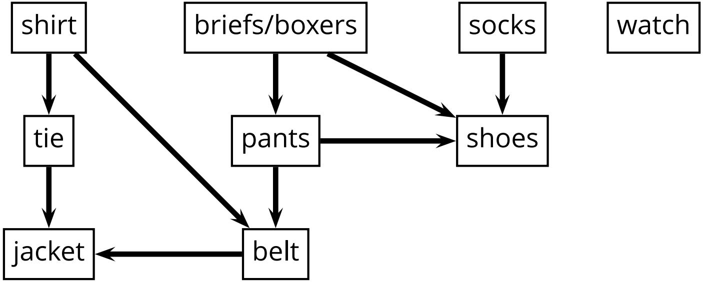
  - find an installation order for a set of software packages
  - such that every package is installed only after all the packages it depends on

Topological Sort Algorithm

Topological Sort Algorithm



Topological Sort Algorithm



TOPOLOGICAL-SORT(G)
1 **DFS**(G)
2 output V sorted in reverse order of $f[\cdot]$

- Given a weighted graph $G = (V, E)$
  - with “weight” function $w : E \longrightarrow \mathbb{R}$

- Given a weighted graph $G = (V, E)$
  - with "weight" function $w : E \longrightarrow \mathbb{R}$
- Find an *acyclic* subset $T \subseteq E$
  - a ***tree***

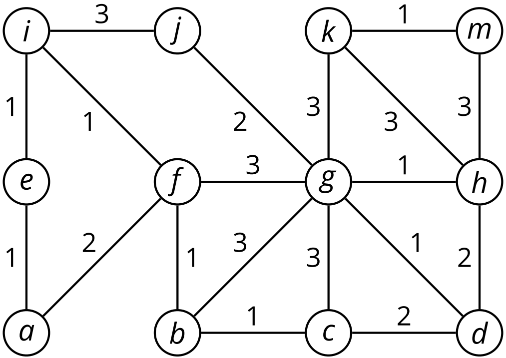
- Given a weighted graph $G = (V, E)$
  - with "weight" function $w : E \longrightarrow \mathbb{R}$
- Find an *acyclic* subset $T \subseteq E$
  - a ***tree***
- T "spans" the entire graph G (i.e., touches all vertexes)
  - a ***spanning tree***

- Given a weighted graph $G = (V, E)$
  - with "weight" function $w : E \longrightarrow \mathbb{R}$
- Find an *acyclic* subset $T \subseteq E$
  - a ***tree***
- T "spans" the entire graph G (i.e., touches all vertexes)
  - a ***spanning tree***
- T's total weight of the tree is minimal

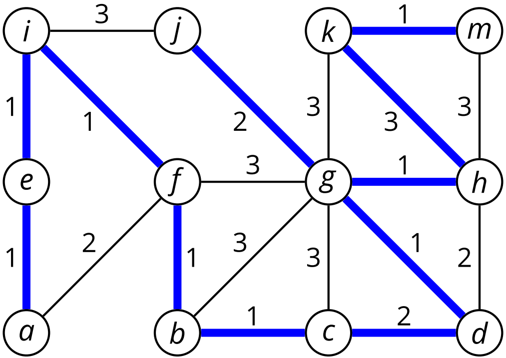
$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

  - a ***minimum-weight spanning tree***, or "minimum spanning tree"

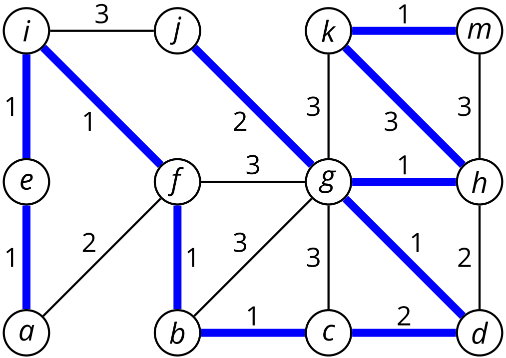
Example



Example

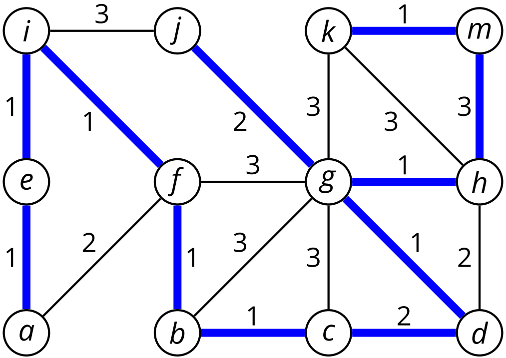


Example



- There may be more than one minimum spanning tree

Example



- There may be more than one minimum spanning tree

General Idea

General Idea

- Build *T* by adding one edge at a time

General Idea

- Build T by adding one edge at a time
- Each time, add $e \in E$ such that
  - e is the ***lightest edge***
  - e ***does not create a cycle***
- Stop when we have covered all vertexes

General Idea

- Build T by adding one edge at a time
- Each time, add $e \in E$ such that
  - e is the ***lightest edge***
  - e ***does not create a cycle***
- Stop when we have covered all vertexes
- This is a *greedy algorithm*

- Build T by adding one edge at a time
- Each time, add $e \in E$ such that
  - e is the ***lightest edge***
  - e ***does not create a cycle***
- Stop when we have covered all vertexes
- This is a *greedy algorithm*
- Does it work?

Refined Idea

```
GENERIC-MST(G, w)
    A = ∅
    while A is not a spanning tree
        find a safe edge e = (u, v)
        A = A ∪ {e}
```

```
GENERIC-MST(G, w)
  A = ∅
  while A is not a spanning tree
      find a safe edge e = (u, v)
      A = A ∪ {e}
```

- ***Invariant:*** A is a subset of a minimum spanning tree

```
GENERIC-MST(G, w)
    A = ∅
    while A is not a spanning tree
        find a safe edge e = (u, v)
        A = A ∪ {e}
```

- ***Invariant:*** A is a subset of a minimum spanning tree
- A ***safe edge*** is an edge that maintains the invariant
  - e is such that, if A is a subset of a minimum spanning tree, then $A \cup \{e\}$ is also a subset of a minimum spanning tree

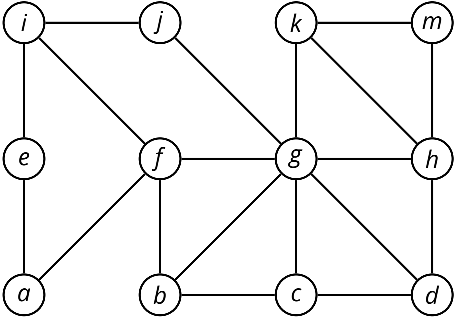
Refined Idea

```
GENERIC-MST(G, w)
A = ∅
while A is not a spanning tree
    find a safe edge e = (u, v)
    A = A ∪ {e}
```

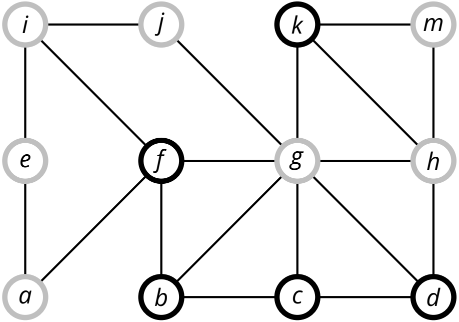
- ***Invariant:*** A is a subset of a minimum spanning tree
- A ***safe edge*** is an edge that maintains the invariant
  - e is such that, if A is a subset of a minimum spanning tree, then $A \cup \{e\}$ is also a subset of a minimum spanning tree
  - more or less the *definition* of a greedy algorithm

- A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a *partition* of V

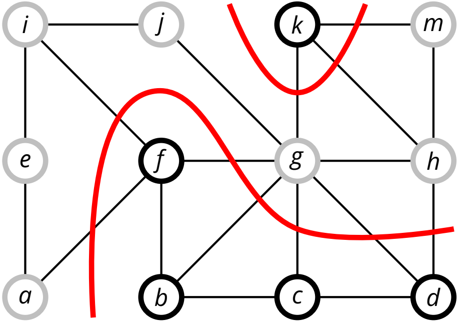
- A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a *partition* of V



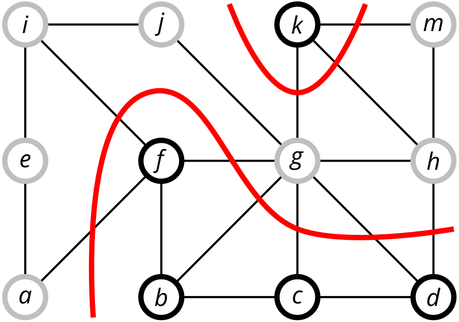
- A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a *partition* of V



- A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a *partition* of V

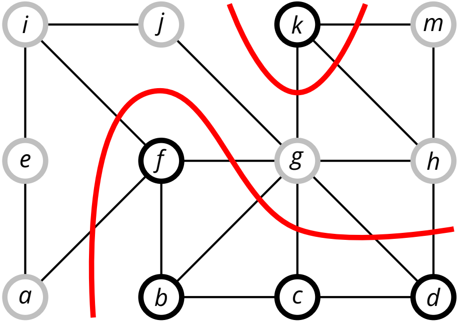


- A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a *partition* of V



- An edge $e = (u, v)$ *crosses* the cut $(S, V - S)$ if $u \in S$ and $v \in V - S$, or vice-versa

- A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a *partition* of V



- An edge $e = (u, v)$ *crosses* the cut $(S, V - S)$ if $u \in S$ and $v \in V - S$, or vice-versa

- A cut $(S, V - S)$ *respects* a set of edges A if no edge in A crosses the cut

Finding a Safe Edge

```
GENERIC-MST(G, w)
    A = ∅
    while A is not a spanning tree
        find a safe edge e = (u, v)
        A = A ∪ {e}
```

```
GENERIC-MST(G, w)
  A = ∅
  while A is not a spanning tree
      find a safe edge e = (u, v)
      A = A ∪ {e}
```

- Let $(S, V - S)$ be a cut of G that respects A

```
GENERIC-MST(G, w)
  A = ∅
  while A is not a spanning tree
      find a safe edge e = (u, v)
      A = A ∪ {e}
```

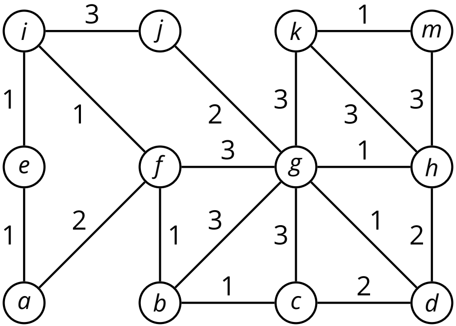
- Let $(S, V - S)$ be a cut of G that respects A
- A minimum-weight edge e that crosses $(S, V - S)$ is a safe edge

Example

- Let $(S, V - S)$ be a cut of G that respects A
- A minimum-weight edge e that crosses $(S, V - S)$ is a safe edge

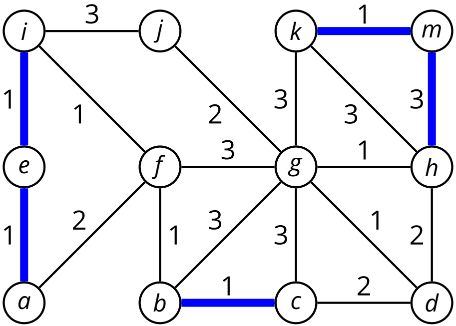
Example

- Let $(S, V - S)$ be a cut of G that respects A
- A minimum-weight edge e that crosses $(S, V - S)$ is a safe edge



Example

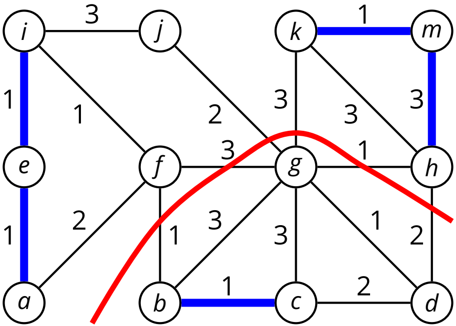
- Let $(S, V - S)$ be a cut of G that respects A
- A minimum-weight edge e that crosses $(S, V - S)$ is a safe edge



- $A = \{(a, e), (e, i), (b, c), (k, m), (m, h)\}$

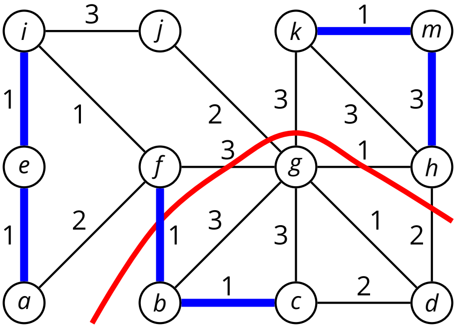
Example

- Let $(S, V - S)$ be a cut of G that respects A
- A minimum-weight edge e that crosses $(S, V - S)$ is a safe edge



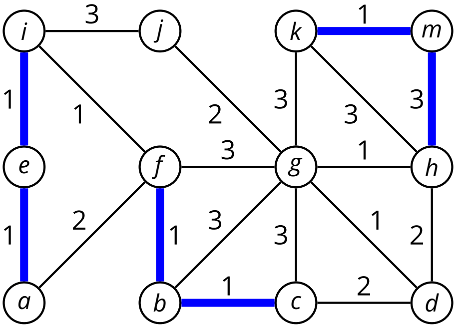
- $A = \{(a, e), (e, i), (b, c), (k, m), (m, h)\}$
- Choosing the cut $(\{a, e, f, h, i, j, k, m\}, \{b, c, d, g\})$

- Let $(S, V - S)$ be a cut of G that respects A
- A minimum-weight edge e that crosses $(S, V - S)$ is a safe edge



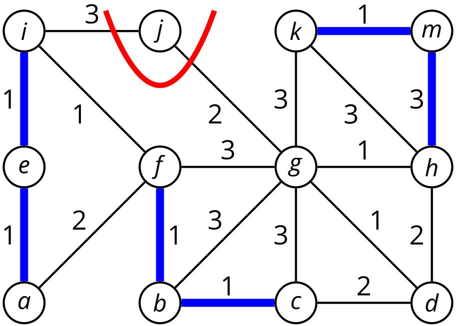
- $A = \{(a, e), (e, i), (b, c), (k, m), (m, h)\}$
- Choosing the cut $(\{a, e, f, h, i, j, k, m\}, \{b, c, d, g\})$
- (f, b) is a safe edge

- Let $(S, V - S)$ be a cut of G that respects A
- A minimum-weight edge e that crosses $(S, V - S)$ is a safe edge



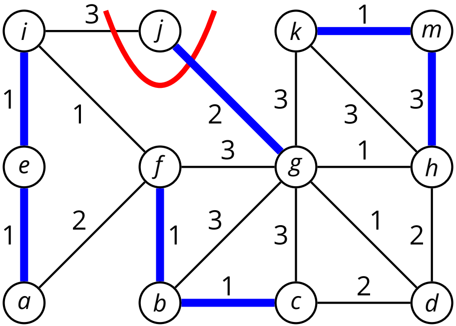
- $A = \{(a, e), (e, i), (b, c), (k, m), (m, h)\}$
- Choosing the cut $(\{a, e, f, h, i, j, k, m\}, \{b, c, d, g\})$
- (f, b) is a safe edge; (g, h) is a safe edge, too

- Let $(S, V - S)$ be a cut of G that respects A
- A minimum-weight edge e that crosses $(S, V - S)$ is a safe edge



- $A = \{(a, e), (e, i), (b, c), (k, m), (m, h)\}$
- Choosing the cut $(\{a, e, f, h, i, j, k, m\}, \{b, c, d, g\})$
- (f, b) is a safe edge; (g, h) is a safe edge, too

- Let $(S, V - S)$ be a cut of G that respects A
- A minimum-weight edge e that crosses $(S, V - S)$ is a safe edge



- $A = \{(a, e), (e, i), (b, c), (k, m), (m, h)\}$
- Choosing the cut $(\{a, e, f, h, i, j, k, m\}, \{b, c, d, g\})$
- (f, b) is a safe edge; (g, h) is a safe edge, too

Concrete Algorithms

- Kruskal's algorithm (1956)
  - based on the generic minimum-spanning-tree algorithm
  - incrementally builds a ***forest*** A

- Kruskal's algorithm (1956)
  - based on the generic minimum-spanning-tree algorithm
  - incrementally builds a ***forest*** A
- Prim's algorithm (1957)
  - based on the generic minimum-spanning-tree algorithm
  - incrementally builds a ***single tree*** A

- *Make-Set*(x) creates a set containing x

Disjoint-Set Data Structure

- *Make-Set*(x) creates a set containing x
- *Find-Set*(x) returns the *representative* of the set containing x
  - $x, y \in S \Rightarrow \textit{Find-Set}(x) = \textit{Find-Set}(y)$
  - $x \in S_1 \wedge y \in S_2 \wedge S_1 \neq S_2 \Rightarrow \textit{Find-Set}(x) \neq \textit{Find-Set}(y)$

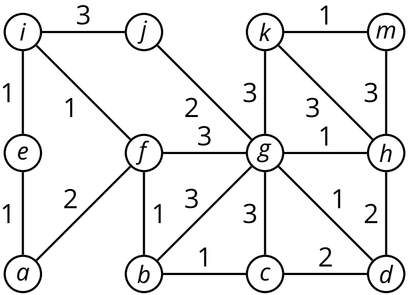
- *Make-Set*(x) creates a set containing x
- *Find-Set*(x) returns the *representative* of the set containing x
  - $x, y \in S \Rightarrow \textit{Find-Set}(x) = \textit{Find-Set}(y)$
  - $x \in S_1 \wedge y \in S_2 \wedge S_1 \neq S_2 \Rightarrow \textit{Find-Set}(x) \neq \textit{Find-Set}(y)$
- *Union*(x, y) joins the sets containing x and y

Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                    // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```

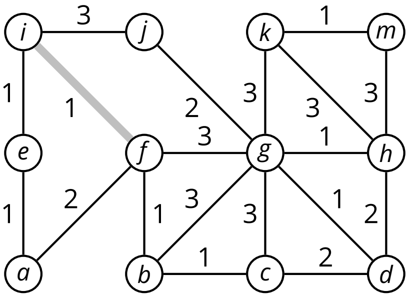
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ V(G)
        MAKE-SET(v)                                  // disjoint-set data structure
    sort E in non-decreasing order by weight w
    for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
```



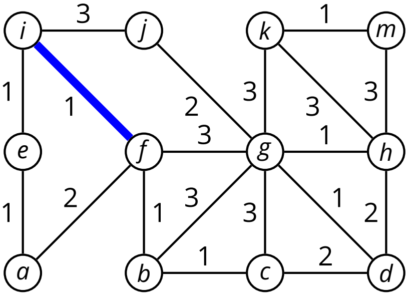
Kruskal’s Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                      // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



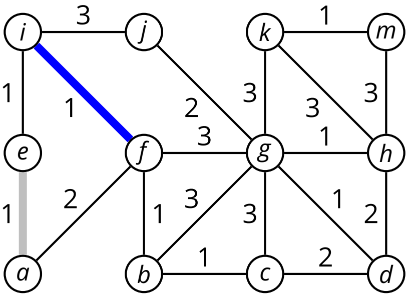
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ V(G)
        MAKE-SET(v)                          // disjoint-set data structure
    sort E in non-decreasing order by weight w
    for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
```



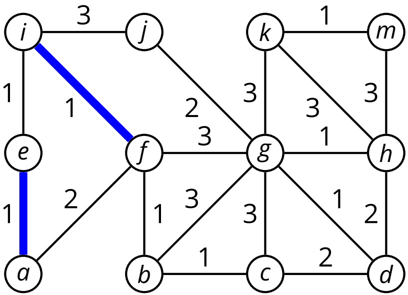
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



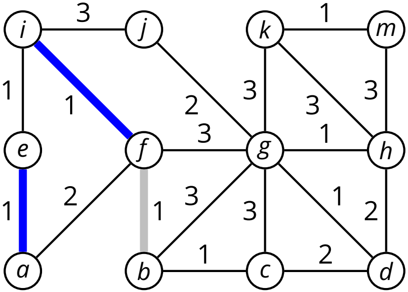
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



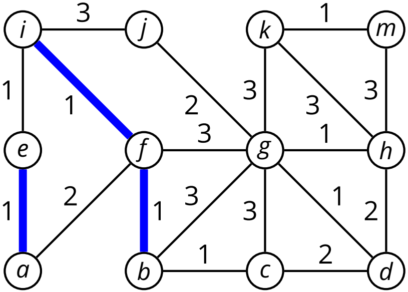
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ V(G)
        MAKE-SET(v)                          // disjoint-set data structure
    sort E in non-decreasing order by weight w
    for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
```



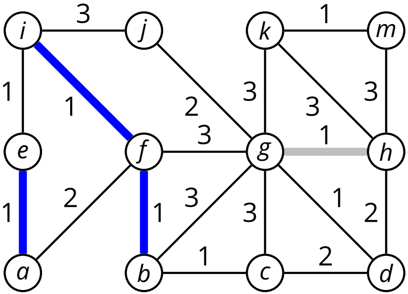
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                              // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```



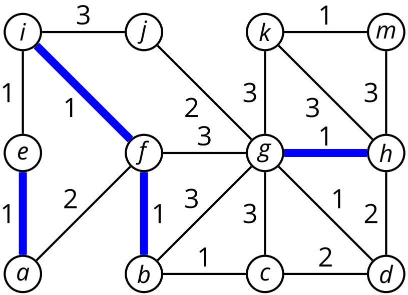
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                     // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```



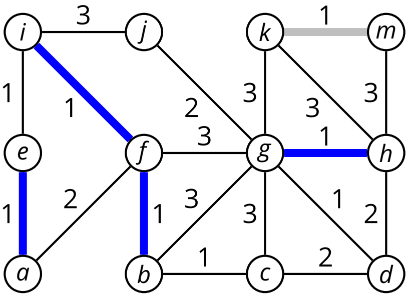
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                      // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```



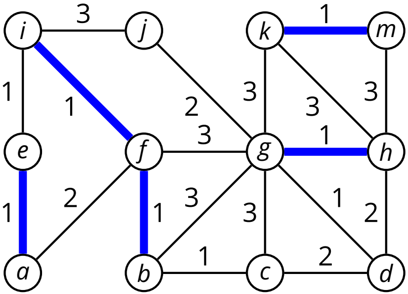
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ V(G)
        MAKE-SET(v)                         // disjoint-set data structure
    sort E in non-decreasing order by weight w
    for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
```



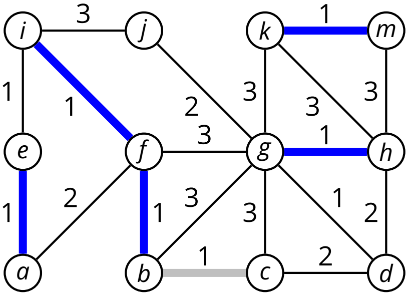
Kruskal’s Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```

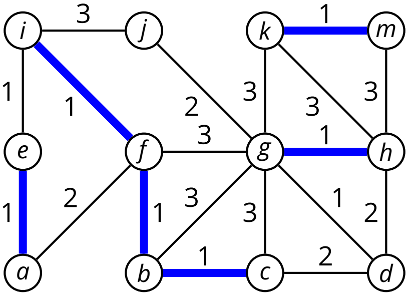


Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                                  // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```

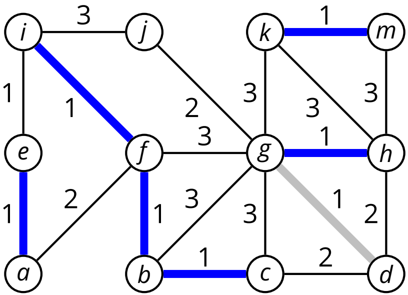


```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ V(G)
        MAKE-SET(v)                          // disjoint-set data structure
    sort E in non-decreasing order by weight w
    for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
```



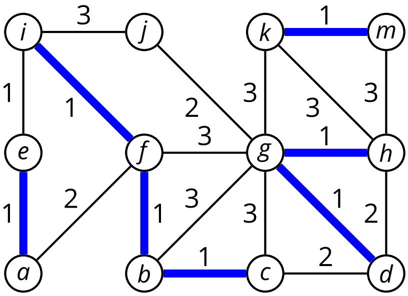
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                    // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



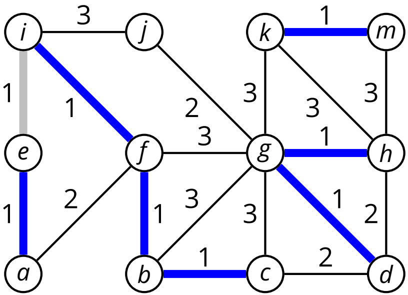
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                    // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```



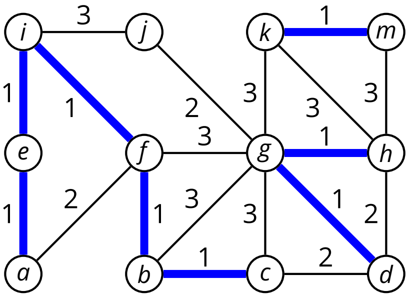
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                          // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```



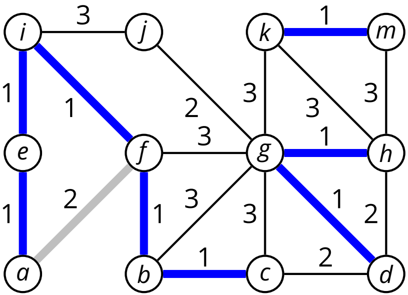
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ V(G)
        MAKE-SET(v)                          // disjoint-set data structure
    sort E in non-decreasing order by weight w
    for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
```



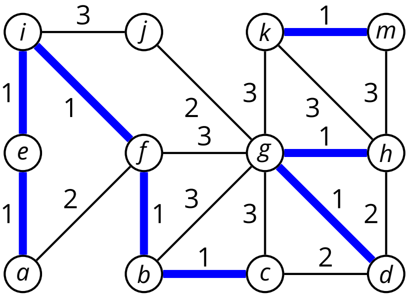
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                         // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



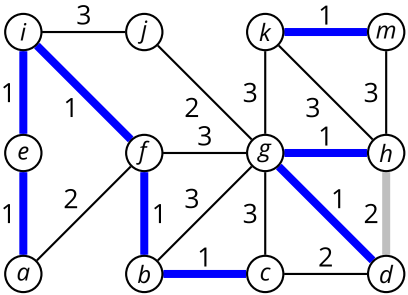
Kruskal’s Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                       // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```

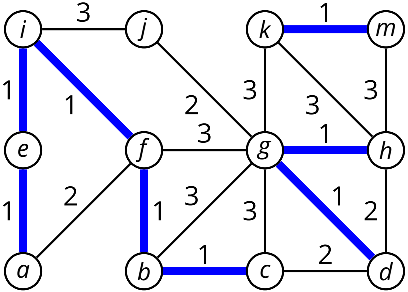


Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                         // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```

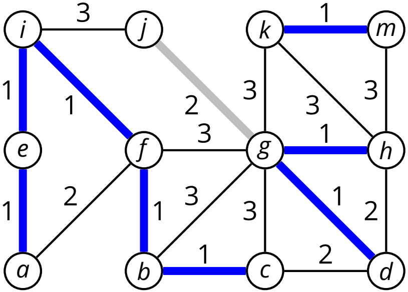


```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ V(G)
        MAKE-SET(v)                          // disjoint-set data structure
    sort E in non-decreasing order by weight w
    for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
```



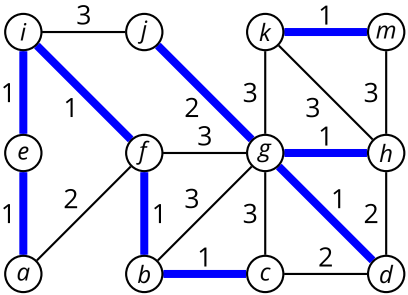
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ V(G)
        MAKE-SET(v)                      // disjoint-set data structure
    sort E in non-decreasing order by weight w
    for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
```



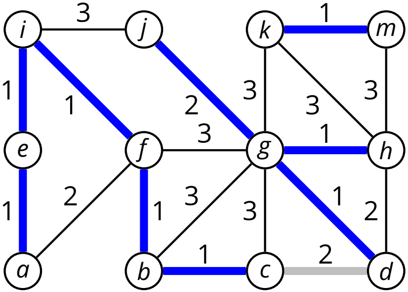
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                        // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



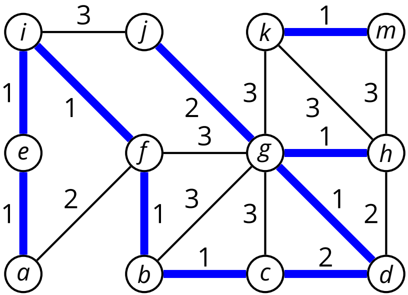
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                              // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



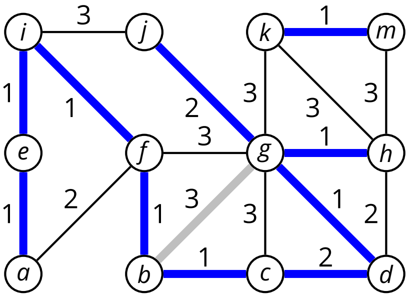
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



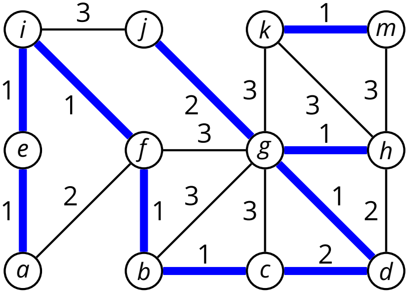
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                        // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```



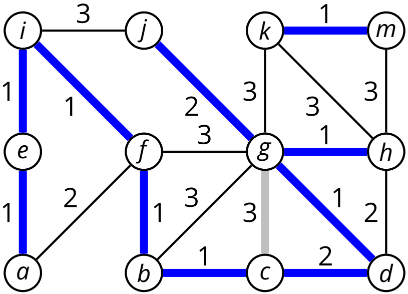
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



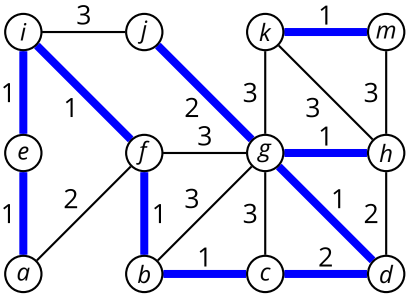
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                          // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```



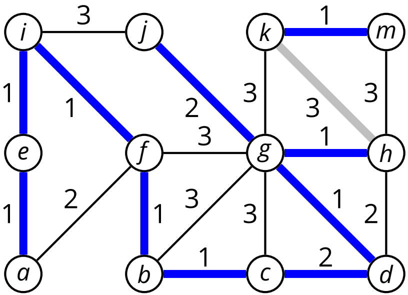
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ V(G)
        MAKE-SET(v)                              // disjoint-set data structure
    sort E in non-decreasing order by weight w
    for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
```



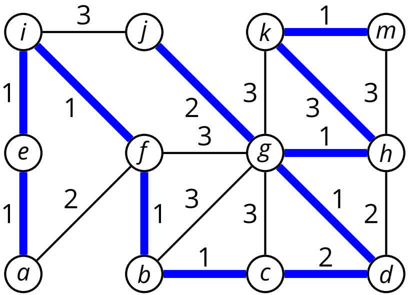
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                       // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```



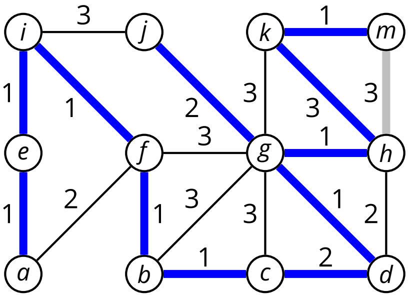
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



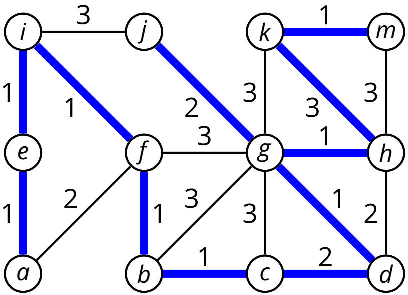
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
        MAKE-SET(v)                       // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
                A = A ∪ {(u, v)}
                UNION(u, v)
```



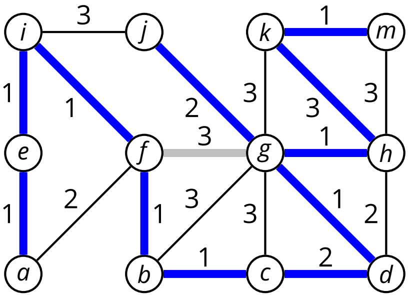
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                          // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



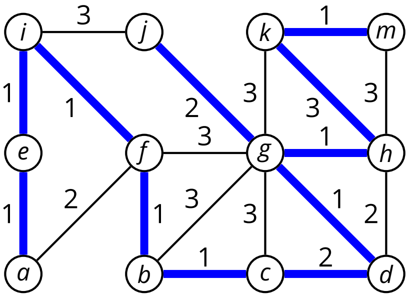
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                      // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```



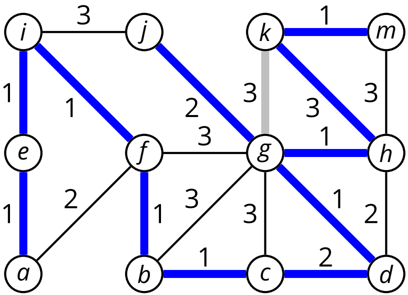
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ V(G)
        MAKE-SET(v)                    // disjoint-set data structure
    sort E in non-decreasing order by weight w
    for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
```



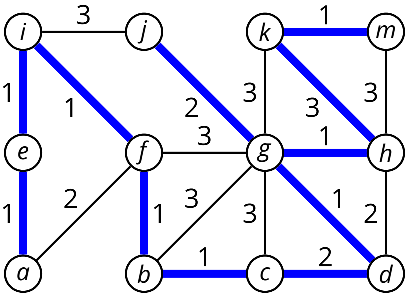
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                      // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



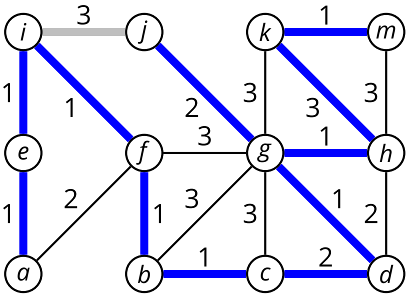
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                    // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```



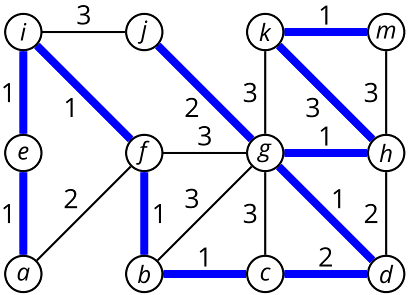
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V(G)
    MAKE-SET(v)                         // disjoint-set data structure
sort E in non-decreasing order by weight w
for each edge (u, v) ∈ E, taken in non-decreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```



Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
        MAKE-SET(v)                         // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
        if FIND-SET(u) ≠ FIND-SET(v)
              A = A ∪ {(u, v)}
              UNION(u, v)
```



```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ V(G)
      MAKE-SET(v)                         // disjoint-set data structure
  sort E in non-decreasing order by weight w
  for each edge (u, v) ∈ E, taken in non-decreasing order by w
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
```

Complexity of MST-KRUSKAL

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ V(G)
3      MAKE-SET(v)                        // disjoint-set data structure
4  sort E in non-decreasing order by weight w
5  for each edge (u, v) ∈ E, taken in non-decreasing order by w
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
```

- $|V|$ times **MAKE-SET** (loop of line 2–3)

Complexity of MST-KRUSKAL

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ V(G)
3      MAKE-SET(v)                       // disjoint-set data structure
4  sort E in non-decreasing order by weight w
5  for each edge (u, v) ∈ E, taken in non-decreasing order by w
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
```

- $|V|$ times **MAKE-SET** (loop of line 2–3)
- $O(|E| \log |E|)$ for sorting E (line 4)

Complexity of MST-KRUSKAL

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ V(G)
3      MAKE-SET(v)                    // disjoint-set data structure
4  sort E in non-decreasing order by weight w
5  for each edge (u, v) ∈ E, taken in non-decreasing order by w
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
```

- $|V|$ times **MAKE-SET** (loop of line 2–3)
- $O(|E| \log |E|)$ for sorting E (line 4)
- $2|E|$ times **FIND-SET**

Complexity of MST-KRUSKAL

```
MST-KRUSKAL(G, w)
1   A = ∅
2   for each vertex v ∈ V(G)
3       MAKE-SET(v)                     // disjoint-set data structure
4   sort E in non-decreasing order by weight w
5   for each edge (u, v) ∈ E, taken in non-decreasing order by w
6       if FIND-SET(u) ≠ FIND-SET(v)
7           A = A ∪ {(u, v)}
8           UNION(u, v)
```

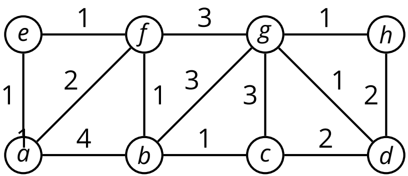
- $|V|$ times **MAKE-SET** (loop of line 2–3)
- $O(|E| \log |E|)$ for sorting E (line 4)
- $2|E|$ times **FIND-SET**
- $O(|E|)$ times **UNION**

Prim's Algorithm

```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```

Prim's Algorithm

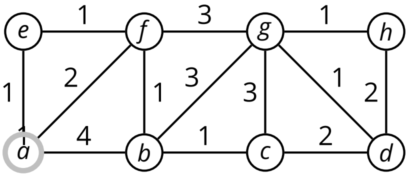
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
        key[u] = ∞
        π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
        u = EXTRACT-MIN(Q) // min by key[u]
        for each v ∈ Adj[u]
                if v ∈ Q ∧ w(u, v) < key[v]
                        π(v) = u
                        key[v] = w(u, v)
```



$$Q = \{(a, 0, \cdot), (b, \infty, \cdot), (c, \infty, \cdot), (d, \infty, \cdot), (e, \infty, \cdot), (f, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$$

Prim's Algorithm

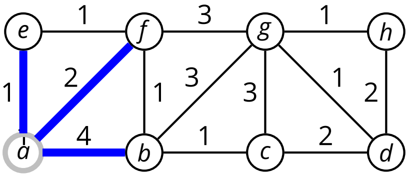
```
MST-PRIM(G, w, r)
  for each vertex u ∈ V(G)
      key[u] = ∞
      π(u) = NIL
  key[r] = 0
  Q = V(G)
  while Q ≠ ∅
      u = EXTRACT-MIN(Q) // min by key[u]
      for each v ∈ Adj[u]
          if v ∈ Q ∧ w(u, v) < key[v]
              π(v) = u
              key[v] = w(u, v)
```



$$Q = \{(b, \infty, \cdot), (c, \infty, \cdot), (d, \infty, \cdot), (e, \infty, \cdot), (f, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$$

Prim's Algorithm

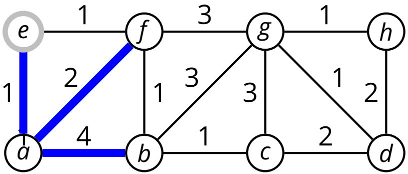
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
        key[u] = ∞
        π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
        u = EXTRACT-MIN(Q) // min by key[u]
        for each v ∈ Adj[u]
                if v ∈ Q ∧ w(u, v) < key[v]
                        π(v) = u
                        key[v] = w(u, v)
```



$Q = \{(e, 1, a), (f, 2, a), (b, 4, a)(c, \infty, \cdot), (d, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$

Prim's Algorithm

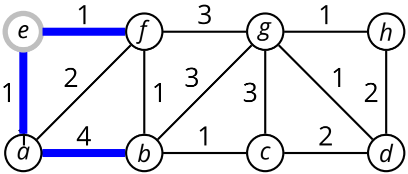
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(f, 2, a), (b, 4, a)(c, \infty, \cdot), (d, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$

Prim's Algorithm

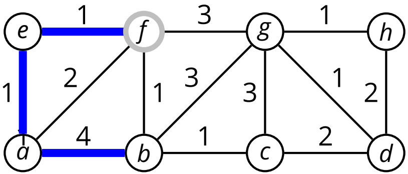
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(f, 1, e), (b, 4, a)(c, \infty, \cdot), (d, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$

Prim's Algorithm

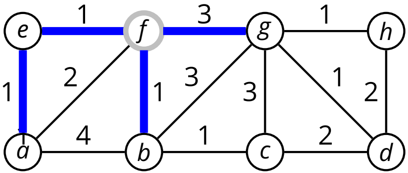
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(b, 4, a)(c, \infty, \cdot), (d, \infty, \cdot), (g, \infty, \cdot), (h, \infty, \cdot)\}$

Prim's Algorithm

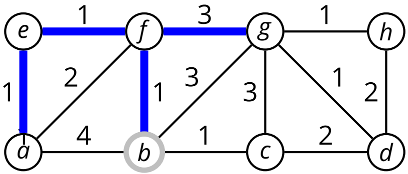
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$$Q = \{(b, 1, f), (g, 3, f), (c, \infty, \cdot), (d, \infty, \cdot), (h, \infty, \cdot)\}$$

Prim's Algorithm

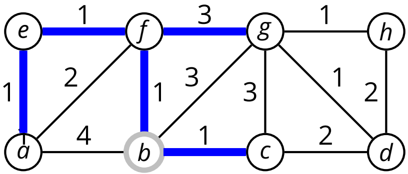
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
        key[u] = ∞
        π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
        u = EXTRACT-MIN(Q) // min by key[u]
        for each v ∈ Adj[u]
                if v ∈ Q ∧ w(u, v) < key[v]
                        π(v) = u
                        key[v] = w(u, v)
```



$Q = \{(g, 3, f), (c, \infty, \cdot), (d, \infty, \cdot), (h, \infty, \cdot)\}$

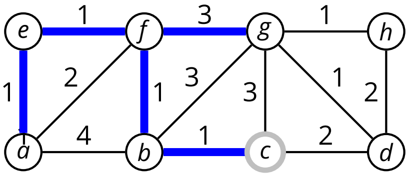
Prim's Algorithm

```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$$Q = \{(c, 1, b), (g, 3, f), (d, \infty, \cdot), (h, \infty, \cdot)\}$$

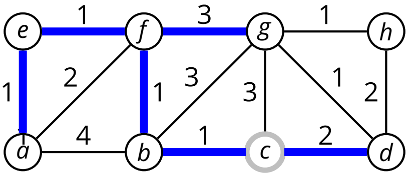
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(g, 3, f), (d, \infty, \cdot), (h, \infty, \cdot)\}$

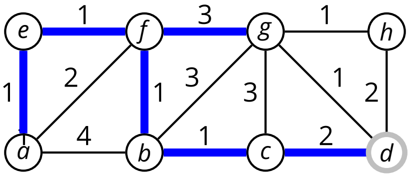
Prim's Algorithm

```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



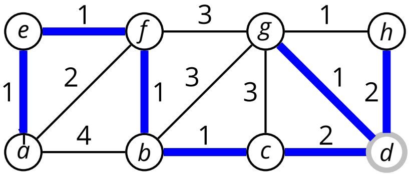
$Q = \{(d, 2, c), (g, 3, f), (h, \infty, \cdot)\}$

```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(g, 3, f), (h, \infty, \cdot)\}$

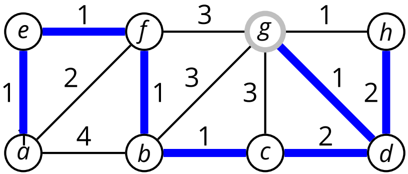
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(g, 1, d), (h, 2, d)\}$

Prim's Algorithm

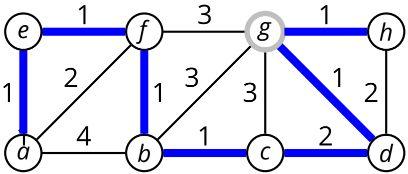
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(h, 2, d)\}$

Prim's Algorithm

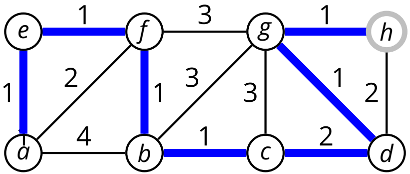
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{(h, 1, g)\}$

Prim's Algorithm

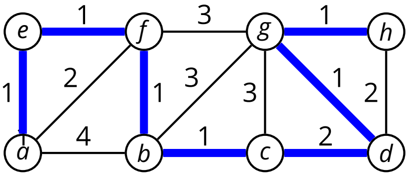
```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{\}$

Prim's Algorithm

```
MST-PRIM(G, w, r)
for each vertex u ∈ V(G)
    key[u] = ∞
    π(u) = NIL
key[r] = 0
Q = V(G)
while Q ≠ ∅
    u = EXTRACT-MIN(Q) // min by key[u]
    for each v ∈ Adj[u]
        if v ∈ Q ∧ w(u, v) < key[v]
            π(v) = u
            key[v] = w(u, v)
```



$Q = \{\}$