Object-oriented programming and data-structures

CS/ENGRD 2110
SUMMER 2018

Lecture 14: Spanning Trees

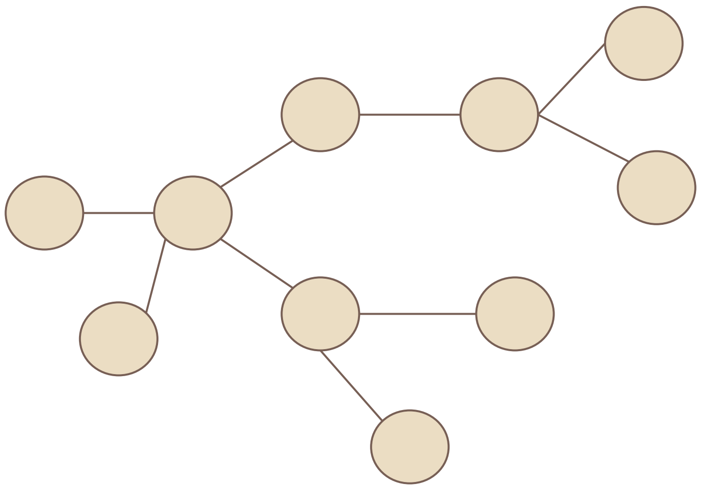
http://courses.cs.cornell.edu/cs2110/2018su

Graph Algorithms

- Search
  - Depth-first search
  - Breadth-first search
- Shortest paths
  - Dijkstra's algorithm
- Spanning trees
  - Prim's algorithm
  - Kruskal's algorithm

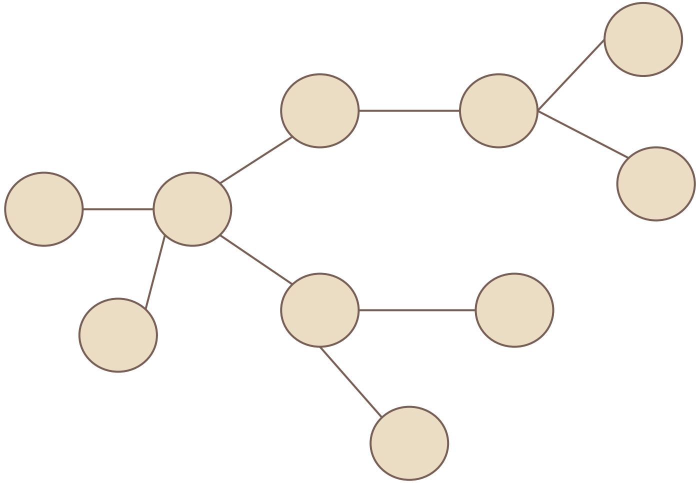
Recall: Trees

- A undirected graph is a **tree** if there is exactly one **simple path** between any pair of vertices.



Recall: Trees

- A undirected graph is a **tree** if there is exactly one **simple path** between any pair of vertices.



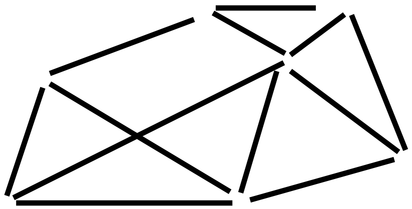
What's the root? It doesn't matter. Any vertex can be root

Facts about trees

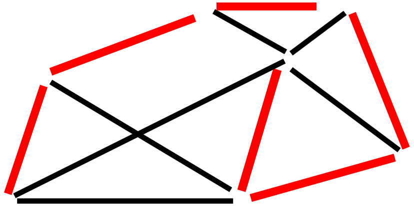
- A tree must necessarily be:
  - Connected
    - A graph is **connected** when there is a path between every pair of vertices
  - #E = #V - 1
  - No cycles

Spanning Trees

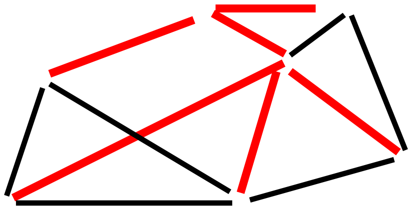
- A spanning tree of a **connected** undirected graph (V,E) is a subgraph (V,E’) that is a tree



- Same set of vertices V
- $E' \subseteq E$
- (V, E') is a tree



- Same set of vertices V
- Maximal set of edges that contains no cycle



- Same set of vertices V
- Minimal set of edges that connect all vertices

Three equivalent definitions

Applications of spanning trees

- Spanning trees represent the minimum set of edges such that all the nodes in the graph are connected

  - Useful for telecommunication applications!
    - How can I connect everyone in my business using the fewest cables

  - Useful for wiring on chips
    - How can I arrange my components such that they can all talk to each other with the fewest cables.

Finding a spanning tree (V1)

- Recall
  - Same set of vertices V
  - Maximal set of edges that contains no cycle

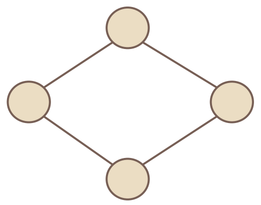
- Define an iterative algorithm that, when discovering a cycle in the graph, removes an edge from that cycle, until no cycles exist.

Finding a spanning tree (V1)

- Recall
  - Same set of vertices V
  - Maximal set of edges that contains no cycle

- Define an iterative algorithm that, when discovering a cycle in the graph, removes an edge from that cycle, until no cycles exist.

Start with the whole graph – it is connected
- While there is a cycle:
  Pick an edge of a cycle and throw it out
  – the graph is still connected (why?)

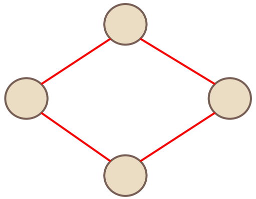


Finding a spanning tree (V1)

- Recall
  - Same set of vertices V
  - Maximal set of edges that contains no cycle

- Define an iterative algorithm that, when discovering a cycle in the graph, removes an edge from that cycle, until no cycles exist.

Start with the whole graph – it is connected
- While there is a cycle:
  Pick an edge of a cycle and throw it out
  – the graph is still connected (why?)



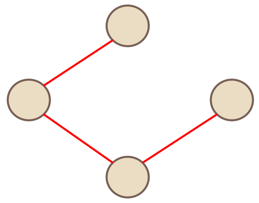
Finding a spanning tree (V1)

- Recall

  - Same set of vertices V
  - Maximal set of edges that contains no cycle

- Define an iterative algorithm that, when discovering a cycle in the graph, removes an edge from that cycle, until no cycles exist.

Start with the whole graph – it is connected
- While there is a cycle:
  Pick an edge of a cycle and throw it out
  – the graph is still connected (why?)



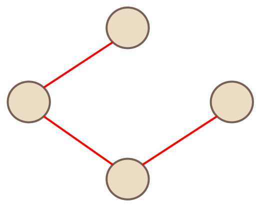
Finding a spanning tree (V1)

- Recall

  - Same set of vertices V
  - Maximal set of edges that contains no cycle

- Define an iterative algorithm that, when discovering a cycle in the graph, removes an edge from that cycle, until no cycles exist.

Start with the whole graph – it is connected
- While there is a cycle:
  Pick an edge of a cycle and throw it out
  – the graph is still connected (why?)



Could have removed a different edge. There can be multiple spanning trees!

Finding a spanning tree (V2)

- Recall
  - Same set of vertices V
  - Minimal set of edges that connect all vertices

- Define a set **A** that maintains following invariant:
  - A is a subset of some spanning tree (nodes in A are connected)

- At each step, determine an edge (u,v) that can add to A without violating invariant
  - A U {(u,v)} is also a subset of a spanning tree
  - Call this edge a **safe edge**

Finding a spanning tree (V2)

- Recall

  - Same set of vertices V
  - Minimal set of edges that connect all vertices

```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```

Finding a spanning tree (V2)

- Recall
  - Same set of vertices V
  - Minimal set of edges that connect all vertices

```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```

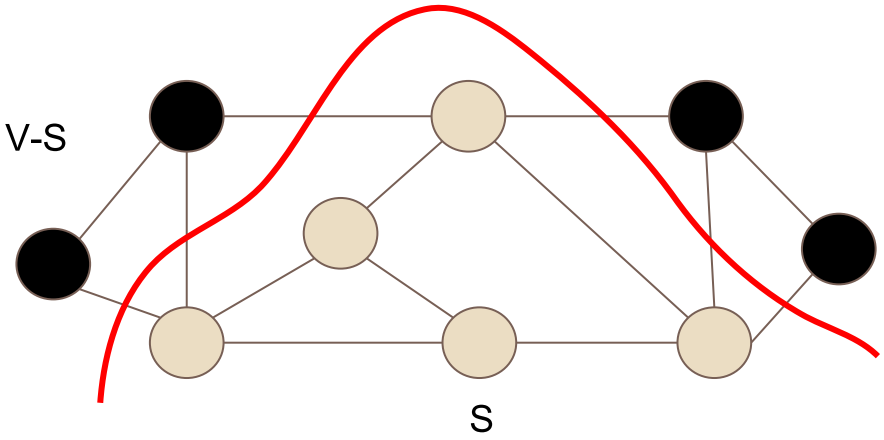
But how to determine what a **safe edge** is?
(One must exist by our loop invariant: A is a subset of a spanning tree T)

Definition: Cuts

- A **cut** (S,V-S) of an undirected graph G = (V,E) is a partition of V.
- We say that an edge (u,v) ∈ **crosses** the cut (S,V-S) if one of its endpoints is in S and the other is in V-S
- A cut **respects** a set A of edges if no edge in A crosses the cut

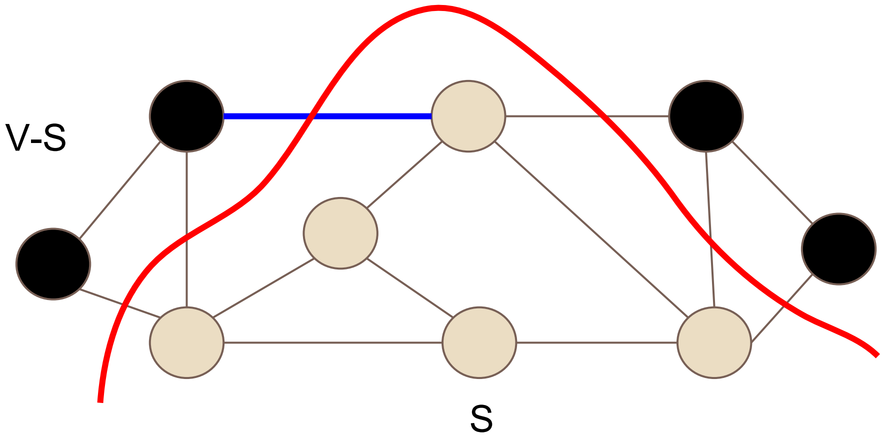
Definition: Cuts

- A **cut** (S,V-S) of an undirected graph G = (V,E) is a partition of V.
- We say that an edge (u,v) ∈ **crosses** the cut (S,V-S) if one of its endpoints is in S and the other is in V-S
- A cut **respects** a set A of edges if no edge in A crosses the cut



Definition: Cuts

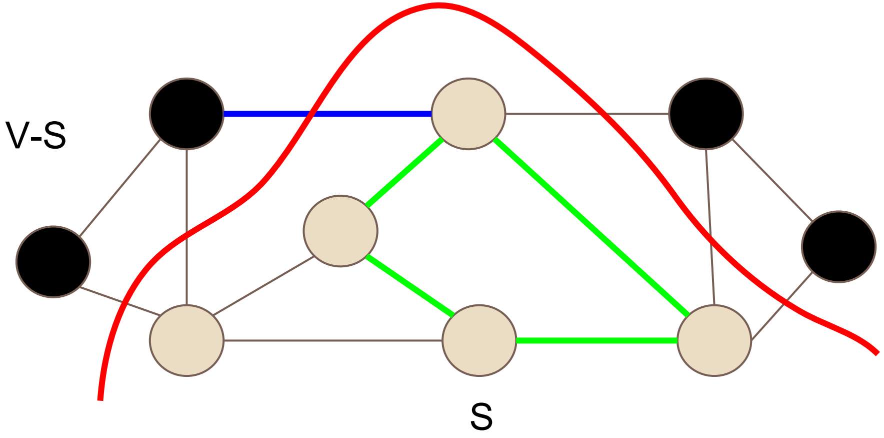
- A **cut** (S,V-S) of an undirected graph G = (V,E) is a partition of V.
- We say that an edge (u,v) ∈ **crosses** the cut (S,V-S) if one of its endpoints is in S and the other is in V-S
- A cut **respects** a set A of edges if no edge in A crosses the cut



Blue edge crosses the cut as it connects a black node to a beige node

Definition: Cuts

- A **cut** (S,V-S) of an undirected graph G = (V,E) is a partition of V.
- We say that an edge (u,v) ∈ **crosses** the cut (S,V-S) if one of its endpoints is in S and the other is in V-S
- A cut **respects** a set A of edges if no edge in A crosses the cut



Blue edge crosses the cut as it connects a black node to a beige node

Cut respects the set A of green edges.

Finding a spanning tree (V2)

- Recall

  - Same set of vertices V
  - Minimal set of edges that connect all vertices

```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```

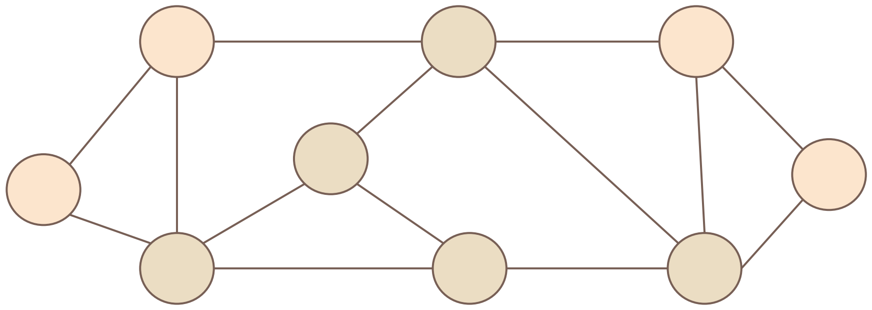
Let G = (V,E) be a connected, undirected graph. Let A be a subset of E that is included in some spanning tree for G. Let (S,V-S) be any cut of G that **respects** A, and let (u,v) be an **edge crossing** (S,V-S), then edge (u,v) is **safe** for A

Finding a spanning tree (V2)

- Recall

> • Same set of vertices V
>
> • Minimal set of edges that connect all vertices

```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```

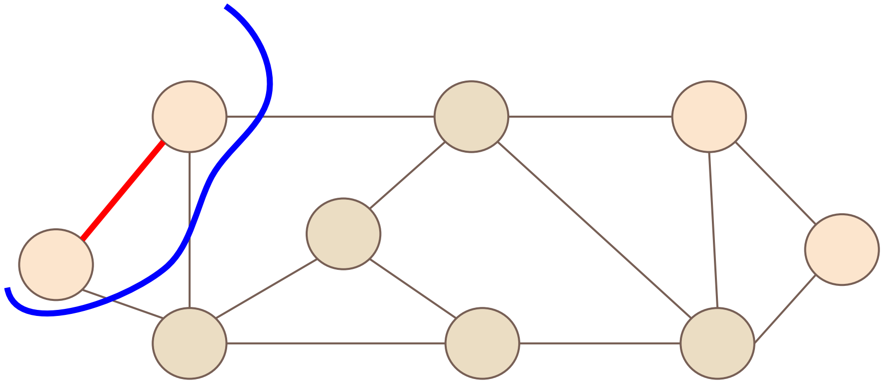


Finding a spanning tree (V2)

- Recall

> - Same set of vertices V
> - Minimal set of edges that connect all vertices

```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```

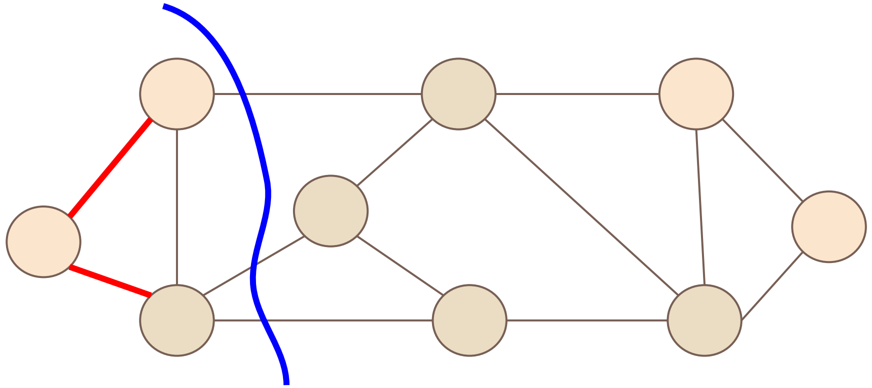


Finding a spanning tree (V2)

- Recall

> - Same set of vertices V
> - Minimal set of edges that connect all vertices

```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```

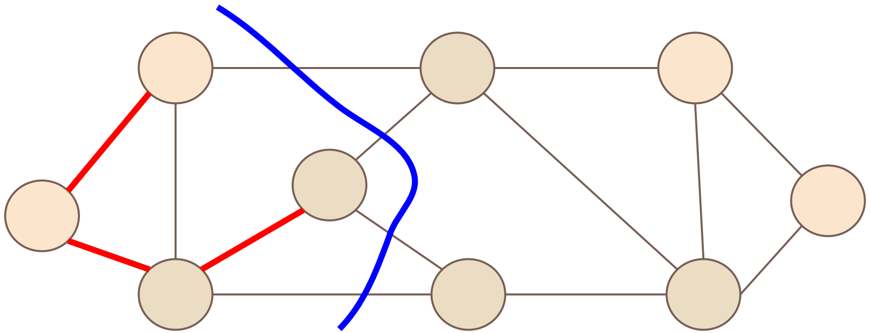


Finding a spanning tree (V2)

- Recall

> - Same set of vertices V
> - Minimal set of edges that connect all vertices

```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```

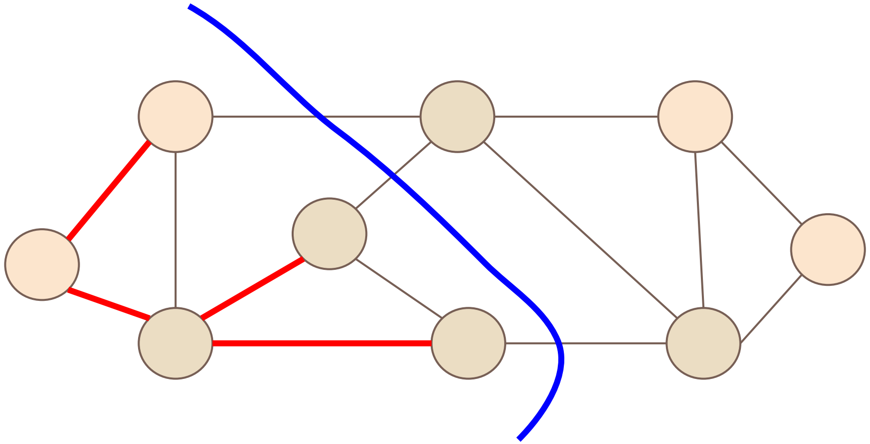


Finding a spanning tree (V2)

- Recall

> - Same set of vertices V
> - Minimal set of edges that connect all vertices

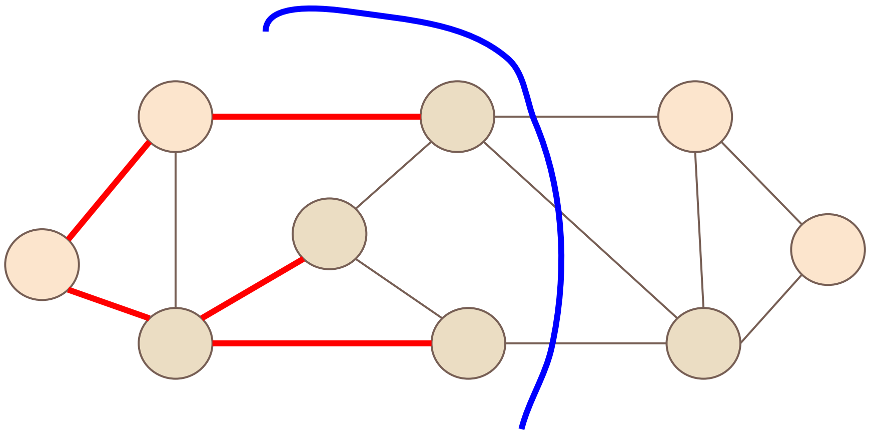
```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```



Finding a spanning tree (V2)

- Recall
  - Same set of vertices V
  - Minimal set of edges that connect all vertices

```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```

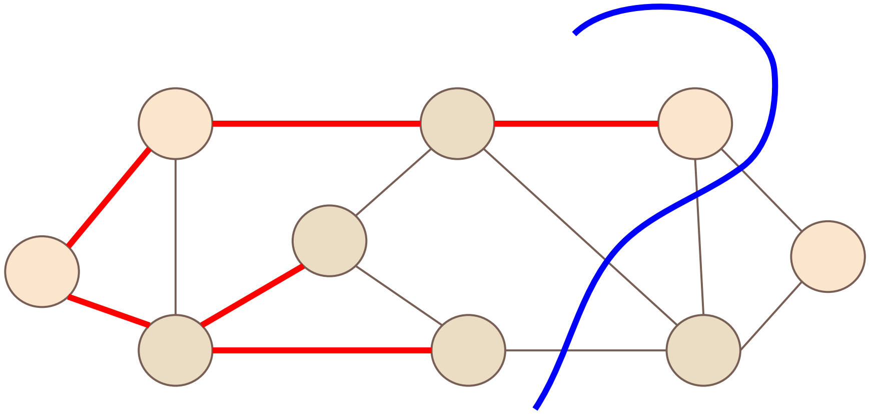


Finding a spanning tree (V2)

- Recall

> - Same set of vertices V
> - Minimal set of edges that connect all vertices

```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```

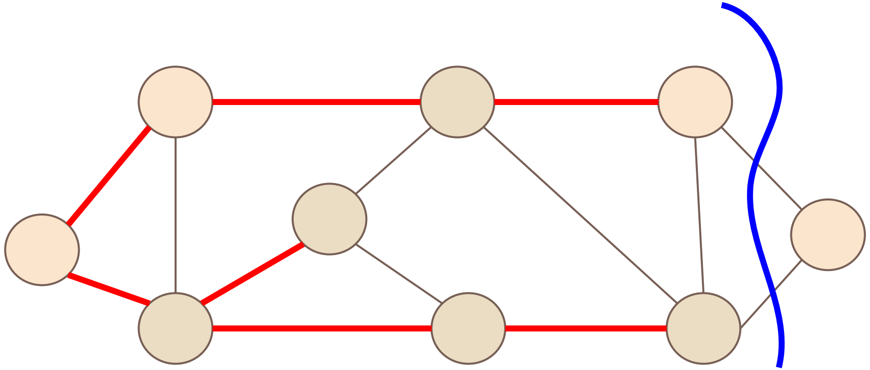


Finding a spanning tree (V2)

- Recall

> - Same set of vertices V
> - Minimal set of edges that connect all vertices

```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```

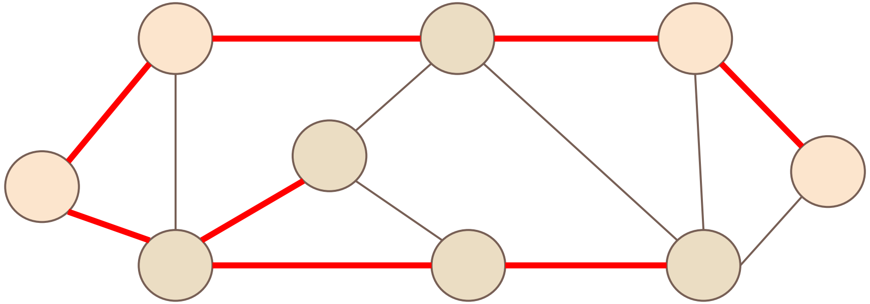


Finding a spanning tree (V2)

- Recall

> - Same set of vertices V
> - Minimal set of edges that connect all vertices

```
A = ∅
// Inv: A is a subset of a spanning tree T
While A does not form a spanning tree
        Find an edge (u,v) that is safe for A
        A = A U {(u,v)}
return A
```



Minimum Spanning Tree

- In a **weighted** graph, want to find the **minimum spanning tree**
  - (Recall that there can be multiple spanning trees)

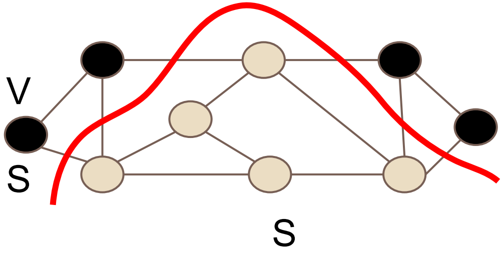
- Want to find the spanning tree with the **minimum weight**

- Formally: finding the minimum spanning tree for a graph is finding the spanning tree whose weight **w(T) is minimised.**

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

Definition: Cuts

- A **cut** (S,V-S) of an undirected graph G = (V,E) is a partition of V.
- We say that an edge (u,v) ∈ **crosses** the cut (S,V-S) if one of its endpoints is in S and the other is in V-S
- A cut **respects** a set A of edges if no edge in A crosses the cut
- An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut



Algorithms of Kruskal and Prim

- **Greedy** algorithms that use a specific rule to determine a **safe edge**

  - Kruskal's algorithm
    - The **set A is a forest** whose vertices are all those of the given graph
    - The same edge added to A is always a least-weight edge in the graph that connects two distinct components

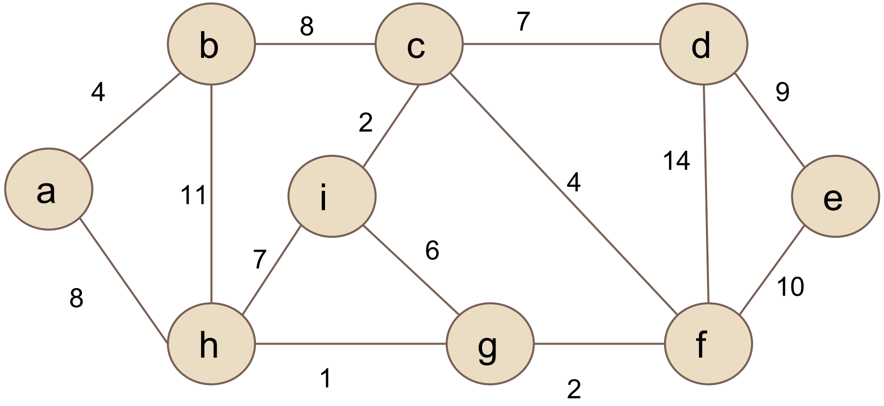
  - Prim's algorithm
    - The **set A forms a single tree.**
    - The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree

Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components

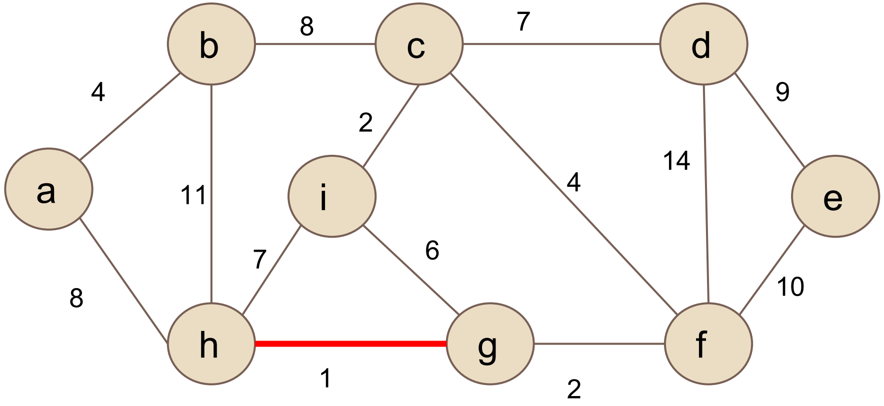
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



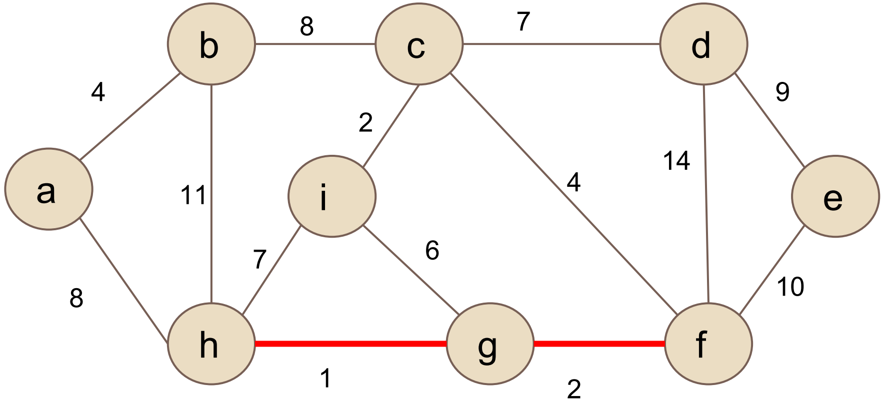
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



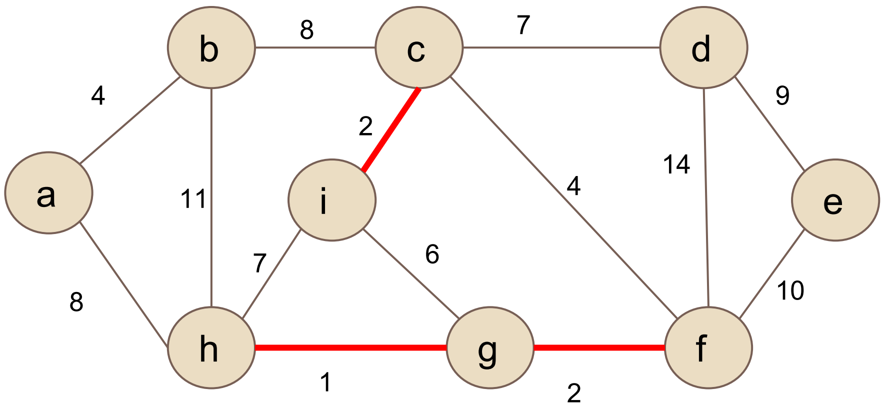
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



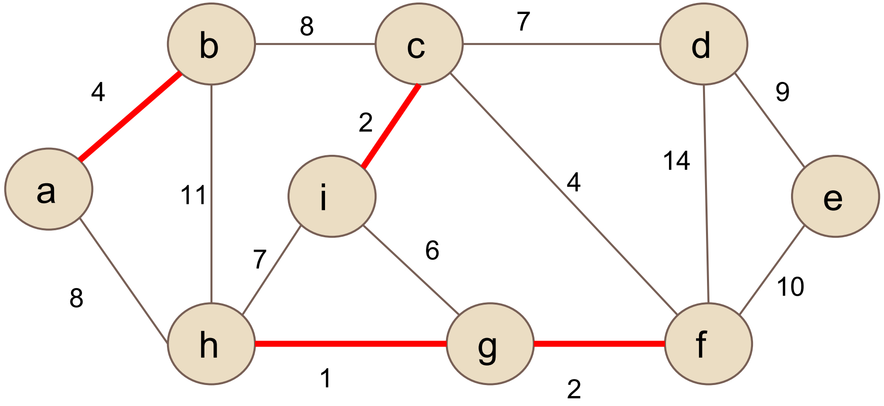
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



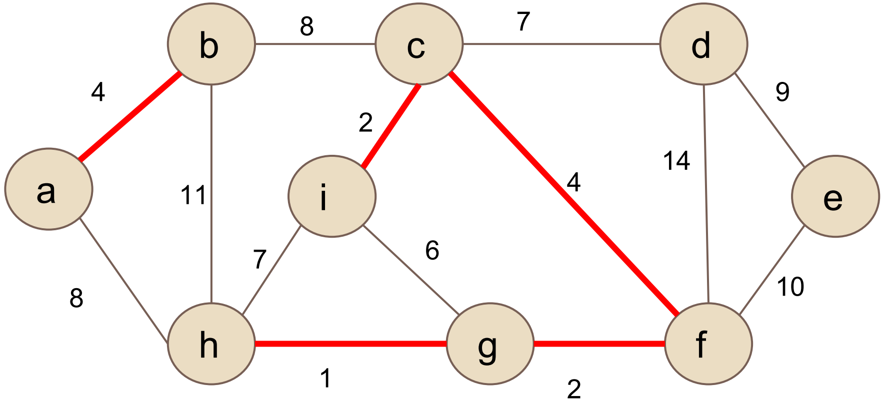
Kruskal’s Algorithm

- Kruskal’s algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



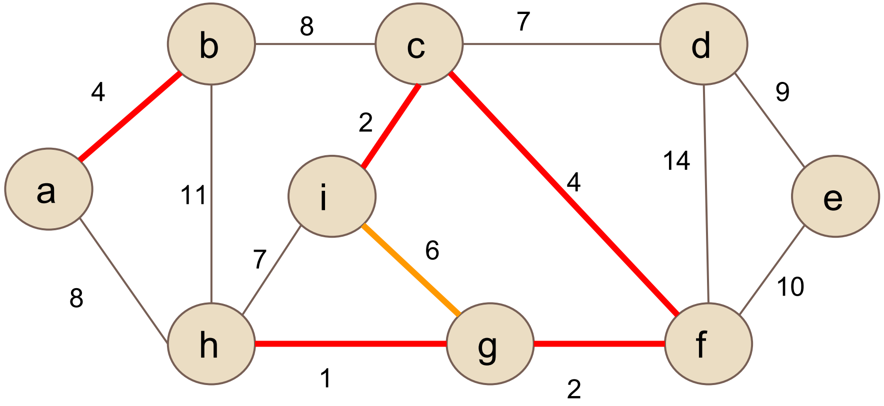
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



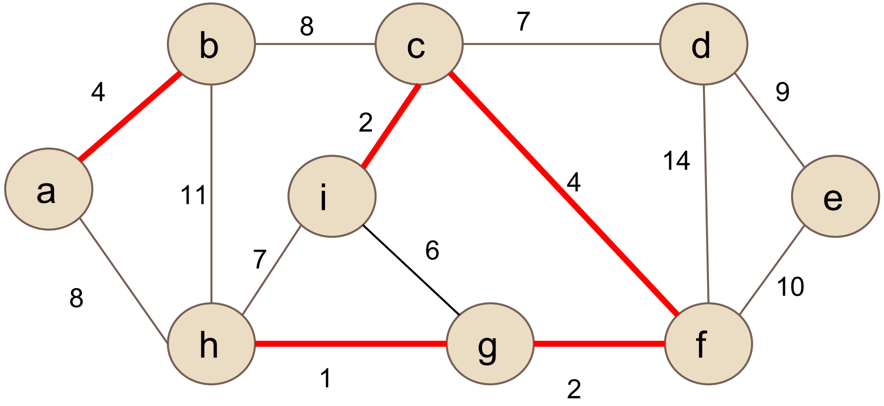
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



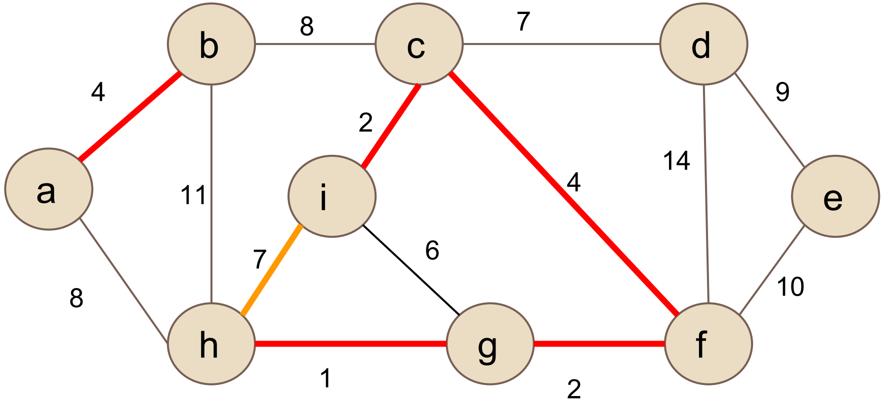
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



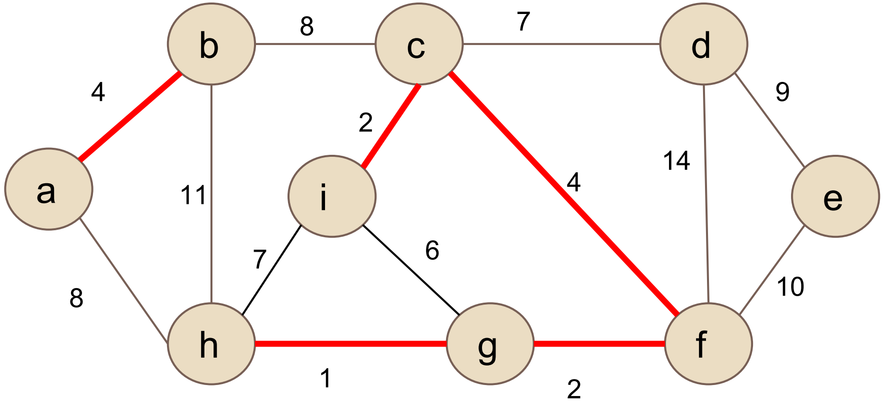
Kruskal’s Algorithm

- Kruskal’s algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



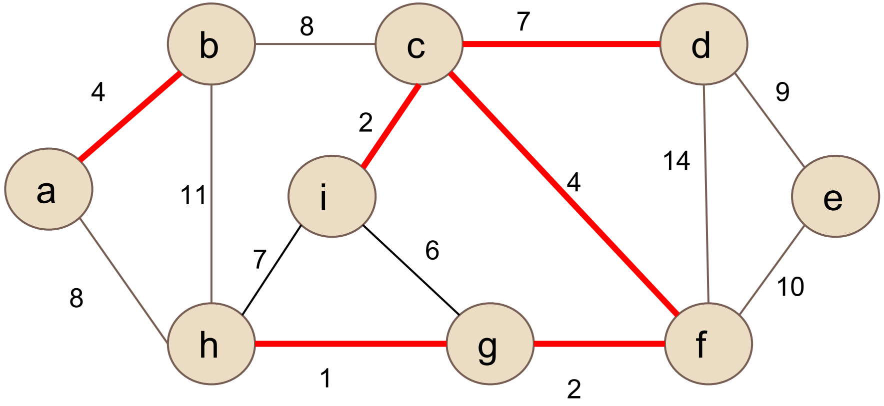
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



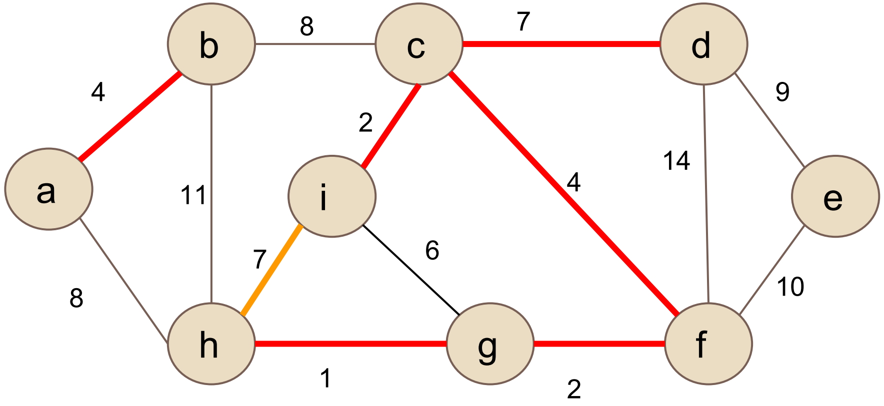
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



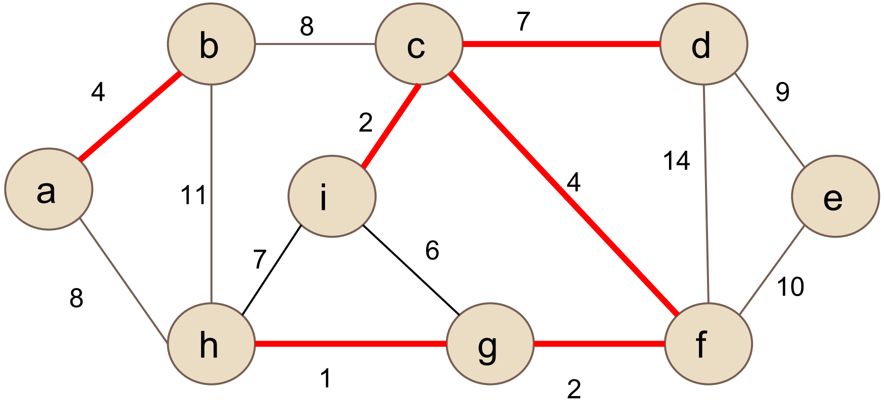
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



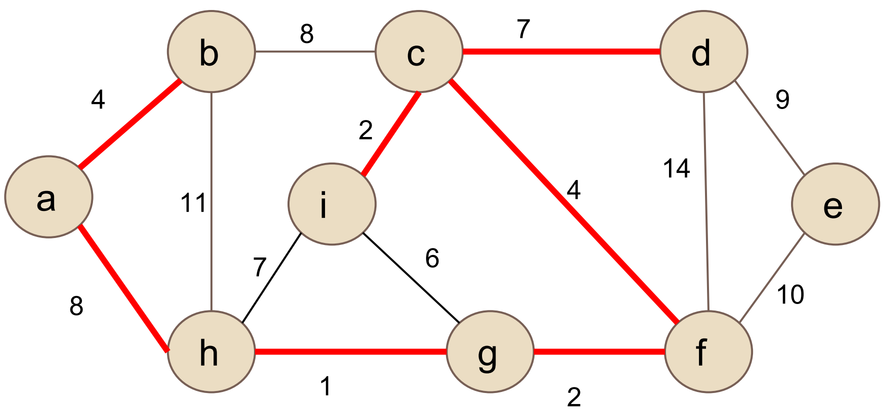
Kruskal’s Algorithm

- Kruskal’s algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



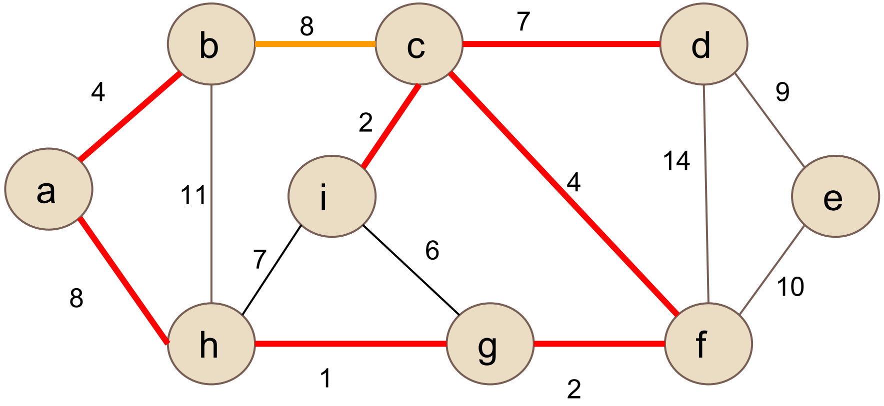
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



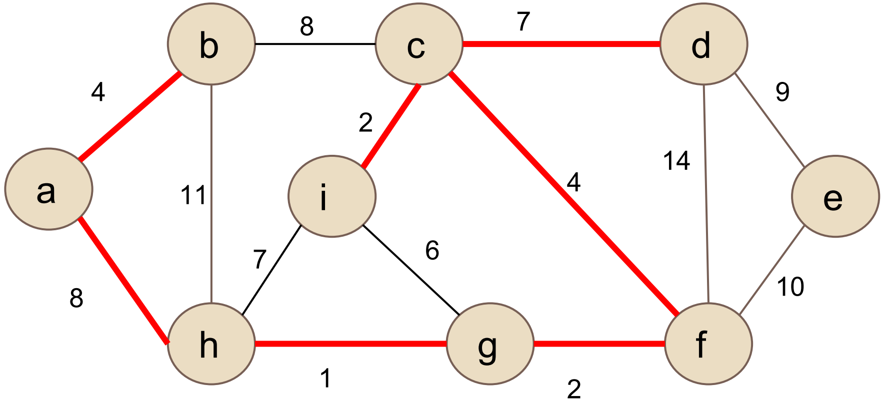
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



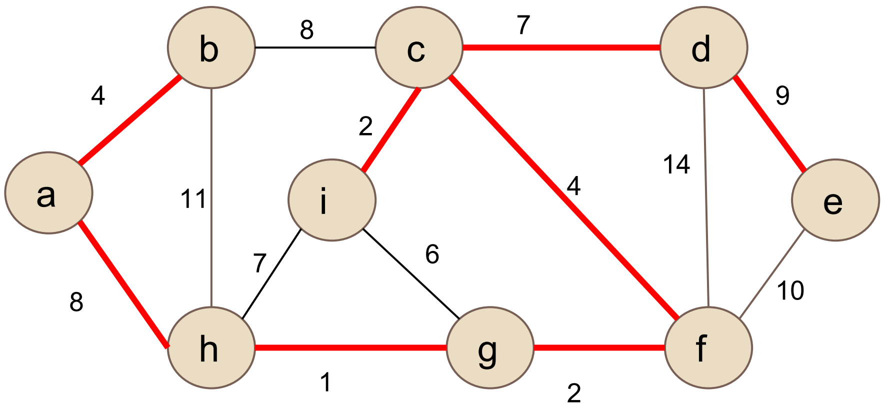
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



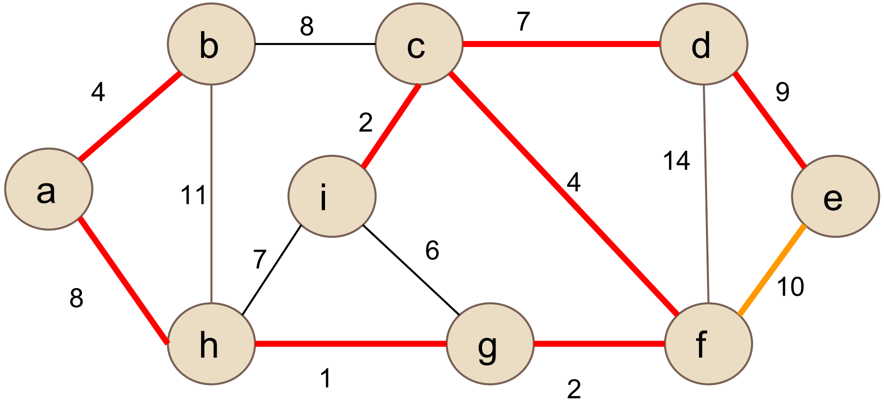
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



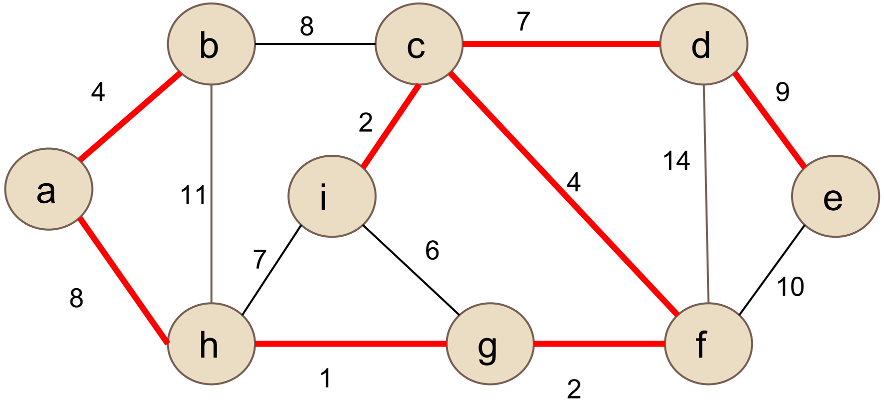
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



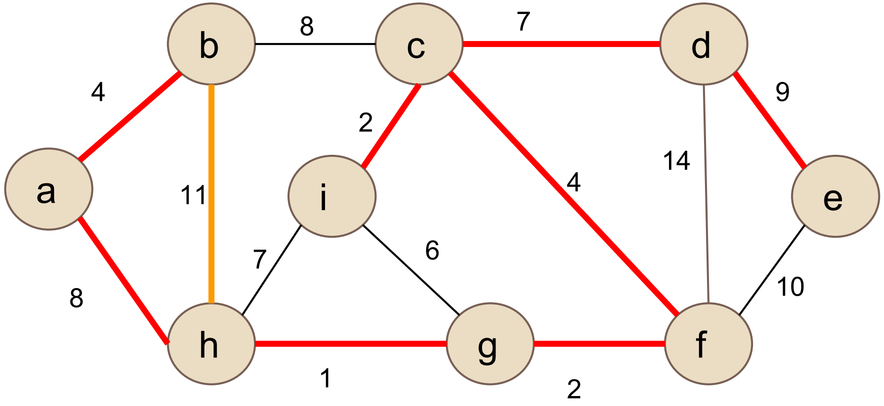
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



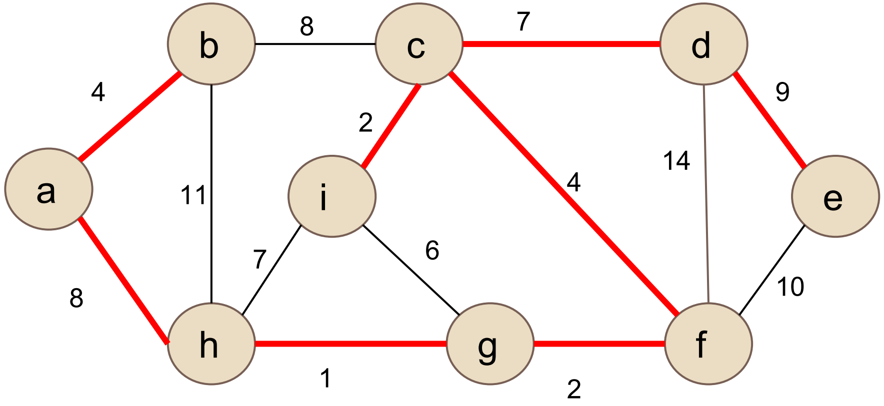
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



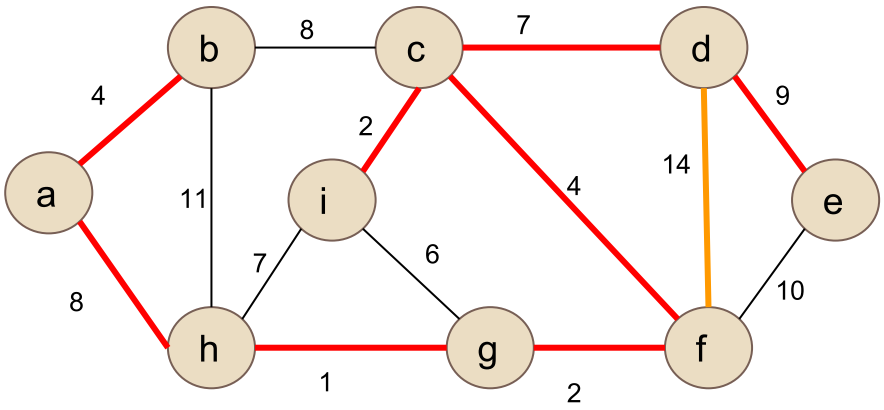
Kruskal’s Algorithm

- Kruskal’s algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



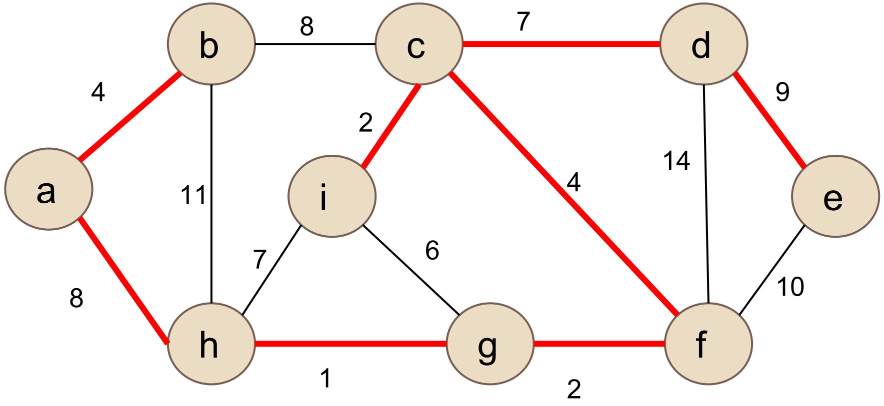
Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



Kruskal's Algorithm

- Kruskal's algorithm
  - The **set A is a forest** whose vertices are all those of the given graph
  - The same edge added to A is always a least-weight edge in the graph that connects two distinct components



Disjoint-Set Datastructures

- An easy way to express Kruskal's algorithm is in terms of **disjoint-set data structure**
- A **disjoint set data structure** maintains a collection $S=\{S_1, S_2, \ldots, S_3\}$ of **disjoint sets**
- Each set is identified by a **representative**, which is some member in the set
  - Some applications care which member we choose, others don't.
- Disjoint set data structures define three operations
  - Make-Set(x)
  - Union(x,y)
  - Find-Set(x)

Disjoint-Set Datastructures

- Disjoint set data structures define three operations
  - **Make-Set(x)**
    - Creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set
  - **Union(x,y)**
    - Merges the sets that contain x and y (S_x and S_y) into a new set that is the union of these two sets. The new representative of this set is either the representative of x, or of y.
  - **Find-Set(x)**
    - Returns a reference to the representative of the (unique) set containing x

Kruskal's Algorithm

```
A = ∅

For each vertex v in G.V:
        Make-Set(v)
// Inv: A is a subset of the minimum spanning tree
Sort the edges of G.E into increasing order by weight w
For each edge (u,v) in G.E, taken in increasing order by weight w:
        If FIND-SET(u) ≠ FIND-SET(v)
                A = A U{(u,v)}
                UNION(u,v)
Return A
```

Kruskal’s Algorithm

```
A = ∅

For each vertex v in G.V:
        Make-Set(v)
// Inv: A is a subset of the minimum spanning tree
Sort the edges of G.E into increasing order by weight w
For each edge (u,v) in G.E, taken in increasing order by weight w:
        If FIND-SET(u) ≠ FIND-SET(v)
                A = A U{(u,v)}
                UNION(u,v)
Return A
```

Initialises set A to the empty set and creates |V| trees, one containing each vertex

Kruskal's Algorithm

```
A = ∅

For each vertex v in G.V:
        Make-Set(v)
// Inv: A is a subset of the minimum spanning tree
Sort the edges of G.E into increasing order by weight w
For each edge (u,v) in G.E, taken in increasing order by weight w:
        If FIND-SET(u) ≠ FIND-SET(v)
                A = A U{(u,v)}
                UNION(u,v)
Return A
```

Initialises set A to the empty set and creates |V| trees, one containing each vertex

Checks, for each edge (u,) whether the endpoints u and v belong to the same tree already. If they do, then the edge (u,v) cannot be added to the forest without creating a cycle, and the edge is discarded. Otherwise, the two vertices belong to different trees.

In this case, adds edge into (u,v)

Kruskal's Algorithm - Complexity

```
A = ∅

For each vertex v in G.V:
        Make-Set(v)
// Inv: A is a subset of the minimum spanning tree
Sort the edges of G.E into increasing order by weight w
For each edge (u,v) in G.E, taken in increasing order by weight w:
        If FIND-SET(u) ≠ FIND-SET(v)
                A = A U{(u,v)}
                UNION(u,v)
Return A
```

Kruskal's Algorithm - Complexity

```
A = ∅

For each vertex v in G.V:
        Make-Set(v)
// Inv: A is a subset of the minimum spanning tree
Sort the edges of G.E into increasing order by weight w
For each edge (u,v) in G.E, taken in increasing order by weight w:
        If FIND-SET(u) ≠ FIND-SET(v)
                A = A U{(u,v)}
                UNION(u,v)
Return A
```

|V| * Make-Set (V)

Kruskal's Algorithm - Complexity

```
A = ∅

For each vertex v in G.V:
        Make-Set(v)
// Inv: A is a subset of the minimum spanning tree
Sort the edges of G.E into increasing order by weight w
For each edge (u,v) in G.E, taken in increasing order by weight w:
        If FIND-SET(u) ≠ FIND-SET(v)
                A = A U{(u,v)}
                UNION(u,v)
Return A
```

|V| * Make-Set (V)

O(E * log E)

Kruskal's Algorithm - Complexity

```
A = ∅

For each vertex v in G.V:
        Make-Set(v)
// Inv: A is a subset of the minimum spanning tree
Sort the edges of G.E into increasing order by weight w
For each edge (u,v) in G.E, taken in increasing order by weight w:
        If FIND-SET(u) ≠ FIND-SET(v)
                A = A U{(u,v)}
                UNION(u,v)
Return A
```

|V| * Make-Set (V)

O(E * log E)

|E| * (Find-Set + Union)

Kruskal's Algorithm - Complexity

```
A = ∅

For each vertex v in G.V:
        Make-Set(v)
// Inv: A is a subset of the minimum spanning tree
Sort the edges of G.E into increasing order by weight w
For each edge (u,v) in G.E, taken in increasing order by weight w:
        If FIND-SET(u) ≠ FIND-SET(v)
                A = A U{(u,v)}
                UNION(u,v)
Return A
```

|V| * Make-Set (V)

O(E * log E)

|E| * (Find-Set + Union)

With the right disjoint-set datastructure, end up with O(E log V)

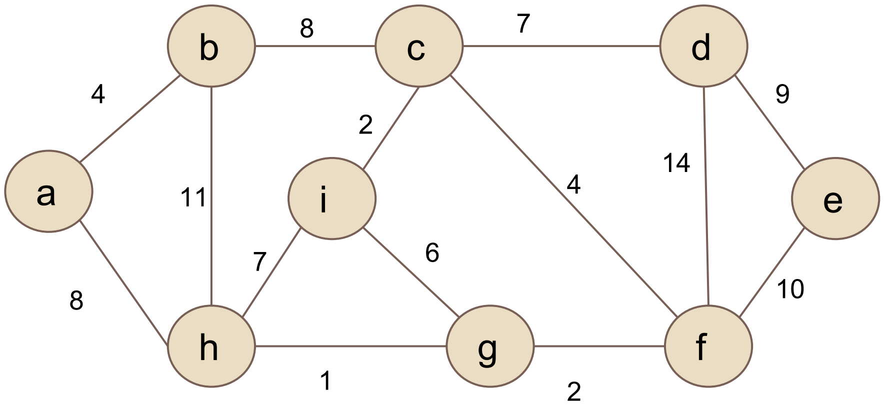
Prim's algorithm

- Prim's algorithm
  - The **set A forms a single tree**
  - The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree
  - Algorithm starts from an arbitrary root vertex r and grows until tree spans all vertices in V
  - Each step adds to the tree A a **light edge** that connects A to an isolated vertex (one on which no edge of A is incident)

Prim's algorithm

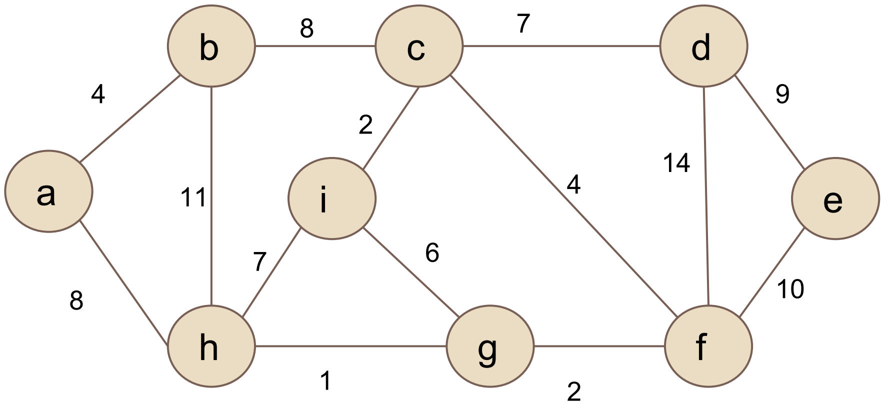
- Prim's algorithm
  - All vertices that are not in the tree reside in a **min-priority queue Q** based on a key attribute v.key
    - v.key is the minimum weight of an edge connecting v to a vertex in **A**
    - v.key= ∞ if there is no such edge
  - Attribute v.π names the parent of v in the tree.
    - v.π = null if no such parent exists

Prim's algorithm



a (∞,nil)
b (∞,nil)
c (∞,nil)
d (∞,nil)
e (∞,nil)
f (∞, nil)
g (∞, nil)
h (∞, nil)
i (∞, nil)

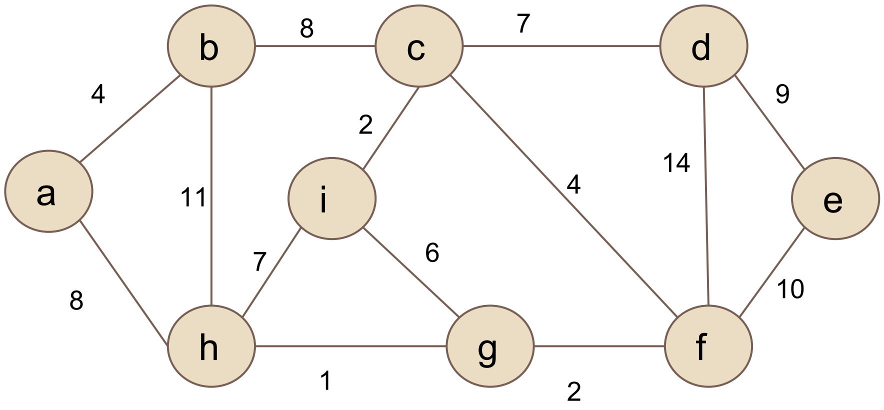
Prim's algorithm



a (∞,nil)
b (∞,nil)
c (∞,nil)
d (∞,nil)
e (∞,nil)
f (∞, nil)
g (∞, nil)
h (∞, nil)
i (∞, nil)

Start with arbitrary root. Here a. Set a.key=0

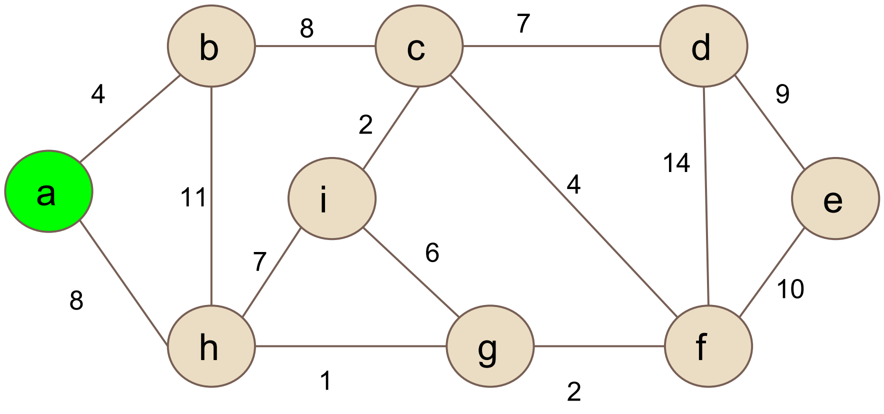
Prim’s algorithm



a (0,nil)
b (∞,nil)
c (∞,nil)
d (∞,nil)
e (∞,nil)
f (∞, nil)
g (∞, nil)
h (∞, nil)
i (∞, nil)

Start with arbitrary root. Here a. Set a.key=0

Prim’s algorithm

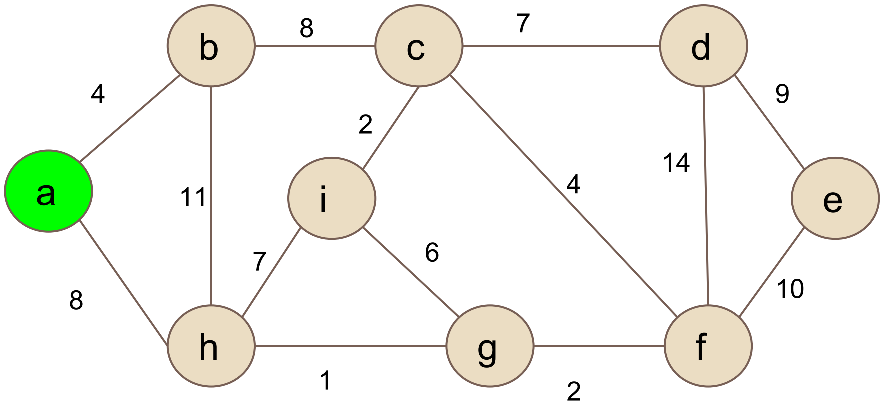


b (∞,nil)
c (∞,nil)
d (∞,nil)
e (∞,nil)
f (∞, nil)
g (∞, nil)
h (∞, nil)
i (∞, nil)

a (0,nil)

Extract minimum of Q and add it to minimum spanning tree.

Prim's algorithm

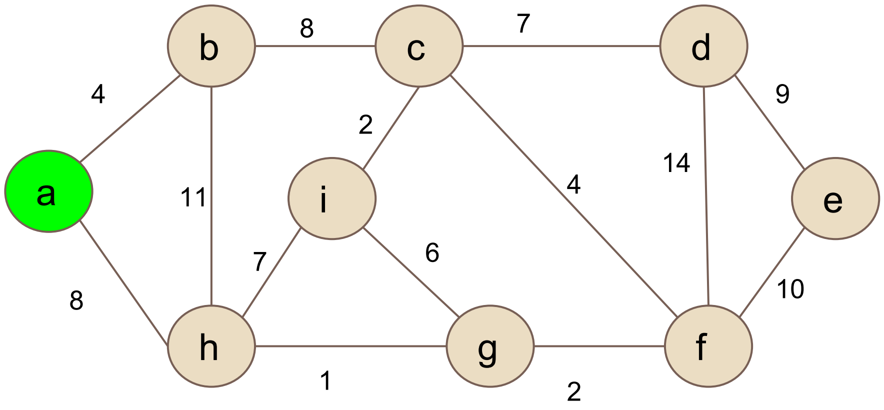


b (∞,nil)
c (∞,nil)
d (∞,nil)
e (∞,nil)
f (∞, nil)
g (∞, nil)
h (∞, nil)
i (∞, nil)

a (0,nil)

For each outgoing edge (a,v) of a:
If v is in Q and w(a,v) < v.key
Update v.π = a
v.key = w(u,v)

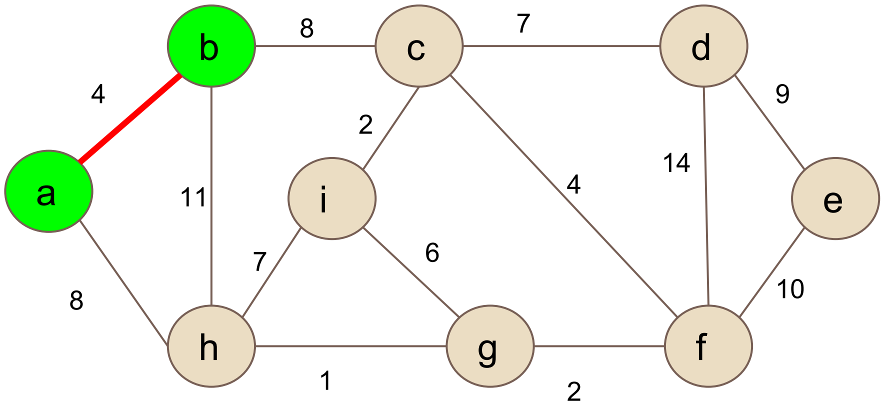
Prim's algorithm



b (4,a)
h (8, a)
c (∞,nil)
d (∞,nil)
e (∞,nil)
f (∞, nil)
g (∞, nil)
i (∞, nil)

For each outgoing edge (a,v) of a:
If v is in Q and w(a,v) < v.key
Update v.π = a
v.key = w(u,v)

Prim's algorithm

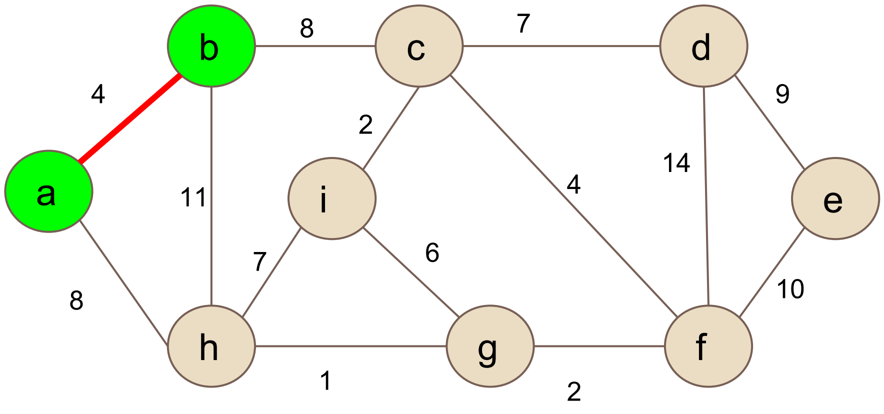


Extract minimum of Q and add it to minimum spanning tree.

h (8, a)
c (∞,nil)
d (∞,nil)
e (∞,nil)
f (∞, nil)
g (∞, nil)
i (∞, nil)

b (4,a)

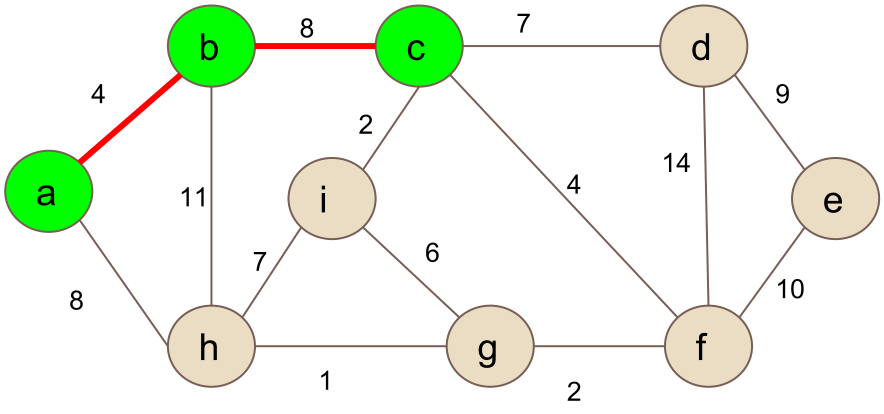
Prim's algorithm



c (8,b)
h (8, a)
d (∞,nil)
e (∞,nil)
f (∞, nil)
g (∞, nil)
i (∞, nil)

For each outgoing edge (a,v) of a:
If v is in Q and w(a,v) < v.key
Update v.π = a
v.key = w(u,v)

Prim's algorithm

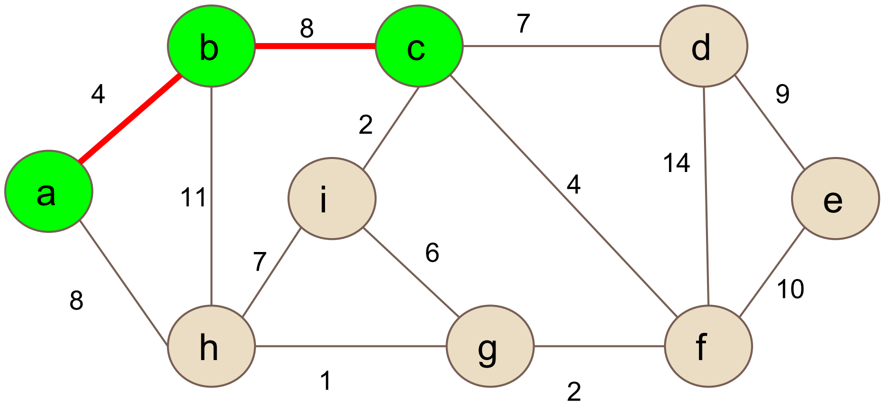


h (8, a)
d (∞,nil)
e (∞,nil)
f (∞, nil)
g (∞, nil)
i (∞, nil)

c (8,b)

Extract minimum of Q and add it to minimum spanning tree.

Prim’s algorithm

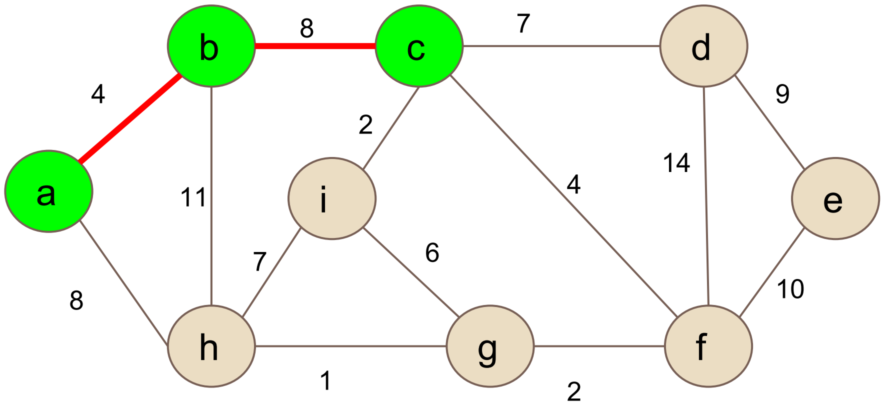


h (8, a)
d (7,c)
e (∞,nil)
f (4, c)
g (∞, nil)
i (2, c)

c (8,b)

For each outgoing edge (a,v) of a:
If v is in Q and w(a,v) < v.key
Update v.π = a
v.key = w(u,v)

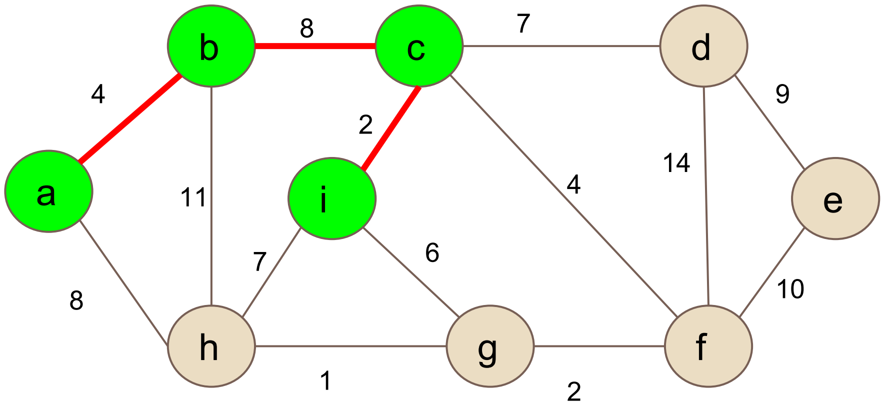
Prim's algorithm



i (2, c)
f (4, c)
h (8, a)
d (7,c)
e (∞,nil)
g (∞, nil)

For each outgoing edge (a,v) of a:
If v is in Q and w(a,v) < v.key
Update v.π = a
v.key = w(u,v)

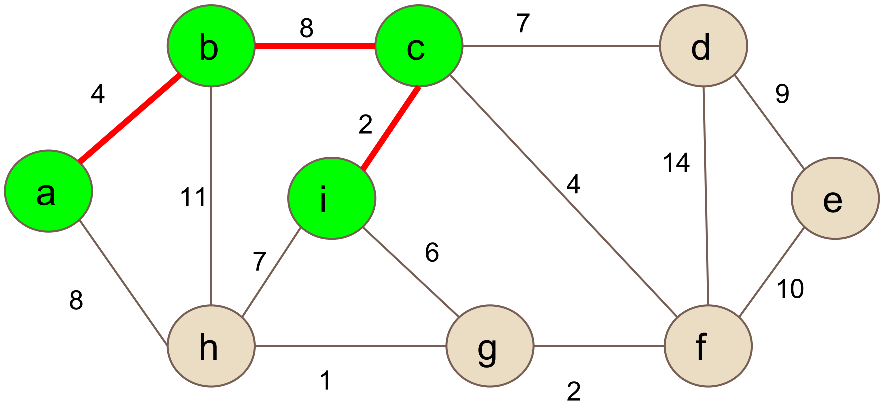
Prim’s algorithm



f (4, c)
h (8, a)
d (7,c)
e (∞,nil)
g (∞, nil)

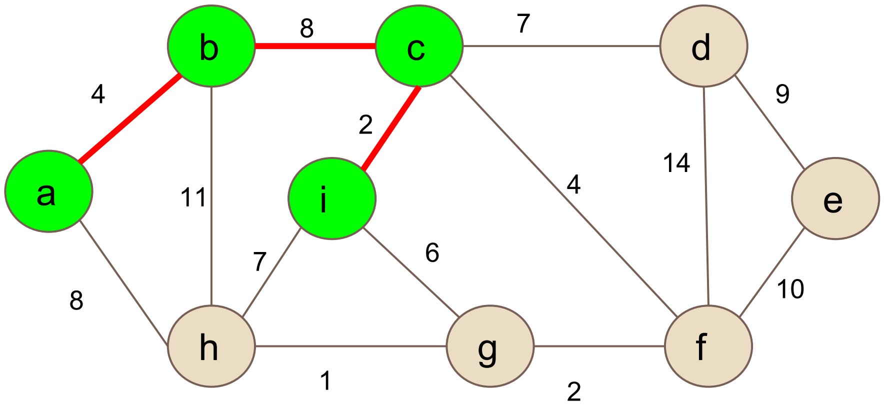
i (2,c)

Prim's algorithm



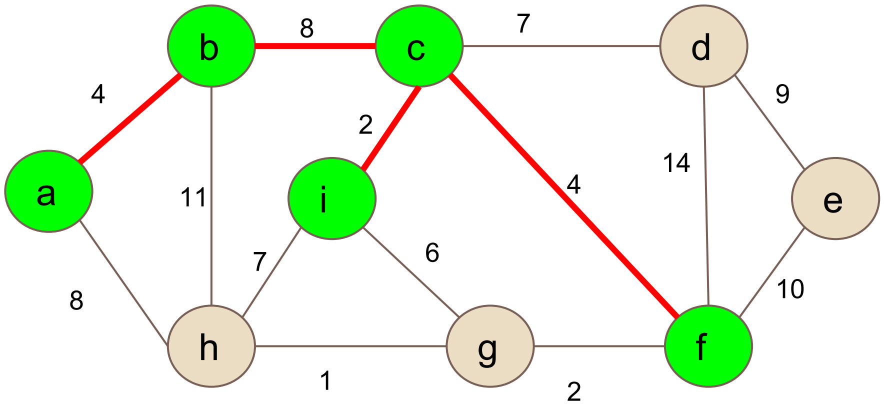
f (4, c)
h (7, i)
d (7,c)
e (∞,nil)
g (6, i)

Prim’s algorithm



f (4, c)
g (6, i)
h (7, i)
d (7,c)
e (∞,nil)

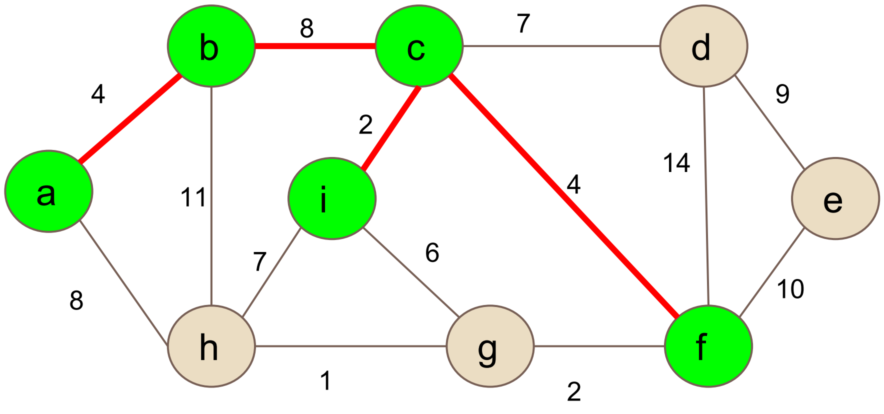
Prim's algorithm



g (6, i)
h (7, i)
d (7,c)
e (∞,nil)

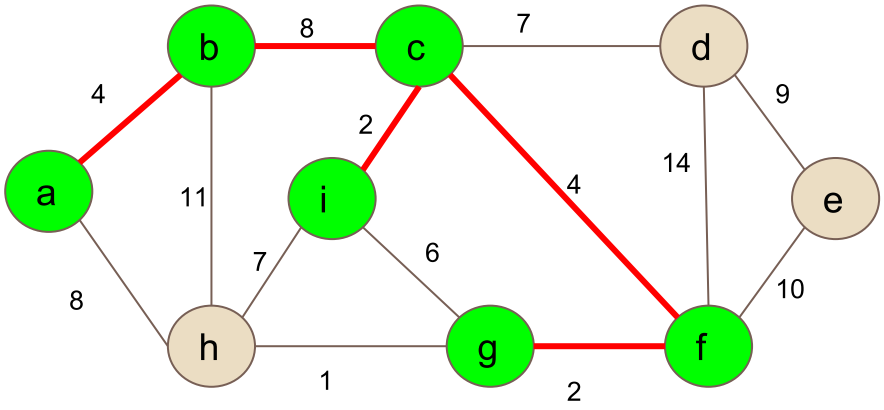
f(4,c)

Prim's algorithm



g (2, f)
h (7, i)
d (7,c)
e (10,f)

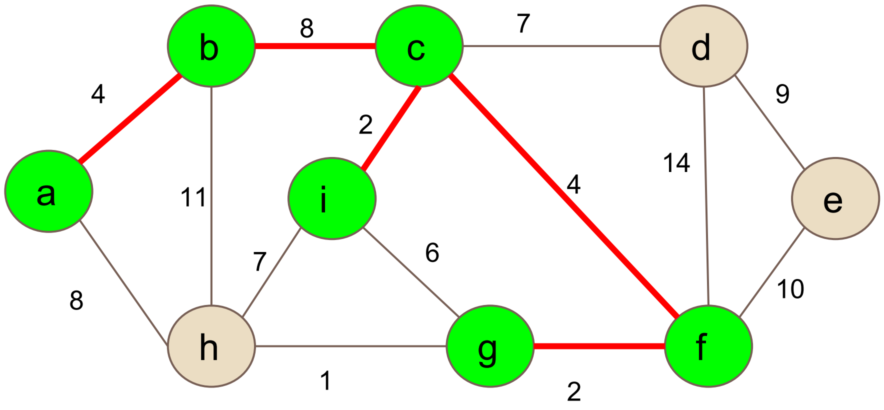
Prim’s algorithm



h (7, i)
d (7,c)
e (10,f)

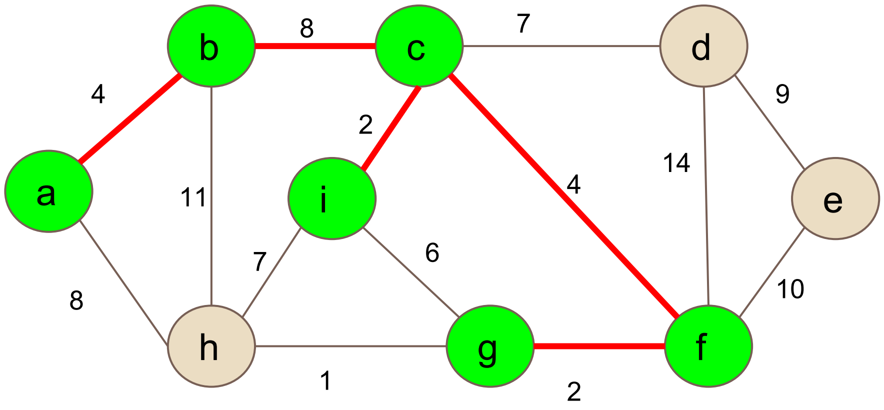
g(2,f)

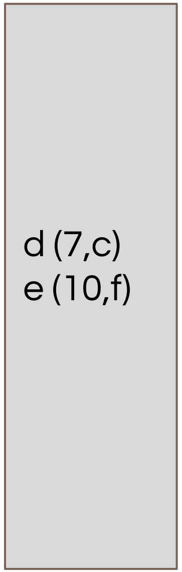
Prim's algorithm

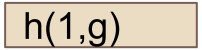


h (1, g)
d (7,c)
e (10,f)

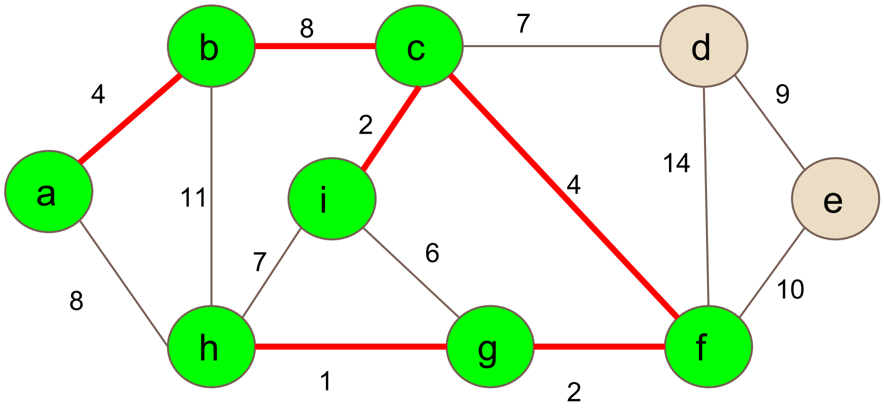
Prim’s algorithm





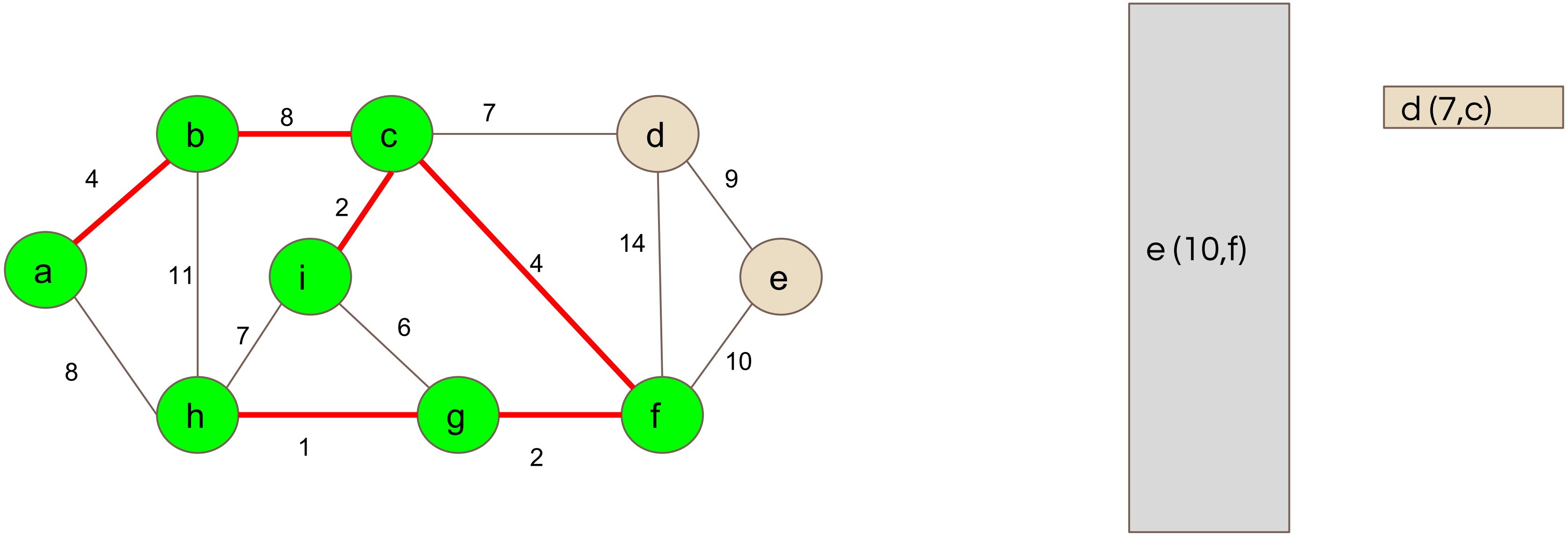


Prim's algorithm

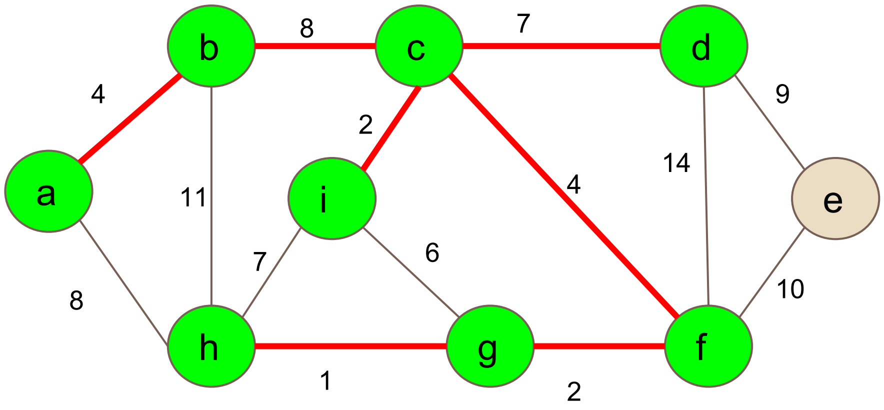


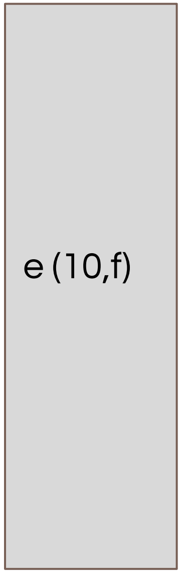
d (7,c)
e (10,f)

Prim’s algorithm

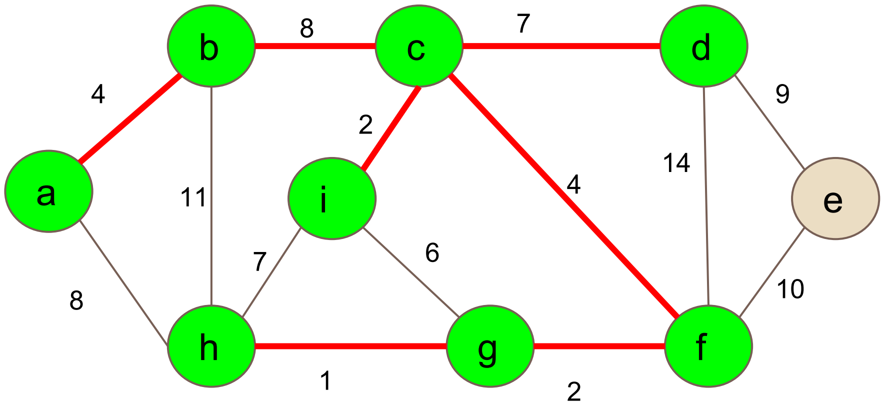


Prim's algorithm



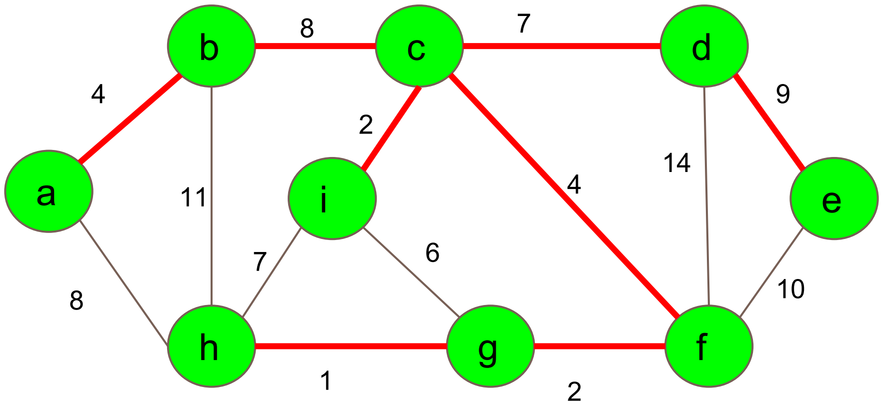


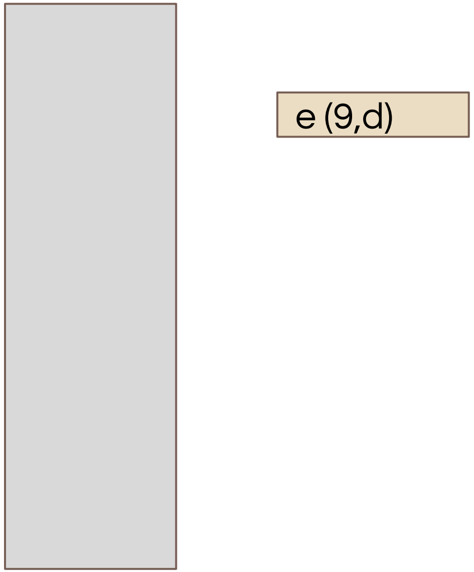
Prim’s algorithm



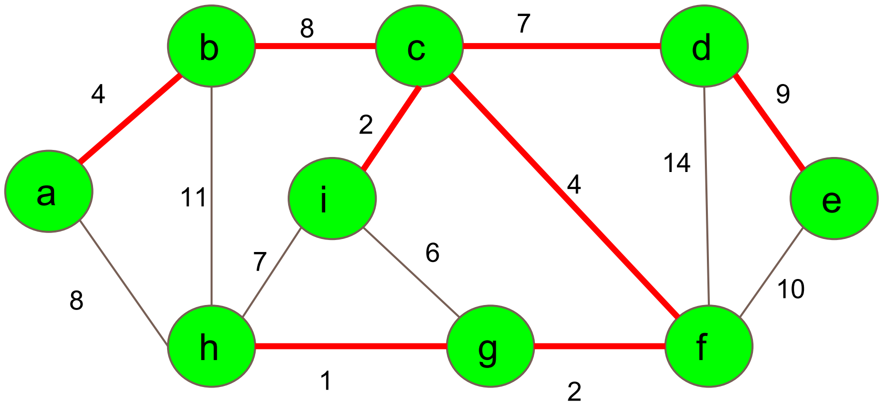
e (9,d)

Prim's algorithm



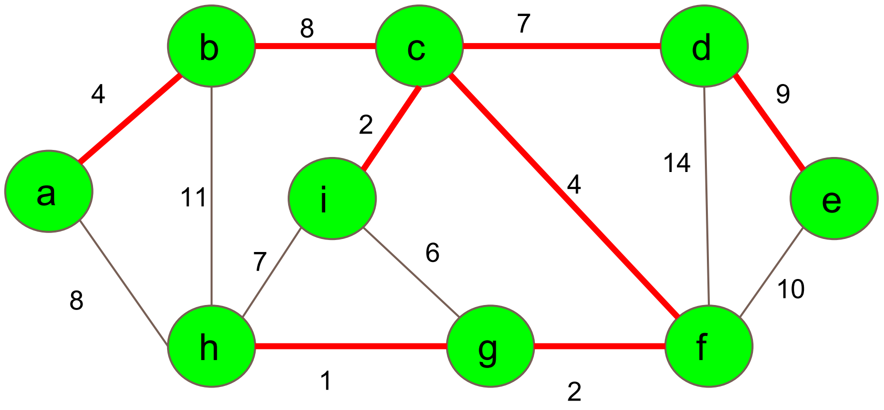


Prim's algorithm

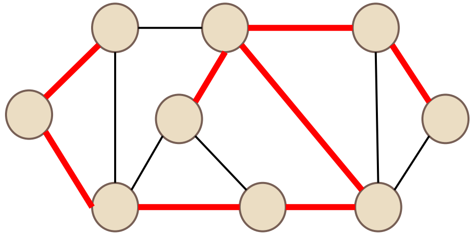


At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree

Prim's algorithm



Do Prim and Kruskal generate the same minimum spanning tree?



Prim's algorithm

```
For each u ∈ G.v:
        u.key = ∞
        u.π = nil
r.key = 0
Q = G.V
while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each edge (u,v):
                If v ∈ Q and w(u,v) < v.key
                        v.π = u
                         v.key = w(u,v)
                        DECREASE-KEY(Q,v,v.key)
```

Prim's algorithm

```
For each u ∈ G.v:
        u.key = ∞
        u.π = nil
r.key = 0
Q = G.V
while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each edge (u,v):
                If v ∈ Q and w(u,v) < v.key
                        v.π = u
                        v.key = w(u,v)
                        DECREASE-KEY(Q,v,v.key)
```

The vertices already placed into the minimum spanning tree are those in V-Q

For all vertices $v \in Q$, if $v.\pi$ is not null, then $v.key < \infty$ and v.key is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree

Prim's algorithm - Complexity

```
For each u ∈ G.v:
        u.key = ∞
        u.π = nil
r.key = 0
Q = G.V
while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each edge (u,v):
                If v ∈ Q and w(u,v) < v.key
                        v.π = u
                         v.key = w(u,v)
                        DECREASE-KEY(Q,v,v.key)
```

Prim's algorithm - Complexity

```
For each u ∈ G.v:
        u.key = ∞
        u.π = nil
r.key = 0
Q = G.V
while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each edge (u,v):
                If v ∈ Q and w(u,v) < v.key
                        v.π = u
                         v.key = w(u,v)
                        DECREASE-KEY(Q,v,v.key)
```

|V| * Insert(Q,v)

|V| * Extract-Min(Q)

|E| * Decrease-Key(Q)

Prim's algorithm - Complexity

```
For each u ∈ G.v:
        u.key = ∞
        u.π = nil
r.key = 0
Q = G.V
while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each edge (u,v):
                If v ∈ Q and w(u,v) < v.key
                        v.π = u
                        v.key = w(u,v)
                        DECREASE-KEY(Q,v,v.key)
```

|V| * Insert(Q,v)

|V| * Extract-Min(Q)

|E| * Decrease-Key(Q)

O(VlogV + ElogV) if use min-heap, O(VlogV + E) if use Fibonacci heaps

Taking a step back ..

- **Greedy algorithm**: An algorithm that uses the heuristic of making the locally optimal choice at each stage with the hope of finding the global optimum.

Taking a step back ..

- **Greedy algorithm**: An algorithm that uses the heuristic of making the locally optimal choice at each stage with the hope of finding the global optimum.

- Dijkstra's shortest-path algorithm makes a locally optimal choice: choosing the node in Q with minimum d value and moving it to the A set.
  - We proved that this leads to the global optimum.

Taking a step back ..

- **Greedy algorithm**: An algorithm that uses the heuristic of making the locally optimal choice at each stage with the hope of finding the global optimum.

- Dijkstra's shortest-path algorithm makes a locally optimal choice: choosing the node in Q with minimum d value and moving it to the A set.
  - We proved that this leads to the global optimum.

- Similarly, Prim's and Kruskal's locally optimum choices of adding a minimum-weight edge also yield the global optimum: a minimum spanning tree.

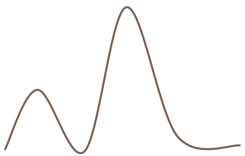
Taking a step back ..

- **Greedy algorithm**: An algorithm that uses the heuristic of making the locally optimal choice at each stage with the hope of finding the global optimum.

- Dijkstra's shortest-path algorithm makes a locally optimal choice: choosing the node in Q with minimum d value and moving it to the A set.
  - We proved that this leads to the global optimum.

- Similarly, Prim's and Kruskal's locally optimum choices of adding a minimum-weight edge also yield the global optimum: a minimum spanning tree.

- BUT: **Greediness does not always work!**



Taking a step back ..

- Prim, BFS, DFS all share a similar code structure

- Breadth-first-search (bfs)
  - best: next in queue
  - update: D[w] = D[v]+1
- Dijkstra's algorithm
  - best: next in priority queue
  - update: D[w] = min(D[w], D[v]+c(v,w))
- Prim's algorithm
  - best: next in priority queue
  - update: D[w] = min(D[w], c(v,w))

```
while (a vertex is unmarked) {
  v= best unmarked vertex
  mark v;
  for (each w adj to v)
        update D[w];
}
```