Minimum Spanning Trees

Chapter: 23 of your textbook

Consider the following problem

- A town has set of houses and set of roads
- Any road connects two and only two houses
- A road connecting houses u and v has repair cost w(u,v)
- Goal: Repair necessary number of roads and no more roads such that
  - Everyone stays connected i.e., can reach every houses from all other houses
  - Total repair cost is minimum

Consider the another problem

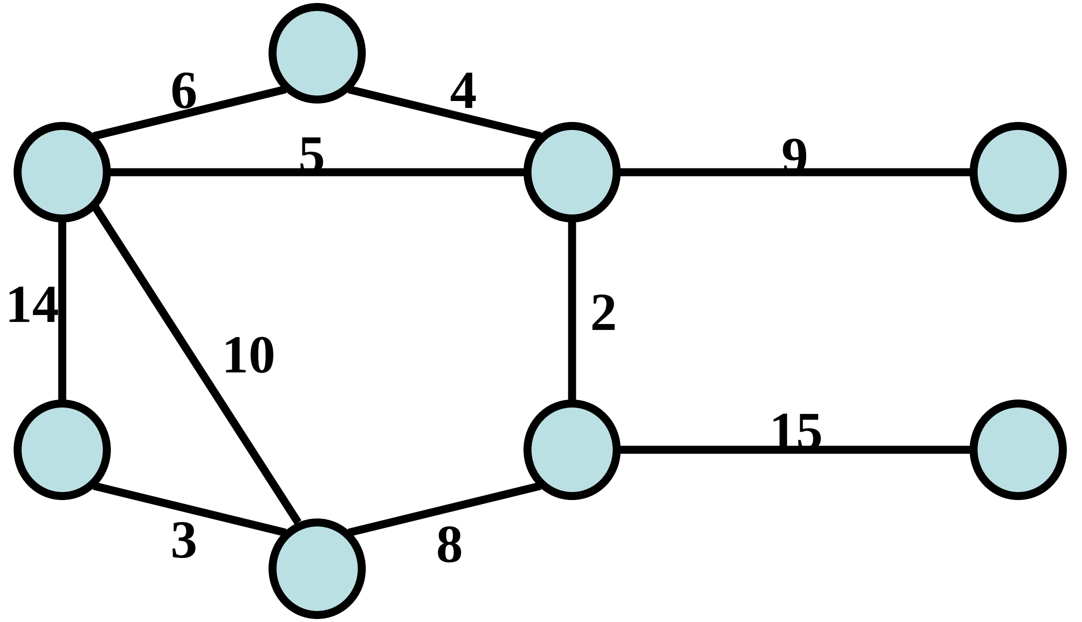
- Electronic circuit designs often need to make the pins of several components electrically equivalent by writing them together
- Wires are typically costly
- Goal: We need just the right amount of wire such that
  - Every pin stays connected i.e., can reach every pin from all other pins
  - Total length of the wires required is minimum

How do we attack such problem?

- Model the problem as a graph
  - Undirected graph G=(V,E)
  - Weight w(u,v) on each edge (u,v)∈ E
  - Find T as a subset of E such that
    - T connects all vertices (T is a spanning tree)
    - Total weight $w(T) = \sum_{(u,v)\in T} w(u,v)$ is minimized
  - A spanning tree whose weight is minimum over all spanning trees is called minimum spanning tree (MST)

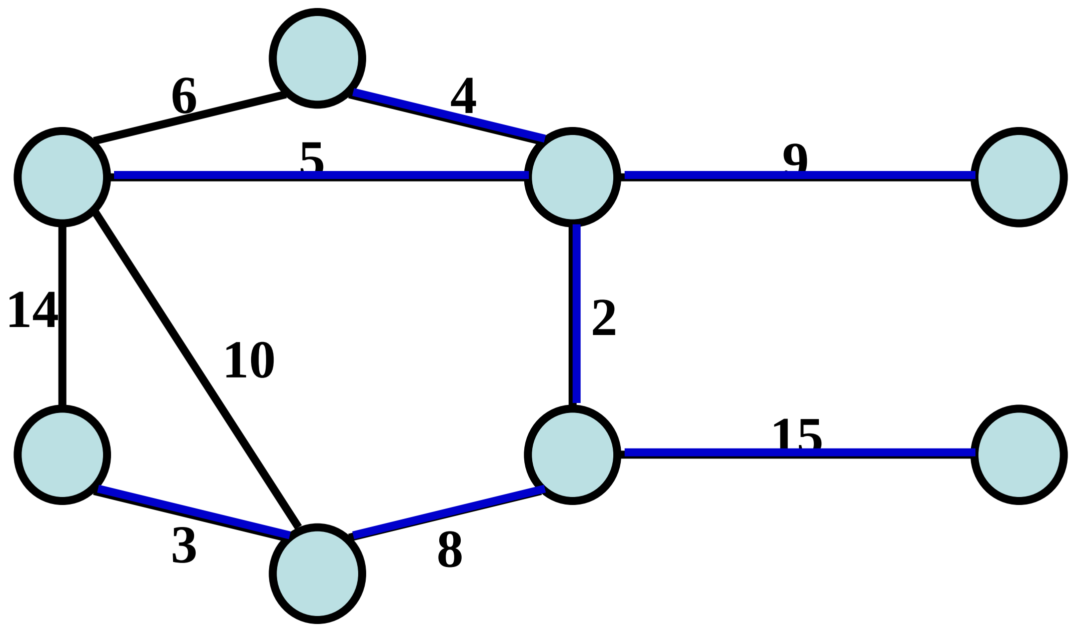
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph:



Minimum Spanning Tree

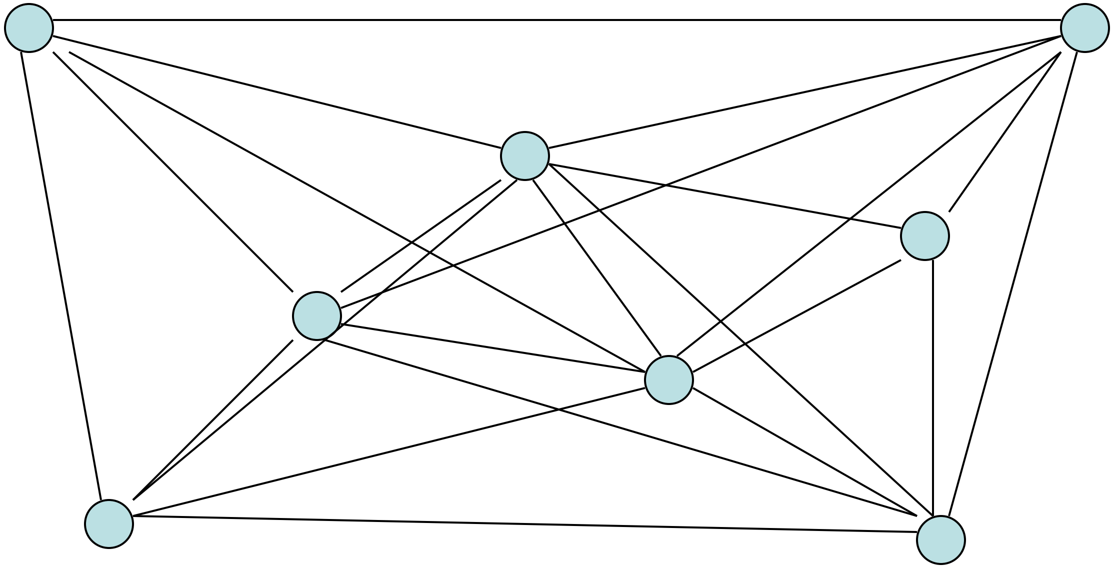
- Problem: given a connected, undirected, weighted graph, find a *spanning tree* using edges that minimize the total weight



- A spanning tree is a tree that connects all vertices (no cycles)
- Number of edges = ?
- A spanning tree has no designated root.

How to find MST?

- Connect every city to the closest city?
  - Does not guarantee a spanning tree
    - Not necessarily



MST

- Some properties
  - It has |V|-1 edges
  - It has no cycle
  - It might not be unique
- Finding a solution
  - We will build a set A of edges
  - Initially A has no edges (empty set)
  - We add edges to A by maintaining the loop invariant: A is a subset of some MST
  - Add only those edges that maintain this loop invariant
  - If A is a subset of some MST and edge (u,v) is safe for A if and only if A $\cup$\{(u,v)\} is also subset of some MST

Generic MST

- Generic MST code is as follows

```
GENERIC-MST(G, w)
A = ∅
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

- How do we find safe edges?

Generic MST

- Generic MST code is as follows

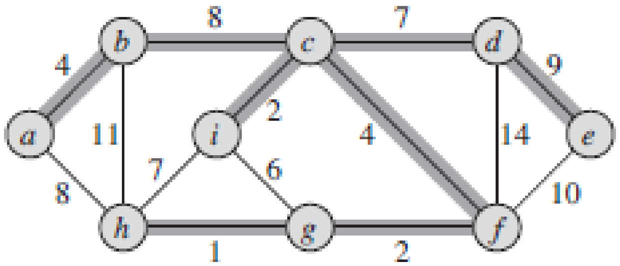
GENERIC-MST(G, w)

$A = \emptyset$
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    $A = A \cup \{(u, v)\}$
return A

- How do we find safe edges?

Finding safe edge

- Let's look at an example
  - Edge (h,g) has the lowest weight of any edge in the graph. Is it safe for A={}?



  - We will first introduce some definitions

Finding safe edge

- Some definitions first
  - Let S be a subset of vertices V and A be subset of edges E
  - A cut(S,V-S) is a partition of vertices into disjoint sets S and V-S
  - An edge (u,v)∈E crosses cut (S,V-S) if one end point of the edge is in V and the other is in V-S
  - A cut respects A if and only if no edge in A crosses the cut
  - An edge is a light edge crossing a cut if and only if its weight is minimum over all edges crossing the cut
    - For a given cut there may be more than one light edge

Main theorem for identifying safe edge

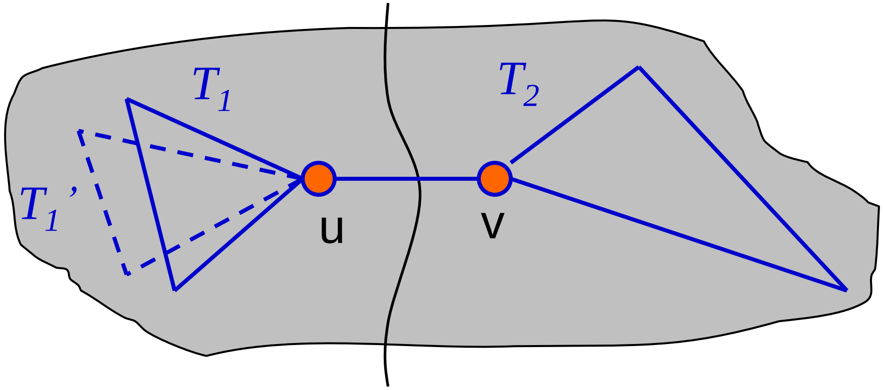
- Theorem
  - Let A be a subset of some MST, (S,V-S) be a cut that respects A, and (u,v) be a light edge crossing (S,V-S). Then (u,v) is safe for A
- Proof:
  - read from the book
  - We will do it in class
- Note that that safe edge selection is “greedy”
  - Out of all edges crossing the cut the we choose the one with minimum weight

MST growing strategy

- In Generic-MST
  - A is a forest containing connected components, initially each component is a single vertex
  - Any safe edge merges two of the components into one, and each component is a tree
  - Since MST has exactly |V|-1 edges, the for loop iterates |V|-1 time, in other words, after adding |V|-1 edges we are down to just one component
- Two strategies for growing
  - Grow a single tree until it covers all vertices (Prim's algorithm)
  - Grow multiple trees and combine them until it becomes a single tree that encompasses all vertices (Kruskal's algorithm)

Minimum Spanning Tree

- MSTs satisfy the *optimal substructure* property: an optimal tree is composed of optimal subtrees
  - Let T be an MST of G with an edge (*u,v*) in the middle
  - Removing (*u,v*) partitions T into two trees T_1 and T_2
  - $w(T) = w(u,v) + w(T_1) + w(T_2)$
- Claim 1: T_1 is an MST of $G_1 = (V_1, E_1)$, and T_2 is an MST of $G_2 = (V_2, E_2)$

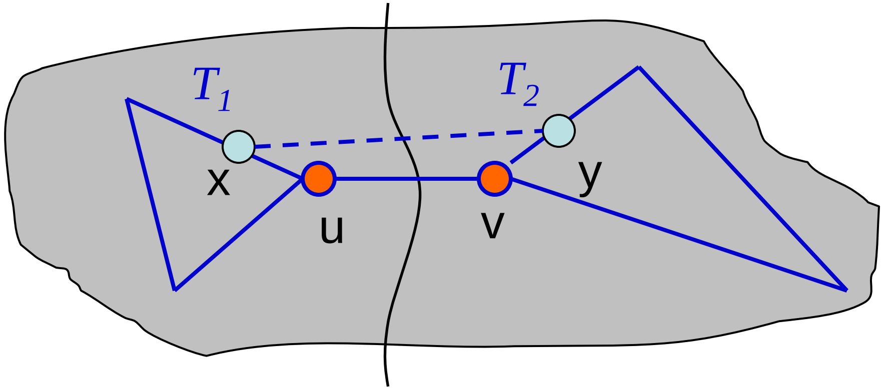


Proof by contradiction:

- if T_1 is not optimal, we can replace T_1 with a better spanning tree, T_1'
- T_1', T_2 and (u, v) form a new spanning tree T'
- $W(T') < W(T)$. Contradiction.

Minimum Spanning Tree

- MSTs satisfy the *optimal substructure* property: an optimal tree is composed of optimal subtrees
  - Let T be an MST of G with an edge (*u,v*) in the middle
  - Removing (*u,v*) partitions T into two trees T_1 and T_2
  - $w(T) = w(u,v) + w(T_1) + w(T_2)$
  - *A light edge (u,v) between two trees T_1 and T_1 (where there is no overlap between T_1 and T_2) is an edge such that u belongs to V_1 and v belongs to V_2*
- Claim 2: (u, v) is the light (minimum weight edge among all edges crossing a cut) edge connecting $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$



Proof by contradiction:

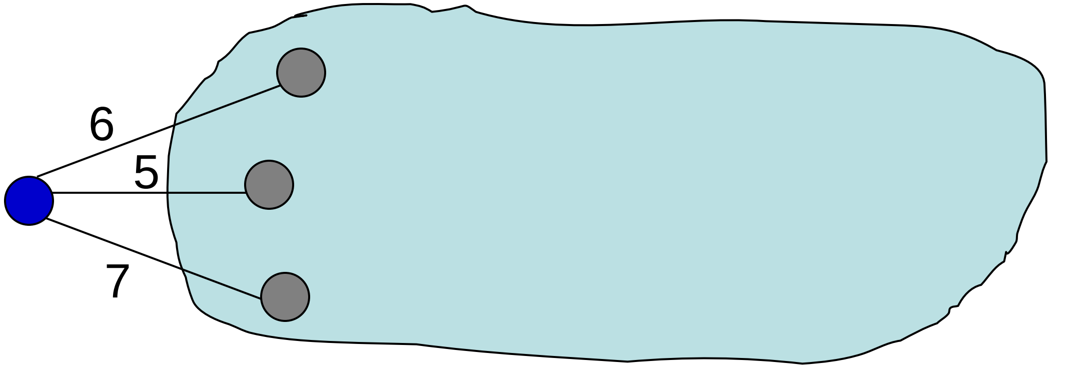
- if (u, v) is not the lightest edge, we can remove it, and reconnect T_1 and T_2 with a lighter edge (x, y)
- T_1, T_2 and (x, y) form a new spanning tree T'
- $W(T') < W(T)$. Contradiction.

Algorithms

- Generic idea:
  - Compute MSTs for sub-graphs
  - Connect two MSTs for sub-graphs with the lightest edge
- Two of the most well-known algorithms
  - Prim’s algorithm
  - Kruskal’s algorithm
  - Let’s first talk about the ideas behind the algorithms without worrying about the implementation and analysis

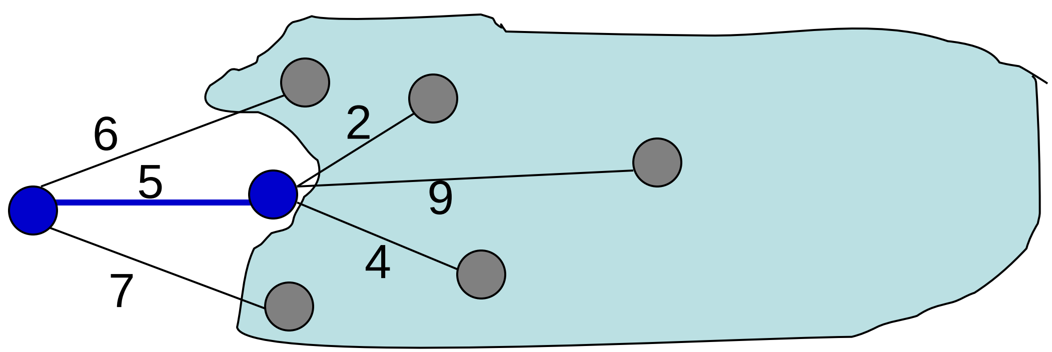
Prim's algorithm

- Basic idea:
  - Start from an arbitrary single node
    - A MST for a single node has no edge
  - Gradually build up a single larger and larger MST



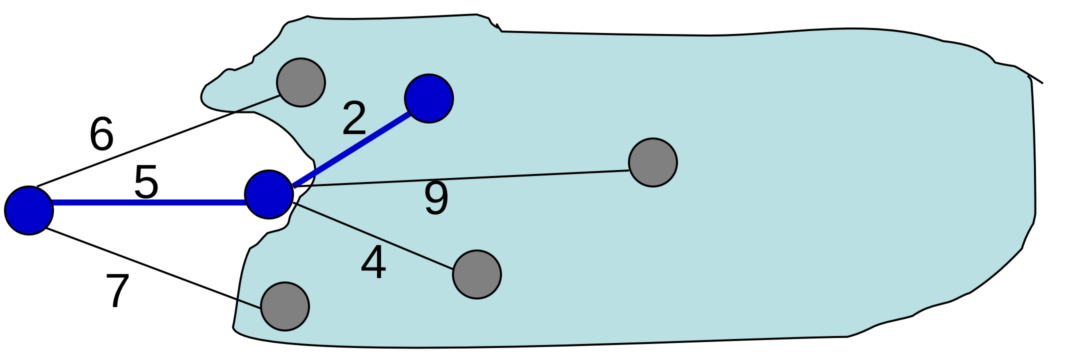
Prim's algorithm

- Basic idea:
  - Start from an arbitrary single node
    - A MST for a single node has no edge
  - Gradually build up a single larger and larger MST



Prim's algorithm

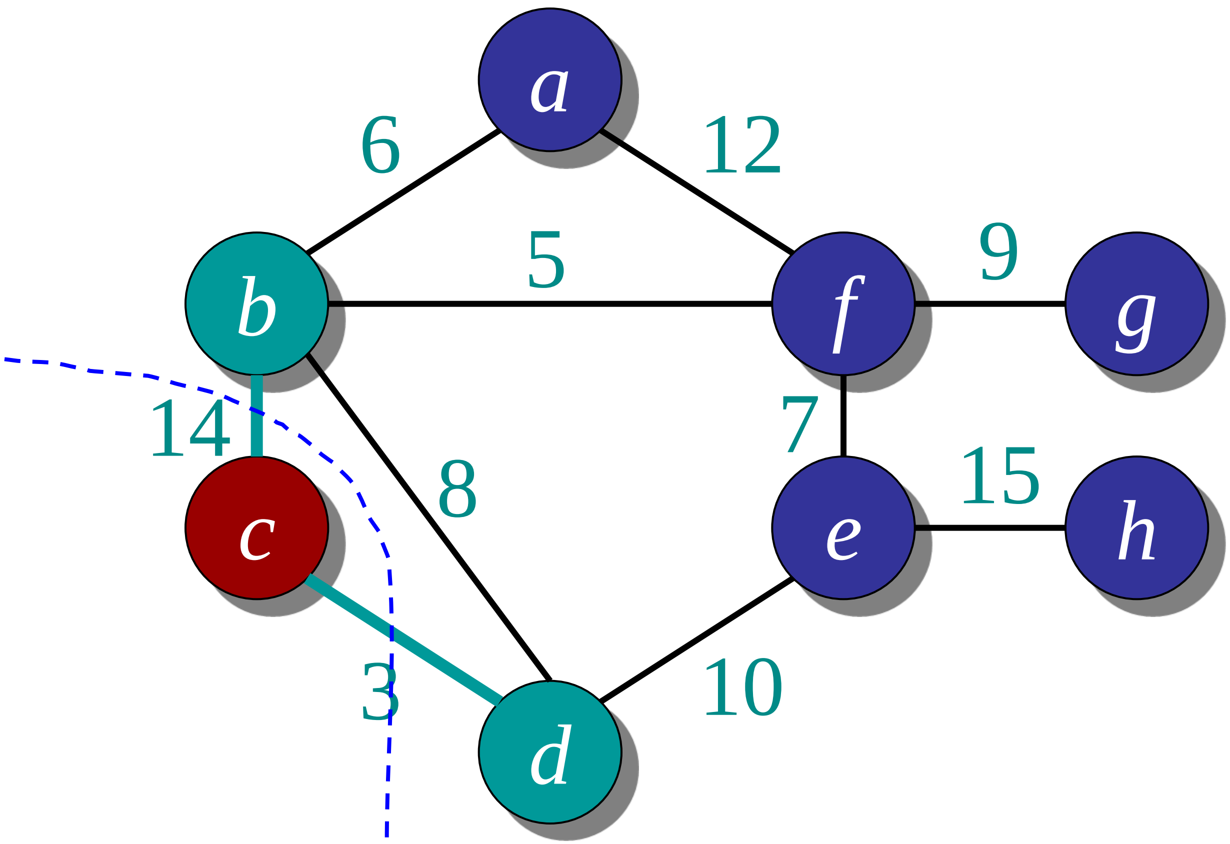
- Basic idea:
  - Start from an arbitrary single node
    - A MST for a single node has no edge
  - Gradually build up a single larger and larger MST



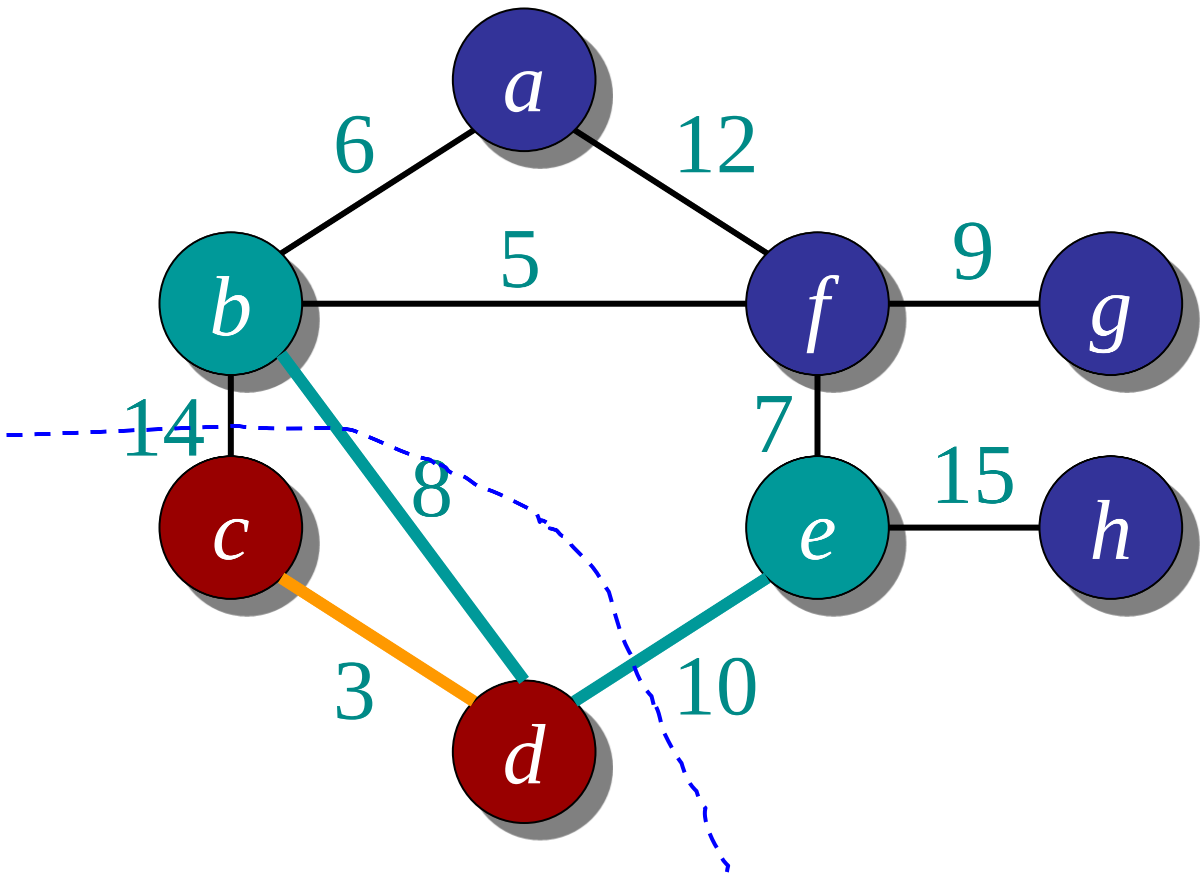
Prim’s algorithm in words

- Randomly pick a vertex as the initial tree T
- Gradually expand into a MST:
  - For each vertex that is not in T but directly connected to some nodes in T
    - Compute its minimum distance to any vertex in T
  - Select the vertex that is closest to T
    - Add it to T

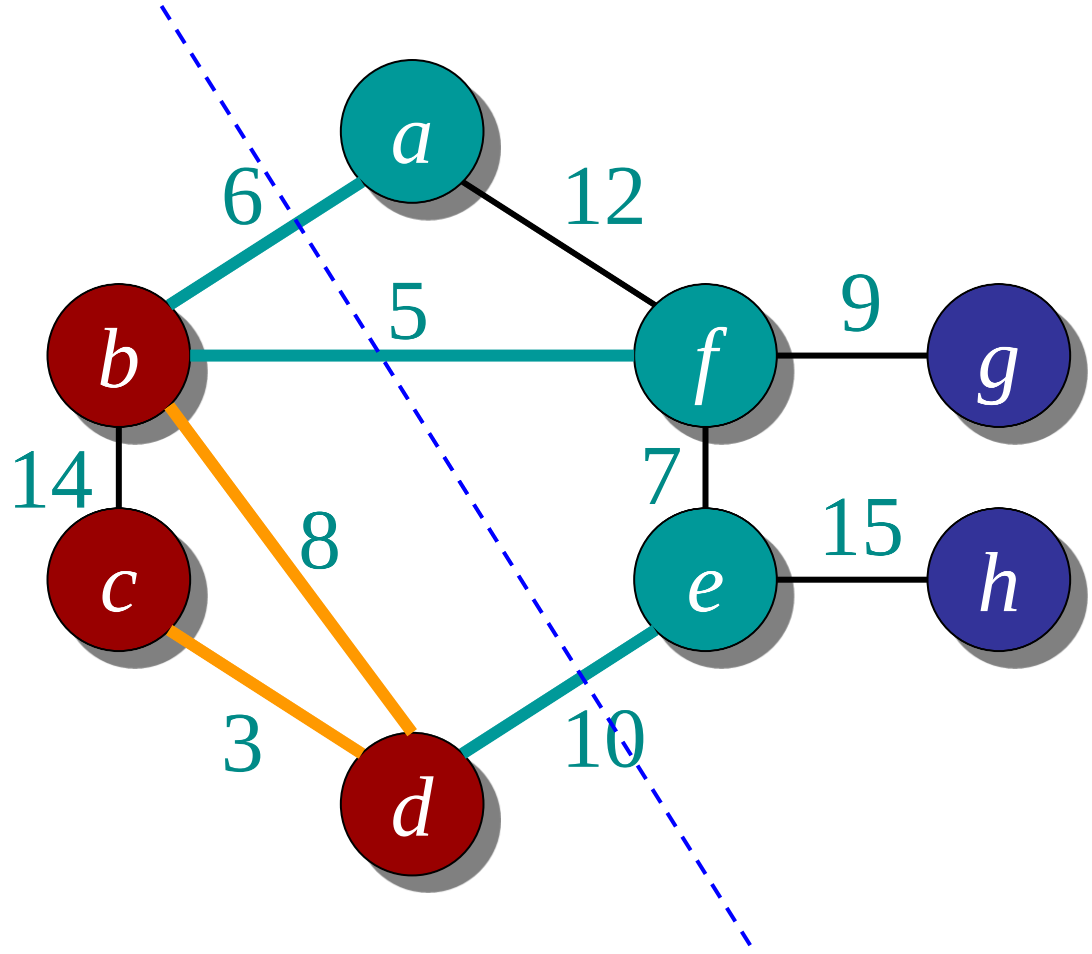
Example



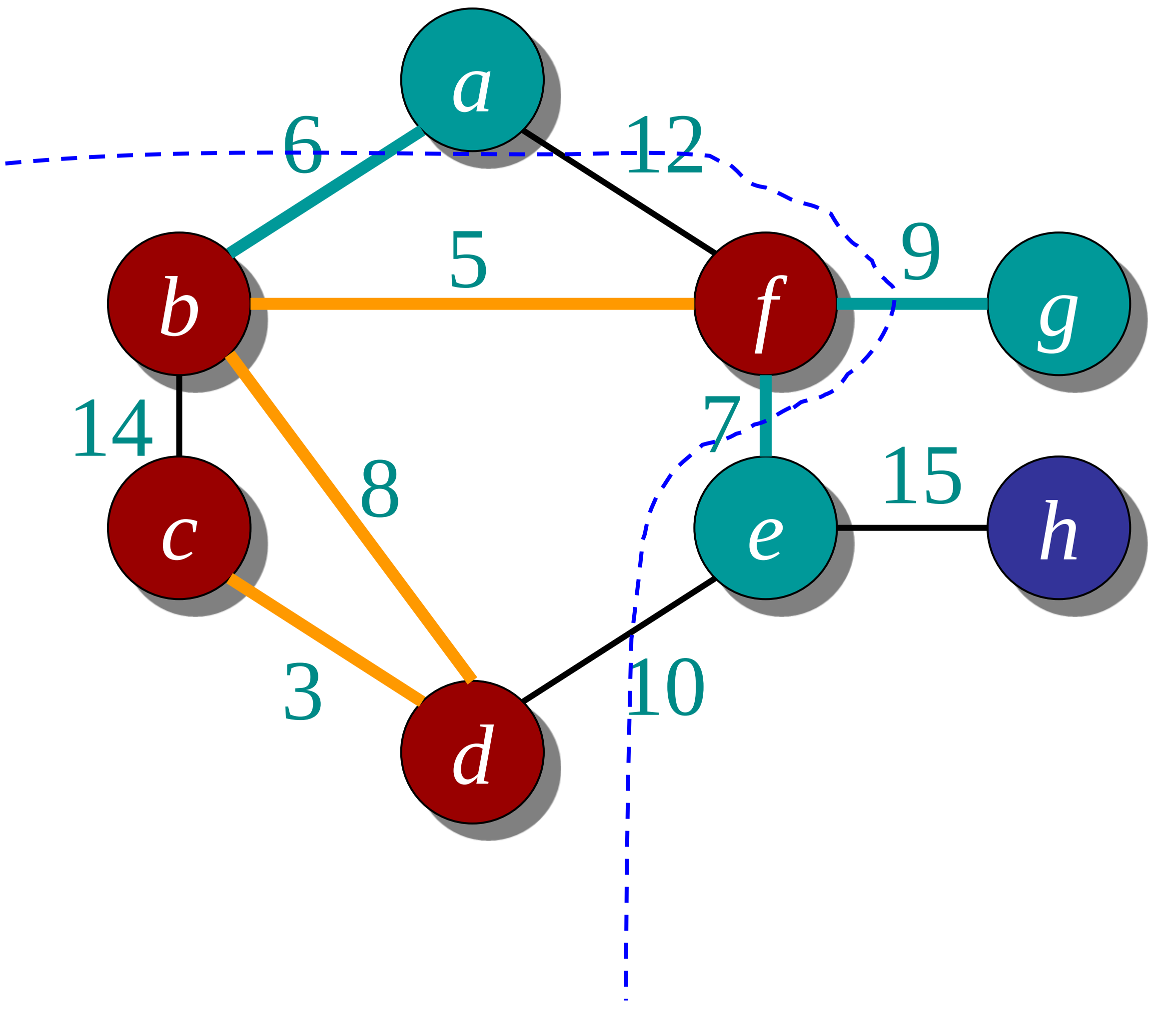
Example



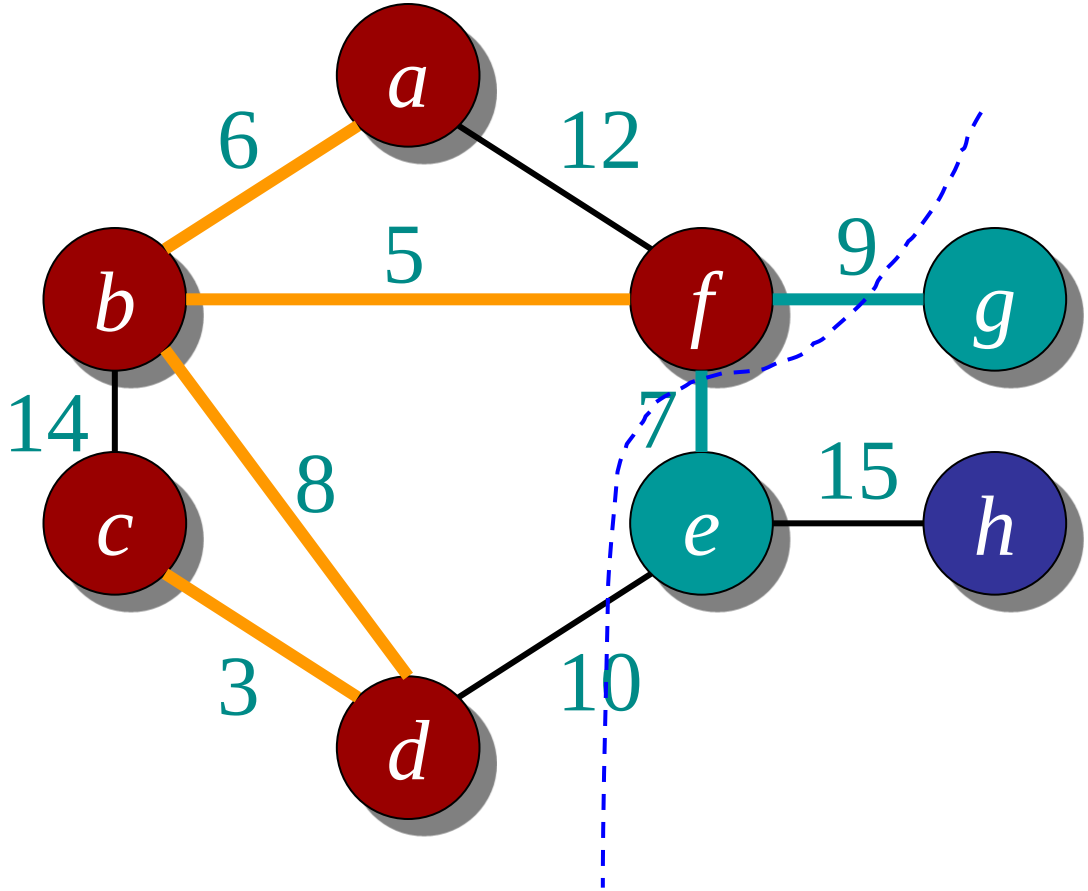
Example



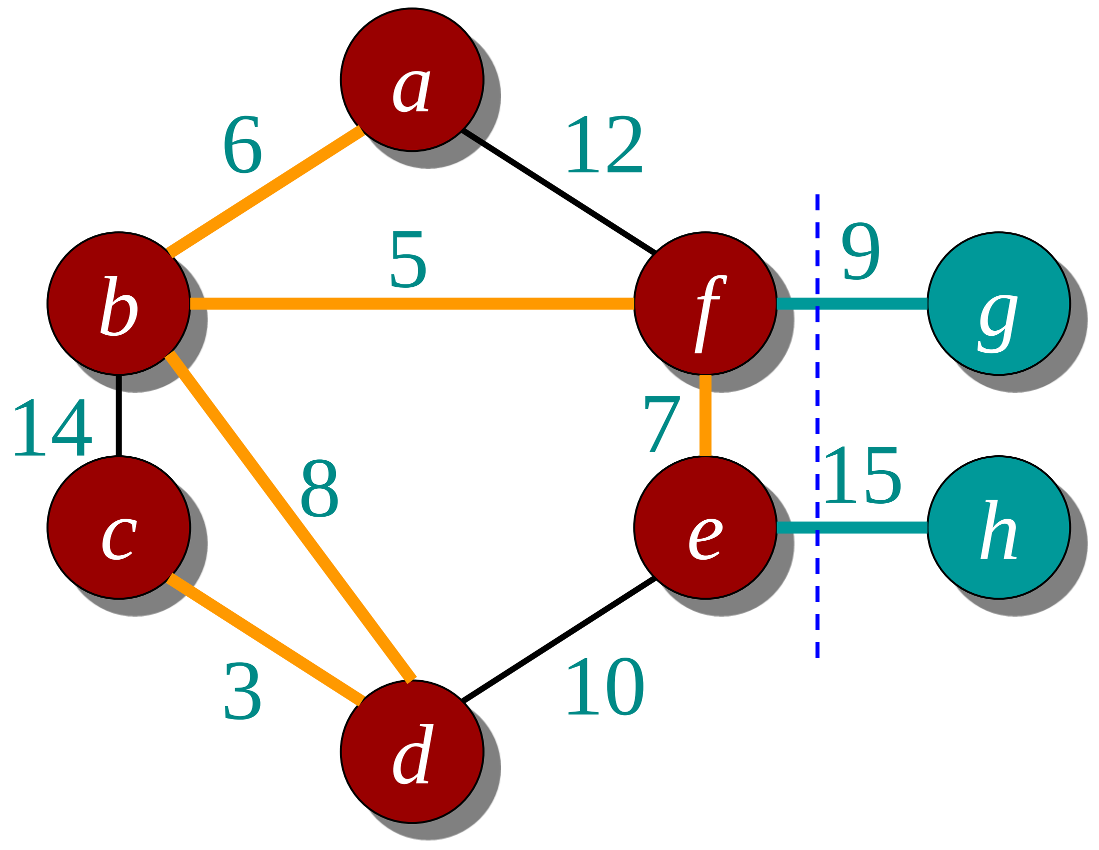
Example



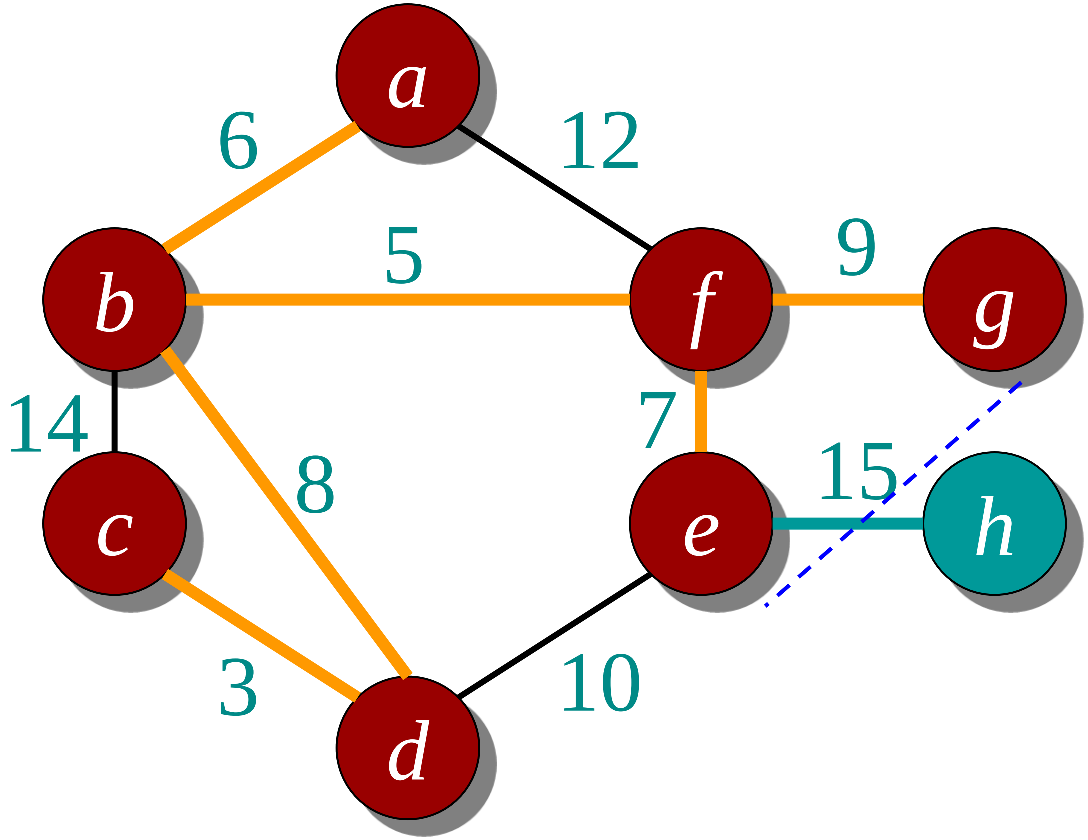
Example



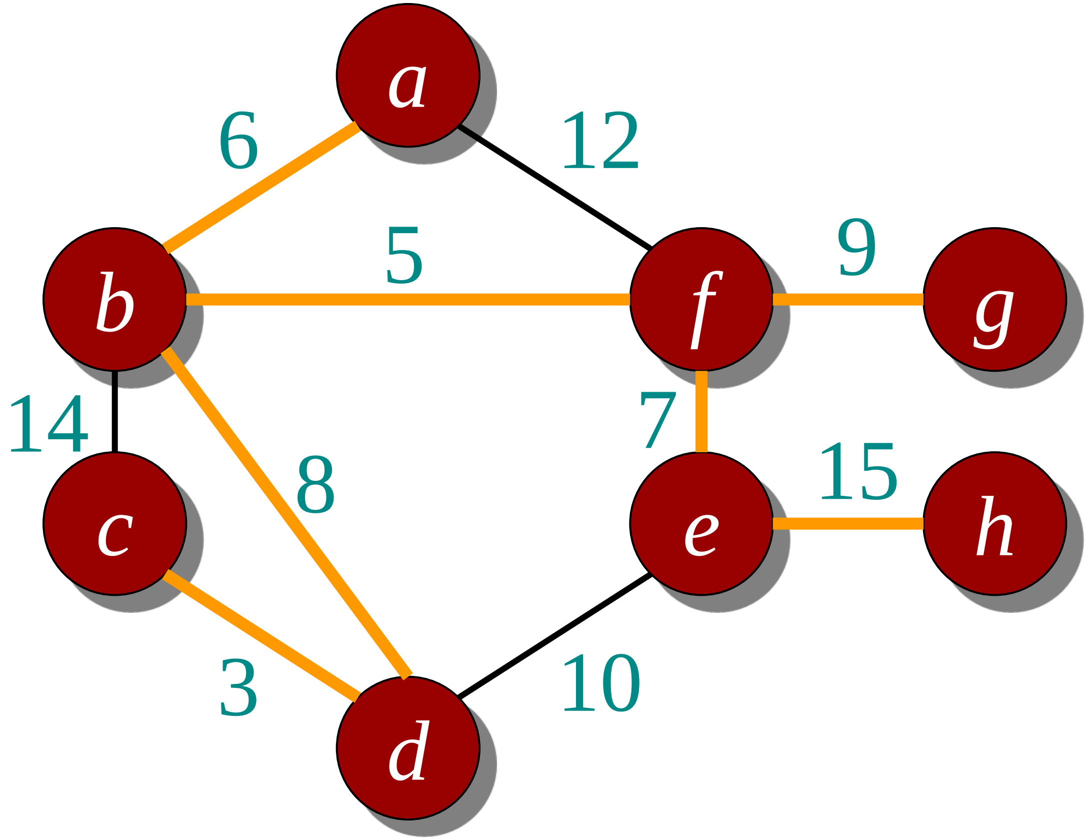
Example



Example



Example



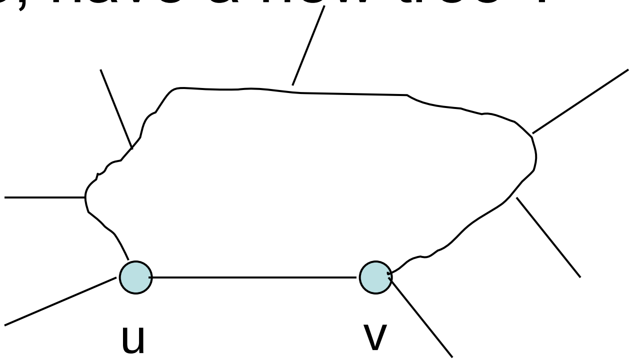
Total weight = 3 + 8 + 6 + 5 + 7 + 9 + 15 = 53

Kruskal’s algorithm

- Basic idea:
  - Grow many small trees
  - Find two trees that are closest (i.e., connected with the lightest edge), join them with the lightest edge
  - Terminate when a single tree forms

Claim

- If edge (u, v) is the lightest among all edges, (u, v) is in a MST
- Proof by contradiction:
  - Suppose that (u, v) is not in any MST
  - Given a MST T, if we connect (u, v), we create a cycle
  - Remove an edge in the cycle, have a new tree T'
  - W(T') < W(T)



By the same argument, the second, third, …, lightest edges, if they do not create a cycle, must be in MST

Kruskal's algorithm in words

- Procedure:
  - Sort all edges into non-decreasing order
  - Initially each node is in its own tree
  - For each edge in the sorted list
    - If the edge connects two separate trees, then
      - join the two trees together with that edge

Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

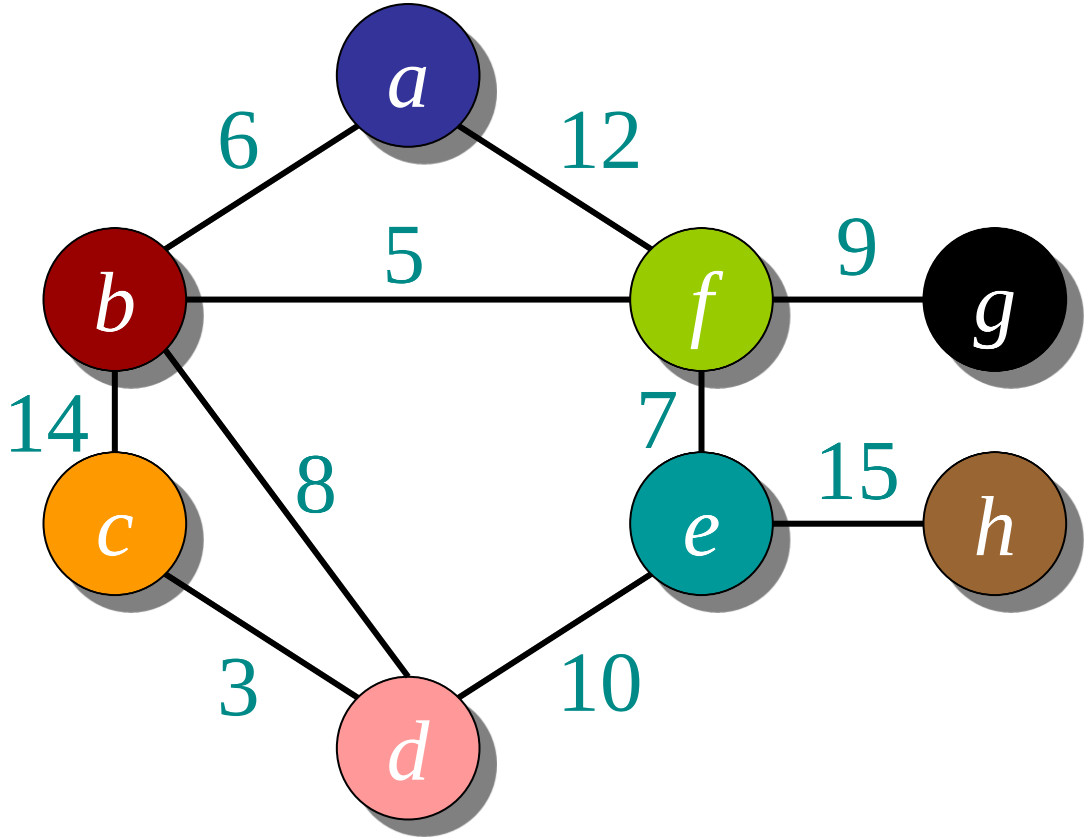
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

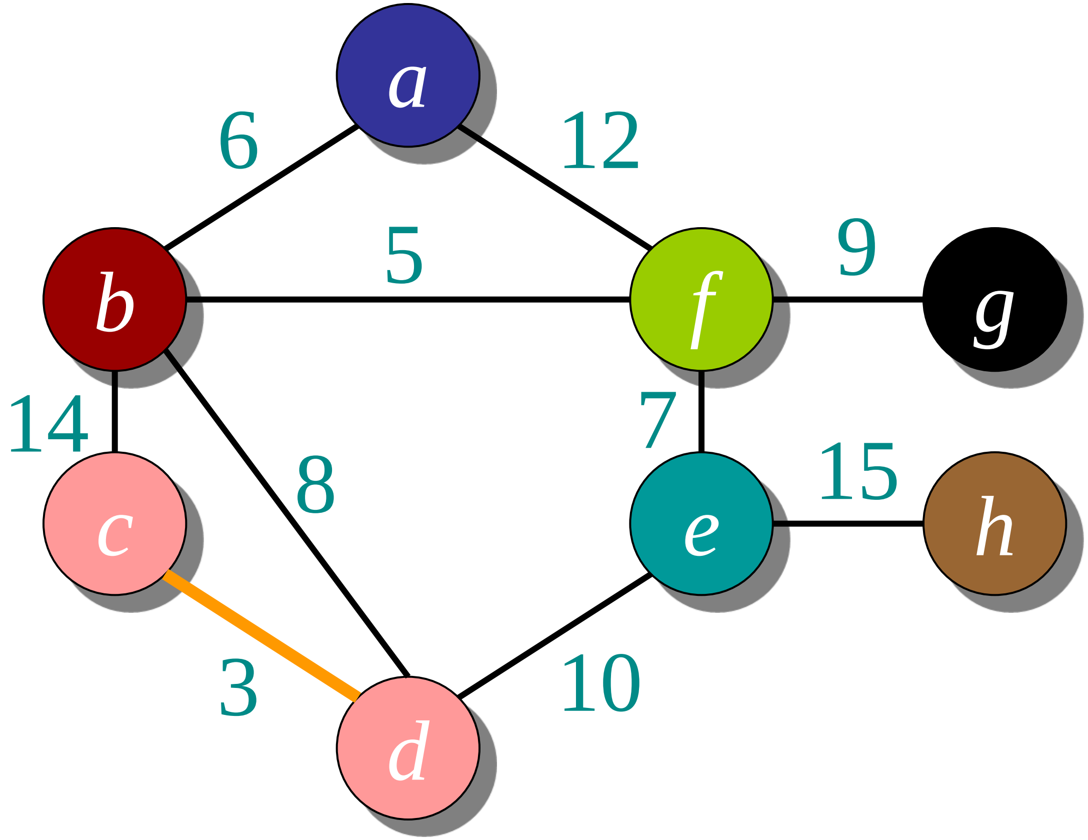
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

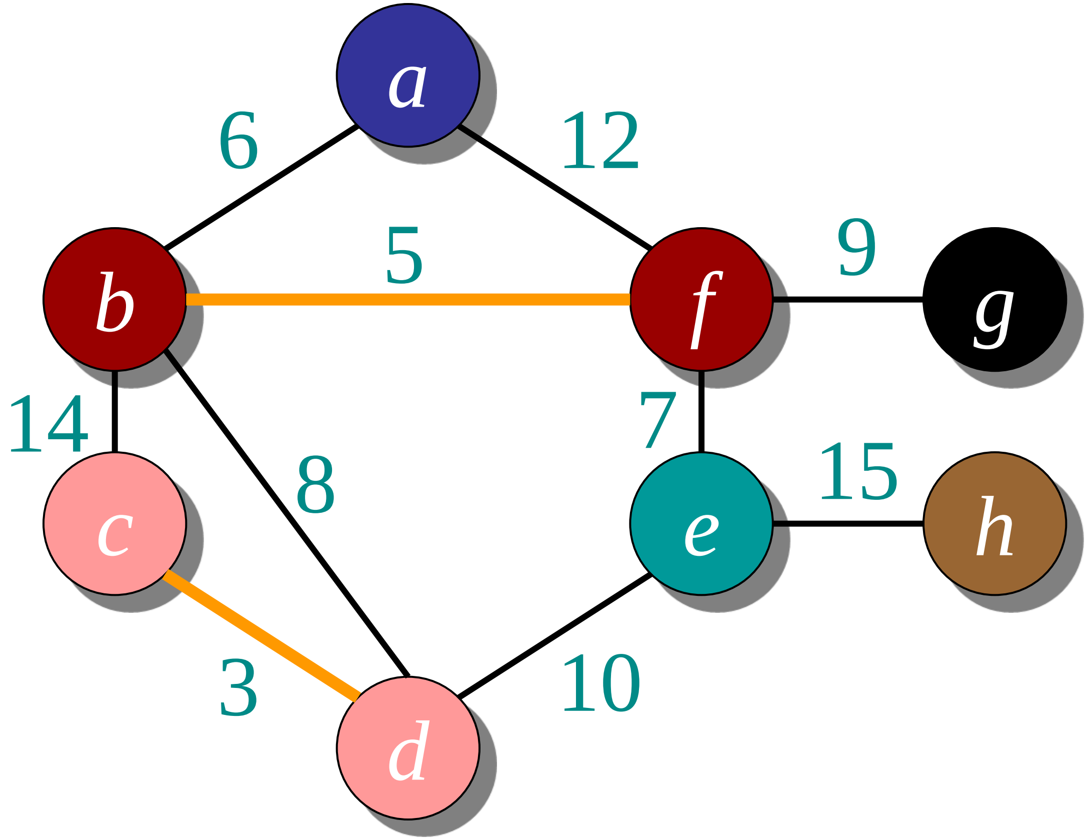
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

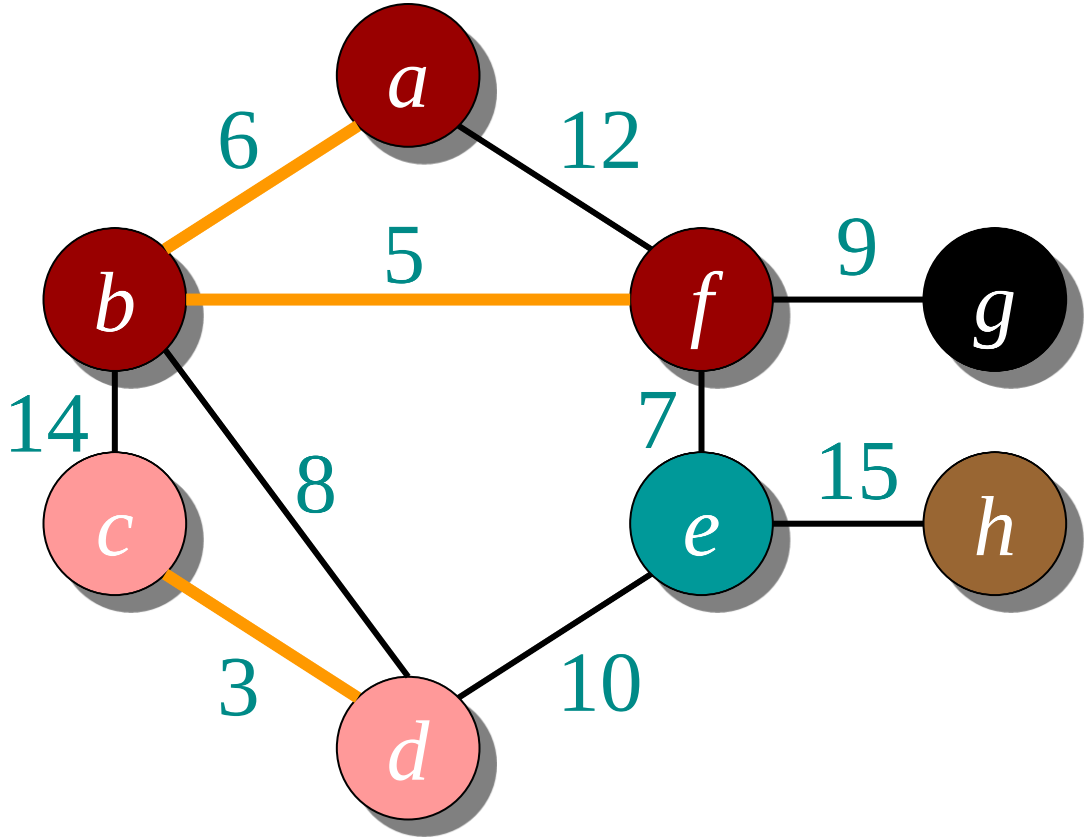
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

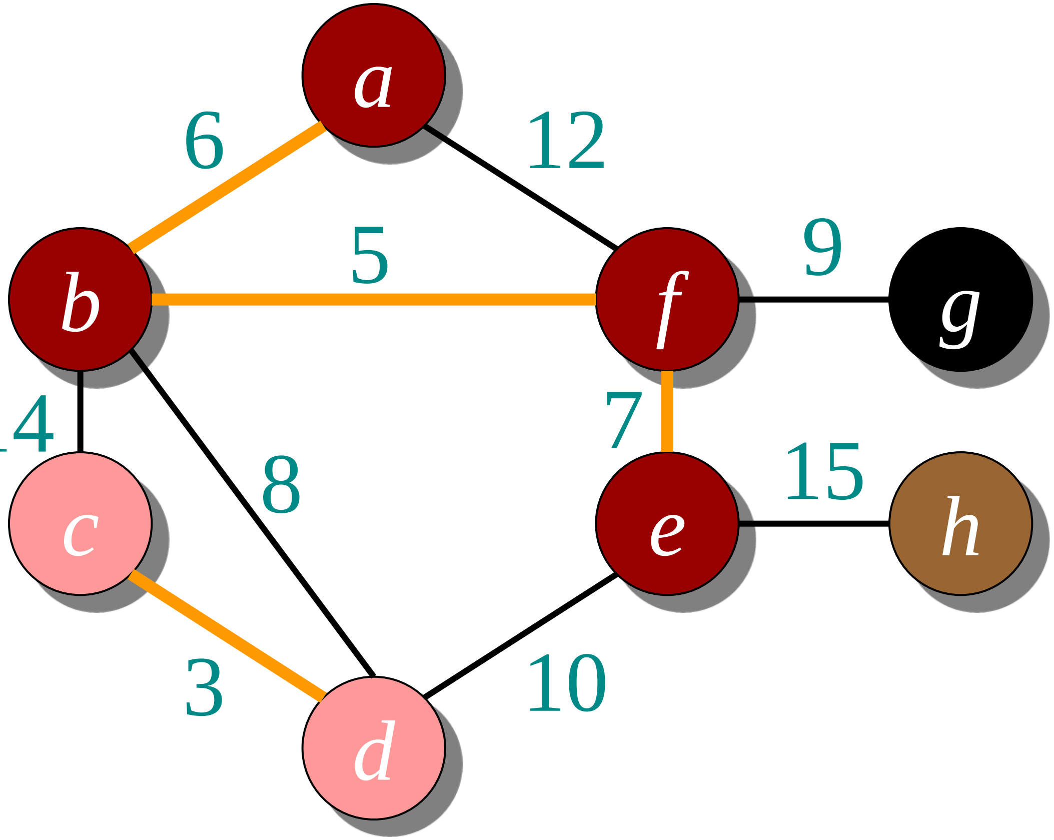
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

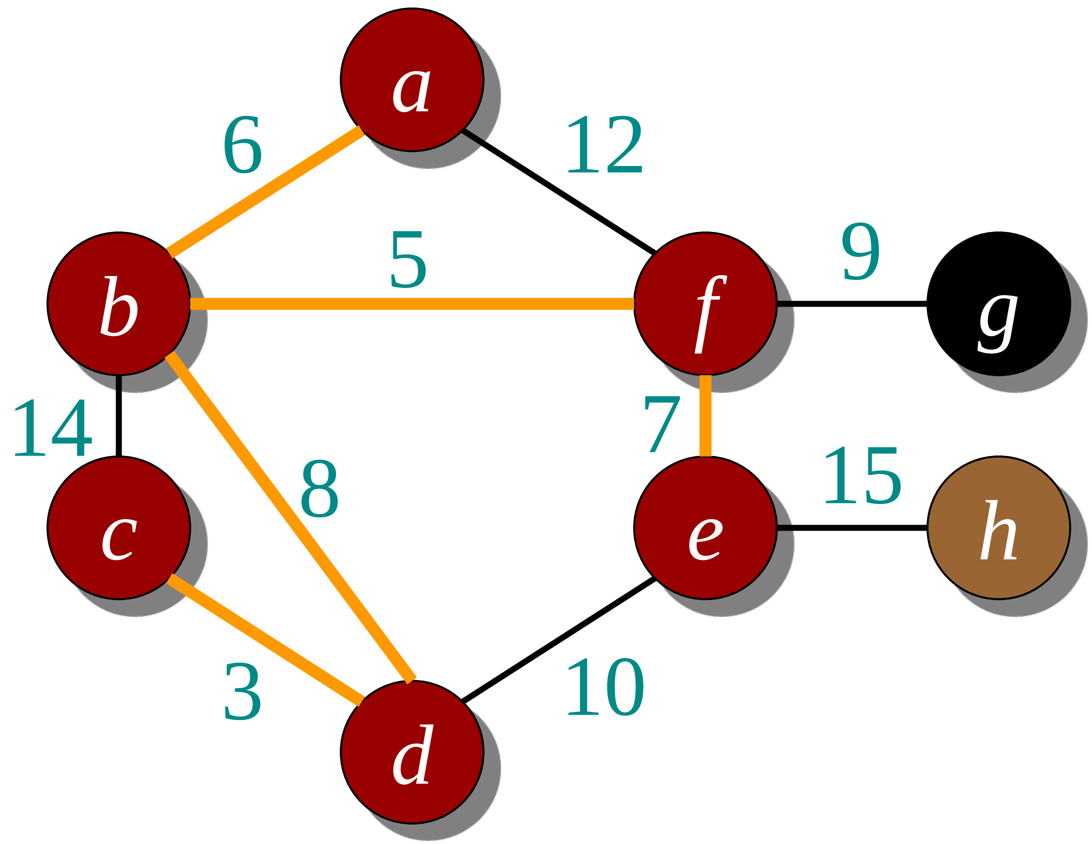
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

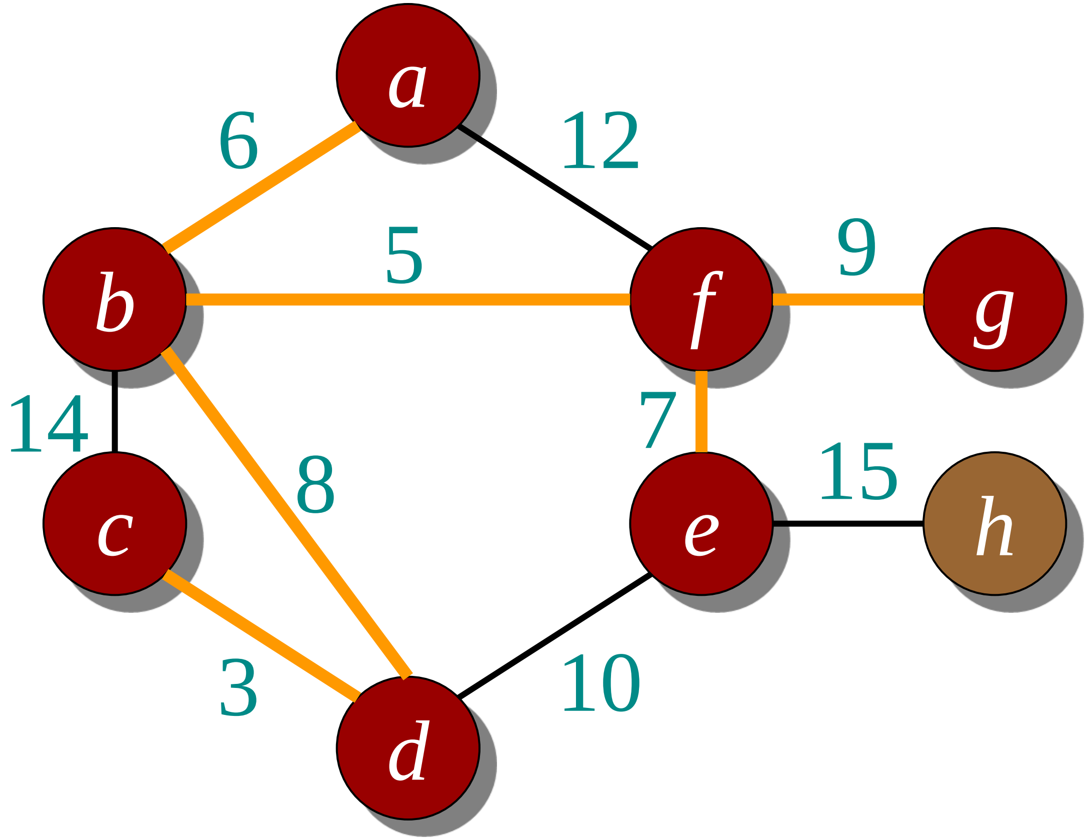
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

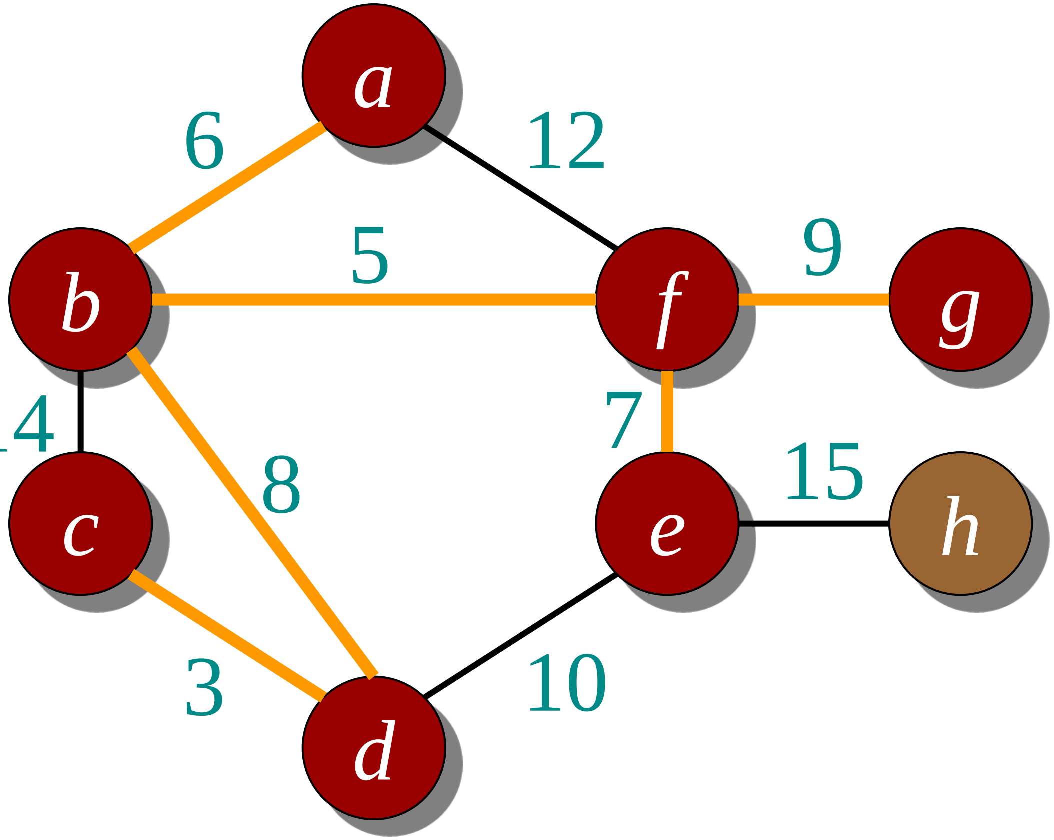
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

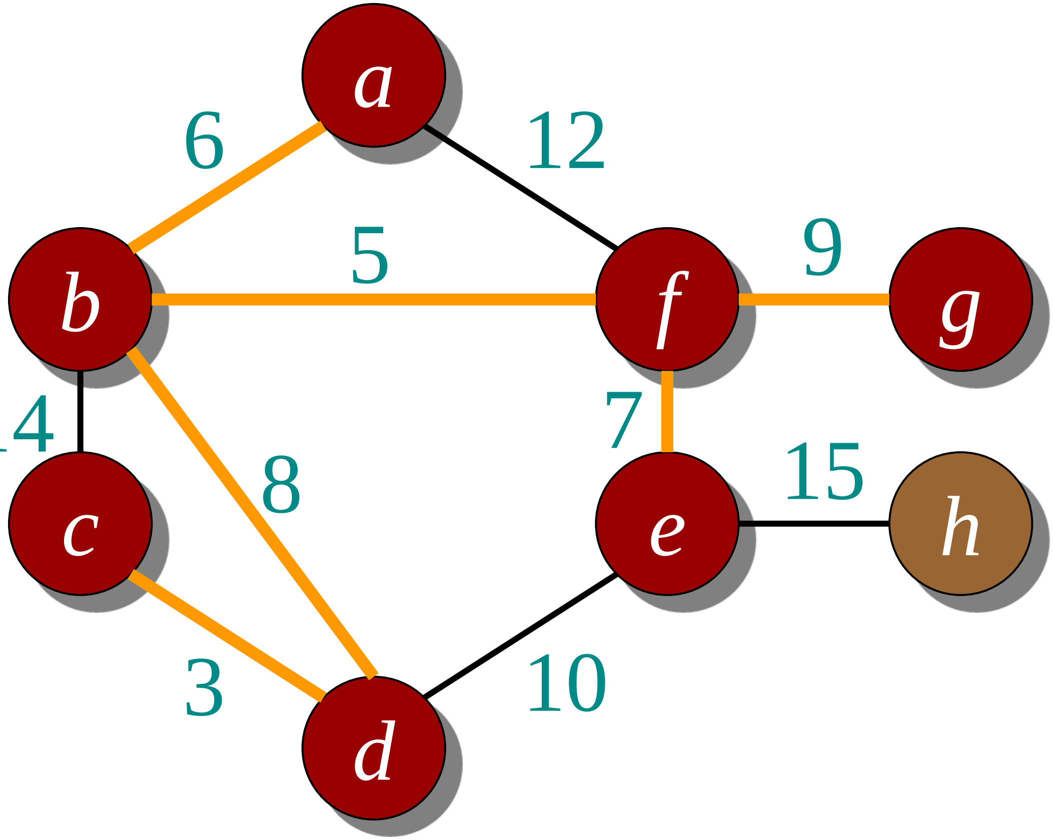
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

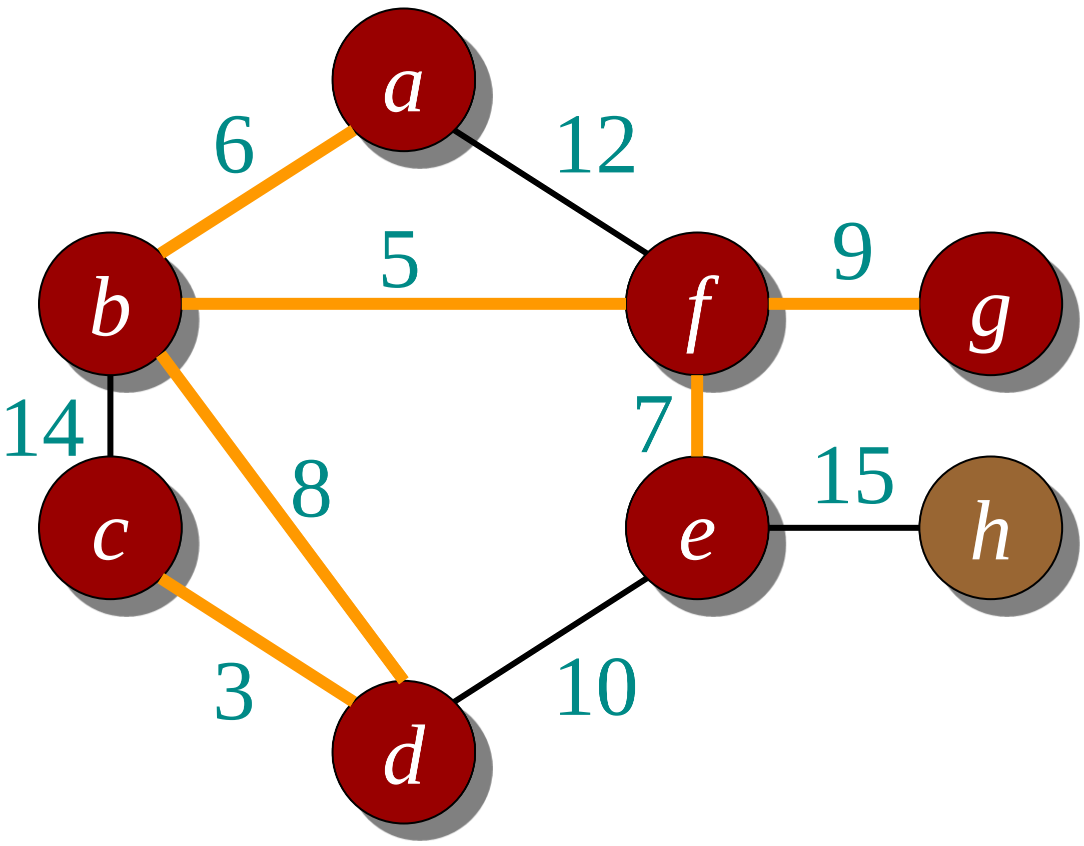
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

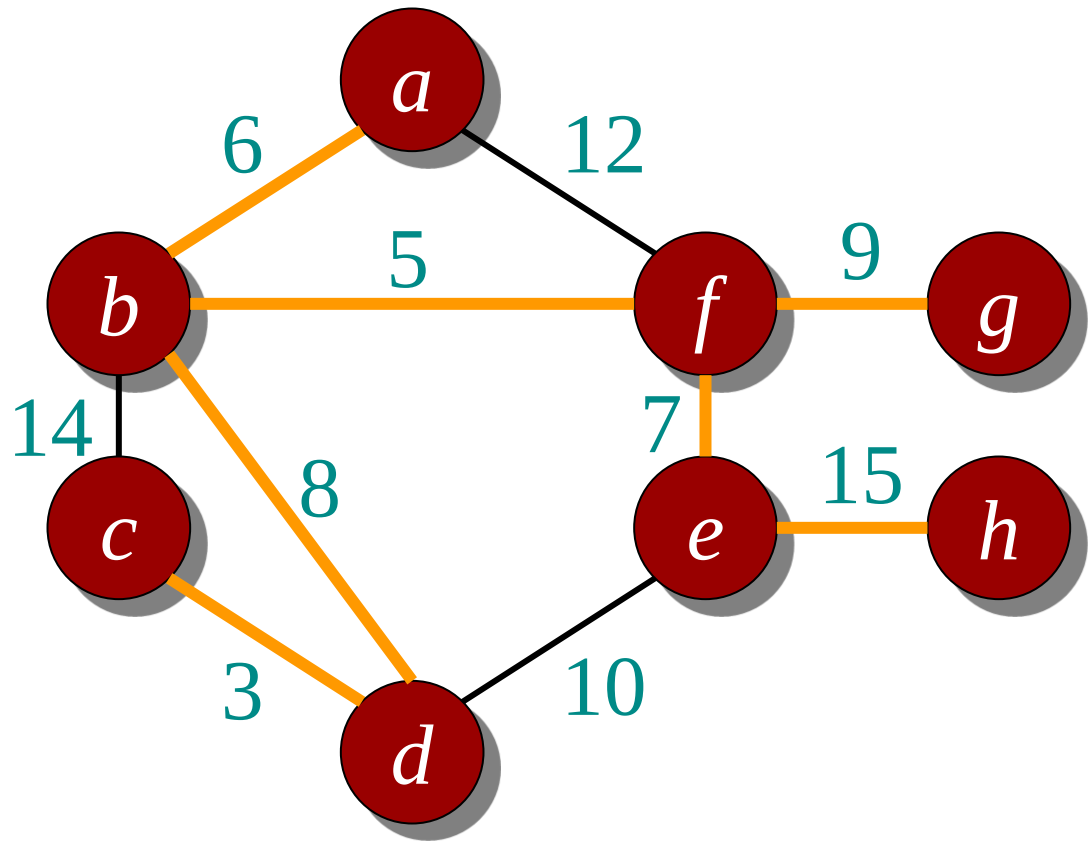
f-g: 9

d-e: 10

a-f: 12

b-c: 14

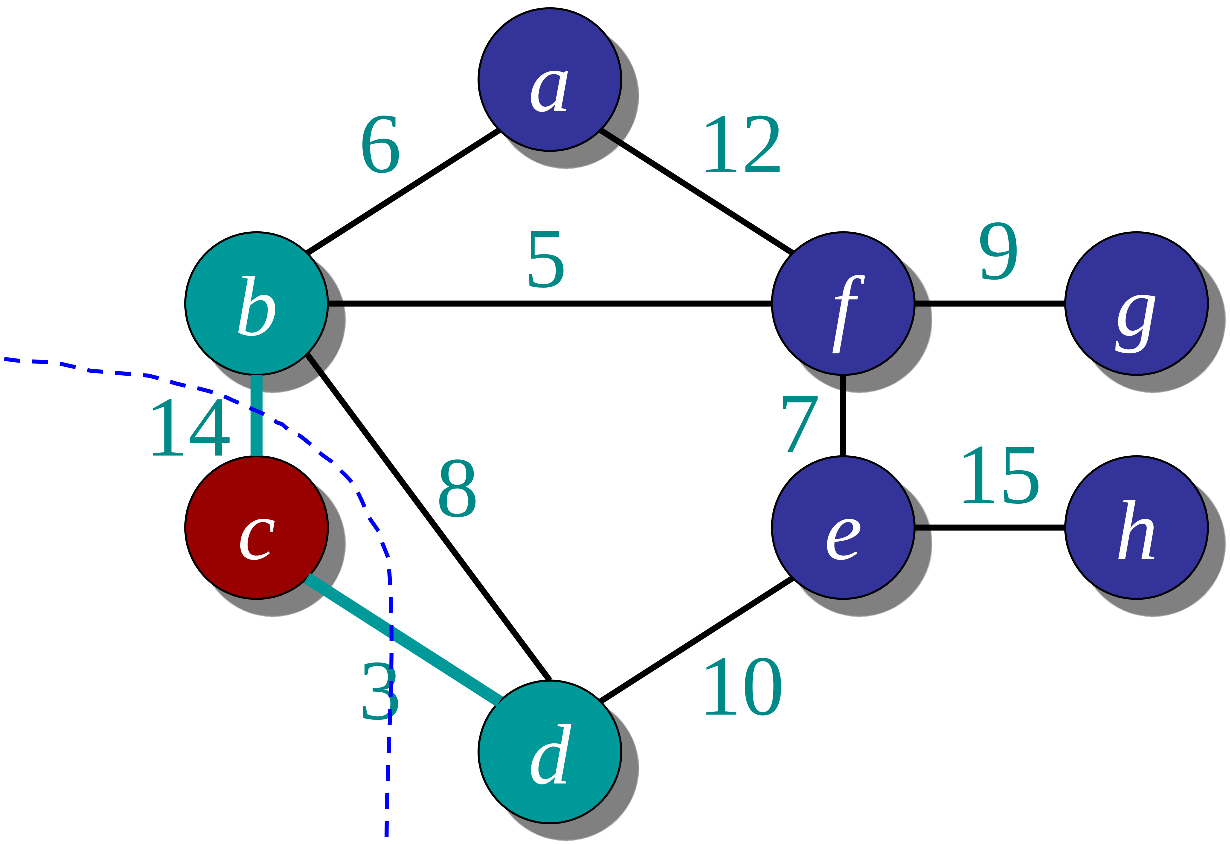
e-h: 15



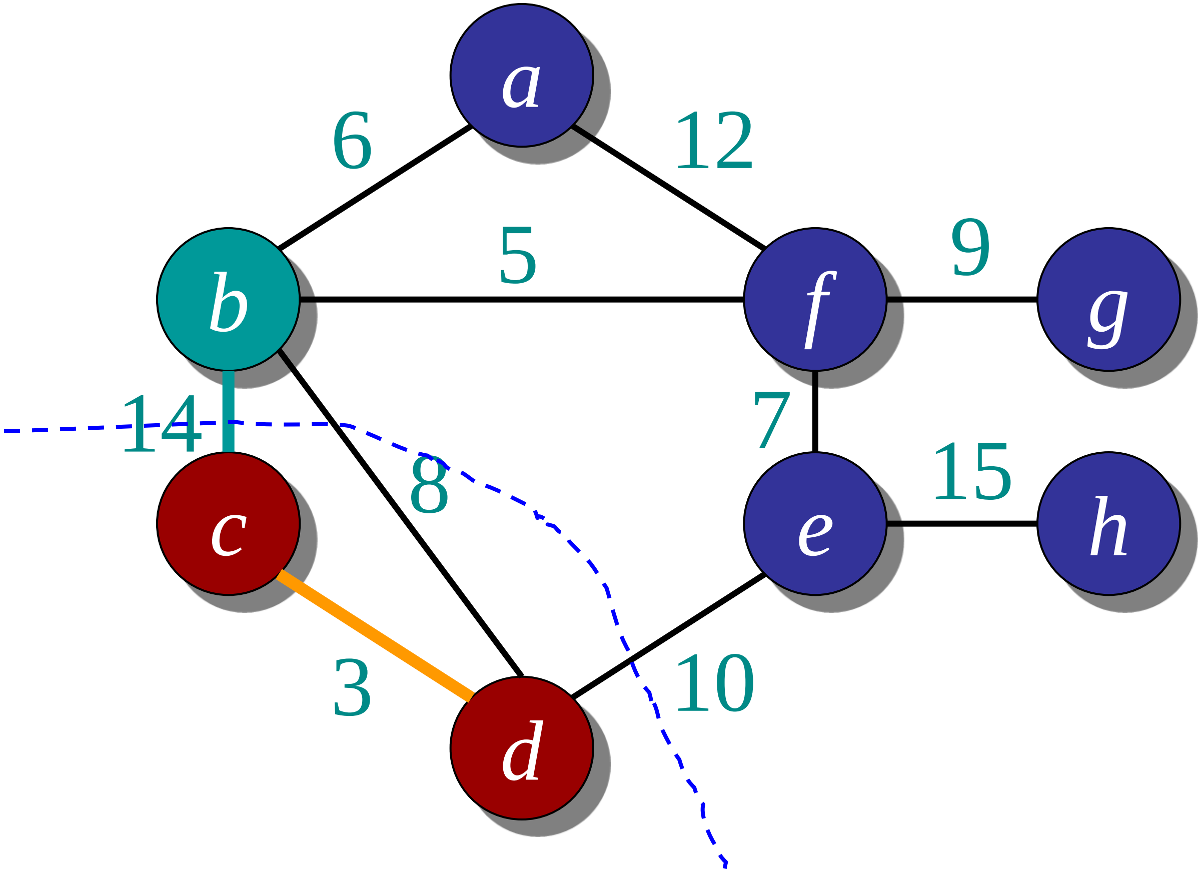
Back to Prim’s algorithm

- Randomly pick a vertex as the initial tree T
- Gradually expand into a MST:
  - For each vertex that is not in T but directly connected to some nodes in T
    - Compute its minimum distance to any vertex in T
  - Select the vertex that is closest to T
    - Add it to T

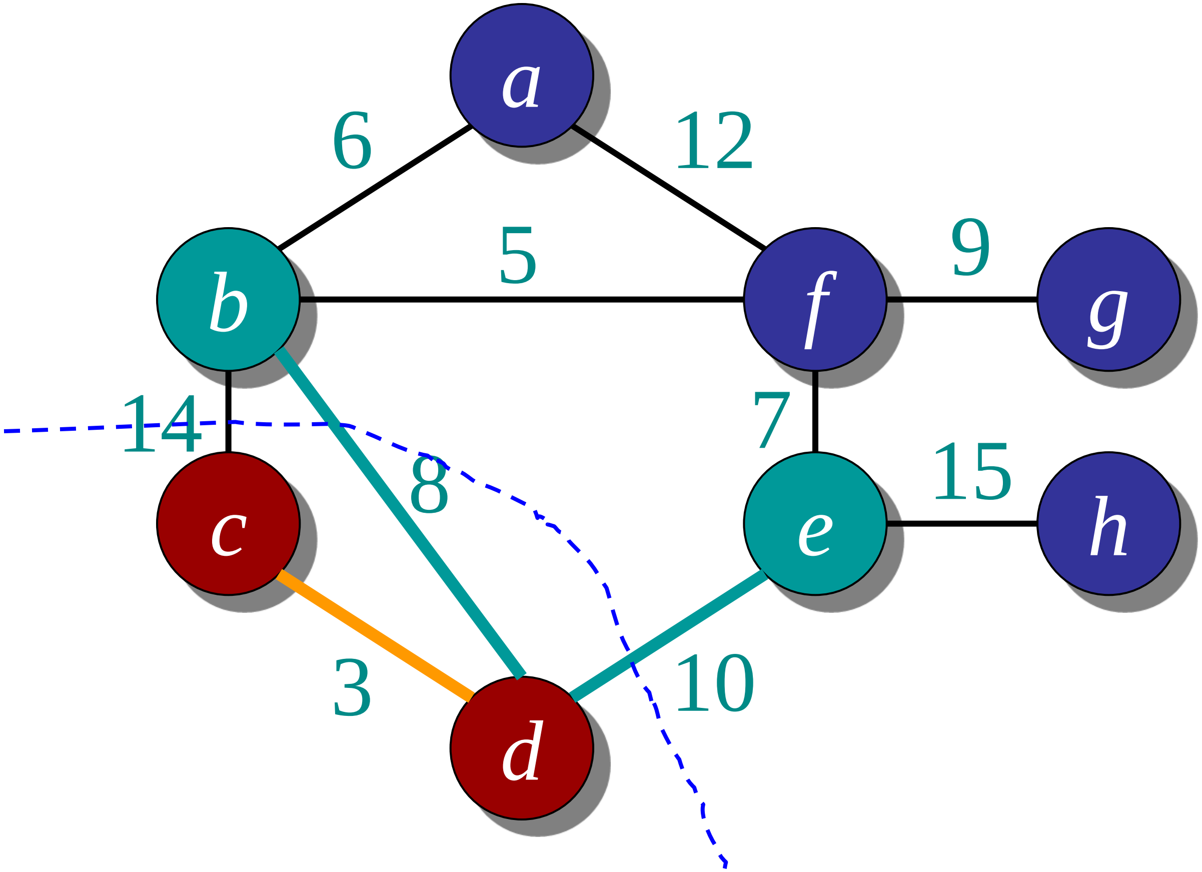
Example



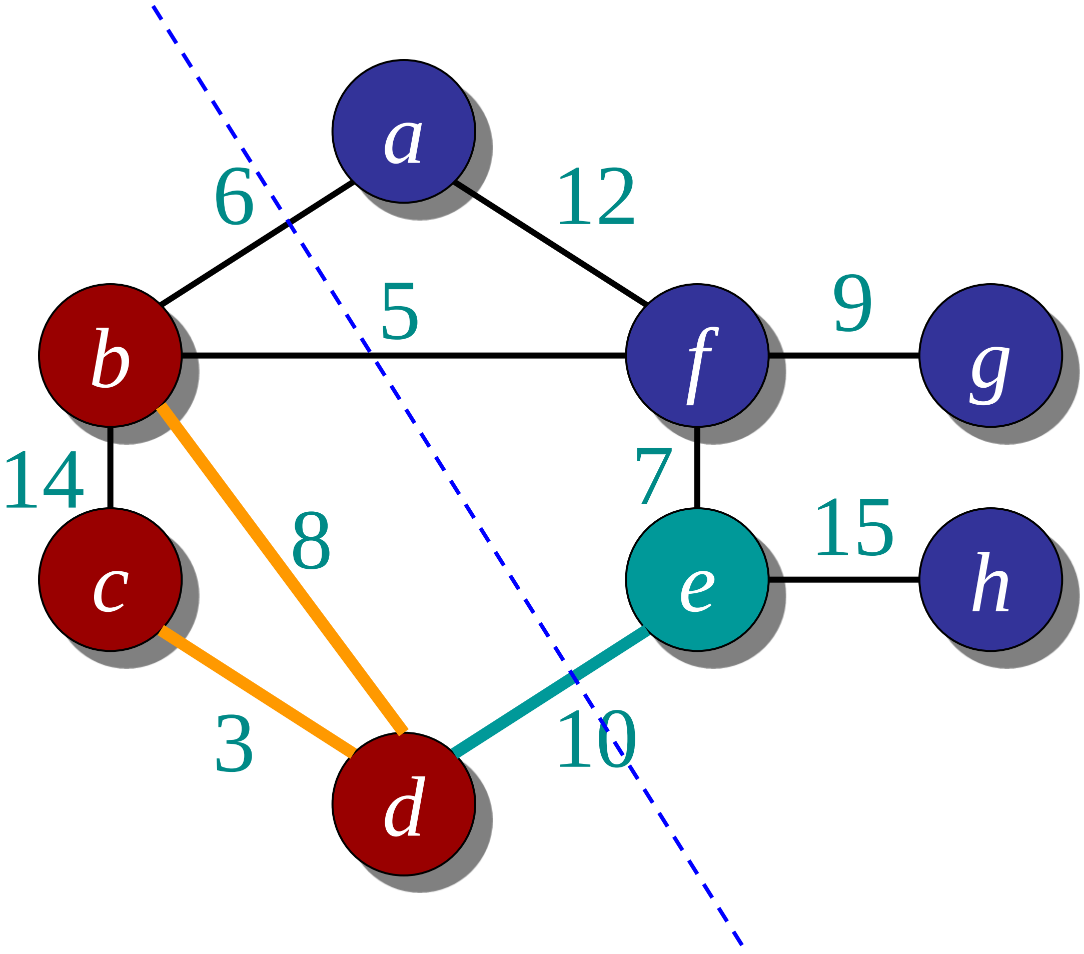
Example



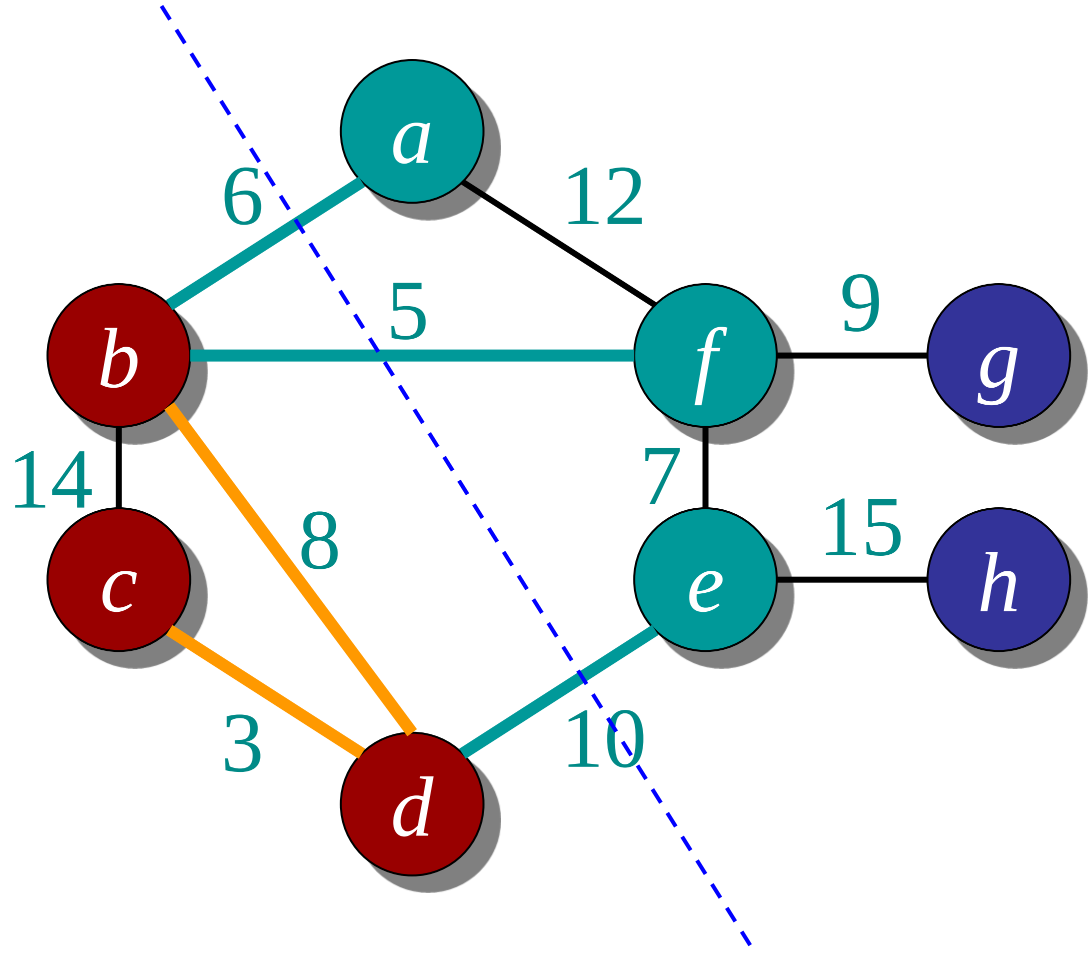
Example



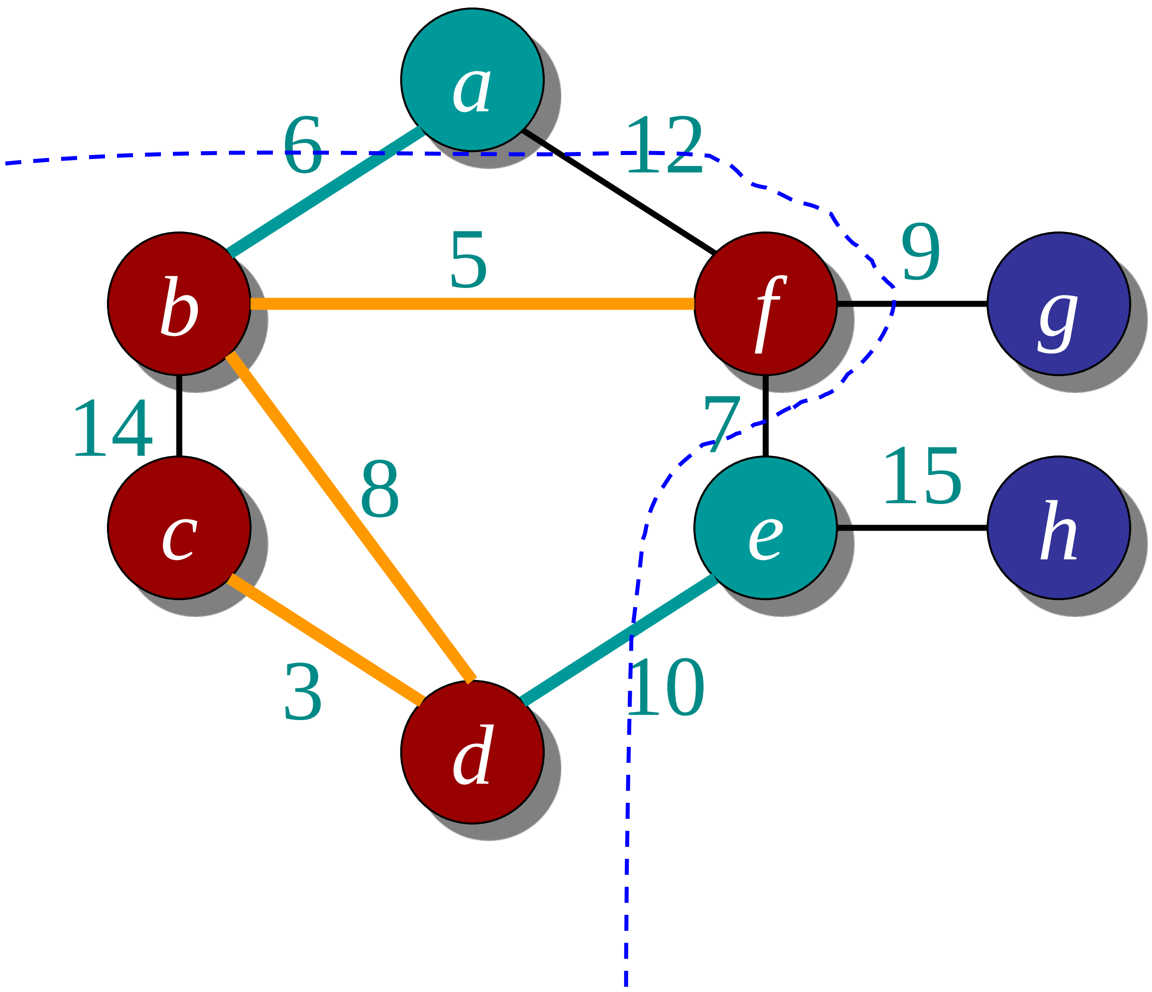
Example



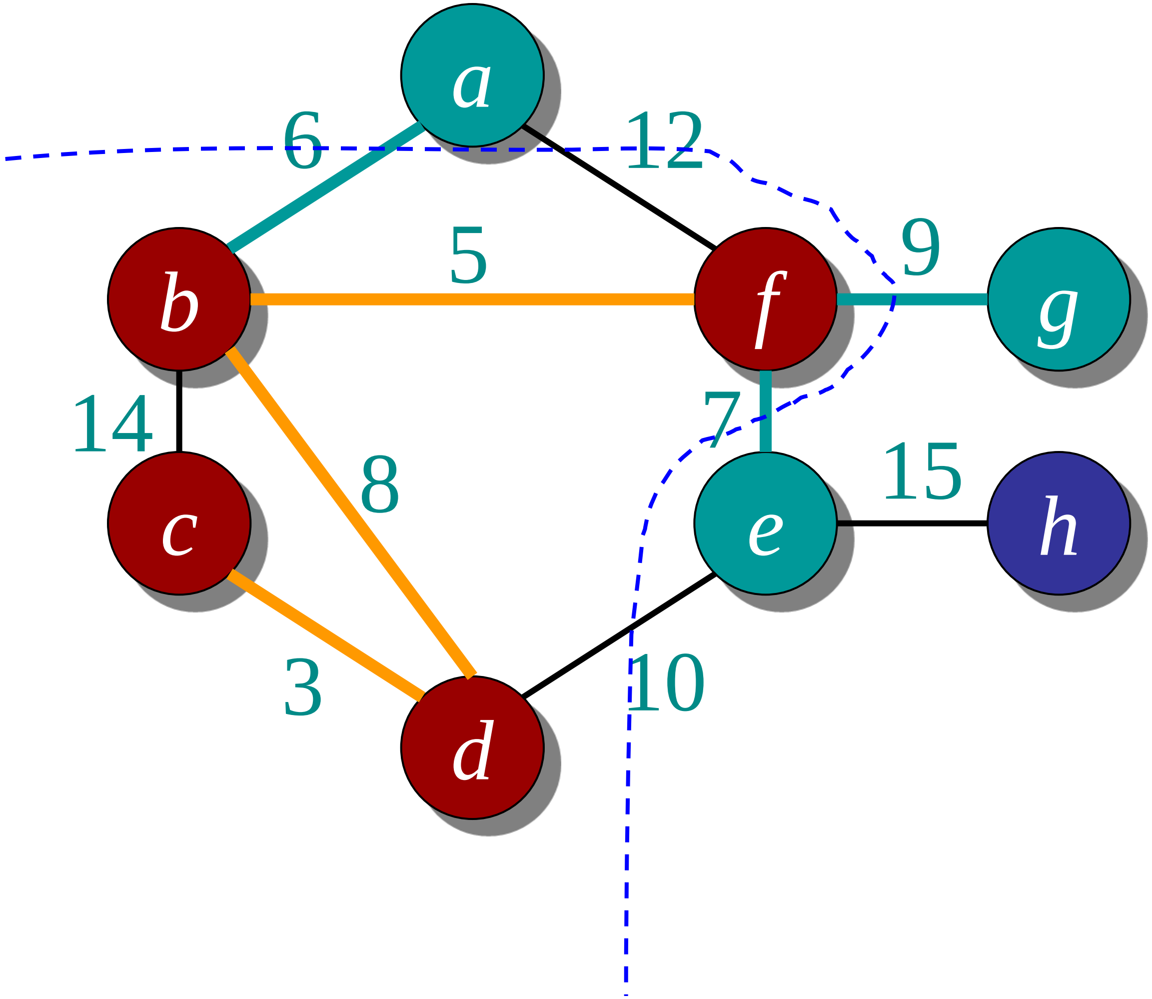
Example



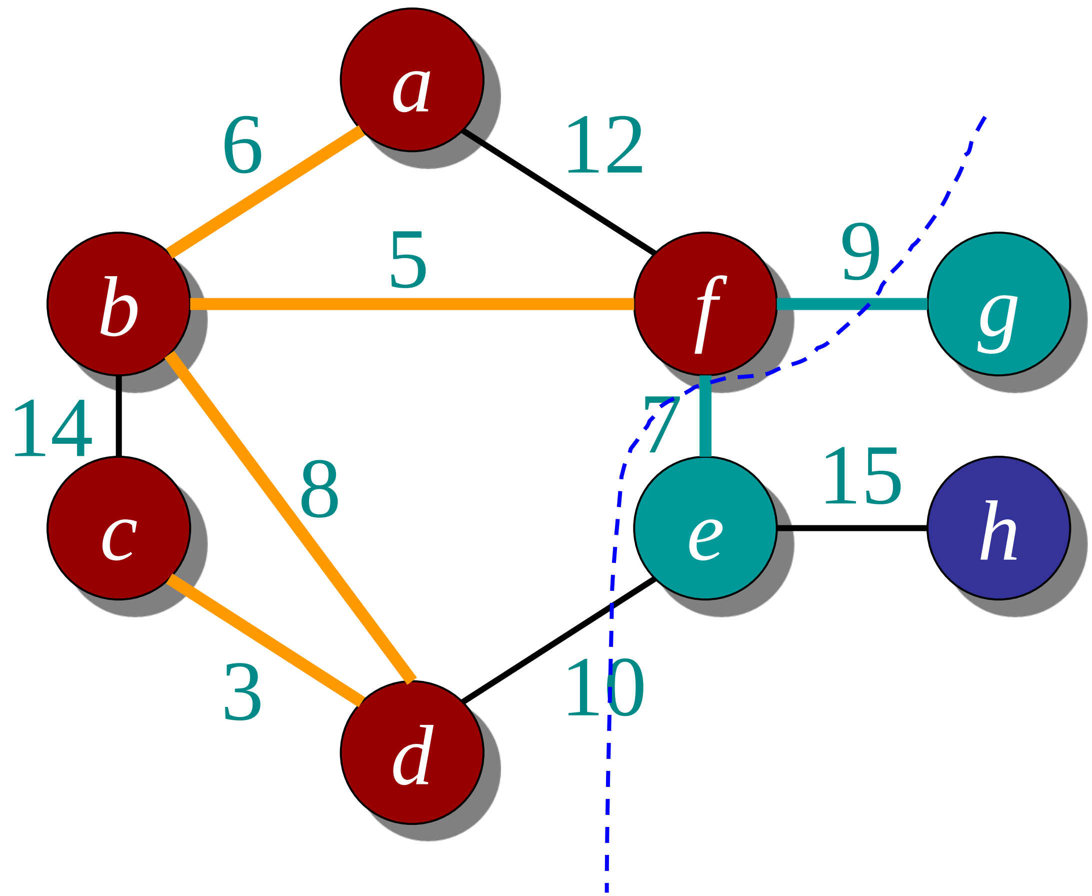
Example



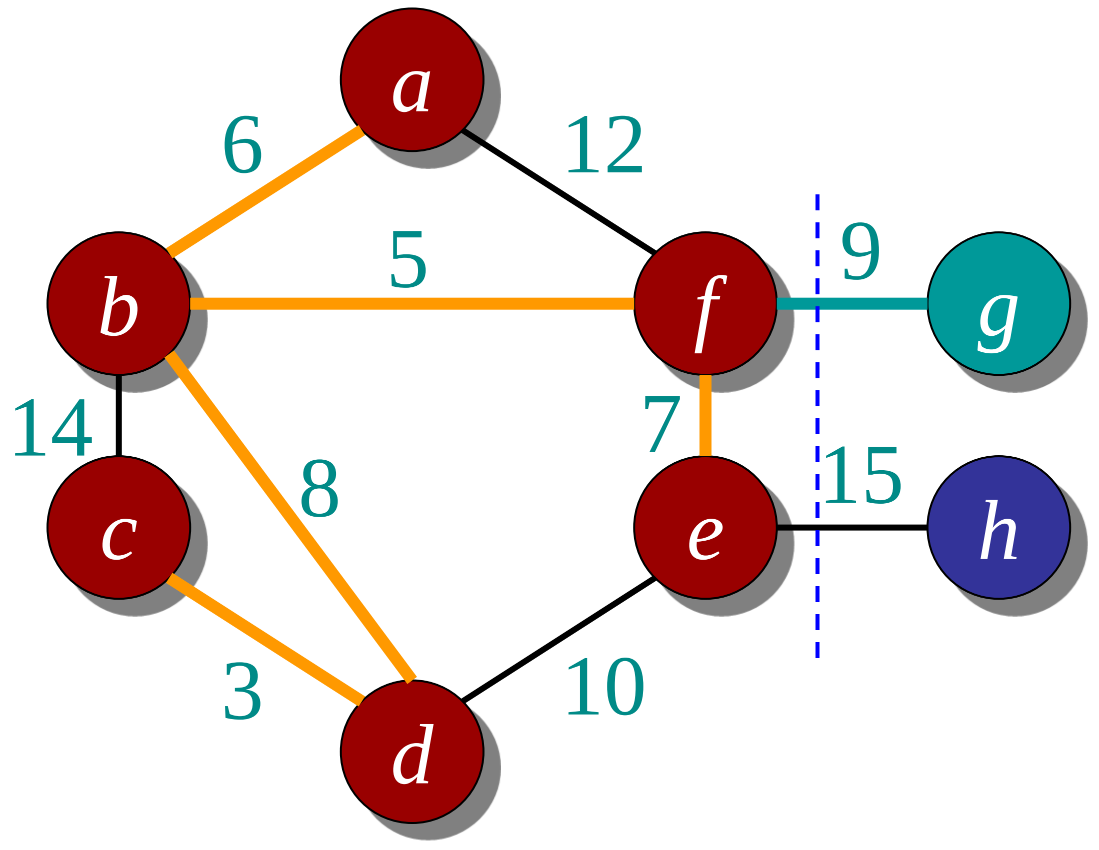
Example



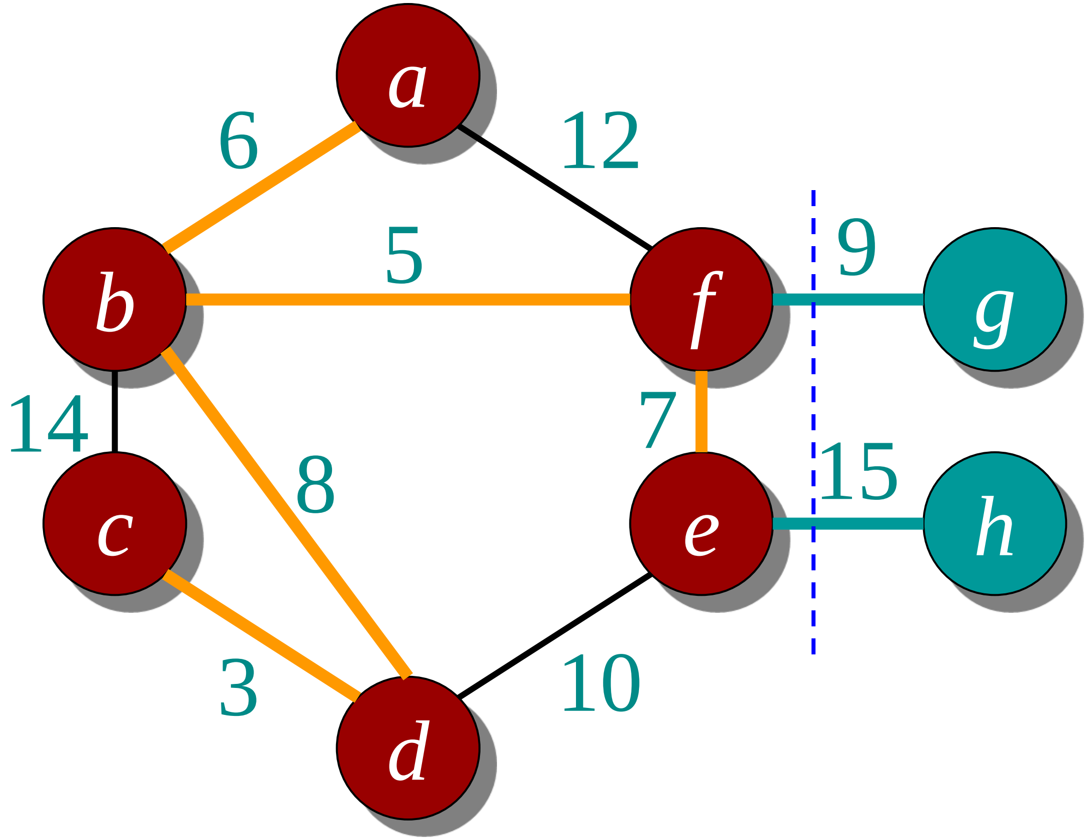
Example



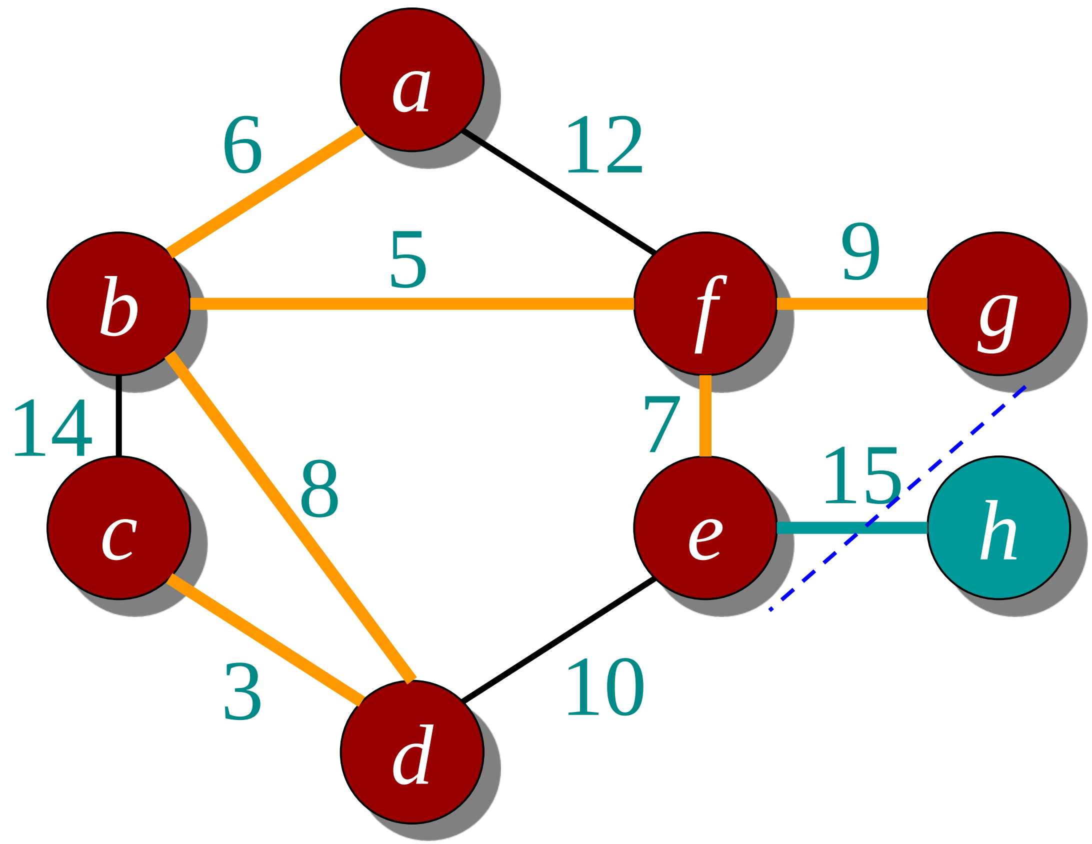
Example



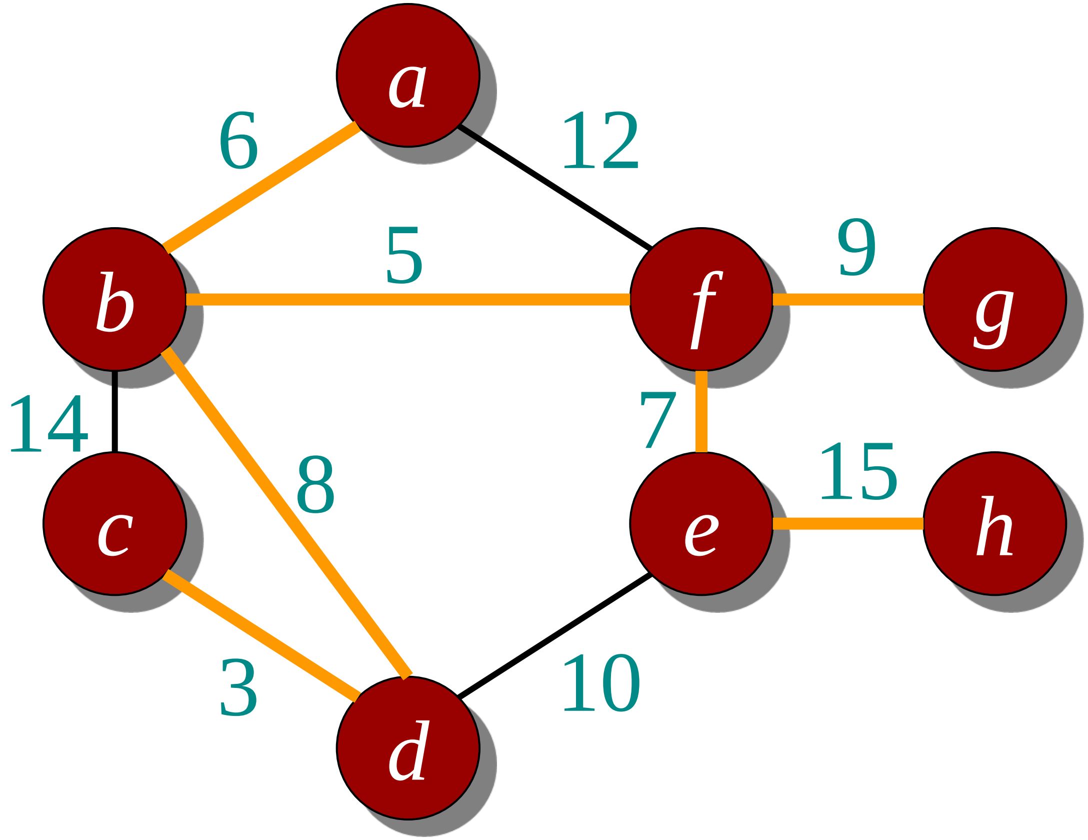
Example



Example



Example



Pseudocode for Prim's algorithm

Given $G = (V, E)$. Output: a MST T.

Randomly select a vertex v

$S = \{v\}$; $A = V \setminus S$; $T = \{\}$.

While (A is not empty)

find a vertex $u \in A$ that connects to vertex $v \in S$ such that $w(u, v) \leq w(x, y)$, for any $x \in A$ and $y \in S$

$S = S \cup \{u\}$; $A = A \setminus \{u\}$; $T = T \cup (u, v)$.

End

Return T

Time complexity

Given G = (V, E), |V|=n, |E|=m. Output: a MST T.

Randomly select a vertex v

S = {v}; A = V \ S; T = {}.

While (A is not empty) n vertices

find a vertex $u \in A$ that connects to vertex $v \in S$ such that $w(u, v) \leq w(x, y)$, for any $x \in A$ and $y \in S$

S = S U {u}; A = A \ {u}; T = T U (u, v).

End

Return T

Time complexity: n * (time spent on red line per vertex)

- Naïve: test all edges => $\Theta(n * m)$ which can be $\Theta(n^3)$ for dense graph
- Improve: keep the list of candidates in an array => $\Theta(n^2)$
- Better: with priority queue => $\Theta(m \log n)$

Idea 1: naive

find a vertex $u \in A$ that connects to vertex $v \in S$ such that $w(u, v) \leq w(x, y)$, for any $x \in A$ and $y \in S$

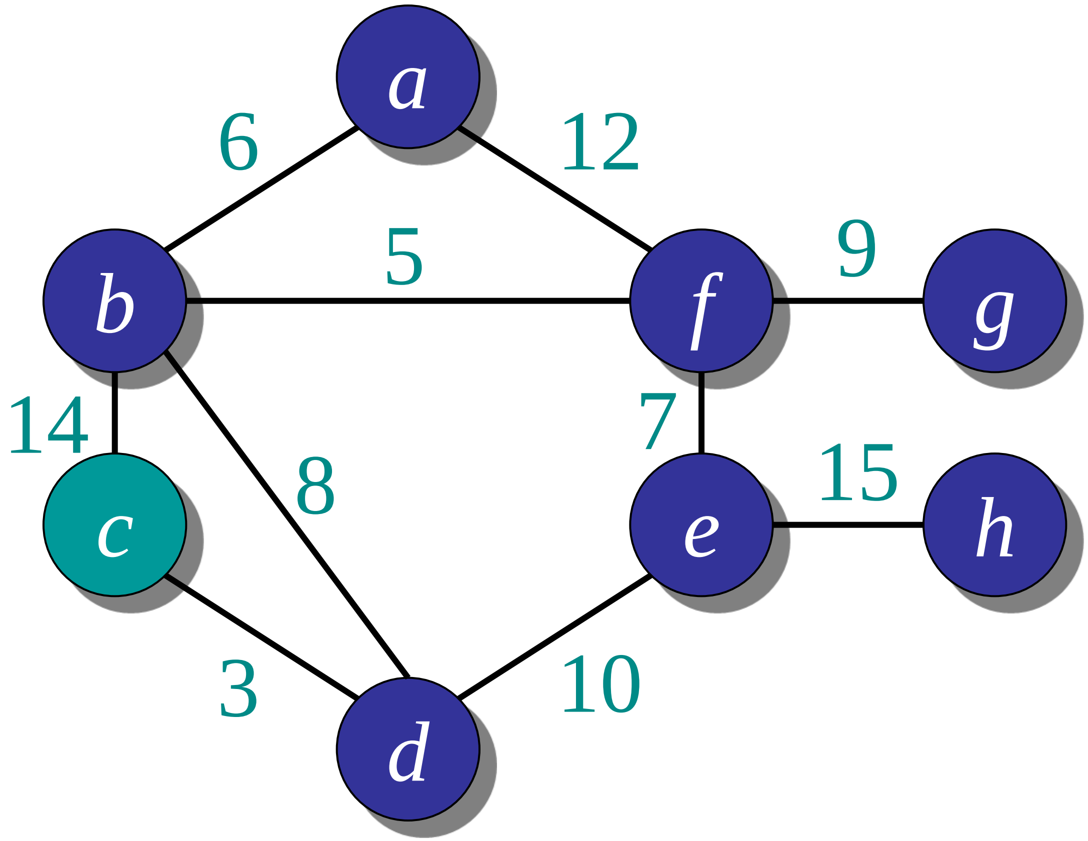
```
min_weight = infinity.
For each edge (x, y) ∈ E
        if x ∈ A, y ∈ S, and w(x, y) < min_weight
                u = x; v = y; min_weight = w(x, y);
```

time spent per vertex: $\Theta(m)$

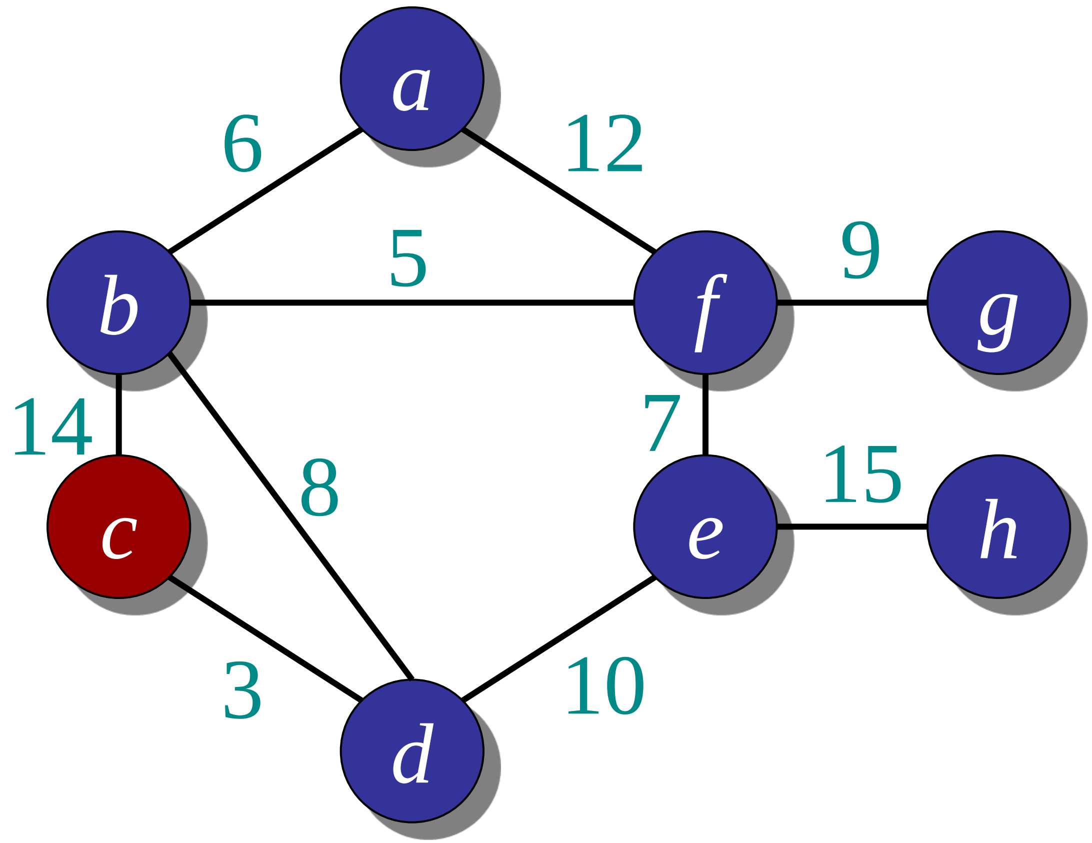
Total time complexity: $\Theta(n*m)$

Idea 2: distance array

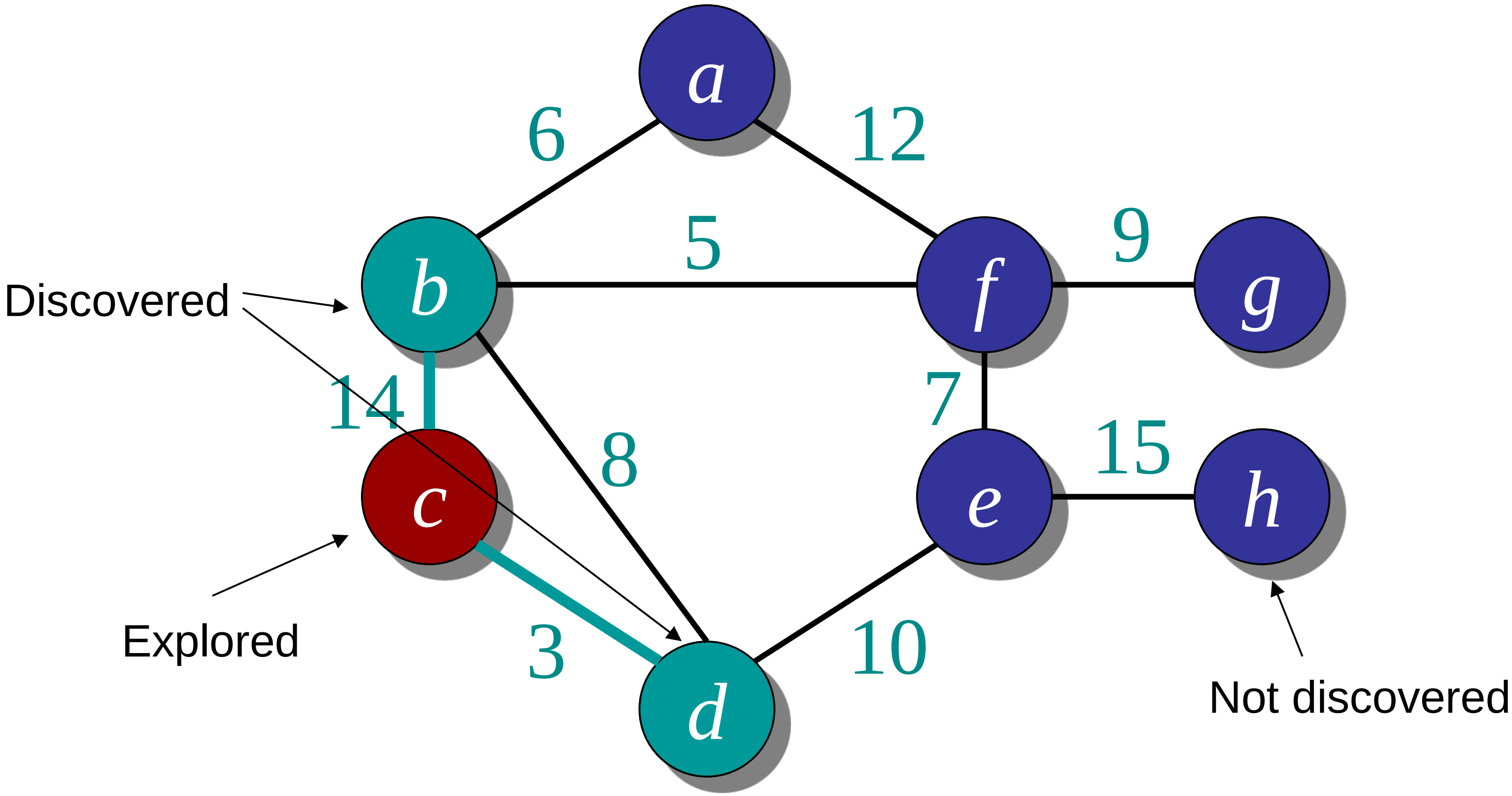
```
// For each vertex v, d[v] is the min distance from v to any node already in S
// p[v] is the parent node of v in the spanning tree
For each v ∈ V
        d[v] = infinity; p[v] = null;
Randomly select a v, d[v] = 1; // d[v]=1 just to ensure proper start
S = {}; A = V; T = {}.
While (A is not empty)
        Search d to find the smallest d[u] > 0.
        S = S U {u}; A = A \ {u}; T = T U (u, p[u]);
        d[u] = 0.
        For each v in adj[u]
                if  d[v] > w(u, v)
                        d[v] = w(u, v);
                        p[v] = u;
End
```



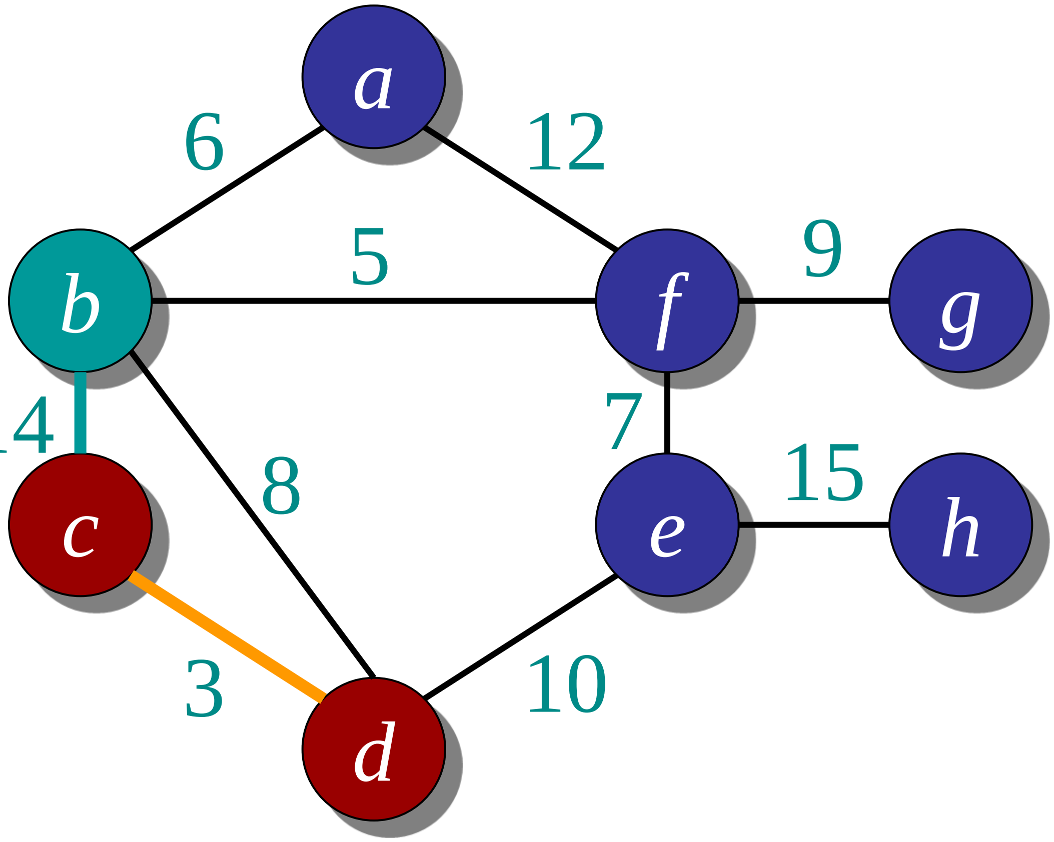
	a	b	c	d	e	f	g	h
d	∞	∞	1	∞	∞	∞	∞	∞
p	/	/	/	/	/	/	/	/



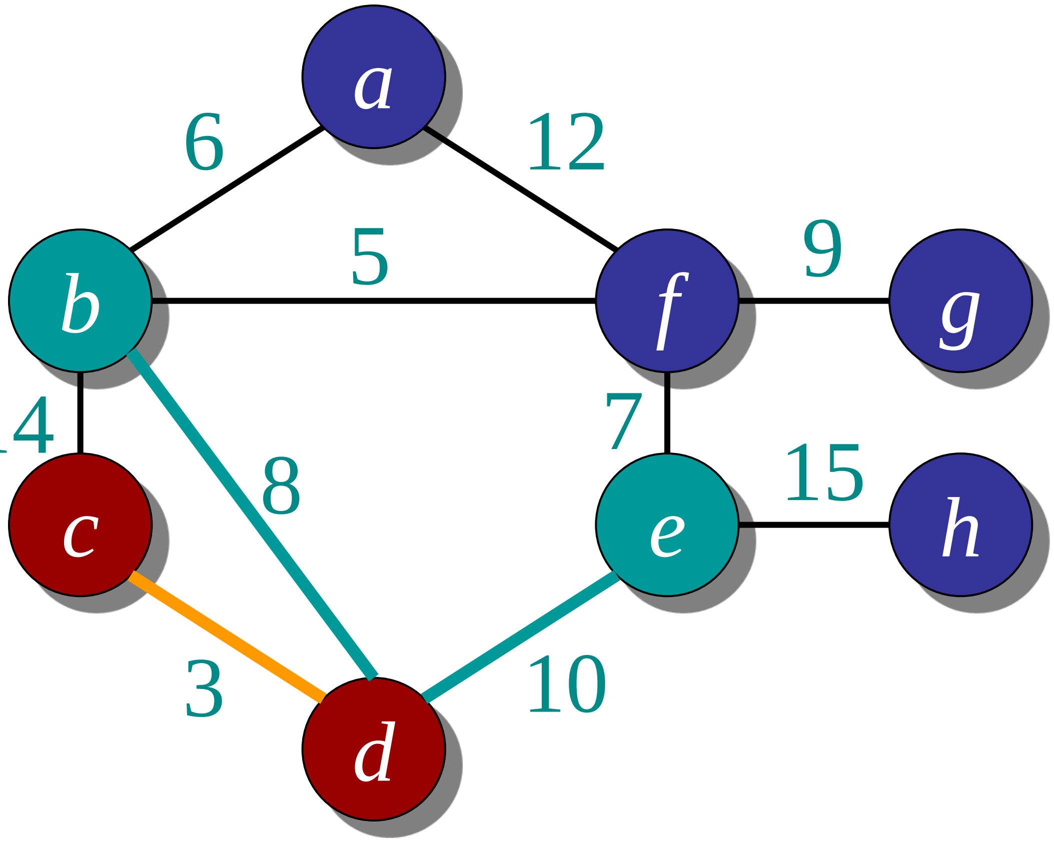
	a	b	c	d	e	f	g	h
d	∞	∞	0	∞	∞	∞	∞	∞
p	/	/	/	/	/	/	/	/



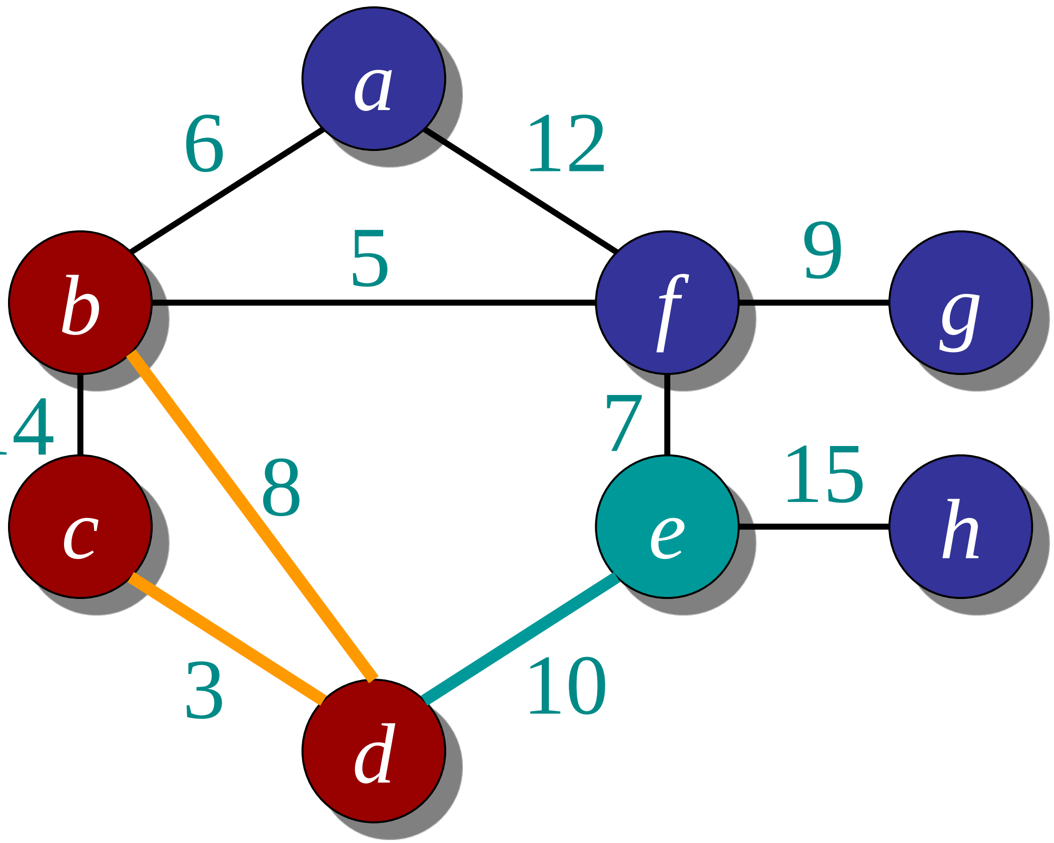
	a	b	c	d	e	f	g	h
d	∞	14	0	3	∞	∞	∞	∞
p	/	c	/	c	/	/	/	/



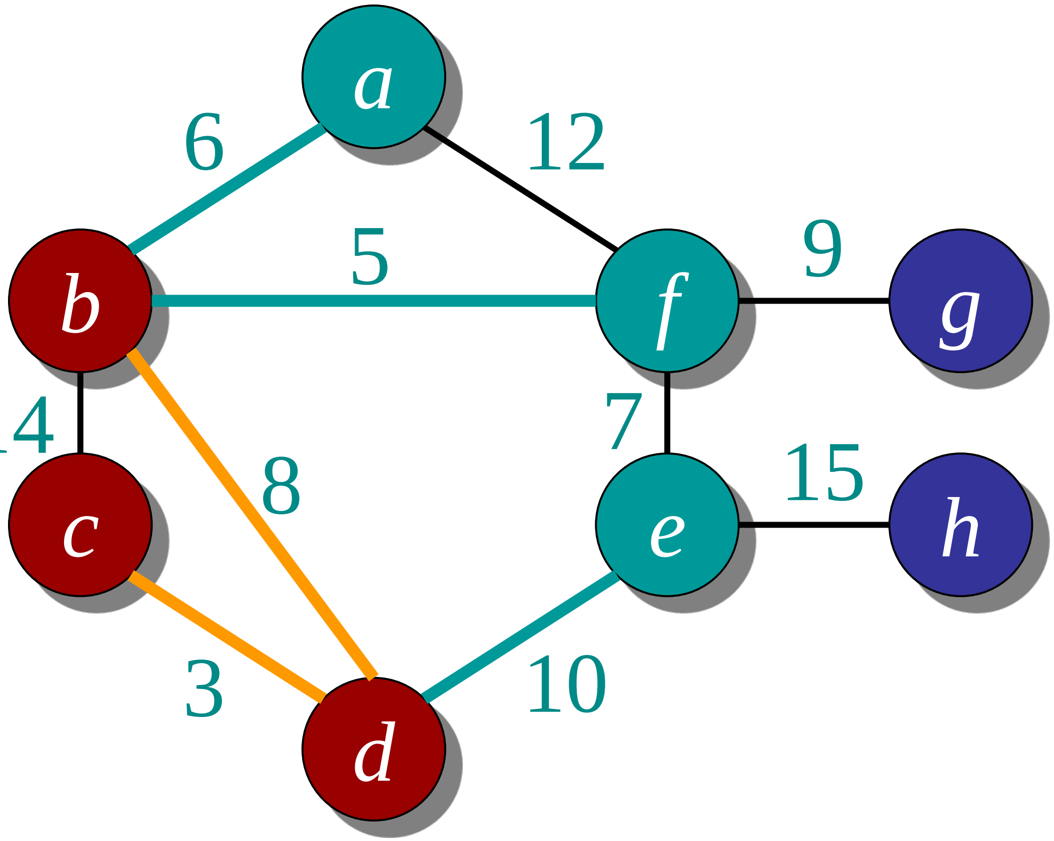
	a	b	c	d	e	f	g	h
d	∞	14	0	0	∞	∞	∞	∞
p	/	c	/	c	/	/	/	/



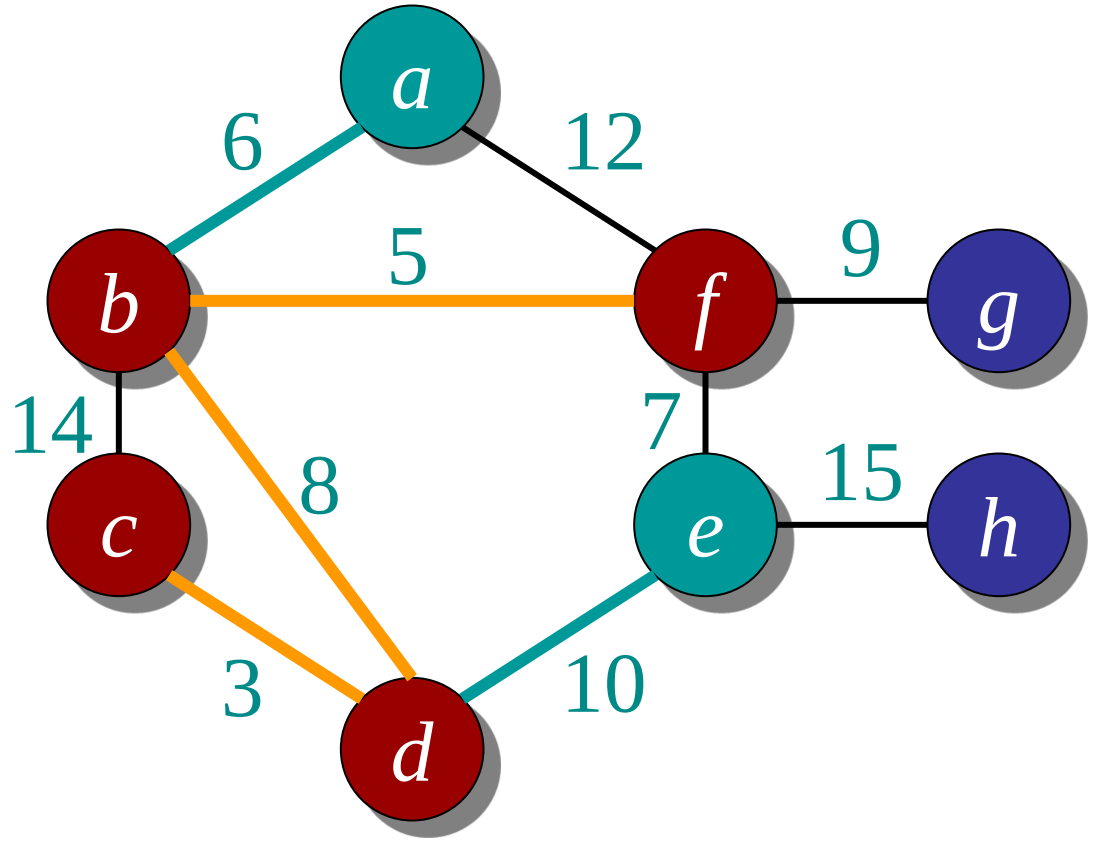
	a	b	c	d	e	f	g	h
d	∞	8	0	0	10	∞	∞	∞
p	/	d	/	c	d	/	/	/



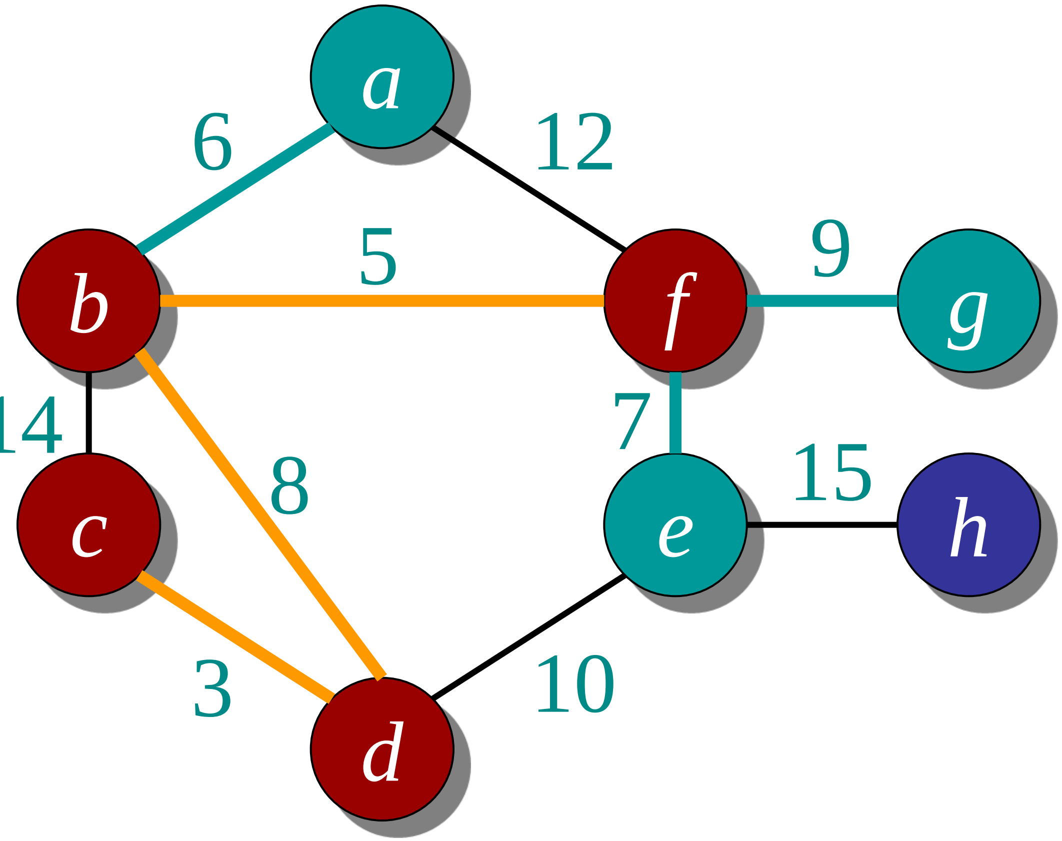
	a	b	c	d	e	f	g	h
d	∞	0	0	0	10	∞	∞	∞
p	/	d	/	c	d	/	/	/



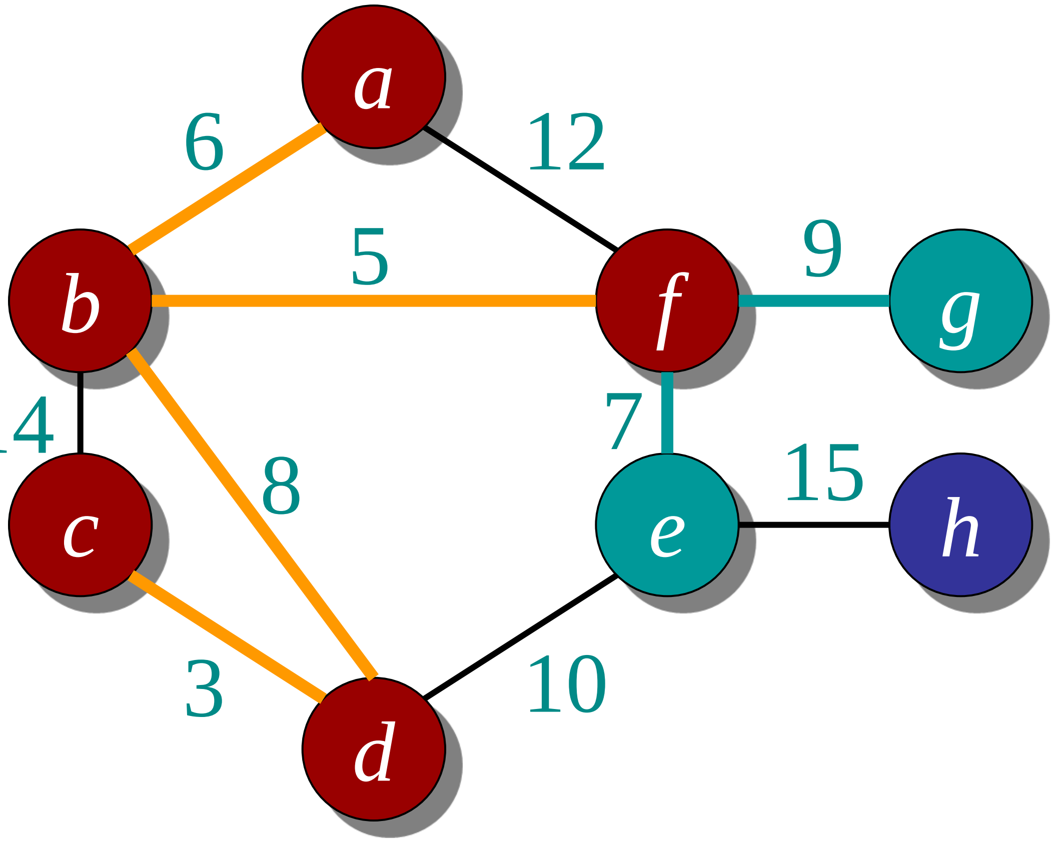
	a	b	c	d	e	f	g	h
d	6	0	0	0	10	5	∞	∞
p	b	d	/	c	d	b	/	/



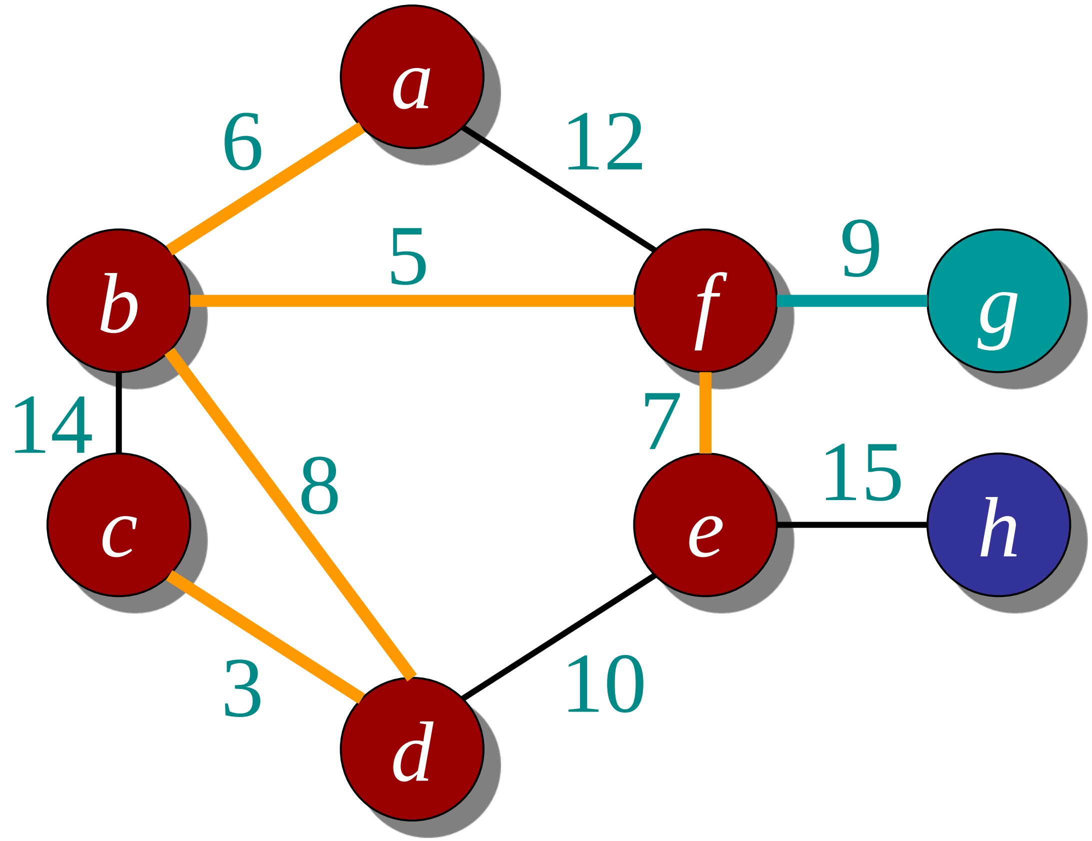
	a	b	c	d	e	f	g	h
d	6	0	0	0	10	0	∞	∞
p	b	d	/	c	d	b	/	/



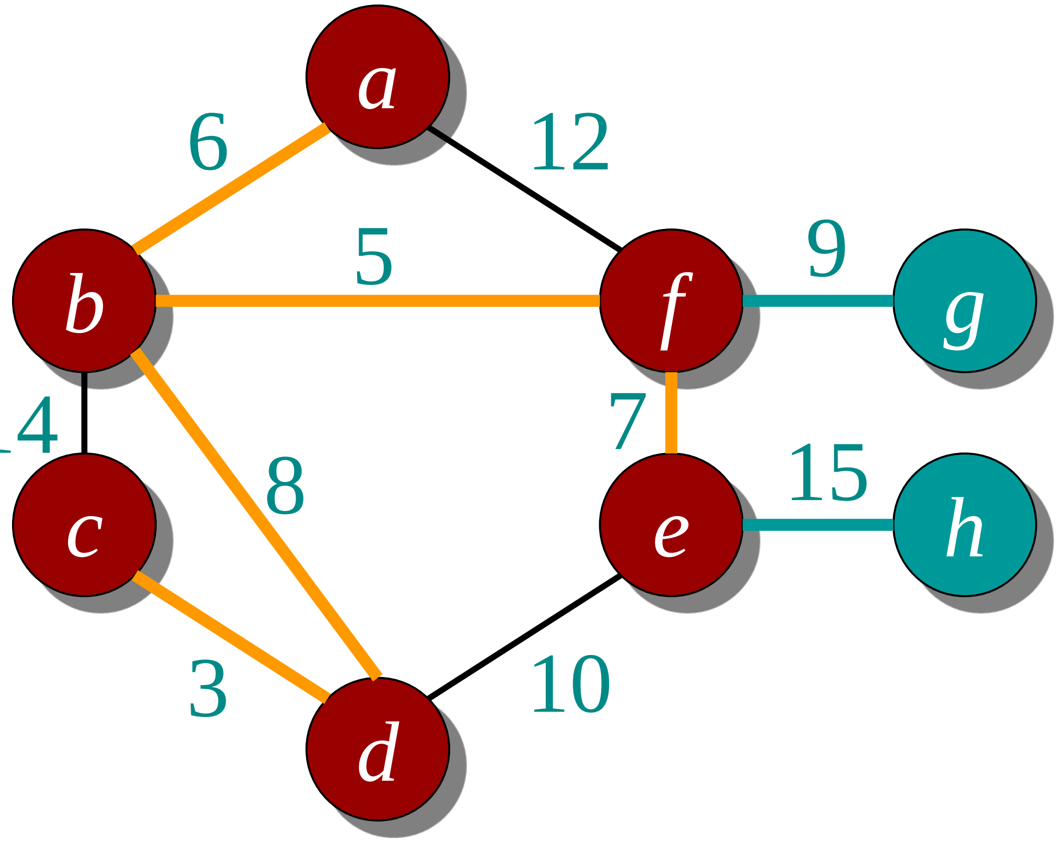
	a	b	c	d	e	f	g	h
d	6	0	0	0	7	0	9	∞
p	b	d	/	c	f	b	f	/



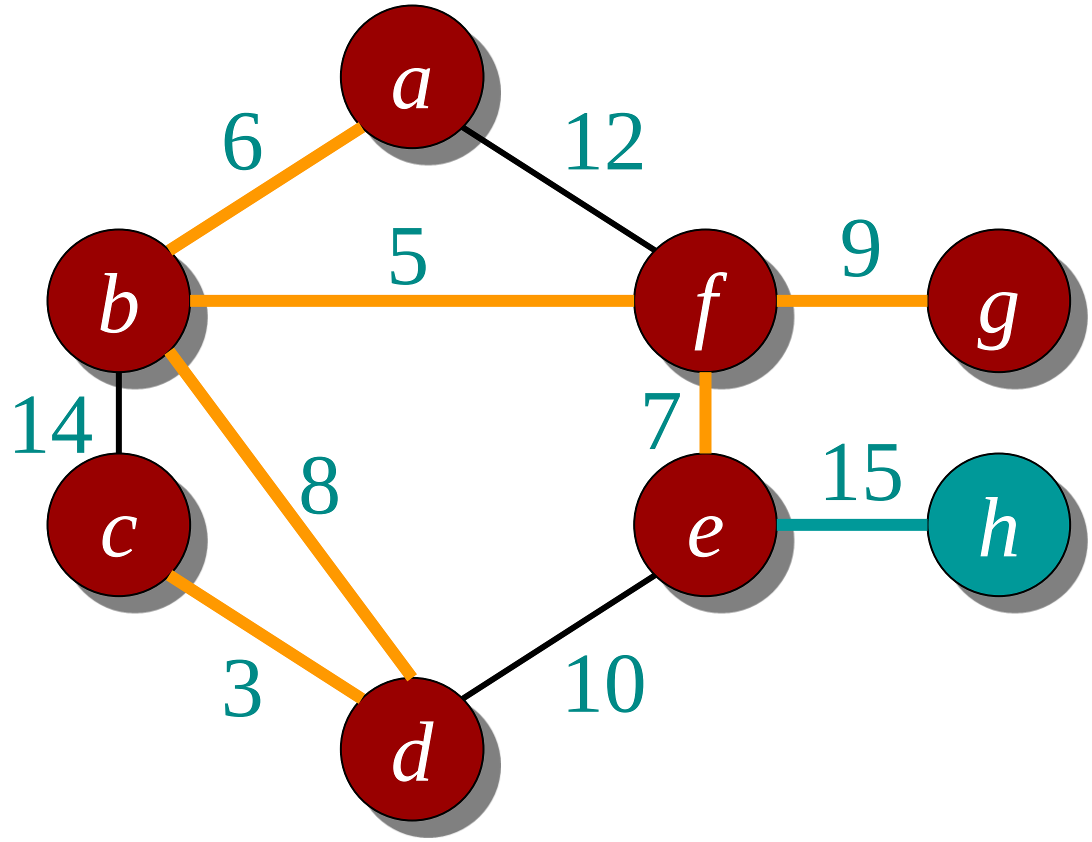
	a	b	c	d	e	f	g	h
d	0	0	0	0	7	0	9	∞
p	b	d	/	c	f	b	f	/



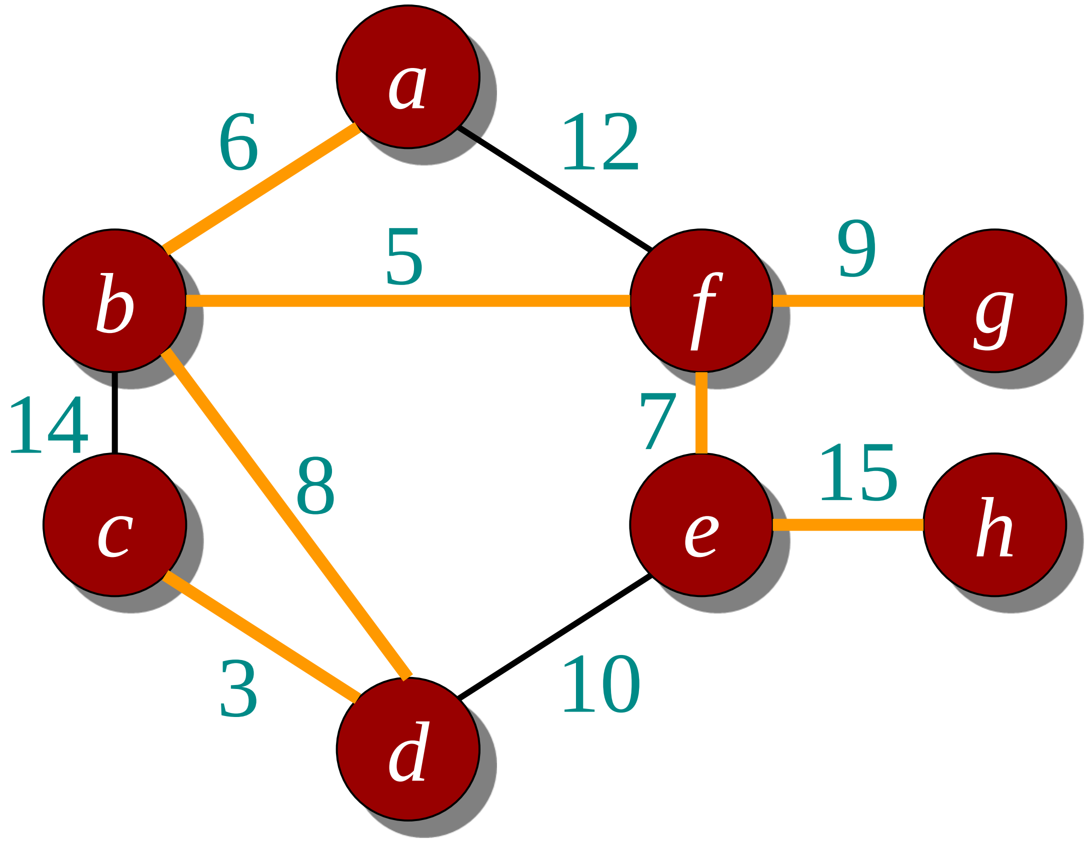
	a	b	c	d	e	f	g	h
d	0	0	0	0	0	0	9	∞
p	b	d	/	c	f	b	f	/



	a	b	c	d	e	f	g	h
d	0	0	0	0	0	0	9	15
p	b	d	/	c	f	b	f	e



	a	b	c	d	e	f	g	h
d	0	0	0	0	0	0	0	15
p	b	d	/	c	f	b	f	e



	a	b	c	d	e	f	g	h
d	0	0	0	0	0	0	0	0
p	b	d	/	c	f	b	f	e

Time complexity?

// For each vertex v, d[v] is the min distance from v to any node already in S

// p[v] is the parent node of v in the spanning tree

For each v $\in$ V

    d[v] = infinity; p[v] = null;

Randomly select a v, d[v] = 1; // d[v]=1 just to ensure proper start

S = {}. T = {}. A = V.

While (A is not empty)

    Search d to find the smallest d[u] > 0.

    S = S U {u}; A = A \ {u}; T = T U (u, p[u]);

    d[u] = 0.

    For each v in adj[u]

        if d[v] > w(u, v)

            d[v] = w(u, v);

            p[v] = u;

End

n vertices

Θ(n) per vertex

O(n) per vertex if using adj list

Θ(n) per vertex if using adj matrix

Overall complexity: $\Theta(n^2)$

Idea 3: priority queue (min-heap)

- Observation
  - In idea 2, we need to search the distance array n times, each time we only look for the minimum element.
  - Distance array size = n
  - $n \times n = n^2$
- Can we do better?
  - Priority queue (heap) enables fast retrieval of min (or max) elements
  - log (n) for most operations

Complete Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
        p[u] = null;
    key[r] = 0;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                ChangeKey(v, w(u,v));
```

Overall running time: ***Θ(m log n)***

Cost per ChangeKey

n vertices

Θ(log n) times

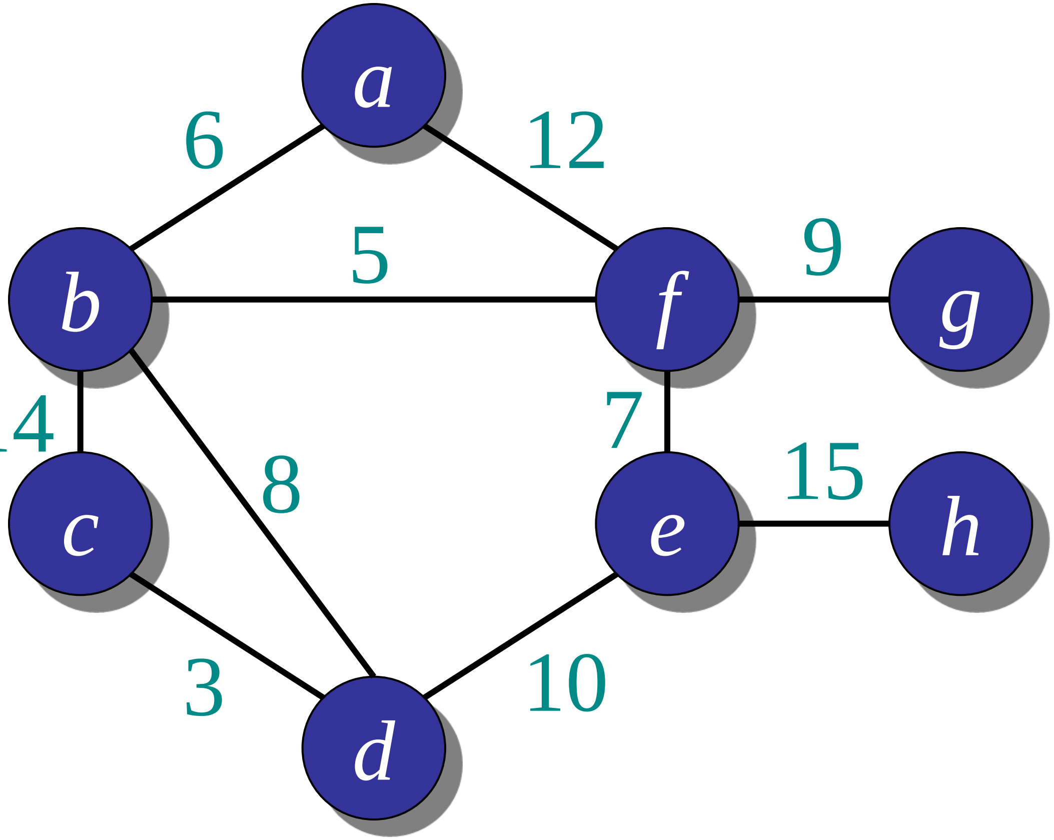
~~$\Theta(n^2)$ times?~~

Θ(m) times (with adj list)

How often is ExtractMin() called?

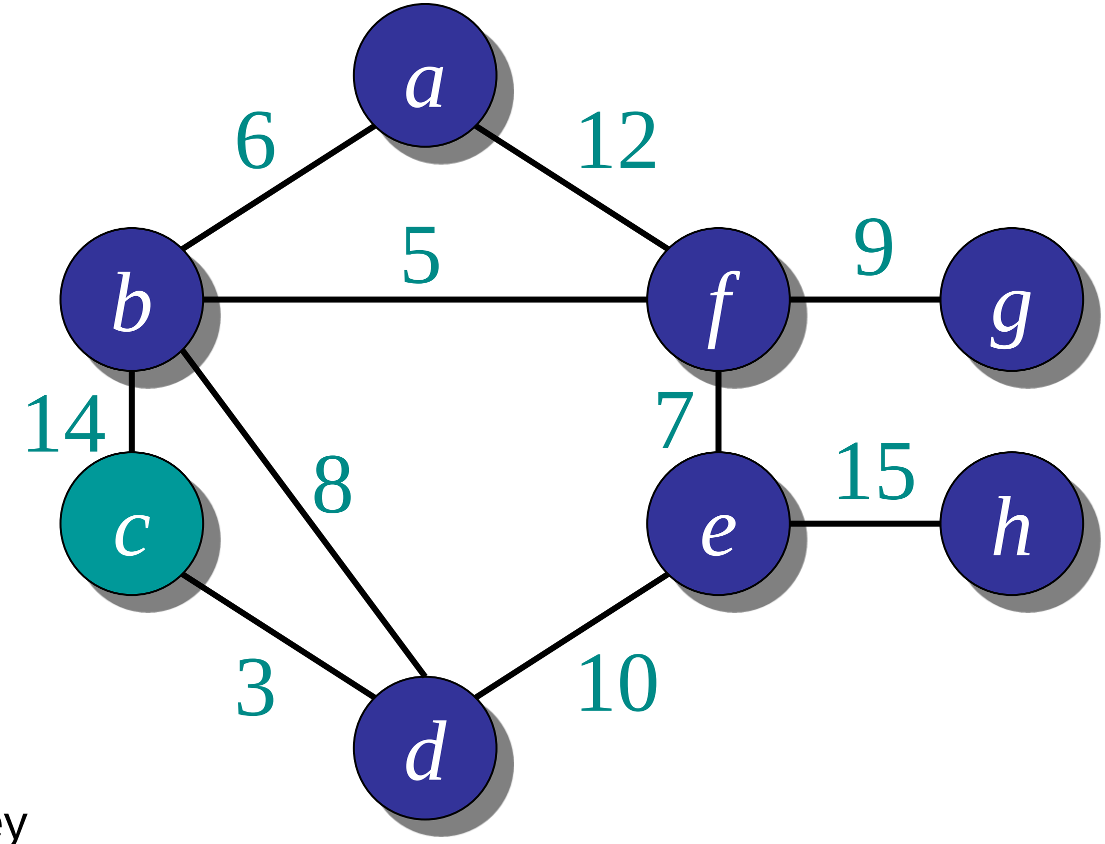
How often is ChangeKey() called?

Example



a	b	c	d	e	f	g	h
∞	∞	∞	∞	∞	∞	∞	∞

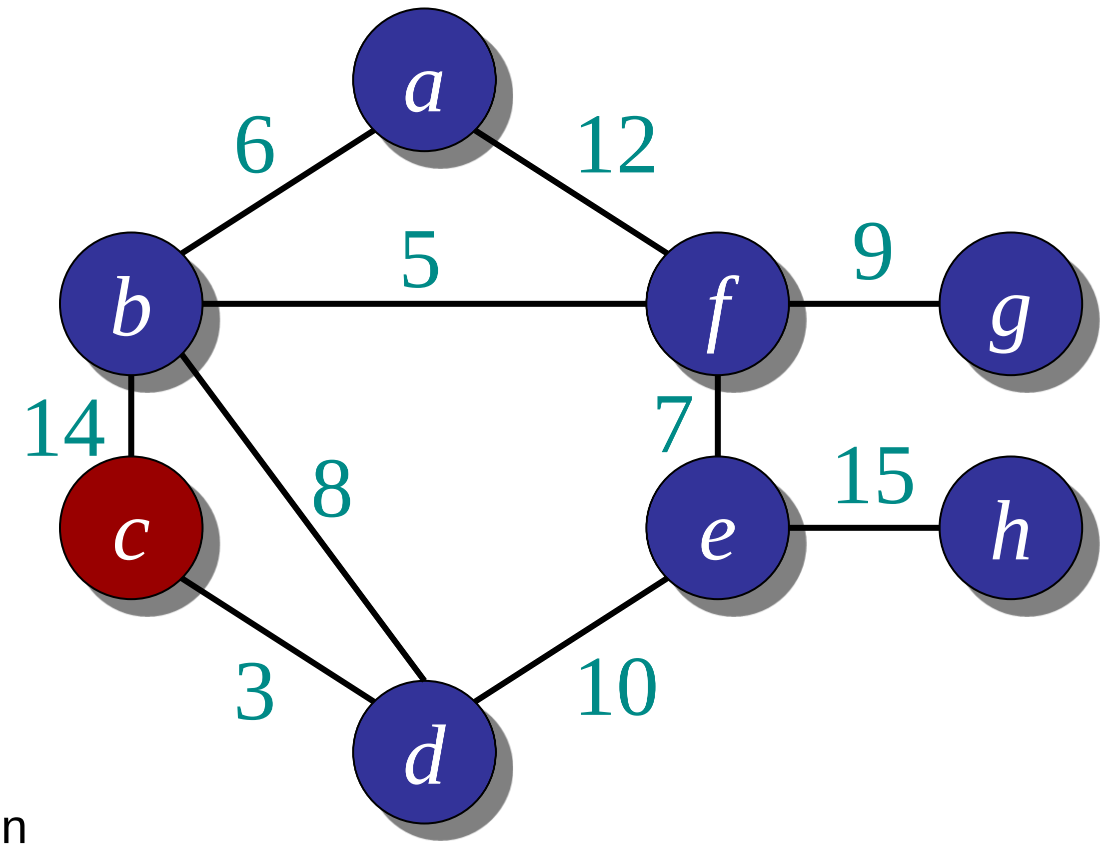
Example



ChangeKey

c	b	a	d	e	f	g	h
0	∞	∞	∞	∞	∞	∞	∞

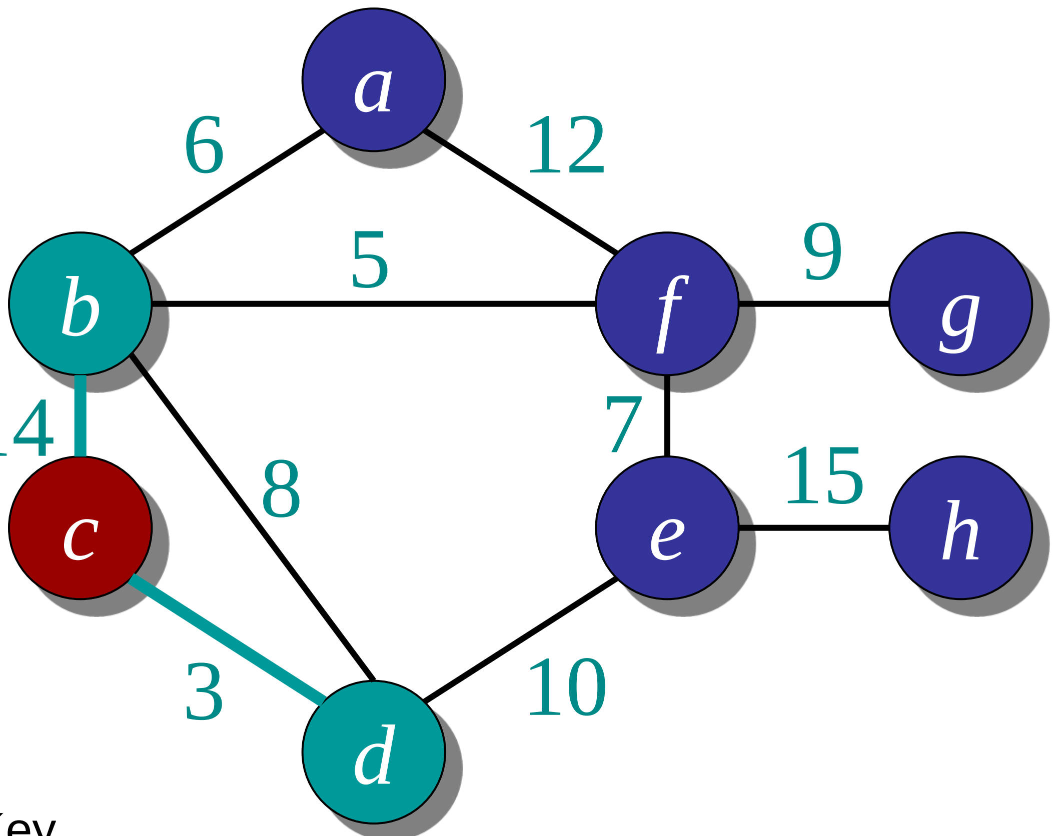
Example



ExctractMin

h	b	a	d	e	f	g
∞	∞	∞	∞	∞	∞	∞

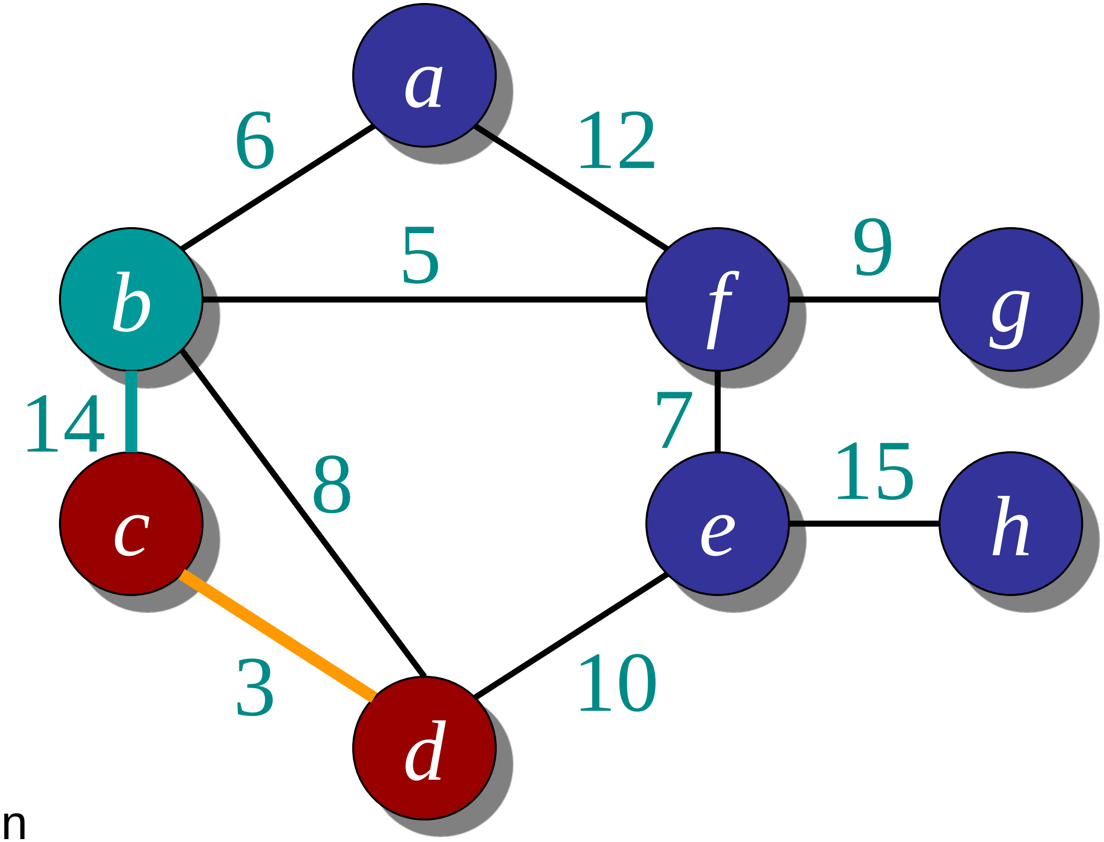
Example



ChangeKey

d	b	a	h	e	f	g
3	14	∞	∞	∞	∞	∞

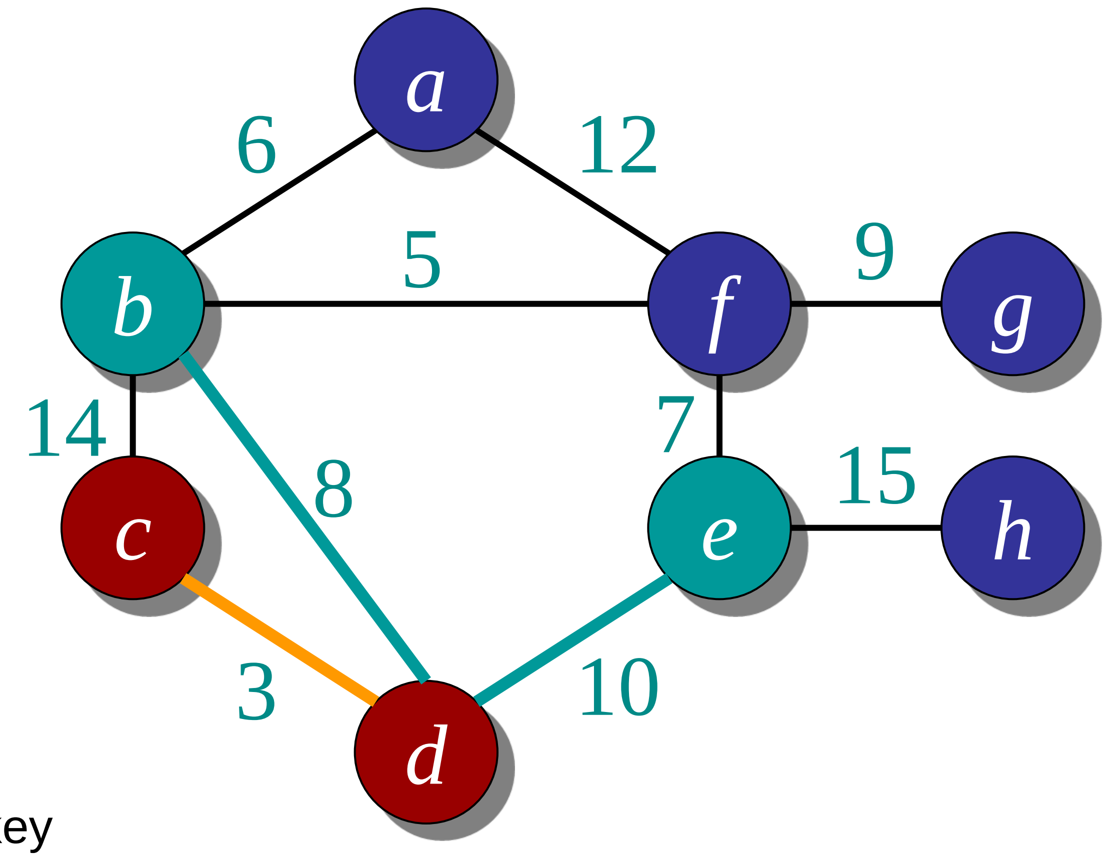
Example



ExctractMin

b	g	a	h	e	f
14	∞	∞	∞	∞	∞

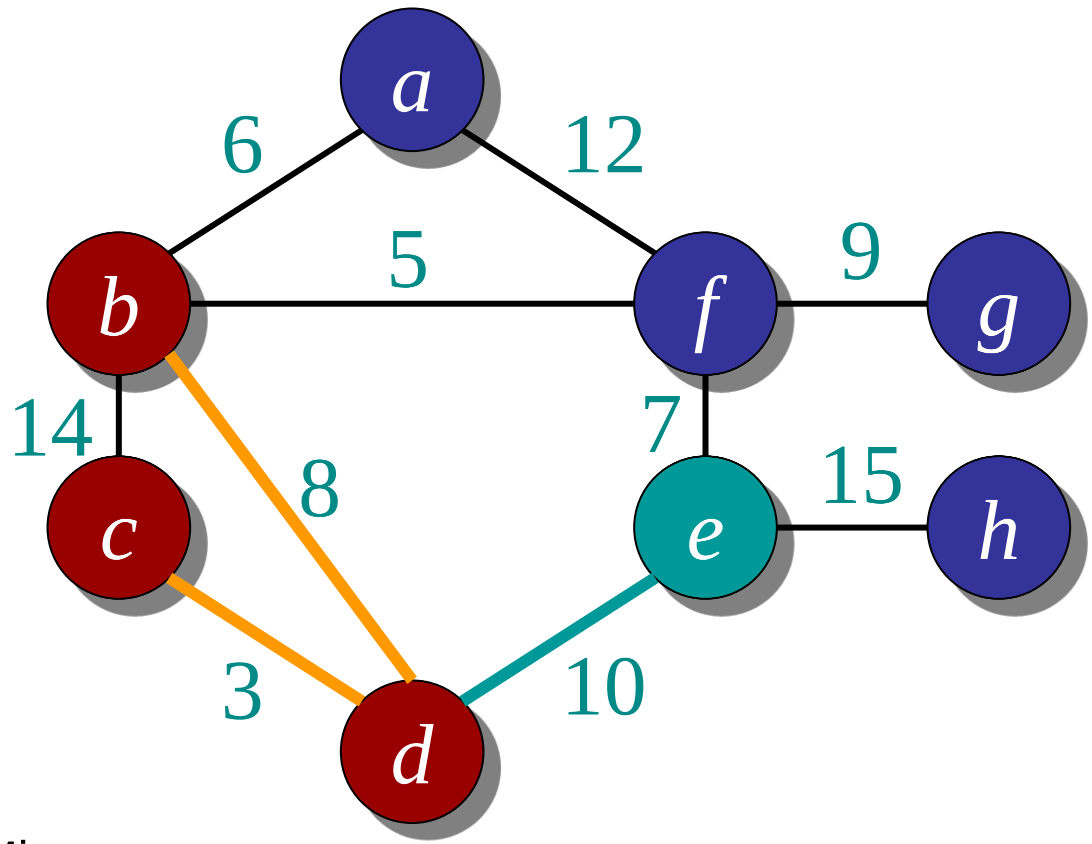
Example



Changekey

b	e	a	h	g	f
8	10	∞	∞	∞	∞

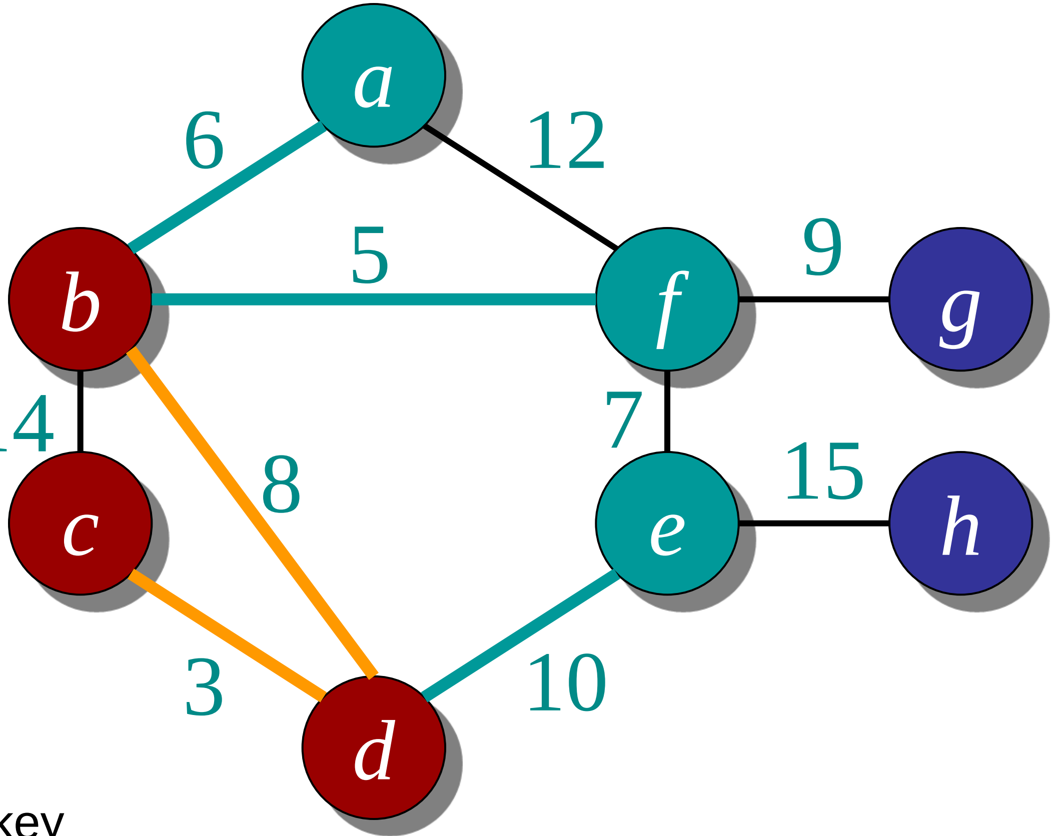
Example



ExtractMin

e	f	a	h	g
10	∞	∞	∞	∞

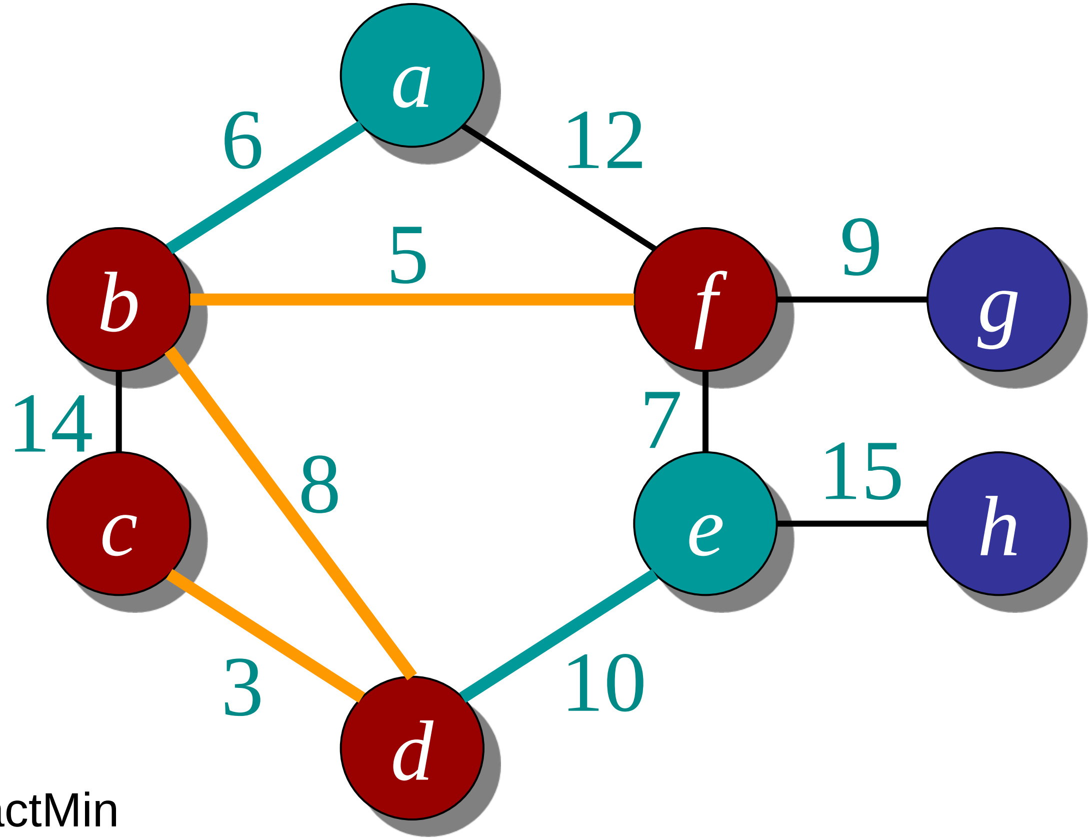
Example



Changekey

f	e	a	h	g
5	10	6	∞	∞

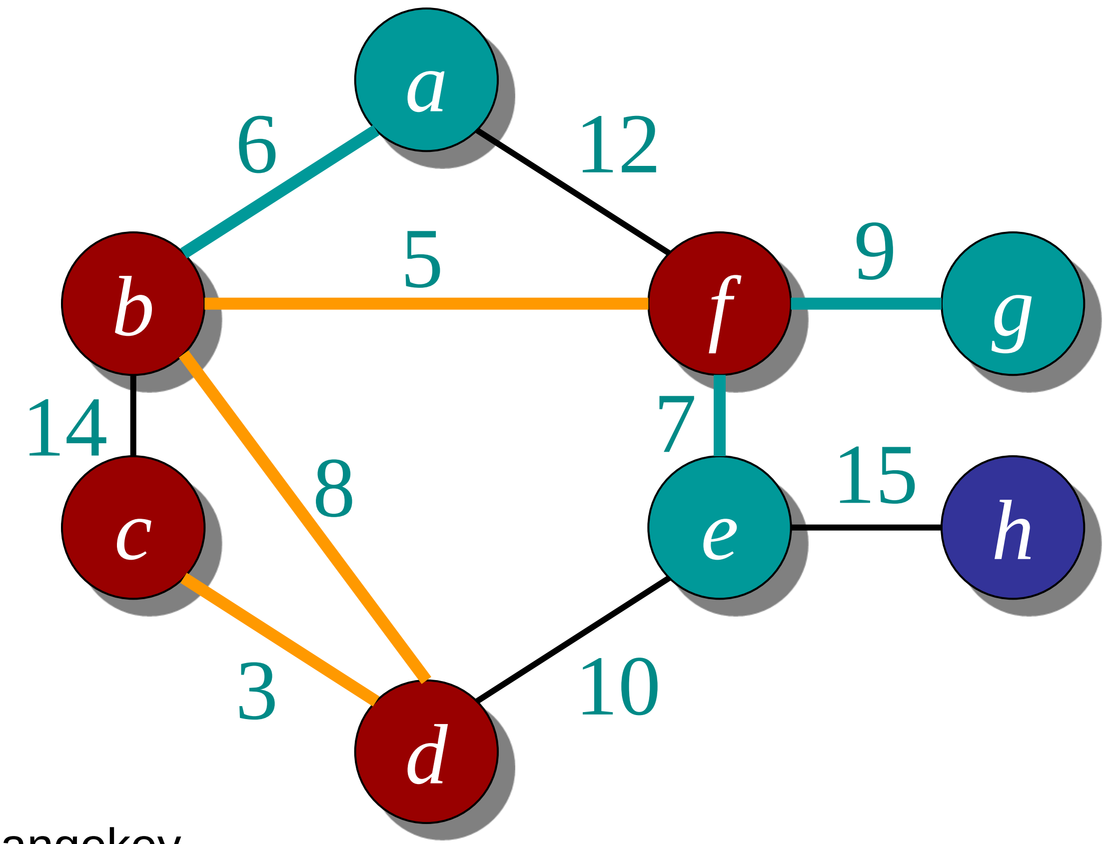
Example



ExtractMin

a	e	g	h
6	10	∞	∞

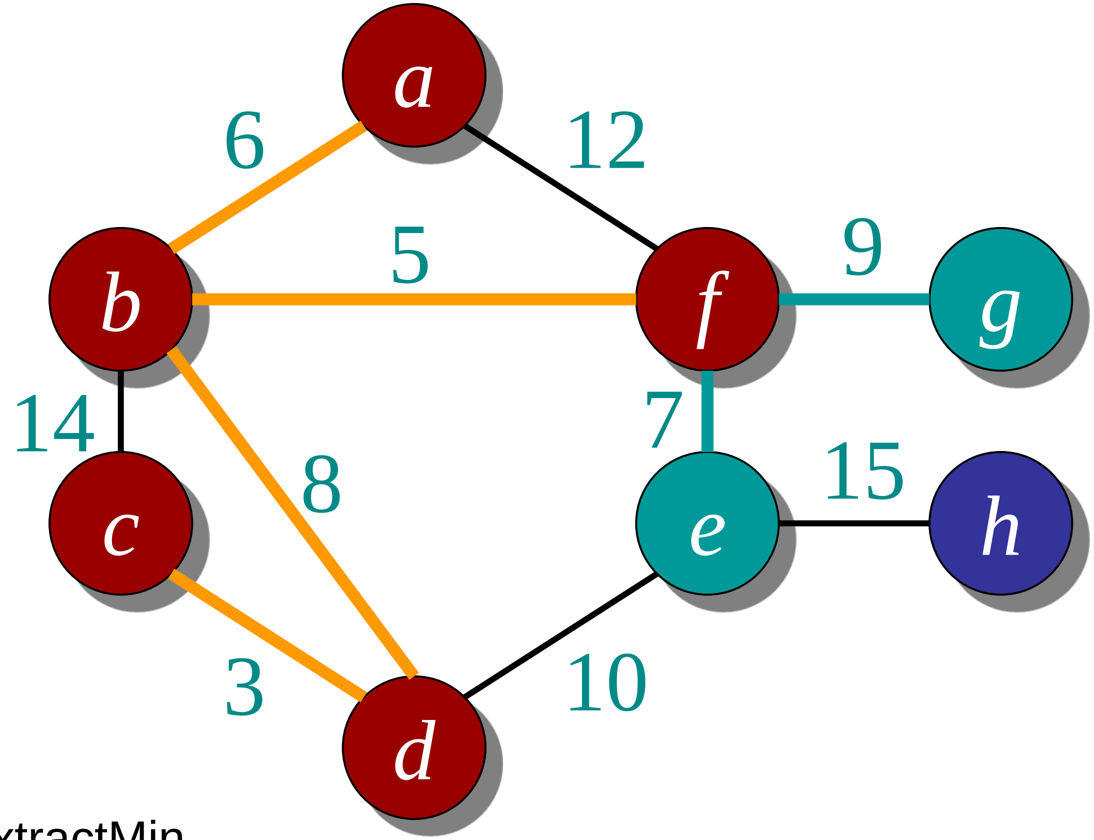
Example



Changekey

a	e	g	h
6	7	9	∞

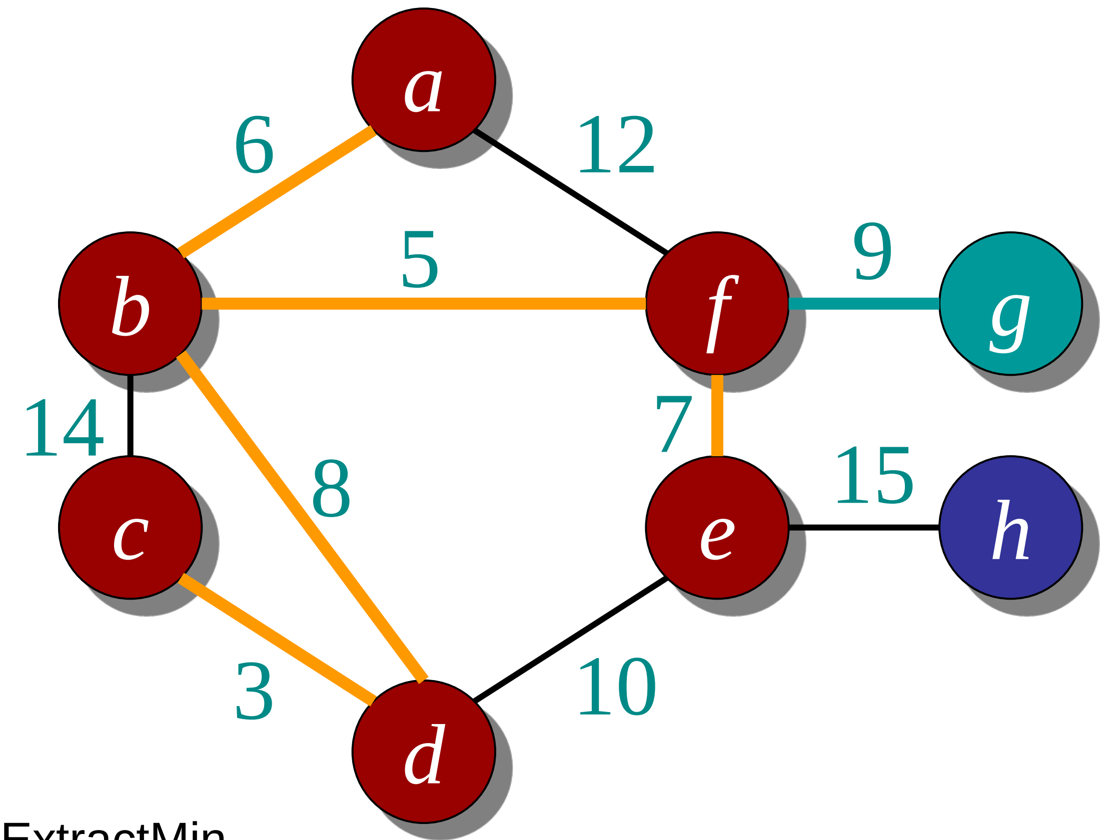
Example



ExtractMin

e	h	g
7	∞	9

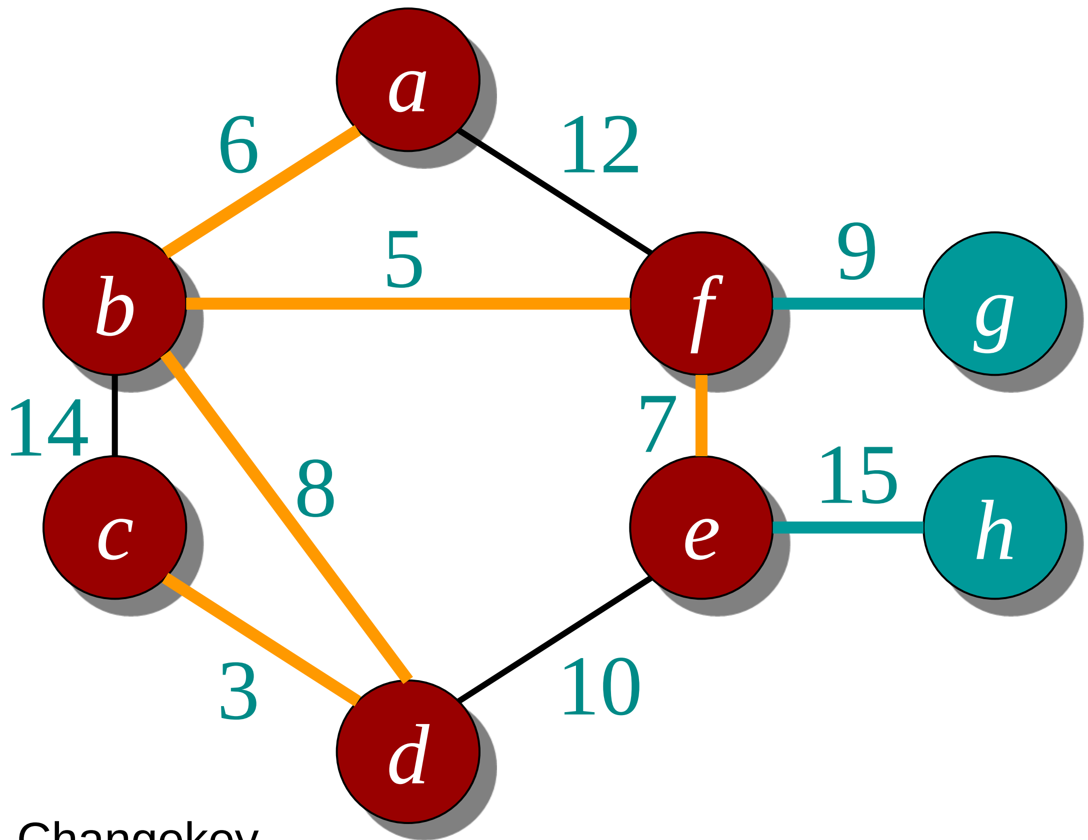
Example



ExtractMin

g	h
9	∞

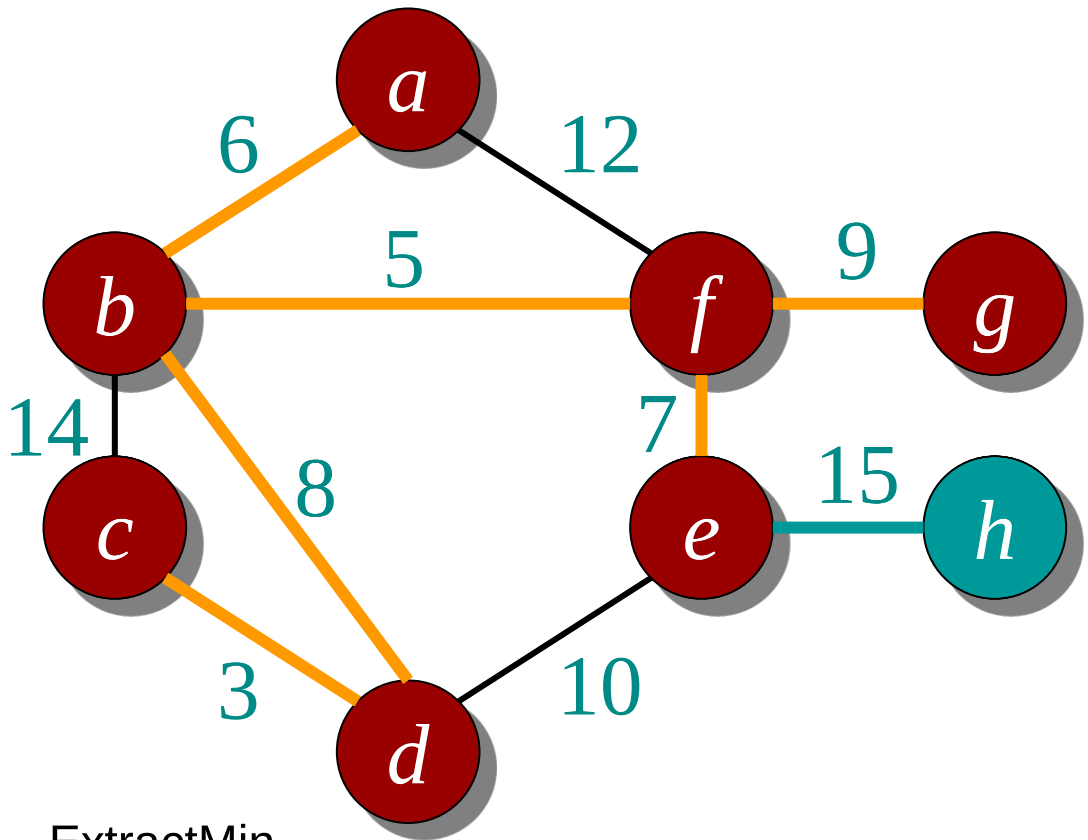
Example



Changekey

g	h
9	15

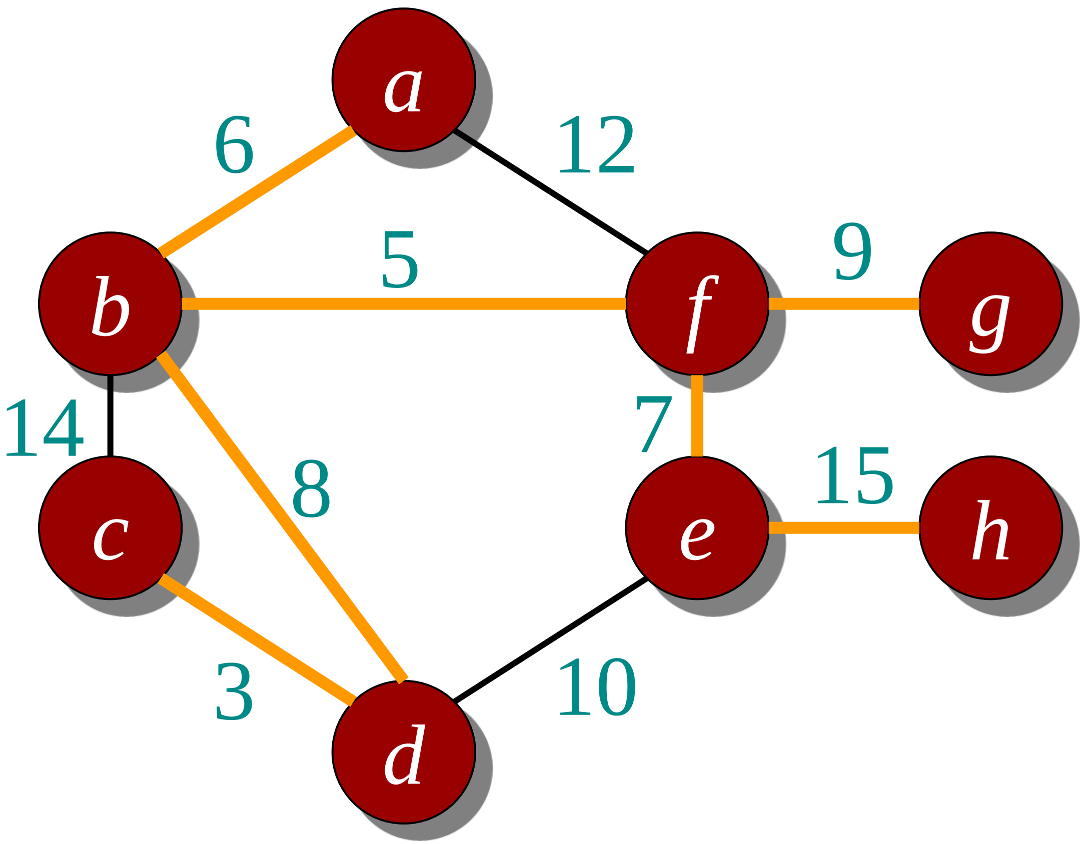
Example



ExtractMin

h
15

Example



Complete Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
        p[u] = null;
    key[r] = 0;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                ChangeKey(v, w(u,v));
```

Overall running time: ***Θ(m log n)***

Cost per ChangeKey

n vertices

Θ(log n) times

~~$\Theta(n^2)$ times?~~

Θ(m) times (with adj list)

How often is ExtractMin() called?

How often is ChangeKey() called?

Kruskal’s algorithm in words

- Procedure:
  - Sort all edges into non-decreasing order
  - Initially each node is in its own tree
  - For each edge in the sorted list
    - If the edge connects two separate trees, then
      - join the two trees together with that edge

Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

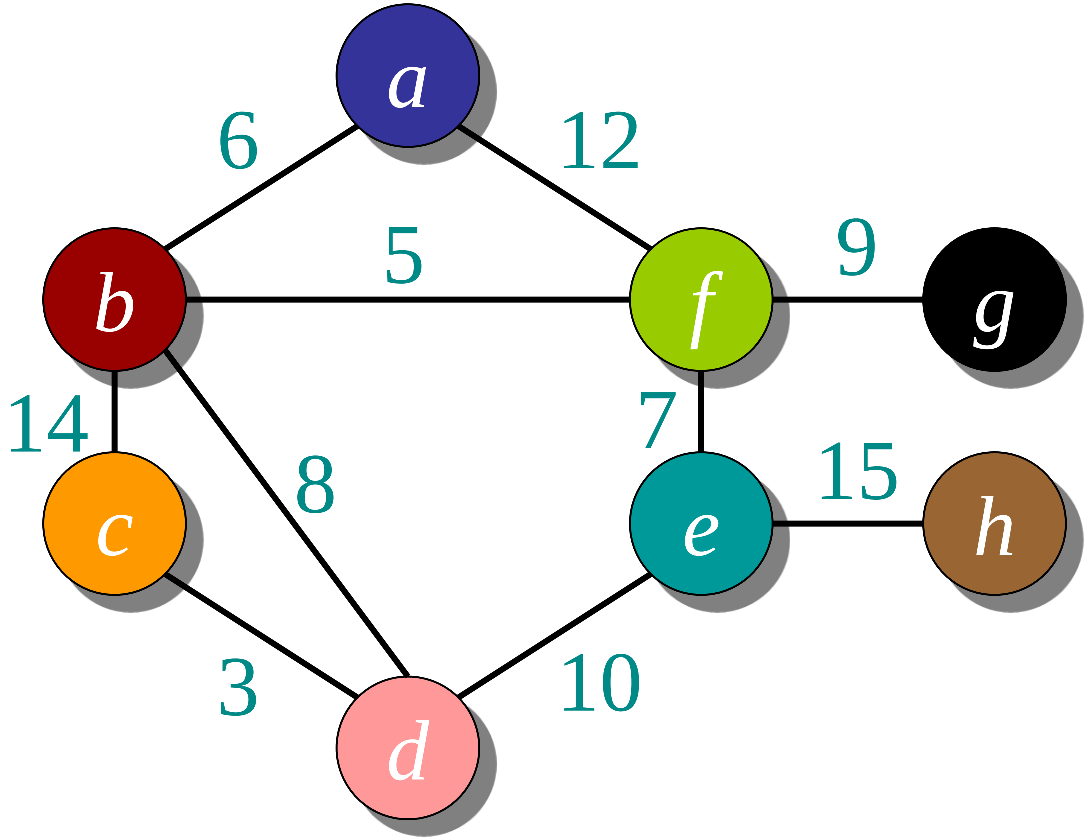
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

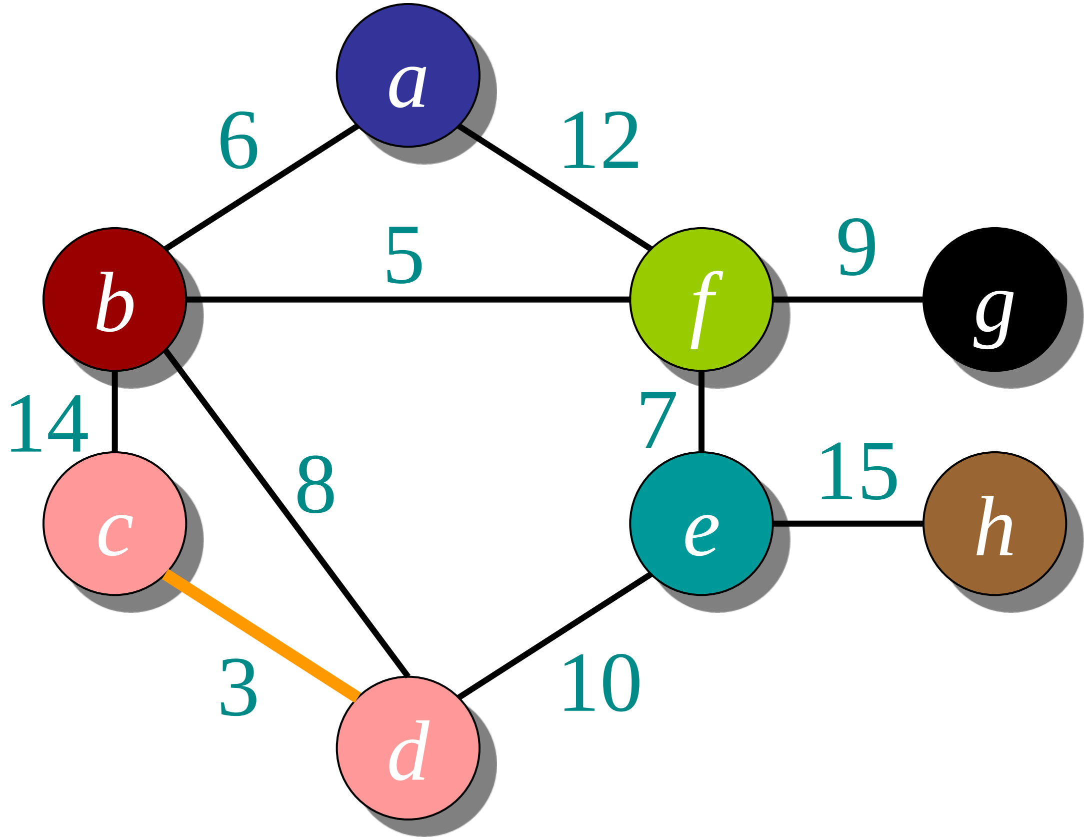
f-g: 9

d-e: 10

a-f: 12

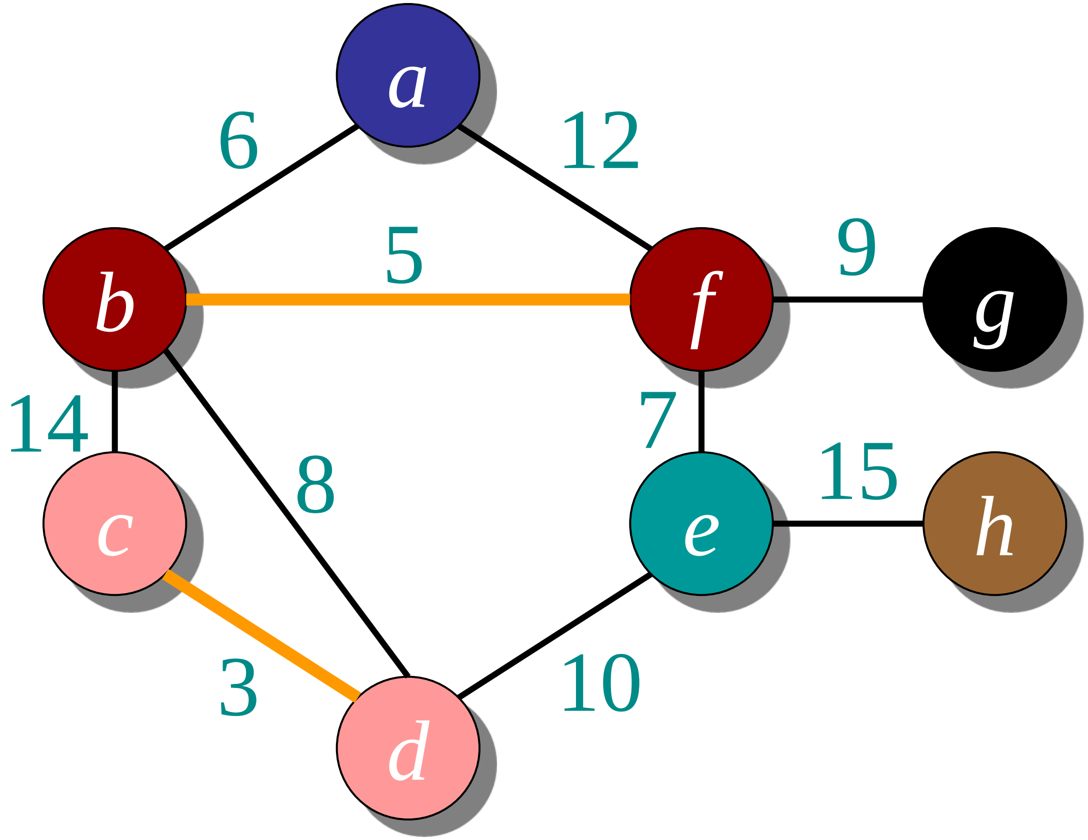
b-c: 14

e-h: 15



Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

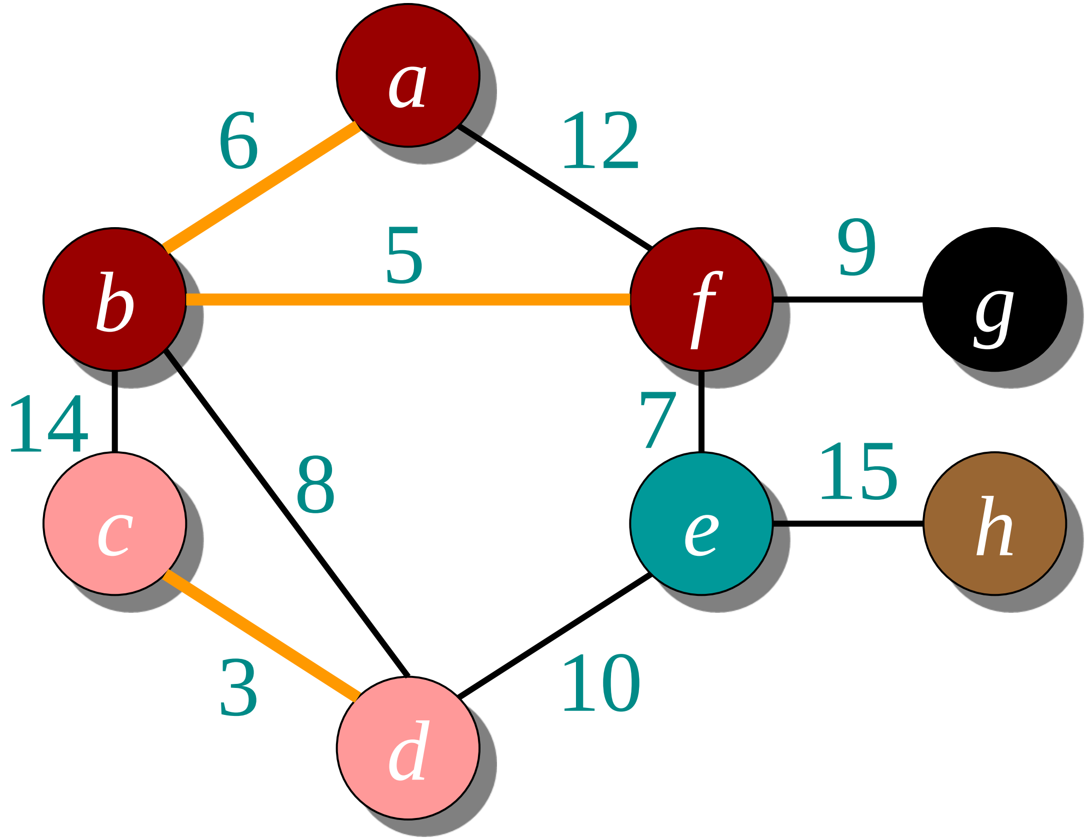
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

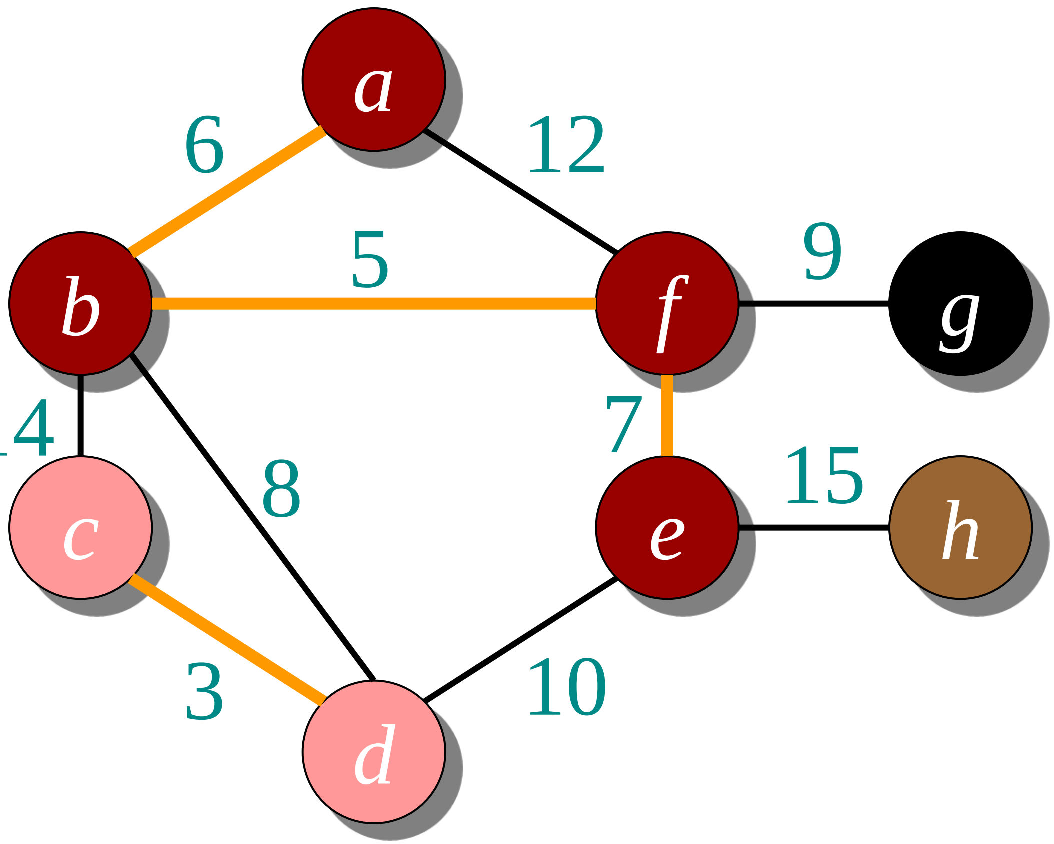
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

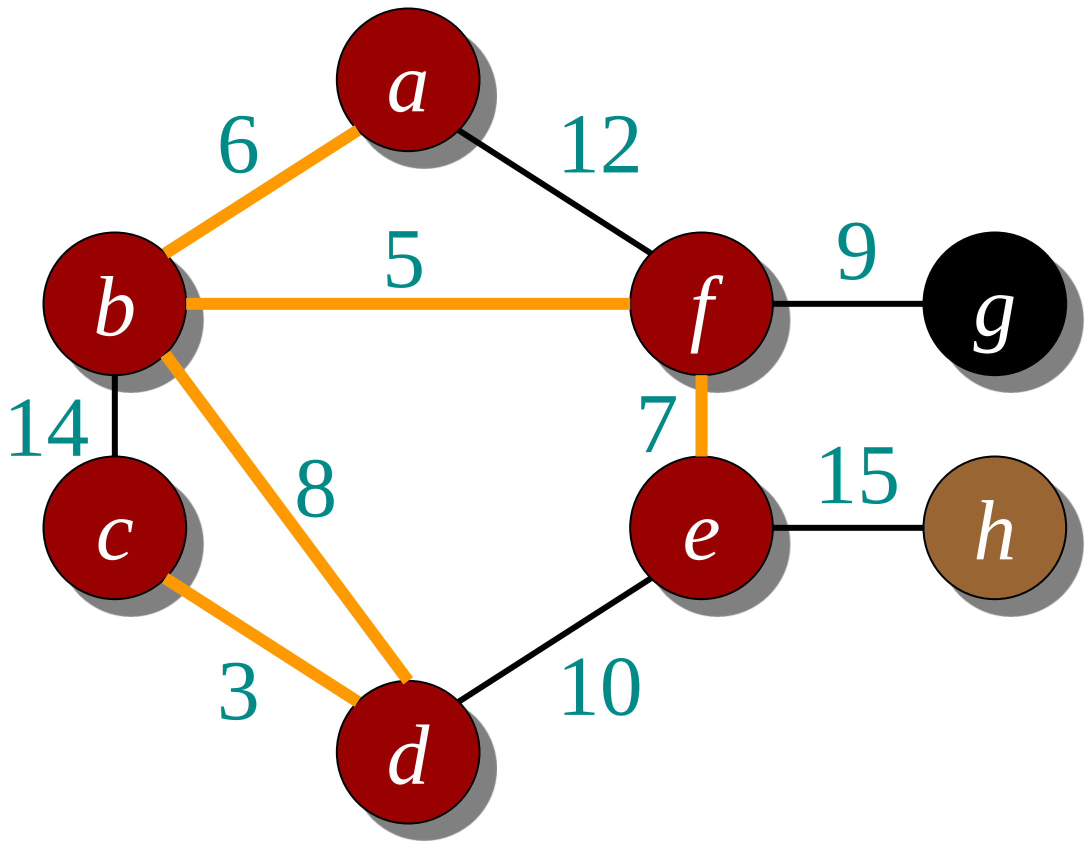
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

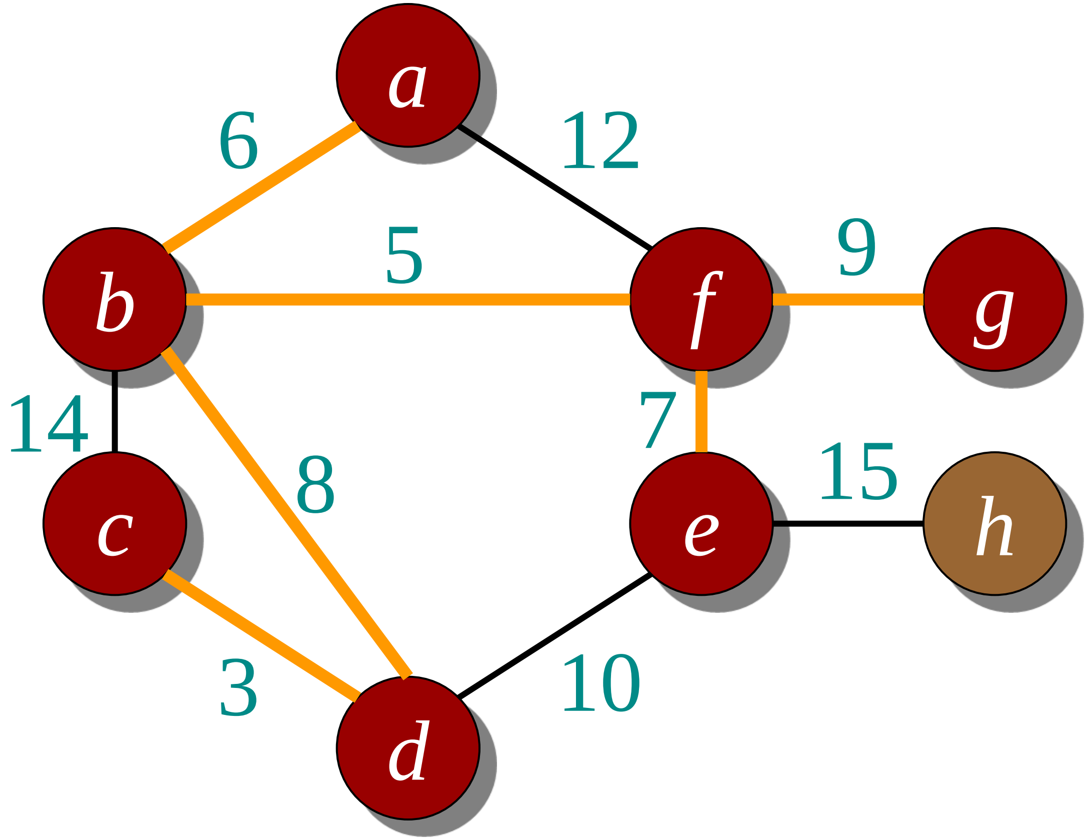
f-g: 9

d-e: 10

a-f: 12

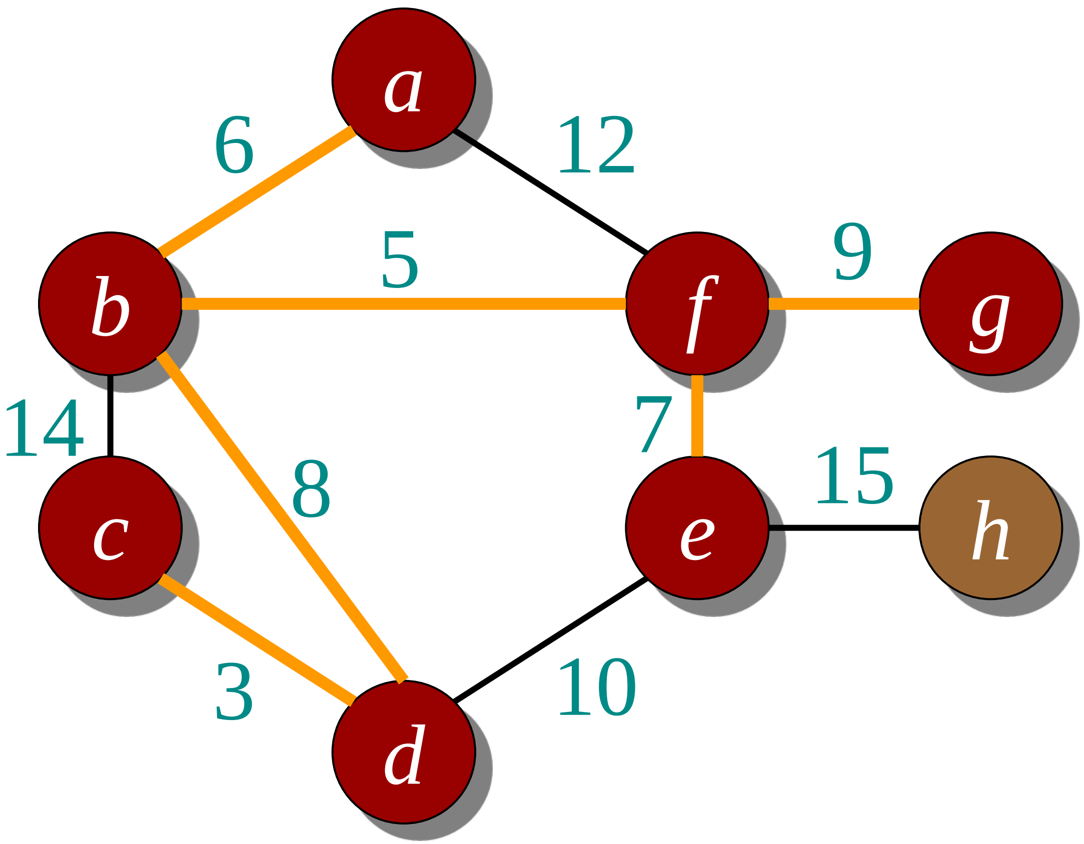
b-c: 14

e-h: 15



Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

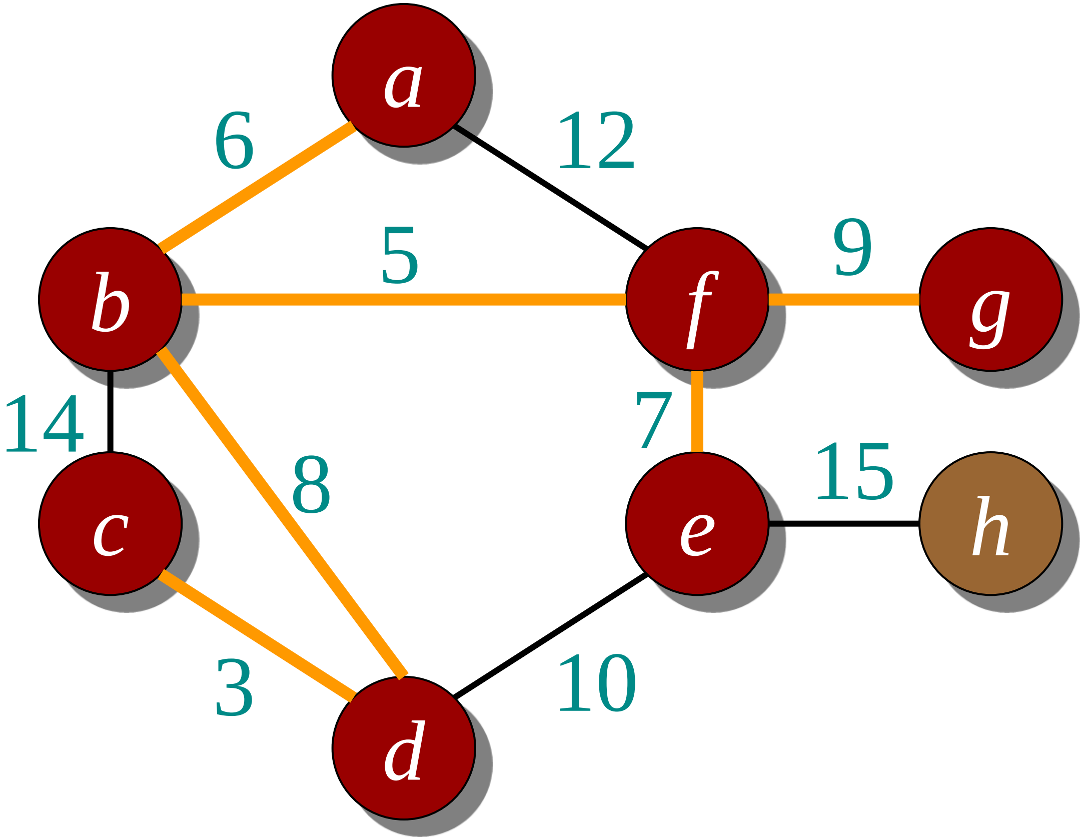
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

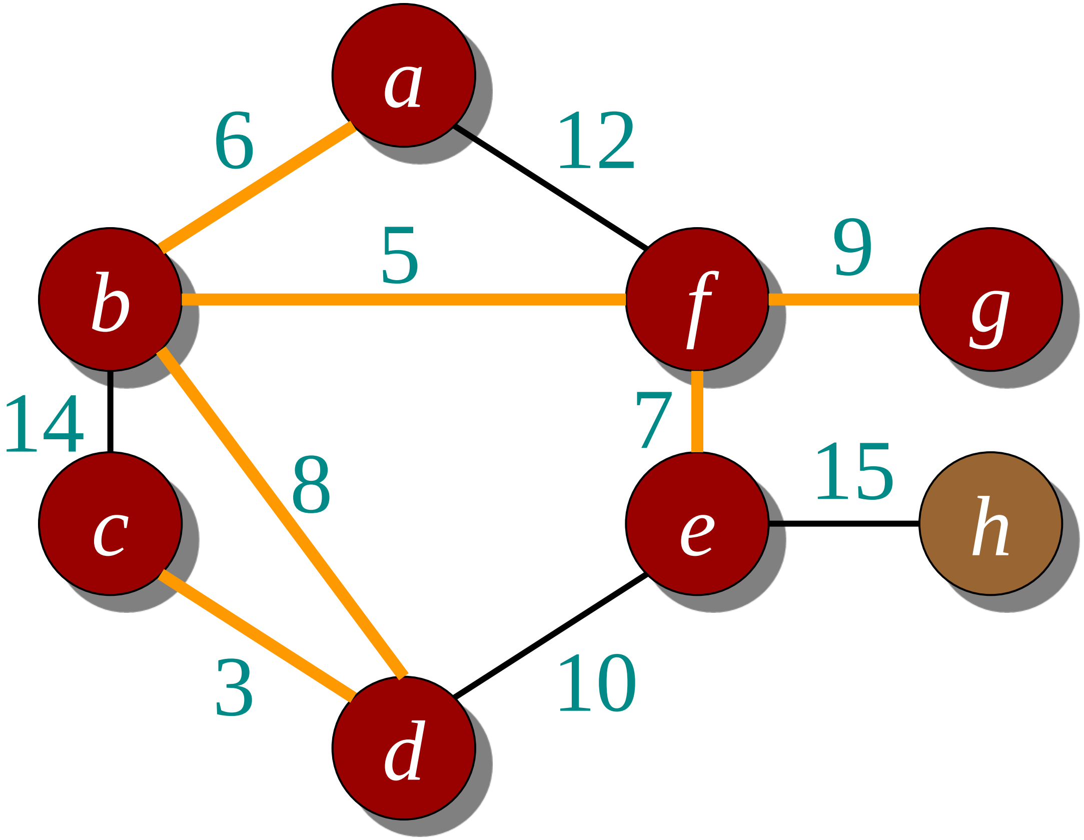
f-g: 9

d-e: 10

a-f: 12

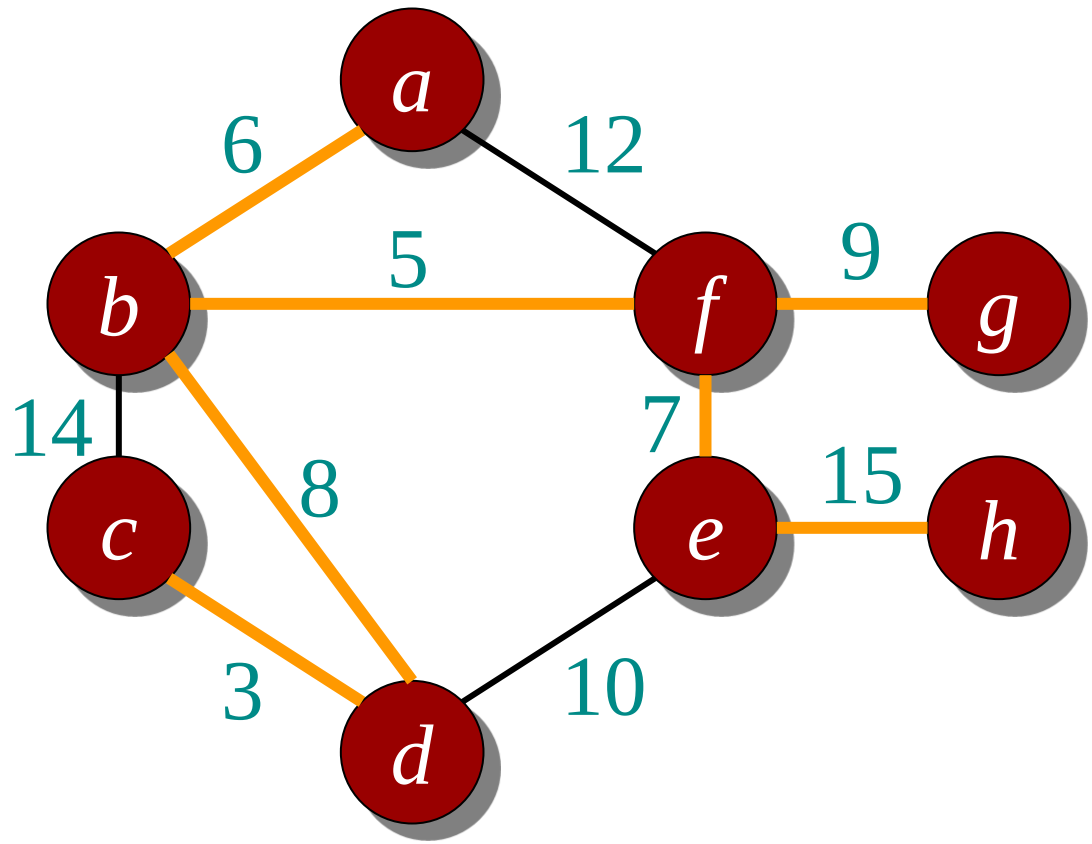
b-c: 14

e-h: 15



Example

c-d:	3
b-f:	5
b-a:	6
f-e:	7
b-d:	8
f-g:	9
d-e:	10
a-f:	12
b-c:	14
e-h:	15



Time complexity

- Depend on implementation
- Pseudocode

sort all edges according to weights

T = {}. tree(v) = v for all v.

for each edge (u, v) in sorted order

if tree(u) != tree(v)

T = T U (u, v);

union (tree(u), tree(v))

$\Theta(m \log m)$
$= \Theta(m \log n)$

m edges

m times

(n-1) times.

Overall time complexity: $m \log n + m\, t + n\, u$

Naïve: $\Theta(nm)$

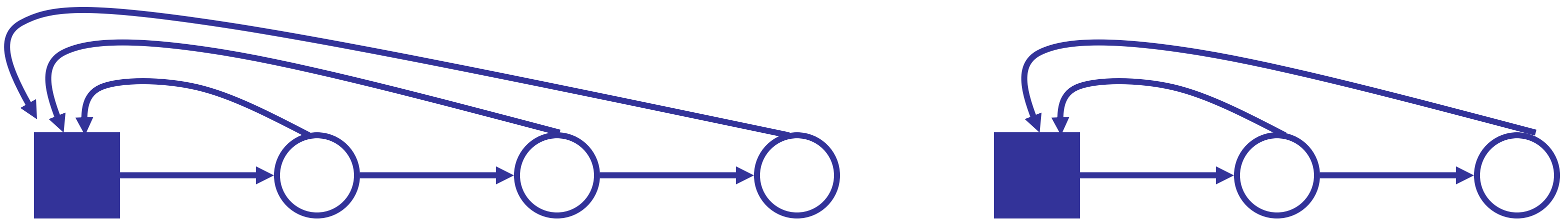
Better implementation: $\Theta(m \log n)$

tree union

tree finding

Disjoint Set Union

- Use a linked list to represent tree elements, with pointers back to root:
  - Each tree has one representative point,- its root
  - If two trees has same root they are identical otherwise they are different



  - tree[u] != tree[v]: O(1)
  - Union(tree[u],tree[v]): “Copy” elements of A into set B by adjusting elements of A to point to B
  - Each union may take O(n) time
  - Precisely n-1 unions
  - $O(n^2)$? (can be done better)

Disjoint Set Union

- Better strategy and analysis
  - Always copy smaller list into larger list
  - Size of combined list is at least twice the size of smaller list
  - Each vertex copied at most $\log n$ times before a single tree emerges
  - Total number of copy operations for n vertex is therefore $O(n \log n)$
- Overall time complexity: $m \log n + m\, t + n\, u$
  - t: time for finding tree root is constant
  - u: time for n union operations is at most nlog (n)
  - $m \log n + m\, t + n\, u = \Theta(m \log n + m + n \log n) = \Theta(m \log n)$
- Conclusion
  - Kruskal's algorithm runs in $\Theta(m \log n)$ for both dense and sparse graph

Example with disjoint set union

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

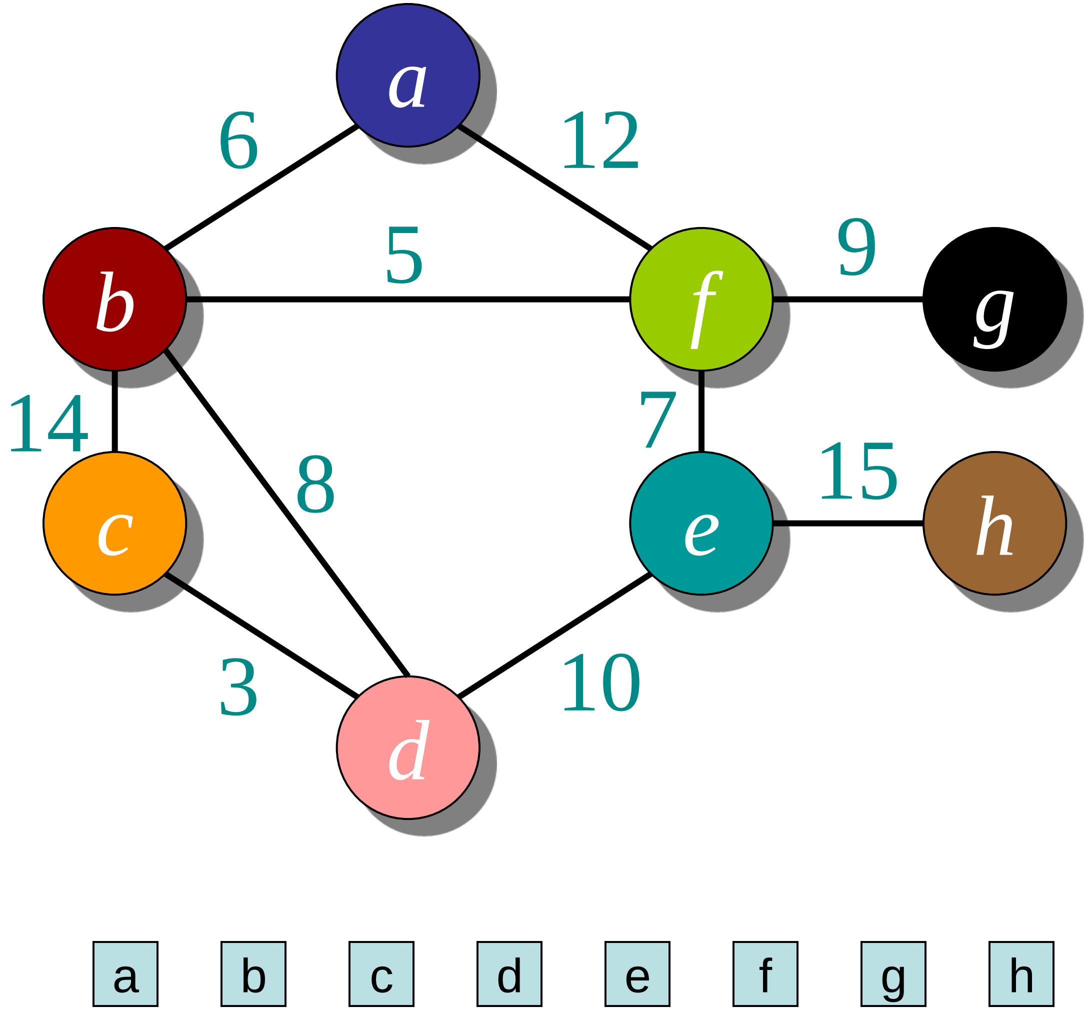
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example with disjoint set union

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

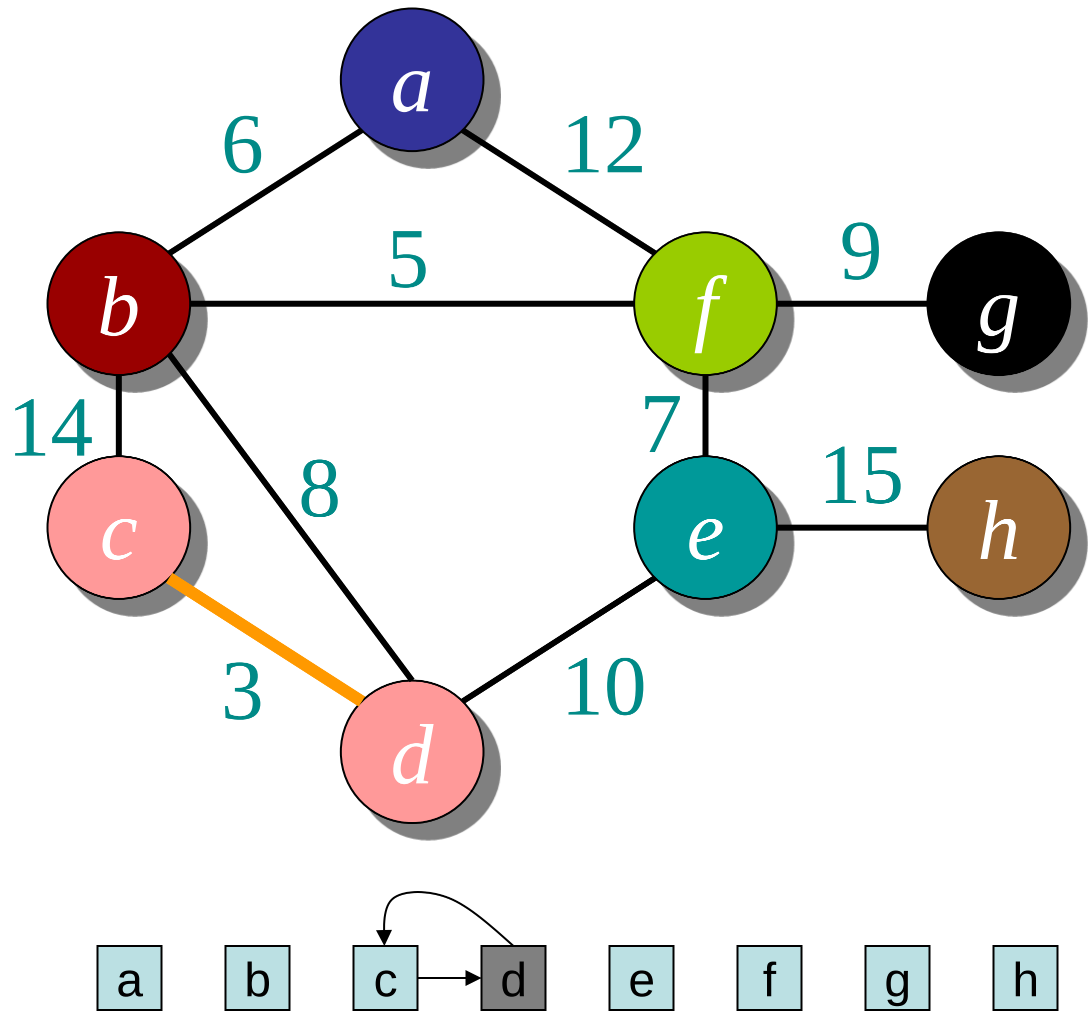
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example with disjoint set union

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

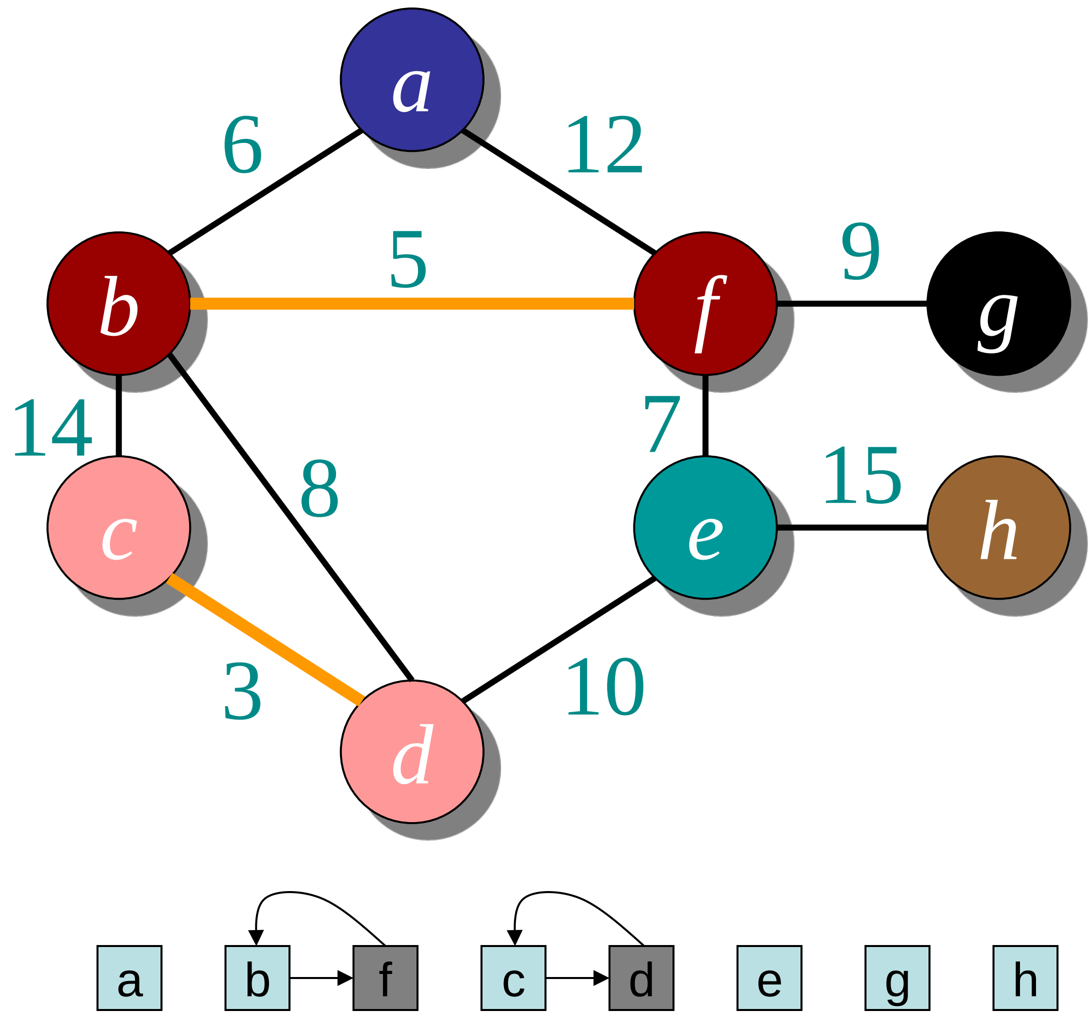
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example with disjoint set union

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

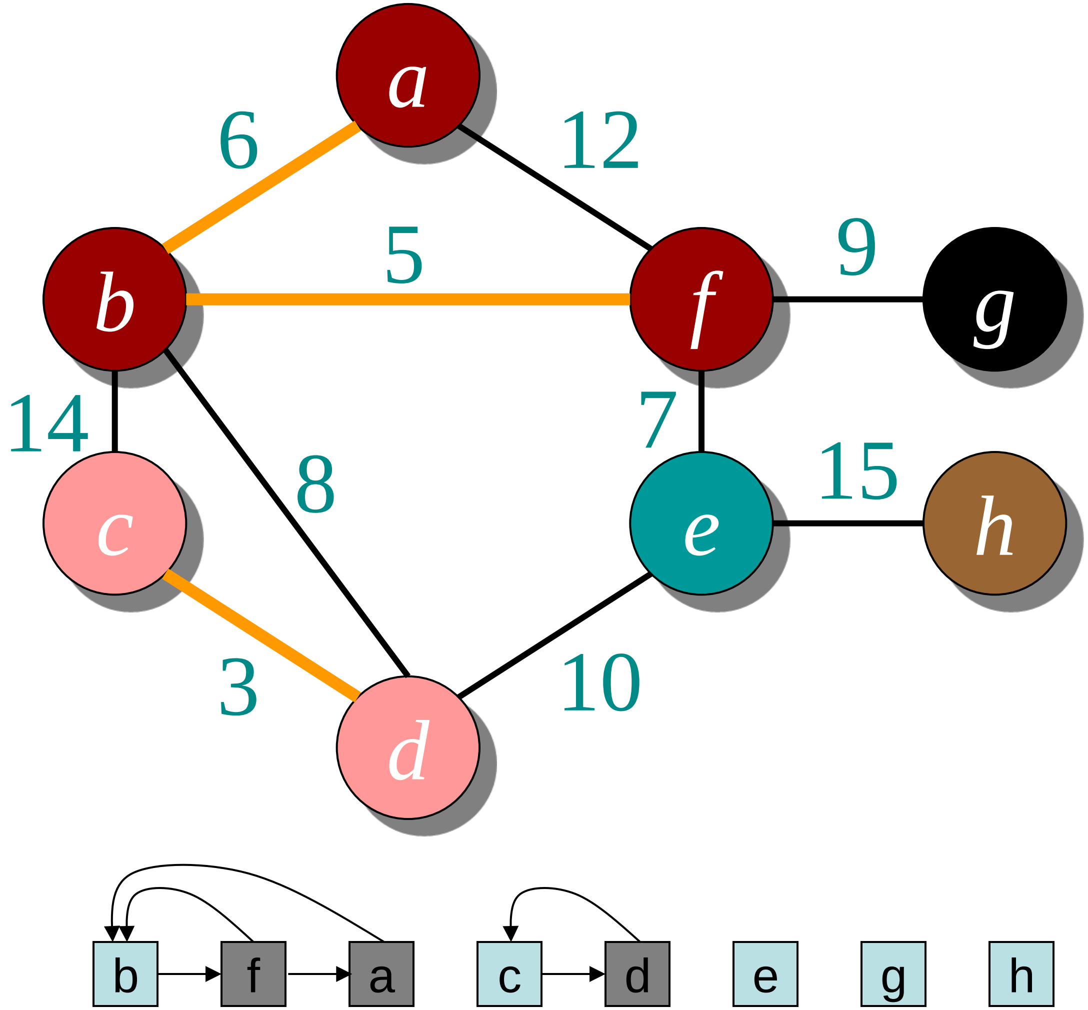
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example with disjoint set union

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

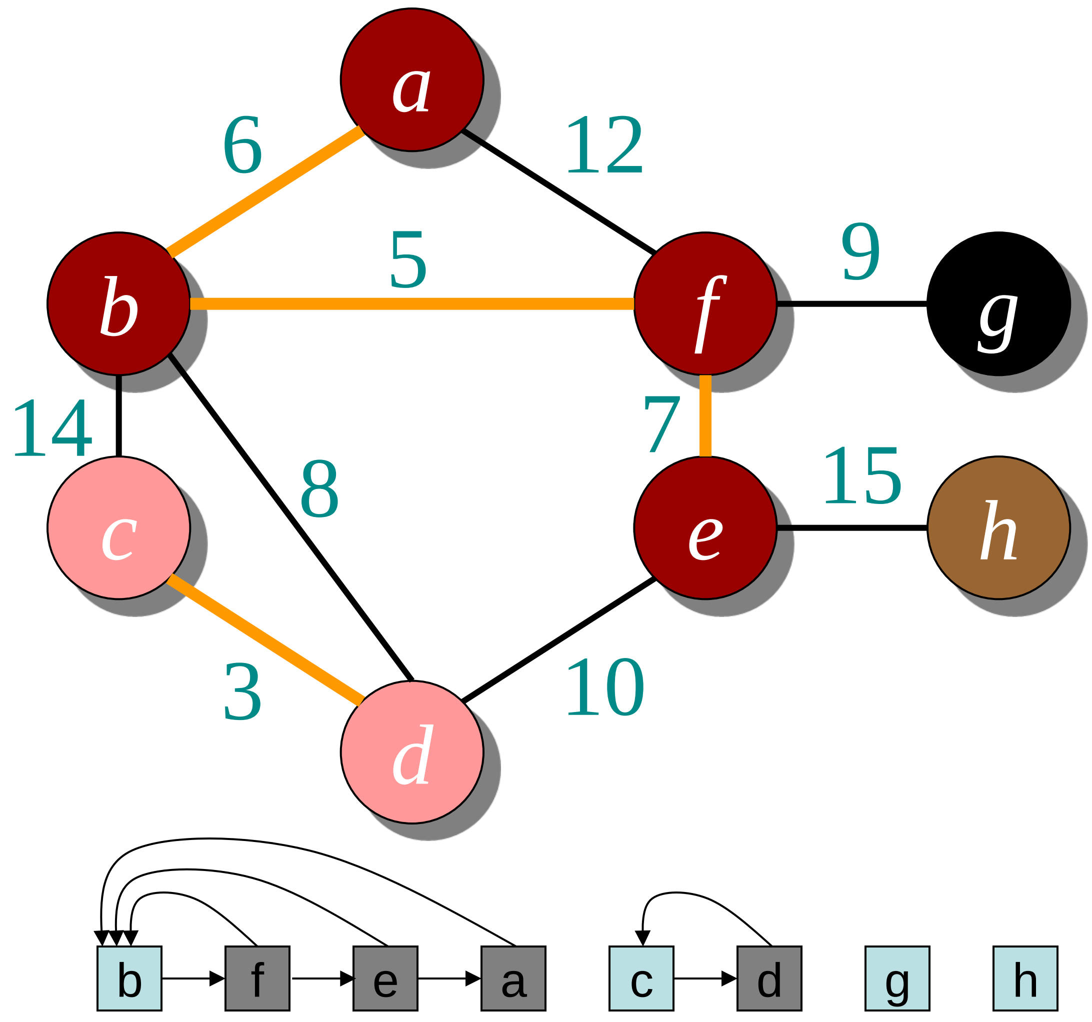
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example with disjoint set union

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

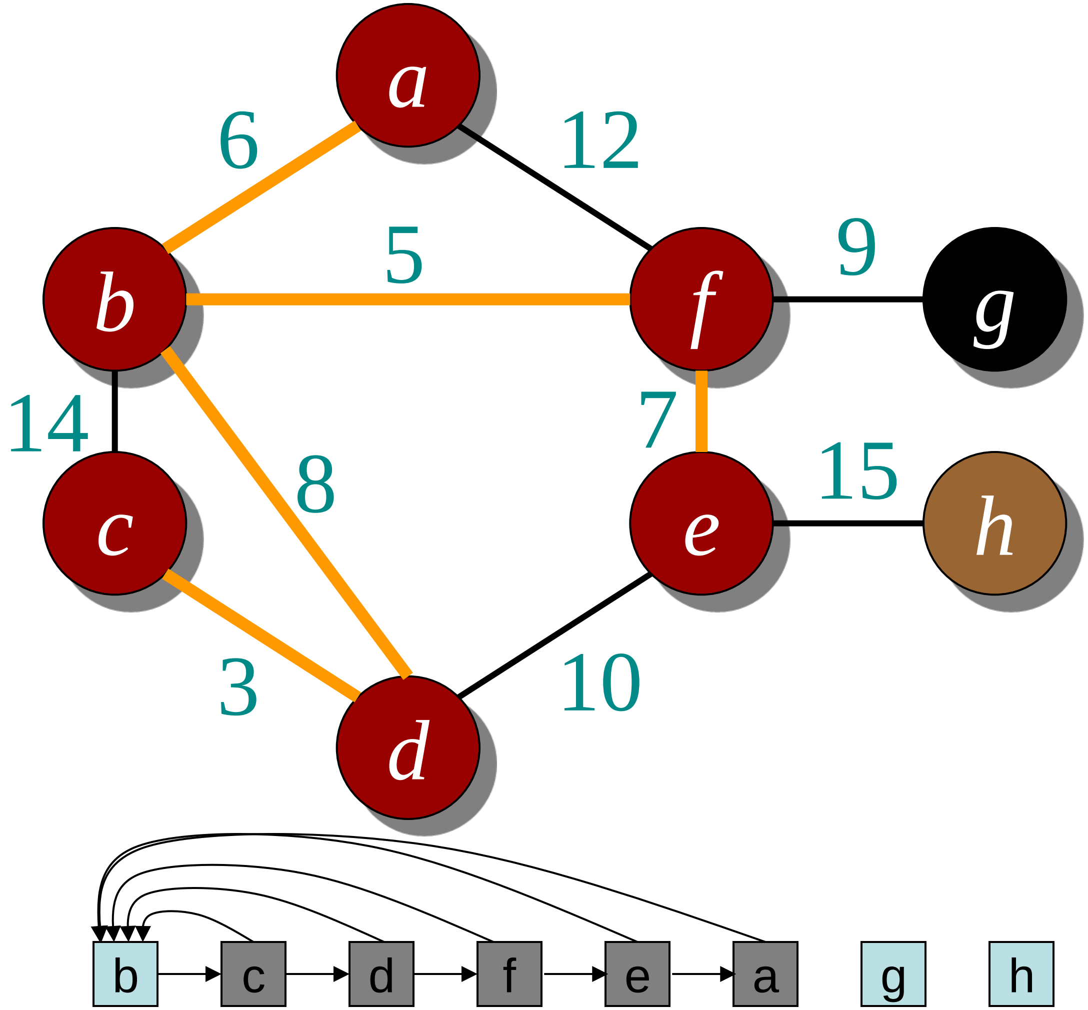
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example with disjoint set union

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

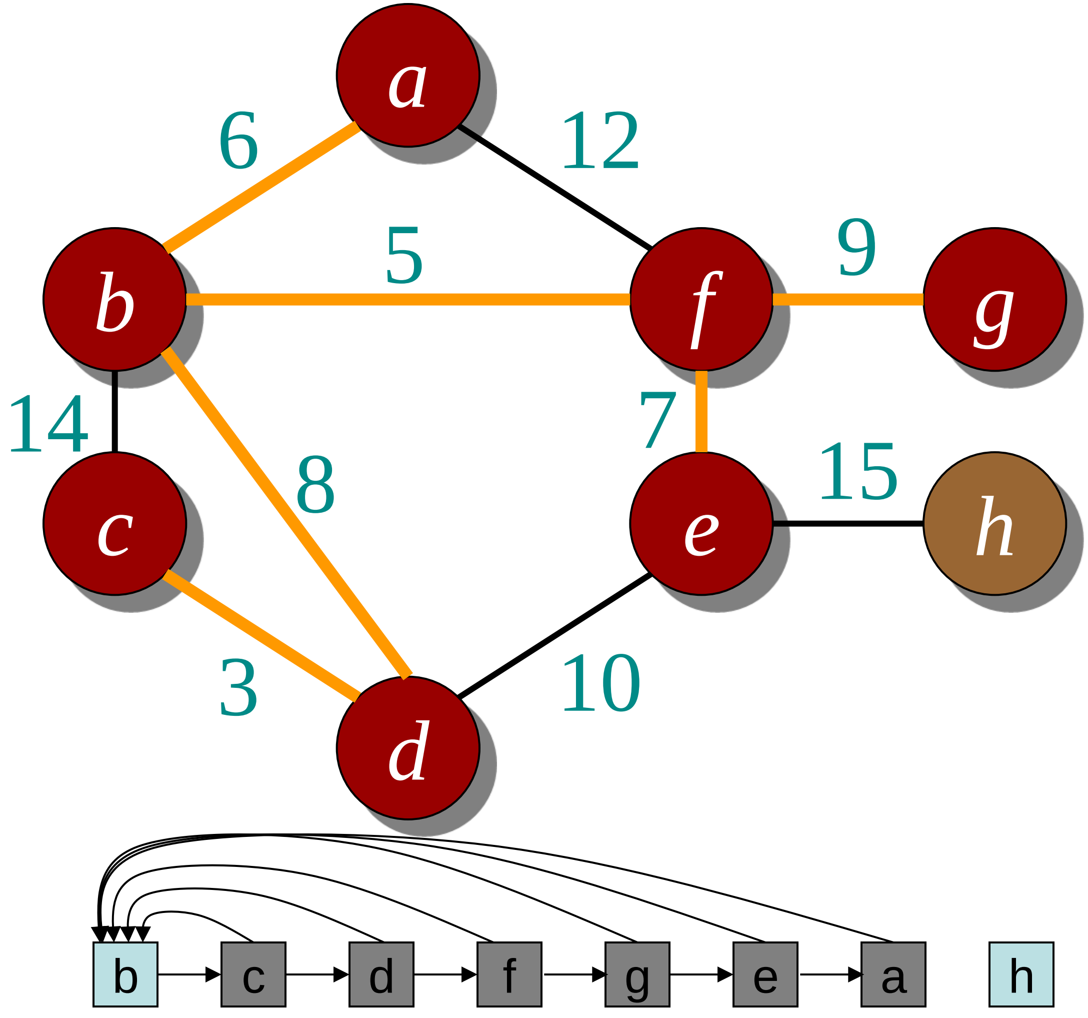
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example with disjoint set union

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

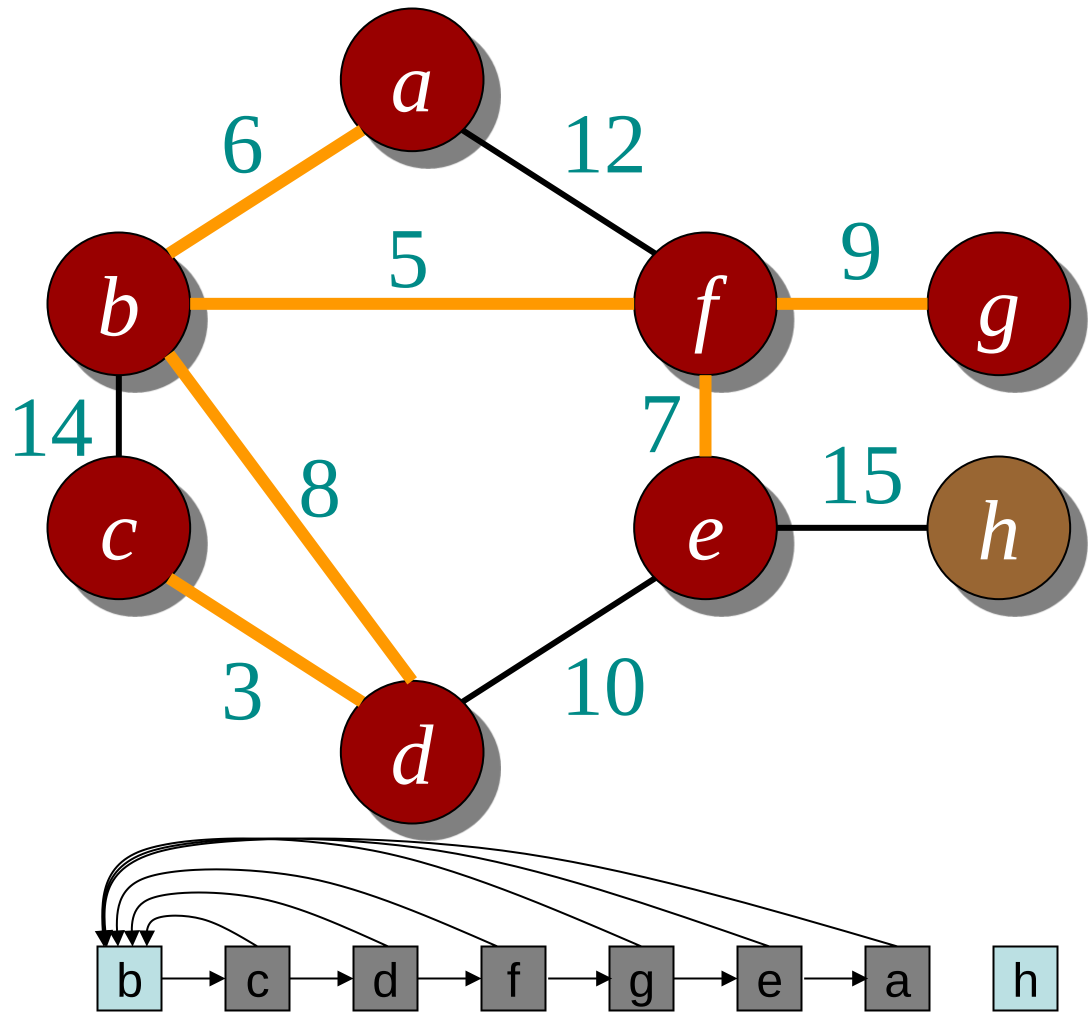
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example with disjoint set union

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

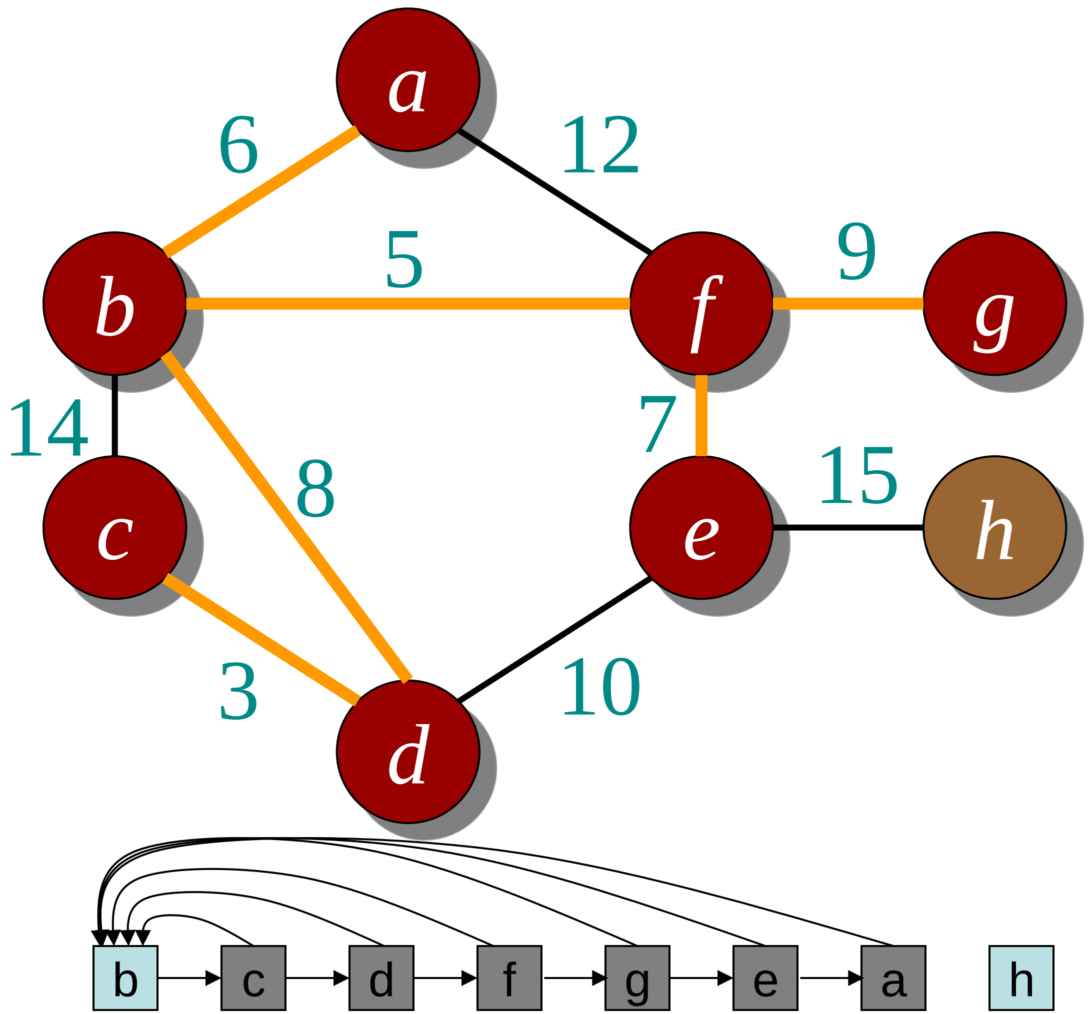
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example with disjoint set union

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

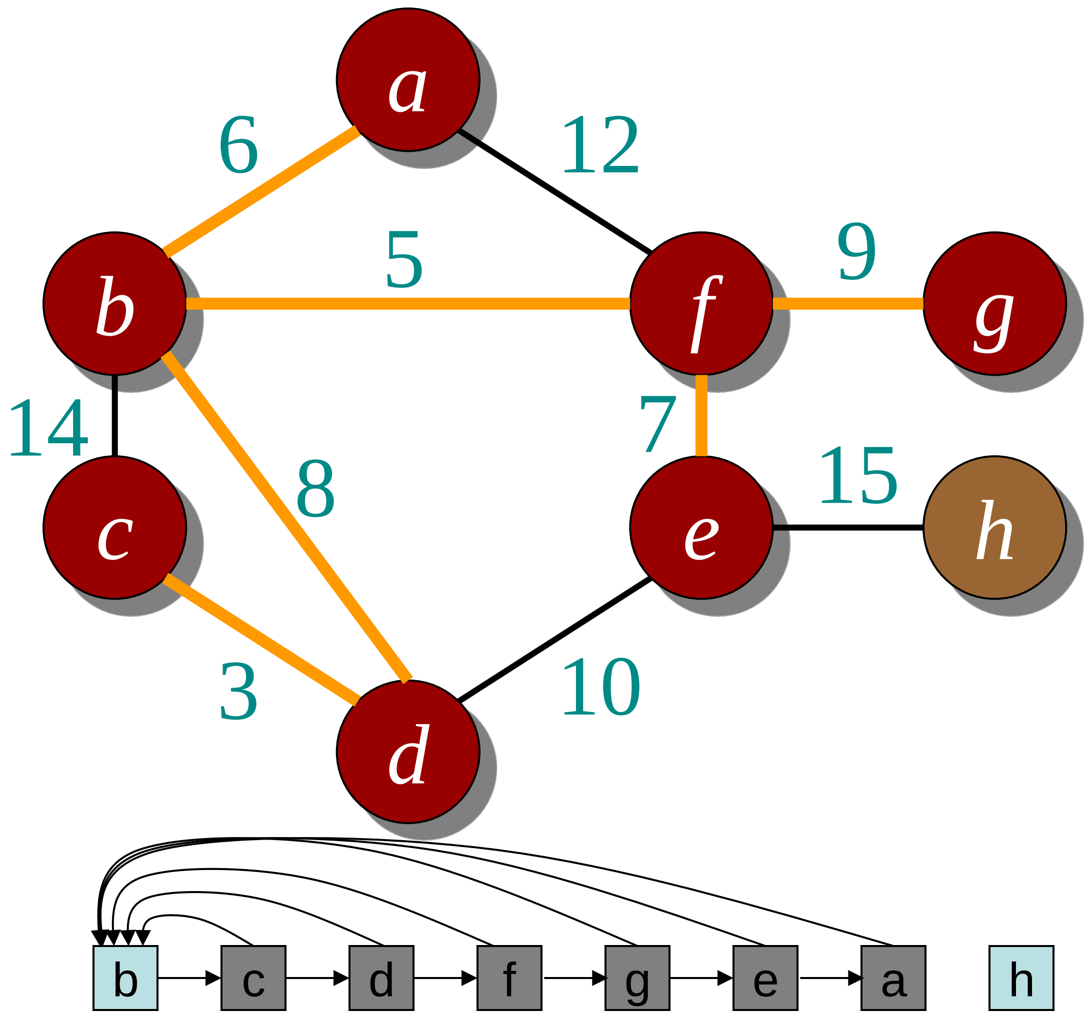
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Example with disjoint set union

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

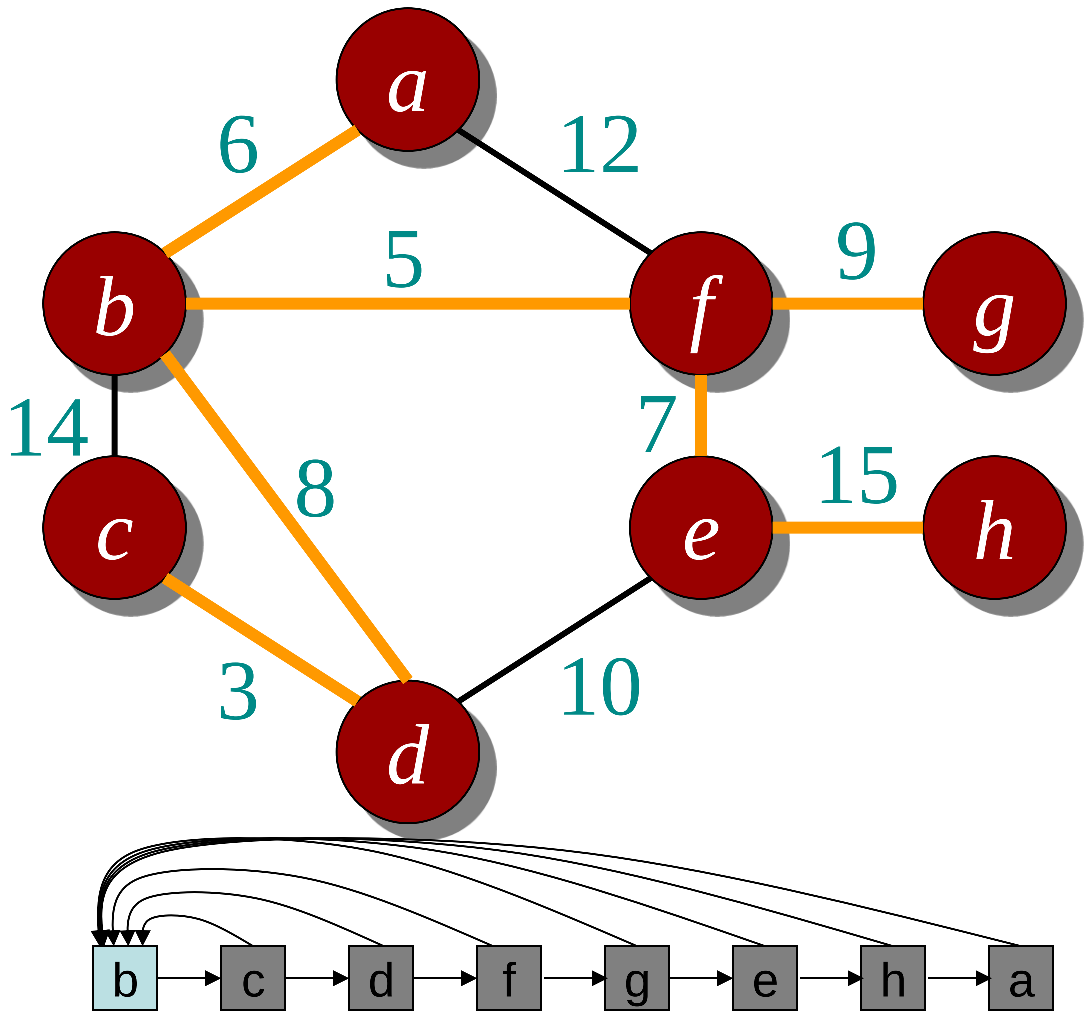
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Disjoint Set Union

- Better strategy and analysis
  - Always copy smaller list into larger list
  - Size of combined list is at least twice the size of smaller list
  - Each vertex copied at most $\log n$ times before a single tree emerges
  - Total number of copy operations for n vertex is therefore $O(n \log n)$
- Overall time complexity: $m \log n + m\,t + n\,u$
  - t: time for finding tree root is constant
  - u: time for n union operations is at most nlog (n)
  - $m \log n + m\,t + n\,u = \Theta(m \log n + m + n \log n) = \Theta(m \log n)$
- Conclusion
  - Kruskal’s algorithm runs in $\Theta(m \log n)$ for both dense and sparse graph
- How about using counting sort?
  - $\Theta(m + n \log n)$

Summary

- Kruskal's algorithm
  - ***Θ(m log n)***
  - ***Possibly Θ(m + n log n) with counting sort***
- Prim's algorithm
  - ***With priority queue : Θ(m log n)***
    - ***Assume graph represented by adj list***
  - ***With distance array : Θ(n^2)***
    - ***Adj list or adj matrix***
  - ***For sparse graphs priority queue wins***
  - ***For dense graphs distance array may be better***