COMP 1201 Algorithmics

Minimum Spanning Trees

Hikmat Farhat

Minimum Spanning Trees

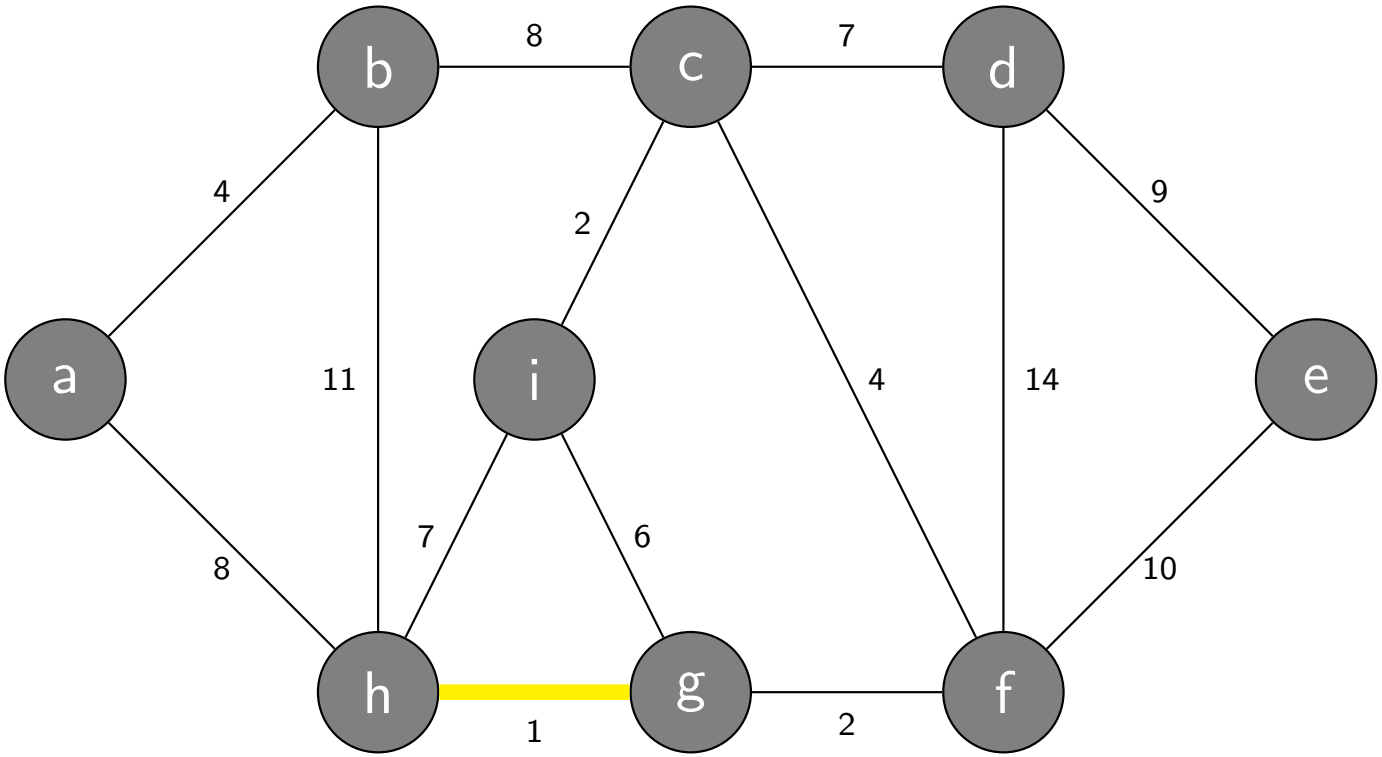
- In many application, when the system is represented by a graph we need to find a **Minimum Spanning Tree** (MST).
- typically we are given a graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$.
- We are looking for a tree $T \subseteq E$ that spans G and has the least weight. (weight of a tree is the sum of the weights of its edges).
- As the name suggest this collection of edges T is
  1. **A tree**.
  2. **Spanning**. meaning connects all the nodes of the graph.
  3. It has the **least** weight of all spanning trees.

Kruskal's Algorithm

- Kruskal's algorithm computes a MST of a given graph.
- Every edge has an associated weight or cost.
- The idea is to build the MST by adding an edge every iteration.
  1. The edges are considered by increasing order of weight.
  2. An edge is added if it doesn't create a cycle.
- The algorithm stops when there are no more edges to consider.

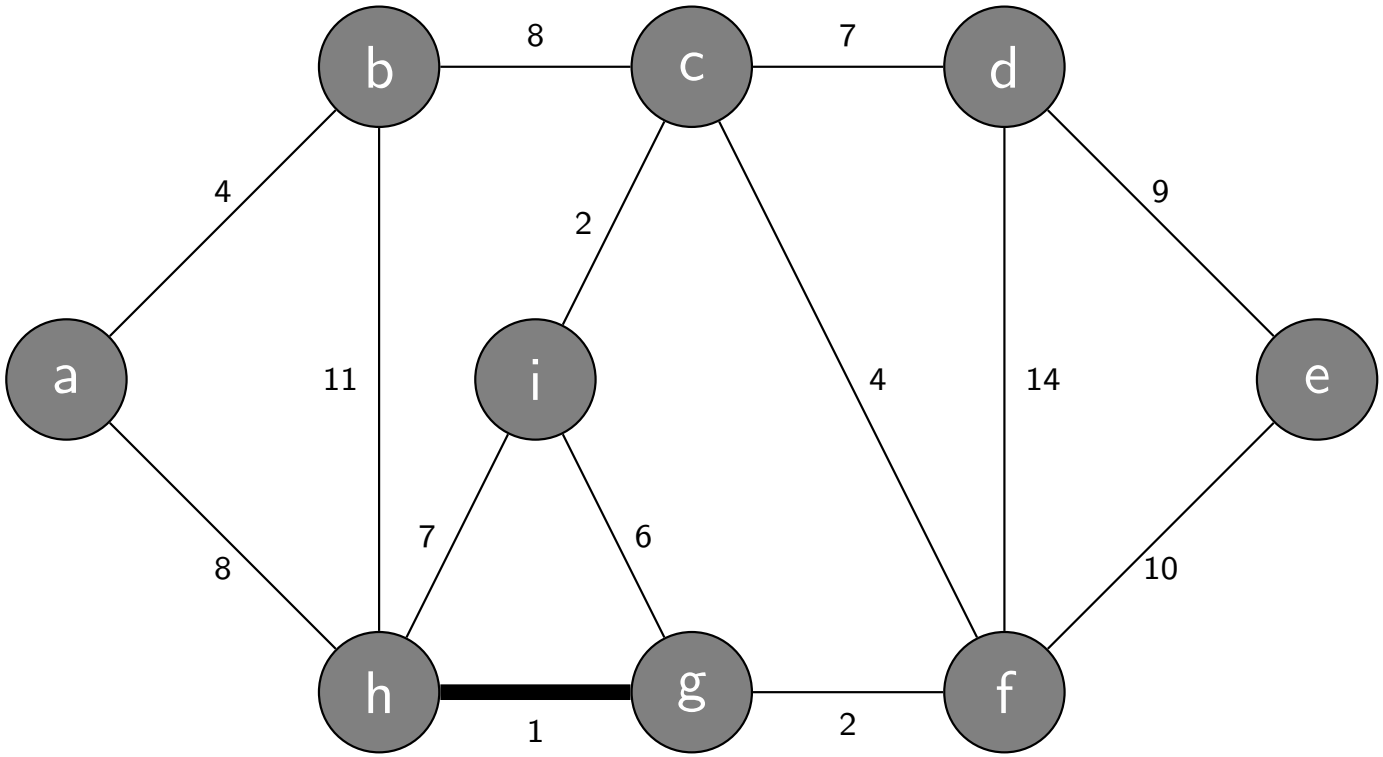
Example (Kruskal's Algorithm)

creates cycle?



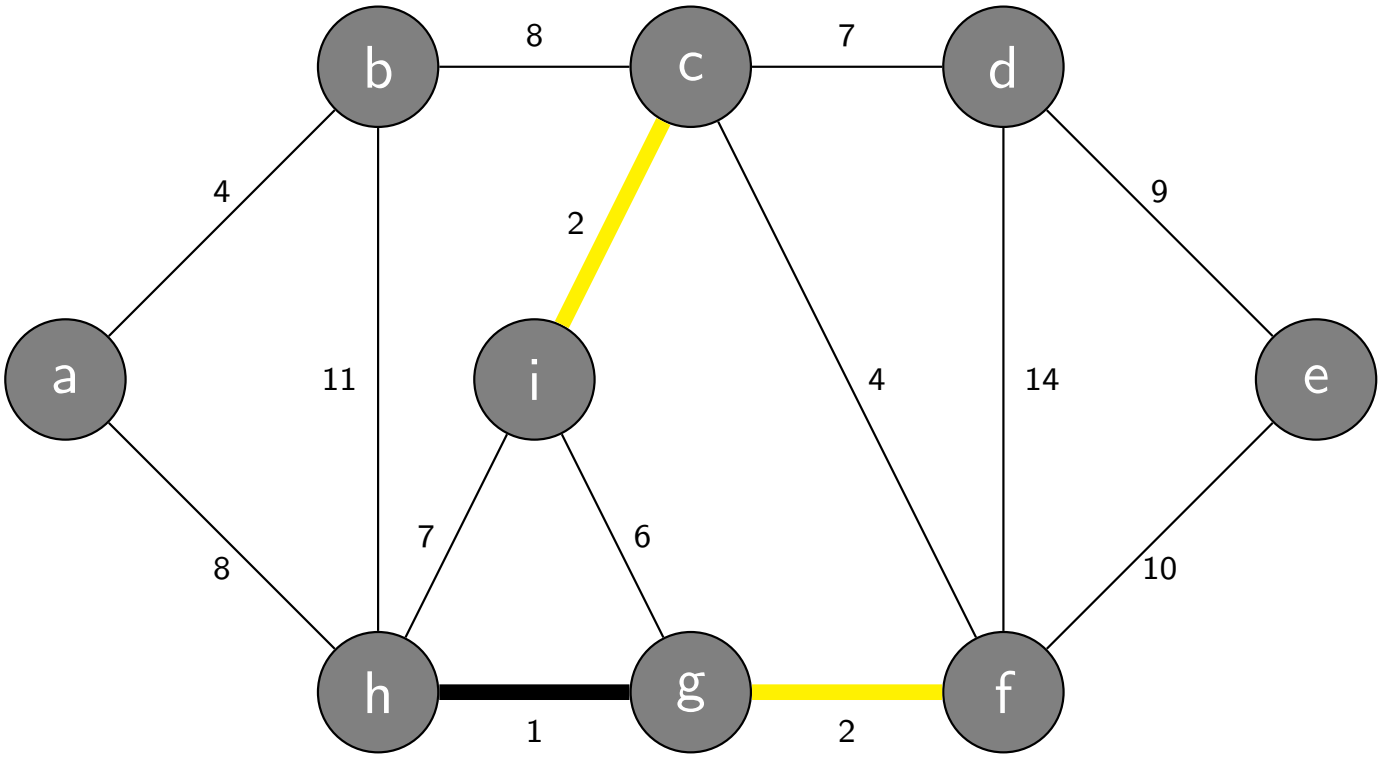
Example (Kruskal's Algorithm)

creates cycle?No, so add it to the MST



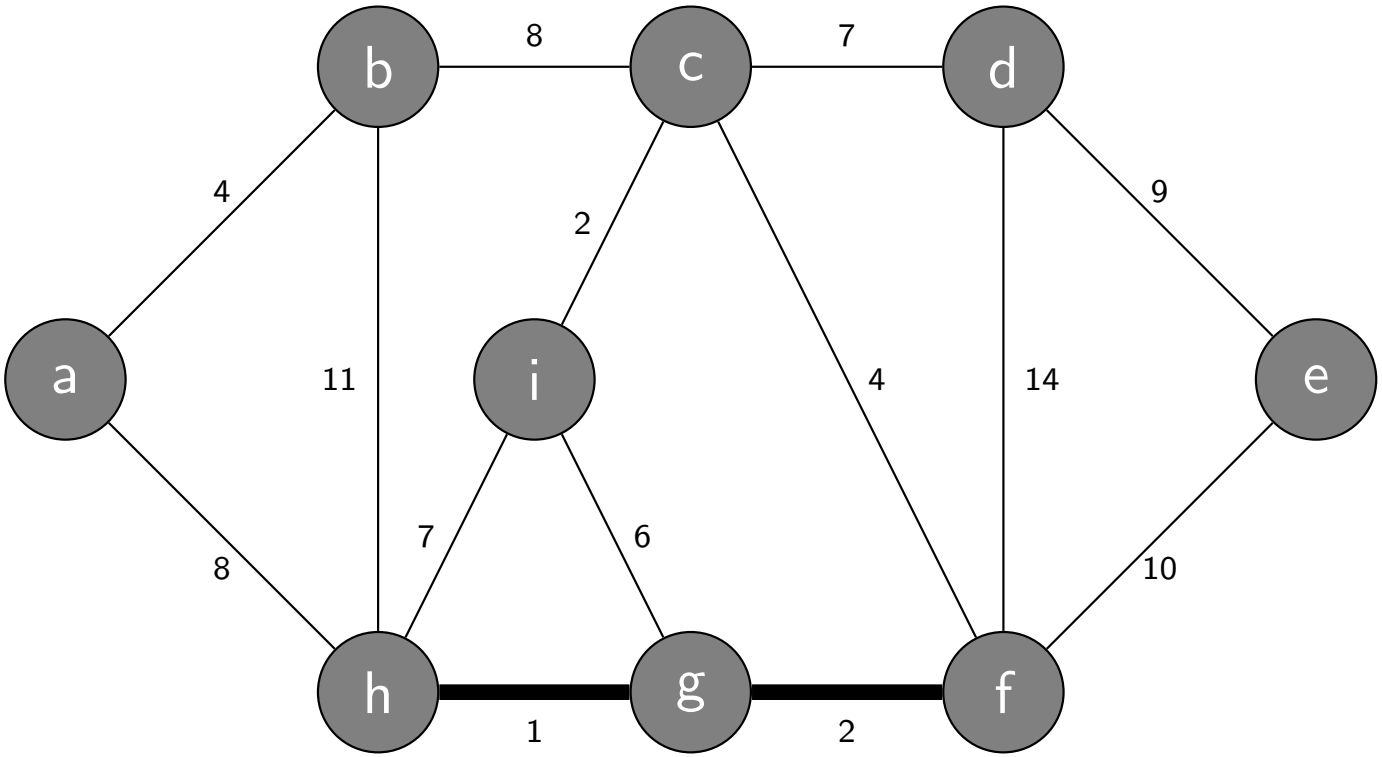
Example (Kruskal's Algorithm)

Two choices



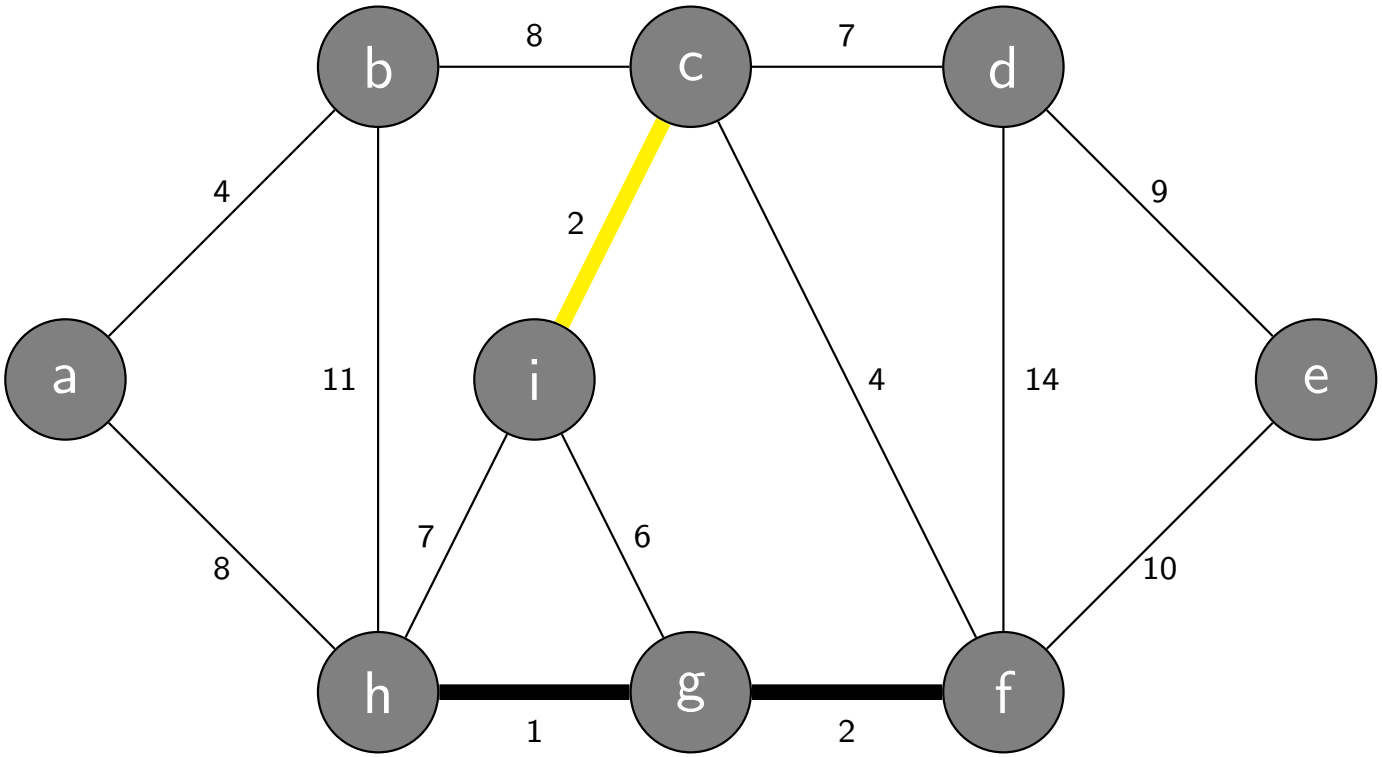
Example (Kruskal's Algorithm)

Two choices. Select one randomly



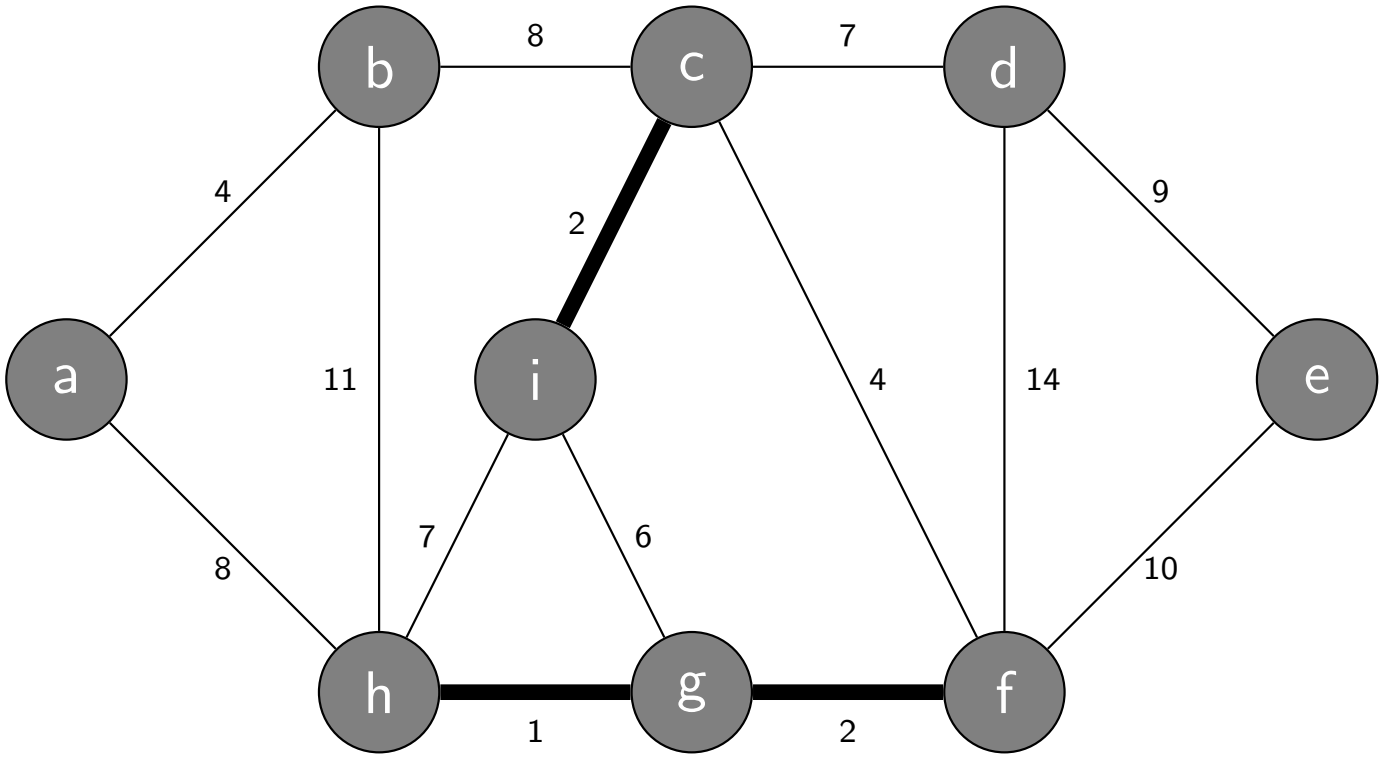
Example (Kruskal's Algorithm)

Creates cycle?



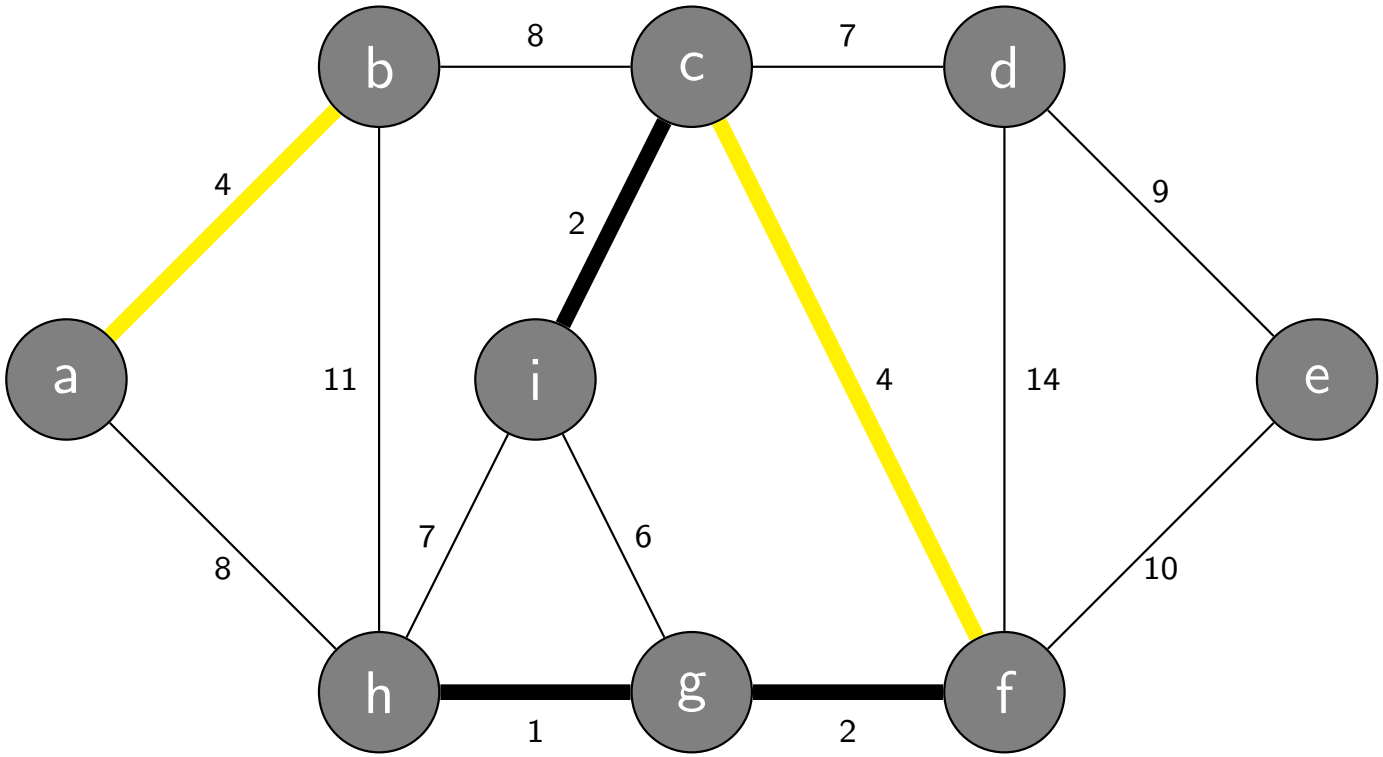
Example (Kruskal's Algorithm)

Creates cycle?No, so add it to the MST



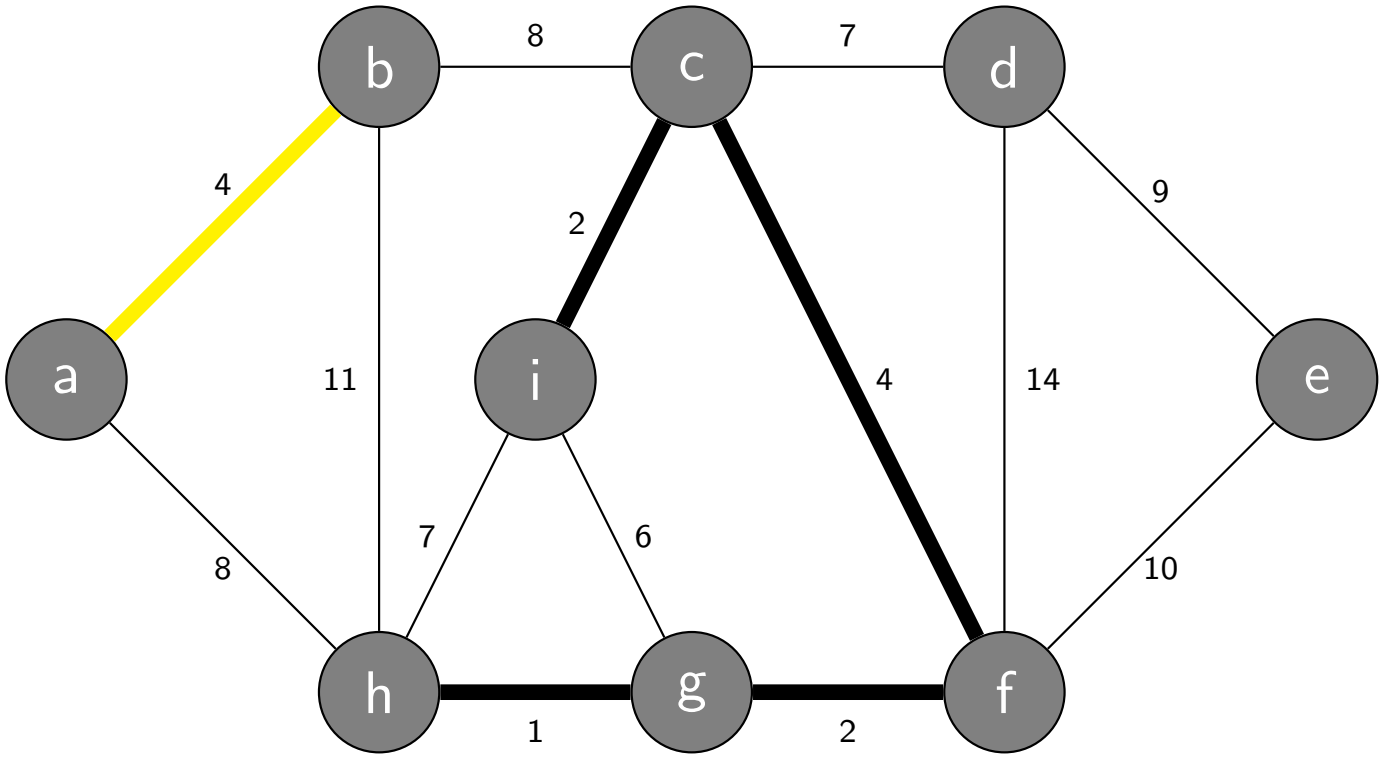
Example (Kruskal's Algorithm)

Two choices.



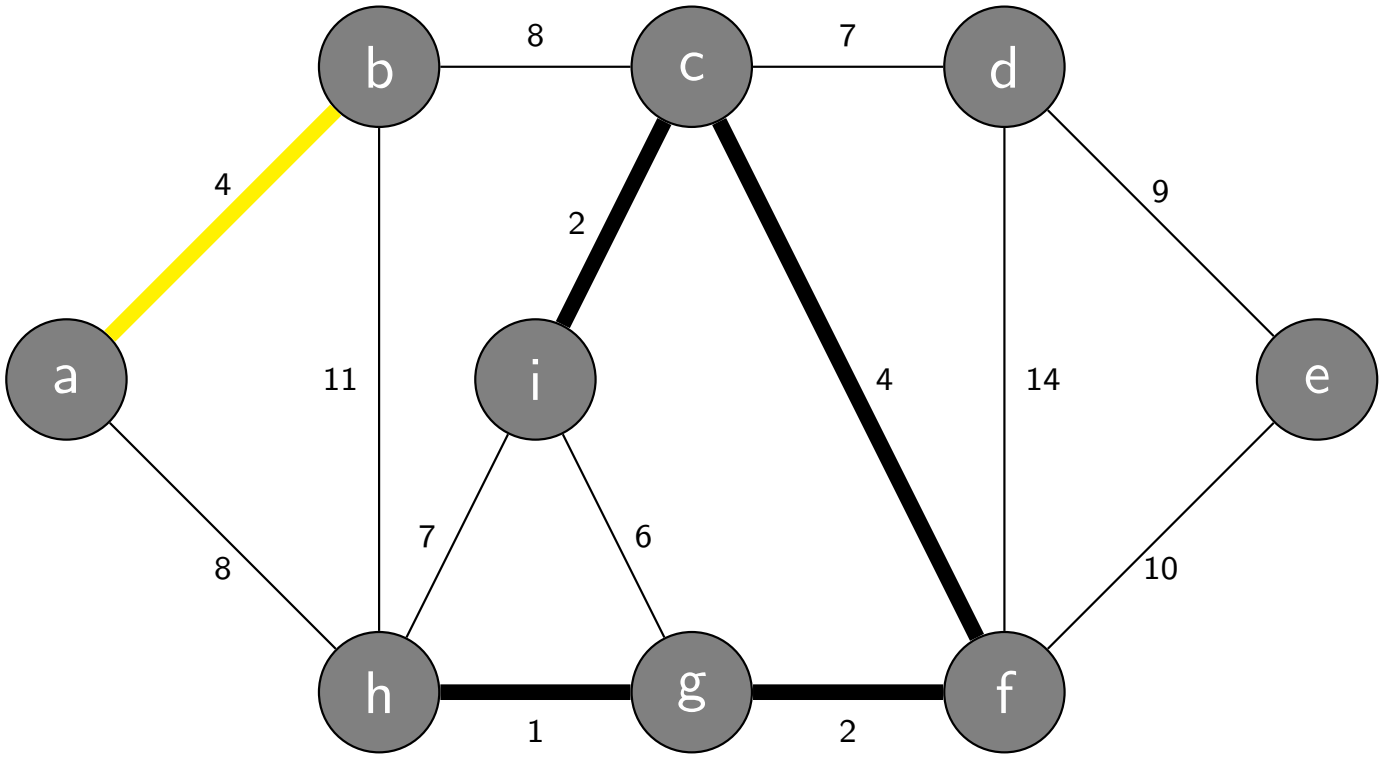
Example (Kruskal's Algorithm)

Two choices. Select one randomly



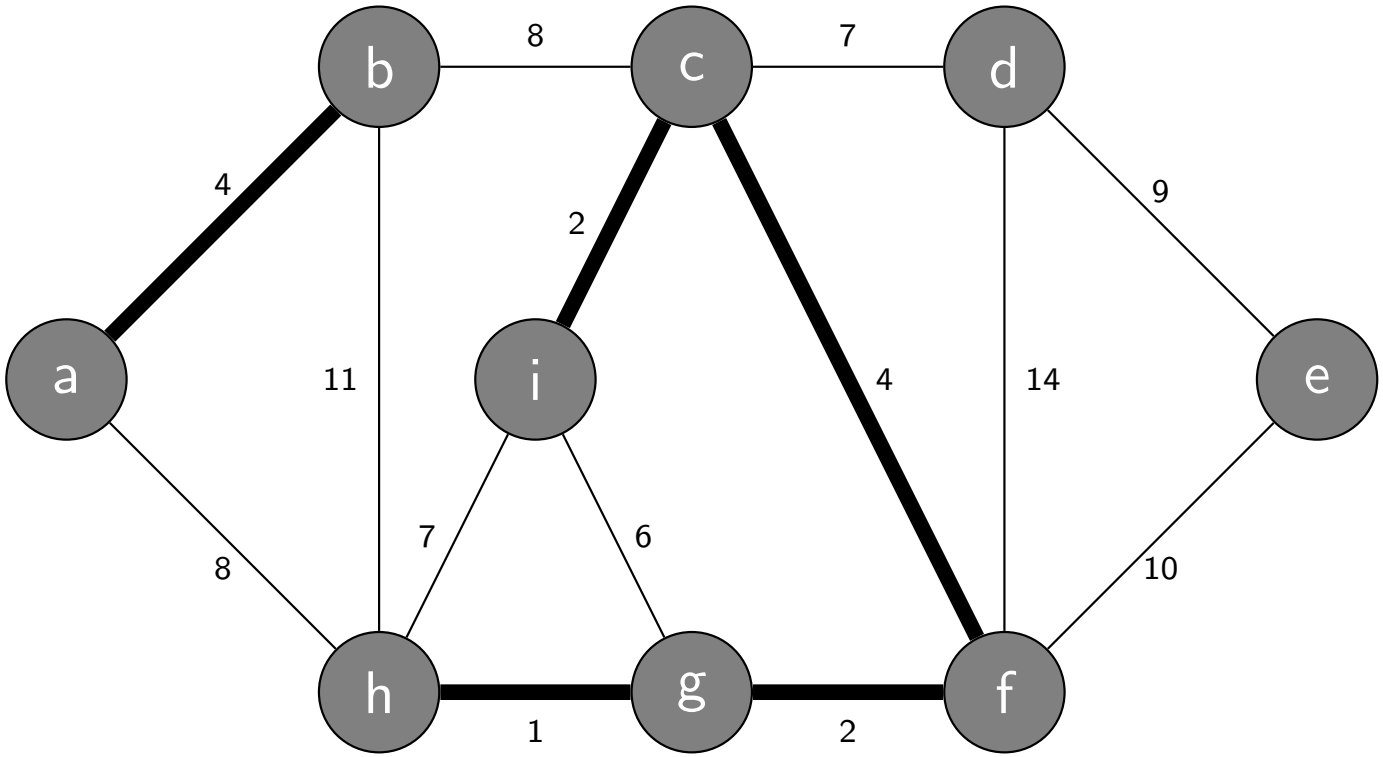
Example (Kruskal's Algorithm)

Creates cycle?



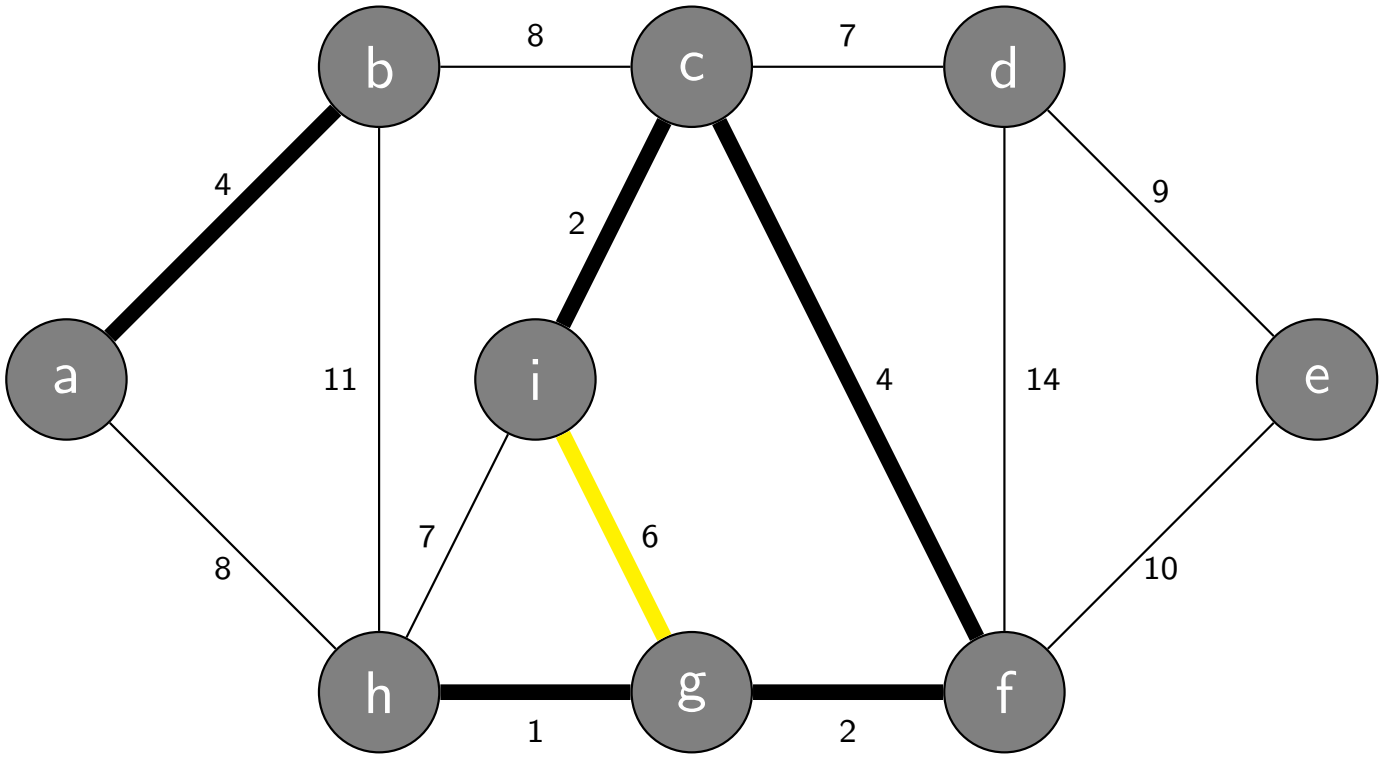
Example (Kruskal's Algorithm)

Creates cycle?No, so add it to the MST



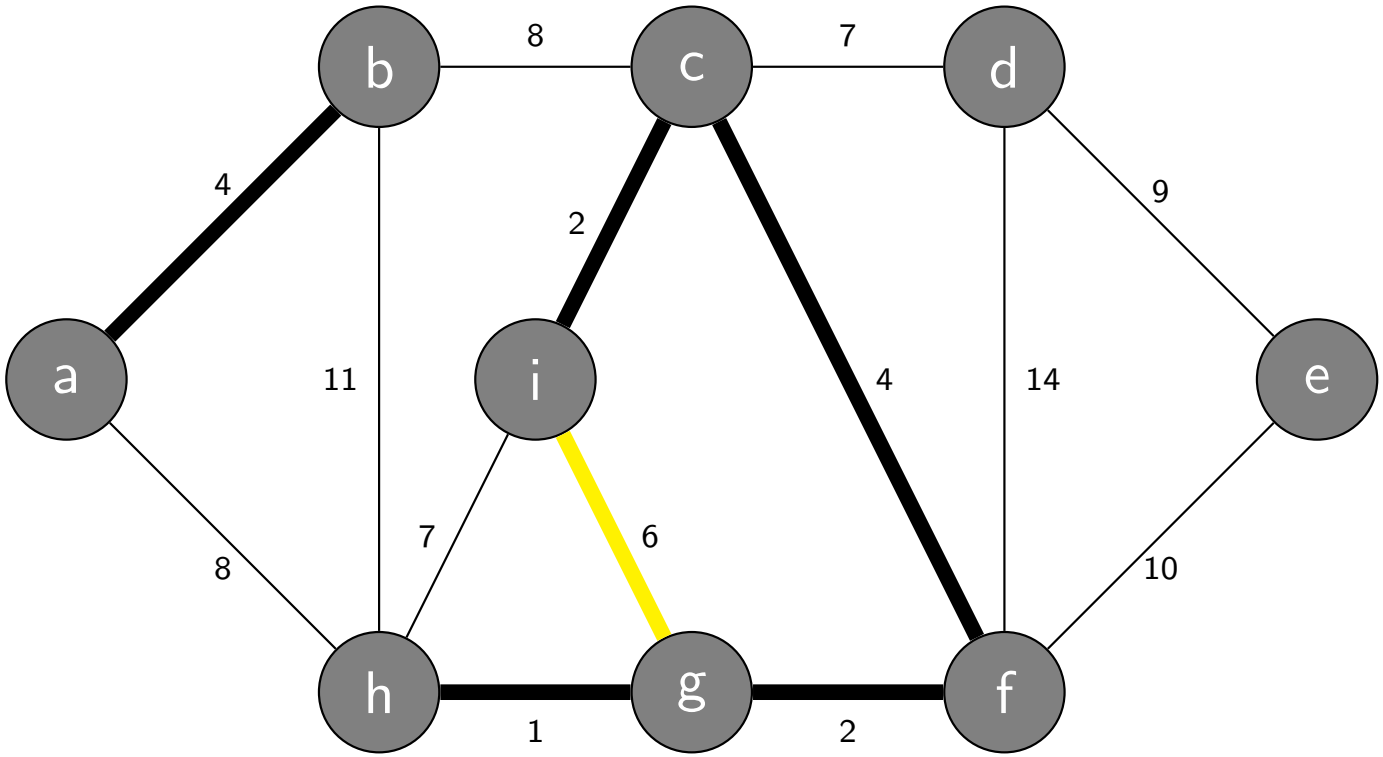
Example (Kruskal's Algorithm)

Creates cycle?



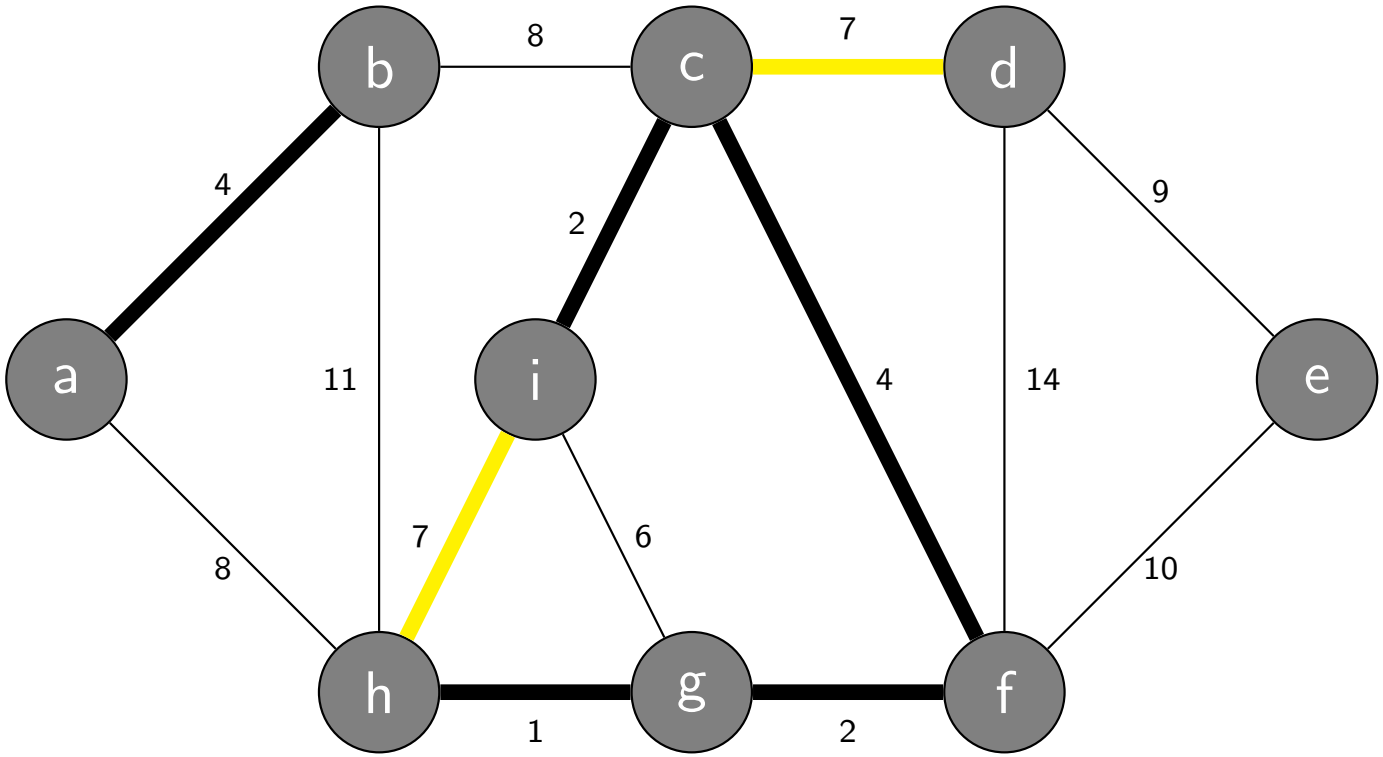
Example (Kruskal's Algorithm)

Creates cycle?Yes, skip this edge



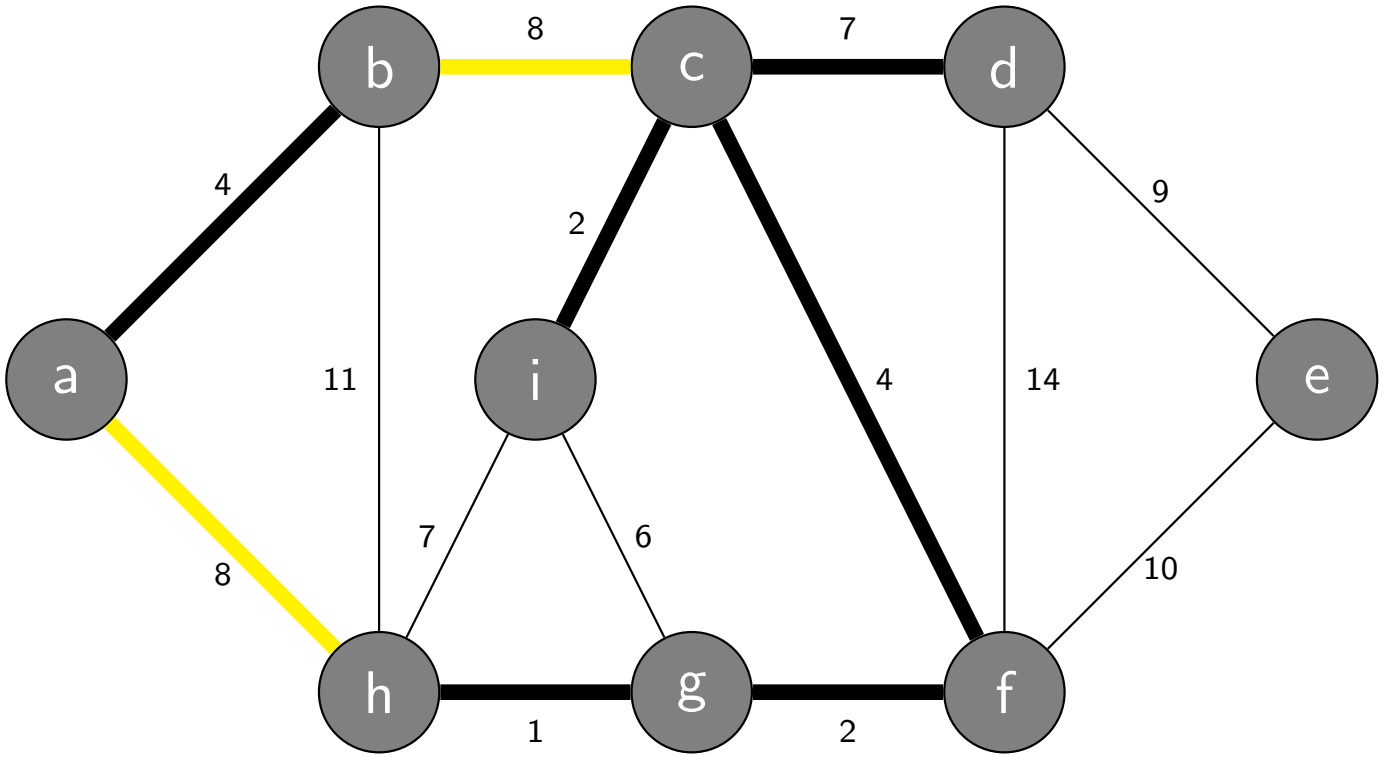
Example (Kruskal's Algorithm)

Two possibilities but one creates a cycle



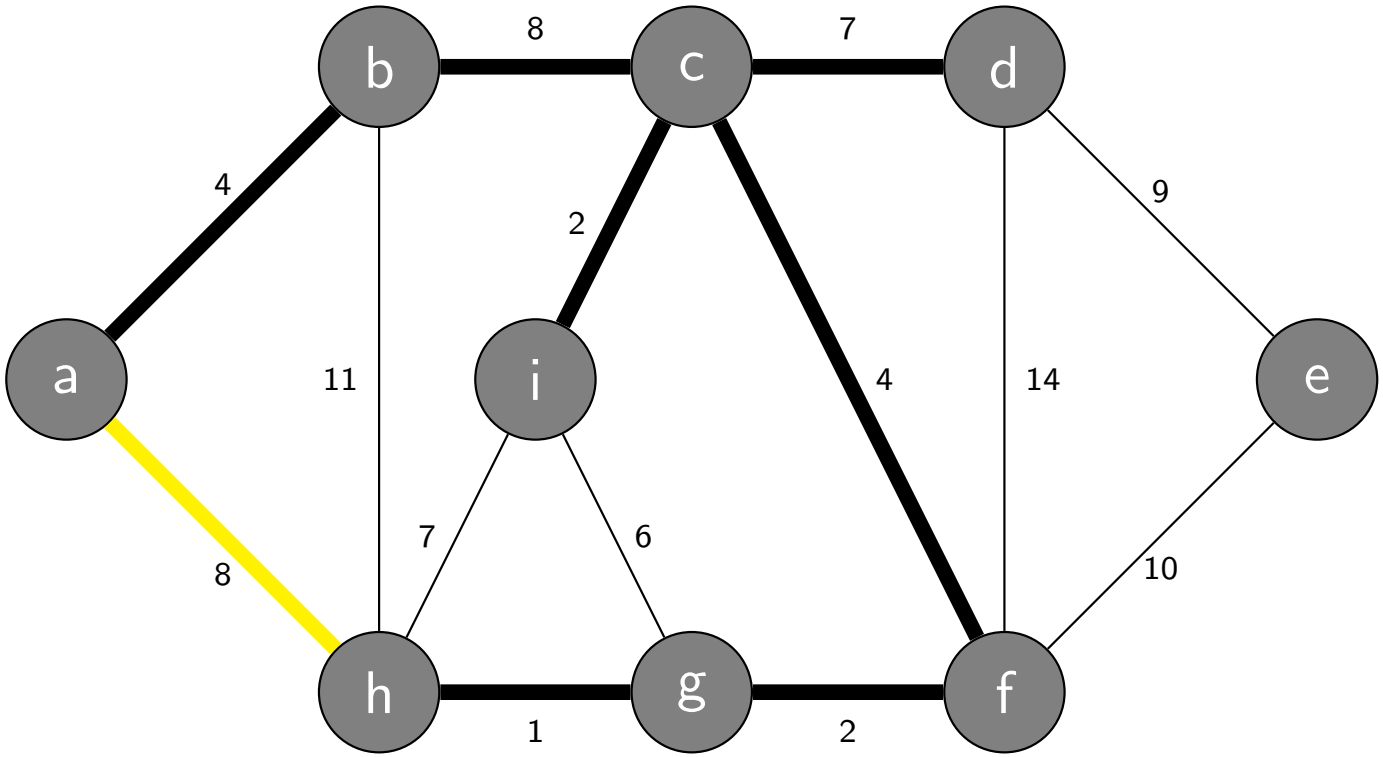
Example (Kruskal's Algorithm)

Two possibilities. Select one randomly.



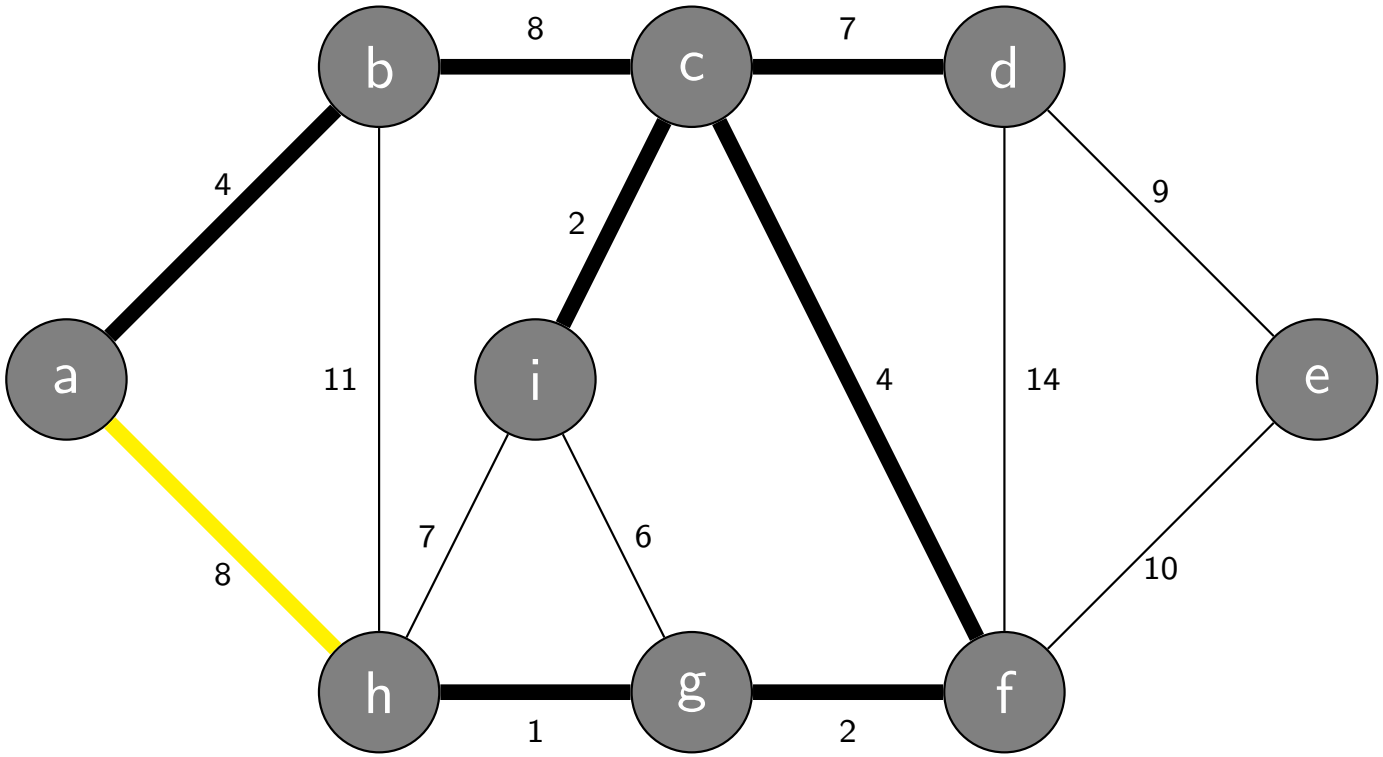
Example (Kruskal's Algorithm)

Creates cycle?



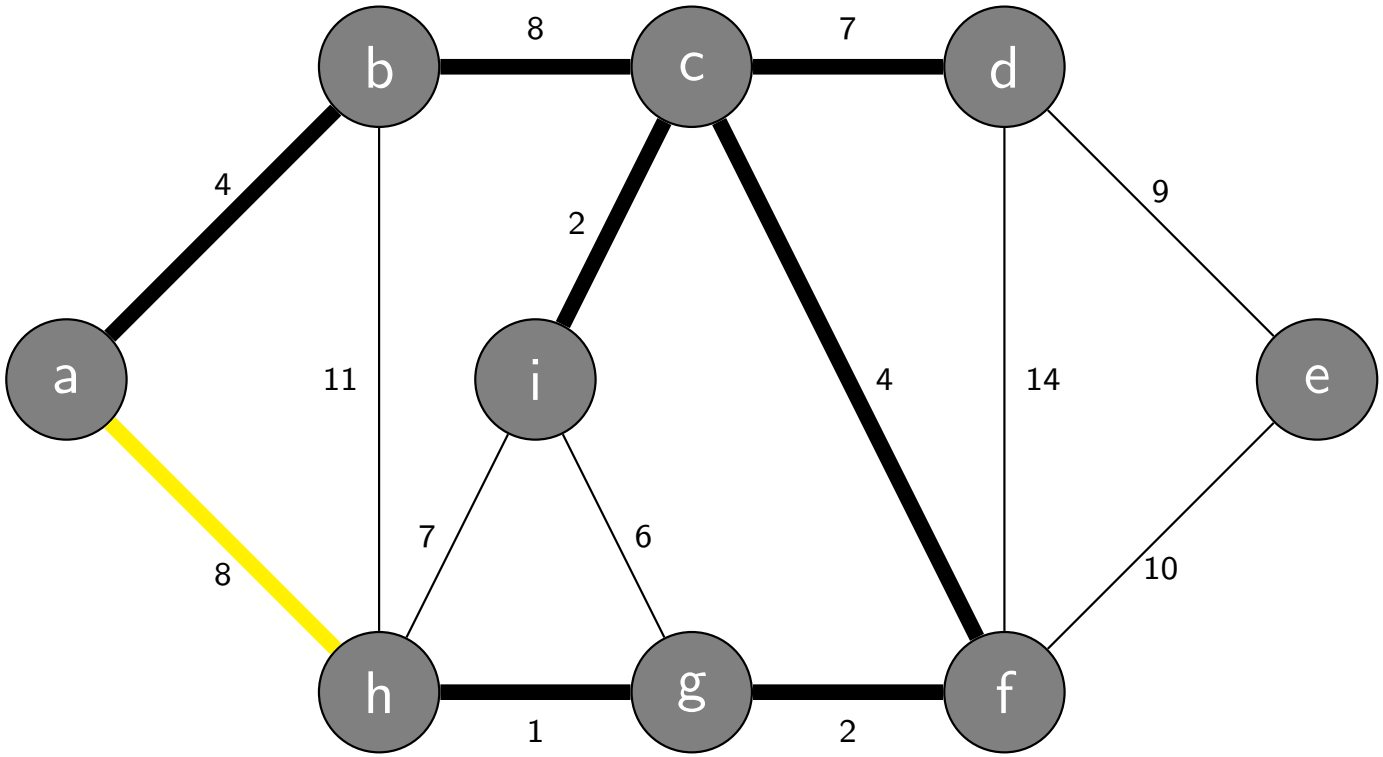
Example (Kruskal's Algorithm)

Creates cycle? Yes, skip this edge.



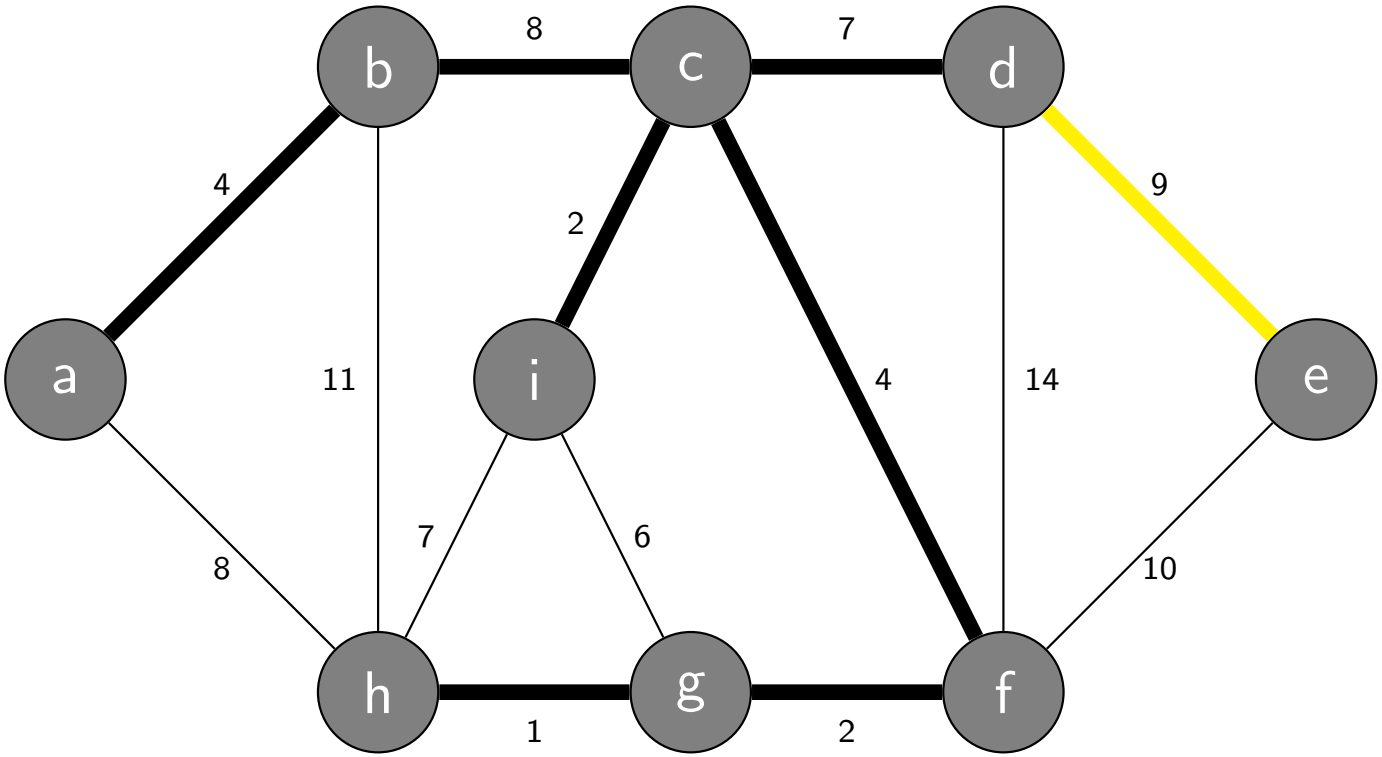
Example (Kruskal's Algorithm)

What if we had selected a-h before b-c?



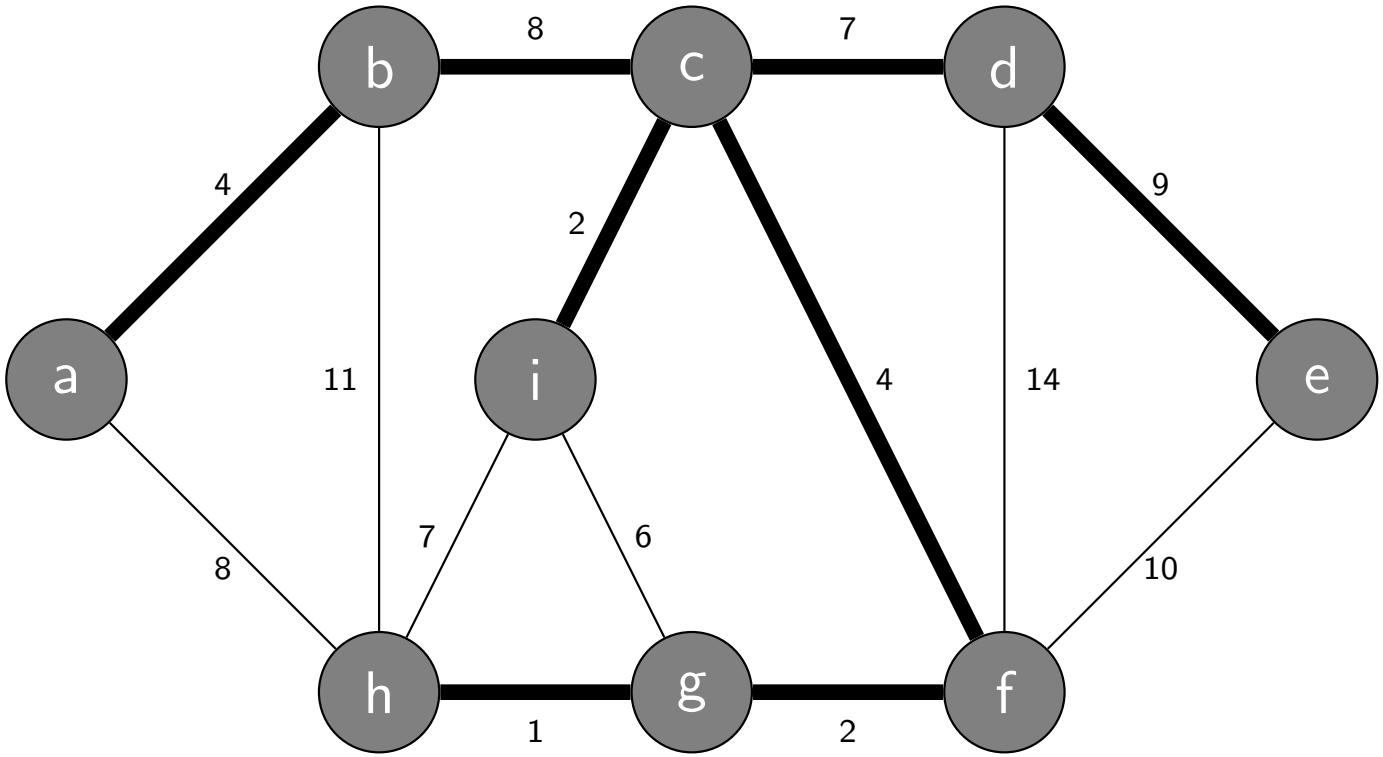
Example (Kruskal's Algorithm)

Creates cycle?



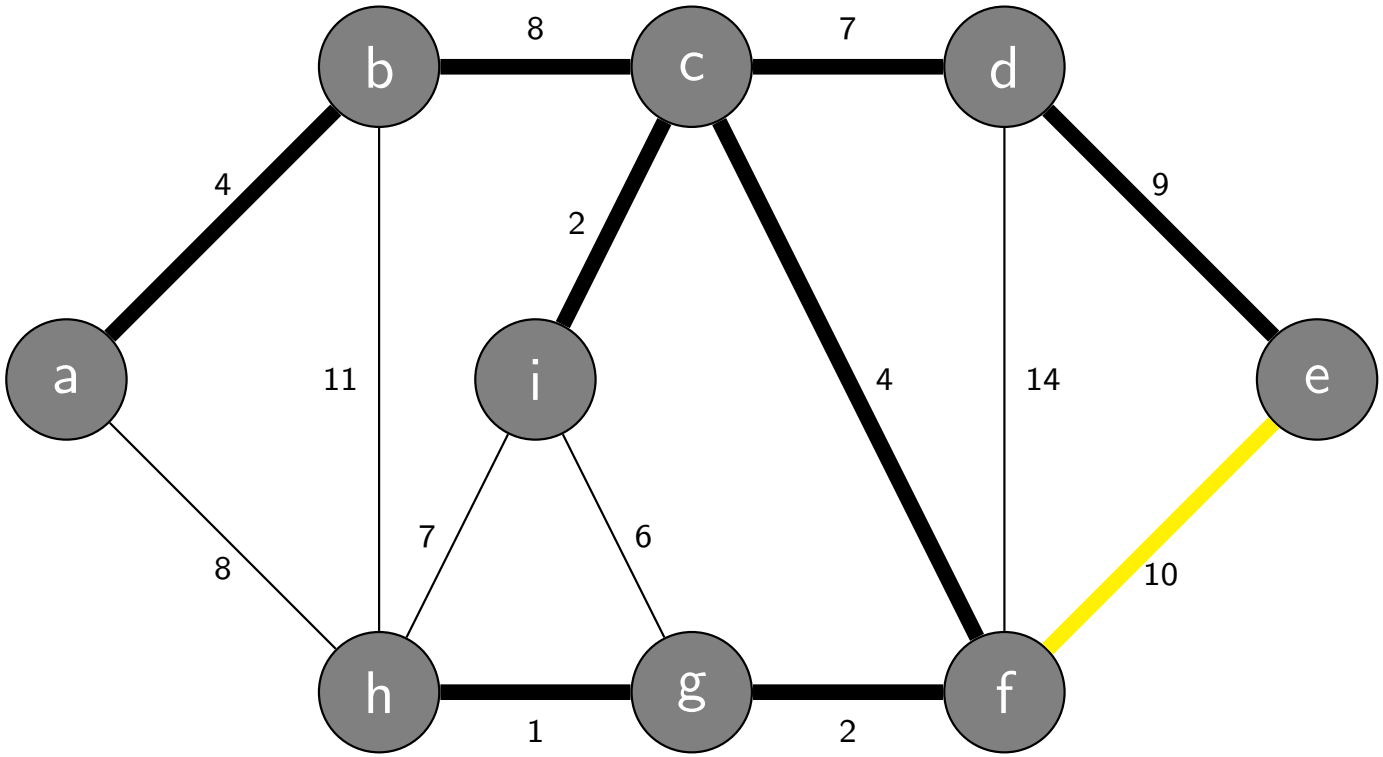
Example (Kruskal's Algorithm)

Creates cycle?No, so add it to the MST.



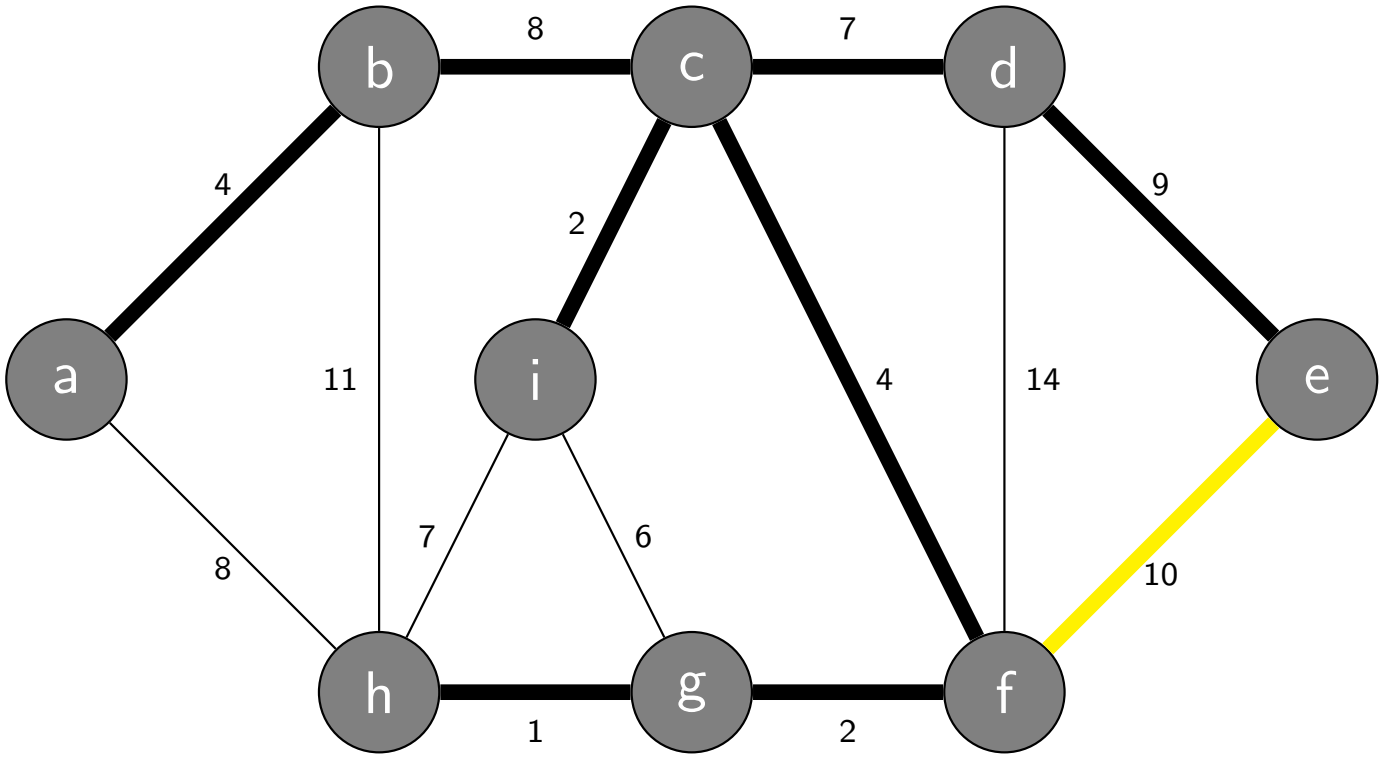
Example (Kruskal's Algorithm)

Creates cycle?



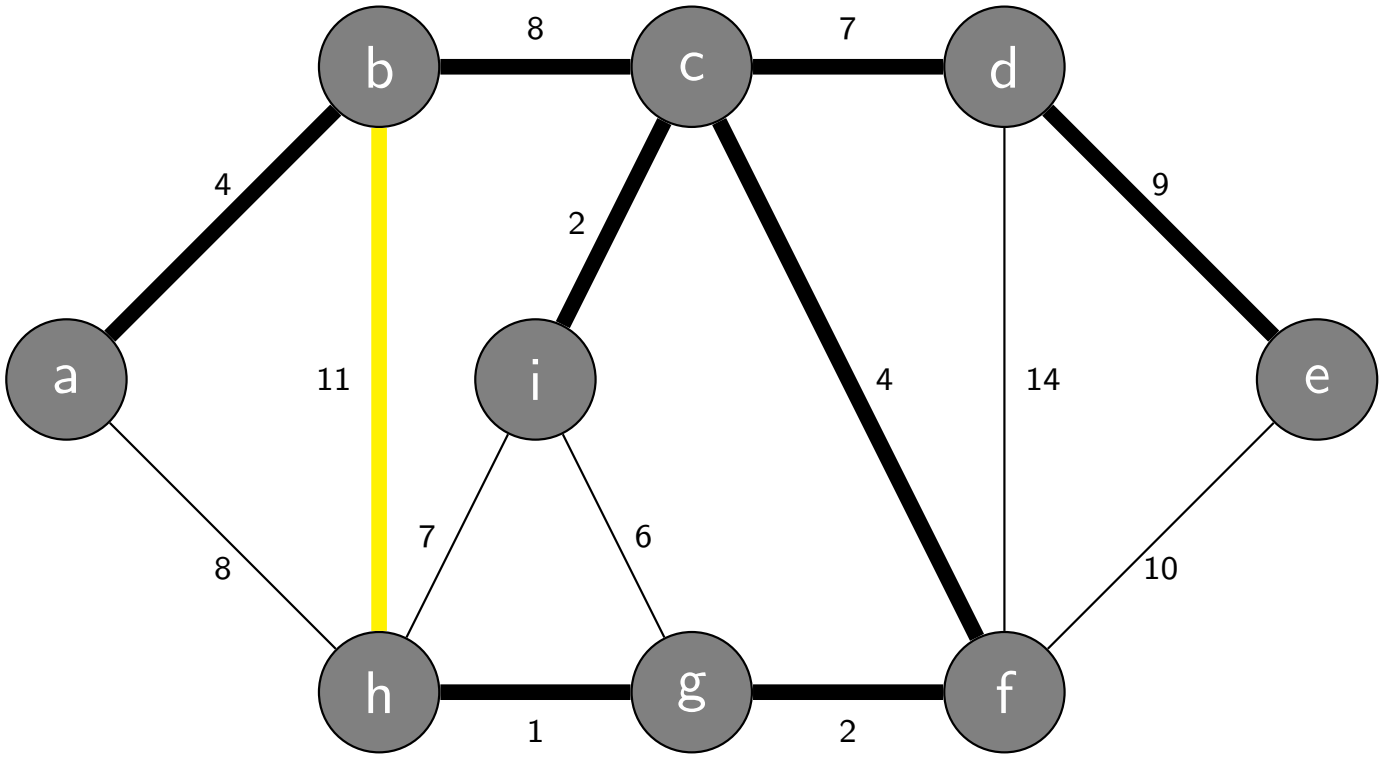
Example (Kruskal's Algorithm)

Creates cycle?Yes so skip this edge.



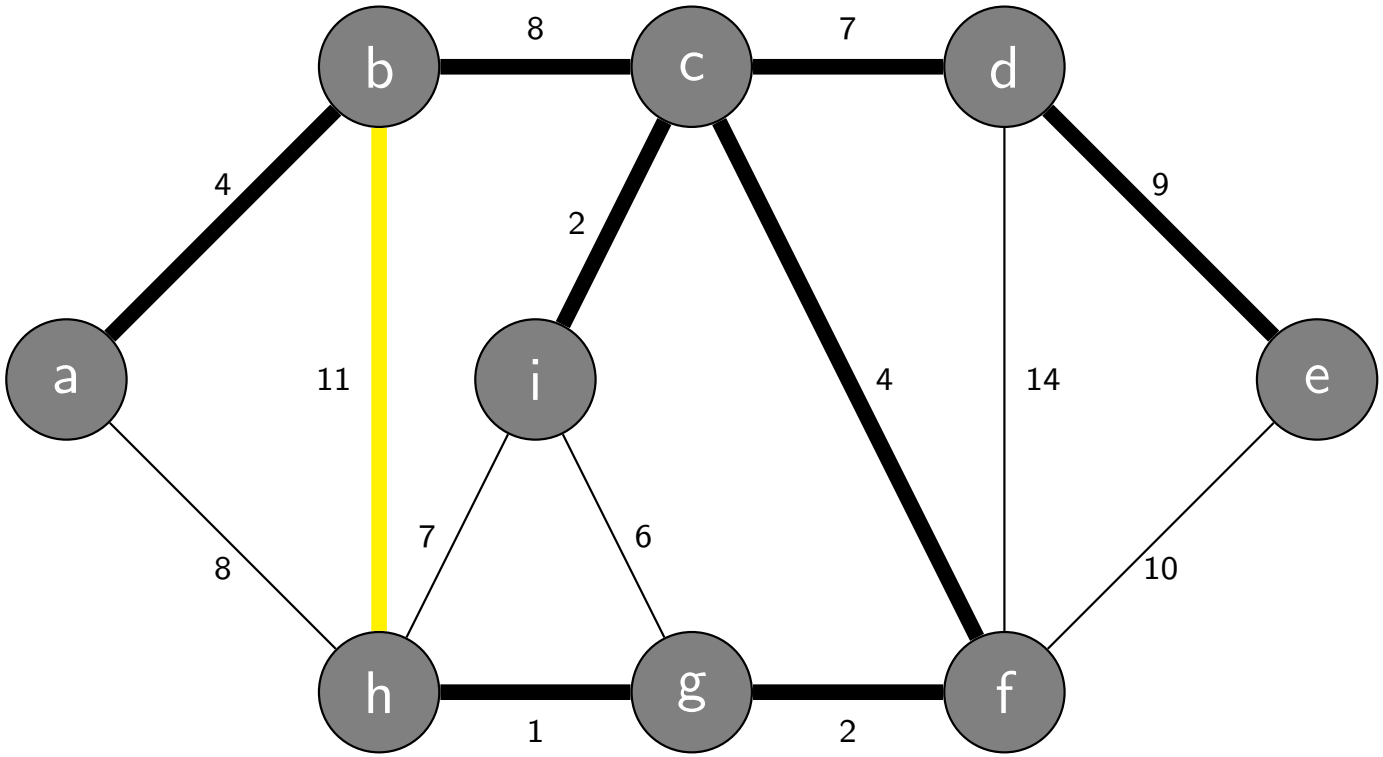
Example (Kruskal's Algorithm)

Creates cycle?



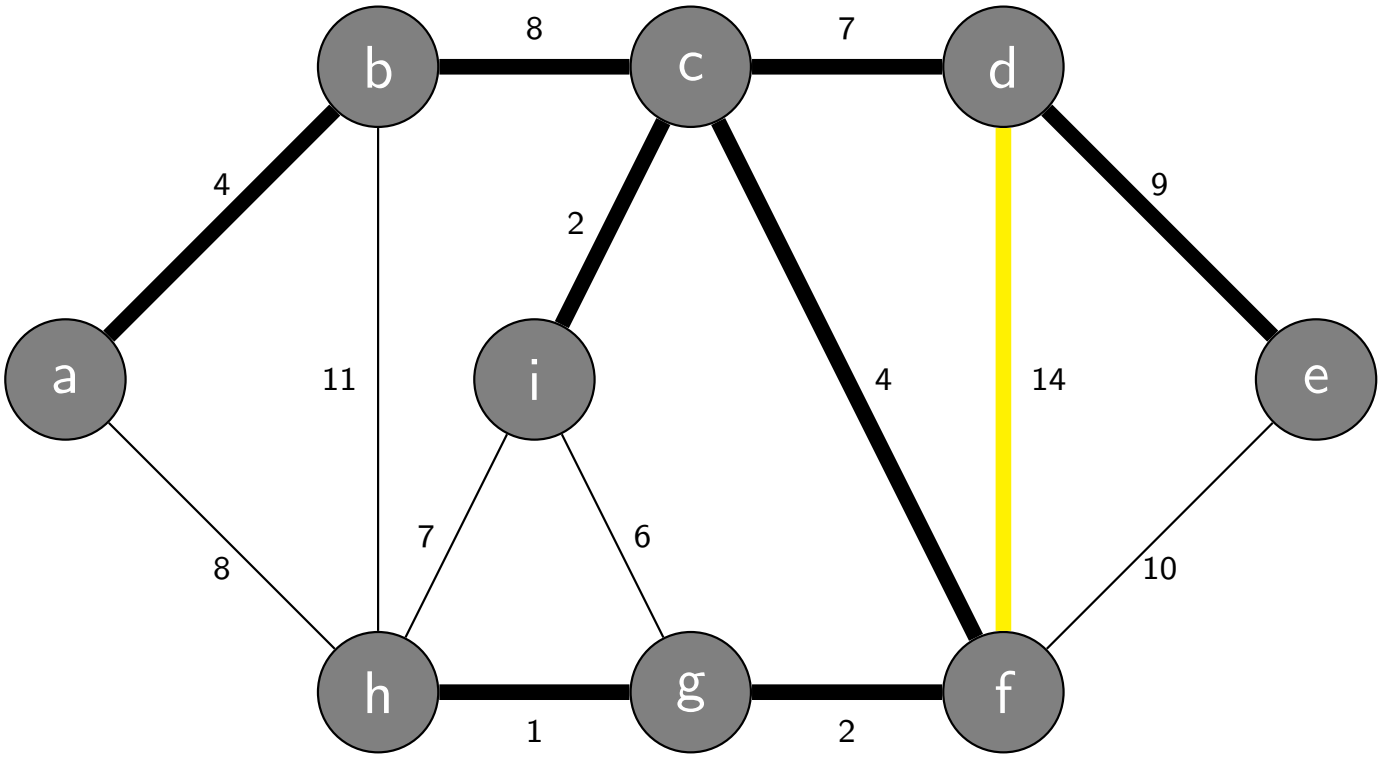
Example (Kruskal's Algorithm)

Creates cycle? Yes so skip this edge.



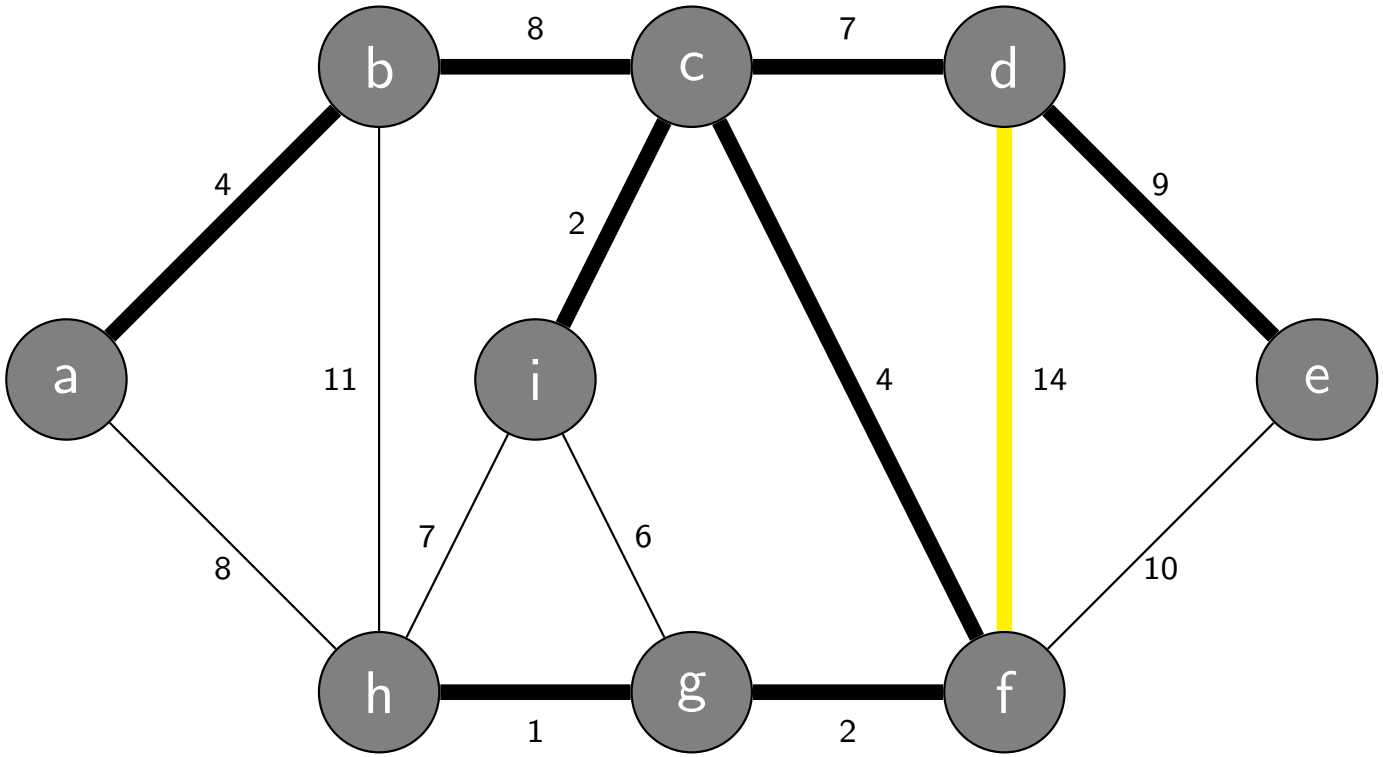
Example (Kruskal's Algorithm)

Creates cycle?



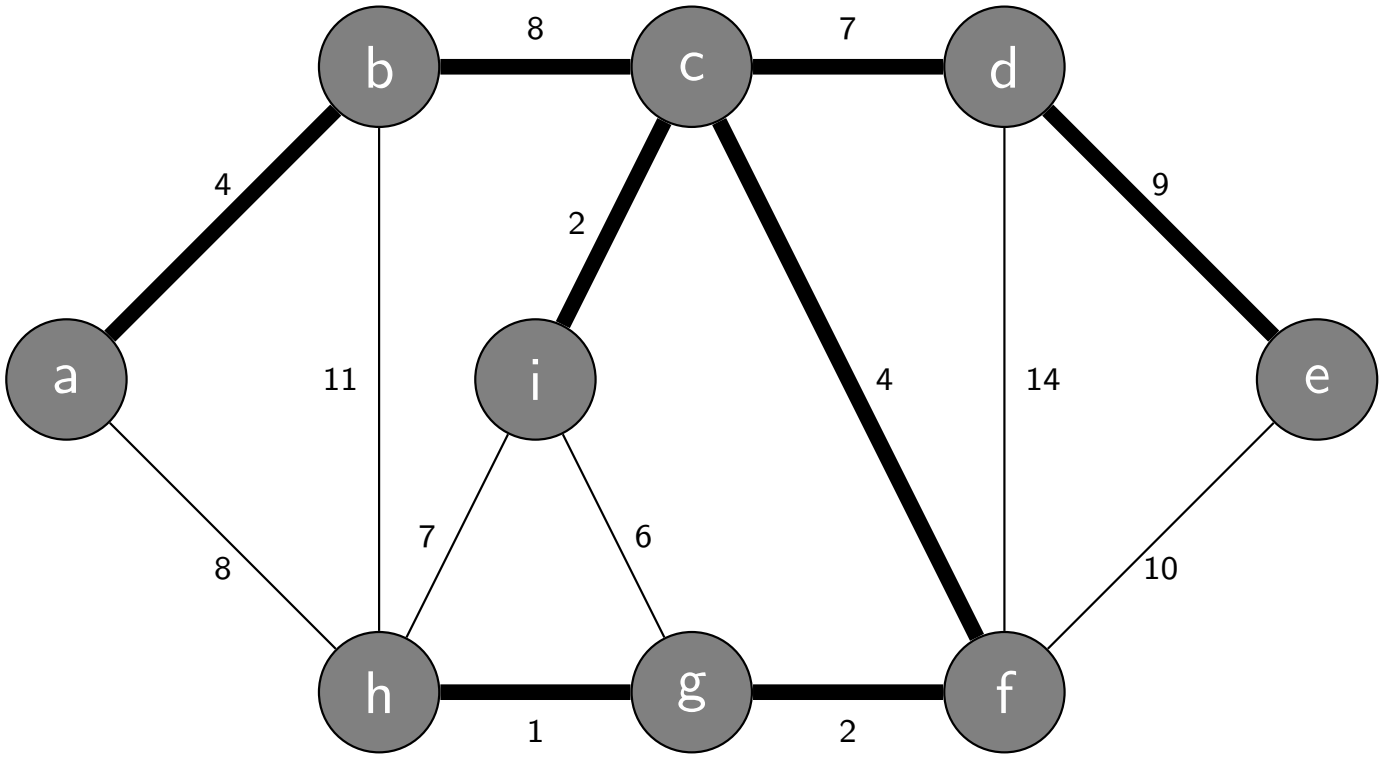
Example (Kruskal's Algorithm)

Creates cycle? Yes so skip this edge.



Example (Kruskal's Algorithm)

No more edges. MST is complete



High level implementation

- Given a graph $G = (V, E)$ with a weight function $w : E \rightarrow R$.
- We build an MST T by adding edges to T one at a time.
- One can iterate over the edges in increasing weight order by sorting E.
- We need a method to check if adding an edge to the current partial solution creates a cycle.
- This can be done by using the union-find data structure.

Disjoint Sets Data Structures

- Given a graph $G = (V, E)$ and arbitrary vertex $v \in V$.
- MAKE-SET(v): create a new set whose only member is v.
- FIND-SET(v):returns a pointer to the representative of the set containing v.
- UNION(u,v):combine the sets containing u and v into a new set.
- An edge $(u, v) \in E$ creates a cycles iff FIND-SET(u) =FIND-SET(v).

Algorithm 1: Kruskal's Algorithm

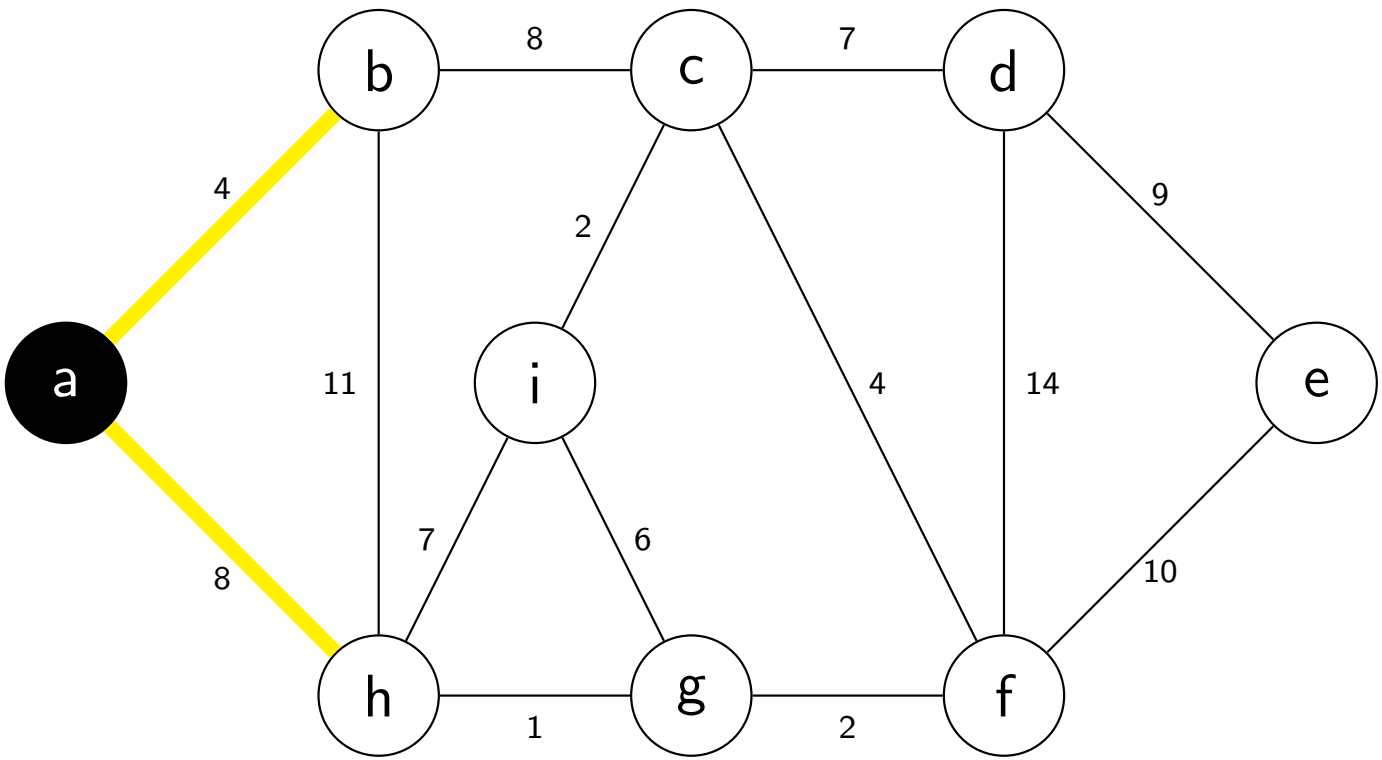
```
MST-KRUSKAL(G)
T ← ∅
foreach v ∈ V do
    MAKE-SET(v)
F ← SORT-EDGES(E)
foreach (u, v) ∈ F do
    if FIND-SET(u) ≠ FIND-SET(v) then
        T ← T ∪ {(u, v)}
        UNION(u, v)
```

Prim's algorithm

- Whereas Kruskal's algorithm is "edge" based, Prim's algorithm is "vertex" based.
- Prim's maintains a set S of vertices, initially containing a single vertex s which could be any vertex in V.
- At each iteration we consider the sets S and $V - S$.
- Find the edge (u, v) with minimum weight such that $u \in S$ and $v \in V - S$.
- Add v to S and (u, v) to T.

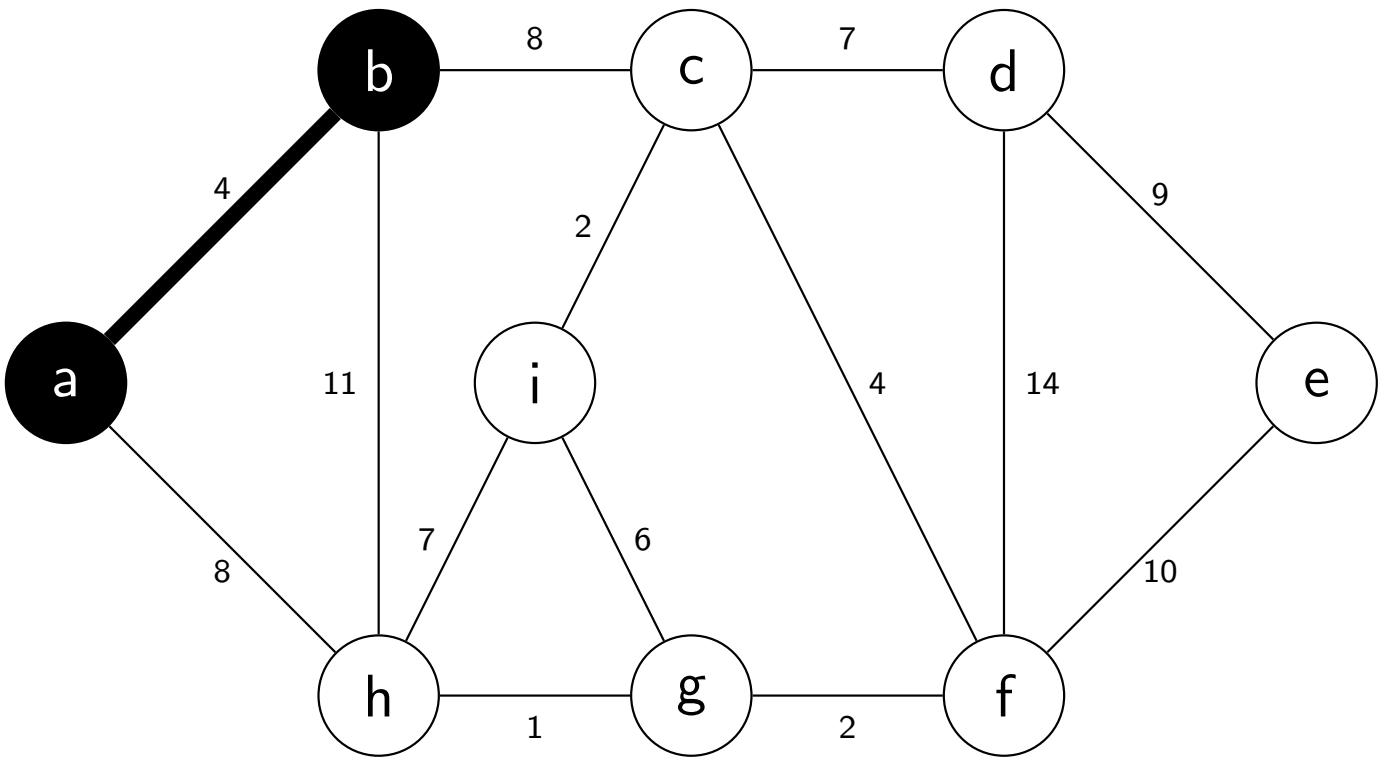
Example (Prim's Algorithm)

$S = \{a\} \quad V - S = \{b, c, d, e, f, g, h, i\}$



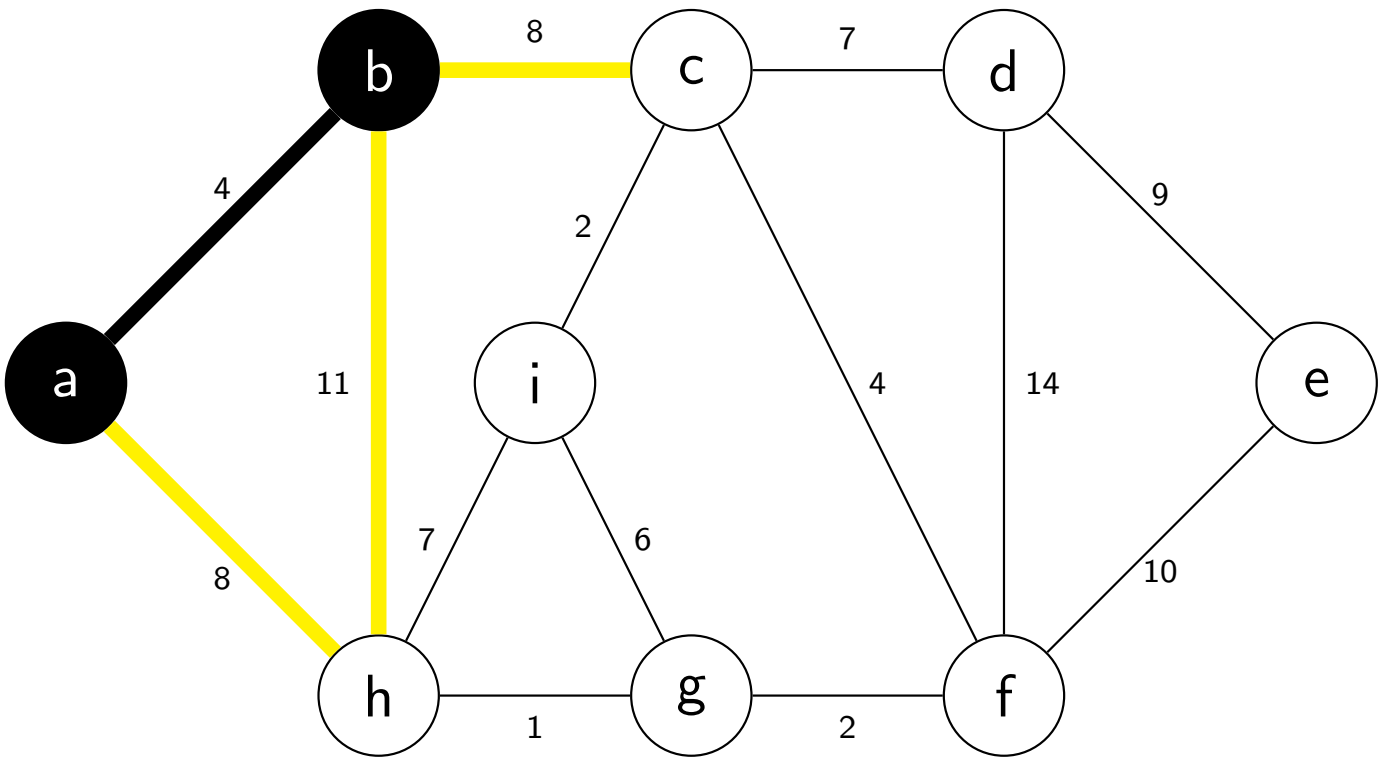
Example (Prim's Algorithm)

$S = \{a, b\} \quad V - S = \{b, c, d, e, f, g, h, i\}$



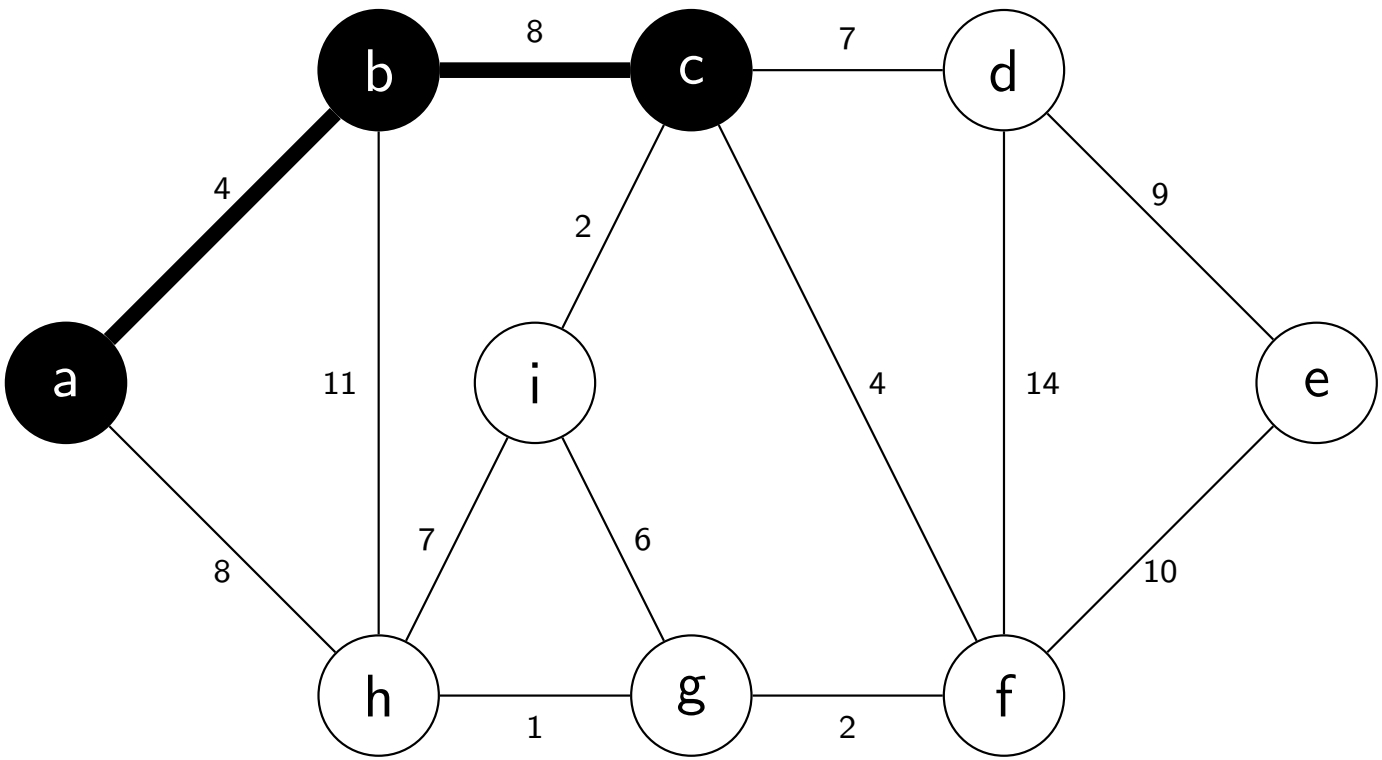
Example (Prim's Algorithm)

$S = \{a, b\} \quad V - S = \{c, d, e, f, g, h, i\}$



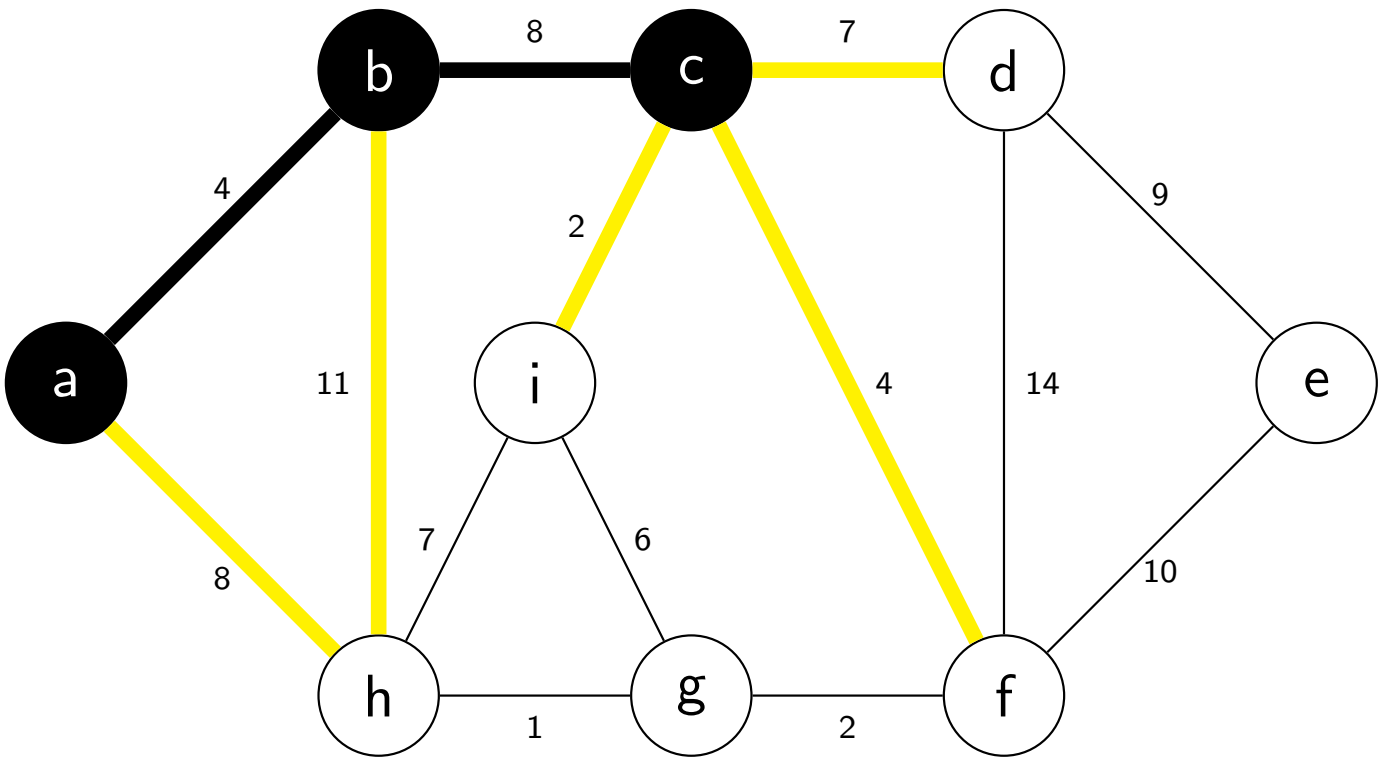
Example (Prim's Algorithm)

$S = \{a, b, c\} \quad V - S = \{d, e, f, g, h, i\}$



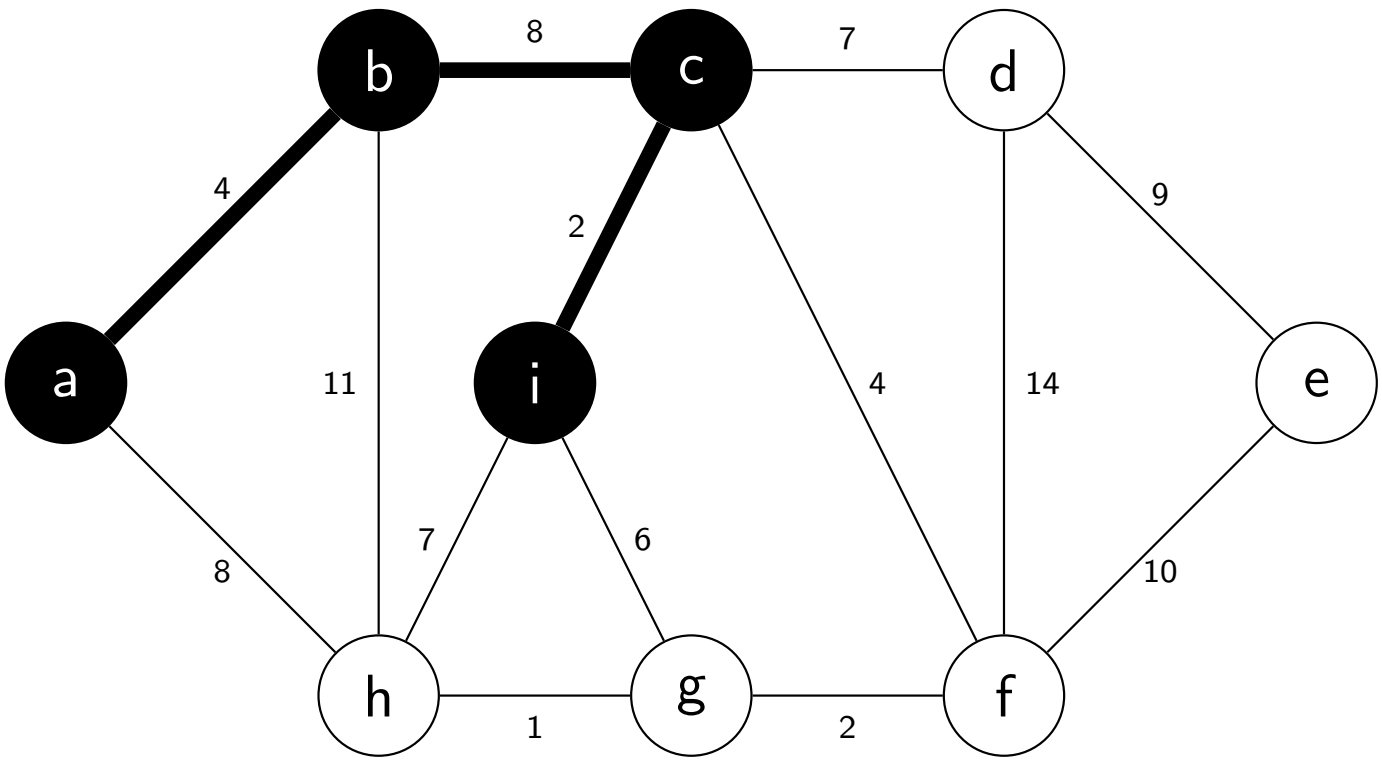
Example (Prim's Algorithm)

$S = \{a, b, c\} \quad V - S = \{d, e, f, g, h, i\}$



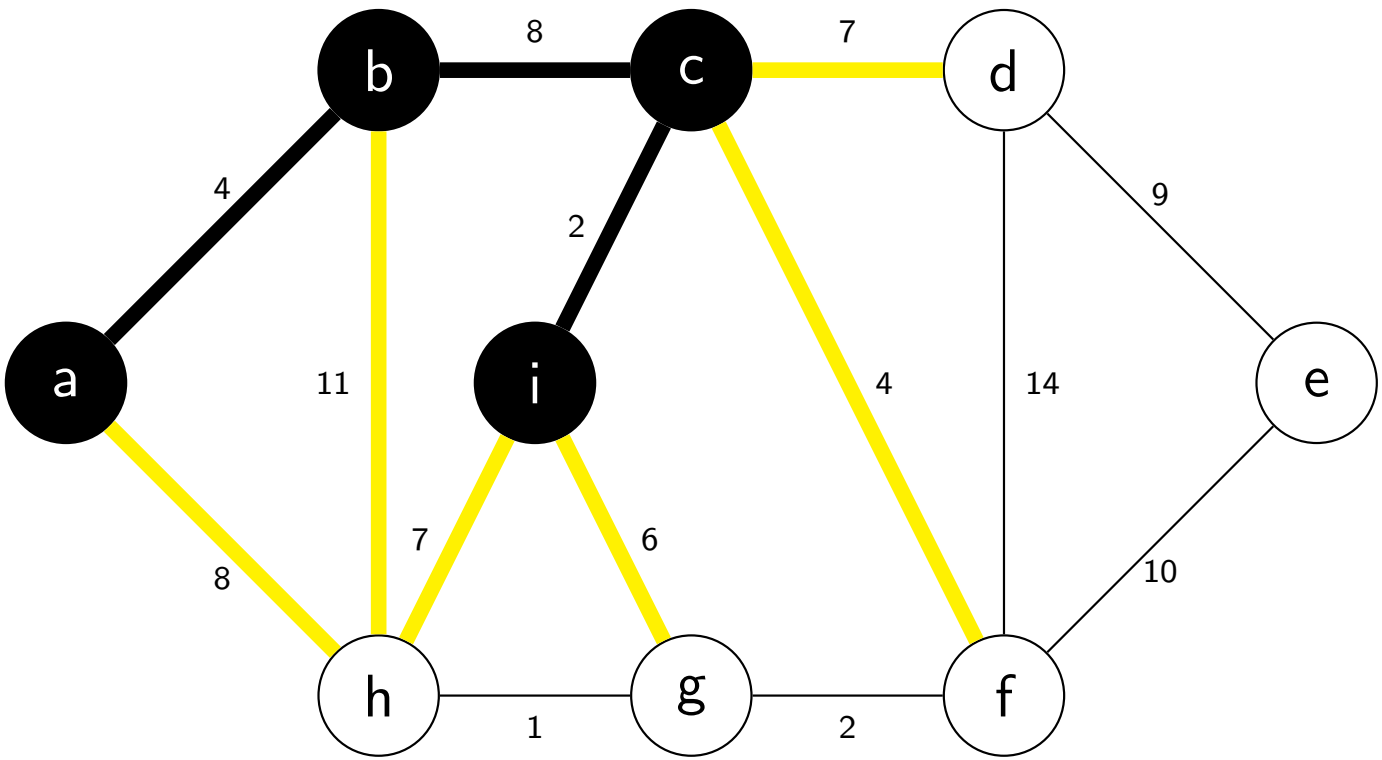
Example (Prim's Algorithm)

$S = \{a, b, c, i\} \quad V - S = \{d, e, f, g, h\}$



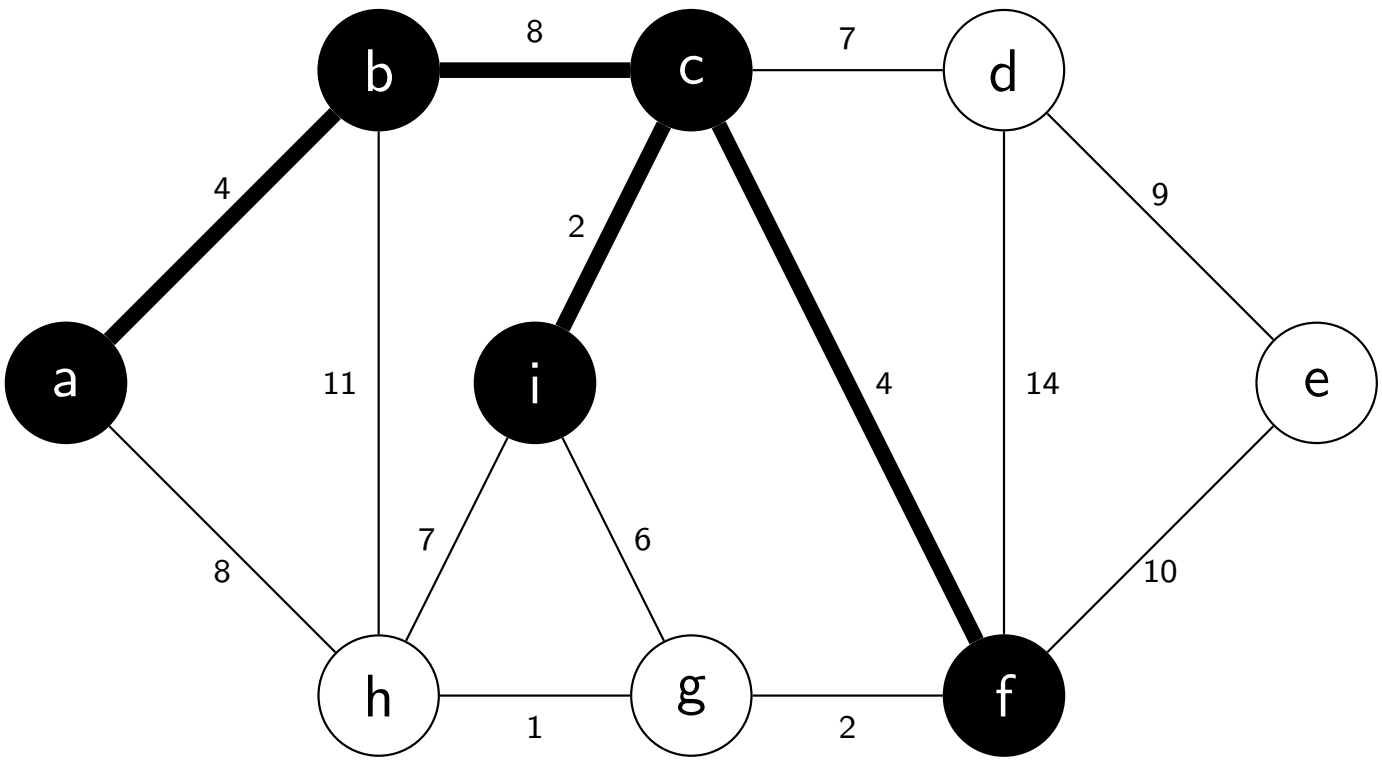
Example (Prim's Algorithm)

$S = \{a, b, c, i\} \quad V - S = \{d, e, f, g, h\}$



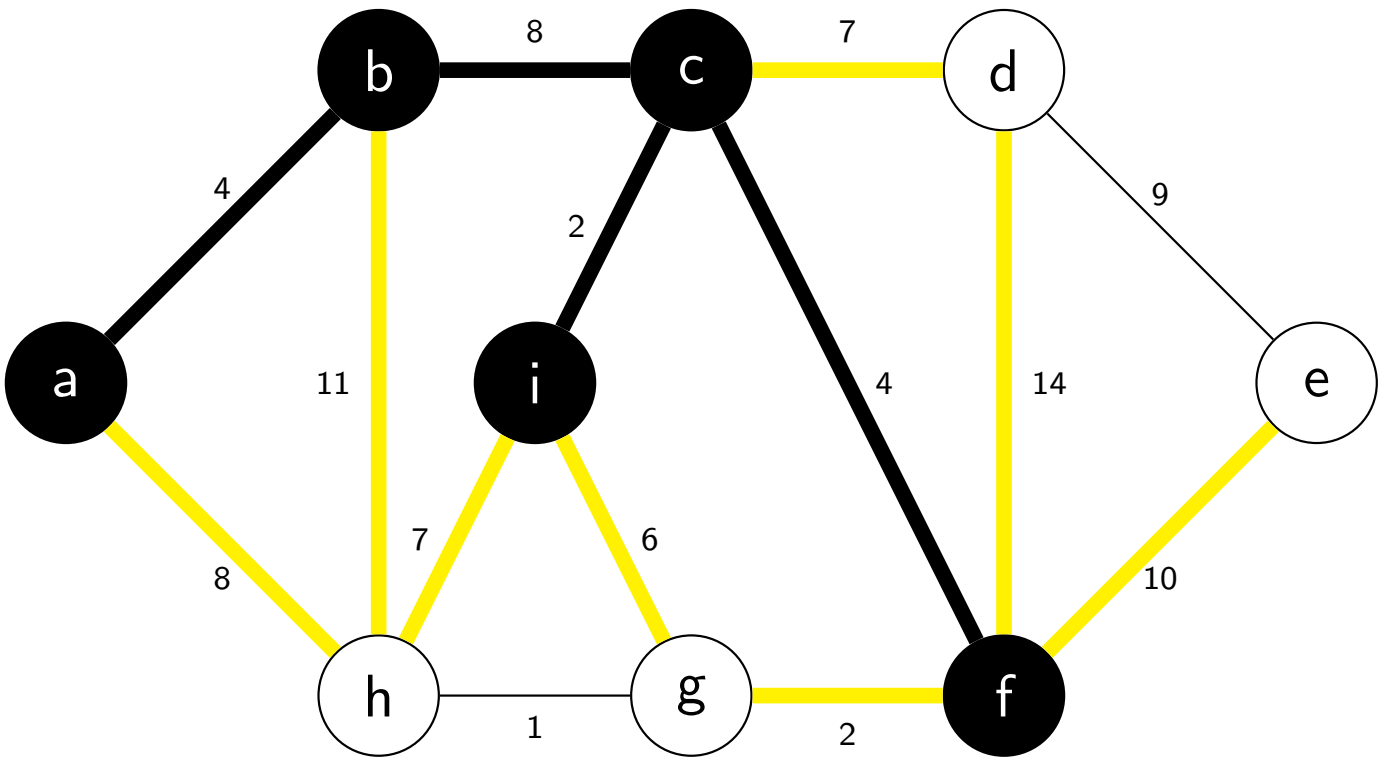
Example (Prim's Algorithm)

$S = \{a, b, c, i, f\} \quad V - S = \{d, e, g, h\}$



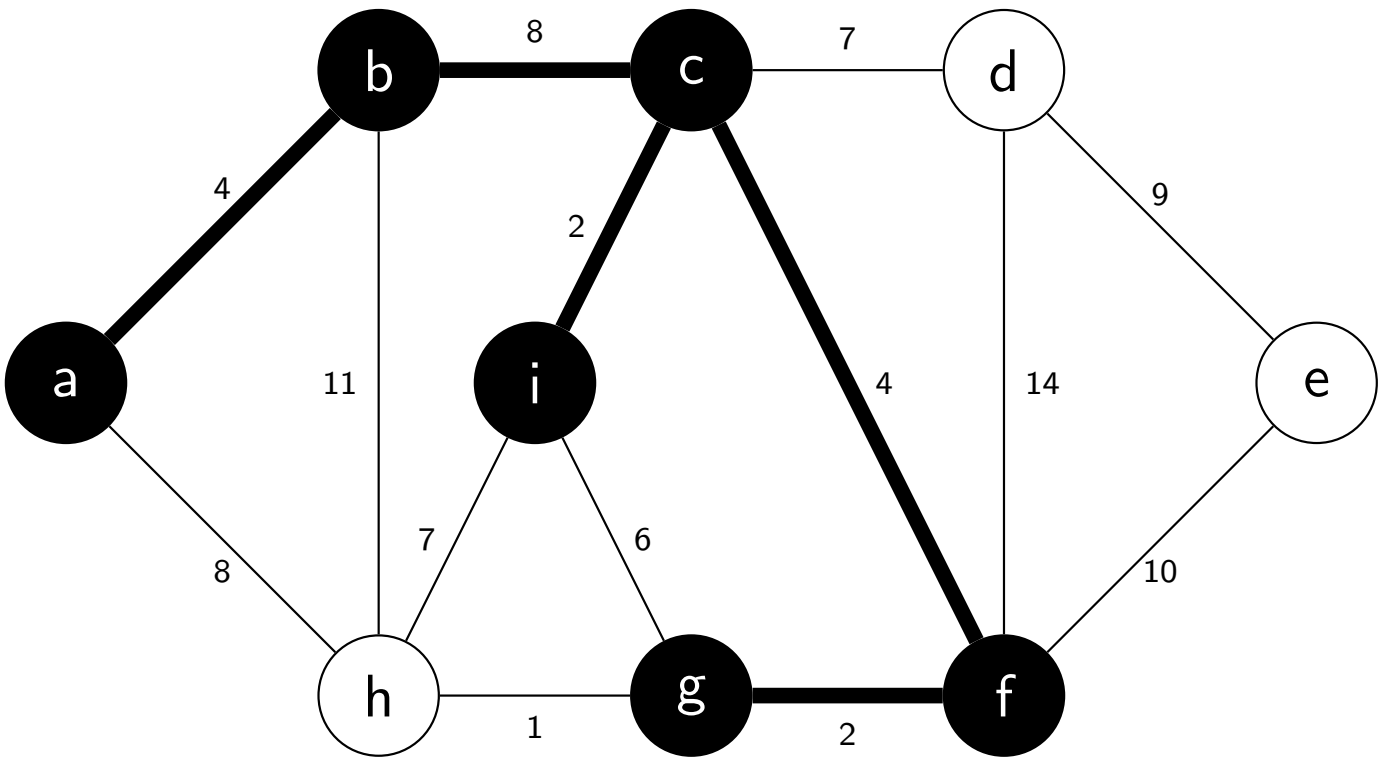
Example (Prim's Algorithm)

$S = \{a, b, c, i, f\} \quad V - S = \{d, e, g, h\}$



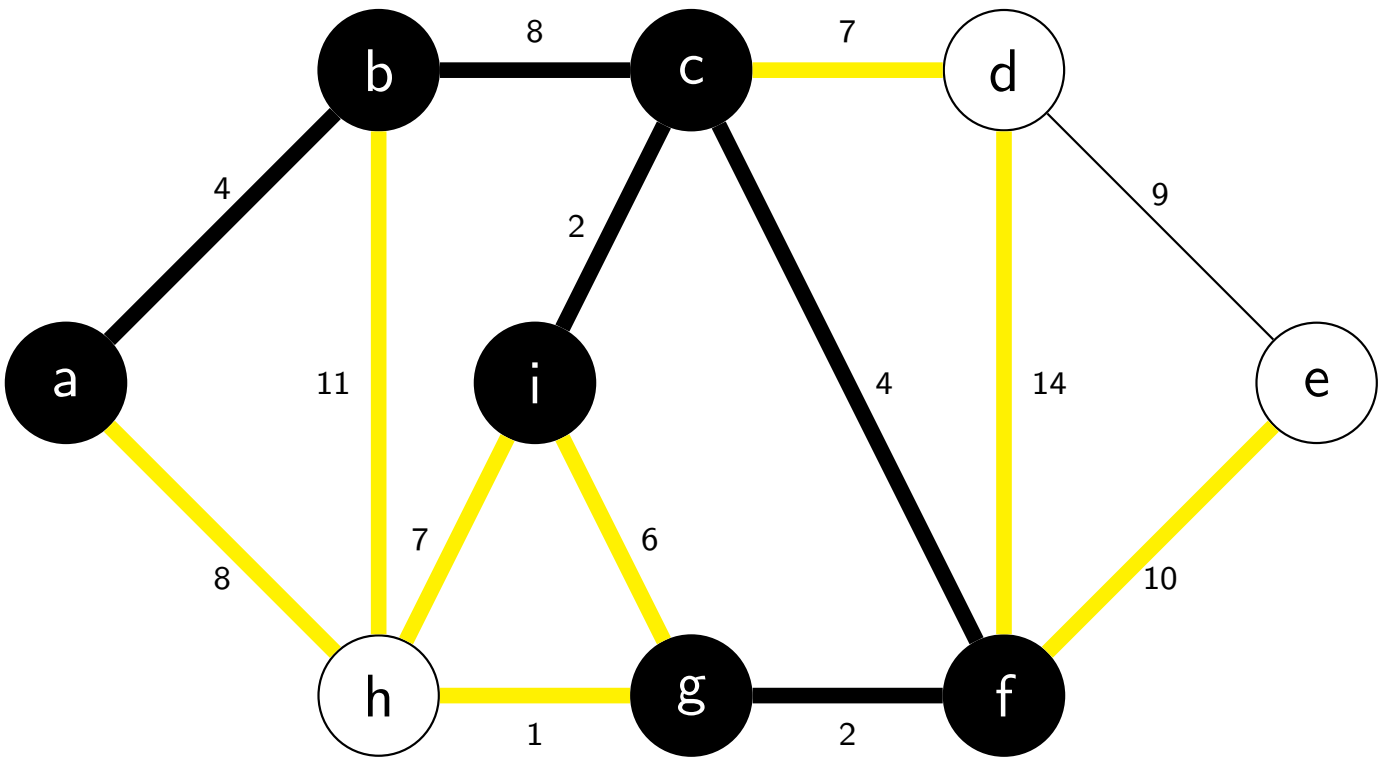
Example (Prim's Algorithm)

$S = \{a, b, c, i, f, g\} \quad V - S = \{d, e, h\}$



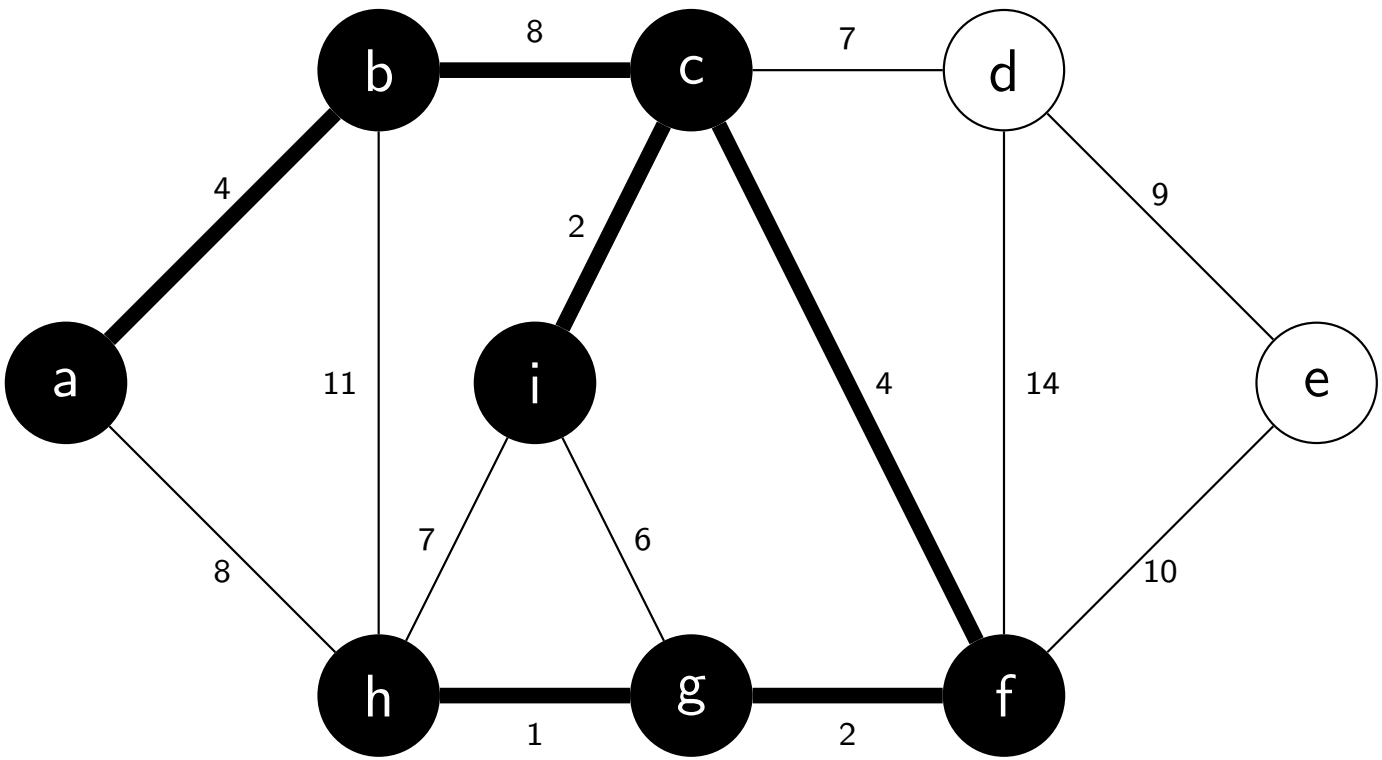
Example (Prim's Algorithm)

$S = \{a, b, c, i, f, g\} \quad V - S = \{d, e, h\}$



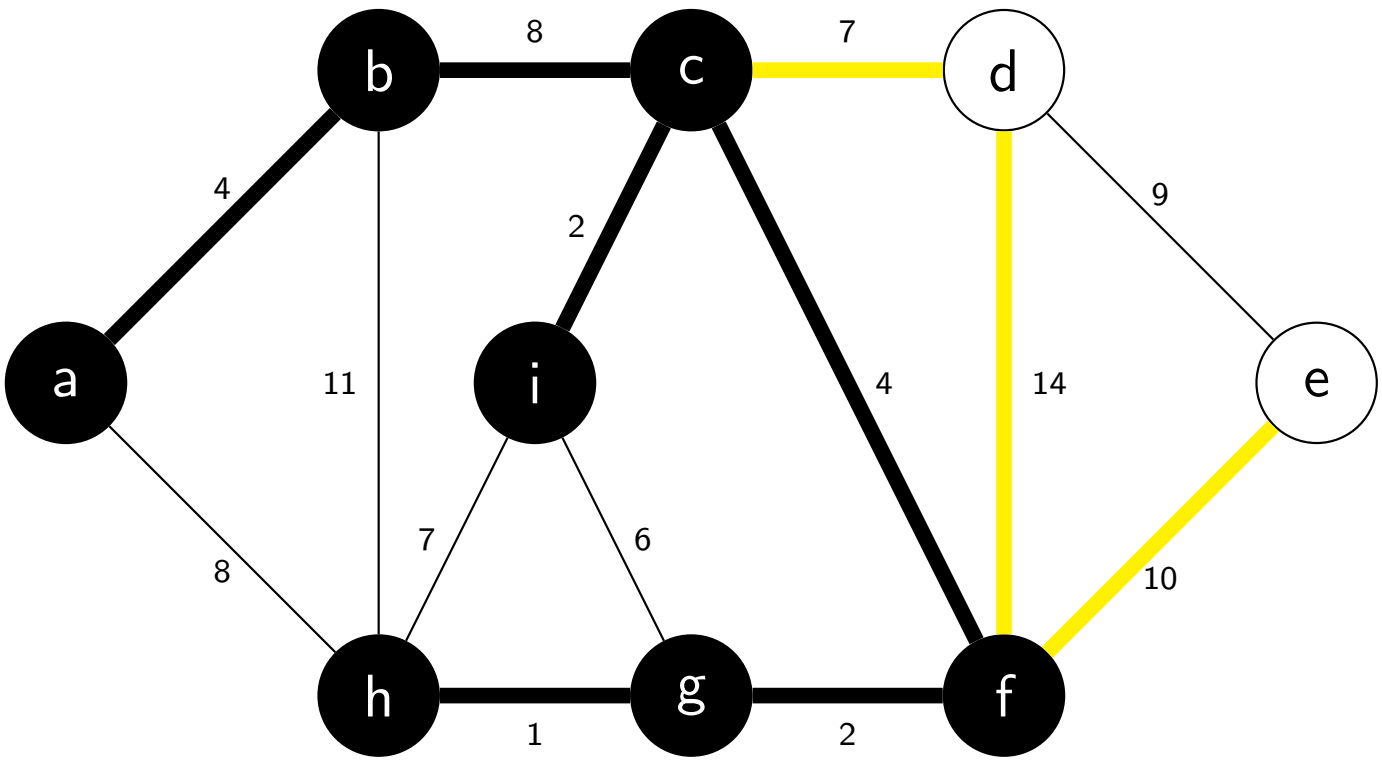
Example (Prim's Algorithm)

$S = \{a, b, c, i, f, g, h\} \quad V - S = \{d, e\}$



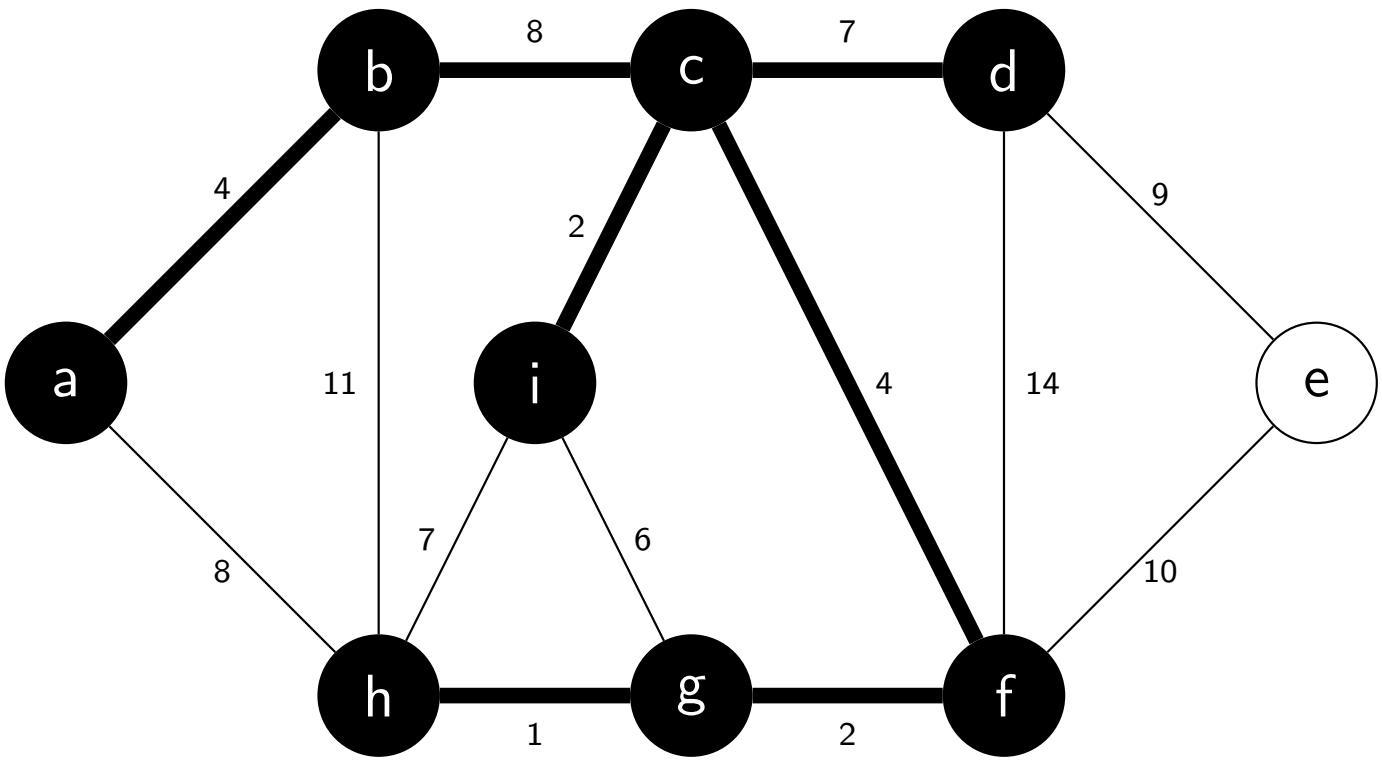
Example (Prim's Algorithm)

$S = \{a, b, c, i, f, g, h\}$ $V - S = \{d, e\}$



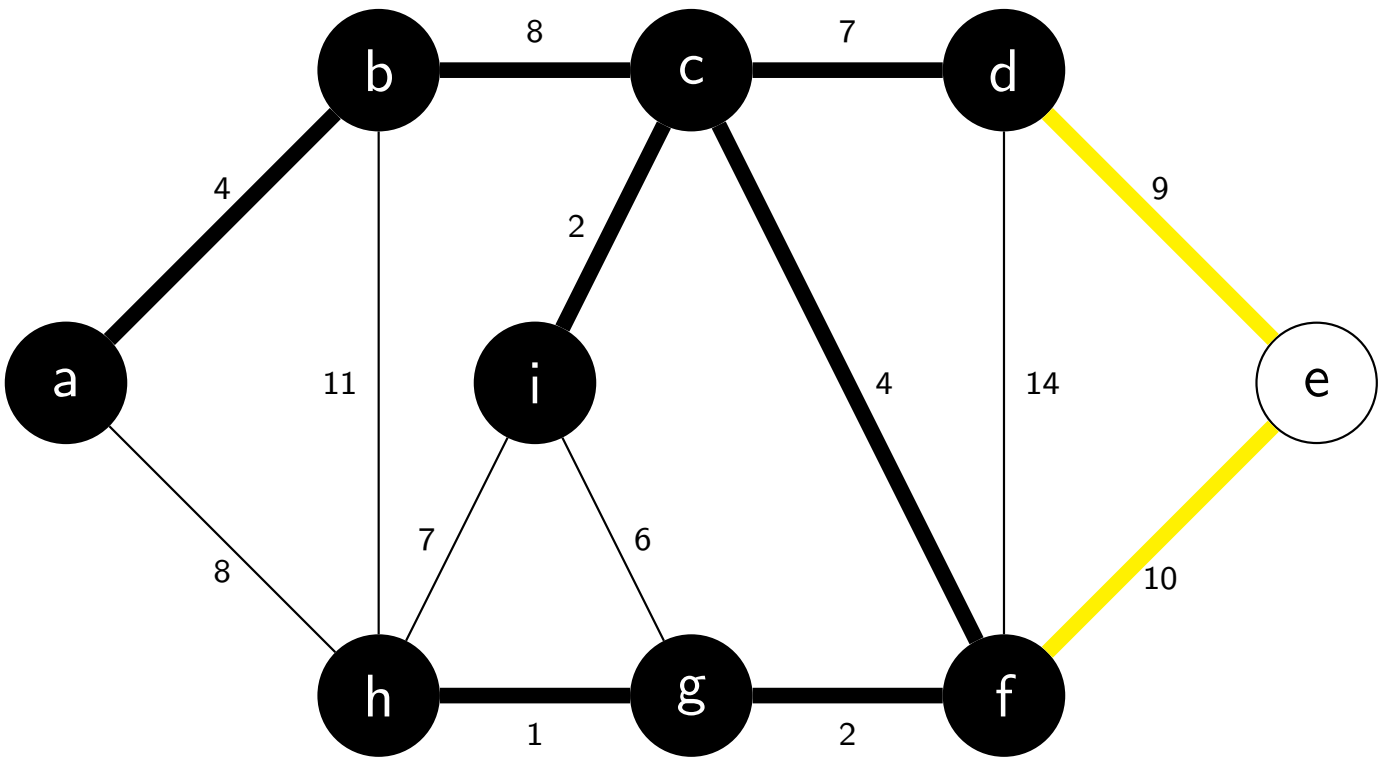
Example (Prim's Algorithm)

$S = \{a, b, c, i, f, g, h, d\} \quad V - S = \{e\}$



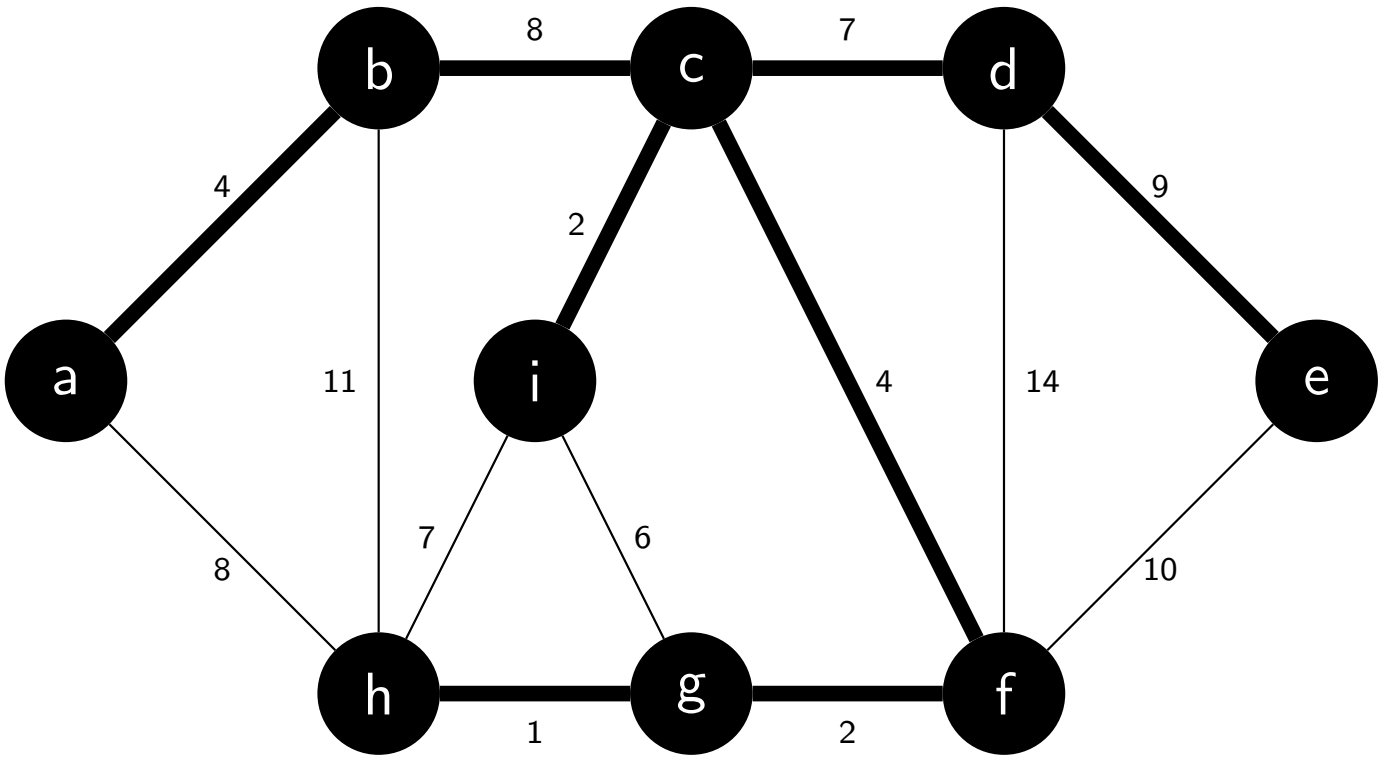
Example (Prim's Algorithm)

$S = \{a, b, c, i, f, g, h, d\} \quad V - S = \{e\}$



Example (Prim's Algorithm)

$S = \{a, b, c, i, f, g, h, d, e\} \quad V - S = \{\}$



High level implementation

Maintain sets S and $V - S$. Initially $S = \emptyset, T = \emptyset$. Then iterate the following:

1. Find $x \in S, y \in V - S$ such that $w(x, y) \leq w(u, v)$ for all $v \in S, u \in V - S$
2. Remove y from $V - S$ and add it to S. Also add edge (x, y) to T
3. Repeat until $V - S = \emptyset$

The challenge is to **efficiently** implement step 1 because brute force is $O(n^2)$ for each iteration.

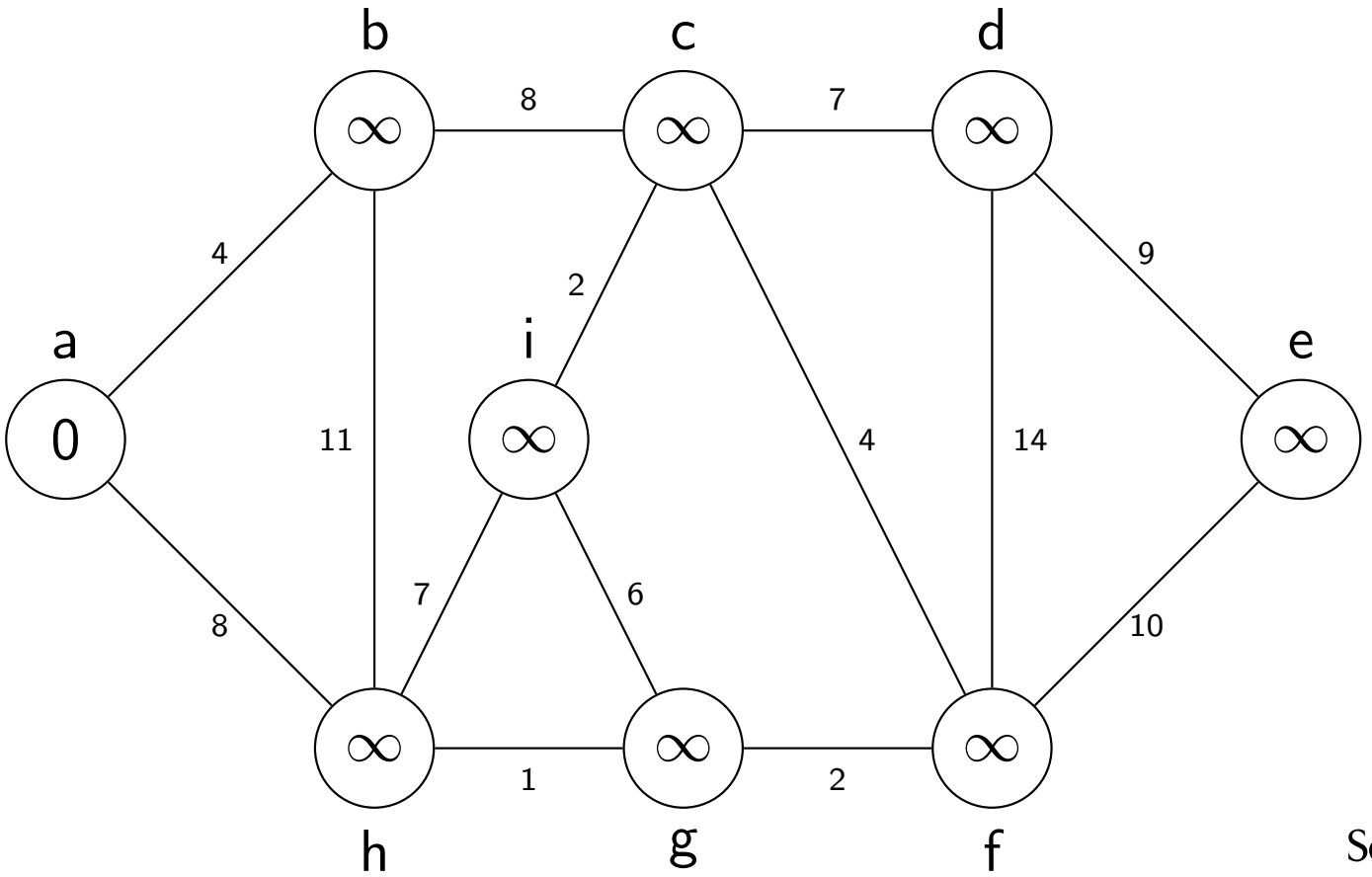
Efficient implementation of step 1

- The trick is to not recompute the minimum edge each time but reuse previous results.
- Every time we add a vertex u to S we update the minimum edge for each vertex $v \in V - S$.

Illutrative example

$$S = \{\} \quad V - S = \{a, b, c, d, e, f, g, h, i\}$$

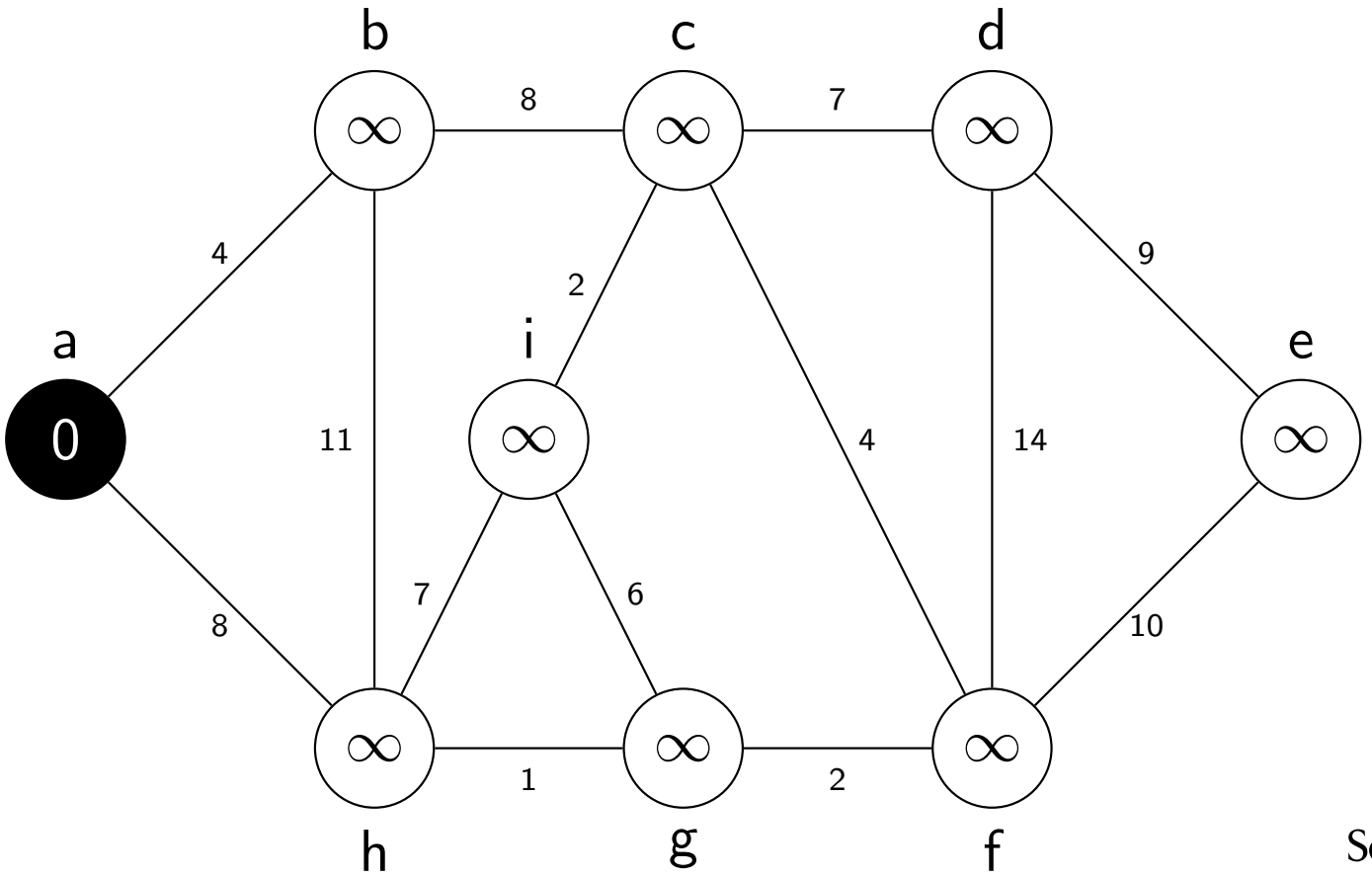
Initially



Illutrative example

$$S = \{a\} \quad V - S = \{b, c, d, e, f, g, h, i\}$$

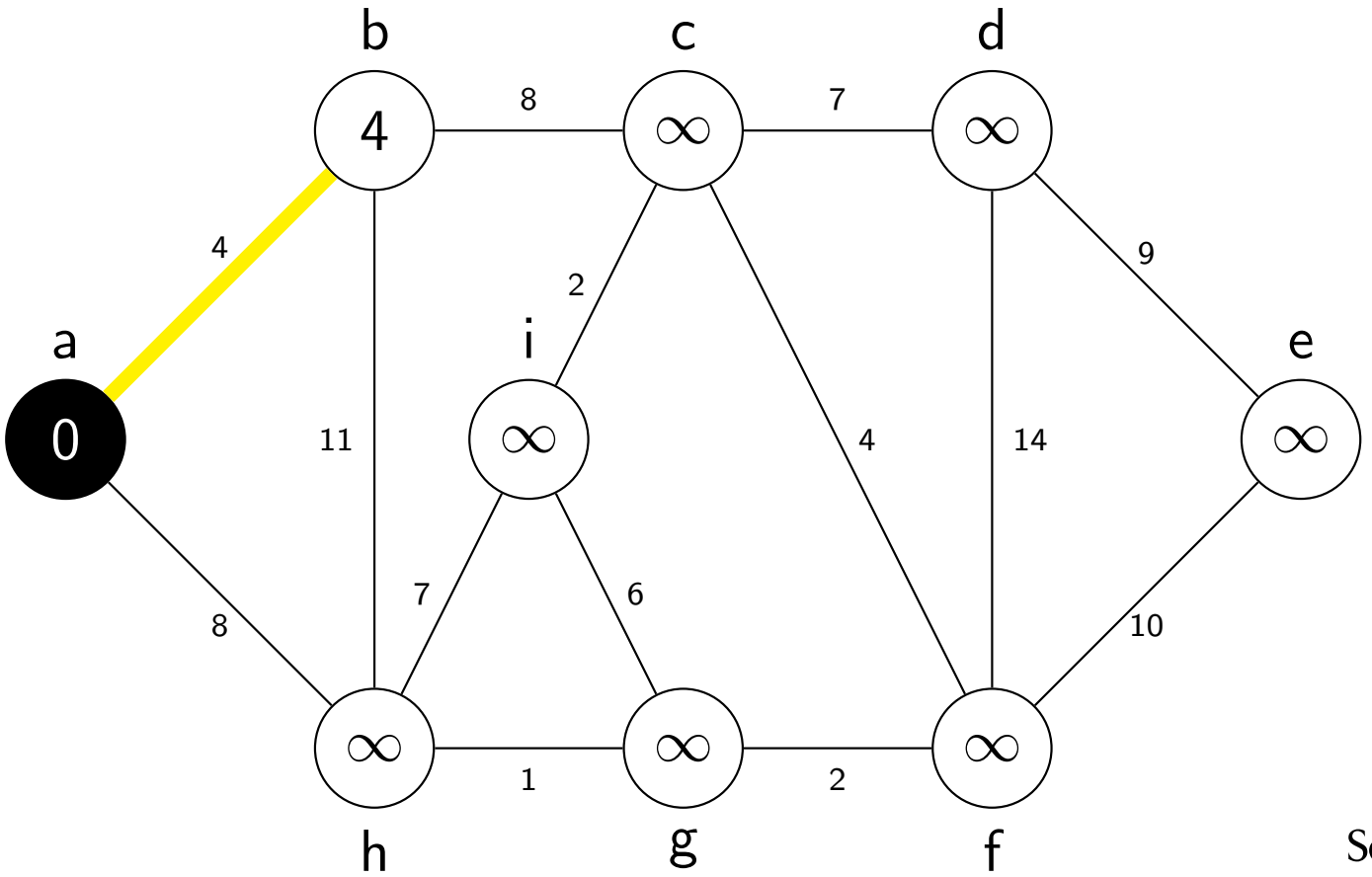
remove vertex with smallest key from $V - S$ and add to S



Illutrative example

$S = \{a\} \quad V - S = \{b, c, d, e, f, g, h, i\}$

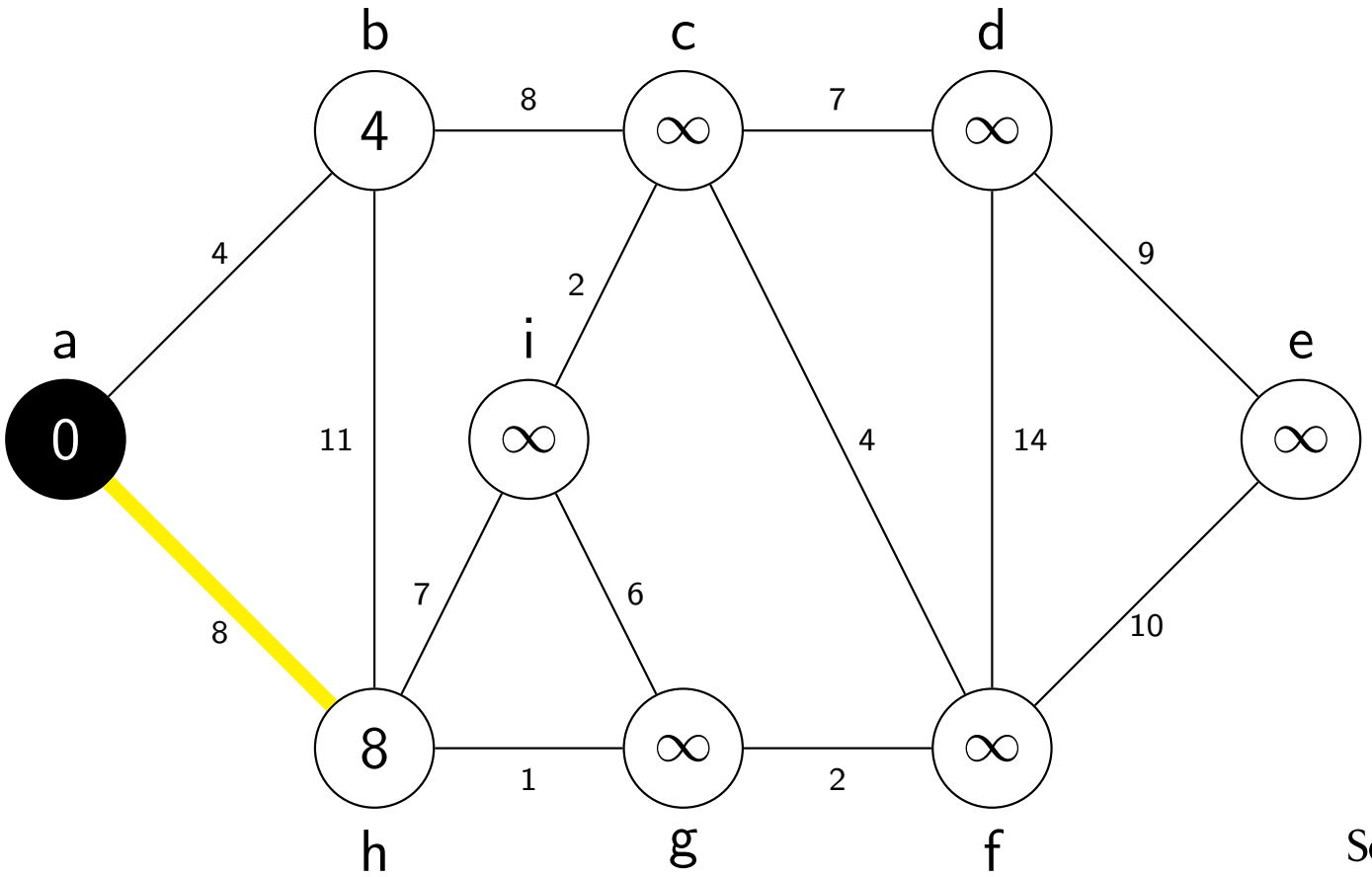
update the neighbors of a



Illutrative example

$S = \{a\} \quad V - S = \{b, c, d, e, f, g, h, i\}$

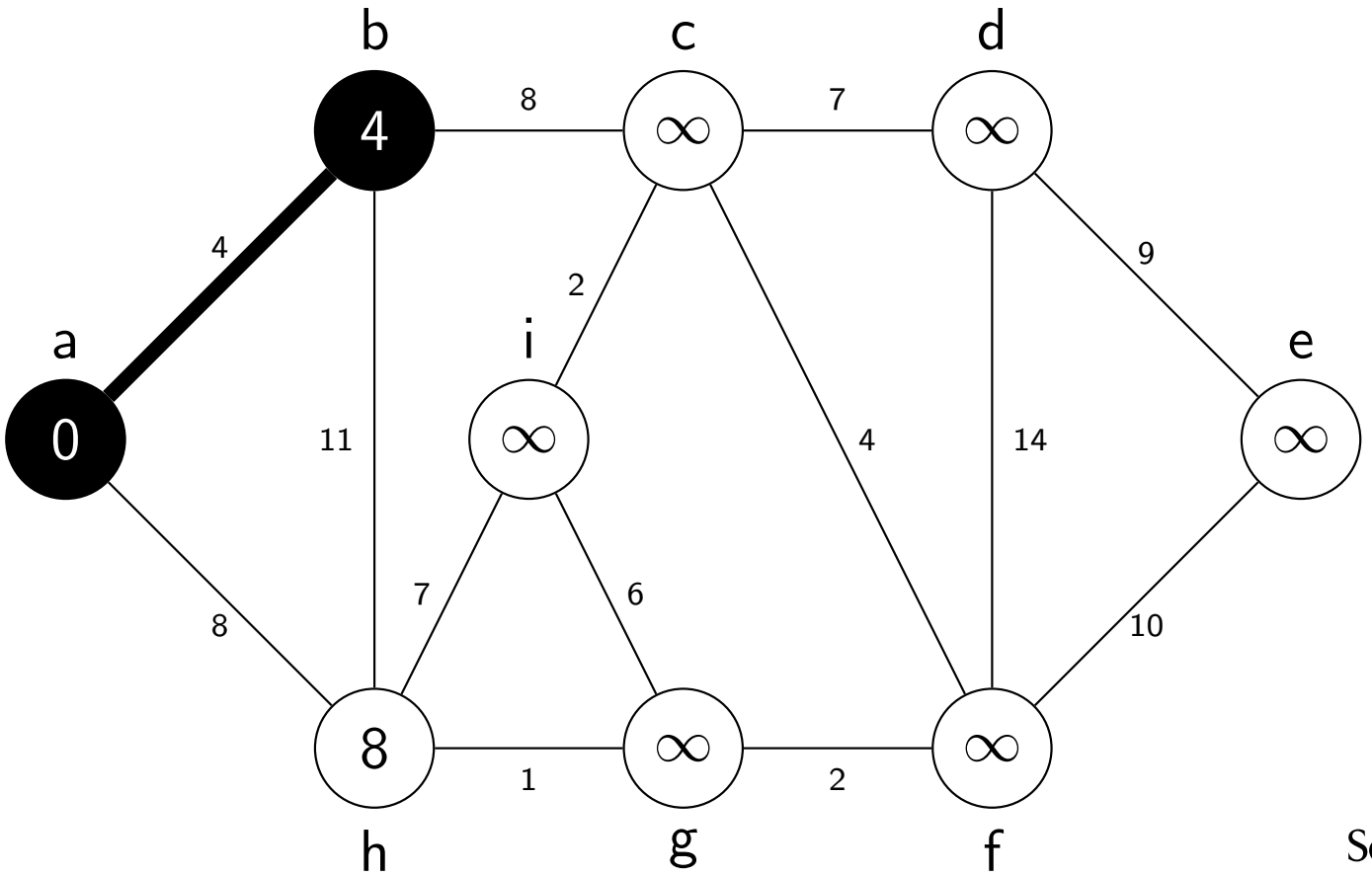
update the neighbors of a



Illutrative example

$$S = \{a, b\} \quad V - S = \{c, d, e, f, g, h, i\}$$

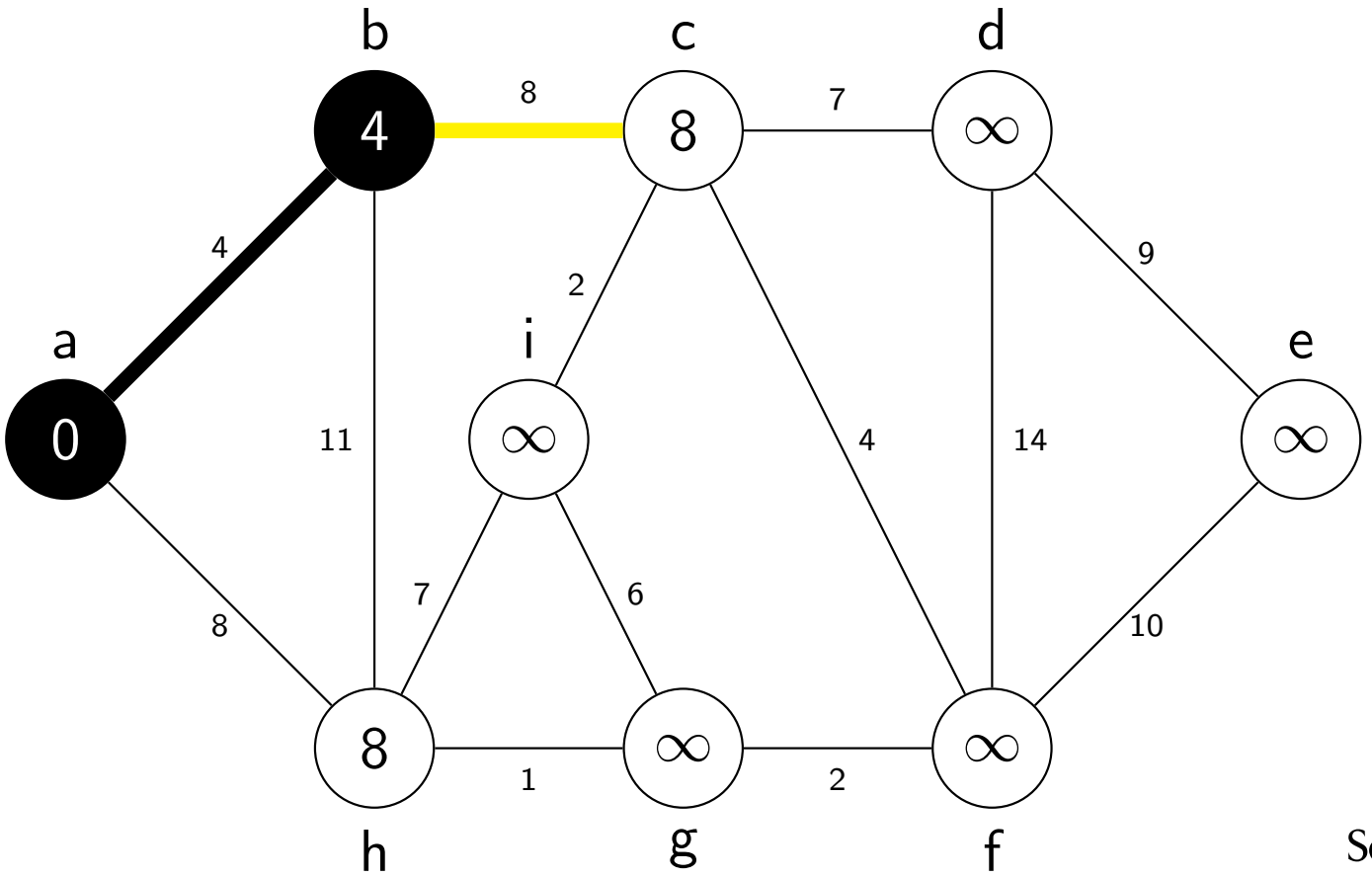
remove vertex with smallest key from $V - S$ and add to S



Illutrative example

$S = \{a, b\} \quad V - S = \{c, d, e, f, g, h, i\}$

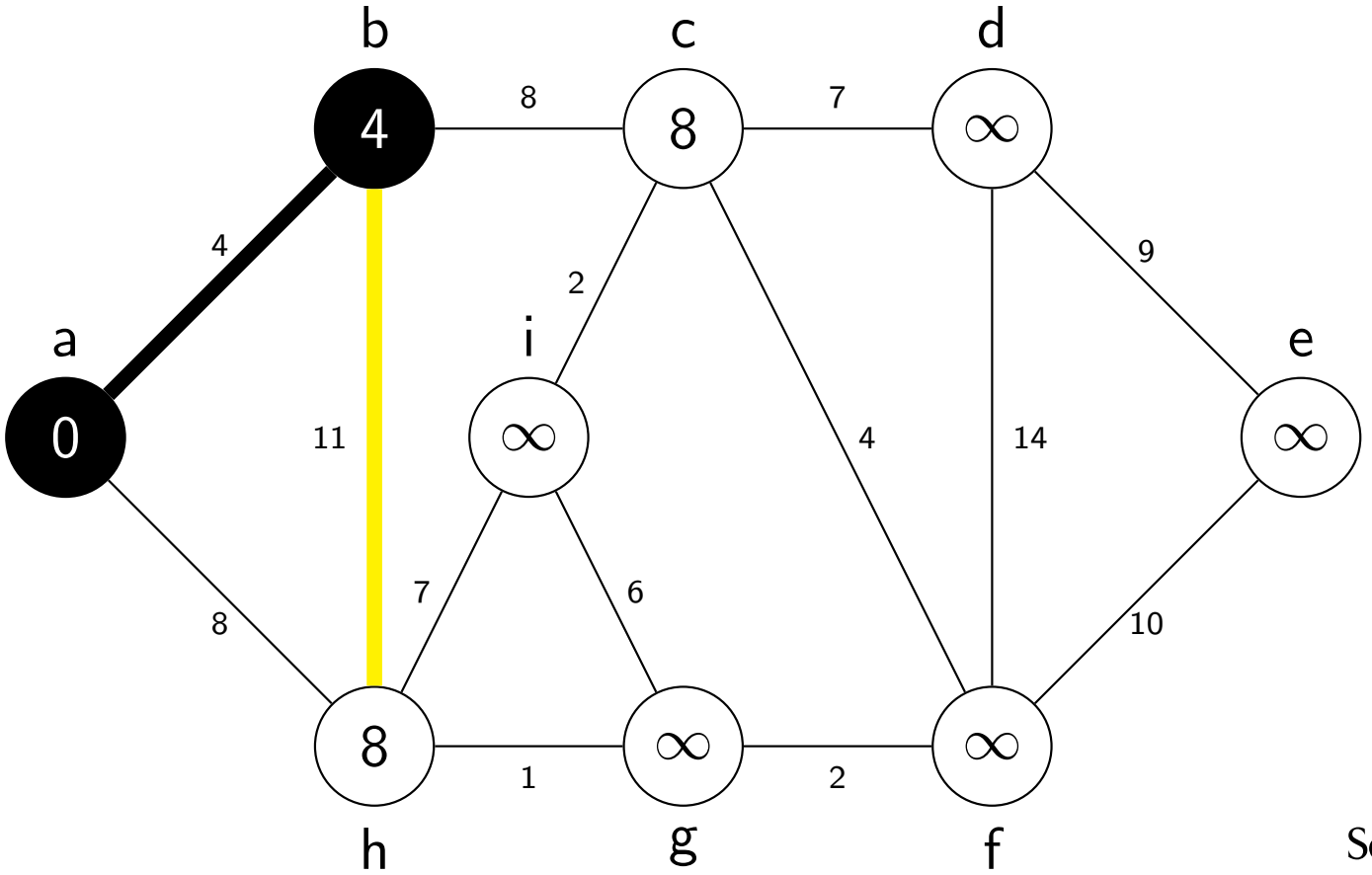
update the neighbors of b



Illutrative example

$S = \{a, b\} \quad V - S = \{c, d, e, f, g, h, i\}$

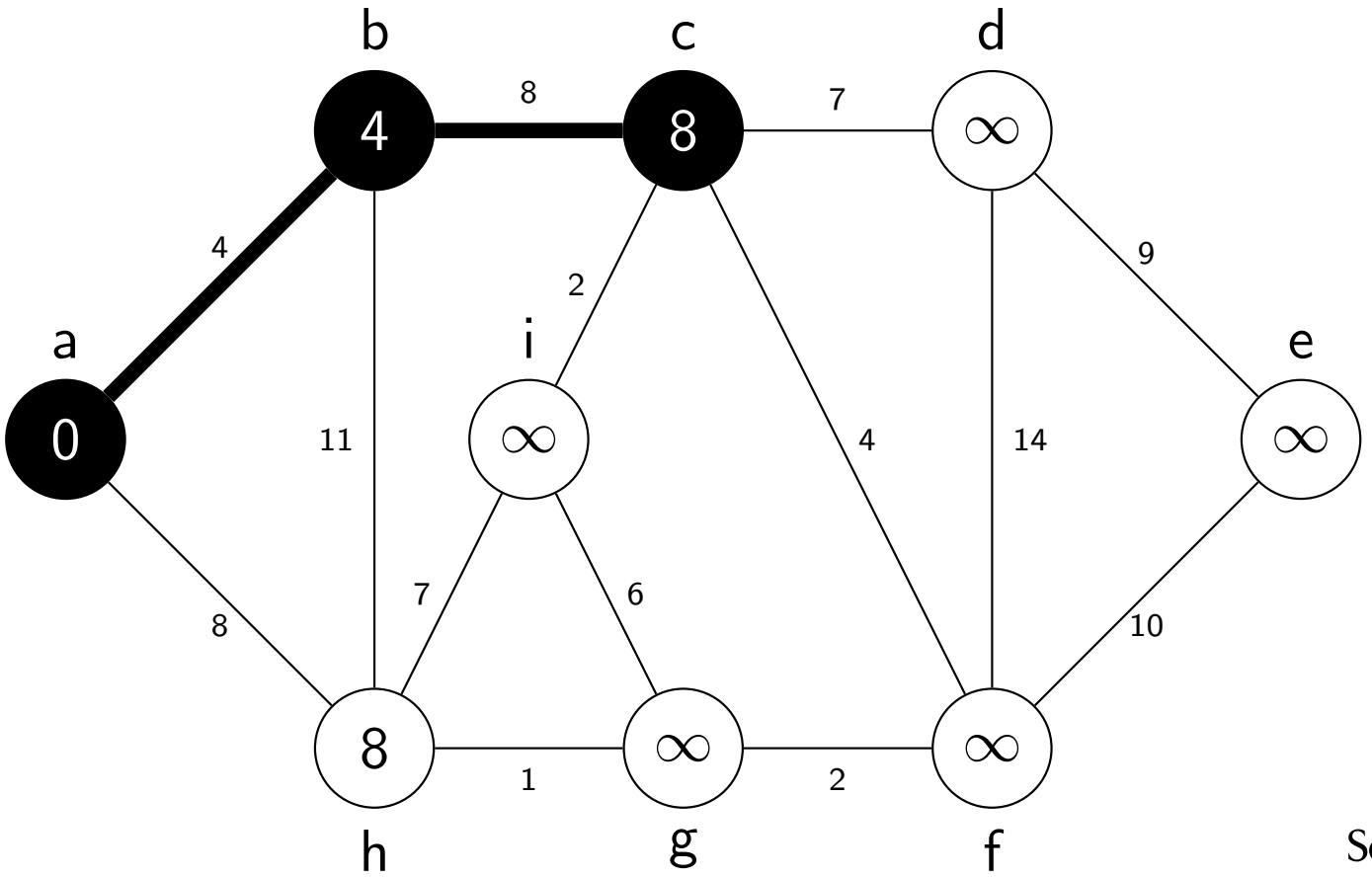
update the neighbors of b



Illutrative example

$$S = \{a, b, c\} \quad V - S = \{d, e, f, g, h, i\}$$

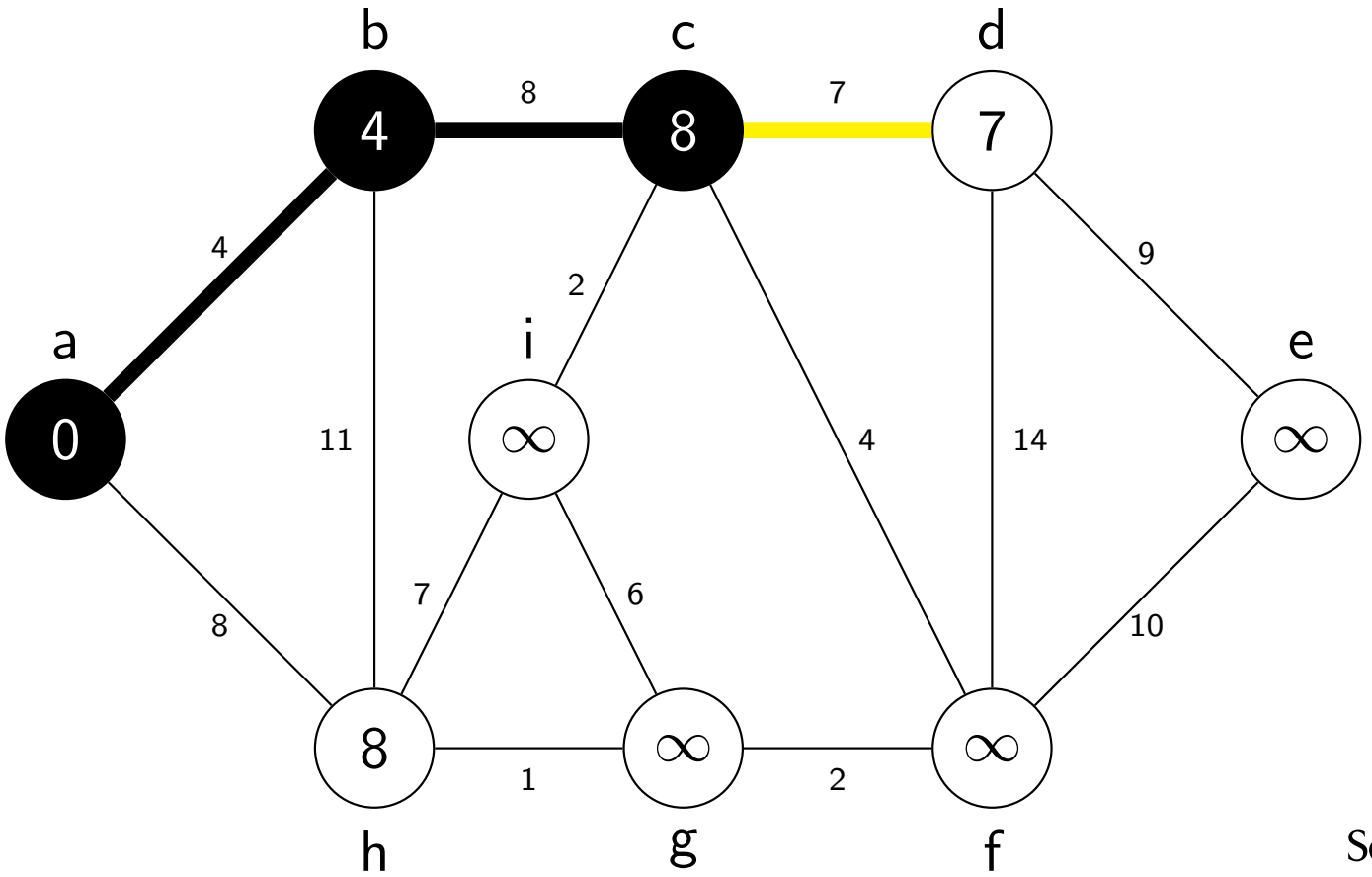
remove vertex with smallest key from $V - S$ and add to S



Illutrative example

$S = \{a, b, c\} \quad V - S = \{d, e, f, g, h, i\}$

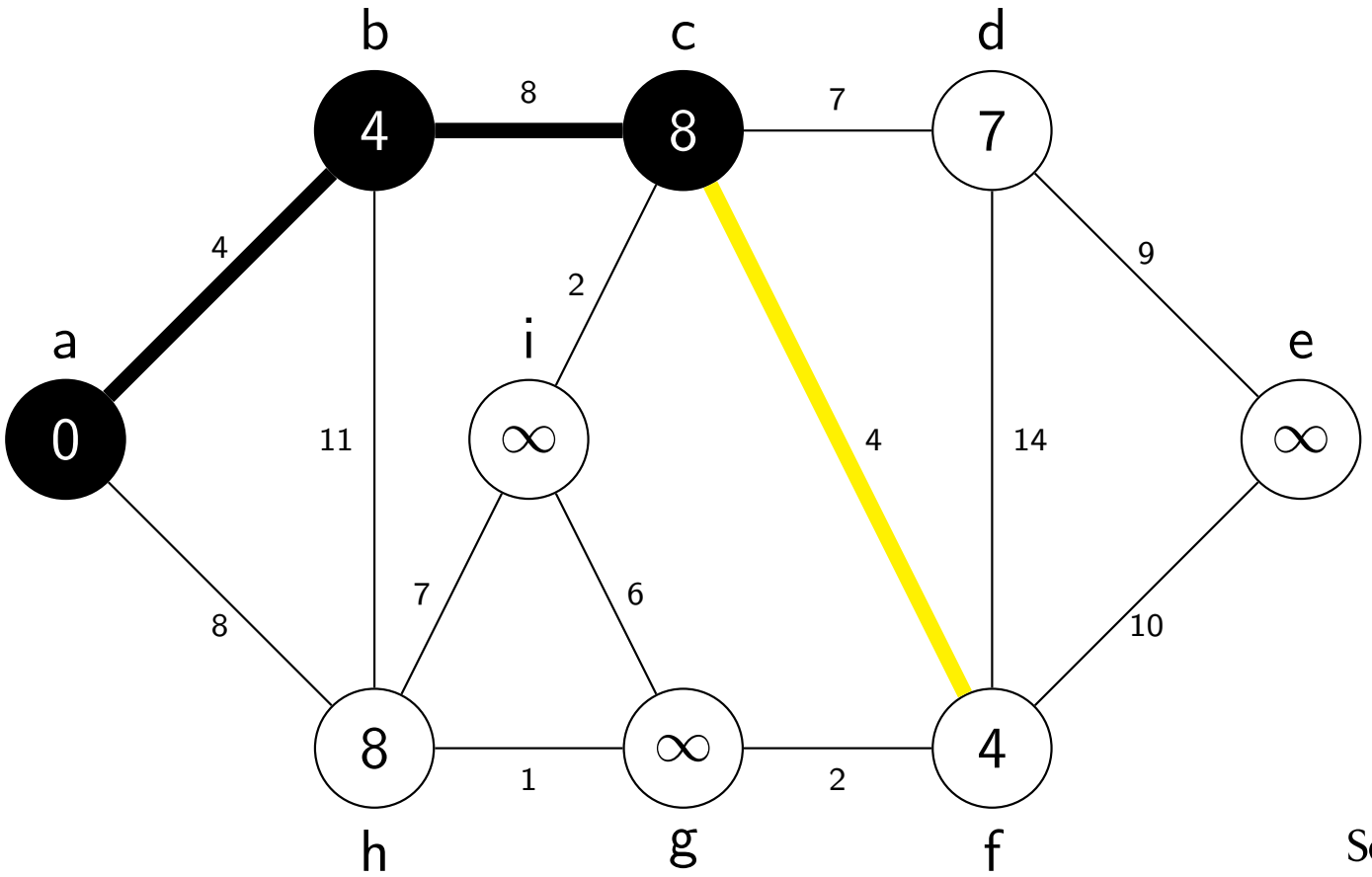
update the neighbors of c



Illutrative example

$S = \{a, b, c\} \quad V - S = \{d, e, f, g, h, i\}$

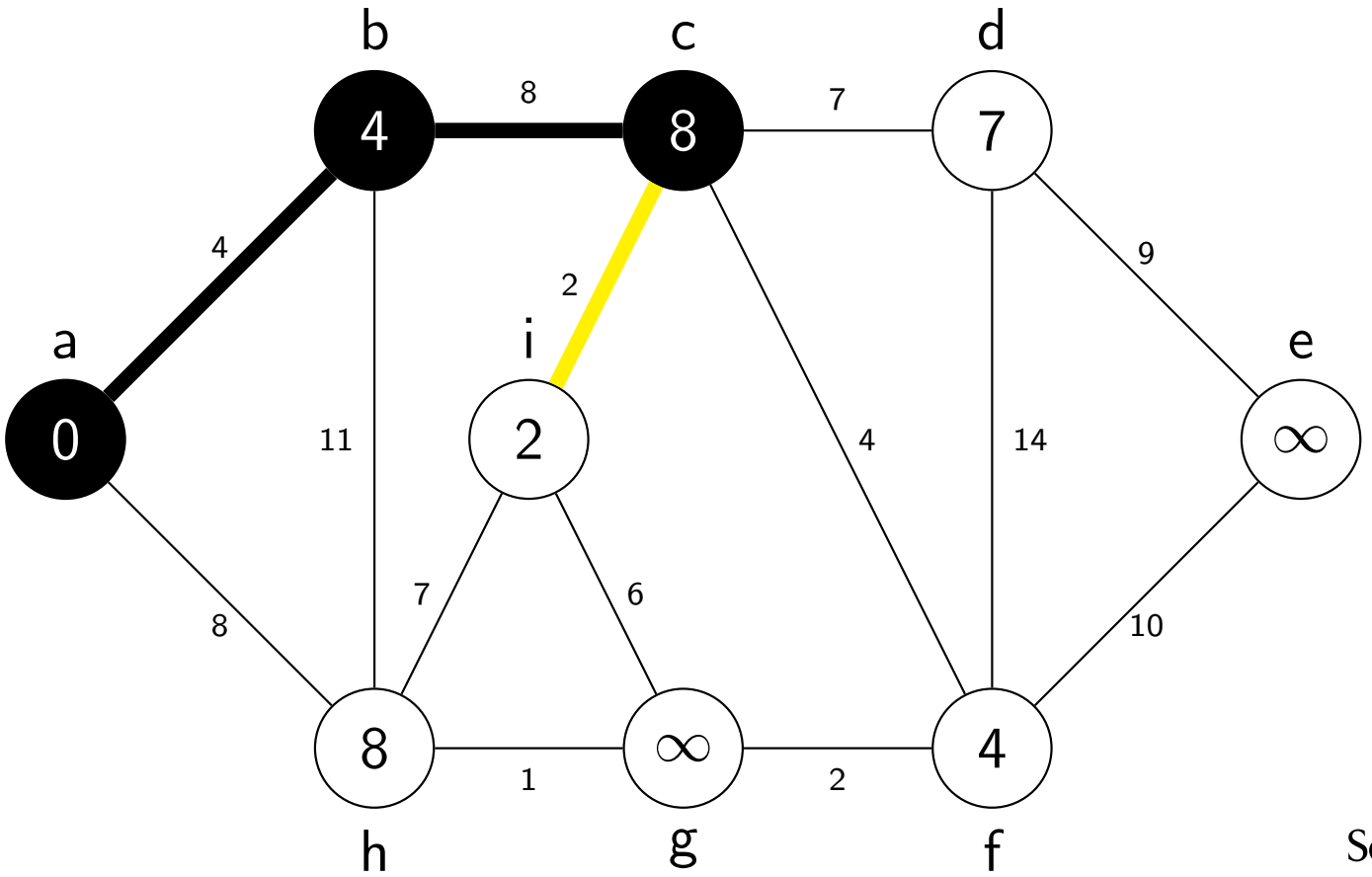
update the neighbors of c



Illutrative example

$S = \{a, b, c\} \quad V - S = \{d, e, f, g, h, i\}$

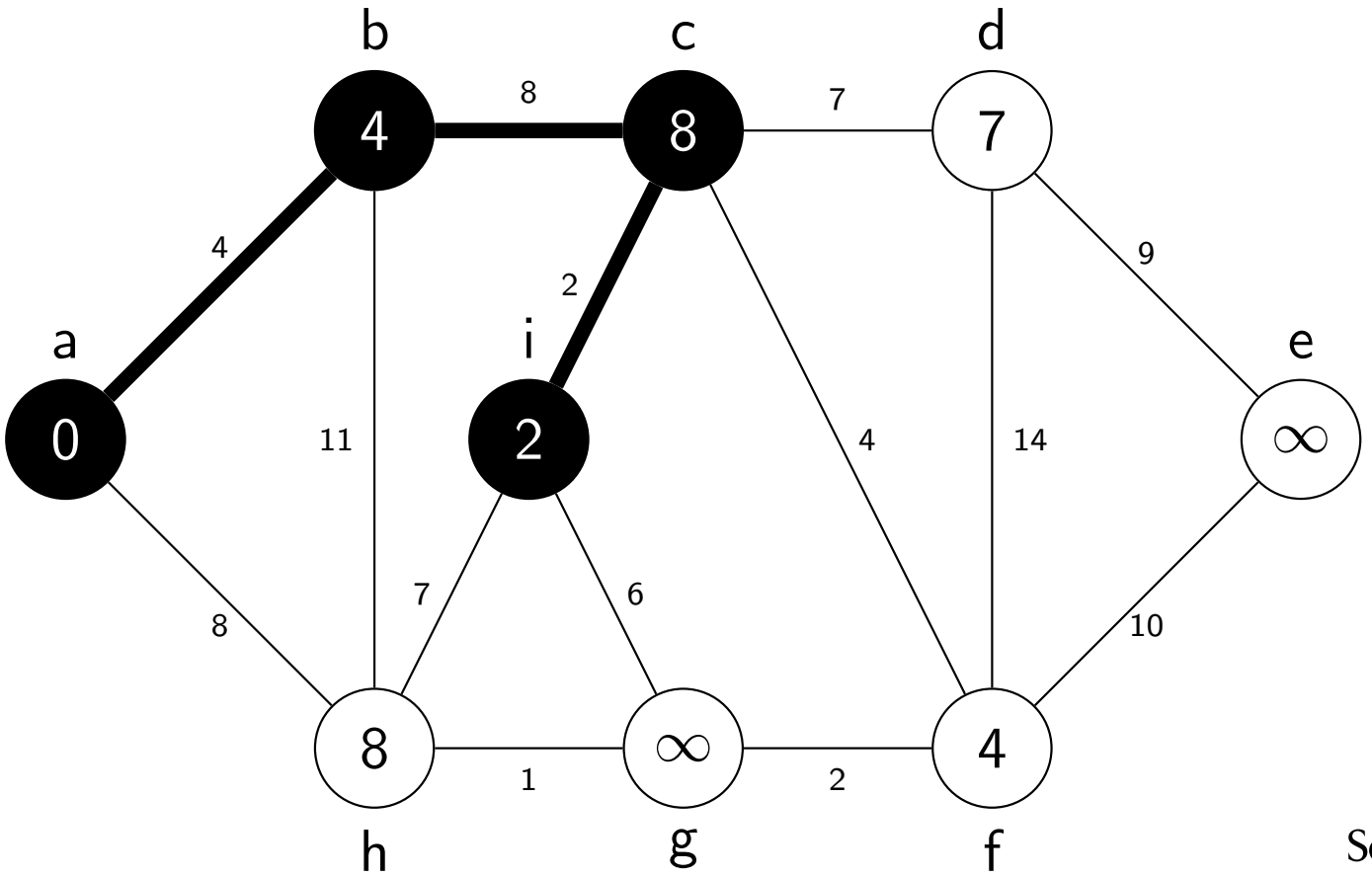
update the neighbors of c



Illutrative example

$$S = \{a, b, c, i\} \quad V - S = \{d, e, f, g, h\}$$

remove vertex with smallest key from $V - S$ and add to S



Pseudo code

Algorithm 2: Prim's algorithm

```
1  foreach v ∈ V do
2  |   v.key ← ∞
3  |   v.p ← NULL
4  s.key ← 0
5  Q ← V
6  T ← ∅
7  while Q ≠ ∅ do
8  |   u ← DELETE-MIN(Q)
9  |   T ← T ∪ {(u, u.p)}
10 |   foreach v ∈ adj[u] ∧ v ∉ S do
11 |   |   if w(u, v) < v.key then
12 |   |   |   v.key ← w(u, v)
13 |   |   |   v.p ← u
```

Notes

- The choice of the "source" vertex on line 4 is arbitrary, any vertex can be chosen as source.
- For clarity an if statement on line 9 was omitted.
- It checks if $u.p$ is null in which case no edge is added to T.
- This occurs only for the first iteration because the source s has no predecessor.
- The usual implementation has S as a hash table for $\Theta(1)$ $v \notin S$ operation and Q as a priority queue for $\Theta(\log n)$ DELETE-MIN.

Why does it work?

- Both Kruskal's and Prim's algorithms are special cases of a general method to obtain a minimum spanning tree.
- The basic idea is based on the following:
- Maintain a set of edges A.
- Before every iteration A is a subset of some minimum spanning tree.
- At each step we add an edge to A such that A is **still** a subset of some MST.
- An edge having that property is called **safe** for A.

`MST(G)`

$A \leftarrow \emptyset$

while *A is not MST* **do**

  find edge (u, v) safe for A

  $A \leftarrow A \cup \{(u, v)\}$

return A

- The above algorithm looks easy.
- But how do we find a safe edge?

Some Definitions

- Let $G = (V, E)$ be a graph with some real-valued weight function $w : E \to R$.
- A **cut** $(S, V - S)$ of the graph G is a **partition** of V.
- We say a cut $(S, V - S)$ **respects** $A \subseteq E$ if no edge in A crosses the cut.
- An edge is said to be a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.

This is why it works

- The reason why both algorithms work is the following theorem

Theorem

Let A be a set of edges included in some minimum spanning tree, $(S, V - S)$ a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

Correctness of Prim's Algorithm

- At the beginning of every iteration (except the first) Prim's algorithm starts by removing u where $u.key$ is minimum. This means that $(u.p, u)$ is a light edge for the cut $(Q, V - Q)$
- Therefore Prim's algorithm is correct.

Correctness of Kruskal's Algorithm

- Prior to every iteration of Kruskal's algorithm we have
  1. A forest (a collection of trees) $G_A = (V, A)$. (initially is A is empty)
  2. Select an edge $(u, v) \in E - A$ with
     1. $w(u, v)$ is minimal.
     2. $u \in T_u$ and $v \notin T_u$ where T_u is a tree in G_A that contains u.
  3. From the above we have that: $(T_u, V - T_u)$ is a cut that respects A and (u, v) is a light edge crossing that cut.
- From the theorem we know that (u, v) is a safe edge for A.

Complexity

- **Kruskal**: we use the union find operations we learned in the beginning of the semester. Let $\mid V \mid = n$ and $\mid E \mid = m$.
- Recall that we use an array id to specify the parent of node in the (logical) tree that represents a given group.
- e.g. node $id[i]$ is the parent of i. Initially each node is its own parent: $id[i] = i$ thus the first **for** loop is $\Theta(n)$.
- Sorting is $\Theta(m \log m)$.
- In our implementation, Union is $\Theta(1)$ and FIND-SET is $\Theta(\log n)$. Therefore the foreach loop is $\Theta(m \log n)$.
- Adding all the contributions we get: $\Theta(n + m \log m + m \log n)$.