CSE 548 / AMS 542: Analysis of Algorithms

Prerequisites Review 6
(Greedy Algorithms: MST, SSSP, …)

Rezaul Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2023

An Activity-Selection Problem

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}

Suppose:

- You are given a set $S = \{a_1, a_2, \dots, a_n\}$ of n proposed *activities* that wish to use a resource, such as a lecture hall, which can serve only one activity at a time.

An Activity-Selection Problem

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Suppose:

- You are given a set $S = \{a_1, a_2, \dots, a_n\}$ of n proposed *activities* that wish to use a resource, such as a lecture hall, which can serve only one activity at a time.
- Each activity a_i has a *start time* s_i and *finish time* f_i, where $0 \le s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$.
- Activities a_i and a_j are *compatible* if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$.

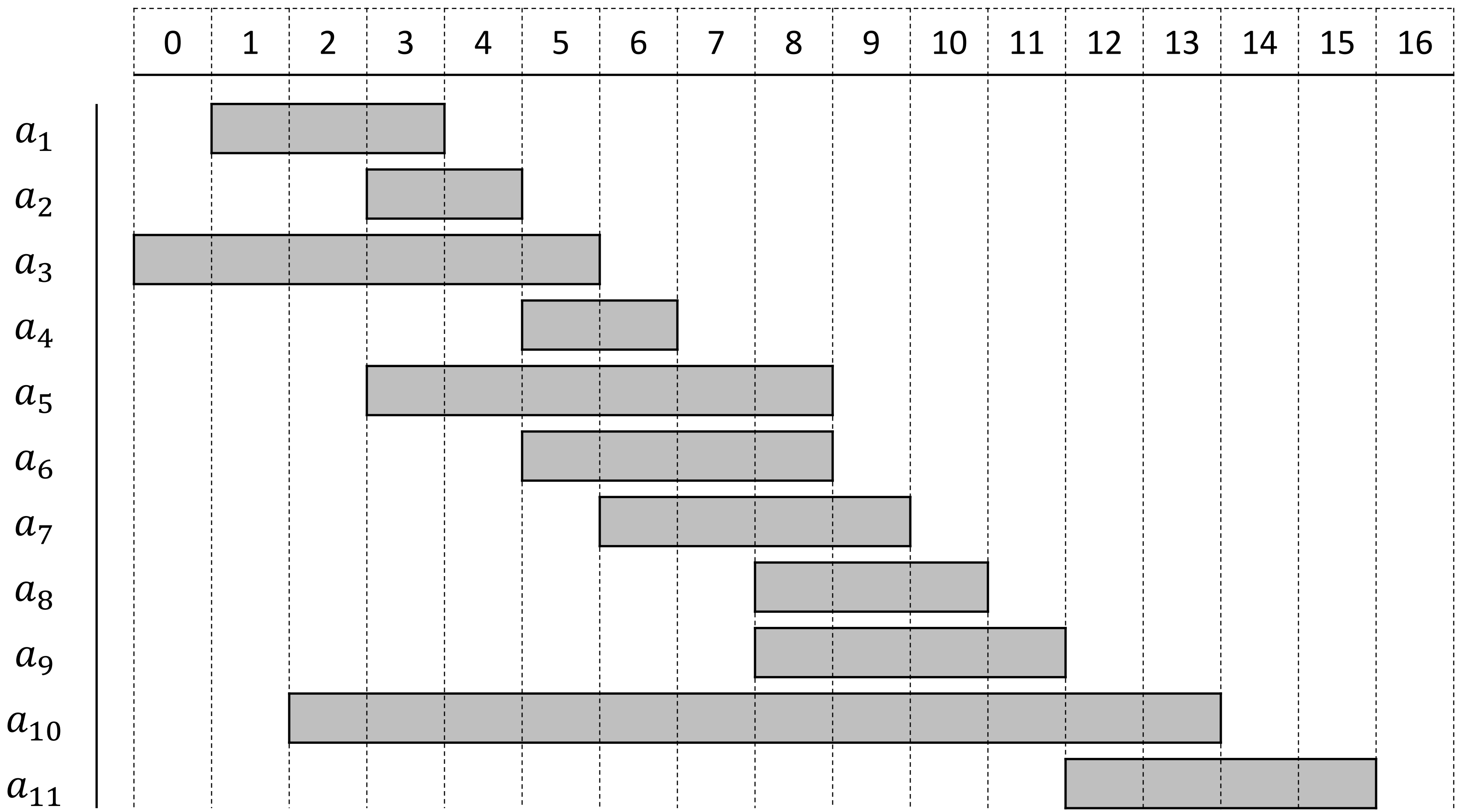
Goal: Select a maximum-size subset of mutually compatible activities.

Assume that the activities are sorted in monotonically non-decreasing order of finish time: $f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n$.

An Activity-Selection Problem

An example set S of activities

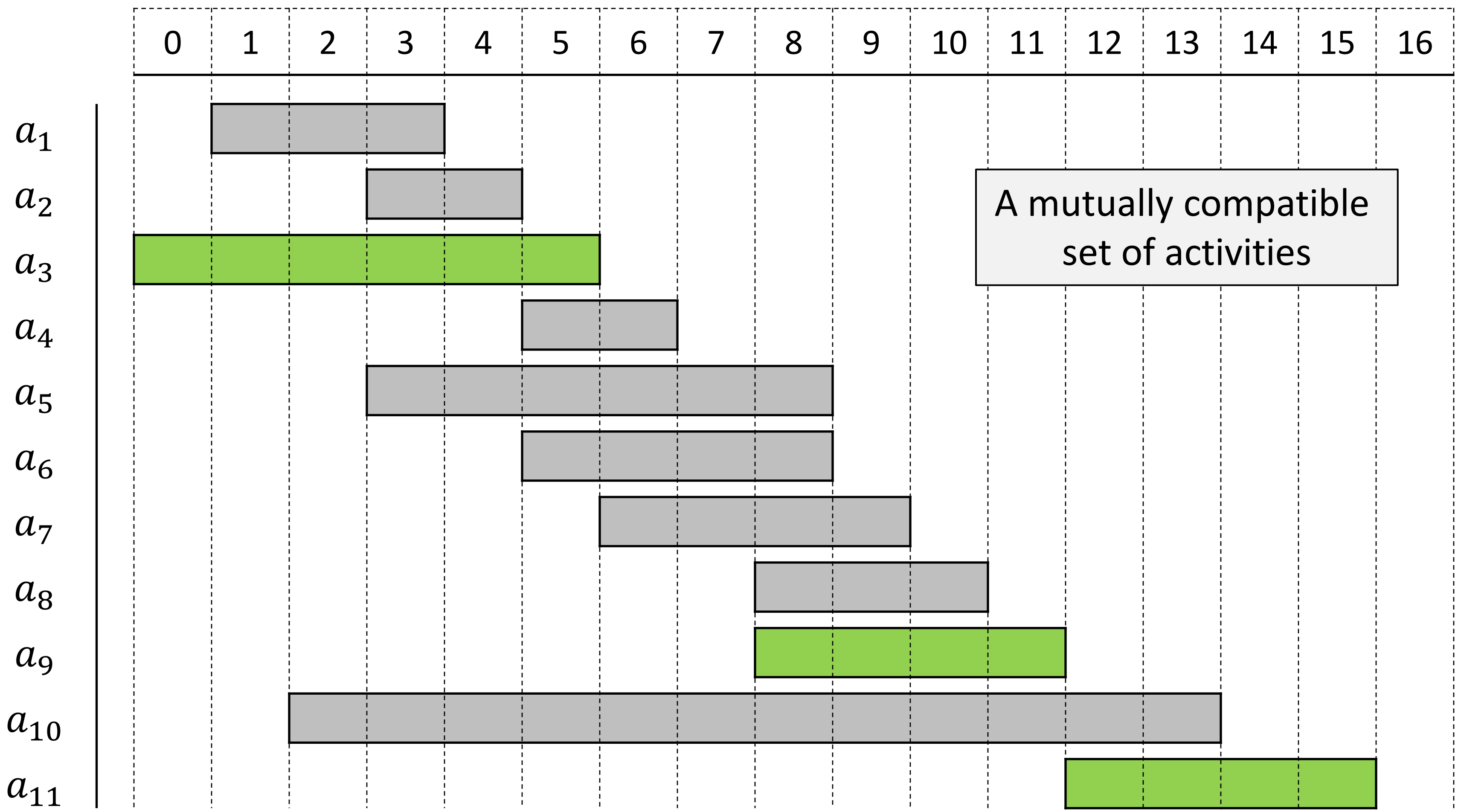
a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



An Activity-Selection Problem

An example set S of activities

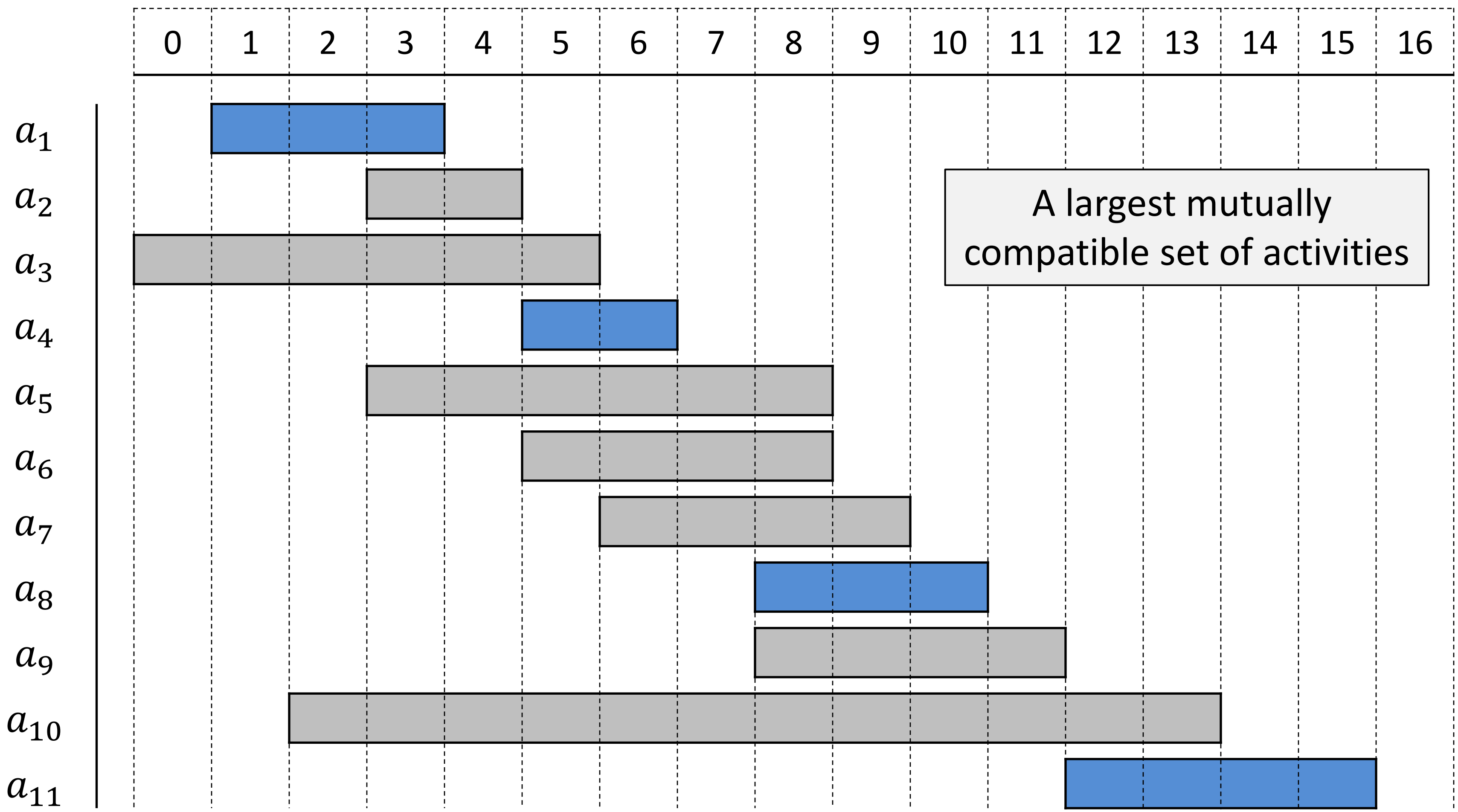
a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



An Activity-Selection Problem

An example set S of activities

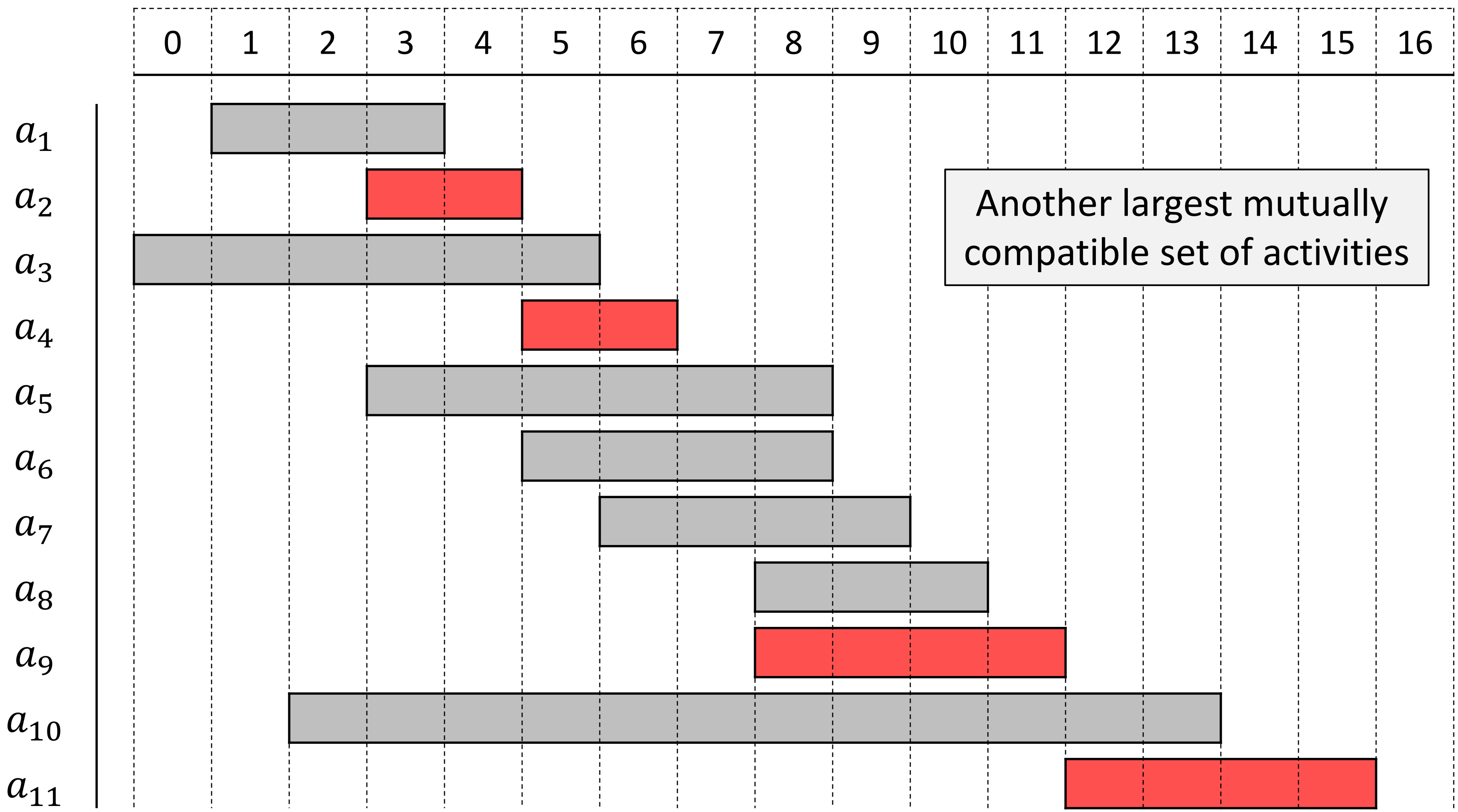
a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



An Activity-Selection Problem

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Activity-Selection: Greedy Choice

Let S_k = the set of activities in S that start after activity a_k finishes.

THEOREM: Consider any nonempty subproblem S_k and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

PROOF: Let A_k = a maximum-size subset of mutually compatible activities in S_k.

Let a_j be the activity in A_k with the earliest finish time.

If $a_j = a_m$, we are done.

If $a_j \neq a_m$, let $A'_k = A_k - \{a_j\} \cup \{a_m\}$.

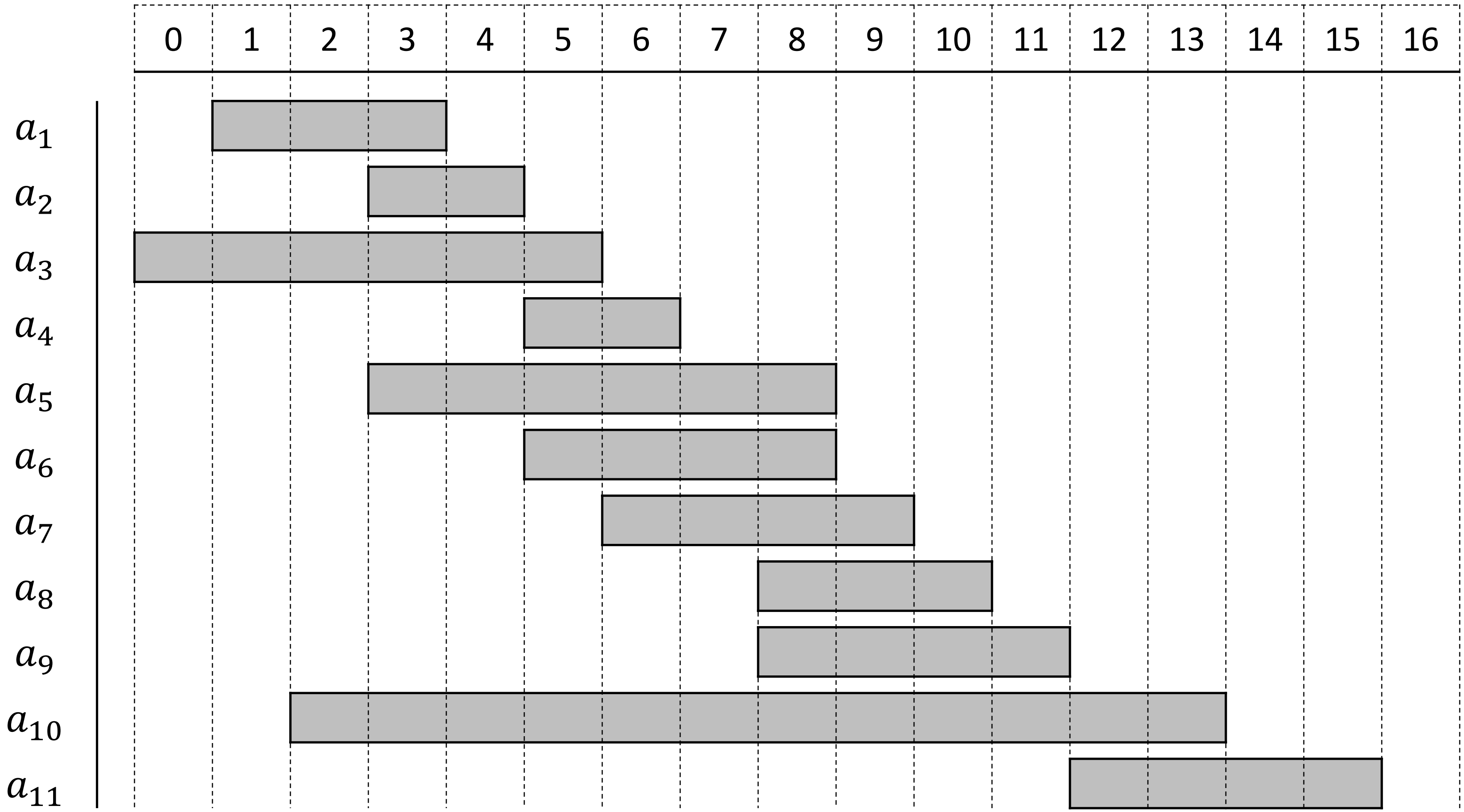
The activities in A'_k are disjoint because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$.

Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m.

Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

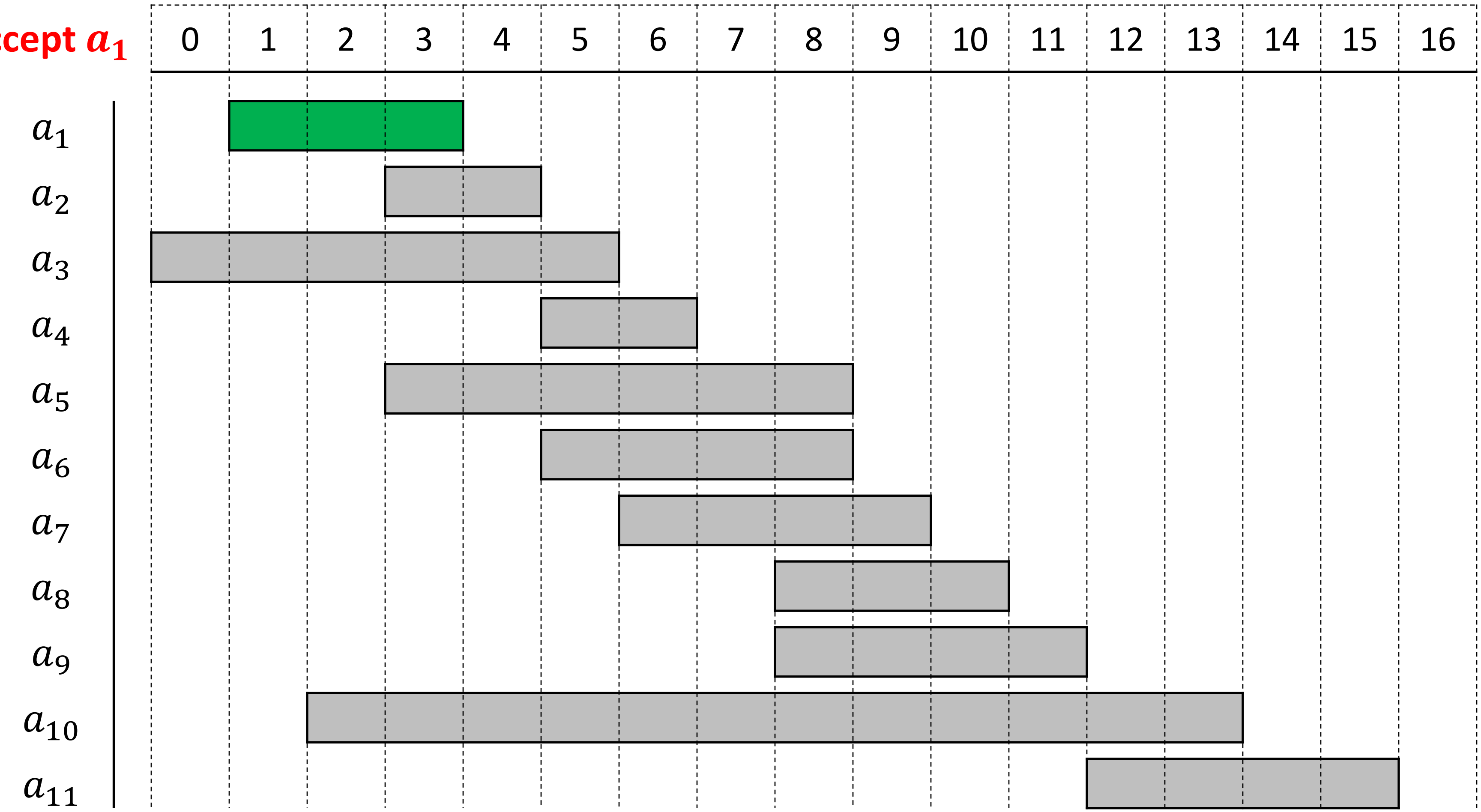


Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

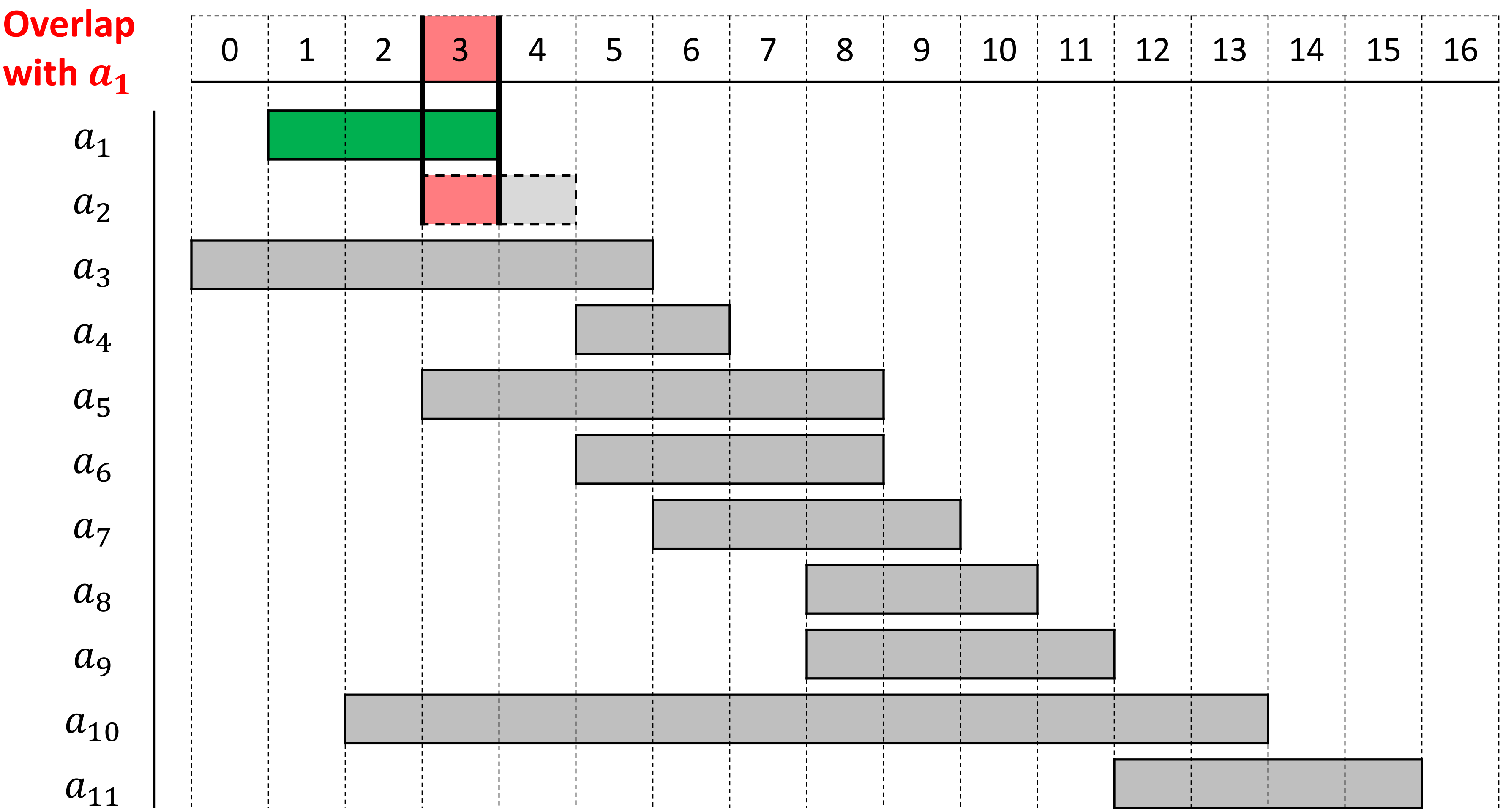
Accept a_1



Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

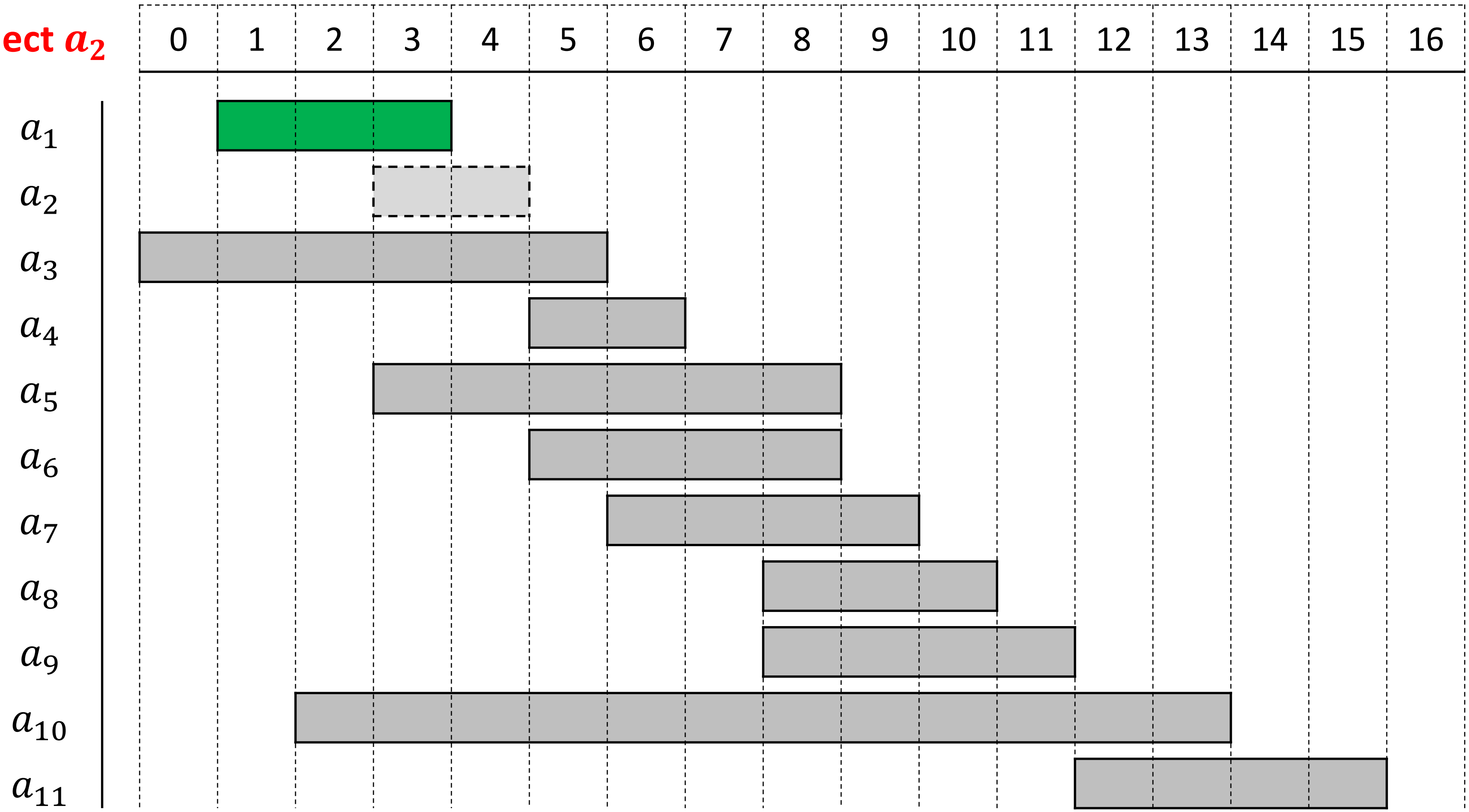


Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

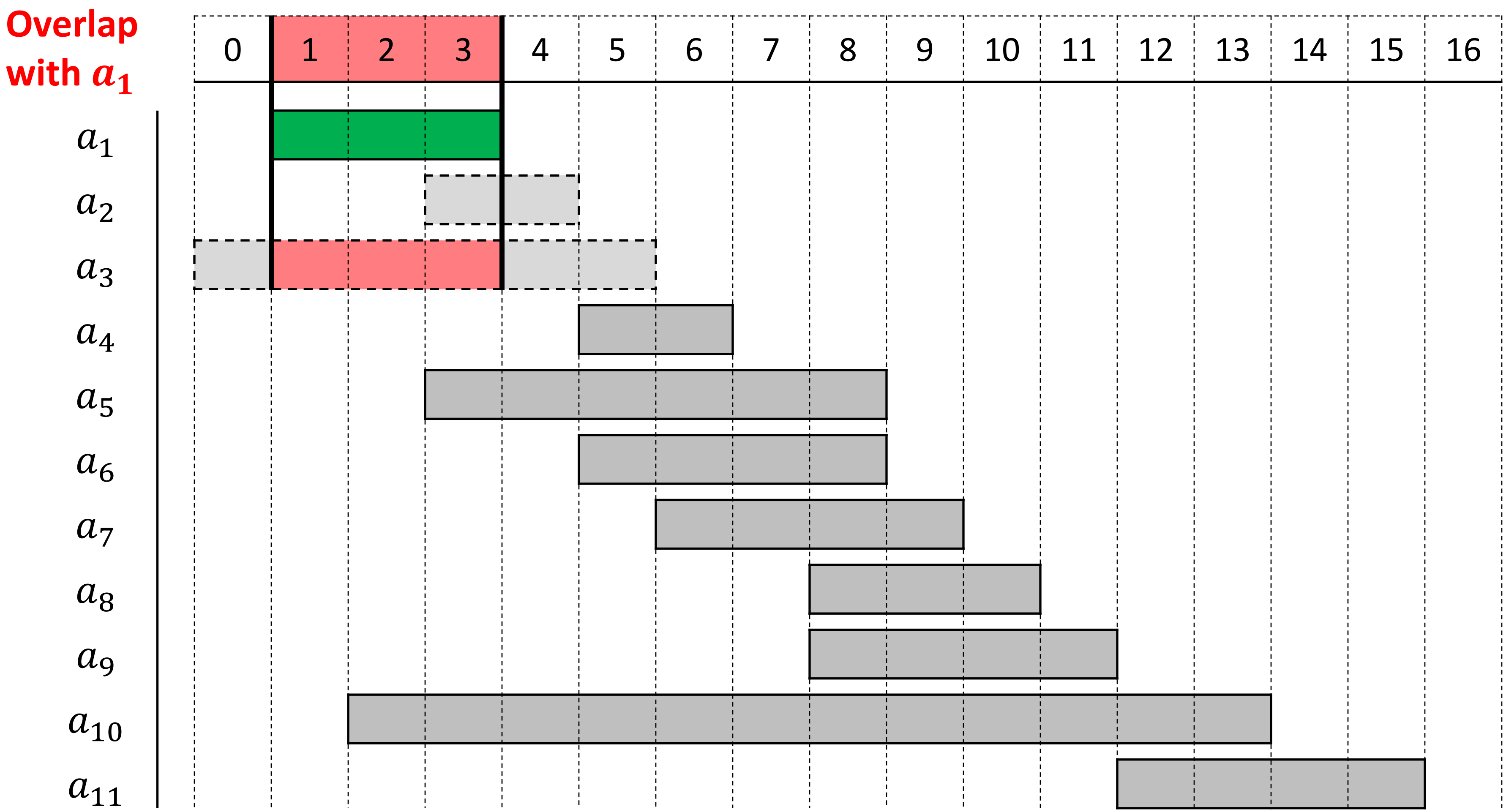
Reject a_2



Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

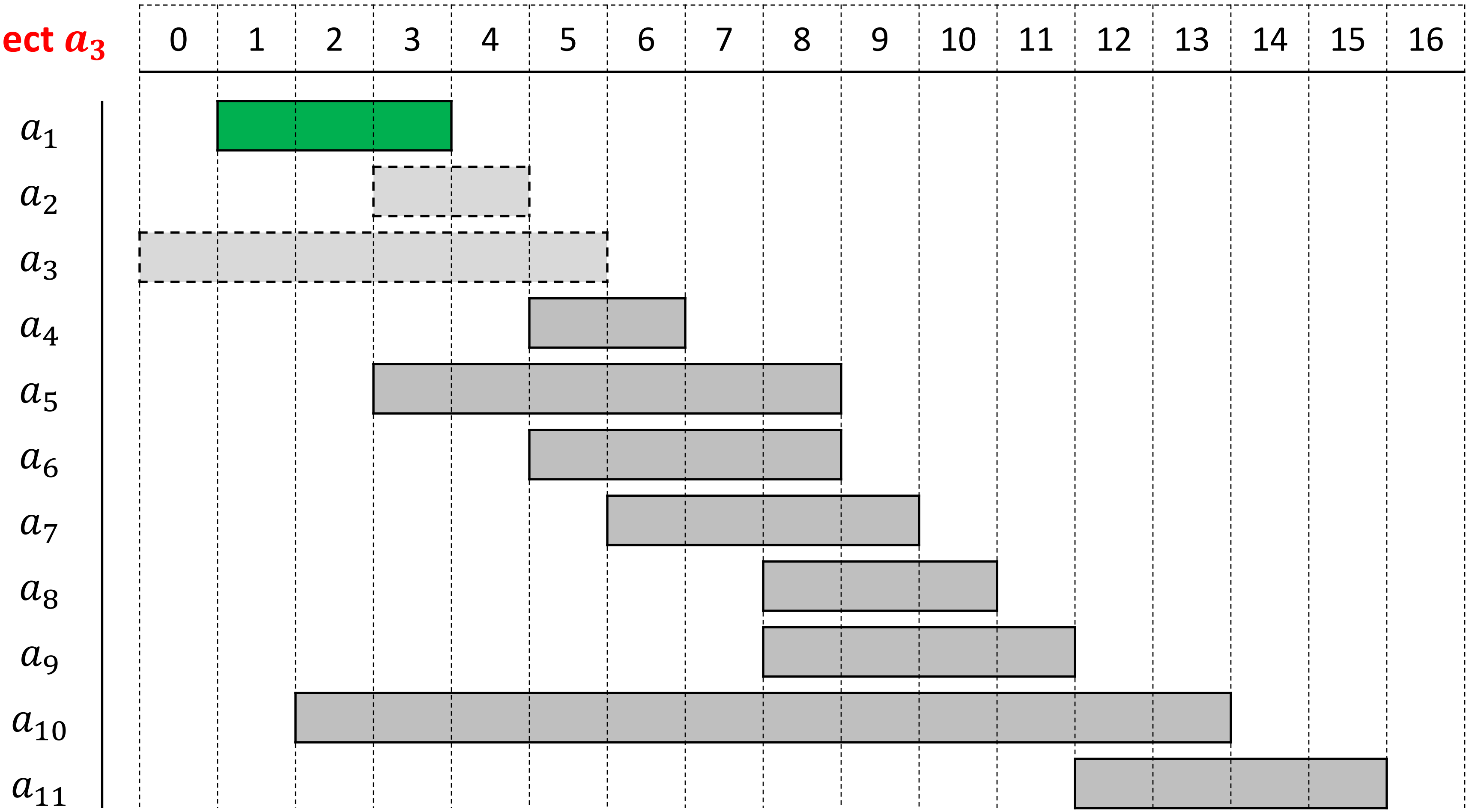


Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Reject a_3

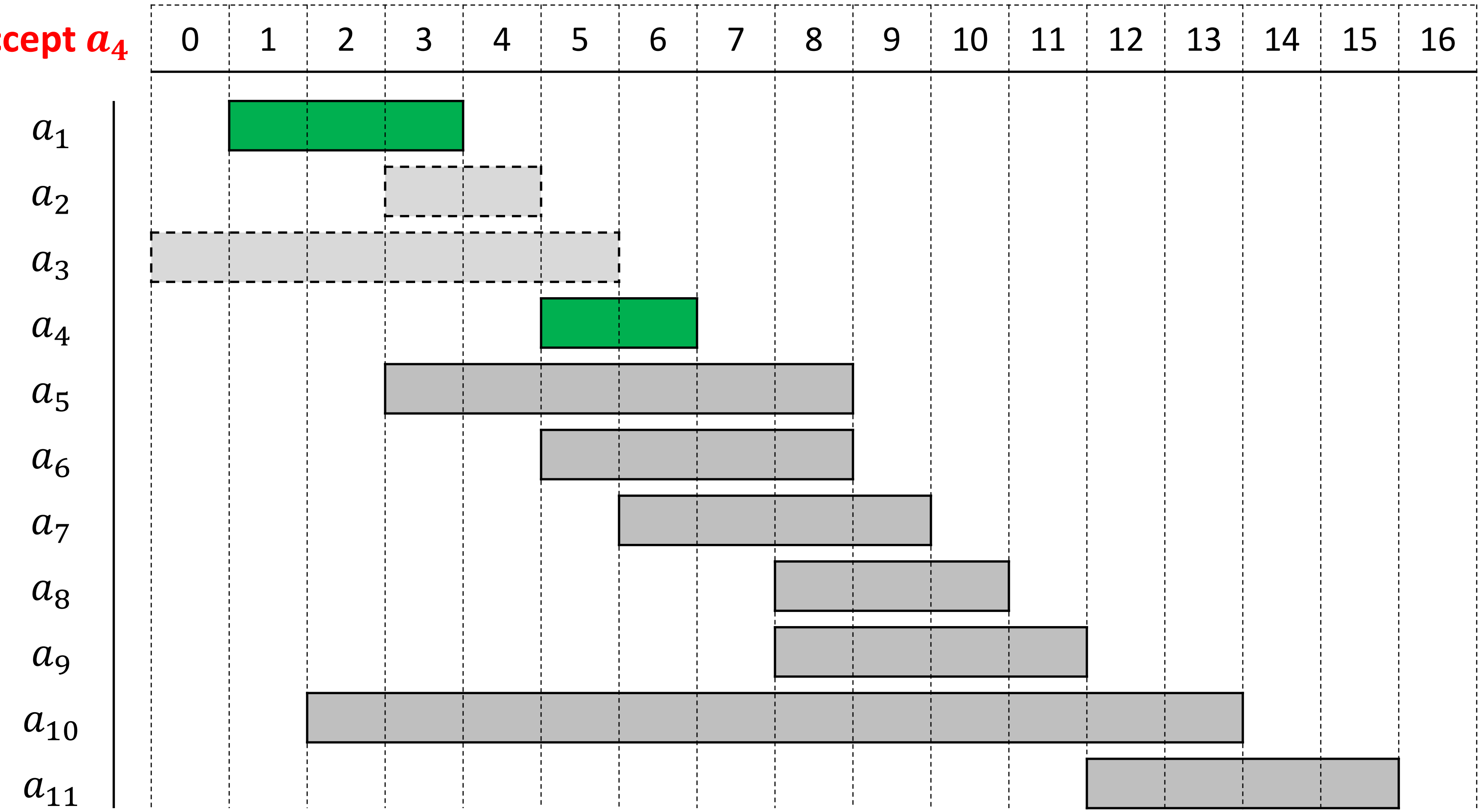


Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

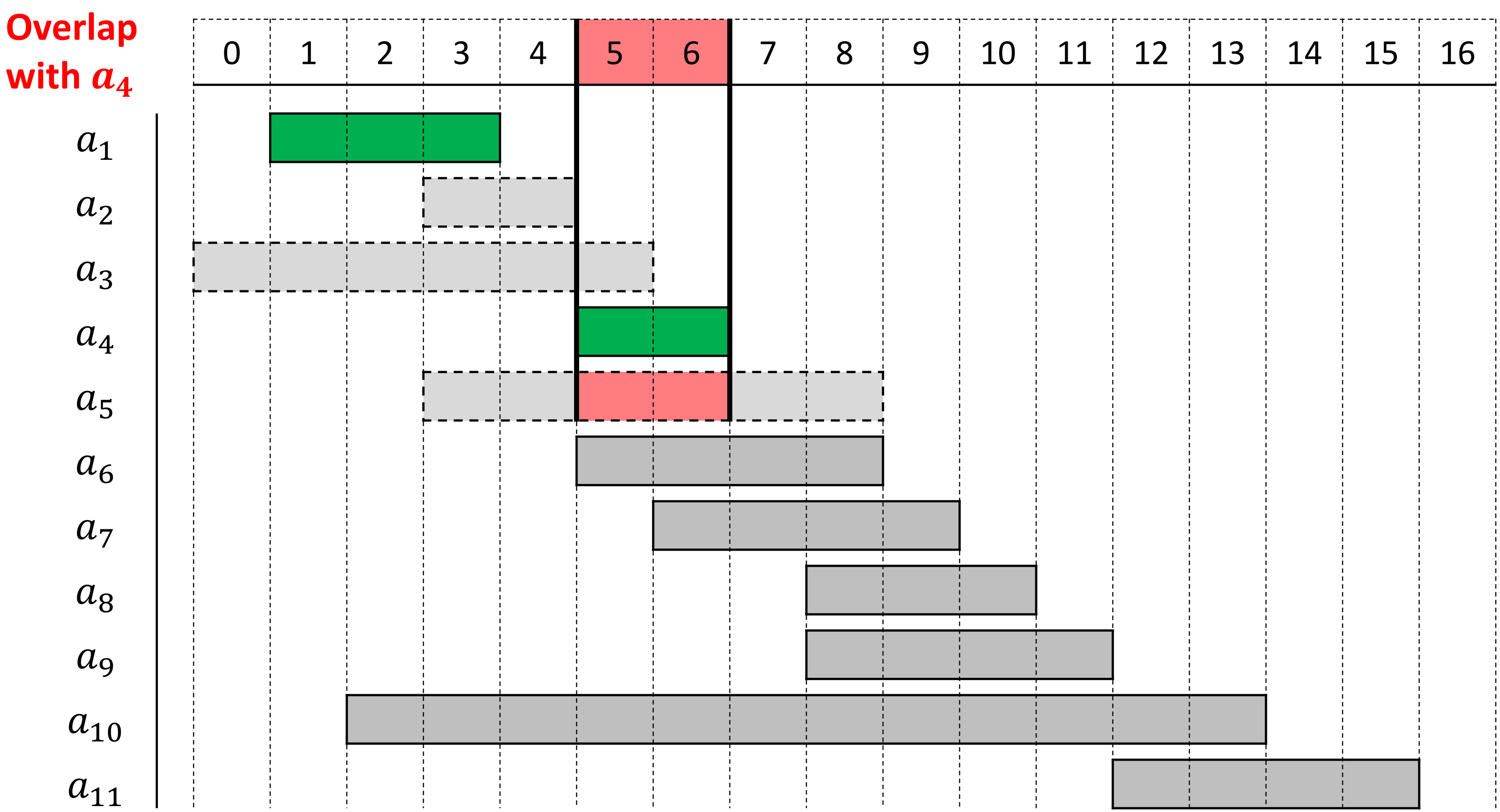
Accept a_4



Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

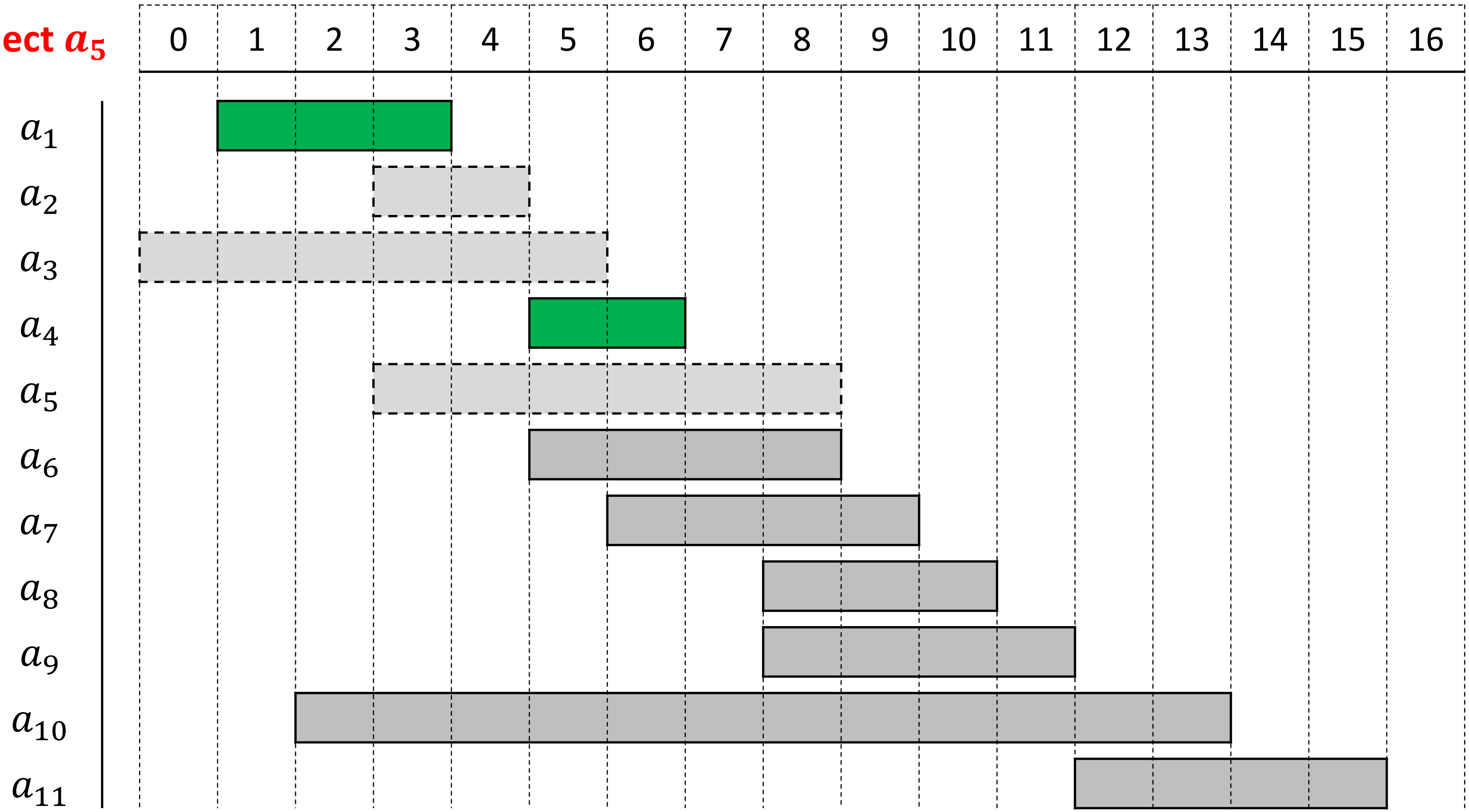


Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

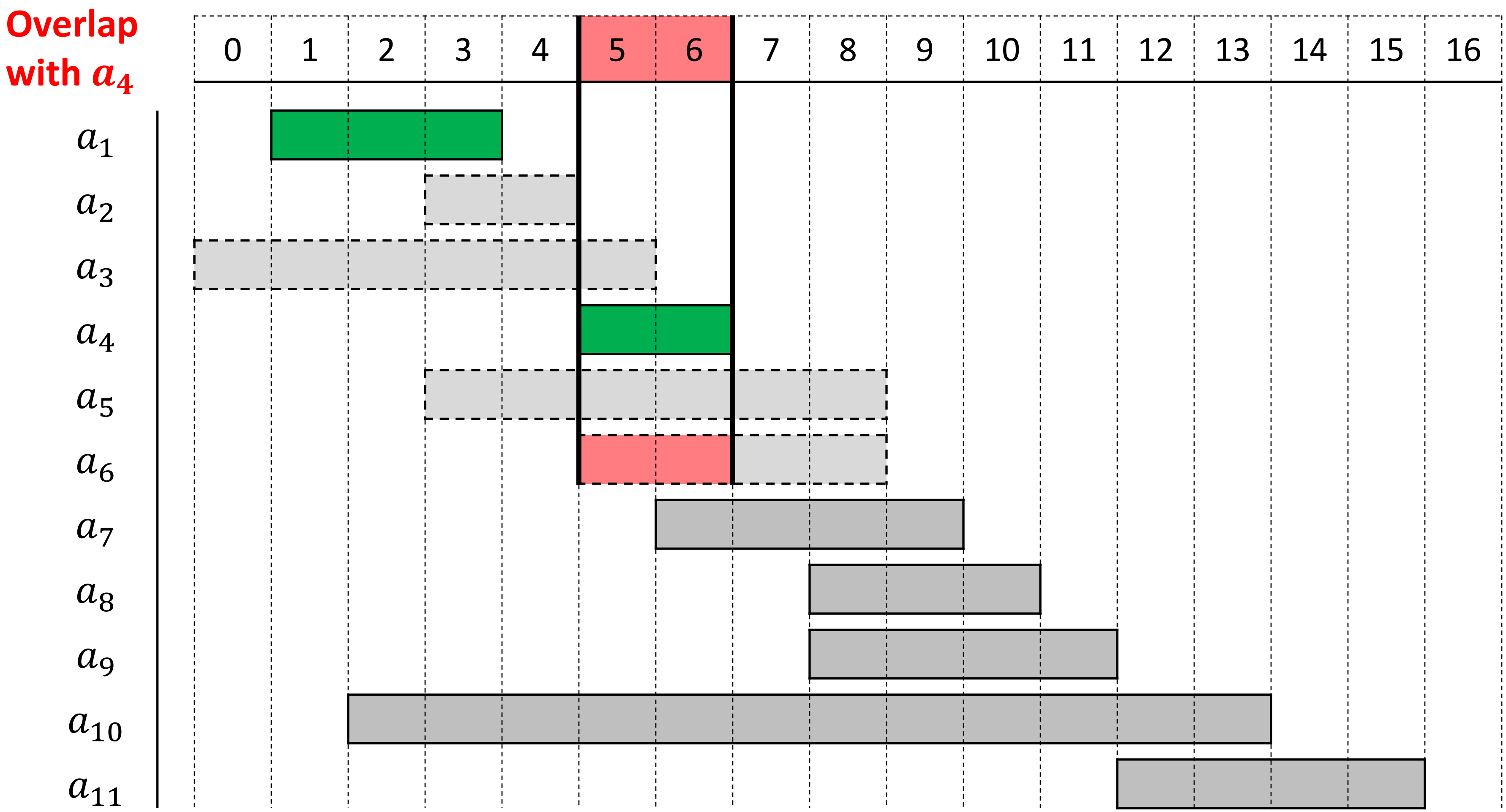
Reject a_5



Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

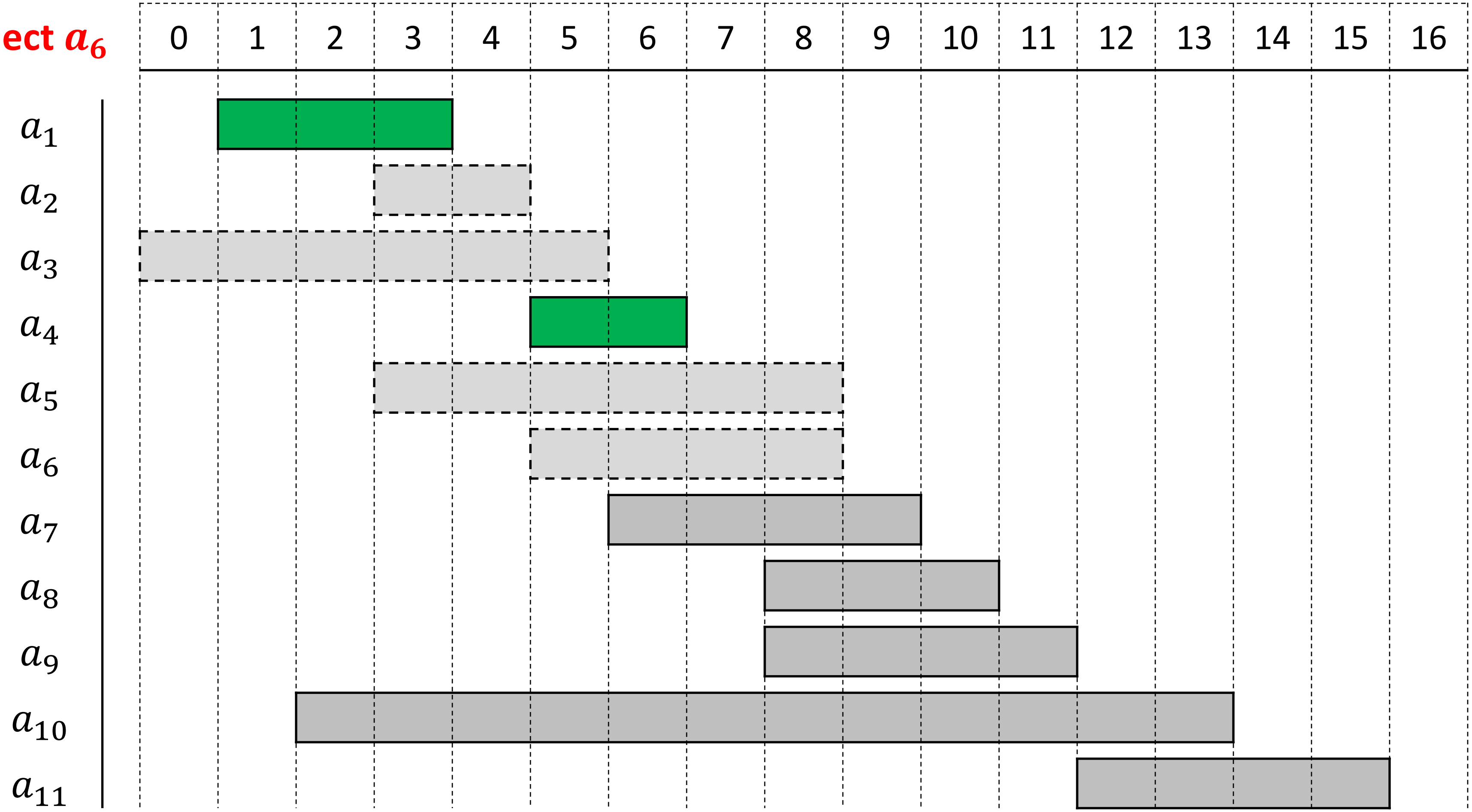


Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Reject a_6

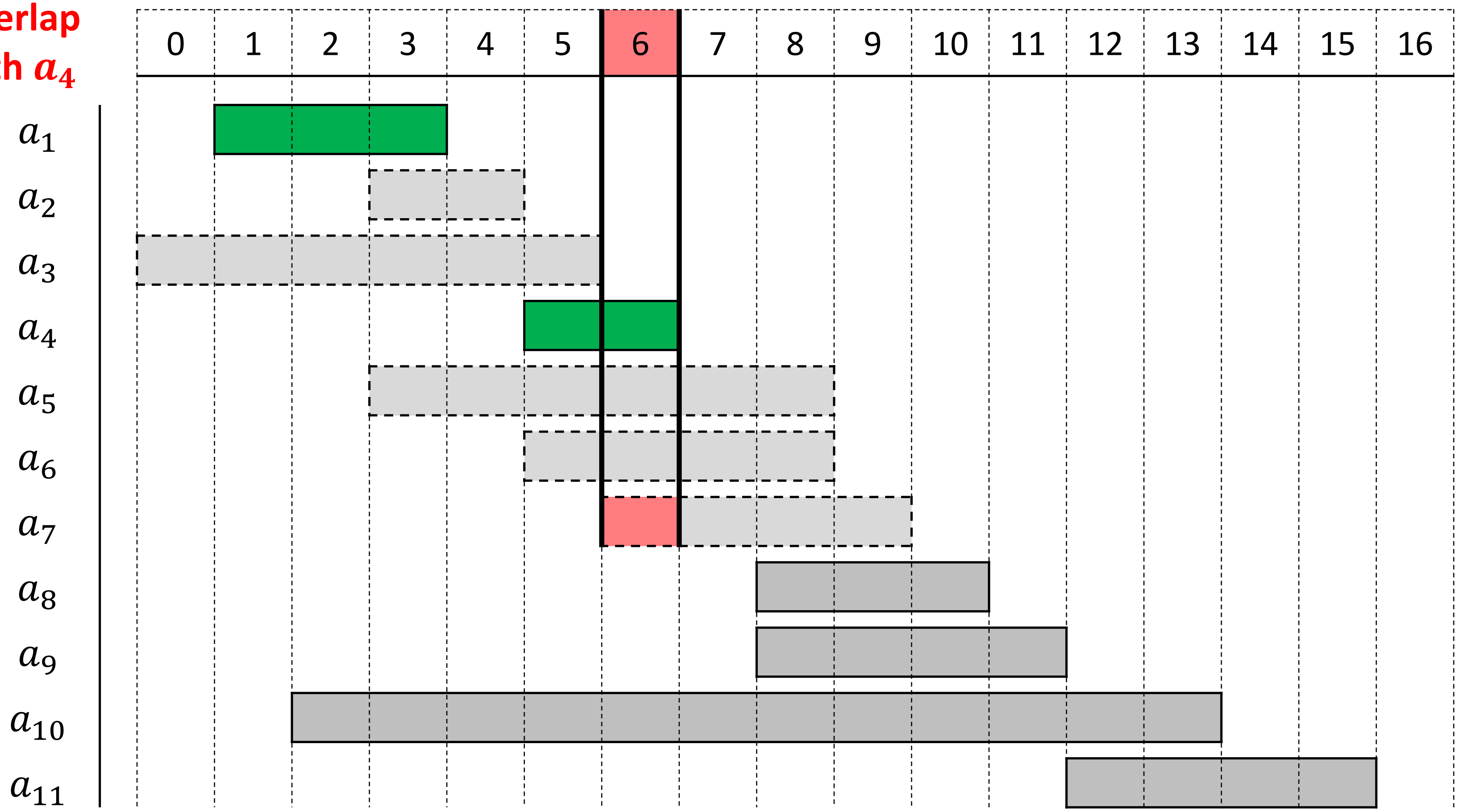


Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Overlap with a_4

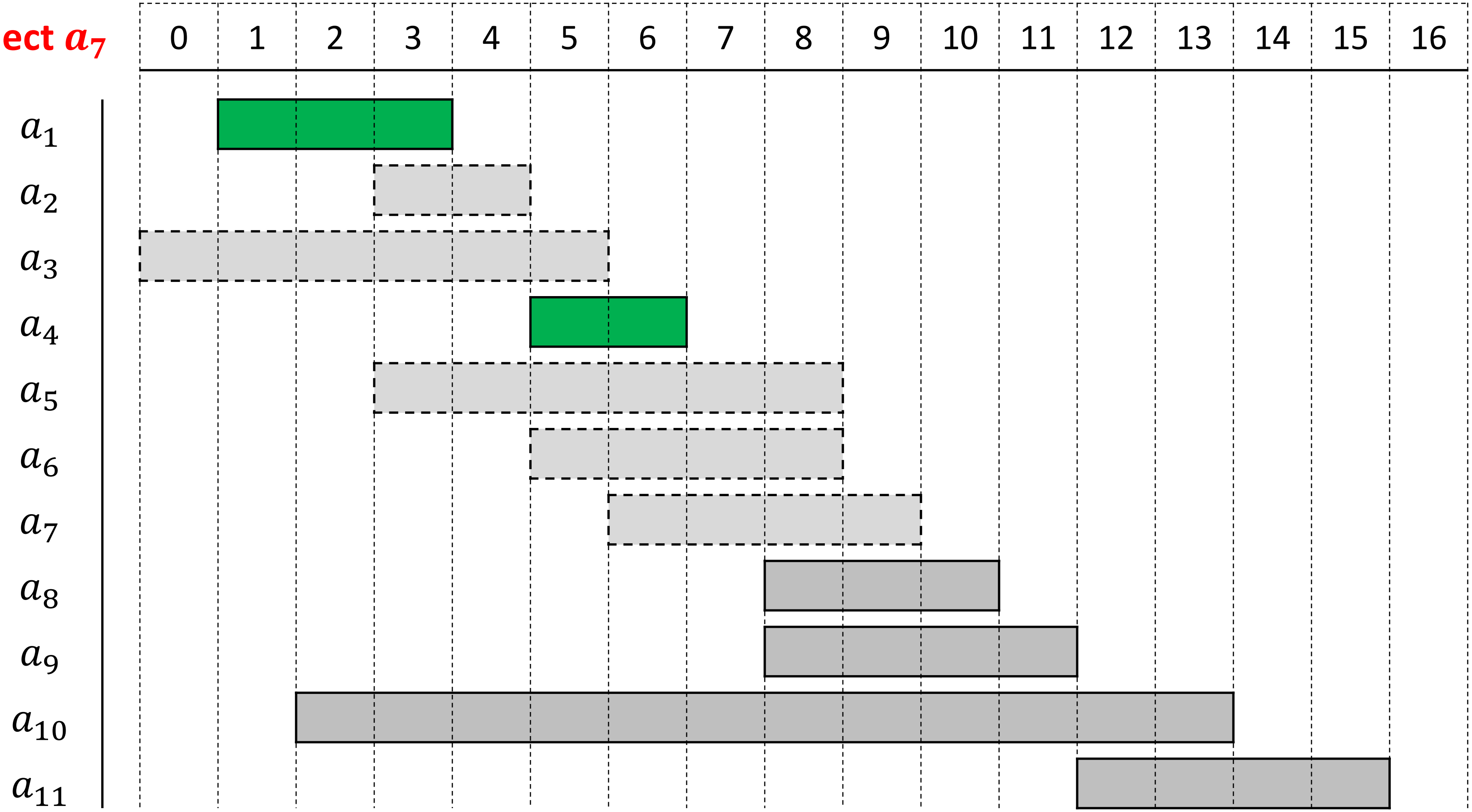


Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

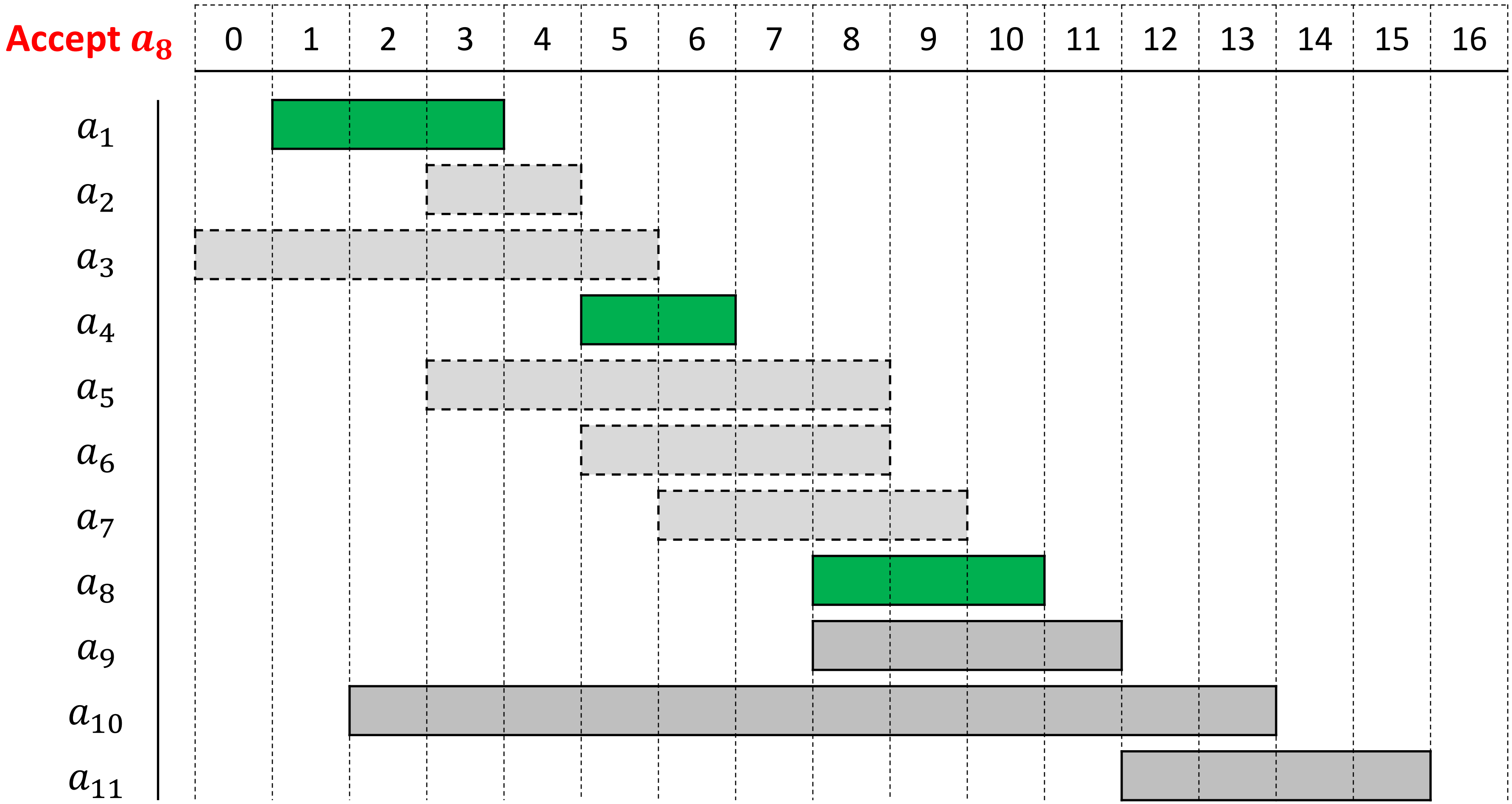
Reject a_7



Greedy Activity Selection

An example set S of activities

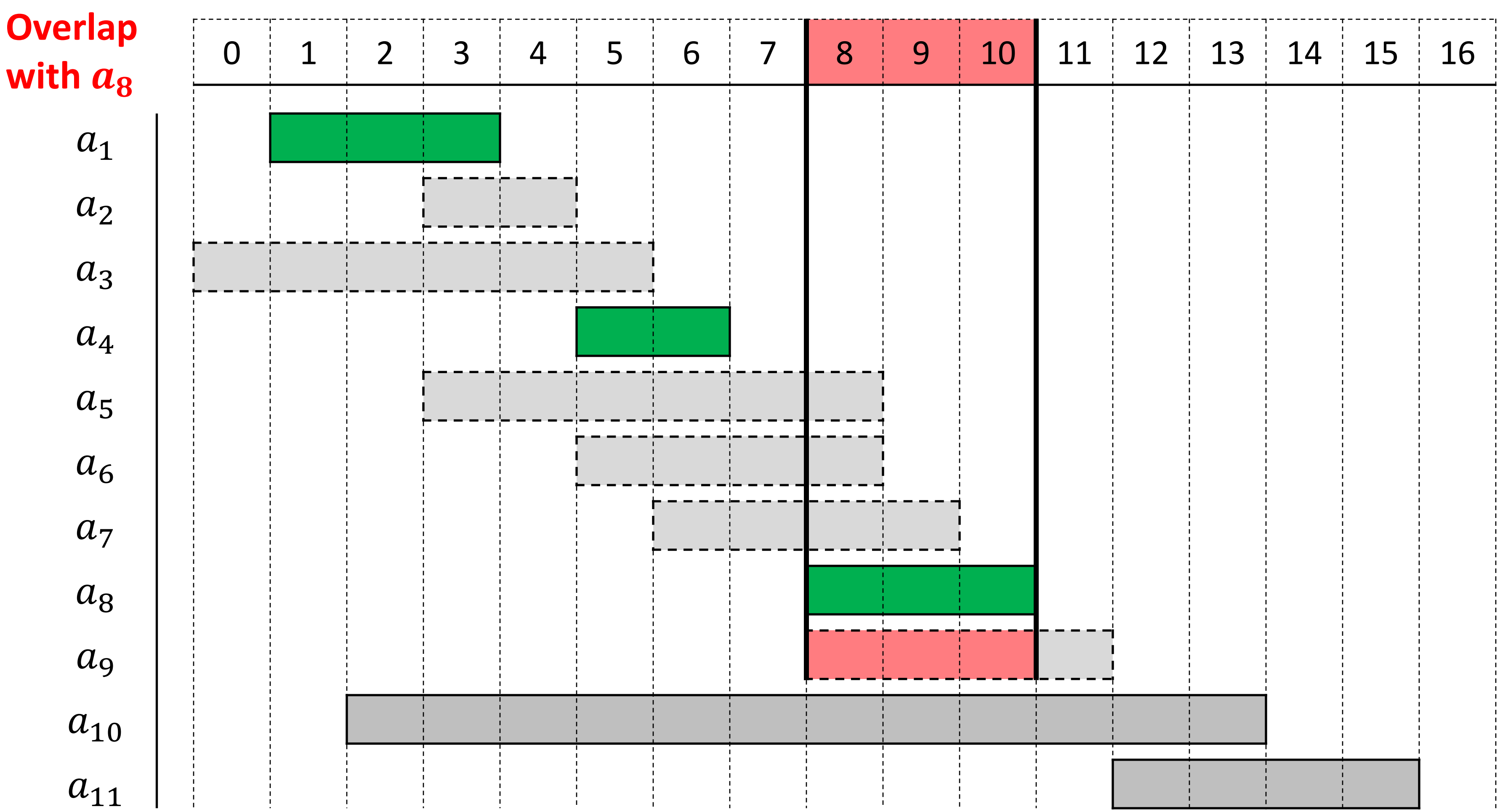
a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

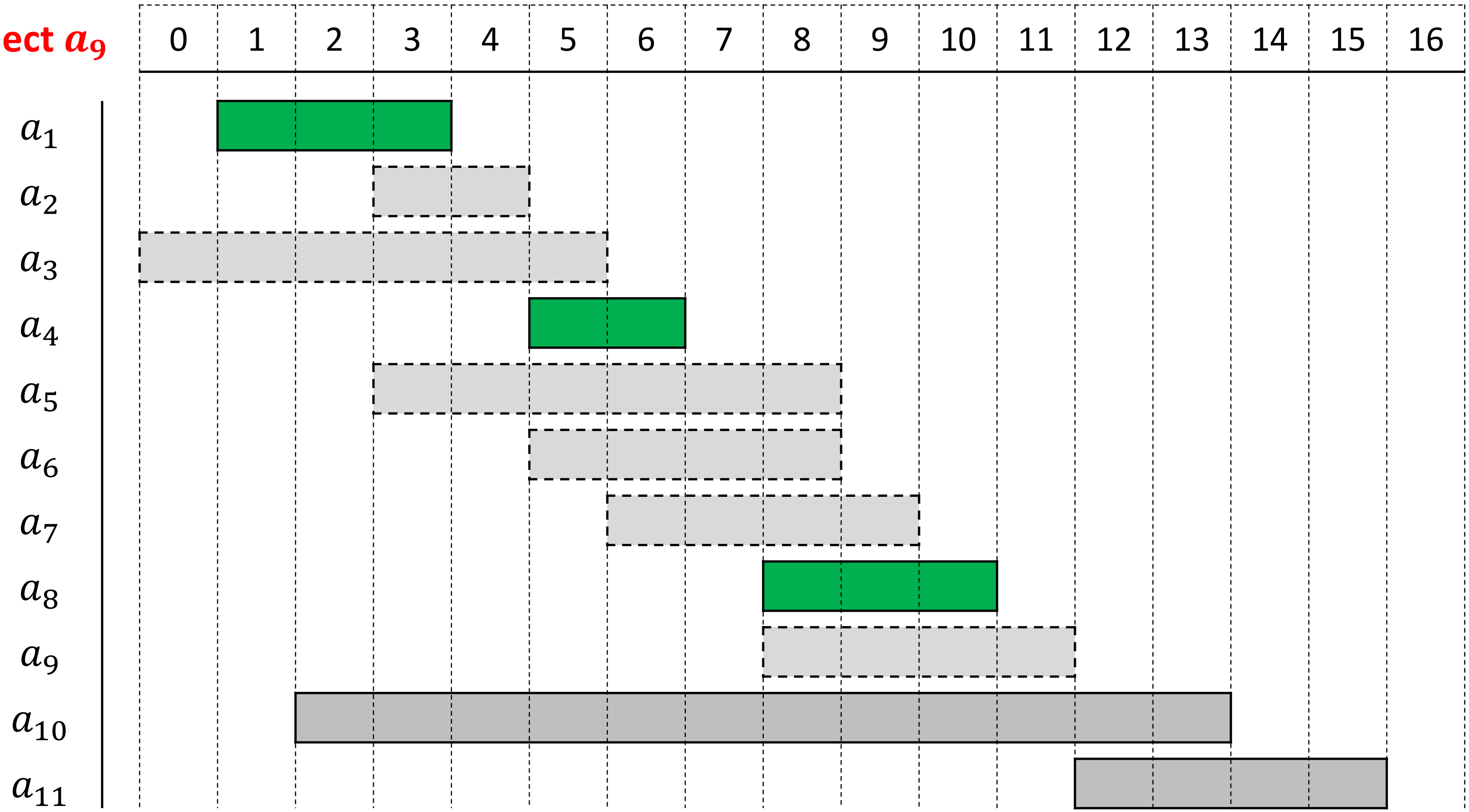


Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

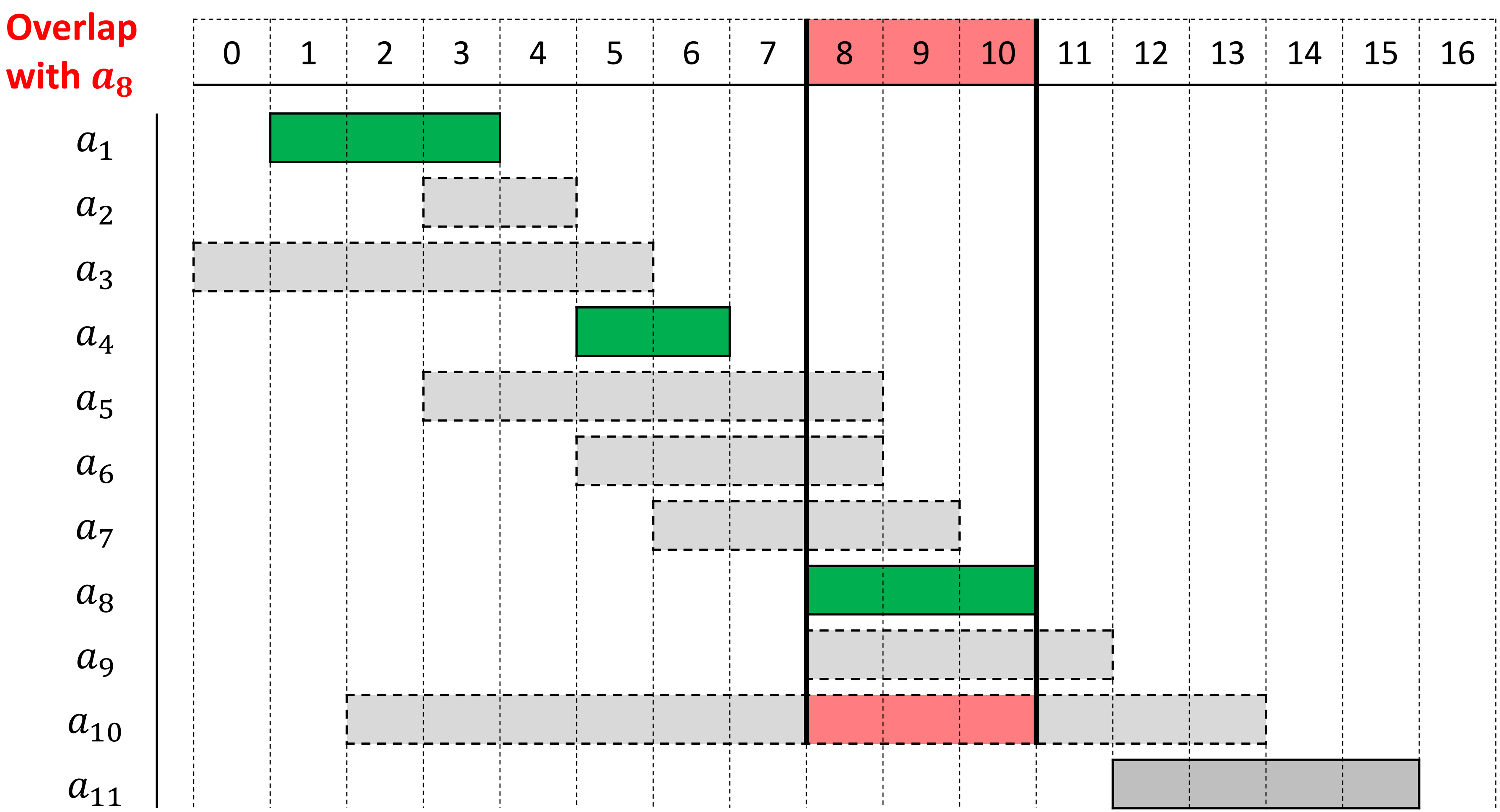
Reject a_9



Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

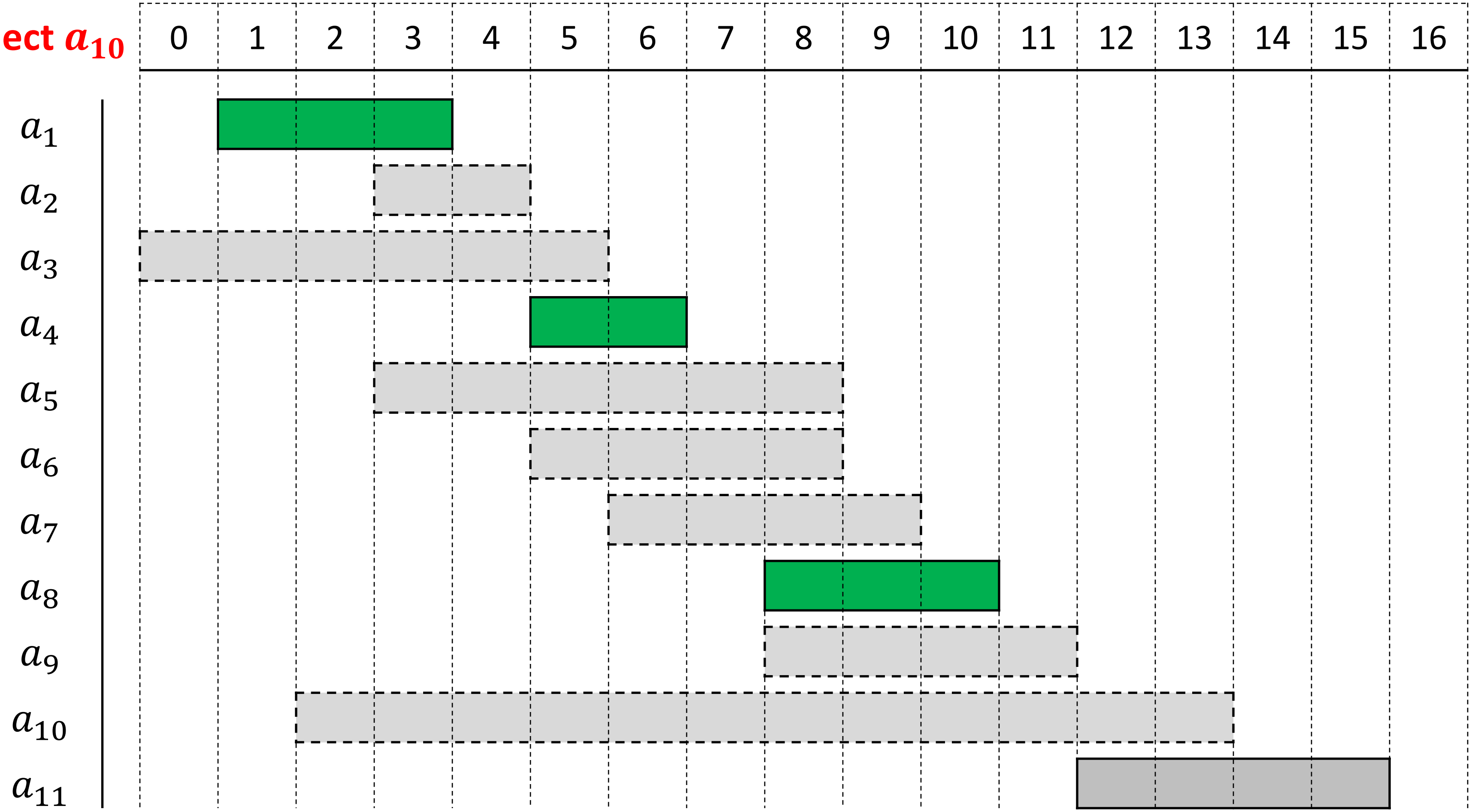


Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Reject a_{10}

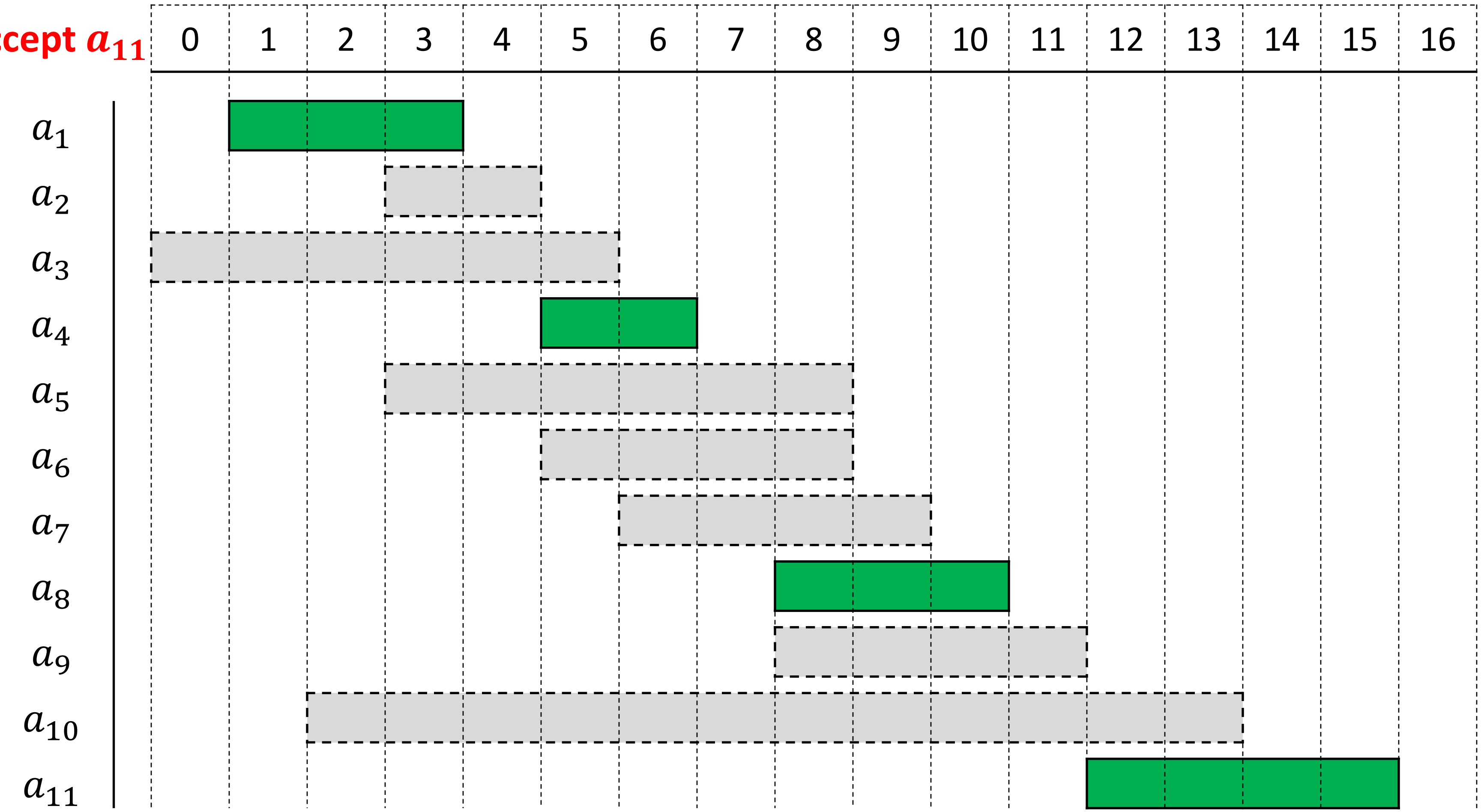


Greedy Activity Selection

An example set S of activities

a_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Accept a_{11}



Greedy Activity-Selection

GREEDY-ACTIVITY-SELECTOR (s, f)

$n \leftarrow s.length$
$A \leftarrow \{a_1\}$
$k \leftarrow 1$
for $m \leftarrow 2$ *to* n *do*
    if $s[m] \geq f[k]$ *then*
        $A \leftarrow A \cup \{a_m\}$
        $k \leftarrow m$
return A

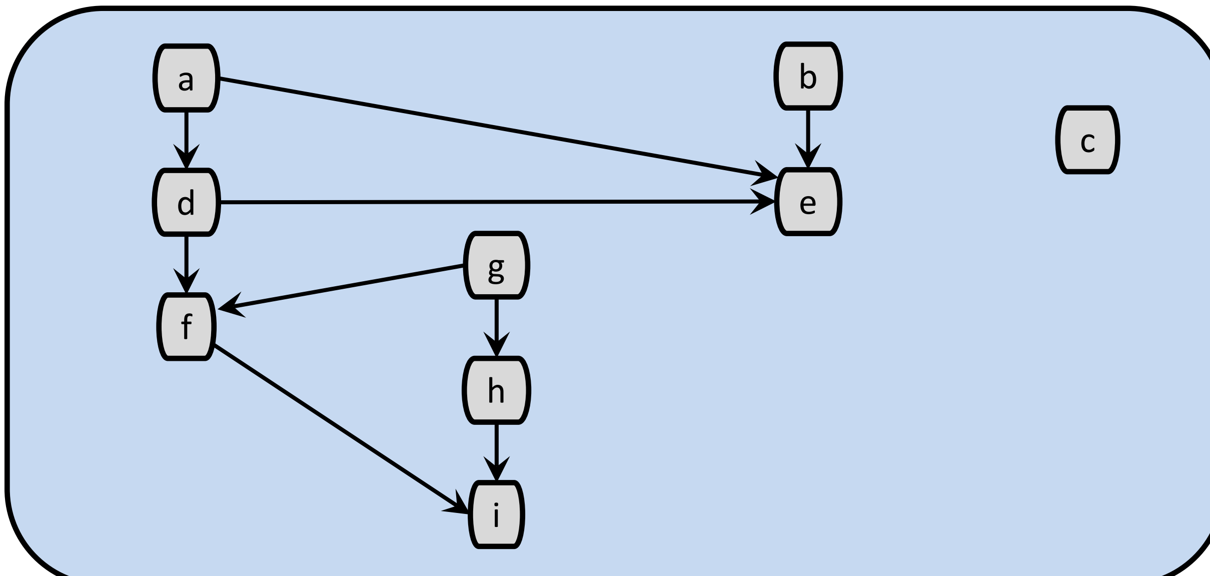
Running time $= \Theta(n)$

Topological Sort

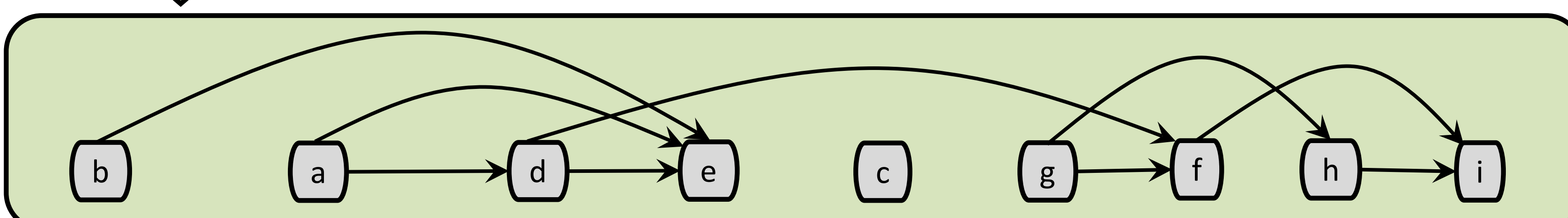
A ***topological sort*** of a DAG (i.e., directed acyclic graph) $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

A Directed Acyclic Graph (DAG)



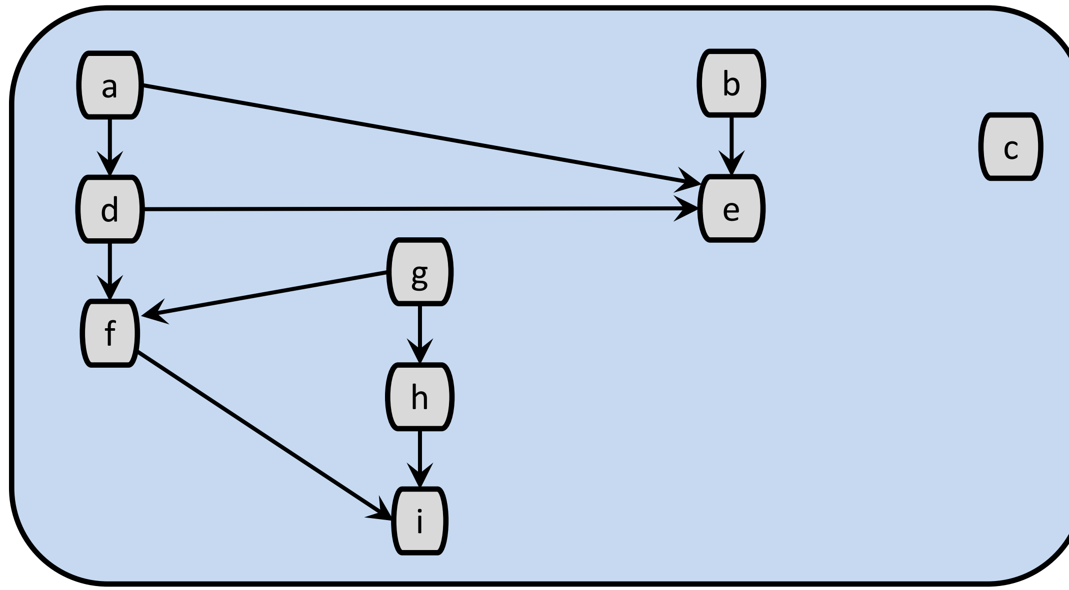
A topological sort of the DAG nodes



Topological Sort

```
GREEDY-TOPOLOGICAL-SORT ( G )
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

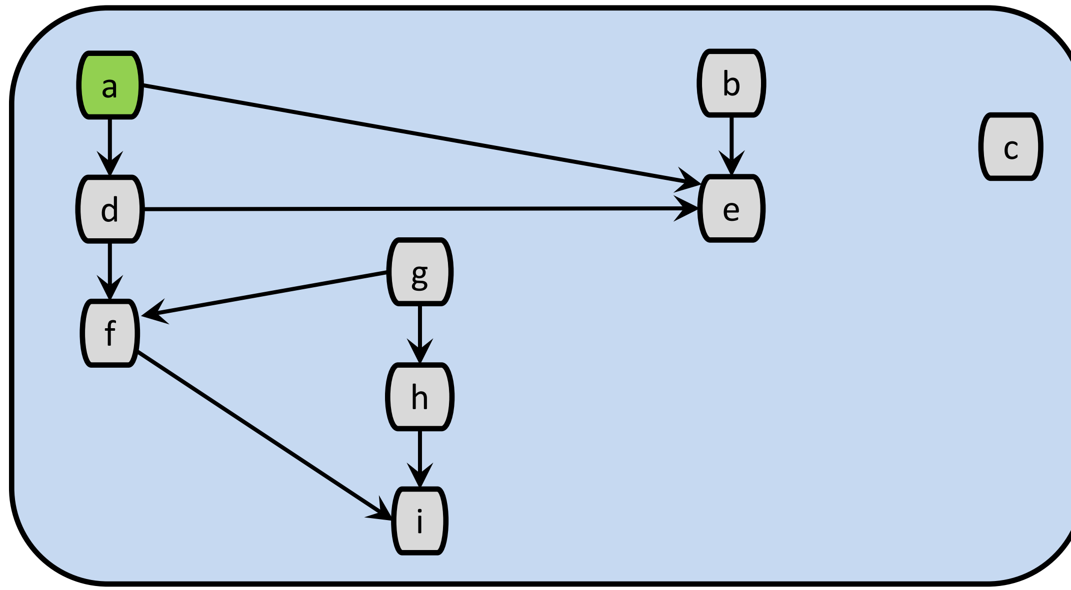
$n \leftarrow 9,\ i \leftarrow 0$



Topological Sort

```
GREEDY-TOPOLOGICAL-SORT ( G )
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

choose node a with no incoming edges



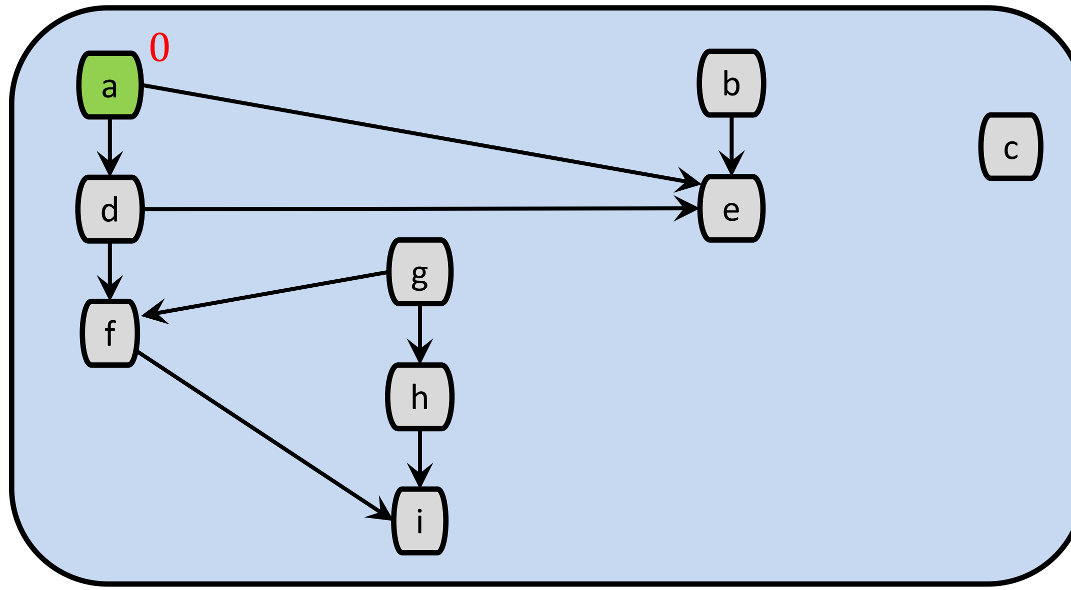
Topological Sort

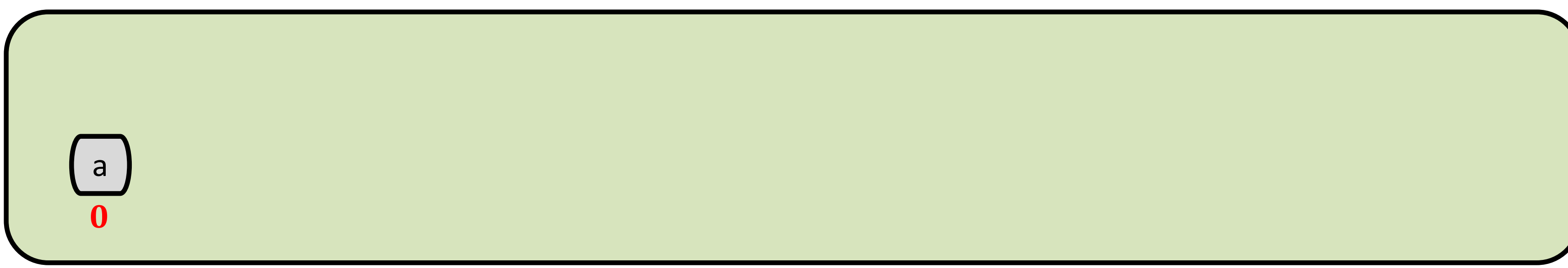
GREEDY-TOPOLOGICAL-SORT (G)

1. $n \leftarrow |G.V|$
2. $i \leftarrow 0$
3. *while* $i < n$ *do*
4. find a node $x \in G.V$ with no incoming edges
5. assign number i to x
6. $i \leftarrow i + 1$
7. remove x with all its outgoing edges from G

assign number $i = 0$ to node a

$i \leftarrow 1$ (increment i)



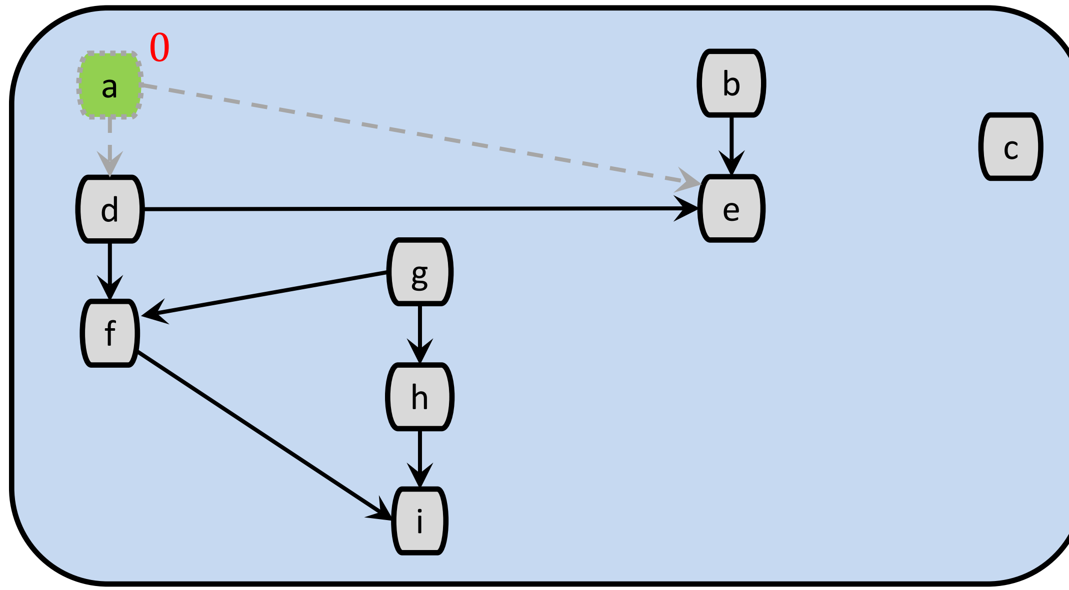


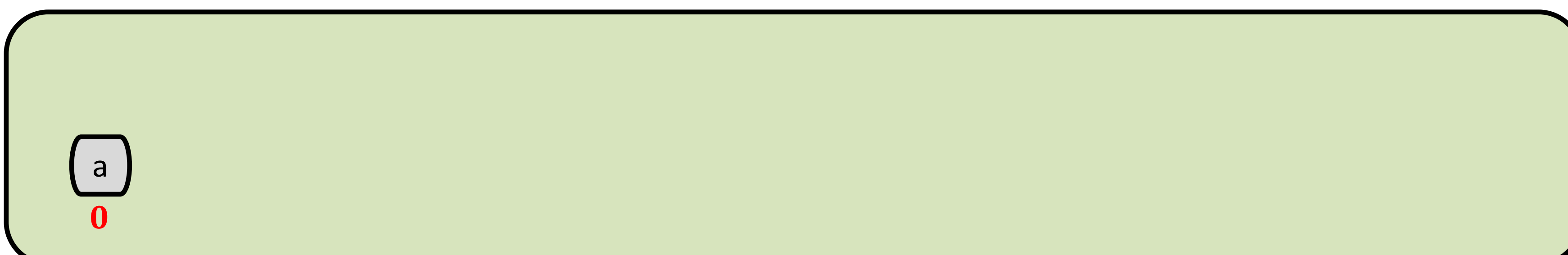
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

```
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

remove node a with all its outgoing edges



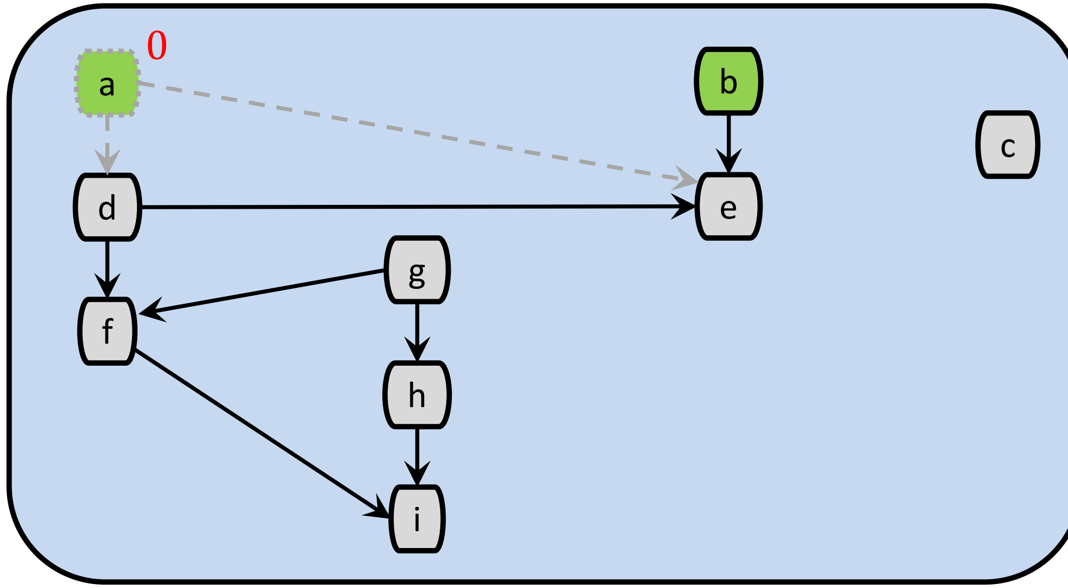


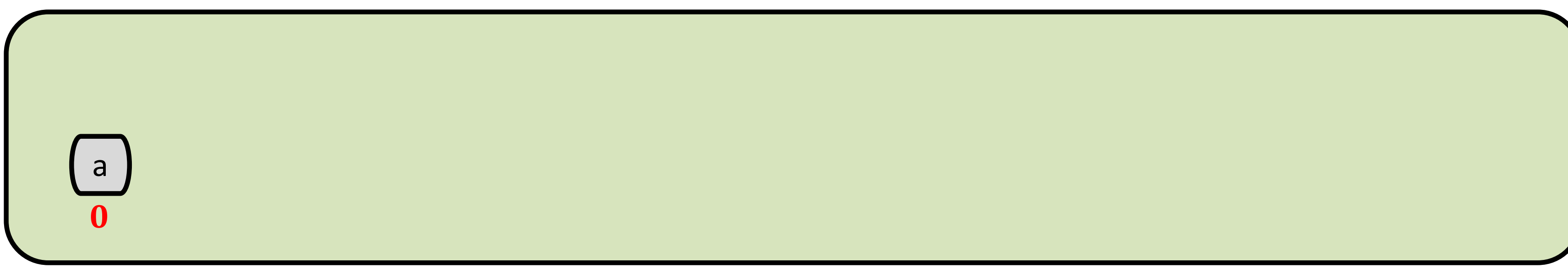
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

```
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

choose node b with no incoming edges



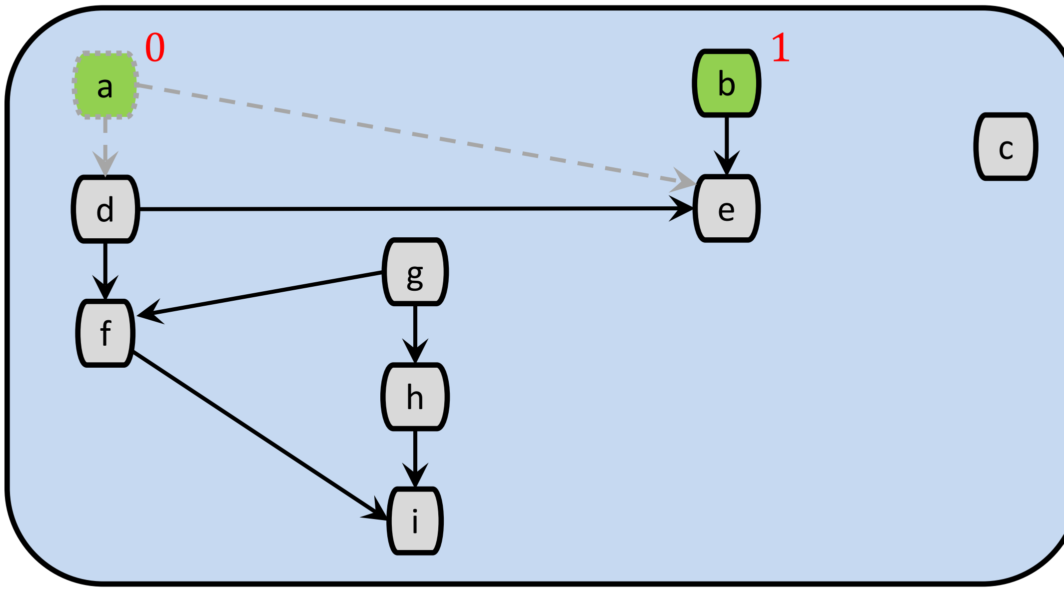


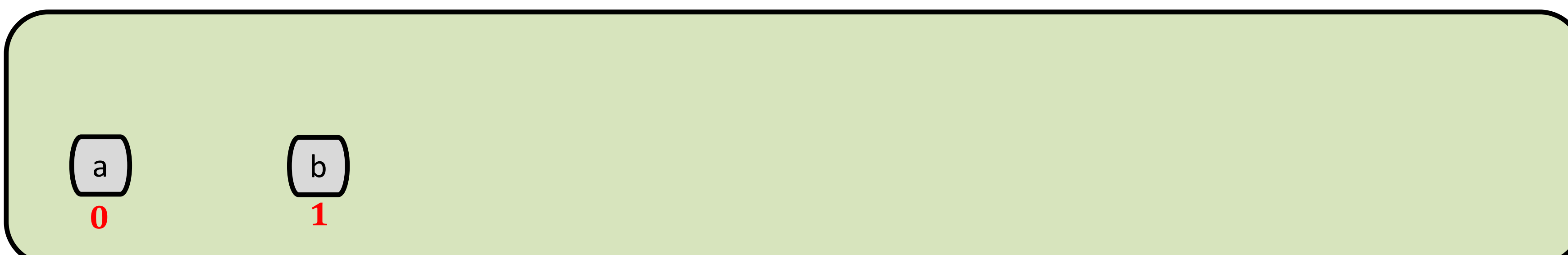
Topological Sort

```
GREEDY-TOPOLOGICAL-SORT ( G )
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

assign number $i = 1$ to node b

$i \leftarrow 2$ (increment i)





Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

$n \leftarrow |G.V|$

$i \leftarrow 0$

while $i < n$ *do*

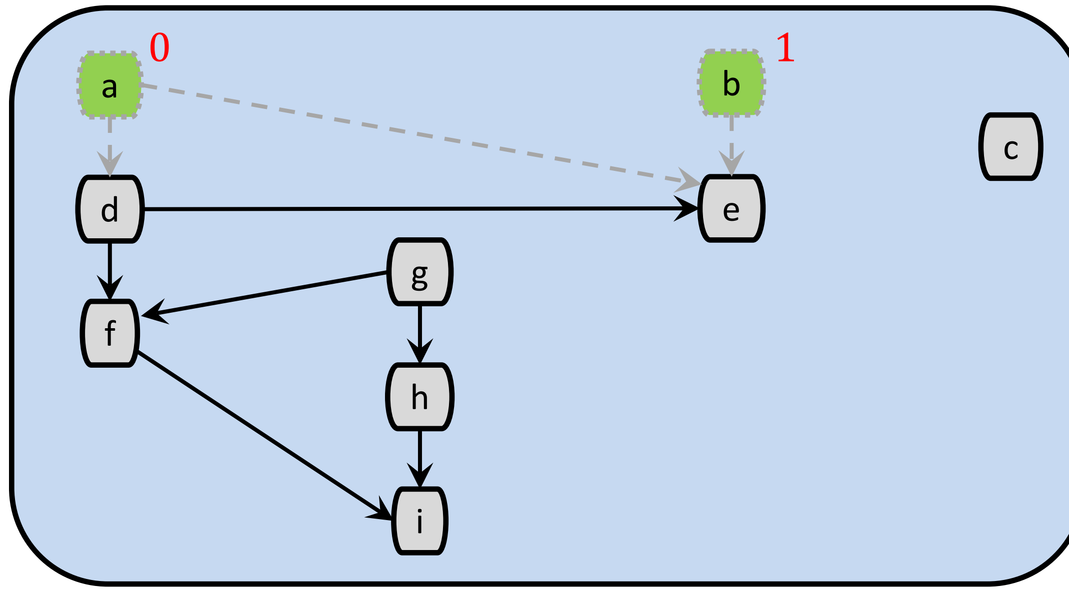
    find a node $x \in G.V$ with no incoming edges

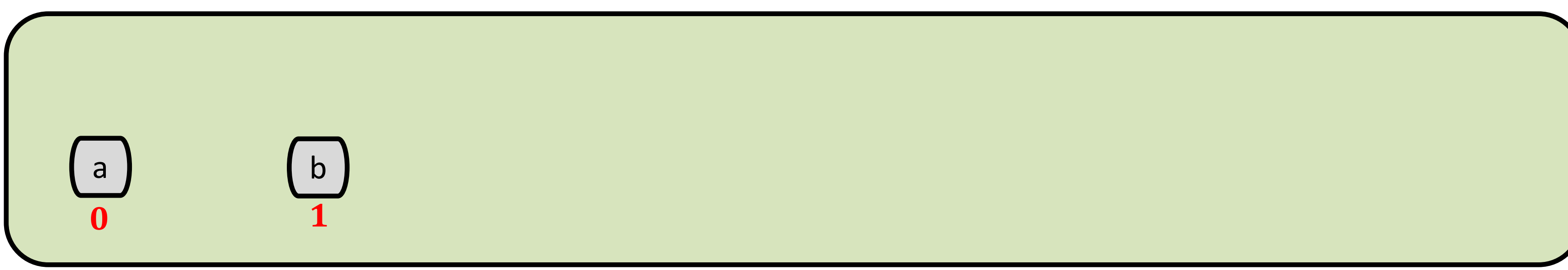
    assign number i to x

    $i \leftarrow i + 1$

    remove x with all its outgoing edges from G

remove node *b* with all its outgoing edges



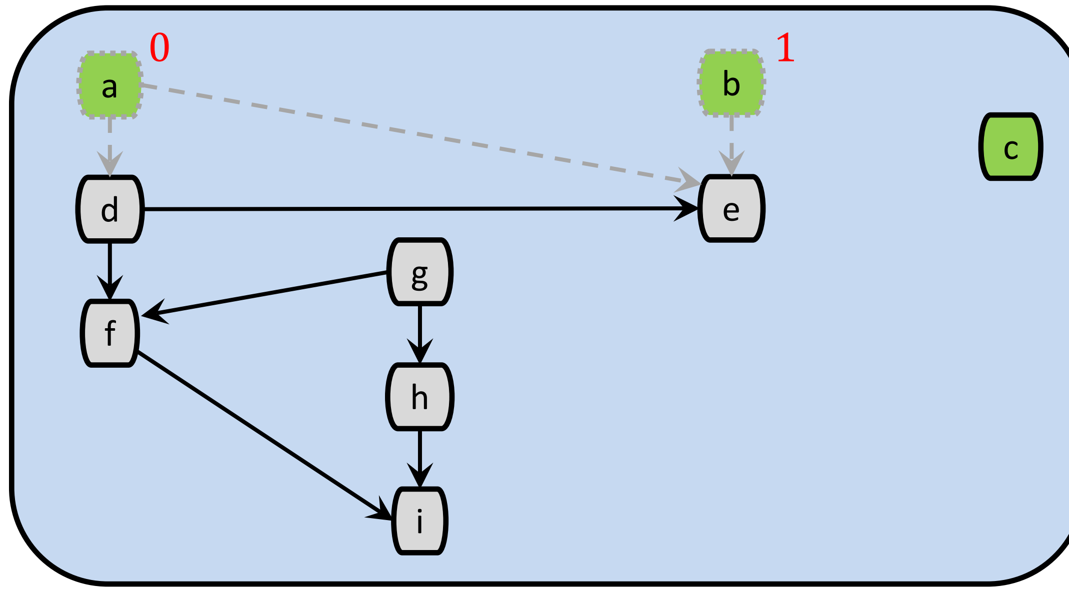


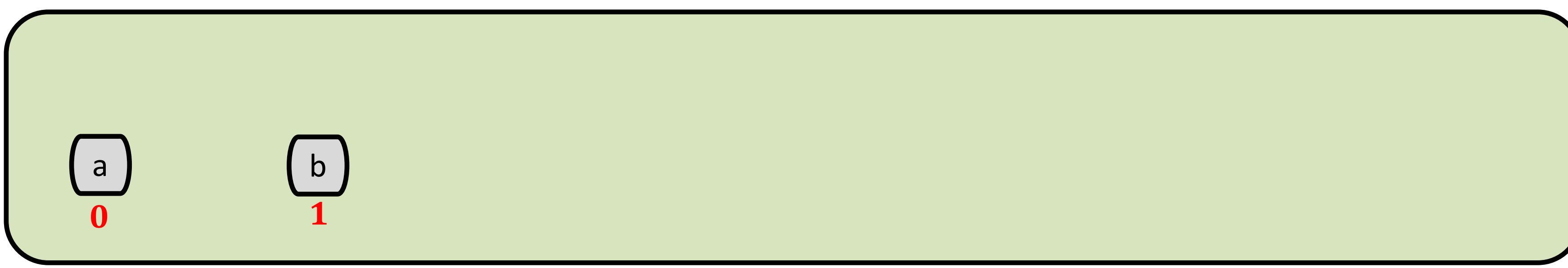
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

```
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

choose node c with no incoming edges





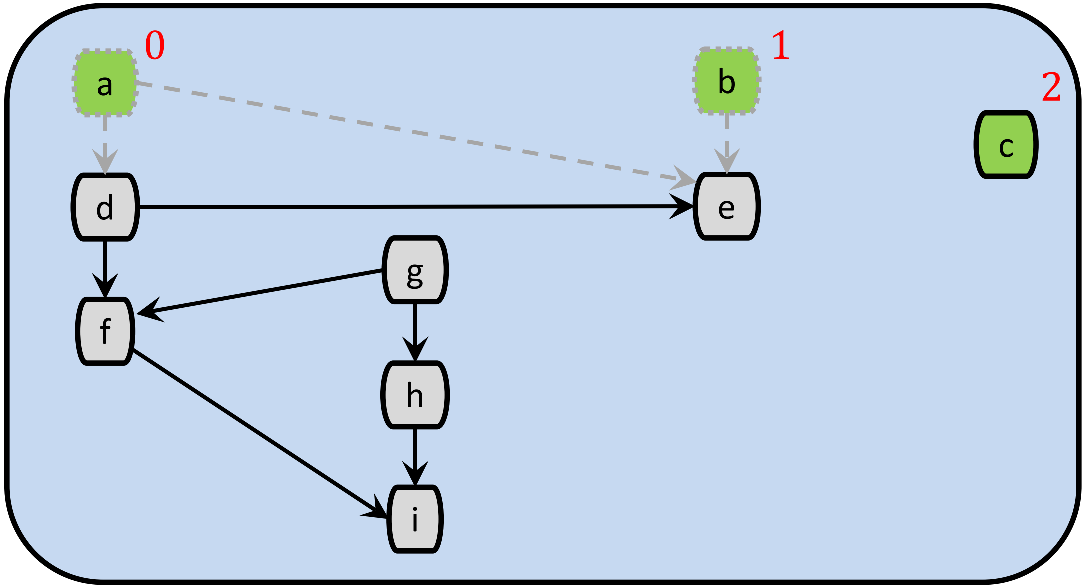
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

1. $n \leftarrow |G.V|$
2. $i \leftarrow 0$
3. *while* $i < n$ *do*
4. find a node $x \in G.V$ with no incoming edges
5. assign number i to x
6. $i \leftarrow i + 1$
7. remove x with all its outgoing edges from G

assign number $i = 2$ to node c

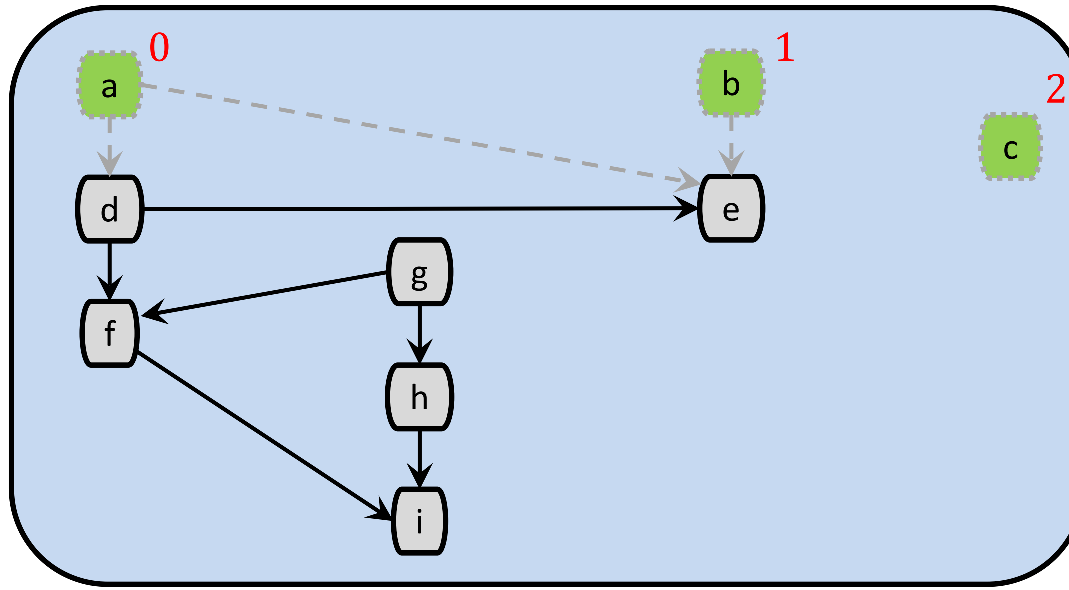
$i \leftarrow 3$ (increment i)

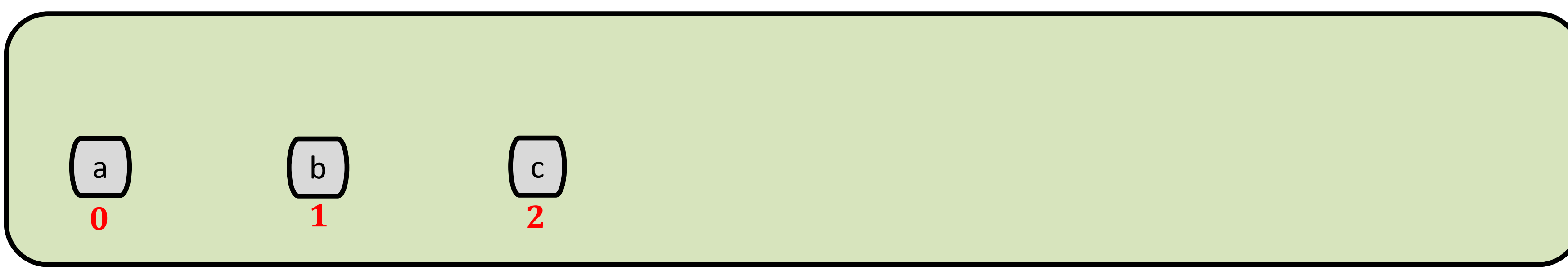


Topological Sort

```
GREEDY-TOPOLOGICAL-SORT ( G )
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

remove node *c* with all its outgoing edges

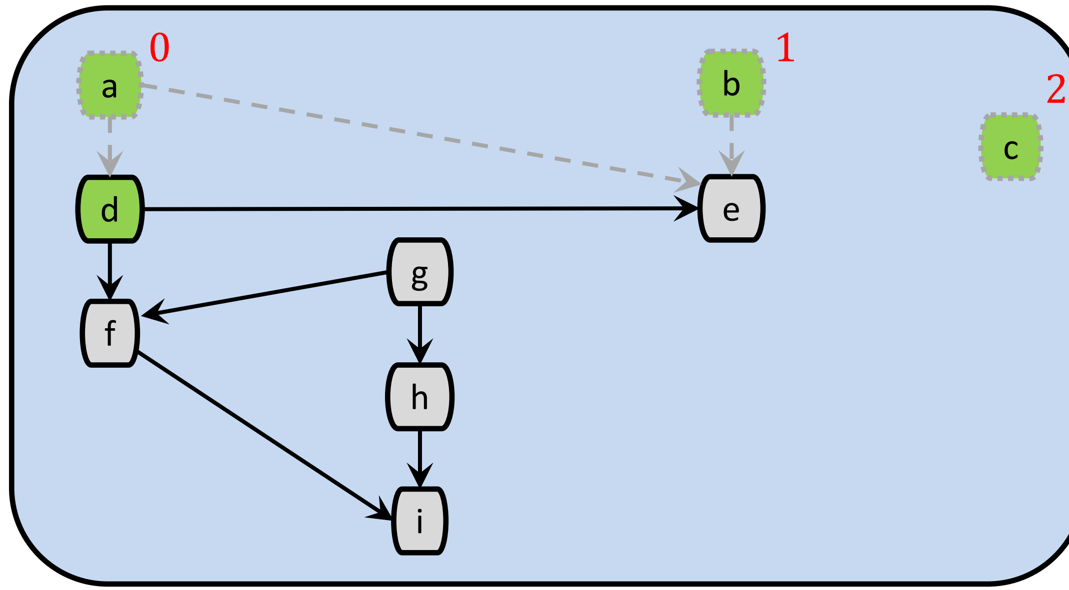


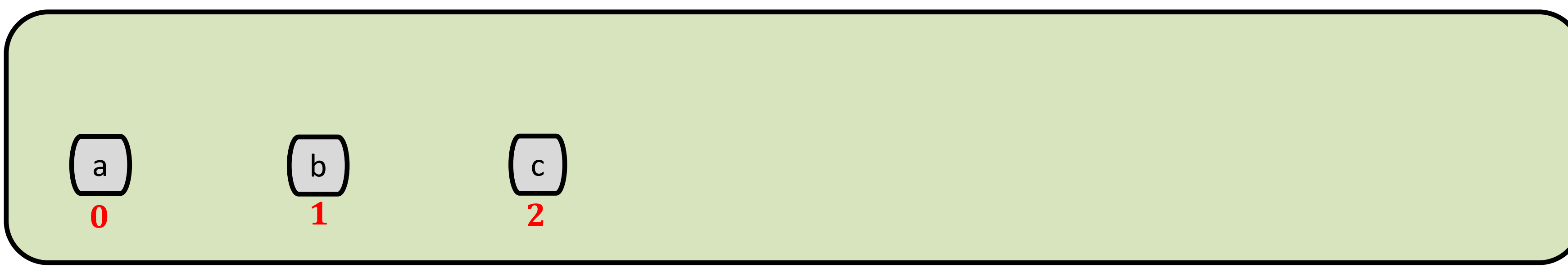


Topological Sort

```
GREEDY-TOPOLOGICAL-SORT ( G )
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

choose node d with no incoming edges





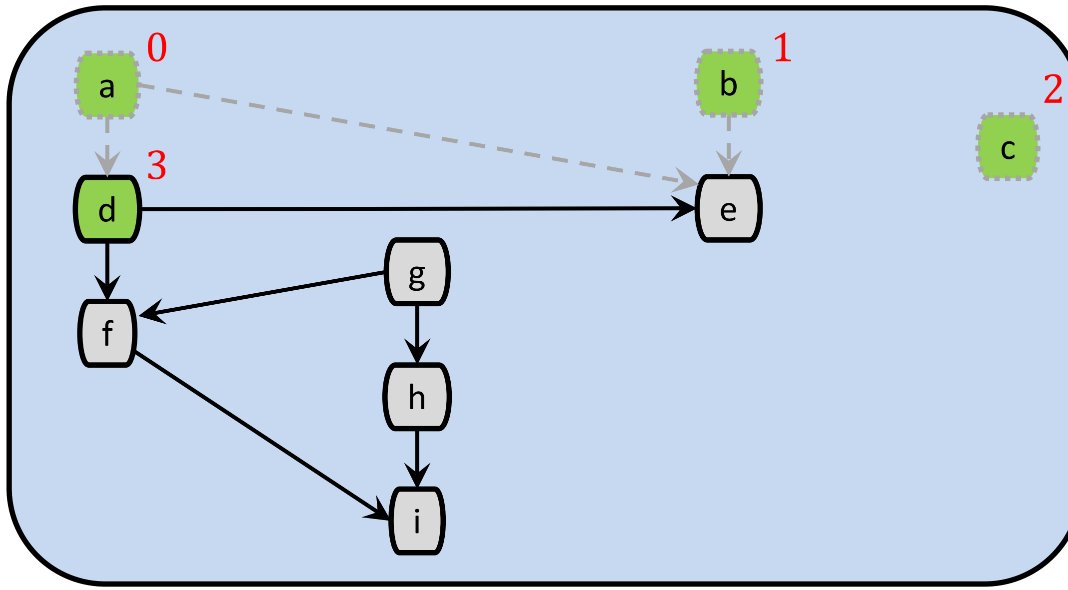
Topological Sort

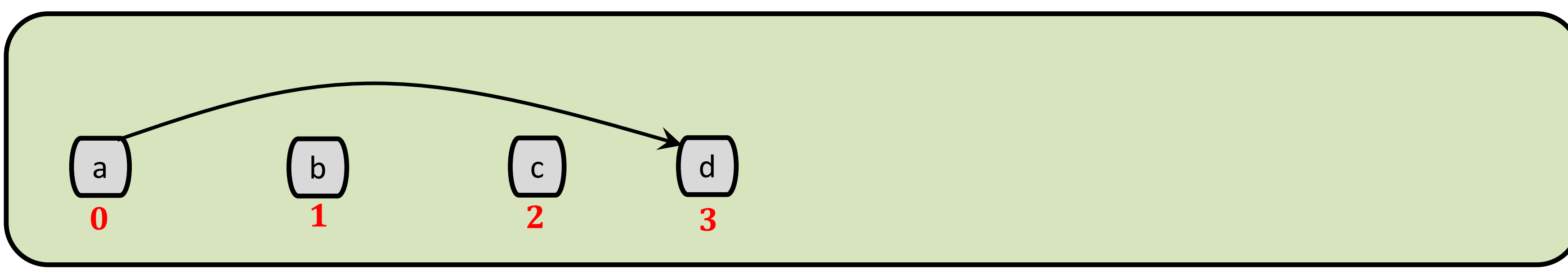
GREEDY-TOPOLOGICAL-SORT (G)

```
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

assign number $i = 3$ to node d

$i \leftarrow 4$ (increment i)



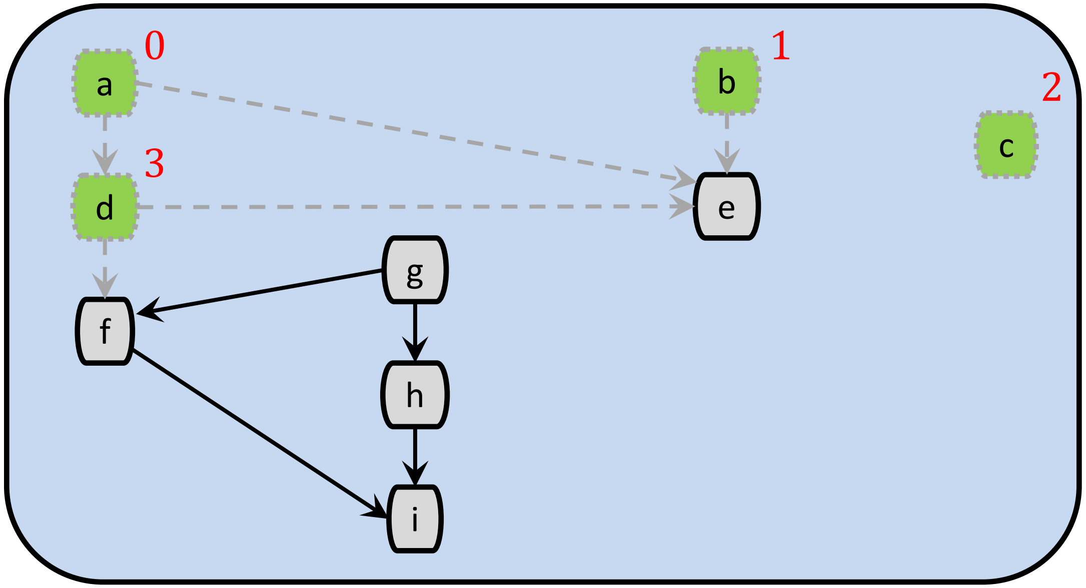


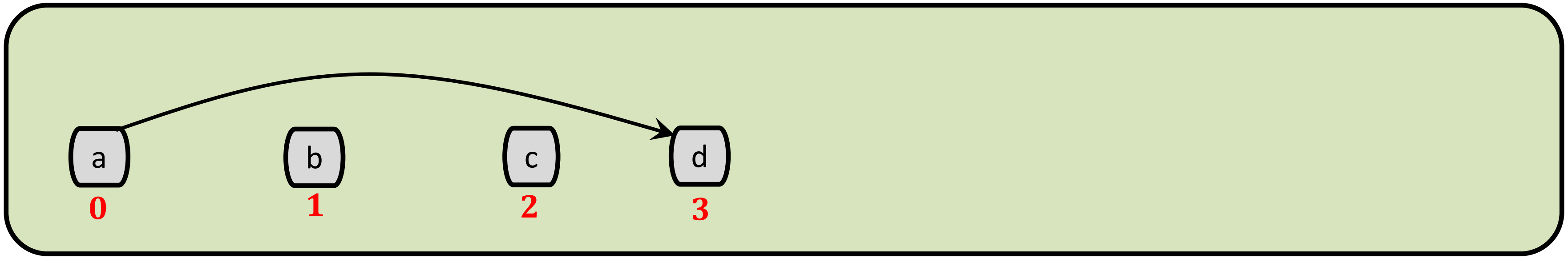
Topological Sort

Greedy-Topological-Sort (G)

1. $n \leftarrow |G.V|$
2. $i \leftarrow 0$
3. *while* $i < n$ *do*
4. find a node $x \in G.V$ with no incoming edges
5. assign number i to x
6. $i \leftarrow i + 1$
7. remove x with all its outgoing edges from G

remove node d with all its outgoing edges



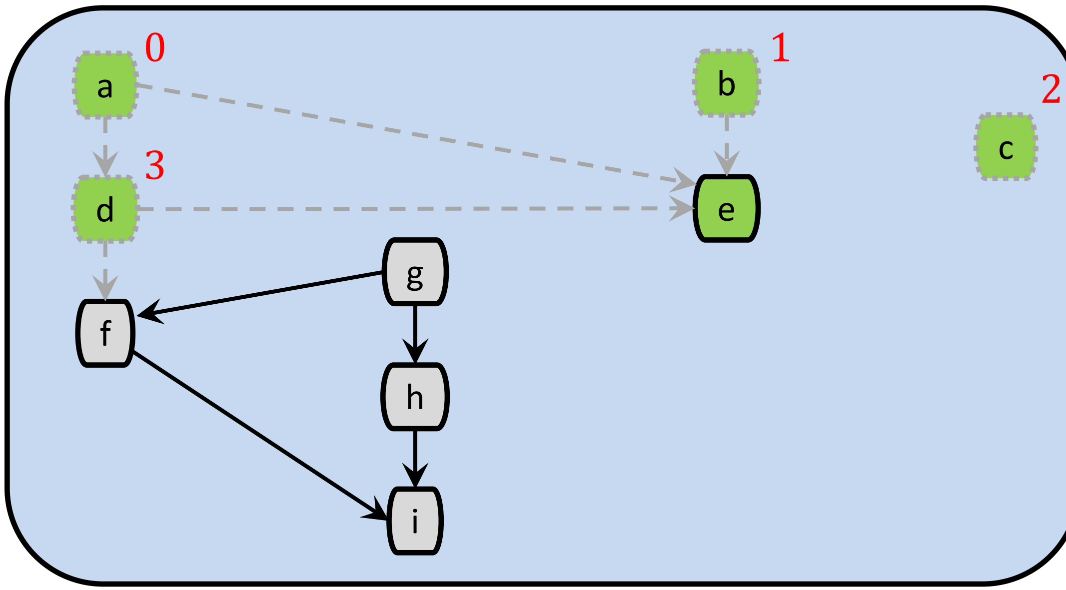


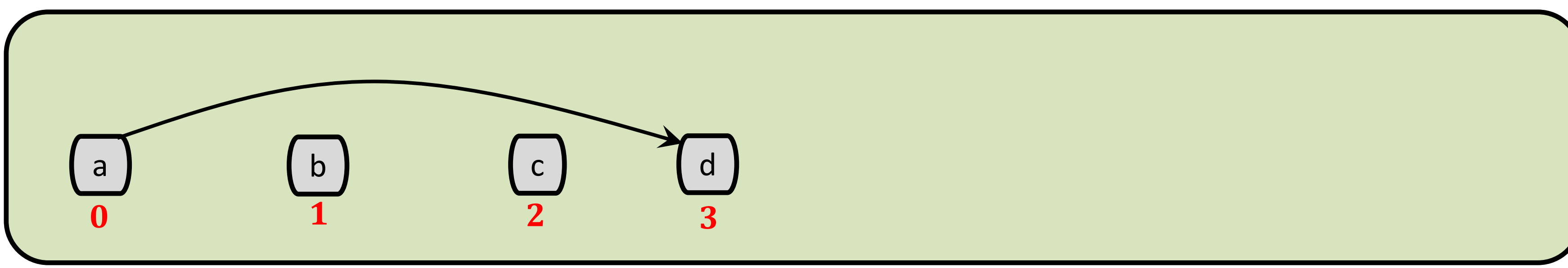
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

```
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

choose node e with no incoming edges





Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

$n \leftarrow |G.V|$

$i \leftarrow 0$

while $i < n$ *do*

  find a node $x \in G.V$ with no incoming edges

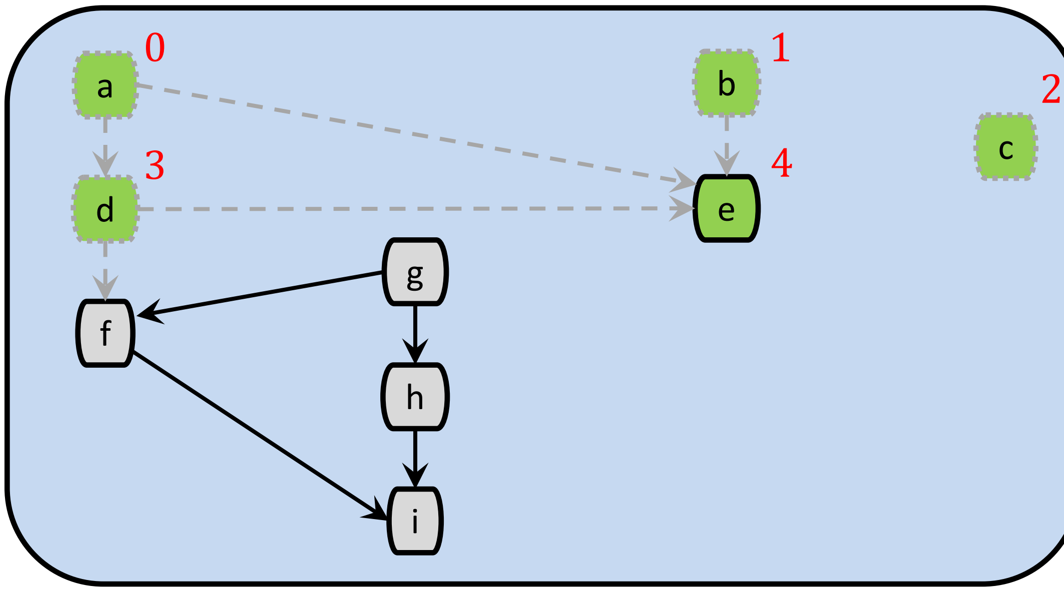
  assign number i to x

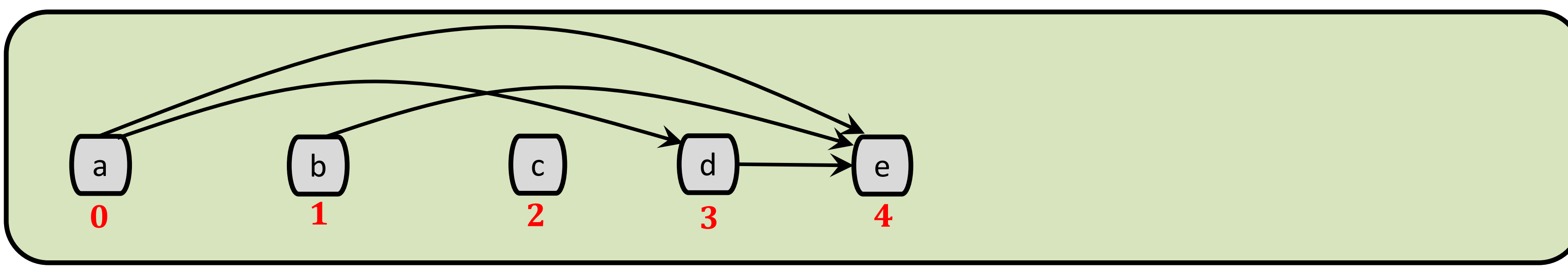
  $i \leftarrow i + 1$

  remove x with all its outgoing edges from G

assign number $i = 4$ to node e

$i \leftarrow 5$ (increment i)



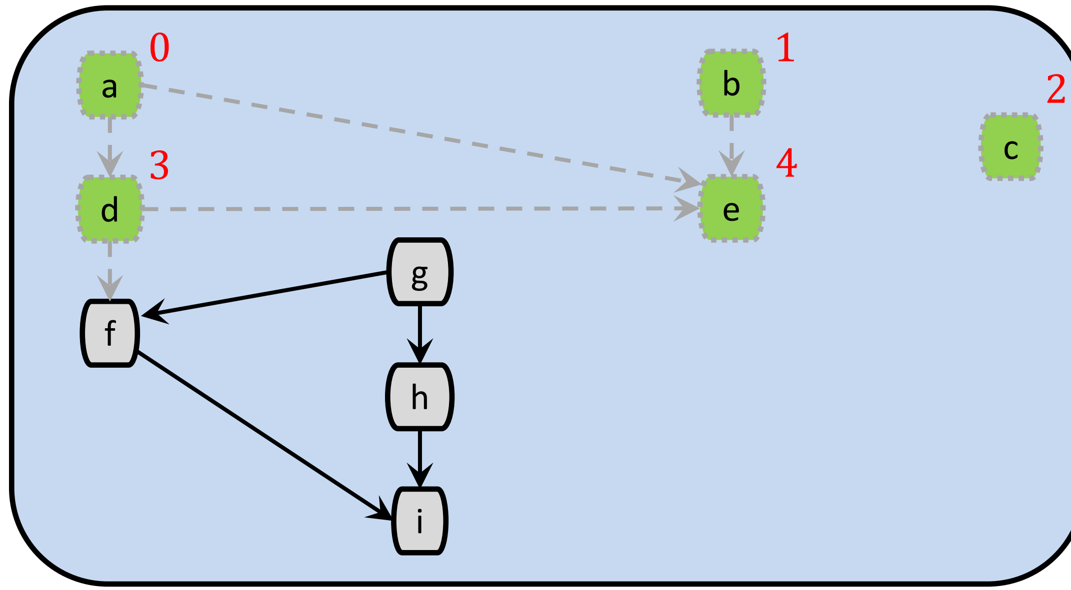


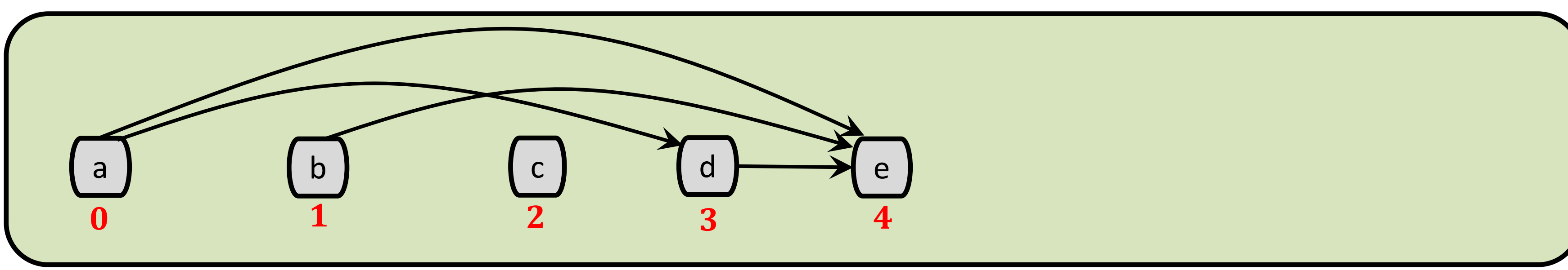
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

```
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

remove node *e* with all its outgoing edges



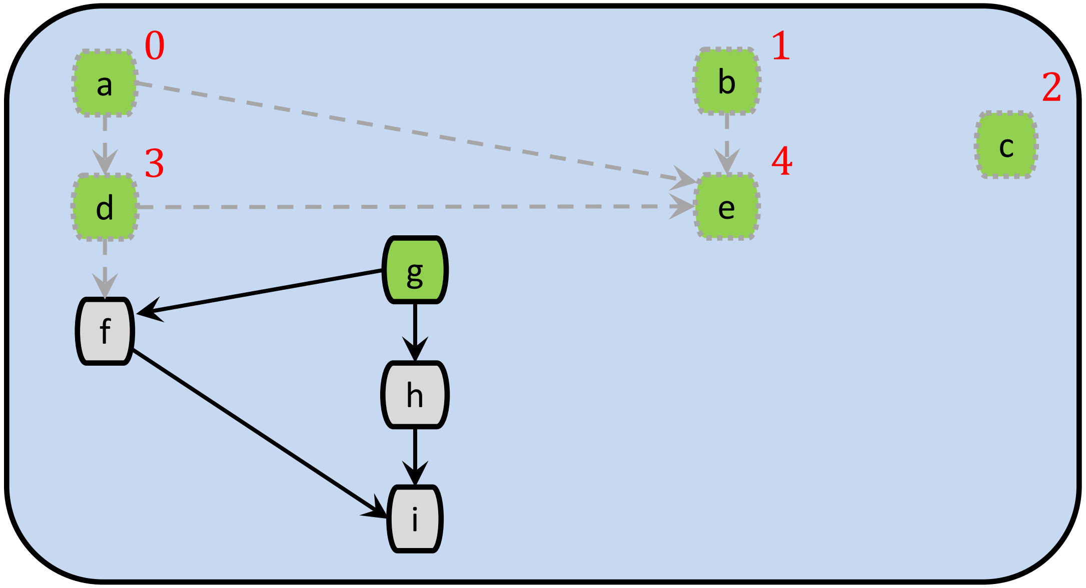


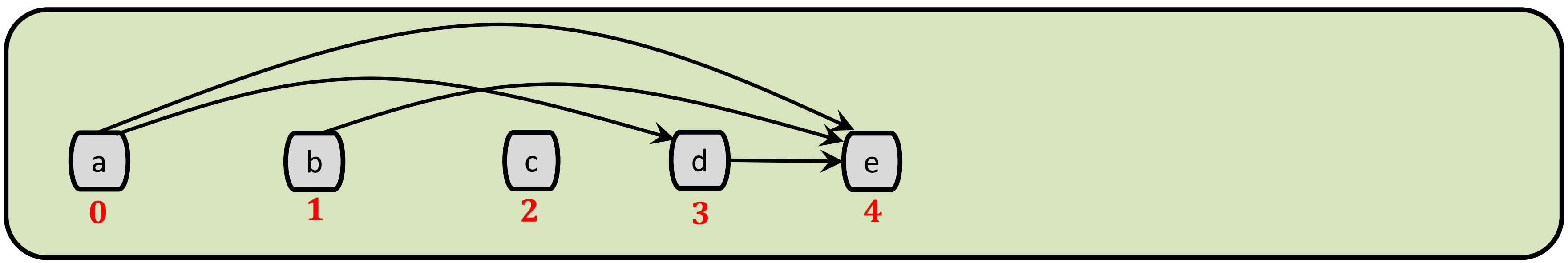
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

1. $n \leftarrow |G.V|$
2. $i \leftarrow 0$
3. *while* $i < n$ *do*
4. find a node $x \in G.V$ with no incoming edges
5. assign number i to x
6. $i \leftarrow i + 1$
7. remove x with all its outgoing edges from G

choose node g with no incoming edges



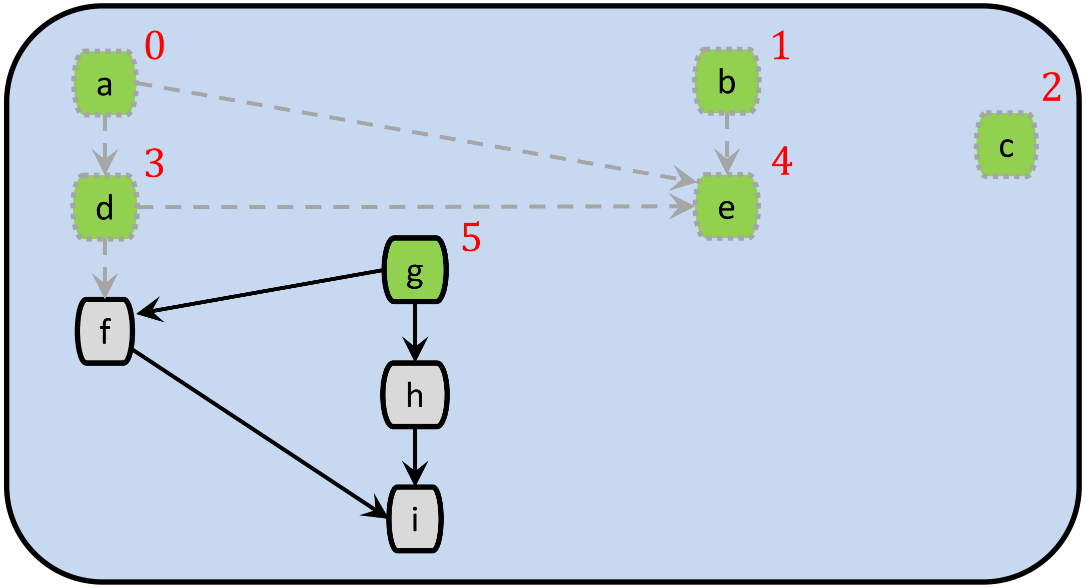


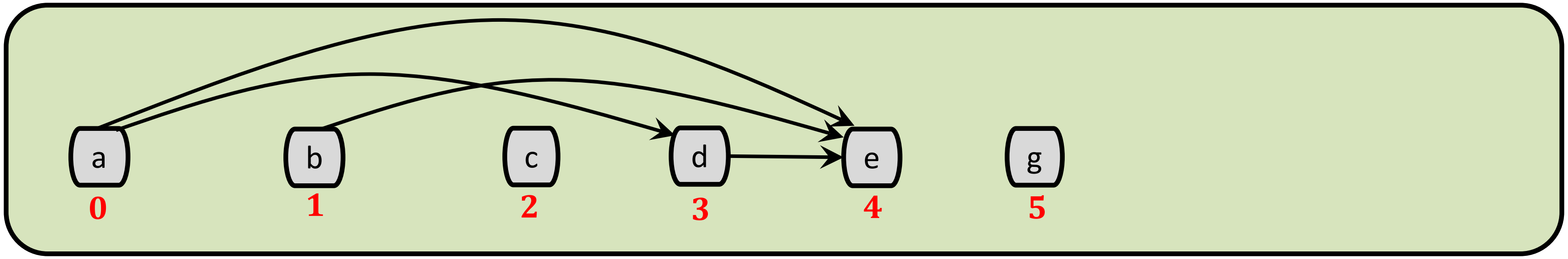
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

1. $n \leftarrow |G.V|$
2. $i \leftarrow 0$
3. *while* $i < n$ *do*
4. find a node $x \in G.V$ with no incoming edges
5. assign number i to x
6. $i \leftarrow i + 1$
7. remove x with all its outgoing edges from G

assign number $i = 5$ to node g
$i \leftarrow 6$ (increment i)



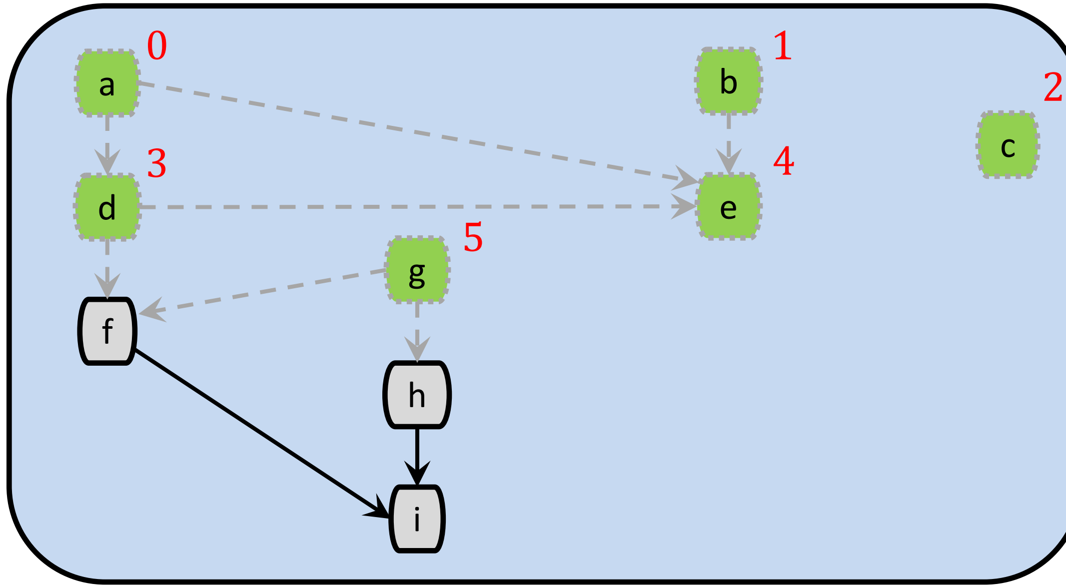


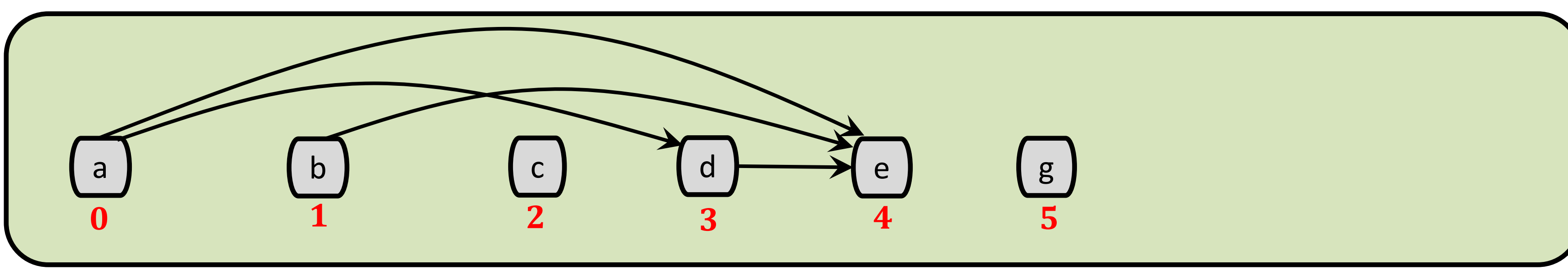
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

$n \leftarrow |G.V|$
$i \leftarrow 0$
while $i < n$ *do*
  find a node $x \in G.V$ with no incoming edges
  assign number i to x
  $i \leftarrow i + 1$
  remove x with all its outgoing edges from G

remove node g with all its outgoing edges



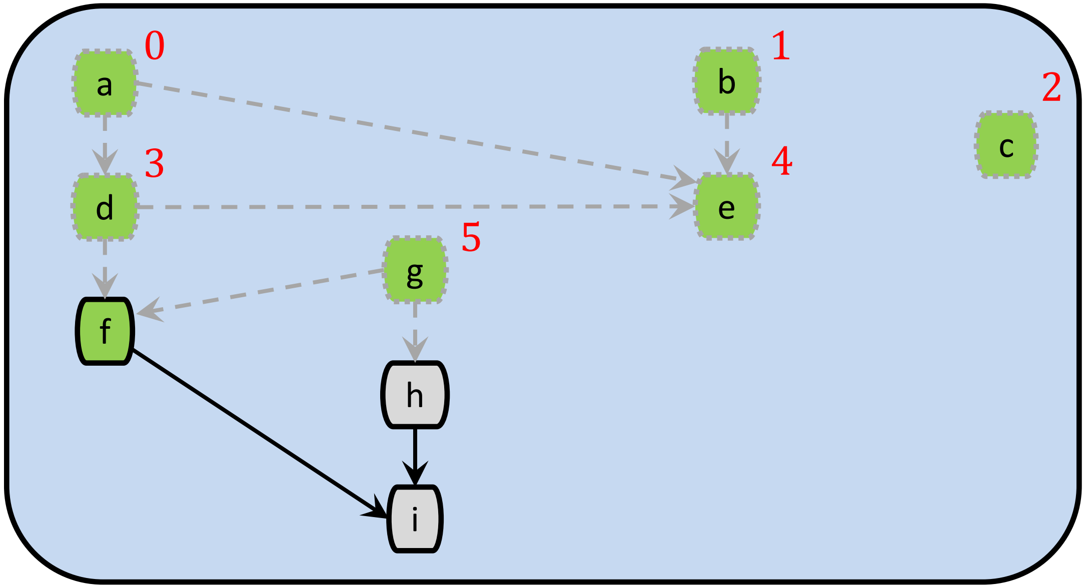


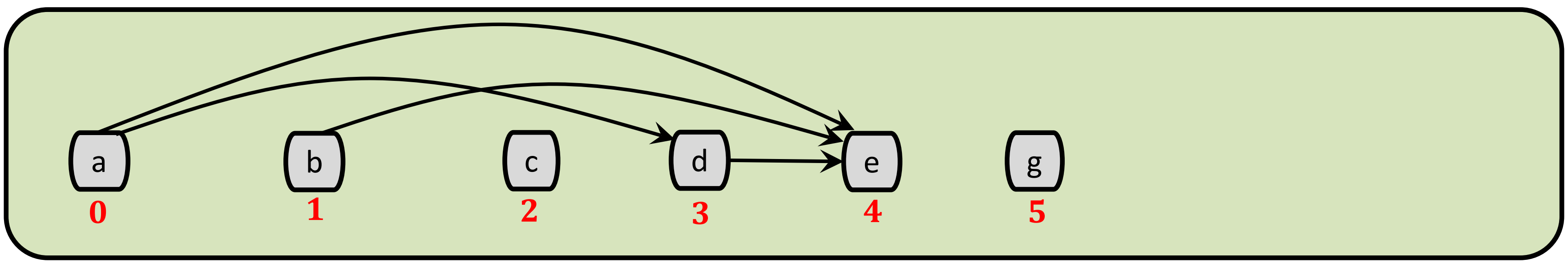
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

1. $n \leftarrow |G.V|$
2. $i \leftarrow 0$
3. *while* $i < n$ *do*
4. find a node $x \in G.V$ with no incoming edges
5. assign number i to x
6. $i \leftarrow i + 1$
7. remove x with all its outgoing edges from G

choose node f with no incoming edges





Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

$n \leftarrow |G.V|$

$i \leftarrow 0$

while $i < n$ *do*

    find a node $x \in G.V$ with no incoming edges

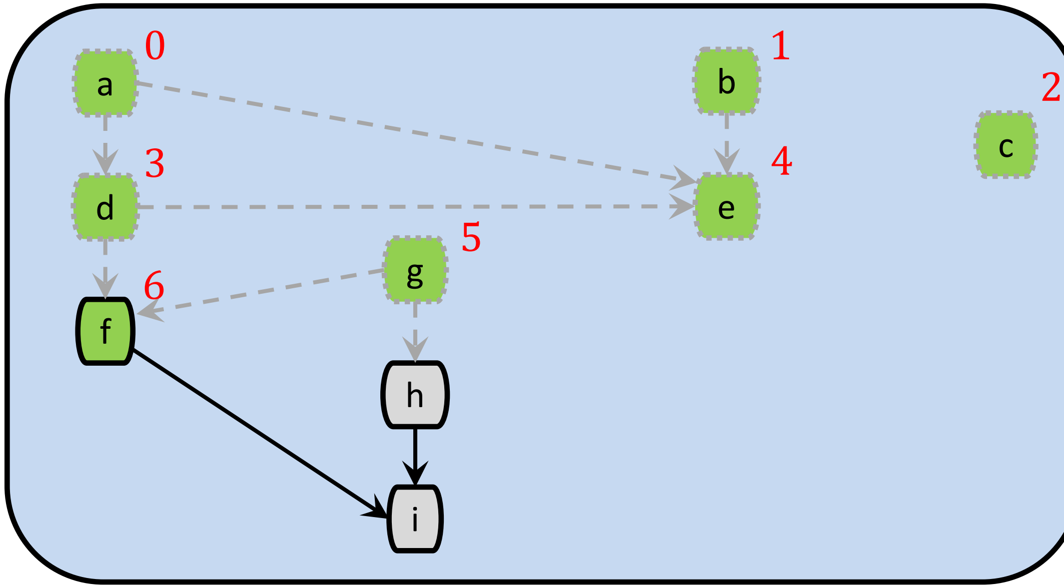
    assign number i to x

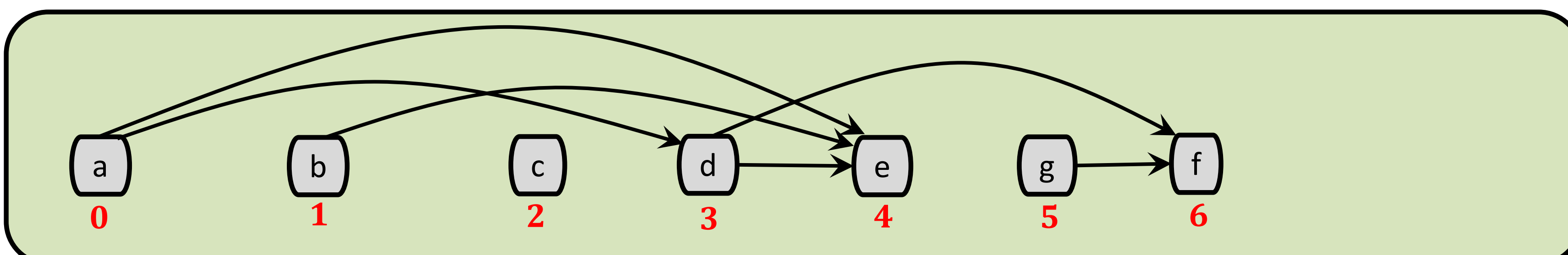
    $i \leftarrow i + 1$

    remove x with all its outgoing edges from G

assign number $i = 6$ to node f

$i \leftarrow 7$ (increment i)

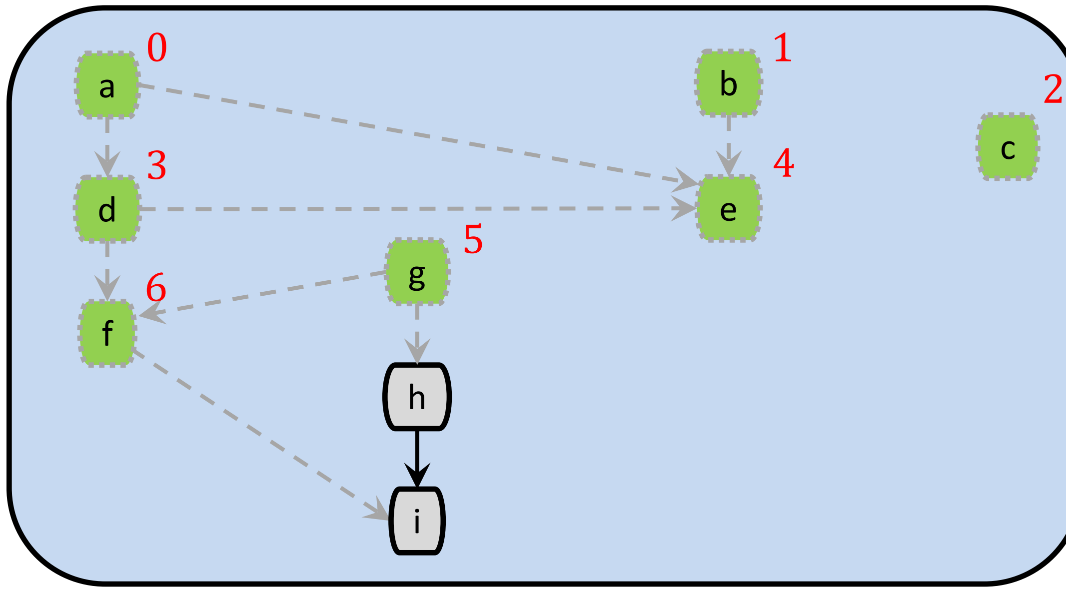


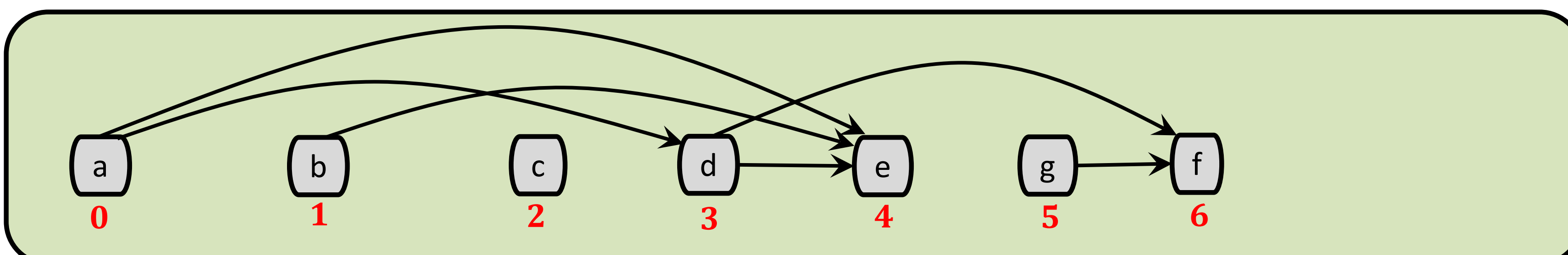


Topological Sort

```
GREEDY-TOPOLOGICAL-SORT ( G )
    n ← |G.V|
    i ← 0
    while i < n do
        find a node x ∈ G.V
        with no incoming edges
        assign number i to x
        i ← i + 1
        remove x with all its
        outgoing edges from G
```

remove node f with all its outgoing edges



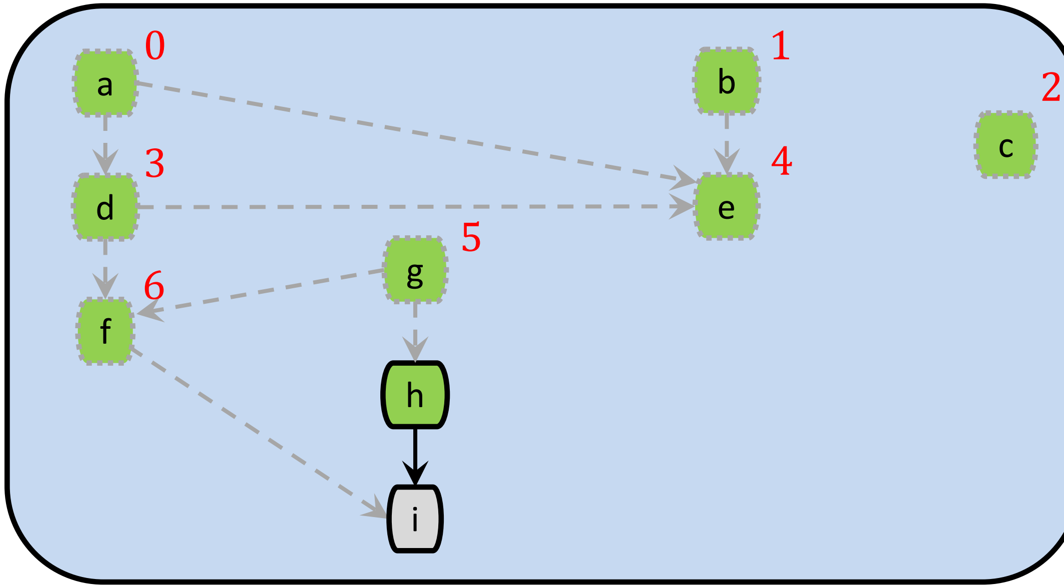


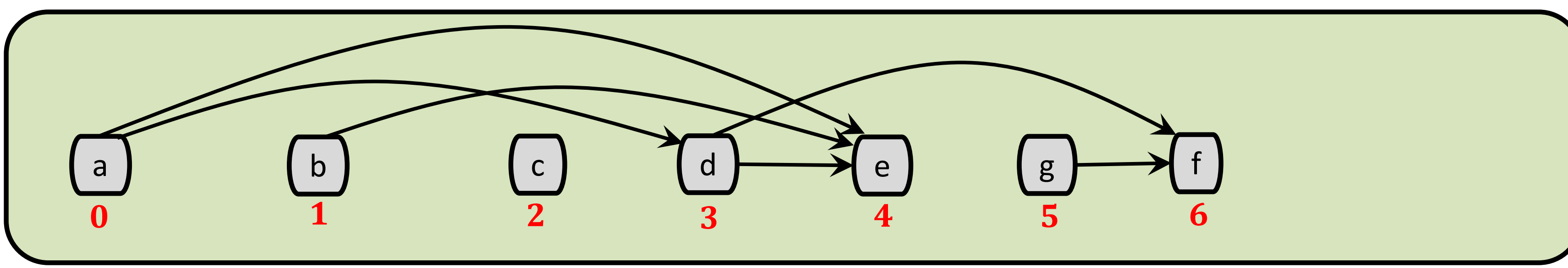
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

1. $n \leftarrow |G.V|$
2. $i \leftarrow 0$
3. *while* $i < n$ *do*
4. find a node $x \in G.V$ with no incoming edges
5. assign number i to x
6. $i \leftarrow i + 1$
7. remove x with all its outgoing edges from G

choose node h with no incoming edges





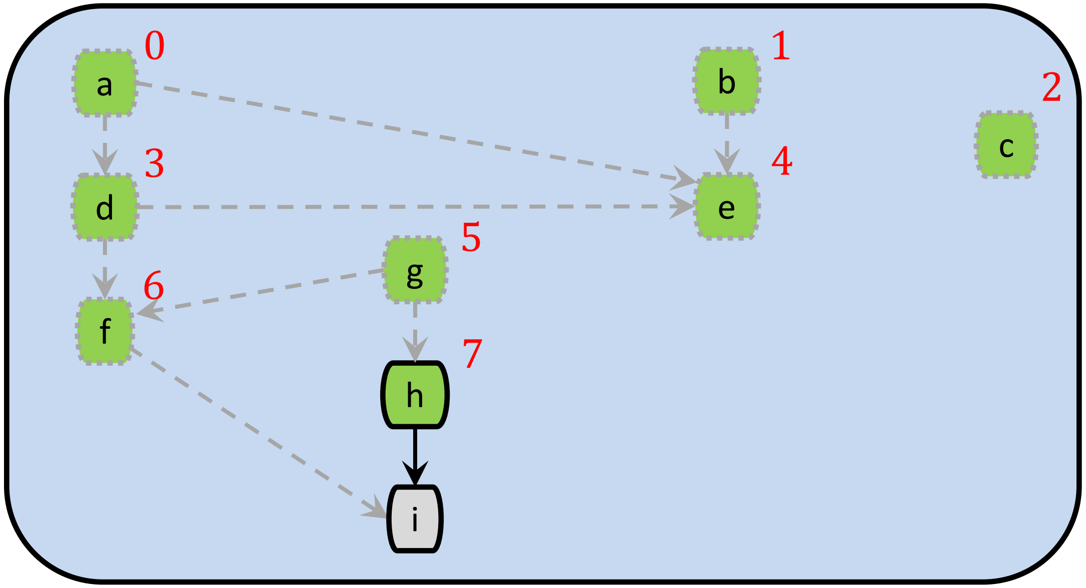
Topological Sort

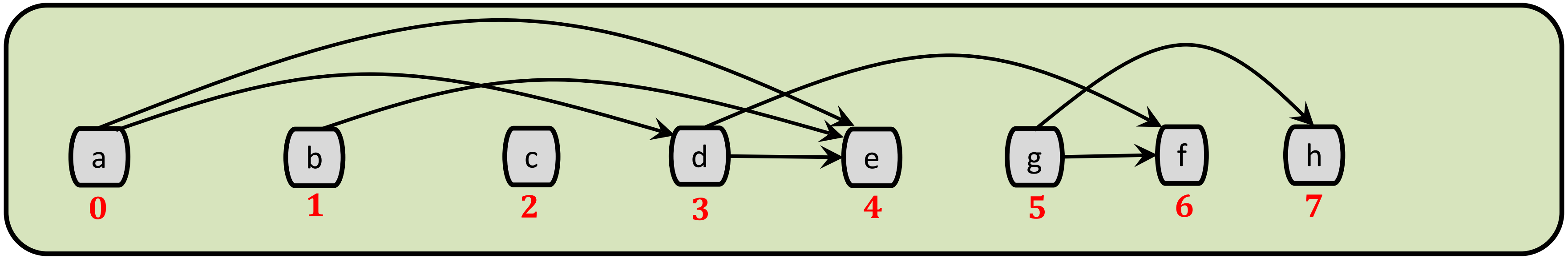
Greedy-Topological-Sort (G)

1. $n \leftarrow |G.V|$
2. $i \leftarrow 0$
3. *while* $i < n$ *do*
4. find a node $x \in G.V$ with no incoming edges
5. assign number i to x
6. $i \leftarrow i + 1$
7. remove x with all its outgoing edges from G

assign number $i = 7$ to node h

$i \leftarrow 8$ (increment i)

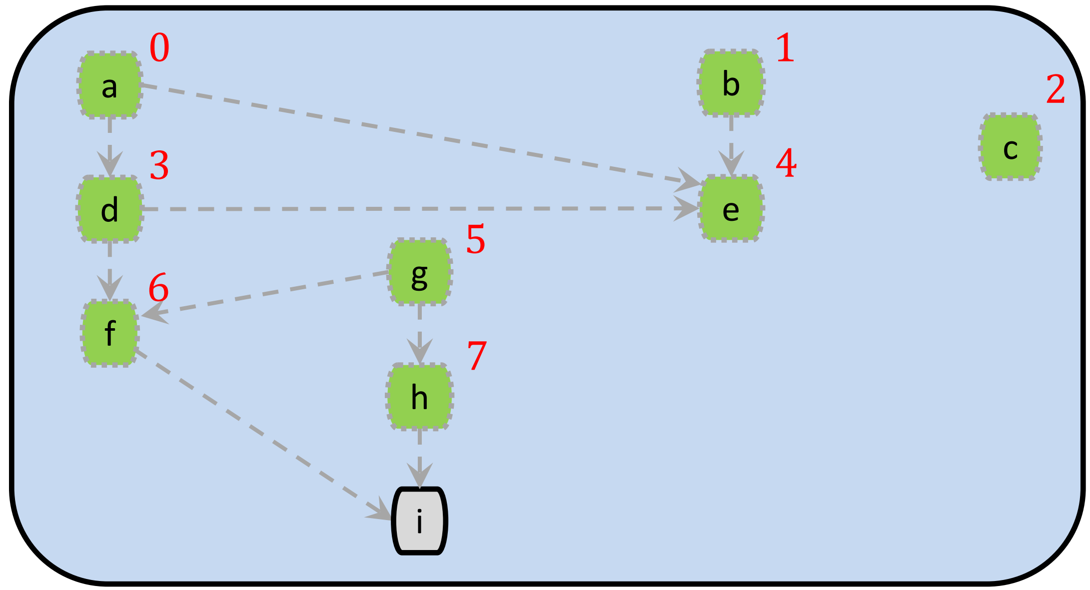


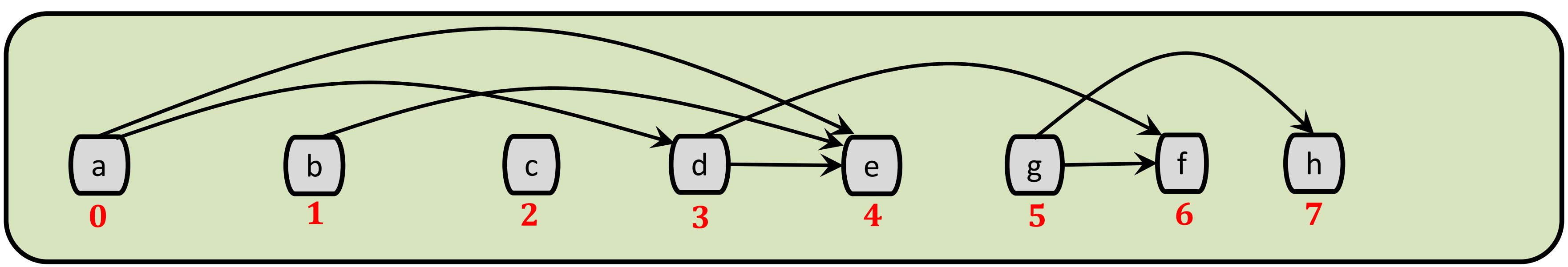


Topological Sort

```
GREEDY-TOPOLOGICAL-SORT ( G )
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

remove node *h* with all its outgoing edges

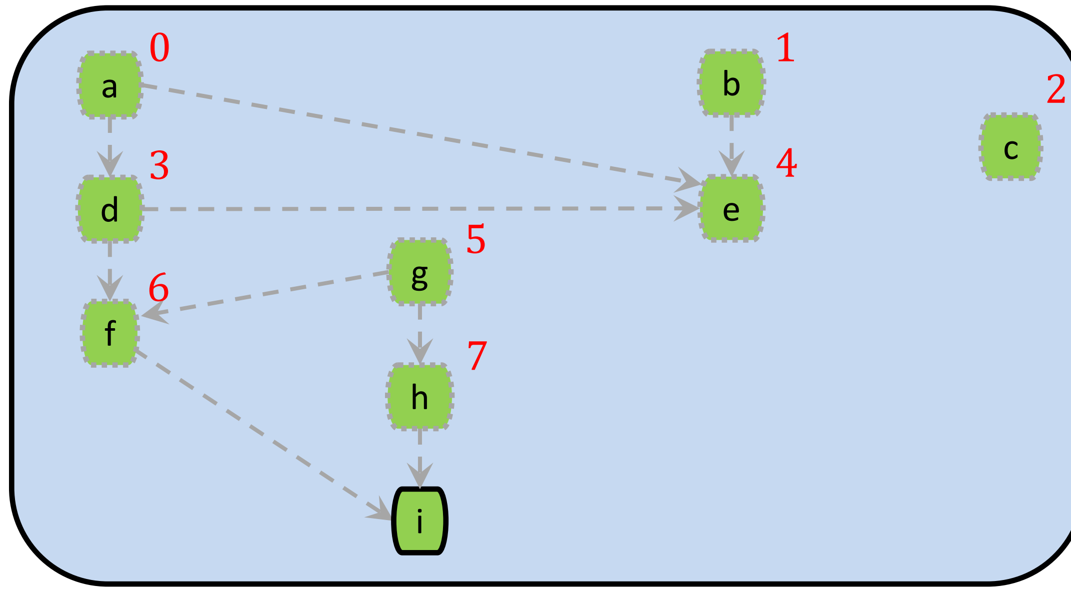


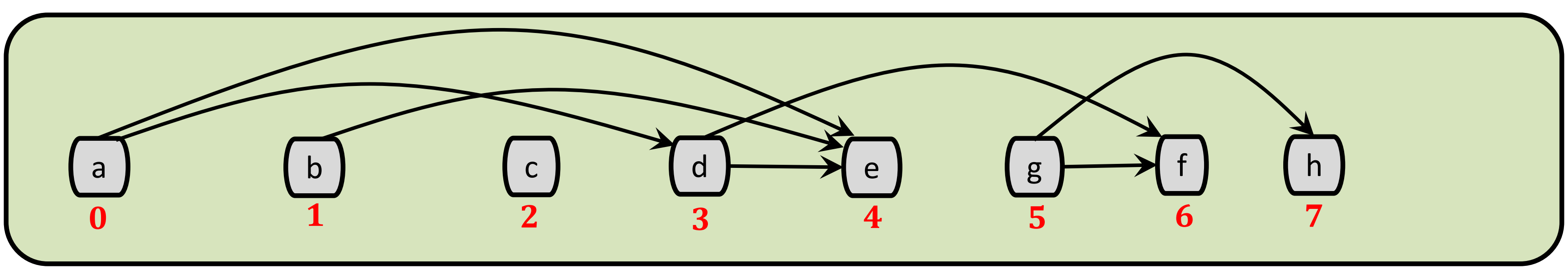


Topological Sort

```
GREEDY-TOPOLOGICAL-SORT ( G )
    n ← |G.V|
    i ← 0
    while i < n do
        find a node x ∈ G.V
        with no incoming edges
        assign number i to x
        i ← i + 1
        remove x with all its
        outgoing edges from G
```

choose node *i* with no incoming edges



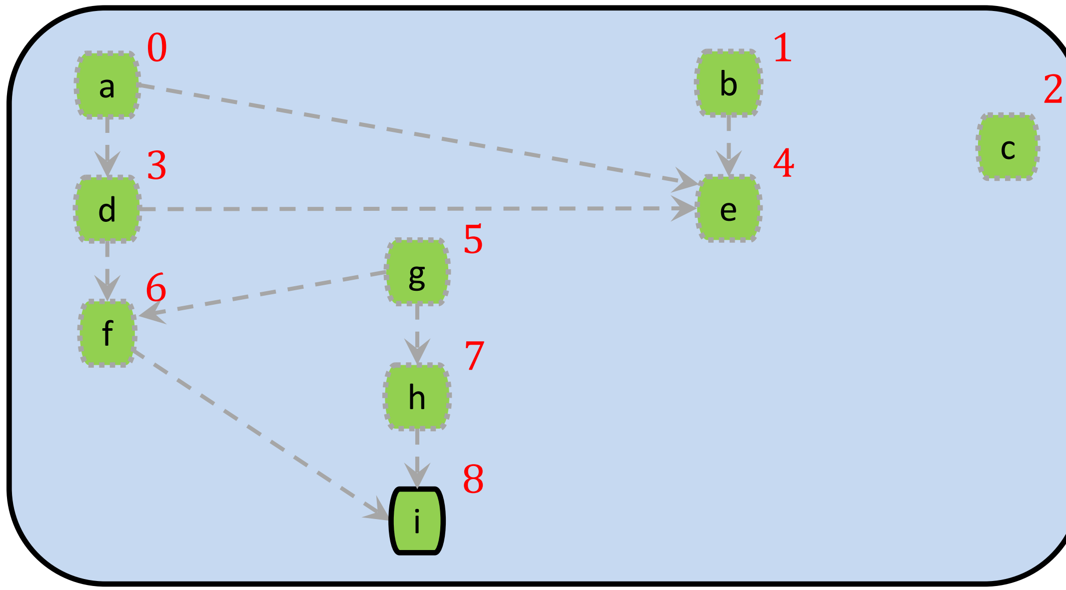


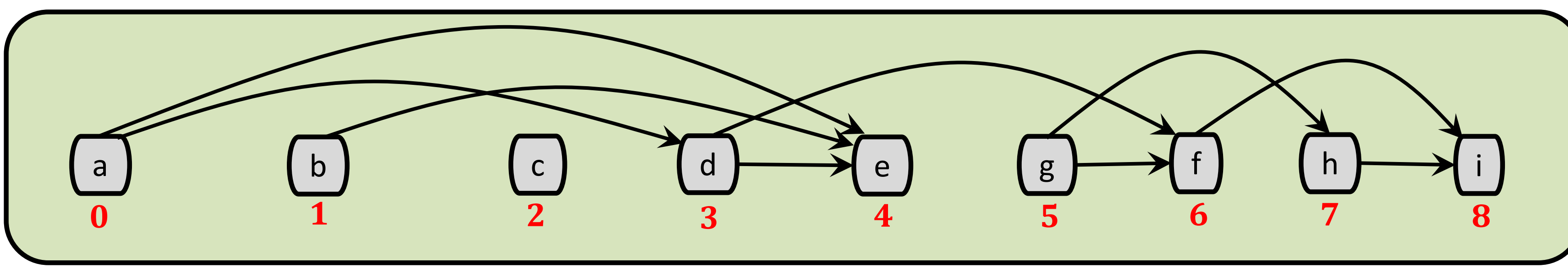
Topological Sort

```
GREEDY-TOPOLOGICAL-SORT ( G )
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

assign number $i = 8$ to node i

$i \leftarrow 9$ (increment i)



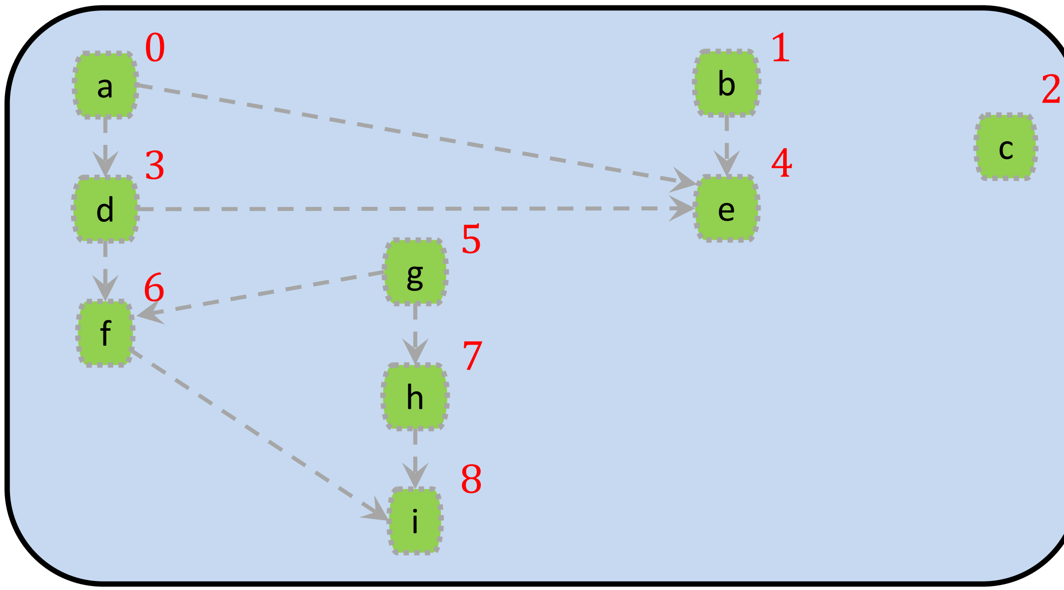


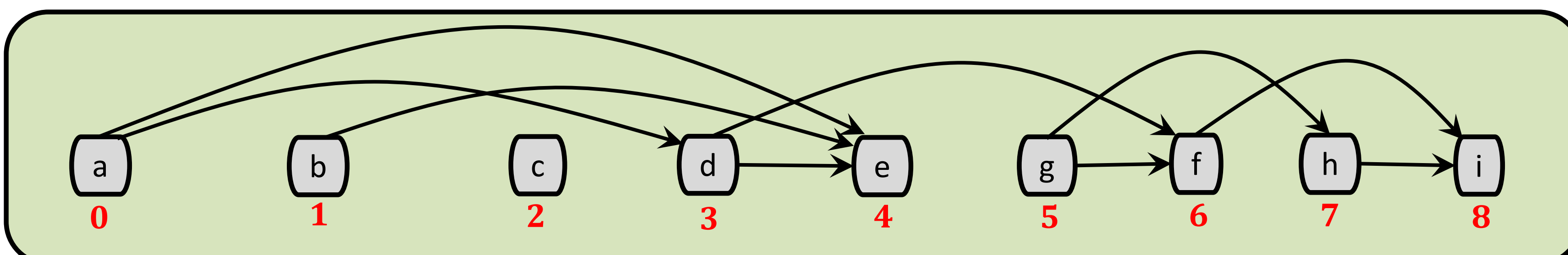
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

$n \leftarrow |G.V|$
$i \leftarrow 0$
while $i < n$ *do*
    find a node $x \in G.V$ with no incoming edges
    assign number i to x
    $i \leftarrow i + 1$
    remove x with all its outgoing edges from G

remove node i with all its outgoing edges



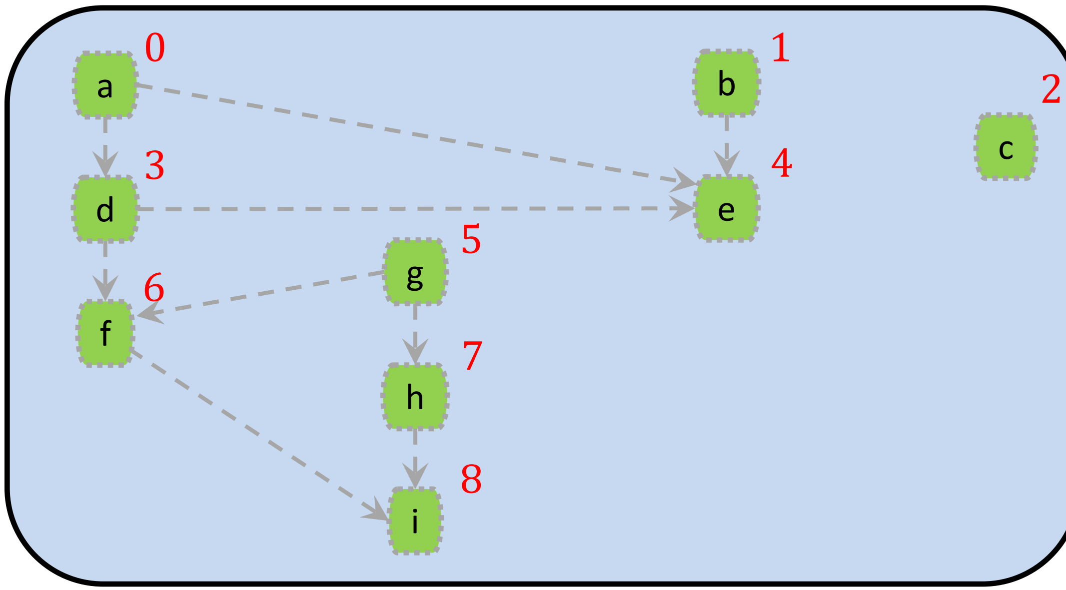


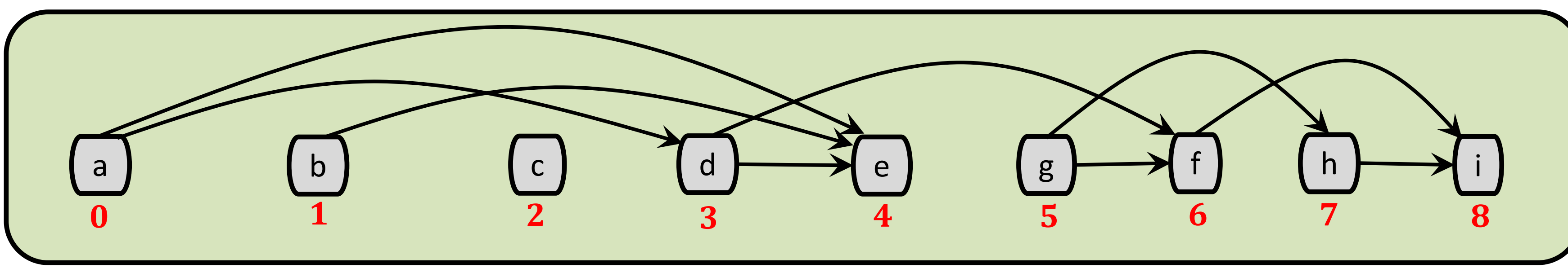
Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

```
n ← |G.V|
i ← 0
while i < n do
    find a node x ∈ G.V
    with no incoming edges
    assign number i to x
    i ← i + 1
    remove x with all its
    outgoing edges from G
```

Done!





Topological Sort

GREEDY-TOPOLOGICAL-SORT (G)

1. $n \leftarrow |G.V|$
2. $i \leftarrow 0$
3. *while* $i < n$ *do*
4. find a node $x \in G.V$ with no incoming edges
5. assign number i to x
6. $i \leftarrow i + 1$
7. remove x with all its outgoing edges from G

GREEDY-TOPOLOGICAL-SORT (G)

1. $n \leftarrow |G.V|$
2. *for* each $v \in G.V$ *do* $d[v] \leftarrow 0$
3. *for* each $(u, v) \in G.E$ *do* $d[v] \leftarrow d[v] + 1$
4. *queue* $Q \leftarrow \emptyset$
5. *for* each $v \in G.V$ *do*
6. *if* $d[v] = 0$ *then* $Q.Enqueue(v)$
7. $i \leftarrow 0$
8. *while* $i < n$ *do*
9. $x \leftarrow Q.Dequeue(\ )$
10. assign number i to x
11. $i \leftarrow i + 1$
12. *for* each $(x, v) \in G.E$ *do*
13. $d[v] \leftarrow d[v] - 1$
14. *if* $d[v] = 0$ *then* $Q.Enqueue(v)$

Let $n = |G.V|$ and $m = |G.E|$.

Then the running time of the algorithm is $O(n + m)$.

The Minimum Spanning Tree (MST) Problem

We are given a weighted connected undirected graph $G = (V, E)$ with vertex set V and edge set E, and a weight function w such that for each edge $(u, v) \in E$, $w(u, v)$ represents its weight.

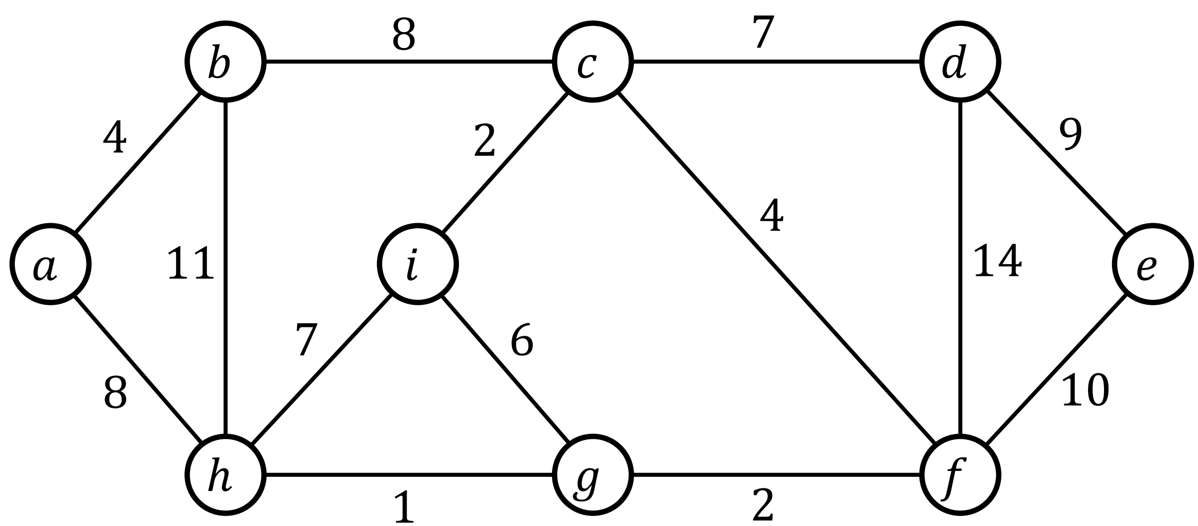
An acyclic subset $T \subseteq E$ that connects all vertices of V must form a tree, which we call a ***spanning tree*** since it "spans" the graph G.

A spanning tree of G can be found easily in $O(n + m)$ time, where $n = |V|$ and $m = |E|$, using a *breadth-first search* (BFS) or a *depth-first search* (DFS).

The ***minimum-spanning-tree (MST) problem*** asks us to find a spanning tree T whose total weight $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized.

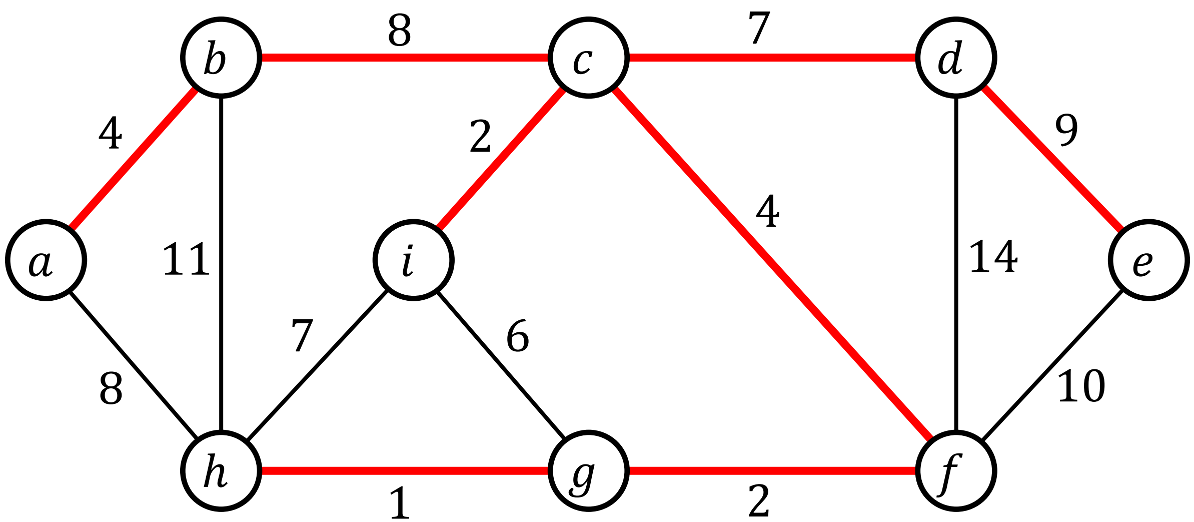
The Minimum Spanning Tree (MST) Problem

A weighted undirected graph



The Minimum Spanning Tree (MST) Problem

A weighted undirected graph



Its MST (in red) of total weight 37

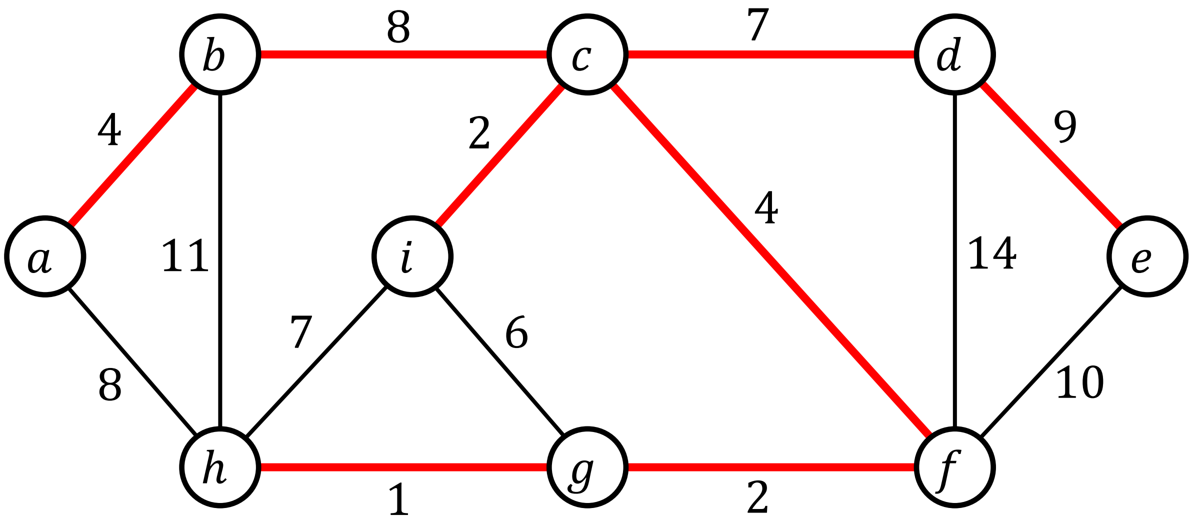
MST: Greedy Strategy for Growing an MST

We are given a weighted connected undirected graph $G = (V, E)$ with vertex set V and edge set E, and a weight function w such that for each edge $(u, v) \in E$, $w(u, v)$ represents its weight.

Suppose set $A \subset E$ is a subset of some MST of G.

Now if edge $(u, v) \in E$ but edge $(u, v) \notin A$, we call (u, v) a ***safe edge*** provided $A \cup \{u, v\}$ is also a subset of an MST of G.

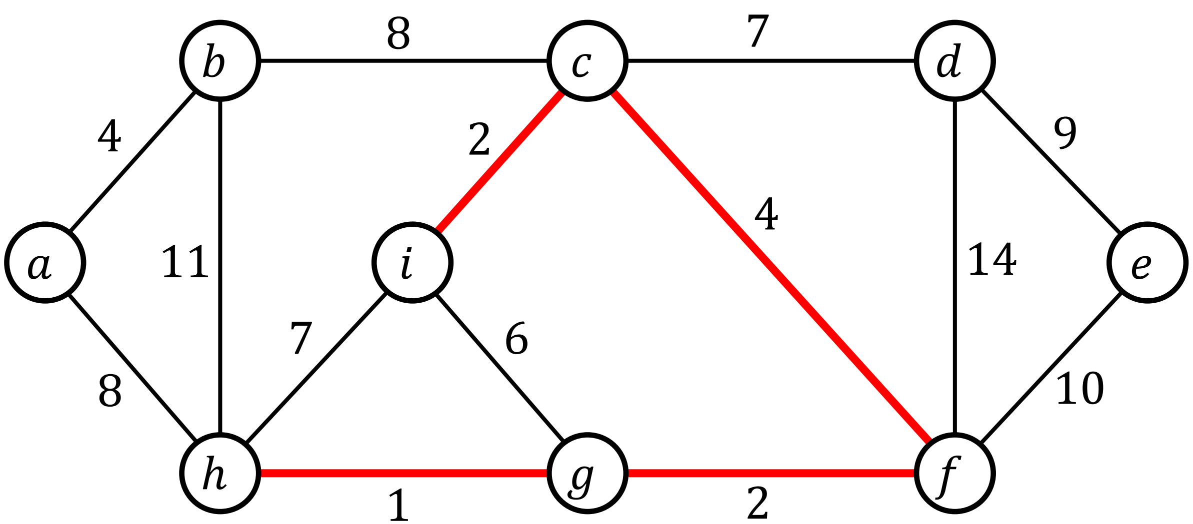
MST: Greedy Strategy for Growing an MST



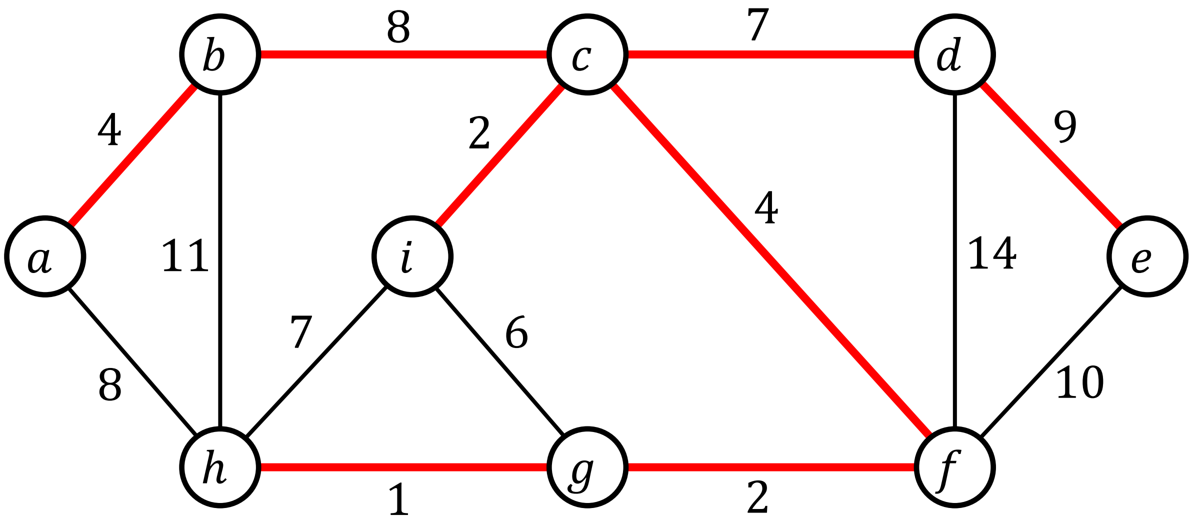
Red edges form an MST.
Let's call it T

Let $A = \{(i,c),(c,f),(f,g),(g,h)\}$

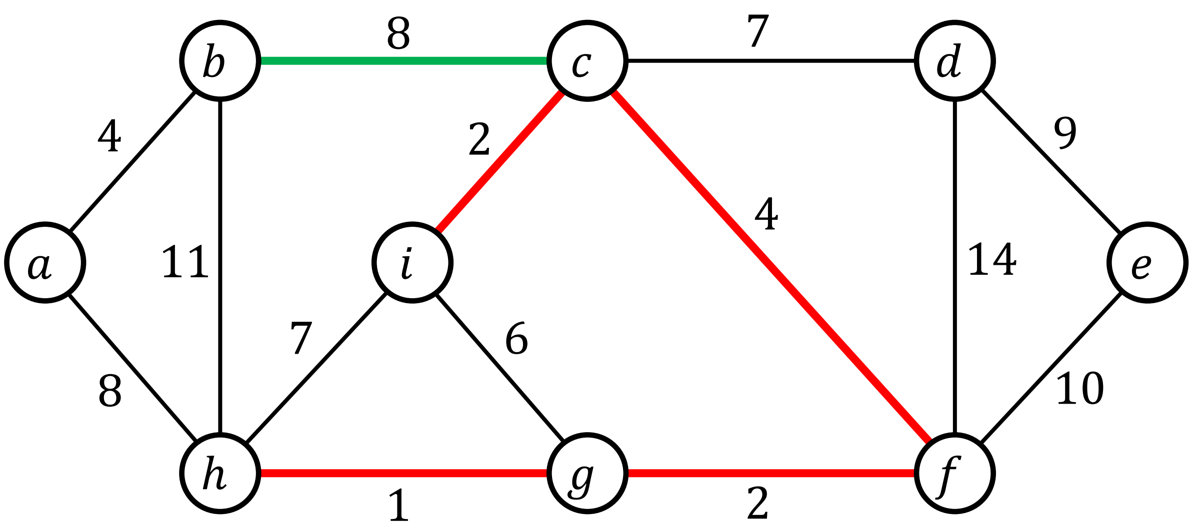
Clearly, $A \subset T$.



MST: Greedy Strategy for Growing an MST



Red edges form an MST.
Let's call it T

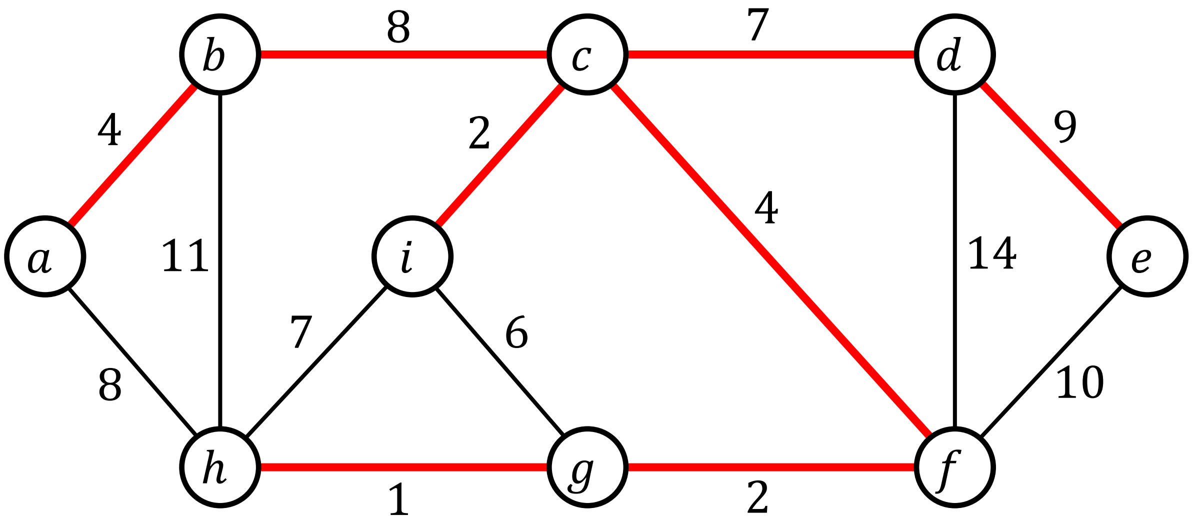


Let $A = \{(i,c),(c,f),(f,g),(g,h)\}$

Clearly, $A \subset T$.

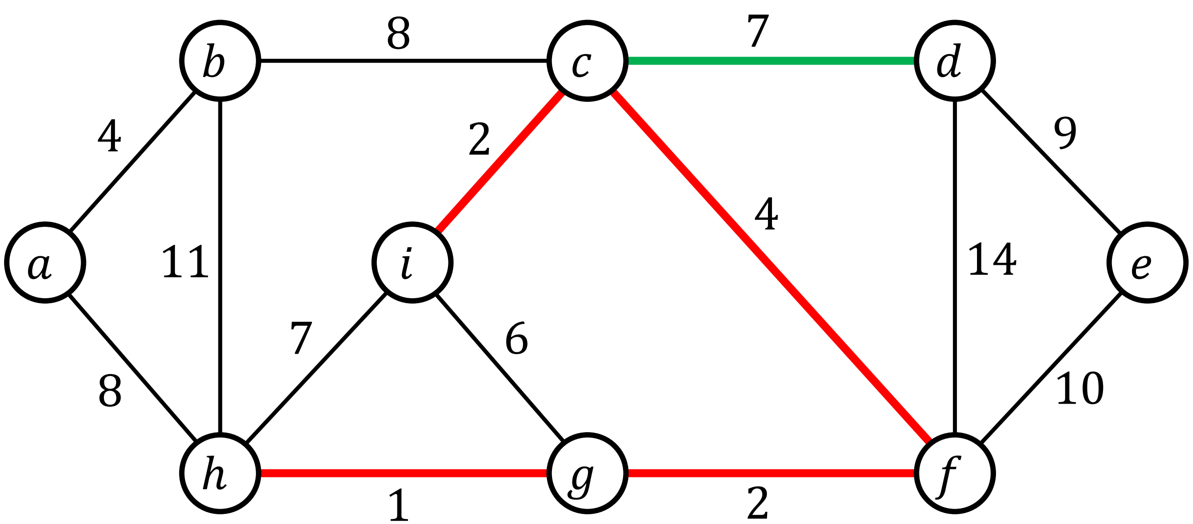
Edge (b,c) is safe because $A \cup \{(b,c)\} \subseteq T$.

MST: Greedy Strategy for Growing an MST



Red edges form an MST.
Let's call it T

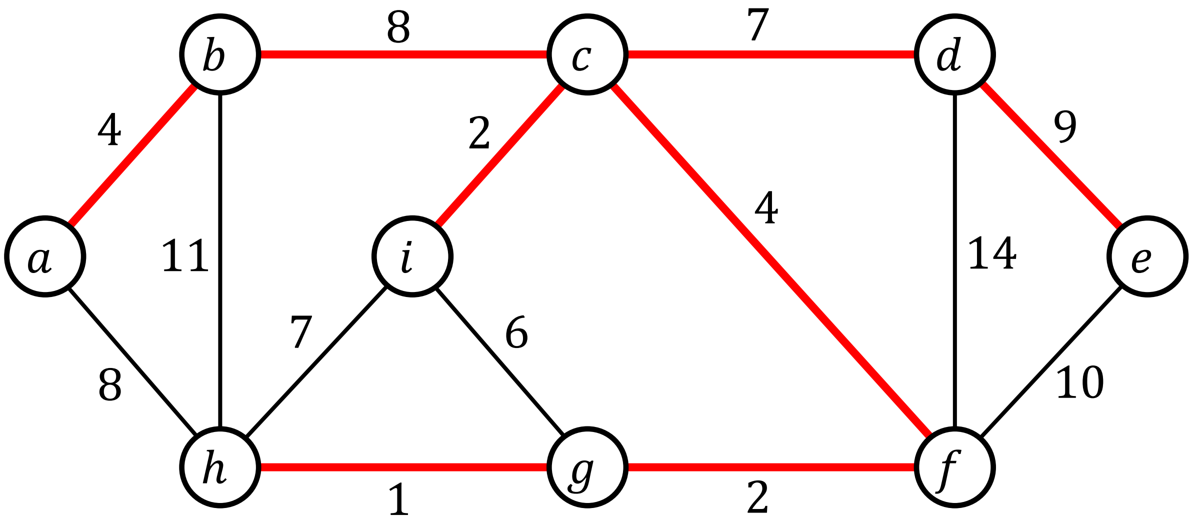
Let $A = \{(i, c), (c, f), (f, g), (g, h)\}$



Clearly, $A \subset T$.

Edge (c, d) is safe because $A \cup \{(c, d)\} \subseteq T$.

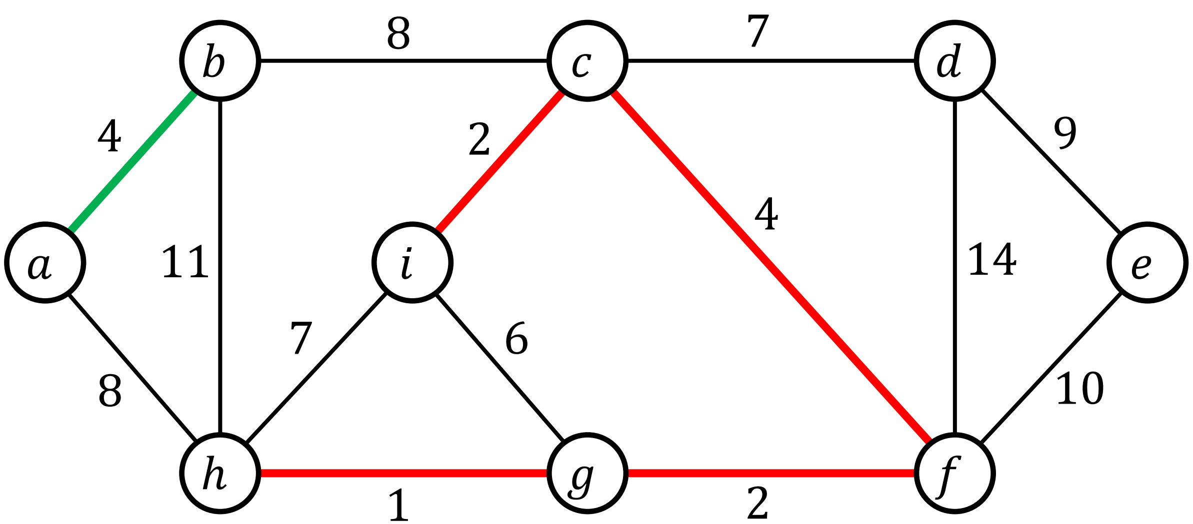
MST: Greedy Strategy for Growing an MST



Red edges form an MST.
Let's call it T

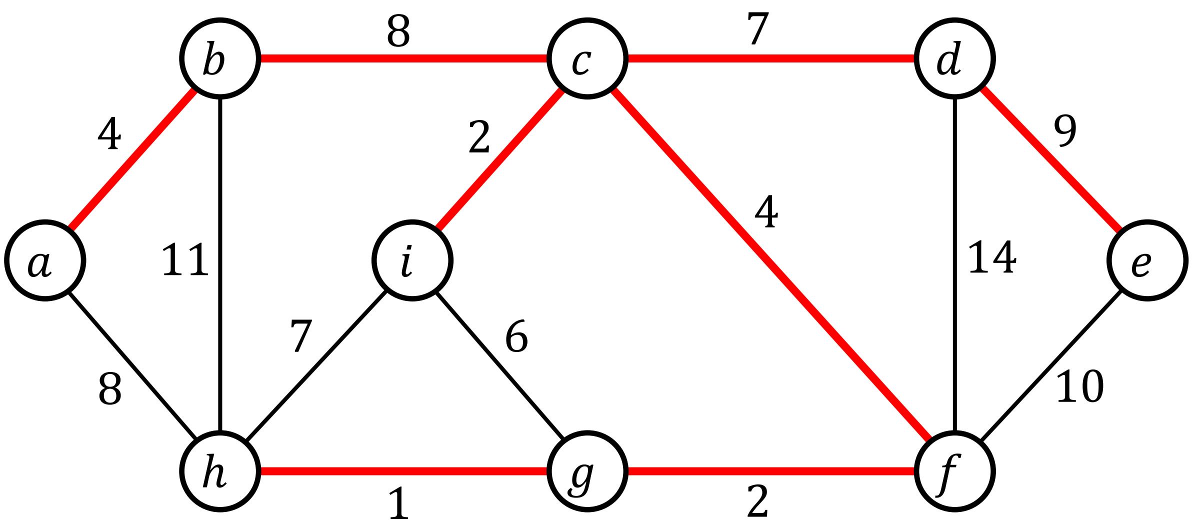
Let $A = \{(i, c), (c, f), (f, g), (g, h)\}$

Clearly, $A \subset T$.



Edge (a, b) is safe because $A \cup \{(a, b)\} \subseteq T$.

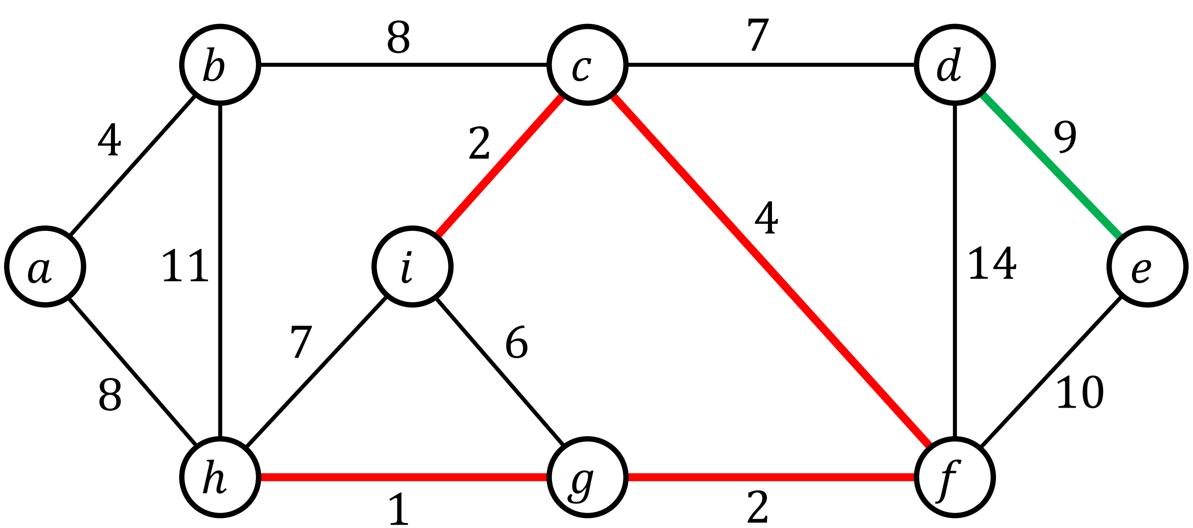
MST: Greedy Strategy for Growing an MST



Red edges form an MST.
Let's call it T

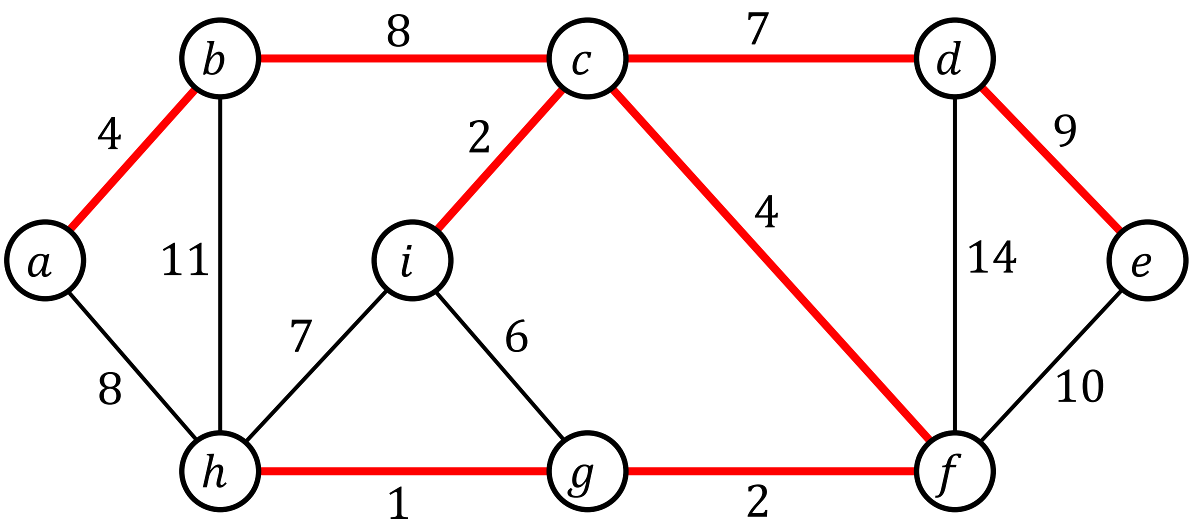
Let $A = \{(i, c), (c, f), (f, g), (g, h)\}$

Clearly, $A \subset T$.



Edge (d, e) is safe because $A \cup \{(d, e)\} \subseteq T$.

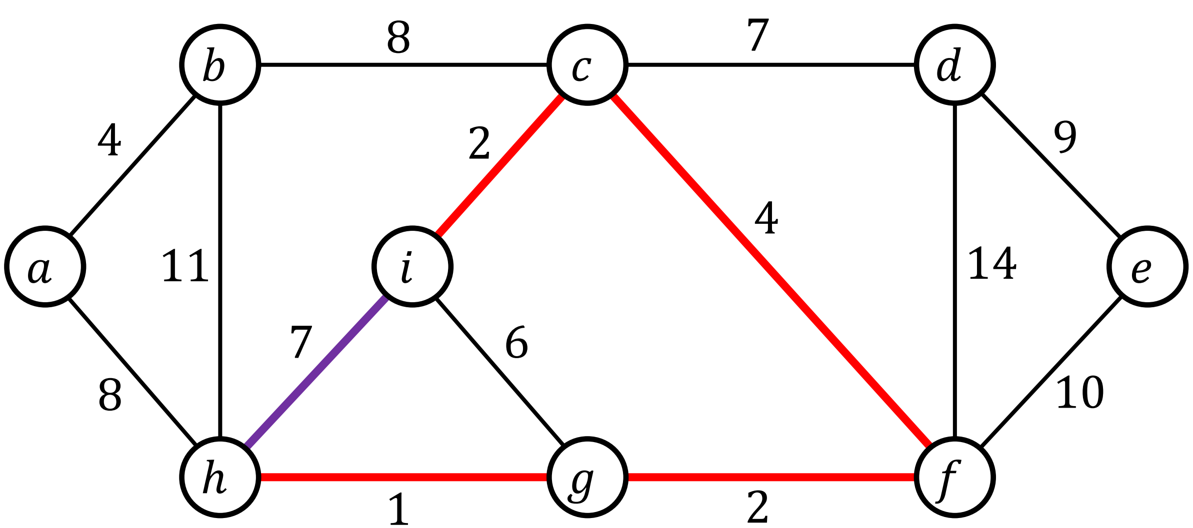
MST: Greedy Strategy for Growing an MST



Red edges form an MST.
Let's call it T

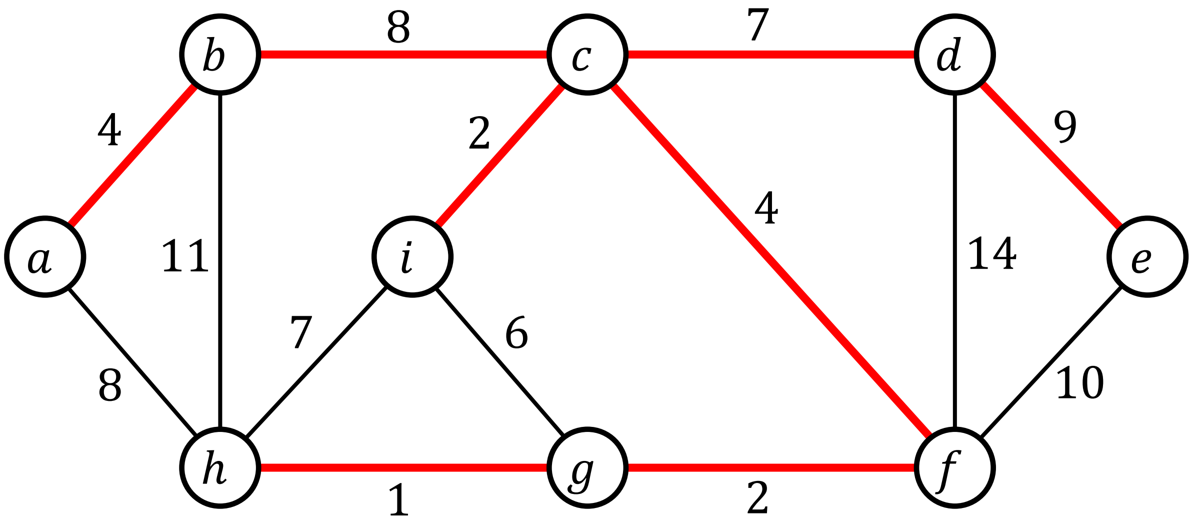
Let $A = \{(i,c),(c,f),(f,g),(g,h)\}$

Clearly, $A \subset T$.

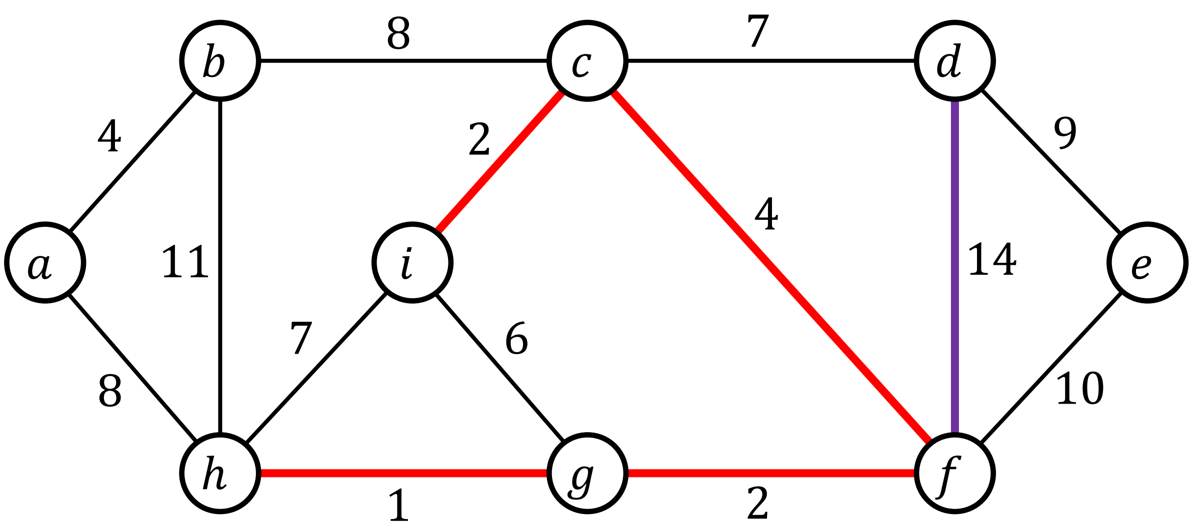


Edge (h,i) is NOT safe
because $A \cup \{(h,i)\}$
Is NOT part of any MST
of the given graph.

MST: Greedy Strategy for Growing an MST



Red edges form an MST.
Let's call it T

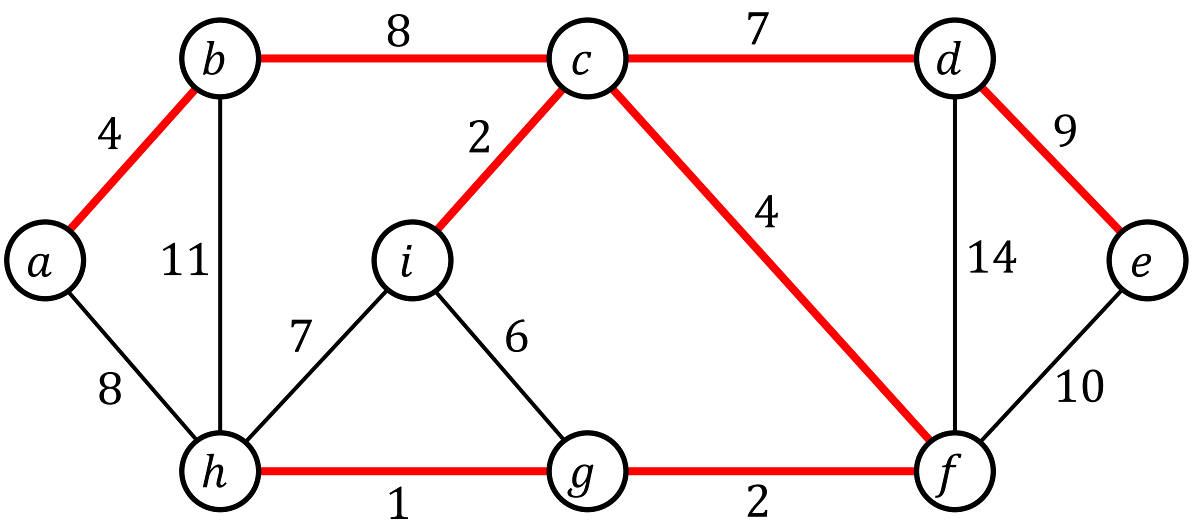


Let $A = \{(i,c),(c,f),(f,g),(g,h)\}$

Clearly, $A \subset T$.

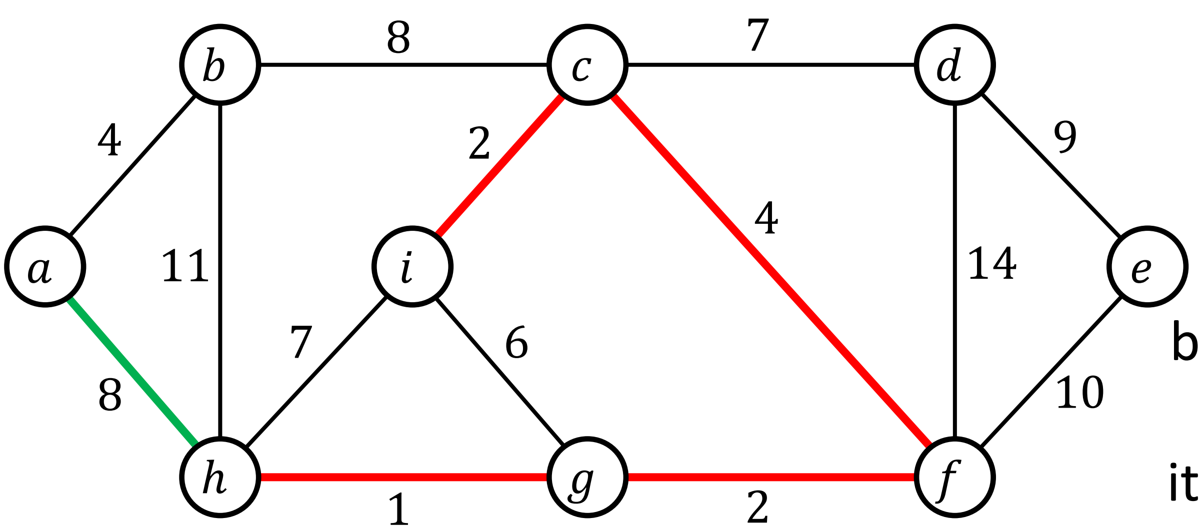
Edge (d,f) is NOT safe because $A \cup \{(d,f)\}$ Is NOT part of any MST of the given graph.

MST: Greedy Strategy for Growing an MST



Red edges form an MST.
Let's call it T

Let $A = \{(i, c), (c, f), (f, g), (g, h)\}$



Clearly, $A \subset T$.

Edge (a, h) is safe
because though $A \cup \{(a, h)\}$
is not a subset of T,
it is a subset of another MST.

MST: Greedy Strategy for Growing an MST

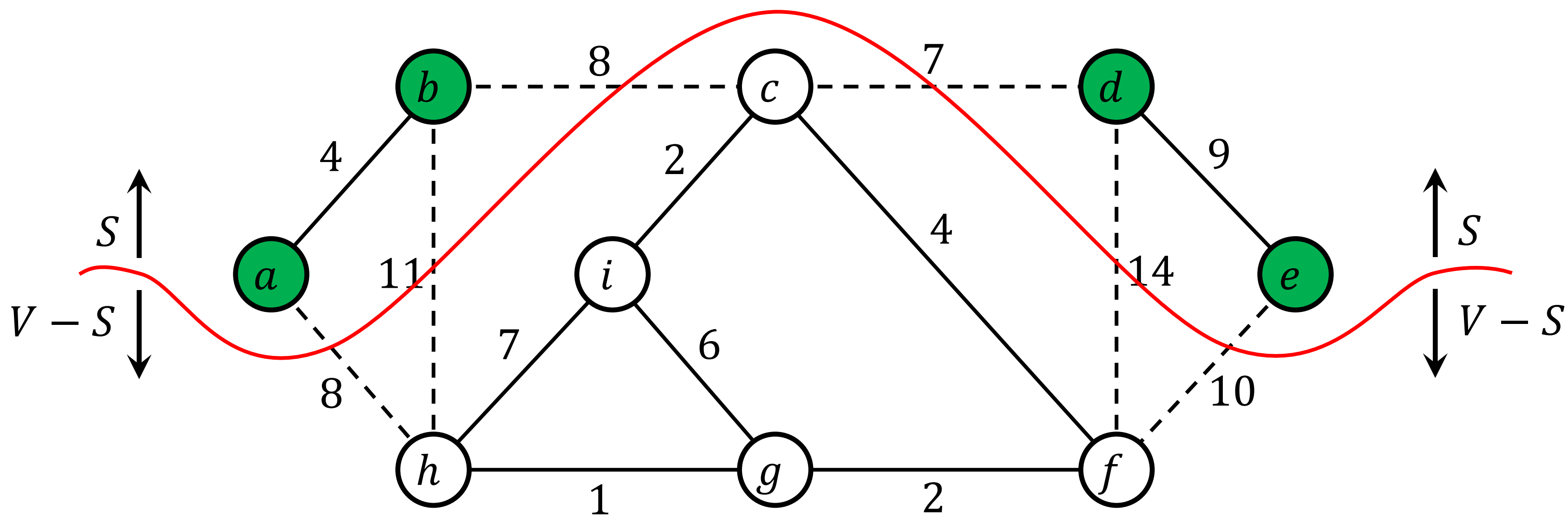
Generic-MST ($G = (V, E)$, w)

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

MST: Finding Safe Edges

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

We say that an edge $(u, v) \in E$ ***crosses*** the cut $(S, V \setminus S)$ if one of its endpoints is in S and the other is in $V \setminus S$.



Green vertices belong to set S, i.e., $S = \{a, b, d, e\}$.

White vertices belong to set $V - S$, i.e., $V - S = \{c, f, g, h, i\}$.

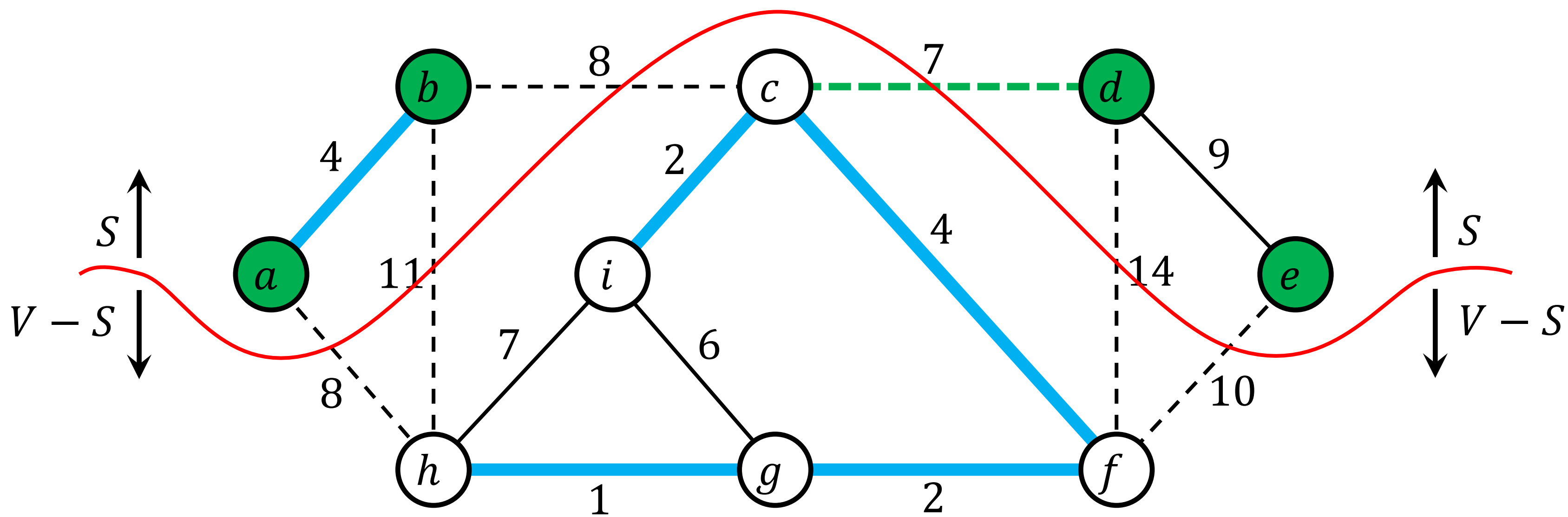
The red line represent the cut $(S, V - S)$.

Dotted edges are the cut edges, i.e., they cross the red line.

MST: Finding Safe Edges

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. Multiple light edges can cross a cut.



Let the blue thick edges form the set A, i.e.,

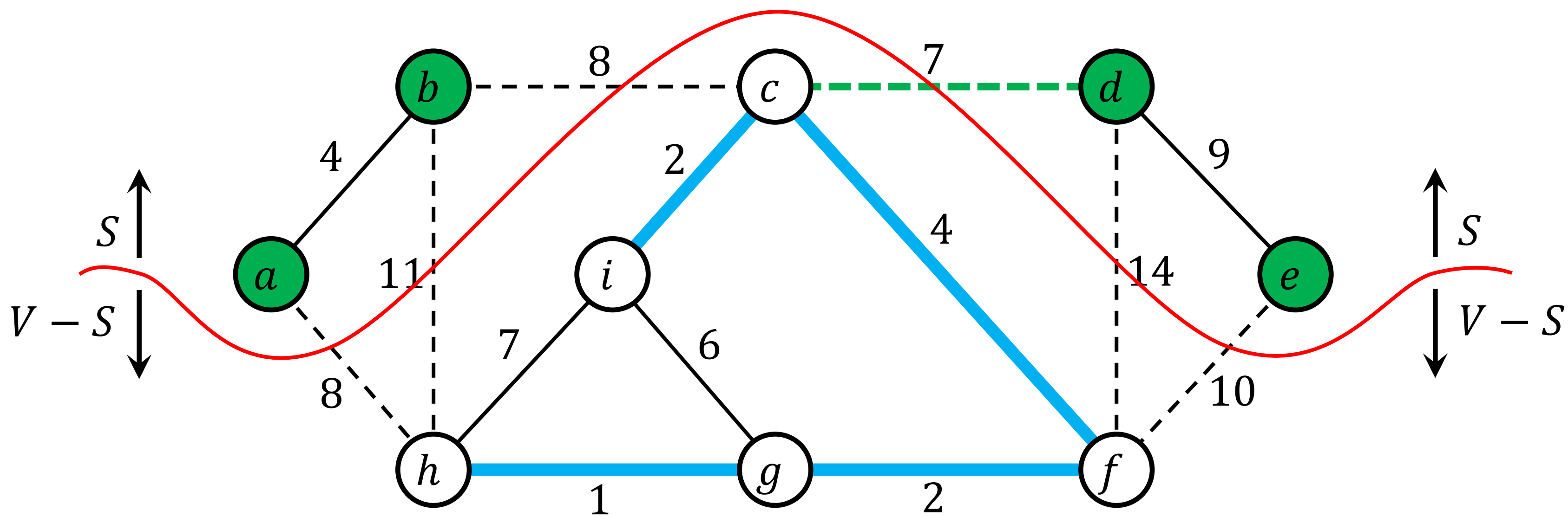
$$A = \{(a,b),(c,f),(c,i),(f,g),(g,h)\}.$$

Then edge (c,d) is a light edge crossing the cut.

MST: Finding Safe Edges

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. Multiple light edges can cross a cut.



The entire set A can be on the same side of the cut, e.g.,

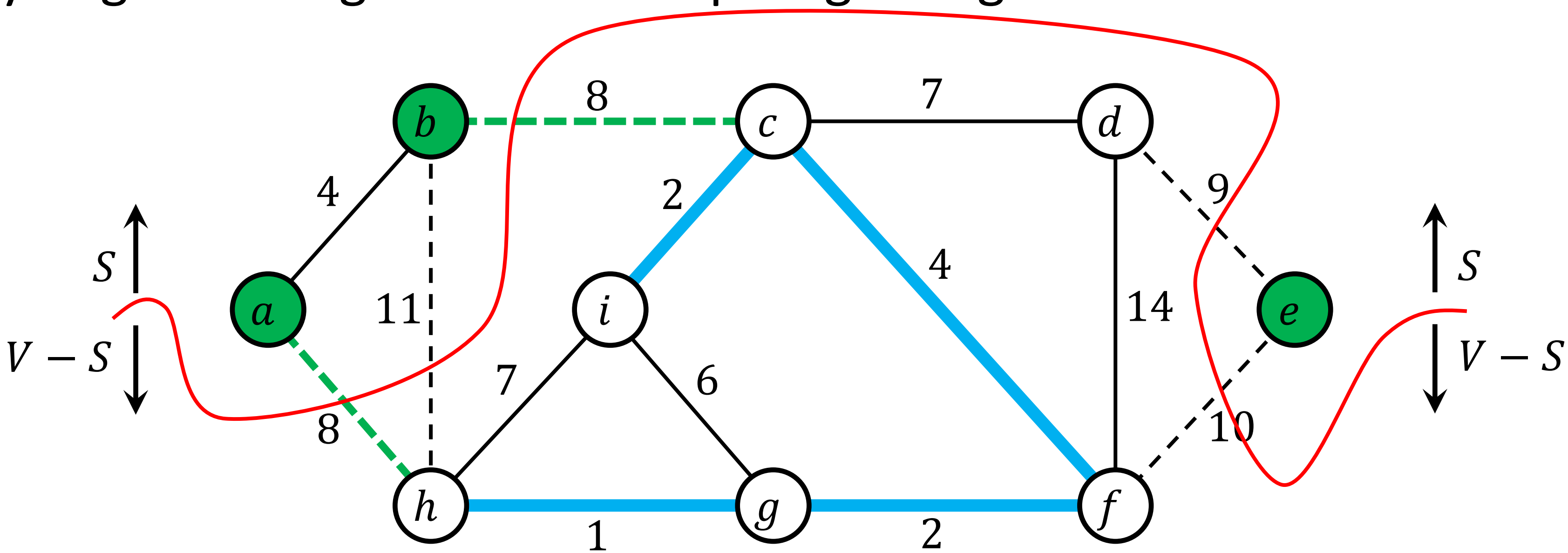
$$A = \{(c, f), (c, i), (f, g), (g, h)\}.$$

Still edge (c, d) is a light edge crossing the cut.

MST: Finding Safe Edges

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. Multiple light edges can cross a cut.



Consider a different cut as shown above.

Consider the same set $A = \{(c, f), (c, i), (f, g), (g, h)\}$.

Now both (a, h) and (b, c) are light edges crossing the cut.

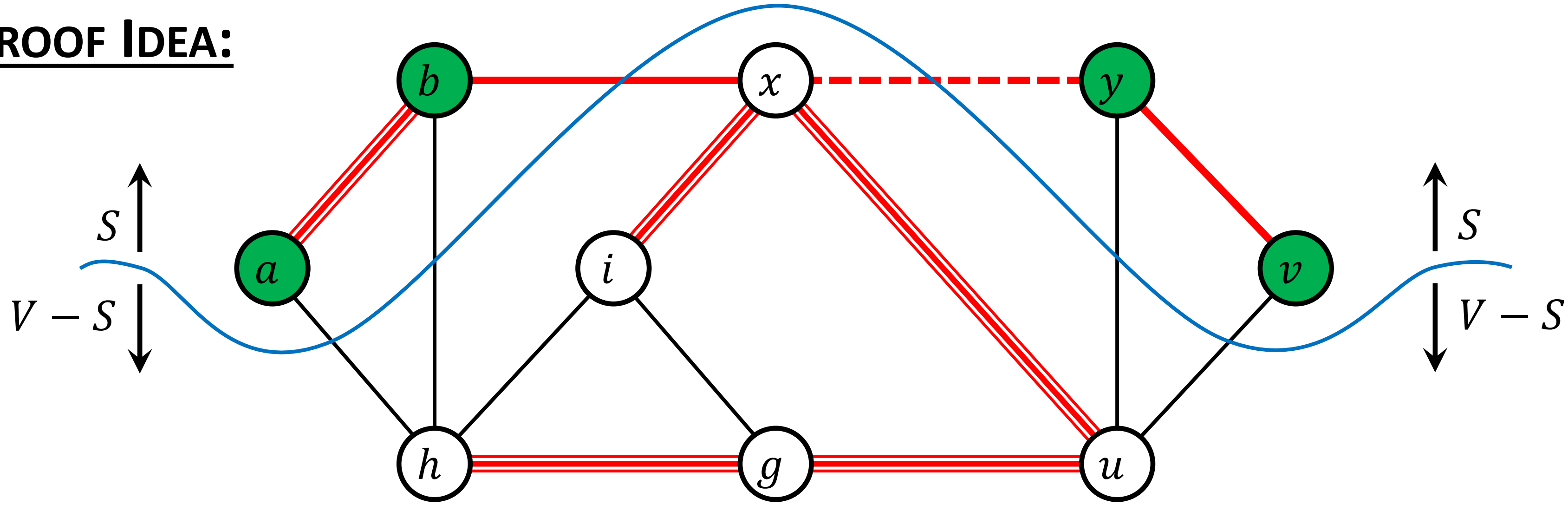
MST: Finding Safe Edges

THEOREM: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let $(S, V \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V \setminus S)$. Then, edge (u, v) is safe for A.

MST: Finding Safe Edges

THEOREM: … … … Let $A \subset E$ is included in some MST T of G, and let $(S, V \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V \setminus S)$. Then, edge (u, v) is safe for A.

PROOF IDEA:



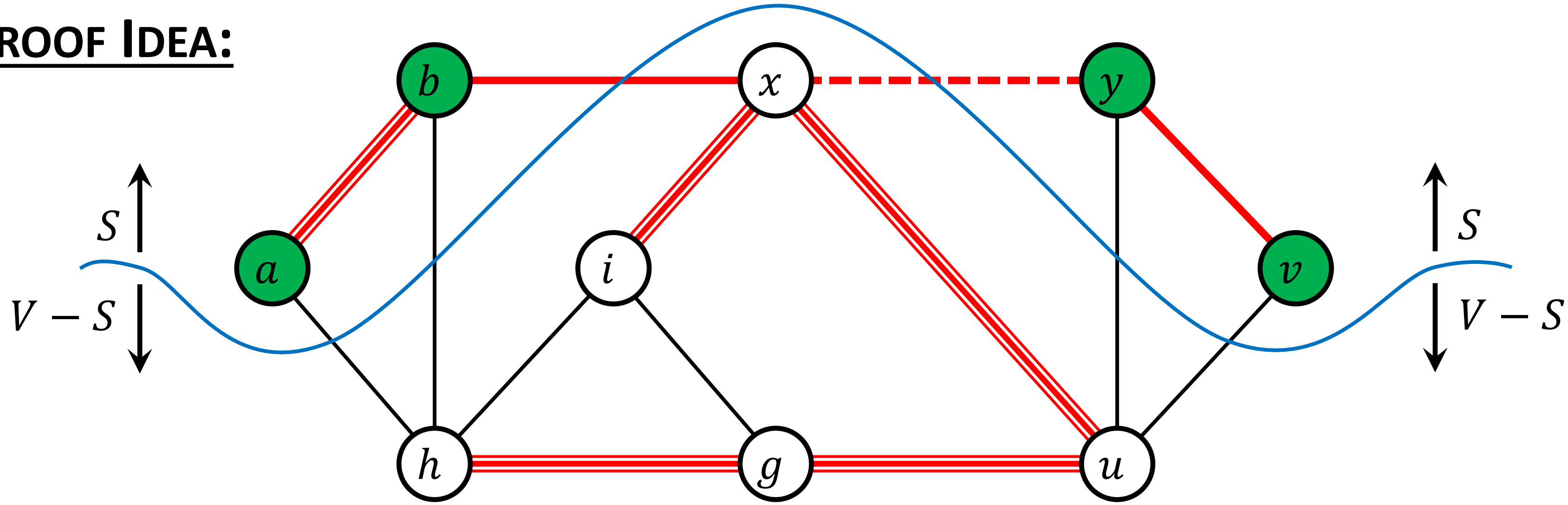
Let (u, v) be a light edge crossing the cut.

Let's assume $(u, v) \notin T$, as otherwise we are done.

MST: Finding Safe Edges

THEOREM: … … … Let $A \subset E$ is included in some MST T of G, and let $(S, V \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V \setminus S)$. Then, edge (u, v) is safe for A.

PROOF IDEA:

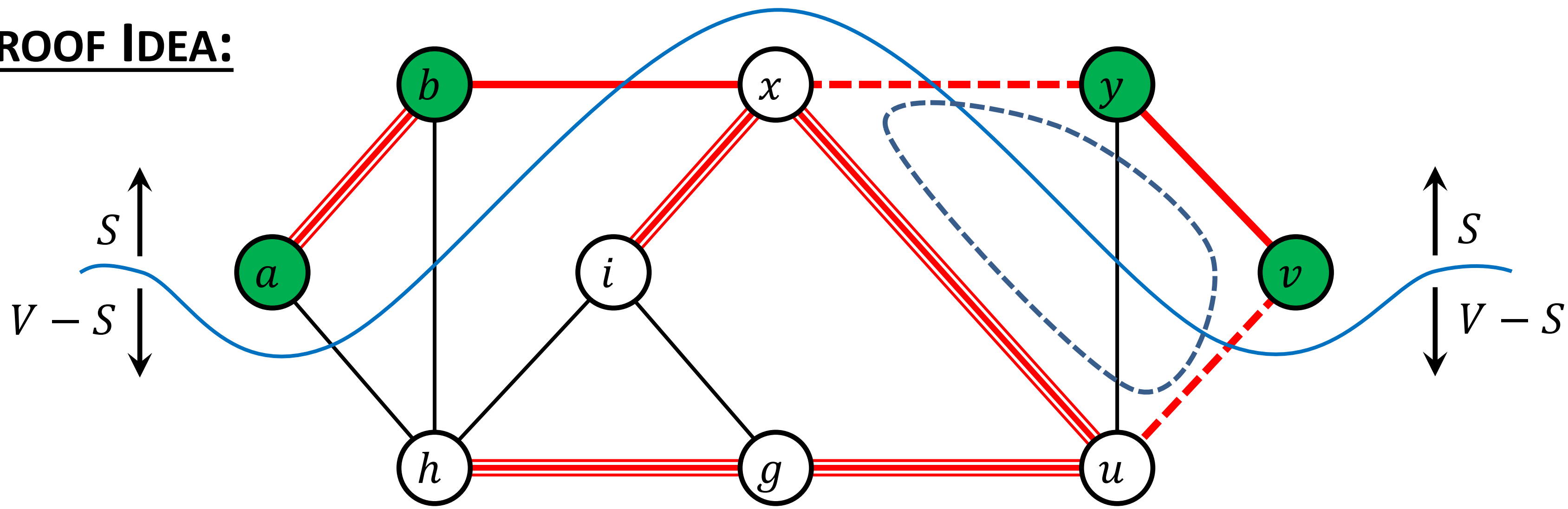


As T is a spanning tree, some edge $(x, y) \in T$ must also cross the cut.

MST: Finding Safe Edges

THEOREM: … … … Let $A \subset E$ is included in some MST T of G, and let $(S, V \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V \setminus S)$. Then, edge (u, v) is safe for A.

PROOF IDEA:



As T is a spanning tree, some edge $(x, y) \in T$ must also cross the cut.

Let's add edge (u, v) to T.

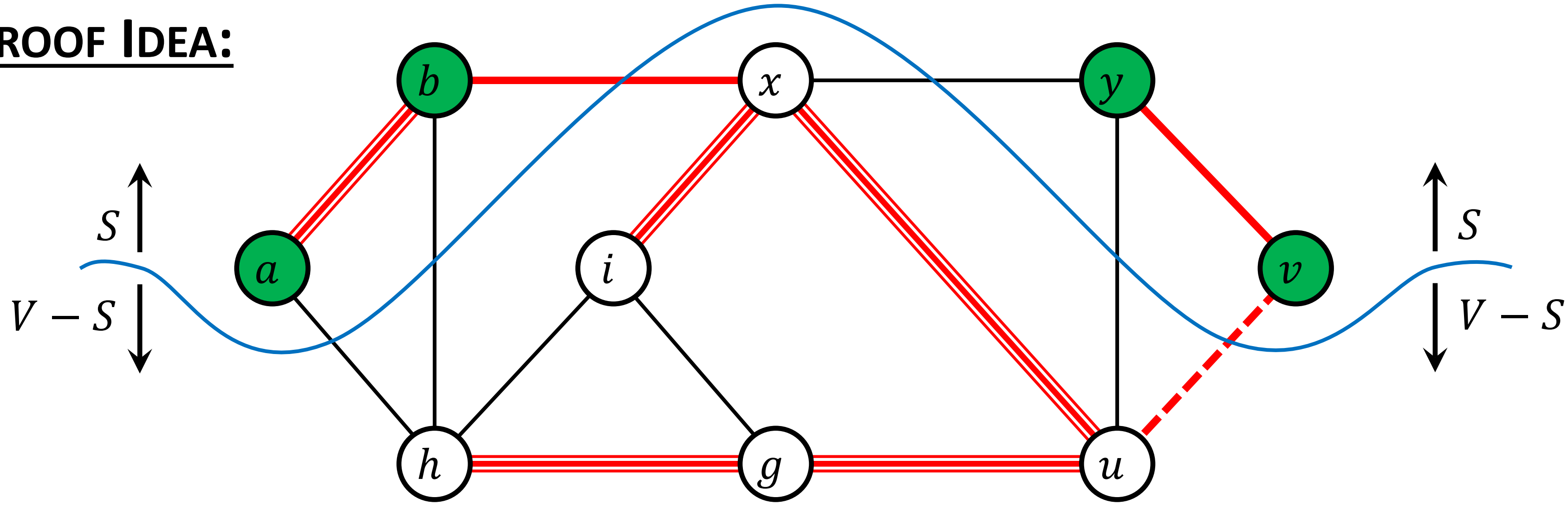
That must form a cycle in $T \cup \{(u, v)\}$.

So, $T \cup \{(u, v)\}$ is not a tree.

MST: Finding Safe Edges

THEOREM: … … … Let $A \subset E$ is included in some MST T of G, and let $(S, V \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V \setminus S)$. Then, edge (u, v) is safe for A.

PROOF IDEA:



We can break the cycle by removing edge (x, y) from $T \cup \{(u, v)\}$.

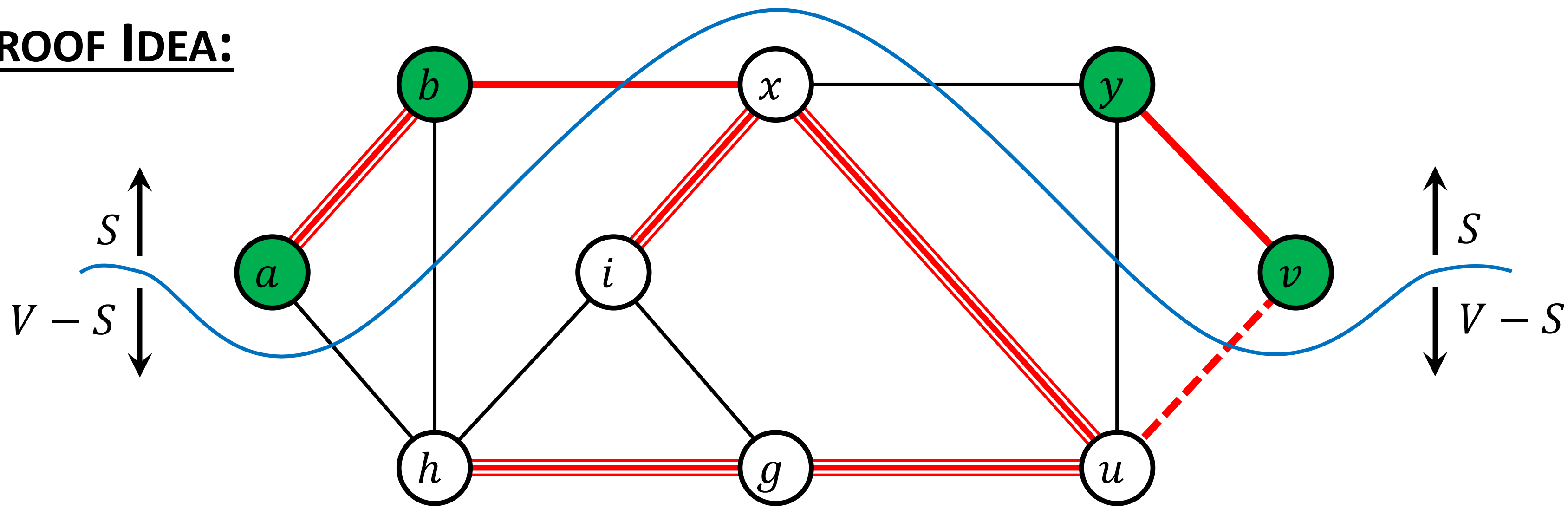
Let $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

Observe that T' is now a spanning tree of G.

MST: Finding Safe Edges

THEOREM: … … … Let $A \subset E$ is included in some MST T of G, and let $(S, V \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V \setminus S)$. Then, edge (u, v) is safe for A.

PROOF IDEA:



Now, $w(T') = w(T - \{(x, y)\} \cup \{(u, v)\})$

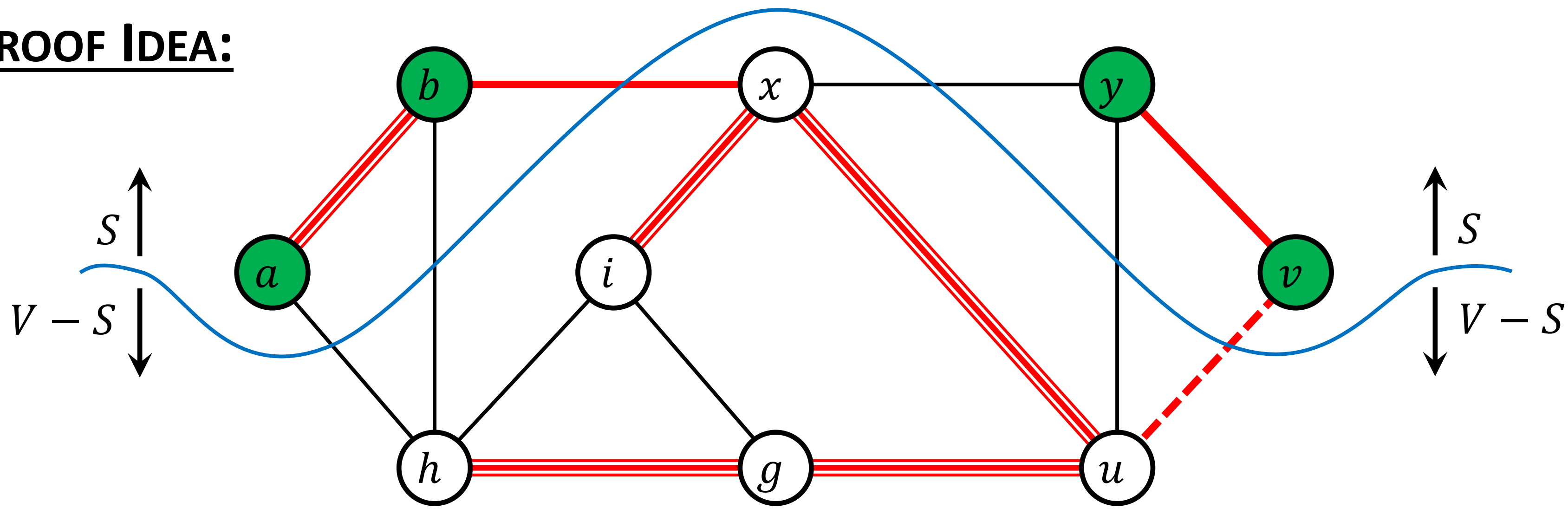
$$= w(T) - w\big((x, y)\big) + w\big((u, v)\big) \le w(T)$$

But we assumed that T is an MST of G, and so $w(T) \le w(T')$

MST: Finding Safe Edges

THEOREM: … … … Let $A \subset E$ is included in some MST T of G, and let $(S, V \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V \setminus S)$. Then, edge (u, v) is safe for A.

PROOF IDEA:



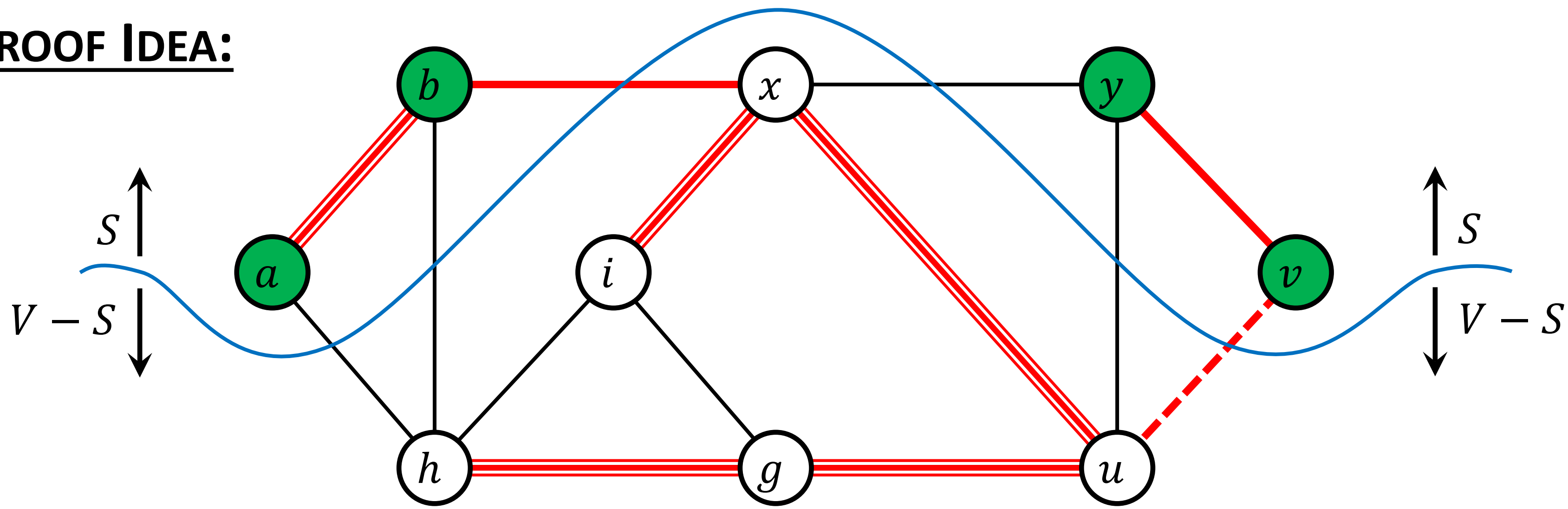
Since, $w(T') \leq w(T)$ and $w(T) \leq w(T')$, we have $w(T') = w(T)$.

So, T' must also be an MST of G.

MST: Finding Safe Edges

THEOREM: … … … Let $A \subset E$ is included in some MST T of G, and let $(S, V \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V \setminus S)$. Then, edge (u, v) is safe for A.

PROOF IDEA:



Since $A \subseteq T$ and $(x, y) \notin A$, we have $A \subseteq T'$.

Thus, $A \cup \{(u, v)\} \subseteq T'$.

Since T' is an MST of G, edge (u, v) is safe for A.

MST: Finding Safe Edges

THEOREM: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let $(S, V \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V \setminus S)$. Then, edge (u, v) is safe for A.

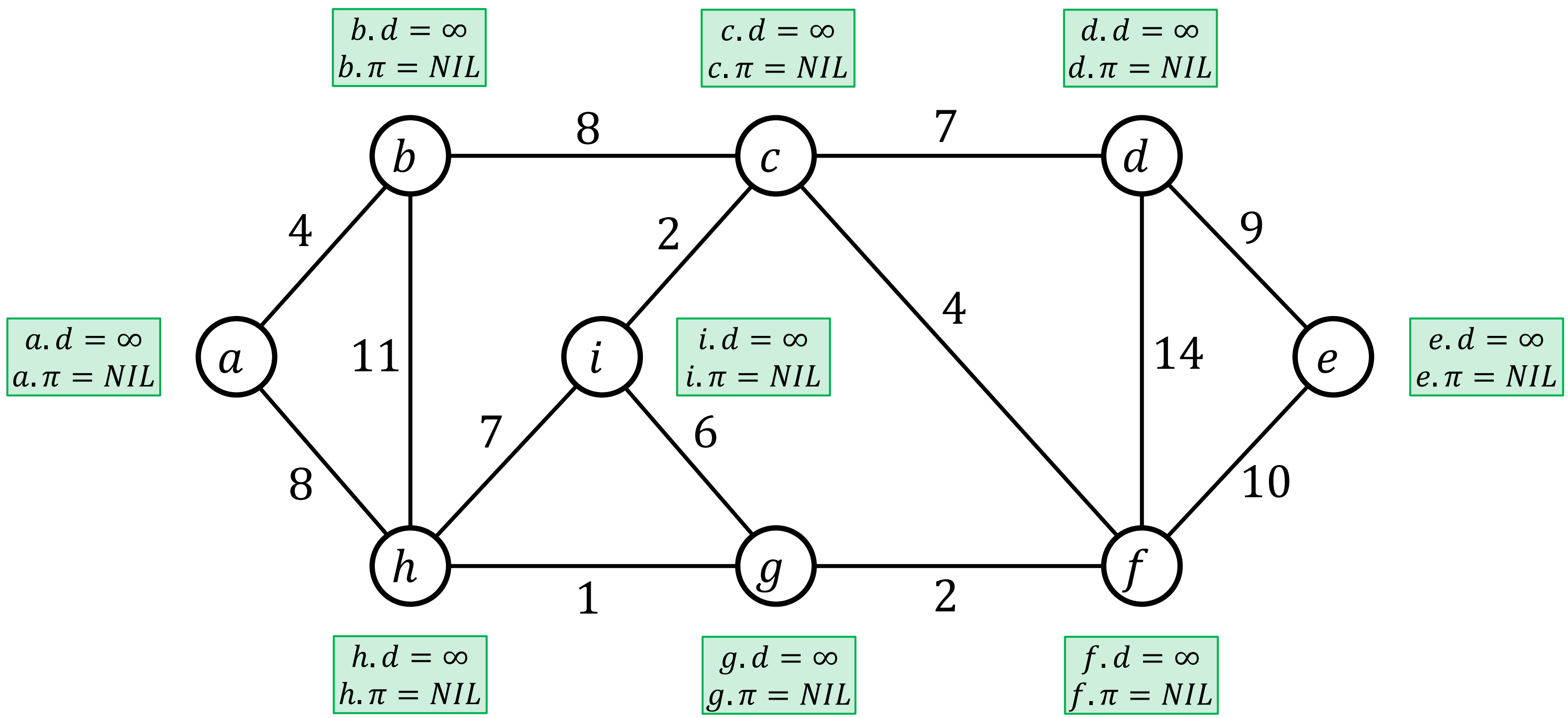
COROLLARY: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$ If (u, v) is a light edge crossing from C to some other component of G_A, then edge (u, v) is safe for A.

MST: Prim's Algorithm

```
MST-Prim ( G = (V, E), w, r )
    for each vertex v ∈ G.V do
        v.d ← ∞
        v.π ← NIL
    r.d ← 0
    Min-Heap Q ← ∅
    for each vertex v ∈ G.V do
        INSERT( Q, v )
    while Q ≠ ∅ do
        u ← EXTRACT-MIN( Q )
        for each (u, v) ∈ G.E do
            if v ∈ Q and w(u, v) < v.d then
                v.d ← w(u, v)
                v.π ← u
                DECREASE-KEY( Q, v, w(u, v) )
```

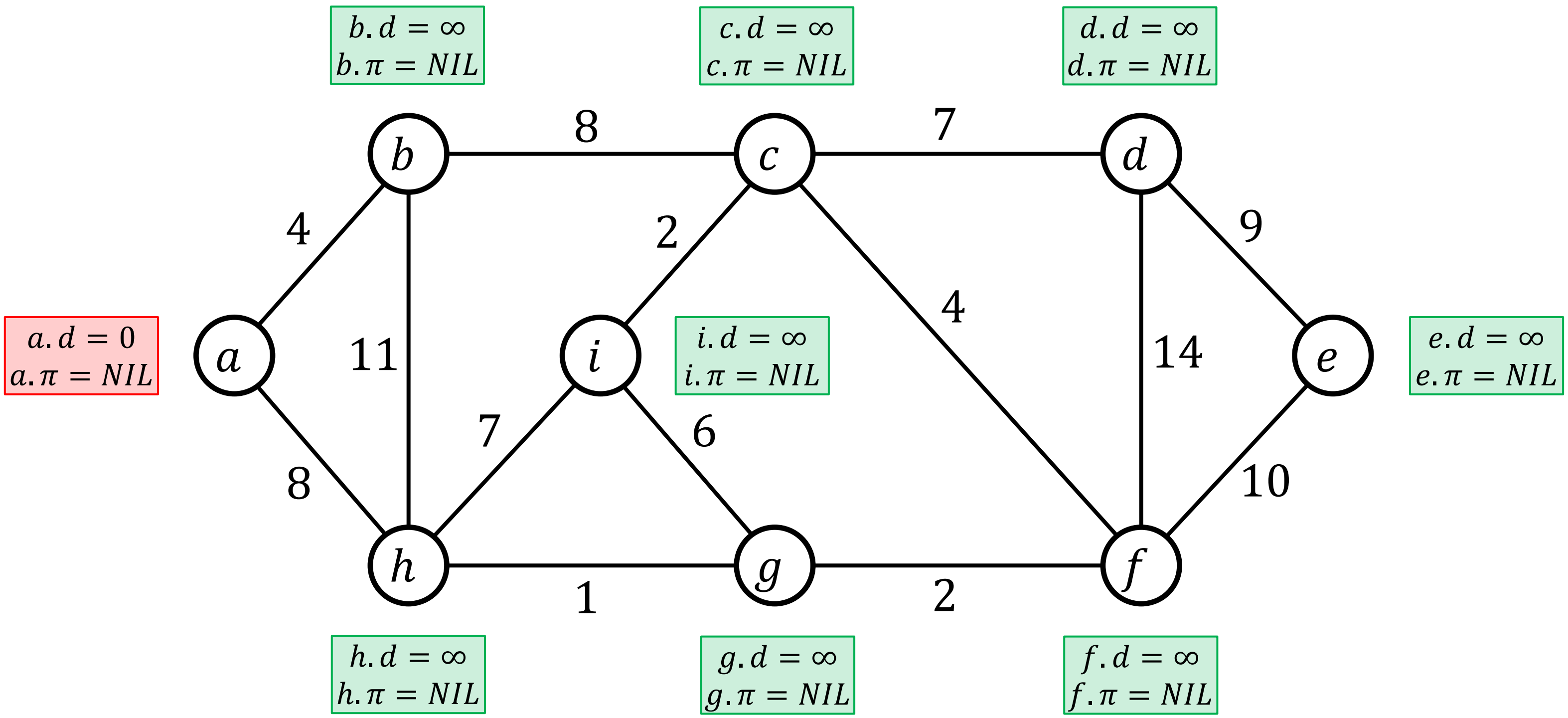
MST: Prim's Algorithm

Initial State



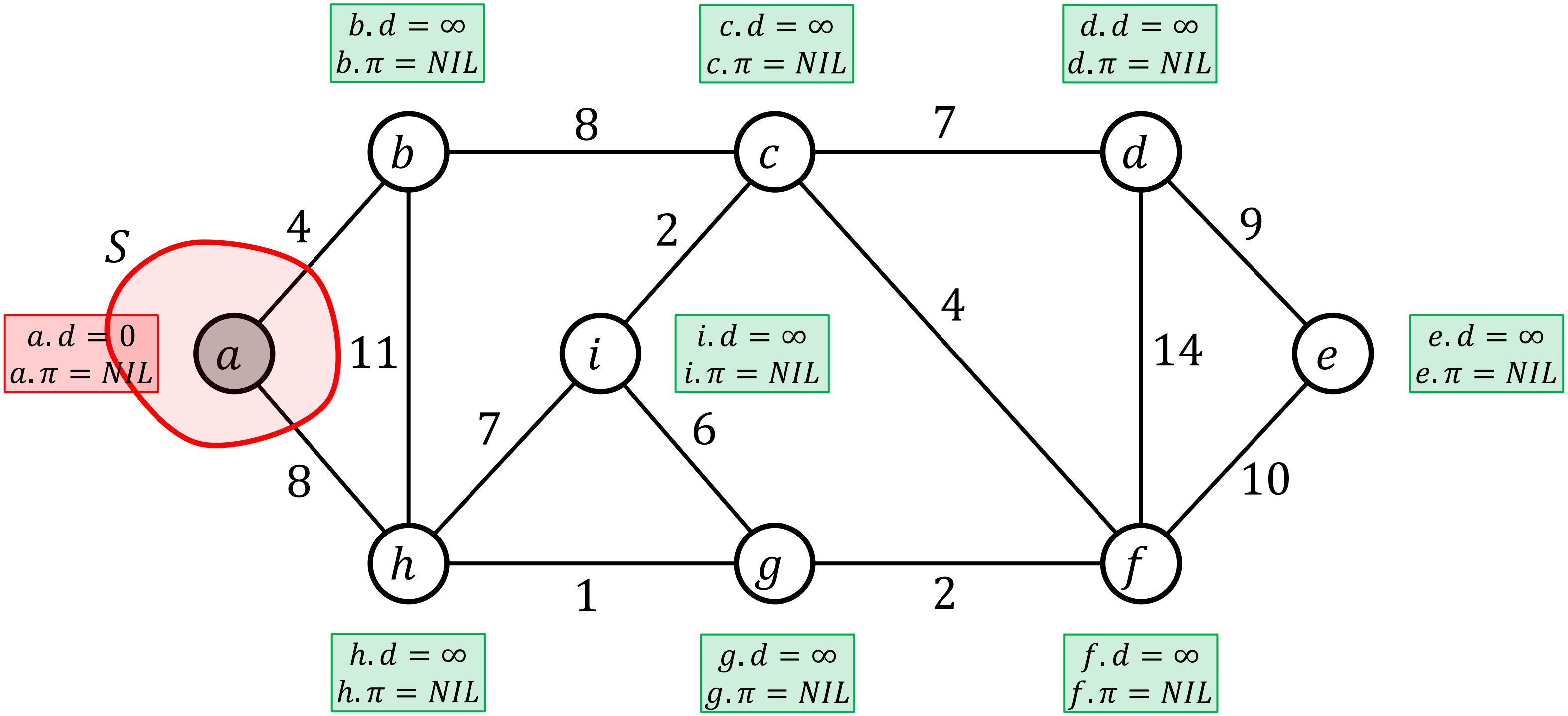
MST: Prim's Algorithm

Initial State

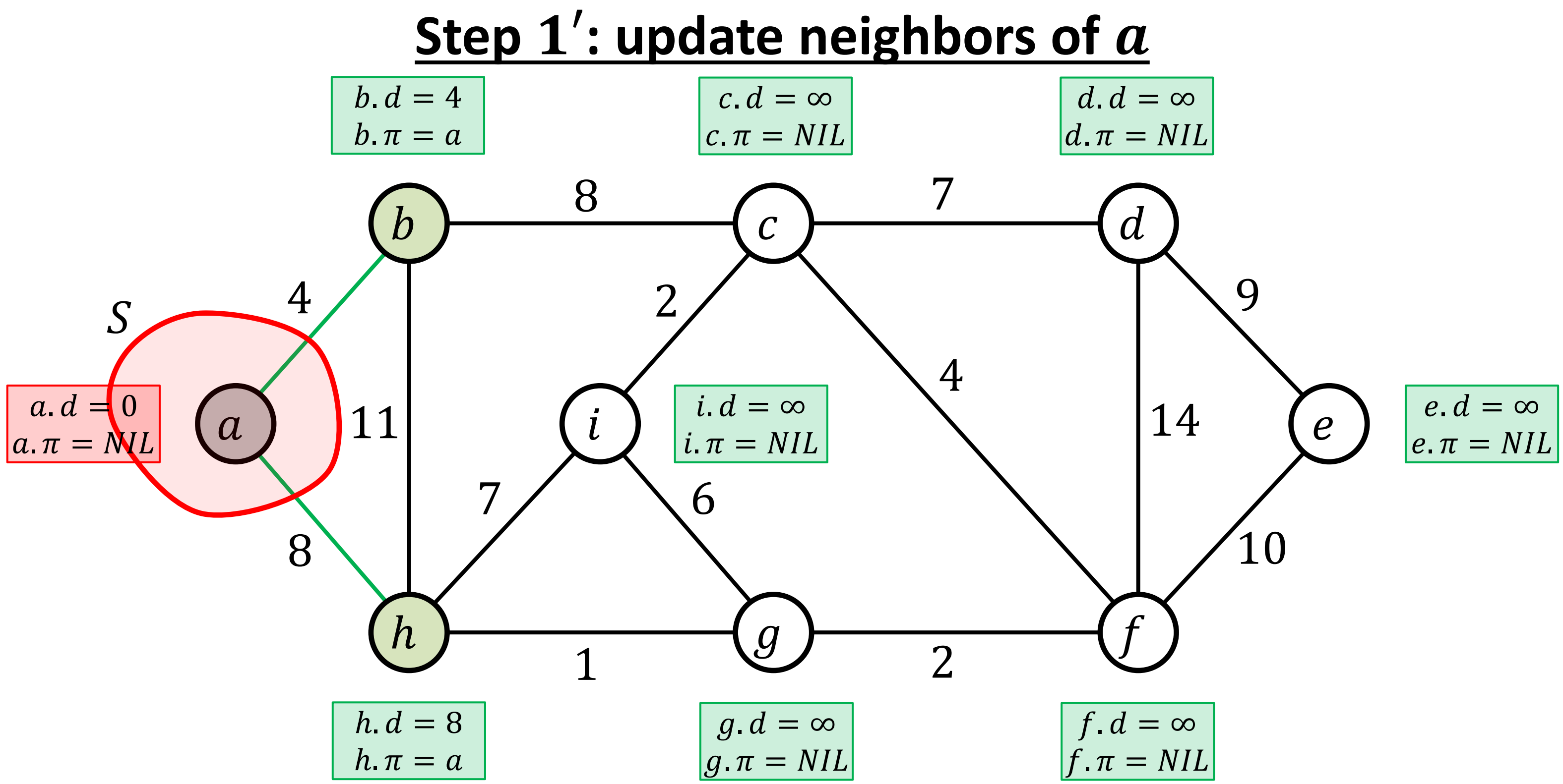


MST: Prim's Algorithm

Step 1: add vertex a to MST



MST: Prim's Algorithm

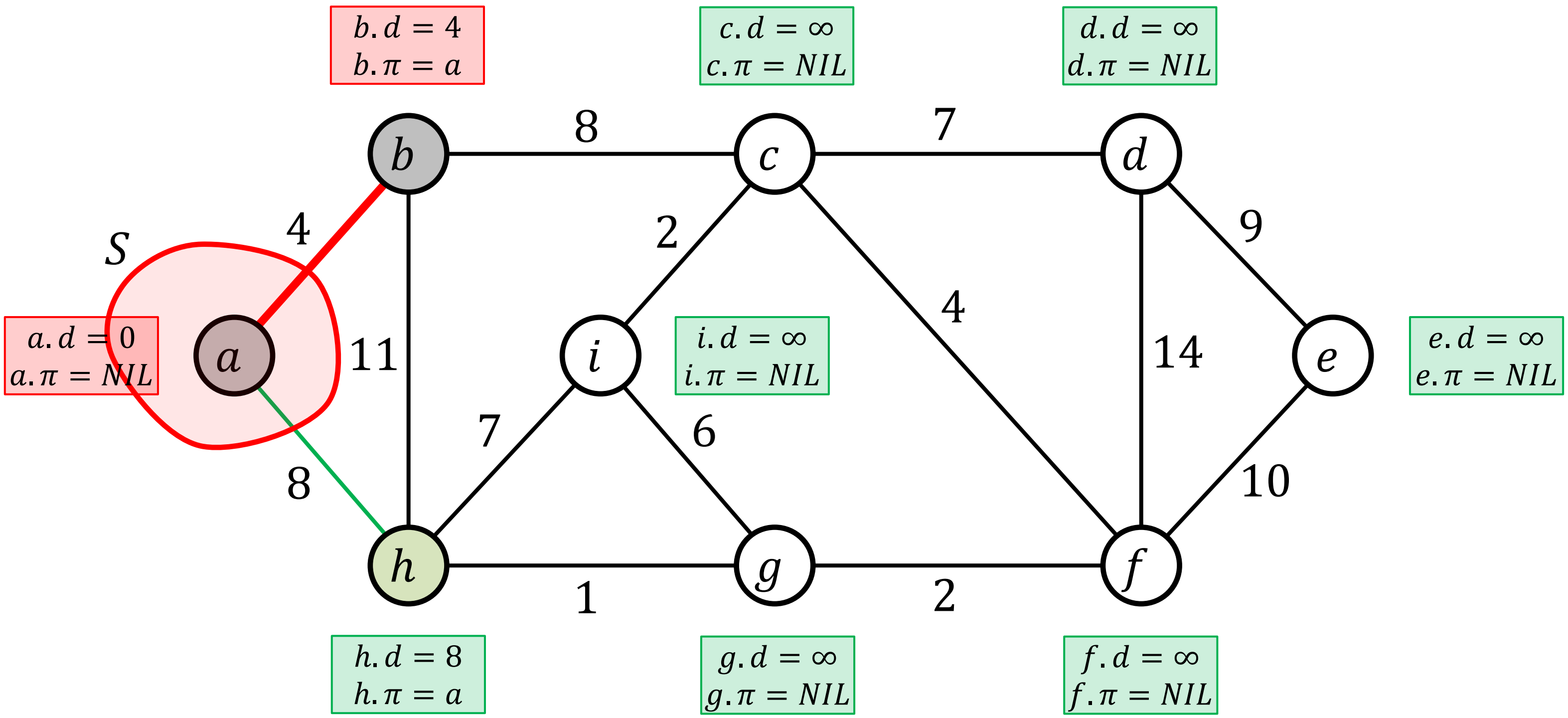


$S = \{a\}$

Cut = $(S, V - S)$

MST: Prim's Algorithm

Step 2: add vertex b through edge (a, b)



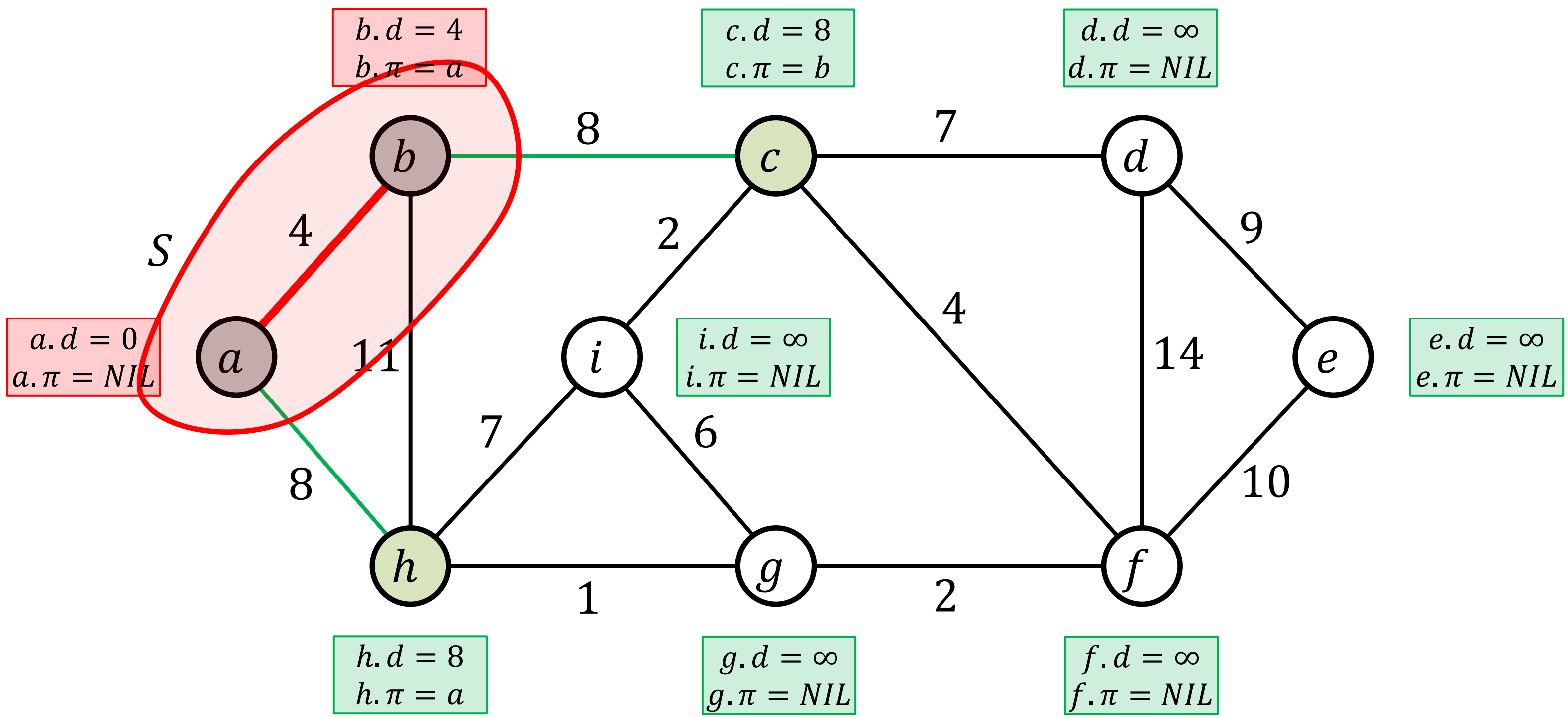
$S = \{a\}$

Cut = $(S, V - S)$

(a, b) is the light edge crossing the cut

MST: Prim's Algorithm

Step $2'$: update neighbors of b

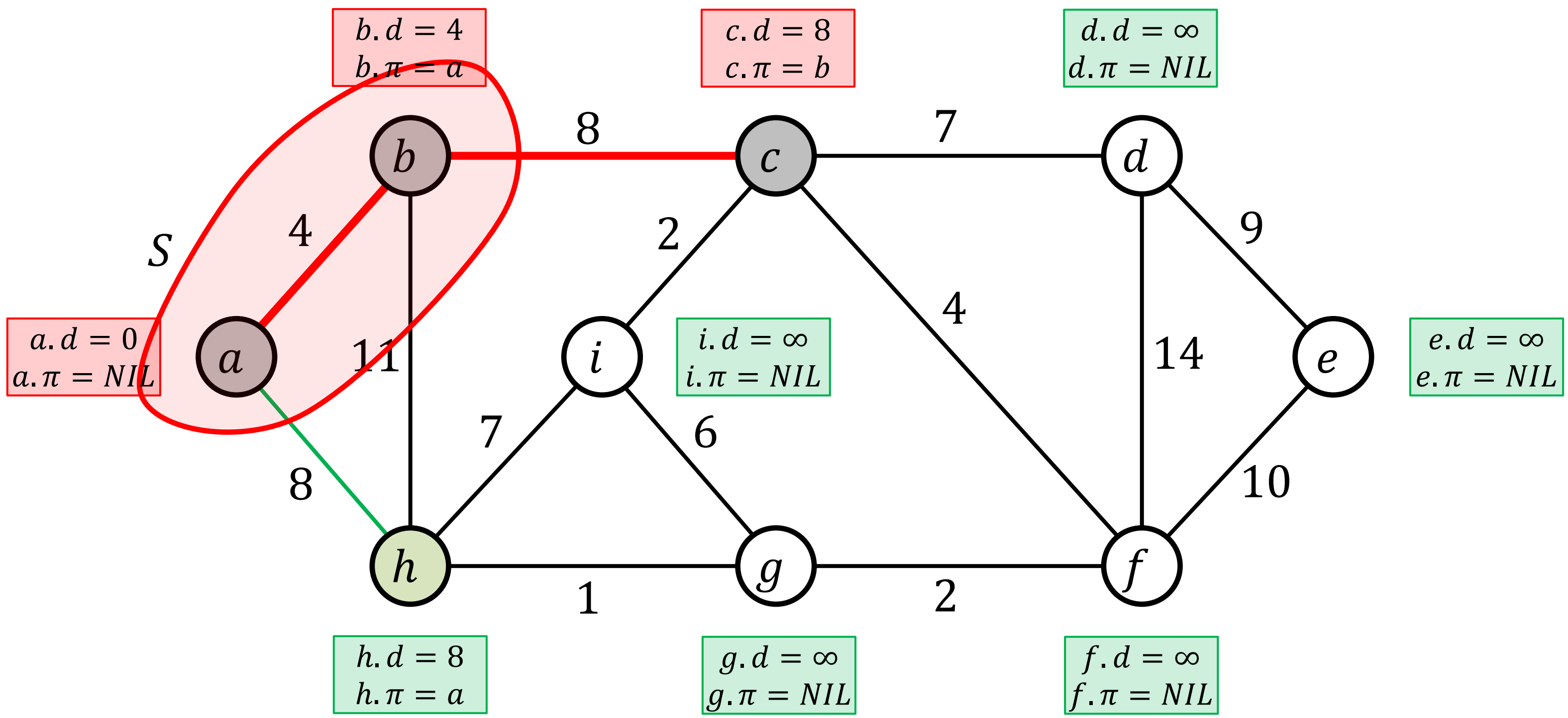


$S = \{a, b\}$

Cut = $(S, V - S)$

MST: Prim's Algorithm

Step 3: add vertex c through edge (b, c)



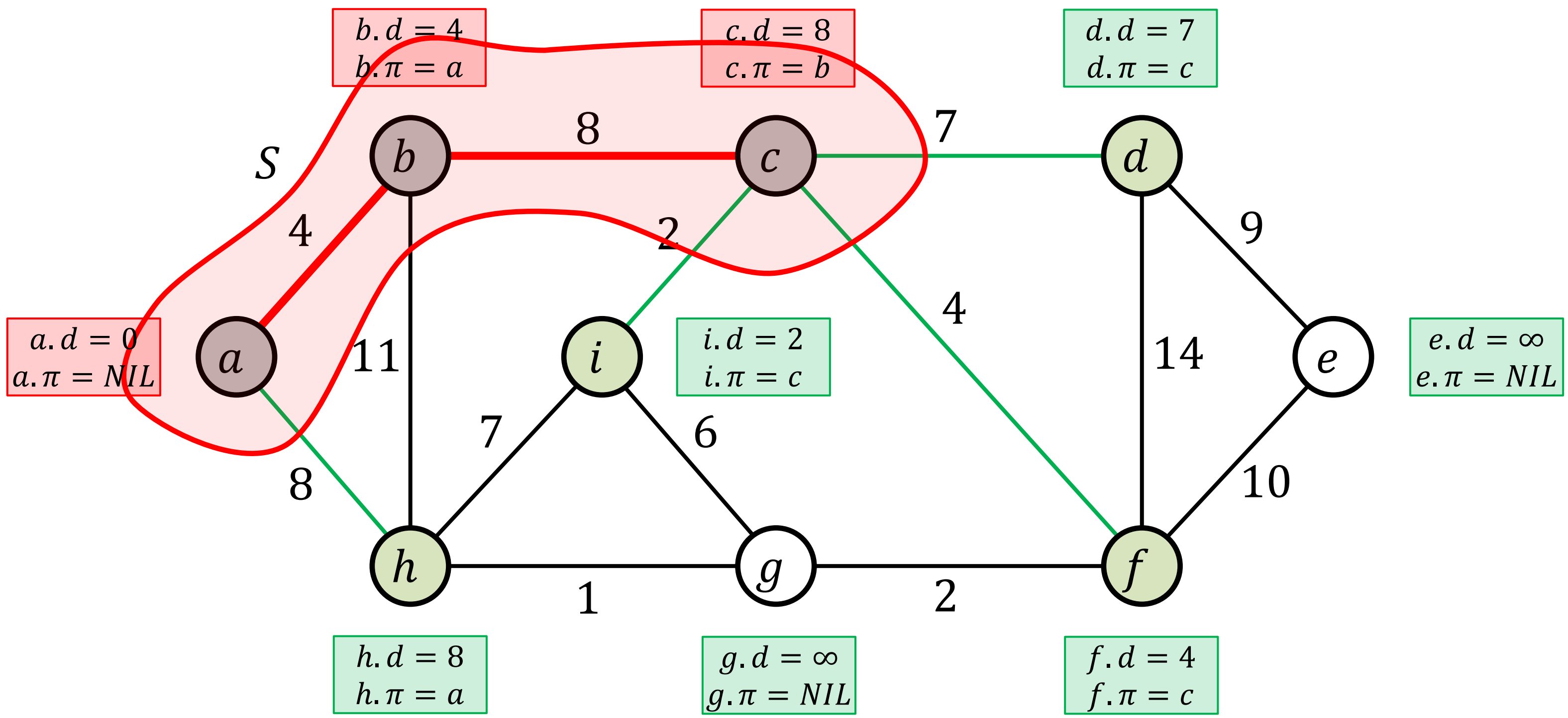
$S = \{a, b\}$

Cut = $(S, V - S)$

(b, c) is a light edge crossing the cut

MST: Prim's Algorithm

Step 3′: update neighbors of c

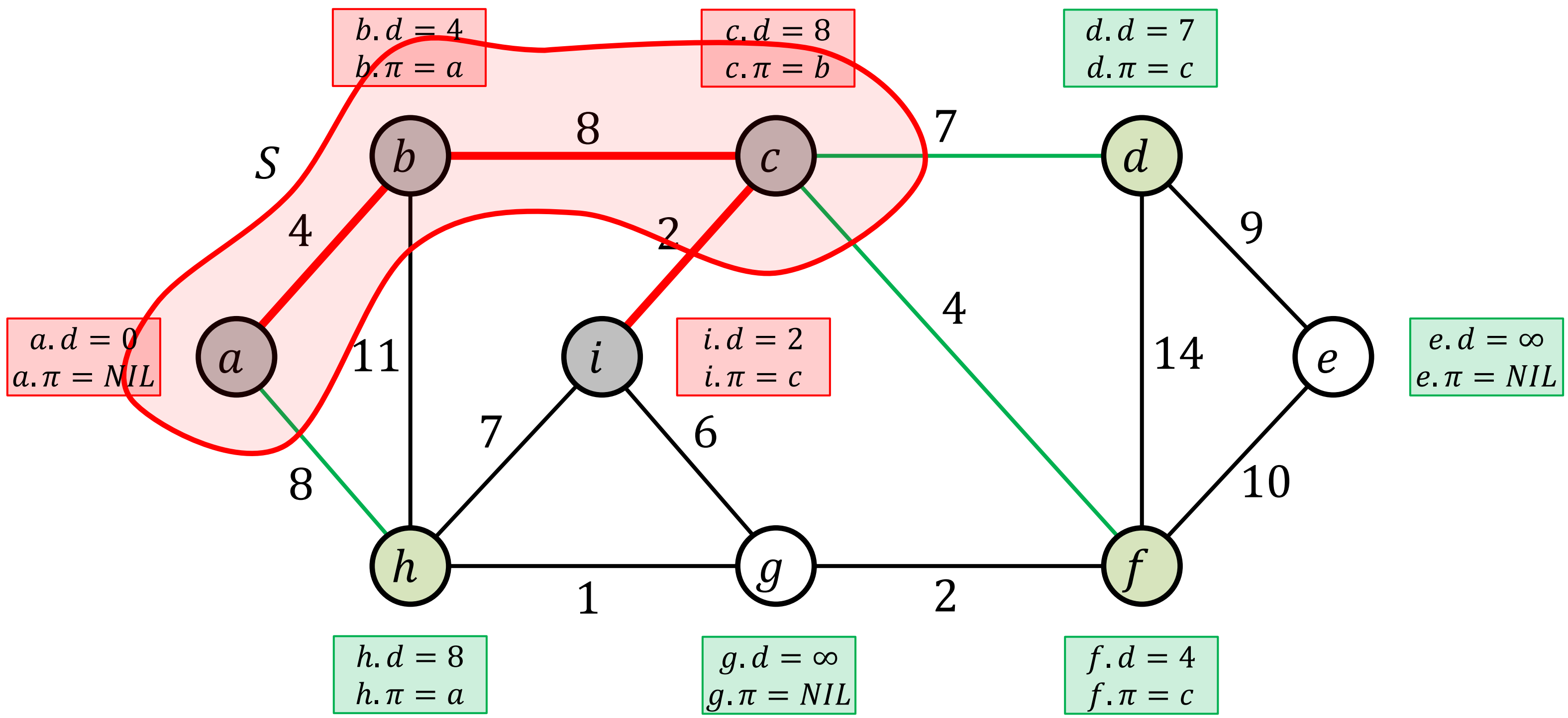


$S = \{a, b, c\}$

Cut = $(S, V - S)$

MST: Prim's Algorithm

Step 4: add vertex i through edge (c, i)



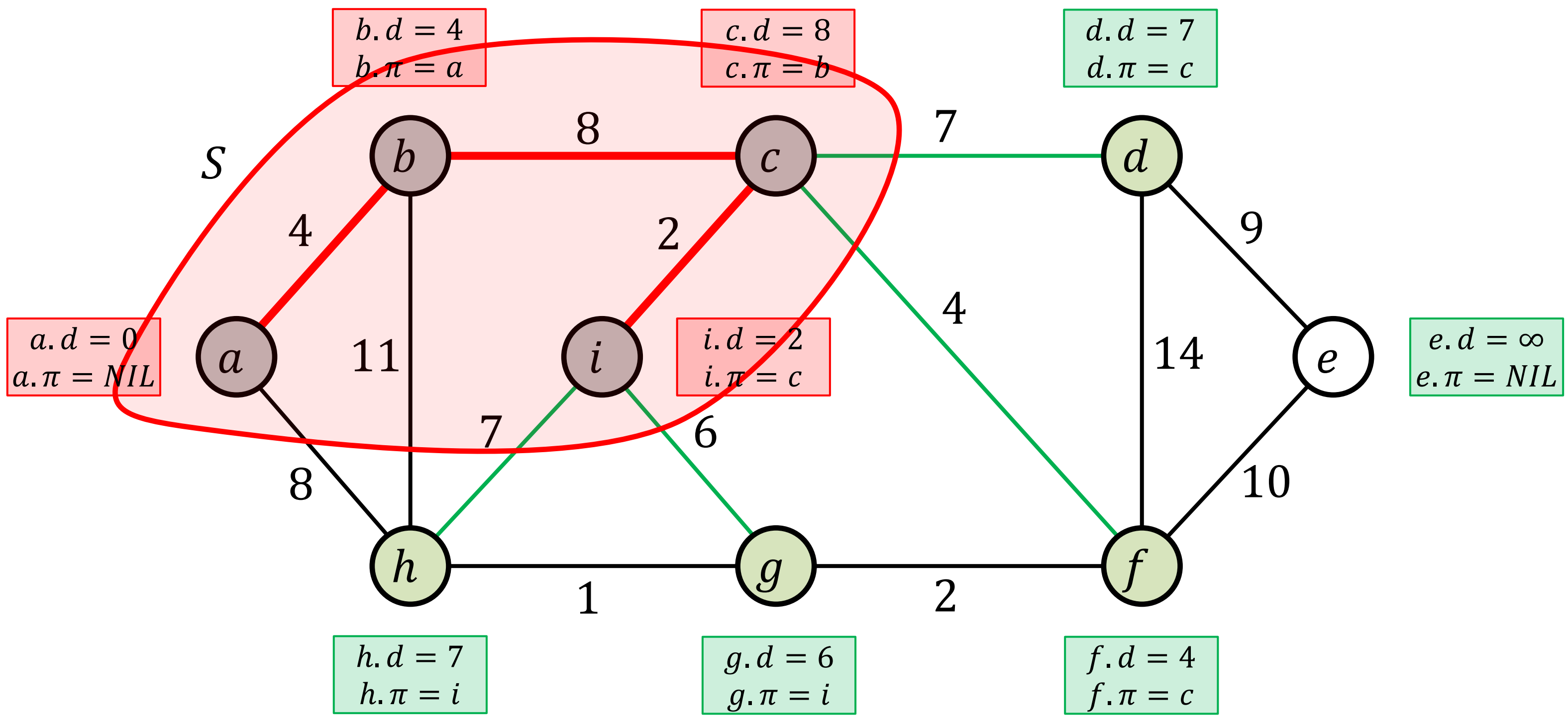
$S = \{a, b, c\}$

Cut = $(S, V - S)$

(c, i) is the light edge crossing the cut

MST: Prim's Algorithm

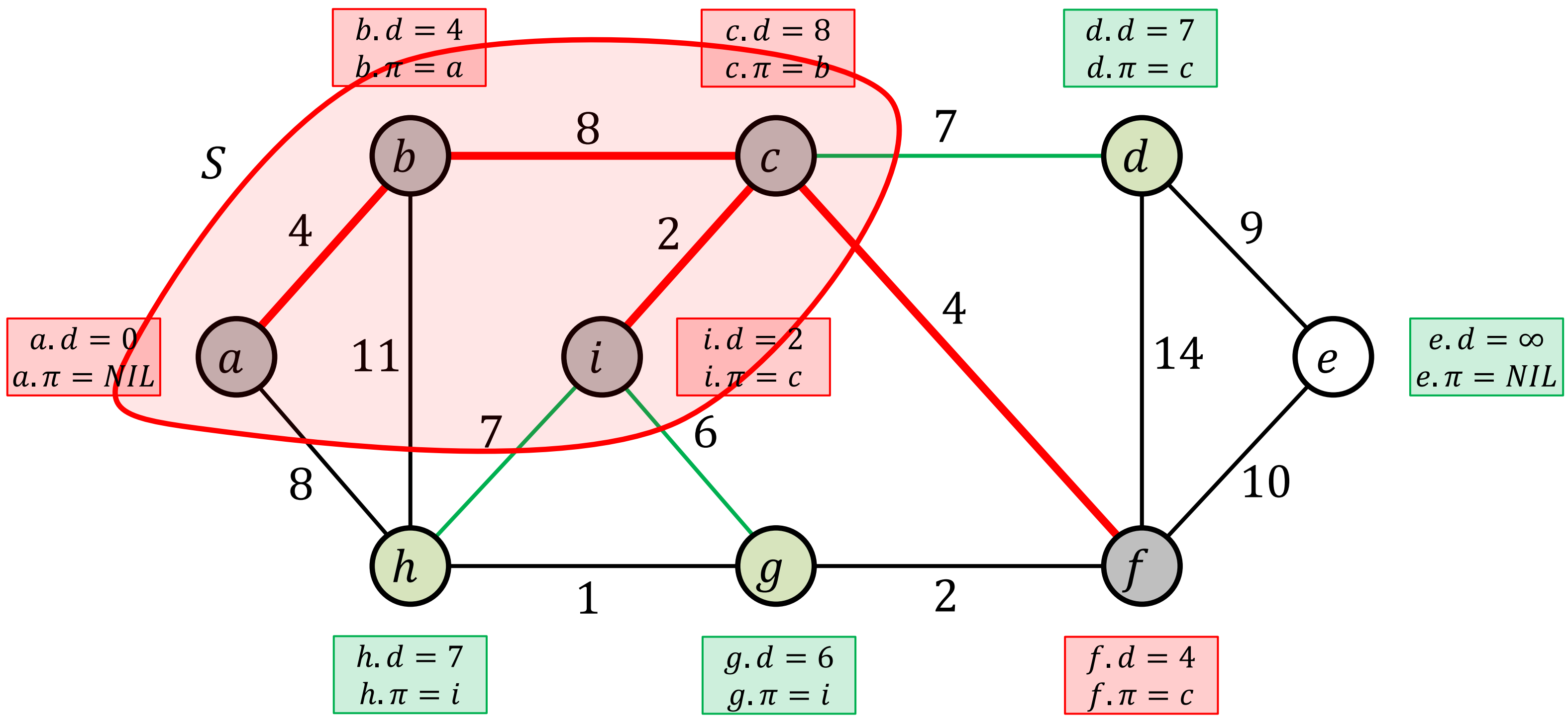
Step $4'$: update neighbors of $\boldsymbol{i}$



$S = \{a, b, c, i\}$
Cut = $(S, V - S)$

MST: Prim's Algorithm

Step 5: add vertex f through edge (c, f)



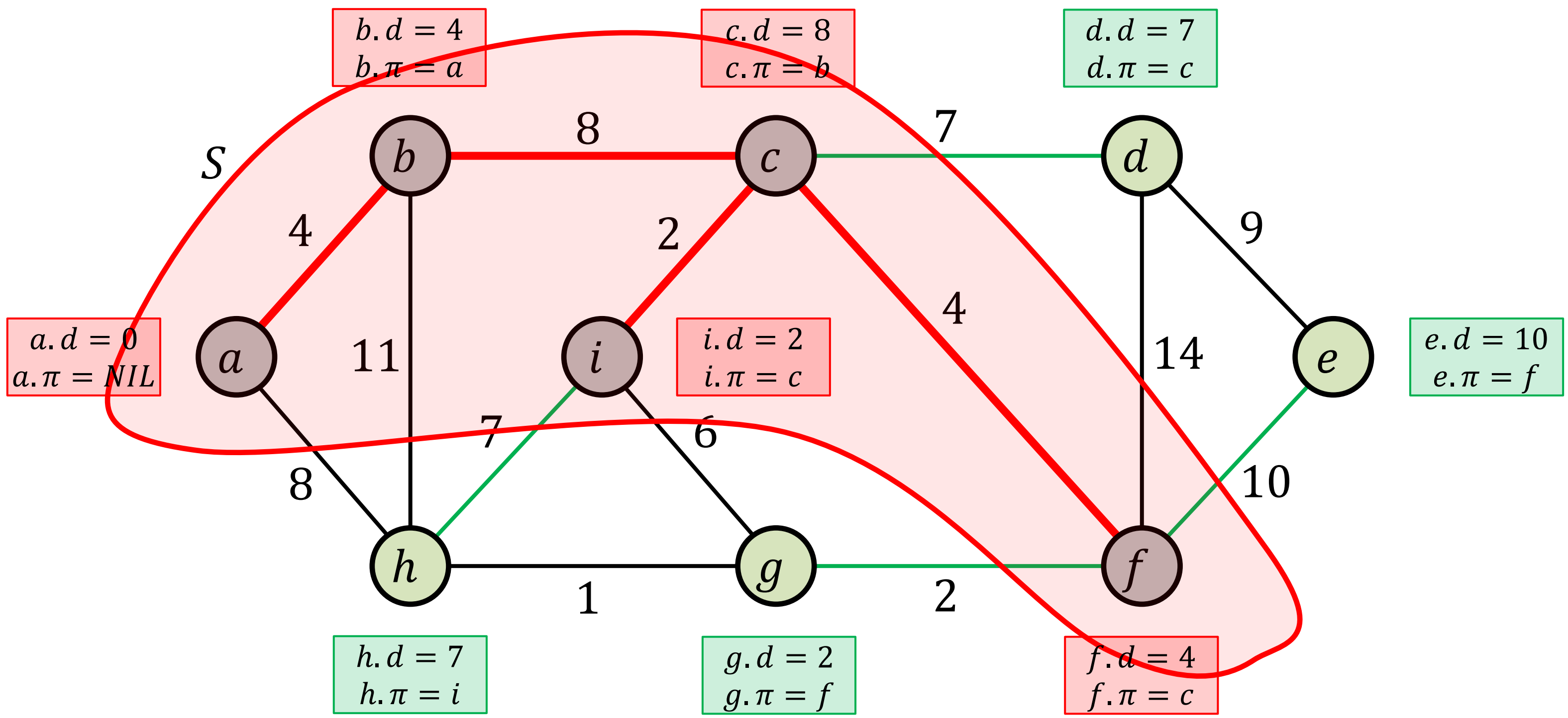
$S = \{a, b, c, i\}$

Cut = $(S, V - S)$

(c, f) is the light edge crossing the cut

MST: Prim's Algorithm

Step $5'$: update neighbors of f

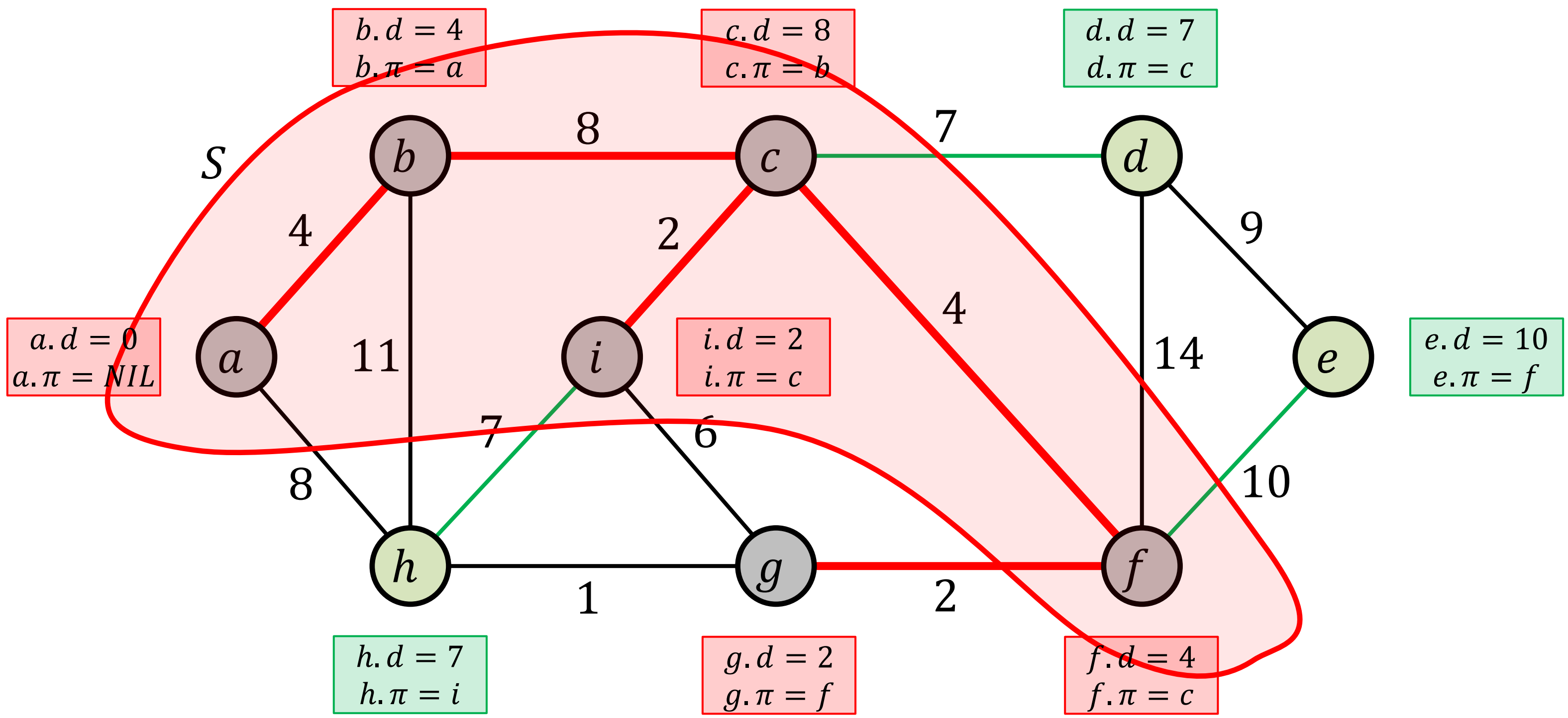


$S = \{a, b, c, i, f\}$

Cut = $(S, V - S)$

MST: Prim's Algorithm

Step 6: add vertex g through edge (f, g)



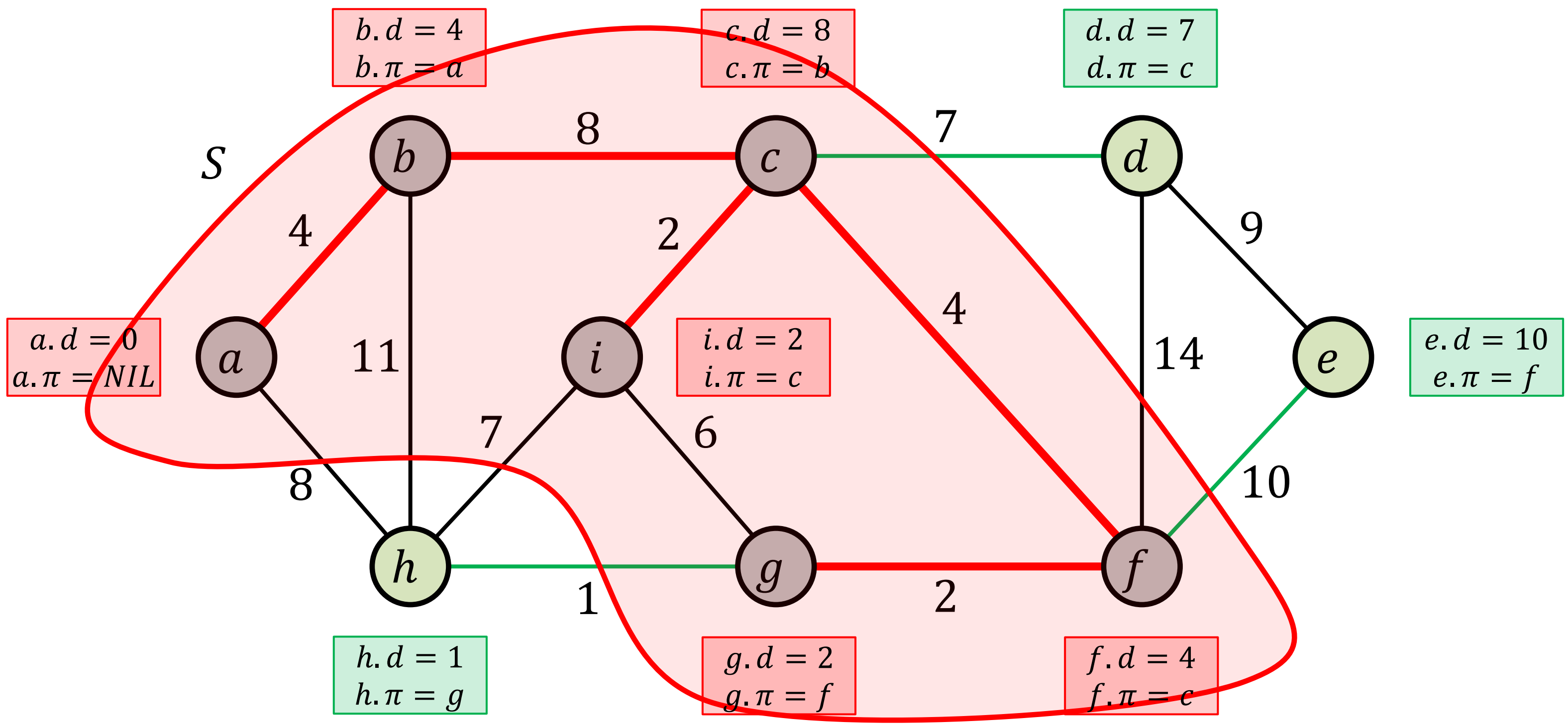
$S = \{a, b, c, i, f\}$

Cut = $(S, V - S)$

(c, g) is the light edge crossing the cut

MST: Prim's Algorithm

Step 6′: update neighbors of g

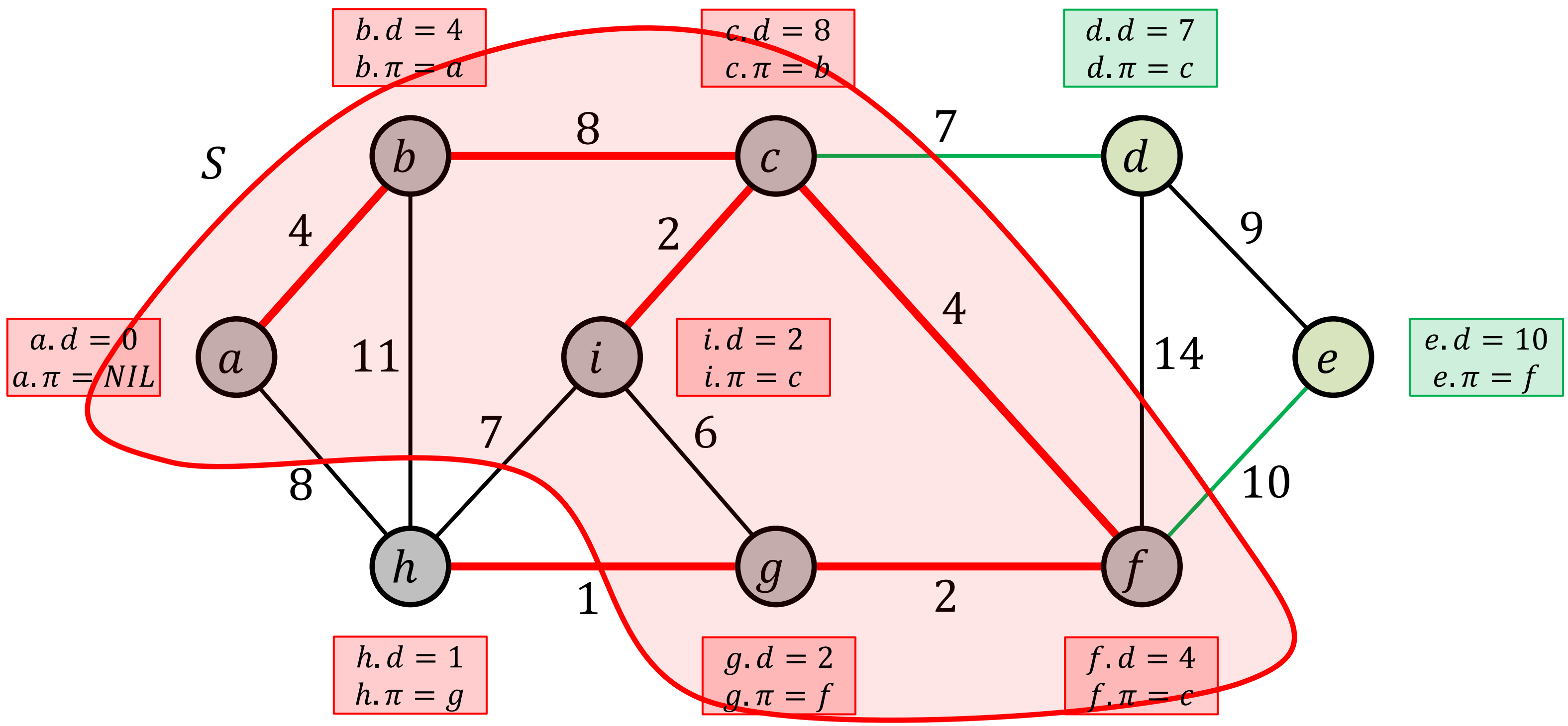


$S = \{a, b, c, i, f, g\}$

Cut = $(S, V - S)$

MST: Prim's Algorithm

Step 7: add vertex h through edge (g, h)



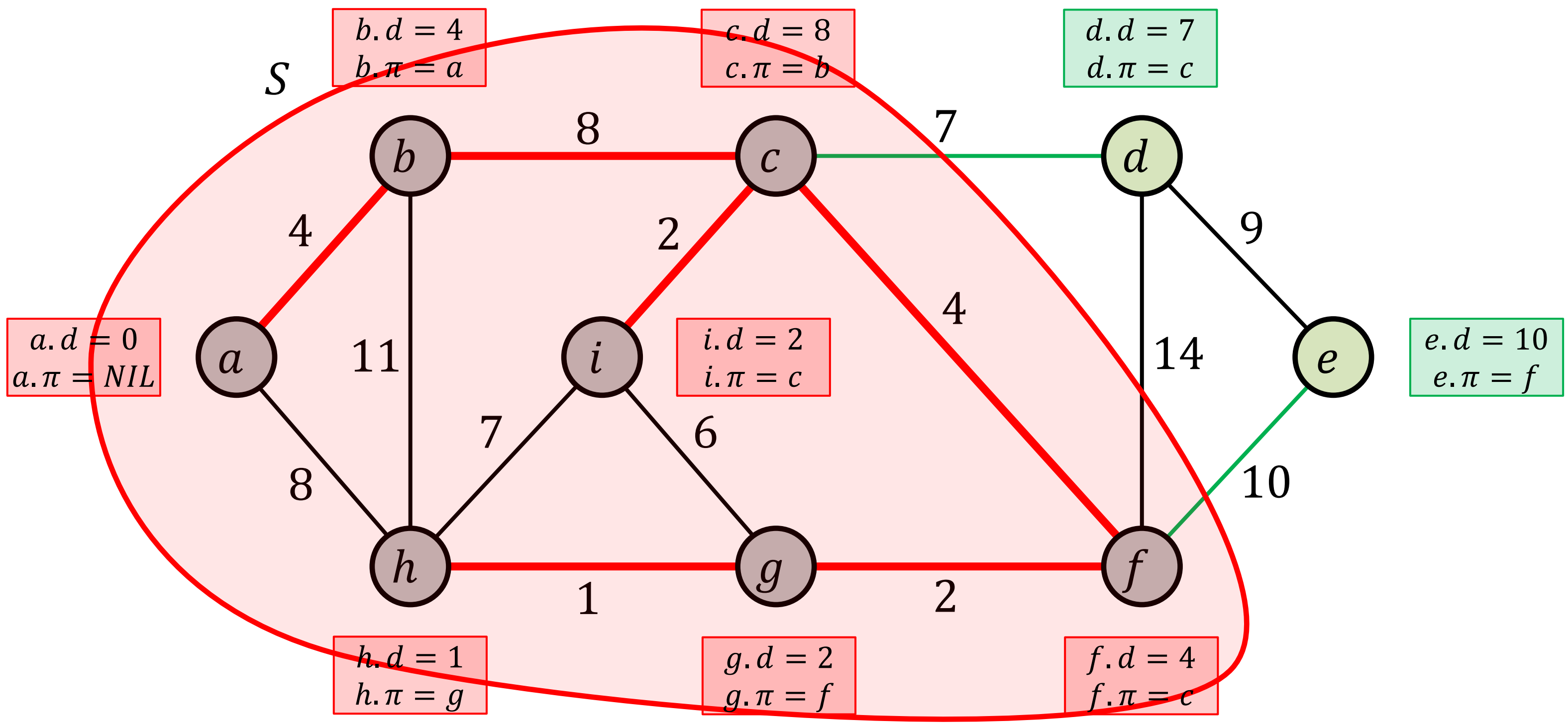
$S = \{a, b, c, i, f, g\}$

Cut = $(S, V - S)$

(g, h) is the light edge crossing the cut

MST: Prim's Algorithm

Step $7'$: update neighbors of h

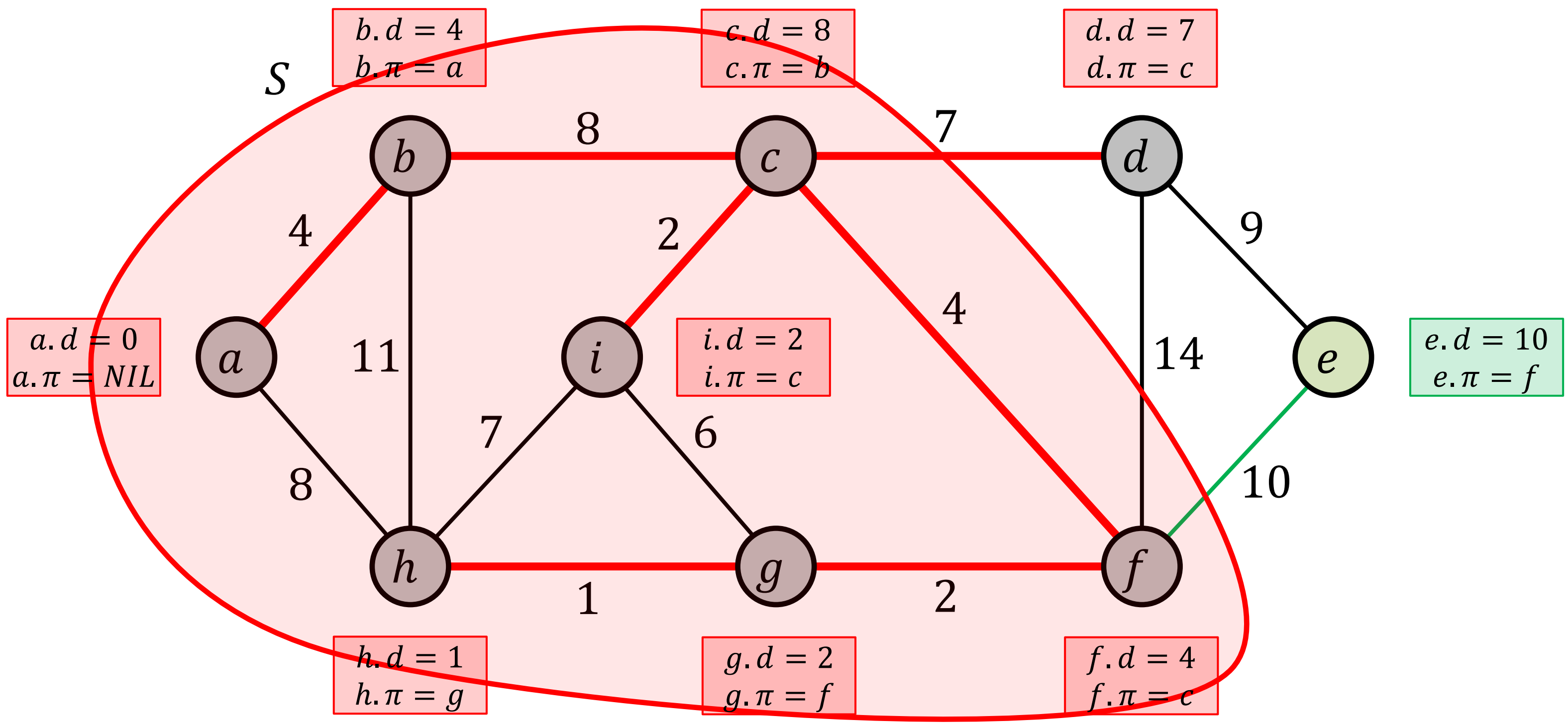


$S = \{a, b, c, i, f, g, h\}$

Cut = $(S, V - S)$

MST: Prim's Algorithm

Step 8: add vertex d through edge (c, d)



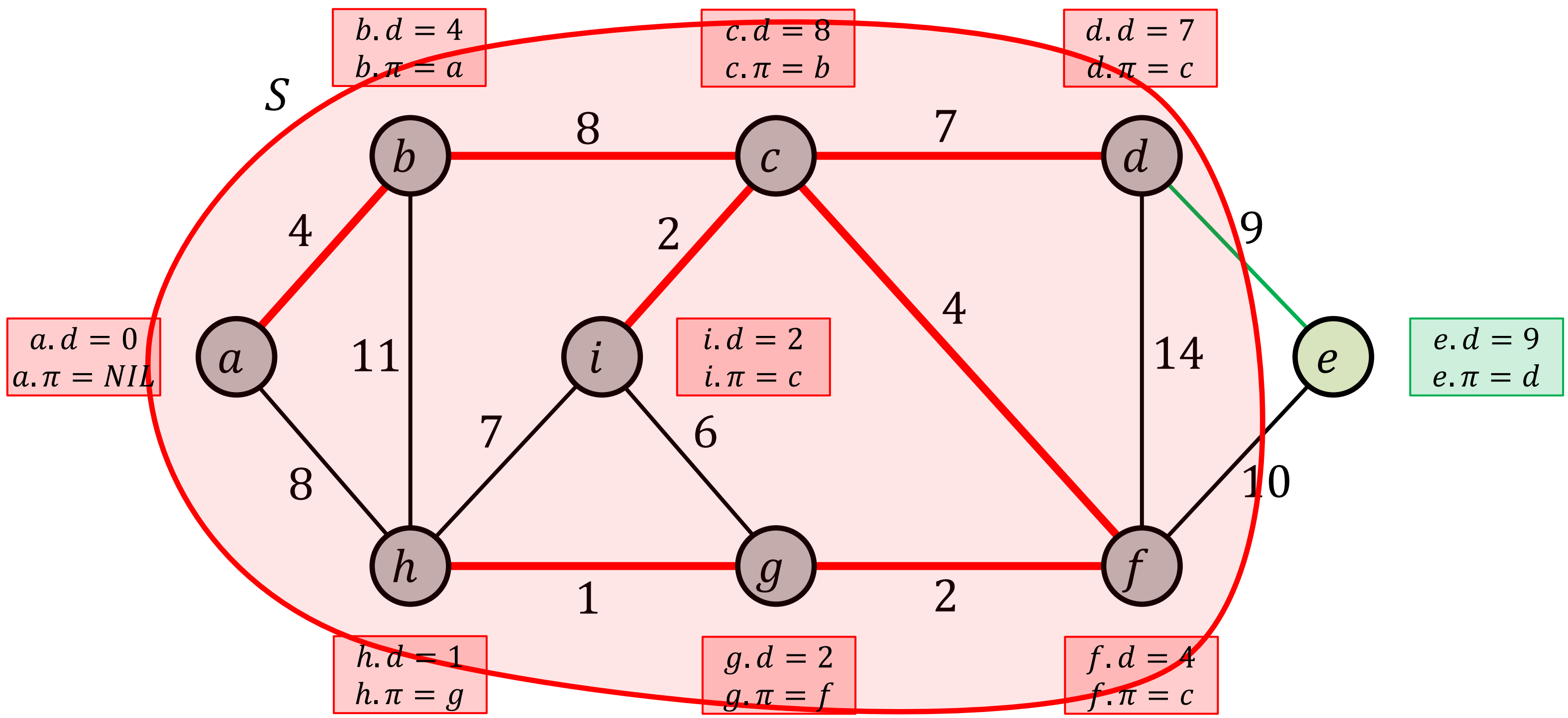
$S = \{a, b, c, i, f, g, h\}$

Cut = $(S, V - S)$

(c, d) is the light edge crossing the cut

MST: Prim's Algorithm

Step $8'$: update neighbors of d

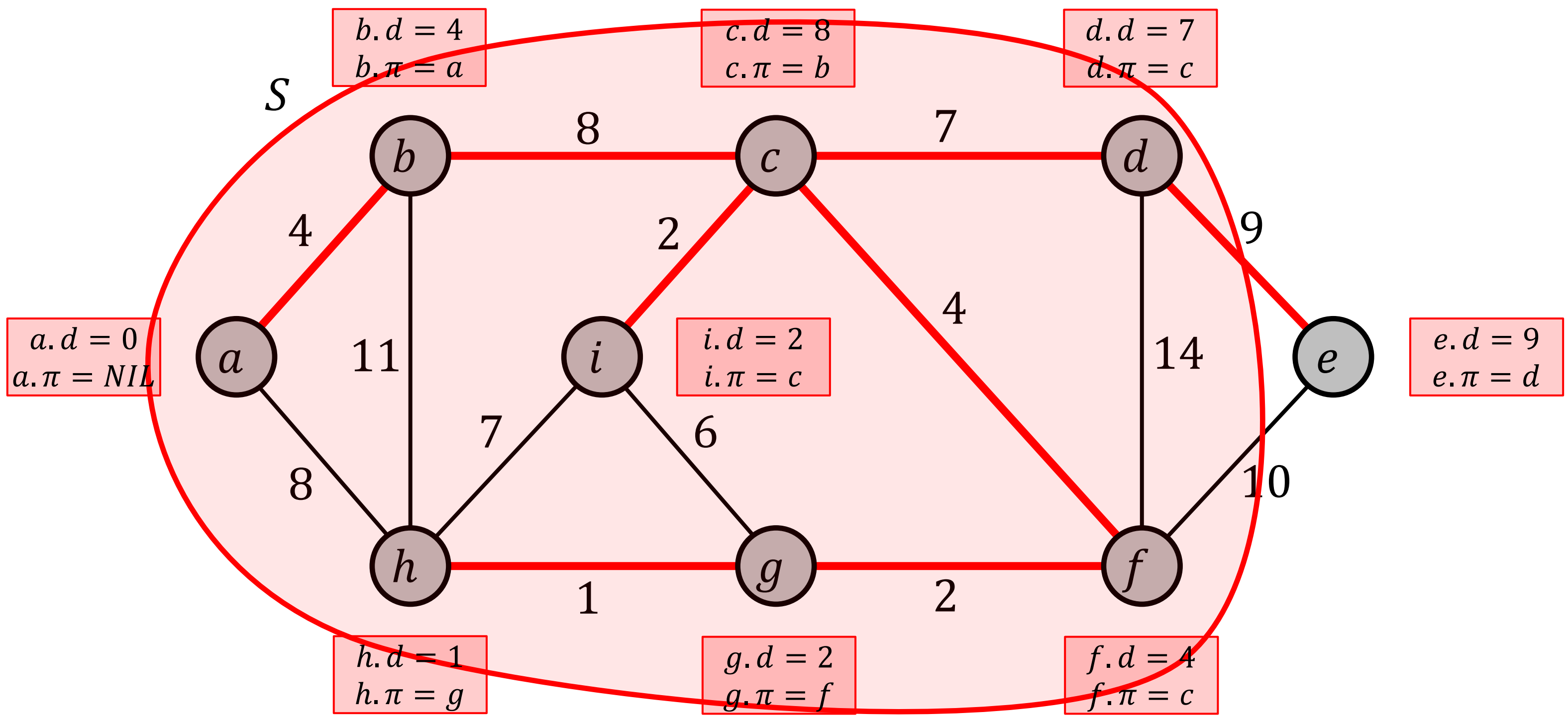


$S = \{a, b, c, i, f, g, h, d\}$

Cut = $(S, V - S)$

MST: Prim's Algorithm

Step 9: add vertex e through edge (d, e)



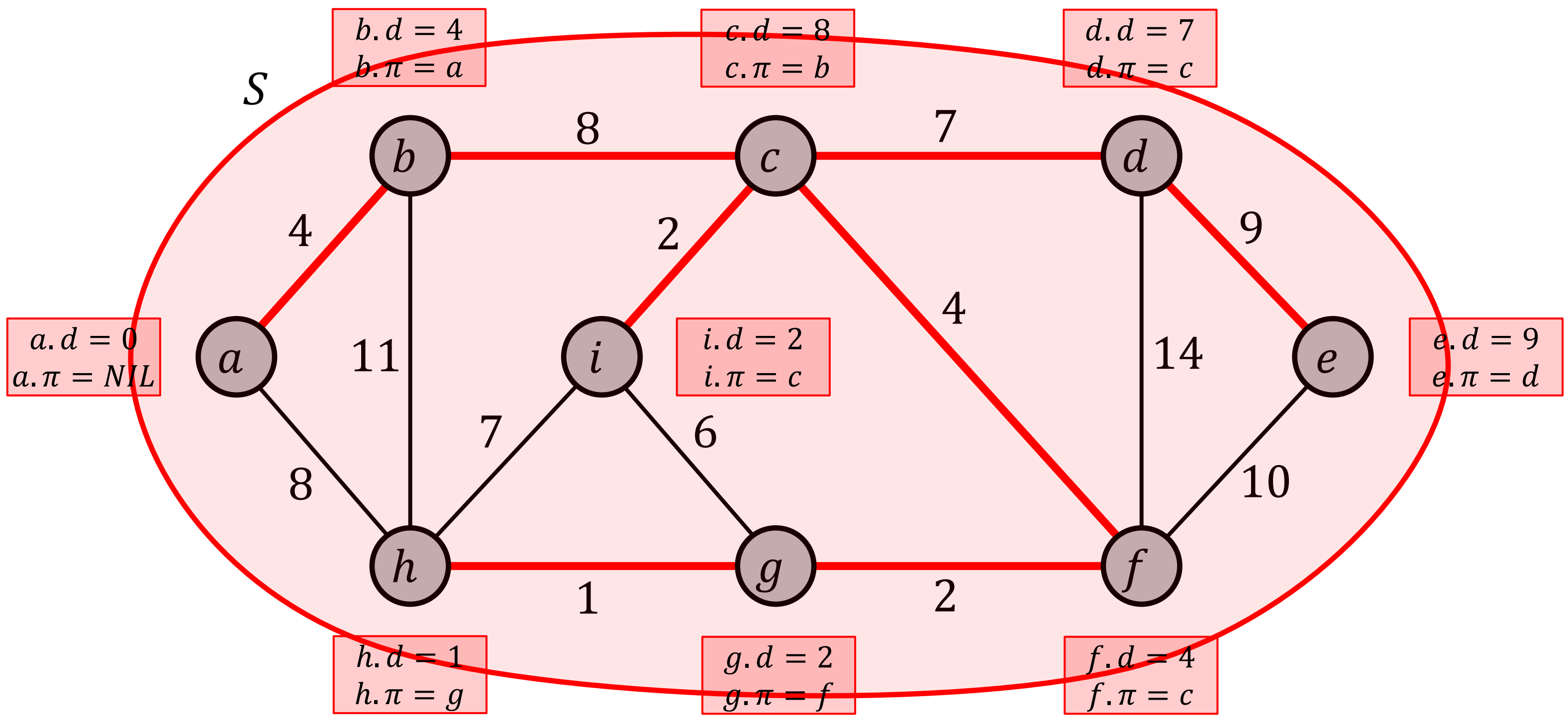
$S = \{a, b, c, i, f, g, h, d\}$

Cut = $(S, V - S)$

(d, e) is the light edge crossing the cut

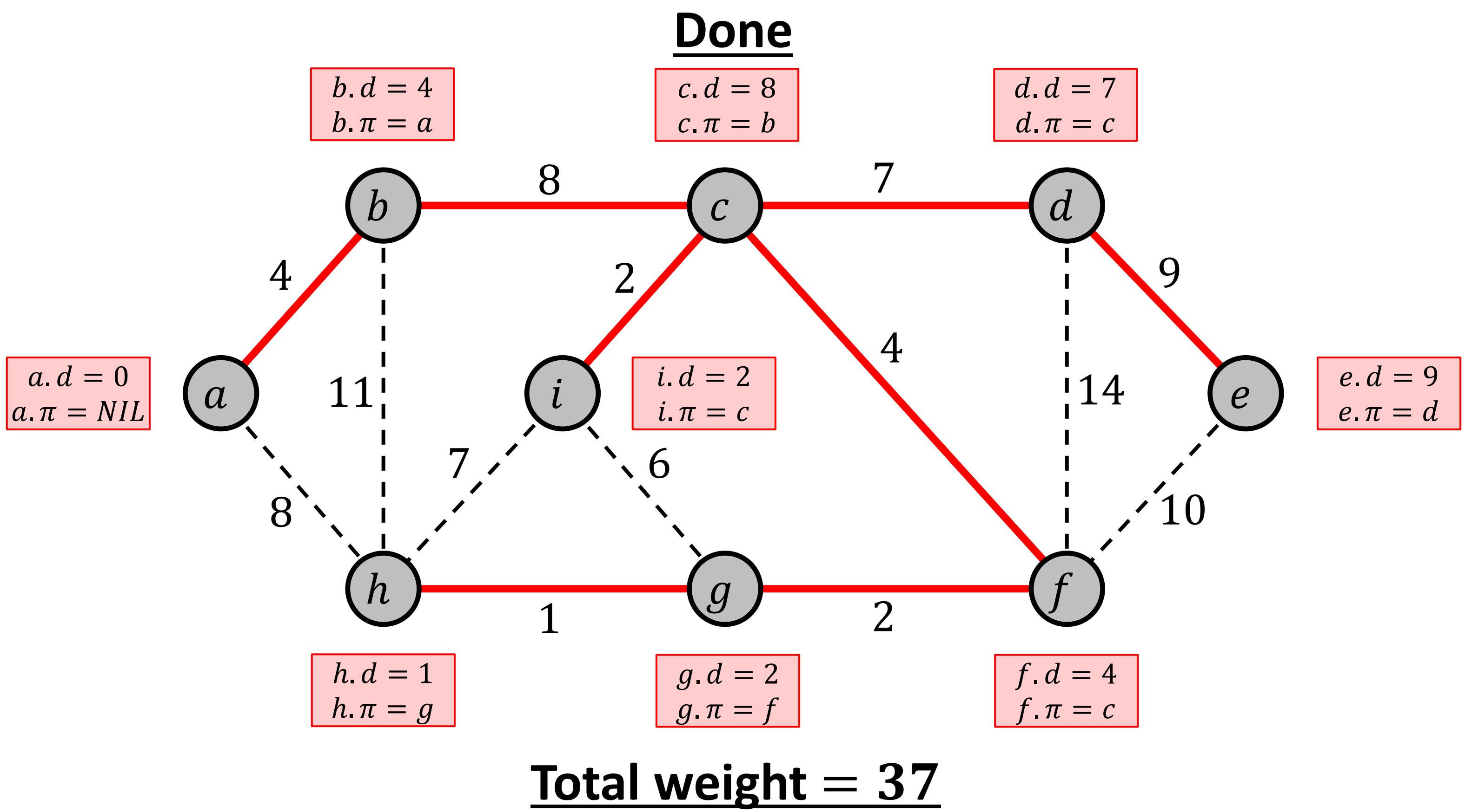
MST: Prim's Algorithm

Step 9′: update neighbors of e



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

MST: Prim's Algorithm



MST: Prim's Algorithm

```
MST-Prim ( G = (V, E), w, r )
    for each vertex v ∈ G.V do
        v.d ← ∞
        v.π ← NIL
    r.d ← 0
    Min-Heap Q ← ∅
    for each vertex v ∈ G.V do
        INSERT( Q, v )
    while Q ≠ ∅ do
        u ← EXTRACT-MIN( Q )
        for each (u, v) ∈ G.E do
            if v ∈ Q and w(u, v) < v.d then
                v.d ← w(u, v)
                v.π ← u
                DECREASE-KEY( Q, v, w(u, v) )
```

Let $n = |V|$ and $m = |E|$

\# *INSERTS* $= n$

\# *EXTRACT-MINS* $= n$

\# *DECREASE-KEYS* $\leq m$

Total cost

$$\leq n(cost_{Insert} + cost_{Extract-Min}) + m\left(cost_{Decrease-Key}\right)$$

MST: Prim's Algorithm

MST-Prim ($G = (V, E),\ w,\ r$)

- *for* each vertex $v \in G.V$ *do*
  - $v.d \leftarrow \infty$
  - $v.\pi \leftarrow NIL$
- $r.d \leftarrow 0$
- Min-Heap $Q \leftarrow \emptyset$
- *for* each vertex $v \in G.V$ *do*
  - INSERT(Q, v)
- *while* $Q \neq \emptyset$ *do*
  - $u \leftarrow$ EXTRACT-MIN(Q)
  - *for* each $(u, v) \in G.E$ *do*
    - *if* $v \in Q$ *and* $w(u, v) < v.d$ *then*
      - $v.d \leftarrow w(u, v)$
      - $v.\pi \leftarrow u$
      - DECREASE-KEY($Q,\ v,\ w(u, v)$)

Let $n = |V|$ and $m = |E|$

For Binary Heap (worst-case costs):

$$cost_{Insert} = O(\log n)$$
$$cost_{Extract-Min} = O(\log n)$$
$$cost_{Decrease-Key} = O(\log n)$$

∴ Total cost (worst-case)

$$= O\big((m + n)\log n\big)$$

MST: Prim's Algorithm

```
MST-Prim ( G = (V, E), w, r )
    for each vertex v ∈ G.V do
        v.d ← ∞
        v.π ← NIL
    r.d ← 0
    Min-Heap Q ← ∅
    for each vertex v ∈ G.V do
        INSERT( Q, v )
    while Q ≠ ∅ do
        u ← EXTRACT-MIN( Q )
        for each (u, v) ∈ G.E do
            if v ∈ Q and w(u, v) < v.d then
                v.d ← w(u, v)
                v.π ← u
                DECREASE-KEY( Q, v, w(u, v) )
```

Let $n = |V|$ and $m = |E|$

For Fibonacci Heap (amortized):

$$cost_{Insert} = O(1)$$
$$cost_{Extract-Min} = O(\log n)$$
$$cost_{Decrease-Key} = O(1)$$

∴ Total cost (amortized)

$$= O(m + n \log n)$$

A Disjoint-Set Data Structure
(for Kruskal's MST Algorithm)

A *disjoint-set data structure* maintains a collection of disjoint dynamic sets. Each set is identified by a *representative* which must be a member of the set.

The collection is maintained under the following operations:

MAKE-SET(x): create a new set $\{x\}$ containing only element x. Element x becomes the representative of the set.

FIND(x): returns a pointer to the representative of the set containing x

UNION(x, y): replace the dynamic sets S_x and S_y containing x and y, respectively, with the set $S_x \cup S_y$

A Disjoint-Set Data Structure (union by rank)

MAKE-SET (x)

1. $\pi(x) \leftarrow x$
2. $rank(x) \leftarrow 0$

LINK (x, y)

1. *if* $rank(x) > rank(y)$ *then* $\pi(y) \leftarrow x$
2. *else* $\pi(x) \leftarrow y$
3. *if* $rank(x) = rank(y)$ *then* $rank(y) \leftarrow rank(y) + 1$

UNION (x, y)

1. LINK (FIND (x), FIND (y))

FIND (x)

1. *if* $x \neq \pi(x)$ *then return* FIND ($\pi(x)$)
2. *else return* x

A Disjoint-Set Data Structure (union by rank)

THEOREM: A sequence of N MAKE-SET, UNION and FIND operations of which exactly n $(\leq N)$ are MAKE-SET operations takes $\mathrm{O}(N \log n)$ time to execute.

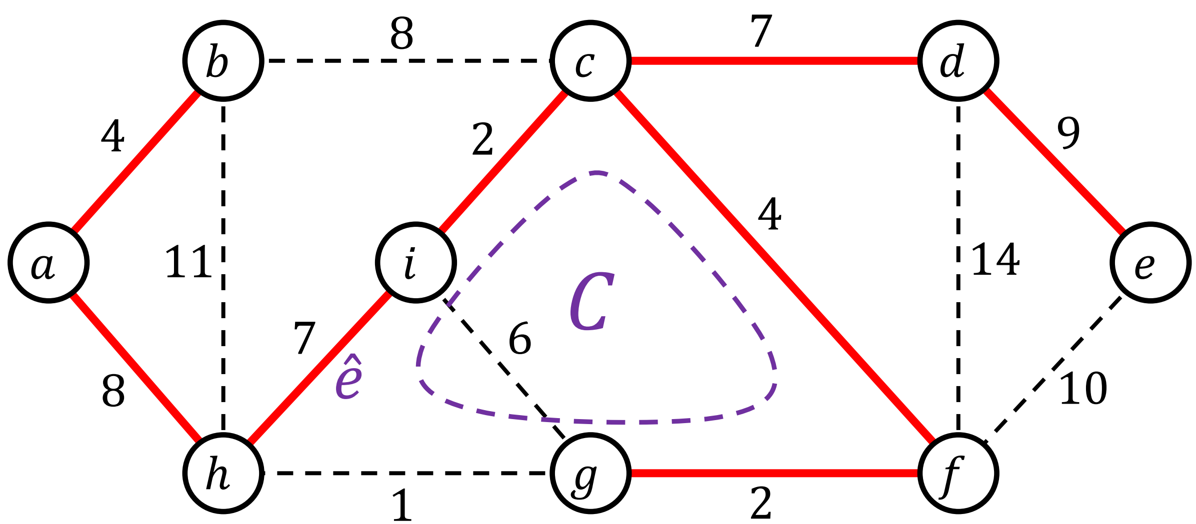
MST: Another Useful Lemma

LEMMA: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let C be a cycle of G with a unique heaviest edge $\hat{e} \in E$. Then $\hat{e}$ cannot be part of any MST of G.

MST: Another Useful Lemma

LEMMA: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let C be a cycle of G with a unique heaviest edge $\hat{e} \in E$. Then $\hat{e}$ cannot be part of any MST of G.

PROOF:

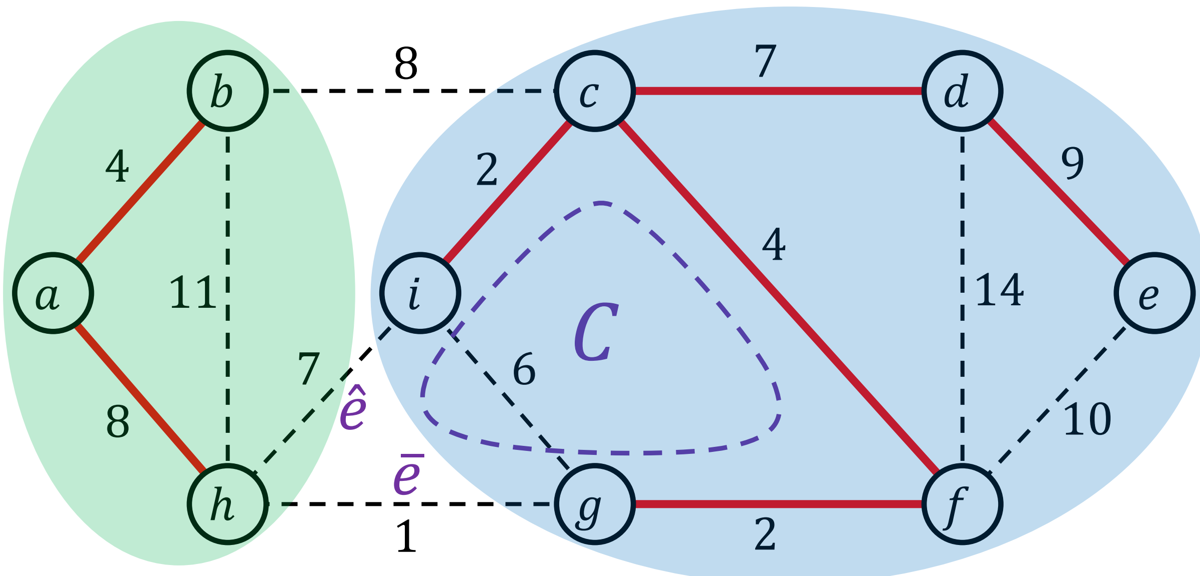


Let $\hat{e}$ be part of some spanning tree T of G.

MST: Another Useful Lemma

Lemma: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let C be a cycle of G with a unique heaviest edge $\hat{e} \in E$. Then $\hat{e}$ cannot be part of any MST of G.

Proof:



Let $\hat{e}$ be part of some spanning tree T of G.

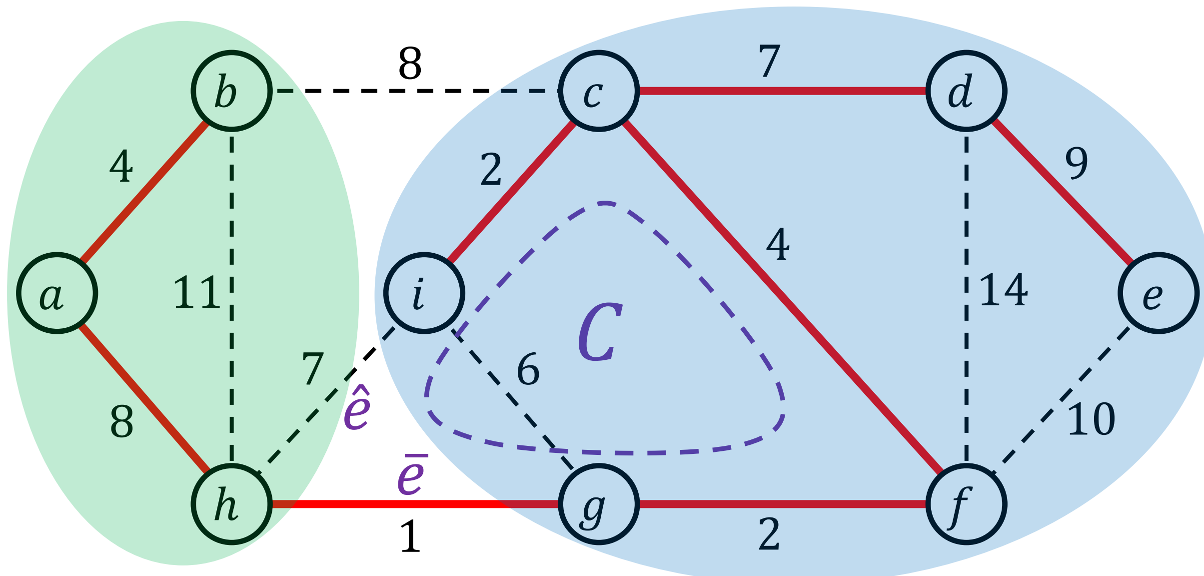
Let's remove $\hat{e}$ from T. Then T will get split into two components.

There must be an edge $\bar{e} \in C$ that reconnects the two components.

MST: Another Useful Lemma

LEMMA: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let C be a cycle of G with a unique heaviest edge $\hat{e} \in E$. Then $\hat{e}$ cannot be part of any MST of G.

PROOF:



Let's add $\bar{e}$ to $T - \{\hat{e}\}$, and let $T' = T \cup \{\bar{e}\} - \{\hat{e}\}$.

Then T' is a spanning tree of G.

Since $w(\hat{e}) > w(\bar{e})$, we get, $w(T') = w(T) - w(\hat{e}) + w(\bar{e}) < w(T)$.

So, T cannot be an MST of G!

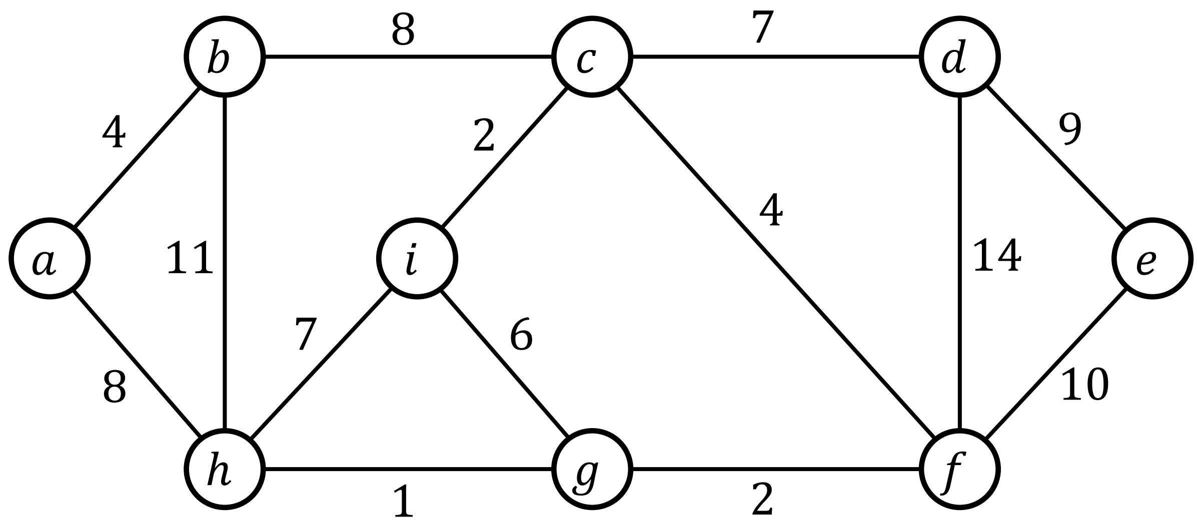
MST: Kruskal's Algorithm

MST-Kruskal ($G = (V, E)$, w)

$A \leftarrow \emptyset$
for each vertex $v \in G.V$ *do*
    MAKE-SET(v)
sort the edges of $G.E$ into nondecreasing order by weight w
for each edge $(u, v) \in G.E$ taken in nondecreasing order by weight *do*
    if *FIND*(u) $\neq$ *FIND*(v) *then*
        $A \leftarrow A \cup \{(u, v)\}$
        UNION(u, v)
return A

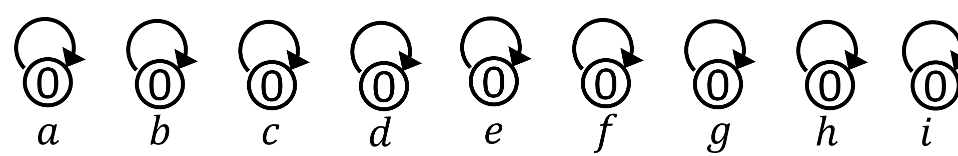
MST: Kruskal's Algorithm

Initial State:



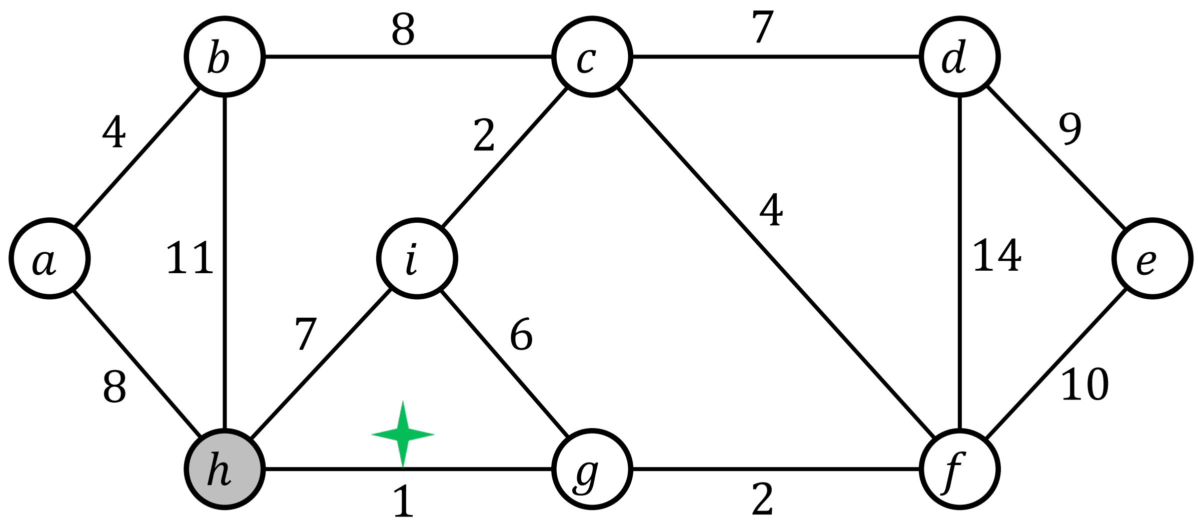
Disjoint-Set Data Structure (union by rank only) :

MAKE-SET(x), $x \in \{a, b, c, d, e, f, g, h, i\}$



MST: Kruskal's Algorithm

(1) edge (h, g):



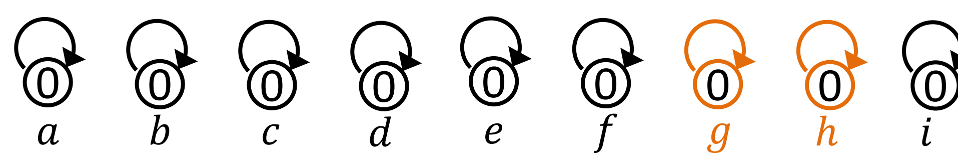
$$S = \{component\ (connected\ through\ red\ edges)\ containing\ h\} = \{h\}$$

Cut = $(S, V - S)$

(h, g) is the light edge crossing the cut

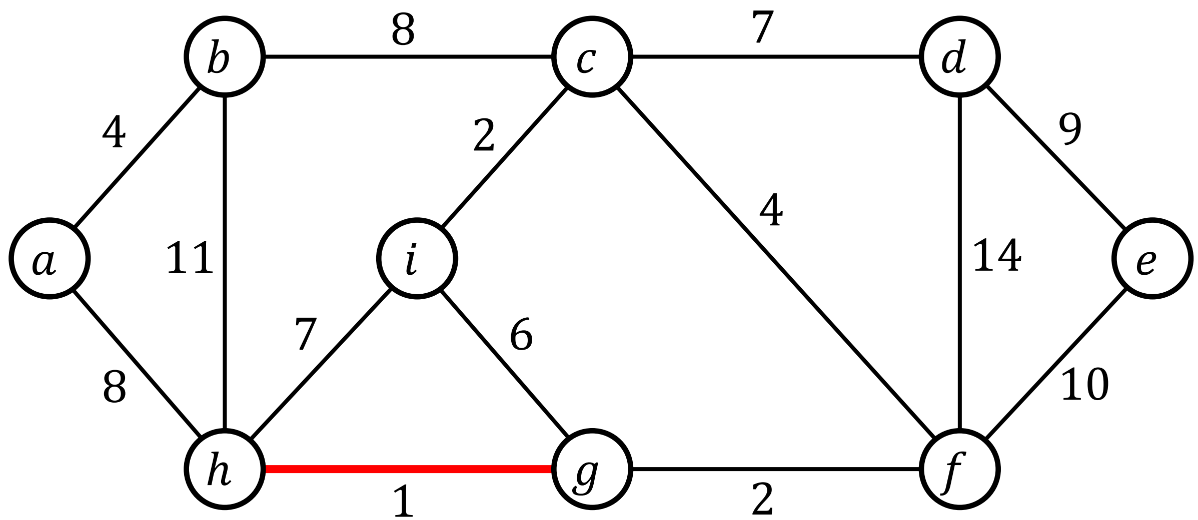
Disjoint-Set Data Structure (union by rank only) :

FIND(h) returns h, FIND(g) returns g



MST: Kruskal's Algorithm

(1) edge (h, g):



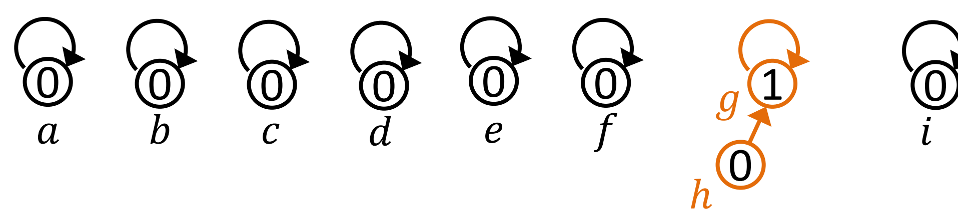
$S = \{component\ (connected\ through\ red\ edges)\ containing\ h\} = \{h\}$

Cut = $(S, V - S)$

(h, g) is the light edge crossing the cut

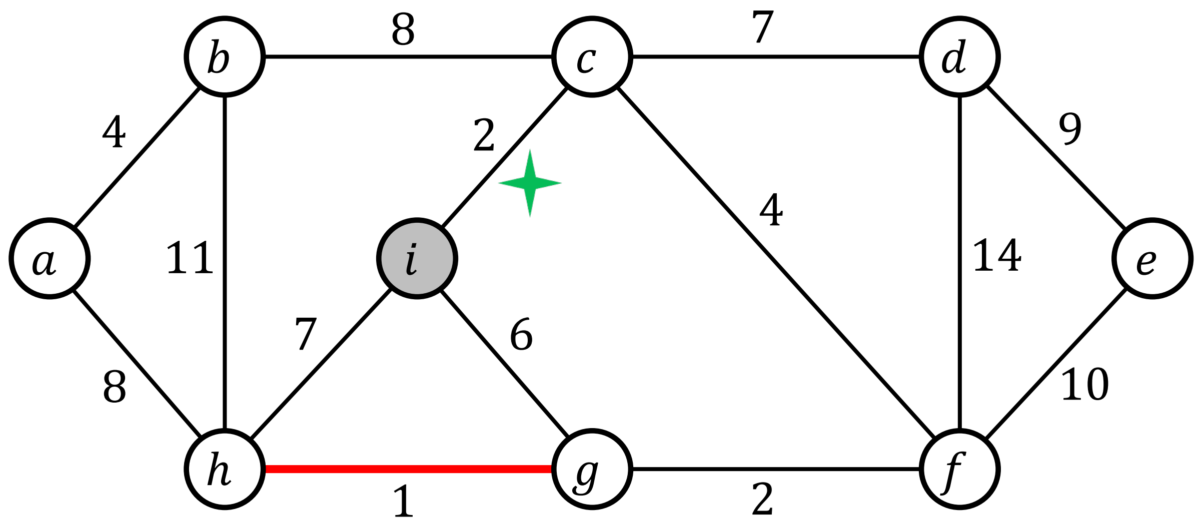
Disjoint-Set Data Structure (union by rank only) :

UNION(h, g)



MST: Kruskal's Algorithm

(2) edge (i, c):



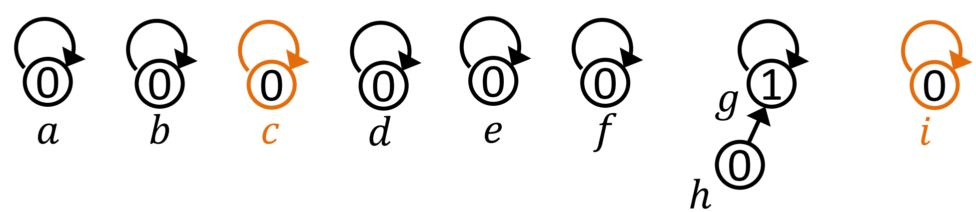
$$S = \{component\ (connected\ through\ red\ edges)\ containing\ i\} = \{i\}$$

Cut = $(S, V - S)$

(i, c) is the light edge crossing the cut

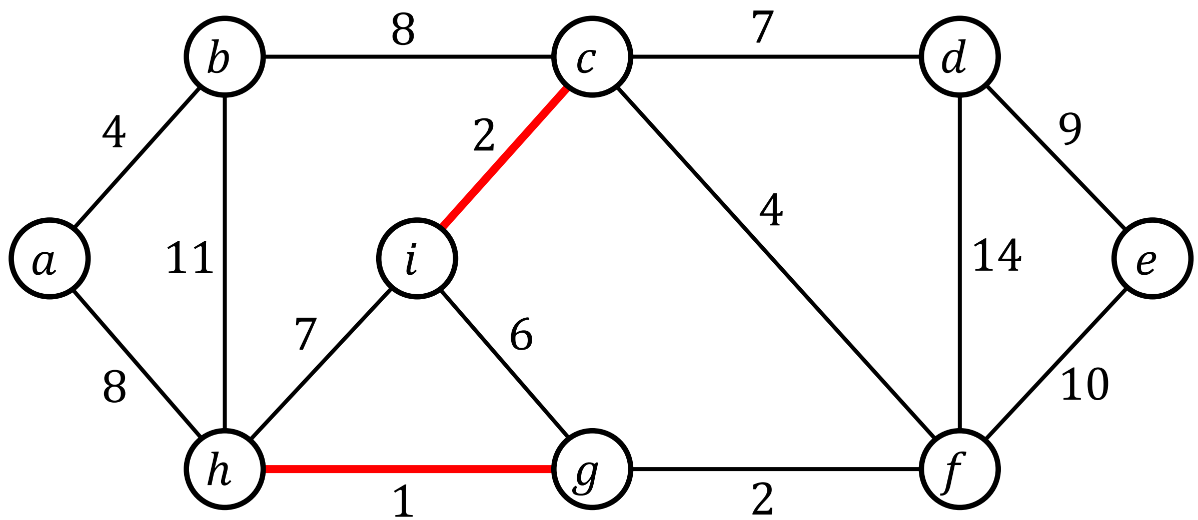
Disjoint-Set Data Structure (union by rank only) :

FIND(i) returns i, FIND(c) returns c



MST: Kruskal's Algorithm

(2) edge (i, c):



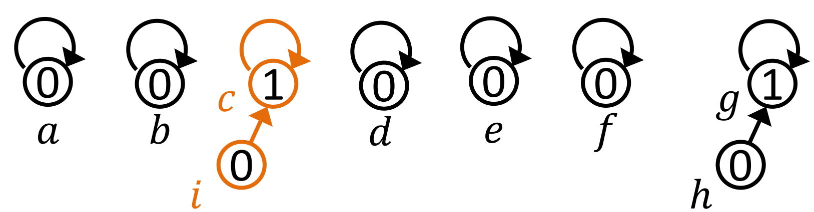
$S = \{component\ (connected\ through\ red\ edges)\ containing\ i\} = \{i\}$

Cut = $(S, V - S)$

(i, c) is the light edge crossing the cut

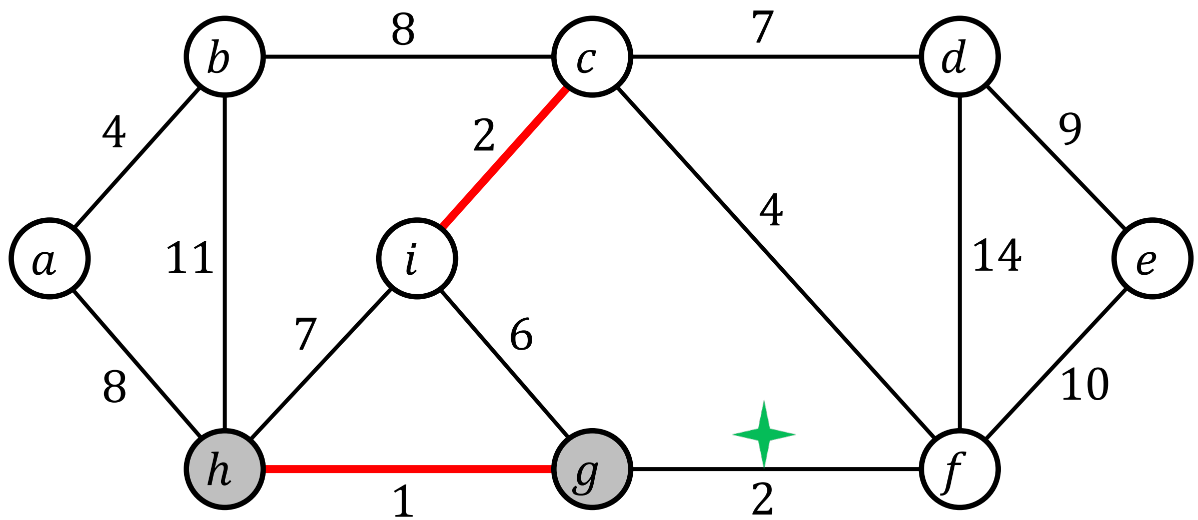
Disjoint-Set
Data Structure
(union by rank only) :

UNION(i, c)



MST: Kruskal's Algorithm

(3) edge (g, f):



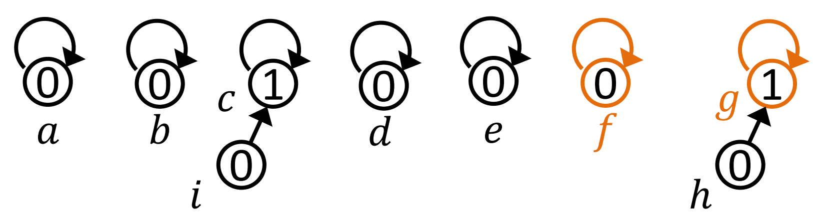
$S = \{component\ (connected\ through\ red\ edges)\ containing\ g\} = \{h, g\}$

Cut = $(S, V - S)$

(g, f) is the light edge crossing the cut

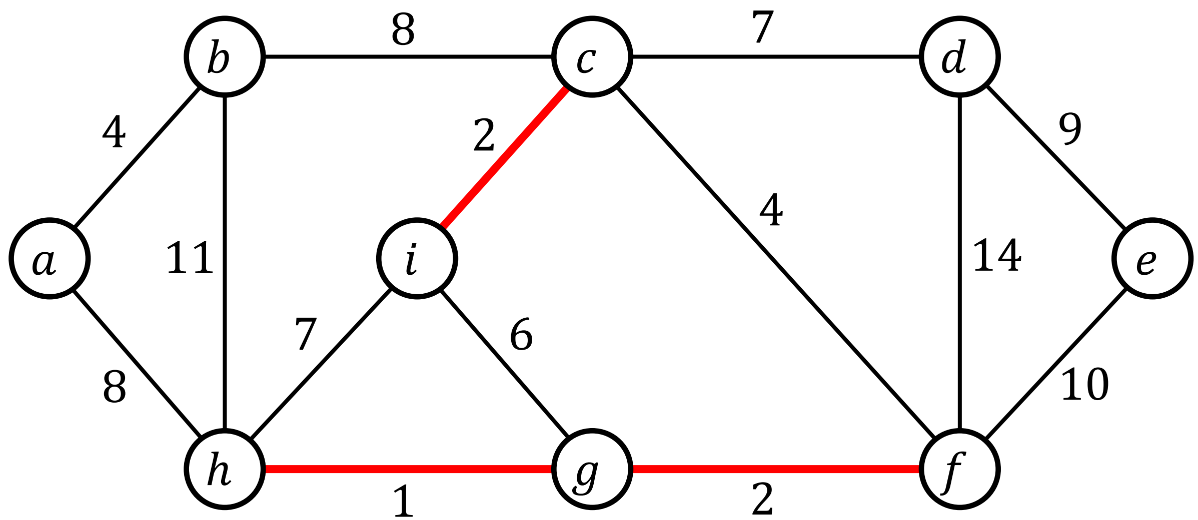
Disjoint-Set Data Structure (union by rank only) :

FIND(g) returns g, FIND(f) returns f



MST: Kruskal's Algorithm

(3) edge (g, f):



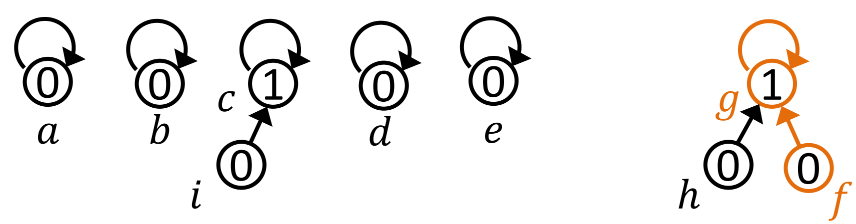
$S = \{component\ (connected\ through\ red\ edges)\ containing\ g\} = \{h, g\}$

Cut = $(S, V - S)$

(g, f) is the light edge crossing the cut

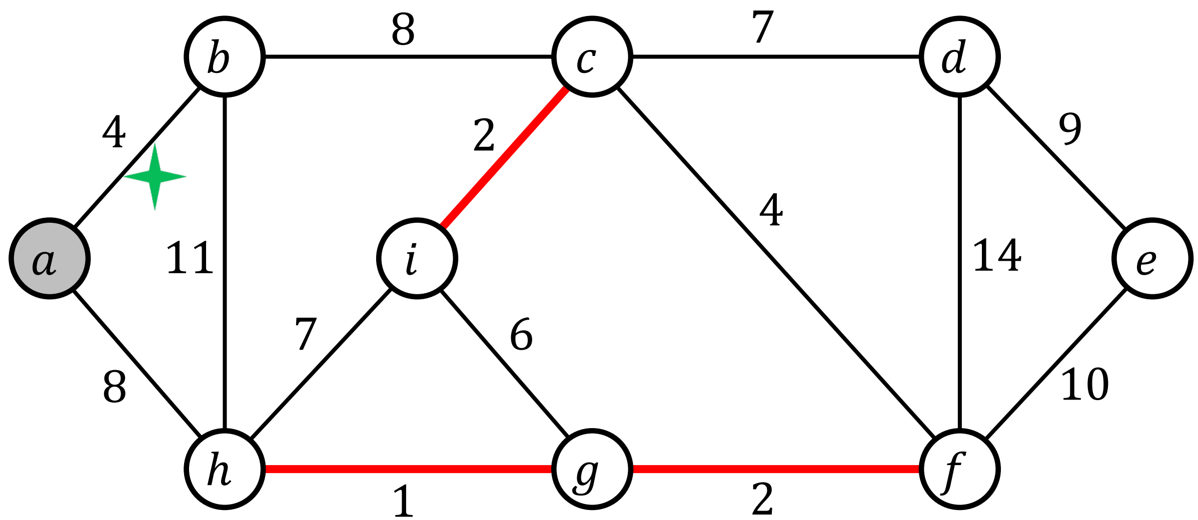
Disjoint-Set Data Structure (union by rank only) :

UNION(g, f)



MST: Kruskal's Algorithm

(4) edge (a, b):



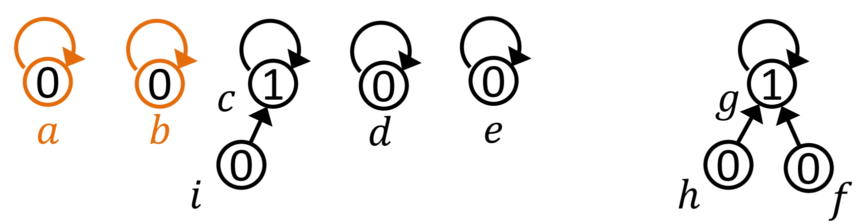
$$S = \{component\ (connected\ through\ red\ edges)\ containing\ a\} = \{a\}$$

Cut = $(S, V - S)$

(a, b) is the light edge crossing the cut

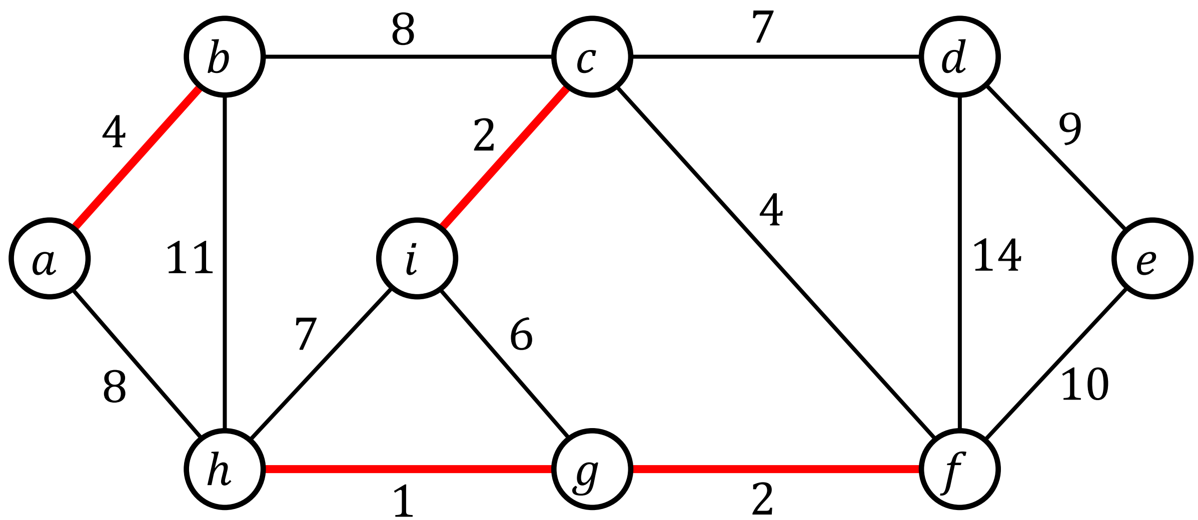
Disjoint-Set Data Structure (union by rank only) :

FIND(a) returns a, FIND(b) returns b



MST: Kruskal's Algorithm

(4) edge (a, b):



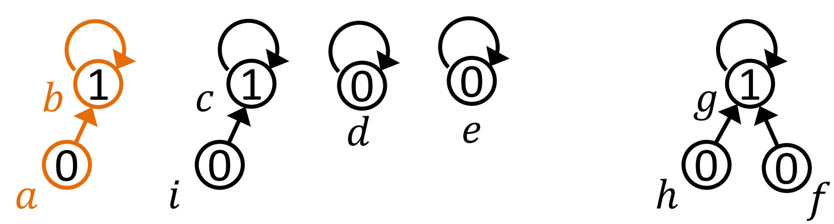
$S = \{component\ (connected\ through\ red\ edges)\ containing\ a\} = \{a\}$

Cut = $(S, V - S)$

(a, b) is the light edge crossing the cut

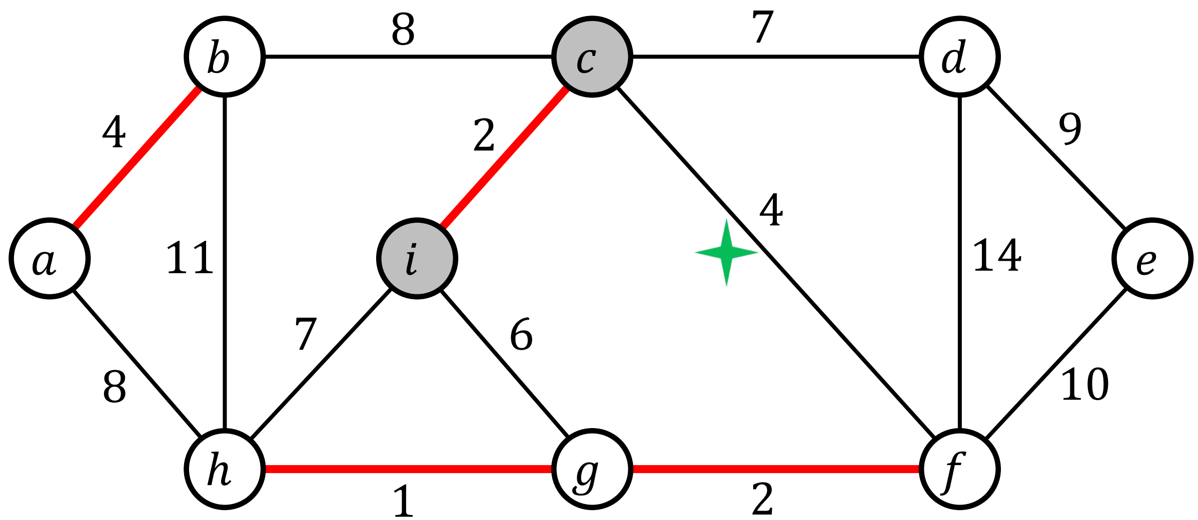
Disjoint-Set Data Structure (union by rank only) :

UNION(a, b)



MST: Kruskal's Algorithm

(5) edge (c, f):



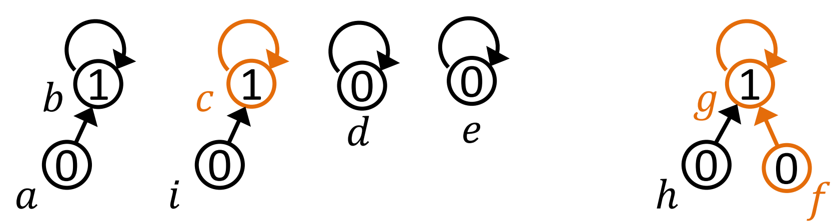
$$S = \{component\ (connected\ through\ red\ edges)\ containing\ c\} = \{c, i\}$$

Cut = $(S, V - S)$

(c, f) is the light edge crossing the cut

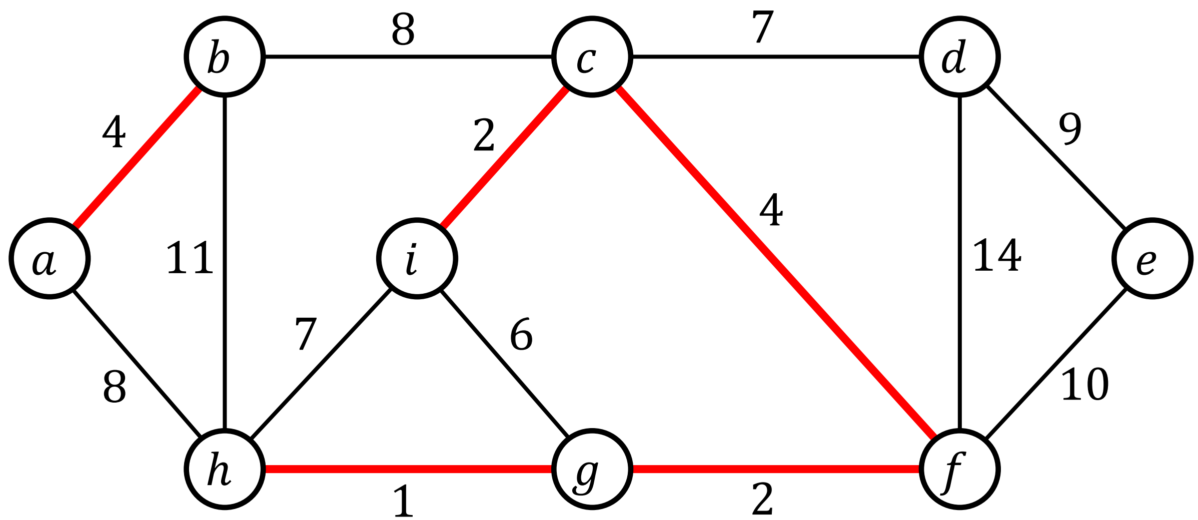
Disjoint-Set Data Structure (union by rank only) :

FIND(c) returns c, FIND(f) returns g



MST: Kruskal's Algorithm

(5) edge (c, f):



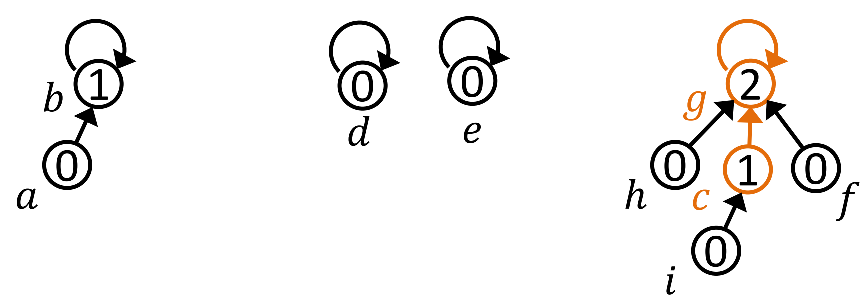
$S = \{component\ (connected\ through\ red\ edges)\ containing\ c\} = \{c, i\}$

Cut = $(S, V - S)$

(c, f) is the light edge crossing the cut

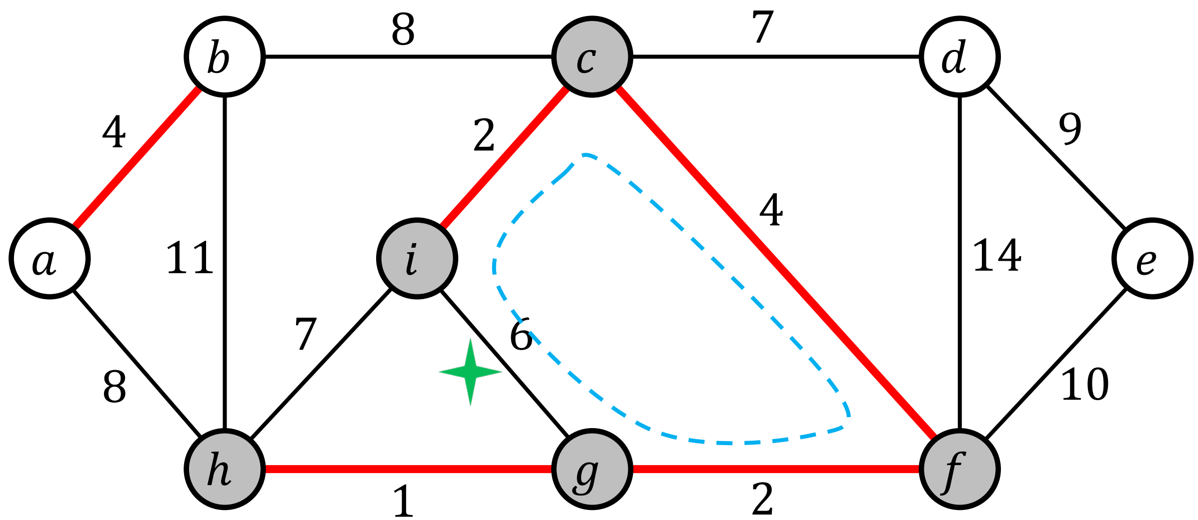
Disjoint-Set Data Structure (union by rank only) :

UNION(c, f)



MST: Kruskal's Algorithm

(6) edge (i, g):



$$S = \{component\ (connected\ through\ red\ edges)\ containing\ i\} = \{i, c, f, g, h\}$$

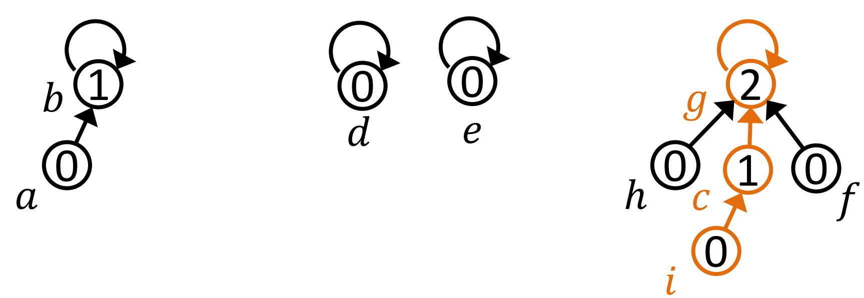
Cut = $(S, V - S)$

(i, g) creates a cycle by connecting two nodes of S, and it is the heaviest edge on that cycle

---

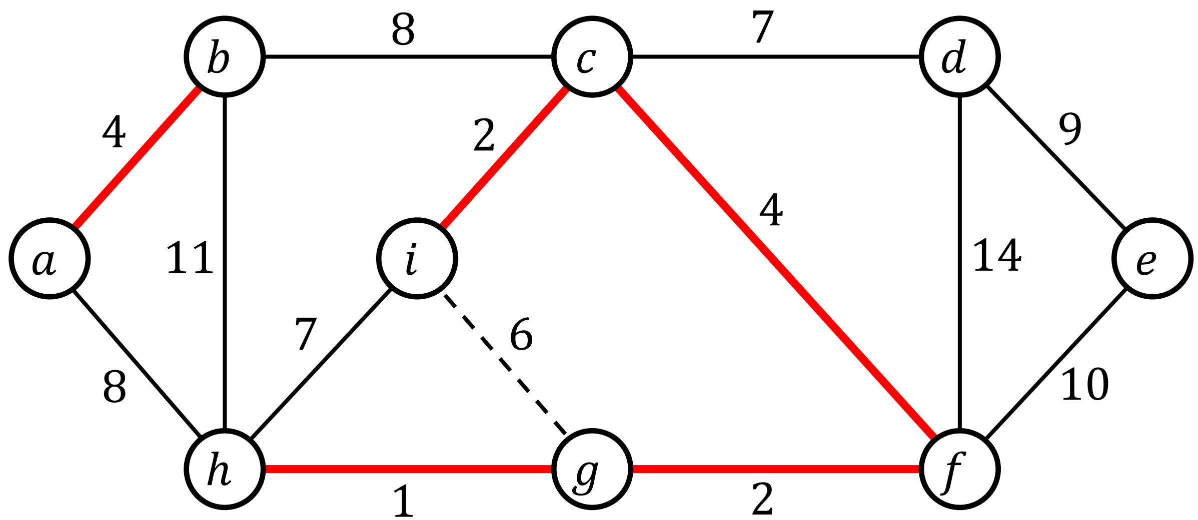
Disjoint-Set Data Structure (union by rank only) :

FIND(i) returns g, FIND(g) returns g



MST: Kruskal's Algorithm

(6) edge (i, g):



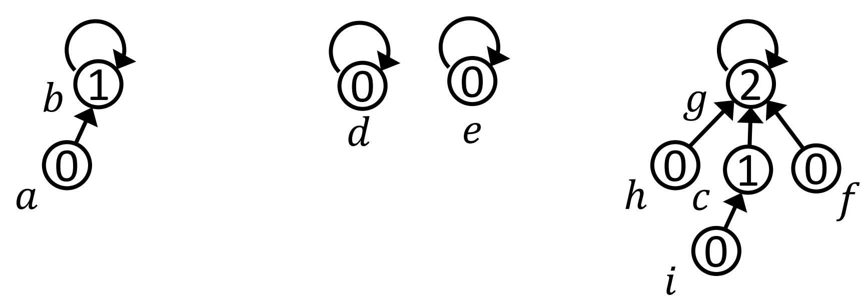
$S = \{component\ (connected\ through\ red\ edges)\ containing\ i\} = \{i, c, f, g, h\}$

Cut = $(S, V - S)$

(i, g) creates a cycle by connecting two nodes of S, and it is the heaviest edge on that cycle

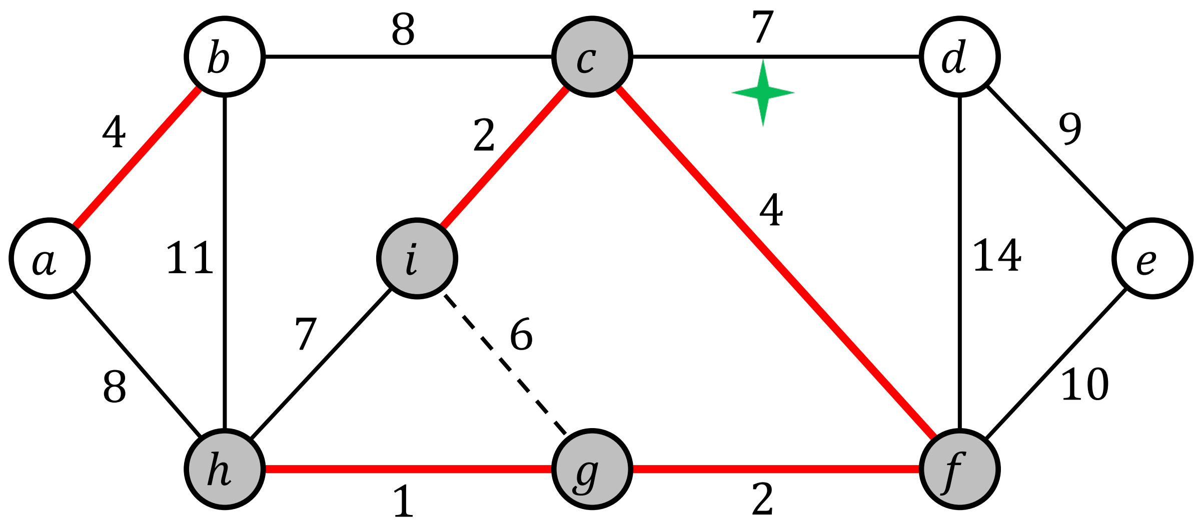
---

Disjoint-Set Data Structure (union by rank only) :



MST: Kruskal's Algorithm

(7) edge (c, d):



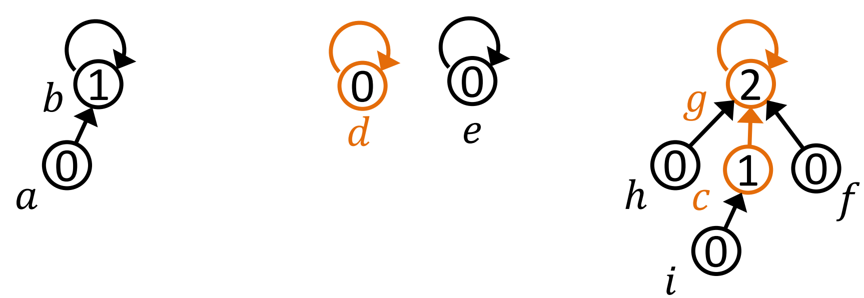
$$S = \{component\ (connected\ through\ red\ edges)\ containing\ c\} = \{i, c, f, g, h\}$$

Cut = $(S, V - S)$

(c, d) is the light edge crossing the cut

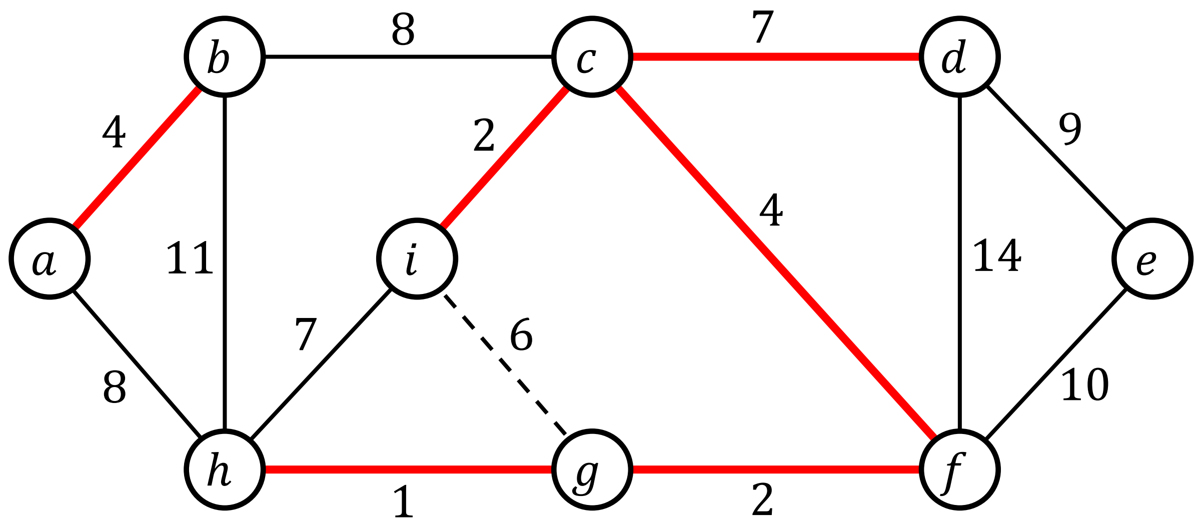
Disjoint-Set Data Structure (union by rank only) :

FIND(c) returns g, FIND(d) returns d



MST: Kruskal's Algorithm

(7) edge (c, d):



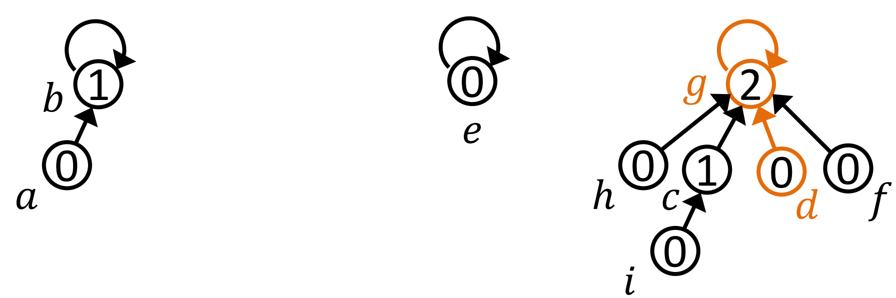
$S = \{component\ (connected\ through\ red\ edges)\ containing\ c\} = \{i, c, f, g, h\}$

Cut = $(S, V - S)$

(c, d) is the light edge crossing the cut

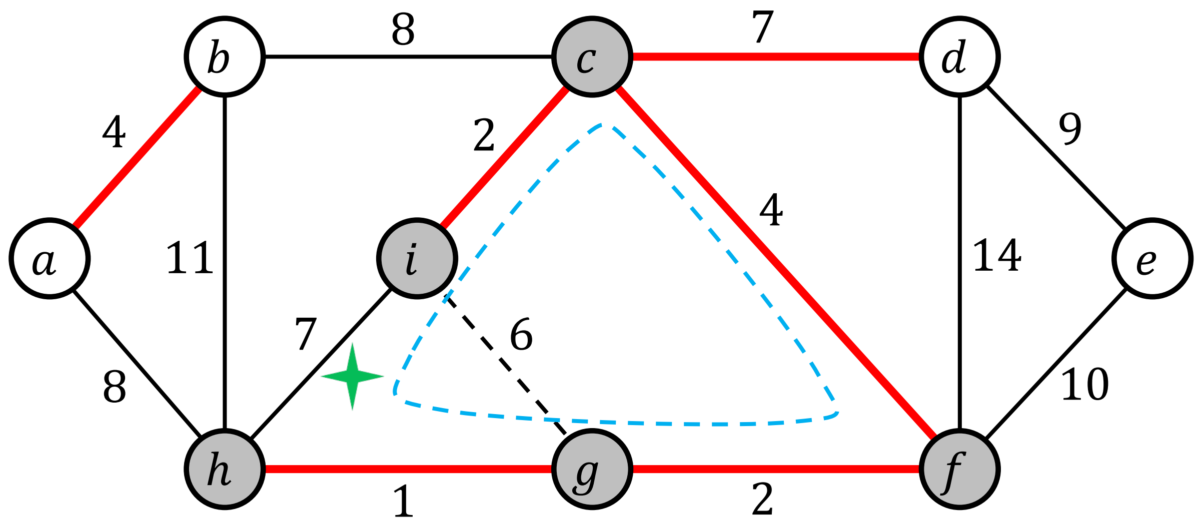
Disjoint-Set Data Structure (union by rank only) :

UNION(c, d)



MST: Kruskal's Algorithm

(8) edge (i, h):



$$S = \{component\ (connected\ through\ red\ edges)\ containing\ i\} = \{i, c, f, g, h\}$$

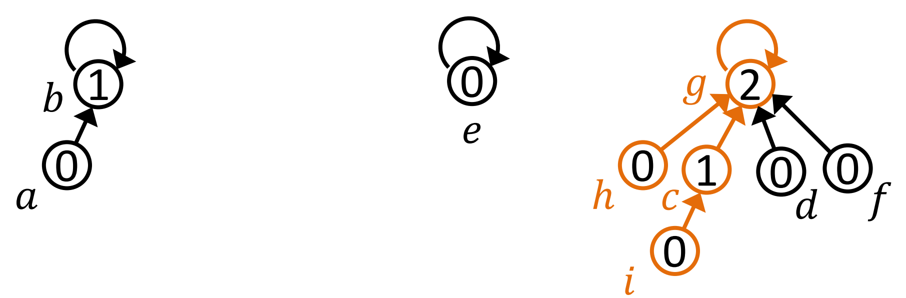
Cut = $(S, V - S)$

(i, h) creates a cycle by connecting two nodes of S, and it is the heaviest edge on that cycle

---

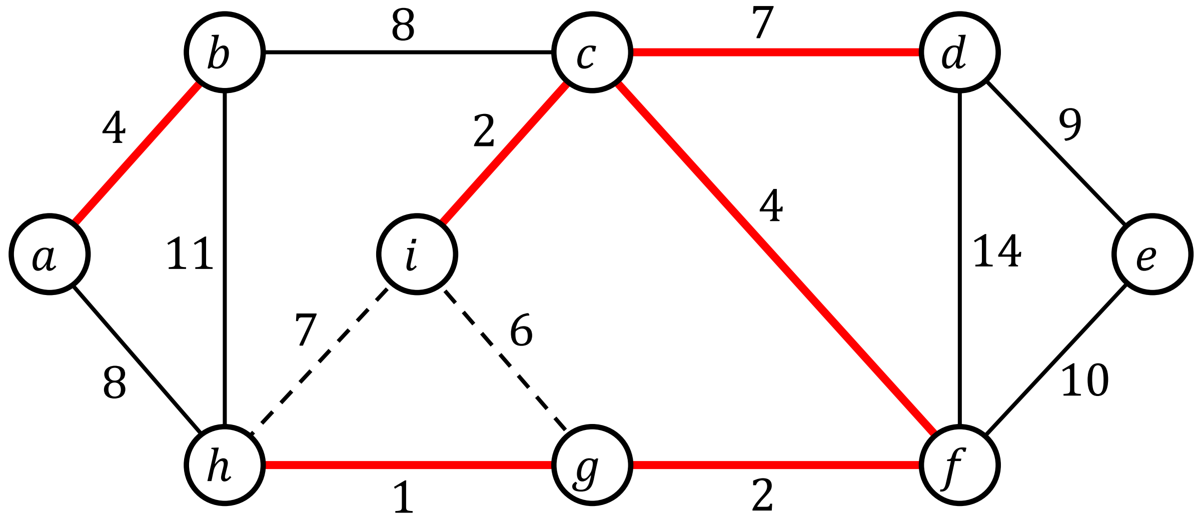
Disjoint-Set Data Structure (union by rank only) :

FIND(i) returns g, FIND(h) returns g



MST: Kruskal's Algorithm

(8) edge $(\boldsymbol{i}, \boldsymbol{h})$:



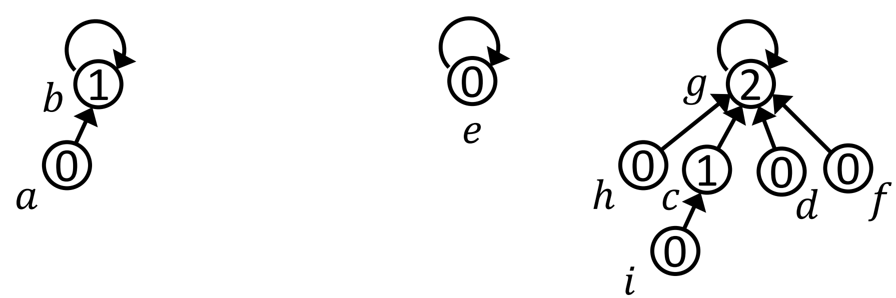
$S = \{component\ (connected\ through\ red\ edges)\ containing\ i\} = \{i, c, f, g, h\}$

Cut = $(S, V - S)$

(i, h) creates a cycle by connecting two nodes of S, and it is the heaviest edge on that cycle

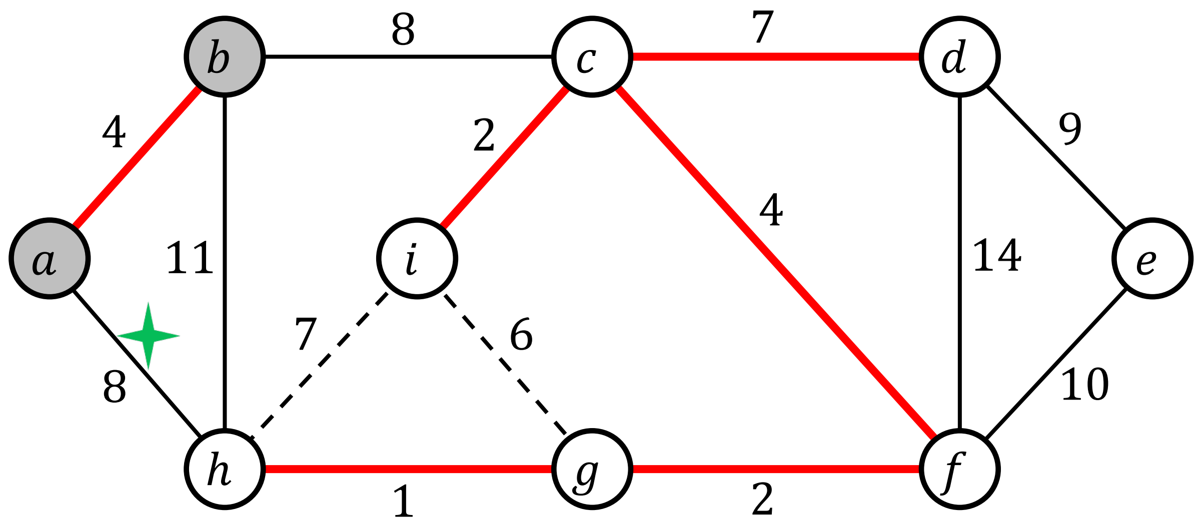
---

Disjoint-Set Data Structure (union by rank only) :



MST: Kruskal's Algorithm

(9) edge (a, h):



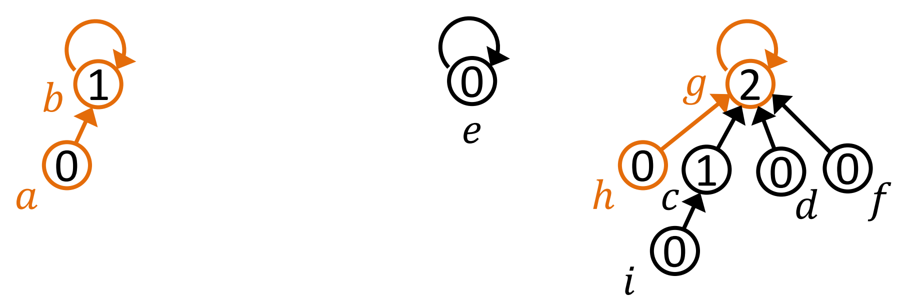
$$S = \{component\ (connected\ through\ red\ edges)\ containing\ a\} = \{a, b\}$$

Cut = $(S, V - S)$

(a, h) is a light edge crossing the cut

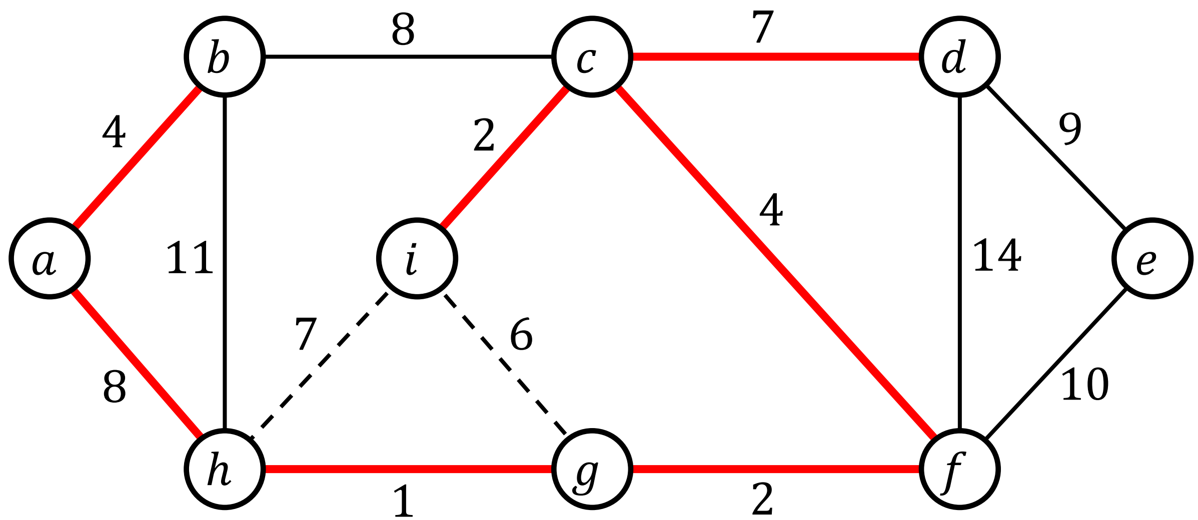
Disjoint-Set Data Structure (union by rank only) :

FIND(a) returns b, FIND(h) returns g



MST: Kruskal's Algorithm

(9) edge (a, h):



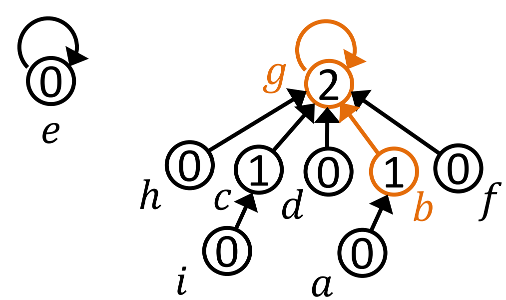
$S = \{component\ (connected\ through\ red\ edges)\ containing\ a\} = \{a, b\}$

Cut = $(S, V - S)$

(a, h) is a light edge crossing the cut

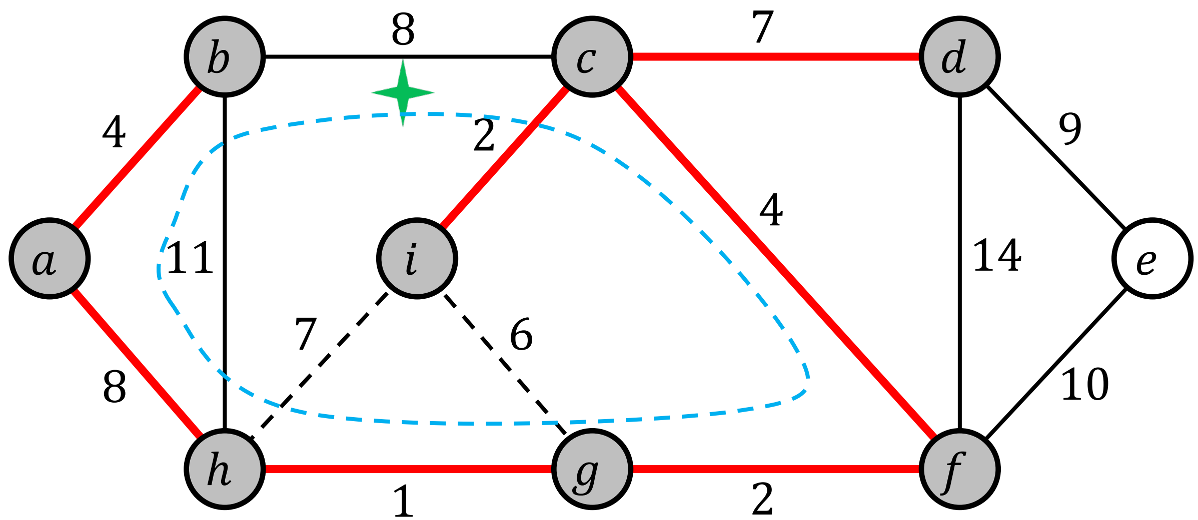
Disjoint-Set Data Structure (union by rank only) :

UNION(a, h)



MST: Kruskal's Algorithm

(10) edge (b, c):



$S = \{component\ (connected\ through\ red\ edges)\ containing\ b\} = \{a, b, c, d, f, g, h, i\}$

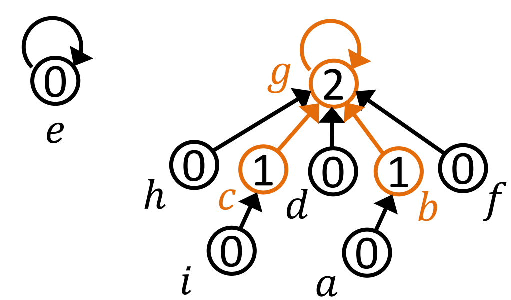
Cut = $(S, V - S)$

(b, c) creates a cycle by connecting two nodes of S, and it is a heaviest edge on that cycle

---

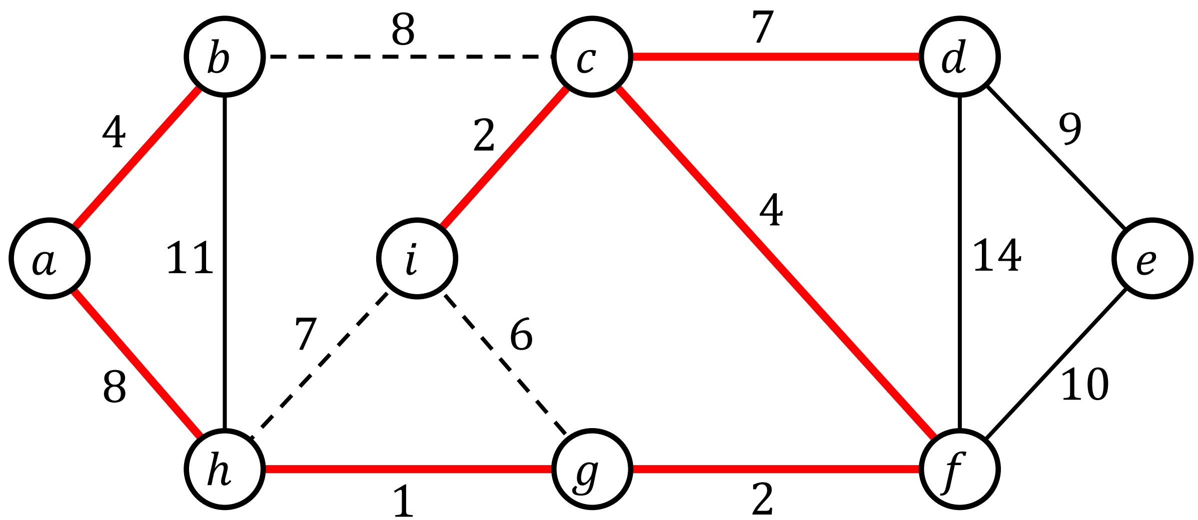
Disjoint-Set Data Structure (union by rank only) :

FIND(b) returns g, FIND(c) returns g



MST: Kruskal's Algorithm

(10) edge (b, c):



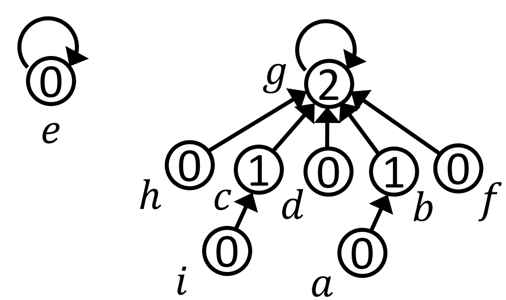
$S = \{component\ (connected\ through\ red\ edges)\ containing\ b\} = \{a, b, c, d, f, g, h, i\}$

Cut = $(S, V - S)$

(b, c) creates a cycle by connecting two nodes of S, and it is a heaviest edge on that cycle

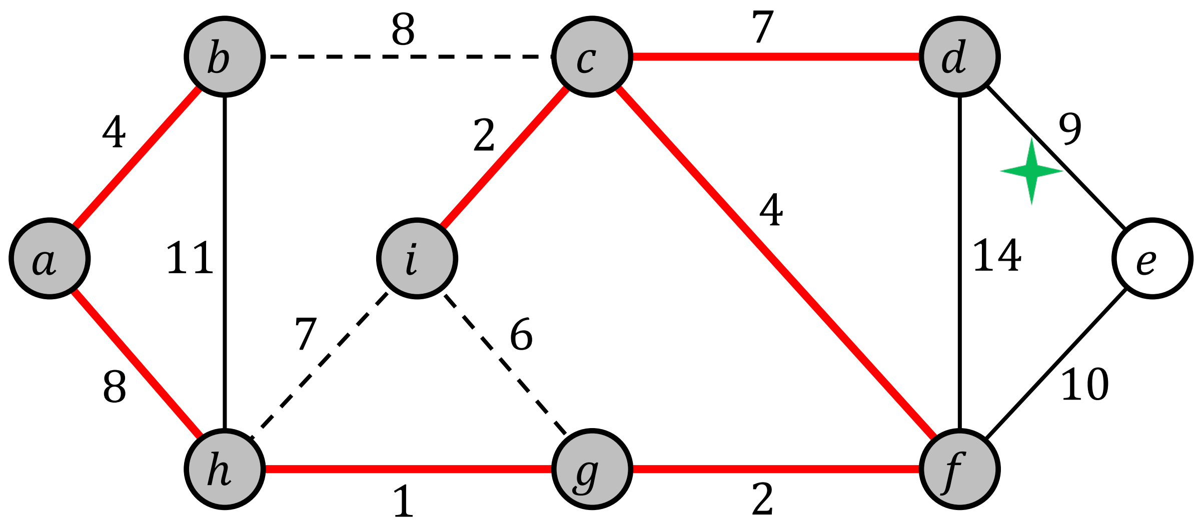
---

Disjoint-Set
Data Structure
(union by rank only) :



MST: Kruskal's Algorithm

(11) edge (d, e):



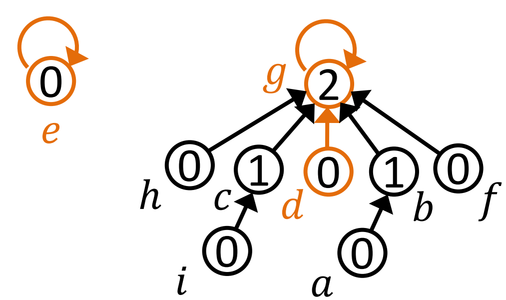
$S = \{component\ (connected\ through\ red\ edges)\ containing\ d\} = \{a, b, c, d, f, g, h, i\}$

Cut = $(S, V - S)$

(d, e) is a light edge crossing the cut

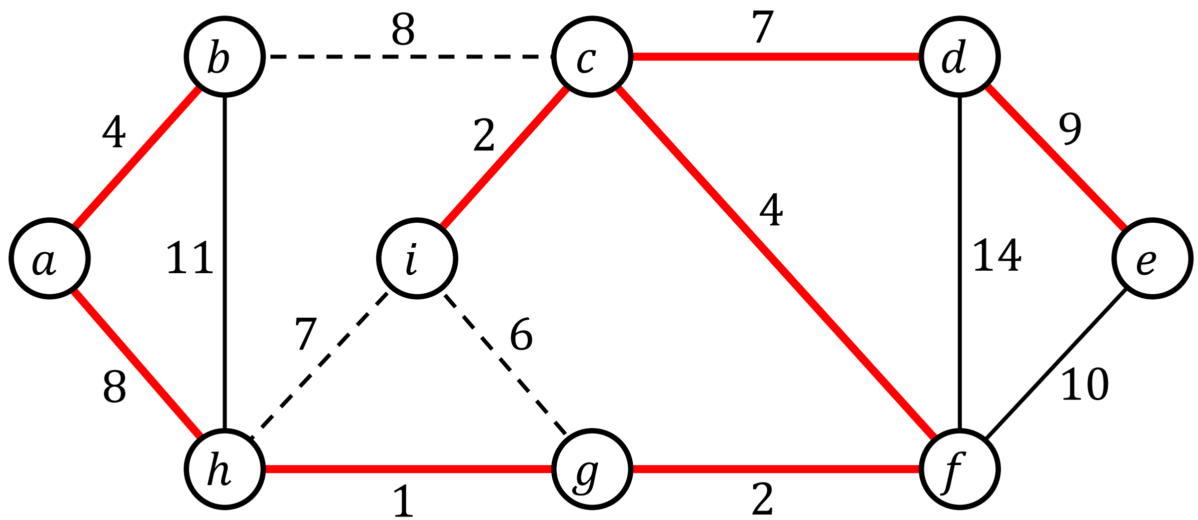
Disjoint-Set Data Structure (union by rank only) :

FIND(d) returns g, FIND(e) returns e



MST: Kruskal's Algorithm

(11) edge (d, e):



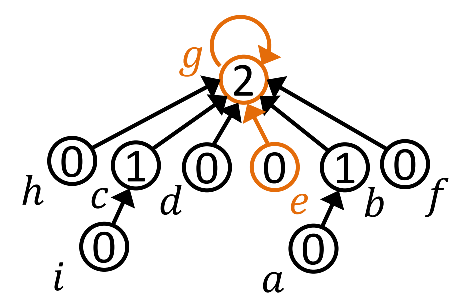
$S = \{component\ (connected\ through\ red\ edges)\ containing\ d\} = \{a, b, c, d, f, g, h, i\}$

Cut = $(S, V - S)$

(d, e) is a light edge crossing the cut

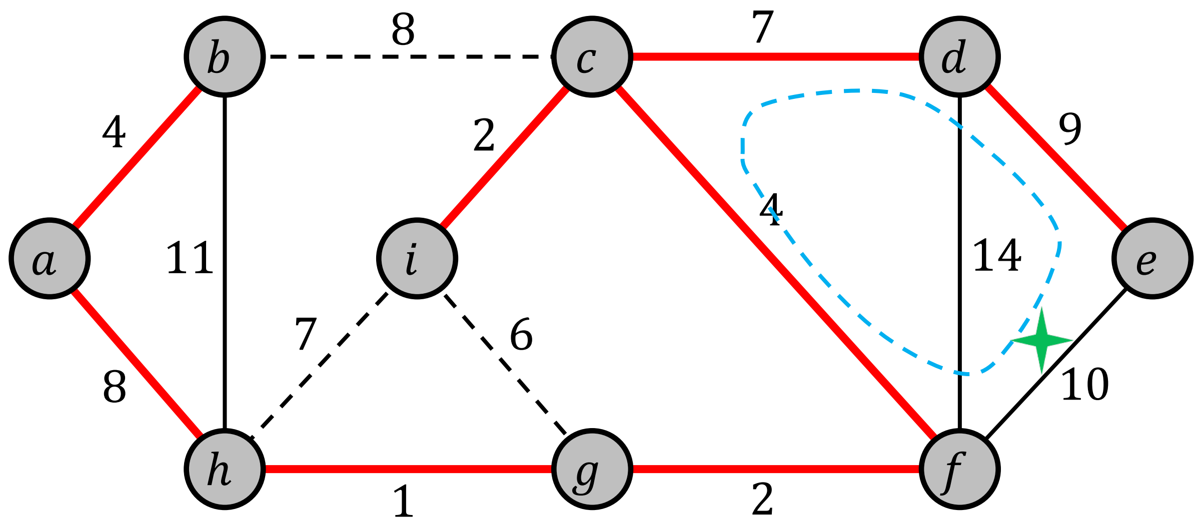
Disjoint-Set Data Structure (union by rank only) :

UNION(d, e)



MST: Kruskal's Algorithm

(12) edge (e, f):



$$S = \{component\ (connected\ through\ red\ edges)\ containing\ e\} = \{a, b, c, d, e, f, g, h, i\}$$

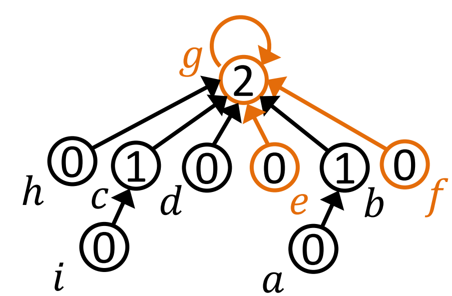
Cut = $(S, V - S)$

(e, f) creates a cycle by connecting two nodes of S, and it is the heaviest edge on that cycle

---

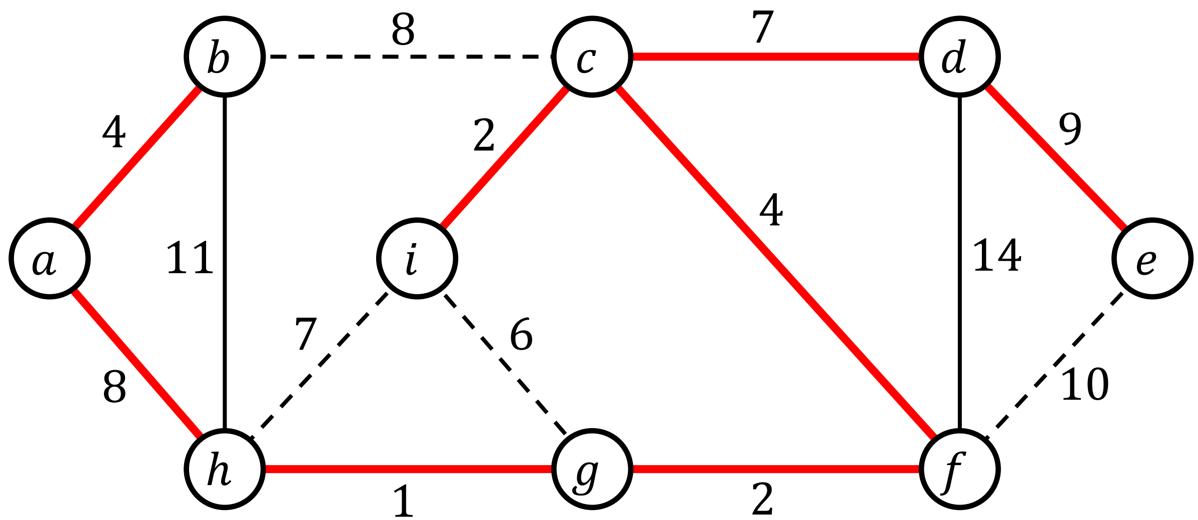
Disjoint-Set Data Structure (union by rank only) :

FIND(e) returns g, FIND(f) returns g



MST: Kruskal's Algorithm

(12) edge (e, f):



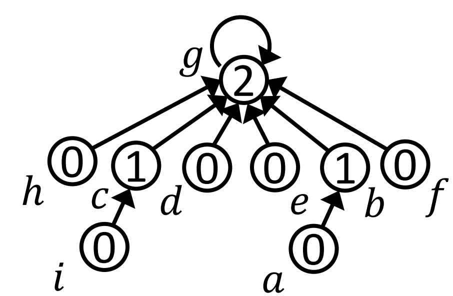
$$S = \{component\ (connected\ through\ red\ edges)\ containing\ e\} = \{a, b, c, d, e, f, g, h, i\}$$

Cut = $(S, V - S)$

(e, f) creates a cycle by connecting two nodes of S, and it is the heaviest edge on that cycle

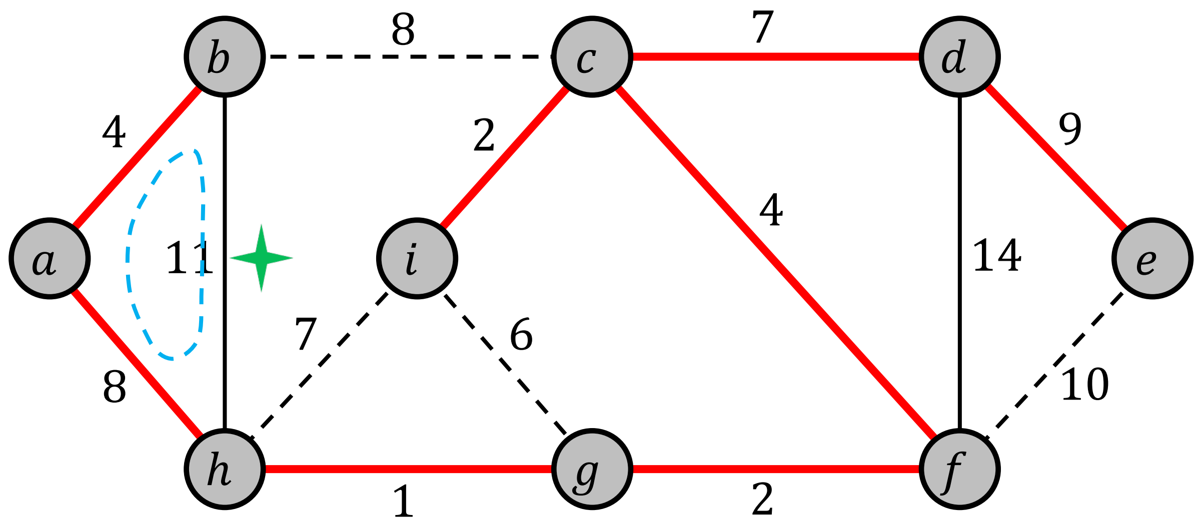
---

Disjoint-Set
Data Structure
(union by rank only) :



MST: Kruskal's Algorithm

(13) edge (b, h):



$S = \{component\ (connected\ through\ red\ edges)\ containing\ b\} = \{a, b, c, d, e, f, g, h, i\}$

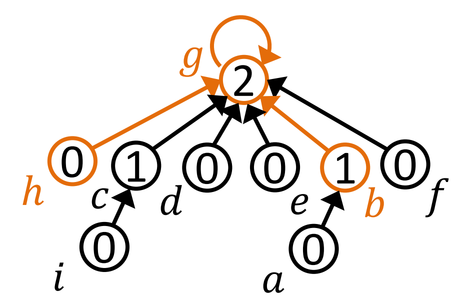
Cut = $(S, V - S)$

(b, h) creates a cycle by connecting two nodes of S, and it is the heaviest edge on that cycle

---

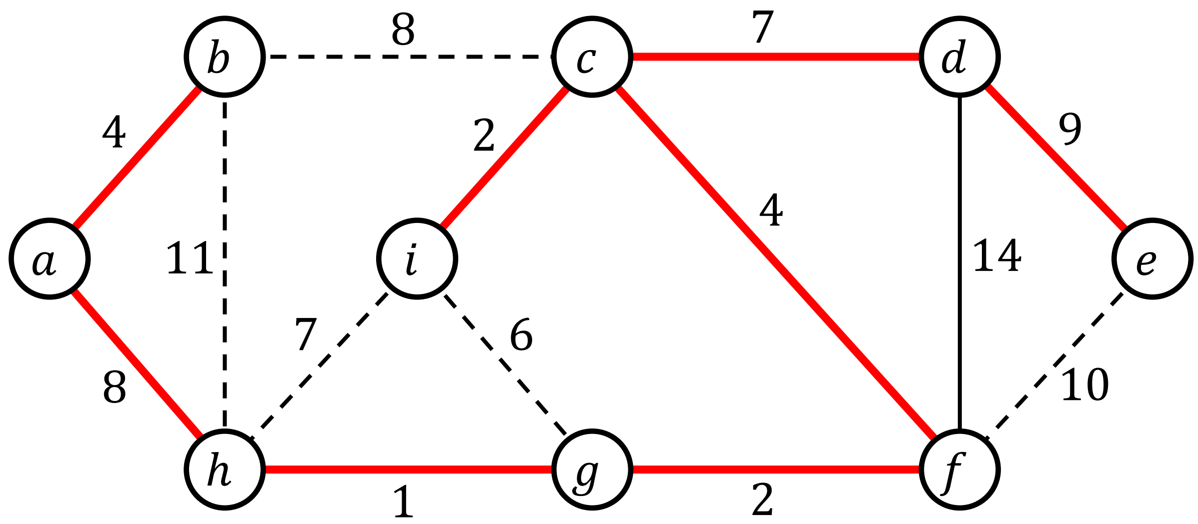
Disjoint-Set Data Structure (union by rank only) :

FIND(b) returns g, FIND(h) returns g



MST: Kruskal's Algorithm

(13) edge (b, h):



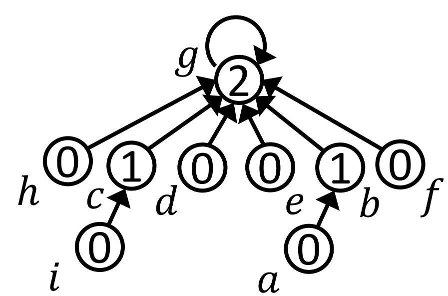
$$S = \{component\ (connected\ through\ red\ edges)\ containing\ b\} = \{a, b, c, d, e, f, g, h, i\}$$

Cut = $(S, V - S)$

(b, h) creates a cycle by connecting two nodes of S, and it is the heaviest edge on that cycle

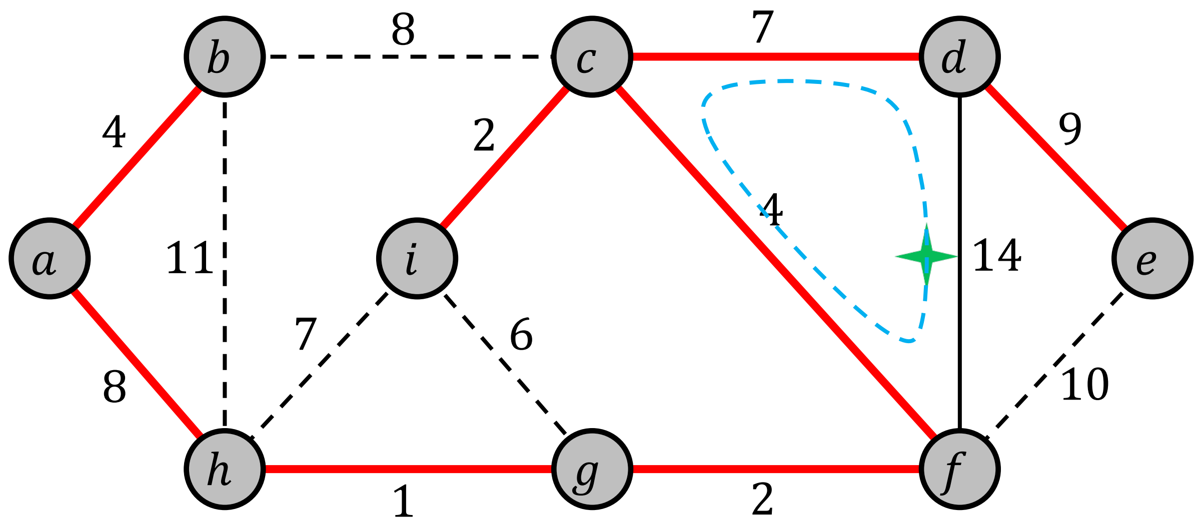
---

Disjoint-Set
Data Structure
(union by rank only) :



MST: Kruskal's Algorithm

(14) edge (d, f):



$S = \{component\ (connected\ through\ red\ edges)\ containing\ d\} = \{a, b, c, d, e, f, g, h, i\}$

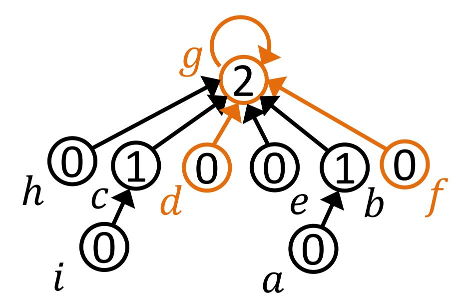
Cut = $(S, V - S)$

(d, f) creates a cycle by connecting two nodes of S, and it is the heaviest edge on that cycle

---

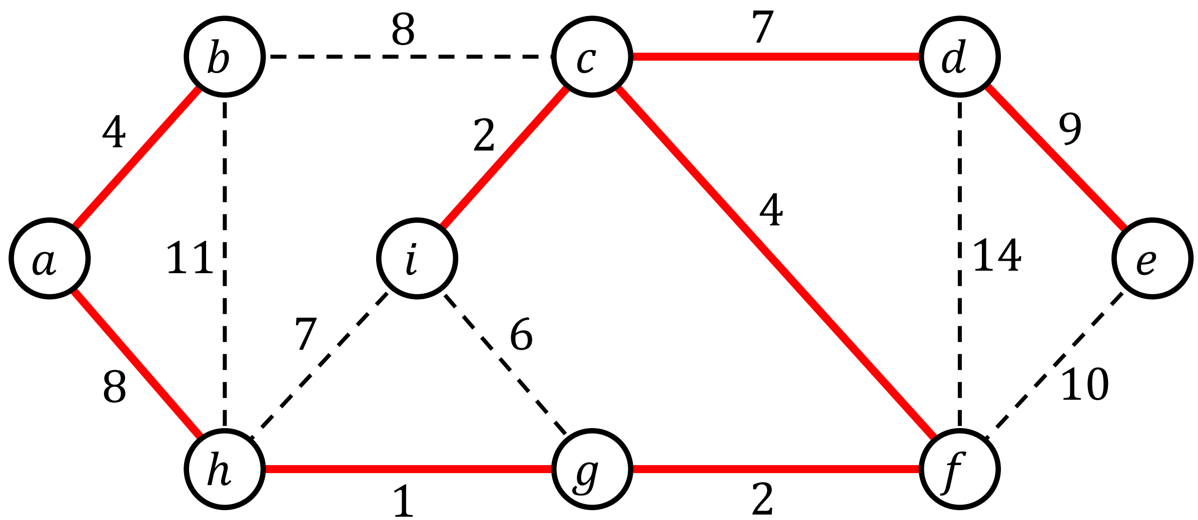
Disjoint-Set Data Structure (union by rank only) :

FIND(d) returns g, FIND(f) returns g



MST: Kruskal's Algorithm

(14) edge (d, f):



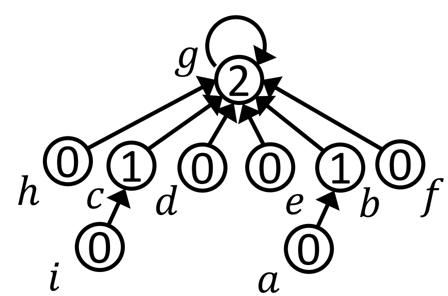
$$S = \{component\ (connected\ through\ red\ edges)\ containing\ d\} = \{a, b, c, d, e, f, g, h, i\}$$

Cut = $(S, V - S)$

(d, f) creates a cycle by connecting two nodes of S, and it is the heaviest edge on that cycle

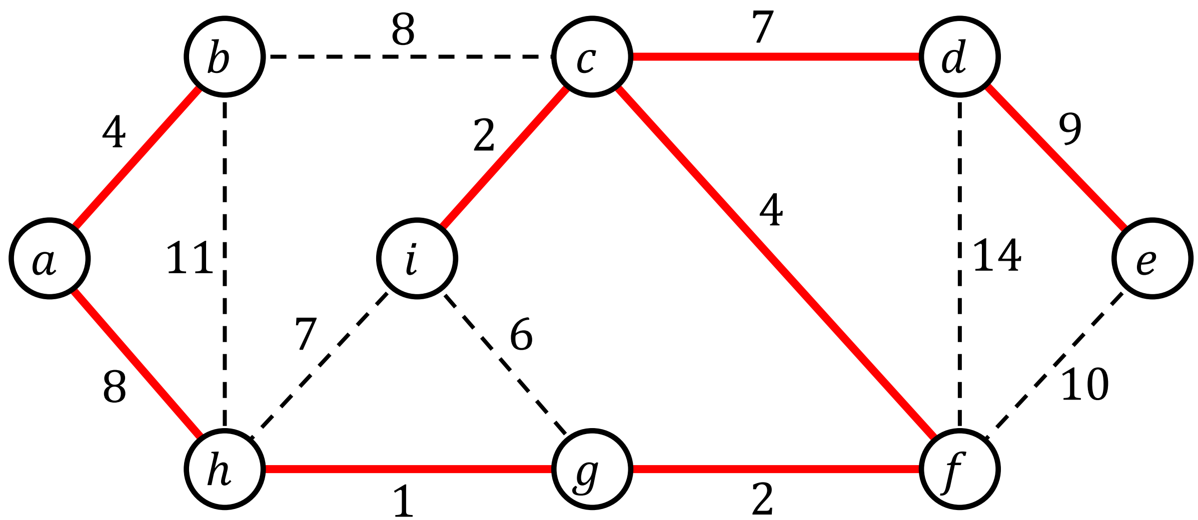
---

Disjoint-Set
Data Structure
(union by rank only) :



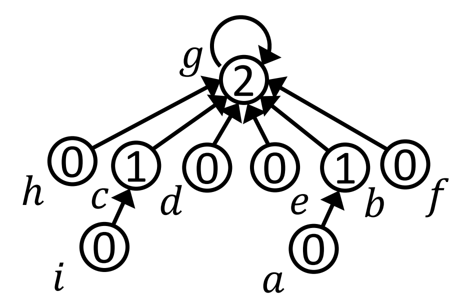
MST: Kruskal's Algorithm

(14) edge (d, f):



Total weight $= 37$

Disjoint-Set
Data Structure
(union by rank only) :



MST: Kruskal's Algorithm
(union by rank)

MST-Kruskal ($G = (V, E)$, w)

1. $A \leftarrow \emptyset$
2. *for* each vertex $v \in G.V$ *do*
3.   Make-Set(v)
4. sort the edges of $G.E$ into nondecreasing order by weight w
5. *for* each edge $(u, v) \in G.E$ taken in nondecreasing order by weight *do*
6.   *if* Find(u) $\neq$ Find(v) *then*
7. $A \leftarrow A \cup \{(u, v)\}$
8.   Union(u, v)
9. *return* A

Let $n = |V|$ and $m = |E|$. Since G is connected, we have $m \geq n - 1$.

Then the sorting in step 4 can be done in $\mathrm{O}(m \log m)$ time.

#disjoint-set operations performed, $N = 2m + 2n - 1$, of which

#Make-Set: n, #Find: $2m$, #Union: $n - 1$

So, total time taken by disjoint-set operations $= \mathrm{O}\big((n + m) \log n\big)$

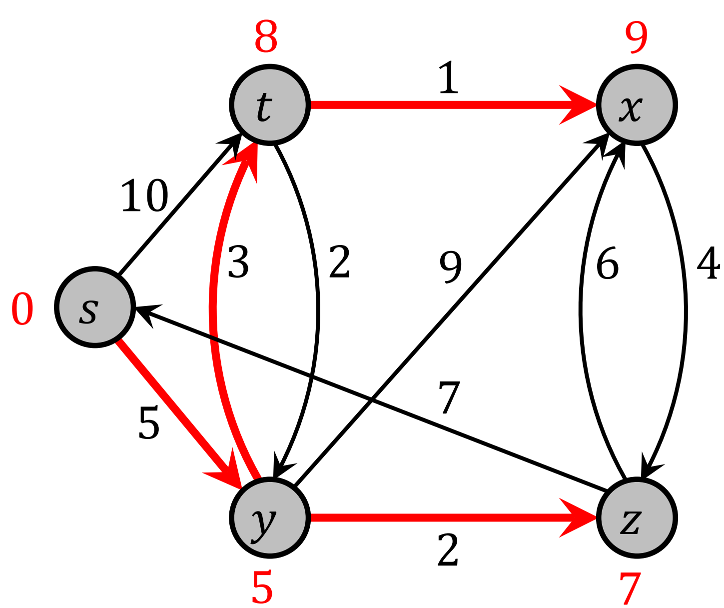
Hence, MST-Kruskal's running time $= \mathrm{O}(m \log m)$

The Single-Source Shortest Paths (SSSP) Problem

We are given a weighted, directed graph $G = (V, E)$ with vertex set V and edge set E, and a non-negative weight function w such that for each edge $(u, v) \in E$, $w(u, v)$ represents its weight.

We are also given a source vertex $s \in V$.

Our goal is to find a shortest path (i.e., a path of the smallest total edge weight) from s to each vertex $v \in V$.



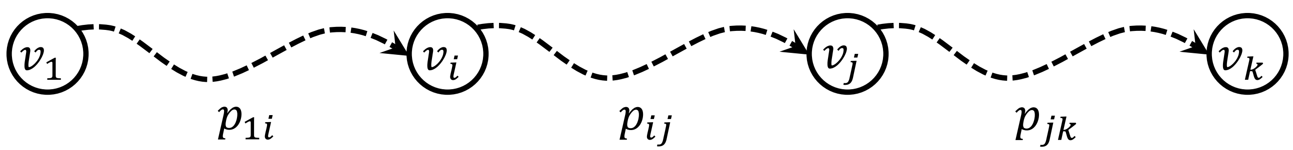
Intuition behind Dijkstra's SSSP Algorithm

Lemma: [subpaths of shortest paths are shortest paths] Given a weighted, directed graph $G = (V, E)$ with weight function $w: E \rightarrow \mathbb{R}$, let $p = v_1 v_2 \dots v_k$ be a shortest path from vertex v_1 to vertex v_k and, for any i and j such that $1 \leq i \leq j \leq k$, let $p_{ij} = v_i v_{i+1} \dots v_j$ be the subpath of p from vertex v_i to vertex v_j. Then p_{ij} is a shortest path from v_i to v_j.

Intuition behind Dijkstra's SSSP Algorithm

Lemma: [subpaths of shortest paths are shortest paths] … … …
let $p = v_1 v_2 \ldots v_k$ be a shortest path from v_1 to v_k and, for any i and j such that $1 \le i \le j \le k$, let $p_{ij} = v_i v_{i+1} \ldots v_j$ be the subpath of p from v_i to v_j. Then p_{ij} is a shortest path from v_i to v_j.

Proof: Let's decompose p as follows.



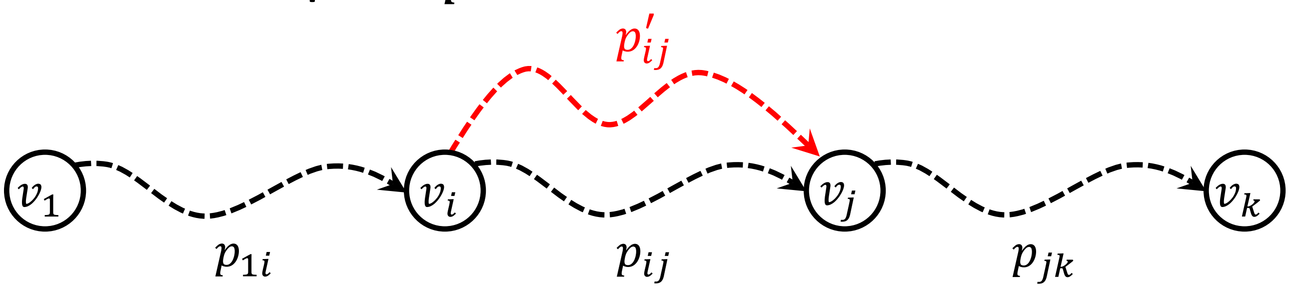
Then weight of path p, $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$.

Intuition behind Dijkstra's SSSP Algorithm

Lemma: [subpaths of shortest paths are shortest paths] … … …

let $p = v_1 v_2 \ldots v_k$ be a shortest path from v_1 to v_k and, for any i and j such that $1 \le i \le j \le k$, let $p_{ij} = v_i v_{i+1} \ldots v_j$ be the subpath of p from v_i to v_j. Then p_{ij} is a shortest path from v_i to v_j.

Proof: Let's decompose p as follows.



Then weight of path p, $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$.

If p_{ij} is not a shortest path, let p'_{ij} be a shorter path from v_i to v_j.

$\therefore w(p_{1i}) + w(p'_{ij}) + w(p_{jk}) < w(p_{1i}) + w(p_{ij}) + w(p_{jk}) = w(p)$,

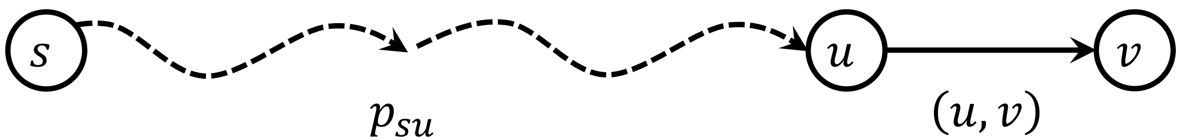
which contradicts our assumption that p is a shortest v_1 to v_k path.

Intuition behind Dijkstra's SSSP Algorithm

Observations: Let $v \in V$ and $v \neq s$.

Consider any shortest path $p_{s,v}$ from s to v.

Path $p_{s,v}$ must reach v through a node u from which v has an incoming edge, i.e., $(u, v) \in E$.



Let $p_{s,u}$ be the subpath of $p_{s,v}$ that goes from s to u.

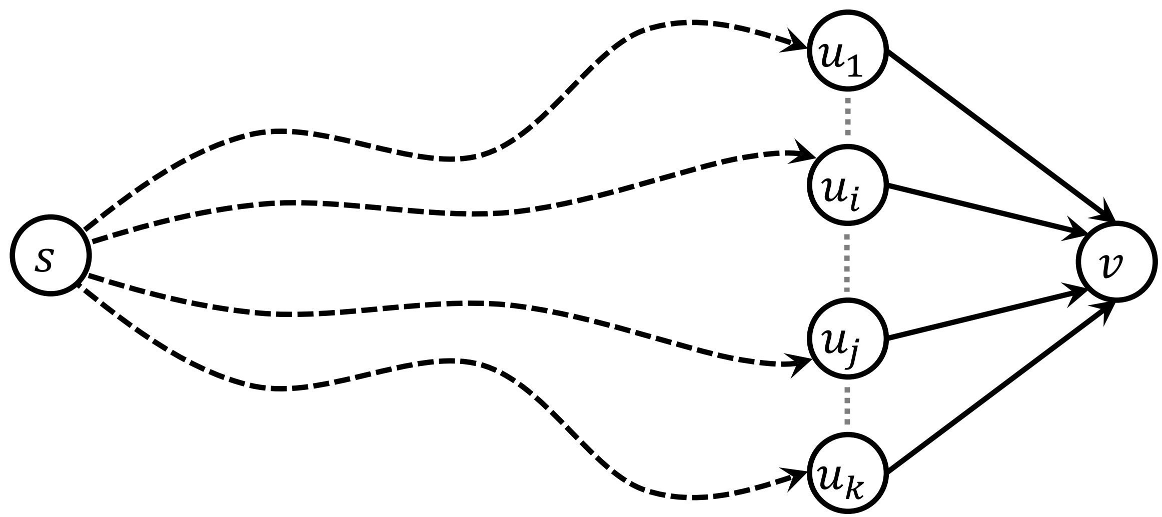
Since subpaths of shortest paths are also shortest paths, $p_{s,u}$ must be a shortest path from s to u.

So, once we know $p_{s,u}$, we can append (u, v) to it to find $p_{s,v}$.

But two questions!

Intuition behind Dijkstra's SSSP Algorithm

First question: v can have multiple incoming edges. How do we know which of them lies on $p_{s,v}$?

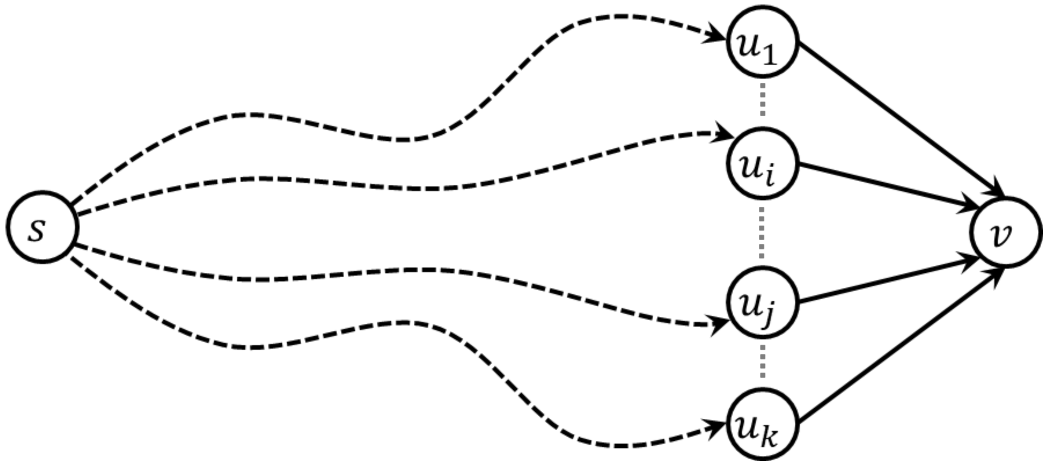


Suppose, v has k incoming edges $(u_1, v), (u_2, v), \dots, (u_k, v)$.

The solution is to maintain a tentative shortest s to v distance $d[v]$ initialized to ∞, and update $d[v]$ to $\min\{d[v], w(p_{s,u}) + w(u, v)\}$ when we find the shortest path $p_{s,u}$ to each $u \in \{u_1, u_2, \dots, u_k\}$.

Intuition behind Dijkstra's SSSP Algorithm

Second question: When do we know that $d[v] = \delta(s, v)$, where $\delta(s, v)$ is the shortest distance from s to v?



Find shortest paths to vertices in non-decreasing order of $\delta(s, \cdot)$.

We start with vertex s because we know $\delta(s, s) = 0$.

Since edge weights are non-negative, any u with $\delta(s, u) > \delta(s, v)$ cannot be on $p_{s,v}$.

So, if $d[v]$ is the smallest among all vertices to which we are yet to find shortest distances, we know that $d[v] = \delta(s, v)$.

Dijkstra's SSSP Algorithm with a Min-Heap
(SSSP: Single-Source Shortest Paths)

Input: Weighted graph $G = (V, E)$ with vertex set V and edge set E, a weight function w, and a source vertex $s \in G[V]$.

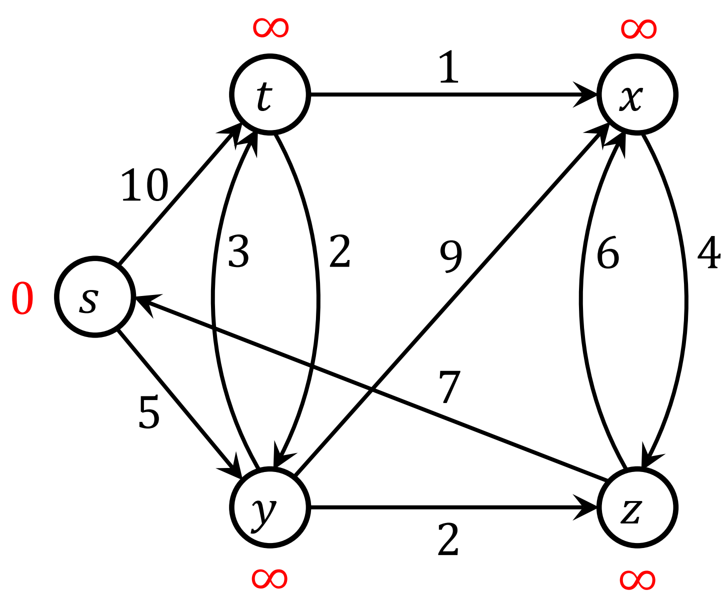
Output: For all $v \in G[V]$, $v.d$ is set to the shortest distance from s to v.

```
Dijkstra-SSSP ( G = (V, E), w, s )

for each vertex v ∈ G.V do
    v.d ← ∞
    v.π ← NIL
s.d ← 0
Min-Heap Q ← ∅
for each vertex v ∈ G.V do
    INSERT( Q, v )
while Q ≠ ∅ do
    u ← EXTRACT-MIN( Q )
    for each (u, v) ∈ G.E do
        if u.d + w(u, v) < v.d then
            v.d ← u.d + w(u, v)
            v.π ← u
            DECREASE-KEY( Q, v, u.d + w(u, v) )
```

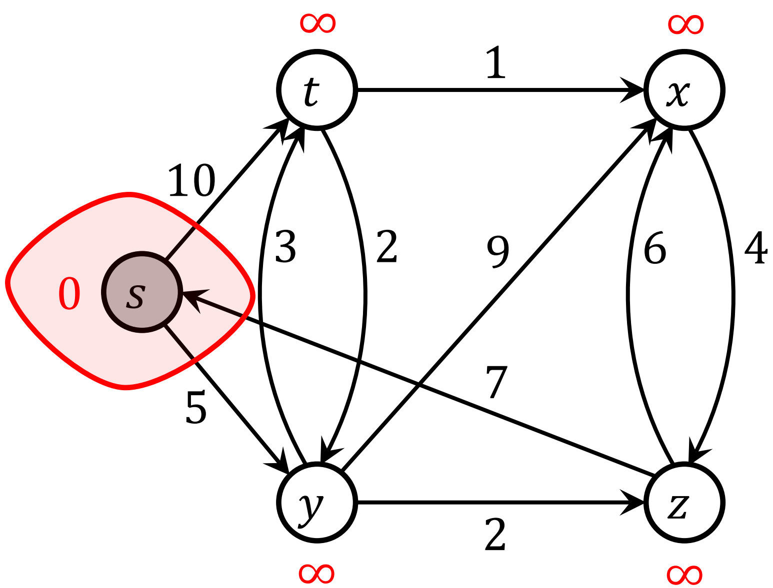
SSSP: Dijkstra's Algorithm

Initial State (with initial tentative distances)



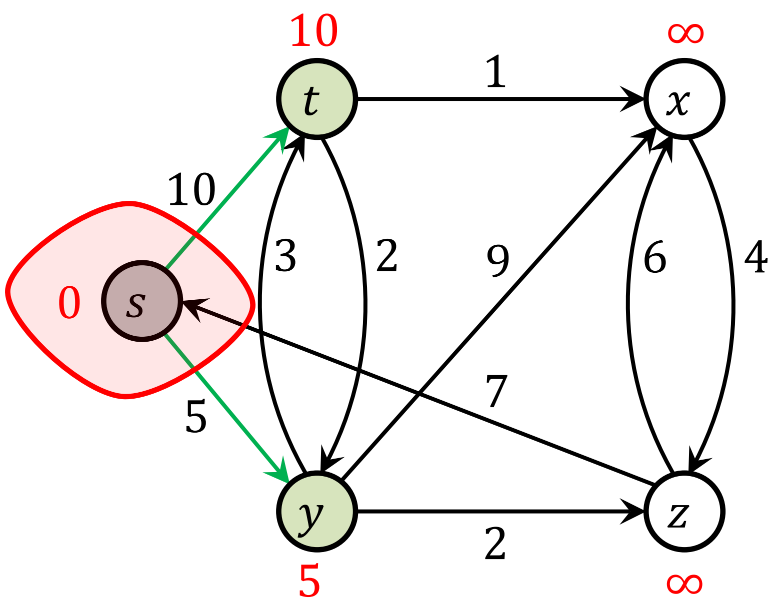
SSSP: Dijkstra's Algorithm

Step 1: add vertex s to SPT



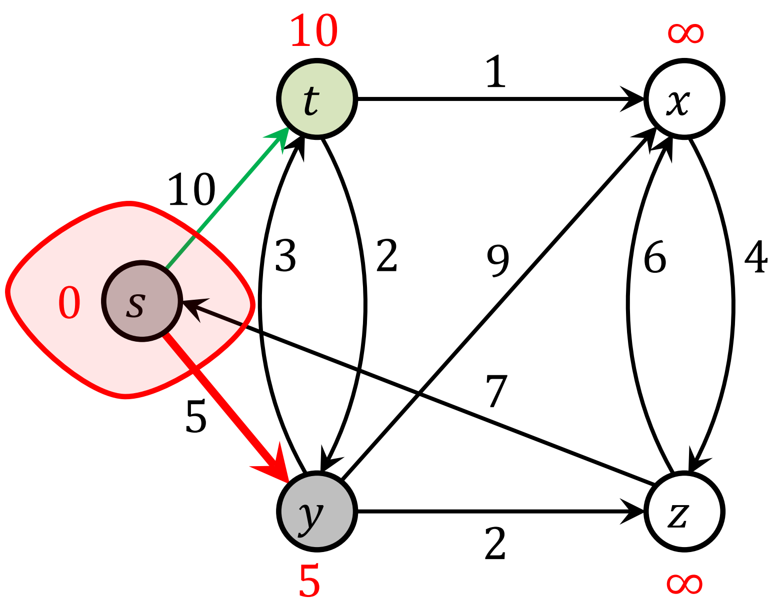
SSSP: Dijkstra's Algorithm

Step $1'$: update neighbors of s



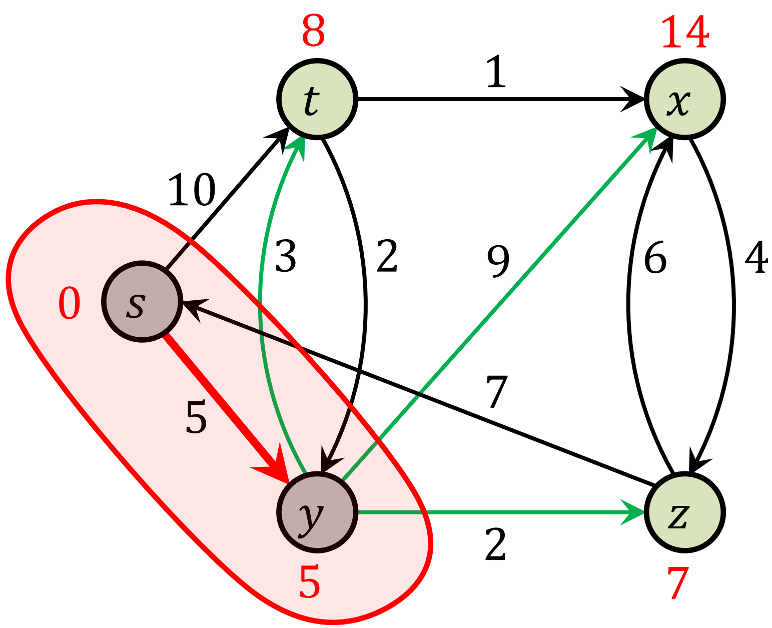
SSSP: Dijkstra's Algorithm

Step 2: add vertex y through edge (s, y)



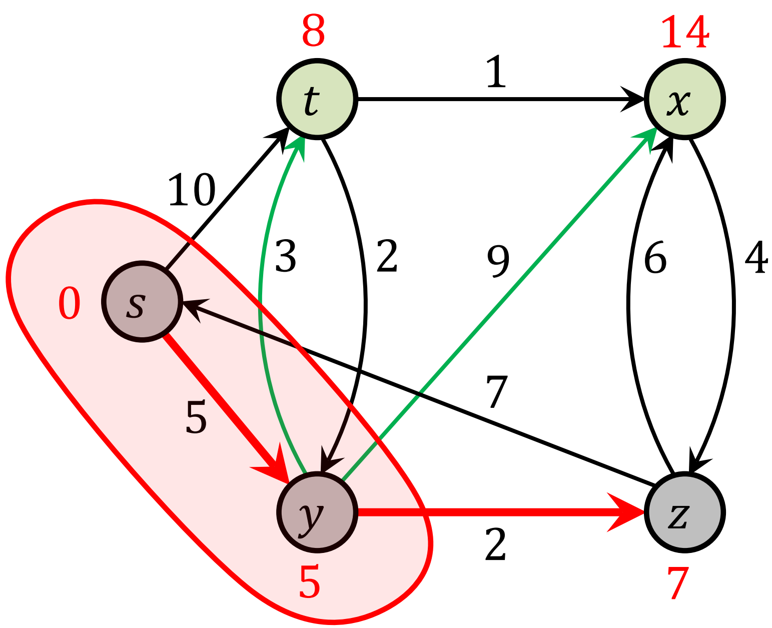
SSSP: Dijkstra's Algorithm

Step $2'$: update neighbors of y



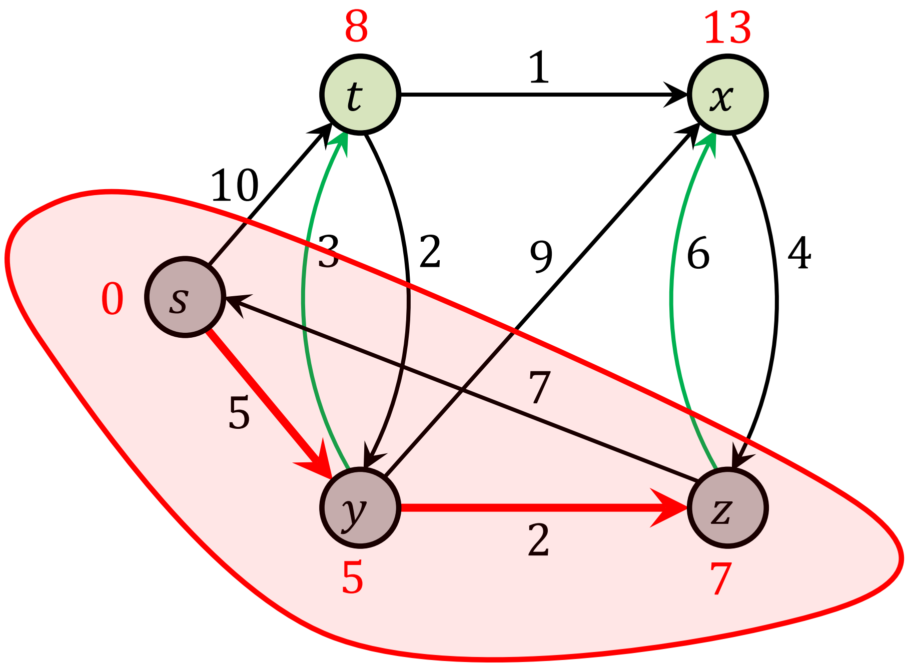
SSSP: Dijkstra's Algorithm

Step 3: add vertex z through edge (y, z)



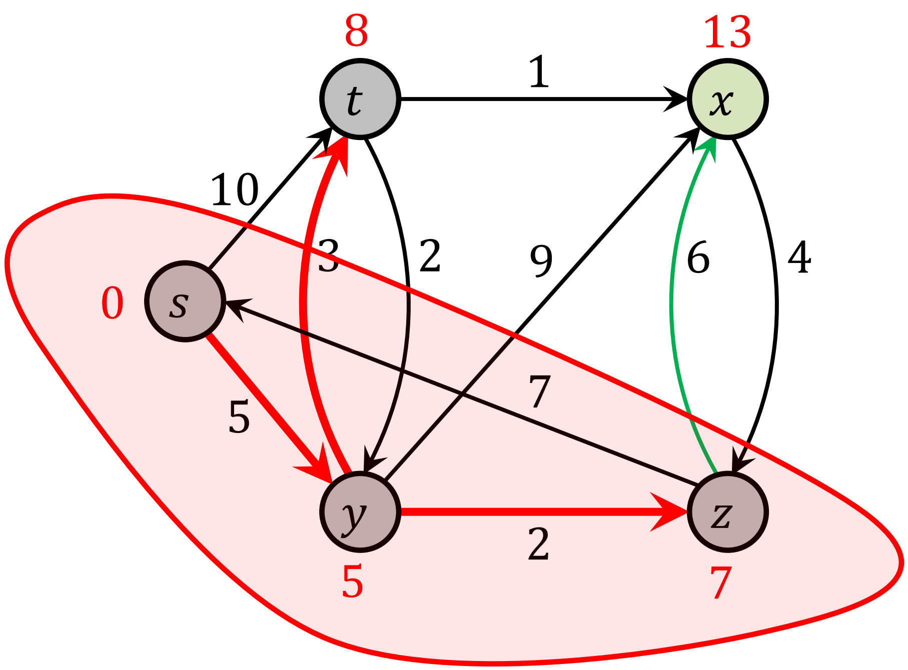
SSSP: Dijkstra's Algorithm

Step $3'$: update neighbors of z



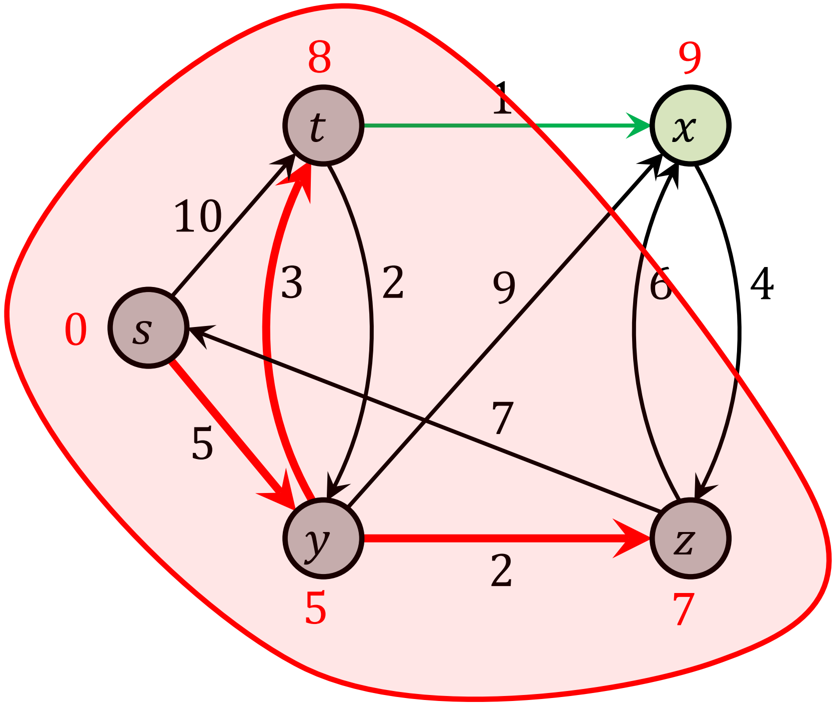
SSSP: Dijkstra's Algorithm

Step 4: add vertex t through edge (y, t)



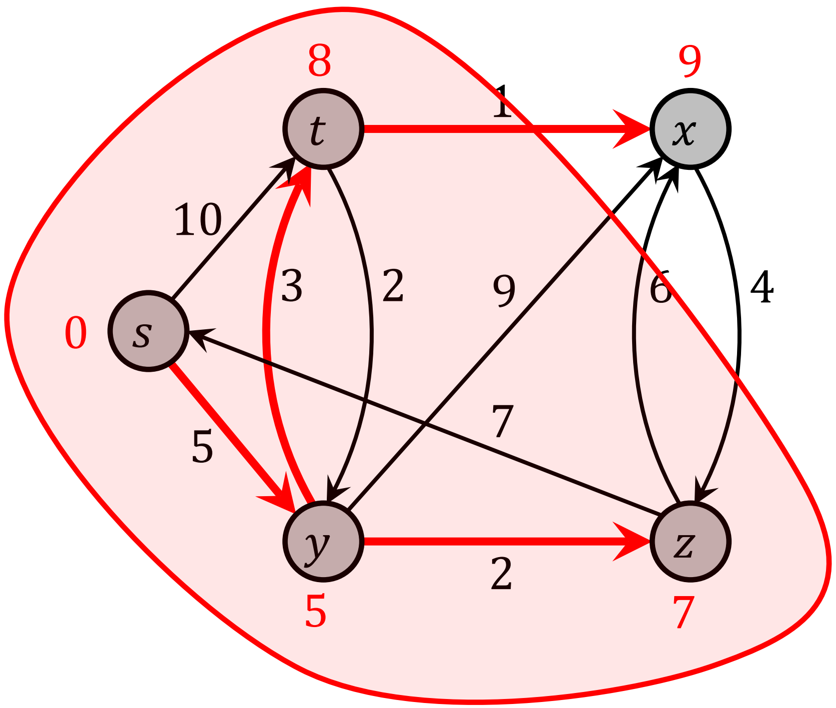
SSSP: Dijkstra's Algorithm

Step $4'$: update neighbors of t



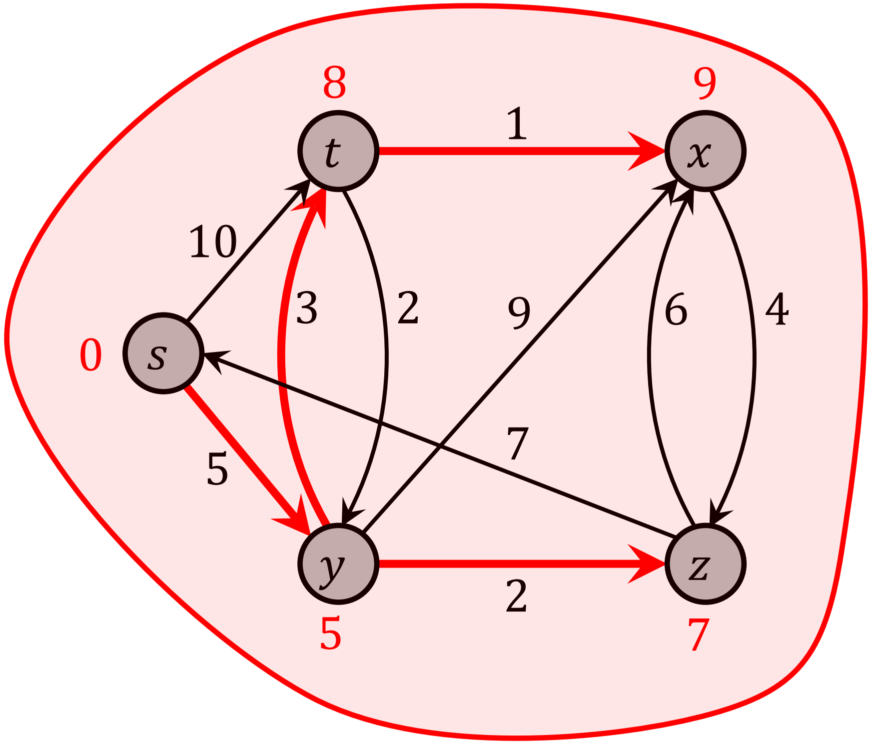
SSSP: Dijkstra's Algorithm

Step 5: add vertex x through edge (t, x)



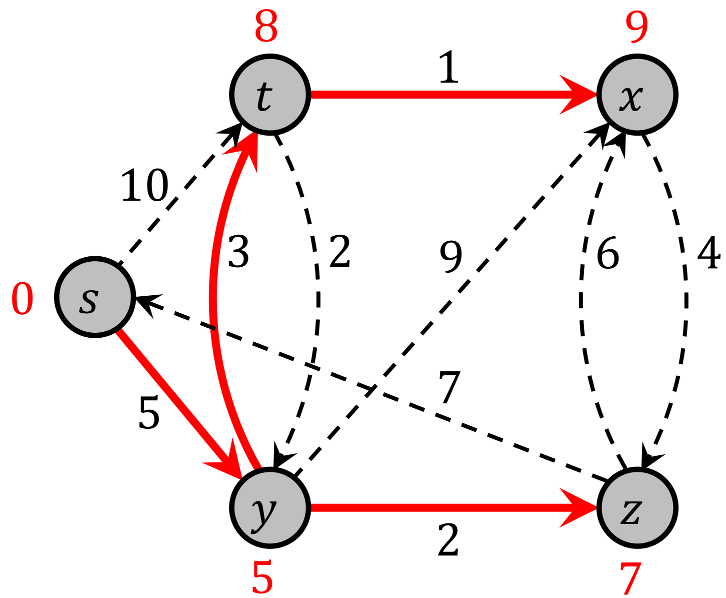
SSSP: Dijkstra's Algorithm

Step $5'$: update neighbors of x



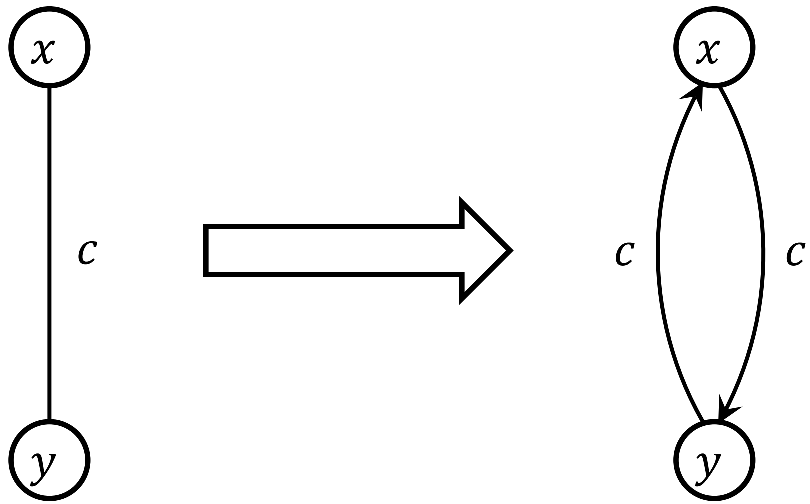
SSSP: Dijkstra's Algorithm

Done



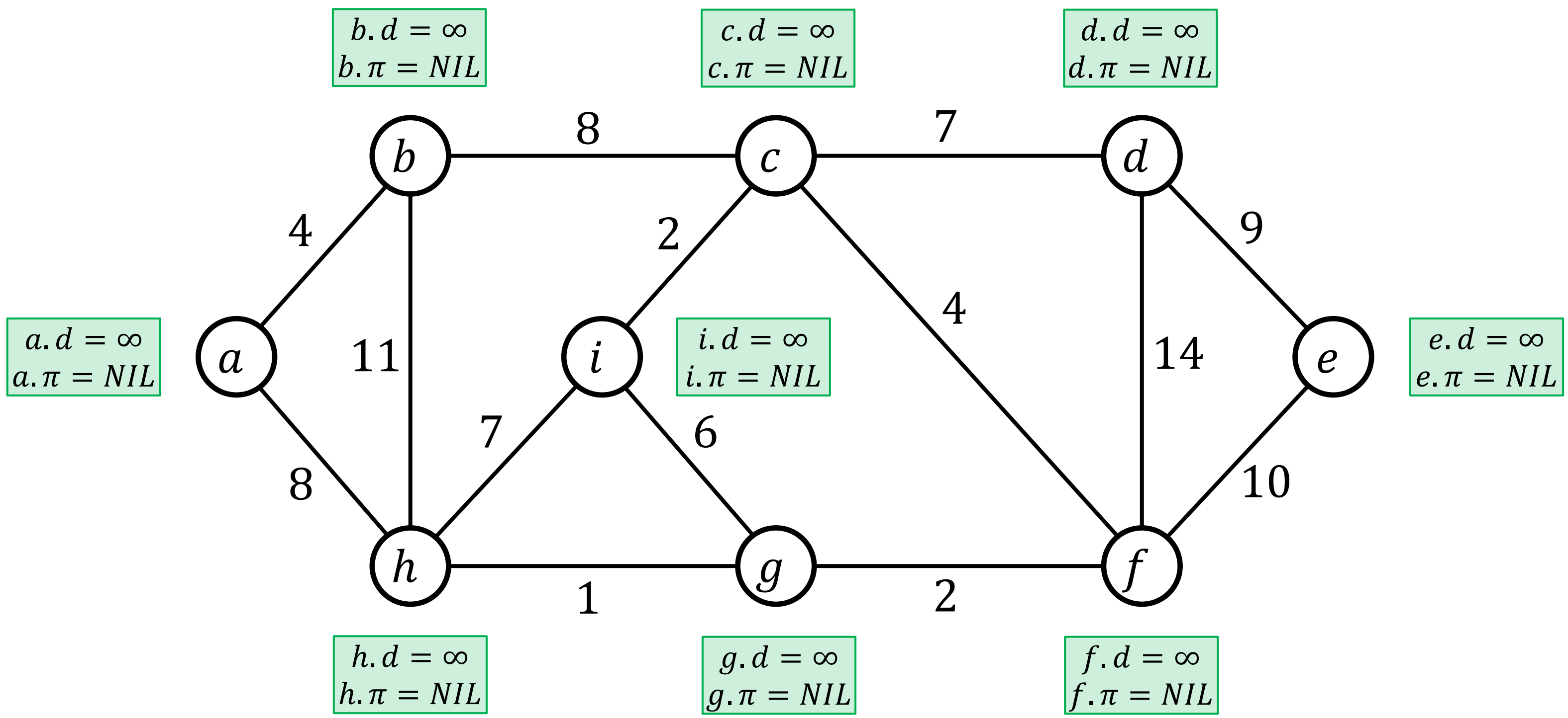
SSSP: Dijkstra's Algorithm

One undirected edge ⇒ Two directed edges



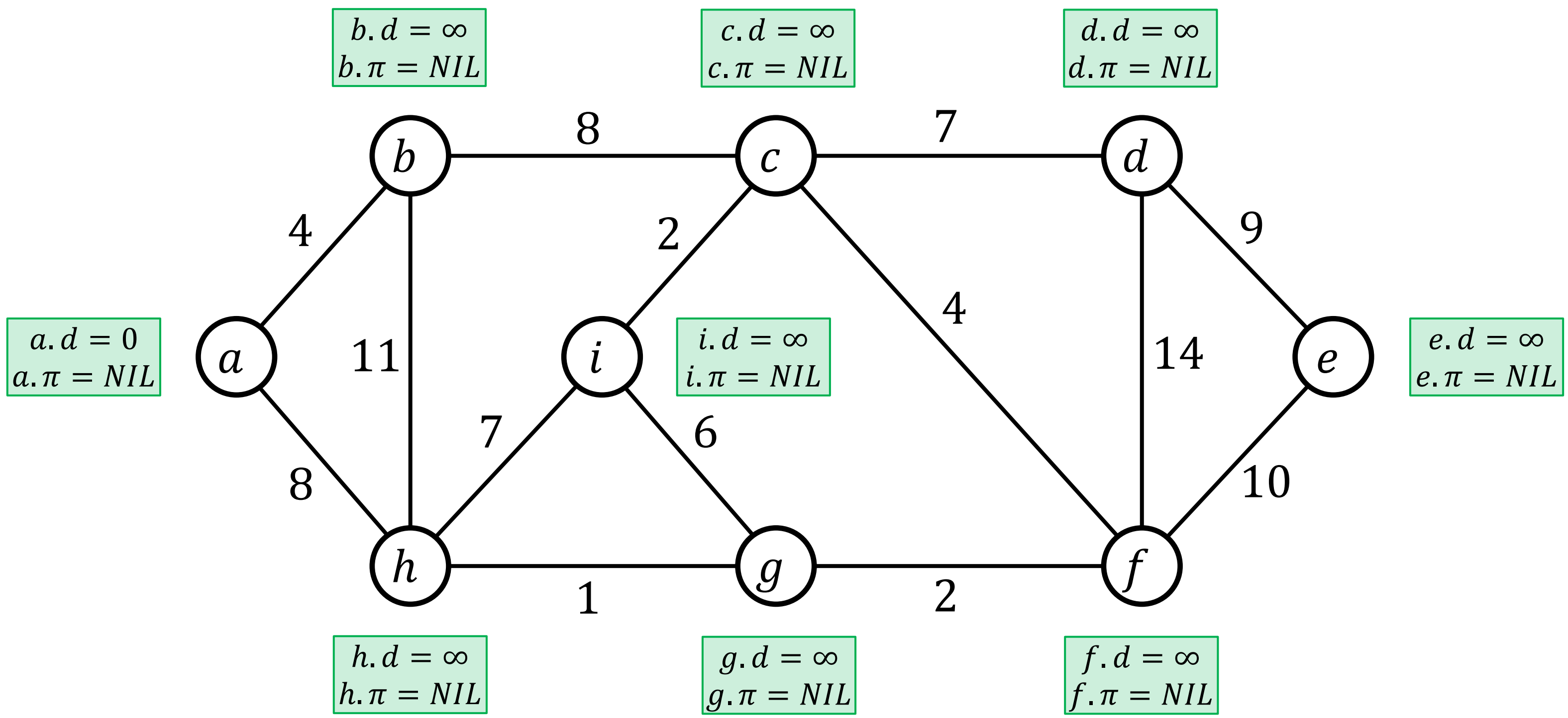
SSSP: Dijkstra's Algorithm

Initial State (with initial tentative distances)



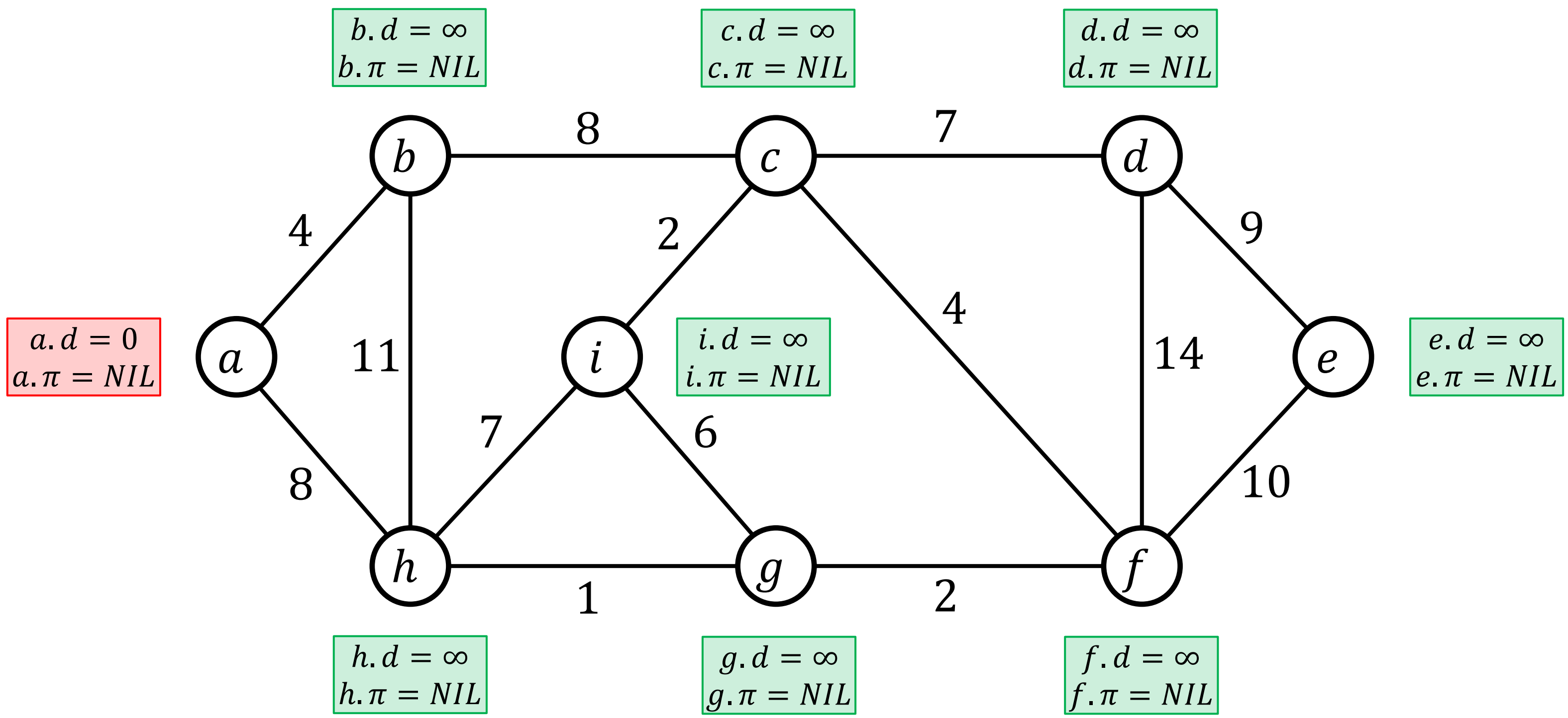
SSSP: Dijkstra's Algorithm

Initial State (with initial tentative distances)



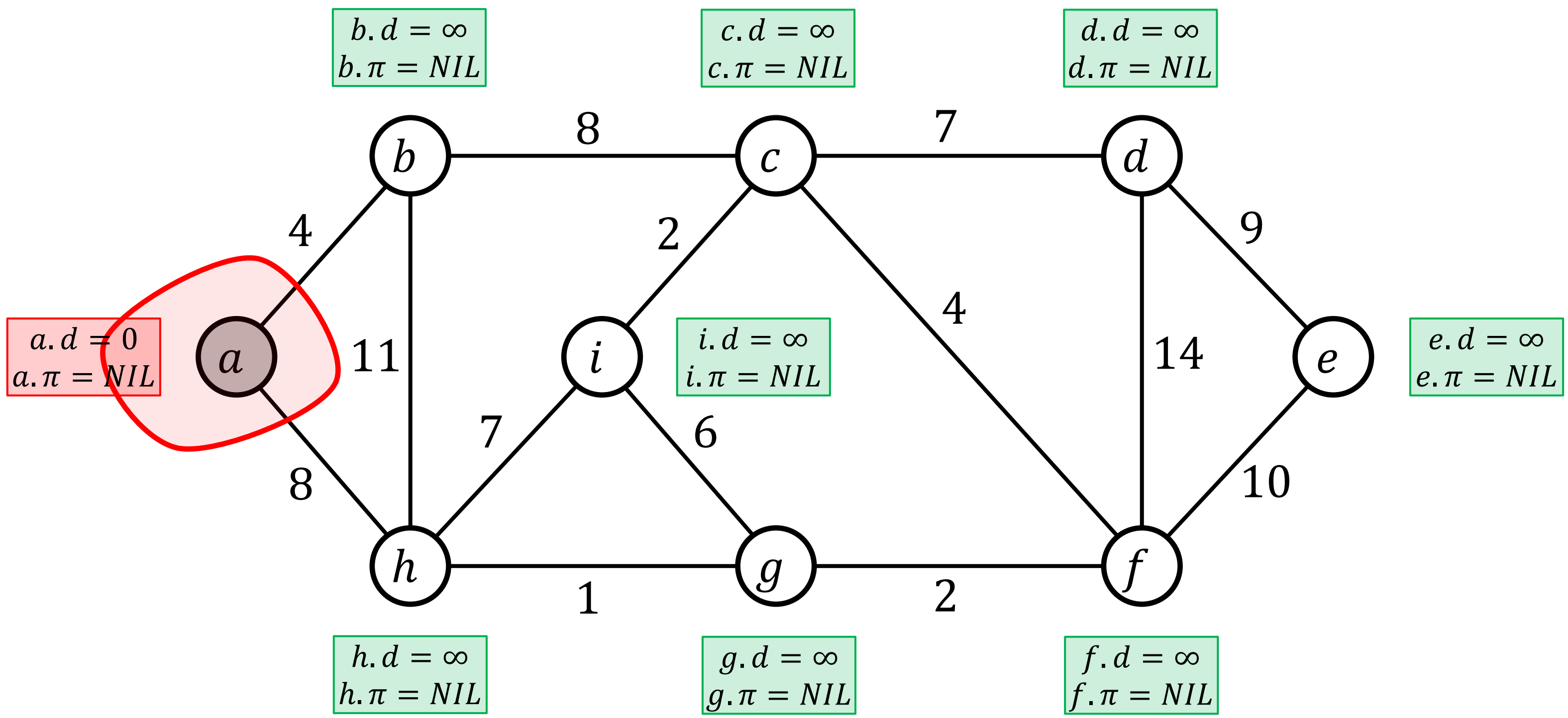
SSSP: Dijkstra's Algorithm

Initial State (with initial tentative distances)



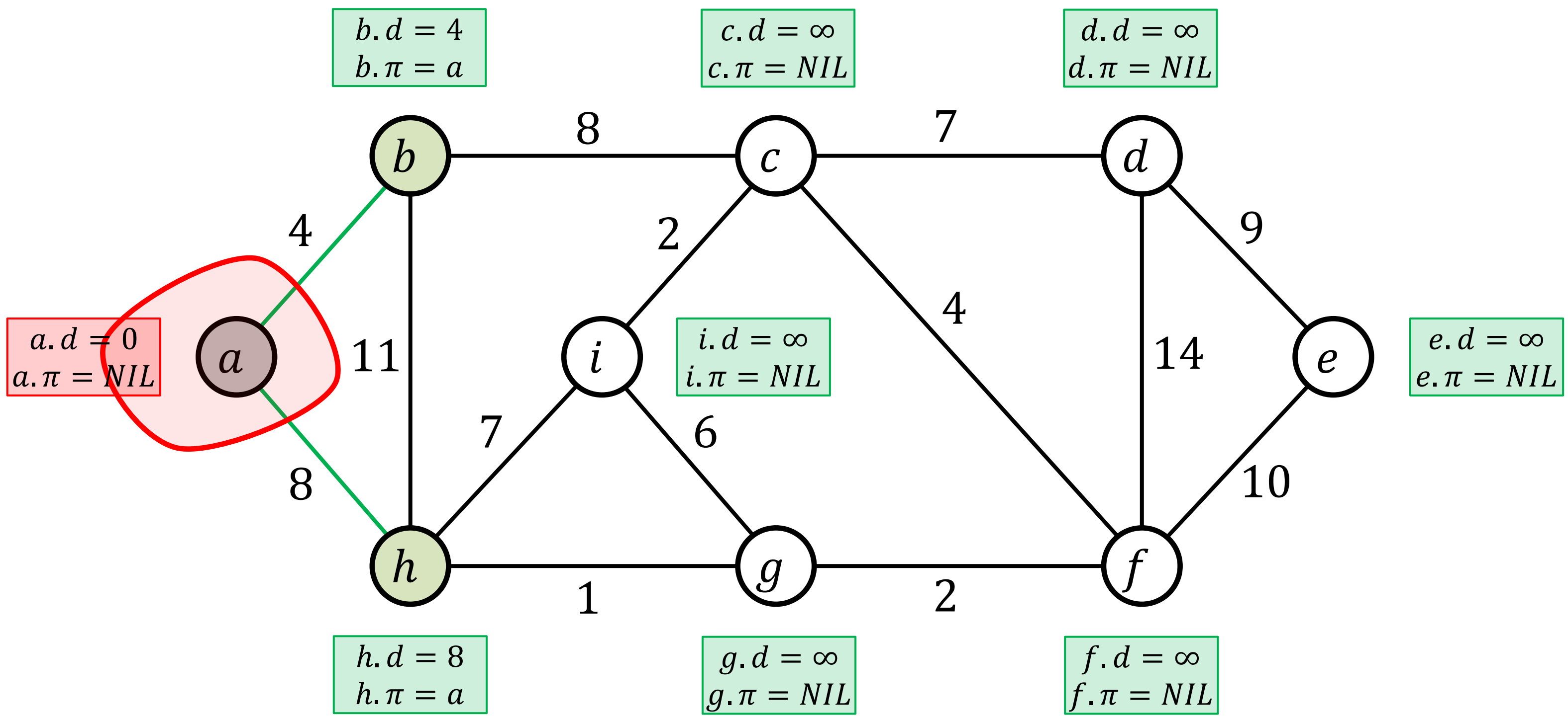
SSSP: Dijkstra's Algorithm

Step 1: add vertex a to SPT



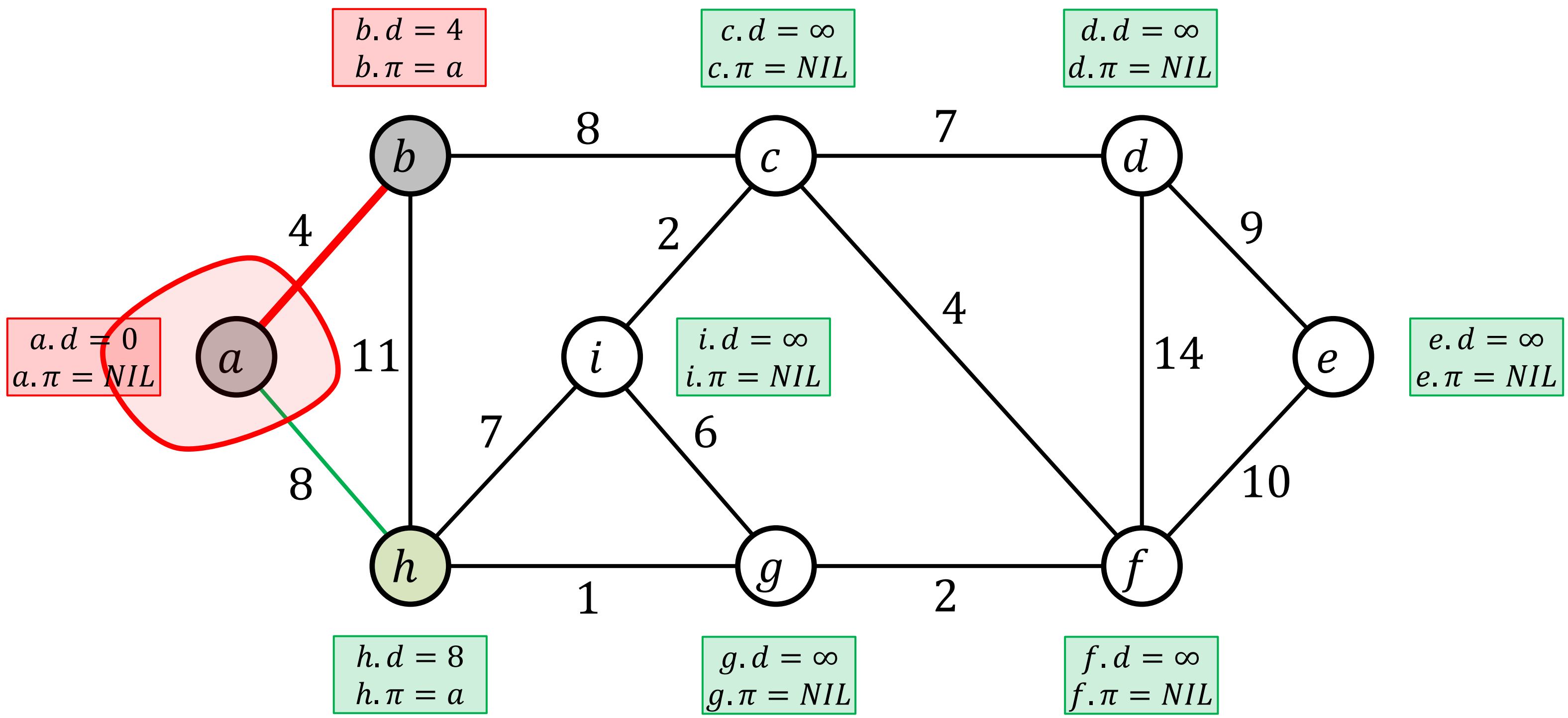
SSSP: Dijkstra's Algorithm

Step $1'$: update neighbors of a



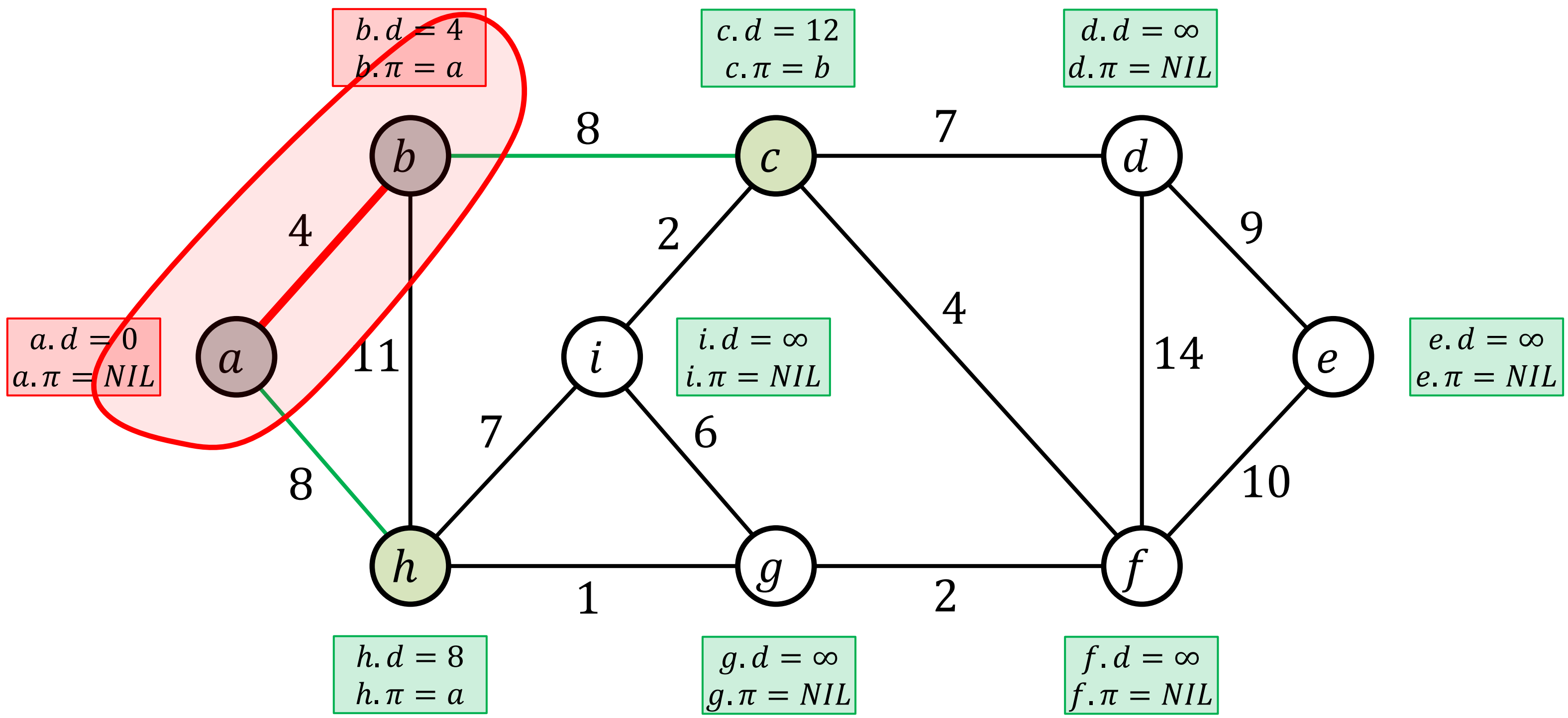
SSSP: Dijkstra's Algorithm

Step 2: add vertex b through edge (a, b)



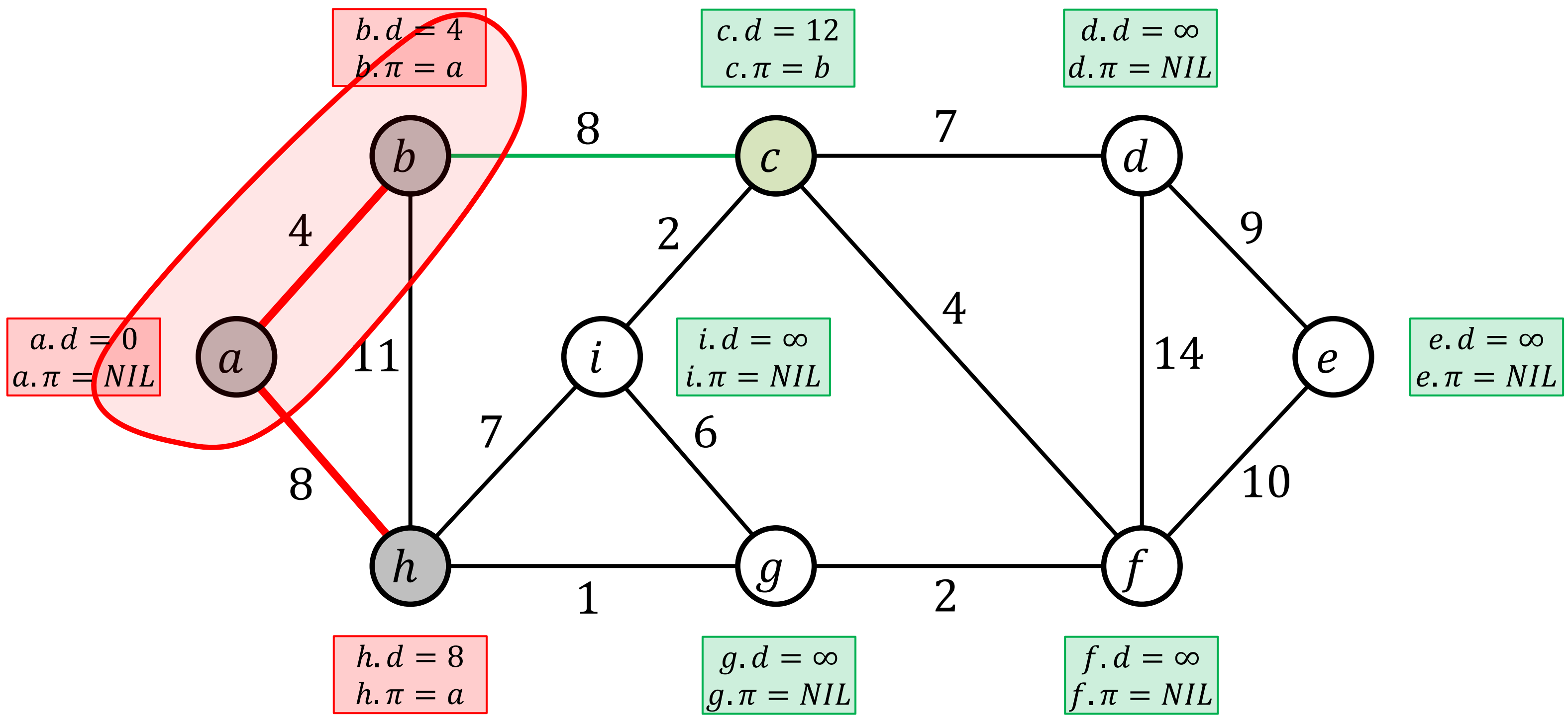
SSSP: Dijkstra's Algorithm

Step $2'$: update neighbors of b



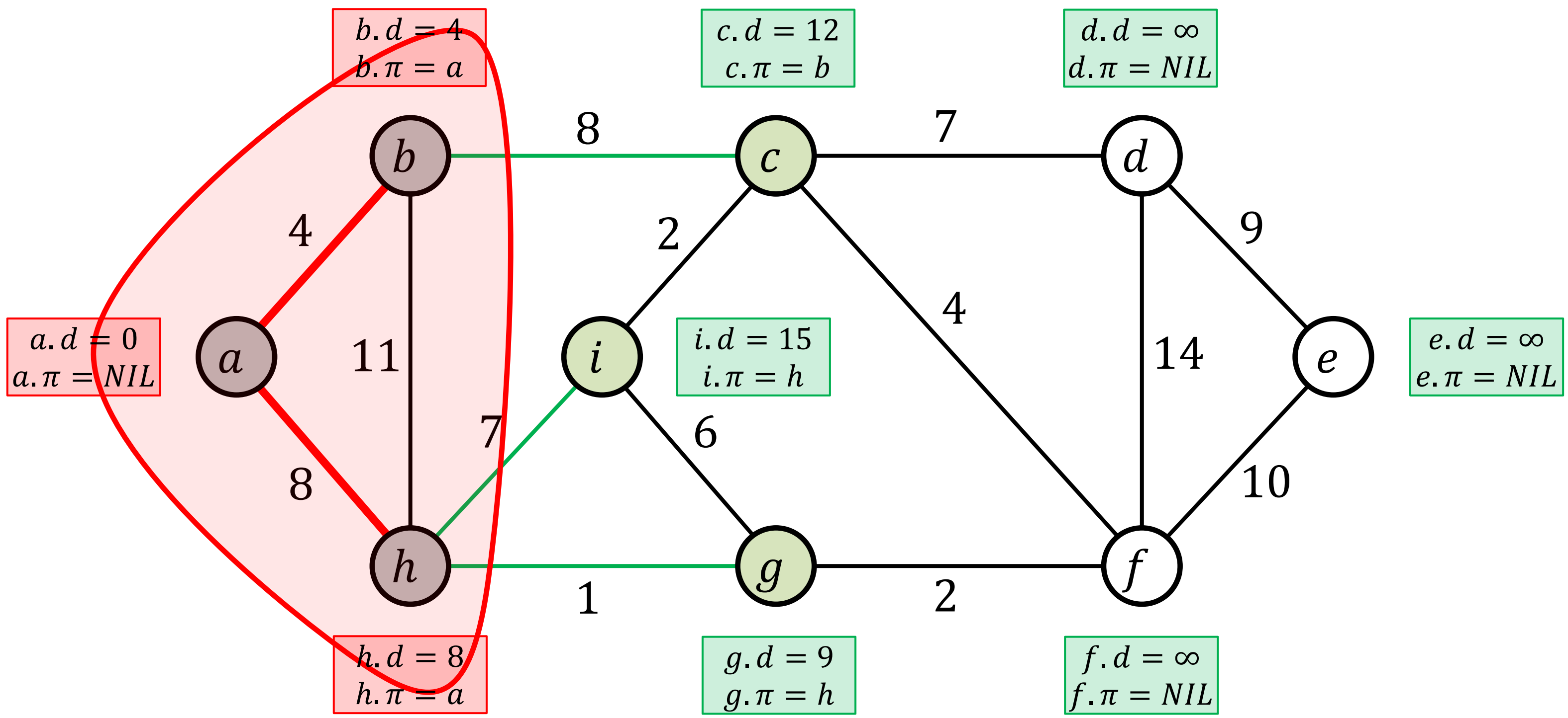
SSSP: Dijkstra's Algorithm

Step 3: add vertex h through edge (a, h)



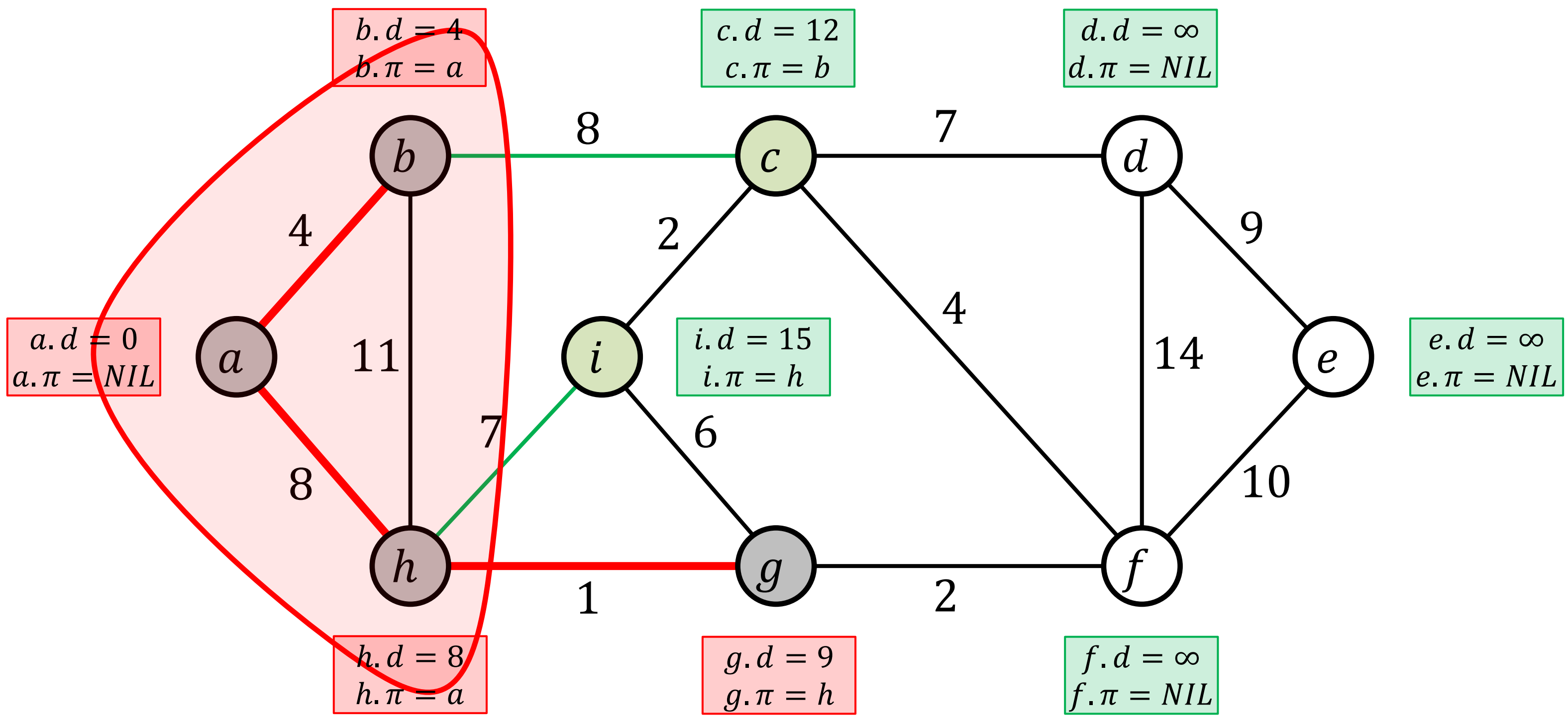
SSSP: Dijkstra's Algorithm

Step $3'$: update neighbors of h



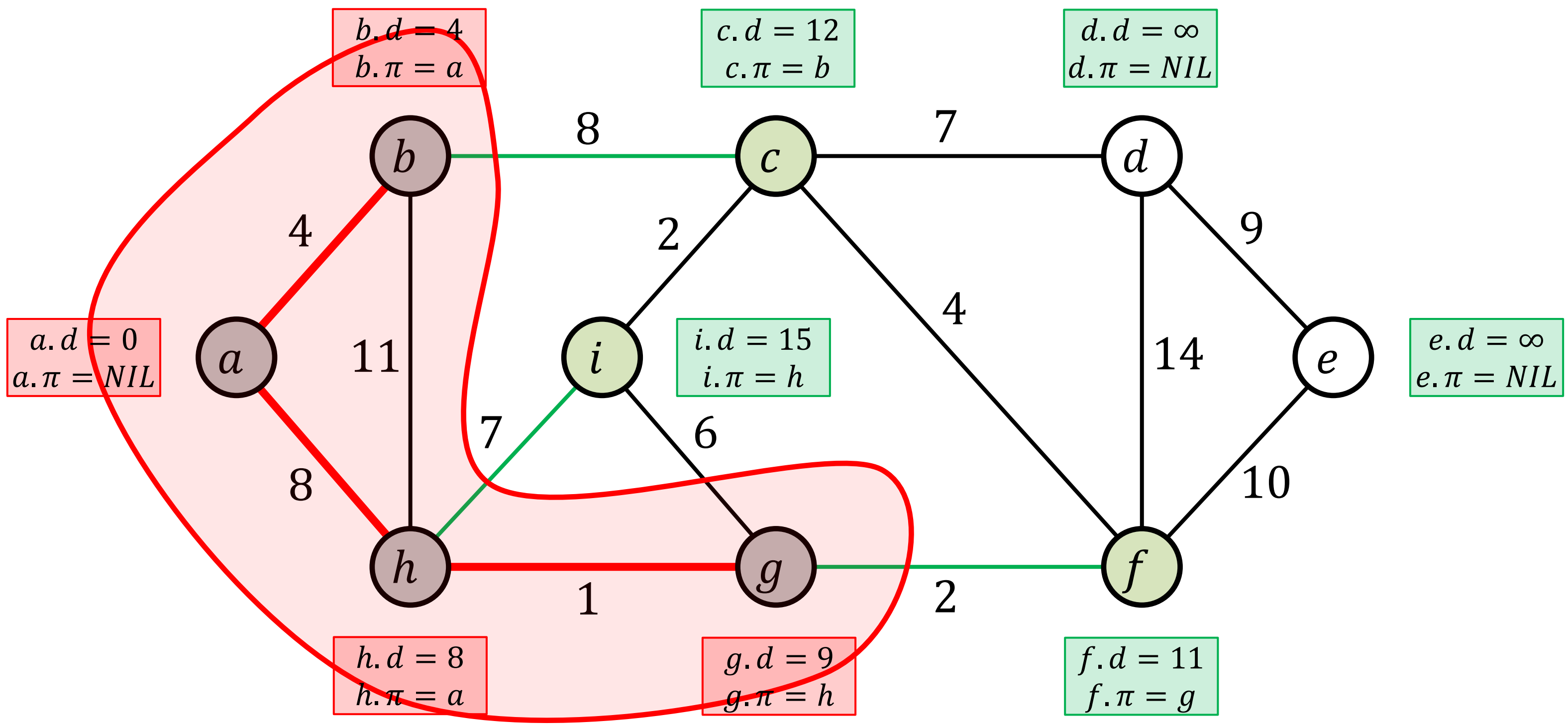
SSSP: Dijkstra's Algorithm

Step 4: add vertex g through edge (h, g)



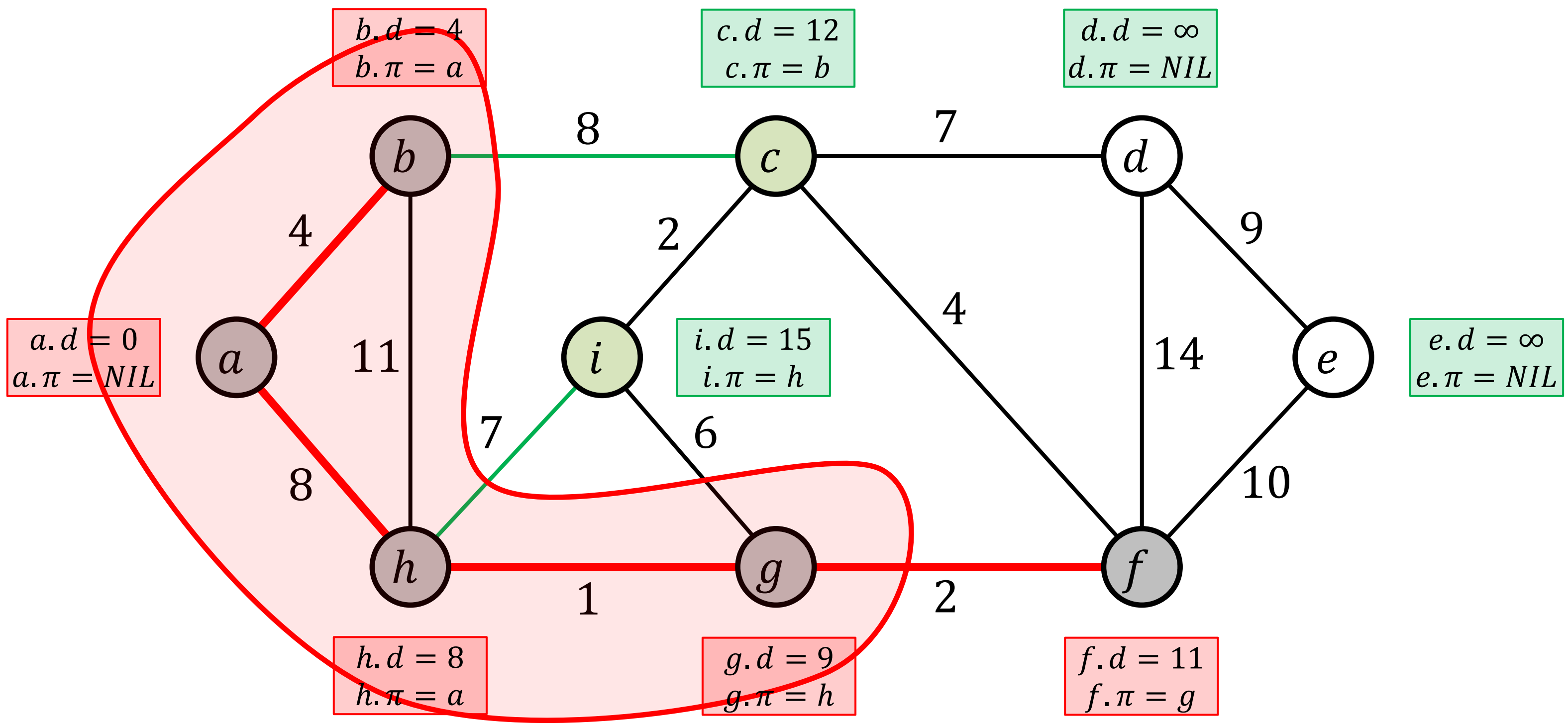
SSSP: Dijkstra's Algorithm

Step $4'$: update neighbors of g

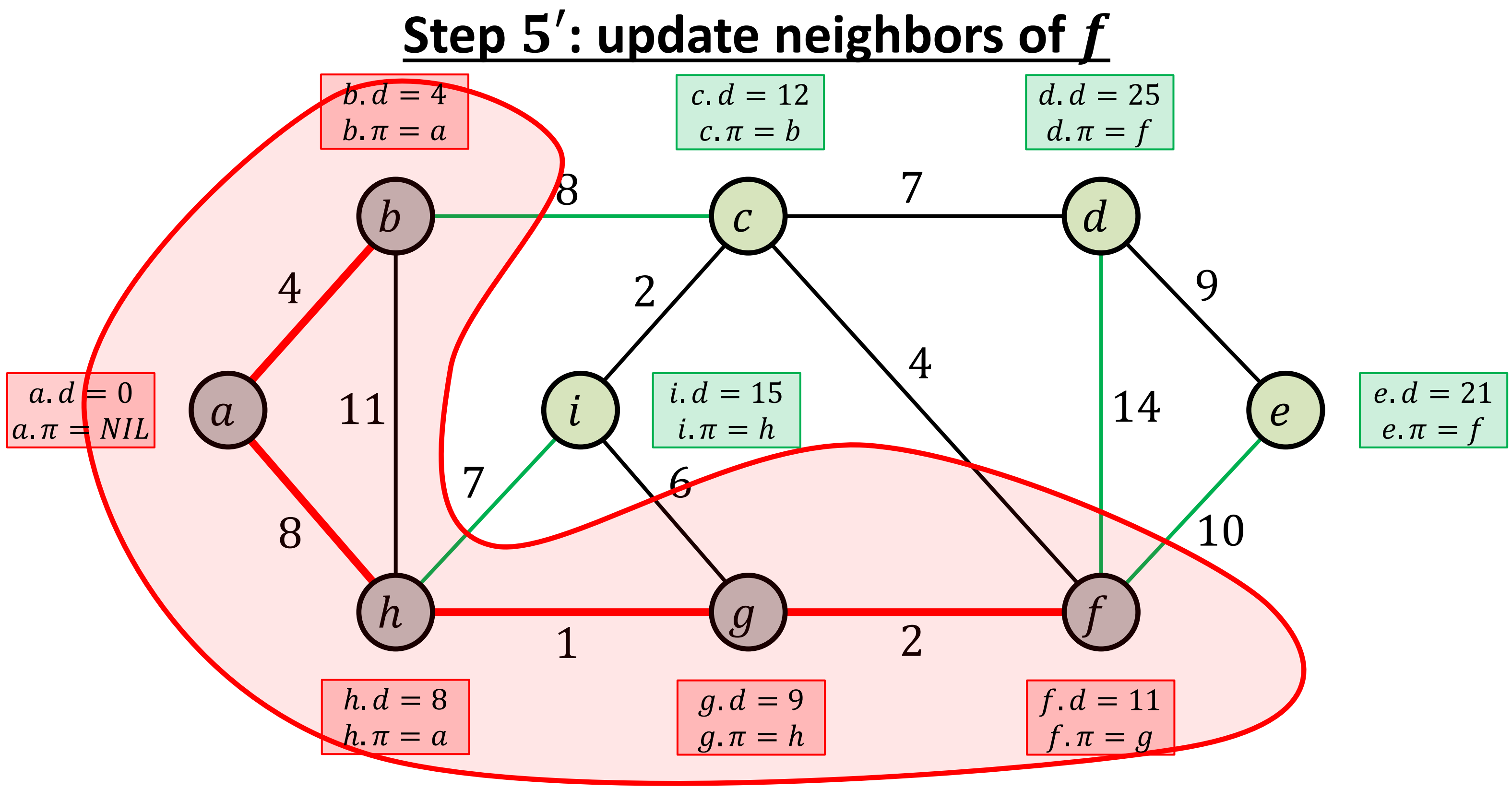


SSSP: Dijkstra's Algorithm

Step 5: add vertex f through edge (g, f)

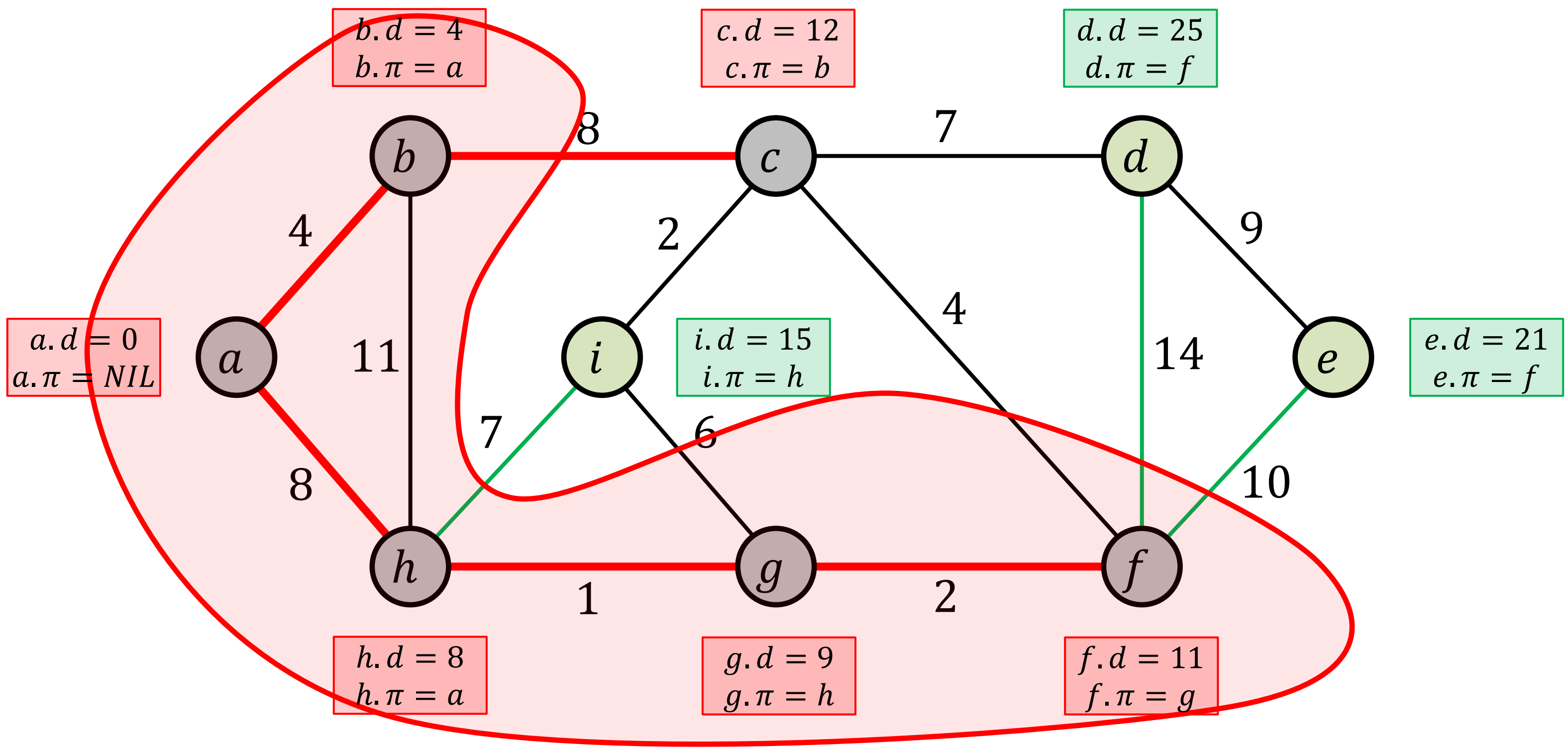


SSSP: Dijkstra's Algorithm

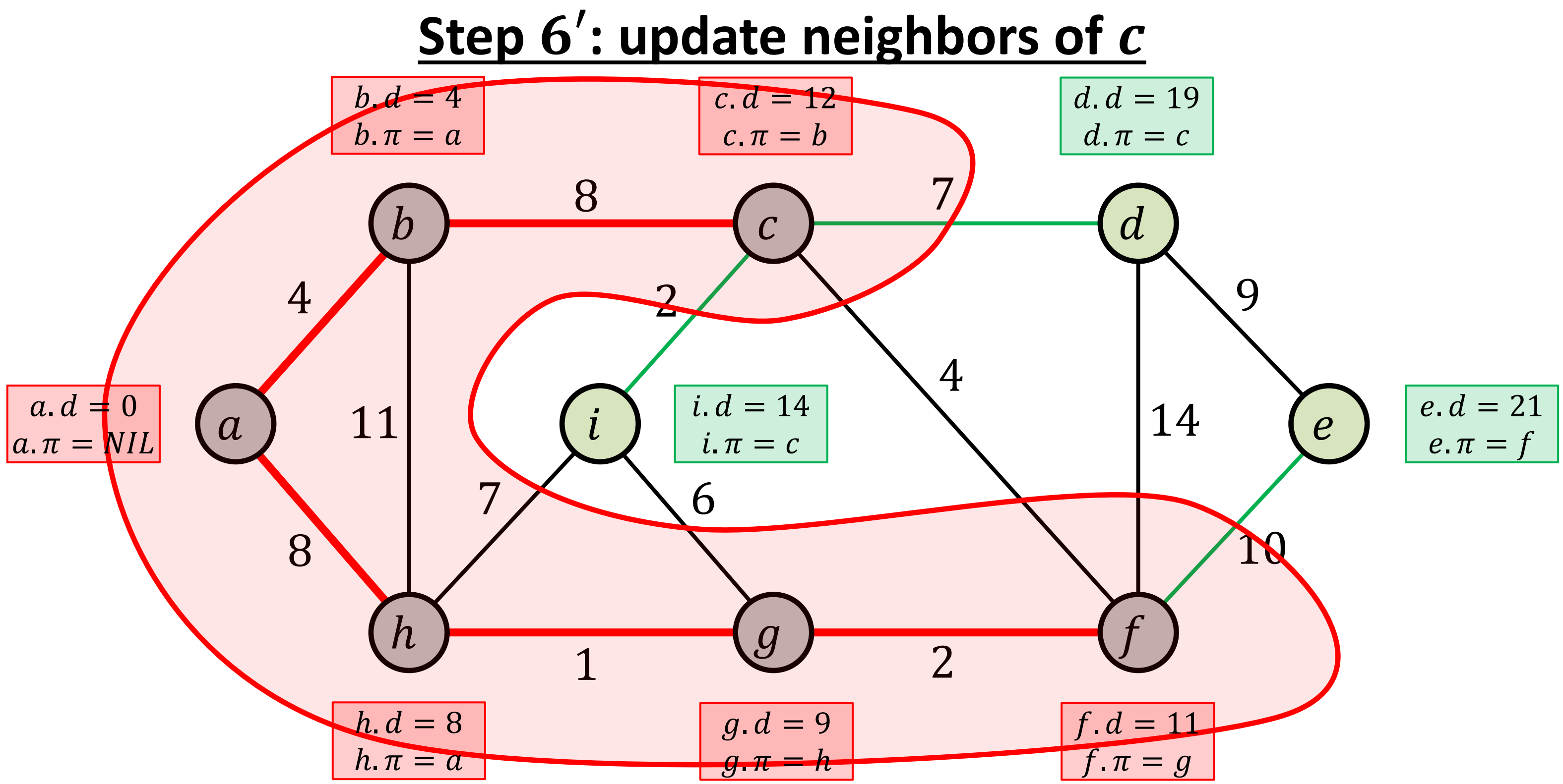


SSSP: Dijkstra's Algorithm

Step 6: add vertex c through edge (b, c)

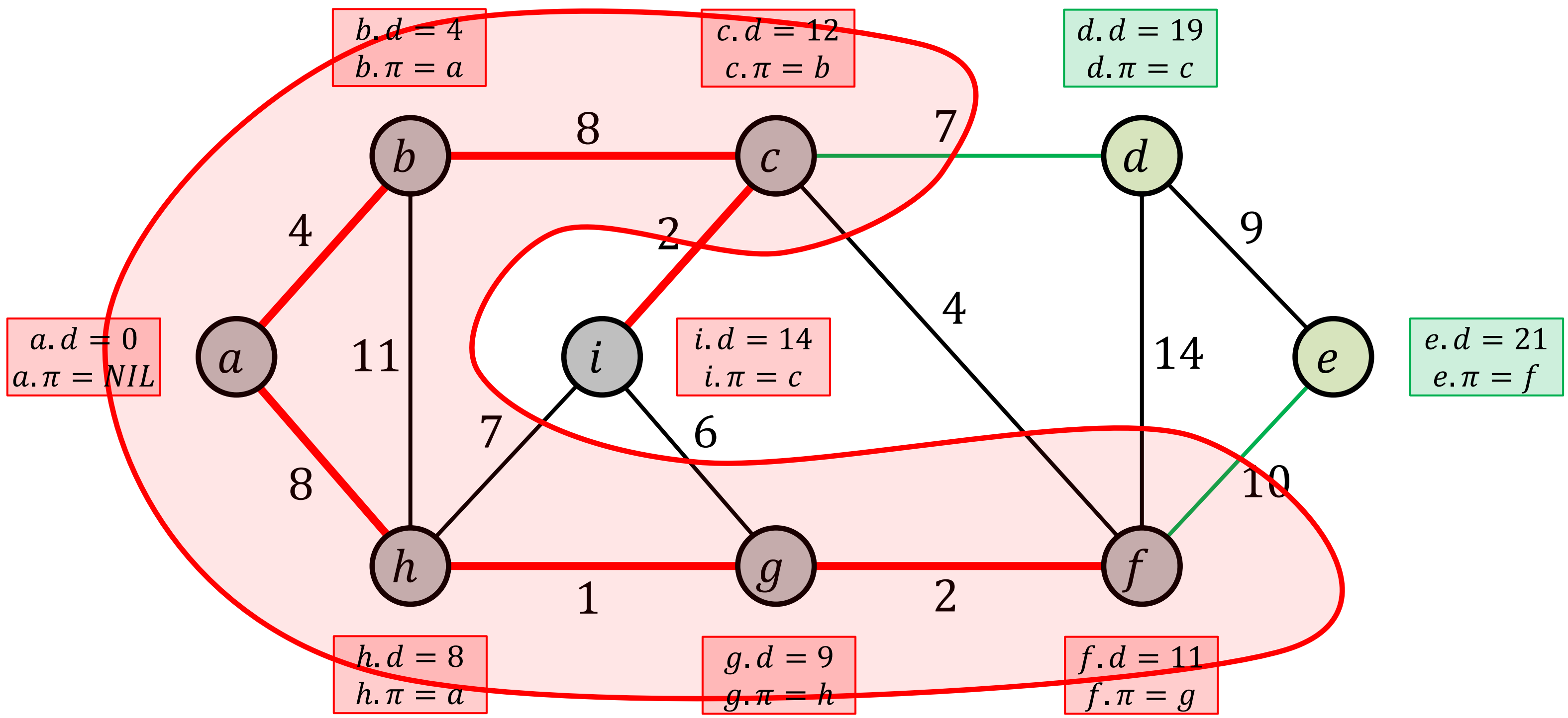


SSSP: Dijkstra's Algorithm



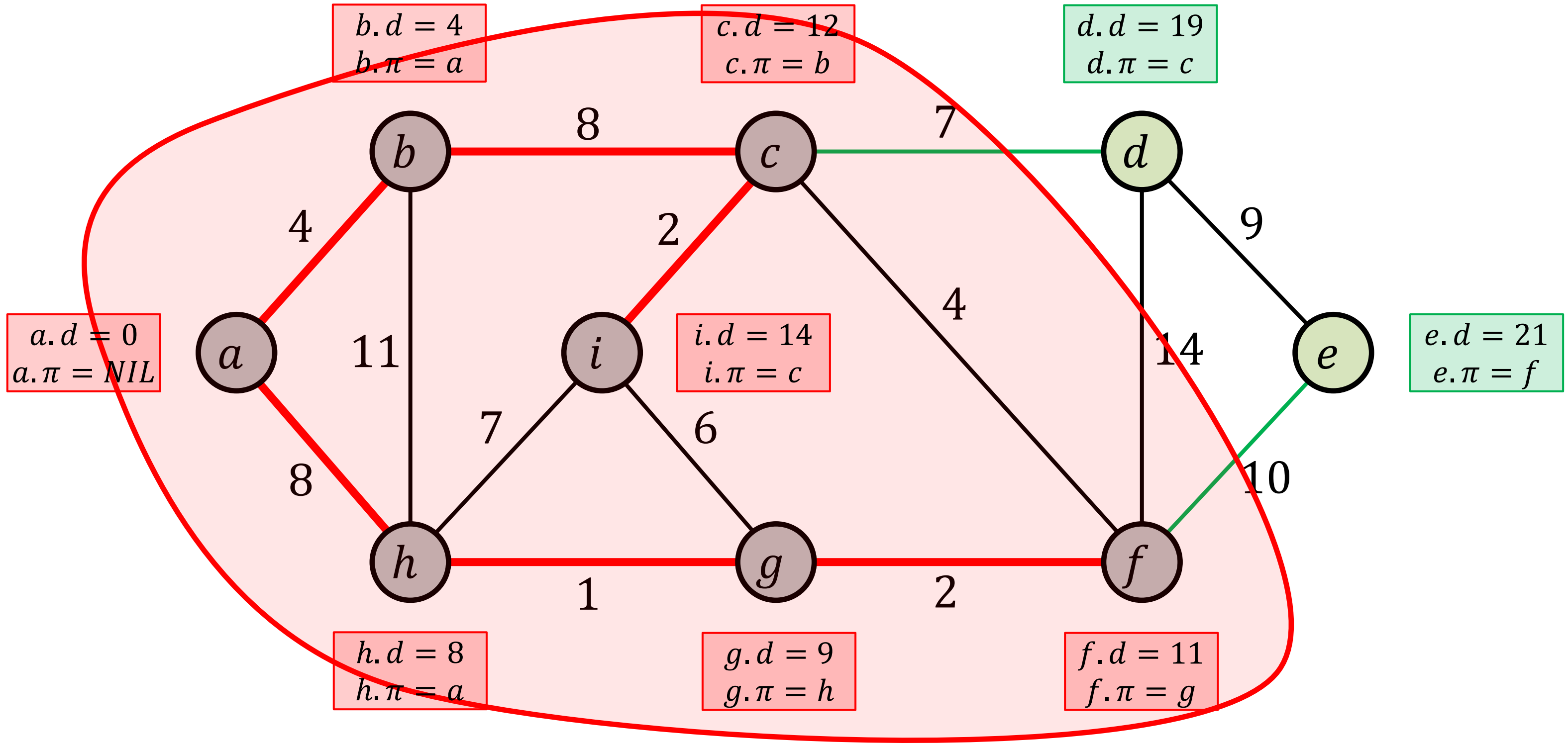
SSSP: Dijkstra's Algorithm

Step 7: add vertex i through edge (c, i)



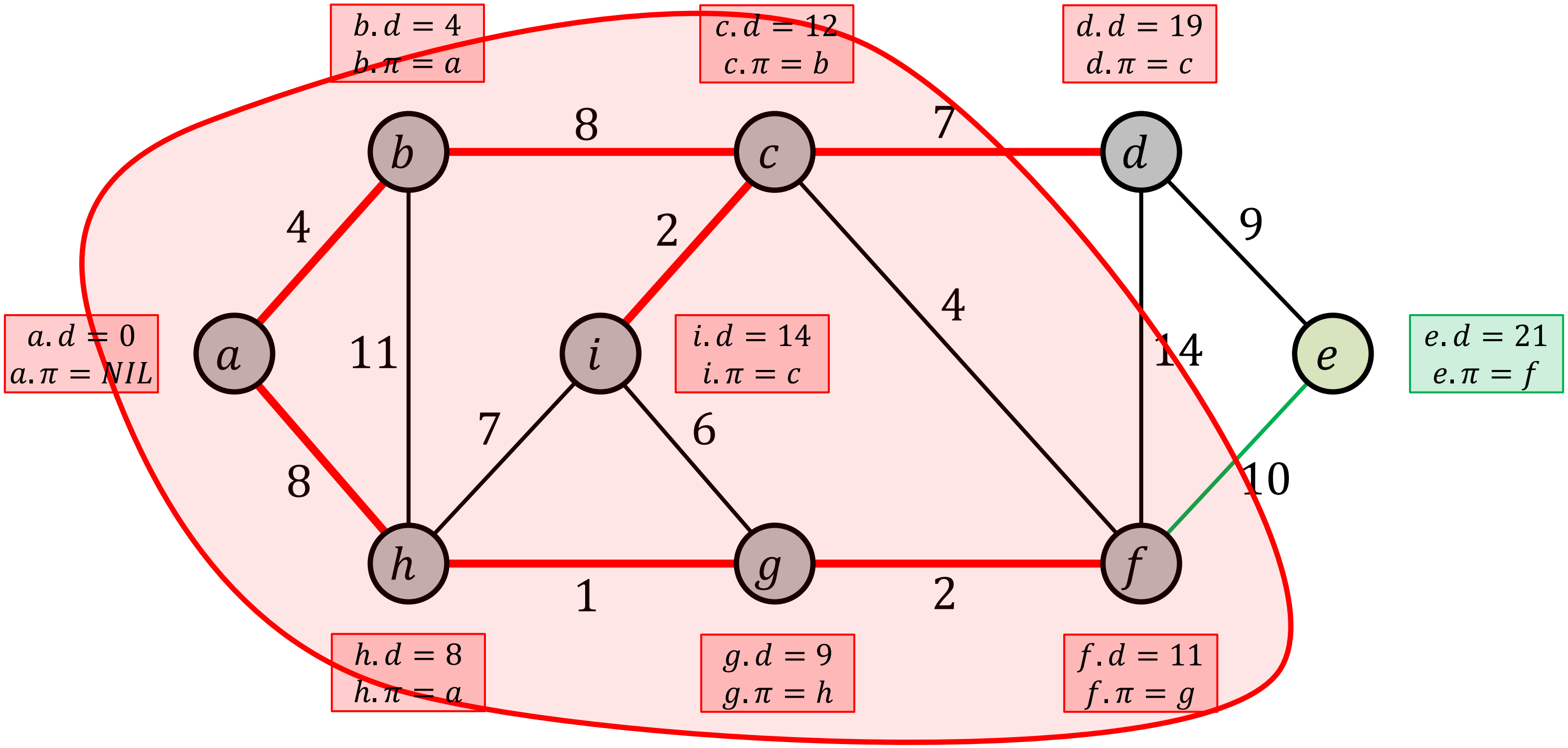
SSSP: Dijkstra's Algorithm

Step 7′: update neighbors of *i*



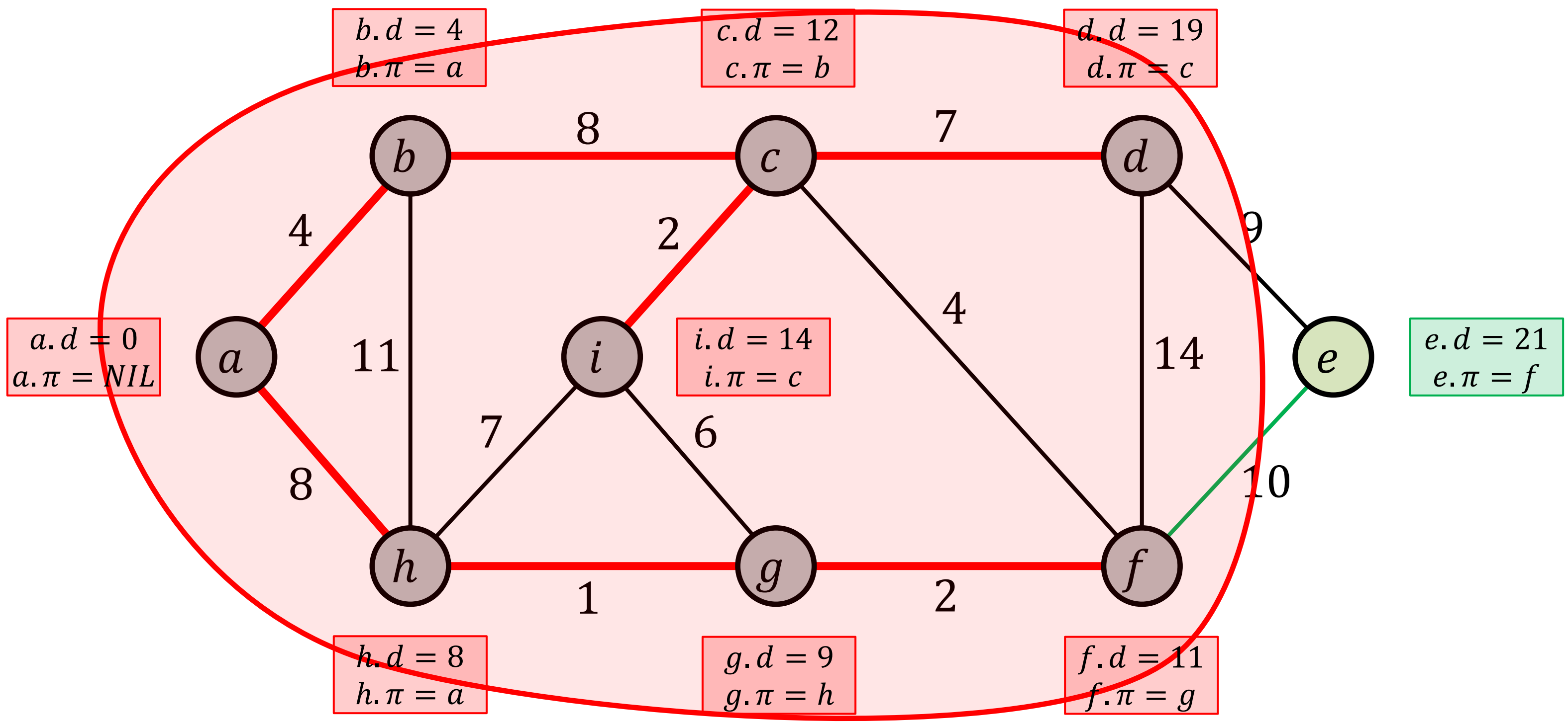
SSSP: Dijkstra's Algorithm

Step 8: add vertex d through edge (c, d)



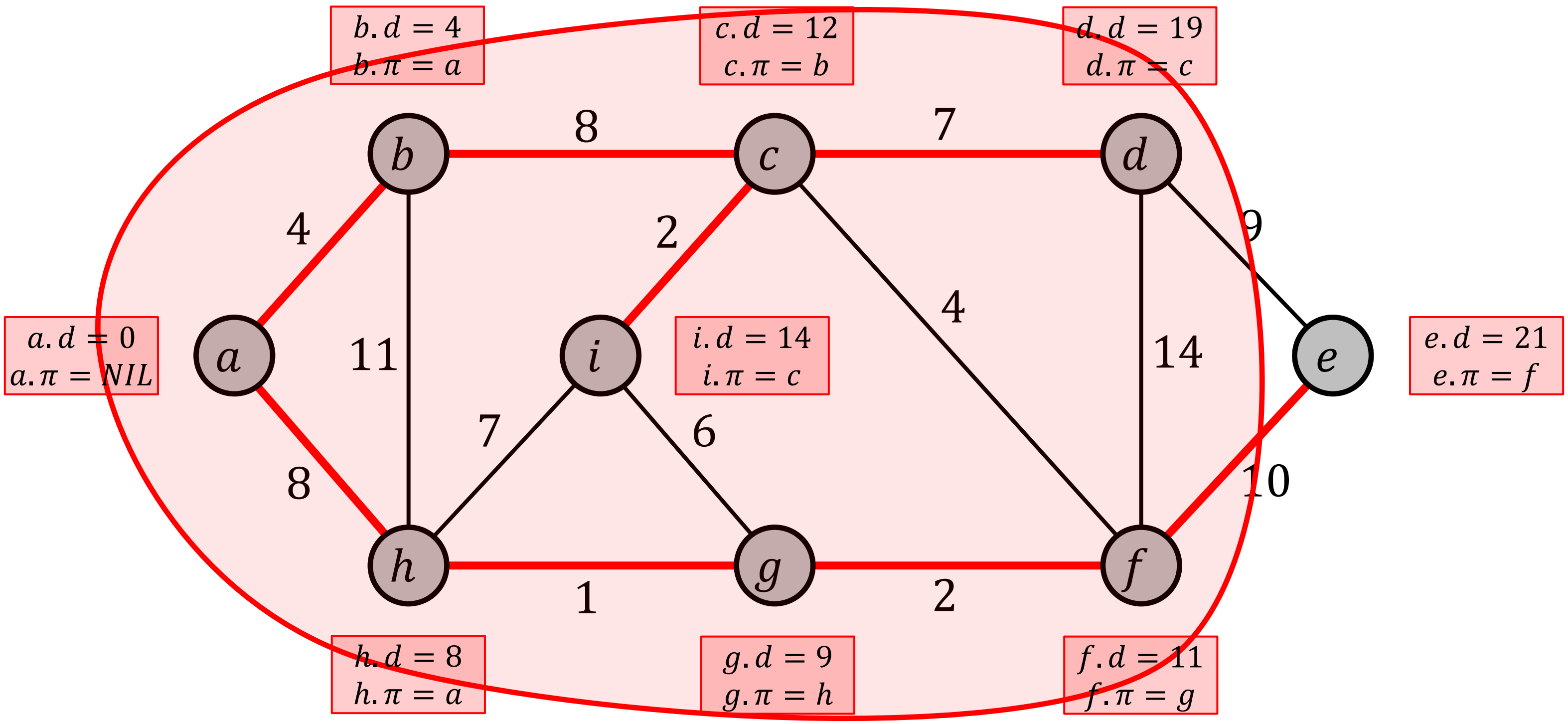
SSSP: Dijkstra's Algorithm

Step $8'$: update neighbors of d

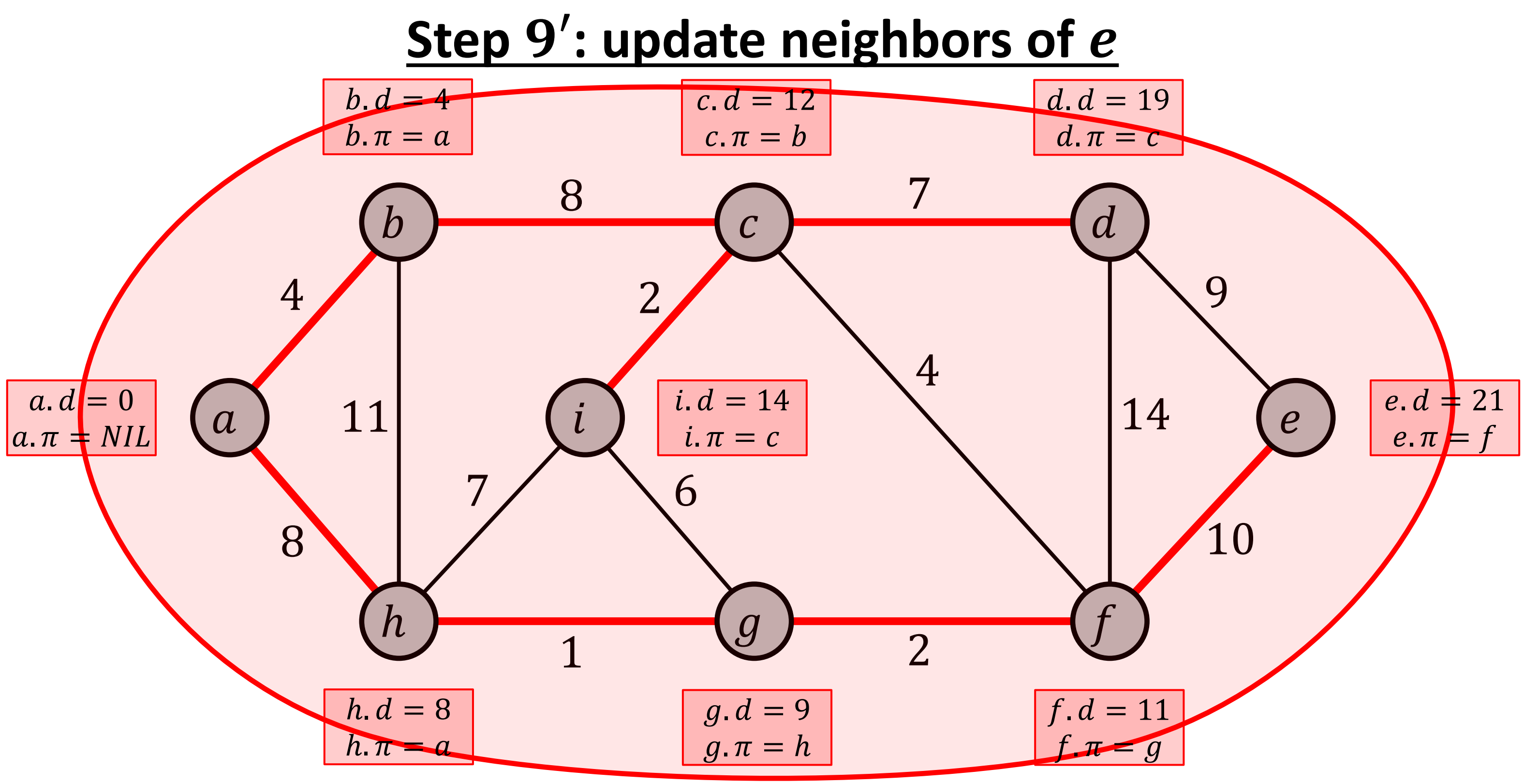


SSSP: Dijkstra's Algorithm

Step 9: add vertex e through edge (f, e)

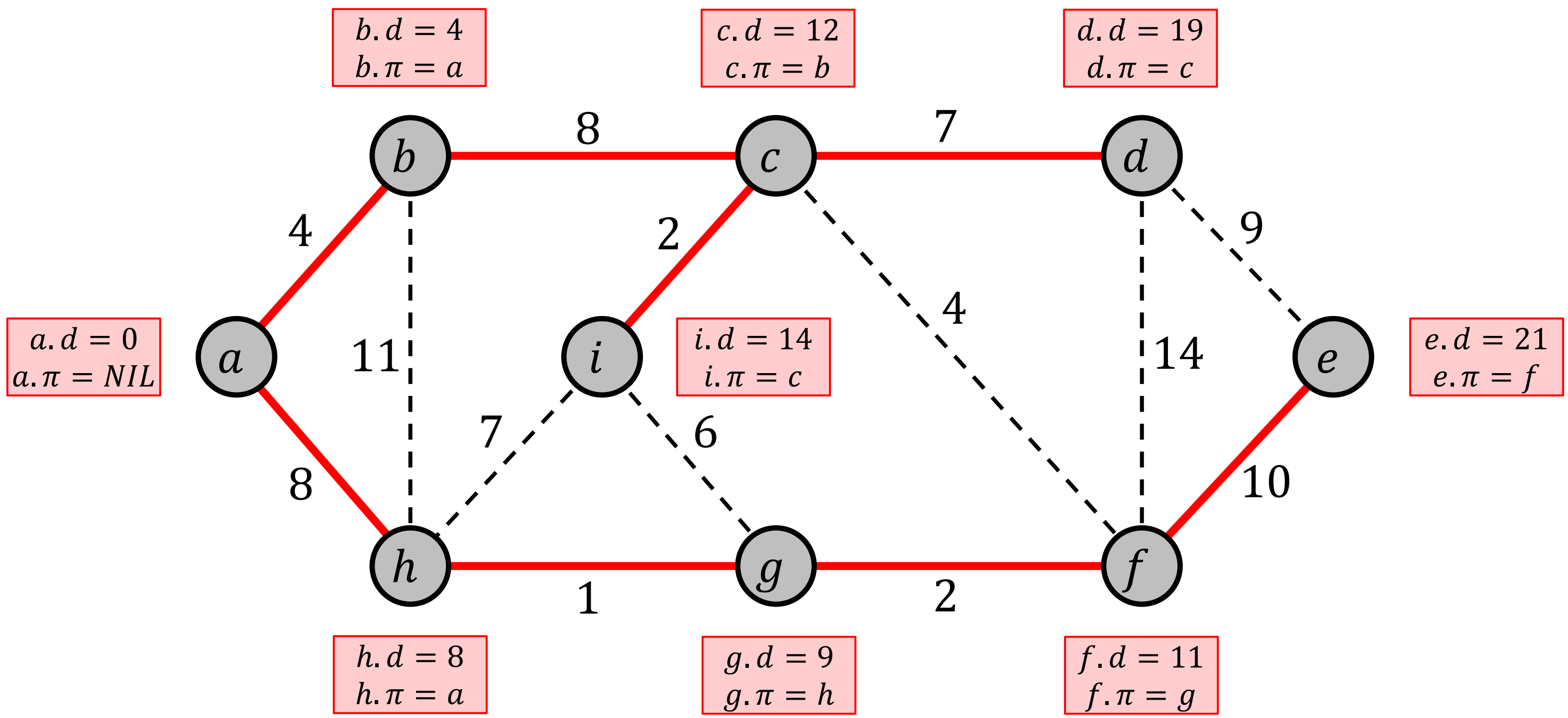


SSSP: Dijkstra's Algorithm



SSSP: Dijkstra's Algorithm

Done



Dijkstra's SSSP Algorithm with a Min-Heap
(SSSP: Single-Source Shortest Paths)

Input: Weighted graph $G = (V, E)$ with vertex set V and edge set E, a weight function w, and a source vertex $s \in G[V]$.

Output: For all $v \in G[V]$, $v.d$ is set to the shortest distance from s to v.

```
Dijkstra-SSSP ( G = (V,E), w, s )
    for each vertex v ∈ G.V do
        v.d ← ∞
        v.π ← NIL
    s.d ← 0
    Min-Heap Q ← ∅
    for each vertex v ∈ G.V do
        INSERT( Q, v )
    while Q ≠ ∅ do
        u ← EXTRACT-MIN( Q )
        for each (u,v) ∈ G.E do
            if u.d + w(u,v) < v.d then
                v.d ← u.d + w(u,v)
                v.π ← u
                DECREASE-KEY( Q, v, u.d + w(u,v) )
```

Let $n = |G[V]|$ and $m = |G[E]|$

\# *INSERTS* $= n$
\# *EXTRACT-MINS* $= n$
\# *DECREASE-KEYS* $\leq m$

Total cost

$$\leq n(cost_{Insert} + cost_{Extract-Min}) + m(cost_{Decrease-Key})$$

Dijkstra's SSSP Algorithm with a Min-Heap
(SSSP: Single-Source Shortest Paths)

Input: Weighted graph $G = (V, E)$ with vertex set V and edge set E, a weight function w, and a source vertex $s \in G[V]$.

Output: For all $v \in G[V]$, $v.d$ is set to the shortest distance from s to v.

```
Dijkstra-SSSP ( G = (V,E), w, s )

    for each vertex v ∈ G.V do
        v.d ← ∞
        v.π ← NIL
    s.d ← 0
    Min-Heap Q ← ∅
    for each vertex v ∈ G.V do
        INSERT( Q, v )
    while Q ≠ ∅ do
        u ← EXTRACT-MIN( Q )
        for each (u,v) ∈ G.E do
            if u.d + w(u,v) < v.d then
                v.d ← u.d + w(u,v)
                v.π ← u
                DECREASE-KEY( Q, v, u.d + w(u,v) )
```

Let $n = |G[V]|$ and $m = |G[E]|$

For Binary Heap (worst-case costs):

$$cost_{Insert} = O(\log n)$$
$$cost_{Extract-Min} = O(\log n)$$
$$cost_{Decrease-Key} = O(\log n)$$

∴ Total cost (worst-case)

$$= O\big((m + n)\log n\big)$$

Dijkstra's SSSP Algorithm with a Min-Heap
(SSSP: Single-Source Shortest Paths)

Input: Weighted graph $G = (V, E)$ with vertex set V and edge set E, a weight function w, and a source vertex $s \in G[V]$.

Output: For all $v \in G[V]$, $v.d$ is set to the shortest distance from s to v.

Dijkstra-SSSP ($G = (V, E),\ w,\ s$)

for each vertex $v \in G.V$ *do*
    $v.d \leftarrow \infty$
    $v.\pi \leftarrow NIL$
$s.d \leftarrow 0$
Min-Heap $Q \leftarrow \emptyset$
for each vertex $v \in G.V$ *do*
    INSERT(Q, v)
while $Q \neq \emptyset$ *do*
    $u \leftarrow$ EXTRACT-MIN(Q)
    for each $(u, v) \in G.E$ *do*
        if $u.d + w(u, v) < v.d$ *then*
            $v.d \leftarrow u.d + w(u, v)$
            $v.\pi \leftarrow u$
            DECREASE-KEY($Q,\ v,\ u.d + w(u, v)$)

Let $n = |G[V]|$ and $m = |G[E]|$

For Fibonacci Heap (amortized):

$$cost_{Insert} = O(1)$$
$$cost_{Extract-Min} = O(\log n)$$
$$cost_{Decrease-Key} = O(1)$$

∴ Total cost (amortized)

$$= O(m + n \log n)$$

<u>Optional</u>
Kruskal's MST algorithm and a Union-Find data structure with union by rank and path compression

A Disjoint-Set Data Structure (union by rank and path compression)

Make-Set (x)

$\pi(x) \leftarrow x$
$rank(x) \leftarrow 0$

Link (x, y)

if $rank(x) > rank(y)$ *then* $\pi(y) \leftarrow x$
else $\pi(x) \leftarrow y$
    if $rank(x) = rank(y)$ *then* $rank(y) \leftarrow rank(y) + 1$

Union (x, y)

Link (Find (x), Find (y))

Find (x)

if $x \neq \pi(x)$ *then* $\pi(x) \leftarrow$ Find ($\pi(x)$)
return $\pi(x)$

A Disjoint-Set Data Structure
(union by rank and path compression)

THEOREM: A sequence of N MAKE-SET, UNION and FIND operations of which exactly n ($\leq N$) are MAKE-SET operations takes $\text{O}\big(N\alpha(n)\big)$ time to execute, where $\alpha(n)$ is the extremely slowly growing *Inverse Ackermann Function* which has a value no larger than 3 for all practical values of n.

MST: Kruskal's Algorithm
(union by rank and path compression)

MST-Kruskal ($G = (V, E)$, w)

1. $A \leftarrow \emptyset$
2. *for* each vertex $v \in G.V$ *do*
3.   *Make-Set*(v)
4. sort the edges of $G.E$ into nondecreasing order by weight w
5. *for* each edge $(u, v) \in G.E$ taken in nondecreasing order by weight *do*
6.   *if Find*(u) $\neq$ *Find*(v) *then*
7. $A \leftarrow A \cup \{(u, v)\}$
8.   *Union*(u, v)
9. *return* A

Let $n = |V|$ and $m = |E|$. Since G is connected, we have $m \geq n - 1$.

Then the sorting in step 4 can be done in $\mathrm{O}(m \log m)$ time.

#disjoint-set operations performed, $N = 2m + 2n - 1$, of which

#Make-Set: n, #Find: $2m$, #Union: $n - 1$

So, total time taken by disjoint-set operations $= \mathrm{O}\big((n + m)\alpha(n)\big)$

Hence, MST-Kruskal's running time $= \mathrm{O}(m \log m)$