

# 图算法篇：最小生成树之Kruskal算法

---

童咏昕

北京航空航天大学  
计算机学院

中国大学MOOC北航《算法设计与分析》

问题的回顾

算法与实例

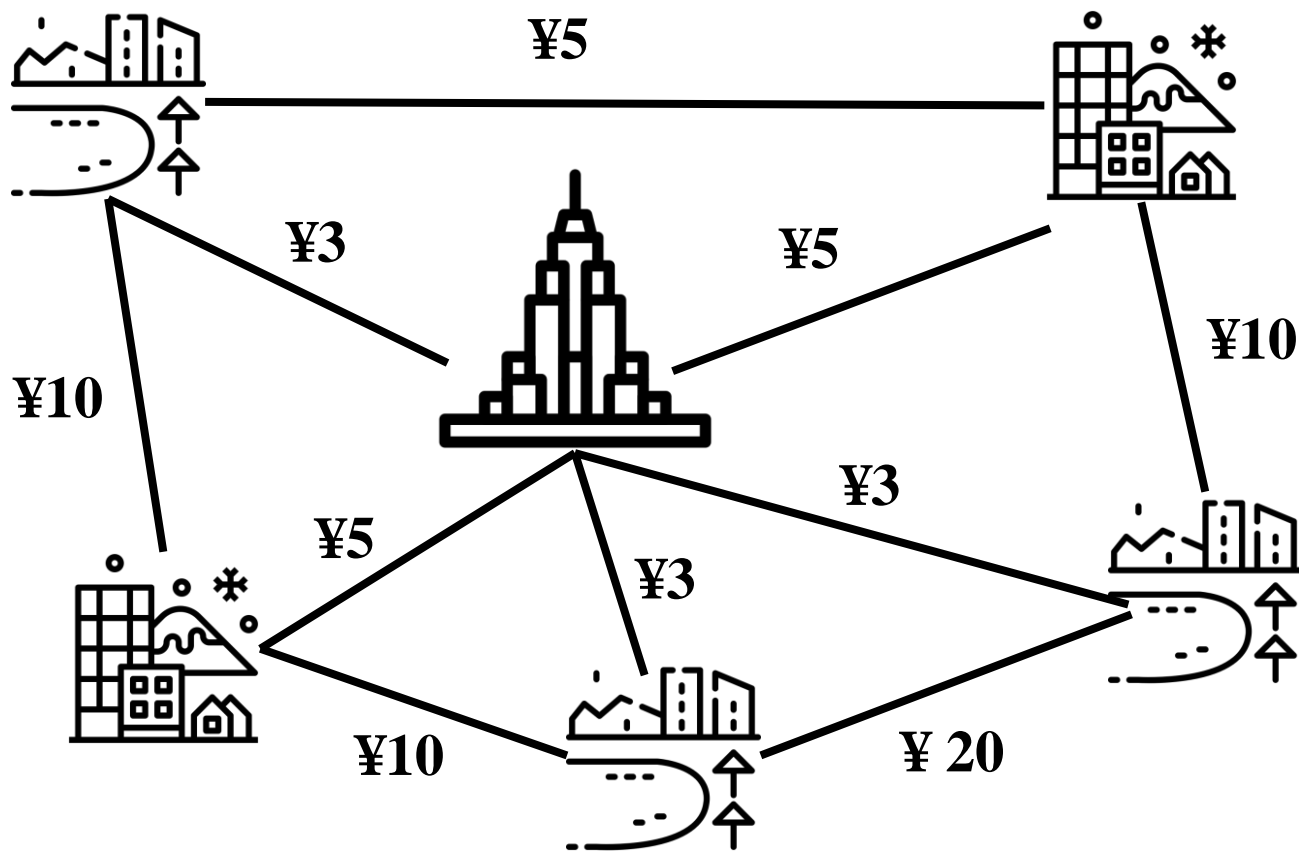
正确性证明

不相交集合

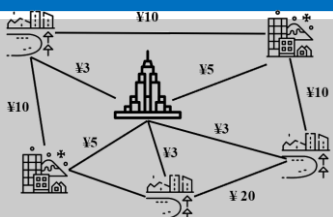
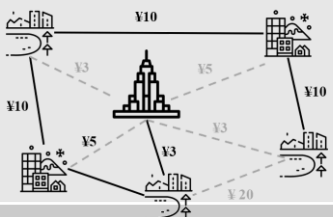
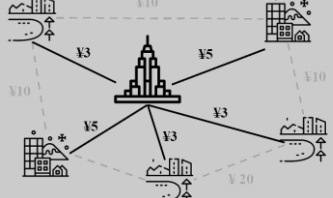
复杂度分析

# 背景回顾：道路修建

- 需要修建道路连通城市，各道路花费不同



问题：连通各城市的最小花费是多少？

| 方案  | 花费  |
|---|-----|
|    | ¥74 |
|   | ¥38 |
|  | ¥19 |

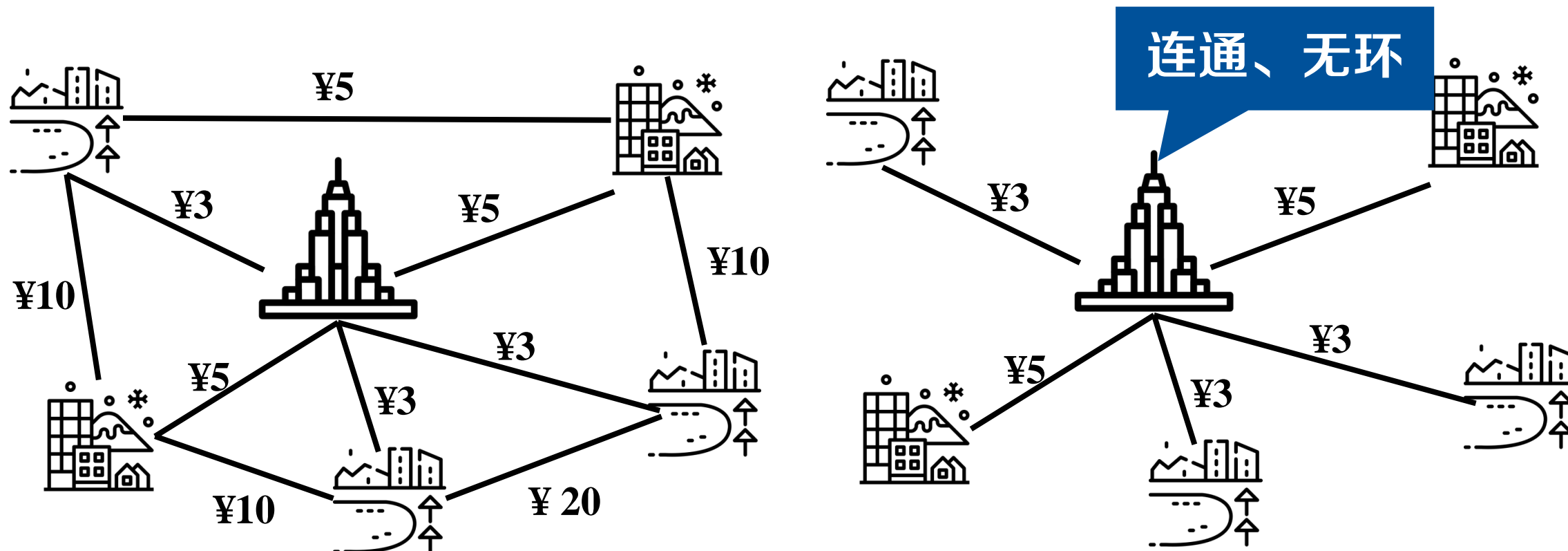
权重最小的连通生成子图

# 概念回顾：生成树



- 生成树(Spanning Tree)

- 图  $T' = \langle V', E' \rangle$  是无向图  $G$  的一个生成子图，并且是连通、无环路的(树)



问题：连通各城市的最小花费是多少？

权重最小的生成树

## 最小生成树问题

### Minimum Spanning Tree Problem

#### 输入

- 连通无向图  $G = \langle V, E, W \rangle$ , 其中  $w(u, v) \in W$  表示边  $(u, v)$  的权重

#### 输出

- 图  $G$  的最小生成树  $T = \langle V_T, E_T \rangle$

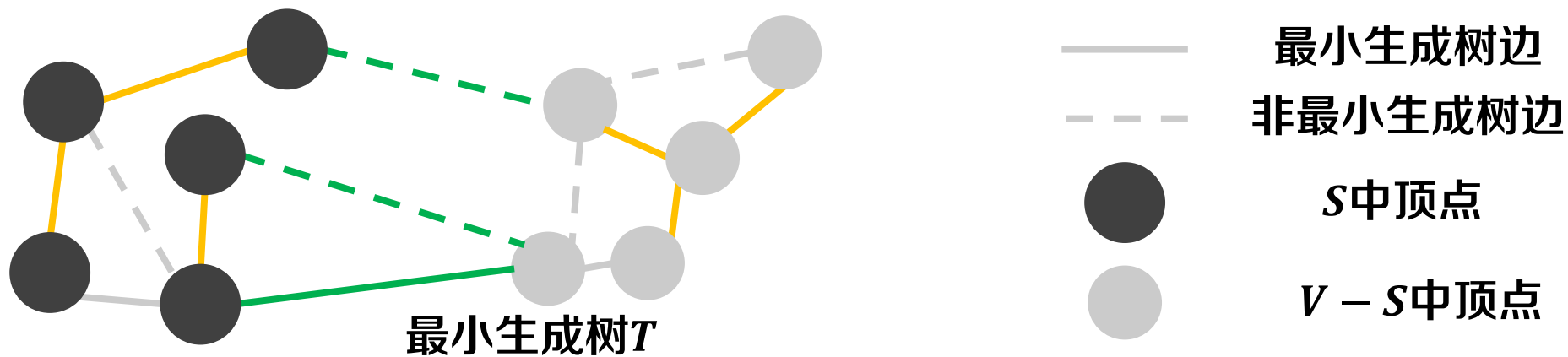
$$\min \sum_{e \in E_T} w(e)$$

优化目标

$$s. t. \quad V_T = V, E_T \subseteq E$$

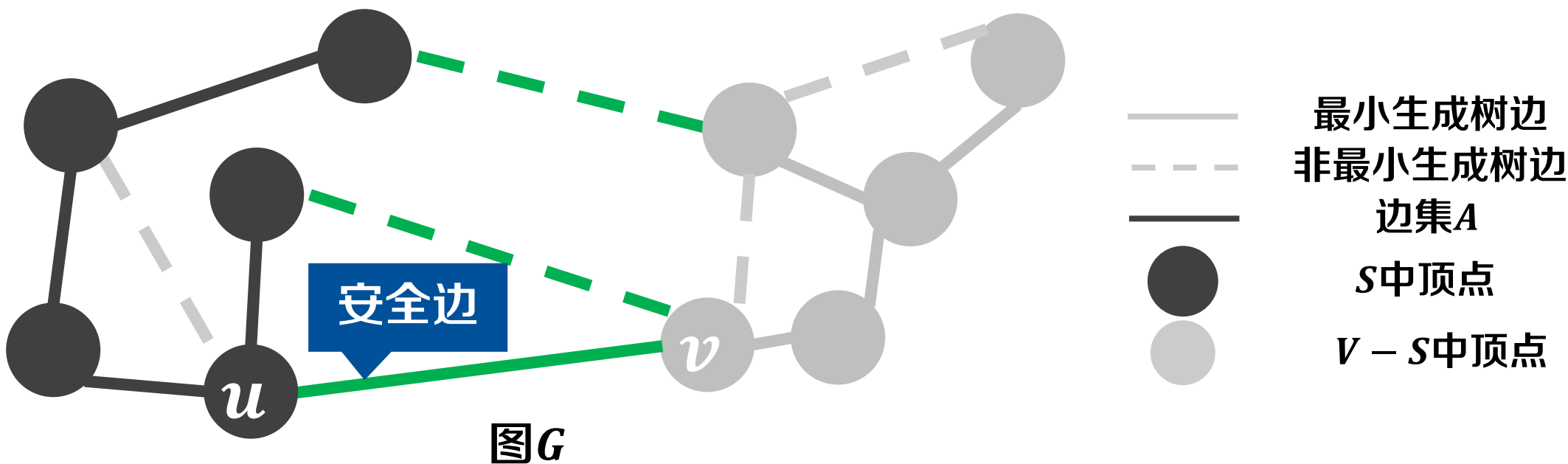
约束条件

- 割(Cut)
  - 图 $G = \langle V, E \rangle$ 是一个连通无向图，割 $(S, V - S)$ 将图 $G$ 的顶点集 $V$ 划分为两部分
- 横跨(Cross)
  - 给定割 $(S, V - S)$ 和边 $(u, v)$ ， $u \in S$ ， $v \in V - S$ ，称边 $(u, v)$ 横跨割 $(S, V - S)$
- 轻边(Light Edge)
  - 横跨割的所有边中，权重最小的称为横跨这个割的一条轻边
- 不妨害(Respect)
  - 如果一个边集 $A$ 中没有边横跨某割，则称该割不妨害边集 $A$



# 框架回顾：安全边辨识定理

- 给定图  $G = \langle V, E \rangle$  是一个带权的连通无向图，令  $A$  为边集  $E$  的一个子集，且  $A$  包含在图  $G$  的某棵最小生成树中
  - 若割  $(S, V - S)$  是图  $G$  中不妨害边集  $A$  的任意割，且  $(u, v)$  是横跨该割的轻边
  - 则对于边集  $A$ ，边  $(u, v)$  是其安全边



# 框架回顾：通用框架

- 生成树是一个连通、无环的生成子图
  - 新建一个空边集 $A$ ，边集 $A$ 可逐步扩展为最小生成树
  - 每次向边集 $A$ 中新增加一条边
    - 需保证边集 $A$ 仍是一个无环图
    - 需保证边集 $A$ 仍是最小生成树的子集

添加一条轻边

问题：如何有效地实现此贪心策略？

Prim算法

Kruskal算法



# 框架回顾：通用框架

- 生成树是一个连通、无环的生成子图
  - 新建一个空边集 $A$ ，边集 $A$ 可逐步扩展为最小生成树
  - 每次向边集 $A$ 中新增加一条边
    - 需保证边集 $A$ 仍是一个无环图
    - 需保证边集 $A$ 仍是最小生成树的子集

添加一条轻边

问题：如何有效地实现此贪心策略？

Prim算法

Kruskal算法

问题的回顾

算法与实例

正确性证明

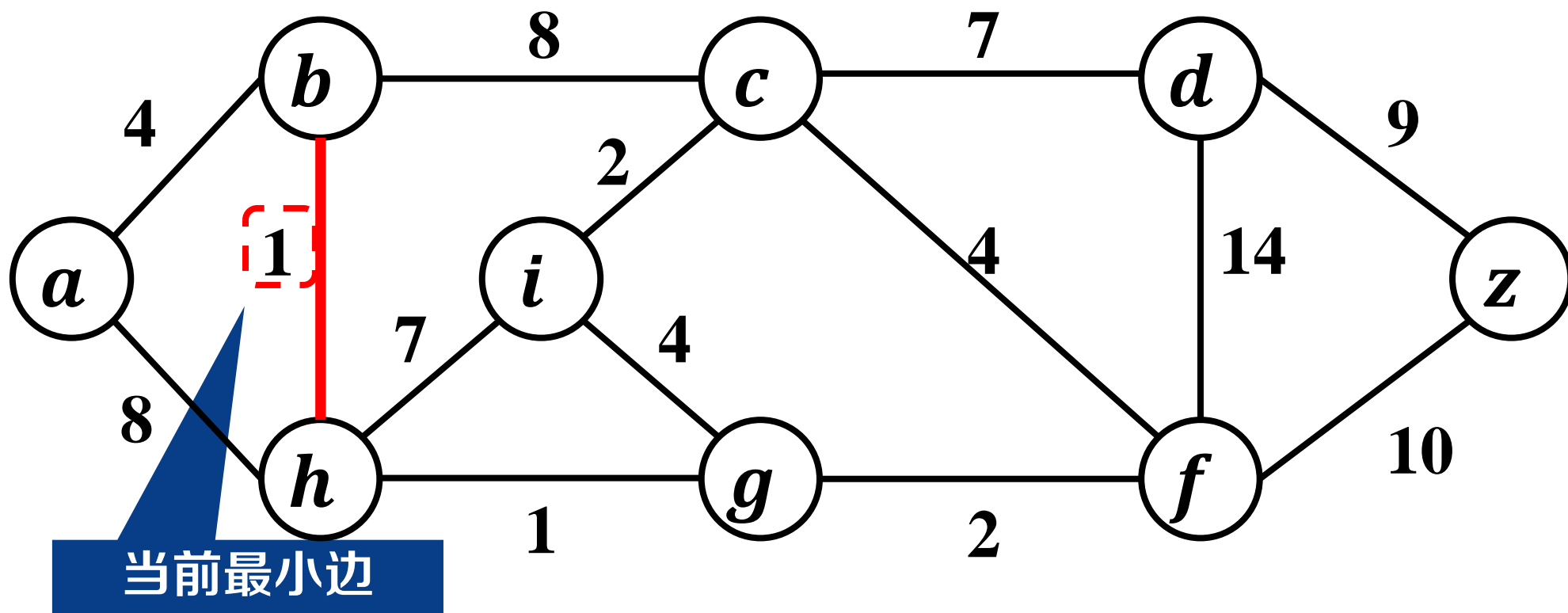
不相交集合

复杂度分析

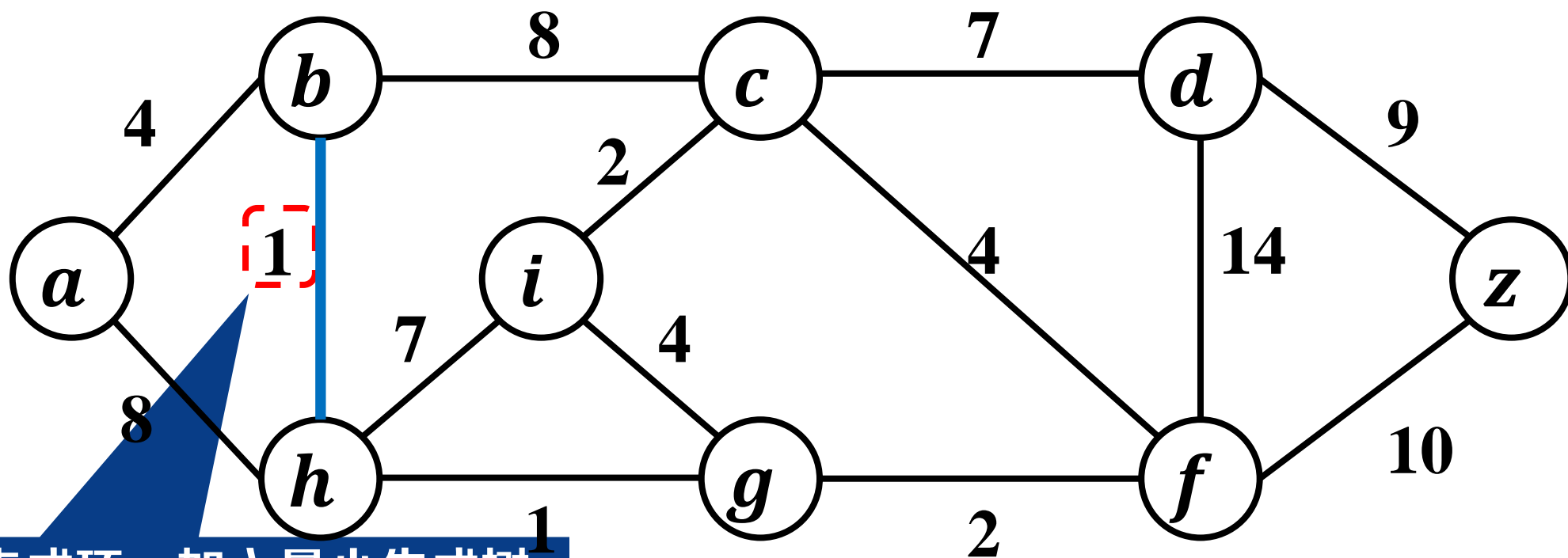
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
  - 需保证边集 $A$ 仍是最小生成树的子集

- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边

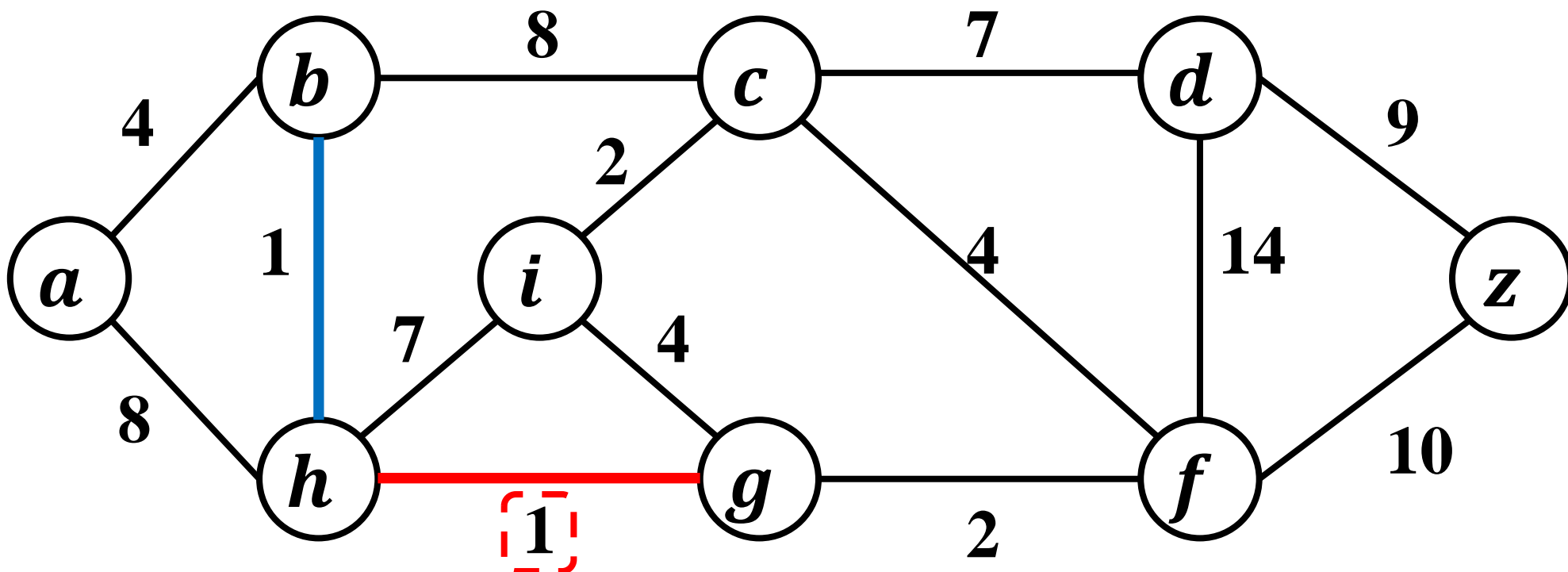
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



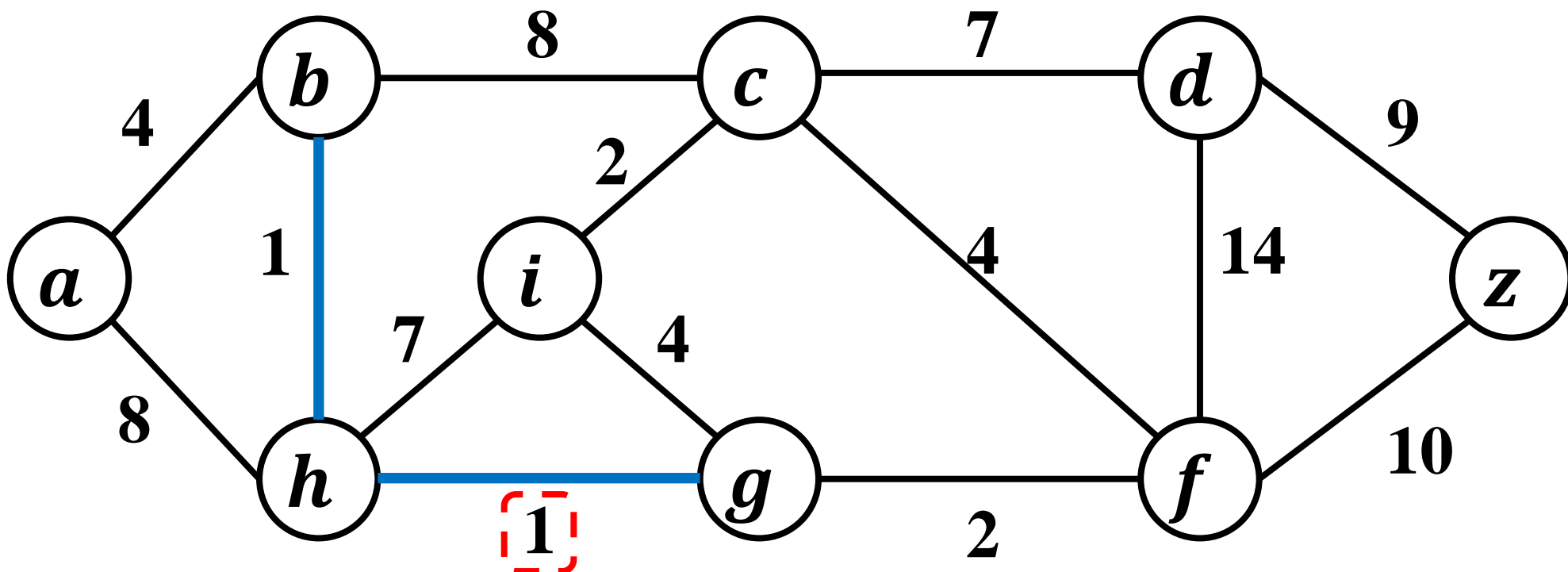
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边

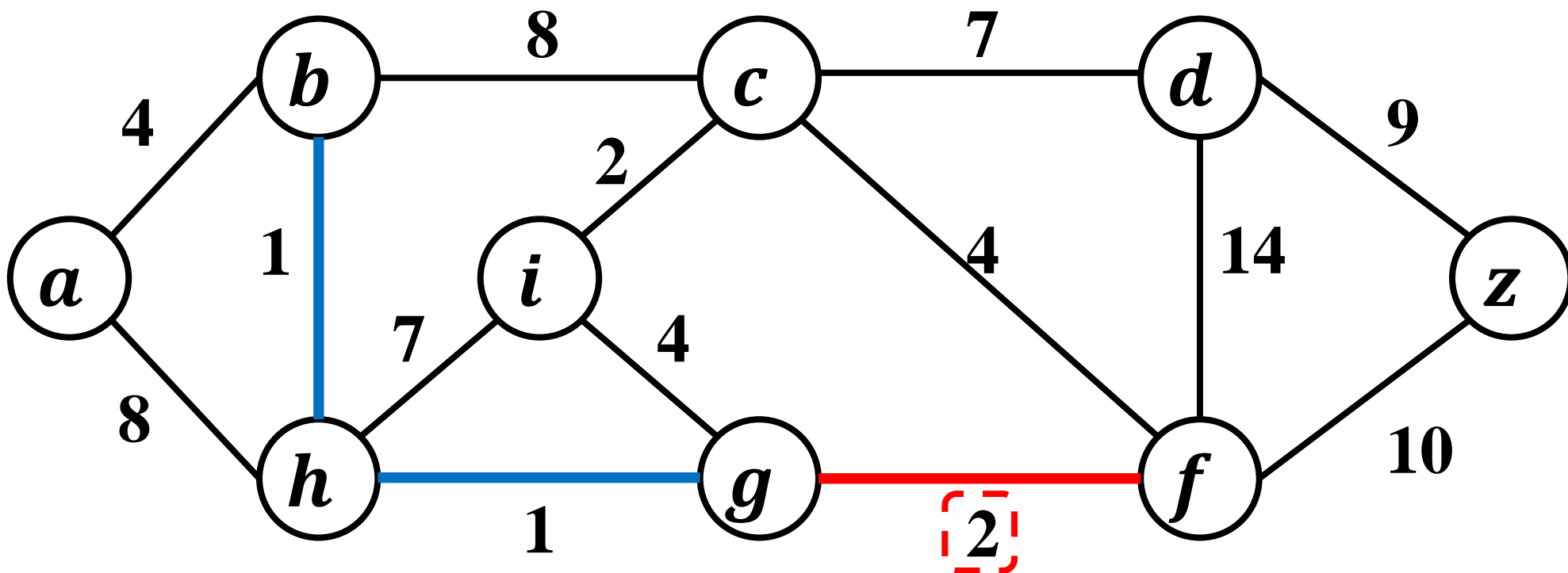


- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边

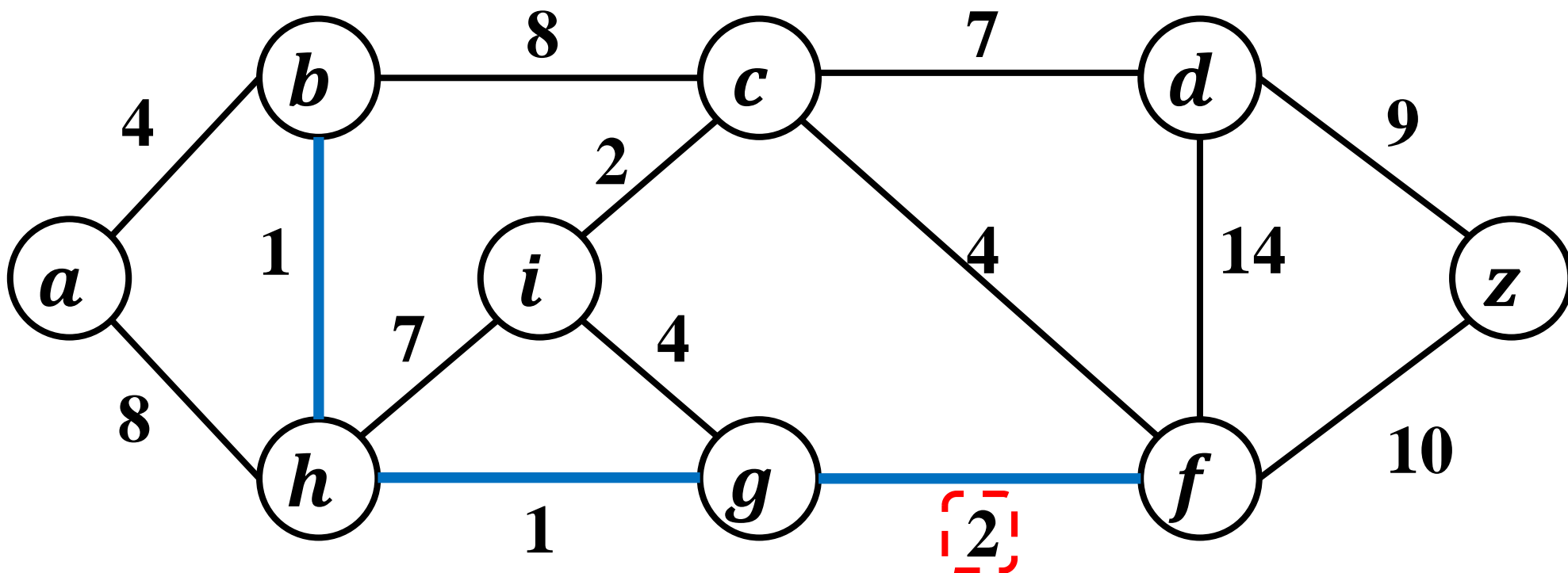




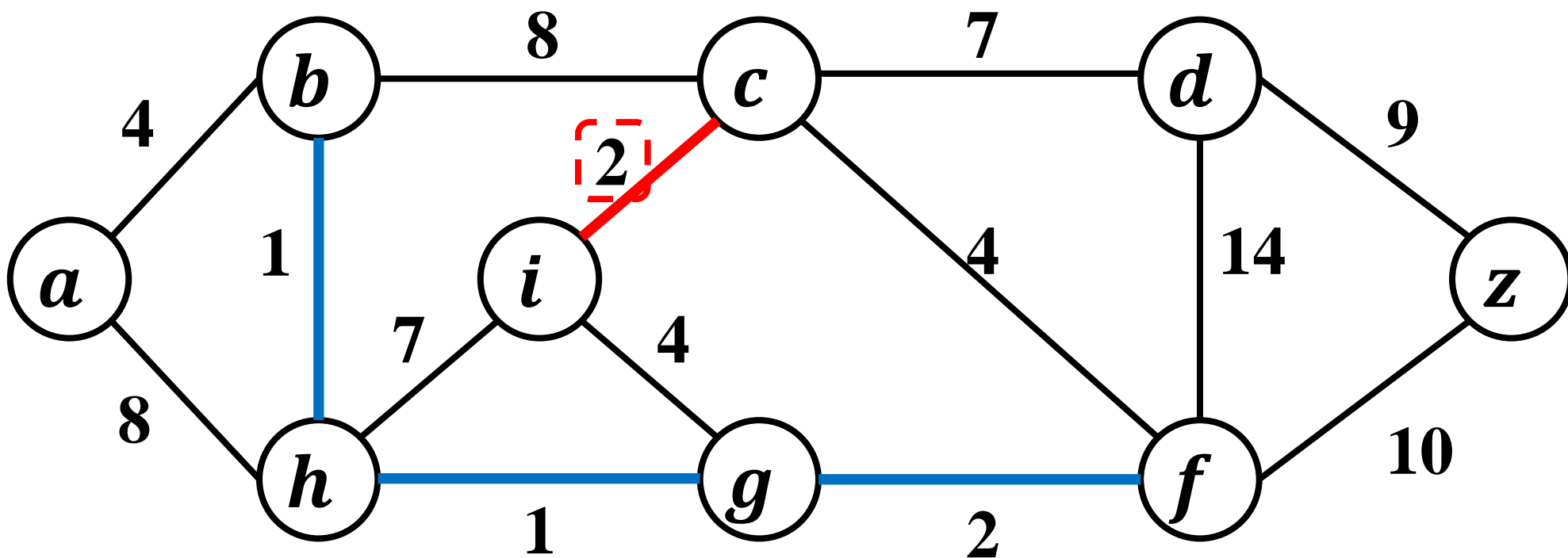
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



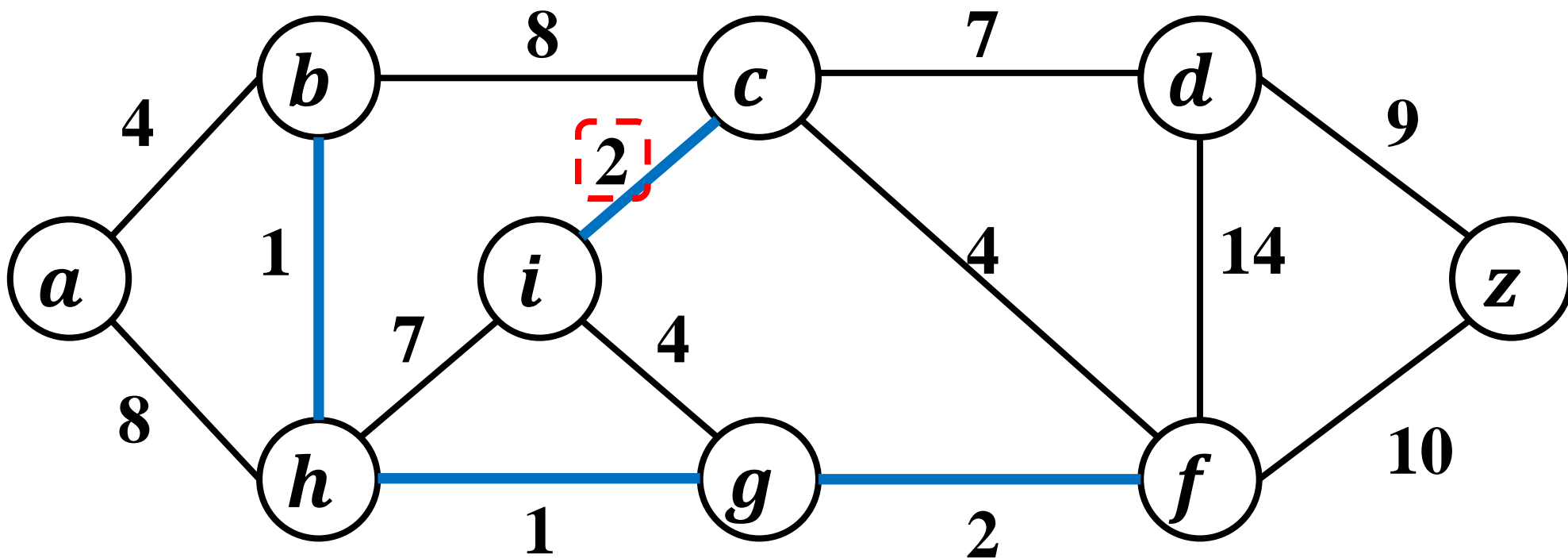
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



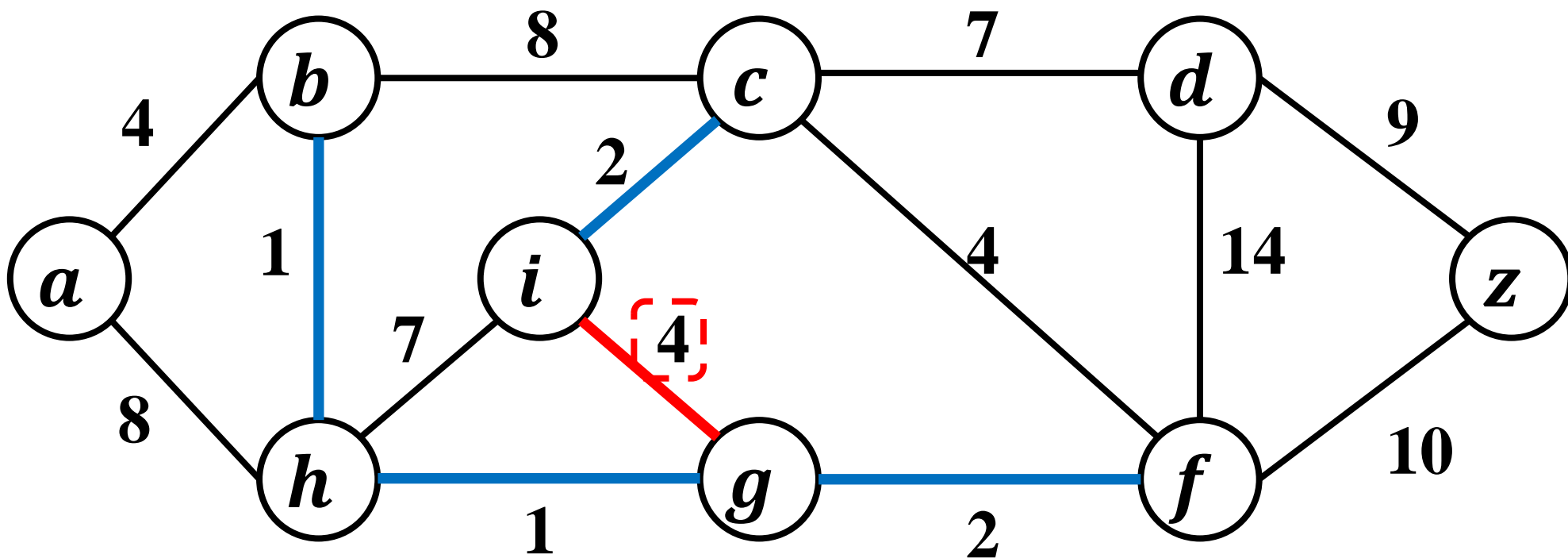
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



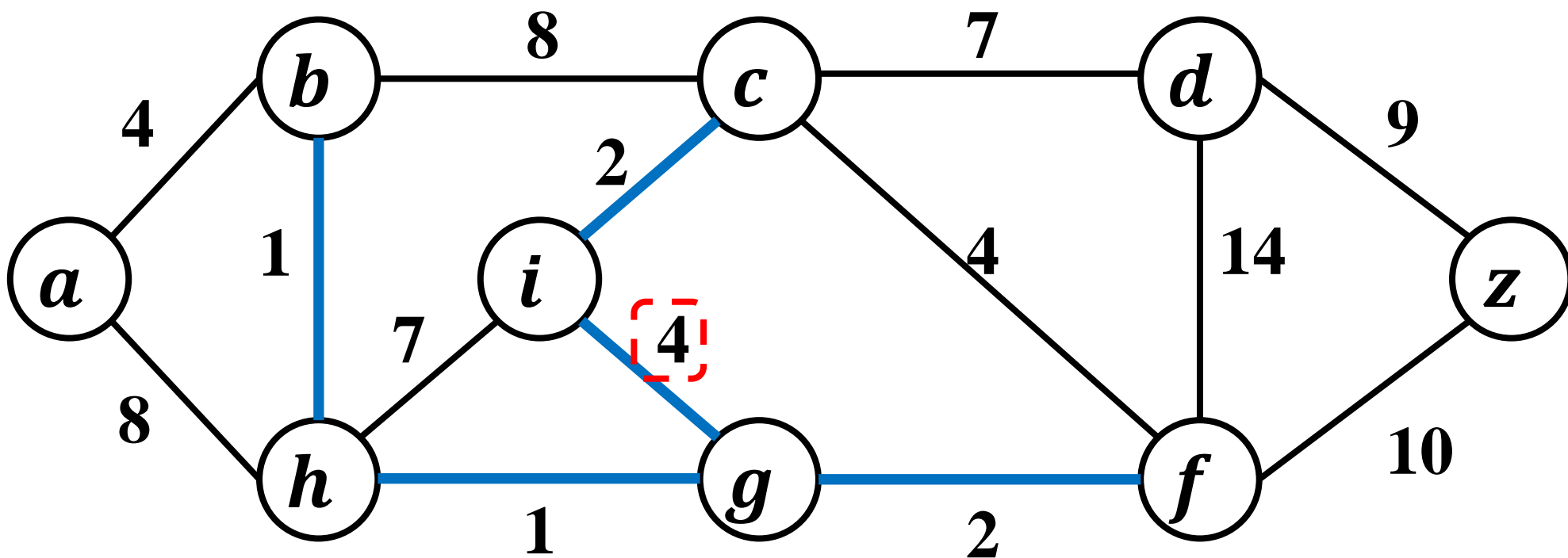
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



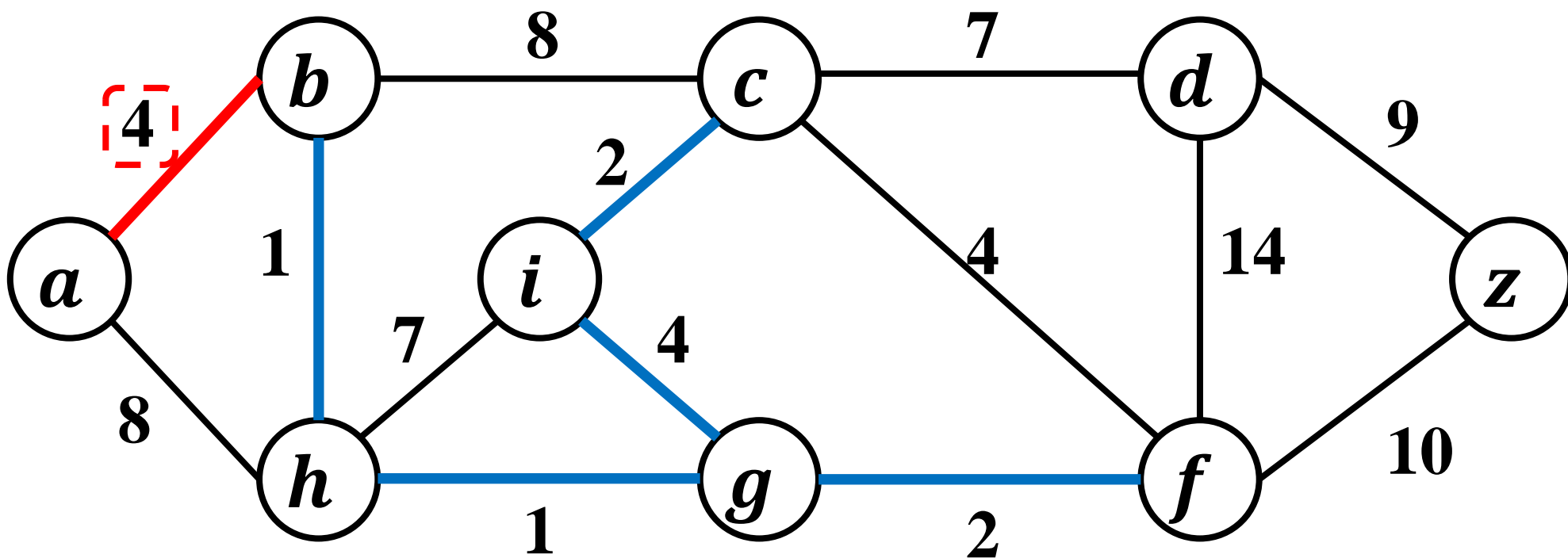
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



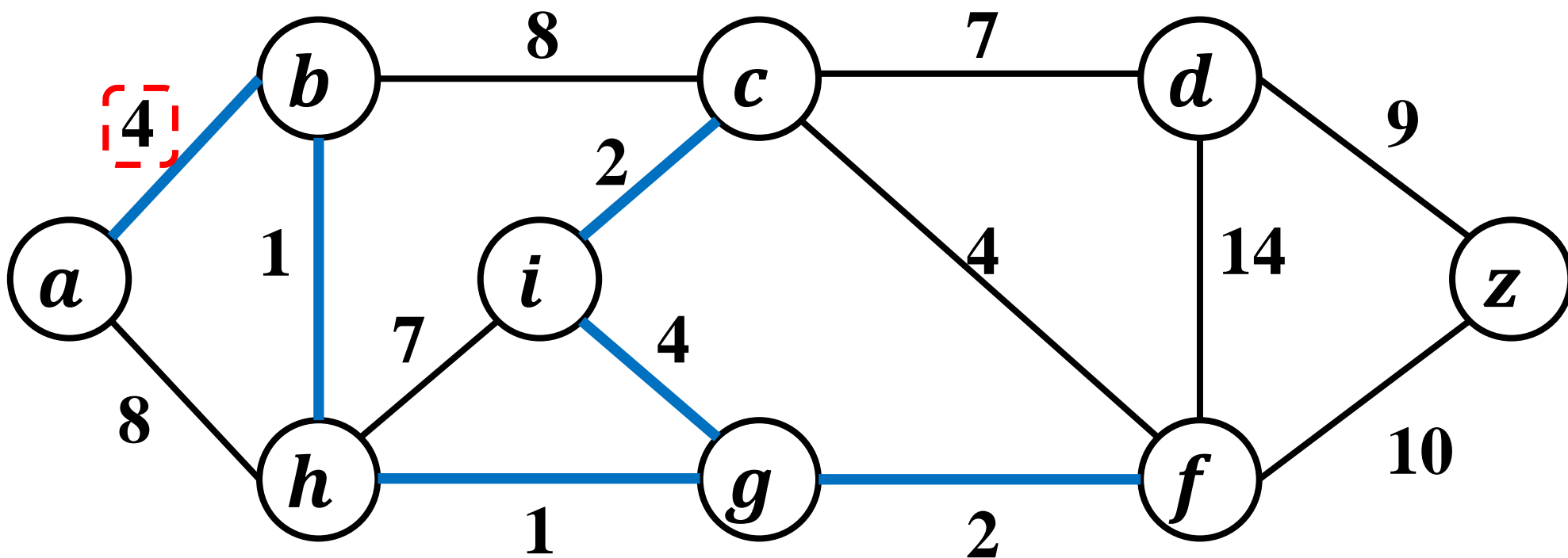
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边

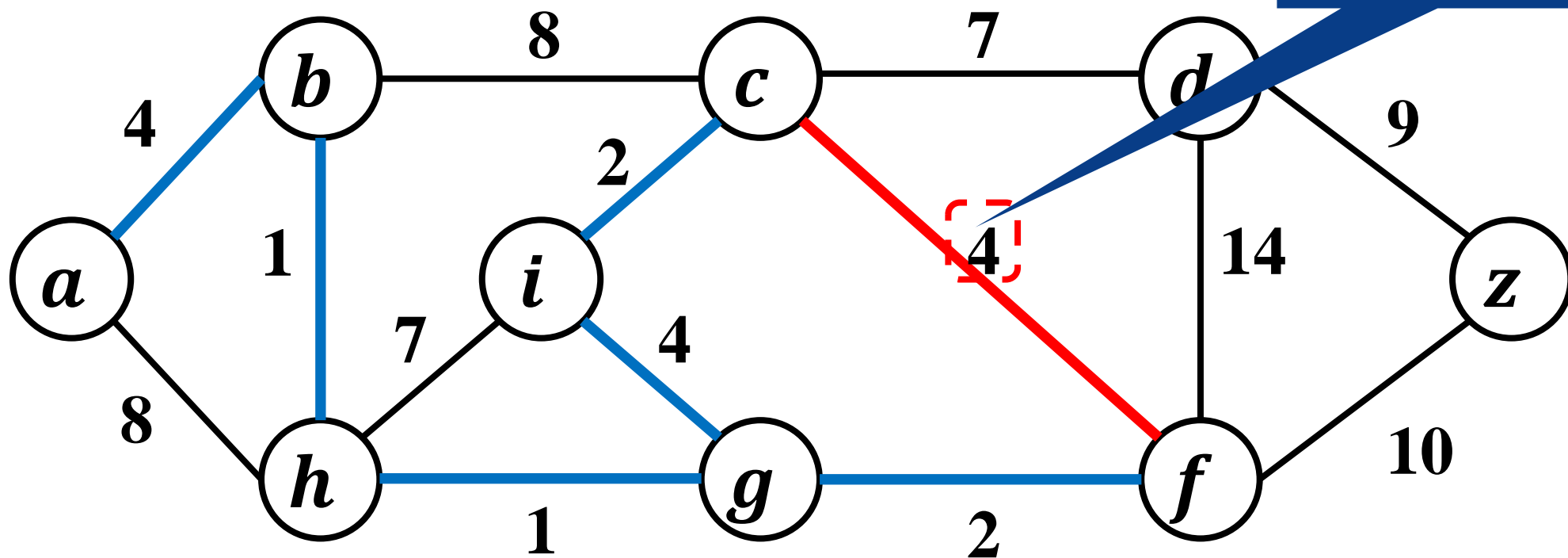


- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边

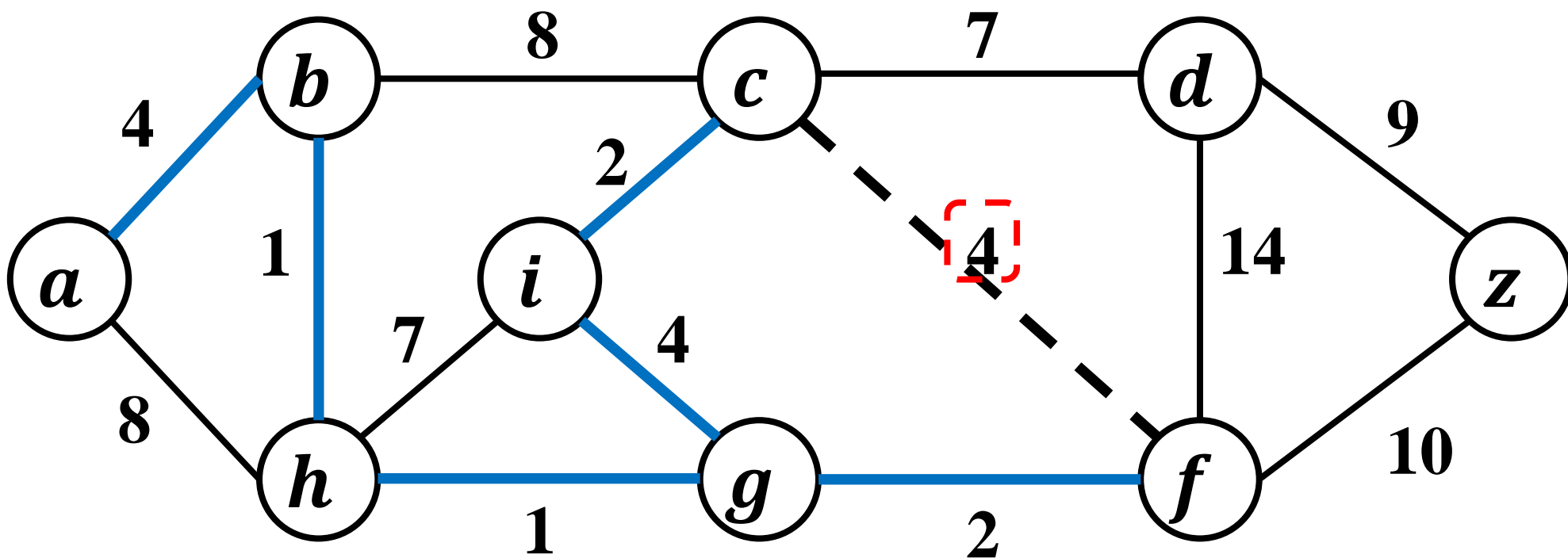




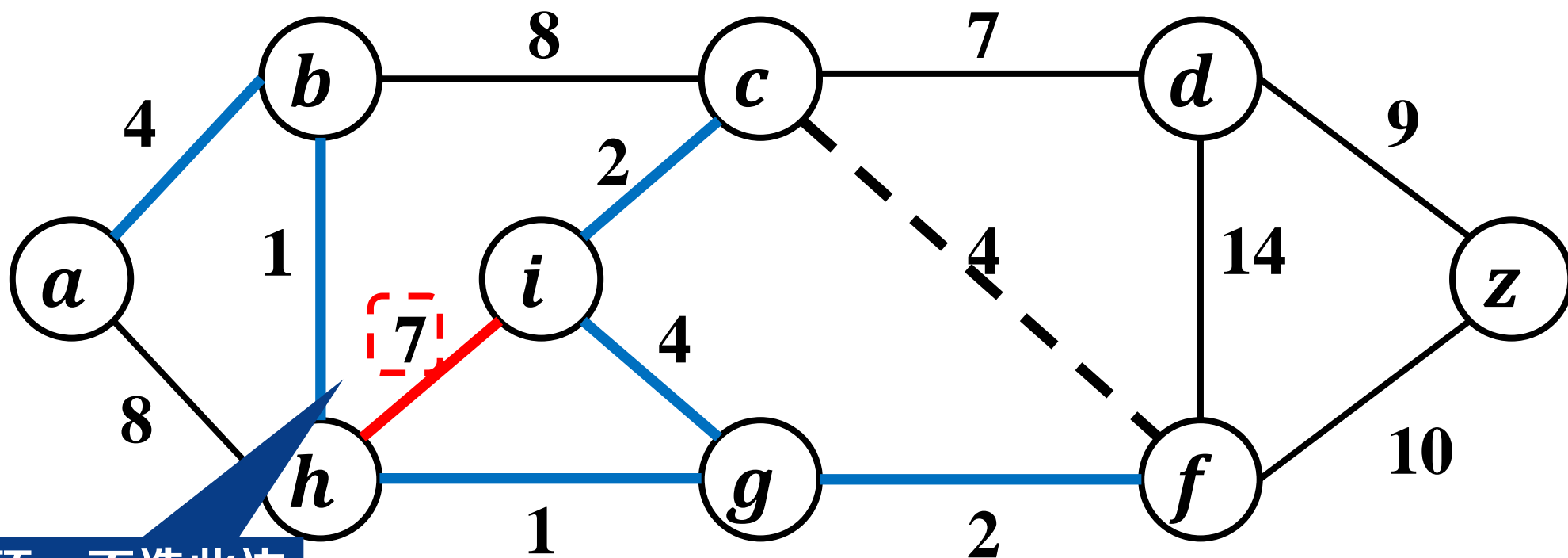
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



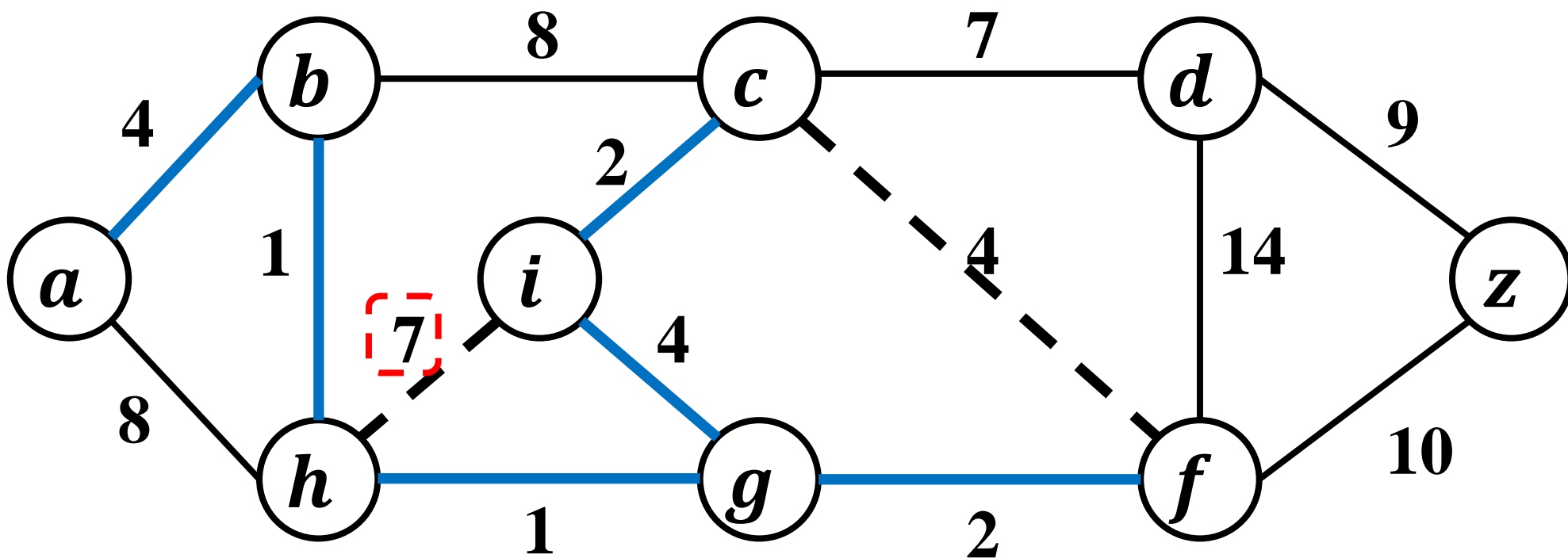
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



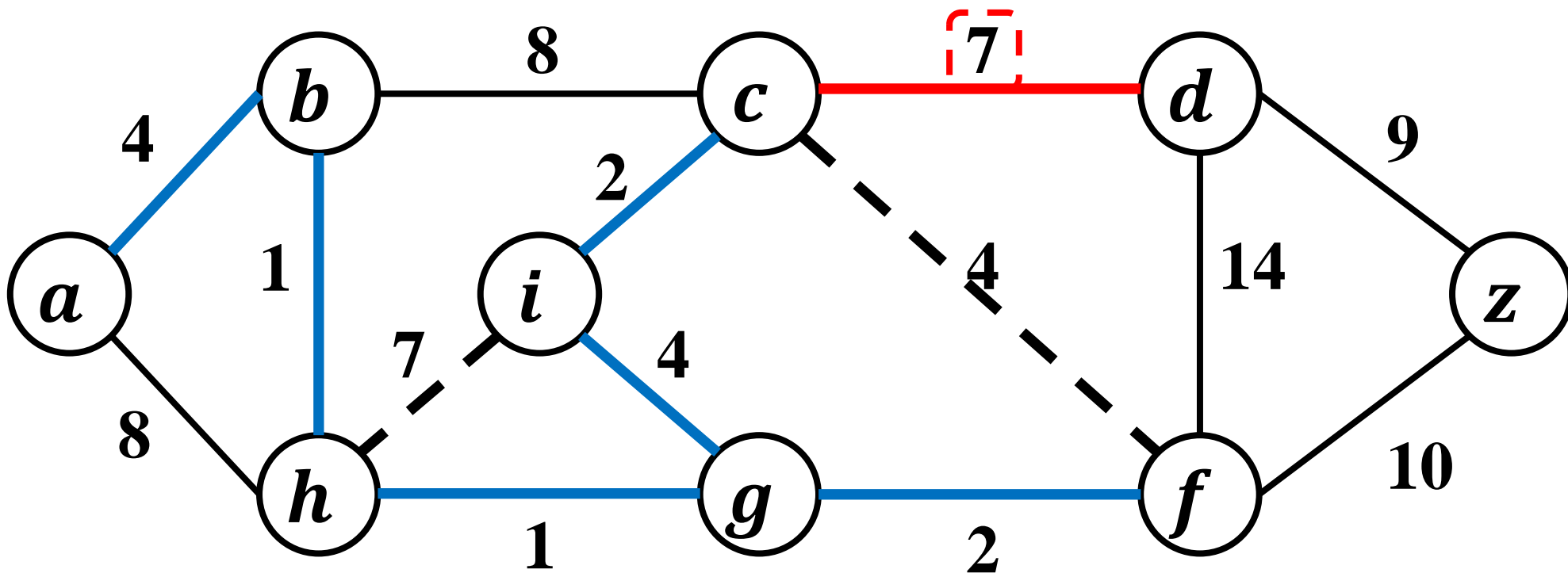
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



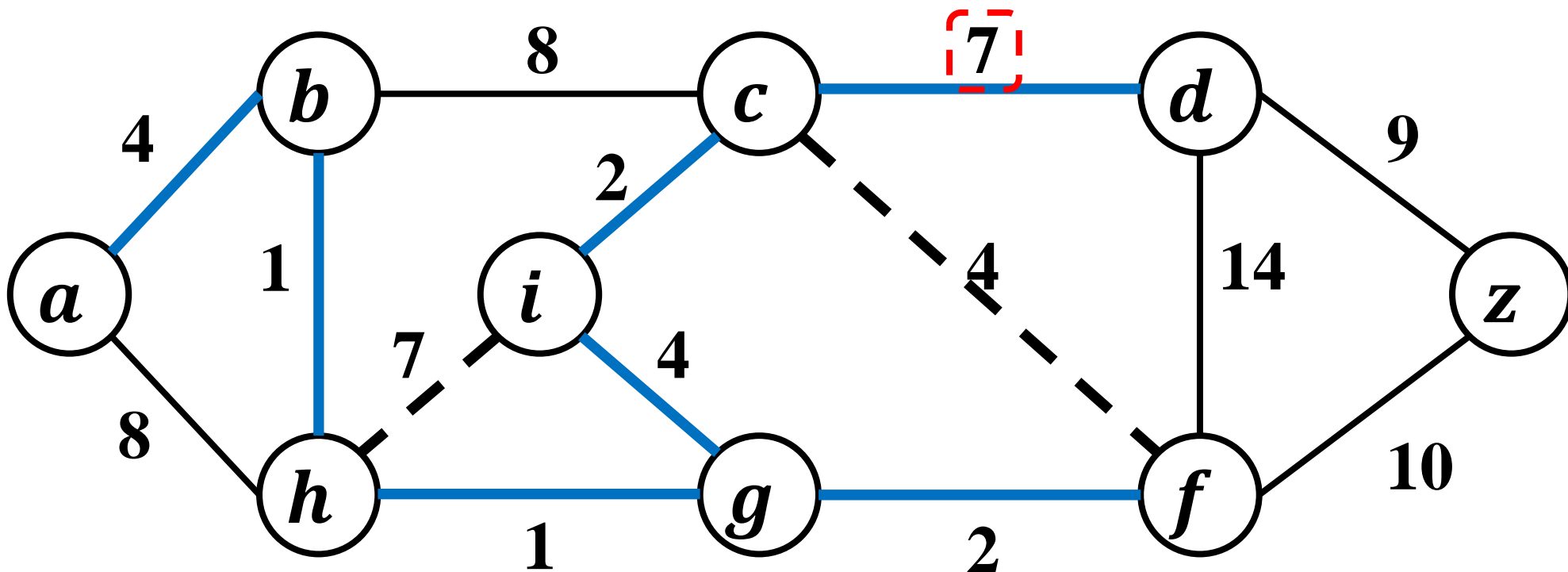
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



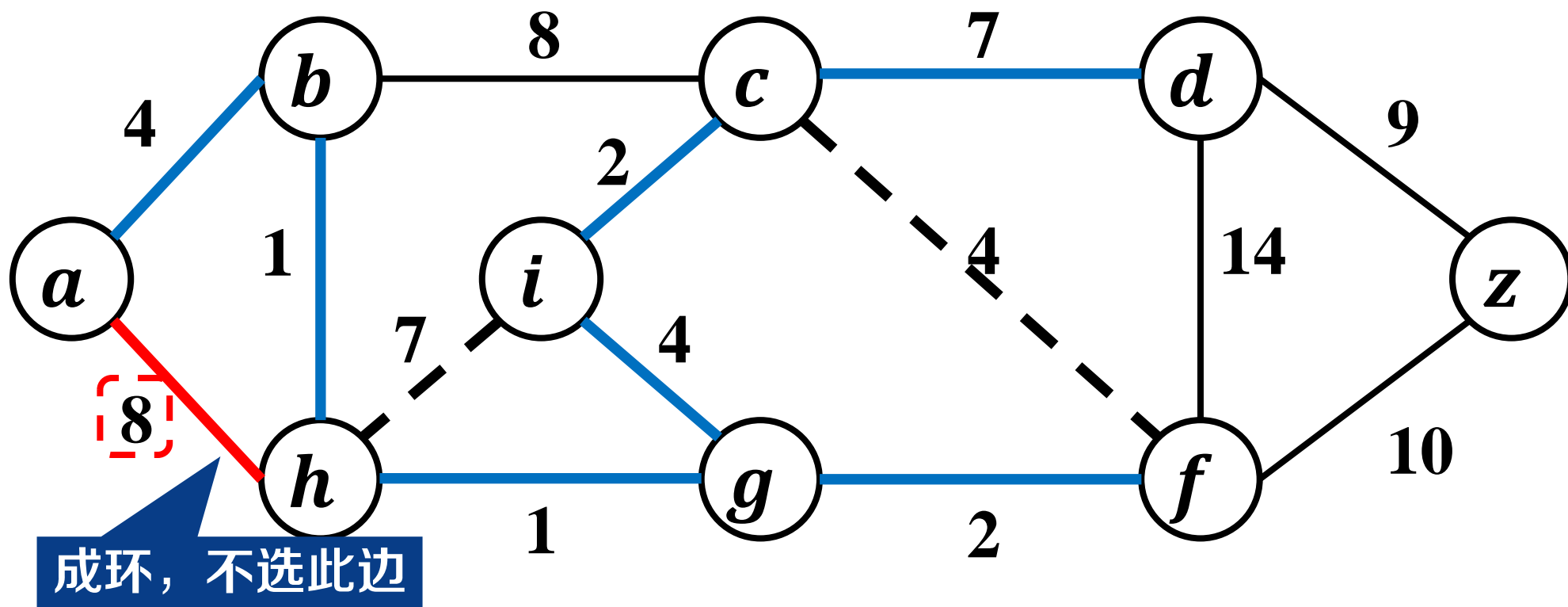
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



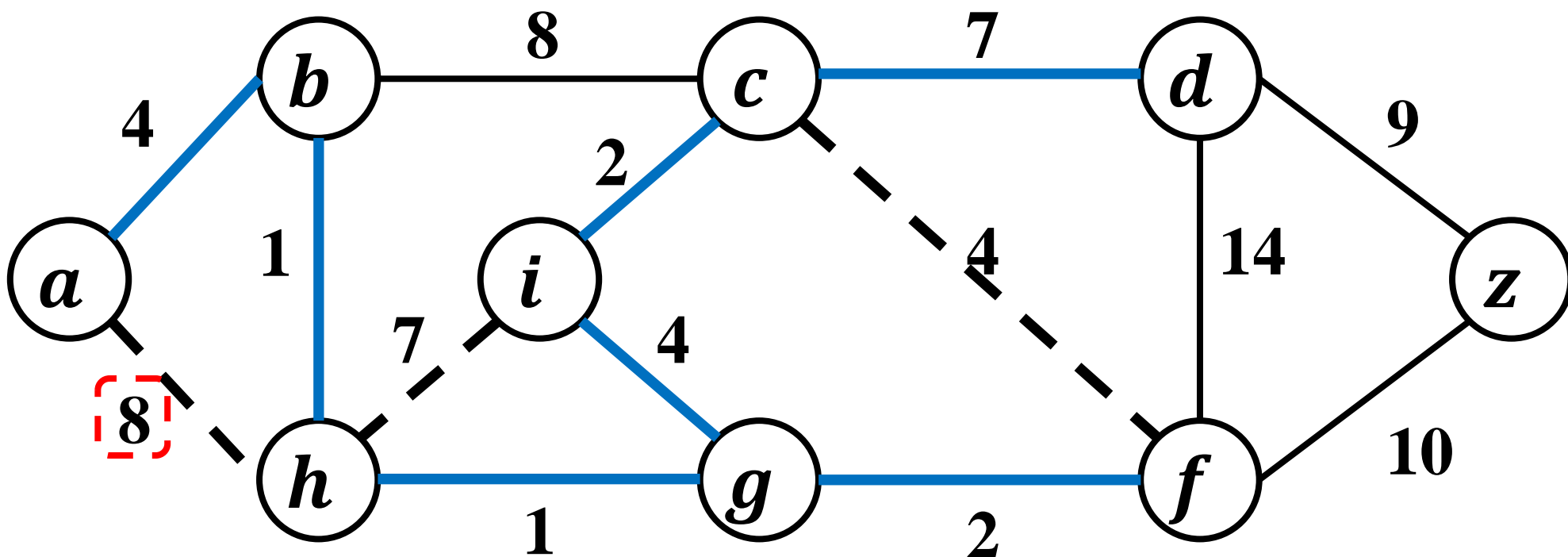
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边

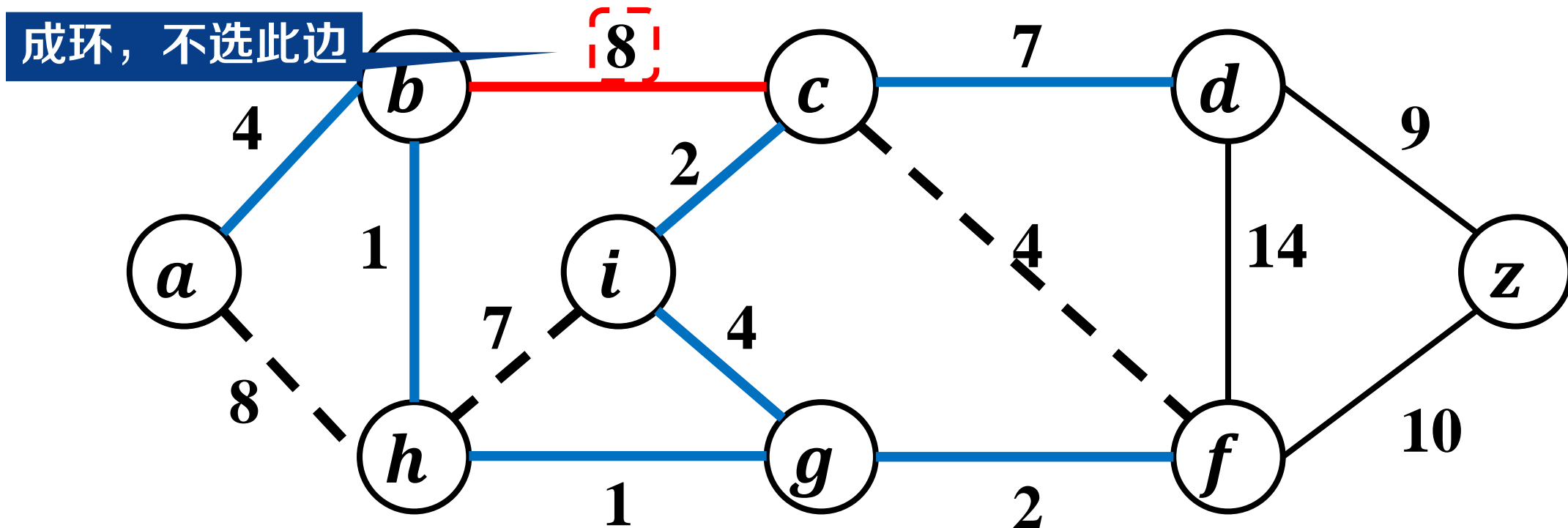


- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边

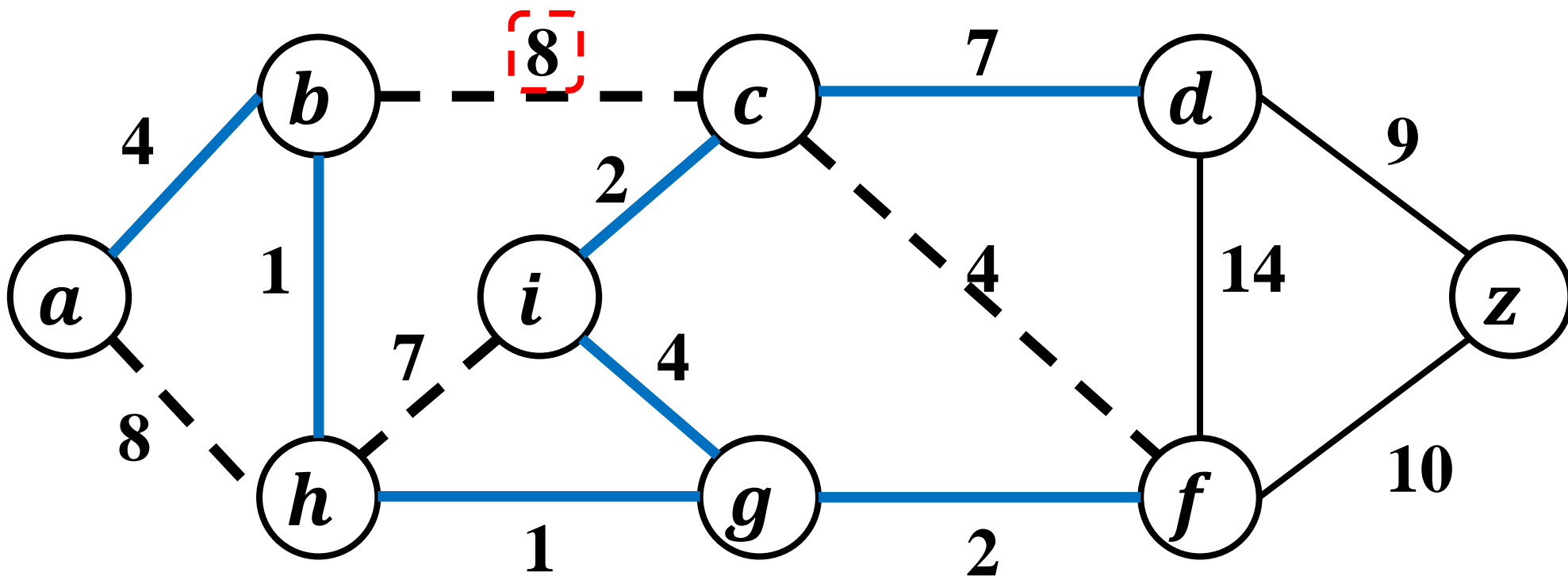




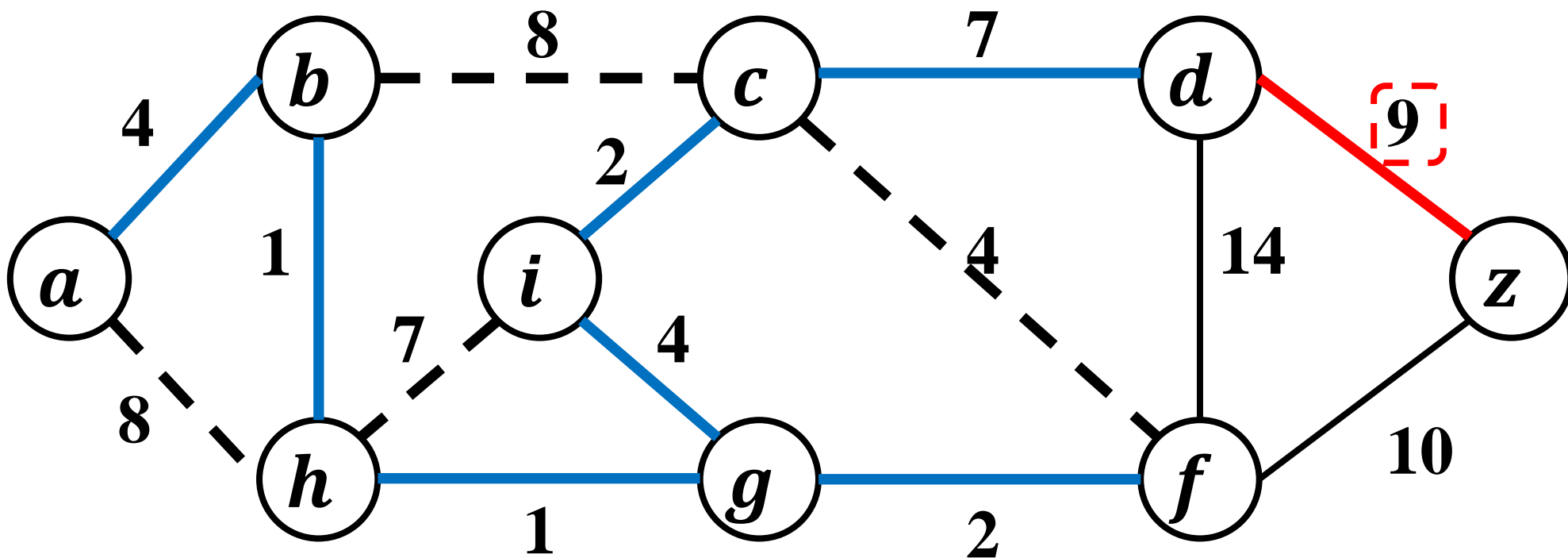
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



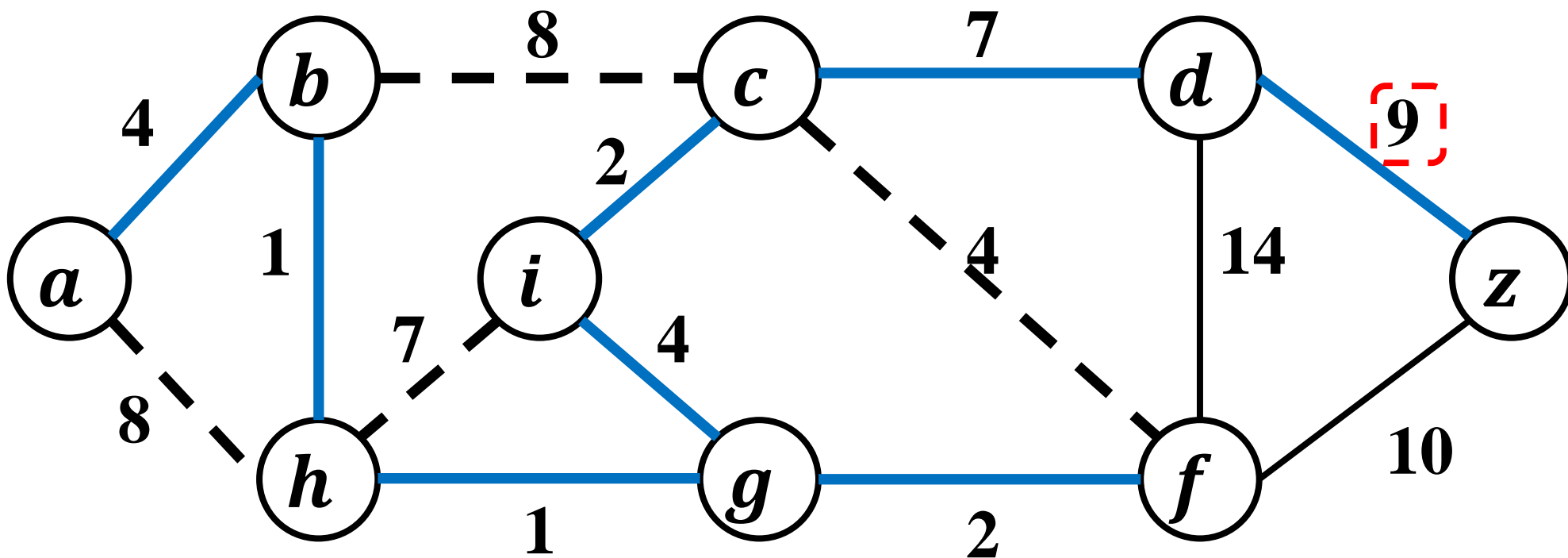
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



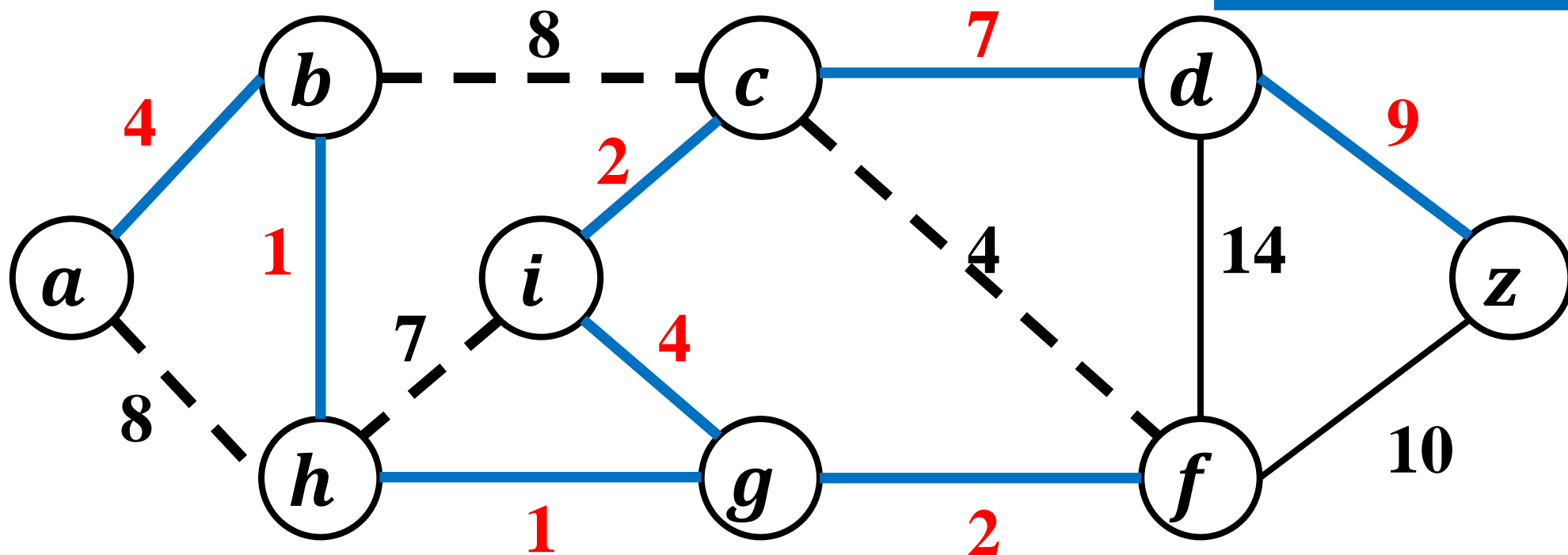
- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



- 算法思想：直接实现通用框架
  - 需保证边集 $A$ 仍是一个无环图
    - 选边时避免成环
  - 需保证边集 $A$ 仍是最小生成树的子集
    - 每次选择当前权重最小边



- Generic-MST( $G$ )

```
 $A \leftarrow \emptyset$   
while 没有形成最小生成树 do  
    | 寻找  $A$  的安全边  $(u, v)$   
    |  $A \leftarrow A \cup (u, v)$   
end  
return  $A$ 
```

不成环的最小边，是一种贪心策略

## • Generic-MST( $G$ )

```
 $A \leftarrow \emptyset$ 
```

```
while 没有形成最小生成树 do
```

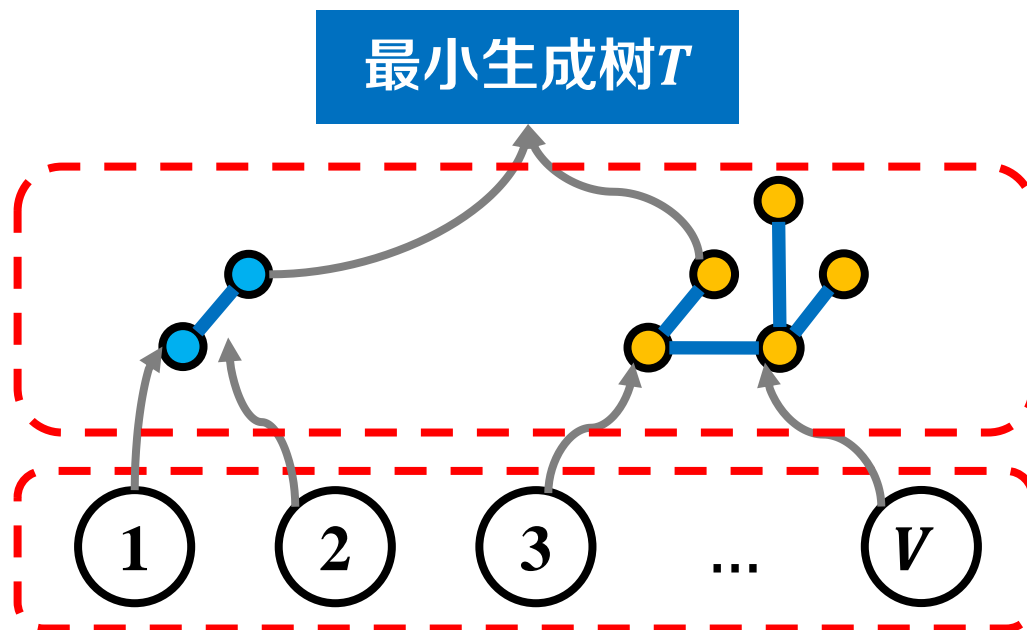
```
    | 寻找  $A$  的安全边  $(u, v)$ 
```

```
    |  $A \leftarrow A \cup (u, v)$ 
```

```
end
```

```
return  $A$ 
```

森林不断合并子树最终形成一棵树  
不成环的最小边，是一种贪心策略



扩大成为原树

选边保持子树

顶点必是子树

## • Generic-MST( $G$ )

```
 $A \leftarrow \emptyset$ 
```

```
while 没有形成最小生成树 do
```

```
    | 寻找 $A$ 的安全边 $(u, v)$ 
```

```
    |  $A \leftarrow A \cup (u, v)$ 
```

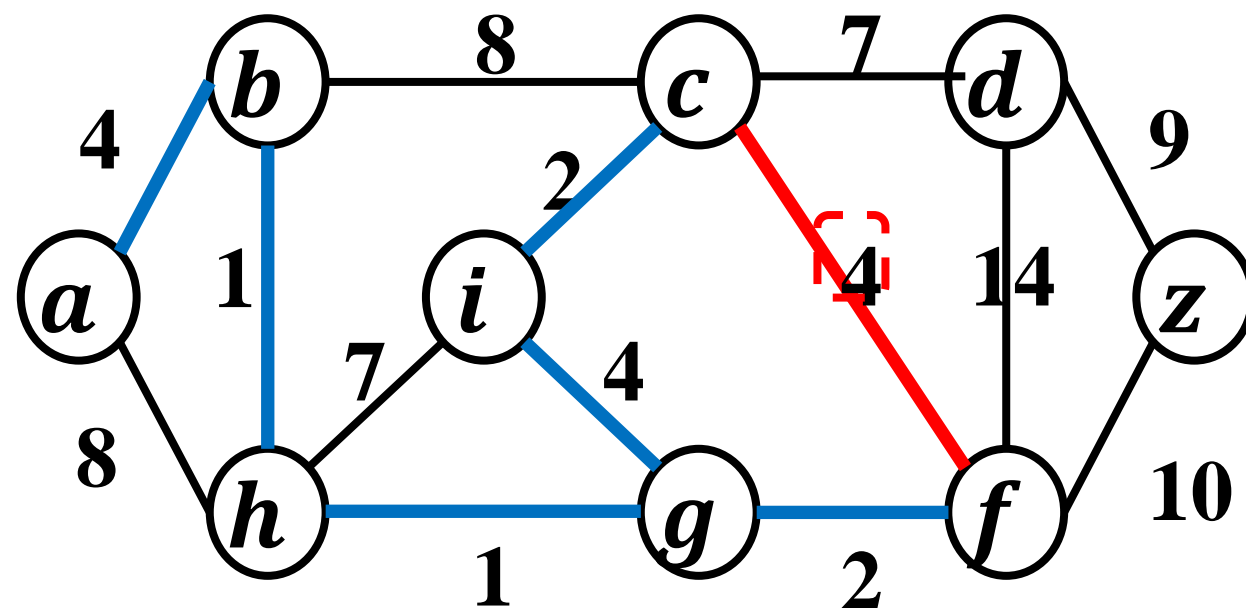
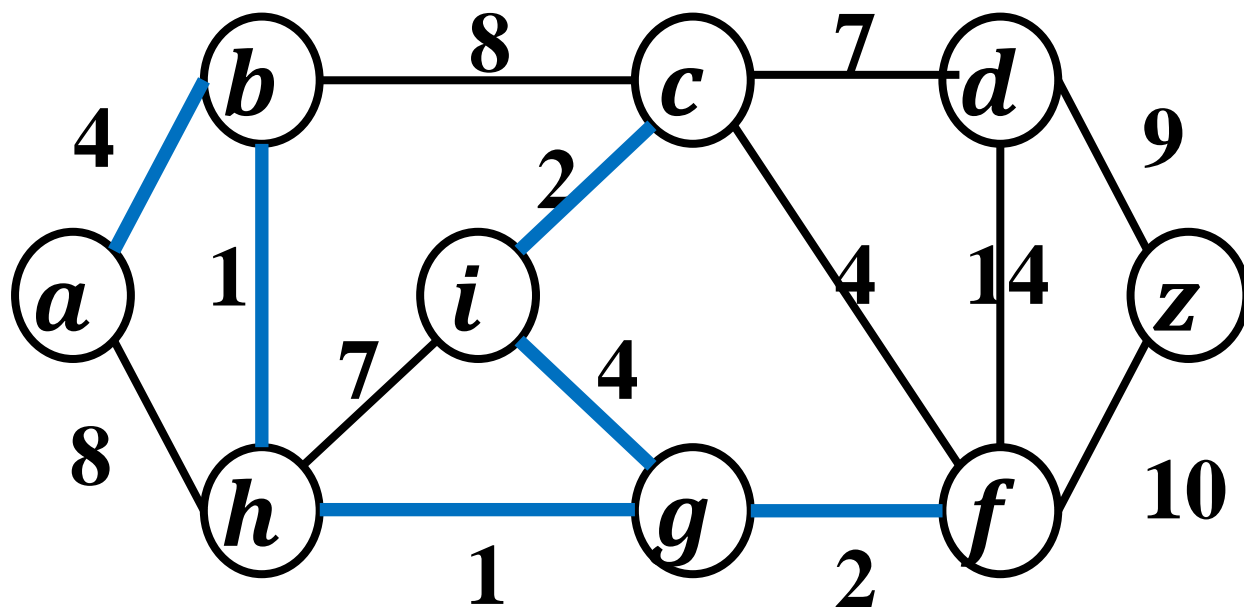
```
end
```

```
return  $A$ 
```

森林不断合并子树最终形成一棵树  
不成环的最小边，是一种贪心策略



判断所选边的顶点是否在一棵子树





## • Generic-MST( $G$ )

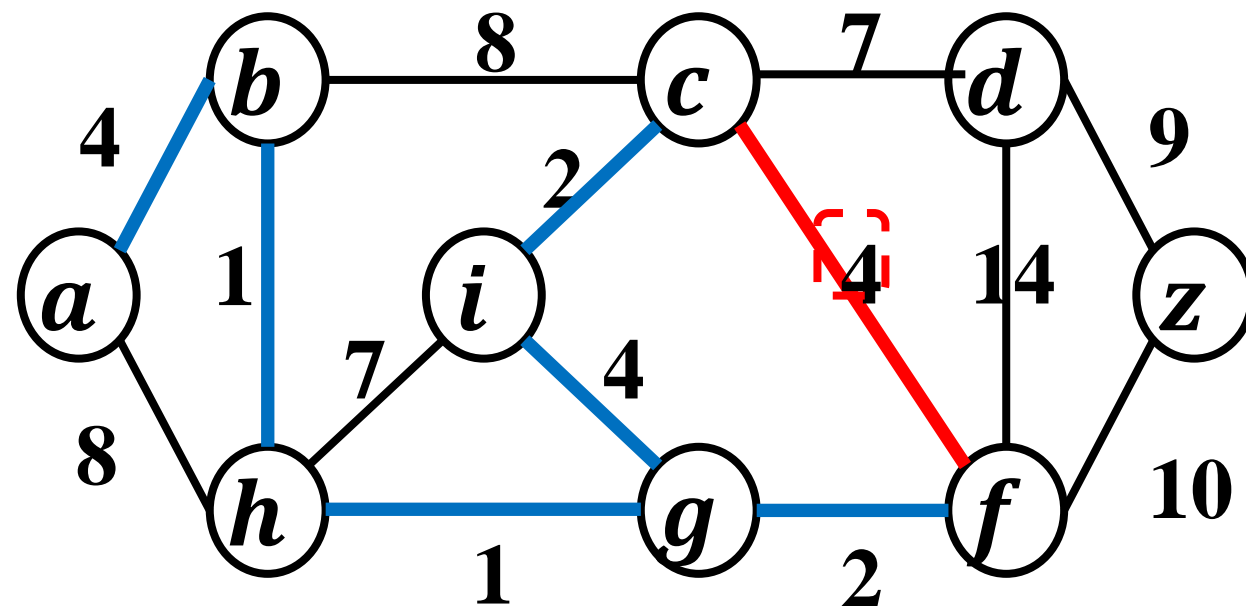
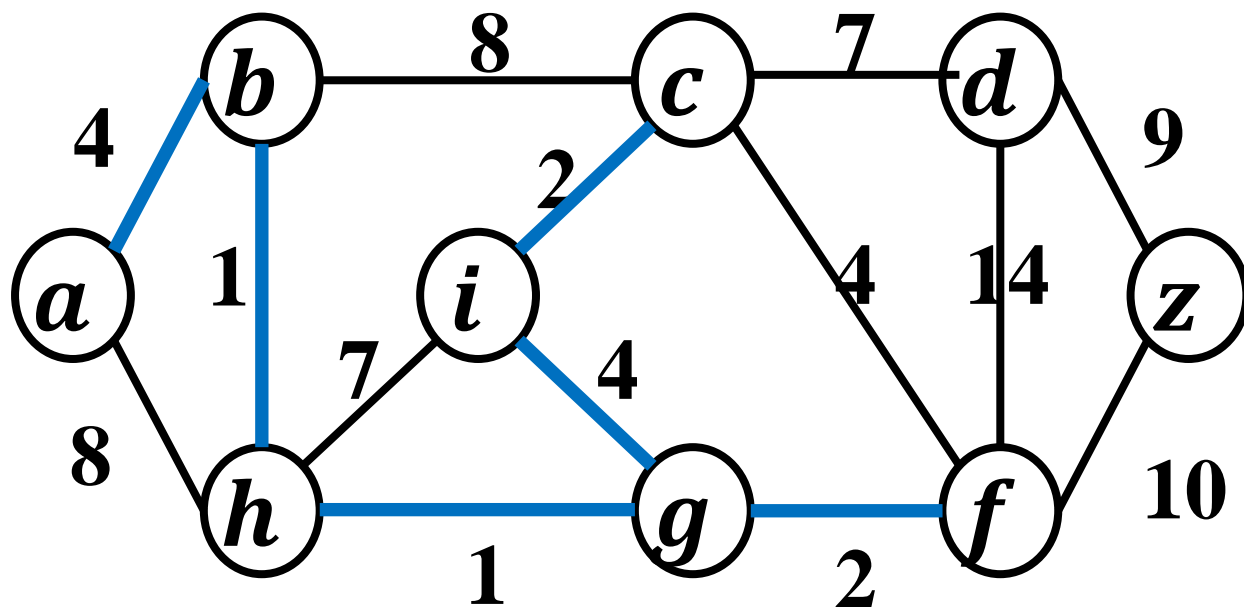
```
A ← ∅  
while 没有形成最小生成树 do  
    | 寻找A的安全边( $u, v$ )!  
    | A ← A ∪ ( $u, v$ )  
end  
return A
```

算法正确性的关键

森林不断合并子树最终形成一棵树  
不成环的最小边，是一种贪心策略



判断所选边的顶点是否在一棵子树



问题的回顾

算法与实例

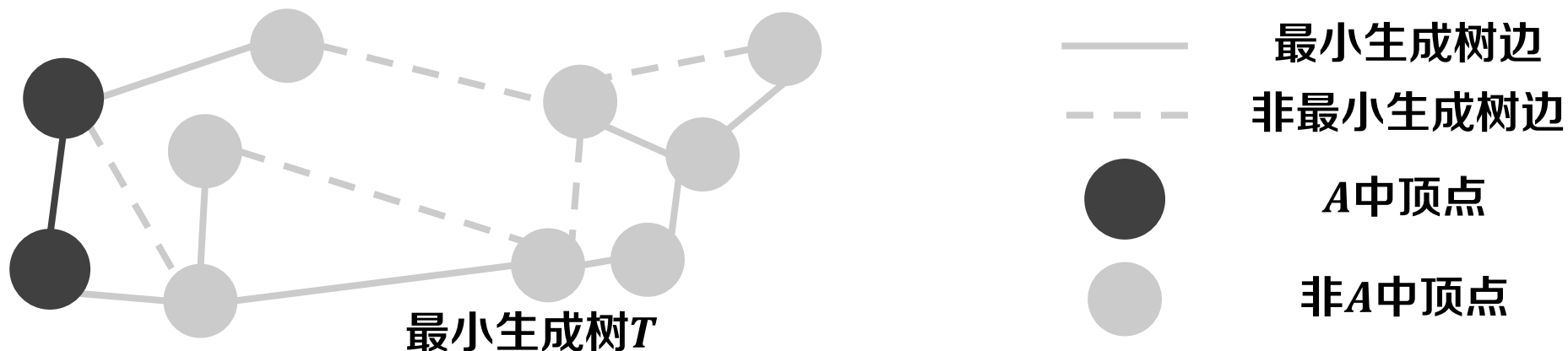
正确性证明

不相交集合

复杂度分析

- 安全边(Safe Edge)

- $A$ 是某棵最小生成树 $T$ 边的子集,  $A \subseteq T$
- $A \cup \{(u, v)\}$ 仍是 $T$ 边的一个子集, 则称 $(u, v)$ 是 $A$ 的**安全边**



若每次向边集 $A$ 中新增**安全边**, 可保证边集 $A$ 是最小生成树的子集

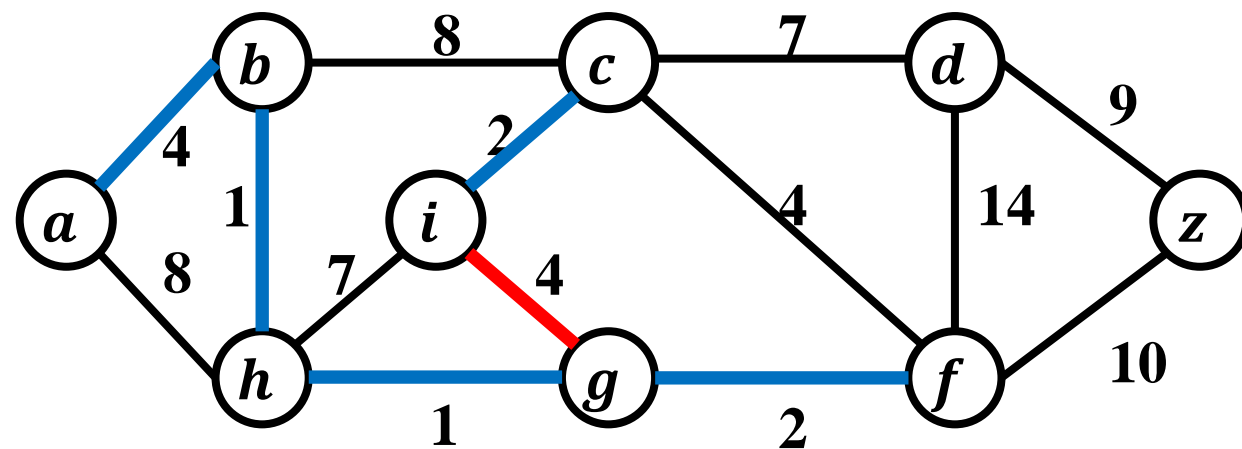
问题: Kruskal算法选边策略能否保证每次都选择了安全边?

- **Kruskal算法选边策略能否保证每次都选择了安全边？——能！**

# 正确性证明



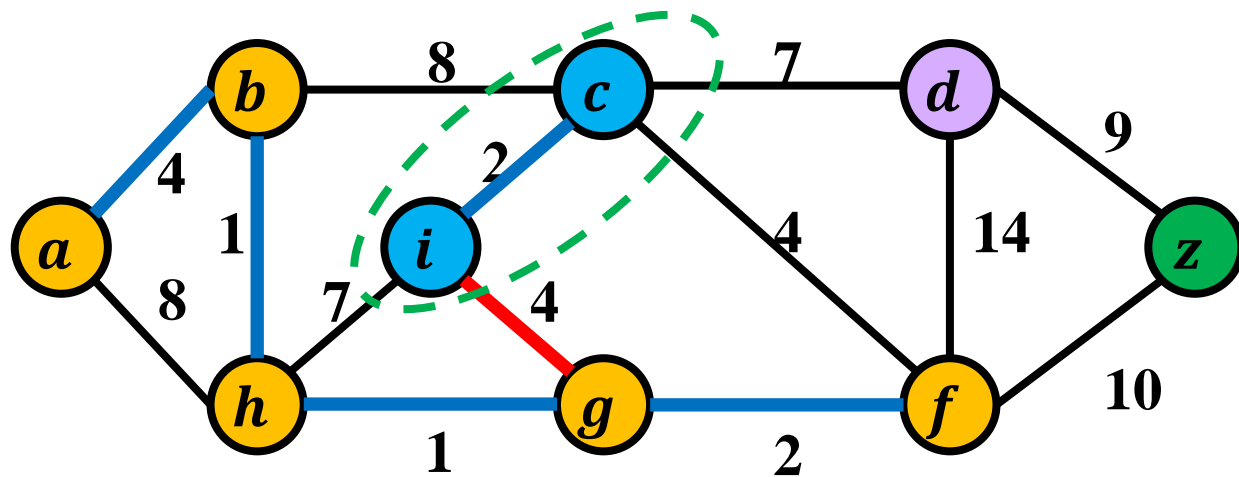
- Kruskal算法选边策略能否每次都选择了安全边? ——能!
- 证明
  - 不妨设当前已选边集为 $A$ , 下一条选择的边是 $(i, g)$



# 正确性证明



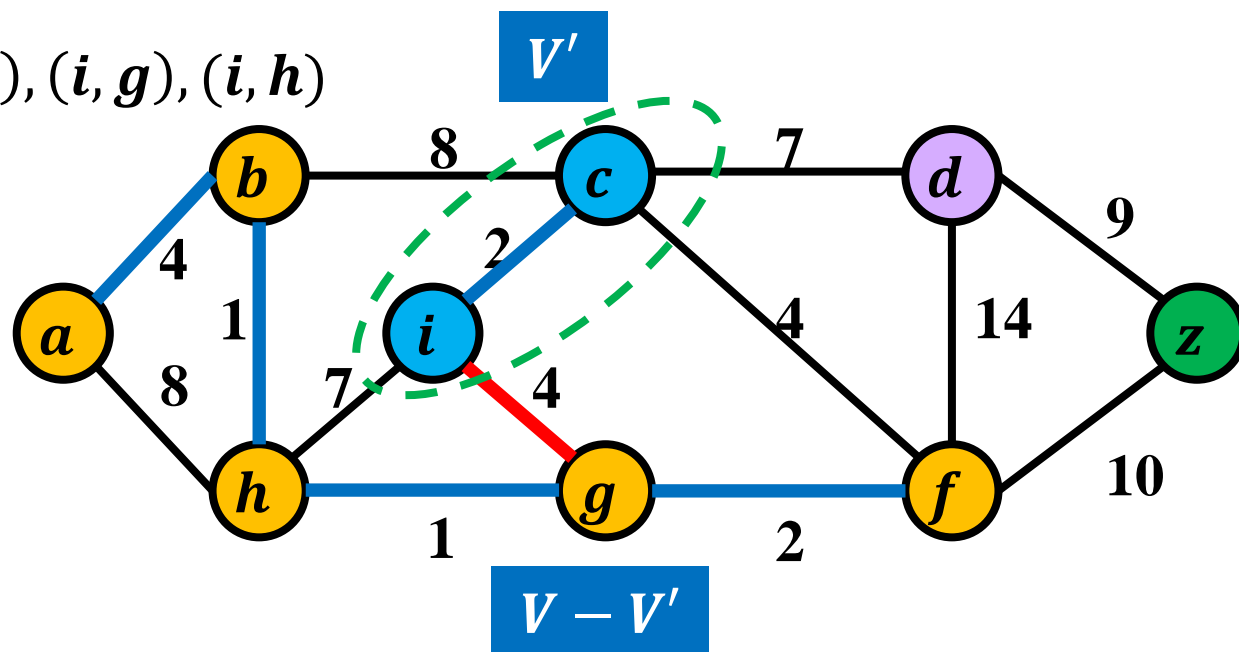
- Kruskal算法选边策略能否每次都选择了安全边? ——能!
- 证明
  - 不妨设当前已选边集为 $A$ , 下一条选择的边是 $(i, g)$
  - 已选边集 $A$ 把图分成为若干棵子树, 其中 $(V', E')$ 是包含顶点 $i$ 的子树



# 正确性证明



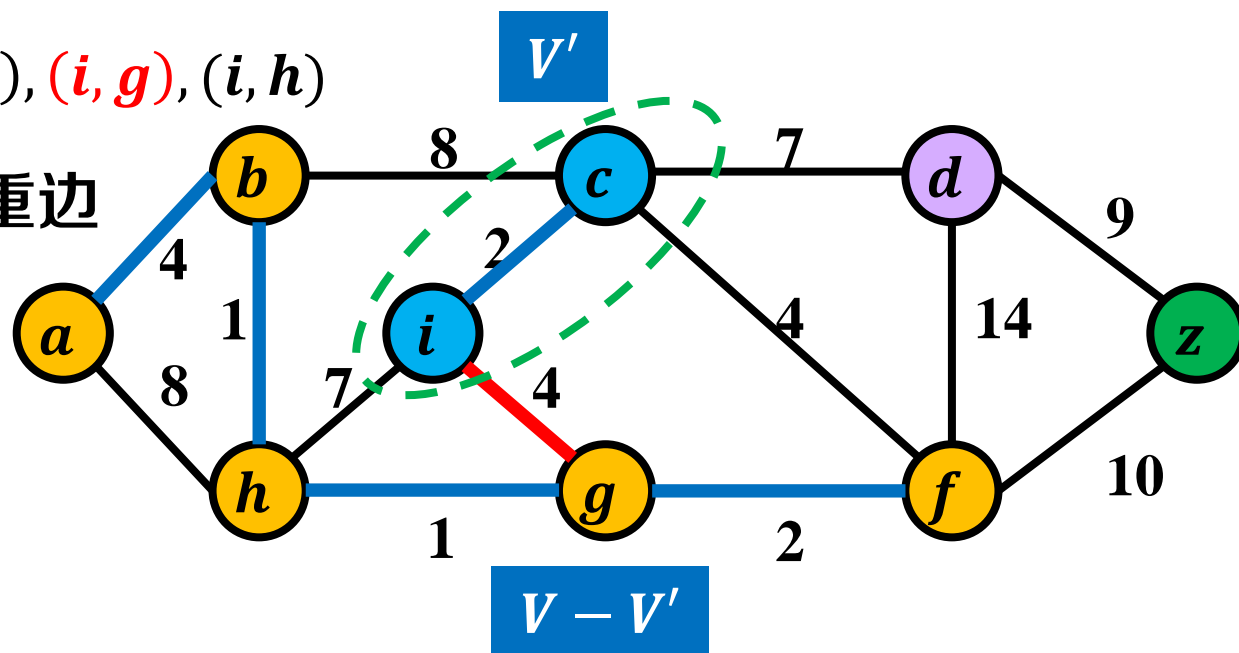
- Kruskal算法选边策略能否每次都选择了安全边？——能！
- 证明
  - 不妨设当前已选边集为 $A$ ，下一条选择的边是 $(i, g)$
  - 已选边集 $A$ 把图分成为若干棵子树，其中 $(V', E')$ 是包含顶点 $i$ 的子树
  - 构造割 $(V', V - V')$ ，割不妨害边集 $A$ ，换言之 $A$ 中的边不会横跨 $(V', V - V')$ 
    - 边集 $A$ 不包含横跨边 $(b, c), (c, d), (c, f), (i, g), (i, h)$



# 正确性证明



- Kruskal算法选边策略能否每次都选择了安全边？——能！
- 证明
  - 不妨设当前已选边集为 $A$ ，下一条选择的边是 $(i, g)$
  - 已选边集 $A$ 把图分成为若干棵子树，其中 $(V', E')$ 是包含顶点 $i$ 的子树
  - 构造割 $(V', V - V')$ ，割不妨害边集 $A$ ，换言之 $A$ 中的边不会横跨 $(V', V - V')$ 
    - 边集 $A$ 不包含横跨边 $(b, c), (c, d), (c, f), (i, g), (i, h)$
  - $(i, g)$ 是横跨割 $(V', V - V')$ 的最小权重边

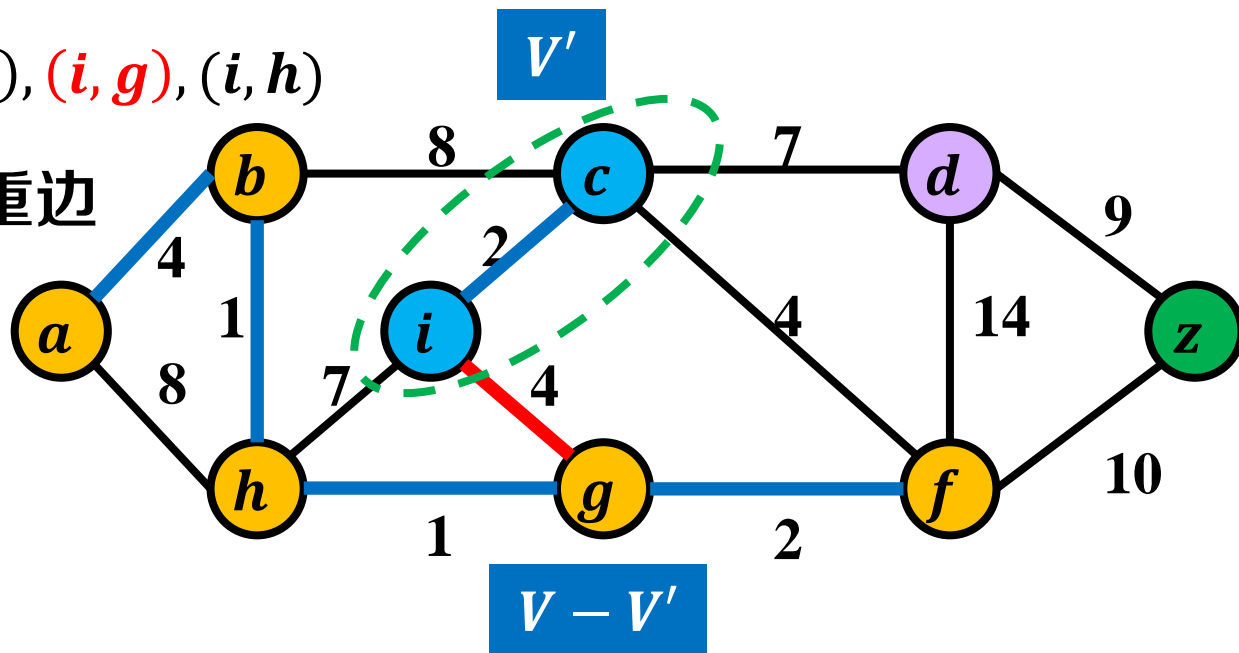




# 正确性证明



- Kruskal算法选边策略能否每次都选择了安全边? ——能!
- 证明
  - 不妨设当前已选边集为 $A$ , 下一条选择的边是 $(i, g)$
  - 已选边集 $A$ 把图分成为若干棵子树, 其中 $(V', E')$ 是包含顶点 $i$ 的子树
  - 构造割 $(V', V - V')$ , 割不妨害边集 $A$ , 换言之 $A$ 中的边不会横跨 $(V', V - V')$ 
    - 边集 $A$ 不包含横跨边 $(b, c), (c, d), (c, f), (i, g), (i, h)$
  - $(i, g)$ 是横跨割 $(V', V - V')$ 的最小权重边
  - $(i, g)$ 是关于割 $(V', V - V')$ 轻边
  - 由于割 $(V', V - V')$ 不妨害边集 $A$
  - 轻边 $(i, g)$ 为安全边



- MST-Kruskal( $G$ )

输入: 图  $G$

输出: 最小生成树

把边按照权重升序排序

按权重排序

$T \leftarrow \{\}$

for  $(u, v) \in E$  do

    if  $u, v$  不在同一子树 then

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

        合并  $u, v$  所在子树

    end

end

return  $T$

- MST-Kruskal( $G$ )

输入: 图  $G$

输出: 最小生成树

把边按照权重升序排序

$T \leftarrow \{\}$

**for**  $(u, v) \in E$  **do**

**if**  $u, v$  不在同一子树 **then**

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

        合并  $u, v$  所在子树

**end**

**end**

**return**  $T$

按照权重从小到大考察边

- MST-Kruskal( $G$ )

输入: 图  $G$

输出: 最小生成树

把边按照权重升序排序

$T \leftarrow \{\}$

for  $(u, v) \in E$  do

    if  $u, v$  不在同一子树 then

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

        合并  $u, v$  所在子树

    end

end

return  $T$

加入该边不成环

- MST-Kruskal( $G$ )

输入: 图  $G$

输出: 最小生成树

把边按照权重升序排序

$T \leftarrow \{\}$

for  $(u, v) \in E$  do

    if  $u, v$  不在同一子树 then

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

        合并  $u, v$  所在子树

    end

end

return  $T$

加入该边，更新连通状态

- MST-Kruskal( $G$ )

输入: 图  $G$

输出: 最小生成树

把边按照权重升序排序

$T \leftarrow \{\}$

for  $(u, v) \in E$  do

    if  $u, v$  不在同一子树 then

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

        合并  $u, v$  所在子树

    end

end

return  $T$

问题: 如何高效判定和维护所选边的顶点是否在一棵子树?

问题的回顾

算法与实例

正确性证明

不相交集合

复杂度分析

- MST-Kruskal( $G$ )

输入: 图  $G$

输出: 最小生成树

把边按照权重升序排序

$T \leftarrow \{\}$

for  $(u, v) \in E$  do

    if  $u, v$  不在同一子树 then

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

        合并  $u, v$  所在子树

    end

end

return  $T$

问题: 如何高效判定和维护所选边的顶点是否在一棵子树?



- MST-Kruskal( $G$ )

输入: 图  $G$

输出: 最小生成树

把边按照权重升序排序

$T \leftarrow \{\}$

for  $(u, v) \in E$  do

    if  $u, v$  不在同一子树 then

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

        合并  $u, v$  所在子树

    end

end

return  $T$



需要高效查找顶点所属子树



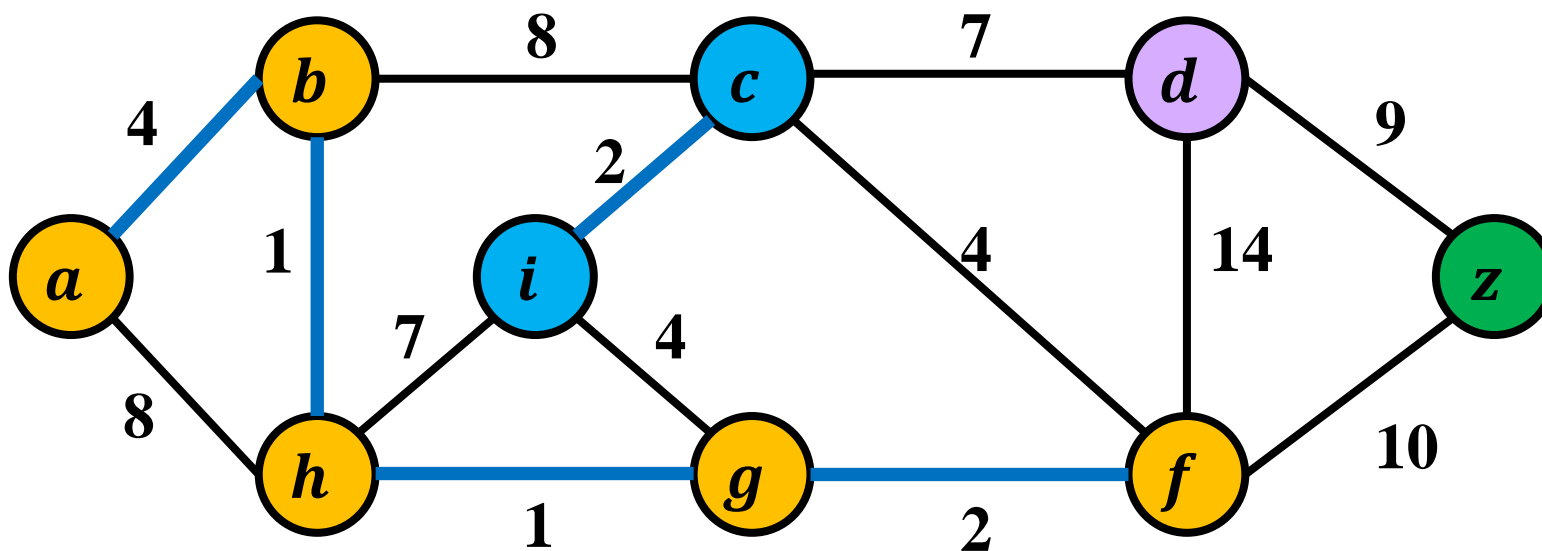
需要高效合并顶点所在子树

同时高效完成两类操作需借助数据结构: 不相交集合

# 不相交集合



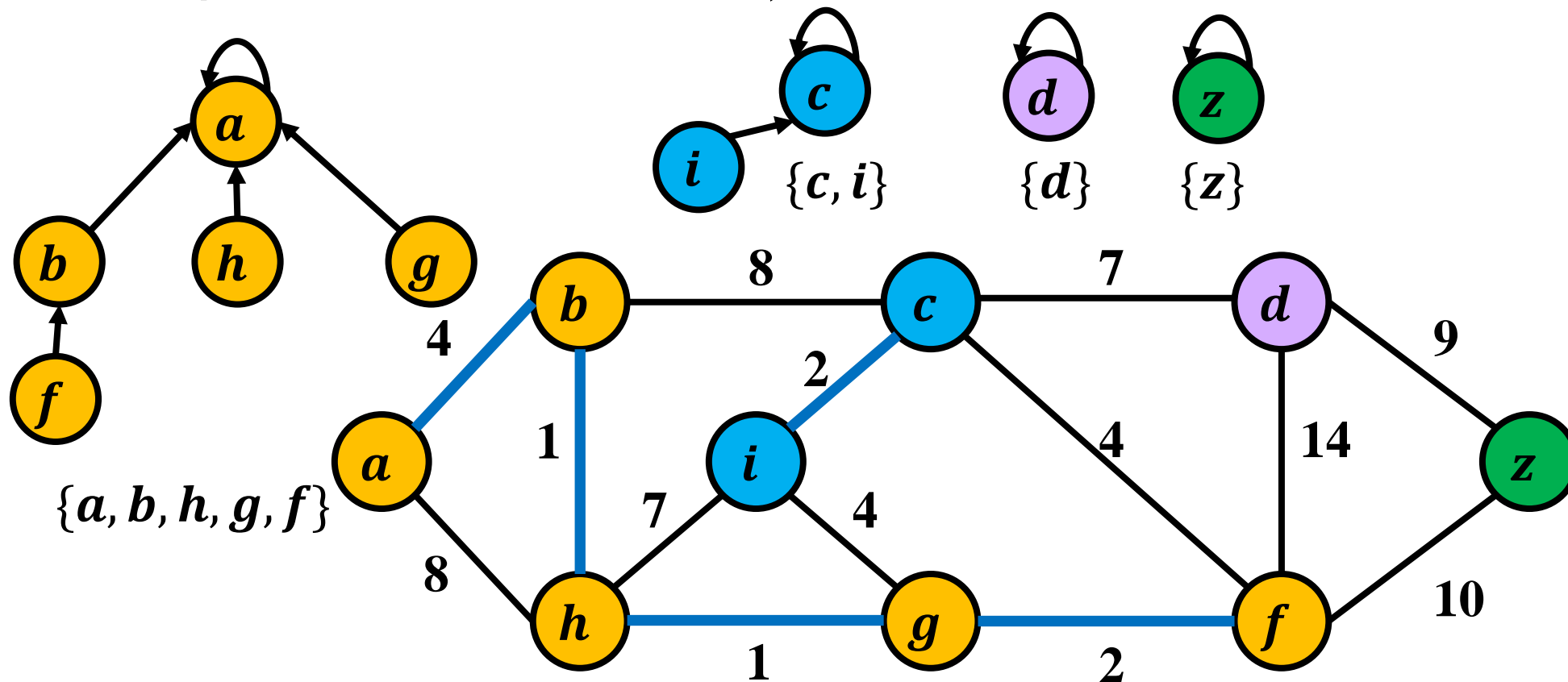
- 把每棵生成子树看作一个顶点集合



# 不相交集合



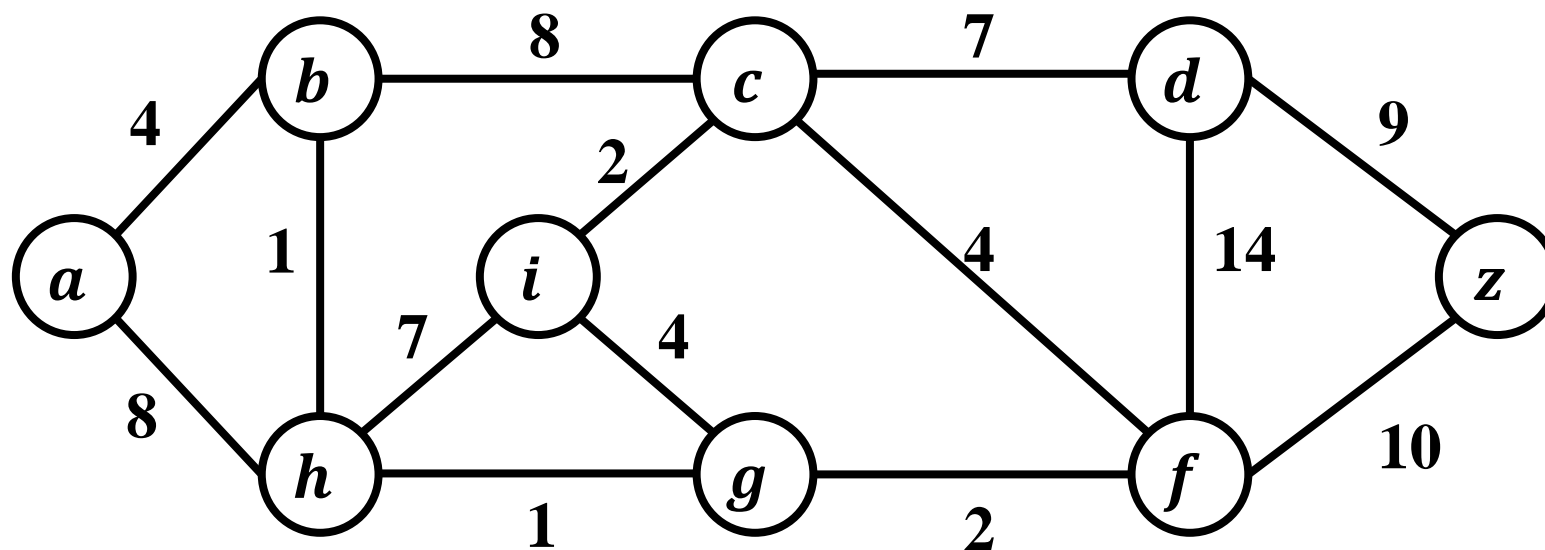
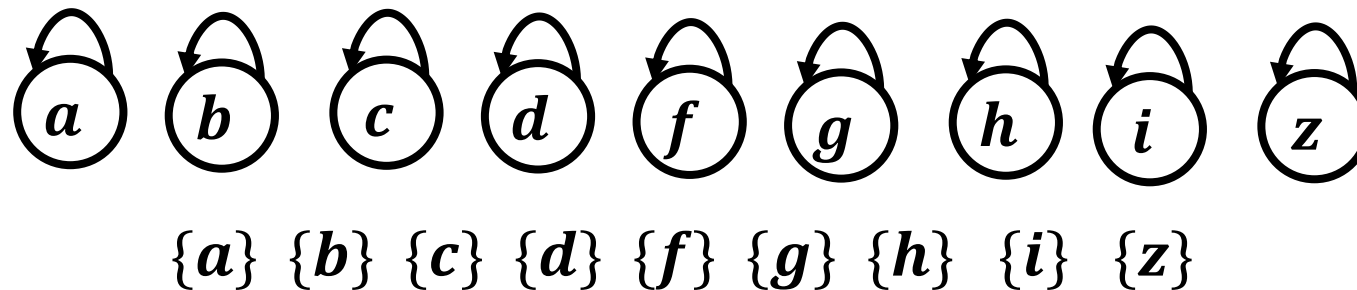
- 把每棵生成子树看作一个顶点集合
  - 每个集合表示为一棵有向树，多个不相交集合构成不相交集合森林
  - 集合元素表示为树结点
  - 树边由子结点指向父结点，根结点有一条指向自身的边



# 不相交集合



- 初始化集合：创建根结点，并设置一条指向自身的边



# 不相交集合：伪代码



- Create-Set( $x$ )

输入: 顶点  $x$

输出: 并查集

$x.parent \leftarrow x$

return  $x$

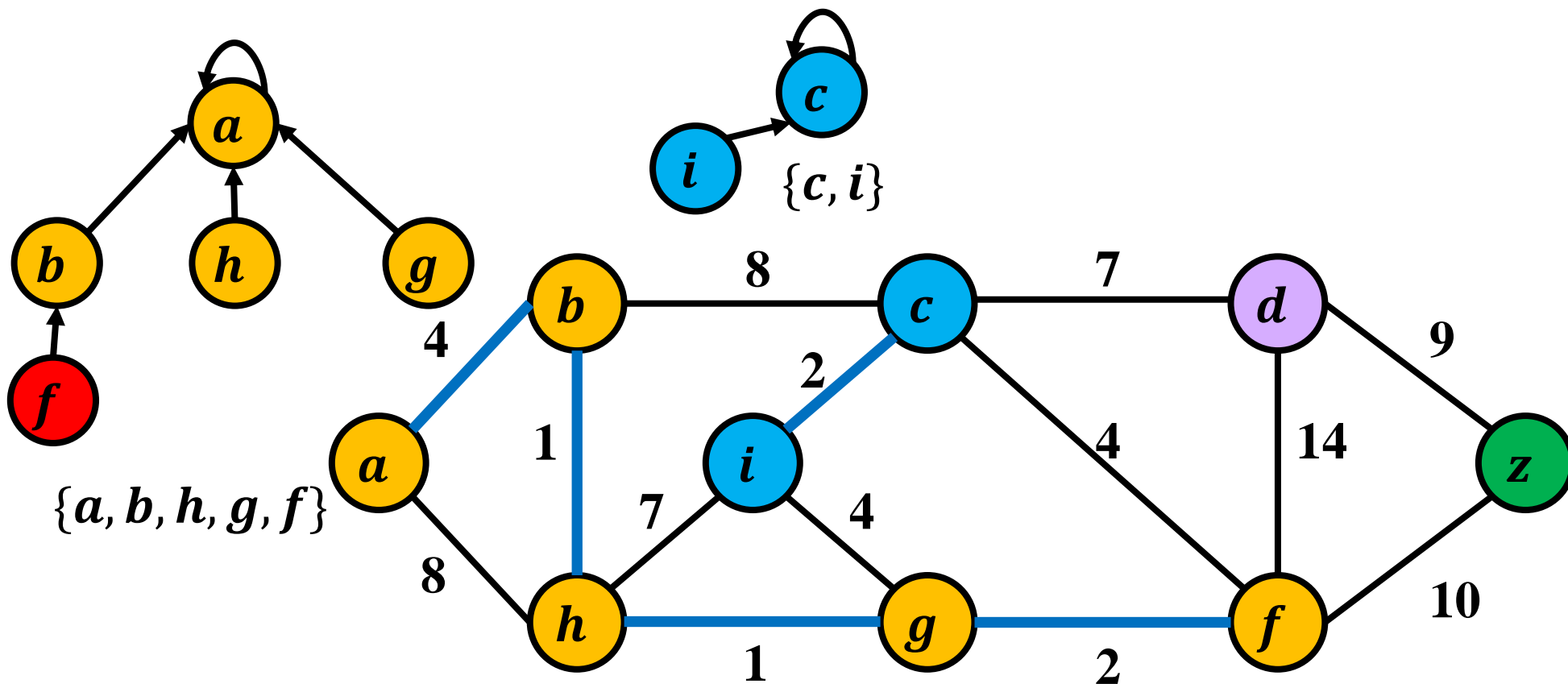
自身为树根

# 不相交集合



- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同

查找 $f$ 的树根：

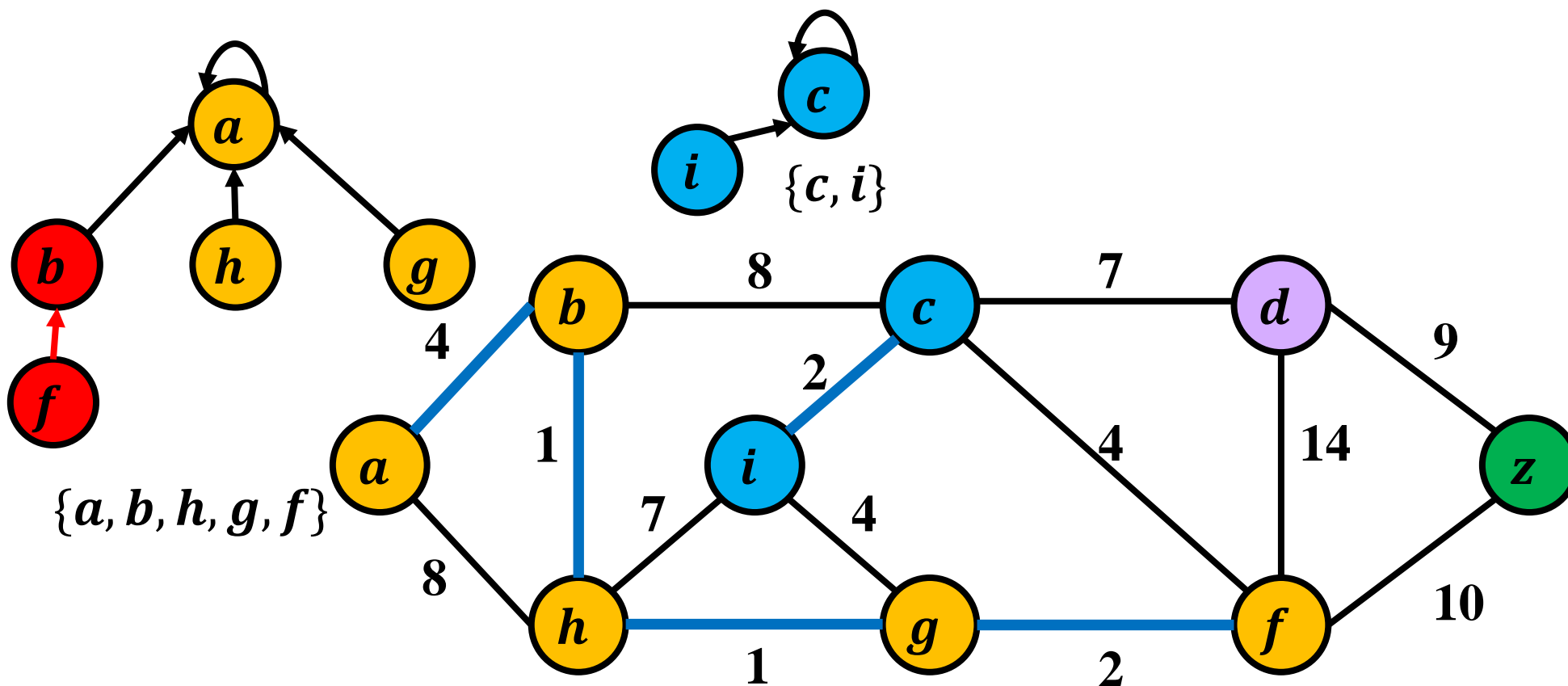


# 不相交集合



- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同

查找 $f$ 的树根：

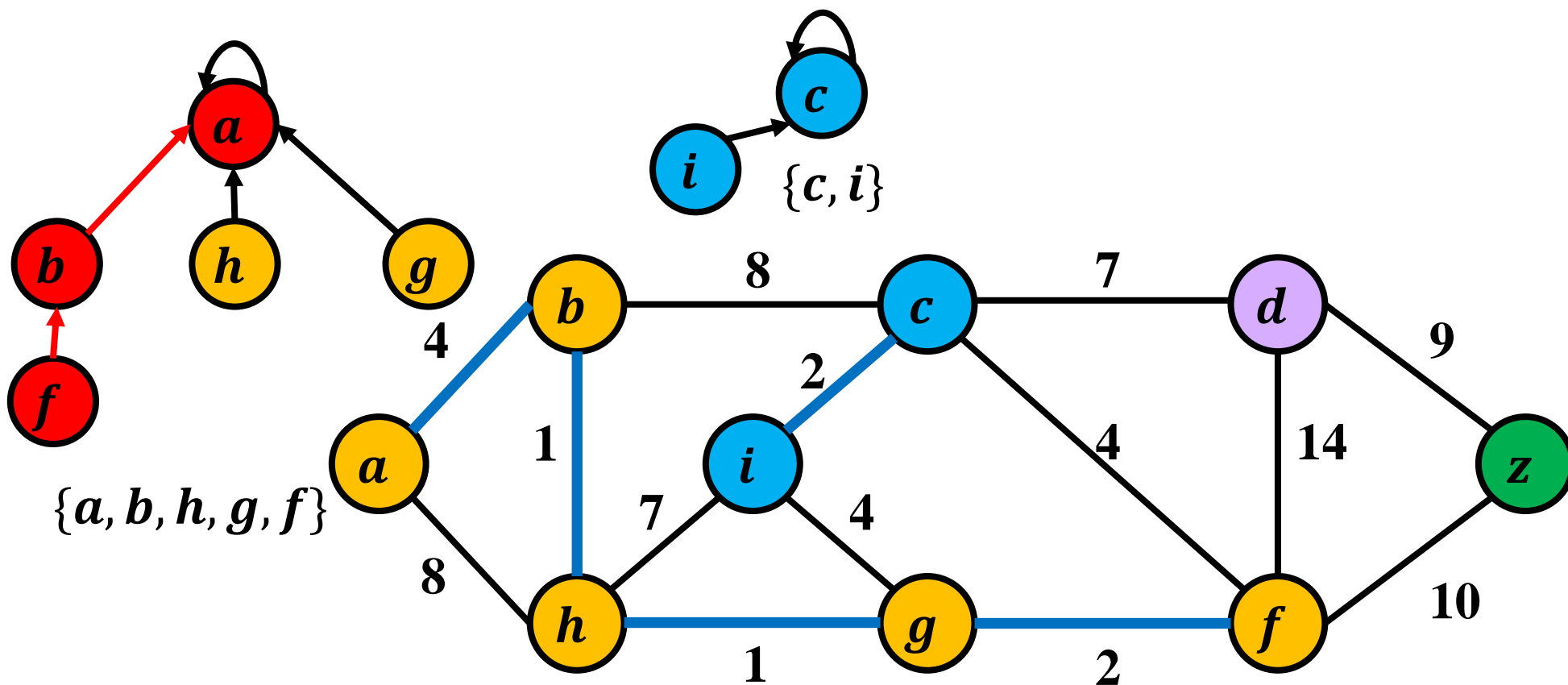


# 不相交集合



- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同

查找 $f$ 的树根：



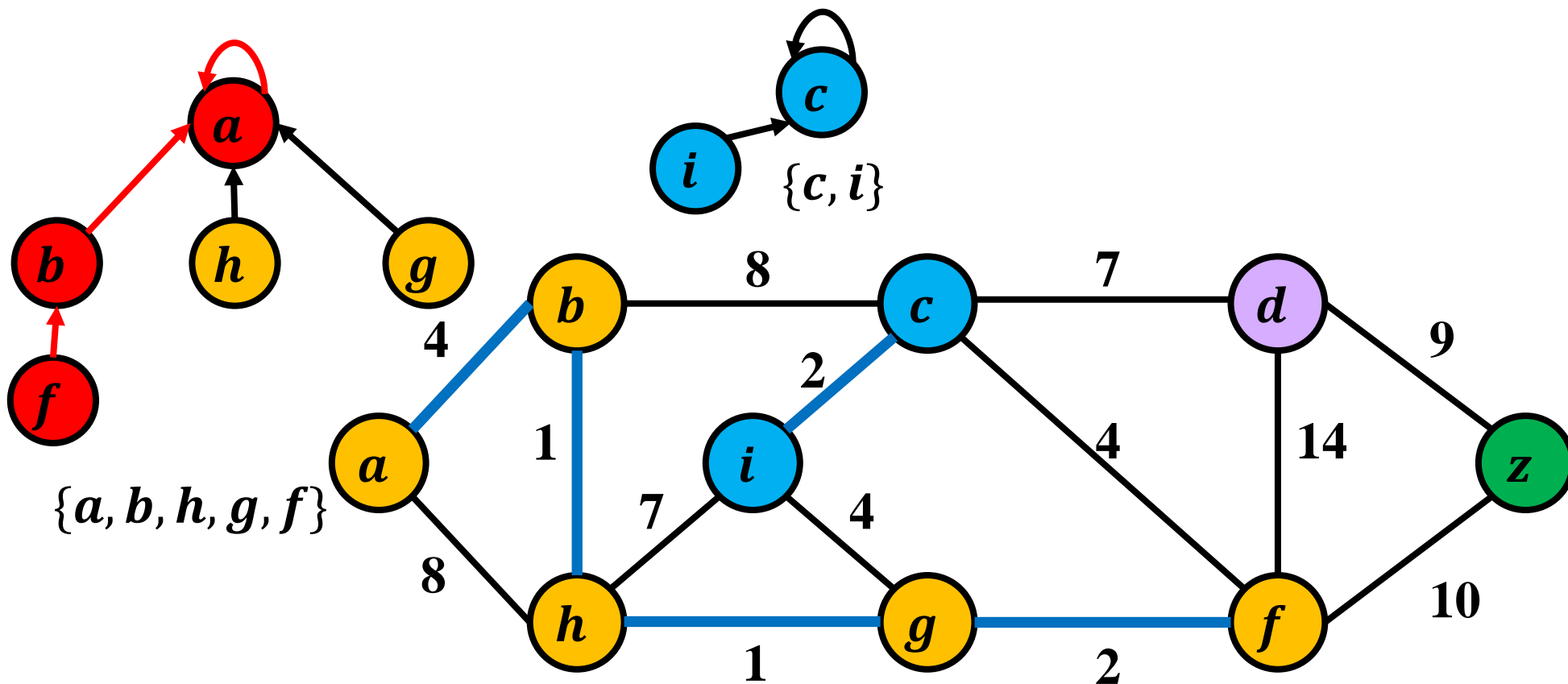


# 不相交集合



- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同

查找 $f$ 的树根： $a$

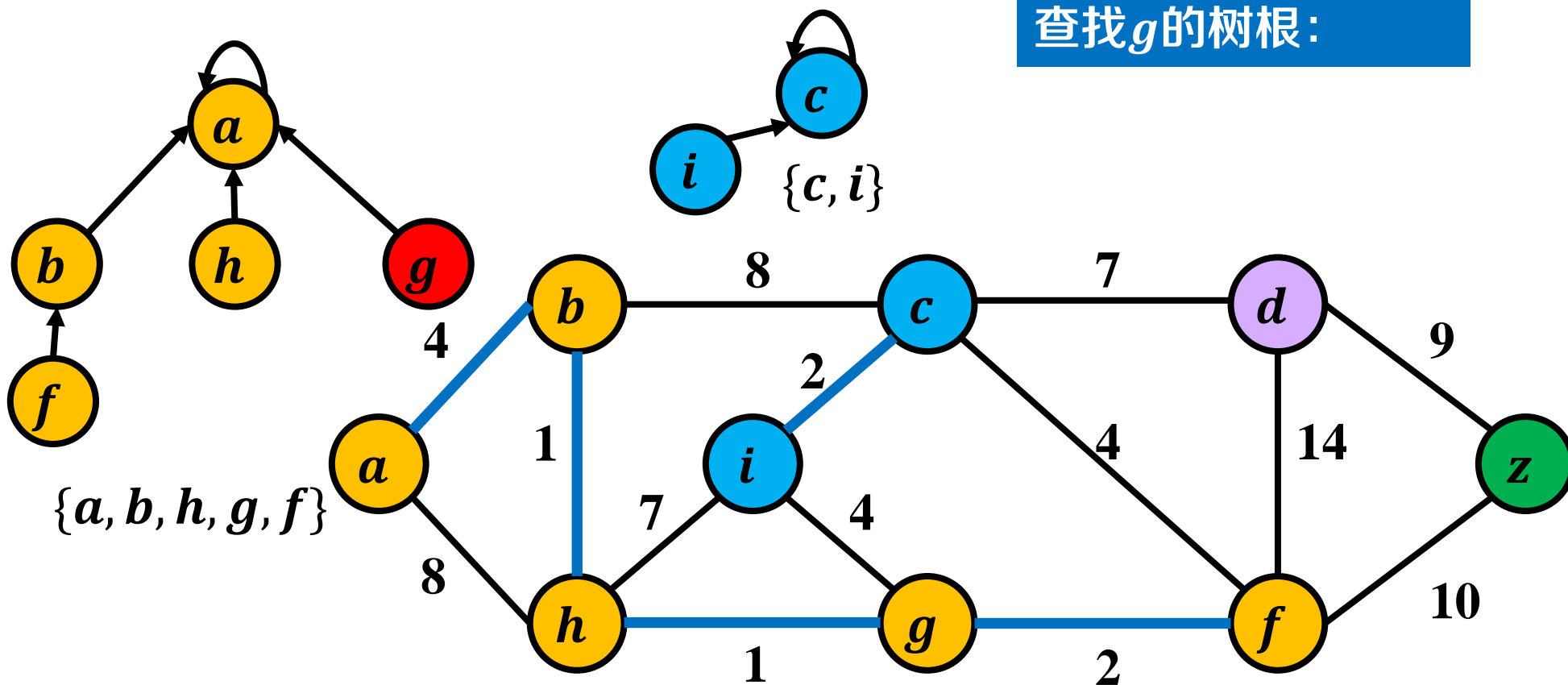


# 不相交集合

- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同

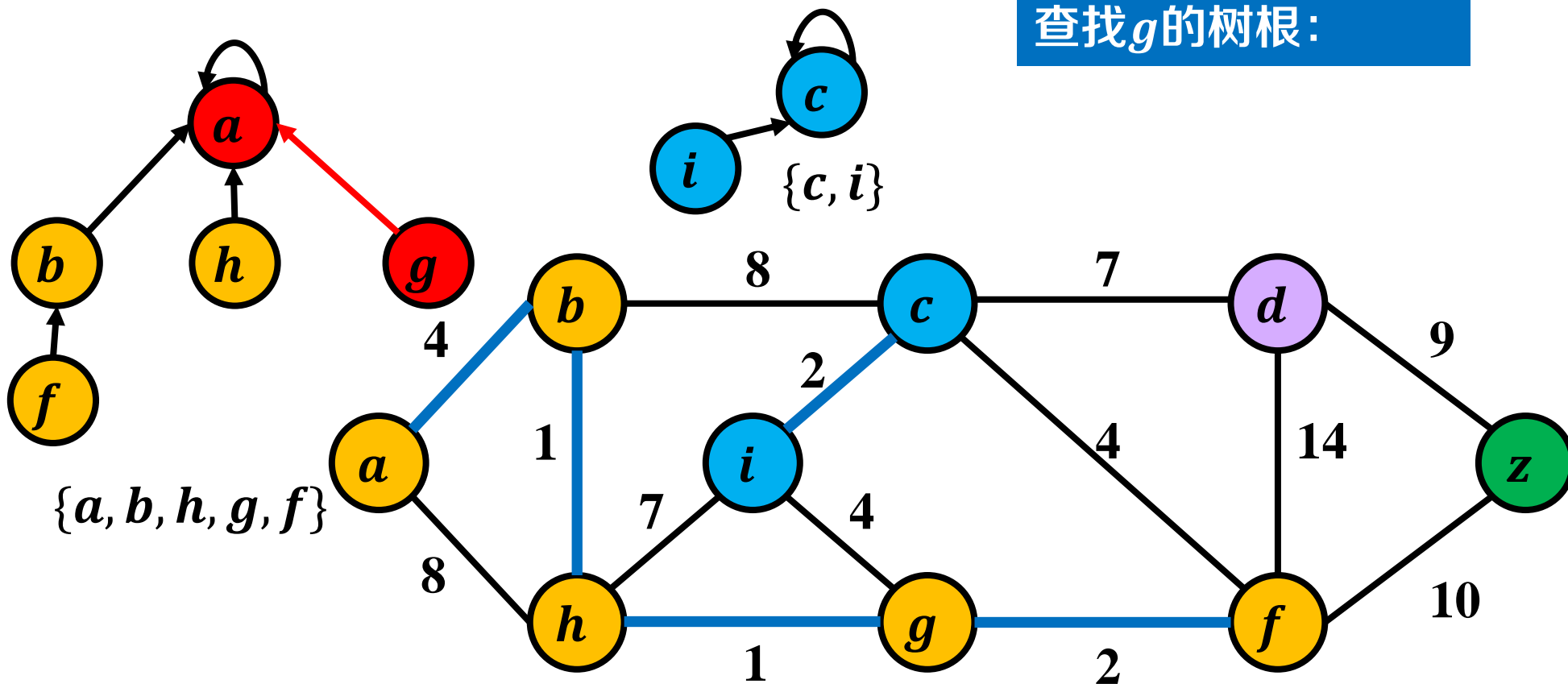
查找 $f$ 的树根： $a$

查找 $g$ 的树根：



- ## 查找 $f$ 的树根: $a$

## 查找 $g$ 的树根:



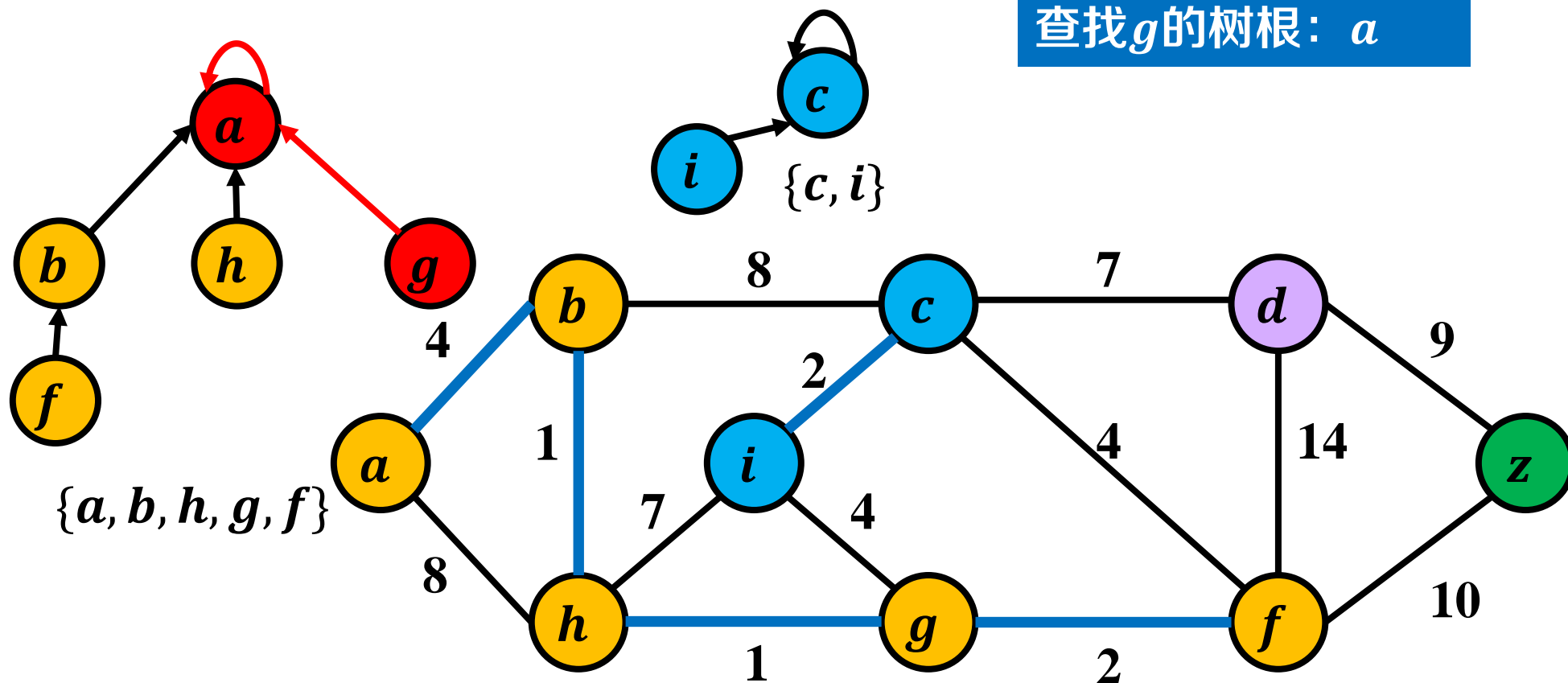
# 不相交集合



- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同

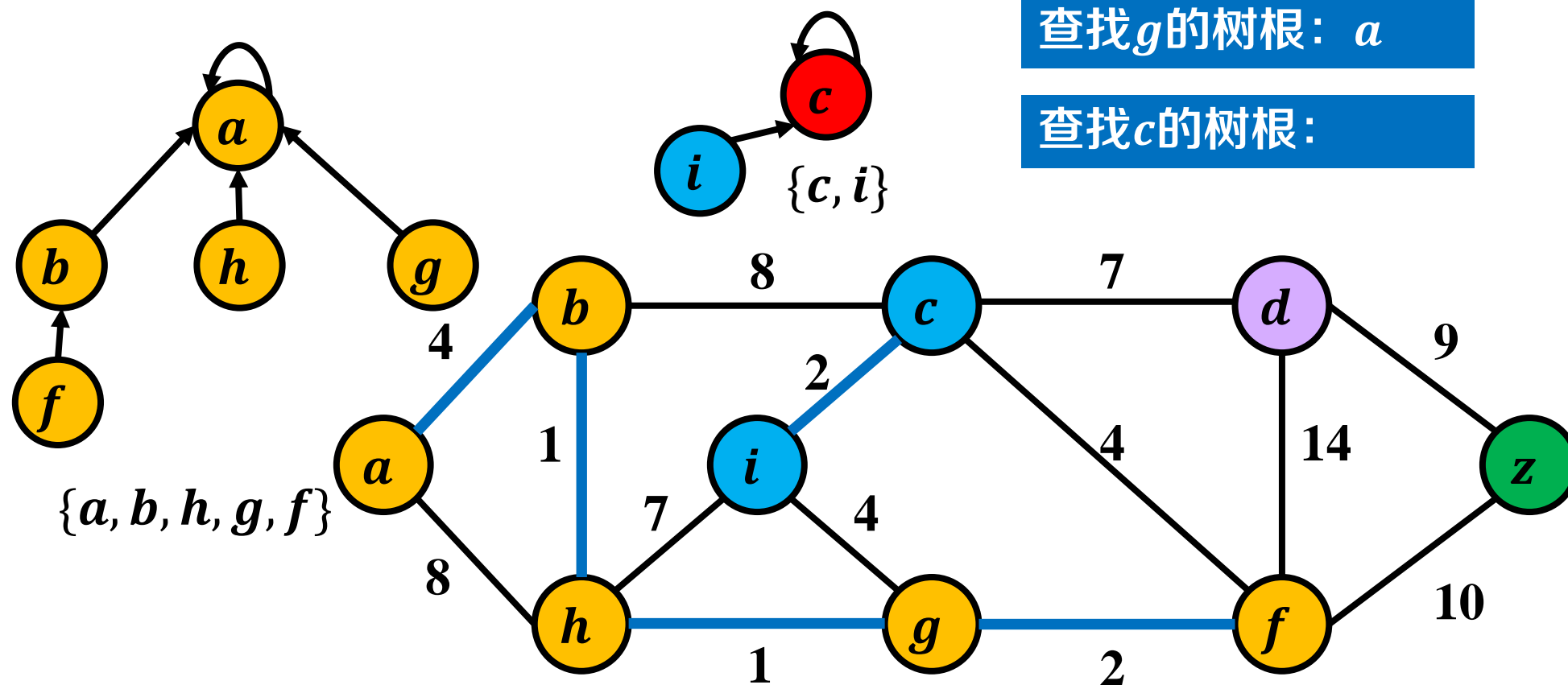
查找 $f$ 的树根： $a$

查找 $g$ 的树根： $a$



# 不相交集合

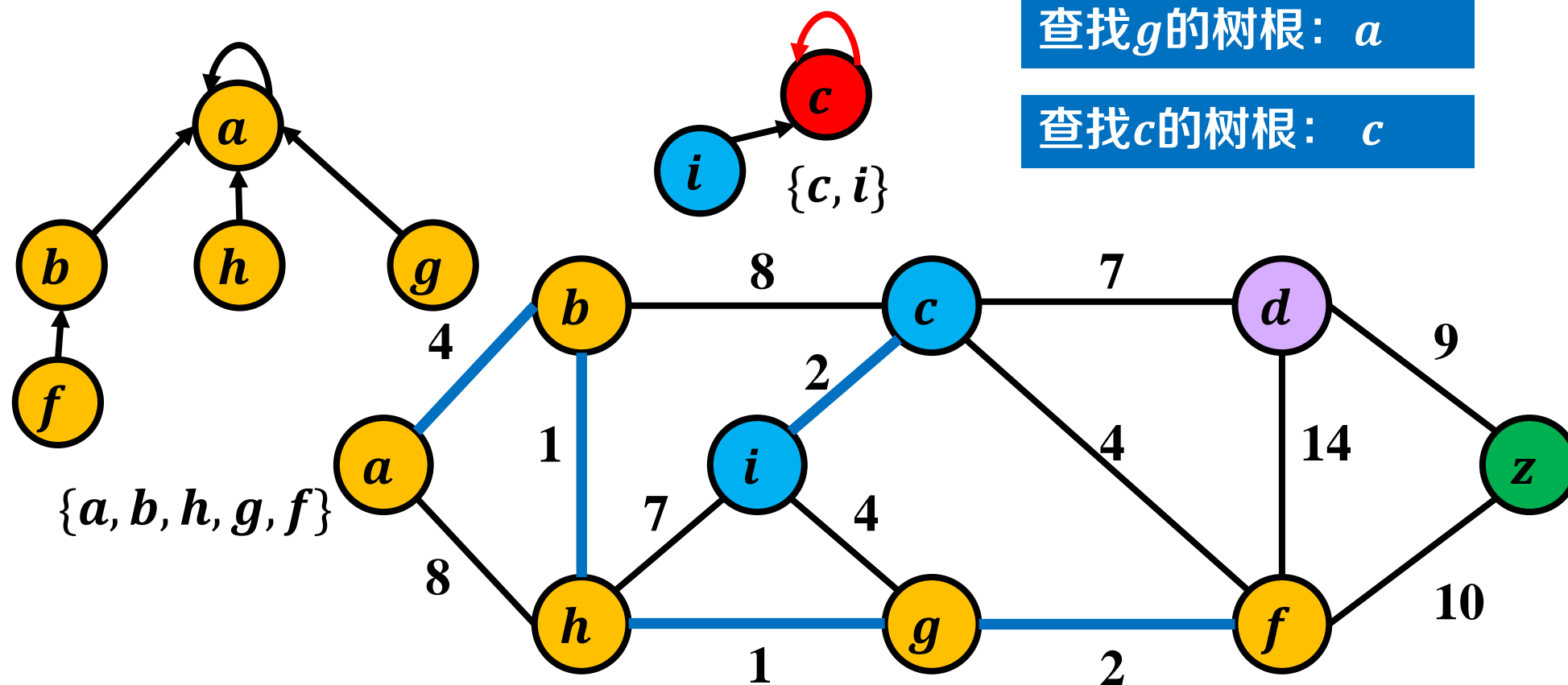
- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同



# 不相交集合



- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同



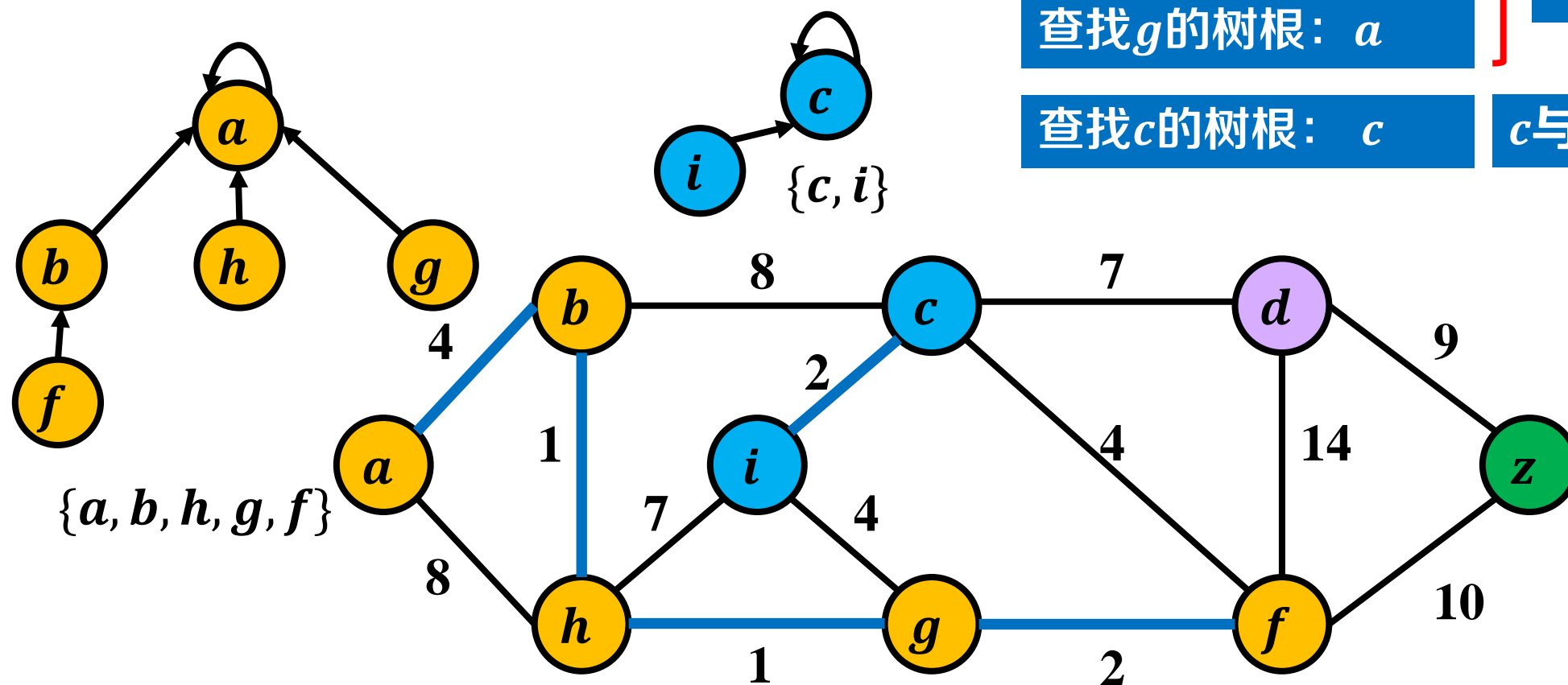
查找  $f$  的树根:  $a$

查找  $g$  的树根:  $a$

查找  $c$  的树根:  $c$

# 不相交集合

- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同



查找  $f$  的树根:  $a$

查找  $g$  的树根:  $a$

查找  $c$  的树根:  $c$

$f$  与  $g$  在同一集合

$c$  与  $f, g$  不在同一集合

# 不相交集合：伪代码



- Create-Set( $x$ )

输入: 顶点  $x$

输出: 不相交集合树

$x.parent \leftarrow x$

return  $x$

- Find-Set( $x$ )

输入: 顶点  $x$

输出: 所属连通分量

while  $x.parent \neq x$  do

|  $x \leftarrow x.parent$

end

return  $x$

回溯查找

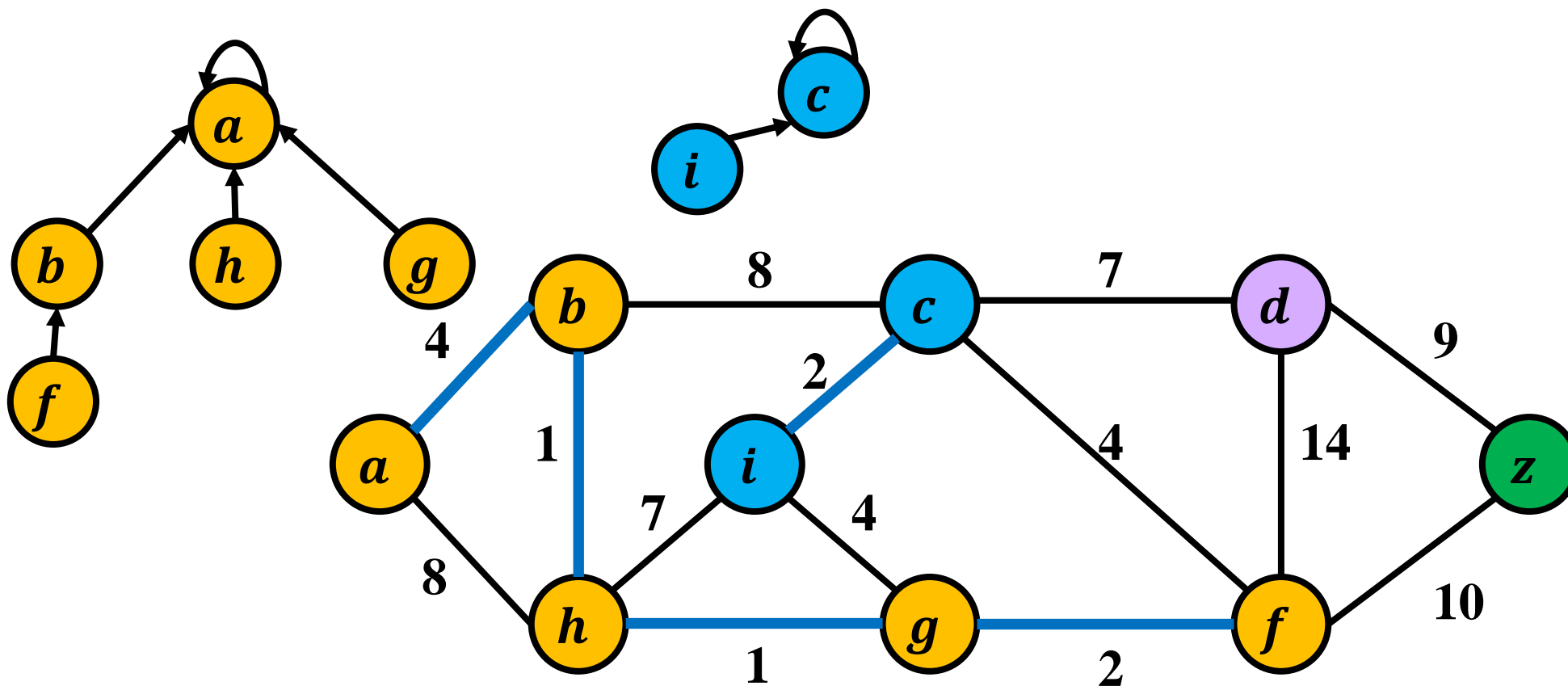


# 不相交集合



- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同
- 合并集合：合并两棵树

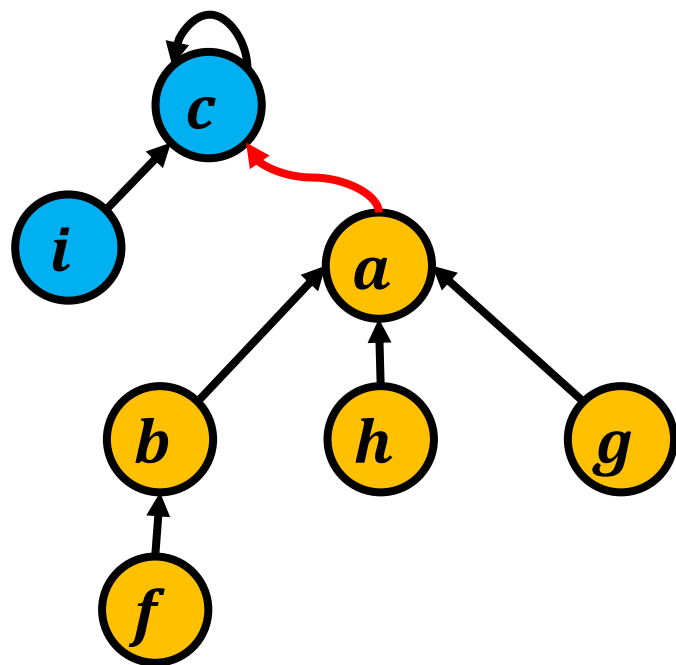
$$\{a, b, h, g, f\} \cup \{c, i\}$$



# 不相交集合

- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同
- 合并集合：合并两棵树

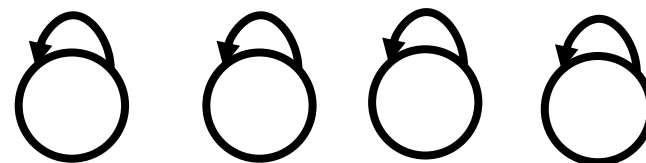
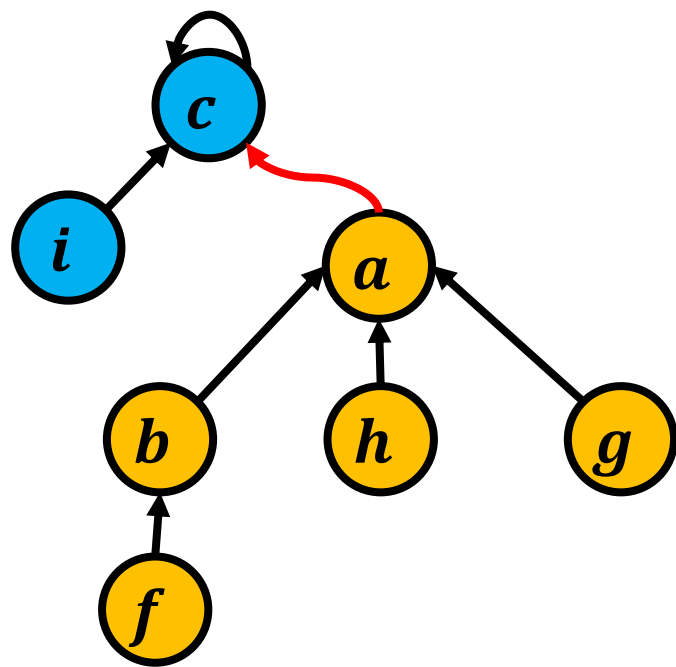
$$\{a, b, h, g, f\} \cup \{c, i\}$$



简单实现：找到两树根，任意连接两棵树

# 不相交集合

- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同
- 合并集合：合并两棵树

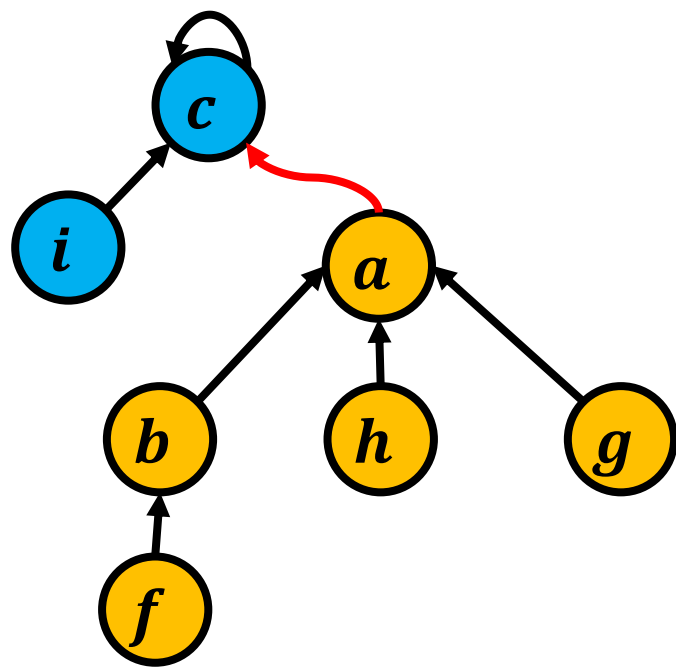


简单实现：找到两树根，任意连接两棵树

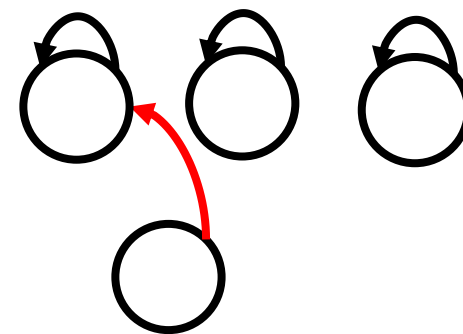
潜在问题：树深度过大，降低查找效率

# 不相交集合

- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同
- 合并集合：合并两棵树



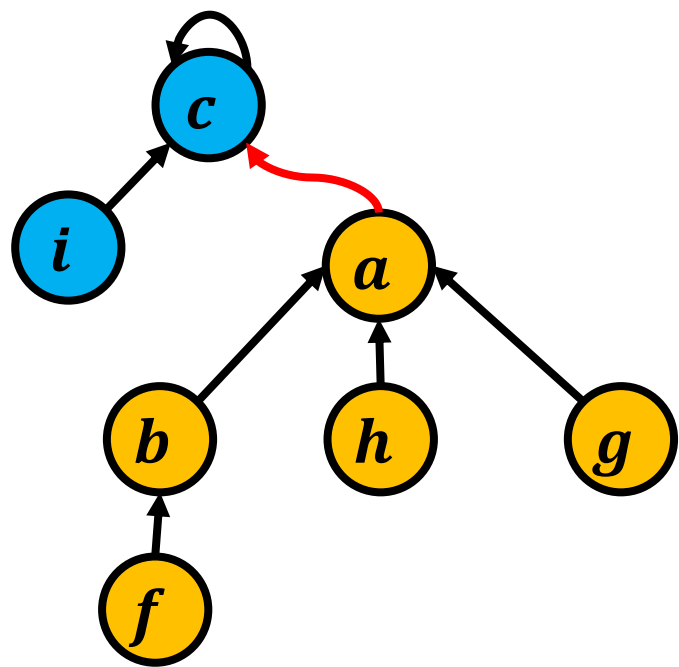
简单实现：找到两树根，任意连接两棵树



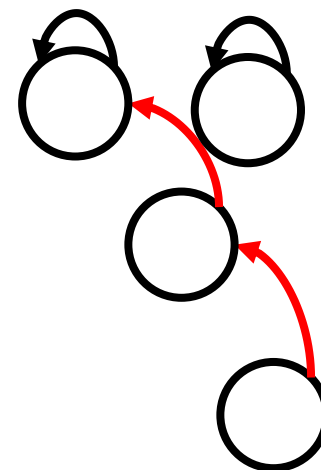
潜在问题：树深度过大，降低查找效率

# 不相交集合

- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同
- 合并集合：合并两棵树



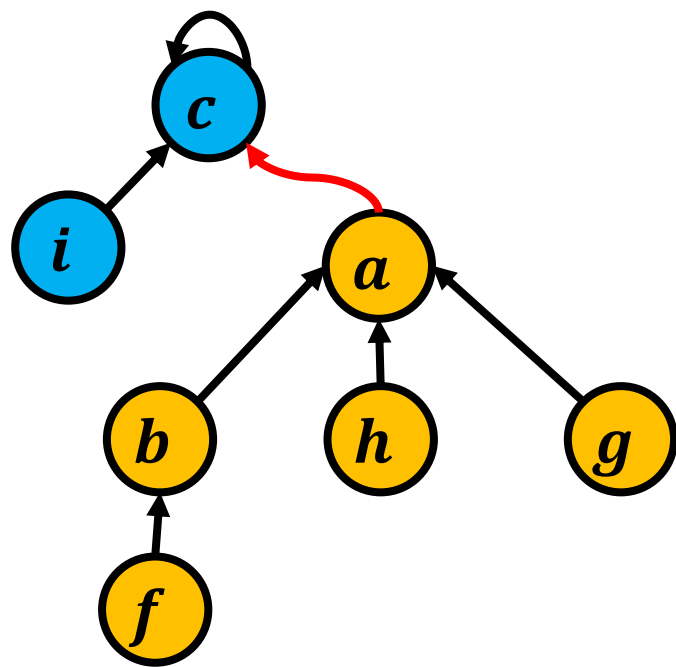
简单实现：找到两树根，任意连接两棵树



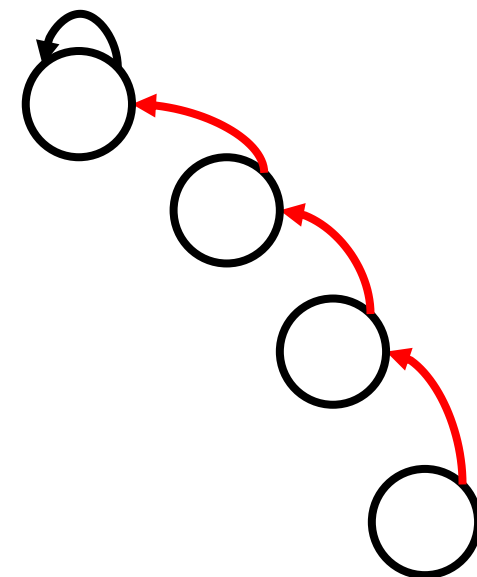
潜在问题：树深度过大，降低查找效率

# 不相交集合

- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同
- 合并集合：合并两棵树



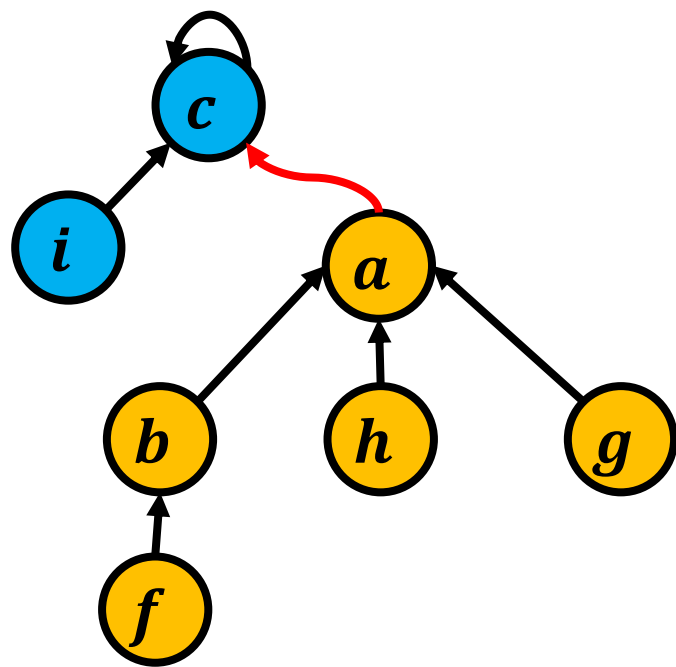
简单实现：找到两树根，任意连接两棵树



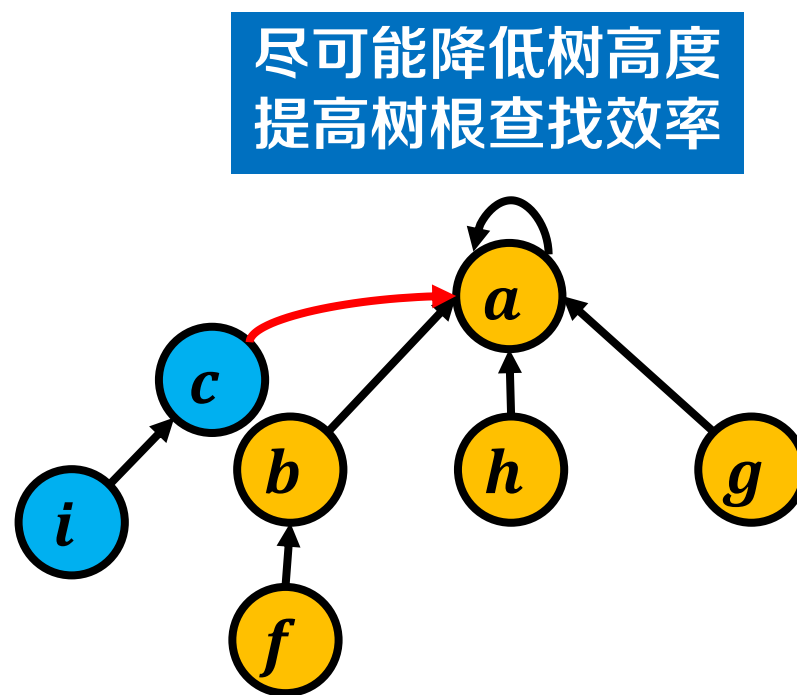
潜在问题：树深度过大，降低查找效率

# 不相交集合

- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同
- 合并集合：合并两棵树



简单实现：找到两树根，任意连接两棵树



高效实现：树高小的树连接到树高大的树上

- Union-Set( $x$ )

输入: 顶点  $x, y$

$a \leftarrow \text{Find-Set}(x)$

$b \leftarrow \text{Find-Set}(y)$

if  $a.\text{height} \leq b.\text{height}$  then

    if  $a.\text{height} = b.\text{height}$  then

$b.\text{height} \leftarrow b.\text{height} + 1$

    end

$a.\text{parent} \leftarrow b$

end

else

$b.\text{parent} \leftarrow a$

end

找到两树根



# 不相交集合：伪代码



- Union-Set( $x$ )

输入: 顶点  $x, y$

$a \leftarrow \text{Find-Set}(x)$

$b \leftarrow \text{Find-Set}(y)$

if  $a.\text{height} \leq b.\text{height}$  then

    if  $a.\text{height} = b.\text{height}$  then

$b.\text{height} \leftarrow b.\text{height} + 1$

    end

$a.\text{parent} \leftarrow b$

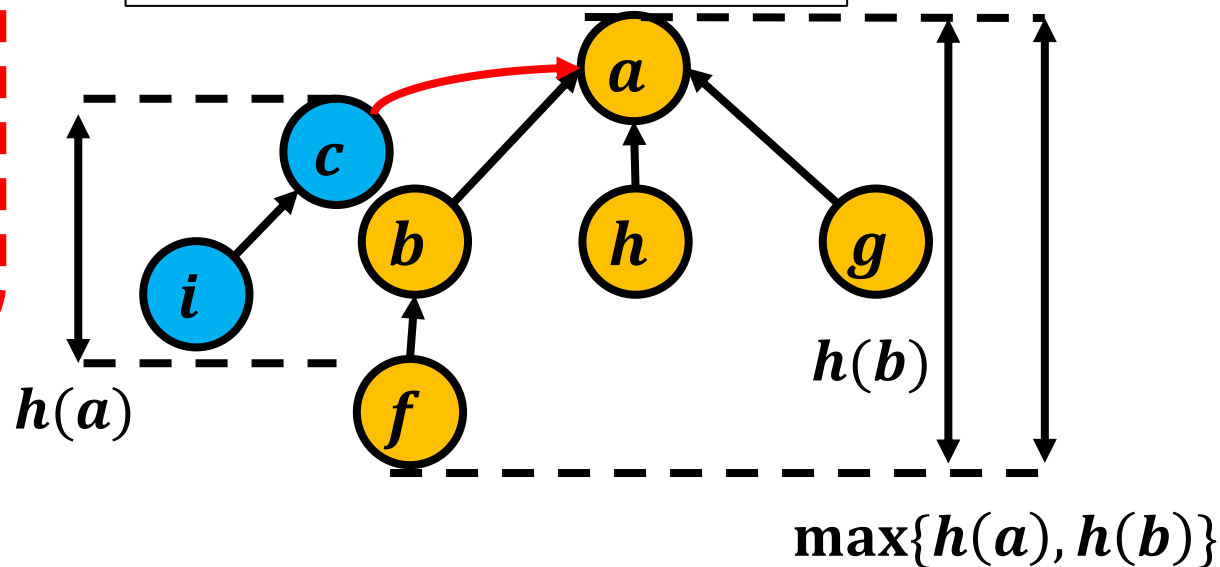
end

else

$b.\text{parent} \leftarrow a$

end

高度小的连到高度大的树



# 不相交集合：伪代码



- Union-Set( $x$ )

输入: 顶点  $x, y$

$a \leftarrow \text{Find-Set}(x)$

$b \leftarrow \text{Find-Set}(y)$

if  $a.\text{height} \leq b.\text{height}$  then

    if  $a.\text{height} = b.\text{height}$  then

$b.\text{height} \leftarrow b.\text{height} + 1$

    end

$a.\text{parent} \leftarrow b$

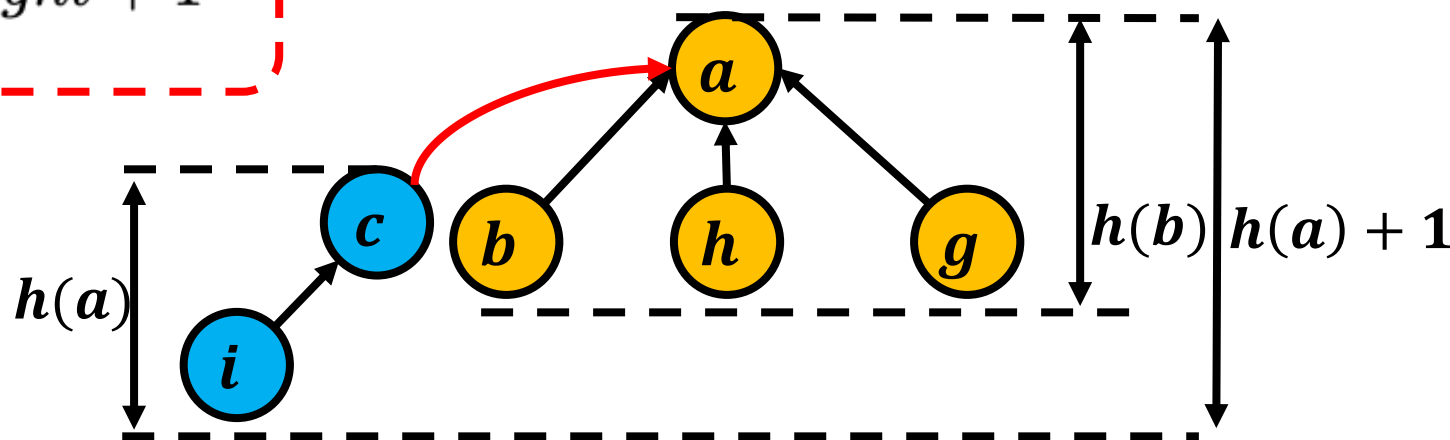
end

else

$b.\text{parent} \leftarrow a$

end

高度相同，需要更新





# 不相交集合：时间复杂度

- Create-Set( $x$ )

输入: 顶点  $x$

输出: 不相交集合树

$x.parent \leftarrow x$

return  $x$

时间复杂度:  $O(1)$

- Find-Set( $x$ )

输入: 顶点  $x$

输出: 所属连通分量

while  $x.parent \neq x$  do

  |  $x \leftarrow x.parent$

end

return  $x$

$h$ 为树高

$O(h)$

时间复杂度:  $O(h)$

- Union-Set( $x$ )

输入: 顶点  $x, y$

$a \leftarrow \text{Find-Set}(x)$

$b \leftarrow \text{Find-Set}(y)$

if  $a.\text{height} \leq b.\text{height}$  then

    if  $a.\text{height} = b.\text{height}$  then

$b.\text{height} \leftarrow b.\text{height} + 1$

    end

$a.\text{parent} \leftarrow b$

end

else

$b.\text{parent} \leftarrow a$

end

$O(h)$

$O(1)$

时间复杂度:  $O(h)$

# 不相交集合：时间复杂度



- Union-Set( $x$ )

输入: 顶点  $x, y$

$a \leftarrow \text{Find-Set}(x)$

$b \leftarrow \text{Find-Set}(y)$

if  $a.\text{height} \leq b.\text{height}$  then

    if  $a.\text{height} = b.\text{height}$  then

$b.\text{height} \leftarrow b.\text{height} + 1$

    end

$a.\text{parent} \leftarrow b$

end

else

$b.\text{parent} \leftarrow a$

end

$O(h)$

$O(1)$

时间复杂度:  $O(h)$

问题: 树的高度 $h$ 和顶点规模 $|V|$ 有何关系?

# 不相交集合：时间复杂度



- 问题：树的高度 $h$ 和顶点规模 $|V|$ 有何关系  $\longrightarrow |V| \geq 2^h$



# 不相交集合：时间复杂度

- 问题：树的高度 $h$ 和顶点规模 $|V|$ 有何关系  $\longrightarrow |V| \geq 2^h$
- 归纳法证明
  - 只有一个顶点，规模 $|V| = 1$ ，高度 $h = 0$ ，显然 $1 \geq 2^0$

# 不相交集合：时间复杂度

- 问题：树的高度 $h$ 和顶点规模 $|V|$ 有何关系  $\longrightarrow |V| \geq 2^h$
- 归纳法证明
  - 只有一个顶点，规模 $|V| = 1$ ，高度 $h = 0$ ，显然 $1 \geq 2^0$
  - 假设：任意不相交集合 $m$ ，高度 $h_m$ 和规模 $V_m$ 满足 $V_m \geq 2^{h_m}$





# 不相交集合：时间复杂度

- 问题：树的高度 $h$ 和顶点规模 $|V|$ 有何关系  $\longrightarrow |V| \geq 2^h$
- 归纳法证明
  - 只有一个顶点，规模 $|V| = 1$ ，高度 $h = 0$ ，显然 $1 \geq 2^0$
  - 假设：任意不相交集合 $m$ ，高度 $h_m$ 和规模 $V_m$ 满足 $V_m \geq 2^{h_m}$
  - 归纳：两不相交集合 $a, b$ 拟做合并，设合并产生的新不相交集合为 $c$

# 不相交集合：时间复杂度

- 问题：树的高度 $h$ 和顶点规模 $|V|$ 有何关系  $\longrightarrow |V| \geq 2^h$
- 归纳法证明
  - 只有一个顶点，规模 $|V| = 1$ ，高度 $h = 0$ ，显然 $1 \geq 2^0$
  - 假设：任意不相交集合 $m$ ，高度 $h_m$ 和规模 $V_m$ 满足 $V_m \geq 2^{h_m}$
  - 归纳：两不相交集合 $a, b$ 拟做合并，设合并产生的新不相交集合为 $c$ 
    - 若 $h_a \neq h_b$ ：  $V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b}$

依照假设

# 不相交集合：时间复杂度

- 问题：树的高度 $h$ 和顶点规模 $|V|$ 有何关系  $\longrightarrow |V| \geq 2^h$
- 归纳法证明
  - 只有一个顶点，规模 $|V| = 1$ ，高度 $h = 0$ ，显然 $1 \geq 2^0$
  - 假设：任意不相交集合 $m$ ，高度 $h_m$ 和规模 $V_m$ 满足 $V_m \geq 2^{h_m}$
  - 归纳：两不相交集合 $a, b$ 拟做合并，设合并产生的新不相交集合为 $c$ 
    - 若 $h_a \neq h_b$ ：  $V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} \geq 2^{\max\{h_a, h_b\}}$

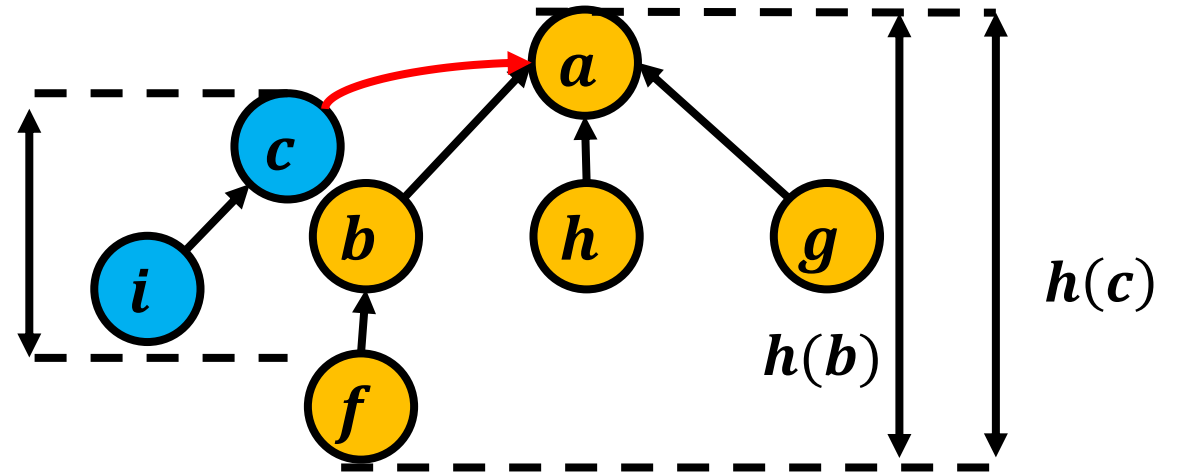
两正数之和大于其中任何一个

# 不相交集合：时间复杂度

- 问题：树的高度 $h$ 和顶点规模 $|V|$ 有何关系  $\longrightarrow |V| \geq 2^h$
- 归纳法证明
  - 只有一个顶点，规模 $|V| = 1$ ，高度 $h = 0$ ，显然 $1 \geq 2^0$
  - 假设：任意不相交集合 $m$ ，高度 $h_m$ 和规模 $V_m$ 满足 $V_m \geq 2^{h_m}$
  - 归纳：两不相交集合 $a, b$ 拟做合并，设合并产生的新不相交集合为 $c$ 
    - 若 $h_a \neq h_b$ ：  $V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} \geq 2^{\max\{h_a, h_b\}} = 2^{h_c}$

```
if a.height ≤ b.height then
  if a.height = b.height then
    | b.height ← b.height + 1
  end
  a.parent ← b
end
else
  | b.parent ← a
end
```

高度不同，新树高为原树中的较大者



$$h(c) = \max\{h(a), h(b)\}$$

# 不相交集合：时间复杂度

- 问题：树的高度 $h$ 和顶点规模 $|V|$ 有何关系  $\longrightarrow |V| \geq 2^h$
- 归纳法证明
  - 只有一个顶点，规模 $|V| = 1$ ，高度 $h = 0$ ，显然 $1 \geq 2^0$
  - 假设：任意不相交集合 $m$ ，高度 $h_m$ 和规模 $V_m$ 满足 $V_m \geq 2^{h_m}$
  - 归纳：两不相交集合 $a, b$ 拟做合并，设合并产生的新不相交集合为 $c$ 
    - 若 $h_a \neq h_b$ ：  $V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} \geq 2^{\max\{h_a, h_b\}} = 2^{h_c}$
    - 若 $h_a = h_b$ ：  $V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} = 2^{h_a+1}$

$$\begin{aligned} & h_a = h_b \\ & 2^{h_a} + 2^{h_b} = 2 \times 2^{h_a} = 2^{h_a+1} \end{aligned}$$

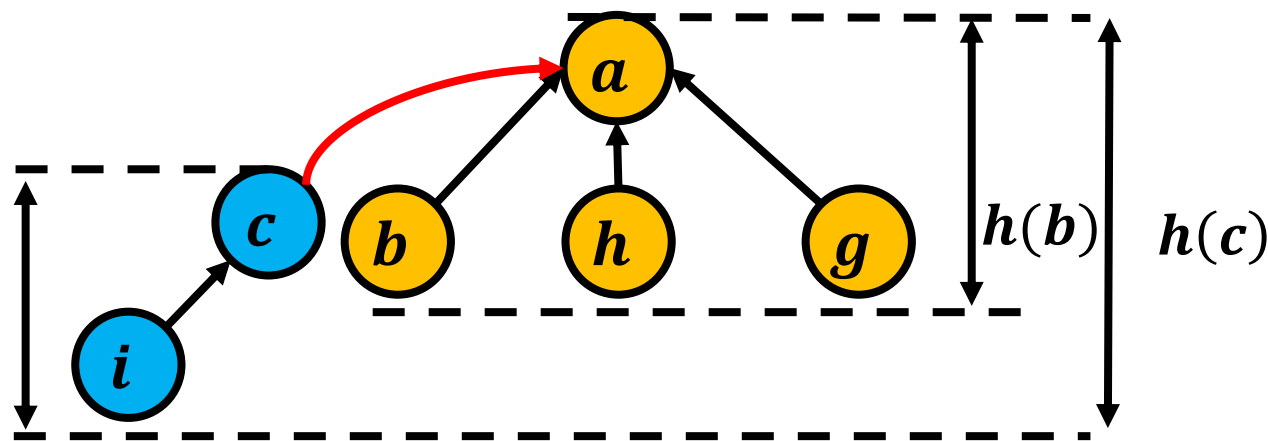
# 不相交集合：时间复杂度

- 问题：树的高度 $h$ 和顶点规模 $|V|$ 有何关系  $\longrightarrow |V| \geq 2^h$
- 归纳法证明
  - 只有一个顶点，规模 $|V| = 1$ ，高度 $h = 0$ ，显然 $1 \geq 2^0$
  - 假设：任意不相交集合 $m$ ，高度 $h_m$ 和规模 $V_m$ 满足 $V_m \geq 2^{h_m}$
  - 归纳：两不相交集合 $a, b$ 拟做合并，设合并产生的新不相交集合为 $c$ 
    - 若 $h_a \neq h_b$ ：  $V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} \geq 2^{\max\{h_a, h_b\}} = 2^{h_c}$
    - 若 $h_a = h_b$ ：  $V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} = 2^{h_a+1} = 2^{h_c}$

```

if a.height < b.height then
    if a.height = b.height then
        | b.height ← b.height + 1
    end
    a.parent ← b
end
else
    | b.parent ← a
end
    
```

高度相同，新树高为原树高+1



$$h(c) = h(a) + 1$$

# 不相交集合：时间复杂度

- 问题：树的高度 $h$ 和顶点规模 $|V|$ 有何关系  $\longrightarrow |V| \geq 2^h$

- 归纳法证明

初始化产生的不相交集合满足 $|V| \geq 2^h$

- 只有一个顶点，规模 $|V| = 1$ ，高度 $h = 0$ ，显然 $1 \geq 2^0$
- 假设：任意不相交集合 $m$ ，高度 $h_m$ 和规模 $V_m$ 满足 $V_m \geq 2^{h_m}$
- 归纳：两不相交集合 $a, b$ 拟做合并，设合并产生的新不相交集合为 $c$ 
  - 若 $h_a \neq h_b$ ：  $V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} \geq 2^{\max\{h_a, h_b\}} = 2^{h_c}$
  - 若 $h_a = h_b$ ：  $V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} = 2^{h_a+1} = 2^{h_c}$

合并产生的不相交集合满足 $|V| \geq 2^h$

- 综上，所有不相交集和都满足 $|V| \geq 2^h$ ，即 $h \leq \log |V|$

# 不相交集合：时间复杂度

- 问题：树的高度 $h$ 和顶点规模 $|V|$ 有何关系  $\longrightarrow |V| \geq 2^h$

- 归纳法证明

初始化产生的不相交集合满足 $|V| \geq 2^h$

- 只有一个顶点，规模 $|V| = 1$ ，高度 $h = 0$ ，显然 $1 \geq 2^0$
- 假设：任意不相交集合 $m$ ，高度 $h_m$ 和规模 $V_m$ 满足 $V_m \geq 2^{h_m}$
- 归纳：两不相交集合 $a, b$ 拟做合并，设合并产生的新不相交集合为 $c$ 
  - 若 $h_a \neq h_b$ ：  $V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} \geq 2^{\max\{h_a, h_b\}} = 2^{h_c}$
  - 若 $h_a = h_b$ ：  $V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} = 2^{h_a+1} = 2^{h_c}$

合并产生的不相交集合满足 $|V| \geq 2^h$

- 综上，所有不相交集和都满足 $|V| \geq 2^h$ ，即 $h \leq \log |V|$
- Create-Set( $x$ ):  $O(1)$
- Find-Set( $x$ ):  $O(h) = O(\log |V|)$
- Union-Set( $x$ ):  $O(h) = O(\log |V|)$



- MST-Kruskal( $G$ )

输入: 图  $G$

输出: 最小生成树

把边按照权重升序排序

$T \leftarrow \{\}$

for  $(u, v) \in E$  do

    if  $u, v$  不在同一子树 then

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

        合并  $u, v$  所在子树

    end

end

return  $T$

问题: 如何高效判定和维护顶点所在的子树?

- MST-Kruskal( $G$ )

输入: 图  $G$

输出: 最小生成树

把边按照权重升序排序

为每个顶点建立不相交集

$T \leftarrow \{\}$

for  $(u, v) \in E$  do

    if Find-Set( $u$ )  $\neq$  Find-Set( $v$ ) then

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

        Union-Set( $u, v$ )

    end

end

return  $T$

创建不相交集

关系检查

合并不相交集

问题的回顾

算法与实例

正确性证明

不相交集合

复杂度分析

- MST-Kruskal( $G$ )

输入: 图  $G$

输出: 最小生成树

把边按照权重升序排序 -----  $O(|E|\log|E|)$

为每个顶点建立不相交集 -----  $O(|V|)$

$T \leftarrow \{\}$

for  $(u, v) \in E$  do

    if Find-Set( $u$ )  $\neq$  Find-Set( $v$ ) then

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

        Union-Set( $u, v$ )

    end

end

return  $T$

$O(\log|V|)$

$O(|E|\log|V|)$

- 时间复杂度

- $O(|E|\log|E| + |E|\log|V|)$

- MST-Kruskal( $G$ )

输入: 图  $G$

输出: 最小生成树

把边按照权重升序排序

-----  $O(|E|\log|E|)$

为每个顶点建立不相交集

-----  $O(|V|)$

$T \leftarrow \{\}$

for  $(u, v) \in E$  do

    if Find-Set( $u$ )  $\neq$  Find-Set( $v$ ) then

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

        Union-Set( $u, v$ )

    end

end

return  $T$

$O(\log|V|)$

$O(|E|\log|V|)$

- 时间复杂度

假设  $|E| = O(|V|^2)$

- $O(|E|\log|E| + |E|\log|V|) = O(|E|\log|V|^2 + |E|\log|V|) = O(|E|\log|V|)$

- Prim算法和Kruskal算法比较

|      | Prim算法       | Kruskal算法     |
|------|--------------|---------------|
| 核心思想 | 保持一棵树，不断扩展   | 子树森林，合并为一棵树   |
| 数据结构 | 优先队列         | 不相交集合         |
| 求解视角 | 微观视角，基于当前点选边 | 宏观视角，基于全局顺序选边 |
| 算法策略 | 都是采用贪心策略的图算法 |               |