图算法篇：最小生成树I

北京航空航天大学
计算机学院

问题定义

最小生成树问题

Minimum Spanning Tree Problem

输入

- 连通无向图 $G =< V, E, W >$, 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 $T =< V_T, E_T >$

优化目标

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

约束条件

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

提纲

问题背景

通用框架

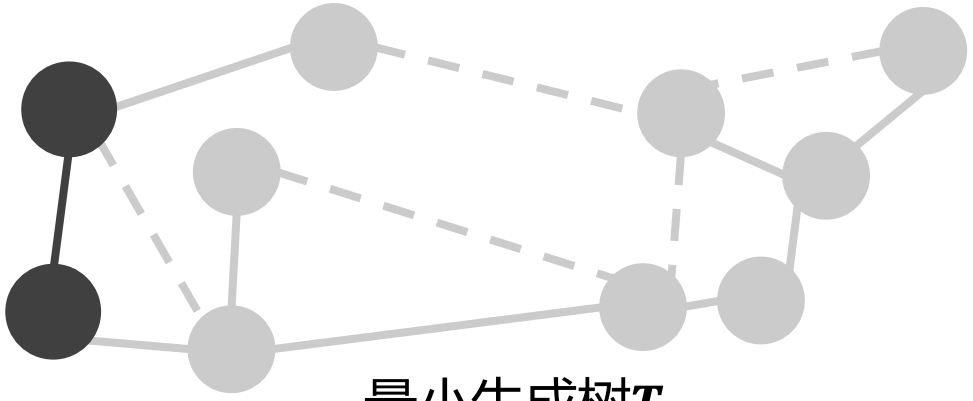
Prim算法

算法实例

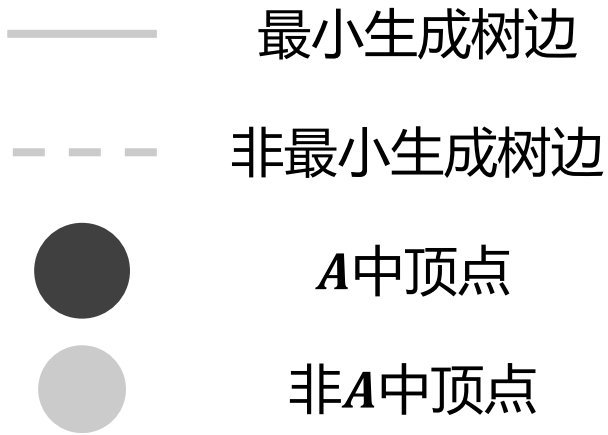
算法分析

通用框架

- 生成树是一个无向图中的连通、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



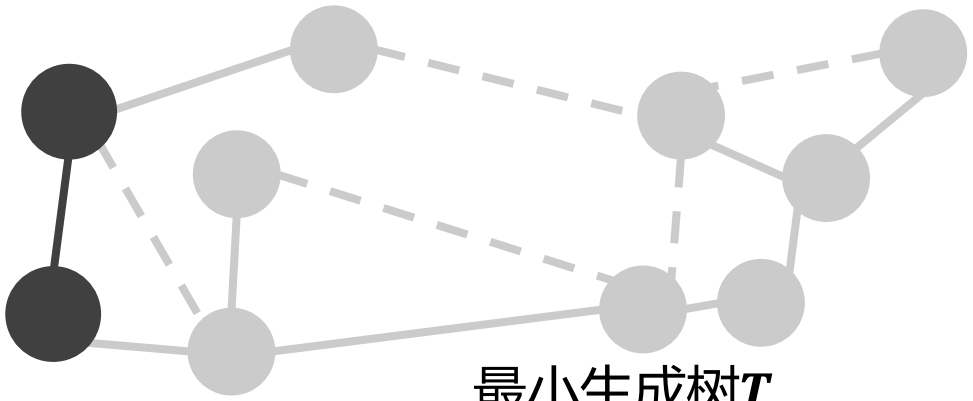
最小生成树T



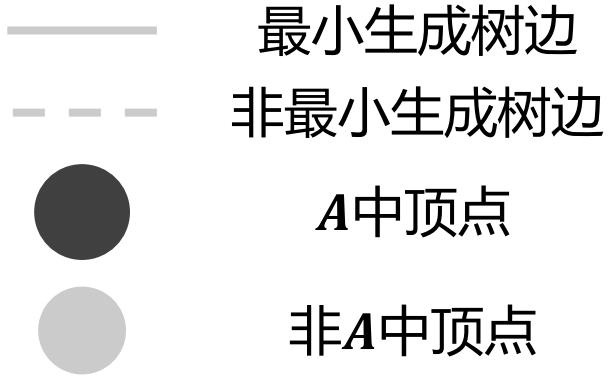
问题：如何保证边集 A 仍是最小生成树的子集？

相关概念

- 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

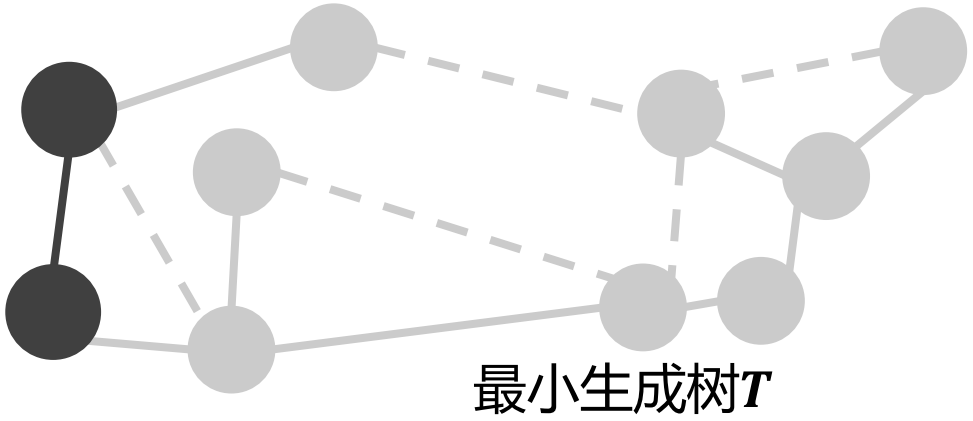


最小生成树T

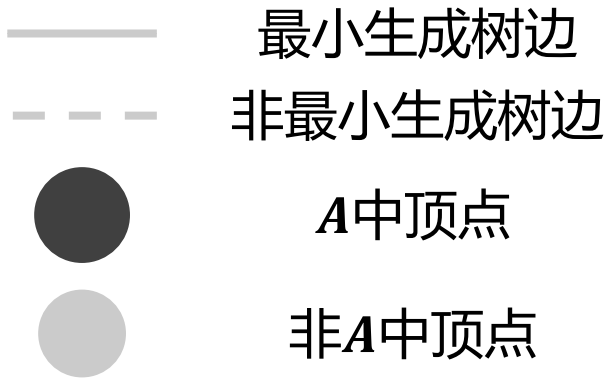


相关概念

- 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



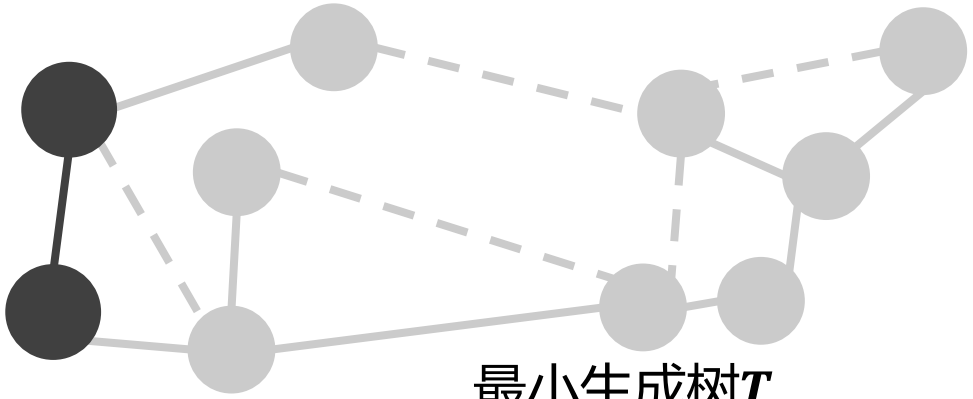
最小生成树T



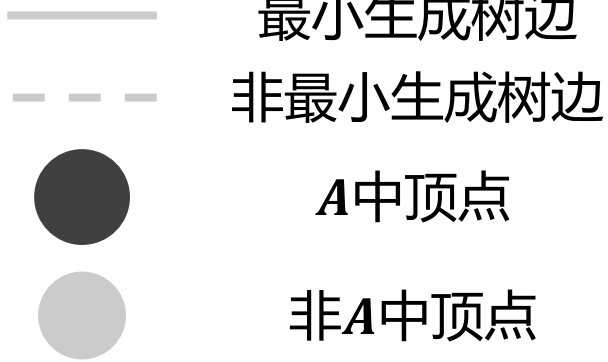
若每次向边集 A 中新增安全边，可保证边集 A 是最小生成树的子集

相关概念

- 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



最小生成树T

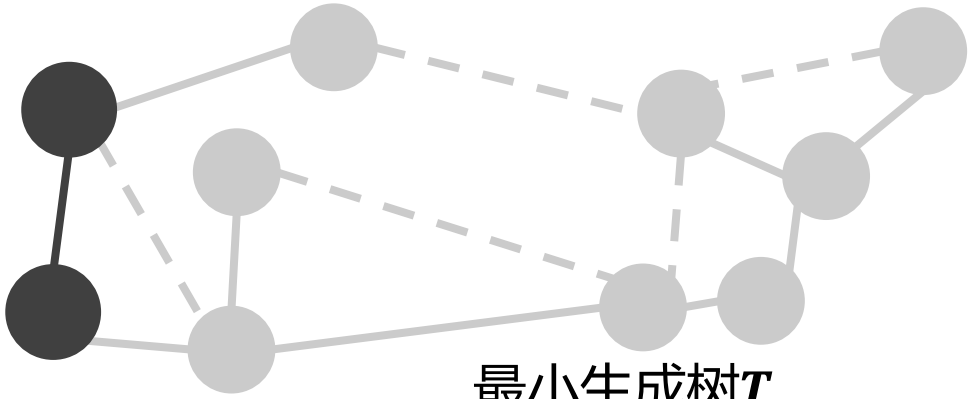


- Generic-MST(G)

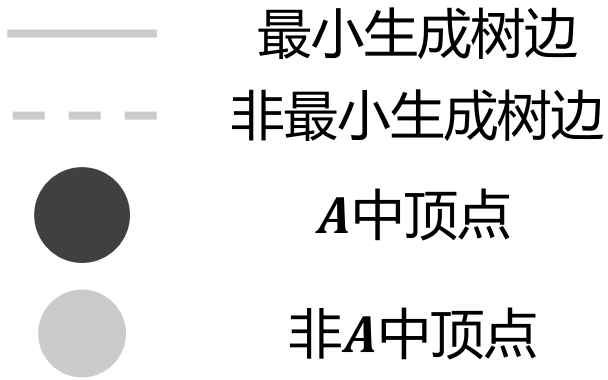
```
A ← ∅
while 没有形成最小生成树 do
    寻找A的安全边(u, v)
    A ← A ∪ (u, v)
end
return A
```

相关概念

- 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



最小生成树T



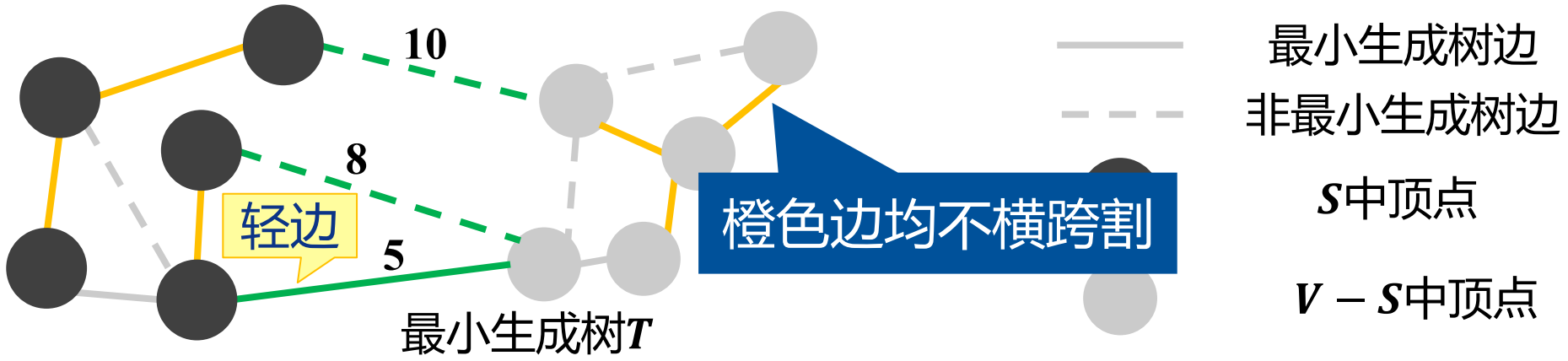
- Generic-MST(G)

```
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
```

问题：如何有效辨识安全边?

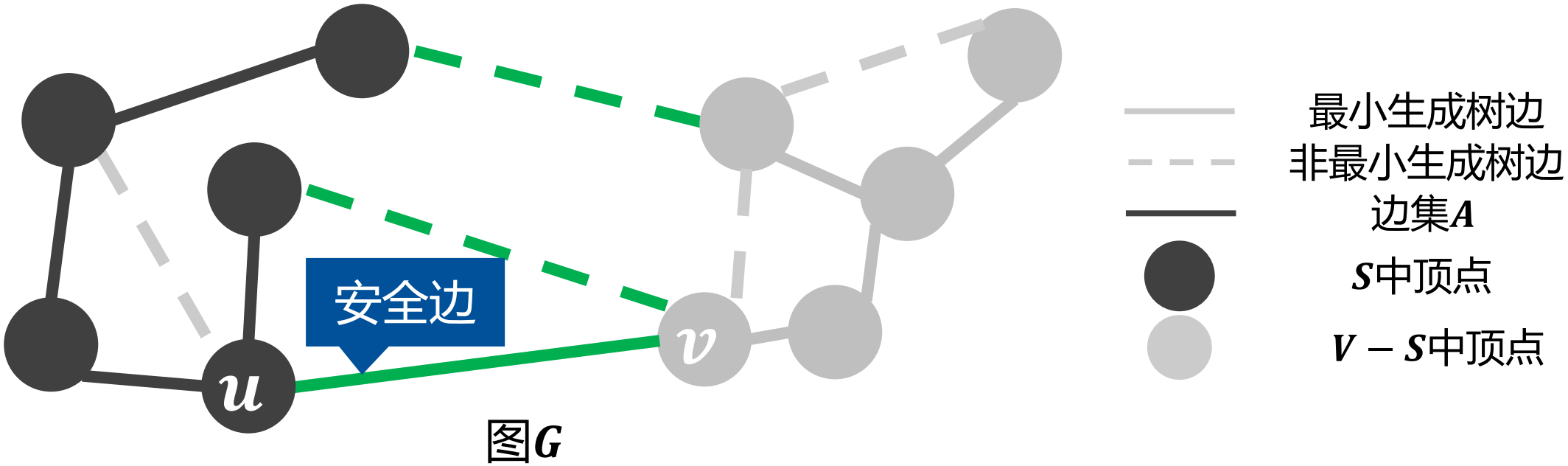
相关概念

- 割(Cut)
  - 图$G =< V, E >$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



安全边辨识定理

- 给定图$G =< V, E >$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



图G

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

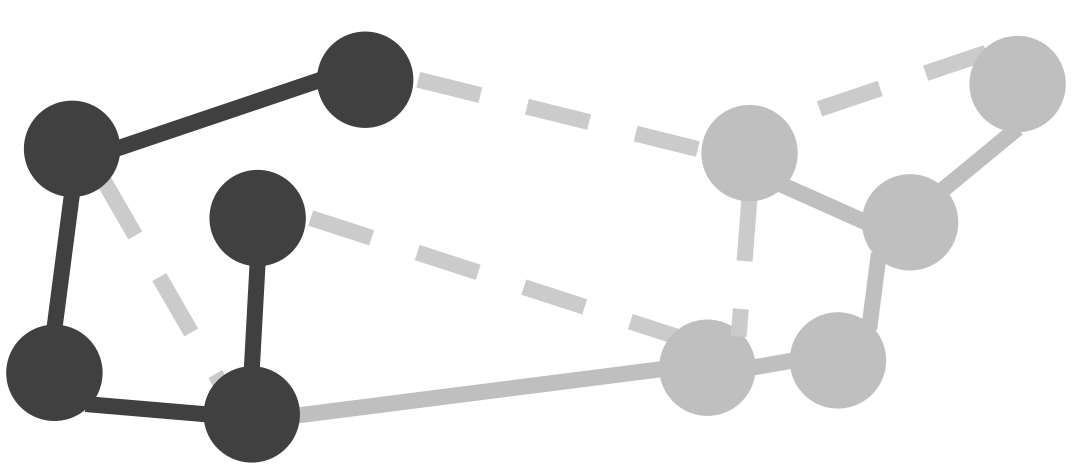
添加一条轻边

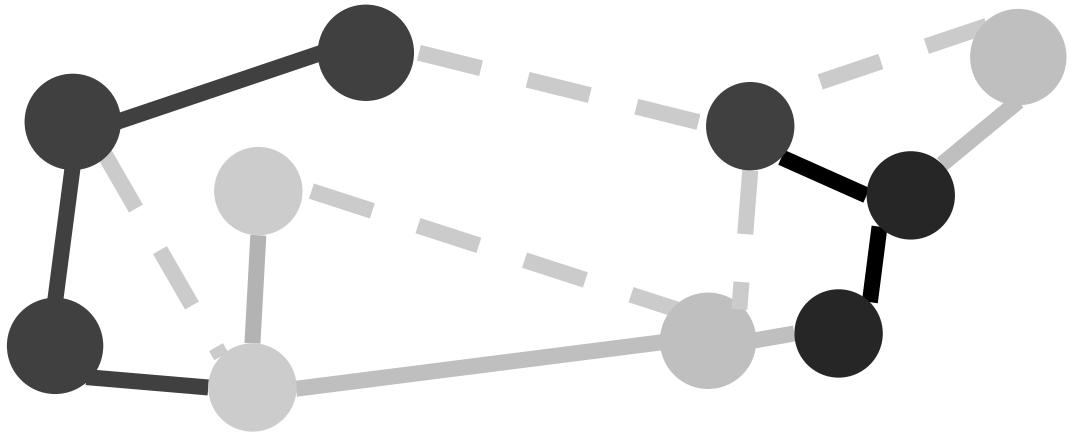
问题：如何有效地实现此贪心策略？

- Prim算法
- Kruskal算法

对比

通用框架	Prim算法	Kruskal算法
判断是否成环	保持树的结构	维护一个森林 （使用并查集）
高效寻找轻边	使用优先队列	只需进行一次排序





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

提纲

问题背景

通用框架

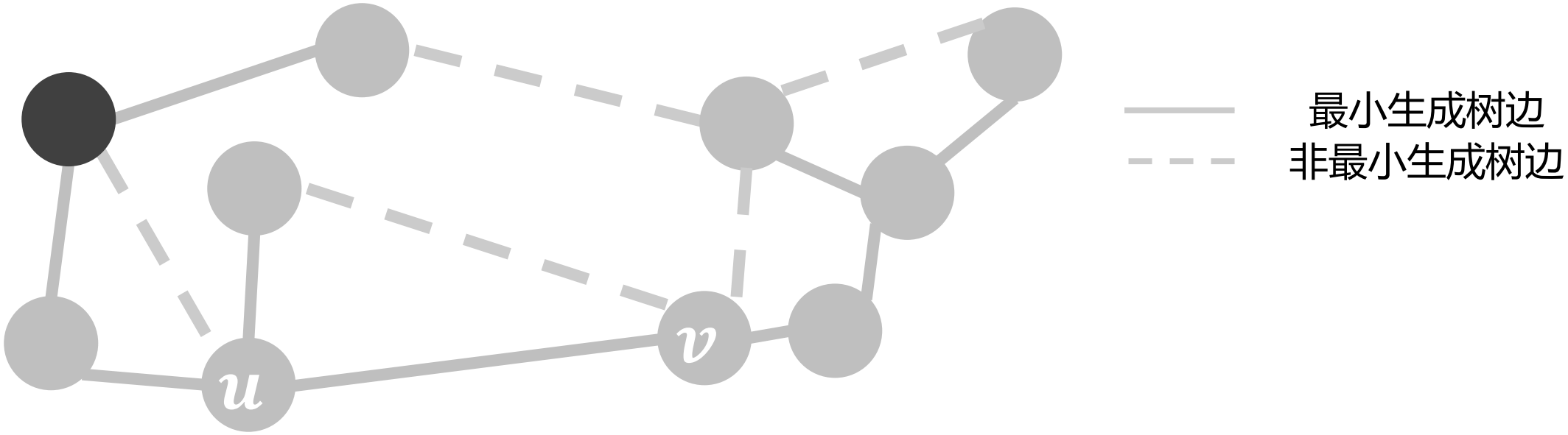
Prim算法

算法实例

算法分析

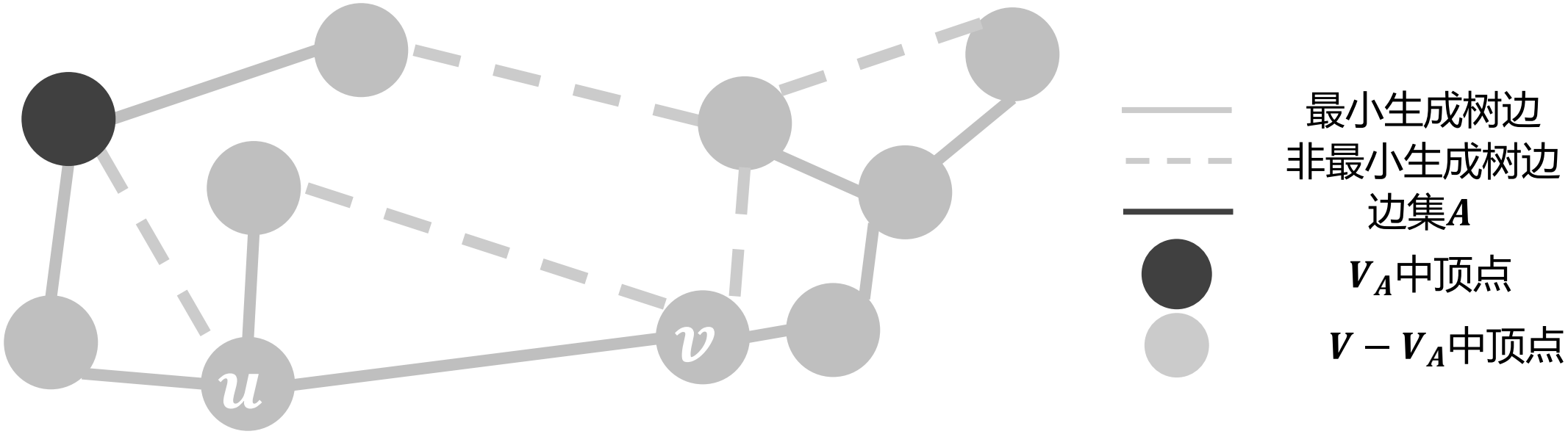
Prim算法

- 算法思想
  - 步骤1：选择任意一个顶点，作为生成树的起始顶点



Prim算法

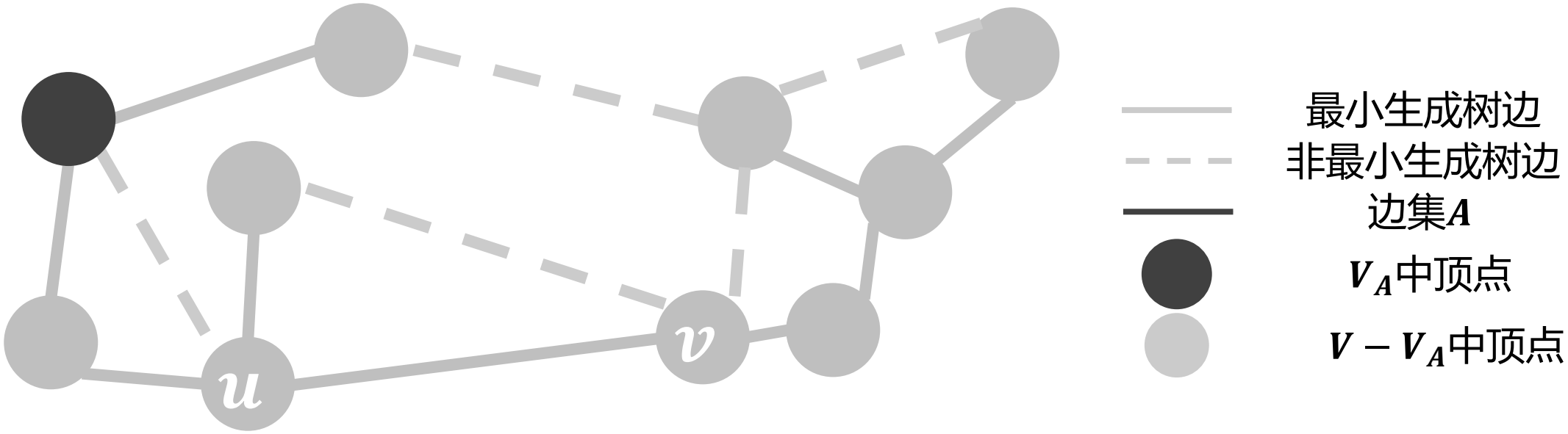
- 算法思想
  - 步骤1：选择任意一个顶点，作为生成树的起始顶点
  - 步骤2：保持边集A始终为一棵树



Prim算法

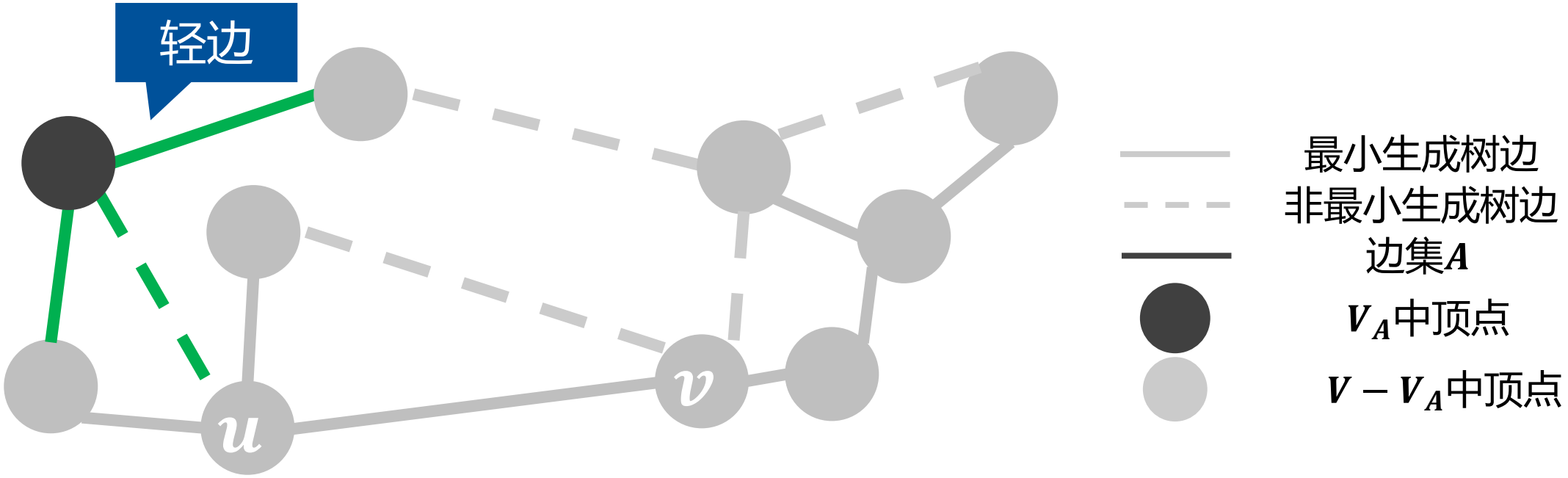
- 算法思想
  - 步骤1：选择任意一个顶点，作为生成树的起始顶点
  - 步骤2：保持边集A始终为一棵树，选择割$(V_A, V-V_A)$

树中顶点



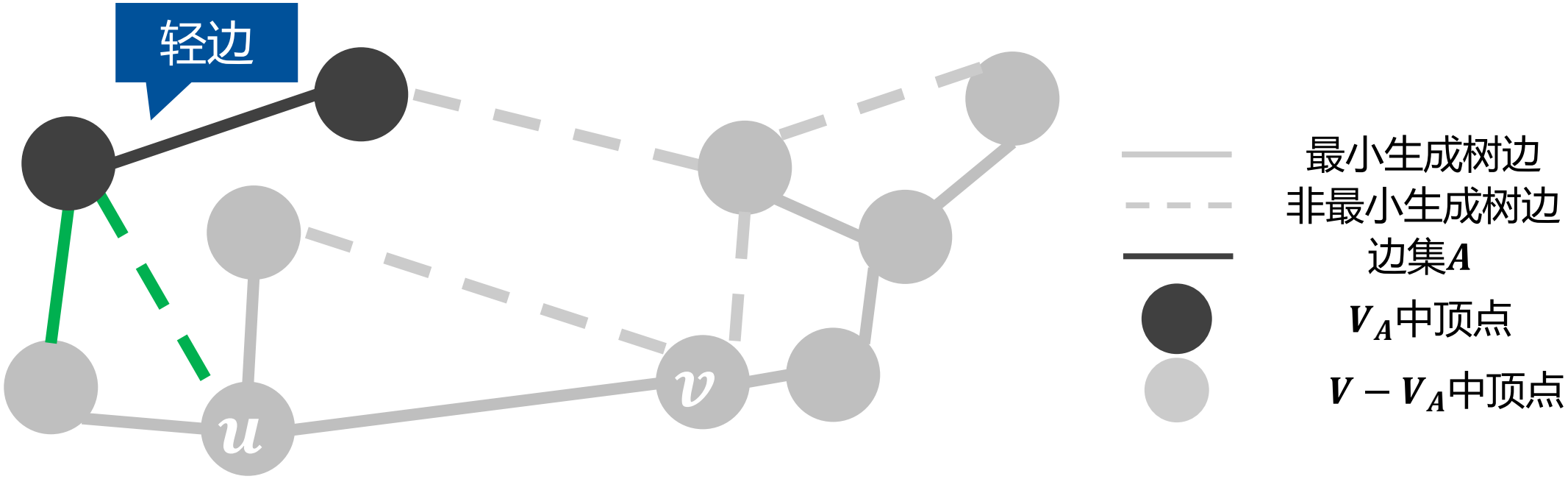
Prim算法

- 算法思想
  - 步骤1：选择任意一个顶点，作为生成树的起始顶点
  - 步骤2：保持边集A始终为一棵树，选择割$(V_A, V-V_A)$
  - 步骤3：选择横跨割$(V_A, V-V_A)$的轻边



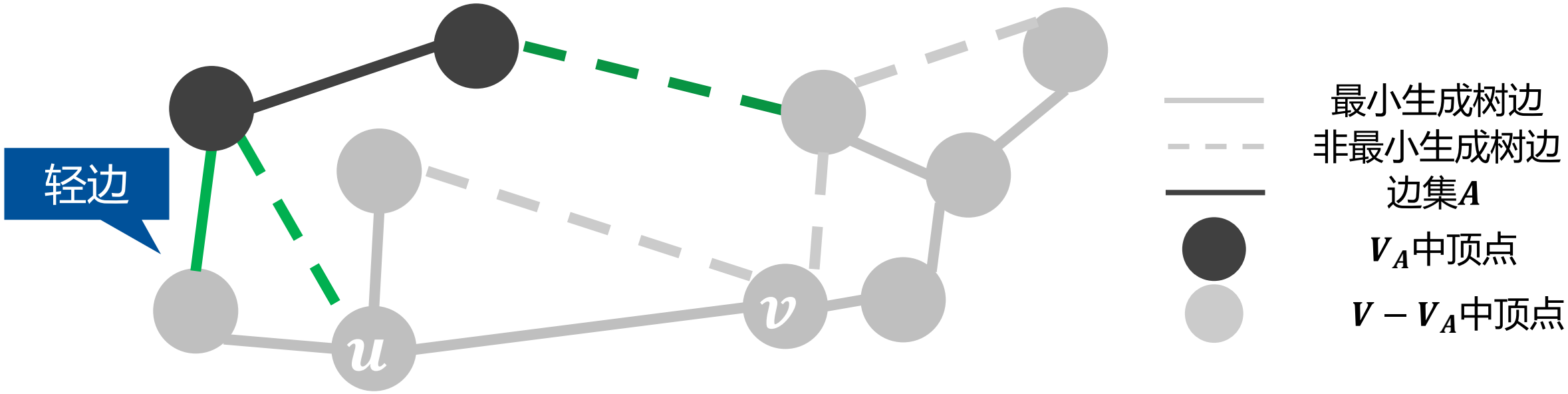
Prim算法

- 算法思想
  - 步骤1：选择任意一个顶点，作为生成树的起始顶点
  - 步骤2：保持边集A始终为一棵树，选择割$(V_A, V-V_A)$
  - 步骤3：选择横跨割$(V_A, V-V_A)$的轻边，添加到边集A中



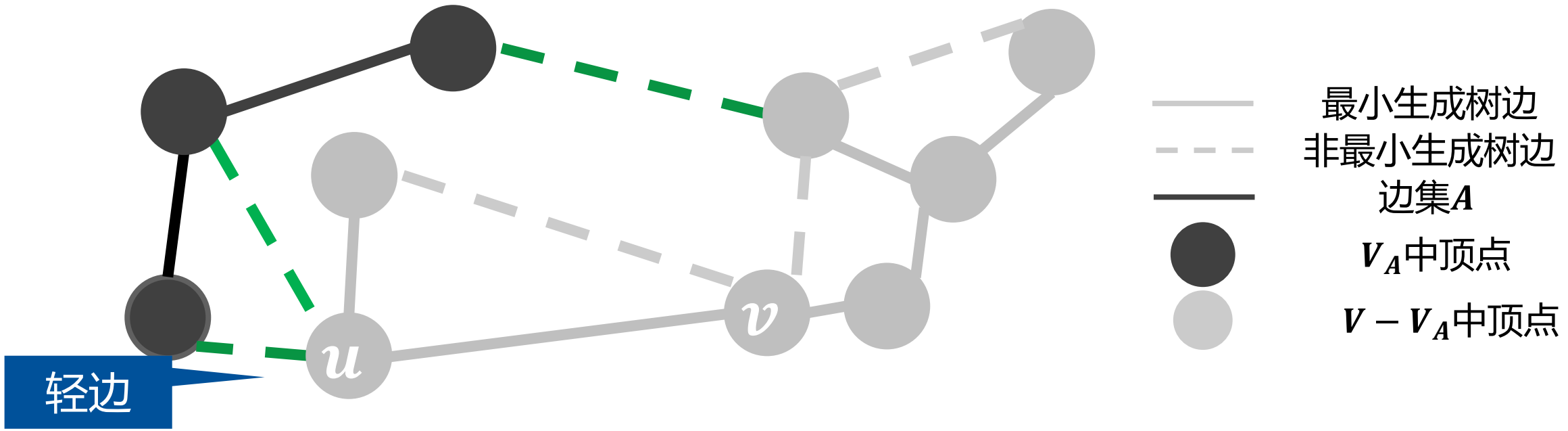
Prim算法

- 算法思想
  - 步骤1：选择任意一个顶点，作为生成树的起始顶点
  - 步骤2：保持边集A始终为一棵树，选择割$(V_A, V-V_A)$
  - 步骤3：选择横跨割$(V_A, V-V_A)$的轻边，添加到边集A中
  - 步骤4：重复步骤2和步骤3，直至覆盖所有顶点



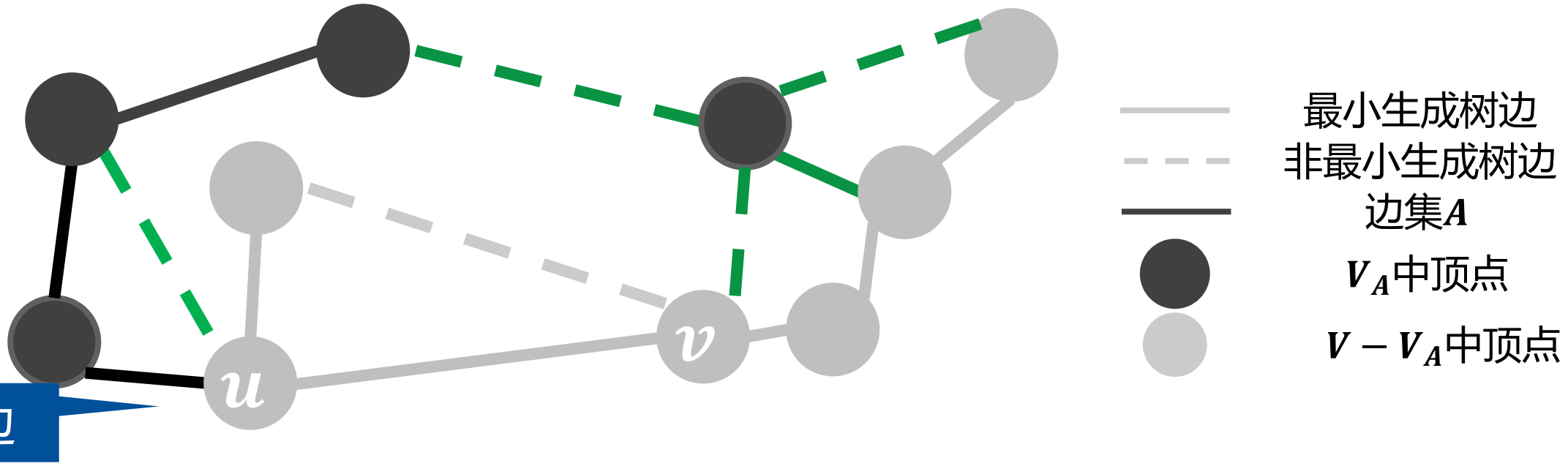
Prim算法

- 算法思想
  - 步骤1：选择任意一个顶点，作为生成树的起始顶点
  - 步骤2：保持边集A始终为一棵树，选择割$(V_A, V-V_A)$
  - 步骤3：选择横跨割$(V_A, V-V_A)$的轻边，添加到边集A中
  - 步骤4：重复步骤2和步骤3，直至覆盖所有顶点



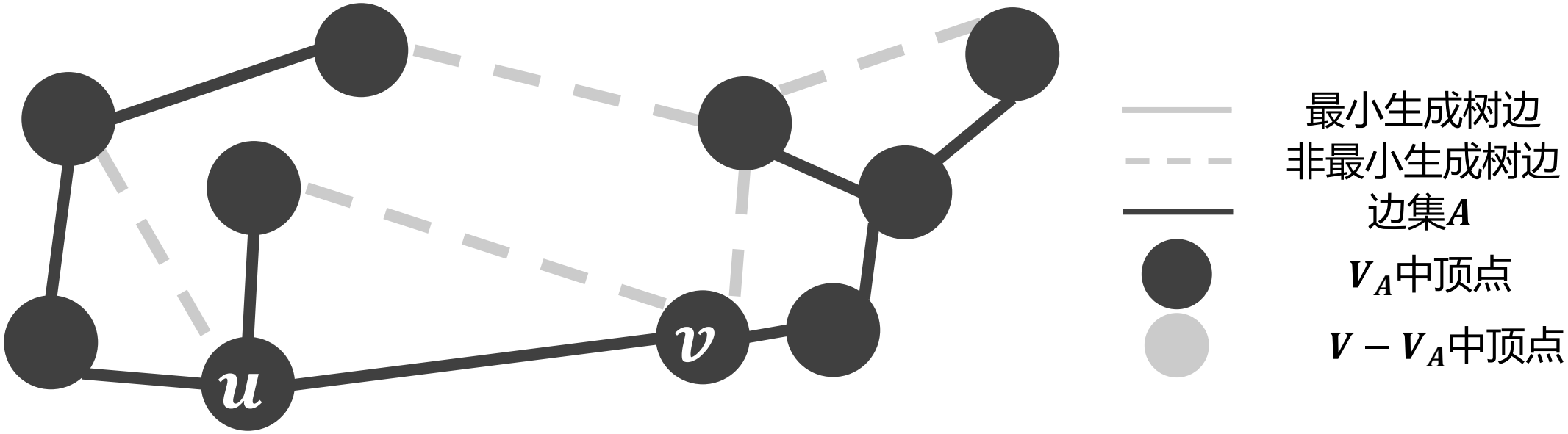
Prim算法

- 算法思想
  - 步骤1：选择任意一个顶点，作为生成树的起始顶点
  - 步骤2：保持边集A始终为一棵树，选择割$(V_A, V-V_A)$
  - 步骤3：选择横跨割$(V_A, V-V_A)$的轻边，添加到边集A中
  - 步骤4：重复步骤2和步骤3，直至覆盖所有顶点



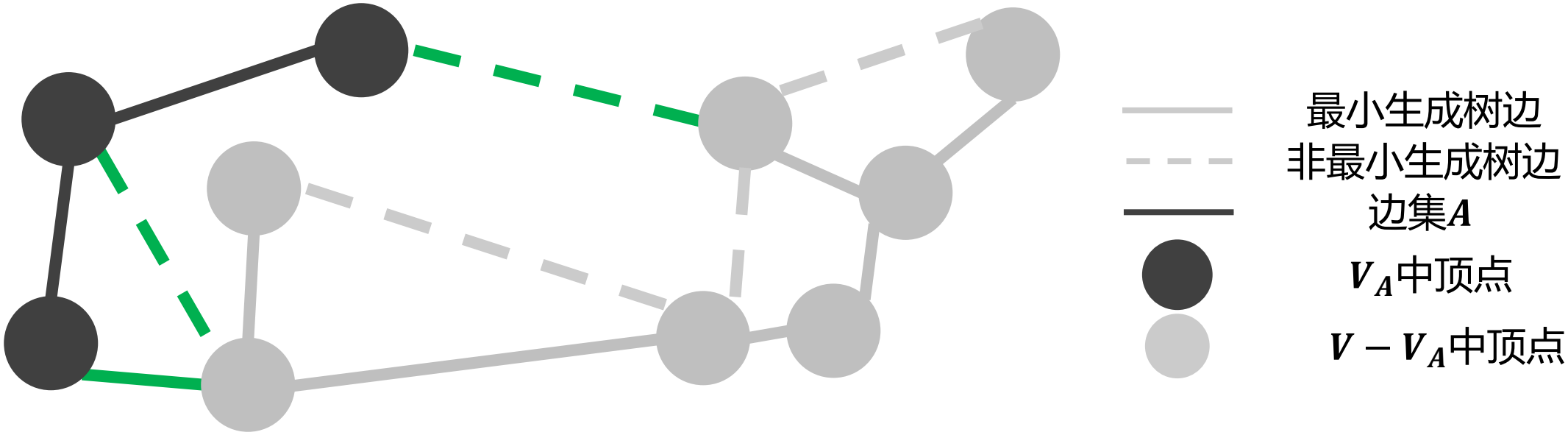
Prim算法

- 算法思想
  - 步骤1：选择任意一个顶点，作为生成树的起始顶点
  - 步骤2：保持边集A始终为一棵树，选择割$(V_A, V - V_A)$
  - 步骤3：选择横跨割$(V_A, V - V_A)$的轻边，添加到边集A中
  - 步骤4：重复步骤2和步骤3，直至覆盖所有顶点



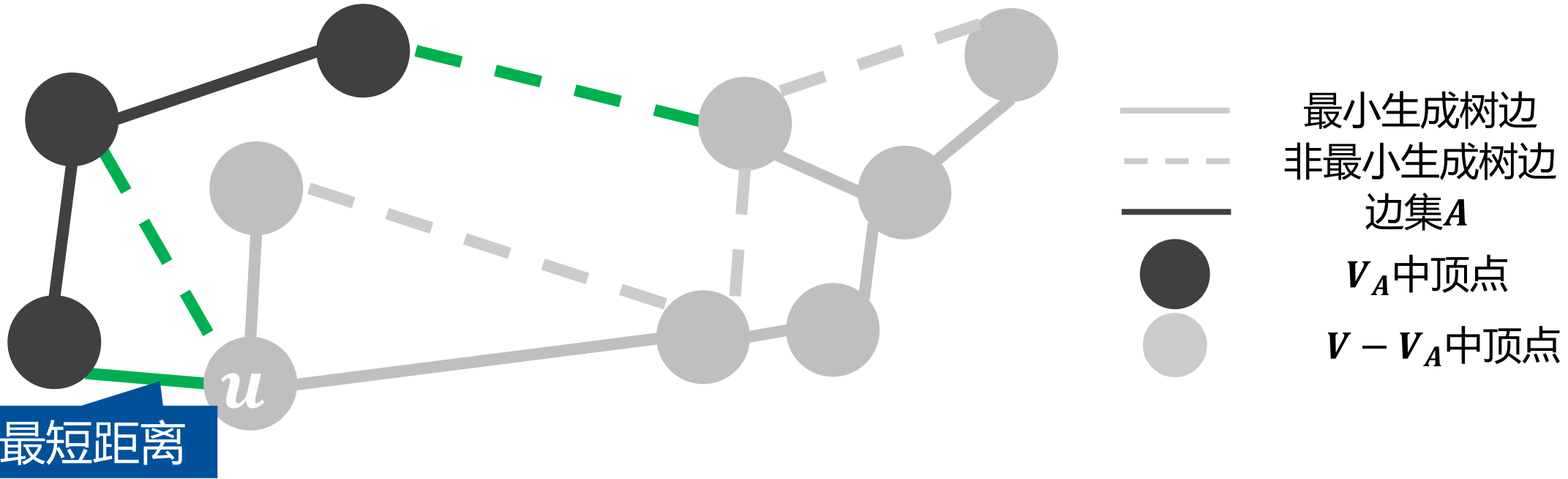
Prim算法

- 辅助数组
  - $color$表示顶点状态
    - 黑色顶点u已覆盖，$u \in V_A$
    - 白色顶点u未覆盖，$u \in V - V_A$



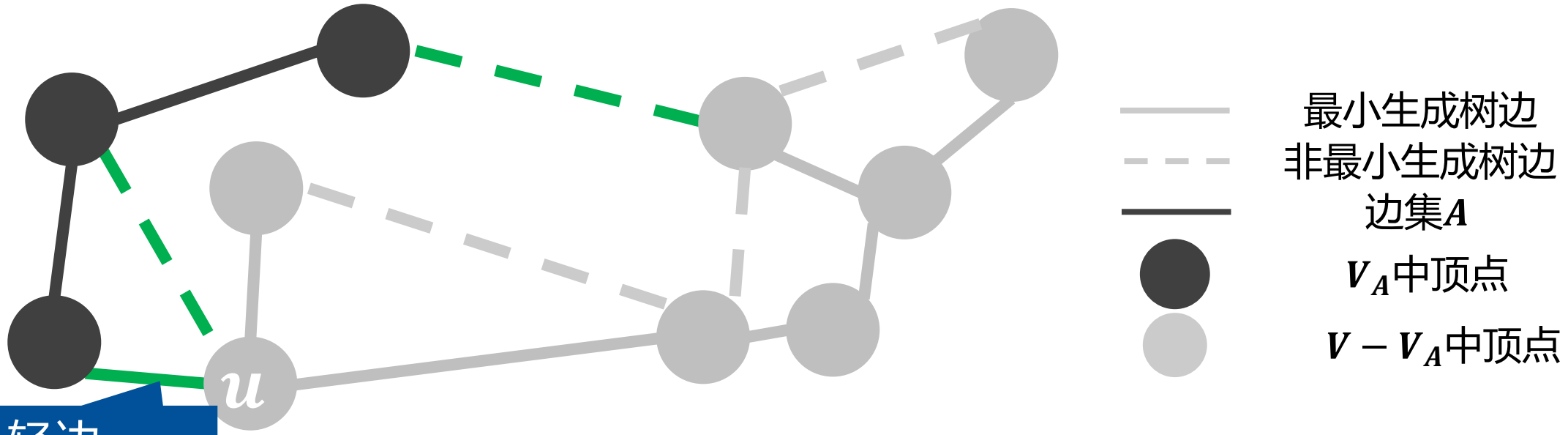
Prim算法

- 辅助数组
  - $color$表示顶点状态
    - 黑色顶点u已覆盖， $u \in V_A$
    - 白色顶点u未覆盖， $u \in V - V_A$
  - $dist$记录横跨$(V_A, V - V_A)$边的权重
    - 顶点集V_A到顶点u的最短距离，$dist[u] = \min\{w(x,u)\}, \forall x \in V_A$



Prim算法

- 辅助数组
  - $color$表示顶点状态
    - 黑色顶点u已覆盖，$u \in V_A$
    - 白色顶点u未覆盖，$u \in V - V_A$
  - $dist$记录横跨$(V_A, V - V_A)$边的权重
    - 顶点集V_A到顶点u的最短距离，$dist[u] = \min\{w(x,u)\}, \forall x \in V_A$
    - 轻边：$\min\{dist[u]\}, \forall u \in V - V_A$



Prim算法

- 辅助数组
  - $color$表示顶点状态
    - 黑色顶点u已覆盖，$u \in V_A$
    - 白色顶点u未覆盖，$u \in V - V_A$
  - $dist$记录横跨$(V_A, V - V_A)$边的权重
    - 顶点集V_A到顶点u的最短距离，$dist[u] = \min\{w(x,u)\}, \forall x \in V_A$
    - 轻边：$\min\{dist[u]\}, \forall u \in V - V_A$
  - $pred$表示前驱顶点
    - $(pred[u], u)$为最小生成树的边

提纲

问题背景

通用框架

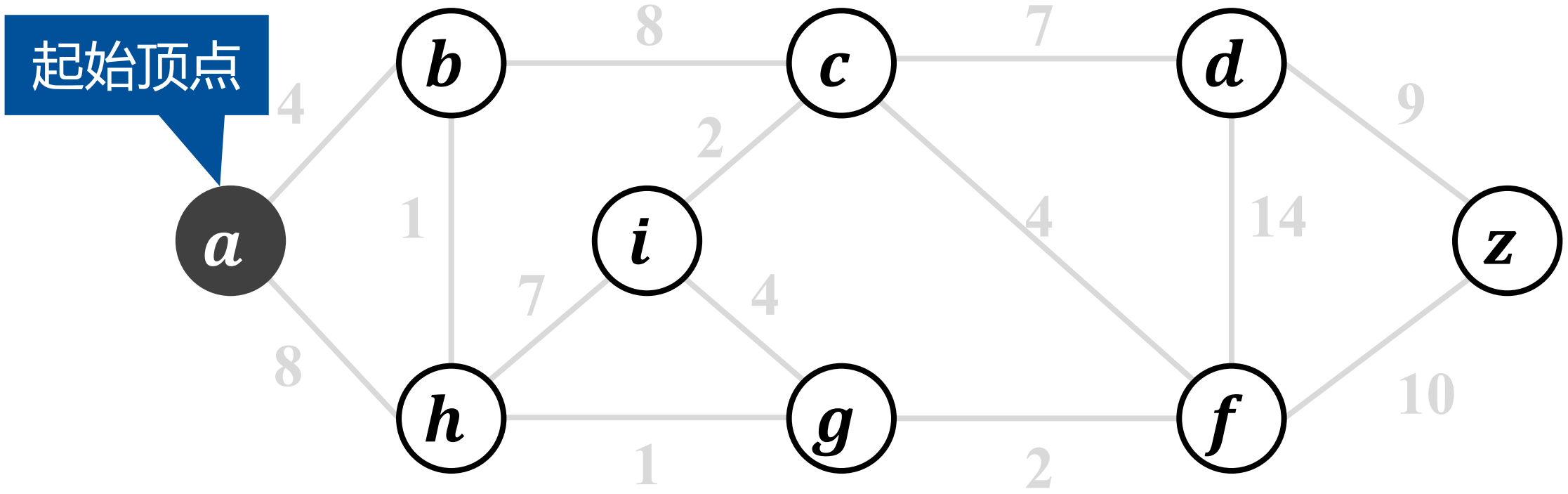
Prim算法

算法实例

算法分析

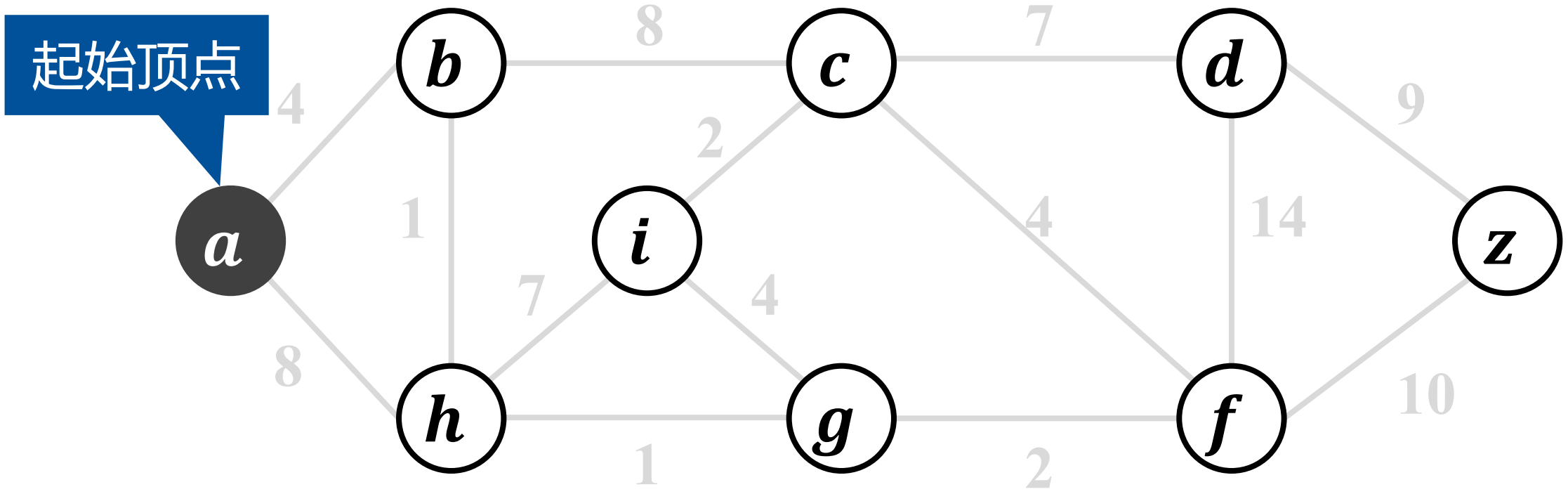
算法实例

V	a	b	c	d	f	g	h	i	z
$color$	W	W	W	W	W	W	W	W	W
$dist$	∞	∞	∞	∞	∞	∞	∞	∞	∞
$pred$	N	N	N	N	N	N	N	N	N



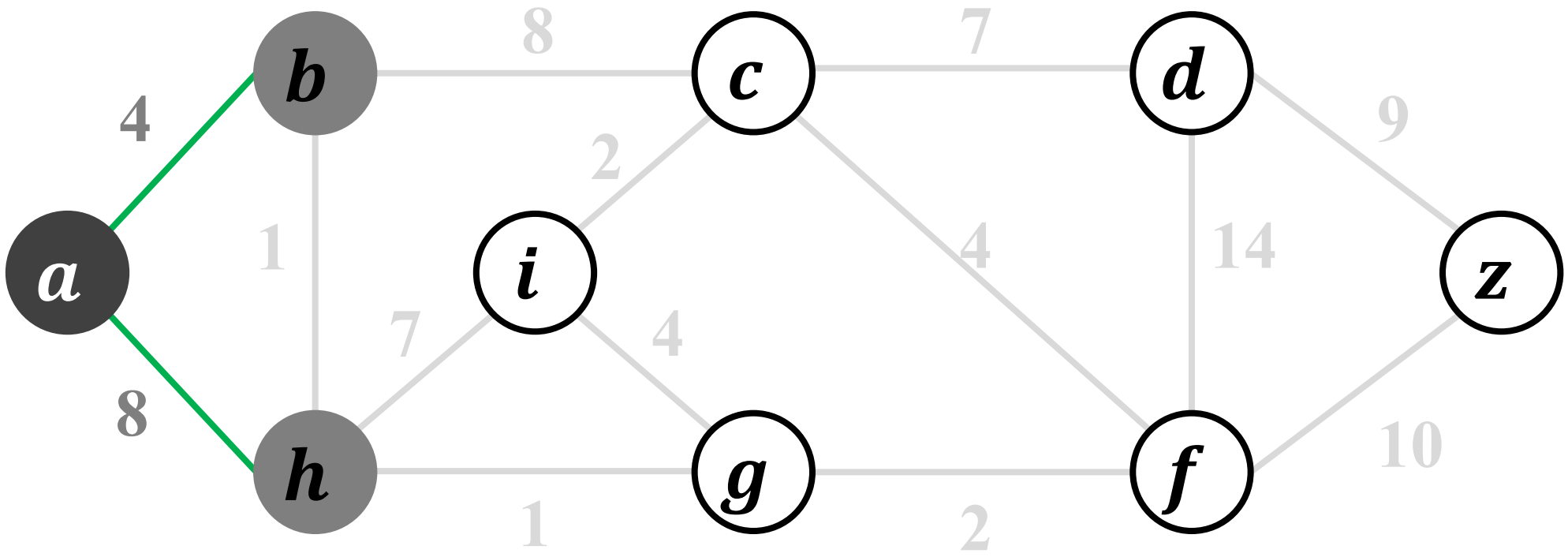
算法实例

V	a	b	c	d	f	g	h	i	z
$color$	B	W	W	W	W	W	W	W	W
$dist$	0	∞	∞	∞	∞	∞	∞	∞	∞
$pred$	N	N	N	N	N	N	N	N	N



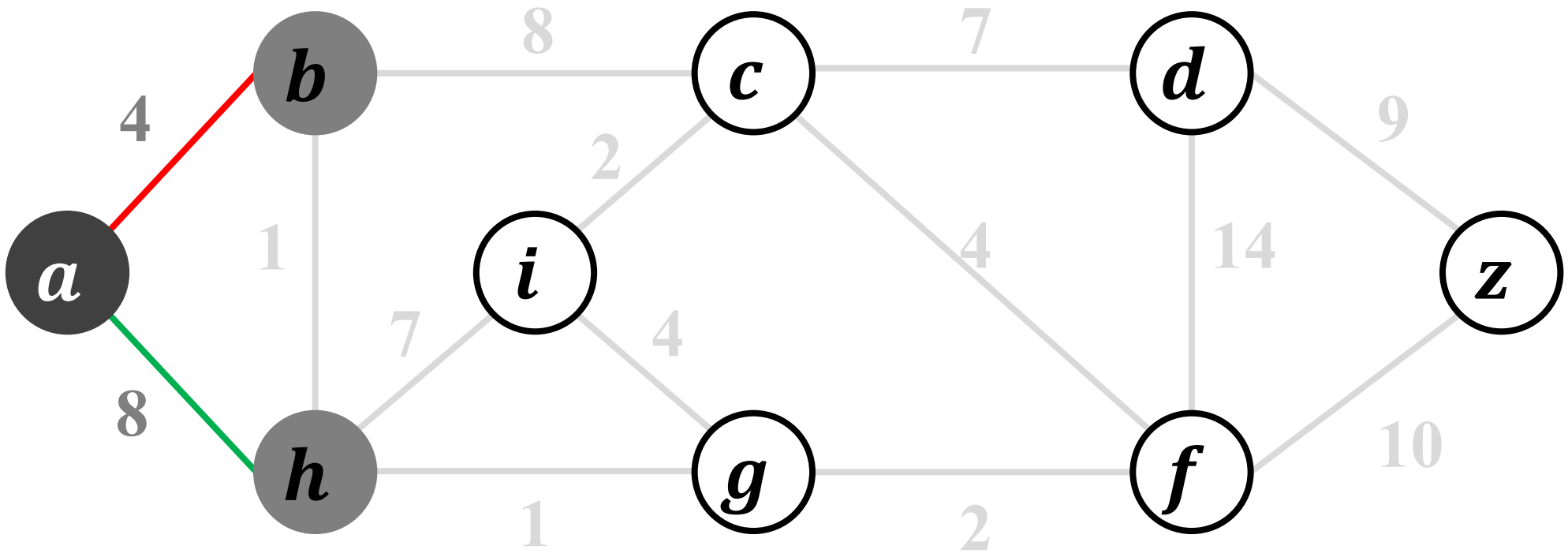
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	W	W	W	W	W	W	W	W
dist	0	4	∞	∞	∞	∞	8	∞	∞
pred	N	a	N	N	N	N	a	N	N



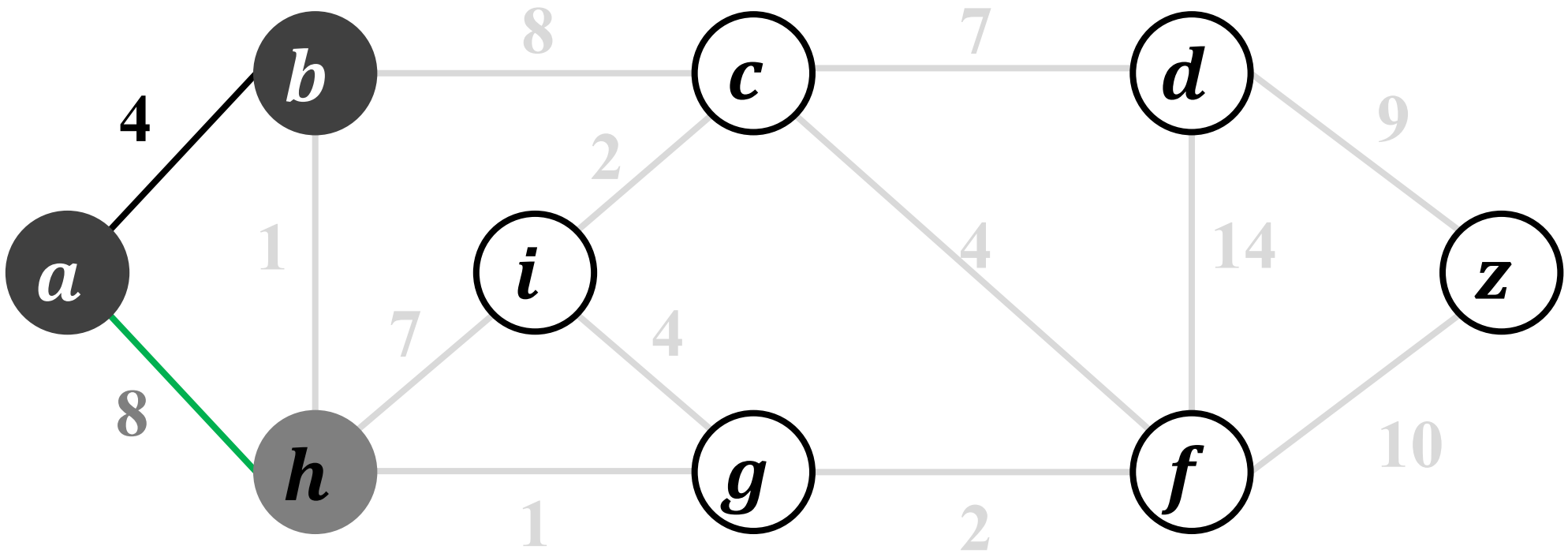
算法实例

V	*a*	*b*	*c*	*d*	*f*	*g*	*h*	*i*	*z*
color	B	W	W	W	W	W	W	W	W
dist	0	4	∞	∞	∞	∞	8	∞	∞
pred	N	*a*	N	N	N	N	*a*	N	N



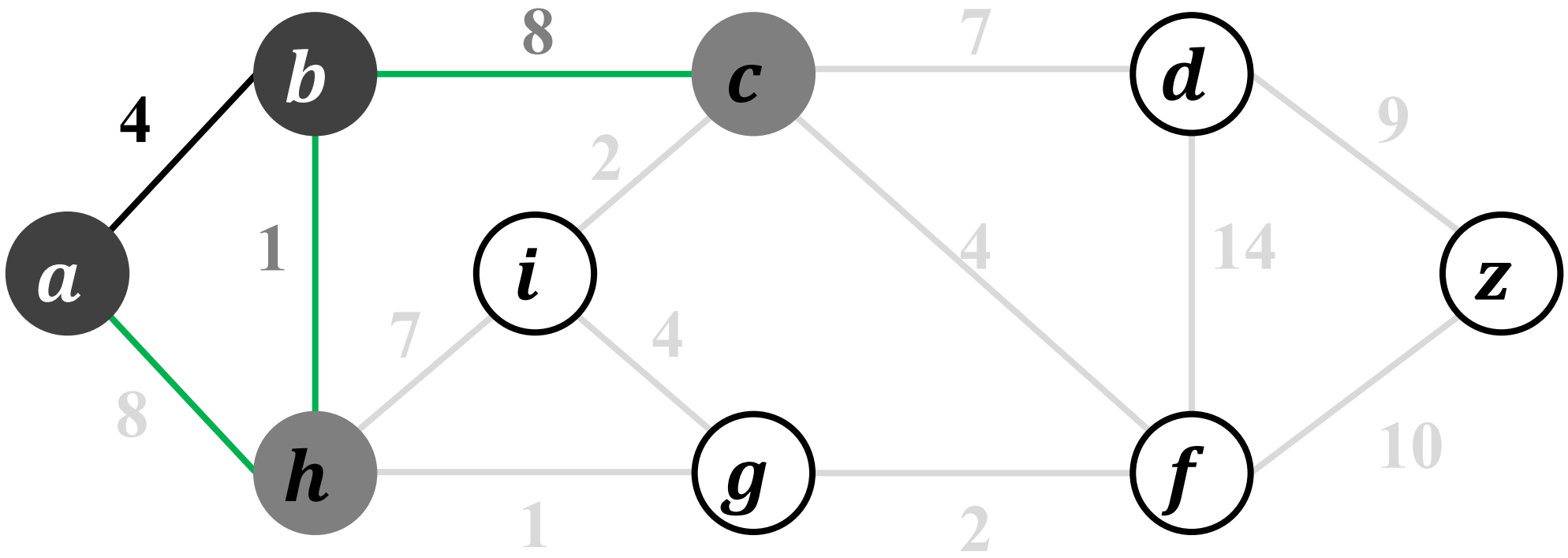
算法实例

V	*a*	*b*	*c*	*d*	*f*	*g*	*h*	*i*	*z*
color	B	B	W	W	W	W	W	W	W
dist	0	4	∞	∞	∞	∞	8	∞	∞
pred	N	*a*	N	N	N	N	*a*	N	N



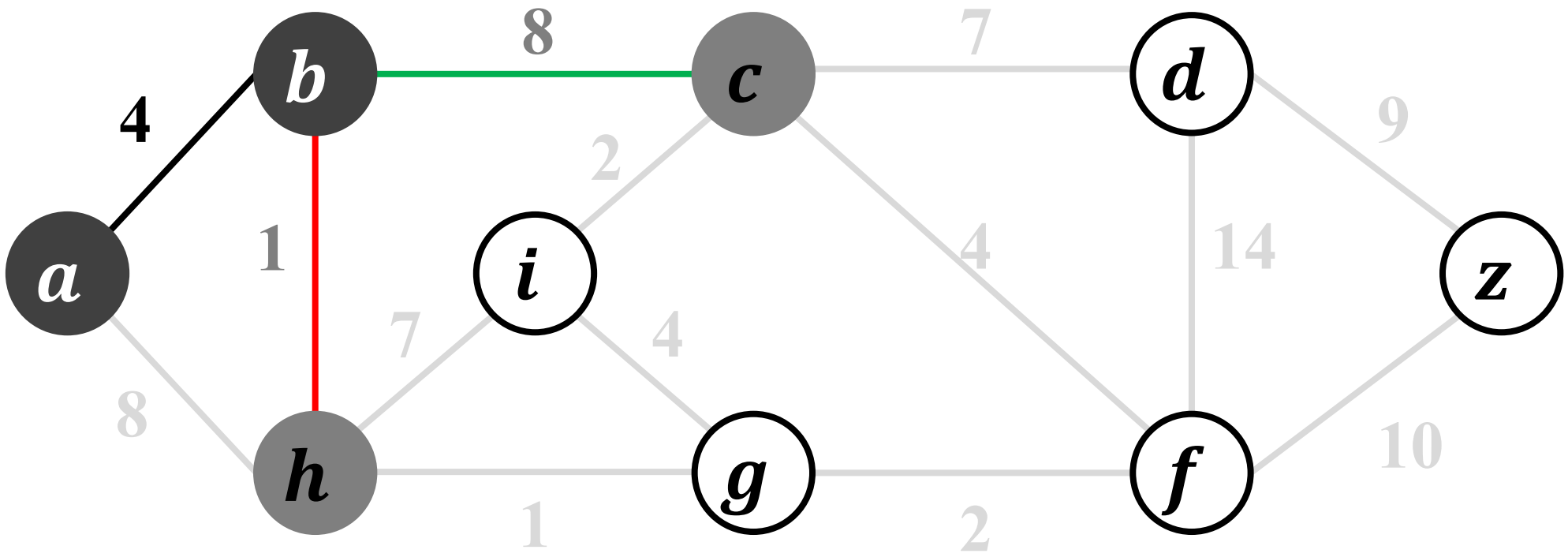
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	W	W	W	W	W	W	W
dist	0	4	8	∞	∞	∞	1	∞	∞
pred	N	*a*	*b*	N	N	N	*b*	N	N



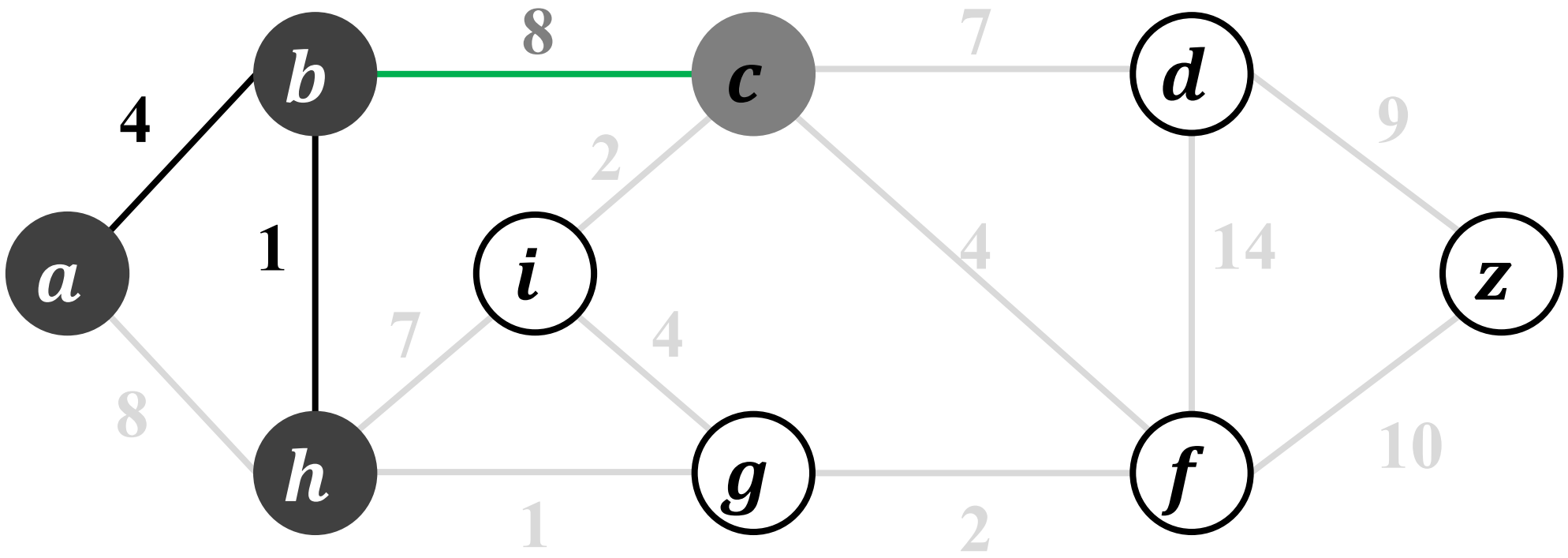
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	W	W	W	W	W	W	W
dist	0	4	8	∞	∞	∞	1	∞	∞
pred	N	a	b	N	N	N	b	N	N



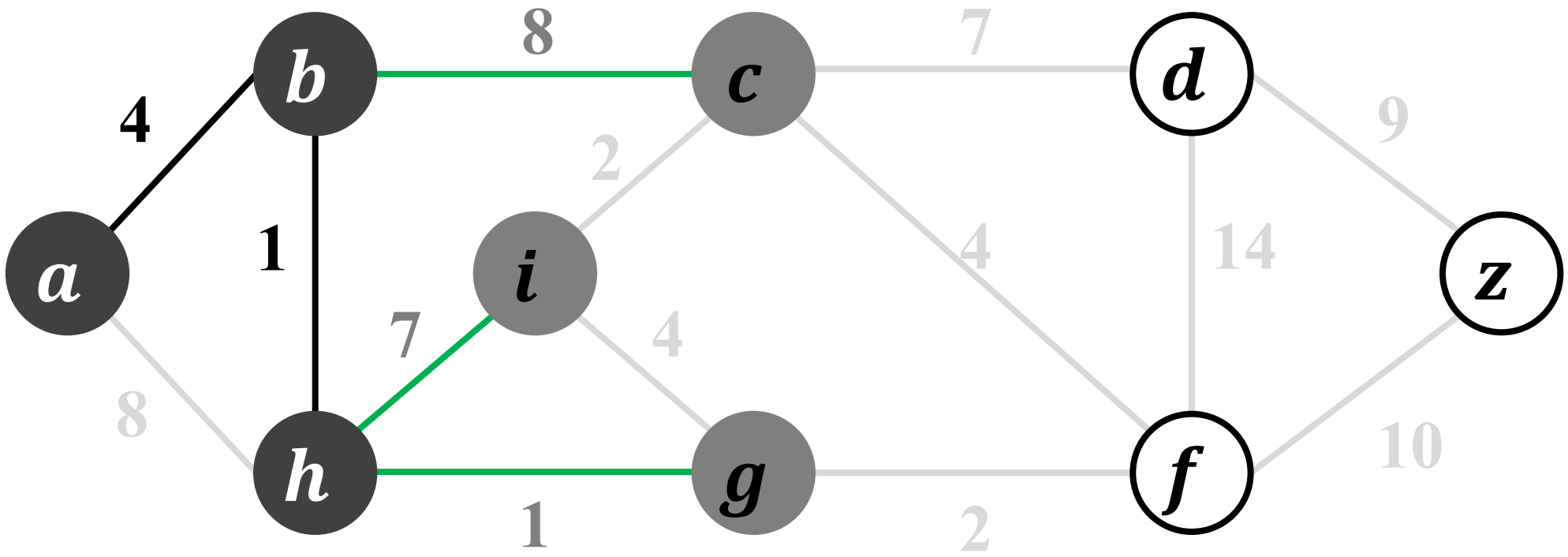
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	W	W	W	W	B	W	W
dist	0	4	8	∞	∞	∞	1	∞	∞
pred	N	*a*	*b*	N	N	N	*b*	N	N



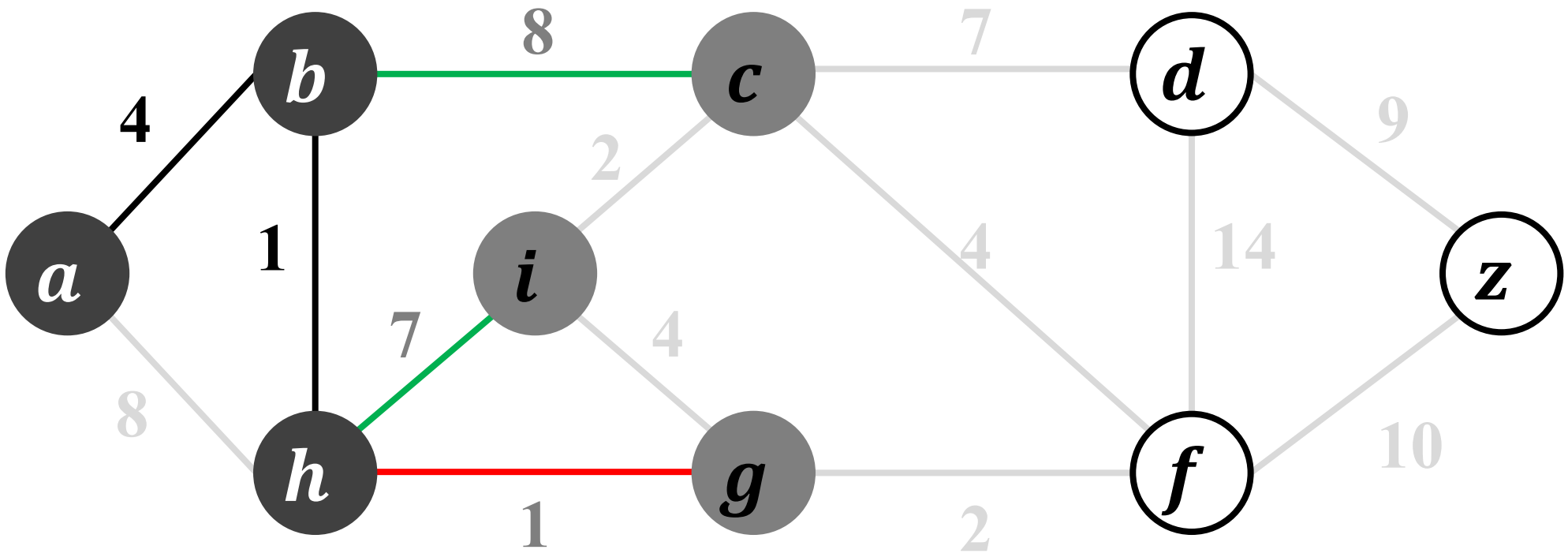
算法实例

V	*a*	*b*	*c*	*d*	*f*	*g*	*h*	*i*	*z*
color	B	B	W	W	W	W	B	W	W
dist	0	4	8	∞	∞	1	1	7	∞
pred	N	*a*	*b*	N	N	*h*	*b*	*h*	N



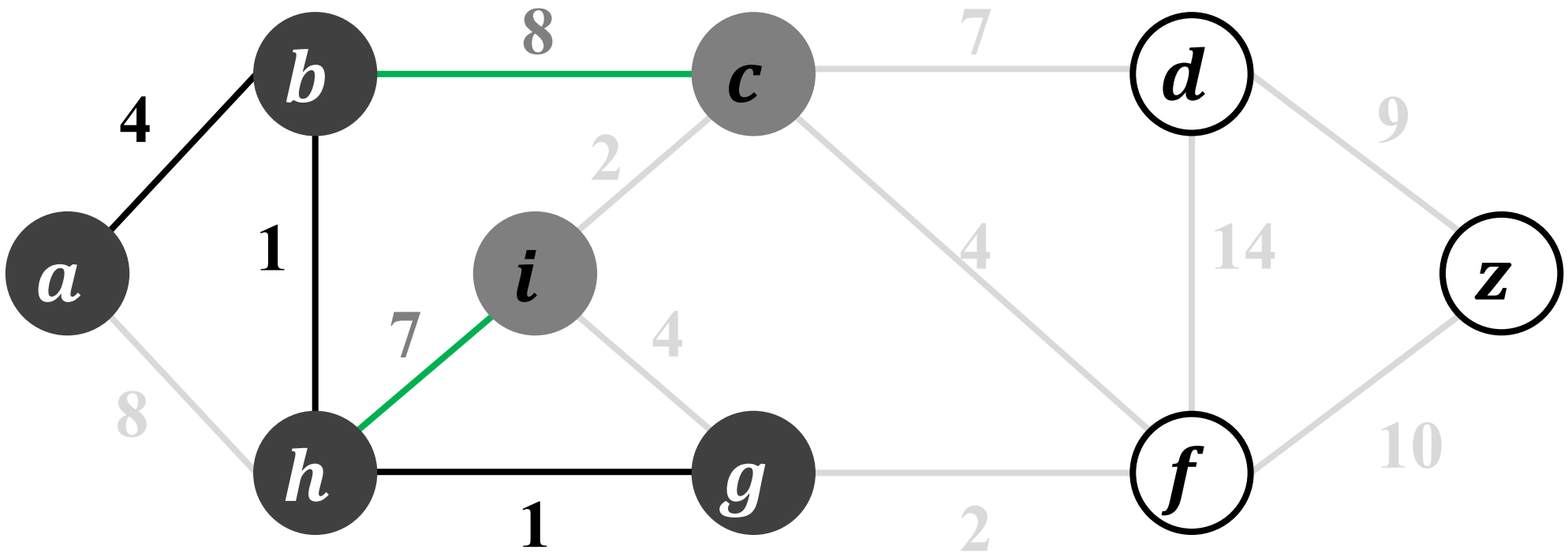
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	W	W	W	W	B	W	W
dist	0	4	8	∞	∞	1	1	7	∞
pred	N	a	b	N	N	h	b	h	N



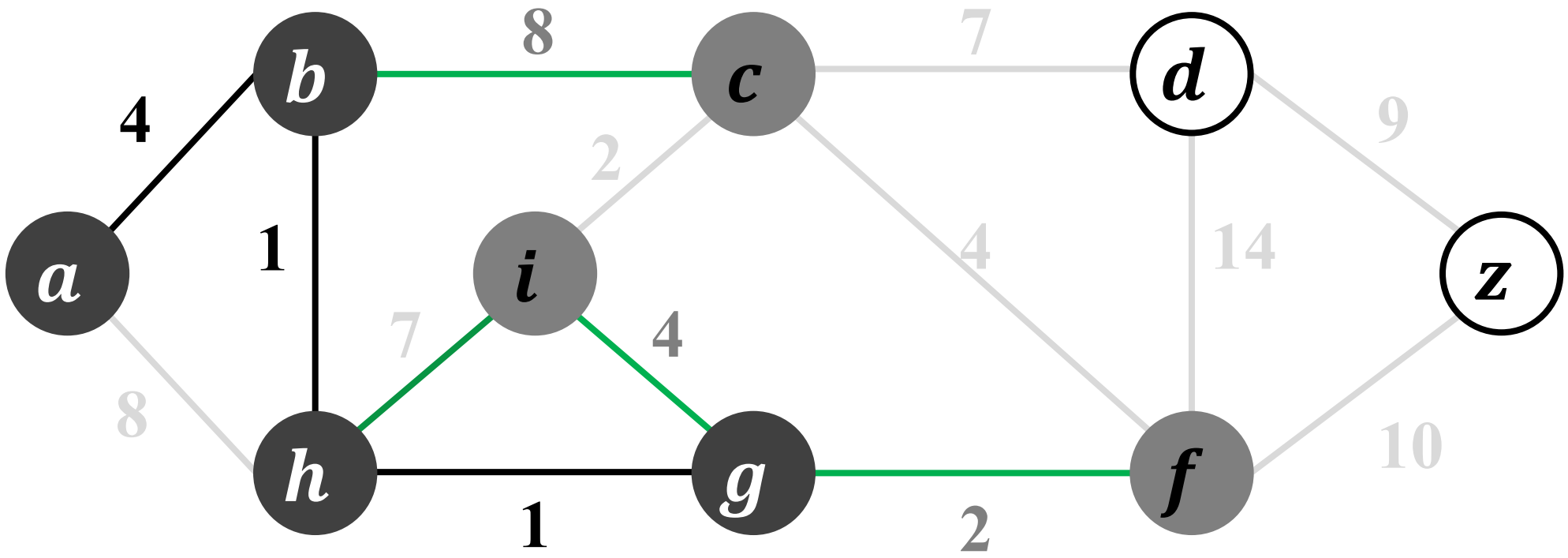
算法实例

V	a	b	c	d	f	g	h	i	z
$color$	B	B	W	W	W	B	B	W	W
$dist$	0	4	8	∞	∞	1	1	7	∞
$pred$	N	a	b	N	N	h	b	h	N



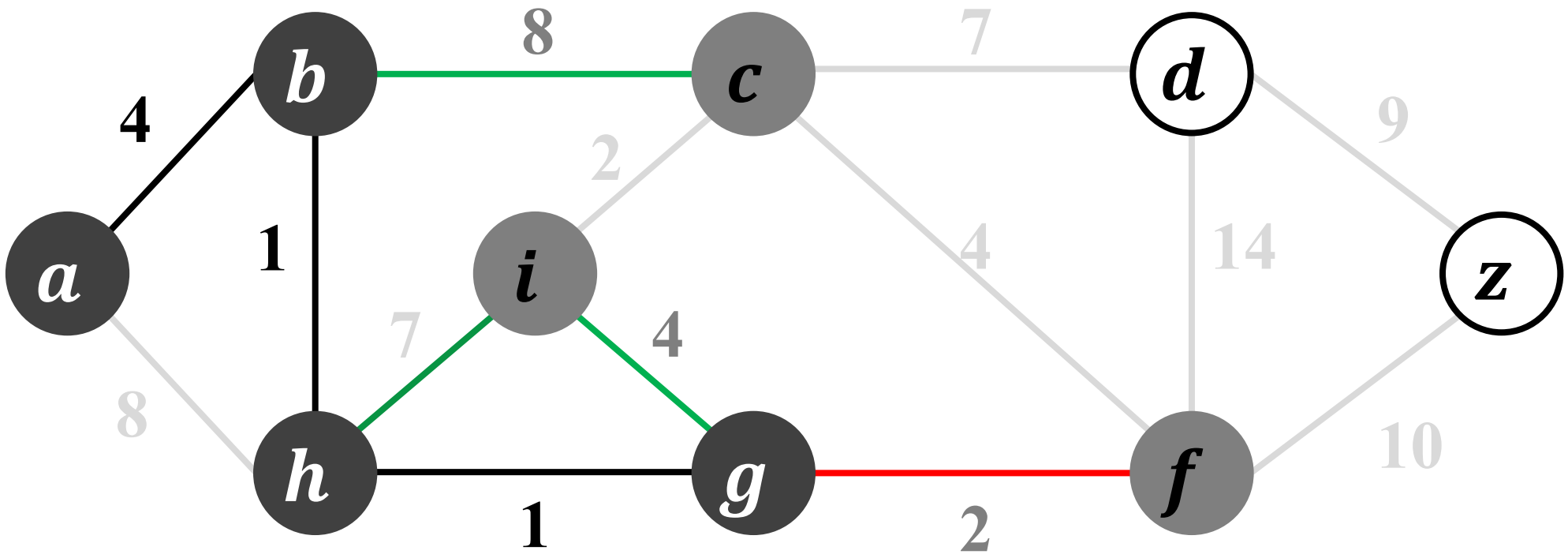
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	W	W	W	B	B	W	W
dist	0	4	8	∞	2	1	1	4	∞
pred	N	a	b	N	g	h	b	g	N



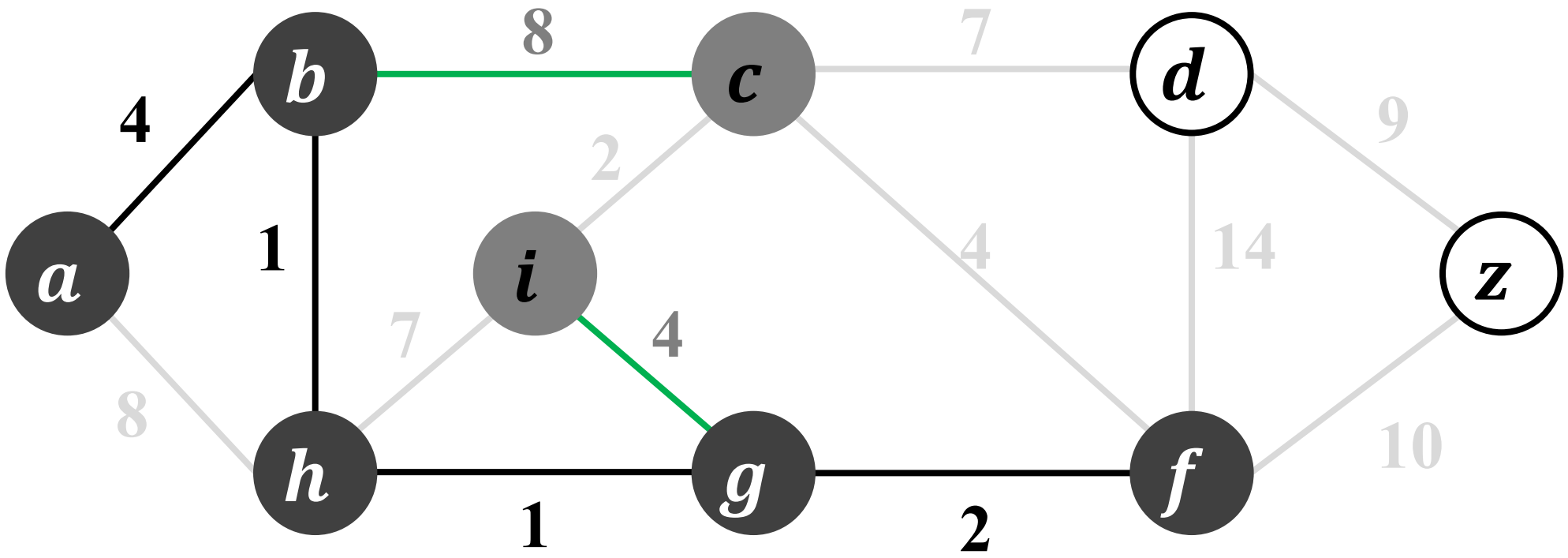
算法实例

V	*a*	*b*	*c*	*d*	*f*	*g*	*h*	*i*	*z*
color	B	B	W	W	W	B	B	W	W
dist	0	4	8	∞	2	1	1	4	∞
pred	N	*a*	*b*	N	*g*	*h*	*b*	*g*	N



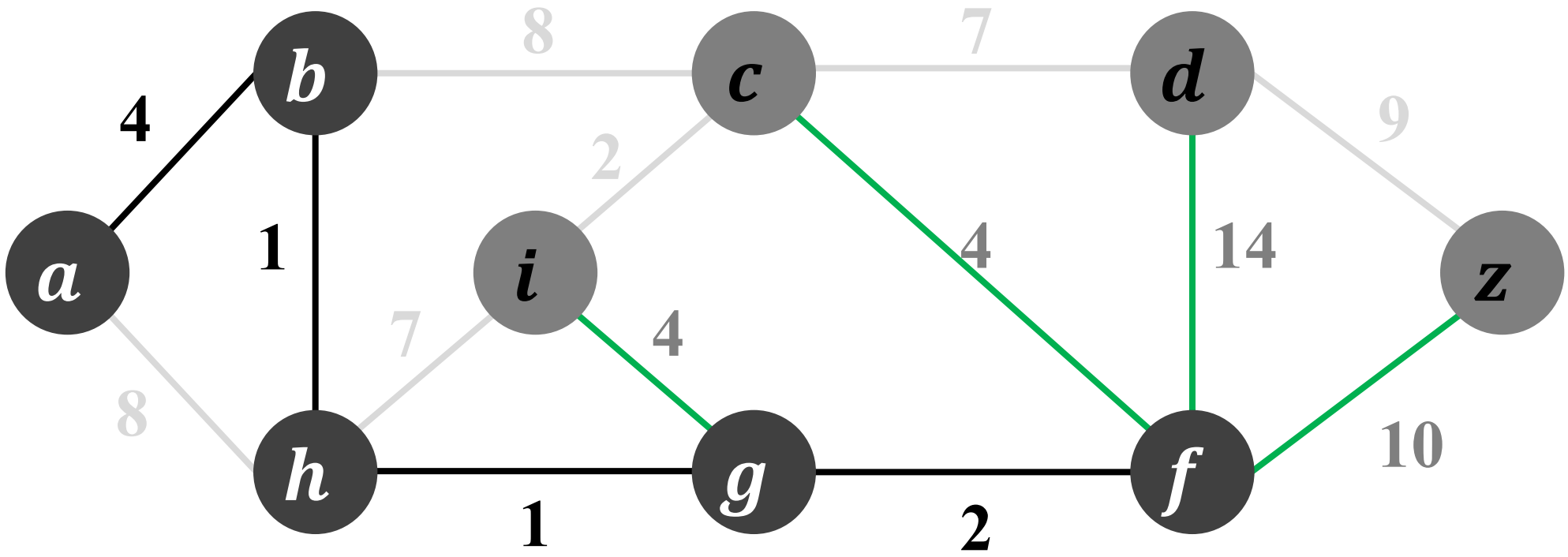
算法实例

V	*a*	*b*	*c*	*d*	*f*	*g*	*h*	*i*	*z*
color	B	B	W	W	B	B	B	W	W
dist	0	4	8	∞	2	1	1	4	∞
pred	N	*a*	*b*	N	*g*	*h*	*b*	*g*	N



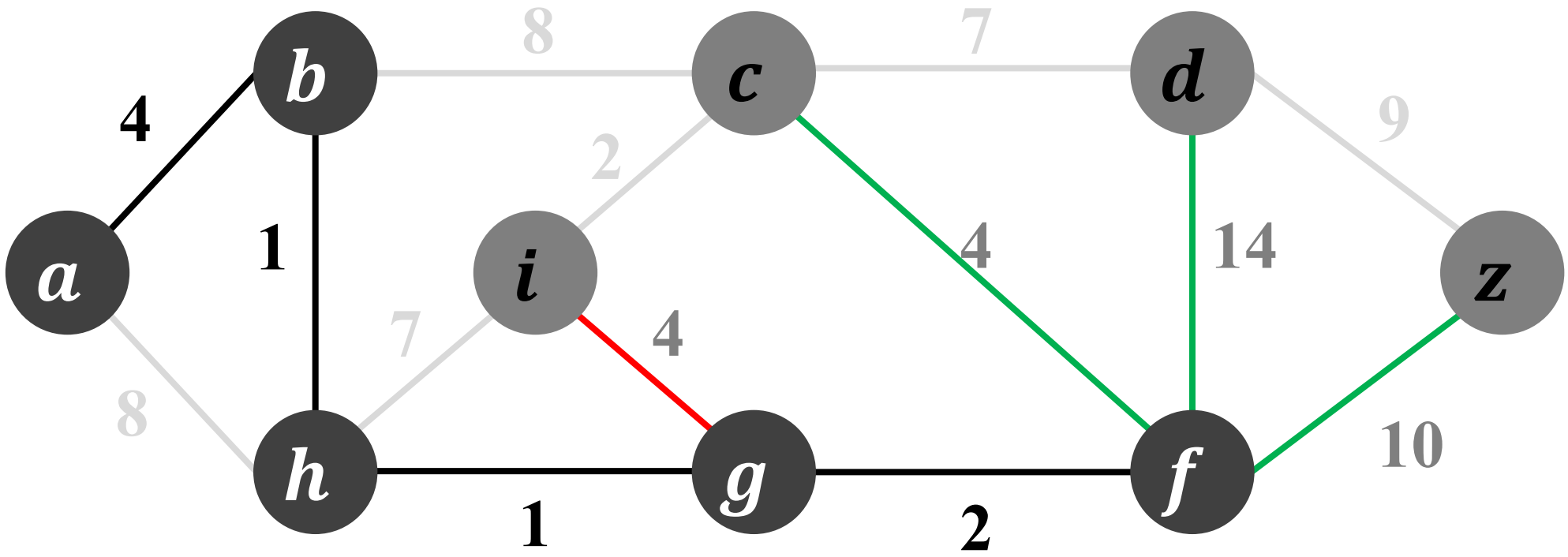
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	W	W	B	B	B	W	W
dist	0	4	4	14	2	1	1	4	10
pred	N	a	f	f	g	h	b	g	f



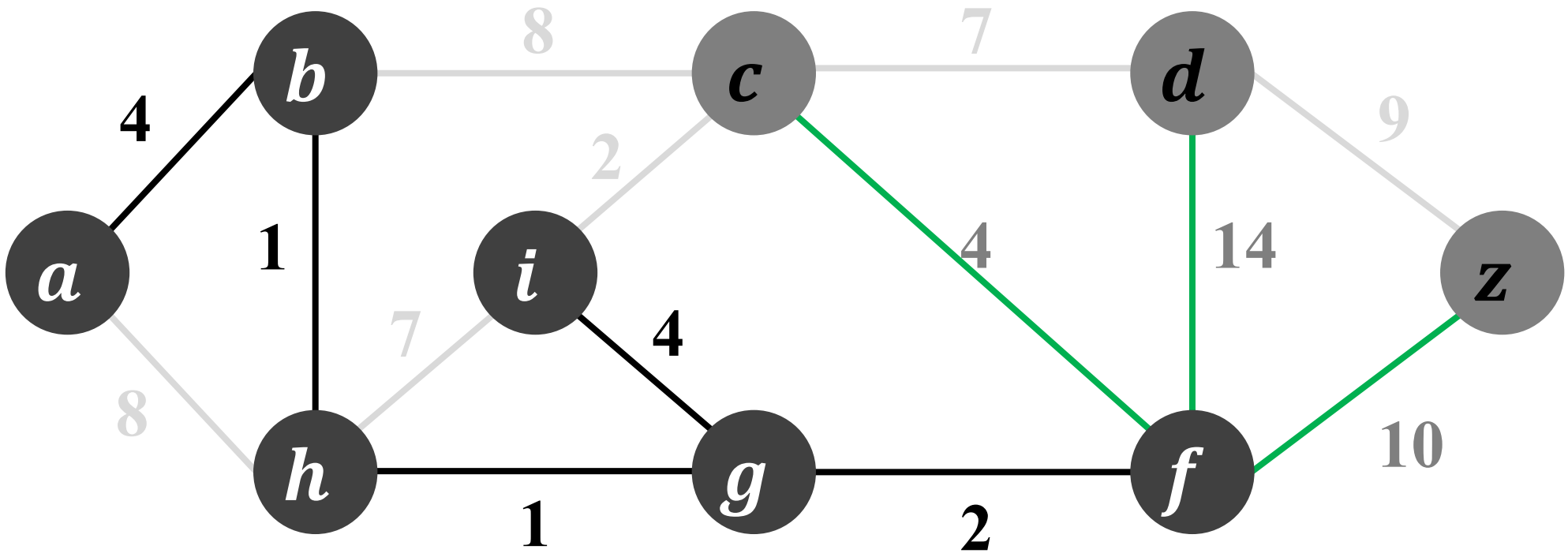
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	W	W	B	B	B	W	W
dist	0	4	4	14	2	1	1	4	10
pred	N	a	f	f	g	h	b	g	f



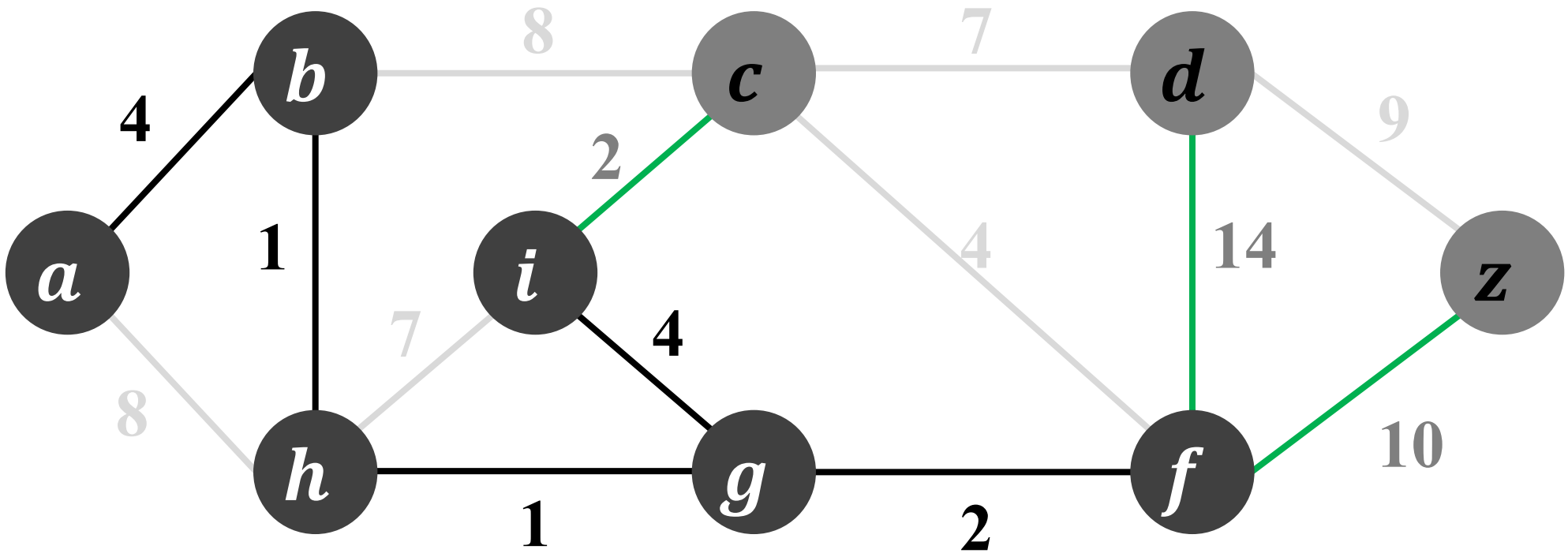
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	W	W	B	B	B	B	W
dist	0	4	4	14	2	1	1	4	10
pred	N	a	f	f	g	h	b	g	f



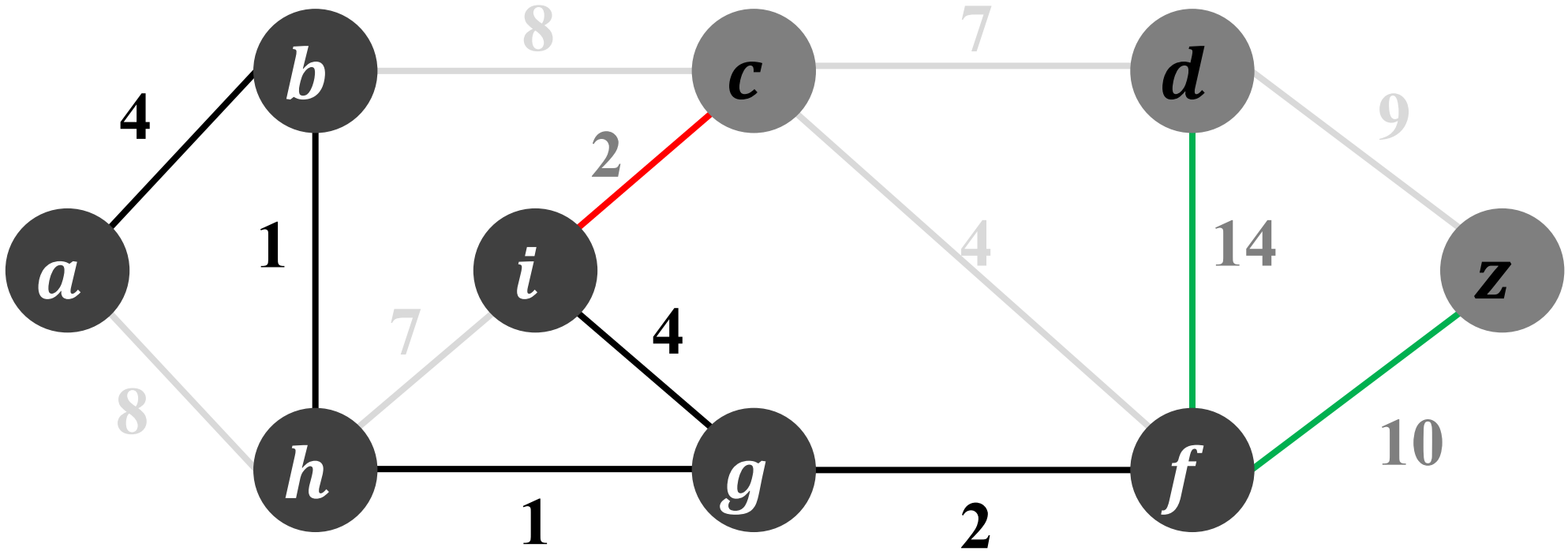
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	W	W	B	B	B	B	W
dist	0	4	2	14	2	1	1	4	10
pred	N	a	i	f	g	h	b	g	f



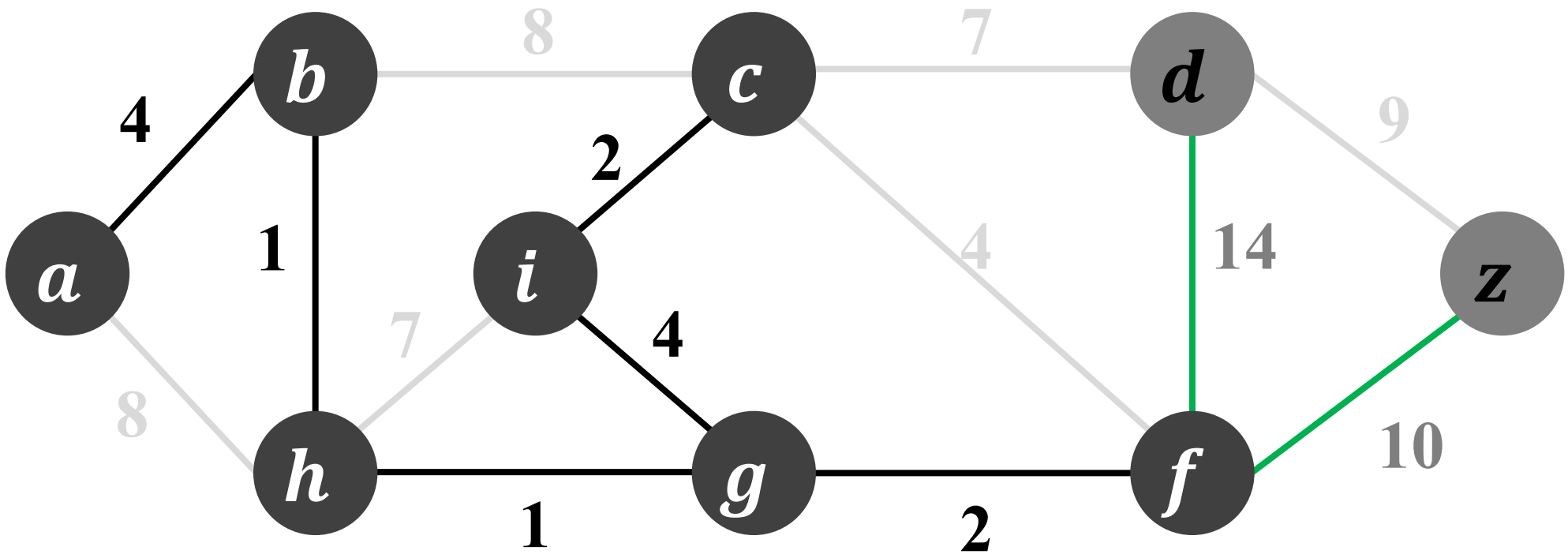
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	W	W	B	B	B	B	W
dist	0	4	2	14	2	1	1	4	10
pred	N	a	i	f	g	h	b	g	f



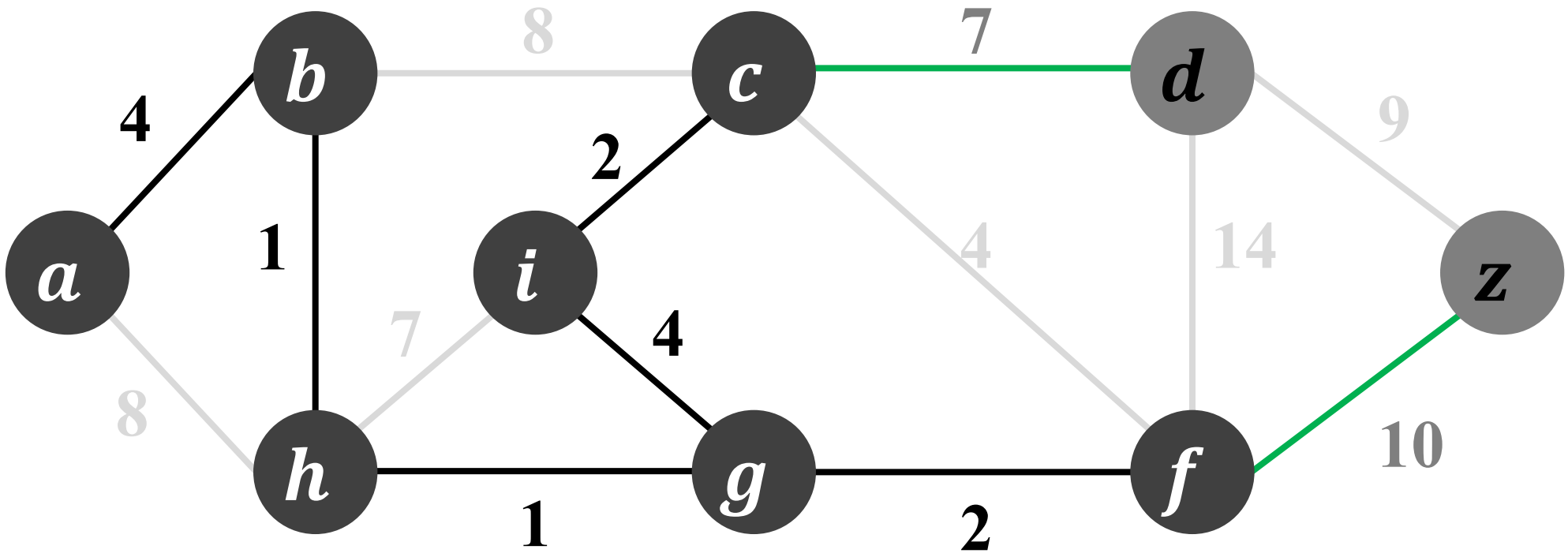
算法实例

V	*a*	*b*	*c*	*d*	*f*	*g*	*h*	*i*	*z*
color	B	B	B	W	B	B	B	B	W
dist	0	4	2	14	2	1	1	4	10
pred	N	*a*	*i*	*f*	*g*	*h*	*b*	*g*	*f*



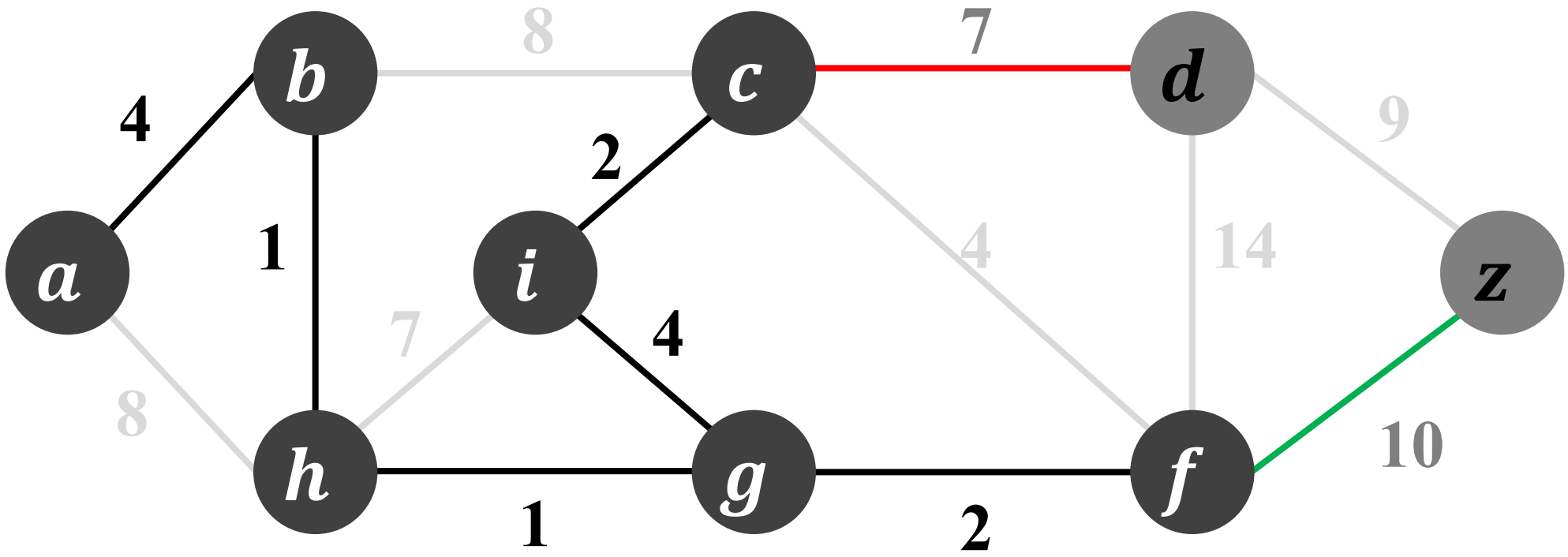
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	B	W	B	B	B	B	W
dist	0	4	2	7	2	1	1	4	10
pred	N	a	i	c	g	h	b	g	f



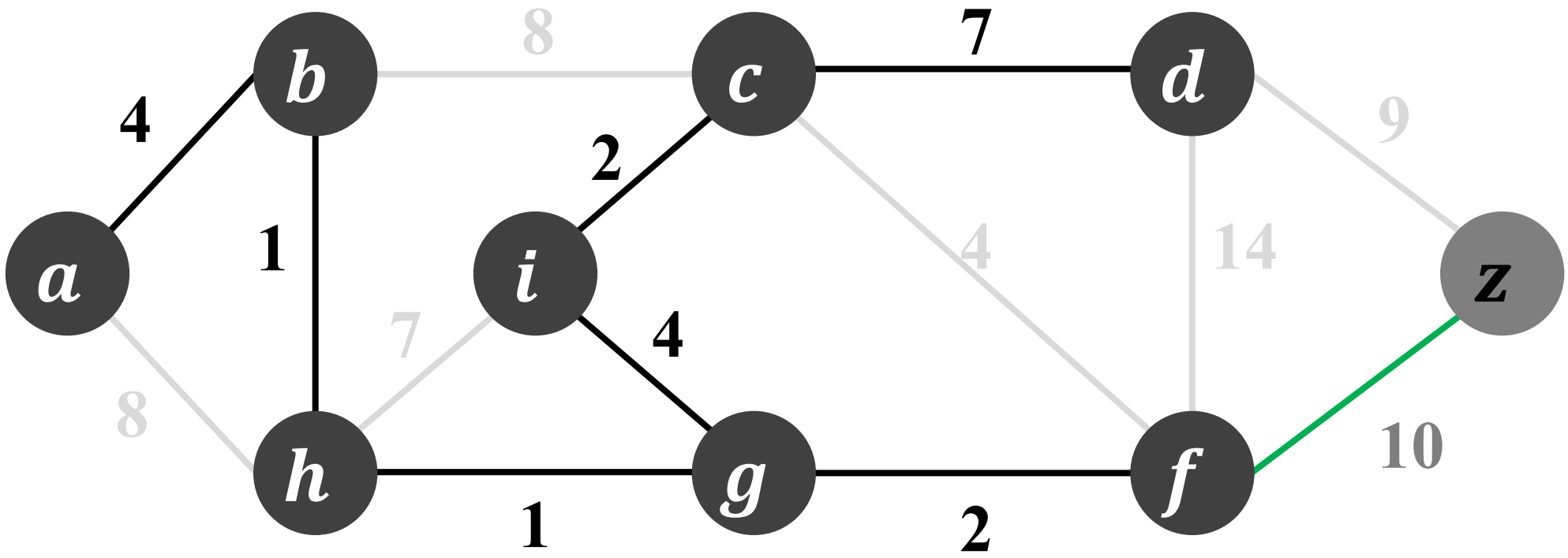
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	B	W	B	B	B	B	W
dist	0	4	2	7	2	1	1	4	10
pred	N	a	i	c	g	h	b	g	f



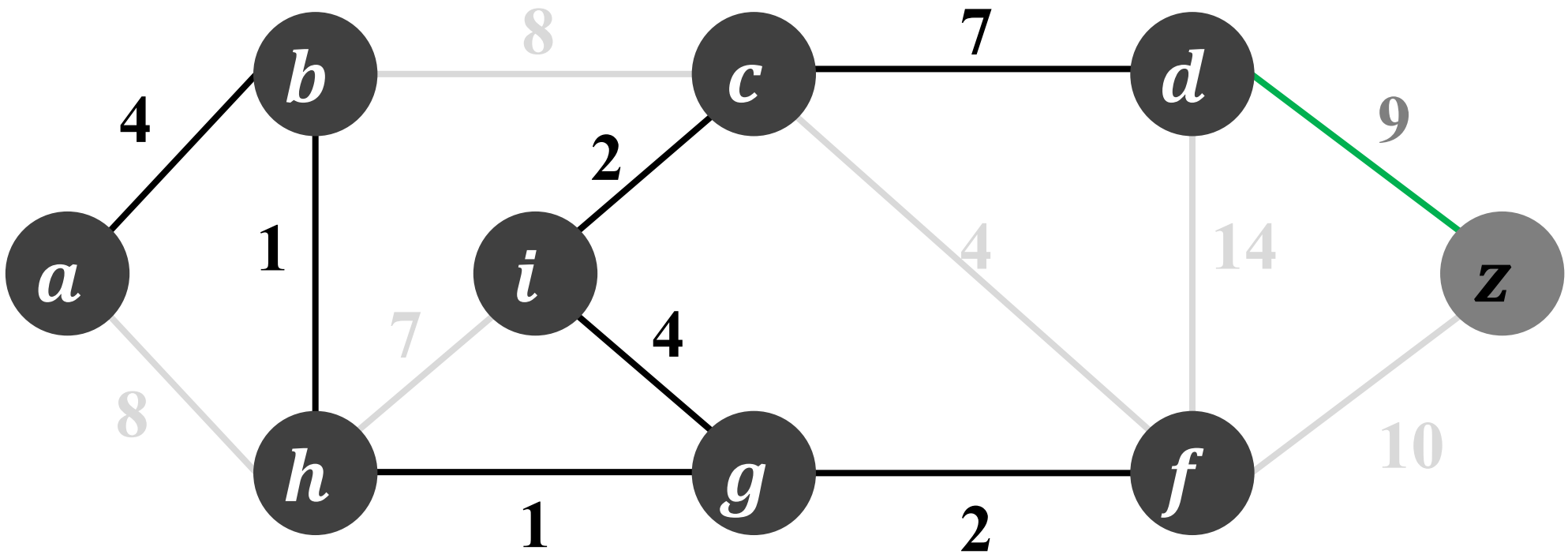
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	B	B	B	B	B	B	W
dist	0	4	2	7	2	1	1	4	10
pred	N	a	i	c	g	h	b	g	f



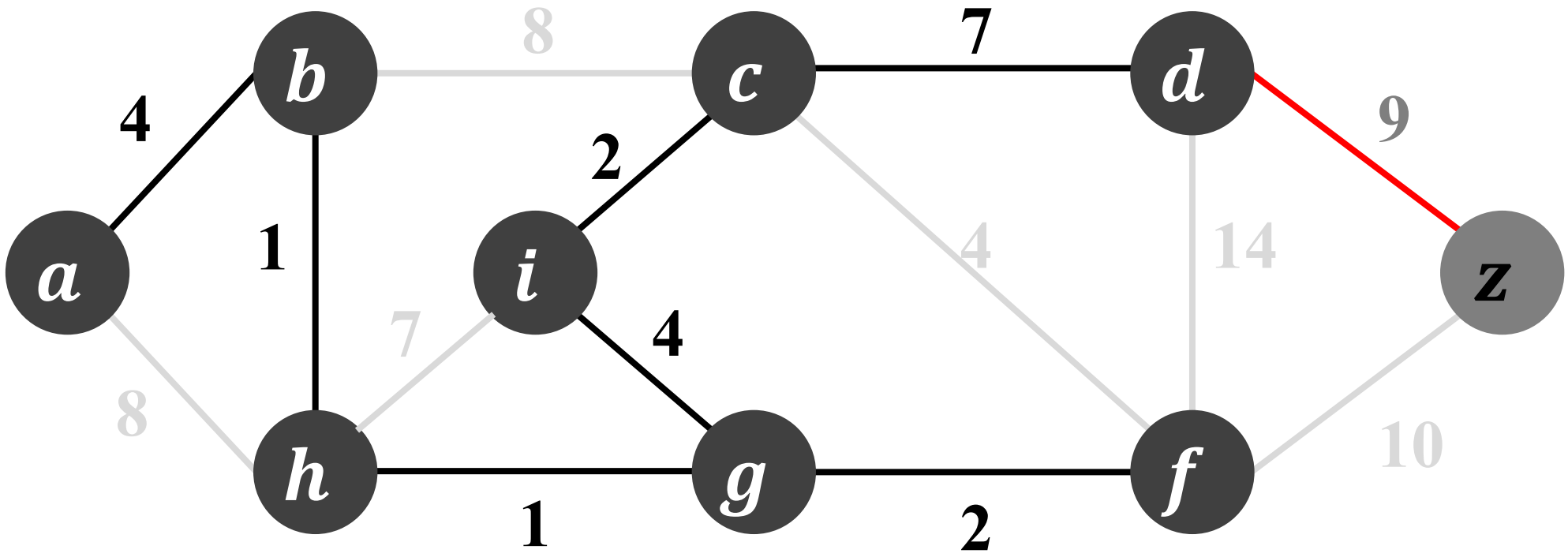
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	B	B	B	B	B	B	W
dist	0	4	2	7	2	1	1	4	9
pred	N	a	i	c	g	h	b	g	d



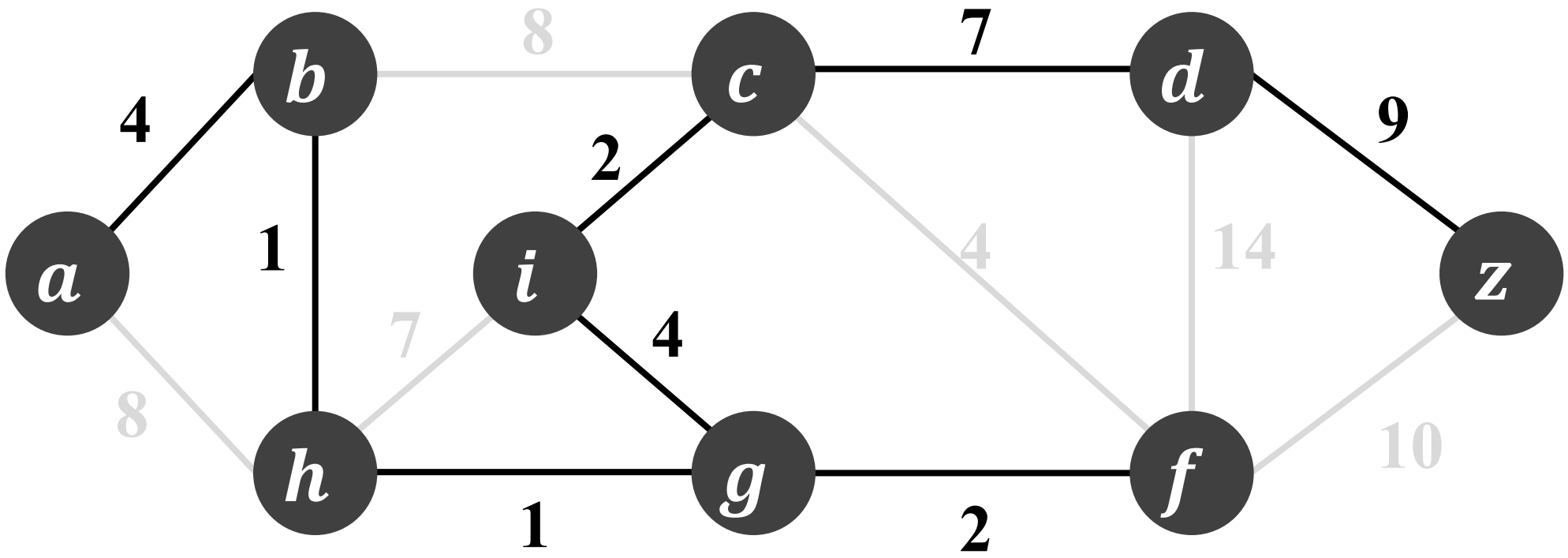
算法实例

V	*a*	*b*	*c*	*d*	*f*	*g*	*h*	*i*	*z*
color	B	B	B	B	B	B	B	B	W
dist	0	4	2	7	2	1	1	4	9
pred	N	*a*	*i*	*c*	*g*	*h*	*b*	*g*	*d*



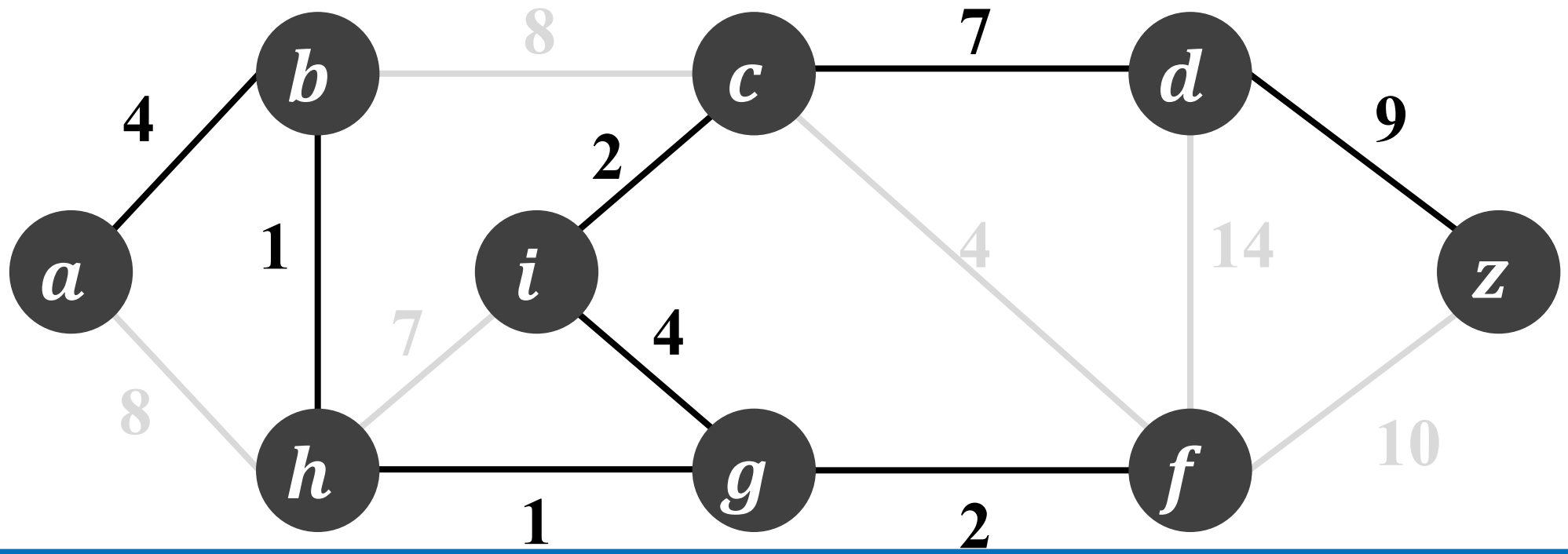
算法实例

V	a	b	c	d	f	g	h	i	z
color	B	B	B	B	B	B	B	B	B
dist	0	4	2	7	2	1	1	4	9
pred	N	a	i	c	g	h	b	g	d



算法实例

V	a	b	c	d	f	g	h	i	z
$color$	B	B	B	B	B	B	B	B	B
$dist$	0	4	2	7	2	1	1	4	9
$pred$	N	a	i	c	g	h	b	g	d



$$W(T) = 0 + 4 + 2 + 7 + 2 + 1 + 1 + 4 + 9 = 30$$

提纲

问题背景

通用框架

Prim算法

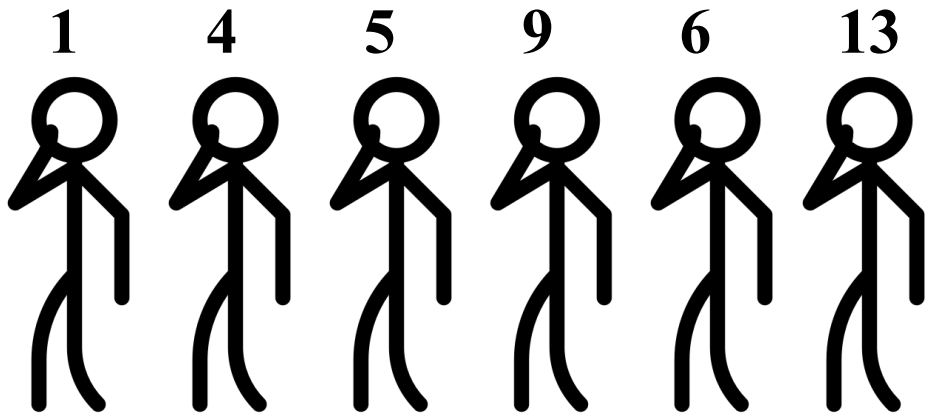
算法实例

算法分析

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列

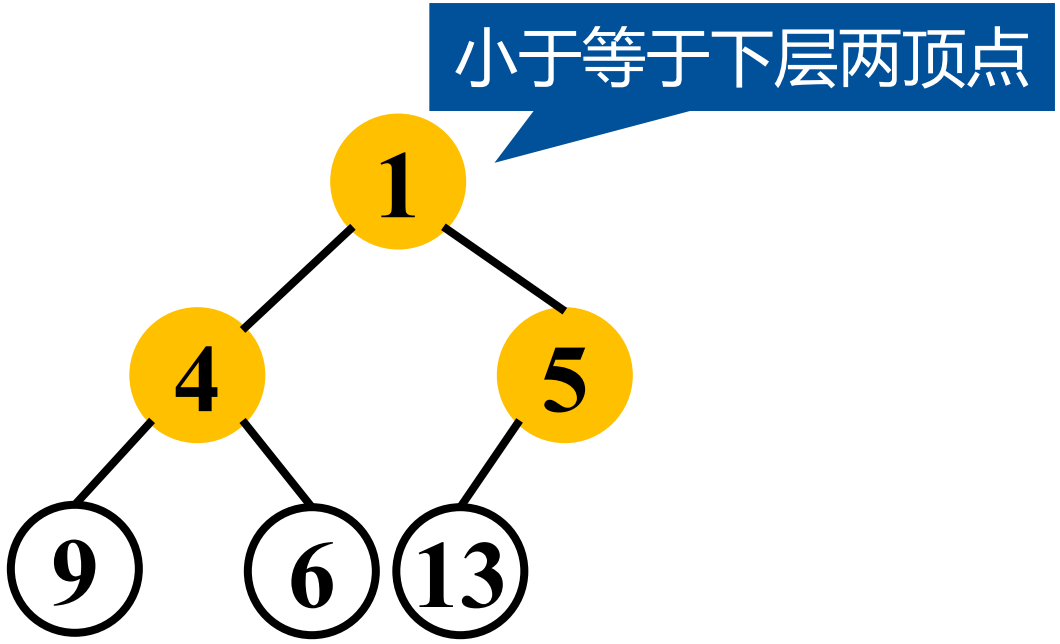


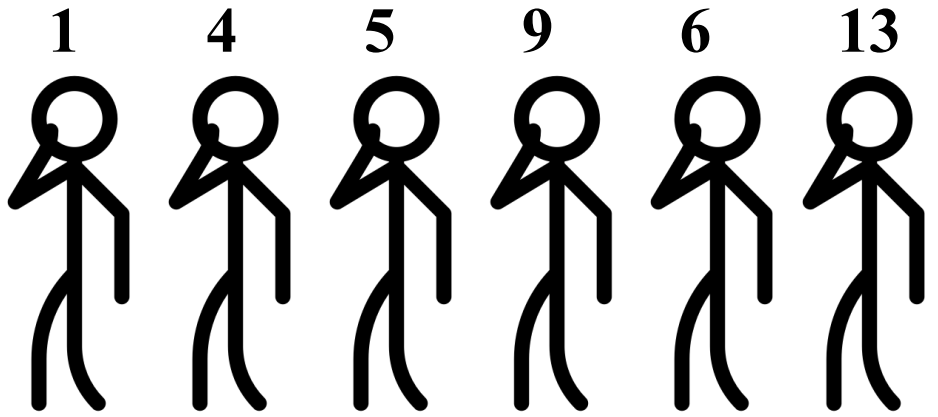
优先队列Q

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列



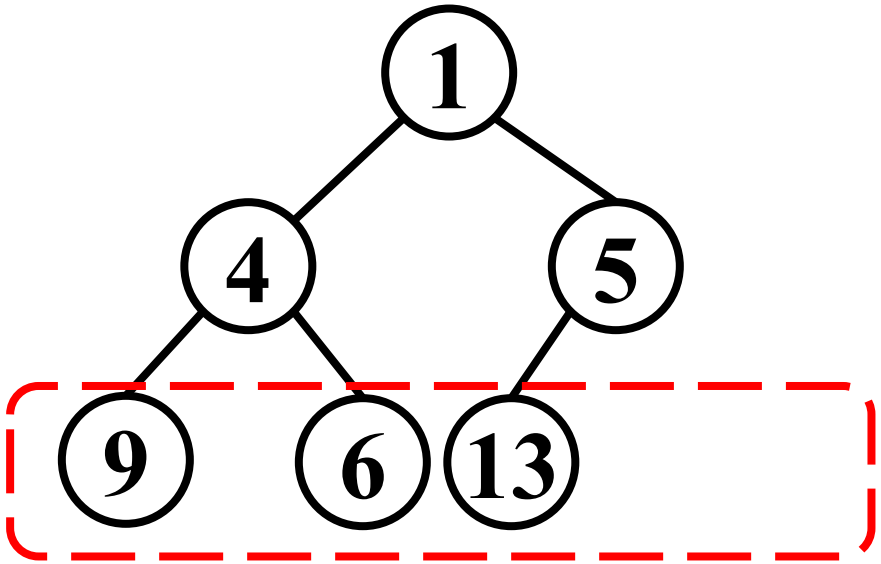


优先队列Q

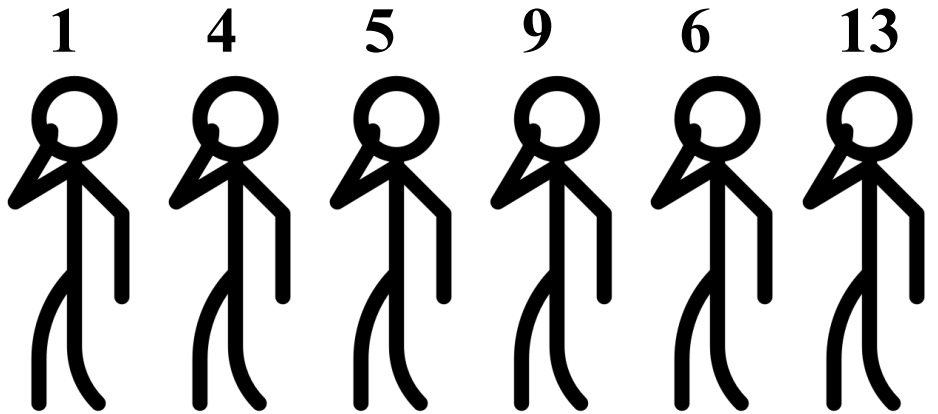
数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列



除最底层，第h层有2^{h-1}个顶点

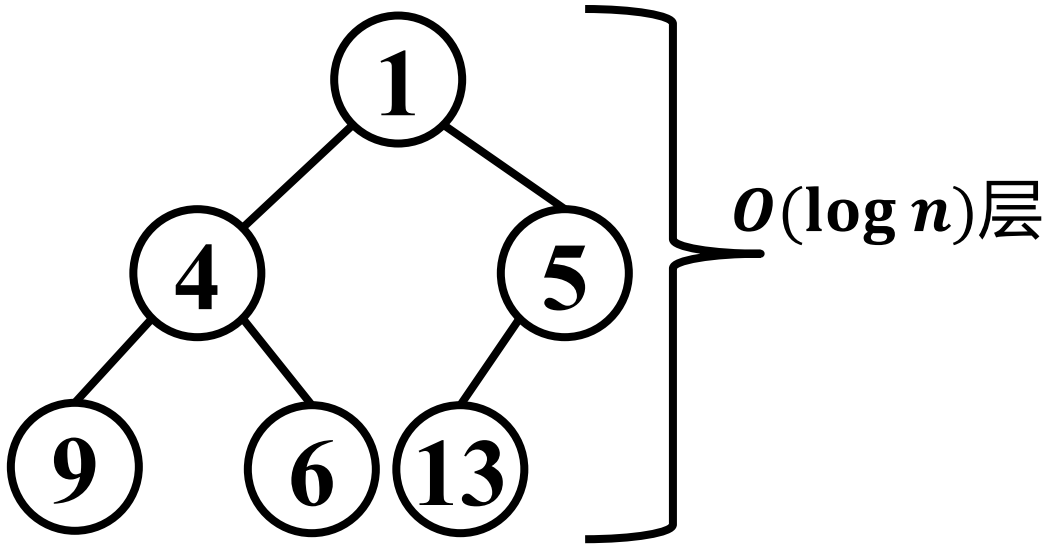


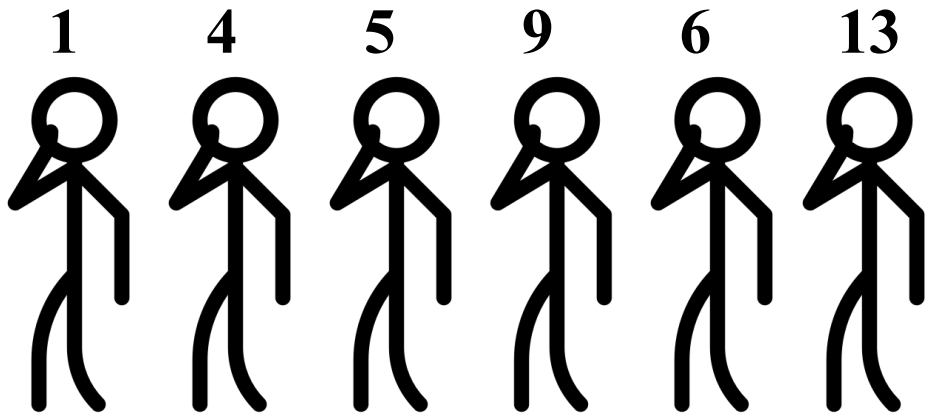
优先队列Q

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列





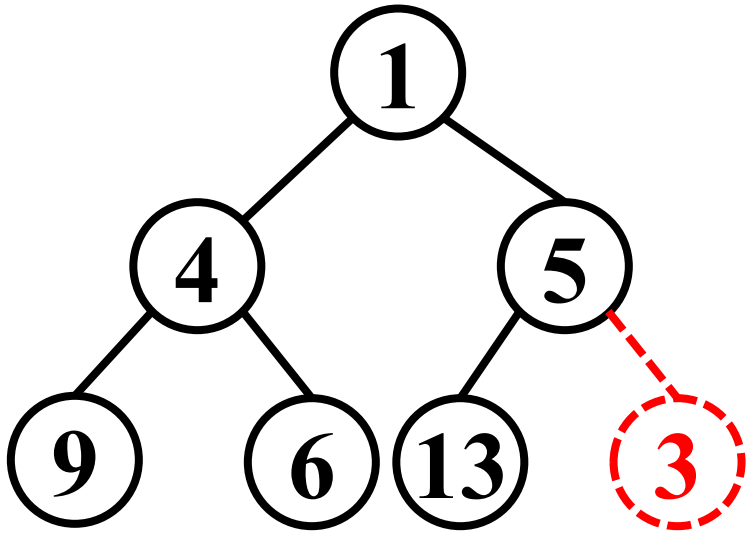
优先队列Q

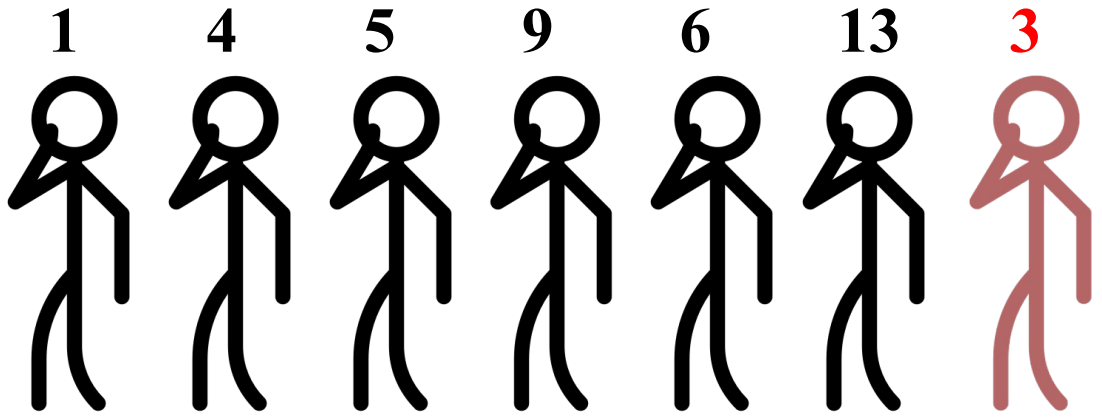
除最底层，第h层有2^{h-1}个顶点

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$



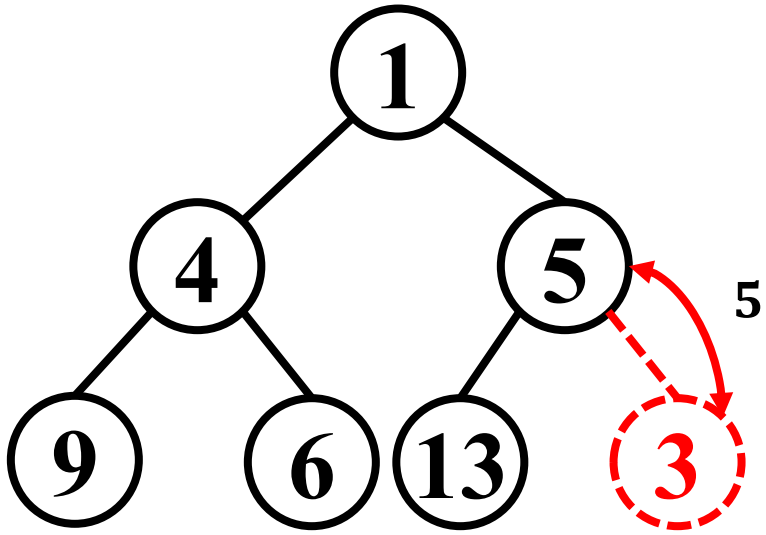


优先队列Q

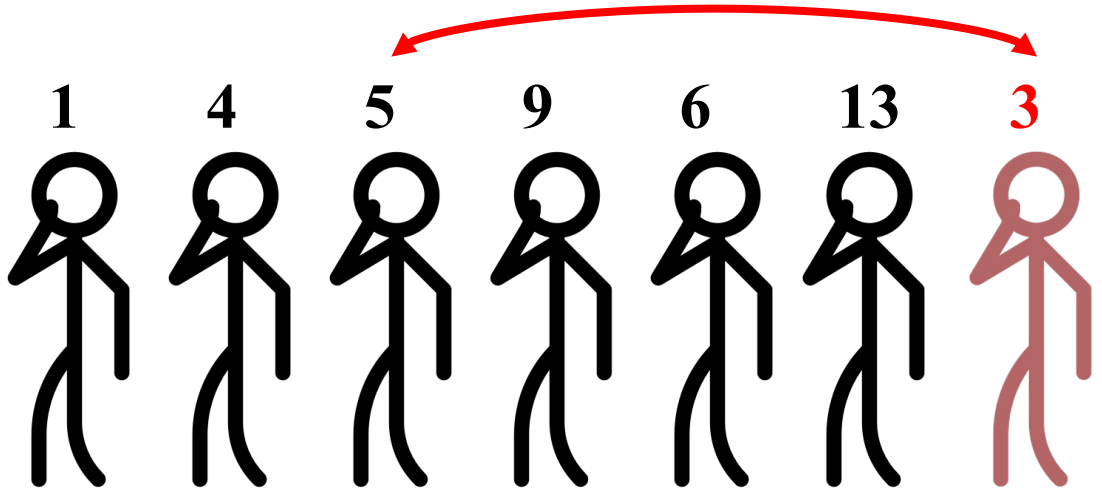
数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$



5 > 3，交换

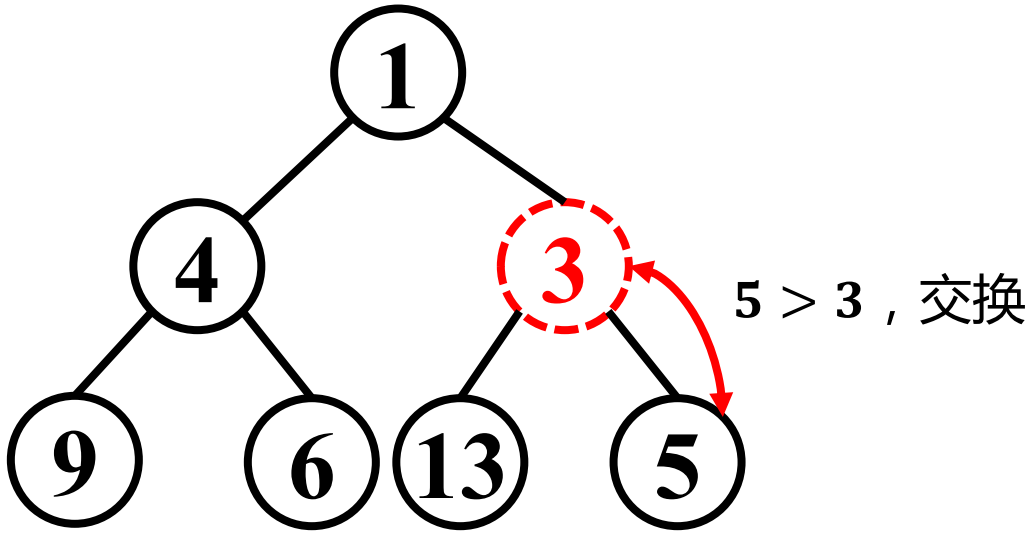


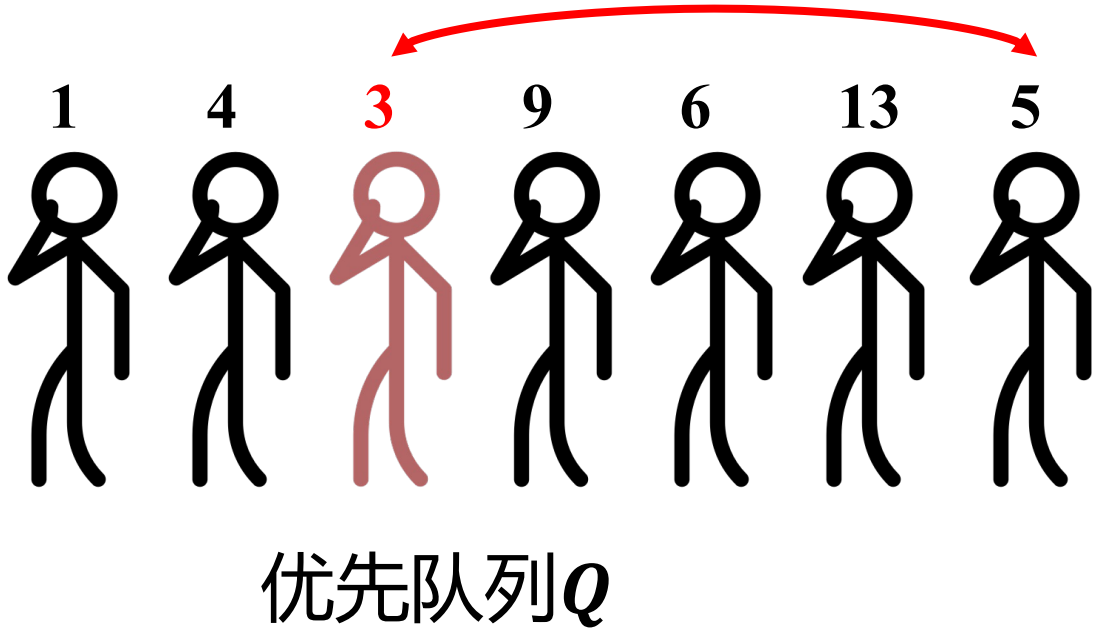
优先队列Q

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$

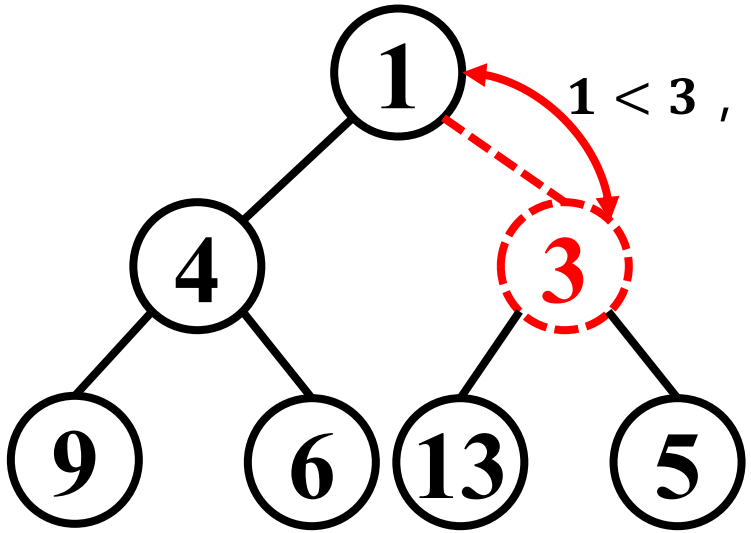


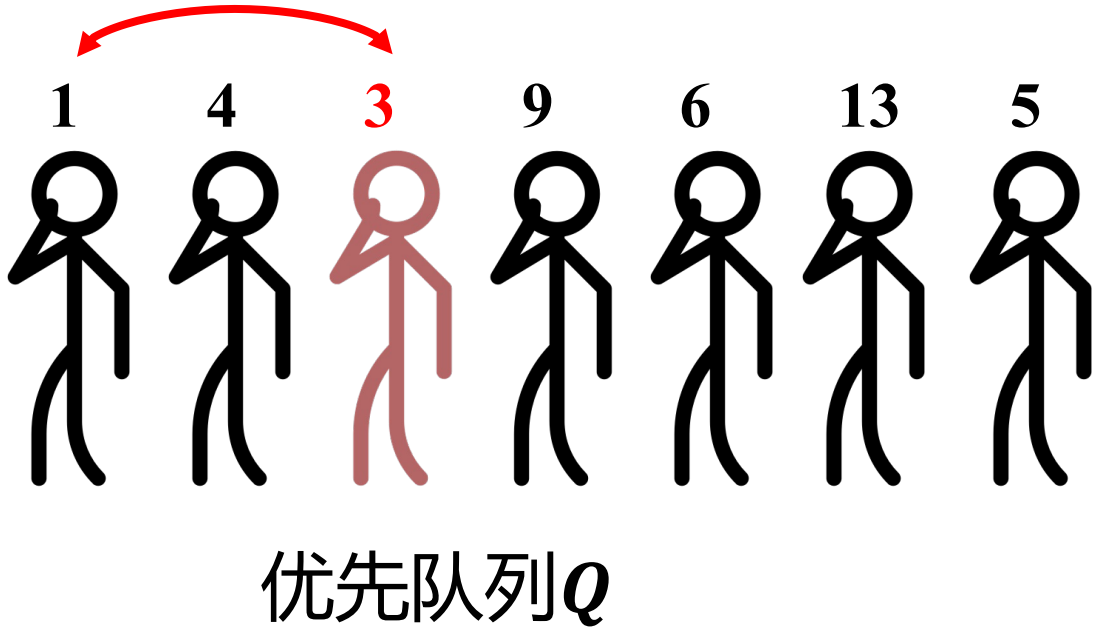


数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$

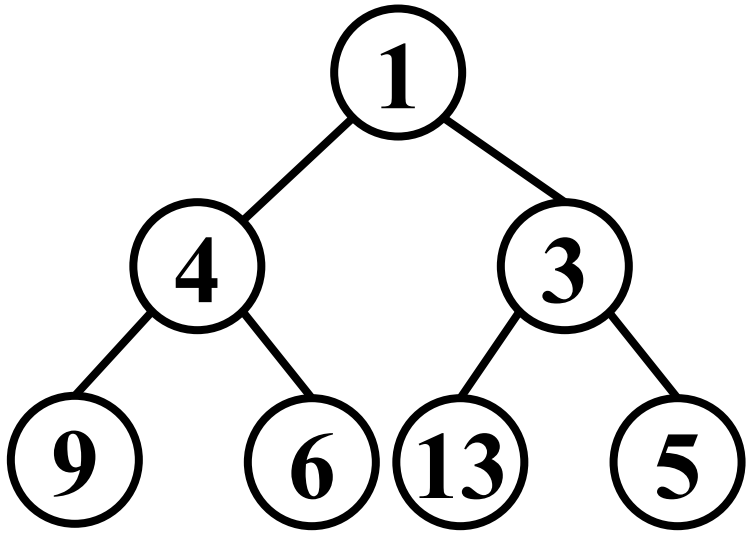


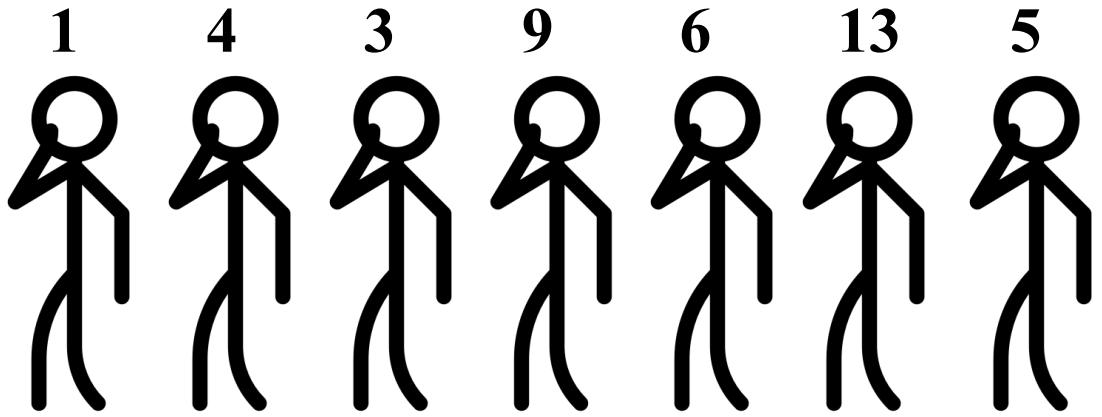


数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$



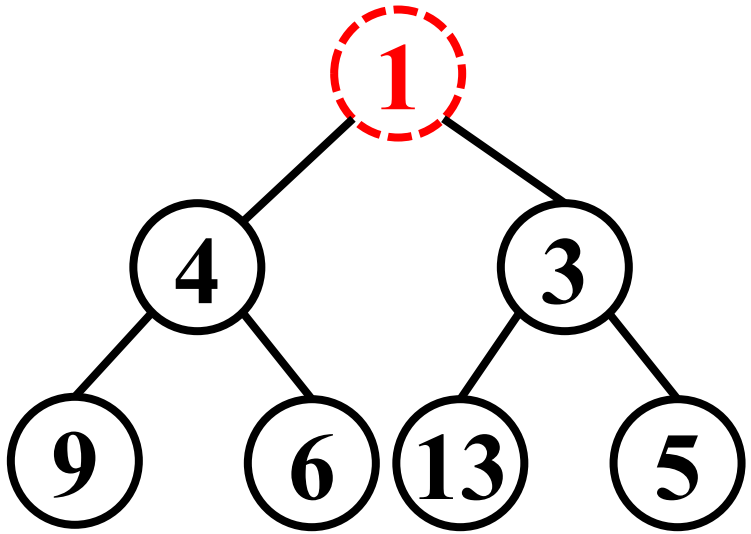


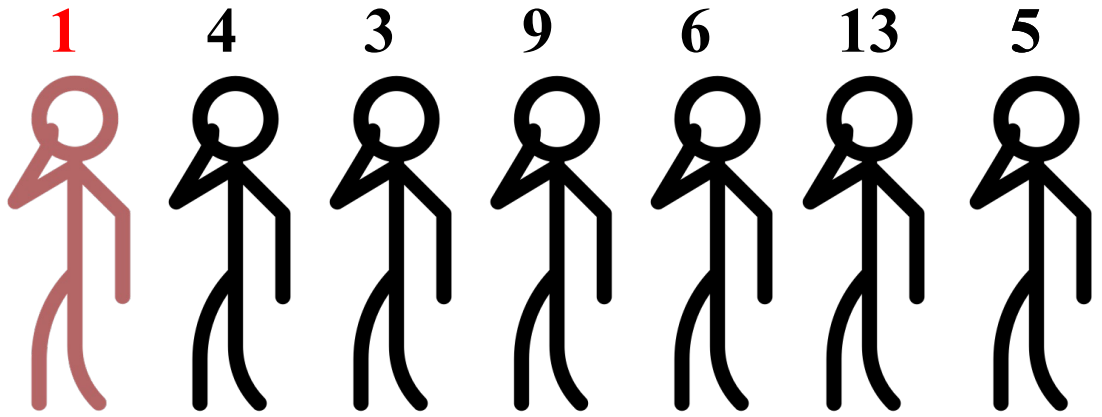
优先队列Q

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$
  - $Q.ExtractMin()$



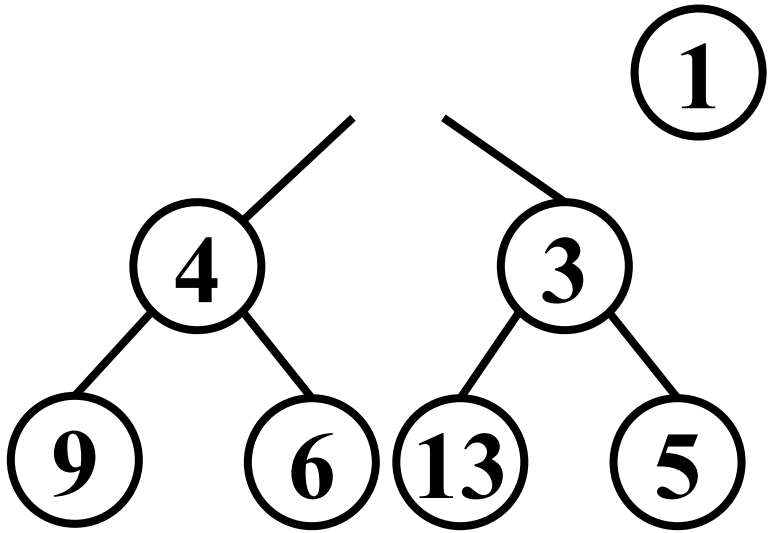


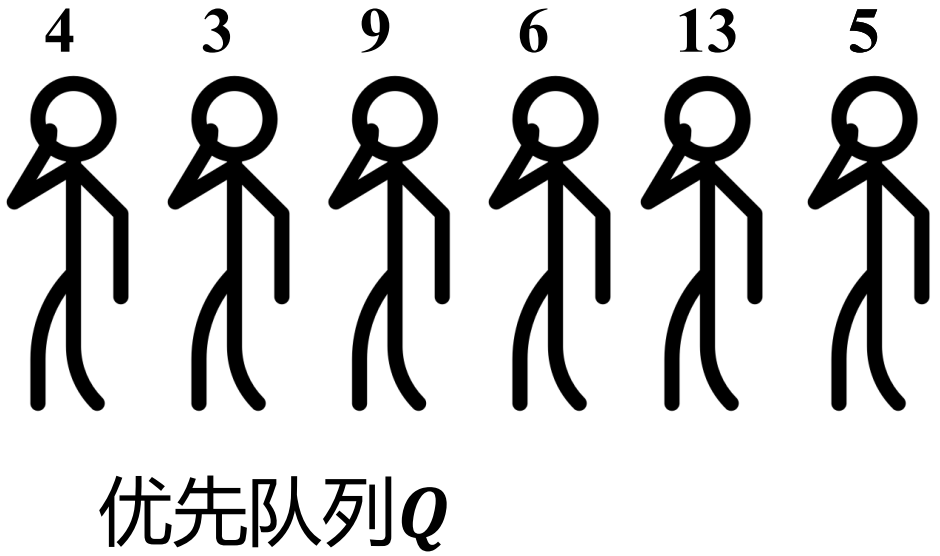
优先队列Q

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$
  - $Q.ExtractMin()$

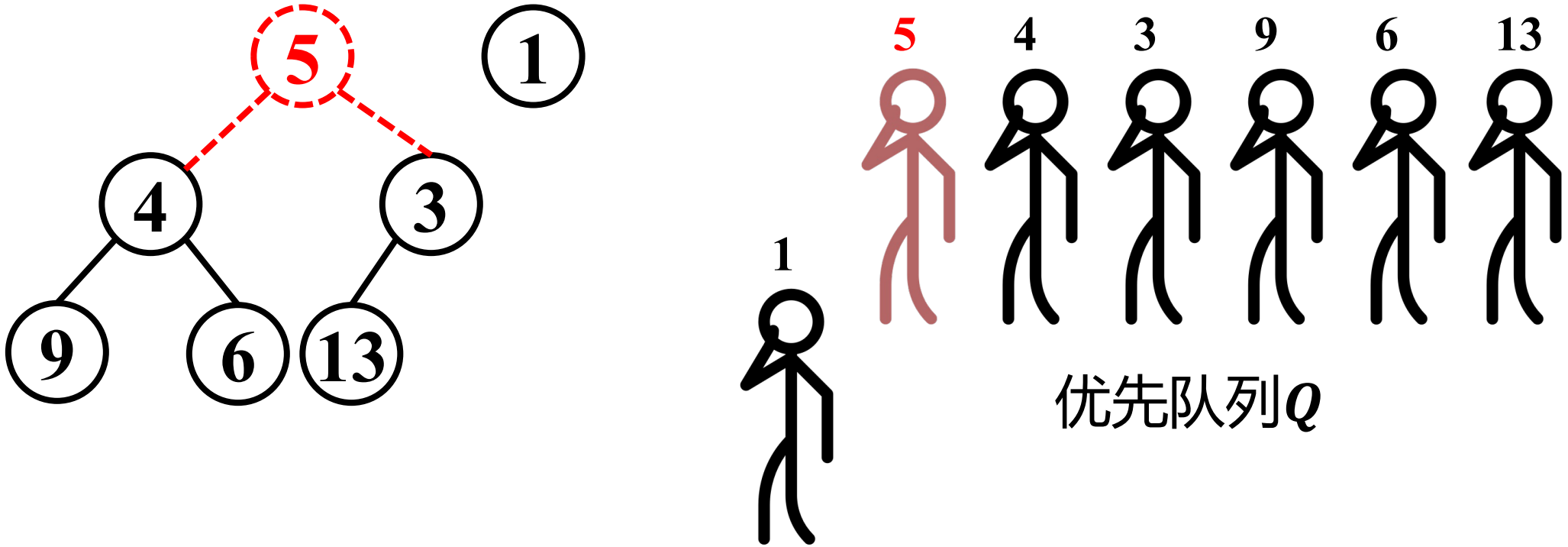




数据结构：优先队列

优先队列

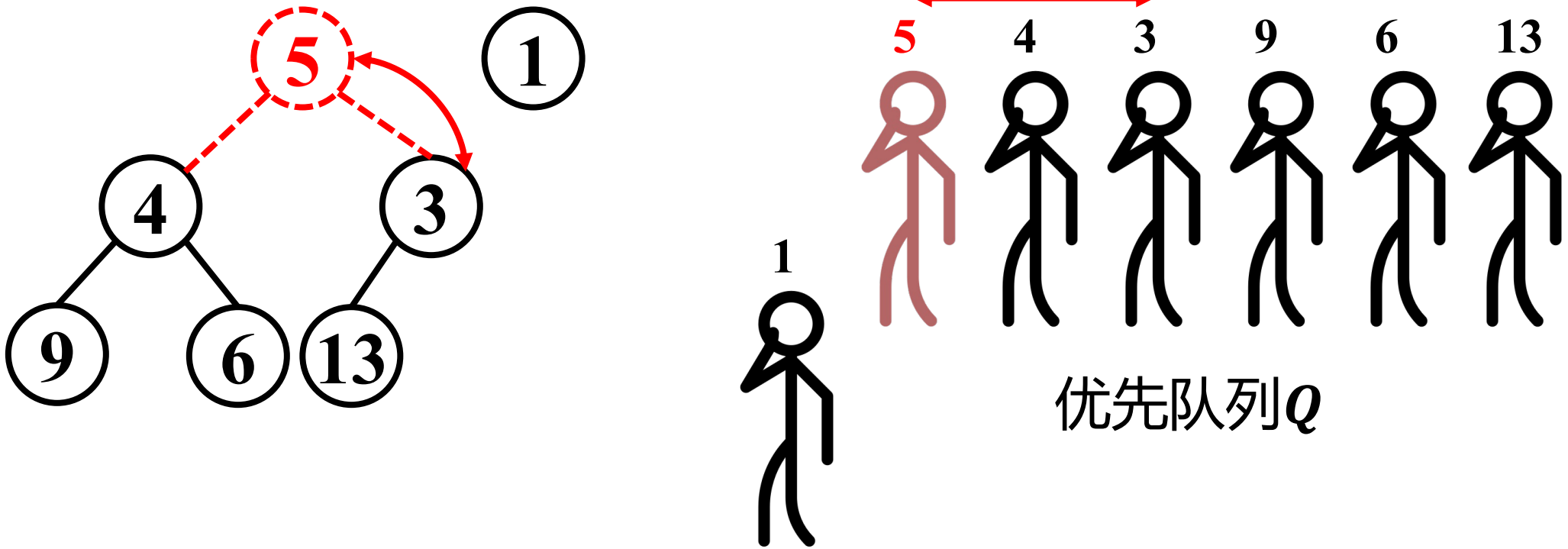
- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$
  - $Q.ExtractMin()$



数据结构：优先队列

优先队列

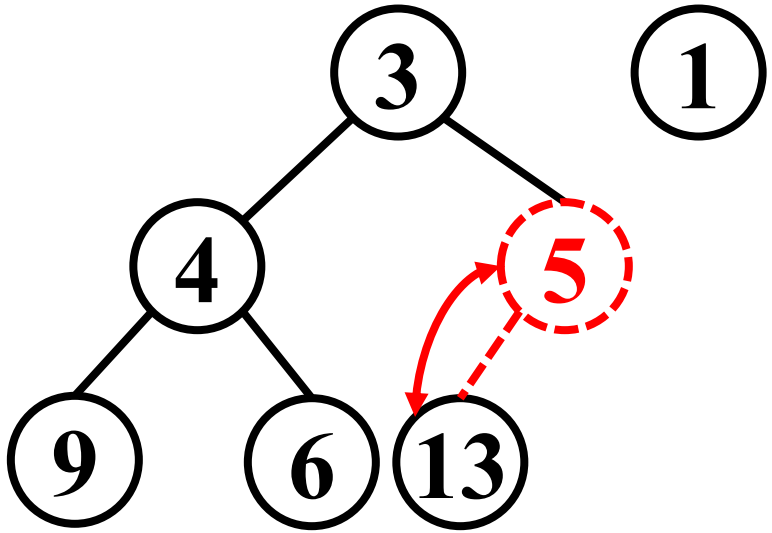
- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$
  - $Q.ExtractMin()$

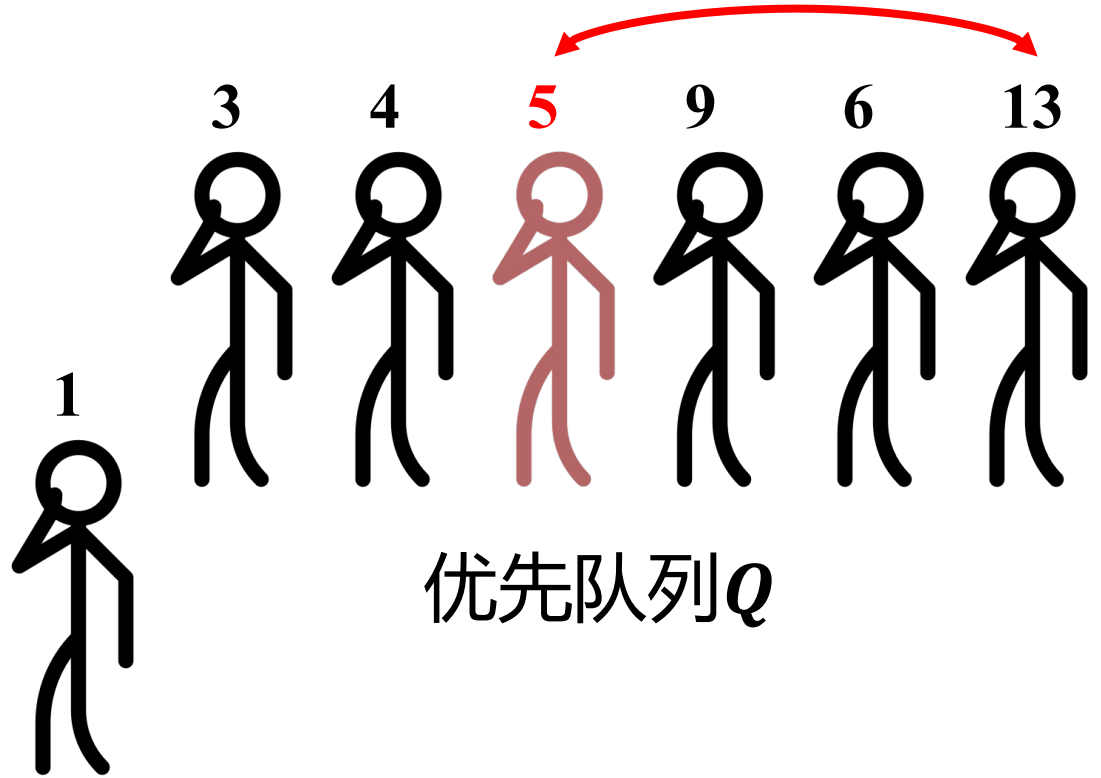


数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$
  - $Q.ExtractMin()$

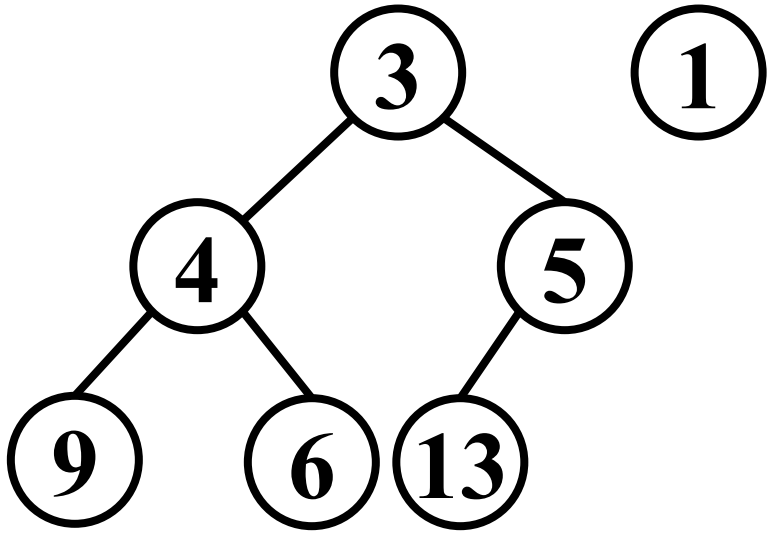


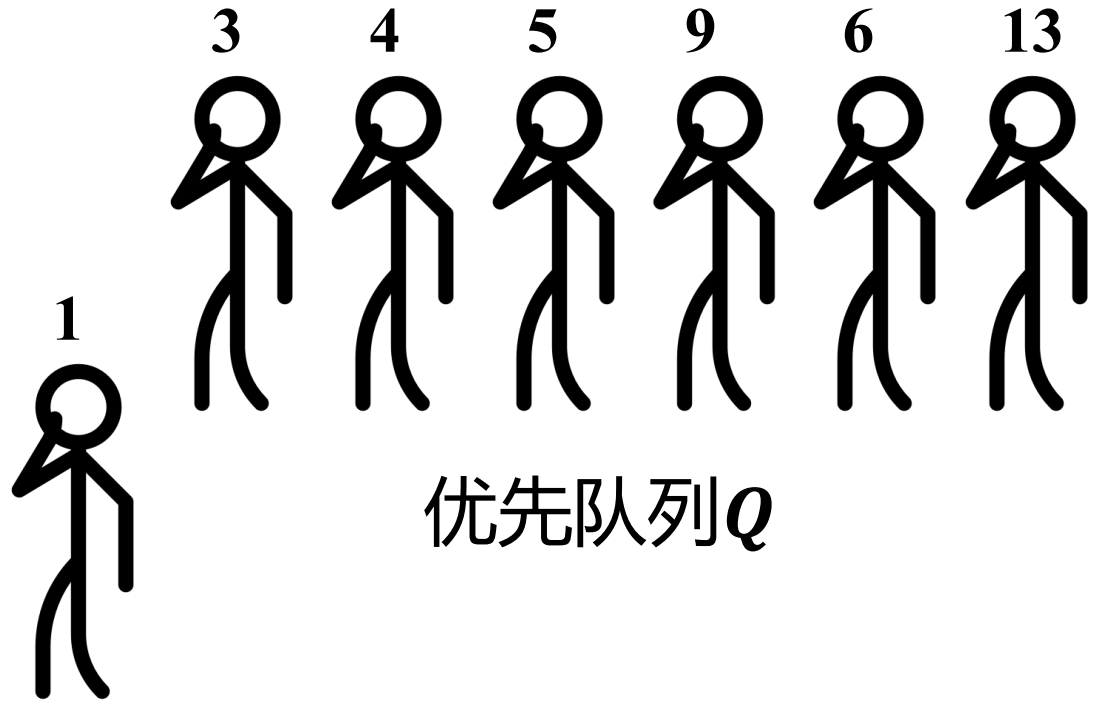


数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$
  - $Q.ExtractMin()$

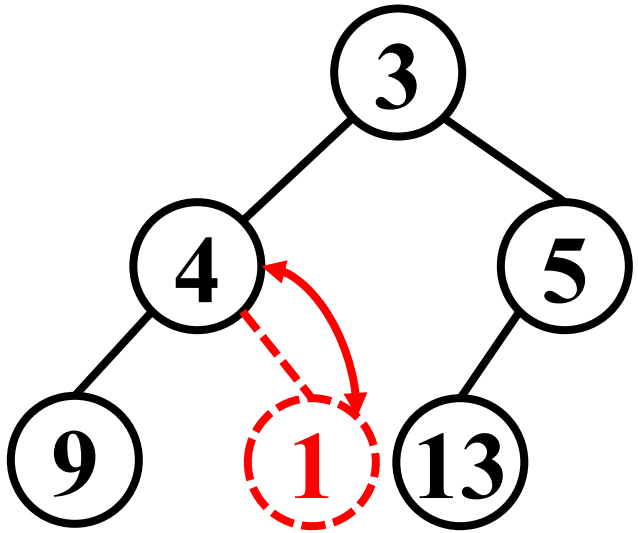


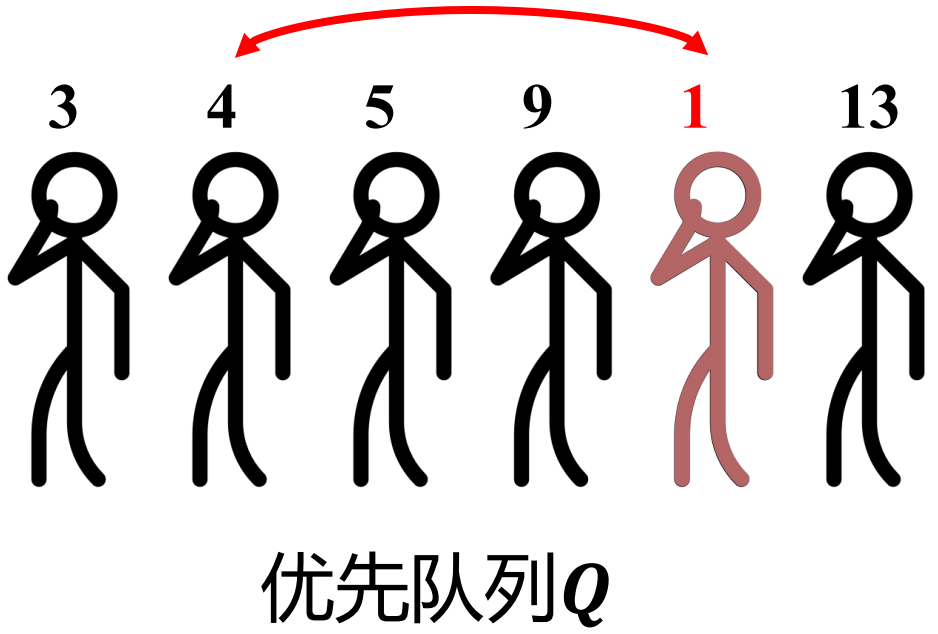


数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$
  - $Q.ExtractMin()$
  - $Q.DecreaseKey()$

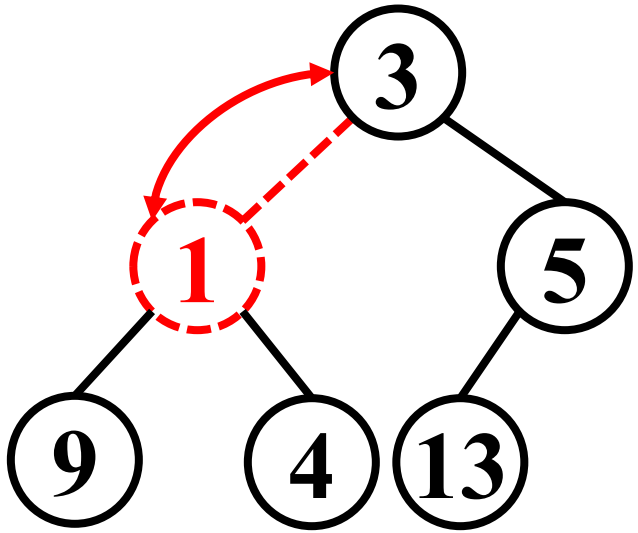


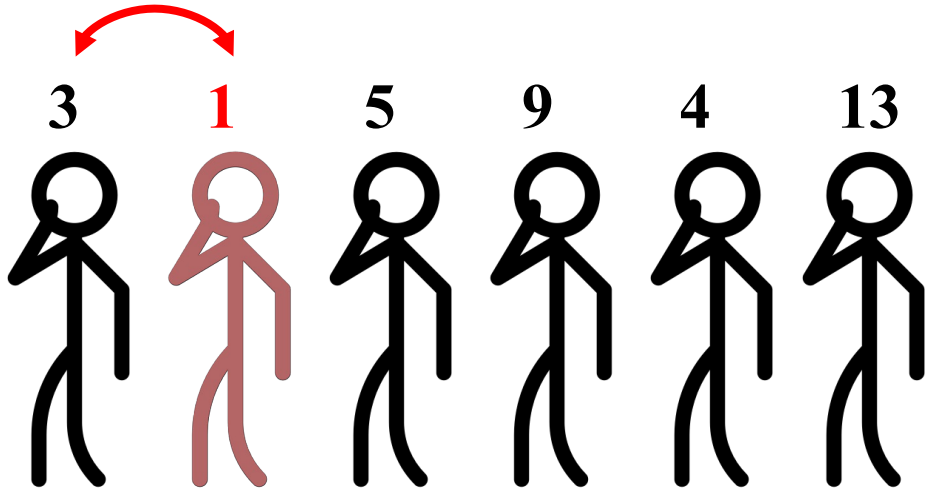


数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$
  - $Q.ExtractMin()$
  - $Q.DecreaseKey()$



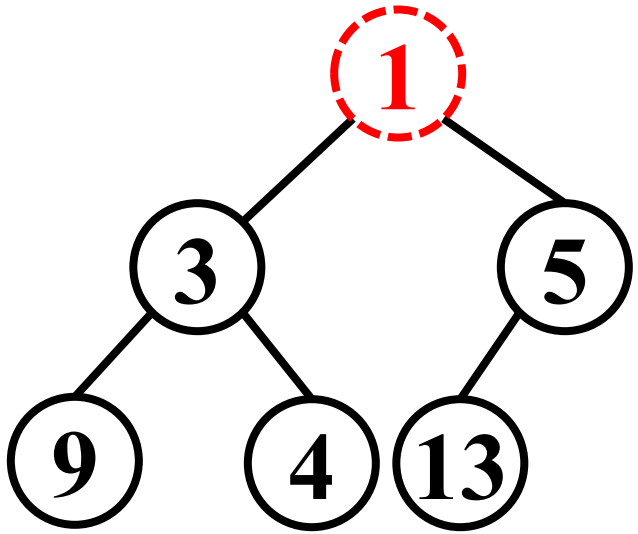


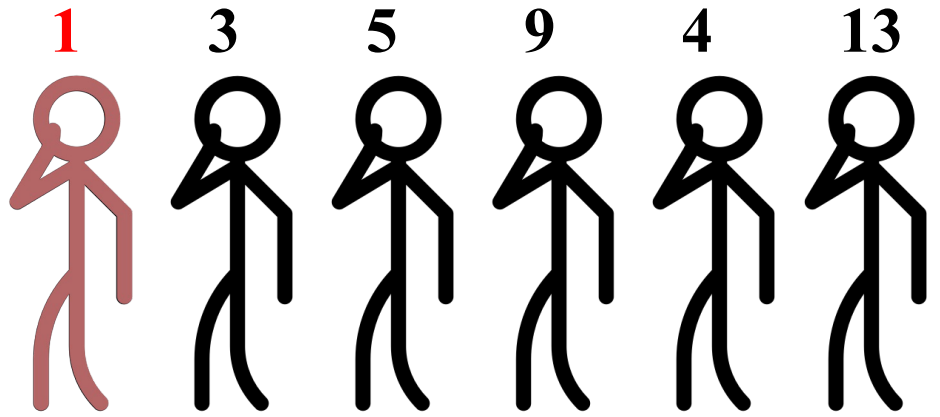
优先队列Q

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$
  - $Q.ExtractMin()$
  - $Q.DecreaseKey()$



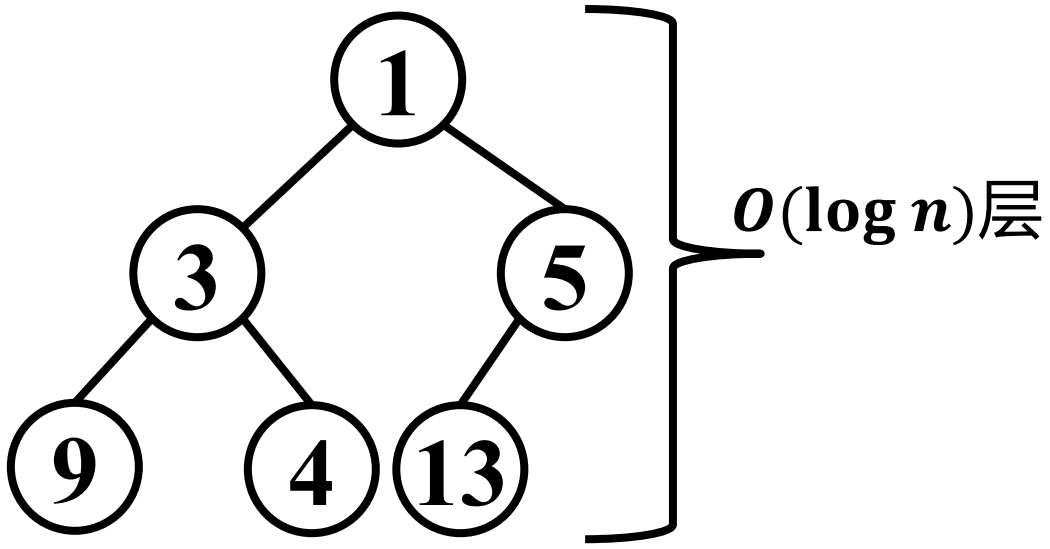


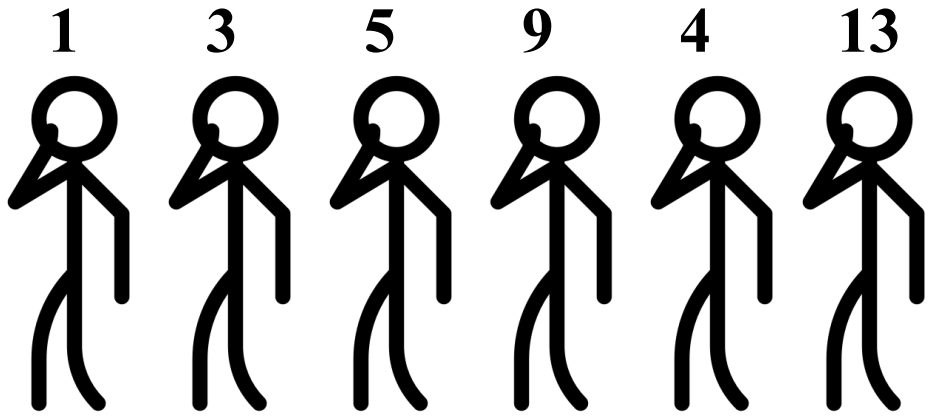
优先队列Q

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$
  - $Q.ExtractMin()$
  - $Q.DecreaseKey()$



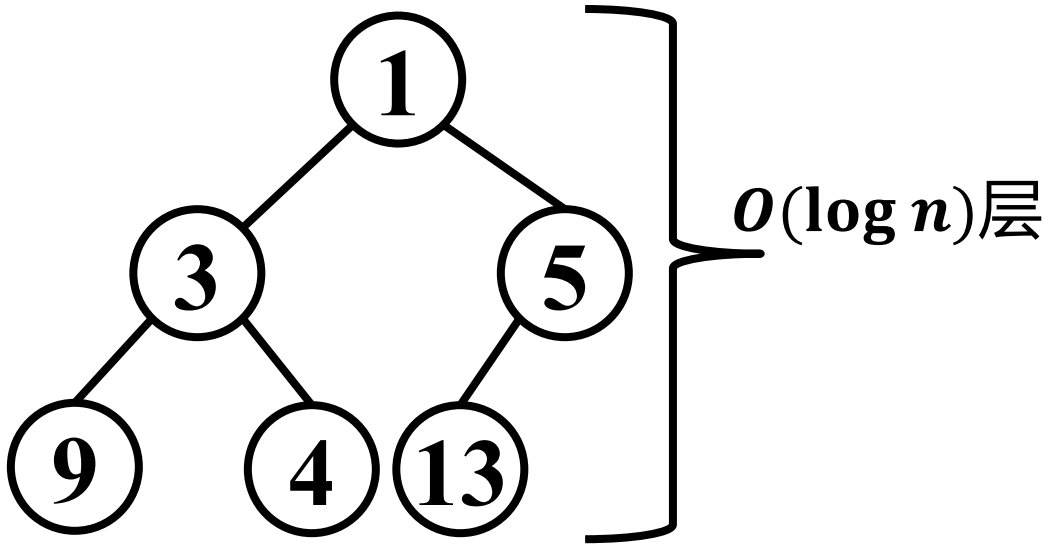


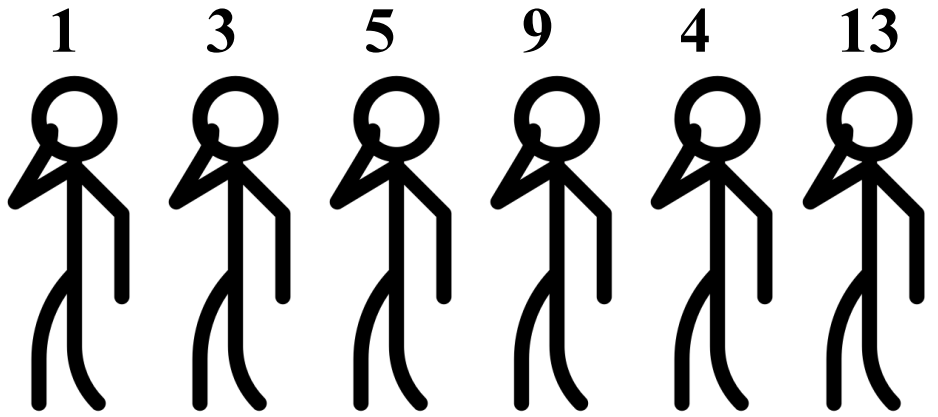
优先队列Q

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$ 时间复杂度$O(\log n)$
  - $Q.ExtractMin()$ 时间复杂度$O(\log n)$
  - $Q.DecreaseKey()$ 时间复杂度$O(\log n)$



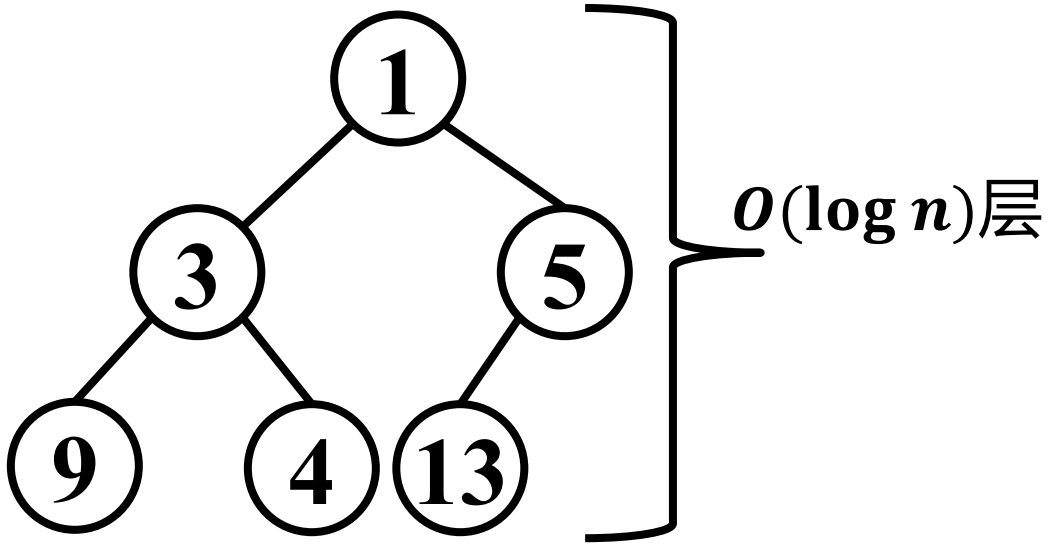


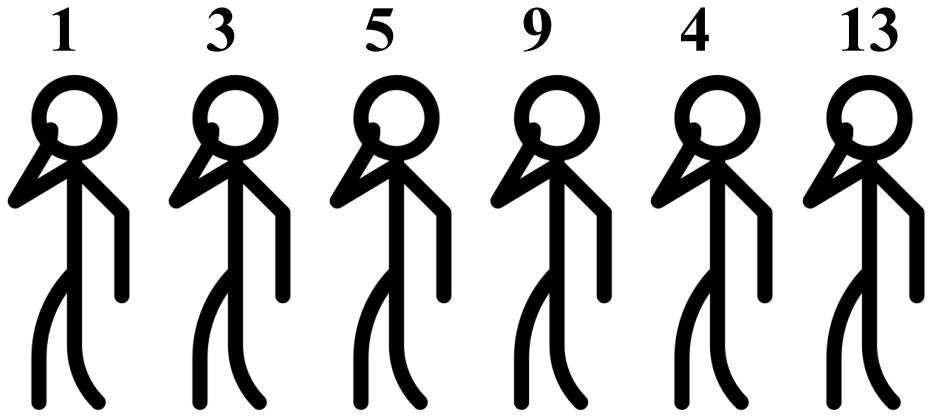
优先队列Q

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$ 时间复杂度$O(\log n)$
  - $Q.ExtractMin()$ 时间复杂度$O(\log n)$
  - $Q.DecreaseKey()$ 时间复杂度$O(\log n)$





优先队列Q

使用优先队列，高效查找的安全边

伪代码

- **MST-Prim-PriQueue(G)**

输入: 图$G =< V, E, W >$
输出: 最小生成树T
新建一维数组$color[1..|V|]$, $dist[1..|V|]$, $pred[1..|V|]$
新建空优先队列Q
//初始化
for $u \in V$ **do**
  $color[u] \leftarrow WHITE$
  $dist[u] \leftarrow \infty$
  $pred[u] \leftarrow NULL$
end
$dist[1] \leftarrow 0$
$Q.Insert(V, dist)$

初始化辅助数组

伪代码

- **MST-Prim-PriQueue(G)**

输入: 图$G =< V, E, W >$
输出: 最小生成树T
新建一维数组$color[1..|V|]$, $dist[1..|V|]$, $pred[1..|V|]$
新建空优先队列Q
//初始化
for $u \in V$ **do**
    $color[u] \leftarrow WHITE$
    $dist[u] \leftarrow \infty$
    $pred[u] \leftarrow NULL$
end
$dist[1] \leftarrow 0$
$Q.Insert(V, dist)$

选择任意起点

伪代码

- **MST-Prim-PriQueue(G)**

输入: 图$G =< V, E, W >$
输出: 最小生成树T
新建一维数组$color[1..|V|]$, $dist[1..|V|]$, $pred[1..|V|]$
新建空优先队列Q
//初始化
for $u \in V$ **do**
    $color[u] \leftarrow WHITE$
    $dist[u] \leftarrow \infty$
    $pred[u] \leftarrow NULL$
end
$dist[1] \leftarrow 0$
$Q.Insert(V, dist)$

初始化优先队列

伪代码

- **MST-Prim-PriQueue(G)**

```
//执行最小生成树算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.Adj[v] do
        if color[u] = WHITE and w(v,u) < dist[u] then
            dist[u] ← w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

依次添加其他顶点

伪代码

- **MST-Prim-PriQueue(G)**

```
//执行最小生成树算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.Adj[v] do
        if color[u] = WHITE and w(v,u) < dist[u] then
            dist[u] ← w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

选择安全边

伪代码

- **MST-Prim-PriQueue(G)**

```
//执行最小生成树算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.Adj[v] do
        if color[u] = WHITE and w(v,u) < dist[u] then
            dist[u] ← w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

更新距离数组，调整优先队列

伪代码

- **MST-Prim-PriQueue(G)**

```
//执行最小生成树算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.Adj[v] do
        if color[u] = WHITE and w(v,u) < dist[u] then
            dist[u] ← w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

标记顶点处理完成

复杂度分析

- **MST-Prim-PriQueue(G)**

输入: 图$G =< V, E, W >$
输出: 最小生成树T
新建一维数组$color[1..|V|]$, $dist[1..|V|]$, $pred[1..|V|]$
新建空优先队列Q
//初始化
for $u \in V$ do
    $color[u] \leftarrow WHITE$
    $dist[u] \leftarrow \infty$
    $pred[u] \leftarrow NULL$
end
$dist[1] \leftarrow 0$
$Q.Insert(V, dist)$

$O(|V|)$

复杂度分析

- **MST-Prim-PriQueue(G)**

```
//执行最小生成树算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()                          ---------- O(log|V|)
    for u ∈ G.Adj[v] do
        if color[u] = WHITE and w(v,u) < dist[u] then
            dist[u] ← w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))         ---------- O(log|V|)
        end
    end
    color[v] ← BLACK
end
```

复杂度分析

- **MST-Prim-PriQueue(G)**

```
//执行最小生成树算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.Adj[v] do
        if color[u] = WHITE and w(v,u) < dist[u] then
            dist[u] ← w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

$O(\log|V|)$

$O(\deg(u)\log|V|)$

$O(\log|V|)$

复杂度分析

- **MST-Prim-PriQueue(G)**

```
//执行最小生成树算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.Adj[v] do
        if color[u] = WHITE and w(v,u) < dist[u] then
            dist[u] ← w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

$O(\log|V|)$

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

$O(\deg(u)\log|V|)$

$O(\log|V|)$

复杂度分析

- **MST-Prim-PriQueue(G)**

```
//执行最小生成树算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.Adj[v] do
        if color[u] = WHITE and w(v,u) < dist[u] then
            dist[u] ← w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

$O(\log|V|)$

$O(\deg(u)\log|V|)$

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

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

$$\sum_{u\in V}\deg(u) = 2|E|$$

复杂度分析

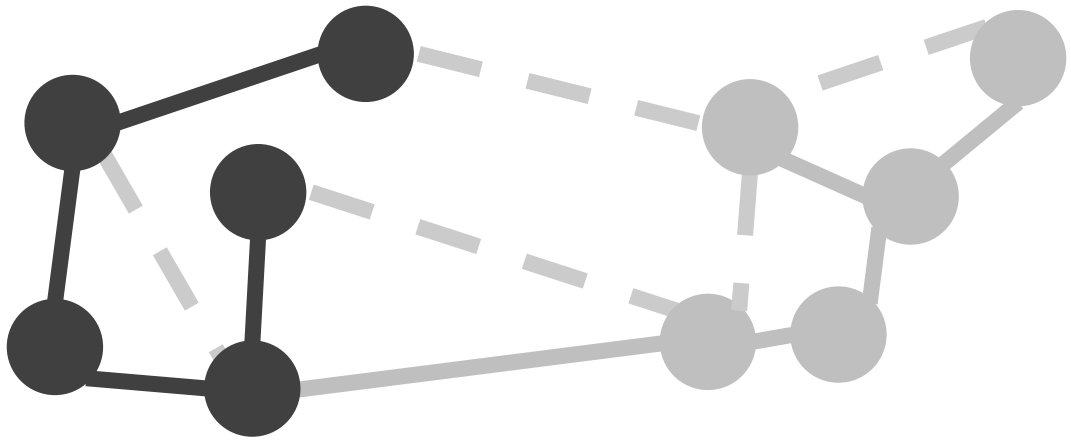
- **MST-Prim-PriQueue(G)**

```
//执行最小生成树算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.Adj[v] do
        if color[u] = WHITE and w(v,u) < dist[u] then
            dist[u] ← w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

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

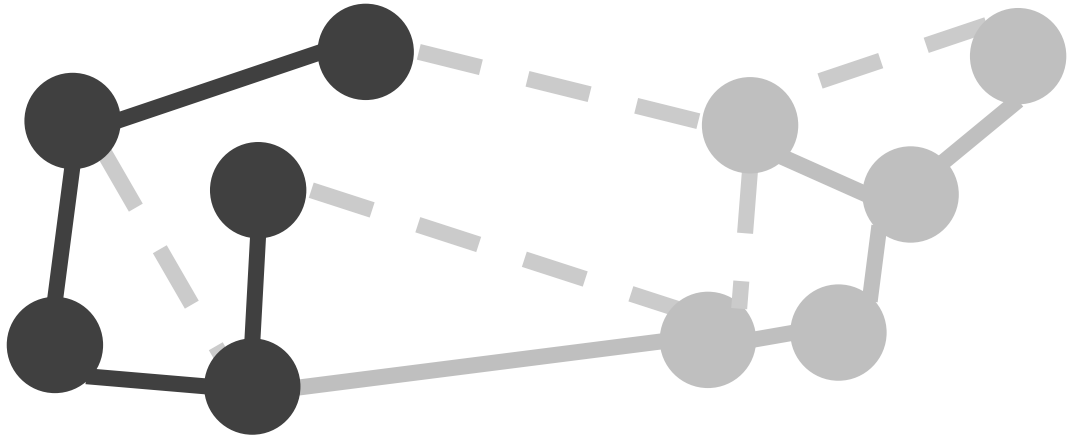
小结

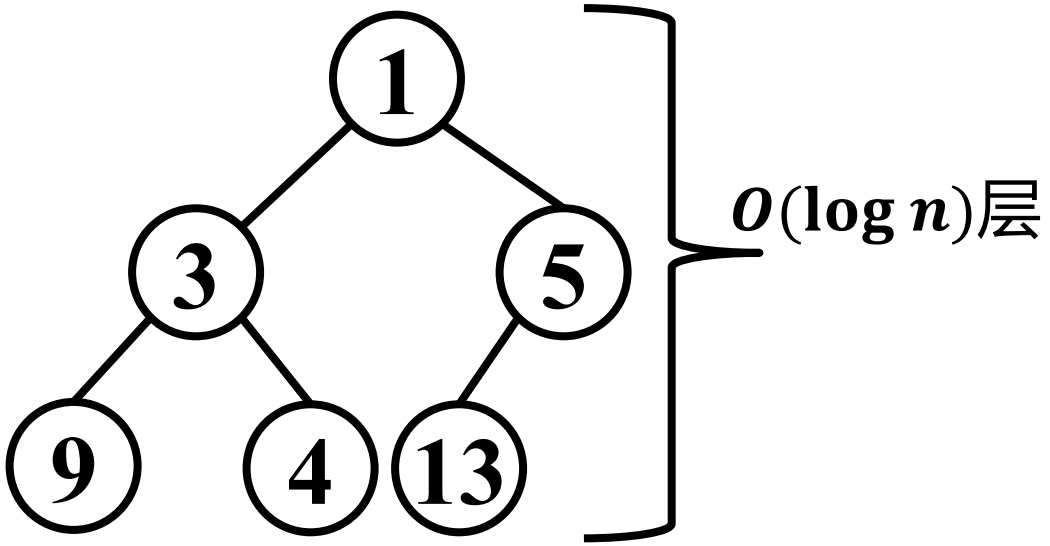
通用框架	Prim算法
判断是否成环	保持树的结构



小结

通用框架	Prim算法
判断是否成环	保持树的结构
高效寻找轻边	使用优先队列





图算法篇：最小生成树之Kruskal算法

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》

提纲

- 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

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

提纲

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

Kruskal算法

- 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

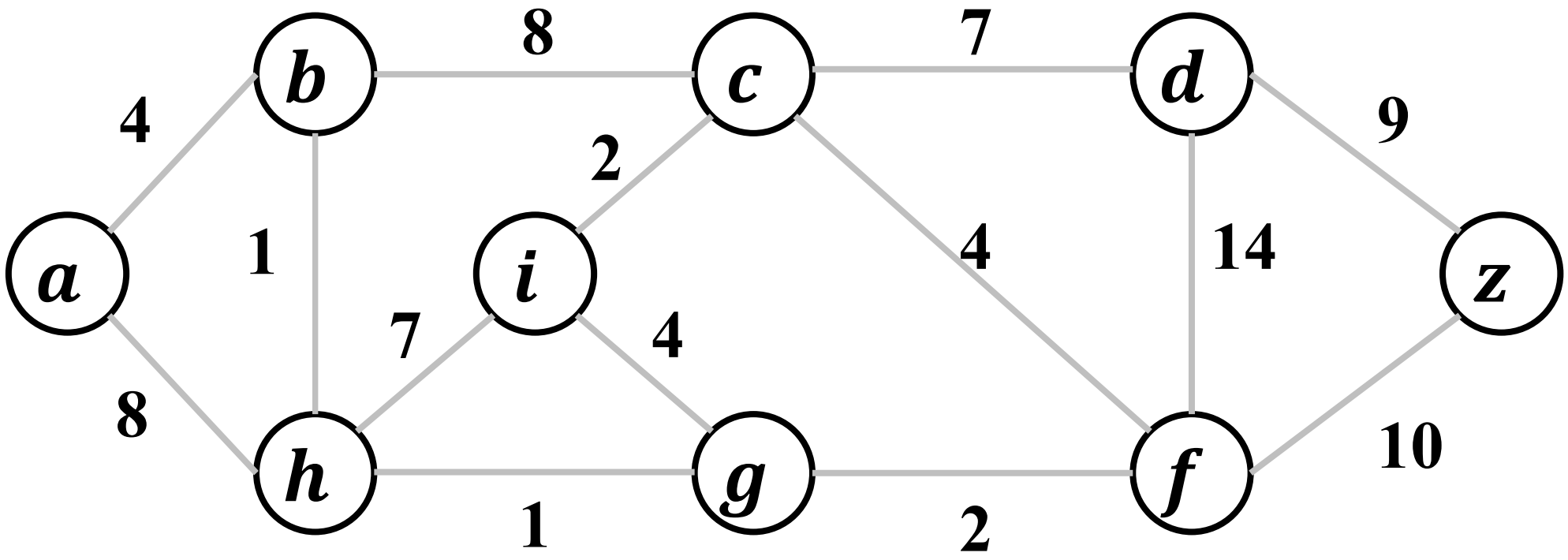
Kruskal算法

- 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

算法实例

- 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

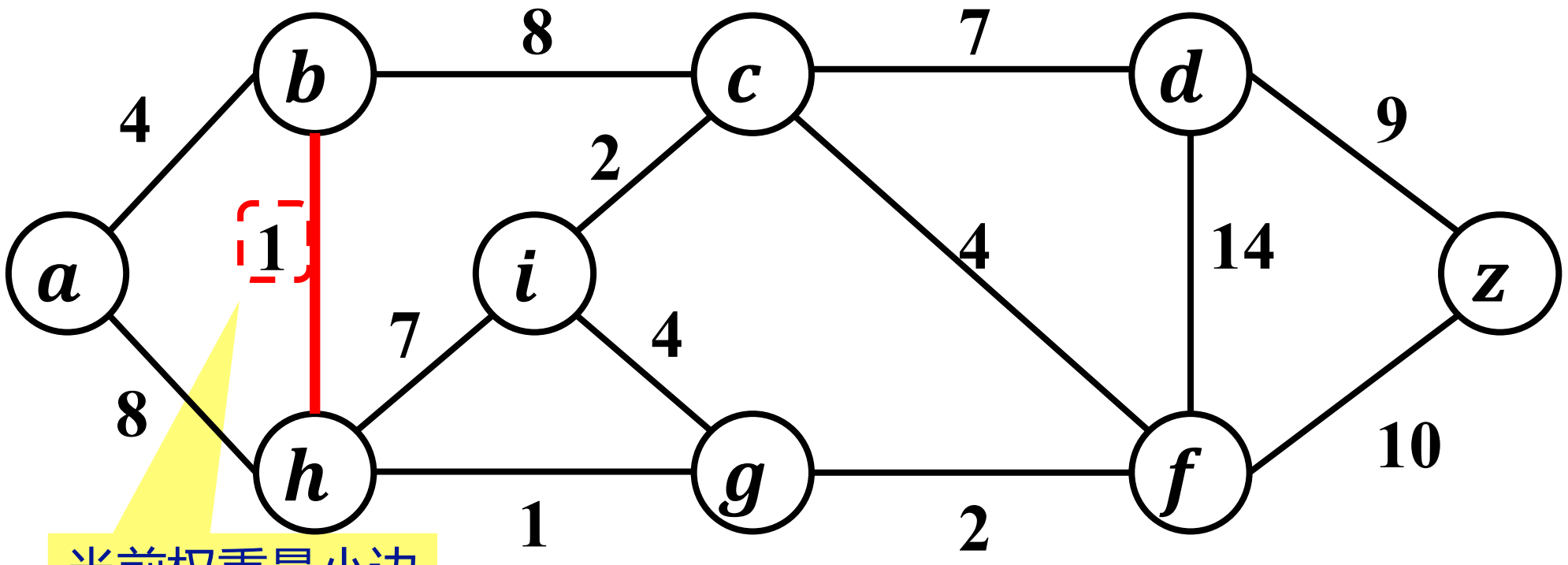
- 每个顶点是一个生成子树
- 一共9个生成子树



算法实例

- 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

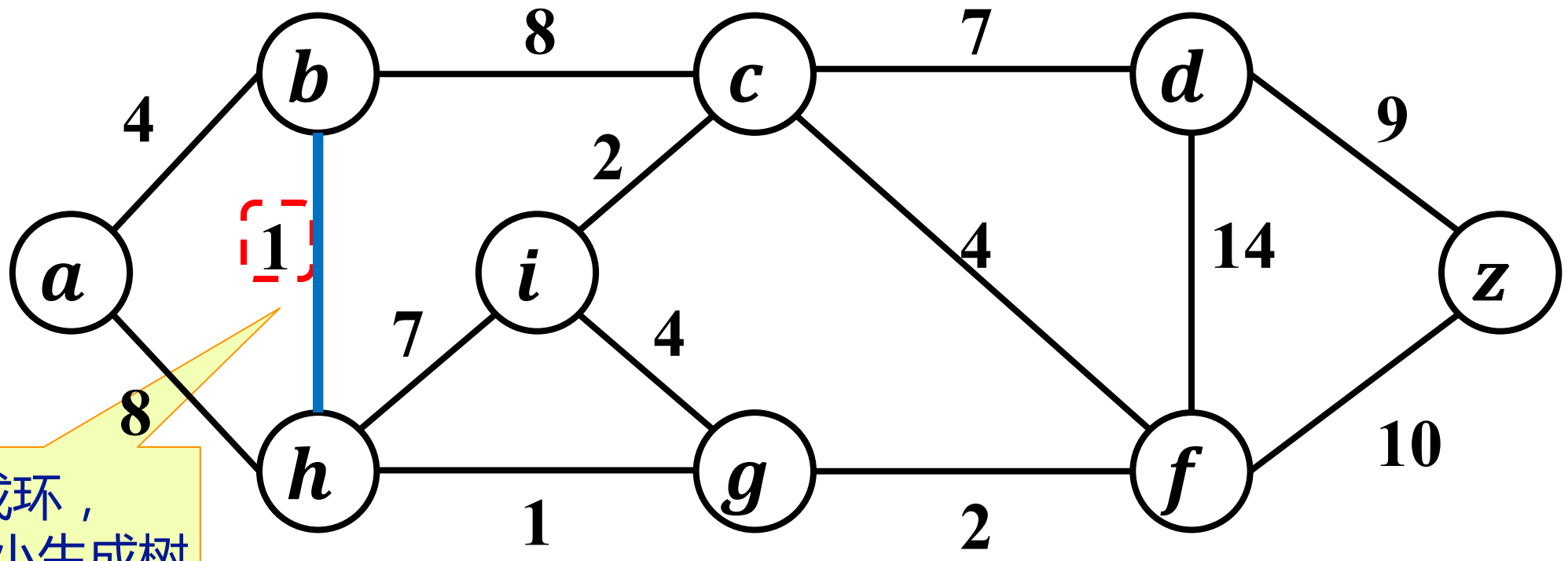
- 每个顶点是一个生成子树
- 一共9个生成子树



算法实例

- 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

- $\{(b,h)\}$构成一棵生成子树，其他顶点每个是一个生成子树
- 一共8个生成子树

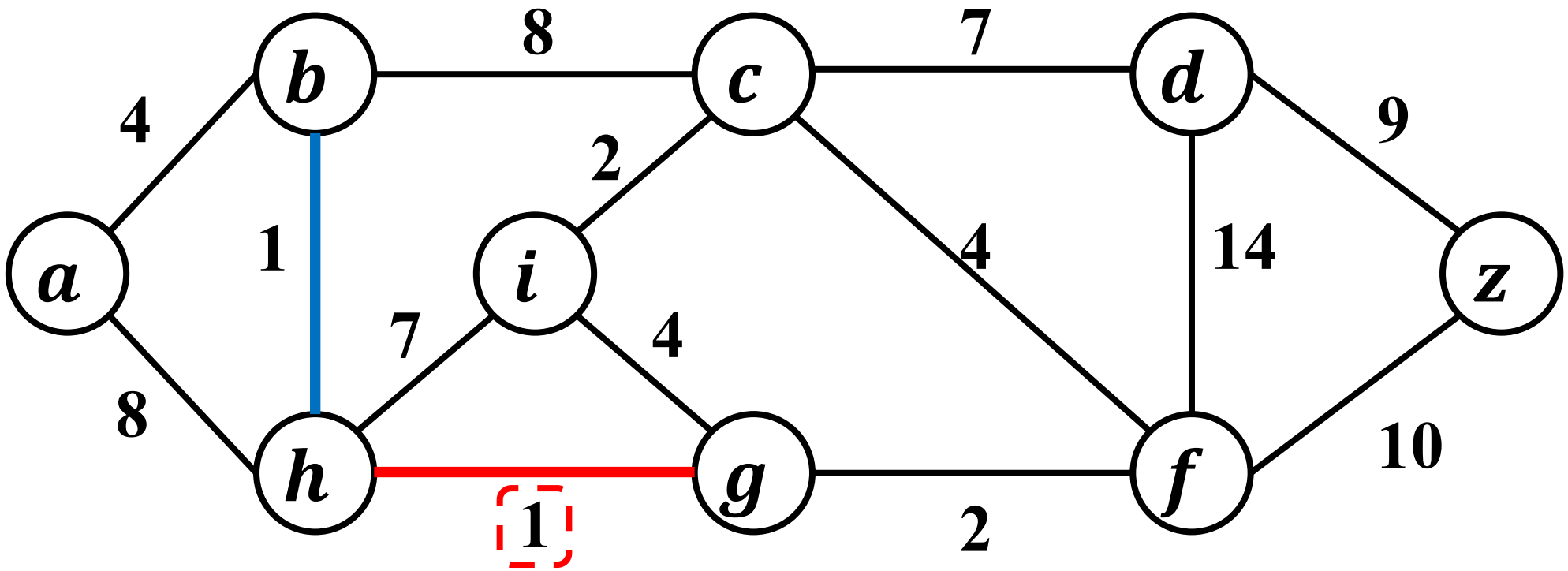


未成环，
加入最小生成树

算法实例

- 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

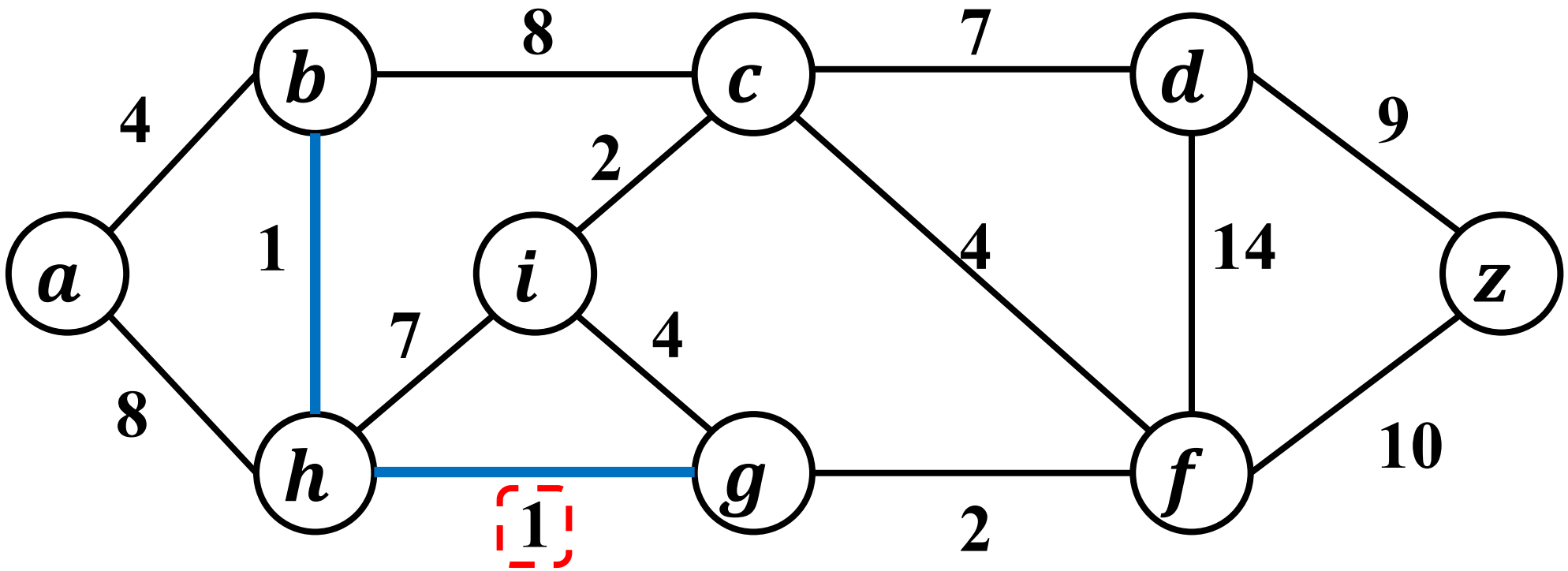
- $\{(b,h)\}$构成一棵生成子树，其他顶点每个是一个生成子树
- 一共8个生成子树



算法实例

- 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

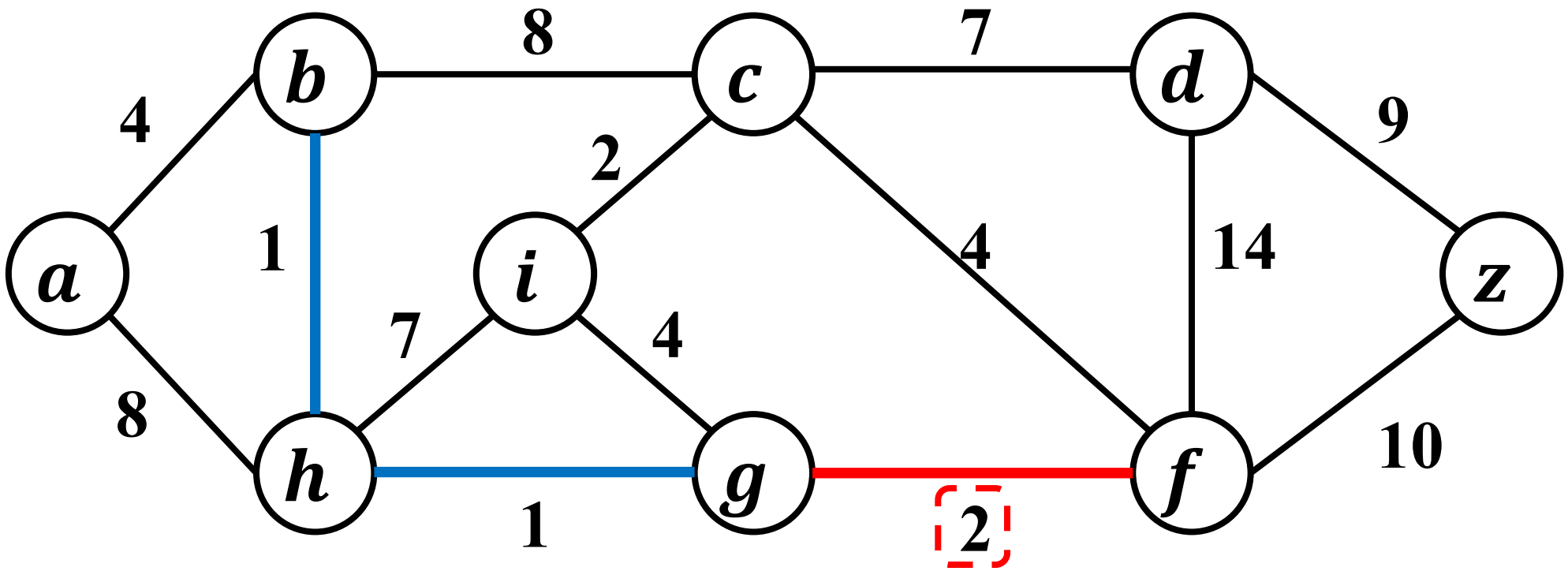
- $\{(b,h),\{h,g\}\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共7个生成子树



算法实例

- 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

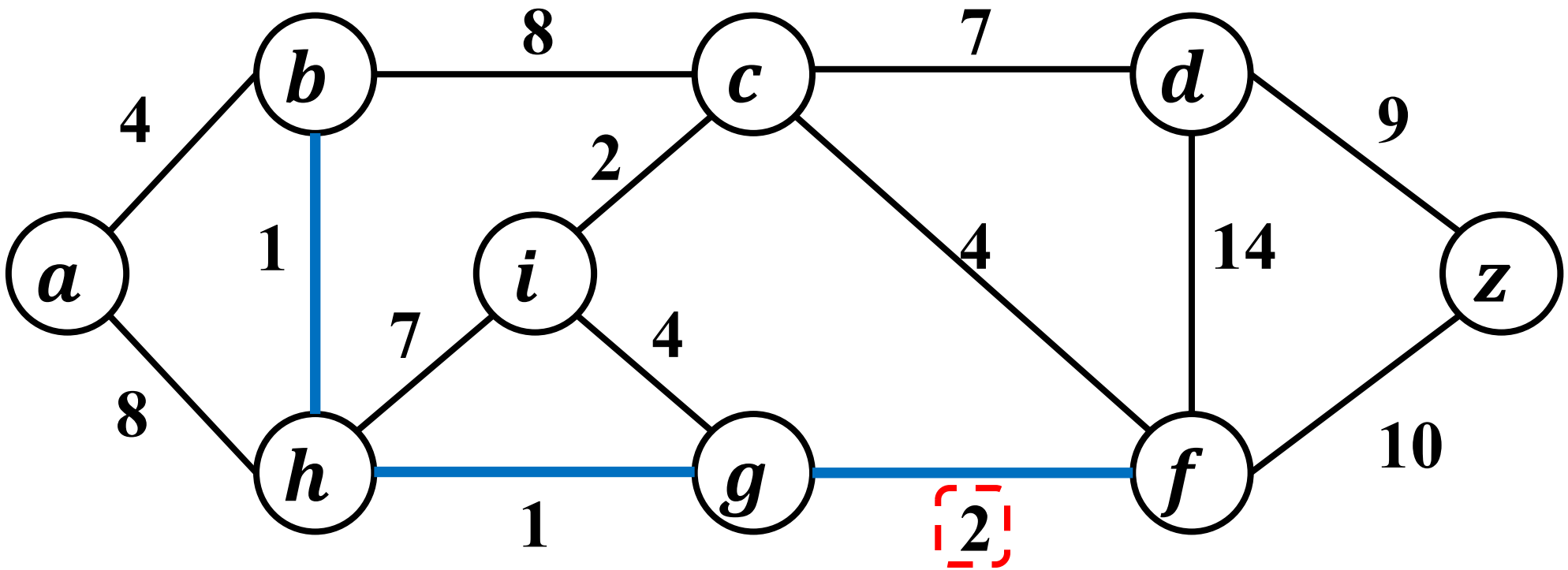
- $\{(b,h),\{h,g\}\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共7个生成子树



算法实例

- 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

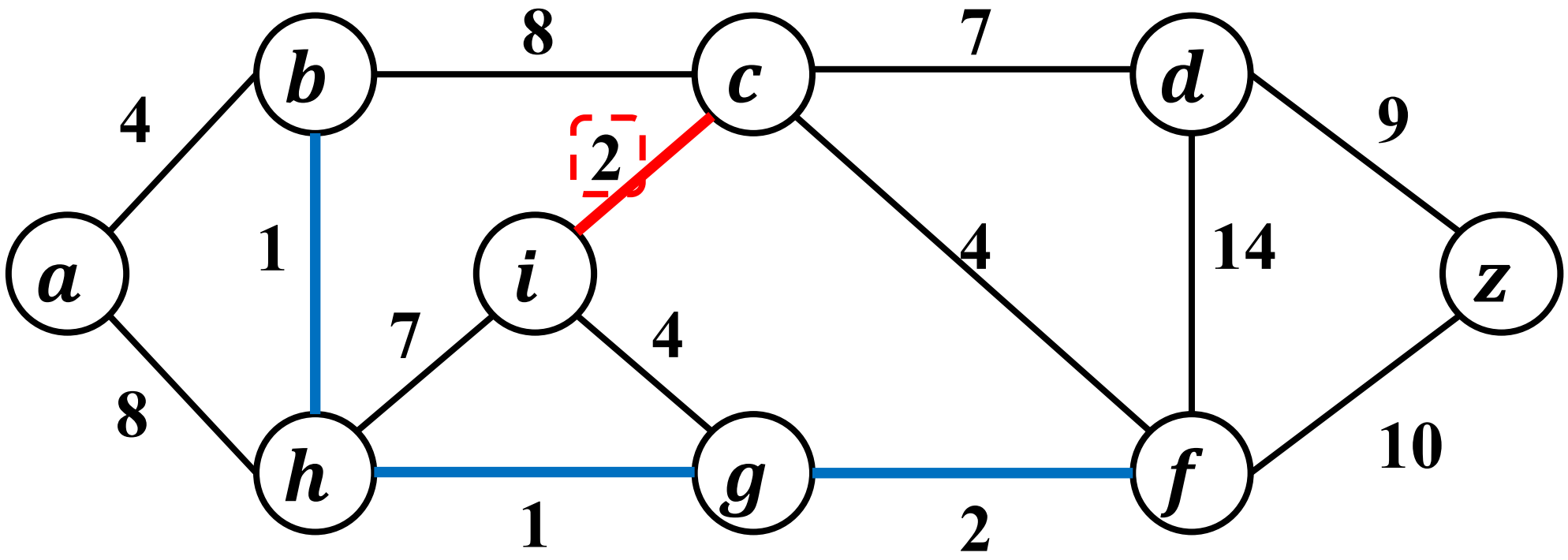
- $\{(b,h),\{h,g\},(g,f)\}$ 构成一棵生成子树 其他顶点每个是一个生成子树
- 一共6个生成子树



算法实例

- 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

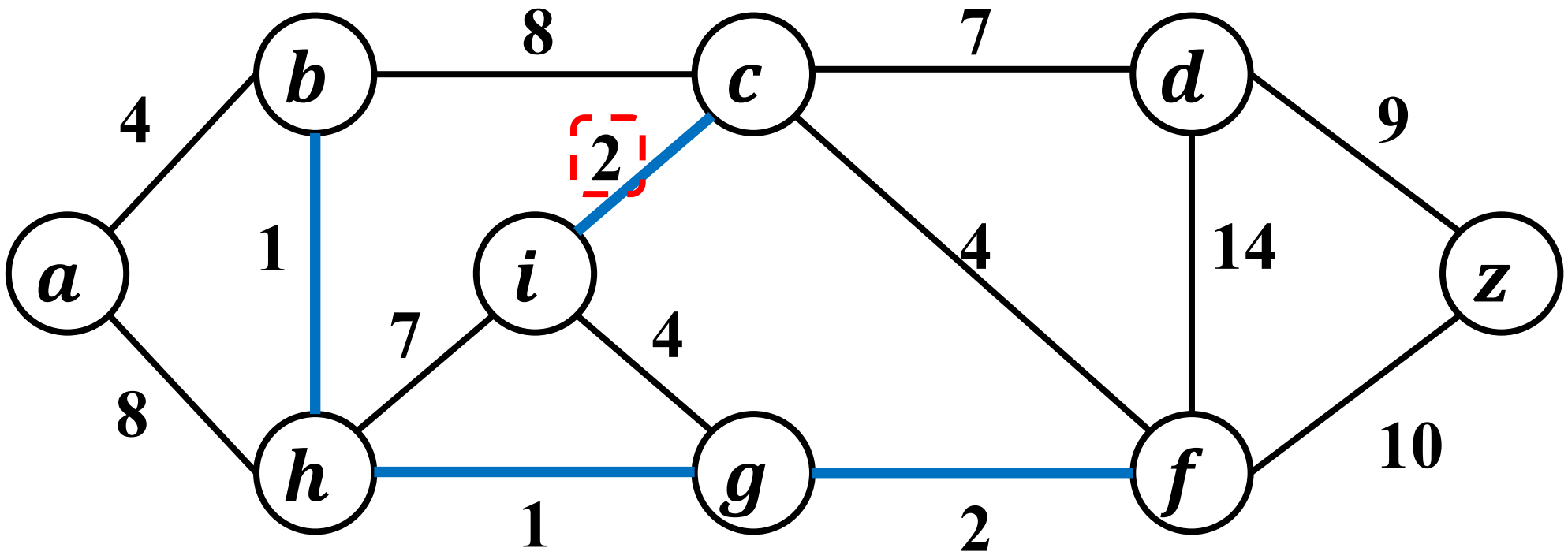
- $\{(b,h),\{h,g\},(g,f)\}$ 构成一棵生成子树 其他顶点每个是一个生成子树
- 一共6个生成子树



算法实例

- 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

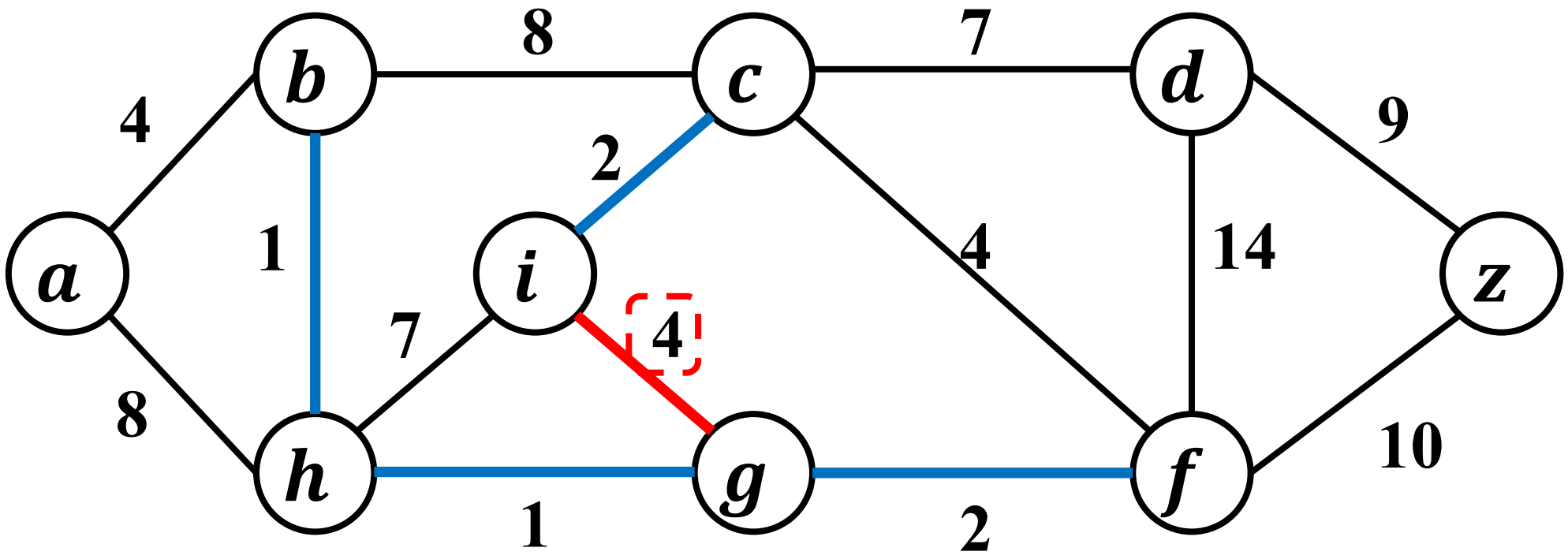
- $\{(b,h),\{h,g\},(g,f)\}$ 构成一棵生成子树，$\{(i,c)\}$构成一棵生成子树，其他顶点每个是一个生成子树
- 一共5个生成子树



算法实例

- 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

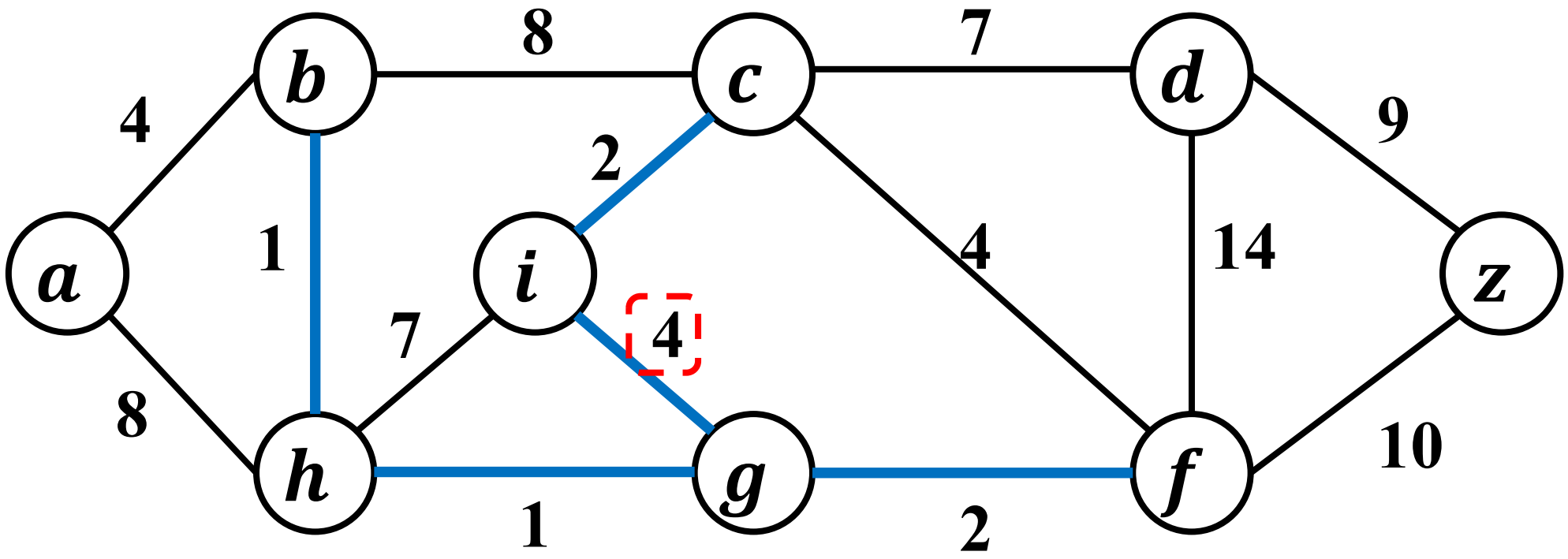
- $\{(b,h),\{h,g\},(g,f)\}$ 构成一棵生成子树，$\{(i,c)\}$构成一棵生成子树，其他顶点每个是一个生成子树
- 一共5个生成子树



算法实例

- 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

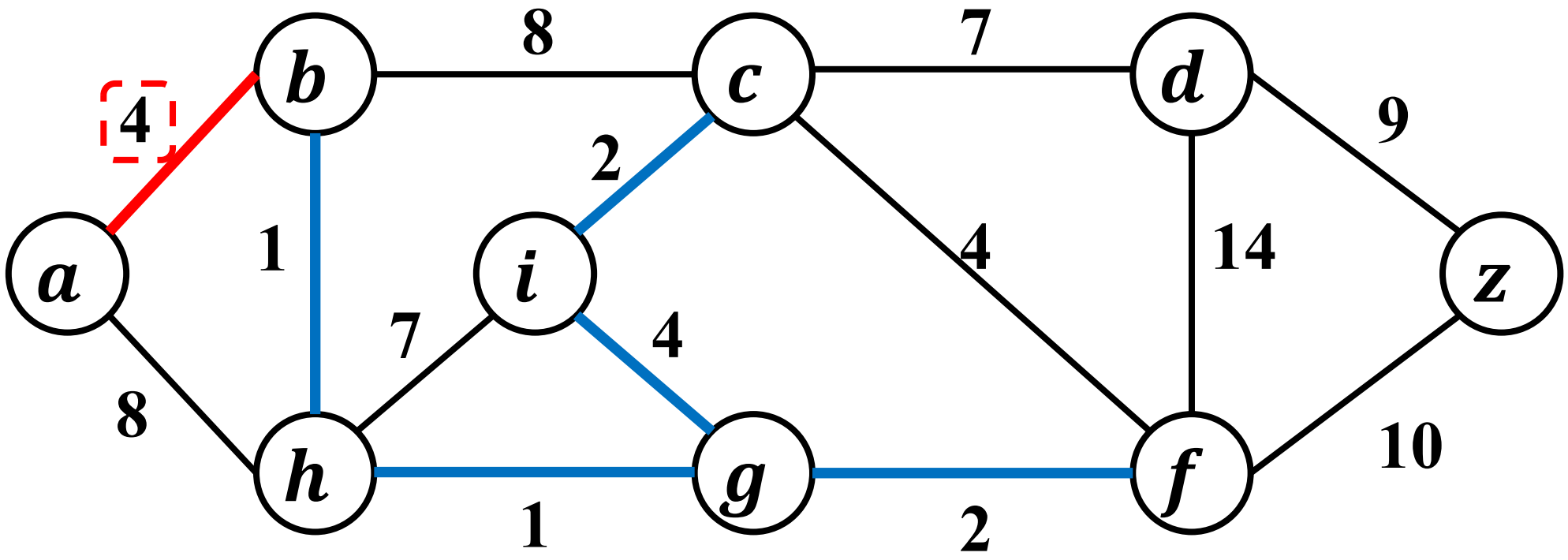
- $\{(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共4个生成子树



算法实例

- 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

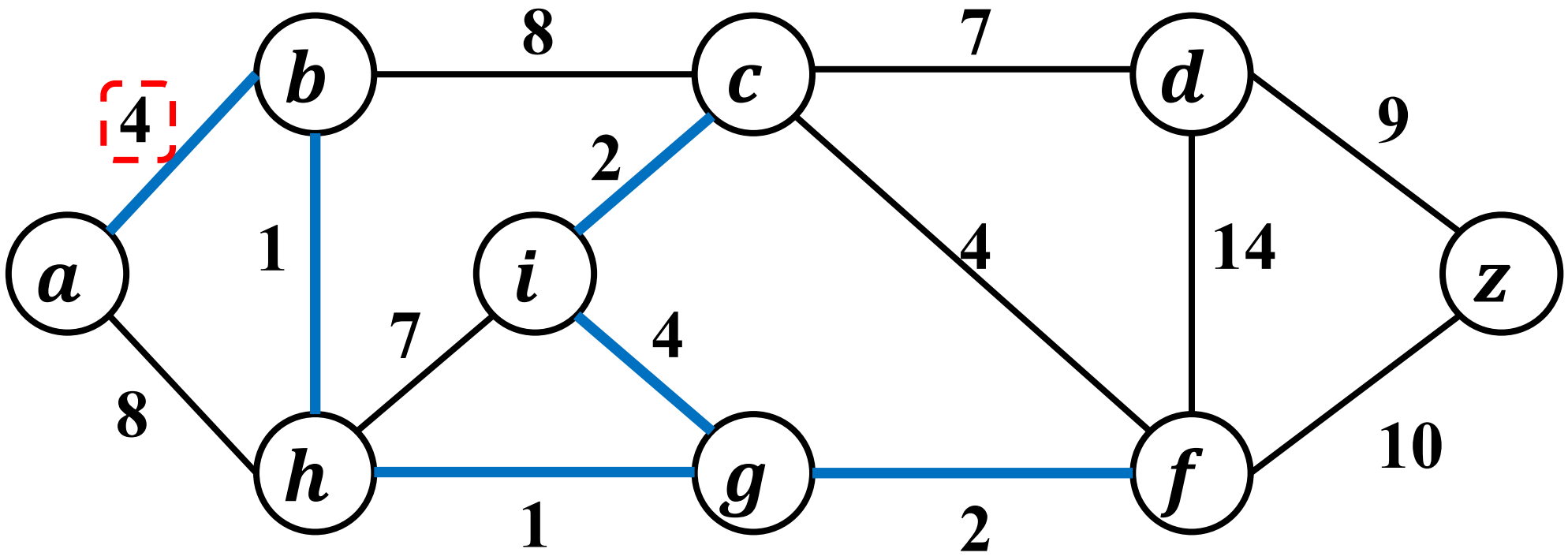
- $\{(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共4个生成子树



算法实例

- 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

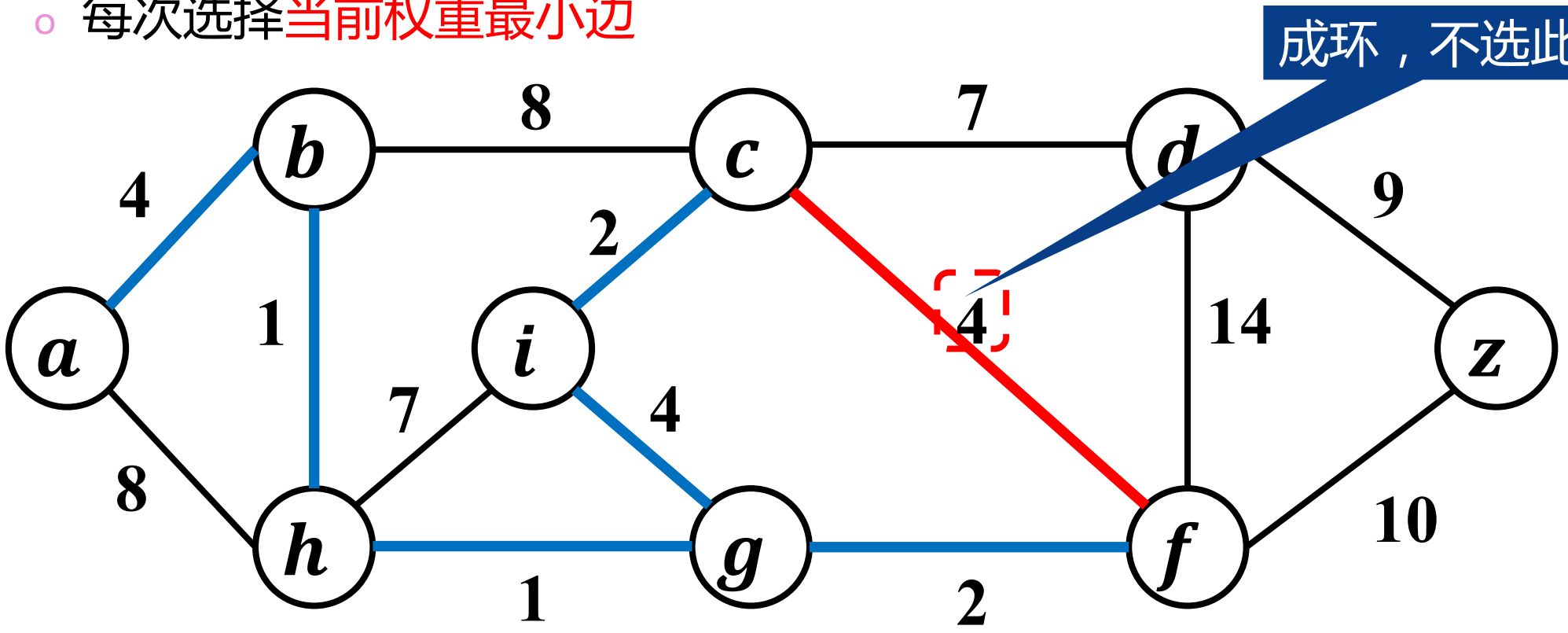
- $\{(a,b)(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共3个生成子树



算法实例

- 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

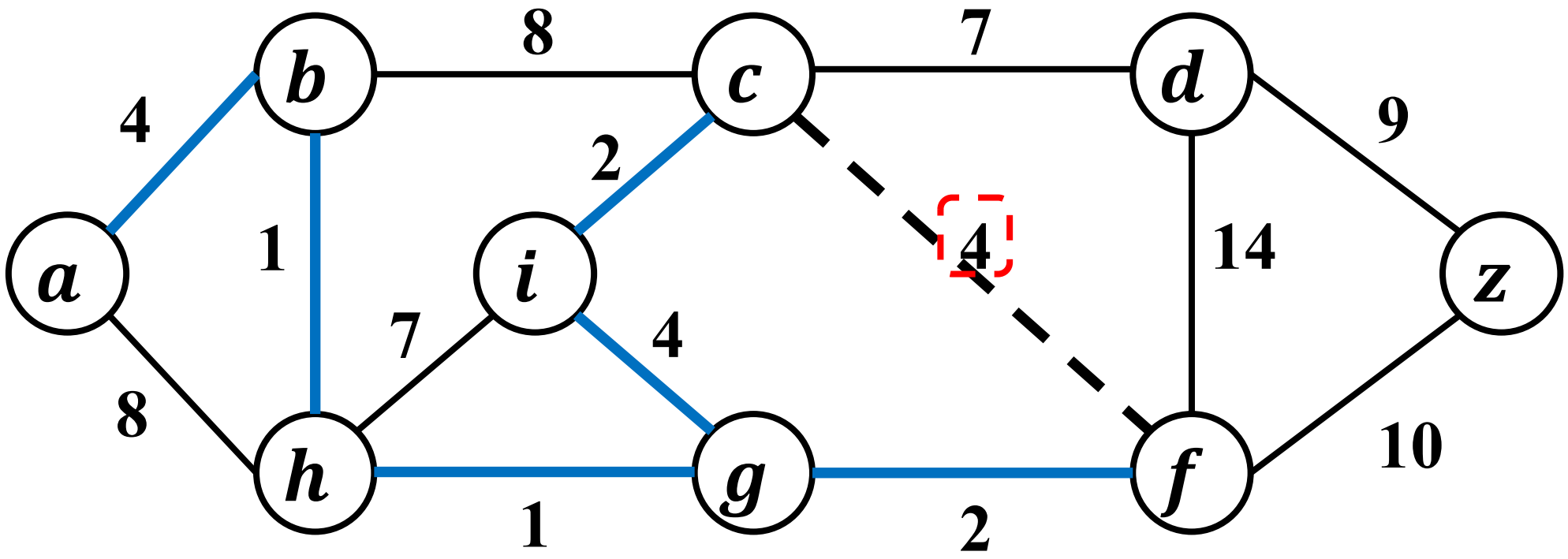
- $\{(a,b)(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共3个生成子树



算法实例

- 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

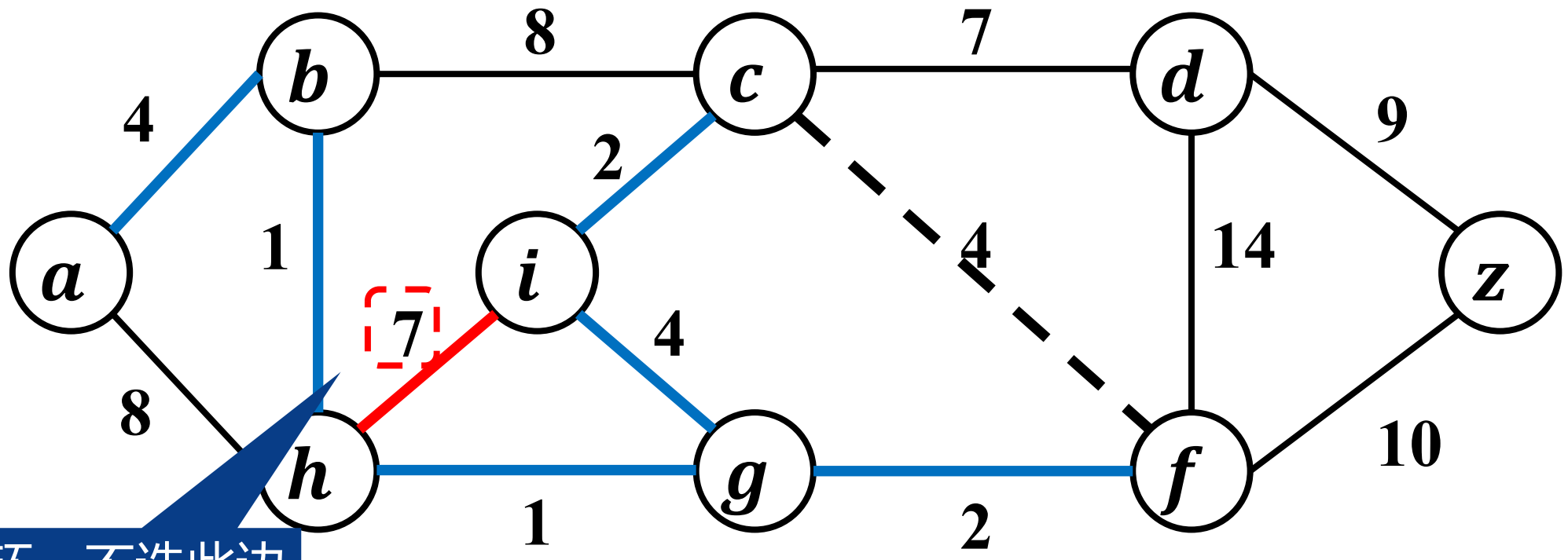
- $\{(a,b)(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共3个生成子树



算法实例

- 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

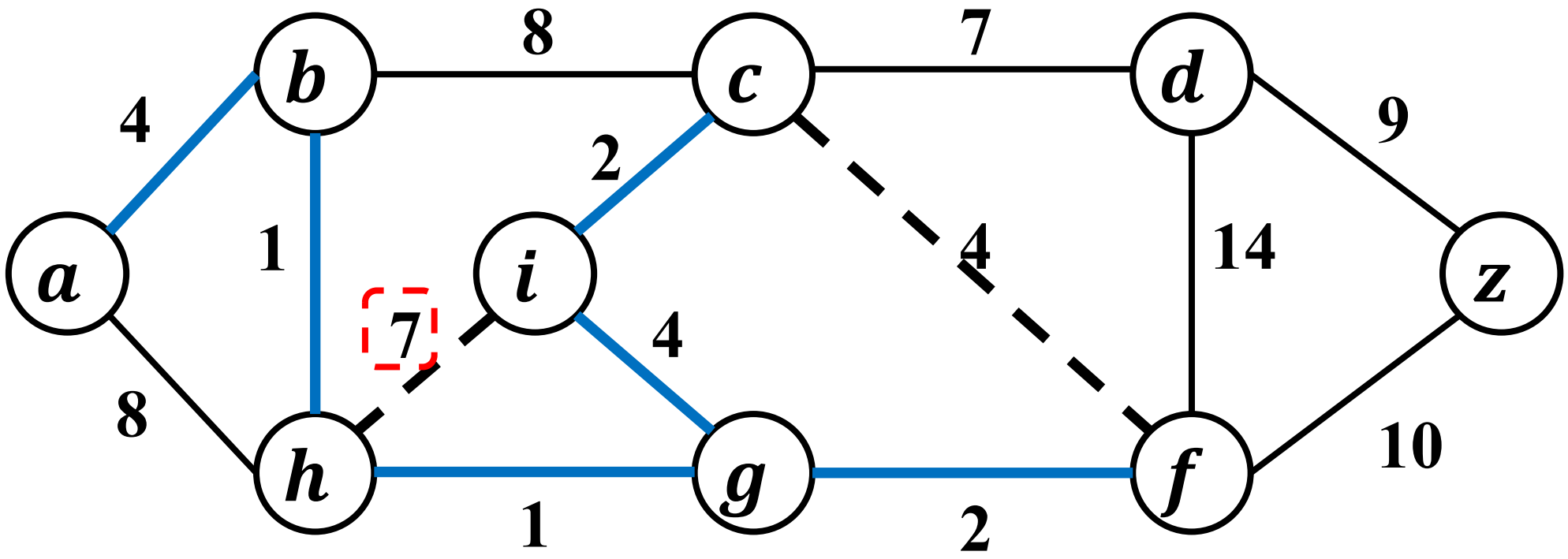
- $\{(a,b)(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共3个生成子树



算法实例

- 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

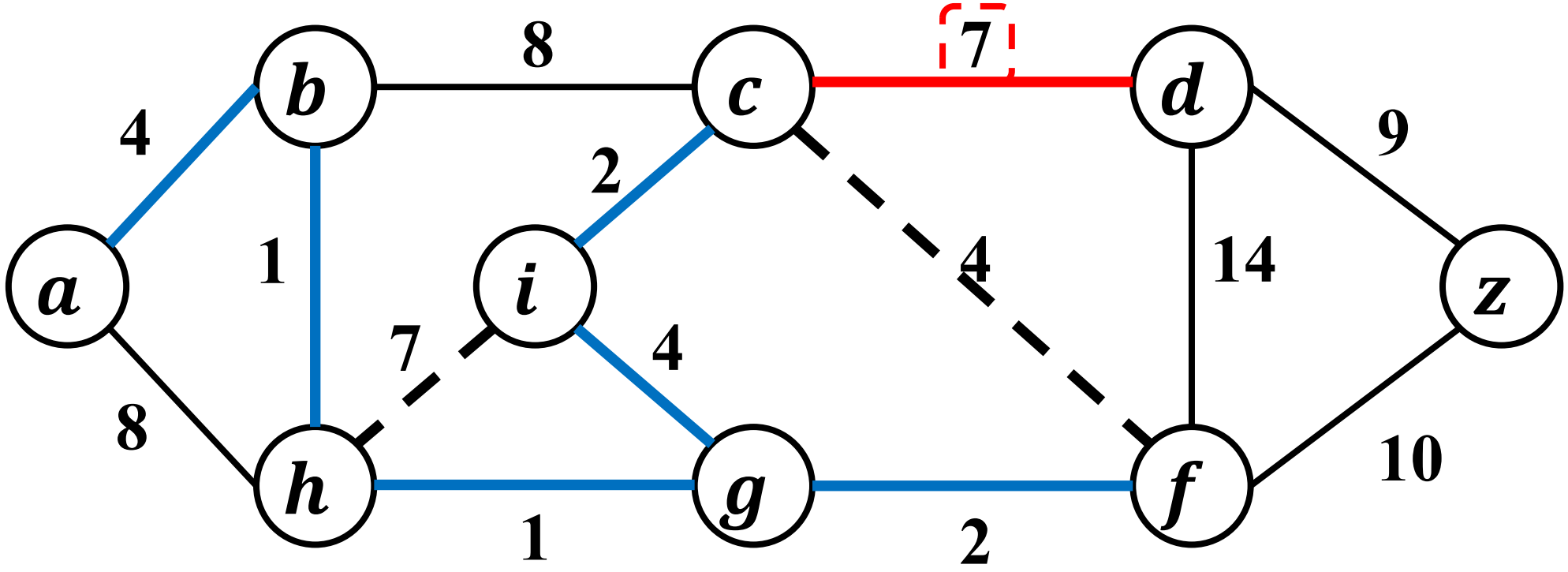
- $\{(a,b)(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共3个生成子树



算法实例

- 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

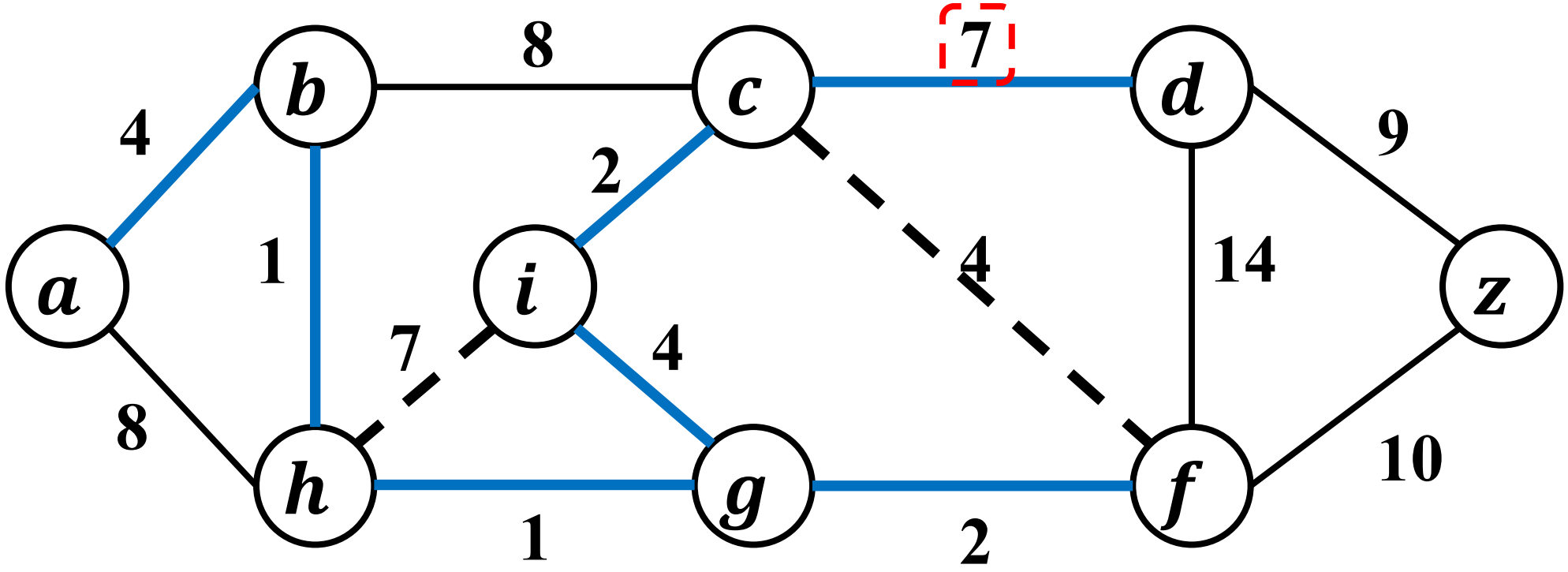
- $\{(a,b)(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共3个生成子树



算法实例

- 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

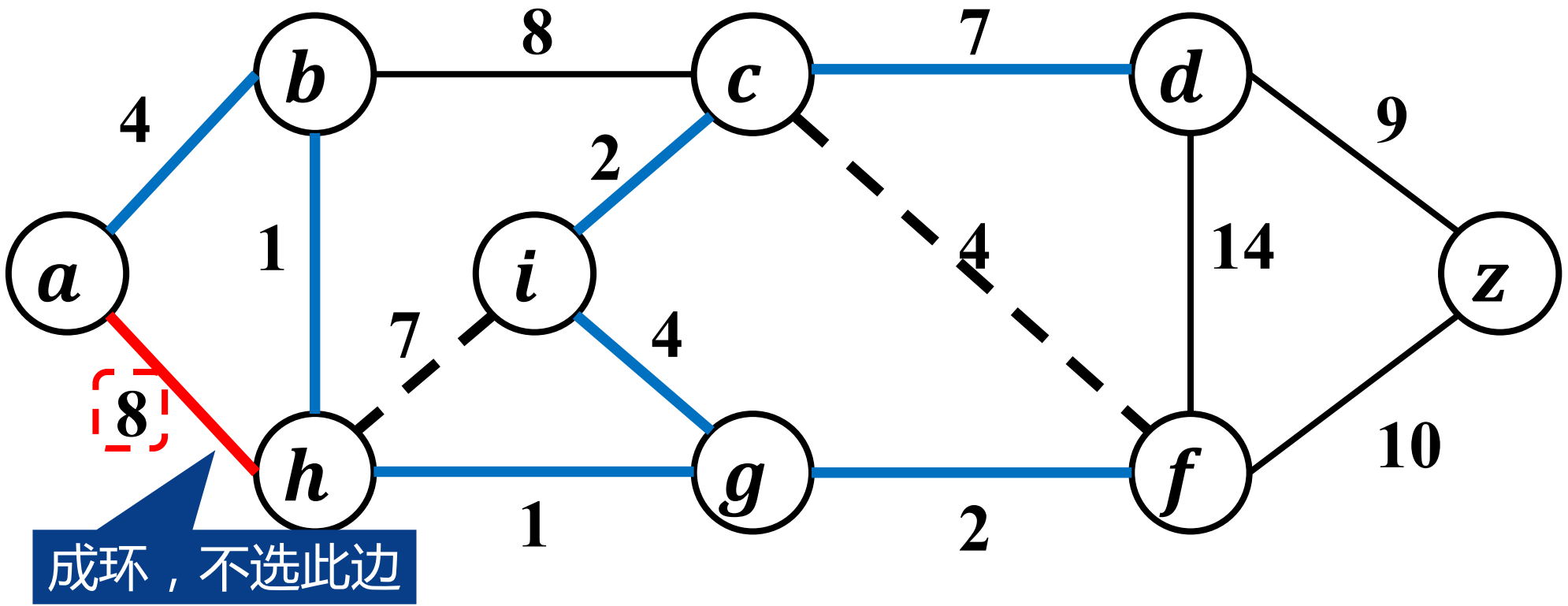
- $\{(a,b),(b,h),\{h,g\},(g,f),(i,c),(i,g),(c,d)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共2个生成子树



算法实例

- 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

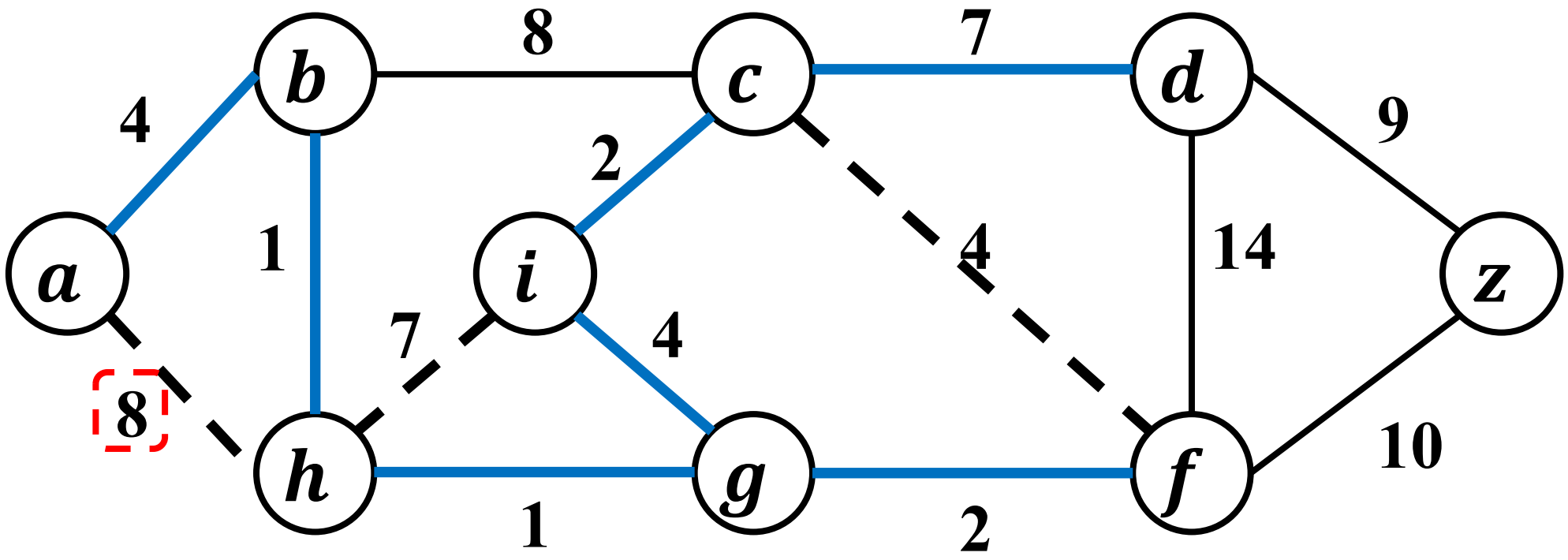
- $\{(a,b),(c,d),(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共2个生成子树



算法实例

- 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

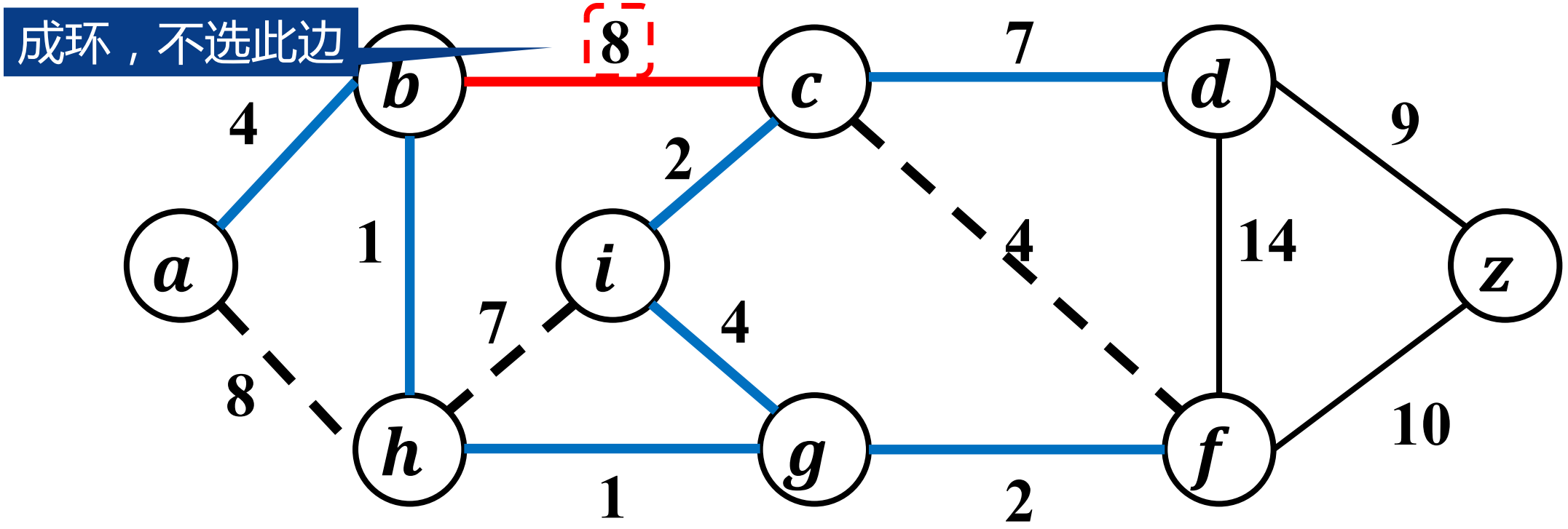
- $\{(a,b),(c,d),(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共2个生成子树



算法实例

- 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

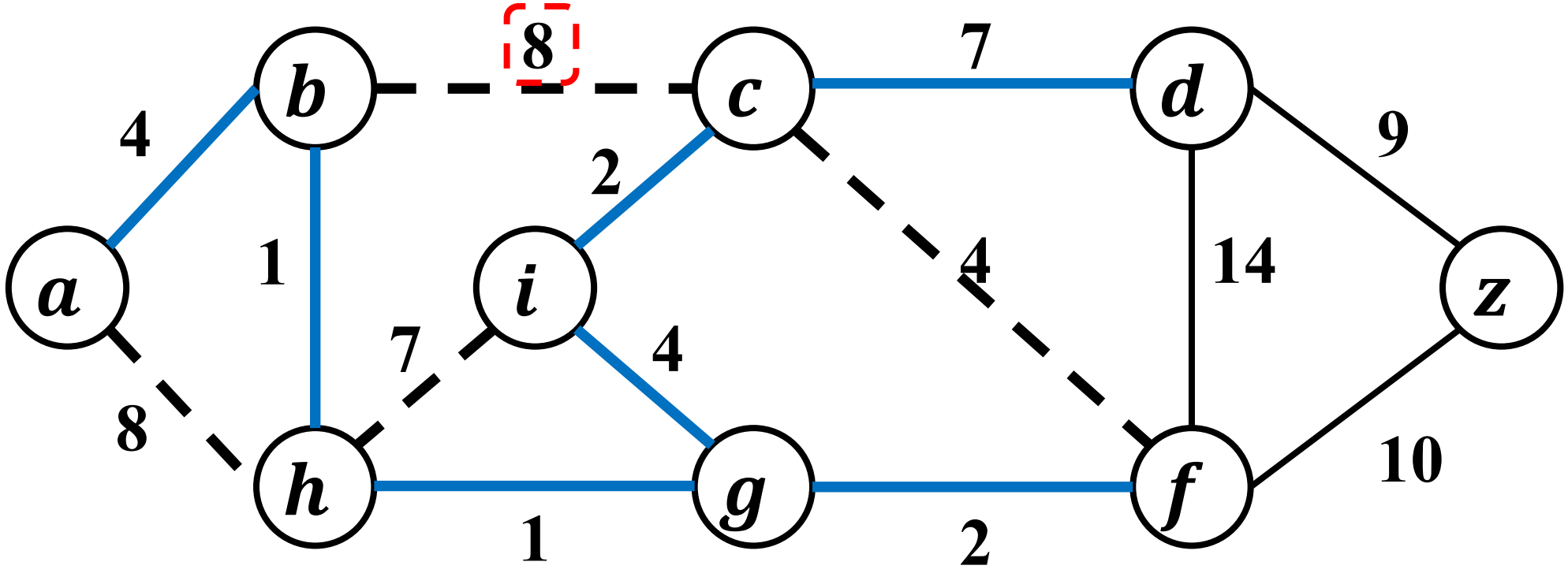
- $\{(a,b),(c,d),(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共2个生成子树



算法实例

- 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

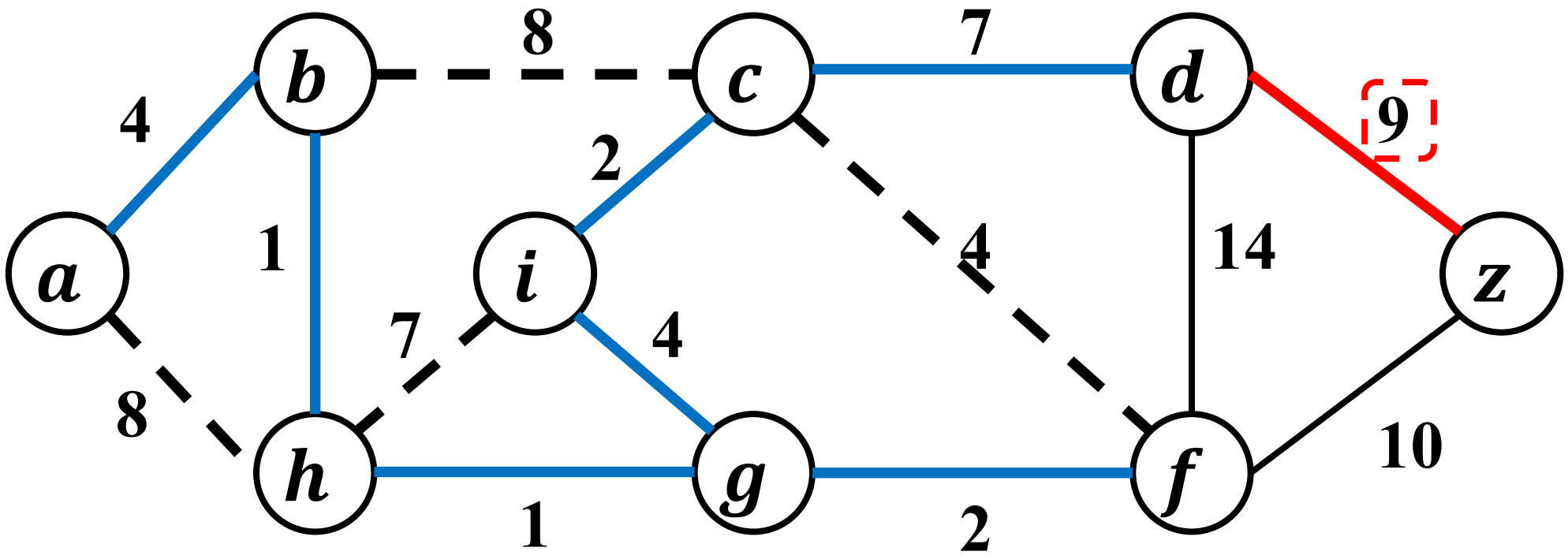
- $\{(a,b),(c,d),(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共2个生成子树



算法实例

- 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

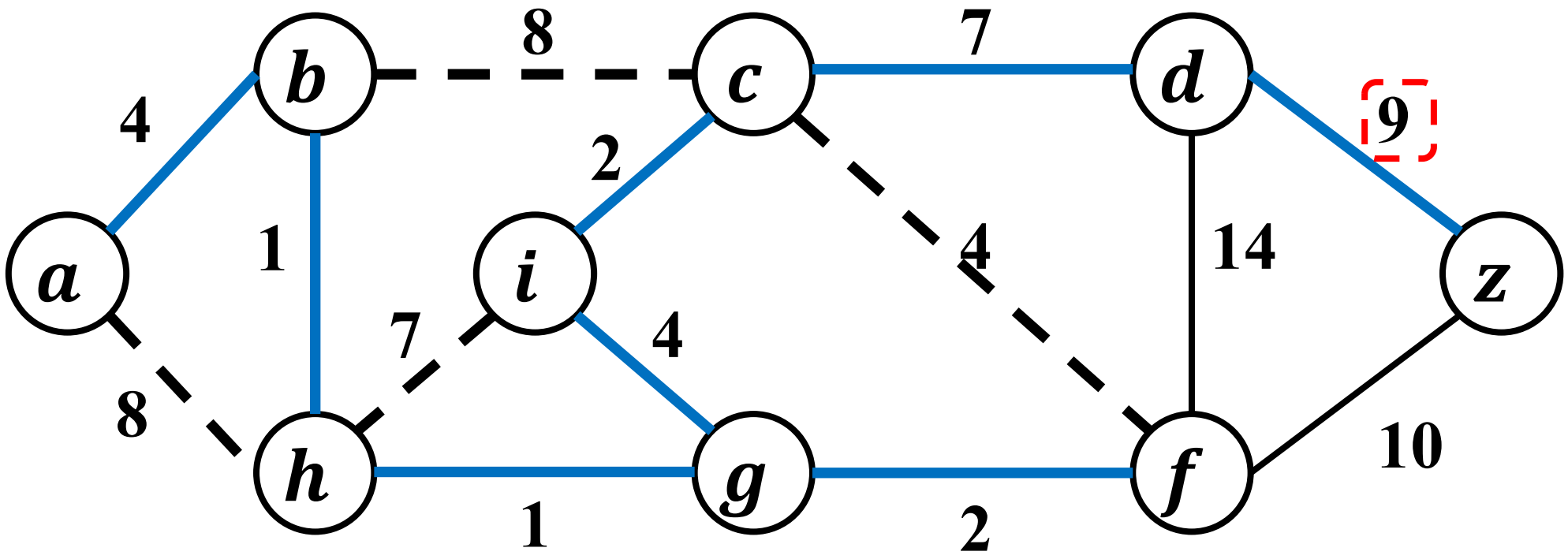
- $\{(a,b),(c,d),(b,h),\{h,g\},(g,f),(i,c),(i,g)\}$ 构成一棵生成子树，其他顶点每个是一个生成子树
- 一共2个生成子树



算法实例

- 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

- $\{(a,b),(c,d),(b,h),\{h,g\},(g,f),(i,c),(i,g),(d,z)\}$ 构成一棵生成子树，
- 一共1个生成子树，即生成树

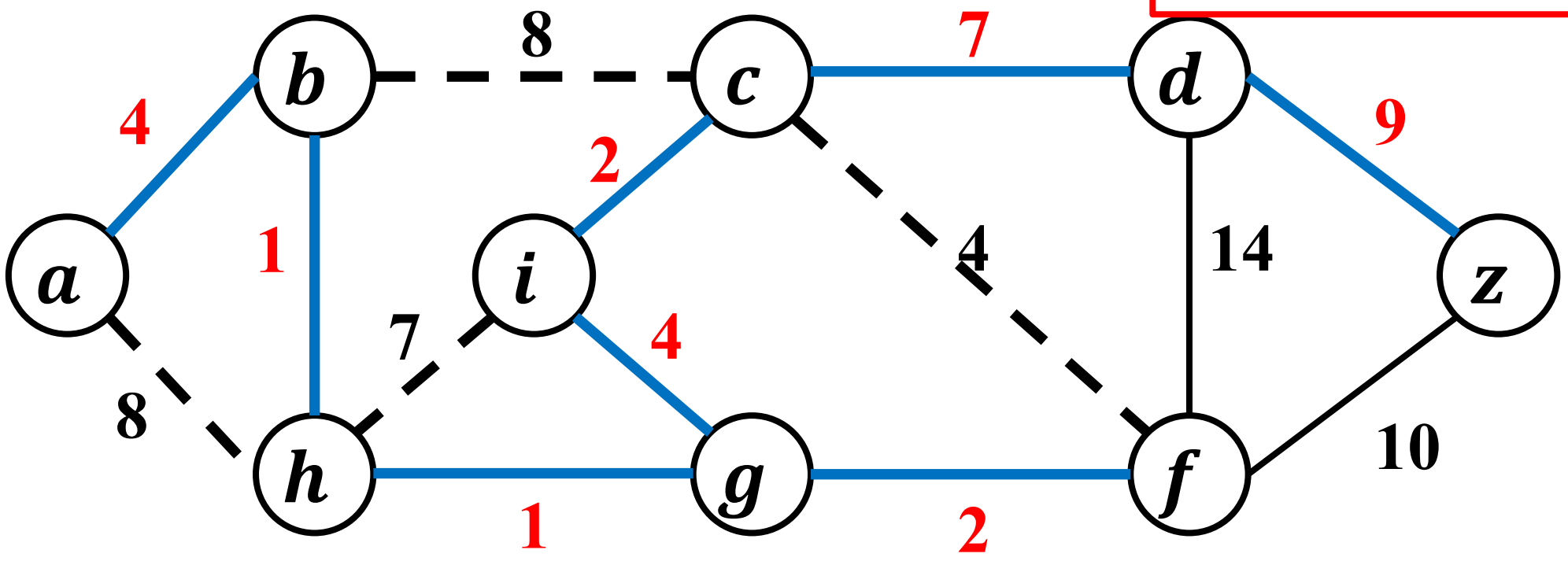


算法实例

- 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

- $\{(a,b),(c,d),(b,h),\{h,g\},(g,f),(i,c),(i,g),(d,z)\}$ 构成一棵生成子树，
- 一共1个生成子树，即生成树

最小生成树权重：30



算法思想精炼

- **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
```

→ 不成环的最小边，是一种贪心策略

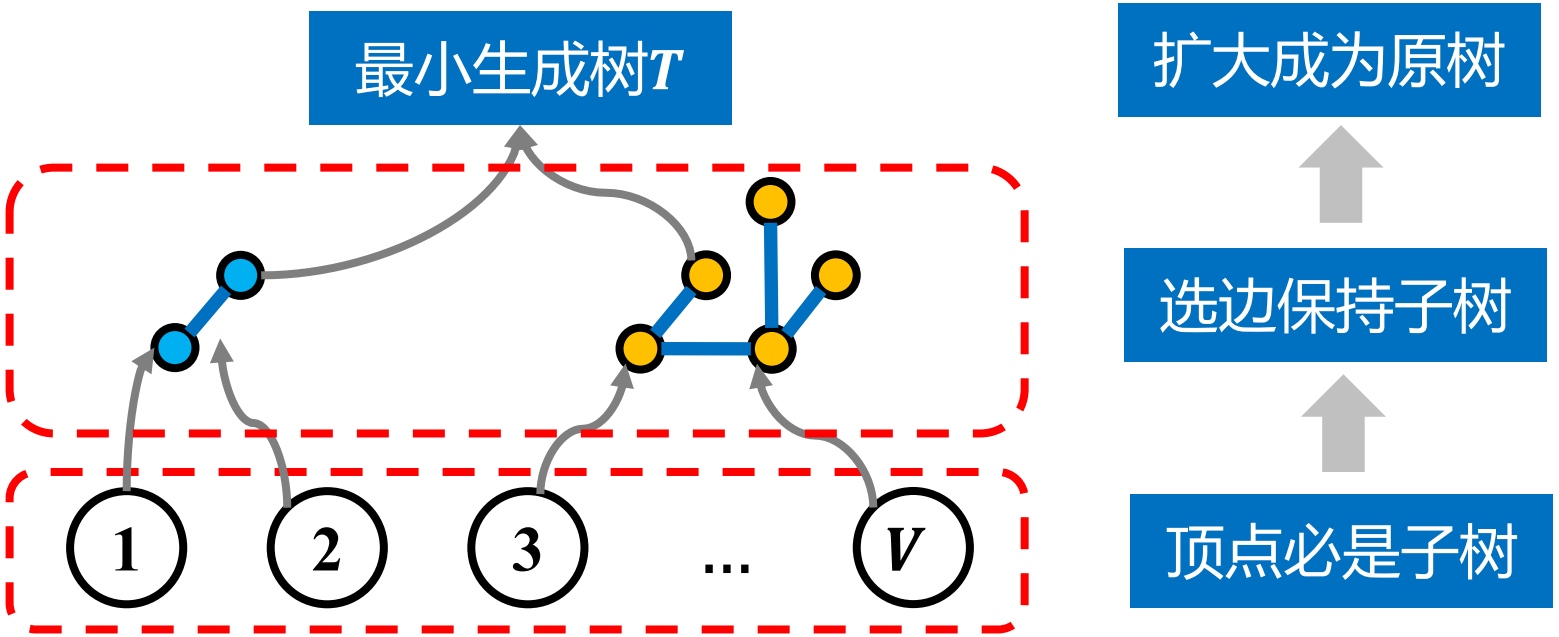
算法思想精炼

- **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
```

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



算法思想精炼

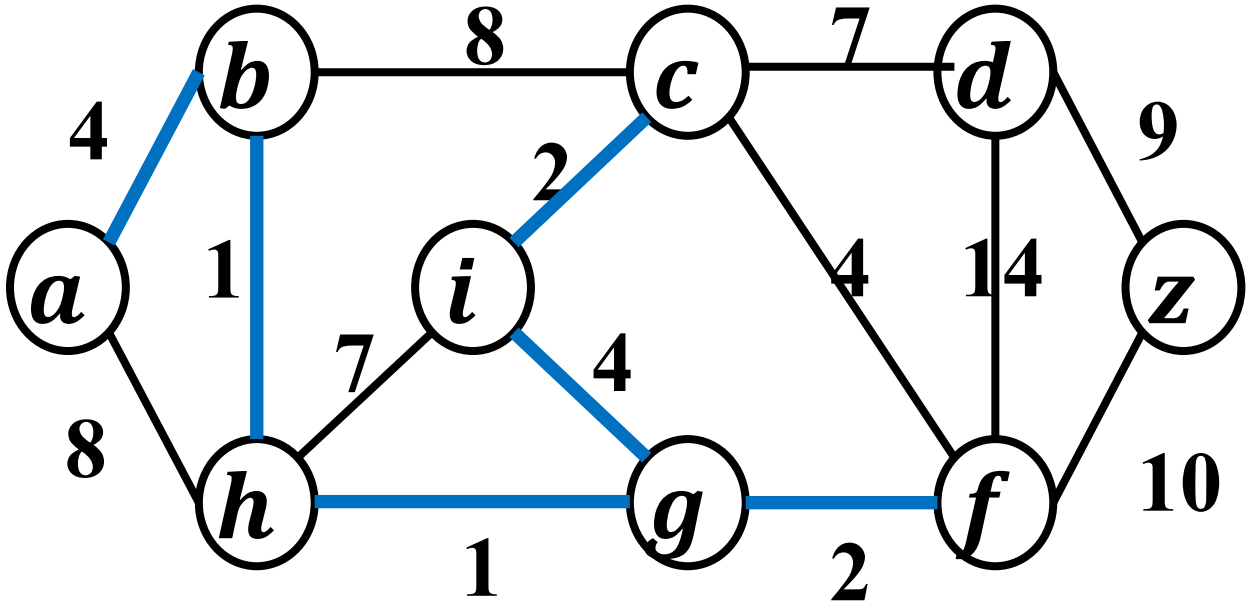
- **Generic-MST(G)**

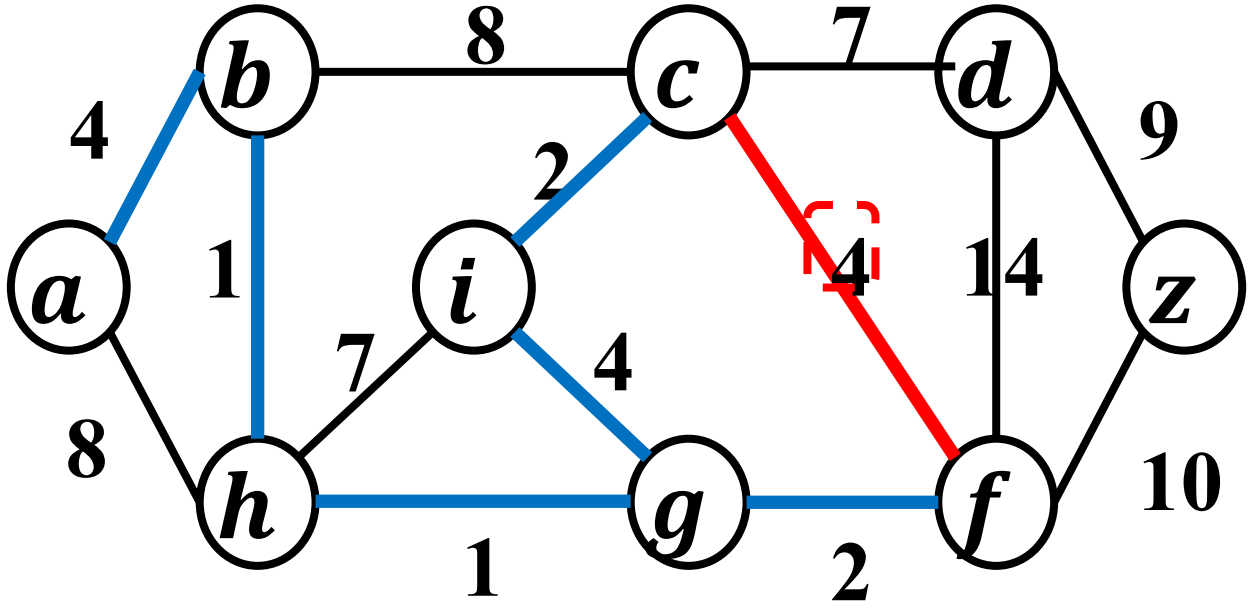
```
A ← ∅
while 没有形成最小生成树 do
    寻找A的安全边(u, v)
    A ← A ∪ (u, v)
end
return A
```

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





算法思想精炼

- **Generic-MST(G)**

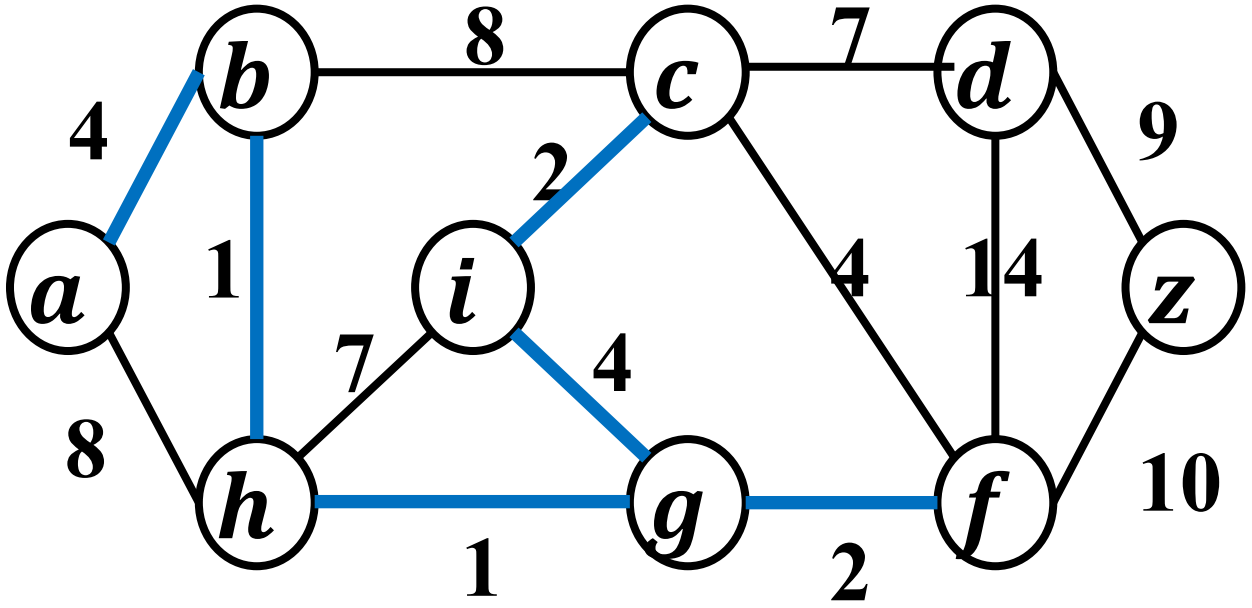
```
A ← ∅
while 没有形成最小生成树 do
    寻找A的安全边(u, v)
    A ← A ∪ (u, v)
end
return A
```

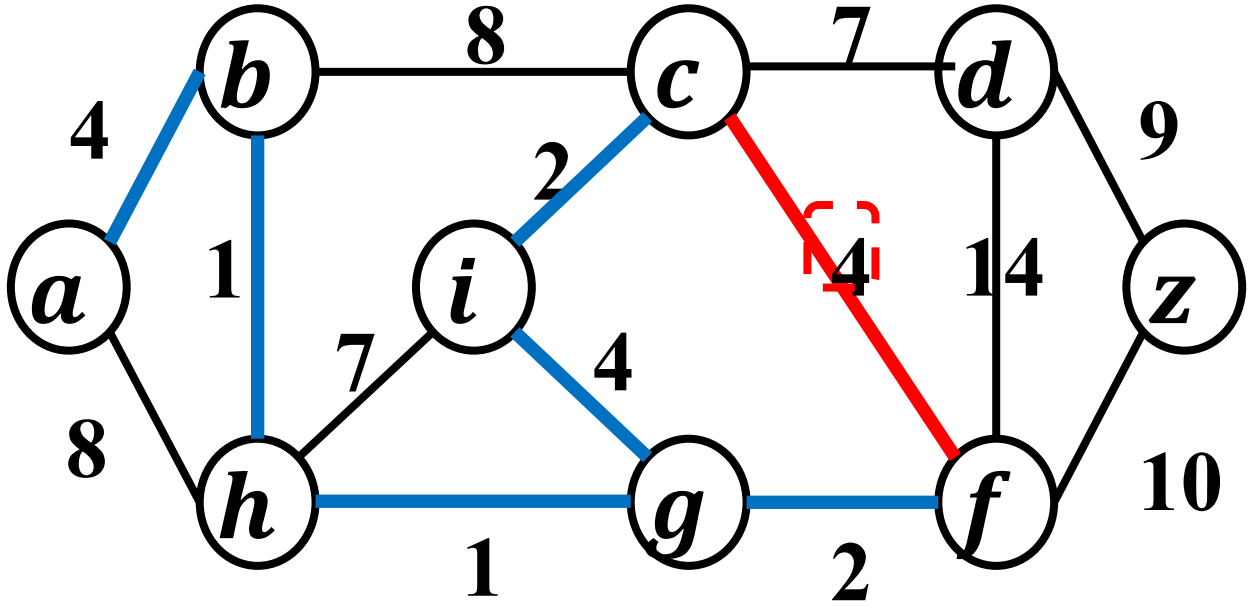
算法正确性的关键

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





提纲

问题的回顾

算法与实例

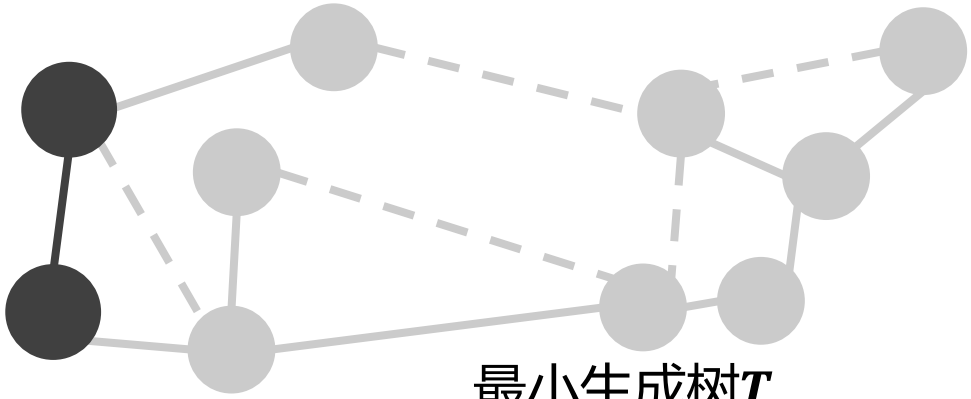
正确性证明

不相交集合

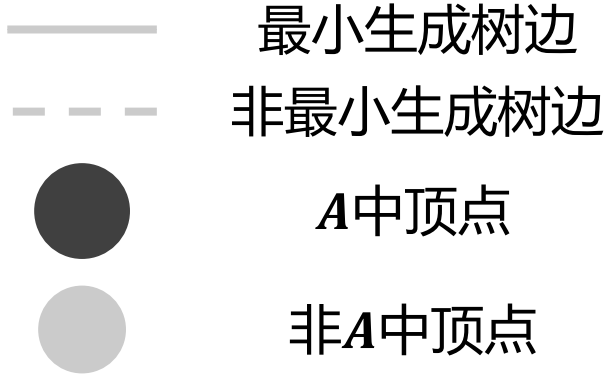
复杂度分析

贪心策略原理回顾

- 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

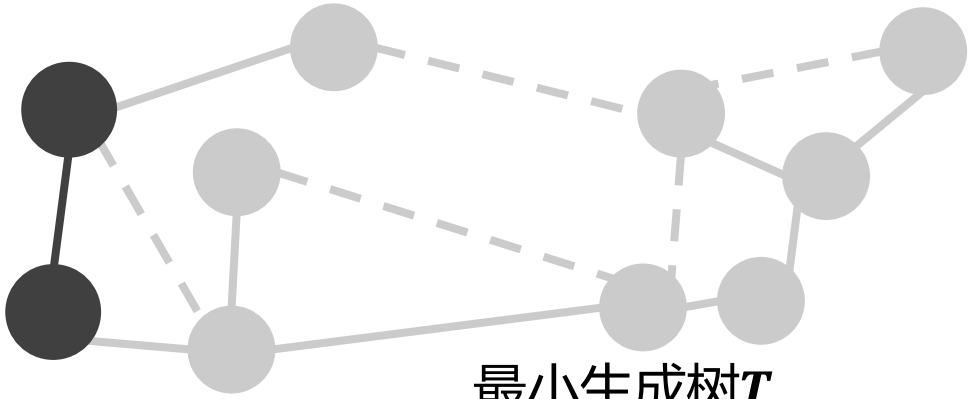


最小生成树T

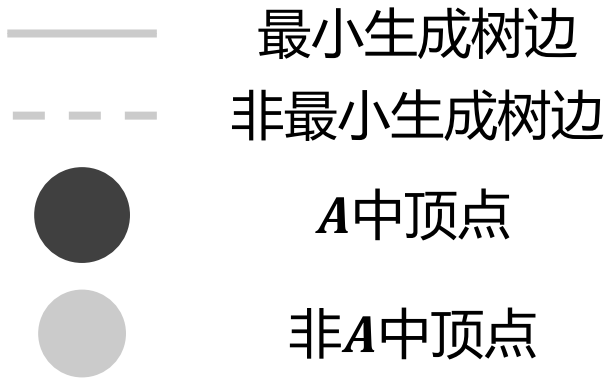


贪心策略原理回顾

- 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



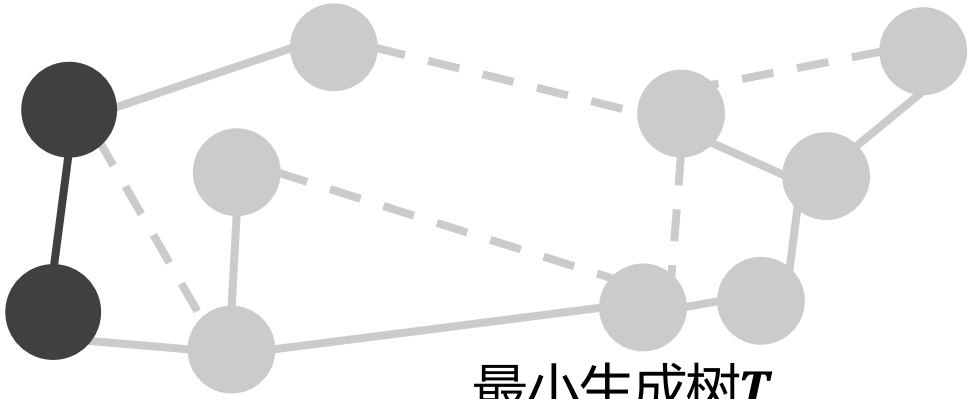
最小生成树T



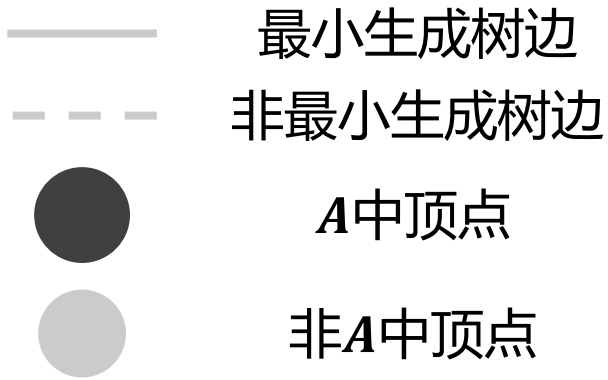
- 若每次向边集 A 中新增权重最小的安全边，可保证边集A是最小生成树的子集

贪心策略原理回顾

- 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



最小生成树T



- 若每次向边集 A 中新增权重最小的安全边，可保证边集A是最小生成树的子集

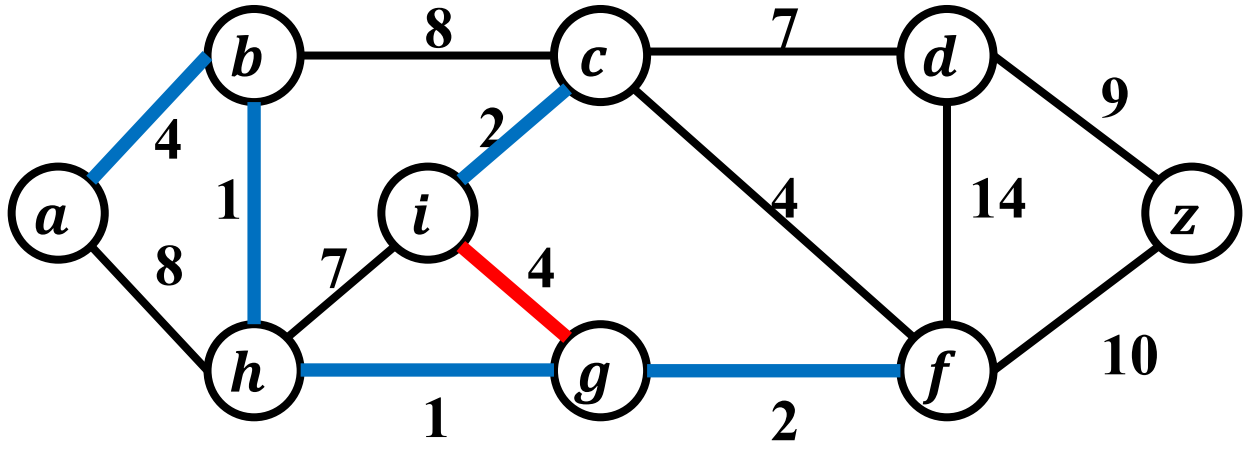
问题：Kruskal算法选边策略能否保证每次都选择了安全边？

正确性证明

- **Kruskal**算法选边策略能否保证每次都选择了安全边？——能！

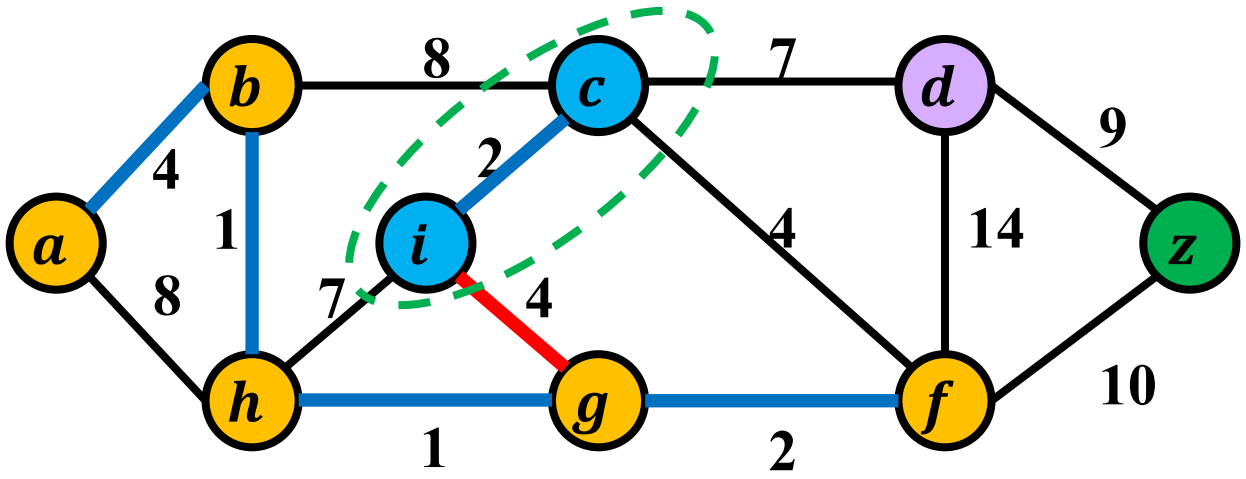
正确性证明

- **Kruskal**算法选边策略能否保证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)



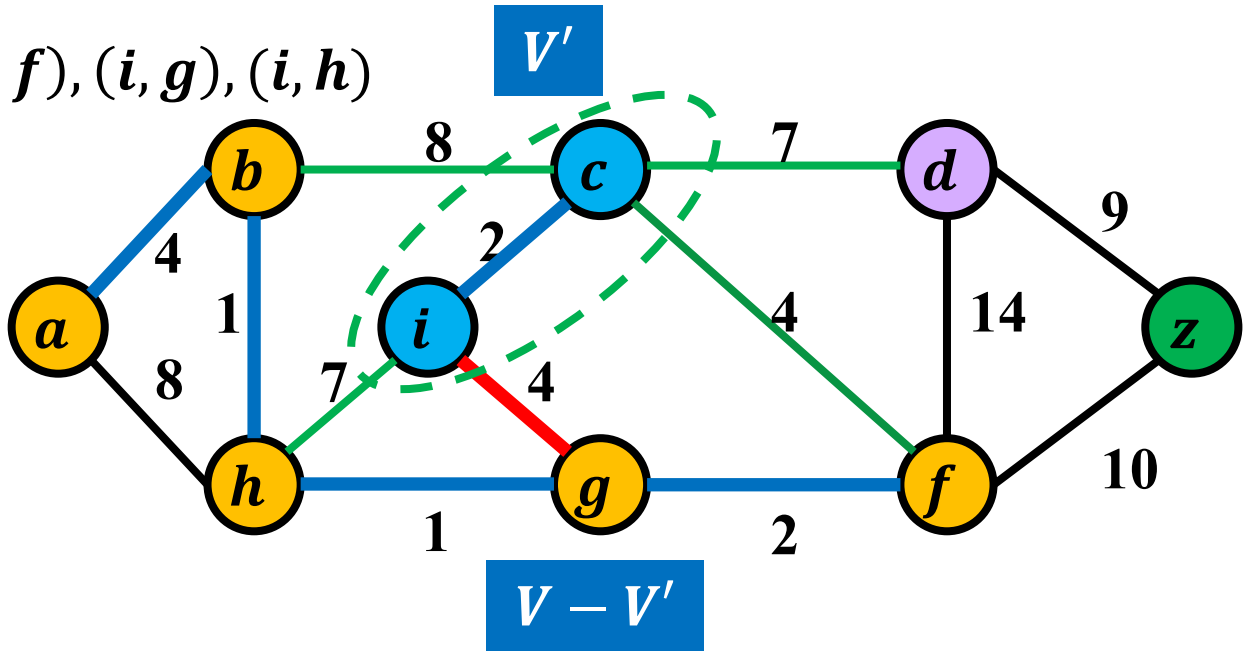
正确性证明

- **Kruskal**算法选边策略能否保证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



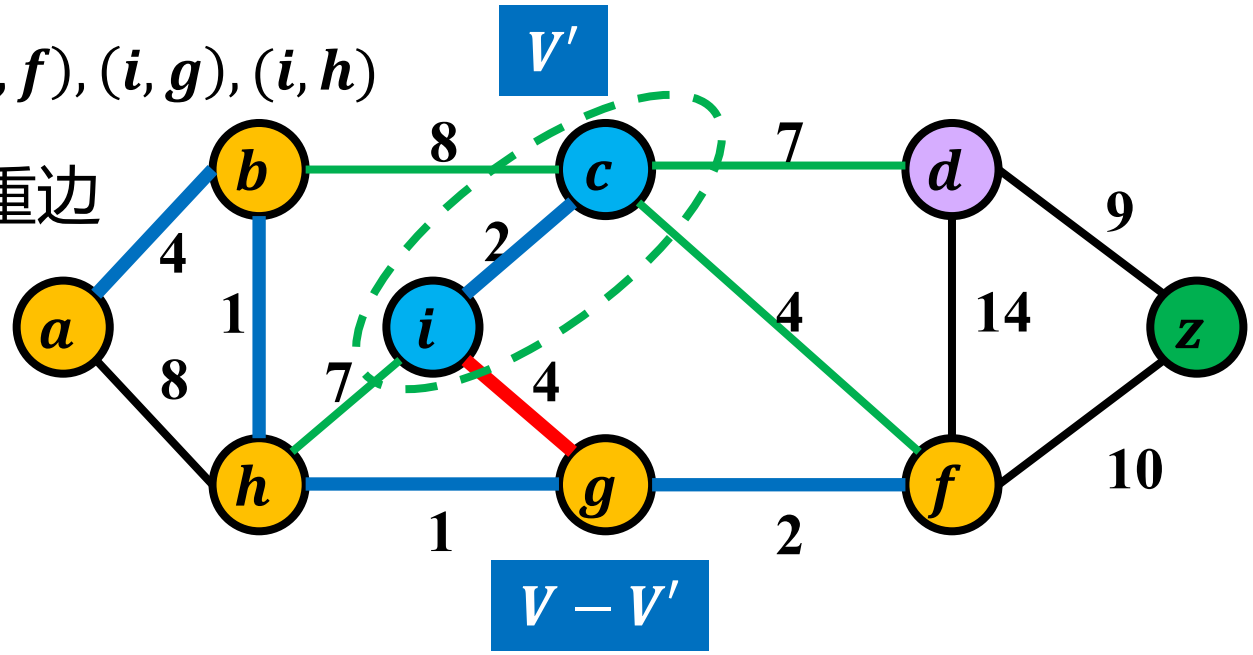
正确性证明

- **Kruskal**算法选边策略能否保证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 $(b,c),(c,d),(c,f),(i,g),(i,h)$



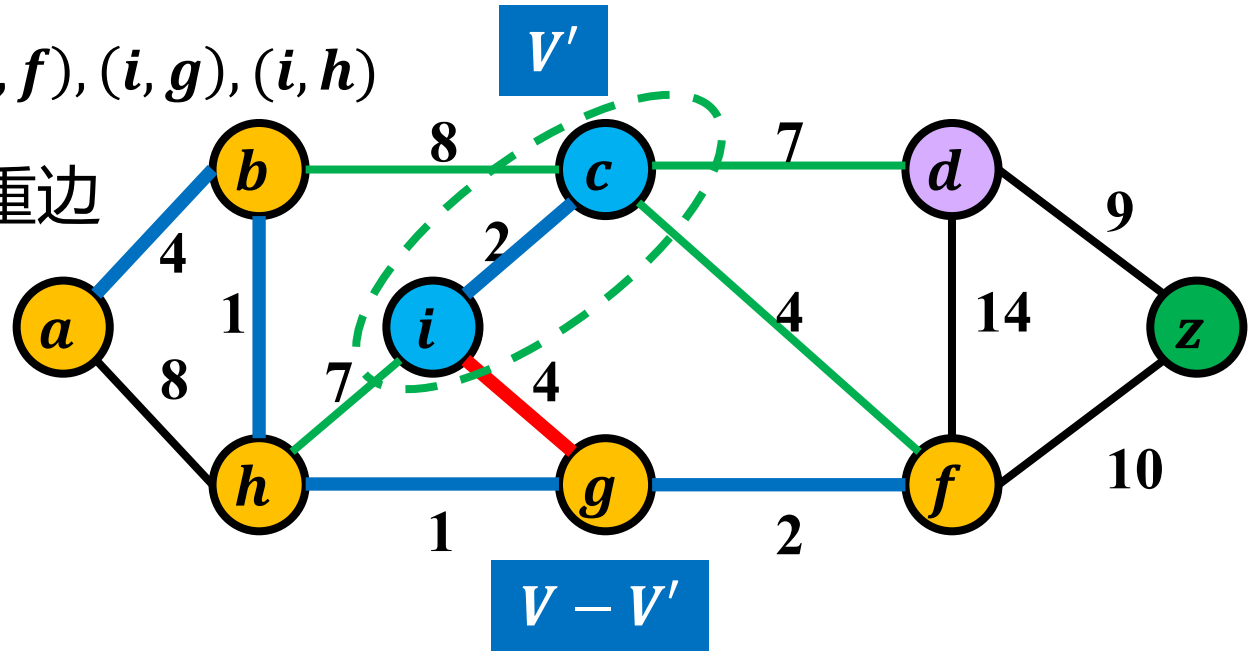
正确性证明

- **Kruskal**算法选边策略能否保证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**算法选边策略能否保证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



伪代码

- **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 ← {}
for (u, v) ∈ E do
    if u, v 不在同一子树 then
        T ← T ∪ {(u, v)}
        合并u, v所在子树
    end
end
return T
```

按权重排序

伪代码

- **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 ← {}
for (u, v) ∈ E do
    if u, v 不在同一子树 then
        T ← T ∪ {(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

伪代码

- **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 ← {}
for (u, v) ∈ E do
    if u, v 不在同一子树 then
        T ← T ∪ {(u, v)}
        合并u, v所在子树
    end
end
return T
```

加入该边不成环

伪代码

- **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

伪代码

- **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 ← {}
for (u, v) ∈ E do
    if u, v 不在同一子树 then
        T ← T ∪ {(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？

提纲

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

不相交集合

- **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

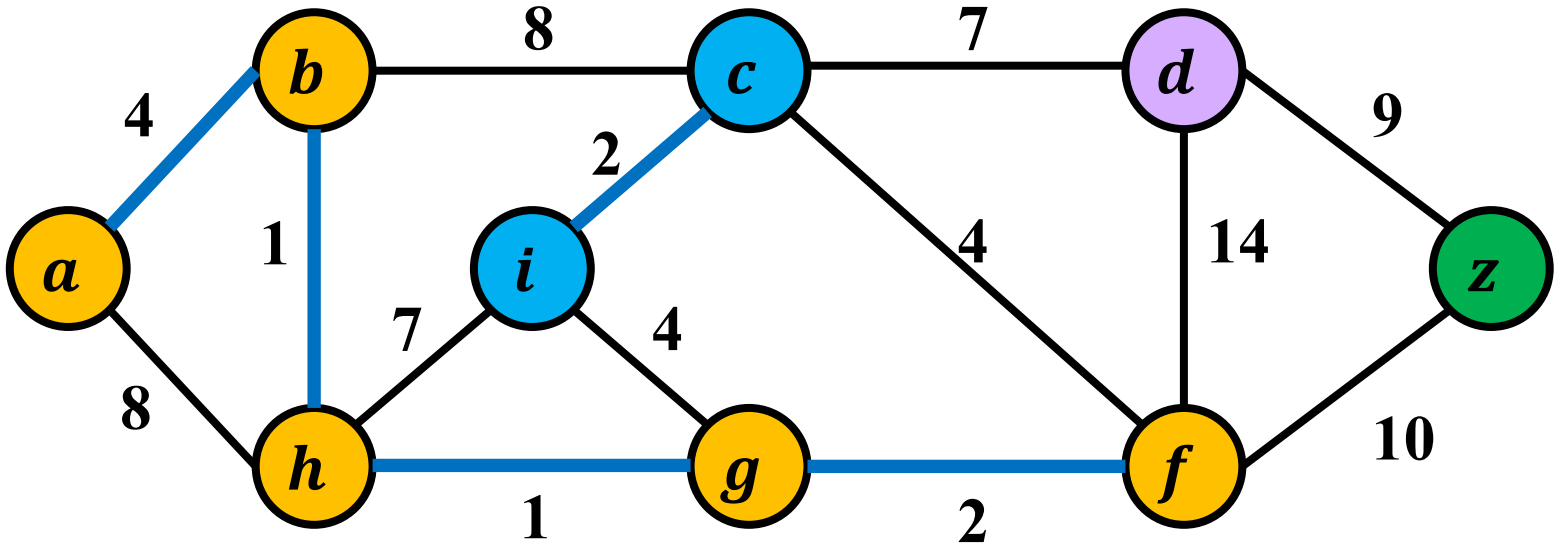
→ 需要高效查找顶点所属子树

→ 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

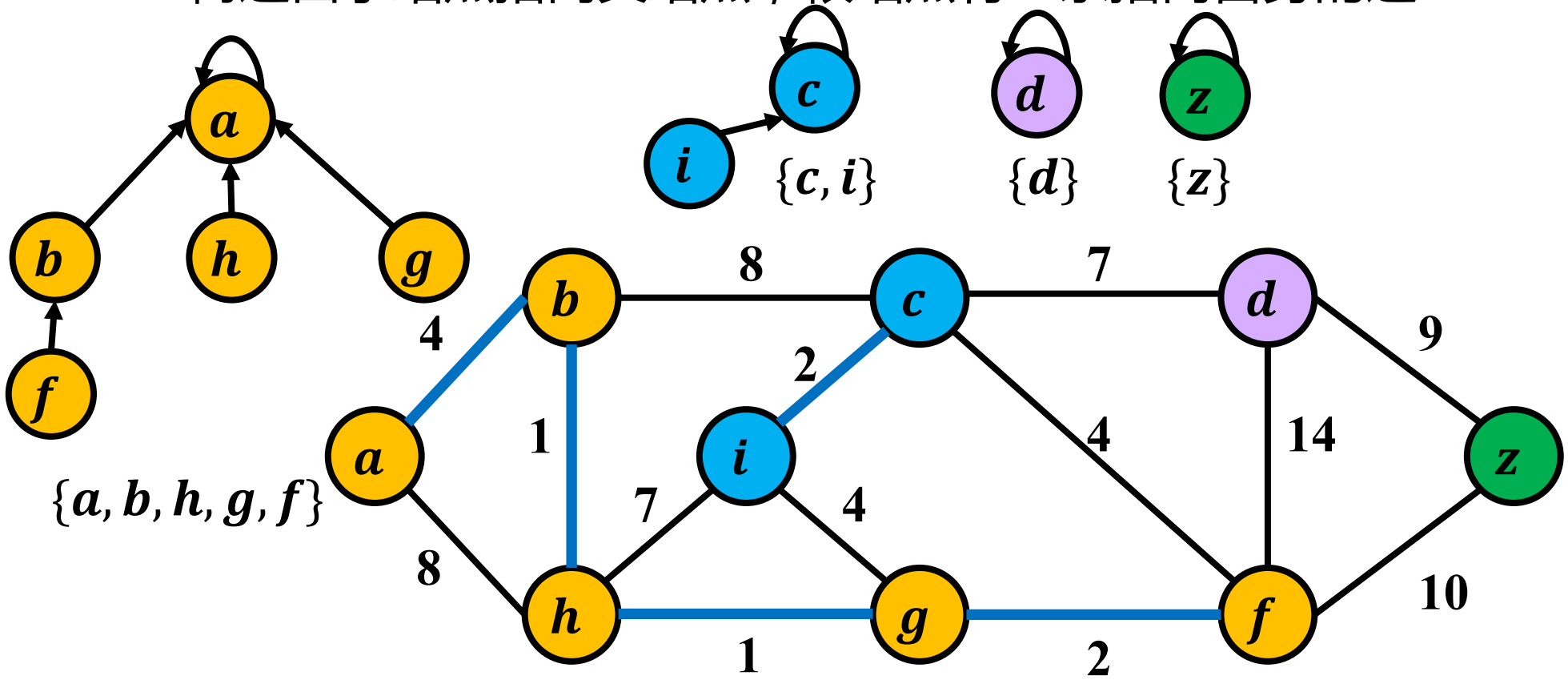
不相交集合

- 把每棵生成子树看作一个顶点集合



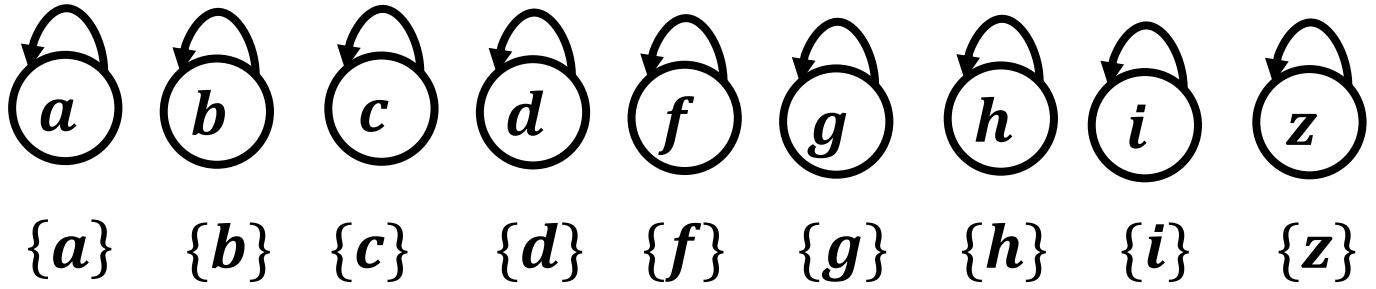
不相交集合

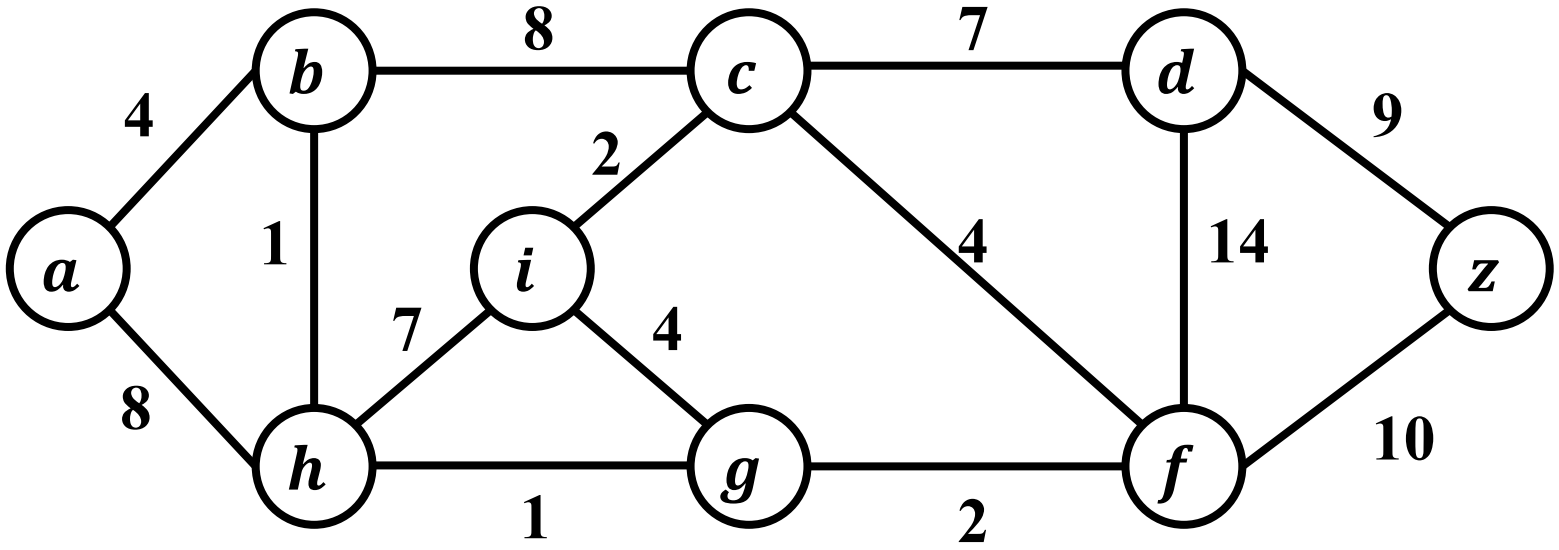
- 把每棵生成子树看作一个顶点集合
  - 每个集合表示为一棵有向树，多个不相交集合构成不相交集合森林
  - 集合元素表示为树结点
  - 树边由子结点指向父结点，根结点有一条指向自身的边



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

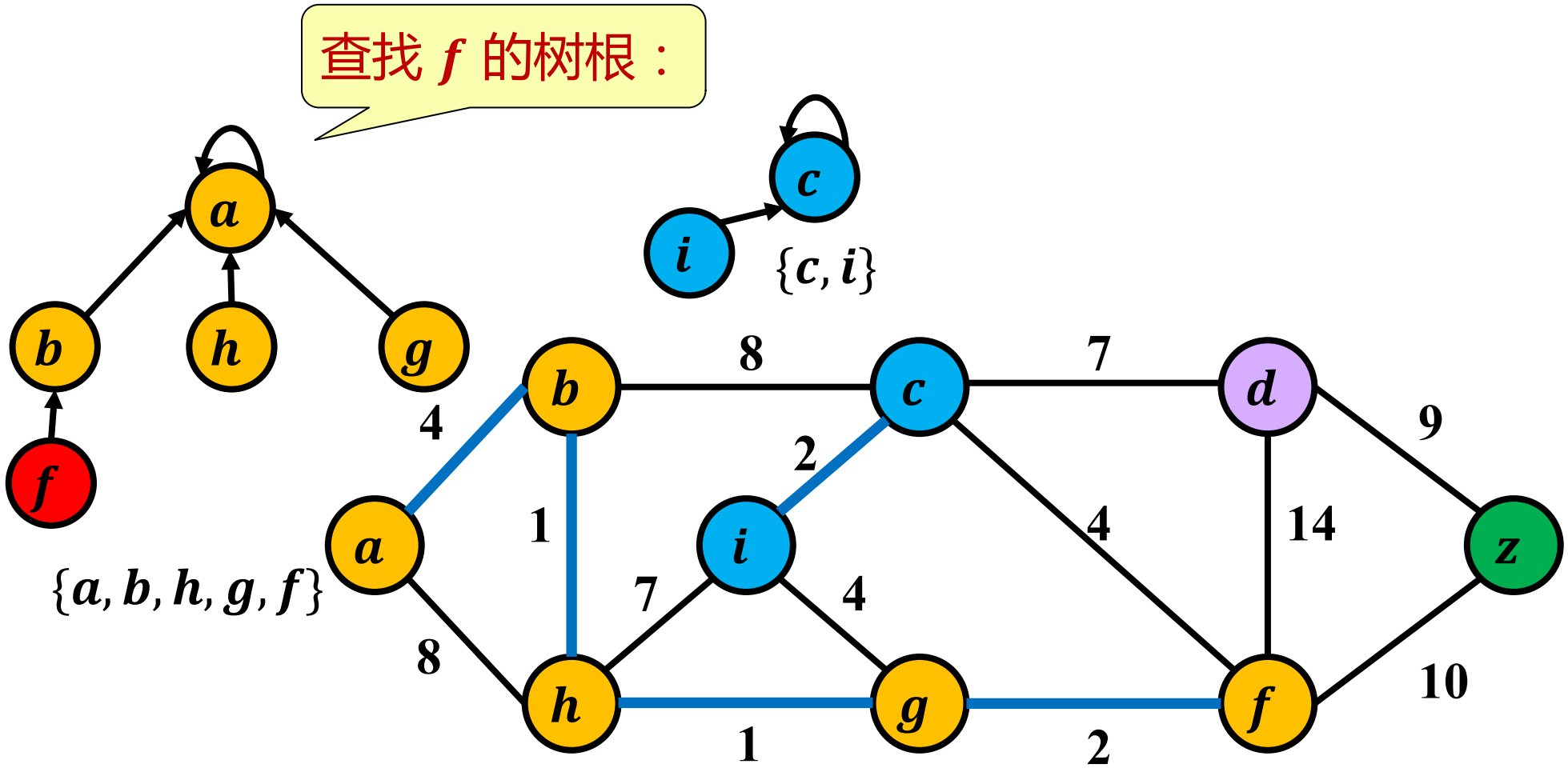
- **Create-Set(x)**

输入: 顶点 x
输出: 并查集
$x.parent \leftarrow x$
return x

自身为树根

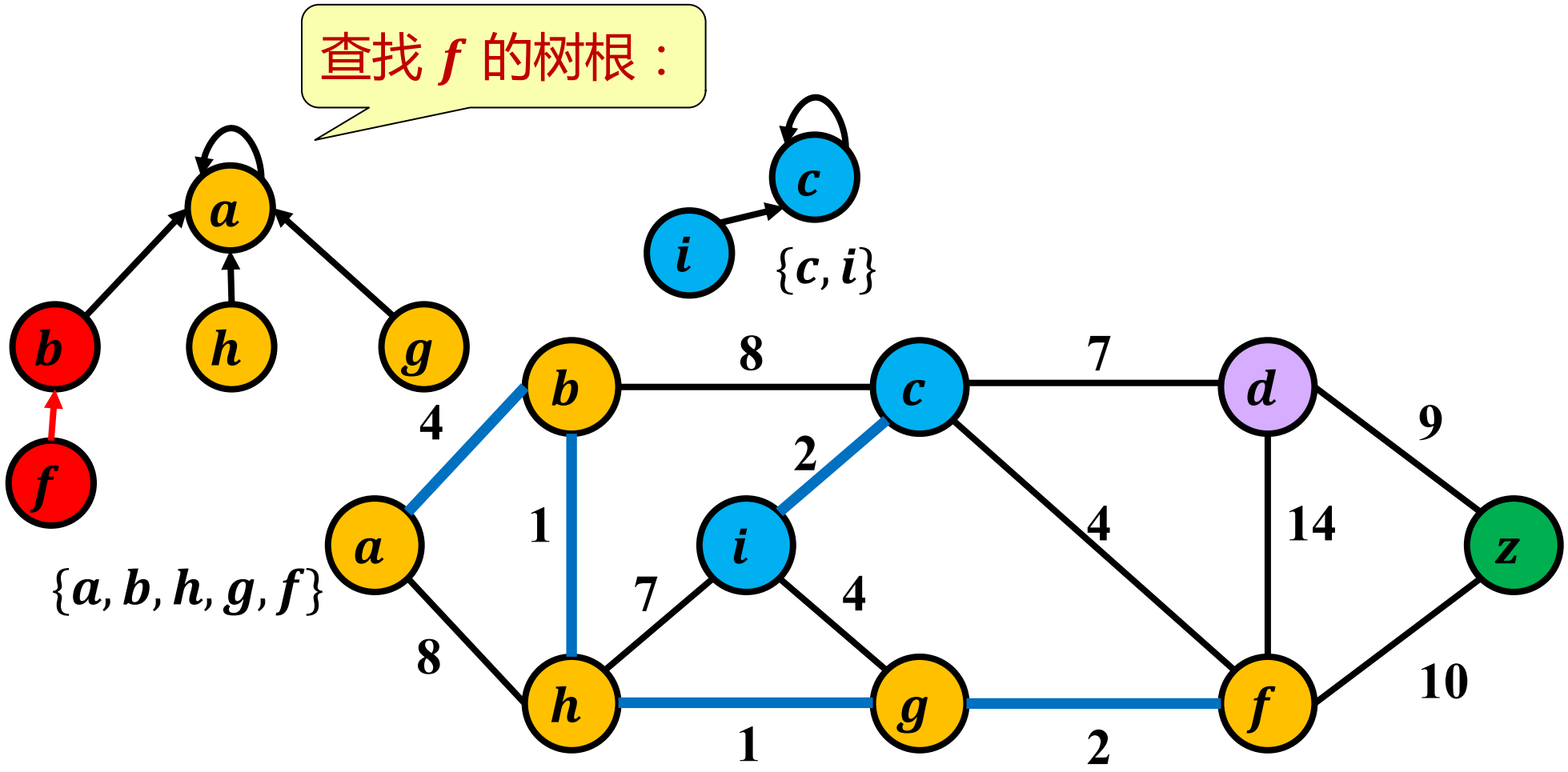
不相交集合

- 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



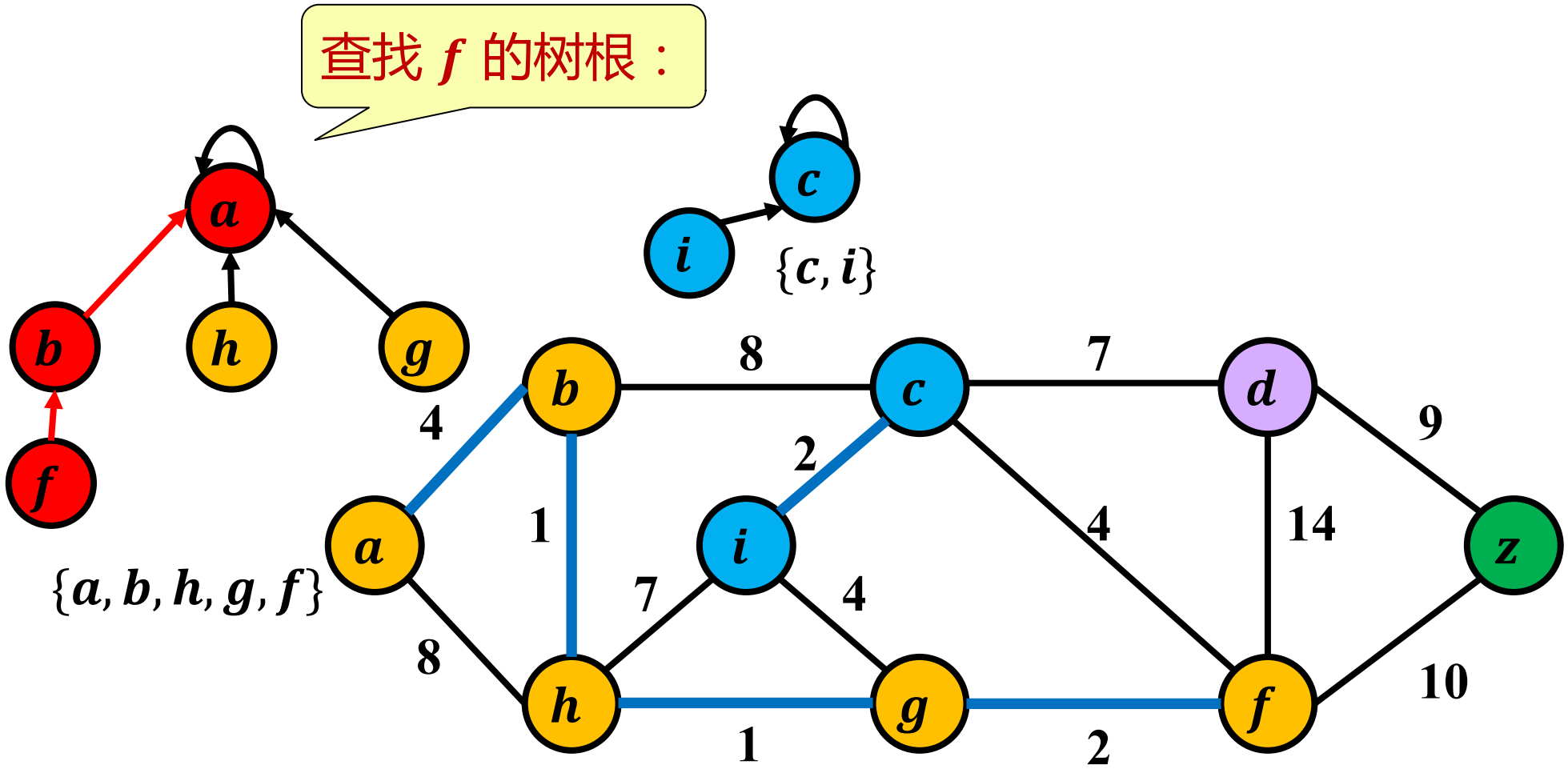
不相交集合

- 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



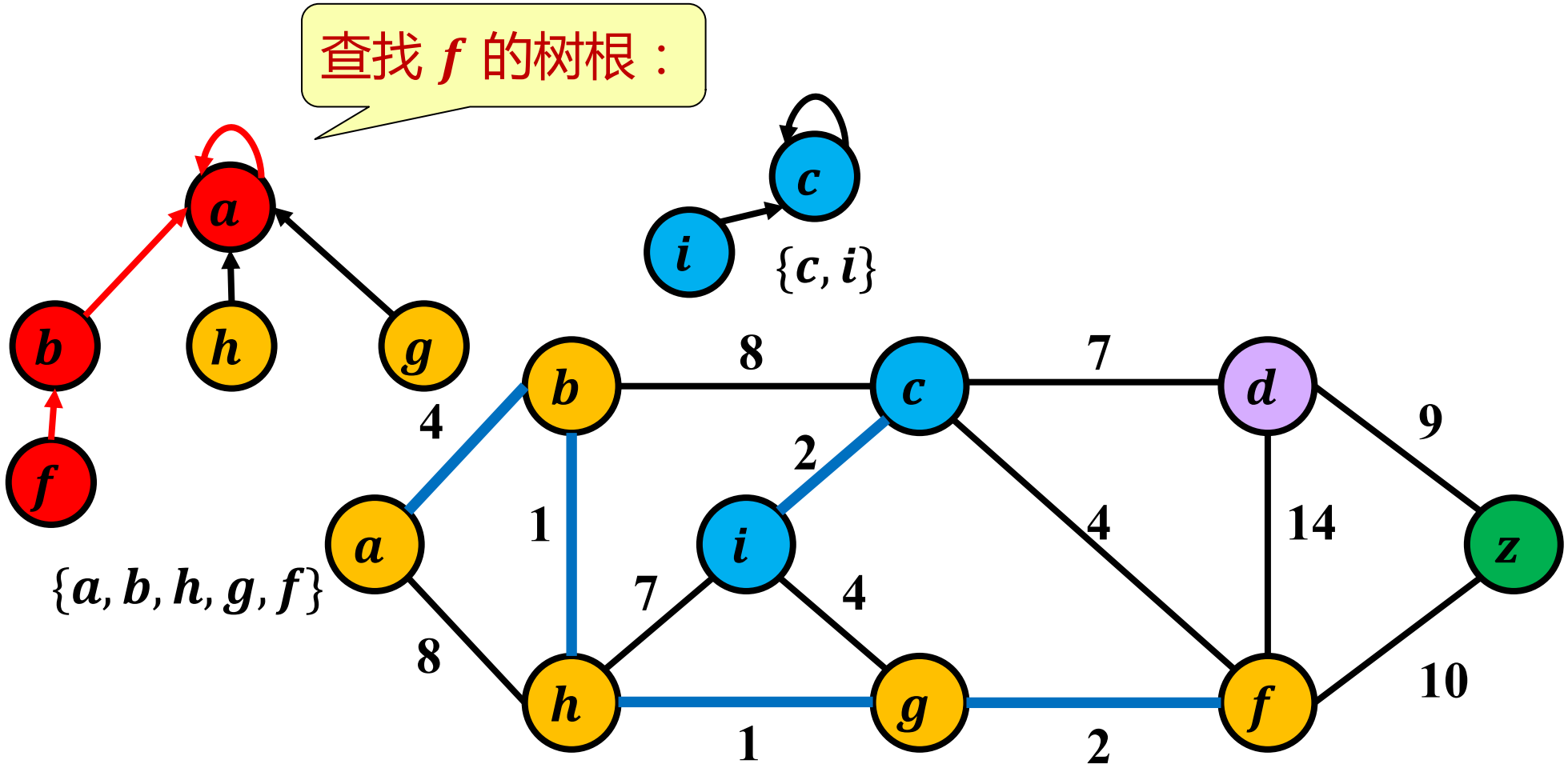
不相交集合

- 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



不相交集合

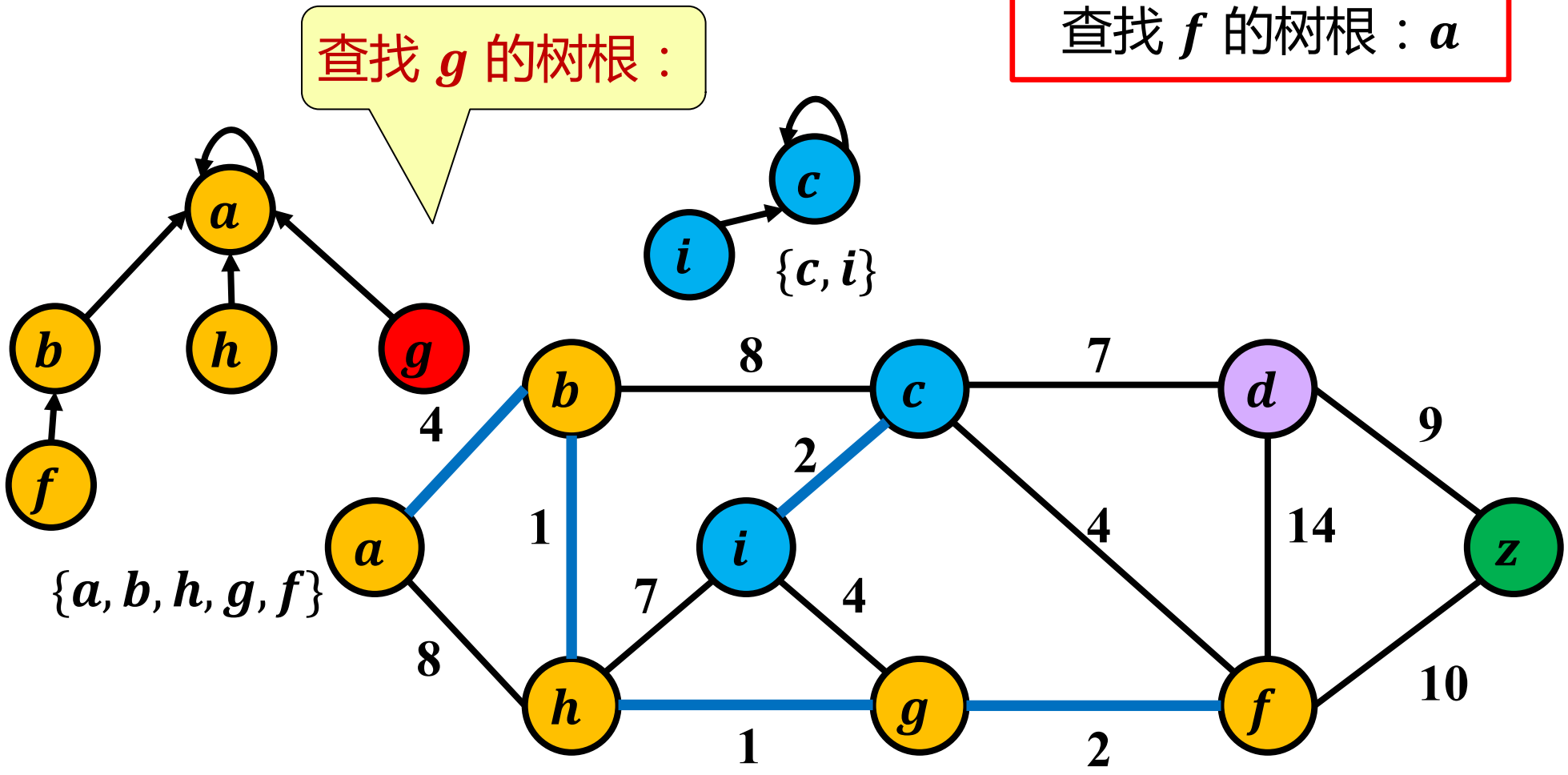
- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同



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

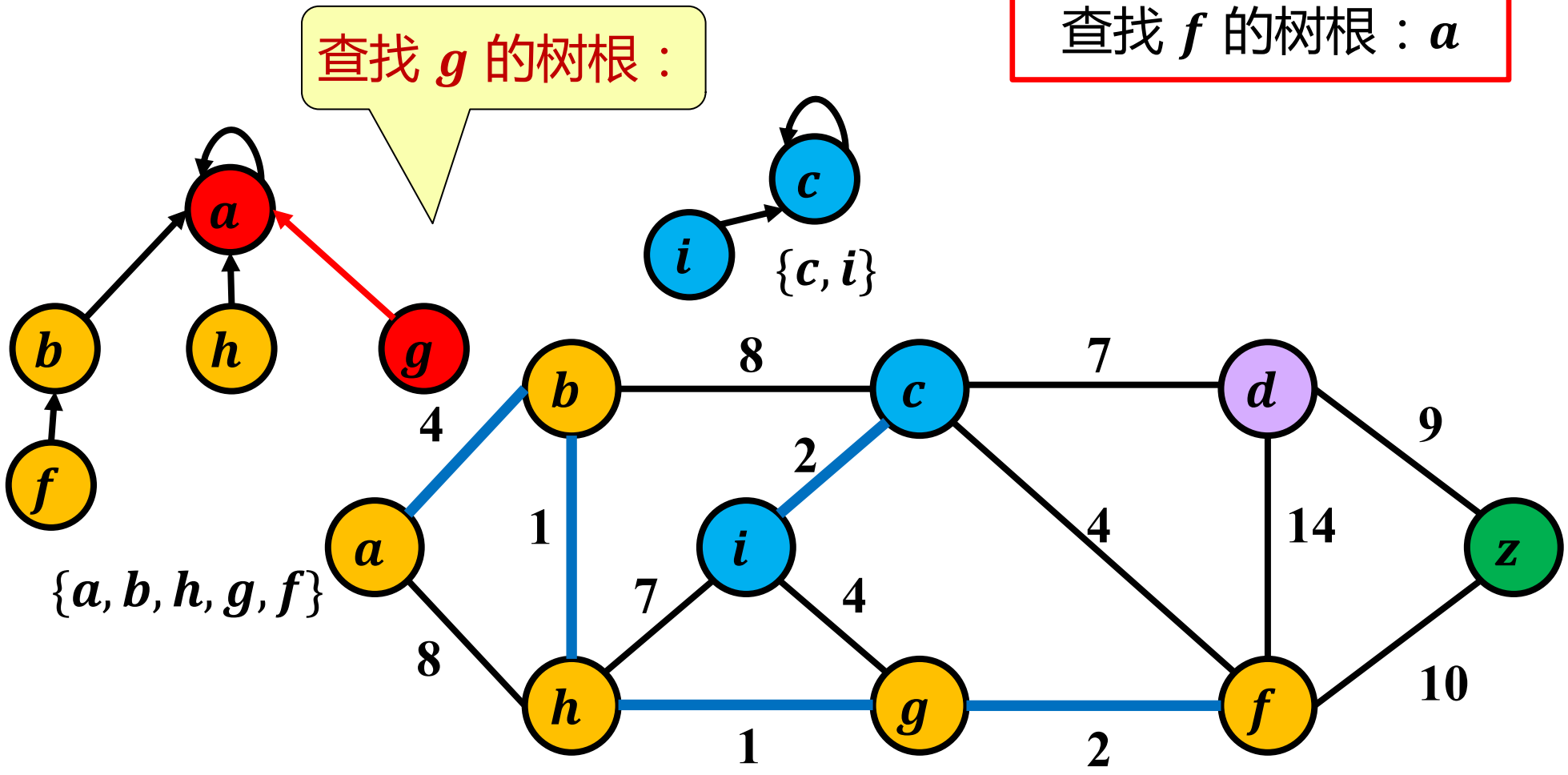
查找 f 的树根：a



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

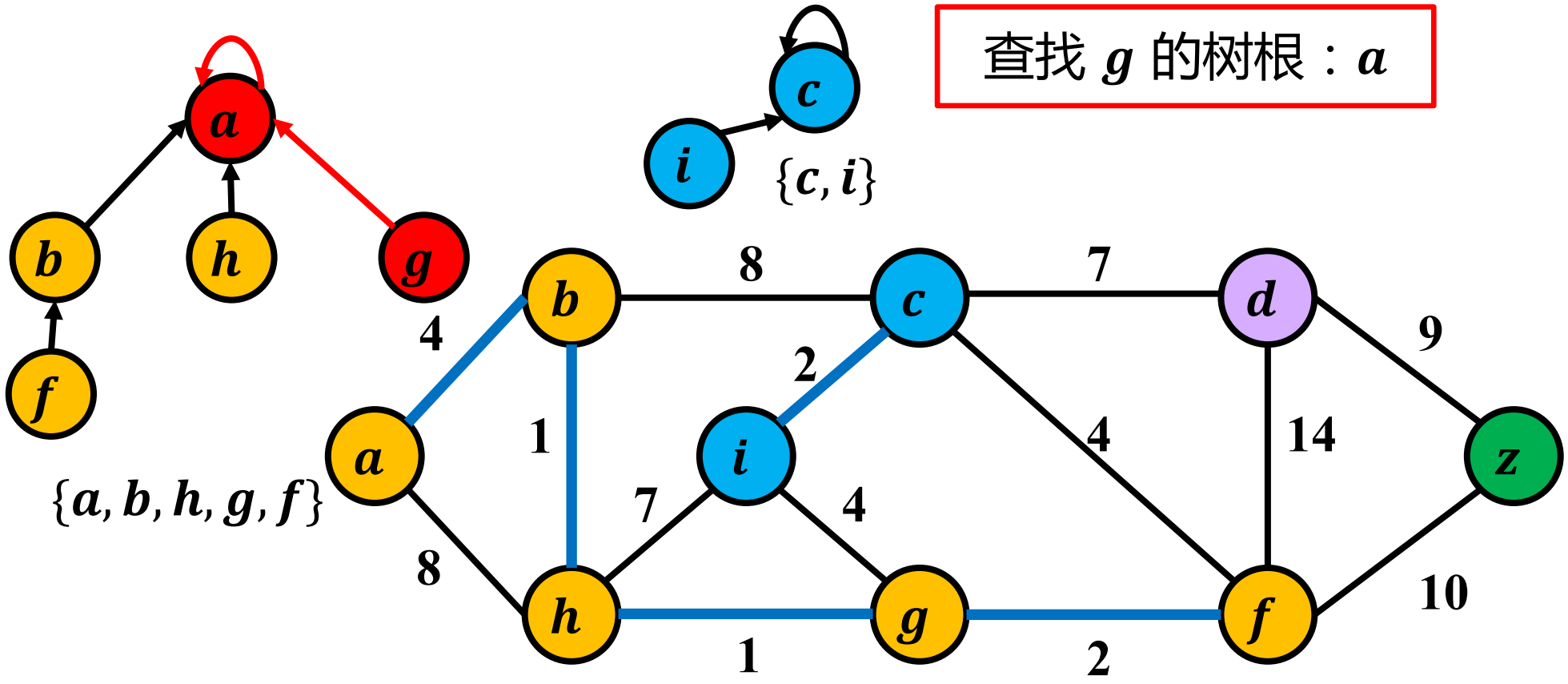


不相交集合

- 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：a

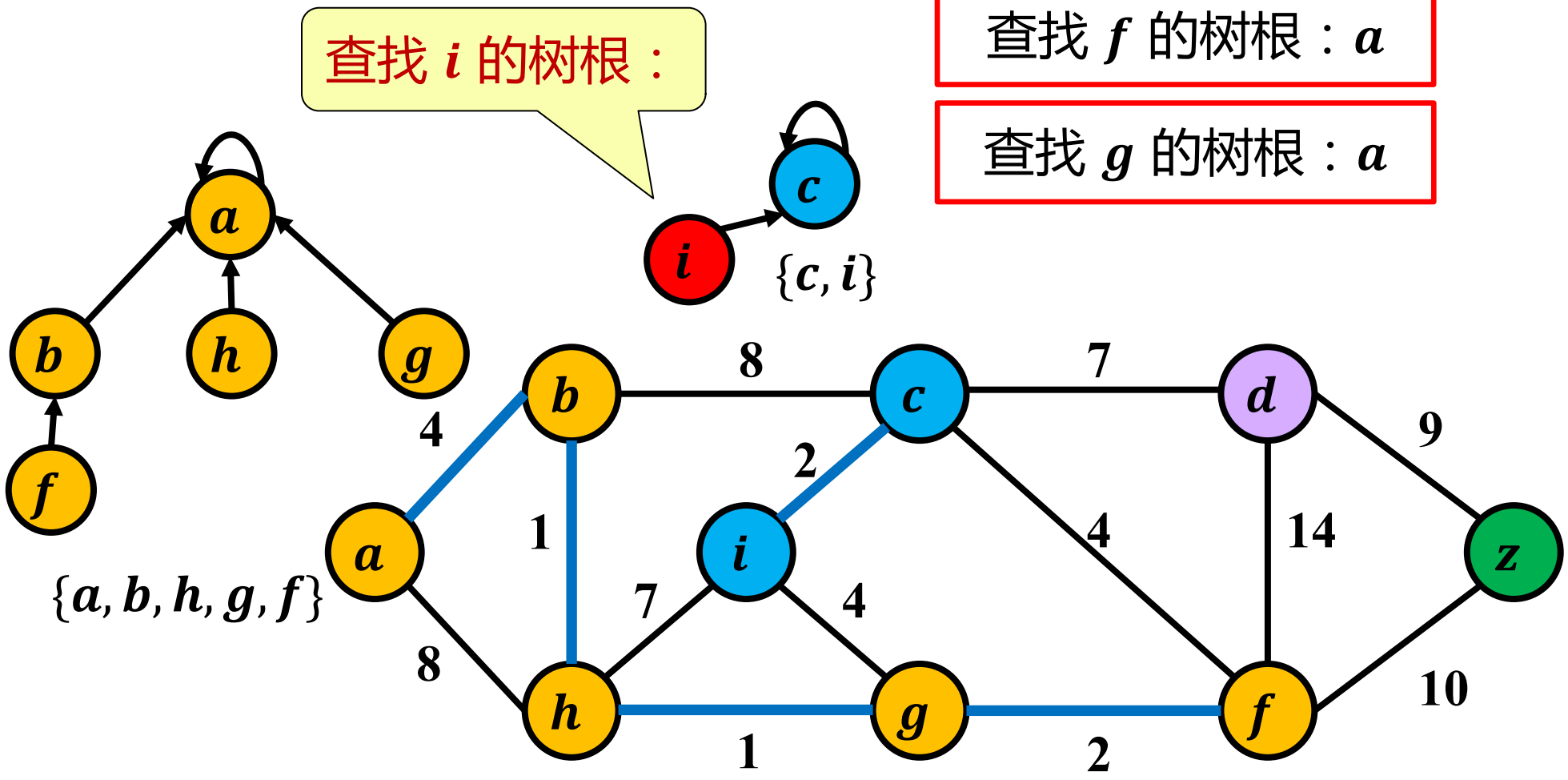


不相交集合

- 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

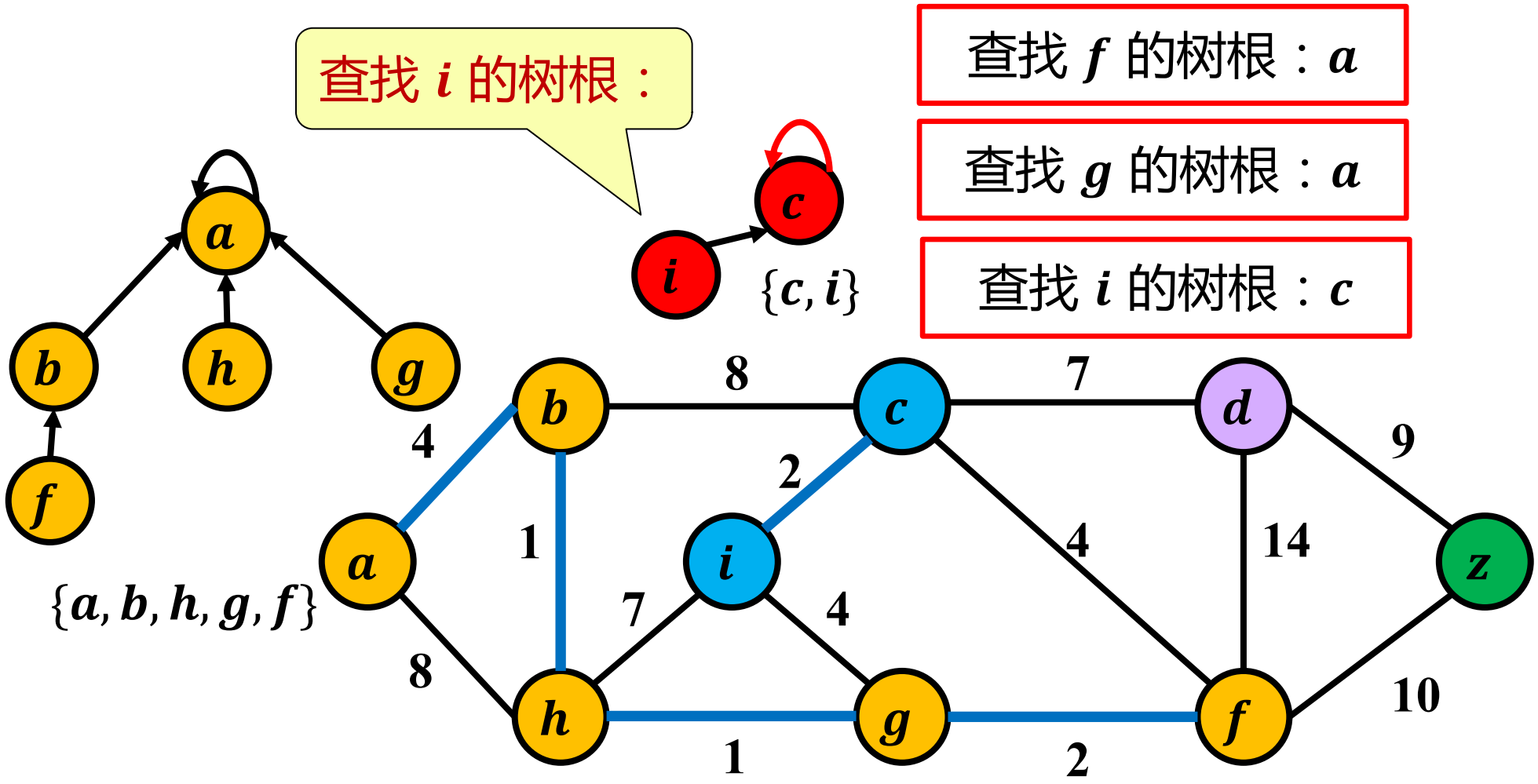
查找 f 的树根：a

查找 g 的树根：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

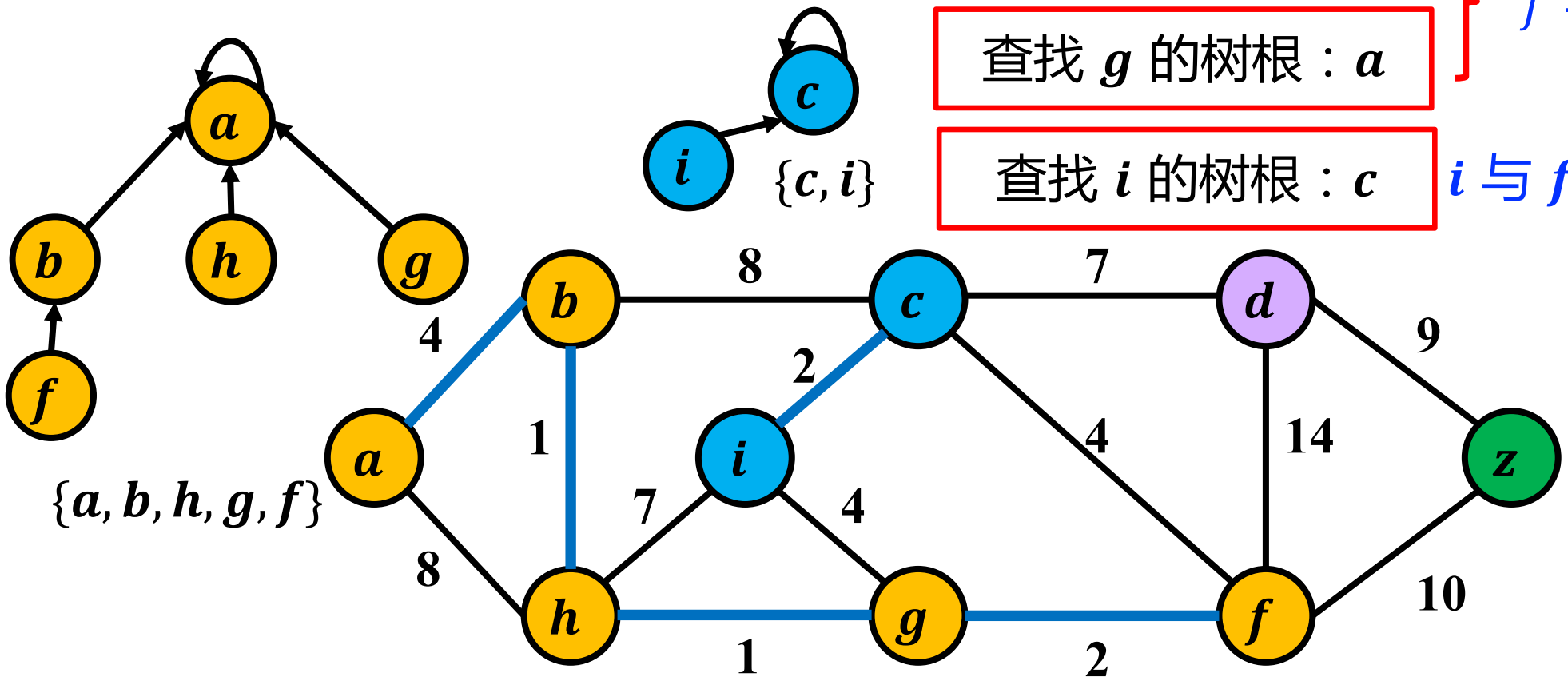
查找 f 的树根：a

查找 g 的树根：a

f 与 g 在同一集合

查找 i 的树根：c

i 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 ← x
return x
```

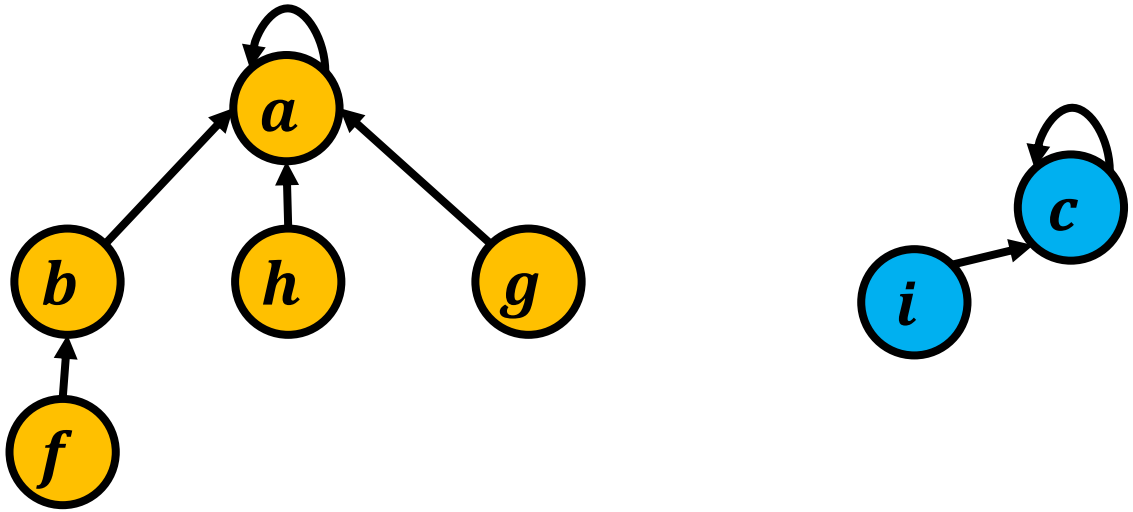
- **Find-Set(x)**

```
输入: 顶点 x
输出: 所属连通分量
while x.parent ≠ x do
|   x ← x.parent
end
return x
```

回溯查找

不相交集合

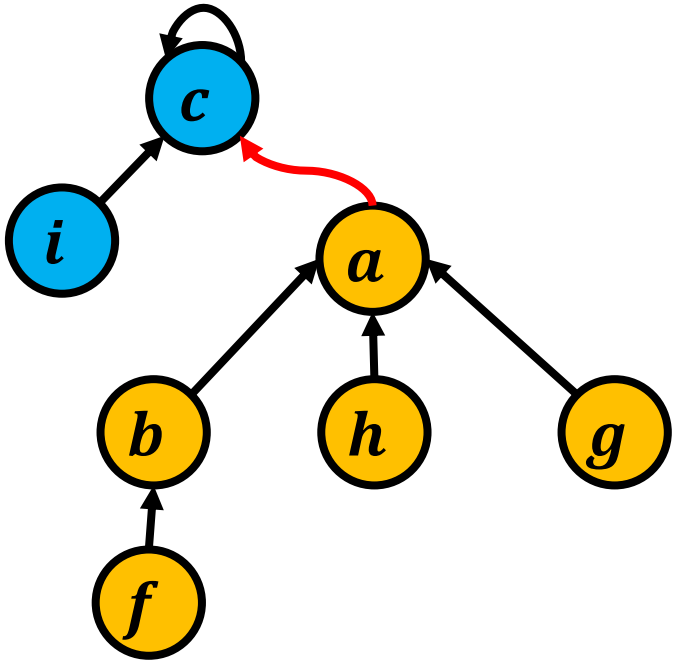
- 初始化集合：创建根结点，并设置一条指向自身的边
- 判定顶点是否在同一集合：回溯查找树根，检查树根是否相同
- 合并集合：合并两棵树



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

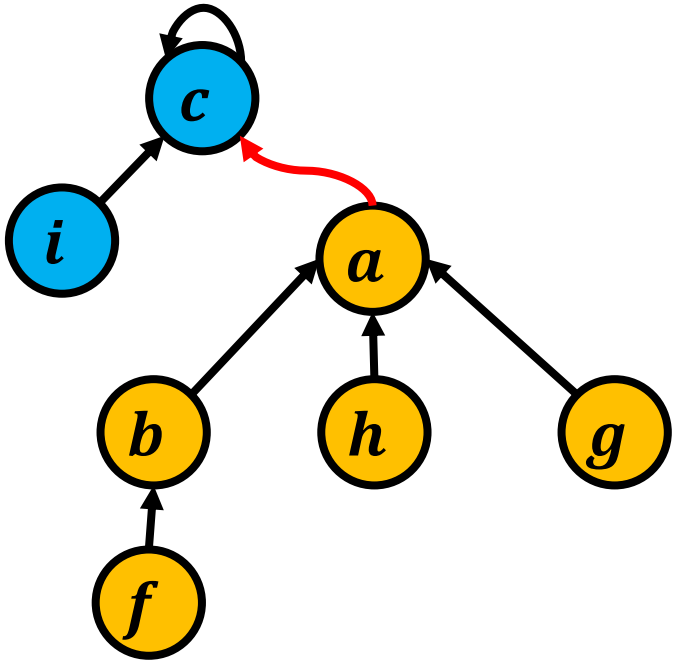
简单实现：找到两树根，任意连接两棵树



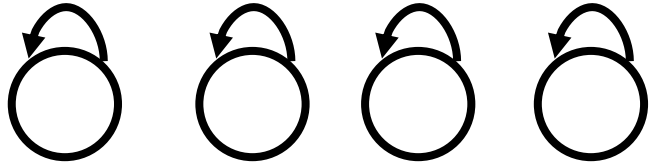
不相交集合

- 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



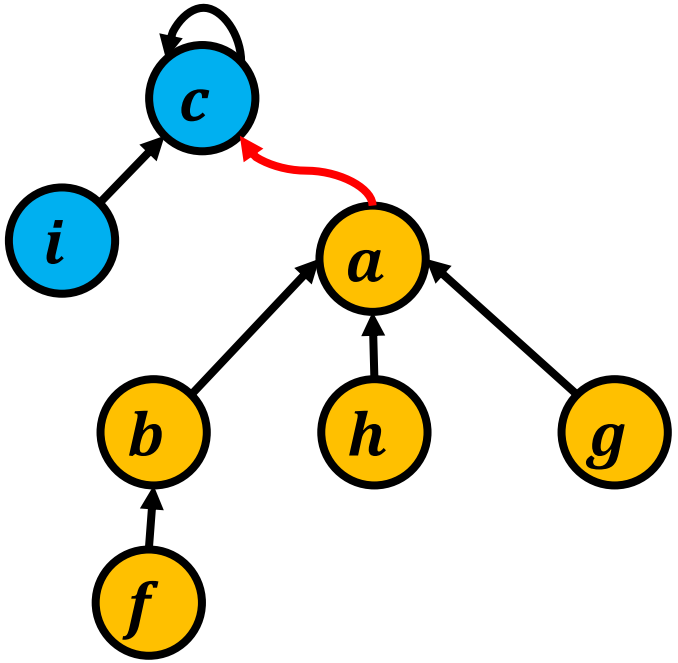
潜在问题：树深度过大，降低查找效率



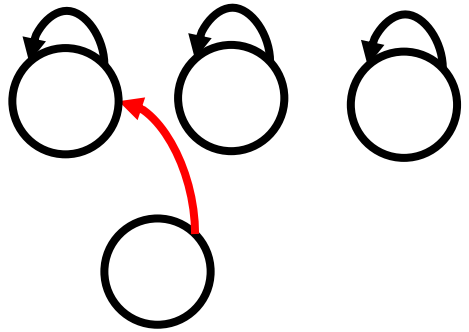
不相交集合

- 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

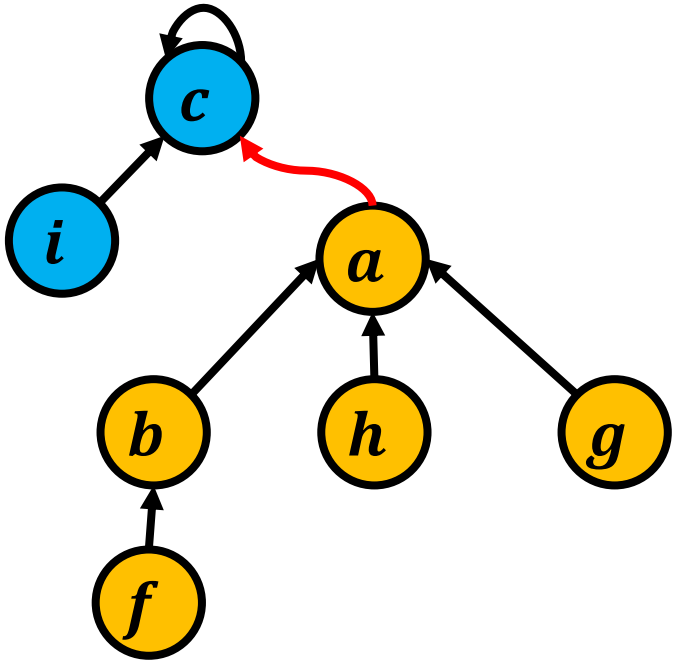


潜在问题：树深度过大，降低查找效率

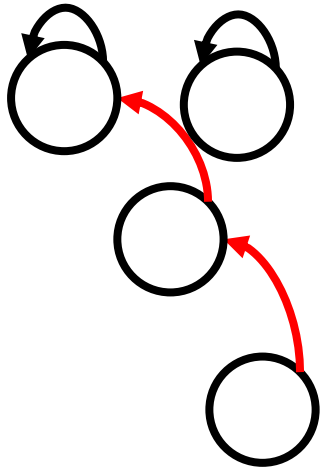
不相交集合

- 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



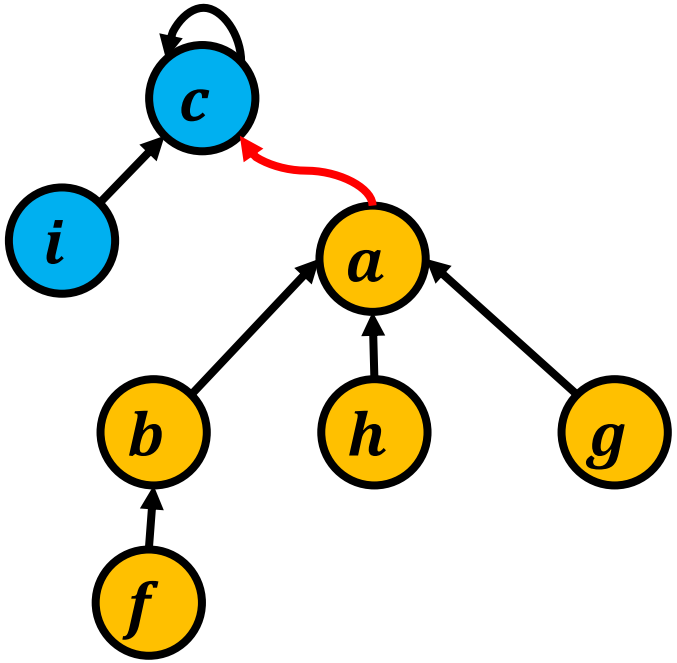
潜在问题：树深度过大，降低查找效率



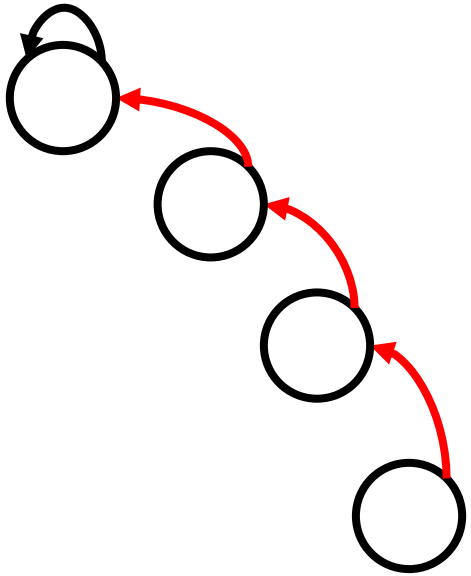
不相交集合

- 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



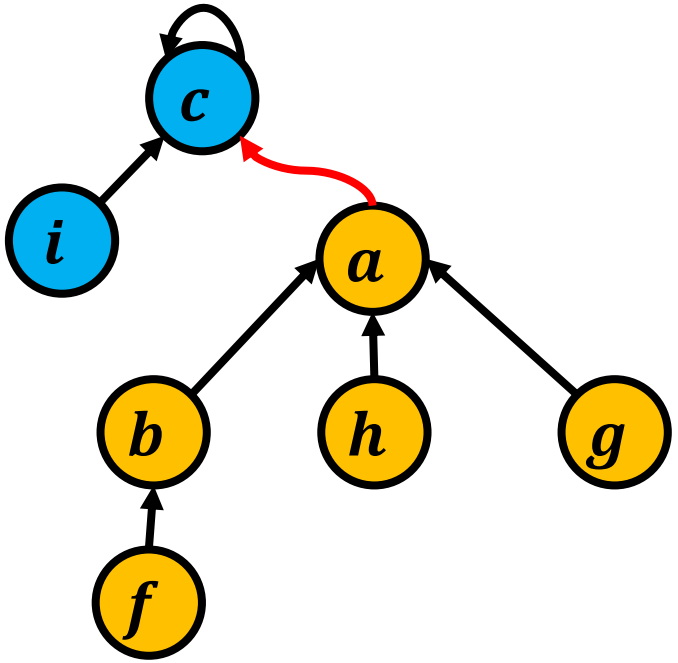
潜在问题：树深度过大，降低查找效率



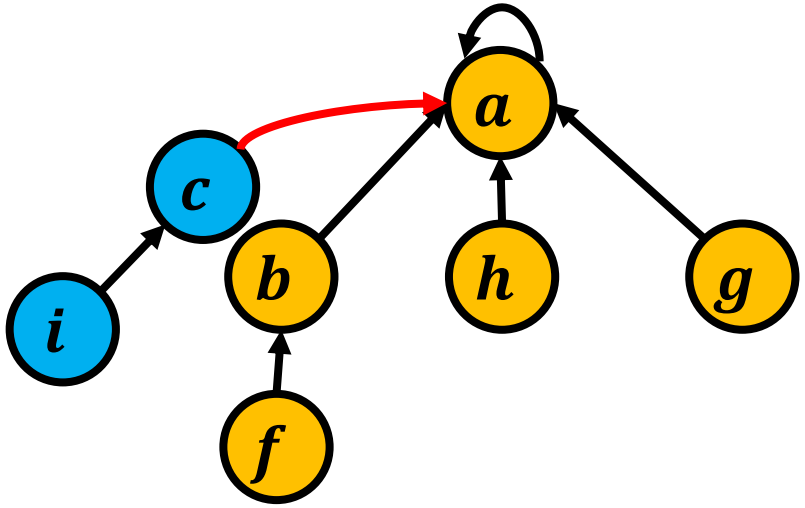
不相交集合

- 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



尽可能降低树高度
提高树根查找效率



高效实现：树高小的树连接到树高大的树上

不相交集合：伪代码

- **Union-Set(x)**

```
输入: 顶点 x, y
a ← Find-Set(x)
b ← Find-Set(y)
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
```

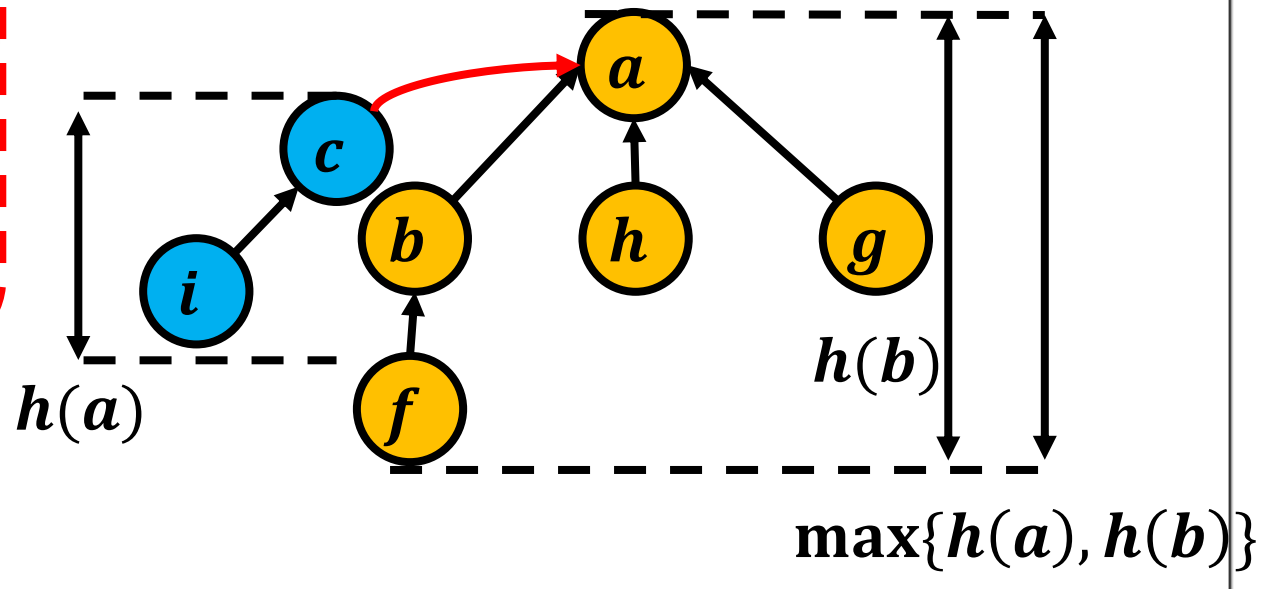
找到两树根

不相交集合：伪代码

- **Union-Set(x)**

```
输入: 顶点 x, y
a ← Find-Set(x)
b ← Find-Set(y)
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

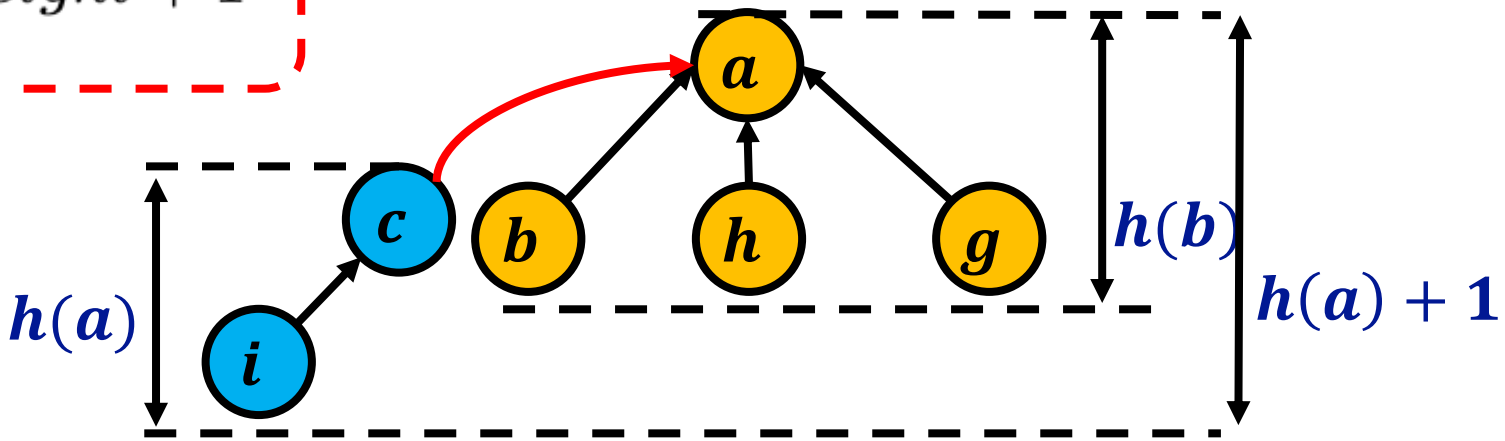


不相交集合：伪代码

- **Union-Set(x)**

```
输入: 顶点 x, y
a ← Find-Set(x)
b ← Find-Set(y)
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，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 ← 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 ≠ x do
  |   x ← x.parent
end
return x
```

$O(h)$

h为树高

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 ← Find-Set(x)
b ← Find-Set(y)
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
```

$O(h)$

$O(1)$

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 ← Find-Set(x)
b ← Find-Set(y)
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
```

$O(h)$

$O(1)$

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⟶$|V| \geq 2^h$
- 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 ⟶ $|V| \geq 2^h$
- 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⟶$|V| \geq 2^h$
- 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 ⟶ $|V| \geq 2^h$
- 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⟶$|V| \geq 2^h$
- 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$h_a \neq h_b$：$V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} \geq 2^{\max\{h_a, h_b\}}$

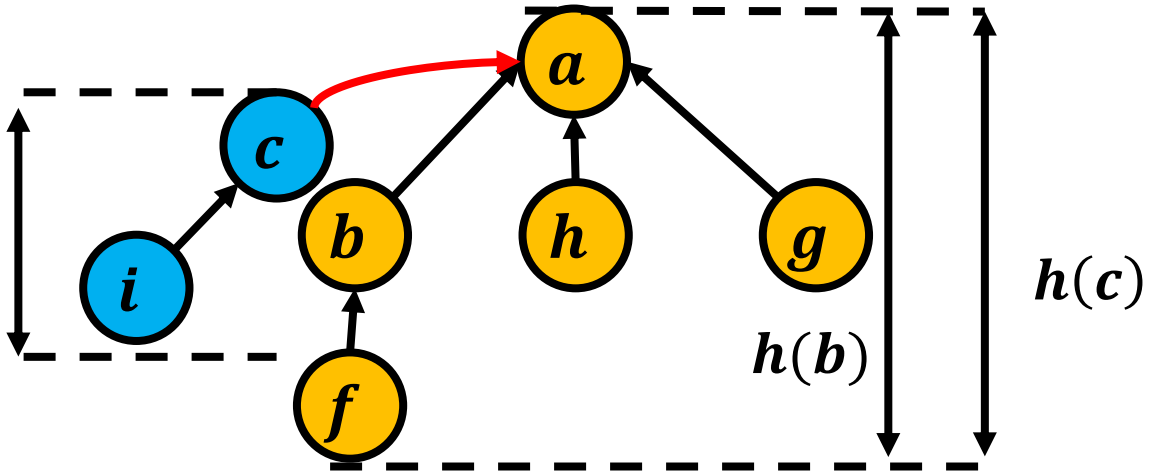
两正数之和大于其中任何一个

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 ⟶ $|V| \geq 2^h$
- 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 ⟶ $|V| \geq 2^h$
- 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$h_a = h_b$：$V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} = 2^{h_a+1}$

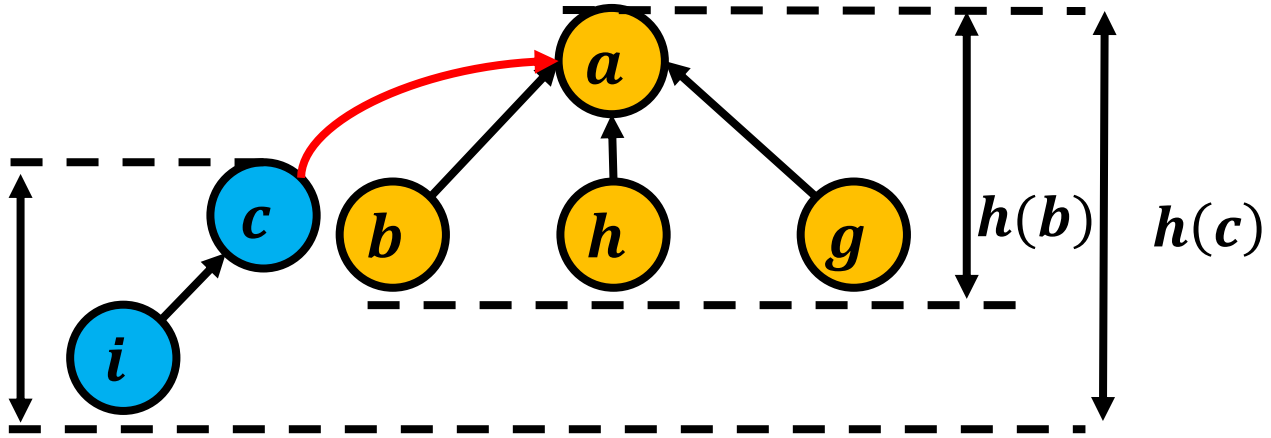
$$h_a = h_b$$
$$2^{h_a} + 2^{h_b} = 2 \times 2^{h_a} = 2^{h_a+1}$$

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⟶$|V| \geq 2^h$
- 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 ⟶ $|V| \geq 2^h$
- 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$h_a = h_b$：$V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} = 2^{h_a+1} = 2^{h_c}$
  - 综上，所有不相交集和都满足$|V| \geq 2^h$，即$h \leq \log|V|$

初始化产生的不相交集合满足$|V| \geq 2^h$

合并产生的不相交集合满足$|V| \geq 2^h$

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 ⟶ $|V| \geq 2^h$
- 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$h_a = h_b$：$V_c = V_a + V_b \geq 2^{h_a} + 2^{h_b} = 2^{h_a+1} = 2^{h_c}$
  - 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)**：$O(h) = O(\log|V|)$

初始化产生的不相交集合满足$|V| \geq 2^h$

合并产生的不相交集合满足$|V| \geq 2^h$

伪代码

- **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 ← {}
for (u, v) ∈ E do
    if u, v 不在同一子树 then
        T ← T ∪ {(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？

伪代码

- **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 ← {}
for (u, v) ∈ E do
    if Find-Set(u) ≠ Find-Set(v) then
        T ← T ∪ {(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

提纲

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

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 ← {}
for (u, v) ∈ E do
    if Find-Set(u) ≠ Find-Set(v) then
        T ← T ∪ {(u, v)}                } O(log|V|)
        Union-Set(u, v)                 }
    end                                 } O(|E|log|V|)
end
return T
```

- 时间复杂度
  - $O(|E|\log|E| + |E|\log|V|)$

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 ← {}
for (u, v) ∈ E do
    if Find-Set(u) ≠ Find-Set(v) then
        T ← T ∪ {(u, v)}                      } O(log|V|)
        Union-Set(u, v)                       } O(|E|log|V|)
    end
end
return T
```

- 时间复杂度

假设$|E| = O(|V|^2)$

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

小结

- **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	

图算法篇：单源最短路径问题之 Dijkstra算法

北京航空航天大学
计算机学院

提纲

- 问题背景
- 算法思想
- 算法实例
- 算法分析
- 算法性质

问题背景

- 从知春路到其他站点，如何安排路线？

问题背景

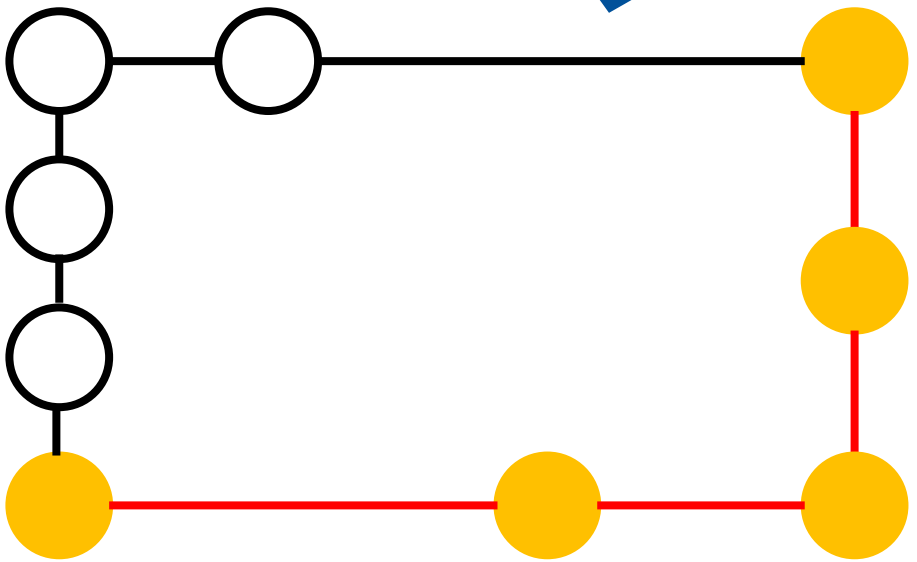
- 从知春路到其他站点，如何安排路线？

问题背景

- 从知春路到其他站点，如何安排路线？

经停最少

无权图

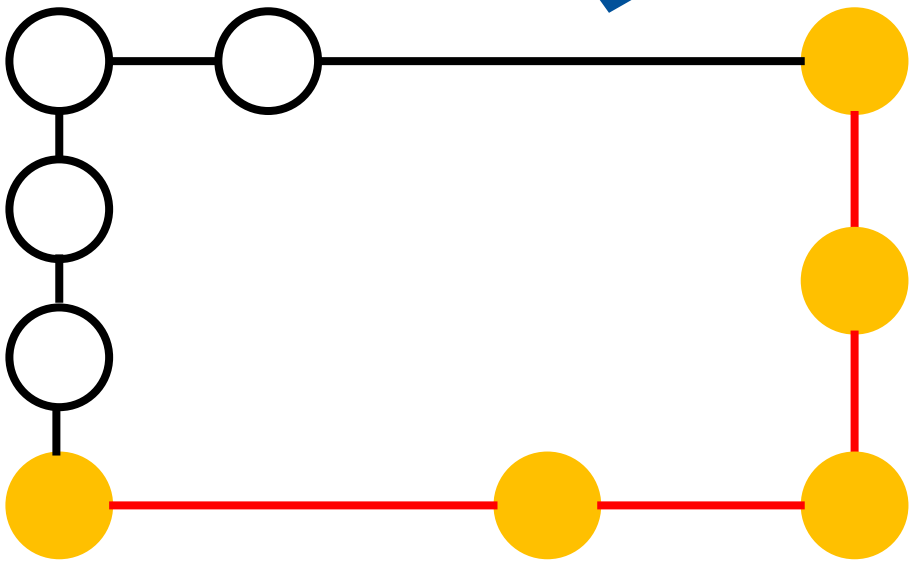


问题背景

- 从知春路到其他站点，如何安排路线？

经停最少

无权图



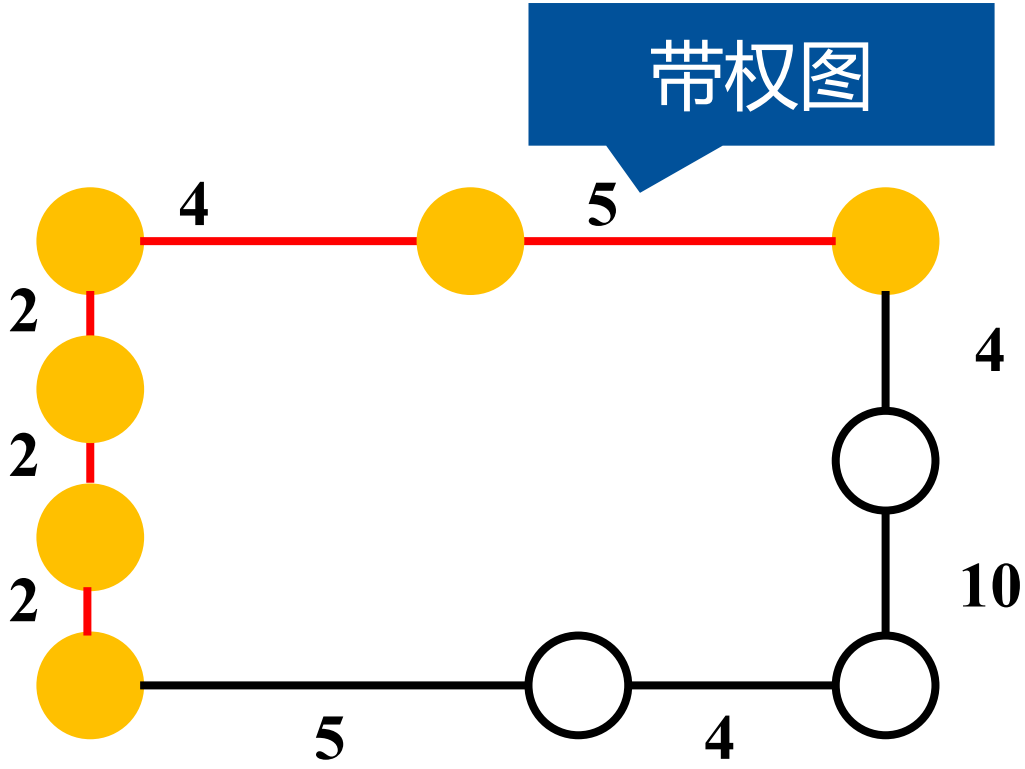
使用广度优先搜索求最短路径

问题背景

- 从知春路到其他站点，如何安排路线？

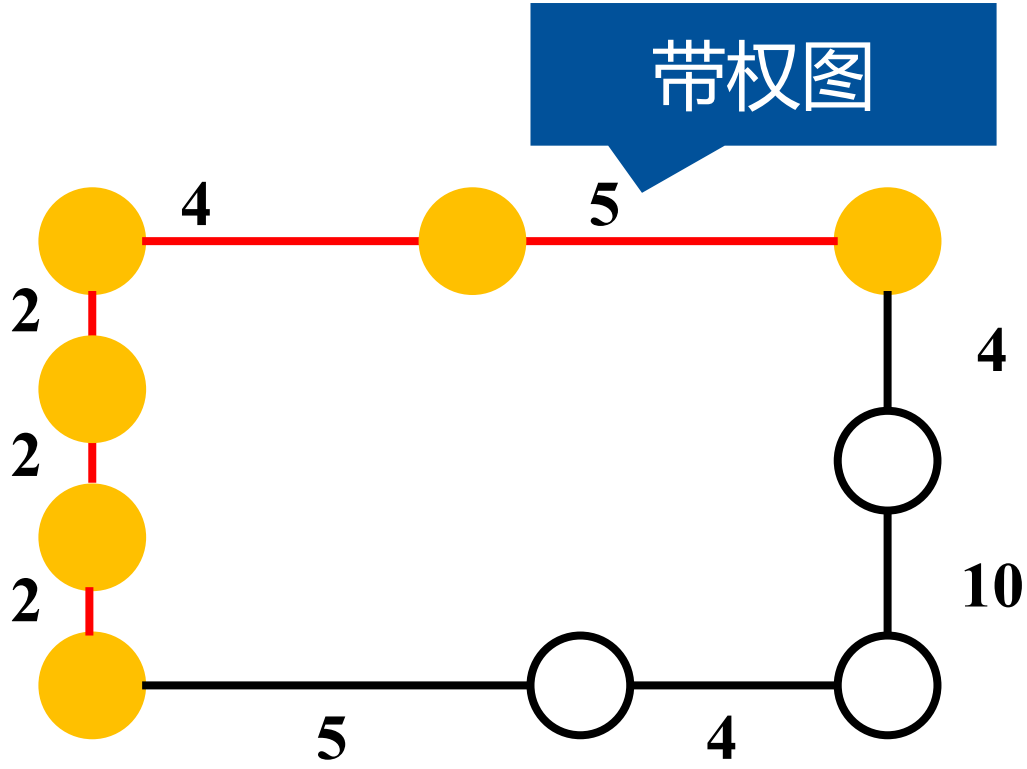
问题背景

- 从知春路到其他站点，如何安排路线？



问题背景

- 从知春路到其他站点，如何安排路线？



问题：如何计算带权图中源点到所有其他顶点的最短路径?

问题定义

单源最短路径问题 (边权为正)

Single Source Shortest Paths Problem with Positive Weights

输入

- 带权图$\boldsymbol{G} =< \boldsymbol{V}, \boldsymbol{E}, \boldsymbol{W} >$，其中$\boldsymbol{w(u,v) \geq 0}$（图中所有边权为正）,$(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{E}$
- 源点编号$\boldsymbol{s}$

问题定义

单源最短路径问题 (边权为正)

Single Source Shortest Paths Problem with Positive Weights

输入

- 带权图$\boldsymbol{G} =< \boldsymbol{V}, \boldsymbol{E}, \boldsymbol{W} >$，其中$\boldsymbol{w(u,v) \geq 0}$（图中所有边权为正）,$(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{E}$
- 源点编号$\boldsymbol{s}$

输出

- 源点 $\boldsymbol{s}$ 到所有其他顶点 $\boldsymbol{t}$ 的最短距离 $\boldsymbol{\delta(s,t)}$ 和最短路径 $< \boldsymbol{s}, \ldots, \boldsymbol{t} >$

提纲

问题背景

算法思想

算法实例

算法分析

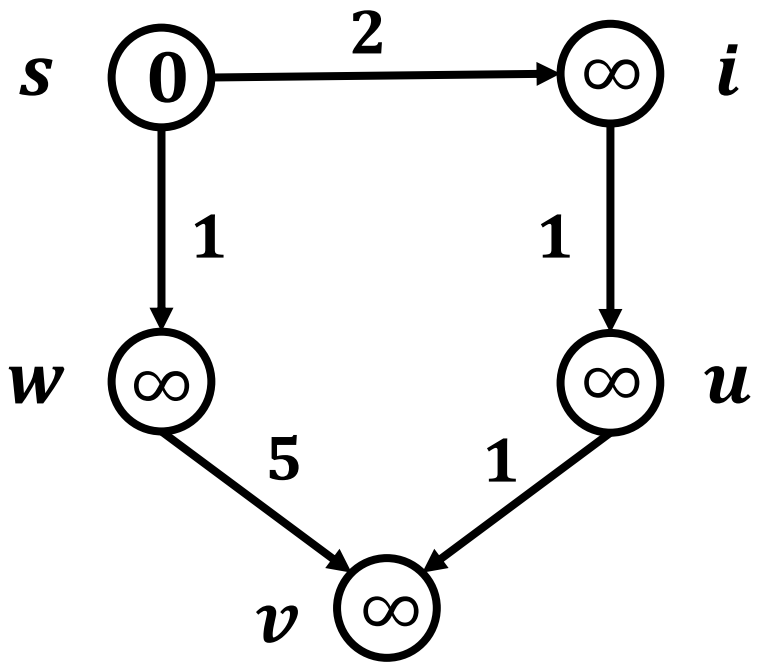
算法性质

算法思想

Edsger W. Dijkstra
1972, Netherlands
ALGOL之父
提出单源最短路径Dijkstra算法

算法思想

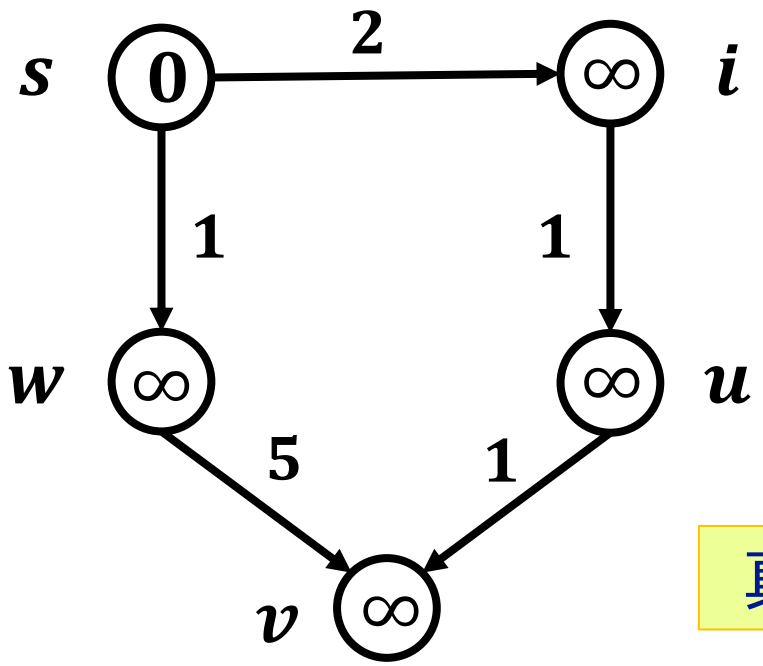
- 辅助数组
  - $dist$表示到源点的距离上界（估计距离）
    - 源点 s，$dist[s] = 0$；其他顶点u，$dist[u]$初始化为∞



V	s	i	w	u	v
$dist$	0	∞	∞	∞	∞

算法思想

- 辅助数组
  - $dist$表示到源点的距离上界（估计距离）
    - 源点 s，$dist[s]=0$；其他顶点u，$dist[u]$初始化为∞
    - $dist[u]$：源点 s 到顶点 u 的距离上界，$\delta(s,u) \le dist[u]$

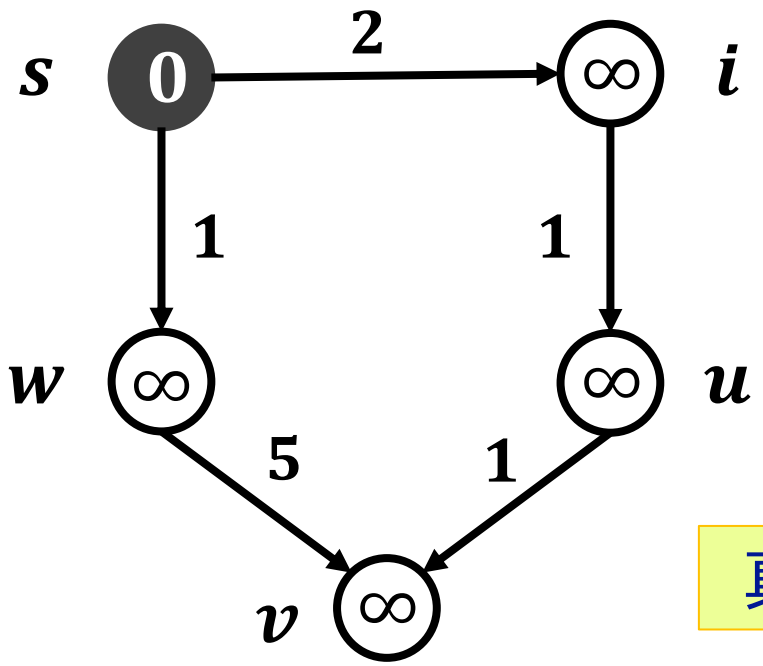


V	s	i	w	u	v
$dist$	0	∞	∞	∞	∞
δ	0	2	1	3	4

真实最短距离

算法思想

- 辅助数组
  - $dist$表示到源点的距离上界（估计距离）
    - 源点 s，$dist[s]=0$；其他顶点u，$dist[u]$初始化为∞
    - $dist[u]$：源点 s 到顶点 u 的距离上界，$\delta(s,u) \le dist[u]$
  - $color$表示顶点状态
    - 黑色：到顶点 u 最短路径已被确定

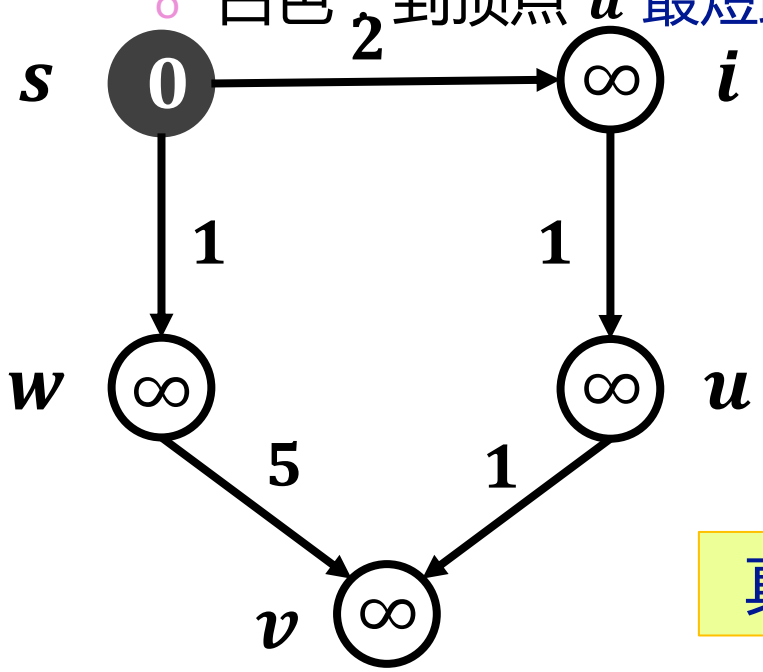


V	s	i	w	u	v
$dist$	0	∞	∞	∞	∞
δ	0	2	1	3	4

真实最短距离

算法思想

- 辅助数组
  - $dist$表示到源点的距离上界（估计距离）
    - 源点 s，$dist[s]=0$；其他顶点u，$dist[u]$初始化为∞
    - $dist[u]$：源点 s 到顶点 u 的距离上界，$\delta(s,u) \le dist[u]$
  - $color$表示顶点状态
    - 黑色：到顶点 u 最短路径已被确定
    - 白色：到顶点 u 最短路径尚未确定

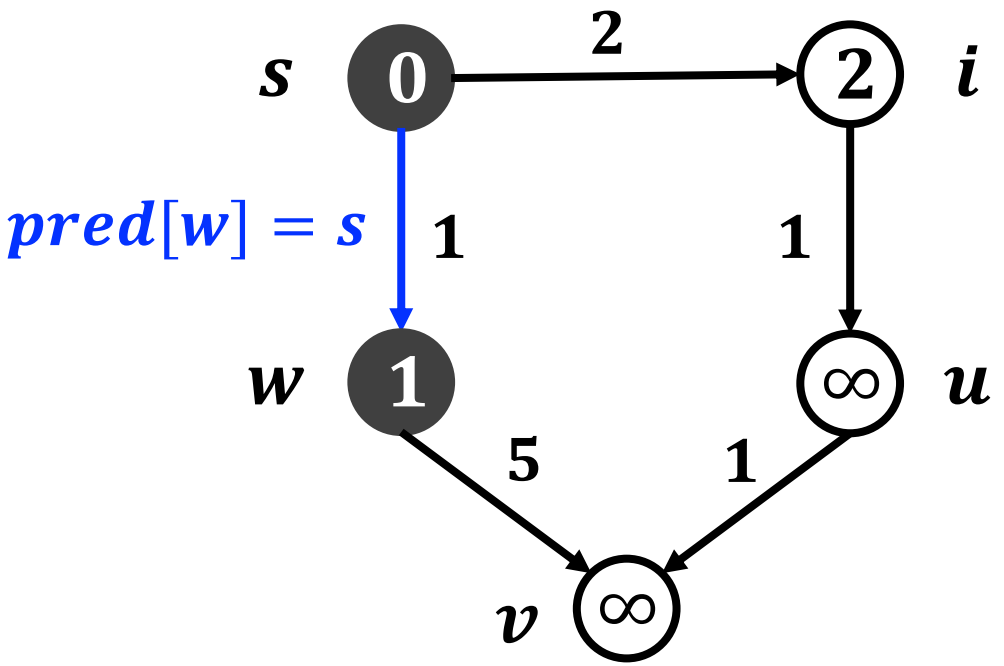


V	s	i	w	u	v
$dist$	0	∞	∞	∞	∞
δ	0	2	1	3	4

真实最短距离

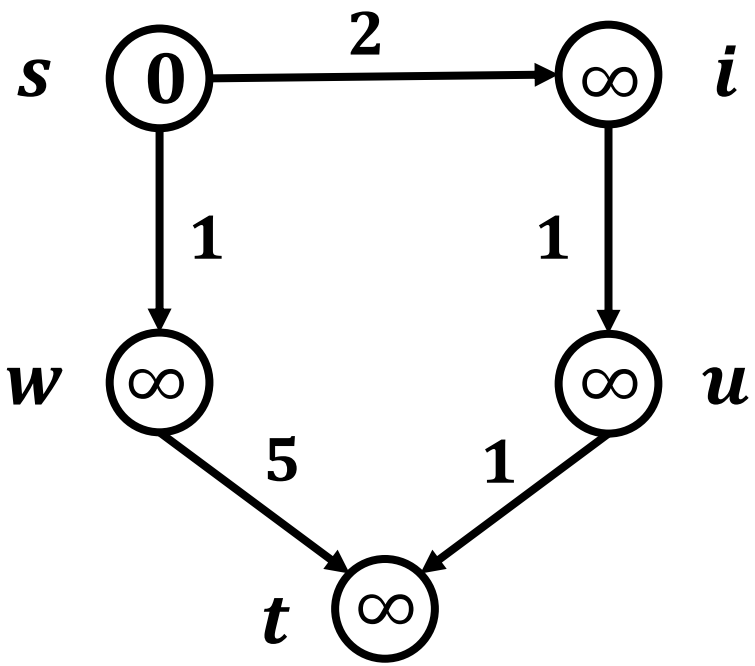
算法思想

- 辅助数组
  - $dist$表示到源点的距离上界（估计距离）
    - 源点 s，$dist[s] = 0$；其他顶点u，$dist[u]$初始化为∞
    - $dist[u]$：源点 s 到顶点 u 的距离上界，$\delta(s,u) \leq dist[u]$
  - $color$表示顶点状态
    - 黑色：到顶点 u 最短路径已被确定
    - 白色：到顶点 u 最短路径尚未确定
  - $pred$ 表示前驱顶点
    - $(pred[u], u)$为最短路径上的边



算法思想

- 核心思想
  - 步骤1：新建空的黑色顶点集V_A

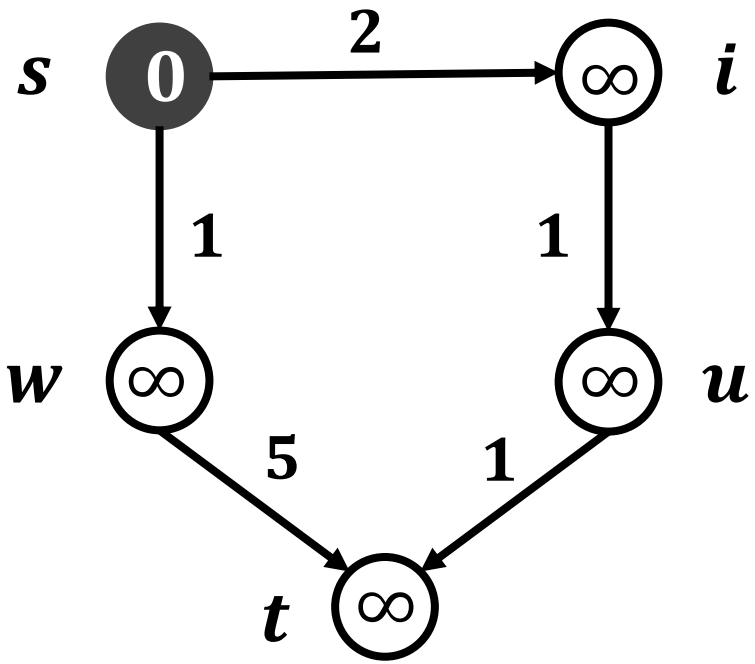


V_A中顶点

$V - V_A$中顶点

算法思想

- 核心思想
  - 步骤1：新建空的黑色顶点集V_A
  - 步骤2：选择一个白色顶点变为黑色（到该顶点最短路被确定）

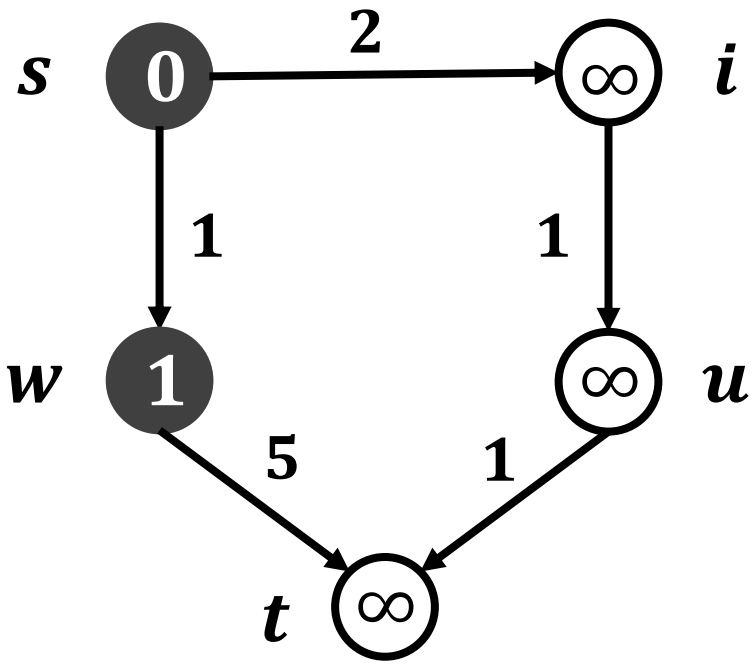


V_A中顶点

$V - V_A$中顶点

算法思想

- 核心思想
  - 步骤1：新建空的黑色顶点集V_A
  - 步骤2：选择一个白色顶点变为黑色（到该顶点最短路被确定）
  - 步骤3：重复步骤2

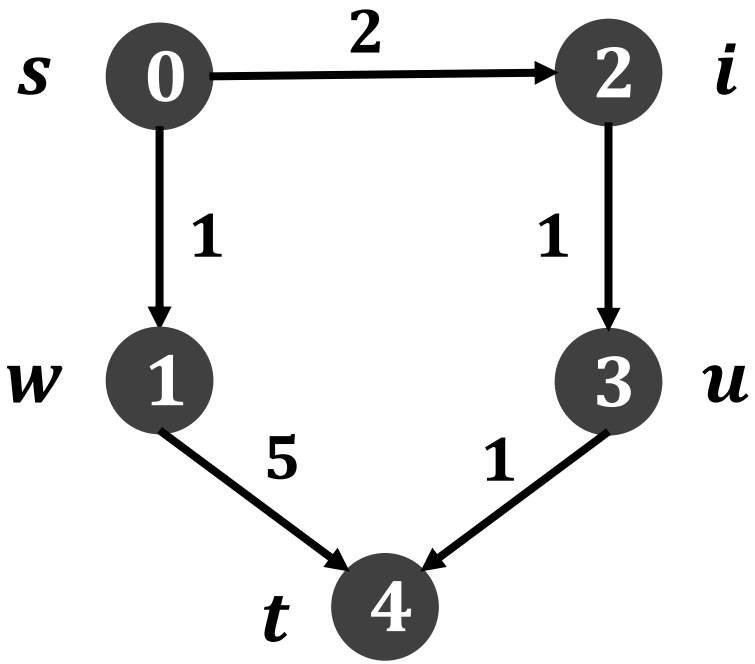


V_A中顶点

$V-V_A$中顶点

算法思想

- 核心思想
  - 步骤1：新建空的黑色顶点集V_A
  - 步骤2：选择一个白色顶点变为黑色（到该顶点最短路被确定）
  - 步骤3：重复步骤2，直至所有顶点均为黑色

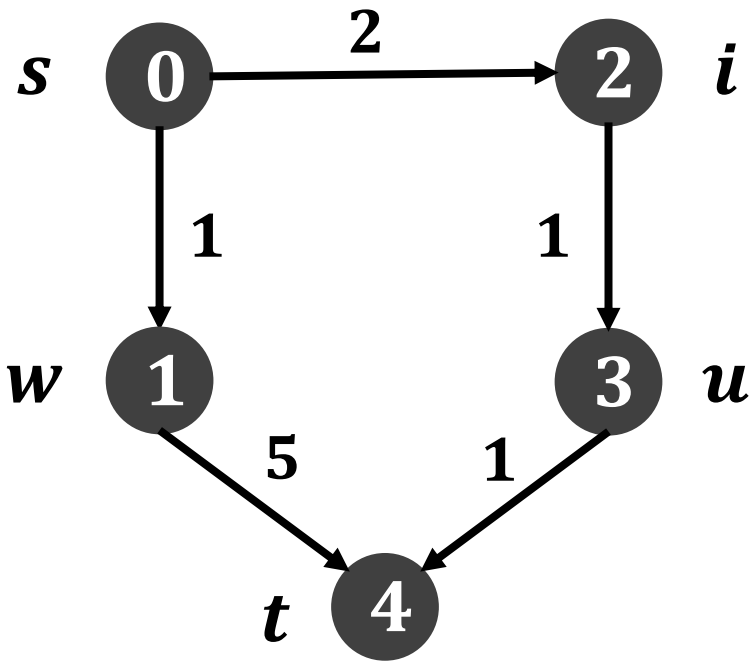


V_A中顶点

$V - V_A$中顶点

算法思想

- 核心思想
  - 步骤1：新建空的黑色顶点集V_A
  - 步骤2：选择一个白色顶点变为黑色（到该顶点最短路被确定）
  - 步骤3：重复步骤2，直至所有顶点均为黑色



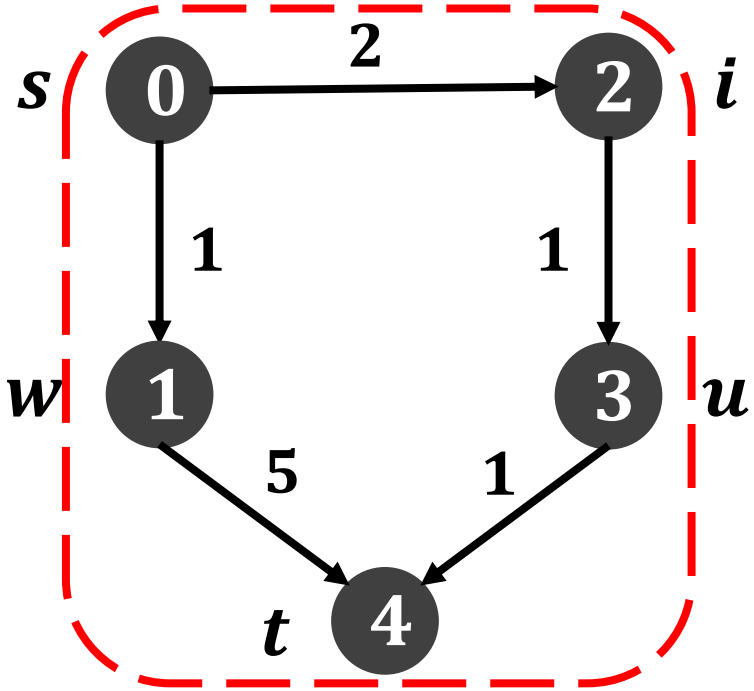
问题：选择哪个白色顶点变为黑色?

V_A中顶点

$V-V_A$中顶点

算法思想

- 核心思想
  - 步骤1：新建空的黑色顶点集V_A
  - 步骤2：选择一个白色顶点变为黑色（到该顶点最短路被确定）
  - 步骤3：重复步骤2，直至所有顶点均为黑色



问题1：选择哪个白色顶点变为黑色?

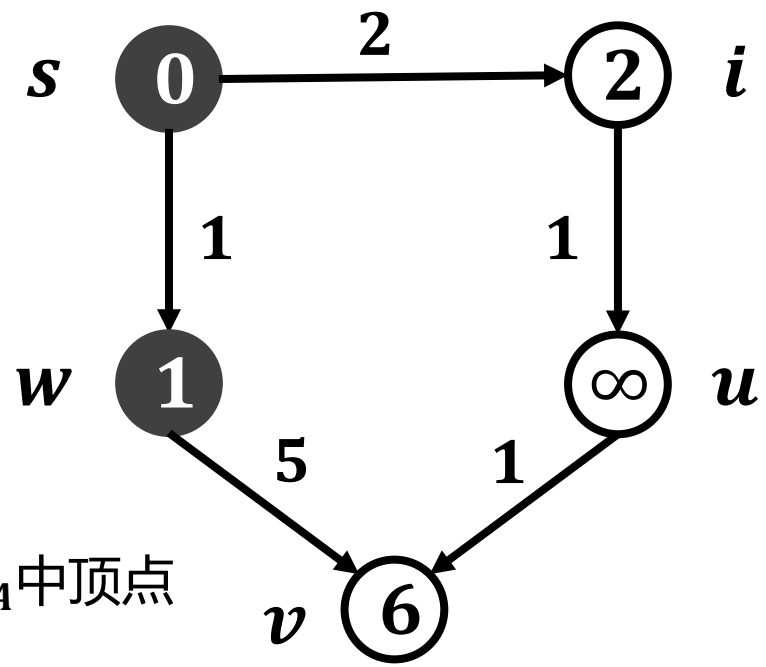
问题2：如何更新各顶点的估计距离?

V_A中顶点

$V-V_A$中顶点

算法思想

- 问题1：选择哪个白色顶点变为黑色？采用贪心策略

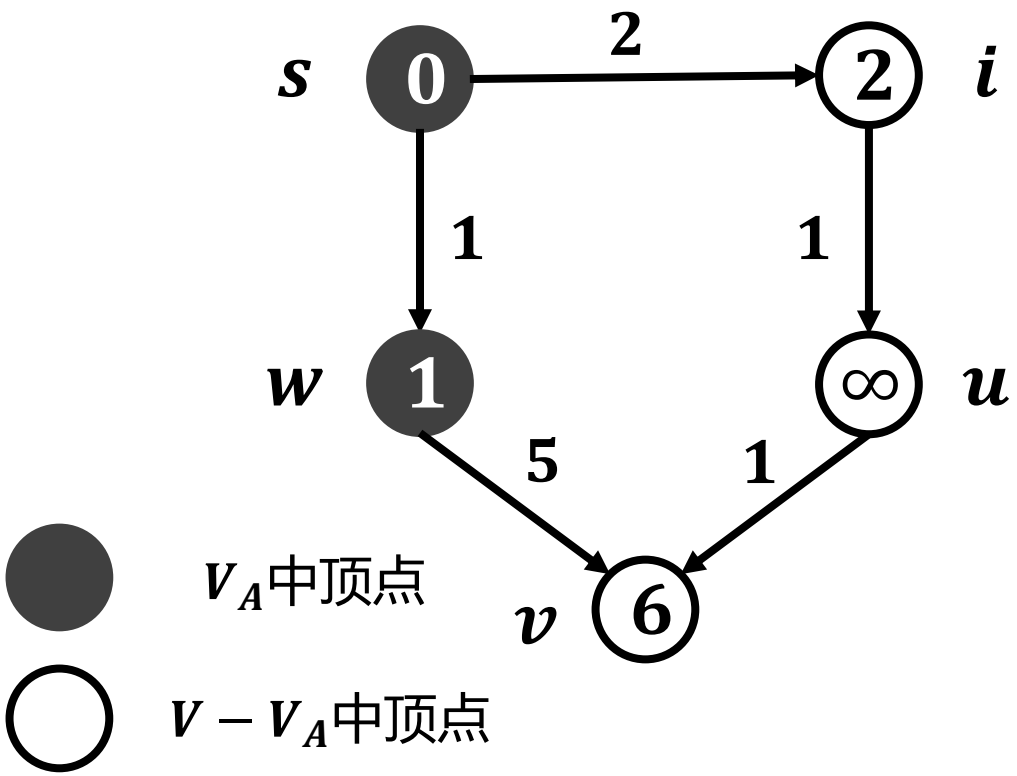


V_A中顶点

$V - V_A$中顶点

算法思想

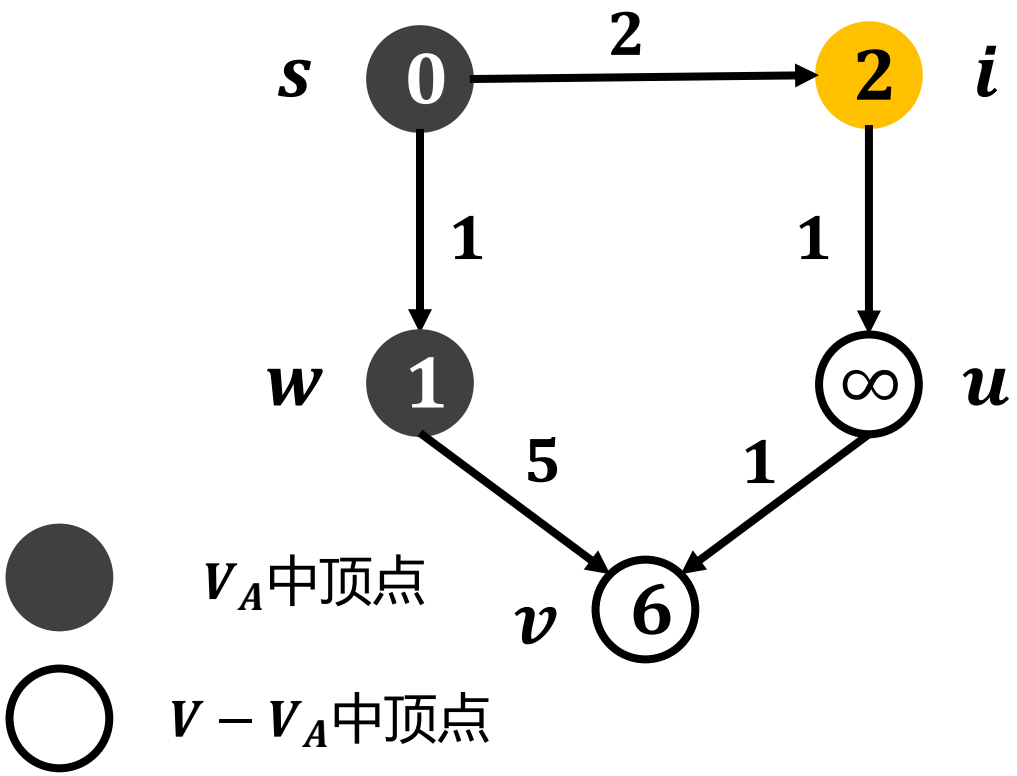
- 问题1：选择哪个白色顶点变为黑色？采用贪心策略
  - 对每个白色顶点 $y \in V - V_A$，都有一个估计距离$dist[y]$



V	s	i	w	u	v
$dist$	0	2	1	∞	6
δ	0	2	1	3	4

算法思想

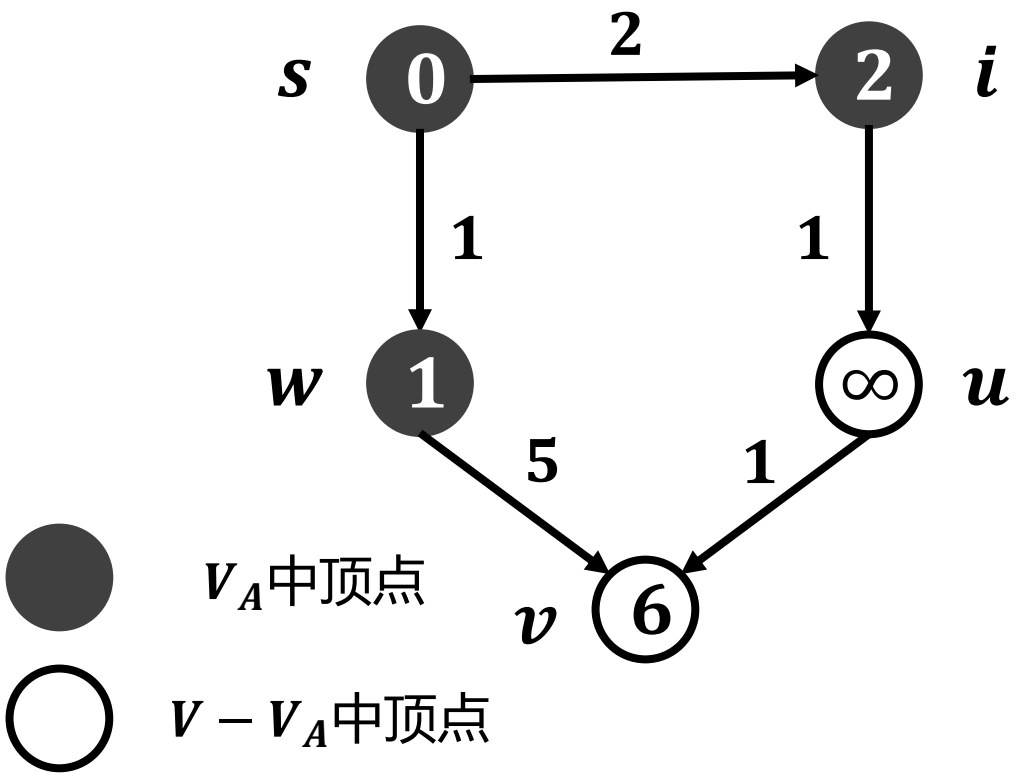
- 问题1：选择哪个白色顶点变为黑色？采用贪心策略
  - 对每个白色顶点 $y \in V - V_A$，都有一个估计距离$dist[y]$
  - 选择估计距离最小的顶点 v，$dist[v] \leq dist[y],\ v, y \in V - V_A$



V	s	i	w	u	v
$dist$	0	2	1	∞	6
δ	0	2	1	3	4

算法思想

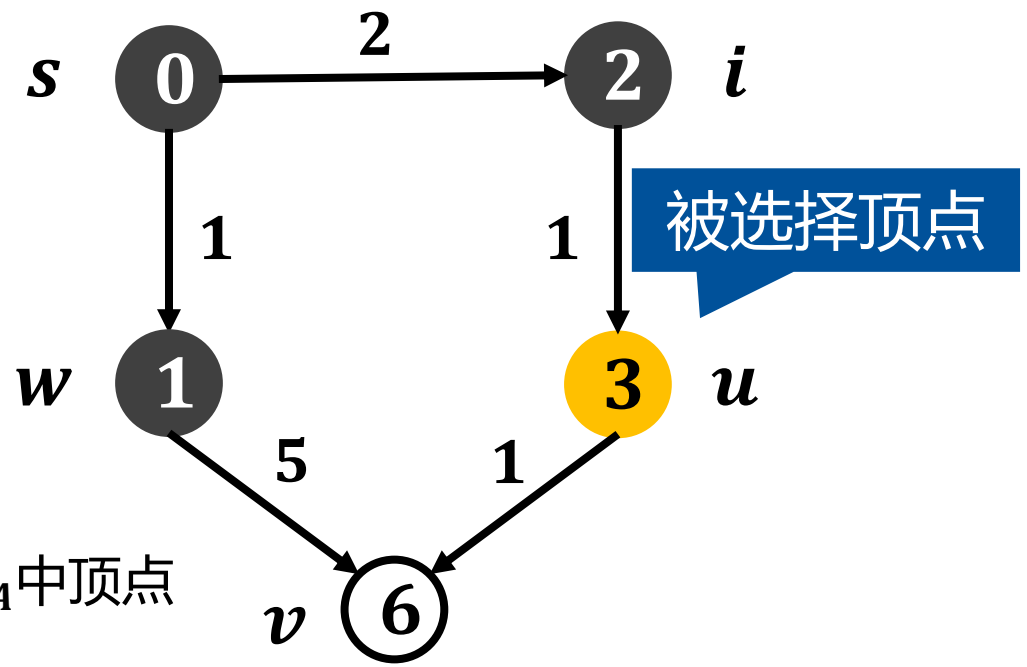
- 问题1：选择哪个白色顶点变为黑色？采用贪心策略
  - 对每个白色顶点 $y \in V - V_A$，都有一个估计距离$dist[y]$
  - 选择估计距离最小的顶点 v，$dist[v] \leq dist[y],\ v, y \in V - V_A$



V	s	i	w	u	v
$dist$	0	2	1	∞	6
δ	0	2	1	3	4

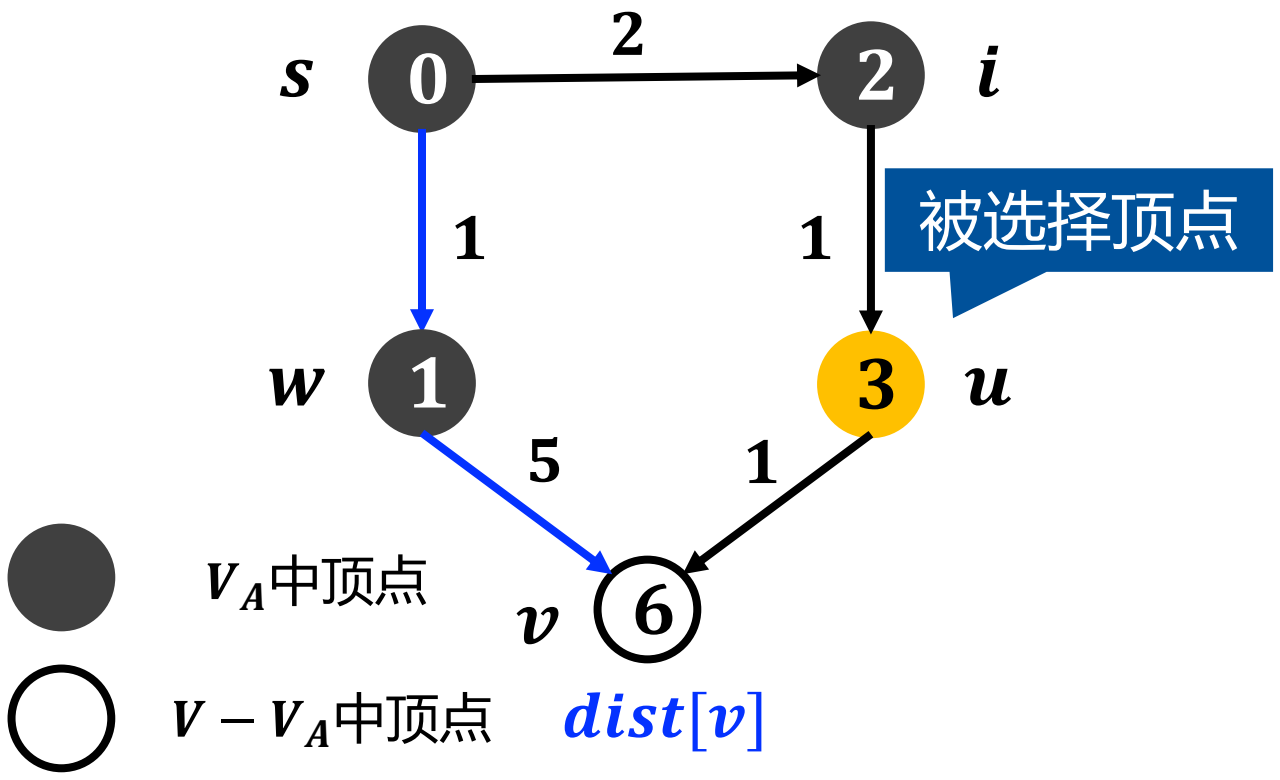
算法思想

- 问题2：如何更新每顶点的估计距离?



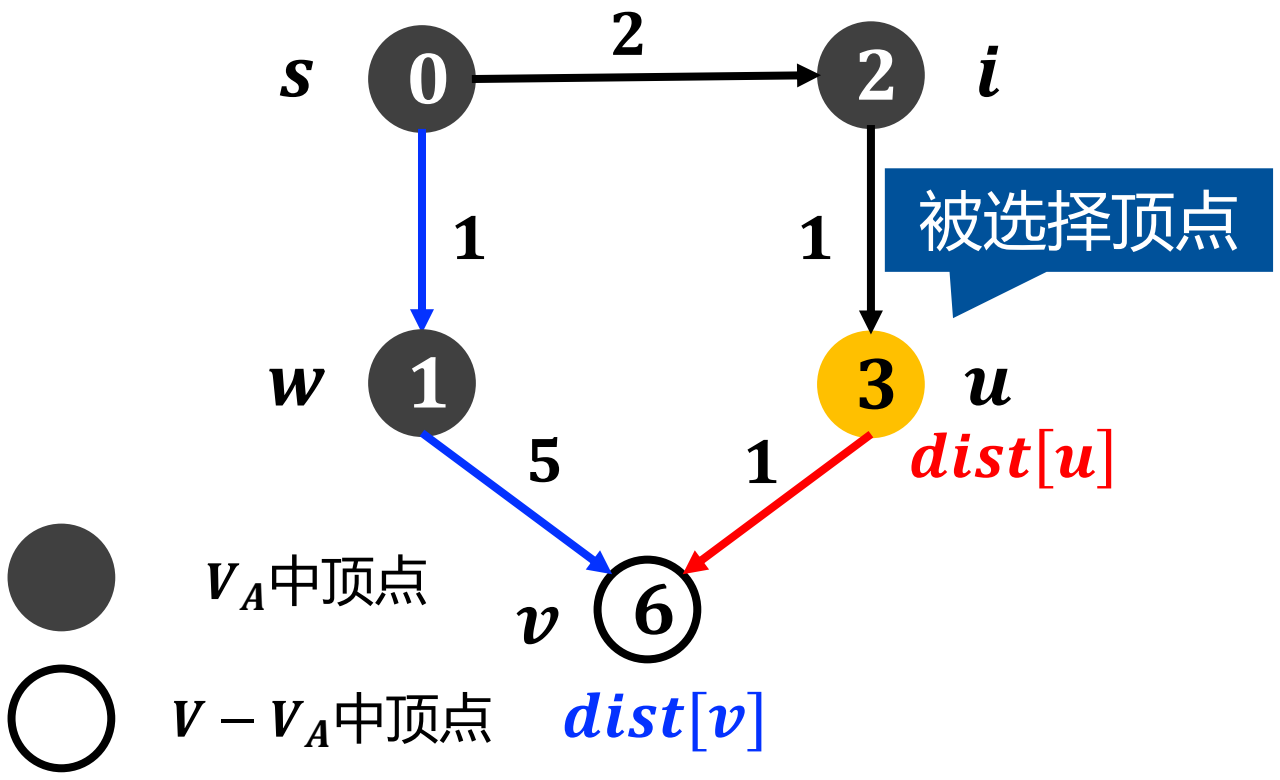
算法思想

- 问题2：如何更新每顶点的估计距离?
  - 当前到顶点 v 的最短路径：$< s, w, v >$，距离为$dist[v]$



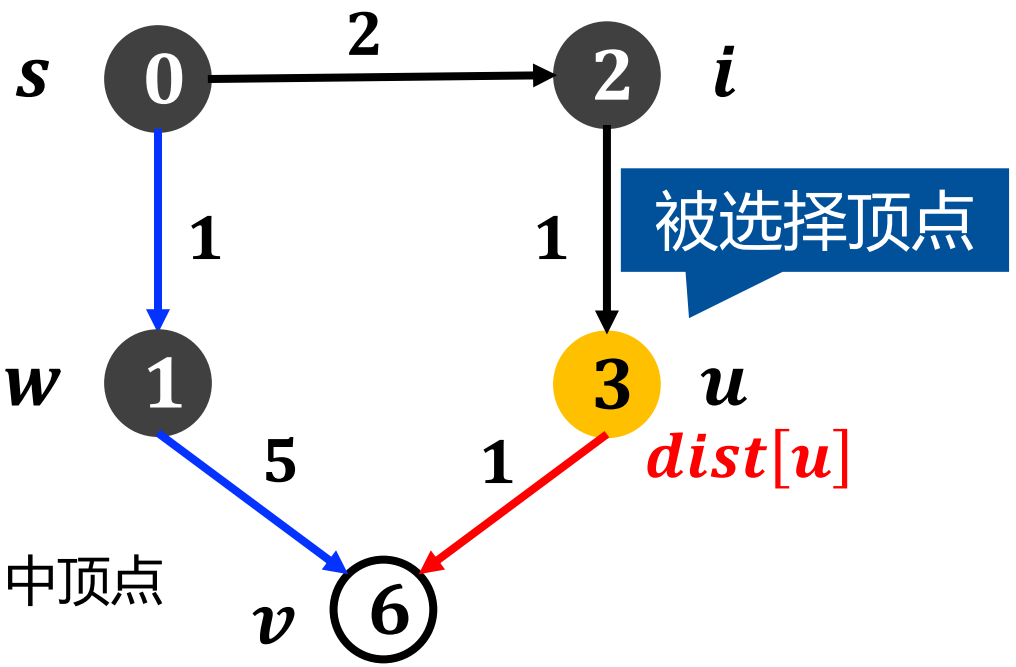
算法思想

- 问题2：如何更新每顶点的估计距离?
  - 当前到顶点 v 的最短路径：$< s, w, v >$，距离为$dist[v]$
  - 通过顶点 u 的新路径：$< s, \ldots, u, v >$，距离为$dist[u] + w(u, v)$



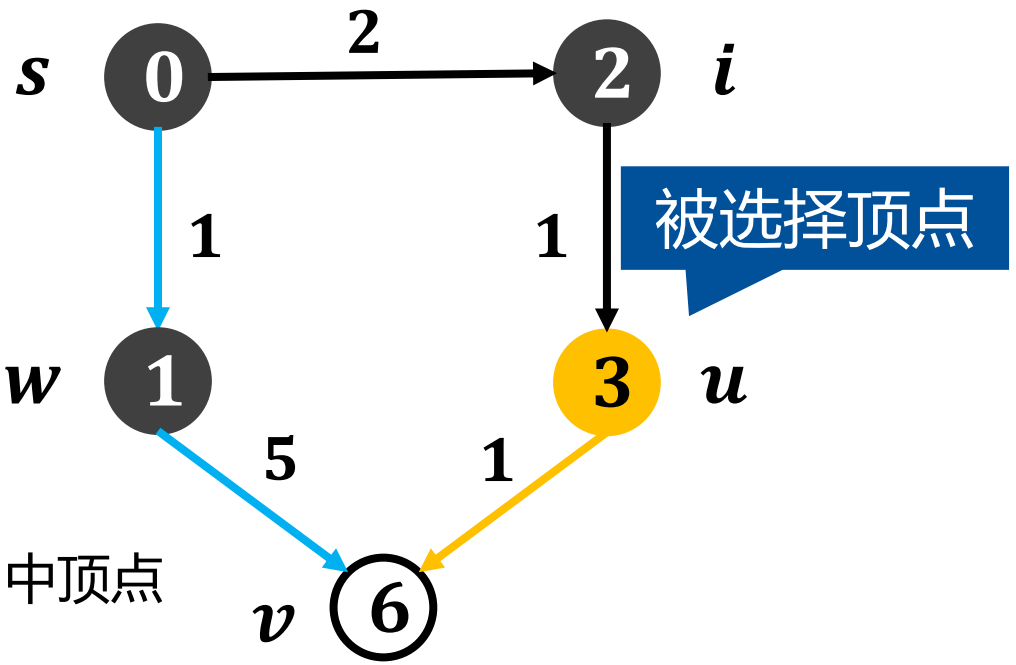
算法思想

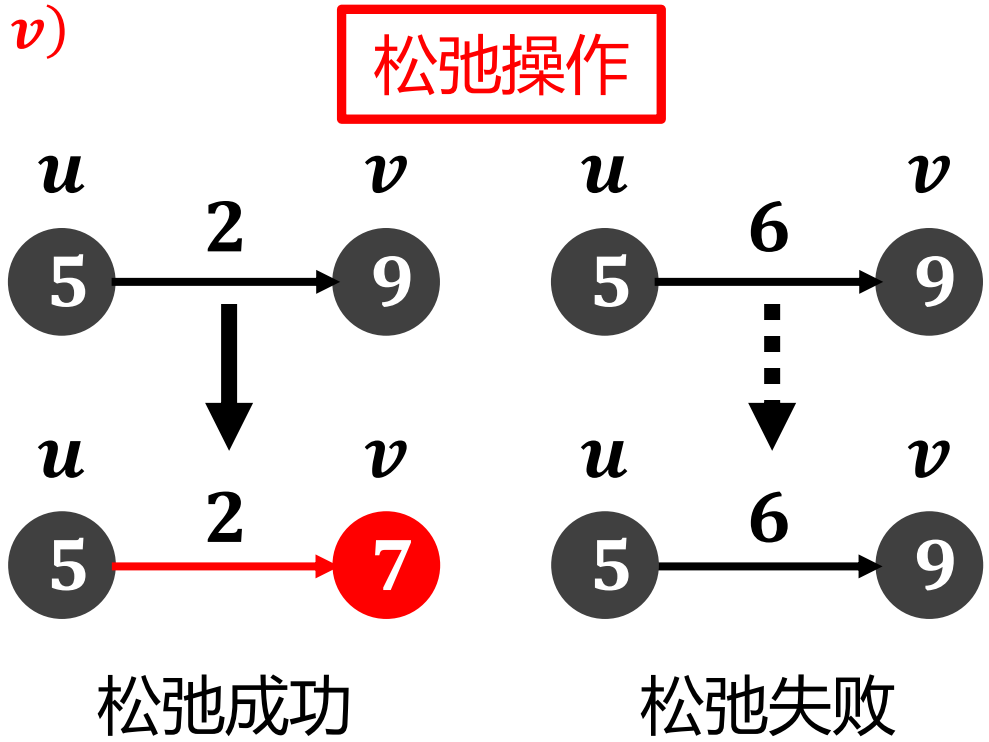
- 问题2：如何更新每顶点的估计距离?
  - 当前到顶点 v 的最短路径：$< s, w, v >$，距离为$dist[v]$
  - 通过顶点 u 的新路径：$< s, \ldots, u, v >$，距离为$dist[u] + w(u, v)$
  - 如果新路径更短($dist[u] + w(u, v) < dist[v]$)
    - 更新$dist[v]$：$dist[v] = dist[u] + w(u, v)$



算法思想

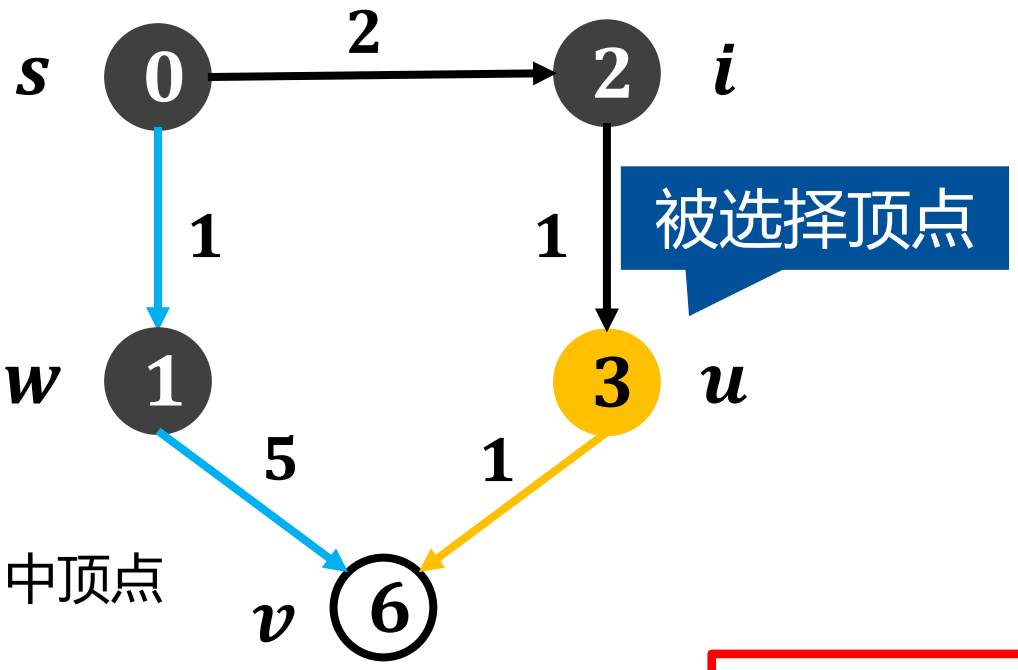
- 问题2：如何更新每顶点的估计距离?
  - 当前到顶点 v 的最短路径：$< s, w, v >$，距离为$dist[v]$
  - 通过顶点 u 的新路径：$< s, \ldots, u, v >$，距离为$dist[u] + w(u, v)$
  - 如果新路径更短($dist[u] + w(u, v) < dist[v]$)
    - 更新$dist[v]$：$dist[v] = dist[u] + w(u, v)$

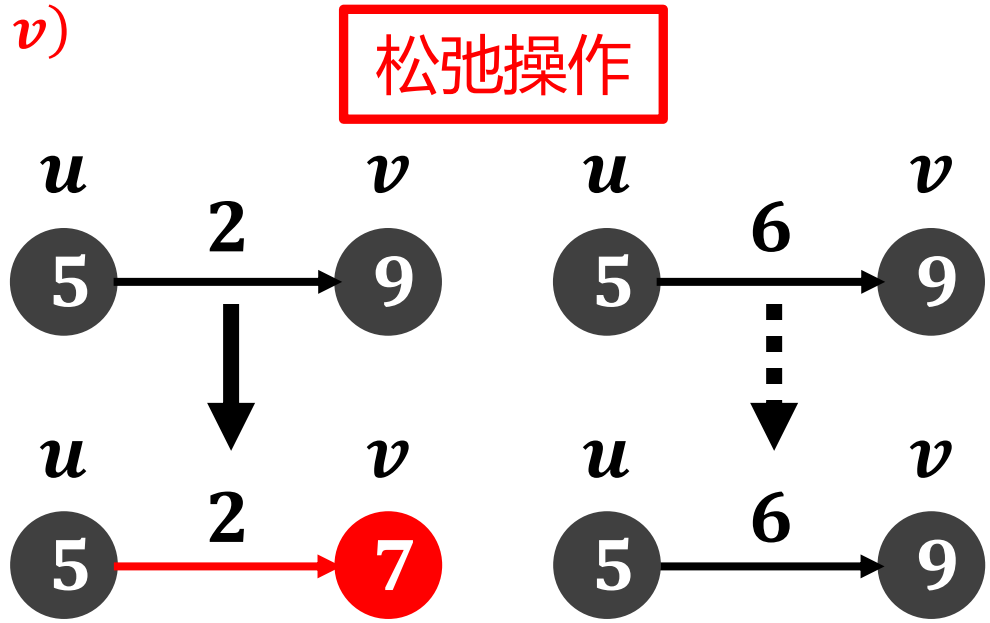




算法思想

- 问题2：如何更新每顶点的估计距离?
  - 当前到顶点 v 的最短路径：$< s, w, v >$，距离为$dist[v]$
  - 通过顶点 u 的新路径：$< s, \ldots, u, v >$，距离为$dist[u] + w(u, v)$
  - 如果新路径更短($dist[u] + w(u, v) < dist[v]$)
    - 更新$dist[v]$：$dist[v] = dist[u] + w(u, v)$





松弛成功　　松弛失败

V_A中顶点

$V - V_A$中顶点

松弛操作的效果：$dist[v] \leq dist[u] + w(u, v)$

提纲

问题背景

算法思想

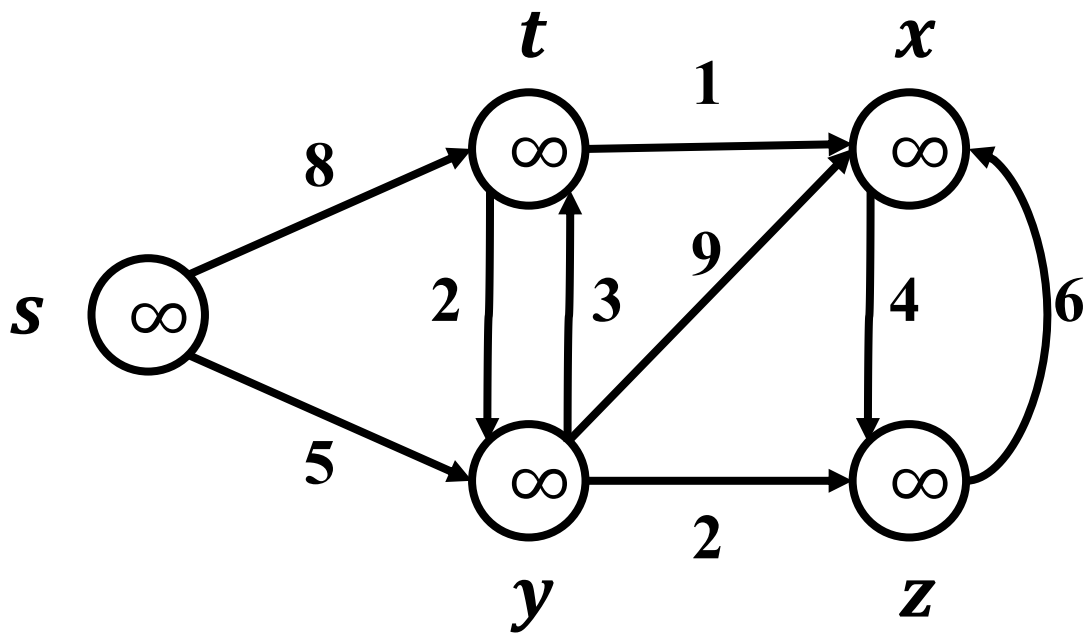
算法实例

算法分析

算法性质

算法实例

V	s	t	x	y	z
$color$	W	W	W	W	W
$pred$	N	N	N	N	N
$dist$	∞	∞	∞	∞	∞



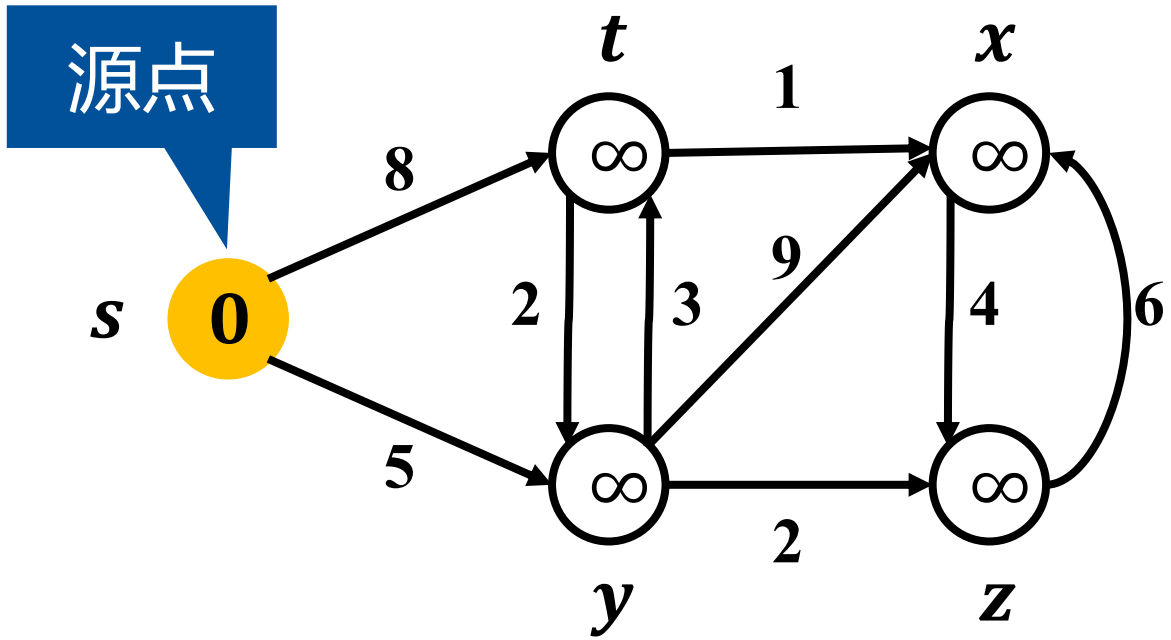
V_A中顶点

$V - V_A$中顶点

u 被选中顶点

算法实例

V	s	t	x	y	z
$color$	W	W	W	W	W
$pred$	N	N	N	N	N
$dist$	0	∞	∞	∞	∞



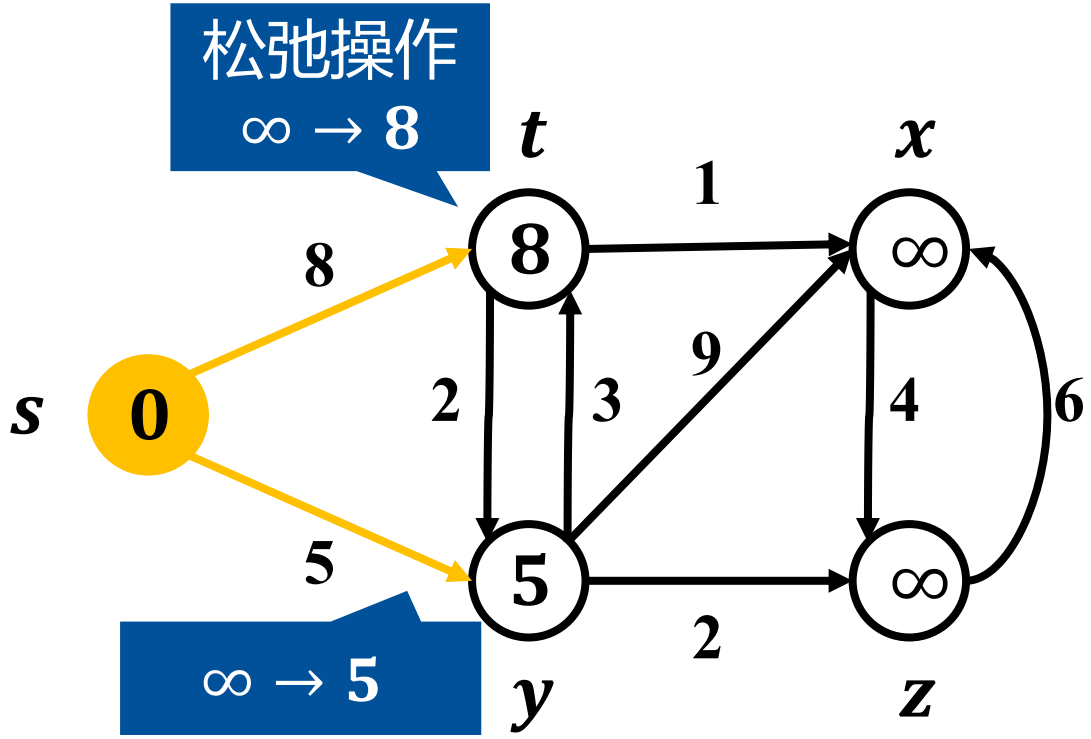
V_A中顶点

$V - V_A$中顶点

u 被选中顶点

算法实例

V	s	t	x	y	z
$color$	W	W	W	W	W
$pred$	N	s	N	s	N
$dist$	0	8	∞	5	∞



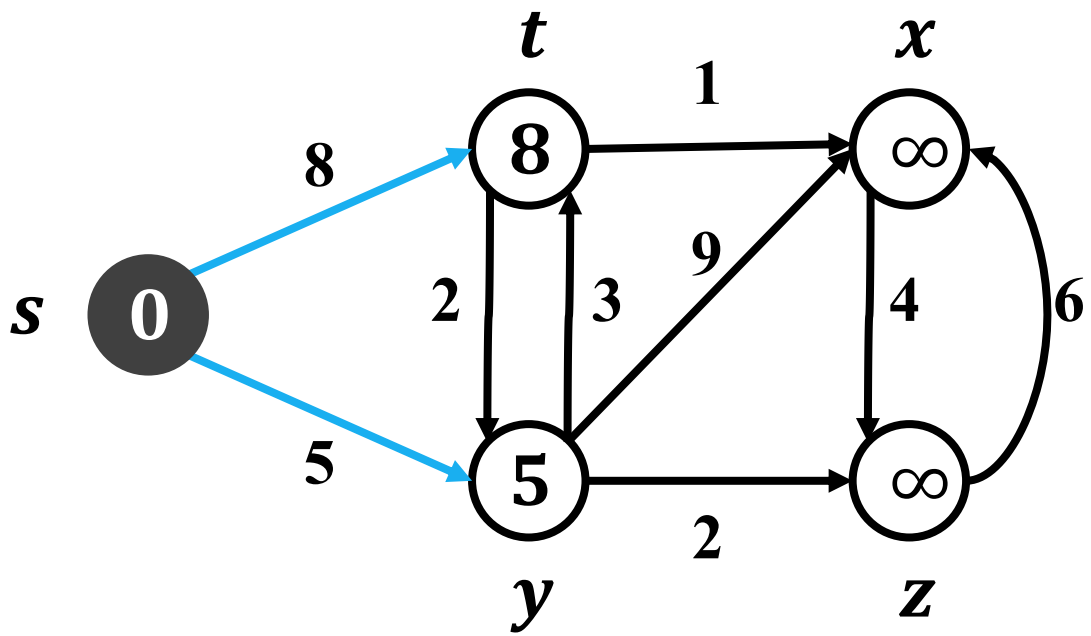
V_A中顶点

$V - V_A$中顶点

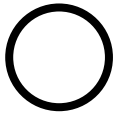
u 被选中顶点

算法实例

V	s	t	x	y	z
$color$	B	W	W	W	W
$pred$	N	s	N	s	N
$dist$	0	8	∞	5	∞



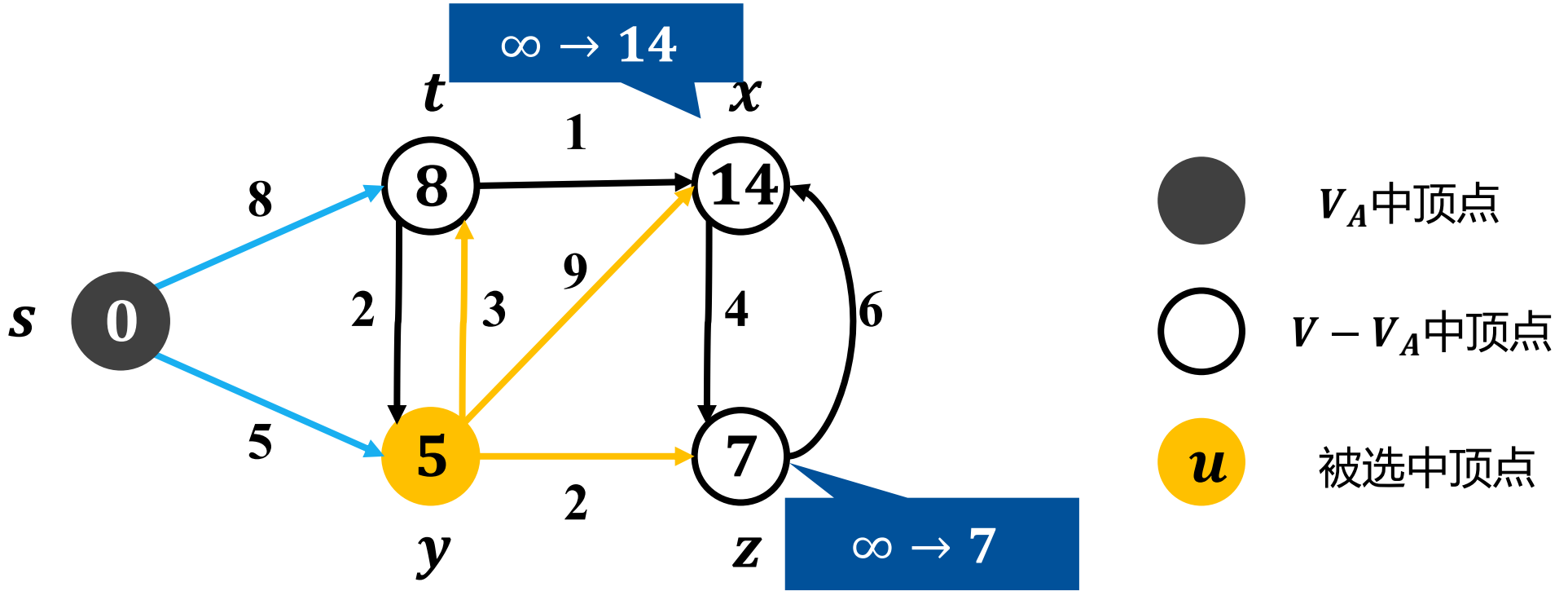

V_A中顶点


$V-V_A$中顶点

被选中顶点

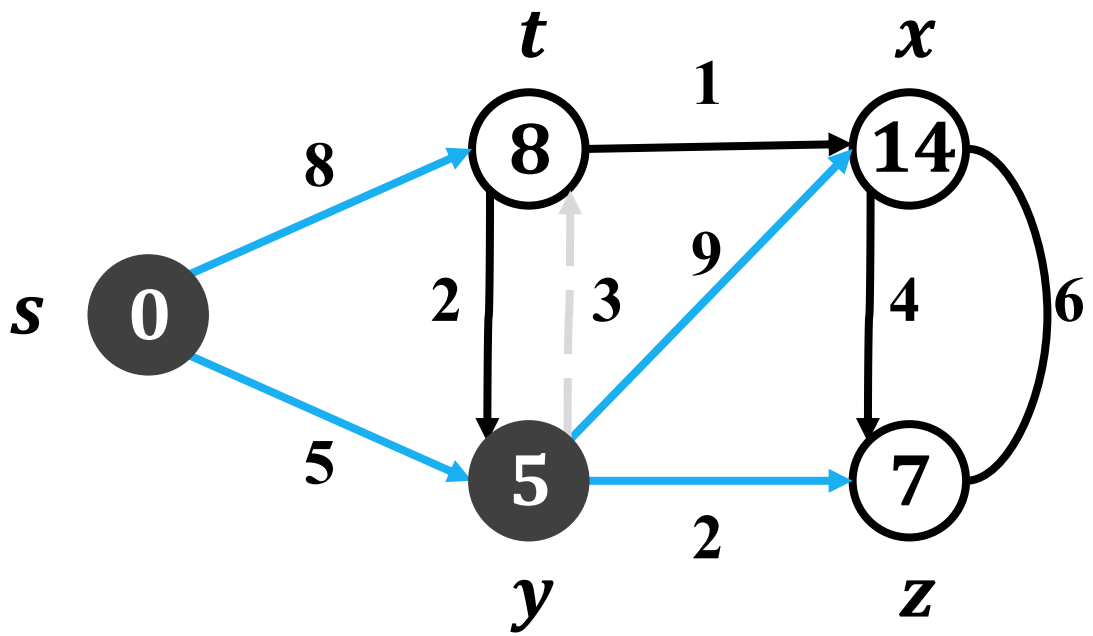
算法实例

V	s	t	x	y	z
$color$	B	W	W	W	W
$pred$	N	s	y	s	y
$dist$	0	8	14	5	7



算法实例

V	s	t	x	y	z
$color$	B	W	W	B	W
$pred$	N	s	y	s	y
$dist$	0	8	14	5	7



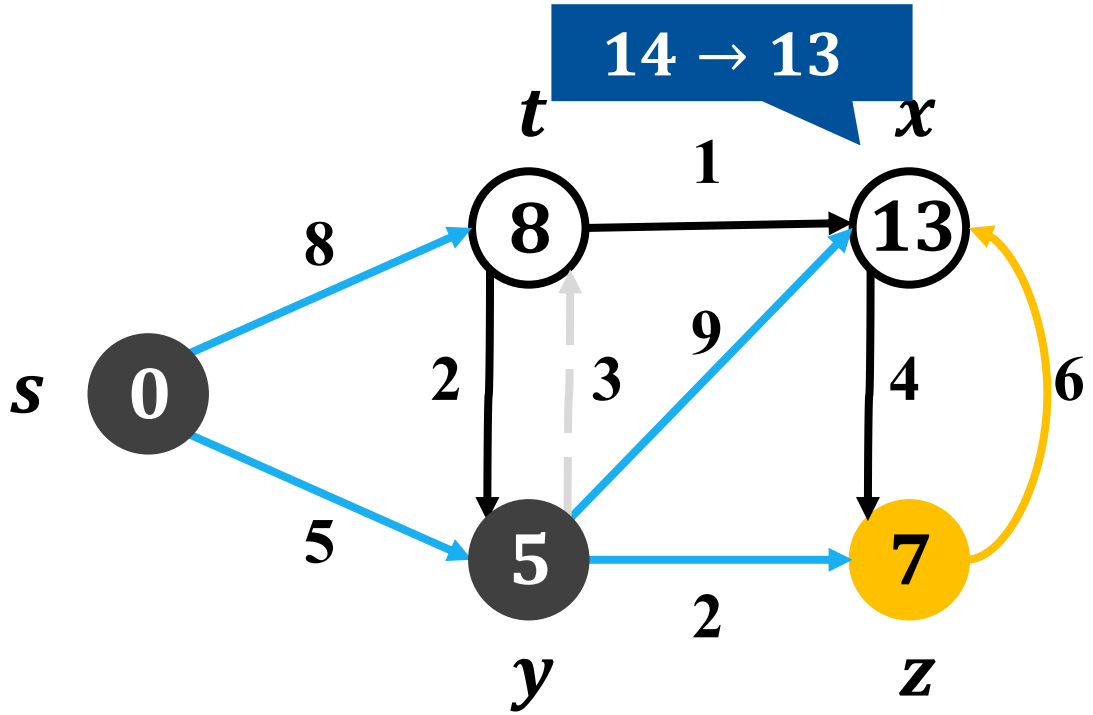
V_A中顶点

$V - V_A$中顶点

u 被选中顶点

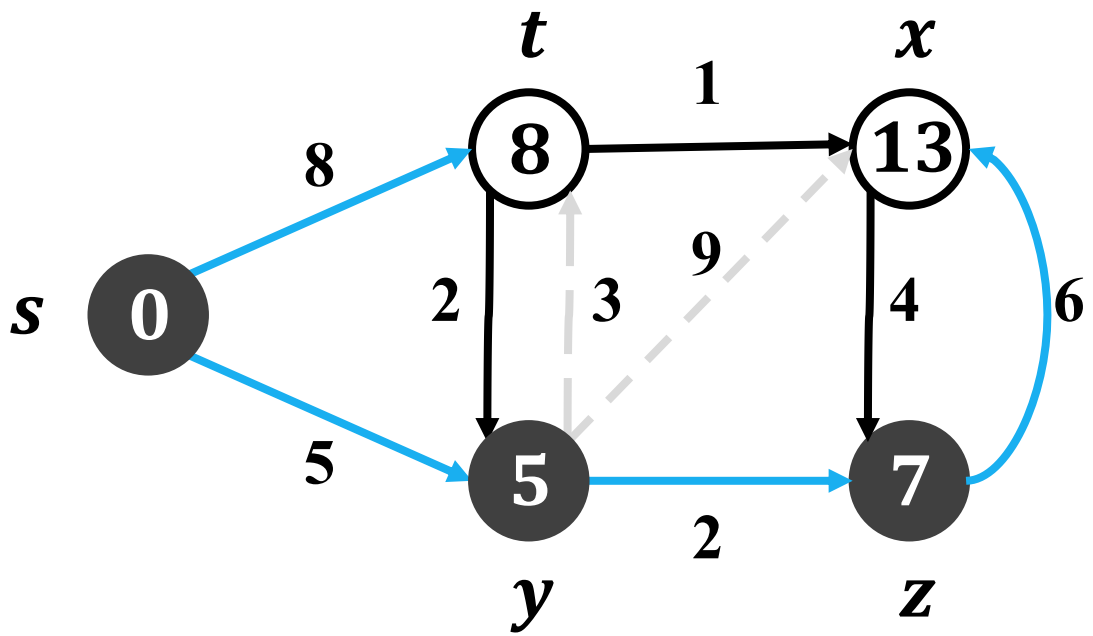
算法实例

V	s	t	x	y	z
$color$	B	W	W	B	W
$pred$	N	s	z	s	y
$dist$	0	8	13	5	7



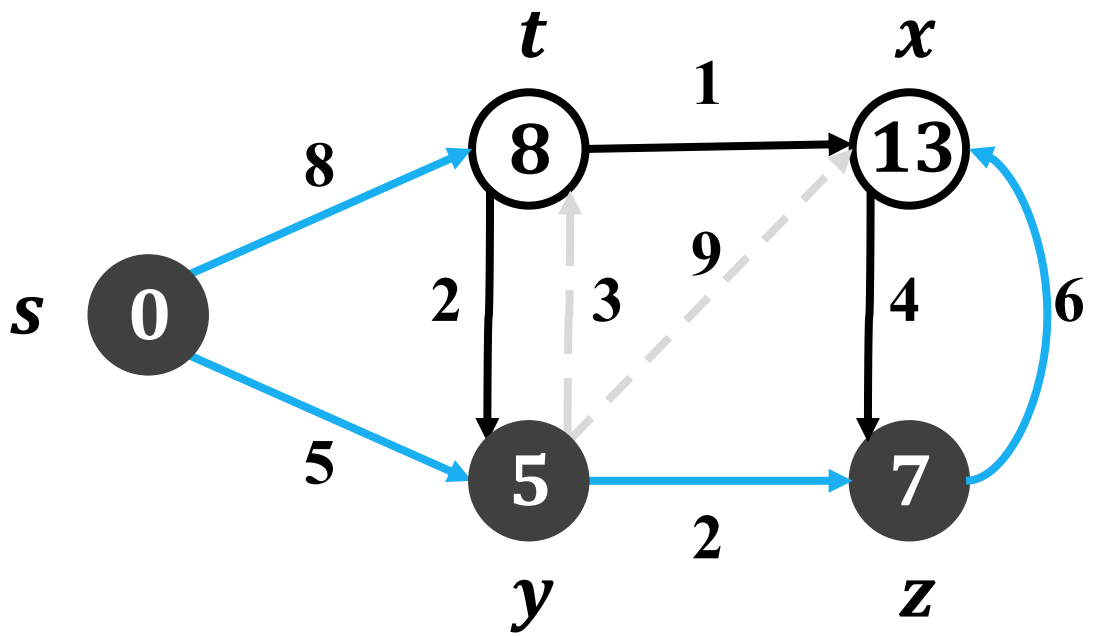
算法实例

V	s	t	x	y	z
$color$	B	W	W	B	B
$pred$	N	s	z	s	y
$dist$	0	8	13	5	7



算法实例

V	s	t	x	y	z
$color$	B	W	W	B	B
$pred$	N	s	z	s	y
$dist$	0	8	13	5	7



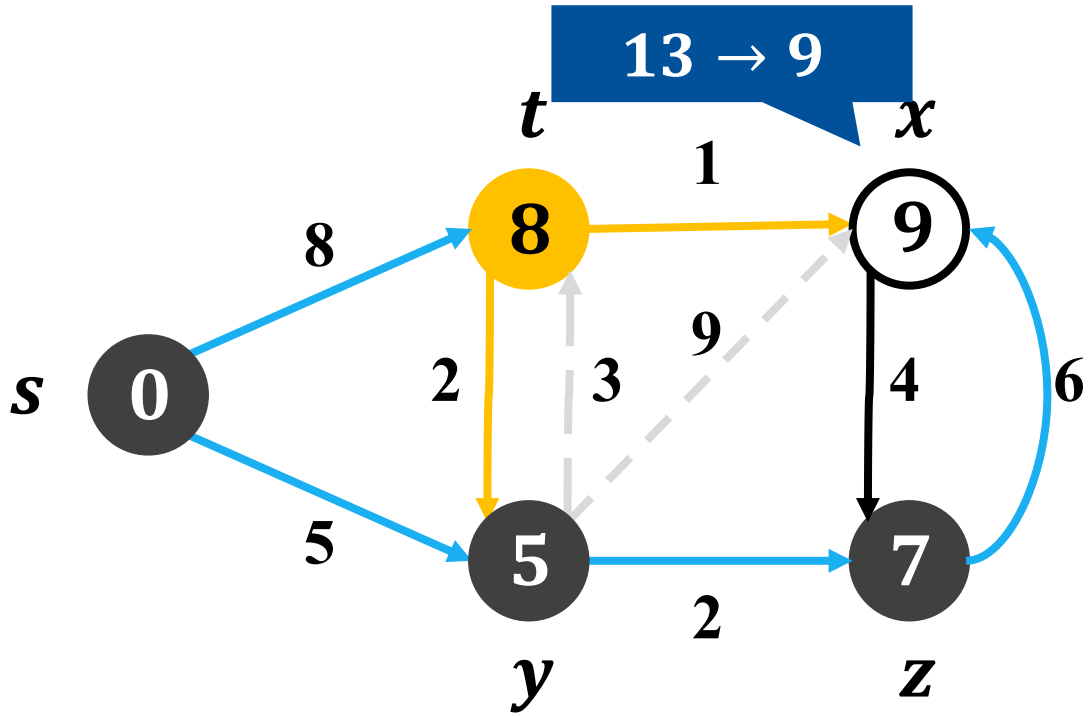
V_A中顶点

$V - V_A$中顶点

u 被选中顶点

算法实例

V	s	t	x	y	z
$color$	B	W	W	B	B
$pred$	N	s	t	s	y
$dist$	0	8	9	5	7



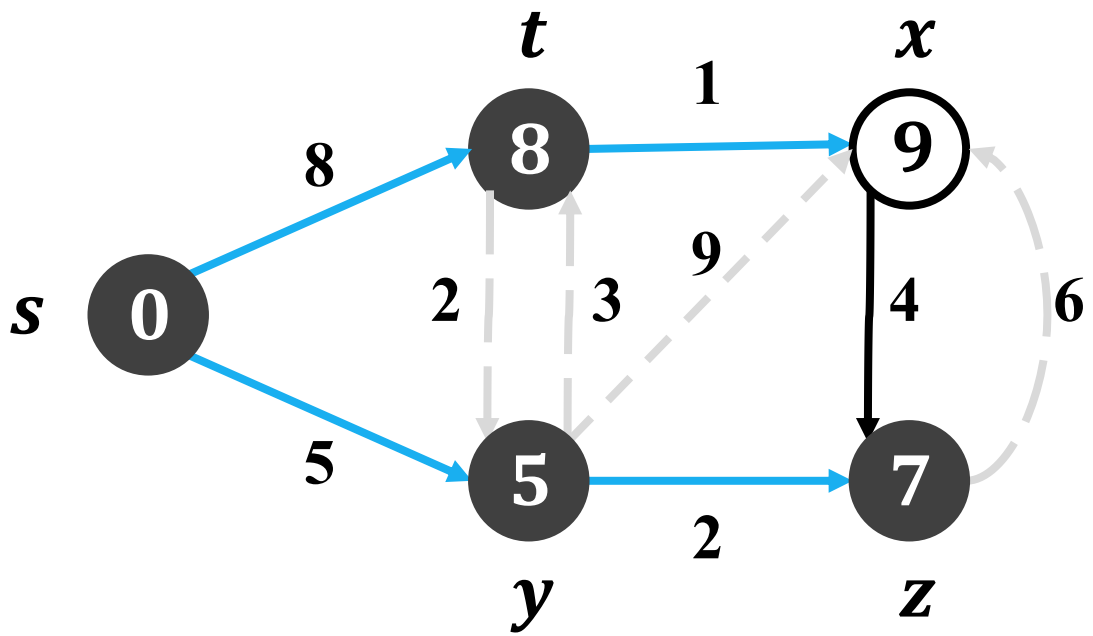
V_A中顶点

$V - V_A$中顶点

u 被选中顶点

算法实例

V	s	t	x	y	z
$color$	B	B	W	B	B
$pred$	N	s	t	s	y
$dist$	0	8	9	5	7



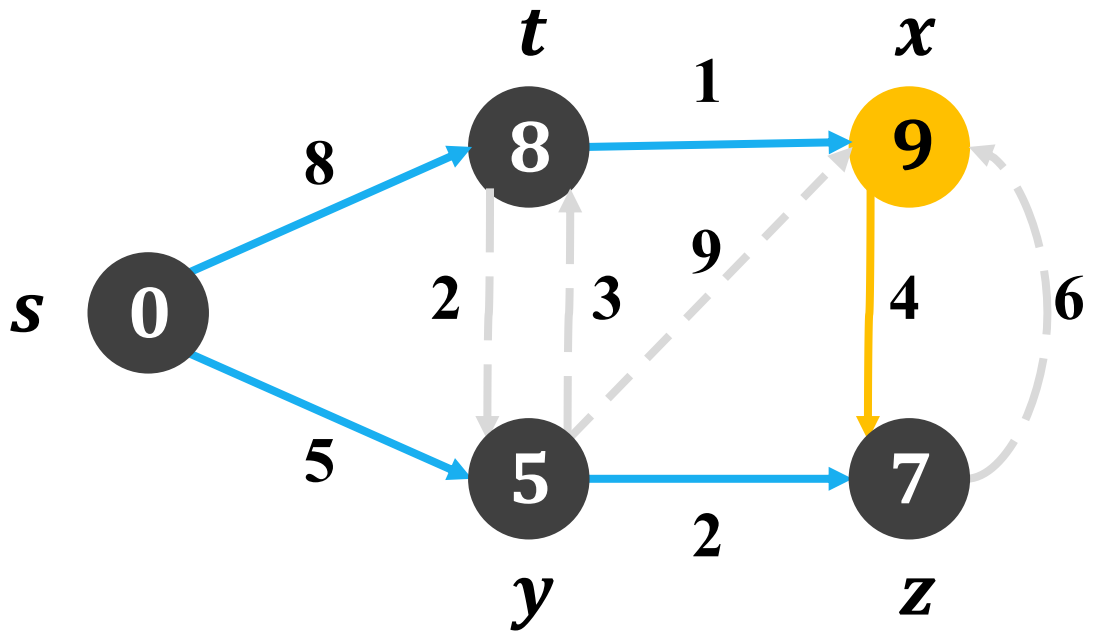
V_A中顶点

$V - V_A$中顶点

u 被选中顶点

算法实例

V	s	t	x	y	z
$color$	B	B	W	B	B
$pred$	N	s	t	s	y
$dist$	0	8	9	5	7



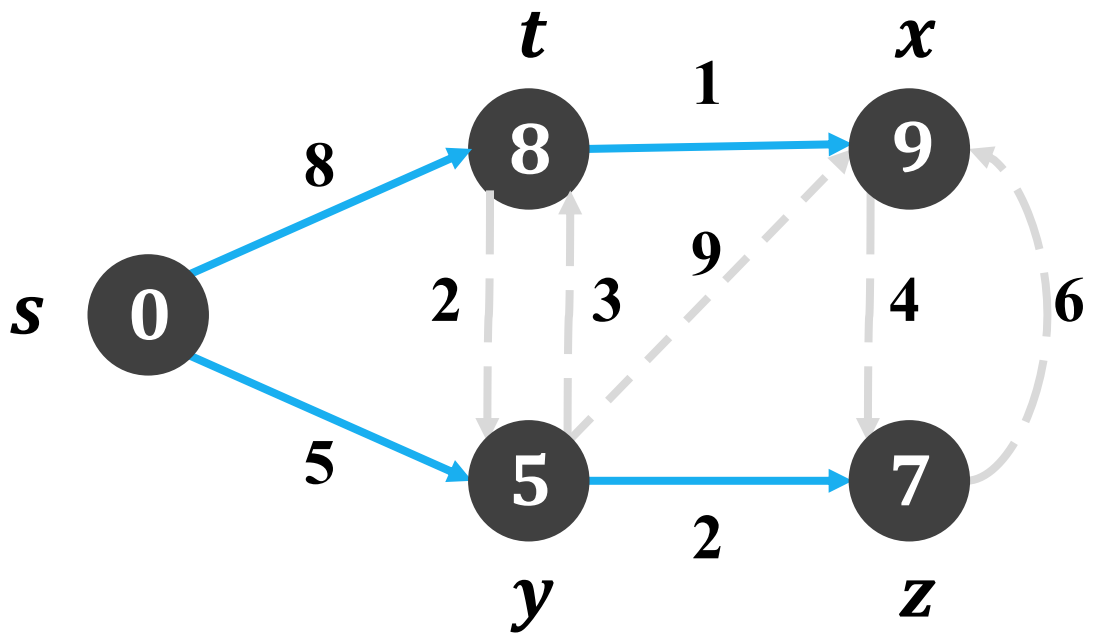
V_A中顶点

$V - V_A$中顶点

u 被选中顶点

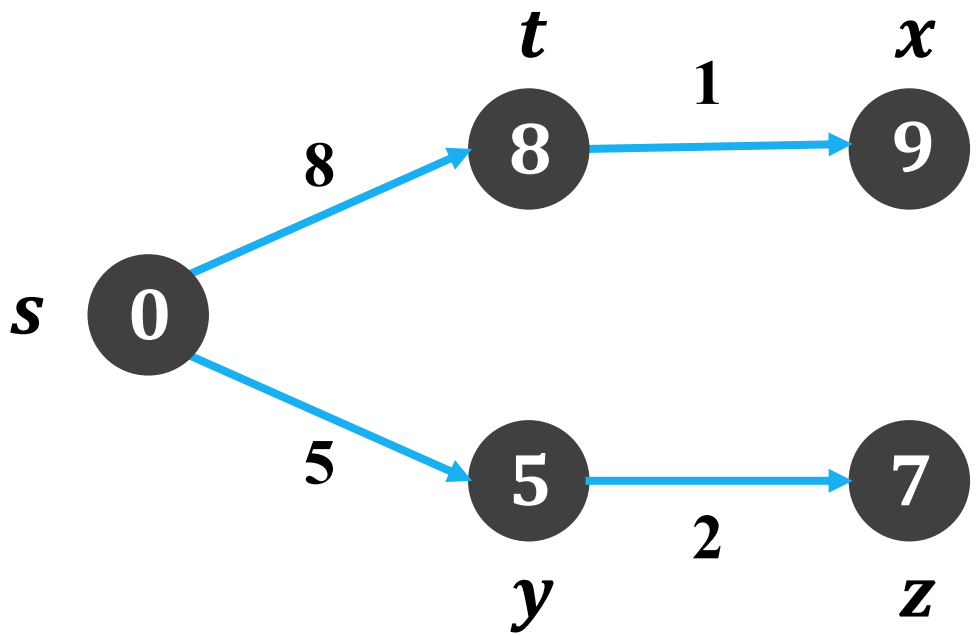
算法实例

V	s	t	x	y	z
$color$	B	B	B	B	B
$pred$	N	s	t	s	y
$dist$	0	8	9	5	7



算法实例

V	s	t	x	y	z
$color$	B	B	B	B	B
$pred$	N	s	t	s	y
$dist$	0	8	9	5	7



提纲

问题背景

算法思想

算法实例

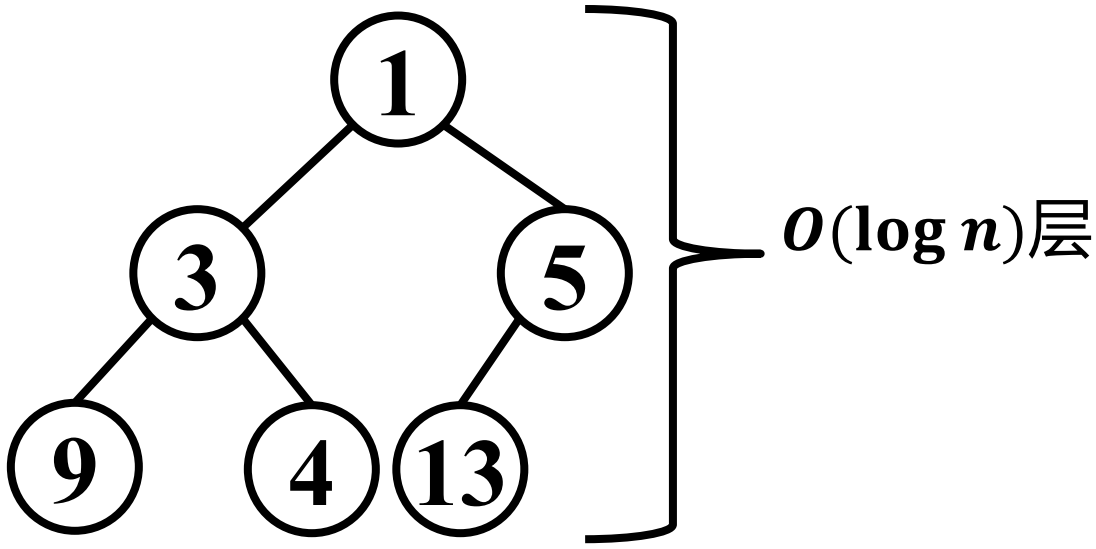
算法分析

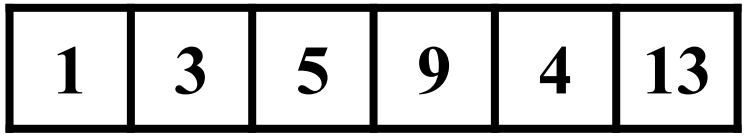
算法性质

数据结构：优先队列

优先队列

- 队列中每个元素有一个关键字，依据关键字大小离开队列
- 通过二叉堆来实现优先队列
  - $Q.Insert()$ 时间复杂度$O(\log n)$
  - $Q.ExtractMin()$ 时间复杂度$O(\log n)$
  - $Q.DecreaseKey()$ 时间复杂度$O(\log n)$





优先队列Q

伪代码

- **Dijkstra-PriQueue(G, s)**

输入: 图$G =< V, E, W >$, 源点s
输出: 单源最短路径P
新建一维数组$color[1..|V|]$, $dist[1..|V|]$, $pred[1..|V|]$
新建空优先队列Q
//初始化
for $u \in V$ **do**
  $color[u] \leftarrow WHITE$
  $dist[u] \leftarrow \infty$
  $pred[u] \leftarrow NULL$
end
$dist[s] \leftarrow 0$
$Q.Insert(V, dist)$

初始化辅助数组

伪代码

- **Dijkstra-PriQueue(G, s)**

输入: 图$G =< V, E, W >$, 源点s
输出: 单源最短路径P
新建一维数组$color[1..|V|]$, $dist[1..|V|]$, $pred[1..|V|]$
新建空优先队列Q
//初始化
for $u \in V$ do
    $color[u] \leftarrow WHITE$
    $dist[u] \leftarrow \infty$
    $pred[u] \leftarrow NULL$
end
$dist[s] \leftarrow 0$
$Q.Insert(V, dist)$

初始化源点和优先队列

伪代码

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.adj[v] do
        if dist[v] + w(v, u) < dist[u] then
            dist[u] ← dist[v] + w(v, u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

依次计算到各点最短路

伪代码

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.adj[v] do
        if dist[v] + w(v,u) < dist[u] then
            dist[u] ← dist[v] + w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

选择最小估计距离的白色顶点

伪代码

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.adj[v] do
        if dist[v] + w(v,u) < dist[u] then
            dist[u] ← dist[v] + w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

对从 u 出发的边进行松弛

伪代码

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.adj[v] do
        if dist[v] + w(v,u) < dist[u] then
            dist[u] ← dist[v] + w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

松弛操作，更新距离上界

伪代码

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.adj[v] do
        if dist[v] + w(v,u) < dist[u] then
            dist[u] ← dist[v] + w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

记录前驱顶点

伪代码

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.adj[v] do
        if dist[v] + w(v,u) < dist[u] then
            dist[u] ← dist[v] + w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

更新优先队列中关键字

伪代码

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.adj[v] do
        if dist[v] + w(v,u) < dist[u] then
            dist[u] ← dist[v] + w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

标记顶点计算完成

时间复杂度分析

- **Dijkstra-PriQueue(G, s)**

输入: 图$G =< V, E, W >$, 源点s
输出: 单源最短路径P
新建一维数组$color[1..|V|]$, $dist[1..|V|]$, $pred[1..|V|]$
新建空优先队列Q
//初始化
for $u \in V$ **do**
  $color[u] \leftarrow WHITE$
  $dist[u] \leftarrow \infty$
  $pred[u] \leftarrow NULL$
end
$dist[s] \leftarrow 0$
$Q.Insert(V, dist)$

$\boldsymbol{O(|V|)}$

时间复杂度分析

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()                                  ------ O(log|V|)
    for u ∈ G.adj[v] do
        if dist[v] + w(v, u) < dist[u] then
            dist[u] ← dist[v] + w(v, u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))                 --  O(log|V|)
        end
    end
    color[v] ← BLACK
end
```

时间复杂度分析

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()                                   O(log|V|)
    for u ∈ G.adj[v] do
        if dist[v] + w(v,u) < dist[u] then
            dist[u] ← dist[v] + w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))      O(log|V|)   } O(deg(u) · log|V|)
        end
    end
    color[v] ← BLACK
end
```

时间复杂度分析

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.adj[v] do
        if dist[v] + w(v,u) < dist[u] then
            dist[u] ← dist[v] + w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

$O(\log|V|)$

$O(\deg(u) \cdot \log|V|)$

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

时间复杂度分析

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.adj[v] do
        if dist[v] + w(v,u) < dist[u] then
            dist[u] ← dist[v] + w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

$O(\log|V|)$

$O(\deg(u)\cdot\log|V|)$

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

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

$$\sum_{u\in V}\deg(u)=2|E|$$

时间复杂度分析

- **Dijkstra-PriQueue(G, s)**

```
//执行单源最短路径算法
while 优先队列Q非空 do
    v ← Q.ExtractMin()
    for u ∈ G.adj[v] do
        if dist[v] + w(v,u) < dist[u] then
            dist[u] ← dist[v] + w(v,u)
            pred[u] ← v
            Q.DecreaseKey((u, dist[u]))
        end
    end
    color[v] ← BLACK
end
```

时间复杂度$O(|E| \cdot \log|V|)$

提纲

问题背景

算法思想

算法实例

算法分析

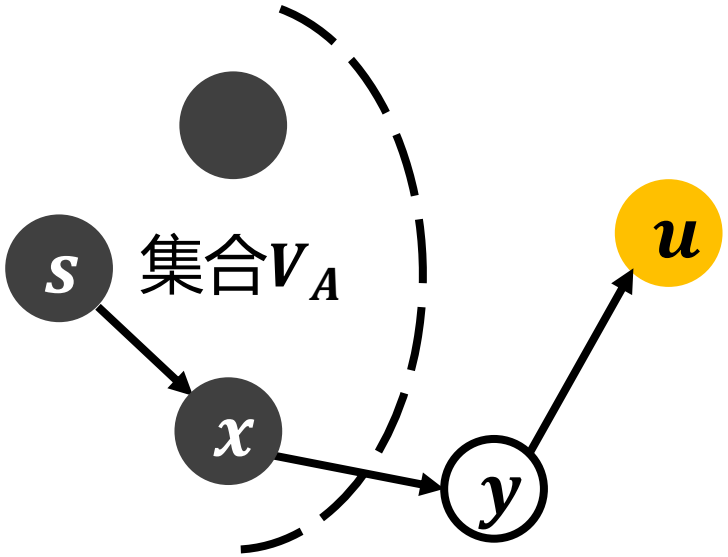
算法性质

正确性证明

- 定理：**Dijkstra**算法中，顶点 u 被添加到 V_A 时，$dist[u] = \delta(s, u)$

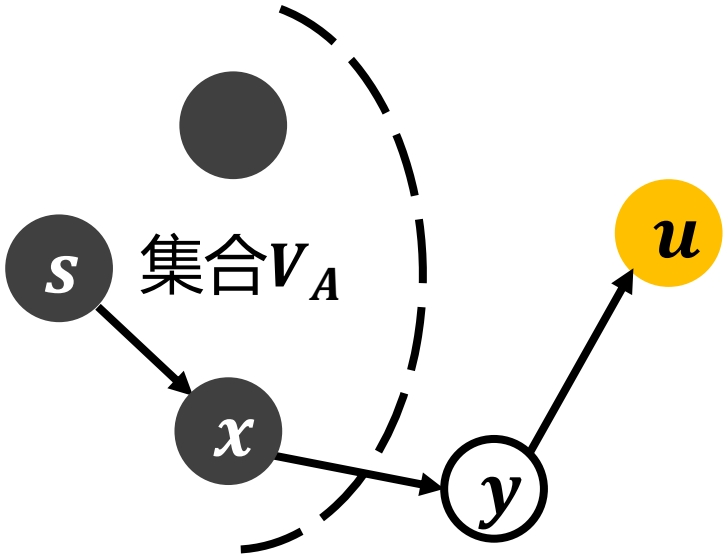
正确性证明

- 定理：Dijkstra算法中，顶点 u 被添加到 V_A 时，$dist[u] = \delta(s,u)$
- 证明：
  - 采用反证法，假设Dijkstra算法将顶点u添加到V_A时，$dist[u] \neq \delta(s,u)$



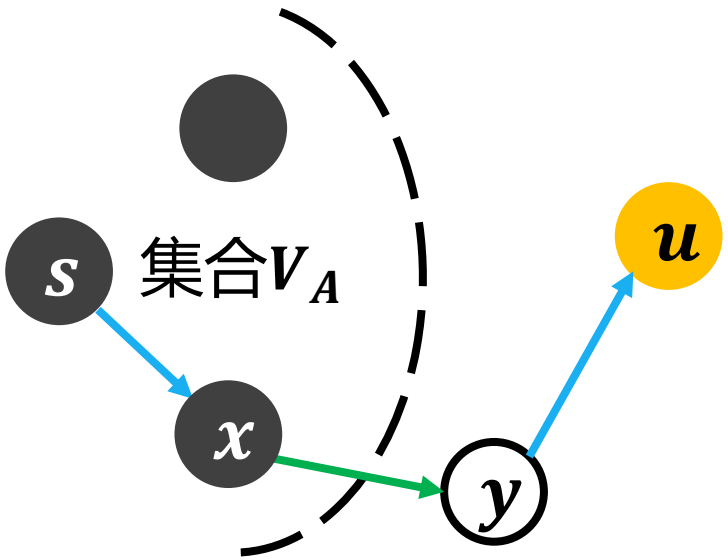
正确性证明

- 定理：Dijkstra算法中，顶点 u 被添加到 V_A 时，$dist[u] = \delta(s,u)$
- 证明：
  - 采用反证法，假设Dijkstra算法将顶点u添加到V_A时，$dist[u] \neq \delta(s,u)$
  - 由于$dist[u]$作为$\delta(s,u)$的上界，故$dist[u] > \delta(s,u)$



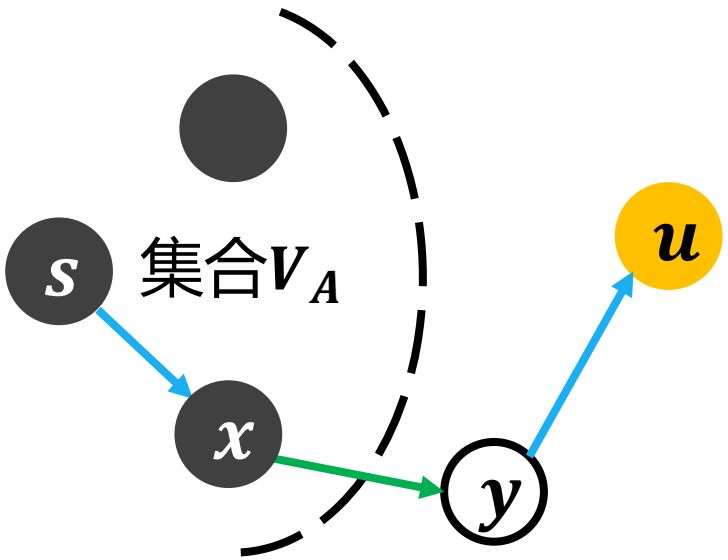
正确性证明

- 定理：Dijkstra算法中，顶点 u 被添加到 V_A 时，$dist[u] = \delta(s,u)$
- 证明：
  - 采用反证法，假设Dijkstra算法将顶点u添加到V_A时，$dist[u] \neq \delta(s,u)$
  - 由于$dist[u]$作为$\delta(s,u)$的上界，故$dist[u] > \delta(s,u)$
  - 应存在一条长度为$\delta(s,u)$的从s到u的最短路径，不妨设其为$< s, \dots, x, y, \dots, u >$，其中边$(x,y)$横跨$< V_A, V - V_A >$，$x \in V_A, y \in V - V_A$



正确性证明

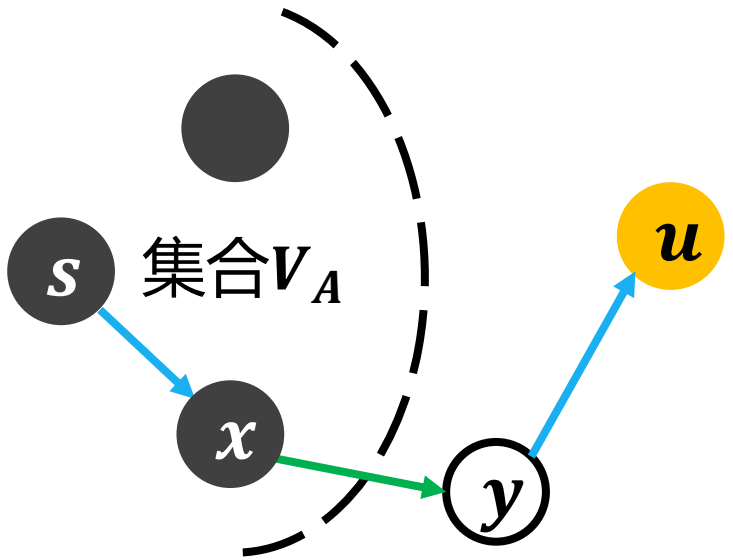
- 定理：Dijkstra算法中，顶点 u 被添加到 V_A 时，$dist[u] = \delta(s,u)$
- 证明：
  - 采用反证法，假设Dijkstra算法将顶点u添加到V_A时，$dist[u] \neq \delta(s,u)$
  - 由于$dist[u]$作为$\delta(s,u)$的上界，故$dist[u] > \delta(s,u)$
  - 应存在一条长度为$\delta(s,u)$的从s到u的最短路径，不妨设其为$< s,\dots,x,y,\dots,u >$，其中边(x,y)横跨$< V_A, V-V_A >$，$x \in V_A, y \in V-V_A$
  - 算法令V_A 中的顶点满足：$dist[x] = \delta(s,x), x \in V_A$



正确性证明

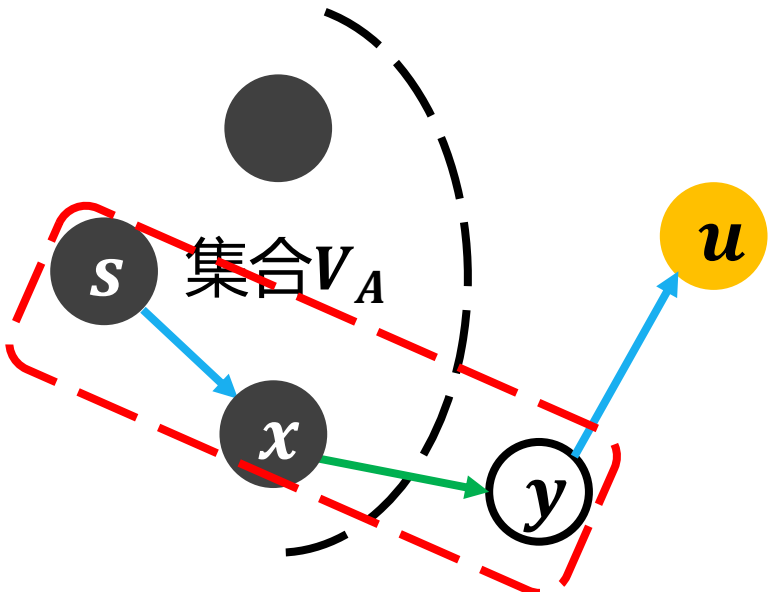
- 定理：Dijkstra算法中，顶点 u 被添加到 V_A 时，$dist[u] = \delta(s,u)$
- 证明：
  - 采用反证法，假设Dijkstra算法将顶点u添加到V_A时，$dist[u] \neq \delta(s,u)$
  - 由于$dist[u]$作为$\delta(s,u)$的上界，故$dist[u] > \delta(s,u)$
  - 应存在一条长度为$\delta(s,u)$的从s到u的最短路径，不妨设其为$< s, \dots, x, y, \dots, u >$，其中边$(x,y)$横跨$< V_A, V - V_A >$，$x \in V_A, y \in V - V_A$
  - 算法令V_A 中的顶点满足：$dist[x] = \delta(s,x), x \in V_A$

问题：$dist[y]$和$\delta(s,y)$具有何种关系?



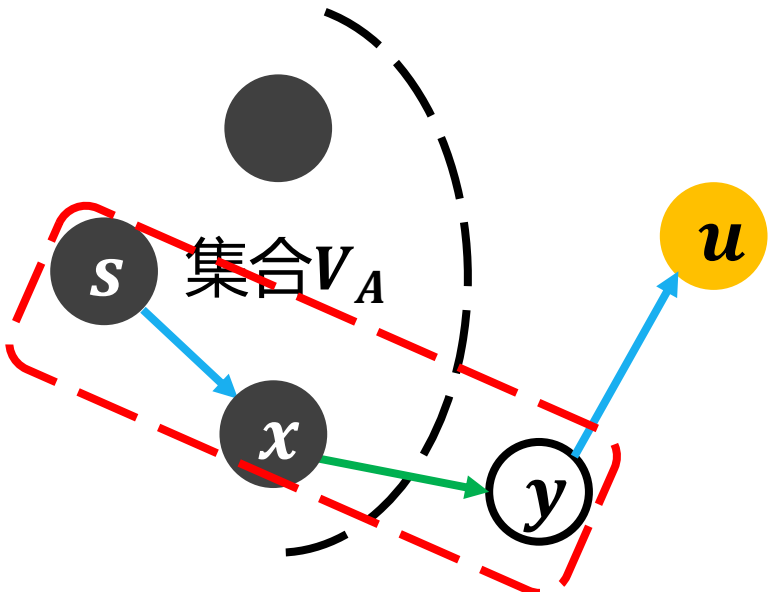
正确性证明

- 定理：**Dijkstra**算法中，顶点u被添加到V_A时，$dist[u] = \delta(s,u)$
- 证明（接上页）：
  - $< s, \dots, x, y >$是最短路径$< s, \dots, x, y, \dots, u >$的子路径



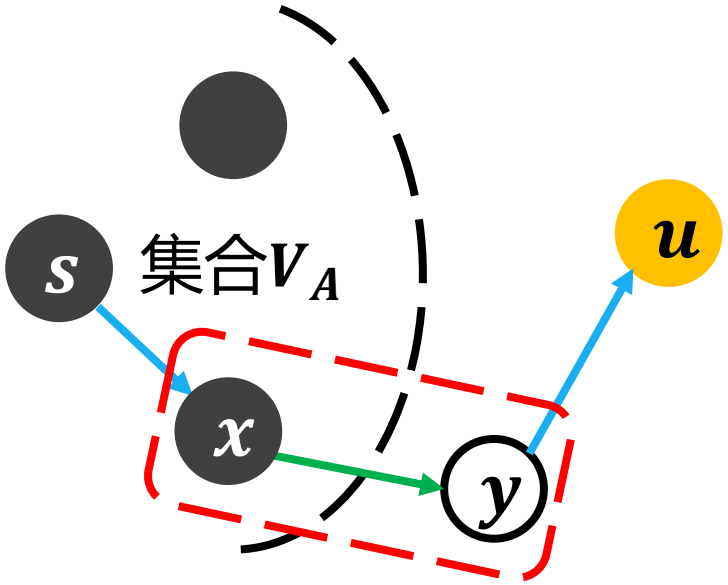
正确性证明

- 定理：Dijkstra算法中，顶点u被添加到V_A时，$dist[u] = \delta(s,u)$
- 证明（接上页）：
  - $< s, \dots, x, y >$是最短路径$< s, \dots, x, y, \dots, u >$的子路径，故：
    - $\delta(s,y) = \delta(s,x) + w(x,y) = dist[x] + w(x,y)$ （公式1）



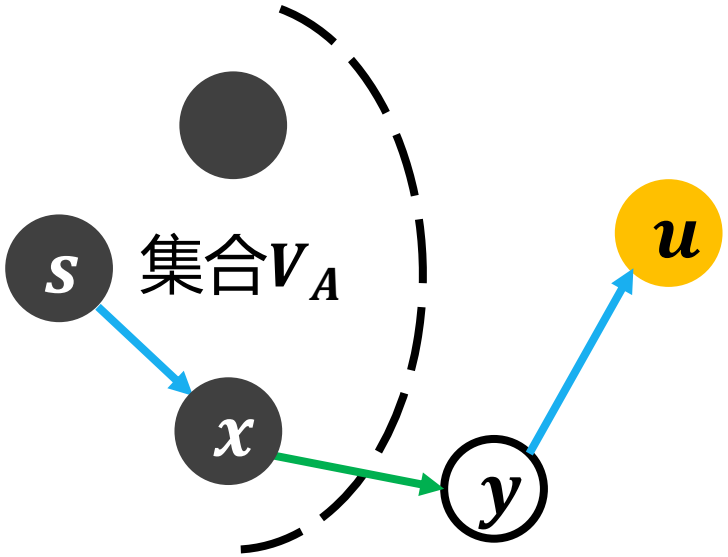
正确性证明

- 定理：Dijkstra算法中，顶点u被添加到V_A时，$dist[u] = \delta(s,u)$
- 证明（接上页）：
  - $< s, \dots, x, y >$是最短路径$< s, \dots, x, y, \dots, u >$的子路径，故：
    - $\delta(s,y) = \delta(s,x) + w(x,y) = dist[x] + w(x,y)$ （公式1）
  - 算法对顶点x出发的所有边（包括边(x,y)）已进行松弛操作，故：
    - $dist[y] \le dist[x] + w(x,y)$（公式2）



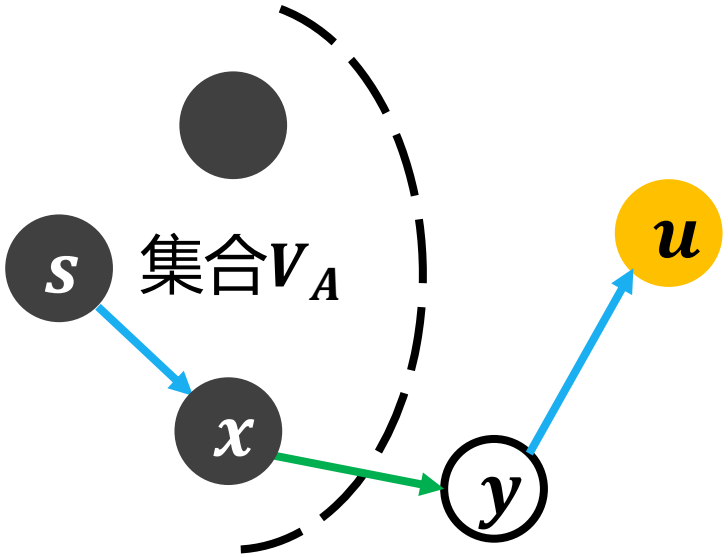
正确性证明

- 定理：Dijkstra算法中，顶点u被添加到V_A时，$dist[u] = \delta(s,u)$
- 证明（接上页）：
  - $< s, \dots, x, y >$是最短路径$< s, \dots, x, y, \dots, u >$的子路径，故：
    - $\delta(s,y) = \delta(s,x) + w(x,y) = dist[x] + w(x,y)$ （公式1）
  - 算法对顶点x出发的所有边（包括边(x,y)）已进行松弛操作，故：
    - $dist[y] \le dist[x] + w(x,y)$（公式2）
  - 合并上述公式1和公式2，可得$dist[y] = \delta(s,y)$



正确性证明

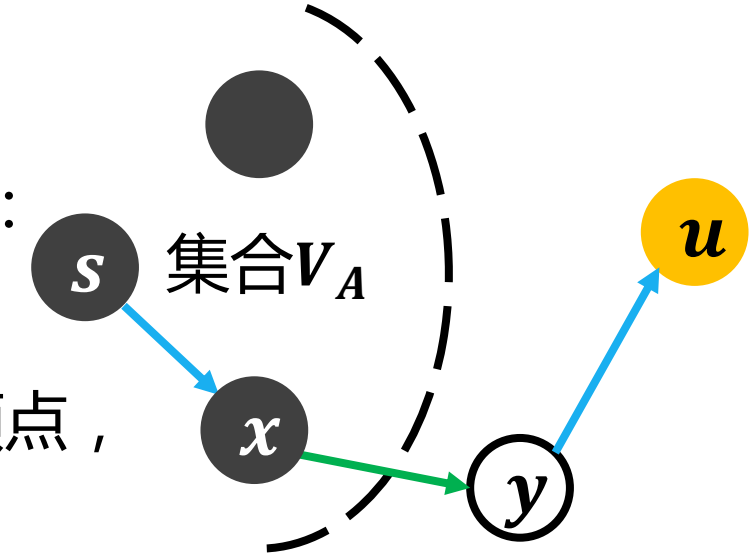
- 定理：Dijkstra算法中，顶点u被添加到V_A时，$dist[u] = \delta(s,u)$
- 证明（接上页）：
  - $< s, \dots, x, y >$是最短路径$< s, \dots, x, y, \dots, u >$的子路径，故：
    - $\delta(s,y) = \delta(s,x) + w(x,y) = dist[x] + w(x,y)$ （公式1）
  - 算法对顶点x出发的所有边（包括边(x,y)）已进行松弛操作，故：
    - $dist[y] \le dist[x] + w(x,y)$（公式2）
  - 合并上述公式1和公式2，可得$dist[y] = \delta(s,y)$
  - 最短路径$< s, \dots, x, y, \dots, u >$中，$y$出现在$u$之前，故：
    - $dist[u] > \delta(s,u) \ge \delta(s,y) = dist[y]$



正确性证明

- 定理：Dijkstra算法中，顶点u被添加到V_A时，$dist[u] = \delta(s,u)$
- 证明（接上页）：
  - $< s, \dots, x, y >$是最短路径$< s, \dots, x, y, \dots, u >$的子路径，故：
    - $\delta(s,y) = \delta(s,x) + w(x,y) = dist[x] + w(x,y)$ （公式1）
  - 算法对顶点x出发的所有边（包括边(x,y)）已进行松弛操作，故：
    - $dist[y] \le dist[x] + w(x,y)$（公式2）
  - 合并上述公式1和公式2，可得$dist[y] = \delta(s,y)$
  - 最短路径$< s, \dots, x, y, \dots, u >$中，$y$出现在$u$之前，故：
    - $dist[u] > \delta(s,u) \ge \delta(s,y) = dist[y]$
  - $dist[u] > dist[y]$，$u \ne y$，u不应是下一个被添加顶点，

故产生矛盾



小结

	广度优先搜索	Dijkstra算法
适用范围	无权图	带权图 （所有边权为正）
数据结构	队列	优先队列
运行时间	$O(\|V\| + \|E\|)$	$O(\|E\| \cdot \log\|V\|)$