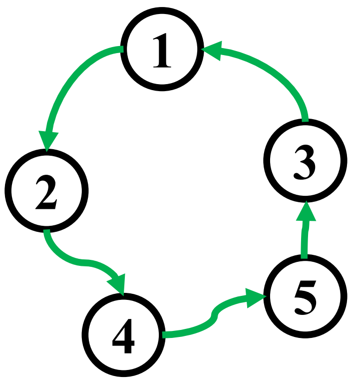
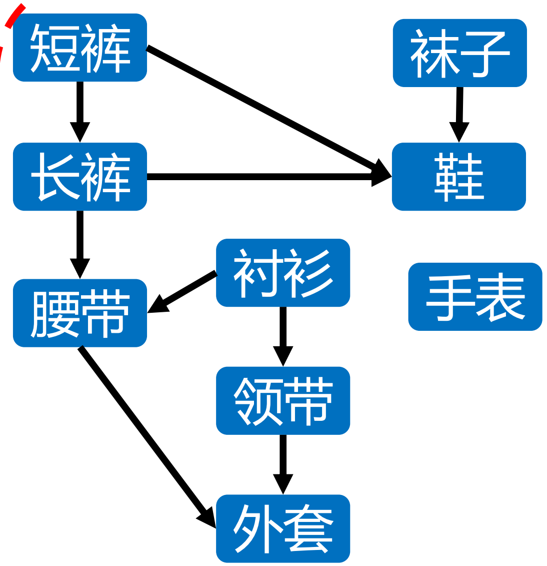
# 图算法篇：拓扑排序

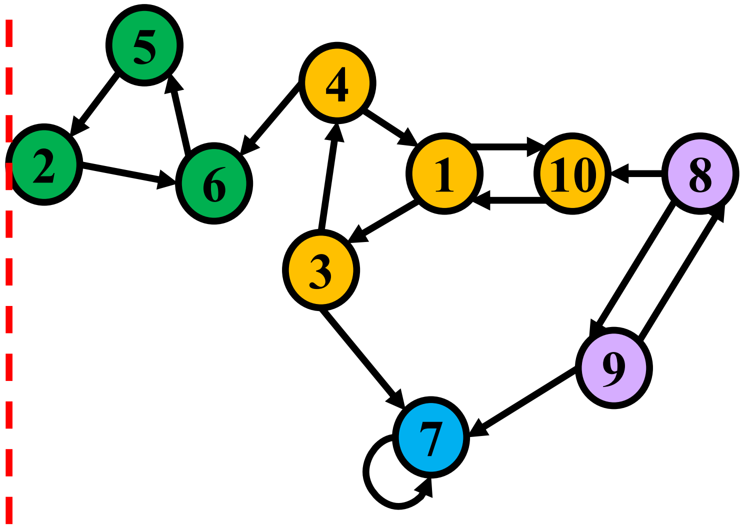
# 深度优先搜索应用




环路的存在性判断




拓扑排序




强连通分量

# 提纲

问题定义

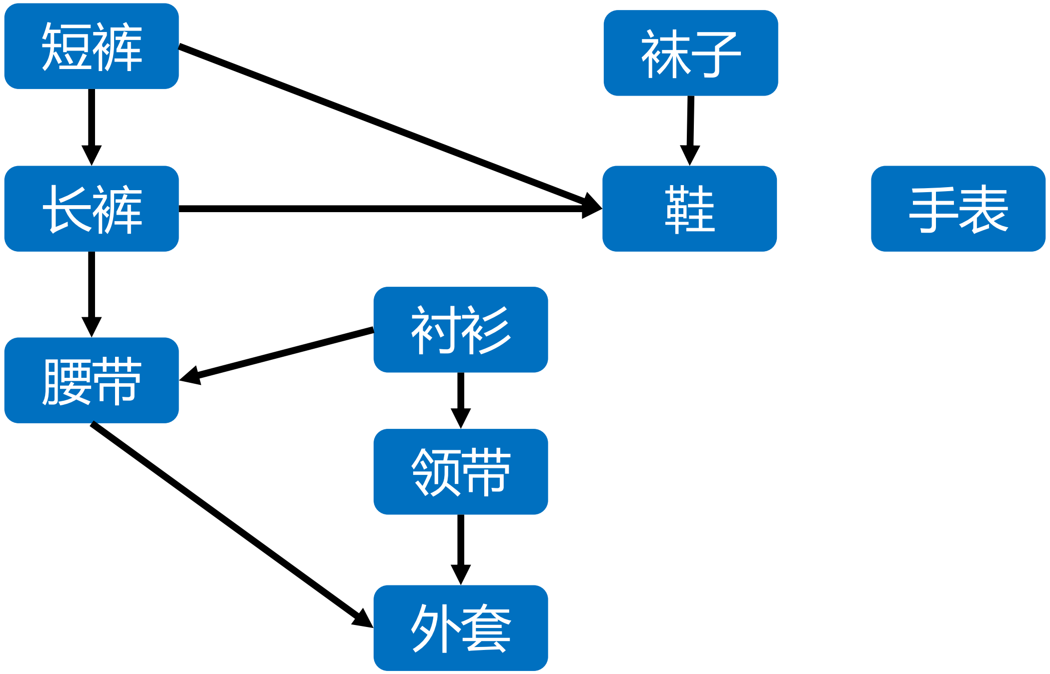
广度优先策略

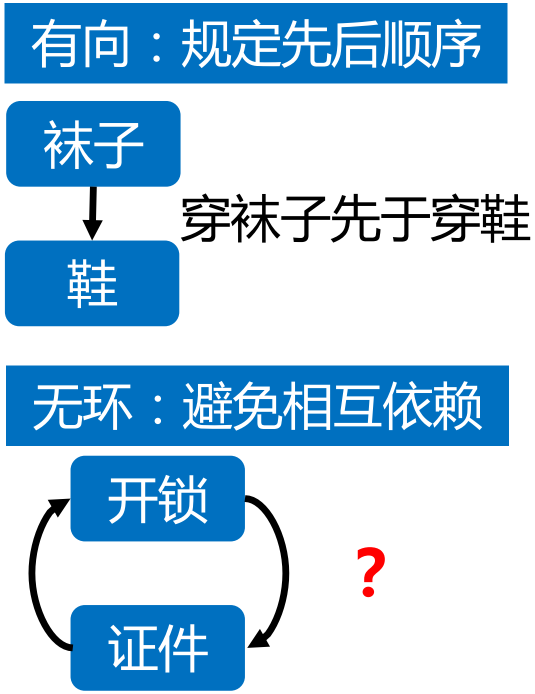
深度优先策略

算法分析

# 问题背景

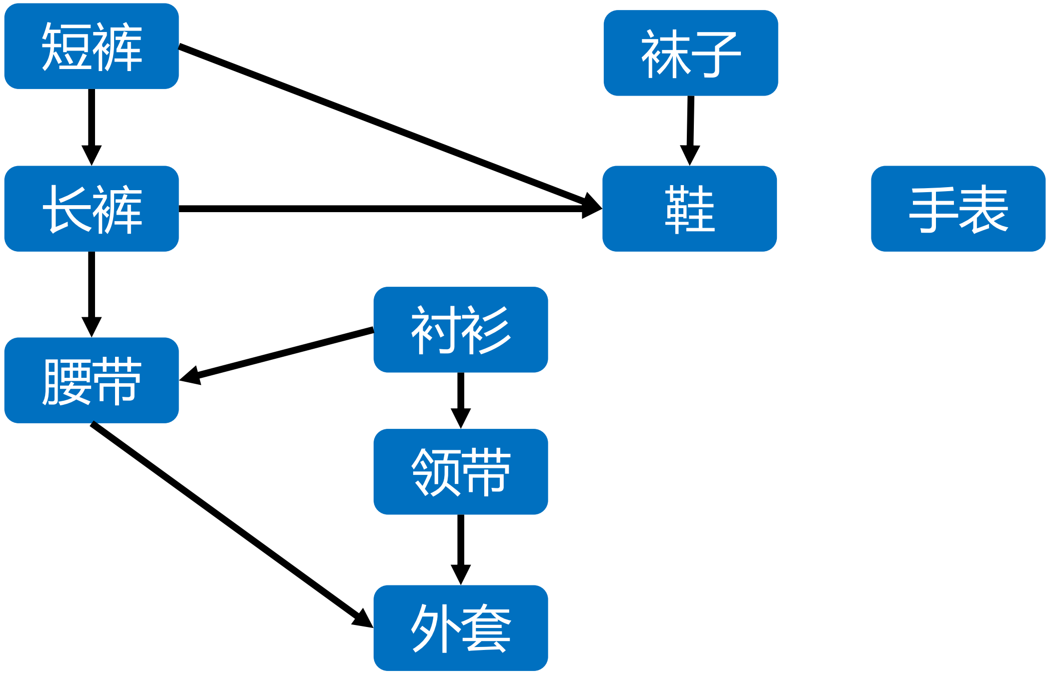
- 穿衣步骤
  - 有向无环图（**Directed Acyclic Graph，DAG**）：表示事件发生的先后顺序



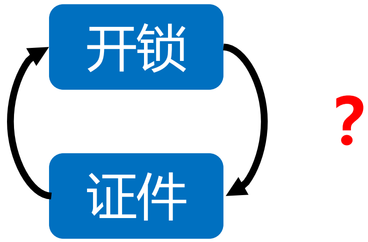
# 问题背景

- 穿衣步骤
  - 有向无环图（**Directed Acyclic Graph，DAG**）：表示事件发生的先后顺序




有向：规定先后顺序

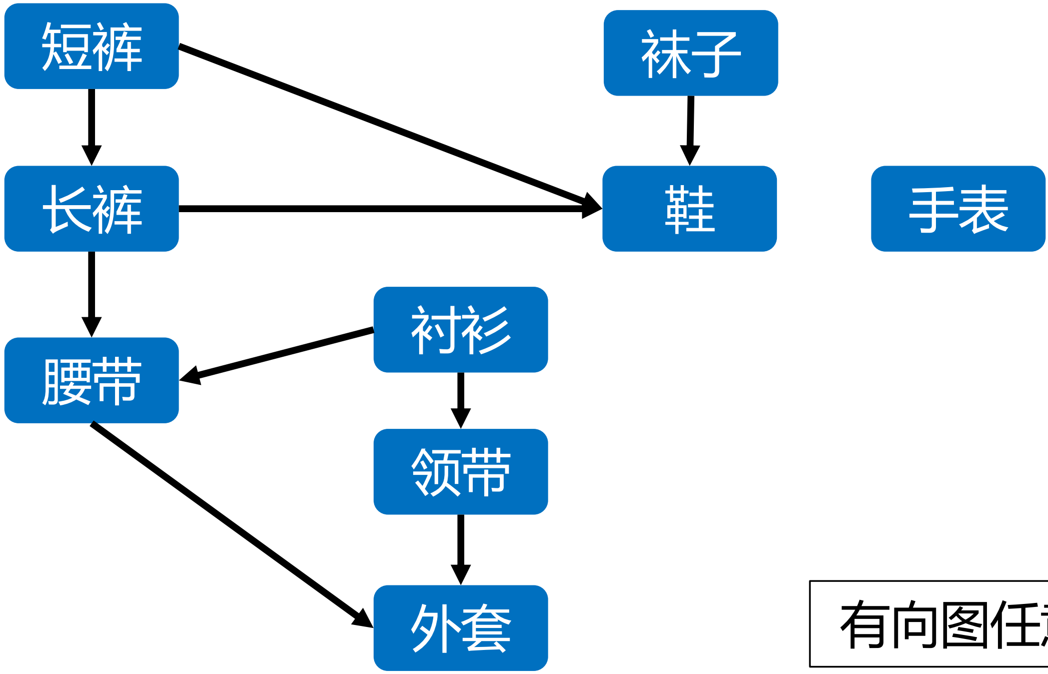

无环：避免相互依赖




问题：如何确定一个可行的穿衣顺序？

# 问题背景

- 穿衣步骤
  - 有向无环图（**Directed Acyclic Graph，DAG**）：表示事件发生的先后顺序




有向图任意边的起点排在终点前面

# 问题定义

## 拓扑排序

**Topological Sort**

输入

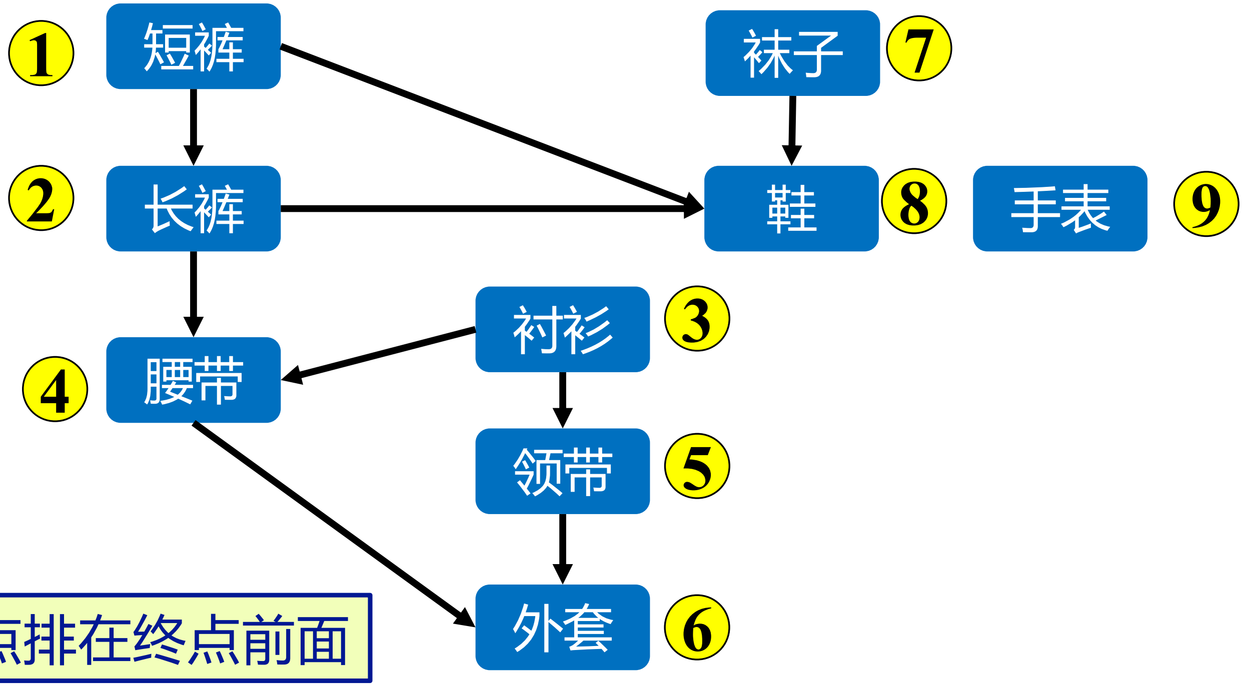
- 有向无环图 $G = < V, E >$

输出

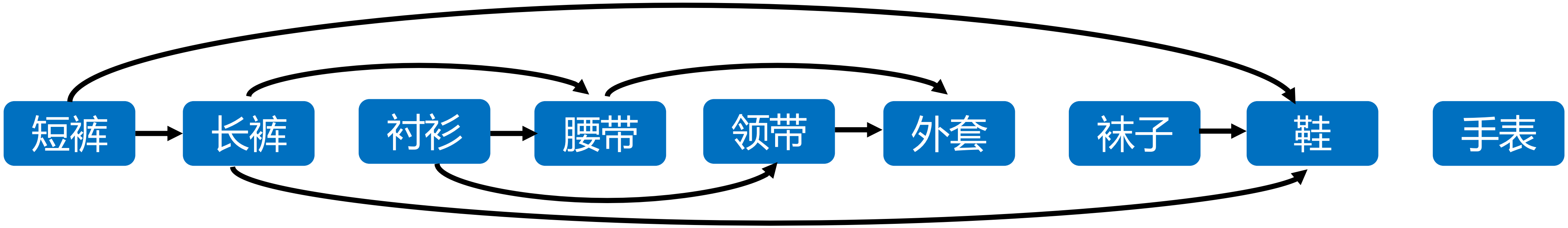
- 图顶点 $V$ 的拓扑序$S$，满足：

  对任意有向边$(u, v)$，排序后 $u$ 在 $v$ 之前

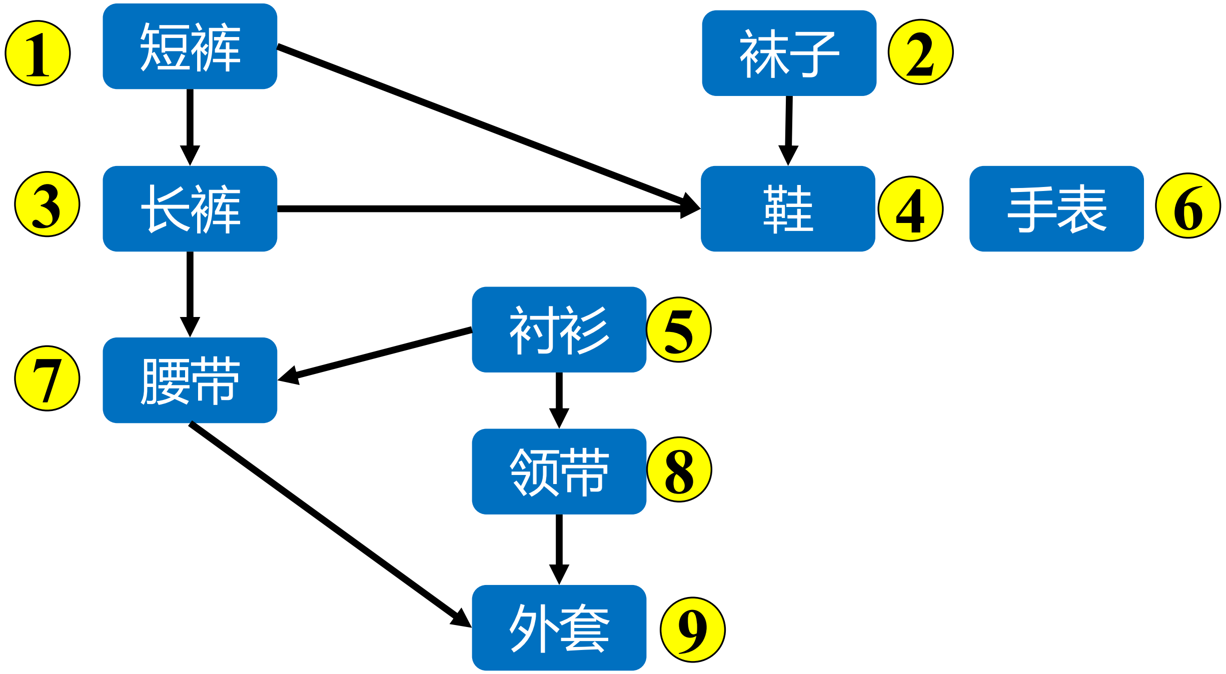
# 拓扑序举例

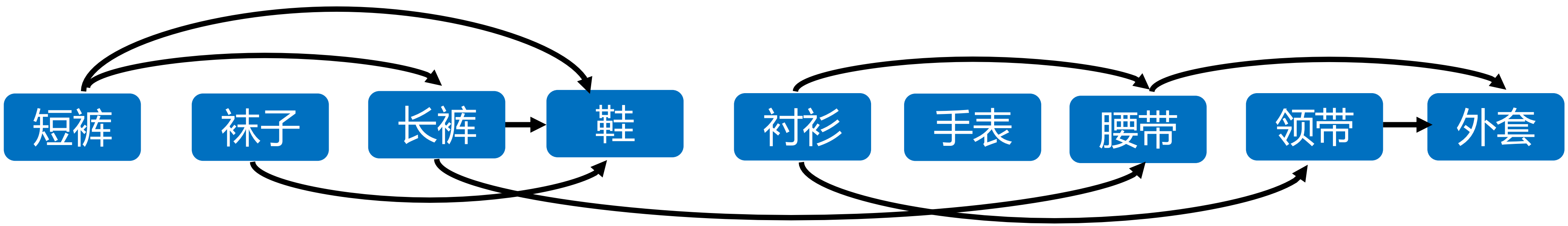



有向图任意边的起点排在终点前面



# 拓扑序不唯一



# 提纲

问题定义

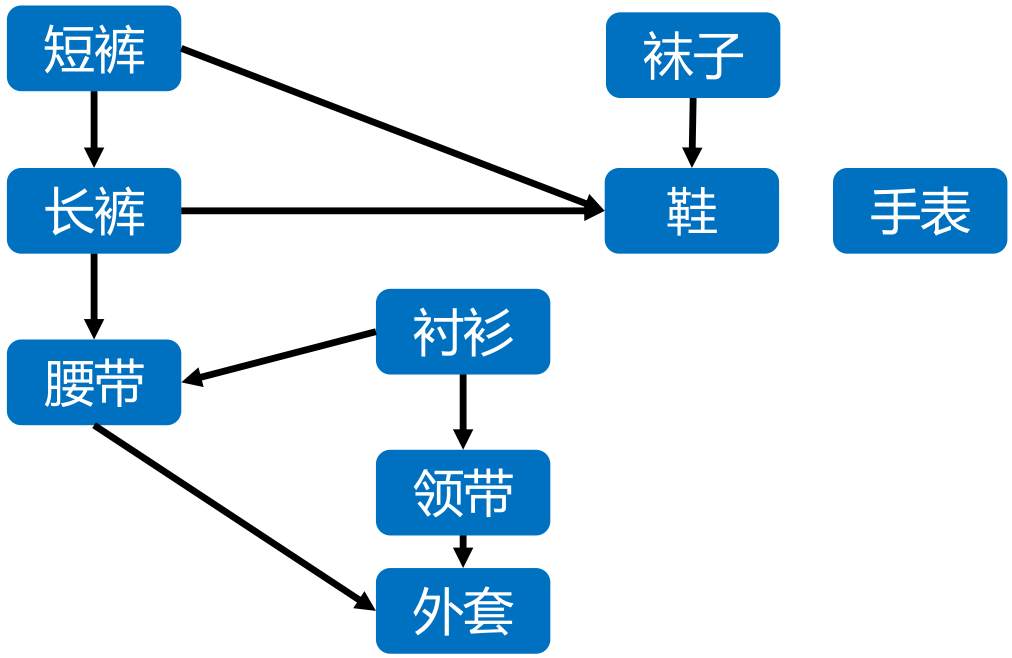
广度优先策略

深度优先策略

算法分析

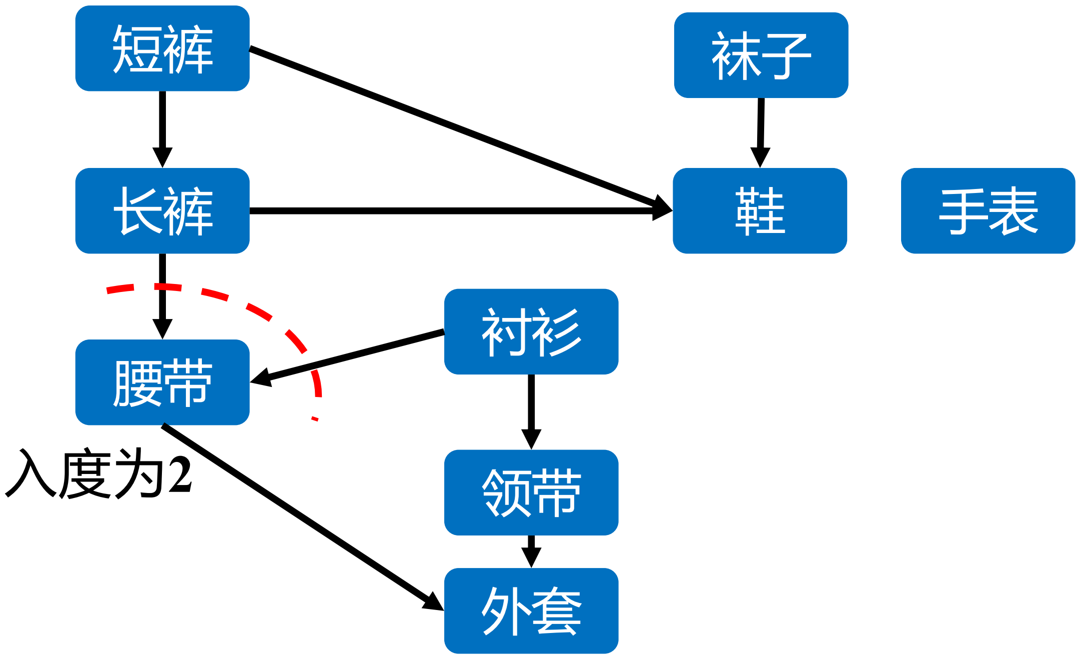
# 算法思想

- 有向图顶点的度分为入度和出度



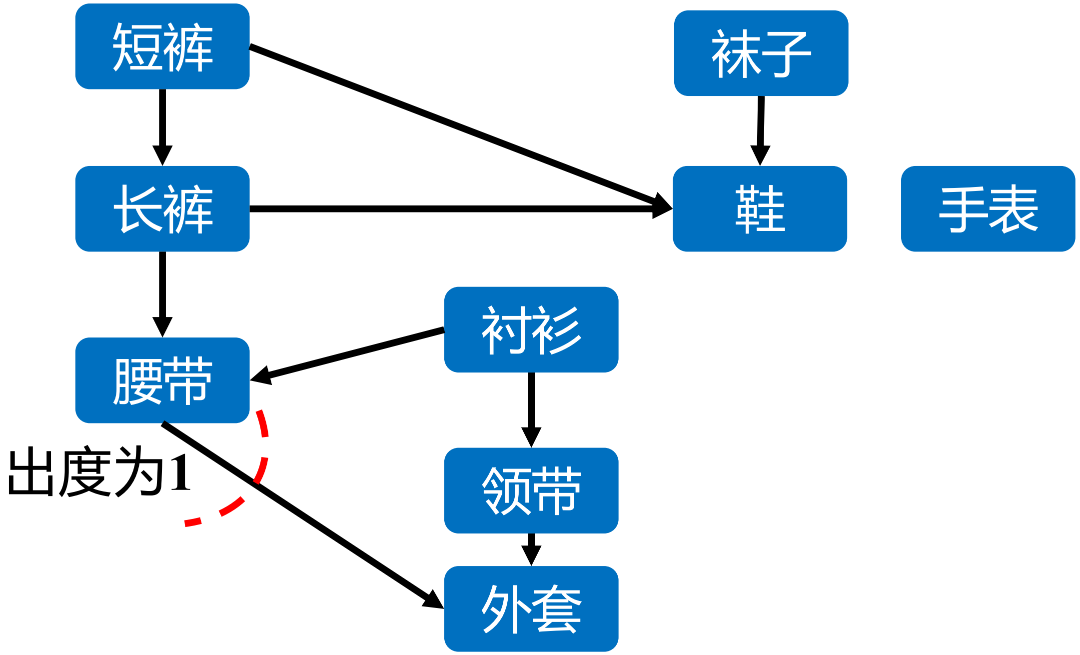
# 算法思想

- 有向图顶点的度分为入度和出度
  - 顶点 $u$ 的入度：终点为 $u$ 的边数



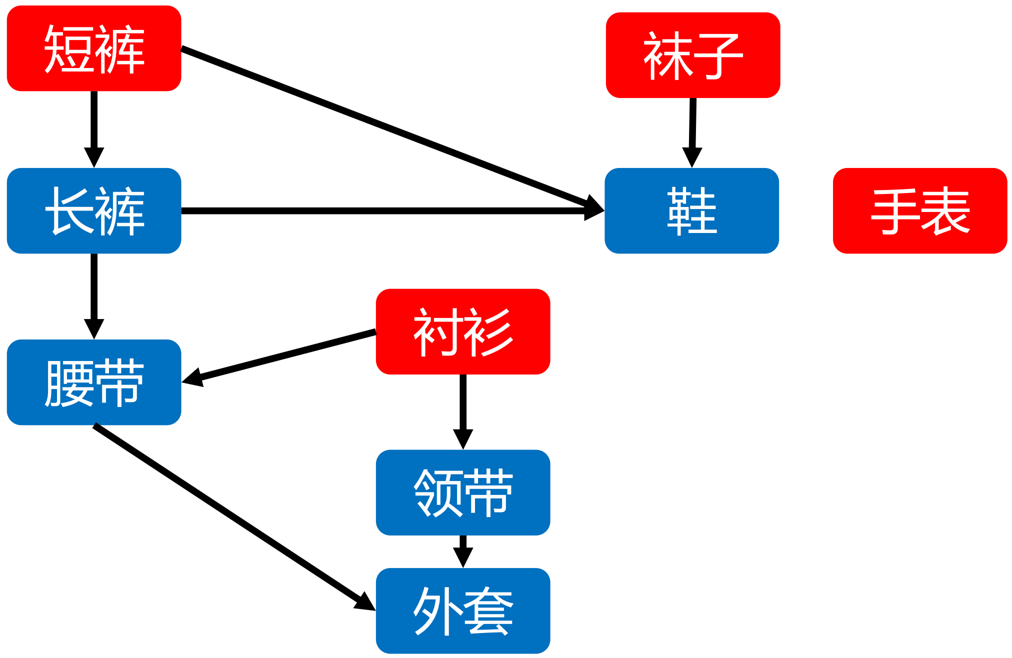
# 算法思想

- 有向图顶点的度分为入度和出度
  - 顶点 $u$ 的入度：终点为 $u$ 的边数
  - 顶点 $u$ 的出度：起点为 $u$ 的边数



# 算法思想

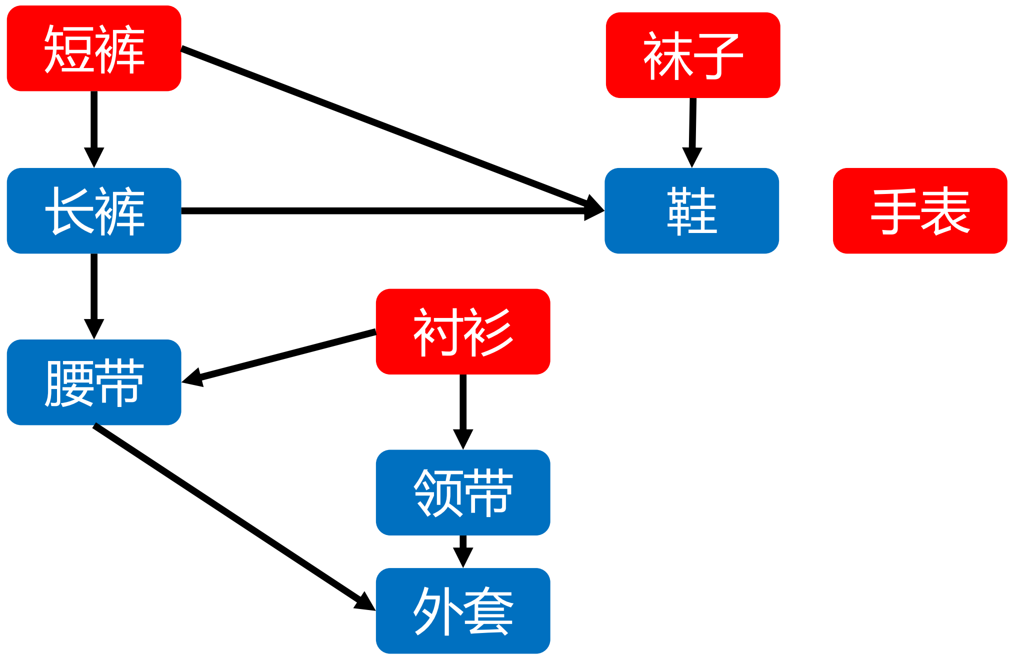
- 有向图顶点的度分为入度和出度
  - 顶点 $u$ 的入度：终点为 $u$ 的边数
  - 顶点 $u$ 的出度：起点为 $u$ 的边数




- 若顶点入度为0
  - 所对应事件无制约，可直接完成

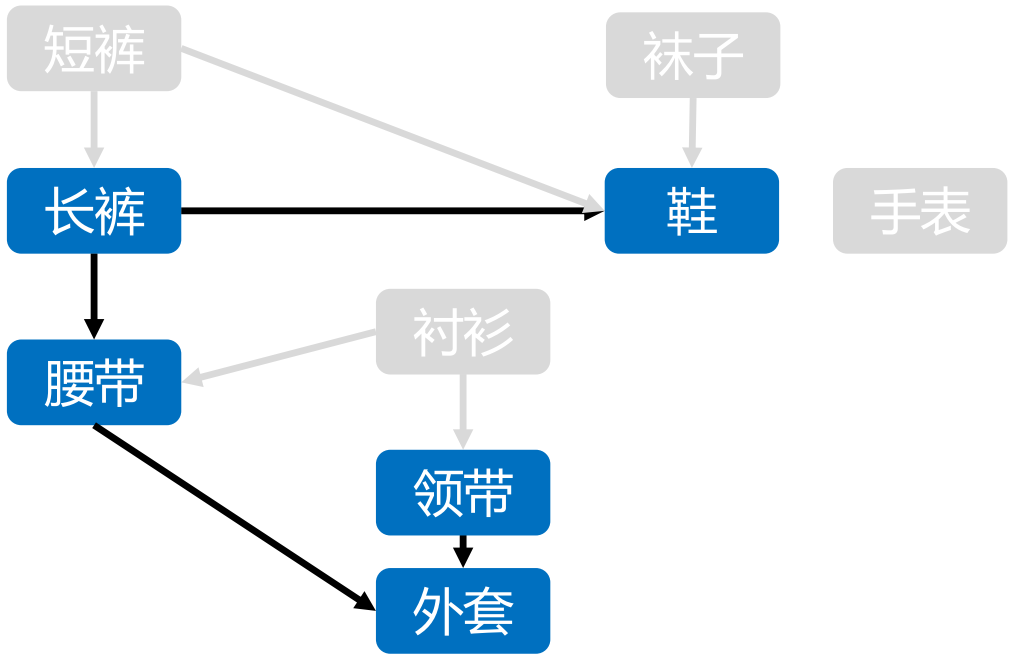
# 算法思想

- 完成入度为0的点对应的事件



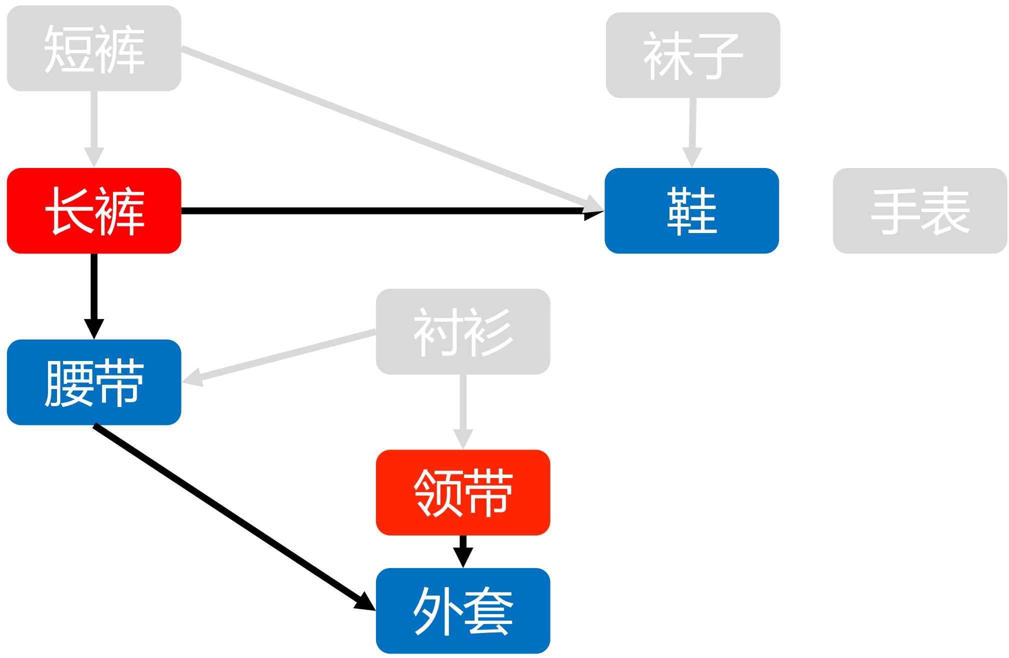
# 算法思想

- 完成入度为0的点对应的事件
- 删除完成事件



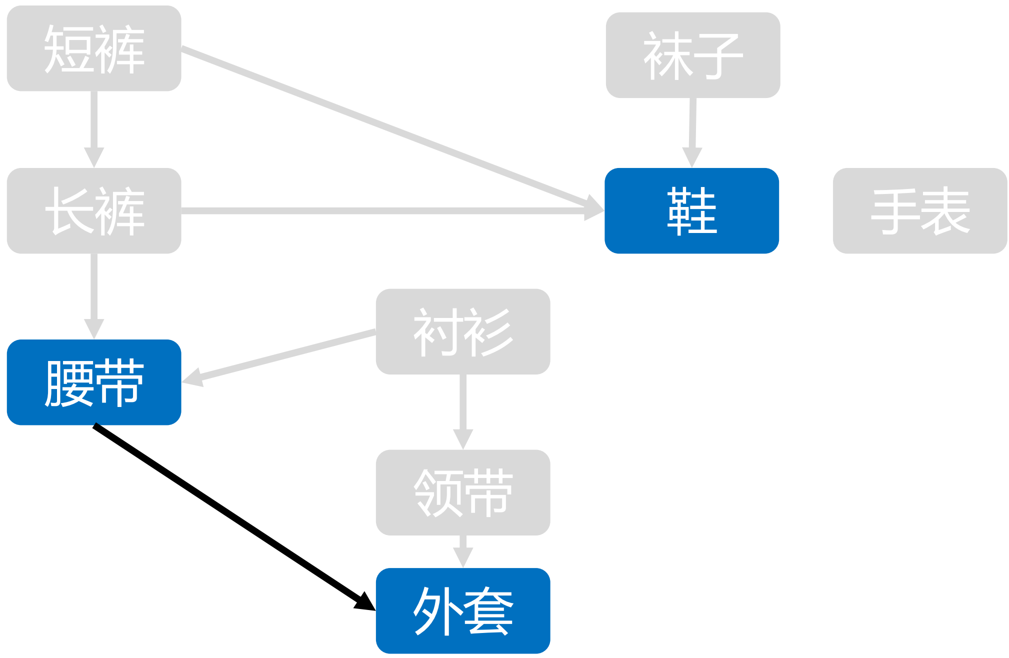
# 算法思想

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成



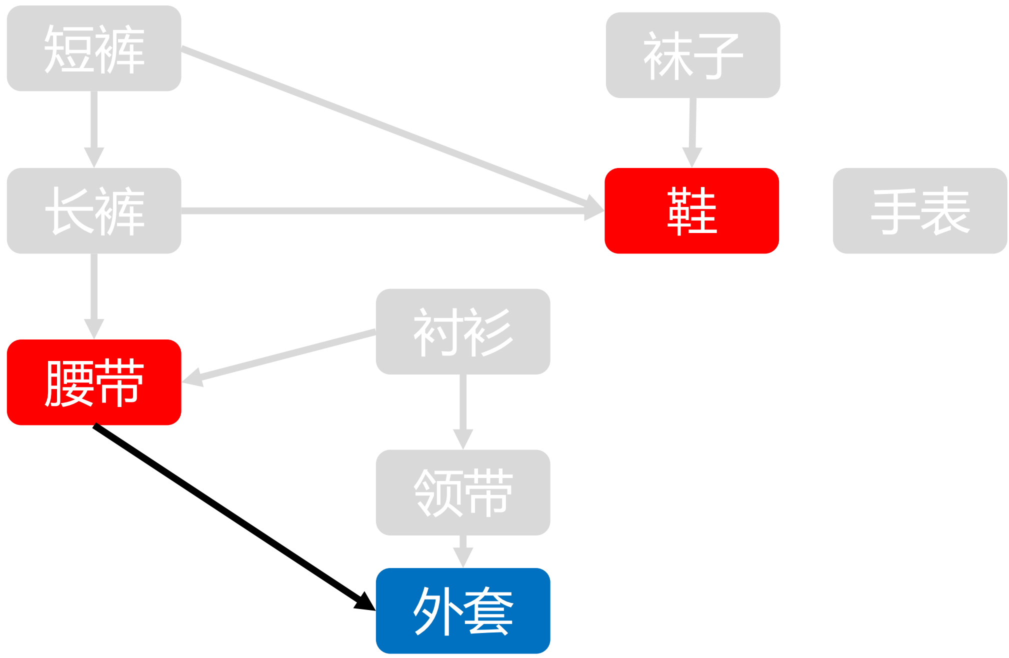
# 算法思想

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成



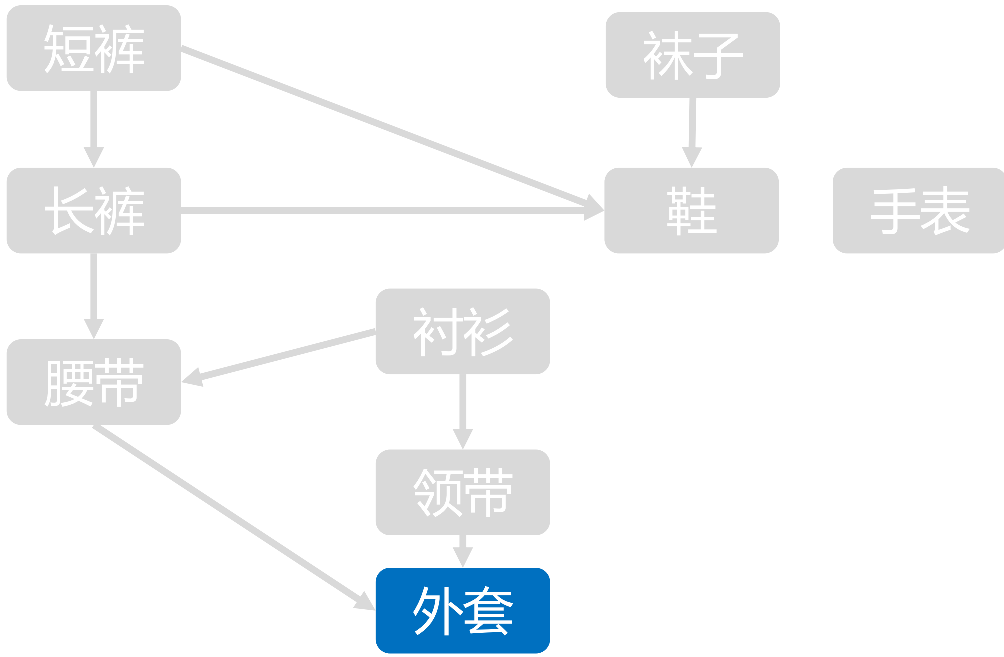
# 算法思想

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成



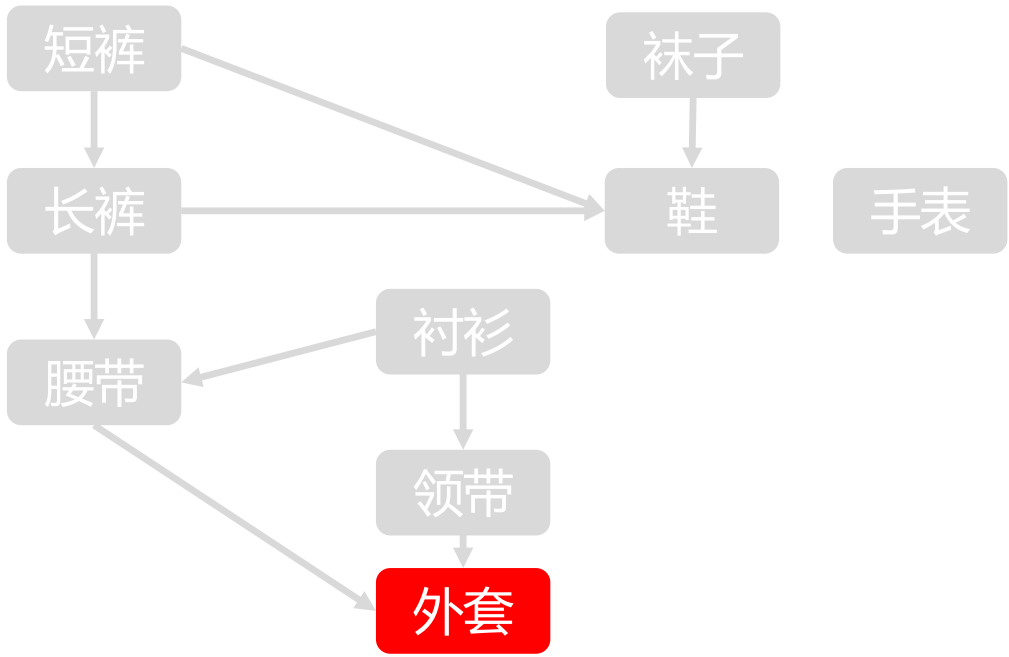
# 算法思想

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成



## 算法思想

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成

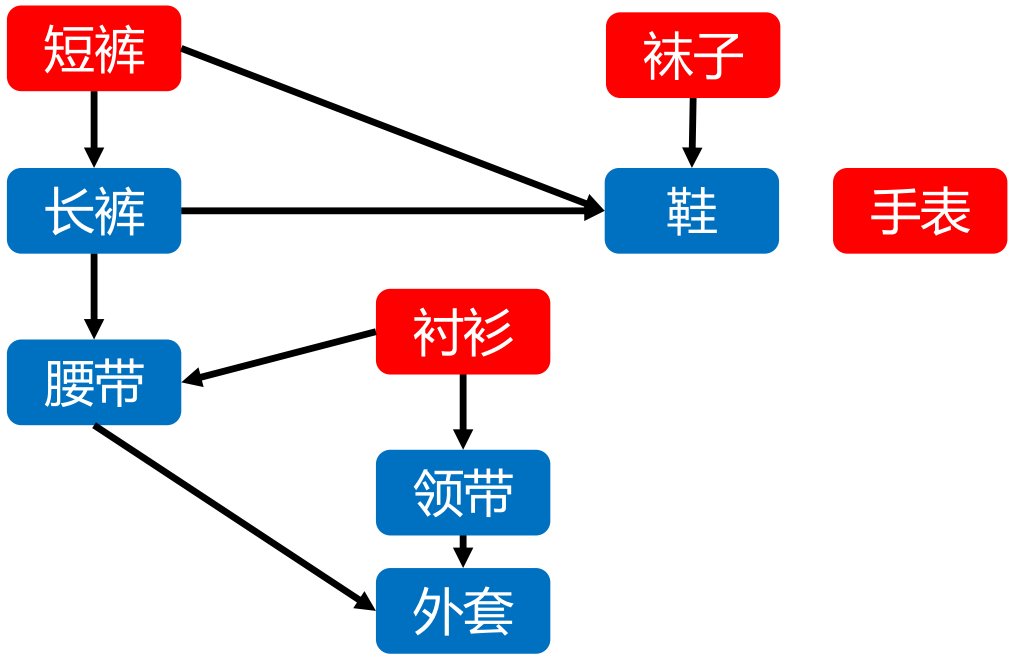


## 算法思想

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成

# 算法实例

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成

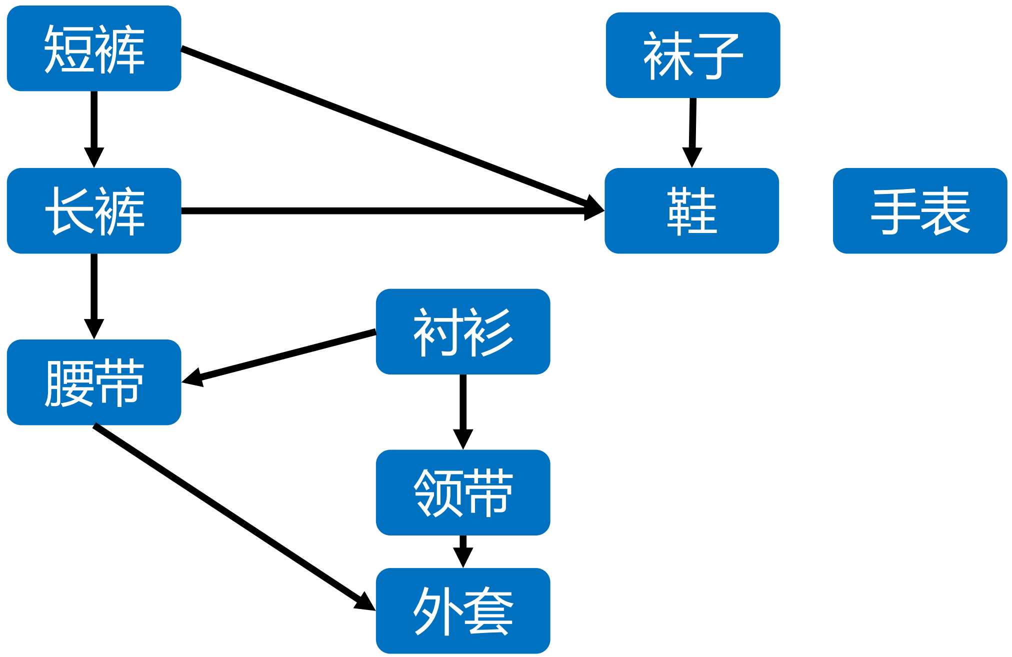



队列： 记录入度为0度点


拓扑序 记录已完成事件

# 算法实例

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成




队列： 记录入度为0度点

短裤 袜子 手表 衬衫

拓扑序 记录已完成事件

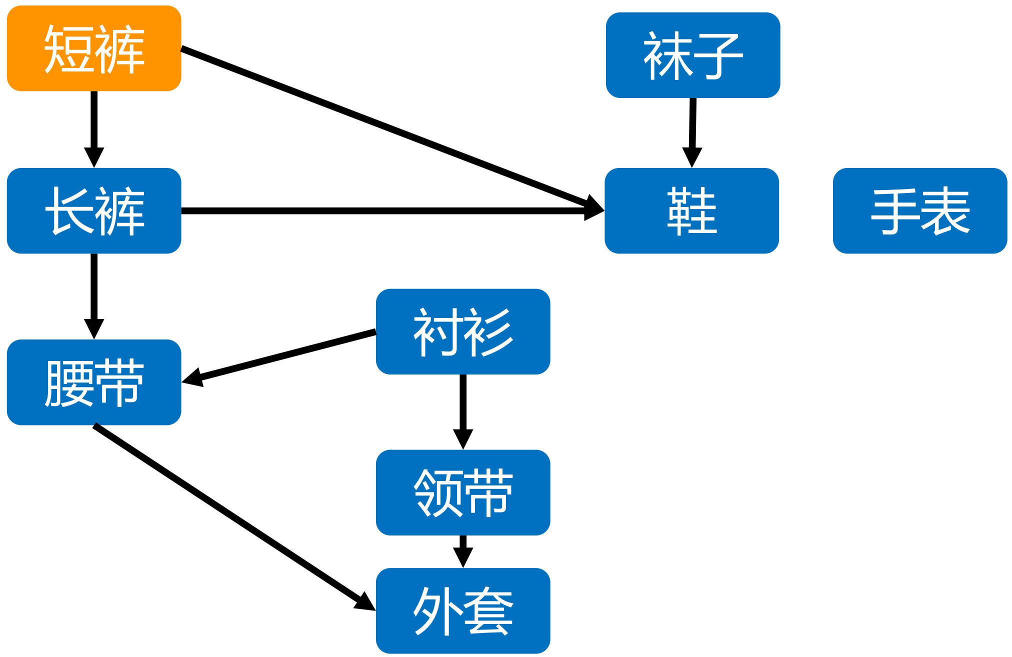
# 算法实例

- 完成**入度为0**的点对应的事件
- 删除完成事件，产生**新的入度为0**点，继续完成

队列：记录入度为0度点

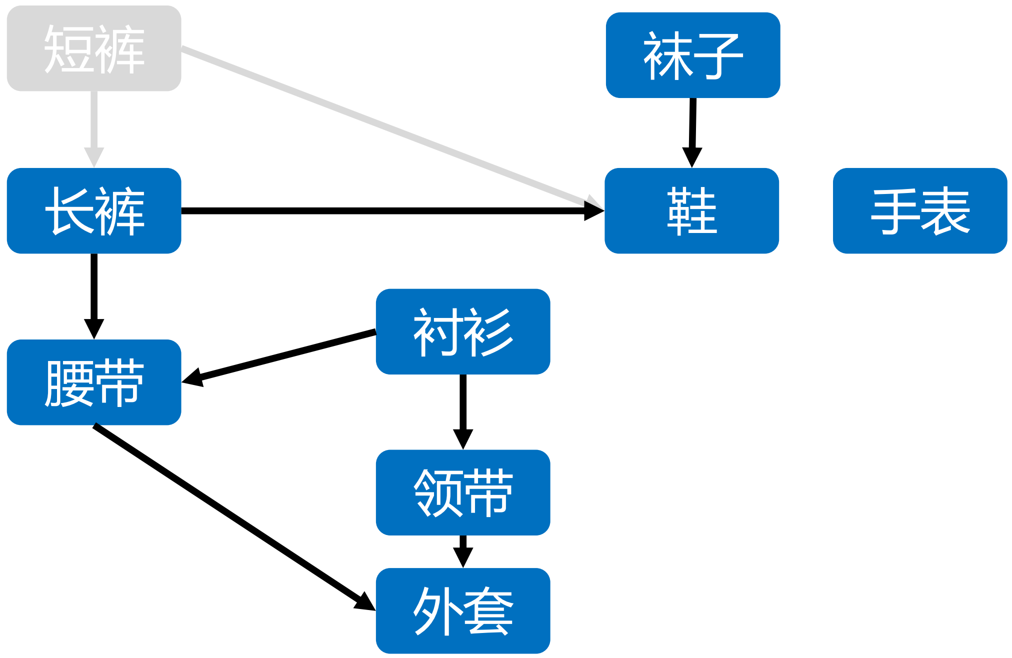
| 短裤 | 袜子 | 手表 | 衬衫 |
|---|---|---|---|

拓扑序　记录已完成事件



# 算法实例

- 完成**入度为0**的点对应的事件
- 删除完成事件，产生**新的入度为0**点，继续完成




队列：记录入度为0度点

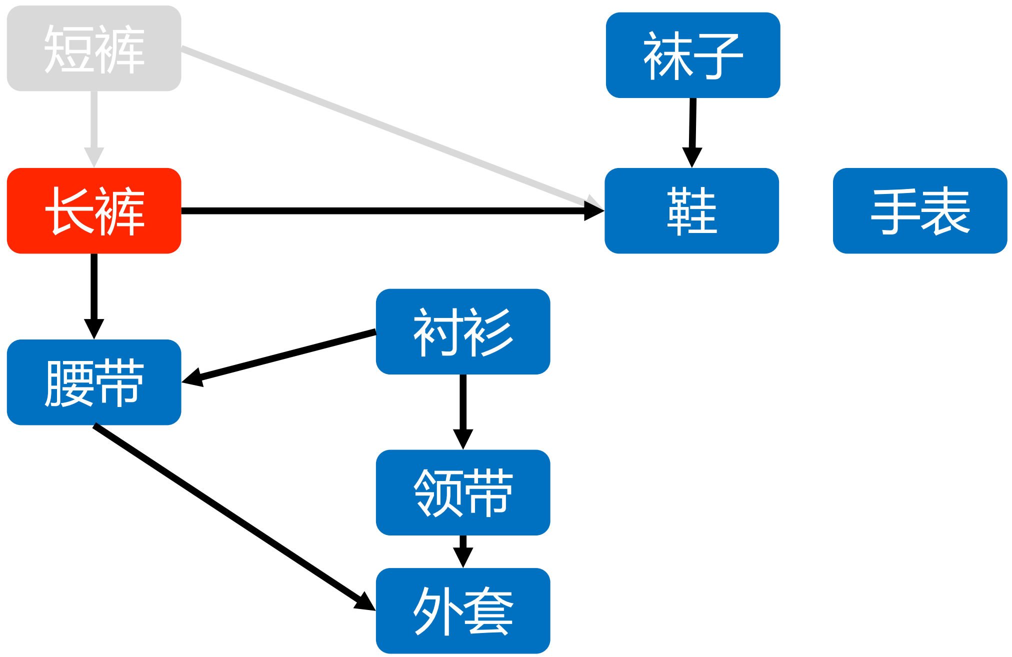
袜子 手表 衬衫

拓扑序 记录已完成事件

短裤

# 算法实例

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成




队列： 记录入度为0度点

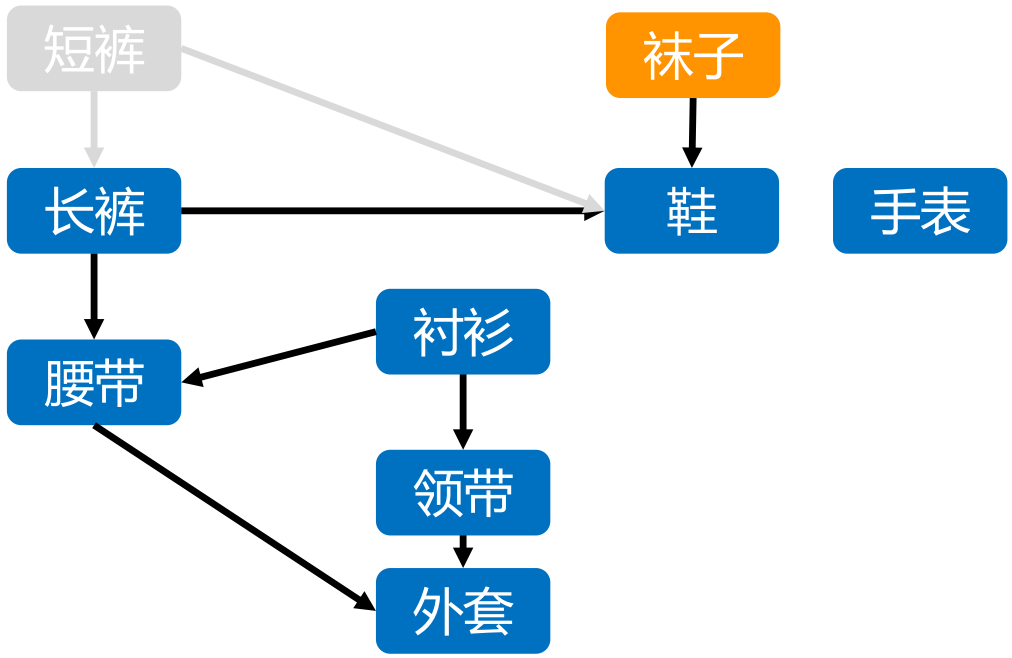
袜子 手表 衬衫 长裤

拓扑序 记录已完成事件

短裤

# 算法实例

- 完成**入度为0**的点对应的事件
- 删除完成事件，产生**新的入度为0**点，继续完成




队列：记录入度为0度点

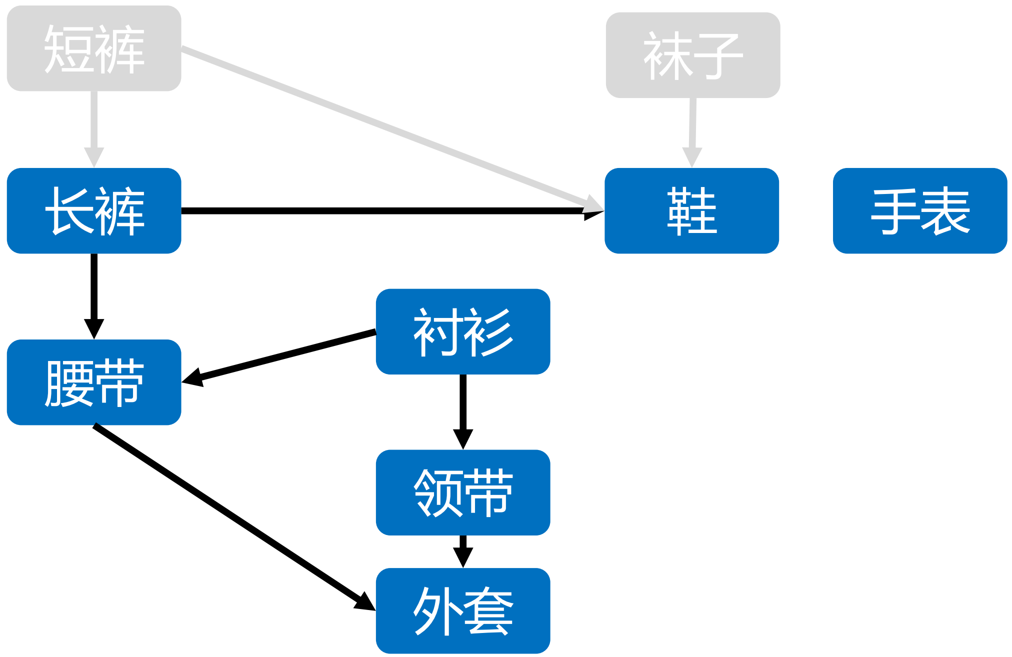
袜子 手表 衬衫 长裤

拓扑序 记录已完成事件

短裤

# 算法实例

- 完成**入度为0**的点对应的事件
- 删除完成事件，产生**新的入度为0**点，继续完成




队列： 记录入度为0度点

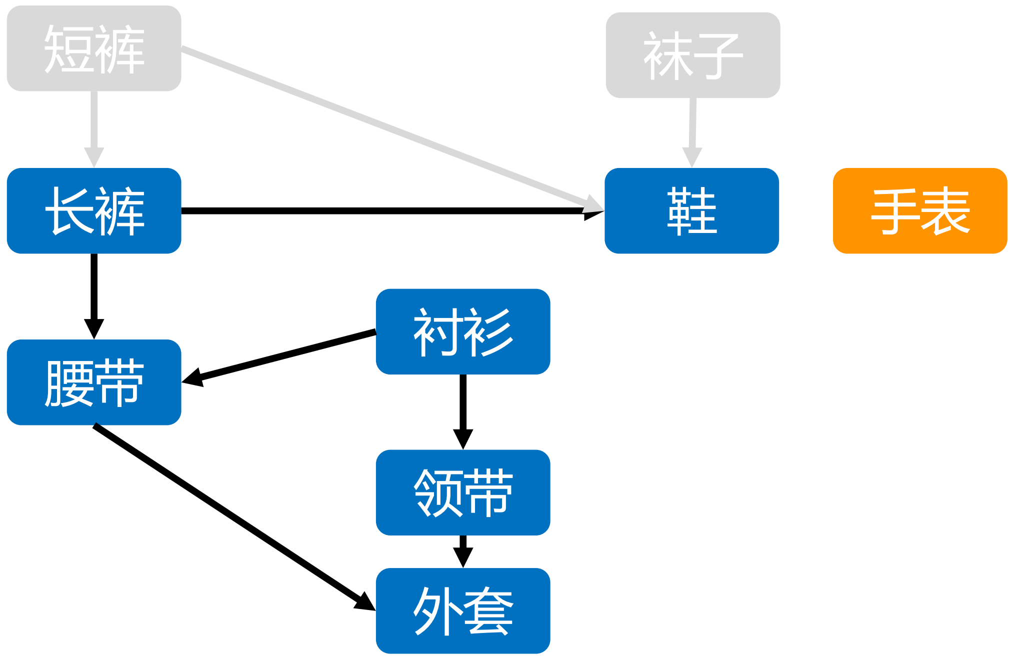
手表 衬衫 长裤

拓扑序 记录已完成事件

短裤 袜子

# 算法实例

- 完成**入度为0**的点对应的事件
- 删除完成事件，产生**新的入度为0**点，继续完成




队列：记录入度为0度点

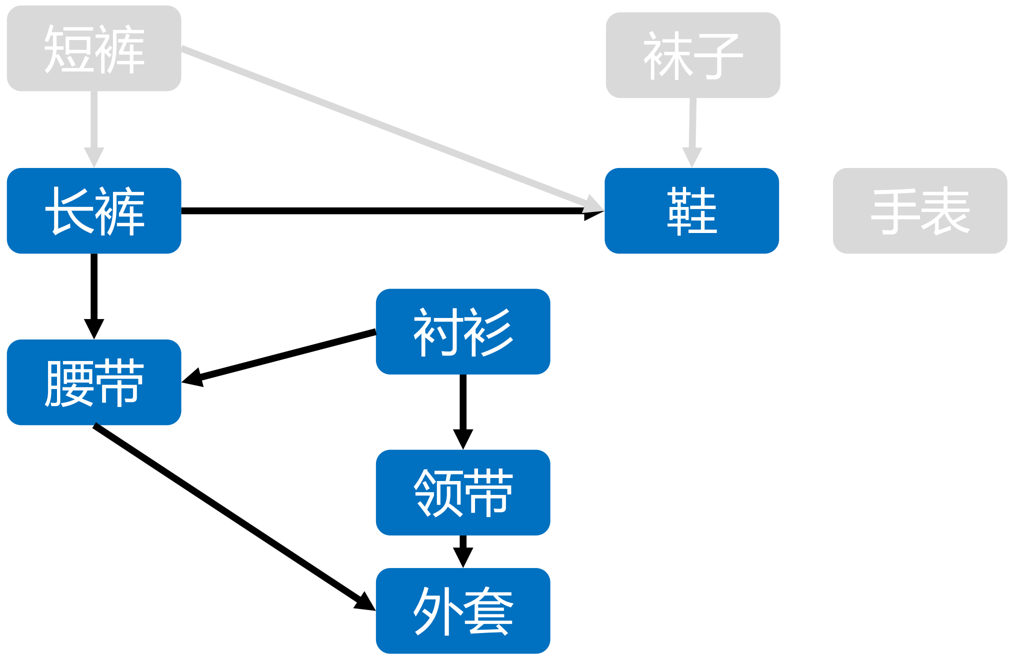
手表 衬衫 长裤

拓扑序 记录已完成事件

短裤 袜子

# 算法实例

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成




队列： 记录入度为0度点

衬衫 长裤

拓扑序 记录已完成事件

短裤 袜子 手表

# 算法实例

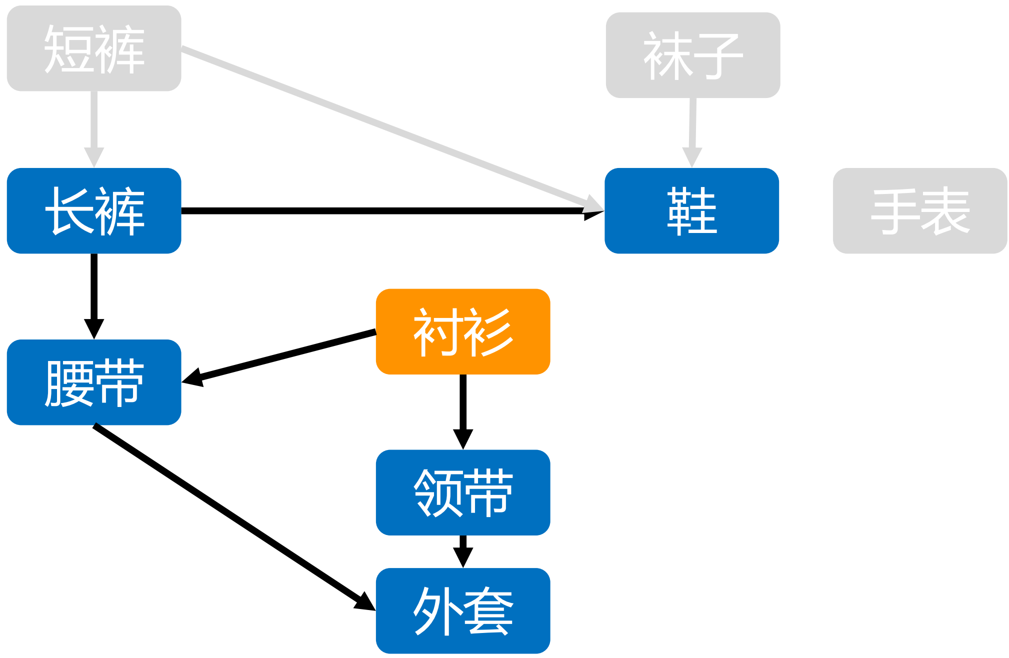
- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成

队列：记录入度为0度点

衬衫 长裤

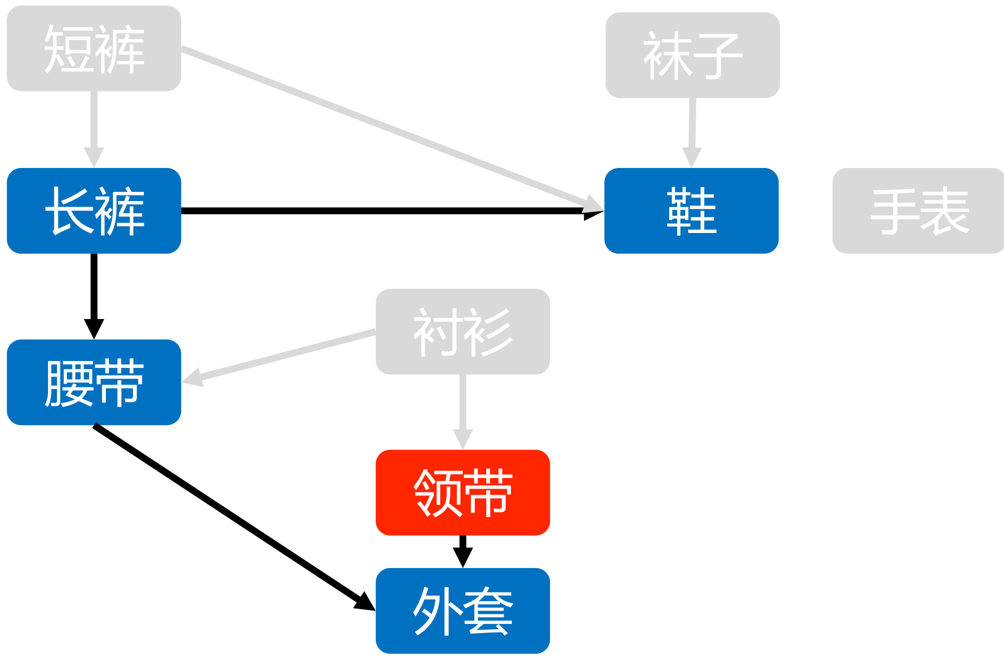
拓扑序 记录已完成事件

短裤 袜子 手表



# 算法实例

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成




队列： 记录入度为0度点

长裤 领带

拓扑序 记录已完成事件

短裤 袜子 手表 衬衫

# 算法实例

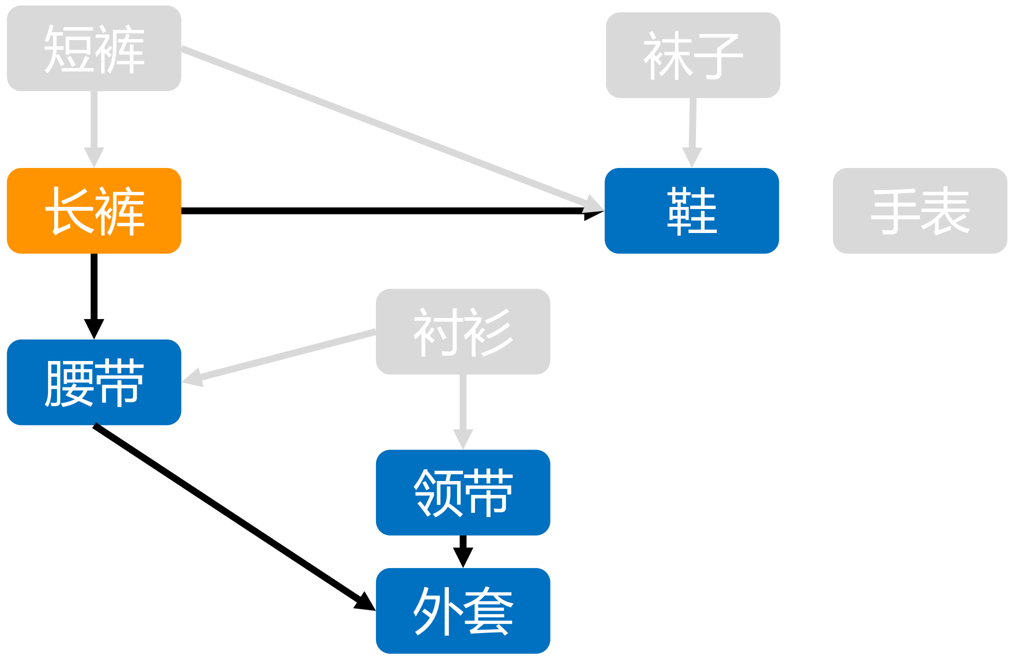
- 完成**入度为0**的点对应的事件
- 删除完成事件，产生**新的入度为0**点，继续完成

队列：记录入度为0度点

长裤 领带

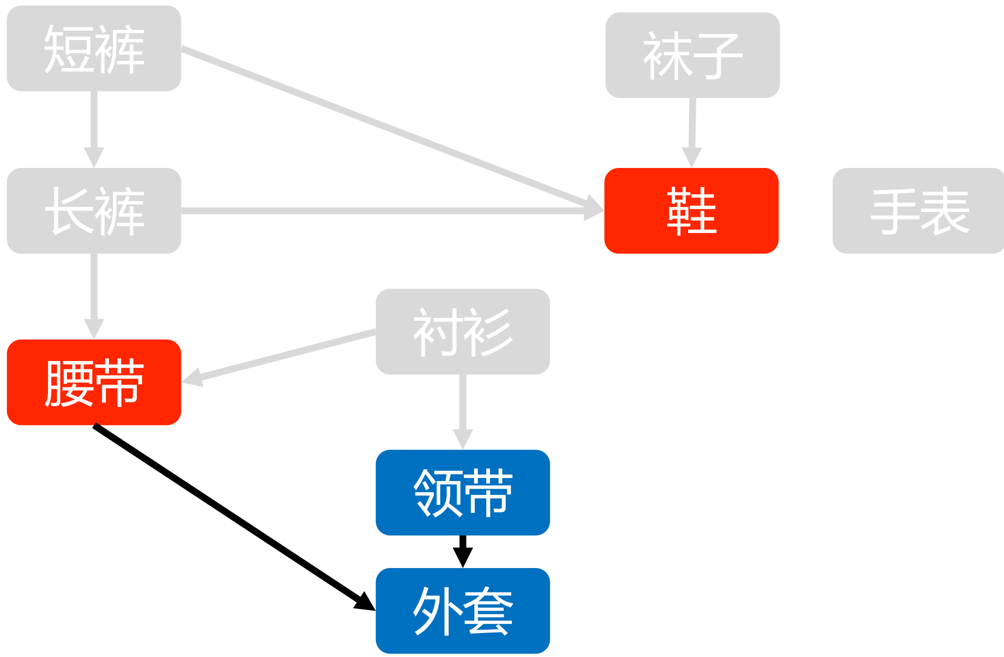
拓扑序 记录已完成事件

短裤 袜子 手表 衬衫



# 算法实例

- 完成**入度为0**的点对应的事件
- 删除完成事件，产生**新的入度为0**点，继续完成




队列： 记录入度为0度点

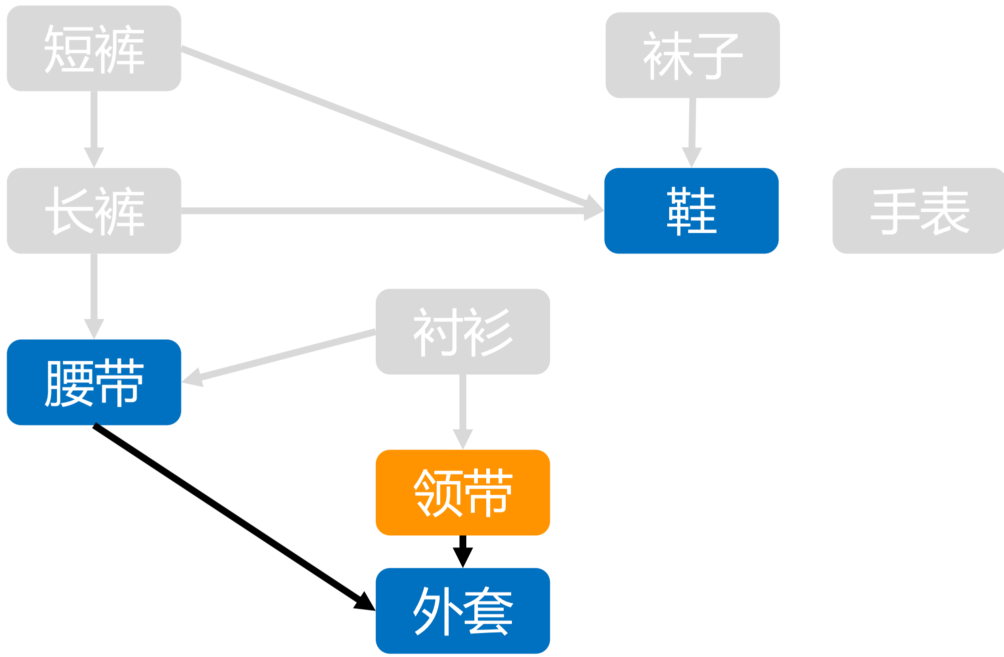
领带 鞋 腰带

拓扑序 记录已完成事件

短裤 袜子 手表 衬衫 长裤

## 算法实例

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成




队列：记录入度为0度点

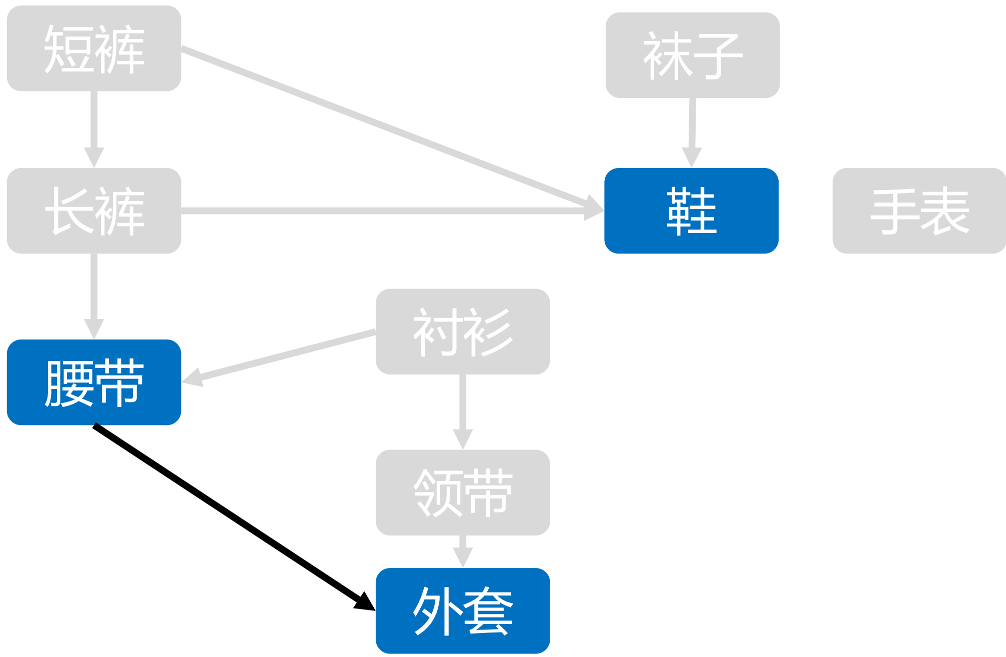
领带 | 鞋 | 腰带

拓扑序 记录已完成事件

短裤 | 袜子 | 手表 | 衬衫 | 长裤

# 算法实例

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成




队列： 记录入度为0度点

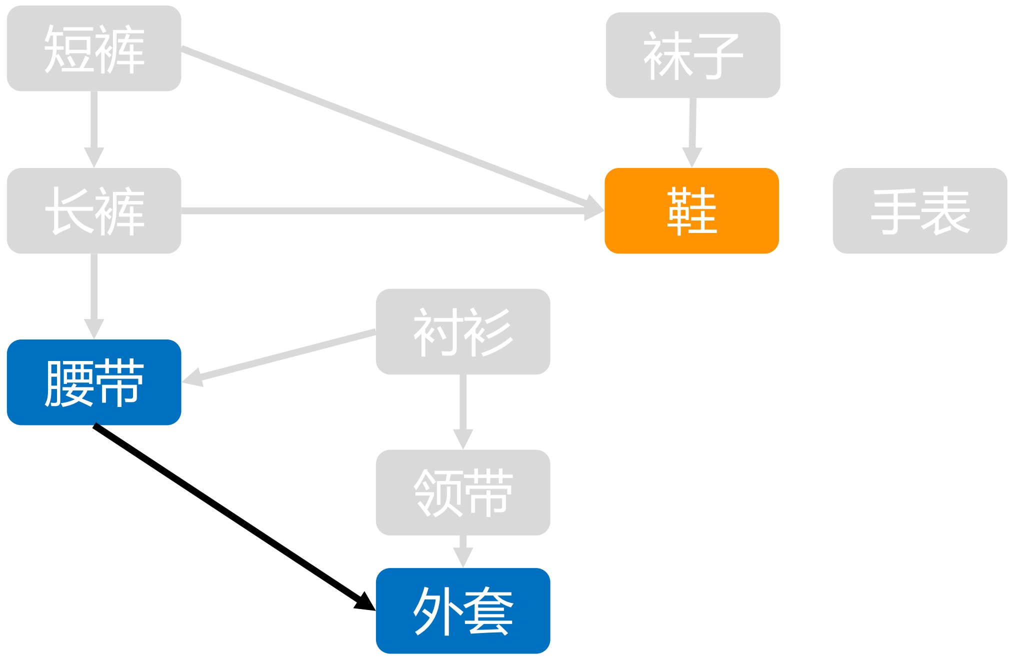
鞋 腰带

拓扑序 记录已完成事件

短裤 袜子 手表 衬衫 长裤 领带

# 算法实例

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成




队列： 记录入度为0度点

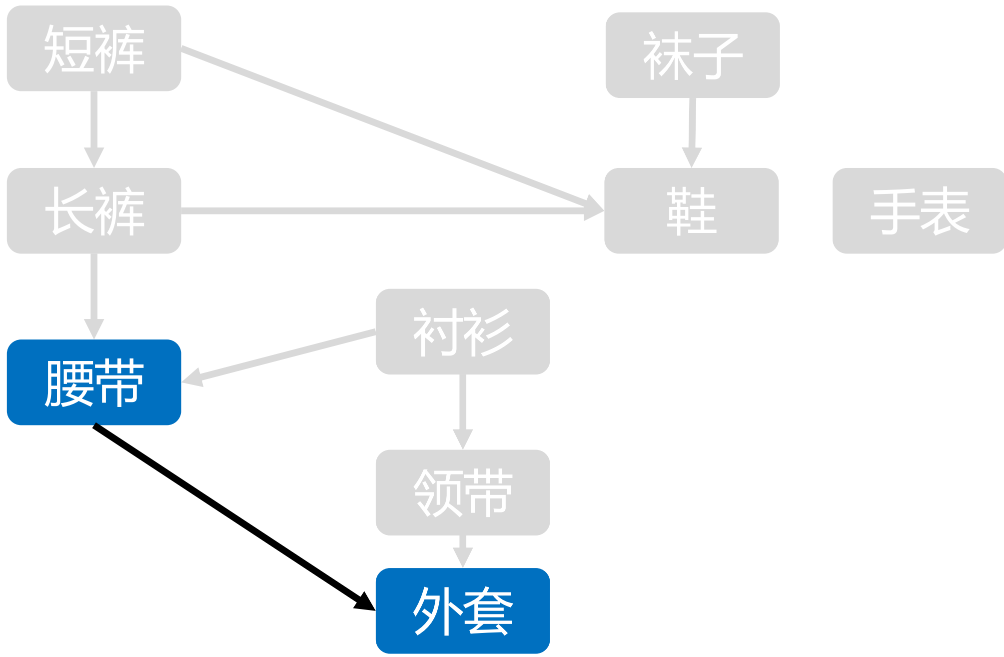
鞋 腰带

拓扑序 记录已完成事件

短裤 袜子 手表 衬衫 长裤 领带

## 算法实例

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成




队列： 记录入度为0度点

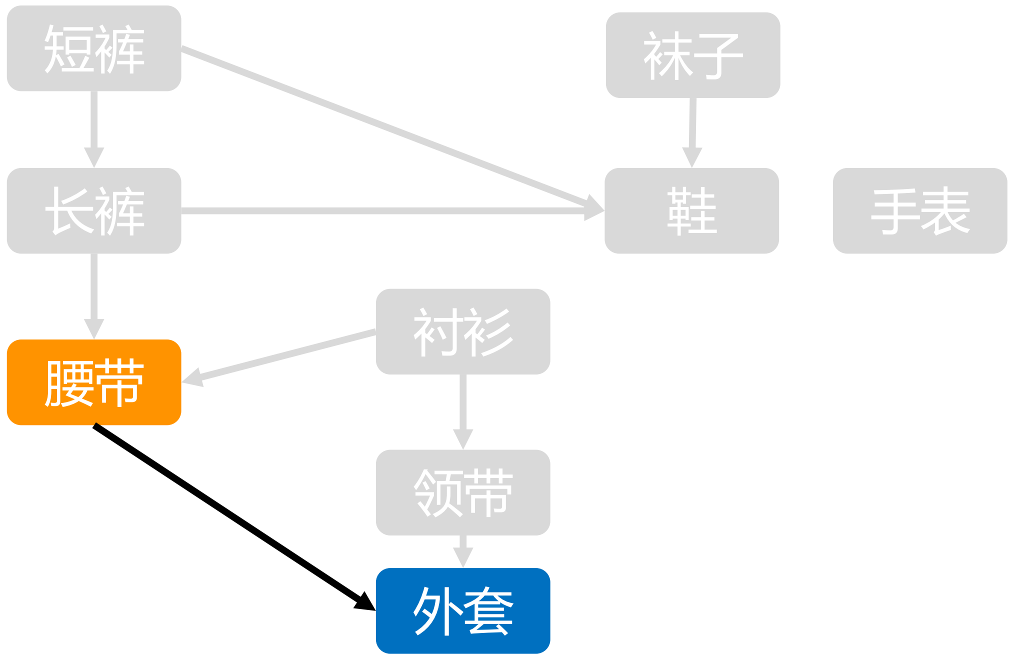
腰带

拓扑序 记录已完成事件

短裤 袜子 手表 衬衫 长裤 领带 鞋

# 算法实例

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成




队列：记录入度为0度点

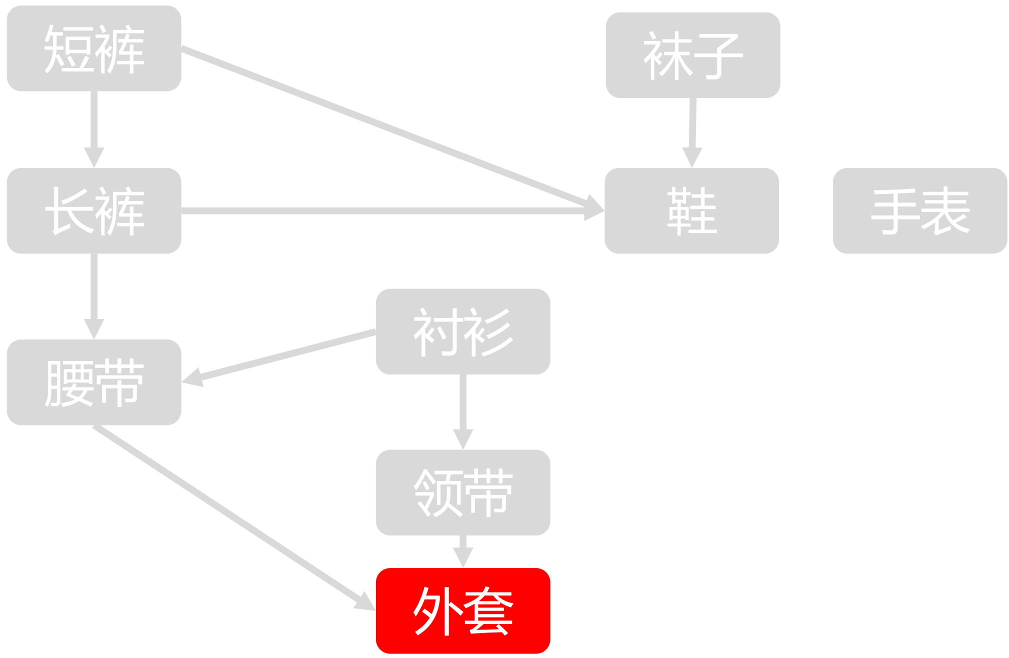
腰带

拓扑序 记录已完成事件

短裤 | 袜子 | 手表 | 衬衫 | 长裤 | 领带 | 鞋

## 算法实例

- 完成入度为0的点对应的事件
- 删除完成事件，产生新的入度为0点，继续完成




队列：记录入度为0度点

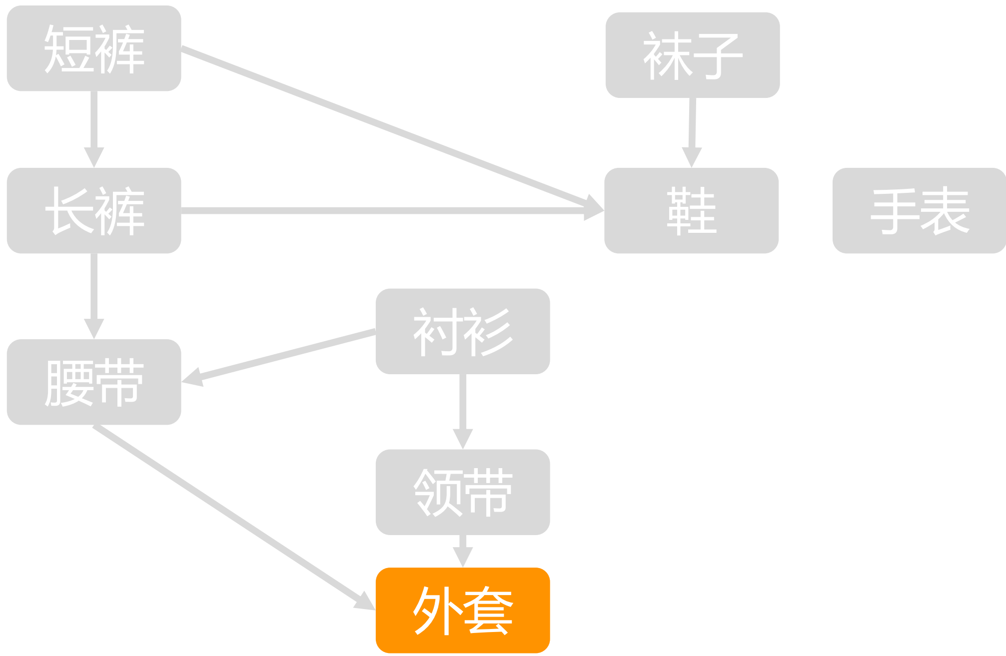
外套

拓扑序 记录已完成事件

短裤 袜子 手表 衬衫 长裤 领带 鞋 腰带

# 算法实例

- 完成**入度为0**的点对应的事件
- 删除完成事件，产生**新的入度为0**点，继续完成




队列：记录入度为0度点

外套

拓扑序 记录已完成事件

短裤 袜子 手表 衬衫 长裤 领带 鞋 腰带

# 算法实例

- 完成**入度为0**的点对应的事件
- 删除完成事件，产生**新的入度为0**点，继续完成




队列： 记录入度为0度点

拓扑序 记录已完成事件

| 短裤 | 袜子 | 手表 | 衬衫 | 长裤 | 领带 | 鞋 | 腰带 | 外套 |
|---|---|---|---|---|---|---|---|---|

# 伪代码

- **Topological-Sort-BFS(*G*)**

**输入:** 图$G$
**输出:** 顶点拓扑序
初始化空队列$Q$
**for** $v \in V$ **do**
  **if** $v.in\_degree = 0$ **then**
    $Q.Enqueue(v)$
  **end**
**end**
**while** $not\ Q.is\_empty()$ **do**
  $u \leftarrow Q.Dequeue()$
  print $u$
  **for** $v \in G.Adj(u)$ **do**
    $v.in\_degree \leftarrow v.in\_degree - 1$
    **if** $v.in\_degree = 0$ **then**
      $Q.Enqueue(v)$
    **end**
  **end**
**end**

初始化

# 伪代码

- **Topological-Sort-BFS(*G*)**

```
输入: 图G
输出: 顶点拓扑序
初始化空队列Q
for v ∈ V do
    if v.in_degree = 0 then
        Q.Enqueue(v)
    end
end
while not Q.is_empty() do
    u ← Q.Dequeue()
    print u
    for v ∈ G.Adj(u) do
        v.in_degree ← v.in_degree − 1
        if v.in_degree = 0 then
            Q.Enqueue(v)
        end
    end
end
```

入度为0，加入队列

# 伪代码

- **Topological-Sort-BFS(*G*)**

```
输入: 图G
输出: 顶点拓扑序
初始化空队列Q
for v ∈ V do
    if v.in_degree = 0 then
        Q.Enqueue(v)
    end
end
while not Q.is_empty() do
    u ← Q.Dequeue()
    print u
    for v ∈ G.Adj(u) do
        v.in_degree ← v.in_degree − 1
        if v.in_degree = 0 then
            Q.Enqueue(v)
        end
    end
end
```

完成事件

# 伪代码

- **Topological-Sort-BFS(*G*)**

```
输入: 图G
输出: 顶点拓扑序
初始化空队列Q
for v ∈ V do
    if v.in_degree = 0 then
        Q.Enqueue(v)
    end
end
while not Q.is_empty() do
    u ← Q.Dequeue()
    print u
    for v ∈ G.Adj(u) do
        v.in_degree ← v.in_degree − 1
        if v.in_degree = 0 then
            Q.Enqueue(v)
        end
    end
end
```

删除事件，更新入度

# 伪代码

- **Topological-Sort-BFS(*G*)**

```
输入: 图G
输出: 顶点拓扑序
初始化空队列Q
for v ∈ V do
    if v.in_degree = 0 then
        Q.Enqueue(v)
    end
end
while not Q.is_empty() do
    u ← Q.Dequeue()
    print u
    for v ∈ G.Adj(u) do
        v.in_degree ← v.in_degree − 1
        if v.in_degree = 0 then
            Q.Enqueue(v)
        end
    end
end
```

入度为0，加入队列

# 复杂度分析

- **Topological-Sort-BFS($G$)**

```
输入: 图G
输出: 顶点拓扑序
初始化空队列Q
for v ∈ V do
    if v.in_degree = 0 then
        Q.Enqueue(v)
    end
end
while not Q.is_empty() do
    u ← Q.Dequeue()
    print u
    for v ∈ G.Adj(u) do
        v.in_degree ← v.in_degree − 1
        if v.in_degree = 0 then
            Q.Enqueue(v)
        end
    end
end
```

$O(|V|)$

$\sum_{v\in V} deg(v) = O(|E|)$

$$O\left(\sum_{v\in V}(1 + |Adj[v]|)\right) = O(|V| + |E|)$$

时间复杂度：$O(|V| + |E|)$

# 提纲

问题定义

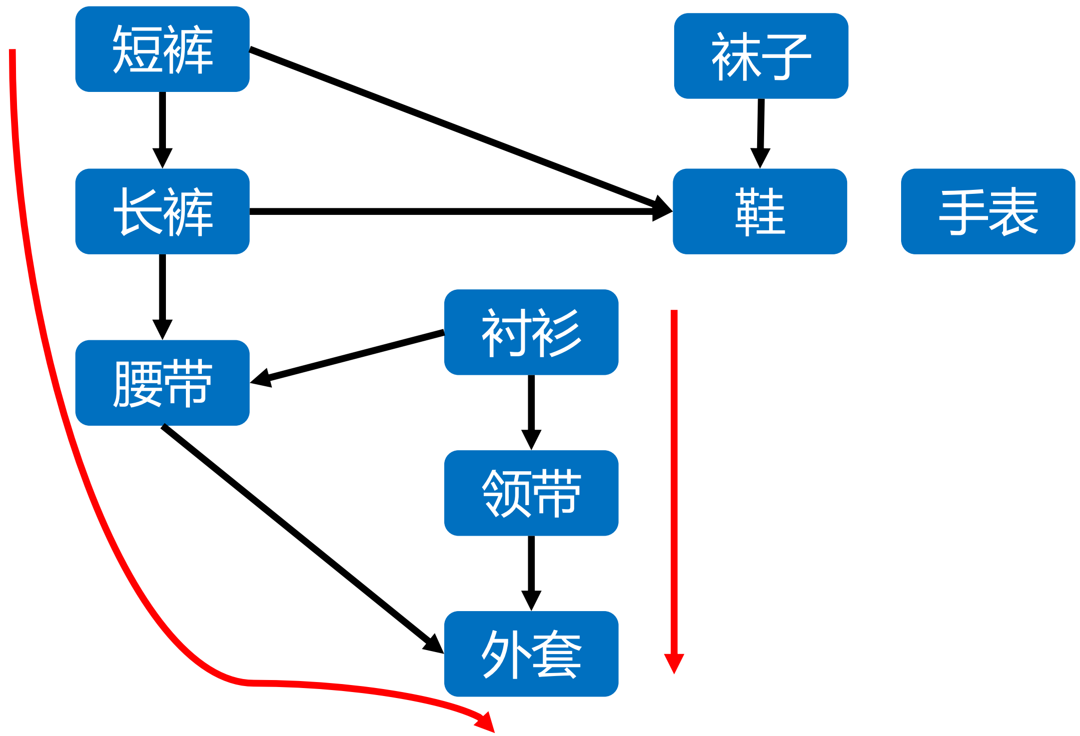
广度优先策略

深度优先策略

算法分析

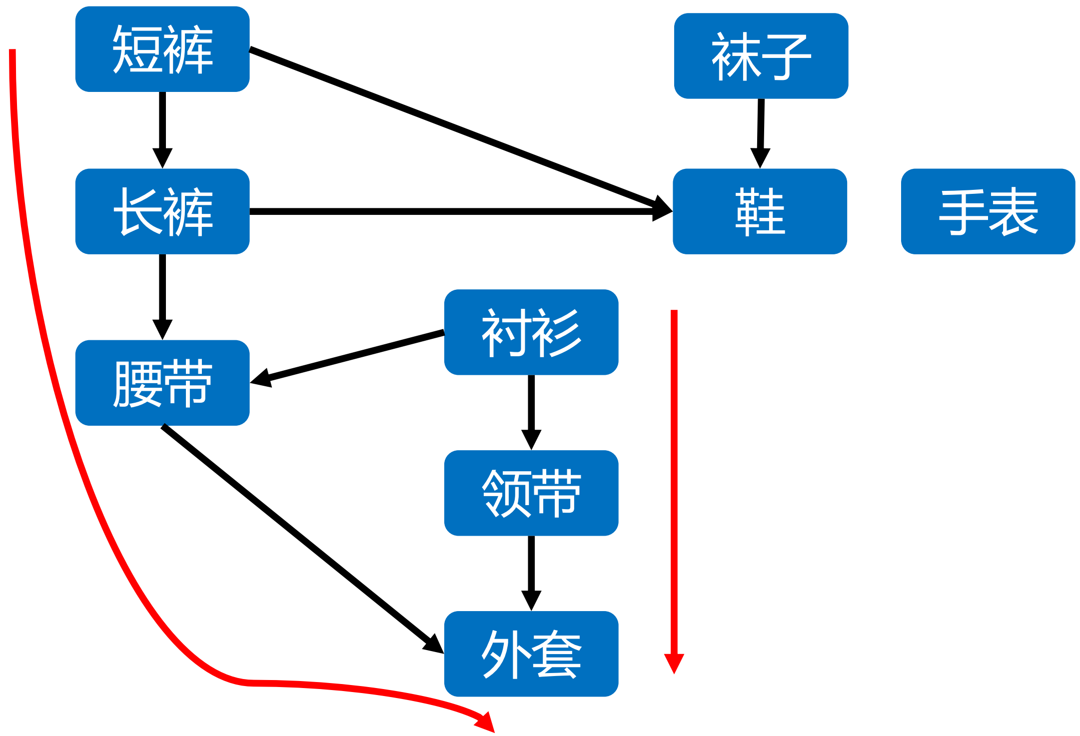
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后



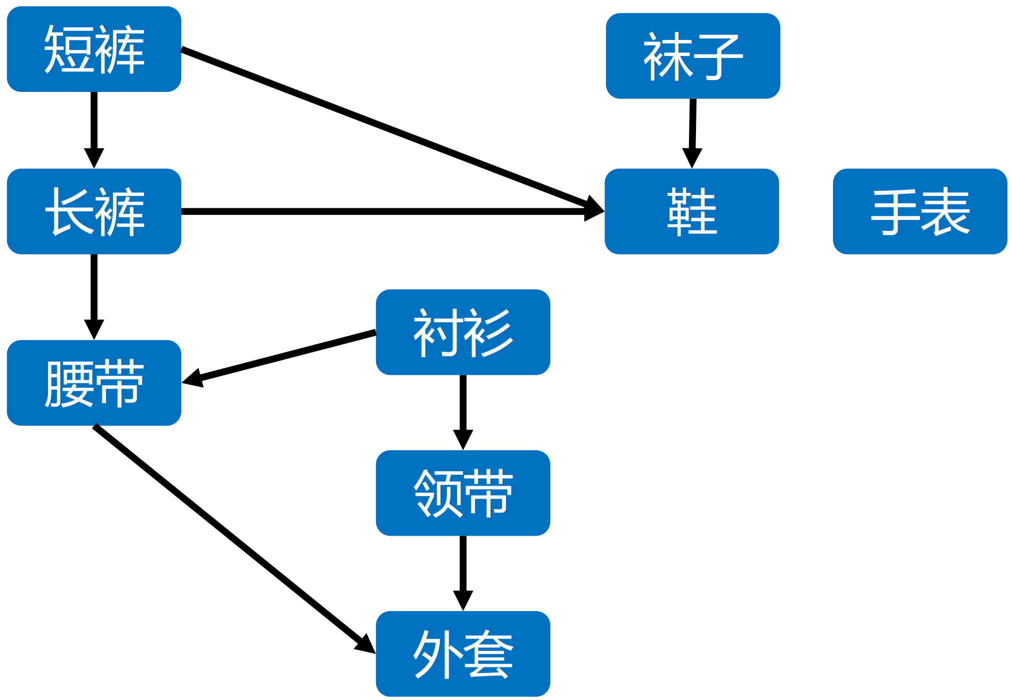
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚
    - 完成时刻越早



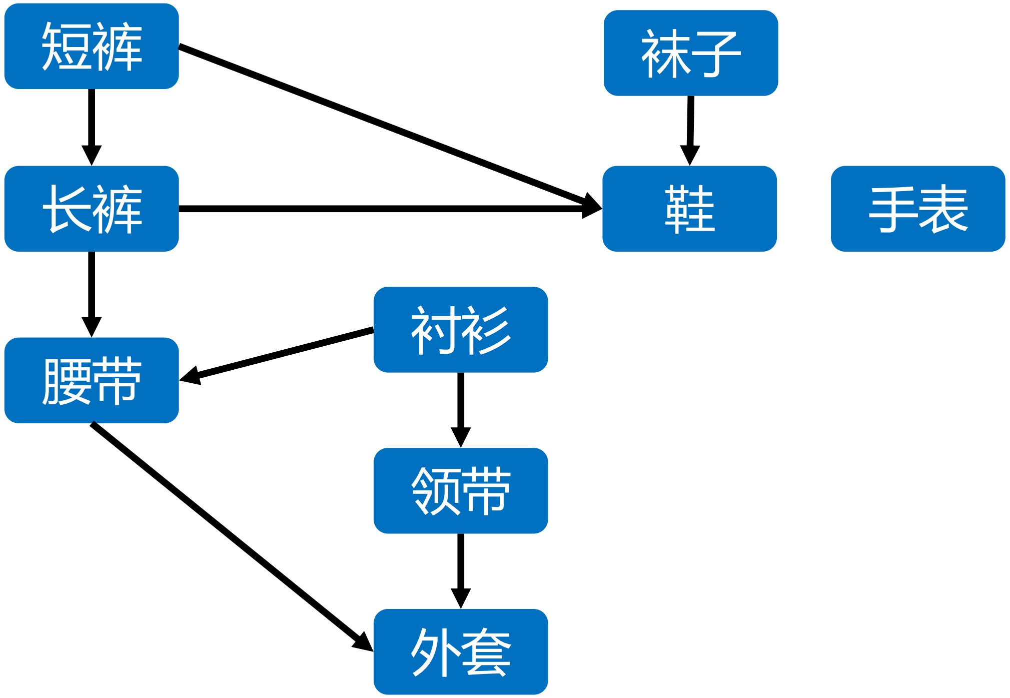
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序
    - 完成时刻越早：按完成时刻逆序



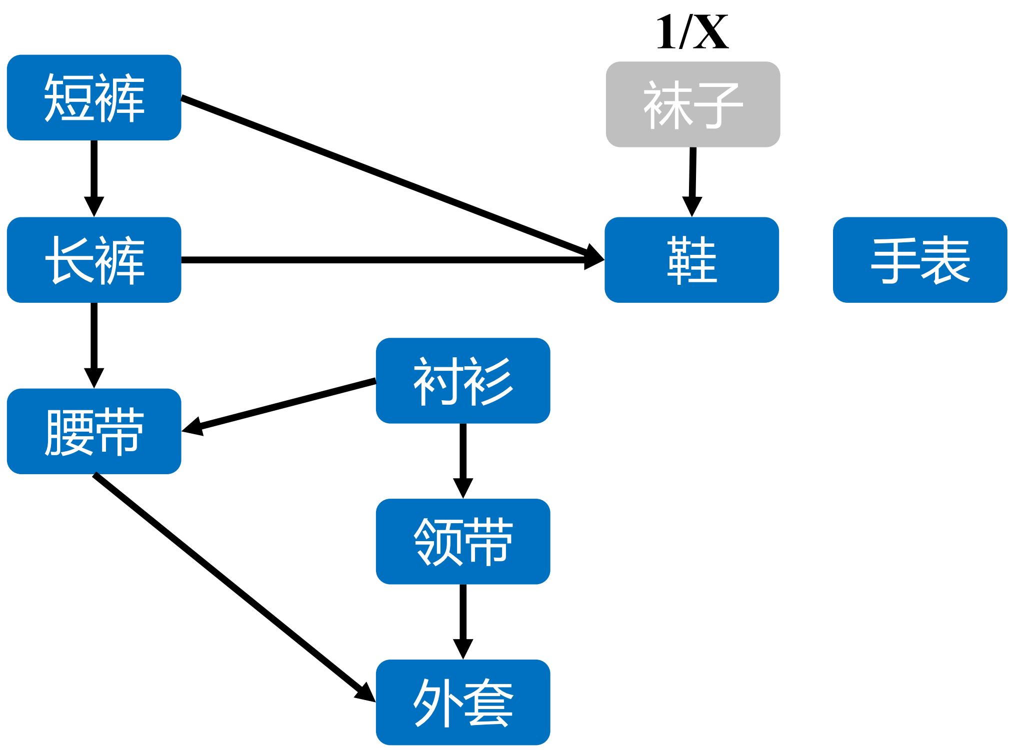
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序？
    - 完成时刻越早：按完成时刻逆序



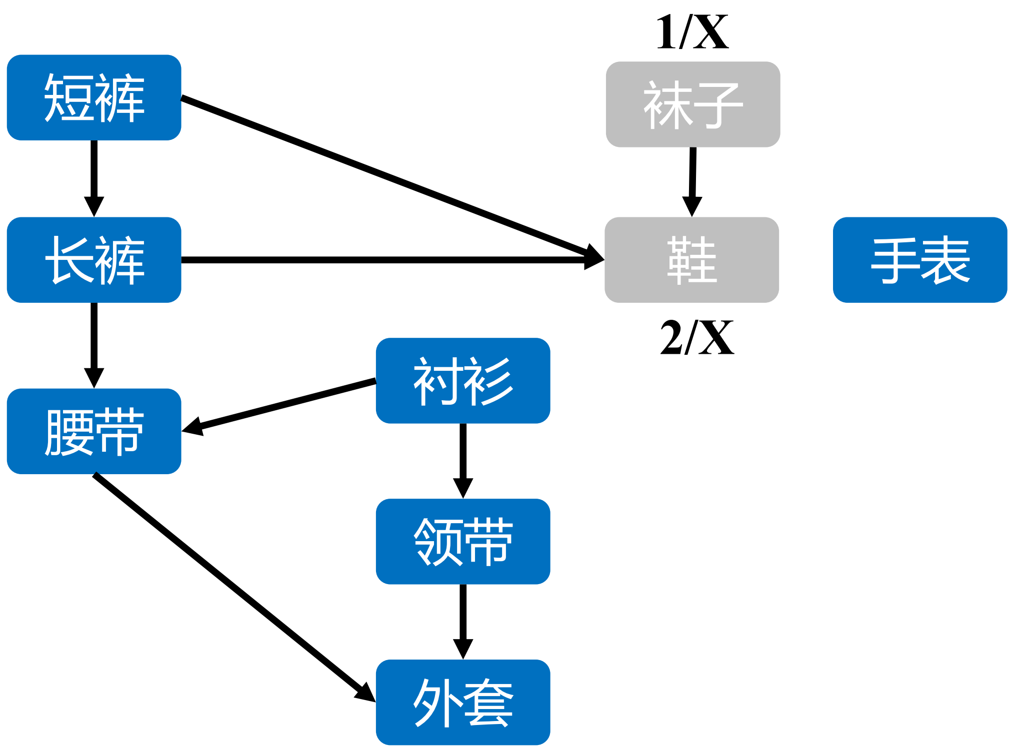
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序？
    - 完成时刻越早：按完成时刻逆序



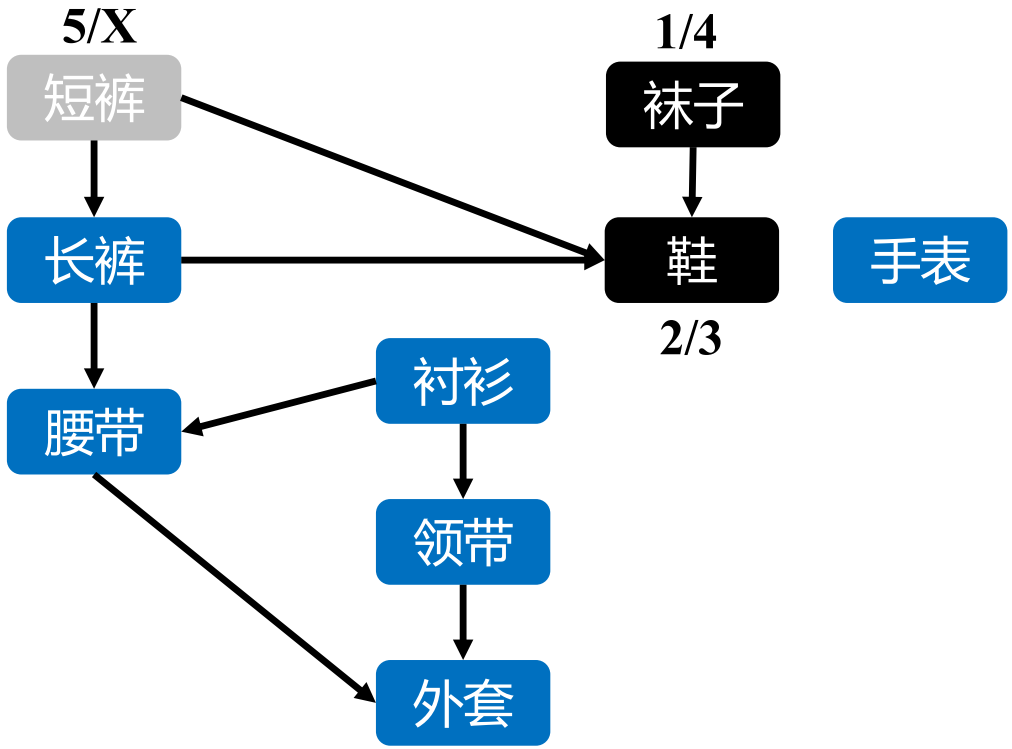
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序?
    - 完成时刻越早：按完成时刻逆序



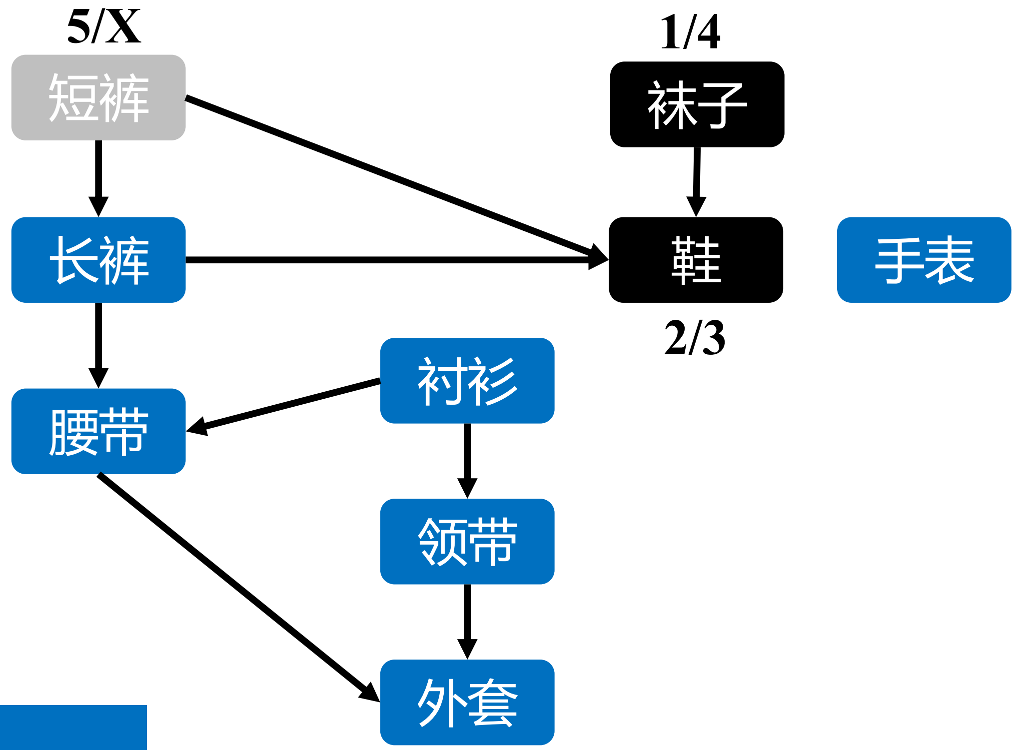
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序?
    - 完成时刻越早：按完成时刻逆序



# 算法思想

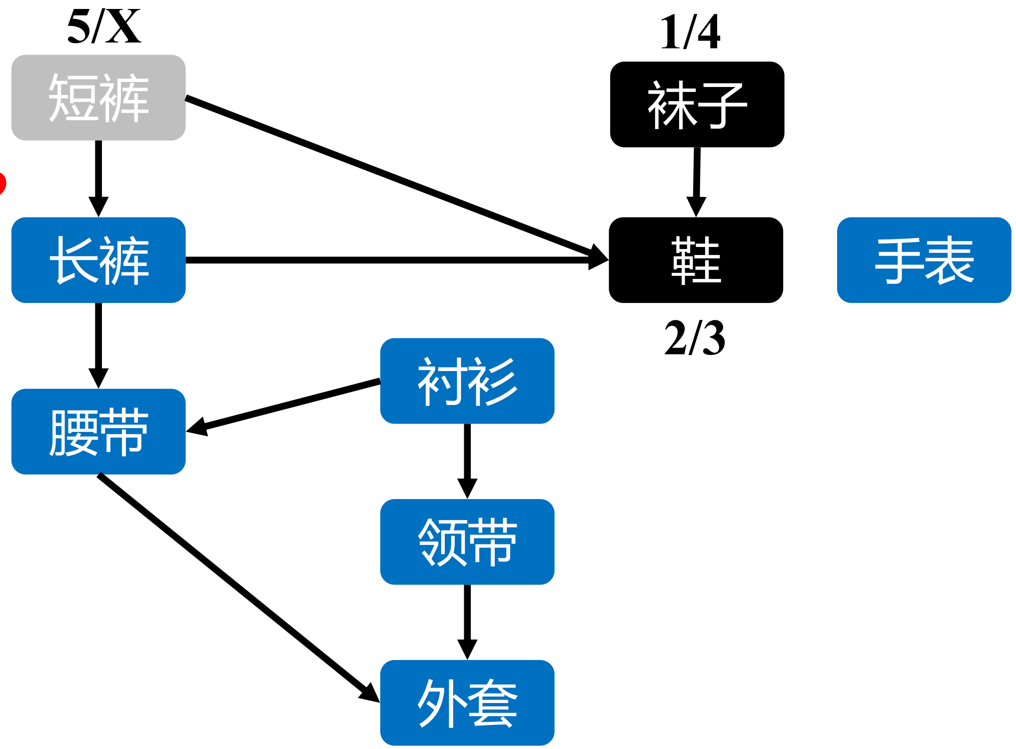
- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序
    - 完成时刻越早：按完成时刻逆序




若按发现时刻顺序执行，会先穿鞋后穿短裤

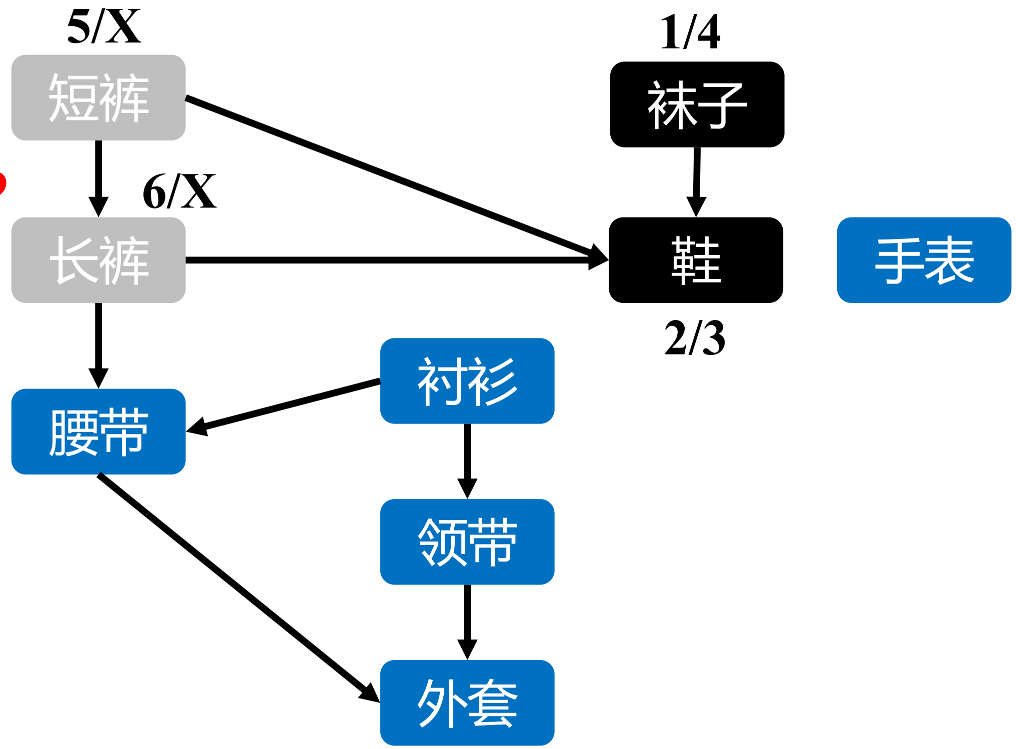
# 算法思想

- **从DFS的视角观察**
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序 ✗
    - 完成时刻越早：按完成时刻逆序？



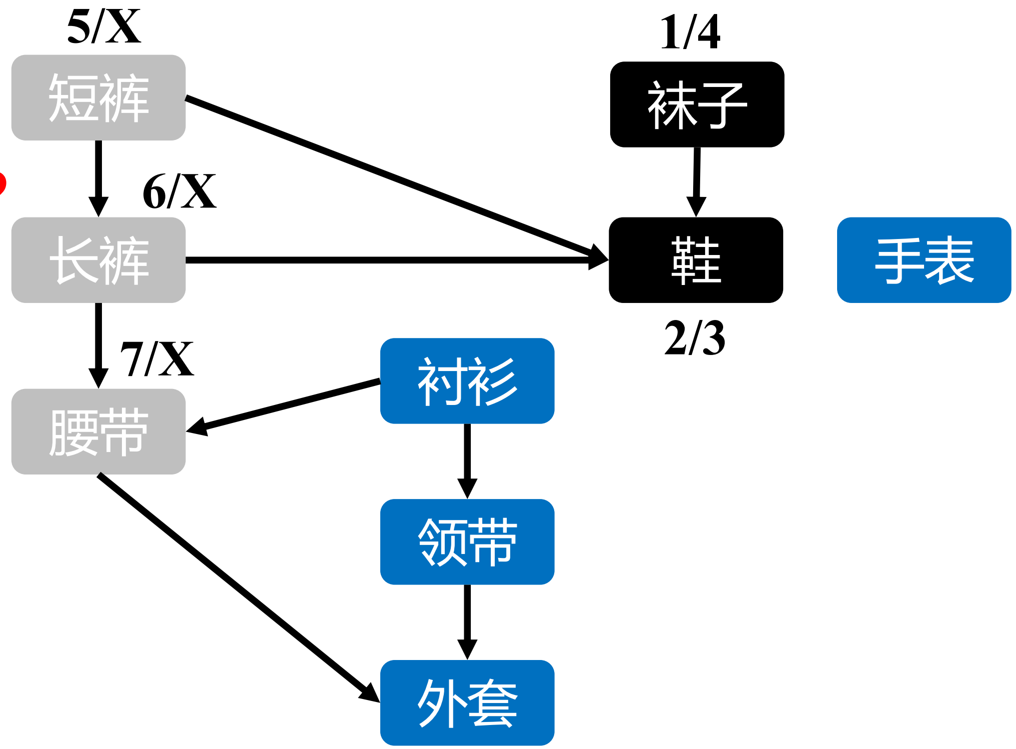
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序 ✗
    - 完成时刻越早：按完成时刻逆序？



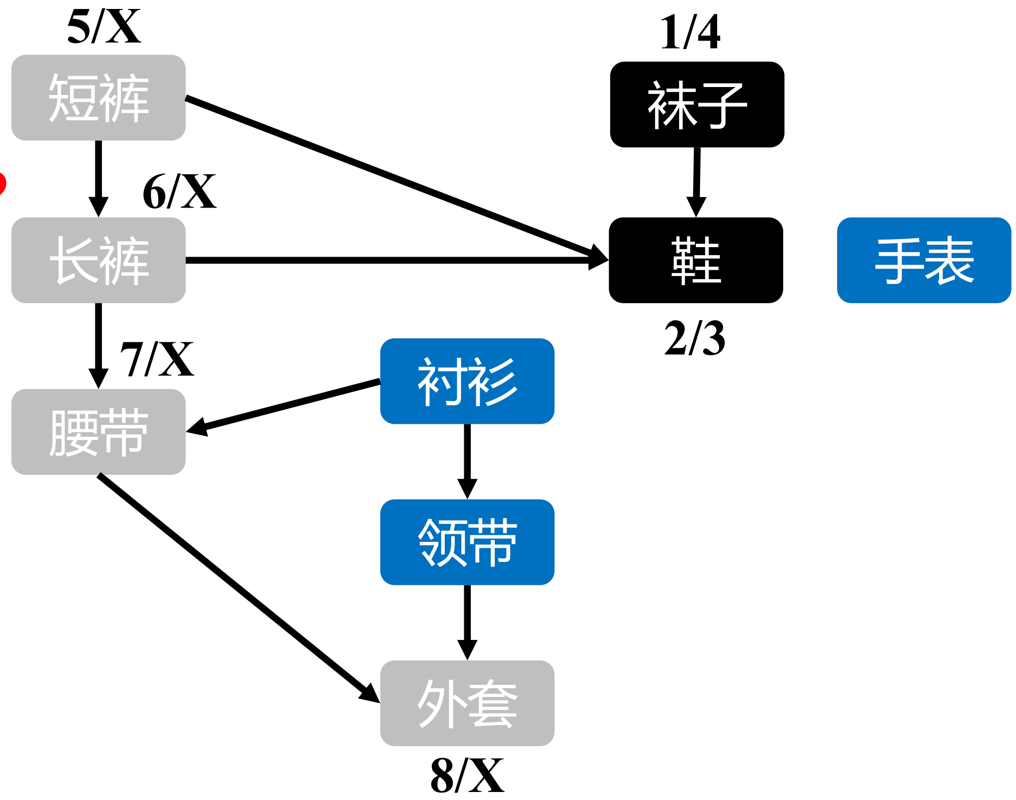
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序
    - 完成时刻越早：按完成时刻逆序？



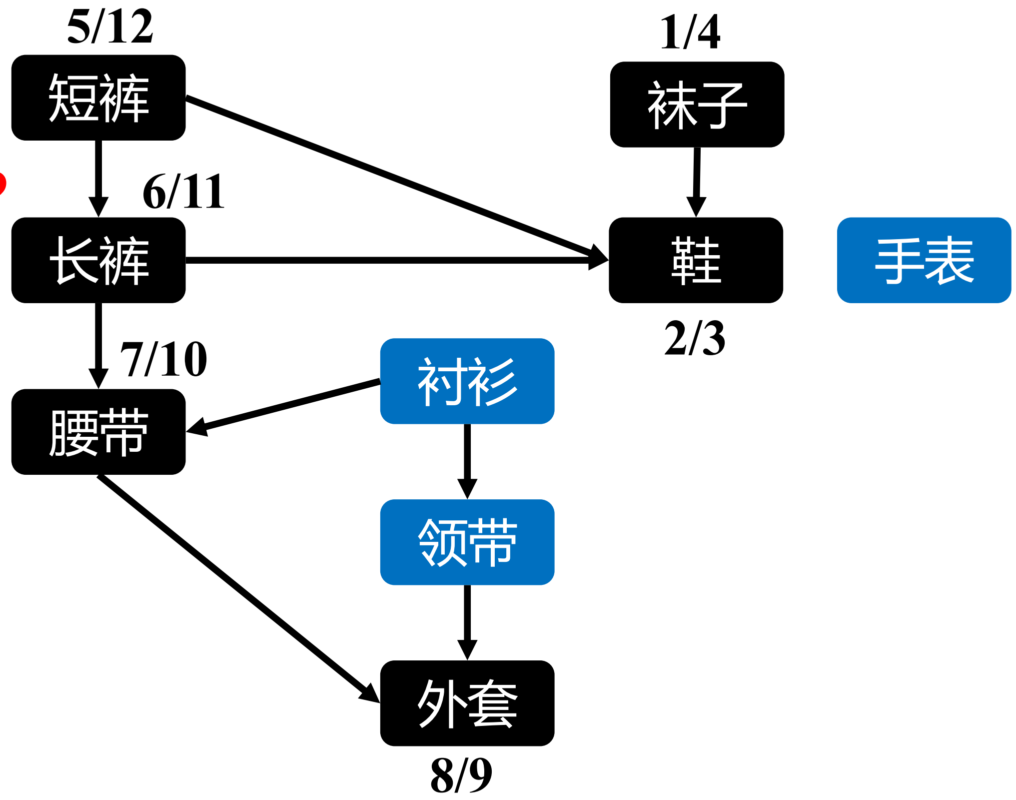
# 算法思想

- **从DFS的视角观察**
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序✗
    - 完成时刻越早：按完成时刻逆序？



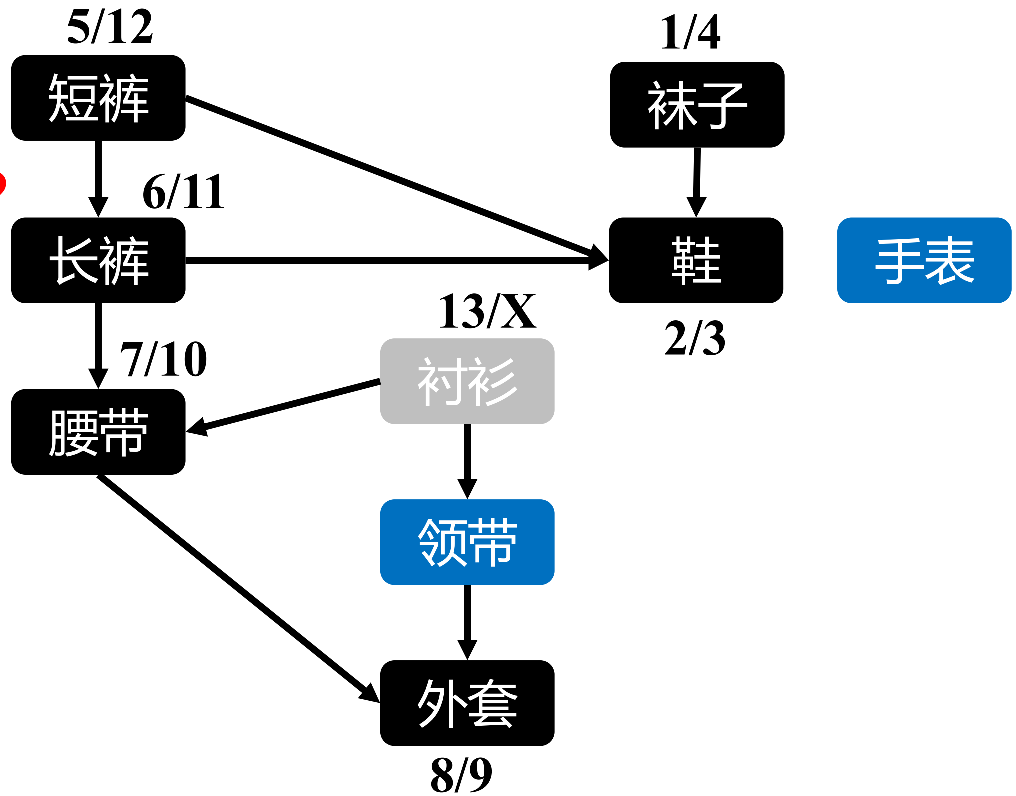
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序✗
    - 完成时刻越早：按完成时刻逆序？



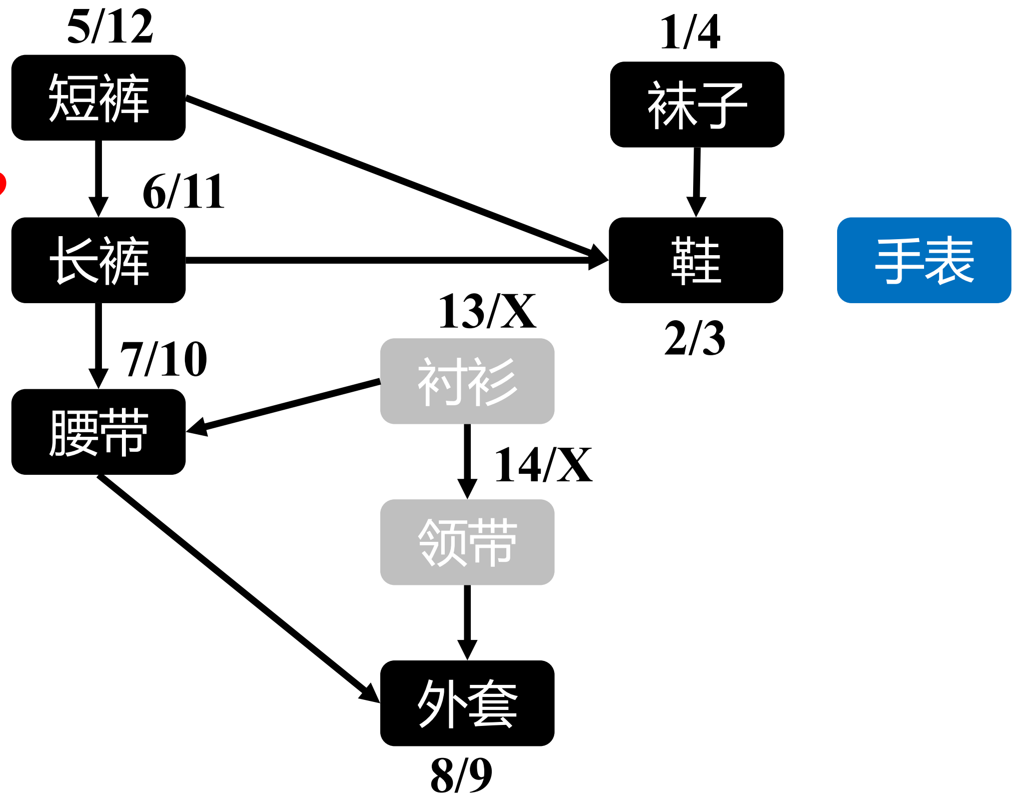
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序 ✗
    - 完成时刻越早：按完成时刻逆序？



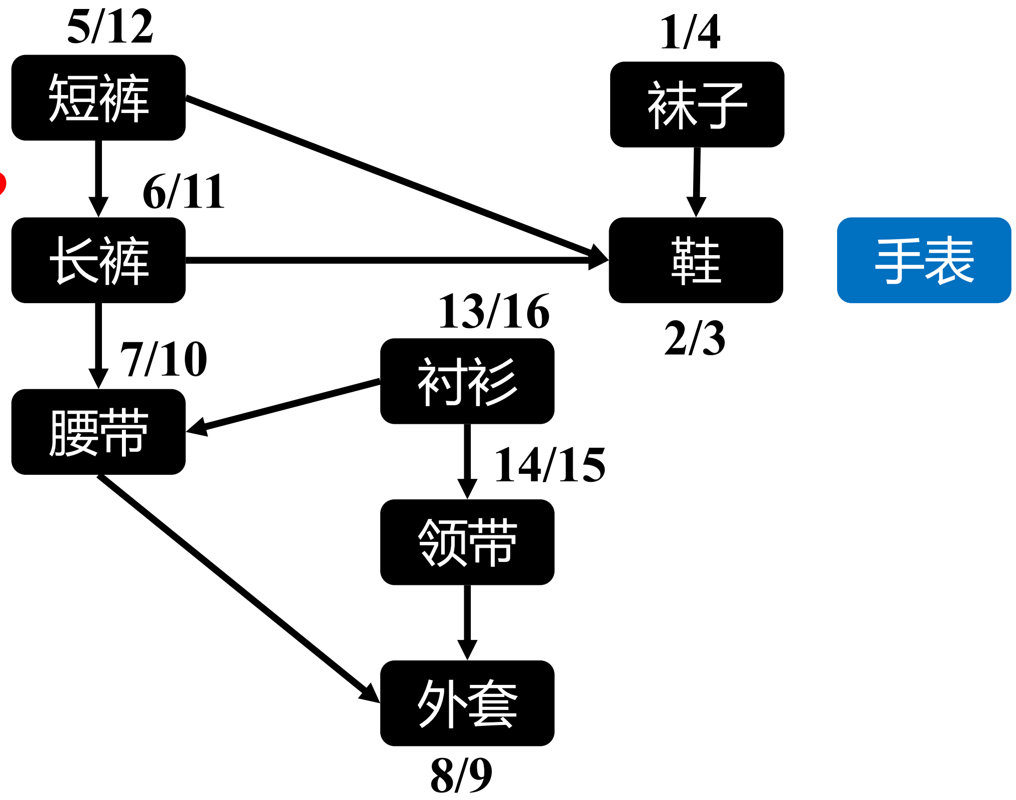
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序
    - 完成时刻越早：按完成时刻逆序?



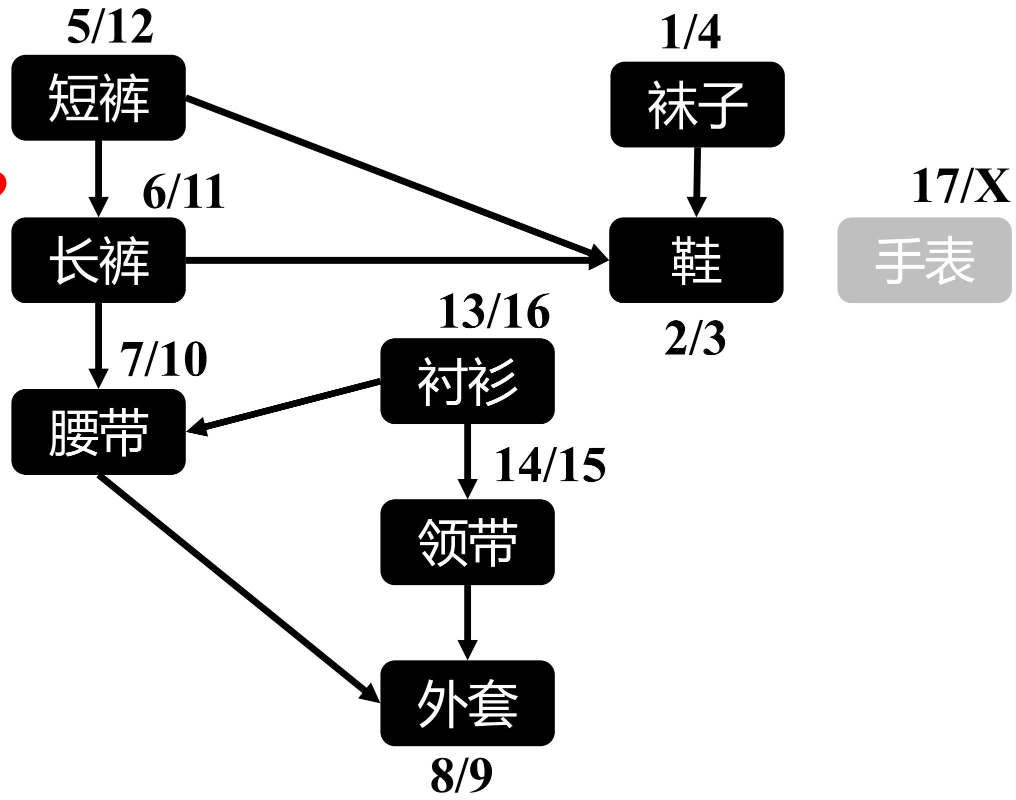
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序
    - 完成时刻越早：按完成时刻逆序？



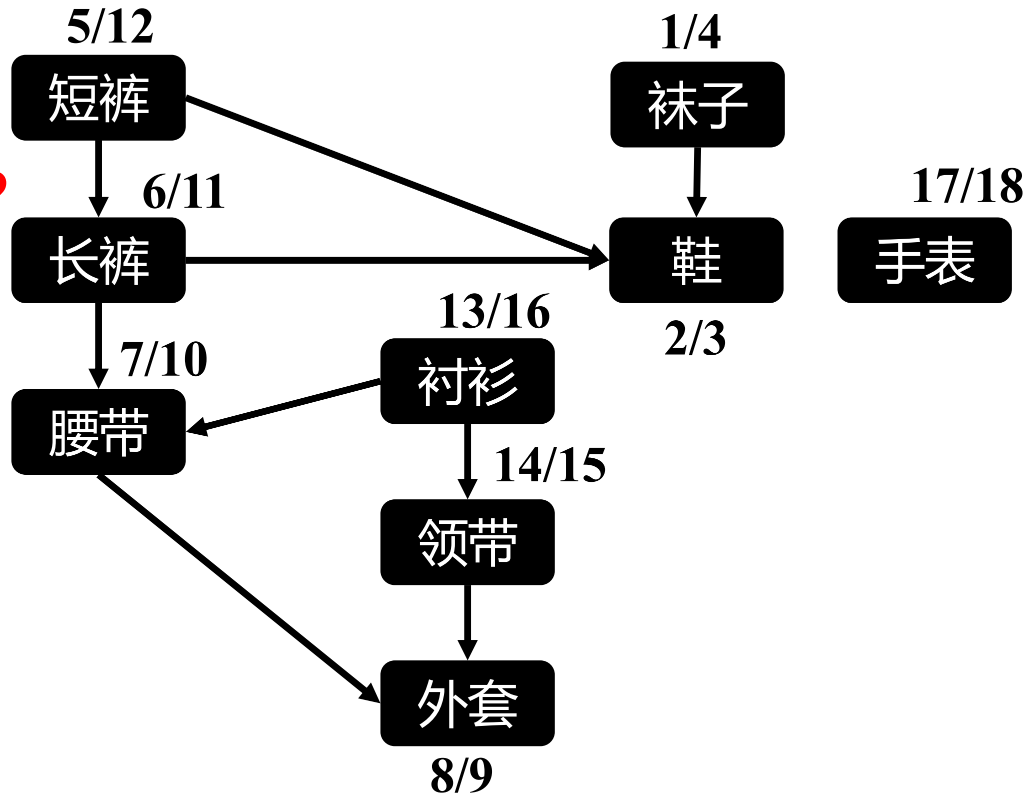
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序
    - 完成时刻越早：按完成时刻逆序?



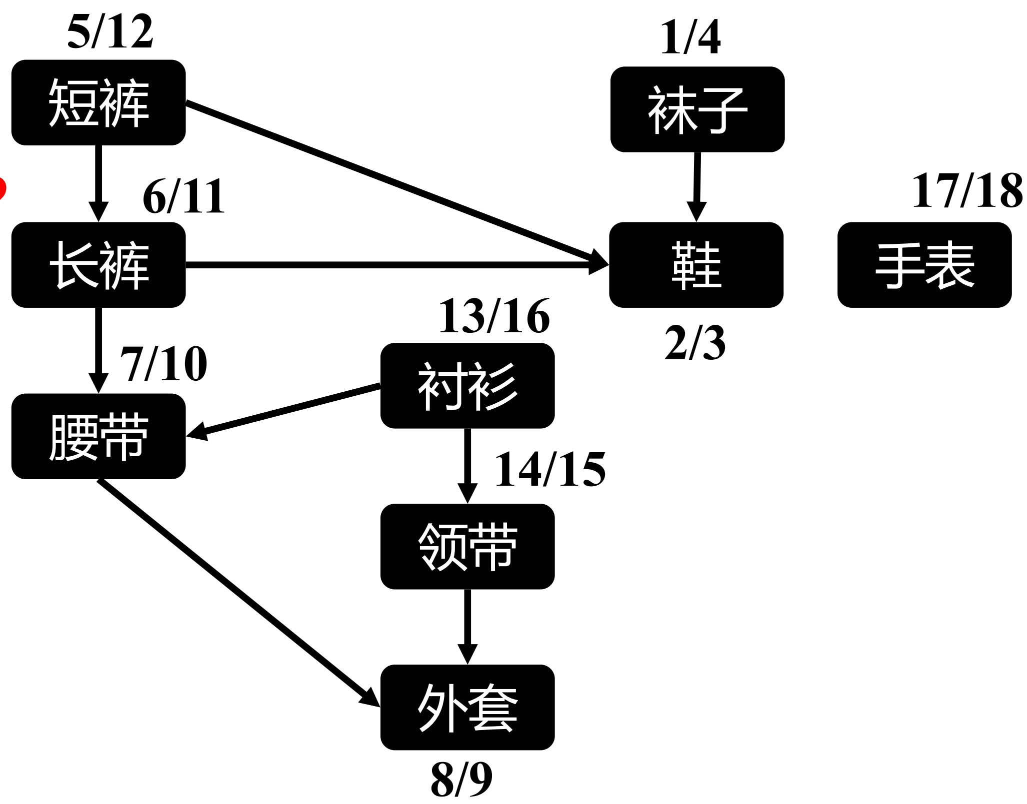
# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序✗
    - 完成时刻越早：按完成时刻逆序？



# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序✗
    - 完成时刻越早：按完成时刻逆序？

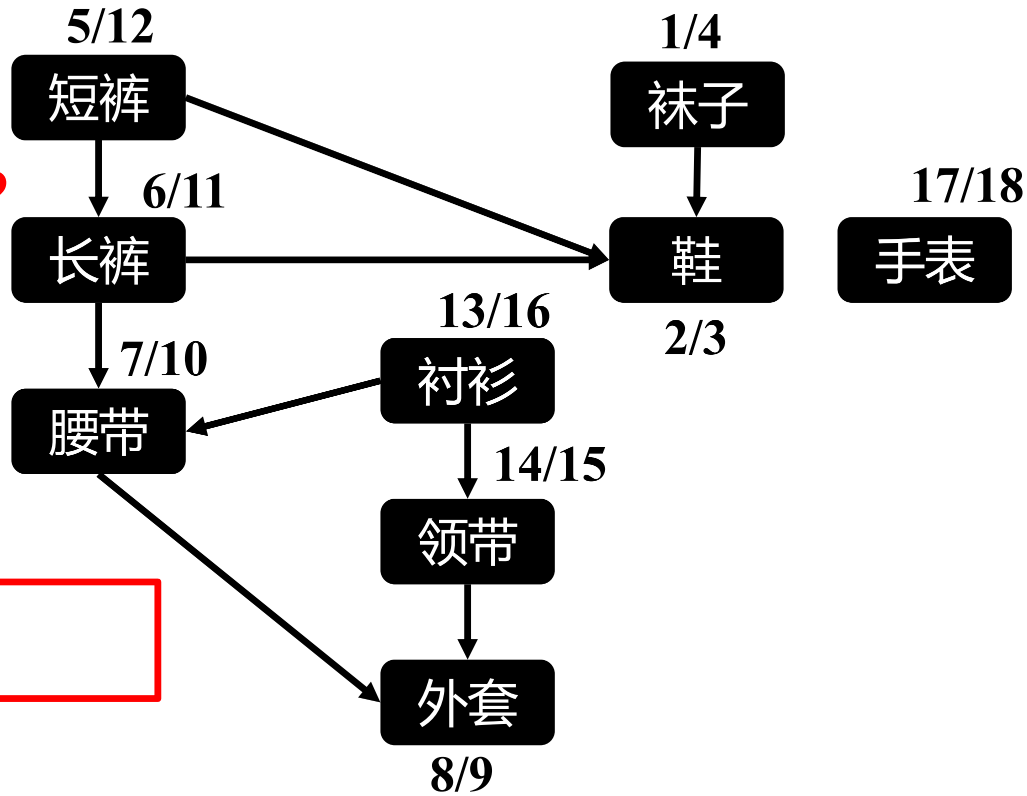



完成时刻逆序排列

手表 衬衫 领带 短裤 长裤 腰带 外套 袜子 鞋

# 算法思想

- 从DFS的视角观察
  - 穿衣顺序和搜索深度有关：深度越深，顺序越靠后
  - 深度越深
    - 发现时刻越晚：按发现时刻顺序
    - 完成时刻越早：按完成时刻逆序?




按完成时刻逆序是否正确？

完成时刻逆序排列

手表 衬衫 领带 短裤 长裤 腰带 外套 袜子 鞋

# 提纲

问题定义

广度优先策略

深度优先策略

算法分析

# 正确性证明

- 深度优先搜索确定的顺序：顶点完成时刻的逆序
- 拓扑序：对任意边 $(\boldsymbol{u}, \boldsymbol{v})$，$\boldsymbol{u}$ 在 $\boldsymbol{v}$ 前面

# 正确性证明

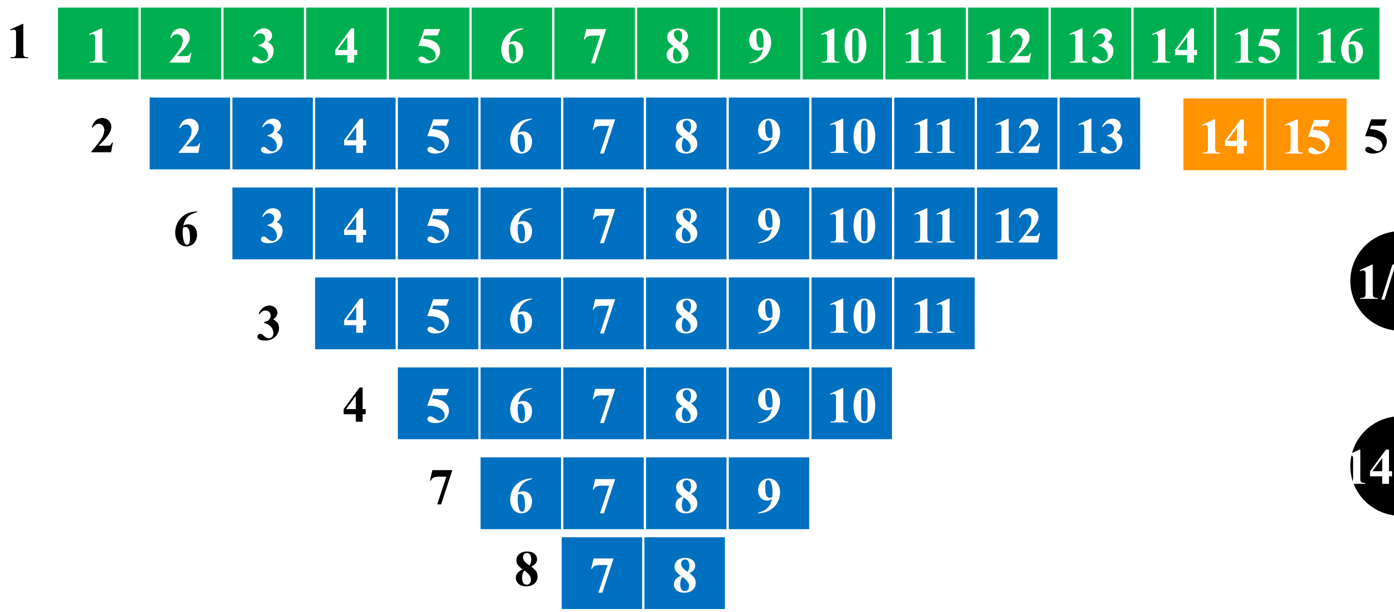
- 深度优先搜索确定的顺序：顶点完成时刻的逆序
- 拓扑序：对任意边 $(\boldsymbol{u}, \boldsymbol{v})$，$\boldsymbol{u}$ 在 $\boldsymbol{v}$ 前面
- 对任意边 $(\boldsymbol{u}, \boldsymbol{v})$，完成时刻满足：$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$ $\Rightarrow$ 算法正确

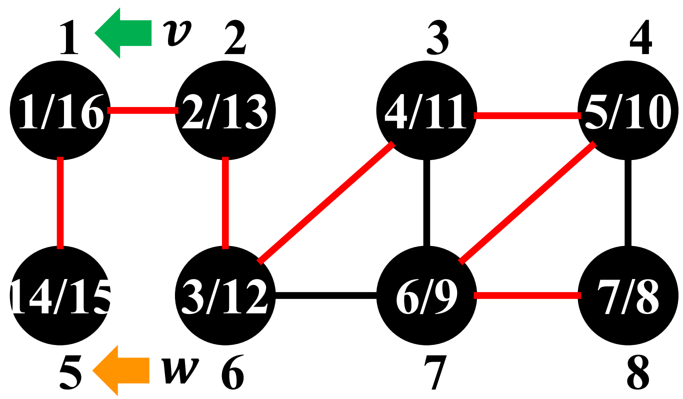
# 正确性证明

- 深度优先搜索确定的顺序：顶点完成时刻的逆序
- 拓扑序：对任意边 $(\boldsymbol{u}, \boldsymbol{v})$，$\boldsymbol{u}$ 在 $\boldsymbol{v}$ 前面
- 对任意边 $(\boldsymbol{u}, \boldsymbol{v})$，完成时刻满足：$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$ ⟶ 算法正确
  - 证明：设当前顶点为$\boldsymbol{u}$，搜索顶点$\boldsymbol{v}$

# 正确性证明

- 深度优先搜索确定的顺序：顶点完成时刻的逆序
- 拓扑序：对任意边 $(u,v)$，$u$ 在 $v$ 前面
- 对任意边 $(u,v)$，完成时刻满足：$f(u) > f(v)$ → 算法正确
  - 证明：设当前顶点为$u$，搜索顶点$v$
    - 若$v$为白色，$v$是$u$的后代，$f(u) > f(v)$ （括号化定理）



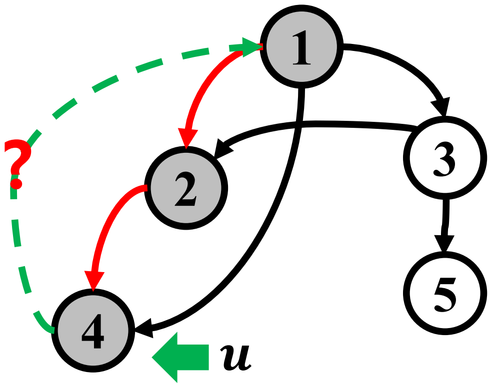



# 正确性证明

- 深度优先搜索确定的顺序：顶点完成时刻的逆序
- 拓扑序：对任意边 $(\boldsymbol{u}, \boldsymbol{v})$，$\boldsymbol{u}$ 在 $\boldsymbol{v}$ 前面
- 对任意边 $(\boldsymbol{u}, \boldsymbol{v})$，完成时刻满足：$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$ → 算法正确
  - 证明：设当前顶点为$\boldsymbol{u}$，搜索顶点$\boldsymbol{v}$
    - 若$\boldsymbol{v}$为白色，$\boldsymbol{v}$是$\boldsymbol{u}$的后代，$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$ （括号化定理）
    - 若$\boldsymbol{v}$为黑色，$\boldsymbol{v}$已经完成，$\boldsymbol{u}$尚未完成，$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$

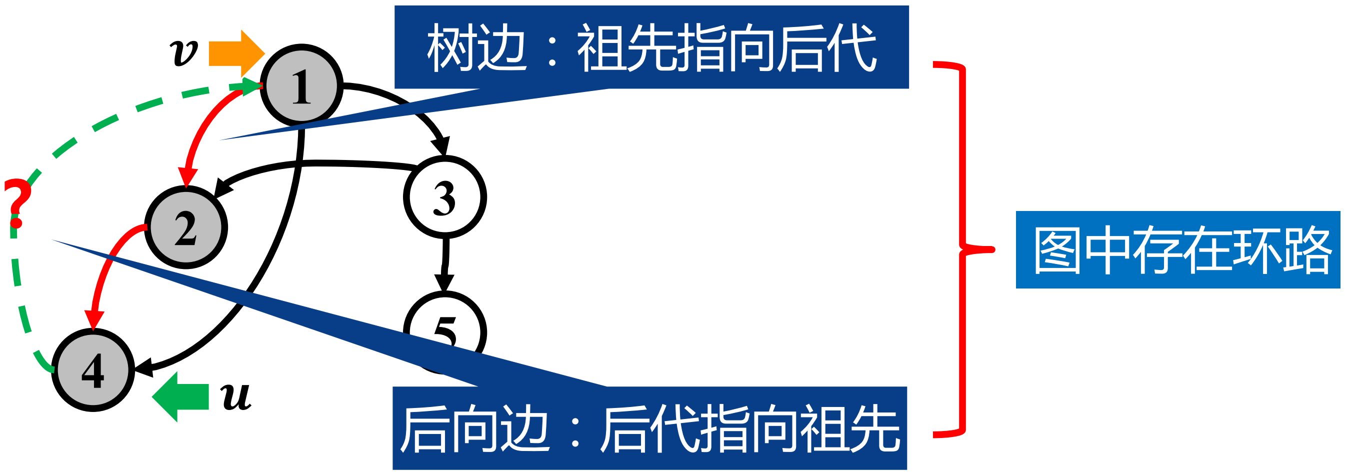
# 正确性证明

- 深度优先搜索确定的顺序：顶点完成时刻的逆序
- 拓扑序：对任意边 $(\boldsymbol{u}, \boldsymbol{v})$，$\boldsymbol{u}$ 在 $\boldsymbol{v}$ 前面
- 对任意边 $(\boldsymbol{u}, \boldsymbol{v})$，完成时刻满足：$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$ → 算法正确
  - 证明：设当前顶点为$\boldsymbol{u}$，搜索顶点$\boldsymbol{v}$
    - 若$\boldsymbol{v}$为白色，$\boldsymbol{v}$是$\boldsymbol{u}$的后代，$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$ （括号化定理）
    - 若$\boldsymbol{v}$为黑色，$\boldsymbol{v}$已经完成，$\boldsymbol{u}$尚未完成，$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$
    - 若$\boldsymbol{v}$是灰色？



# 正确性证明

- 深度优先搜索确定的顺序：顶点完成时刻的逆序
- 拓扑序：对任意边 $(\boldsymbol{u},\boldsymbol{v})$，$\boldsymbol{u}$ 在 $\boldsymbol{v}$ 前面
- 对任意边 $(\boldsymbol{u},\boldsymbol{v})$，完成时刻满足：$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$ ⟶ 算法正确
  - 证明：设当前顶点为$\boldsymbol{u}$，搜索顶点$\boldsymbol{v}$
    - 若$\boldsymbol{v}$为白色，$\boldsymbol{v}$是$\boldsymbol{u}$的后代，$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$ （括号化定理）
    - 若$\boldsymbol{v}$为黑色，$\boldsymbol{v}$已经完成，$\boldsymbol{u}$尚未完成，$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$
    - 若$\boldsymbol{v}$是灰色？不可能！因为有向无环图不存在后向边



# 正确性证明

- 深度优先搜索确定的顺序：顶点完成时刻的逆序
- 拓扑序：对任意边 $(\boldsymbol{u},\boldsymbol{v})$，$\boldsymbol{u}$ 在 $\boldsymbol{v}$ 前面
- 对任意边 $(\boldsymbol{u},\boldsymbol{v})$，完成时刻满足：$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$ → 算法正确
  - 证明：设当前顶点为$\boldsymbol{u}$，搜索顶点$\boldsymbol{v}$
    - 若$\boldsymbol{v}$为白色，$\boldsymbol{v}$是$\boldsymbol{u}$的后代，$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$ （括号化定理）
    - 若$\boldsymbol{v}$为黑色，$\boldsymbol{v}$已经完成，$\boldsymbol{u}$尚未完成，$\boldsymbol{f}(\boldsymbol{u}) > \boldsymbol{f}(\boldsymbol{v})$
    - 若$\boldsymbol{v}$是灰色？不可能！因为有向无环图不存在后向边

# 伪代码

- **Topological-Sort-DFS(*G*)**

**输入:** 图$G$
**输出:** 顶点拓扑序
$L \leftarrow DFS(G)$
**return** $L.reverse()$

数组中元素逆序排列

# 伪代码

- **Topological-Sort-DFS(*G*)**

**输入:** 图$G$
**输出:** 顶点拓扑序
$L \leftarrow DFS(G)$
**return** $L.reverse()$

问题：如何在搜索过程中得到按完成时刻顺序排列的顶点？

# 伪代码

- **DFS($G$)**

```
输入: 图 G
新建数组 color[1..V], L[1..V]
for v ∈ V do
 |  color[v] ← WHITE
end
for v ∈ V do
 |  if color[v] = WHITE then
 |   |  L' ← DFS-Visit(G, v)
 |   |  向L结尾追加L'
 |  end
end
return L
```

- **DFS-Visit($G, v$)**

```
输入: 图 G, 顶点 v
输出: 按完成时刻从早到晚排列的顶点 L
color[v] ← GRAY
初始化空队列 L
for w ∈ G.Adj[v] do
 |  if color[w] = WHITE then
 |   |  向L追加 DFS-Visit(G, w)
 |  end
end
color[v] ← BLACK
向L结尾追加顶点v
return L
```

顶点按完成时刻排列

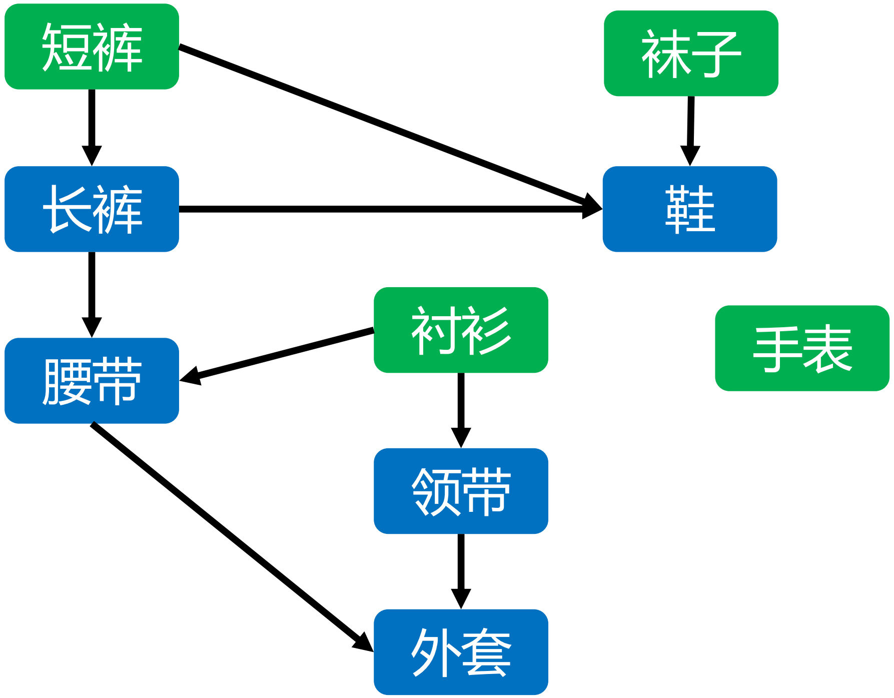
# 时间复杂度

- **Topological-Sort-DFS($G$)**

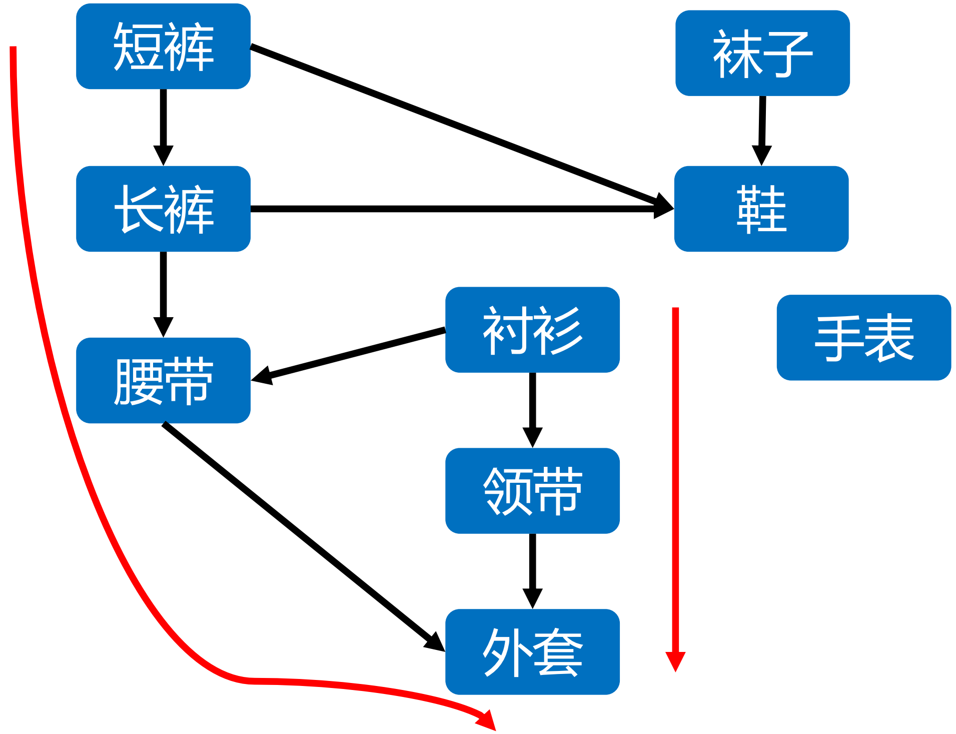
**输入:** 图$G$
**输出:** 顶点拓扑序
$L \leftarrow DFS(G)$
**return** $L.reverse()$

时间复杂度：$O(|V| + |E|)$

# 小结




广度优先
顺序思想：把容易完成的事优先完成



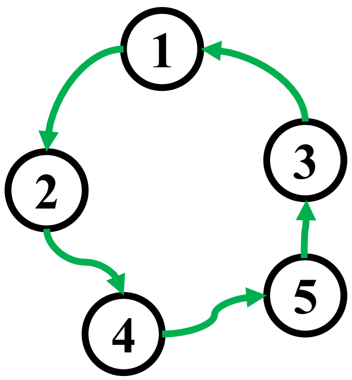

深度优先
逆序思想：把不易完成的事放到后面

# 图算法篇：强连通分量

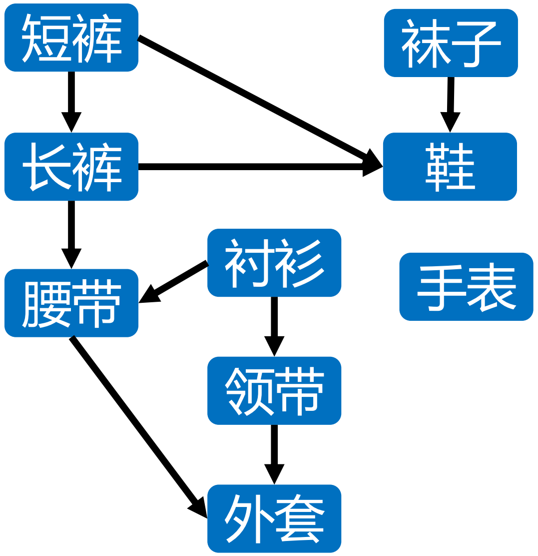
北京航空航天大学
计算机学院

中国大学MOOC北航《算法设计与分析》

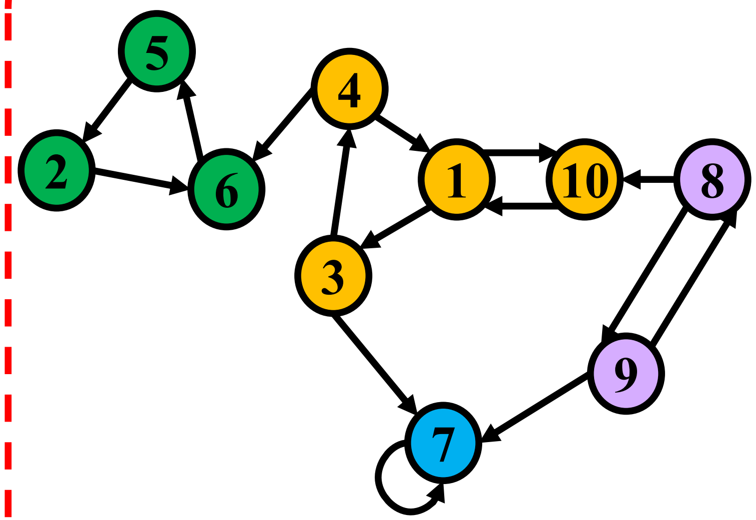
# 深度优先搜索应用




环路的存在性判断




拓扑排序



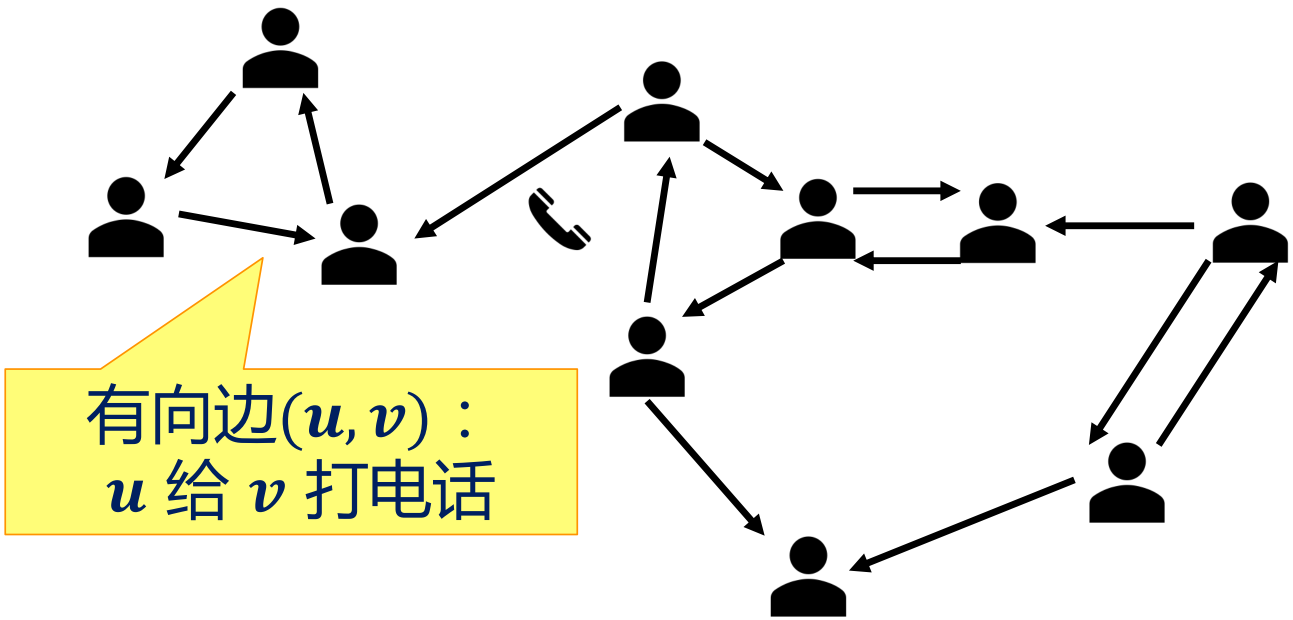

强连通分量

# 提纲

- 问题背景与定义
- 算法框架与实例
- 伪代码与复杂度
- 算法正确性证明

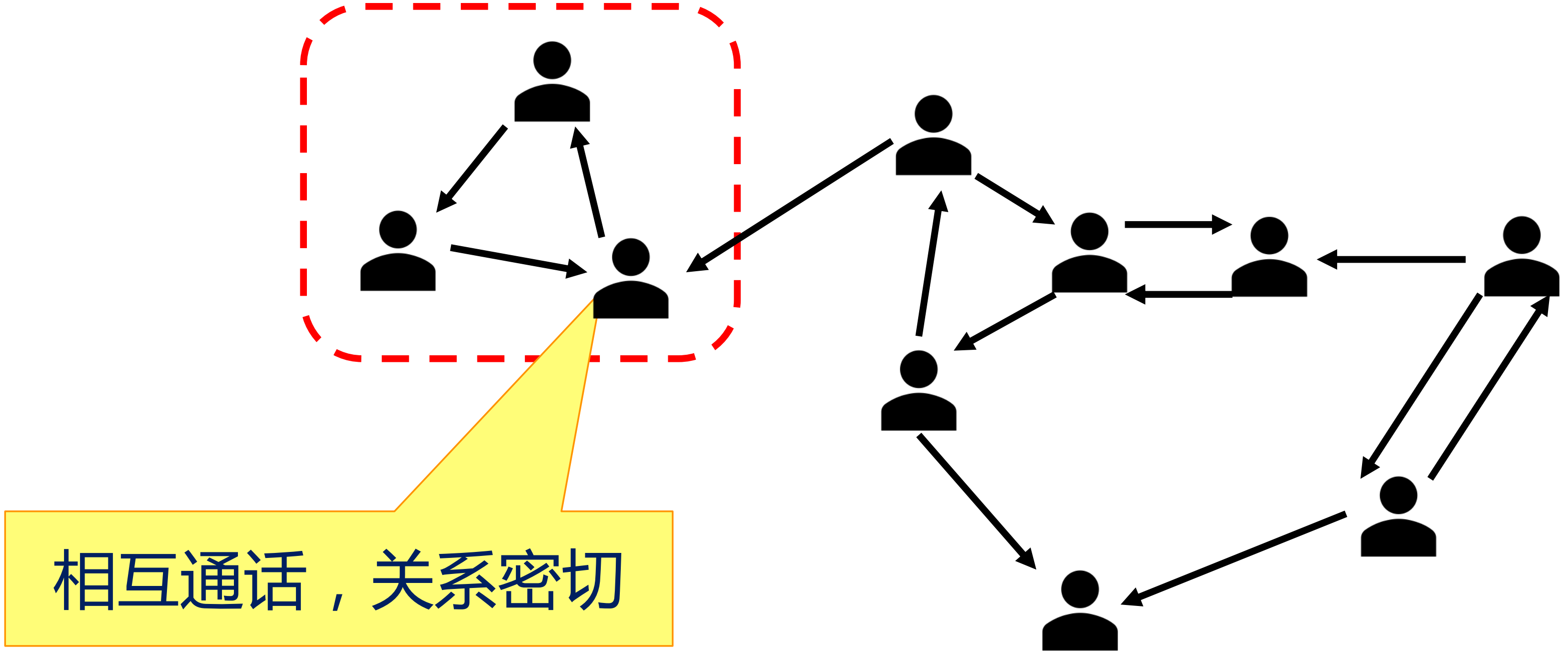
# 问题背景

- 社交圈划分
  - 如何把人群按通话记录划分成不同的社交圈？



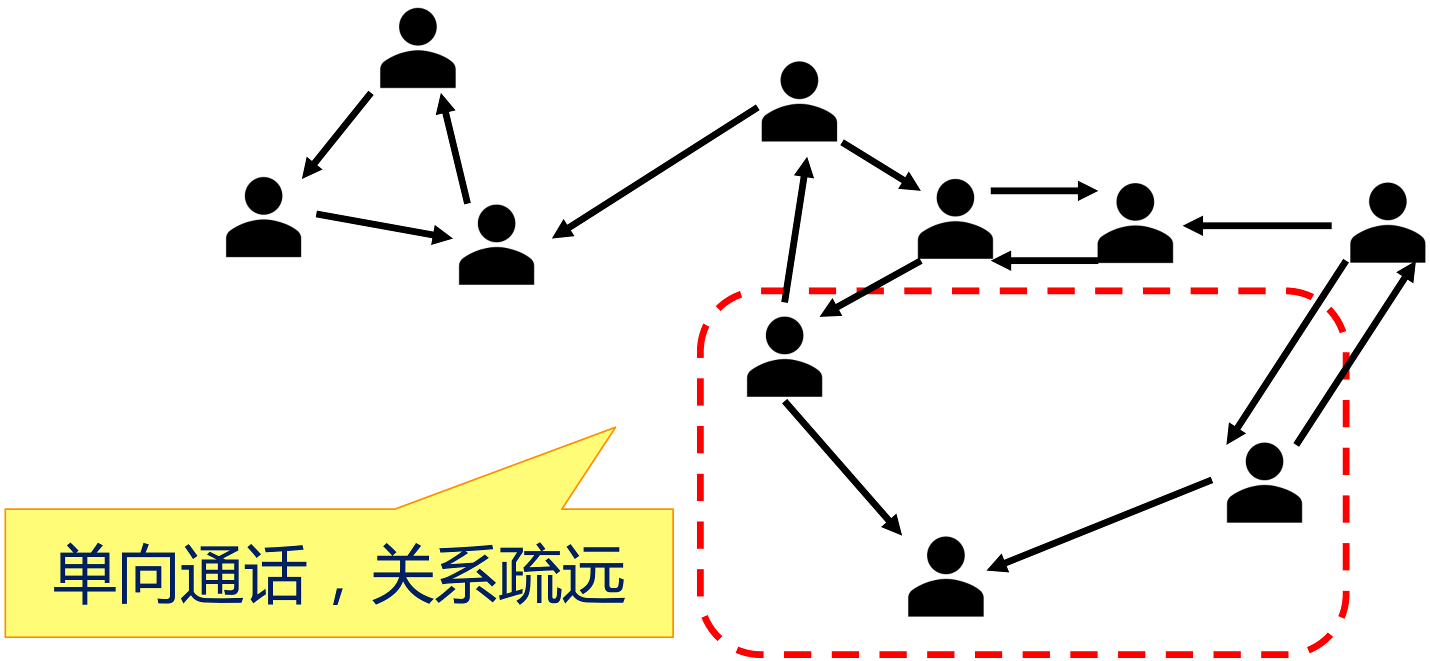
# 问题背景

- 社交圈划分
  - 如何把人群按通话记录划分成不同的社交圈？



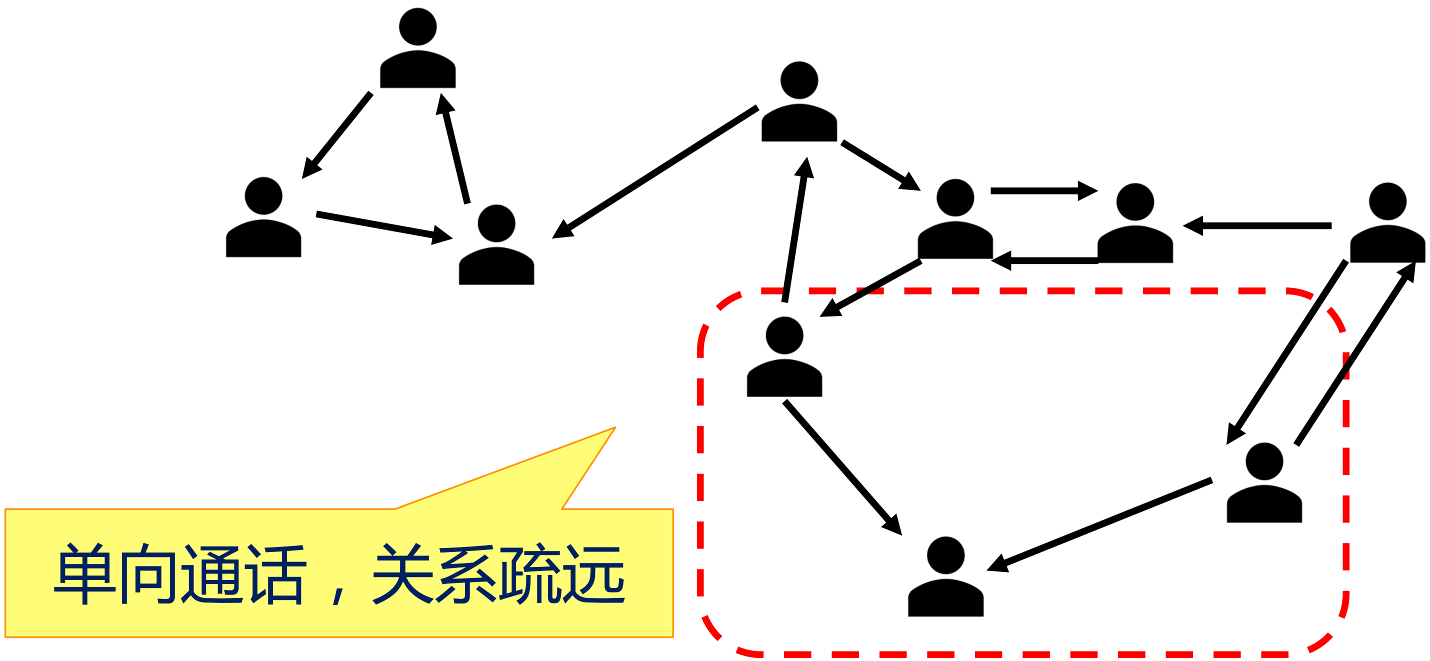
# 问题背景

- 社交圈划分
  - 如何把人群按通话记录划分成不同的社交圈？



# 问题背景

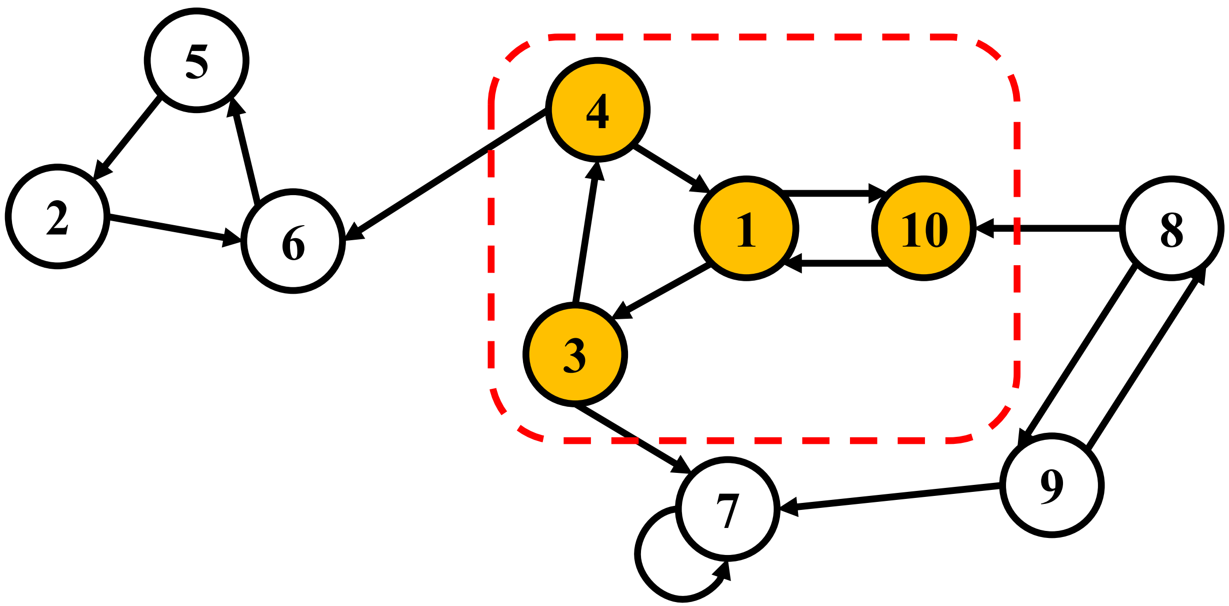
- 社交圈划分
  - 如何把人群按通话记录划分成不同的社交圈？




问题：如何严格定义关系的亲密程度？

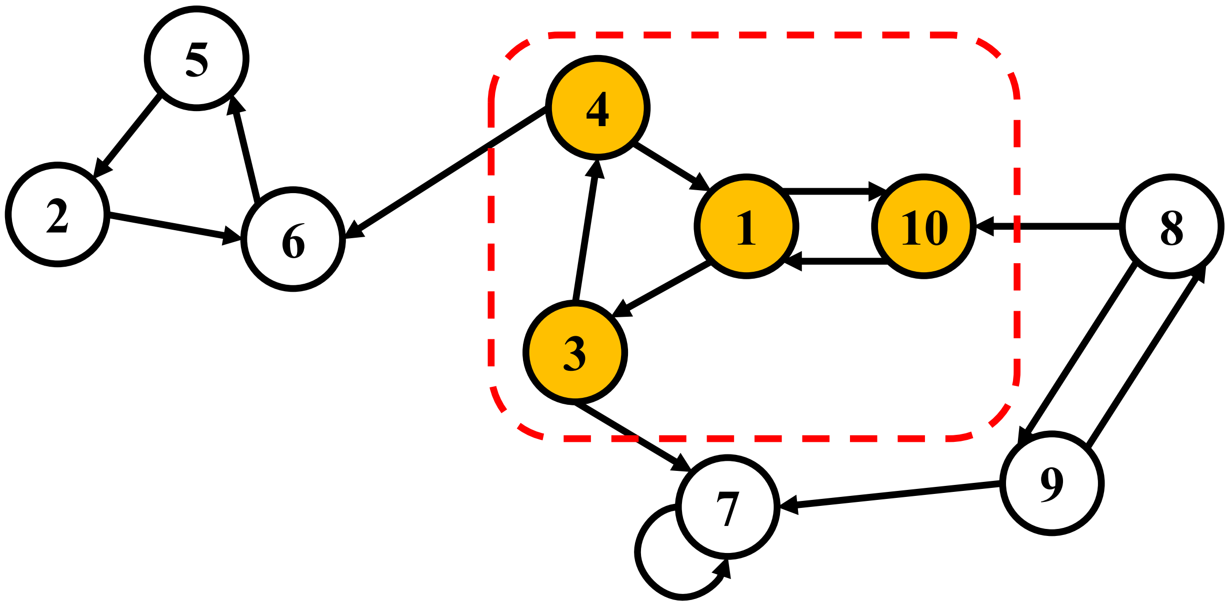
# 问题背景

- 社交圈划分
  - 如何把人群按通话记录划分成不同的社交圈？
- 强连通分量
  - 一个强连通分量是顶点的子集



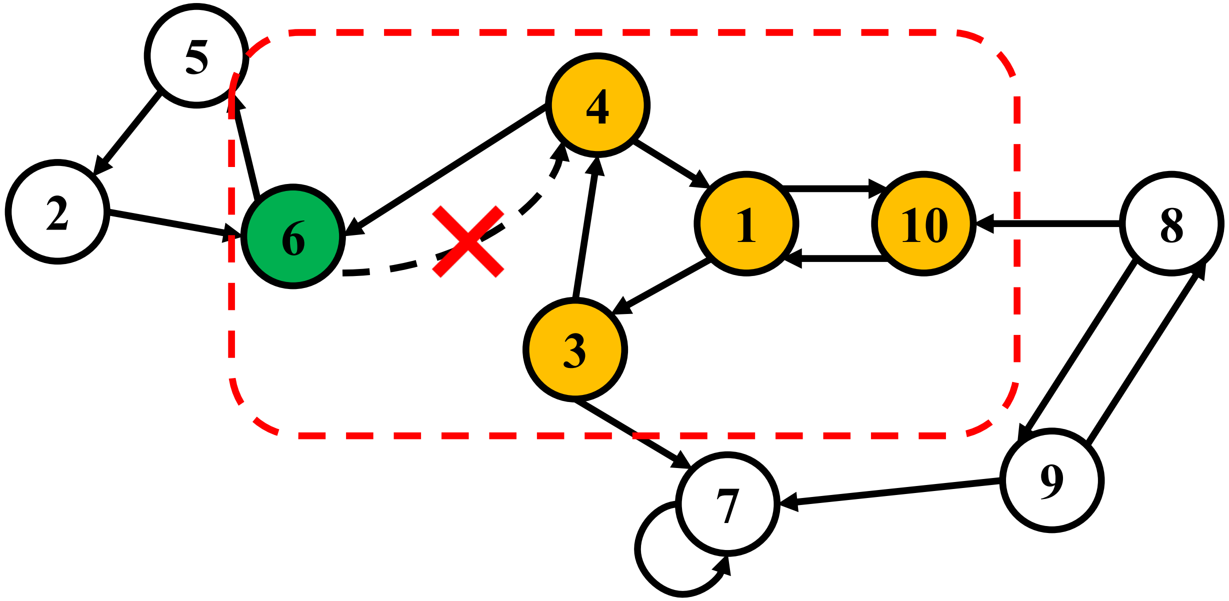
# 问题背景

- 社交圈划分
  - 如何把人群按通话记录划分成不同的社交圈？
- 强连通分量
  - 一个强连通分量是顶点的子集
  - 强连通分量中任意两点相互可达



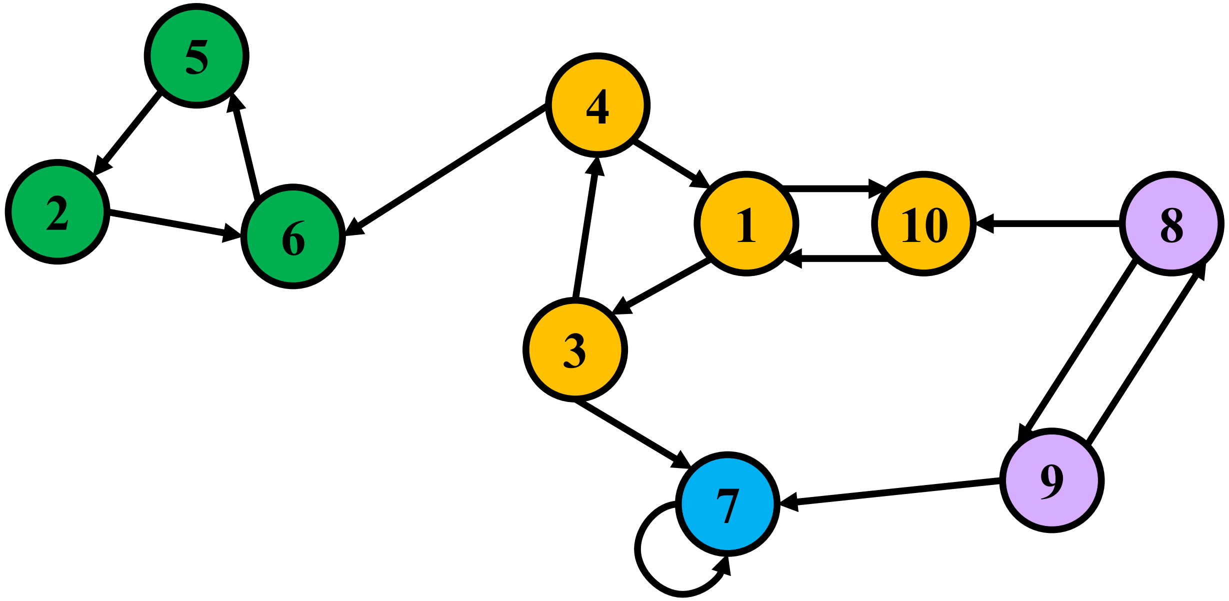
# 问题背景

- 社交圈划分
  - 如何把人群按通话记录划分成不同的社交圈？
- 强连通分量
  - 一个强连通分量是顶点的子集
  - 强连通分量中任意两点相互可达
  - 满足最大性：加入新顶点，不保证相互可达



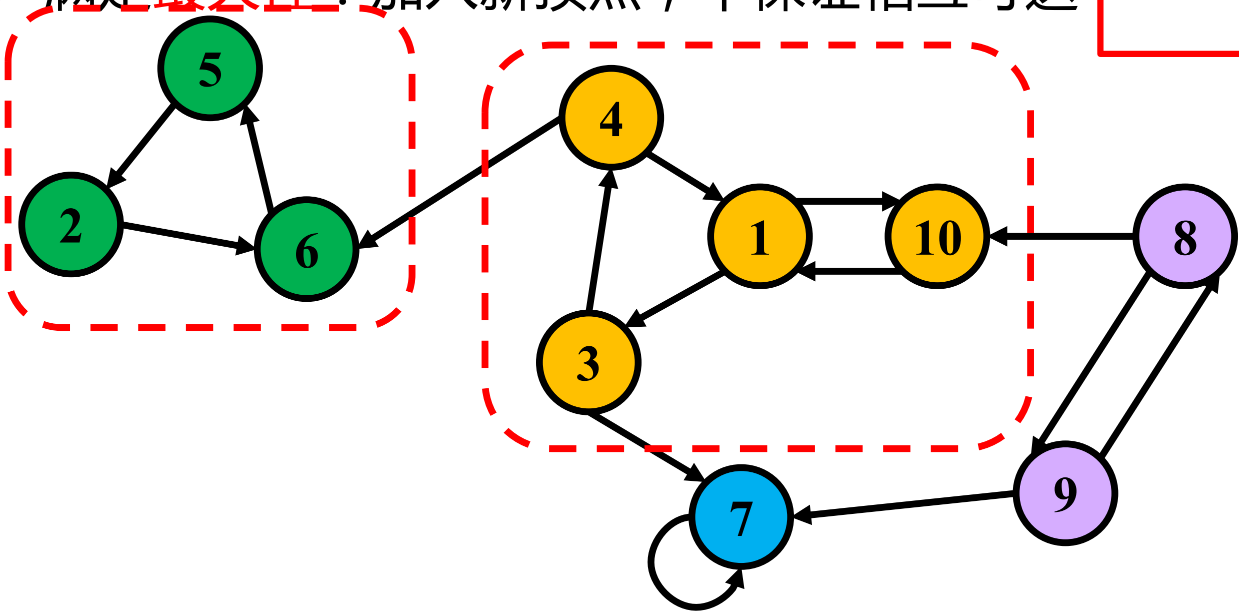
# 问题背景

- 社交圈划分
  - 如何把人群按通话记录划分成不同的社交圈？
- 强连通分量
  - 一个强连通分量是顶点的子集
  - 强连通分量中任意两点相互可达
  - 满足最大性：加入新顶点，不保证相互可达



# 问题背景

- 社交圈划分
  - 如何把人群按通话记录划分成不同的社交圈？
- 强连通分量
  - 一个强连通分量是顶点的子集
  - 强连通分量中任意两点相互可达
  - 满足最大性：加入新顶点，不保证相互可达




- 特性：任意两个强连通分量不相交
  （反证易得：若相交，破坏最大性）

# 问题定义

## 强连通分量

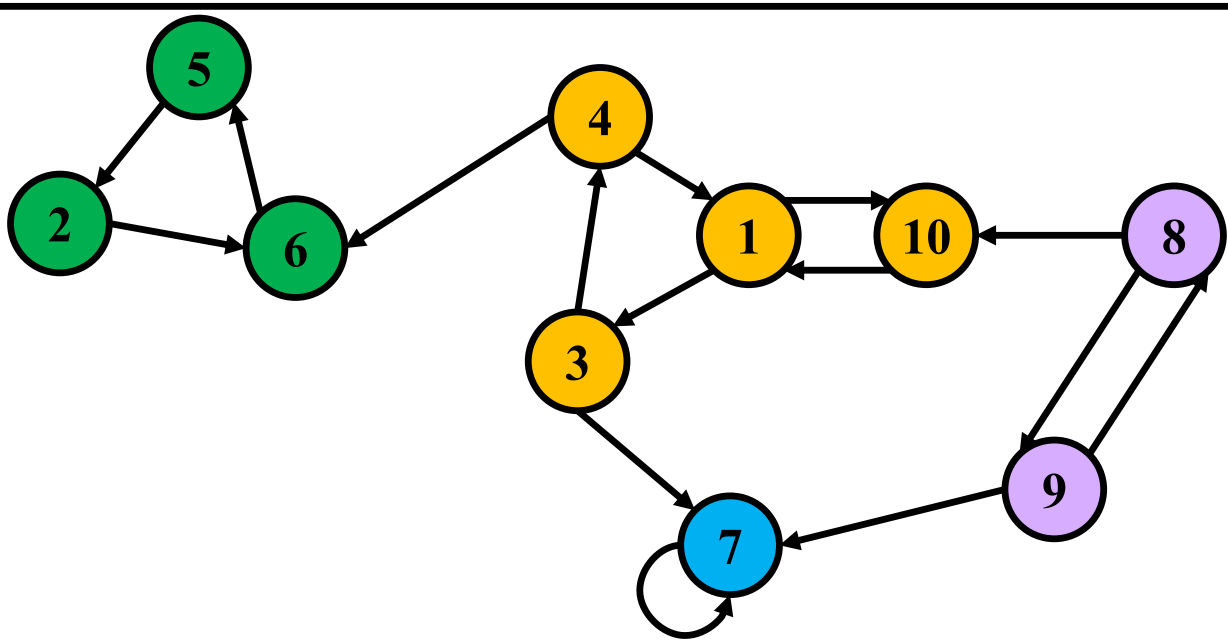
**Strongly Connected Components**

输入

- 有向图$G=<V,E>$

输出

- 图的所有强连通分量$C_1, C_2, \ldots C_n$



# 提纲

问题背景与定义

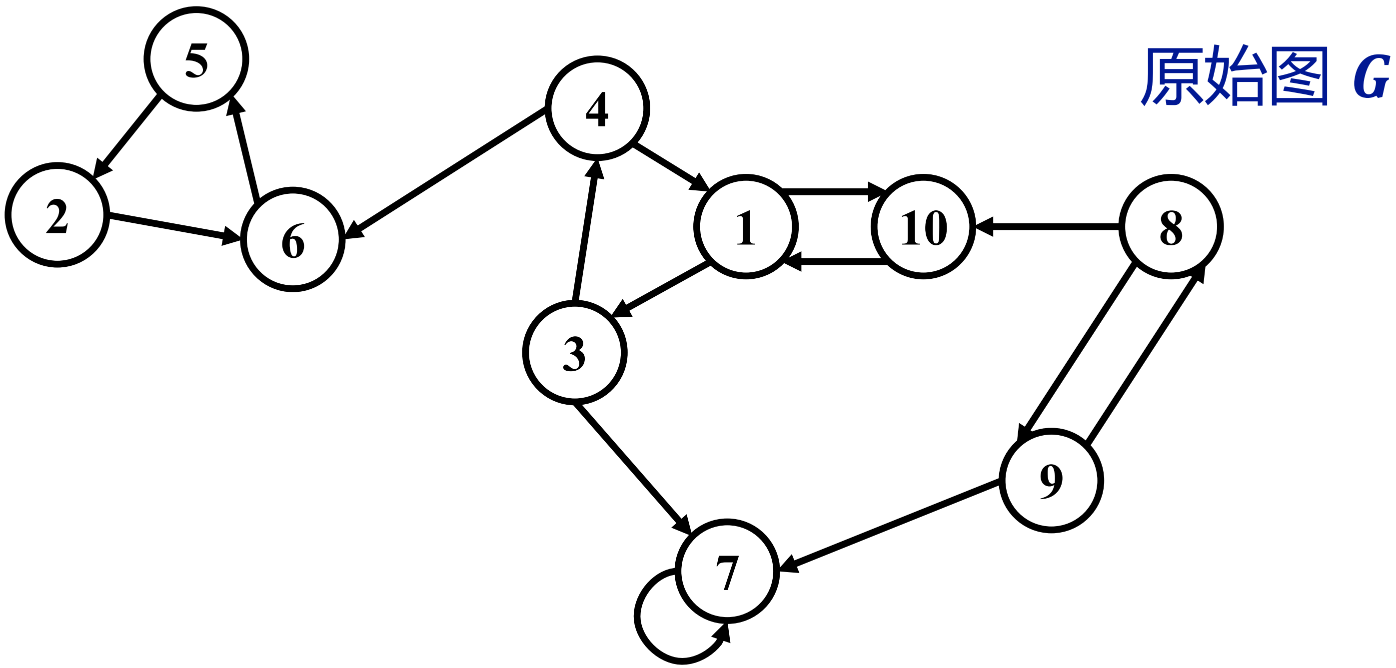
算法框架与实例

伪代码与复杂度

算法正确性证明

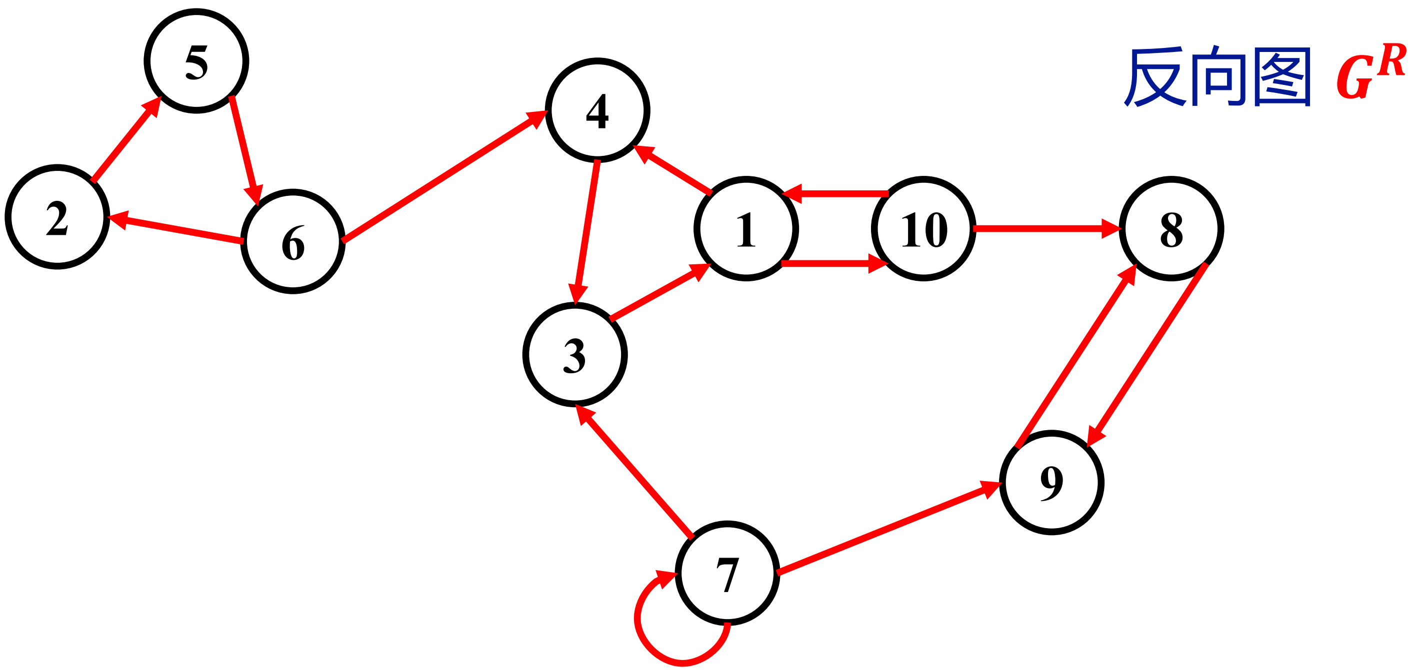
# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$



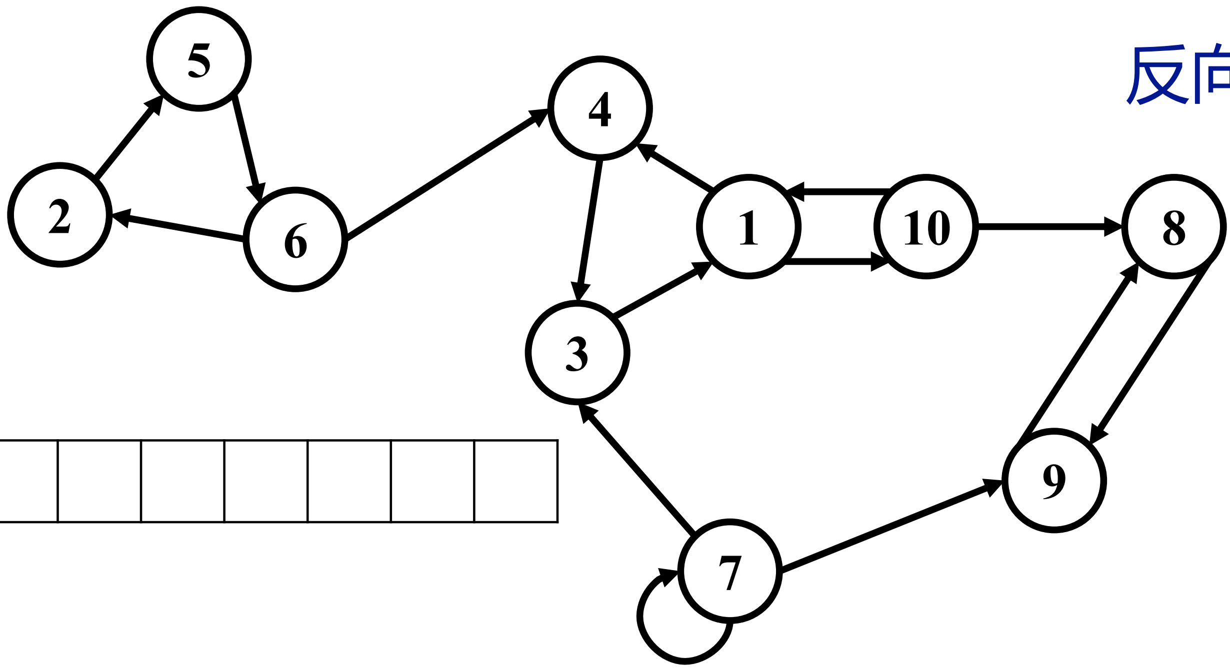
# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$



# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$




反向图 $G^R$

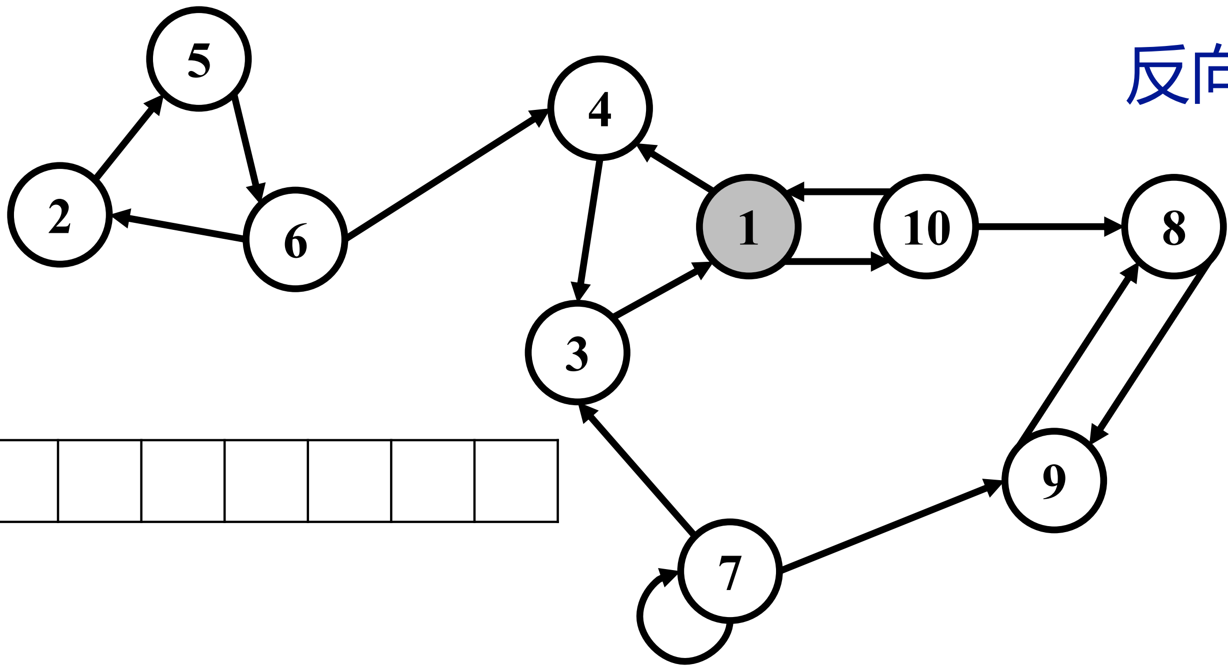
$L$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$

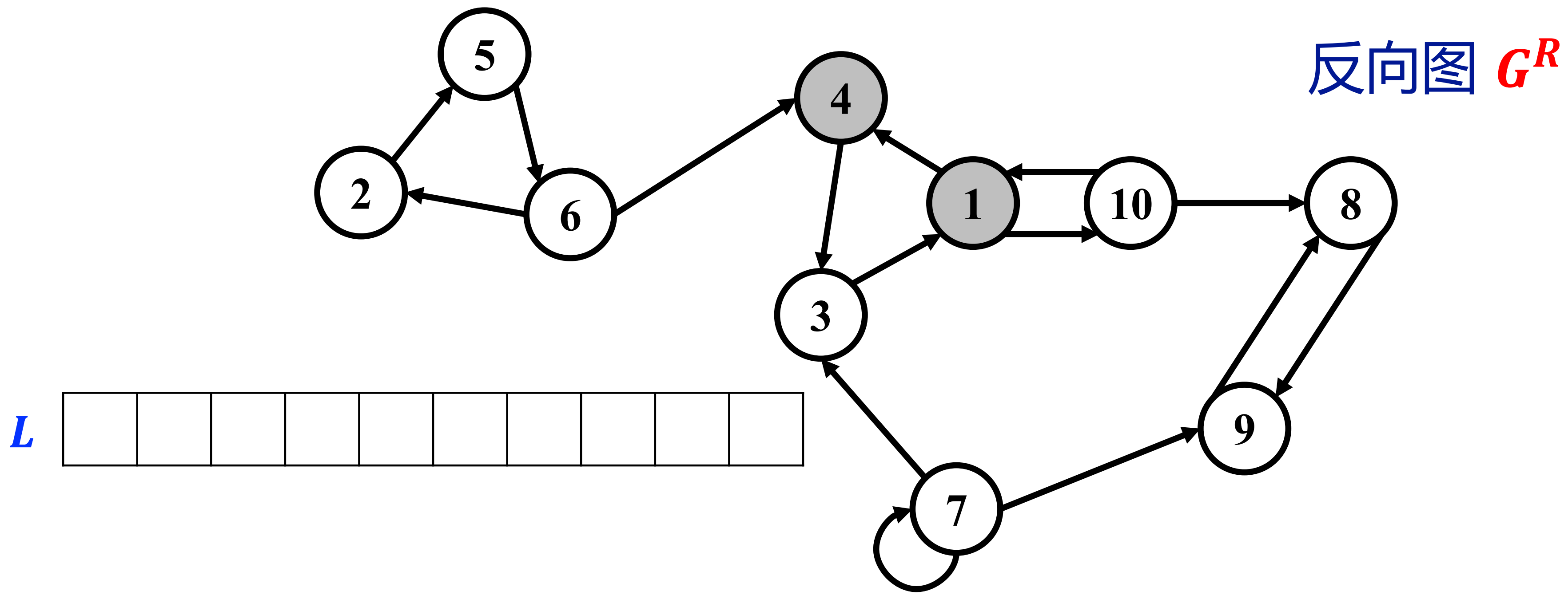
反向图 $G^R$




$L$

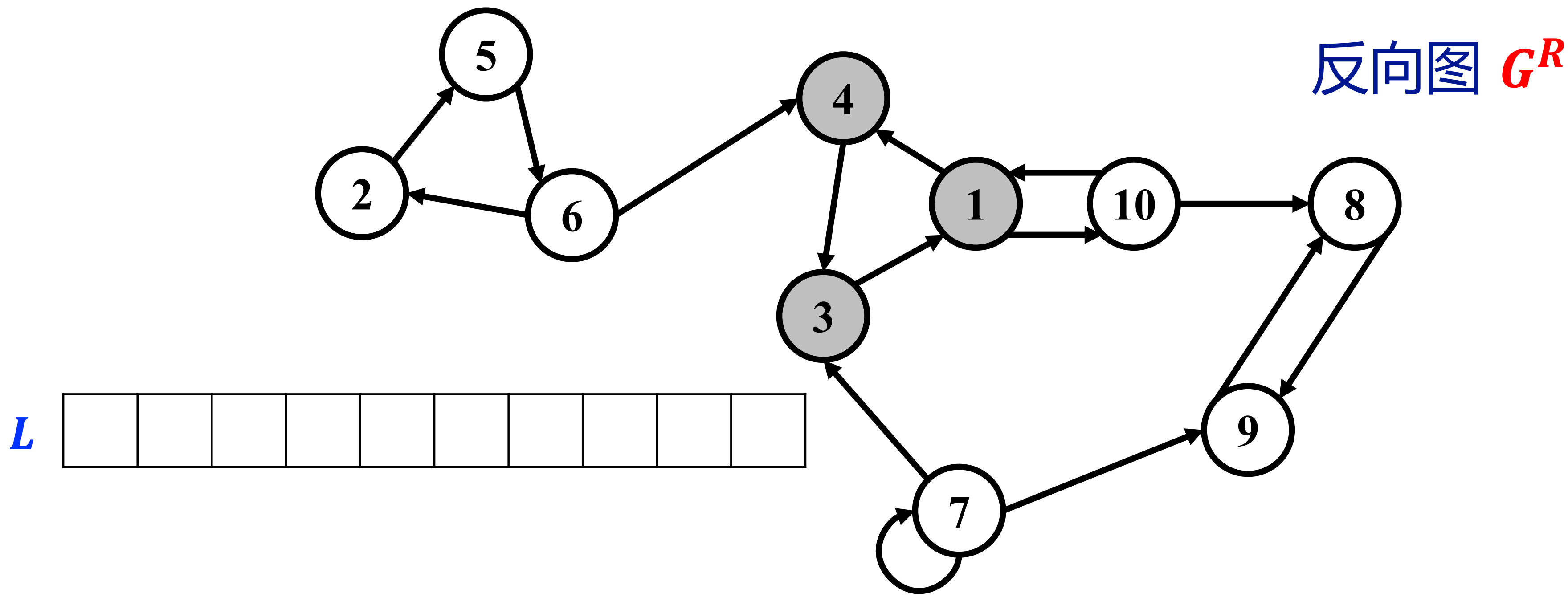
# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$



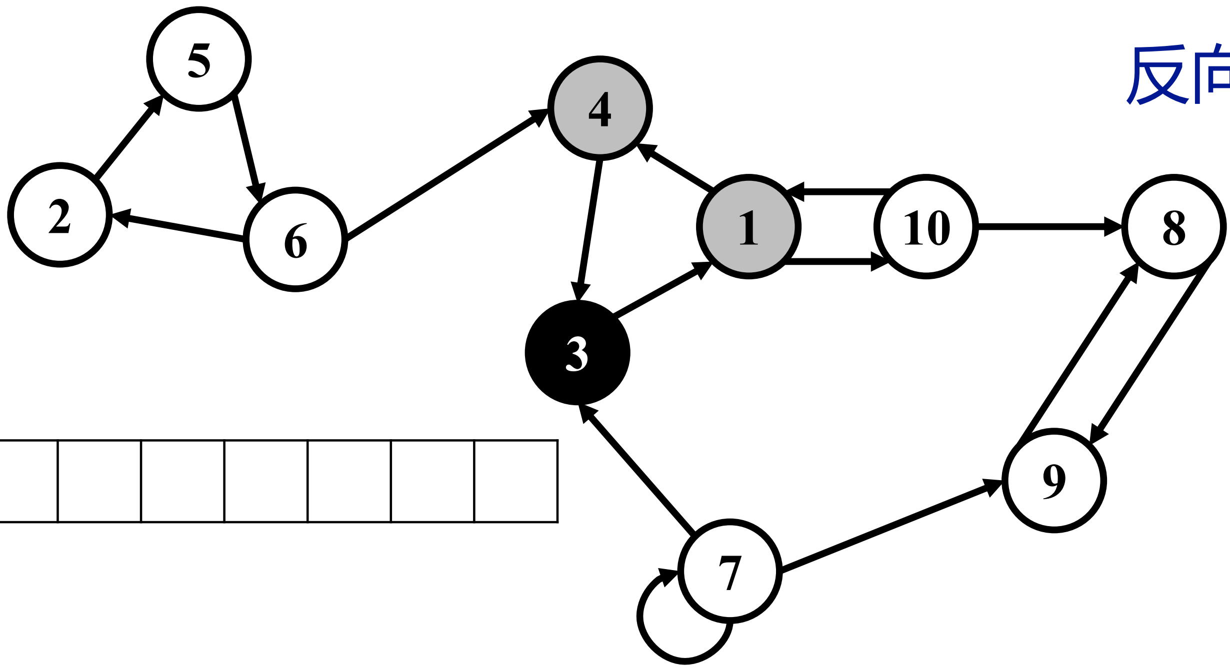
# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$



# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$

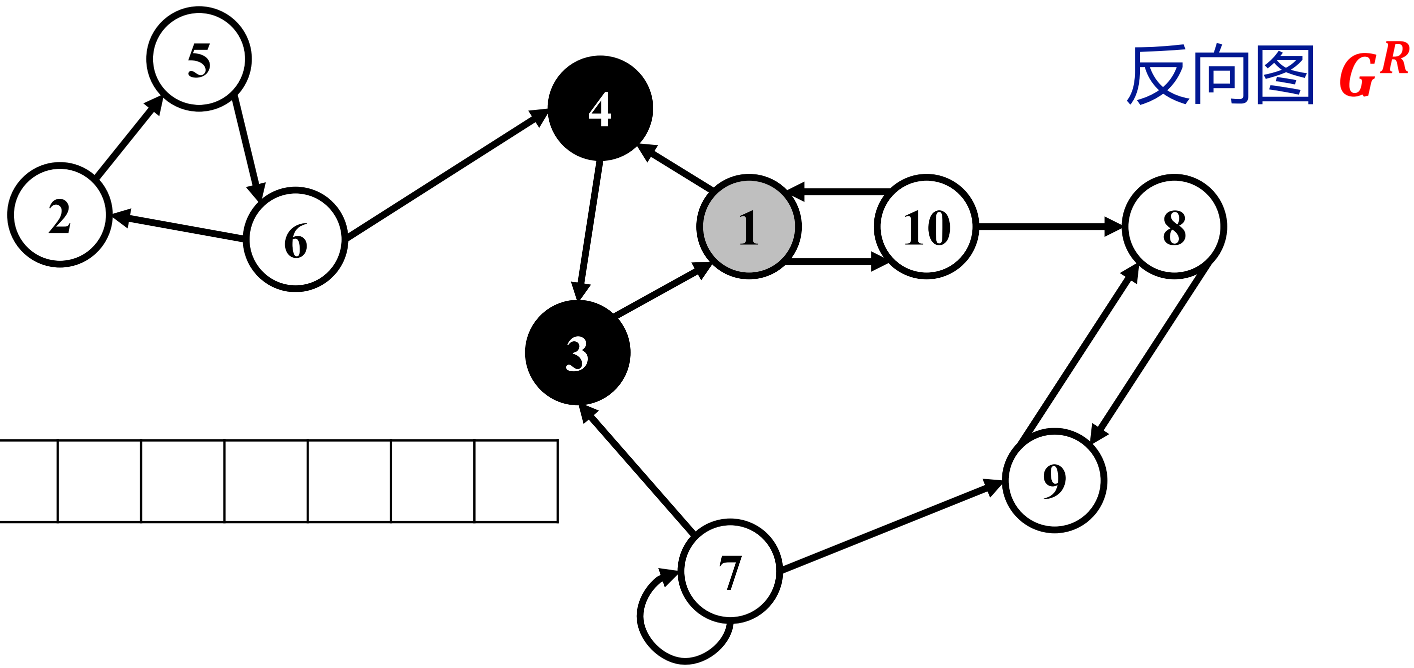



反向图 $G^R$

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $L$ | 3 | | | | | | | | | |

# 算法框架与实例

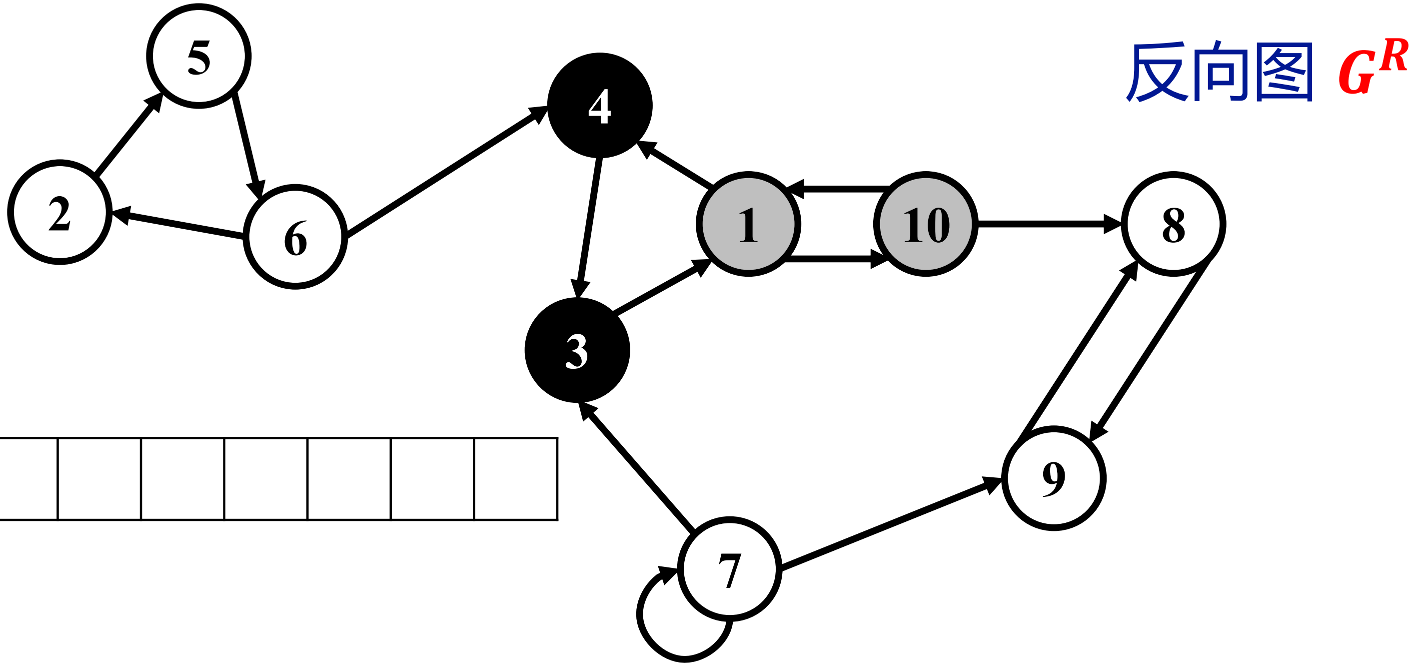
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$




| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $L$ | 3 | 4 | | | | | | | | |

# 算法框架与实例

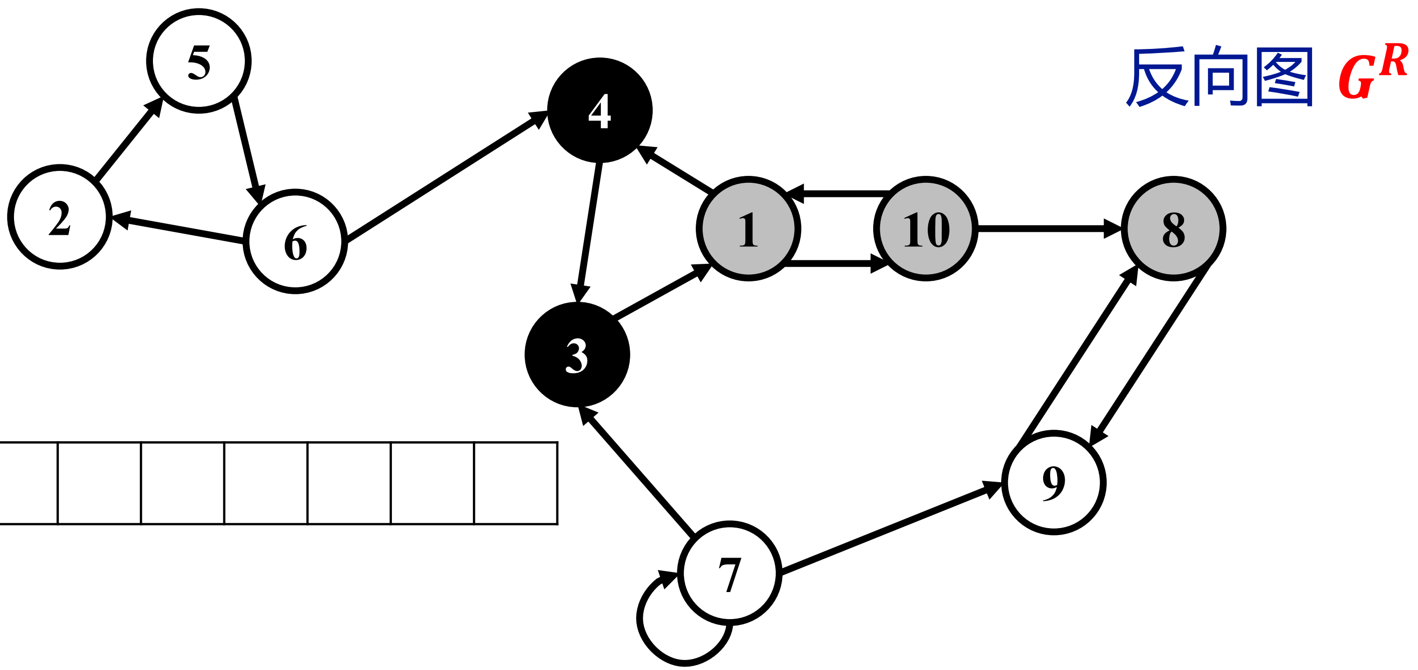
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | | | | | | | | |

# 算法框架与实例

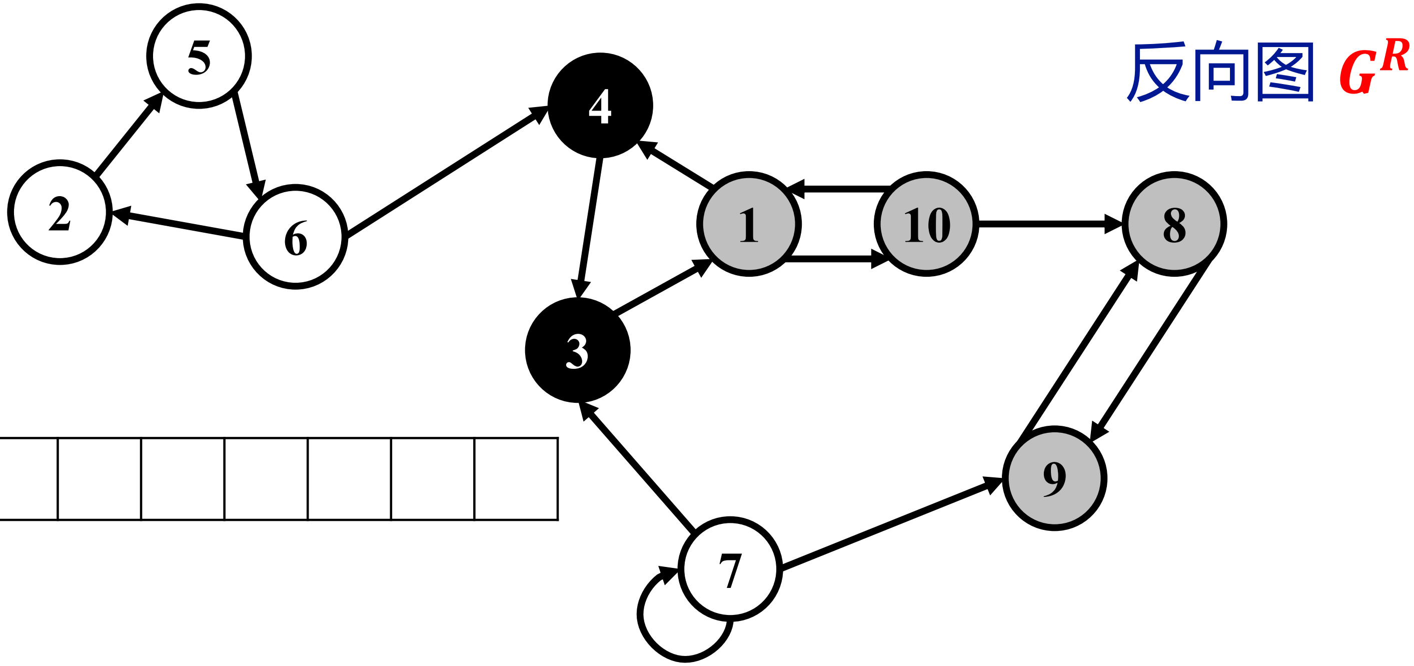
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | | | | | | | | |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$

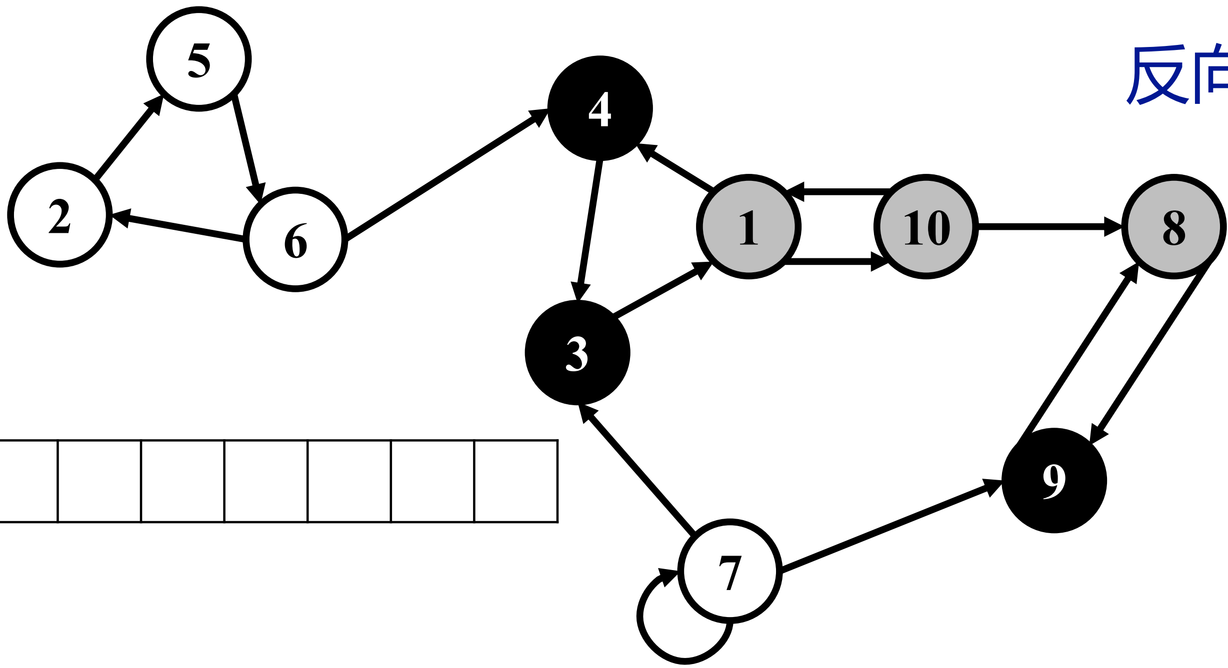



| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | | | | | | | | |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$

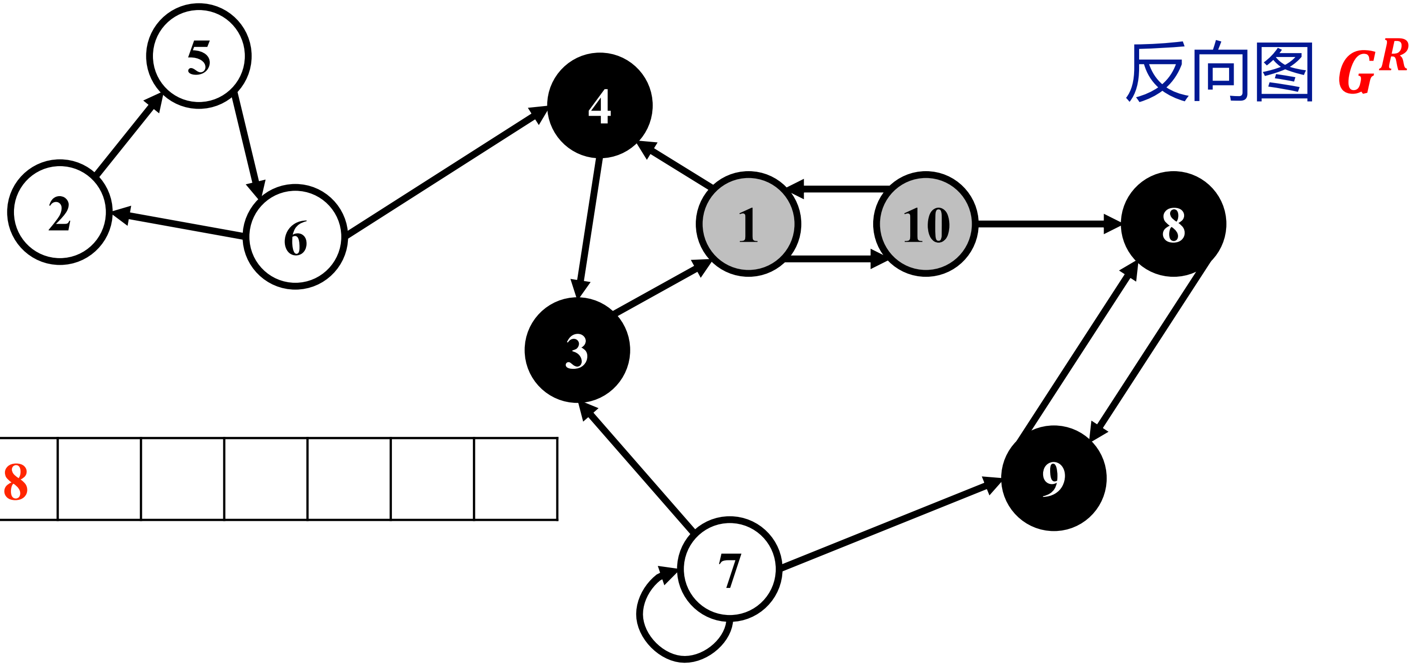
反向图 $G^R$




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | | | | | | | |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$

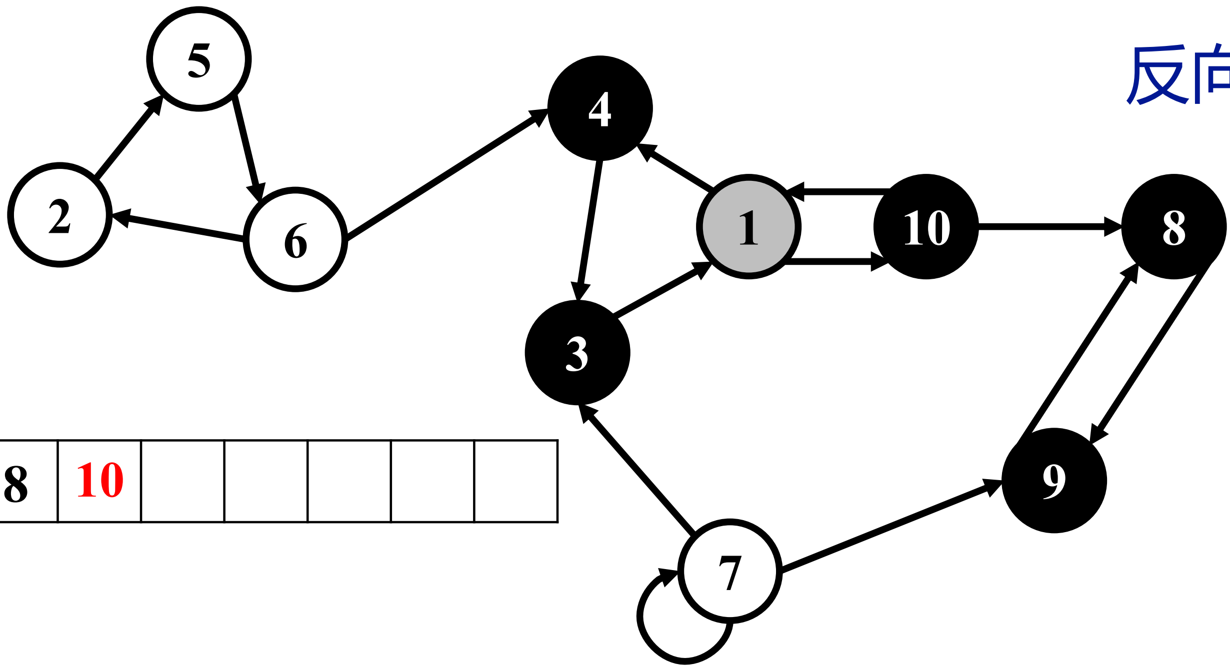



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $L$ | 3 | 4 | 9 | 8 | | | | | | |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$

反向图 $G^R$

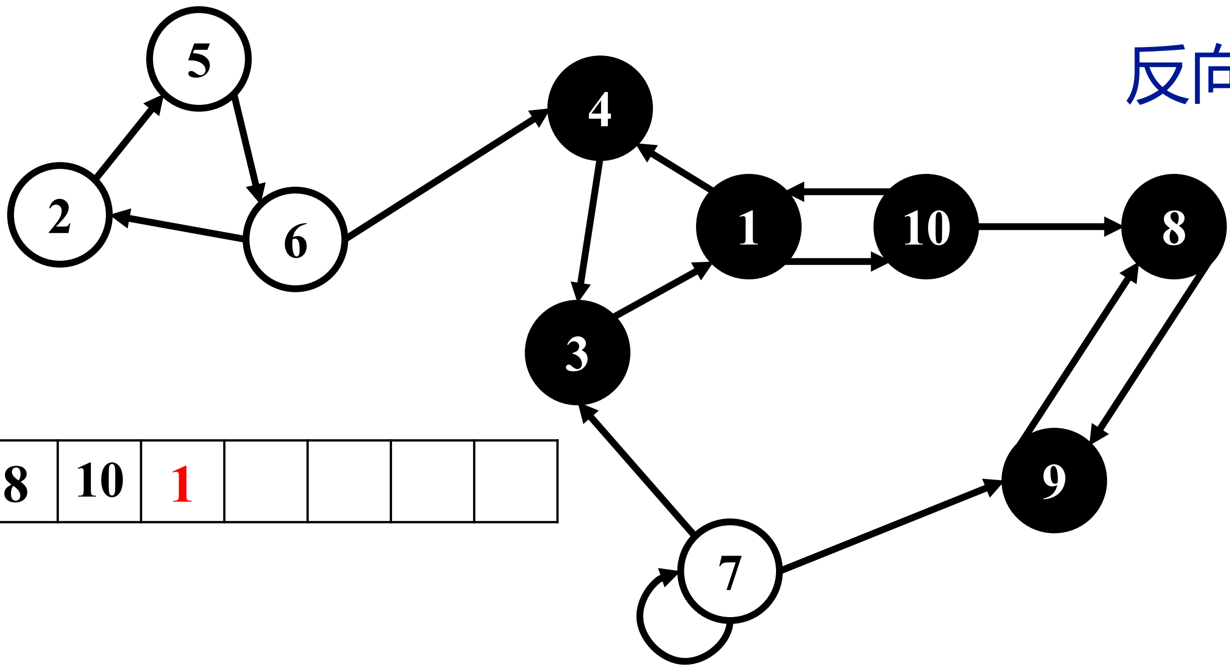



| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | | | | | |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$

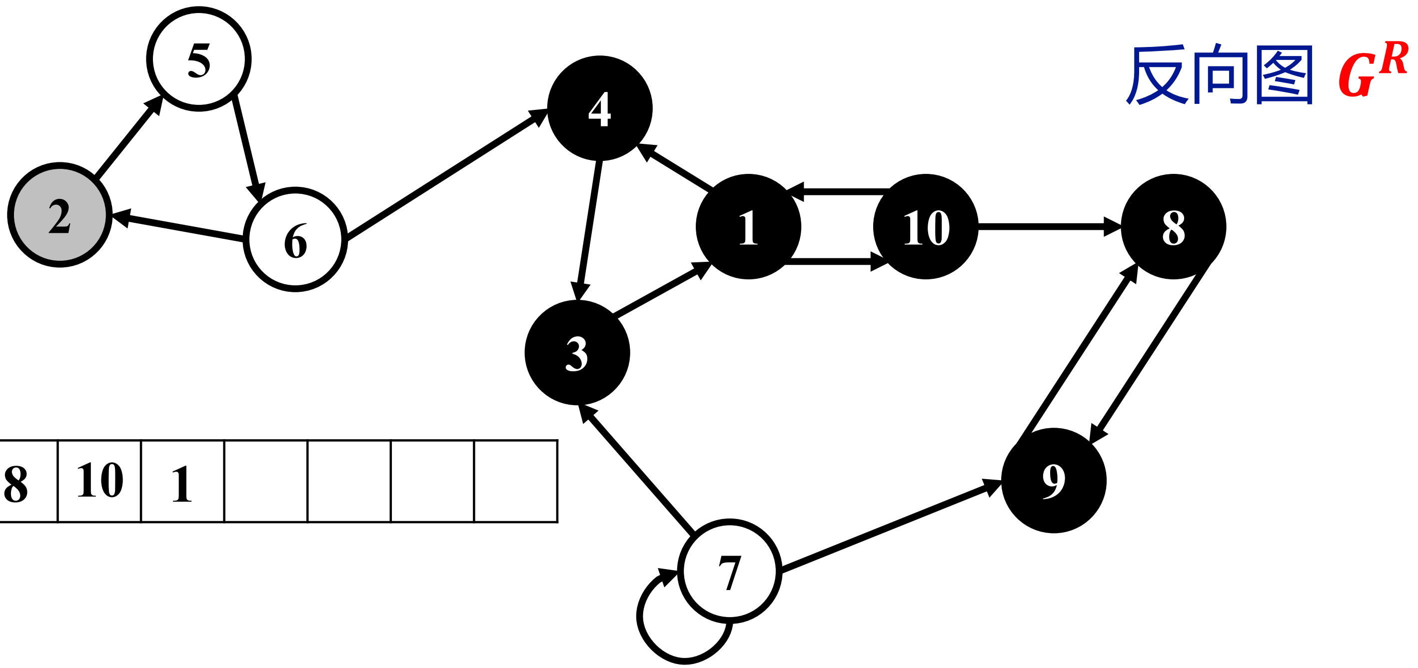
反向图 $G^R$




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | | | | |

# 算法框架与实例

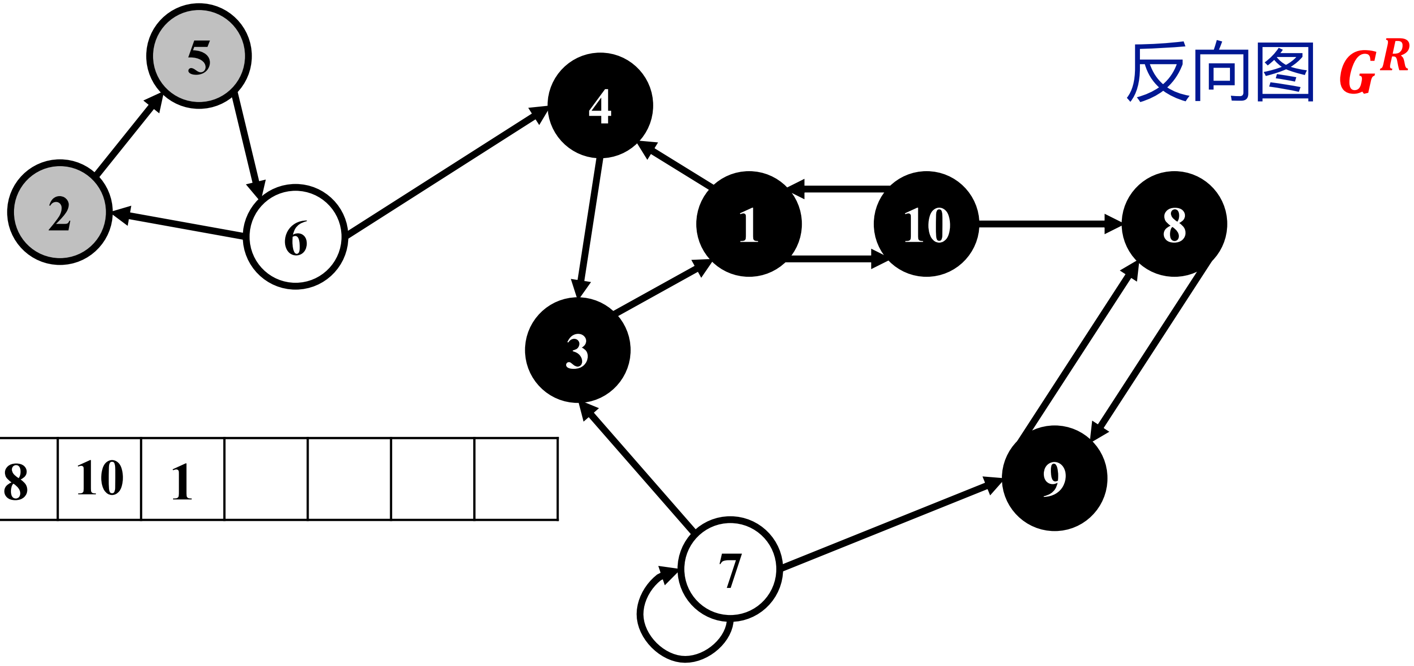
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$




| $L$ | 3 | 4 | 9 | 8 | 10 | 1 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|

# 算法框架与实例

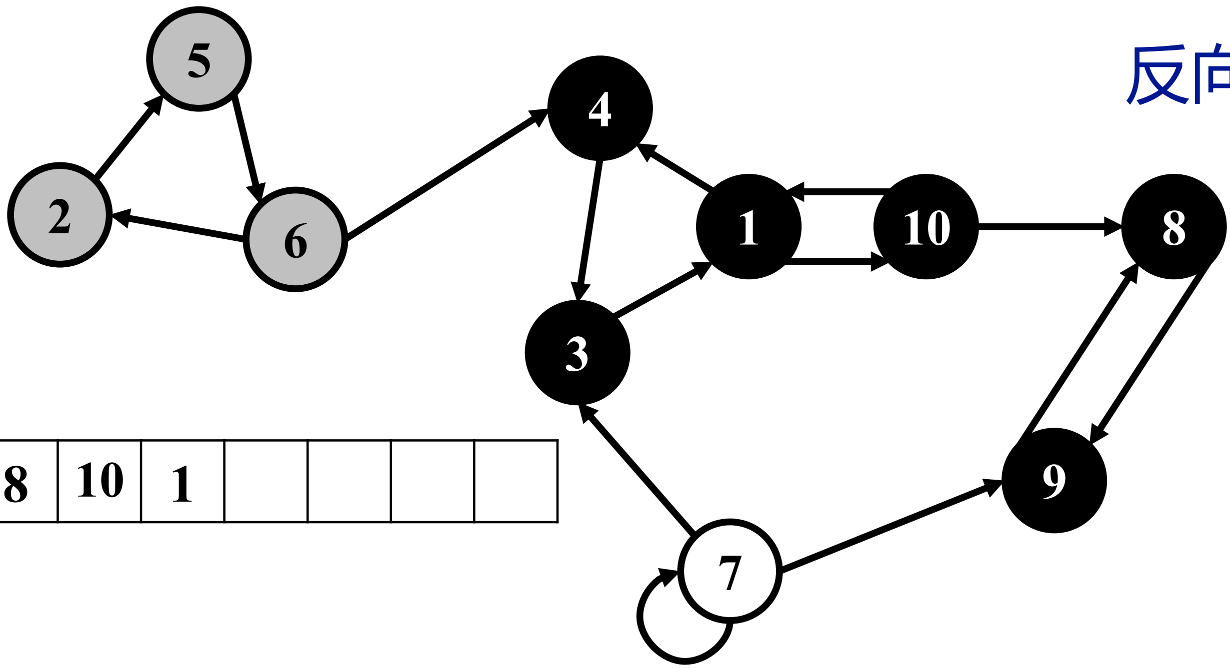
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | | | | |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$

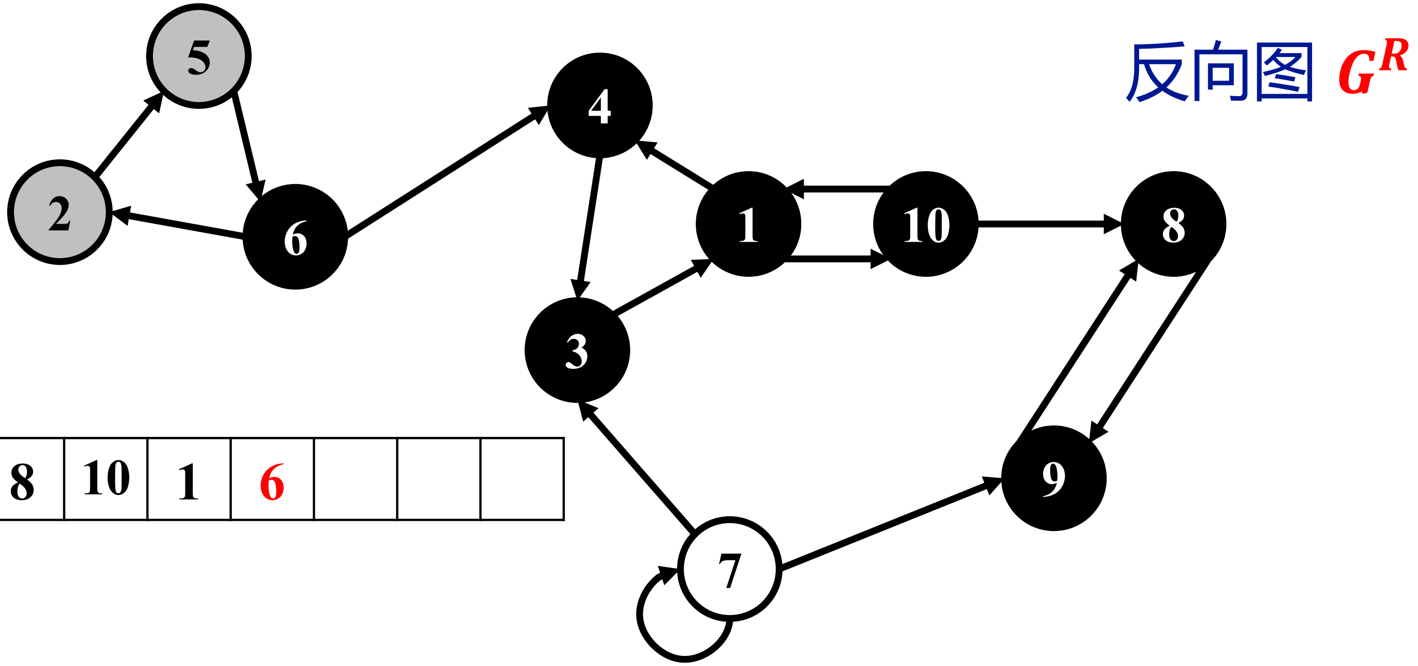



反向图 $G^R$

| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | | | | |

# 算法框架与实例

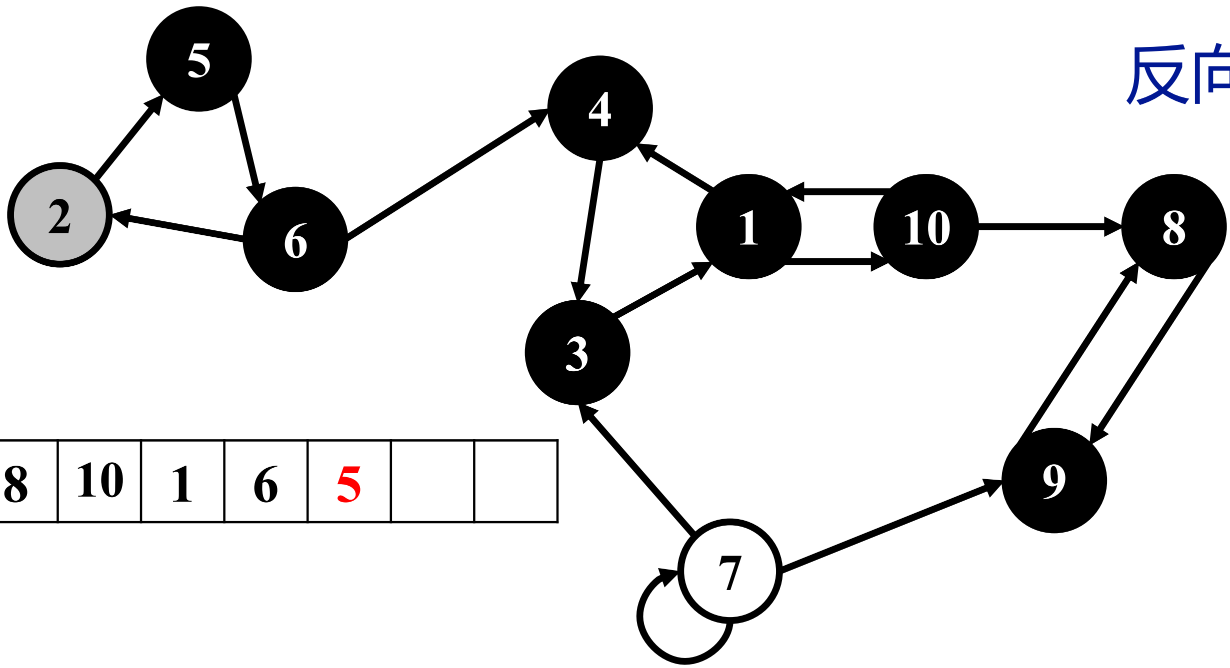
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序$L$




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | | | |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$

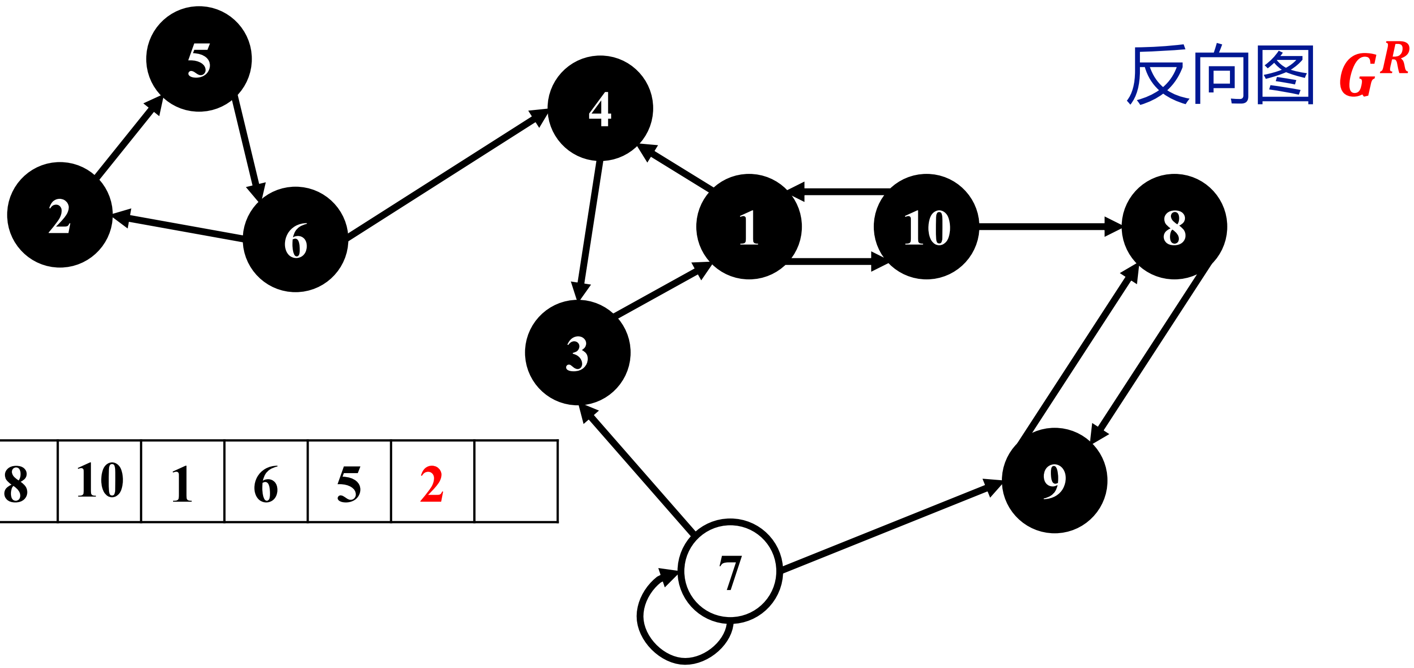


反向图 $G^R$

| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | | |

# 算法框架与实例

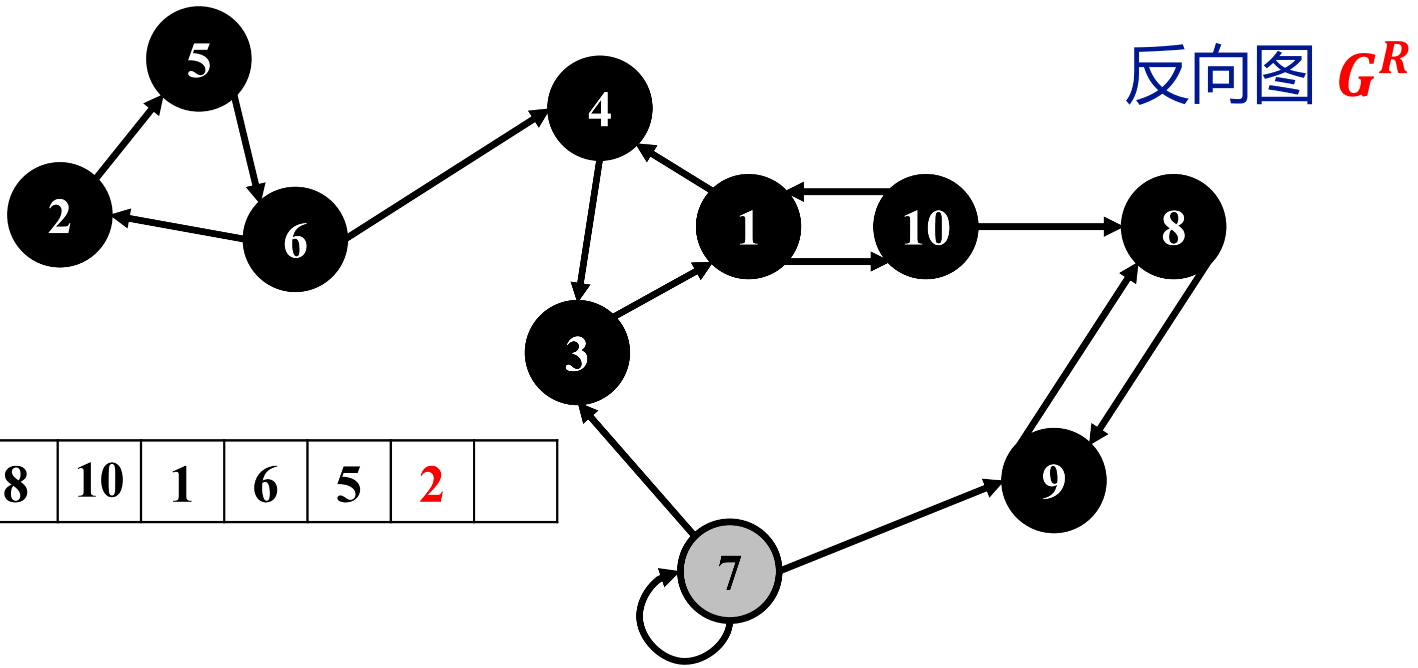
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$




| $L$ | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | |
|---|---|---|---|---|---|---|---|---|---|---|

# 算法框架与实例

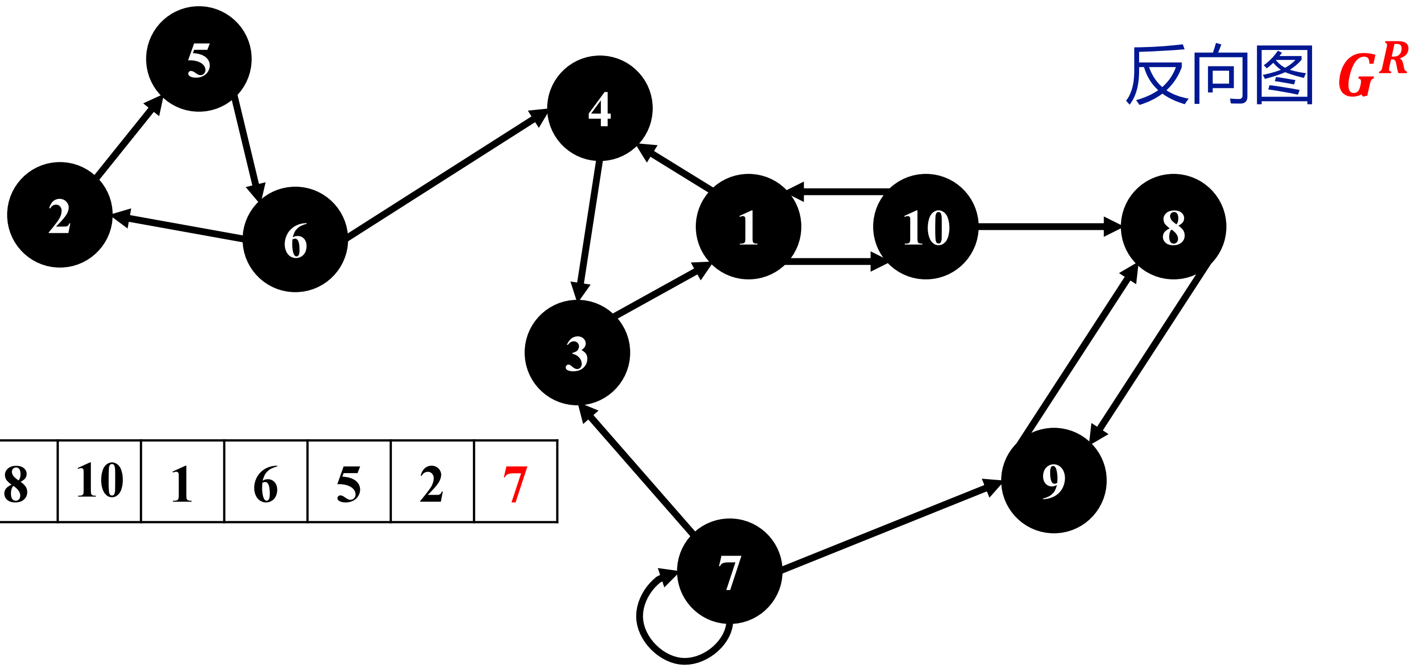
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | |

# 算法框架与实例

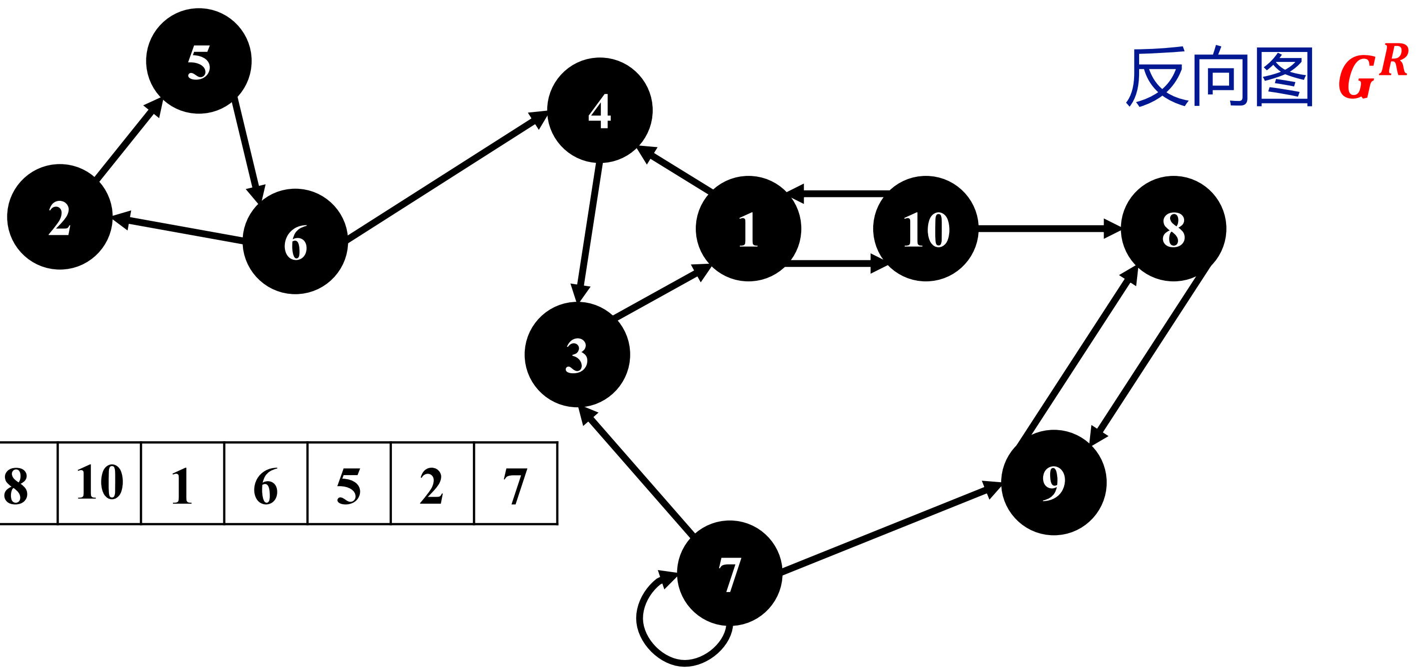
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 算法框架与实例

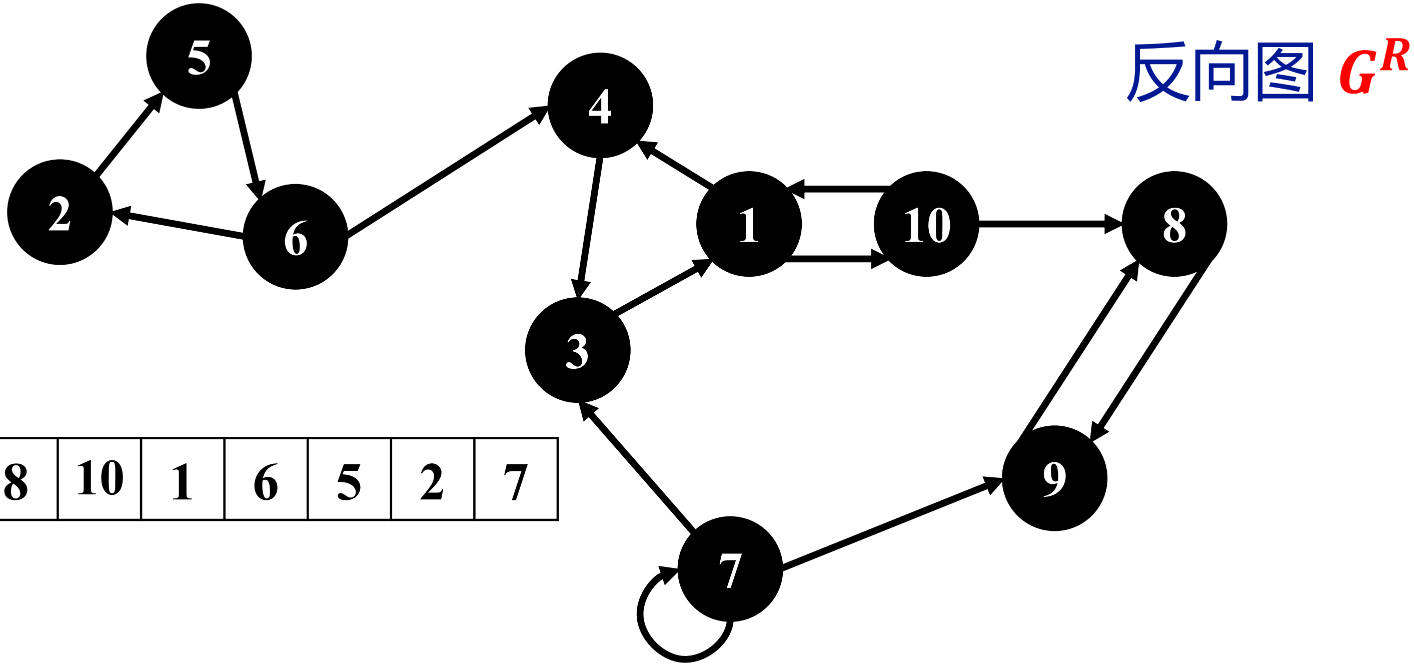
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 算法框架与实例

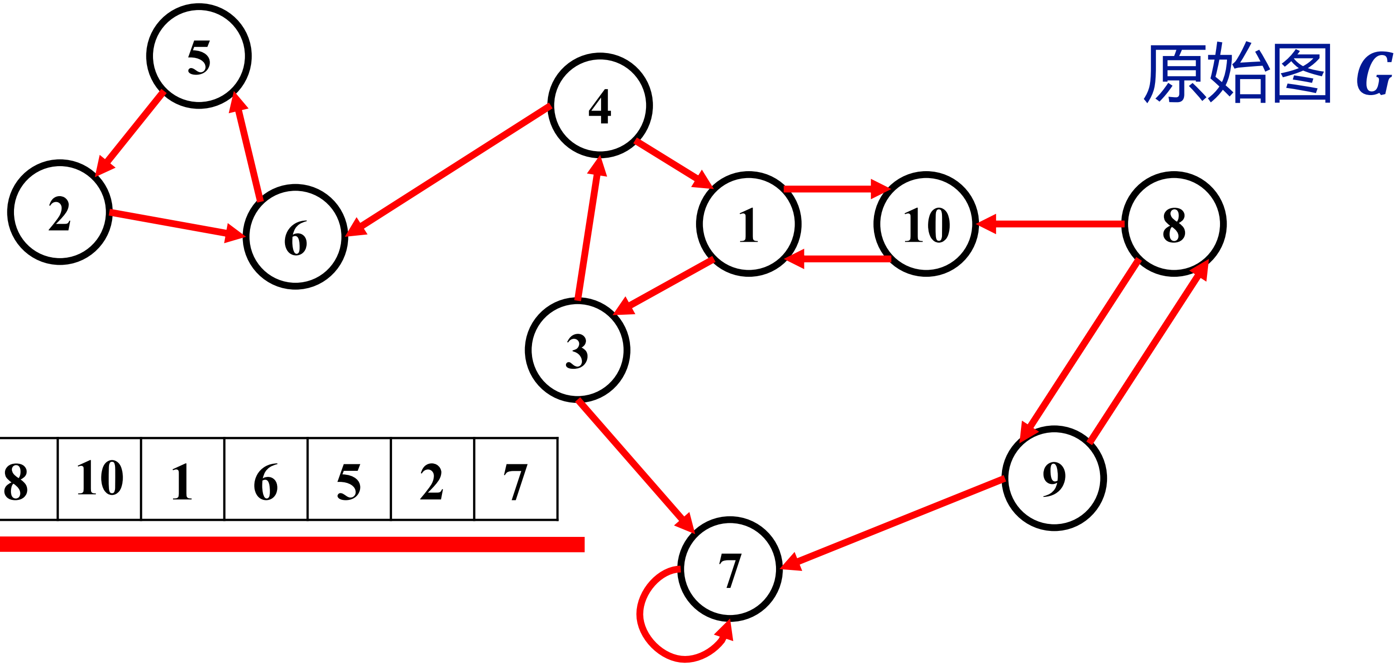
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$
- 步骤3：在$G$上按 $L$ 逆序 执行DFS，得到强连通分量




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 算法框架与实例

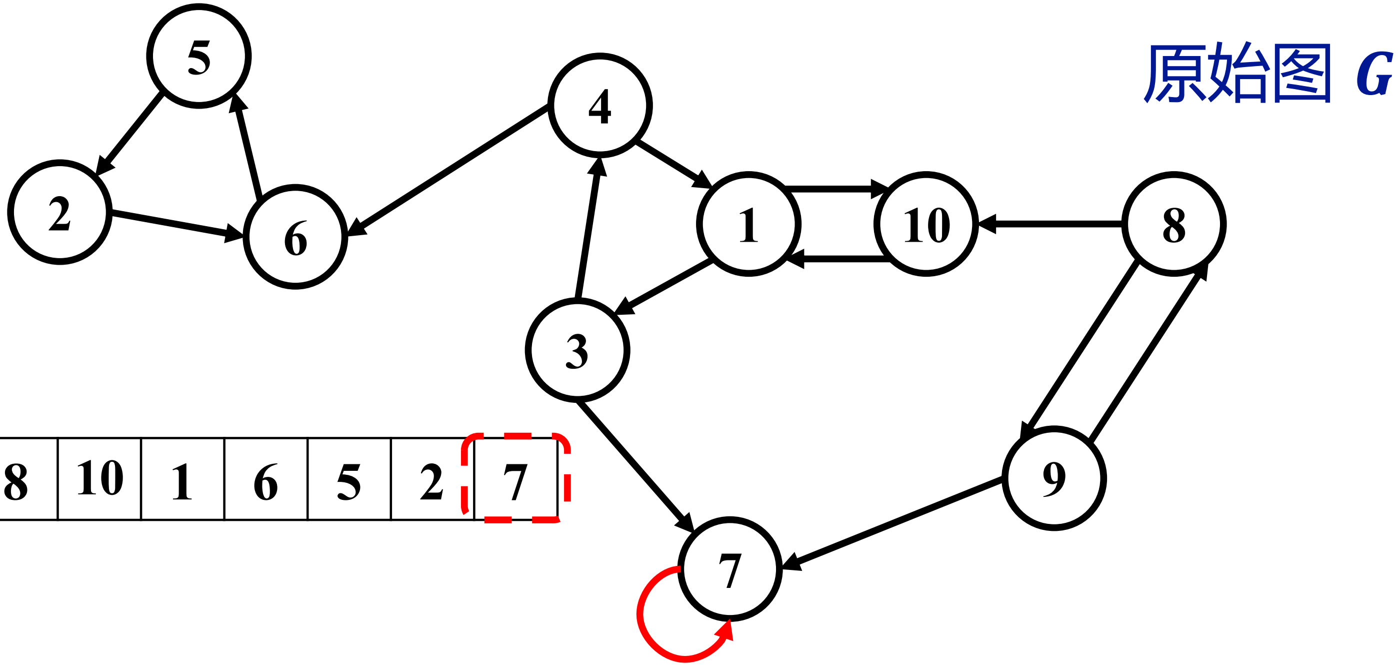
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$
- 步骤3：在$G$上按 $L$ 逆序 执行DFS，得到强连通分量




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 算法框架与实例

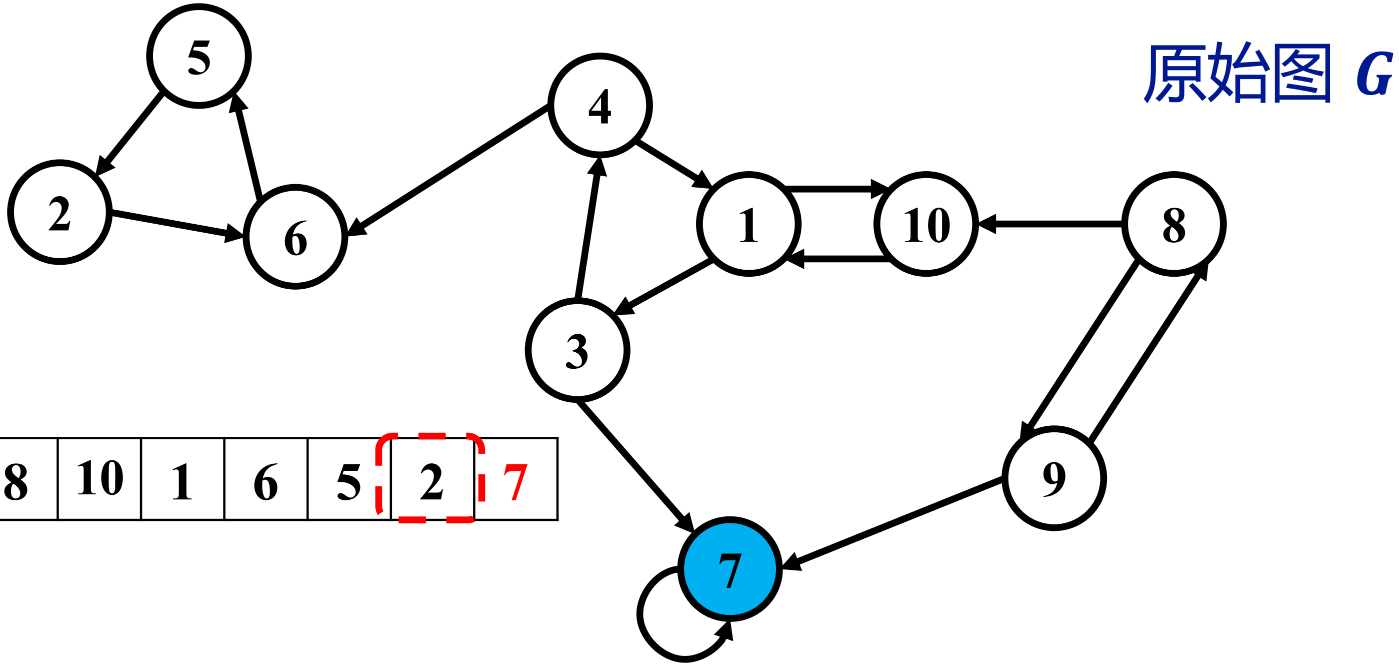
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$
- 步骤3：在$G$上按 $L$ 逆序 执行DFS，得到强连通分量




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 算法框架与实例

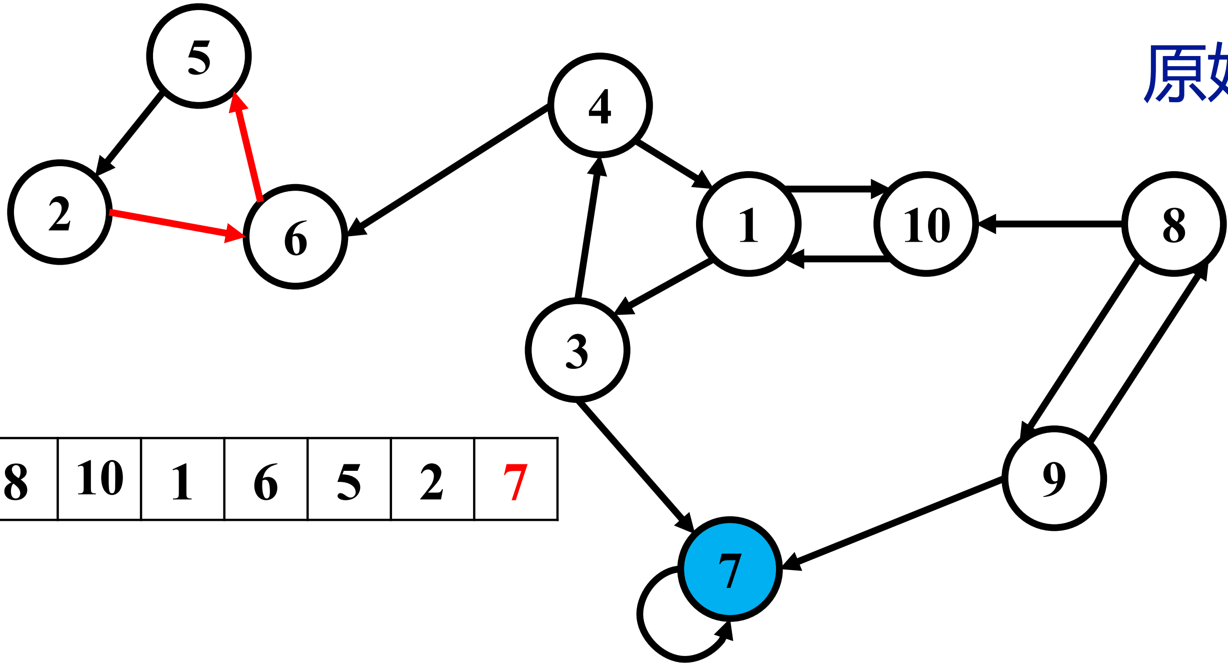
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$
- 步骤3：在$G$上按 $L$ 逆序 执行DFS，得到强连通分量




| $L$ | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$
- 步骤3：在$G$上按 $L$ 逆序 执行DFS，得到强连通分量



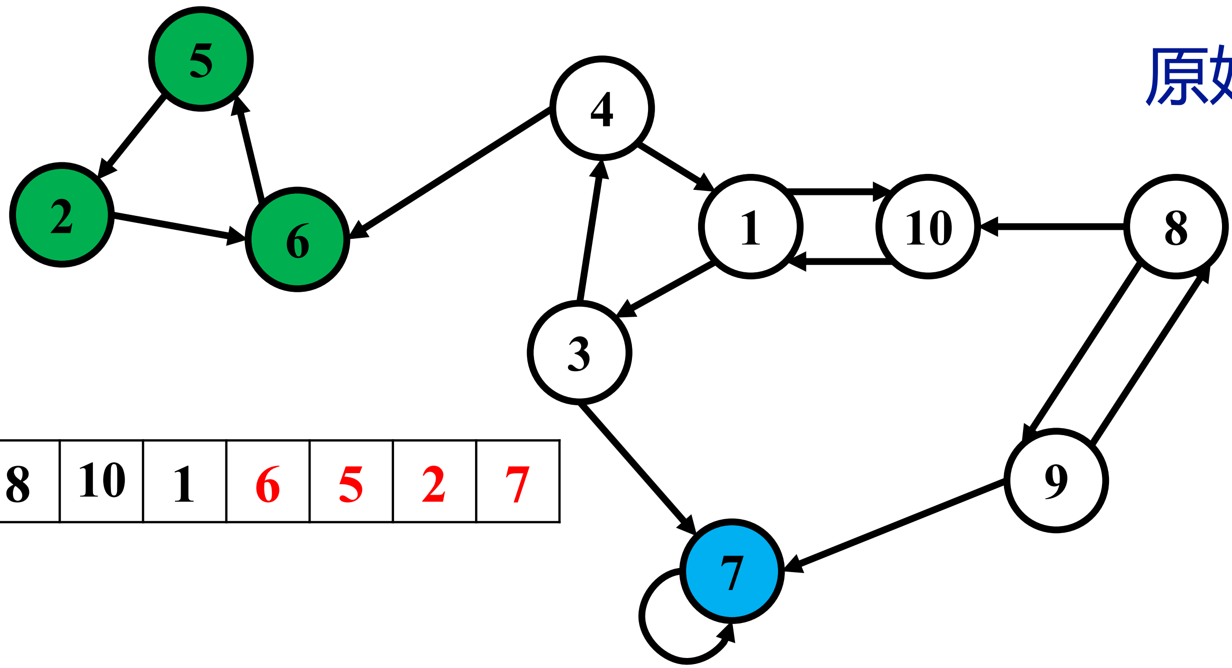

原始图 $G$

| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$
- 步骤3：在$G$上按 $L$ 逆序 执行DFS，得到强连通分量

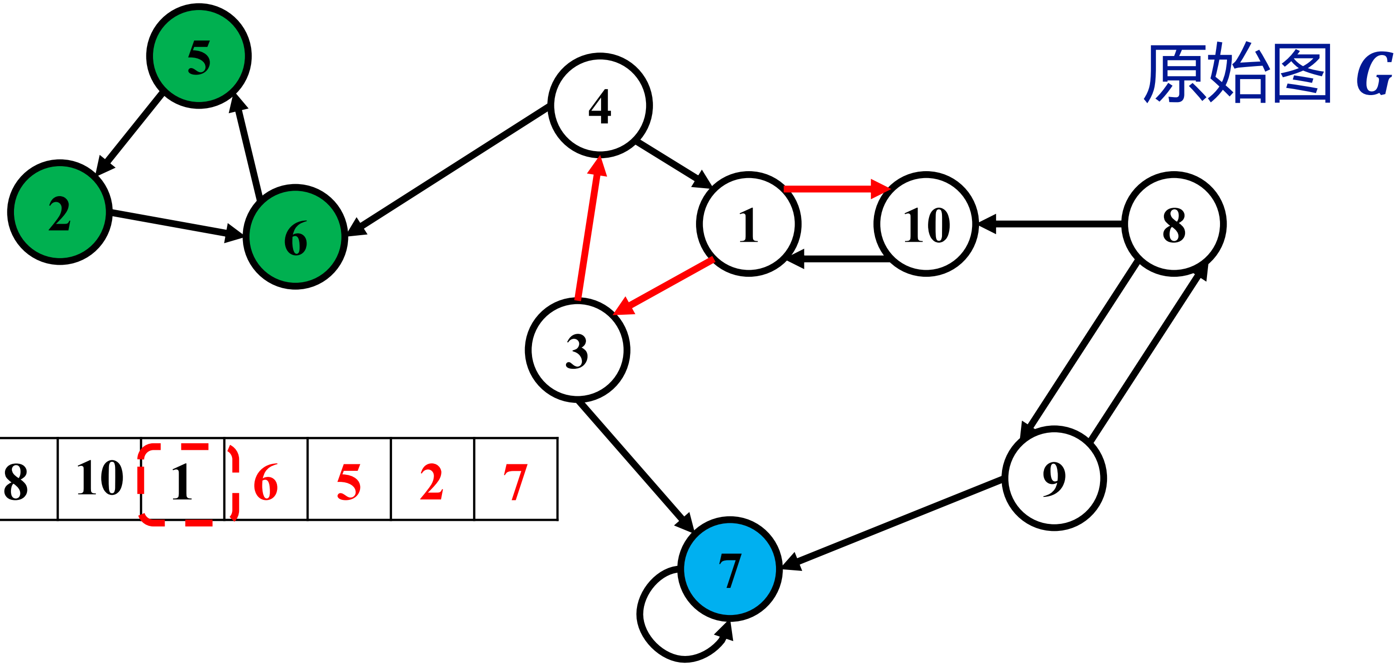
原始图 $G$




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

## 算法框架与实例

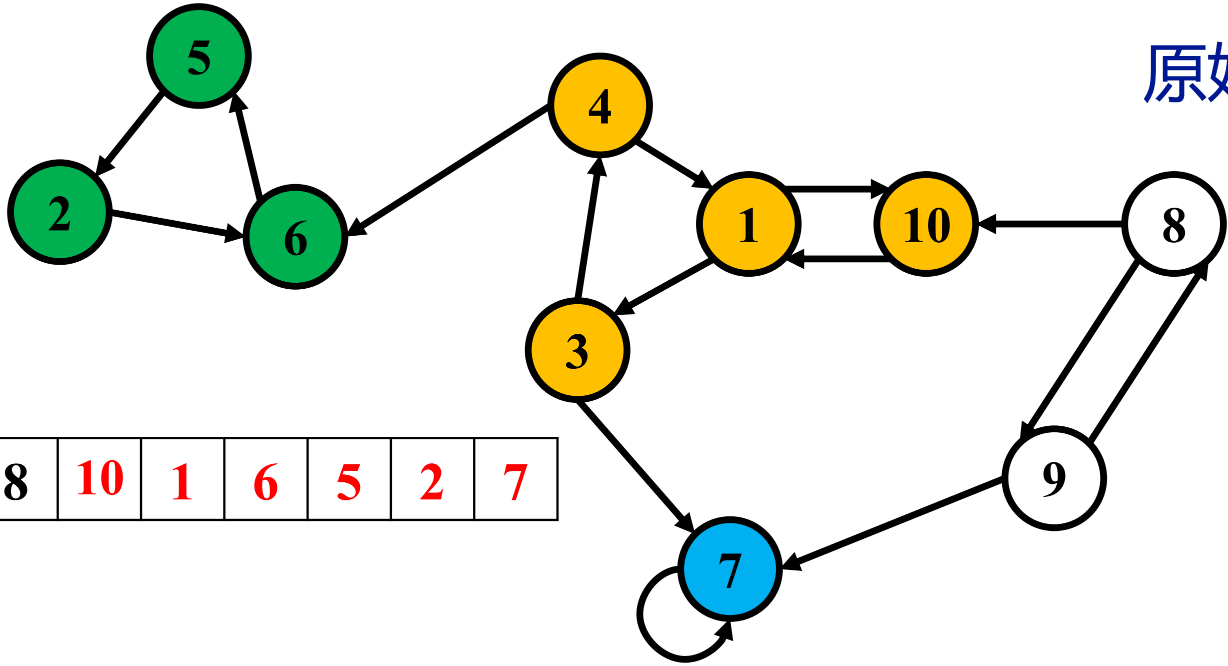
- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$
- 步骤3：在$G$上按 $L$ 逆序 执行DFS，得到强连通分量




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $L$ | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$
- 步骤3：在$G$上按 $L$ 逆序 执行DFS，得到强连通分量

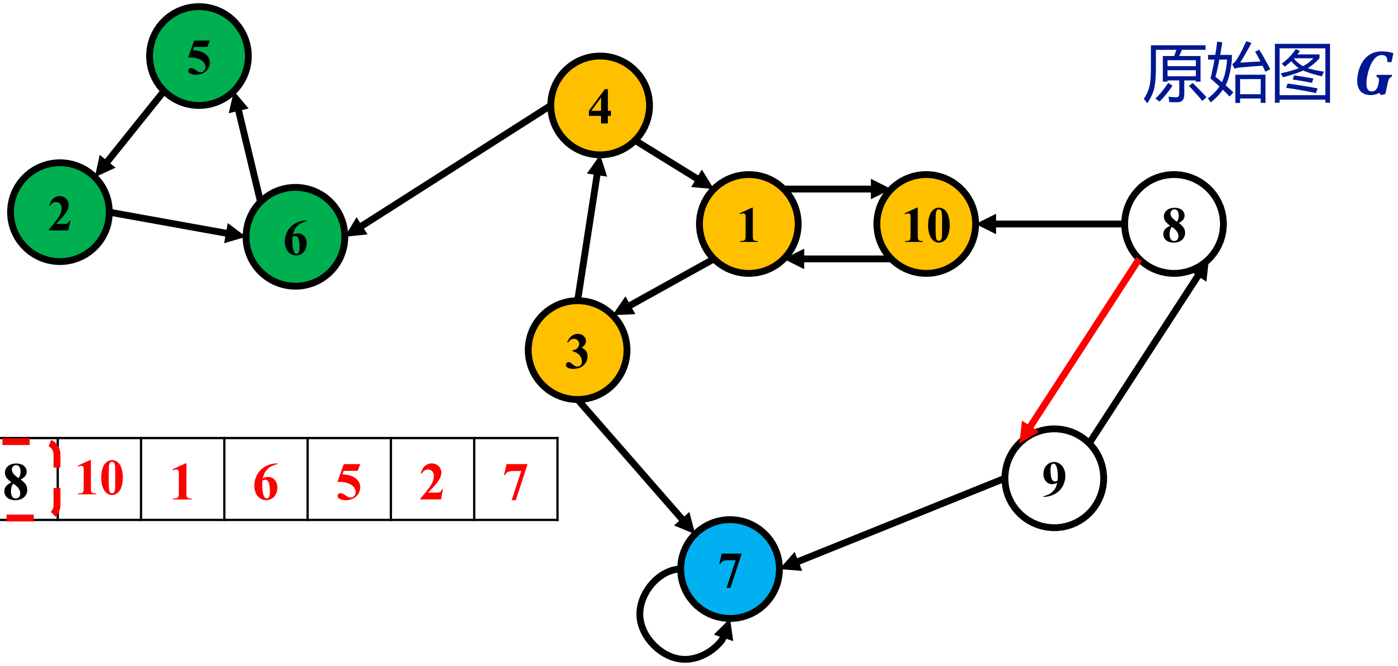



原始图 $G$

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $L$ | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$
- 步骤3：在$G$上按 $L$ 逆序 执行DFS，得到强连通分量

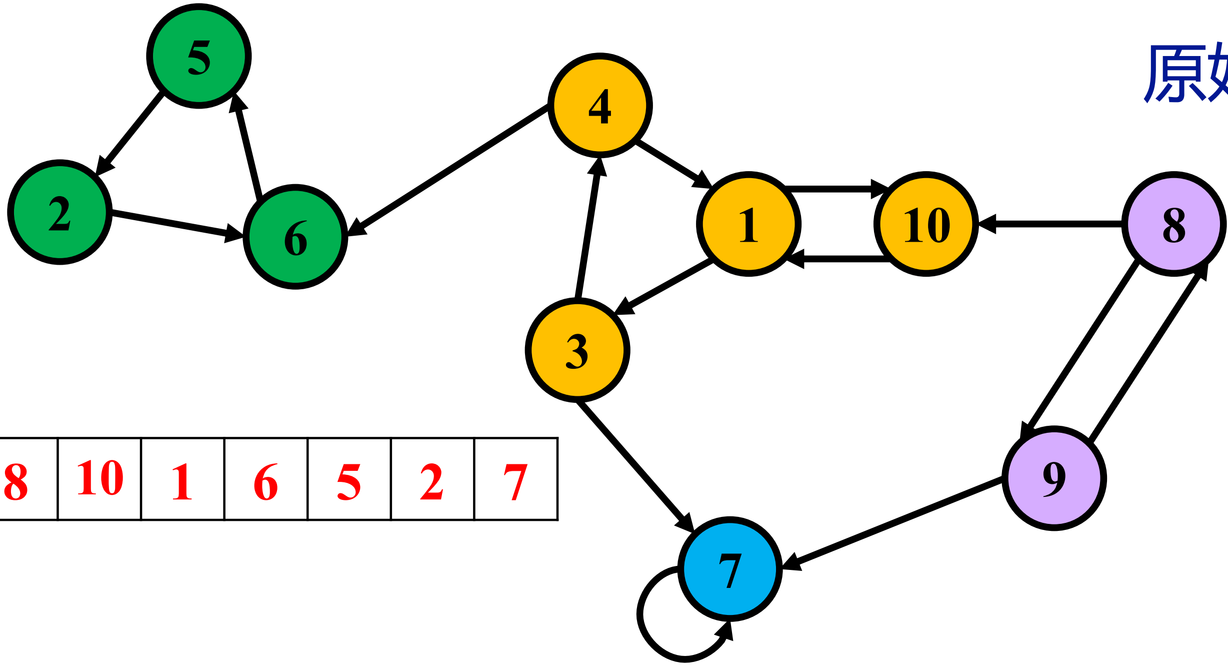



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $L$ | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 算法框架与实例

- 步骤1：把边反向，得到反向图$G^R$
- 步骤2：在$G^R$上执行DFS，得到顶点完成时刻顺序 $L$
- 步骤3：在$G$上按 $L$ 逆序 执行DFS，得到强连通分量

原始图 $G$




| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $L$ | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 提纲

问题背景与定义

算法框架与实例

伪代码与复杂度

算法正确性证明

# 伪代码

- **Strongly-Connected-Component(*G*)**

**输入: 图** $G$
**输出: 强连通分量**

$R \leftarrow \{\}$
$G^R \leftarrow G.reverse()$ 构造反向图
$L \leftarrow \mathrm{DFS}(G^R)$
$color[1..V] \leftarrow WHITE$
**for** $i \leftarrow L.length()$ $downto$ 1 **do**
  $u \leftarrow L[i]$
  **if** $color[u] = WHITE$ **then**
    $L_{scc} \leftarrow \text{DFS-Visit}(G, u)$
    $R \leftarrow R \cup set(L_{scc})$
  **end**
**end**
**return** $R$

# 伪代码

- **Strongly-Connected-Component(*G*)**

**输入:** **图** $G$
**输出:** **强连通分量**

$R \leftarrow \{\}$
$G^R \leftarrow G.reverse()$
$L \leftarrow \text{DFS}(G^R)$
$color[1..V] \leftarrow WHITE$
**for** $i \leftarrow L.length()$ $downto$ 1 **do**
  $u \leftarrow L[i]$
  **if** $color[u] = WHITE$ **then**
    $L_{scc} \leftarrow \text{DFS-Visit}(G, u)$
    $R \leftarrow R \cup set(L_{scc})$
  **end**
**end**
**return** $R$

在反向图上执行**DFS**

# 伪代码

- **Strongly-Connected-Component(*G*)**

```
输入: 图 G
输出: 强连通分量
R ← {}
G^R ← G.reverse()
L ← DFS(G^R)
color[1..V] ← WHITE
for i ← L.length() downto 1 do
    u ← L[i]
    if color[u] = WHITE then
        L_scc ← DFS-Visit(G, u)
        R ← R ∪ set(L_scc)
    end
end
return R
```

按 ***L*** 逆序在原图执行 **DFS**

# 伪代码

- **Strongly-Connected-Component(*G*)**

**输入:** 图 $G$
**输出:** 强连通分量
$R \leftarrow \{\}$
$G^R \leftarrow G.reverse()$
$L \leftarrow \text{DFS}(G^R)$
$color[1..V] \leftarrow WHITE$
**for** $i \leftarrow L.length()$ $downto$ 1 **do**
    $u \leftarrow L[i]$
    **if** $color[u] = WHITE$ **then**
        $L_{scc} \leftarrow \text{DFS-Visit}(G, u)$
        $R \leftarrow R \cup set(L_{scc})$
    **end**
**end**
**return** $R$

如何在搜索过程中得到$L$？

# 伪代码

- **DFS($G$)**

```
输入: 图 G
新建数组 color[1..V], L[1..V]
for v ∈ V do
|   color[v] ← WHITE
end
for v ∈ V do
|   if color[v] = WHITE then
|   |   L′ ← DFS-Visit(G, v)
|   |   向L结尾追加L′
|   end
end
return L
```

- **DFS-Visit($G, v$)**

```
输入: 图 G, 顶点 v
输出: 按完成时刻从早到晚排列的顶点 L
color[v] ← GRAY
for w ∈ G.Adj[v] do
|   if color[w] = WHITE then
|   |   L ← DFS-Visit(G, w)
|   end
end
color[v] ← BLACK
向L结尾追加顶点v
return L
```

顶点按完成时刻排列

# 复杂度分析

- **Strongly-Connected-Component($G$)**

**输入:** 图 $G$
**输出:** 强连通分量

$R \leftarrow \{\}$

$G^R \leftarrow G.reverse()$ $O(|V|+|E|)$

$L \leftarrow \text{DFS}(G^R)$ $O(|V|+|E|)$ 第一次深度优先搜索

$color[1..V] \leftarrow WHITE$

**for** $i \leftarrow L.length()$ $downto$ 1 **do**
  $u \leftarrow L[i]$
  **if** $color[u] = WHITE$ **then**
    $L_{scc} \leftarrow \text{DFS-Visit}(G, u)$
    $R \leftarrow R \cup set(L_{scc})$
  **end**
**end**

$O(|V|+|E|)$ 第二次深度优先搜索

**return** $R$

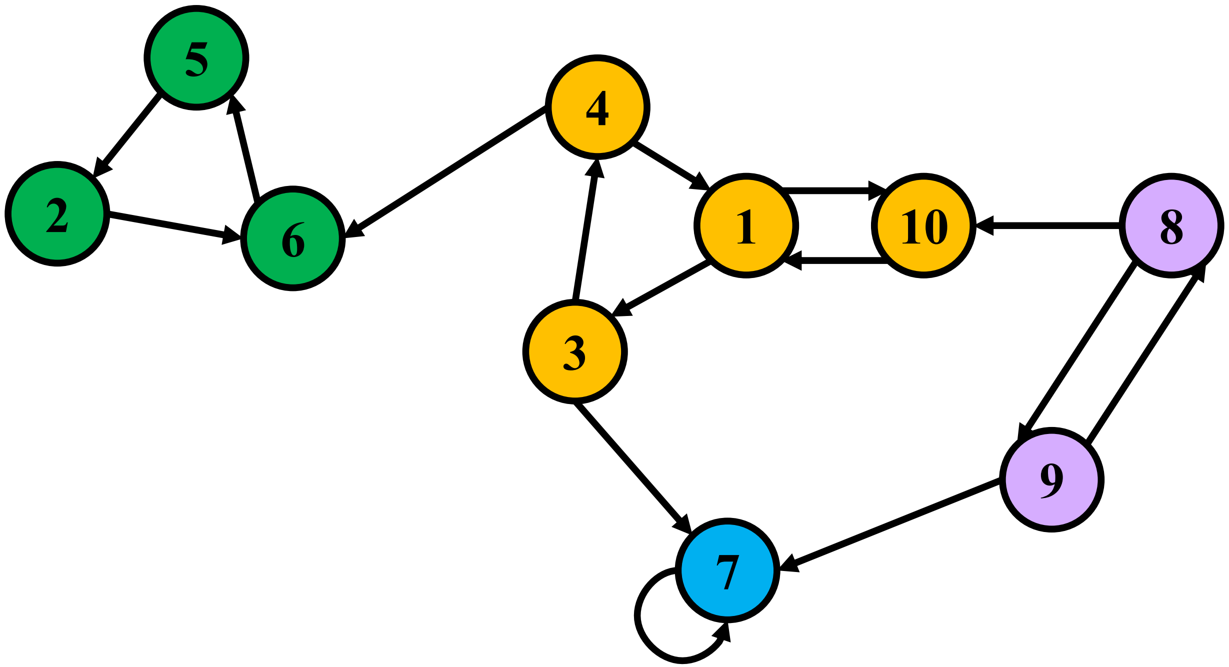
时间复杂度：$O(|V|+|E|)$

# 提纲

- 问题背景与定义
- 算法框架与实例
- 伪代码与复杂度
- **算法正确性证明**

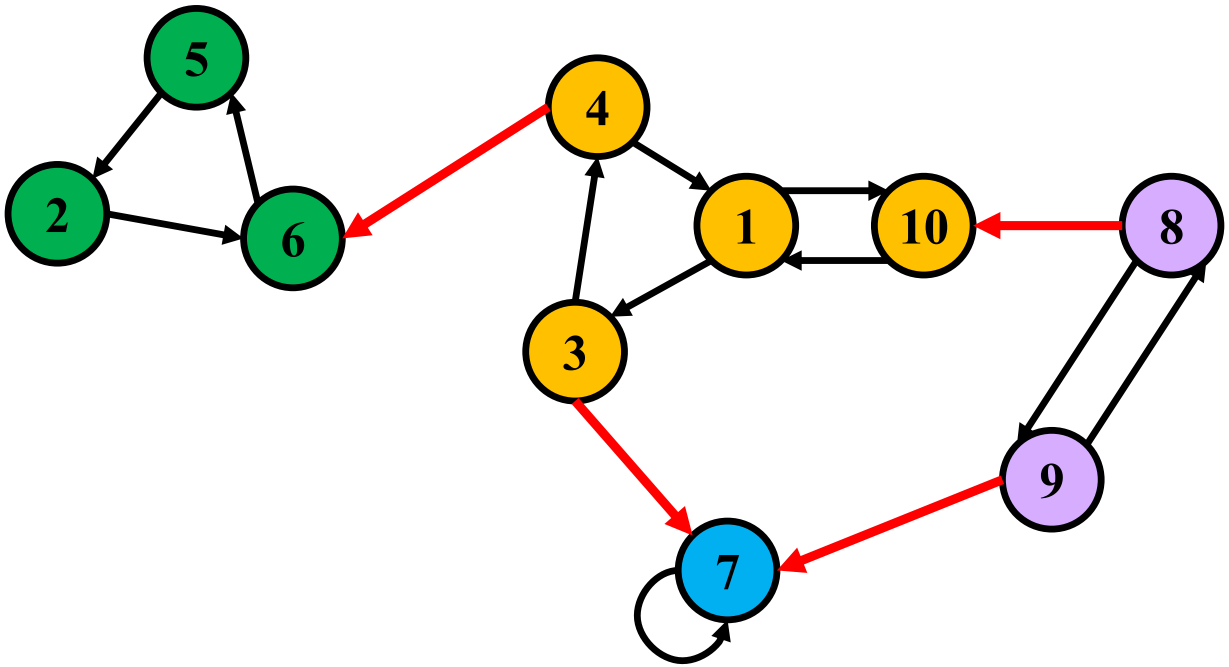
# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图



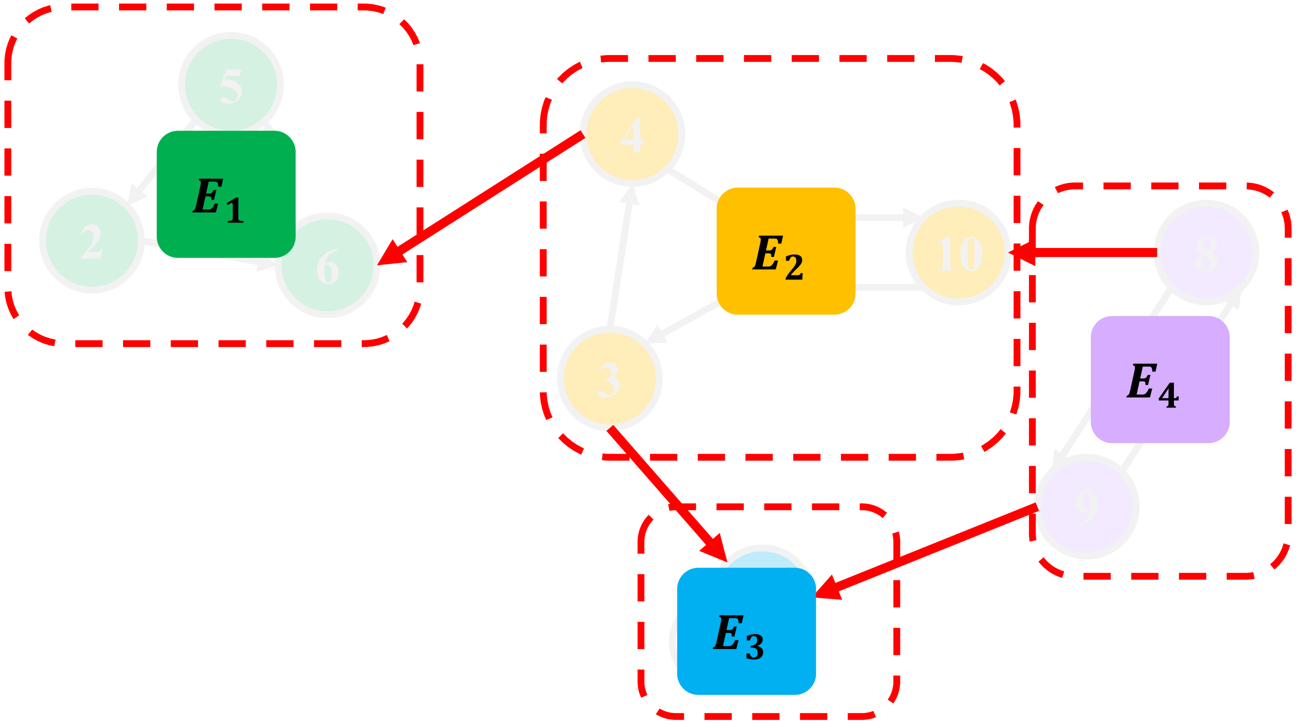
# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图



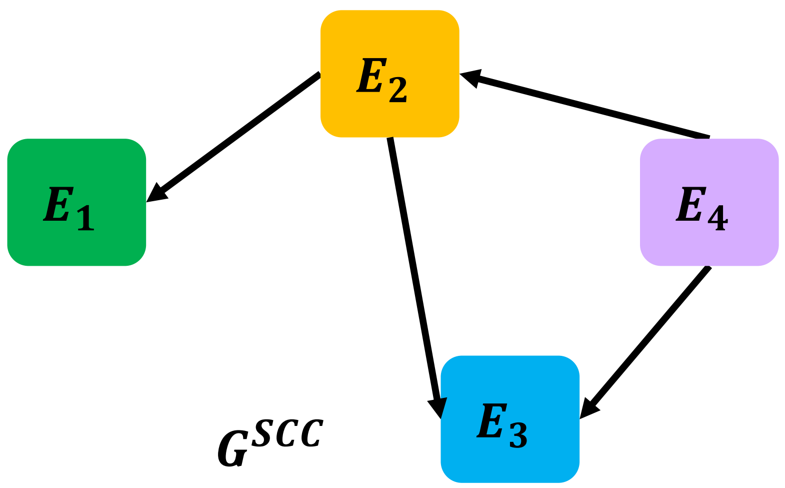
# 正确性证明

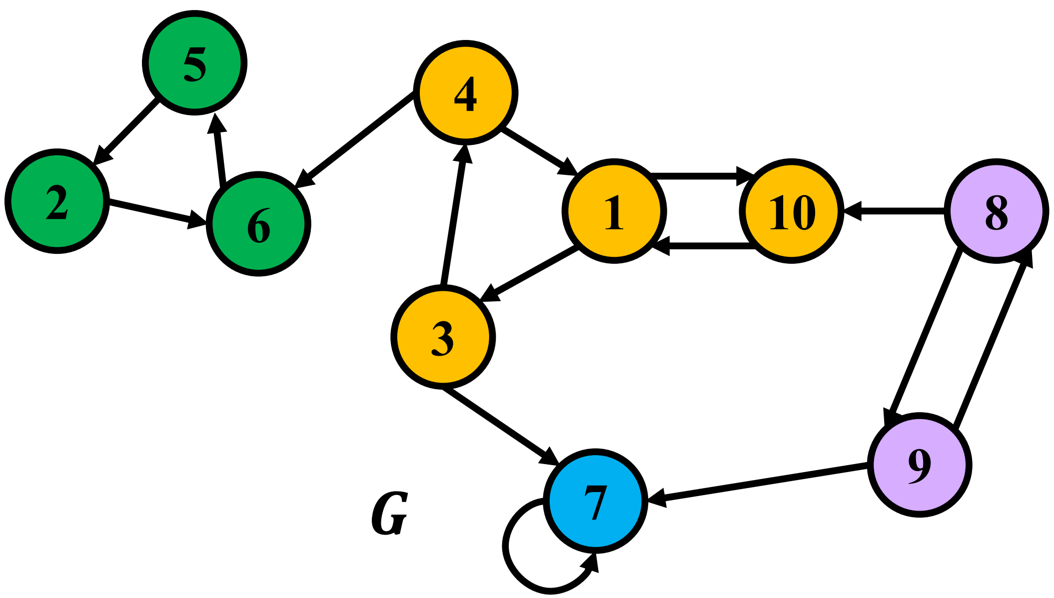
- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图



# 正确性证明

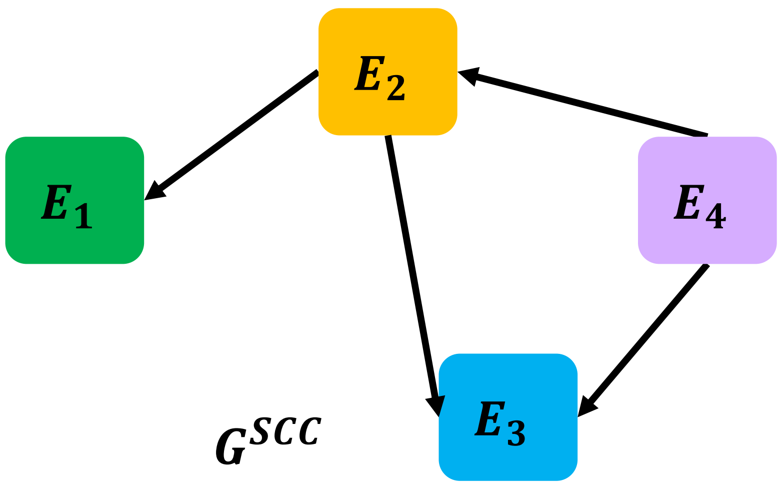
- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图

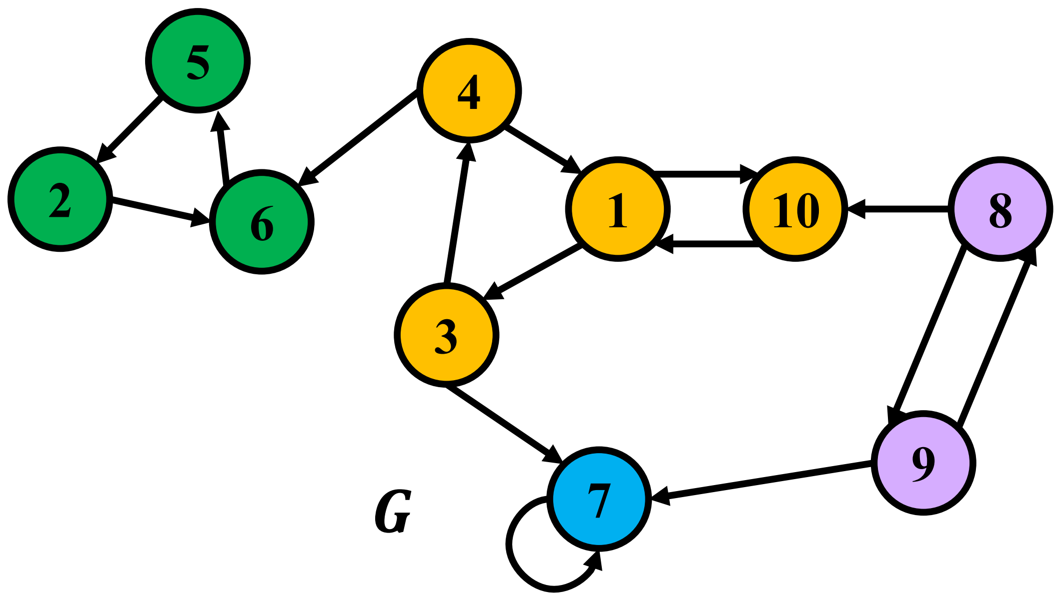


# 正确性证明

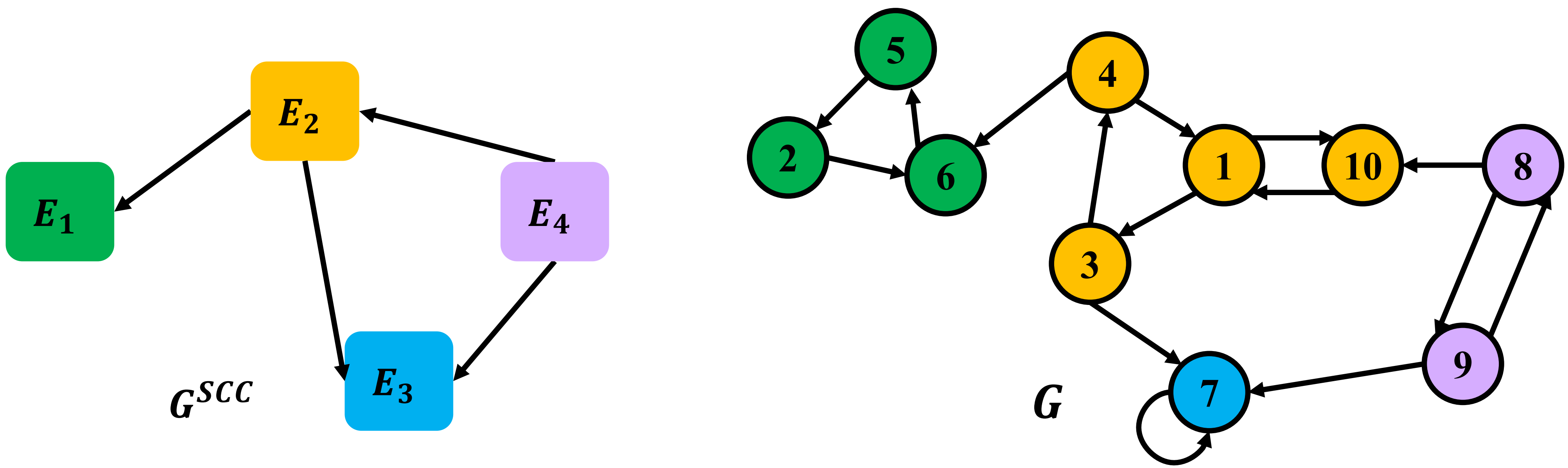
- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图



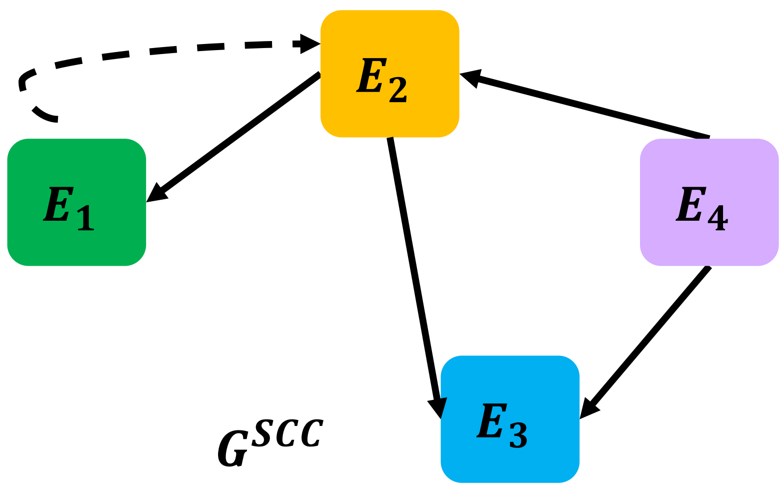
# 正确性证明

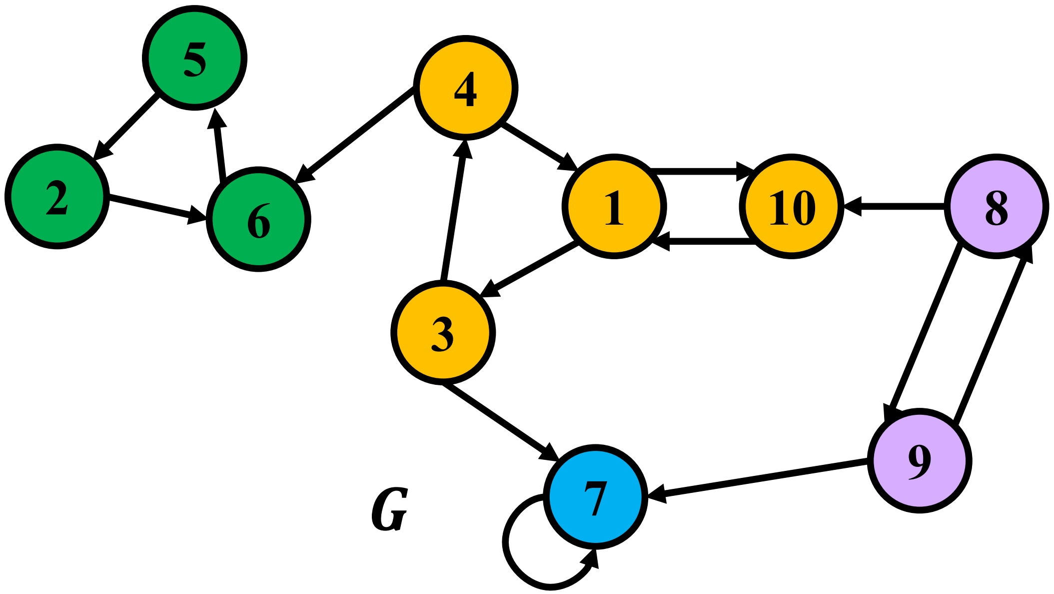
- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
  - 反证：若存在环



# 正确性证明

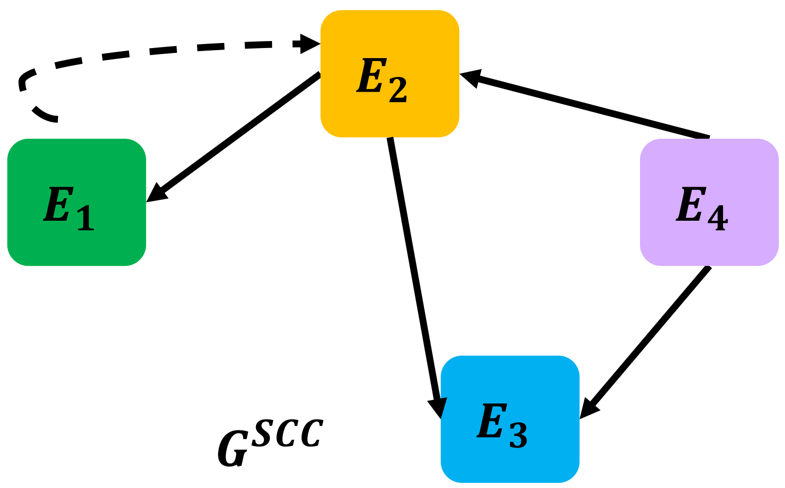
- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
  - 反证：若存在环

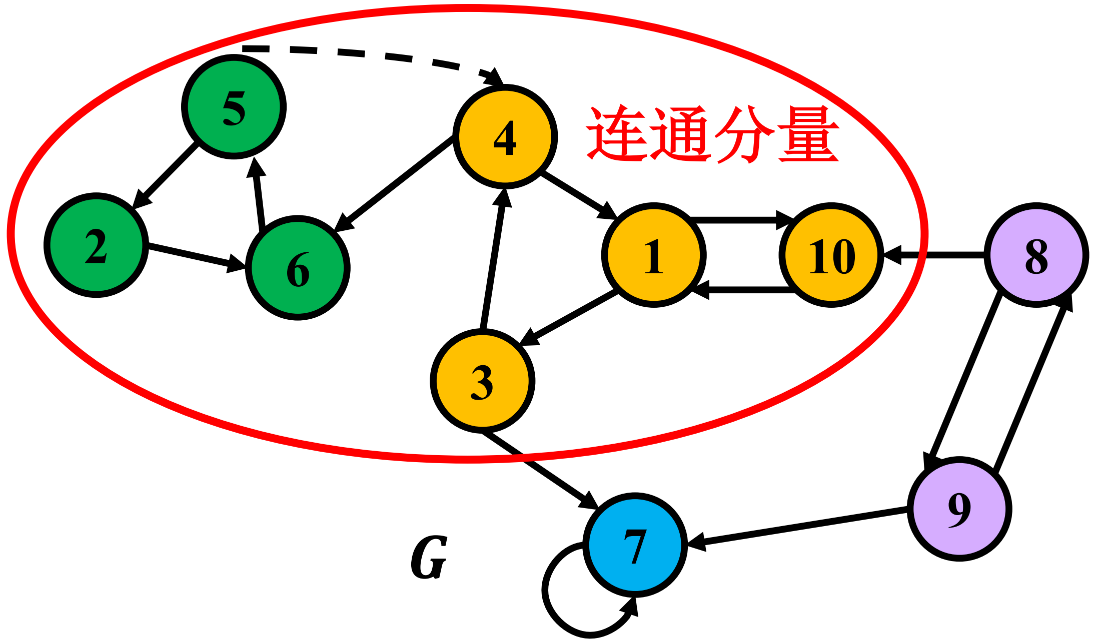


# 正确性证明

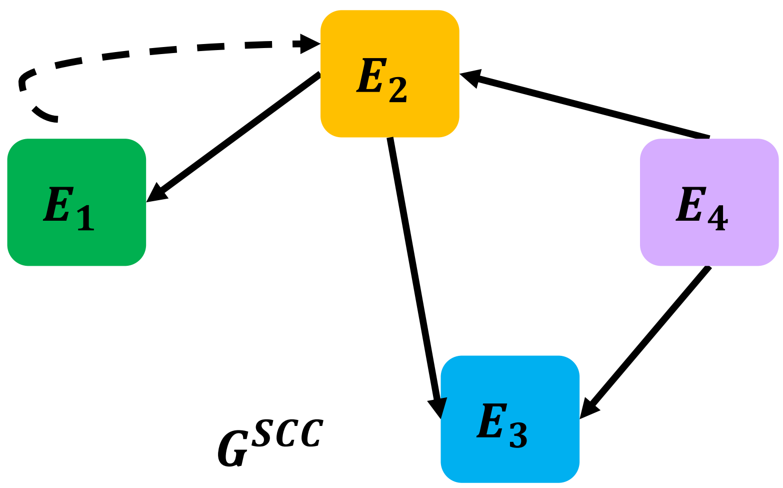
- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
  - 反证：若存在环

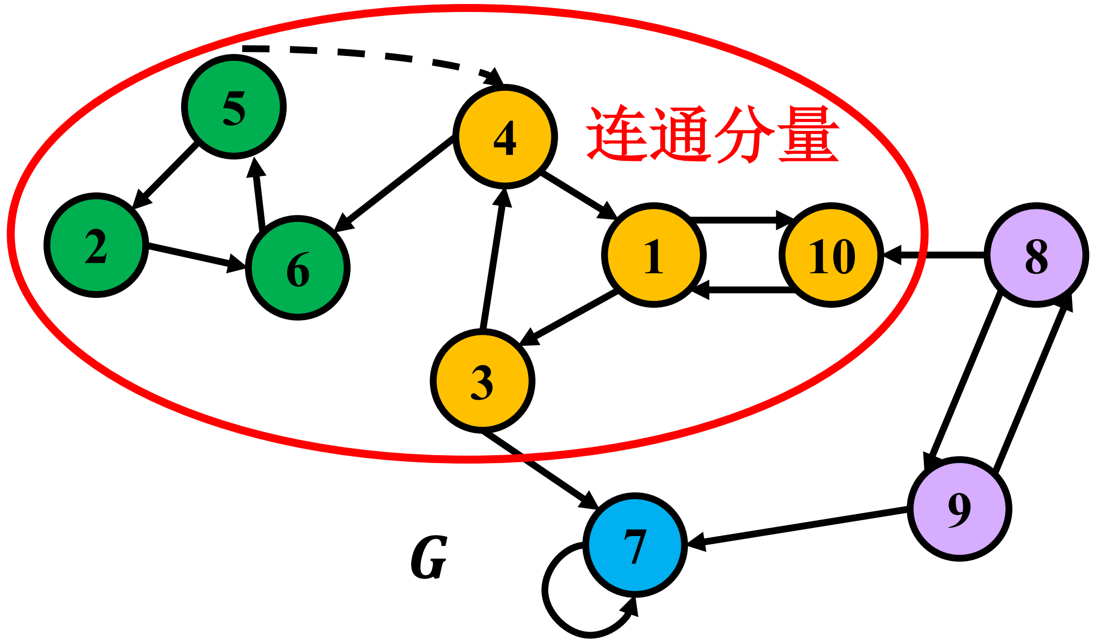


# 正确性证明

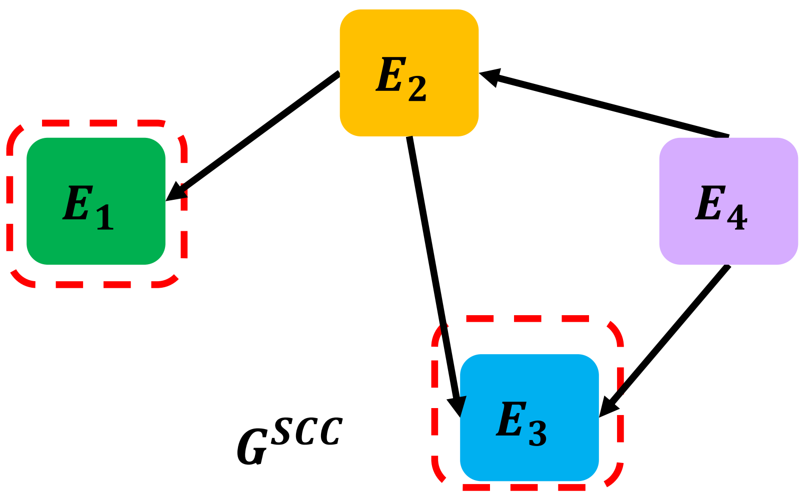
- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
  - 反证：若存在环，两强连通分量中顶点相互可达，与最大性矛盾



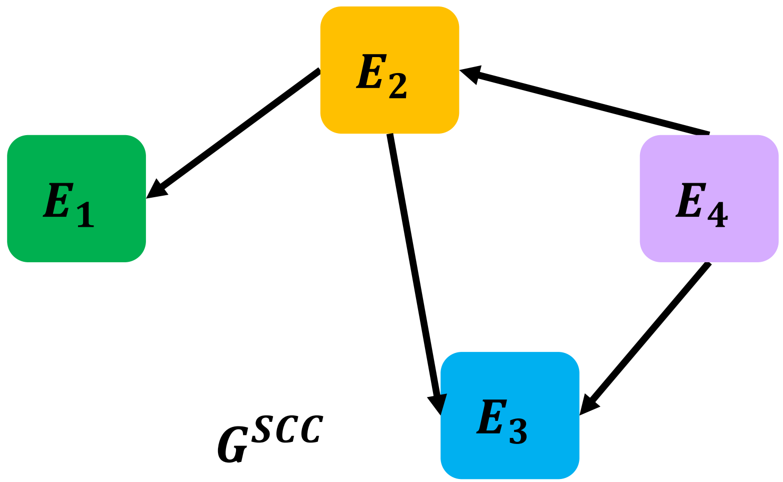
# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点



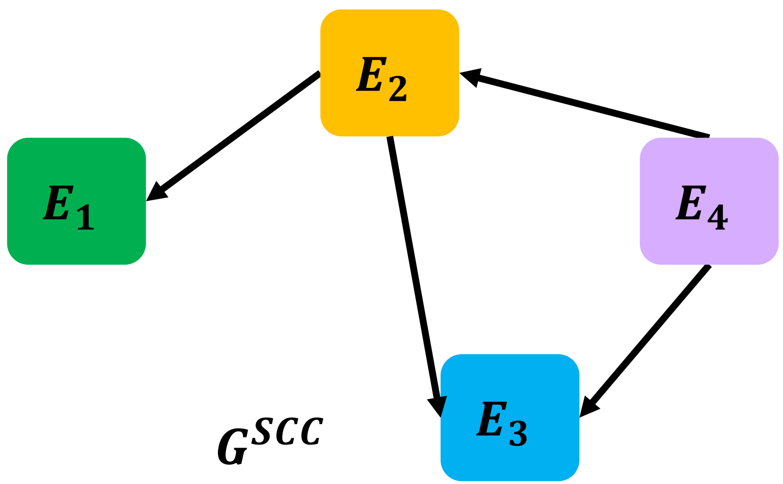
# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点
  - 性质1：$G^{SCC}$中存在至少一个$SCC_{Sink}$



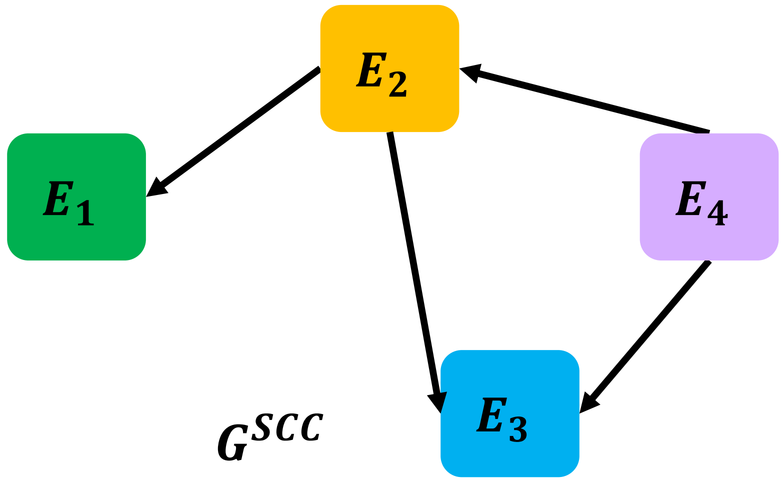
# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点
  - 性质1：$G^{SCC}$中存在至少一个$SCC_{Sink}$
  - 反证：若不存在，所有点均有出度，必存在环，矛盾



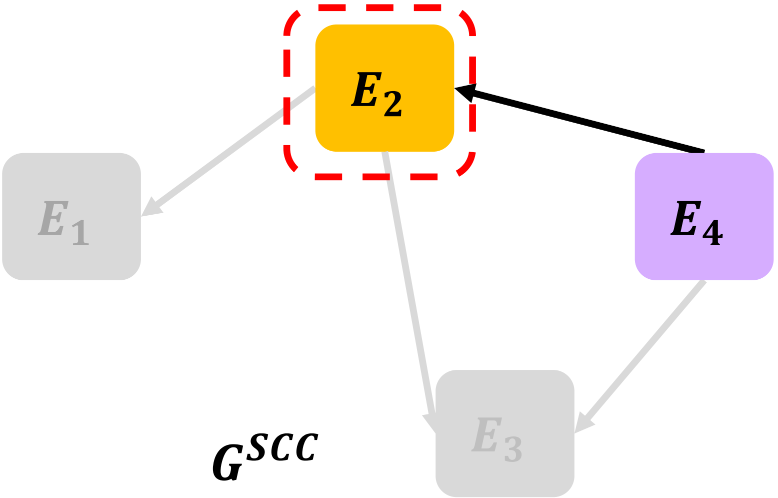
# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点
  - 性质1：$G^{SCC}$中存在至少一个$SCC_{Sink}$
  - 性质2：删除 $SCC_{Sink}$，会产生新的$SCC_{Sink}$



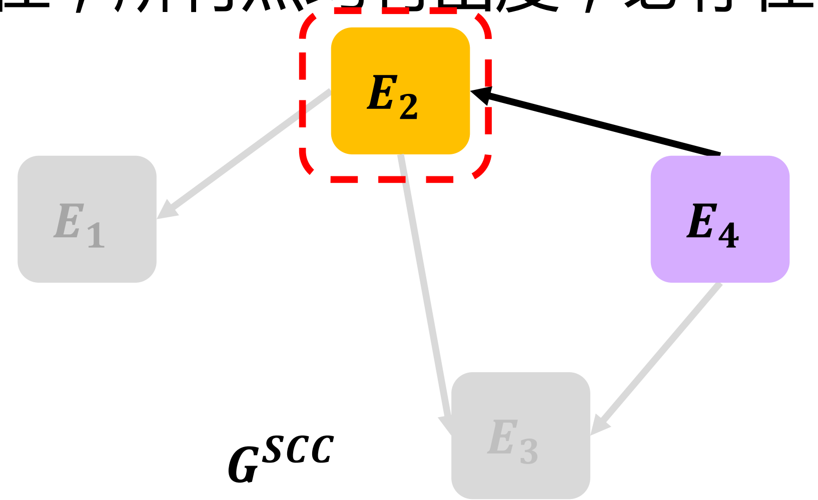
# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点
  - 性质1：$G^{SCC}$中存在至少一个$SCC_{Sink}$
  - 性质2：删除 $SCC_{Sink}$，会产生新的$SCC_{Sink}$



# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点
  - 性质1：$G^{SCC}$中存在至少一个$SCC_{Sink}$
  - 性质2：删除 $SCC_{Sink}$，会产生新的$SCC_{Sink}$
  - 反证：若不存在，所有点均有出度，必存在环；而无环图子图必无环，矛盾

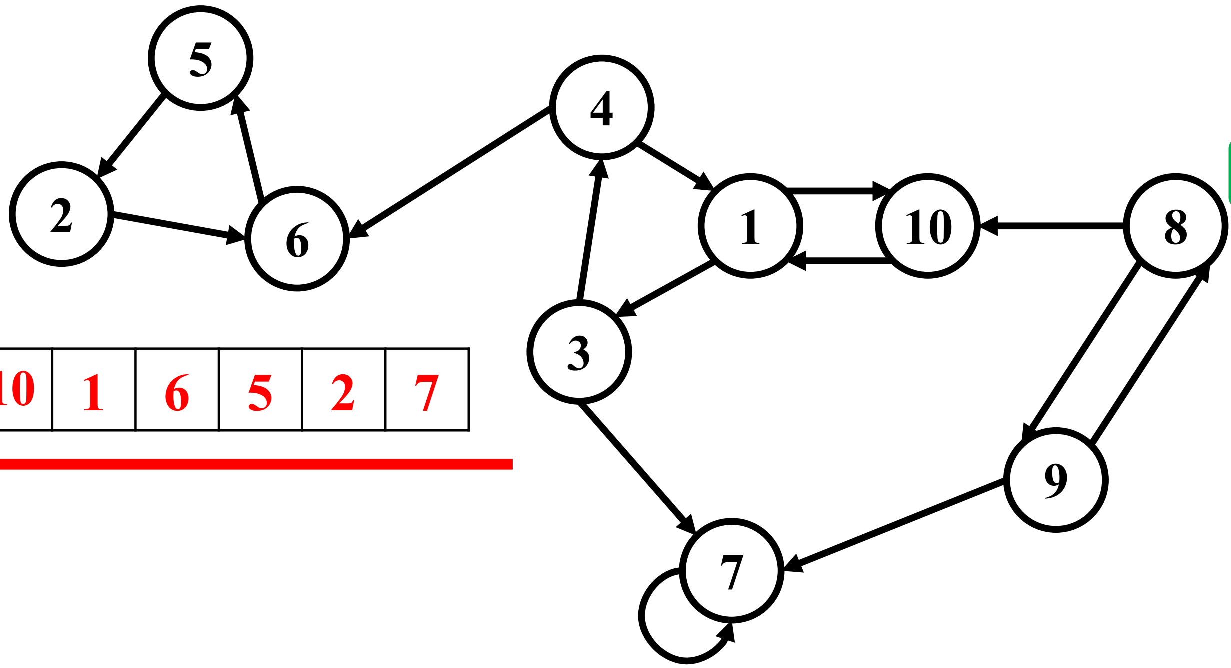


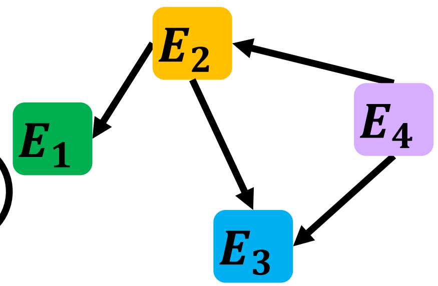

$G^{SCC}$

# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点
  - 性质1：$G^{SCC}$中存在至少一个$SCC_{Sink}$
  - 性质2：删除 $SCC_{Sink}$，会产生新的$SCC_{Sink}$

结合性质观察
算法第2次DFS



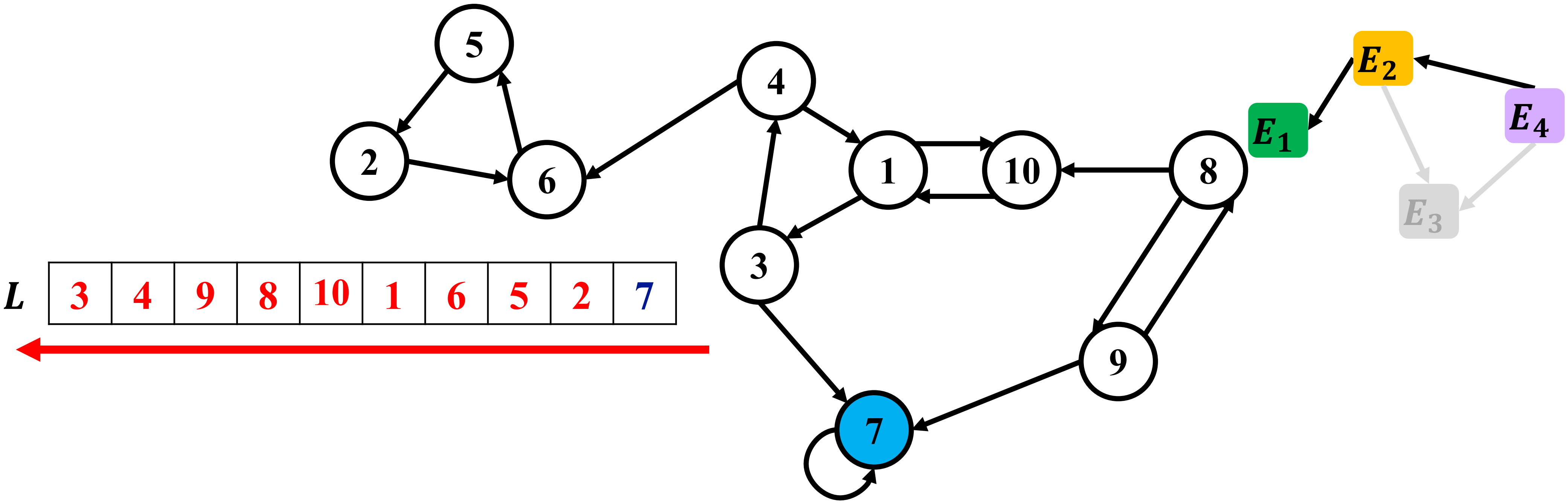



| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点
  - 性质1：$G^{SCC}$中存在至少一个$SCC_{Sink}$
  - 性质2：删除 $SCC_{Sink}$，会产生新的$SCC_{Sink}$

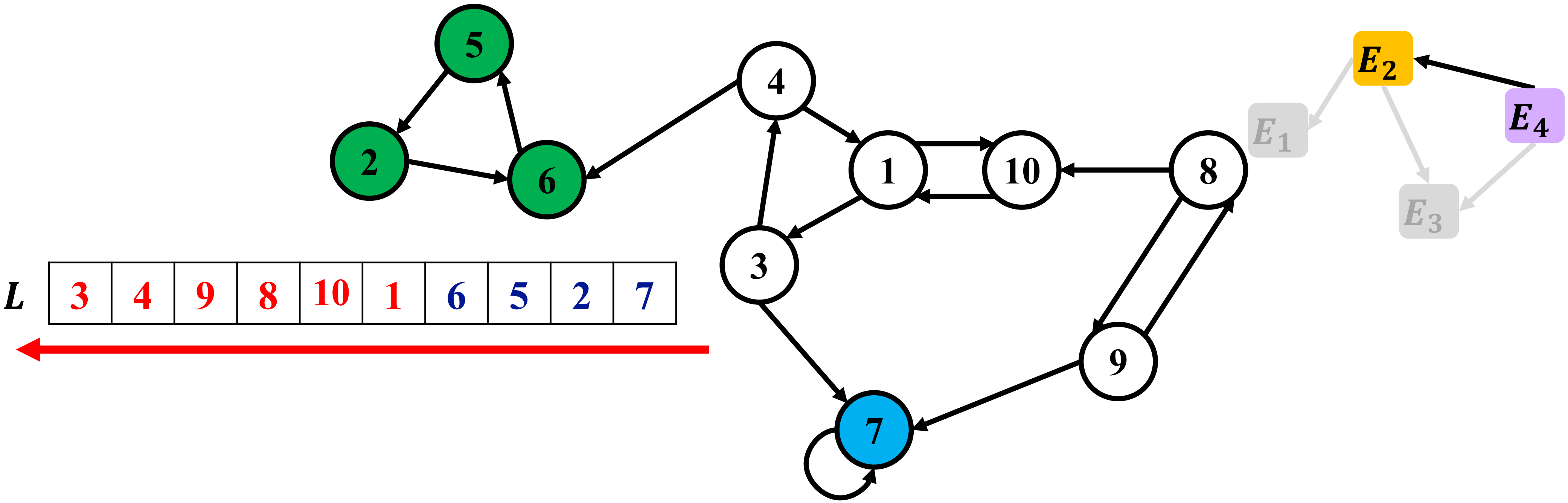
结合性质观察
算法第2次DFS



# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点
  - 性质1：$G^{SCC}$中存在至少一个$SCC_{Sink}$
  - 性质2：删除 $SCC_{Sink}$，会产生新的$SCC_{Sink}$

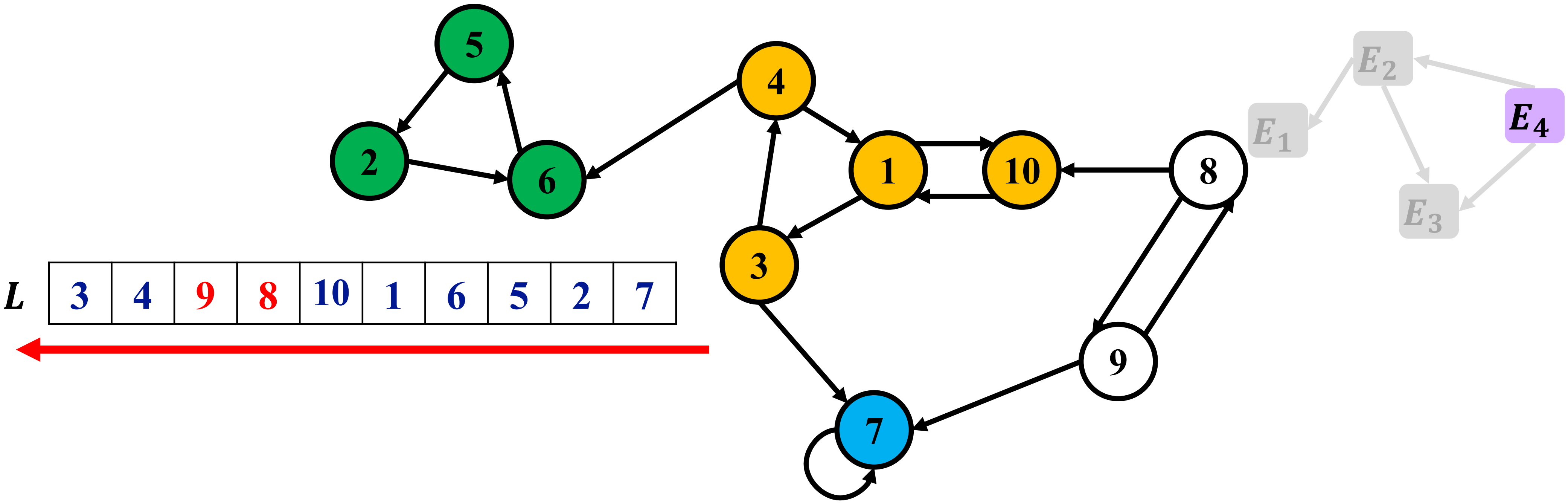
结合性质观察
算法第2次DFS



# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点
  - 性质1：$G^{SCC}$中存在至少一个$SCC_{Sink}$
  - 性质2：删除 $SCC_{Sink}$，会产生新的$SCC_{Sink}$

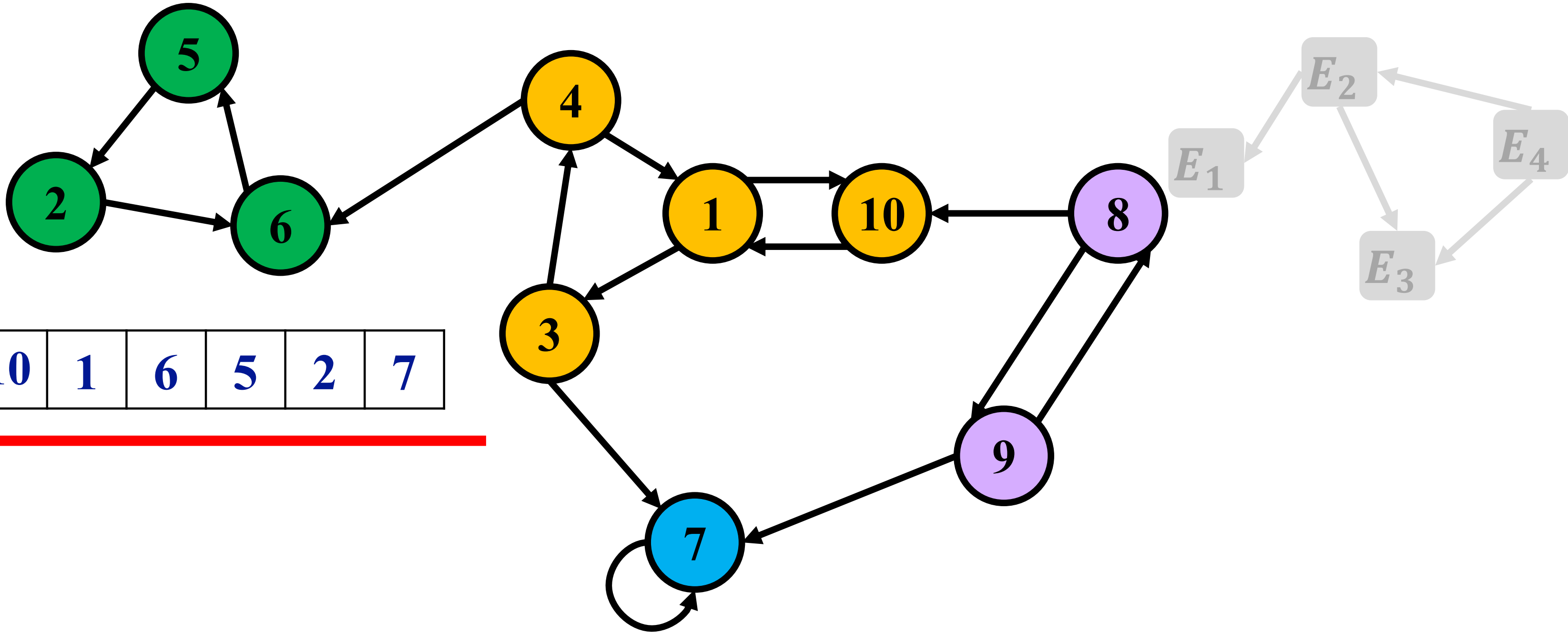
结合性质观察
算法第2次DFS



# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点
  - 性质1：$G^{SCC}$中存在至少一个$SCC_{Sink}$
  - 性质2：删除 $SCC_{Sink}$，会产生新的$SCC_{Sink}$

结合性质观察算法第2次DFS

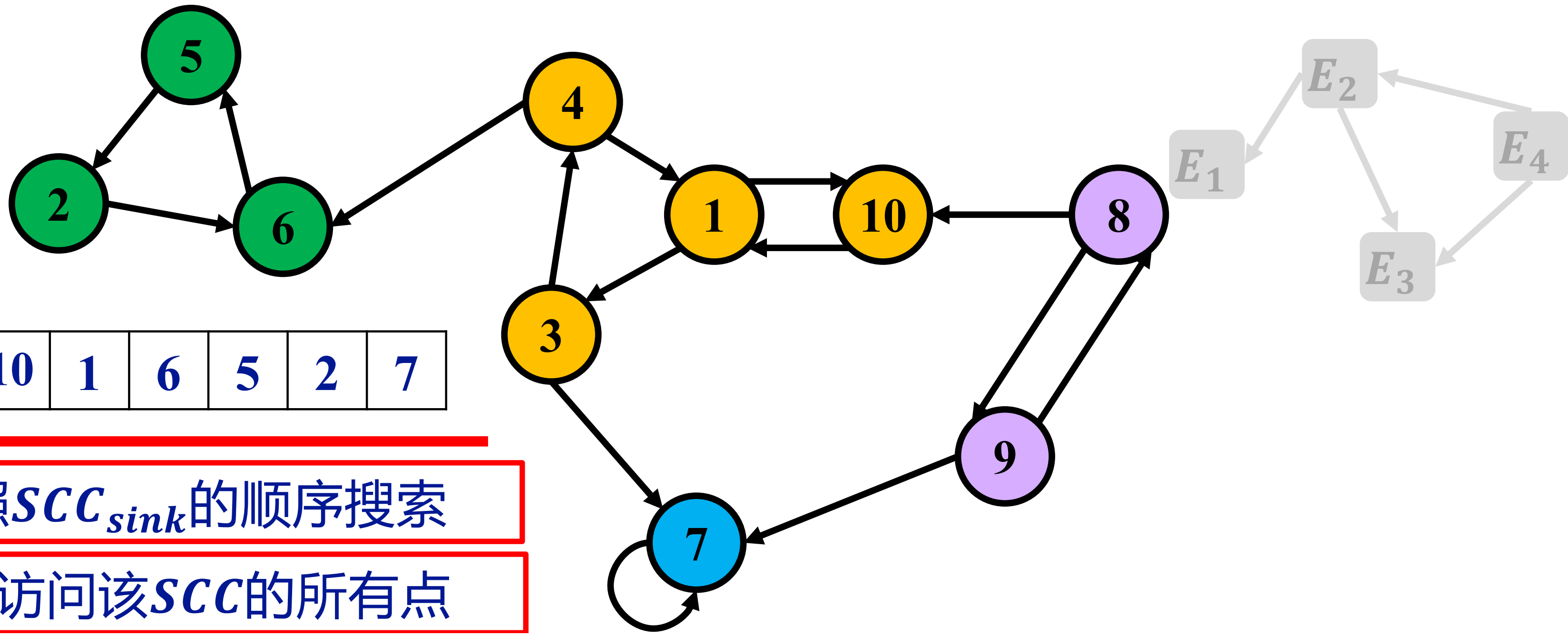



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $L$ | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

# 正确性证明

- 强连通分量图$G^{SCC}$：把强连通分量看作一个点，得到有向图
  - 性质：$G^{SCC}$一定是有向无环图
- $SCC_{Sink}$：$G^{SCC}$中出度为 0 的点
  - 性质1：$G^{SCC}$中存在至少一个$SCC_{Sink}$
  - 性质2：删除 $SCC_{Sink}$，会产生新的$SCC_{Sink}$

结合性质观察
算法第2次DFS




| $L$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 4 | 9 | 8 | 10 | 1 | 6 | 5 | 2 | 7 |

第2次DFS按照$SCC_{sink}$的顺序搜索

每次搜索恰好访问该$SCC$的所有点

# 正确性证明

通过按$L$逆序执行DFS实现

？

第2次DFS按照$SCC_{sink}$的顺序搜索

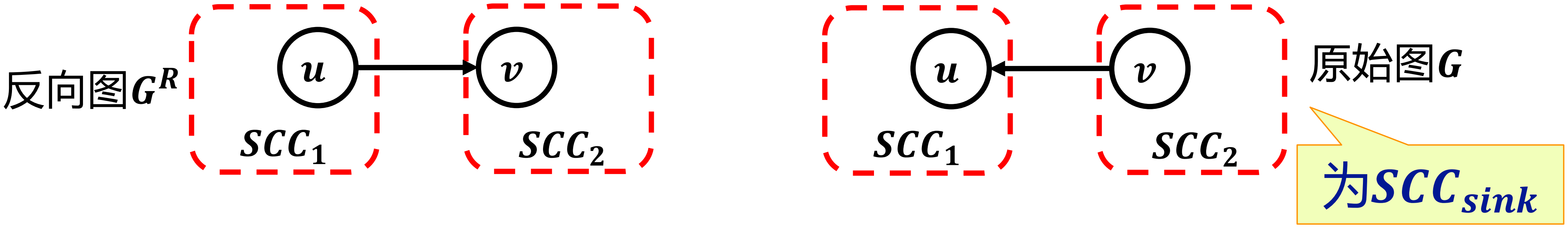
？

每次搜索恰好访问该$SCC$的所有点

算法正确

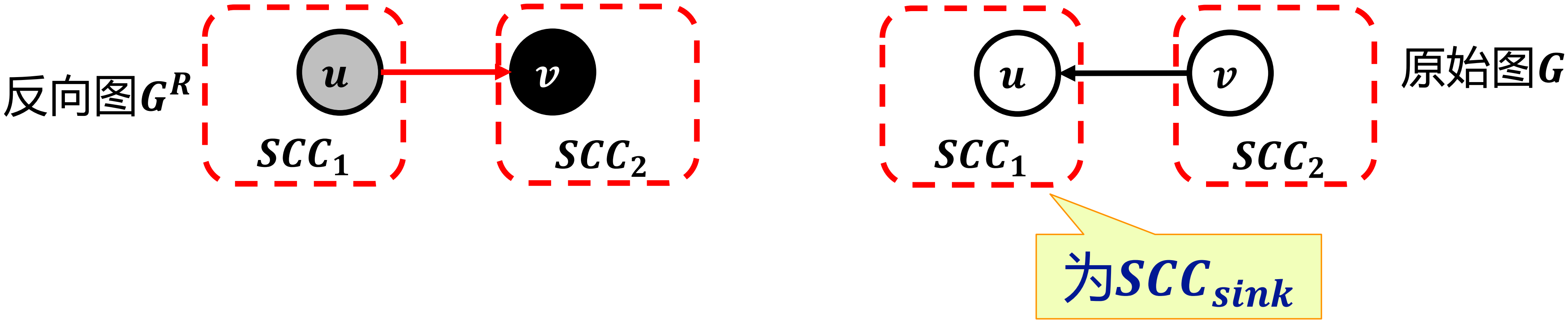
# 正确性证明

- 给定反向图$\boldsymbol{G^R}$，存在边$(\boldsymbol{u}, \boldsymbol{v})$，$\boldsymbol{u} \in \boldsymbol{SCC_1}$，$\boldsymbol{v} \in \boldsymbol{SCC_2}$



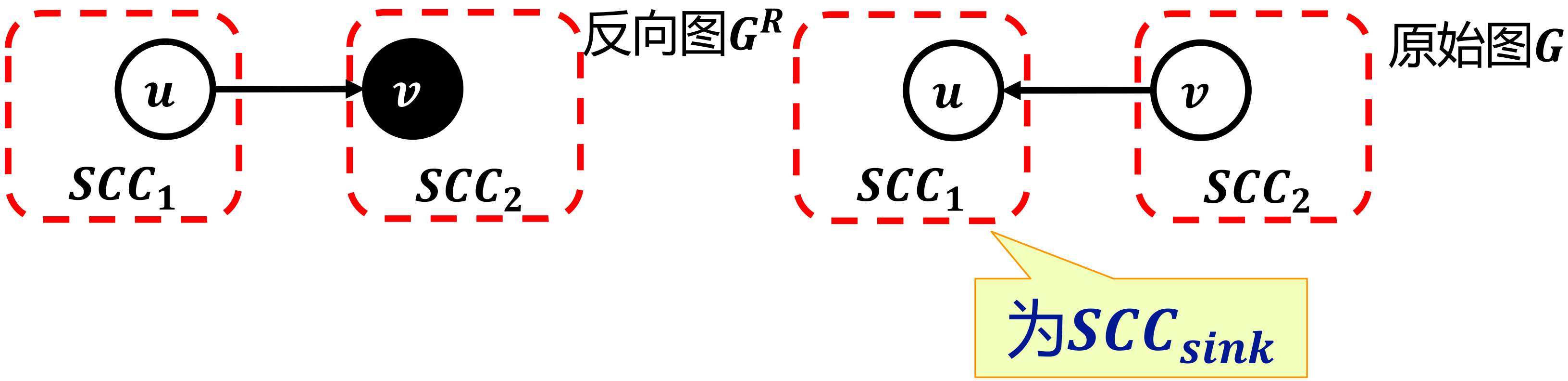
# 正确性证明

- 给定反向图$G^R$，存在边$(u,v)$，$u \in SCC_1$，$v \in SCC_2$
  - 若先搜索$u$，会从$u$搜索$v$，则$f(v) < f(u)$



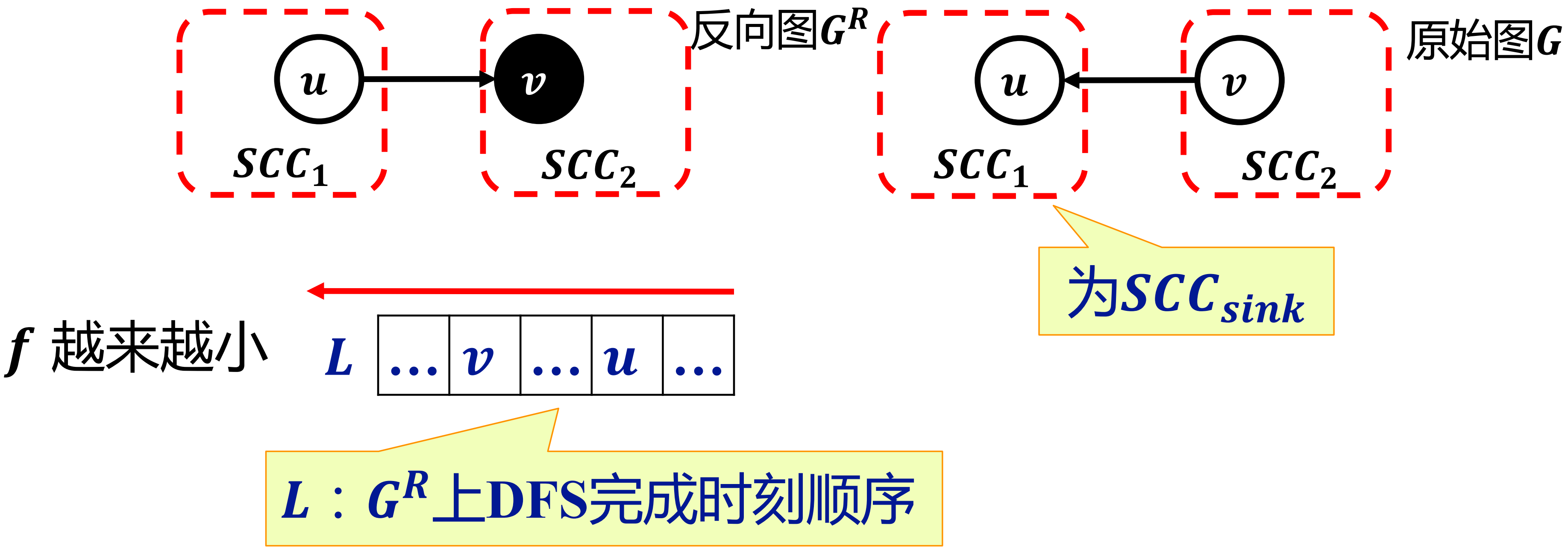
# 正确性证明

- 给定反向图$G^R$，存在边$(u,v)$，$u \in SCC_1$，$v \in SCC_2$
  - 若先搜索$u$，会从$u$搜索$v$，则$f(v) < f(u)$
  - 若先搜索$v$，$v$搜索完成才开始搜索$u$，则$f(v) < f(u)$



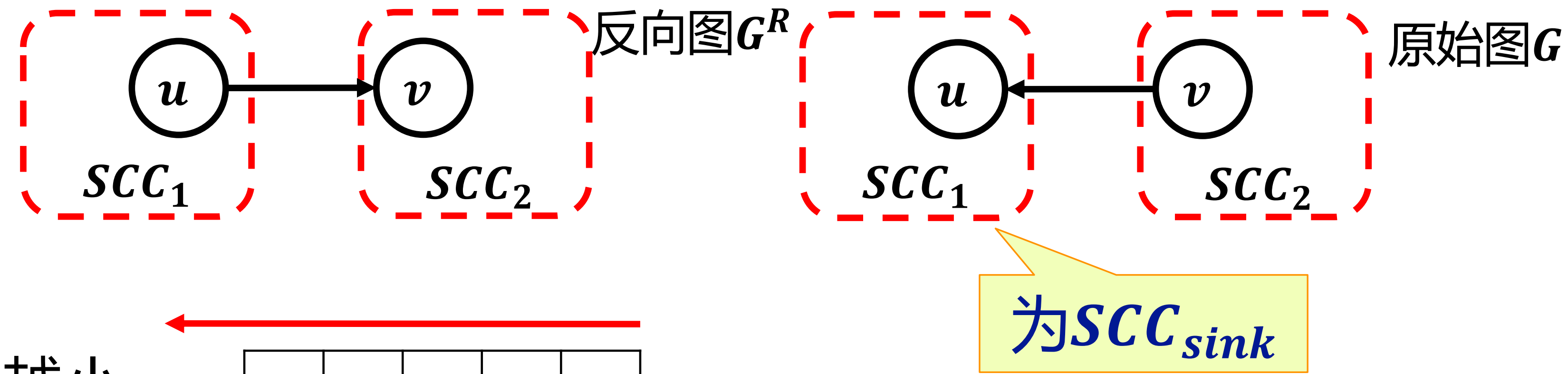
# 正确性证明

- 给定反向图$G^R$，存在边$(u,v)$，$u \in SCC_1$，$v \in SCC_2$
  - 若先搜索$u$，会从$u$搜索$v$，则$f(v) < f(u)$
  - 若先搜索$v$，$v$搜索完成才开始搜索$u$，则$f(v) < f(u)$



# 正确性证明

- 给定反向图$G^R$，存在边$(u,v)$，$u \in SCC_1$，$v \in SCC_2$
  - 若先搜索$u$，会从$u$搜索$v$，则$f(v) < f(u)$
  - 若先搜索$v$，$v$搜索完成才开始搜索$u$，则$f(v) < f(u)$
  - 所以按$L$的逆序，总是先搜索$u$，符合$SCC_{sink}$的顺序



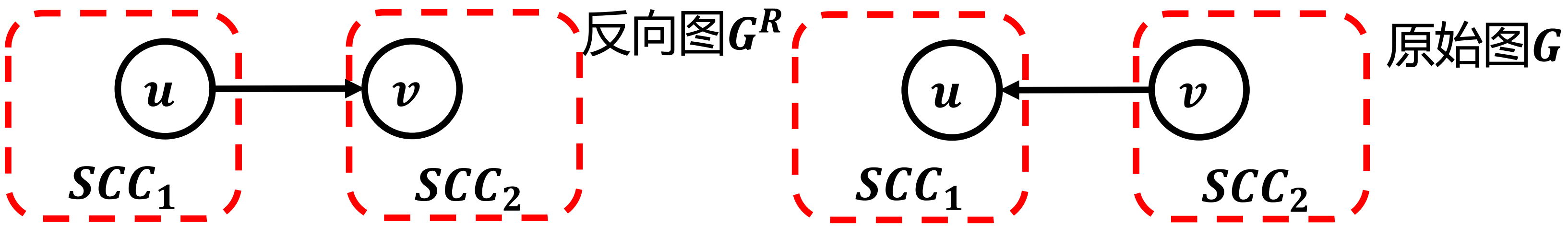

$f$ 越来越小

| $L$ | $\dots$ | $v$ | $\dots$ | $u$ | $\dots$ |
|---|---|---|---|---|---|

$L$：$G^R$上DFS完成时刻顺序

# 正确性证明

- 给定反向图$G^R$，存在边$(u,v)$，$u \in SCC_1$，$v \in SCC_2$
  - 若先搜索$u$，会从$u$搜索$v$，则$f(v) < f(u)$
  - 若先搜索$v$，$v$搜索完成才开始搜索$u$，则$f(v) < f(u)$
  - 所以按$L$的逆序，总是先搜索$u$，符合$SCC_{sink}$的顺序




为$SCC_{sink}$

通过按$L$逆序执行DFS实现

第2次DFS按照$SCC_{sink}$的顺序搜索

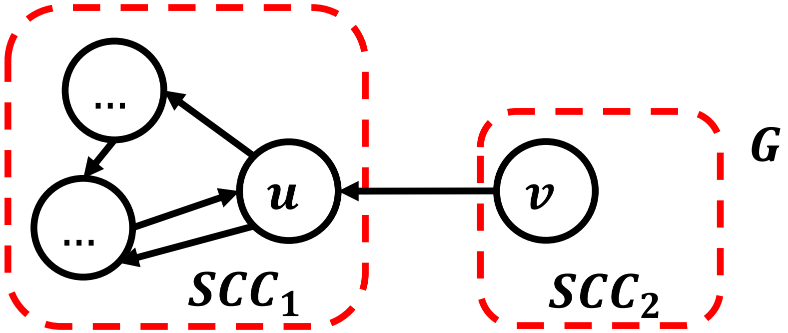
？

每次搜索恰好访问该$SCC$的所有点

算法正确

# 正确性证明

- 强连通分量内，顶点相互可达 ➡ DFS可以访问到该$SCC$所有点
- $SCC_{sink}$出度为0 ➡ DFS不会访问该$SCC$以外的点




通过按$L$逆序执行DFS实现

↓ ✓

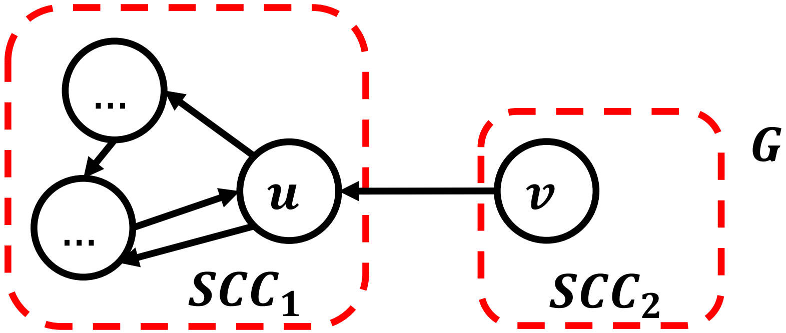
第2次DFS按照$SCC_{sink}$的顺序搜索 ➡ 算法正确

↓ ?

每次搜索恰好访问该$SCC$的所有点

# 正确性证明

- 强连通分量内，顶点相互可达 → DFS可以访问到该$SCC$所有点
- $SCC_{sink}$出度为0 → DFS不会访问该$SCC$以外的点




通过按$L$逆序执行DFS实现

↓ ✓

第2次DFS按照$SCC_{sink}$的顺序搜索

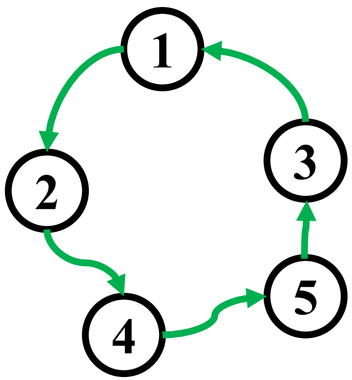
↓ ✓

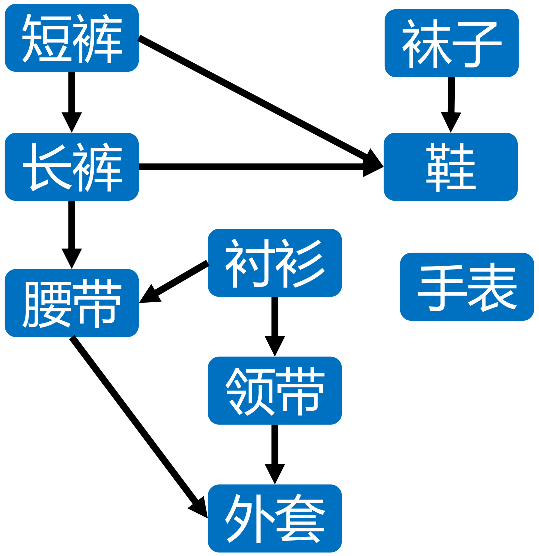
每次搜索恰好访问该$SCC$的所有点

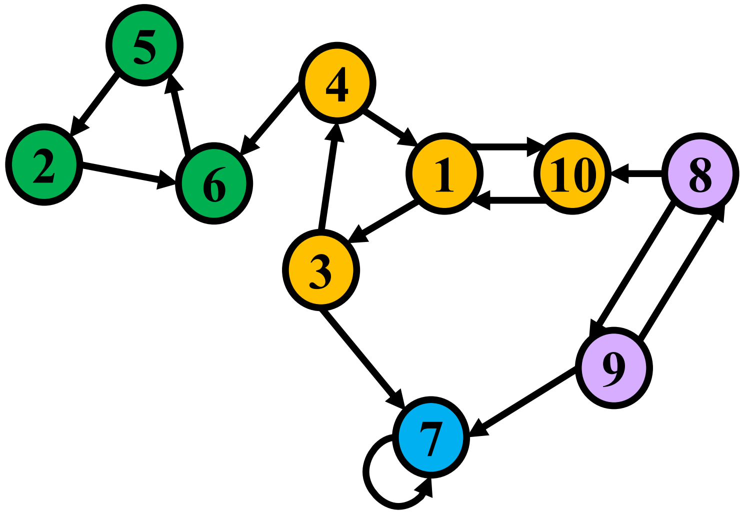
→ 算法正确

# 小结

- 深度优先搜索的应用




环的存在性判定

拓扑排序

强连通分量

利用深度优先搜索边的性质

利用深度优先搜索点的性质（括号化定理）

# 图算法篇：最小生成树I

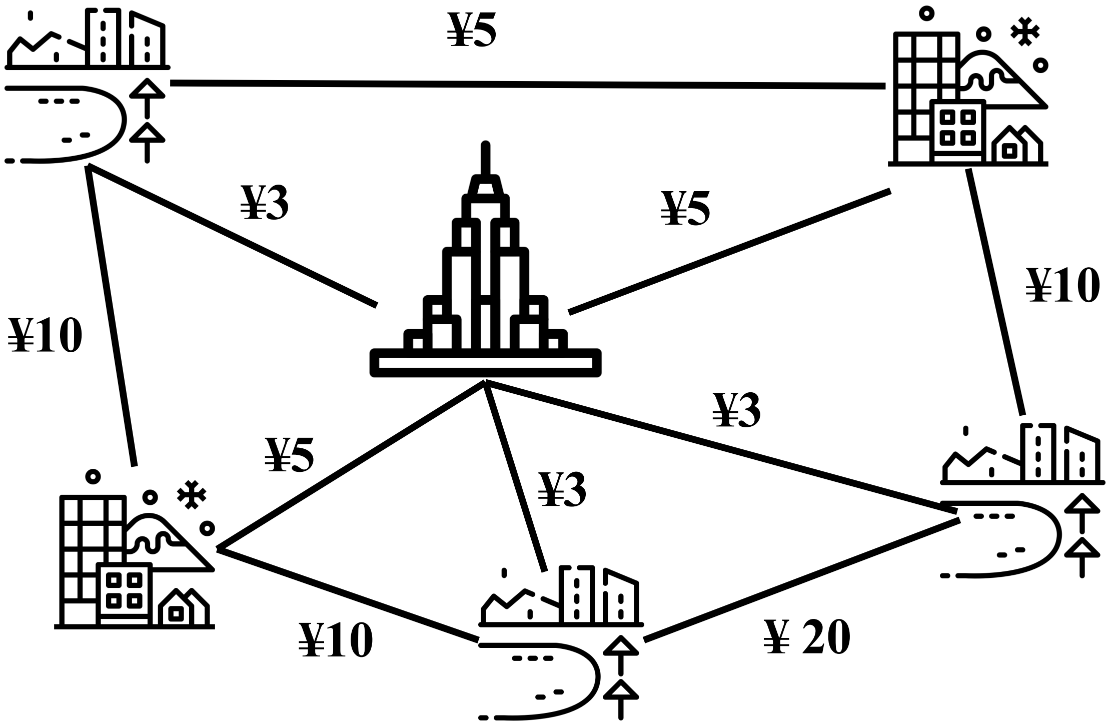
北京航空航天大学
计算机学院

# 提纲

- 问题背景
- 通用框架
- Prim算法
- 算法实例
- 算法分析

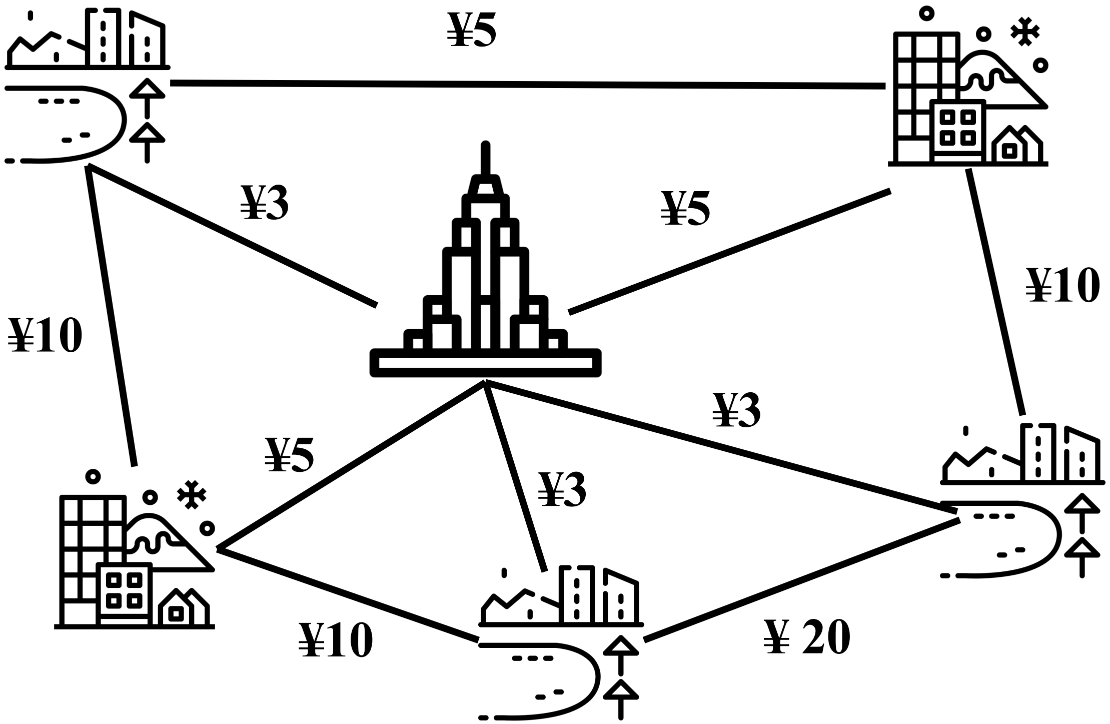
# 问题背景：道路修建

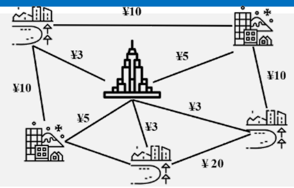
- 需要修建道路连通城市，各道路花费不同



# 问题背景：道路修建

- 需要修建道路连通城市，各道路花费不同

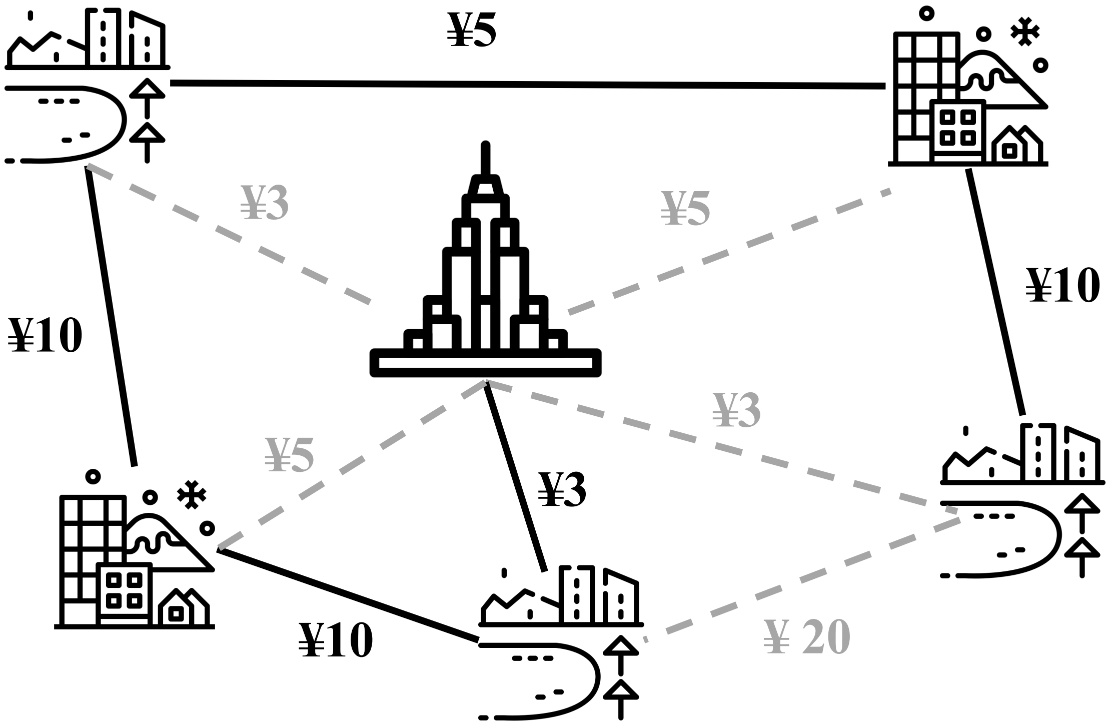


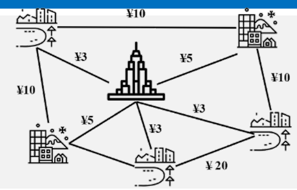
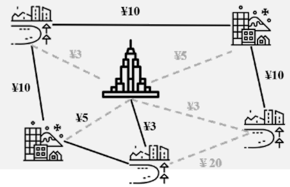

| 方案 | 花费 |
| --- | --- |
|  | ¥74 |

$$\text{花费：}10 + 10 + 10 + 5 + 20 + 3 + 3 + 5 + 3 + 5 = 74$$

# 问题背景：道路修建

- 需要修建道路连通城市，各道路花费不同

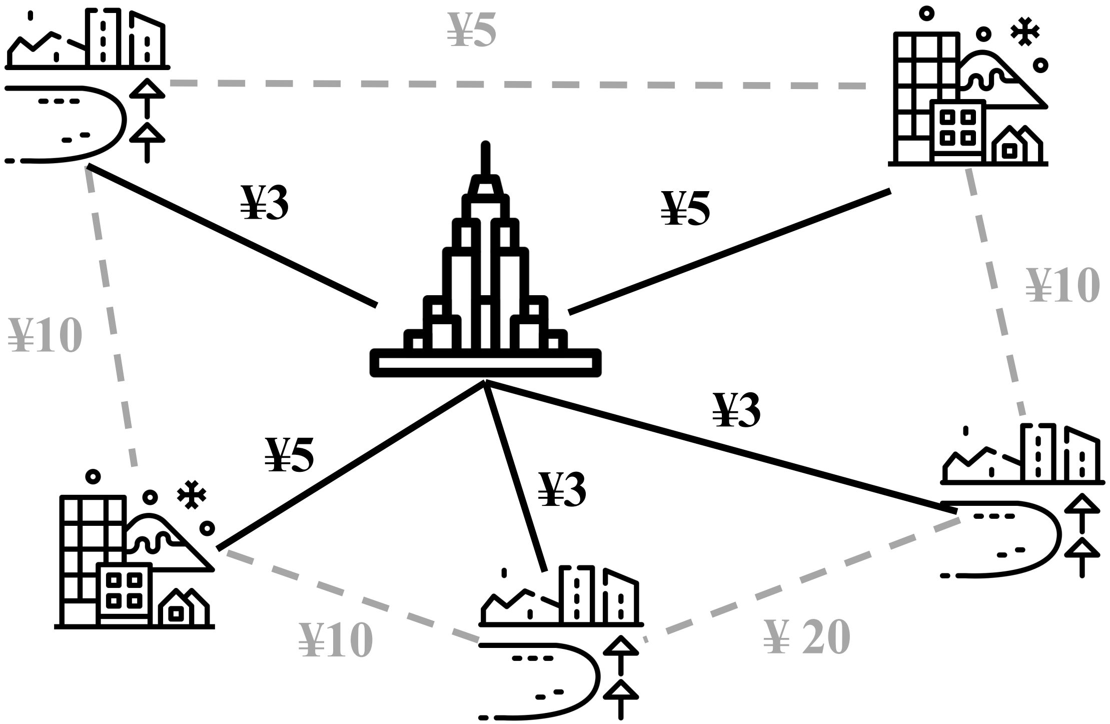


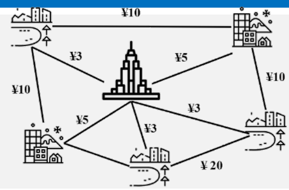
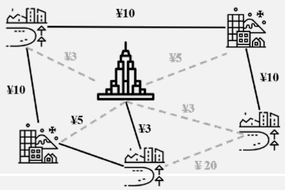
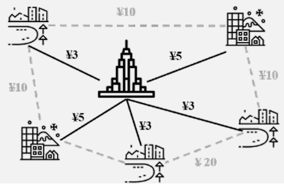

| 方案 | 花费 |
|---|---|
|  | ¥74 |
|  | ¥38 |

花费：$\mathbf{10+10+10+5+3=38}$

# 问题背景：道路修建

- 需要修建道路连通城市，各道路花费不同

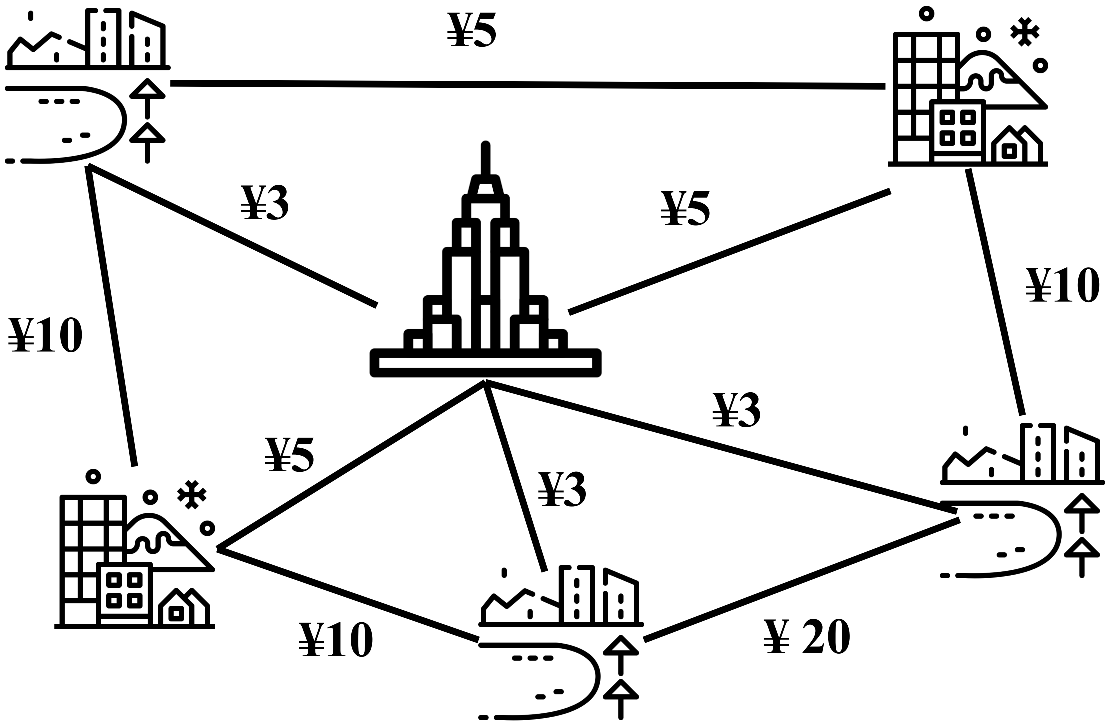


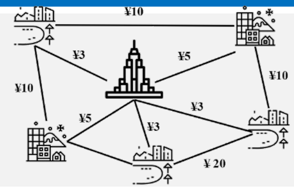
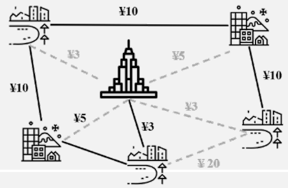
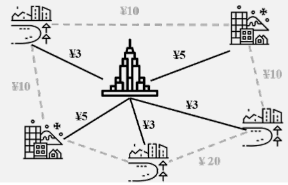

| 方案 | 花费 |
| --- | --- |
|  | ¥74 |
|  | ¥38 |
|  | ¥19 |

花费：$3 + 3 + 5 + 3 + 5 = 19$

# 问题背景：道路修建

- 需要修建道路连通城市，各道路花费不同

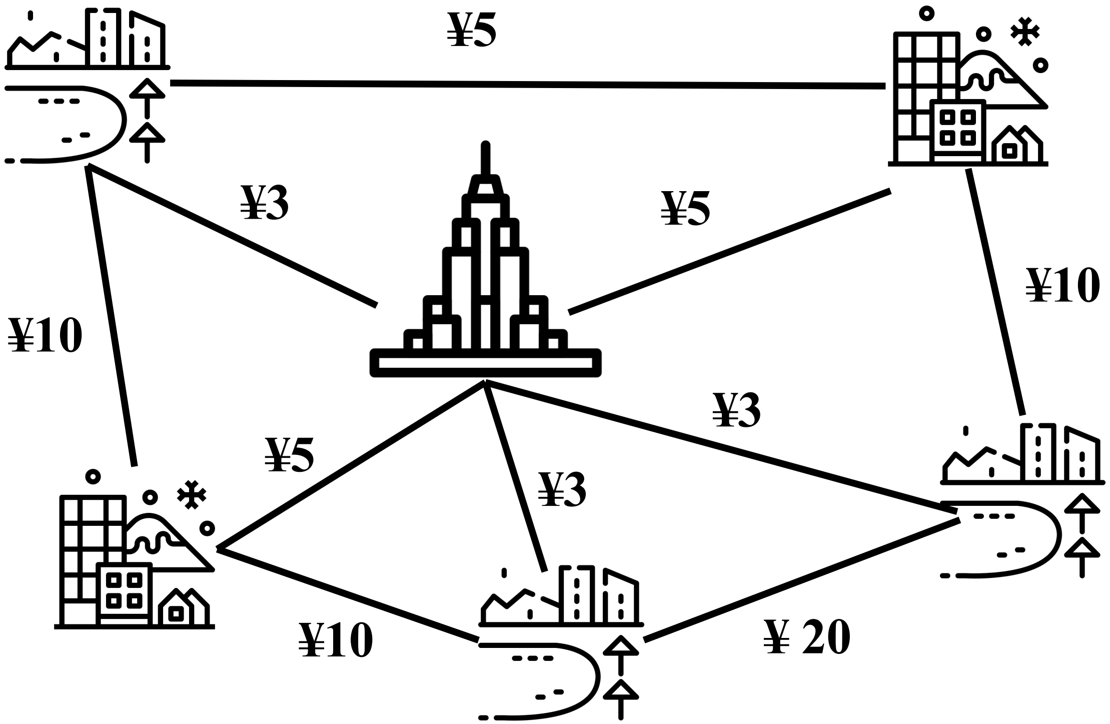



| 方案 | 花费 |
|---|---|
|  | ¥74 |
|  | ¥38 |
|  | ¥19 |

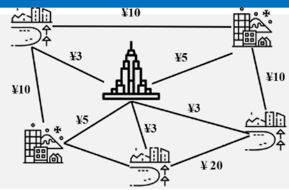
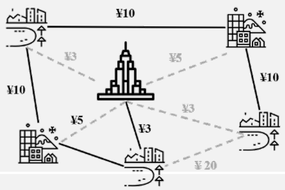
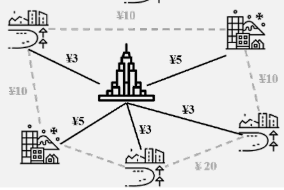
问题：连通各城市的最小花费是多少？

# 问题背景：道路修建

- 需要修建道路连通城市，各道路花费不同



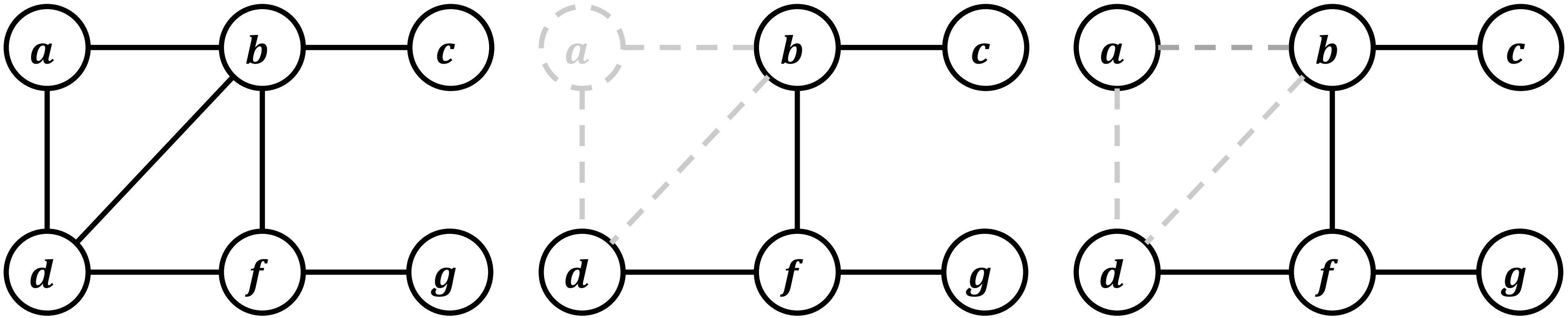

问题：连通各城市的最小花费是多少？

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

权重最小的连通生成子图

# 图的概念回顾：生成子图

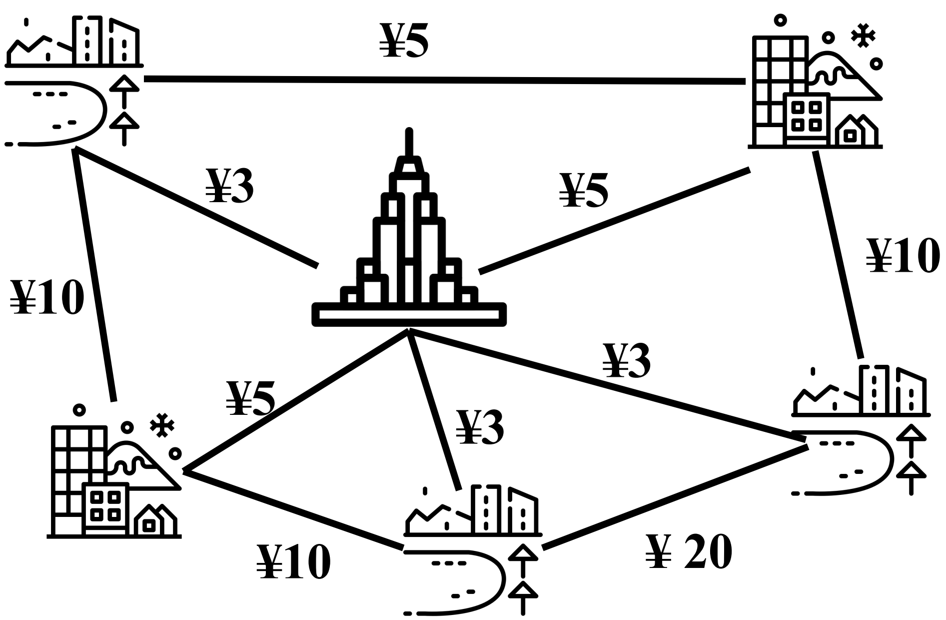
- 子图(Subgraph)
  - 如果 $V' \subseteq V, E' \subseteq E$，则称图$G' =< V', E' >$是图$G$的一个子图
- 生成子图(Spanning Subgraph)
  - 如果$V' = V, E' \subseteq E$，则称图$G' =< V', E' >$是图$G$的一个生成子图

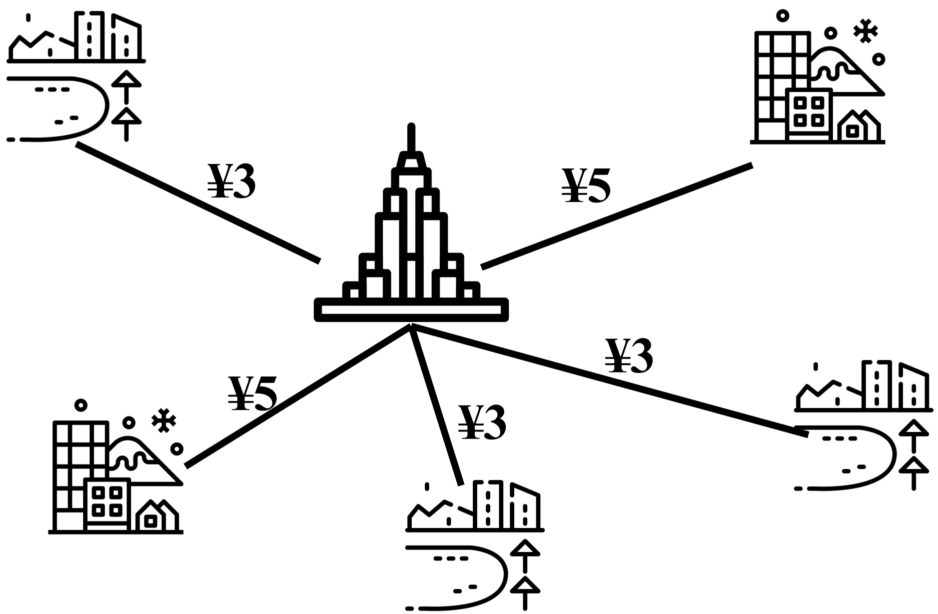



图$G$　　　$G$的子图　　　$G$的生成子图

# 图的概念：生成树

- 生成树(Spanning Tree)
  - 图$T' =< V', E' >$是无向图 $G$ 的一个生成子图，并且是连通、无环路的(树)



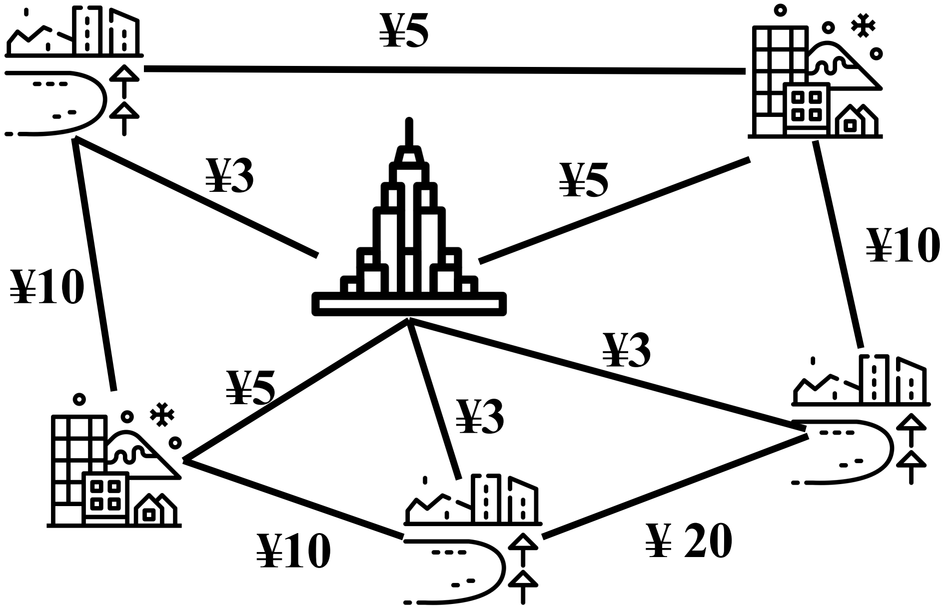



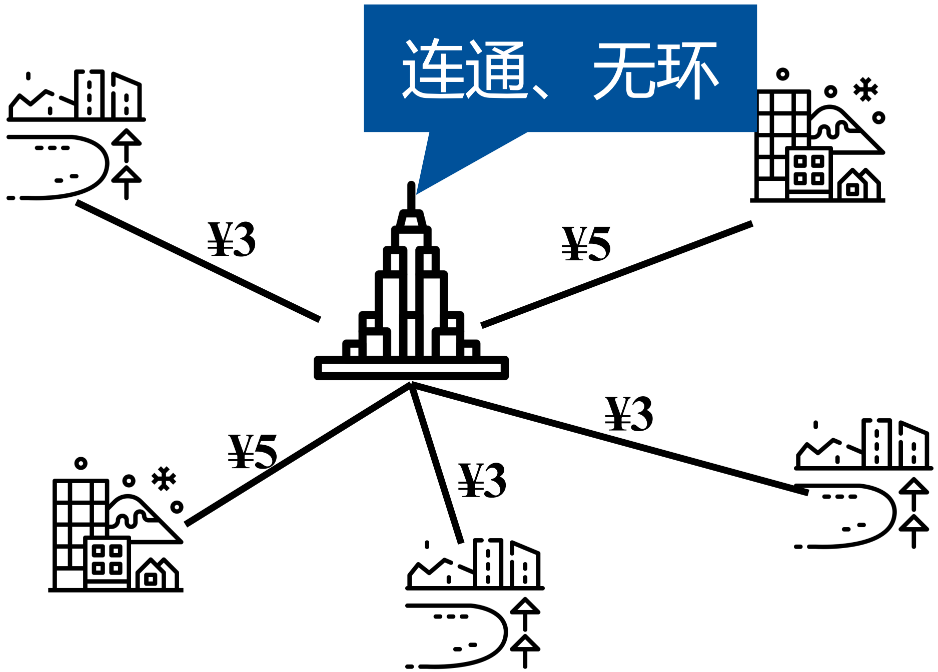

问题：连通各城市的最小花费是多少？

权重最小的连通生成子图

# 图的概念：生成树

- 生成树(Spanning Tree)
  - 图$T' = < V', E' >$是无向图 $G$ 的一个生成子图，并且是连通、无环路的(树)



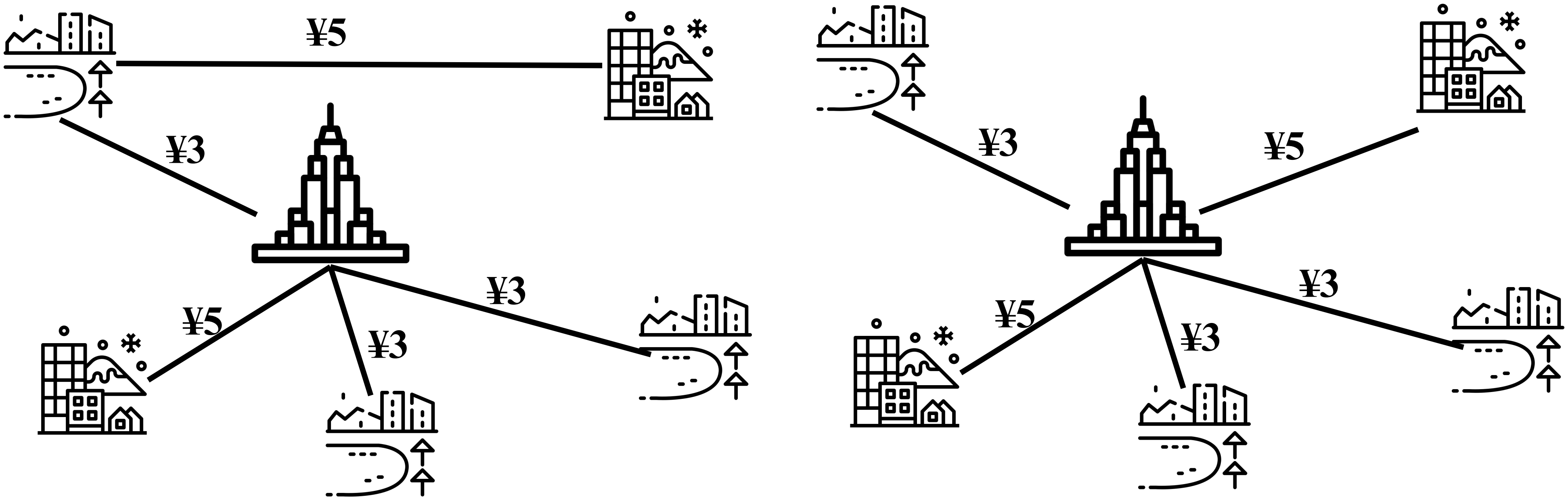




问题：连通各城市的最小花费是多少？

权重最小的生成树

# 图的概念：生成树

- 生成树(**Spanning Tree**)
  - 图$T' =< V',E' >$是无向图 $G$ 的一个生成子图，并且是连通、无环路的(树)
- 权重最小的生成树可能不唯一！



# 问题定义

## 最小生成树问题

**Minimum Spanning Tree Problem**

输入

- 连通无向图 $\boldsymbol{G} = < \boldsymbol{V}, \boldsymbol{E}, \boldsymbol{W} >$, 其中 $\boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{W}$ 表示边 $(\boldsymbol{u}, \boldsymbol{v})$的权重

# 问题定义

## 最小生成树问题

**Minimum Spanning Tree Problem**

输入

- 连通无向图 $\boldsymbol{G} =< \boldsymbol{V}, \boldsymbol{E}, \boldsymbol{W} >$, 其中 $\boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{W}$ 表示边 $(\boldsymbol{u}, \boldsymbol{v})$的权重

输出

- 图$\boldsymbol{G}$的最小生成树$\boldsymbol{T} =< \boldsymbol{V}_{\boldsymbol{T}}, \boldsymbol{E}_{\boldsymbol{T}} >$

# 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$T =< V_T, E_T >$

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

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

# 问题定义

**最小生成树问题**

**Minimum Spanning Tree Problem**

输入

- 连通无向图 $\boldsymbol{G} =< \boldsymbol{V}, \boldsymbol{E}, \boldsymbol{W} >$, 其中 $\boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{W}$ 表示边 $(\boldsymbol{u}, \boldsymbol{v})$的权重

输出

- 图$\boldsymbol{G}$的最小生成树$\boldsymbol{T} =< \boldsymbol{V_T}, \boldsymbol{E_T} >$

优化目标

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

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

# 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

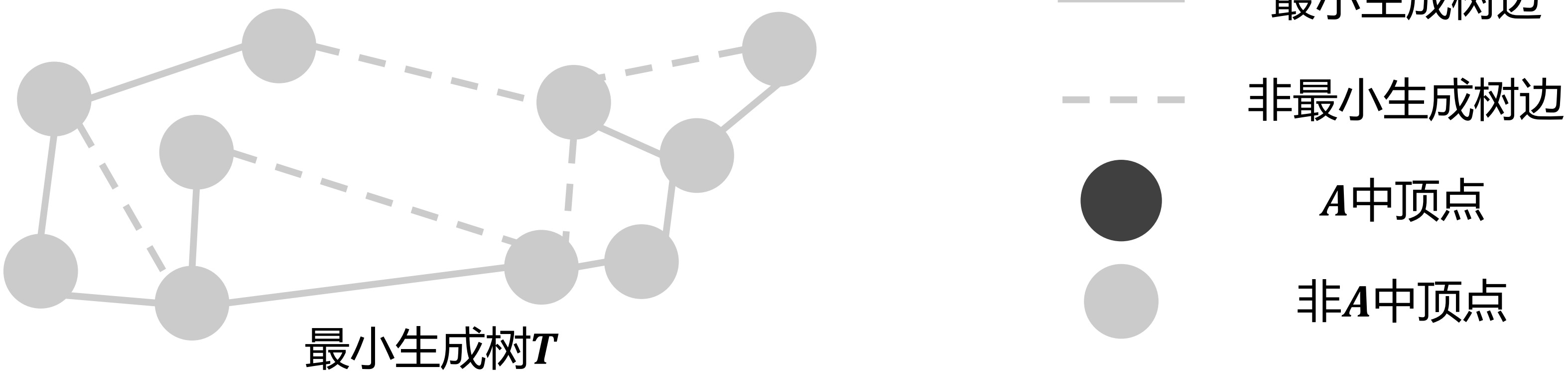
Prim算法

算法实例

算法分析

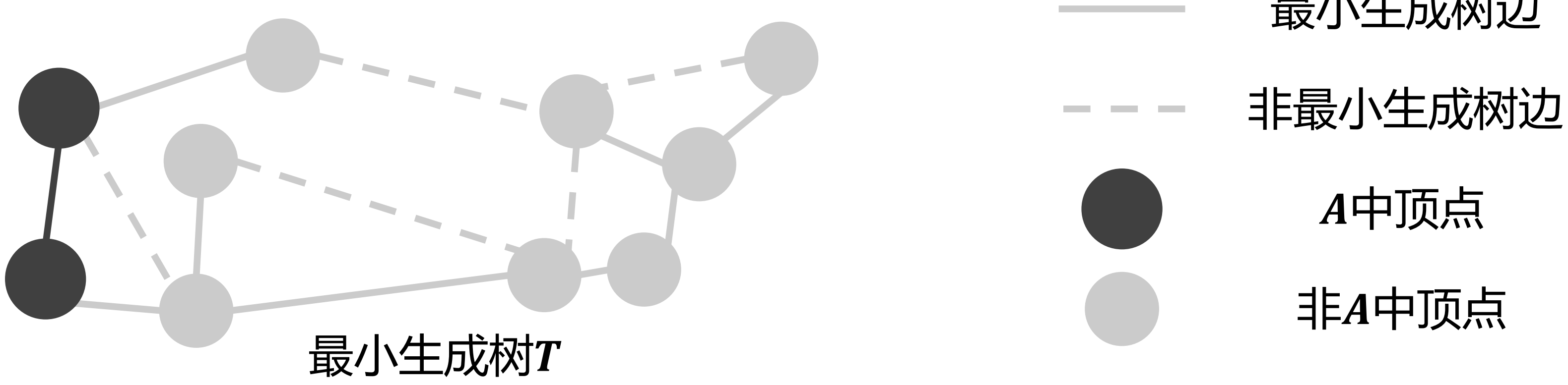
# 通用框架

- 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



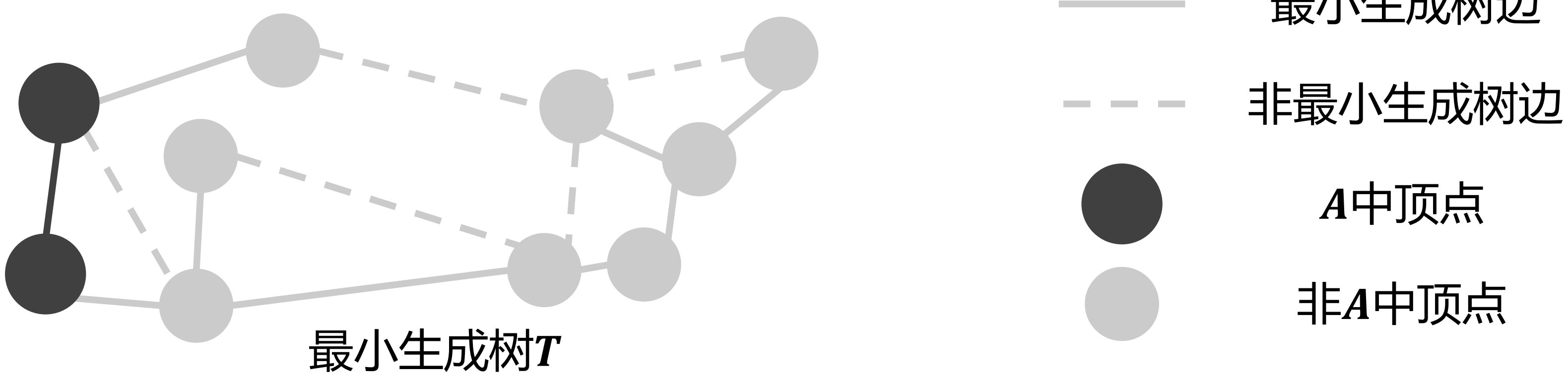
# 通用框架

- 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 $\boldsymbol{A}$，边集 $\boldsymbol{A}$ 可逐步扩展为最小生成树
  - 每次向边集 $\boldsymbol{A}$ 中新增加一条边



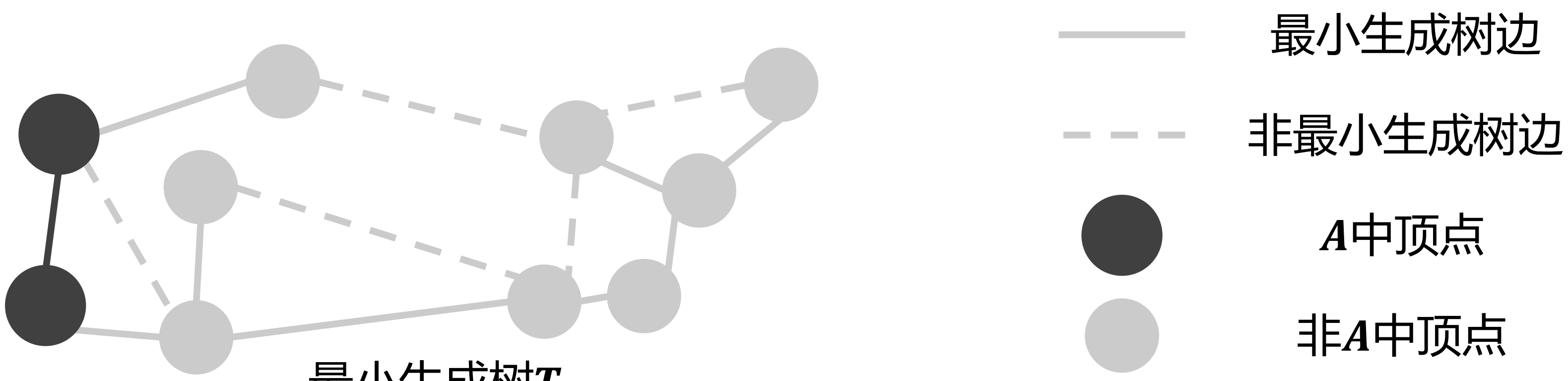
# 通用框架

- 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



# 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 $\boldsymbol{A}$，边集 $\boldsymbol{A}$ 可逐步扩展为最小生成树
  - 每次向边集 $\boldsymbol{A}$ 中新增加一条边
    - 需保证边集 $\boldsymbol{A}$ 仍是一个无环图
    - 需保证边集 $\boldsymbol{A}$ 仍是最小生成树的子集

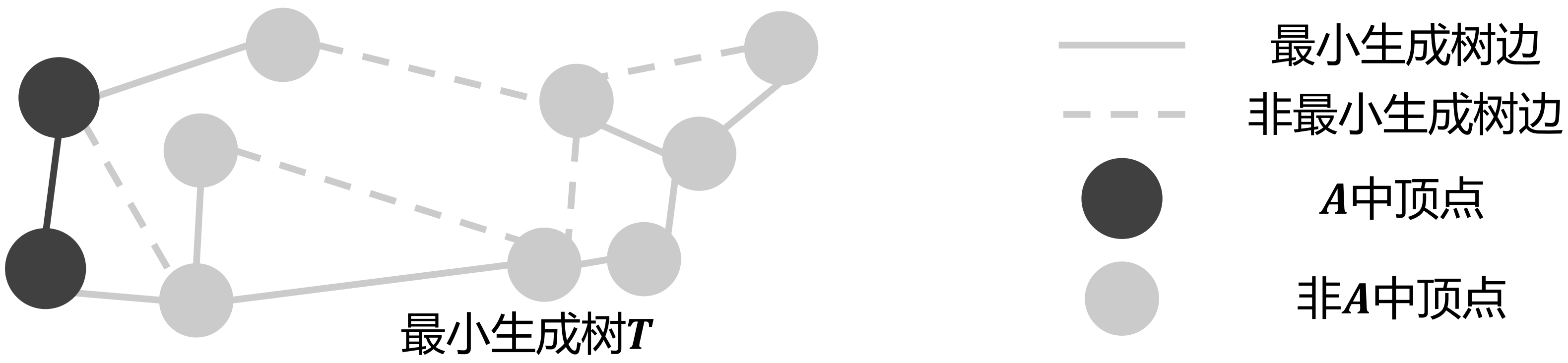



最小生成树$T$

问题：如何保证边集 $\boldsymbol{A}$ 仍是最小生成树的子集？

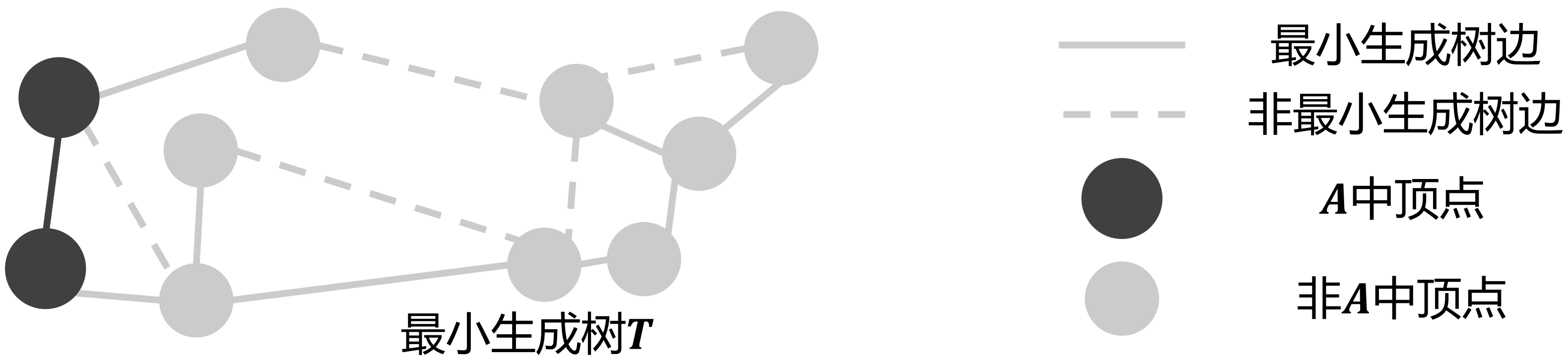
# 相关概念

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



# 相关概念

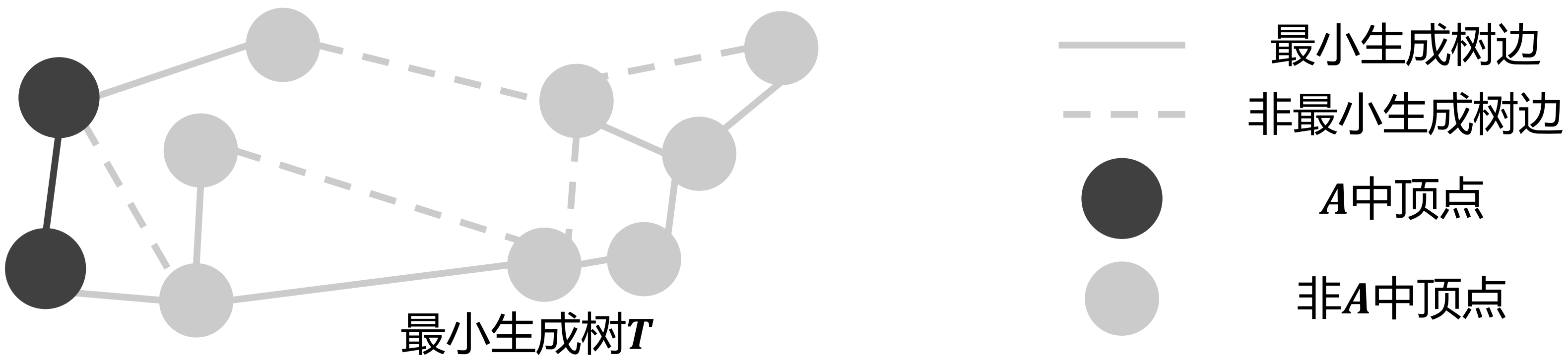
- 安全边(Safe Edge)
  - $\boldsymbol{A}$ 是某棵最小生成树 $\boldsymbol{T}$ 边的子集，$\boldsymbol{A} \subseteq \boldsymbol{T}$
  - $\boldsymbol{A} \cup \{(\boldsymbol{u}, \boldsymbol{v})\}$仍是 $\boldsymbol{T}$ 边的一个子集，则称 $(\boldsymbol{u}, \boldsymbol{v})$ 是 $\boldsymbol{A}$ 的安全边




若每次向边集 $\boldsymbol{A}$ 中新增安全边，可保证边集 $\boldsymbol{A}$ 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

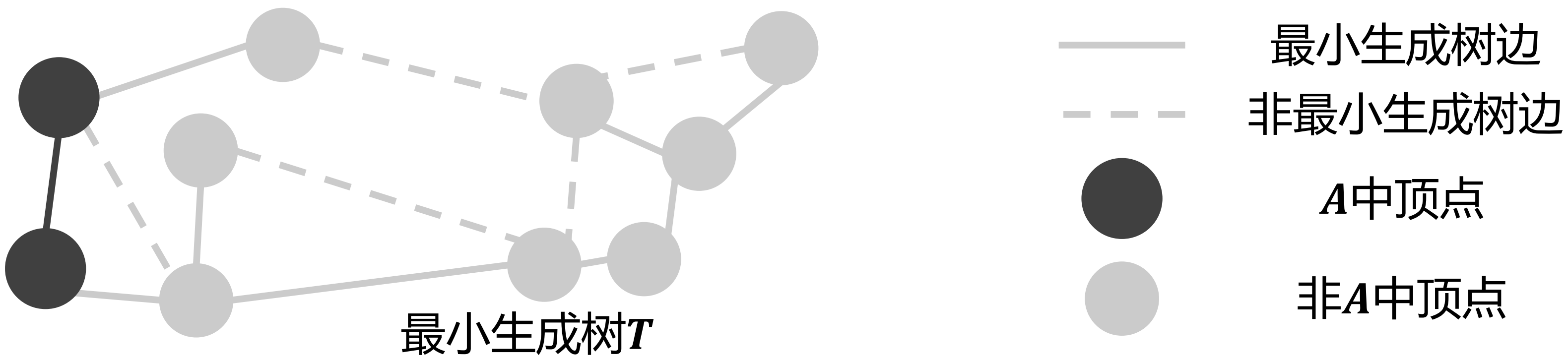



- Generic-MST($G$)

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



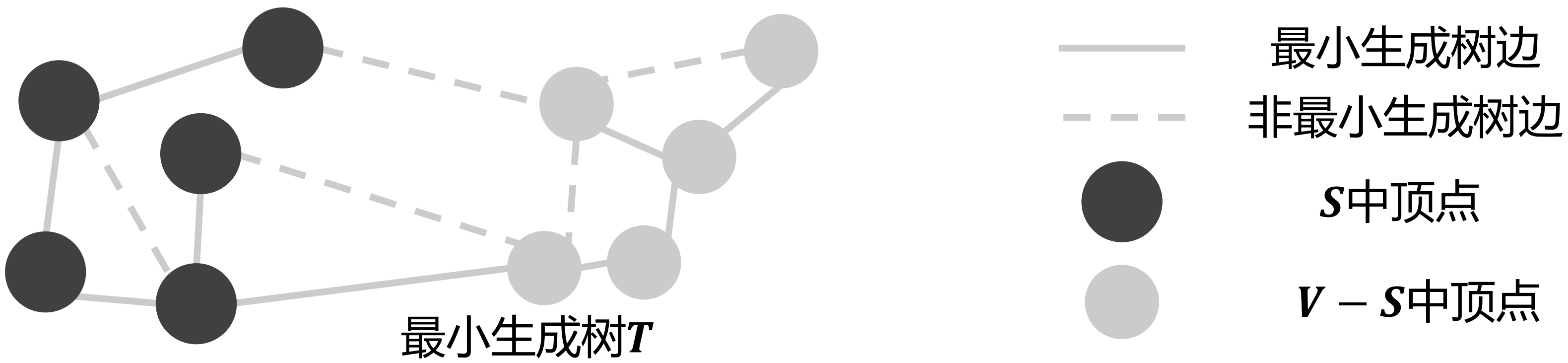

- Generic-MST($G$)

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

问题：如何有效辨识安全边?

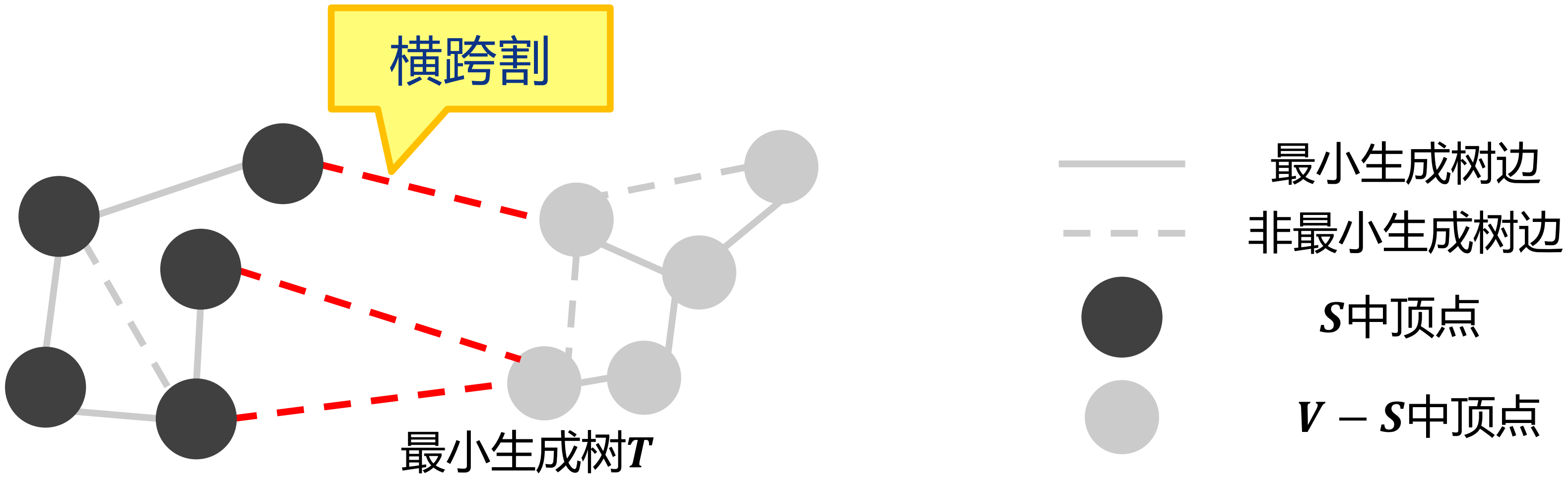
# 相关概念

- 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



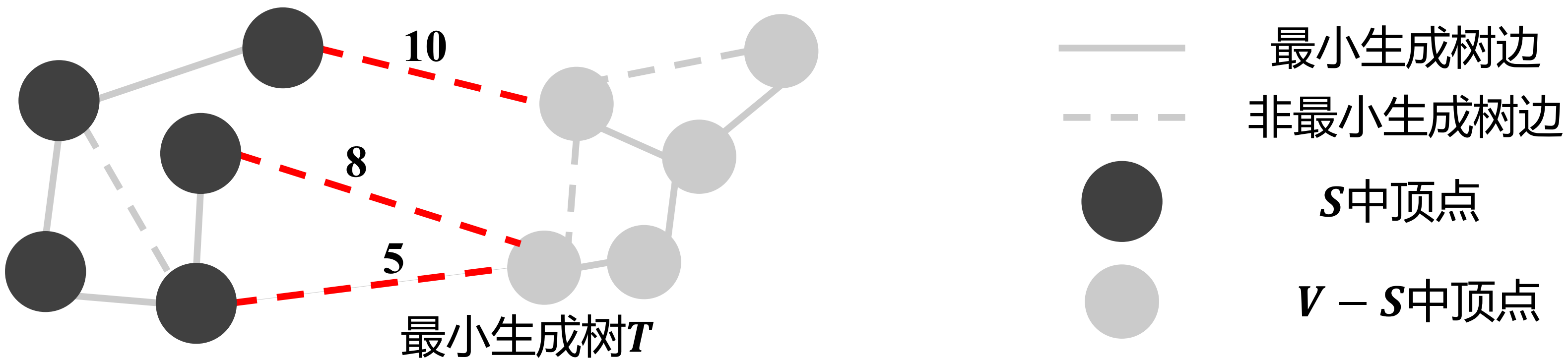
# 相关概念

- 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$(S, V - S)$



# 相关概念

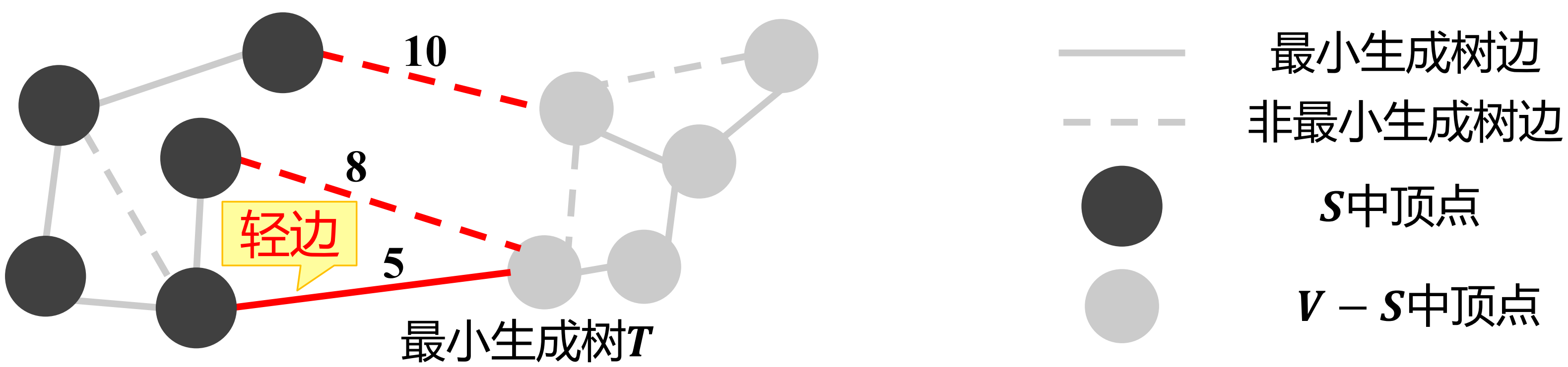
- 割(Cut)
  - 图$G=<V,E>$是一个连通无向图，割$(S,V-S)$将图 $G$的顶点集 $V$划分为两部分
- 横跨(Cross)
  - 给定割$(S,V-S)$和边 $(u,v)$，$u\in S,\ v\in V-S$，称边 $(u,v)$ 横跨割$(S,V-S)$
- 轻边(Light Edge)




最小生成树$T$

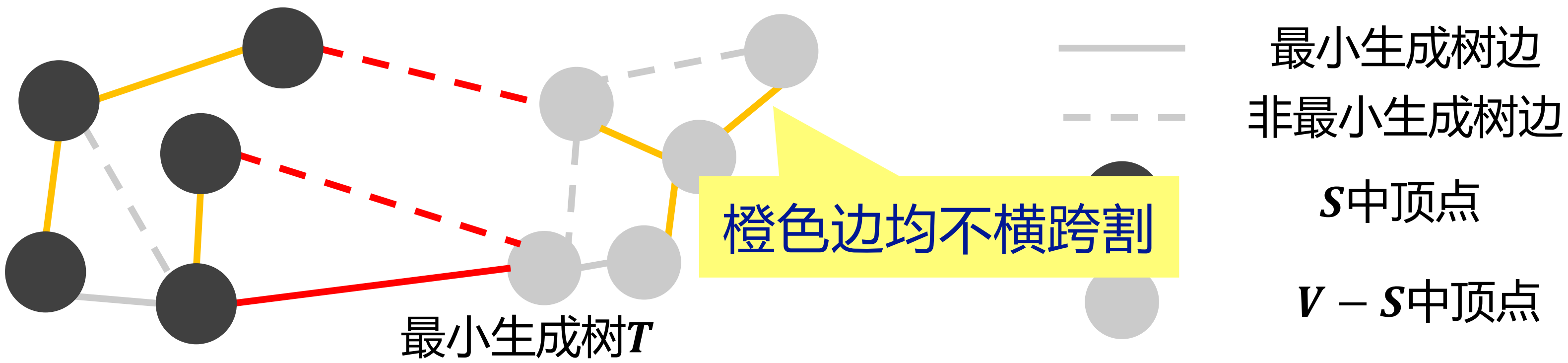
# 相关概念

- 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



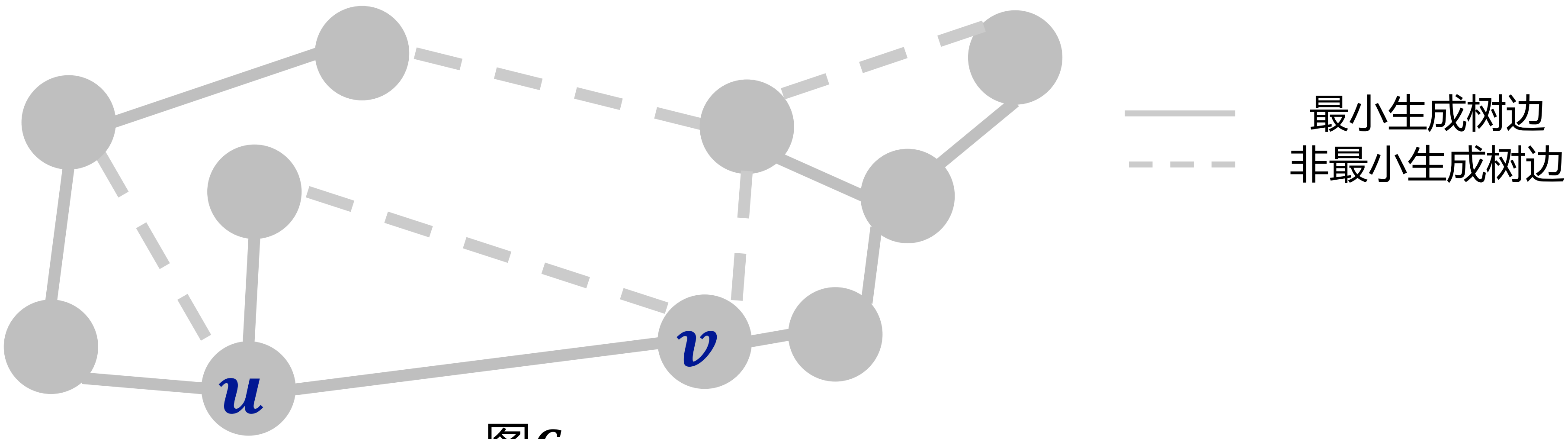
# 相关概念

- 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

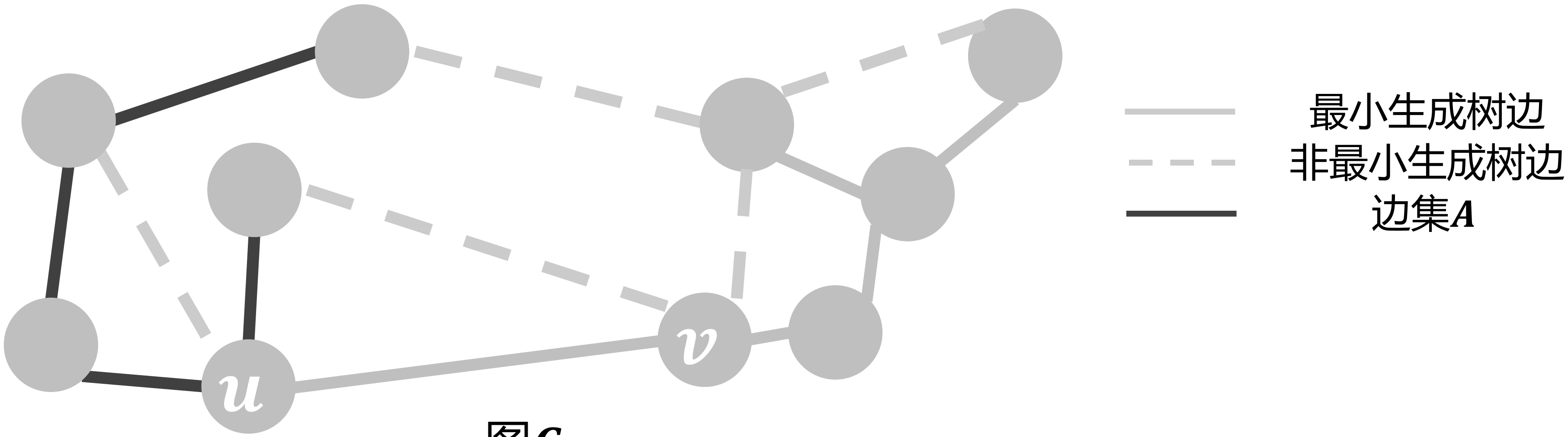
- 给定图$G=<V,E>$是一个带权的连通无向图




图$G$

# 安全边辨识定理

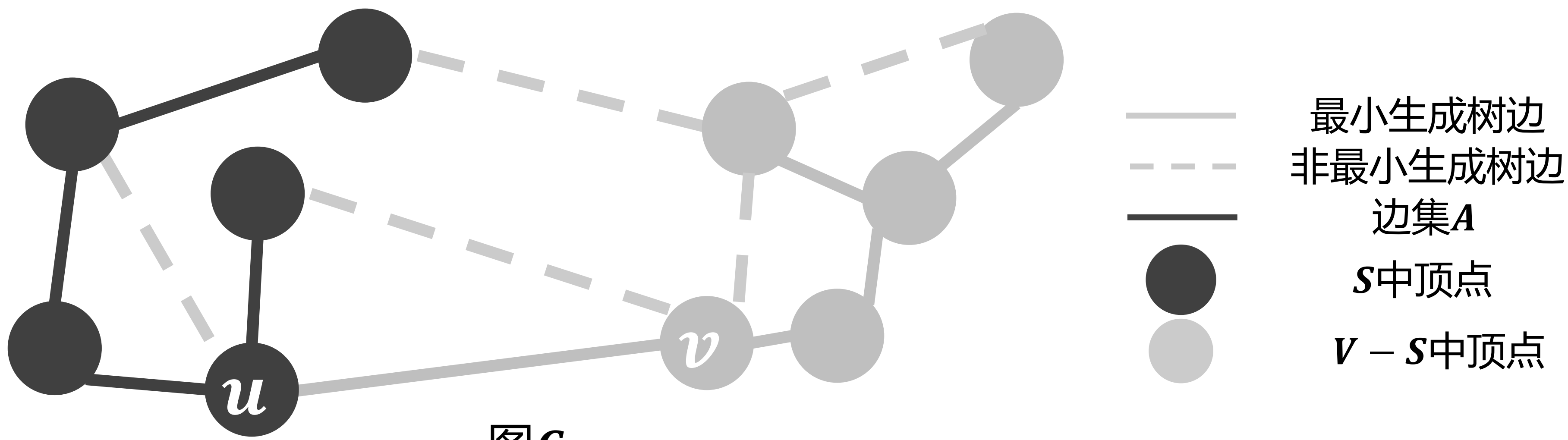
- 给定图$G=<V,E>$是一个带权的连通无向图，令$A$为边集$E$的一个子集，且$A$包含在图$G$的某棵最小生成树中




图$G$

# 安全边辨识定理

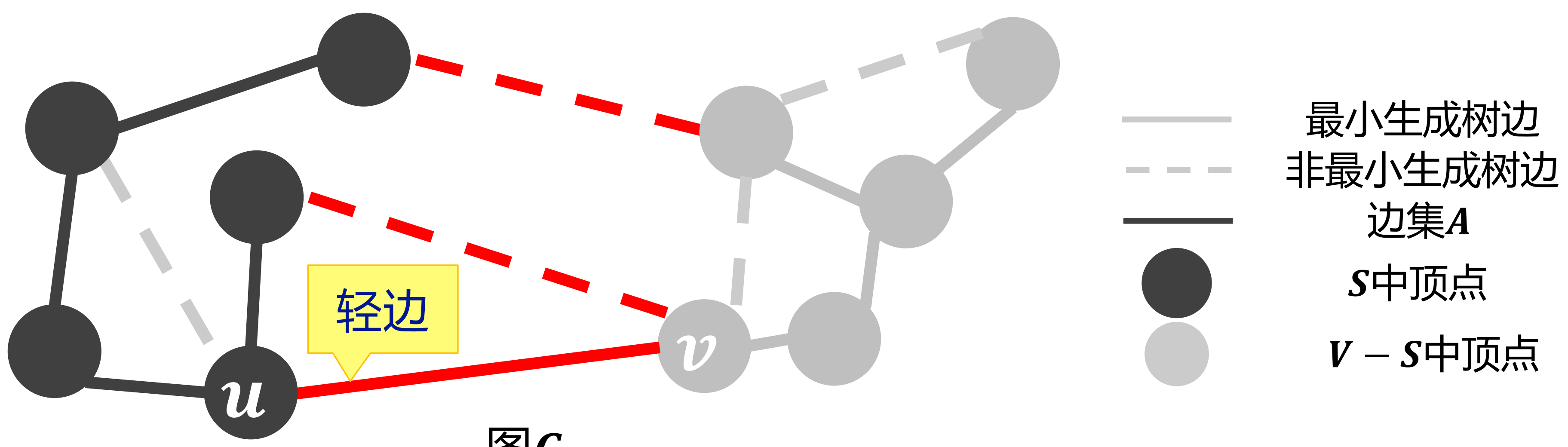
- 给定图$G=<V,E>$是一个带权的连通无向图，令$A$为边集$E$的一个子集，且$A$包含在图$G$的某棵最小生成树中
  - 若割$(S,V-S)$是图$G$中不妨害边集$A$的任意割




图$G$

# 安全边辨识定理

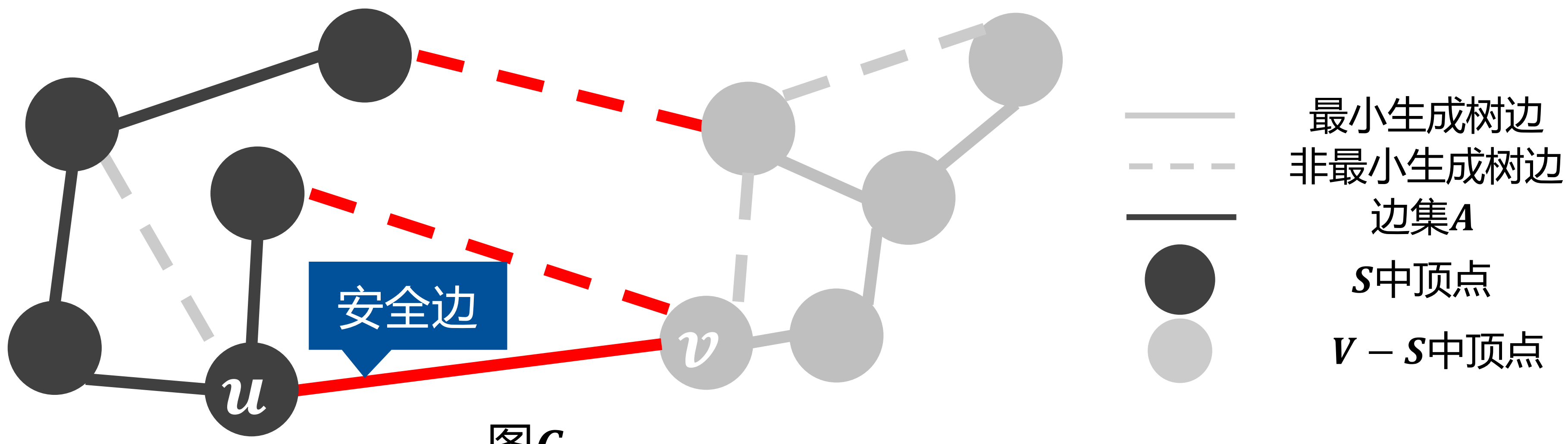
- 给定图$\boldsymbol{G} =< \boldsymbol{V}, \boldsymbol{E} >$是一个带权的连通无向图，令$\boldsymbol{A}$为边集$\boldsymbol{E}$的一个子集，且$\boldsymbol{A}$包含在图$\boldsymbol{G}$的某棵最小生成树中
  - 若割$(\boldsymbol{S}, \boldsymbol{V}-\boldsymbol{S})$是图$\boldsymbol{G}$中不妨害边集$\boldsymbol{A}$的任意割，且$(\boldsymbol{u}, \boldsymbol{v})$是横跨该割的轻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




图$\boldsymbol{G}$

# 安全边辨识定理

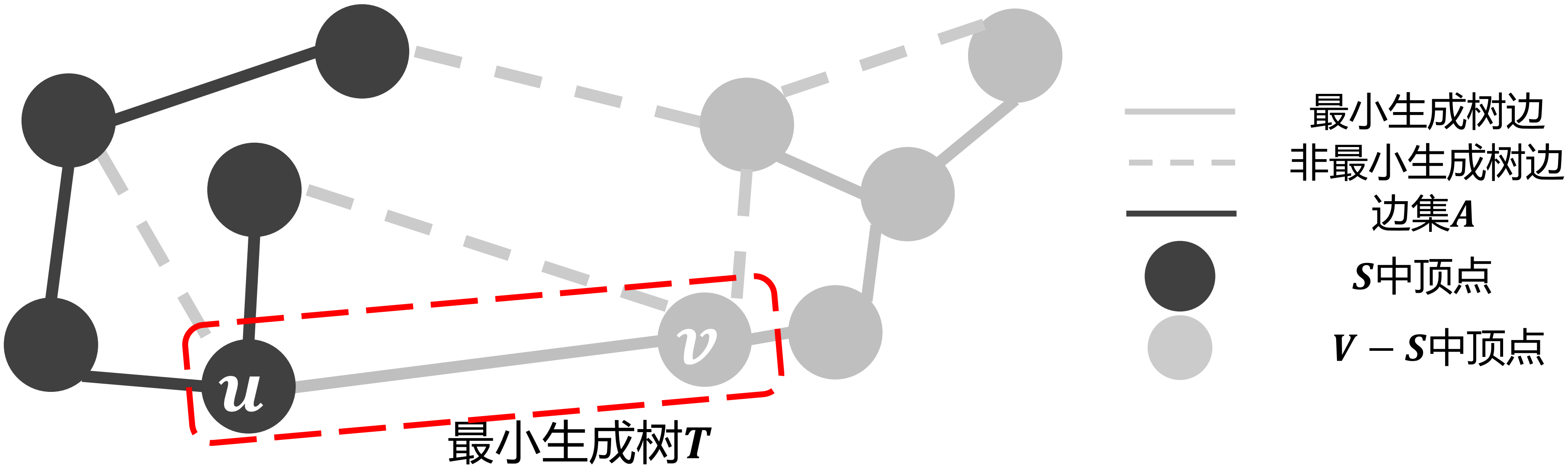
- 给定图$G=<V,E>$是一个带权的连通无向图，令$A$为边集$E$的一个子集，且$A$包含在图$G$的某棵最小生成树中
  - 若割$(S,V-S)$是图$G$中不妨害边集$A$的任意割，且$(u,v)$是横跨该割的轻边
  - 则对于边集$A$，边$(u,v)$是其安全边




图$G$

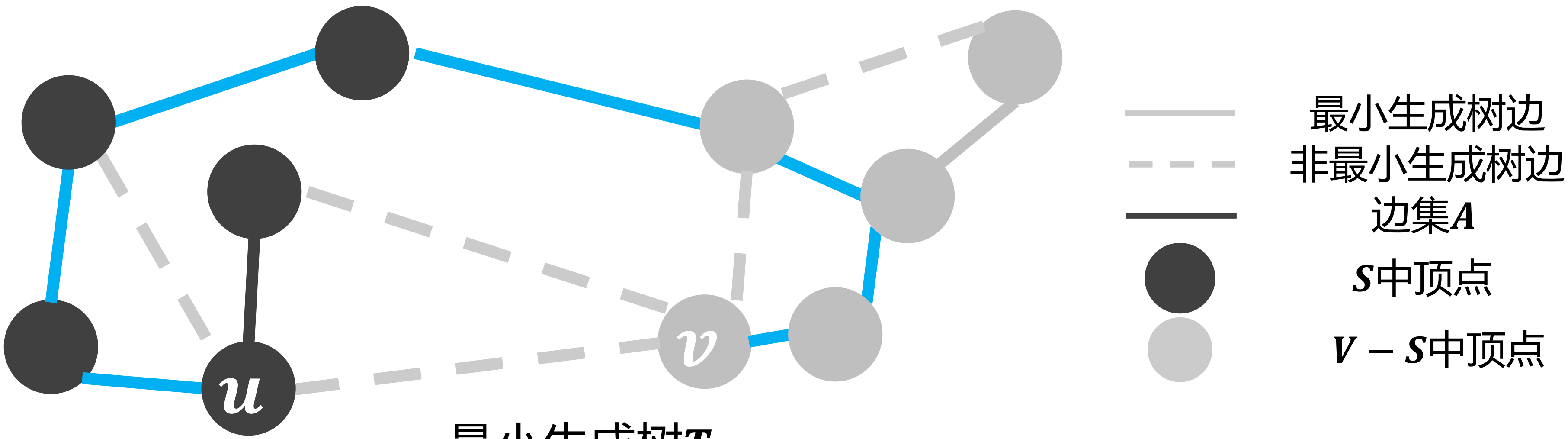
# 安全边辨识定理

- 证明
  - 若$(\boldsymbol{u},\boldsymbol{v}) \in \boldsymbol{T}$，由于$\boldsymbol{A} \subseteq \boldsymbol{T}$，则$\boldsymbol{A} \cup \{(\boldsymbol{u},\boldsymbol{v})\} \subseteq \boldsymbol{T}$，由安全边定义可证



# 安全边辨识定理

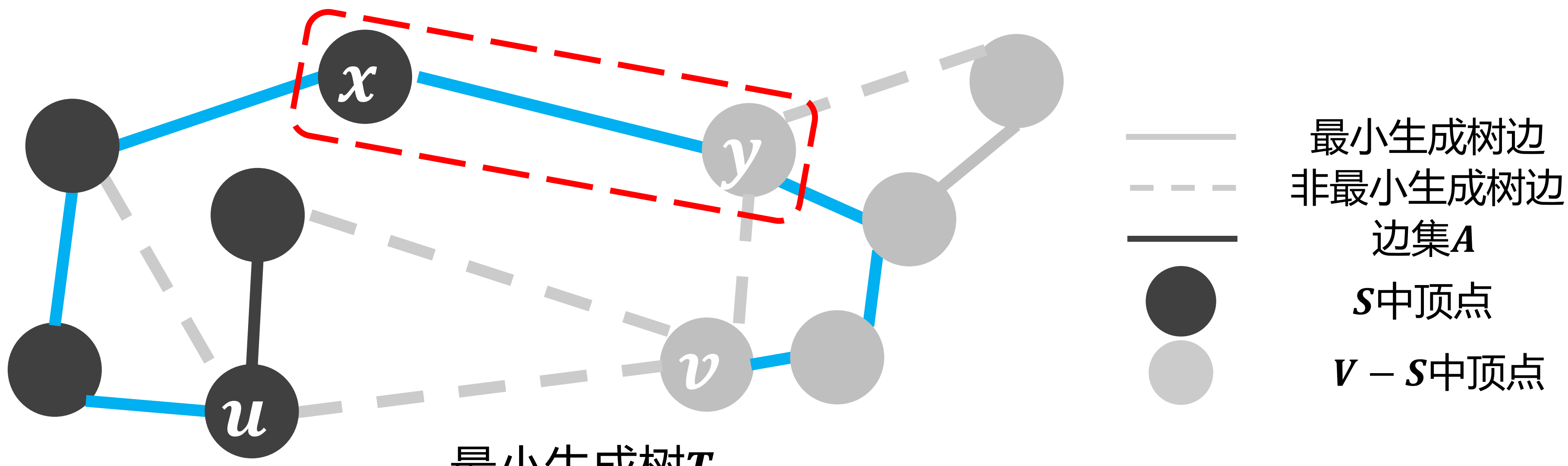
- 证明
  - 若$(\boldsymbol{u},\boldsymbol{v}) \in \boldsymbol{T}$，由于$\boldsymbol{A} \subseteq \boldsymbol{T}$，则$\boldsymbol{A} \cup \{(\boldsymbol{u},\boldsymbol{v})\} \subseteq \boldsymbol{T}$，由安全边定义可证
  - 若$(\boldsymbol{u},\boldsymbol{v}) \notin \boldsymbol{T}$，则$\boldsymbol{T}$中必存在$\boldsymbol{u}$到$\boldsymbol{v}$的路径$\boldsymbol{P}$




最小生成树$\boldsymbol{T}$

# 安全边辨识定理

- 证明
  - 若$(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{T}$，由于$\boldsymbol{A} \subseteq \boldsymbol{T}$，则$\boldsymbol{A} \cup \{(\boldsymbol{u}, \boldsymbol{v})\} \subseteq \boldsymbol{T}$，由安全边定义可证
  - 若$(\boldsymbol{u}, \boldsymbol{v}) \notin \boldsymbol{T}$，则$\boldsymbol{T}$中必存在$\boldsymbol{u}$到$\boldsymbol{v}$的路径$\boldsymbol{P}$
    - 不妨设路径$\boldsymbol{P}$中，横跨割$(\boldsymbol{S}, \boldsymbol{V}-\boldsymbol{S})$的一条边为$(\boldsymbol{x}, \boldsymbol{y})$

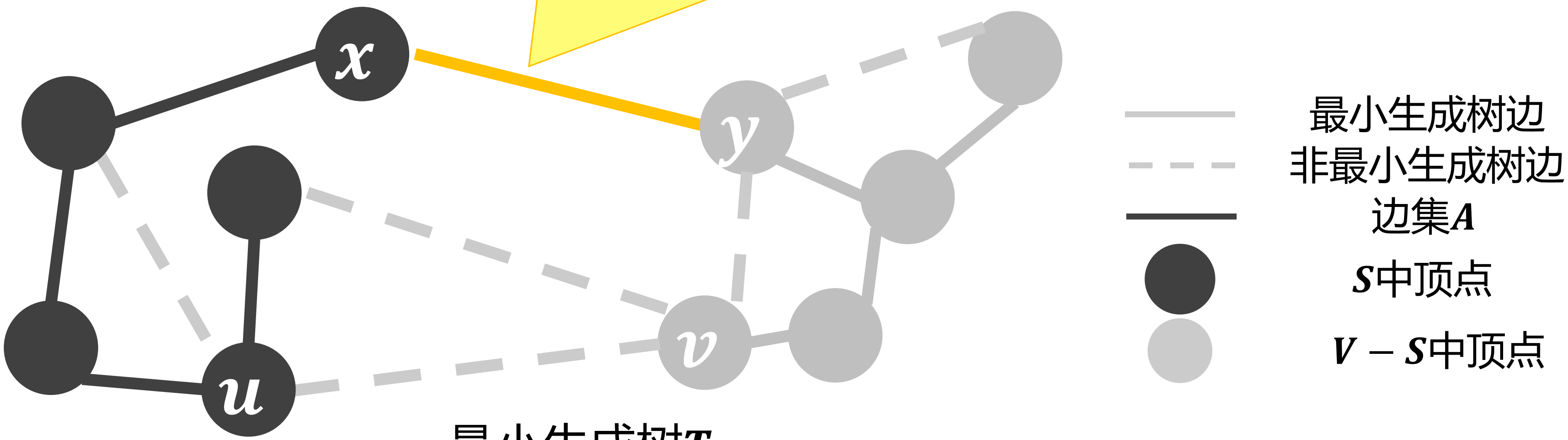



最小生成树$\boldsymbol{T}$

# 安全边辨识定理

- 证明
  - 若$(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{T}$，由于$\boldsymbol{A} \subseteq \boldsymbol{T}$，则$\boldsymbol{A} \cup \{(\boldsymbol{u}, \boldsymbol{v})\} \subseteq \boldsymbol{T}$，由安全边定义可证
  - 若$(\boldsymbol{u}, \boldsymbol{v}) \notin \boldsymbol{T}$，则$\boldsymbol{T}$中必存在$\boldsymbol{u}$到$\boldsymbol{v}$的路径$\boldsymbol{P}$
    - 不妨设路径$\boldsymbol{P}$中，横跨割$(\boldsymbol{S}, \boldsymbol{V}-\boldsymbol{S})$的一条边为$(\boldsymbol{x}, \boldsymbol{y})$

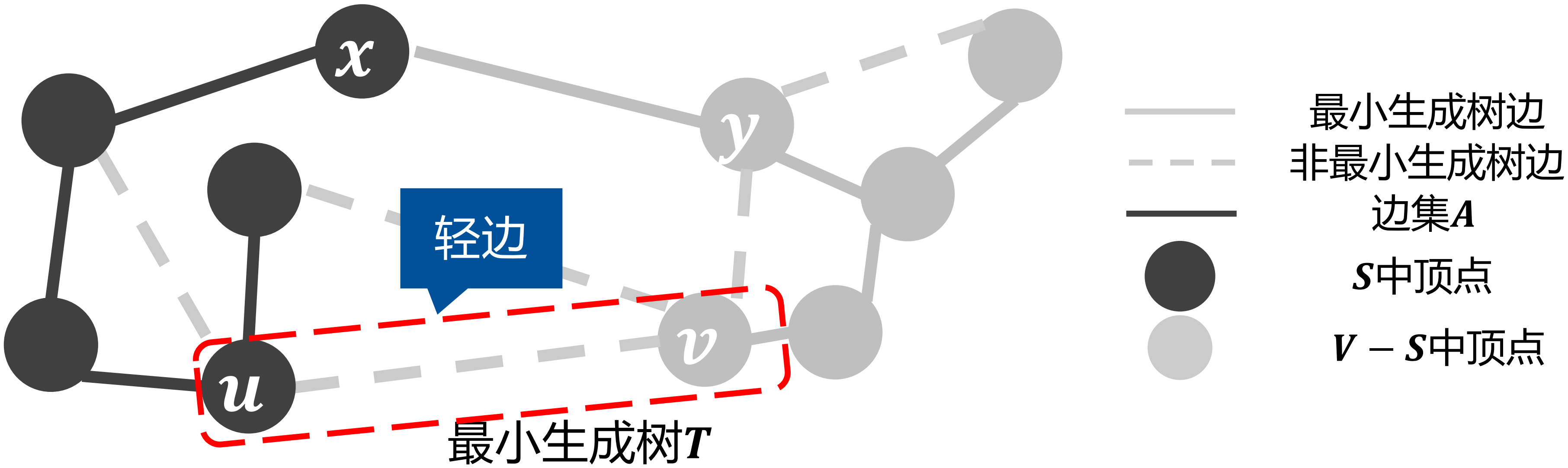
割不妨害边集 $\boldsymbol{A}$，$(\boldsymbol{x}, \boldsymbol{y}) \notin \boldsymbol{A}$




最小生成树$\boldsymbol{T}$

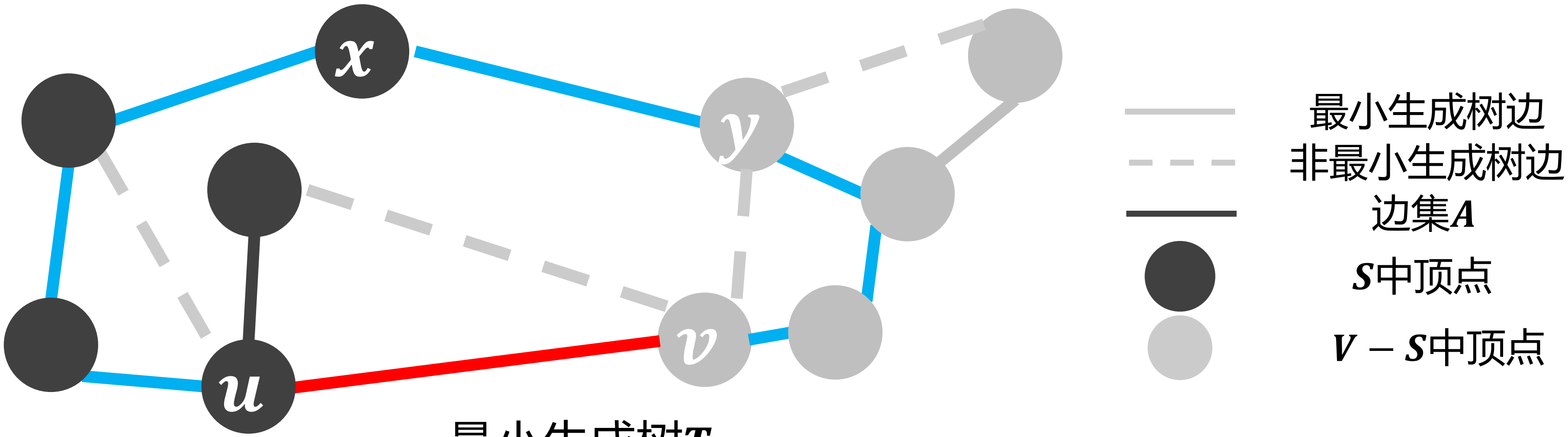
# 安全边辨识定理

- 证明
  - 若$(\boldsymbol{u},\boldsymbol{v}) \in \boldsymbol{T}$，由于$\boldsymbol{A} \subseteq \boldsymbol{T}$，则$\boldsymbol{A} \cup \{(\boldsymbol{u},\boldsymbol{v})\} \subseteq \boldsymbol{T}$，由安全边定义可证
  - 若$(\boldsymbol{u},\boldsymbol{v}) \notin \boldsymbol{T}$，则$\boldsymbol{T}$中必存在$\boldsymbol{u}$到$\boldsymbol{v}$的路径$\boldsymbol{P}$
    - 不妨设路径$\boldsymbol{P}$中，横跨割$(\boldsymbol{S},\boldsymbol{V}-\boldsymbol{S})$的一条边为$(\boldsymbol{x},\boldsymbol{y})$
    - 边$(\boldsymbol{u},\boldsymbol{v})$是横跨割的轻边，所以$\boldsymbol{w}(\boldsymbol{u},\boldsymbol{v}) \leq \boldsymbol{w}(\boldsymbol{x},\boldsymbol{y})$



# 安全边辨识定理

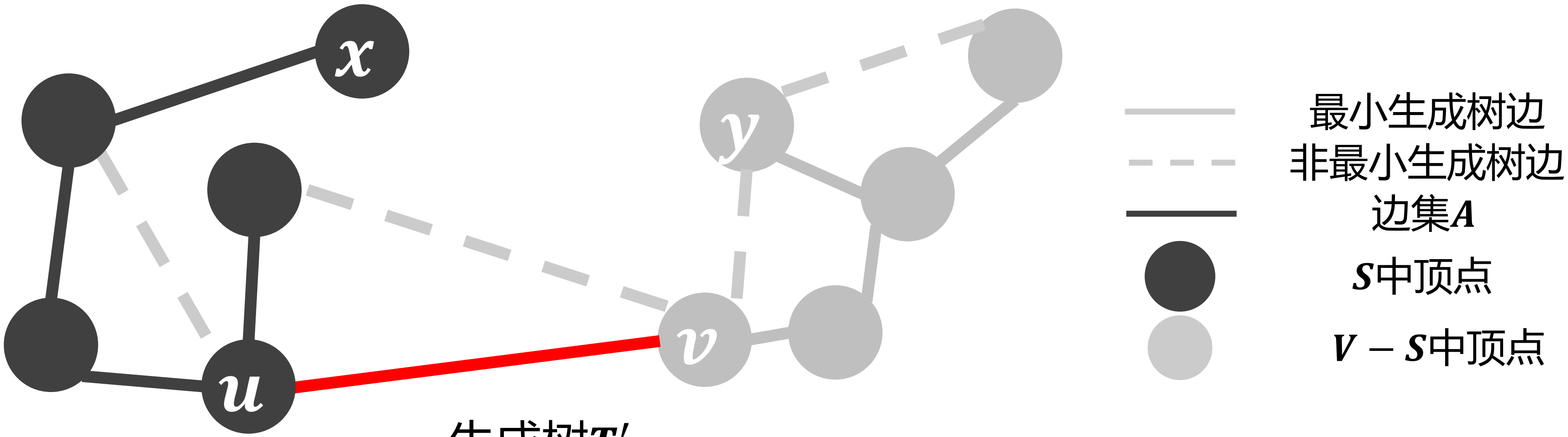
- 证明
  - 若$(\boldsymbol{u},\boldsymbol{v}) \in \boldsymbol{T}$，由于$\boldsymbol{A} \subseteq \boldsymbol{T}$，则$\boldsymbol{A} \cup \{(\boldsymbol{u},\boldsymbol{v})\} \subseteq \boldsymbol{T}$，由安全边定义可证
  - 若$(\boldsymbol{u},\boldsymbol{v}) \notin \boldsymbol{T}$，则$\boldsymbol{T}$中必存在$\boldsymbol{u}$到$\boldsymbol{v}$的路径$\boldsymbol{P}$
    - 不妨设路径$\boldsymbol{P}$中，横跨割$(\boldsymbol{S},\boldsymbol{V}-\boldsymbol{S})$的一条边为$(\boldsymbol{x},\boldsymbol{y})$
    - 边$(\boldsymbol{u},\boldsymbol{v})$是横跨割的轻边，所以$\boldsymbol{w}(\boldsymbol{u},\boldsymbol{v}) \leq \boldsymbol{w}(\boldsymbol{x},\boldsymbol{y})$
    - 将边$(\boldsymbol{u},\boldsymbol{v})$加入到$\boldsymbol{T}$中会形成环路




最小生成树$\boldsymbol{T}$

# 安全边辨识定理

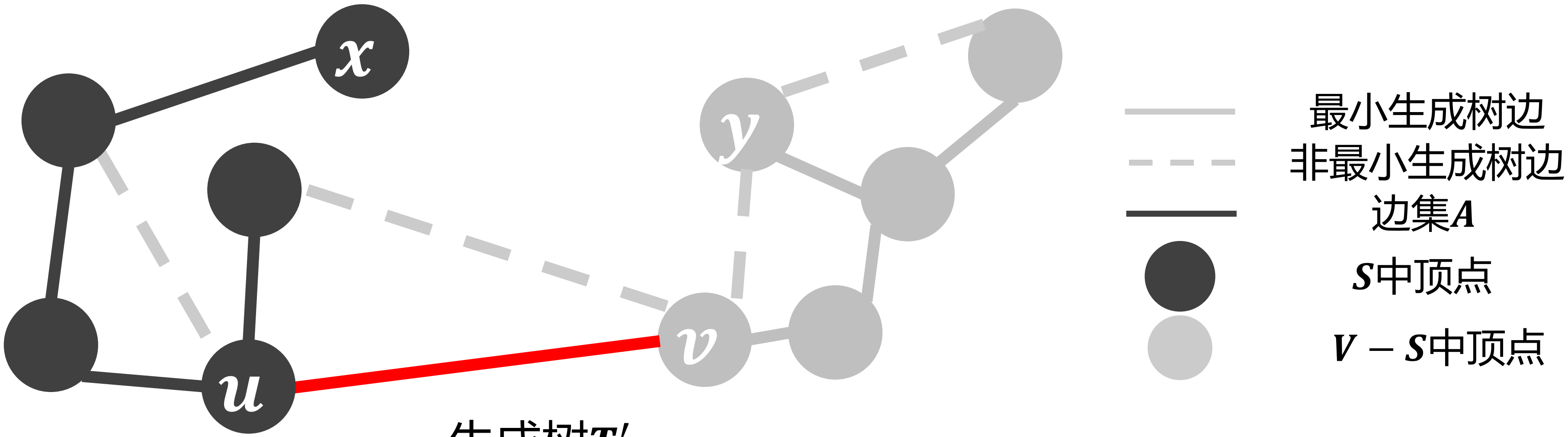
- 证明
  - 若$(\boldsymbol{u},\boldsymbol{v}) \in \boldsymbol{T}$，由于$\boldsymbol{A} \subseteq \boldsymbol{T}$，则$\boldsymbol{A} \cup \{(\boldsymbol{u},\boldsymbol{v})\} \subseteq \boldsymbol{T}$，由安全边定义可证
  - 若$(\boldsymbol{u},\boldsymbol{v}) \notin \boldsymbol{T}$，则$\boldsymbol{T}$中必存在$\boldsymbol{u}$到$\boldsymbol{v}$的路径$\boldsymbol{P}$
    - 不妨设路径$\boldsymbol{P}$中，横跨割$(\boldsymbol{S},\boldsymbol{V}-\boldsymbol{S})$的一条边为$(\boldsymbol{x},\boldsymbol{y})$
    - 边$(\boldsymbol{u},\boldsymbol{v})$是横跨割的轻边，所以$\boldsymbol{w}(\boldsymbol{u},\boldsymbol{v}) \leq \boldsymbol{w}(\boldsymbol{x},\boldsymbol{y})$
    - 将边$(\boldsymbol{u},\boldsymbol{v})$加入到$\boldsymbol{T}$中会形成环路，再去掉边$(\boldsymbol{x},\boldsymbol{y})$会形成另一棵树$\boldsymbol{T}'$




生成树$\boldsymbol{T}'$

# 安全边辨识定理

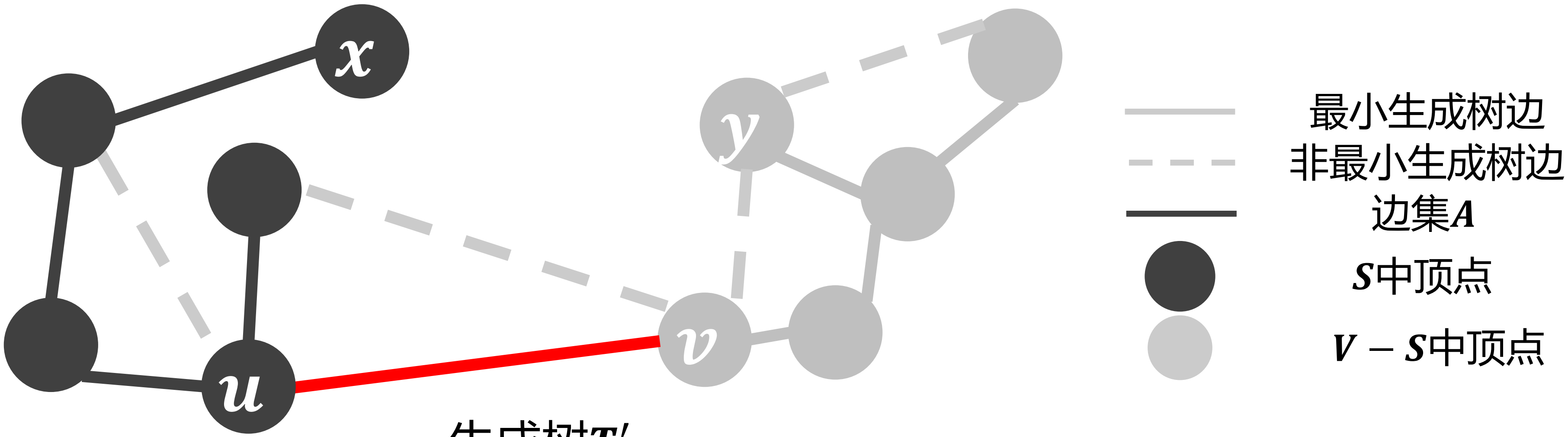
- 证明
  - 若$(\boldsymbol{u},\boldsymbol{v}) \in \boldsymbol{T}$，由于$\boldsymbol{A} \subseteq \boldsymbol{T}$，则$\boldsymbol{A} \cup \{(\boldsymbol{u},\boldsymbol{v})\} \subseteq \boldsymbol{T}$，由安全边定义可证
  - 若$(\boldsymbol{u},\boldsymbol{v}) \notin \boldsymbol{T}$，则$\boldsymbol{T}$中必存在$\boldsymbol{u}$到$\boldsymbol{v}$的路径$\boldsymbol{P}$
    - 不妨设路径$\boldsymbol{P}$中，横跨割$(\boldsymbol{S},\boldsymbol{V}-\boldsymbol{S})$的一条边为$(\boldsymbol{x},\boldsymbol{y})$
    - 边$(\boldsymbol{u},\boldsymbol{v})$是横跨割的轻边，所以$\boldsymbol{w}(\boldsymbol{u},\boldsymbol{v}) \leq \boldsymbol{w}(\boldsymbol{x},\boldsymbol{y})$
    - 将边$(\boldsymbol{u},\boldsymbol{v})$加入到$\boldsymbol{T}$中会形成环路，再去掉边$(\boldsymbol{x},\boldsymbol{y})$会形成另一棵树$\boldsymbol{T}'$
    - $\boldsymbol{w}(\boldsymbol{T}') = \boldsymbol{w}(\boldsymbol{T}) + \boldsymbol{w}(\boldsymbol{u},\boldsymbol{v}) - \boldsymbol{w}(\boldsymbol{x},\boldsymbol{y})$




生成树$\boldsymbol{T}'$

# 安全边辨识定理

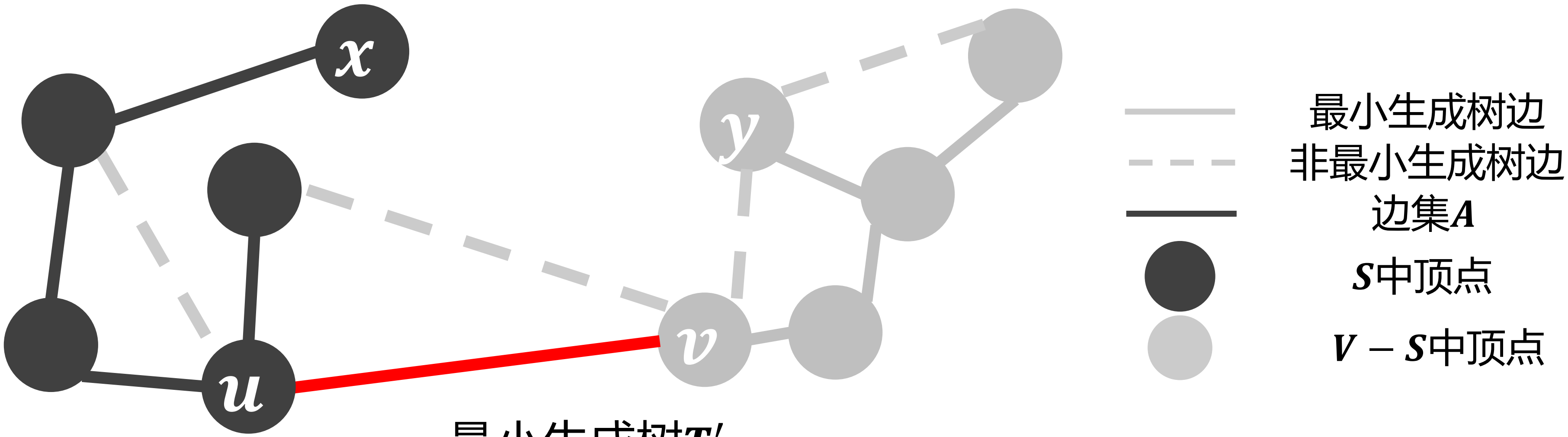
- 证明
  - 若$(\boldsymbol{u},\boldsymbol{v}) \in \boldsymbol{T}$，由于$\boldsymbol{A} \subseteq \boldsymbol{T}$，则$\boldsymbol{A} \cup \{(\boldsymbol{u},\boldsymbol{v})\} \subseteq \boldsymbol{T}$，由安全边定义可证
  - 若$(\boldsymbol{u},\boldsymbol{v}) \notin \boldsymbol{T}$，则$\boldsymbol{T}$中必存在$\boldsymbol{u}$到$\boldsymbol{v}$的路径$\boldsymbol{P}$
    - 不妨设路径$\boldsymbol{P}$中，横跨割$(\boldsymbol{S},\boldsymbol{V}-\boldsymbol{S})$的一条边为$(\boldsymbol{x},\boldsymbol{y})$
    - 边$(\boldsymbol{u},\boldsymbol{v})$是横跨割的轻边，所以$\boldsymbol{w}(\boldsymbol{u},\boldsymbol{v}) \leq \boldsymbol{w}(\boldsymbol{x},\boldsymbol{y})$
    - 将边$(\boldsymbol{u},\boldsymbol{v})$加入到$\boldsymbol{T}$中会形成环路，再去掉边$(\boldsymbol{x},\boldsymbol{y})$会形成另一棵树$\boldsymbol{T}'$
    - $\boldsymbol{w}(\boldsymbol{T}') = \boldsymbol{w}(\boldsymbol{T}) + \boldsymbol{w}(\boldsymbol{u},\boldsymbol{v}) - \boldsymbol{w}(\boldsymbol{x},\boldsymbol{y}) \leq \boldsymbol{w}(\boldsymbol{T})$




生成树$\boldsymbol{T}'$

# 安全边辨识定理

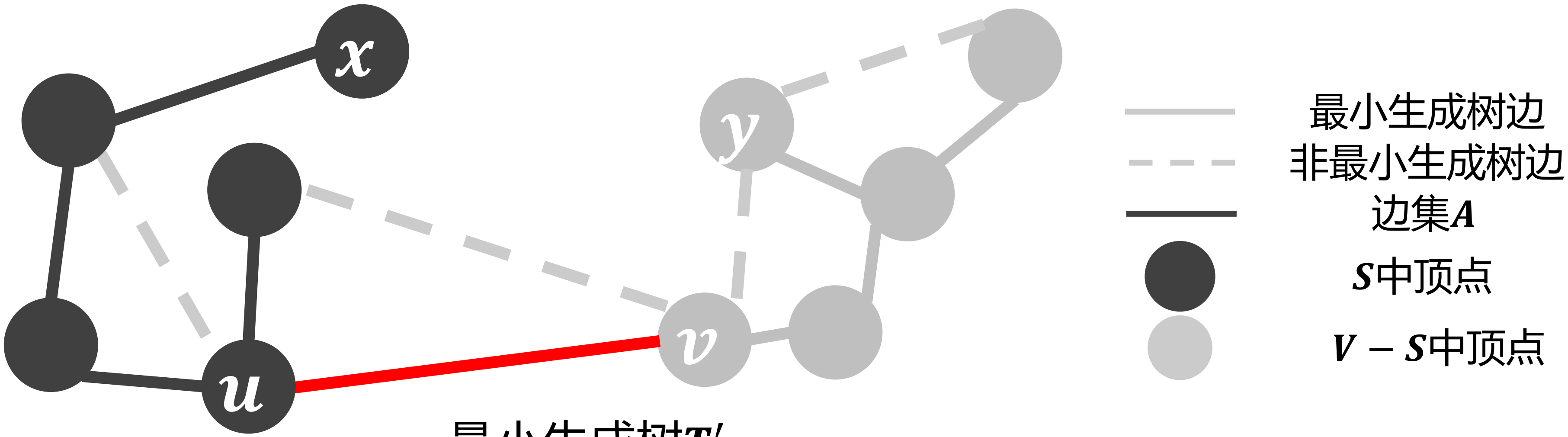
- 证明
  - 若$(\boldsymbol{u},\boldsymbol{v}) \in \boldsymbol{T}$，由于$\boldsymbol{A} \subseteq \boldsymbol{T}$，则$\boldsymbol{A} \cup \{(\boldsymbol{u},\boldsymbol{v})\} \subseteq \boldsymbol{T}$，由安全边定义可证
  - 若$(\boldsymbol{u},\boldsymbol{v}) \notin \boldsymbol{T}$，则$\boldsymbol{T}$中必存在$\boldsymbol{u}$到$\boldsymbol{v}$的路径$\boldsymbol{P}$
    - 不妨设路径$\boldsymbol{P}$中，横跨割$(\boldsymbol{S},\boldsymbol{V}-\boldsymbol{S})$的一条边为$(\boldsymbol{x},\boldsymbol{y})$
    - 边$(\boldsymbol{u},\boldsymbol{v})$是横跨割的轻边，所以$\boldsymbol{w}(\boldsymbol{u},\boldsymbol{v}) \leq \boldsymbol{w}(\boldsymbol{x},\boldsymbol{y})$
    - 将边$(\boldsymbol{u},\boldsymbol{v})$加入到$\boldsymbol{T}$中会形成环路，再去掉边$(\boldsymbol{x},\boldsymbol{y})$会形成另一棵树$\boldsymbol{T}'$
    - $\boldsymbol{w}(\boldsymbol{T}') \leq \boldsymbol{w}(\boldsymbol{T})$，$\boldsymbol{T}'$也是最小生成树




最小生成树$\boldsymbol{T}'$

# 安全边辨识定理

- 证明
  - 若$(\boldsymbol{u},\boldsymbol{v}) \in \boldsymbol{T}$，由于$\boldsymbol{A} \subseteq \boldsymbol{T}$，则$\boldsymbol{A} \cup \{(\boldsymbol{u},\boldsymbol{v})\} \subseteq \boldsymbol{T}$，由安全边定义可证
  - 若$(\boldsymbol{u},\boldsymbol{v}) \notin \boldsymbol{T}$，则$\boldsymbol{T}$中必存在$\boldsymbol{u}$到$\boldsymbol{v}$的路径$\boldsymbol{P}$
    - 不妨设路径$\boldsymbol{P}$中，横跨割$(\boldsymbol{S},\boldsymbol{V}-\boldsymbol{S})$的一条边为$(\boldsymbol{x},\boldsymbol{y})$
    - 边$(\boldsymbol{u},\boldsymbol{v})$是横跨割的轻边，所以$\boldsymbol{w}(\boldsymbol{u},\boldsymbol{v}) \leq \boldsymbol{w}(\boldsymbol{x},\boldsymbol{y})$
    - 将边$(\boldsymbol{u},\boldsymbol{v})$加入到$\boldsymbol{T}$中会形成环路，再去掉边$(\boldsymbol{x},\boldsymbol{y})$会形成另一棵树$\boldsymbol{T}'$
    - $\boldsymbol{w}(\boldsymbol{T}') \leq \boldsymbol{w}(\boldsymbol{T})$，$\boldsymbol{T}'$也是最小生成树，$\boldsymbol{A} \cup \{(\boldsymbol{u},\boldsymbol{v})\} \subseteq \boldsymbol{T}'$，边$(\boldsymbol{u},\boldsymbol{v})$是安全边




最小生成树$\boldsymbol{T}'$

# 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$\boldsymbol{A}$，边集$\boldsymbol{A}$可逐步扩展为最小生成树
  - 每次向边集$\boldsymbol{A}$中新增加一条边
    - 需保证边集$\boldsymbol{A}$仍是一个无环图
    - 需保证边集$\boldsymbol{A}$仍是最小生成树的子集

# 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

# 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？

# 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