

贪心算法

教材：第9章

找零

- 想象你是一名店员在找零，你想要使用最少数量的硬币

- 策略：
 - 总是选择最大且可行的硬币
- 示例：37美分
 - 1一枚25分的硬币（还需12分）
 - 1一角硬币（还差2分）
 - 2一分硬币（2什么？）



这是一个“贪心算法”

永远做出当下看起来最好的选择



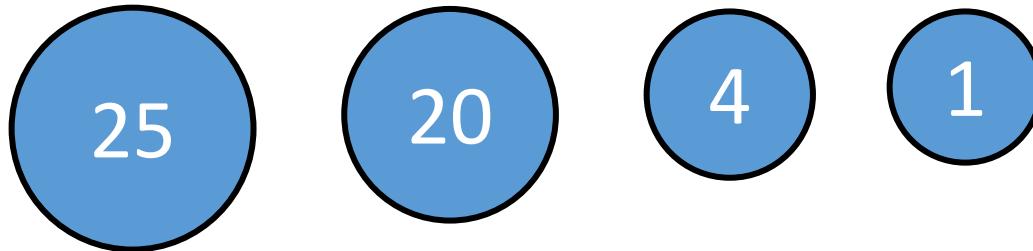
找零——算法

Algorithm MakeChange(N) sum = 0
coins = {} // 需要找零

的硬币集合，当 $sum < N$ 时 do 选择面值为 $\leq (N-sum)$ 中最大的硬币
 X sum $+= X.value$ coins $+= \{X\}$ endwhile return coins END

这个算法总是能给出 最佳结果吗？

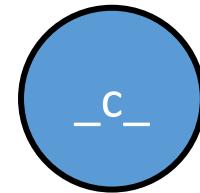
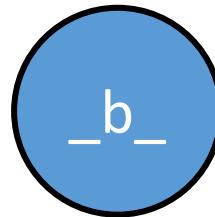
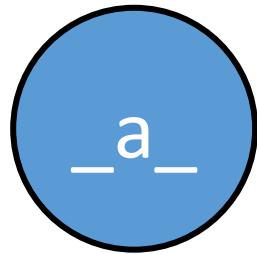
- 对于美加硬币，是的
 - 有或没有一分硬币
- 但如果硬币是：



- 而你必须付出28美分？
 - 贪心算法的结果：25、1、1、1 → 4枚硬币
 - 但有一种3枚硬币的解法

谜题

- 构造一个“更小”的反例
 - 如果你的硬币面额是：



- 而你必须凑出 __x__ 分？
- 也就是说，找到 a 、 b 、 c 、 x ，使得贪心算法给出 3 个硬币的解，尽管存在 2 个硬币的解



故事寓意

- 贪心算法并不总是能给出最优的问题的一般解法
- 但有时它们确实可以

优化问题与判定问题

- 一种**优化问题**是指你希望找到的不只是任意解，而是最佳解
 - 与**判定问题**相对——“是否存在解？”
 - 判定问题的答案是是/否
 - 优化问题涉及最小化或最大化
- 贪心算法试图解决优化问题

记住背包问题

- 优化版本：
 - 给定 N 件物品及其重量 + 和价值，还有一个承载容量为 W 的背包，盗贼能偷走的物品的最大总体价值是多少？
- 判定版本：
 - 给定 N 件物品及其重量 + 和价值，还有一个承载容量为 W 的背包，盗贼能否偷到价值为 V 的物品？

贪心算法

- 用于求解优化问题
- 通过一系列选择构建解
- 始终选择当前“最优”的可行选项
 - “最优”选择是指让我们最接近最优解的那个（例如：拿最大的可行硬币）
- 你希望通过在每一步选择一个局部最优解，最终到达一个全局最优解

贪心算法

- 贪心选择性质：
 - 可行的：必须满足问题的约束条件
 - 如果你要凑出17美分，你不会选择一枚25美分的硬币
 - 局部最优：在该步骤上于所有可行选择中做出最佳的局部选择
 - 如果你要凑出14美分，你会选择一枚10美分的，而不是5美分的硬币
 - 假设：确定这一点是“相当高效”的（想想背包问题——如何找到“最佳”选择）
 - 不可撤销：一旦制定，在算法的后续步骤中无法更改

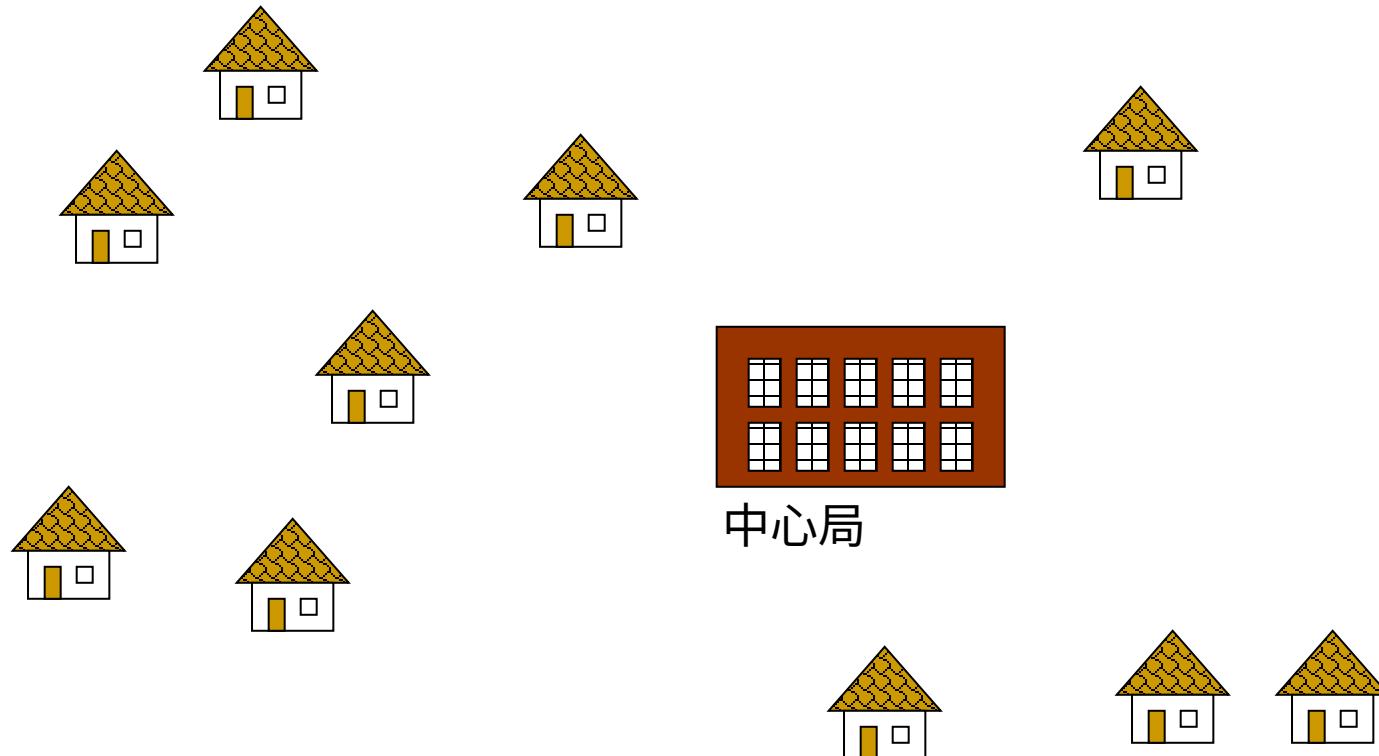
贪心算法

- 我们将研究以下问题的贪心算法：
 - 寻找图的最小生成树 (MST)
 - Prim 算法
 - Kruskal 算法
 - 在图中从单一源点寻找最短路径
 - Dijkstra 算法
 - 为图着色

贪心算法 要点概览

1. 迭代构建解
2. 在每一步选择要加入的“最佳”项
 - 选择最佳项的思路应当“简单”

一个现实世界的问题： 构建一个（物理）网络



旁注（有
用的背景）

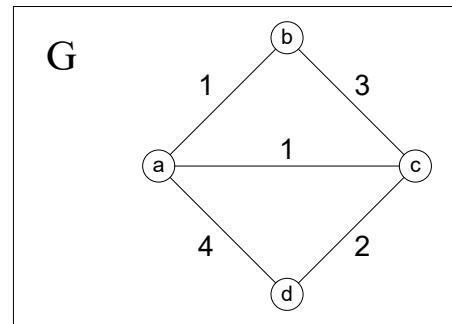
最小生成树

- 最小生成树 (**MST**) 是连通无向带权图的一个子图 **G**, 满足以下条件
 - 它包含所有顶点 (“**生成**”)
 - 它无环 (“**树**”)
 - 其边的总权重在所有可能的生成树中是**最小的**
- 最小生成树可能不是唯一的

侧栏（有用的背景信息）

最小生成树（续）

考虑图 G 的所有生成树：



每棵生成树的权重由其边的权重之和给出……

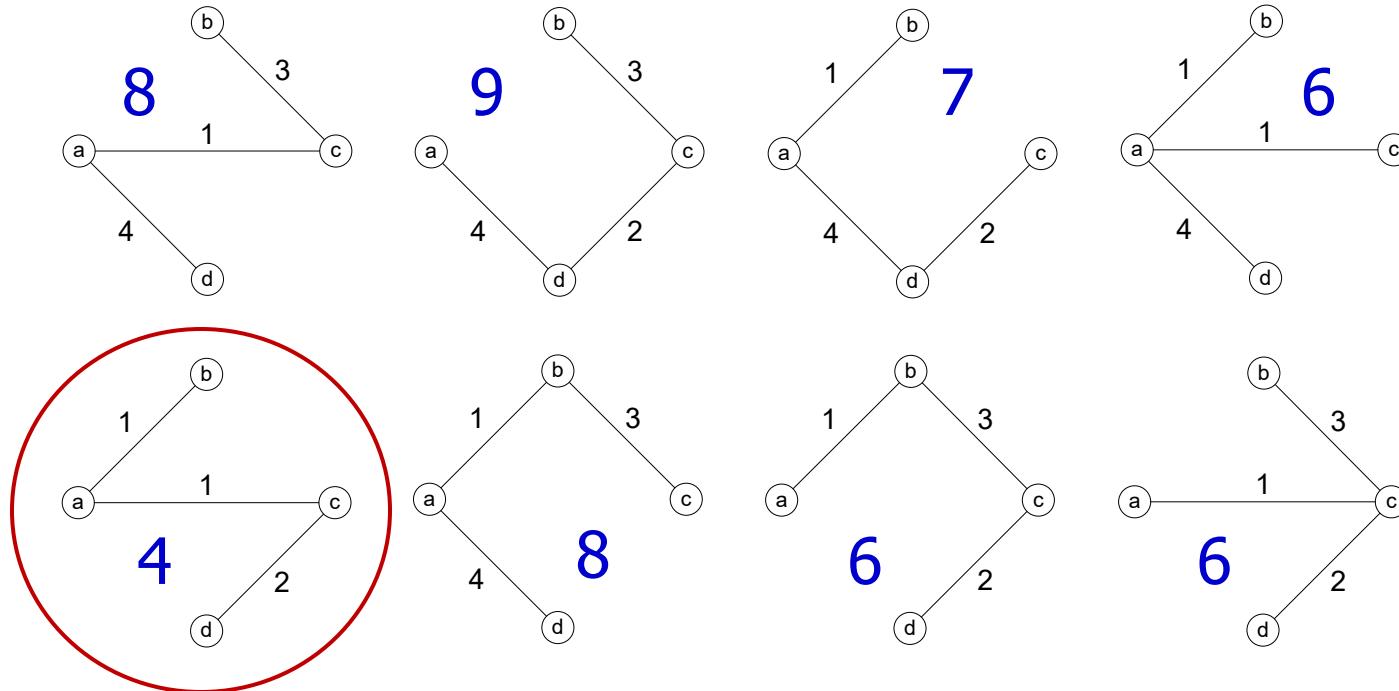


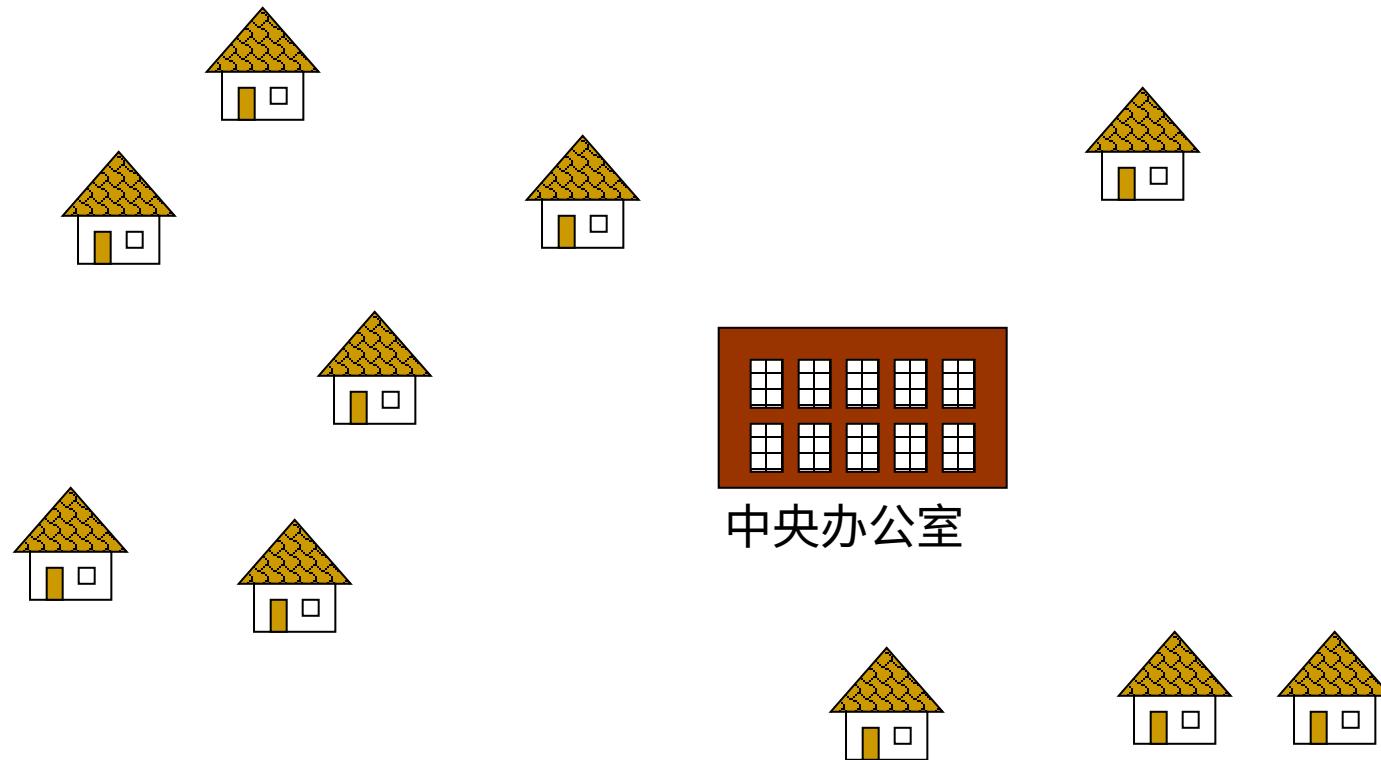
图 G 的最小生成树如下
graph, and it has a weight of 4.
we

If y 构造最小生成树

在完全图上：

- 结果：
 - 是一个 树 (显然)
 - 因此 连通
 - 连接了 所有节点
 - 使用 最低成本

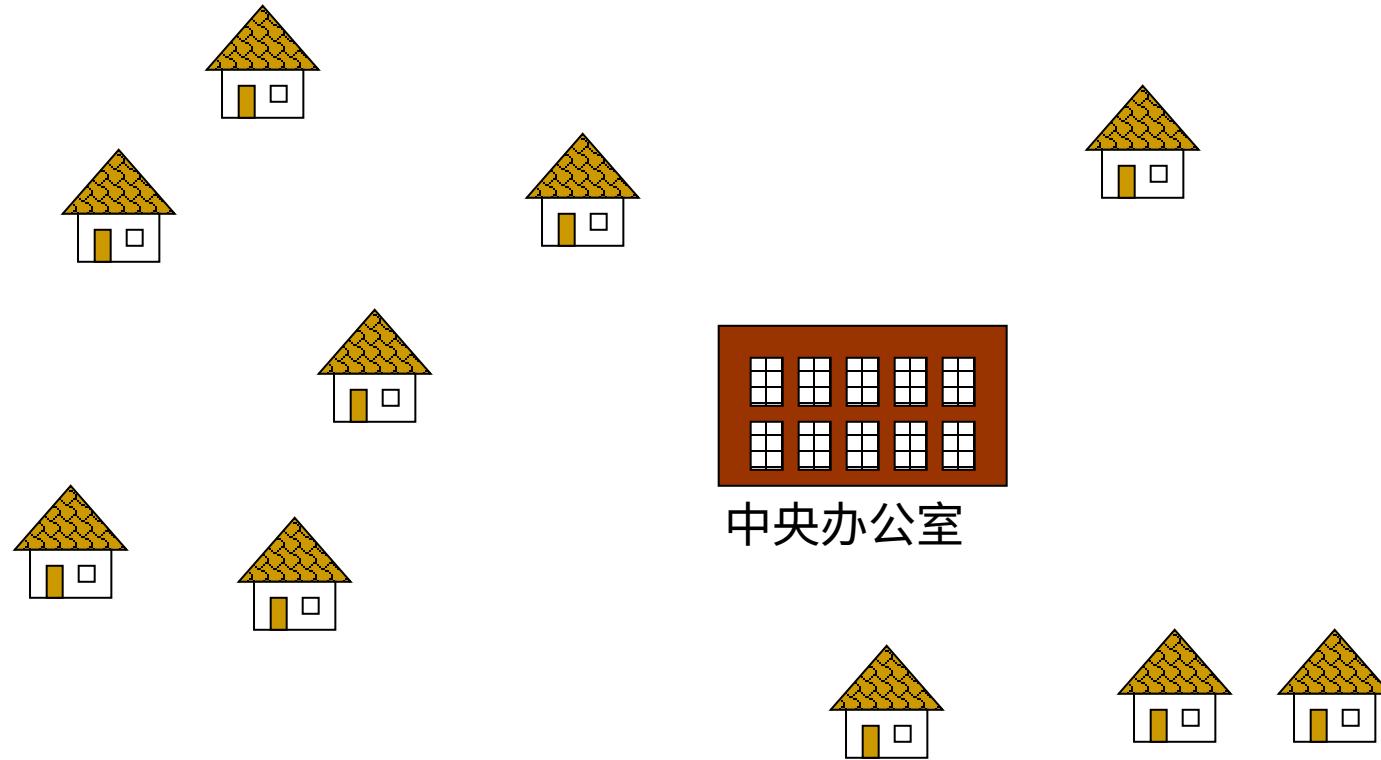
回到我们的小村庄

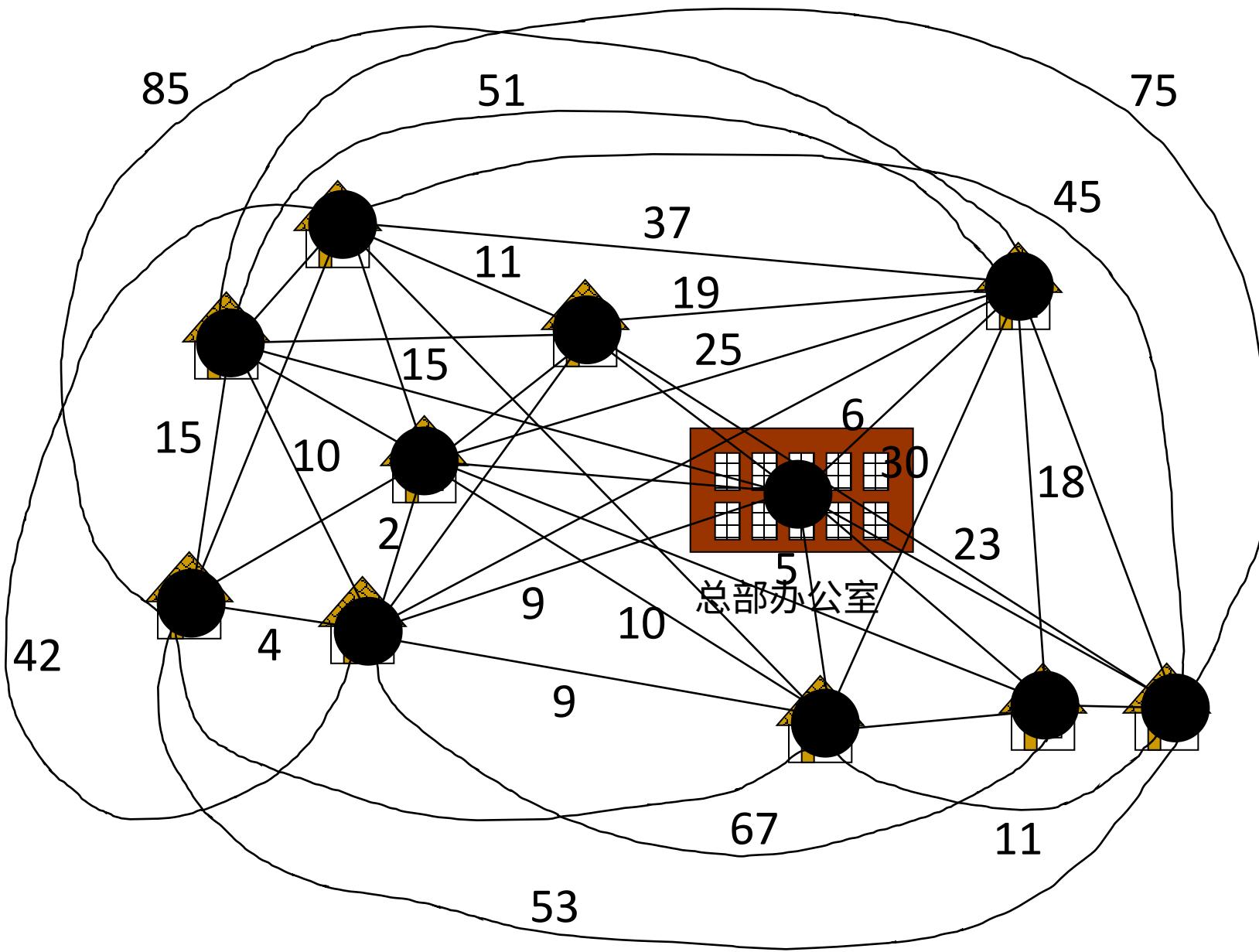


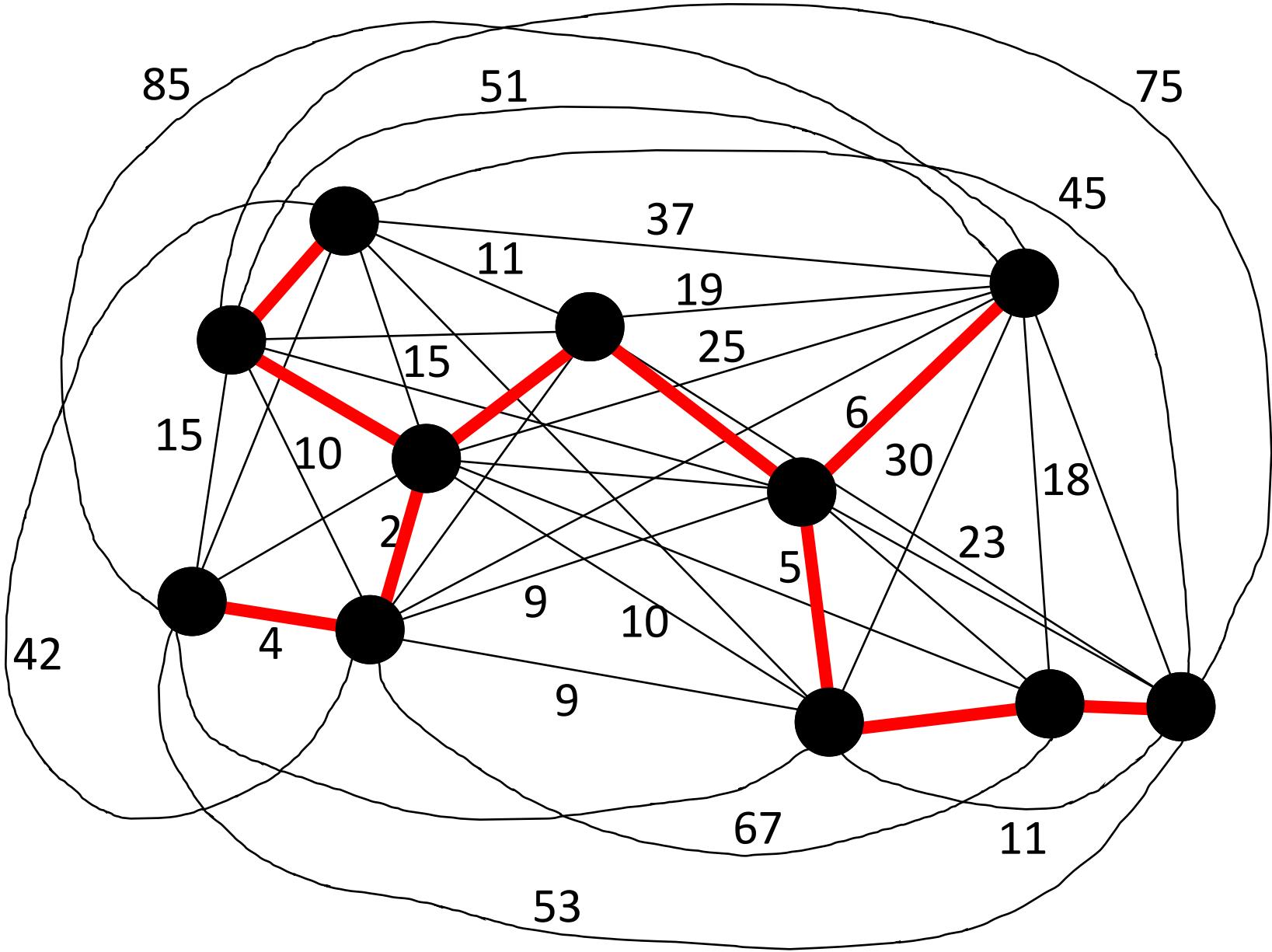
让我们用最小生成树来解决这个问题

将其表示为图

- 顶点是所有需要连接的节点
- 每一种可能的连接都对应一条边
 - 即 N 个顶点的完全图
- 每条边都有一个与之相关的“权重”
 - 从节点 A 到节点 B 铺设电线的成本
- 现在找到最小生成树
 - 这如何解决问题?
 - 生成树 → 所有节点都已连接
 - 最低成本树 → 最便宜的网络可能性





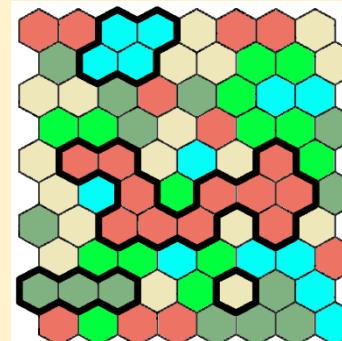


提醒：用图解决问题，

策略二

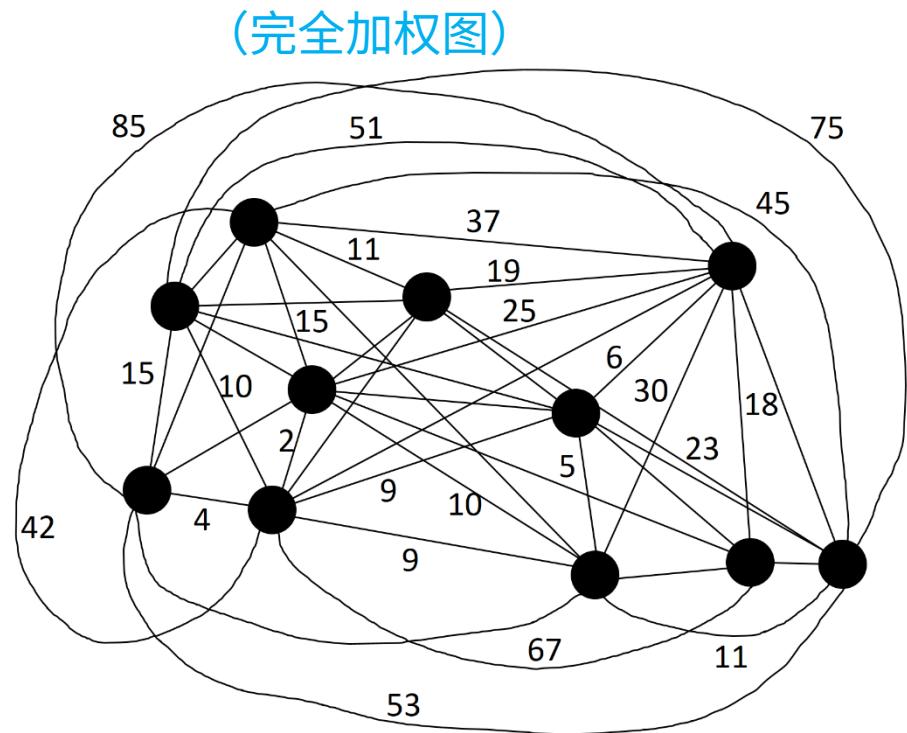
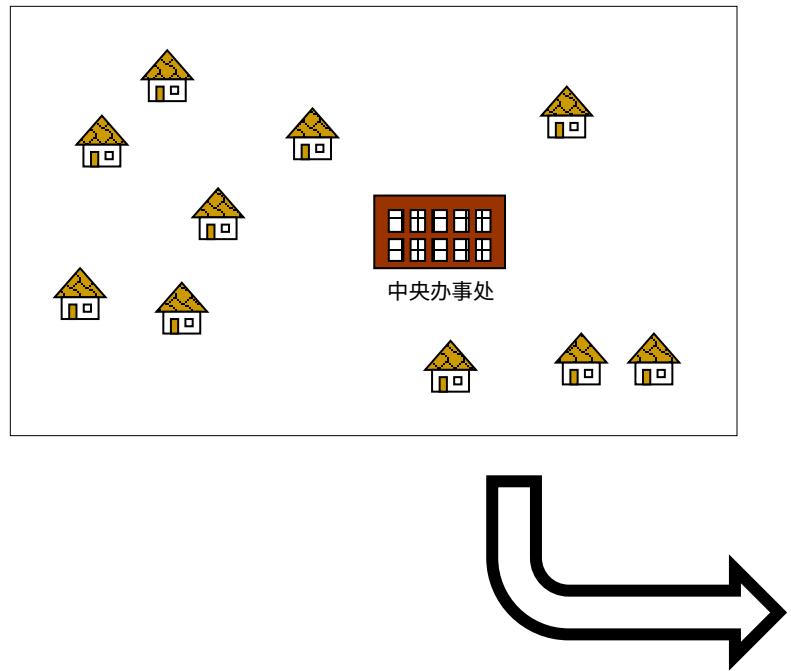
1. 将问题“巧妙地”表示为图
2. 将该图输入图算法
3. 使用算法输出确定问题的答案

我们也在“计数地图区域”
问题上使用了策略二（不
同的图算法）



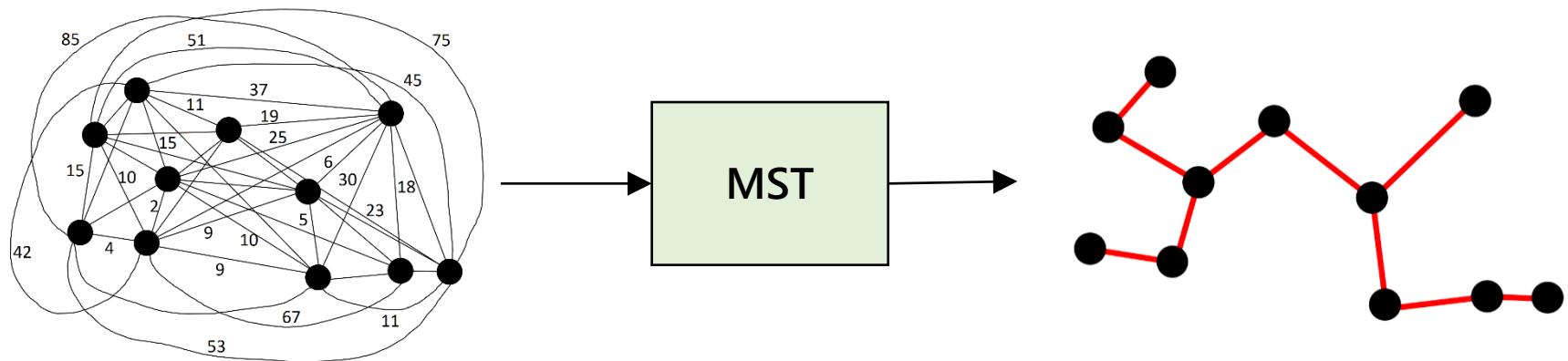
示例：我们的小村庄

1. 将问题“巧妙地”表示为图



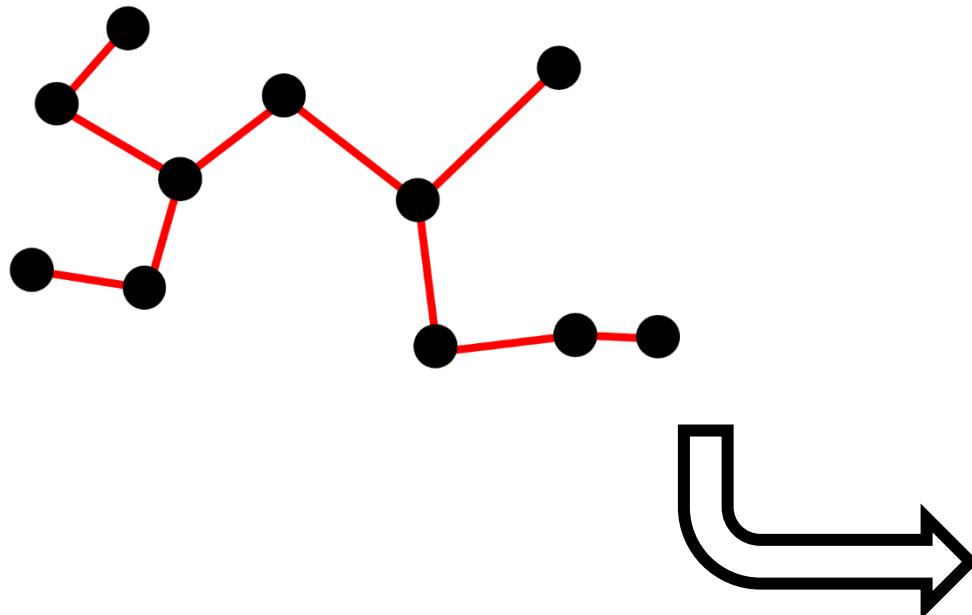
示例：我们的小村庄

2. 将图输入图算法



示例：我们的小村庄

3. 使用输出结果来确定你问题的答案



解决方案：
将房屋 A 连接到 B
将房屋 B 连接到 C
将房屋 C 连接到 D
将房屋 D 连接到 Central

... 等
等。

呼。

- 现在我们还需要下面这个：

MST

- 事实上，我们将要看其中的两个：

Prim

克鲁斯克
尔

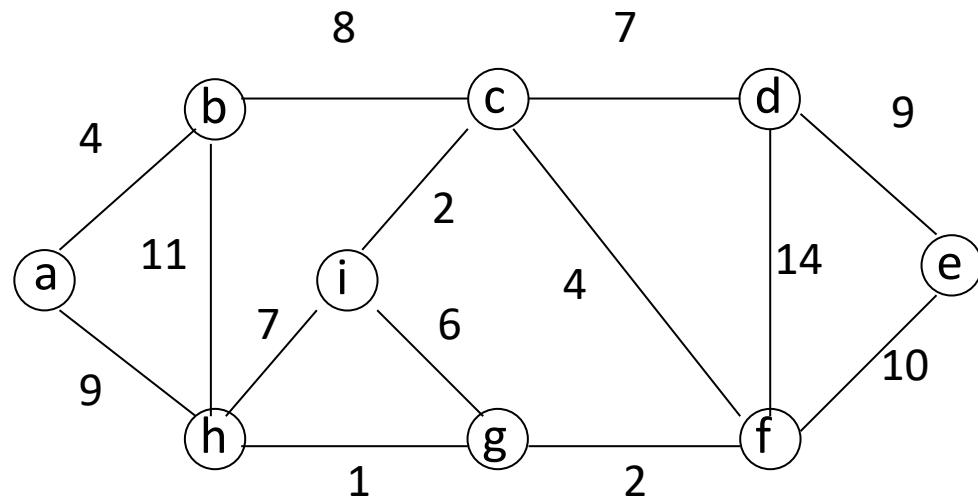
贪心算法：Prim 算 法

教材：第9.1章

普里姆算法

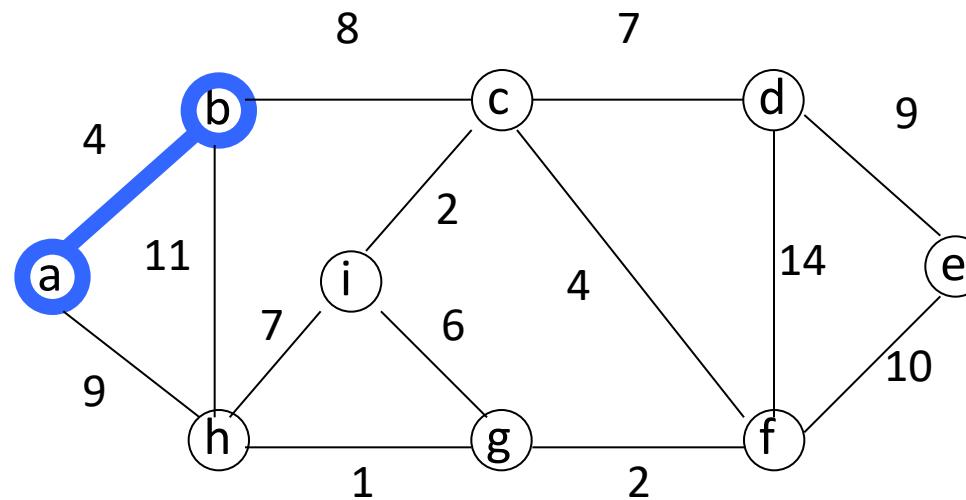
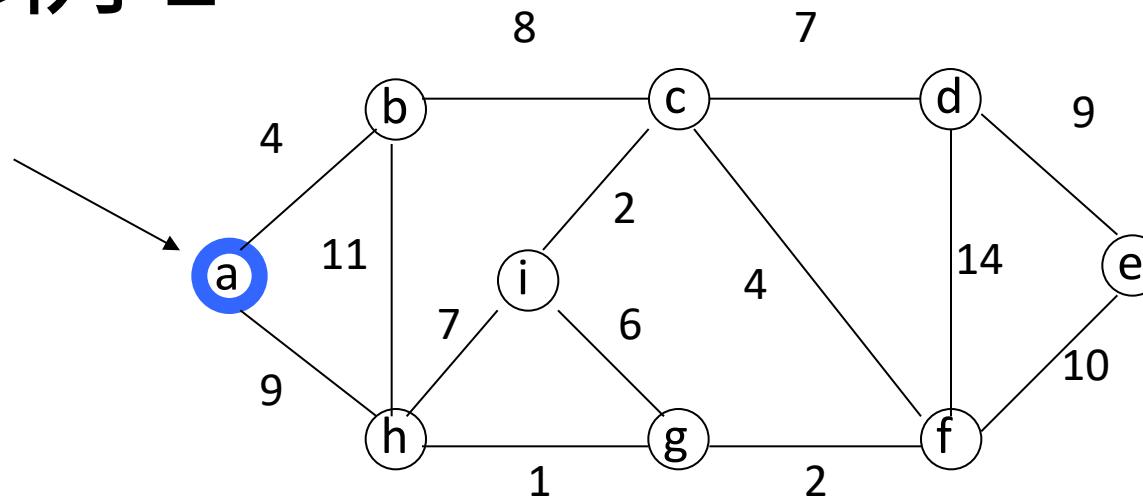
- 贪心算法要点：
 1. 迭代地构建解
 2. 每一步做出贪心选择并将其加入解中
- 普里姆算法：
 - 从任意一个顶点开始
 - 每次迭代的贪心选择：
 - 具有一个顶点已在集合内、另一个顶点在集合外的最低代价边

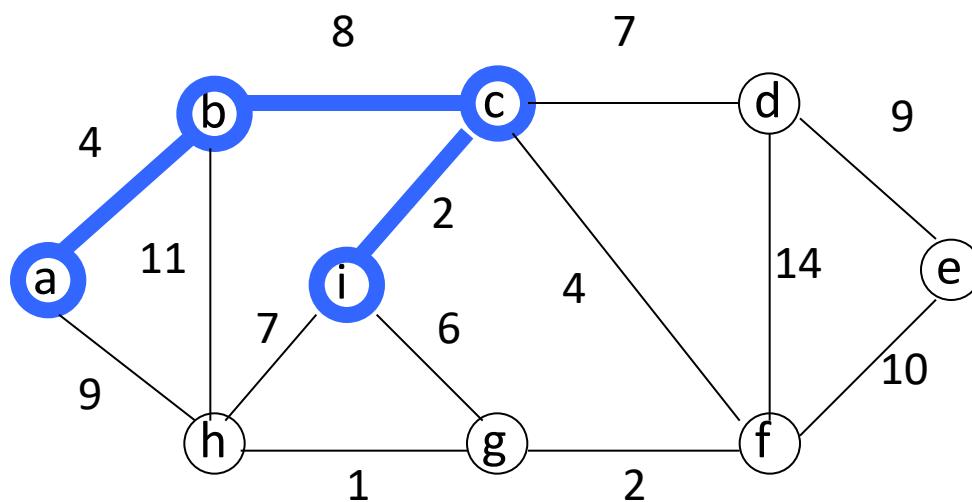
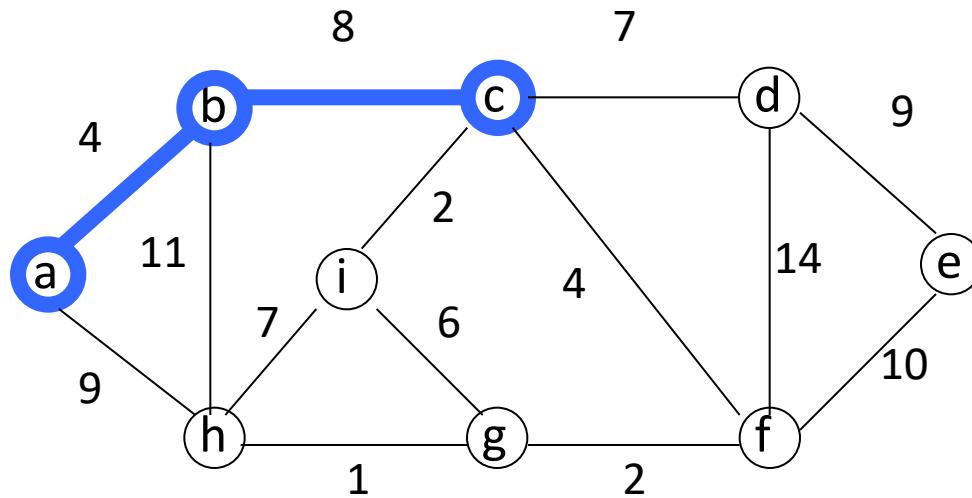
示例 1

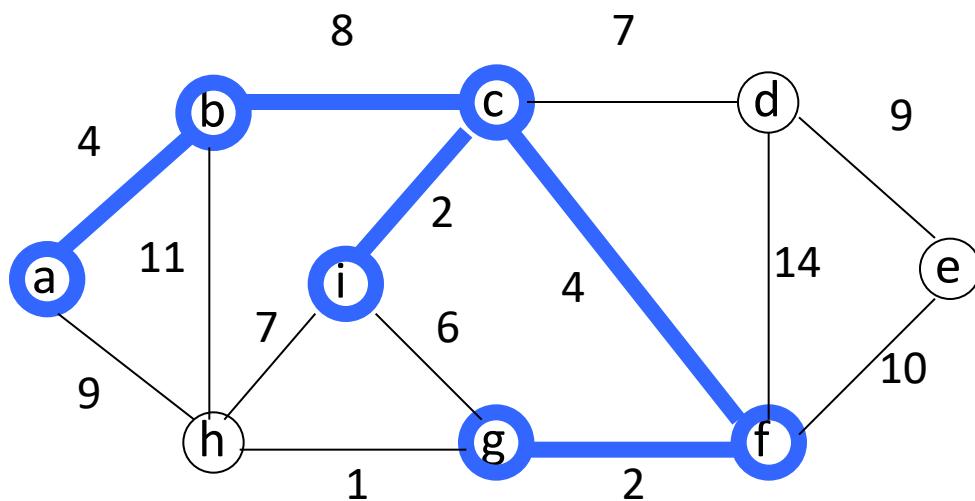
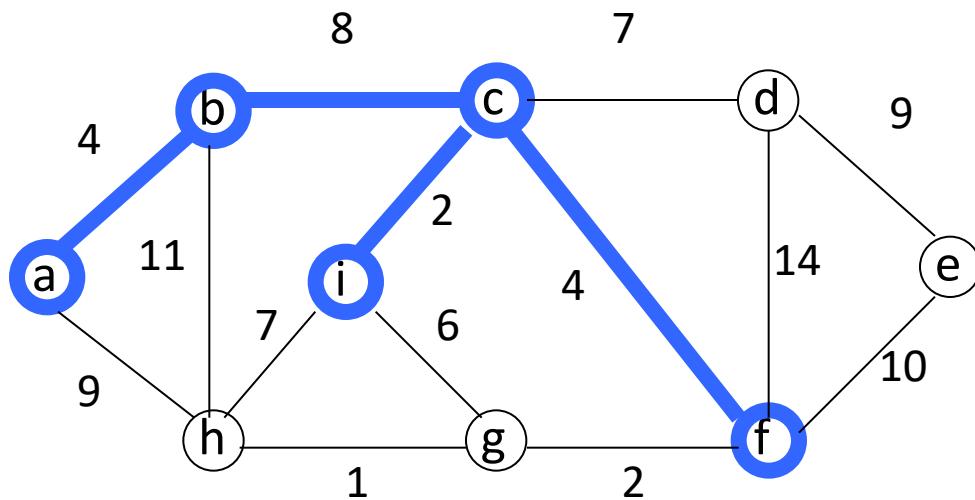


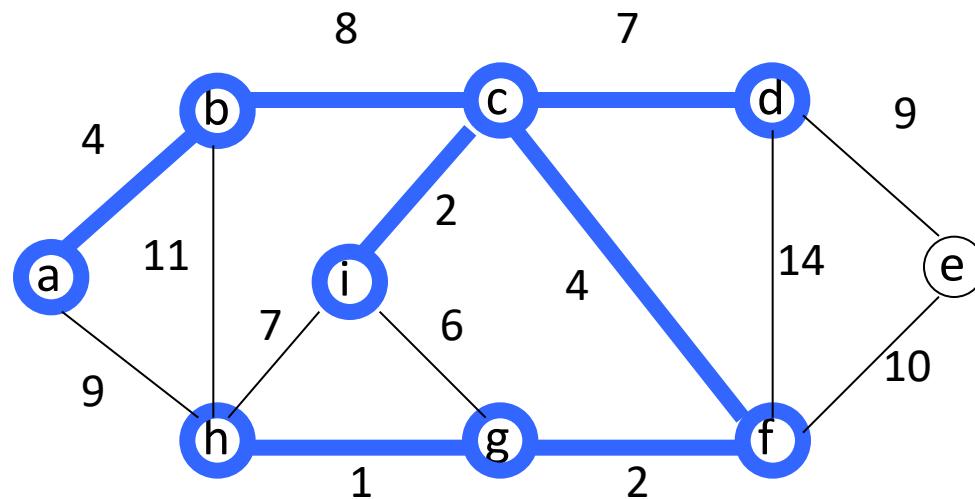
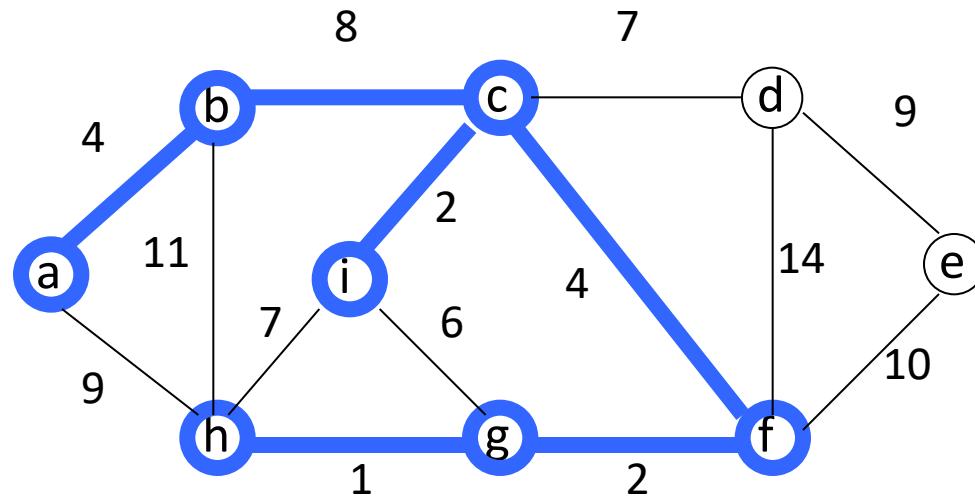
示例 1

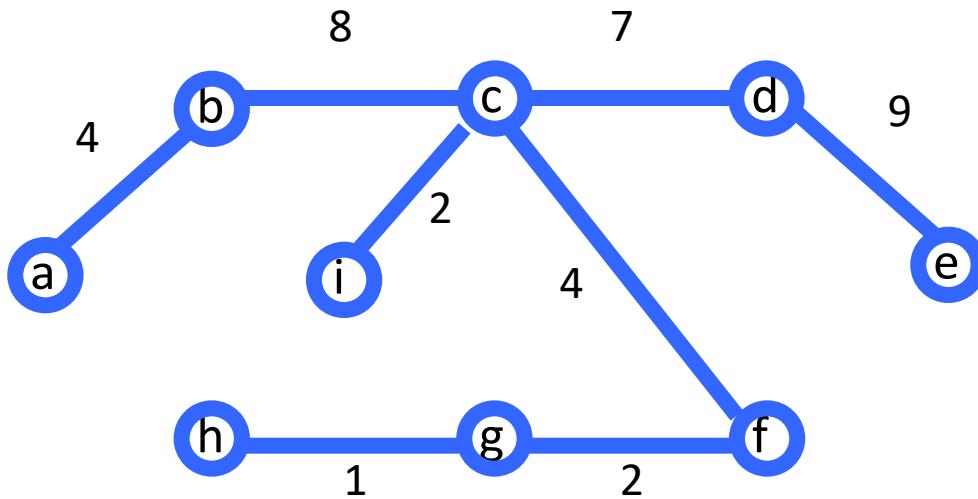
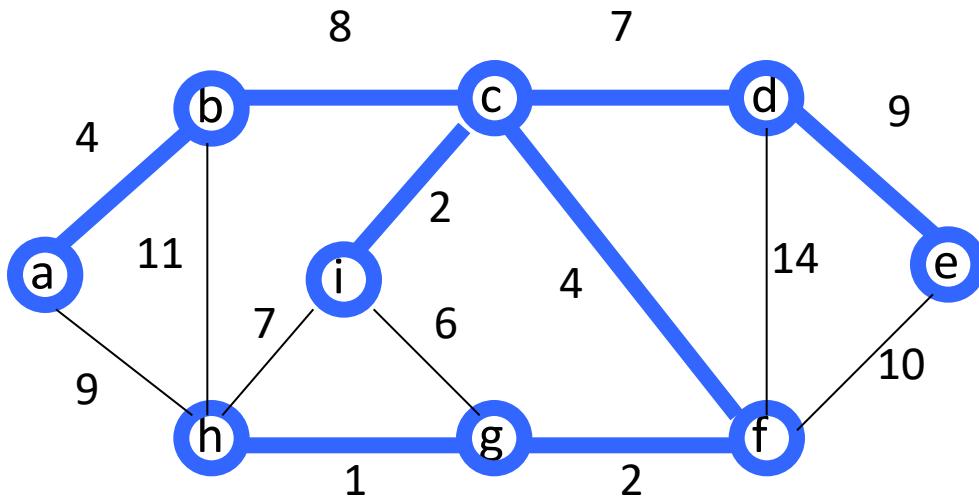
根
顶点









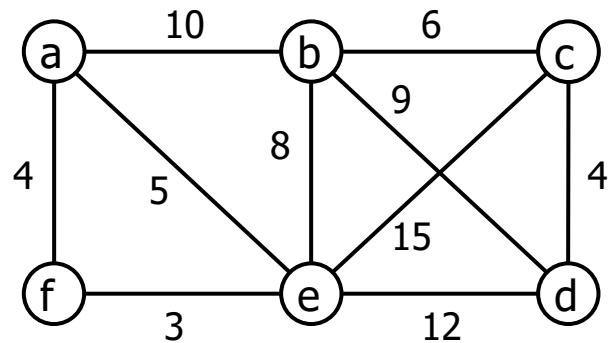


Prim 算法

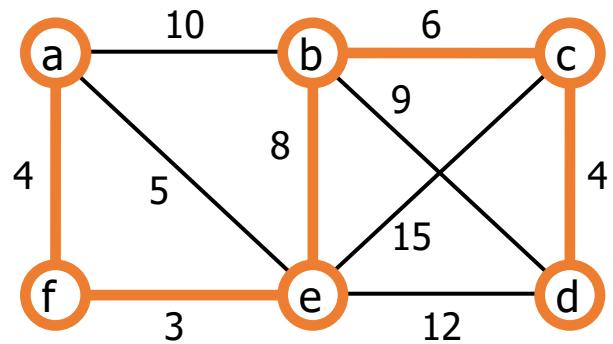
Algorithm Prim(G)

```
vT ← {v0}                                // init tree with one (arbitrary) vertex
ET ← ∅                                     // 用没有边的树初始化
for i ← 1 to N-1 do                          // 循环直到所有顶点都被加入到树中
    从 E 中找到一条最小权重边 e=(u,v)
    where u is in VT (in the tree)
    并且 v 在 V-VT 中 (尚未在树中)
    vT ← vT ∪ {v}                            // 将顶点 v 加入到树中
    ET ← ET ∪ {e}                           // 将边 (u,v) 加入到树中
return T = (VT, ET)
```

示例 2



示例 2



贪婪算法： Kruskal 算法

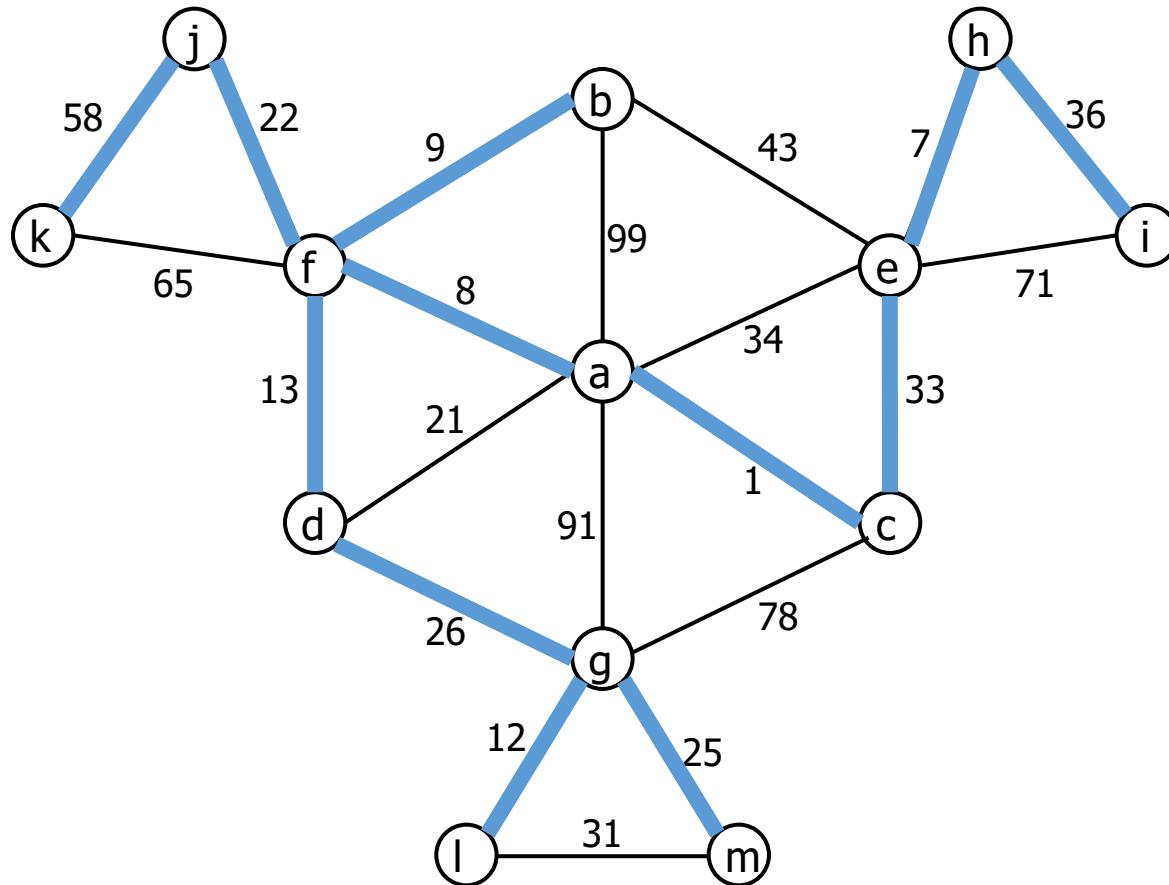
教材：第9.2章

背景

- 另一个我们正在讨论的“贪心算法”之一
检查：
 - 图的最小生成树
 - 普里姆算法
 - 克鲁斯卡尔算法
 - 图中单源最短路径
 - 迪杰斯特拉算法
 - 图着色

Kruskal (概览)

- 重复添加不会引入环路的最小权重边
- 示例：



Kruskal 算法 (基本思路)

Kruskal(G)

按权重将 E 的边按升序排序

$V_T \leftarrow V // T$ 包含 G 的所有顶点 $E_T \leftarrow \emptyset$ // 从
T 中没有边开始 count $\leftarrow 0$

$k \leftarrow 0$ // 在 G 的边上遍历索引 while count < |V|-1
do // 当 T 有这么多条边时完成 $k \leftarrow k + 1$

if $E_T \cup \{e_k\}$ 是无环的 // 将此边添加到 T 是否安全? $E_T \leftarrow E_T \cup \{e_k\} //$
…则添加它 count \leftarrow count + 1

return T = (V_T, E_T)

这三点是“效率挑战”

Kruskal 算法：实现挑战

1. 对边进行排序

- 我们知道几种 $O(N \log N)$ 的方法
- 哪个方法最适合我们？

2. 判断添加一条边是否会形成环

- 也许可以用 DFS 或 BFS 来检测环？
 - 这些是 $O(N^2)$ ，我们必须进行 $O(N)$ 次
 - 我们能将 $O(N^3)$ 改进吗？
- 答案是肯定的，借助巧妙的数据结构可以做到

Disj不相交子集 (又名“并查集”)

- 一组互不相交的子集——任何元素在任一时刻只能属于一个子集
- 数据结构上的操作：
 - **Makeset(x)** – 创建一个包含元素 x 的新子集
 - **Find(x)** – 返回包含 x 的子集
 - **Union(x,y)** – 将包含 x 和 y 的子集合并

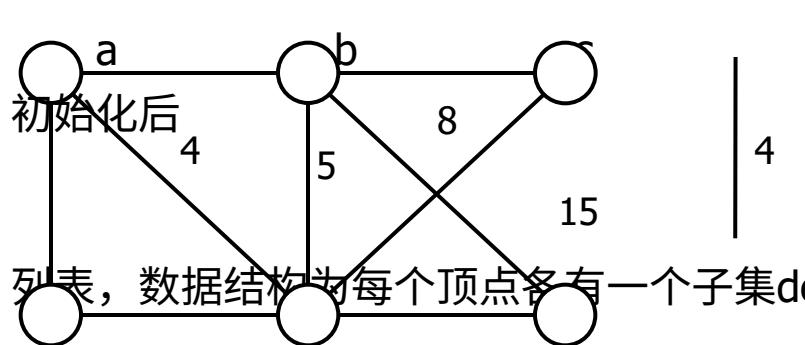
并查集示例

```
for x in [1..8] do
    makeset(x)
        → DS is now {1} {2} {3} {4} {5} {6} {7} {8}
union(2,7)
        → DS is now {1} {2,7} {3} {4} {5} {6} {8}
union(1,4)
        → DS is now {1,4} {2,7} {3} {5} {6} {8}
y ← find(4)
        → y is now {1,4}
union(y,3)
        → DS is now {1,4,3} {2,7} {5} {6} {8}
x ← find(1)
        → x is now {1,4,3}
y ← find(7)
        → y is now {2,7}
union(x,y)
        → DS is now {1,4,3,2,7} {5} {6} {8}
```

Kruskal 算法与并查集

- 维护生成树 T 中顶点的并查集数据结构
- 初始时每个顶点都是一个独立的子集
- 当一条边 (u,v) 被加入到 T 时：
 $\text{DS.union}(u,v)$
- 每个子集对应一个连通分量
 - 它也是一棵树——最终最小生成树（MST）的一个子集
- 如果 u,v 在同一个子集中 不要添加边
 - 那会形成一个环
- 最后 DS 中只会有一个子集
 - T 是一个连通分量

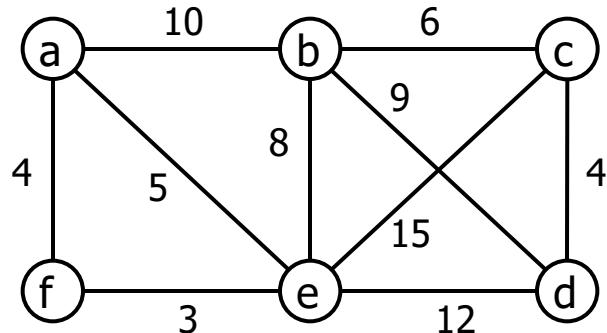
另一个 Kruskal 示例（使用不相交子集）



- 优先队列包含已排序的边
-

PQ	子集	解
<u>键：值</u>	{a} {b} {c} {d} {e} {f}	
3 : ef		(a)
4 : af		(b)
4 : cd		(c)
5 : ae		
6 : bc		(f)
8 : be		(e)
9 : bd		(d)
10:ab		
12:de		
15:ce		

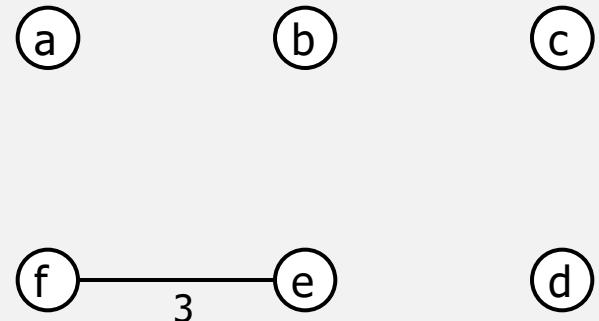
另一个 Kruskal 示例（使用不相交子集）



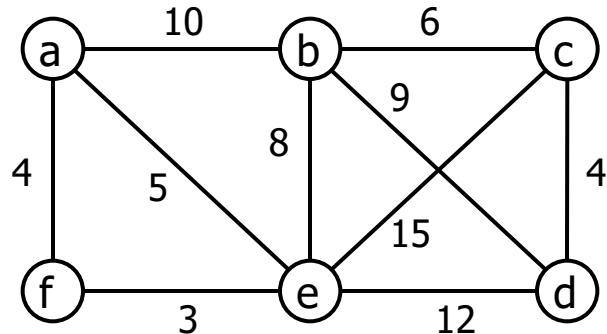
- 迭代 1 之后
- 边 ef 已添加
- e、f 子集已合并

PQ	子集
<u>key: value</u>	{a} {b} {c} {d} {e} {f}
3 : ef	{a} {b} {c} {d} {e, f}
4 : af	
4 : cd	
5 : ae	
6 : bc	
8 : be	
9 : bd	
10:ab	
12:de	
15:ce	

解答



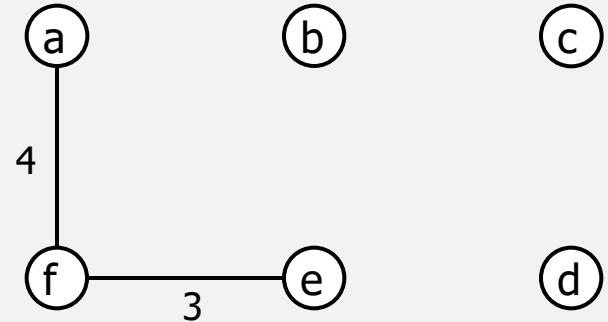
另一个 Kruskal 示例（使用不相交子集）



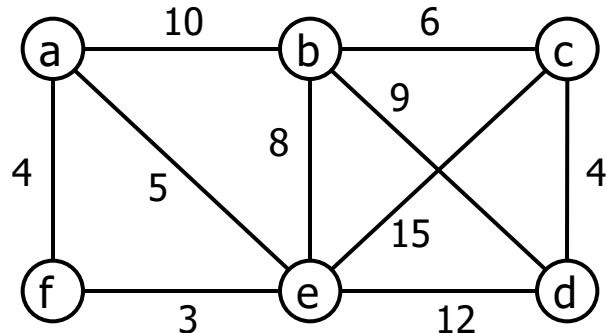
- 迭代 2 之后
- 边 af 已被加入
- 子集 a、f 合并

PQ	子集
<u>key: value</u>	{a} {b} {c} {d} {e} {f}
3:ef	{a} {b} {c} {d} {e, f}
4:af	{a, e, f} {b} {c} {d}
4:cd	
5:ae	
6:bc	
8:be	
9:bd	
10:ab	
12:de	
15:ce	

解答



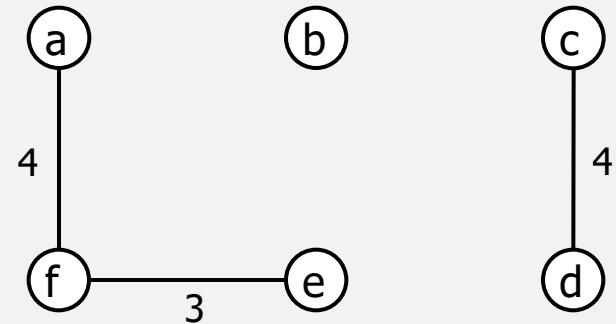
另一个 Kruskal 示例（使用不相交子集）



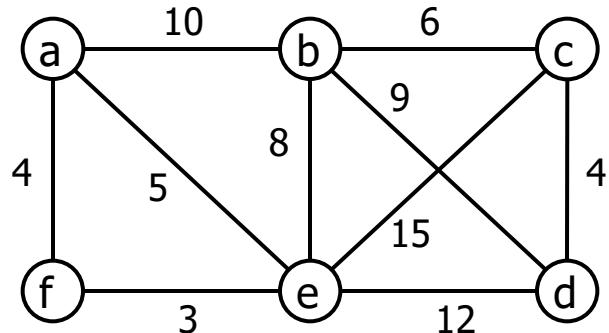
- 第 3 次迭代后
- 边 cd 已被加入
- c、d 子集已合并

PQ	子集
<u>key: value</u>	{a} {b} {c} {d} {e} {f}
3:ef	{a} {b} {c} {d} {e, f}
4:af	{a, e, f} {b} {c} {d}
4:cd	{a, e, f} {b} {c, d}
5:ae	
6:bc	
8:be	
9:bd	
10:ab	
12:de	
15:ce	

解答



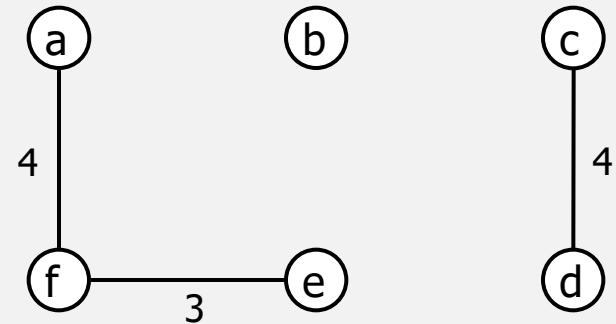
另一个 Kruskal 示例（使用不相交子集）



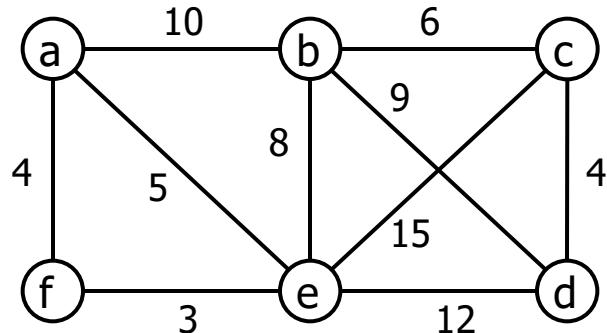
- 第 4 次迭代没有变化
- a 和 e 属于同一子集
- 边 ae 未被加入，因为它会导致环路

PQ	子集
<u>key: value</u>	{a} {b} {c} {d} {e} {f}
3:ef	{a} {b} {c} {d} {e, f}
4:af	{a,e,f} {b} {c} {d}
4:cd	{a,e,f} {b} {c, d}
5:ae	{a, e, f} {b} {c, d}
6:bc	
8:be	
9:bd	
10:ab	
12:de	
15:ce	

解答



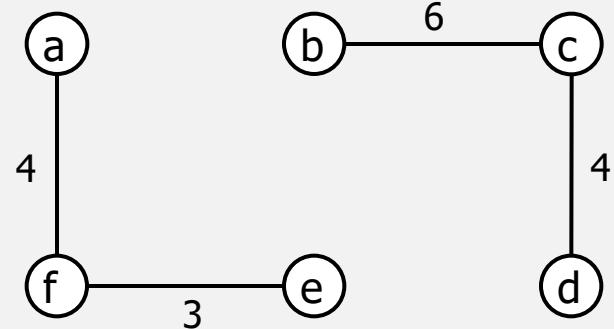
另一个 Kruskal 示例（使用不相交子集）



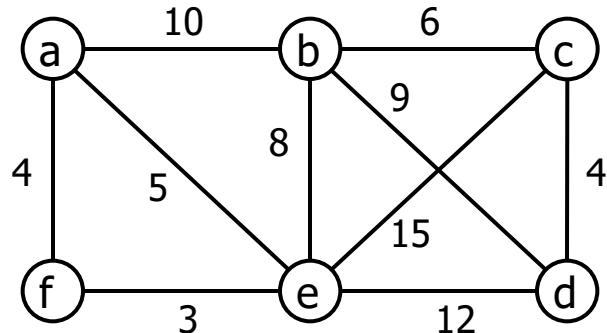
- 迭代 5 之后
- 边 bc 已被加入
- b、c 子集已合并

PQ	子集
<u>key: value</u>	{a} {b} {c} {d} {e} {f}
3:ef	{a} {b} {c} {d} {e, f}
4:af	{a, e, f} {b} {c} {d}
4:cd	{a, e, f} {b} {c, d}
5:ae	{a, e, f} {b} {c, d}
6:bc	{a, e, f} {b, c, d}
8:be	
9:bd	
10:ab	
12:de	
15:ce	

解答



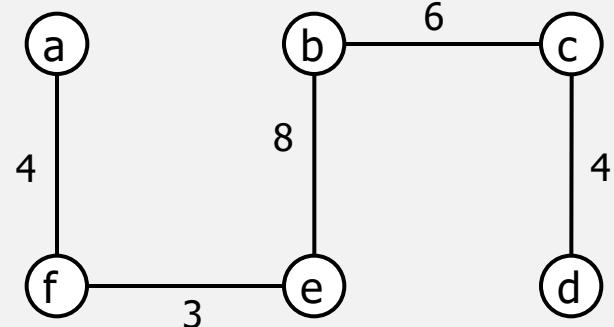
另一个 Kruskal 示例（使用不相交子集）



- 在第 6 次迭代之后
- 边 be 已被添加
- 已添加 N-1 条边，主循环结束
- 算法返回解

PQ	键：值	子集
		{a} {b} {c} {d} {e} {f}
3:ef		{a} {b} {c} {d} {e, f}
4:af		{a, e, f} {b} {c} {d}
4:cd		{a, e, f} {b} {c, d}
5:ae		{a, e, f} {b} {c, d}
6:bc		{a, e, f} {b, c, d}
8:be		{a, e, f, b, c, d}
9:bd		
10:ab		
12:de		
15:ce		

解答



Kruskal 算法与优先队列 + 不相交子集

A1 将 G_m 中的所有顶点添加到 T // 添加顶点但不添加边
创建优先队列 PQ // 用于保存候选边
创建集合 DS // 每个顶点的不相交子集 对 G 中的每个顶点 v 执行

对 G 中的每条边 e 执行
 $DS.makeset(v)$

$PQ.add(e.weight, e)$ // 按最小权重的边的优先队列

当 T 的边数少于 $n-1$ 时执行

$(u,v) \leftarrow PQ.removeMin()$ // 获取下一条最小的边

$cu \leftarrow DS.find(u)$

$cv \leftarrow DS.find(v)$

$if cu \neq cv$ then // 确保 u,v 不在同一子集中

$T.addEdge(u,v)$ // 加入边
 $DS.union(cu, cv)$

return T

Kruskal 算法的效率

- 使用高效的并查集算法，最耗时的步骤是对边权重的初始排序
 - $O(|E| \log(|E|))$
 - 记住在最坏情况下， $|E|$ 是 $|V|^2$
 - 因此这也是 $O(|V|^2 \log(|V|))$
 - 由于我们通常用 N 表示图中的顶点数，所以这是 $O(N^2 \log N)$

Prim 与 Kruskal 的要点概览

- 相同的问题：最小生成树（MST）
- 两者都是贪心算法
- 两者都一次加入一条边
 - Prim 的贪心选择：扩展树的最小边
 - 正在构建的图（树）始终保持连通，每次+添加一个顶点/一条边
 - Kruskal 的算法：不会形成环的最小边
 - 正在构建的图是一个森林，所有顶点已存在，我们只是在添加边

贪心算法： Dijkstra 算法

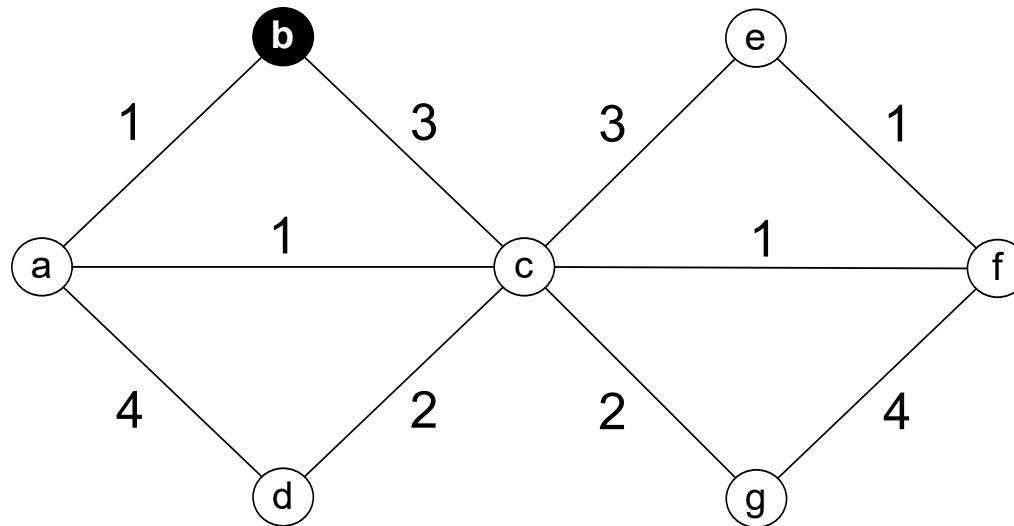
教材：第9.3章

背景

- 我们将要研究的另一个“贪心算法”：
 - 图的最小生成树
 - Prim 算法
 - Kruskal 算法
 - 图中单源最短路径
 - Dijkstra 算法
 - 图着色

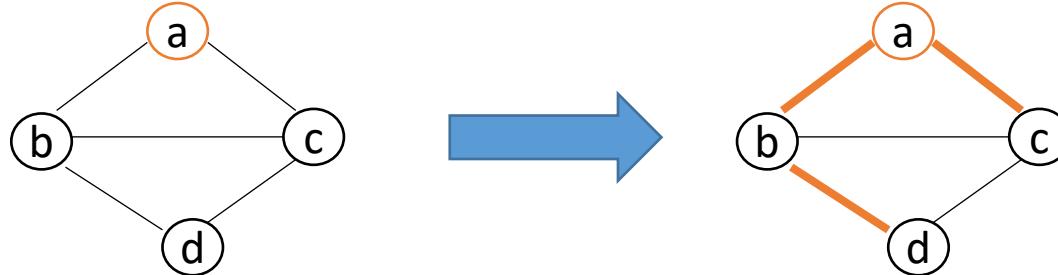
问题： 单源最短路径

- 从选定的顶点（源点）到每个其他顶点寻找最短路径

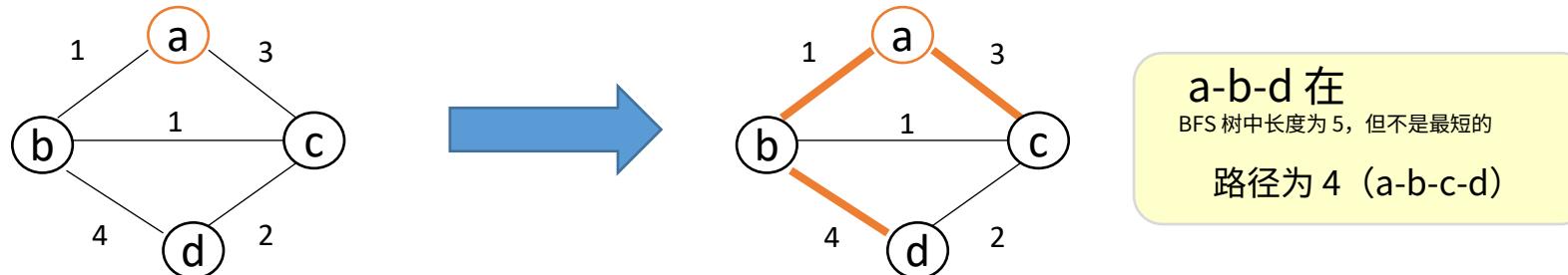


BFS 怎么样？

- 对于无权图，简单/基础的 BFS 已经可以做到这一点：



- ……但对于有权图则不行。考虑 a 到 d 的距离：



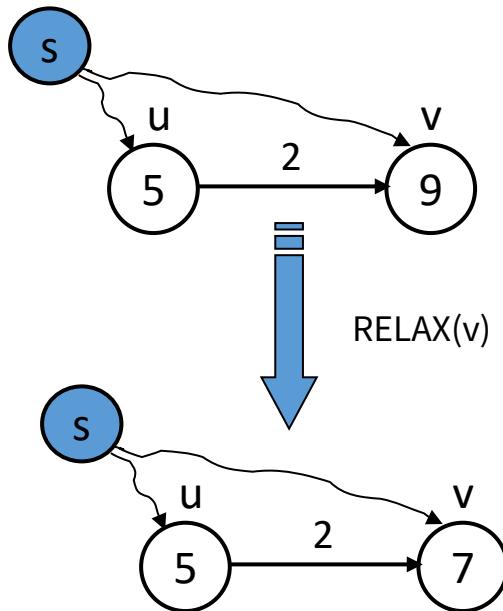
- 在有权图中查找最短路径的算法需要在将边包含进解之前考虑该边的权重

戴克斯特拉算法的思想

- 记住到所有顶点的已知最短距离
 - 初始时对所有顶点为“无穷”
- 选择距离最近且未处理的顶点
 - “最近”的定义待定
- 查看它的所有邻居
- 更新它们已知的最短距离（“松弛”）
- 重复

松弛

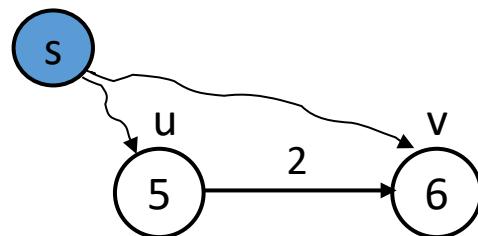
- Dijkstra 指 “松弛” 一个顶点
- 含义：更新到 v 的已知最短路径



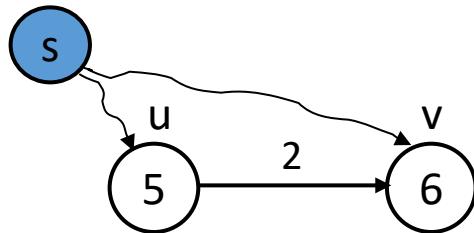
我们处于一个中间阶段：到目前为止我们“知道”从 s 到 u 的代价为 5，从 s 到 v 的代价为 9

利用关于边 (u,v) 的新信息，我们现在知道有一条到 v 的更便宜的路径

放松 —— 另一个例子



↓ 放松(v)

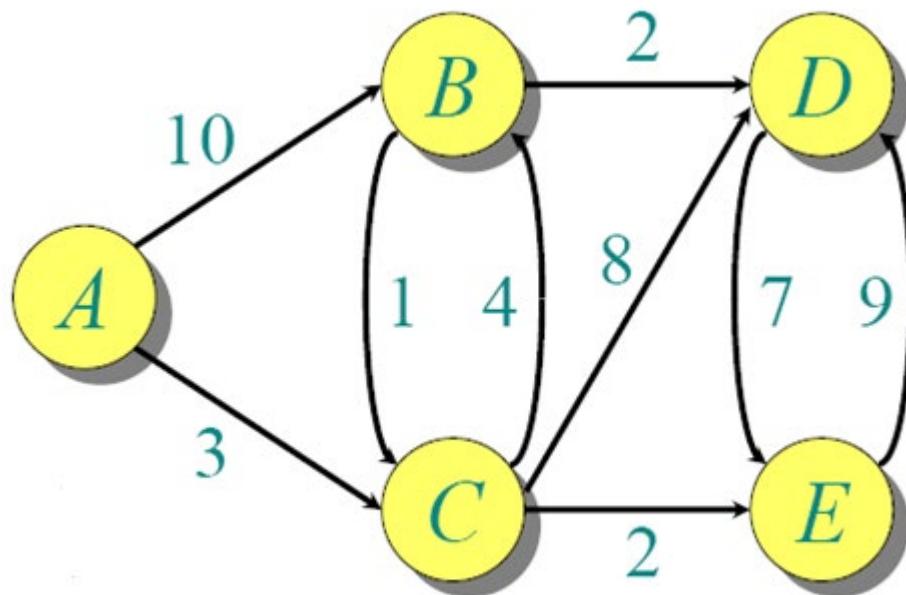


5+2 不比 6 更好

没有改进，因此这次
不做任何更改

迪杰斯特拉示例

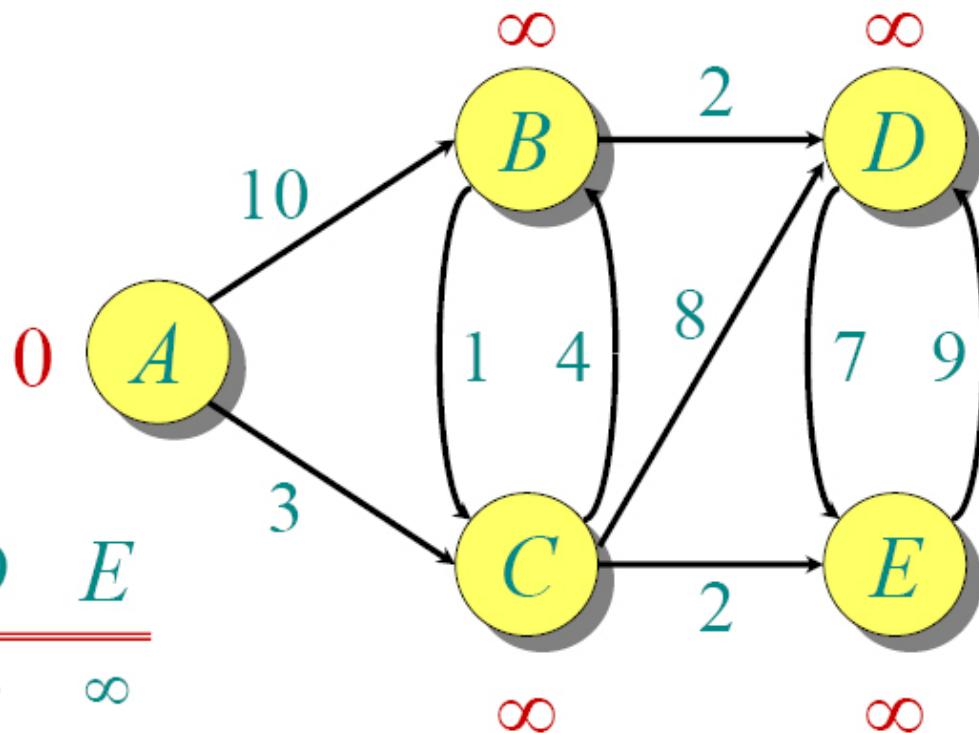
找到从 A 到所有其他顶点的最短路径



Dijkstra 示例

Initialize:

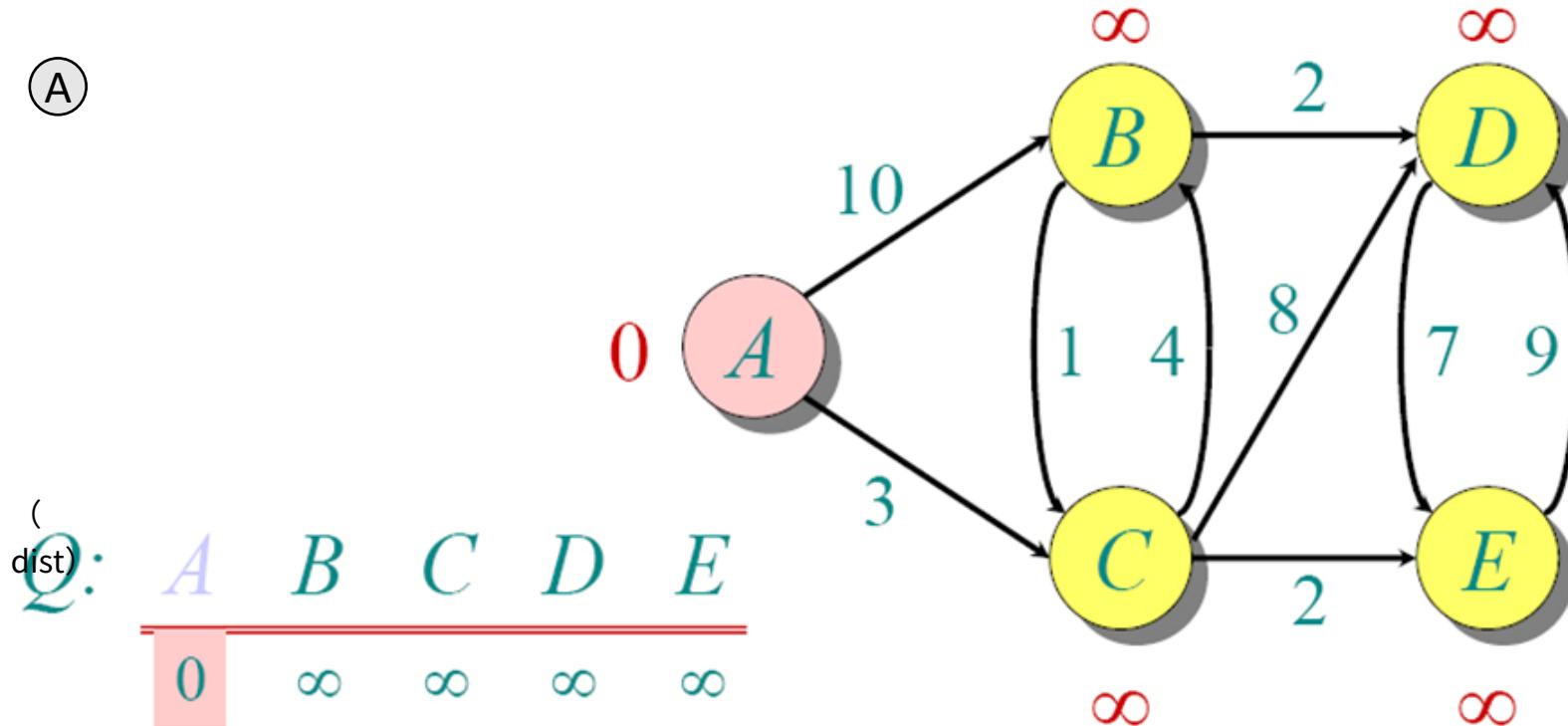
Q : (dist)	A	B	C	D	E
	0	∞	∞	∞	∞



$S: \{\}$

Dijkstra 算法

添加顶点 A

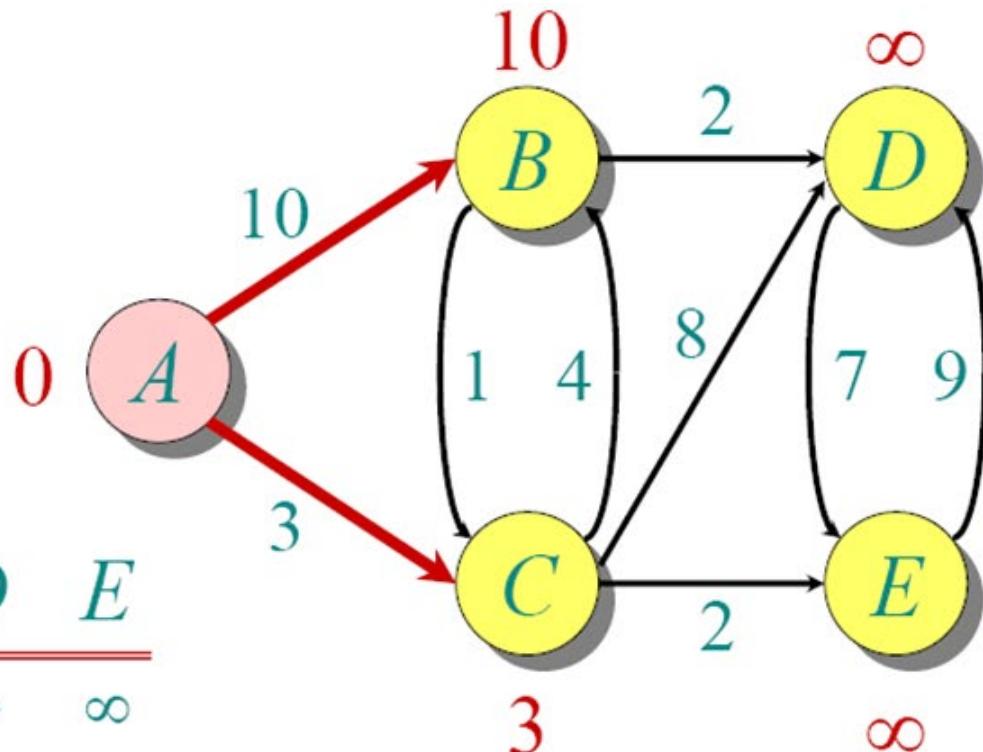


Dijkstra 算法示例

放松 A 的邻居

(A)

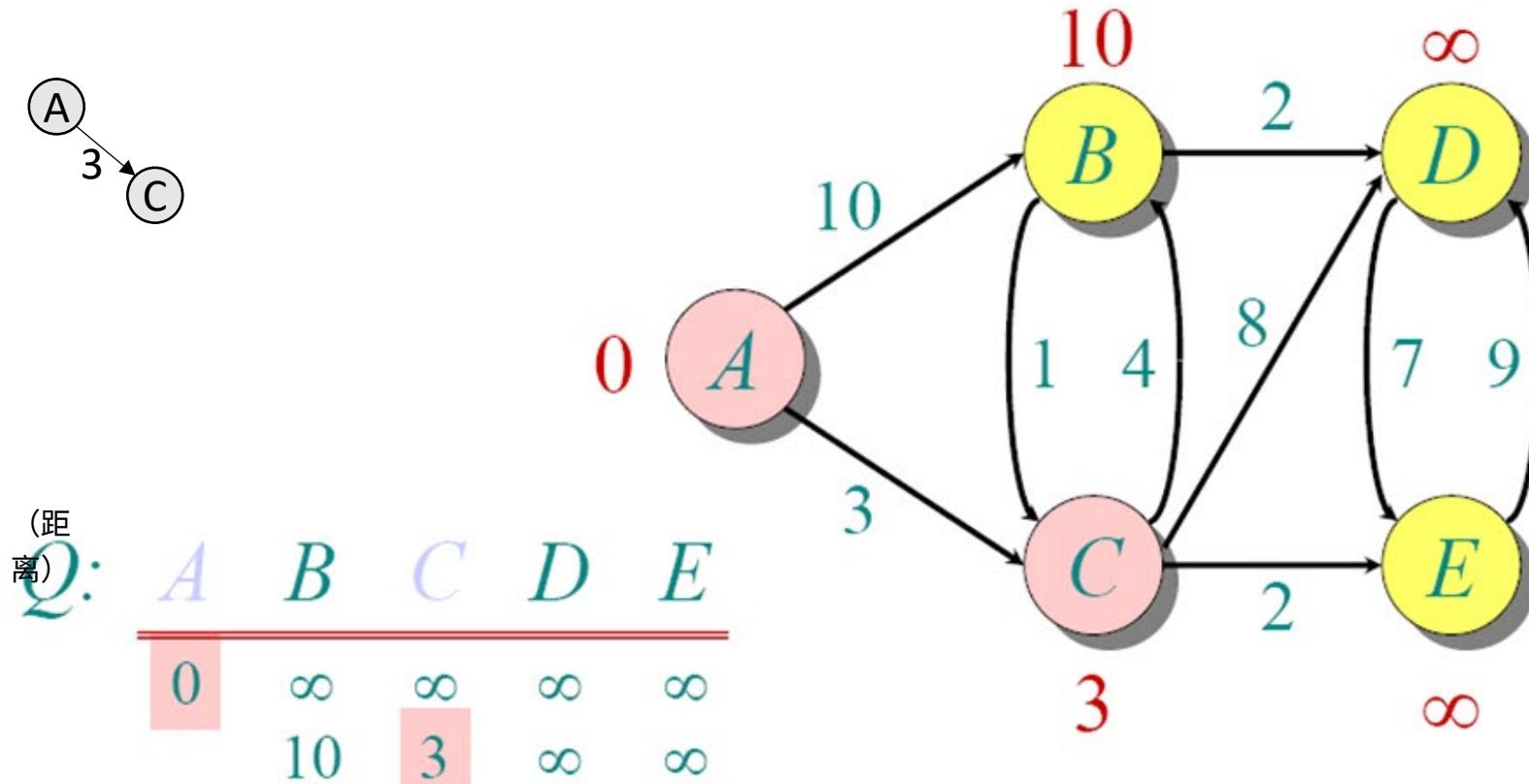
$Q:$ (距离)	A	B	C	D	E
	0	∞	∞	∞	∞
	10	3	∞	∞	



$S: \{ A \}$

Dijkstra 算法

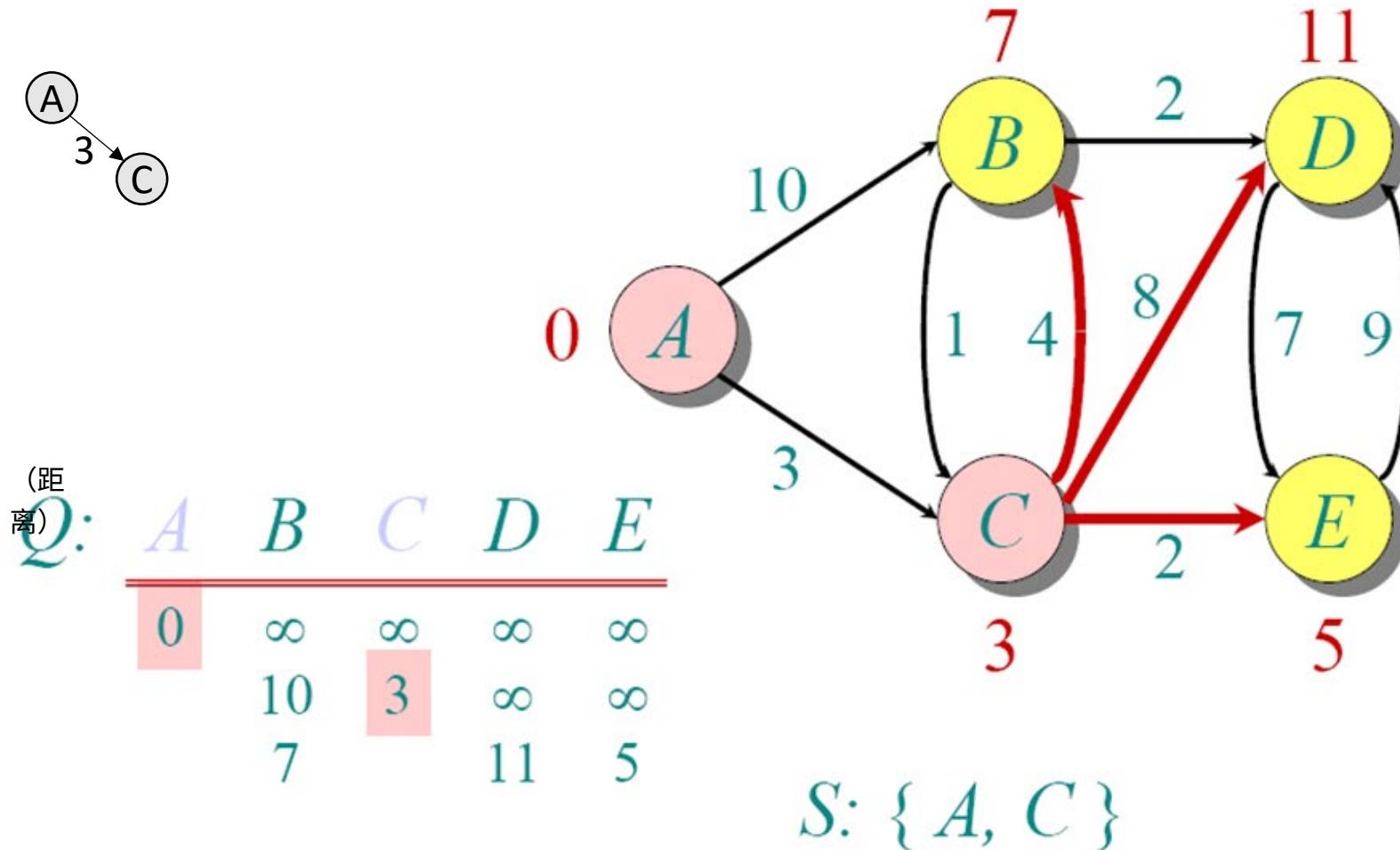
添加顶点 C



S: { A, C }

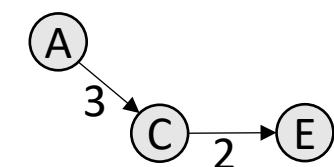
Dijkstra 算法

放松 C 的邻居节点

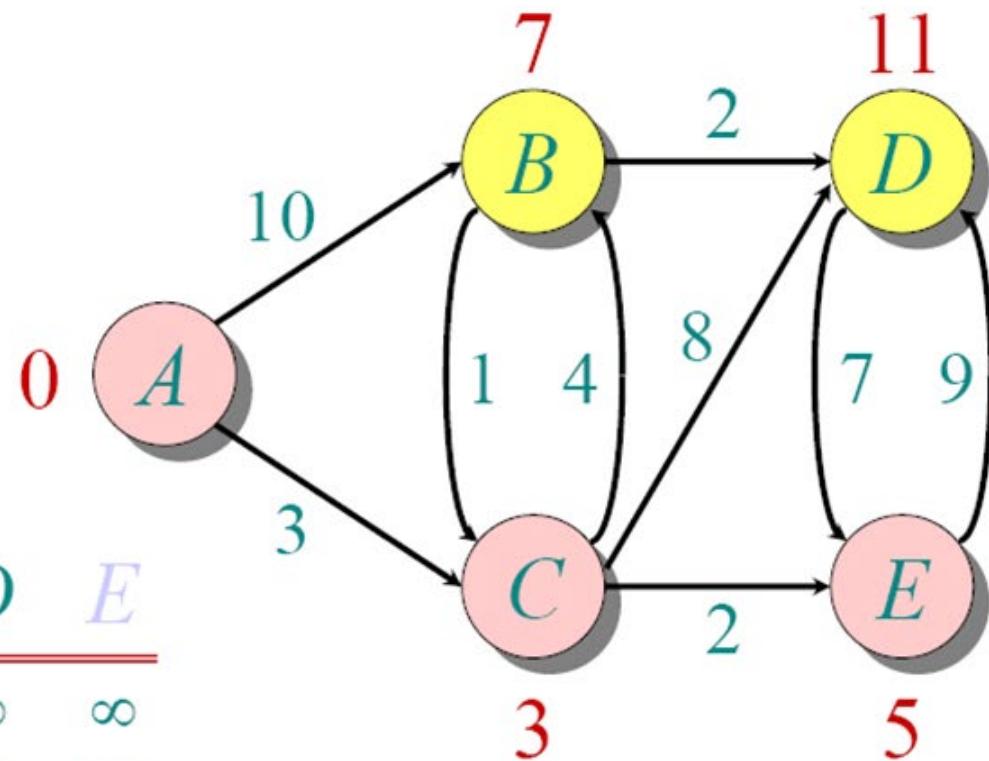


迪杰斯特拉示例

添加顶点 E



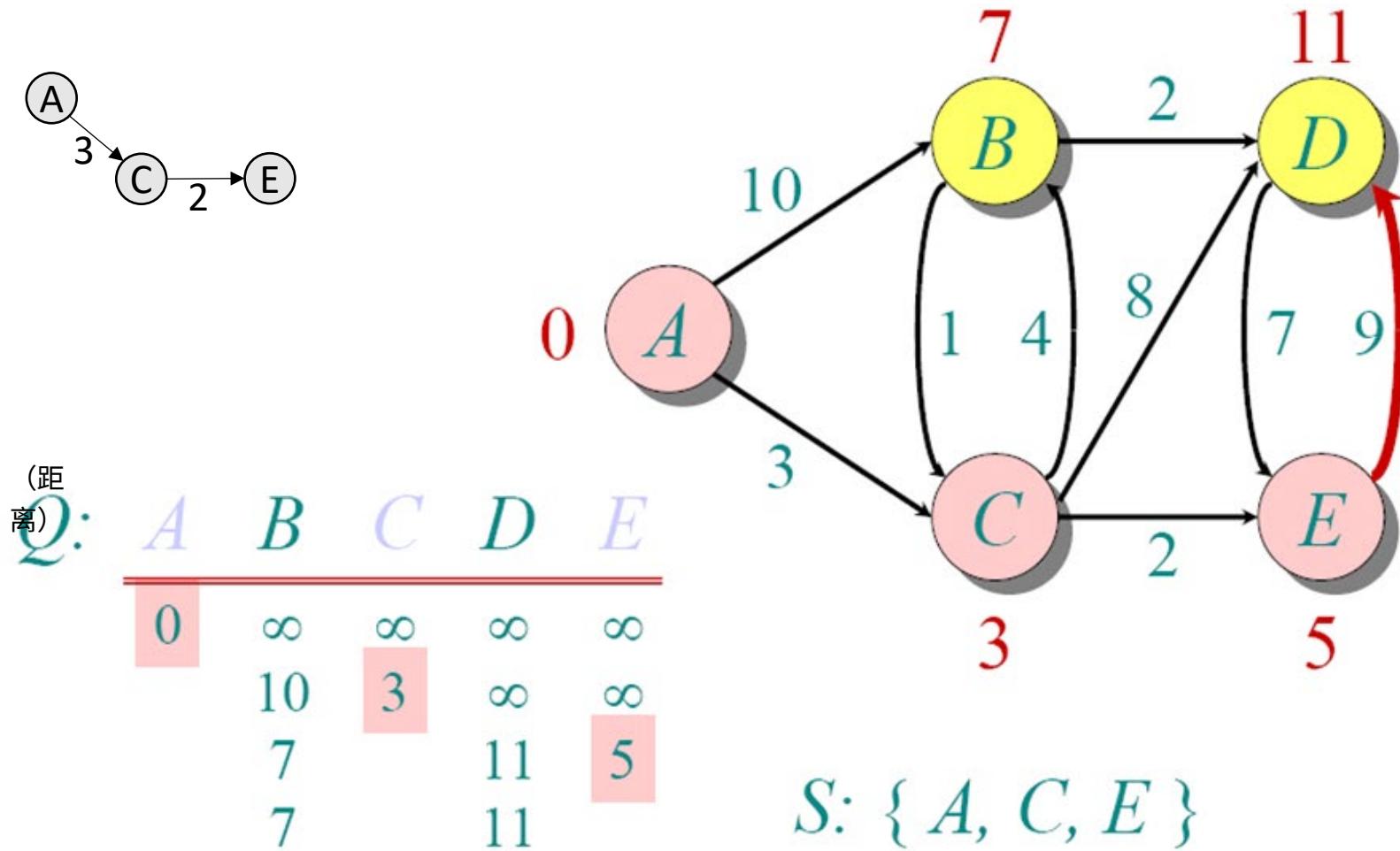
(距离)	A	B	C	D	E
0	0	∞	∞	∞	∞
10			3	∞	∞
7				11	5



S: { A, C, E }

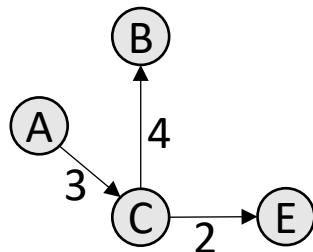
Dijkstra 算法

对 E 的邻居进行松弛



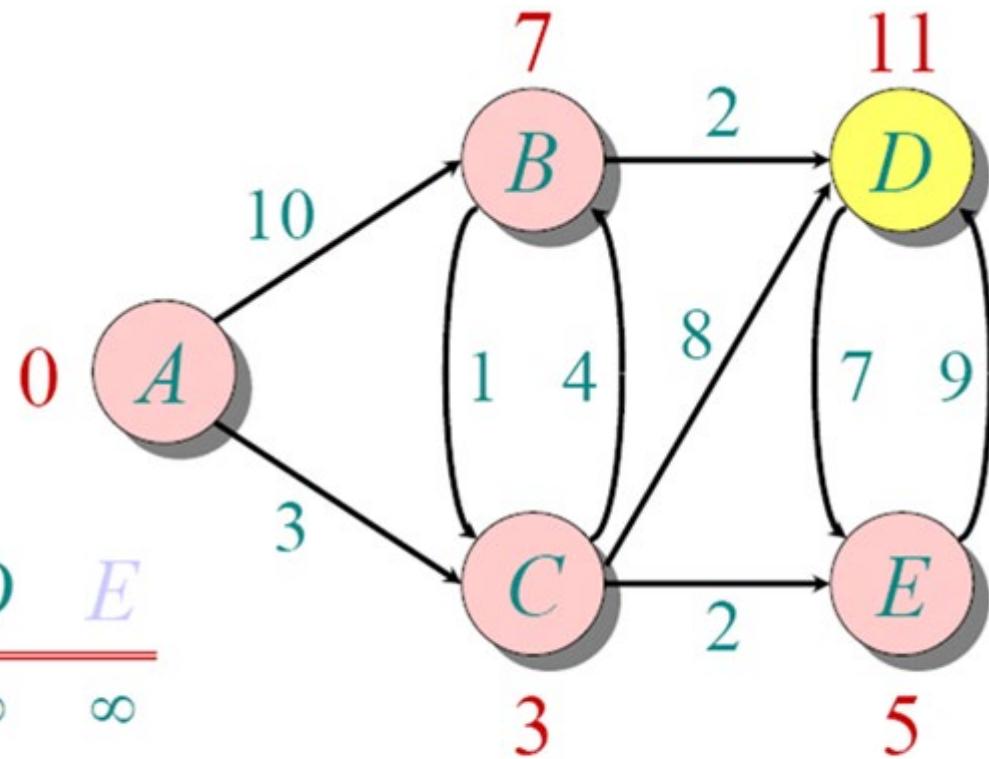
Dijkstra 算法

添加顶点 B



Q: (距离)

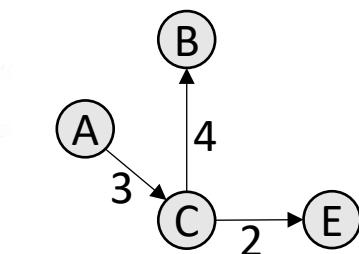
	A	B	C	D	E
0	0	∞	∞	∞	∞
10	10	3	∞	∞	
7	7	7	11	5	11



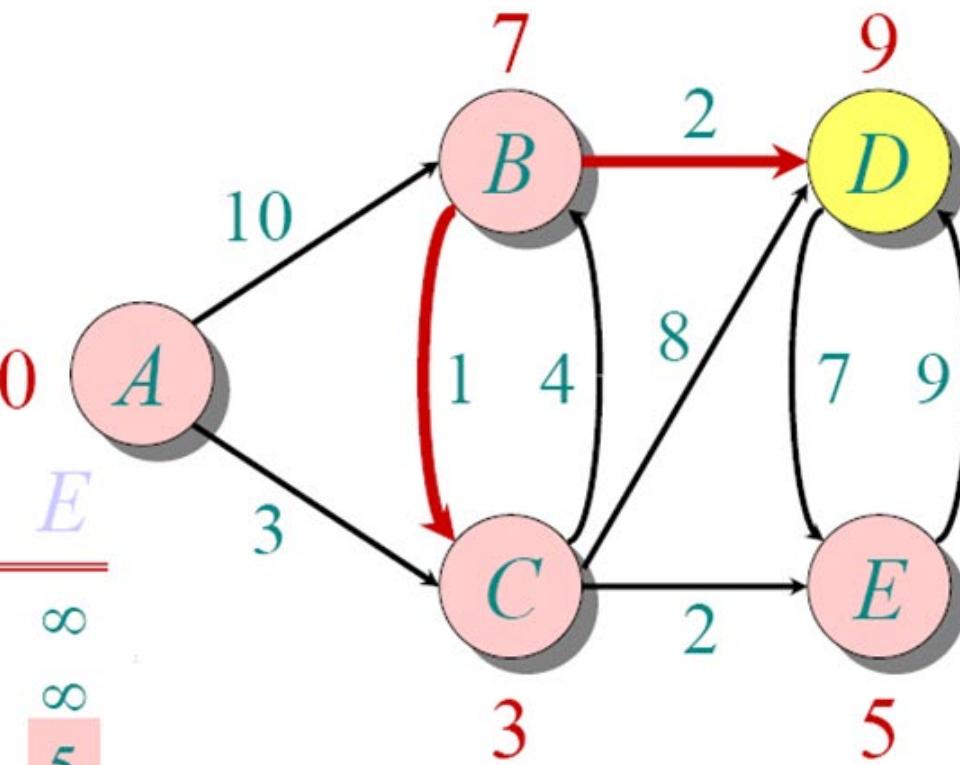
S: { A, C, E, B }

迪杰斯特拉示例

放松 B 的邻居

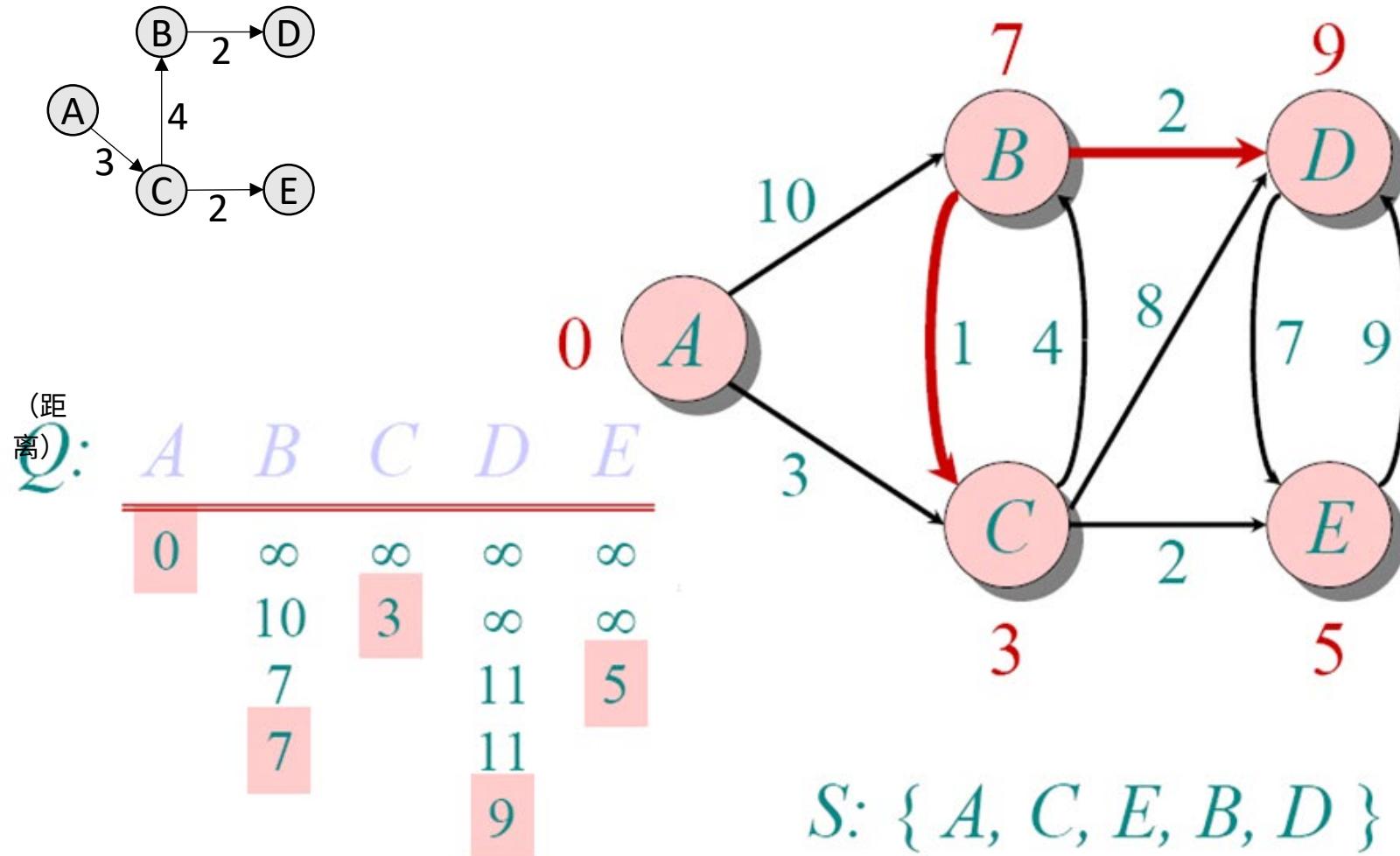


(距离)		A	B	C	D	E
0		0	∞	∞	∞	∞
10		10	3	∞	∞	
7		7	7	11	5	



$S: \{ A, C, E, B \}$

Dijkstra 算法



Dijkstra 算法

- 构建以起始顶点为根的最短路径树
- 这是一种贪心算法：每次加入当前最近的顶点，然后加入下一个最近的顶点，如此类推（直到所有顶点都被加入）

高层次伪代码：

1. 初始化 d 和 $prev$
2. 将所有顶点按与源点的距离作为键加入优先队列(PQ)
3. 当优先队列中仍有顶点时
4. 从优先队列中取出下一个顶点 u
5. 对于每个与 u 相邻的顶点 v
6. 如果 v 仍在优先队列中，则松弛 v

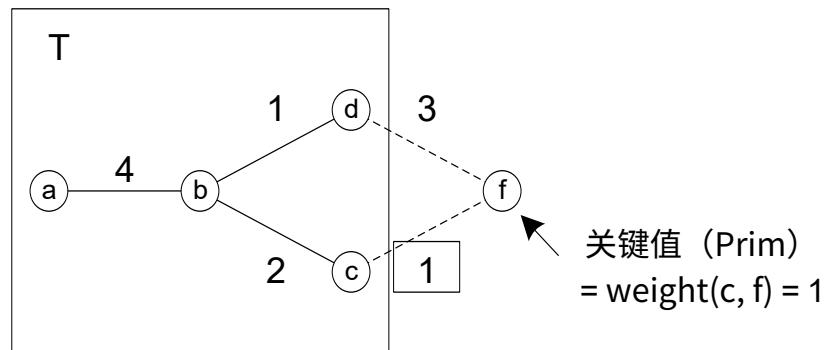
1. 松弛(v):
 2. 如果 $d[u] + w(u,v) < d[v]$
 3. $d[v] \leftarrow d[u] + w(u,v)$
 4. $prev[v] \leftarrow u$
 5. $PQ.updateKey(d[v], v)$

来自 Dijkstra 算法的输出

- Dijkstra 算法有（至少）两种可能的输出：
 - 从 v 到所有其他顶点的最短路径树
 - 从 v 到所有顶点的最短路径总费用的列表（映射）其他顶点。即该列表告诉你对于从 v 可达的所有顶点的 “ \min^- $distance(v, w)$ ”。

Dijkstra 与 Prim 的相似性

- 两者都从图 G 中累积一棵边构成的树 T
- 每次迭代：选择与当前已构建树相邻的最小优先级边
- 在 Prim 算法中，边的优先级就是该边的权重



- 在 Dijkstra 算法中，“优先级” 是边 (u, v) 的权重加上从起点到 v 的父节点的距离

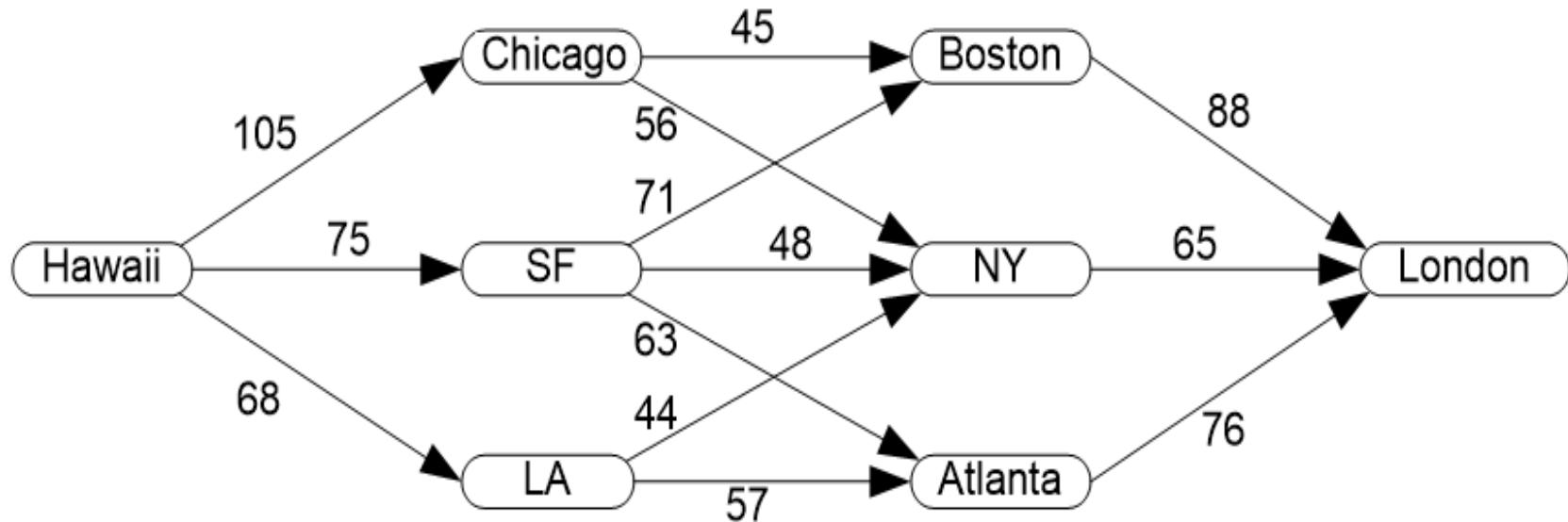
Dijkstra 算法的示例应用

- 假设伦敦想从夏威夷购买新鲜菠萝。
- 没有直飞航班，但有许多可能的中转路线。
- 为了使总体运输成本最低，最佳路线是什么？

输入：城市间运费

- 檀香山到芝加哥 105
- 檀香山到旧金山 75
- 檀香山到洛杉矶 68
- 芝加哥到波士顿 45
- 芝加哥到纽约 56
- 旧金山 到 波士顿 71
- 旧金山 到 纽约 48
- 旧金山 到 亚特兰大 63
- 洛杉矶 到 纽约 44
- 洛杉矶 到 亚特兰大 57
- 波士顿 到 伦敦 88
- 纽约 到 伦敦 65
- 亚特兰大 到 伦敦 76

问题的图模型

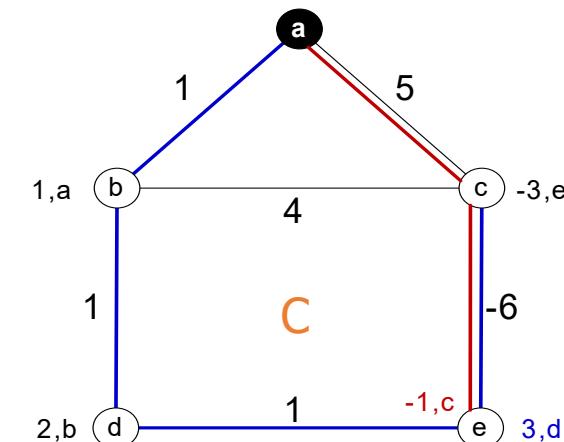
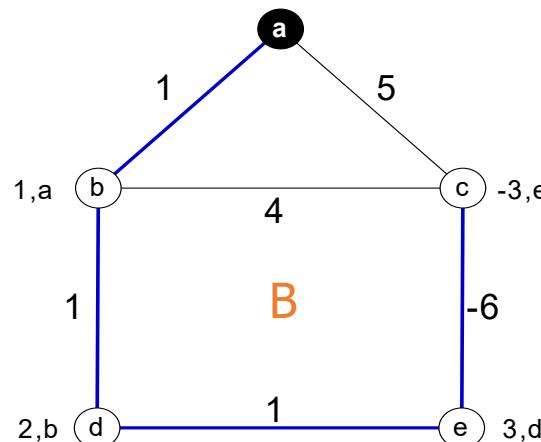
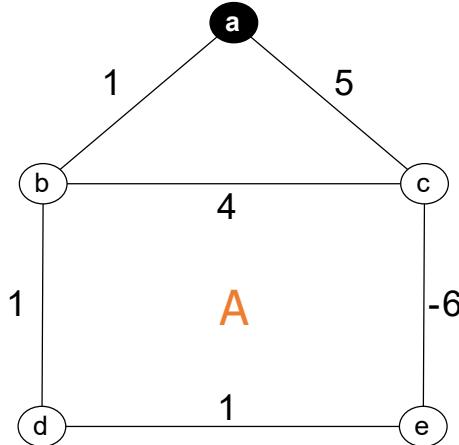


Apply Dij^{kstra 的算法}
g查找最便宜
(附：到其它所有城市的最低费用)

p从夏威夷到伦敦的最低费用

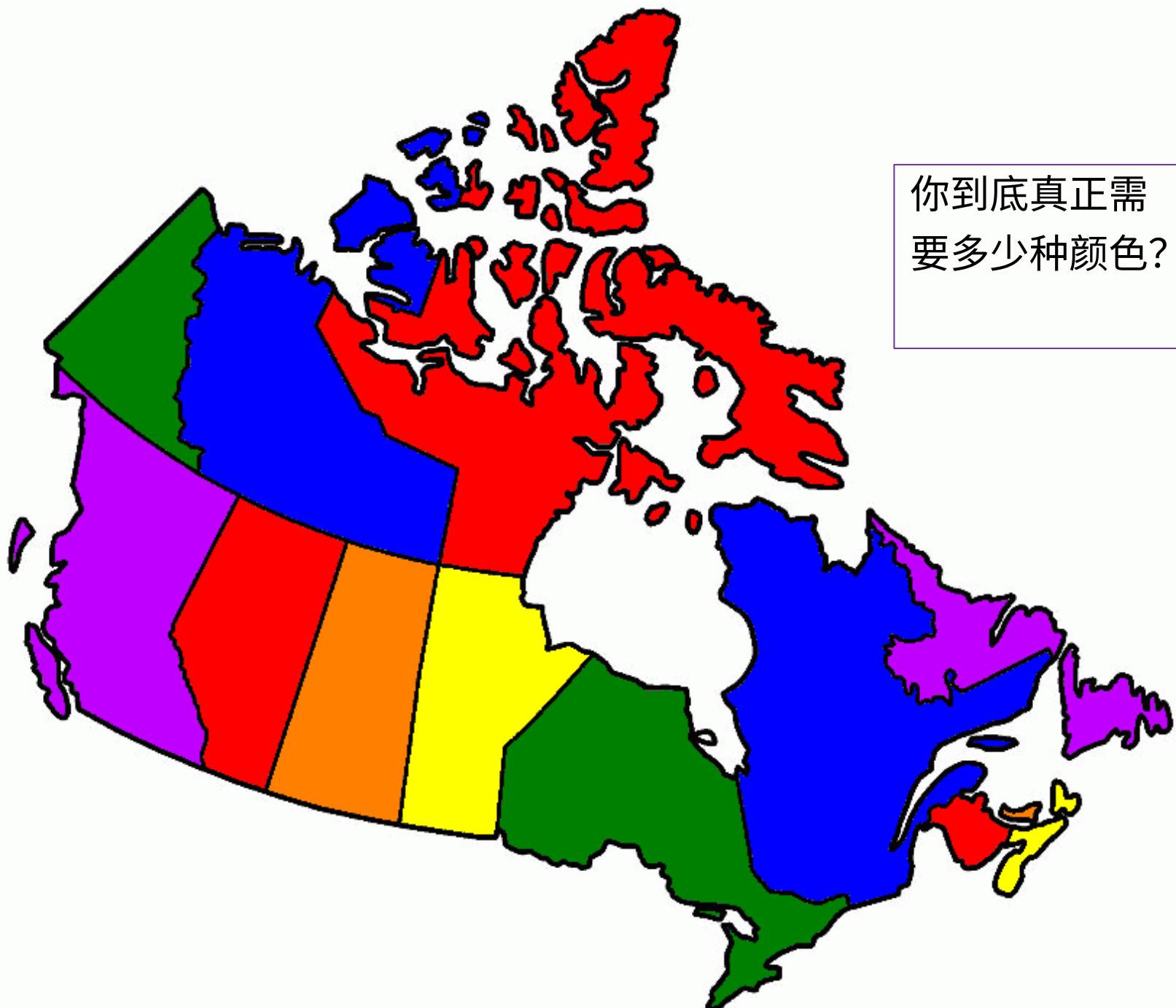
Dijkstra 的局限：负权边

- Dijkstra 算法在存在负权边时不适用
- 如果我们向 T 中添加一条新边，且它为负权，那么可能存在通过该新顶点到已在 T 中顶点的更短路径
- 例如，考虑下面的图 A。
 - 图 B 是在图 A 上运行 Dijkstra 算法的结果。
 - 但显然在图 C 中存在一条例如 a-c-e 的路径，它比在图 B 中找到的路径更短。因此在这个包含负边权的图上，Dijkstra 算法并不适用。



贪心算法：图着色

教科书：多次提及，但未深入讲解。请在索引中查找
“图着色”。



你到底真正需
要多少种颜色?

地图着色

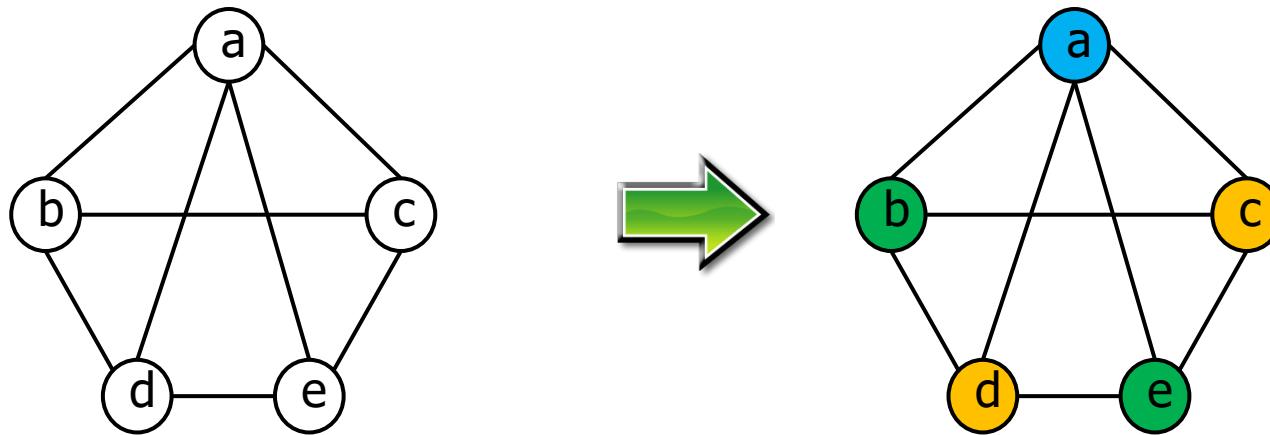
- 问题：为地图上的区域着色
 - 共享边界的区域必须使用不同的颜色
 - 仅在单点相遇不构成共享边界
- 作为一个判定问题：
 - 这幅地图可以用 N 种颜色着色吗？
- 作为一个优化问题：
 - 为给这张地图着色，所需的最少颜色数是多少？

图表示

- 为每个区域创建一个顶点
- 如果区域相邻则在它们之间加一条边
- 将问题重新表述为图论问题：
 - 为图的顶点分配颜色，使得相邻的顶点颜色不同

图着色问题

- 用尽可能少的颜色对图着色，要求相邻的两个顶点颜色不同
- 示例：

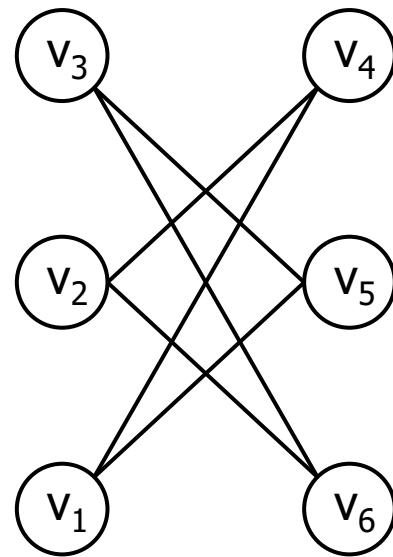
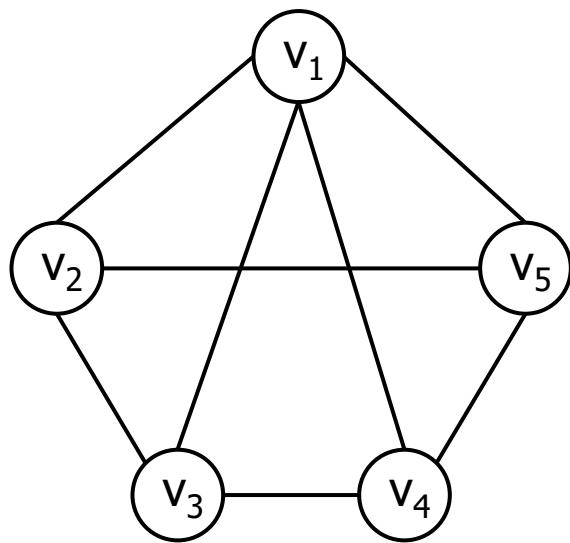


我们说该图是3-可着色

图着色 —— 贪心算法

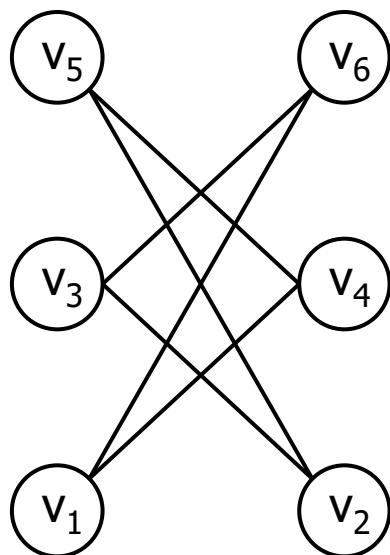
- 从只有一种颜色开始
- 按特定顺序考虑顶点 v_1, \dots, v_n
- 对于每个 v_i , 分配第一个未被任何 v_i 的邻居使用的可用颜色
- 如果所有颜色都被邻居使用, 则添加一种新颜色

示例



这个算法是最优的吗？

- 考虑前面的图，但顶点编号不同



- 之前只需要两种颜色
- 考虑顶点的顺序很重要
- 贪心算法并不总是产生最优解
- 但像蛮力那样，它们通常也值得考虑
因为它们可能易于实现

谜题——仅供娱乐！

- 制作一个表示平面地图并且需要4种颜色的图

练习题

1. 第9.1章，第324页，第9题
2. 第9.2章，第331页，第1、2题
3. 第9.3章，第337页，第1、2、4题