

算法设计与分析

第五讲 贪心算法

哈尔滨工业大学
李穆



找零钱问题：给定面额为1, 2, 5, 10, 20, 50的硬币以及一个总金额n，如何用最少的硬币数量凑成总金额n？



找零钱问题：给定面额为1, 2, 5, 10, 20, 50的硬币以及一个总金额n，如何用最少的硬币数量凑成总金额n？

◆分治算法



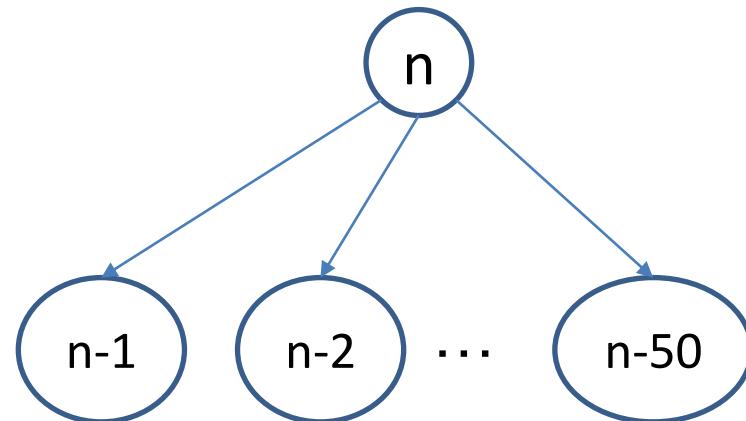
找零钱问题：给定面额为1, 2, 5, 10, 20, 50的硬币以及一个总金额n，如何用最少的硬币数量凑成总金额n？

- ◆ 分治算法
 - 复杂度



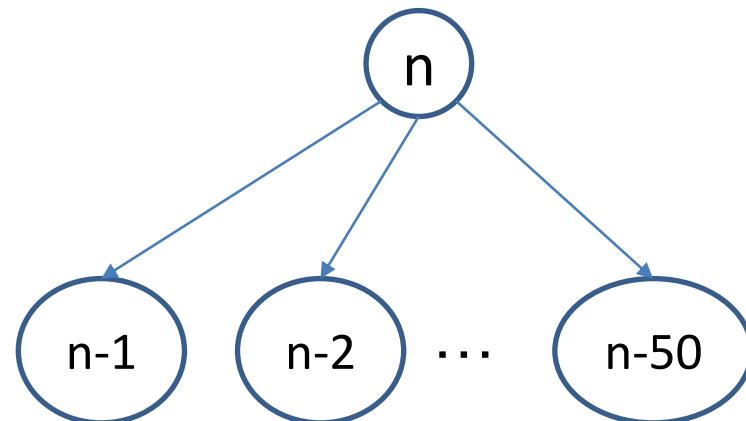
找零钱问题：给定面额为1, 2, 5, 10, 20, 50的硬币以及一个总金额n，如何用最少的硬币数量凑成总金额n？

- ◆ 分治算法
 - 复杂度



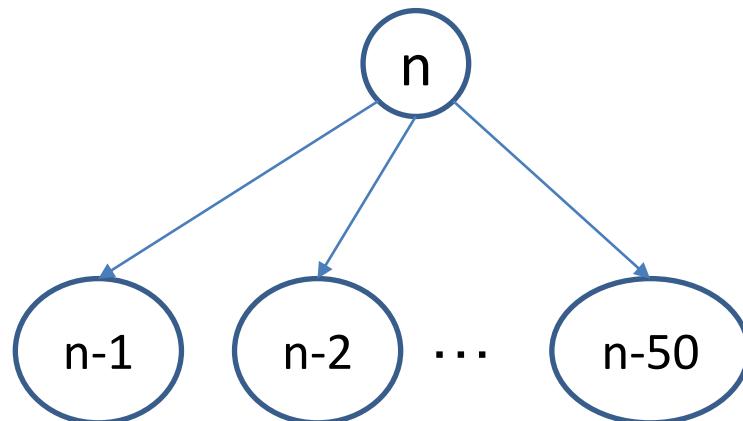
找零钱问题：给定面额为1, 2, 5, 10, 20, 50的硬币以及一个总金额n，如何用最少的硬币数量凑成总金额n？

- ◆ 分治算法
 - 复杂度
- 动态规划



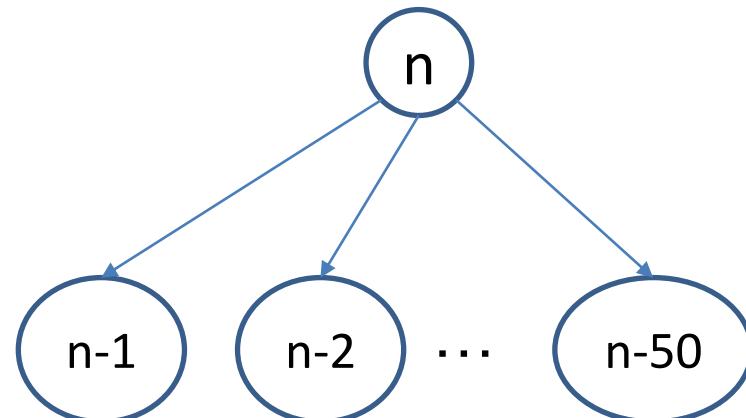
找零钱问题：给定面额为1, 2, 5, 10, 20, 50的硬币以及一个总金额n，如何用最少的硬币数量凑成总金额n？

- ◆ 分治算法
 - 复杂度
- 动态规划
 - ✓ 时间复杂度
 - ✓ 空间复杂度



找零钱问题：给定面额为1, 2, 5, 10, 20, 50的硬币以及一个总金额n，如何用最少的硬币数量凑成总金额n？

- ◆ 分治算法
 - 复杂度
- 动态规划
 - ✓ 时间复杂度
 - ✓ 空间复杂度
- 更好方法？



提纲

5.1 贪心法的基本原理

5.2 任务安排问题

5.3 哈夫曼编码问题

5.4 最小生成树



贪心算法的基本概念

- 贪心算法的基本思想

贪心算法的基本概念

- 贪心算法的基本思想
 - 求解最优化问题的算法包含一系列步骤
 - 每一步都有一组这样

贪心算法的基本概念

- 贪心算法的基本思想

- 求解最优化问题的算法包含一系列步骤
- 每一步都有一组这样
- 作出在当前看来最好的这样
- 希望通过作出局部优化这样达到全局优化这样

贪心算法的基本概念

- 贪心算法的基本思想

- 求解最优化问题的算法包含一系列步骤
- 每一步都有一组这样
- 作出在当前看来最好的这样
- 希望通过作出局部优化这样达到全局优化这样
- 贪心算法不一定总产生优化解
- 贪心算法是否产生优化解，需严格证明

贪心算法的基本概念

- 贪心算法的基本思想
 - 求解最优化问题的算法包含一系列步骤
 - 每一步都有一组这样
 - 作出在当前看来最好的这样
 - 希望通过作出局部优化这样达到全局优化这样
 - 贪心算法不一定总产生优化解
 - 贪心算法是否产生优化解，需严格证明
- 贪心算法产生优化解的条件
 - 贪心选择性
 - 优化子结构

贪心这样性



贪心这样性



• 贪心这样性

若一个优化问题的全局优化解可以通过局部优化这样得到，则该问题称为具有贪心这样性.



贪心这样性



- 贪心这样性

若一个优化问题的全局优化解可以通过局部优化这样得到，则该问题称为具有贪心这样性.

- 一个问题是否具有贪心这样性需证明

优化子结构



若一个优化问题的优化解包含它的子问题的优化解，则称其具有优化子结构



与动态规划方法的比较



与动态规划方法的比较

- 动态规划方法可用的条件
 - 优化子结构
 - 子问题重叠性
 - 子问题空间小



与动态规划方法的比较

- 动态规划方法可用的条件

- 优化子结构
 - 子问题重叠性
 - 子问题空间小

- 贪心方法可用的条件

- 优化子结构
 - 贪心选择性



与动态规划方法的比较

- 动态规划方法可用的条件
 - 优化子结构
 - 子问题重叠性
 - 子问题空间小
- 贪心方法可用的条件
 - 优化子结构
 - 贪心选择性
- 可用动态规划方法时，贪心方法可能不适用

贪心算法正确性证明方法

- 证明算法所求解的问题具有优化子结构
- 证明算法所求解的问题具有贪心选择性
- 证明算法确实按照贪心选择性进行局部优化选择



与动态规划方法的比较

- 共同点：贪心算法和动态规划算法都要求问题具有优化子结构性质
- 动态规划方法
 - 在每一步所做的选择通常依赖于子问题的解
 - 以自底向上方式，先解小子问题，再求解大子问题
- 贪心方法
 - 在每一步先做出当前看起来最好的选择
 - 然后再求解本次选择后产生的剩余子问题
 - 每次选择既不依赖于子问题的解，也不依赖于未来的选择
 - 以自顶向下方式，逐步进行贪心选择，不断减少子问题规模

提纲

5.1 贪心法的基本原理

5.2 任务安排问题

5.3 哈夫曼编码问题

5.4 最小生成树



人工智能导论

电工学

操作系统

算法设计与
分析

大学物理

机器学习概率论

汇编语言

程序语言设计

time

Horseback Riding (骑马)



- 1、教室排课，如何提高利用率
- 2、参加会议，很多场报告，听哪些
- 3、出去旅游，如何安排日程，参观哪些景点

运动安排问题

设 $S = \{a_1, a_2, \dots, a_n\}$ 是 n 个活动的集合，各个活动使用同一个资源，资源在同一时间只能为一个活动使用。每个活动 a_i 有起始时间 s_i ，终止时间 f_i ， $s_i < f_i$ 。



运动安排问题

设 $S = \{a_1, a_2, \dots, a_n\}$ 是 n 个活动的集合，各个活动使用同一个资源，资源在同一时间只能为一个活动使用。每个活动 a_i 有起始时间 s_i ，终止时间 f_i ， $s_i < f_i$ 。

问：如何安排活动，能够安排尽量多的活动。最多能安排多少个活动？



运动安排问题

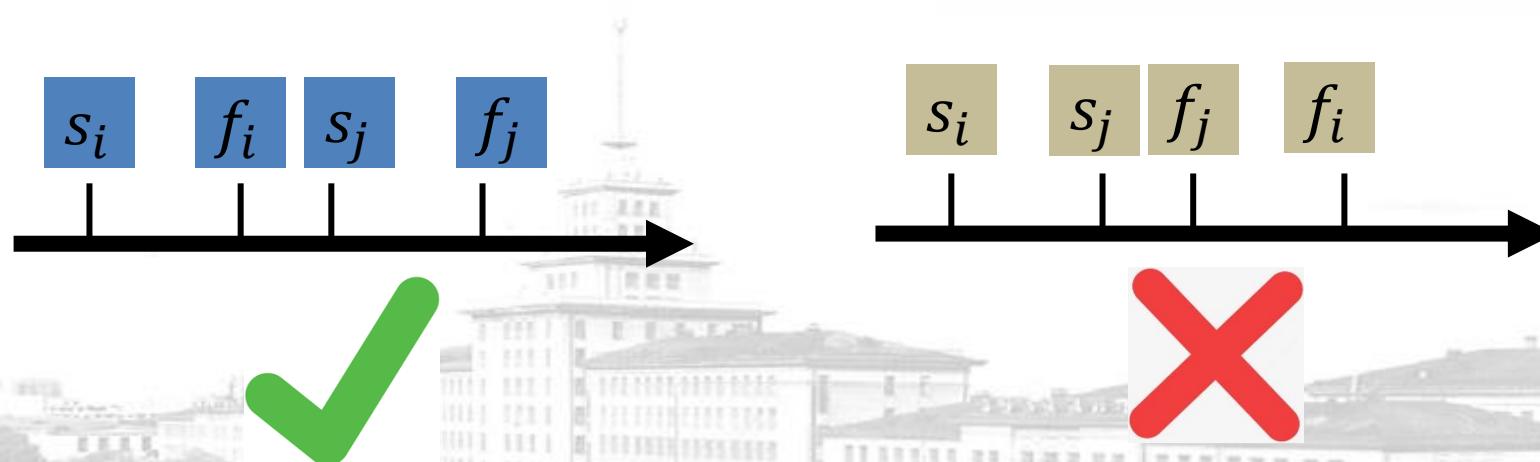


场地

人员

活动的相容性

- 如果两个活动的时间段没有重叠，那么这两个活动可以同时安排，那么它们就是相容的



运动安排问题

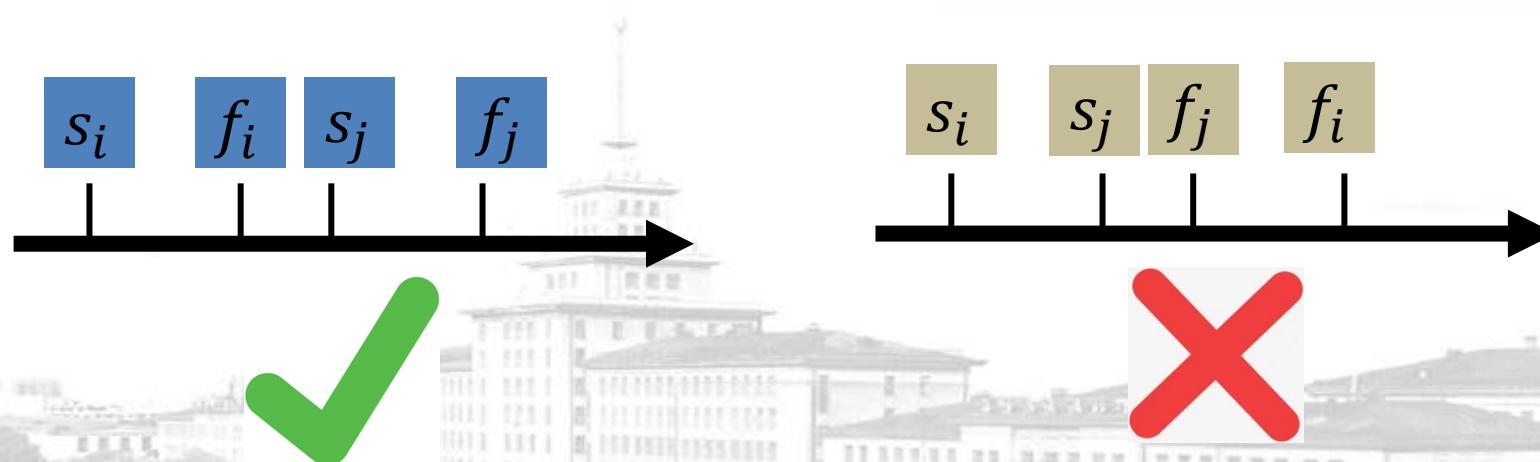


场地

人员

活动的相容性

- 如果两个活动的时间段没有重叠，那么这两个活动可以同时安排，那么它们就是相容的
- 条件： $s_i > f_j$ 或者 $s_j > f_i$

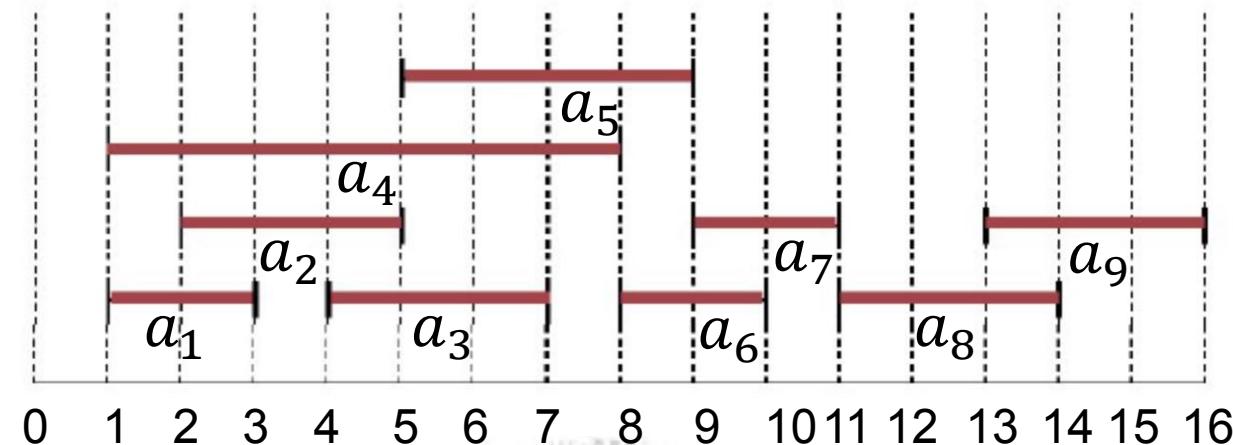


问题的定义



例子：

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



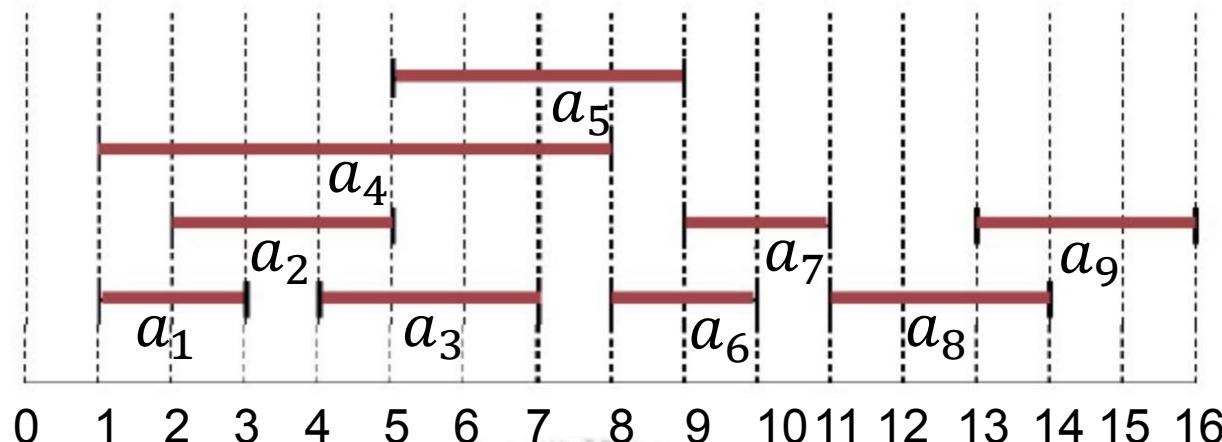
可能的最大活动集合

问题的定义



例子：

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



可能的最大活动集合

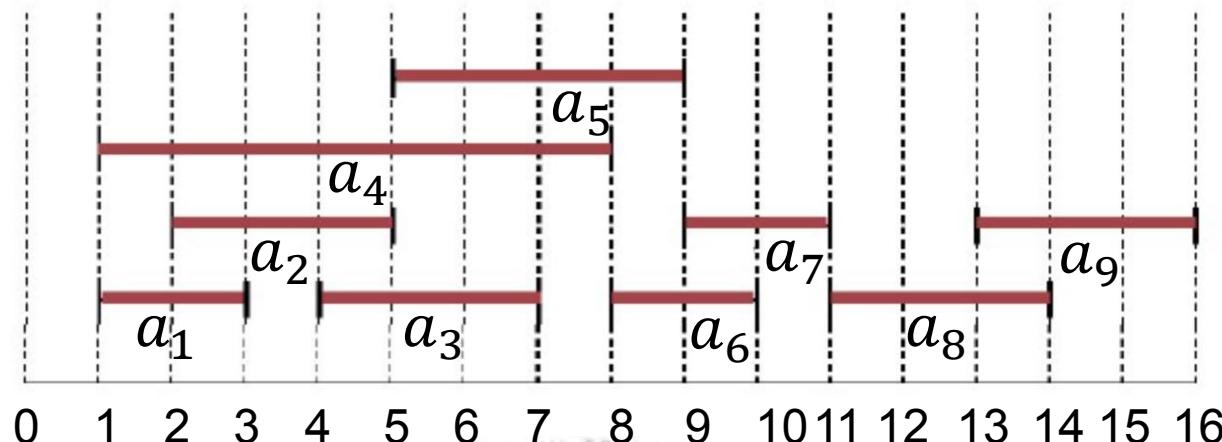
- $\{a_1, a_3, a_6, a_8\}$

问题的定义



例子：

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



可能的最大活动集合

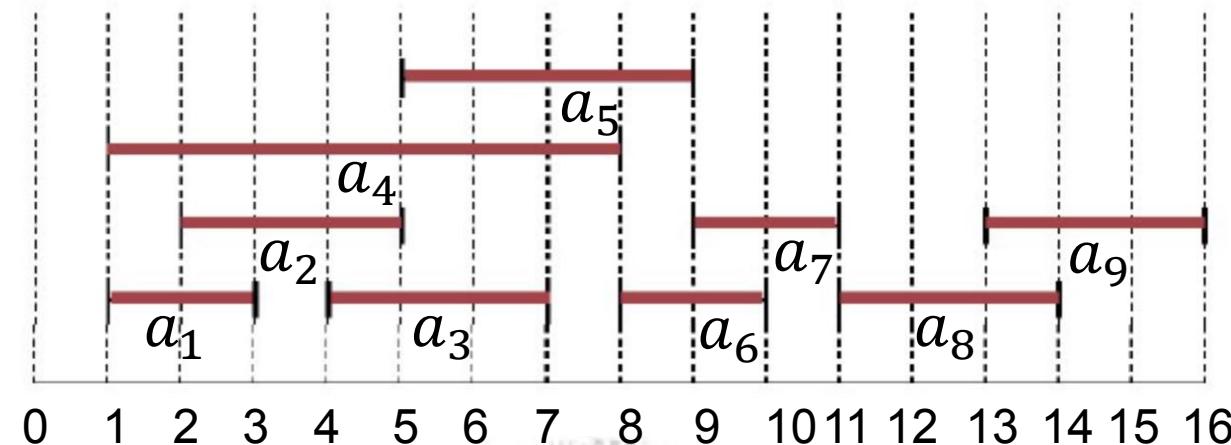
- $\{a_1, a_3, a_6, a_8\}$
- $\{a_2, a_5, a_7, a_9\}$

问题的定义



例子：

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



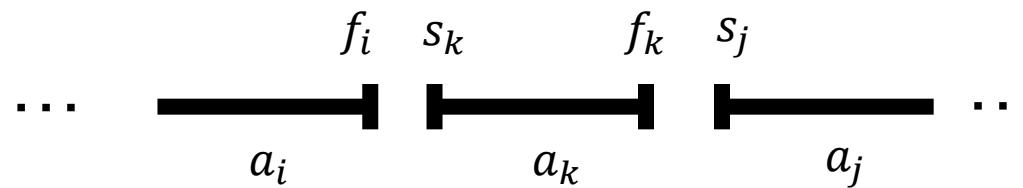
可能的最大活动集合

- $\{a_1, a_3, a_6, a_8\}$
- $\{a_2, a_5, a_7, a_9\}$
- $\{a_1, a_5, a_7, a_9\}$

- 子问题定义：

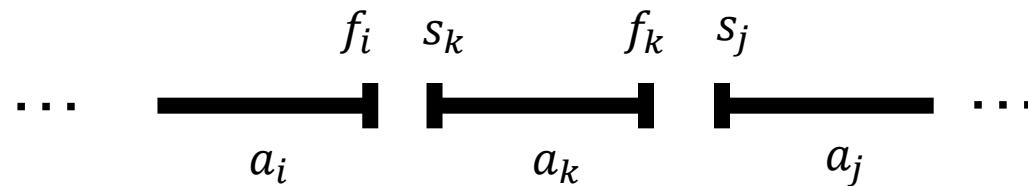


● 子问题定义：



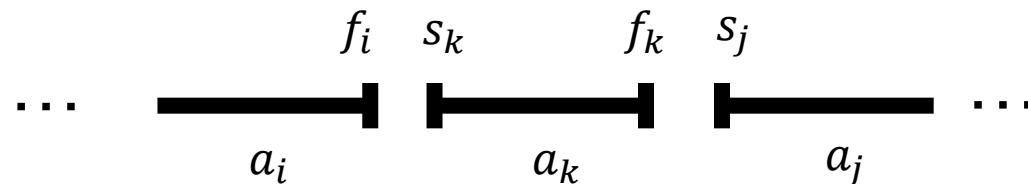
- 子问题定义：

$S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$, 在活动 a_i 结束之后开始，在活动 a_j 开始前结束的活动集合



- 子问题定义：

$S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$, 在活动 a_i 结束之后开始，在活动 a_j 开始前结束的活动集合



如何表示原来的问题集合 S ? (边界问题)

- ◆ $a_0 = [-1, 0); a_{n+1} = [f_{max} + 1, f_{max} + 2)$
- ◆ $S_{0n} = S; 0 < k < n + 1$

- 子问题划分

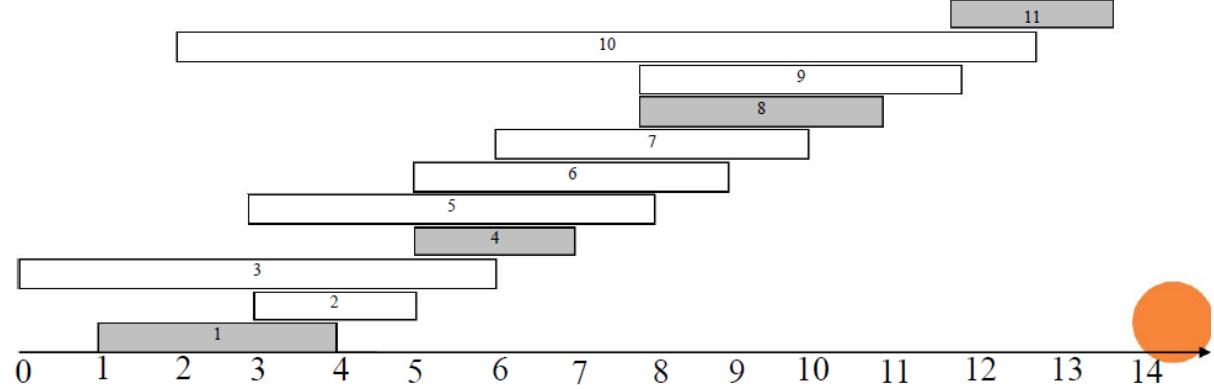
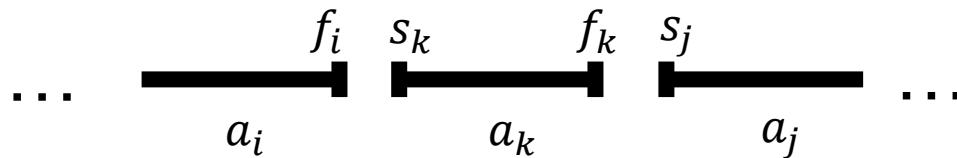
假设 a_k 包含在 S_{ij} 的一个最优解中，那么可以把 S_{ij} 划分成两个子问题：

- ◆ S_{ik} : 开始于 a_i 结束后，结束在 a_k 开始前的活动集合
- ◆ S_{kj} : 开始于 a_k 结束后，结束在 a_j 开始前的活动集合

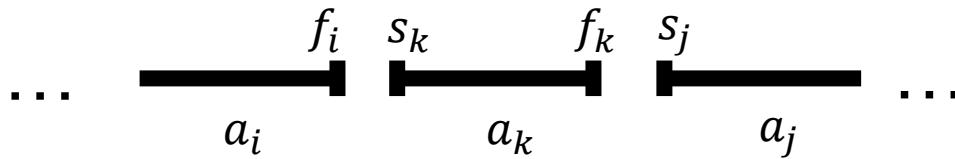
- 优化子结构证明：

- $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
- $|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$
- 若 S_{ij} 的最优解中包含 a_k ，那么 A_{ik} 和 A_{kj} 分别是 S_{ik} 和 S_{kj} 的最优解。（反证法）

- 子问题定义: $S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$

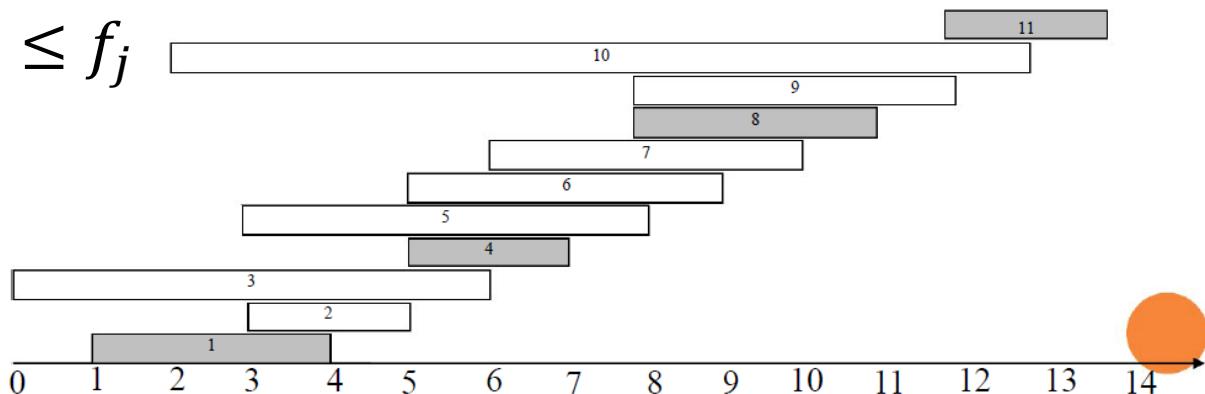


- 子问题定义: $S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$



如何将当前问题拆分成子问题?

- ◆ 以活动结束时间单调递增的方式对活动进行排序
- ◆ $f_0 \leq f_1 \leq f_2 \leq \dots \leq f_n \leq f_{n+1}$
- ◆ 如果 $i \leq j$, 那么 $f_i \leq f_j$



给定: $f_0 \leq f_1 \leq f_2 \leq \dots \leq f_n \leq f_{n+1}$

那么, $i \geq j \rightarrow S_{ij} = \emptyset$ 。

证明:

若存在某个 $a_k \in S_{ij}$, 那么

$f_i \leq s_k < f_k \leq s_j < f_j$ 。

另一方面, $i \geq j \rightarrow f_i \geq f_j$ 。

矛盾 !

给定: $f_0 \leq f_1 \leq f_2 \leq \dots \leq f_n \leq f_{n+1}$

那么, $i \geq j \rightarrow S_{ij} = \emptyset$ 。

证明:

若存在某个 $a_k \in S_{ij}$, 那么

$f_i \leq s_k < f_k \leq s_j < f_j$ 。

另一方面, $i \geq j \rightarrow f_i \geq f_j$ 。

矛盾 !

- 只需考虑 $0 \leq i < j \leq n + 1$ 的情况, 其他子问题是空集。

递归定义优化解的代价：

- ◆ 令 $c[i, j]$ 表示 S_{ij} 中最大相容活动的数量
- ◆ $i \geq j \rightarrow c[i, j] = 0$
- ◆ 若 $S_{ij} \neq \emptyset$, 那么一定存在一个 $a_k \in S_{ij}$, 使得 $c[i, j] = c[i, k] + 1 + c[k, j]$
- ◆ $c[i, j] = \begin{cases} 0, & \text{若 } S_{ij} = \emptyset \\ \max_{\substack{i < k < j \\ a_k \in S_{ij}}} \{c[i, k] + 1 + c[k, j]\}, & \text{若 } S_{ij} \neq \emptyset \end{cases}$

运动安排问题

$$c[i,j] = \begin{cases} 0, & \text{若 } S_{ij} = \emptyset \\ \max_{i < k < j} \{c[i,k] + 1 + c[k,j]\}, & \text{若 } S_{ij} \neq \emptyset \end{cases}$$

- 复杂度分析
 - ◆ 直接暴力递归（时间复杂度？）
 - ◆ 动态规划（时间复杂度？）
 - ◆ 有没有更简答的方法？

运动安排问题

设 $S = \{1, 2, \dots, n\}$ 是 n 个活动的集合，各个活动使用同一个资源，资源在同一时间只能为一个活动使用。每个活动 i 有起始时间 s_i ，终止时间 f_i ， $s_i < f_i$ 。

问：如何安排活动，能够安排尽量多的活动。最多能安排多少个活动？

思路：



运动安排问题

设 $S = \{1, 2, \dots, n\}$ 是 n 个活动的集合，各个活动使用同一个资源，资源在同一时间只能为一个活动使用。每个活动 i 有起始时间 s_i ，终止时间 f_i ， $s_i < f_i$ 。

问：如何安排活动，能够安排尽量多的活动。最多能安排多少个活动？

思路：

- 优先安排时间最短的活动

运动安排问题

设 $S = \{1, 2, \dots, n\}$ 是 n 个活动的集合，各个活动使用同一个资源，资源在同一时间只能为一个活动使用。每个活动 i 有起始时间 s_i ，终止时间 f_i ， $s_i < f_i$ 。

问：如何安排活动，能够安排尽量多的活动。最多能安排多少个活动？

思路：

- 优先安排时间最短的活动
- 优先安排开始时间最早的活动



运动安排问题

设 $S = \{1, 2, \dots, n\}$ 是 n 个活动的集合，各个活动使用同一个资源，资源在同一时间只能为一个活动使用。每个活动 i 有起始时间 s_i ，终止时间 f_i ， $s_i < f_i$ 。

问：如何安排活动，能够安排尽量多的活动。最多能安排多少个活动？

思路：

- 优先安排时间最短的活动
- 优先安排开始时间最早的活动
- 优先安排结束时间最早的活动

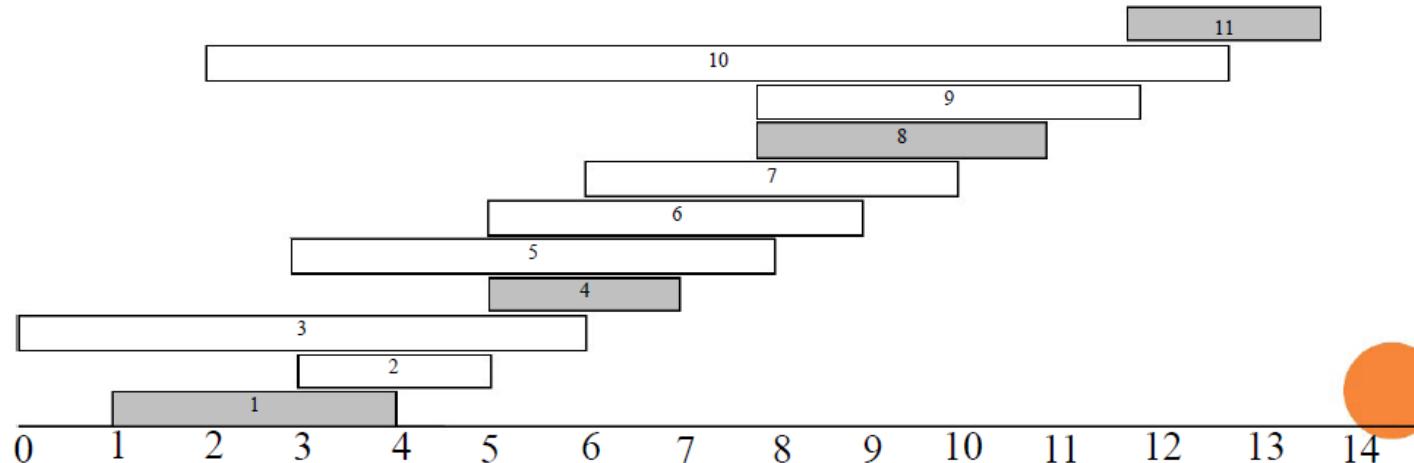
运动安排問題

贪心算法

引理1：

$S_{ij} \neq \emptyset$, 令 a_m 表示 S_{ij} 中结束最早的活动, 也即 $a_m = \min\{f_k : a_k \in S_{ij}\}$, 那么

1. a_m 必然包含在 S_{ij} 的某个最大相容集合里 (最优解)
2. $S_{im} = \emptyset$, 可以排除一个子问题, 仅仅需要考虑 S_{mj}



运动安排问题

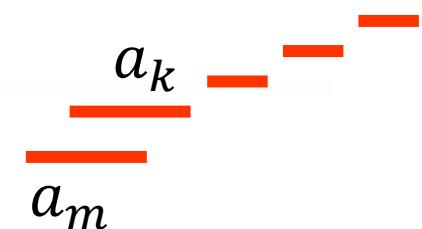
贪心算法

引理1：

$S_{ij} \neq \emptyset$, 令 a_m 表示 S_{ij} 中结束最早的活动, 也即 $a_m =$

$\min\{f_k : a_k \in S_{ij}\}$, 那么

1. a_m 必然包含在 S_{ij} 的某个最大相容集合里 (最优解)



运动安排问题

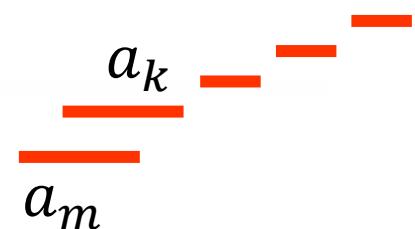
贪心算法

引理1：

$S_{ij} \neq \emptyset$, 令 a_m 表示 S_{ij} 中结束最早的活动, 也即 $a_m = \min\{f_k : a_k \in S_{ij}\}$, 那么

1. a_m 必然包含在 S_{ij} 的某个最大相容集合里 (最优解)

证明：令 A_{ij} 表示 S_{ij} 的一个最优解，将 A_{ij} 中的活动按照结束时间升序排列， a_k 为 A_{ij} 中的第一个活动。



运动安排问题

贪心算法

引理1：

$S_{ij} \neq \emptyset$, 令 a_m 表示 S_{ij} 中结束最早的活动, 也即 $a_m = \min\{f_k : a_k \in S_{ij}\}$, 那么

1. a_m 必然包含在 S_{ij} 的某个最大相容集合里 (最优解)

证明：令 A_{ij} 表示 S_{ij} 的一个最优解，将 A_{ij} 中的活动按照结束时间升序排列， a_k 为 A_{ij} 中的第一个活动。

- 1、若 $a_m = a_k$, 那么引理显然成立



引理1：

$S_{ij} \neq \emptyset$, 令 a_m 表示 S_{ij} 中结束最早的活动, 也即 $a_m = \min\{f_k : a_k \in S_{ij}\}$, 那么

1. a_m 必然包含在 S_{ij} 的某个最大相容集合里 (最优解)

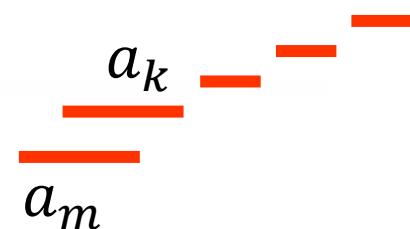
证明：令 A_{ij} 表示 S_{ij} 的一个最优解, 将 A_{ij} 中的活动按照结束时间升序排列, a_k 为 A_{ij} 中的第一个活动。

1、若 $a_m = a_k$, 那么引理显然成立

2、若 $a_m \neq a_k$, 构造解 $B_{ij} = A_{ij} - \{a_k\} + \{a_m\}$

a_m 结束在 a_k 前, 替换后 B_{ij} 也是一个相容集合

且 $|B_{ij}| = |A_{ij}|$, B_{ij} 也是 S_{ij} 的一个最优解



运动安排问题

$$c[i, j] = \begin{cases} 0, & \text{若 } S_{ij} = \emptyset \\ \max_{i < k < j} \{c[i, k] + 1 + c[k, j]\}, & \text{若 } S_{ij} \neq \emptyset \end{cases}$$

引理： $S_{ij} \neq \emptyset$ ，令 a_m 表示 S_{ij} 中结束最早的活动，也即 $a_m = \min\{f_k : a_k \in S_{ij}\}$ ，那么

1. a_m 必然包含在 S_{ij} 的某个最大相容集合里（最优解）
2. $S_{im} = \emptyset$ ，可以排除一个子问题，仅仅需要考虑 S_{mj}

	动态规划	引理（贪心）
需要求解子问题数量	2	1
需要考虑的情况	$O(j-i-1)$	1

引理2：设 a_m 是 S_{ij} 中结束最早的活动， A_{ij} 是 S_{ij} 的一个最优解且包含 a_m ，那么 $A_{ij} - \{a_m\}$ 是 S_{mj} 的一个优化解。

证明：(反证法)

假设 $A_{ij} - \{a_m\}$ 不是 S_{mj} 的一个优化解。

那么，设 B 是 S_{mj} 的一个优化解。

$$|B| > |A_{ij} - \{a_m\}|.$$

由于 a_m 不包含在 S_{mj} 中，那么

$B \cup \{a_m\}$ 是 A_{ij} 的一个解，且 $|B \cup \{a_m\}| > |A_{ij}|$,

矛盾！

引理3：设 a_{m1} 是 S_{ij} 中结束最早的活动， a_{m2} 是 $S_{m1,j}$ 中结束最早的活动，那么 S_{ij} 的一个优化解为 $A_{ij} = \bigcup_{t=1}^k \{a_{mt}\}$ 。

证明：用数学归纳法证明。设 S_{ij} 的优化解为某个 B_{ij}

- 1、当 $|B_{ij}| = 1$ ，根据引理1， $|A_{ij}| = |B_{ij}| = 1$ ，显然成立。
- 2、假设 $|B_{ij}| < k$ 时，结论成立，也即 $|A_{ij}| = |B_{ij}|$ 。

那么，当 $|B_{ij}| = k$ 时， $B_{ij} = \{a_{m1}\} \cup B_{m1,j}$ 。（贪心选择性）

根据引理2， $B_{m1,j}$ 是 $S_{m1,j}$ 的最优解。

$|B_{m1,j}| = |B_{ij}| - 1 < k$ ，利用归纳法 $B_{m1,j} = \bigcup_{t=2}^k \{a_{mt}\}$ 。

$B_{ij} = \{a_{m1}\} \cup B_{m1,j} = \{a_{m1}\} \cup \bigcup_{t=2}^k \{a_{mt}\} = \bigcup_{t=1}^k \{a_{mt}\}$

贪心算法

运动安排问题

输入: $S_{0,n+1}$;

输出: 最大相容集合

$A \leftarrow \{\}$;

$m \leftarrow 0$;

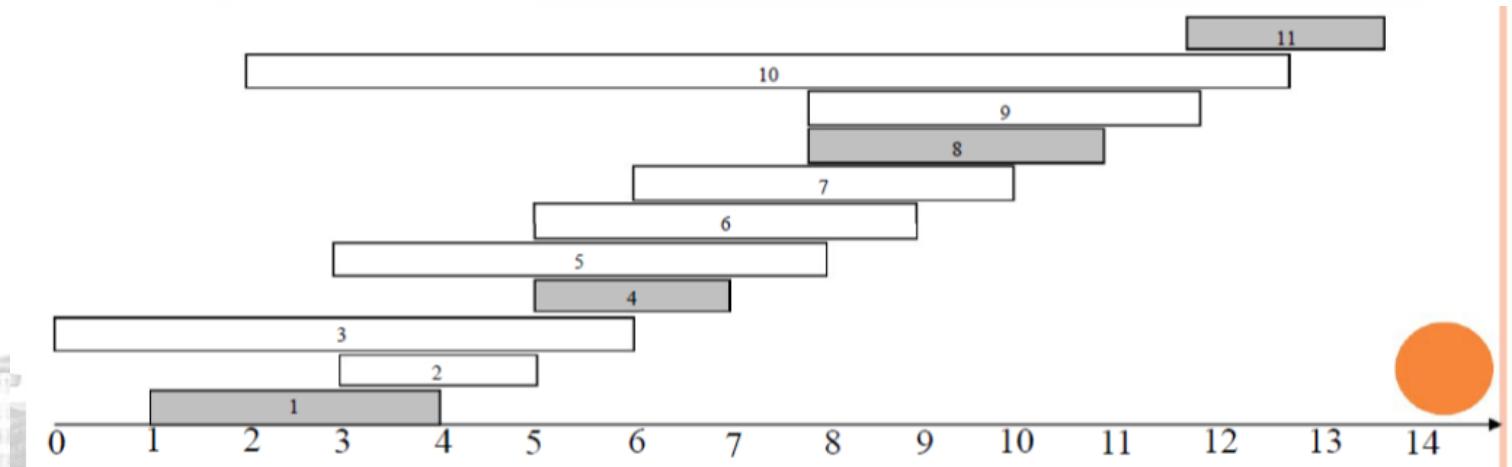
While $S_{m,n+1} \neq \emptyset$ Do

$a_t \leftarrow S_{m,n+1}$ 中结束最早的活动;

$A \leftarrow A \cup \{a_t\}$;

$m \leftarrow t$;

Return A



假设要在足够多的会场里安排一批活动，并希望使用尽可能少的会场。设计一个有效的 贪心算法进行安排。

输入格式：

第一行有 **1** 个正整数 k ，表示有 k 个待安排的活动。接下来的 k 行中，每行有 **2** 个正整数，分别表示 k 个待安排的活动开始时间
和结束时间。时间以 **0** 点开始的分钟计。

输出格式：

输出最少会场数。

把一些气球挂在一个用XY平面的平坦墙上，假设气球是2维的。气球用一个二维整数数组 `points` 表示，其中 `points[i] = [xstart, xend]` 表示一个水平直径在 `xstart` 到 `xend` 之间的气球。你不知道气球的确切y坐标， $y > 0$ 。

我们可以可以从x轴上任意一个位置 x_i 垂直向上射出一支箭（朝着正y方向），所有跟箭有交集的气球都会被射爆（ $xstart \leq x_i \leq xend$ ）。假设不限制箭的数量。

给定数组 `points`，返回爆破所有气球所需的最少箭数。

Input: `points = [[10,16],[2,8],[1,6],[7,12]]`

Output: 2

Explanation:

$x = 6$ 处射出一支箭，射爆 `[2,8]` 和 `[1,6]`。 $x = 11$ 处射出一支箭，射爆 `[10,16]` 和 `[7,12]`。

提纲

5.1 贪心法的基本原理

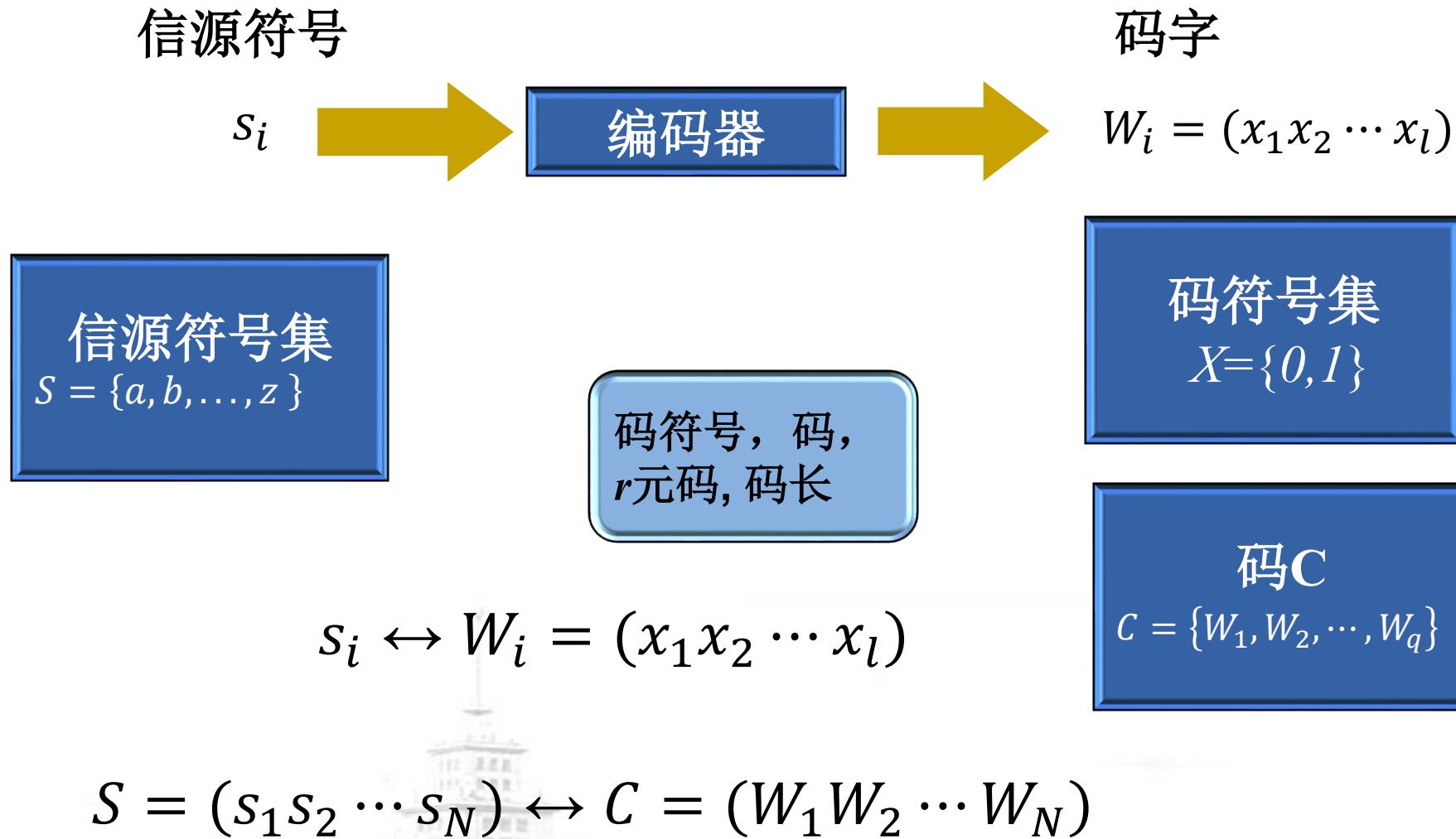
5.2 任务安排问题

5.3 哈夫曼编码问题

5.4 最小生成树



编码器



对于无失真编码，映射应为一一映射，且可逆。

5.1 编码器

例：

信源符号	码1	码2	码3	码4	码5
a	00	0	0	1	1
b	01	01	11	01	10
c	10	001	00	001	100
d	11	111	11	0001	1000

编码能正确译码的条件：

所有码字都互不相同，则称为**非奇异码(non-singular)**，一个编码要想正确译码，必须是**非奇异码**。

5.1 编码器

例：

信源符号	码1	码2	码3	码4	码5
a	00	0	0	1	1
b	01	01	11	01	10
c	10	001	00	001	100
d	11	111	11	0001	1000

编码能正确译码的条件：

所有码字都互不相同，则称为**非奇异码(non-singular)**，一个编码要想正确译码，必须是**非奇异码**。

码元序列：001001
0,01,001 — abc?
001,0,01 — cab?

*N*次扩展码

- 将待编码的符号进行*N*次扩展

$$S^N = (\alpha_j, j = 1, 2, \dots, q^N) \quad C^N = (W_j, j = 1, 2, \dots, q^N)$$

◆ 例：

信源符号	A	B	C	D
码字	0	01	001	111

二次扩展码

信源符号	AA	AB	AC	AD	BA	...	DD
码字	00	001	0001	0111	010	...	111111



*N*次扩展码

- 将待编码的符号进行*N*次扩展

$$S^N = (\alpha_j, j = 1, 2, \dots, q^N) \quad C^N = (W_j, j = 1, 2, \dots, q^N)$$

◆ 例：

信源符号	A	B	C	D
码字	0	01	001	111

二次扩展码

信源符号	AA	AB	AC	AD	BA	...	DD
码字	00	001	0001	0111	010	...	111111

对于一组编码，若对于任意有限的整数*N*，其*N*次扩展码均为非奇异的，则称之为唯一可译码。

即时性

信源符号	码1	码2	码3
a	1	1	0
b	01	10	10
c	001	100	110
d	0001	1000	111



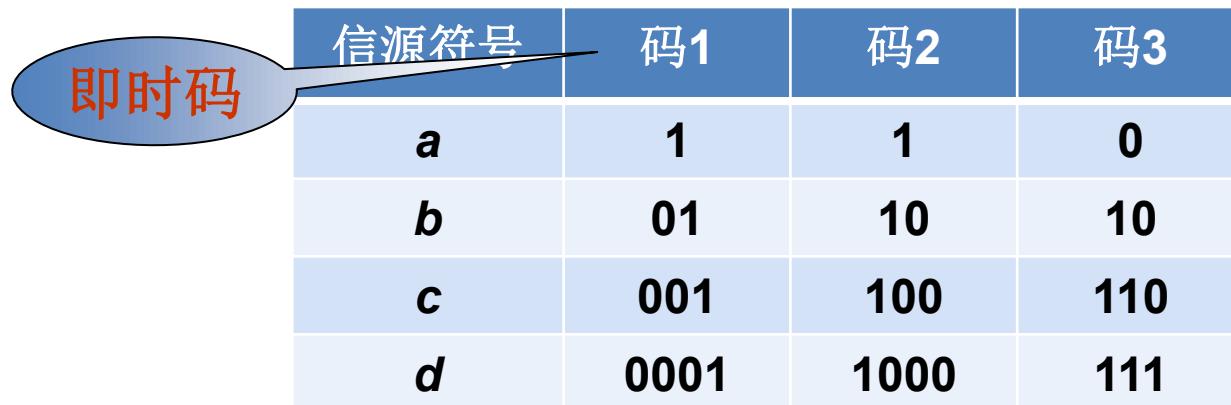
即时性

信源符号	码1	码2	码3
a	1	1	0
b	01	10	10
c	001	100	110
d	0001	1000	111

无需考虑后续的码符号即可从码符号序列中译出码字，这样的惟一可译码称为**即时码**（instantaneous code**瞬时码**、**非延长码**）。



即时性



信源符号	码1	码2	码3
a	1	1	0
b	01	10	10
c	001	100	110
d	0001	1000	111

无需考虑后续的码符号即可从码符号序列中译出码字，这样的惟一可译码称为**即时码** (*instantaneous code* 瞬时码、非延长码)。

即时性



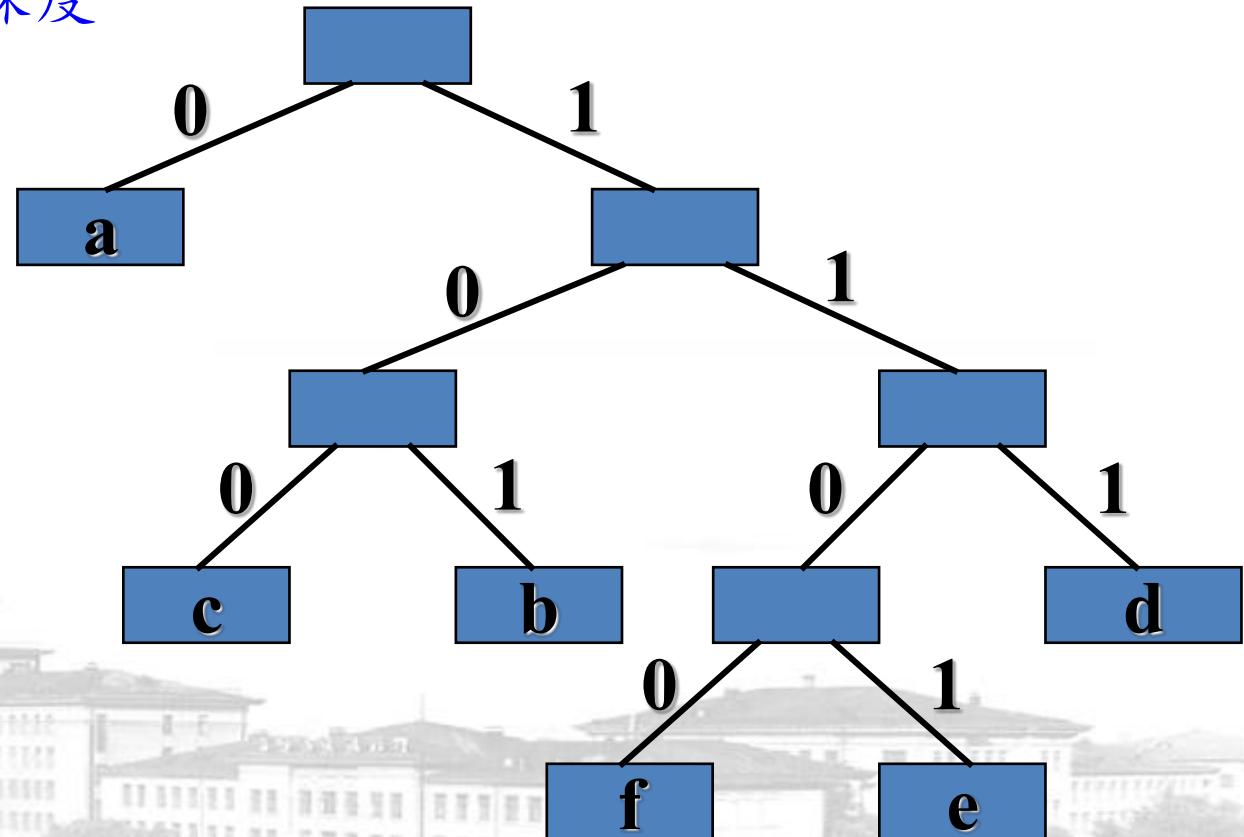
信源符号	码1	码2	码3
a	1	1	0
b	01	10	10
c	001	100	110
d	0001	1000	111

无需考虑后续的码符号即可从码符号序列中译出码字，这样的惟一可译码称为**即时码** (*instantaneous code* 瞬时码、非延长码)。

一个惟一可译码成为即时码的充要条件是其中任何一个码字都不是其他码字的前缀

计算机编码

- 前缀编码可以表示成一个二叉树
 - 每个字符是树的叶子
 - 从根结点到叶子结点的路径为对应字符的编码
 - 前缀编码树不唯一
 - 叶子节点不在同一个深度
 - 不是二叉搜索树



计算机编码

- qwertyui_opasdfg+hjklzxcv
- 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111
01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000
01101010 01101011 01101100 01111010 01111000 01100011 01110110



计算机编码

- `qwertyui_opasdfg+hjklzxcv`
- 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111
01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000
01101010 01101011 01101100 01111010 01111000 01100011 01110110
- `everyday english sentence`
- 01100101 01110110 01100101 01110010 01111001 01100100 01100001 01111001 00100000
01100101 01101110 01100111 01101100 01101001 01110011 01101000 00100000 01110011
01100101 01101110 01110100 01100101 01101110 01100011 01100101

计算机编码

- everyday english sentence
 - 01100101 01110110 01100101 01110010 01111001 01100100 01100001 01111001 00100000
01100101 01101110 01100111 01101100 01101001 01110011 01101000 00100000 01110011
01100101 01101110 01110100 01100101 01101110 01100011 01100101
- qwertyui_opasdfg+hjklzxcv
 - 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111
01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000
01101010 01101011 01101100 01111010 01111000 01100011 01110110

计算机编码

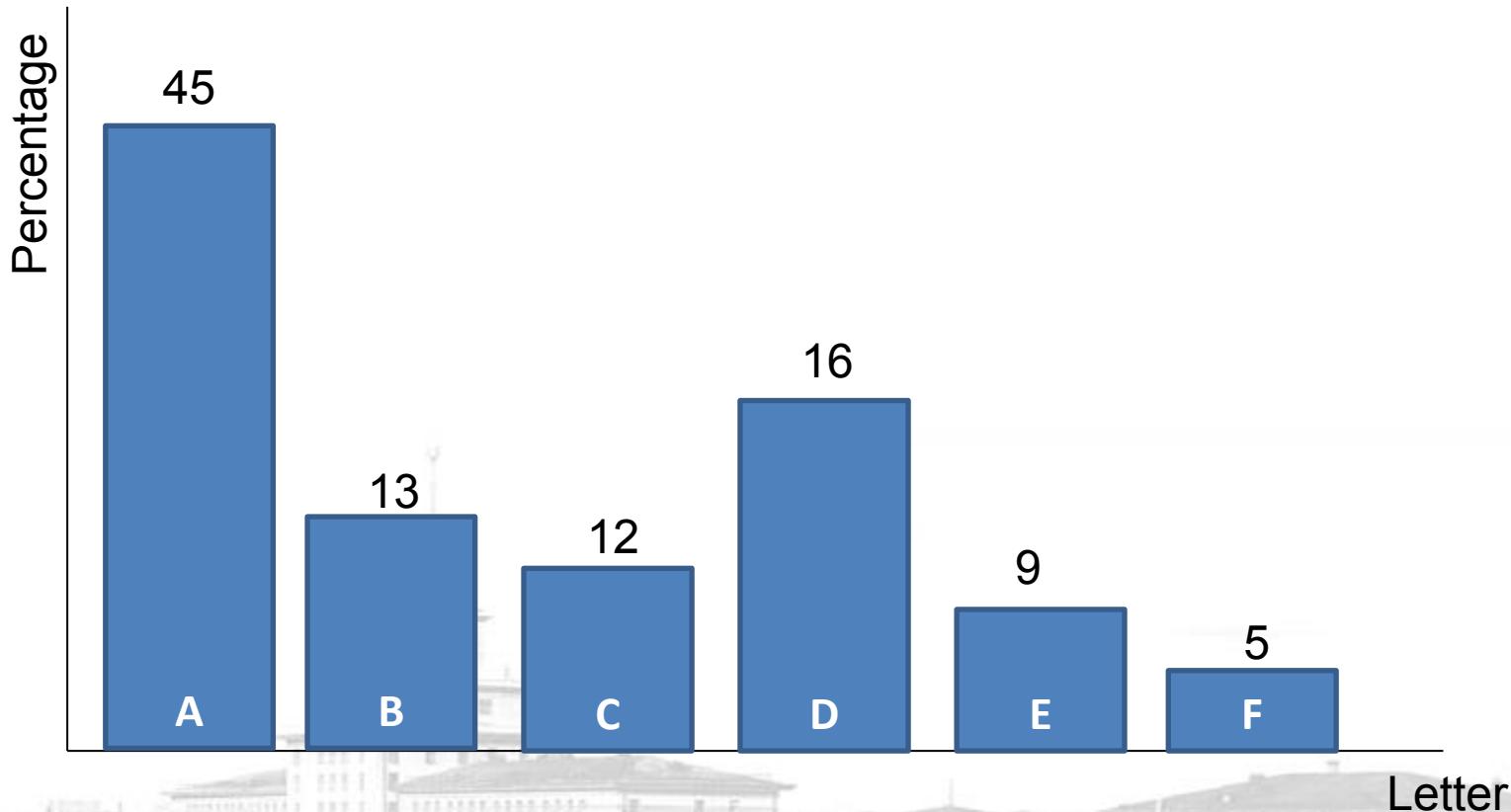
如果 e 出现多次的话，
ASCII码将会非常低效！

- everyday english sentence
- 01100101 01110110 01100101 01110010 01111001 01100100 01100001 01111001 00100000
01100101 01101110 01100111 01101100 01101001 01110011 01101000 00100000 01110011
01100101 01101110 01110100 01100101 01101110 01100011 01100101
- qwertyui_opasdfg+hjklzxcv
- 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111
01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000
01101010 01101011 01101100 01111010 01111000 01100011 01110110



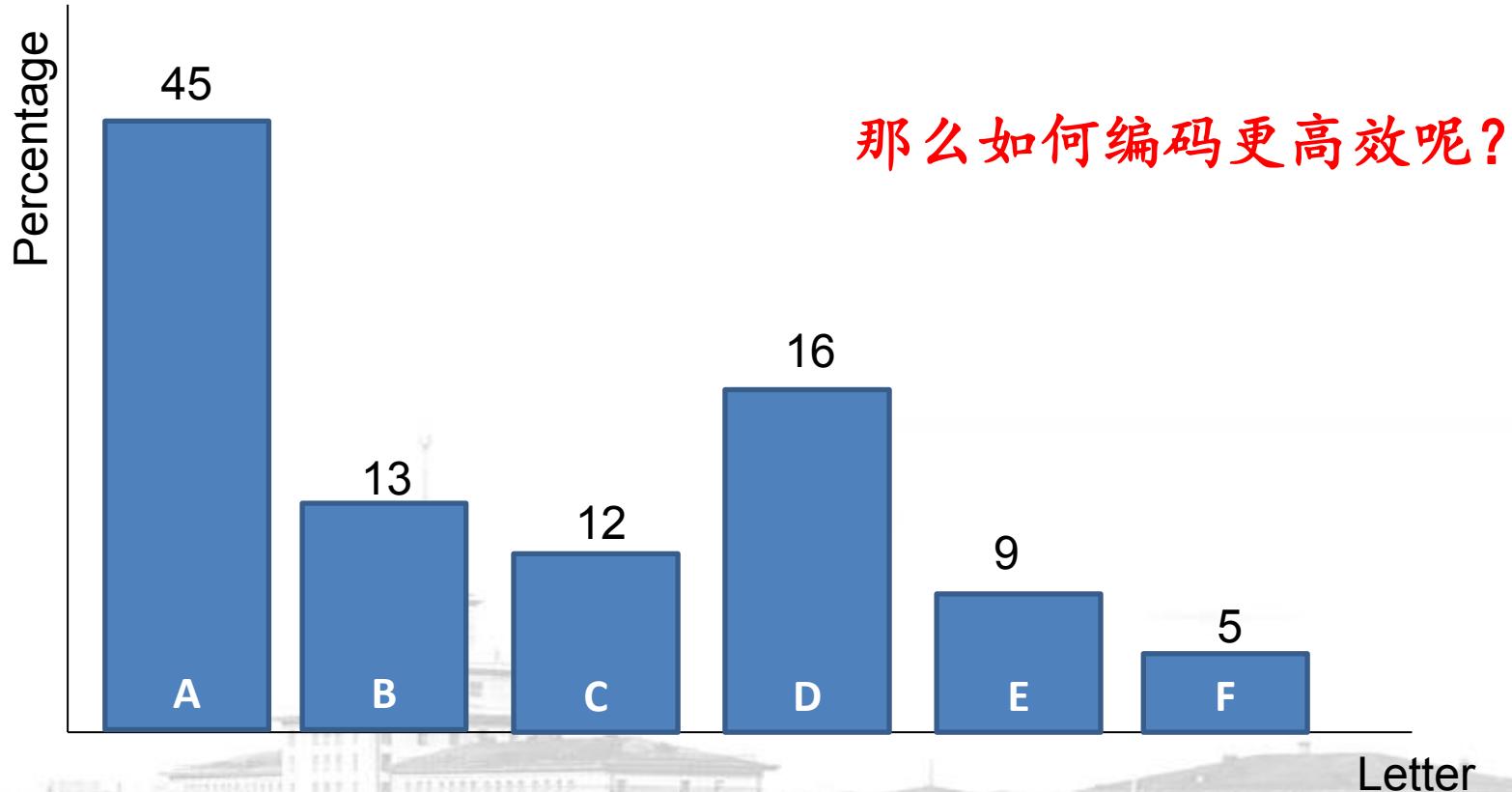
计算机编码

- 字符在使用中有一个使用频率的分布，例如：



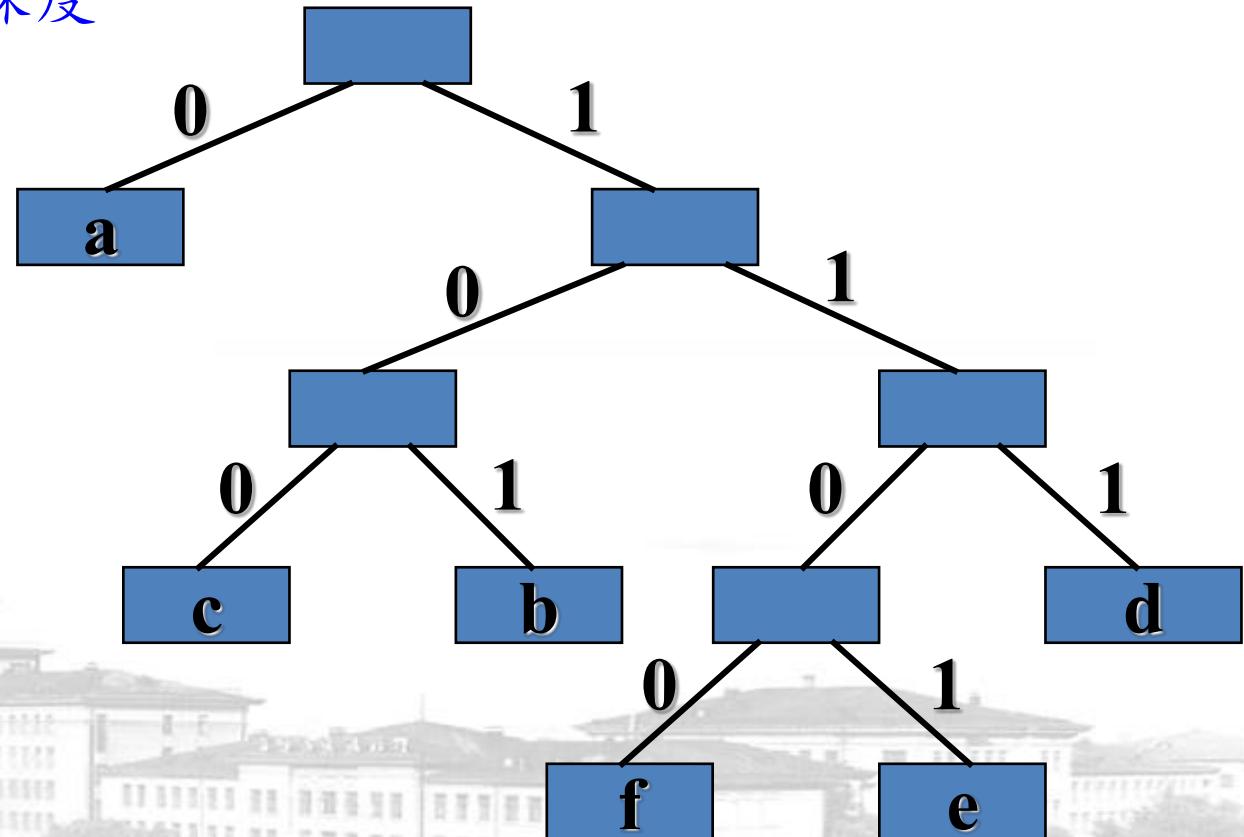
计算机编码

- 字符在使用中有一个使用频率的分布，例如：



计算机编码

- 前缀编码可以表示成一个二叉树
 - 每个字符是树的叶子
 - 从根结点到叶子结点的路径为对应字符的编码
 - 前缀编码树不唯一
 - 叶子节点不在同一个深度
 - 不是二叉搜索树



编码树T的代价

- 设 C 是字母表, $\forall c \in C$
- $f(c)$ 是 c 在文件中出现的频率
- $d_T(c)$ 是叶子 c 在树 T 中的深度, 即 c 的编码长度
- T 的代价是编码所有字符的期望编码长度:

$$B(T) = \sum_{c \in C} f(c)d_T(c)$$

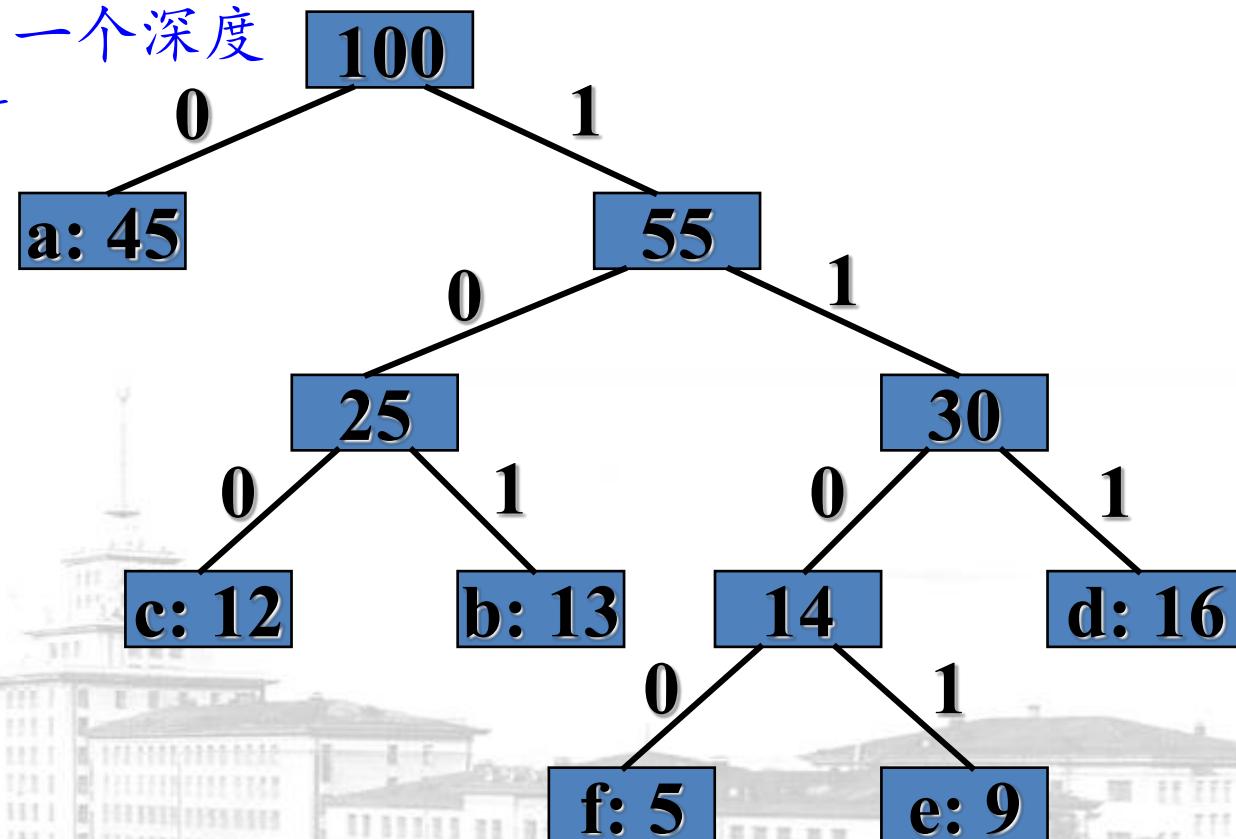
计算机编码

- 前缀编码可以表示成一个二叉树
 - 每个字符是树的叶子
 - 从根结点到叶子结点的路径为对应字符的编码
 - 前缀编码树不唯一
 - 叶子节点不在同一个深度
 - 不是二叉搜索树



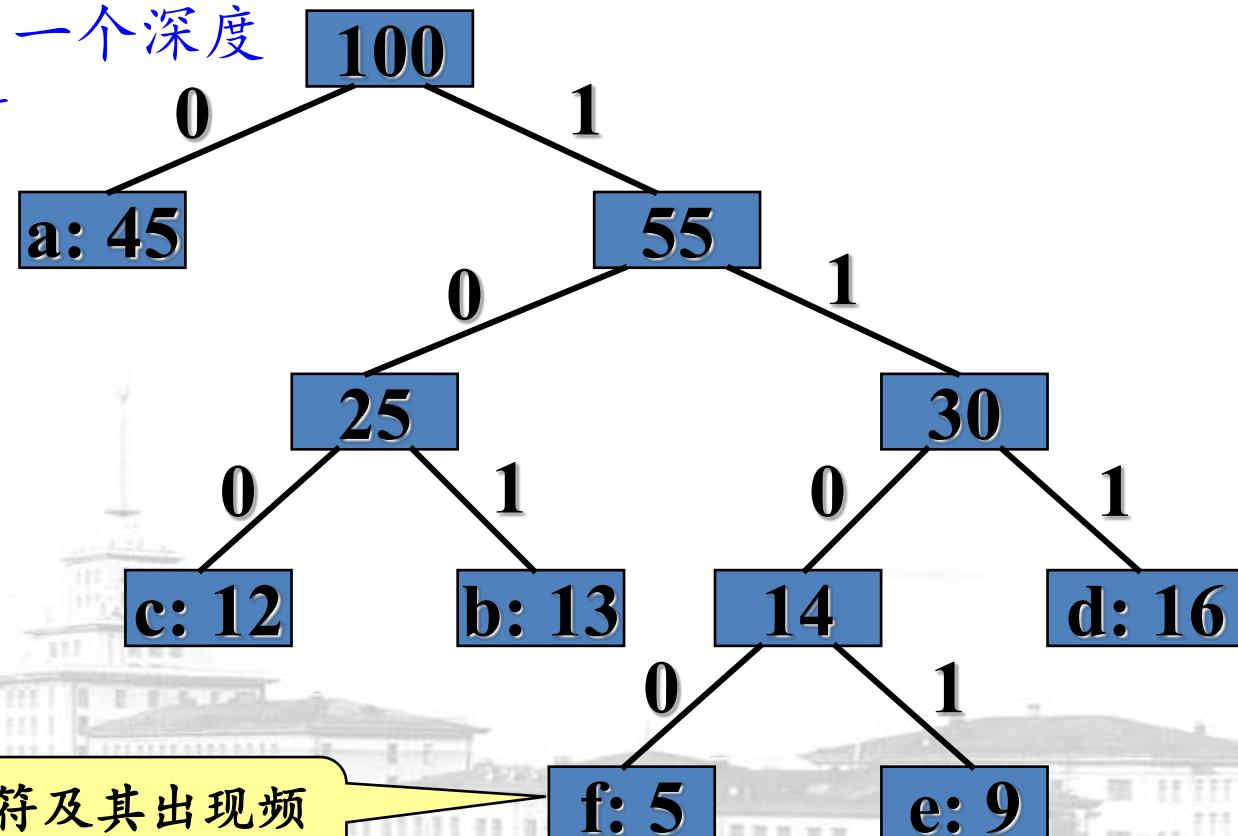
计算机编码

- 前缀编码可以表示成一个二叉树
 - 每个字符是树的叶子
 - 从根结点到叶子结点的路径为对应字符的编码
 - 前缀编码树不唯一
 - 叶子节点不在同一个深度
 - 不是二叉搜索树



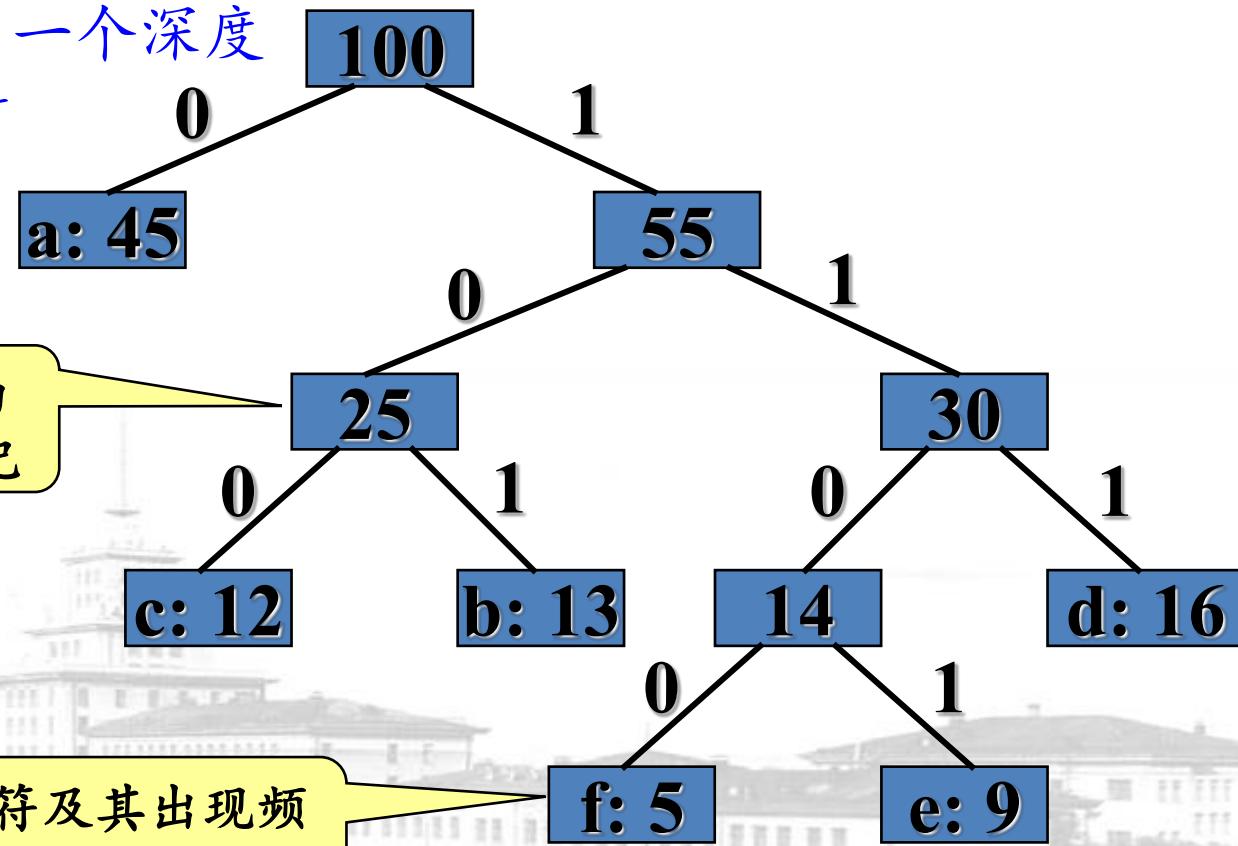
计算机编码

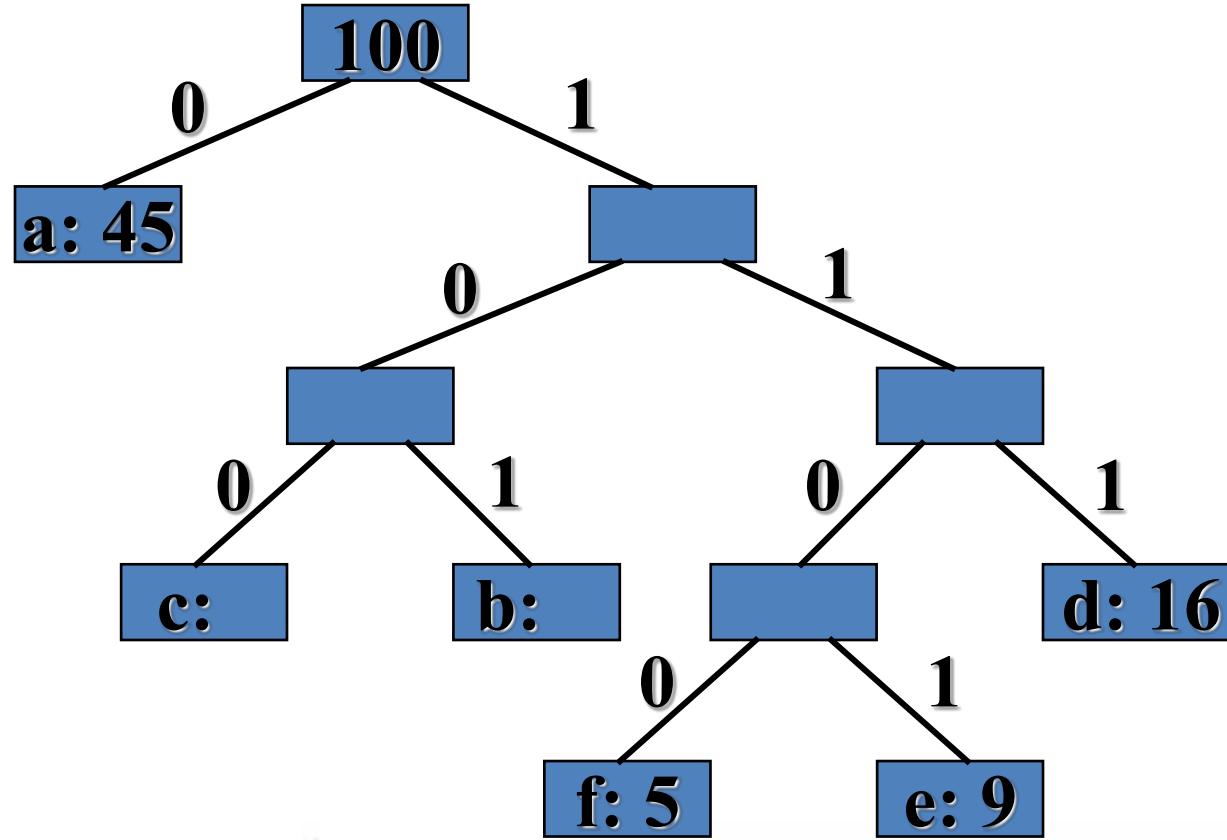
- 前缀编码可以表示成一个二叉树
 - 每个字符是树的叶子
 - 从根结点到叶子结点的路径为对应字符的编码
 - 前缀编码树不唯一
 - 叶子节点不在同一个深度
 - 不是二叉搜索树



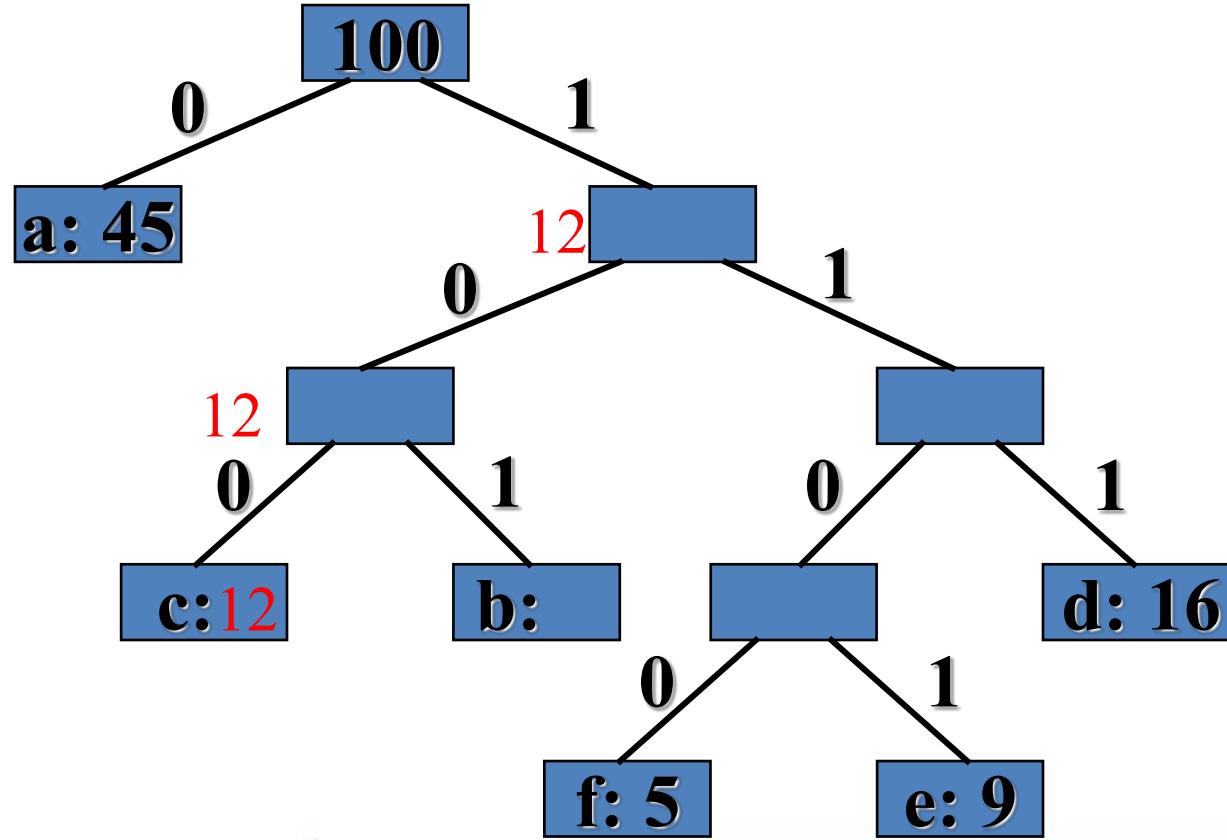
计算机编码

- 前缀编码可以表示成一个二叉树
 - 每个字符是树的叶子
 - 从根结点到叶子结点的路径为对应字符的编码
 - 前缀编码树不唯一
 - 叶子节点不在同一个深度
 - 不是二叉搜索树



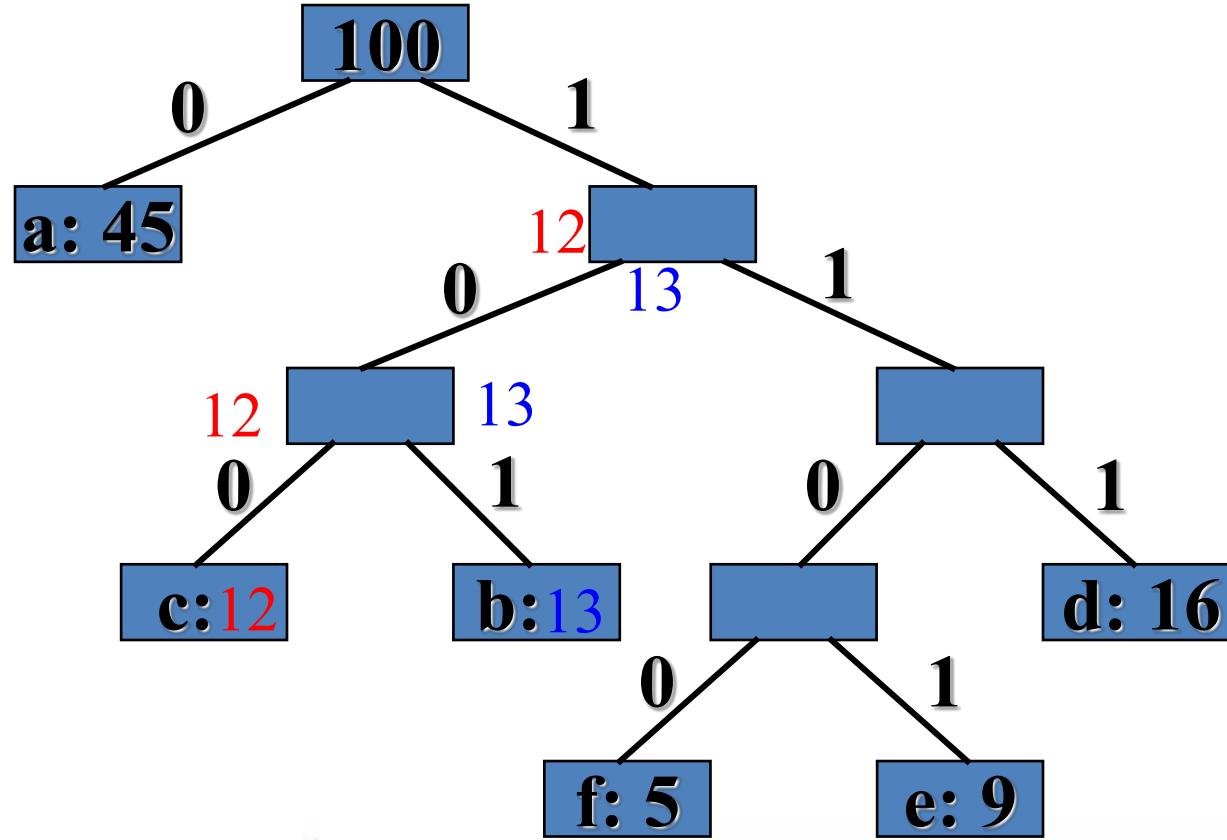


$$B(T) = \sum_{c \in C} f(c)d_T(c) = (\sum_{k \in T} f(k)) - f(\text{root})$$



$$f(c)d(c) = 3 * 12 = 12 + 12 + 12$$

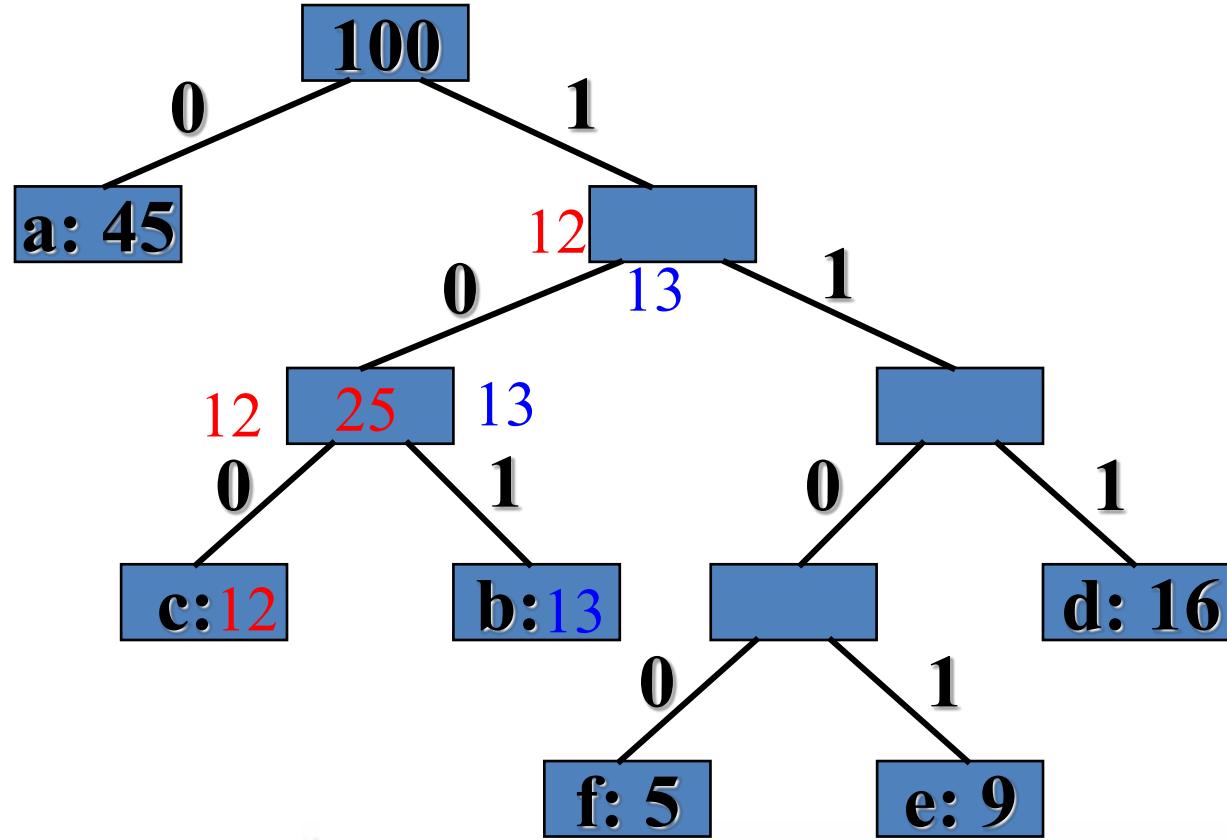
$$B(T) = \sum_{c \in C} f(c)d_T(c) = (\sum_{k \in T} f(k)) - f(root)$$



$$f(c)d(c) = 3 * 12 = 12 + 12 + 12$$

$$f(b)d(b) = 3 * 13 = 13 + 13 + 13$$

$$B(T) = \sum_{c \in C} f(c)d_T(c) = (\sum_{k \in T} f(k)) - f(root)$$



$$f(c)d(c) = 3 * 12 = 12 + 12 + 12$$

$$f(b)d(b) = 3 * 13 = 13 + 13 + 13$$

$$B(T) = \sum_{c \in C} f(c)d_T(c) = (\sum_{k \in T} f(k)) - f(root)$$

问题的定义

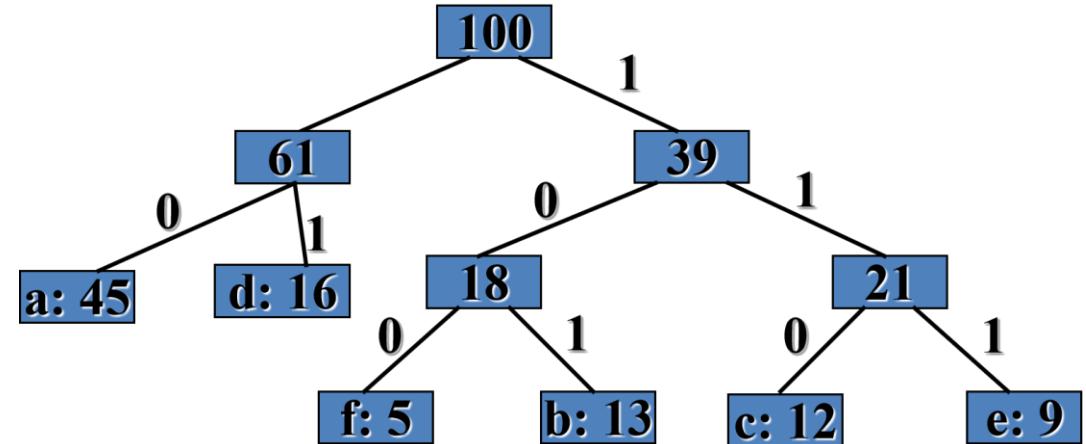
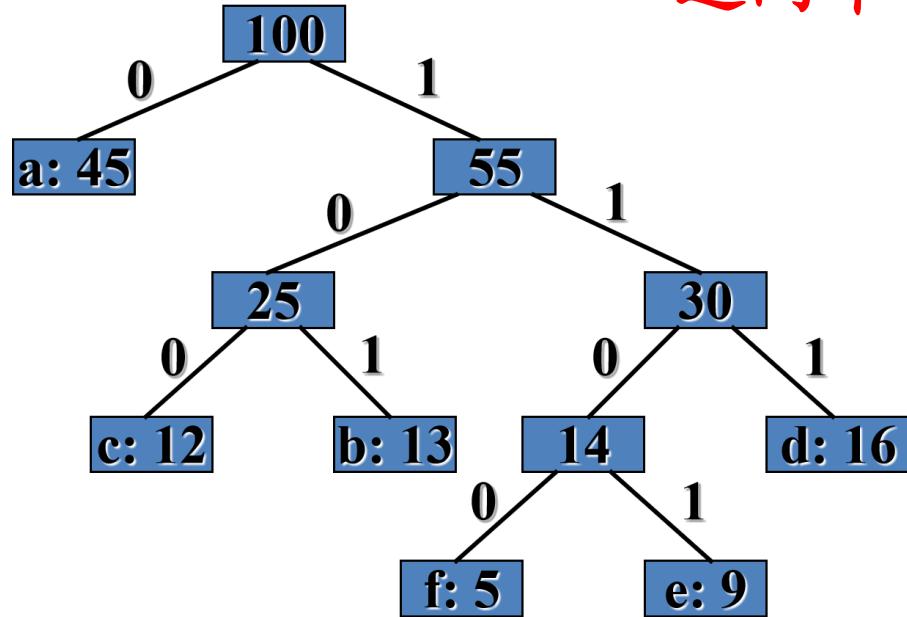
输入：字母表 $C = \{c_1, c_2, \dots, c_n\}$,

对应的频率表 $F = \{f(c_1), f(c_2), \dots, f(c_n)\}$

输出：具有最小 $B(T)$ 的字母表 C 的前缀编码树，即 C
的哈夫曼编码树

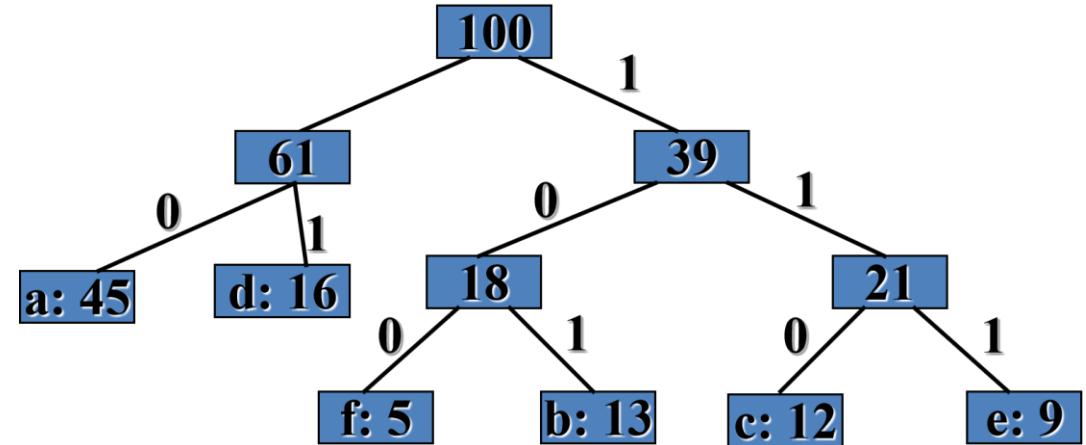
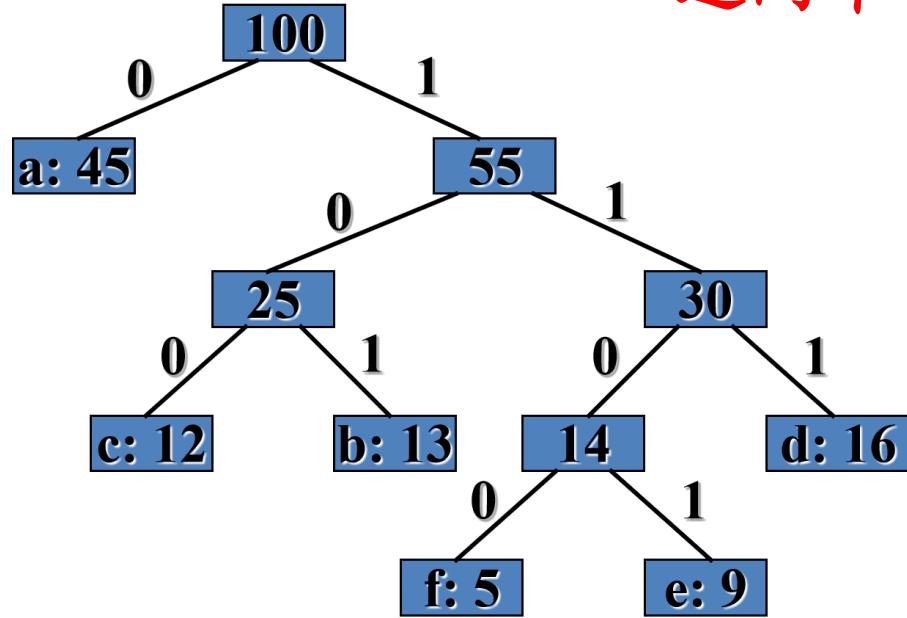
计算机编码

这两个前缀编码树，哪个更好？



计算机编码

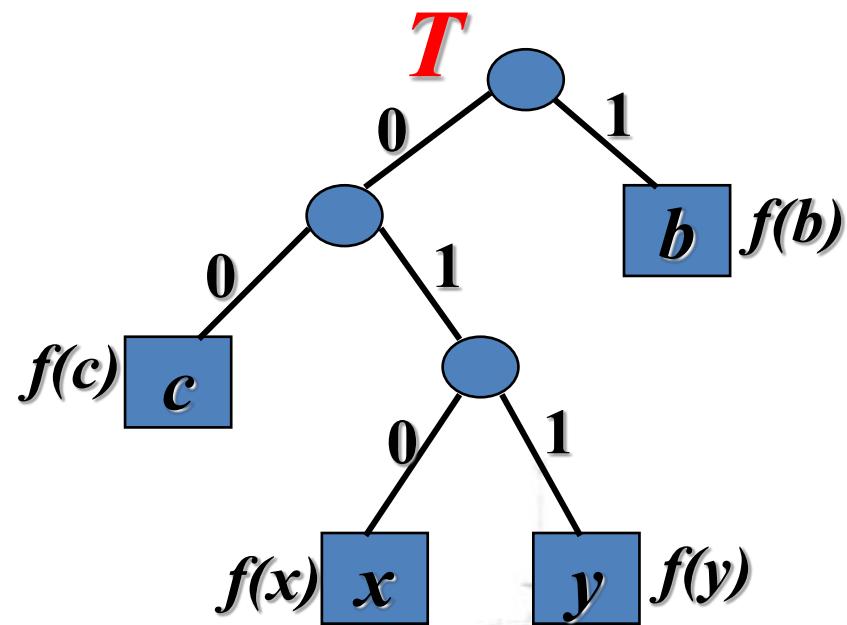
这两个前缀编码树，哪个更好？



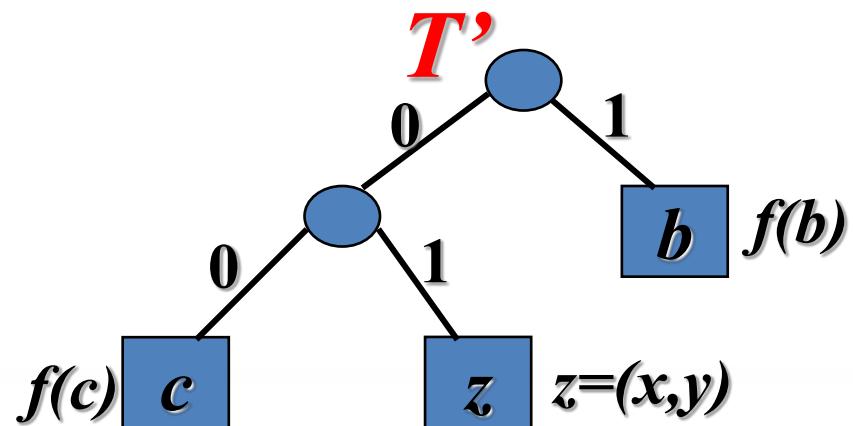
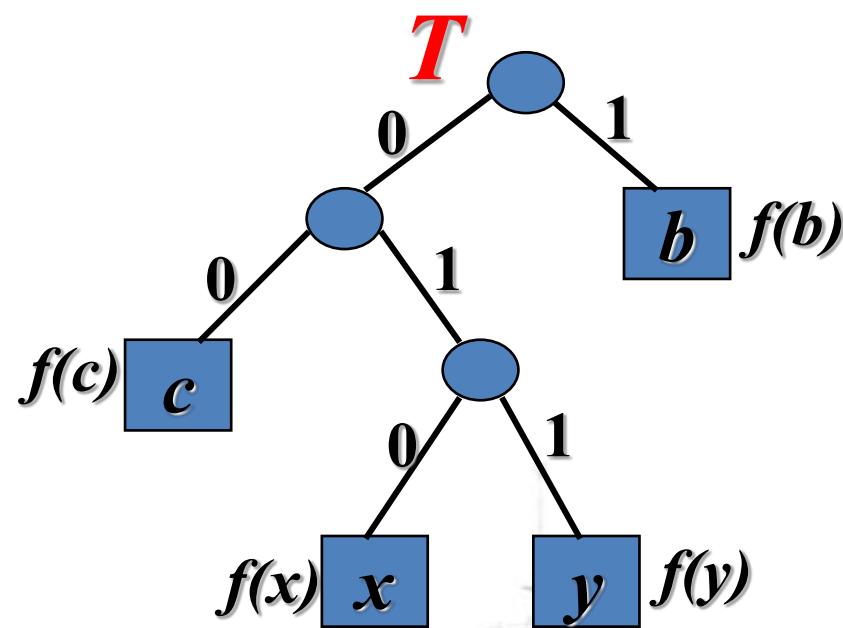
贪心思想：

循环地选择具有最低频率的两个结点，
生成一棵子树，直至形成树

子问题拆分



子问题拆分



优化解的结构分析

- 我们需要证明
 - 优化前缀树问题具有优化子结构
 - 优化前缀树问题具有贪心选择性



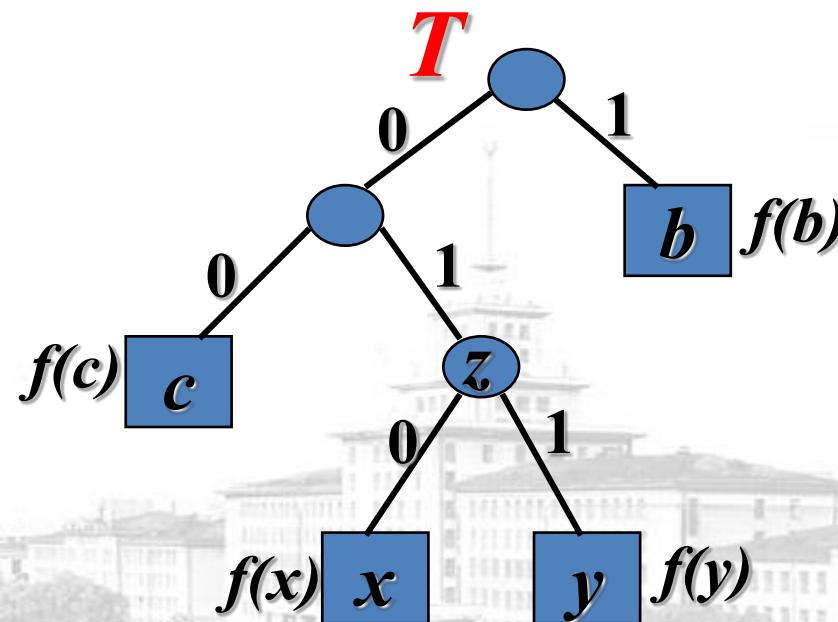
• 优化子结构

引理1.设 T 是字母表 C 的优化前缀树， $\forall c \in C, f(c)$ 是 c 在文件中出现的频率. 设 x, y 是 T 中任意两个相邻叶结点， z 是它们的父结点，则 z 作为频率是 $f(z) = f(x) + f(y)$ 的字符， $T' = T - \{x, y\}$ 是字母表 $C' = C - \{x, y\} \cup \{z\}$ 的优化前缀编码树.



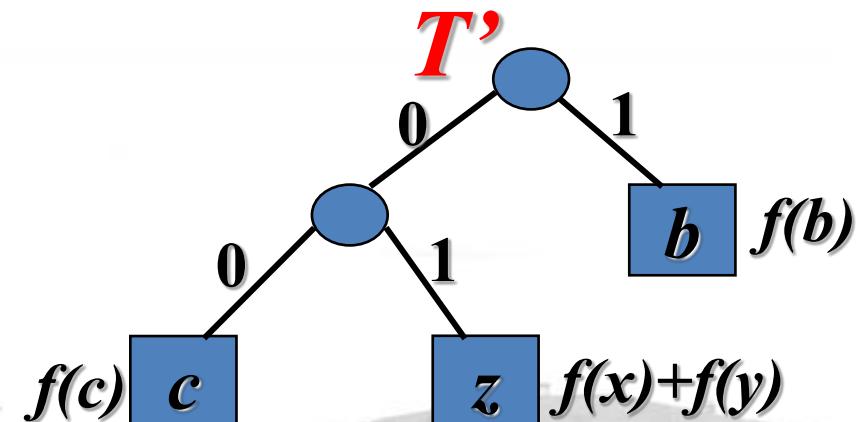
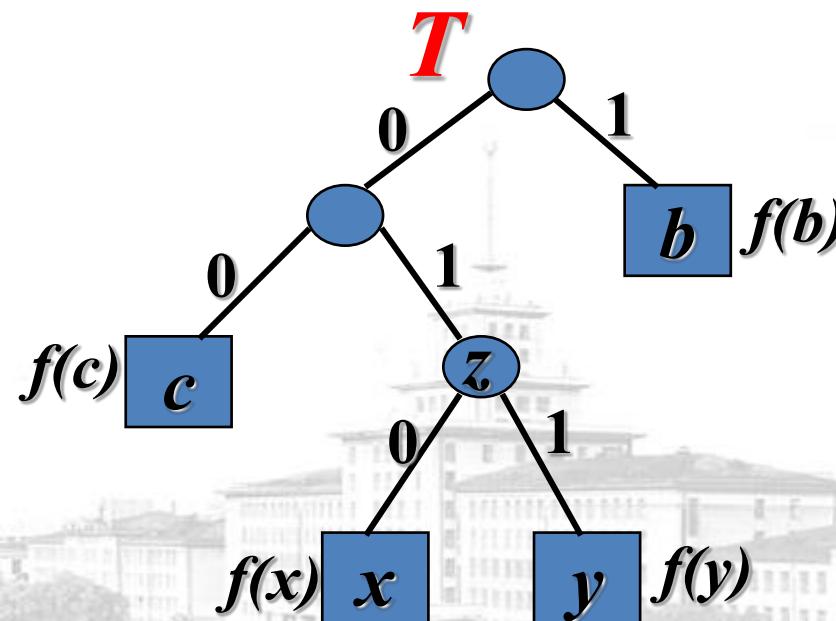
• 优化子结构

引理1.设 T 是字母表 C 的优化前缀树， $\forall c \in C, f(c)$ 是 c 在文件中出现的频率。设 x, y 是 T 中任意两个相邻叶结点， z 是它们的父结点，则 z 作为频率是 $f(z) = f(x) + f(y)$ 的字符， $T' = T - \{x, y\}$ 是字母表 $C' = C - \{x, y\} \cup \{z\}$ 的优化前缀编码树。



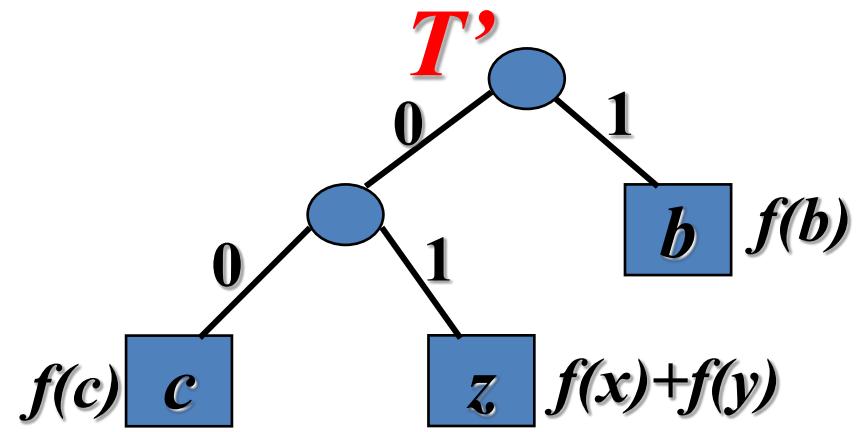
• 优化子结构

引理1.设 T 是字母表 C 的优化前缀树， $\forall c \in C, f(c)$ 是 c 在文件中出现的频率。设 x, y 是 T 中任意两个相邻叶结点， z 是它们的父结点，则 z 作为频率是 $f(z) = f(x) + f(y)$ 的字符， $T' = T - \{x, y\}$ 是字母表 $C' = C - \{x, y\} \cup \{z\}$ 的优化前缀编码树。



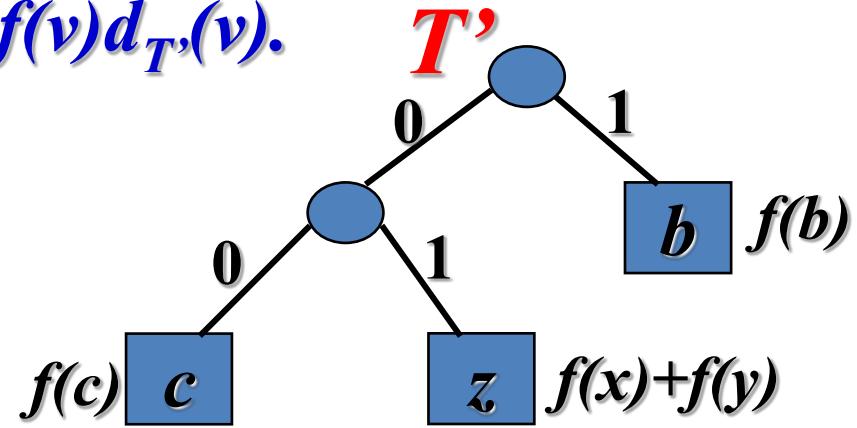
证. 证 $B(T)=B(T')+f(x)+f(y)$.

证. 证 $B(T)=B(T')+f(x)+f(y)$.



证. 证 $B(T) = B(T') + f(x) + f(y)$.

$\forall v \in C - \{x, y\}, d_T(v) = d_{T'}(v), f(v)d_T(v) = f(v)d_{T'}(v)$.

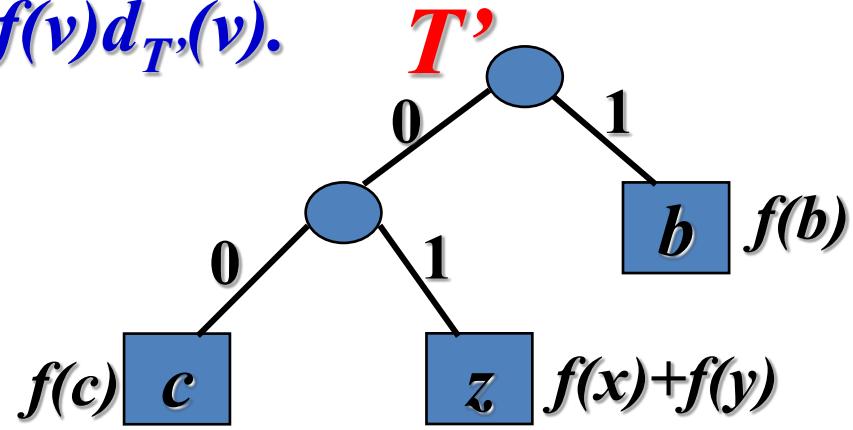


证. 证 $B(T)=B(T')+f(x)+f(y)$.

$\forall v \in C - \{x, y\}, d_T(v) = d_{T'}(v), f(v)d_T(v) = f(v)d_{T'}(v)$.

由于 $d_T(x) = d_T(y) = d_{T'}(z) + 1$,

$$\begin{aligned} & f(x)d_T(x) + f(y)d_T(y) \\ &= (f(x) + f(y))(d_{T'}(z) + 1) \\ &= (f(x) + f(y))d_{T'}(z) + (f(x) + f(y)) \end{aligned}$$



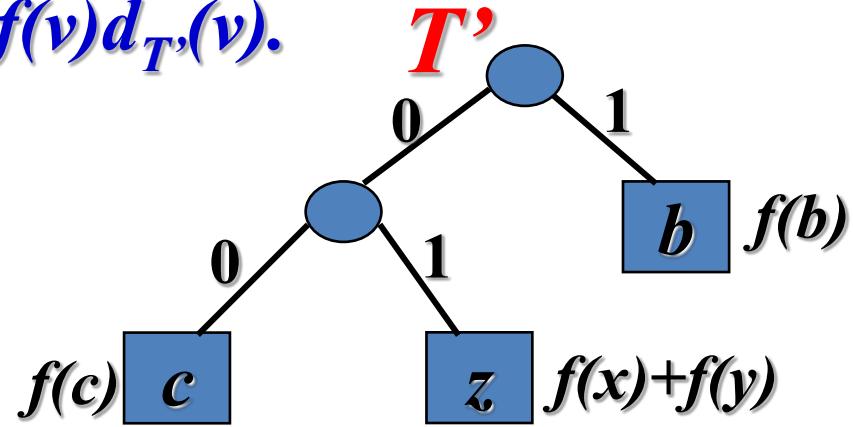
证. 证 $B(T)=B(T')+f(x)+f(y)$.

$\forall v \in C-\{x,y\}, d_T(v)=d_{T'}(v), f(v)d_T(v)=f(v)d_{T'}(v)$.

由于 $d_T(x)=d_T(y)=d_{T'}(z)+1$,

$$\begin{aligned} & f(x)d_T(x)+f(y)d_T(y) \\ &= (f(x)+f(y))(d_{T'}(z)+1) \\ &= (f(x)+f(y))d_{T'}(z)+(f(x)+f(y)) \end{aligned}$$

由于 $f(x)+f(y)=f(z), f(x)d_T(x)+f(y)d_T(y)=f(z)d_{T'}(z)+(f(x)+f(y))$.



证. 证 $B(T)=B(T')+f(x)+f(y)$.

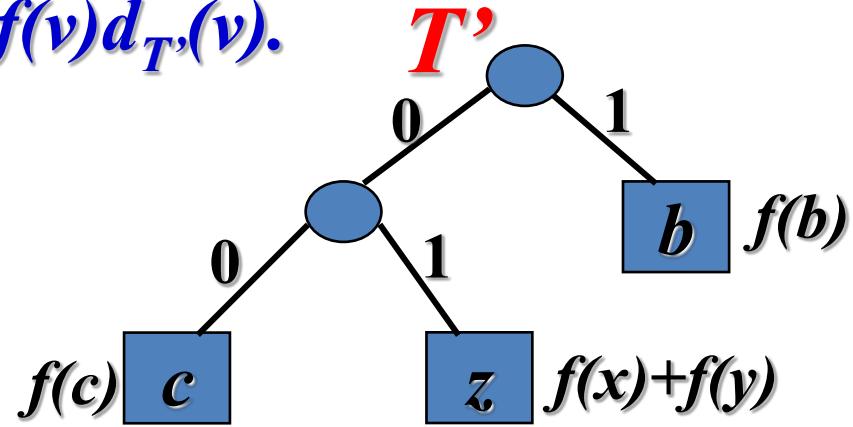
$\forall v \in C-\{x,y\}, d_T(v)=d_{T'}(v), f(v)d_T(v)=f(v)d_{T'}(v)$.

由于 $d_T(x)=d_T(y)=d_{T'}(z)+1$,

$$\begin{aligned} & f(x)d_T(x)+f(y)d_T(y) \\ &= (f(x)+f(y))(d_{T'}(z)+1) \\ &= (f(x)+f(y))d_{T'}(z)+(f(x)+f(y)) \end{aligned}$$

由于 $f(x)+f(y)=f(z), f(x)d_T(x)+f(y)d_T(y)=f(z)d_{T'}(z)+(f(x)+f(y))$.

于是 $B(T)=B(T')+f(x)+f(y)$.



证. 证 $B(T)=B(T')+f(x)+f(y)$.

$\forall v \in C-\{x,y\}, d_T(v)=d_{T'}(v), f(v)d_T(v)=f(v)d_{T'}(v)$.

由于 $d_T(x)=d_T(y)=d_{T'}(z)+1$,

$$f(x)d_T(x)+f(y)d_T(y)$$

$$=(f(x)+f(y))(d_{T'}(z)+1)$$

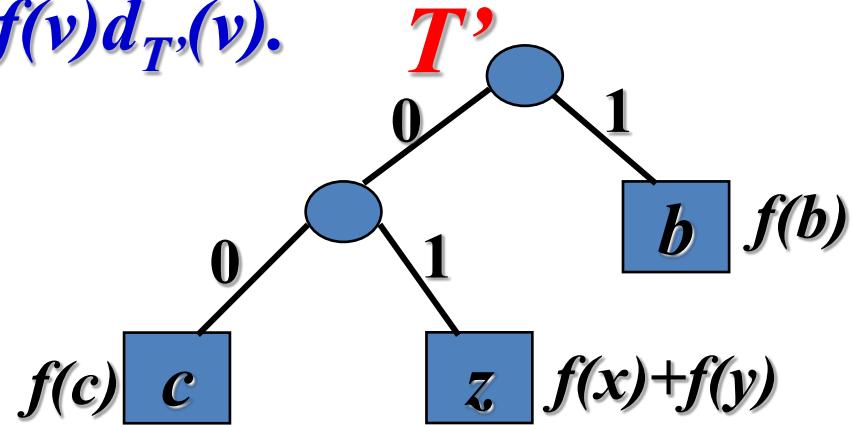
$$=(f(x)+f(y))d_{T'}(z)+(f(x)+f(y))$$

由于 $f(x)+f(y)=f(z), f(x)d_T(x)+f(y)d_T(y)=f(z)d_{T'}(z)+(f(x)+f(y))$.

于是 $B(T)=B(T')+f(x)+f(y)$.

若 T' 不是 C' 的优化前缀编码树,

则必存在 T'' , 使 $B(T'') < B(T')$.



证. 证 $B(T)=B(T')+f(x)+f(y)$.

$\forall v \in C-\{x,y\}, d_T(v)=d_{T'}(v), f(v)d_T(v)=f(v)d_{T'}(v)$.

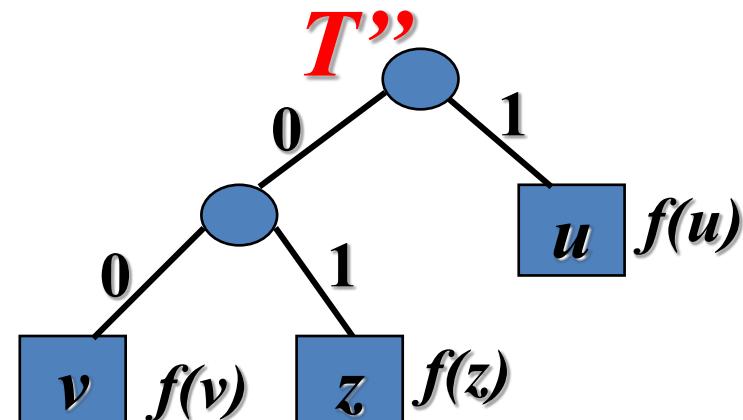
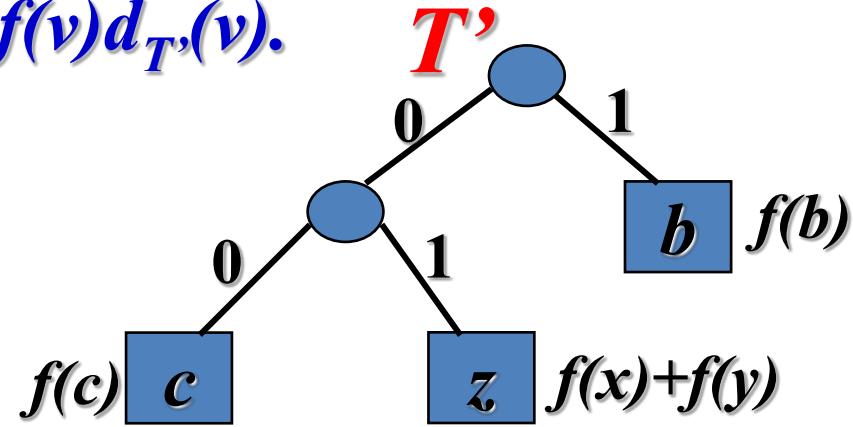
由于 $d_T(x)=d_T(y)=d_{T'}(z)+1$,

$$\begin{aligned} & f(x)d_T(x)+f(y)d_T(y) \\ &= (f(x)+f(y))(d_{T'}(z)+1) \\ &= (f(x)+f(y))d_{T'}(z)+(f(x)+f(y)) \end{aligned}$$

由于 $f(x)+f(y)=f(z), f(x)d_T(x)+f(y)d_T(y)=f(z)d_{T'}(z)+(f(x)+f(y))$.

于是 $B(T)=B(T')+f(x)+f(y)$.

若 T' 不是 C' 的优化前缀编码树,
则必存在 T'' , 使 $B(T'') < B(T')$.



证. 证 $B(T)=B(T')+f(x)+f(y)$.

$\forall v \in C-\{x,y\}, d_T(v)=d_{T'}(v), f(v)d_T(v)=f(v)d_{T'}(v)$.

由于 $d_T(x)=d_T(y)=d_{T'}(z)+1$,

$$\begin{aligned} & f(x)d_T(x)+f(y)d_T(y) \\ &= (f(x)+f(y))(d_{T'}(z)+1) \\ &= (f(x)+f(y))d_{T'}(z)+(f(x)+f(y)) \end{aligned}$$

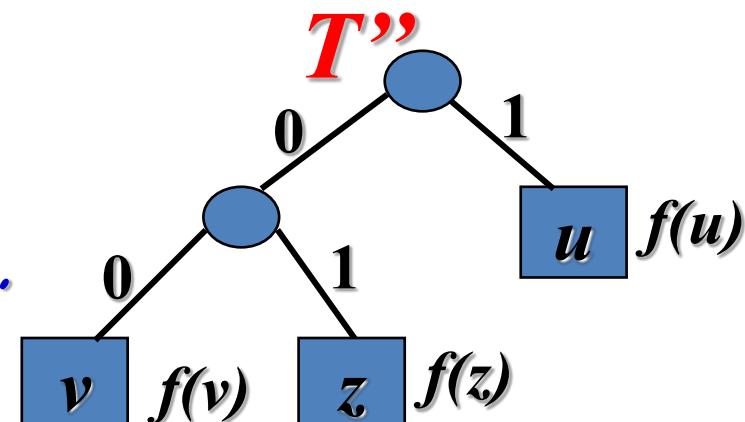
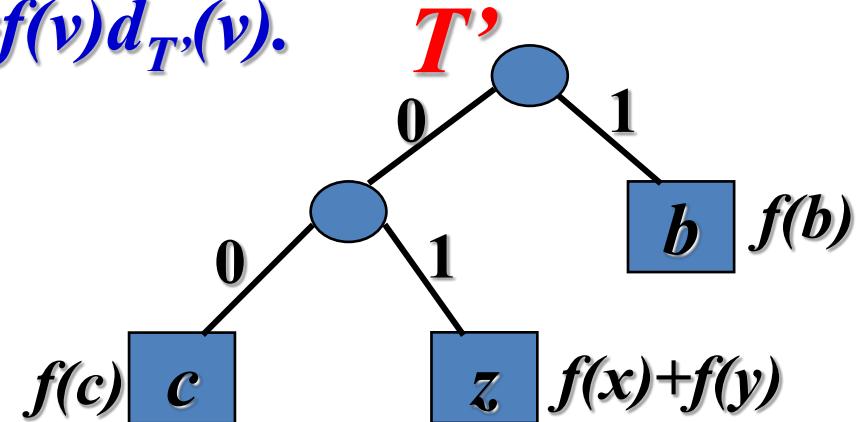
由于 $f(x)+f(y)=f(z), f(x)d_T(x)+f(y)d_T(y)=f(z)d_{T'}(z)+(f(x)+f(y))$.

于是 $B(T)=B(T')+f(x)+f(y)$.

若 T' 不是 C' 的优化前缀编码树,

则必存在 T'' , 使 $B(T'') < B(T')$.

因为 z 是 C' 中字符, 它必为 T'' 中的叶子.



证. 证 $B(T)=B(T')+f(x)+f(y)$.

$\forall v \in C-\{x,y\}, d_T(v)=d_{T'}(v), f(v)d_T(v)=f(v)d_{T'}(v)$.

由于 $d_T(x)=d_T(y)=d_{T'}(z)+1$,

$$\begin{aligned} & f(x)d_T(x)+f(y)d_T(y) \\ &= (f(x)+f(y))(d_{T'}(z)+1) \\ &= (f(x)+f(y))d_{T'}(z)+(f(x)+f(y)) \end{aligned}$$

由于 $f(x)+f(y)=f(z), f(x)d_T(x)+f(y)d_T(y)=f(z)d_{T'}(z)+(f(x)+f(y))$.

于是 $B(T)=B(T')+f(x)+f(y)$.

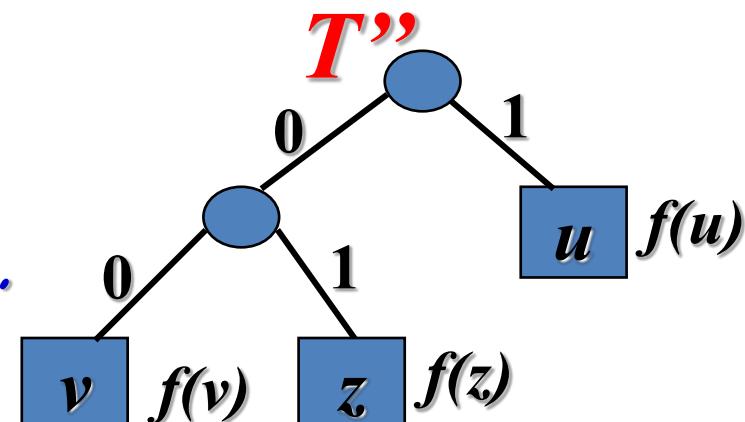
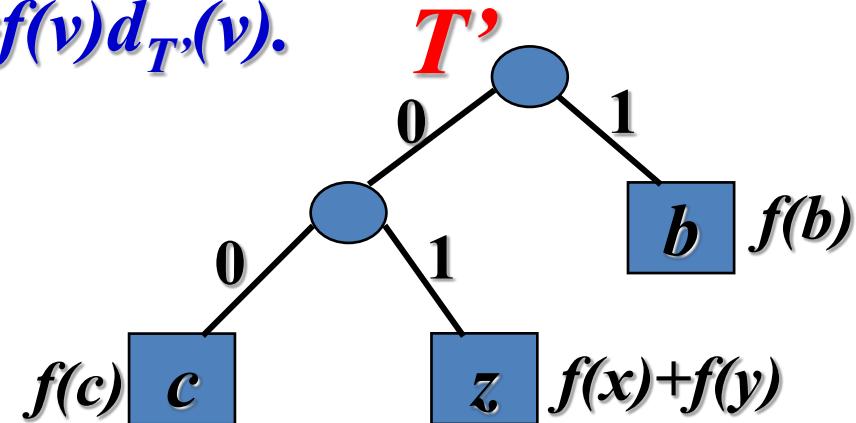
若 T' 不是 C' 的优化前缀编码树,

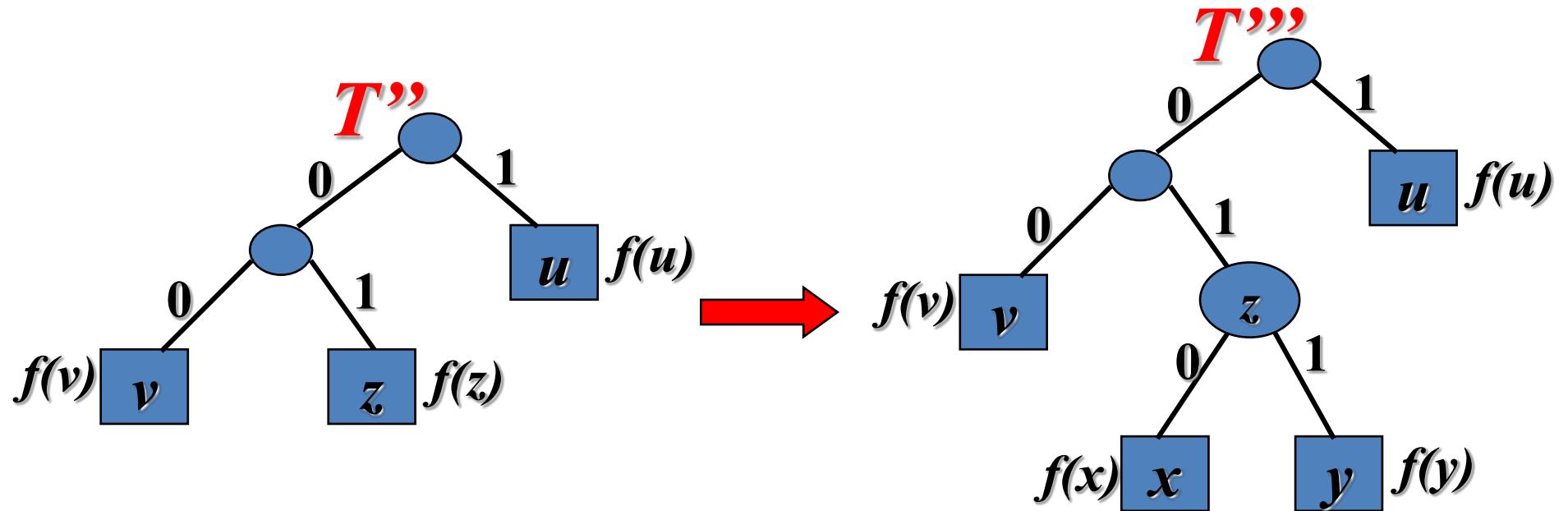
则必存在 T'' , 使 $B(T'') < B(T')$.

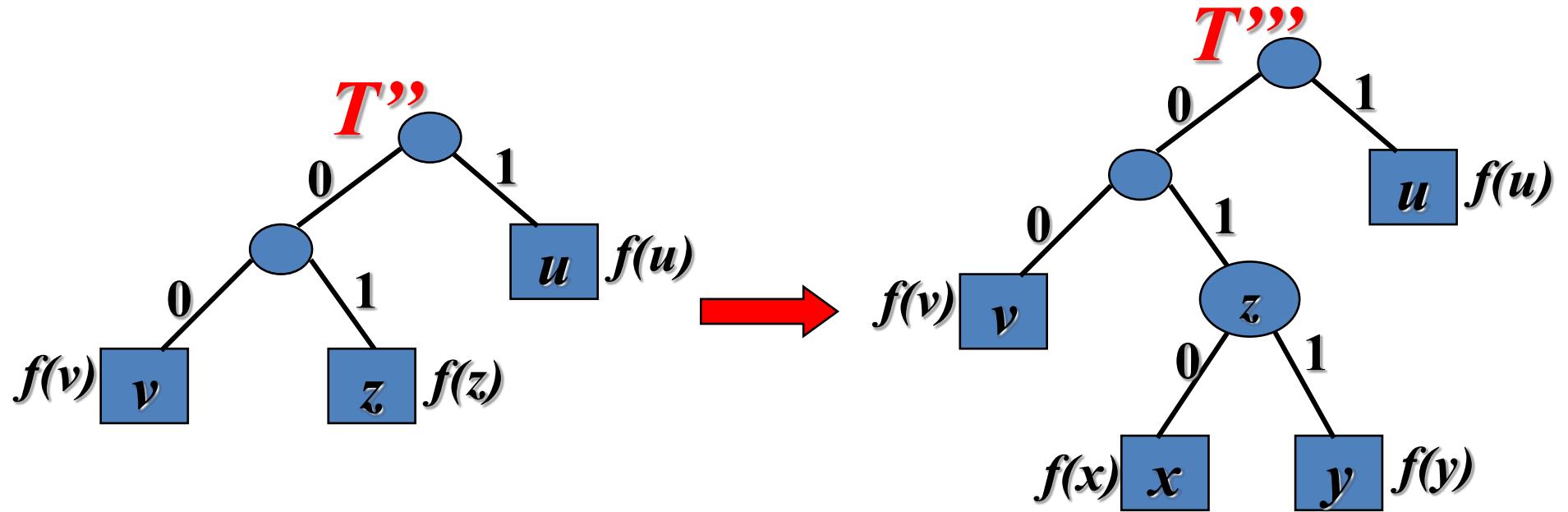
因为 z 是 C' 中字符, 它必为 T'' 中的叶子.

把结点 x 与 y 加入 T'' , 作为 z 的子结点,

则得到 C 的一个如下前缀编码树 T''' :

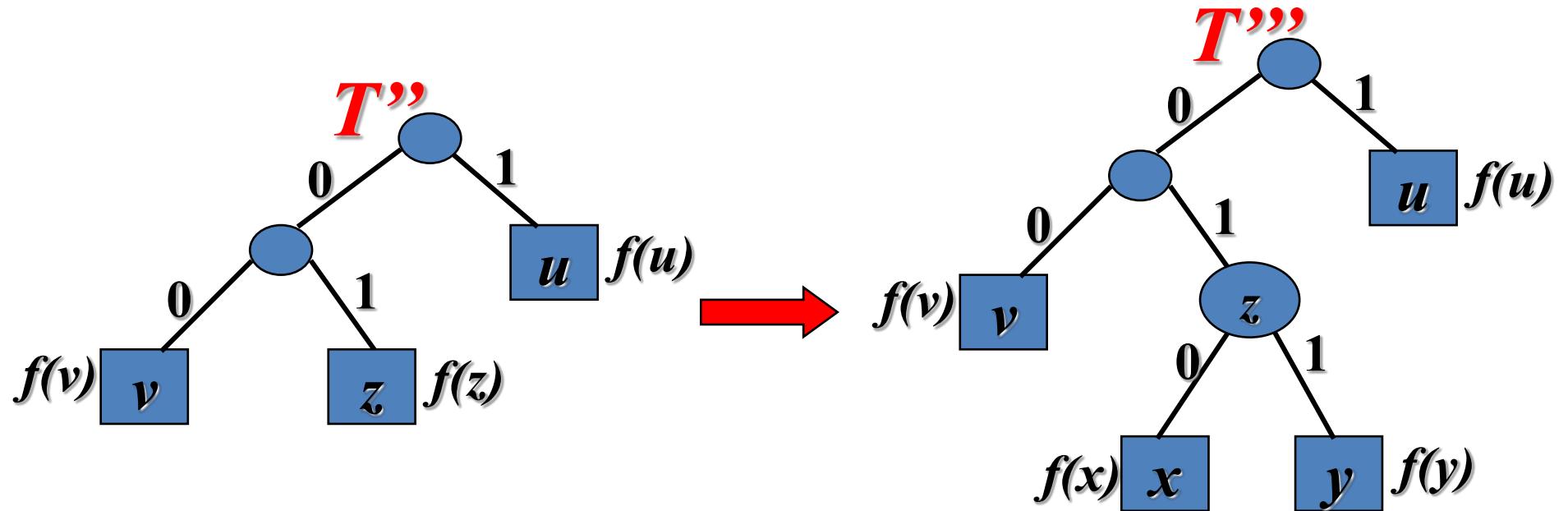






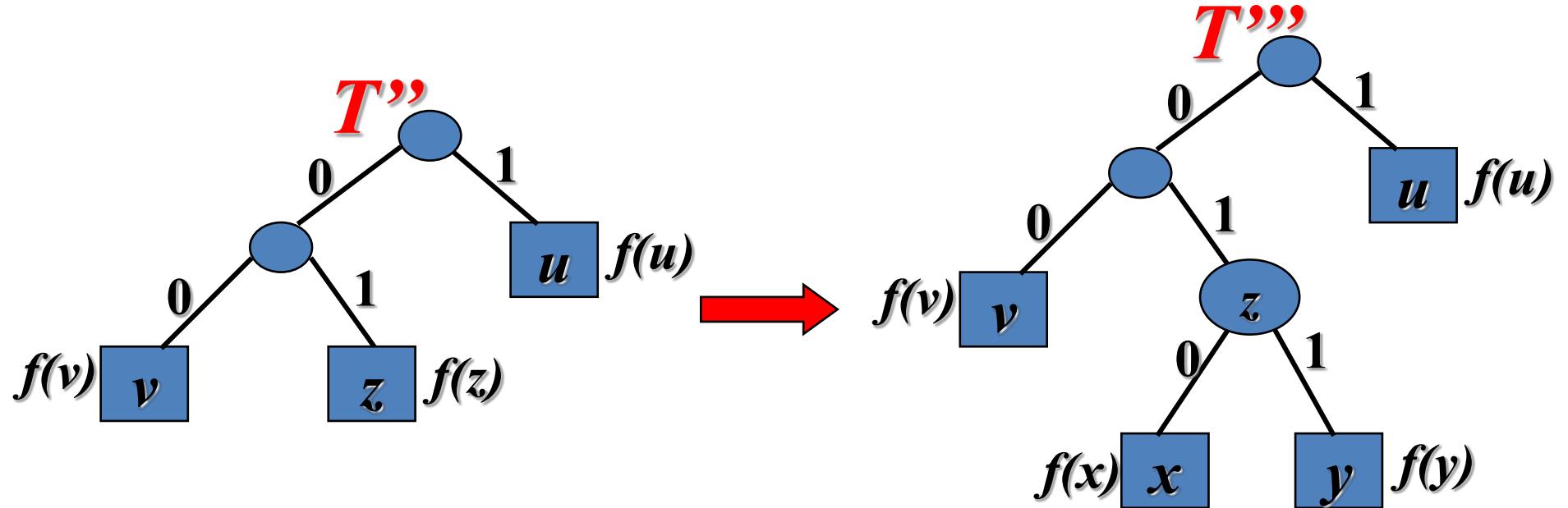
T''' 代价为：

$$B(T''') = \dots + (f(x) + f(y))(d_{T''}(z) + 1)$$



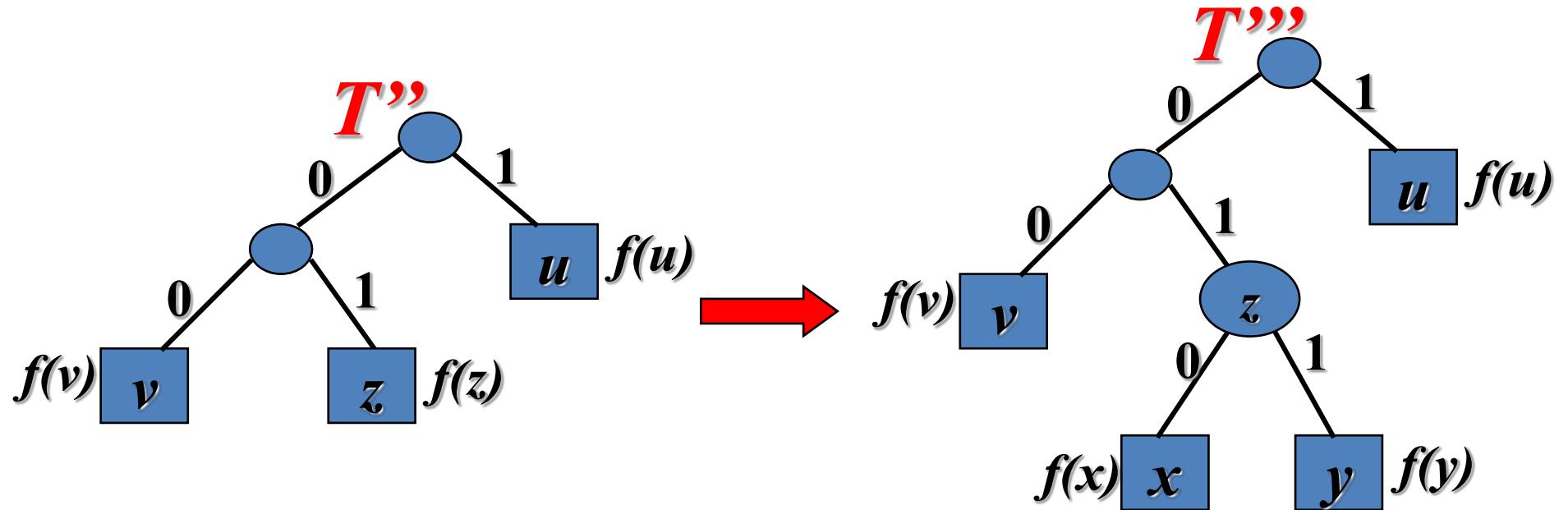
T''' 代价为：

$$\begin{aligned}
 B(T''') &= \dots + (f(x) + f(y))(d_{T''}(z) + 1) \\
 &= \dots + f(z)d_{T''}(z) + (f(x) + f(y)) \quad (d_{T''}(z) = d_T(z))
 \end{aligned}$$



T''' 代价も：

$$\begin{aligned}
 B(T''') &= \dots + (f(x) + f(y))(d_{T''}(z) + 1) \\
 &= \dots + f(z)d_{T''}(z) + (f(x) + f(y)) \quad (d_{T''}(z) = d_{T'}(z)) \\
 &= B(T'') + f(x) + f(y) < B(T') + f(x) + f(y) = B(T)
 \end{aligned}$$



T''' 代价为：

$$\begin{aligned}
 B(T''') &= \dots + (f(x) + f(y))(d_{T''}(z) + 1) \\
 &= \dots + f(z)d_{T''}(z) + (f(x) + f(y)) \quad (d_{T''}(z) = d_T(z)) \\
 &= B(T'') + f(x) + f(y) < B(T') + f(x) + f(y) = B(T)
 \end{aligned}$$

与 \$T\$ 是优化的矛盾，故 \$T'\$ 是 \$C'\$ 的优化编码树。

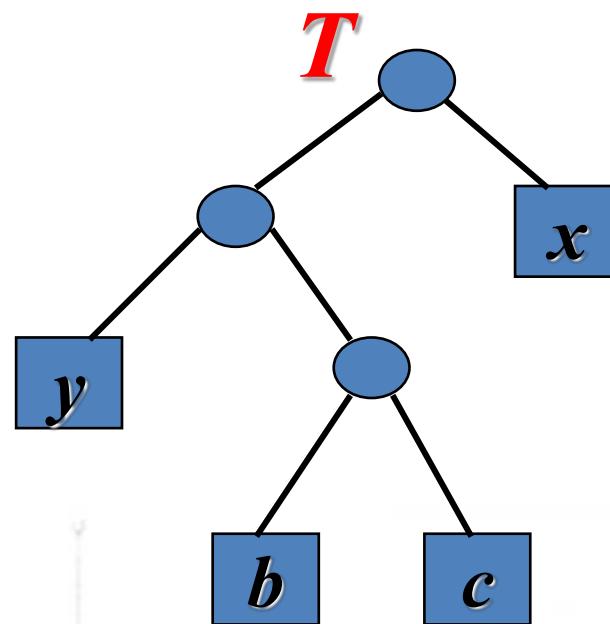
- 贪心选择性

引理2. 设 C 是字母表， $\forall c \in C$ ， c 具有频率 $f(c)$ ， x 、 y 是 C 中具有最小频率的两个字符，则存在一个 C 的优化前缀树， x 与 y 的编码具有相同长度，且仅在最末一位不同。

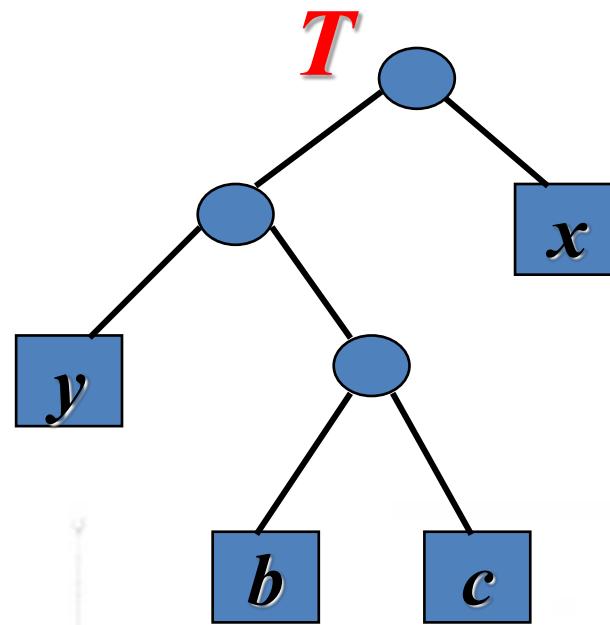
证: 设 T 是 C 的优化前缀树, 且 b 和 c 是具有最大深度的两个兄弟字符:



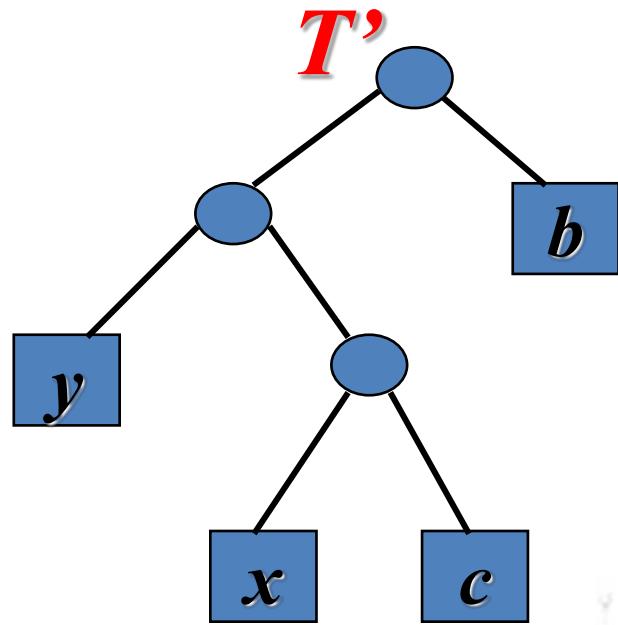
证: 设 T 是 C 的优化前缀树, 且 b 和 c 是具有最大深度的两个兄弟字符:

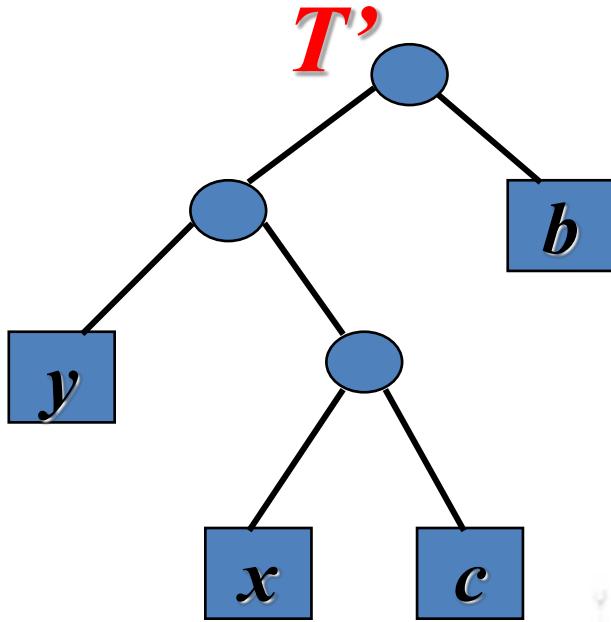


证：设 T 是 C 的优化前缀树，且 b 和 c 是具有最大深度的两个兄弟字符：

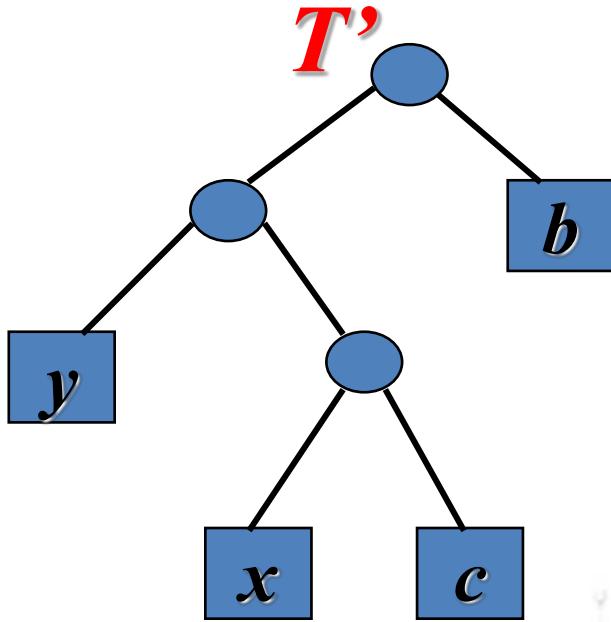


不失一般性，设 $f(b) \leq f(c)$, $f(x) \leq f(y)$. 因 x 与 y 是具有最低频率的字符, $f(b) \geq f(x)$, $f(c) \geq f(y)$. 交换 T 的 b 和 x . 从 T 构造 T' :

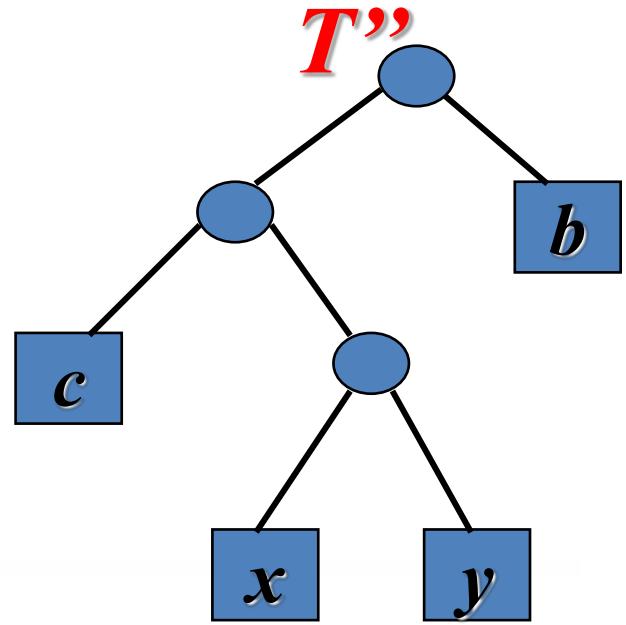




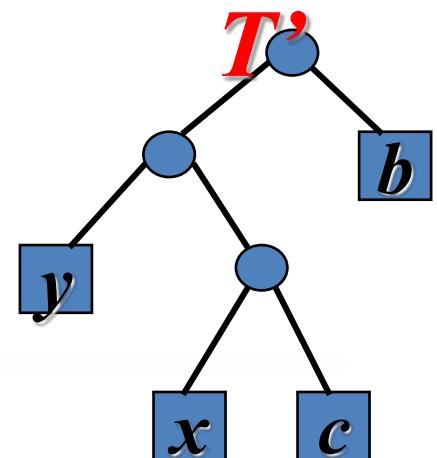
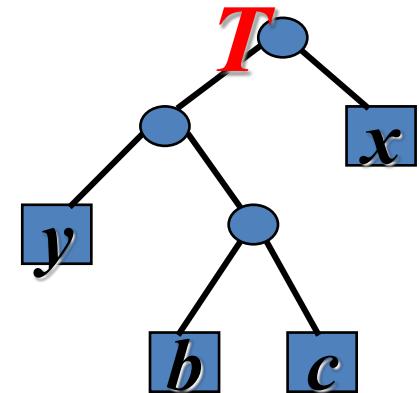
替换 y 和 c
构造 T''



交换 y 和 c
构造 T''



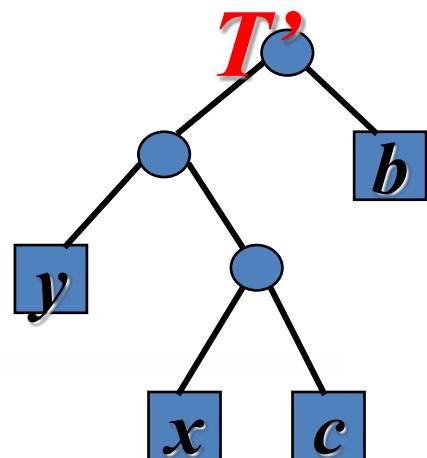
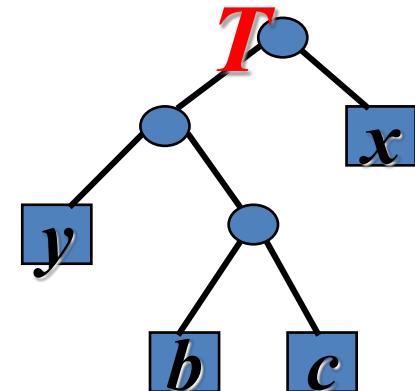
证 T'' 是最优化前缀树.



证 T'' 是最优化前缀树.

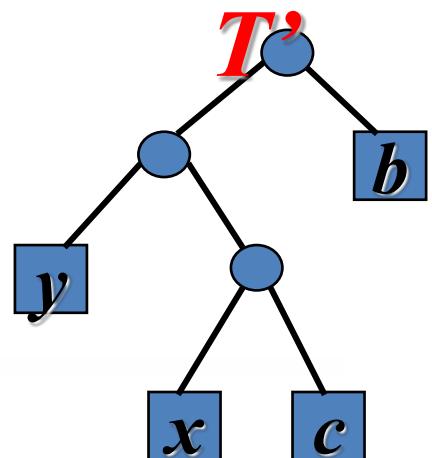
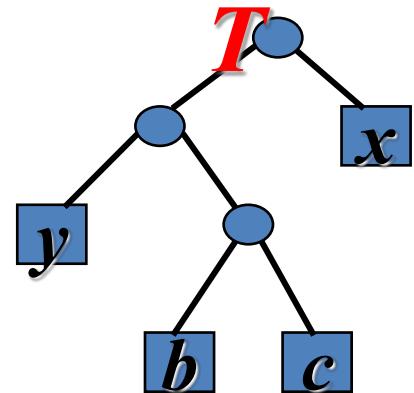
$B(T) - B(T')$

=



证 T'' 是最优化前缀树.

$$\begin{aligned} & B(T) - B(T') \\ &= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c) \end{aligned}$$

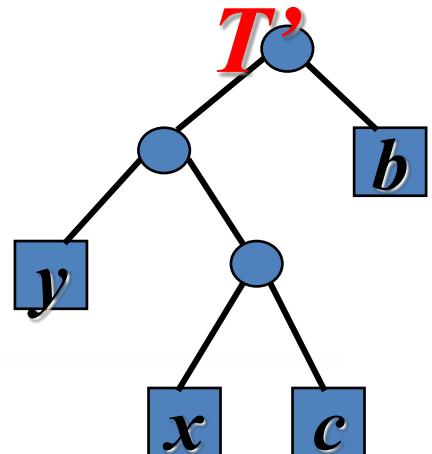
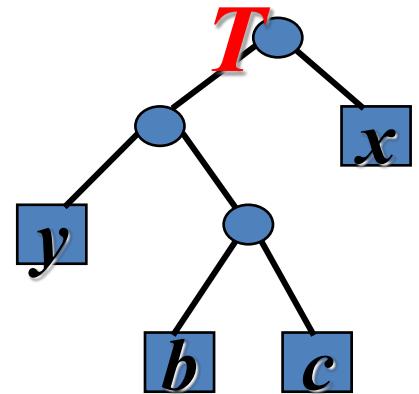


证 T'' 是最优化前缀树.

$B(T) - B(T')$

$$= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$$



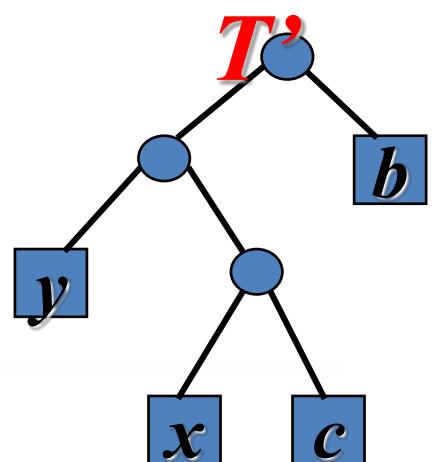
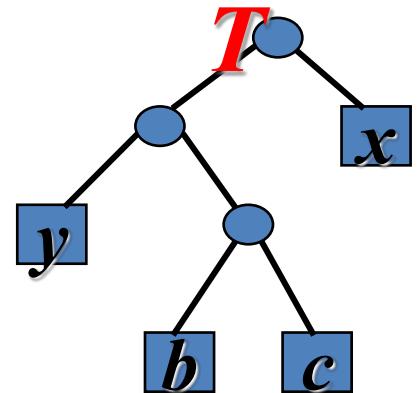
证 T'' 是最优化前缀树.

$B(T) - B(T')$

$$= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(b) - f(b)d_{T'}(x)$$



证 T'' 是最优化前缀树.

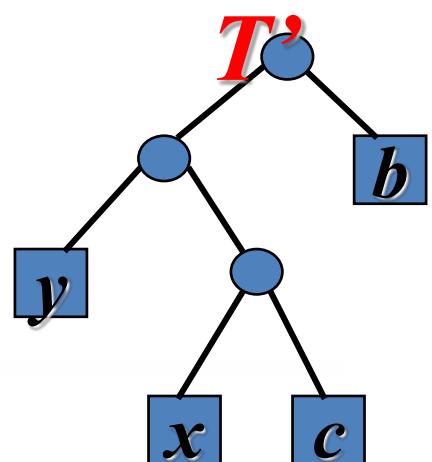
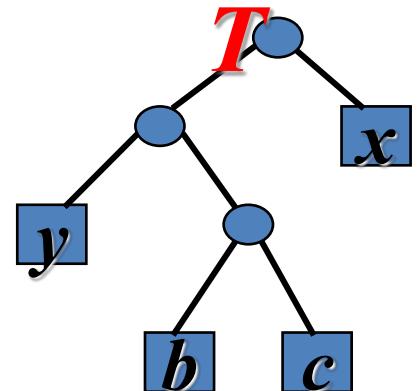
$B(T) - B(T')$

$$= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(b) - f(b)d_{T'}(x)$$

$$= (f(b) - f(x))(d_{T'}(b) - d_{T'}(x)).$$



证 T'' 是最优化前缀树.

$B(T) - B(T')$

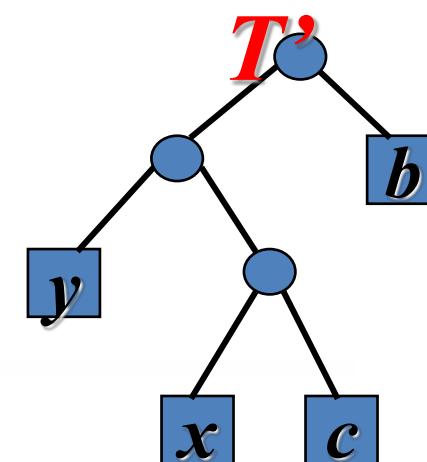
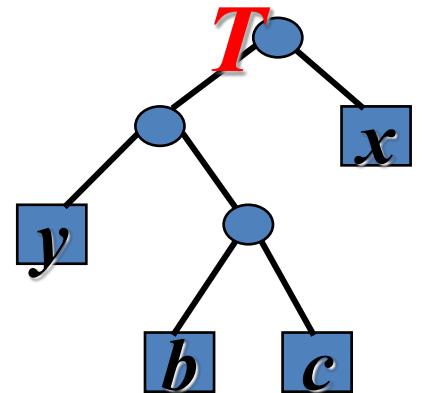
$$= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(b) - f(b)d_{T'}(x)$$

$$= (f(b) - f(x))(d_{T'}(b) - d_{T'}(x)).$$

$\because f(b) \geq f(x), d_{T'}(b) \geq d_{T'}(x)$ (因为 b 的深度最大)



证 T'' 是最优化前缀树.

$B(T) - B(T')$

$$= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

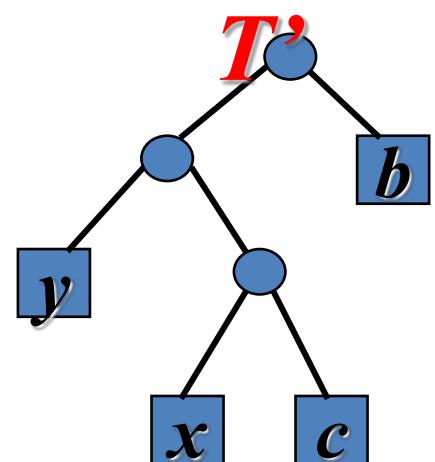
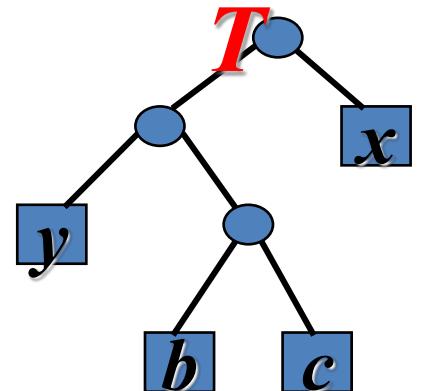
$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(b) - f(b)d_{T'}(x)$$

$$= (f(b) - f(x))(d_{T'}(b) - d_{T'}(x)).$$

$\because f(b) \geq f(x), d_{T'}(b) \geq d_{T'}(x)$ (因为 b 的深度最大)

$\therefore B(T) - B(T') \geq 0, B(T) \geq B(T')$



证 T'' 是最优化前缀树.

$B(T) - B(T')$

$$= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$$

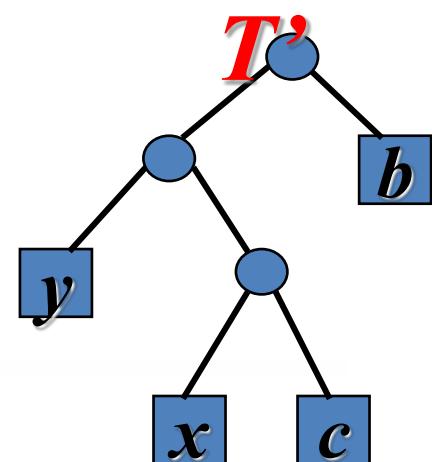
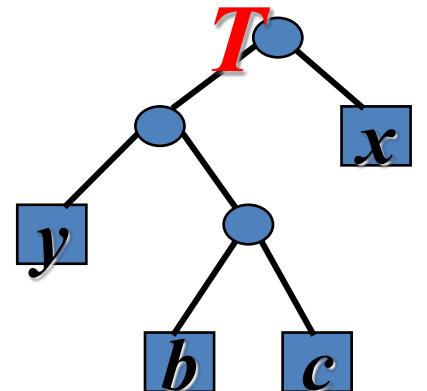
$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(b) - f(b)d_{T'}(x)$$

$$= (f(b) - f(x))(d_{T'}(b) - d_{T'}(x)).$$

$\because f(b) \geq f(x), d_{T'}(b) \geq d_{T'}(x)$ (因为 b 的深度最大)

$\therefore B(T) - B(T') \geq 0, B(T) \geq B(T')$

同理可证 $B(T') \geq B(T'')$. 于是 $B(T) \geq B(T'')$.



证 T'' 是最优化前缀树.

$$B(T) - B(T')$$

$$= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(b) - f(b)d_{T'}(x)$$

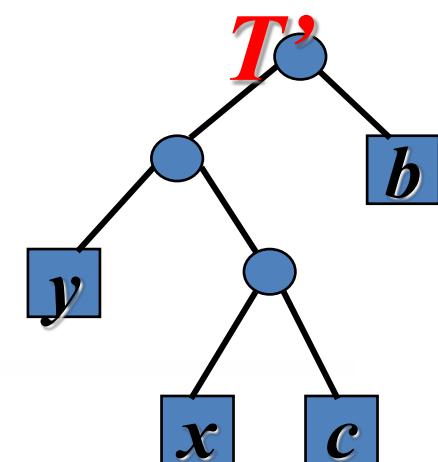
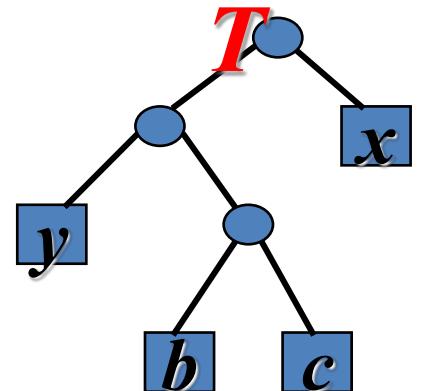
$$= (f(b) - f(x))(d_{T'}(b) - d_{T'}(x)).$$

$\because f(b) \geq f(x), d_{T'}(b) \geq d_{T'}(x)$ (因为 b 的深度最大)

$$\therefore B(T) - B(T') \geq 0, B(T) \geq B(T')$$

同理可证 $B(T') \geq B(T'')$. 于是 $B(T) \geq B(T'')$.

由于 T 是最优化的, 所以 $B(T) \leq B(T'')$.



证 T'' 是最优化前缀树.

$B(T) - B(T')$

$$= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(b) - f(b)d_{T'}(x)$$

$$= (f(b) - f(x))(d_{T'}(b) - d_{T'}(x)).$$

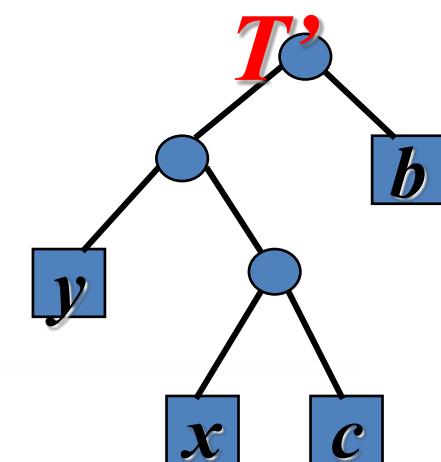
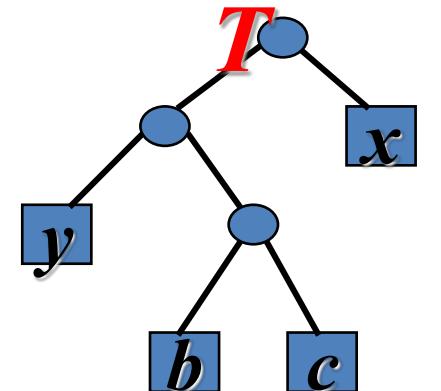
$\because f(b) \geq f(x), d_{T'}(b) \geq d_{T'}(x)$ (因为 b 的深度最大)

$$\therefore B(T) - B(T') \geq 0, B(T) \geq B(T')$$

同理可证 $B(T') \geq B(T'')$. 于是 $B(T) \geq B(T'')$.

由于 T 是最优化的, 所以 $B(T) \leq B(T'')$.

于是, $B(T) = B(T'')$, T'' 是 C 的最优化前缀编码树.



证 T'' 是最优化前缀树.

$B(T) - B(T')$

$$= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(b) - f(b)d_{T'}(x)$$

$$= (f(b) - f(x))(d_{T'}(b) - d_{T'}(x)).$$

$\because f(b) \geq f(x), d_{T'}(b) \geq d_{T'}(x)$ (因为 b 的深度最大)

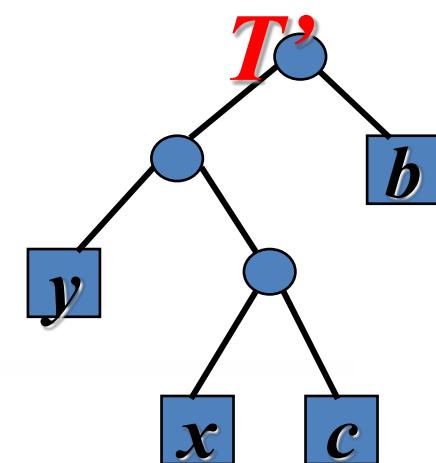
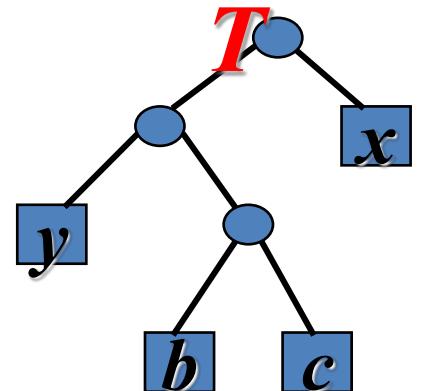
$$\therefore B(T) - B(T') \geq 0, B(T) \geq B(T')$$

同理可证 $B(T') \geq B(T'')$. 于是 $B(T) \geq B(T'')$.

由于 T 是最优化的, 所以 $B(T) \leq B(T'')$.

于是, $B(T) = B(T'')$, T'' 是 C 的最优化前缀编码树.

在 T'' 中, x 和 y 具有相同长度编码, 且仅最后一位不同.



• 基本思想

- 循环地选择具有最低频率的两个结点，生成一棵子树，直至形成树
- 初始： $f:5, e:9, c:12, b:13, d:16, a:45$

示例

例如: f:5, e:9, c:12, b:13, d:16, a:45

循环地选择具有最低频率的两个结点，生成一棵子树，直至所有结点形成
一棵树

A: 45

B: 13

C: 12

D: 16

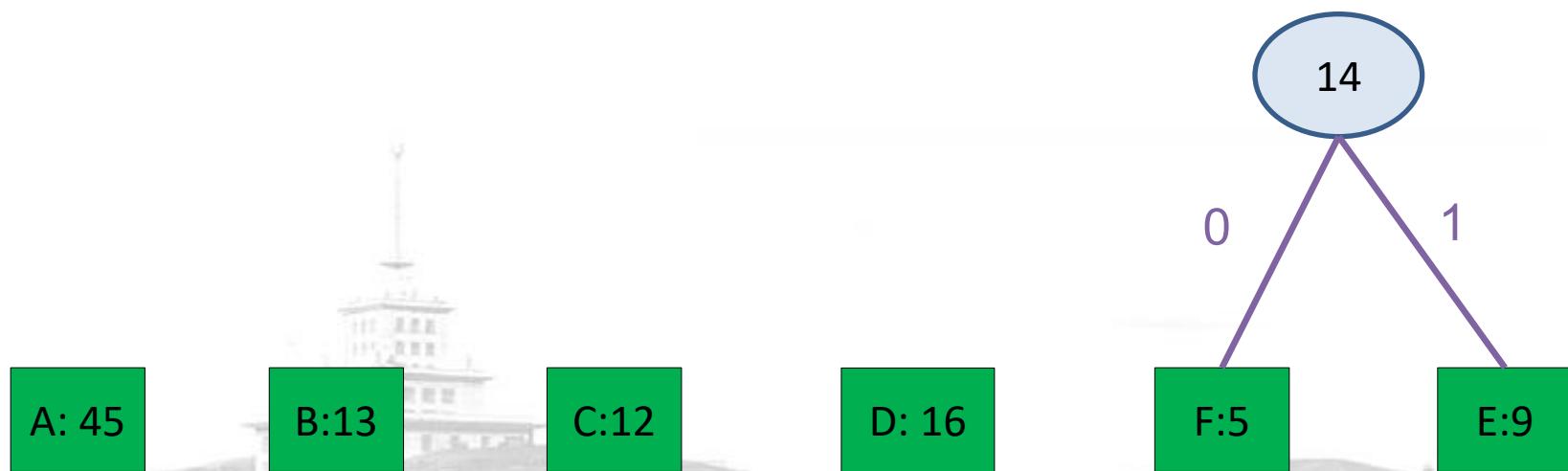
F: 5

E: 9

示例

例如: f:5, e:9, c:12, b:13, d:16, a:45

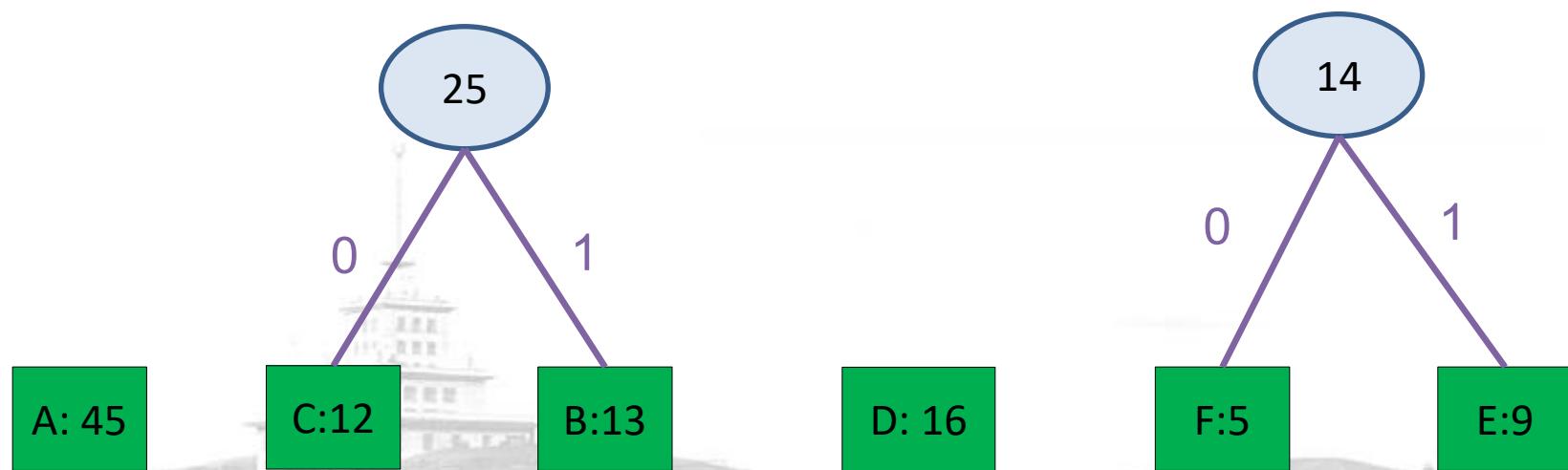
循环地选择具有最低频率的两个结点，生成一棵子树，直至所有结点形成一棵树



示例

例如: f:5, e:9, c:12, b:13, d:16, a:45

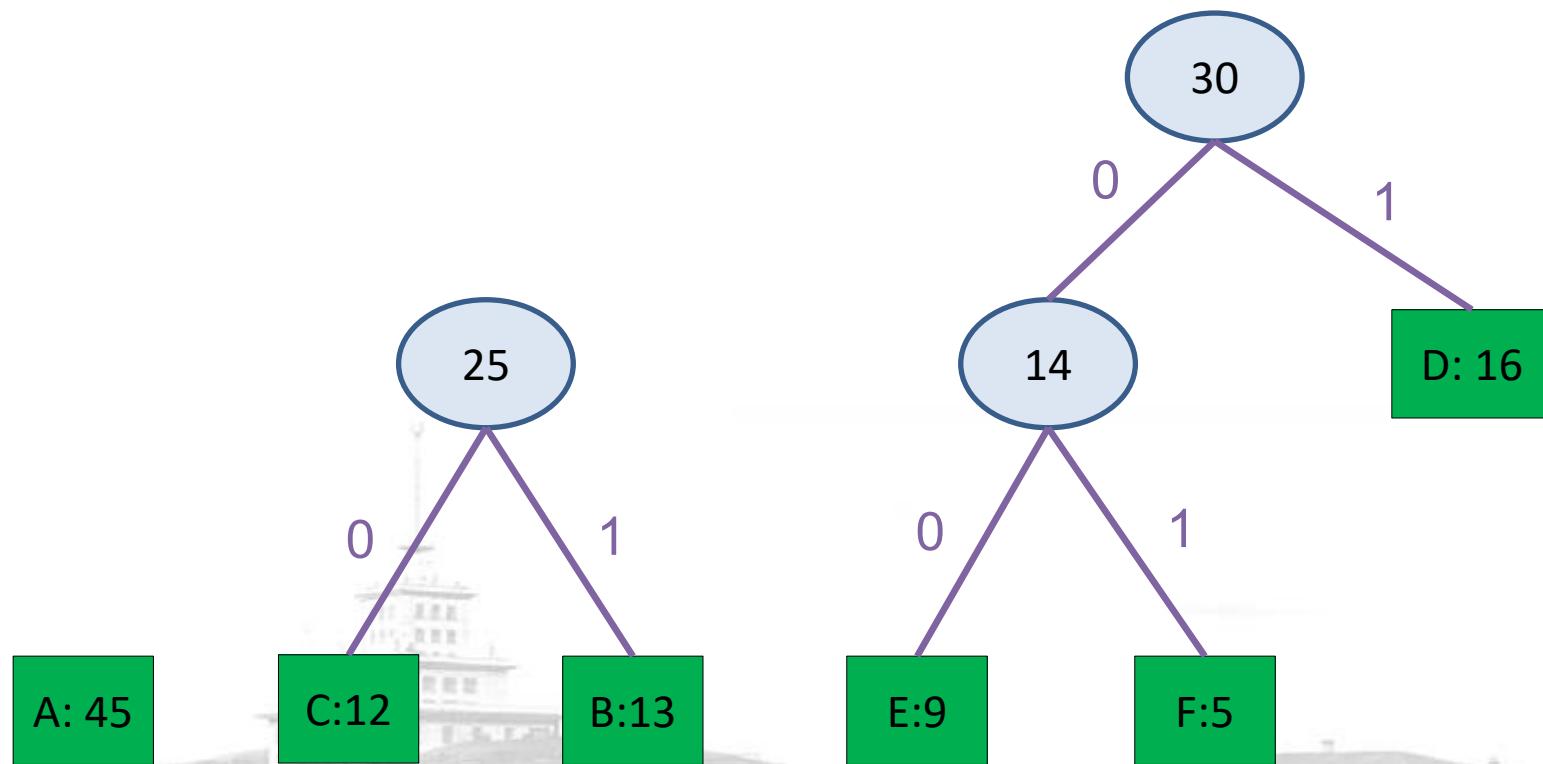
循环地选择具有最低频率的两个结点，生成一棵子树，直至所有结点形成一棵树



示例

例如: f:5, e:9, c:12, b:13, d:16, a:45

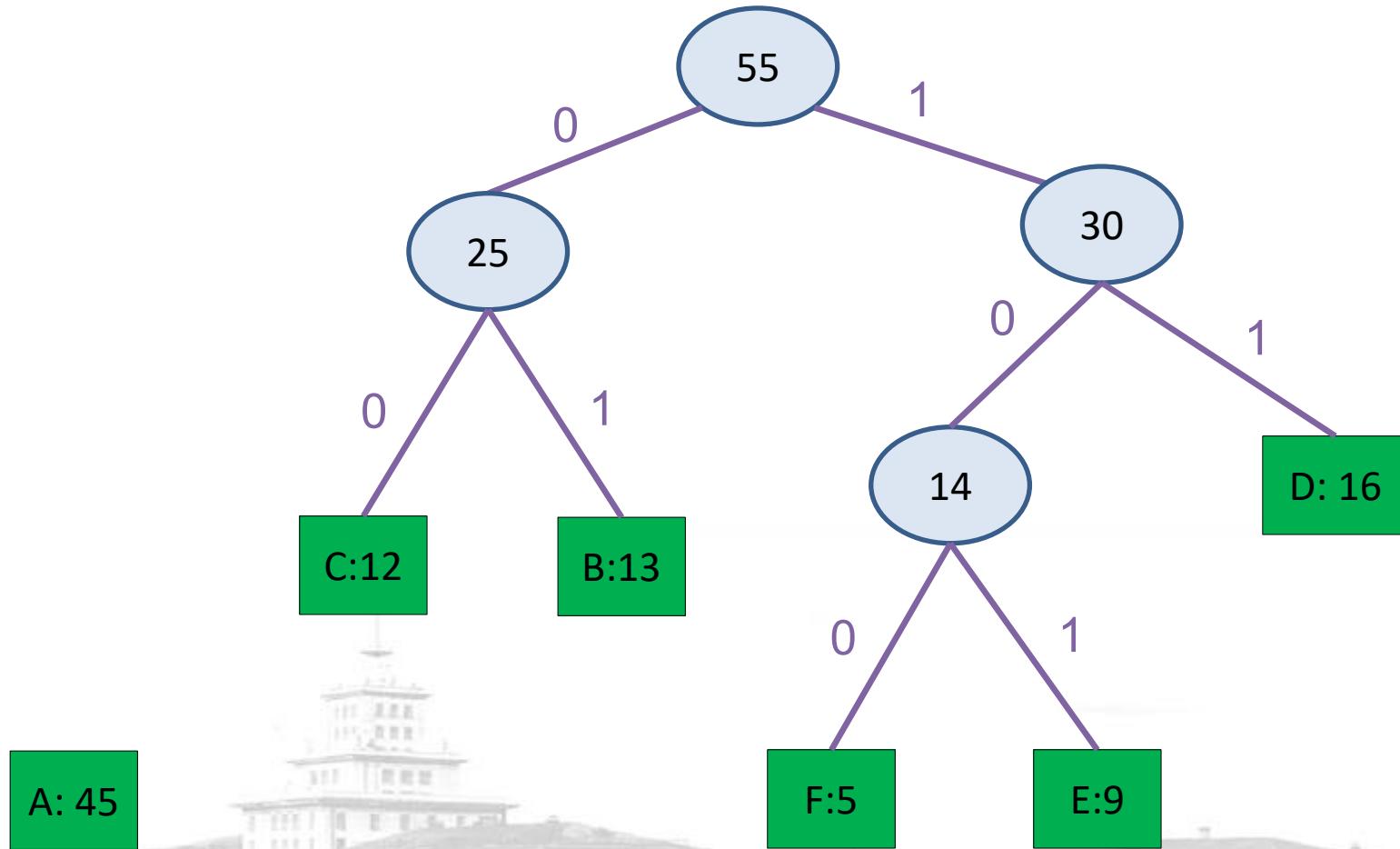
循环地选择具有最低频率的两个结点，生成一棵子树，直至所有结点形成一棵树



示例

例如: f:5, e:9, c:12, b:13, d:16, a:45

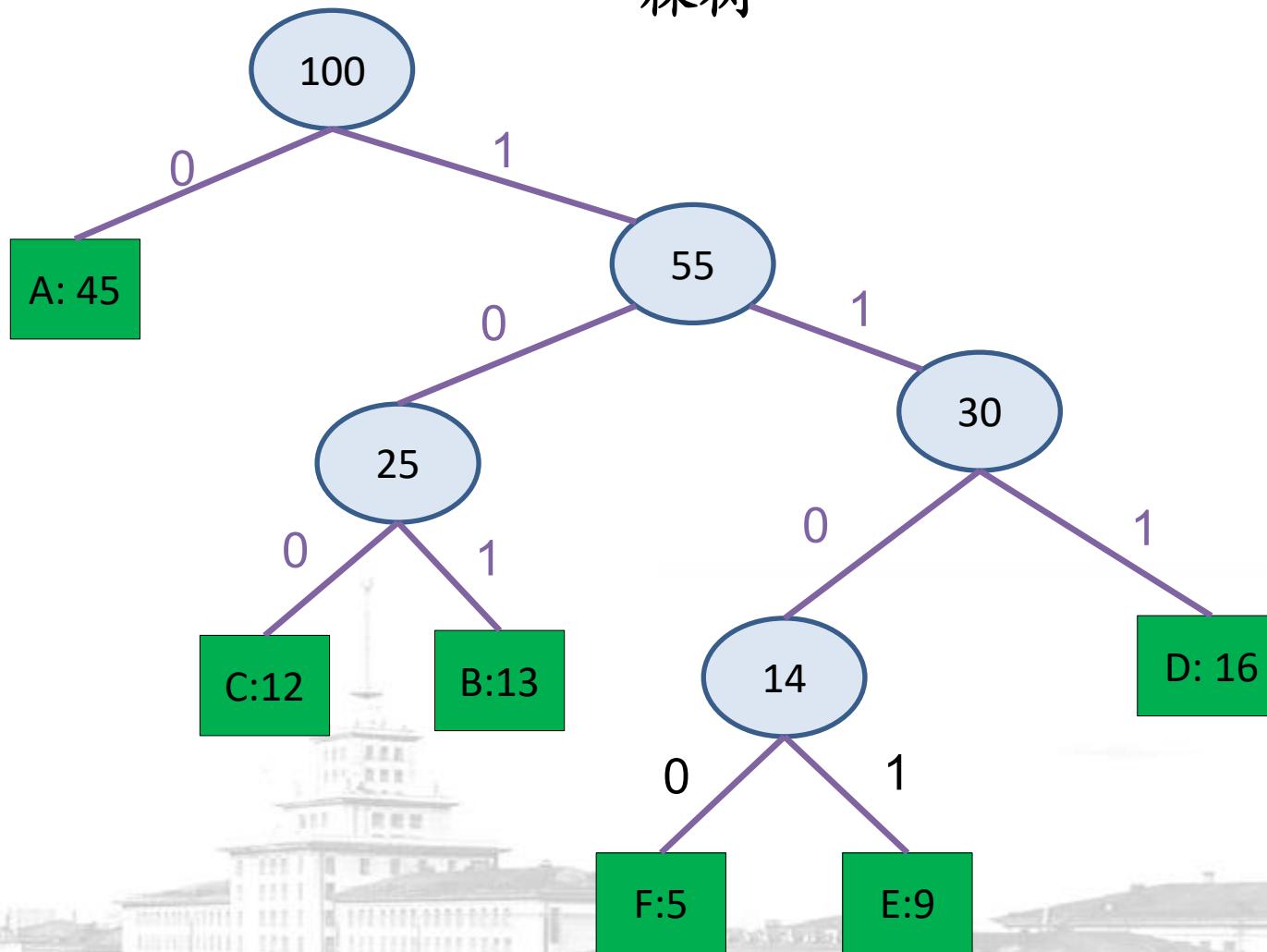
循环地选择具有最低频率的两个结点，生成一棵子树，直至所有结点形成
一棵树



示例

例如: f:5, e:9, c:12, b:13, d:16, a:45

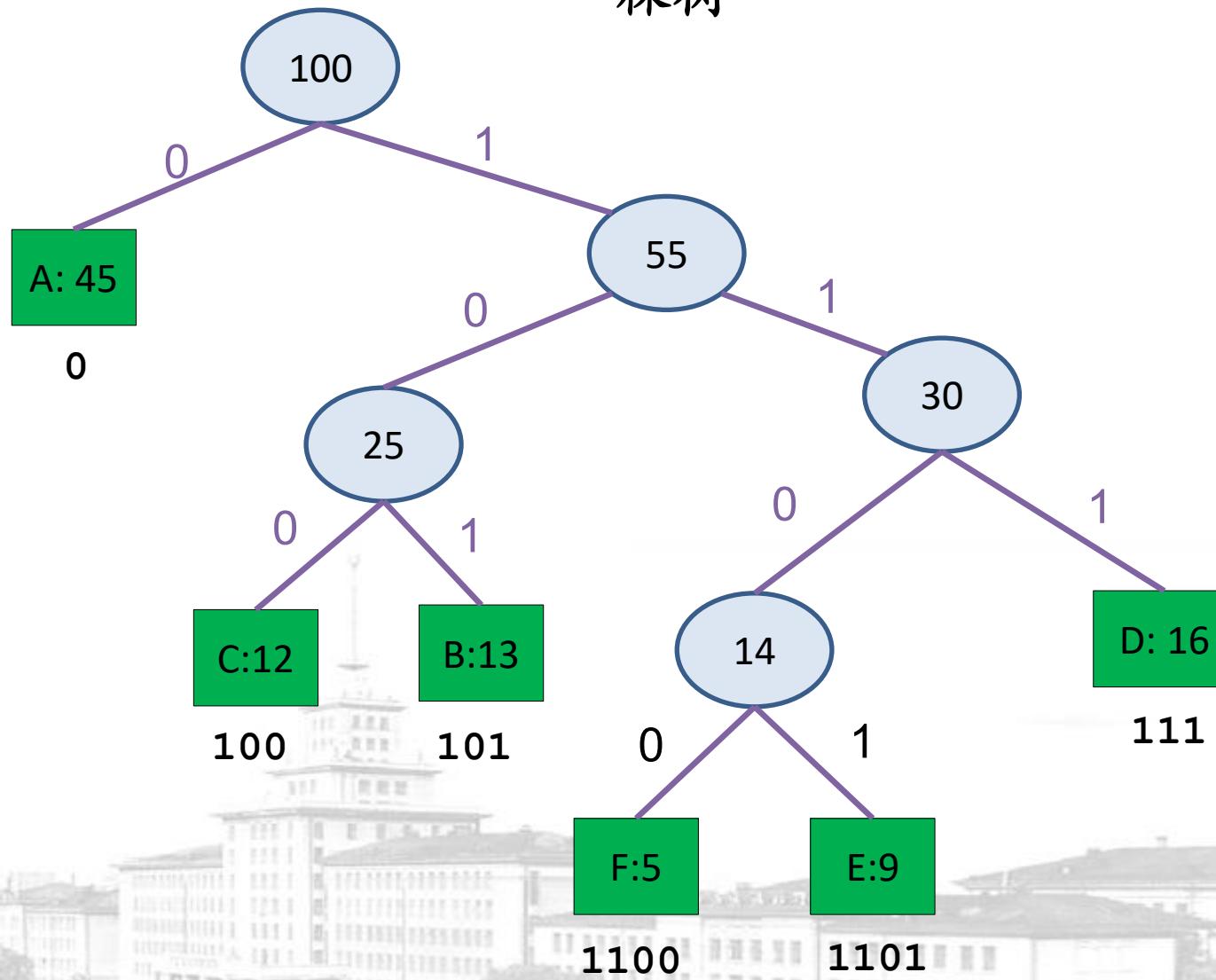
循环地选择具有最低频率的两个结点，生成一棵子树，直至所有结点形成
一棵树



示例

例如: f:5, e:9, c:12, b:13, d:16, a:45

循环地选择具有最低频率的两个结点，生成一棵子树，直至所有结点形成
一棵树



- 贪心算法(使用堆操作实现)

Huffman(C, F)

1. $n \leftarrow |C|;$
2. $Q \leftarrow C; /*$ 用BUILD-HEAP建立堆 */
3. FOR $i \leftarrow 1$ To $n-1$ Do
4. $z \leftarrow \text{Allocate-Node}();$
5. $x \leftarrow \text{left}[z] \leftarrow \text{Extract-MIN}(Q); /*$ 堆操作 */
6. $y \leftarrow \text{right}[z] \leftarrow \text{Extract-MIN}(Q); /*$ 堆操作 */
7. $f(z) \leftarrow f(x) + f(y);$
8. $\text{Insert}(Q, z); /*$ 堆操作 */
9. Return

- 设 Q 由一个堆实现
- 第 2 步用堆排序的 BUILD-HEAP 实现: $O(n)$
- 每个堆操作要求 $O(\log n)$, 循环 $n-1$ 次: $O(n \log n)$
- $T(n) = O(n) + O(n \log n) = O(n \log n)$



正确性证明

定理. Huffman算法产生一个优化前缀编码树

证. 由于引理1、引理2成立，而且Huffman算法按照引理2的贪心这样性确定的规则进行局部优化这样，所以Huffman算法产生一个优化前缀编码树。

- 1、果园里有n堆果子，达达决定把所有的果子合成一堆。
- 2、每一次合并，达达可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。
- 3、所有的果子经过 $n-1$ 次合并之后，就只剩下一堆了。
- 4、合并果子时总共消耗的体力等于每次合并所耗体力之和。
- 5、每个果子重量都为 1

设计出合并的次序方案，使达达耗费的体力最少，并输出这个最小的体力耗费值。

输入：

例如有 33 种果子，数目依次为 1, 2, 91, 2, 9。

解释：

可以先将 1、21、2 堆合并，新堆数目为 33，耗费体力为 33。

接着，将新堆与原先的第三堆合并，又得到新的堆，数目为 1212，耗费体力为 1212。

所以达达总共耗费体力=3+12=15=3+12=15。

可以证明 1515 为最小的体力耗费值。

提纲

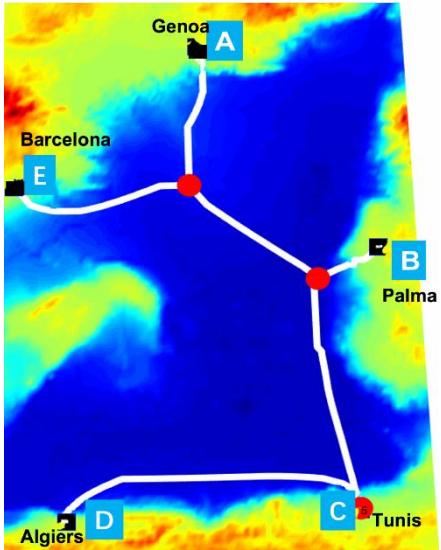
5.1 贪心法的基本原理

5.2 任务安排问题

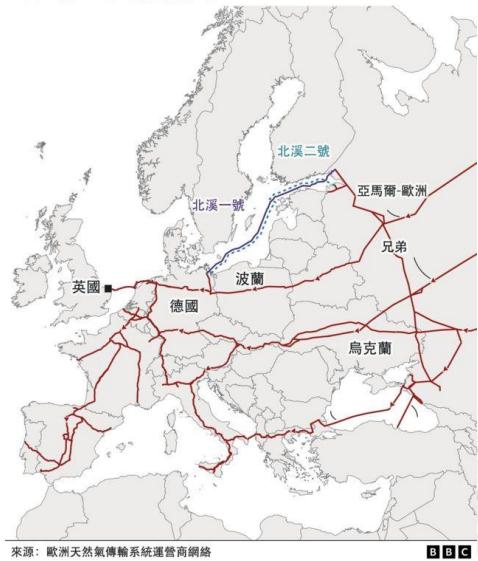
5.3 哈夫曼编码问题

5.4 最小生成树

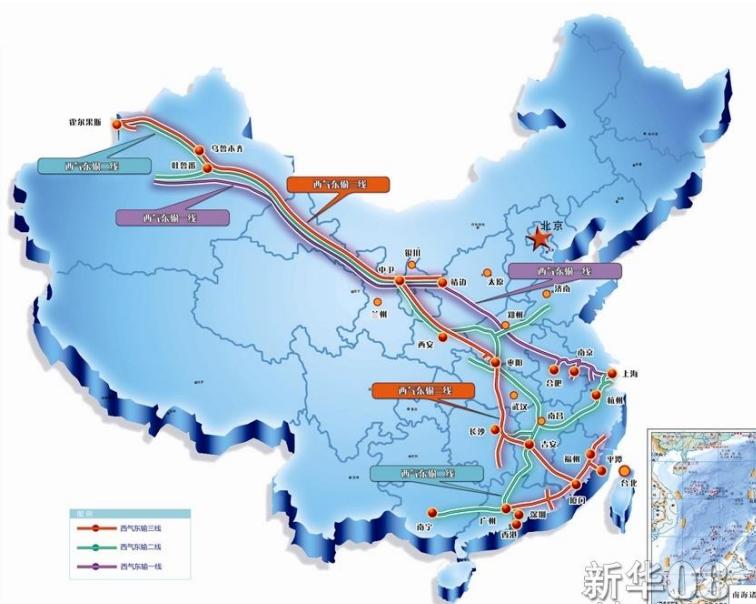




歐洲最重要的輸氣管道



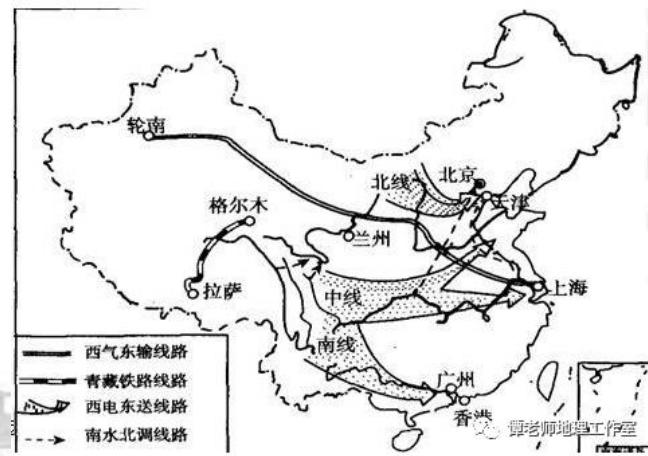
來源：歐洲天然氣傳輸系統運營商網絡



西氣東輸一線、二線、三線天然氣管道工程走向示意圖

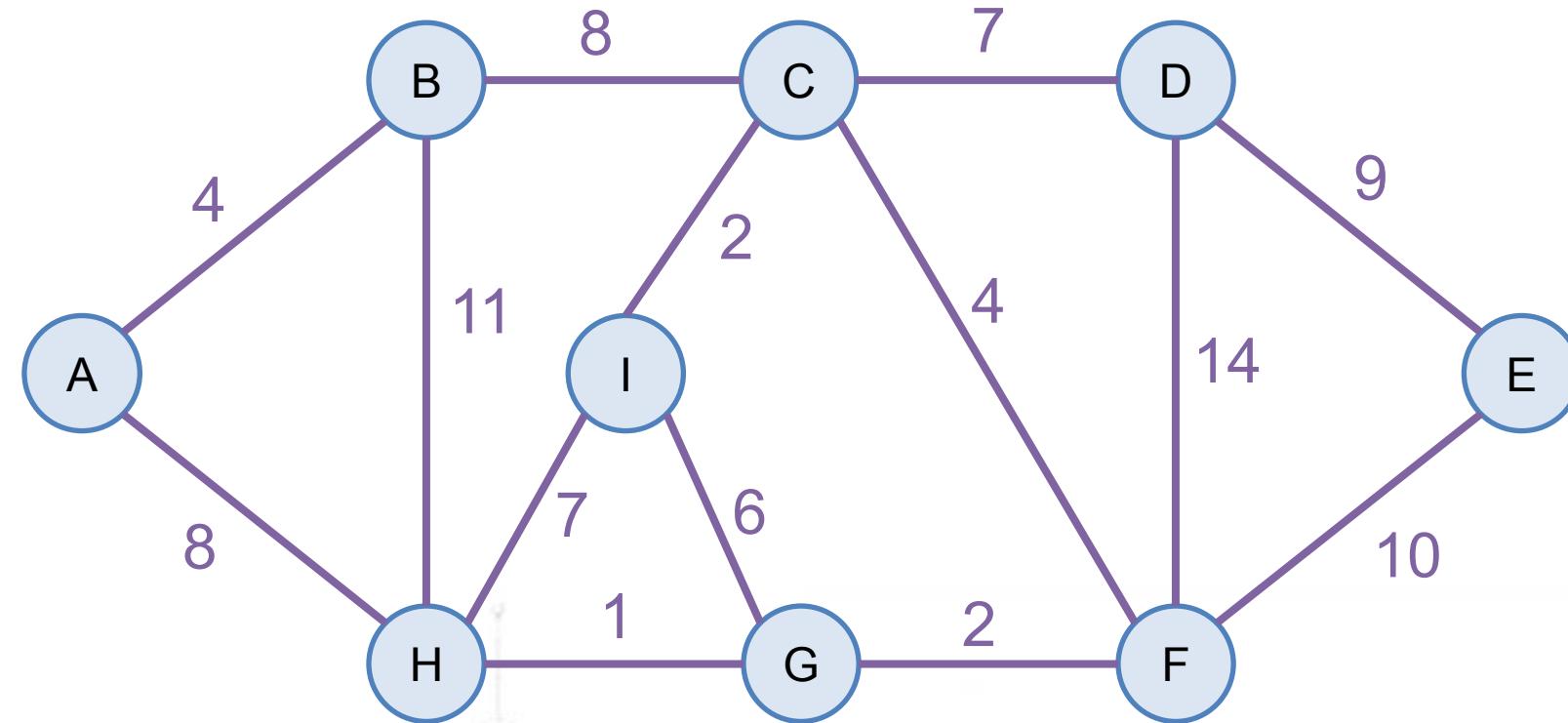


新华08
xinhud08.com



最小生成树 (Minimum Spanning Tree)

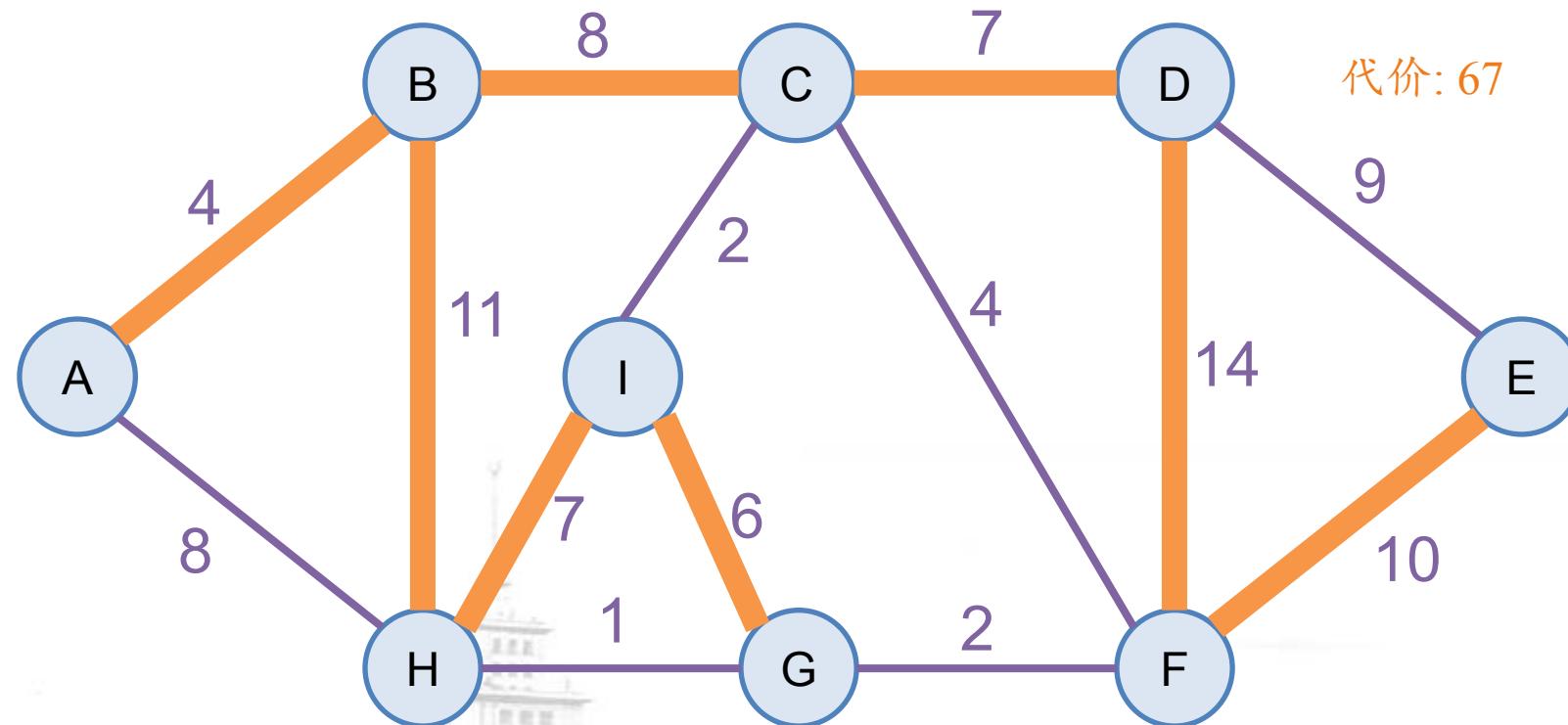
$G = (V, E)$ 是一个边加权无向连通图



- **生成树**: 连通图 G 的一个子图如果是一棵包含 G 的所有顶点的树，则该子图称为 G 的生成树。
- 生成树是连通图的极小连通子图。所谓极小是指：若在树中任意增加一条边，则将出现一个回路；若去掉一条边，将会使之变成非连通图。

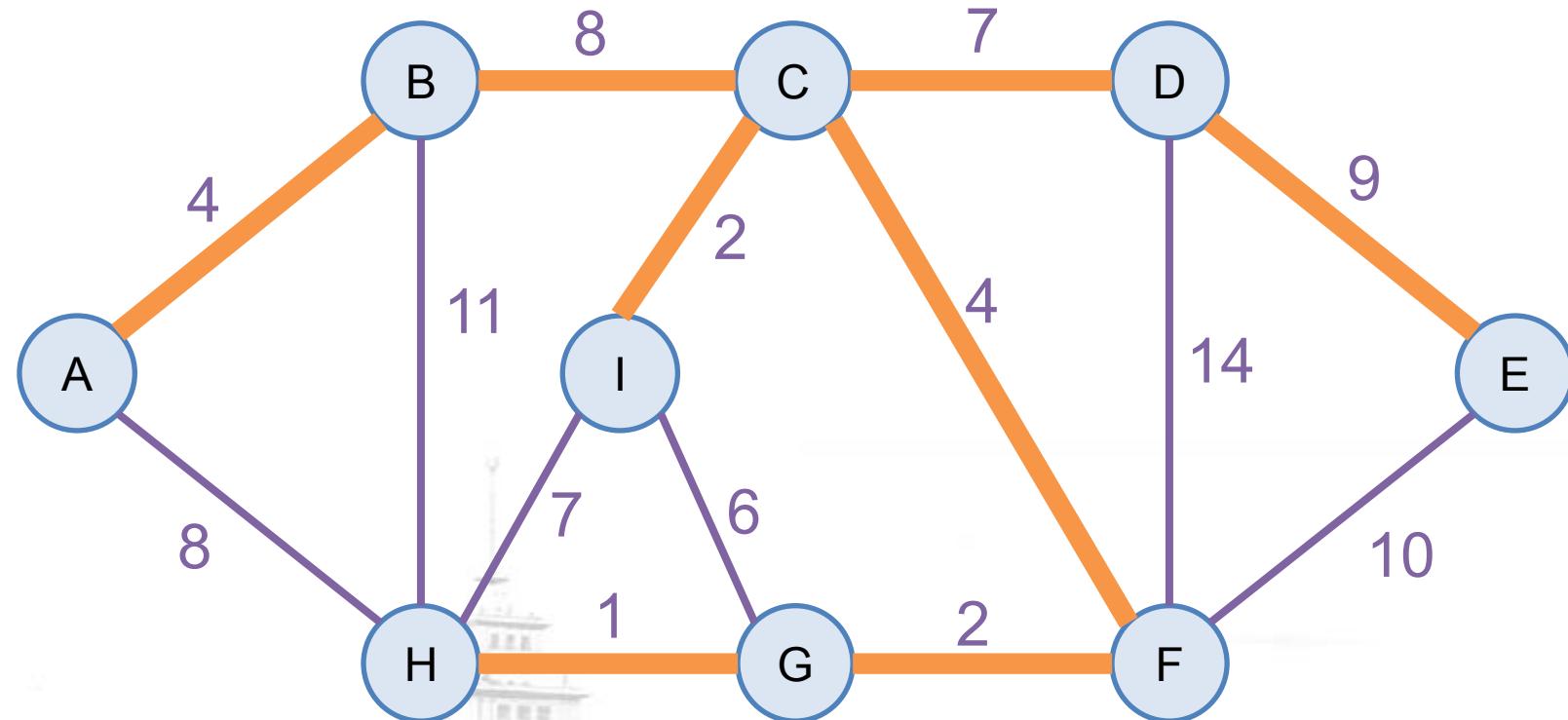
最小生成树 (Minimum Spanning Tree)

- 生成树是一个连接了图中 G 所有结点 V 的树
- 生成树的代价是生成树上所有边的权重之和



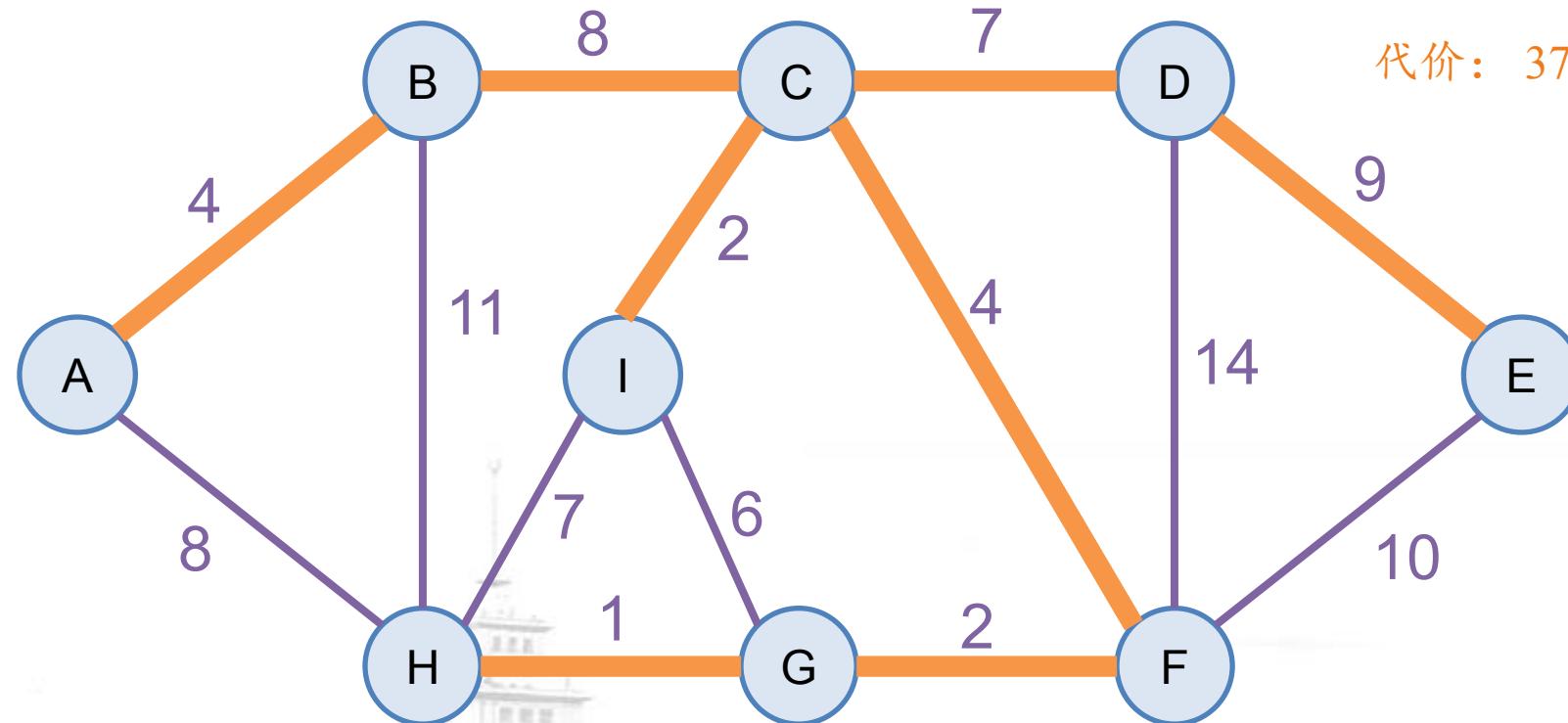
最小生成树 (Minimum Spanning Tree)

- 生成树是一个连接图中所有结点的树
- 生成树的代价是生成树上所有边的权重之和

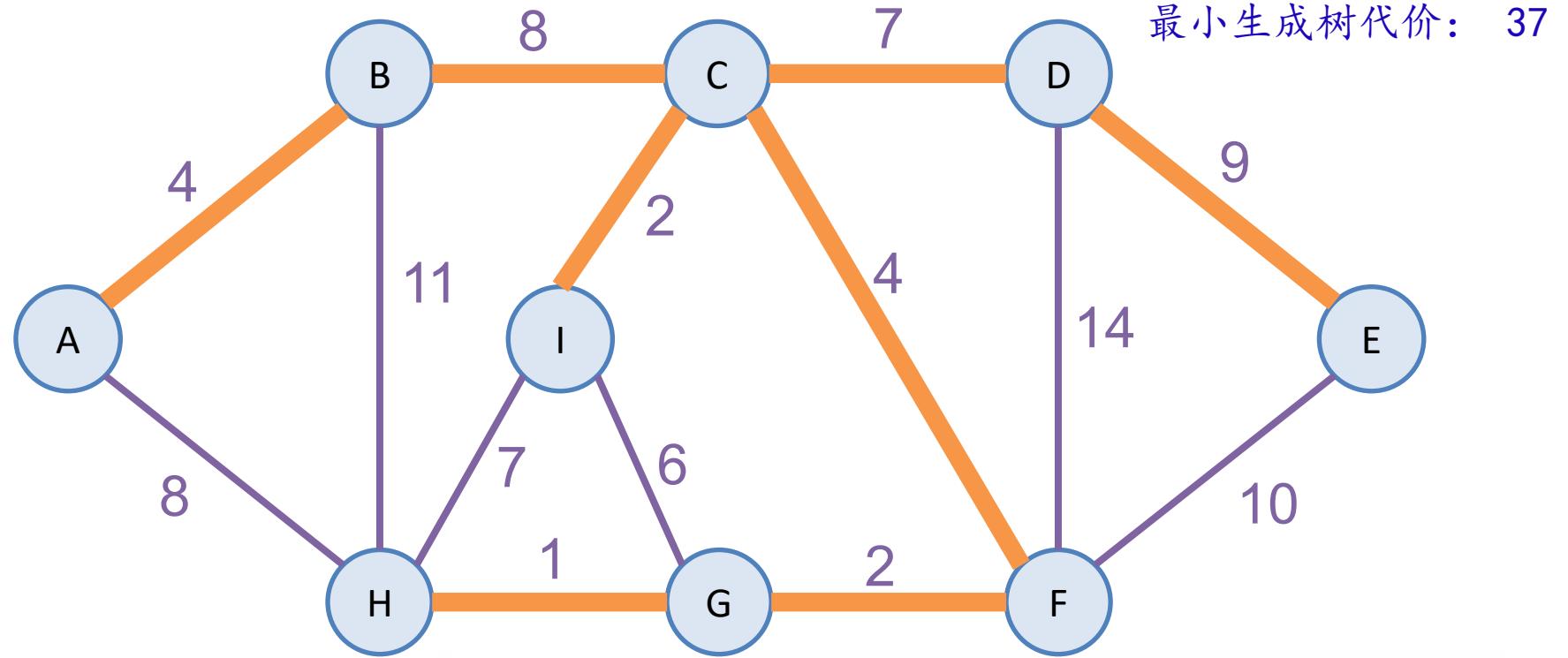


最小生成树 (Minimum Spanning Tree)

- 生成树是一个连接图中所有结点的树
- 生成树的代价是生成树上所有边的权重之和



问题定义



输入：一个边加权无向连通图 $G = (V, E)$

输出：最小代价的生成树 **(不唯一)**

问题的定义



问题的定义

- 生成树

- 设 $G=(V, E)$ 是一个边加权无向连通图. G 的生成树是无向树 $S=(V, T)$, $T \subseteq E$, 以下用 T 表示 S .
- 如果 $W: E \rightarrow \{\text{实数}\}$ 是 G 的权函数, T 的权值定义为 $W(T) = \sum_{(u,v) \in T} W(u,v)$.



问题的定义

- 生成树
 - 设 $G=(V, E)$ 是一个边加权无向连通图. G 的生成树是无向树 $S=(V, T)$, $T \subseteq E$, 以下用 T 表示 S .
 - 如果 $W: E \rightarrow \{\text{实数}\}$ 是 G 的权函数, T 的权值定义为 $W(T) = \sum_{(u,v) \in T} W(u,v)$.
- 最小生成树
 - G 的最小生成树是 $W(T)$ 最小的 G 之生成树.

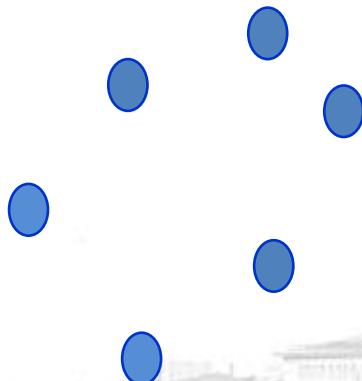


问题的定义

- 生成树
 - 设 $G=(V, E)$ 是一个边加权无向连通图. G 的生成树是无向树 $S=(V, T)$, $T \subseteq E$, 以下用 T 表示 S .
 - 如果 $W: E \rightarrow \{\text{实数}\}$ 是 G 的权函数, T 的权值定义为 $W(T) = \sum_{(u,v) \in T} W(u,v)$.
- 最小生成树
 - G 的最小生成树是 $W(T)$ 最小的 G 之生成树.
- 问题的定义
 - 输入: 无向连通图 $G=(V, E)$, 权函数 W
 - 输出: G 的最小生成树

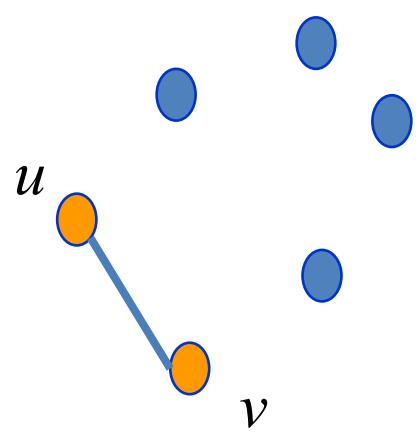
子問題划分

点数: 6



子問題划分

点数: 6

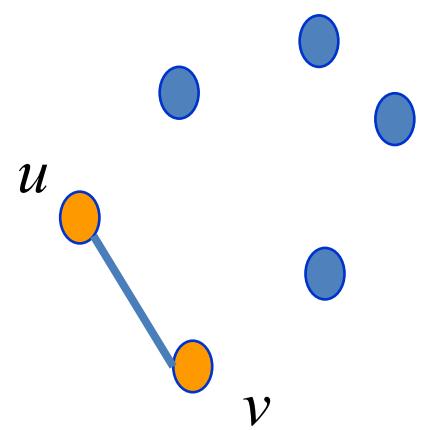


点数: 6

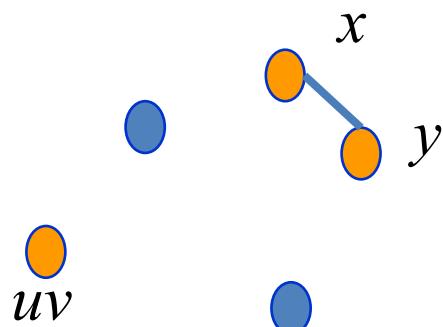


子問題划分

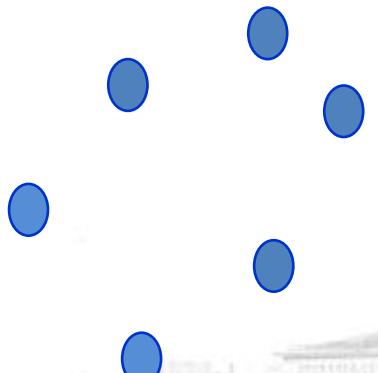
点数: 6



点数: 5

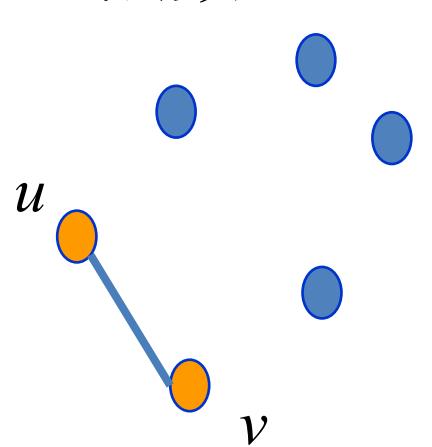


点数: 6

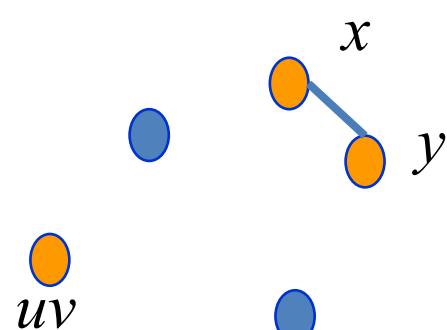


子問題划分

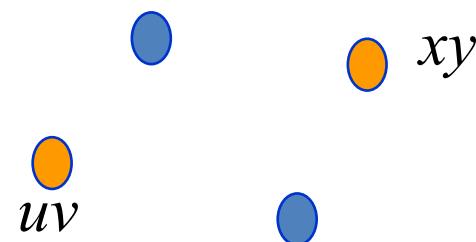
点数: 6



点数: 5



点数: 4

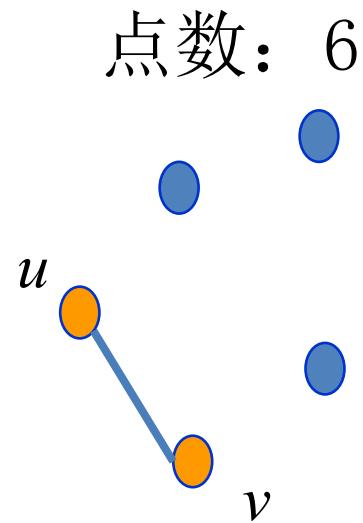


点数: 6

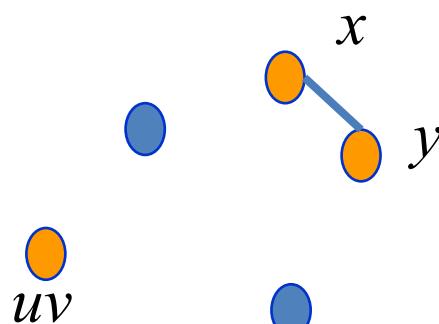


子問題划分

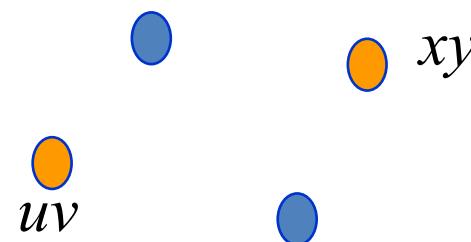
点数: 6



点数: 5



点数: 4

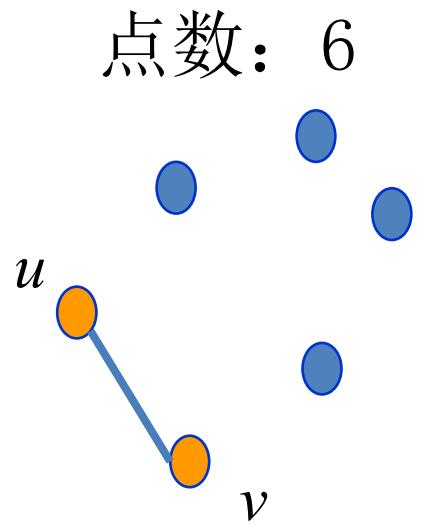


点数: 6

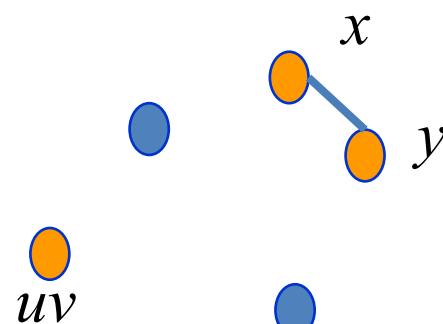


子問題划分

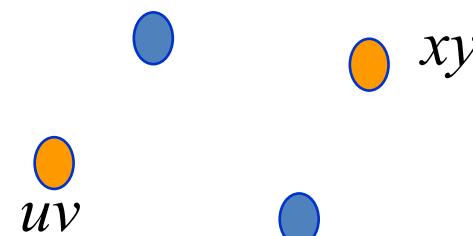
点数: 6



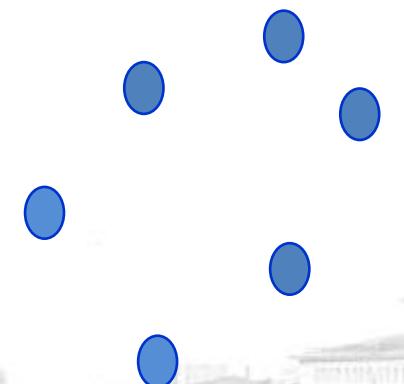
点数: 5



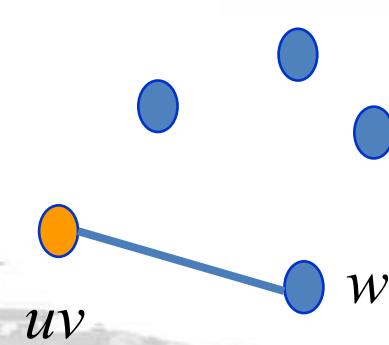
点数: 4



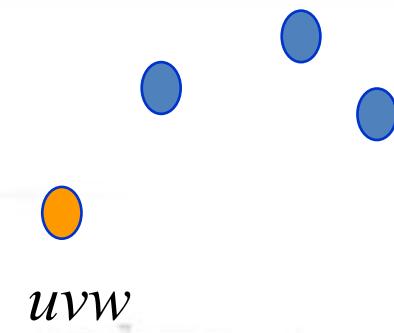
点数: 6



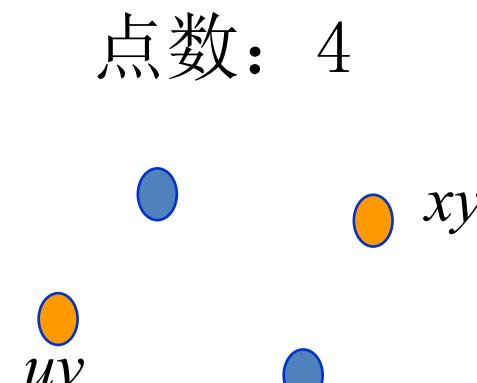
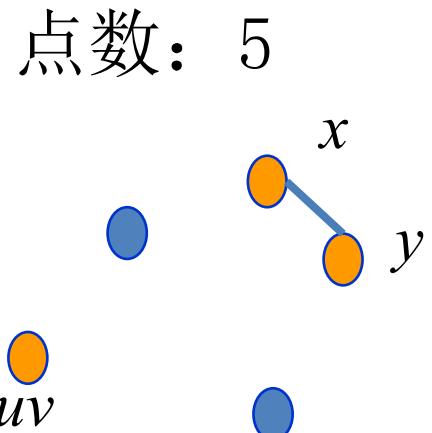
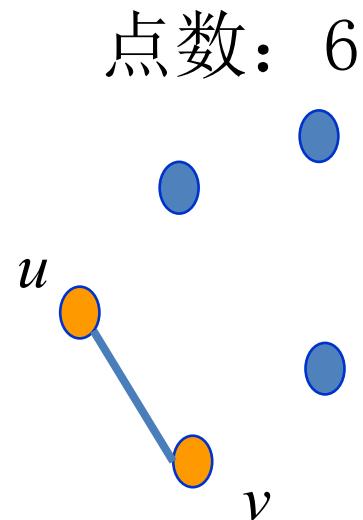
点数: 5



点数: 4

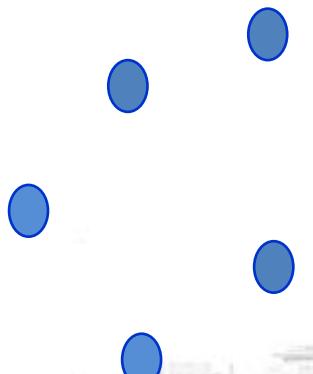


子問題划分

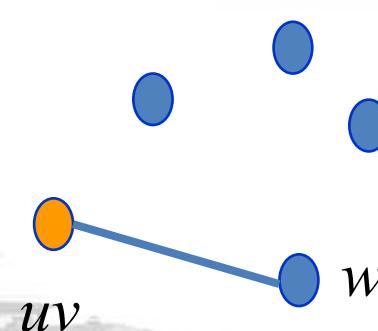


加边法

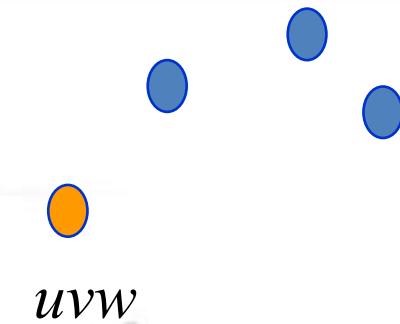
点数: 6



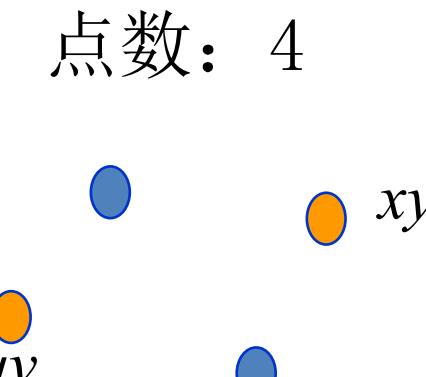
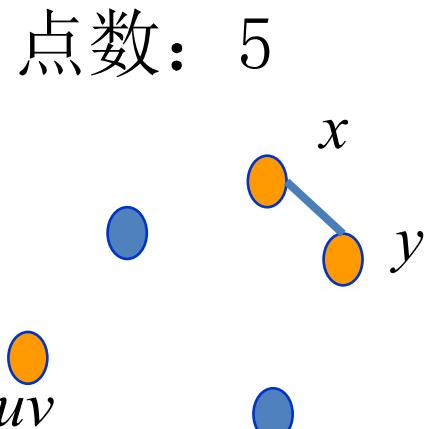
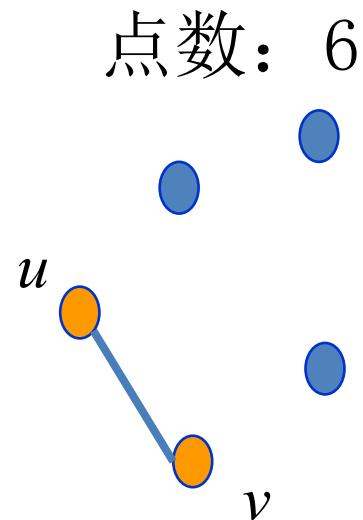
点数: 5



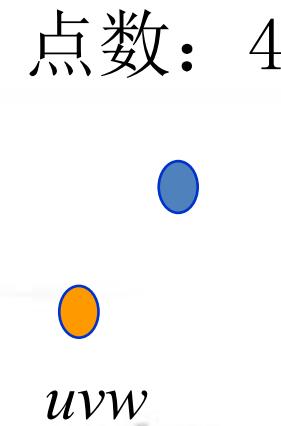
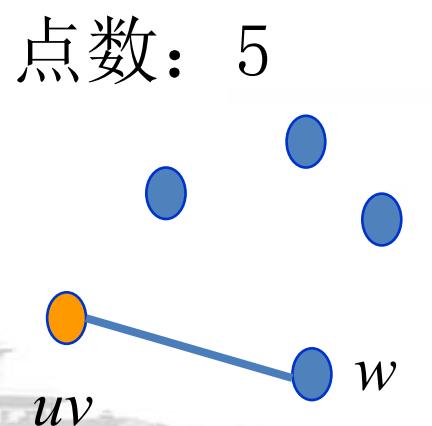
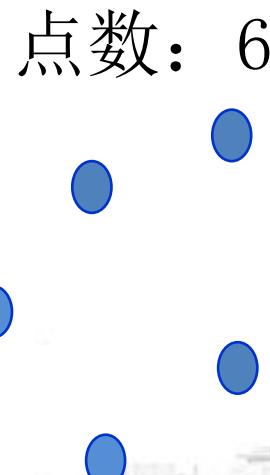
点数: 4



子問題划分



加边法



加点法

Kruskal算法

克鲁斯卡尔算法的基本思想：

设 $G=(V,E)$ 是连通网，用 T 来记录 G 上最小生成树边的集合。

- (1)从 G 中取最短边 e ，如果边 e 所关联的两个顶点不在 T 的同一个连通分量中，则将该边加入 T ；
- (2)从 G 中删除边 e ；
- (3)重复(1)和(2)两个步骤，直到 T 中有 $n-1$ 条边。

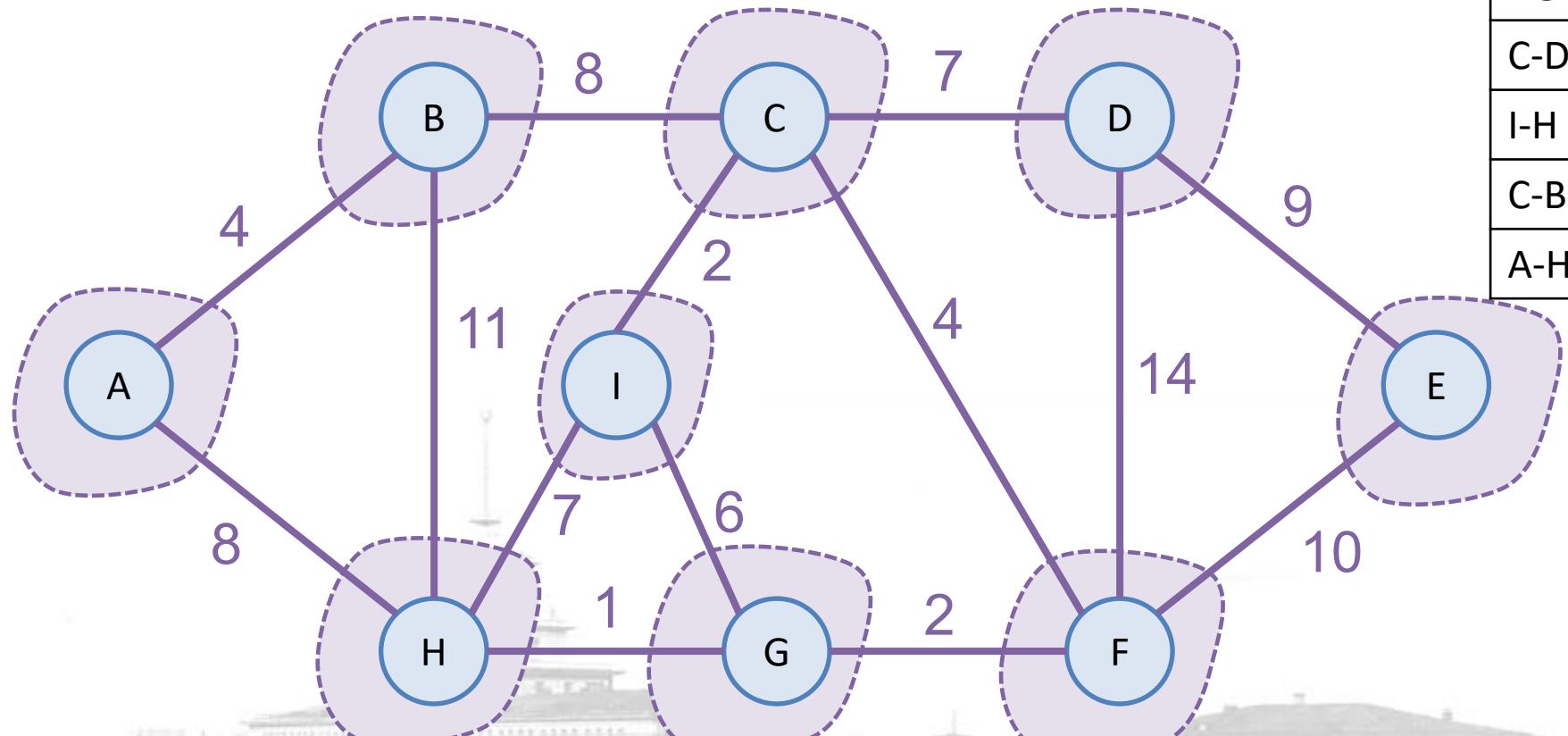
Kruskal算法

Kruskal算法是从森林到树：

每一步都通过合并规模小的树，减小森林的规模，得到更大的树

开始时，每个顶点都在自己的树中

H-G	1
I-C	2
G-F	2
A-B	4
C-F	4
I-G	6
C-D	7
I-H	7
C-B	8
A-H	8

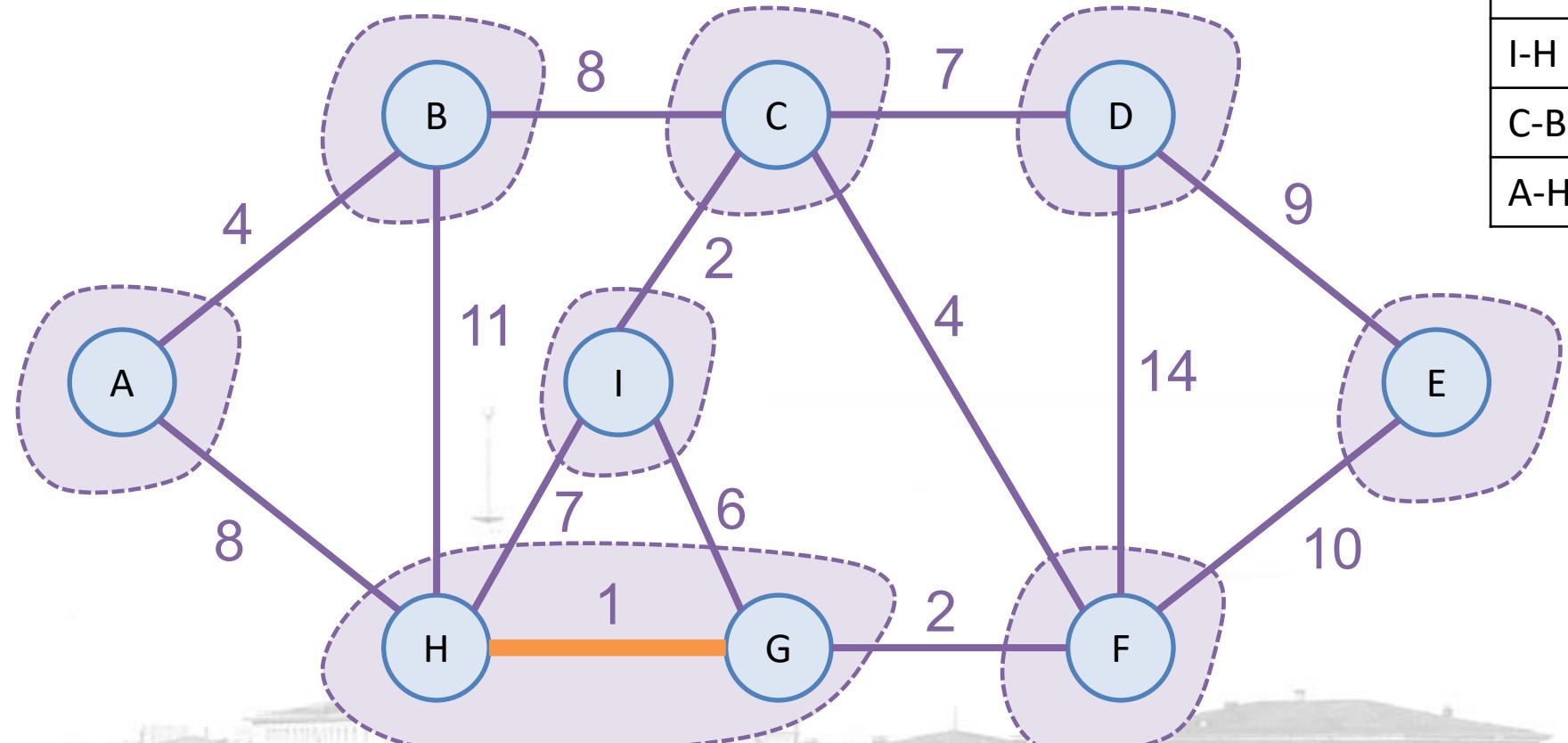


Kruskal算法

Kruskal算法是从森林到树：

每一步都通过合并规模小的树，减小森林的规模，得到更大的树

I-C	2
G-F	2
A-B	4
C-F	4
I-G	6
C-D	7
I-H	7
C-B	8
A-H	8

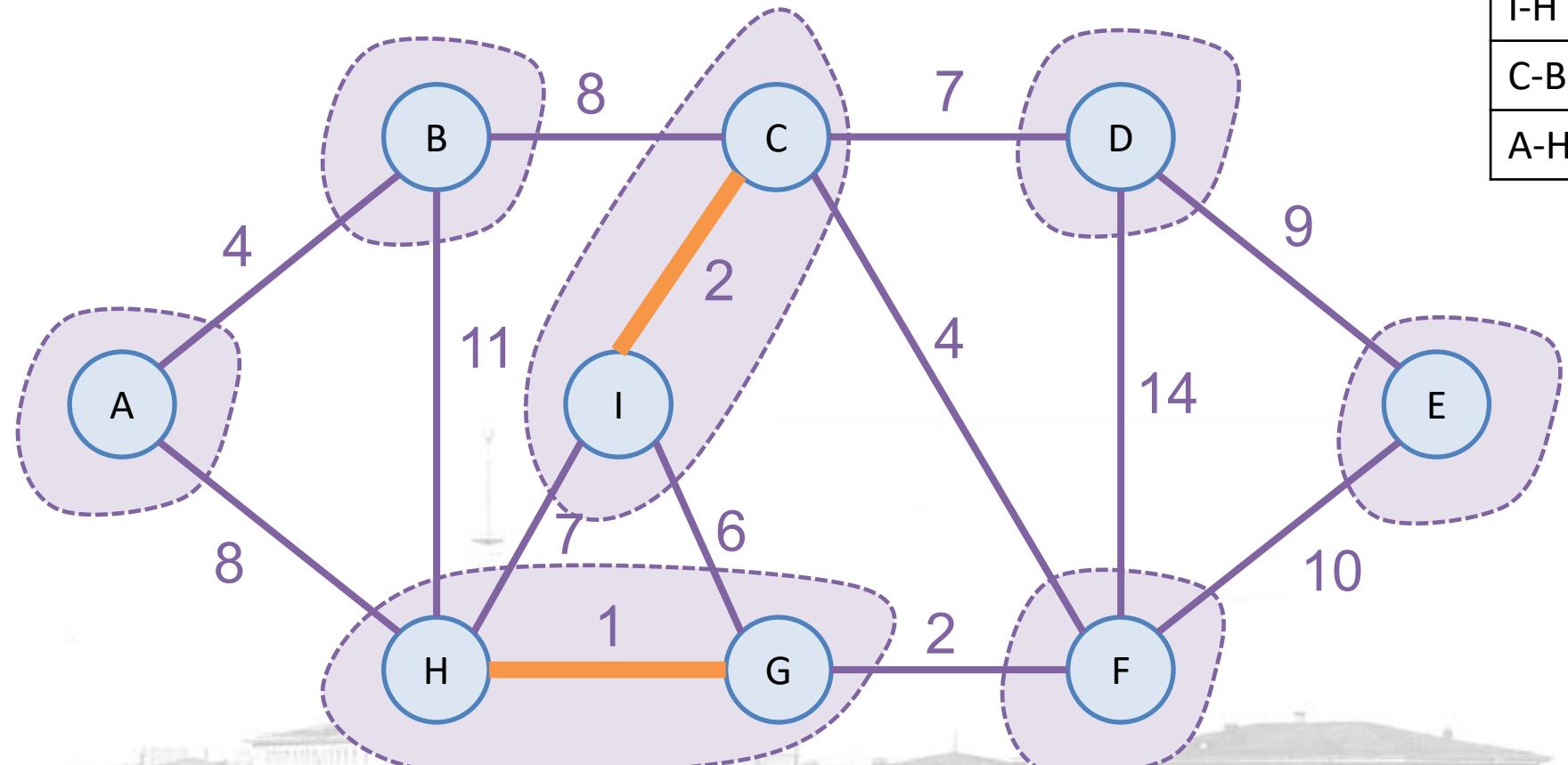


Kruskal算法

Kruskal算法是从森林到树：

每一步都通过合并规模小的树，减小森林的规模，得到更大的树

G-F	2
A-B	4
C-F	4
I-G	6
C-D	7
I-H	7
C-B	8
A-H	8

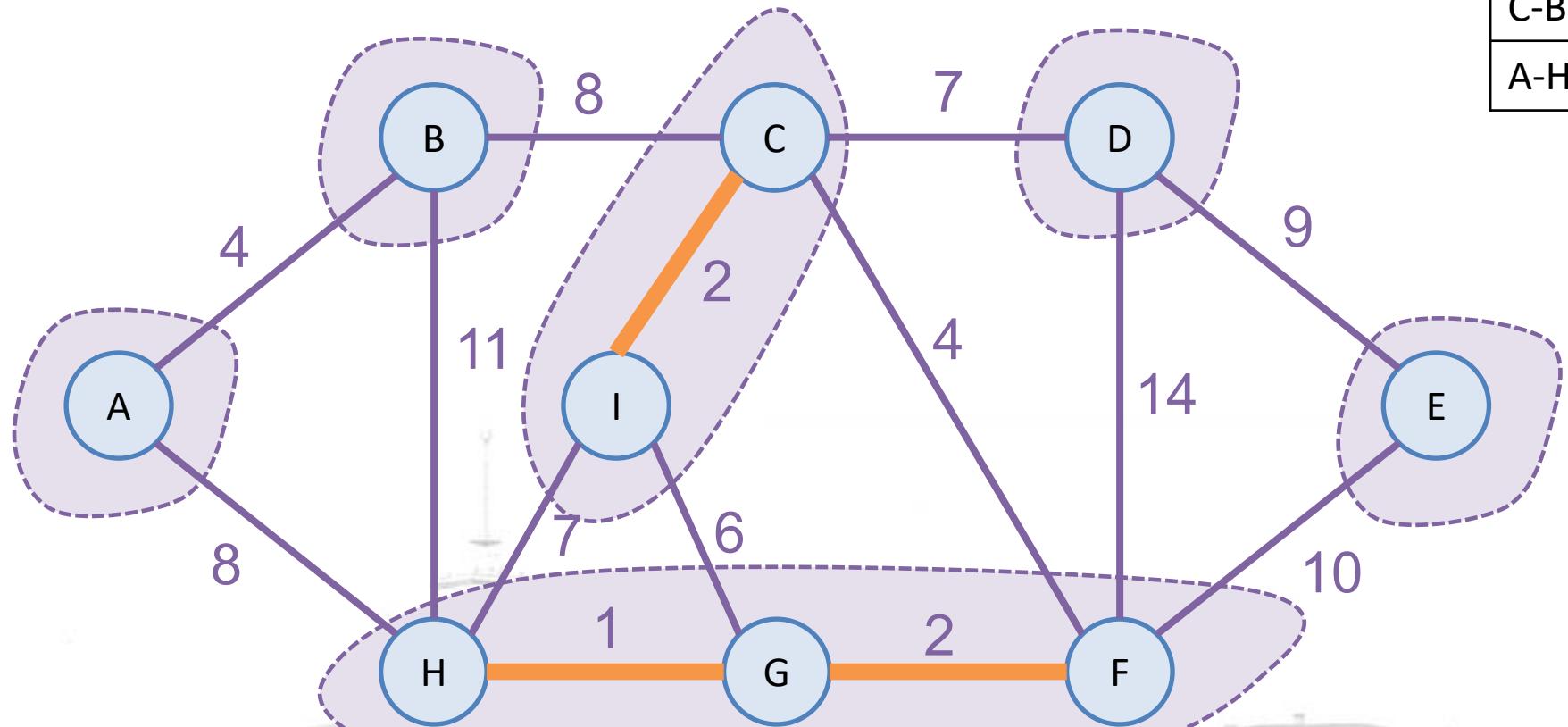


Kruskal算法

Kruskal算法是从森林到树：

每一步都通过合并规模小的树，减小森林的规模，得到更大的树

A-B	4
C-F	4
I-G	6
C-D	7
I-H	7
C-B	8
A-H	8

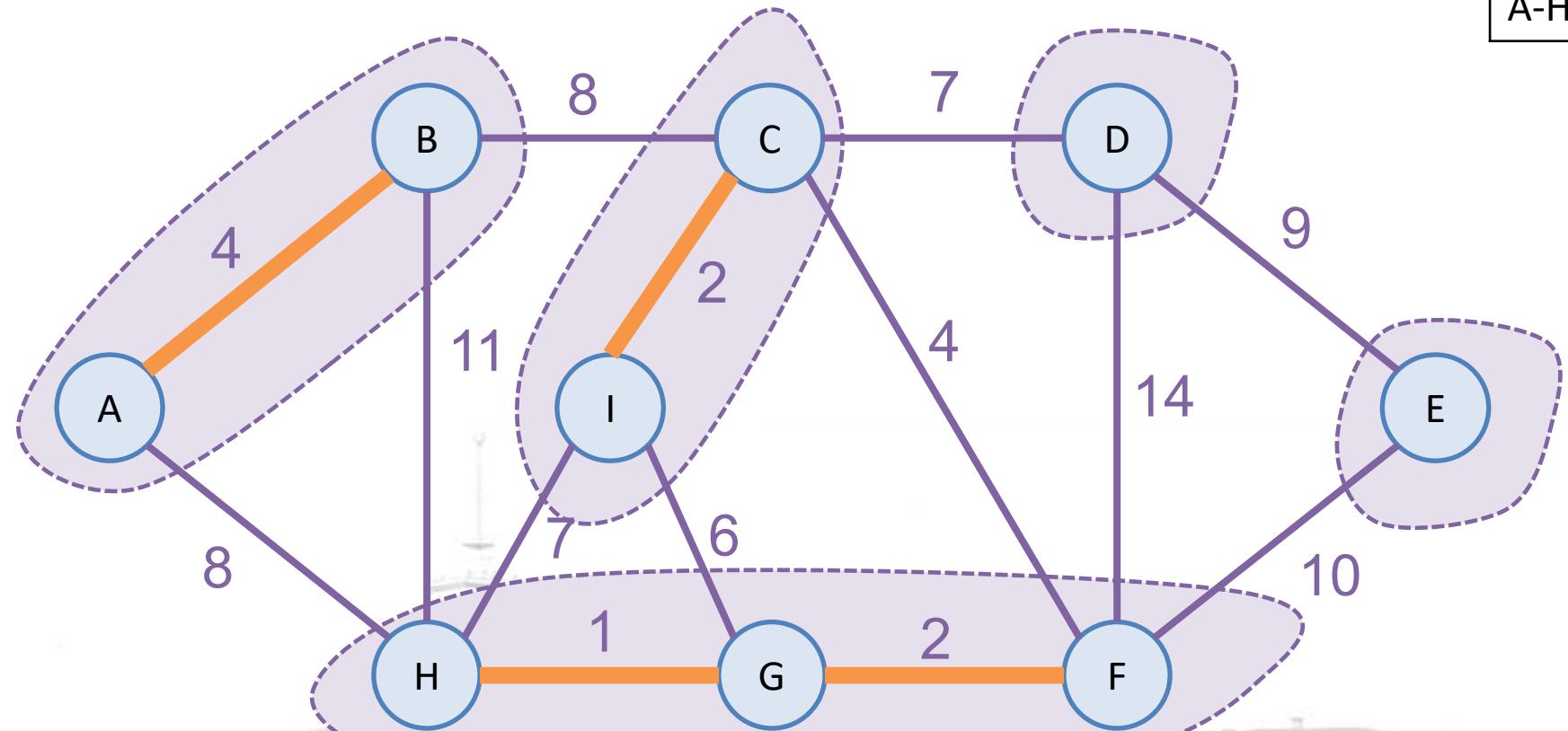


Kruskal算法

Kruskal算法是从森林到树：

每一步都通过合并规模小的树，减小森林的规模，得到更大的树

C-F	4
I-G	6
C-D	7
I-H	7
C-B	8
A-H	8

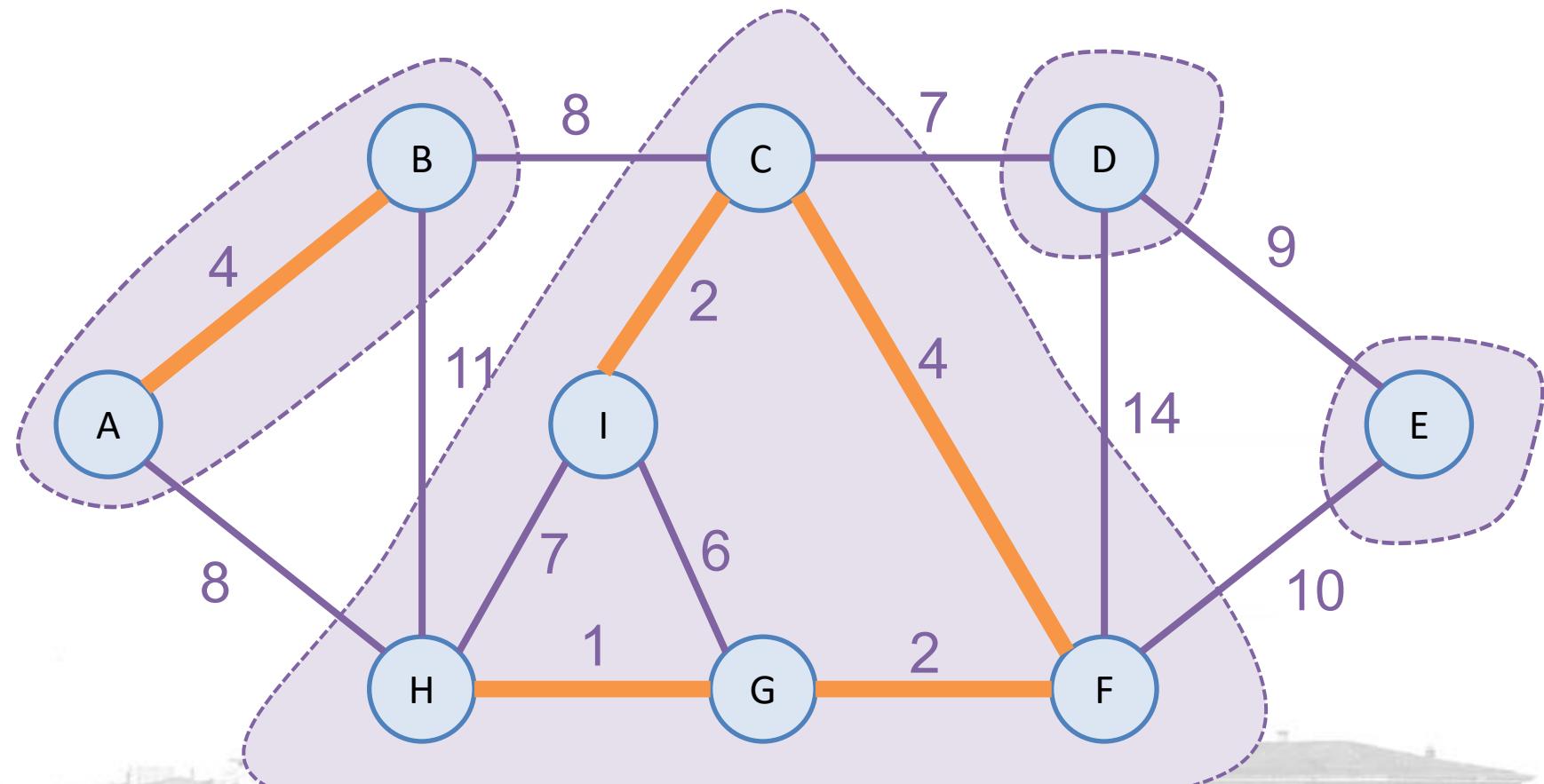


I-G	6
C-D	7
I-H	7
C-B	8
A-H	8

Kruskal算法

Kruskal算法是从森林到树：

每一步都通过合并规模小的树，减小森林的规模，得到更大的树

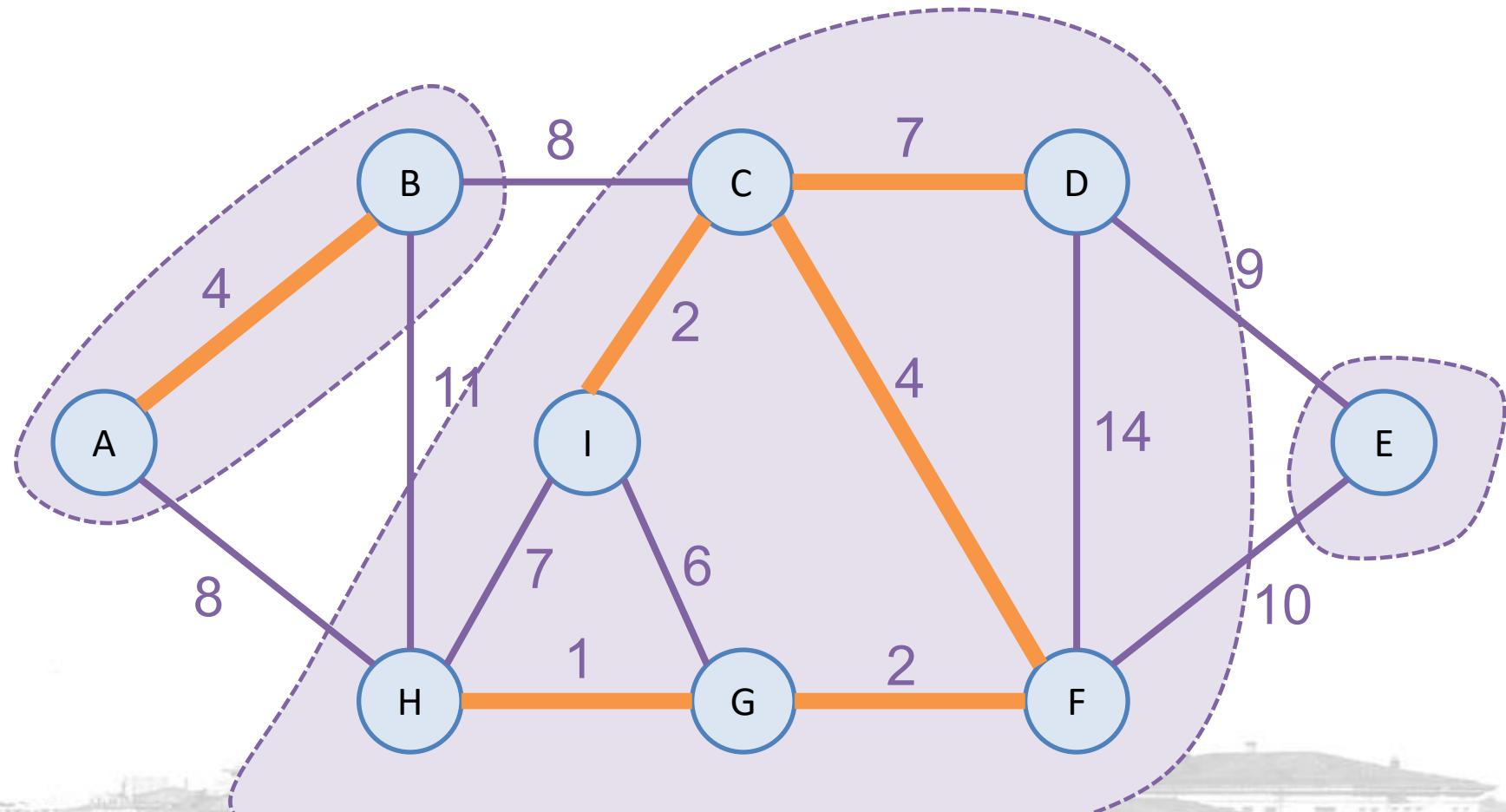


H-H	7
C-B	8
A-H	8
D-E	9

Kruskal算法

Kruskal算法是从森林到树：

每一步都通过合并规模小的树，减小森林的规模，得到更大的树

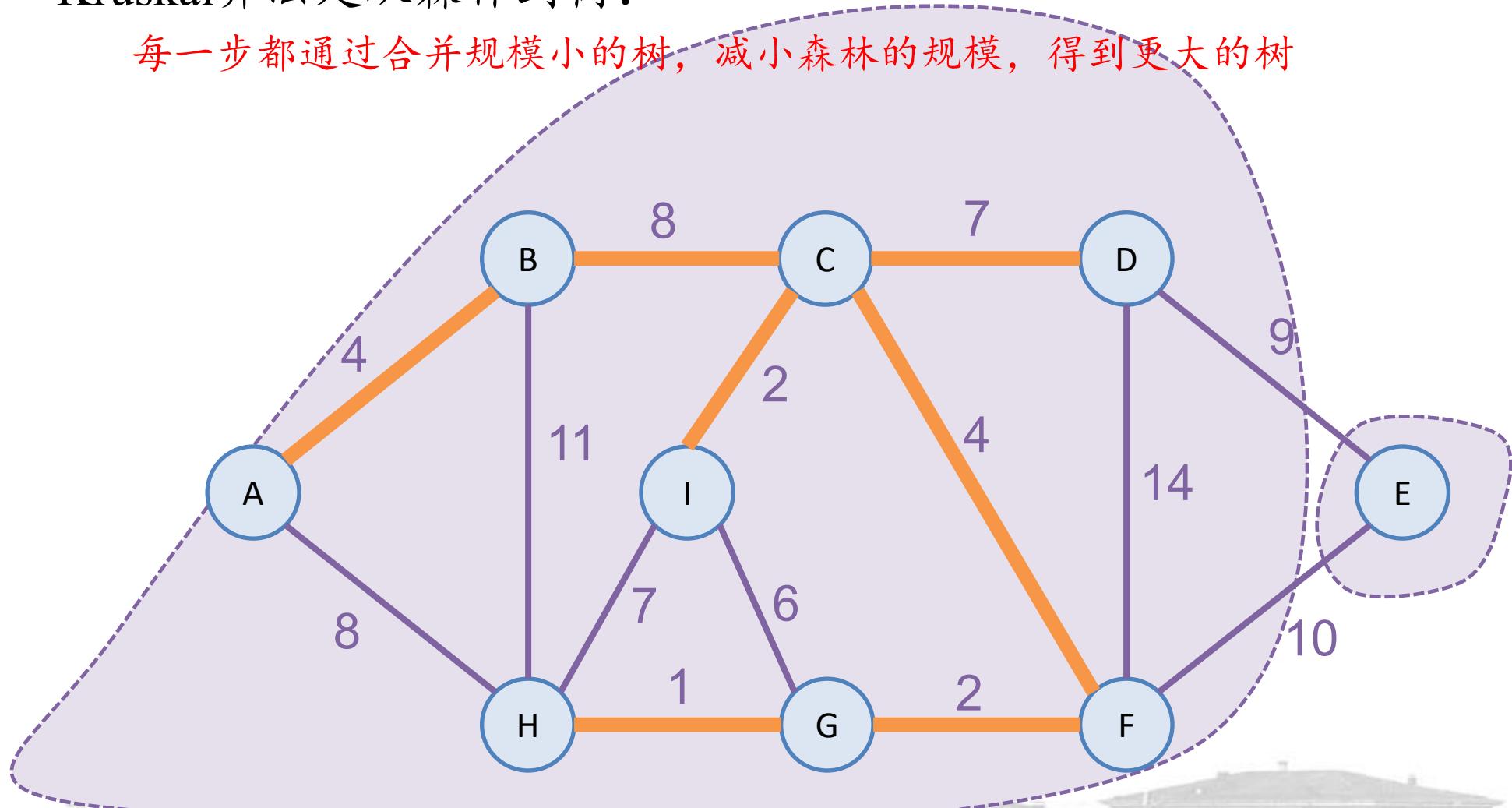


A-H	8
D-E	9

Kruskal算法

Kruskal算法是从森林到树：

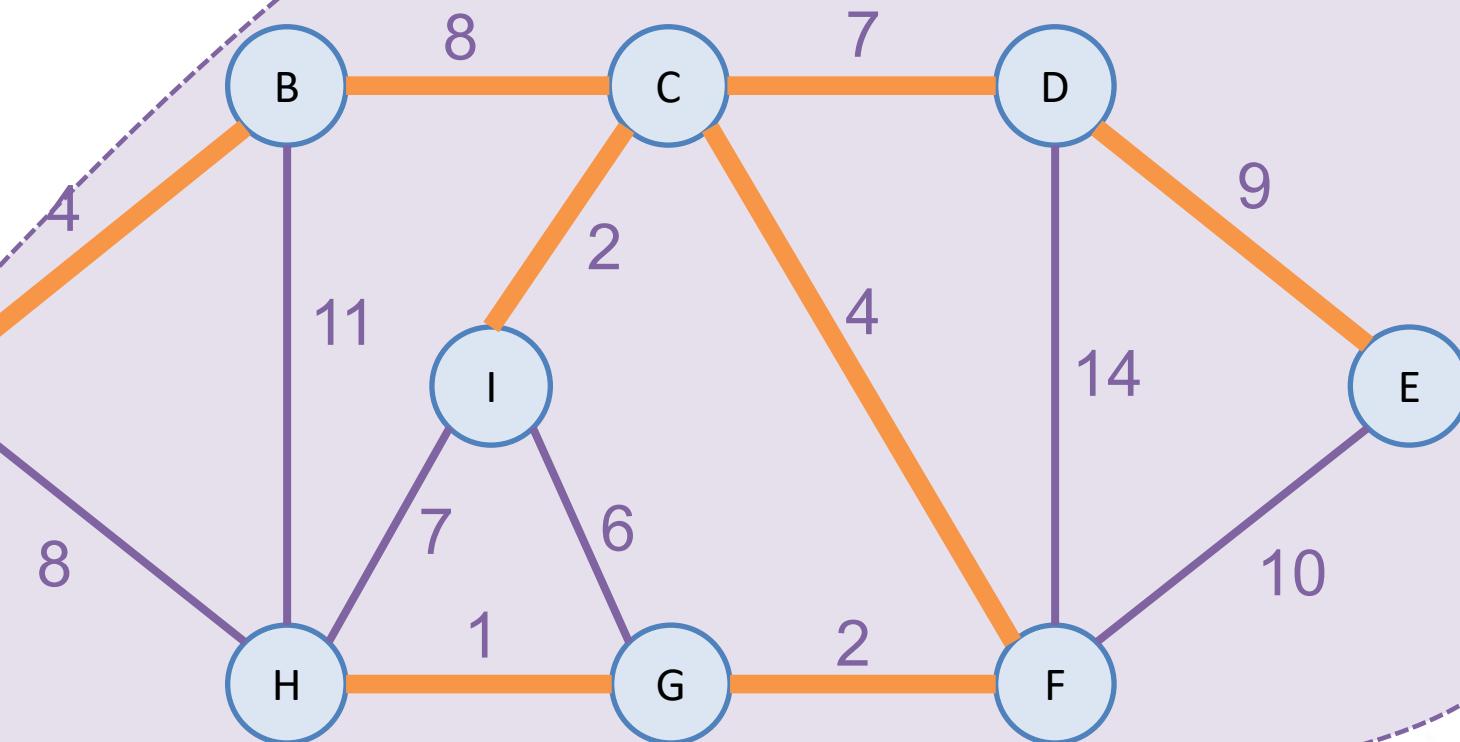
每一步都通过合并规模小的树，减小森林的规模，得到更大的树



Kruskal算法

Kruskal算法是从森林到树：

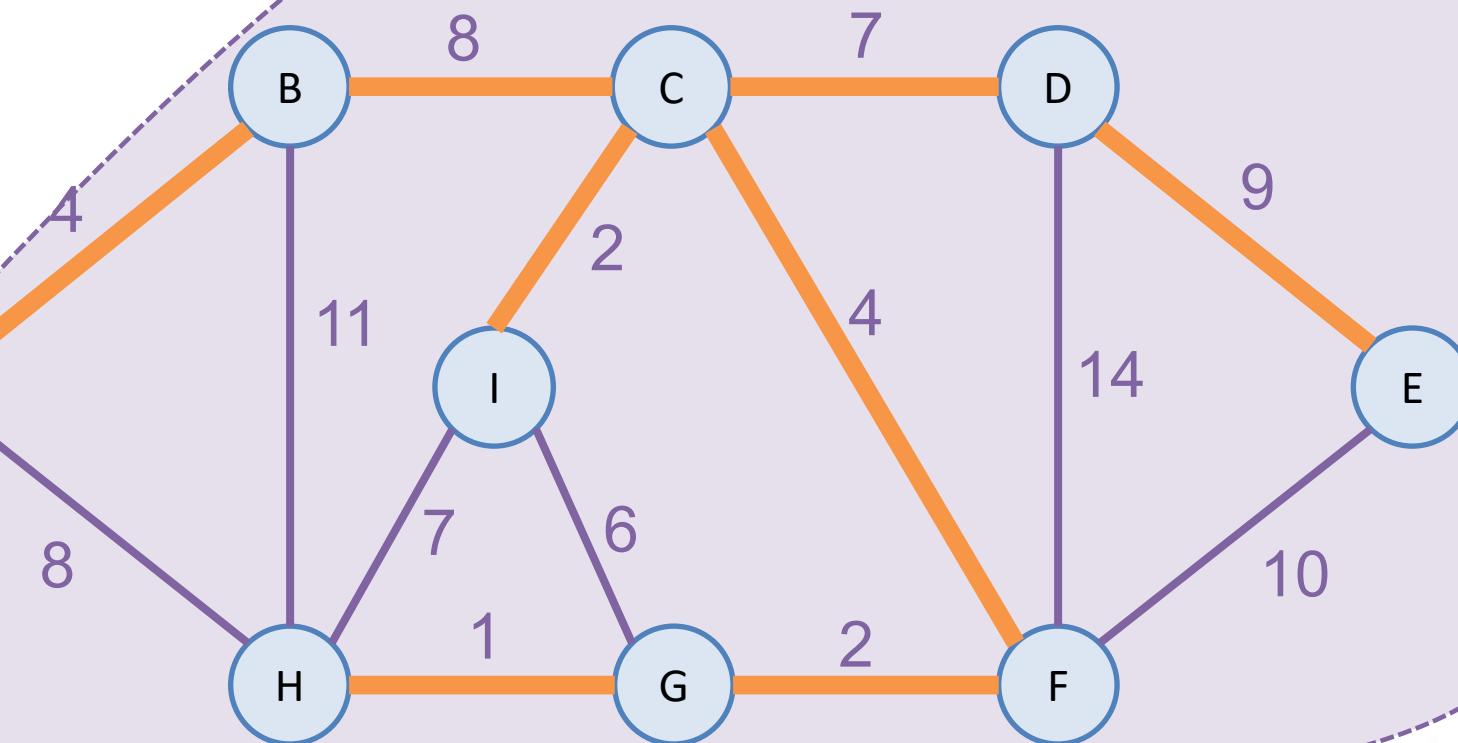
每一步都通过合并规模小的树，减小森林的规模，得到更大的树



Kruskal算法

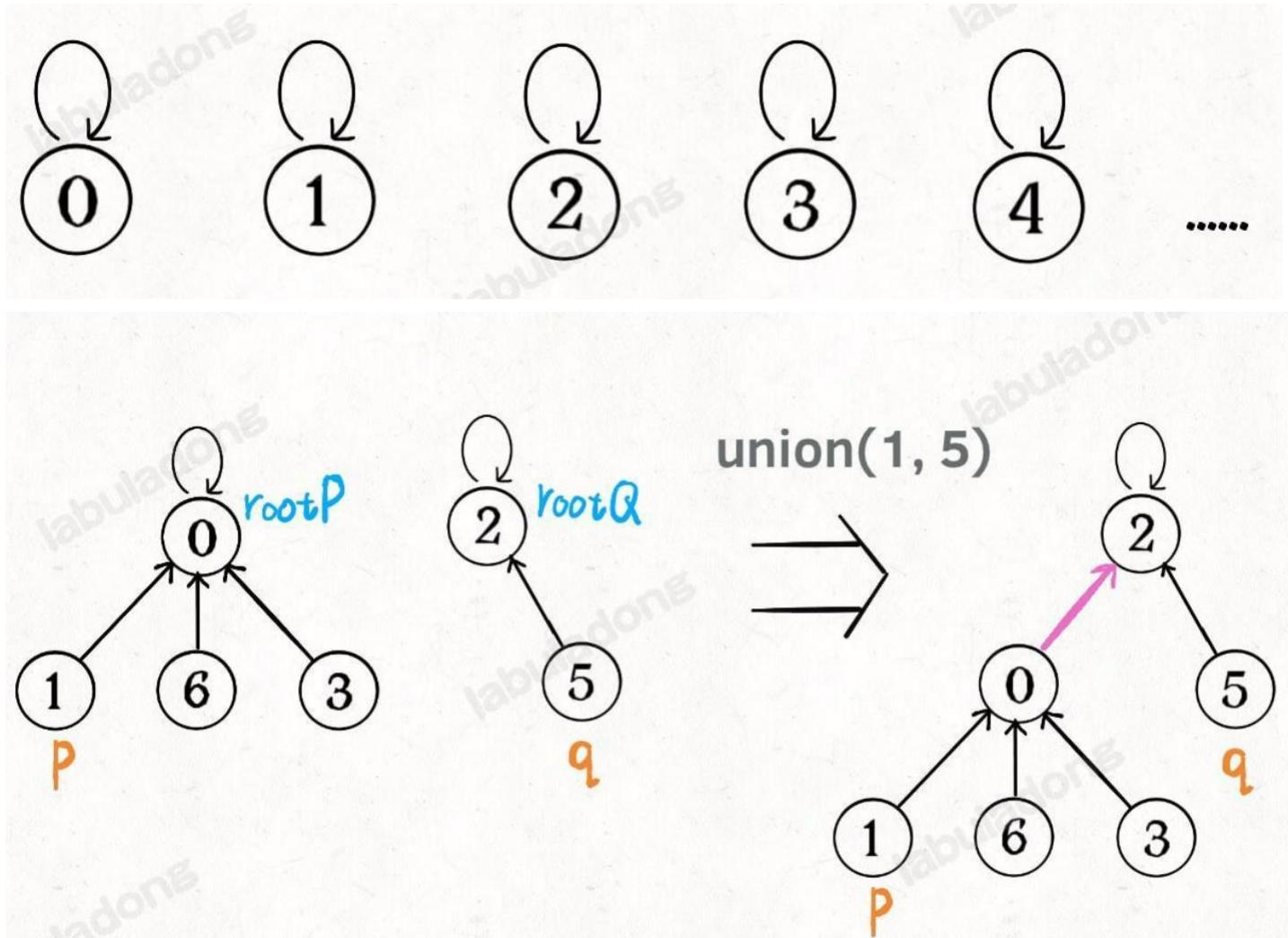
Kruskal算法是从森林到树：

每一步都通过合并规模小的树，减小森林的规模，得到更大的树



当只有一棵树的时候，停止！

并查集



Kruskal算法

MST-Kruskal(G, W)

Input: 无向连通图 $G = (V, E)$, 权值函数 W

Output: G 的最小生成树

1. $S = \emptyset$; /*初始化为空树*/
2. FOR $\forall v \in V$ DO
 3. Make-Set(v); /*创建|V|棵树*/
4. 按照 W 权值的非递减顺序排序 E ;
5. FOR $\forall (u, v) \in E$ (按照 W 权值的非递减顺序) DO
 6. IF Find-Set(u) ≠ Find-Set(v)
 /*检查节点 u, v 是否属于同一棵树 */
 7. $S = S \cup \{(u, v)\}$; /*边 (u, v) 加入到集合 S */
 8. Union(u, v); /*对节点 u, v 所属的子树进行合并*/
9. Return S.

算法复杂性

- 令 $n=|\mathcal{V}|, m=|\mathcal{E}|$



算法复杂性

- 令 $n=|V|, m=|E|$
- 第4步需要时间: $O(m \log m)$



算法复杂性

- 令 $n=|V|, m=|E|$
- 第4步需要时间: $O(m \log m)$
- 第2-3步执行 $O(n)$ 个*Make-Set*操作



算法复杂性

- 令 $n=|V|, m=|E|$
- 第4步需要时间: $O(m \log m)$
- 第2-3步执行 $O(n)$ 个*Make-Set*操作
第5-8步执行 $O(m)$ 个*Find-Set*和 $O(n)$ 个*Union*操作



算法复杂性

- 令 $n=|V|, m=|E|$
- 第4步需要时间: $O(m \log m)$
- 第2-3步执行 $O(n)$ 个*Make-Set*操作
第5-8步执行 $O(m)$ 个*Find-Set*和 $O(n)$ 个*Union*操作
需要时间: $O((n+m)\alpha(n))$ // 阿克曼反函数

算法复杂性

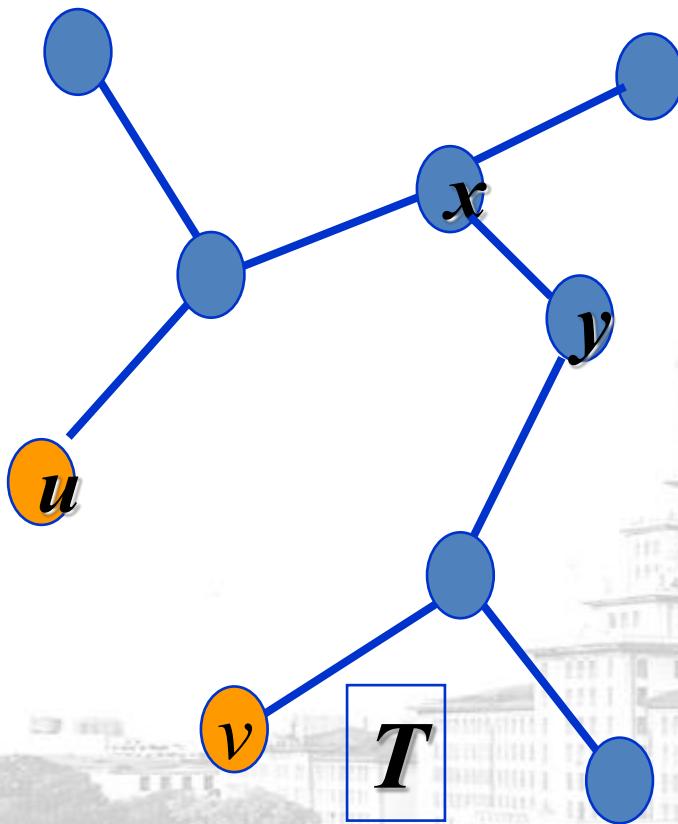
- 令 $n=|V|, m=|E|$
- 第4步需要时间: $O(m \log m)$
- 第2-3步执行 $O(n)$ 个*Make-Set*操作
第5-8步执行 $O(m)$ 个*Find-Set*和 $O(n)$ 个*Union*操作
需要时间: $O((n+m)\alpha(n))$ // 阿克曼反函数
- $m \geq n-1$ (因为G连通), $\alpha(n) \leq \log n \leq \log m$

算法复杂性

- 令 $n=|V|, m=|E|$
- 第4步需要时间: $O(m \log m)$
- 第2-3步执行 $O(n)$ 个*Make-Set*操作
第5-8步执行 $O(m)$ 个*Find-Set*和 $O(n)$ 个*Union*操作
需要时间: $O((n+m)\alpha(n))$ // 阿克曼反函数
- $m \geq n-1$ (因为G连通), $\alpha(n) \leq \log n \leq \log m$
- 总时间复杂性: $O(m \log m)$

贪心选择性

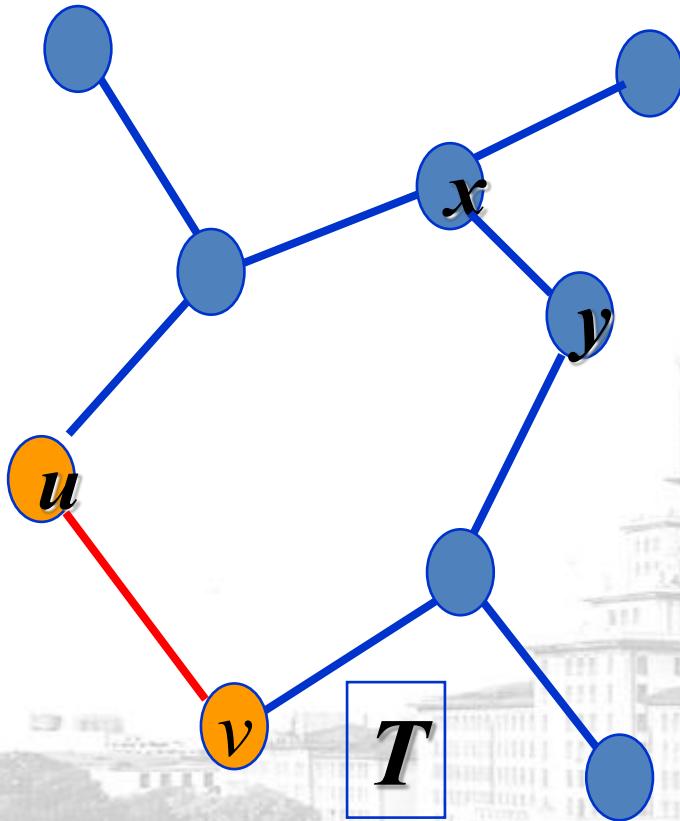
定理1. 设 uv 是 G 中权值最小的边，则必有一棵最小生成树包含边 uv .



证明：设 T 是 G 的一棵MST
若 $uv \in T$, 结论成立;
否则, 如右图所示

贪心选择性

定理1. 设 uv 是 G 中权值最小的边，则必有一棵最小生成树包含边 uv .

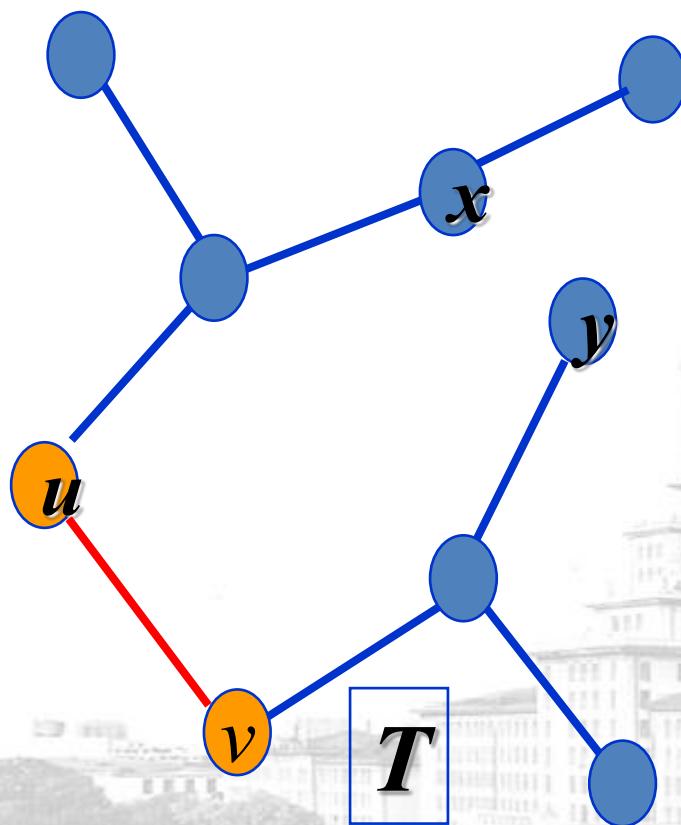


证明：设 T 是 G 的一棵MST
若 $uv \in T$, 结论成立;
否则, 如右图所示

贪心选择性

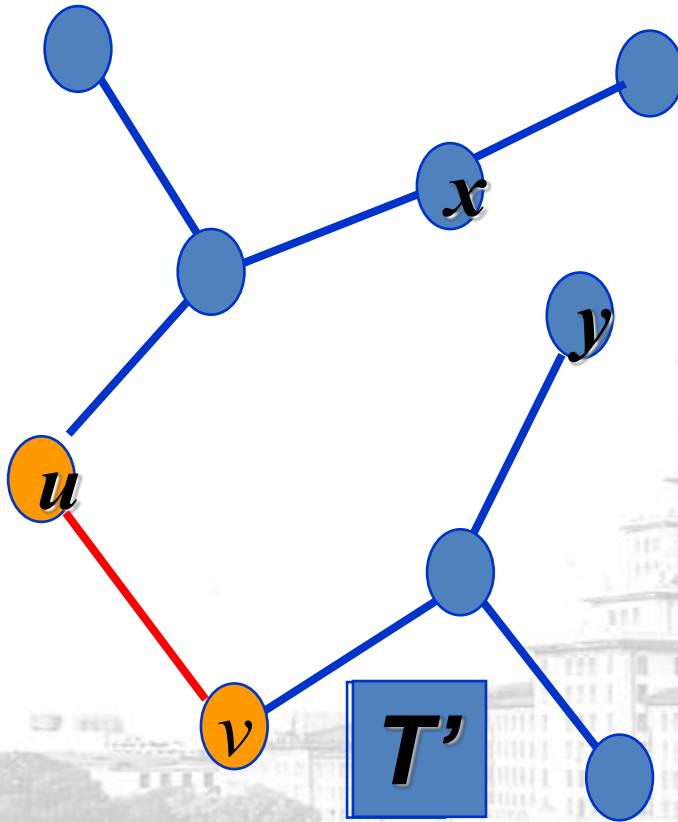
定理1. 设 uv 是 G 中权值最小的边，则必有一棵最小生成树包含边 uv .

证明：设 T 是 G 的一棵 MST
若 $uv \in T$, 结论成立;
否则, 如右图所示



贪心选择性

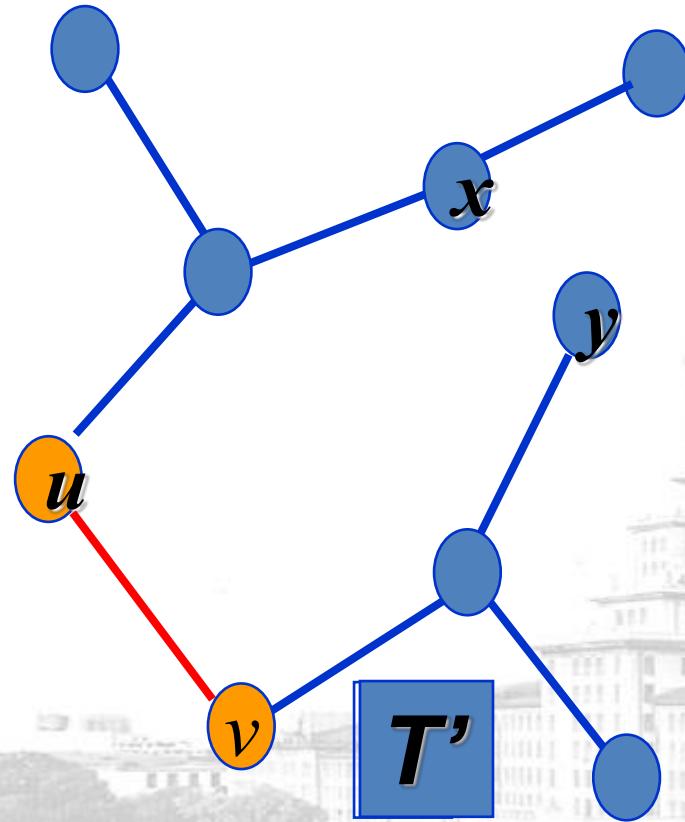
定理1. 设 uv 是 G 中权值最小的边，则必有一棵最小生成树包含边 uv .



证明：设 T 是 G 的一棵MST
若 $uv \in T$, 结论成立；
否则，如右图所示

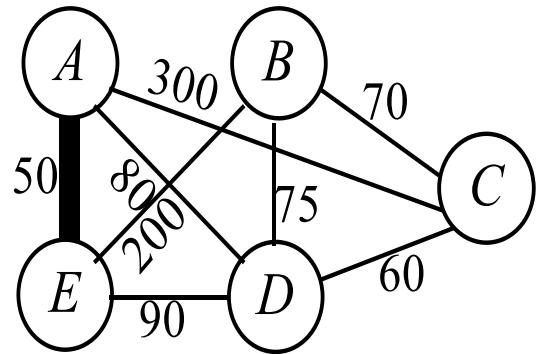
贪心选择性

定理1. 设 uv 是 G 中权值最小的边，则必有一棵最小生成树包含边 uv .

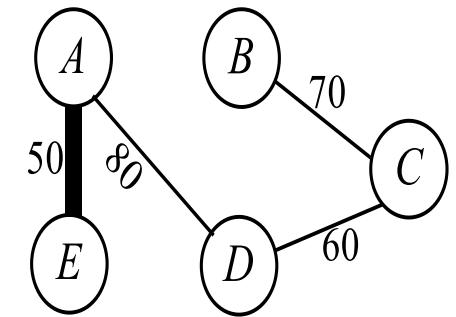
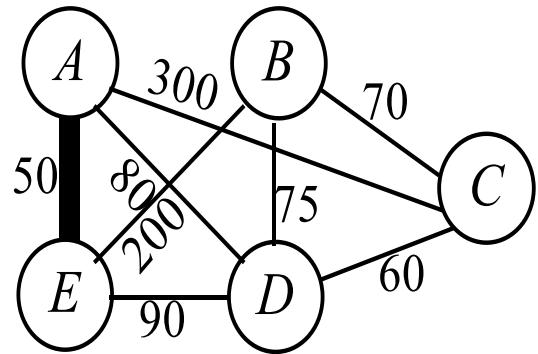


证明：设 T 是 G 的一棵 **MST**
若 $uv \in T$, 结论成立；
否则，如右图所示
在 T 中添加 uv 边，产生环
删除环中不同于 uv 的权值最
小的边 xy ，得到 T' 。
 $w(T') = w(T) - w(xy) + w(uv)$
 $\leq w(T)$

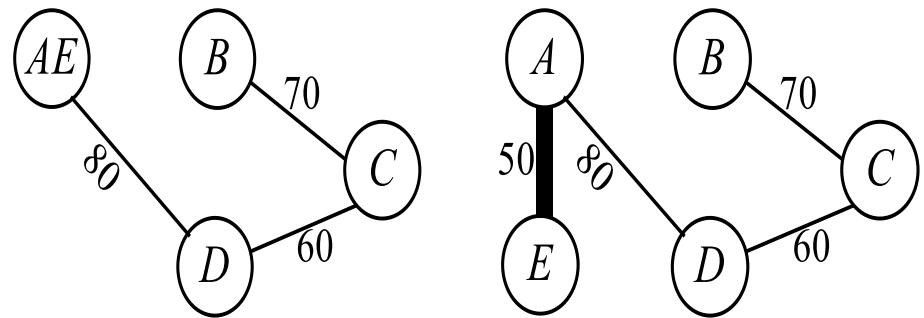
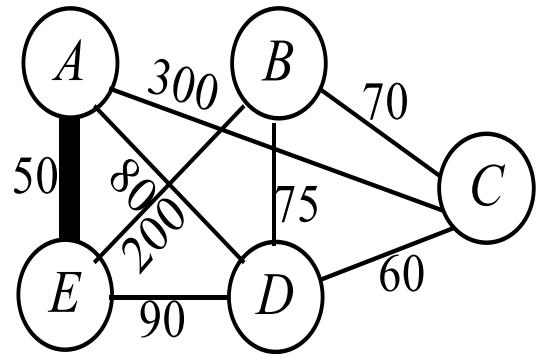
优化子结构



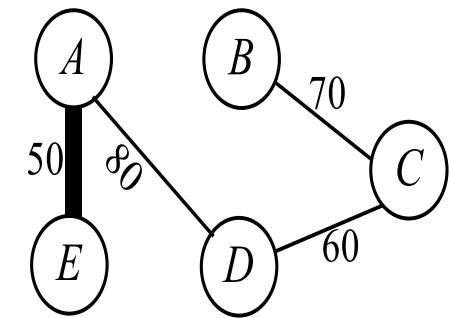
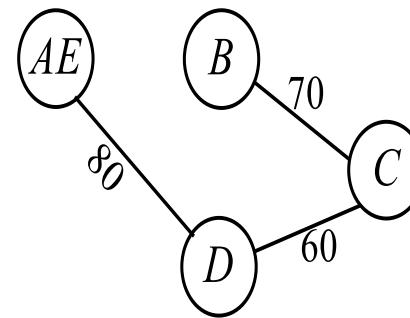
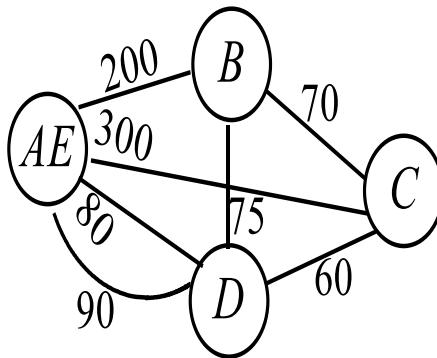
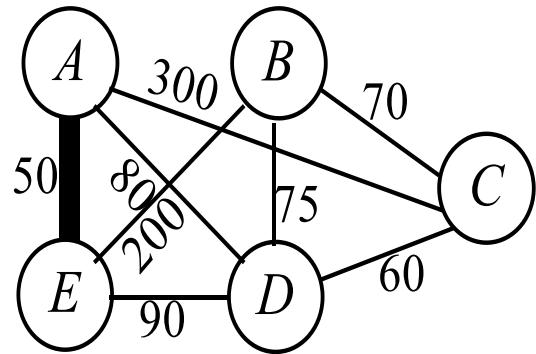
优化子结构



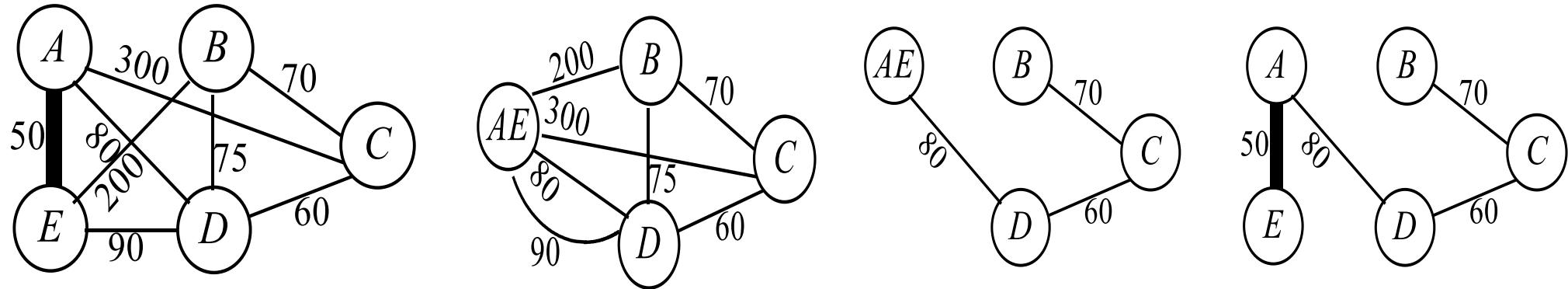
优化子结构



优化子结构



优化子结构

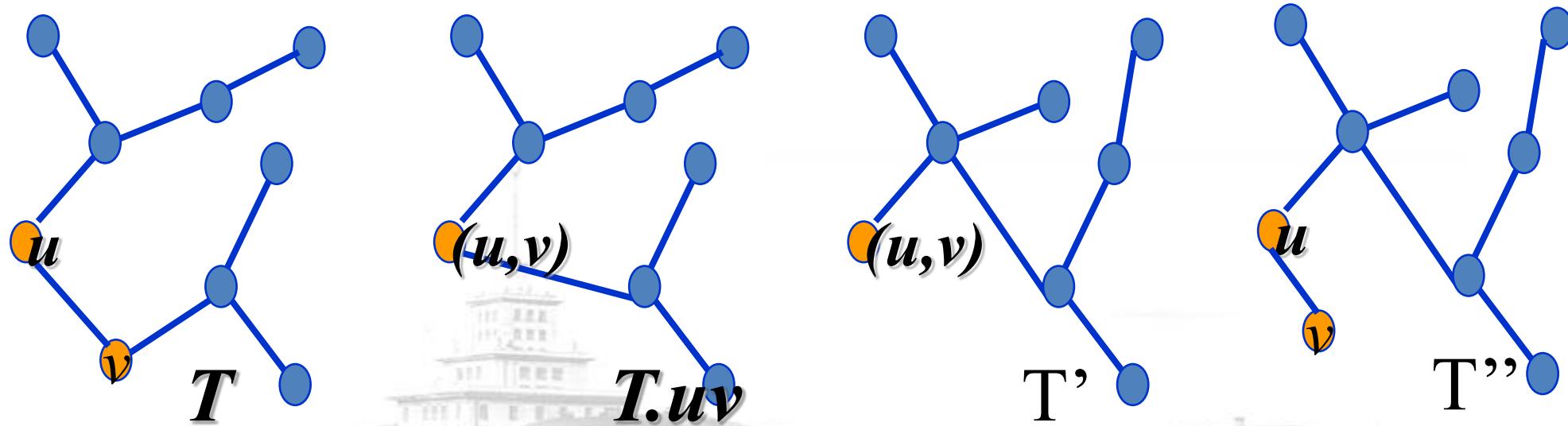


收缩图 G 的边 uv — $G \bullet uv$

- 用新顶点 C_{uv} 代替边 uv
- 将 G 中原来与 u 或 v 关联的边与 C_{uv} 关联
- 删除 C_{uv} 到其自身的边

上述操作的逆操作称为扩张

定理2.给定加权无向连通图 $G=(V,E)$,权值函数为 $W:E\rightarrow R$, $uv\in E$ 是 G 中权值最小的边。设 T 是 G 的包含 uv 的一棵最小生成树，则 $T\cdot uv$ 是 $G\cdot uv$ 的一棵最小生成树。



定理2.给定加权无向连通图 $G=(V,E)$,权值函数为 $W:E\rightarrow R$, $uv\in E$ 是 G 中权值最小的边。设 T 是 G 的包含 uv 的一棵最小生成树，则 $T\cdot uv$ 是 $G\cdot uv$ 的一棵最小生成树。

证明. 由于 $T\cdot uv$ 是不含回路的连通图且包含了 $G\cdot uv$ 的所有顶点，因此， $T\cdot uv$ 是 $G\cdot uv$ 的一棵生成树。下面证明 $T\cdot uv$ 是 $G\cdot uv$ 的代价最小的生成树。

若不然，存在 $G\cdot uv$ 的生成树 T' 使得 $W(T') < W(T\cdot uv)$ 。显然， T' 中包含顶点 C_{uv} 且是连通的，因此 $T''=T'\cup C_{uv}$ 包含 G 的所有顶点且不含回路，故 T'' 是 G 的一棵生成树。但， $W(T'') = W(T')+W(uv) < W(T\cdot uv)+W(uv)=W(T)$,这与 T 是 G 的最小生成树矛盾。

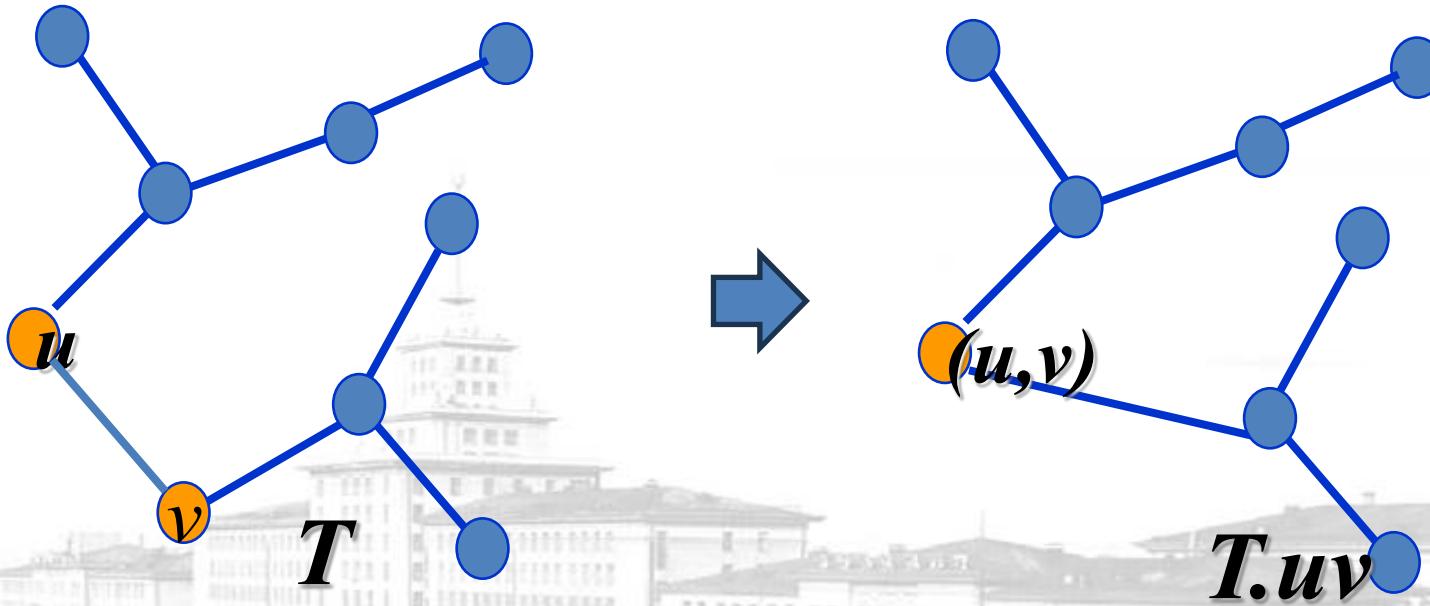
算法正确性

定理3. MST-Kruskal(G, W)算法能够产生图 G 的最小生成树.

证. 因为算法按照贪心选择性进行局部优化选择.

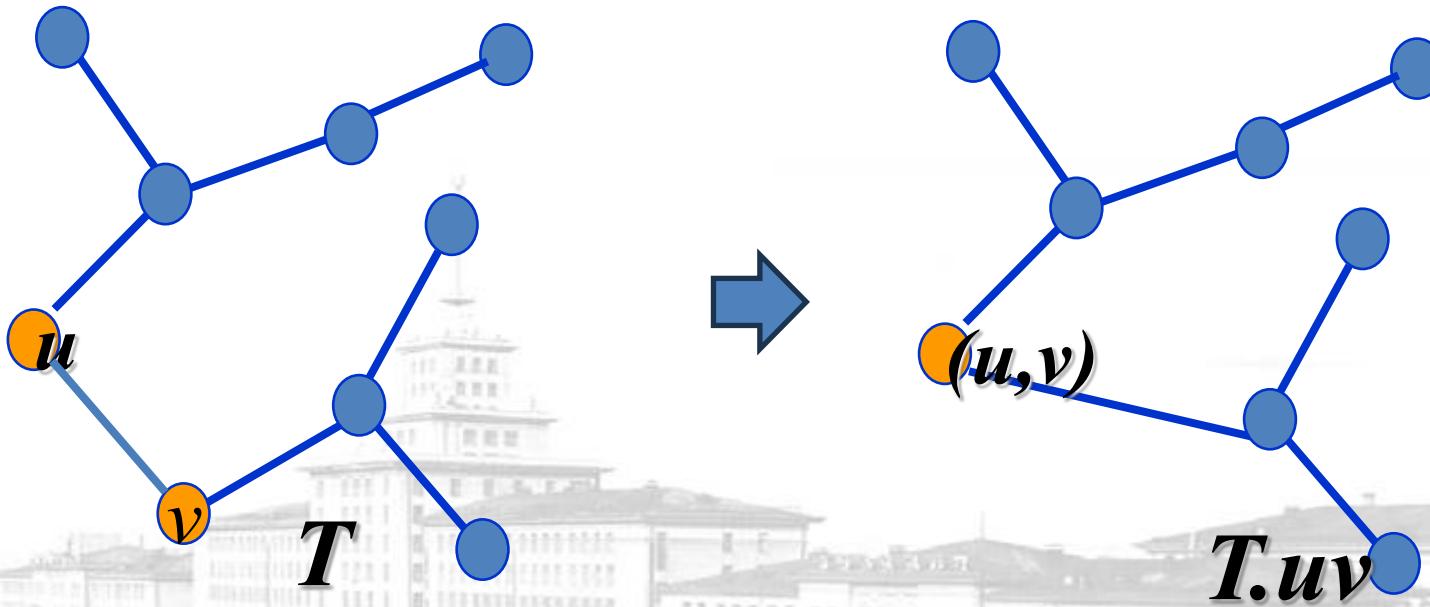


证明： n 表示图G中顶点集合V中的点的个数



证明：n表示图G中顶点集合V中的点的个数

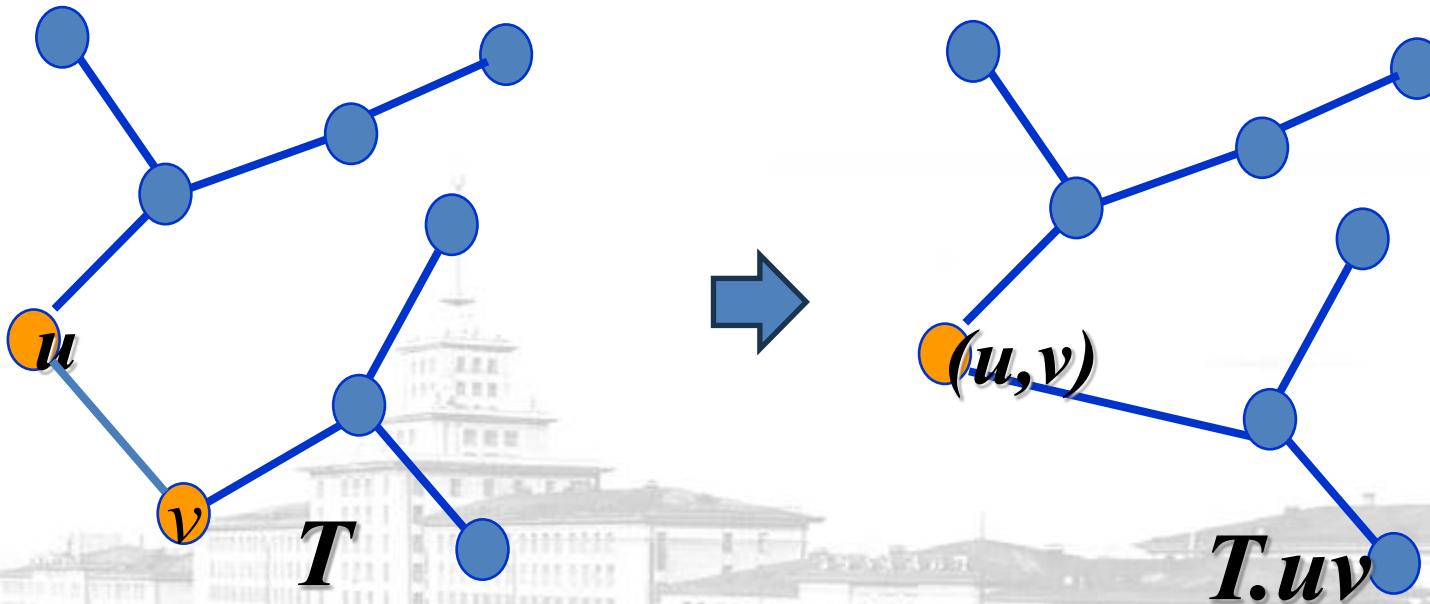
1、n=2时，只有两个点。生成树只有一种，根据kruskal算法产生的生成树必定是最小生成树。



证明：n表示图G中顶点集合V中的点的个数

1、 $n=2$ 时，只有两个点。生成树只有一种，根据kruskal算法产生的生成树必定是最小生成树。

2、假设 $n < k$ 时，通过kruskal算法得到的生成树是最小生成树。

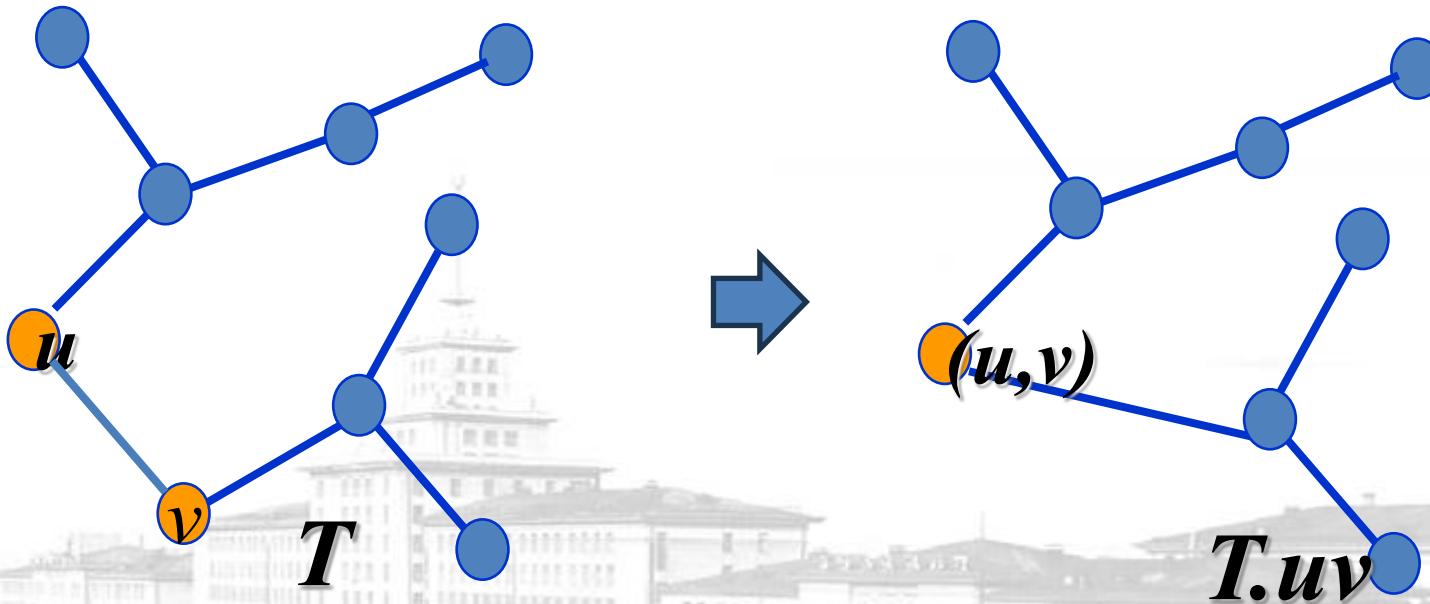


证明：n表示图G中顶点集合V中的点的个数

1、 $n=2$ 时，只有两个点。生成树只有一种，根据kruskal算法产生的生成树必定是最小生成树。

2、假设 $n < k$ 时，通过kruskal算法得到的生成树是最小生成树。

当 $n=k$ 时，考虑如下情况。不失一般性，设 (u,v) 是图G中最短的边。那么根据贪心选择性，存在一颗G的最小生成树T包含边 (u,v) 。



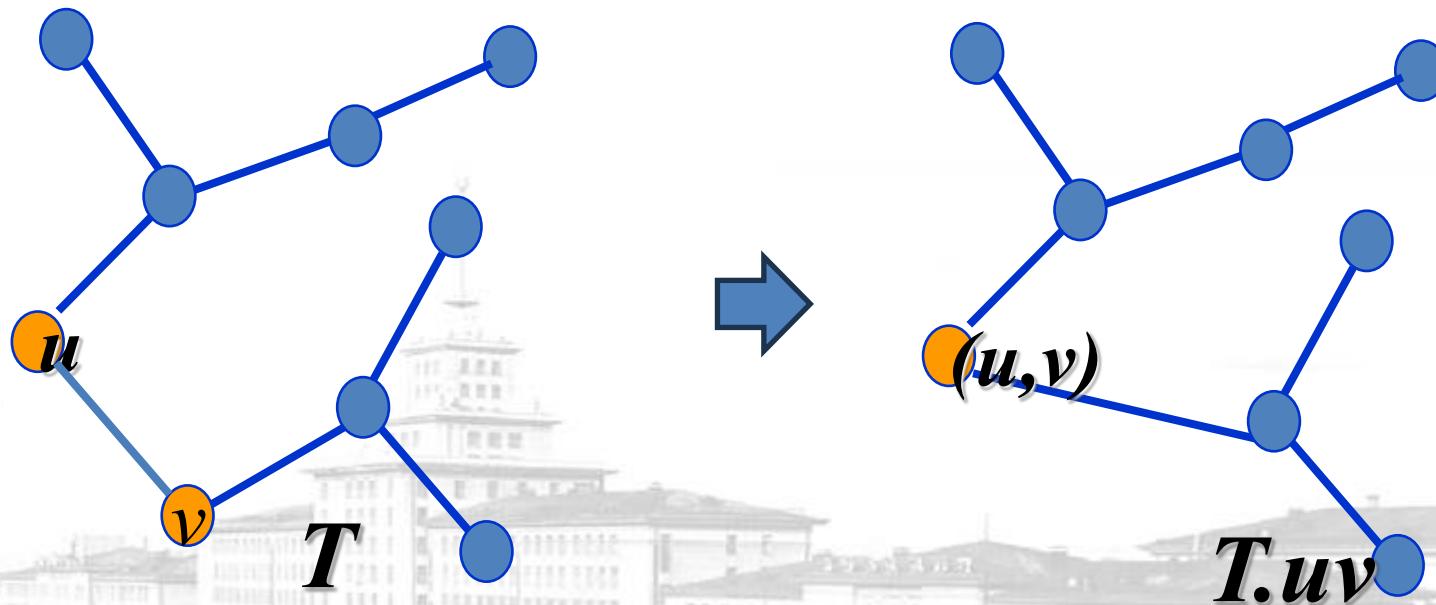
证明：n表示图G中顶点集合V中的点的个数

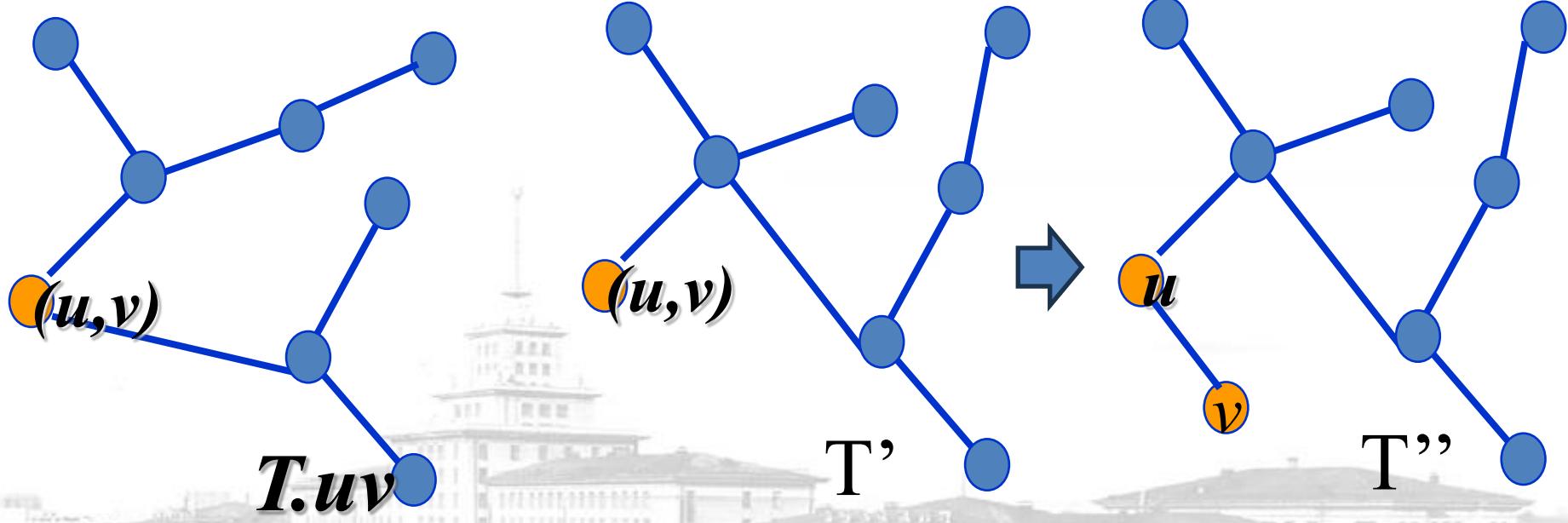
1、 $n=2$ 时，只有两个点。生成树只有一种，根据kruskal算法产生的生成树必定是最小生成树。

2、假设 $n < k$ 时，通过kruskal算法得到的生成树是最小生成树。

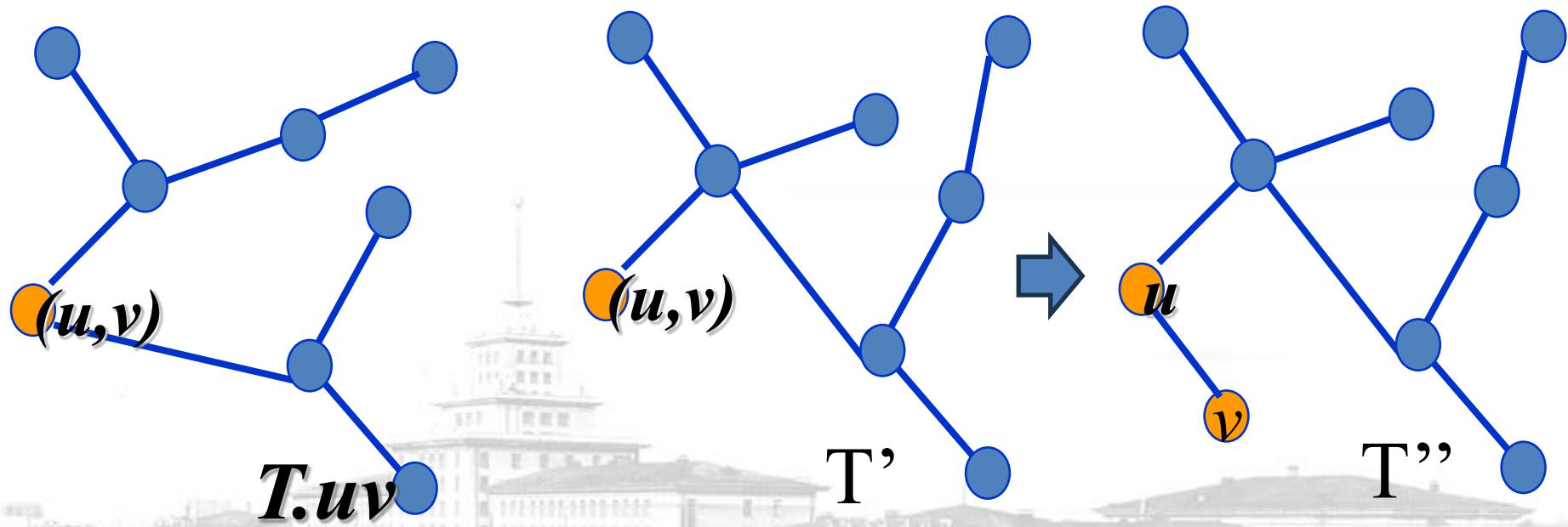
当 $n=k$ 时，考虑如下情况。不失一般性，设 (u,v) 是图G中最短的边。那么根据贪心选择性，存在一颗G的最小生成树T包含边 (u,v) 。

(1) 收缩 (u,v) ，得到树 T_{uv} 和图 G_{uv} 。根据优化子结构， T_{uv} 是 G_{uv} 的最小生成树。



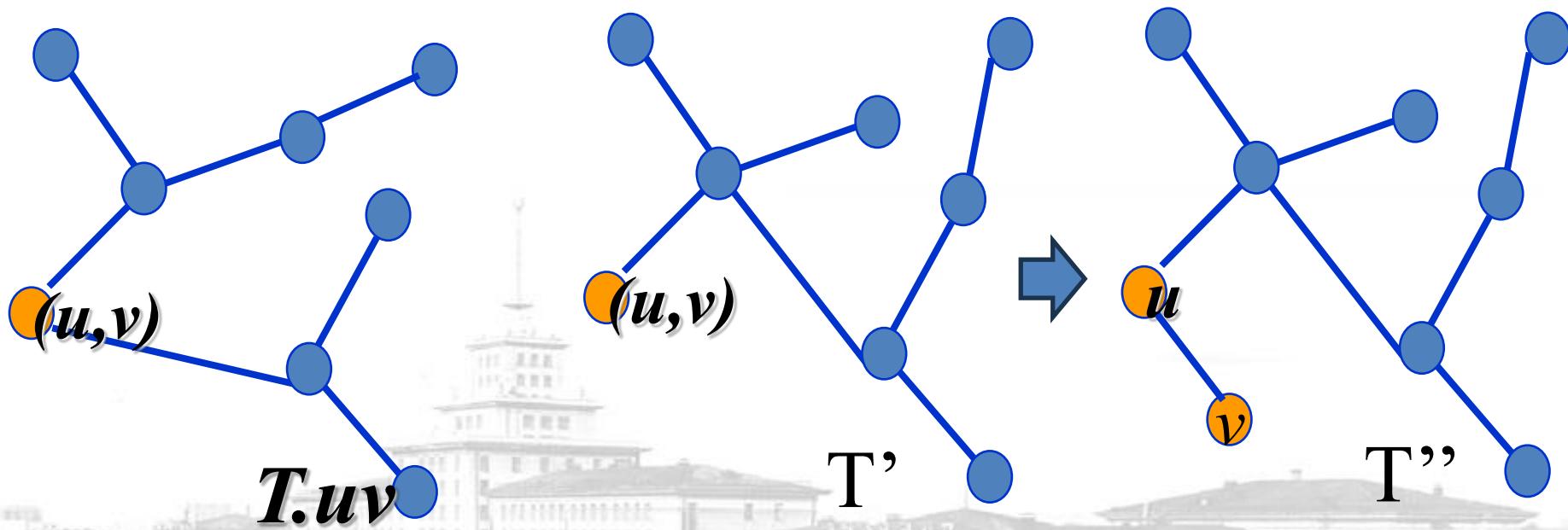


(2) 因为 G_{uv} 的顶点数 $n=k-1 < k$, 根据假设, 存在一棵由kruskal算法建树的 T' 是 G_{uv} 的最小生成树。



(2) 因为 G_{uv} 的顶点数 $n=k-1 < k$, 根据假设, 存在一棵由kruskal算法建树的 T' 是 G_{uv} 的最小生成树。

(3) 讨论 T' 和 T_{uv} 。若 T_{uv} 跟 T' 一致, 那么通过kruskal算法产生的生成树就是 G 的最小生成树。若不然, 我们用 T' 替换 T_{uv} , 对 T' 扩展 uv , 得到 T'' 。显然 T'' 是 G 的一棵通过kruskal算法建树的生成树。

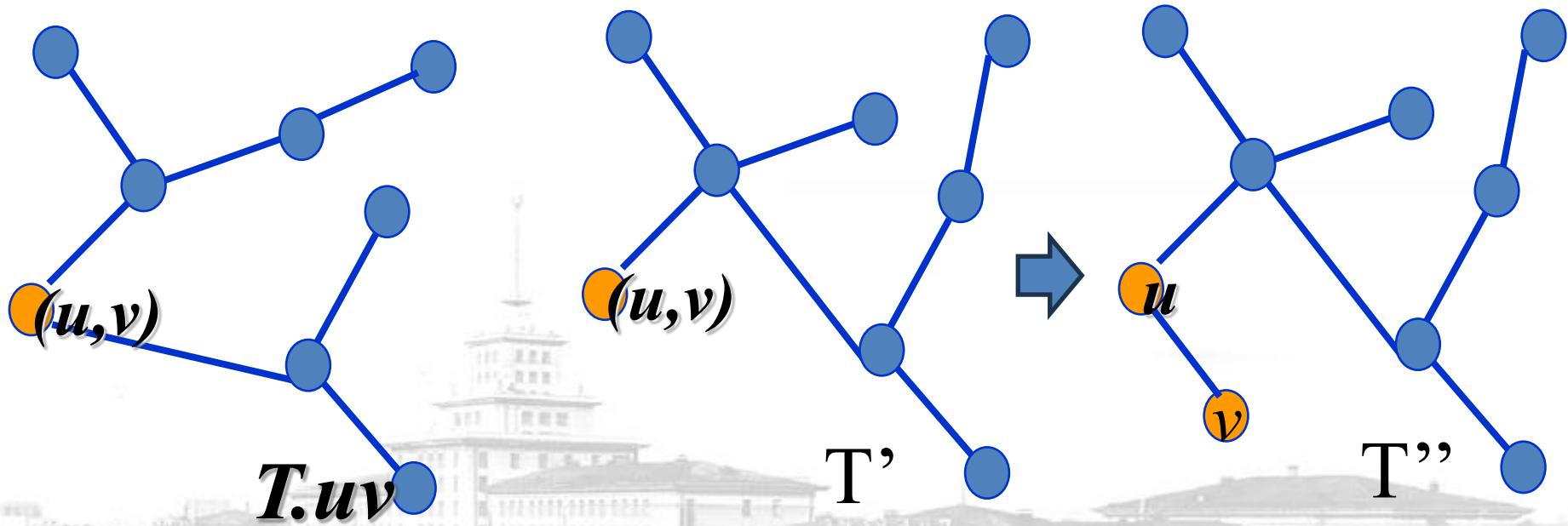


(2) 因为 G_{uv} 的顶点数 $n=k-1 < k$, 根据假设, 存在一棵由kruskal算法建树的 T' 是 G_{uv} 的最小生成树。

(3) 讨论 T' 和 T_{uv} 。若 T_{uv} 跟 T' 一致, 那么通过kruskal算法产生的生成树就是 G 的最小生成树。若不然, 我们用 T' 替换 T_{uv} , 对 T' 扩展 uv , 得到 T'' 。显然 T'' 是 G 的一棵通过kruskal算法建树的生成树。

$$W(T'') = W(T') + W(u,v) = W(T_{uv}) + W(u,v) = W(T)$$

通过kruskal算法构造的 T'' 是一棵 G 的最小生成树



Prim算法

- 贪心思想
 - 也称“加点法”
 - 每次迭代选择轻量级边对应的点，加入到最小生成树中。算法从某一个顶点开始，逐渐长大直到覆盖整个连通图的所有顶点。



Prim算法

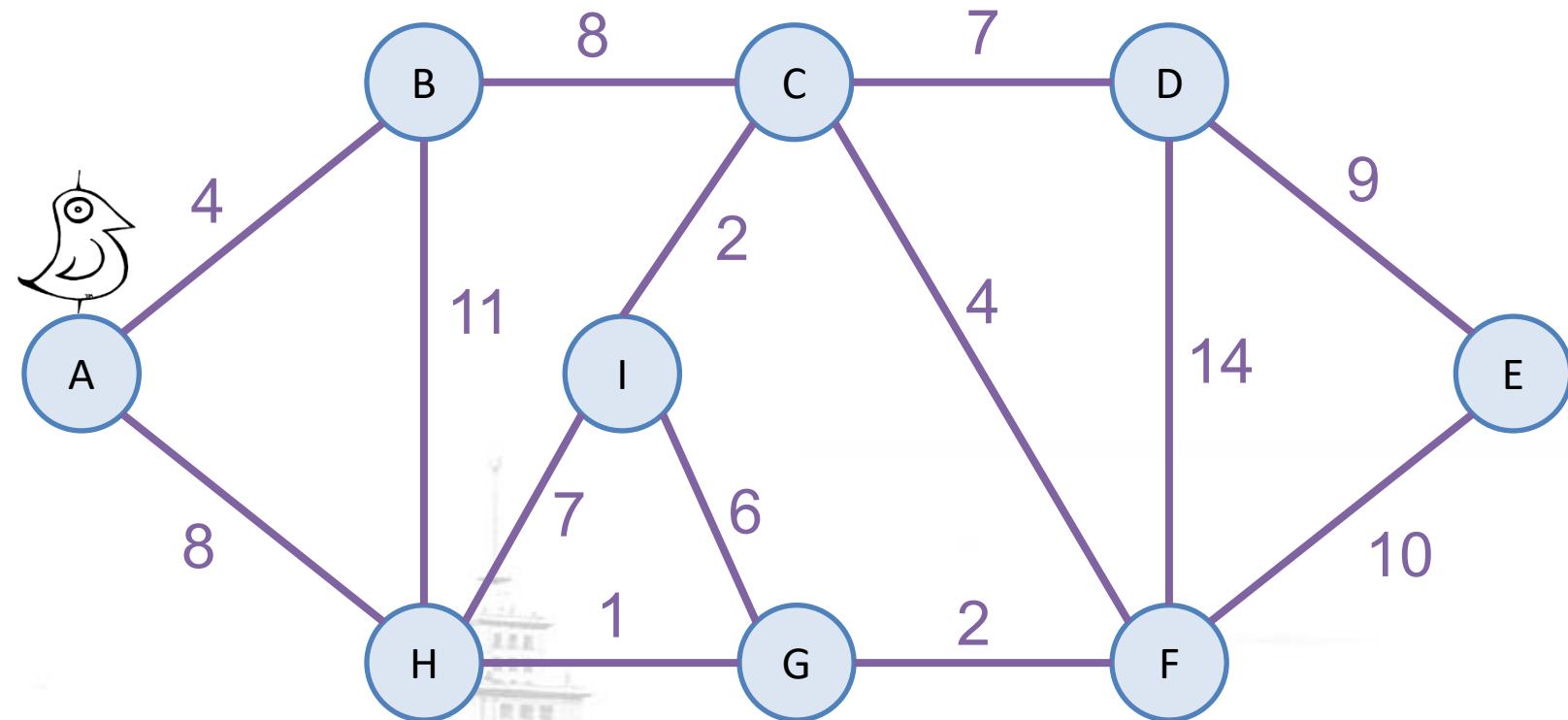
- 贪心思想

- 也称“加点法”
- 每次迭代选择轻量级边对应的点，加入到最小生成树中。算法从某一个顶点开始，逐渐长大直到覆盖整个连通图的所有顶点。

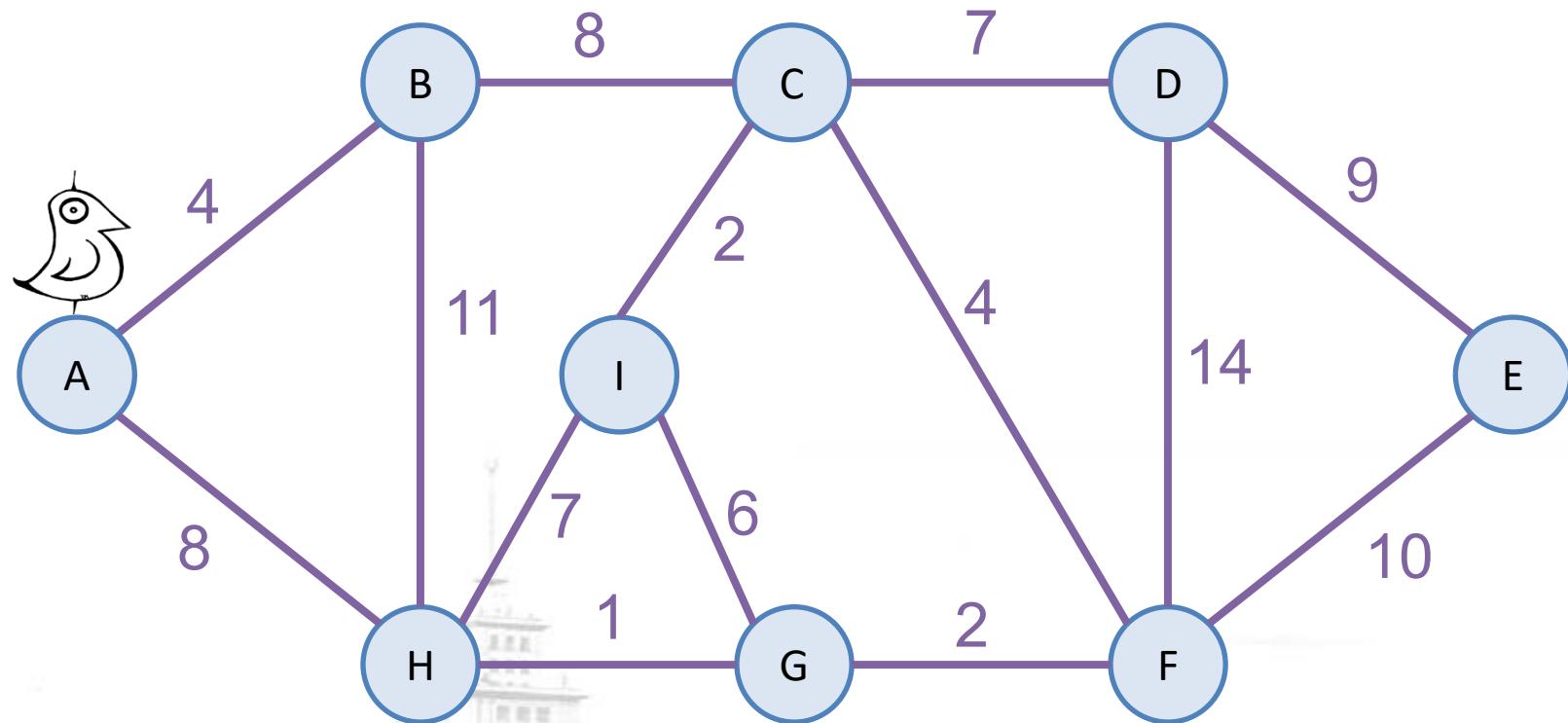
普里姆算法的基本思想：

1. 取图中任意一个顶点 v 作为生成树的根，之后往生成树上添加新的顶点 w 。
2. 在添加的顶点 w 和已经在生成树上的顶点 v 之间必定存在一条边，并且该边的权值在所有连通顶点 v 和 w 之间的边中取值最小。
3. 继续往生成树上添加顶点，直至生成树上含有 n 个顶点为止。

Prim算法



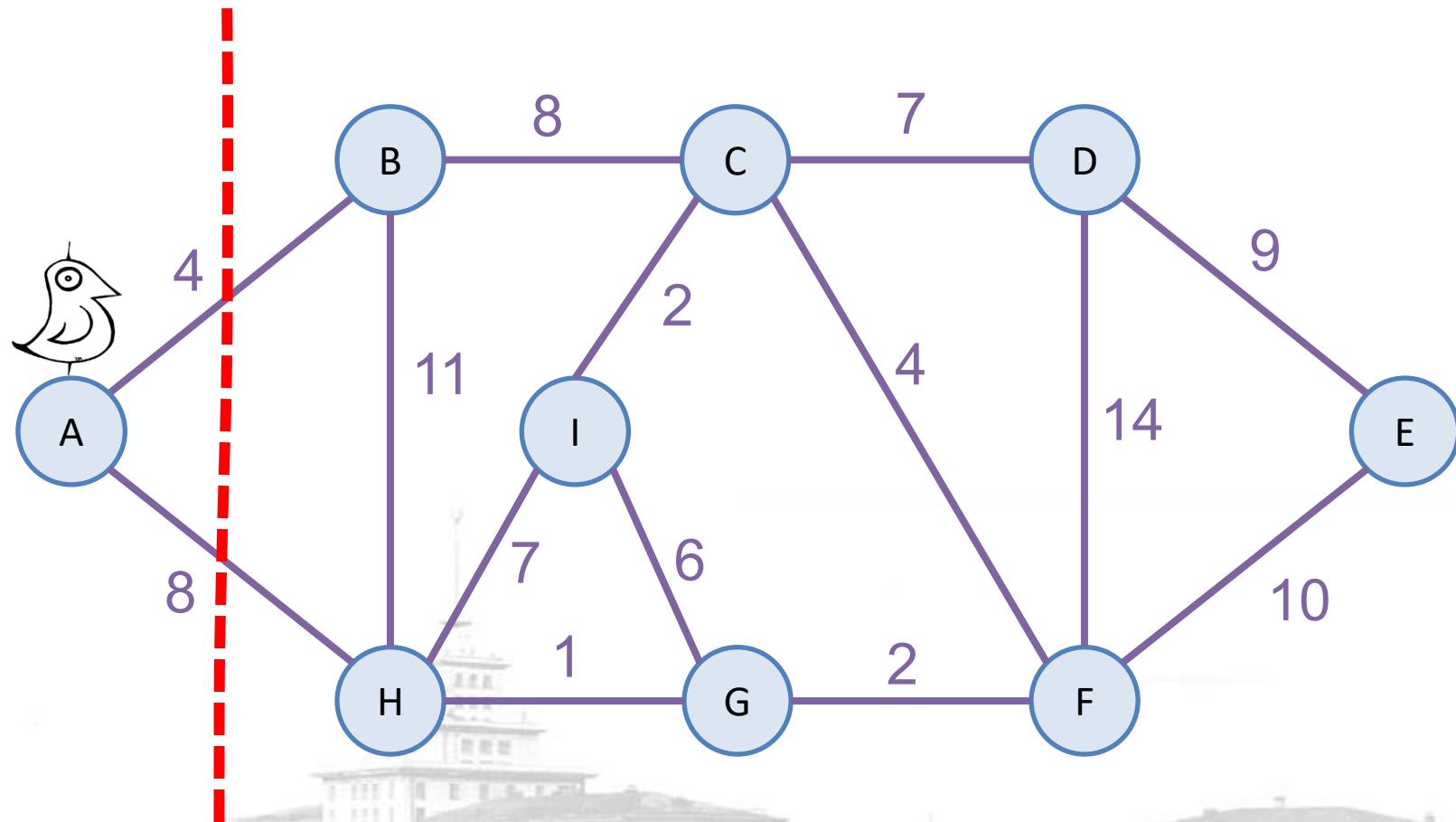
Prim算法



$$S = \emptyset$$

$$(u, v) = (A, B)$$

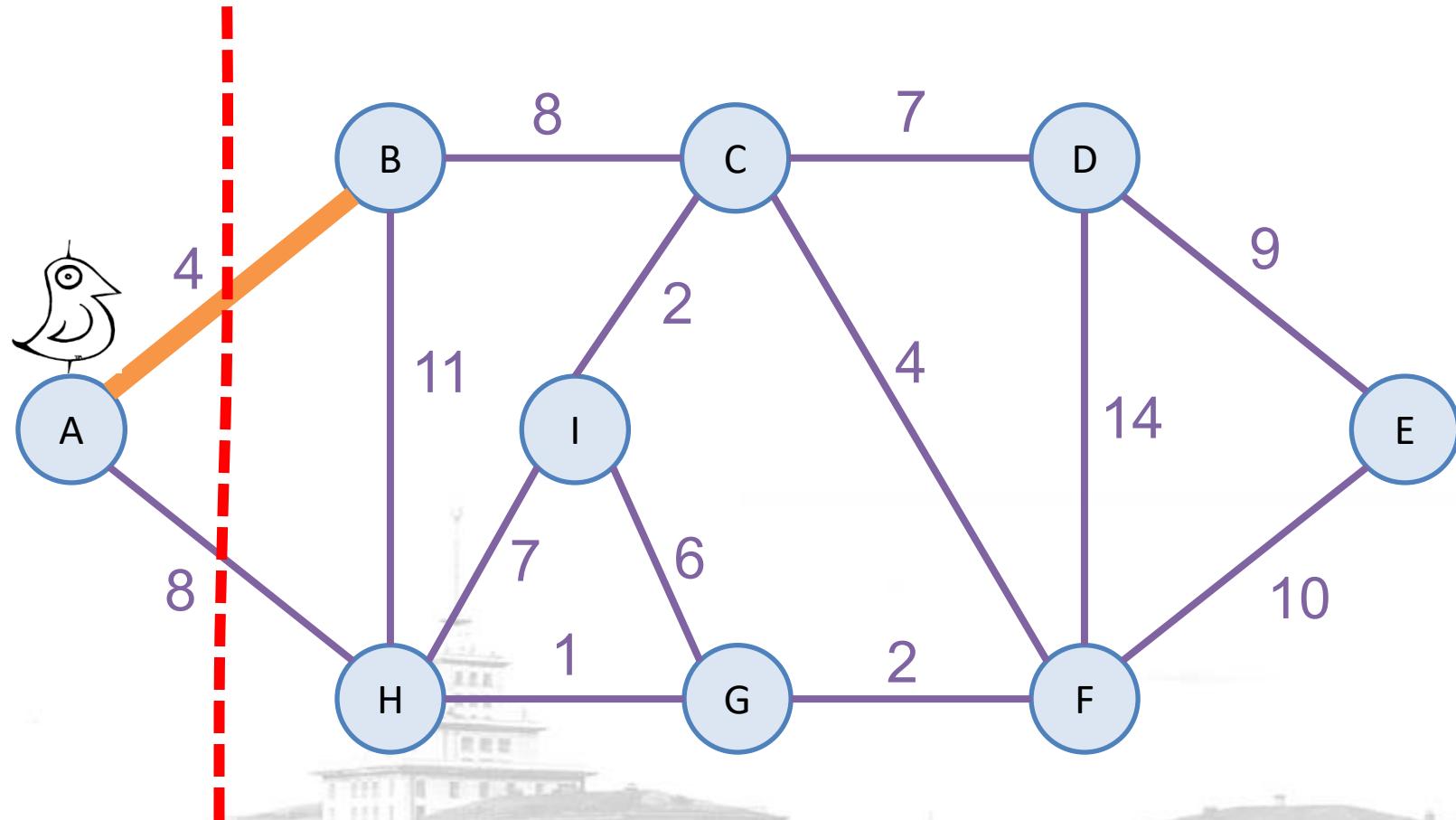
Prim 算法



$$S = \emptyset$$

$$(u, v) = (A, B)$$

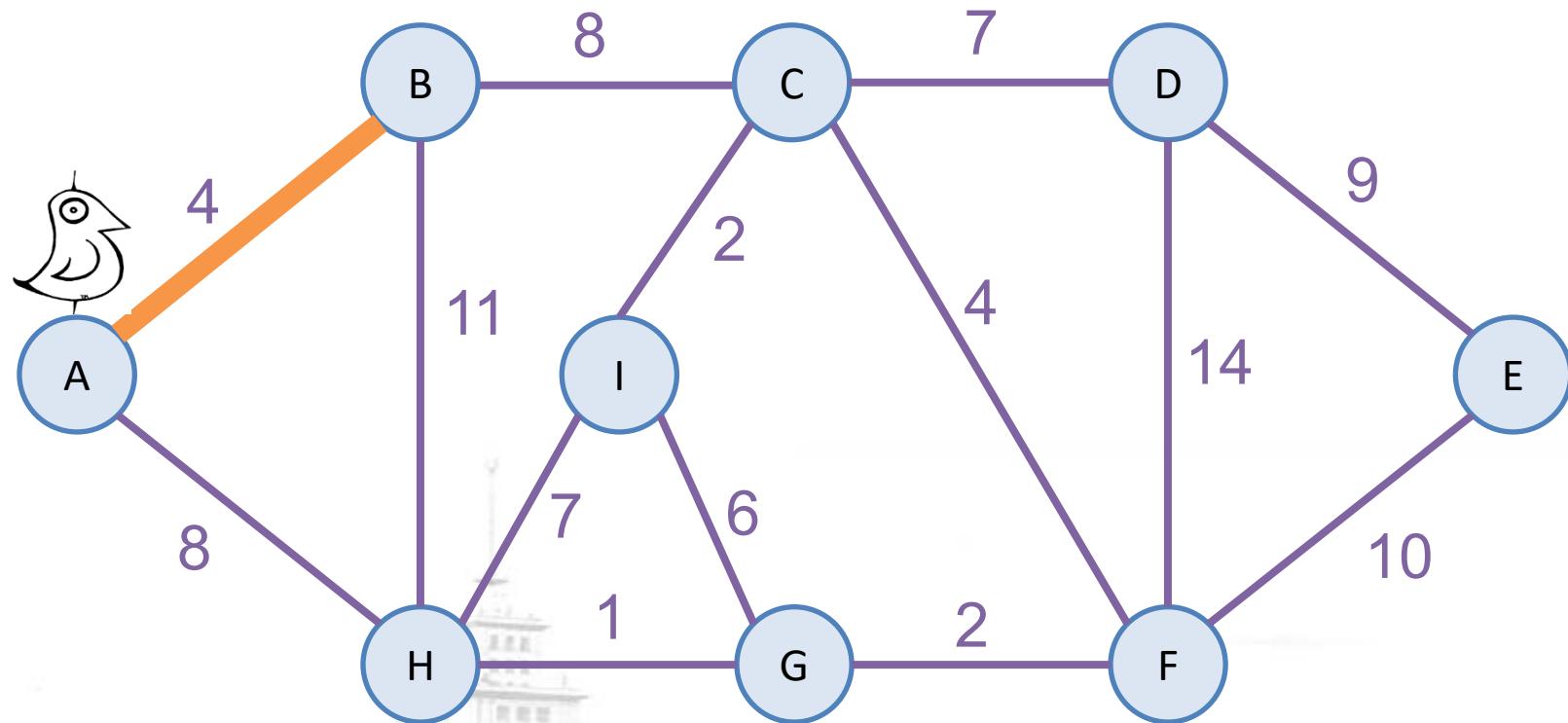
Prim 算法



$$S = \emptyset$$

$$(u, v) = (A, B)$$

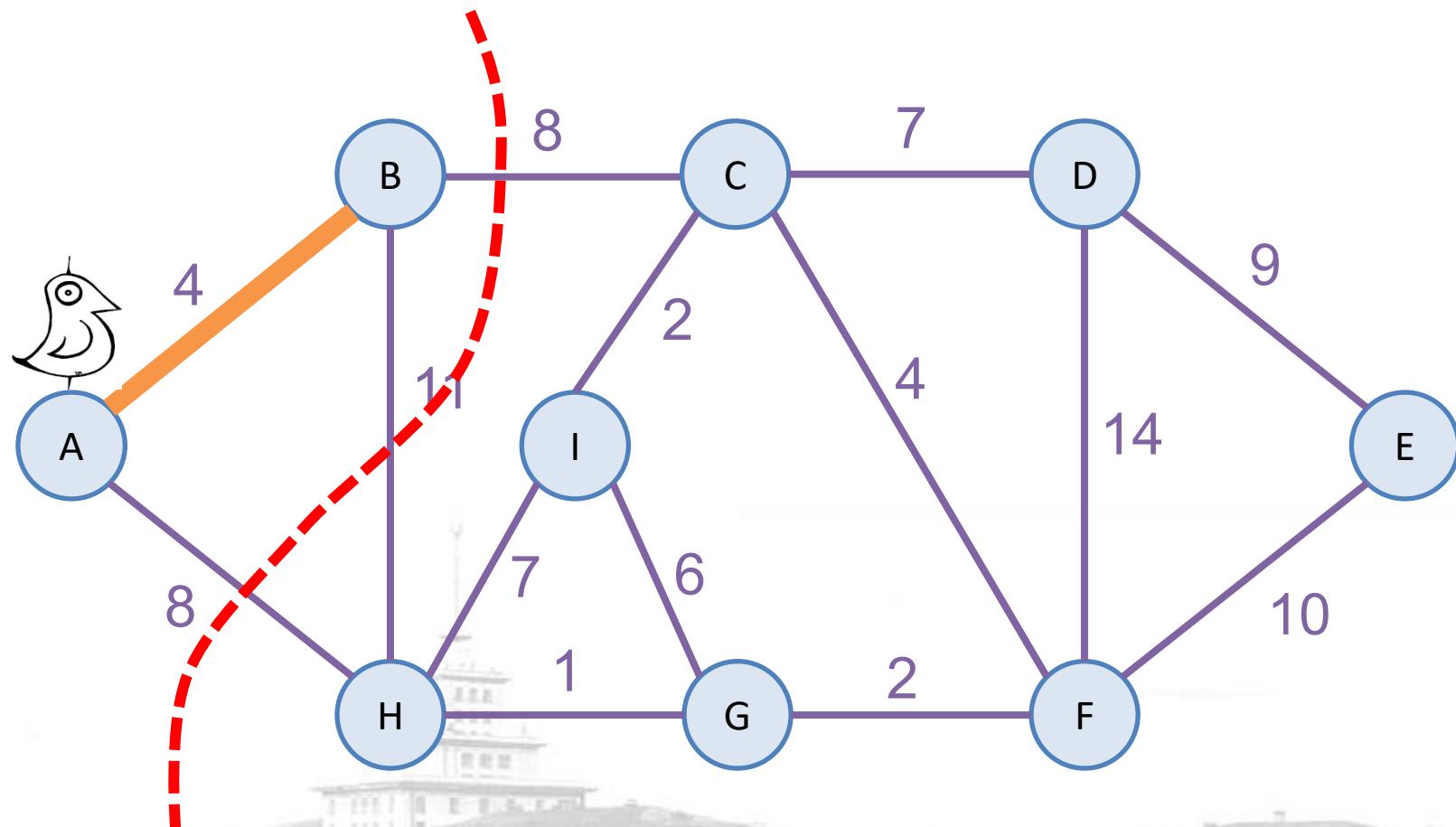
Prim 算法



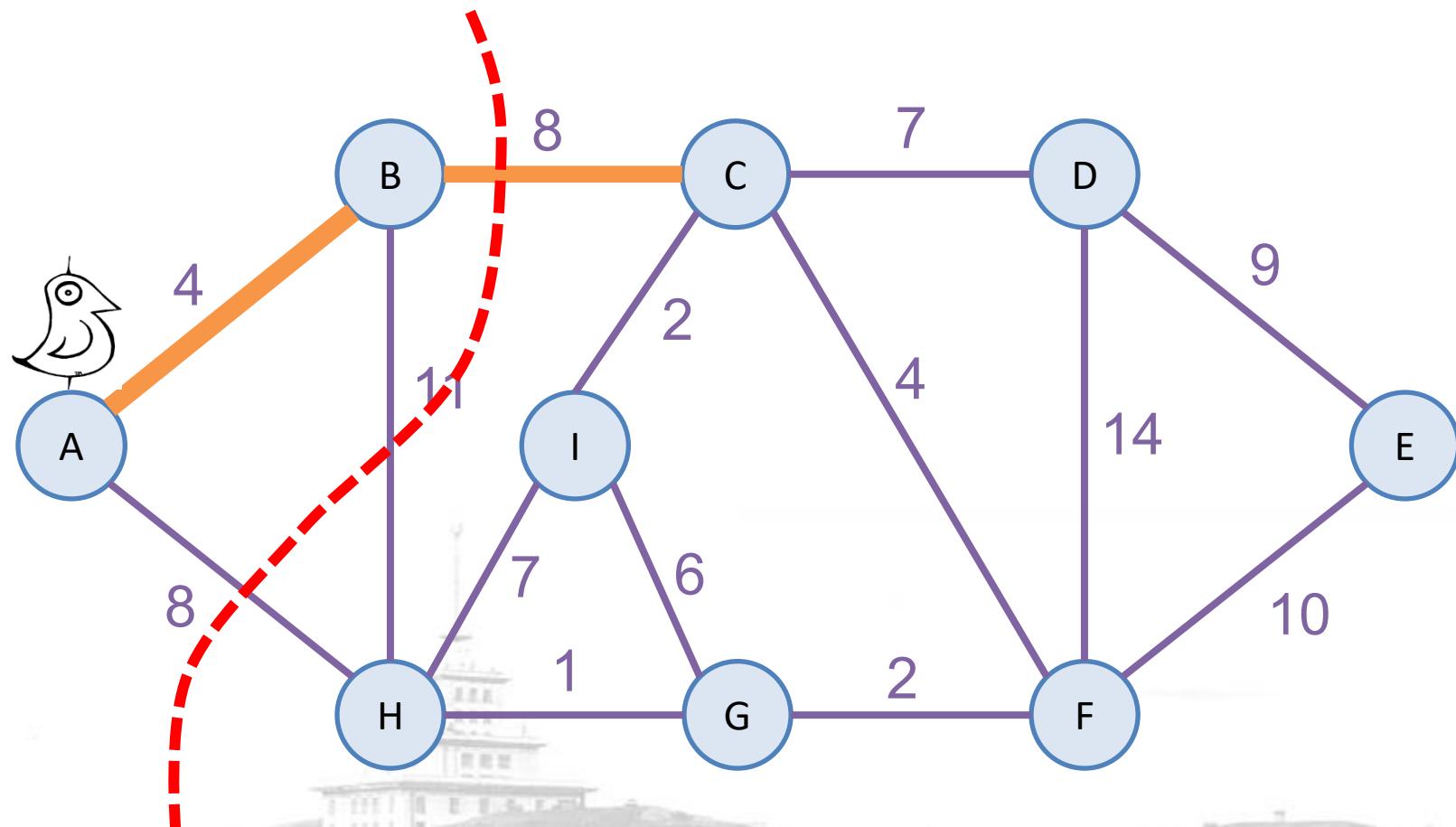
$$S = \{(A, B)\}$$

$$(u, v) = (B, C)$$

Prim 算法



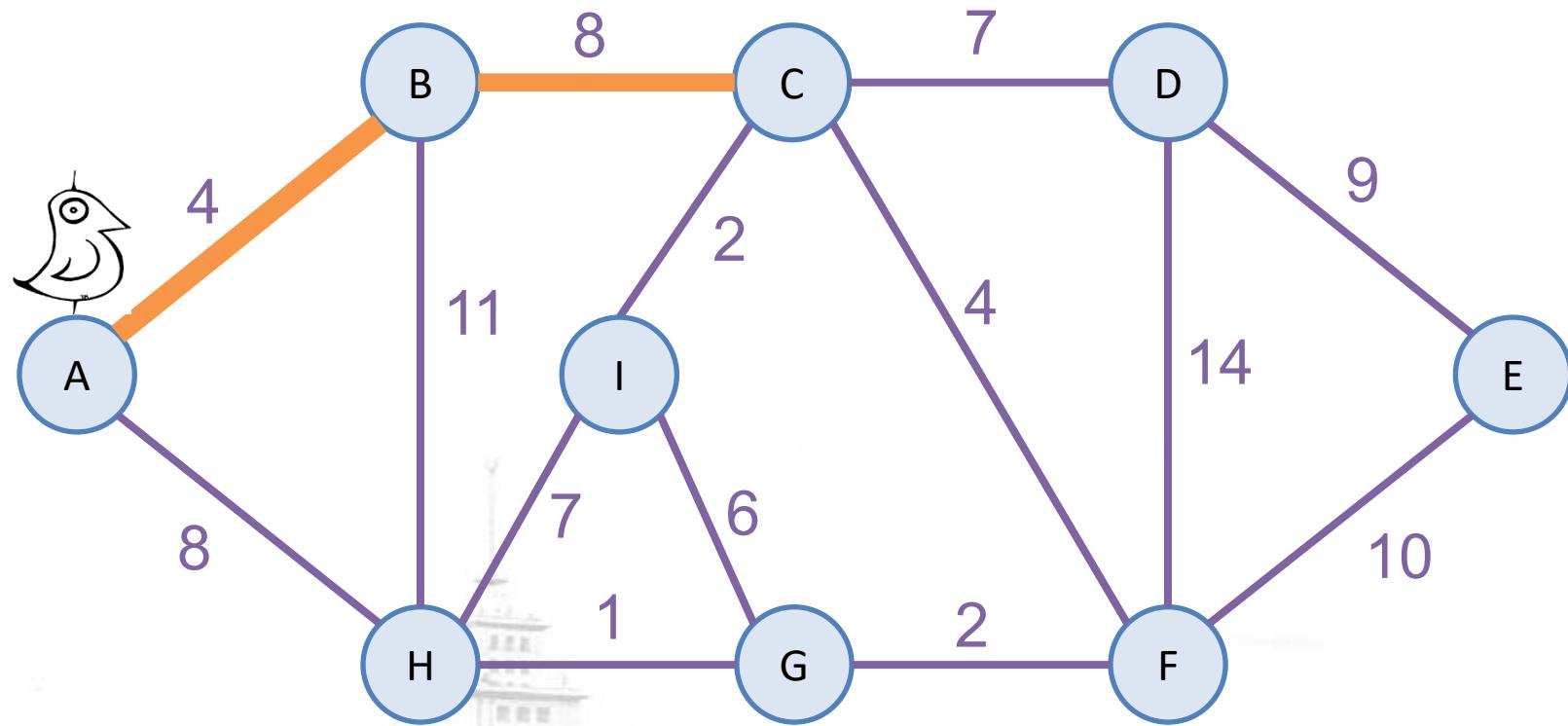
Prim 算法



$$S = \{(A, B)\}$$

$$(u, v) = (B, C)$$

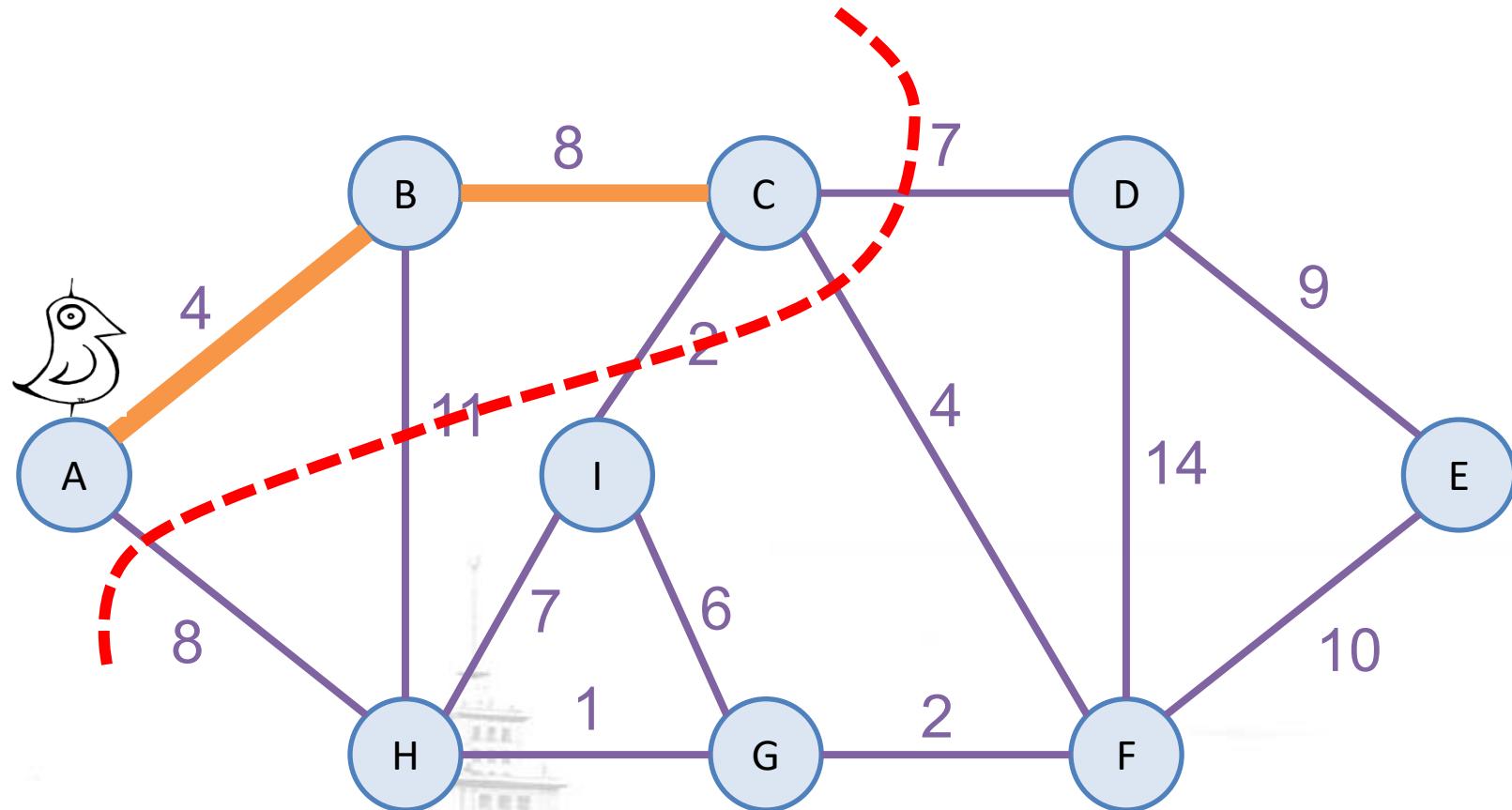
Prim算法



$$S = \{(A, B), (B, C)\}$$

$$(u, v) = (C, I)$$

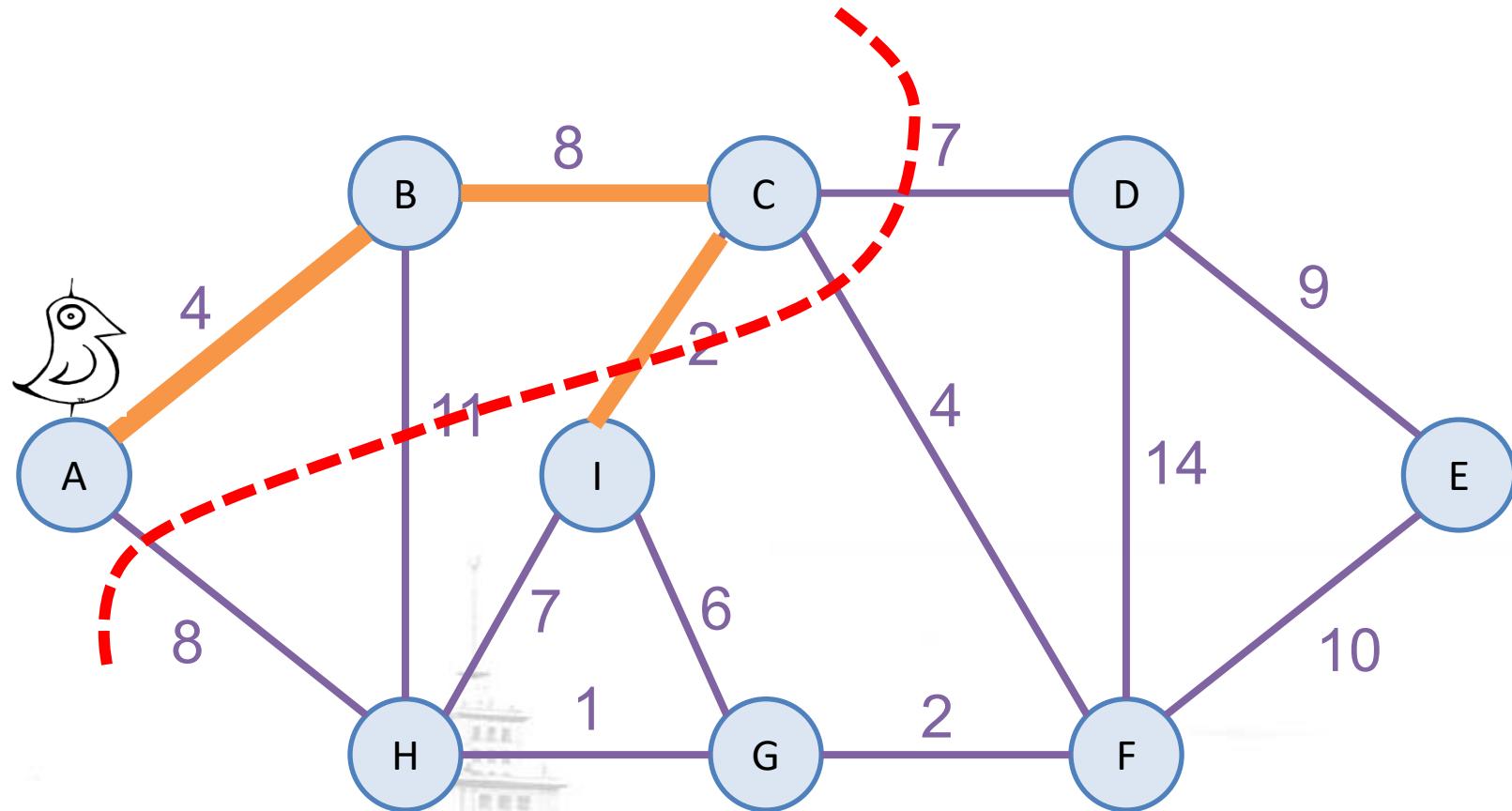
Prim 算法



$$S = \{(A, B), (B, C)\}$$

$$(u, v) = (C, I)$$

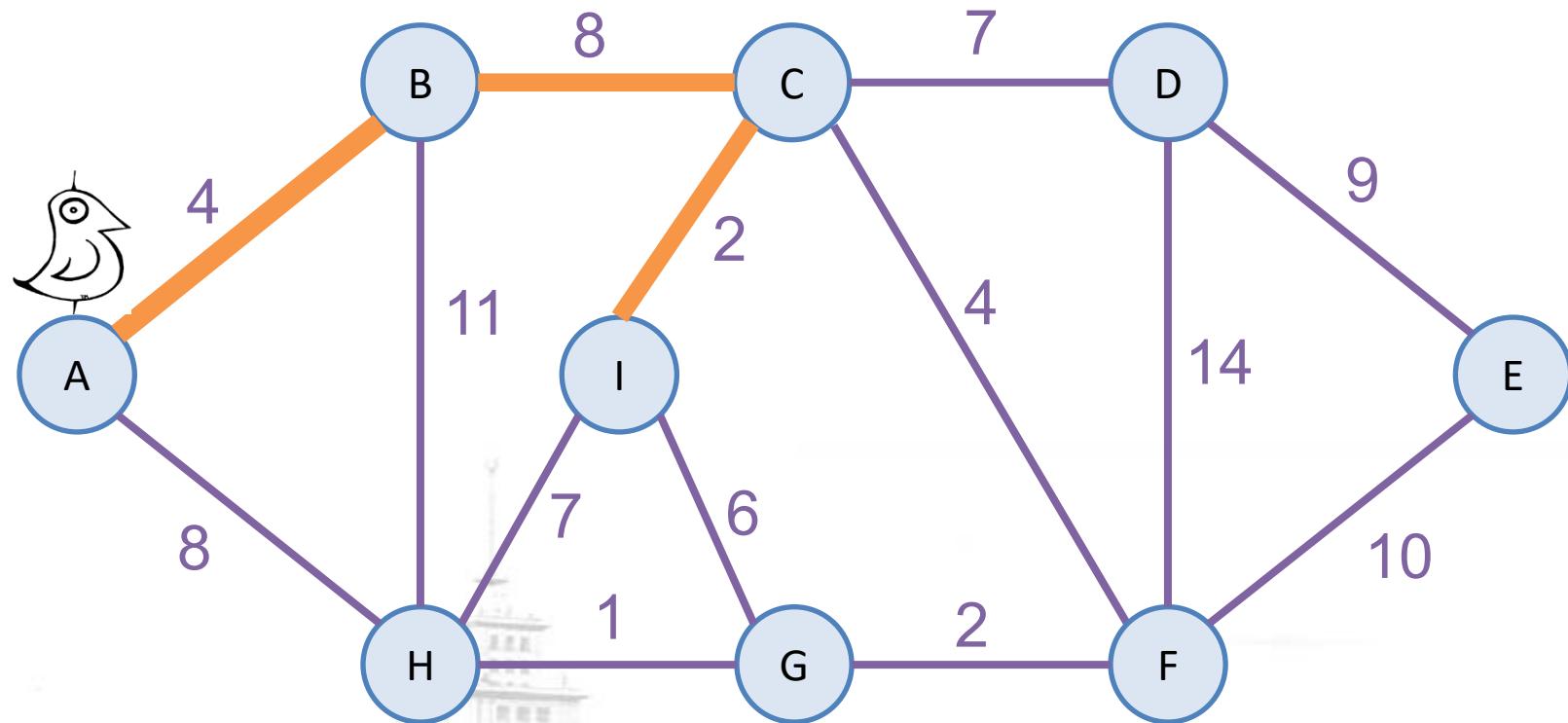
Prim 算法



$$S = \{(A, B), (B, C)\}$$

$$(u, v) = (C, I)$$

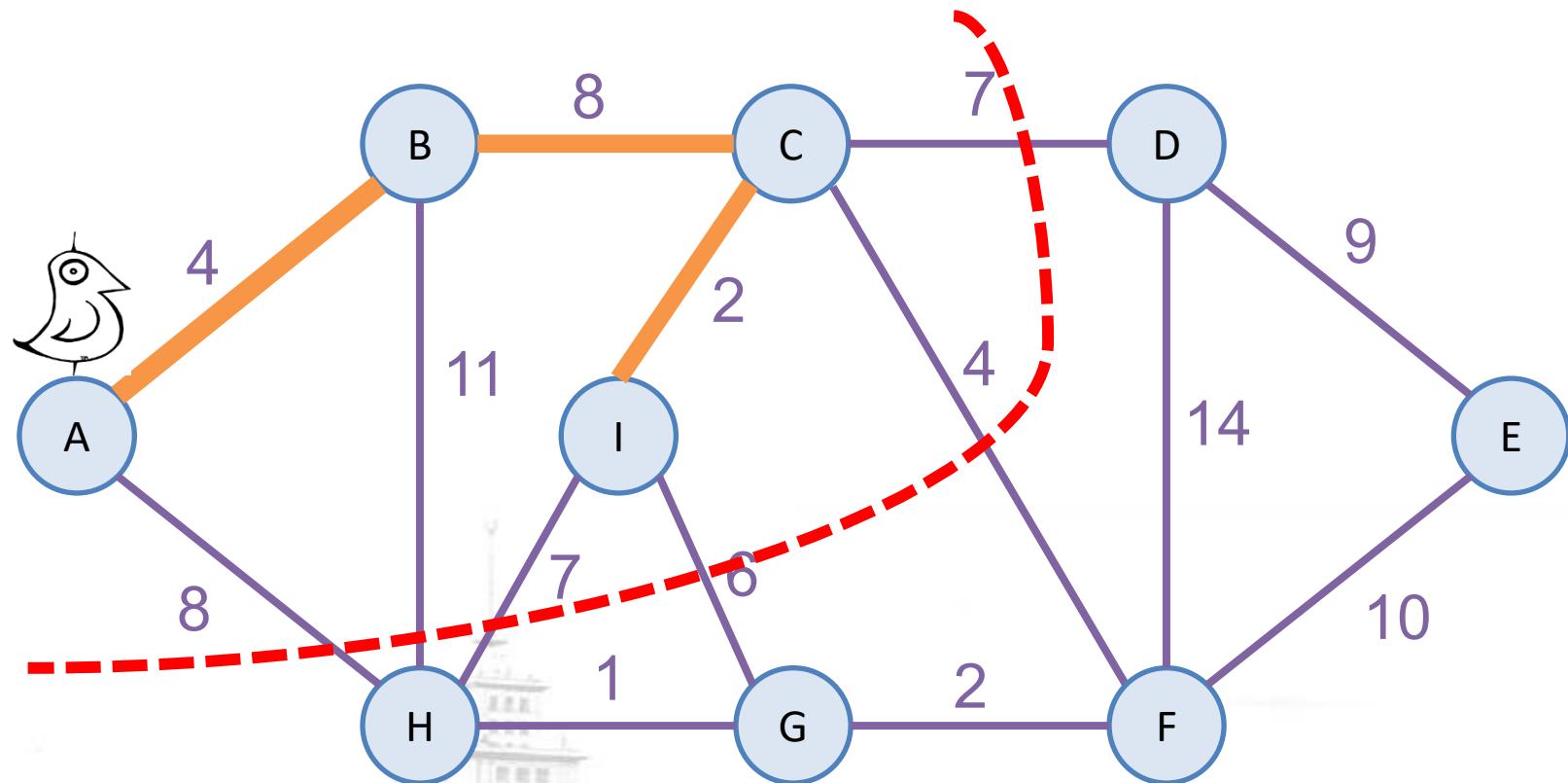
Prim算法



$$S = \{(A, B), (B, C), (C, I)\}$$

$$(u, v) = (C, F)$$

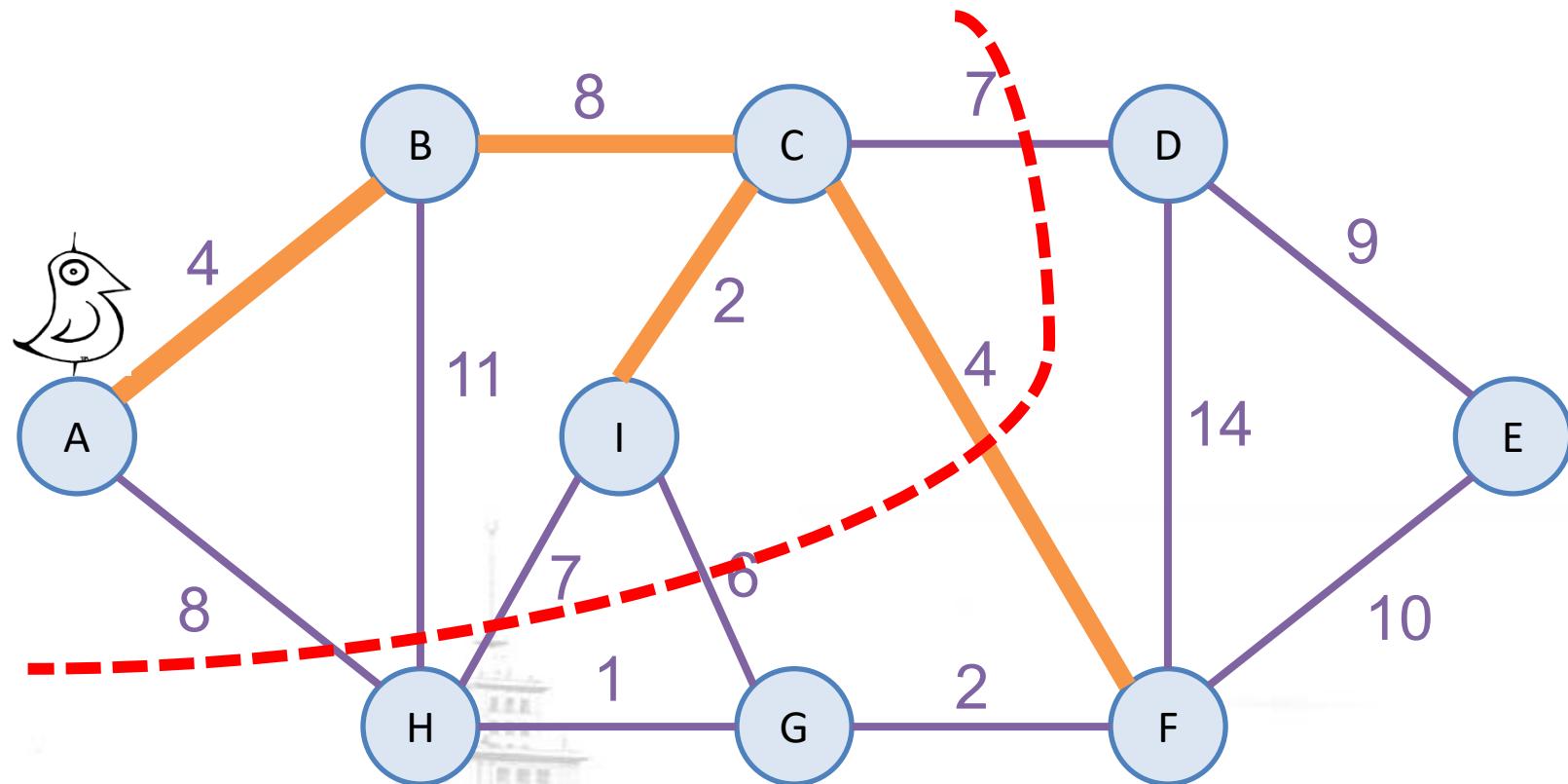
Prim算法



$$S = \{(A, B), (B, C), (C, I)\}$$

$$(u, v) = (C, F)$$

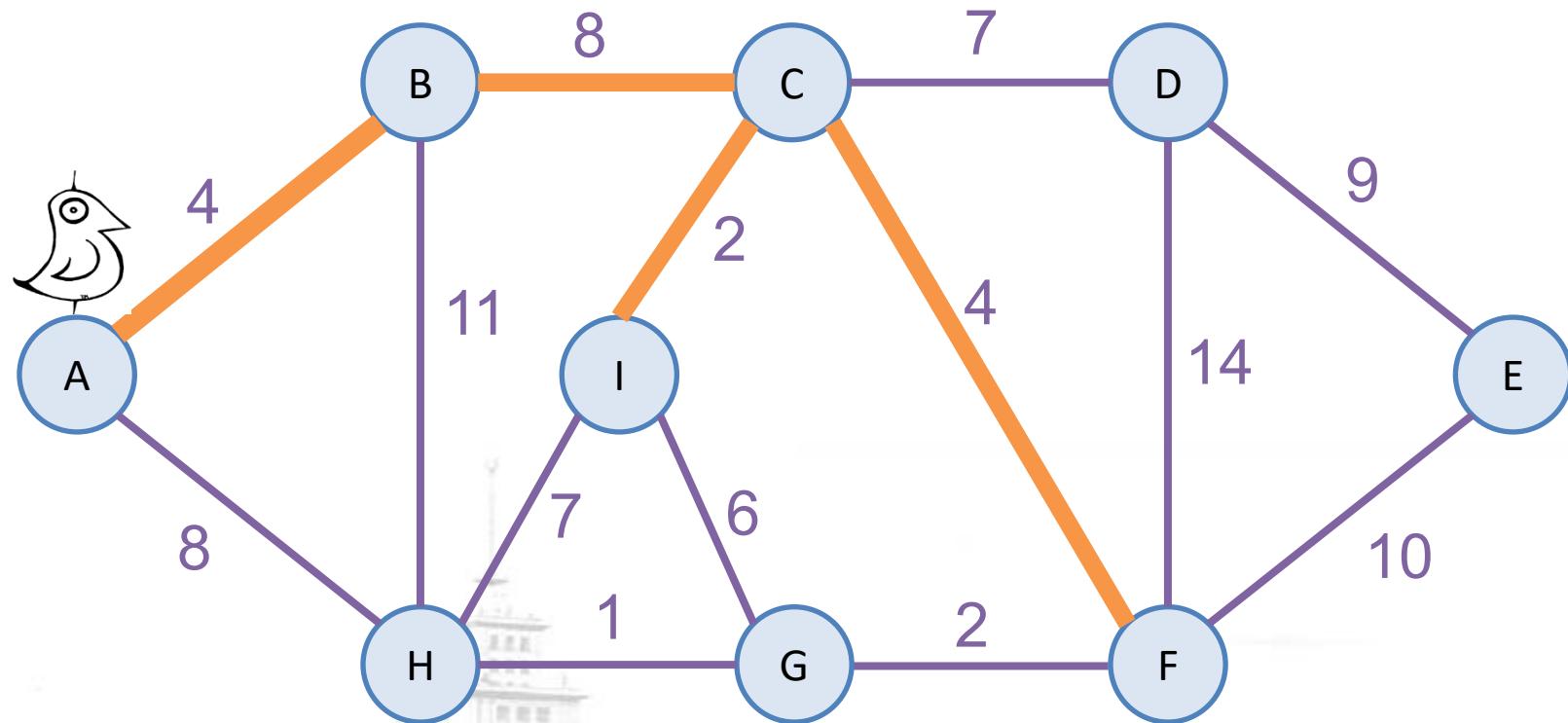
Prim算法



$$S = \{(A, B), (B, C), (C, I)\}$$

$$(u, v) = (C, F)$$

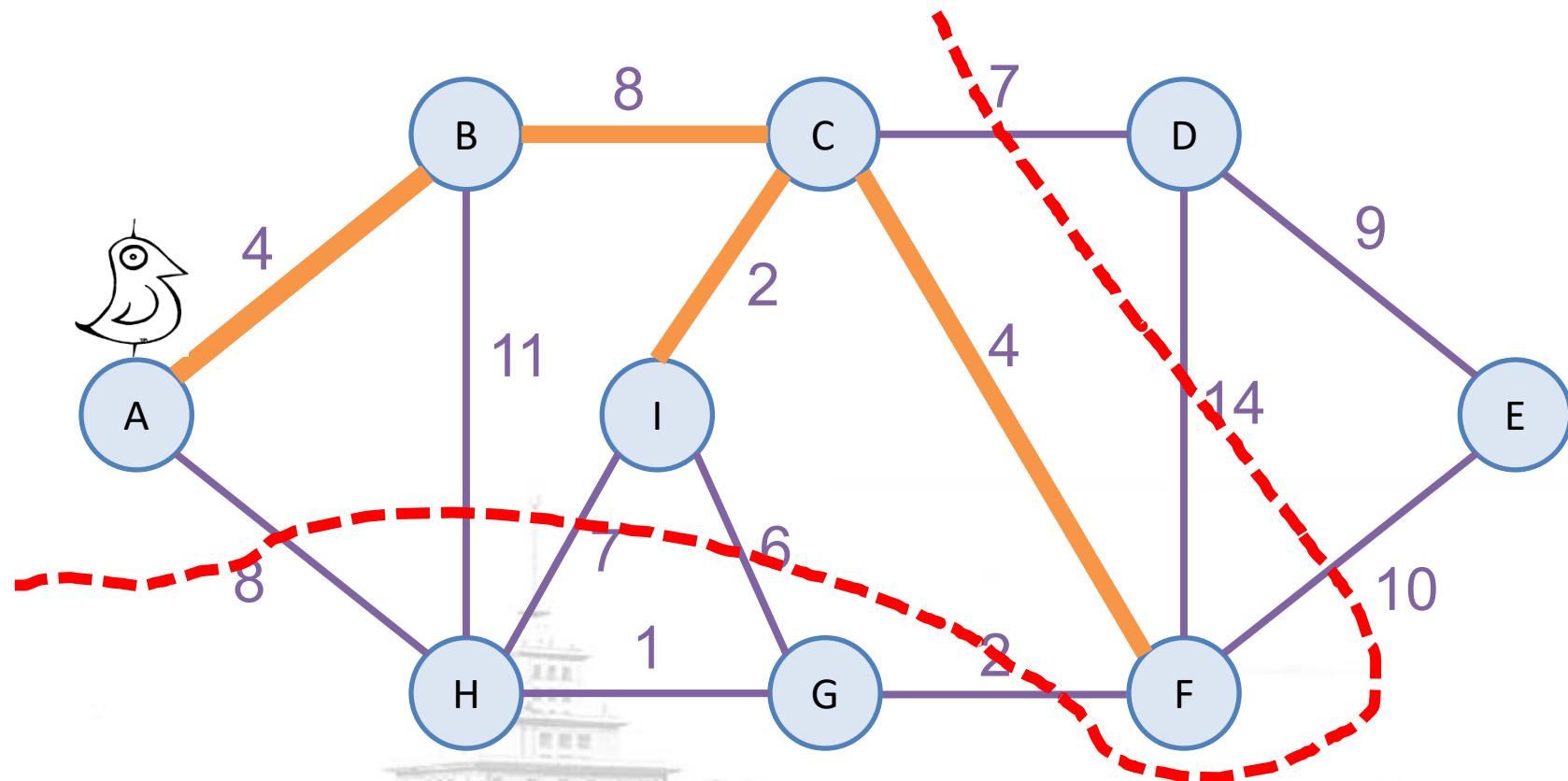
Prim算法



$$S = \{(A, B), (B, C), (C, I), (C, F)\}$$

$$(u, v) = (G, F)$$

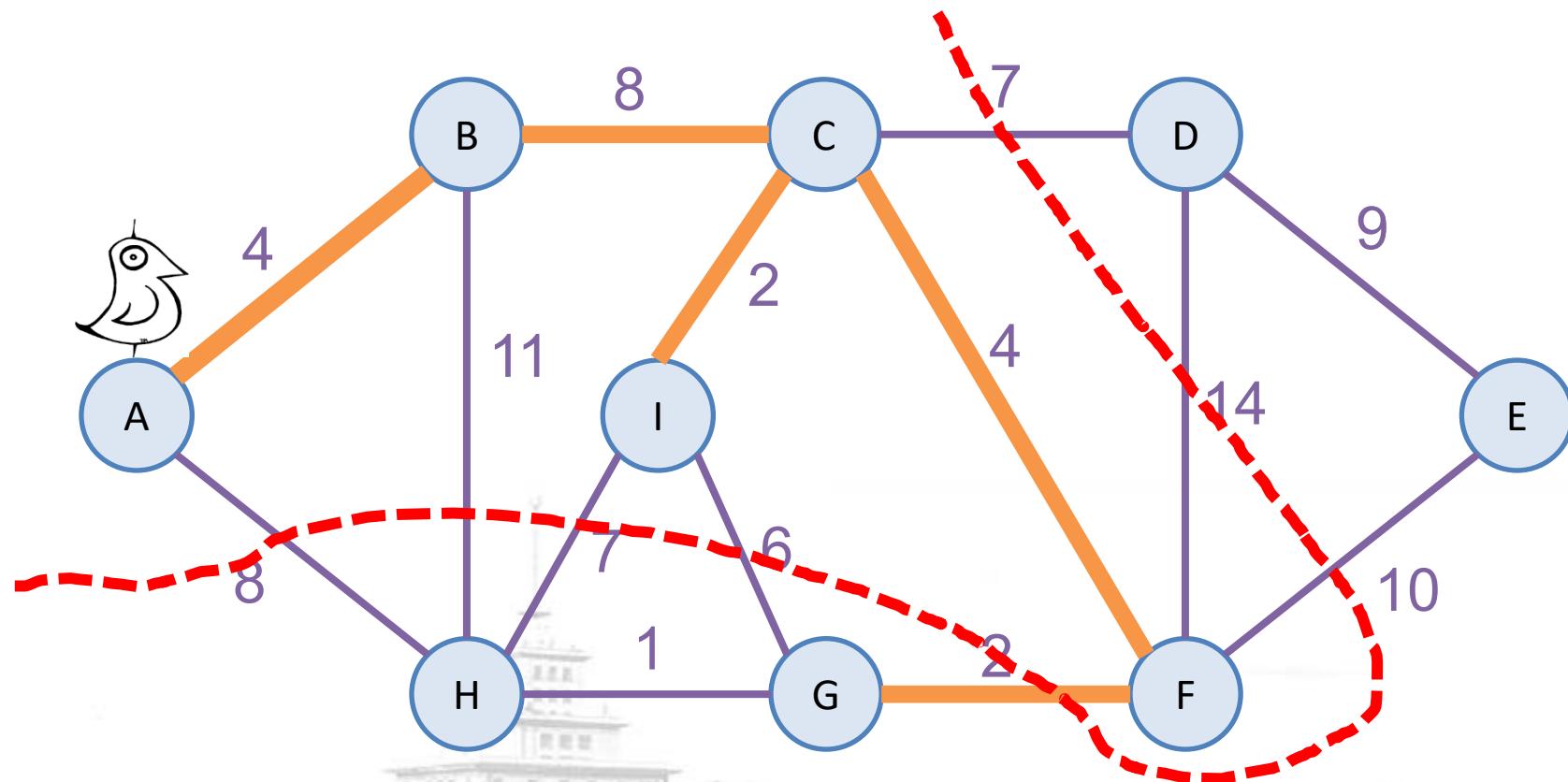
Prim 算法



$$S = \{(A, B), (B, C), (C, I), (C, F)\}$$

$$(u, v) = (G, F)$$

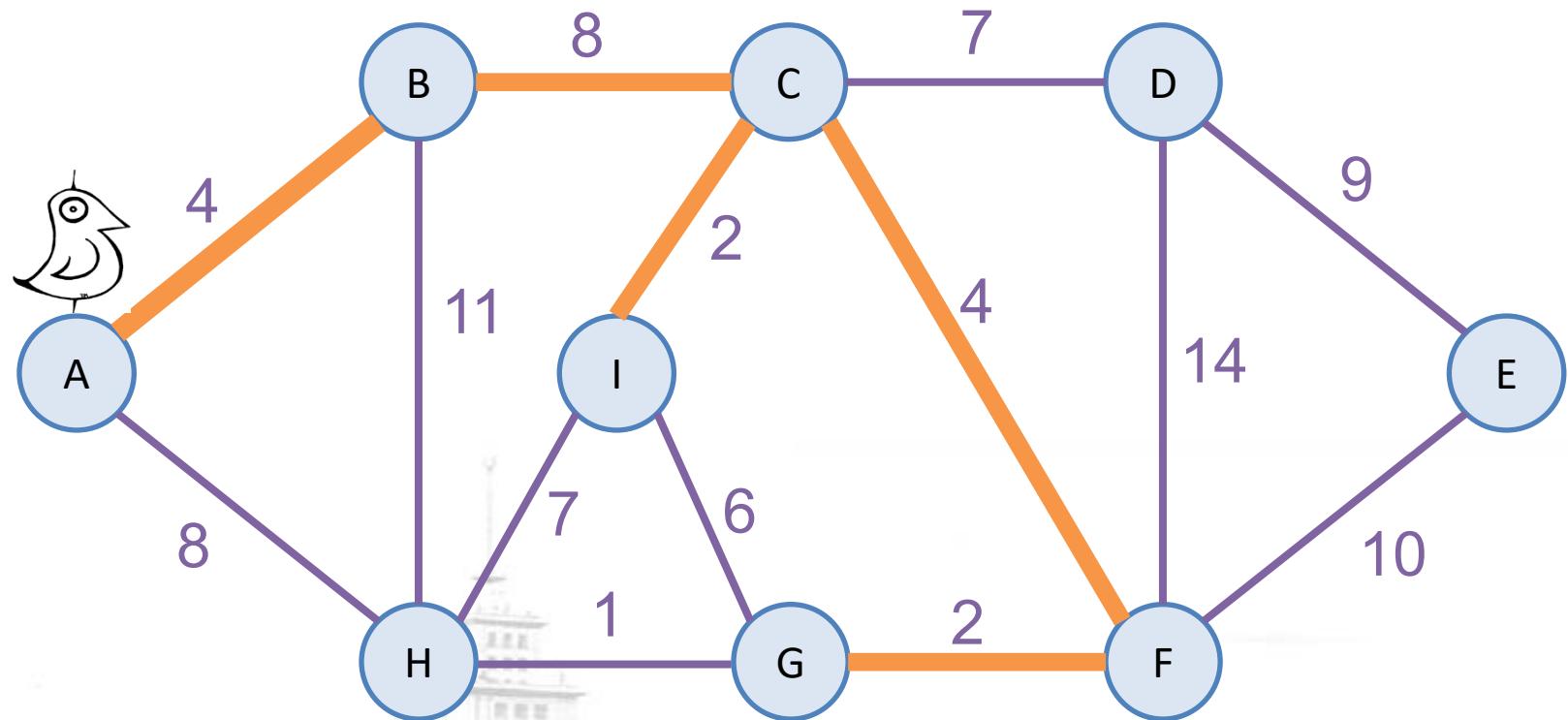
Prim 算法



$$S = \{(A, B), (B, C), (C, F)\}$$

$$(u, v) = (G, F)$$

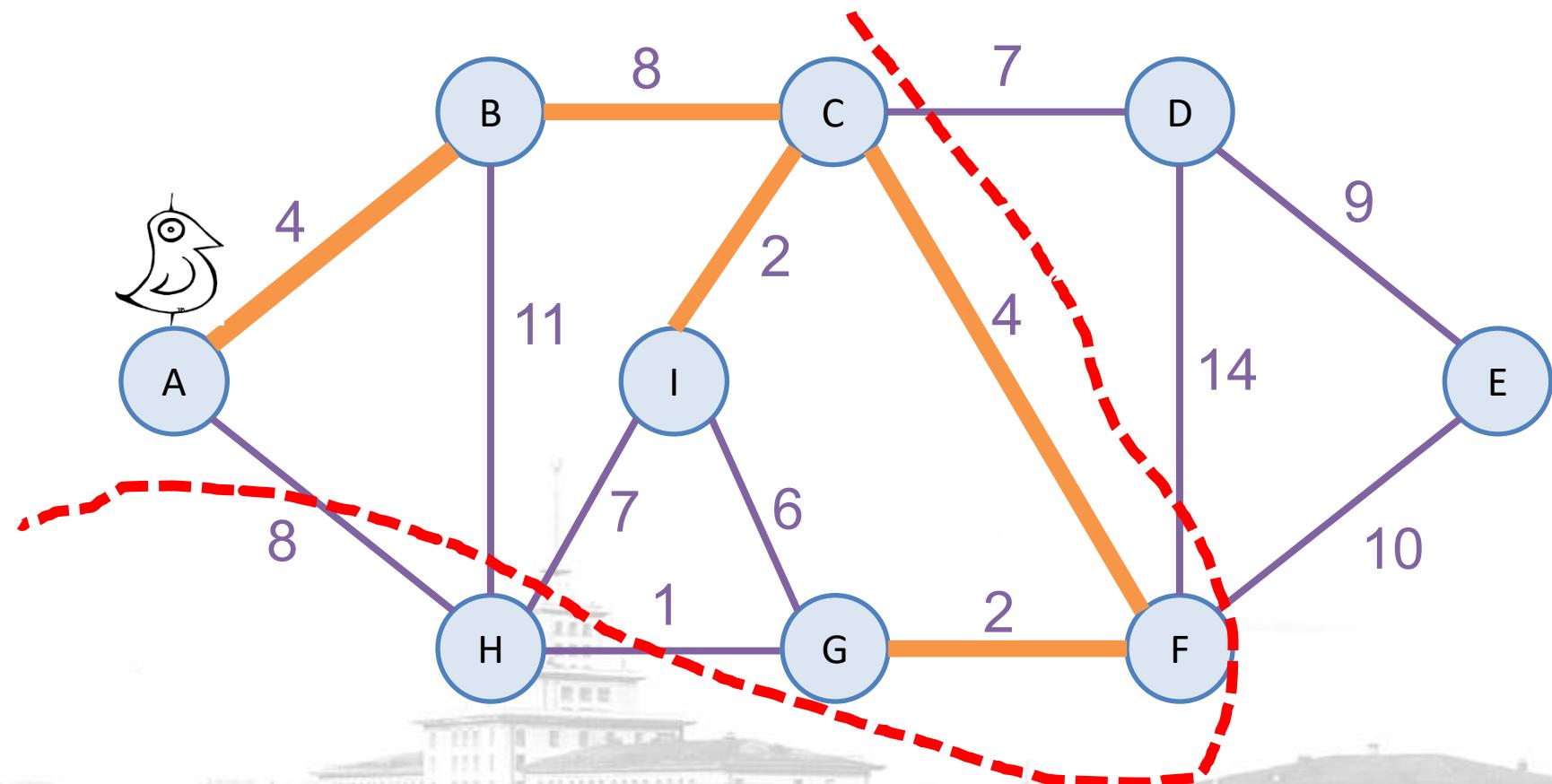
Prim算法



$$S = \{(A, B), (B, C), (C, F), (G, F)\}$$

$$(u, v) = (H, G)$$

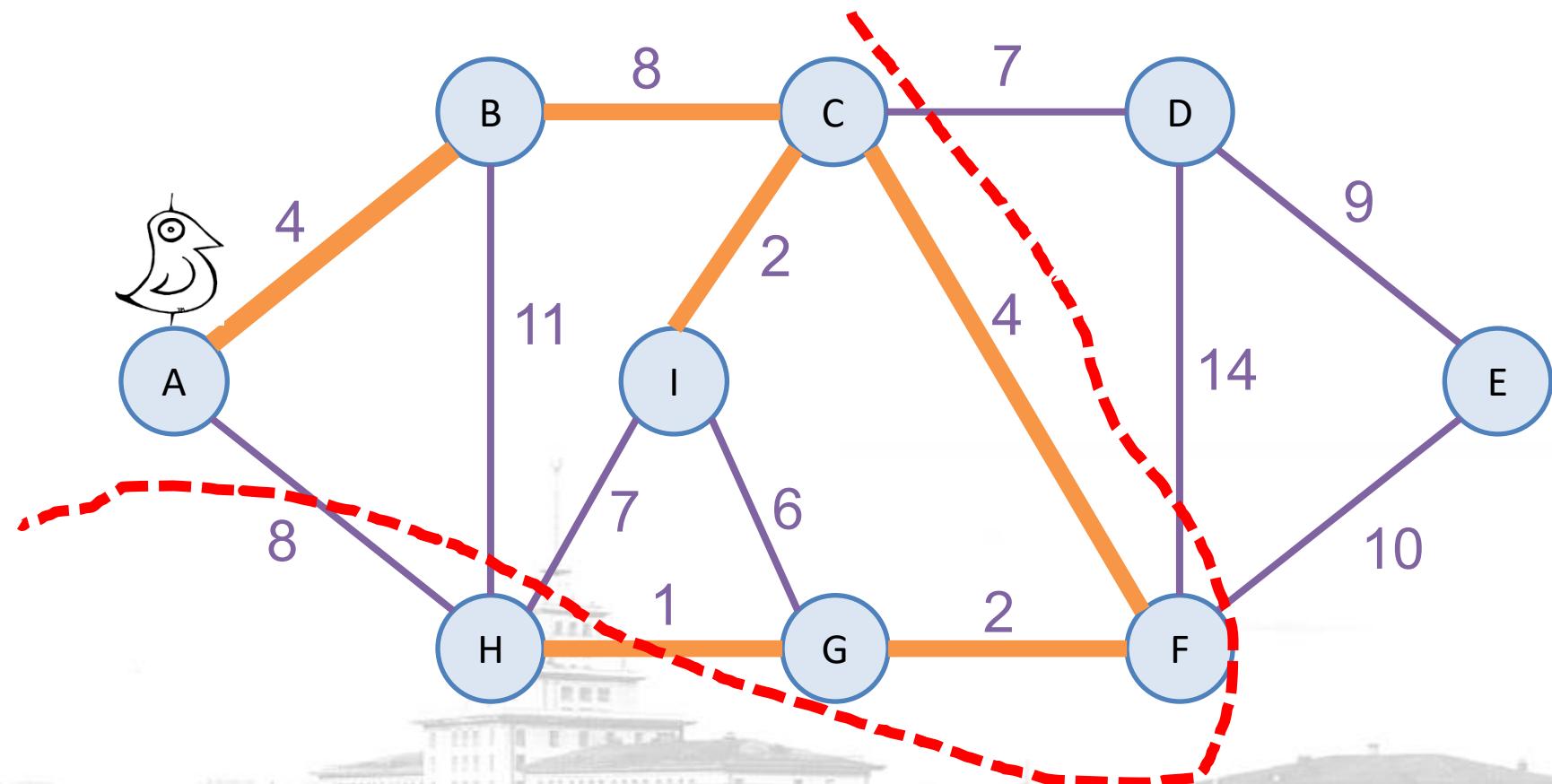
Prim 算法



$$S = \{(A, B), (B, C), (C, I), (C, F), (G, F)\}$$

$$(u, v) = (H, G)$$

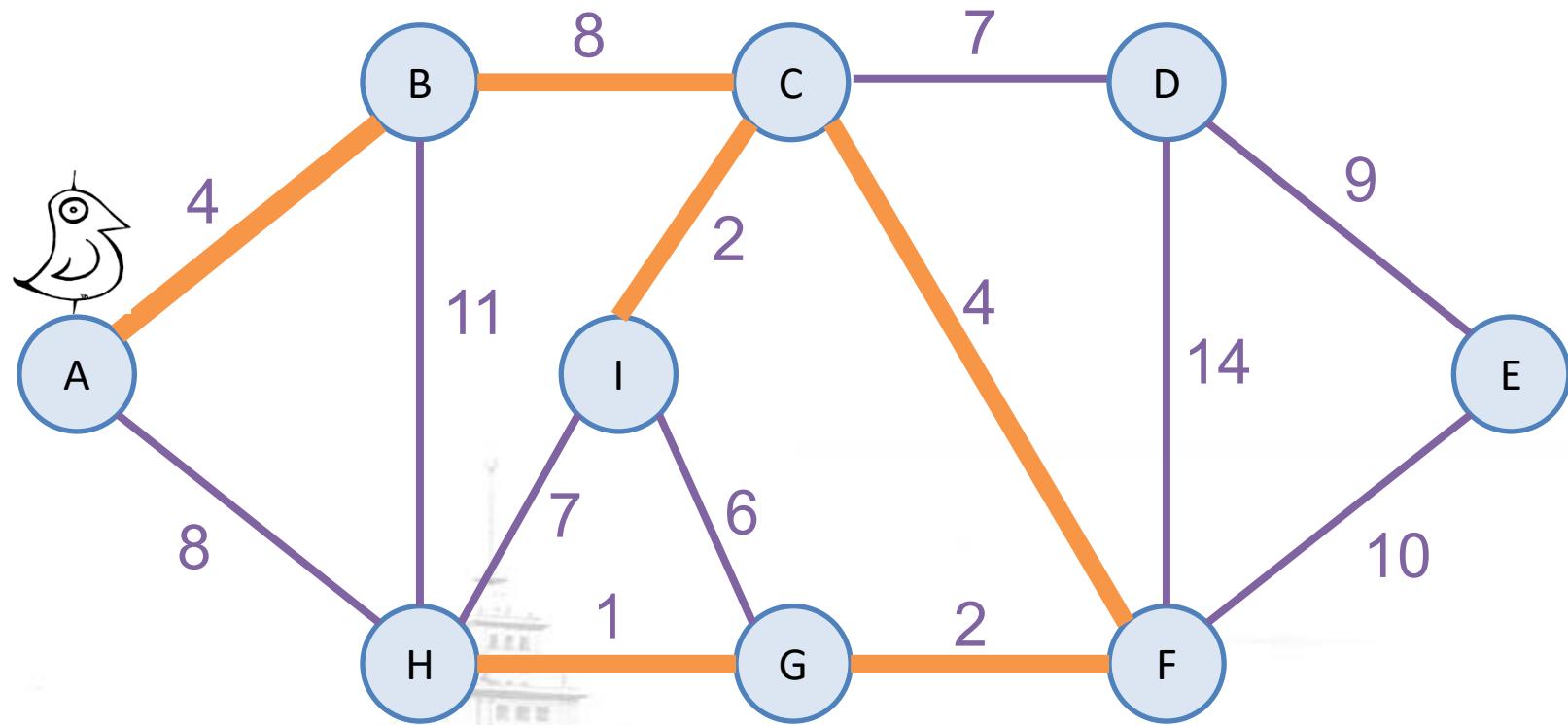
Prim 算法



$$S = \{(A, B), (B, C), (C, F), (F, G), (G, H)\}$$

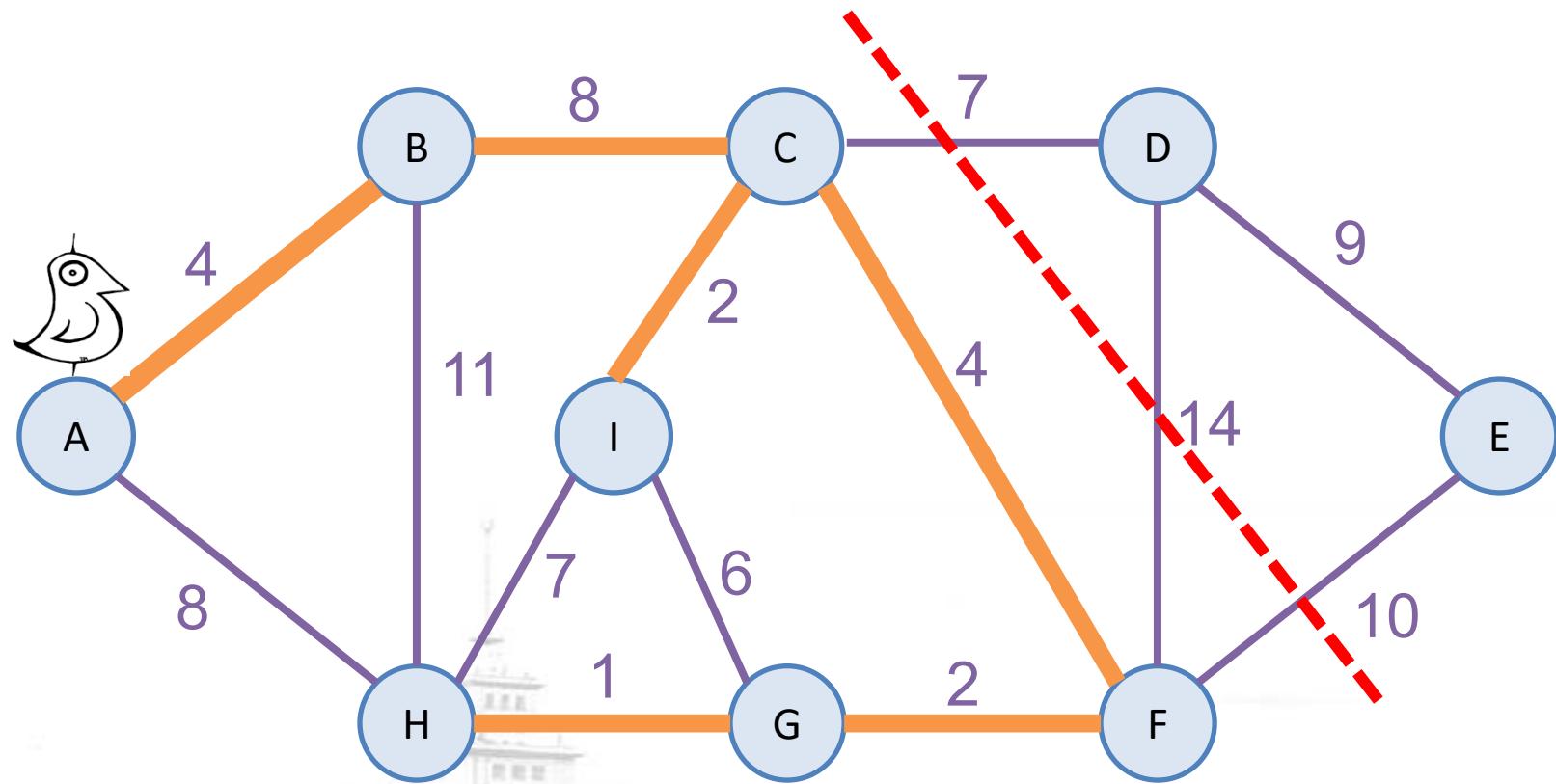
$$(u, v) = (H, G)$$

Prim算法



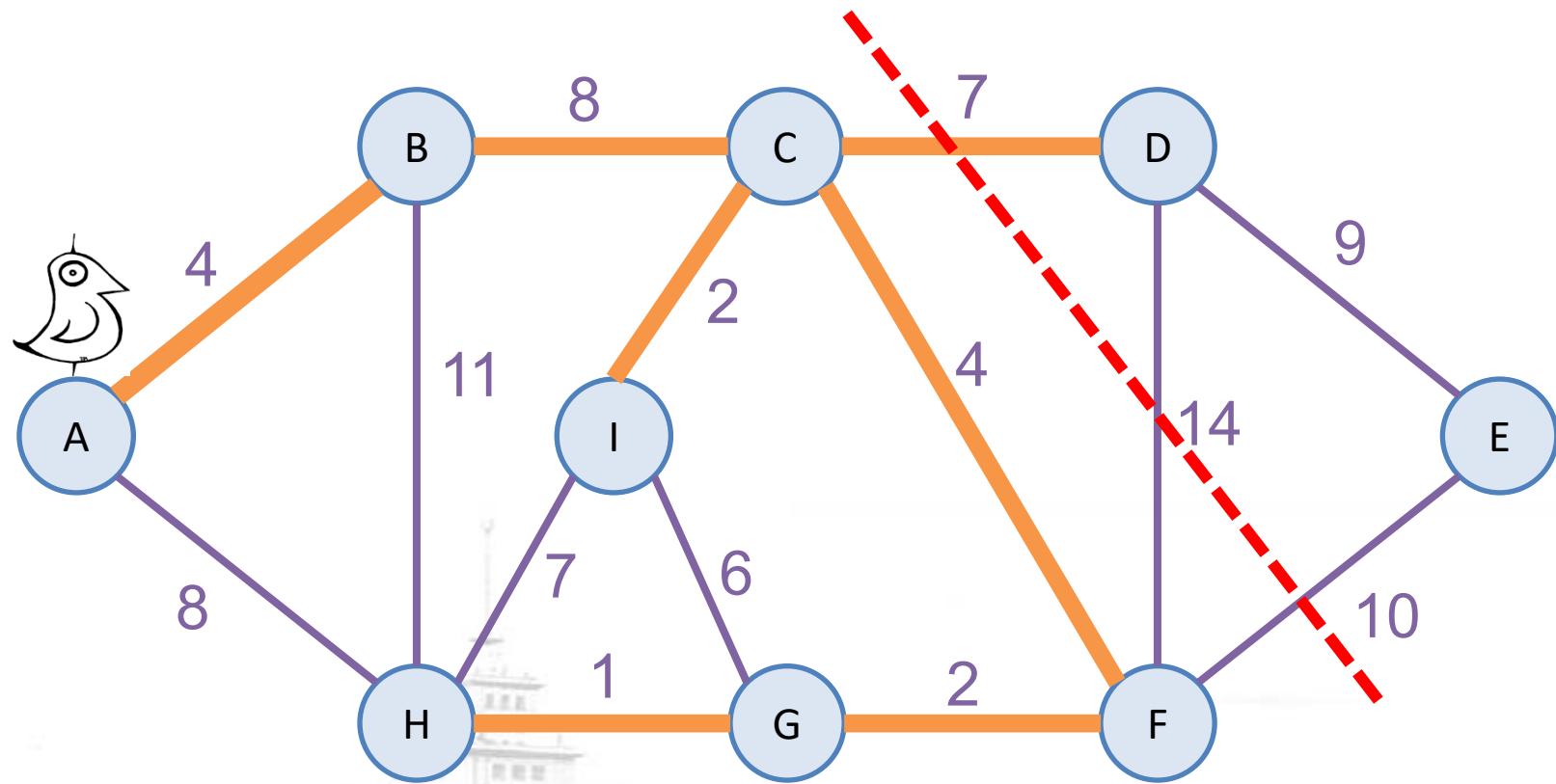
$$S = \{(A, B), (B, C), (C, I), (C, F), (G, F), (H, G)\} \quad (u, v) = (C, D)$$

Prim算法



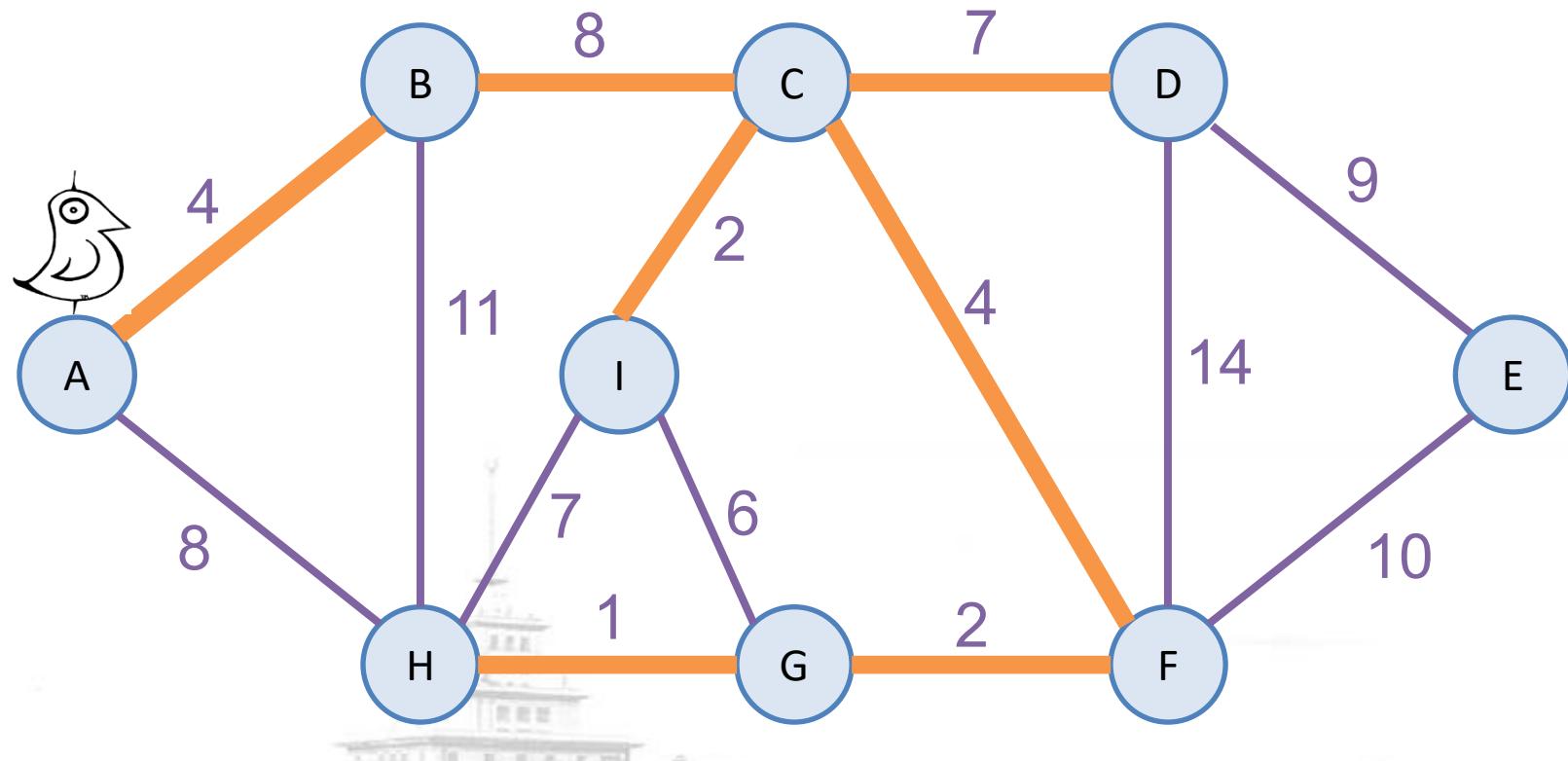
$$S = \{(A, B), (B, C), (C, I), (C, F), (G, F), (H, G)\} \quad (u, v) = (C, D)$$

Prim算法



$$S = \{(A, B), (B, C), (C, I), (C, F), (G, F), (H, G)\} \quad (u, v) = (C, D)$$

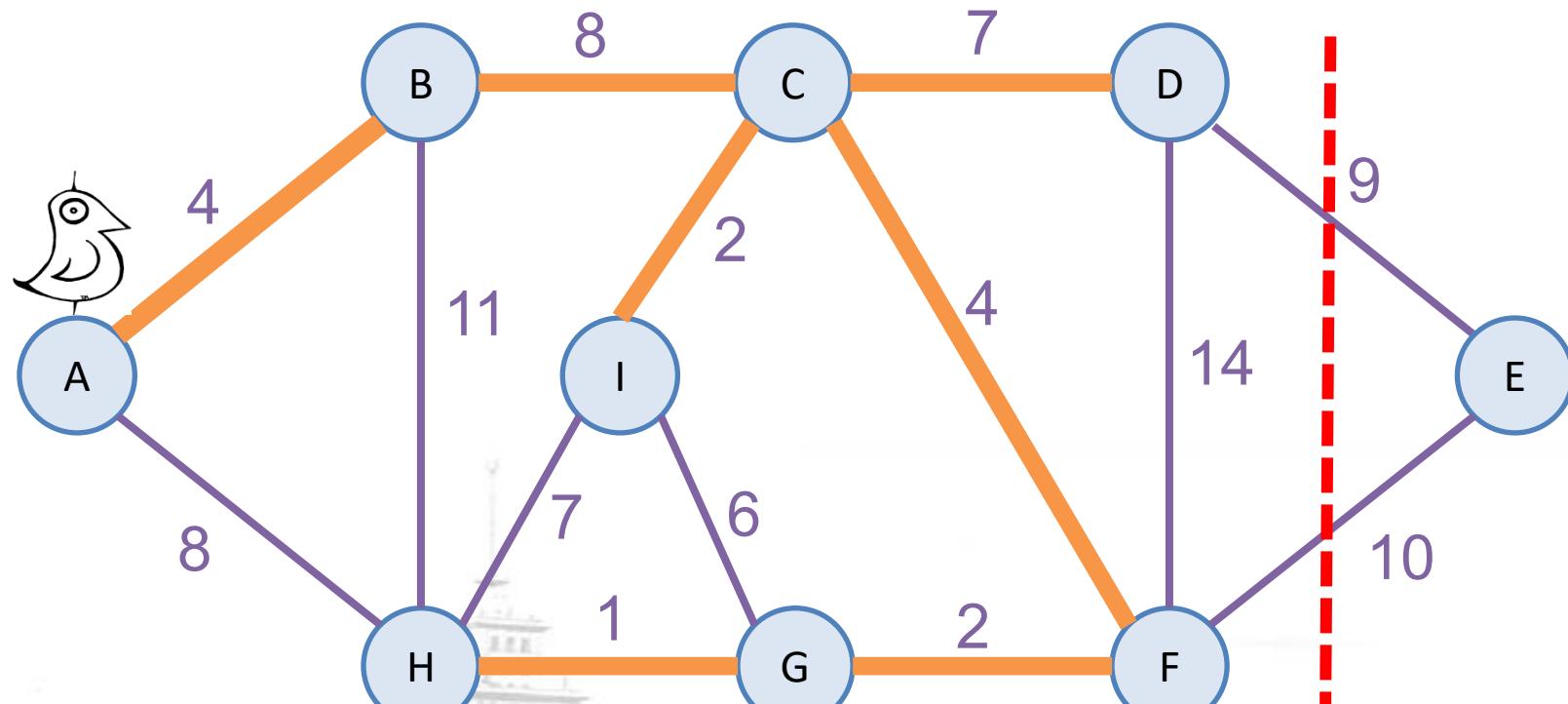
Prim算法



$$S = \{(A, B), (B, C), (C, F), (G, F), (H, G), (C, D)\}$$

$$(u, v) = (D, E)$$

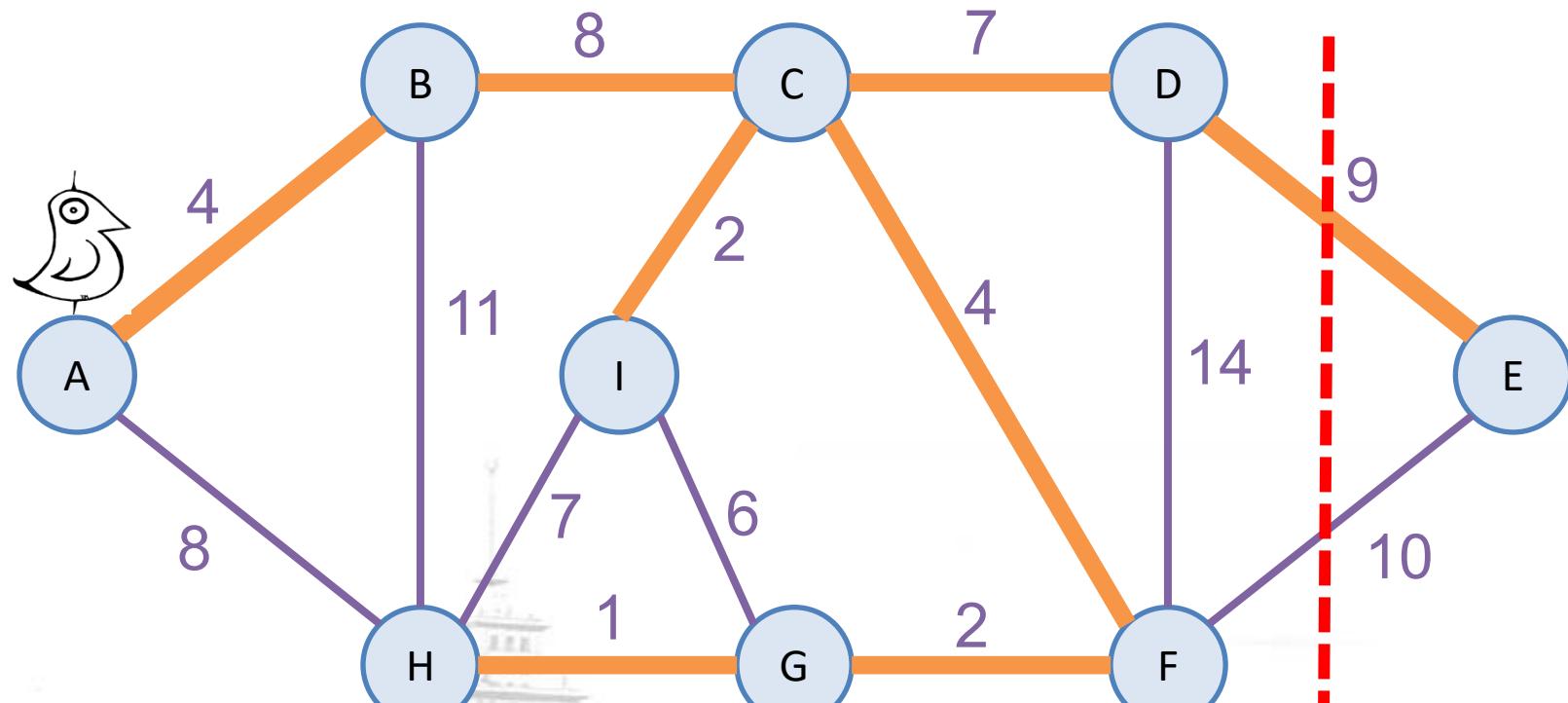
Prim算法



$$S = \{(A, B), (B, C), (C, I), (C, F), (G, F), (H, G), (C, D)\}$$

$$(u, v) = (D, E)$$

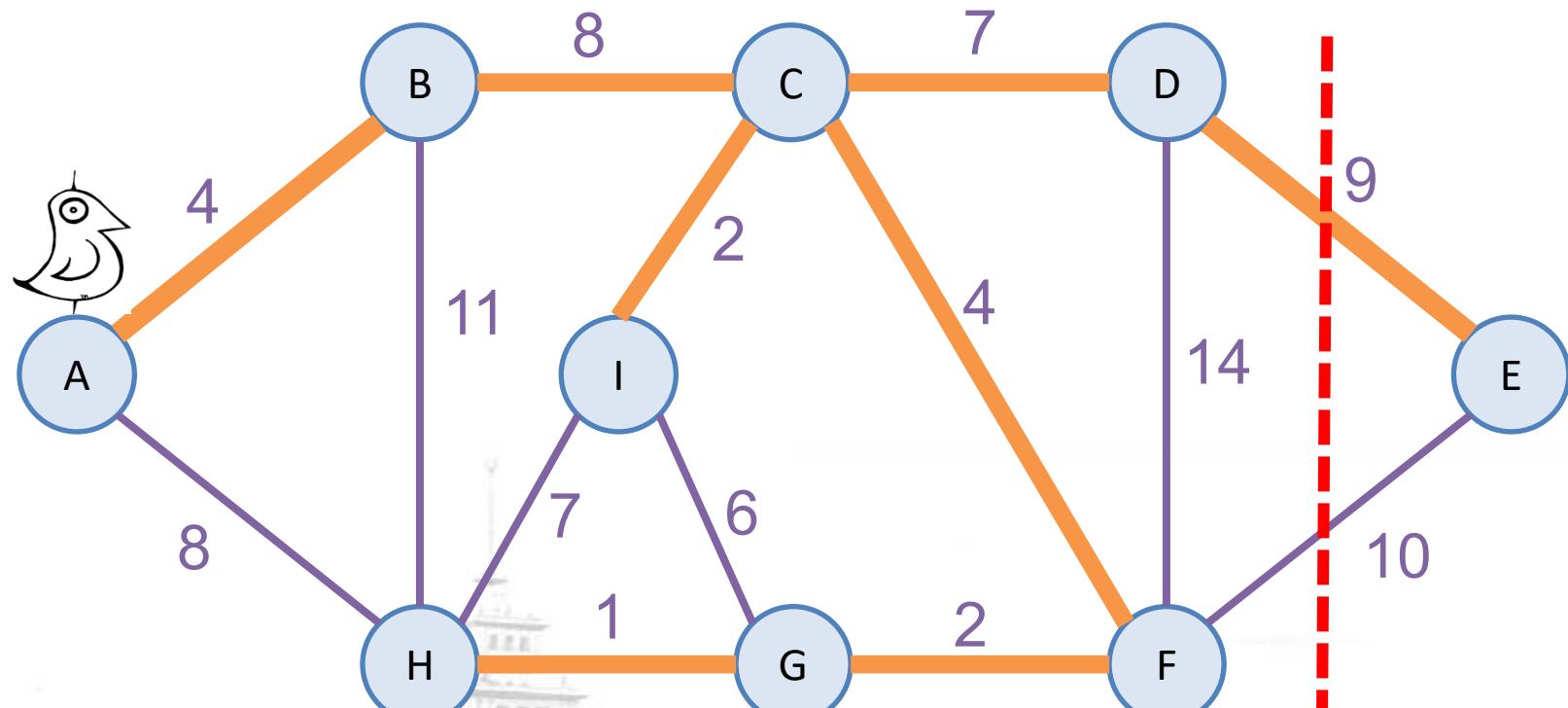
Prim算法



$$S = \{(A, B), (B, C), (C, I), (C, F), (G, F), (H, G), (C, D)\}$$

$$(u, v) = (D, E)$$

Prim算法



逐渐长大直到覆盖整个连通图的所有顶点。

$$S = \{(A, B), (B, C), (C, I), (C, F), (G, F), (H, G), (F, D)\}$$

$$(u, v) = (D, E)$$

算法描述(1)

MST-Prim(G, W, r)

Input 连通图 G , 权值函数 W , 树根 r

Output G 的一棵以 r 为根的生成树

1. $C \leftarrow \{r\}$, $T \leftarrow \emptyset$;
2. 建堆 Q 维护 C 与 $V-C$ 之间的边
3. **While** $C \neq V$ **do**
4. $uv \leftarrow \text{Extract_Min}(Q)$ $// u \in C, v \in V-C$
5. $C \leftarrow C \cup \{v\}$; $T \leftarrow T \cup \{uv\}$;
6. **for** $\forall x \in \text{Adj}[v]$ **do**
7. **if** $x \in C$ **then** 将 vx 从 Q 中删除
8. **Else** 将 vx 插入 Q
9. **Return** T

算法描述(1)

MST-Prim(G, W, r)

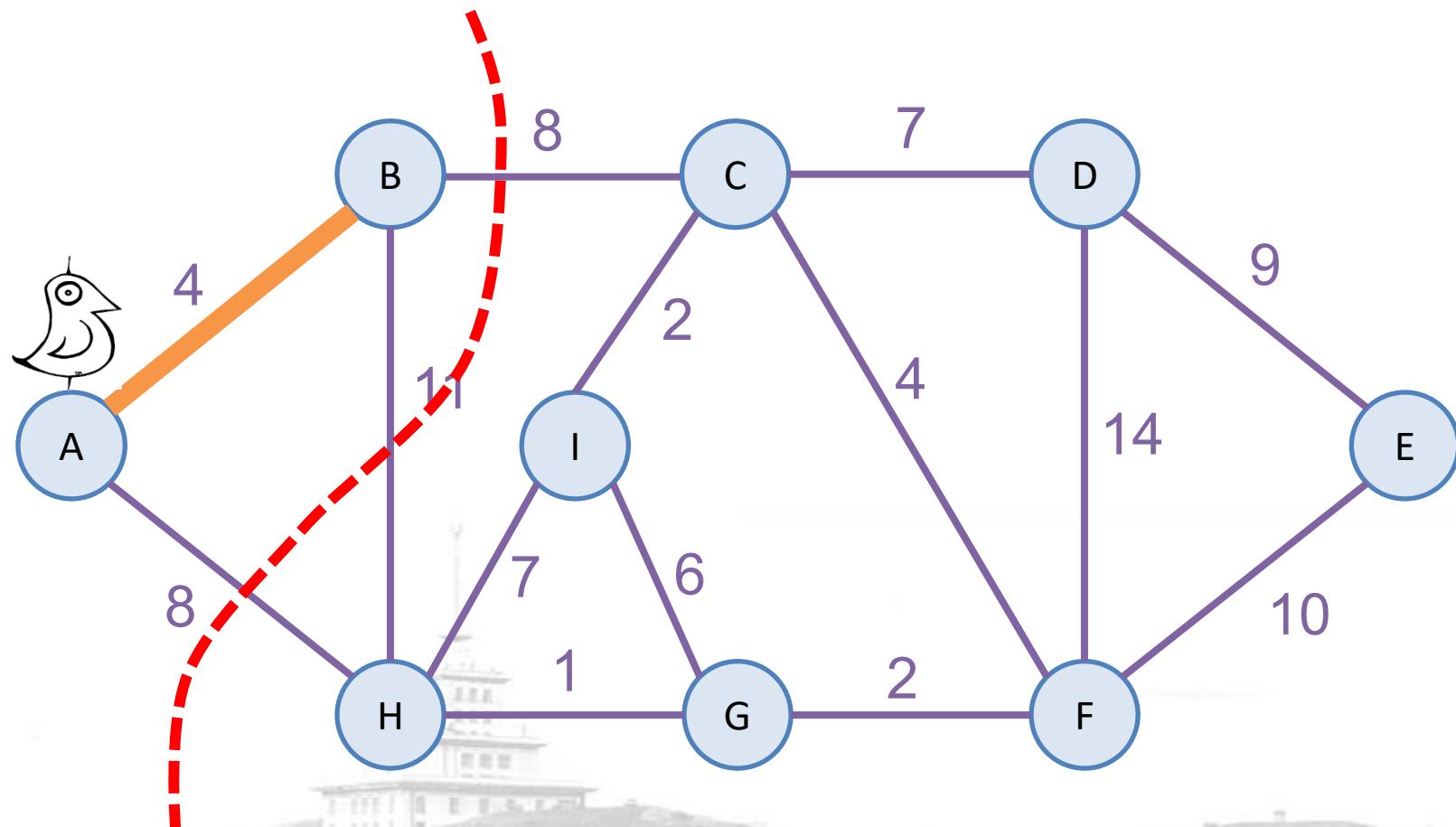
Input 连通图 G , 权值函数 W , 树根 r

Output G 的一棵以 r 为根的生成树

1. $C \leftarrow \{r\}$, $T \leftarrow \emptyset$;
2. 建堆 Q 维护 C 与 $V-C$ 之间的边
3. **While** $C \neq V$ **do**
4. $uv \leftarrow \text{Extract_Min}(Q)$ $// u \in C, v \in V-C$
5. $C \leftarrow C \cup \{v\}$; $T \leftarrow T \cup \{uv\}$;
6. **for** $\forall x \in \text{Adj}[v]$ **do**
7. **if** $x \in C$ **then** 将 vx 从 Q 中删除
8. **Else** 将 vx 插入 Q
9. **Return** T

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

Prim 算法



$$S = \{(A, B)\}$$

$$(u, v) = (B, C)$$

Prim算法

MST-Prim(G, W, r)

Input: 无向连通图 $G = (V, E)$, 权值函数 W , 初始结点 r

Output: G 的最小生成树

1. FOR $\forall v \in V$ DO
2. $key[v] \leftarrow \infty$; /*保存当前联通集合到结点v的最小代价*/
3. $\pi(v) \leftarrow \text{null}$; /*结点v在联通集合中的父节点*/
4. $key[r] \leftarrow 0$; /*保证结点r第一个被处理*/
5. $Q \leftarrow V[G]$; /*最小优先队列*/
6. WHILE $Q \neq \emptyset$ DO
7. $u \leftarrow \text{Extract-Min}(Q)$; /*找到Q中横跨割 $(V-Q, Q)$ 的轻量级边的端点*/
8. FOR $\forall v \in \text{Adj}[u]$ DO /*更新与u邻接的, 但不在树中结点v的key及其父结点*/
9. IF $v \in Q$ and $w(u, v) < key[v]$ Then
10. $\pi(v) \leftarrow u$;
11. $key[v] \leftarrow w(u, v)$; /*更新信息*/
12. Return $A = \{(v, \pi(v)) \mid v \in V - r\}$.

Prim算法

MST-Prim(G, W, r)

Input: 无向连通图 $G = (V, E)$, 权值函数 W , 初始结点 r

Output: G 的最小生成树

```
1. FOR  $\forall v \in V$  DO
2.   key[v]  $\leftarrow \infty$ ; /*保存当前联通集合到结点v的最小代价*/
3.    $\pi(v) \leftarrow \text{null}$ ; /*结点v在联通集合中的父节点*/
4.   key[r]  $\leftarrow 0$ ; /*保证结点r第一个被处理*/
5.   Q  $\leftarrow V[G]$ ; /*最小优先队列*/
6. WHILE  $Q \neq \emptyset$  DO
7.   u  $\leftarrow \text{Extract-Min}(Q)$ ; /*找到Q中横跨割  $(V-Q, Q)$  的轻量级边的端点*/
8.   FOR  $\forall v \in \text{Adj}[u]$  DO /*更新与u邻接的, 但不在树中结点v的key及
   其父结点*/
9.     IF  $v \in Q$  and  $w(u, v) < \text{key}[v]$  Then
10.        $\pi(v) \leftarrow u$ ;
11.       key[v]  $\leftarrow w(u, v)$ ; /*更新信息*/
12. Return A= $\{(v, \pi(v)) \mid v \in V - r\}$ . O(n+nlogn+mlogn)=O(mlogn)
```

Prim算法

MST-Prim(G, W, r)

Input: 无向连通图 $G = (V, E)$, 权值函数 W , 初始结点 r

Output: G 的最小生成树

```
1. FOR  $\forall v \in V$  DO
2.   key[v]  $\leftarrow \infty$ ; /*保存当前联通集合到结点v的最小代价*/
3.    $\pi(v) \leftarrow \text{null}$ ; /*结点v在联通集合中的父节点*/
4.   key[r]  $\leftarrow 0$ ; /*保证结点r第一个被处理*/
5.   Q  $\leftarrow V[G]$ ; /*最小优先队列*/
6. WHILE  $Q \neq \emptyset$  DO
7.   u  $\leftarrow \text{Extract-Min}(Q)$ ; /*找到Q中横跨割  $(V-Q, Q)$  的轻量级边的端点*/
8.   FOR  $\forall v \in \text{Adj}[u]$  DO /*更新与u邻接的, 但不在树中结点v的key及
   其父结点*/
9.     IF  $v \in Q$  and  $w(u, v) < \text{key}[v]$  Then
10.        $\pi(v) \leftarrow u$ ;
11.       key[v]  $\leftarrow w(u, v)$ ; /*更新信息*/
12. Return A= $\{(v, \pi(v)) \mid v \in V - r\}$ . O(n+nlogn+mlogn)=O(mlogn)
```

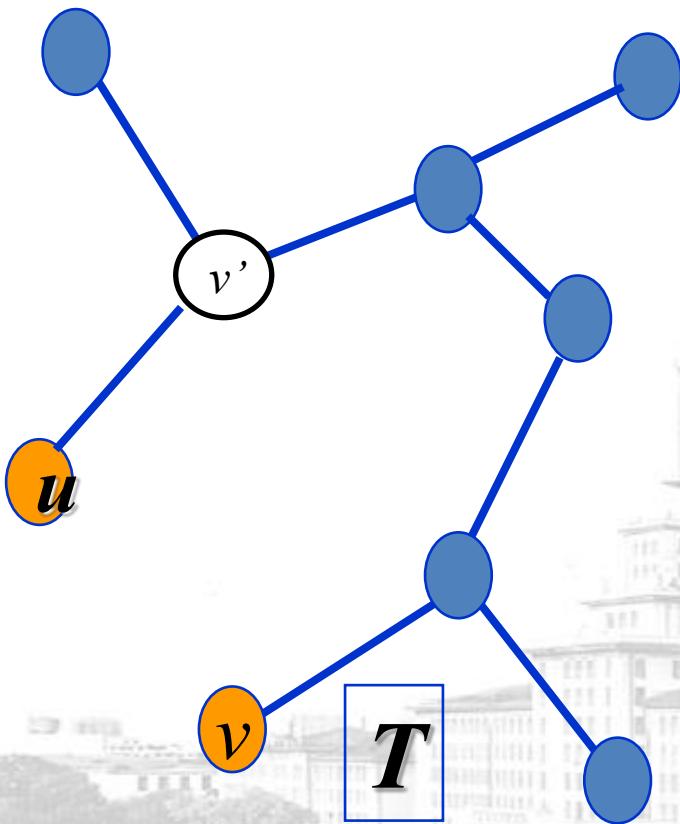
斐波那契堆

操作	二项堆	斐波那契堆
Make-Heap	$O(n \lg n)$	$O(n \lg n)$
Insert	$O(\lg n)$	$O(1)$
Minimum	$O(1)$	$O(1)$
Extract-Min	$O(\lg n)$	$O(\lg n)$
Decrease-Key	$O(\lg n)$	$O(1)$
Delete	$O(\lg n)$	$O(\lg n)$



贪心选择性

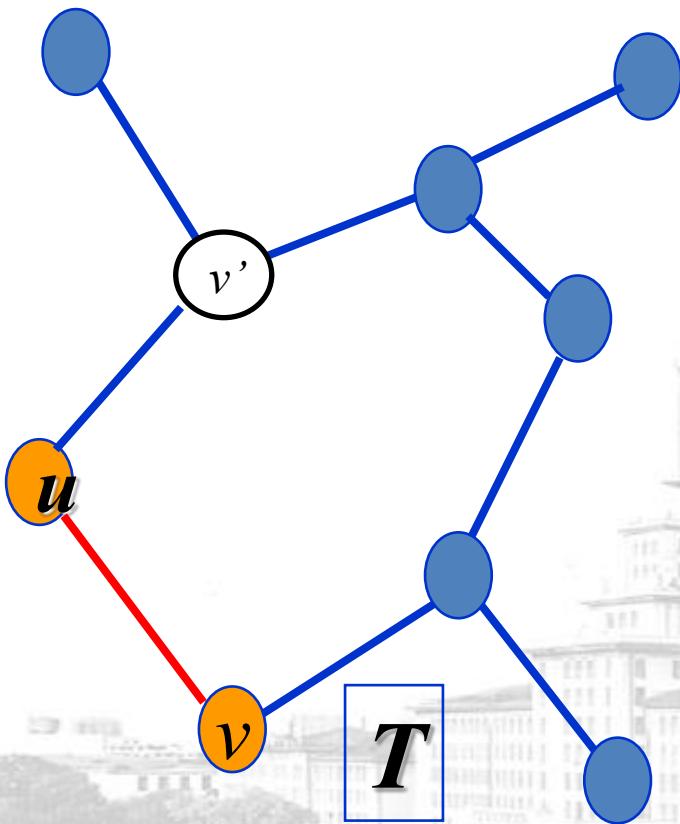
定理1. 设 uv 是 G 中与顶点 u 关联的权值最小的边，则必有一棵最小生成树包含边 uv .



证明：设 T 是 G 的一棵MST
若 $uv \in T$, 结论成立;
否则, 如左图所示

贪心选择性

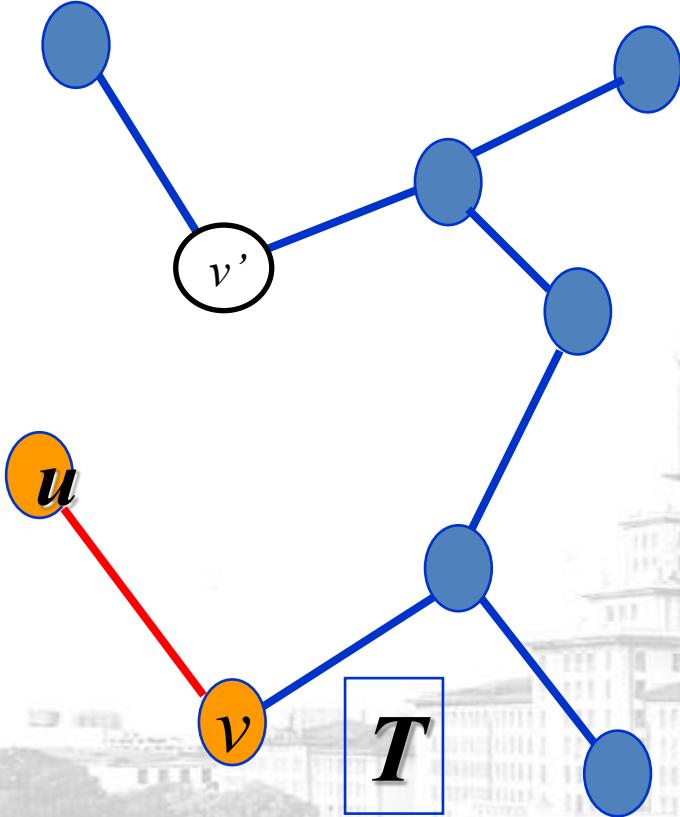
定理1. 设 uv 是 G 中与顶点 u 关联的权值最小的边，则必有一棵最小生成树包含边 uv .



证明：设 T 是 G 的一棵MST
若 $uv \in T$, 结论成立;
否则, 如左图所示

贪心选择性

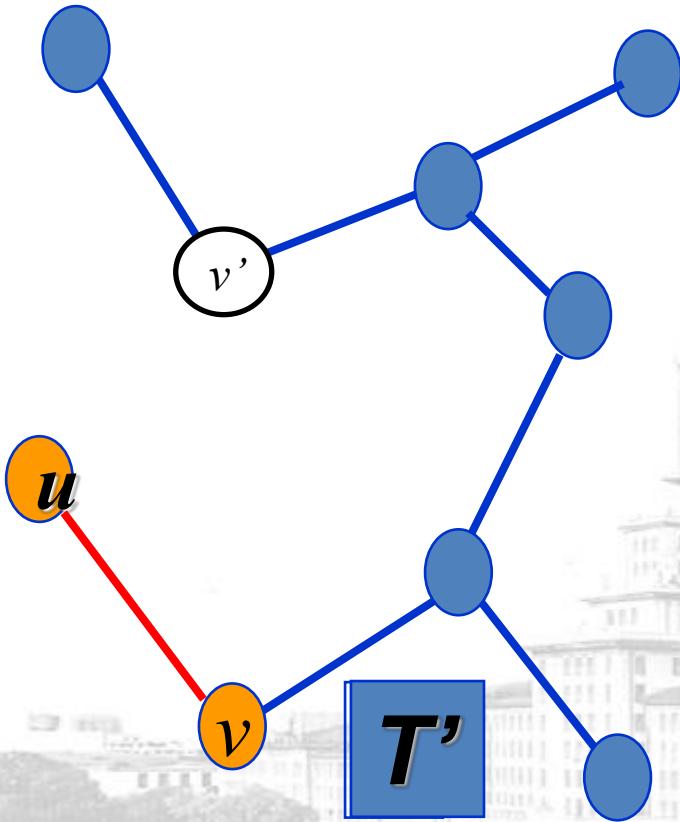
定理1. 设 uv 是 G 中与顶点 u 关联的权值最小的边，则必有一棵最小生成树包含边 uv .



证明：设 T 是 G 的一棵MST
若 $uv \in T$, 结论成立;
否则, 如左图所示

贪心选择性

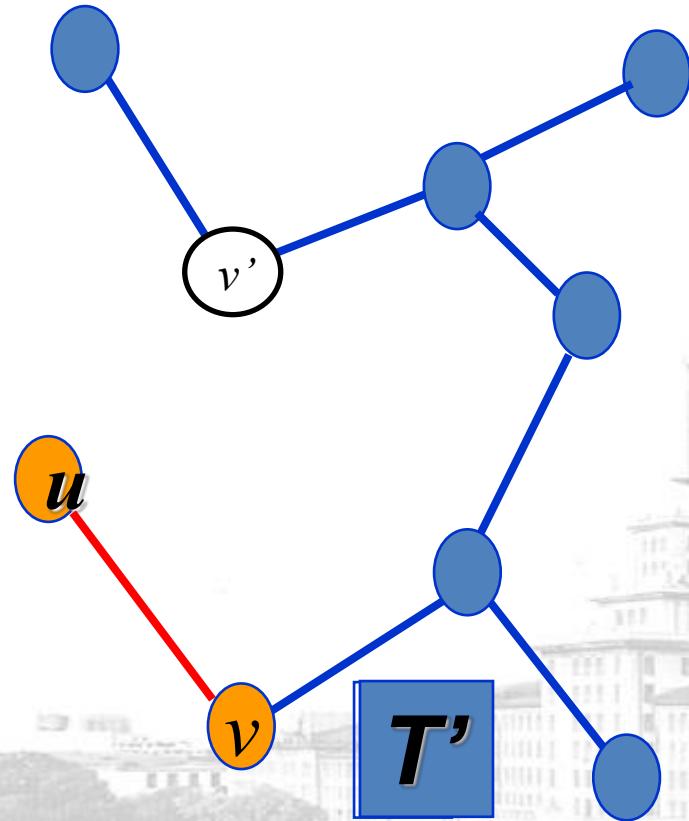
定理1. 设 uv 是 G 中与顶点 u 关联的权值最小的边，则必有一棵最小生成树包含边 uv .



证明：设 T 是 G 的一棵MST
若 $uv \in T$, 结论成立;
否则, 如左图所示

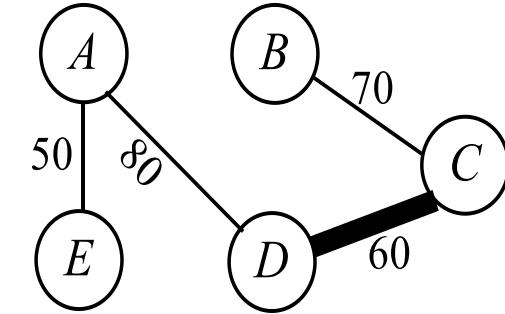
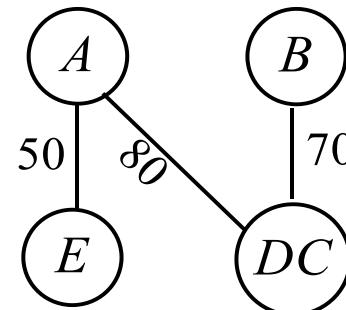
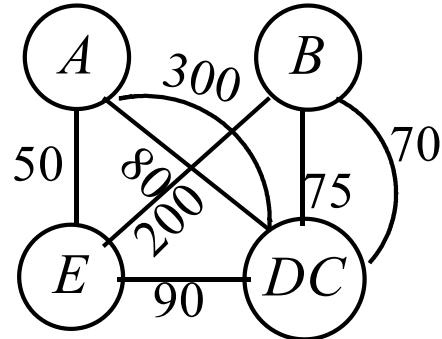
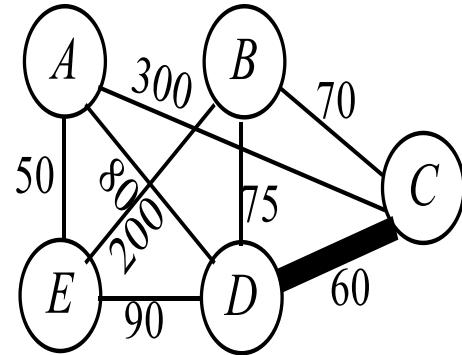
贪心选择性

定理1. 设 uv 是 G 中与顶点 u 关联的权值最小的边，则必有一棵最小生成树包含边 uv .



证明：设 T 是 G 的一棵MST
若 $uv \in T$, 结论成立;
否则, 如左图所示
在 T 中添加 uv 边, 产生环, 环中顶点 u 的度为 2, 即存在 $uv' \in T$.
删除环中边 uv' , 得到 T' .
 $w(T') = w(T) - w(uv') + w(uv)$
 $\leq w(T)$

优化子结构



收缩图 G 的边 uv — $G \bullet uv$

- 用新顶点 C_{uv} 代替边 uv
- 将 G 中原来与 u 或 v 关联的边与 C_{uv} 关联
- 删除 C_{uv} 到其自身的边

上述操作的逆操作称为扩张

定理2.给定加权无向连通图 $G=(V,E)$,权值函数为 $W:E\rightarrow R$, $uv\in E$ 是 G 中顶点 u 关联的权值最小的边。设 T 是 G 的包含 uv 的一棵最小生成树，则 $T\cdot uv$ 是 $G.uv$ 的一棵最小生成树。

证明. 同Kruskal算法优化子结构的证明。



算法正确性

定理2. MST-Prim(G, W) 算法能够产生图 G 的最小生成树.

证. 因为算法按照贪心选择性进行局部优化选择.



证明：n表示图G中顶点集合V中的点的个数

1、 $n=2$ 时，只有两个点。生成树只有一种，根据prim算法产生的生成树必定是最小生成树。

2、假设 $n < k$ 时，通过prim算法得到的生成树是最小生成树。

当 $n=k$ 时，考虑如下情况。不失一般性，设我们从某个顶点u开始建树，且 (u,v) 是图G中与u相连的最短边。那么根据贪心选择性，存在一颗G的最小生成树T包含边 (u,v) 。

(1) 收缩 (u,v) ，得到树 T_{uv} 和图 G_{uv} 。根据优化子结构， T_{uv} 是 G_{uv} 的最小生成树。

(2) 因为 G_{uv} 的顶点数 $n=k-1 < k$ ，根据假设，存在一颗由prim算法建树的 T' 是 G_{uv} 的最小生成树。

(3) 讨论 T' 和 T_{uv} 。若 T_{uv} 跟 T' 一致，那么通过prim算法产生的生成树就是G的最小生成树。若不然，我们用 T' 替换 T_{uv} ，对 T' 扩展 uv ，得到 T'' 。显然 T'' 是G的一颗通过prim算法建树的生成树。

$$W(T'') = W(T') + W(u,v) = W(T_{uv}) + W(u,v) = W(T)$$

通过prim算法构造的 T'' 是一颗G的最小生成树

Prim vs Kruskal (复杂度)

- Prim算法:
- Kruskal算法:

并查集: $O(m \log m)$



Prim vs Kruskal (复杂度)

- Prim算法:
二叉最小堆: $O(m \log n)$, 斐波那契堆: $O(n \log n + m)$
- Kruskal算法:
并查集: $O(m \log m)$

使用场景:

稀疏图: $m=O(n)$, 使用kruskal, 三者都是 $n \log n$ 的复杂度, kruskal的数据结构更简单, 速度快, 且实现简单。

稠密图: $m=O(n^2)$, 使用prim算法, prim算法复杂度低。