

找零问题 · 一

Coin Changing (1)

刘 铎

liuduo@bjtu.edu.cn



找零问题 (1, 5, 10)



- 假定有面值分别为1元、5元、10元的三种硬币，各自有**足够多个**。设计一种方法，用**最少**的硬币数给顾客支付指定金额
- 例如：设法凑38元



找零问题 —— 收银员算法



- 收银员算法：在每次迭代中，都选择不会超过还需要支付给顾客的金额的、面值最大的硬币

找零问题 —— 收银员算法



- 一般情形:

m 种面值的硬币, 每种都有足够多枚, 凑 n 元

对硬币面值进行排序 使得 $c_1 < c_2 < \dots < c_m$

1. $S \leftarrow \emptyset$ ← 多重集合, 表示所选择的硬币
2. **while** ($n \neq 0$) **do**
3. 令 k 表示满足 $c_k \leq n$ 的最大值
4. **if** ($k = 0$) **then**
 //意即找不到可以找给顾客的硬币
5. **return** "no solution found"
6. $n \leftarrow n - c_k$
7. $S \leftarrow S \cup \{k\}$
8. **return** S

找零问题 —— 收银员算法



- 这些步骤可以看作是人类对贪婪策略的一种模仿
 - 求解过程是多步判断过程
 - 在每一步，它总是做出目前看起来最好的可行选择
 - 进行局部最优选择，以期得到全局最优解
- 选择不多于剩余的零钱的价值最高的硬币
- 问题：收银员算法是否最优？

找零问题 —— 收银员算法



- 定理：对于面值10、5和1的情况，收银员算法可以得到最优解。
- 证明：
 - 策略：将最优解与收银员算法（贪婪算法）给出的解进行（逐分量）比较，并使用反证法通过“替换”寻找矛盾
 - 该策略可用于许多问题
 - 设最优解为 (x, y, z) ，表示：使用 x 个面值为10的硬币、 y 个面值为5的硬币和 z 个面值为1的硬币
 - 设由收银员算法得到的解为 (x', y', z') ，表示：使用 x' 个面值为10的硬币、 y' 个面值为5的硬币和 z' 个面值为1的硬币
 - 之后只需证明 $x + y + z = x' + y' + z'$ 即可
 - 事实上是证明了这两个解是相同的

找零问题 —— 收银员算法



	Greedy	OPT
10元硬币	x'	x
5元硬币	y'	y
1元硬币	z'	z
总面值	$10x' + 5y' + z'$	$10x + 5y + z$

- 先比较 x 和 x'
- 证明了 $x = x'$ 之后再比较 y 和 y'
- 再完成此证明后，最后比较 z 和 z'

- 首先比较 x 和 x'
- 考虑到贪婪算法的流程，所以**必定**有 $x' \geq x$
- 如果 $x' > x$ （即 $x \neq x'$ ），那么
 - 由 $10x + 5y + z = 10x' + 5y' + z'$
可得
 $5y + z = 10(x' - x) + (5y' + z')$
 ≥ 10

找零问题 —— 收银员算法



	Greedy	OPT
10元硬币	x'	x
5元硬币	y'	y
1元硬币	z'	z
总面值	$10x' + 5y' + z'$	$10x + 5y + z$

- 先比较 x 和 x'
- 证明了 $x = x'$ 之后再比较 y 和 y'
- 再完成此证明后，最后比较 z 和 z'

- 如果 $x' > x$ (即 $x \neq x'$) , 那么
$$5y + z \geq 10$$
- 三种可能:
 - (1) $y \geq 2$,
此时可以将 2 个 5 元硬币替换为 1 个 10 元硬币, 硬币总数减少 1
 - (2) $y = 1$
此时可以将 1 个 5 元硬币和 5 个 1 元硬币替换为 1 个 10 元硬币, 硬币总数减少 5
 - (3) $y = 0$
此时可以将 10 个 1 元硬币替换为 1 个 10 元硬币, 硬币总数减少 9
- 无论如何都与“最优解”产生矛盾

找零问题 —— 收银员算法



	Greedy	OPT
10元硬币	x'	x
5元硬币	y'	y
1元硬币	z'	z
总面值	$10x' + 5y' + z'$	$10x + 5y + z$

- 于是 “ $x' > x$ ” 不成立
- 即 $x = x'$ 成立
- 之后类似地可以证明: $y = y'$
- 再由

$$10x + 5y + z = 10x' + 5y' + z'$$

- 即可得到 $z = z'$

- 先比较 x 和 x'
- 证明了 $x = x'$ 之后再比较 y 和 y'
- 再完成此证明后, 最后比较 z 和 z'

找零问题 —— 收银员算法



- 如果现在有面值分别为1元、5元、7元、10元的四种硬币呢？
 - 依然假定各自有足够多个
- 例如要凑出 19 元
 - 收银员算法： $1 \times 10 + 1 \times 7 + 2 \times 1 = 19$
 - 共需要4个硬币
 - 但存在更好的方法： $2 \times 7 + 1 \times 5 = 19$
 - 共需要3个硬币
- 有时贪婪算法并不能保证得到最优解

找零问题 —— 收银员算法



- 更一般的找零问题？
 - 可以使用动态规划（dynamic programming）
或者整数规划（integer programming）

贪婪策略

/贪婪算法/贪心算法/贪心策略

刘 铎

liuduo@bjtu.edu.cn



找零问题 —— 收银员算法



● 一般情形

对硬币面值进行排序 使得 $c_1 < c_2 < \dots < c_m$

1. $S \leftarrow \emptyset$

2. **while** ($n \neq 0$) **do**

3. 令 k 表示满足 $c_k \leq n$ 的最大值

4. **if** ($k = 0$) **then**

 //意即找不到可以找给顾客的硬币

5. **return** "no solution found"

6. $n \leftarrow n - c_k$

7. $S \leftarrow S \cup \{k\}$

8. **return** S

从上一个 k 开始降序寻找即可

找零问题 —— 收银员算法



- 贪婪策略的特点（优势/不足）
 - 简单、快速
 - 有时贪婪算法可以得到最优解
 - 需要进行正确性证明
 - 有时贪婪算法**并不能**保证得到最优解
 - 局部最优并不总能“走向”全局最优
 - 例如棋牌游戏/比赛中，短期的牺牲可能带来长期的受益
 - 但可以快速得到一个可行解或近似解

贪婪算法/贪心算法/贪婪策略



- 它的求解过程是多步判断过程
- 在每一步，它总是做出目前看起来最好的选择
- 它进行局部最优选择，以期得到全局最优解

贪婪策略

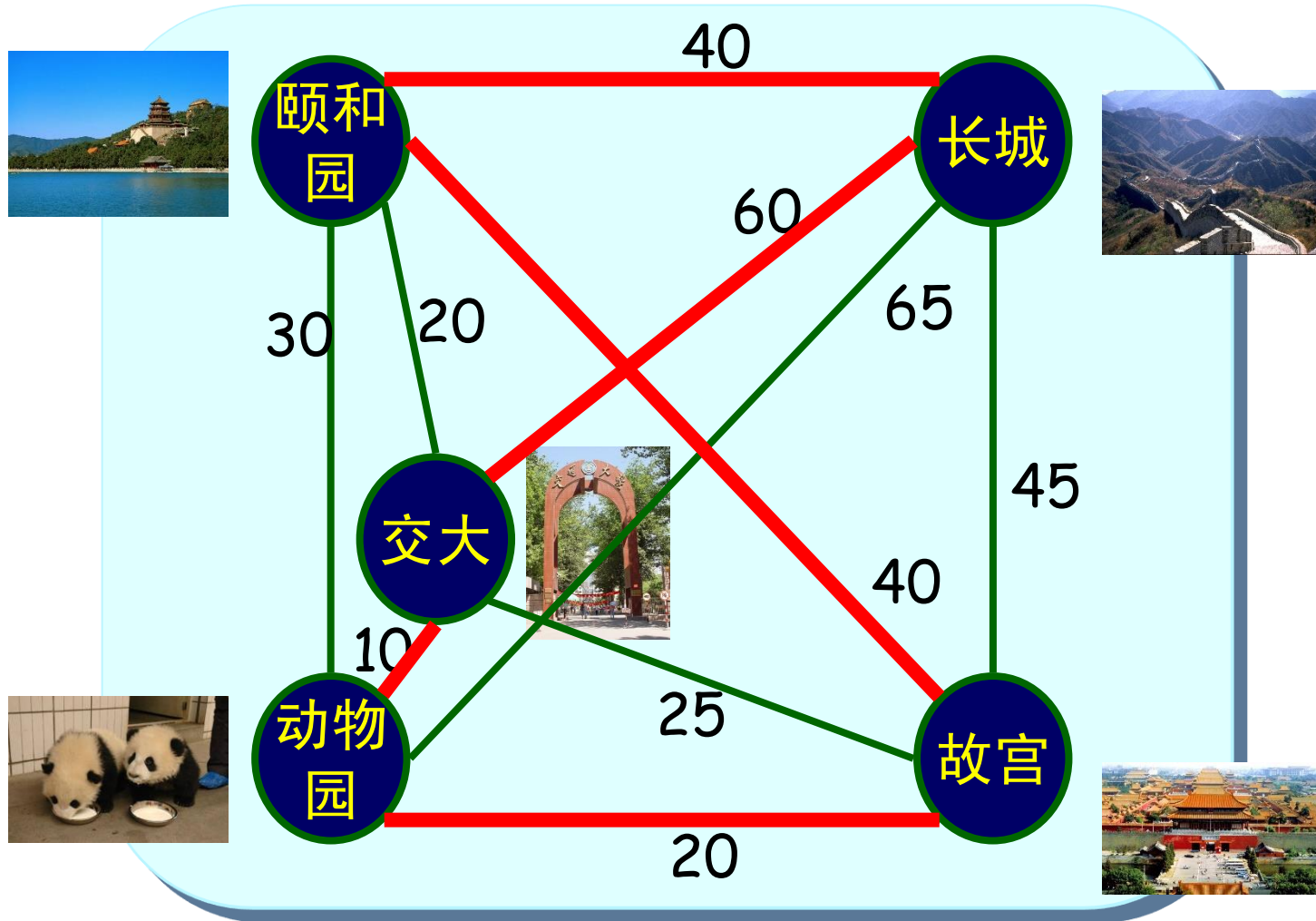


- 贪婪策略示例一：
 - 从 n 个给定的数值中选择 k 个数值，使这些 k 个数值的总和最大

示例二：旅行商问题

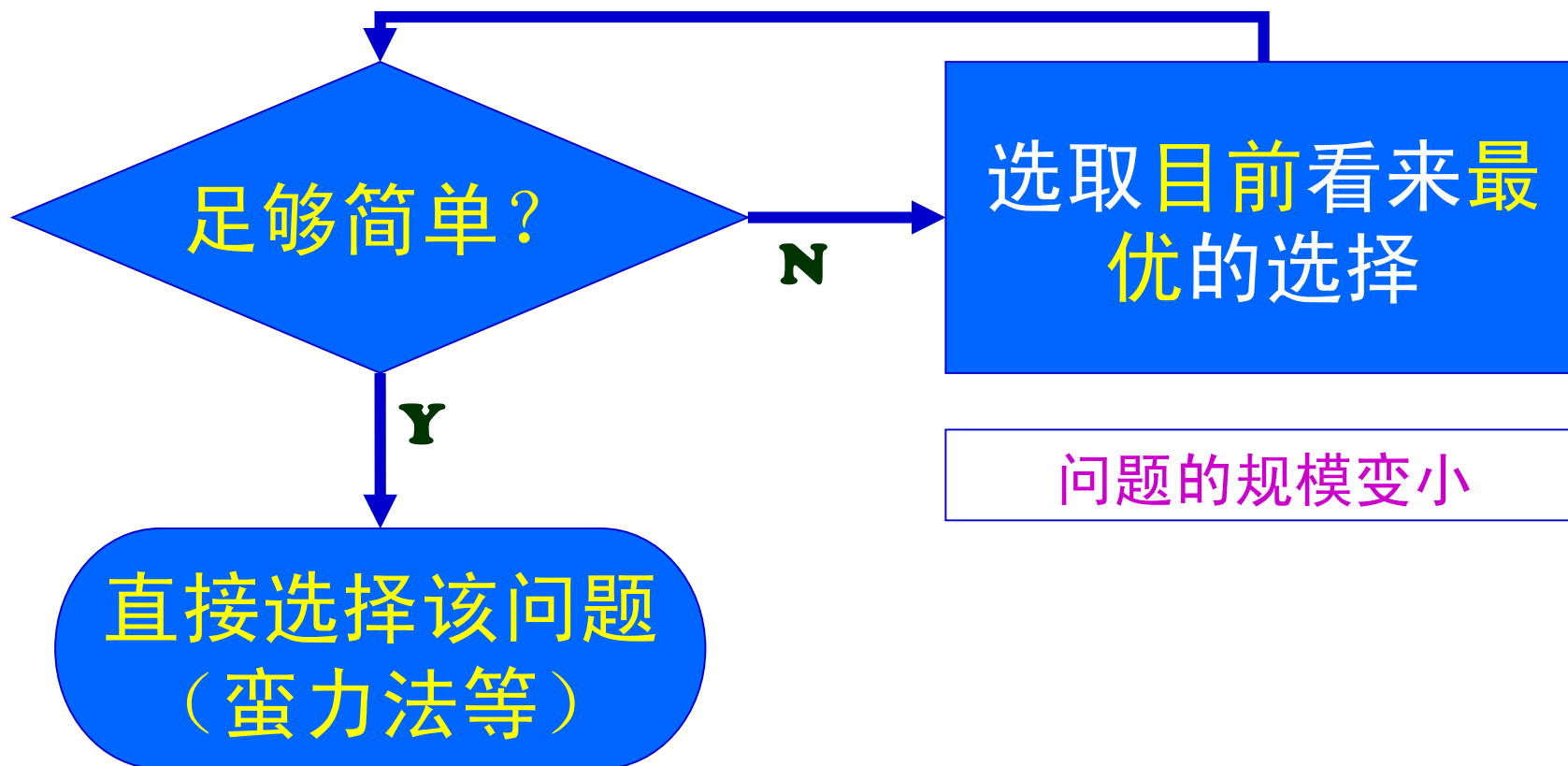


- **旅行商问题（Traveling Salesman Problem, TSP）**，
又称作**旅行推销员问题、货郎担问题**
 - 有一个推销员，要到 n 个城市推销商品，这 n 个城市两两之间的距离是已知的，他希望找到一条**最短**的路线，**走遍所有的城市**，最后再**回到**他出发的城市
 - 使用图论的语言描述就是：给定一个权值都为正数的赋权完全图，求各边权值和最小的哈密顿回路



贪婪策略
¥170

贪婪策略



贪婪策略的应用实例



- 可以得到最优解：
 - 找零问题的部分实例
 - 最小支撑树（Minimum Spanning Tree, MST）
 - 单源最短道路问题
 - Huffman编码
 -
- 不能保证得到最优解：
 - 旅行商问题（Traveling Salesman Problem, TSP）
 - 背包问题
 -

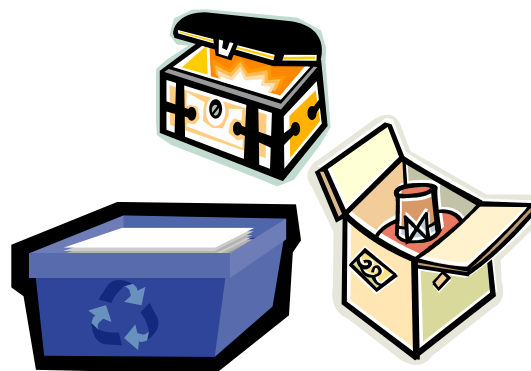
装载问题 · 一

Packing Problem (I)



刘 铎

liuduo@bjtu.edu.cn



装载问题



- 共有 n 个集装箱 $1, 2, \dots, n$
- 可以选择若干装上轮船
- 集装箱 i 的重量 w_i , 轮船装载重量限制为 C
 - 为简化讨论, 假定 w_i 彼此不同
- (不考虑体积等其他限制) 如何进行选择可以使得装上船的集装箱**总个数最多**?
 - 不妨设对所有 i 都有 $w_i \leq C$

装载问题



- 即寻找集合 $U \subseteq \{1, 2, \dots, n\}$
- 满足 $\sum_{u \in U} w(u) \leq C$
- 使得 $|U|$ 最大化
- 是后文介绍的“背包问题”的特例
 - 相当于各个物品的价值均为 1

装载问题



- 如何进行选择可以使得装上船的集装箱**总个数最多**？
- $n = 3, C = 5, w_1 = 1, w_2 = 2, w_3 = 3$
- 最优解
 - 物品 1 和 2 ， 或者
 - 物品 1 和 3 ， 或者
 - 物品 2 和 3

装载问题



- 贪婪策略
 - 将集装箱按照从轻到重排序，轻者先装
- 定理：上述贪婪策略是最优的。
- 证明：（大体思路）
 - 考虑最优解和贪婪解
 - 进行比较
 - 那么可能有多个最优解怎么办？

装载问题



- $n = 3, C = 5, w_1 = 1, w_2 = 2, w_3 = 3$

- 最优解

- 物品 1 和 2 , 或者
- 物品 1 和 3 , 或者
- 物品 2 和 3

可能有多**个**最优解怎么办?

那么就选择和贪婪解最**“相近”** 的

和贪婪解最“相近”的最优解



- 设

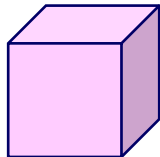
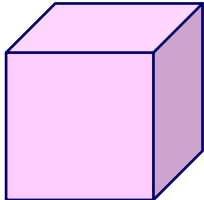
- S_G 为贪婪解，包含 k 个集装箱，重量以递增顺序排列为 $w_{G,1}, w_{G,2}, \dots, w_{G,k}$
- S_O 为一个最优解，包含 m 个集装箱，重量以递增顺序排列为 $w_{O,1}, w_{O,2}, \dots, w_{O,m}$
 - 显然有 $k \leq m$
- S_G 和 S_O 的相似度定义为
$$\max\{ i \mid 1 \leq i \leq k, 1 \leq i \leq k, \text{对所有 } 1 \leq j \leq i \text{ 都有 } w_{G,j} = w_{O,j} \}$$
 - 若 $w_{G,1} \neq w_{O,1}$ 则定义 S_G 和 S_O 的相似度定义为 0

装载问题



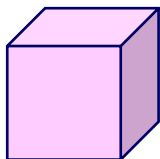
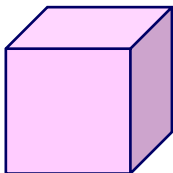
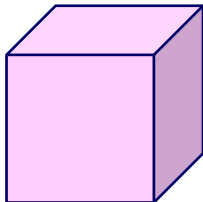
- 证明:

- 比较贪婪解和（最像贪婪解的）的最优解

Greedy: 1 2 3 4 5  ... 

最“像”贪婪解
的最优解

OPT':

1 2 3 4 5   ... 

更“像”

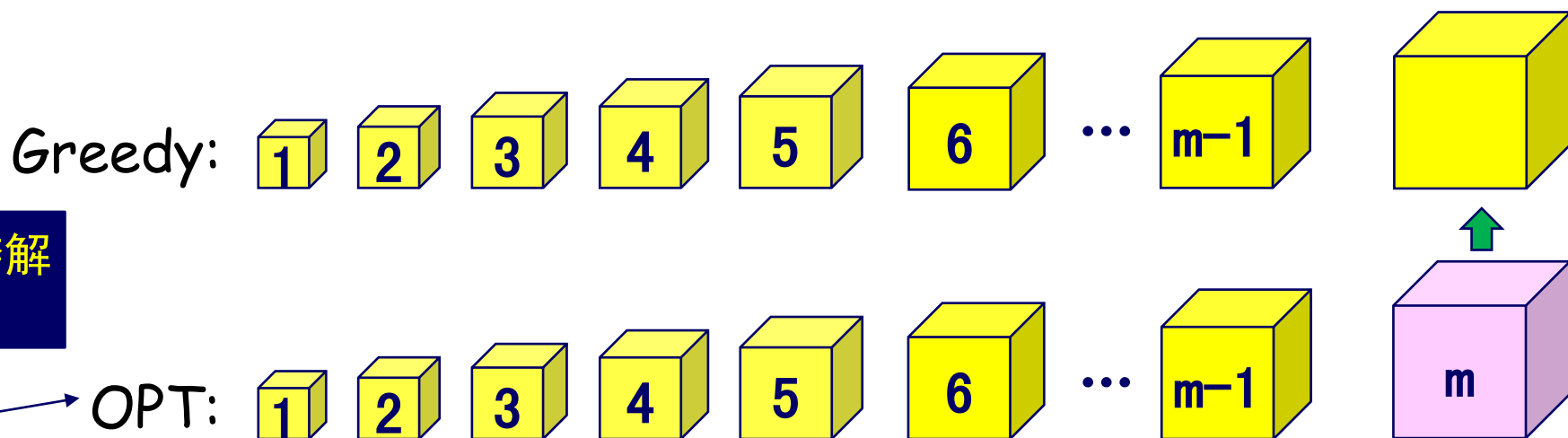
先不用比较

装载问题



- 证明:

- 比较贪婪解和（最像贪婪解的）的最优解



表明贪婪解的箱子个数不会比最“像”贪婪解的最优解的箱子个数少

当然也不会多

综合上一页得到：贪婪解和最“像”贪婪解的最优解完全相同

一般的装载问题



- 共有 n 个集装箱 $1, 2, \dots, n$
- 可以选择若干装上轮船
- 集装箱 i 的重量 w_i , 轮船装载重量限制为 C
 - 不妨设对所有 i 都有 $w_i \leq C$
- （不考虑体积等其他限制）如何进行选择可以使得装上船的集装箱**总重量最大**？

一般的装载问题



- 即寻找集合 $U \subseteq \{1, 2, \dots, n\}$
- 满足 $\sum_{u \in U} w(u) \leq C$
- 使得 $\sum_{u \in U} w(u)$ 最大化
- 是后文介绍的“背包问题”的特例
 - 相当于各个物品的价值均等于重量

一般的装载问题



- 即寻找集合 $U \subseteq \{1, 2, \dots, n\}$
- 满足 $\sum_{u \in U} w(u) \leq C$
- 使得 $\sum_{u \in U} w(u)$ 最大化
- 例如 $n = 3, C = 5, w_1 = 1, w_2 = 2, w_3 = 3$
- 此时前述“贪婪策略”不能得到最优解

活动选择问题

Interval Scheduling

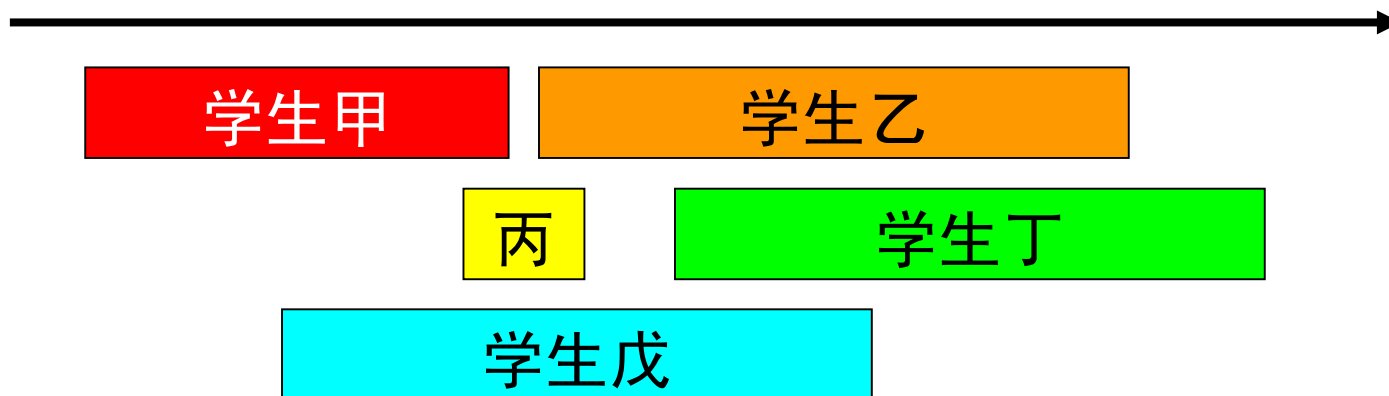


活动选择问题



- 问题描述:

- 有很多同学报名参加值班工作，帮助刘老师
- 每个同学都有自己可能值班的时间段
- 但是由于客观原因所限，任何一个时刻，只能容纳一名值班同学
- 刘老师其实没有什么让他们干的，但是希望能有尽可能多的同学来值班，可以监督他们学习，也能让他们挣一些零花钱



活动选择问题

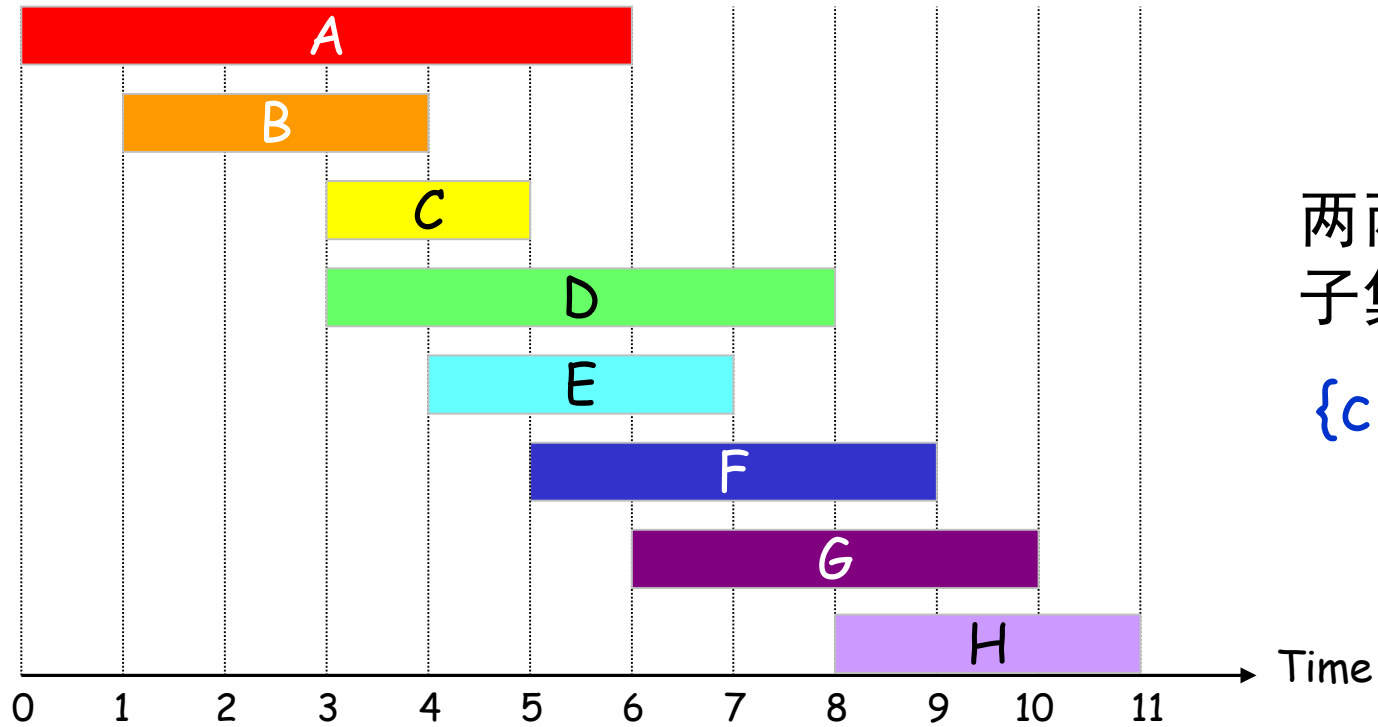


- n 项活动的集合 $S = \{1, 2, \dots, n\}$
- 活动 j 开始于时刻 s_j , 结束于时刻 f_j
 - $f_j - s_j > 0$
- 称两个活动**相容**, 指的是它们在时间上没有 (大于0的) 重叠
- 目标: 所有活动的两两相容的**最大**子集
 - 包含满足条件的活动数目最大
 - 可能不唯一

活动选择问题



- 示例:



两两相容活动
子集之一:

$\{c, f\}$

活动选择问题 —— 贪婪策略



- 【最早开始时刻优先】按开始时刻 s_j 的升序考虑各个活动
- 【最早结束时刻优先】按结束时刻 f_j 的升序考虑各个活动
- 【最短持续时间优先】按持续时间的长度 $f_j - s_j$ 的升序考虑各个活动
- 【最少冲突数量优先】对于每个活动 j ，计算与之冲突的活动的数目 c_j 。按冲突活动数量的升序考虑各个活动

活动选择问题



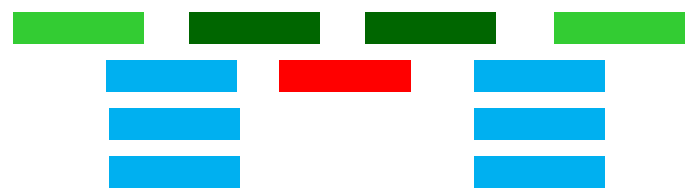
- 反例



【最早开始时刻优先】 ❌



【最短持续时间优先】 ❌



【最少冲突优先】 ❌

活动选择问题



● 【最早结束时刻优先】

1. 对活动进行排序 使得 $f_1 \leq f_2 \leq \dots \leq f_n$

$O(n \log n)$

2. $A \leftarrow \emptyset$

3. **for** $j = 1$ **to** n **do**

4. **if** (活动 j 与 A 中所有活动都相容)

5. $A \leftarrow A \cup \{j\}$

6. **return** A

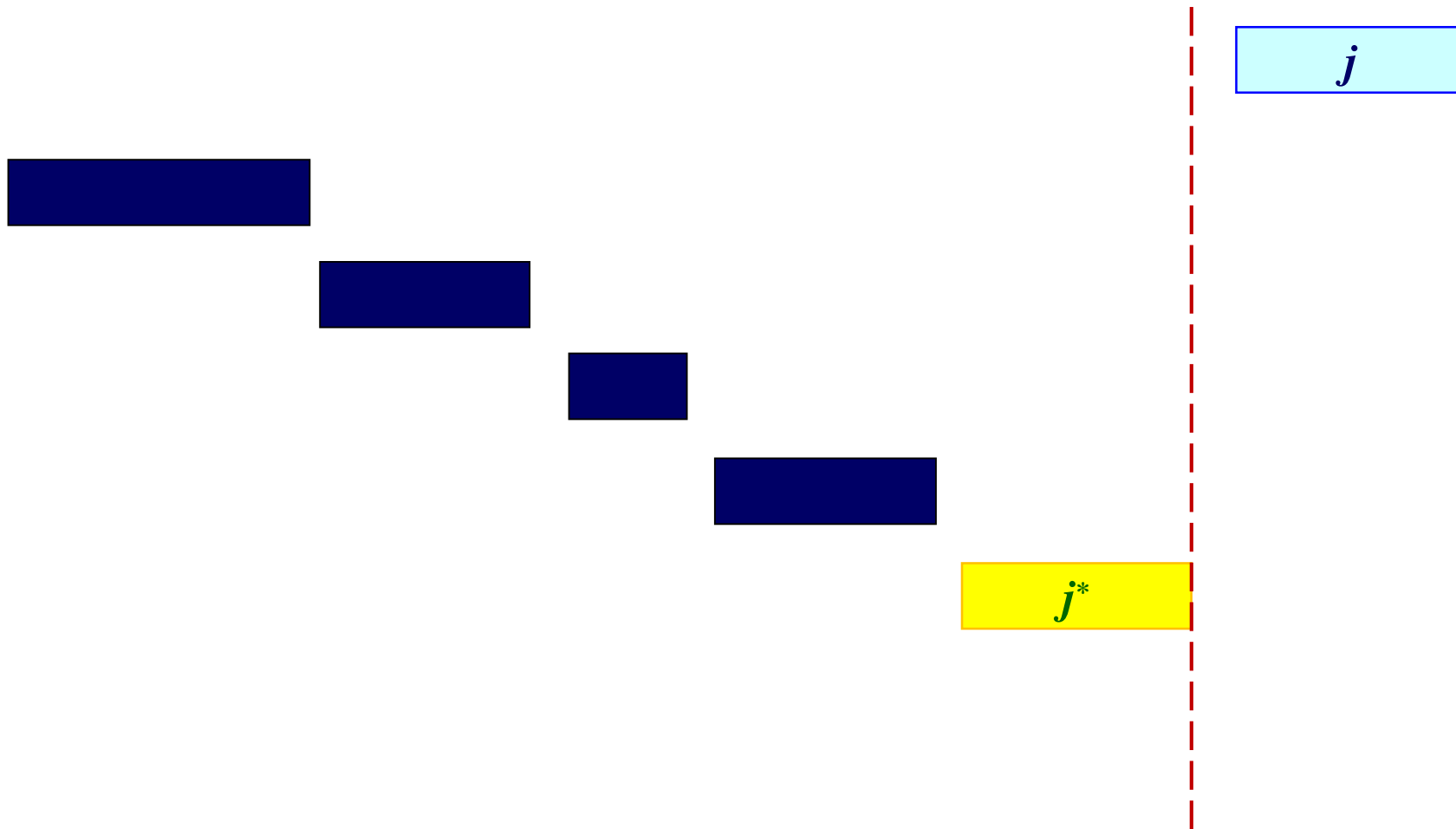
如何判断?

假设之前最后加入 A 的是活动 j^*

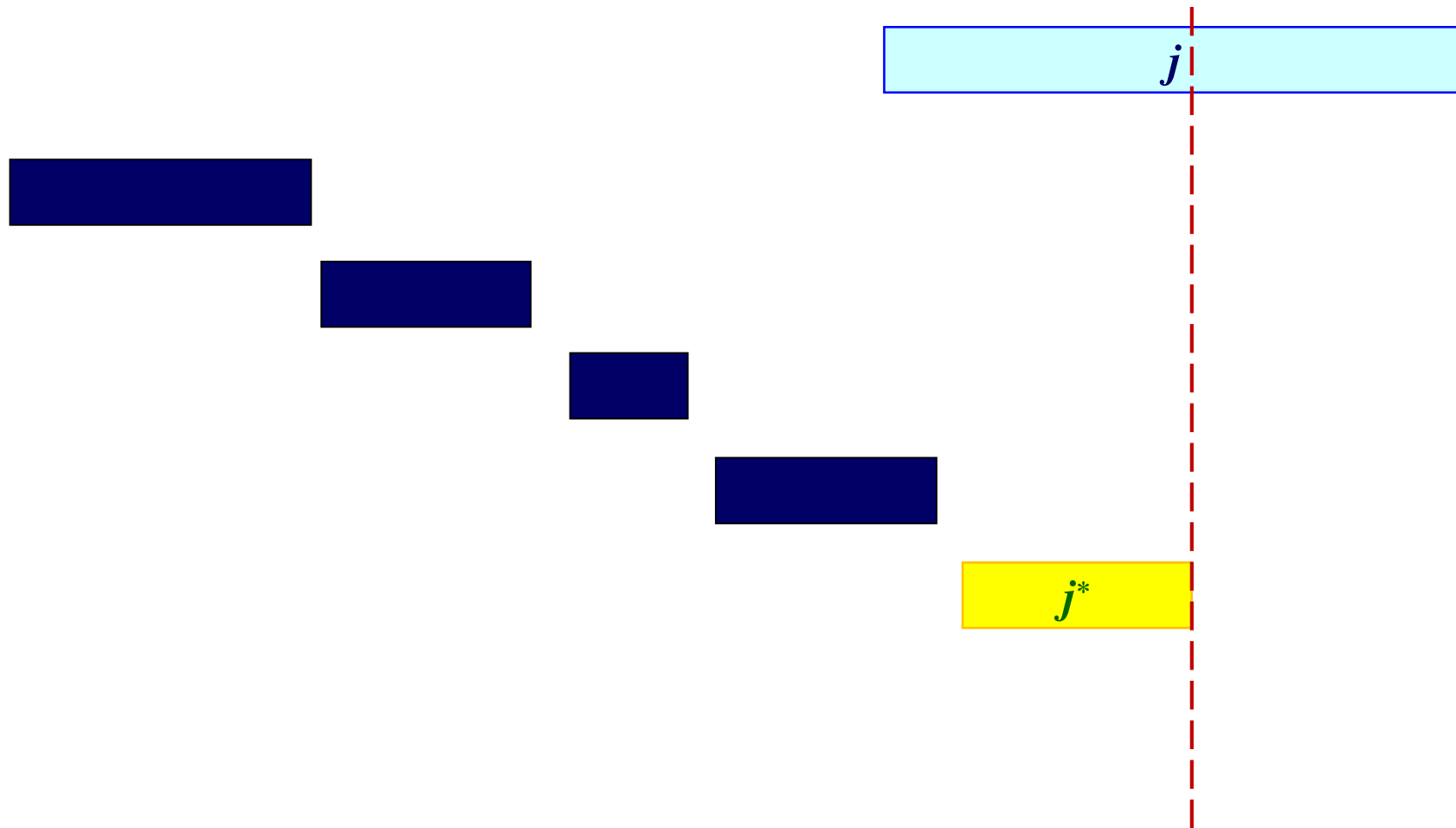
于是, 活动 j 和 A 中所有活动都相容当且仅当 $s_j \geq f_{j^*}$

不断记录和更新

活动选择问题



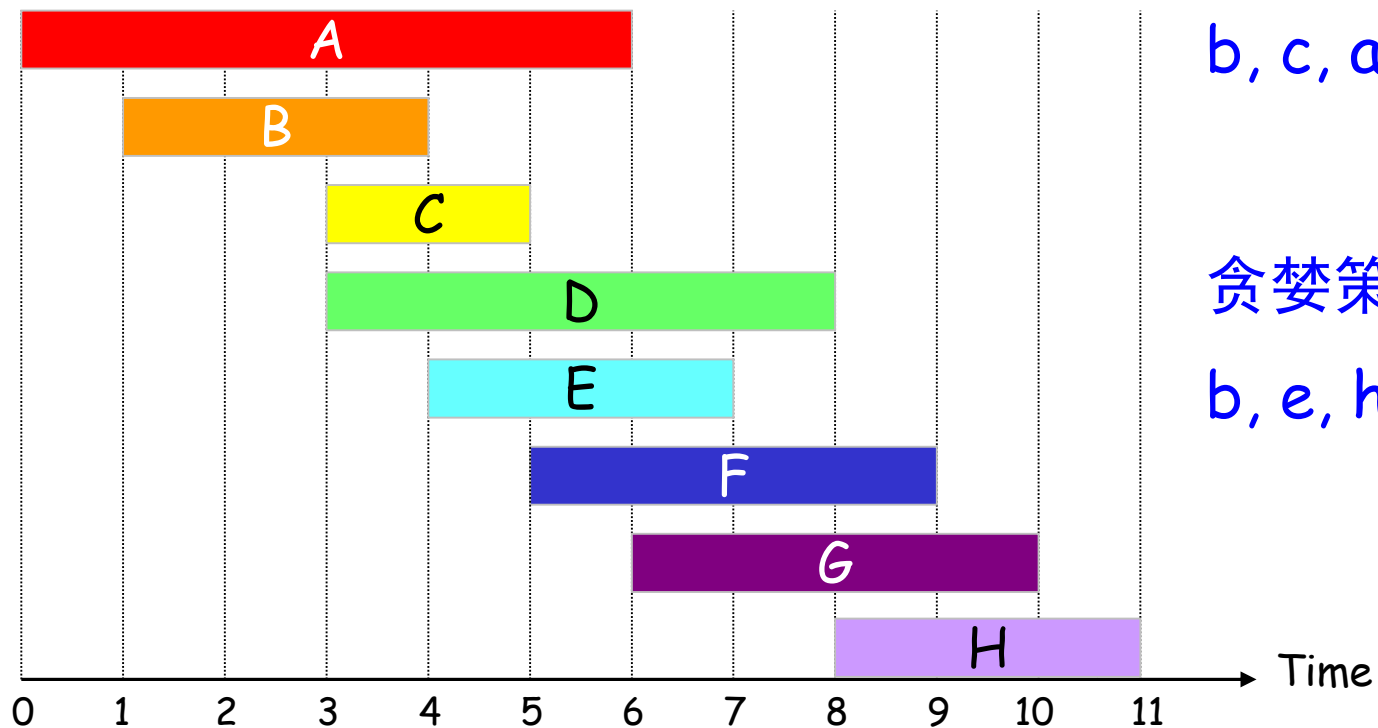
活动选择问题



活动选择问题



● 示例:



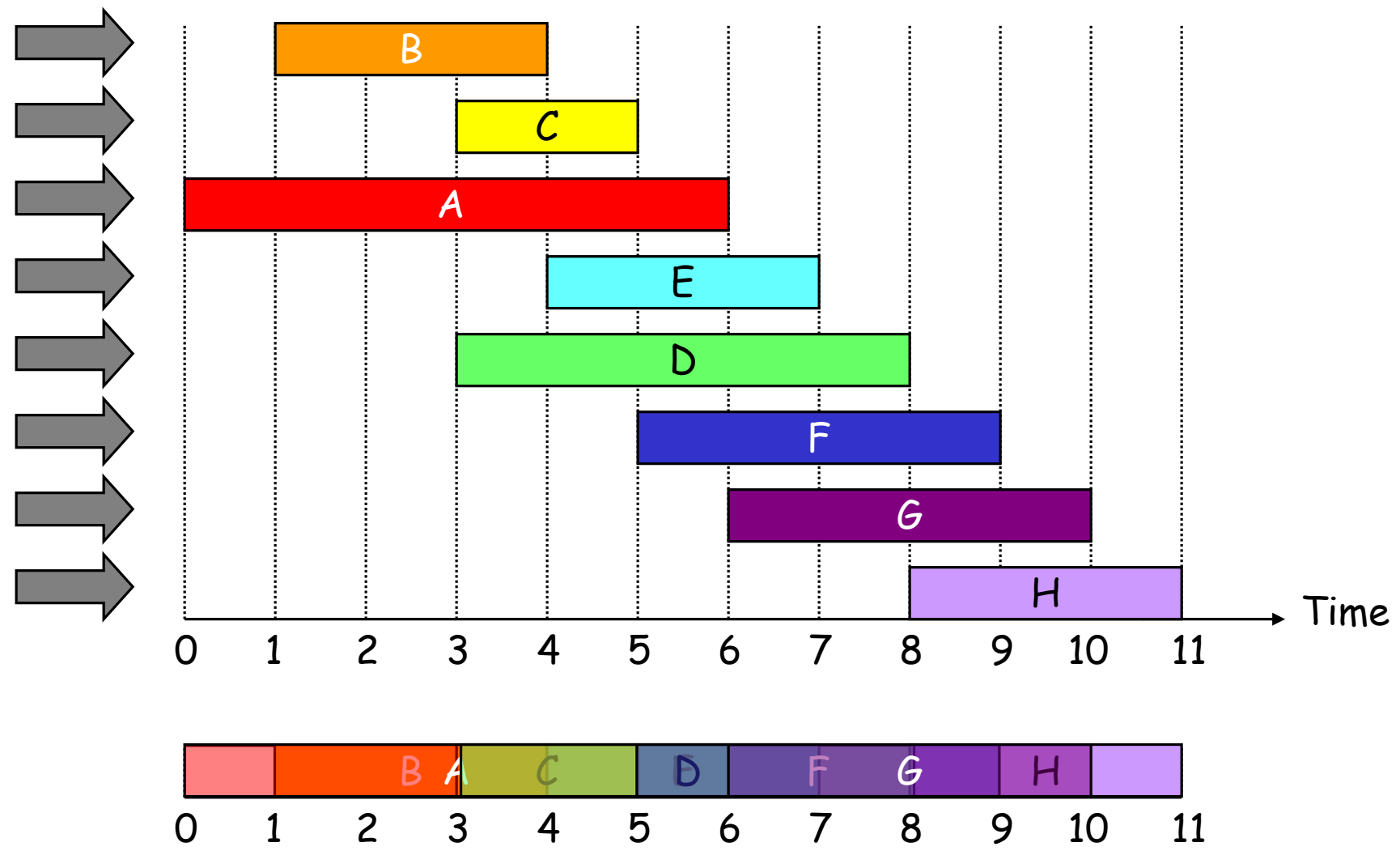
活动排序为:

b, c, a, e, d, f, g, h.

贪婪策略将选择:

b, e, h.

活动选择问题



活动选择问题



- 示例二:

- 活动 (s, f) :

- $(0, 10), (3, 4), (2, 8), (1, 5), (4, 5), (4, 8), (5, 6), (7, 9)$

- 根据 f_j 排序:

- $(3, 4), (1, 5), (4, 5), (5, 6), (4, 8), (2, 8), (7, 9), (0, 10)$

- 所选择的活动的:

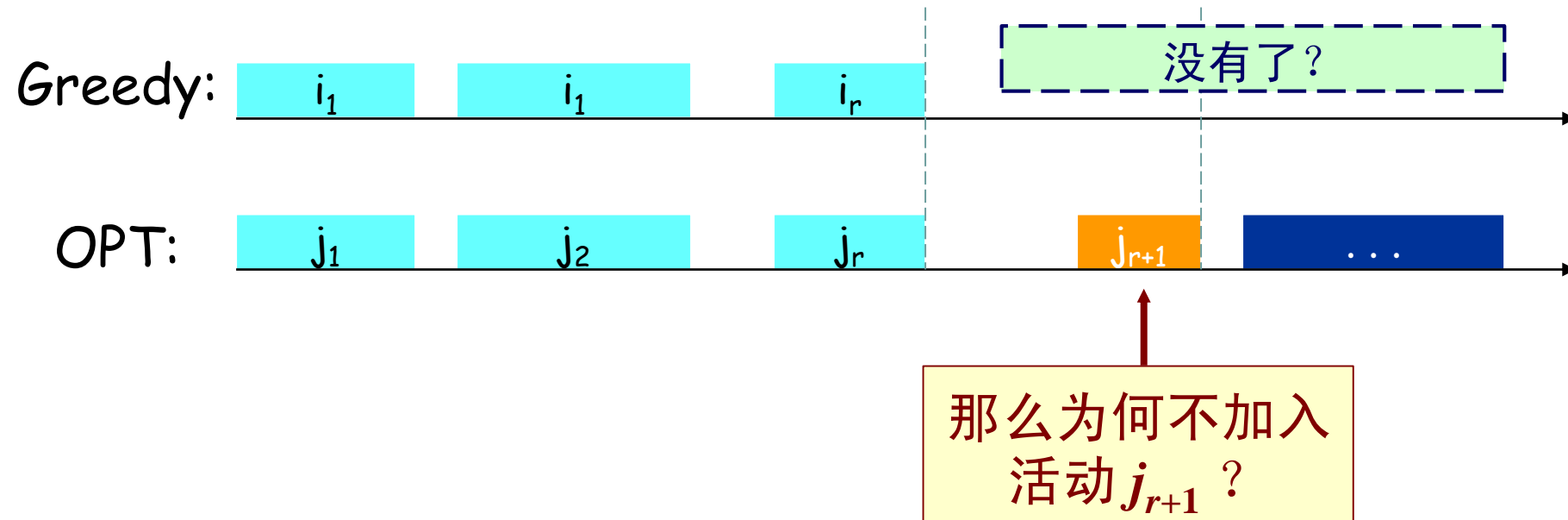
- $(3, 4), (4, 5), (5, 6), (7, 9)$

活动选择问题

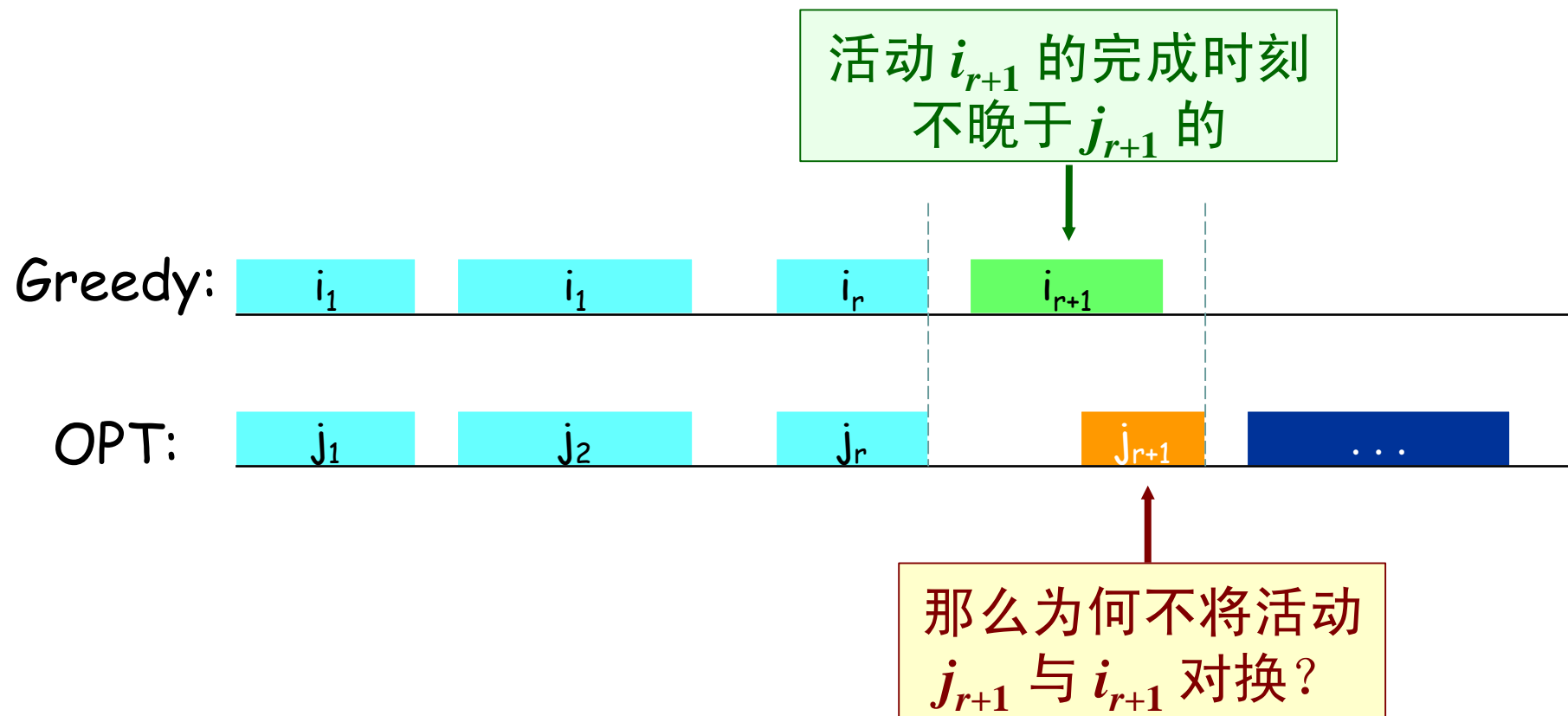


- 定理：上述贪婪策略是最优的。
- 证明：（基本思路：比较+交换）
 - 假设上述贪婪策略得到的解不是最优的
 - 假设 i_1, i_2, \dots, i_k 是上述贪婪解依次选择的活动的
 - 假设 j_1, j_2, \dots, j_m 是一个最优解，且具有满足 $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$ 的最大可能非负整数 r
 - 换言之，所有最优解中，“从左向右”看最像贪婪解的

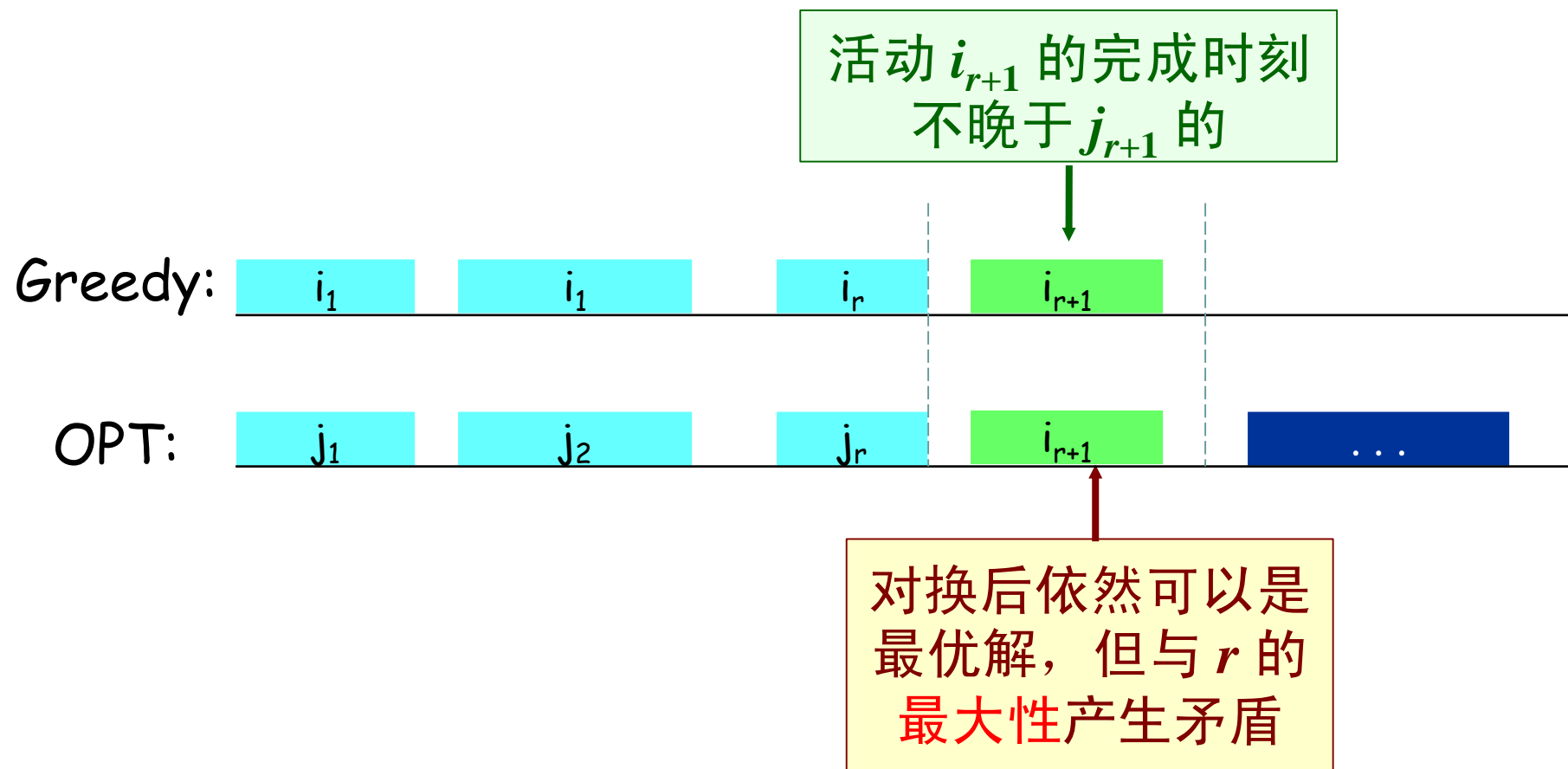
活动选择问题



活动选择问题



活动选择问题



活动安排/活动划分

Interval Partitioning

刘 铎

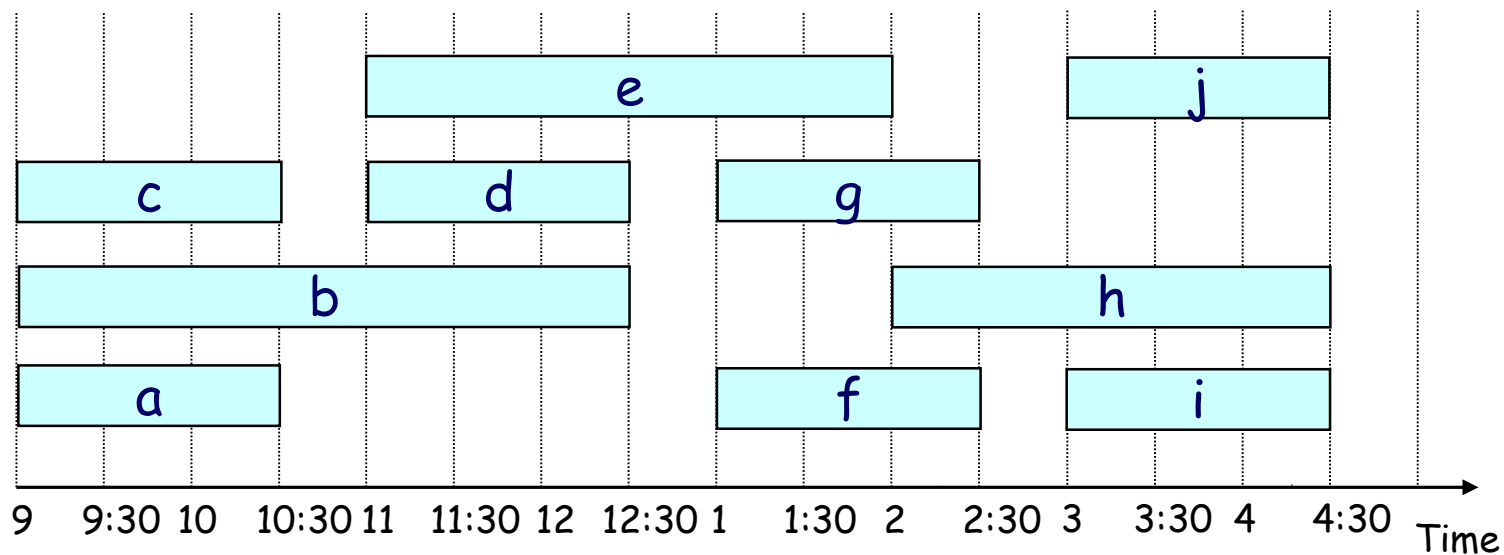
liuduo@bjtu.edu.cn



活动划分



- 给定一系列活动（都有各自的开始时刻和结束时刻）
- 例：



活动划分

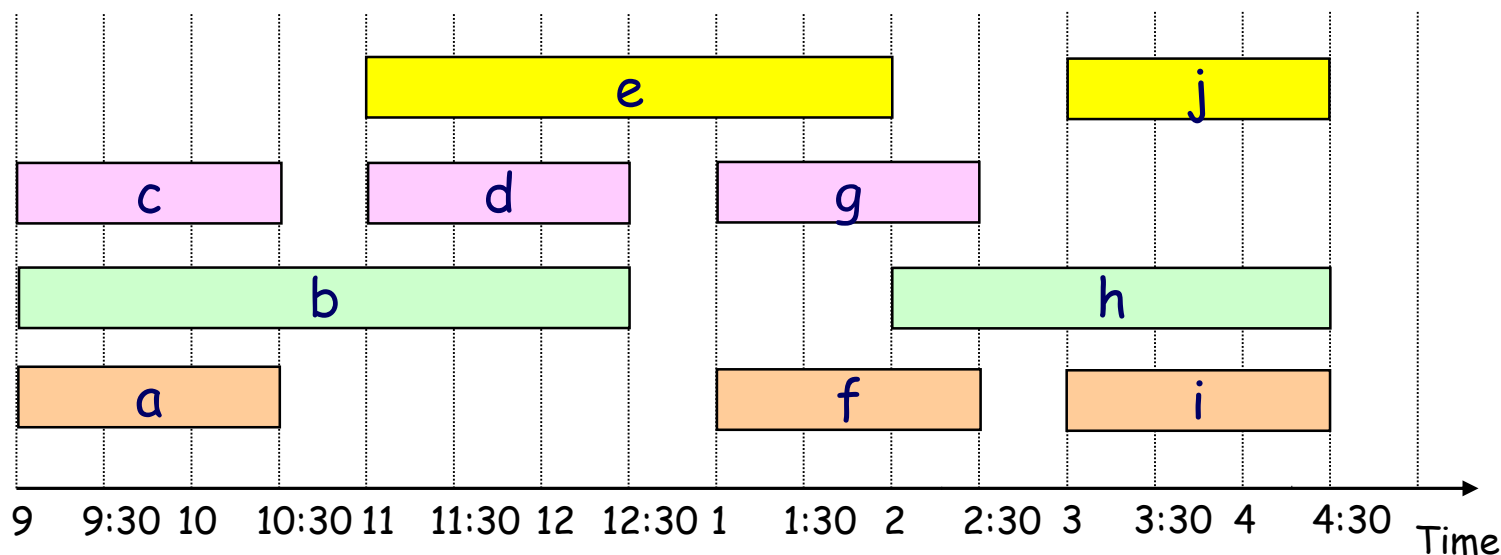


- 给定一系列活动 $\{1, 2, \dots, n\}$
- 活动 j 的开始时刻为 s_j , 结束时刻为 f_j
 - $f_j - s_j > 0$
- 这些活动必须安排在多个礼堂中, 且**不得**有冲突 (具有大于0的重叠时间)
- 设计一个算法, 计算达到此目的所需的**最少**礼堂数
 - 注意, 起止时刻是固定的, 而且在同一个礼堂中不能同时举办两个活动

活动划分



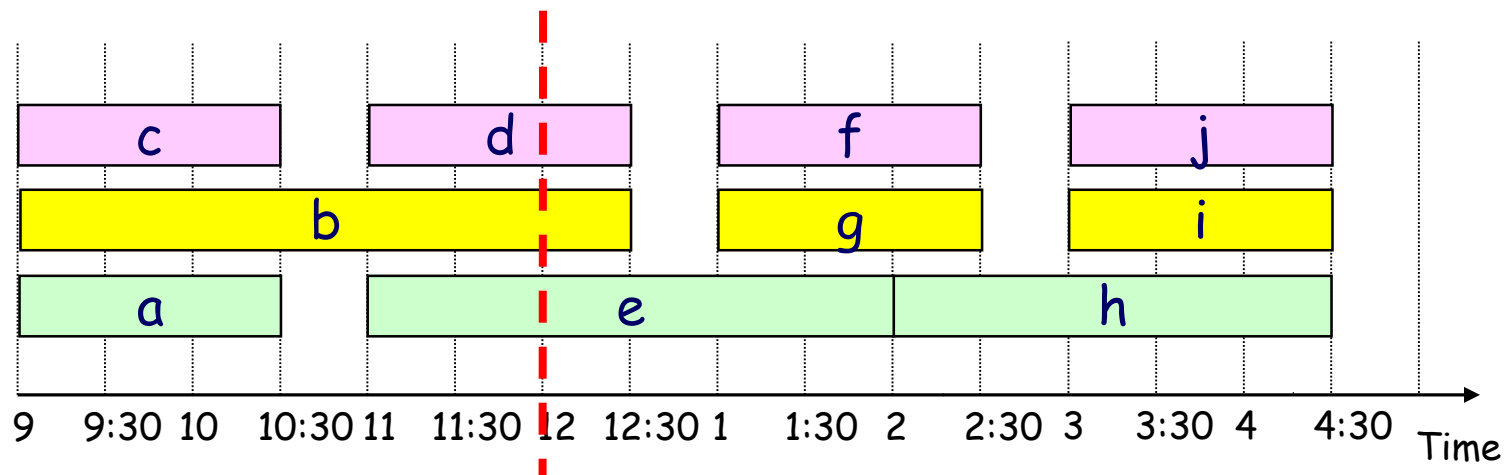
- 可以将活动视为实轴上的闭区间，我们必须为每个区间指定一种颜色，使得有所重叠的两个区间指定的颜色不同。最少需要多少种颜色？
- 例



活动划分



- 示例的最优解:



- 是否可以使用更少的礼堂?

- 不可以

活动划分



- **深度**：任意时刻同时处于进行中（不考虑当时结束或开始的活动）的最大活动数
- 关键性观察结果：**所需礼堂数量 \geq 深度**
- 问：是否总是存在一个礼堂数**等于**深度的时间表？

活动划分 —— 贪婪策略

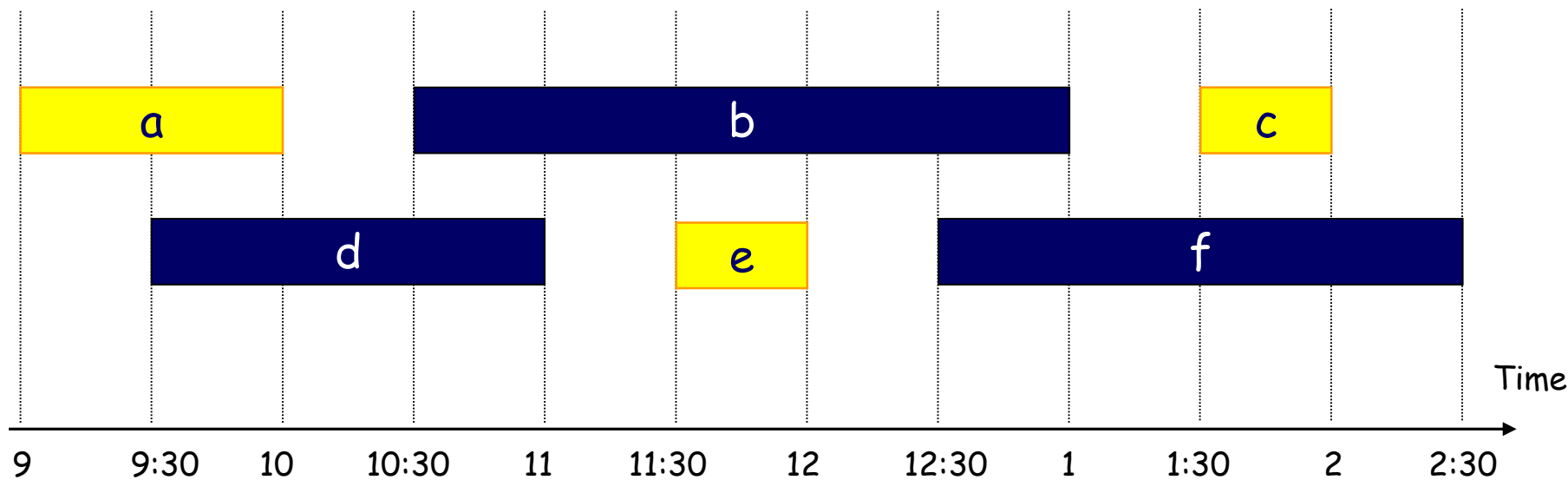


- 【最早开始时刻优先】按开始时刻 s_j 的升序考虑各个活动
- 【最早结束时刻优先】按结束时刻 f_j 的升序考虑各个活动
- 【最短持续时间优先】按持续时间的长度 $f_j - s_j$ 的升序考虑各个活动
- 【最少冲突数优先】对于每个活动 j ，计算与之冲突的活动的数量 c_j 。按冲突活动数量的升序考虑各个活动

活动划分



● 反例



- 【最早结束时刻优先】、【最短持续时间优先】、【最少冲突优先】都会选择 *a*、*e*、*c* 在同一个礼堂中，*d* 将被安排在第二个礼堂，而 *b* 被安排在第 **三** 个礼堂中.....



● 【最早开始时刻优先】

1. 对活动进行排序 使得 $s_1 \leq s_2 \leq \dots \leq s_n$
2. $d \leftarrow 0$
3. **for** $j = 1$ **to** n **do**
4. **if** (活动 j 开始时有礼堂 k 处于空闲)
5. 将活动 j 安排在礼堂 k
6. **else**
7. 安排一个新的礼堂 $d + 1$ 并将活动 j 安排在礼堂 $d + 1$
8. $d \leftarrow d + 1$

$O(n \log n)$

如何判断?

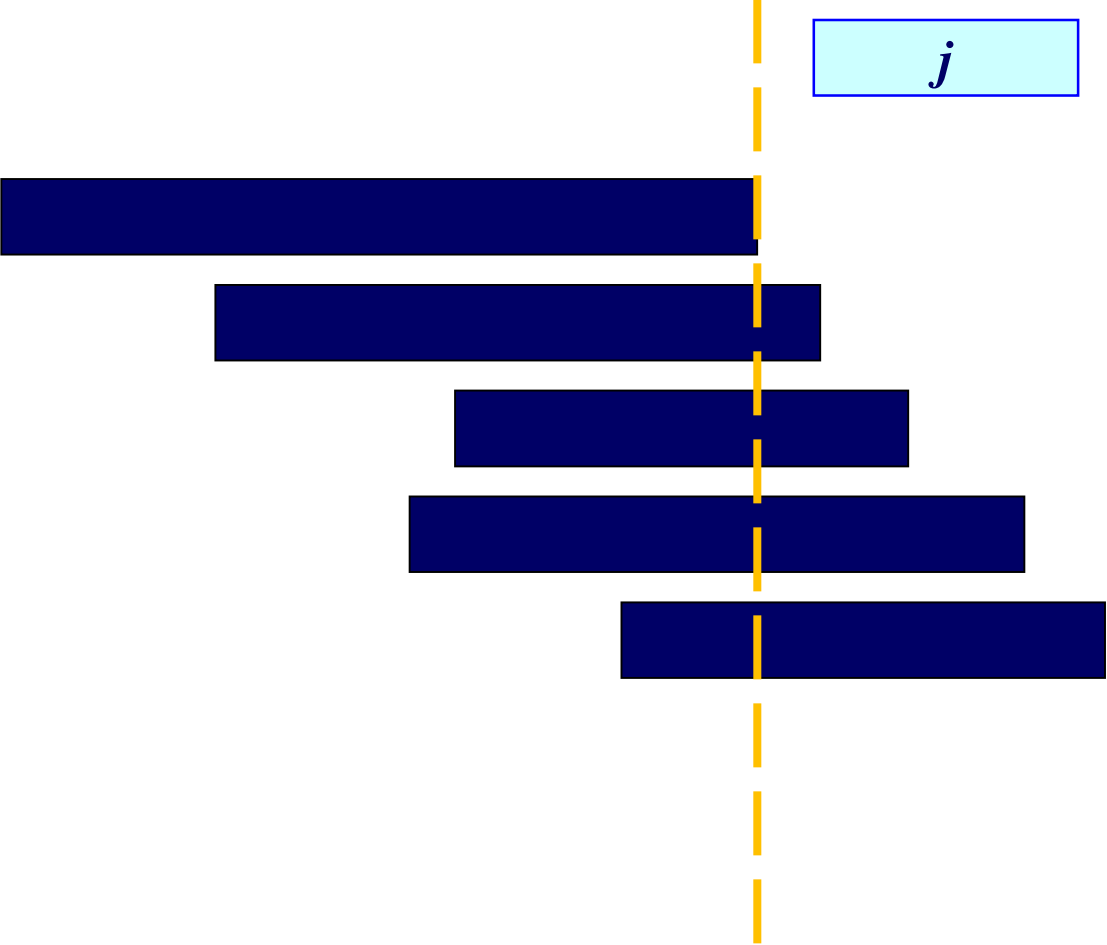
懒人策略

对于每个礼堂 k ，维护它最后一个活动的结束时刻
使用优先级队列维护各个礼堂

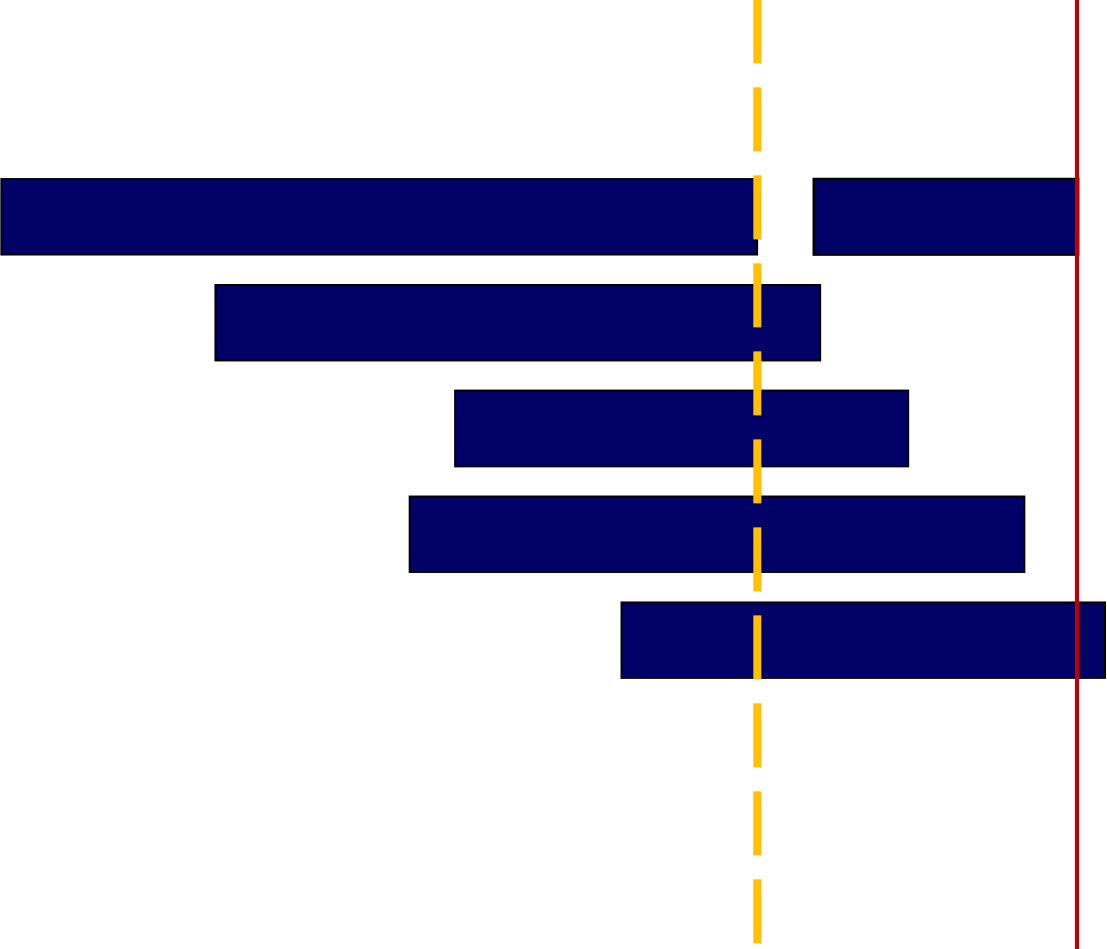
不断记录和更新

复杂度 $O(\log n)$

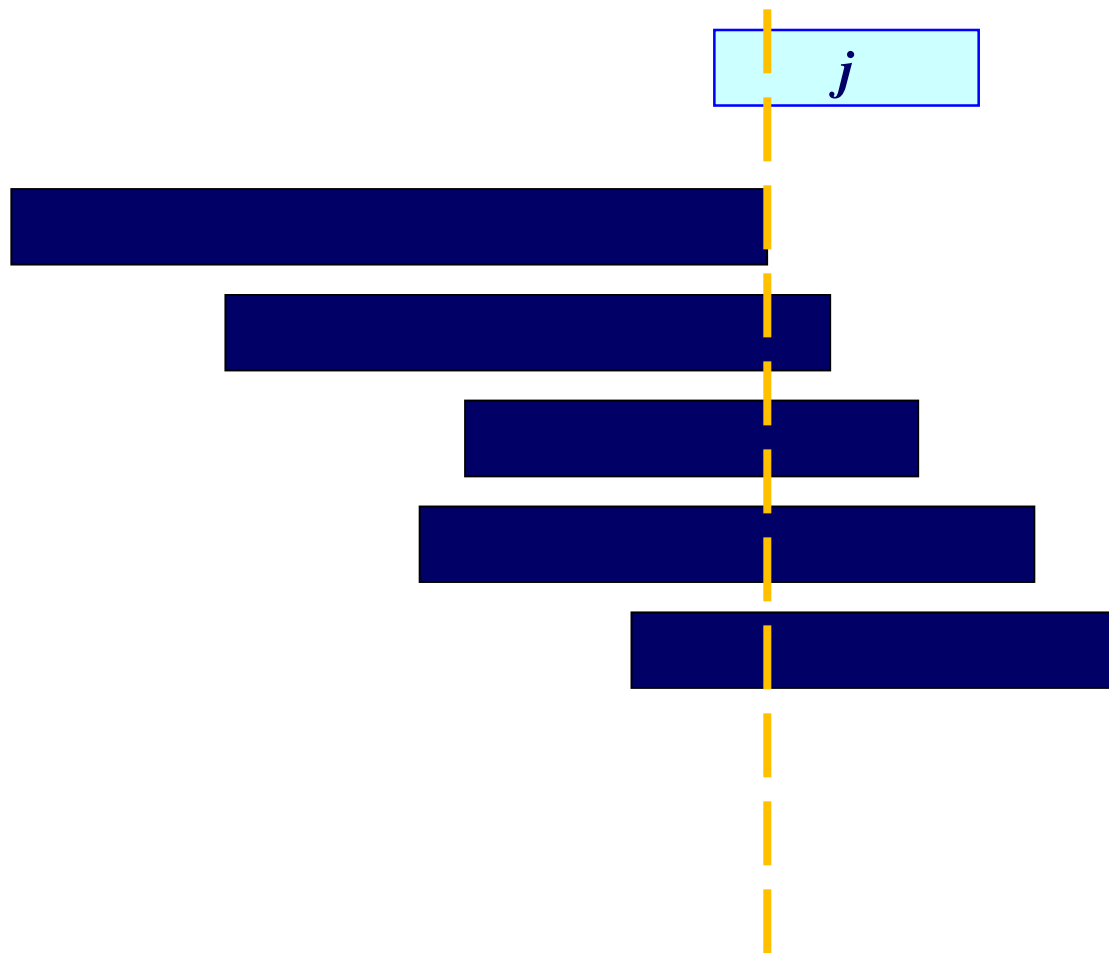
活动划分



活动划分



活动划分



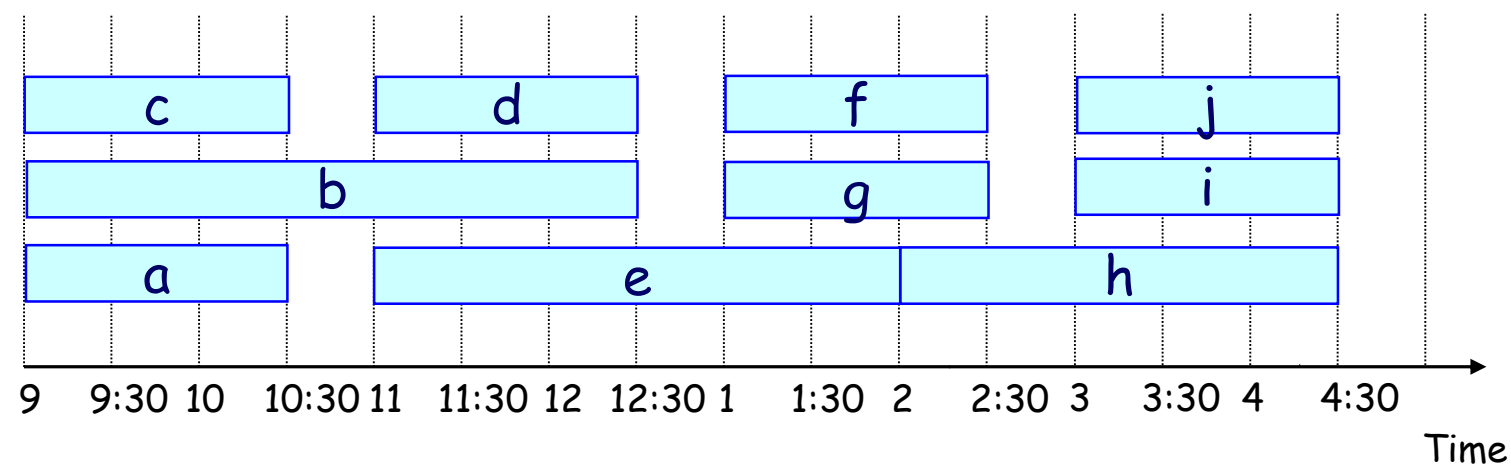
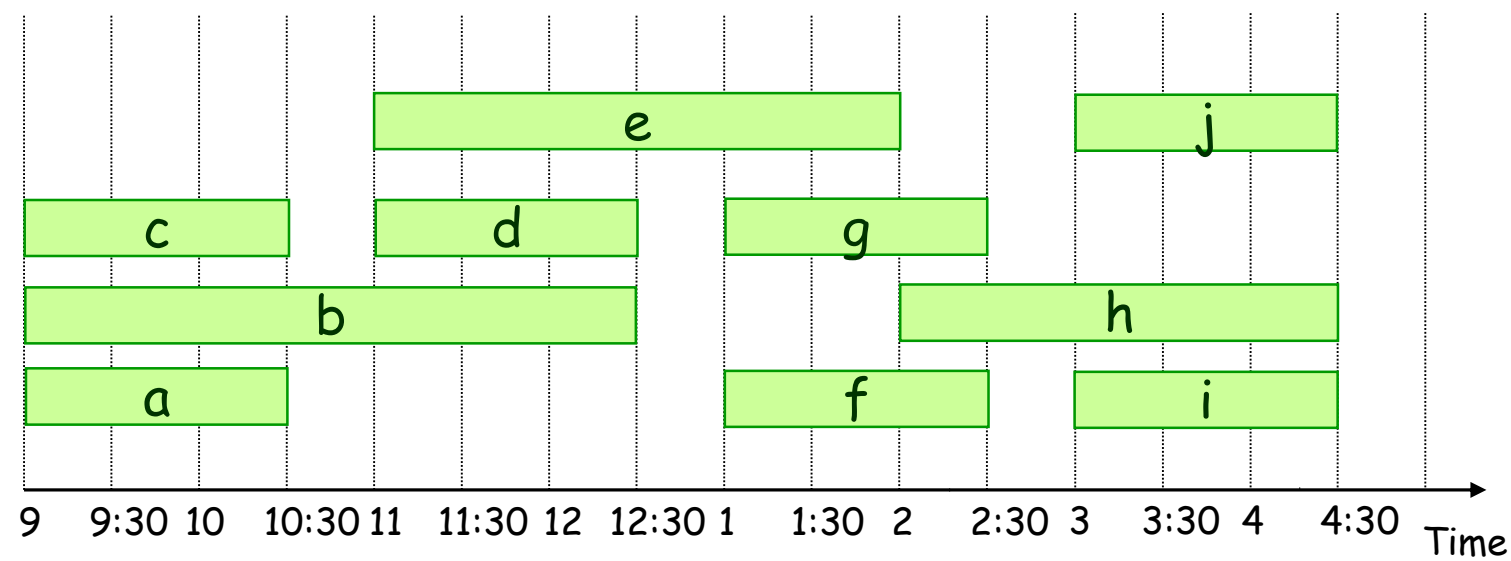
$$\begin{aligned} \text{Max}\{s_i\} &\leq s_j \\ s_j &< \text{Min}\{f_i\} \end{aligned}$$

说明活动 j 开始的时刻
其他礼堂还有活动在进行

活动划分



- 例

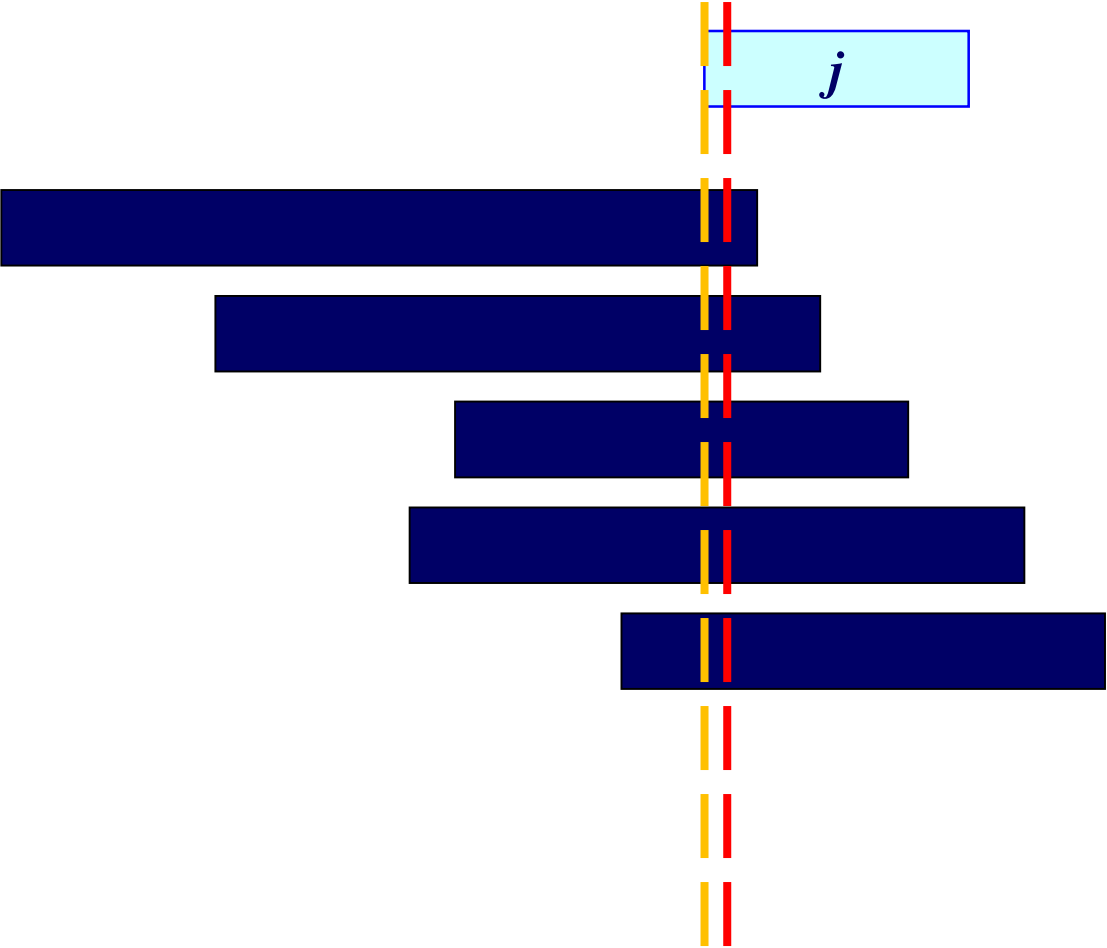


活动划分



- 定理：上述贪婪策略是最优的。
- 证明：（基本思路：贪婪算法总是达到“深度值”）
 - 令 d 表示贪婪解安排的礼堂数目
 - 礼堂 d 被“开放使用”是因为需要安排的下一个活动（记之为 j ）与其他 $d-1$ 个礼堂中的活动皆有时间冲突
 - 由于各个活动是按照开始时间排序的，因此其他 $d-1$ 个礼堂中的与活动 j 皆有时间冲突的活动的开始时间都不晚于 s_j
 - 因此在时刻 $s_j + \varepsilon$ 时，同时进行的活动至少有 d 个
 - 其中 ε 是一个非常小的正数值
 - 说明深度 $\geq d$
 - 因此任何一个安排方案都将至少使用 d 个礼堂

活动划分



最小延迟调度

Scheduling to Minimize Lateness



最小延迟调度



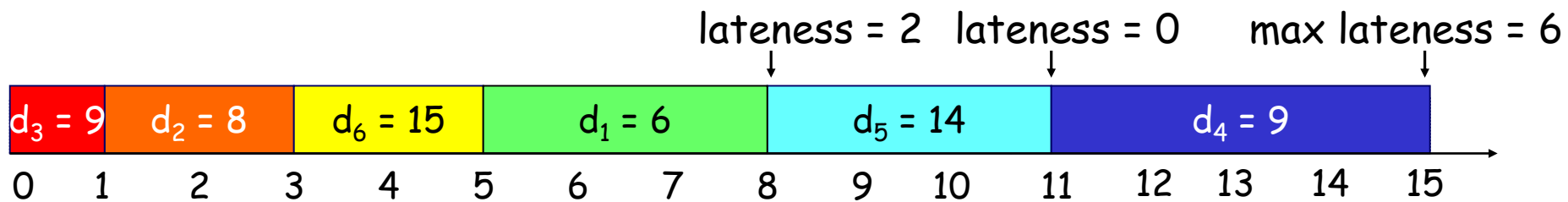
- 有多个任务，而只有一台任务处理机，各个任务**必须串行**
- 每个任务 i 需要使用 t_i 个单位时间进行处理，而且有完成时限 d_i
 - 假定 t_i 和 d_i 都是正整数
- 如果任务 i 在时刻 s_i 开始，则它的完成时刻为 $f_i = s_i + t_i$ ，它的**延迟**（lateness）定义为 $l_i = \max \{ 0, f_i - d_i \}$
- 目标就是找到一种安排各个任务被处理的次序，使得他们的**最大延迟**（ $\max l_i$ ）达到**最小**
 - 事实上从后文可知，一旦各个任务被处理的次序确定，那么各个任务的开始时刻、完成时刻和延迟就都很容易计算

最小延迟调度



- 例:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



最小延迟调度 —— 贪婪策略



- 【最短任务优先】
 - 按任务所需时间 t_i 的升序考虑各个任务
- 【紧急任务优先】
 - 按任务的时限 d_i 的升序考虑各个任务
- 【最小松弛时间优先】
 - 按 $d_i - t_i$ 的升序考虑各个活动

最小延迟调度



- 反例

$l_2=0, l_1=0$

0 10 11

	1	2
t_j	1	10
d_j	100	10

【最短任务优先】

$l_1=0, l_2=1$

0 1 11

$l_1=0, l_2=1$

0 1 11

	1	2
t_j	1	10
d_j	2	10

【最小松弛时间优先】

$l_1=9, l_2=0$

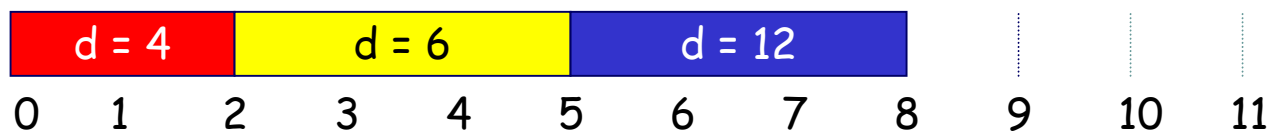
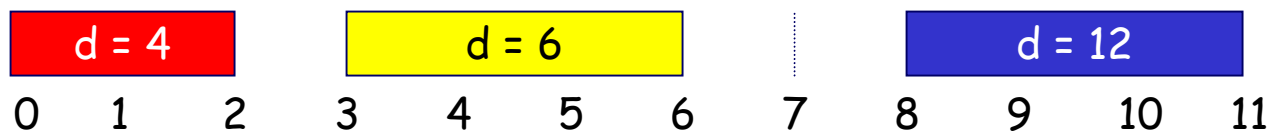
0 9 11

最小延迟调度



- 【紧急任务优先】

- 按任务的时限 d_i 的升序考虑各个任务
- 观察结果：最优调度方案可以**无空闲时间**



最小延迟调度



- 【紧急任务优先】

- 按任务的时限 d_i 的不降次序考虑各个任务
- 观察结果：最优调度方案可以无空闲时间

对任务进行排序 使得 $d_1 \leq d_2 \leq \dots \leq d_n$

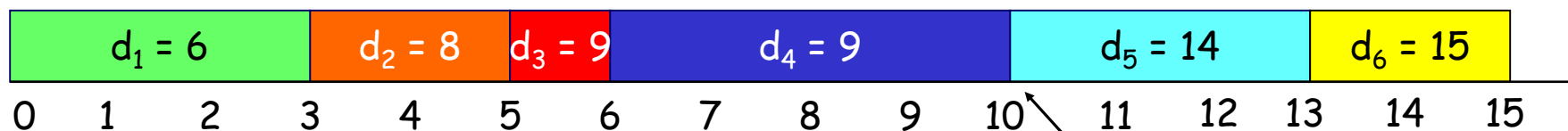
$O(n \log n)$

$f_0 \leftarrow 0$

for $i = 1$ **to** n

$f_i \leftarrow f_{i-1} + t_i$

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



max lateness = 1

最小延迟调度



- 【紧急任务优先】

- 按任务的时限 d_i 的不降次序考虑各个任务
- 观察结果：最优调度方案可以无空闲时间

对任务进行排序 使得 $d_1 \leq d_2 \leq \dots \leq d_n$

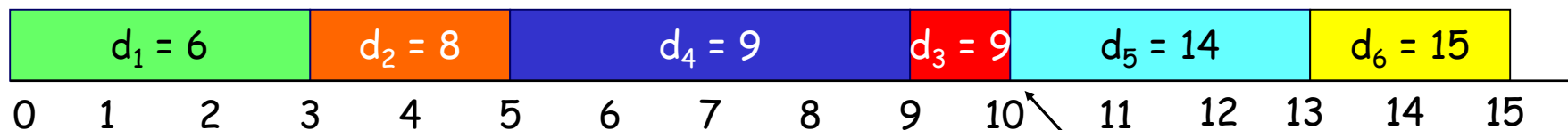
$O(n \log n)$

$f_0 \leftarrow 0$

for $i = 1$ **to** n

$f_i \leftarrow f_{i-1} + t_i$

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



max lateness = 1

最小延迟调度



- 定理：上述贪婪策略是最优的

最小延迟调度

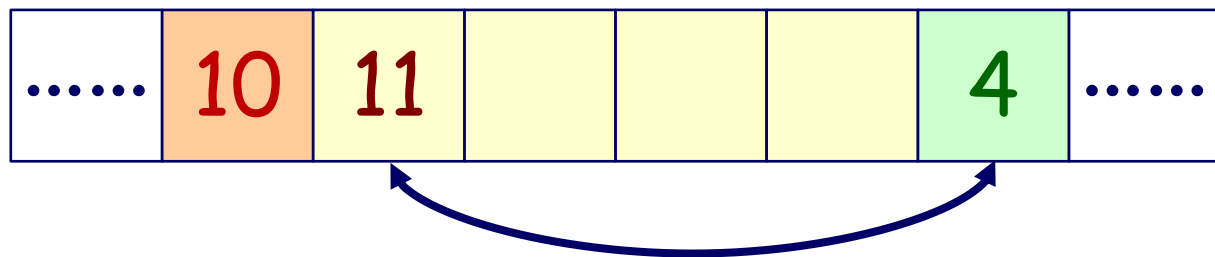
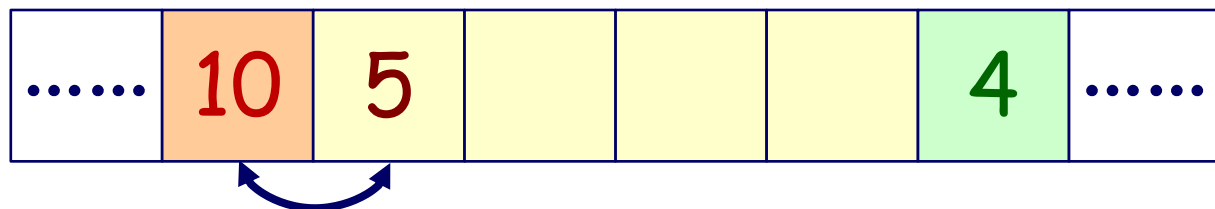
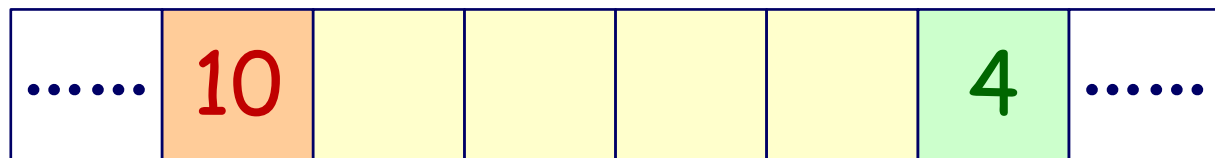


- 定义：任务 i 和任务 j 形成**逆序**： $f(i) > f(j)$ 且 $d_i < d_j$
 - 即 $d_i < d_j$ 但是任务 j 在任务 i 之前被处理
- 观察结果1： 贪婪解中不存在逆序
- 观察结果2： 如果一个（无空闲时间的）调度方案中有一对逆序，那么就一定存在两个**相邻**的任务形成逆序

最小延迟调度



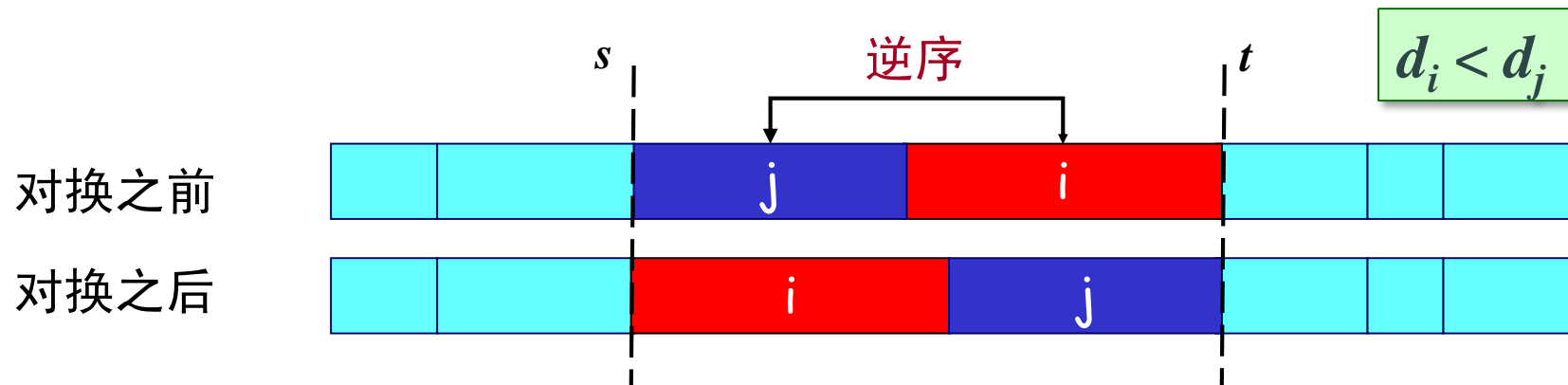
- 观察结果2：如果一个（无空闲时间的）调度方案中有一对逆序，那么就一定存在两个**相邻**的任务形成逆序



最小延迟调度



- 对换相邻逆序任务 i 和 j ，会如何？



交换 i, j 对其他任务的延迟时间没有影响

交换 i, j 后

任务 i 的延迟时间不会增加

任务 j 的延迟时间有可能增加（变得超时了），但不超过 $t - d_j$

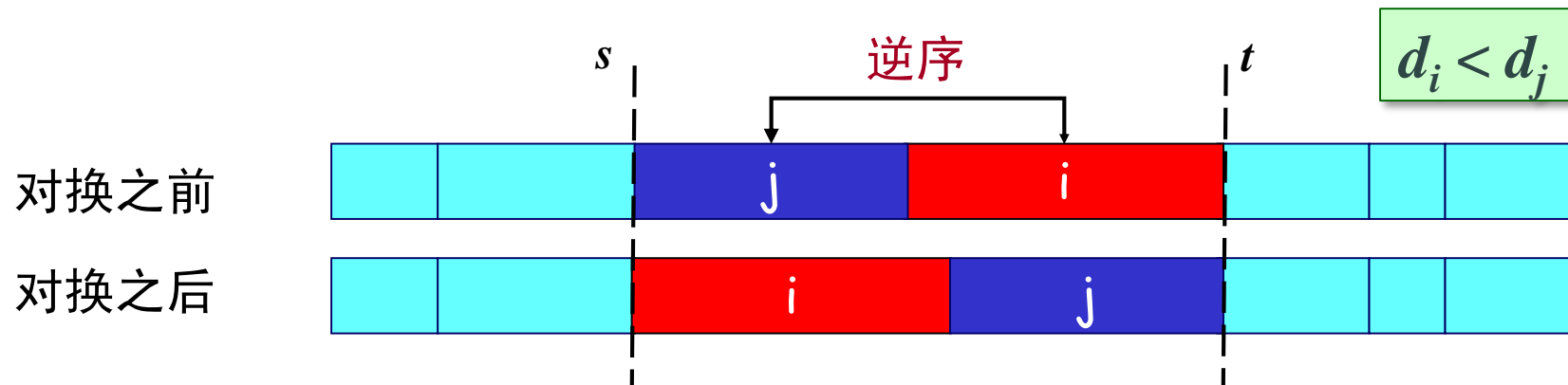
最大延迟时间值不会增加

$$\begin{aligned} &< t - d_i \\ &\leq l(i) \end{aligned}$$

最小延迟调度



- 对换相邻逆序任务 i 和 j ，会如何？



交换 i, j 对总逆序数的影响

交换 i, j 后

浅蓝色任务与任务 i 形成的逆序对 **不** 变化

浅蓝色任务与任务 j 形成的逆序对 **不** 变化

任务 i 与任务 j 不再形成逆序对

总逆序数严格减少

最小延迟调度



- 定理：上述贪婪策略是最优的。
- 证明：（基本思路：比较+交换）
 - 假设 S^* 是具有最少逆序数的最优解
 - 假设 S^* 中不存在逆序，则其就是贪婪解
 - 如果 S^* 中存在逆序，那么它就一定存在相邻的逆序，记之为 $i-j$
 - 对换任务 i 和任务 j ，并不会增加最大延迟（又由“最优解”可知事实上最大延迟保持不变），但会严格减少总逆序数
 - 与 S^* 的定义产生矛盾

最小延迟调度



- 一般而言，这类“规划次序”“调度”问题
 - 可以考虑比较“相邻”的二者
 - 交换次序后是否/何时会提高总的“评价值”
- 当然要“具体问题具体分析”

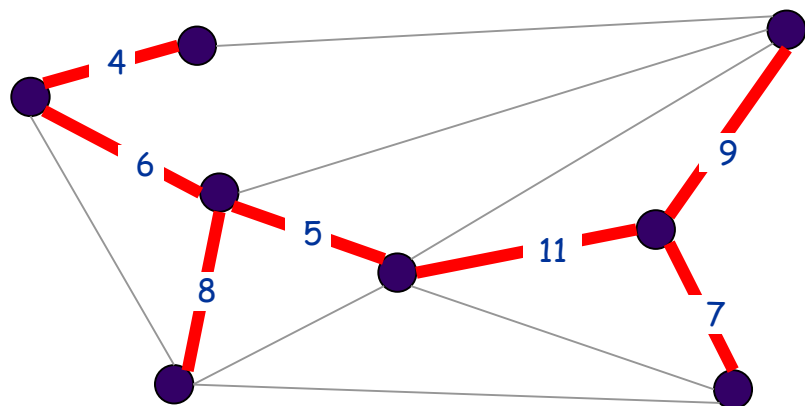
贪婪算法的分析策略



- 贪婪算法保持领先
 - 论证在贪心算法的每一步后，其解至少和其他算法一样好
 - 例如找零问题（1，5，10）
- 参数交换
 - 在不影响解的质量的前提下，将任意解逐步转化为贪婪算法得到的解
 - 例如（计件的）装载问题、活动选择问题、最小延迟调度
- 结构性特征
 - 发现一个简单的“结构性”界，之后断言每个可能的解决方案都必须有某个确定的值。然后证明贪婪算法总是达到这个界
 - 例如活动划分问题

最小支撑树

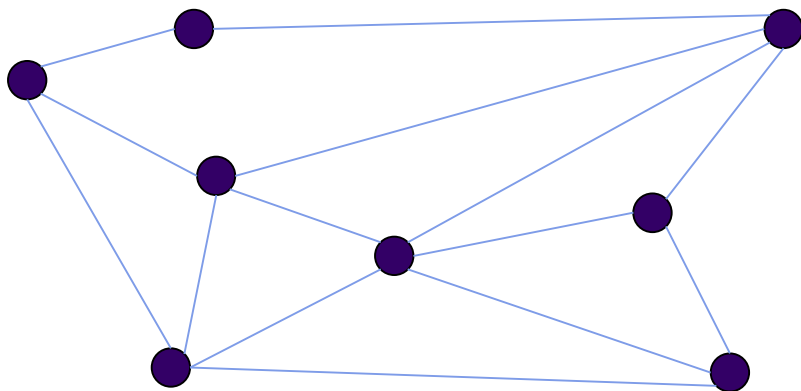
Minimum Spanning Tree, MST



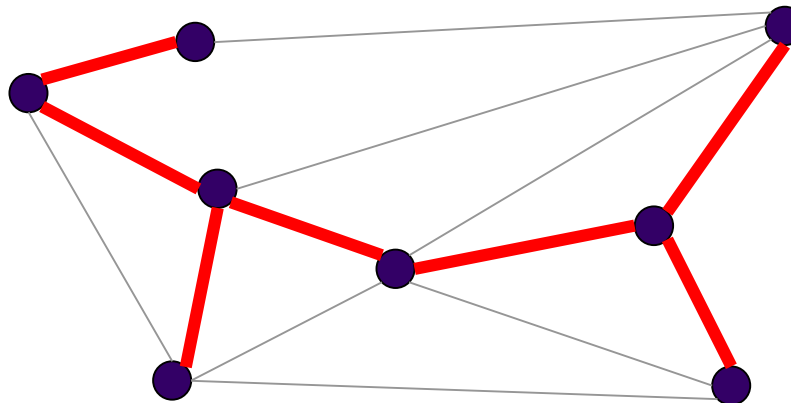
支撑树 (*Spanning Tree*)



- 支撑树/生成树 设 $T = (V, F)$ 是 $G = (V, E)$ 的子图。则下述陈述彼此等价：
 - T 是 G 的一棵支撑树
 - T 是连通且无圈的
 - T 是连通的且具有 $|V|-1$ 条边
 - T 是无圈的且具有 $|V|-1$ 条边
 - T 是极小连通的：移除 T 任何一条边都会使得其不再连通
 - T 是极大无圈的：向 T 中添加任何一条新的边都会产生一个简单回路（圈）
 - T 中任一对相异顶点之间都存在唯一的简单道路



$G = (V, E)$



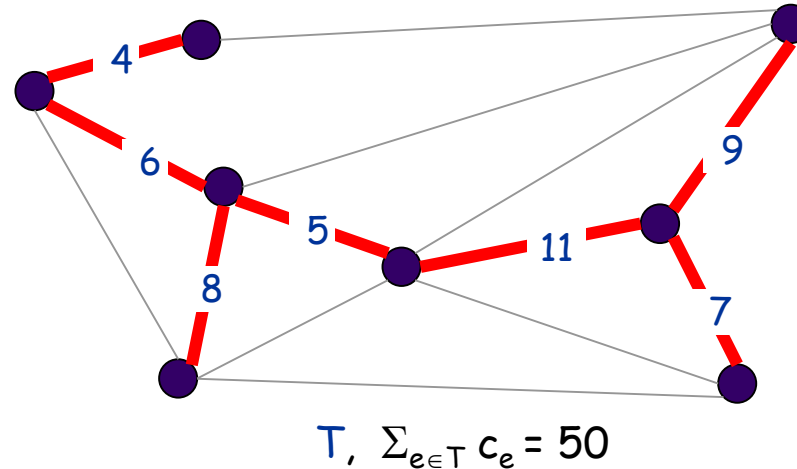
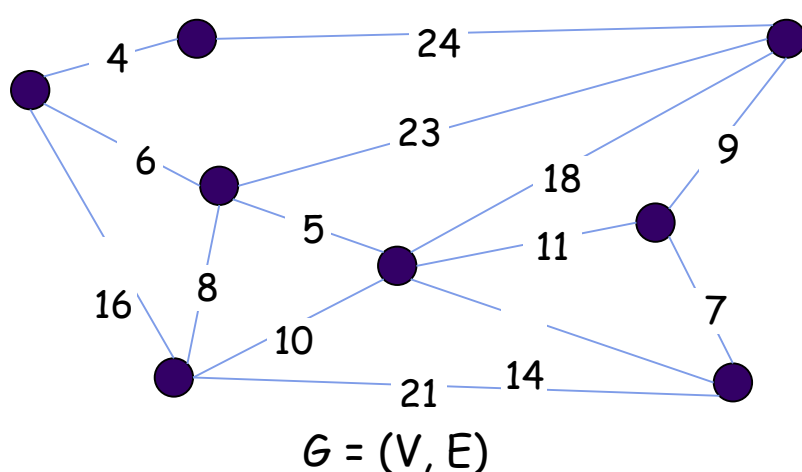
T

最小支撑树 (MST)



- 最小支撑树

- 给定一个无向连通赋权图，该图所有支撑树中各边权值之和（记做cost）最小者称为这个图的一棵**最小支撑树**（**minimal spanning tree, MST**）或最小生成树



最小支撑树 (MST)



- MST 是一个很基本的组合问题
- 具有很多应用
- 例如：网络设计、人脸验证、以太网的网桥、TSP的近似算法等

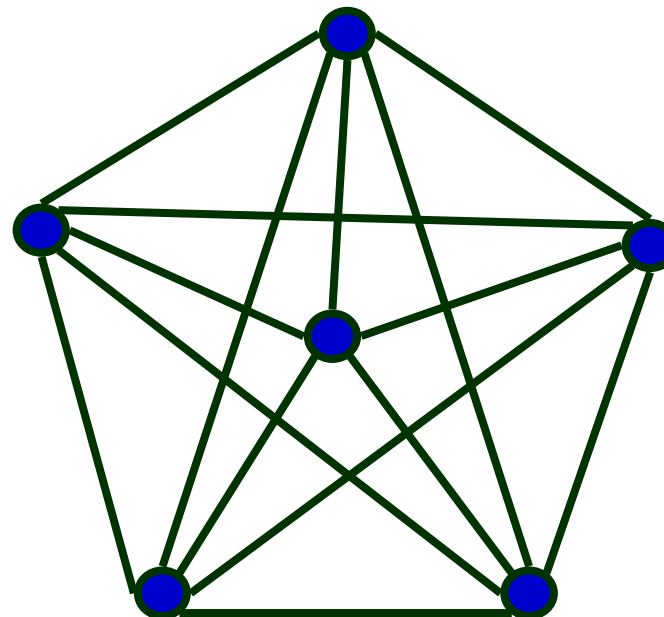
最小支撑树 (MST)



- Cayley定理 (1889年) :

- n ($n > 1$) 阶顶点标号完全图 K_n 有 n^{n-2} 棵不同的标号支撑树

- 无法使用蛮力法解决



MST算法 —— 选择边



- Kruskal算法

- 将所有边按照权值递增（不减）次序排序。依序考察每条边：如果某条边没有和已经选出来的边构成回路，就将其加入备选集合，之后继续考察下一条边。直至那考察完所有的边，或者选择了足够数目的边为止

- Prim算法

- 从某一顶点开始，保持为树的前提下，通过重复添加顶点和边将其生长为 n 个顶点的树。每次当有多个选择时，都选择权值最小的边

MST算法 —— 丢弃边



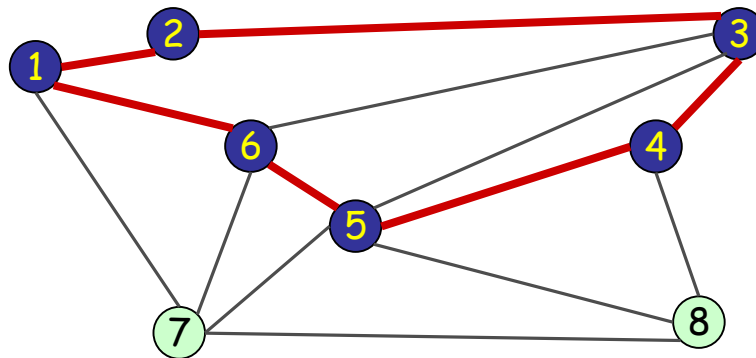
- Reverse delete算法

- 从连通图开始。在保持连通性的前提下，不断删除权值最高的边，直至找不到这样的边为止
- 考察任一回路，删除其中权值最高的边，直至图中不存在回路为止

圈与割

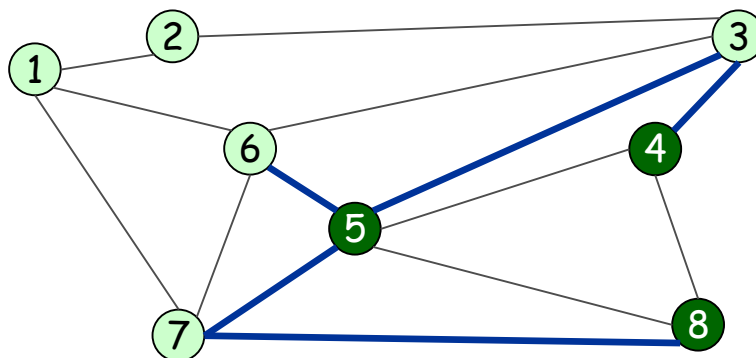


- **圈 (cycle)** 形如 $a-b, b-c, c-d, \dots, y-z, z-a$ 的边序列



Cycle $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$

- **割 (cut)** 顶点集的一个非空真子集 S 。相对应的**割集** (cut set) D 指的是端点**包含且仅包含** S 中一个点的边集合

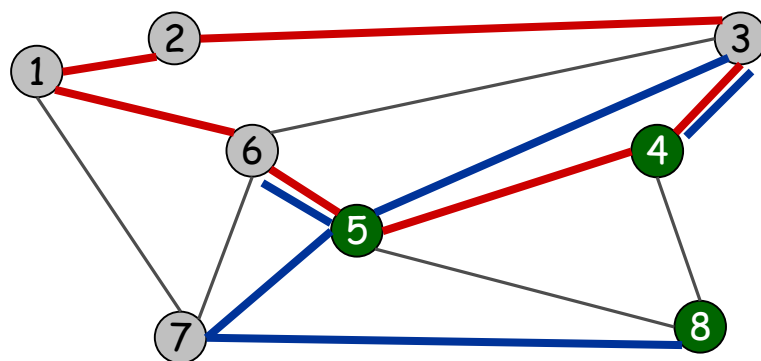


Cut $S = \{4, 5, 8\}$
Cutset $D = 5-6, 5-7, 3-4, 3-5, 7-8$

圈与割的交集

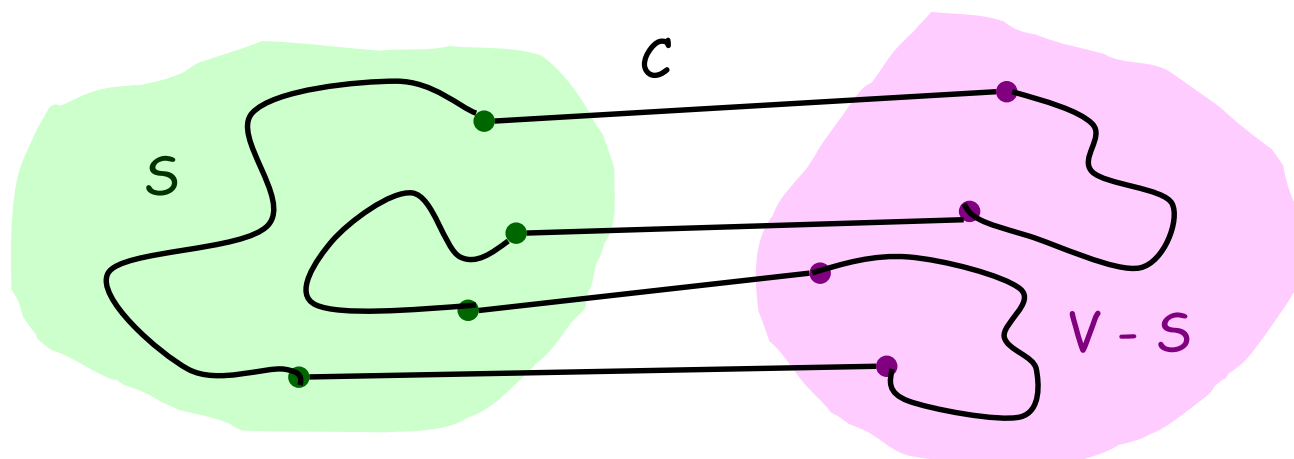


- 断言：任一圈（简单回路）与任一割集的交集有偶数条边



Cycle $C = \{1-2, 2-3, 3-4, 4-5, 5-6, 6-1\}$
Cutset $D = \{3-4, 3-5, 5-6, 5-7, 7-8\}$
二者交集 = $\{3-4, 5-6\}$

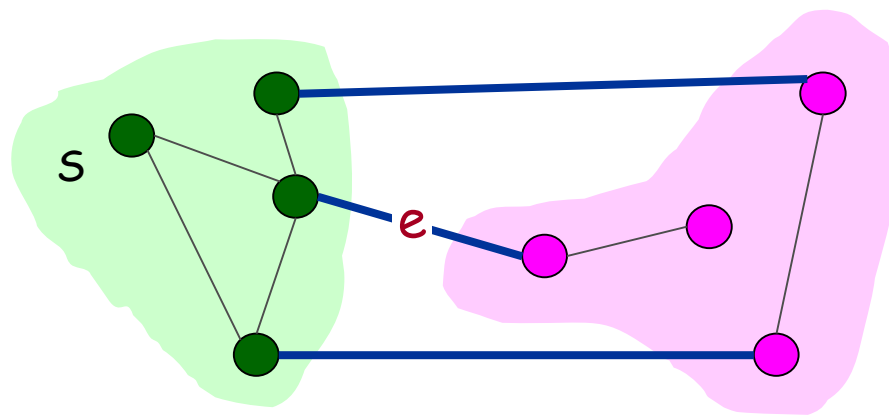
- 证明（大意）：如图所示



割性质



- 假定 各边的权值 c_e 彼此不同
- 割性质: 设 S 是顶点集合 V 的一个真子集, e 是 S 对应的割集 D 中最轻 (权重最小) 的边, 则 G 的 **MST** T^* 必然包含 e



e is in the MST

割性质



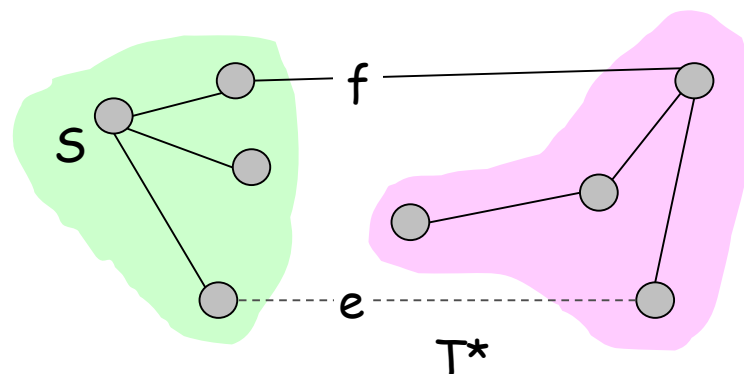
- 证明：（“交换”论证）

- 假设边 e 不在 T^* 中
- 将 e 加入添加到 T^* 中将形成图 G 中的回路 C
- 边 e 同时在回路 C 和割集 D 中，因此必定还存在图 G 中的另一条边 f 也同时在回路 C 和割集 D 中
- 边 f 的权值严格大于 e 的权值
- $T' = T^* \cup \{e\} - \{f\}$ 也是 G 的一棵支撑树
- 由 $c_e < c_f$ 可知 $\text{cost}(T') < \text{cost}(T^*)$
- 于是产生矛盾

极大无圈

前述断言

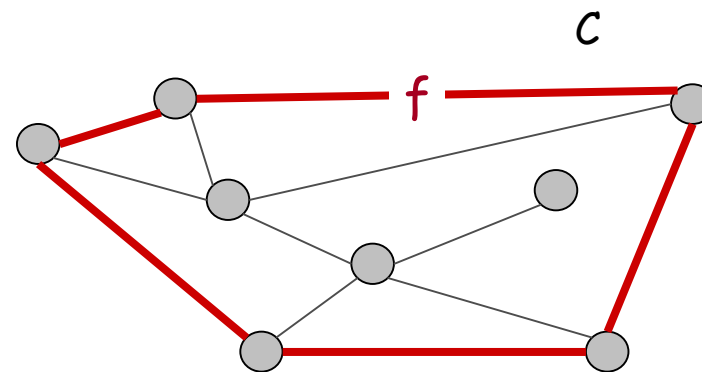
连通且边数不变



圈性质



- 假定各边的权值 c_e 彼此不同
- **圈性质**: 设 C 是图 G 中的一个圈（即简单回路）， f 是 C 中最重（权重最大）的边，则 G 的 **MST T^*** 必然不包含 f



f is not in the MST

圈性质



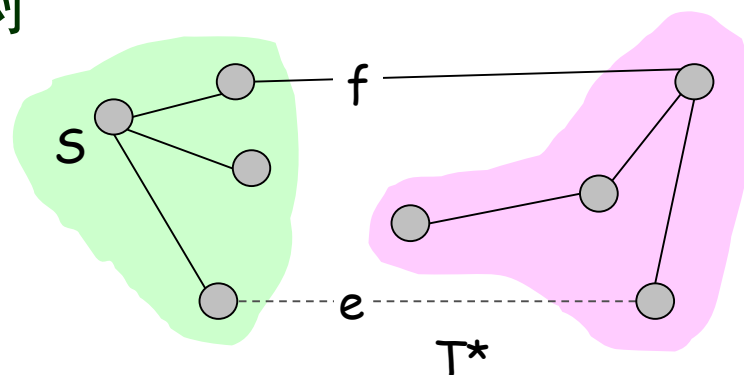
- 证明：（“交换”论证）

- 假设边 f 在 T^* 中
- 将 f 从 T^* 中删除将得到图 G 的一个割 S
- 假设 D 是 S 对应的割集
- 边 f 同时在回路 C 和割集 D 中，因此必定还存在图 G 中的另一条边 e 也同时在回路 C 和割集 D 中
- 边 e 的权值严格小于 f 的权值
- $T' = T^* \cup \{e\} - \{f\}$ 也是 G 的一棵支撑树
- 由 $c_e < c_f$ 可知 $\text{cost}(T') < \text{cost}(T^*)$
- 于是产生矛盾

极小连通

前述断言

连通且边数不变

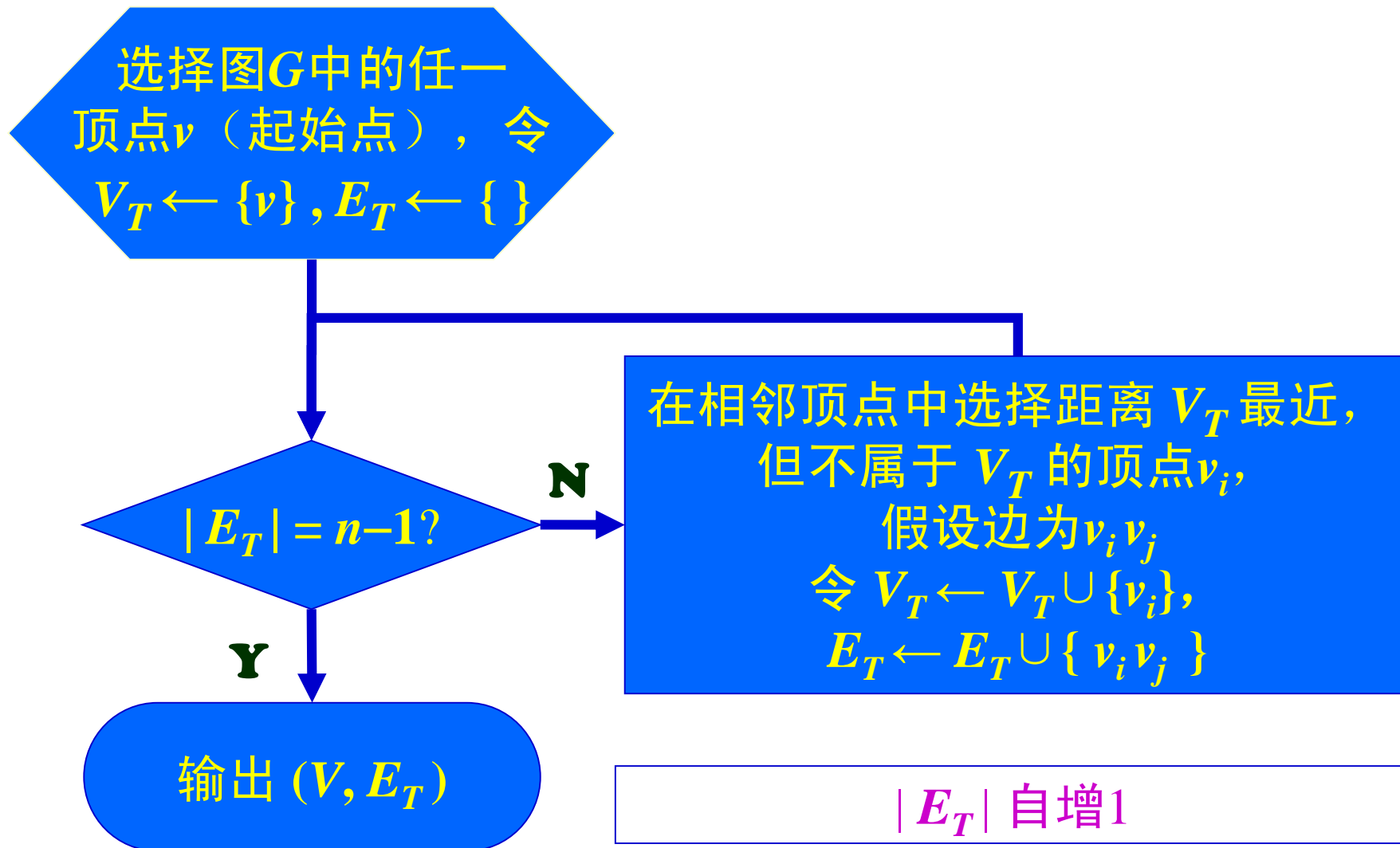


Prim算法的基本思想



- Prim算法 [Jarník 1930, Dijkstra 1957, Prim 1959]
- 通过增加具有最小权值的边（“贪婪”行为）使一棵从单顶点开始的树“生长”
 - 不断以最小代价“吸引”不在目前树中的点
 - 中间的部分解总是某个最小支撑树的子树

Prim算法



Prim算法的正确性证明



- Prim算法 [Jarník 1930, Dijkstra 1957, Prim 1959]

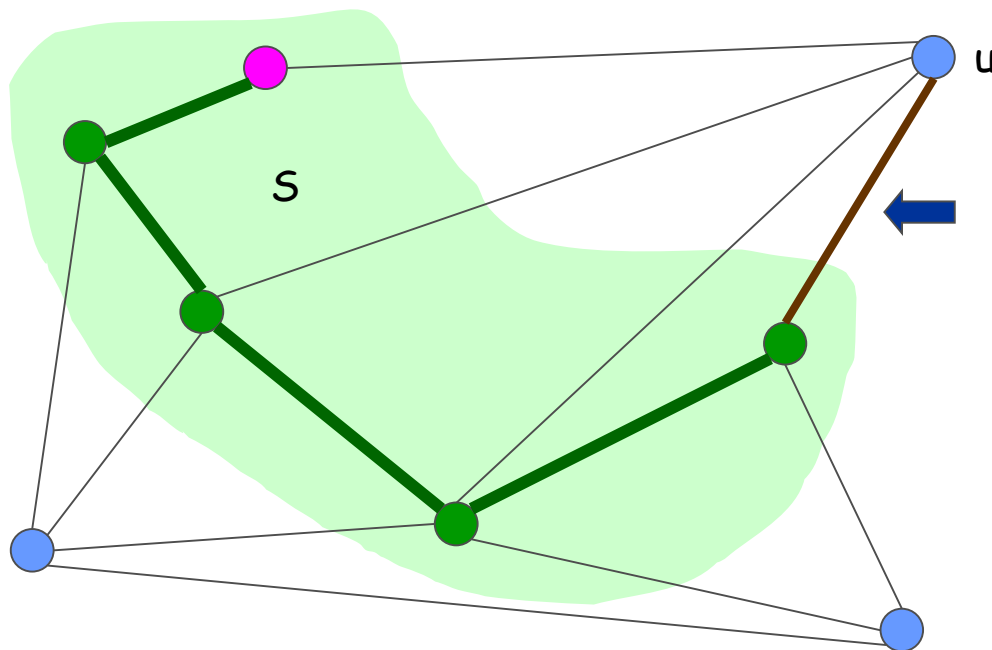
- 初始化 $V_T = \{\text{任一顶点}\}$

- 之后使用归纳法

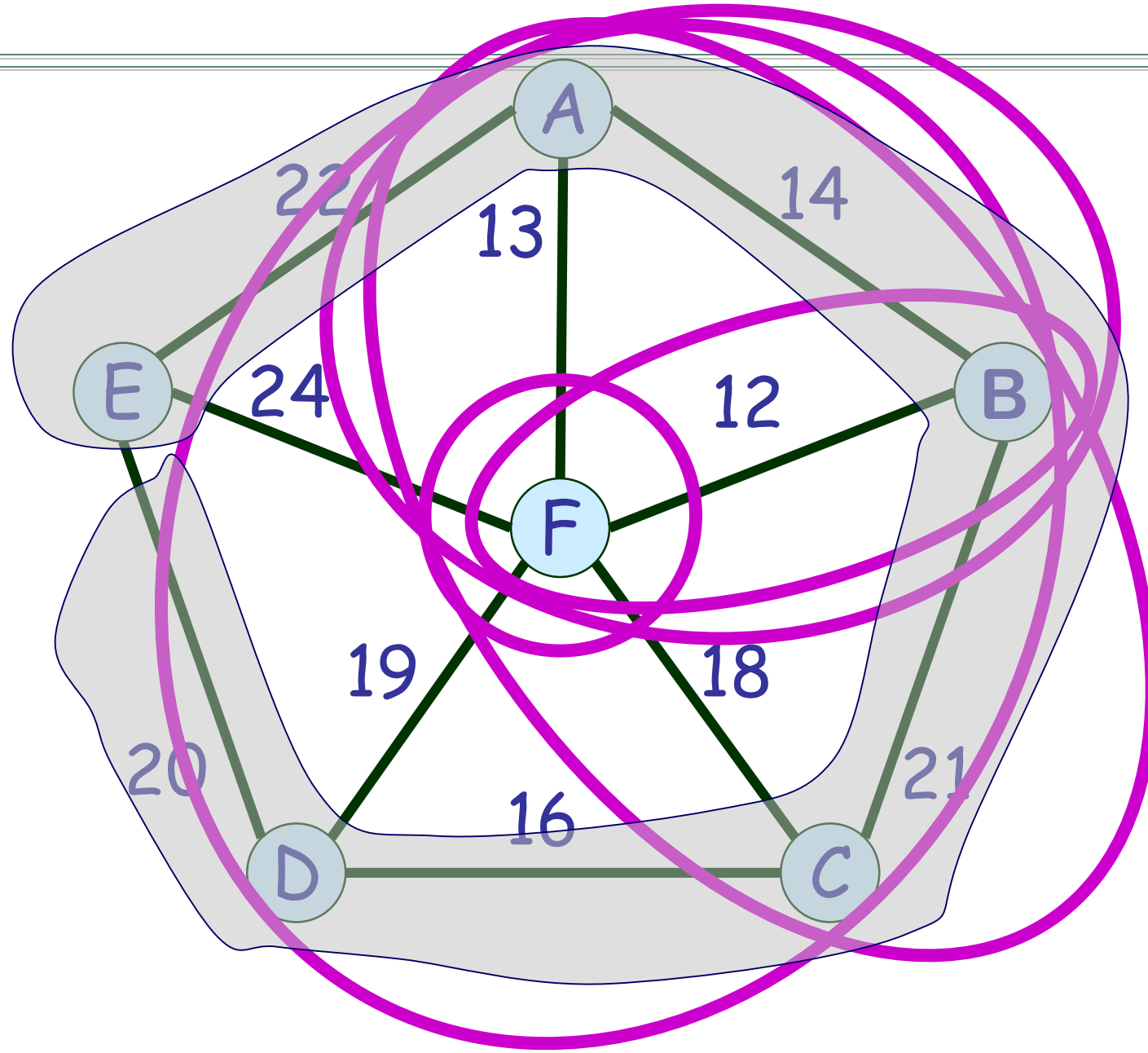
- 将“割性质”应用于 V_T

- 在对应于 V_T 的割集中选择权值最小的边，并向 V_T 中添加相应的新顶点 u

假定 各边的权值彼此不同



Prim算法



Prim 算法的伪代码



Algorithm Prim (G)

输入：赋权连通图 $G = (V, E)$

输出： G 的最小支撑树（之一）的边集合 E_T

1. $V_T \leftarrow \{v_0\}$ // v_0 可以是图中任一顶点
2. $E_T \leftarrow \emptyset$
3. **for** $i \leftarrow 1$ **to** $|V| - 1$ **do** //也可以写作 **while** $V - V_T \neq \emptyset$ **do**
4. 选择 $V - V_T$ 构成的割边集中的最轻边，
 即边 $e^* = (v^*, u^*)$ 满足 $v^* \in V_T$, $u^* \in V - V_T$ 且权值最小
5. $V_T \leftarrow V_T \cup \{u^*\}$
6. $E_T \leftarrow E_T \cup \{e^*\}$
7. **return** E_T

Prim 算法的实现



- 实现：使用优先级队列
 - 维护已访问（explored）顶点的集合 S （即前文中的 V_T ）
 - 对于每个未访问（unexplored）的顶点 v ，维护 $cost[v]$ 为 v 与 S 中顶点间所有边的权值的最小值
 - 维护前驱变量 $pred[v]$

Prim 算法的实现



- 实现：使用优先级队列，以邻接列表方式存储图
 - 维护已访问（explored）顶点的集合 S
 - 对于每个未访问（unexplored）的顶点 v ，维护 $\text{cost}[v]$ 为 v 与 S 中顶点间所有边的权值的最小值
 - 维护前驱变量 $\text{pred}[v]$

```
1. for each  $v \in V$  do
2.      $\text{cost}[v] \leftarrow \infty$ 
3.      $\text{pred}[v] \leftarrow \text{NIL}$ 
4.  $\text{cost}[v_0] \leftarrow 0$ 
5.  $S \leftarrow \emptyset$ 
6.  $H \leftarrow V$  //  $H$  是关于  $V - S$  的一个关于  $\text{cost}$  的优先级队列
7. while  $H \neq \emptyset$  do
8.      $u \leftarrow \text{EXTRACT-MIN}(H)$ 
9.      $S \leftarrow S \cup \{u\}$ 
10.    for each  $z \in \text{adj}(u) - S$  do
11.        if  $\text{cost}[z] > w(u, z)$  then
12.             $\text{cost}[z] \leftarrow w(u, z)$ 
13.             $\text{pred}[z] \leftarrow u$ 
```

$\text{adj}(u)$ 表示 u 的
相邻顶点集合

Prim 算法的实现



- 实现：使用优先级队列，以邻接列表方式存储图
 - n 次查找并调整 + $O(m)$ 次调整
 - 使用无序数组实现优先级队列的话总时间复杂度是 $O(n^2)$ ；使用二叉堆实现优先级队列的话时间复杂度是 $O((m+n) \log n)$ （也是 $O(m \log n)$ 的——因 n 是 $O(m)$ 的）

```
1. for each  $v \in V$  do
2.    $cost[v] \leftarrow \infty$ 
3.    $pred[v] \leftarrow \text{NIL}$ 
4.  $cost[v_0] \leftarrow 0$ 
5.  $S \leftarrow \emptyset$ 
6.  $H \leftarrow V$  //  $H$  是关于  $V - S$  的一个优先级队列
7. while  $H \neq \emptyset$  do
8.    $u \leftarrow \text{EXTRACT-MIN}(H)$ 
9.    $S \leftarrow S \cup \{u\}$ 
10.  for each  $z \in adj(u) - S$  do
11.    if  $cost[z] > w(u, z)$  then
12.       $cost[z] \leftarrow w(u, z)$ 
13.       $pred[z] \leftarrow u$ 
```

循环共 n 次

查找并调整

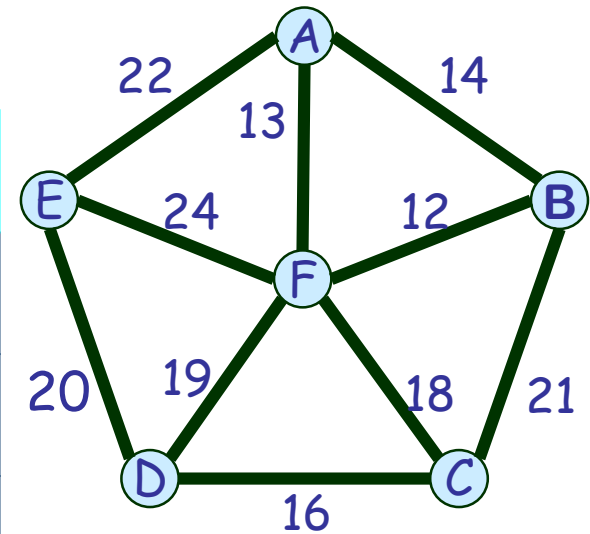
z 的数目 $\leq u$ 的度数
总的执行次数为 $O(m)$

调整

Prim 算法



S	A	B	C	D	E	F
{}	0/N	∞ /N	∞ /N	∞ /N	∞ /N	∞ /N
{A}		14/A	∞ /N	∞ /N	22/A	13/A
{A,F}		12/F	18/F	19/F	22/A	
{A,F,B}			18/F	19/F	22/A	
{A,F,B,C}				16/C	22/A	
{A,F,B,C,D}					20/D	
{A,F,B,C,D,E}						

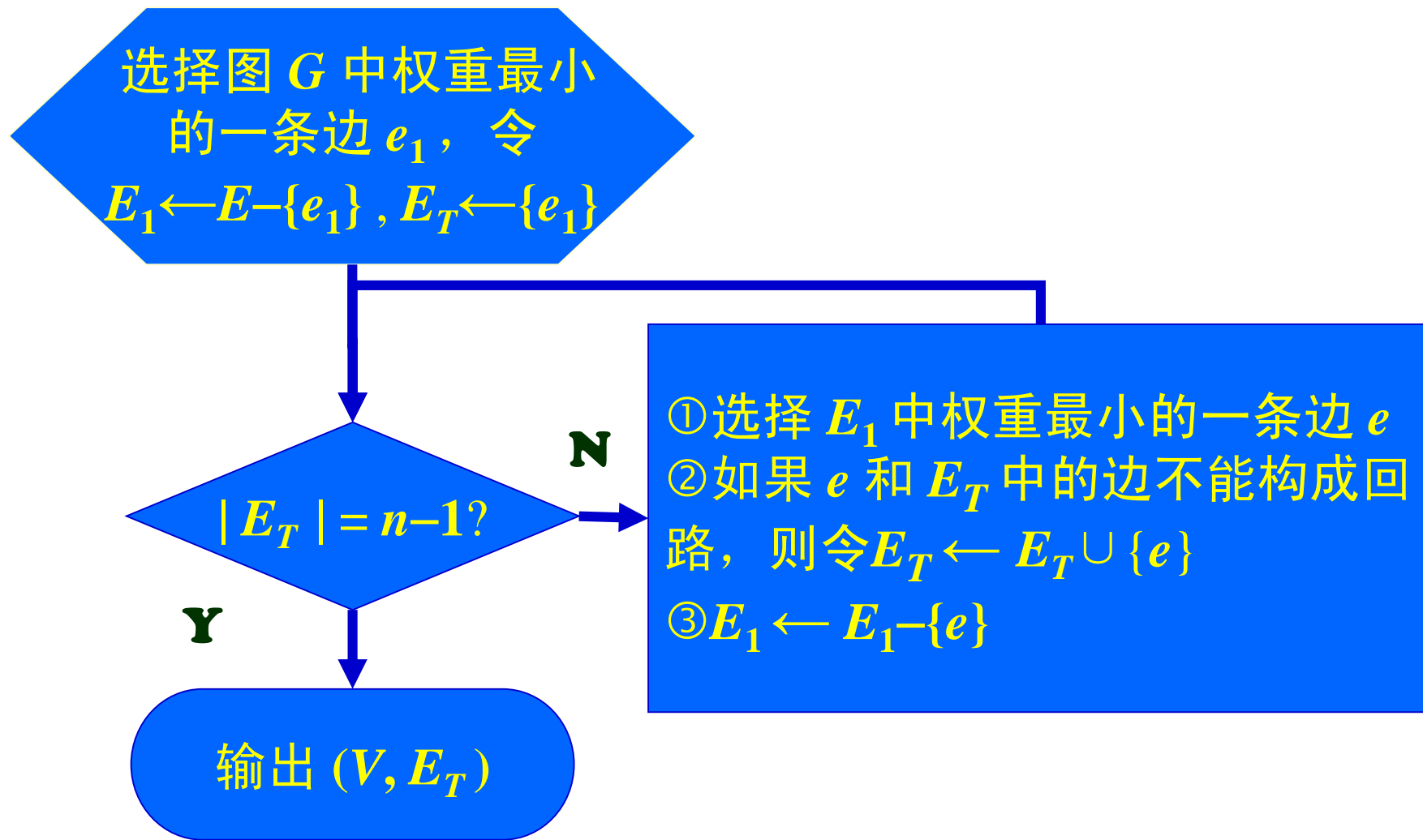


Kruskal算法 [Kruskal, 1956]



- 将所有边按照权值递增（不减）次序进行排序
- 将每个顶点都看做一棵孤立的树，初始的森林记为 F_0
- 每次添加一条边，不断将森林中的两棵不同的树合并为一棵新的树
- 得到一系列森林 F_1, F_2, \dots, F_{n-1}
- 每次添加的都是符合要求的、权值最小的边
 - 需要有效地检测/回避圈的方法
- 当所有顶点都在同一棵树中时，算法终止

Kruskal算法



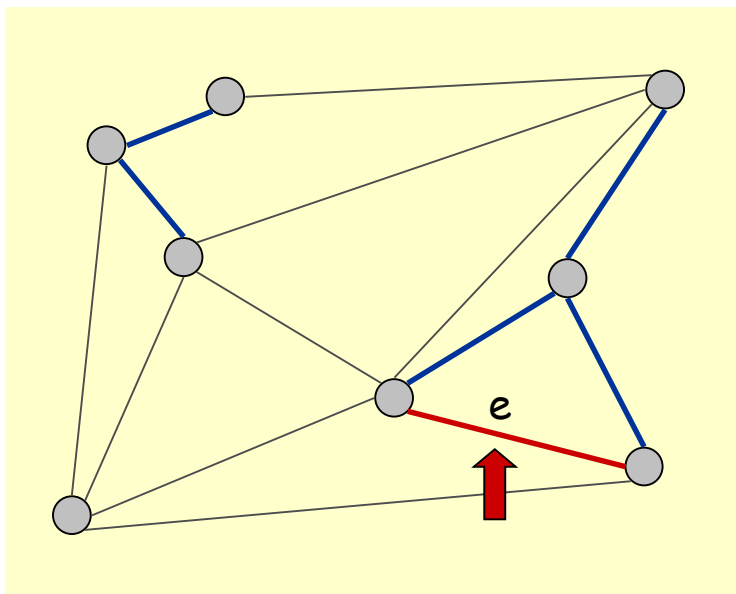
Kruskal算法的正确性证明



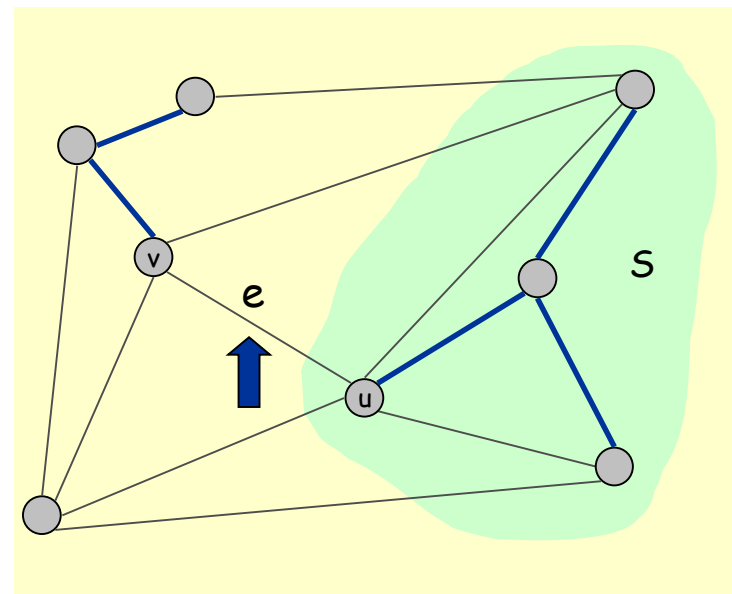
- Kruskal算法 [Kruskal, 1956]

假定 各边的权值彼此不同

- 以权值升序依次考虑各条边
- 情形1: 如果将 e 添加到 E_T 中会形成一个圈, 则根据“圈性质”可丢弃 e
- 情形2: 否则, 根据“割性质”可将 $e=(u, v)$ 加入 E_T , 其中割集对应的顶点的集合 S 选择为 u 所在的连通分支中的顶点集合



Case 1

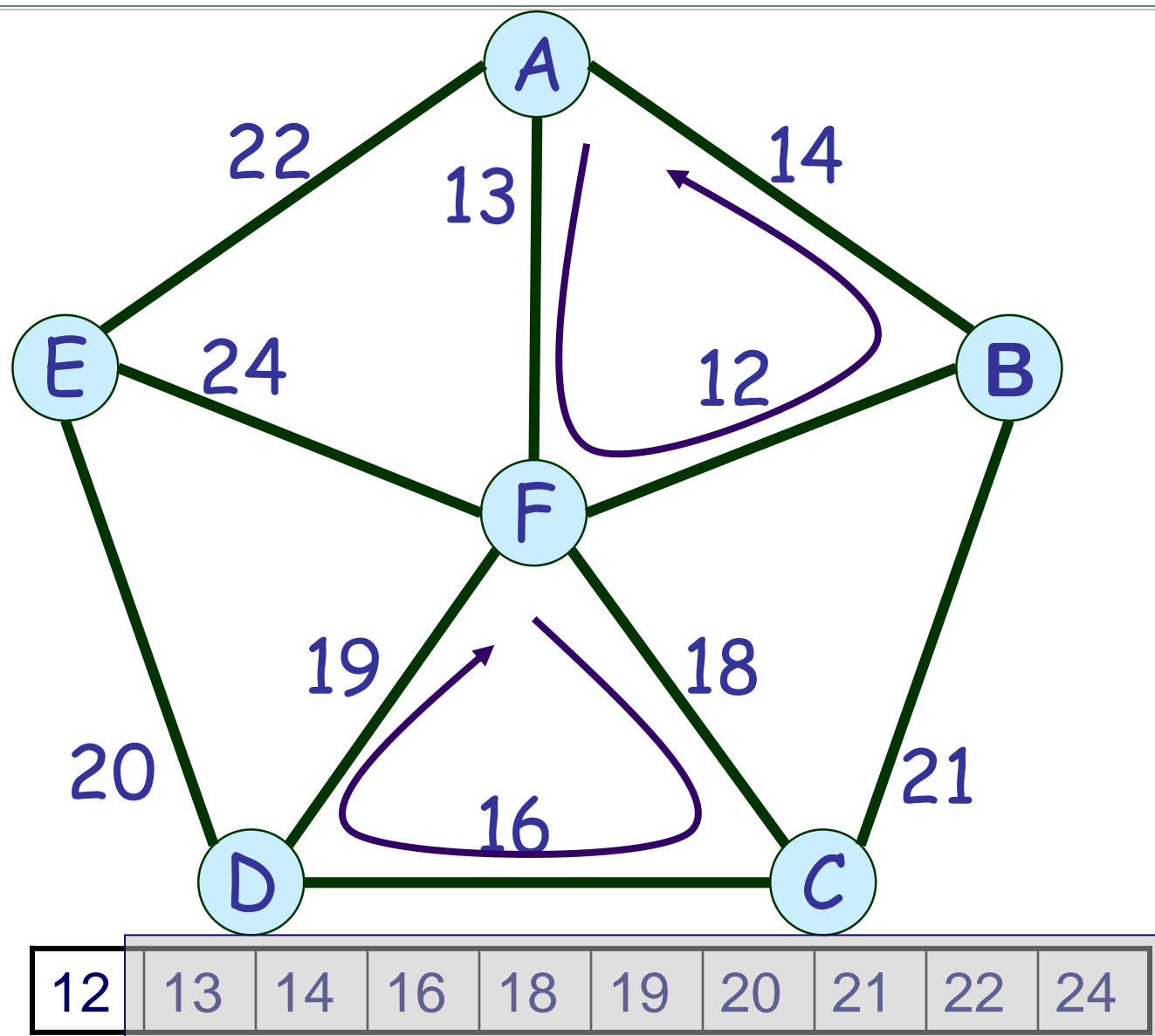


Case 2

Kruskal 算法



- 例



Kruskal 算法



Algorithm Kruskal (G)

输入：赋权连通图 $G = (V, E)$

输出： G 的最小支撑树（之一）的边集合 E_T

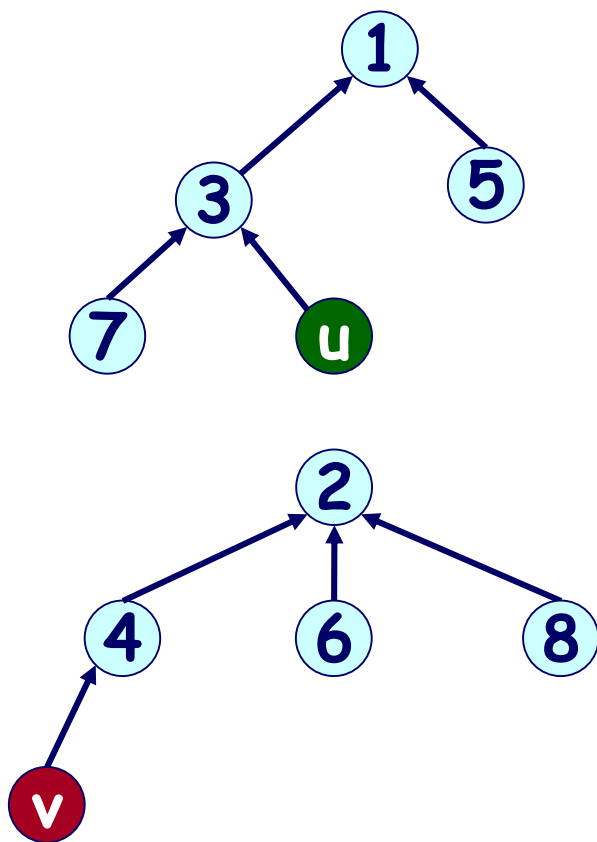
1. 将所有边按照权值递增（不减）次序进行排序为 e_1, e_2, \dots, e_m
2. $E_T \leftarrow \emptyset$
3. **for** $i=1$ **to** m //或者 **while** $|E_T| < n - 1$ **do**
4. 记 e_i 的两端为 u 和 v
5. **if** u 和 v 分属 (V, E_T) 中不同的连通分支 **then**
6. $E_T \leftarrow E_T \cup \{ e_i \}$
7. **return** E_T

Kruskal 算法



- 如何在添加新边时检查是否会形成圈？（即目前该边两端是否分属不同的连通分支）
 - 森林中的各棵树都使用（不相交的）集合（SET）表示
 - 若 $(u, v) \in E$ 且 u, v 在同一个集合中，那么添加边 $(u, v) \in E$ 后将会形成圈
 - 若 $(u, v) \in E$ 且 $u \in S_1, v \in S_2$ ，则将 S_1 和 S_2 合并
 - 查找（find）与 合并（union）

Kruskal 算法



将每个子集都表示为（特殊的）根树

（有向边的方向与通常的根树相反）

- 每棵这样的树的顶点数和相应连通分支中的元素个数都相等
 - 树的每个节点都唯一标记以子集中的某元素
- 用根来标记各个集合

注意：

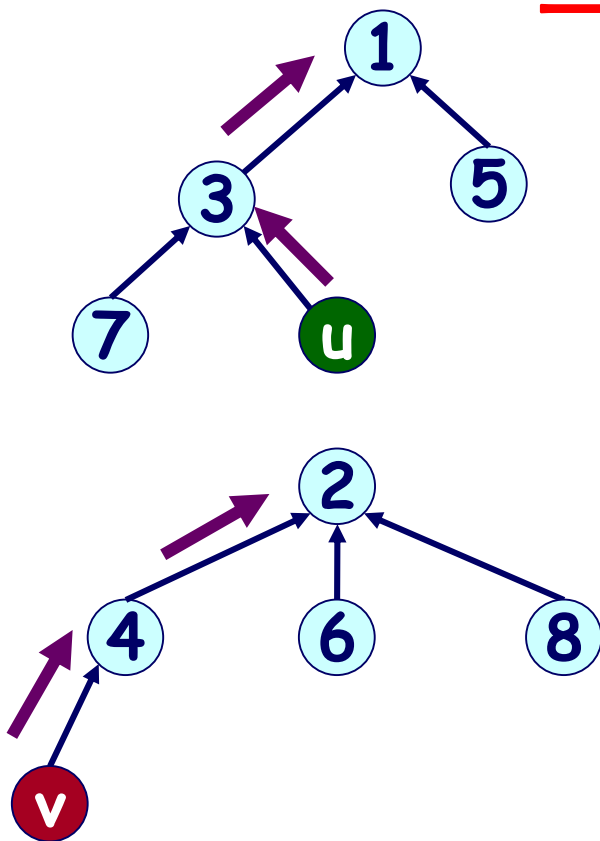
不要将这个“树”与试图构造的MST混淆

Kruskal 算法

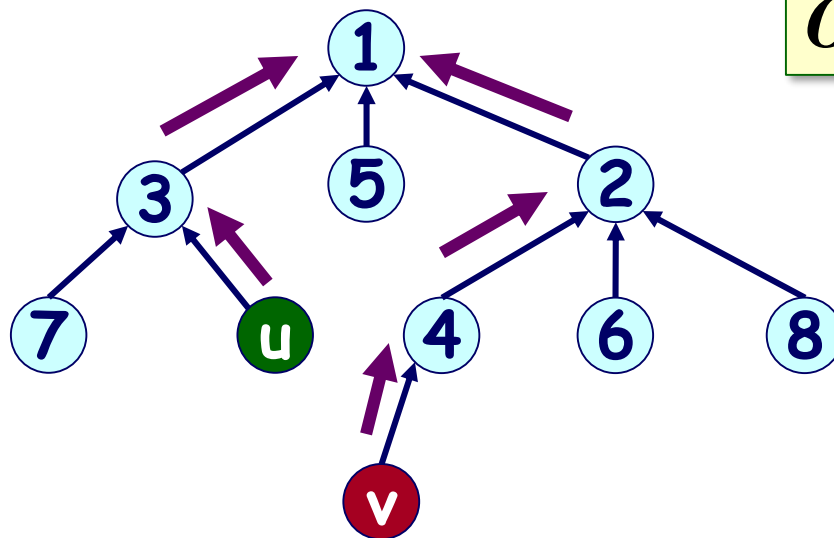


查找 (find)

查找的复杂度为
 $O(\text{所在根树的高度})$



u 和 v 在不同集合

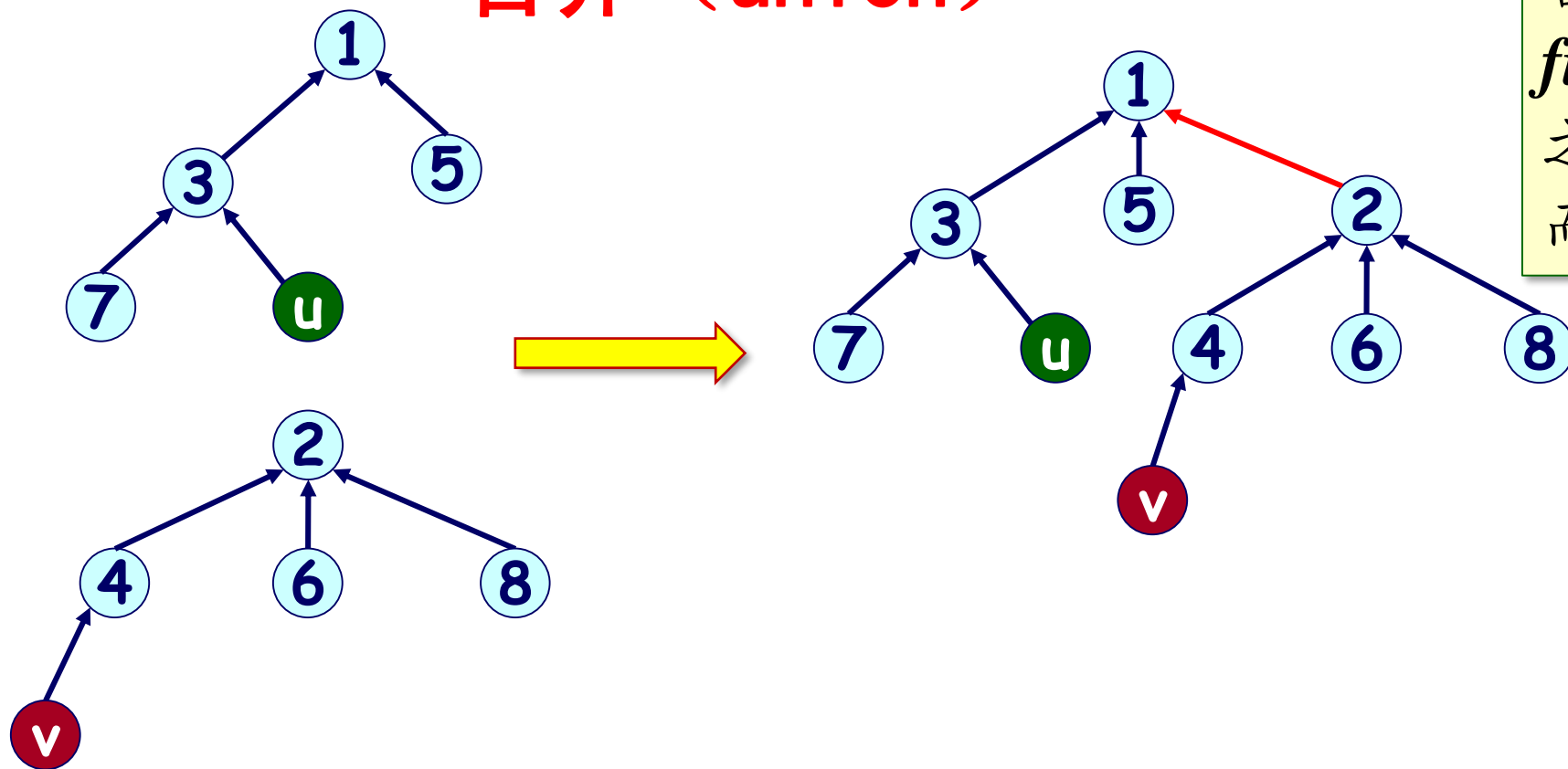


u 和 v 在同一集合

Kruskal 算法



合并 (union)



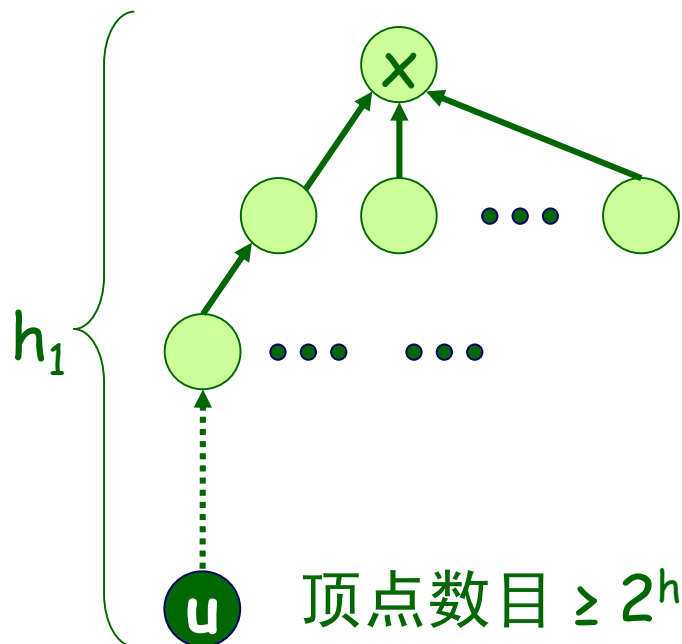
合并的复杂度由 $find(u)$ 和 $find(v)$ 决定
之后添加一条有向边
而已

u 和 v 在不同集合

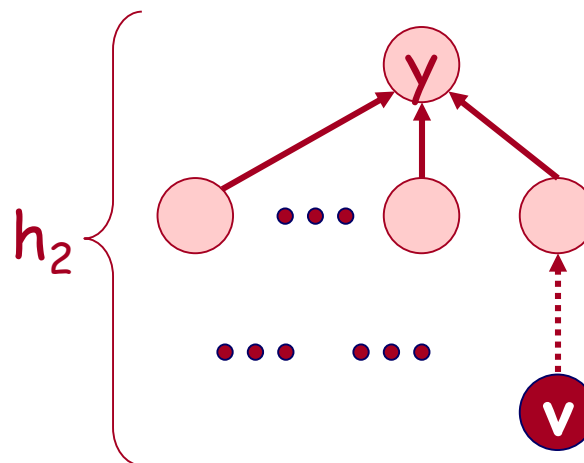
Kruskal 算法



合并 (union)



顶点数目 $\geq 2^{h_1}$



顶点数目 $\geq 2^{h_2}$

可以通过之后两页
的构造方式对此进
行归纳证明

等价于 $h_1 \leq \log_2(\text{顶点数目})$

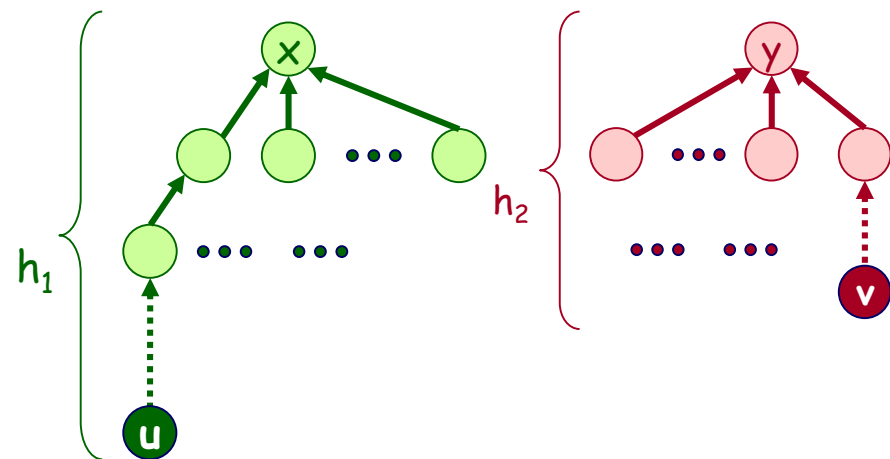
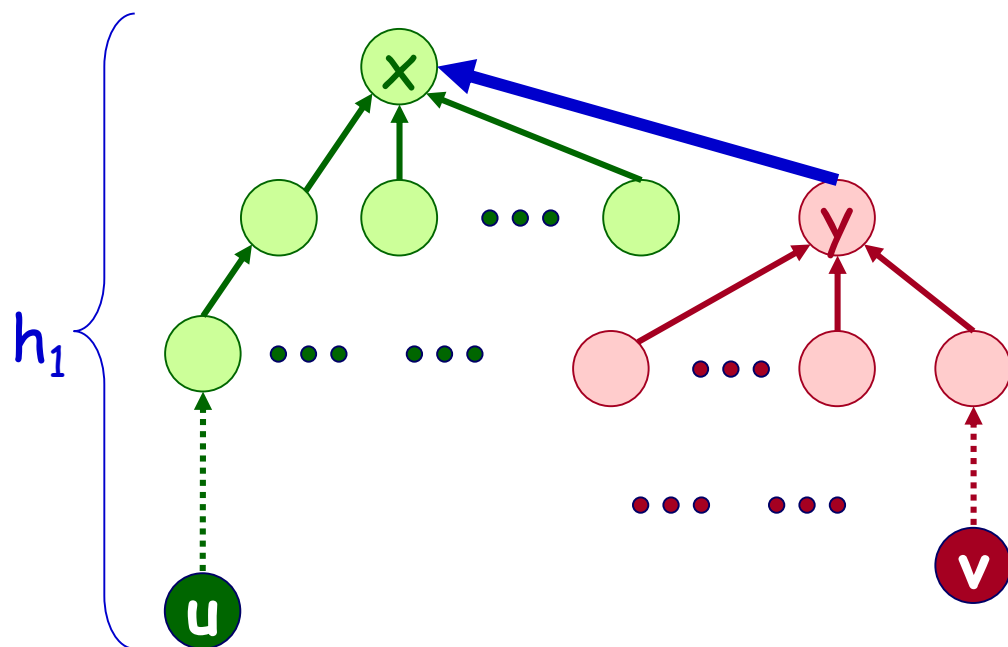
不失一般性地假设 $h_1 \geq h_2$

于是所有根树的最大可能高度为 $O(\log n)$
因此查找的复杂度为 $O(\log n)$

Kruskal 算法



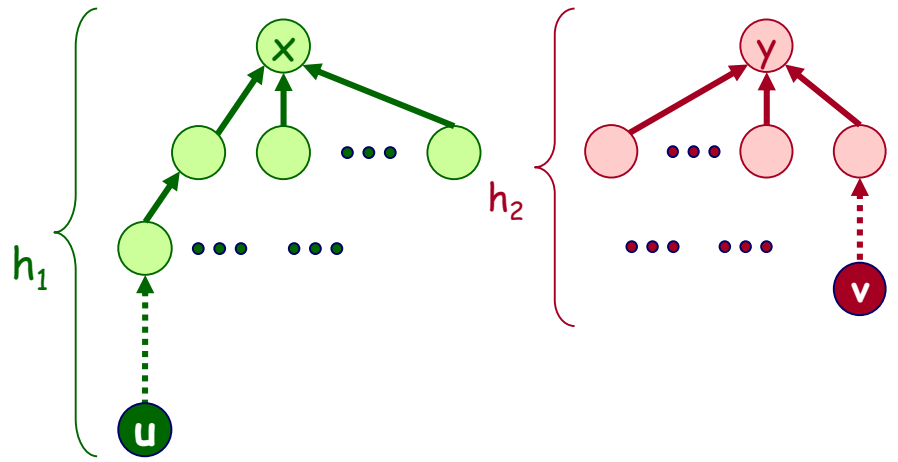
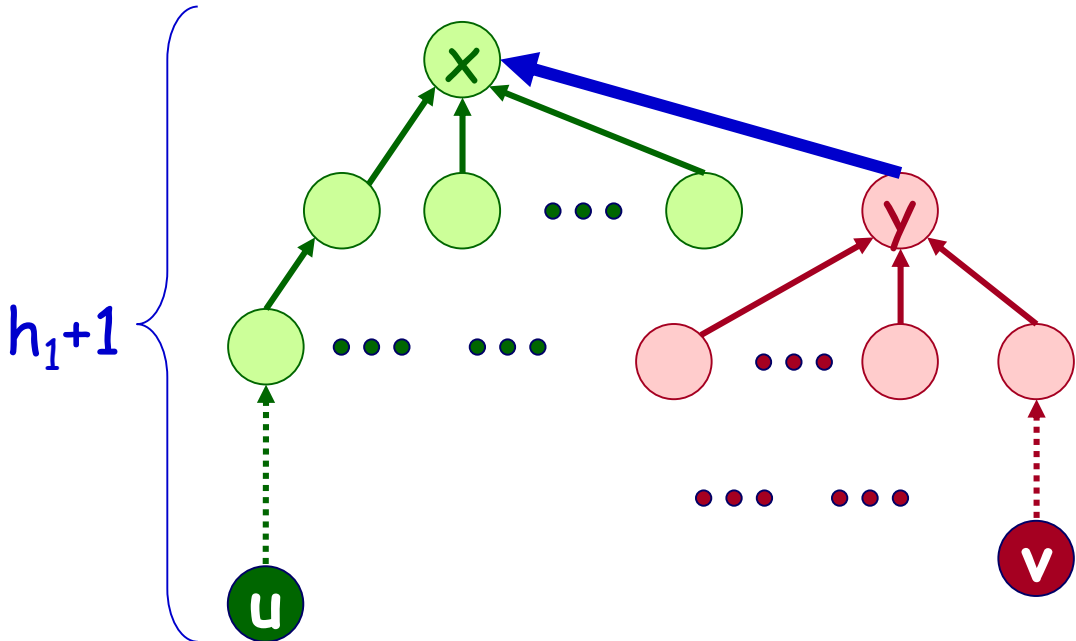
1. $h_1 > h_2$



Kruskal 算法



2. $h_1 = h_2$



Kruskal 算法的实现



- 实现：使用并查集数据结构

- 排序： $O(m \log n)$

$m \leq n^2 \Rightarrow \log m$ 是 $O(\log n)$ 的

- 查找与合并： $O(\log n)$

Algorithm Kruskal (G)

1. 将所有边按照权值递增（不减）次序进行排序为 e_1, e_2, \dots, e_m

2. $E_T \leftarrow \emptyset$

3. **foreach** ($u \in V$) **do** 为每个 u 建立一个集合

4. **for** $i = 1$ **to** m

意即： u 和 v 是否在不同的连通分支（子树）中？

5. $(u, v) \leftarrow e_i$

6. **if** u 和 v 在不同的集合中 **then**

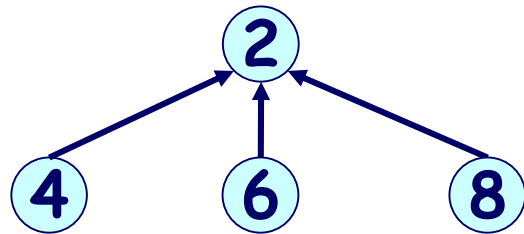
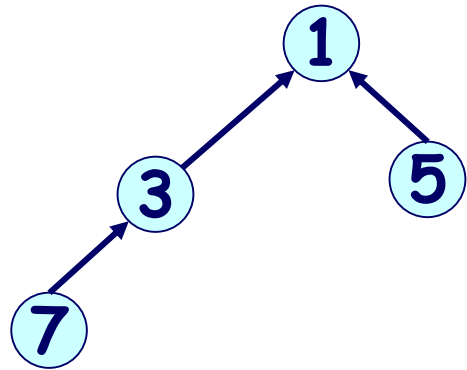
7. $E_T \leftarrow E_T \cup \{ e_i \}$

8. 合并 u 和 v 所在的集合

将两个不同的连通分支（子树）进行合并（为一棵新的子树）

9. **return** E_T

Kruskal 算法



x	f[x]	height
1	1	2
2	2	1
3	1	无意义
4	2	无意义
5	1	无意义
6	2	无意义
7	3	无意义
8	2	无意义

Initial

foreach ($u \in V$)

$f[u] \leftarrow u$

$height[u] \leftarrow 0$

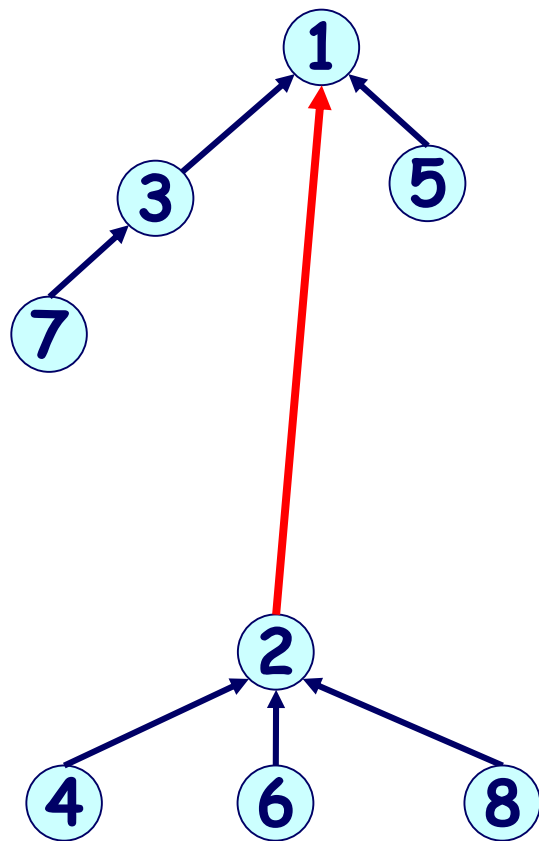
Find (x)

while not ($f[x] = x$)

$x \leftarrow f[x]$

return x

Kruskal 算法



x	f[x]	height
1	1	2
2	1	无意义
3	1	无意义
4	2	无意义
5	1	无意义
6	2	无意义
7	3	无意义
8	2	无意义

Union (x, y)

$x \leftarrow \text{Find}(x), y \leftarrow \text{Find}(y)$

if $\text{height}[x] < \text{height}[y]$ **then**

$f[x] \leftarrow y$

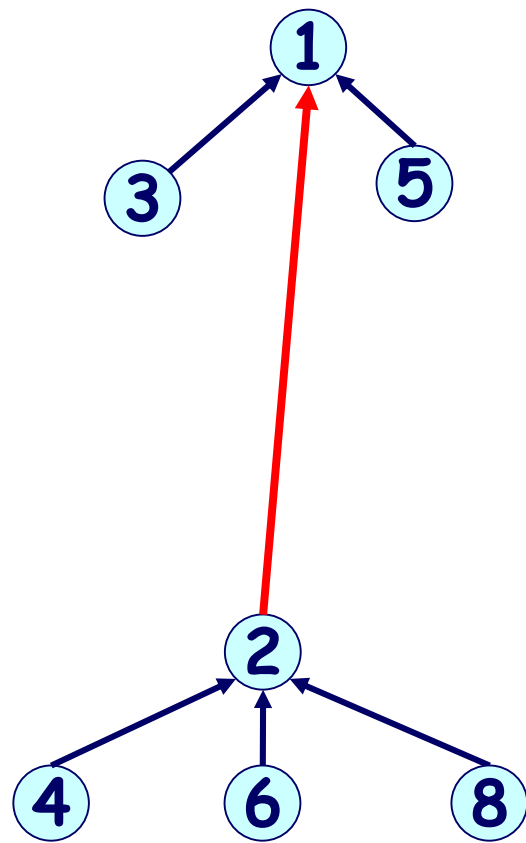
else

$f[y] \leftarrow x$

if $\text{height}[x] = \text{height}[y]$ **then**

$\text{height}[x] \leftarrow \text{height}[x] + 1$

Kruskal 算法



x	f[x]	height
1	1	2
2	1	无意义
3	1	无意义
4	2	无意义
5	1	无意义
6	2	无意义
7	3	无意义
8	2	无意义

```
Union ( x, y )
  x ← Find(x), y ← Find(y)
  if height[x] < height[y] then
    f[x] ← y
  else
    f[y] ← x
  if height[x] = height[y] then
    height[x] ← height[x] + 1
```

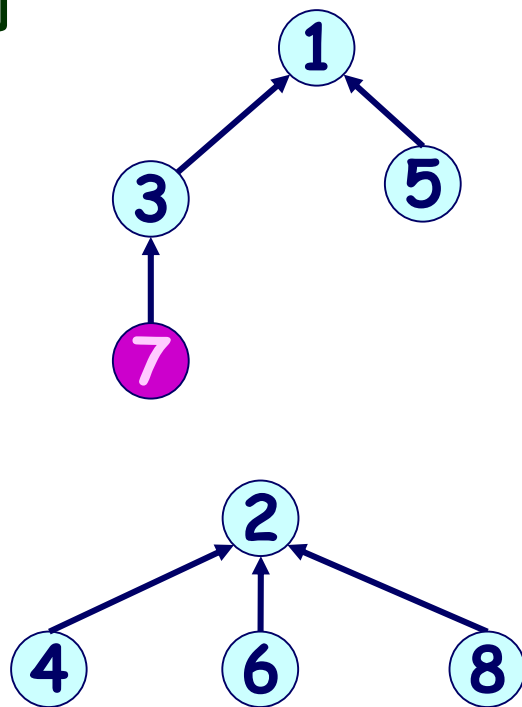
Kruskal 算法



- 道路压缩

- 在调用 **Find**(x) 后, 执行

- $f[x] \leftarrow \text{Find}(x)$

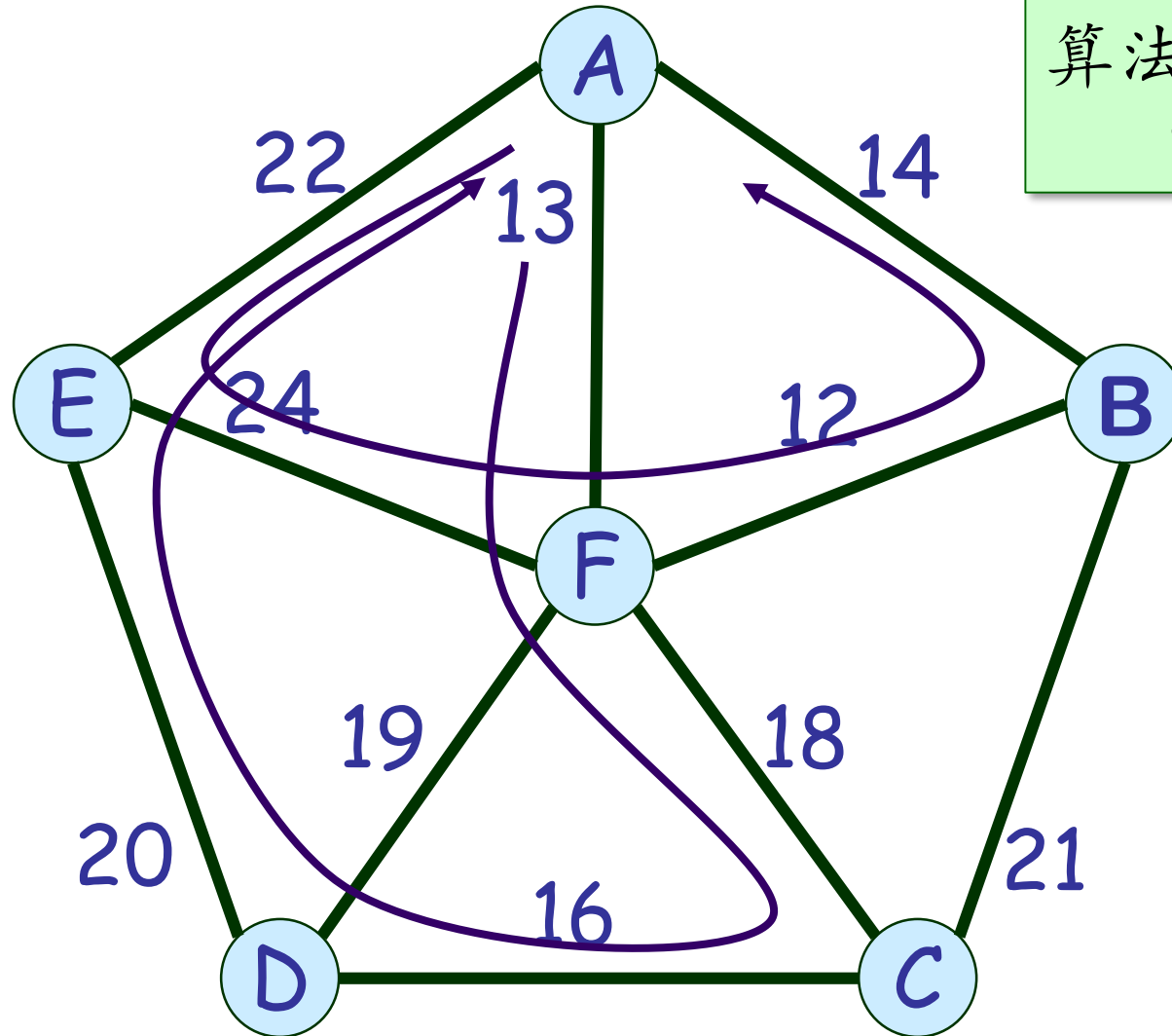


x	f[x]	height
1	1	2
2	2	1
3	1	无意义
4	2	无意义
5	1	无意义
6	2	无意义
7	1	无意义
8	2	无意义

Reverse delete算法



算法正确性证明：
“圈性质”

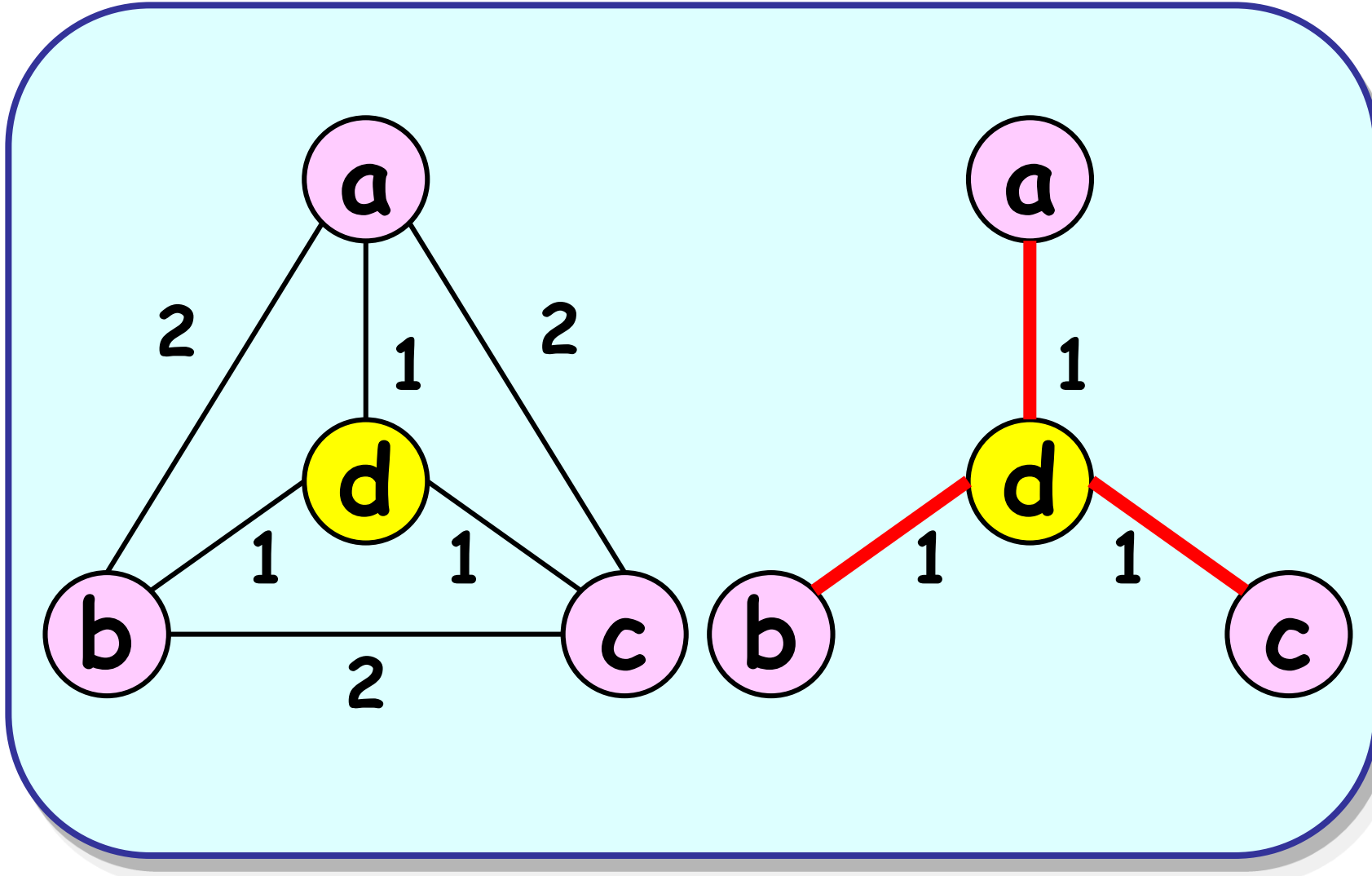


Steiner 树

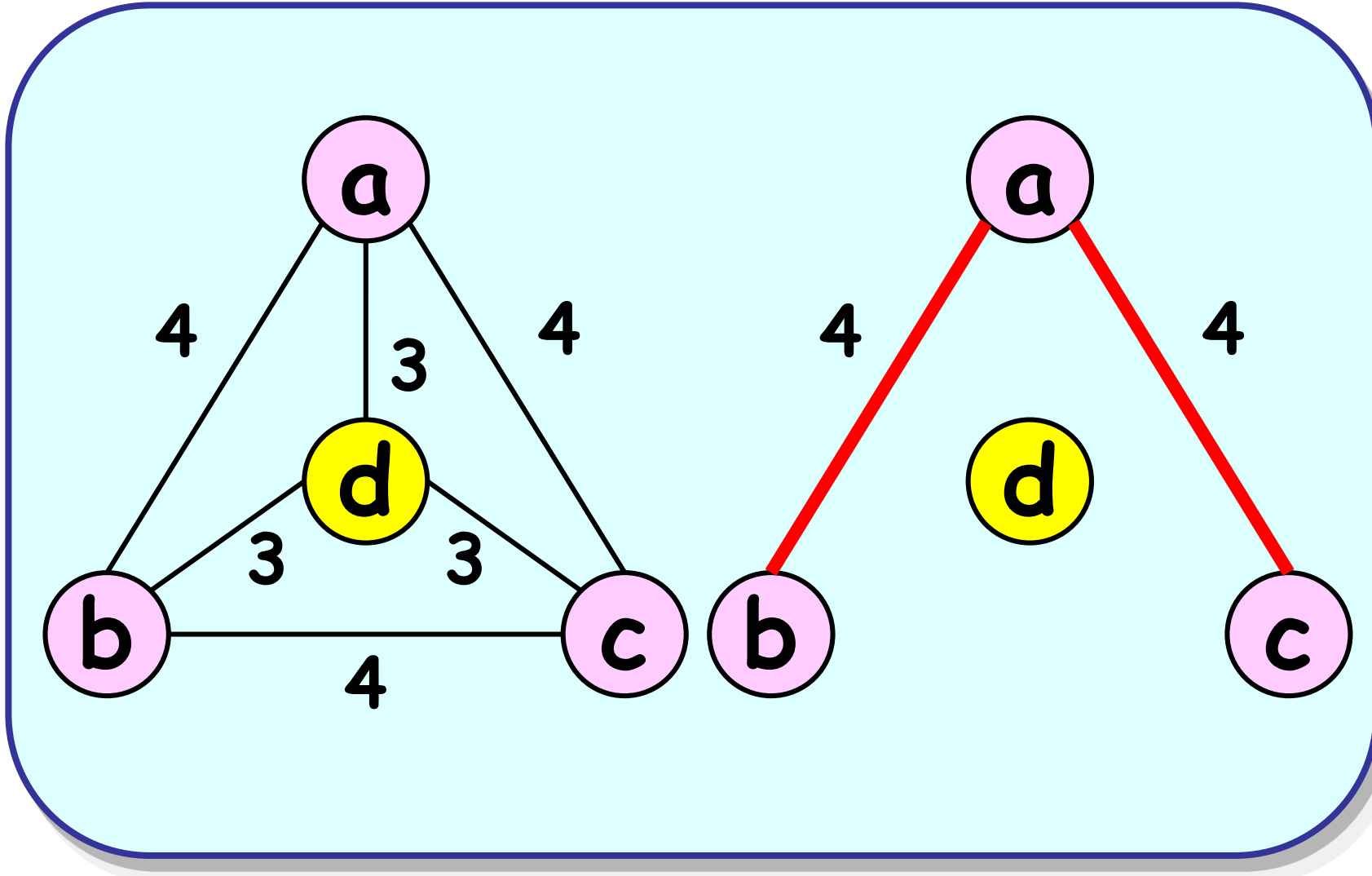


- 设 $(G=(V, E), W)$ 是无向连通赋权图, $R \subseteq V$, 在 G 的所有包含 R 中所有顶点的子图中, 总权值最小的树称为 G 的**斯坦纳树 (Steiner tree)**
- 当 $R = V$ 时, 斯坦纳树问题即是最小支撑树问题

Steiner 树



Steiner 树



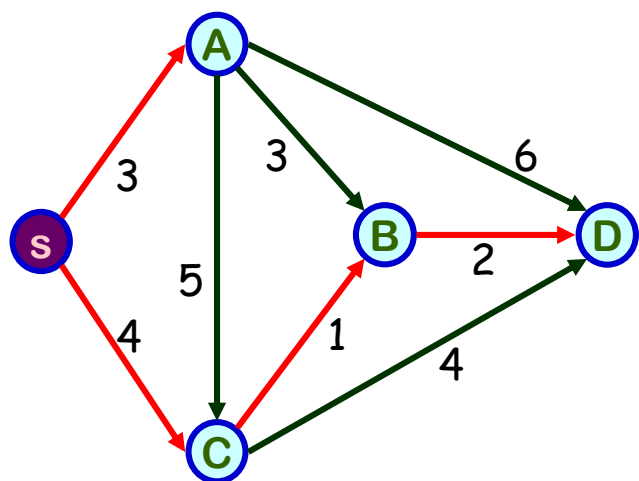
Steiner 树



- 虽然斯坦纳树问题与最小支撑树问题具有相似之处，然而斯坦纳树问题则难得多
- 属于NPC问题，事实上，它是卡普（Richard Manning Karp）证明的第一批21个NPC问题之一

单源最短道路

(*Single Source Shortest Path, SSSP*)

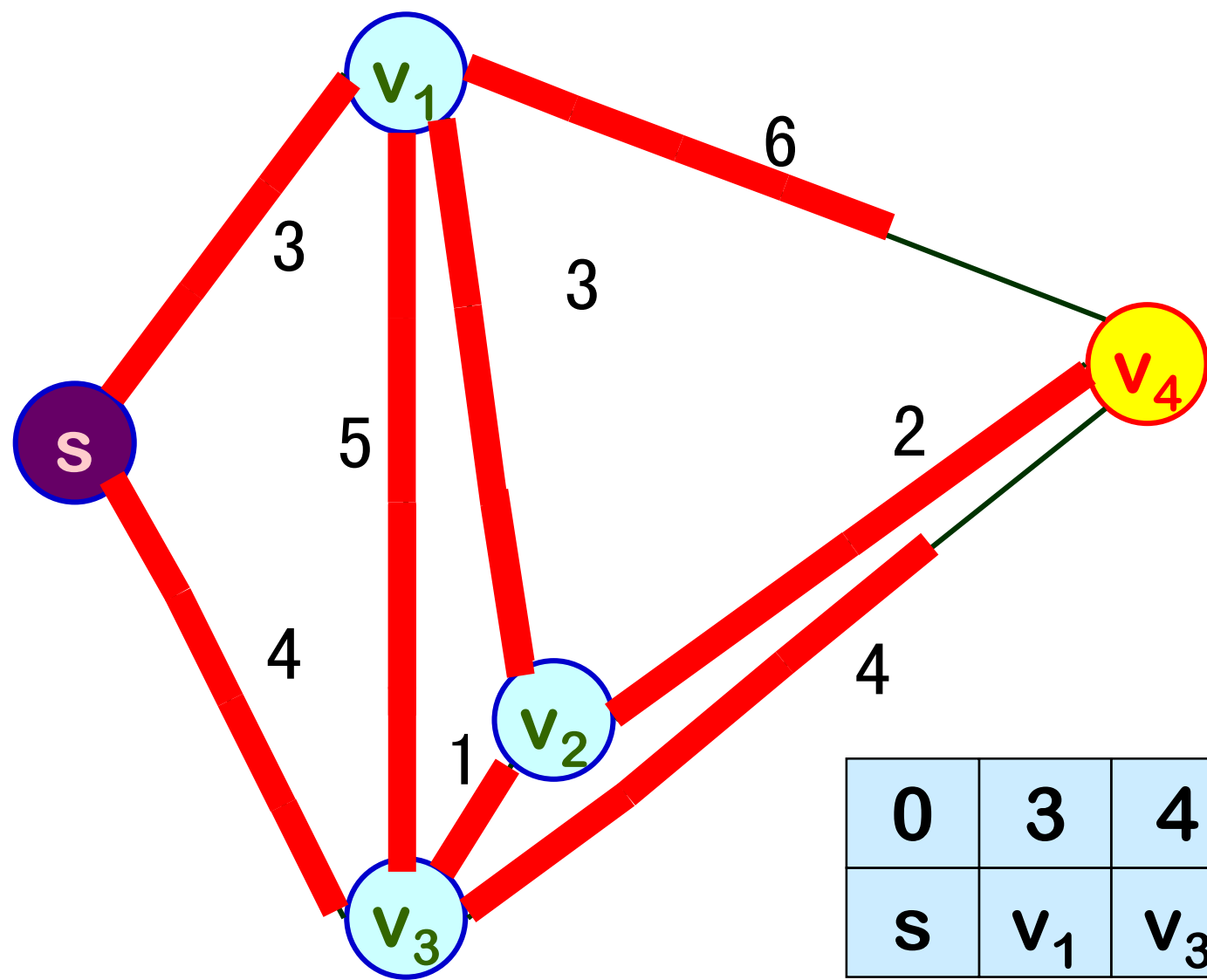


刘铎

liuduo@bjtu.edu.cn



单源最短道路

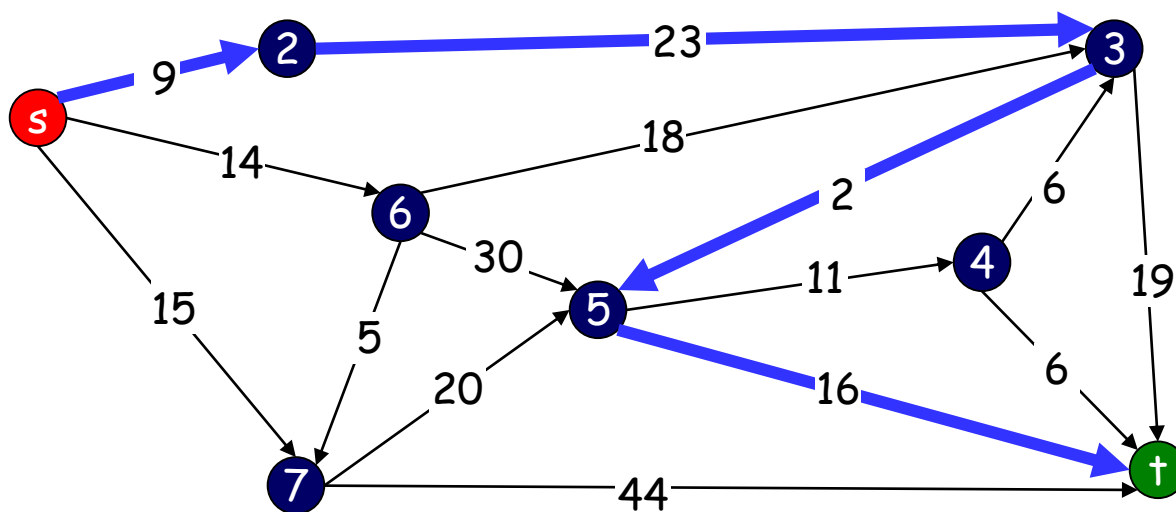


0	3	4	5	7
s	v ₁	v ₃	v ₂	v ₄

最短道路问题



- 假设 $G=(V, E, W)$ 为赋权图，边权值都是非负值，则图中一条道路的**长度 (length)** 是指该道路中各条边权重之和
- 一对给定顶点之间的一条最短道路指的是：给定顶点 s , $t \in V$ ，找到从 s 开始到 t 结束的最小赋权道路（之一）



道路 $s-2-3-5-t$ 的长度
= $9 + 23 + 2 + 16$
= 48

单源最短道路

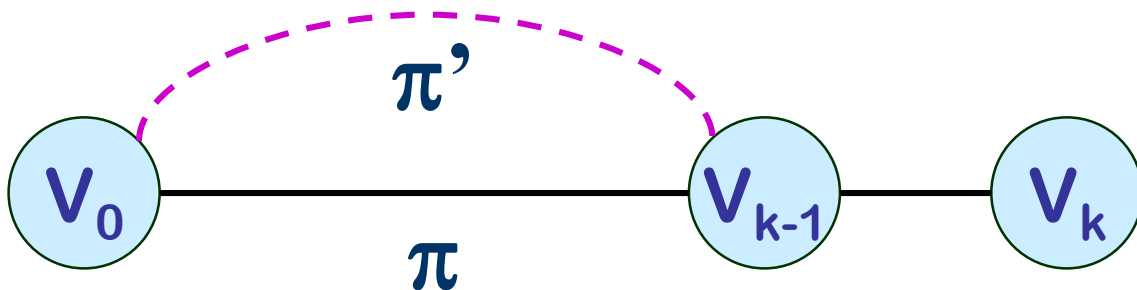


- 定理

- 设 $\pi: v_0, v_1, \dots, v_{k-1}, v_k$ 是图 G 中顶点 v_0 到 v_k 的一条最短道路，则 v_0, v_1, \dots, v_{k-1} 是 v_0 到 v_{k-1} 的最短道路（之一）

- 证明

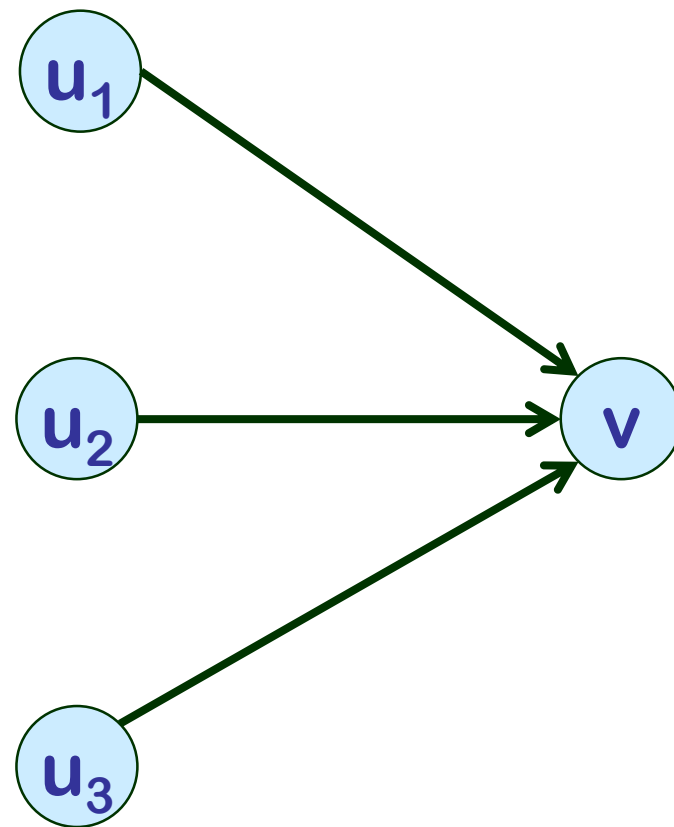
- 若存在另一条从 v_0 到 v_{k-1} 的更短的道路 v_0, u_1, \dots, v_{k-1} ，则 $\pi': v_0, u_1, \dots, v_{k-1}, v_k$ 是 v_0 到 v_k 的更短的道路，产生矛盾



单源最短道路



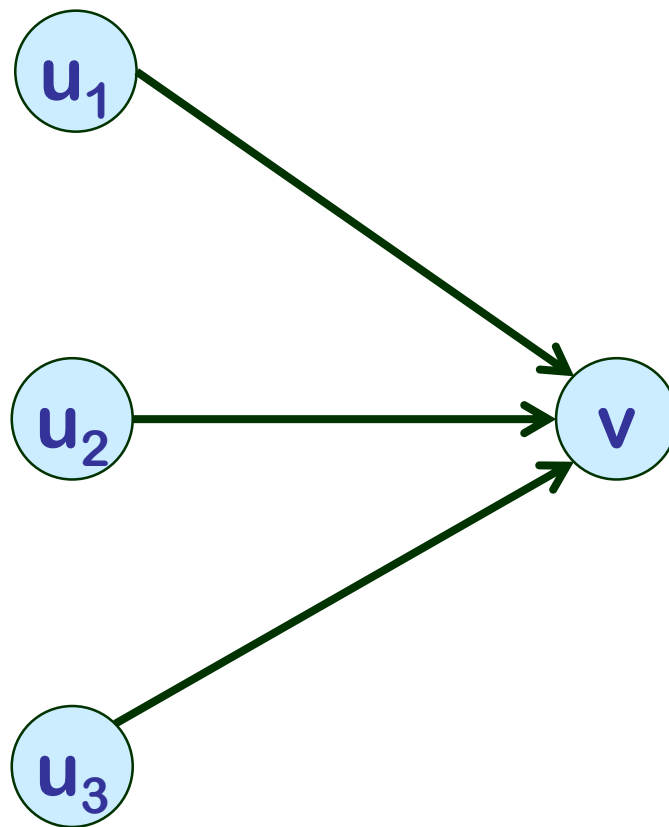
- 假设以下的所有 $d(u)$ 均已计算
- 而 $d(v)$ 尚未计算
- $d(v) = \min\{ d(u) + l_{uv} \}$,
其中 $u \in V$ 且 uv 相邻



单源最短道路



- $d(v) = \min\{ d(u) + l_{uv} \}$, 其中 $u \in V$ 且 uv 相邻



单源最短道路



- 定理

- 假设图 G 是一个赋权图，边权值都是非负值， s 是 G 的一个顶点（单源），则存在 G 的一棵支撑（根）树 T ，使得在 T 中 s 到任意顶点 u 的道路，就是 G 中 s 到 u 的最短道路（之一）
- 称 T 为 G 的**最短道路树**
 - 若 G 是无向图，则 T 是 G 的一棵支撑树
 - 若 G 是有向图，则 T 是 G 的一棵支撑根树
- Dijkstra于1959年给出了求唯一源点到其它各顶点的最短道路（之一）的算法

Dijkstra 算法

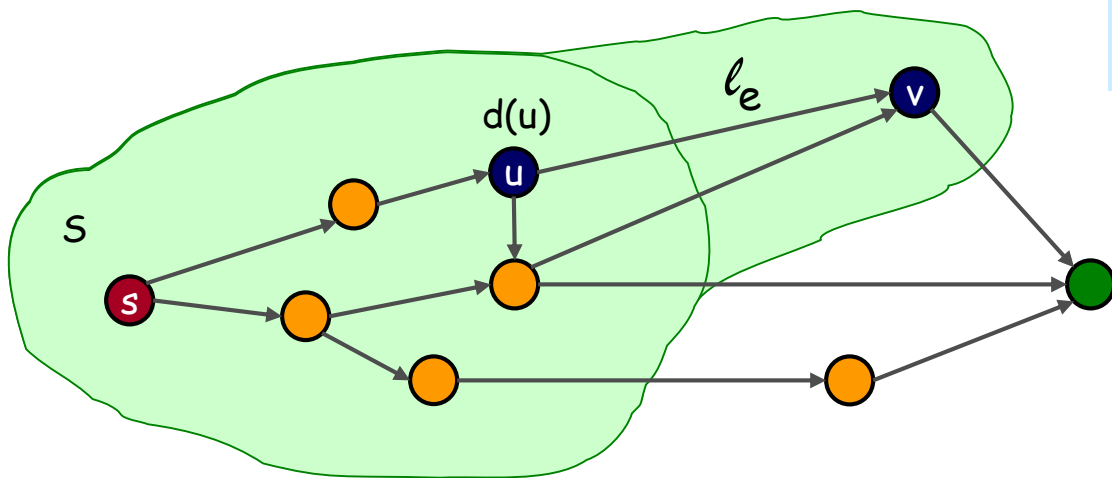


- 维护一个已探索 (explored) 顶点的集合 S
- 记 $d(v)$ 表示从图 G 中给定顶点 s 到 v 的最短道路的长度
- 初始化 $S \leftarrow \emptyset, d(s) \leftarrow 0$
- 不断选择使下式达到最小值的未探索顶点 v

$$\pi(v) = \min_{e = (u, v) : u \in S} d(u) + \ell_e,$$

将 v 将加入 S , 并令 $d(v) \leftarrow \pi(v)$

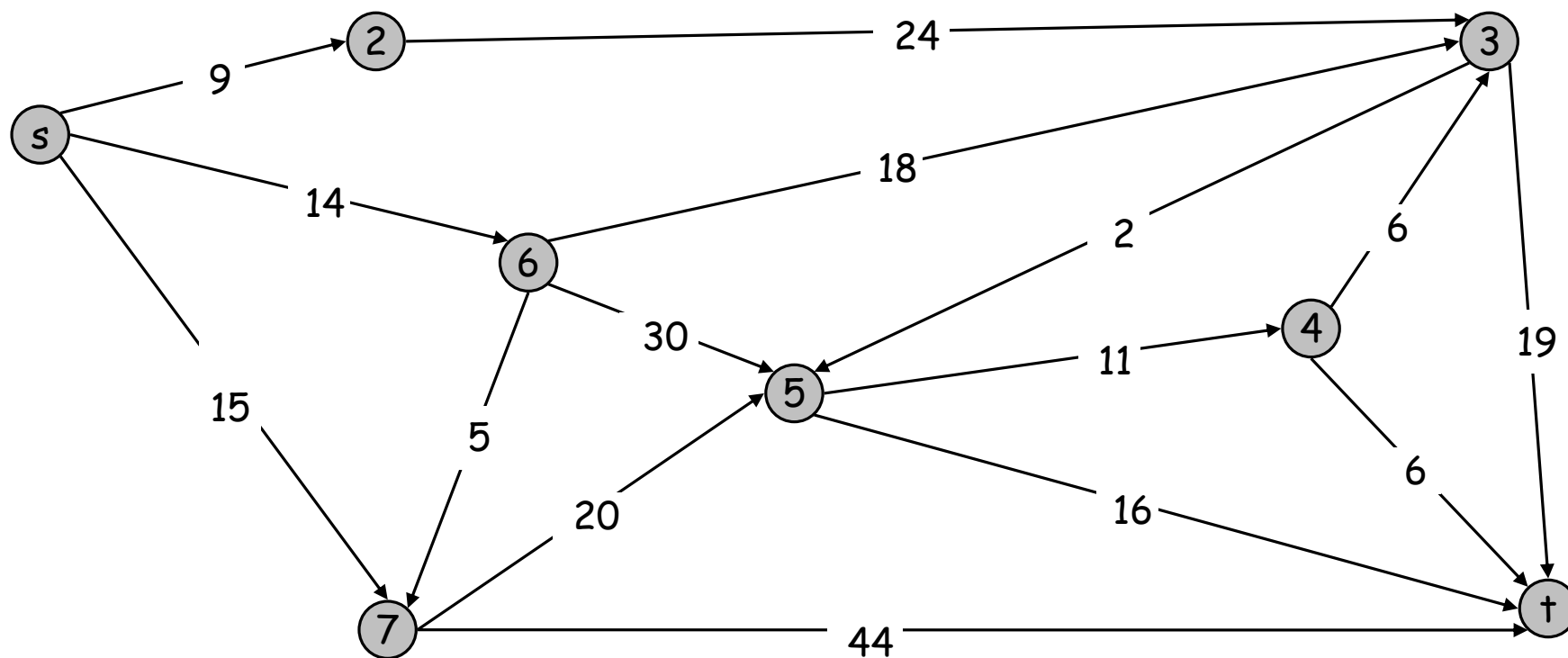
到某个已探索顶点 u
的最短道路之后附加
一条边 (u, v)



Dijkstra 算法



- 找到从 s 到 t 的最短道路



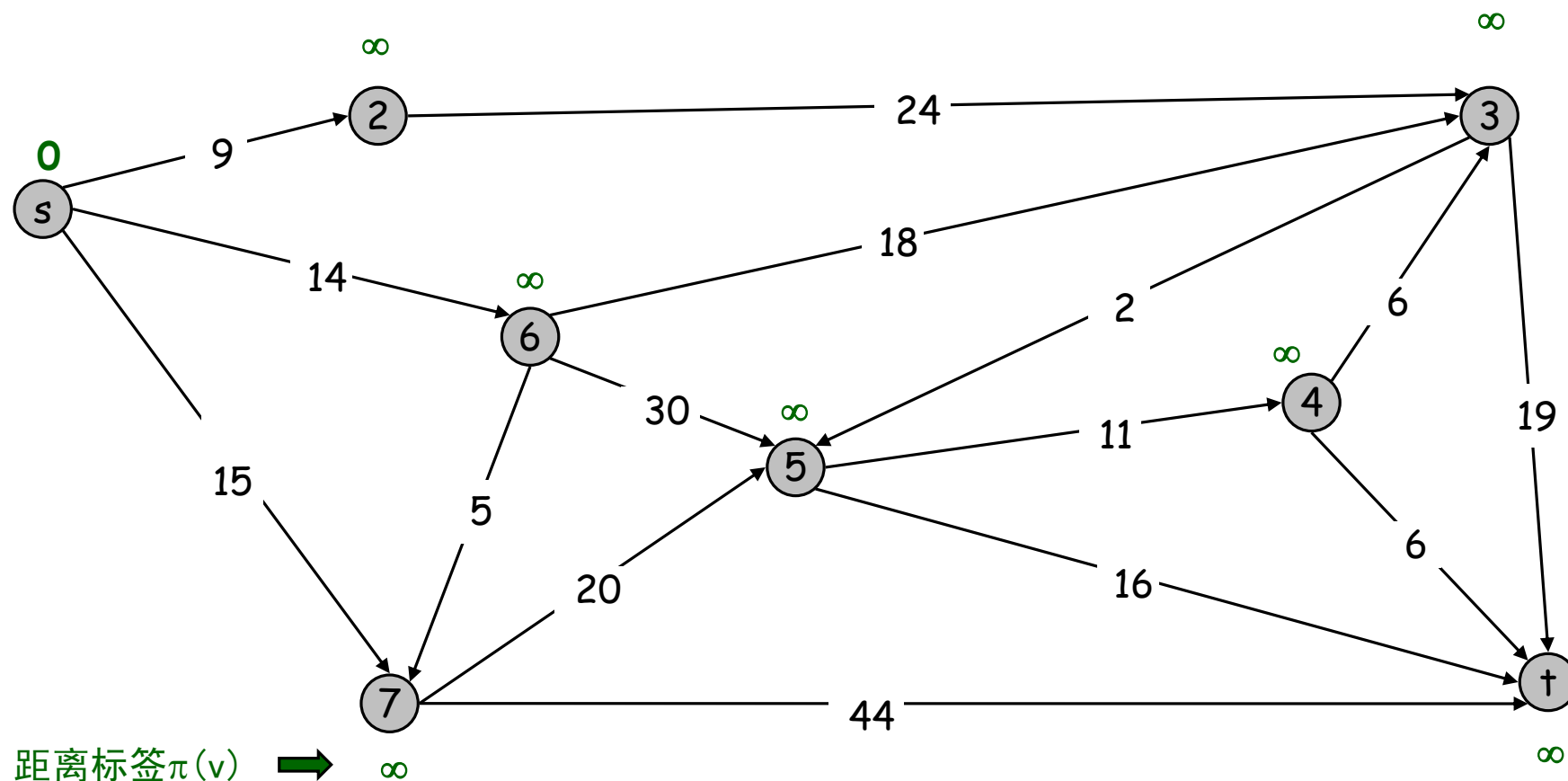
Dijkstra 算法



$S = \{ \}$

$PQ = \{ s, 2, 3, 4, 5, 6, 7, \dagger \}$

优先级队列

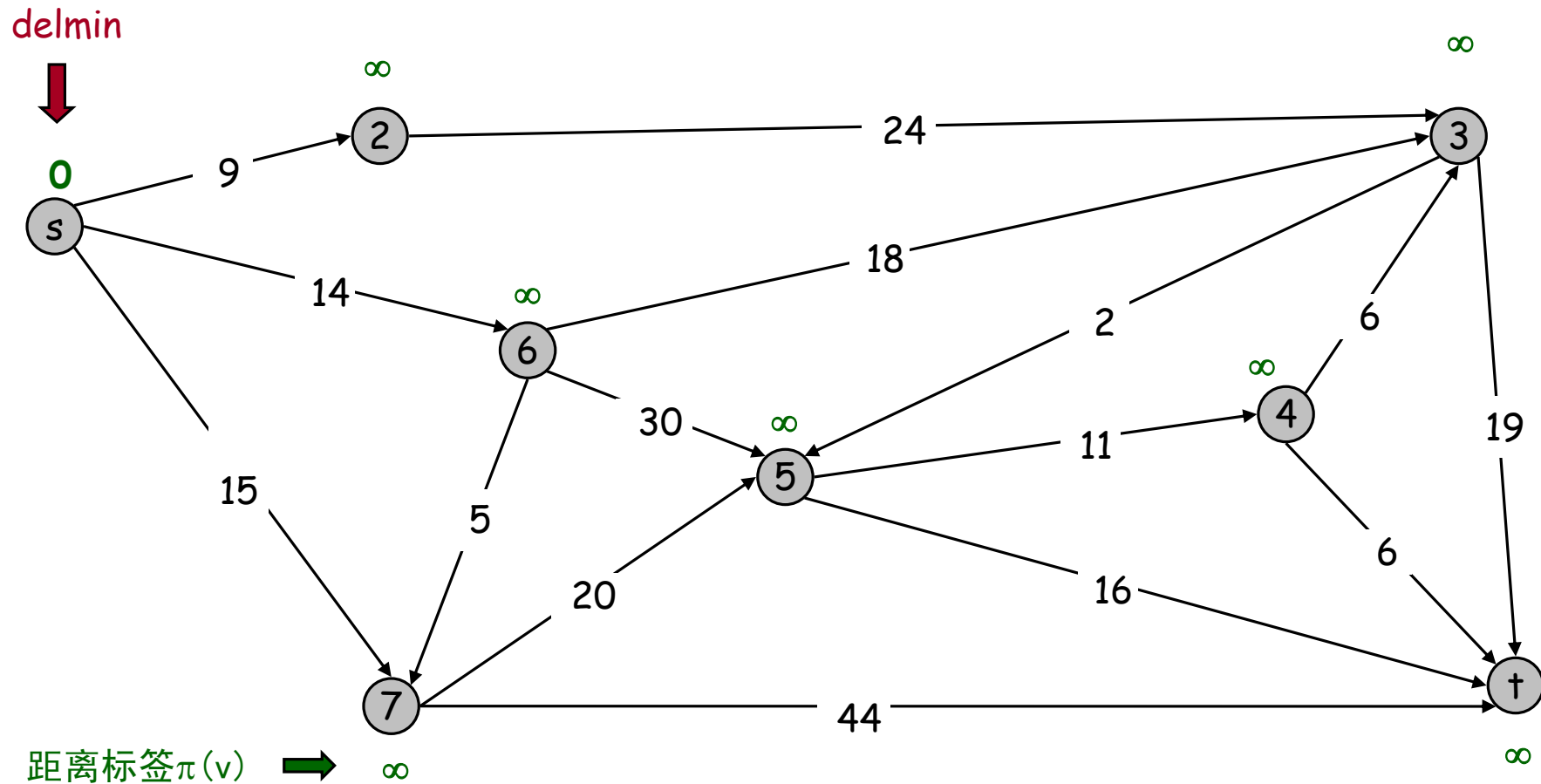


Dijkstra 算法

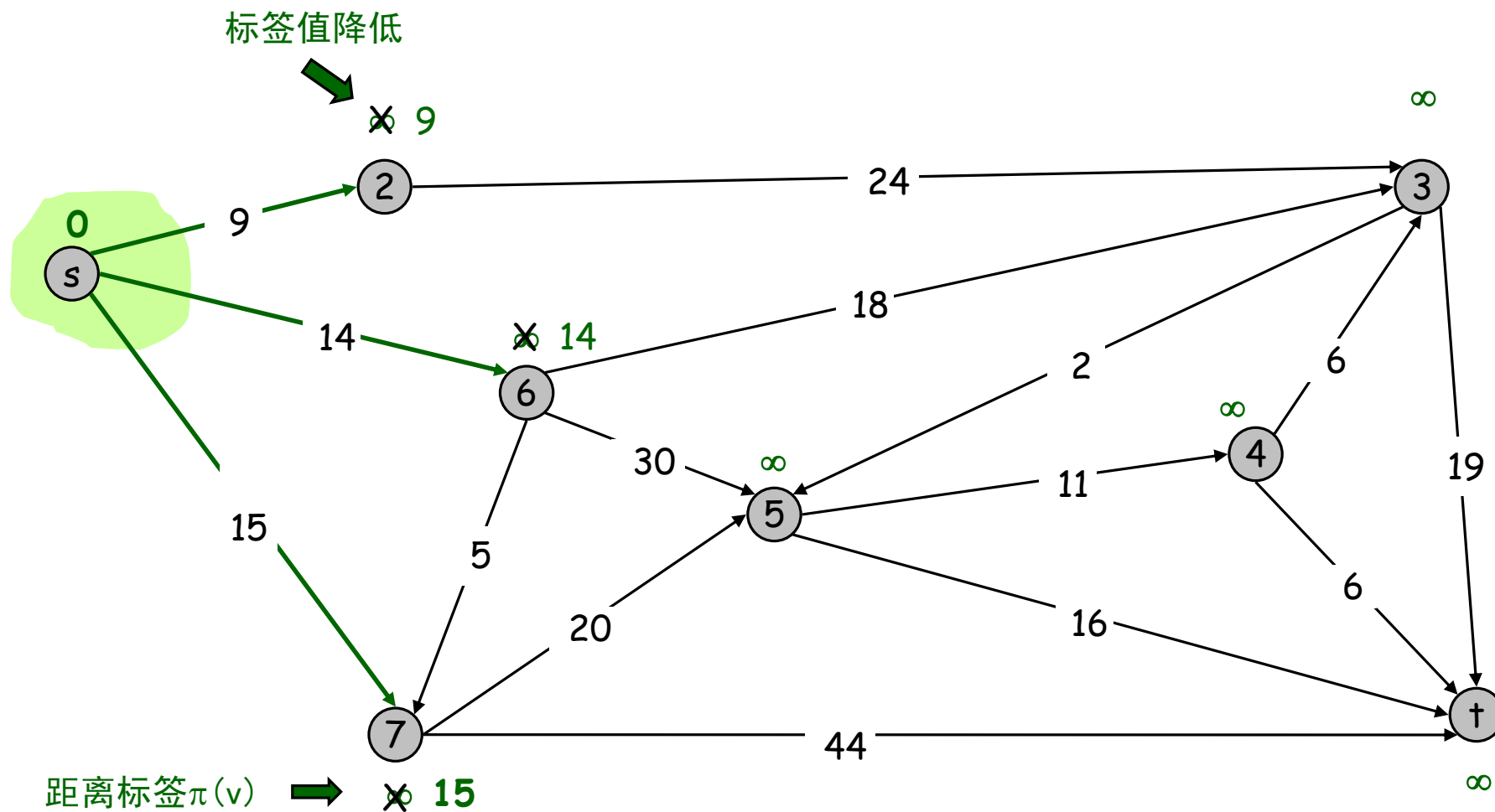


$S = \{ \}$

$PQ = \{ \textcolor{red}{s}, 2, 3, 4, 5, 6, 7, \dagger \}$



A cartoon illustration of a blue character wearing a white chef's hat. The character is holding a glass of pink drink with a straw. To the right is a large, multi-layered cake with various toppings. In front of the character is a bowl of food, possibly soup or a salad. The entire scene is enclosed in a red dotted border. The text "BLUEBUDDIE" is written in blue capital letters at the bottom right of the image.

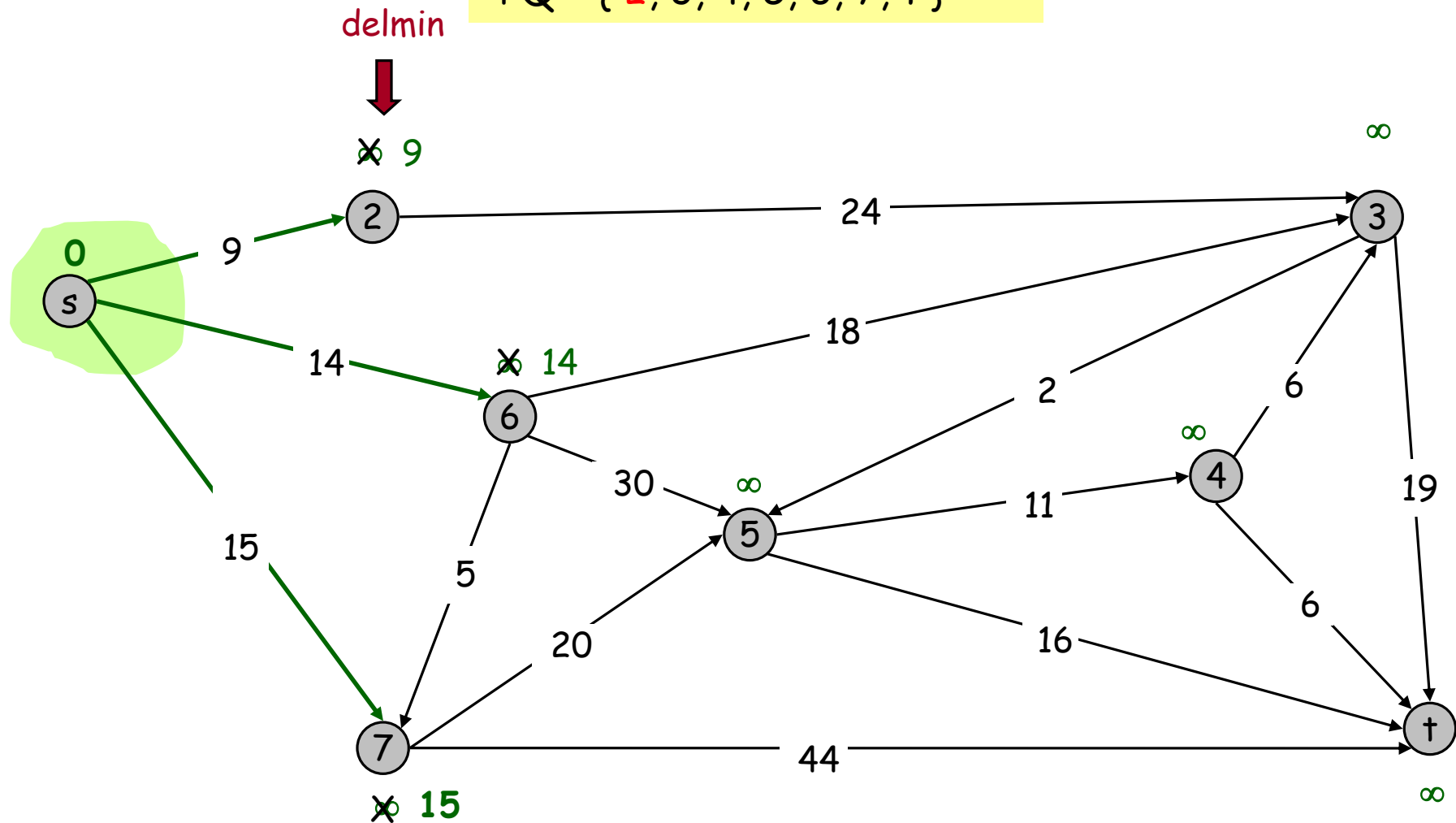
$$PQ = \{2, 3, 4, 5, 6, 7, +\}$$


Dijkstra 算法



$S = \{s\}$

$PQ = \{2, 3, 4, 5, 6, 7, \dagger\}$

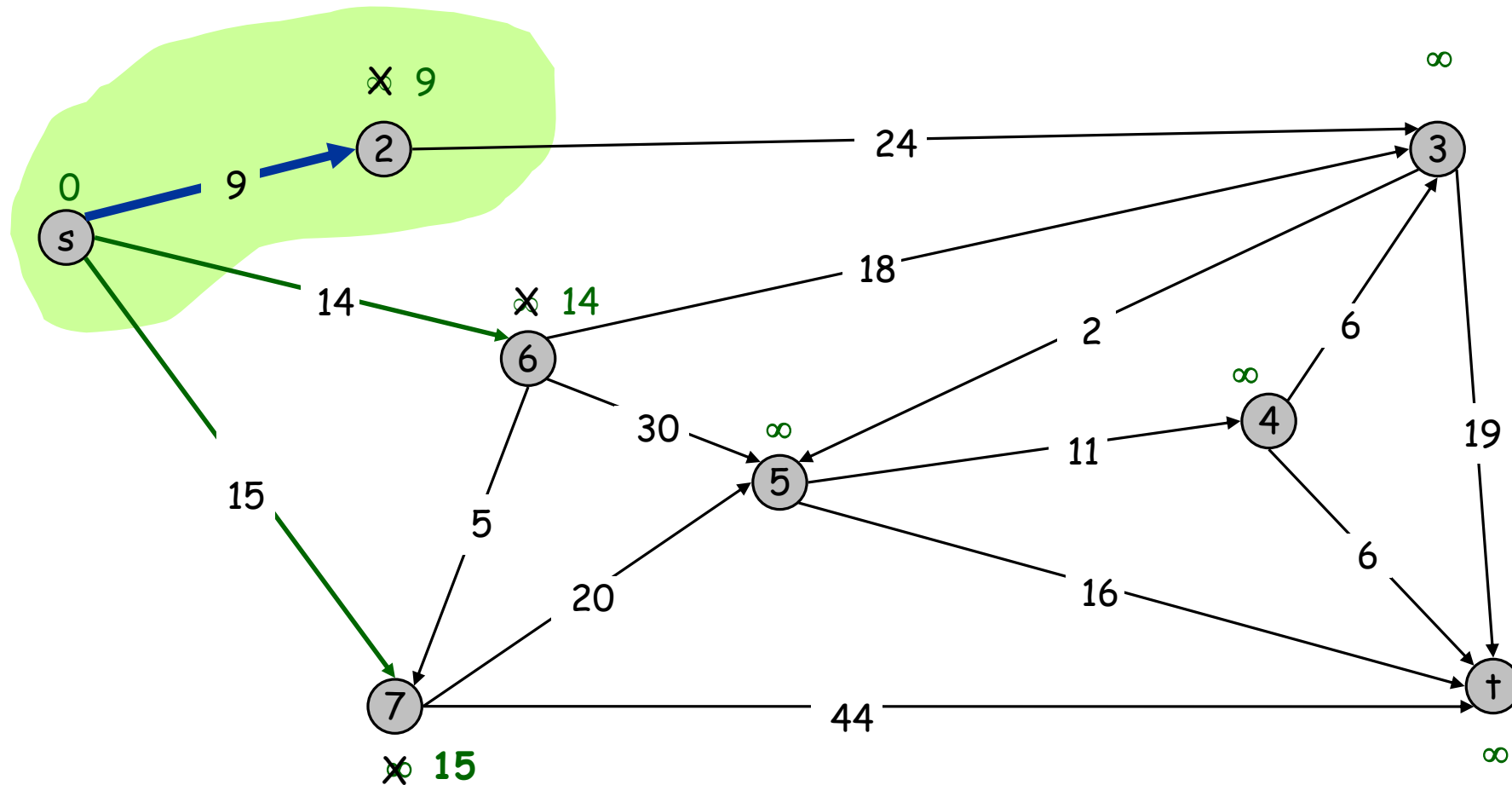


Dijkstra 算法



$S = \{s, 2\}$

$PQ = \{3, 4, 5, 6, 7, \dagger\}$

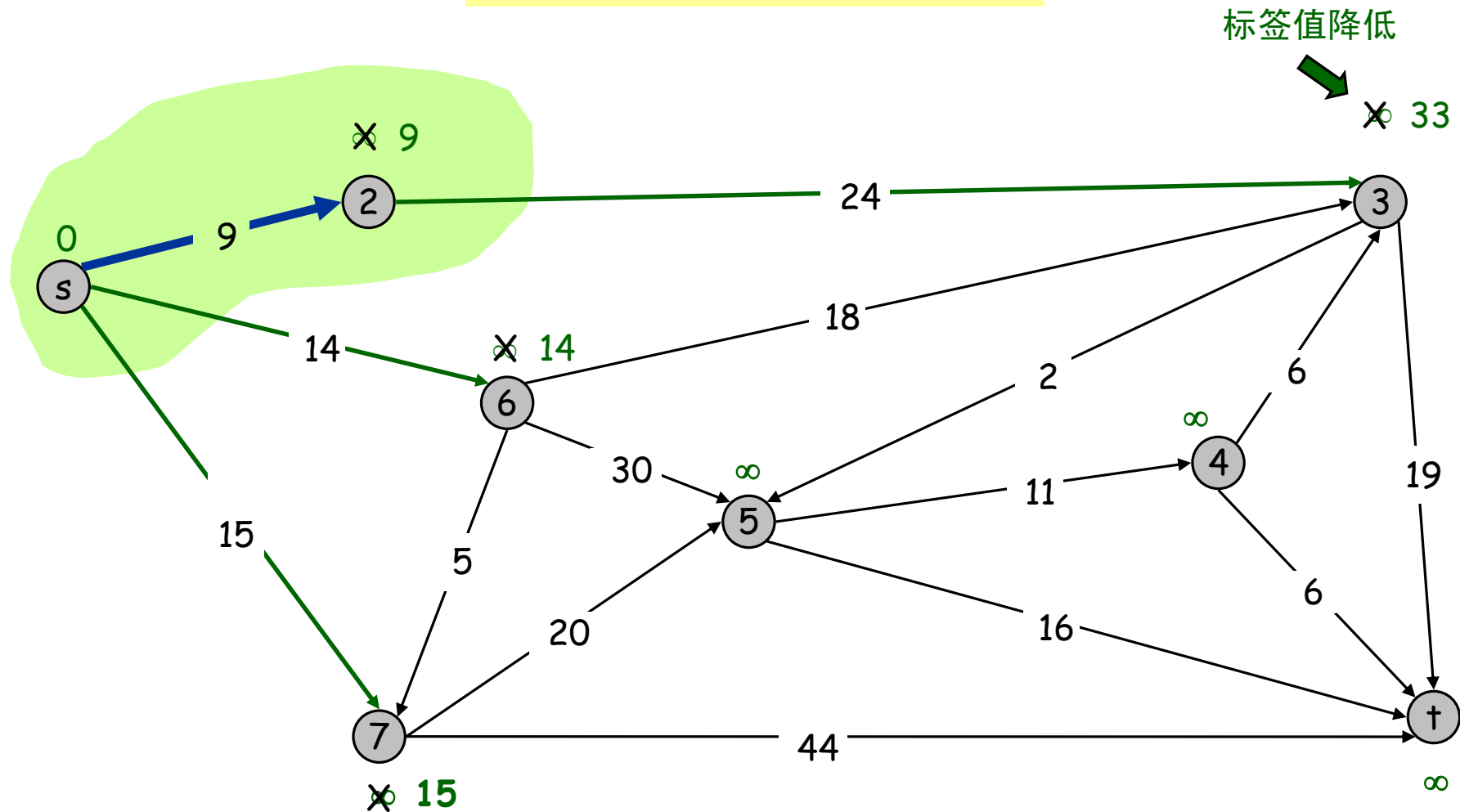


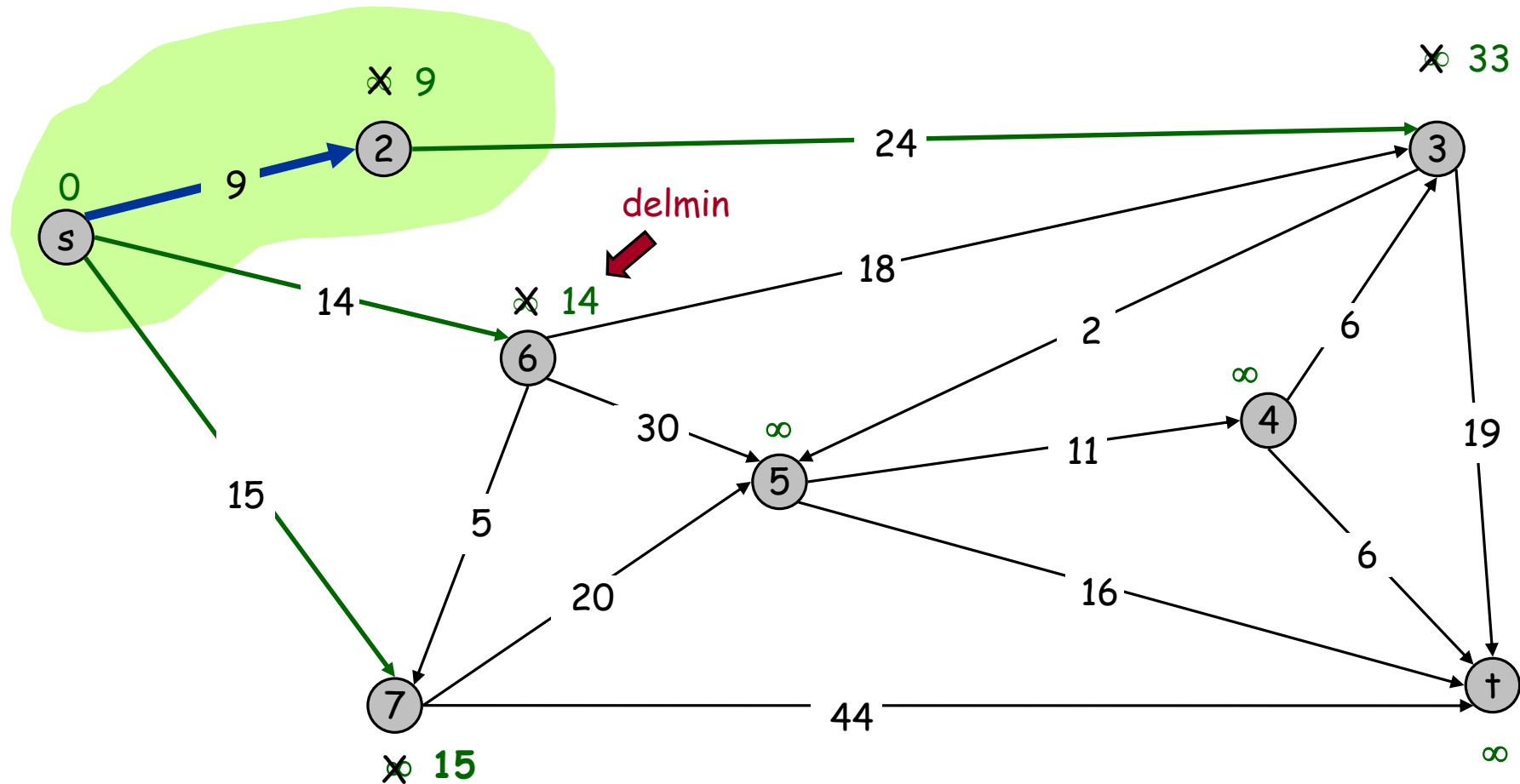
Dijkstra 算法



$S = \{s, 2\}$

$PQ = \{3, 4, 5, 6, 7, \dagger\}$



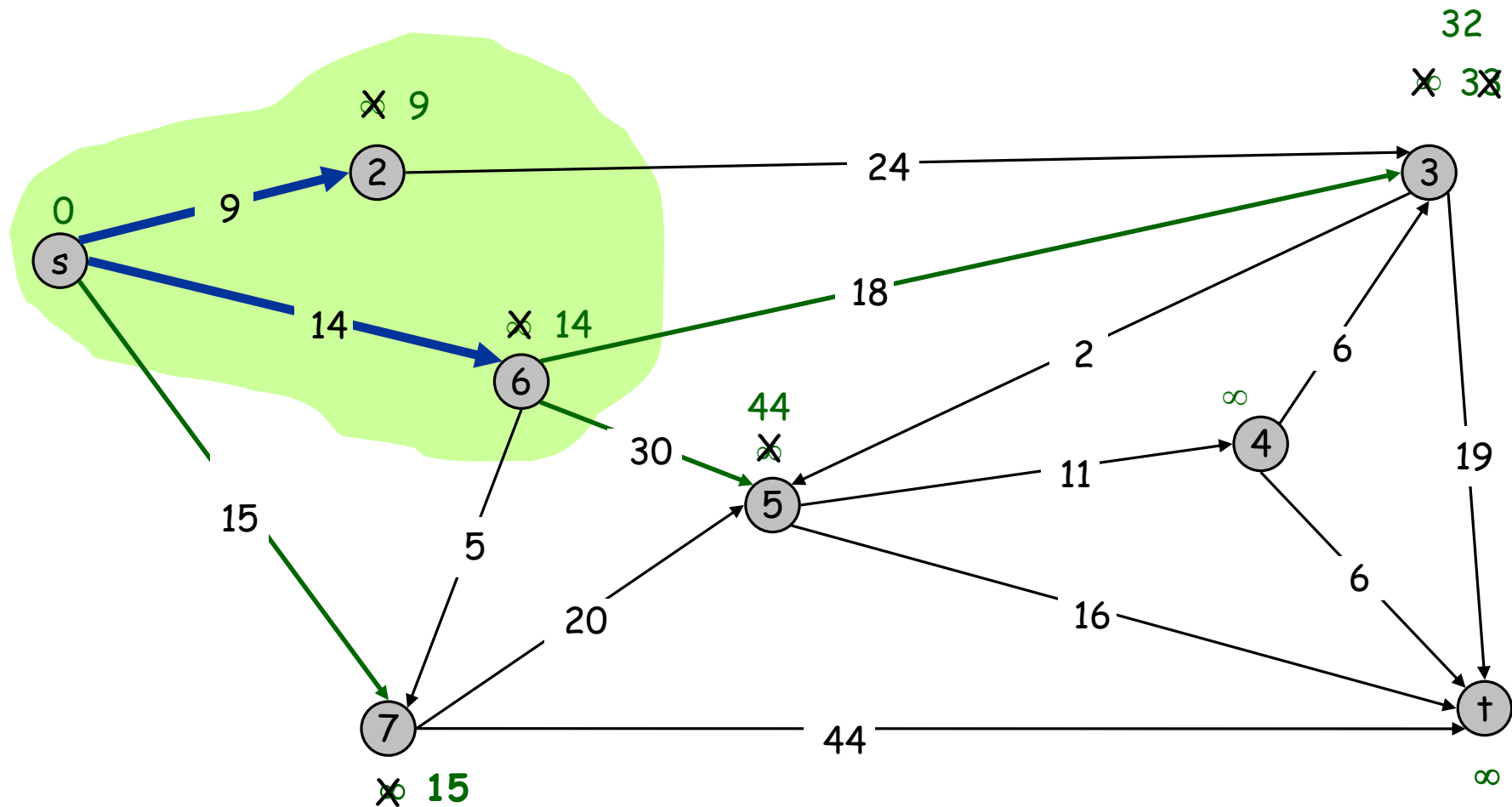
$$PQ = \{3, 4, 5, 6, 7, +\}$$


Dijkstra 算法



$S = \{s, 2, 6\}$

$PQ = \{3, 4, 5, 7, \dagger\}$

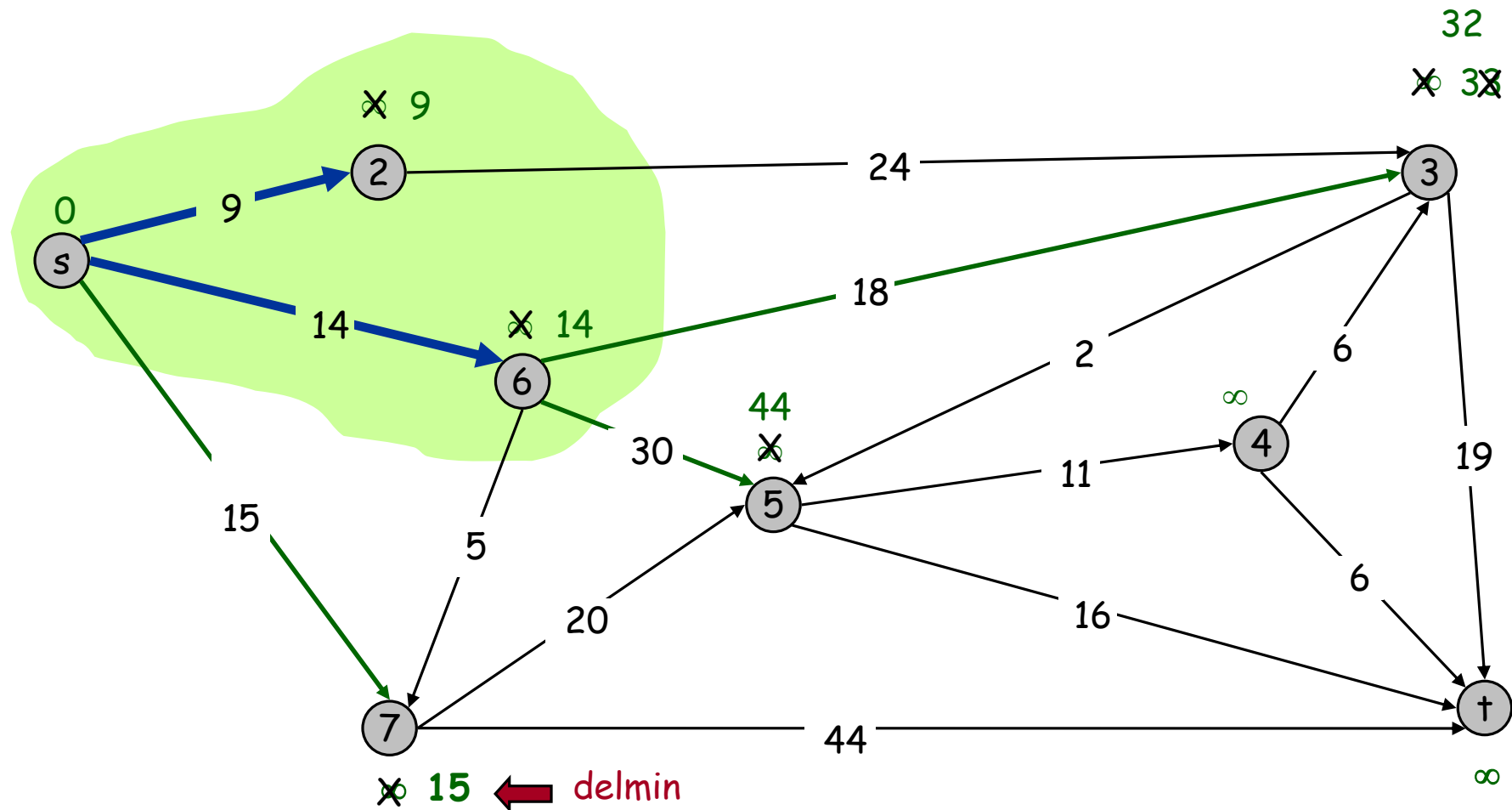


Dijkstra 算法



$S = \{s, 2, 6\}$

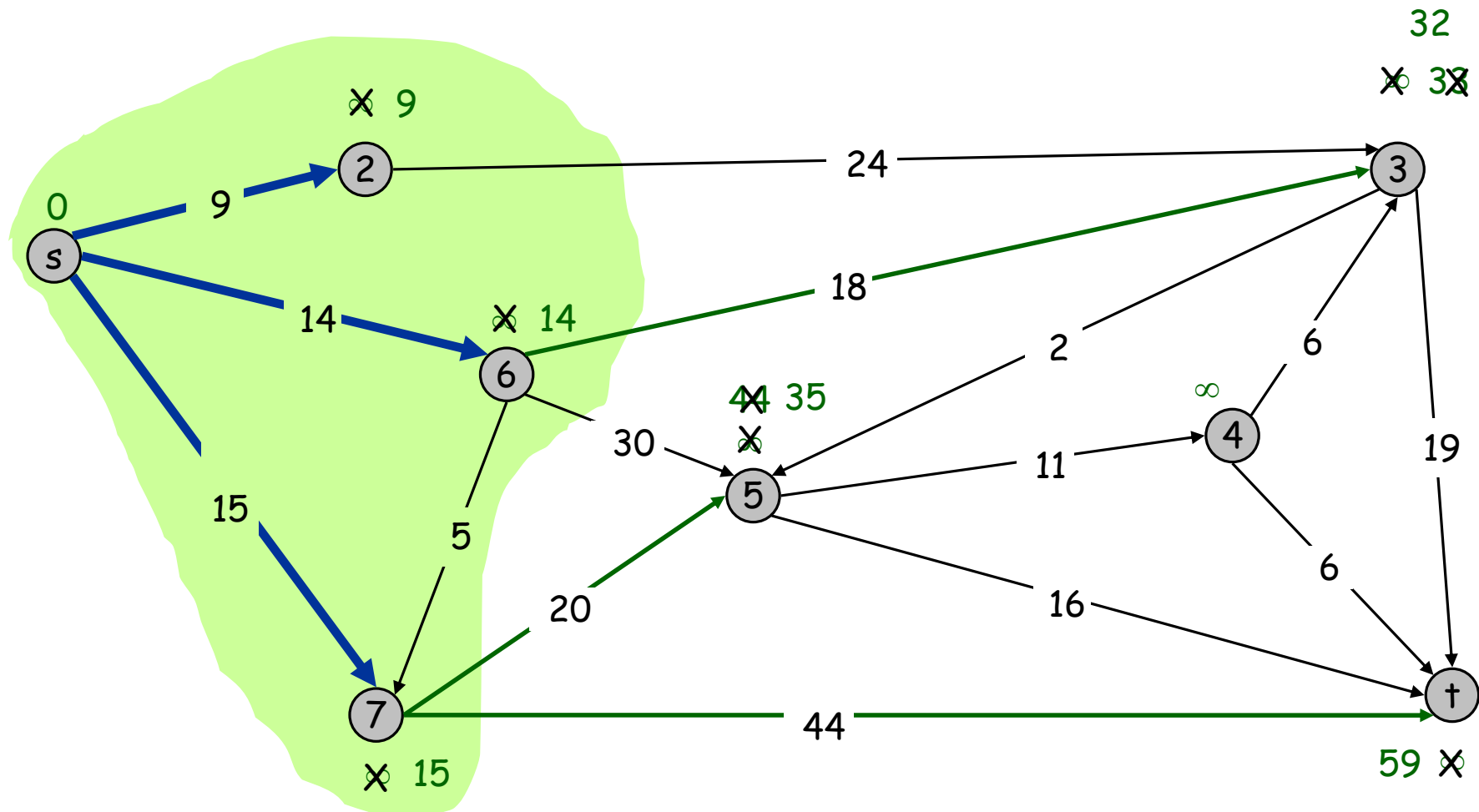
$PQ = \{3, 4, 5, 7, \dagger\}$



Dijkstra 算法



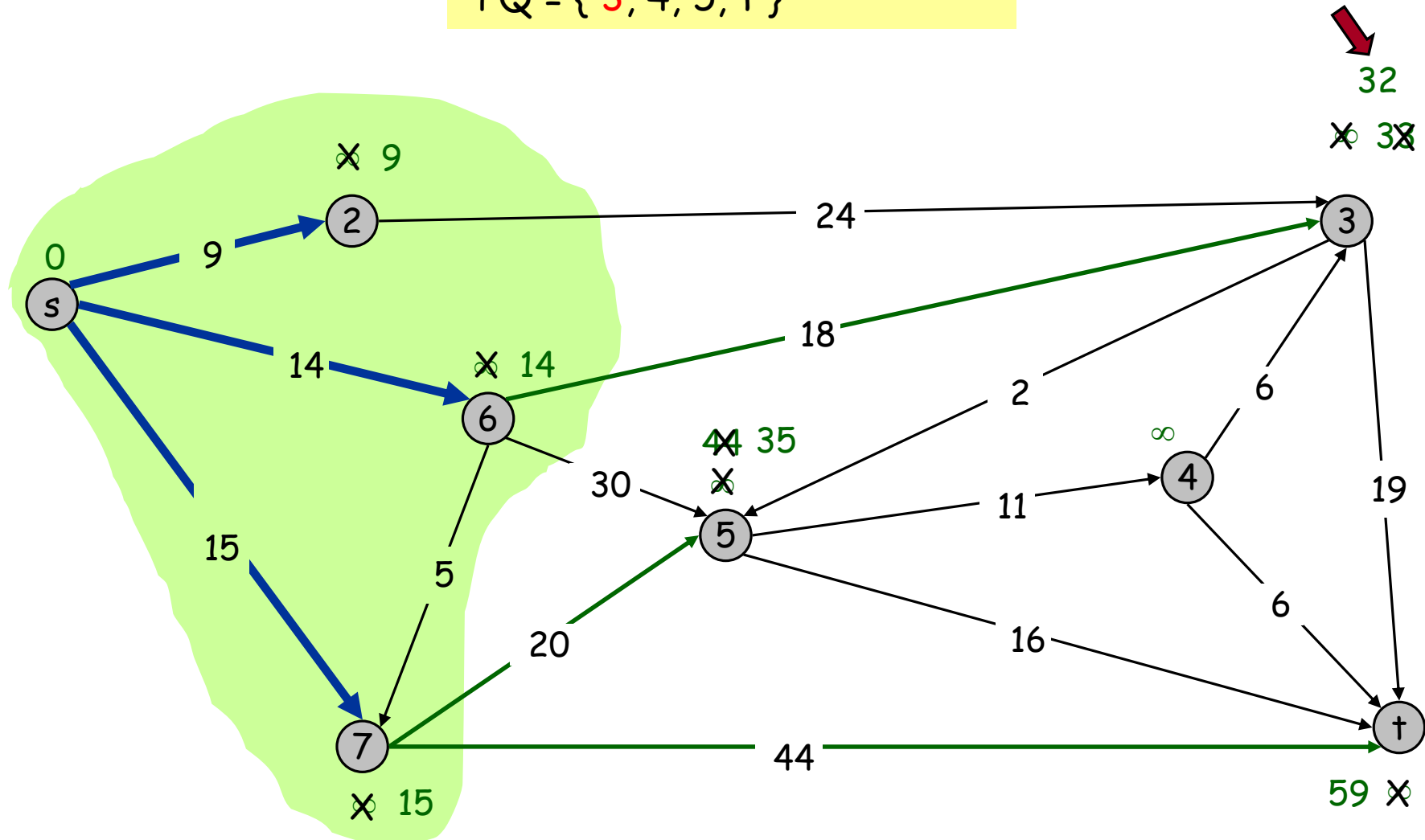
$S = \{s, 2, 6, 7\}$
 $PQ = \{3, 4, 5, \dagger\}$



Dijkstra 算法



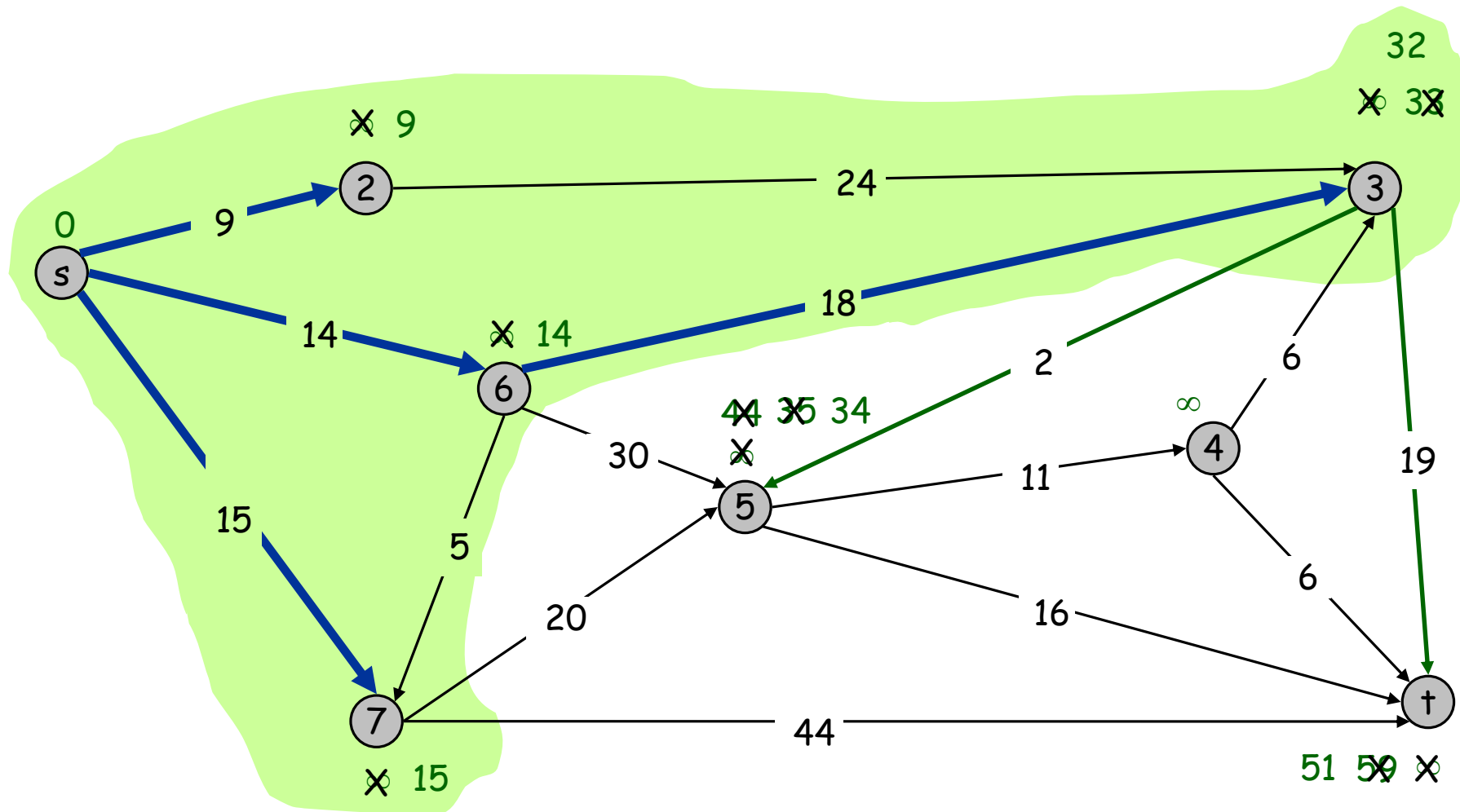
$S = \{s, 2, 6, 7\}$
 $PQ = \{ \textcolor{red}{3}, 4, 5, \dagger \}$



Dijkstra 算法



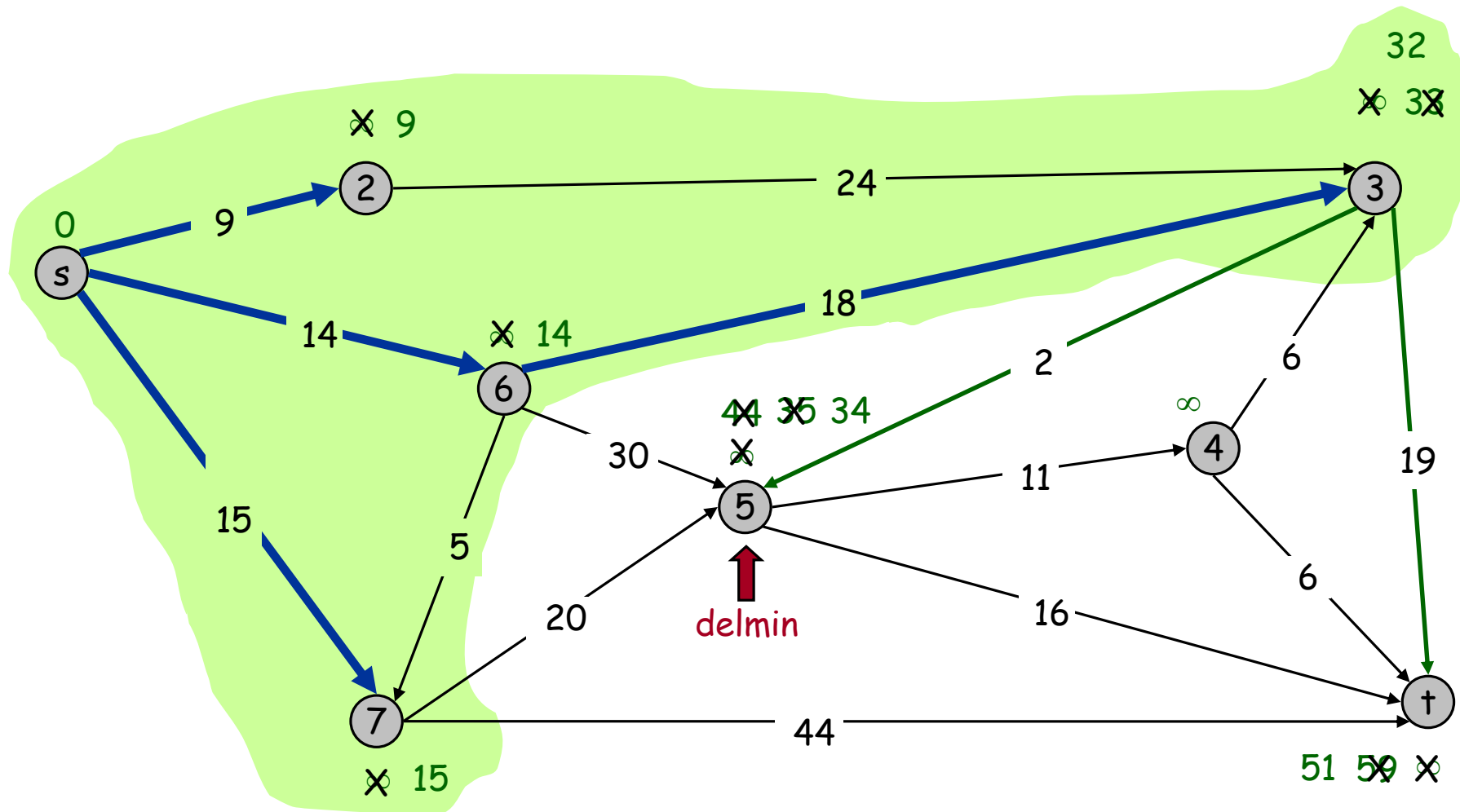
$S = \{s, 2, 3, 6, 7\}$
 $PQ = \{4, 5, \dagger\}$



Dijkstra 算法



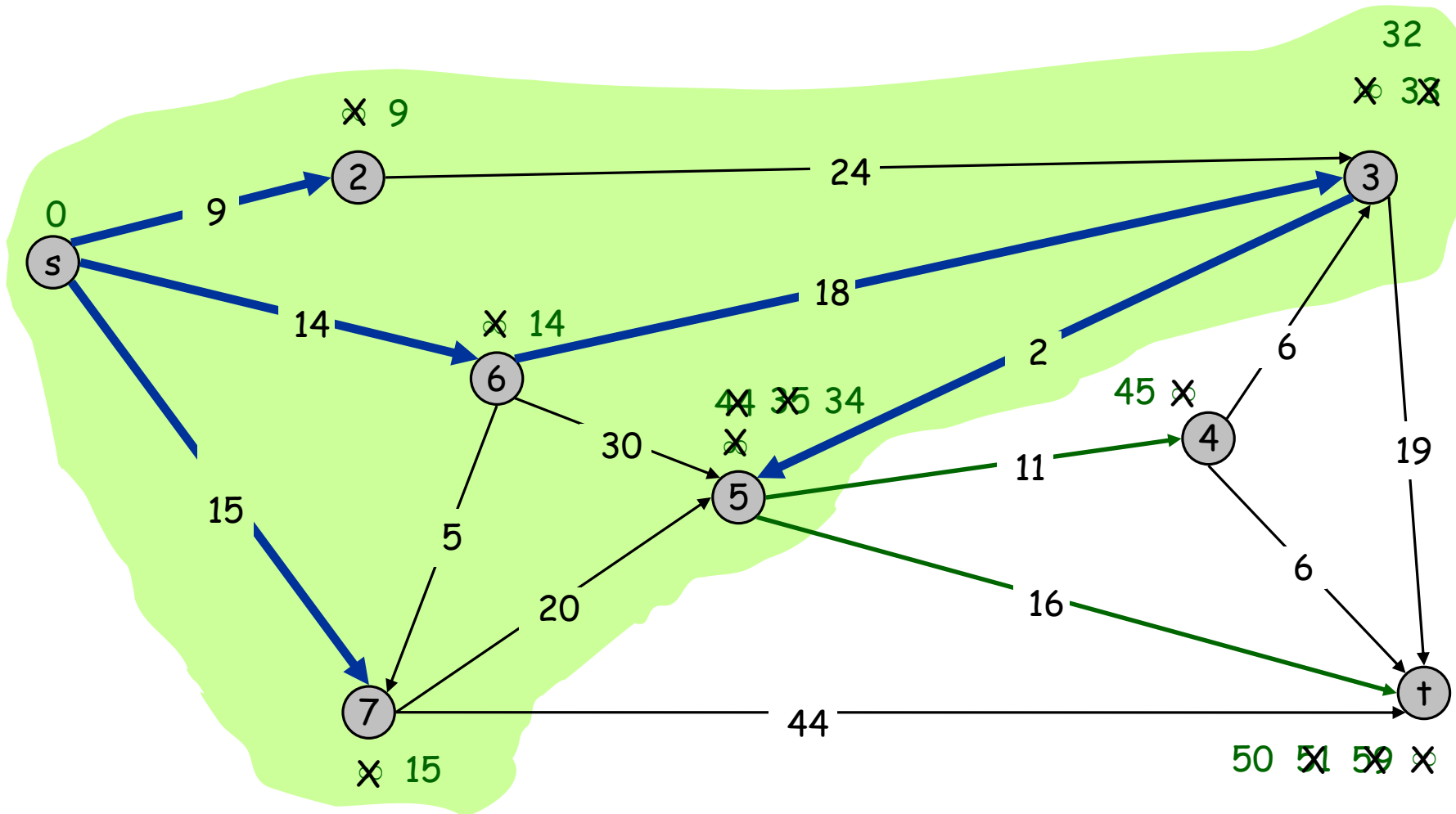
$S = \{s, 2, 3, 6, 7\}$
 $PQ = \{4, 5, \dagger\}$



Dijkstra 算法



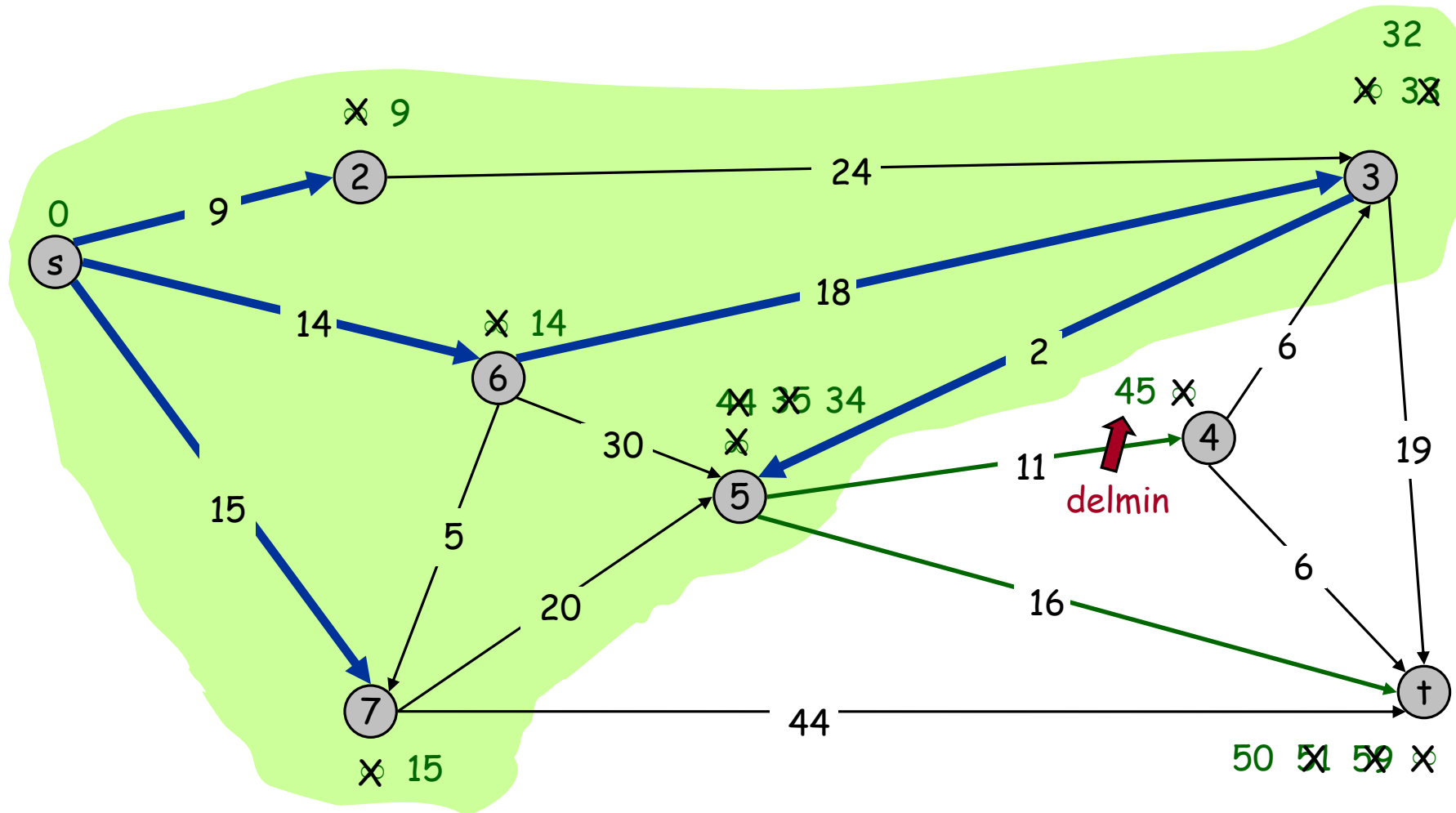
$S = \{s, 2, 3, 5, 6, 7\}$
 $PQ = \{4, \dagger\}$



Dijkstra 算法



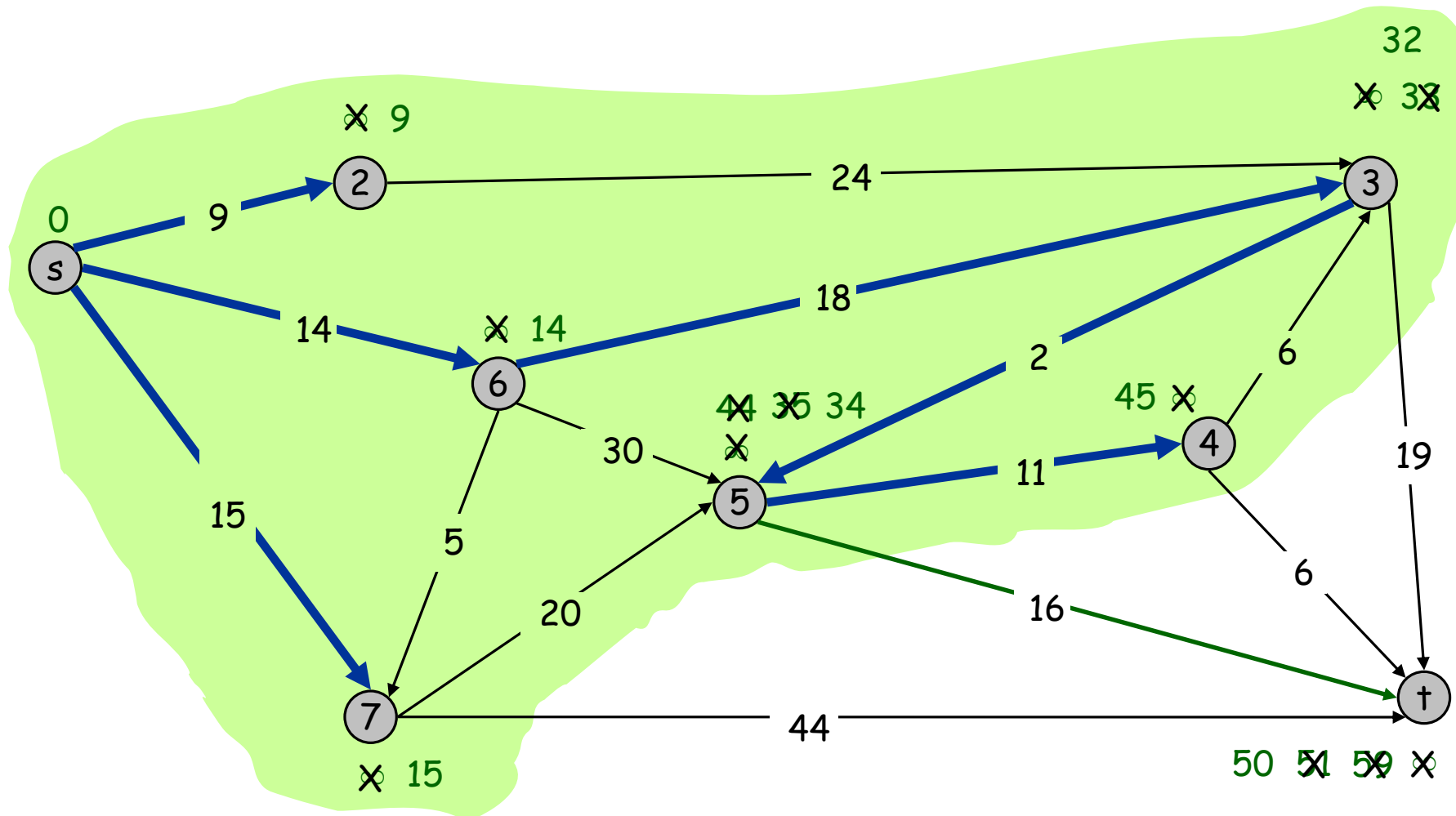
$S = \{s, 2, 3, 5, 6, 7\}$
 $PQ = \{4, \dagger\}$



Dijkstra 算法



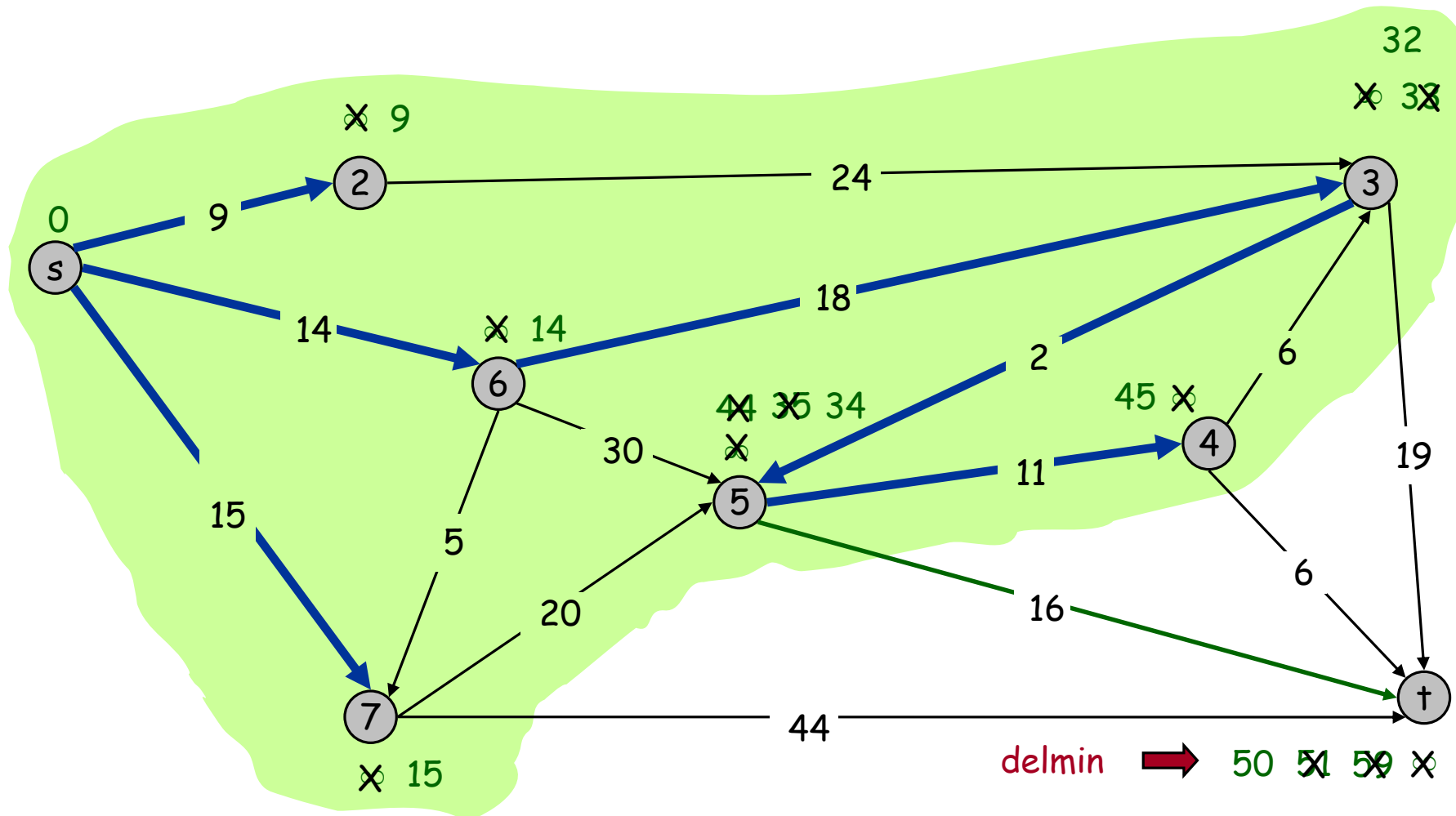
$S = \{s, 2, 3, 4, 5, 6, 7\}$
 $PQ = \{t\}$



Dijkstra 算法



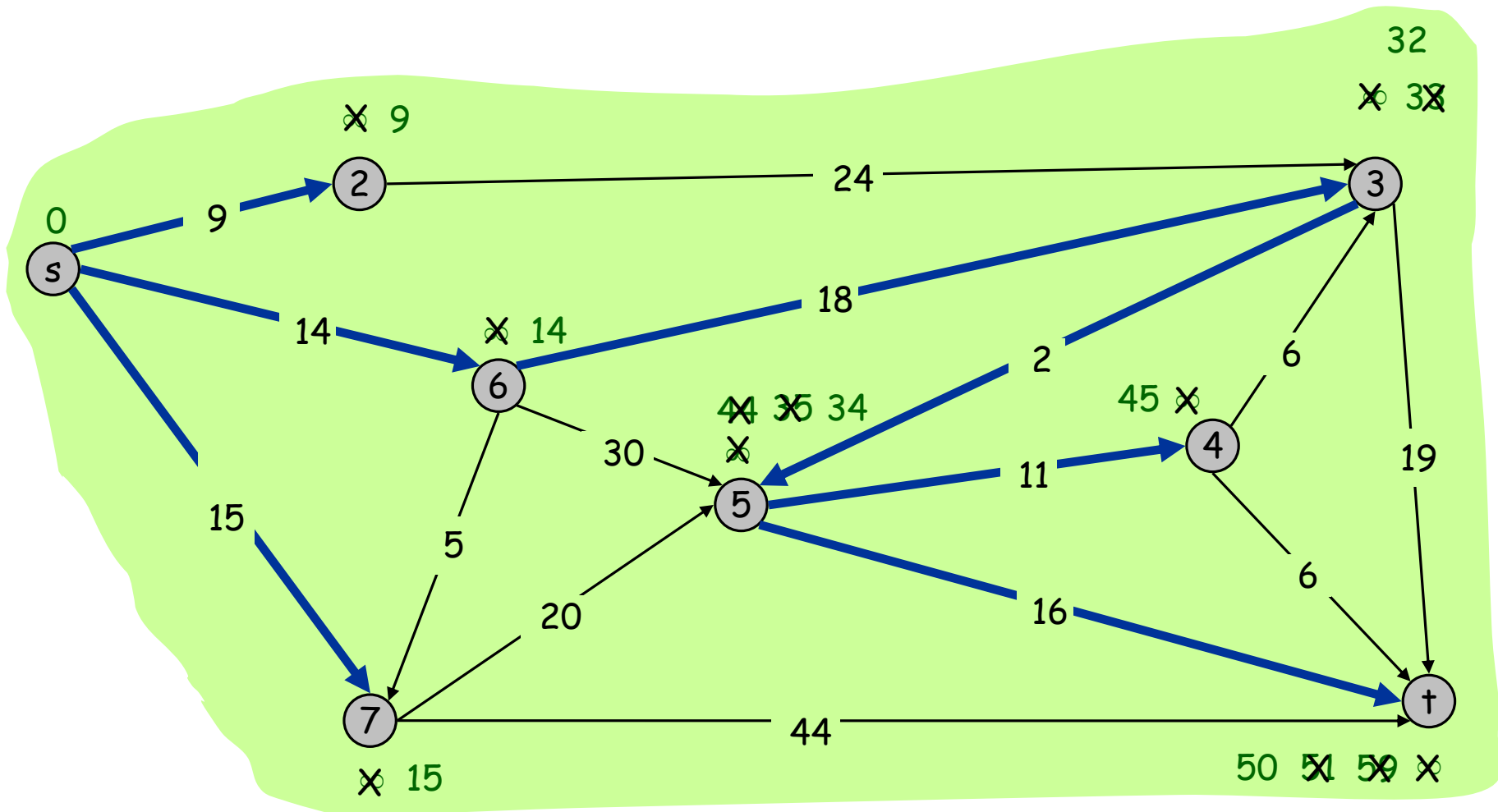
$S = \{s, 2, 3, 4, 5, 6, 7\}$
 $PQ = \{ \text{†} \}$



Dijkstra 算法



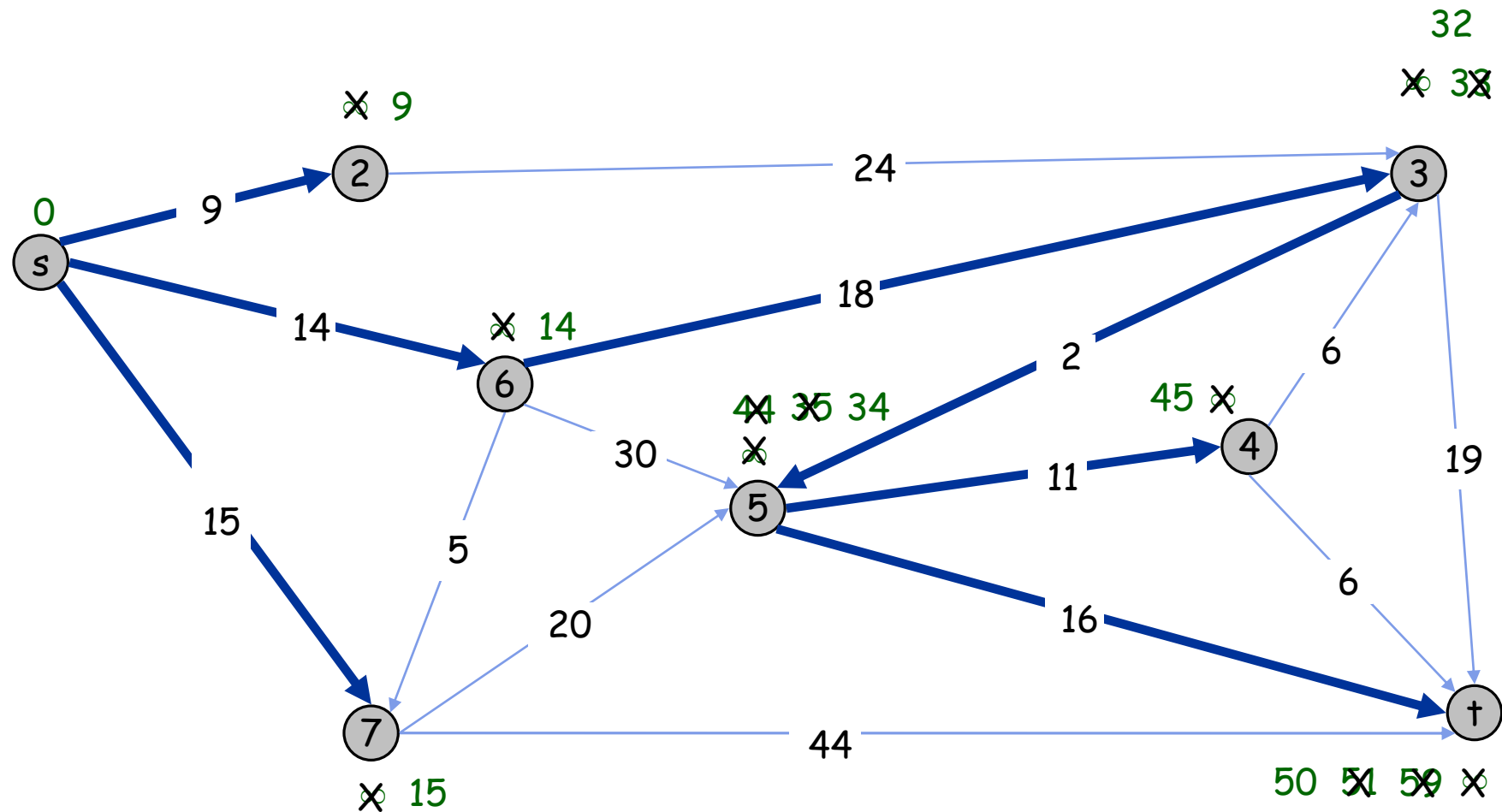
$S = \{s, 2, 3, 4, 5, 6, 7, t\}$
 $PQ = \{\}$



Dijkstra 算法



$S = \{s, 2, 3, 4, 5, 6, 7, t\}$
 $PQ = \{\}$



Dijkstra 算法：正确性证明



- 不变式：对于每个顶点 $u \in S$ ， $d(u)$ 是最短 s - u 道路的长度
- 证明：（通过对 $|S|$ 进行归纳完成）
- 基础情况： $|S| = 1$ 时该不变式显然成立
- 归纳假设：假设 $|S| = k$ ($k \geq 1$) 时该不变式成立
 - 设 v 为下一个将添加到 S 中的顶点， (u, v) 为所选边
 - 最短 s - u 道路附加边 (u, v) 是长度为 $\pi(v)$ 的 s - v 道路
 - 考虑任一 s - v 道路 P 。以下将证明它的长度不会小于 $\pi(v)$
 - 设 (x, y) 是 P 中首次离开 S 的边， P' 是到 x 为止的子道路
 - 事实上， P 离开 S 时就已经“足够长”了

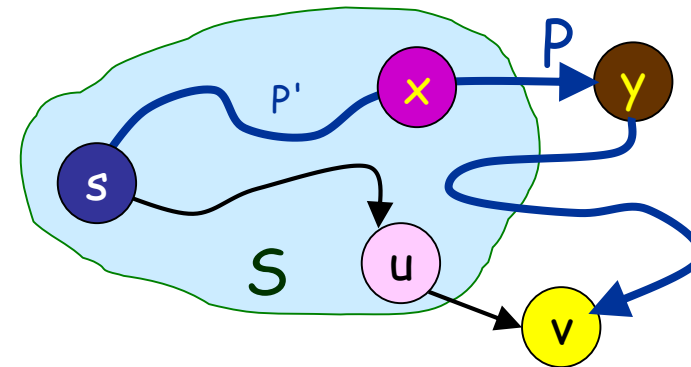
$$\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v)$$

↑
边的权值非负

↑
归纳假设

↑
 $\pi(y)$ 的定义

↑
Dijkstra 算法选取了 v 而不是 y



Dijkstra 算法：记录最短道路

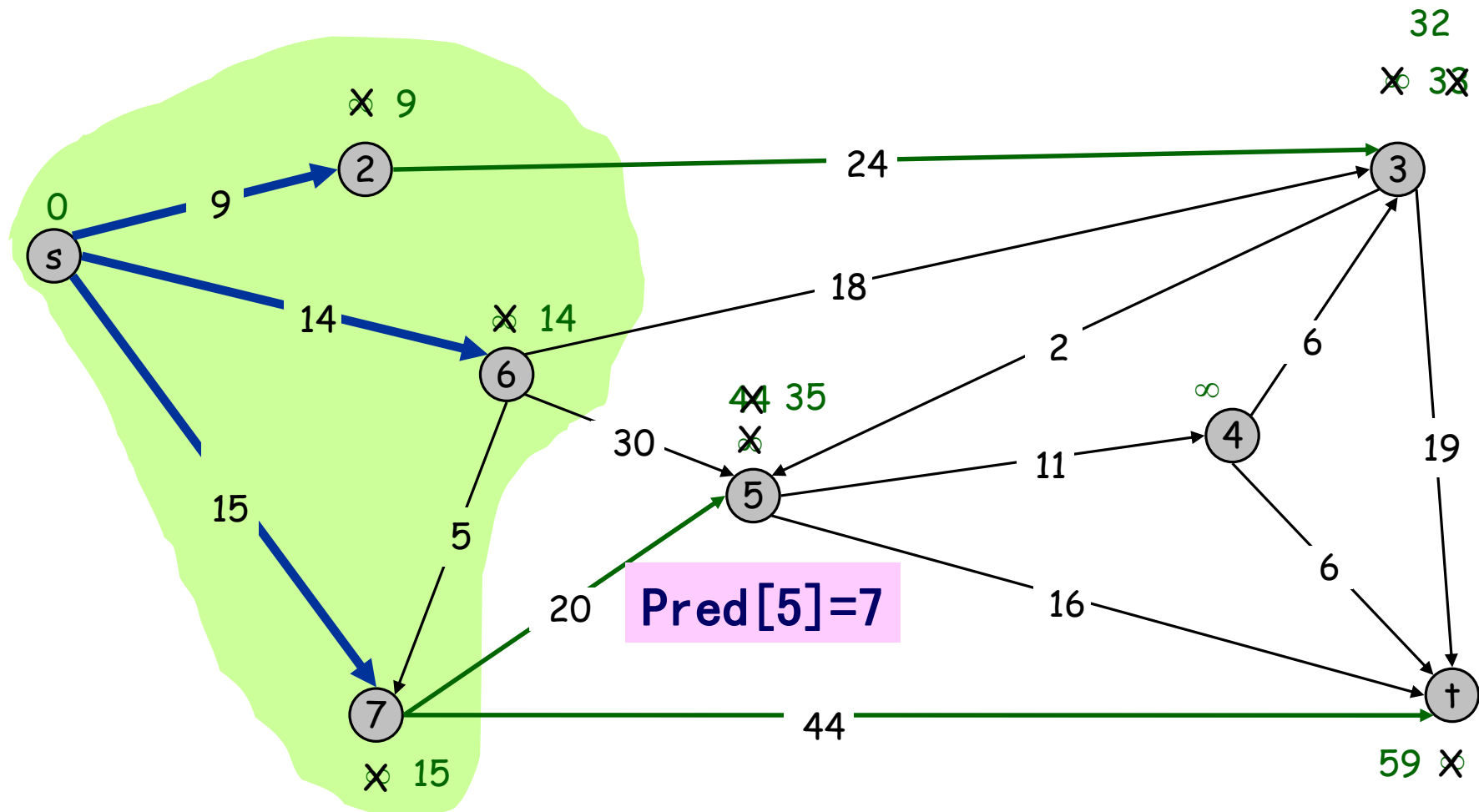


- 为每个顶点 $v \in V$ 维护其前趋 $pred[v]$ ，其是在当前到 v 的最短道路中在 v 之前的顶点
- 使用 $pred$ 值，回溯给出最短道路

Dijkstra 算法：记录最短道路



$S = \{s, 2, 6, 7\}$
 $PQ = \{3, 4, 5, \dagger\}$



Dijkstra 算法



1. **for each** $v \in V$ **do**
2. $d[v] \leftarrow \infty$
3. $pred[v] \leftarrow \text{NIL}$
4. $d[s] \leftarrow 0$
5. $S \leftarrow \emptyset$
6. $H \leftarrow V$ // H 是关于 $V - S$ 的一个关于cost的优先级队列
7. **while** $H \neq \emptyset$ **do**
8. $u \leftarrow \text{EXTRACT-MIN}(H)$
9. $S \leftarrow S \cup \{u\}$
10. **for each** $z \in adj(u) - S$ **do**
11. **if** $d[z] > d[u] + w(u, z)$ **then**
12. $d[z] \leftarrow d[u] + w(u, z)$
13. $pred[z] \leftarrow u$

Dijkstra 算法的分析



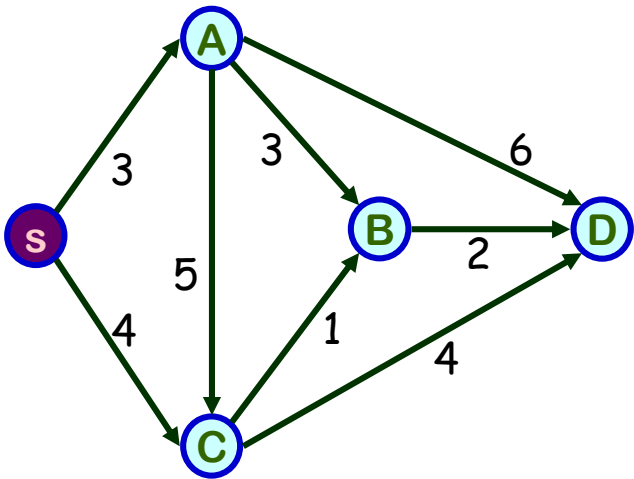
- 复杂度

- 使用无序数组存储优先级队列: $O(n^2)$
- 以邻接列表方式存储图并使用堆作为优先级队列
 - 对于二叉堆, 该算法需要 $O((m+n) \log n)$ 时间 (也是 $O(m \log n)$ 的)

Dijkstra 算法



S	s	A	B	C	D
{}	0/N	∞ /N	∞ /N	∞ /N	∞ /N
{s}		3/s	∞ /N	4/s	∞ /N
{s,A}			6/A	4/s	9/A
{s,A,C}			5/C		8/C
{s,A,C,B}					7/B
{s,A,C,B,D}					





Prim 算法与 Dijkstra 算法



```
1.  for each  $v \in V$  do
2.       $d[v] \leftarrow \infty$ 
3.       $pred[v] \leftarrow \text{NIL}$ 
4.   $d[s] \leftarrow 0$ 
5.   $S \leftarrow \emptyset$ 
6.   $H \leftarrow V$  //  $H$  是关于  $V - S$  的一个优先级队列
7.  while  $H \neq \emptyset$  do
8.       $u \leftarrow \text{EXTRACT-MIN}(H)$ 
9.       $S \leftarrow S \cup \{u\}$ 
10.   for each  $z \in adj(u) - S$  do
11.       if  $d[z] > d[u] + w(u, z)$  then
12.            $d[z] \leftarrow d[u] + w(u, z)$ 
13.            $pred[z] \leftarrow u$ 
```

```
if  $d[z] > w(u, z)$  then
     $d[z] \leftarrow w(u, z)$ 
     $pred[z] \leftarrow u$ 
```

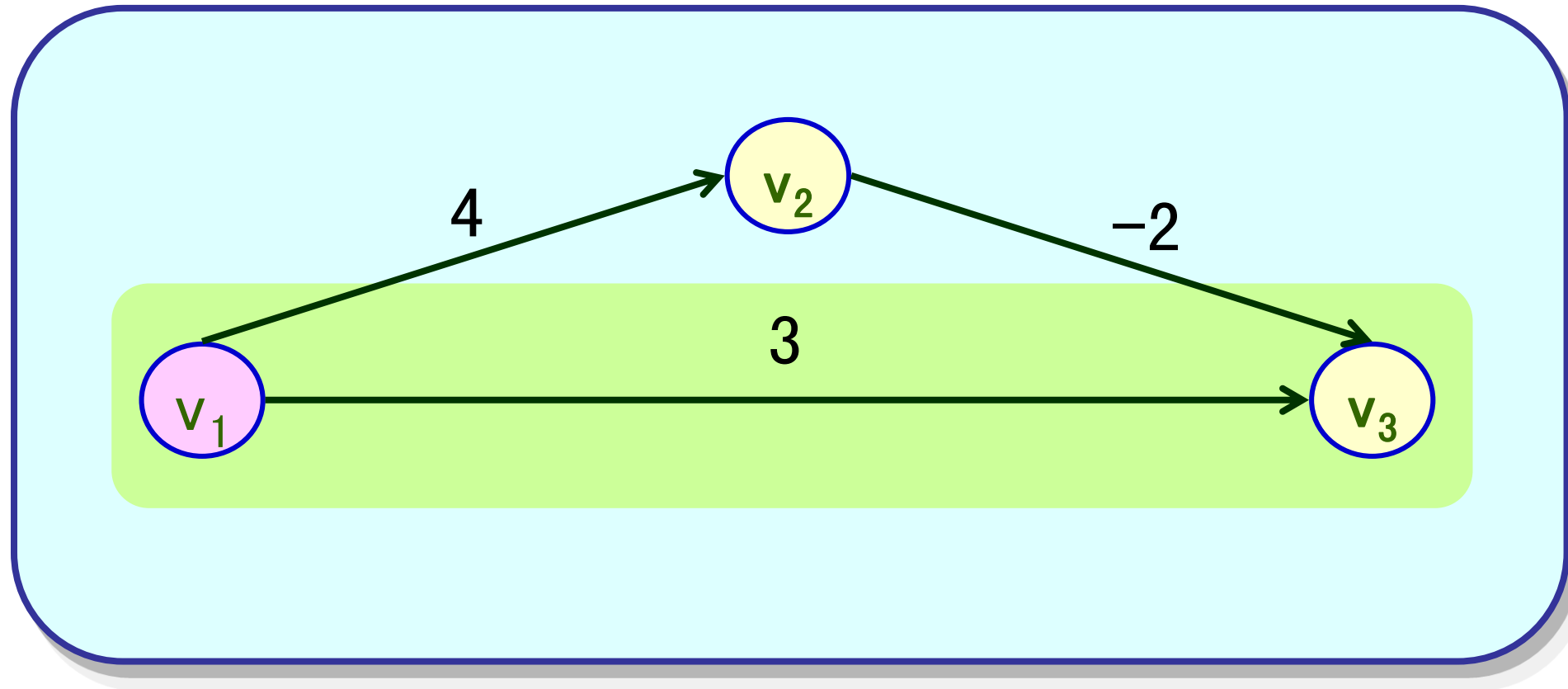
单源最短道路



- 注记:

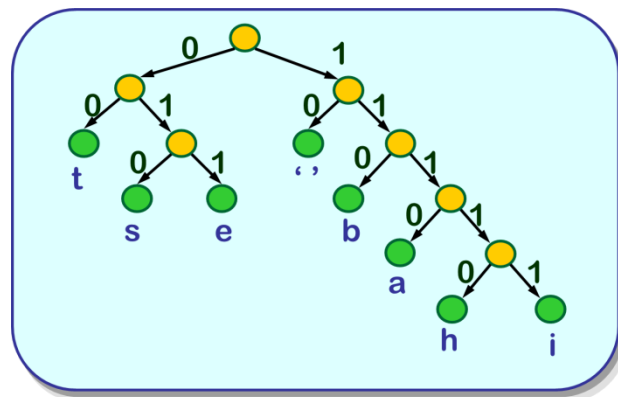
- 该算法解决的是已知起始点求最短道路的问题；而已知终点求最短道路的问题与之完全相同
- 该算法只适合于边权值都是**非负值**的情况

单源最短道路



Huffman算法

Huffman's algorithm



前缀码



- 使用“0”和“1”对“bat is the best”进行二进制编码
- 各个字符的出现次数如下表所示：

字符	space	t	b	e	s	a	h	i
出现次数	4	3	2	2	2	1	1	1

前缀码



- 对 “bat is the best” 进行编码
- 各个字符的出现次数及码字（Code1）如下表所示：

字符	space	t	b	e	s	a	h	i
出现次数	4	3	2	2	2	1	1	1
编码	000	001	010	011	100	101	110	111
48 比特	010 101 001 000 111 100 000 001 110 011 000 010 011 100 001 000							

前缀码



- 对 “bat is the best” 进行编码
- 各个字符的出现次数及码字（Code1）如下表所示：

字符	space	t	b	e	s	a	h	i
出现次数	4	3	2	2	2	1	1	1
编码	000	001	010	011	100	101	110	111
48 比特	4×3	3×3	2×3	2×3	2×3	1×3	1×3	1×3

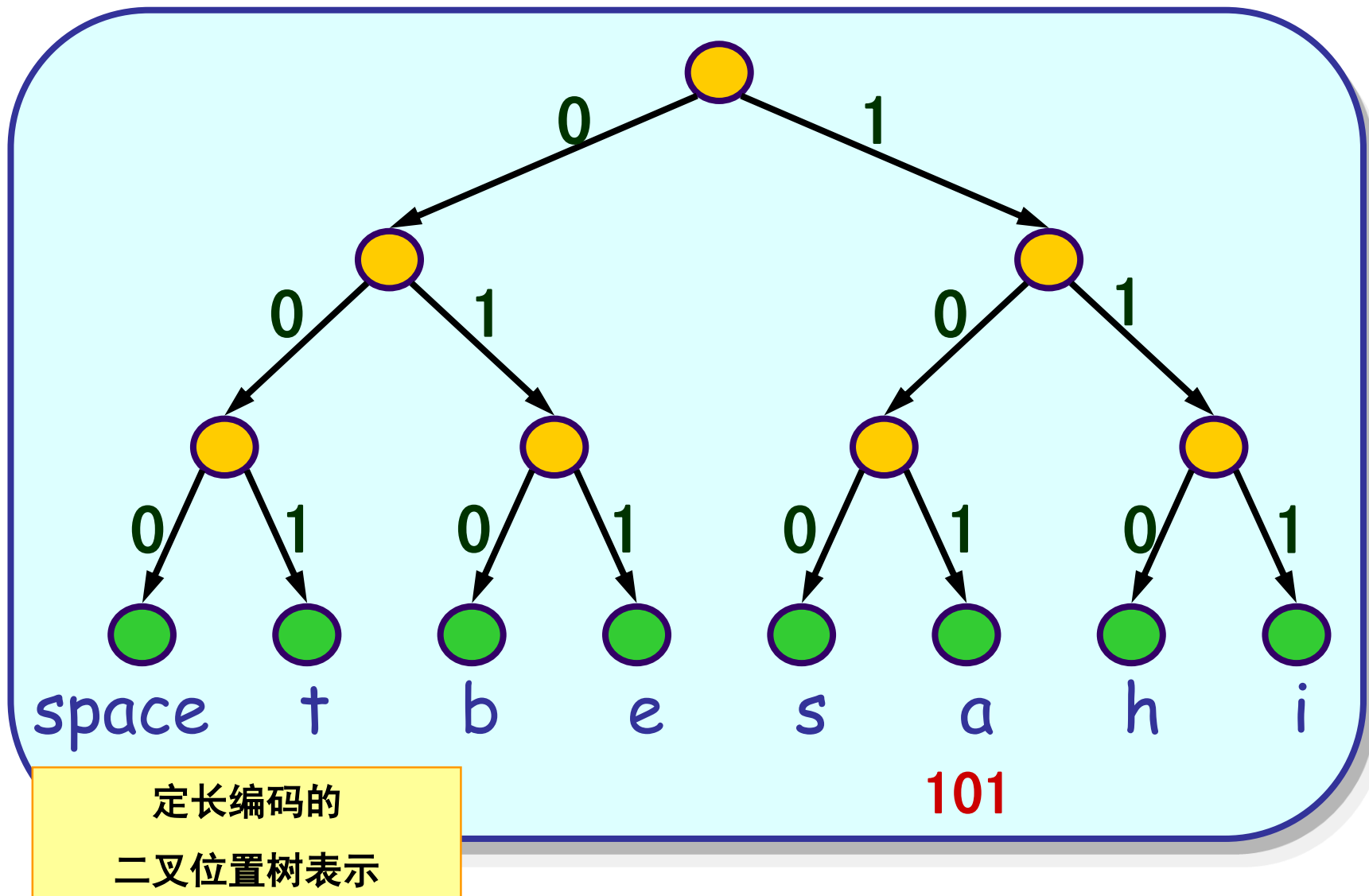
前缀码



- 对 “bat is the best” 进行编码
- 各个字符的出现次数及码字（Code1）如下表所示：

字符	space	t	b	e	s	a	h	i
出现次数	4	3	2	2	2	1	1	1
编码	000	001	010	011	100	101	110	111
解码	1 1 0 1 0 1 0 1 0 1 1 1 0 0 1 1 0 0							
	h a b i t s							

前缀码



前缀码



- 对 “bat is the best” 进行编码
- 各个字符的出现次数及码字（Code2）如下表所示：

字符	space	t	b	e	s	a	h	i
出现次数	4	3	2	2	2	1	1	1
编码	10	00	110	011	010	1110	11110	11111
46 比特	110 1110 00 10 11111 010 10 00 11110 011 10 110 011 010 00 10							

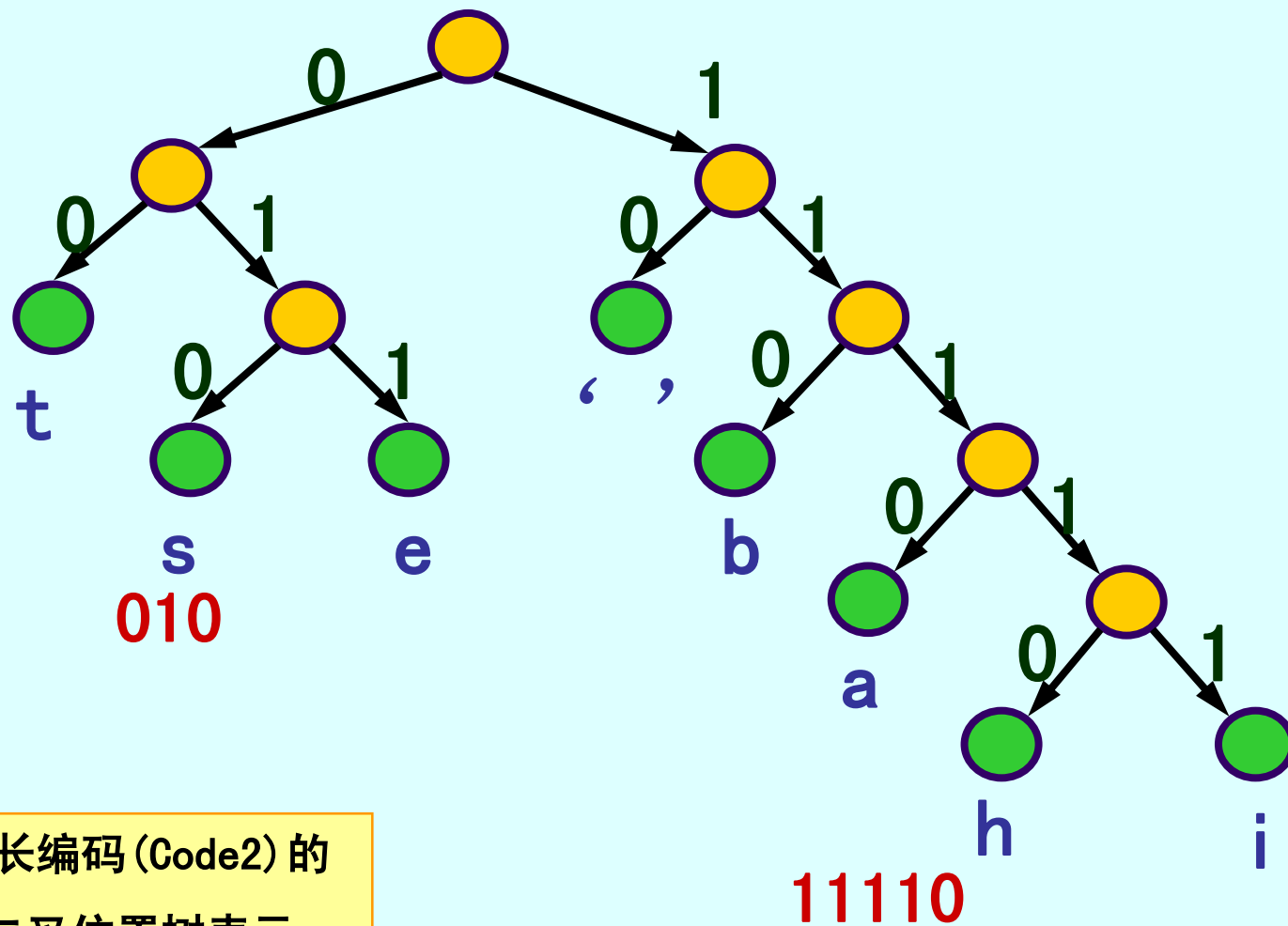
前缀码



- 对 “bat is the best” 进行编码
- 各个字符的出现次数及码字（Code2）如下表所示：

字符	space	t	b	e	s	a	h	i
出现次数	4	3	2	2	2	1	1	1
编码	10	00	110	011	010	1110	11110	11111
46 比特	4×2	3×2	2×3	2×3	2×3	1×4	1×5	1×5

前缀码



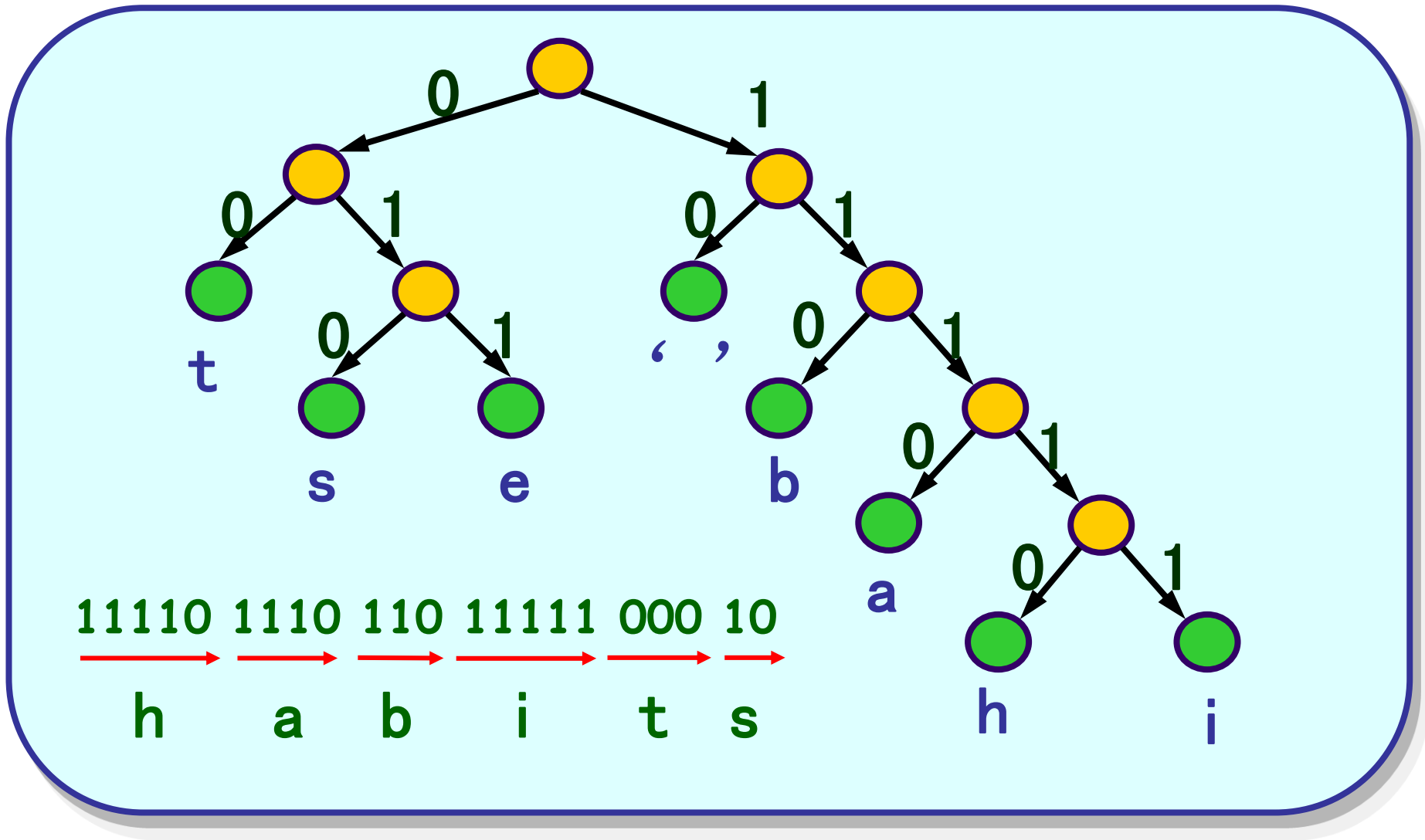
前缀码



- 对 “bat is the best” 进行编码
- 各个字符的出现次数及码字（Code2）如下表所示：

字符	space	t	b	e	s	a	h	i
出现次数	4	3	2	2	2	1	1	1
编码	10	00	110	011	010	1110	11110	11111
解码	1 1 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 0 0 1 0							
	h a b i t s							

前缀码



前缀码



- 对 “bat is the best” 进行编码
- 各个字符的出现次数及码字（Code3）如下表所示：

字符	space	t	b	e	s	a	h	i
出现次数	4	3	2	2	2	1	1	1
编码	00	01	010	001	11	110	111	1
37 比特	010 110 01 00 1 11 00 01 111 001 00 010 001 11 01 00							

前缀码



- 对 “bat is the best” 进行编码
- 各个字符的出现次数及码字（Code3）如下表所示：

字符	space	t	b	e	s	a	h	i
出现次数	4	3	2	2	2	1	1	1
编码	00	01	010	001	11	110	111	1
37 比特	4×2	3×2	2×3	2×3	2×2	1×3	1×3	1×1



前缀码



- 对 “bat is the best” 进行编码
- 各个字符的出现次数及码字（Code3）如下表所示：

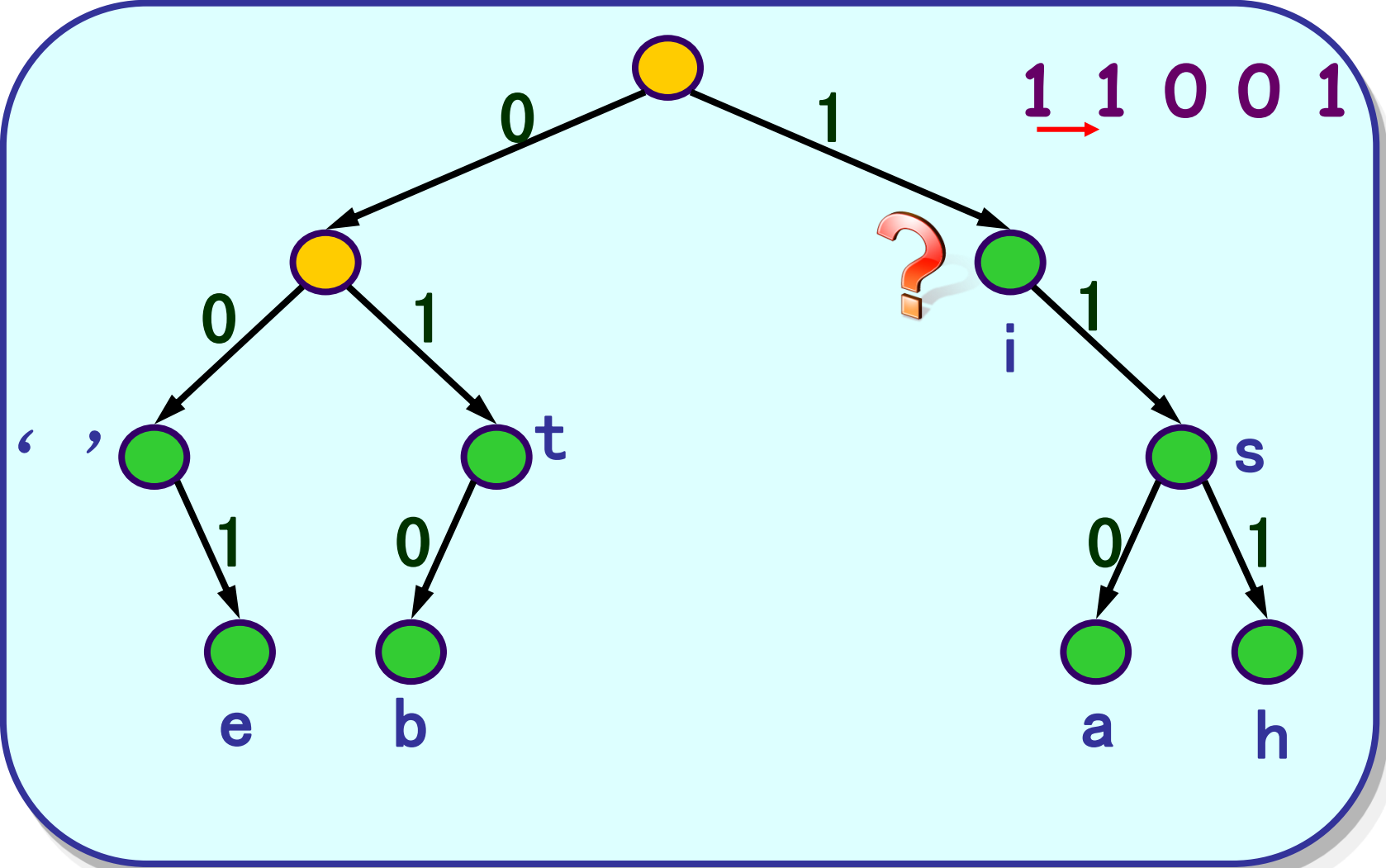
字符	space	t	b	e	s	a	h	i
出现次数	4	3	2	2	2	1	1	1
编码	00	01	010	001	11	110	111	1
解码	1 1 0 0 1							
	ii is? iie? s is? se? at?							

前缀码



- 设 $a_1a_2\cdots a_{n-1}a_n$ 为长度为 n 的符号串，称 $a_1, a_1a_2, \dots, a_1a_2\cdots a_{n-1}$ 分别为该符号串的长度为 $1, 2, \dots, n-1$ 的**前缀 (prefix)**
- 例
 - 0011的长为1~3的前缀: 0, 00, 001

前缀码



前缀码



- 设 $A = \{\beta_1, \beta_2, \dots, \beta_m\}$ 为一个符号串集合，若对于任意的 $\beta_i, \beta_j \in A, i \neq j, \beta_i, \beta_j$ 互不为前缀，则称 A 为**前缀码**（**prefix code**）或**无前缀码**（**prefix-free code**）
- 若 β_i 都是由 0, 1 组成的符号串，则称 A 为**二元前缀码**
 - 不加特殊说明时，所称的“前缀码”默认指二元前缀码

前缀码



字符	space	t	b	e	s	a	h	i
Code1	000	001	010	011	100	101	110	111
Code2	10	00	110	011	010	1110	11110	11111
Code3	00	01	010	001	11	110	111	1



前缀码



- 一个编码方案也可以使用二叉位置树来表示：
 - 对于树中的分枝点，令与它左孩子（如果存在）关联的边标记为0
 - 与右孩子（如果存在）关联的边标记为1
 - 对每个顶点而言，其编码就是由根到该顶点的道路中各边标号依次构成的序列
- 一般地讲，任何一个二元前缀码编码方案的二叉位置树表示中表示符号的顶点都一定是叶子
- 给定任一个二叉位置树编码后，各叶子的码构成的集合（或其子集）是一个前缀码

前缀码



- 假设一个二元前缀码编码方案中包括 t 种符号，每种符号在消息中出现的次数为 w_i ，其编码长度为 l_i ，则消息的总长度为 $\sum_{i=1}^t w_i \cdot l_i$

编码的目标是希望 $\sum_{i=1}^t w_i \cdot l_i$ 的值尽可能小



- 设二叉树 T 有 t 个叶子 v_1, v_2, \dots, v_t , 分别赋予它们一个权值（非负实数）为 w_1, w_2, \dots, w_t , 称 $W(T) = \sum_{i=1}^t w_i \cdot l(v_i)$ 为 T 的**权（weight）**, 其中 $l(v_i)$ 是 v_i 的层数

后文证明中, 为书写的方便,
用 $w[v_i]$ 表示 w_i , 用 $l[v_i]$ 表示 $l(v_i)$

前綴碼

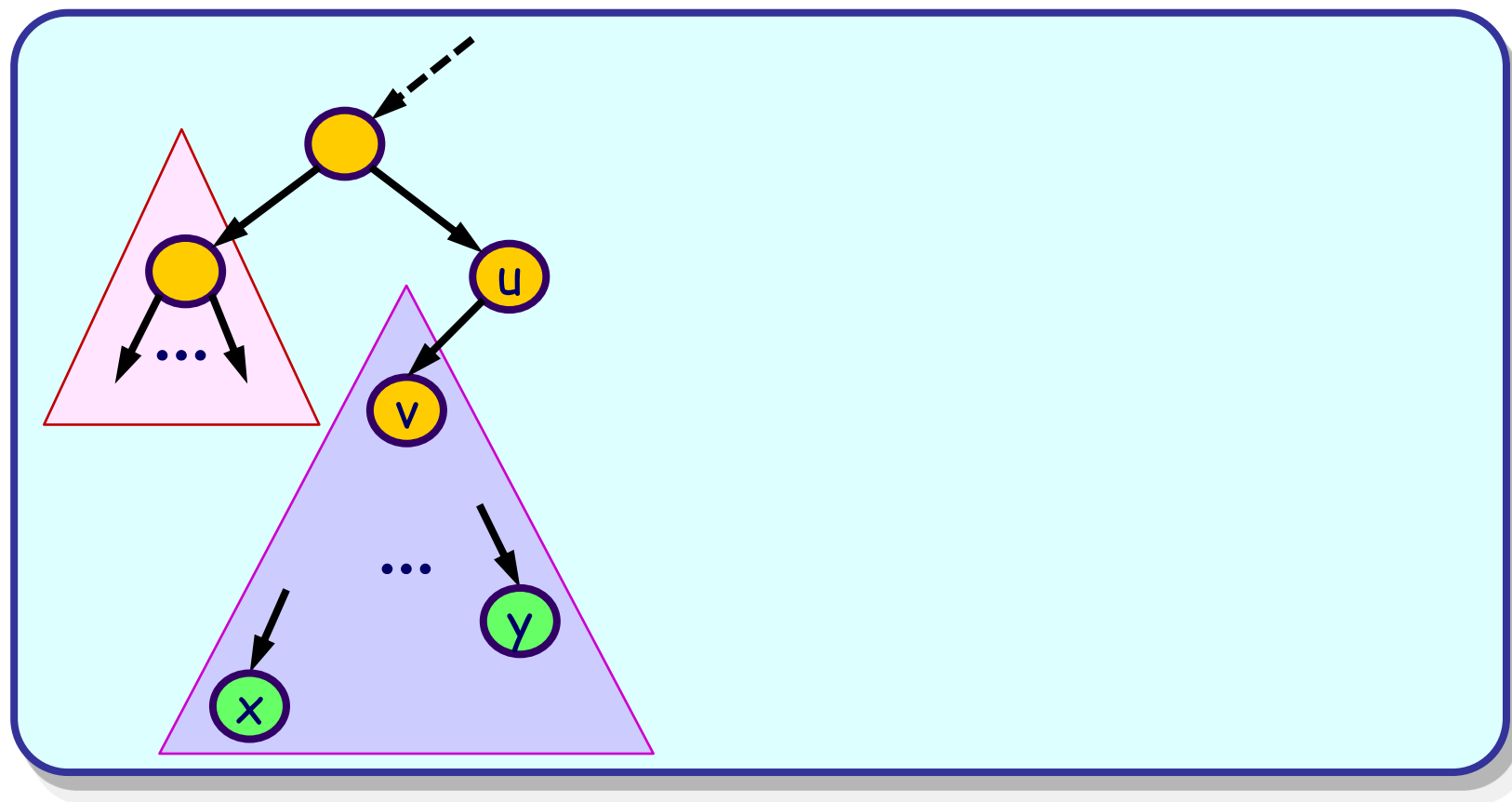


Char	space	t	b	e	s	a	h	i
Code1	000	001	010	011	100	101	110	111
48 bits	010 101 001 000 111 100 000 001 110 011 000 010 011 100 001 000							
Code2	10	00	110	011	010	1110	11110	11111
46 bits	110 1110 00 10 11111 010 10 00 11110 011 10 110 011 010 00 10							

二元前缀码与二叉位置树



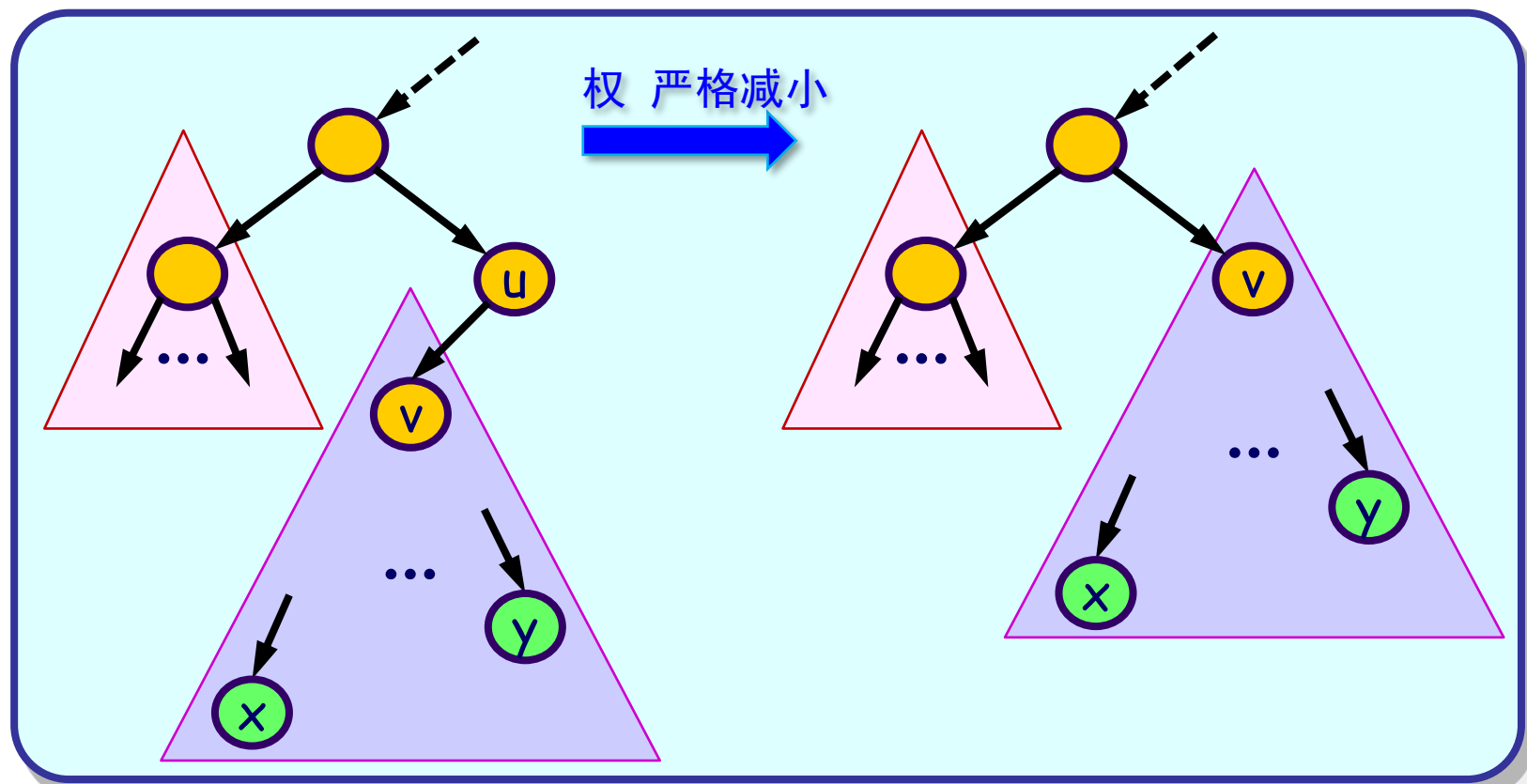
- 由此，假定对应于编码方案的二叉位置树的分枝点出度都为 2
- 否则，例如：



二元前缀码与二叉位置树



- 由此，假定对应于编码方案的二叉位置树的分枝点出度都为 2
- 否则，例如：



前缀码



- 在所有有 t 个叶子，带权 w_1, w_2, \dots, w_t 的二叉树中，权最小的二叉树称为**最优二叉树**（**optimal binary tree**）

Huffman算法



- Huffman于1952年给出了最优二叉树的构造方法（即构造最优二元前缀码的方法），因此最优二叉树也称作**Huffman树**，对应的最优二元前缀码也称作**Huffman码**
- 它使用可变长度的编码表来编码字符
- 该算法的基本思想是使得从根到权值大的叶子的道路较短，反之从根到权值小的叶子的道路较长

Huffman算法



- Huffman算法基于贪婪策略
- 输入：一组符号及其非负权值 w_1, w_2, \dots, w_t
- 输出：叶子顶点的权值为 w_1, w_2, \dots, w_t 的最优二叉树的根 v

Huffman算法



输入：一组符号及其非负权值 w_1, w_2, \dots, w_t

输出：叶子顶点的权值为 w_1, w_2, \dots, w_t 的最优二叉树的根 v

1. 构造 t 个顶点 v_1, v_2, \dots, v_t ，权值分别为 w_1, w_2, \dots, w_t ， $S \leftarrow \{v_1, v_2, \dots, v_t\}$

2. 若 $|S|=1$ ，则输出 S 中唯一的元素，否则

2.1. 假设 x 和 y 是 S 中权值最小的两个顶点

2.2. 构造一个新的顶点 z ，令 z 的左孩子是 x ，右孩子是 y ，权值为 x 和 y 权值之和

2.3. $S \leftarrow (S - \{x, y\}) \cup \{z\}$ ，返回步骤 2

Huffman算法



HUFFMAN (C)

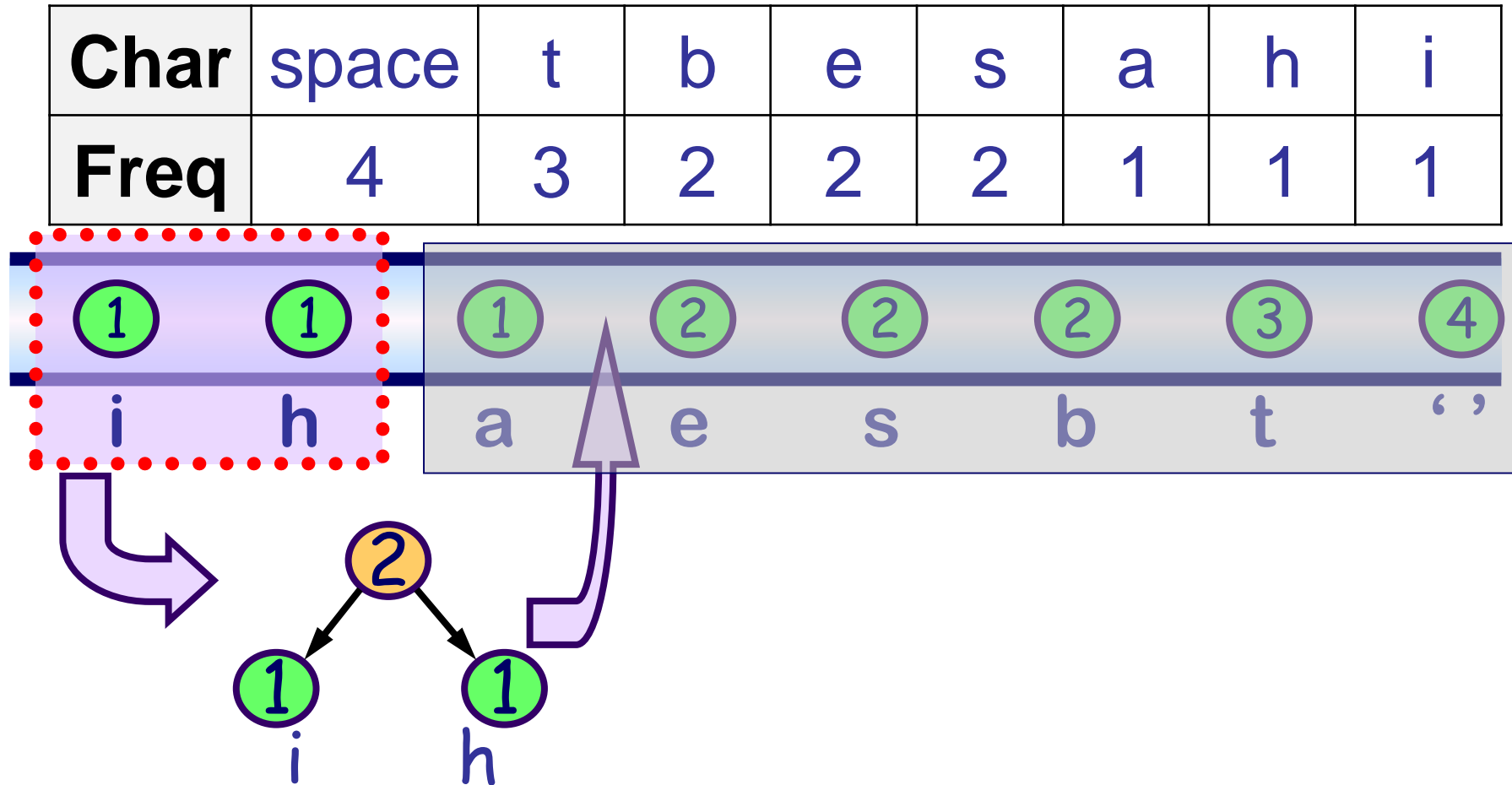
输入: $C = \{x_1, x_2, \dots, x_n\}, w(x_1), \dots, w(x_n)$

1. $n \leftarrow |C|$
2. $Q \leftarrow C$ // Q 是优先级队列
3. **for** $i = 1$ **to** $n - 1$ **do**
4. $z \leftarrow \text{ALLOCATE-NODE}()$
5. $z.\text{left} \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$
6. $z.\text{right} \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$
7. $w(z) \leftarrow w(x) + w(y)$
8. $\text{INSERT}(Q, z)$
9. **return** $\text{EXTRACT-MIN}(Q)$

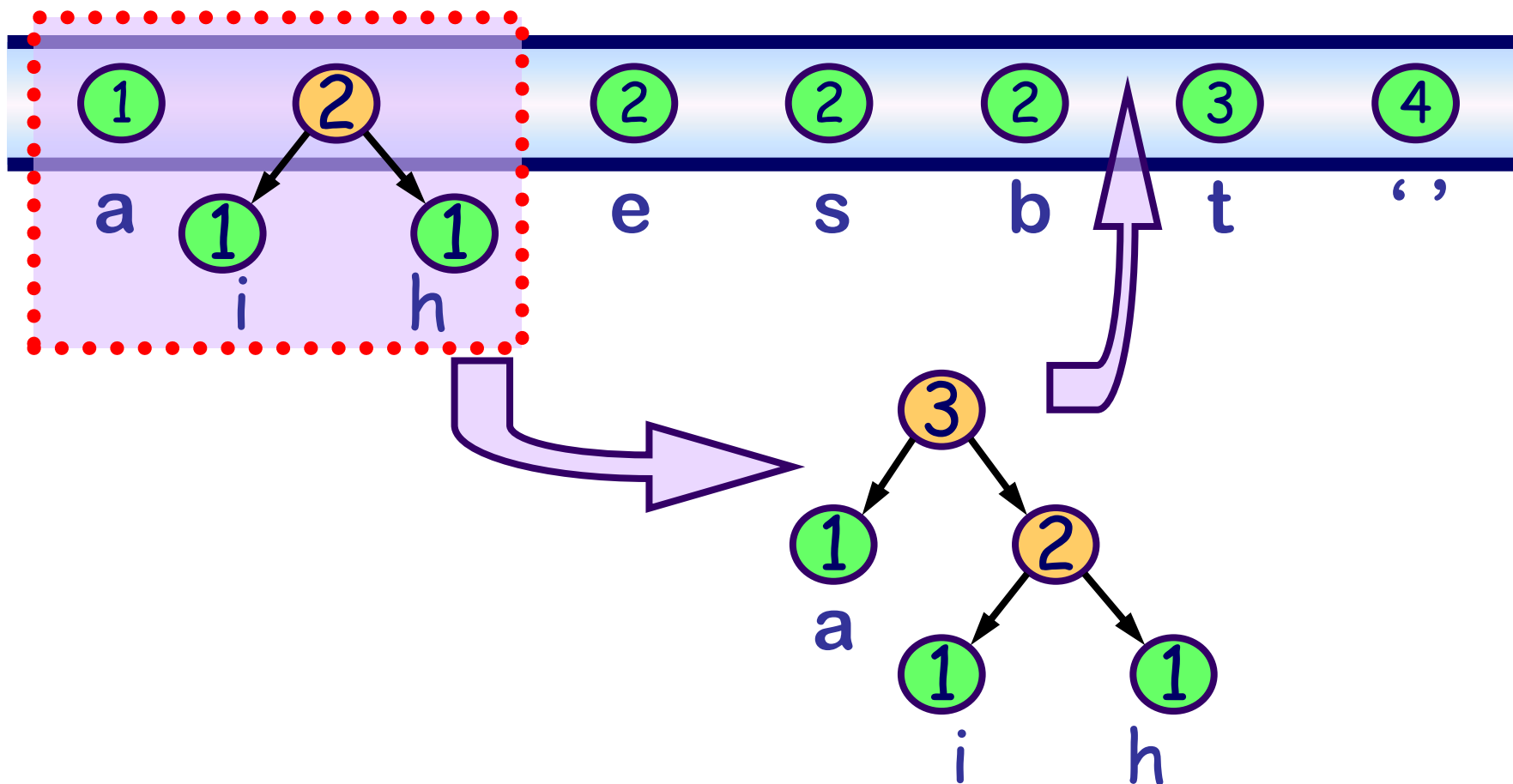
Huffman算法



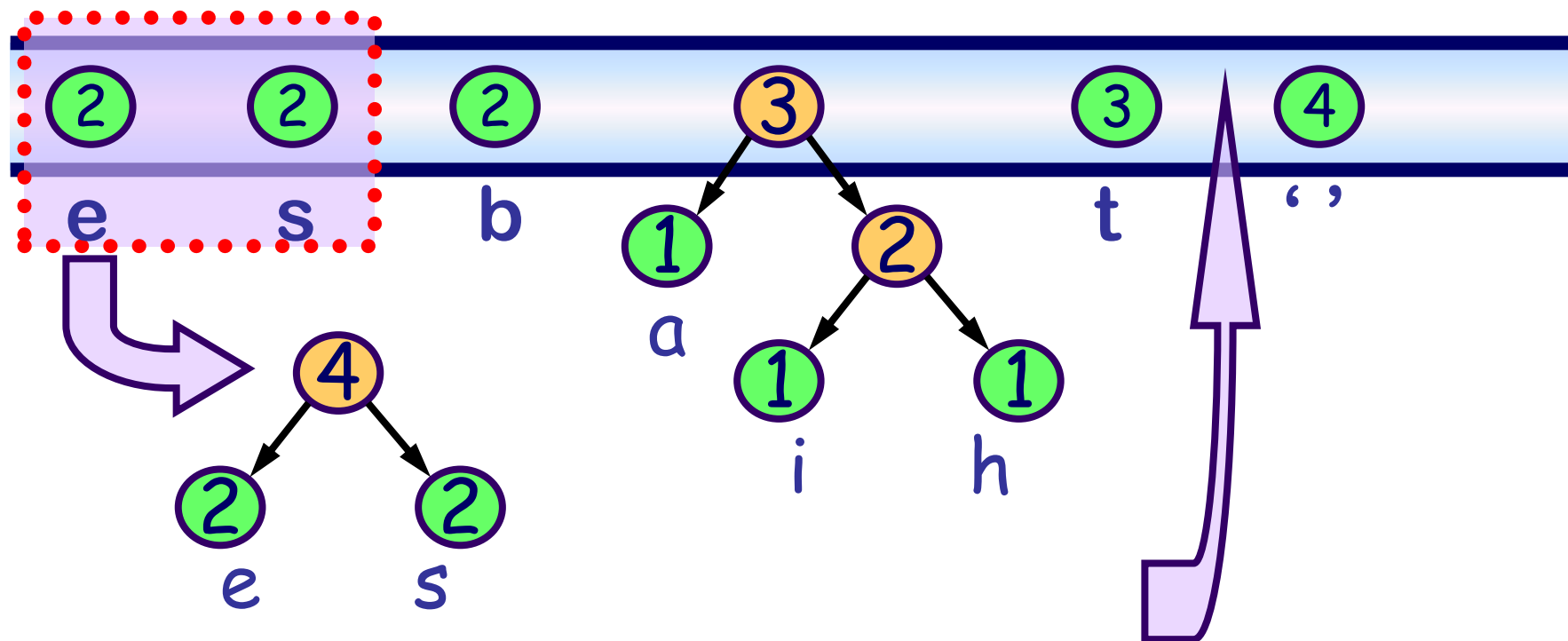
● 例



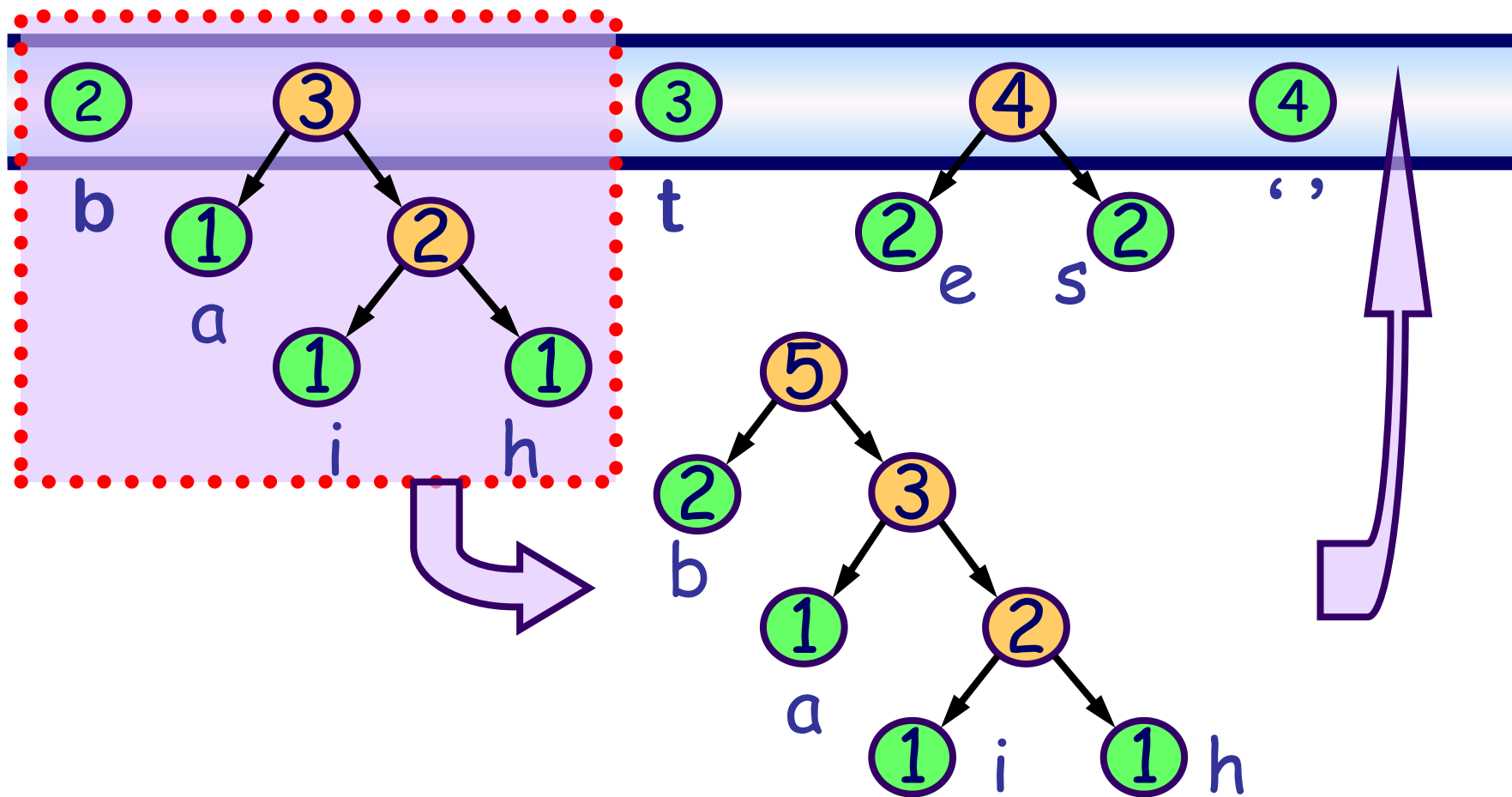
Huffman算法



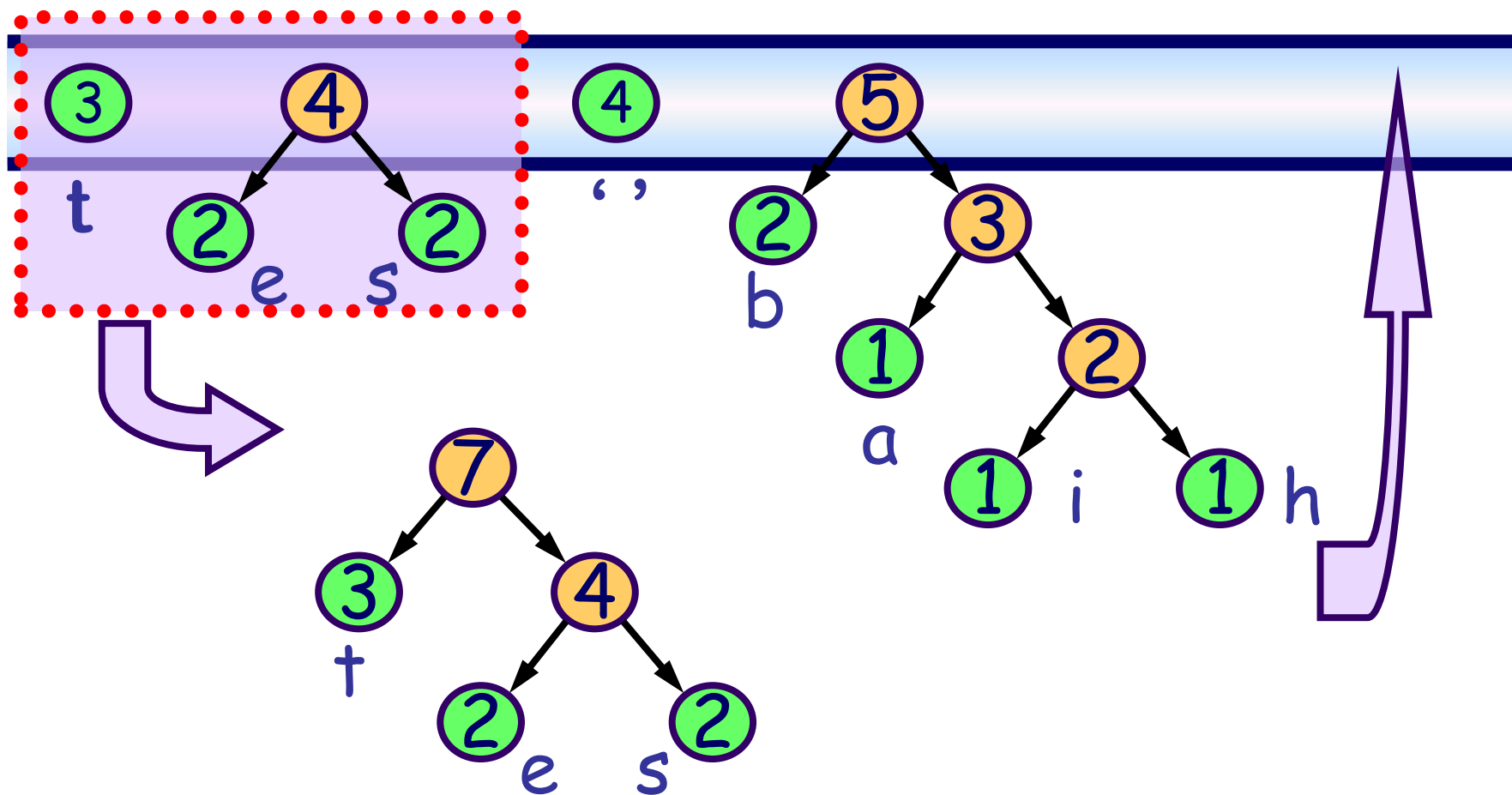
Huffman算法



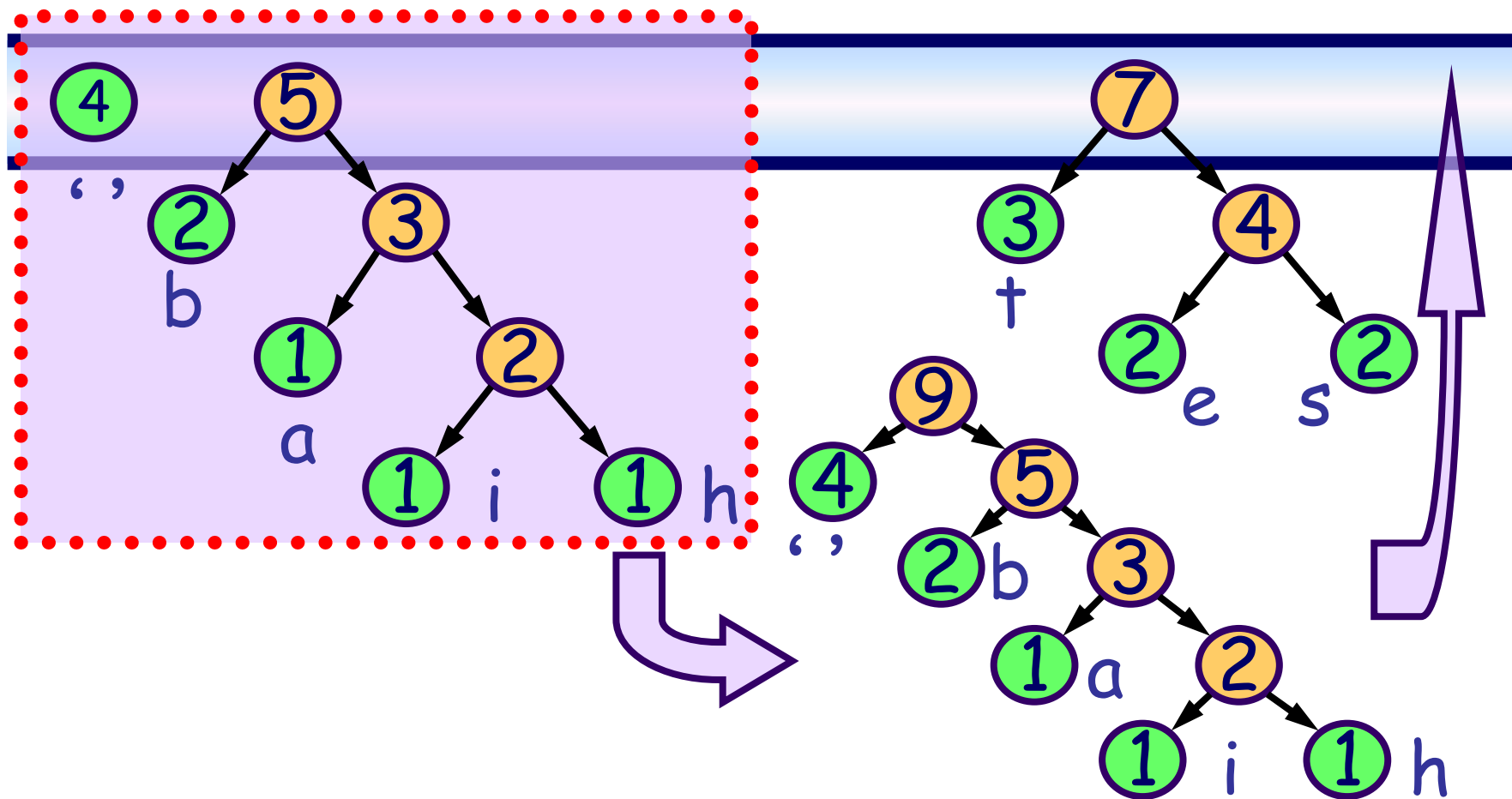
Huffman算法

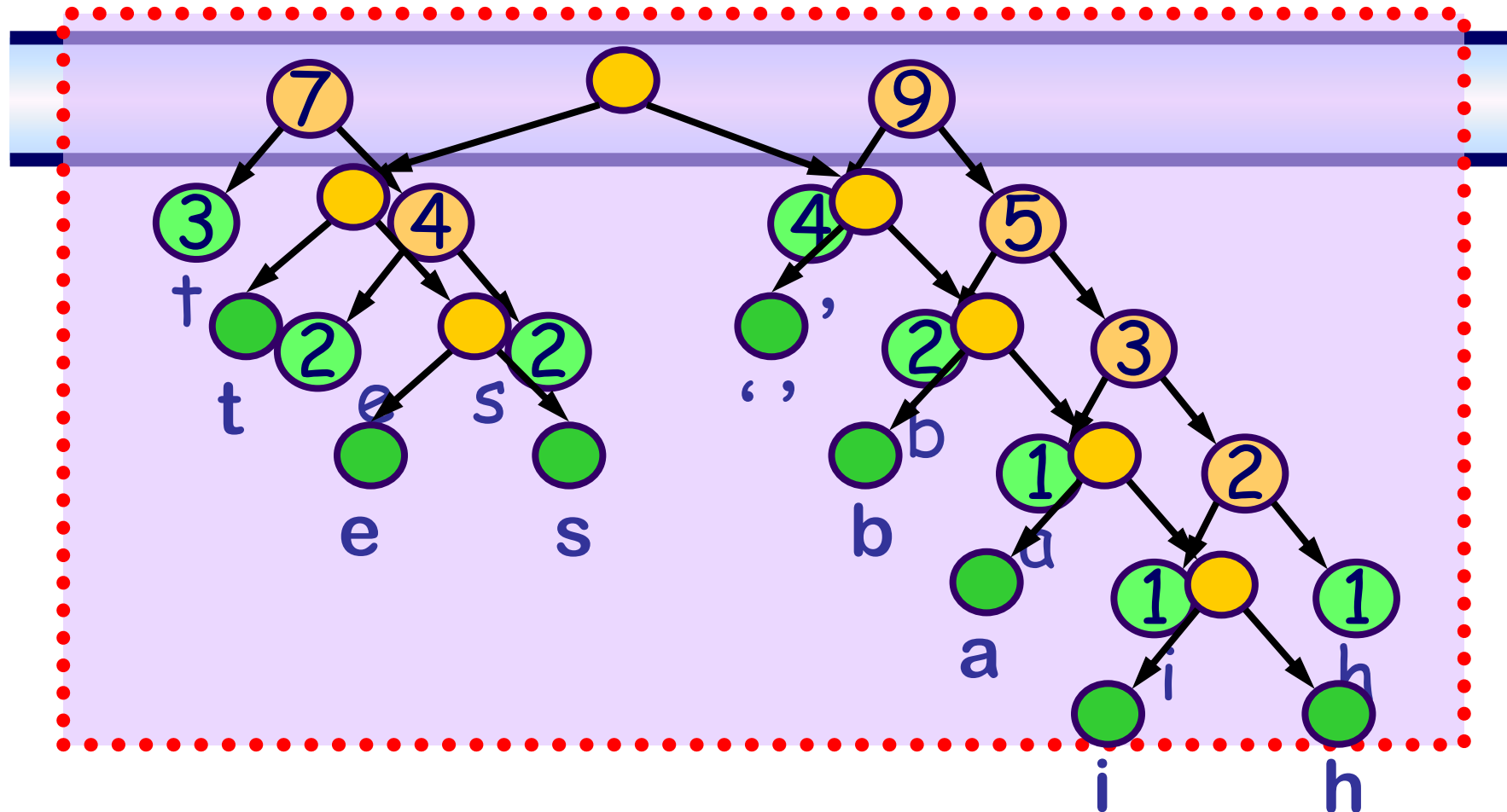


Huffman算法



Huffman算法





Huffman算法



HUFFMAN (C)

时间复杂度: $O(n \log n)$

输入: $C = \{x_1, x_2, \dots, x_n\}, w(x_1), \dots, w(x_n)$

1. $n \leftarrow |C|$

2. $Q \leftarrow C$

// Q 是优先级队列

3. **for** $i = 1$ **to** $n - 1$ **do**

4. $z \leftarrow \text{ALLOCATE-NODE}()$

$O(1)$

5. $z.\text{left} \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$

$O(\log n)$

6. $z.\text{right} \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$

$O(\log n)$

7. $w(z) \leftarrow w(x) + w(y)$

8. **INSERT** (Q, z)

$O(\log n)$

9. **return** $\text{EXTRACT-MIN}(Q)$

Huffman算法



- 定理

- Huffman算法对任意规模为 n ($n \geq 2$) 的字符集 C ，都可以得到关于 C 的最优前缀码的二叉树。

Huffman算法的正确性证明

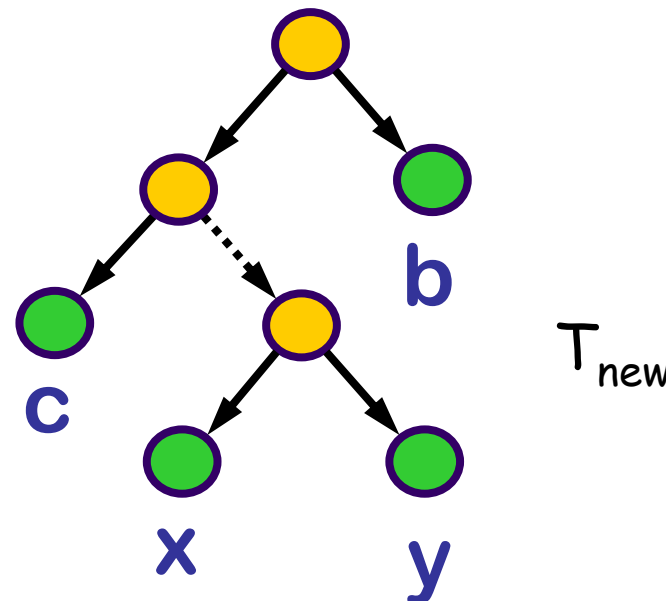
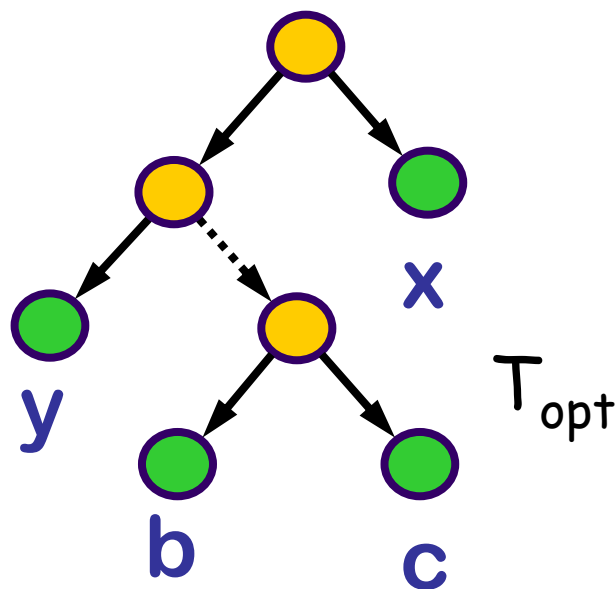


- 假设 x 和 y 是 C 中权值最小的两个字符
 - 即贪婪算法中**最先**选择的两个字符
 - 如果有多个权值最小的字符（即权值相同）则任选两个即可
- 性质一：
 - 存在一棵关于 C 的最优二叉树 T_{opt} ，使得 x 和 y 在 T_{opt} 中处于最大层，且二者为兄弟顶点。

性质一的证明



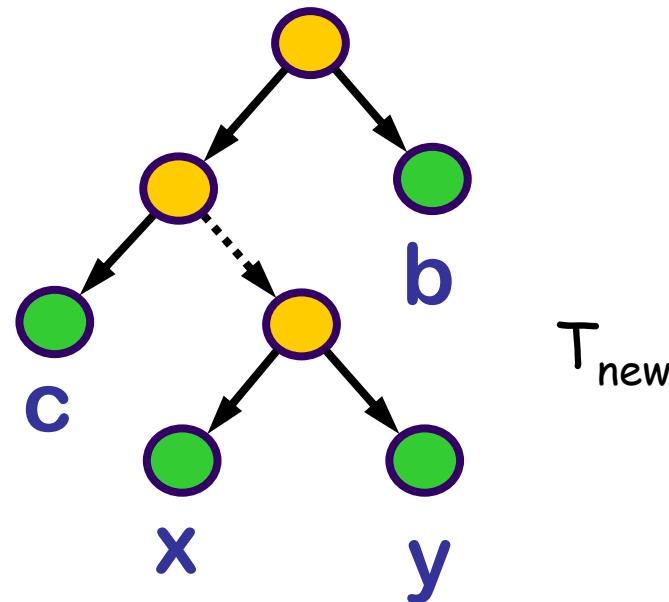
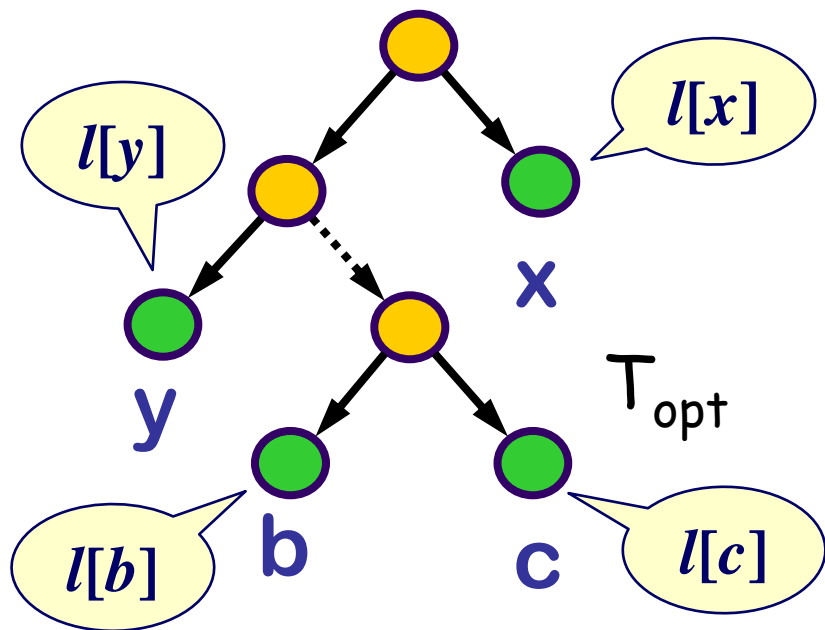
- 考察 T_{opt} 中深度最大的两个的兄弟结点，不妨记之为 b 和 c
- 将 b 与 x 对换， c 与 y 对换
- 计算 $W(T_{\text{opt}}) - W(T_{\text{new}})$
- 如果 $W(T_{\text{opt}}) - W(T_{\text{new}}) \geq 0$ 即证明了性质1



性质一的证明



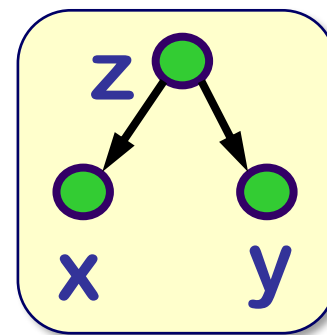
- $$\begin{aligned} & W(T_{\text{opt}}) - W(T_{\text{new}}) \\ &= (w[x]l[x] + w[y]l[y] + w[b]l[b] + w[c]l[c]) - \\ &\quad (w[x]l[b] + w[y]l[c] + w[b]l[x] + w[c]l[y]) \\ &= \underbrace{(l[b] - l[x])(w[b] - w[x])} + \underbrace{(l[c] - l[y])(w[c] - w[y])} \\ &\geq 0 \end{aligned}$$



Huffman算法的正确性证明



- 假设 x 和 y 是 C 中权值最小的两个字符
 - 即贪婪算法中**最先**选择的两个字符
 - 性质二：
 - 令 z 是一个权值为 $w[z] = w[x] + w[y]$ 的新字符，并令 $C' = (C - \{x, y\}) \cup \{z\}$ 。
- 设 T' 为关于 C' 的最优二叉树。然后我们可以用替换 z 的方法从 T' 得到一棵关于 C 的最优二叉树 T



Huffman算法的正确性证明



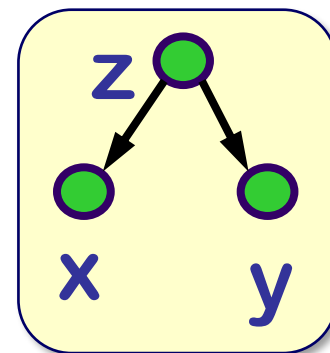
● 性质二:

“合成” x 和 y 得到 z 和 C'

使用Huffman算法构造
 C' 的最优二叉树 T'

在由 x 和 y 替换回 z 后由 T'
得到一棵关于 C 的二叉树 T

T 是关于 C 的一棵**最优**二叉树



论证

性质二的证明



- 证明:

- $$\begin{aligned} W(T) &= W(T') + w[x]l[x] + w[y]l[y] - w[z]l[z] \\ &= W(T') + w[x](l[z]+1) + w[y](l[z]+1) - (w[x]l[x] + w[y])l[z] \\ &= W(T') + w[x] + w[y] \end{aligned} \quad \dots (1)$$

- 由性质一可知，我们可以选择关于 C 的最优二叉树 T_{opt} 满足 x 和 y 在 T_{opt} 中处于最大层，且二者为兄弟顶点
- 使用反证法证明 $W(T) = W(T_{\text{opt}})$
- 否则的话必然有 $W(T) > W(T_{\text{opt}})$... (2)

性质二的证明



- 证明:

- 设 T 是通过用三个结点替换 z 从 T' 得到的树

- $W(T) = W(T') + w[x] + w[y]$... (1)

- 选择关于 C 的最优二叉树 T_{opt} 满足 x 和 y 在 T_{opt} 中处于最大层, 且二者为兄弟顶点

- 使用反证法证明 $W(T) = W(T_{\text{opt}})$

- 否则的话必然有 $W(T) > W(T_{\text{opt}})$... (2)

- 从 T_{opt} 中删除 x 和 y , 并得到关于 C' 的另一棵树 T''

- $W(T'') = W(T_{\text{opt}}) - w[x]l[x] - w[y]l[x] + (w[x] + w[y])(l[x] - 1)$

$$= W(T_{\text{opt}}) - w[x] - w[y]$$

$$< W(T) - w[x] - w[y] = w(T')$$

由(2)式

由(1)式

- 得到 $W(T'') < W(T')$, 与假设相矛盾: T' 是关于 C' 的最优二叉树

Huffman算法



- 定理

- Huffman算法对任意规模为 n ($n \geq 2$) 的字符集 C ，都可以得到关于 C 的最优前缀码的二叉树。

- 证明：

- 对 $|C|$ 进行归纳，并由性质二即可得。

二路归并模式

2-way Merge Patterns

刘铎

liuduo@bjtu.edu.cn



二路归并模式



● 例

- 假设有3个有序列表 L_1 、 L_2 和 L_3 ，长度分别为30、20和10
- 希望将它们归并为一个有序列表
- 但每次只能对两个有序列表进行归并操作
- 希望找到一个最佳的归并模式，使总比较次数最小化

二路归并模式



- 例:

- 可以先归并 L_1 和 L_2 , 共使用 $30+20=50$ 次比较, 得到一个长度为 50 的有序列表 L_4
- 然后将此列表 L_4 与 L_3 归并, 又使用了 $50+10=60$ 次比较
- 因此总比较次数为 $50 + 60 = 110$

$L_1 : 30$ $L_2 : 20$ $L_3 : 10$

二路归并模式



- 例:

- 或者，可以先归并 L_2 和 L_3 ，共使用 $20+10=30$ 次比较，得到一个长度为 30 的有序列表 L_5
- 然后将此列表 L_5 与 L_1 归并，又使用了 $30+30=60$ 次比较
- 因此总比较次数为 $30 + 60 = 90$
- 显然后一种归并模式是更优的

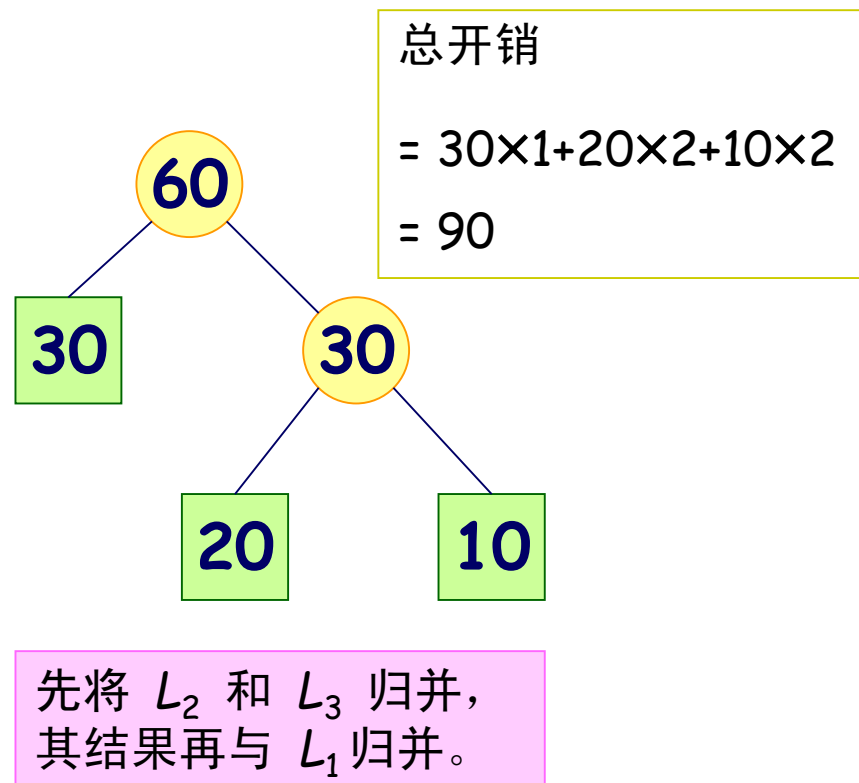
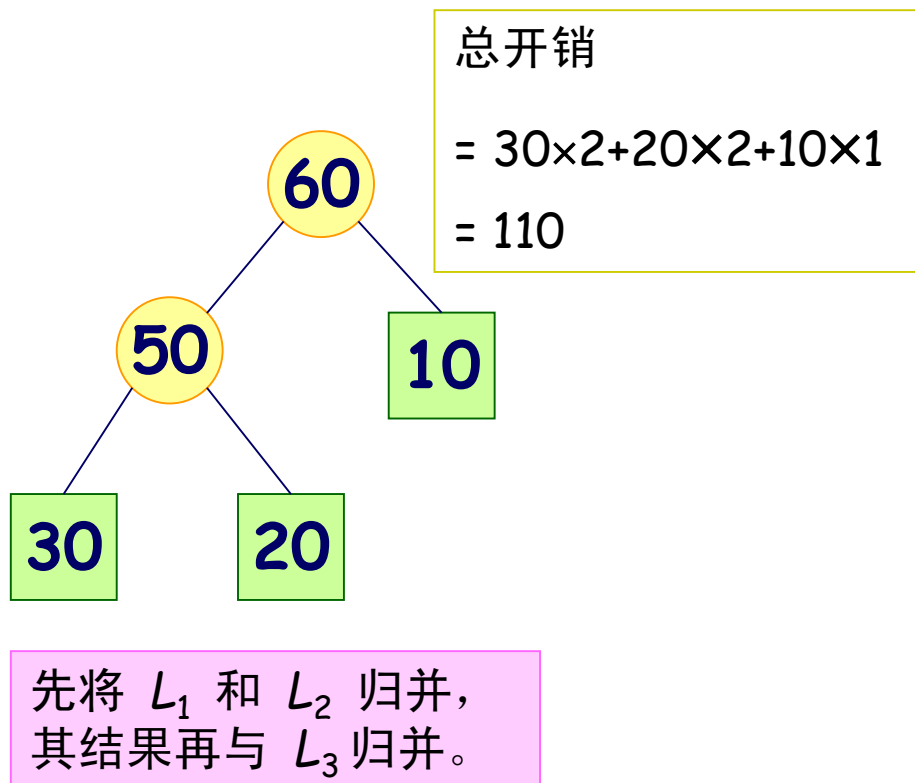
$L_1 : 30$ $L_2 : 20$ $L_3 : 10$

二路归并模式



- 二叉归并树
 - 使用一棵二叉树来描述归并模式
 - 叶子顶点是初始列表
 - 每次两个顶点的归并都会创建一个新的父结点，其大小是两个子结点的大小之和
- 例如， 前述两种不同的归并模式可表示如下图所示：

二路归并模式



归并开销 = 所有 根到叶子顶点的道路长度 \times 叶子顶点的权值 之和
= 所有 叶子顶点的层数 \times 叶子顶点的权值 之和

二路归并模式



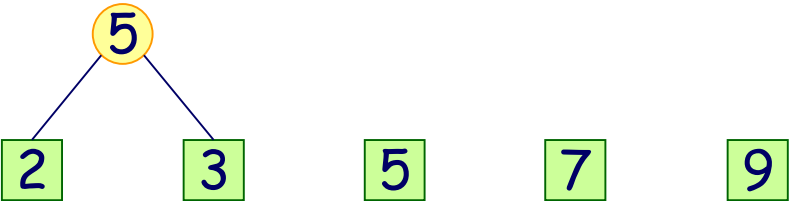
- 类似于Huffman算法
- 时间复杂度为 $O(n \log n)$

二路归并模式 —— 示例



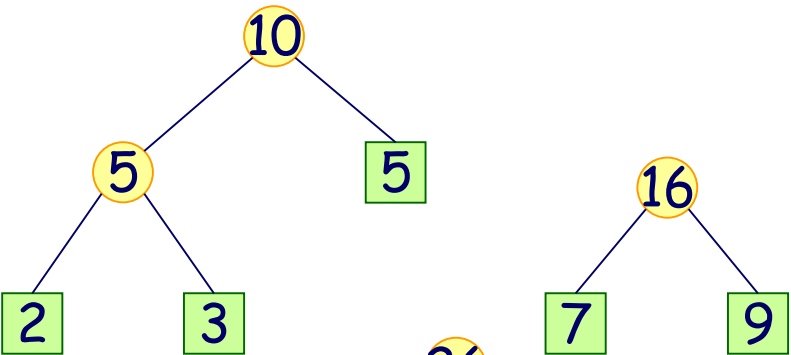
2 3 5 7 9

初始化：5个赋权叶节点

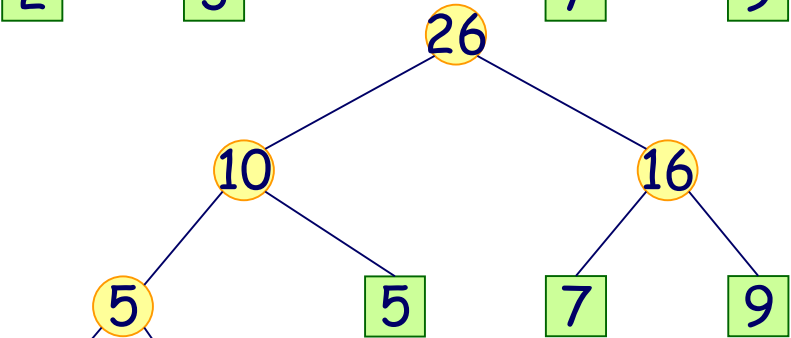


迭代1：将2和3归并为5

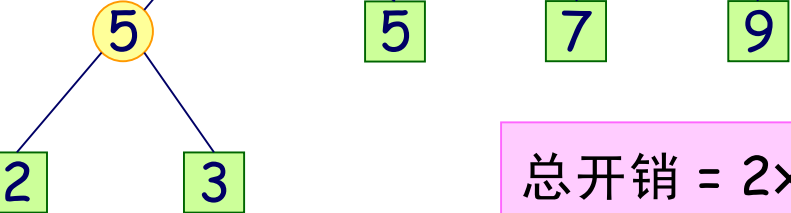
迭代2：将5和5归并为10



迭代3：将7和9（从7、9和10中选择）归并为16



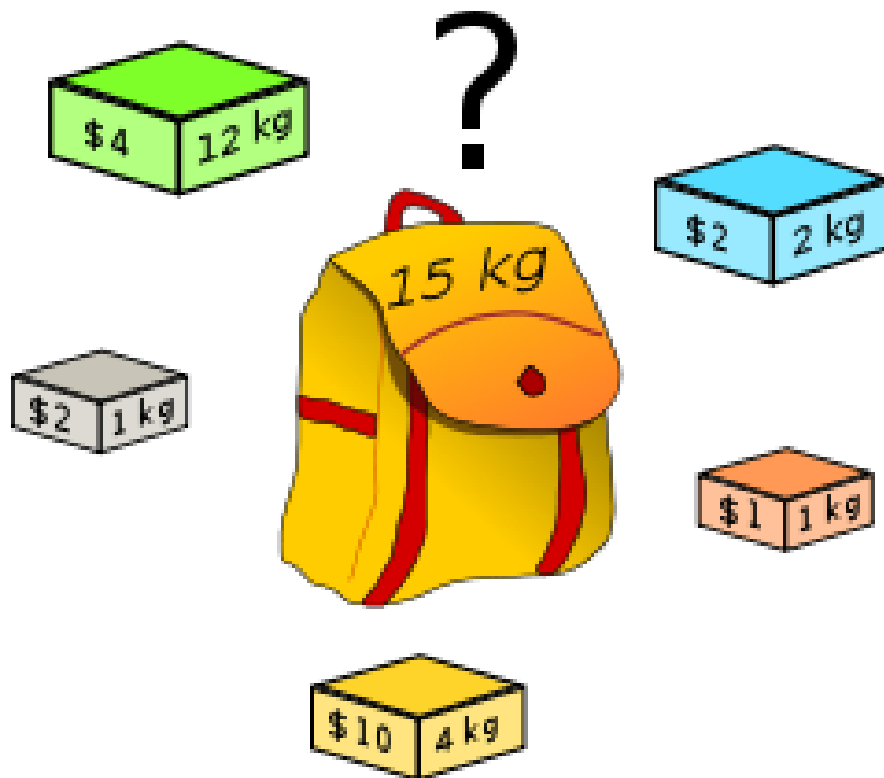
迭代4：将10和16归并为26



总开销 = $2 \times 3 + 3 \times 3 + 5 \times 2 + 7 \times 2 + 9 \times 2 = 57$

背包问题

Knapsack problem



背包问题



- 贼，夜入豪宅，可偷之物甚多，而负重能力有限，偷哪些才更加不枉此行？
- 例：
 - 他的背包只能装**11**公斤赃物
 - 被盗者家中共有**5**块金子（或**5**件物品），每块金子的纯度不尽相同
 - 因此每块金子都有一定的重量和价值
 - 称作“**0-1**”是因为每块金子或者整块被拿走（**1**）、或者完全不拿（**0**）
- 他必须选择要偷的金子，在总重量不能超过**11**公斤的前提下，最大限度地提高他的赃物的总价值

W = 11

物品 Item	价值 Value	重量 Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

背包问题



- 背包问题是一个典型的离散优化/组合优化问题：
 - 有 n 种物品
 - 物品 i 的重量为 w_i ，价值为 v_i
 - 可装入容量为 C 的背包
 - 选择将哪些物品装入背包，使得在总重量不超过 C 的前提下，总价值达到最大

背包问题

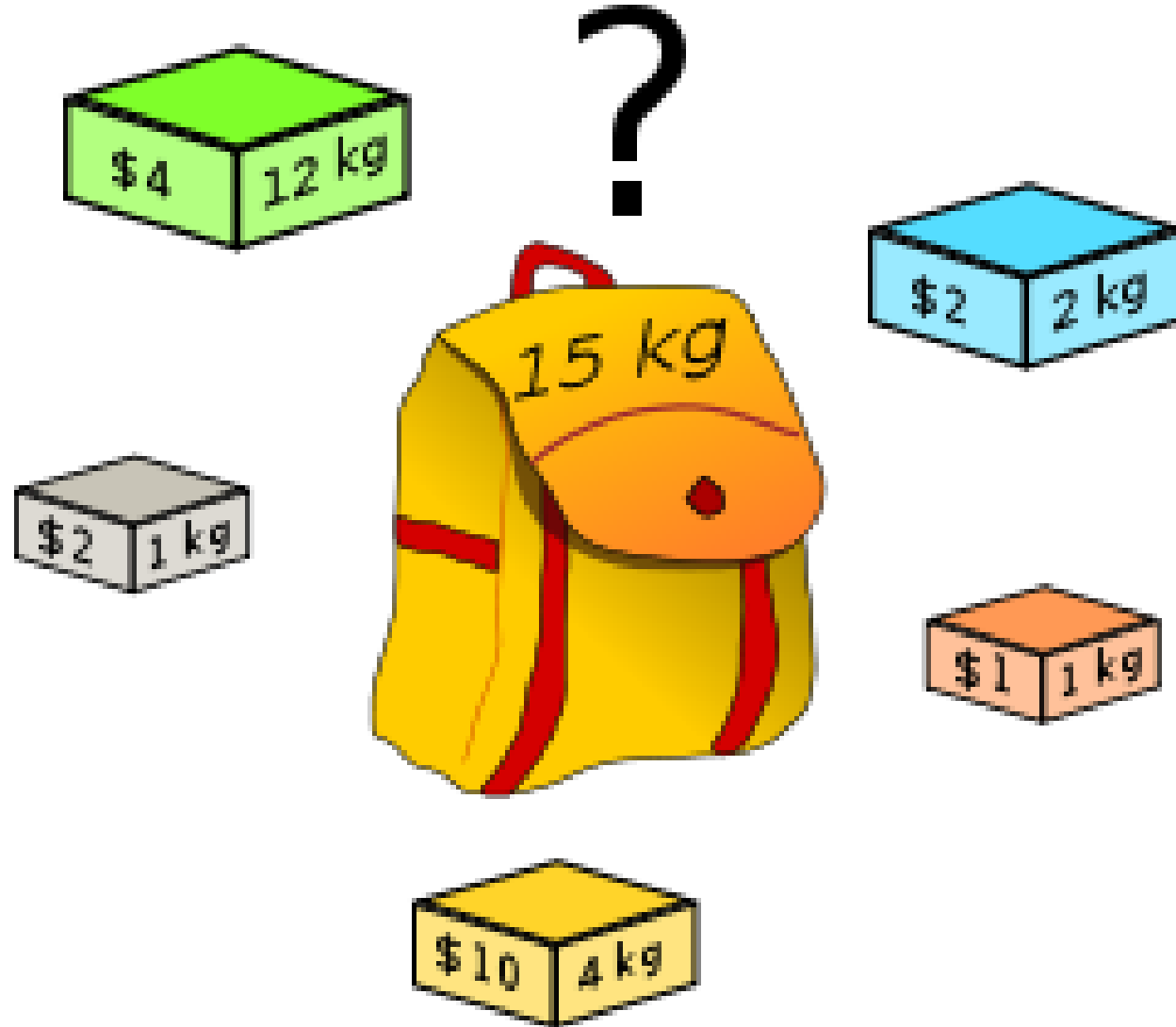


- 背包问题是一个典型的离散优化/组合优化问题
- 受限条件下的资源分配问题
- 背包问题的判定性问题/版本是：“在总重量不超过 C 的情况下，是否可以获得至少为 V 的总价值？”

背包问题



- 示例:

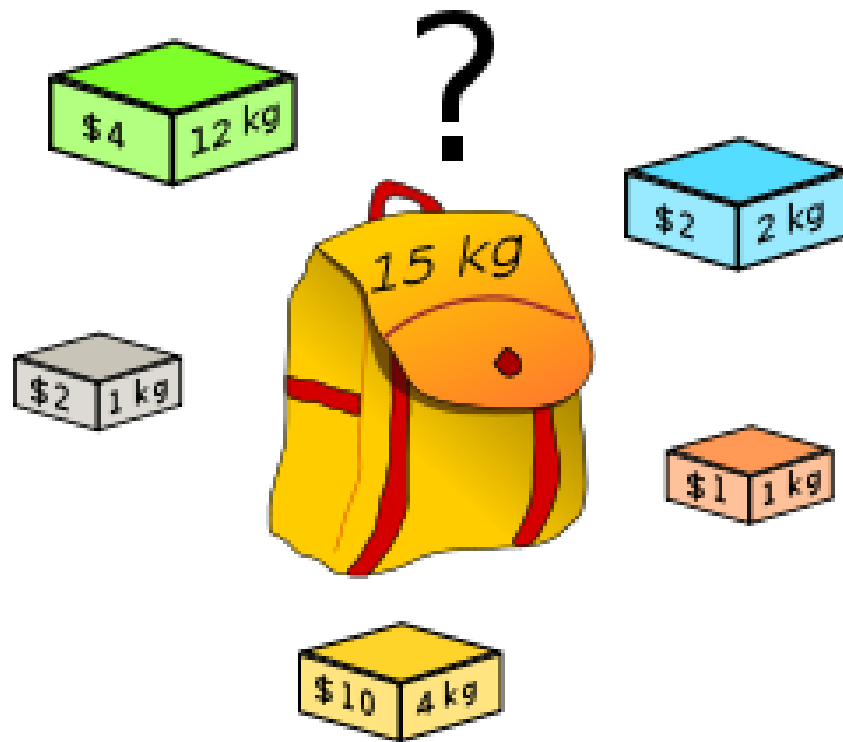


背包问题



● 解:

- 如果每种物品都有**足够多个**（任意多个/无穷多个），那么就选择三个黄色的和三个灰色的
- 如果每种物品都**只有一个**，那么就选择除绿色的之外的其他所有物品



背包问题



- 对于第 i 种物品，当在考虑是否将其装入背包时，可能有多种选择：
 - 要么全部装入背包，要么全部不装入背包，而不能只装入物品 i 的一部分
 - “0-1背包问题（**0-1 knapsack problem**）”，意即物品 i 只允许拿0个或1个
 - 最常见（常用）的一类背包问题
 - 可抽象表示为：

$$\text{Max} (\sum v_j x_j)$$

$$\text{s.t. } \sum w_j x_j \leq C, x_j \in \{0, 1\}$$

背包问题



- 对于第 i 种物品，当在考虑是否将其装入背包时，可能有多种选择：
 - 该种物品可以选取的个数存在上限 M_j
 - 受限背包问题 (bounded knapsack problem)
 - 该种物品可以选取多个，选取的数目没有限制
 - 无限制背包问题 (unbounded knapsack problem)
 - 该种物品可以选取0到1之间的分数个
 - 分数背包问题 (fractional knapsack problem)

背包问题



- 对于第 i 种物品，当在考虑是否将其装入背包时，可能有多种选择：
 - 该种物品可以选取的个数存在上限 M_j
 - 受限背包问题 (bounded knapsack problem)
 - $\text{Max} (\sum v_j x_j)$, s.t. $\sum w_j x_j \leq C, x_j \in \{0, 1, 2, \dots, M_j\}$
 - 可以转化为0-1背包问题：将该种物品复制为 M_j 种
 - 该种物品可以选取多个，选取的数目没有限制
 - 无限制背包问题 (unbounded knapsack problem)
 - $\text{Max} (\sum v_j x_j)$, s.t. $\sum w_j x_j \leq C, x_j \in \{0, 1, 2, \dots\}$
 - 可以转化为受限背包问题：根据背包容量计算每种物品的可选取数目上限

背包问题 —— 贪婪策略



- 有如下 3 种不同的贪婪策略：
 - （最小重量优先策略）按照重量的非降顺序依次考虑各个物品，只要背包的剩余容量允许即选取
 - （最大价值优先策略）按照价值的非升顺序依次考虑各个物品，只要背包的剩余容量允许即选取
 - （最大价值-重量比优先策略）计算每个物品的价值-重量比，按照这个比值的非升顺序依次考虑各个物品，只要背包的剩余容量允许即选取
- 这 3 种想法都是很自然的，但是否可以确保得到最优方案？

背包问题 —— 贪婪策略



● 示例： $n = 5$ ， $C = 100$

w	20	30	40	50	60
v	20	30	44	55	60
v/w	1	1	1.1	1.1	1

表 1

策略	x_i					总价值
Min w_i	1	1	1	0	0	94
Max v_i	0	0	1	0	1	104
Max v_i/w_i	0	0	1	1	0	99
Optimal	1	1	0	1	0	105

表 2

Dantzig的贪婪算法



- George Dantzig在1957年提出了一种**贪婪近似算法**
- 就是最大价值-重量比优先策略
- 之前已经看到了它并不能保证得到最优解
- 那么，它能够多么“接近”最优？

Dantzig的贪婪算法



- 对于无限制背包问题
 - 如果 A 是最优值，那么Dantzig的贪婪算法可以保证至少达到 $A/2$ 的总价值值
- 对于受限背包问题
 -

Dantzig的贪婪算法



- $C = W_0$
- 你的选择是什么？



$$\begin{aligned} w &= 1 \\ v &= 1 \end{aligned}$$

$$\begin{aligned} w &= W_0 \\ v &= W_0 - \varepsilon \end{aligned}$$

Dantzig的贪婪算法



- 对于无限制背包问题
 - 如果 A 是最优值，那么Dantzig的贪婪算法可以保证至少达到 $A/2$ 的总价值
- 对于受限背包问题
 - 可能非常差
 - 但可以稍作修改，使得可以保证得到最优值的一半——留作学生自行查阅相关资料
- 对于分数背包问题

Dantzig的贪婪算法



- 对于分数背包问题，Dantzig的贪婪算法可以保证达到最优值
- 示例：

$$n = 3, C = 20, (v_1, v_2, v_3) = (25, 24, 15)$$

$$(w_1, w_2, w_3) = (18, 15, 10)$$

$$\text{解: } v_1/w_1 = 25/18 = 1.32$$

$$v_2/w_2 = 24/15 = 1.6$$

$$v_3/w_3 = 15/10 = 1.5$$

$$\text{最优解: } x_1 = 0, x_2 = 1, x_3 = 1/2$$

贪婪策略

/贪婪算法/贪心算法/贪心策略



贪婪算法/贪心算法/贪婪策略



- 它的求解过程是多步判断过程
- 在每一步，它总是做出目前看起来最好的选择
- 它进行局部最优选择，以期得到全局最优解

贪婪策略的特点（优势/不足）



- 简单、快速
- 有时贪婪算法可以得到最优解
 - 需要进行正确性证明
- 有时贪婪算法**并不能**保证得到最优解
 - 局部最优并不总能“走向”全局最优
 - 例如棋牌游戏/比赛中，短期的牺牲可能带来长期的受益
 - 但可以快速得到一个可行解或近似解

贪婪算法的分析策略



- 贪婪算法保持领先
 - 论证在贪心算法的每一步后，其解至少和其他算法一样好
 - 例如找零问题（1, 5, 10）
- 参数交换
 - 在不影响解的质量的前提下，将任意解逐步转化为贪婪算法得到的解
 - 例如（计件的）装载问题、活动选择问题、最小延迟调度
- 结构性特征
 - 发现一个简单的“结构性”界，之后断言每个可能的解决方案都必须有某个确定的值。然后证明贪婪算法总是达到这个界
 - 例如活动划分问题

End

