



上海师范大学
Shanghai Normal University

《算法设计与分析》

8-贪心算法 (Greedy)

杨启哲

上海师范大学信机学院计算机系

2023 年 10 月 29 日



- 什么是贪心算法
- 贪心算法设计

► 什么是贪心算法

回顾: Dijkstra 算法

回顾一下 Dijkstra 算法，我们实际上每次在做这样一件事：

回顾：Dijkstra 算法

回顾一下 Dijkstra 算法，我们实际上每次在做这样一件事：

- 从 Y 中选取目前距离最近的一个点 u ，并将其加入 X 中。

回顾一下 Dijkstra 算法，我们实际上每次在做这样一件事：

- 从 Y 中选取目前距离最近的一个点 u ，并将其加入 X 中。

这其实是一个贪心的策略。

贪心算法

一个贪心算法会一直去选择当前情况下最优的解。



贪心算法



上海师范大学
Shanghai Normal University

对于贪心算法而言,



对于贪心算法而言,

- 设计是容易的, 因为只要选择当前最优解即可。



对于贪心算法而言,

- 设计是容易的, 因为只要选择当前最优解即可。
- 但证明其正确性却是困难的, 因为我们需要证明每次选择最优解的策略最终能够得到全局最优解。



对于贪心算法而言,

- 设计是容易的, 因为只要选择当前最优解即可。
- 但证明其正确性却是困难的, 因为我们需要证明每次选择最优解的策略最终能够得到全局最优解。

对于贪心算法而言,

- 设计是容易的, 因为只要选择当前最优解即可。
- 但证明其正确性却是困难的, 因为我们需要证明每次选择最优解的策略最终能够得到全局最优解。

关于贪心算法

贪心算法不总是一直能保证获取最优解, 但是对于很多问题, 其策略是有效的。

我们先以一个简单的零钱兑换问题来说明关于贪心算法的思想。

我们先以一个简单的零钱兑换问题来说明关于贪心算法的思想。

零钱兑换问题

假设你现在手上有面值分别为 $a_1, \dots, a_n$ ($a_1 < a_2 < \dots < a_n$) 的硬币，现在你需要用这些硬币来凑出一个面值为 M 的钱，问你最少需要多少个硬币？

我们不妨假定 $a_1 = 1$ ，这能保证我们一定能够凑出面值为 M 的钱。



我们先以一个简单的零钱兑换问题来说明关于贪心算法的思想。

零钱兑换问题

假设你现在手上有面值分别为 $a_1, \dots, a_n$ ($a_1 < a_2 < \dots < a_n$) 的硬币，现在你需要用这些硬币来凑出一个面值为 M 的钱，问你最少需要多少个硬币？

我们不妨假定 $a_1 = 1$ ，这能保证我们一定能够凑出面值为 M 的钱。

你的策略是什么？



零钱兑换问题-贪心策略

我们考察如下的一个策略:

我们考察如下的一个策略：

- 每次都选择面值最大的硬币 a_i 。

我们考察如下的一个策略：

- 每次都选择面值最大的硬币 a_i 。

这个策略是否一定能成功？

我们考察如下的一个策略：

- 每次都选择面值最大的硬币 a_i 。

这个策略是否一定能成功？

1. 面值为：1, 2, 5？

我们考察如下的一个策略：

- 每次都选择面值最大的硬币 a_i 。

这个策略是否一定能成功？

1. 面值为：1, 2, 5？
2. 面值为：1, 2, 7, 10？



特殊情况



上海师范大学
Shanghai Normal University

我们能证明, 当面值满足: $1, c, c^2, c^3, \dots, c^{n-1}$ 的时候贪心策略是成功的。

我们能证明，当面值满足： $1, c, c^2, c^3, \dots, c^{n-1}$ 的时候贪心策略是成功的。

证明. 令 $M = (m_1, \dots, m_n)$ 是一个兑换策略，若 M 是一个最优解，则一定有：

$$\text{对任意的 } i \in \{1, \dots, n-1\} \text{ 我们有 } m_i < c \quad (1)$$

我们能证明，当面值满足： $1, c, c^2, c^3, \dots, c^{n-1}$ 的时候贪心策略是成功的。

证明. 令 $M = (m_1, \dots, m_n)$ 是一个兑换策略，若 M 是一个最优解，则一定有：

$$\text{对任意的 } i \in \{1, \dots, n-1\} \text{ 我们有 } m_i < c \quad (1)$$

事实上不妨假设 $m_j \geq c, j \in \{1, \dots, n-1\}$ ，注意到 $c \cdot c^j = c^{j+1}$ ，从而兑换策略 $M' = \{m_1, \dots, m_j - c, m_{j+1} + 1, \dots, m_n\}$ 会是一个更优的策略。

我们能证明，当面值满足: $1, c, c^2, c^3, \dots, c^{n-1}$ 的时候贪心策略是成功的。

证明. 令 $M = (m_1, \dots, m_n)$ 是一个兑换策略，若 M 是一个最优解，则一定有：

$$\text{对任意的 } i \in \{1, \dots, n-1\} \text{ 我们有 } m_i < c \quad (1)$$

事实上不妨假设 $m_j \geq c, j \in \{1, \dots, n-1\}$ ，注意到 $c \cdot c^j = c^{j+1}$ ，从而兑换策略 $M' = \{m_1, \dots, m_j - c, m_{j+1} + 1, \dots, m_n\}$ 会是一个更优的策略。

我们再证明，满足条件 1 的策略是唯一的。反设存在两个兑换策略 $M_1 = (m_{11}, \dots, m_{1n})$ 和 $M_2 = (m_{21}, \dots, m_{2n})$ 都满足条件 1，则我们有：

$$\sum_{i=1}^{n-1} (m_{1i} - m_{2i})c^i = c^n(m_{2n} - m_{1n}) \quad (2)$$

证明续. 若 $m_{2n} - m_{1n} \neq 0$, 不妨令其 ≥ 1 , 则我们有:

$$\sum_{i=1}^{n-1} (m_{1i} - m_{2i})c^i \leq \sum_{i=1}^{n-1} (c-1) \cdot c^i = (c-1) \cdot \frac{c^n - 1}{c-1} = c^n - 1 < c^n \quad (3)$$

与 2 矛盾, 从而 $m_{2n} - m_{1n} = 0$ 。

证明续. 若 $m_{2n} - m_{1n} \neq 0$, 不妨令其 ≥ 1 , 则我们有:

$$\sum_{i=1}^{n-1} (m_{1i} - m_{2i})c^i \leq \sum_{i=1}^{n-1} (c-1) \cdot c^i = (c-1) \cdot \frac{c^n - 1}{c-1} = c^n - 1 < c^n \quad (3)$$

与 2 矛盾, 从而 $m_{2n} - m_{1n} = 0$ 。

以相同的方式从大到小逐一考虑 $m_{1i} - m_{2i}$ 的值, 我们可以最终得到, 等式 2 成立当且仅当 $M_1 = M_2$, 从而满足条件 1 的策略是唯一的。

证明续. 若 $m_{2n} - m_{1n} \neq 0$, 不妨令其 ≥ 1 , 则我们有:

$$\sum_{i=1}^{n-1} (m_{1i} - m_{2i})c^i \leq \sum_{i=1}^{n-1} (c-1) \cdot c^i = (c-1) \cdot \frac{c^n - 1}{c-1} = c^n - 1 < c^n \quad (3)$$

与 2 矛盾, 从而 $m_{2n} - m_{1n} = 0$ 。

以相同的方式从大到小逐一考虑 $m_{1i} - m_{2i}$ 的值, 我们可以最终得到, 等式 2 成立当且仅当 $M_1 = M_2$, 从而满足条件 1 的策略是唯一的。

最后注意到由于贪心策略给出的策略满足条件 1, 从而贪心策略给出了该情况下的最优解。 □

零钱兑换问题的回顾

我们通过零钱兑换问题展示了贪心算法的特点。

我们通过零钱兑换问题展示了贪心算法的特点。

贪心算法

我们通过零钱兑换问题展示了贪心算法的特点。

贪心算法

- 贪心算法的形式是简单的，如在零钱兑换中我们每次尽可能选取最大的零钱就行。

我们通过零钱兑换问题展示了贪心算法的特点。

贪心算法

- 贪心算法的形式是简单的，如在零钱兑换中我们每次尽可能选取最大的零钱就行。
- 贪心策略并不总能成功。

我们通过零钱兑换问题展示了贪心算法的特点。

贪心算法

- 贪心算法的形式是简单的，如在零钱兑换中我们每次尽可能选取最大的零钱就行。
- 贪心策略并不总能成功。
- 贪心策略在很多情况下也可以是成功的，但需要一些正确性的证明。

我们通过零钱兑换问题展示了贪心算法的特点。

贪心算法

- 贪心算法的形式是简单的，如在零钱兑换中我们每次尽可能选取最大的零钱就行。
- 贪心策略并不总能成功。
- 贪心策略在很多情况下也可以是成功的，但需要一些正确性的证明。

我们通过零钱兑换问题展示了贪心算法的特点。

贪心算法

- 贪心算法的形式是简单的，如在零钱兑换中我们每次尽可能选取最大的零钱就行。
- 贪心策略并不总能成功。
- 贪心策略在很多情况下也可以是成功的，但需要一些正确性的证明。

关于更多

1. 要想解决所有情况下的零钱兑换问题，我们需要动态规划的思想。

我们通过零钱兑换问题展示了贪心算法的特点。

贪心算法

- 贪心算法的形式是简单的，如在零钱兑换中我们每次尽可能选取最大的零钱就行。
- 贪心策略并不总能成功。
- 贪心策略在很多情况下也可以是成功的，但需要一些正确性的证明。

关于更多

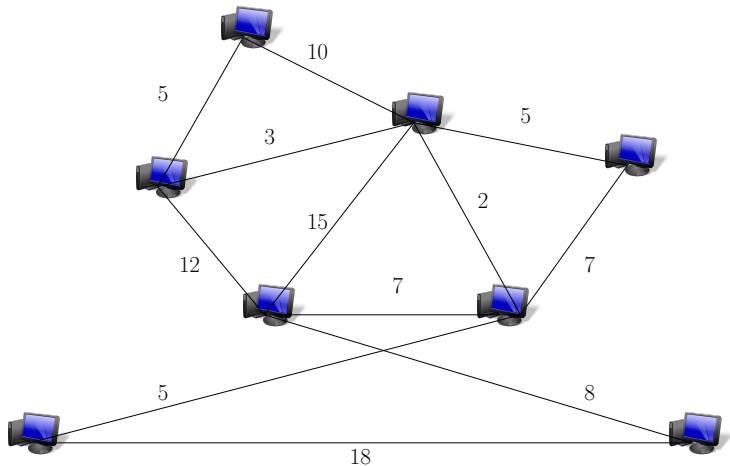
1. 要想解决所有情况下的零钱兑换问题，我们需要动态规划的思想。
2. 虽然贪心的策略并不总能成功，但其实它在近似算法中也有很多的运用。

贪心算法设计

最小生成树-问题提出

假设你现在需要将一组计算机连接起来，你需要在这些计算机之间铺设光纤，每条光纤的长度是不同的，你需要找到一种铺设方案，使得铺设光纤的总长度最小。

假设你现在需要将一组计算机连接起来，你需要在这些计算机之间铺设光纤，每条光纤的长度是不同的，你需要找到一种铺设方案，使得铺设光纤的总长度最小。



我们可以将上述例子抽象成一张有权重的无向图，则问题转化为：

问题 1.

给定一个含权重的无向图 $G = (V, E, \omega)$ ，我们需要选出足够多的边集 T ，使得在其子图 (V, T) 中任何两个顶点之间都有一条路径，且这些边的权重之和最小。

我们可以将上述例子抽象成一张有权重的无向图，则问题转化为：

问题 1.

给定一个含权重的无向图 $G = (V, E, \omega)$ ，我们需要选出足够多的边集 T ，使得在其子图 (V, T) 中任何两个顶点之间都有一条路径，且这些边的权重之和最小。

- 由于满足上述要求的子图一定不会存在一个圈，因此我们实际上求的子图是一棵树。我们称满足上述要求的子图为 G 的**最小生成树**。

我们可以将上述例子抽象成一张有权重的无向图，则问题转化为：

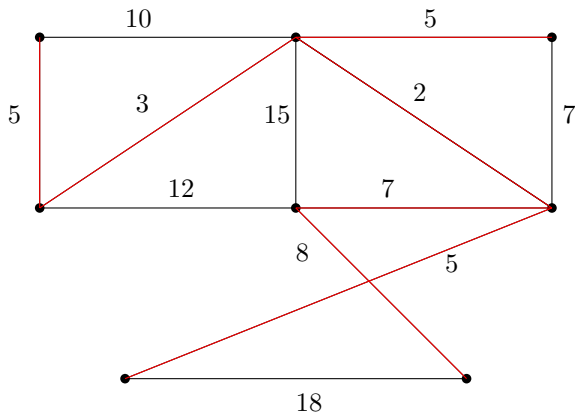
问题 1.

给定一个含权重的无向图 $G = (V, E, \omega)$ ，我们需要选出足够多的边集 T ，使得在其子图 (V, T) 中任何两个顶点之间都有一条路径，且这些边的权重之和最小。

- 由于满足上述要求的子图一定不会存在一个圈，因此我们实际上求的子图是一棵树。我们称满足上述要求的子图为 G 的**最小生成树**。
- 我们默认 G 是连通的，否则我们的要求可以单独作用在每个连通的分图上。

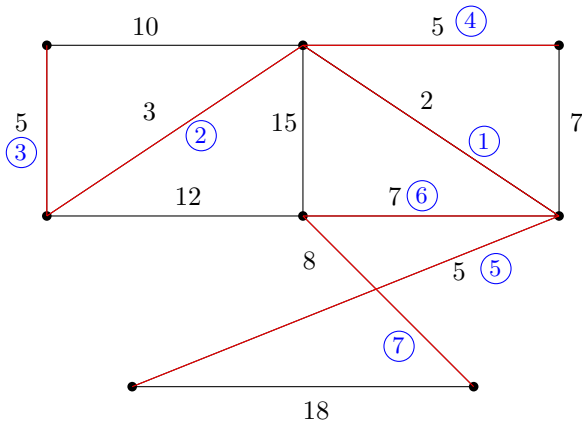
下图给出了上述例子对应的带权重无向图。

- 其中红色边集即为对应的最小生成树。

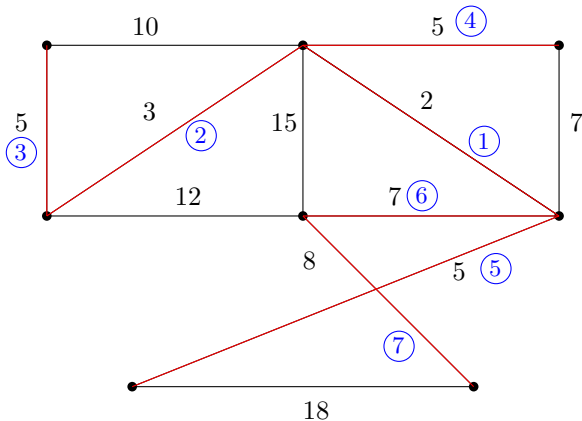


- 既然要求最小权重的生成树，那我每次选取当前权重最小的边，只要保证不成圈就行。

- 既然要求最小权重的生成树，那我每次选取当前权重最小的边，只要保证不成圈就行。



- 既然要求最小权重的生成树，那我每次选取当前权重最小的边，只要保证不成圈就行。



这就是 Kruskal 算法。

算法: Kruskal

输入: 含权连通无向图 $G = (V, E)$, $V = \{1, 2, \dots, n\}$

输出: G 生成的最小生成树所组成的边集 T

- 1: 按非降序的权重 E 进行排序, 得到 $E = \{e_1, e_2, \dots, e_m\}$
- 2: $T = \emptyset$
- 3: **for** $i = 1$ to m **do**
- 4: **if** $T \cup \{e_i\}$ 不成圈 **then**
- 5: $T = T \cup \{e_i\}$
- 6: **return** T

算法: Kruskal

输入: 含权连通无向图 $G = (V, E)$, $V = \{1, 2, \dots, n\}$

输出: G 生成的最小生成树所组成的边集 T

- 1: 按非降序的权重 E 进行排序, 得到 $E = \{e_1, e_2, \dots, e_m\}$
- 2: $T = \emptyset$
- 3: **for** $i = 1$ **to** m **do**
- 4: **if** $T \cup \{e_i\}$ 不成圈 **then**
- 5: $T = T \cup \{e_i\}$
- 6: **return** T

补充说明

判断 $T \cup \{e_i\}$ 是否成圈可以使用并查集, 即之前讲过的 UnionFind 数据结构。

定理 2.

Kruskal 算法能够正确的求出最小生成树。

证明. 令算法找到的生成树为 T ，其边的加入顺序为 $\{e_1, e_2, \dots, e_{n-1}\}$ 。设 T^* 为 G 的最小生成树，我们来证明 $\omega(T) = \omega(T^*)$ 。

定理 2.

Kruskal 算法能够正确的求出最小生成树。

证明. 令算法找到的生成树为 T , 其边的加入顺序为 $\{e_1, e_2, \dots, e_{n-1}\}$ 。设 T^* 为 G 的最小生成树, 我们来证明 $\omega(T) = \omega(T^*)$ 。

将 T^* 的边也按权重从小到大排列为 $\{e'_1, e'_2, \dots, e'_{n-1}\}$ 。我们对 k 归纳证明:

$$\omega(e_1) + \dots + \omega(e_k) = \omega(e'_1) + \dots + \omega(e'_k)$$

定理 2.

Kruskal 算法能够正确的求出最小生成树。

证明. 令算法找到的生成树为 T , 其边的加入顺序为 $\{e_1, e_2, \dots, e_{n-1}\}$ 。设 T^* 为 G 的最小生成树, 我们来证明 $\omega(T) = \omega(T^*)$ 。

将 T^* 的边也按权重从小到大排列为 $\{e'_1, e'_2, \dots, e'_{n-1}\}$ 。我们对 k 归纳证明:

$$\omega(e_1) + \dots + \omega(e_k) = \omega(e'_1) + \dots + \omega(e'_k)$$

初始情况是 $k = 0$, 显然成立。假设 $\leq k - 1$ 成立, 考察 k 时的情况, 我们有:

定理 2.

Kruskal 算法能够正确的求出最小生成树。

证明. 令算法找到的生成树为 T , 其边的加入顺序为 $\{e_1, e_2, \dots, e_{n-1}\}$ 。设 T^* 为 G 的最小生成树, 我们来证明 $\omega(T) = \omega(T^*)$ 。

将 T^* 的边也按权重从小到大排列为 $\{e'_1, e'_2, \dots, e'_{n-1}\}$ 。我们对 k 归纳证明:

$$\omega(e_1) + \dots + \omega(e_k) = \omega(e'_1) + \dots + \omega(e'_k)$$

初始情况是 $k = 0$, 显然成立。假设 $\leq k - 1$ 成立, 考察 k 时的情况, 我们有:

1. $\omega(e_k) \leq \omega(e'_k)$ 。这是由算法保证的, 算法要求选择此时权重最小的边。

定理 2.

Kruskal 算法能够正确的求出最小生成树。

证明. 令算法找到的生成树为 T , 其边的加入顺序为 $\{e_1, e_2, \dots, e_{n-1}\}$ 。设 T^* 为 G 的最小生成树, 我们来证明 $\omega(T) = \omega(T^*)$ 。

将 T^* 的边也按权重从小到大排列为 $\{e'_1, e'_2, \dots, e'_{n-1}\}$ 。我们对 k 归纳证明:

$$\omega(e_1) + \dots + \omega(e_k) = \omega(e'_1) + \dots + \omega(e'_k)$$

初始情况是 $k = 0$, 显然成立。假设 $\leq k - 1$ 成立, 考察 k 时的情况, 我们有:

1. $\omega(e_k) \leq \omega(e'_k)$ 。这是由算法保证的, 算法要求选择此时权重最小的边。
2. $\omega(e_k) \geq \omega(e'_k)$ 。反设其不成立, 即 $\omega(e_k) < \omega(e'_k)$, 则考察边集 $T^* \cup \{e_k\}$, 其一定包含一个包含 e_k 的圈, 从而存在 e'_i 使得 $T' = T^* \cup \{e_k\} - e'_i$ 依旧是图 G 的生成树, 然而 $\omega(T') < \omega(T^*)$, 与 T^* 的最小性矛盾,

定理 2.

Kruskal 算法能够正确的求出最小生成树。

证明. 令算法找到的生成树为 T , 其边的加入顺序为 $\{e_1, e_2, \dots, e_{n-1}\}$ 。设 T^* 为 G 的最小生成树, 我们来证明 $\omega(T) = \omega(T^*)$ 。

将 T^* 的边也按权重从小到大排列为 $\{e'_1, e'_2, \dots, e'_{n-1}\}$ 。我们对 k 归纳证明:

$$\omega(e_1) + \dots + \omega(e_k) = \omega(e'_1) + \dots + \omega(e'_k)$$

初始情况是 $k = 0$, 显然成立。假设 $\leq k - 1$ 成立, 考察 k 时的情况, 我们有:

1. $\omega(e_k) \leq \omega(e'_k)$ 。这是由算法保证的, 算法要求选择此时权重最小的边。
2. $\omega(e_k) \geq \omega(e'_k)$ 。反设其不成立, 即 $\omega(e_k) < \omega(e'_k)$, 则考察边集 $T^* \cup \{e_k\}$, 其一定包含一个包含 e_k 的圈, 从而存在 e'_i 使得 $T' = T^* \cup \{e_k\} - e'_i$ 依旧是图 G 的生成树, 然而 $\omega(T') < \omega(T^*)$, 与 T^* 的最小性矛盾,

定理 2.

Kruskal 算法能够正确的求出最小生成树。

证明. 令算法找到的生成树为 T , 其边的加入顺序为 $\{e_1, e_2, \dots, e_{n-1}\}$ 。设 T^* 为 G 的最小生成树, 我们来证明 $\omega(T) = \omega(T^*)$ 。

将 T^* 的边也按权重从小到大排列为 $\{e'_1, e'_2, \dots, e'_{n-1}\}$ 。我们对 k 归纳证明:

$$\omega(e_1) + \dots + \omega(e_k) = \omega(e'_1) + \dots + \omega(e'_k)$$

初始情况是 $k = 0$, 显然成立。假设 $\leq k - 1$ 成立, 考察 k 时的情况, 我们有:

1. $\omega(e_k) \leq \omega(e'_k)$ 。这是由算法保证的, 算法要求选择此时权重最小的边。
2. $\omega(e_k) \geq \omega(e'_k)$ 。反设其不成立, 即 $\omega(e_k) < \omega(e'_k)$, 则考察边集 $T^* \cup \{e_k\}$, 其一定包含一个包含 e_k 的圈, 从而存在 e'_i 使得 $T' = T^* \cup \{e_k\} - e'_i$ 依旧是图 G 的生成树, 然而 $\omega(T') < \omega(T^*)$, 与 T^* 的最小性矛盾,

从而我们有 $\omega(e_k) = \omega(e'_k)$, 因此命题对 k 也成立, 即 $\omega(T) = \omega(T^*)$ 。

□

► Kruskal 算法的时间分析



上海师范大学
Shanghai Normal University

Kruskal 算法的时间分析

- Kruskal 算法对边排序需要 $O(m \log m)$ 的时间。

- Kruskal 算法对边排序需要 $O(m \log m)$ 的时间。
- 判断是否有圈可以利用并查集，一共至多执行 $2m$ 次 Find 操作和 $n - 1$ 次 Union 操作，因此总共耗费 $O(m \log * n)$ 的时间。

- Kruskal 算法对边排序需要 $O(m \log m)$ 的时间。
- 判断是否有圈可以利用并查集，一共至多执行 $2m$ 次 Find 操作和 $n - 1$ 次 Union 操作，因此总共耗费 $O(m \log * n)$ 的时间。
- 一共会往 T 里增加 $n - 1$ 条边。

- Kruskal 算法对边排序需要 $O(m \log m)$ 的时间。
- 判断是否有圈可以利用并查集，一共至多执行 $2m$ 次 Find 操作和 $n - 1$ 次 Union 操作，因此总共耗费 $O(m \log * n)$ 的时间。
- 一共会往 T 里增加 $n - 1$ 条边。

- Kruskal 算法对边排序需要 $O(m \log m)$ 的时间。
- 判断是否有圈可以利用并查集，一共至多执行 $2m$ 次 Find 操作和 $n - 1$ 次 Union 操作，因此总共耗费 $O(m \log^* n)$ 的时间。
- 一共会往 T 里增加 $n - 1$ 条边。

因此，算法的运行时间为 $O(m \log m) = O(m \log n)$ 。

定理 3.

Kruskal 算法可以在 $O(m \log n)$ 内求出 G 的最小生成树。

► Kruskal 算法正确的另一视角-分割性质

我们换个角度再来理解下 Kruskal 算法的正确性。

Kruskal 算法正确的另一视角-分割性质

我们换个角度再来理解下 Kruskal 算法的正确性。

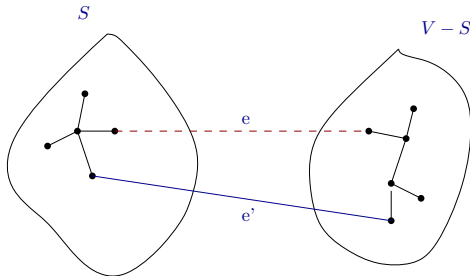
假设为了构造最小生成树，我们已经选择了一些边，这些边将图上的顶点划分成了若干个部分，下面性质说明，跨越这些部分中的最短边也是某个最小生成树的一部分。

我们换个角度再来理解下 Kruskal 算法的正确性。

假设为了构造最小生成树，我们已经选择了一些边，这些边将图上的顶点划分成了若干个部分，下面性质说明，跨越这些部分中的最短边也是某个最小生成树的一部分。

分割性质

设 $G = (V, E, \omega)$ 是一个含权重的连通无向图， X 是 G 的某个最小生成树的一部分，令 S 是 V 的一个子集，满足 X 中没有横跨 S 和 $V - S$ 的边，设 e 是 G 中连接 T 中的一个顶点和 $V - T$ 中的一个顶点的最短边，则 $X \cup \{e\}$ 是 G 的某个最小生成树的一部分。



算法: Prim

输入: 含权连通无向图 $G = (V, E)$, $V = \{1, 2, \dots, n\}$

输出: G 生成的最小生成树所组成的边集 T

```
1:  $T = \emptyset$ ,  $X = \{1\}$ ,  $Y = \{V\} - \{1\}$ 
2: for  $y \leftarrow 2$  to  $n$  do
3:   if  $(1, y) \in E$  then
4:      $n(y) = 1$ 
5:      $c(y) \leftarrow \omega(1, y)$ 
6:   else  $c(y) \leftarrow \infty$ 
7: for  $j \leftarrow 1$  to  $n - 1$  do
8:   从  $Y$  中选取  $w(y)$  最小的点  $u$ 
9:    $T = T \cup \{(u, n(u))\}$ 
10:   $X = X \cup \{u\}$ ,  $Y = Y - \{u\}$ 
11:  for  $w \in Y \wedge (y, w) \in E$  do
12:    if  $\omega(y, w) < c(w)$  then
13:       $n(w) \leftarrow y$ ,  $c(w) \leftarrow \omega(y, w)$ 
14: return  $T$ 
```

▷ $n(y)$ 记录当前最短边的另一端点

▷ $c(y)$ 记录当前最短边的权重

- 由分割性质, Prim 算法的正确性是显然的。

- 由分割性质，Prim 算法的正确性是显然的。
- Prim 算法与 Dijkstra 算法的流程基本相同，所以其复杂性是一样的，取决于优先队列的实现。

- 由分割性质，Prim 算法的正确性是显然的。
- Prim 算法与 Dijkstra 算法的流程基本相同，所以其复杂性是一样的，取决于优先队列的实现。
 - 如果使用普通数组，时间为 $O(n^2)$ 。

- 由分割性质，Prim 算法的正确性是显然的。
- Prim 算法与 Dijkstra 算法的流程基本相同，所以其复杂性是一样的，取决于优先队列的实现。
 - 如果使用普通数组，时间为 $O(n^2)$.
 - 如果使用二叉堆，时间为 $O(m \log n)$.

- 由分割性质，Prim 算法的正确性是显然的。
- Prim 算法与 Dijkstra 算法的流程基本相同，所以其复杂性是一样的，取决于优先队列的实现。
 - 如果使用普通数组，时间为 $O(n^2)$.
 - 如果使用二叉堆，时间为 $O(m \log n)$.

- 由分割性质, Prim 算法的正确性是显然的。
- Prim 算法与 Dijkstra 算法的流程基本相同, 所以其复杂性是一样的, 取决于优先队列的实现。
 - 如果使用普通数组, 时间为 $O(n^2)$.
 - 如果使用二叉堆, 时间为 $O(m \log n)$.

定理 4.

使用二分堆作为优先队列的实现时, Prim 算法可以在 $O(m \log n)$ 内求出 G 的最小生成树。

我们现在来介绍另一个问题-文件压缩。

问题 5.

假设现在有一个字符型文件，我们希望将其尽可能的压缩文件，但能很容易的重建文件。我们知道的信息有，文件中一共有 n 个字符，分别为 $\{c_1, \dots, c_n\}$ ，每个字符出现的次数为 $f(c_1), f(c_2), \dots, f(c_n)$ 。我们的目标是找到一种压缩方式 τ ，令该压缩方式下， c_i 转换成的字符长度为 $\tau(c_i)$ ，使得最后文件的总长度最小，即最小化： $\sum_{i=1}^n f(c_i) \cdot \tau(c_i)$ 的值。

我们现在来介绍另一个问题-文件压缩。

问题 5.

假设现在有一个字符型文件，我们希望将其尽可能的压缩文件，但能很容易的重建文件。我们知道的信息有，文件中一共有 n 个字符，分别为 $\{c_1, \dots, c_n\}$ ，每个字符出现的次数为 $f(c_1), f(c_2), \dots, f(c_n)$ 。我们的目标是找到一种压缩方式 τ ，令该压缩方式下， c_i 转换成的字符长度为 $\tau(c_i)$ ，使得最后文件的总长度最小，即最小化： $\sum_{i=1}^n f(c_i) \cdot \tau(c_i)$ 的值。

定长压缩

一个很自然的方法是定长编码压缩，比如假设一共有 $n = 2^k$ 个不同的编码，则我们可以使用 k 位的 01 串来编码每个字符。比如如果文章有 4 个不同的字符 $\{A, B, C, D\}$ ，则我们可以使用 00, 01, 10, 11 去表示，



定长编码似乎非常有道理，但我们考虑下面这个情况，两个文件由 4 个字符 {A, B, C, D} 组成，但其出现次数分别是：

- $f(A) = 25, f(B) = 25, f(C) = 25, f(D) = 25.$
- $f(A) = 1, f(B) = 1, f(C) = 1, f(D) = 97.$

定长编码似乎非常有道理，但我们考虑下面这个情况，两个文件由 4 个字符 {A, B, C, D} 组成，但其出现次数分别是：

- $f(A) = 25, f(B) = 25, f(C) = 25, f(D) = 25$.
- $f(A) = 1, f(B) = 1, f(C) = 1, f(D) = 97$.

两种情况此时都会用一个 400 位的 01 串表示。但对于第二种情况，是否可能还存在更简单的编码？

定长编码似乎非常有道理，但我们考虑下面这个情况，两个文件由 4 个字符 {A, B, C, D} 组成，但其出现次数分别是：

- $f(A) = 25, f(B) = 25, f(C) = 25, f(D) = 25$.
- $f(A) = 1, f(B) = 1, f(C) = 1, f(D) = 97$.

两种情况此时都会用一个 400 位的 01 串表示。但对于第二种情况，是否可能还存在更简单的编码？

如果我们令：

$$A : 100, B : 101, C : 11, D : 0$$

用这种编码的话，第二个文件只需要 105 位的 01 串就可以表达了。

定长编码似乎非常有道理，但我们考虑下面这个情况，两个文件由 4 个字符 {A, B, C, D} 组成，但其出现次数分别是：

- $f(A) = 25, f(B) = 25, f(C) = 25, f(D) = 25$.
- $f(A) = 1, f(B) = 1, f(C) = 1, f(D) = 97$.

两种情况此时都会用一个 400 位的 01 串表示。但对于第二种情况，是否可能还存在更简单的编码？

如果我们令：

$$A : 100, B : 101, C : 11, D : 0$$

用这种编码的话，第二个文件只需要 105 位的 01 串就可以表达了。

通过选择合适的变长编码可以减小文件表示的数目！

变长的编码有一个问题，就是可能会出现二义性。假设某个文件中的 a, b, c 分别用如下编码：

$a : 10, b, 100, : 0$

变长的编码有一个问题，就是可能会出现二义性。假设某个文件中的 a, b, c 分别用如下编码：

$a : 10, b, 100, : 0$

那么对于字符串 100100100:

- 其想表达的是 bbb?

变长的编码有一个问题，就是可能会出现二义性。假设某个文件中的 a, b, c 分别用如下编码：

$a : 10, b, 100, : 0$

那么对于字符串 100100100:

- 其想表达的是 bbb?
- 还是 acbb?

变长的编码有一个问题，就是可能会出现二义性。假设某个文件中的 a, b, c 分别用如下编码：

$a : 10, b, 100, : 0$

那么对于字符串 100100100:

- 其想表达的是 bbb?
- 还是 acbb?
- ...

变长的编码有一个问题，就是可能会出现二义性。假设某个文件中的 a, b, c 分别用如下编码：

$a : 10, b, 100, : 0$

那么对于字符串 100100100:

- 其想表达的是 bbb?
- 还是 acbb?
- ...

变长的编码有一个问题，就是可能会出现二义性。假设某个文件中的 a, b, c 分别用如下编码：

$a : 10, b, 100, : 0$

那么对于字符串 100100100:

- 其想表达的是 bbb?
- 还是 acbb?
- ...

为了避免歧义，我们引入前缀码的概念，即任何字符的编码都不是其他字符编码的前缀。

定义 6.

如果一个编码满足前缀码的性质，即任何一个字符的编码不会是其其他某个字符编码的前缀，则称其为**前缀码**。



例 7.

考察下面堆字符的一个编码：

$! : 101, A : 11, B : 00, C : 010, D : 100, R : 100$

其是一个前缀码，对于任何一个由其编码的字符串，其意义是唯一的。

例 7.

考察下面堆字符的一个编码：

! : 101, A : 11, B : 00, C : 010, D : 100, R : 100

其是一个前缀码，对于任何一个由其编码的字符串，其意义是唯一的。

- 1100011110101110011000111101 对应的字符串为 ABRACADABRA!.

例 7.

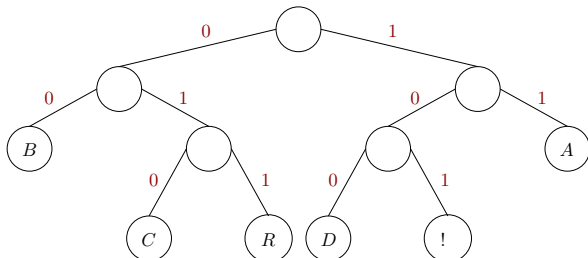
考察下面堆字符的一个编码：

! : 101, A : 11, B : 00, C : 010, D : 100, R : 100

其是一个前缀码，对于任何一个由其编码的字符串，其意义是唯一的。

- 11000111101011100110001111101 对应的字符串为 ABRACADABRA!.

前缀码可以由一棵二叉树来表示，比如上面的例子对应的二叉树为：



接下来我们介绍一种构造前缀码的方法，即 Huffman 编码。

接下来我们介绍一种构造前缀码的方法，即 Huffman 编码。



David Huffman



Robert Fano



Claude Shannon

接下来我们介绍一种构造前缀码的方法，即 Huffman 编码。



David Huffman



Robert Fano



Claude Shannon

- 其直观的思想是，我们希望出现次数多的字符编码尽可能的短。

接下来我们介绍一种构造前缀码的方法，即 Huffman 编码。



David Huffman



Robert Fano



Claude Shannon

- 其直观的思想是，我们希望出现次数多的字符编码尽可能的短。
- 对于前缀码，所有字符都是对应在叶子节点上的，因此算法优先选择出现次数少的字符，将其合并成父节点。

算法: Huffman

输入: 一个 n 个字符的集合 $C = \{c_1, \dots, c_n\}$ 和其字符对应出现的频度: $\{f(c_1), \dots, f(c_n)\}$

输出: C 的一个 Huffman 编码对应的树 (V, T)

- 1: 根据频度将所有字符插入最小堆 H
- 2: $V \leftarrow C, T = \emptyset$
- 3: **for** $i = 1$ to $n - 1$ **do**
- 4: $c_1 \leftarrow \text{DeleteMin}(H)$
- 5: $c_2 \leftarrow \text{DeleteMin}(H)$
- 6: $f(v) \leftarrow f(c_1) + f(c_2)$ ▷ v 是一个构造出来的 c_1, c_2 的父节点
- 7: $\text{Insert}(H, v)$
- 8: $V \leftarrow V \cup \{v\}$
- 9: $T \leftarrow T \cup \{(v, c_1), (v, c_2)\}$
- 10: **return** (V, T)

算法: Huffman

输入: 一个 n 个字符的集合 $C = \{c_1, \dots, c_n\}$ 和其字符对应出现的频度: $\{f(c_1), \dots, f(c_n)\}$

输出: C 的一个 Huffman 编码对应的树 (V, T)

- 1: 根据频度将所有字符插入最小堆 H
- 2: $V \leftarrow C, T = \emptyset$
- 3: **for** $i = 1$ to $n - 1$ **do**
- 4: $c_1 \leftarrow \text{DeleteMin}(H)$
- 5: $c_2 \leftarrow \text{DeleteMin}(H)$
- 6: $f(v) \leftarrow f(c_1) + f(c_2)$ ▷ v 是一个构造出来的 c_1, c_2 的父节点
- 7: $\text{Insert}(H, v)$
- 8: $V \leftarrow V \cup \{v\}$
- 9: $T \leftarrow T \cup \{(v, c_1), (v, c_2)\}$
- 10: **return** (V, T)

时间复杂性: $O(n \log n)!$

Huffman 编码的效果



上海师范大学
Shanghai Normal University

	a	b	c	d	e	f
频度	45	13	12	16	9	5
定长编码	000	001	010	011	100	101
Huffman 编码	0	101	100	111	1101	1100

	a	b	c	d	e	f
频度	45	13	12	16	9	5
定长编码	000	001	010	011	100	101
Huffman 编码	0	101	100	111	1101	1100

假设一共有 100000 个字符。

	a	b	c	d	e	f
频度	45	13	12	16	9	5
定长编码	000	001	010	011	100	101
Huffman 编码	0	101	100	111	1101	1100

假设一共有 100000 个字符。

- 定长编码需要 300000 位。

	a	b	c	d	e	f
频度	45	13	12	16	9	5
定长编码	000	001	010	011	100	101
Huffman 编码	0	101	100	111	1101	1100

假设一共有 100000 个字符。

- 定长编码需要 300000 位。
- Huffman 编码需要 224000 位。

	a	b	c	d	e	f
频度	45	13	12	16	9	5
定长编码	000	001	010	011	100	101
Huffman 编码	0	101	100	111	1101	1100

假设一共有 100000 个字符。

- 定长编码需要 300000 位。
- Huffman 编码需要 224000 位。

	a	b	c	d	e	f
频度	45	13	12	16	9	5
定长编码	000	001	010	011	100	101
Huffman 编码	0	101	100	111	1101	1100

假设一共有 100000 个字符。

- 定长编码需要 300000 位。
- Huffman 编码需要 224000 位。

Huffman 编码是压缩率最高的无损编码。

引理 8.

令 C 是一个字母表。对其中每个字符 $c \in C$, $f(c)$ 为其频率。令 x, y 是其频率最低的两个字符, 则存在一个 C 的最优前缀码, 使得 x, y 的编码字符长度相同, 且只差最后一个二进制不相同。

引理 8.

令 C 是一个字母表。对其中每个字符 $c \in C$, $f(c)$ 为其频率。令 x, y 是其频率最低的两个字符, 则存在一个 C 的最优前缀码, 使得 x, y 的编码字符长度相同, 且只差最后一个二进制不相同。

引理 9.

令 C 是一个字母表。对其中每个字符 $c \in C$, $f(c)$ 为其频率。令 x, y 是其频率最低的两个字符。令 C' 是字母表 C 去掉 x, y 加入一个新的字符 z 后得到的字母表。 C' 也定义了其字符的频率 f' , $f'(c)$ 与 $f(c)$ 相同, 除了定义 $f'(z) = f(x) + f(y)$ 。则对于 C' 的一个最优前缀码对应的编码树 T , 将其中代表 z 的叶子节点替换成一个以 x, y 为孩子的内部节点得到新的树 T' , 则 T' 是 C 的一个最优前缀码对应的编码树。

引理 8.

令 C 是一个字母表。对其中每个字符 $c \in C$, $f(c)$ 为其频率。令 x, y 是其频率最低的两个字符, 则存在一个 C 的最优前缀码, 使得 x, y 的编码字符长度相同, 且只差最后一个二进制不相同。

引理 9.

令 C 是一个字母表。对其中每个字符 $c \in C$, $f(c)$ 为其频率。令 x, y 是其频率最低的两个字符。令 C' 是字母表 C 去掉 x, y 加入一个新的字符 z 后得到的字母表。 C' 也定义了其字符的频率 f' , $f'(c)$ 与 $f(c)$ 相同, 除了定义 $f'(z) = f(x) + f(y)$ 。则对于 C' 的一个最优前缀码对应的编码树 T , 将其中代表 z 的叶子节点替换成一个以 x, y 为孩子的内部节点得到新的树 T' , 则 T' 是 C 的一个最优前缀码对应的编码树。

定理 10.

Huffman 算法可以正确的构造出最优前缀码。

最后一个例子则是调度竞争共享资源的活动选择问题。

问题 11

[活动选择问题].

假设有 n 个活动的集合 $S = \{a_1, \dots, a_n\}$ 。这些活动使用同一个资源，而这个资源在某个时刻只能供一个活动使用。每个活动 a_i 都有一个开始时间 s_i 和结束时间 f_i ，且满足 $0 \leq s_i < f_i < +\infty$ 。如果两个活动 a_i, a_j 的时间没有重叠，也就是说有 $s_i \geq f_j$ 或者 $s_j \geq f_i$ ，则称它们是兼容的。问题是：我们希望选出一个最大兼容活动集合。

最后一个例子则是调度竞争共享资源的活动选择问题。

问题 11

[活动选择问题].

假设有 n 个活动的集合 $S = \{a_1, \dots, a_n\}$ 。这些活动使用同一个资源，而这个资源在某个时刻只能供一个活动使用。每个活动 a_i 都有一个开始时间 s_i 和结束时间 f_i ，且满足 $0 \leq s_i < f_i < +\infty$ 。如果两个活动 a_i, a_j 的时间没有重叠，也就是说有 $s_i \geq f_j$ 或者 $s_j \geq f_i$ ，则称它们是兼容的。问题是：我们希望选出一个最大兼容活动集合。

补充说明

为了方便起见，我们假设 a_i 是按结束时间单调递增排序的，即 $f_1 \leq f_2 \leq \dots \leq f_n$ 。

例 12.

考察下面一个例子：

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

例 12.

考察下面一个例子:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- $\{a_3, a_9, a_{10}\}$ 是兼容的。

例 12.

考察下面一个例子:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- $\{a_3, a_9, a_{10}\}$ 是兼容的。
- $\{a_1, a_2, a_8, a_{11}\}$ 是不兼容的。

例 12.

考察下面一个例子:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- $\{a_3, a_9, a_{10}\}$ 是兼容的。
- $\{a_1, a_2, a_8, a_{11}\}$ 是不兼容的。
- 一个最大兼容活动集合为 $\{a_1, a_4, a_8, a_{11}\}$, 另一个为 $\{a_2, a_4, a_9, a_{10}\}$.

活动选择问题-贪心策略



上海师范大学
Shanghai Normal University

- 既然要求最大兼容活动集合，那我每次选取当前结束时间最早的活动就行。

- 既然要求最大兼容活动集合，那我每次选取当前结束时间最早的活动就行。

算法 ActivitySelector(S, f)

输入: 活动集合 $S = \{a_1, \dots, a_n\}$, 按结束时间单调递增排序, 即 $f_1 \leq f_2 \leq \dots \leq f_n$

输出: S 的一个最大兼容活动子集 A

```
1:  $A = \{a_1\}$ 
2:  $k = 1$ 
3: for  $i = 2$  to  $n$  do
4:   if  $s_i \geq f_k$  then
5:      $A = A \cup \{a_i\}$ 
6:      $k = i$ 
7: return  $A$ 
```

- 既然要求最大兼容活动集合，那我每次选取当前结束时间最早的活动就行。

算法 ActivitySelector(S, f)

输入: 活动集合 $S = \{a_1, \dots, a_n\}$, 按结束时间单调递增排序, 即 $f_1 \leq f_2 \leq \dots \leq f_n$

输出: S 的一个最大兼容活动子集 A

```
1:  $A = \{a_1\}$ 
2:  $k = 1$ 
3: for  $i = 2$  to  $n$  do
4:   if  $s_i \geq f_k$  then
5:      $A = A \cup \{a_i\}$ 
6:      $k = i$ 
7: return  $A$ 
```

时间复杂性: $O(n)$!

引理 13.

令 S_k 表示在 a_k 结束后开始的任務集合。則我們有，若 a_{m_k} 是 S_k 中結束時間最早的活動，則 a_{m_k} 在 S_k 的某個最大兼容活動集中。

引理 13.

令 S_k 表示在 a_k 结束后开始的任務集合。則我們有，若 a_{m_k} 是 S_k 中結束時間最早的活動，則 a_{m_k} 在 S_k 的某個最大兼容活動集中。

证明. 令 A_k 是 S_k 的某個最大兼容活動集，並且 a_{l_k} 是 A_k 中結束時間最早的任务。

引理 13.

令 S_k 表示在 a_k 结束后开始的任務集合。則我們有，若 a_{m_k} 是 S_k 中結束時間最早的活動，則 a_{m_k} 在 S_k 的某個最大兼容活動集中。

证明. 令 A_k 是 S_k 的某個最大兼容活動集，並且 a_{l_k} 是 A_k 中結束時間最早的任務。

- 如果 $a_{m_k} = a_{l_k}$ ，顯然 $a_{m_k} \in A_k$ ，引理成立。

引理 13.

令 S_k 表示在 a_k 结束后开始的任務集合。則我們有，若 a_{m_k} 是 S_k 中結束時間最早的活動，則 a_{m_k} 在 S_k 的某個最大兼容活動集中。

证明. 令 A_k 是 S_k 的某個最大兼容活動集，並且 a_{l_k} 是 A_k 中結束時間最早的任務。

- 如果 $a_{m_k} = a_{l_k}$ ，顯然 $a_{m_k} \in A_k$ ，引理成立。
- 如果 $a_{m_k} \neq a_{l_k}$ ，則考慮 $A'_k = (A_k - \{a_{l_k}\}) \cup \{a_{m_k}\}$ ，我們有：

$$\forall a, b \in A'_k \Rightarrow f_a \leq s_b \vee f_b \leq s_a$$

引理 13.

令 S_k 表示在 a_k 结束后开始的任務集合。則我們有，若 a_{m_k} 是 S_k 中結束時間最早的活動，則 a_{m_k} 在 S_k 的某個最大兼容活動集中。

证明. 令 A_k 是 S_k 的某個最大兼容活動集，並且 a_{l_k} 是 A_k 中結束時間最早的任務。

- 如果 $a_{m_k} = a_{l_k}$ ，顯然 $a_{m_k} \in A_k$ ，引理成立。
- 如果 $a_{m_k} \neq a_{l_k}$ ，則考慮 $A'_k = (A_k - \{a_{l_k}\}) \cup \{a_{m_k}\}$ ，我們有：

$$\forall a, b \in A'_k \Rightarrow f_a \leq s_b \vee f_b \leq s_a$$

引理 13.

令 S_k 表示在 a_k 结束后开始的任務集合。則我們有，若 a_{m_k} 是 S_k 中結束時間最早的活動，則 a_{m_k} 在 S_k 的某個最大兼容活動集中。

证明. 令 A_k 是 S_k 的某個最大兼容活動集，並且 a_{l_k} 是 A_k 中結束時間最早的任務。

- 如果 $a_{m_k} = a_{l_k}$ ，顯然 $a_{m_k} \in A_k$ ，引理成立。
- 如果 $a_{m_k} \neq a_{l_k}$ ，則考慮 $A'_k = (A_k - \{a_{l_k}\}) \cup \{a_{m_k}\}$ ，我們有：

$$\forall a, b \in A'_k \Rightarrow f_a \leq s_b \vee f_b \leq s_a$$

從而 A'_k 也是 S_k 的某個最大兼容活動子集。



引理 13.

令 S_k 表示在 a_k 结束后开始的任務集合。則我們有，若 a_{m_k} 是 S_k 中結束時間最早的活動，則 a_{m_k} 在 S_k 的某個最大兼容活動集中。

證明. 令 A_k 是 S_k 的某個最大兼容活動集，並且 a_{l_k} 是 A_k 中結束時間最早的任務。

- 如果 $a_{m_k} = a_{l_k}$ ，顯然 $a_{m_k} \in A_k$ ，引理成立。
- 如果 $a_{m_k} \neq a_{l_k}$ ，則考慮 $A'_k = (A_k - \{a_{l_k}\}) \cup \{a_{m_k}\}$ ，我們有：

$$\forall a, b \in A'_k \Rightarrow f_a \leq s_b \vee f_b \leq s_a$$

從而 A'_k 也是 S_k 的某個最大兼容活動子集。



定理 14.

算法 ActivitySelector(S, f) 可以正確的求出最大兼容活動子集。

本节内容

- 贪心算法
 - 什么是贪心算法
 - 贪心算法的特点
- 贪心算法设计举例
 - 最小生成树-Kruskal 算法和 Prim 算法。
 - 文件压缩-Huffman 编码。
 - 活动选择问题。

本节内容

- 贪心算法
 - 什么是贪心算法
 - 贪心算法的特点
- 贪心算法设计举例
 - 最小生成树-Kruskal 算法和 Prim 算法。
 - 文件压缩-Huffman 编码。
 - 活动选择问题。

第八周作业

本周作业和编程作业都已经发布到课程主页上。

截至时间：2023 年 11 月 6 日周一晚上 11:59。