贪心算法

陈长建

计算机科学系

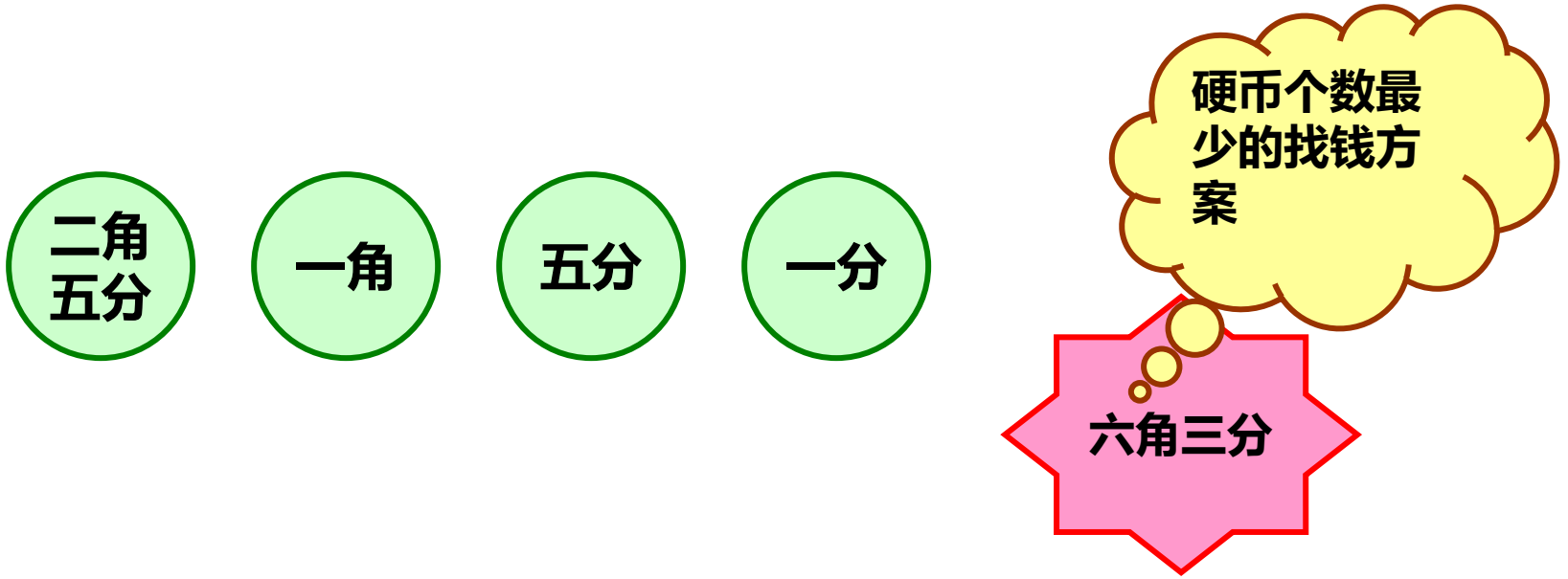
第4章 贪心算法

学习要点

- 理解贪心算法的概念。
- 掌握贪心算法的基本要素：
  - 最优子结构性质、贪心选择性质。
- 理解贪心算法与动态规划算法的差异。
- 通过应用范例学习动态规划算法设计策略：
  - 活动安排问题；背包问题；哈夫曼编码问题；单源最短路劲问题；最小生成树问题。

贪心算法应用举例——找硬币

- 假设有四种硬币，它们的面值分别为五角、一角、五分和一分。现找顾客六角三分钱，请给出硬币个数最少的找钱方案。



分析

- 找硬币问题本身具有最优子结构性质，它可以用动态规划算法来解。但显然贪心算法更简单，更直接且解题效率更高。
- 该算法利用了问题本身的一些特性，例如硬币面值的特性。
- 再例：如果有一分、五分和一角三种不同的硬币，要找给顾客一角五分钱。

结论

- 顾名思义，贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。
- 当然，希望贪心算法得到的最终结果也是整体最优的。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。如单源最短路径问题，最小生成树问题等。
- 在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

贪心法的基本思路

- 从问题的某一个初始解出发逐步逼近给定的目标，以尽可能快地求得更好的解。当达到某算法中的某一步不能再继续前进时，算法停止。
- 实现该算法的过程：

从问题的某一初始

while 能朝给定

求出可行解

由所有解元素组合成问题的一个可行解。

该算法存在的问题：

1. 不能保证求得的最后解是最佳的；

2. 不能用来求最大或最小解问题；

3. 只能求满足某些约束条件的可行解的范围。

例1 - 活动安排问题

- 活动安排问题：要求高效地安排一系列争用某一公共资源的活动。
- 举例：

活动序号	1	2	3	4	5	6	7	8	9	10	11
起始时间	1	3	0	5	3	5	6	8	8	2	12
结束时间	4	5	6	7	8	9	10	11	12	13	14

问题定义

- 设有n个活动的集合E={1,2,…,n}，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。（临界资源）
- 每个活动i都有一个要求使用该资源的起始时间s_i和一个结束时间f_i，且$s_i<f_i$。
- 如果选择了活动i，则它在半开时间区间$[s_i, f_i)$内占用资源。若区间$[s_i, f_i)$与区间$[s_j, f_j)$不相交，则称活动i与活动j是相容的。即，当$s_i \geq f_j$或$s_j \geq f_i$时，活动i与活动j相容，
- 问题就是选择一个*由互相兼容的活动组成的最大集合*。

活动安排问题的贪心算法

```
template<class Type>
void GreedySelector(int n, Type s[], Type f[], bool A[])
{ //各活动的起始时间和结束时间存储在数组s和f中
  //且按结束时间的非递减排序: f1≤f2≤…≤fn排列。
  A[1]=true;  //用集合A存储所选择的活动
  int j=1;
  for(int i=2; i<=n; i++) {
       //将与j相容的具有最早完成时间的相容活动加入集合A
     if(s[i]>=f[j]) {A[i]=true; j=i; }
     else A[i]=false; }
}
```

算法分析

- 设集合a包含已被选择的活动， 初始时为空。所有待选择的活动按结束时间的非递减顺序排列：

$$f_1 \le f_2 \le \cdots f_n$$

- **变量*j*指出最近加入a的活动序号。由于按结束时间非递减顺序来考虑各项活动的，所以f_j总是a中所有活动的最大结束时间**

```
void GreedySelector(int n,Type s[],Type f[],bool A[]){
A[1]=true;
 int j=1;
 for(int i=2; i<=n; i++){
      if(s[i]>=f[j])
        { A[i]=true; j=i; }
      else A[i]=false;
     }
}
```

首先将活动1加入集合a；

初始化j为1；

然后从活动2出发逐一检查：

☆ 若待选活动i与a中的所有活动兼容，即活动i 的s_i不早于最近加入a中的j活动的f_j，则活动i加入a集合，并取代活动j的位置。

☆否则不选择该活动。

由于输入活动是以完成时间的非递减排列，所选择的下一个活动总是可被合法调度的活动中具有最早结束时间的那个，所以算法是一个“贪心的”选择，即使得使剩余的可安排时间段极大化，以便安排尽可能多的相容活动。

结论

- 算法GreedySelector的效率极高。当输入的活动已按结束时间的非减序排列，算法只需O(n)的时间就可安排n个活动，使最多的活动能相容地使用公共资源。
- 如果所给出的活动未按非减序排列，可以用O(nlogn)的时间重排。

举例

- 设待安排的11个活动的开始时间和结束时间按结束时间的非减序排列如下：

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	8	9	10	11	12	13	14

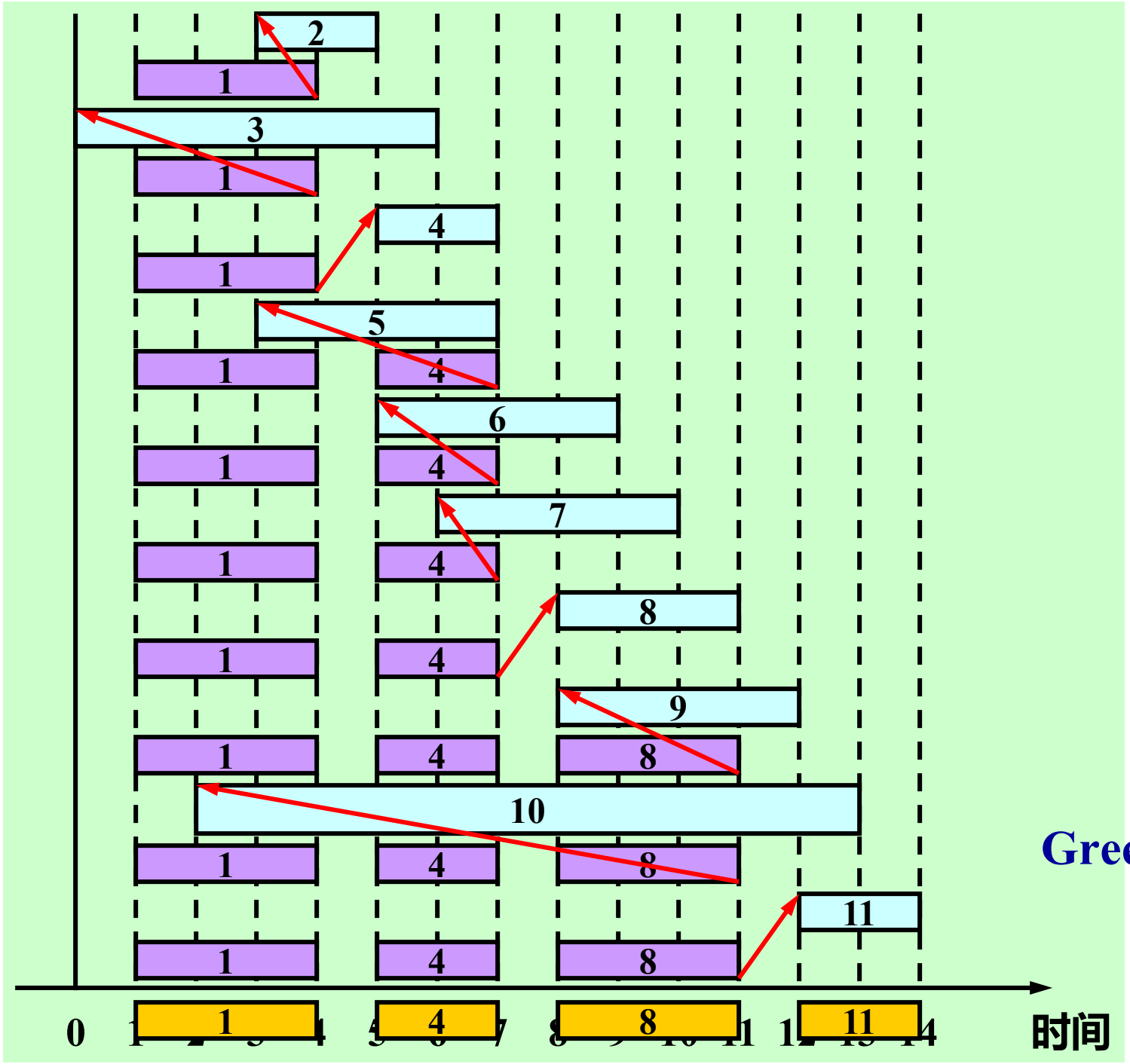


图4-1 算法GreedySelector的计算过程

补充说明

- 贪心算法并不总能求得问题的整体最优解。
- 但对于活动安排问题，贪心算法GreedySelector却总能求得的整体最优解，即它最终所确定的相容活动集合a的规模最大。这个结论可以用数学归纳法证明。

用数学归纳法证明活动安排问题

- 设集合E={1，2，…，n}为所给的活动集合。由于E中活动按结束时间的非减序排列，故活动1有最早完成时间。
- 证明I：活动安排问题有一个最优解以贪心选择开始，即该最优解中包含活动1。
- 证明II：对集合E中所有与活动1相容的活动进行活动安排求得最优解的子问题。

证明I：活动安排问题有一个最优解以贪心选择开始，即该最优解中包含活动1。

设$A \subseteq E$是所给的活动安排问题的一个最优解，

且A中活动也按结束时间非减序排列，A中第一个活动为活动k。

若k=1，则A就是一个以贪心选择开始的最优解。

若k>1，则设 $B = A - \{k\} \cup \{1\}$

由于$f_1 \leq f_k$，且A中的活动是相容的，故B中的活动也是相容的。

结论： 由于B中活动个数与A中活动个数相同，且A是最优的，故B也是最优的。总存在以贪心选择开始的最优活动方案。

贪心选择性质

证明II：对E中所有与*活动1相容的活动*进行活动安排求得最优解的子问题。

- 即需证明：若A是原问题的最优解，则$A'=A-\{1\}$是活动安排问题$E'=\{i\in E: s_i \geq f_1\}$的最优解。
- 如果能找到E'的一个最优解B'，它包含比A'更多的活动，则将活动1加入到B'中将产生E的一个解B，它包含比A更多的活动。这与A的最优性矛盾。
- 结论：每一步所做的贪心选择问题都将问题简化为一个更小的与原问题具有相同形式的子问题。

最优子结构性质

证明III：每次选最小结束时间的活动，定可得到最优解。

引理3. 设 $S=\{1, 2, \cdots, n\}$ 是 n 个活动集合，$f_0=0$，l_i 是 $S_i=\{ j\in S \mid s_j \geq f_{i-1}\}$ 中具有最小结束时间 f_{l_i} 的活动. 设A是S的包含活动1的优化解，其中

$f_1 \leq \cdots \leq f_n$，则 $A=\bigcup_{i=1}^{k}\{l_i\}$

证. 对$|A|$作归纳法.

当$|A|=1$时，由引理1，命题成立.

设$|A|<k$时，命题成立.

当$|A|=k$时，由引理2，$A=\{1\}\cup A_1$，

A_1是 $S_2=\{ j\in S \mid s_j \geq f_1 \}$ 的优化解.

由归纳假设，$A_1=\bigcup_{i=2}^{k}\{l_i\}$

于是，$A=\bigcup_{i=1}^{k}\{l_i\}$.

贪心选择性质

贪心算法的基本要素

- 对于一个具体的问题，如何知道是否可用贪心算法解决此问题，以及能否得到问题的最优解？这个问题很难给予肯定的回答。
- 但是，从许多可以用贪心算法求解的问题中看到这类问题一般具有两个重要的性质：贪心选择性质和最优子结构性质。

1. 贪心选择性质

- 贪心选择性质：所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。
- 动态规划算法通常以自底向上的方式解各子问题，每步所作的选择依赖于相关子问题的解。贪心算法仅在当前状态下作出局部最优选择，再去解作出这个选择后产生的相应的子问题。

分析

- 对于一个具体问题，要确定它是否具有贪心选择的性质，我们必须证明**每一步所作的贪心选择最终能够导致问题的最优解。**
- 通常可以首先证明问题的一个整体最优解是从贪心选择开始的，而且作了贪心选择后，原问题简化为一个规模更小的类似子问题。然后，用数学归纳法证明，通过每一步作贪心选择，最终可得到问题的一个整体最优解。
- 其中，证明贪心选择后的问题简化为规模更小的类似子问题的关键在于利用该问题的最优子结构性质。

2. 最优子结构性质

- 当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。
- 问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的**关键特征。**

3. 贪心算法与动态规划算法的差异

- 贪心算法和动态规划算法都要求问题具有最优子结构性质，这是两类算法的一个共同点。
- 但是，对于具有最优子结构的问题应该选用贪心算法还是动态规划算法求解？是否能用动态规划算法求解的问题也能用贪心算法求解？

回顾：0-1背包问题（动态规划）

- 给定n种物品和一个背包。物品i的重量是w_i，其价值为v_i，背包的容量为c。应如何选择装入背包的物品，使得装入背包中物品的总价值最大？
- 在选择装入背包中的物品时，对每种物品i只有两种选择：即装入或不装入背包。不能将物品i装入背包多次，也不能只装入部分的物品i。因此，该问题称为0-1背包问题。

问题的形式化描述

- 此问题的形式化描述为，给定c>0,w_i>0,v_i>0,1≤i≤n，要求找出一个n元0-1向量 (x_1,x_2, .. ,x_n)，其中x_i∈{0,1}，使得对w_ix_i求和小于等于c，并且对v_ix_i求和达到最大。
- 因此，0-1背包问题是一个特殊的整数规划问题。

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le c \\ x_i \in \{0,1\}, \quad 1 \le i \le n \end{cases}$$

0-1背包问题的子问题递归关系

- 设所给0-1背包问题的子问题：

$$\max \sum_{k=i}^{n} v_k x_k$$

$$\begin{cases} \sum_{k=1}^{n} w_k x_k \le j \\ x_k \in \{0,1\}, \quad i \le k \le n \end{cases}$$

m(i,j)是背包容量为j，可选择物品为i,i+1,…,n时0-1背包问题的最优值。由于0-1背包问题的最优子结构性质，可以建立计算m(i,j)的递归式如下：

$$m(i,j) = \begin{cases} \max\{m(i+1,j), m(i+1,j-w_i)+v_i\} & j \ge w_i \\ m(i+1,j) & 0 \le j \le w_i \end{cases}$$

$$m(n,j) = \begin{cases} v_n & j \ge w_n \\ 0 & 0 \le j \le w_n \end{cases}$$

背包问题

- 与0-1背包问题类似，所不同的是在选择物品i装入背包时，可以选择物品i的一部分，而不一定要全部装入背包。
- 此问题的形式化描述为，给定c>0,w_i>0,v_i>0,1≤i≤n，要求找出一个n元0-1向量 $(x_1,x_2,..,x_n)$，其中$0 \le x_i \le 1$，$1 \le i \le n$，使得对$w_i x_i$求和小于等于c，并且对$v_i x_i$求和达到最大。

贪心算法解背包问题的基本步骤

- 首先计算每种物品单位重量的价值v_i/w_i；
- 然后，依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。
- 若将这种物品全部装入背包后，背包内的物品总重量未超过c，则选择单位重量价值次高的物品并尽可能多地装入背包。依此策略一直地进行下去，直到背包装满为止。

举例

- 有3种物品，背包的容量为50千克。物品1重10千克，价值60元；物品2重20千克，价值100元；物品3重30千克，价值120元。
- 用贪心算法求背包问题。

背包

有3种物品，背包的容量为50千克。物品1重10千克，价值60元；物品2重20千克，价值100元；物品3重30千克，价值120元。

- 贪心策略：物品1，6元/千克；物品2，5元/千克；物品3，4元/千克。

物品3，20kg	80	
	+	
物品2，20kg	100	＝￥240
	+	
物品1，10kg	60	

具体算法

```
void Knapsack(int n, float M, float v[],float w[], float x[])
{  sort(n,v,w);
   int i;
   for(i=1;i<=n;i++)  x[i]=0;
   float c=M;
   for(i=1;i<=n;i++)  {
         if(w[i]>c)  break;
         x[i]=1;
         c-=w[i];
        }
    if(i<=n)  x[i]=c/w[i];
}
```

该算法前提：

所有物品在集合中按其单位重量的价值从小到大排列。

- 算法Knapspack的主要计算时间在于将各种物品按其单位重量的价值从小到大排序，算法的时间复杂度O(nlogn)。
- 对于0-1背包问题，贪心选择之所以不能得到最优解是因为在这种情况下，它无法保证最终能将背包装满，部分闲置的背包空间使每公斤背包空间的价值降低了。