

# 算法分析与设计

## 第5章 贪心算法

---

主讲人：甘文生 PhD

Email: [wsgan001@gmail.com](mailto:wsgan001@gmail.com)

暨南大学网络空间安全学院

Fall 2021

Jinan University, China

# 第5章 贪心算法

## 内容提要:

- 理解贪心算法 (Greedy Method) 的概念
- 掌握贪心算法的基本要素
- 理解贪心算法与动态规划算法的差异
- 通过范例学习贪心算法设计策略

# 旅行商问题

- **旅行商问题 (Traveling Salesman Problem, TSP)**
  - 由威廉哈密顿爵士和英国数学家T. P. Kirkman于19世纪初提出。
  - 问题描述：有若干个城市，任何两个城市之间的距离都是确定的，现要求某旅行商从某城市出发，必须经过每一个城市且只在该城市逗留一次，最后回到出发的城市，问如何确定一条最短的线路保证其旅行的费用最少？
- **中国邮递员问题 (Chinese Postman Problem, CPP)**
  - 问题描述：一个邮递员从邮局出发，到所辖街道投邮件，最后返回邮局，如果他必须走遍所辖的每条街道至少一次，那么他应该如何选择投递路线，使所走的路程最短？
  - 中国数学家管梅谷在1962年首先提出该问题，并给出解法“奇偶点图上作业法”。图论问题

# 图论、旅行商、邮递员问题

- **第1类**：遍历完所有的边而不能有重复，即“一笔画问题”或“欧拉路径”；(欧拉图)
- **第2类**：遍历完所有的顶点而没有重复，即“哈密尔顿问题”(哈密尔顿图)
- **第3类**：遍历完所有的边而可以有重复，即“中国邮递员问题”；
- **第4类**：遍历完所有的顶点而可以重复，即“旅行推销员问题”。

第一和第三类问题已有完满的解决，而第二和第四类问题只得到了部分解决。

**TSP和CPP问题的区别**：图由边和顶点组成，CPP需要访问所有边，TSP需要访问所有顶点。

# 最优化问题

一般特征：问题有 $n$ 个输入，问题的解是由这 $n$ 个输入的某个子集组成，这个子集必须满足某些事先给定的条件。

- 约束条件：子集必须满足的条件；
- 可行解：满足约束条件的子集；可行解可能不唯一；
- 目标函数：用来衡量可行解优劣的标准，一般以函数形式给出；
- 最优解：能够使目标函数取极值（极大或极小）的可行解。

**最优化问题求解分类**：根据描述问题约束条件和目标函数的数学模型的特性和问题的求解方法的不同，可分为：线性规划、整数规划、非线性规划、动态规划、分支限界法等**精确算法**。

**贪心方法**：一种改进的分级的处理方法，可对满足上述特征的某些问题方便地求解，属于**近似算法**。即每次在做选择时，总是先选择具有相同特征的那个解，即“贪心解”。

# 最优化问题

!!! 贪心方法问题的一般特征：问题的解是由n个输入的、满足某些事先给定的条件的子集组成。

## 1) 一般方法

根据题意，选取一种度量标准。然后按照这种度量标准对n个输入排序，并按序一次输入一个量。如果这个输入和当前已构成在这种量度意义下的部分最优解加在一起不能产生一个可行解，则不把此输入加到这部分解中。否则，将当前输入合并到部分解中从而得到包含当前输入的新的部分解。

## 2) 贪心方法

能够得到某种量度意义下的最优解的分级处理方法称为贪心方法。

3) 使用贪心策略求解的关键：选取能够得到问题最优解的量度标准。

注：贪心解  $\neq$  最优解

直接将目标函数作为量度标准也不一定能够得到问题的最优解

# 贪心算法 (Greedy Algorithm)

- 当某问题具有**最优子结构**性质时，可用动态规划法求解(精解确、大规模等问题效率低)，但有时用贪心算法求解会更简单、更有效。
- 顾名思义，**贪心算法**总是作出在当前看来最好的选择，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，其希望得到的最终结果是整体最优的。虽然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，贪心算法不能得到整体最优解，但其**最终结果是最优解的近似**。

# 贪心算法 (Greedy Algorithm)

```
procedure GREEDY(A, n) //A(1: n)包含n个输入
  solutions  $\leftarrow$   $\varnothing$  //将解向量solution初始化为空
  for i  $\leftarrow$  1 to n do
    x  $\leftarrow$  SELECT(A) //按照度量标准，从A中选择一个输入，其值赋予x，并将
    之从A中删除
    if FEASIBLE(solution, x) //判定x是否可以包含在解向量中，即是否能共
    同构成可行解
      then solutions  $\leftarrow$  UNION(solution, x) //目标函数将x和当前的解向
      量合并成新的解向量，并修改
    endif
  repeat
  return solution
```

- ❑ **贪心方法的抽象化控制：**一旦能选择出某个问题的最优量度标准，那么用贪心方法求解该问题会特别简单有效。



# 贪心算法 vs 动态规划

- 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，但是两者存在着巨大的差别。
- (1) 动态规划是先分析子问题，再做选择。而贪心算法是先做贪心选择，做完选择后，生成子问题，然后再去求解子问题；
- (2) 动态规划每一步可能会产生多个子问题，而贪心算法每一步只会产生一个子问题 (为非空)；
- (3) 从特点上看，动态规划是 **自底向上** 解决问题，而贪心算法是 **自顶向下** 解决问题。

# 活动安排问题

- 设有 $n$ 个活动的集合 $E = \{1, 2, \dots, n\}$ ,
  - 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。
  - 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: 选出最大的相容活动子集合。

# 活动安排问题

## □ 运用动态规划方法

### ● 步骤1: 分析最优解的结构特征

— 构造子问题空间:

$$S_{ij} = \{ a_k \in S : f_i \leq s_k < f_k \leq s_j \}$$

$S_{ij}$ 包含了所有与 $a_i$ 和 $a_j$ 相兼容的活动, 并且与不迟于 $a_i$ 结束和不早于 $a_j$ 开始的活动兼容。此外, 虚构活动 $a_0$ 和 $a_{n+1}$ , 其中 $f_0=0, s_{n+1}=\infty$ 。原问题即为寻找 $S_{0,n+1}$ 中最大兼容活动子集。

# 活动安排问题

- 证明原问题具有最优子结构性质。即：若已知问题 $S_{ij}$ 的最优解 $A_{ij}$ 中包含活动 $a_k$ ，则在 $S_{ij}$ 最优解中的针对 $S_{ik}$ 的解 $A_{ik}$ 和针对 $S_{kj}$ 的解 $A_{kj}$ 也必定是最优的。（反证法即可）
- 证明可以根据子问题的最优解来构造出原问题的最优解。  
一个非空子问题 $S_{ij}$ 的任意解中必包含了某项活动 $a_k$ ，而 $S_{ij}$ 的任一最优解中都包含了其子问题实例 $S_{ik}$ 和 $S_{kj}$ 的最优解（根据最优子结构性质）。因此，可以构造出 $S_{ij}$ 的最大兼容子集。

# 活动安排问题

## □ 步骤2：递归地定义最优解的值

设 $c[i, j]$ 为 $S_{ij}$ 中最大兼容子集中的活动数。当 $S_{ij} = \phi$ 时,  $c[i, j] = 0$ 。  
对于一个非空子集 $S_{ij}$ , 如果 $a_k$ 在 $S_{ij}$ 的最大兼容子集中被使用, 则子问题 $S_{ik}$ 和 $S_{kj}$ 的最大兼容子集也被使用。从而:

$$c[i, j] = c[i, k] + c[k, j] + 1$$

由于 $S_{ij}$ 的最大子集一定使用了 $i$ 到 $j$ 中的某个值 $k$ , 通过检查所有可能的 $k$ 值, 就可以找到最好的一个。因此,  $c[i, j]$ 的完整递归定义为:

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

# 活动安排问题

- **问题：** $k$ 有 $j-i-1$ 种选择，然后一一遍历求出所有的选择，每种选择会导致2个完全不同的子问题产生，因此，动态规划算法的计算量比较大！
- 一个直观想法是**直接选择** $k$ 的值，使得一个子问题为空，从而加快计算速度！这就导致了贪心算法！
- 思考：在该问题中，贪心算法产生了2个子问题？但是其中一个为空的？

# 活动安排问题

## □ 用贪心算法

**贪心策略：**对输入的活动以其完成时间的**非减序**排列，算法每次总是选择**具有最早完成时间**的相容活动加入最优解集中。直观上，按这种方法选择相容活动为未安排活动留下尽可能多的时间。也就是说，该算法的贪心选择的意义是**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。

**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

# 活动安排问题

## □ 用贪心算法

**贪心策略：**对输入的活动以其完成时间的**非减序**排列，算法每次总是选择**具有最早完成时间**的相容活动加入最优解集中。

**为什么选择这个最优策略？**

1. 如果选择最早开始时间为最优策略，那么有一个问题：如果它的结束时间最晚，晚到整个舞台使用时间，显然这个策略是有问题的。
2. 如果选择最短活动时间为最优策略，那么如果它是最晚开始的，晚到整个舞台的最后使用时间，显然这个策略也不行。
3. 那么就剩下以最早结束时间为最优策略。稍后的详解可以证明这个选择是对的。

**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



# 活动安排问题

11个活动已按结束时间排序，用贪心算法求解：

i	1	2	3	4	5	6	7	8	9	10	11
start_time <sub>i</sub>	1	3	0	5	3	5	6	8	8	2	12
finish_time <sub>i</sub>	4	5	6	7	8	9	10	11	12	13	14

time	a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>	a <sub>4</sub>	a <sub>5</sub>	a <sub>6</sub>	a <sub>7</sub>	a <sub>8</sub>	a <sub>9</sub>	a <sub>10</sub>	a <sub>11</sub>
0											
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											
11											
12											
13											
14											

相容活动：{a<sub>3</sub>, a<sub>9</sub>, a<sub>11</sub>},  
{a<sub>1</sub>, a<sub>4</sub>, a<sub>8</sub>, a<sub>11</sub>}, {a<sub>2</sub>, a<sub>4</sub>, a<sub>9</sub>, a<sub>11</sub>}



time	a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>	a <sub>4</sub>	a <sub>5</sub>	a <sub>6</sub>	a <sub>7</sub>	a <sub>8</sub>	a <sub>9</sub>	a <sub>10</sub>	a <sub>11</sub>
0											
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											
11											
12											
13											
14											

# 活动安排问题

时间复杂性分析：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 $O(n\log n)$ 的时间重排。

```
1 package dynamic;
2
3 /*
4  * 代码描述：
5  * 贪心算法一开始选择活动1，并将 j 初始化为1。然后依次检查活动 i 是否与当前已选择的所有活动相容。
6  * 活动 i 与当前集合 A 中所有活动相容的充分且必要的条件是其开始时间 s 不早于最近加入集合 a 中的活
7  * 动 j 的结束时间 fj， si ≥ fj。若活 动 i 与之相容，则 i 成为最近
8  * 动 j 的位置。
9  */
10
11 public class GreedySelector {
12
13     private static int[] s= {0,1,3,0,5,3,5,6,8,8,2,12};
14     private static int[] f= {0,4,5,6,7,8,9,10,11,12,13,14};
15     private static boolean a[]=new boolean[f.length];
16     private static int n=f.length-1;
17
18     public static int greedySelector(int[] s,int[] f,boolean a)
19     /*
20      * s[]记录活动开始时间
21      * f[]记录活动结束时间，而且按降序排列
22      * a[i]=true记录加入到安排表内的活动
23      */
24     a[1]=true; //贪心算法一开始选择活动1
25     int j=1; //将j初始化为1
26     int count=1; //记录加入到安排表内的活动数
27     for(int i=2;i<=n;i++){ //开始选择活动1以后的活动
28         if(s[i]>=f[j]){ //如果剩余活动的开始时间晚于或等于前一活动的结束时间，那么加入到安排表内
29             a[i]=true; //表示这个活动i可以加入到安排表内
30             j=i; //用于下一次剩余活动开始时间与上一个加入
31             count++; //活动数增加
32         }
33         else a[i]=false; //否则这个活动不加入到安排表内
34     }
35     System.out.print("加入到安排表内的活动为：活动");
36     for(int k=1;k<=n;k++) {
37         if(a[k]==true)
38             System.out.print(k+"、");
39     }
40     return count;
41 }
42
43 public static void main(String[] args) {
44     // TODO Auto-generated method stub
45
46     greedySelector(s,f,a);
47 }
48
49 }
```

# 活动安排问题

Recursive-Activity-Selector( $s, f, i, j$ )

```
{  
  ①  $m \leftarrow i+1$ ;  
  ② while  $m < j$  and  $s_m < f_i$   
  ③   do  $m \leftarrow m+1$   
  ④ if  $m < j$   
  ⑤   then return  $\{a_m\}$  UNION Recursive-Activity-Selector( $s, f, m, j$ )  
  ⑥   else return  $\varnothing$   
}
```

说明:

- 1) 数组 $s$ 和 $f$ 表示活动的开始和结束时间,  $n$ 个输入活动已经按照活动结束时间进行单调递增顺序排序;
- 2) 算法②~③目的是寻找 $S_{ij}$ 中最早结束的第一个活动, 即找到与 $a_i$ 兼容的第一个活动 $a_m$ , 利用 $a_m$ 与 $S_{mj}$ 的最优子集的并集构成 $S_{ij}$ 的最优子集;
- 3) 时间复杂度 $O(n)$ 。

# 活动安排问题

- Recursive-Activity-Selector属于递归性贪心算法，它以对自己的递归调用的并操作结束，几乎为尾递归调用，因此可以转化为迭代形式：

Greedy-Activity-Selector(s, f )

```
{  
    n ← length[s];  
    A ← {a1}  
    i ← 1 // 下标i记录了最近加入A的活动ai  
    for m ← 2 to n // 寻找Si,n+1中最早结束的兼容活动  
        do if sm ≥ fi  
            then A ← A U {am}  
                i ← m  
    return A;  
}
```

# 附录：尾递归调用

❑ `function story() {`

从前有座山，山上有座庙，庙里有个老和尚，一天老和尚对小和尚讲故事：`story()`  
*// 尾递归，进入下一个函数不再需要上一个函数的环境了，得出结果以后直接返回。*  
`}`

尾递归函数中，递归调用是递归函数中的最后一条指令，并且在函数中只有一次递归调用。

❑ `function story() {`

从前有座山，山上有座庙，庙里有个老和尚，一天老和尚对小和尚讲故事：`story()`  
，小和尚听了，找了块豆腐撞死了 *// 非尾递归，下一个函数结束以后此函数还有后续，所以必须保存本身的环境以供处理返回值。*  
`}`

# 附录：尾递归调用

// 树形递归- 最直观的递归实现

```
1.  int fib_1(int n) {  
2.  if (n <= 1) {  
3.      return 1;  
4.  } else {  
5.      return fib_1(n - 1) + fib_1(n - 2);  
6.  }  
7.  }
```

// 尾递归

```
1.  int fib_4(int n) {  
2.  int fib_iter(int a , int b , int n){  
3.  if (n == 0) {  
4.      return a;  
5.  }else {  
6.      return fib_iter(b, a+b, n-1);  
7.  }
```

//尾递归，进入下一个函数不再需要上一个函数的环境，得出结果以后直接返回。

```
1.  }  
2.  }  
3.  return fib_iter(1 , 1 , n);
```

# 活动安排问题

## □ 贪心算法的正确性证明:

**定理5.1:** 对于任意非空子问题 $S_{ij}$ , 设 $a_m$ 是 $S_{ij}$ 中具有最早结束时间的活动, 即  $f_m = \min\{f_k : a_k \in S_{ij}\}$ , 则:

- 1) 活动 $a_m$ 在 $S_{ij}$ 的某最大兼容活动子集中被使用;
- 2) 子问题 $S_{im}$ 为空, 故选择 $a_m$ 能让子问题 $S_{mj}$ 为唯一可能非空的子问题。

□ **Exchange argument**, 即交换两个算法里的一个步骤(或元素), 得到一个新的最优的算法, 从而得到矛盾, 原命题成立。

□ **反证法**, 即首先假设我们的贪心算法不成立(即在原命题的题设下, 结论不成立), 然后推理出明显矛盾的结果, 从而下结论说假设不成立, 贪心算法得证。



# 活动安排问题

**定理5.1:** 对于任意非空子问题 $S_{ij}$ , 设 $a_m$ 是 $S_{ij}$ 中具有最早结束时间的活动, 即  $f_m = \min\{f_k: a_k \in S_{ij}\}$ , 则:

- 1) 活动 $a_m$ 在 $S_{ij}$ 的某最大兼容活动子集中被使用;
- 2) 子问题 $S_{im}$ 为空, 选择 $a_m$ 使子问题 $S_{mj}$ 为唯一可能非空的子问题。

**证明** 先证第2部分。假设 $S_{im}$ 非空, 因此有活动 $a_k$ 满足 $f_i \leq s_k < f_k \leq s_m < f_m$ 。  $a_k$ 同时也在 $S_{ij}$ 中, 且具有比 $a_m$ 更早的结束时间, 这与 $a_m$ 的选择相矛盾, 故 $S_{im}$ 为空。



# 活动安排问题

**定理5.1:** 对于任意非空子问题 $S_{ij}$ , 设 $a_m$ 是 $S_{ij}$ 中具有最早结束时间的活动, 即  $f_m = \min\{f_k: a_k \in S_{ij}\}$ , 则:

- 1) 活动 $a_m$ 在 $S_{ij}$ 的某最大兼容活动子集中被使用;
- 2) 子问题 $S_{im}$ 为空, 选择 $a_m$ 使子问题 $S_{mj}$ 为唯一可能非空的子问题。

**证明:** 再证第1部分。设 $A_{ij}$ 为 $S_{ij}$ 的最大兼容活动子集, 且将 $A_{ij}$ 中的活动按结束时间单调递增排序。设 $a_k$ 为 $A_{ij}$ 的第一个活动。如果 $a_k = a_m$ , 则得到结论, 即 $a_m$ 在 $S_{ij}$ 的某个最大兼容子集中被使用。如果 $a_k \neq a_m$ , 则构造子集 $B_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$ 。因为在活动 $A_{ij}$ 中,  $a_k$ 是第一个结束的活动, 而 $f_m \leq f_k$ , 所以 $B_{ij}$ 中的活动是不相交的, 即 $B_{ij}$ 中活动是兼容的。同时,  $B_{ij}$ 中活动个数与 $A_{ij}$ 中活动个数一致, 因此 $B_{ij}$ 是包含 $a_m$ 的 $S_{ij}$ 的最大兼容活动集合。

# 贪心算法的基本思想

## 基本思想：

- 从问题的某一个初始解出发，通过一系列的贪心选择，即当前状态下的局部最优选择，逐步逼近给定的目标，尽可能快地求得更好的解。
- 在贪心算法(Greedy Method)中采用逐步构造/分级最优解的方法。在每个阶段，都作出一个按某个评价函数最优的决策，该最优评价函数称为贪心准则(Greedy Criterion)
- 贪心算法的正确性，要证明按贪心准则求得的解是全局最优解。

# 贪心算法的基本步骤

## 基本步骤:

- ① 决定问题的最优子结构;
- ② 设计出一个递归解;
- ③ 证明在递归的任一阶段, 最优选择之一总是贪心选择, 那么做贪心选择总是安全的。
- ④ 证明通过做贪心选择, 所有子问题(除一个以外)都为空, 即只产生一个子问题。
- ⑤ 设计出一个实现贪心策略的递归算法。
- ⑥ (性能角度) 将递归算法转换成迭代算法。

# 贪心算法的基本要素

- 对于一个具体的问题，怎么知道是否可用贪心算法解此问题，以及能否得到问题的最优解呢？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2个重要的性质：**贪心选择性质**和**最优子结构性质**。

## 一、贪心选择性质

所谓**贪心选择性质**是指所求问题的**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，而贪心算法则通常以**自顶向下**的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。

对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解，否则得到的是**近优解**。

# 贪心算法的基本要素

## 二、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。但是，需要注意的是，并非所有具有最优子结构性质的问题都可以采用贪心策略来得到最优解。

贪心算法只需考虑一个选择(即贪心的选择)；在做贪心选择时，子问题之一必须是空的，因此只留下一个非空子问题。

# 贪心算法 vs 动态规划

- 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，但是两者存在着巨大差别。

## Dynamic Programming

- At each step, the choice is determined based on solutions of subproblems.
- Sub-problems are solved first.
- Bottom-up approach
- Can be slower, more complex

## Greedy Algorithms

- At each step, we quickly make a choice that currently looks best.  
-A local optimal (greedy) choice.
- Greedy choice can be made first before solving further sub-problems.
- Top-down approach
- Usually faster, simpler



# 贪心算法 vs 动态规划

- 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，但是两者存在着巨大的差别。
- (1) 动态规划是先分析子问题，再做选择。而贪心算法是先做贪心选择，做完选择后，生成了子问题，然后再去求解子问题；
- (2) 动态规划每一步可能会产生多个子问题，而贪心算法的每一步只会产生一个子问题；
- (3) 从特点上看，动态规划是 自底向上 解决问题，而贪心算法则是 自顶向下 解决问题。

# 与递归、分治的联系

- **分治策略**用于解决原问题与子问题结构相似的问题，对于各子问题相互独立的情况，一般用递归实现；
- **动态规划**用于解决子问题有重复求解的情况，既可以用递归实现，也可以用迭代实现；
- **贪心算法**用于解决具有贪心选择性质的问题，既可以用递归实现，也可以用迭代实现，因为很多递归贪心算法都是尾递归，很容易改成迭代贪心算法；
- **递归是实现手段**，分治策略是解决问题的思想，动态规划和贪心算法很多时候会使用记录子问题运算结果的递归实现。



# 贪心算法的基本要素

## □ 0-1 背包问题 0/1 Knapsack Problem

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？

在选择装入背包的物品时，对每种物品  $i$  只有2种选择，即装入背包或不装入背包。不能将物品 $i$  装入背包多次，也不能只装入部分的物品  $i$  。

## □ 小数背包问题 Fractional Knapsack Problem

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， $1 \leq i \leq n$ 。

□ 这2类问题都具有**最优子结构**性质，极为相似，但背包问题可以用贪心算法求解，而0-1背包问题却不能用贪心算法求解。

# 贪心算法的基本要素

例：若  $n = 3$ ,  $w = (10, 20, 30)$ ,  $v = (60, 100, 120)$ ,  $c = 50$ , 则

□ 对于0-1背包问题，可行解为：

$$(x_1, x_2, x_3) = (0, 1, 1) = 220$$

$$(x_1, x_2, x_3) = (1, 1, 0) = 160$$

$$(x_1, x_2, x_3) = (1, 0, 1) = 180$$

最优解为：选择物品2和物品3，总价值为220

□ 对于小数背包问题，按照物品价值率最大的贪心选择策略，其解为(10, 20, 20)，总价值为240。

□ 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。事实上，在考虑0-1背包问题时，应比较选择该物品和不选择该物品所导致的最终方案，然后再作出最好选择。由此就导出许多互相重叠的子问题。这是该问题可用动态规划算法求解的另一重要特征。实际上，动态规划算法的确可以有效地解0-1背包问题。

# 小数背包问题

- **问题描述：**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？这里，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。

$$\begin{aligned} \max & \sum_{i=1}^n v_i x_i && x_i \text{ 为装入物品 } i \text{ 的比例} \\ \left\{ \begin{array}{l} \sum_{i=1}^n w_i x_i \leq c \quad w_i > 0 \\ 0 \leq x_i \leq 1 \quad i = 1, 2, \dots, n \\ v_i > 0, w_i > 0, c > 0 \quad i = 1, 2, \dots, n \end{array} \right. && \begin{array}{l} w_i \text{ 为重量, } c \text{ 为背包容量} \\ v_i \text{ 为价值} \\ v_i / w_i \text{ 为价值率(单位重量价值)} \end{array} \end{aligned}$$

- **例子：** $n=3, c=20, v=(25, 24, 15), w=(18, 15, 10)$ ，列举4个可行解：

	$(x_1, x_2, x_3)$	$\sum w_i x_i$	$\sum v_i x_i$
①	$(1/2, 1/3, \frac{1}{4})$	16.5	24.5
②	$(1, 2/15, 0)$	20	28.2
③	$(0, 2/3, 1)$	20	31
④	$(0, 1, \frac{1}{2})$	20	31.5 (最优解)

# 小数背包问题

## □ 贪心策略设计：

策略1：按**价值最大**贪心，是目标函数增长最快。

按价值排序从高到低选择物品→②解(次最优)

策略2：按**重量最小**贪心，使背包增长最慢。

按重量排序从小到大选择物品→③解(次最优解)

策略3：按**价值率最大**贪心，使单位重量价值增长最快。

按价值率排序从大到小选择物品→④解(最优)

# 小数背包问题

## □ 算法:

```
GreedyKnapsack( n, M, v[], w[], x[] )
{ //按价值率最大贪心选择
  Sort( n, v, w); //使得 $v_1/w_1 \geq v_2/w_2 \geq \dots \geq v_n/w_n$ 
  for i = 1 to n do x[i]=0;
  c = M;
  for i = 1 to n do
  {
    if( w[i] > c) break;
    x[i]=1;
    c-=w[i];
  }
  if( i ≤ n) x[i] = c/w[i]; //使物品i是选择的最后一项
}
```

## □ 时间复杂度: $T(n) = O(n \lg n)$

# 小数背包问题

## □ 贪心选择的最优性证明

**定理：**如果  $v_1/w_1 \geq v_2/w_2 \geq \dots \geq v_n/w_n$ ，则 GreedyKnapsack 算法对于给定的背包问题实例生成一个最优解

**证明基本思想：**

把贪心解与任一最优解相比较，如果这两个解不同，就去找开始不同的第一个  $x_i$ ，然后设法用贪心解的  $x_i$  去代换最优解的  $x_i$ ，并证明最优解在 **分量代换之后** 其总价值保持不变，反复进行下去，直到新产生的最优解与贪心解完全一样，从而证明贪心解是最优解。



# 小数背包问题

— 证明：设 $(x_1, \dots, x_n)$ 是贪心算法求得的解

Case1: 所有 $x_i = 1$ 。显然该解就是最优解。

Case2: 设 $X = (1, \dots, 1, x_j, 0, \dots, 0)$   $x_j \neq 1, 1 \leq j \leq n$ 。下证 $X$ 就是最优解。设问题的最优解 $Y = (y_1, \dots, y_n)$ , 则存在 $k$ 使得 $y_k \neq x_k$ 的最小下标 (否则 $Y = X$ , 得证)。

首先, 可以证明 $y_k < x_k$ 。(反证: 若 $y_k > x_k$ , 则 $x_k \neq 1, \therefore k \geq j$

$$\Rightarrow \sum_{i=1}^n w_i y_i \geq \sum_{i=1}^k w_i y_i > \sum_{i=1}^k w_i x_i \geq \sum_{i=1}^j w_i x_i = c \therefore Y \text{ 不是可行解, 矛盾})$$

下面改造 $Y$ 成为新解 $Z = (z_1, \dots, z_n)$ , 并使 $Z$ 仍为最优解。将 $y_k$ 增加到 $x_k$ , 从 $(y_{k+1}, \dots, y_n)$ 中减同样的重量使总量仍是 $c$ 。即,

$$z_i = x_i \quad i = 1, 2, \dots, k; \quad \text{和} \quad w_k(z_k - y_k) = \sum_{i=k+1}^n w_i(y_i - z_i)$$

# 小数背包问题

$$\begin{aligned}\therefore \sum_{i=1}^n w_i z_i &= \sum_{i=1}^{k-1} w_i z_i + w_k z_k + \sum_{i=k+1}^n w_i z_i \\ &= \sum_{i=1}^{k-1} w_i y_i + w_k z_k + \left( \sum_{i=k+1}^n w_i y_i - w_k (z_k - y_k) \right) = \sum_{i=1}^n w_i y_i = c \\ \therefore \sum_{i=1}^n v_i z_i &= \sum_{i=1}^n v_i y_i + (z_k - y_k) v_k - \sum_{i=k+1}^n (y_i - z_i) v_i \quad // \because Z \text{ 由 } Y \text{ 变得的} \\ &= \sum_{i=1}^n v_i y_i + (z_k - y_k) w_k v_k / w_k - \sum_{i=k+1}^n (y_i - z_i) w_i v_i / w_i \\ &\geq \sum_{i=1}^n v_i y_i + \left[ (z_k - y_k) w_k - \sum_{i=k+1}^n (y_i - z_i) w_i \right] v_k / w_k \quad // \text{利用 } v_i / w_i \downarrow \\ &= \sum_{i=1}^n v_i y_i\end{aligned}$$

$\therefore Z$ 也是最优解, 且 $z_i = x_i \ i=1, \dots, k$ ; 重复上面过程  $\Rightarrow X$ 为最优解。



# 小数背包问题

## □ 特殊的0-1背包问题:

如果  $w_1 \leq w_2 \leq \dots \leq w_n$ ,  $v_1 \geq v_2 \geq \dots \geq v_n$ , 则  $v_1/w_1 \geq v_2/w_2 \geq \dots \geq v_n/w_n$ , 此时可以用贪心法求最优解。

```
O-1-Knapsack( v[], w[], n, c )
{  //输出x[1...n]
  for i=1 to n do x[i]=0;
  value = 0.0;
  for i=1 to n do
  {
    if ( w[i]<c )
      { x[i] = 1; c-= w[i]; value +=v[i]; }
    else break;
  }
  return value;
}
```

# 最优装载问题

## □ 问题的描述:

轮船载重为 $c$ ，集装箱重量为 $w_i$  ( $i = 1, 2, \dots, n$ )，在装载体积不受限制的情况下，将尽可能多的集装箱装上船。

## □ 形式化定义:

$$\begin{aligned} & \max \sum_{i=1}^n x_i \\ & s.t. \begin{cases} \sum_{i=1}^n w_i x_i \leq c & w_i > 0 \\ x_i \in \{0, 1\} & i = 1, 2, \dots, n \end{cases} \end{aligned}$$

# 最优装载问题

- **贪心策略：**从剩下的货箱中，选择重量最小的货箱。这种选择次序可以保证所选的货箱总重量最小，从而可以装载更多的货箱。根据这种贪心策略，首先选择最轻的货箱，然后选择次轻的货箱，如此下去直到所有货箱均装上船或者船上不能容纳其他任何一个货箱。
- **计算实例：**假设  $n=8$ ,  $[w_1, w_2, \dots, w_8] = [100, 200, 50, 90, 150, 50, 20, 80]$ ,  $c=400$ 。

利用贪心算法时，所考察货箱的顺序为7, 3, 6, 8, 4, 1, 5, 2。  
货箱7, 3, 6, 8, 4, 1的总重量为390个单位且已被装载，剩下的装载能力为10个单位，小于剩下的任何一个货箱。在这种贪心解决算法中得到  $[x_1, x_2, \dots, x_8] = [1, 0, 1, 1, 0, 1, 1, 1]$ ，且  $\sum x_i = 6$

# 最优装载问题

## □ 算法描述:

ContainerLoading(  $x[]$ ,  $w[]$ ,  $c$ ,  $n$  )

```
{ //x[i]=1当且仅当货箱i被装载, 对重量按间接寻址方式排序
  new t[n+1];           //产生数组t, 用于间接寻址
  IndirectSort(w, t, n); //此时,  $w[t[i]] \leq w[t[i+1]]$ ,  $1 \leq i < n$ 
  for i = 1 to n do      //初始化x
    x[i] = 0;
  for(i=1; i ≤ n && w[t[i]] ≤ c; i++)
  { //按重量次序选择物品
    x[t[i]] = 1;
    c = c - w[t[i]];    //c为剩余容量
  }
  delete t[];
}
```

时间复杂度:  $O(n \lg n)$

# 最优装载问题

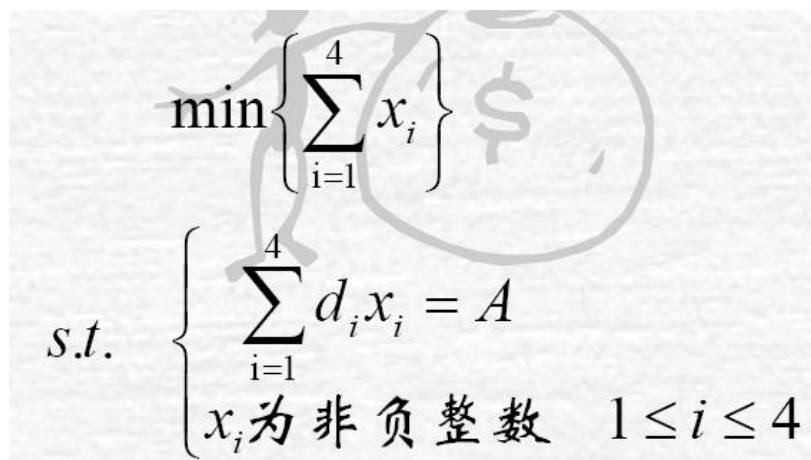
## □ 贪心性质证明:

不失一般性, 假设货箱都排好序, 即  $w_i \leq w_{i+1}$  ( $1 \leq i \leq n$ )。令  $x = [x_1, \dots, x_n]$  为用贪心算法获得的解,  $y = [y_1, \dots, y_n]$  为一个最优解, 分若干步可以将  $y$  转化为  $x$ , 转换过程中每一步都产生一个可行的新  $y$ , 且  $\sum y_i$  ( $1 \leq i \leq n$ ) 的值不变(即仍为最优解), 从而证明了  $x$  为最优解。

# 找钱问题

## □ 问题定义:

使用2角5分, 1角, 5分和1分四种面值的硬币时(各种硬币数量不限), 设计一个找A分钱的贪心算法, 并证明算法能产生一个最优解。设货币种类 $P = \{p_1, p_2, p_3, p_4\}$ ,  $d_i$ 和 $x_i$ 分别是 $p_i$ 的货币单位和选择数量, 问题的形式描述为:


$$\begin{aligned} & \min \left\{ \sum_{i=1}^4 x_i \right\} \\ \text{s.t. } & \begin{cases} \sum_{i=1}^4 d_i x_i = A \\ x_i \text{ 为非负整数} & 1 \leq i \leq 4 \end{cases} \end{aligned}$$

# 找钱问题

- 贪心策略

$$\begin{aligned}x_1 &= \lfloor A/d_1 \rfloor & x_3 &= \lfloor (A - d_1x_1 - d_2x_2)/d_3 \rfloor \\x_2 &= \lfloor (A - d_1x_1)/d_2 \rfloor & x_4 &= A - d_1x_1 - d_2x_2 - d_3x_3\end{aligned}$$

- 算法

GreedyChange( d[], x[], A)

{//输出x[1..4]

  d[1]=25, d[2]=10, d[3]=5, d[4]=1;

  for i=1 to 3 do

  {  x[i]=A/d[i]; A-=x[i]\*d[i];

  }

  x[4]=A;

}



# 找钱问题

## □ 最优子结构性证明:

设 $X=(x_1, x_2, x_3, x_4)$ 是问题钱数为 $A$ 的最优解, 则 $X'=(0, x_2, x_3, x_4)$ 是子问题钱数为 $A-d_1x_1$ 的最优解。

下面反证: 若不然,  $Y=(0, y_2, y_3, y_4)$ 是子问题钱数为 $A-d_1x_1$ 的最优解, 即 $Y$ 优于 $X'$

$$\sum_{i=2}^4 y_i < \sum_{i=2}^4 x_i \quad \text{且} \quad \sum_{i=2}^4 d_i y_i = A - d_1 x_1$$

$$\Rightarrow x_1 + \sum_{i=2}^4 y_i < \sum_{i=1}^4 x_i \quad \text{且} \quad d_1 x_1 + \sum_{i=2}^4 d_i y_i = A$$

$\Rightarrow (x_1, y_2, y_3, y_4)$ 比 $(x_1, x_2, x_3, x_4)$ 更优, 矛盾。

# 找钱问题

## □ 贪心选择性质证明:

设  $X=(x_1, x_2, x_3, x_4)$  是贪心解,  $Y=(y_1, y_2, y_3, y_4)$  是最优解.

可以证明:  $x_1 = y_1$

如果  $x_1 < y_1$ , 由  $x_1 = \lfloor A/d_1 \rfloor \Rightarrow x_1 \leq A/d_1 < x_1 + 1 \Rightarrow d_1 x_1 \leq A <$

$d_1(x_1 + 1)$ ; 因此,  $\sum_{i=1}^4 d_i y_i \geq d_1 y_1 \geq d_1(x_1 + 1) > A$ , 产生矛盾;

如果  $x_1 > y_1$ , 则  $10y_2 + 5y_3 + 1y_4 \geq 25$  (否则,  $25y_1 + 10y_2 + 5y_3 + 1y_4 < 25(y_1 + 1) \leq 25x_1 + 10x_2 + 5x_3 + 1x_4 = A$ , 矛盾), 因此用一个25分的硬币替代等值的低于25分的硬币若干个 (至少  $\geq 3$ ), 于是  $y$  不是最优解;

综上, 只能  $x_1 = y_1$ ; 类似可以继续证得  $x_2 = y_2$ ,  $x_3 = y_3$ ,  $x_4 = y_4$ ;  
 $\therefore X$  是最优解

# 找钱问题

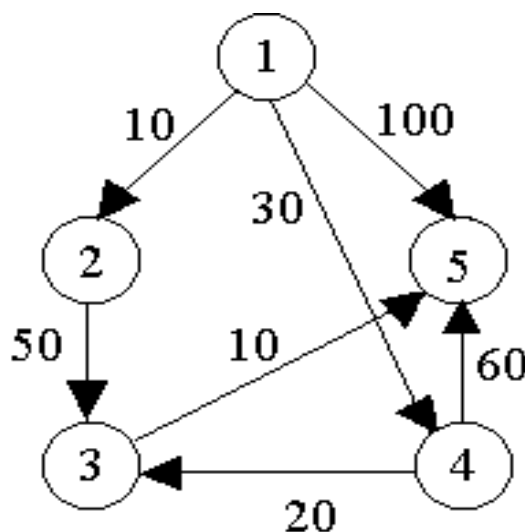
## □ 思考：

如果硬币面值改为1分、5分和1角1分三种，要找给顾客的是1角5分，是否可以用贪心算法？

# 单源最短路径

## □ 问题描述:

给定带权有向图 $G=(V,E)$ ，其中每条边的权是非负实数。另外，还给定 $V$ 中的一个顶点，称为**源**。现在要计算从源到所有其它各顶点的最短**路长度**。这里路的长度是指路上各边权之和。该问题通常称为**单源最短路径问题**。



**求解：**计算顶点1(源)到所有其他顶点之间的最短路径。

# 单源最短路径

## □ 迪杰斯特拉(Dijkstra)算法

**基本思想：**设置顶点集合S并不断地作**贪心选择**来扩充这个集合。

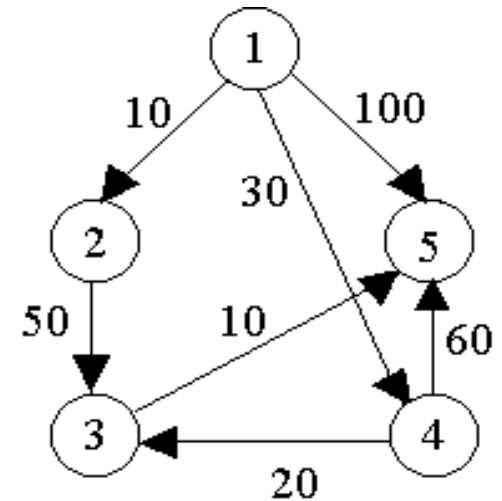
一个顶点属于集合S当且仅当从源到该顶点的最短路径长度已知。

## □ 步骤：

- ① 初始时，S中仅含有源。设u是G的某一个顶点，把从源点到u且中间只经过S中顶点的路称为**从源点到u的特殊路径**，并用**数组dist**记录当前每个顶点所对应的最短特殊路径长度。
- ② 每次从V-S中取出具有最短特殊路长度的顶点u (**贪心策略**)，将u添加到S中，**同时对数组dist作必要的修改 (why?)**。
- ③ 直到S包含了所有V中顶点，此时，dist就记录了从源到所有其它顶点之间的最短路径长度。

# 单源最短路径

例如，对右图中的有向图，应用迪杰斯特拉算法计算从源顶点1到其它顶点间最短路径的过程列如下表所示。



迭代	S	u	dist[2]	dist[3]	dist[4]	dist[5]
初始	{1}	-	10	maxint	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60



# 单源最短路径

## ● 算法描述如下:

```
Dijkstra( int n, int v, Type dist[], int prev[], Type ** c)
{ // 单源最短路径问题的迪杰斯特拉算法,
  bool s[maxint];
  for ( int i = 1; i <= n; i++) {
    dist[i] = c[v][i];
    s[i] = false;
    if ( dist[i] == maxint) prev[i] = 0;
    else prev[i] = v;
  }
  dist[v] = 0; s[v] = true;
  for ( int i = 1; i < n; i++) {
    int temp = maxint;
    int u = v;
    for ( int j = 1; j <= n; j++)
      if ( (!s[j]) && (dist[j] < temp)) { u = j; temp = dist[j]; }
    s[u] = true;
    for ( int j = 1; j <= n; j++)
      if ( (!s[j]) && (c[u][j] < maxint)) {
        Type newdist = dist[u] + c[u][j];
        if ( newdist < dist[j] ) { dist[j] = newdist; prev[j] = u; }
      }
  }
}
```

其中:

$c[i][j]$  表示边  $(i,j)$  的权,  $n$  是顶点个数,  $v$  表示源,  
 $dist[i]$  表示当前从源到顶点  $i$  的最短特殊路径长度

思考: 本算法只是给出了从源顶点到其他顶点间的  
最短路径长度, 并没有记录相应的最短路径。  
该如何修改才可以记录相应的最短路径?



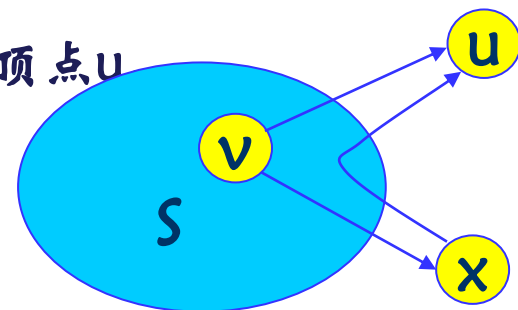
# 单源最短路径

## □ 算法的运算时间：

对于一个具有 $n$ 个顶点和 $e$ 条边的带权有向图，如果用带权邻接矩阵表示这个图，那么Dijkstra算法的主循环体需要 $O(n)$ 时间。这个循环需要执行 $n-1$ 次，所以完成循环需要 $O(n^2)$ 时间。算法的其余部分所需要时间不超过 $O(n^2)$ 。

# 单源最短路径

- **贪心策略为：**从 $V-S$ 中选择具有最短特殊路径的顶点 $u$ ，从而确定从源到 $u$ 的最短路径长度 $\text{dist}[u]$ 。



- **贪心选择性质证明：**

**证明：(反证法)** 即证明从源到 $u$ 没有更短的其他路径。假设存在一条从源到 $u$ 且长度比 $\text{dist}[u]$ 更短的路，设该路初次走出 $S$ 之外到达顶点为 $x \notin V-S$ ，然后徘徊于 $S$ 内外若干次，最后离开达到 $u$ ，如上图所示。在该路径上，分别记 $d(v,x)$ ， $d(x,u)$ 和 $d(v,u)$ 为顶点 $v$ 到顶点 $x$ ，顶点 $x$ 到顶点 $u$ 和顶点 $v$ 到顶点 $u$ 的路长，那么

$$\text{dist}[x] \leq d(v,x) \quad d(v,x) + d(x,u) = d(v,u) < \text{dist}[u]$$

利用边权的非负性，可知 $d(x,u) \geq 0$ ，从而推得 $\text{dist}[x] < \text{dist}[u]$ 。

此为矛盾。这就证明 $\text{dist}[u]$ 是从源到顶点 $u$ 的最短路径长度。

# 最小生成树

## □ 问题描述:

设  $G = (V, E)$  是无向连通带权图，即一个网络。E 中每条边  $(v, w)$  的权为  $c[v][w]$ 。如果  $G$  的子图  $G'$  是一棵包含  $G$  的所有顶点的树，则称  $G'$  为  $G$  的生成树。生成树上各边权的总和称为该生成树的耗费。在  $G$  的所有生成树中，耗费最小的生成树称为  $G$  的最小生成树 (MST: minimum-cost spanning tree)。

# 最小生成树的应用

- **应用实例：通信线路设计、电子线路设计等。**在设计通信网络时，用图的顶点表示城市，用边 $(v,w)$ 的权 $c[v][w]$ 表示建立城市 $v$ 和城市 $w$ 之间的通信线路所需的费用，则最小生成树就给出了建立通信网络的最经济的方案。
- **应用实例：县县通高速、万家灯火通电等。**在偏僻的村庄，村户都散落在村庄四处，需要供电建设，使用一条电缆将每家每户连接起来，由于村户距离间隔远，需要考虑到成本问题。
- **其他的应用实例。。。。**

# 最小生成树

## □ 最小生成树性质(MST: minimum-cost spanning tree)

设 $G=(V,E)$ 是连通带权图,  $U$ 是 $V$ 的真子集。如果 $(u,v) \in E$ , 且 $u \in U$ ,  $v \in V-U$ , 且在所有这样的边中,  $(u,v)$ 的权 $c[u][v]$ 最小, 那么一定存在 $G$ 的一棵最小生成树, 它以 $(u,v)$ 为其中一条边。这个性质有时也称为**MST性质**。

用贪心算法设计策略可以设计出构造最小生成树的有效算法。常用的构造最小生成树的**Prim算法**和**Kruskal算法**都可以看作是应用贪心算法设计策略的例子。它们做贪心选择的方式不同, 但是都利用了**最小生成树性质**。

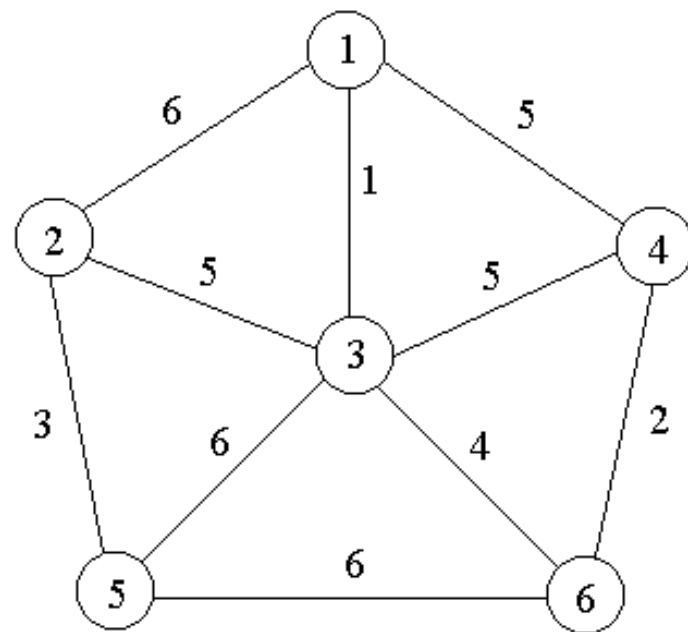
# 最小生成树—Prim算法

## □ Prim算法基本思想:

设 $G = (V, E)$ 是连通带权图,  $V = \{1, 2, \dots, n\}$ , Prim算法首先置  $S = \{1\}$ , 然后, 只要 $S$ 是 $V$ 的真子集, 就作贪心选择: 选取满足条件  $i \in S, j \in V-S$ , 且 $c[i][j]$ 最小的边, 将顶点 $j$ 添加到 $S$ 中。这个过程一直进行到 $S = V$ 时为止。在这个过程中选取到的所有边恰好构成 $G$ 的一棵最小生成树。

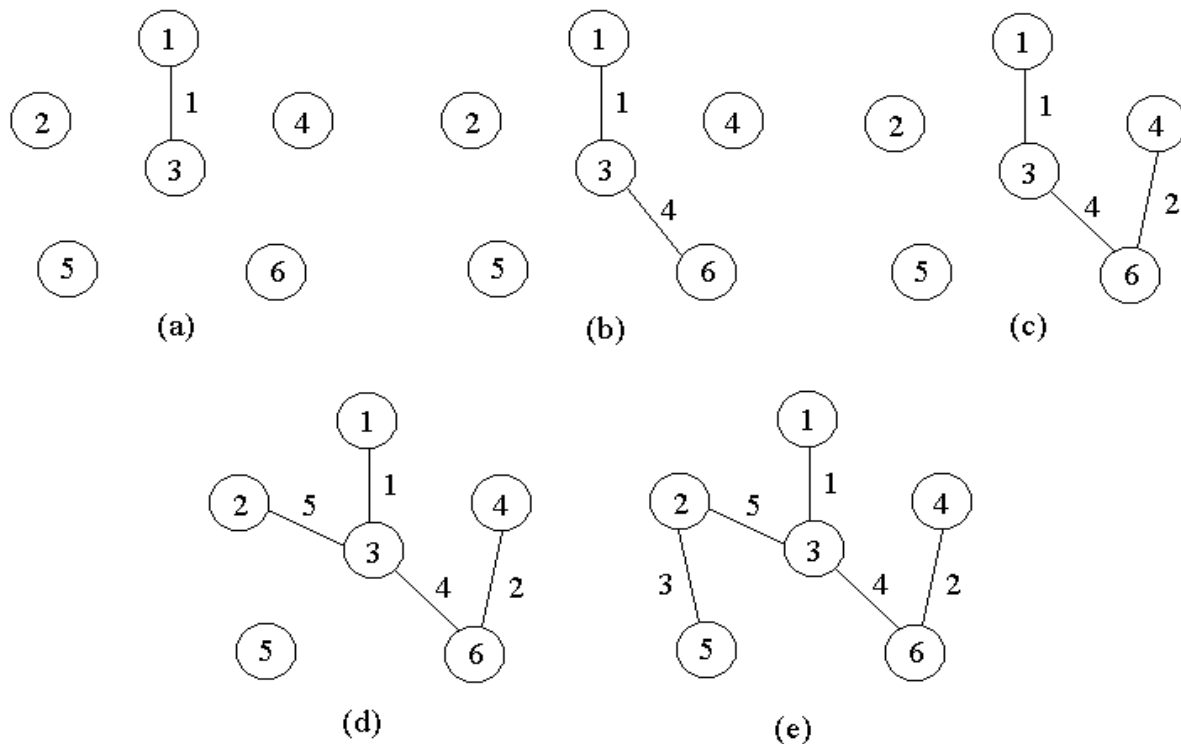
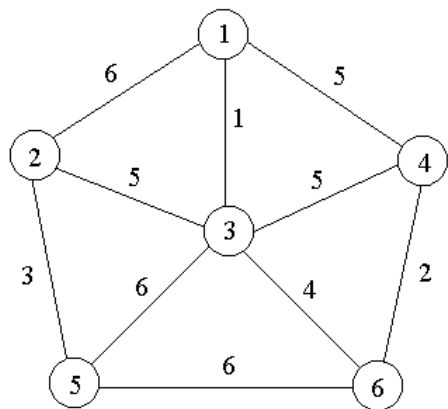
# 最小生成树—Prim算法

- 利用最小生成树性质和数学归纳法容易证明，上述算法中的边集合 $T$ 始终包含 $G$ 的某棵最小生成树中的边。因此，在算法结束时， $T$ 中的所有边构成 $G$ 的一棵最小生成树。
- 例如，对于右图中的带权图，按Prim算法选取边的过程？





# 最小生成树—Prim算法



□ Prim算法：参见教材

□ 时间复杂度： $O(V \lg V + E \lg V) = O(E \lg V)$

# 最小生成树—Kruskal算法

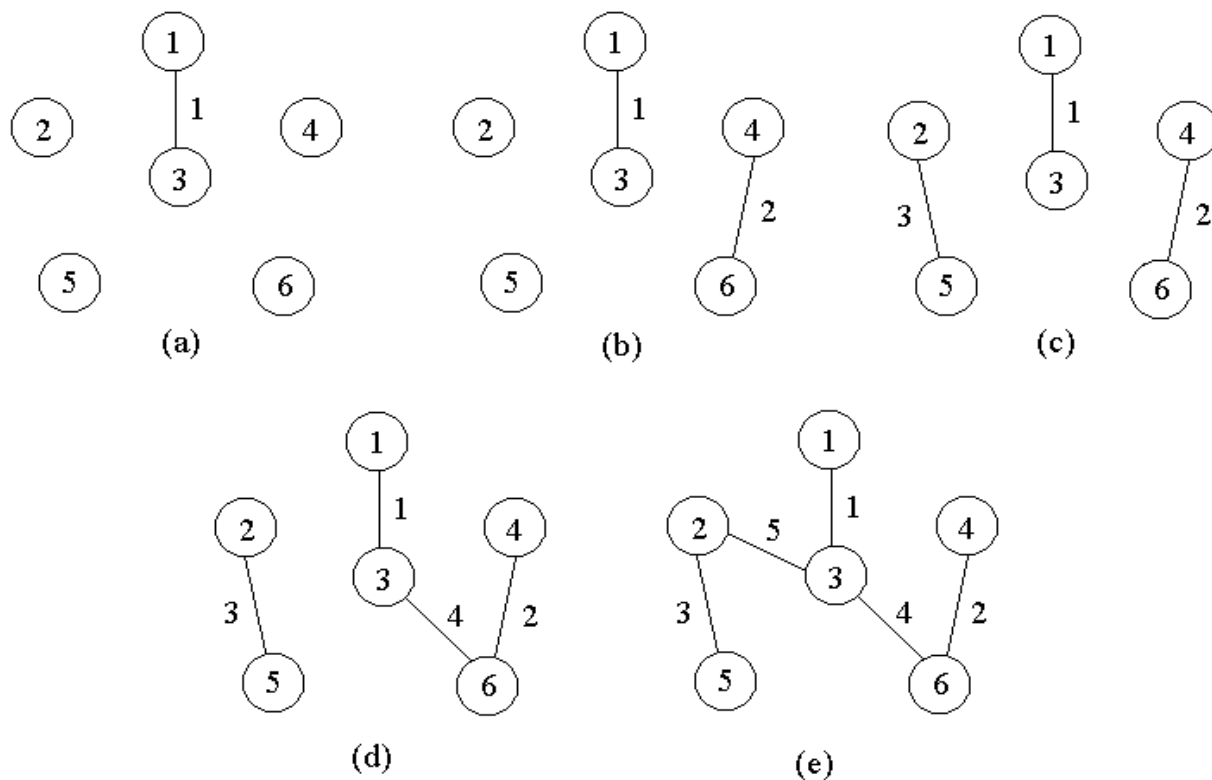
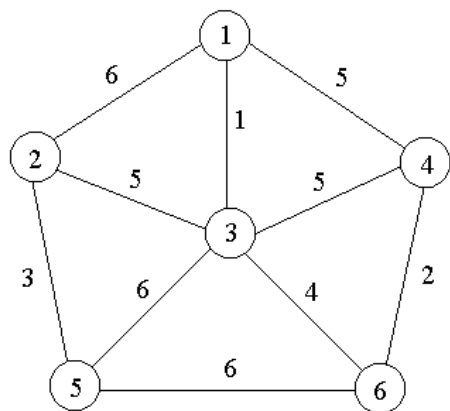
## □ 基本思想：

首先将 $G$ 的 $n$ 个顶点看成 $n$ 个孤立的连通分支。将所有的边按权从小到大排序。然后从第一条边开始，**按照边权递增的顺序查看每一条边**，并按下述方法连接2个不同的连通分支：

- 当查看到第 $k$ 条边 $(v,w)$ 时，如果端点 $v$ 和 $w$ 分别是当前2个不同的连通分支 $T_1$ 和 $T_2$ 中的顶点时，就用边 $(v,w)$ 将 $T_1$ 和 $T_2$ 连接成一个连通分支，然后继续查看第 $k+1$ 条边；
- 如果端点 $v$ 和 $w$ 在当前的同一个连通分支中，就直接再查看第 $k+1$ 条边。这个过程一直进行到只剩下一个连通分支时为止。

# 最小生成树—Kruskal算法

□ 对连通带权图，Kruskal算法得到的最小生成树上的边，如图所示。



# 最小生成树—Kruskal算法

## □ 实现细节：

关于集合的一些基本运算可用于实现Kruskal算法。按权的递增顺序查看等价于对优先队列执行Remove-Min运算，可以用堆实现这个优先队列。 对一个由连通分支组成的集合不断进行修改，需要用到抽象数据类型并查集Union-Find所支持的基本运算。

## □ Kruskal算法：参见教材

## □ 时间复杂度： $O(E \log E)$

当  $|E| > |V|^2$  时，Kruskal算法比Prim算法差；

当  $|E| < |V|^2$  时，Kruskal算法却比Prim算法好得多。



# 贪心算法的总结

- (1) 动态规划是先分析子问题，再做选择；而贪心算法的法则是先做贪心选择，做完选择后，生成了子问题，然后再去求解子问题。
  - ✓ 解动态规划问题一般是自底向上，从小子问题处理至大子问题。贪心算法所做的当前选择可能要依赖于已经做出的所有选择，但不依赖于有待于做出的选择或子问题的解。因此，贪心算法通常是自顶向下地做出贪心选择，不断地将给定的问题实例归约为更小的问题。贪心算法划分子问题的结果，通常是仅存在一个非空的子问题。
- (2) 动态规划每一步可能会产生多个子问题，而贪心算法每一步只会产生一个子问题；
- (3) 从特点上看，动态规划是 **自底向上** 解决问题，而贪心算法是 **自顶向下** 解决问题。

# 贪心算法的总结

- 适合用贪心法的问题应具有最优子结构性质：原问题的最优解包含其子问题的最优解。
  - ✓ 背包问题其原问题的最优解是在背包的容积的限定下利润达到最大，实际上是一个单位容积利润最大的问题。它包含子问题的最优。
  - ✓ 带限期的作业调度，是在限期的约束下，利润最大。子问题得最优符合这一原则。
- 但不是所有的问题都能找到贪心算法。例如，0/1背包问题，即在背包问题多了一个限制： $x_i \in \{0,1\}$ ，物品*i*不能拆开。
- 对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的一个整体最优解。-----如何证明？
- 首先考察问题的一个整体最优解，并证明可修改这个最优解，使其贪心选择开始。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。



谢谢！

---

**Q & A**

**作业： 5.1, 5.2**