算法基础

主讲人：　庄连生

Email: { ***lszhuang@ustc.edu.cn*** **}**

Spring 2018**，USTC**

University of Science and Technology of China

第十讲 贪心算法

内容提要：

- 理解贪心算法的概念
- 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
- 通过范例学习动态规划算法设计策略

10.1 活动安排问题

- 当一个问题具有最优子结构性质时，可用动态规划法求解，但有时用贪心算法求解会更加的简单有效。

- 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。如单源最短路经问题，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。

10.1 活动安排问题

- 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。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**。也就是说，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。

10.1 活动安排问题

- （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：

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

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。

10.1 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。

- **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：

$$c[i,j]=\begin{cases} 0 & Sij=\phi \\ \max\limits_{i<k<j}\{c[i,k]+c[k,j]+1\} & Sij\neq\phi \end{cases}$$

10.1 活动安排问题

- 问题：

  k有j-i-1种选择，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！

10.1 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。

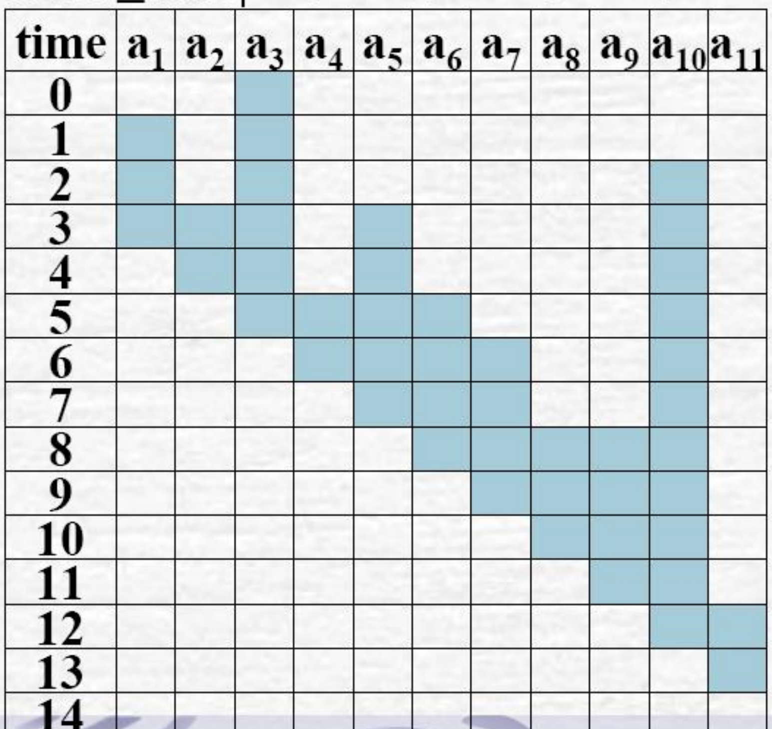
例：设待安排的11个活动的开始时间和结束时间按结束时间的非减序排列如下：

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

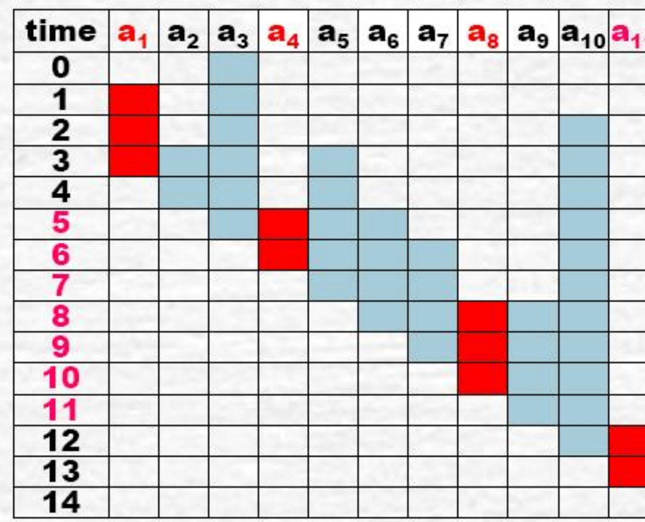
10.1 活动安排问题

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
$start\_time_i$	1	3	0	5	3	5	6	8	8	2	12
$finish\_time_i$	4	5	6	7	8	9	10	11	12	13	14



相容活动：$\{a_3, a_9, a_{11}\}$, $\{a_1,a_4,a_8,a_{11}\}$, $\{a_2,a_4,a_9,a_{11}\}$



10.1 活动安排问题

```
Recursive-Activity-Selector(s, f, i, j)
{
①   m ← i+1;
②   while m<j  and sm<fi
③         do   m ← m+1
④   if m<j
⑤         then  return {am} U Recursive-Activity-Selector(s, f, m, j)
⑥          else    return  ϕ
}
```

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)。

10.1 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，几乎就是“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 ← $\{a_1\}$
  i ← 1　　　　　　//下标i记录了最近加入A的活动a_i
  for m ← 2 to n　　//寻找$S_{i,n+1}$中最早结束的兼容活动
    do if $s_m \geqslant f_i$
      then A ← A U $\{a_m\}$
        i ← m
  return A;
}

10.1 活动安排问题

- 贪心算法的正确性证明：

定理16.1：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，所以选择a_m将使子问题S_{mj}为唯一可能非空的子问题。

10.1 活动安排问题

定理16.1： 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，所以选择a_m将使子问题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$f_i \leqslant s_k < f_k \leqslant s_m < f_m$。$a_k$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。

10.1 活动安排问题

定理16.1： 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，所以选择a_m将使子问题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。

10.2 贪心算法的基本要素

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——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，就是要证明按贪心准则求得的解是全局最优解。

10.2 贪心算法的基本要素

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。

⑤ 设计出一个实现贪心策略的递归算法。

⑥ 将递归算法转换成迭代算法。

10.2 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。

10.2 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。

10.2 贪心算法的基本要素

- 0-1背包问题

  给定n种物品和一个背包。物品i的重量是Wi，其价值为Vi，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

  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≤i≤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。

10.2 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：

  (x1, x2, x3) = (0, 1, 1) = 220

  (x1, x2, x3) = (1, 1, 0) = 160

  (x1, x2, x3) = (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。 这正是该问题可用动态规划算法求解的另一重要特征。 实际上也是如此， 动态规划算法的确可以有效地解0-1背包问题。

10.2 贪心算法的基本要素

- 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。

Dynamic Programming	Greedy Algorithms
At each step, the choice is determined based on solutions of subproblems.	At each step, we quickly make a choice that currently looks best. -A local optimal (greedy) choice.
Sub-problems are solved first.	Greedy choice can be made first before solving further sub-problems.
Bottom-up approach	Top-down approach
Can be slower, more complex	Usually faster, simpler

10.3 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。

$$\max\sum_{i=1}^{n} v_i x_i \qquad x_i\text{为装入物品}i\text{的比例}$$

$$\begin{cases}\sum_{i=1}^{n} w_i x_i \le c \qquad w_i > 0 \\ 0 \le x_i \le 1 \qquad i = 1,2,\dots,n \\ v_i > 0,\ w_i > 0,\ c > 0\ i = 1,2,\dots,n\end{cases}$$

w_i为重量，c为背包容量

v_i为价值

v_i/w_i为价值率(单位重量价值)

- 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, 1/4)	16.5	24.5
②	(1, 2/15, 0)	20	28.2
③	(0, 2/3, 1)	20	31
④	(0, 1, 1/2)	20	31.5 (最优解)

10.3 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)

10.3 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$v_1/w_1 \geqslant v_2/w_2 \geqslant \cdots \geqslant v_n/w_n$

    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:** $T(n) = O(n\lg n)$

10.3 小数背包问题

- 贪心选择的最优性证明

定理： 如果$v_1/w_1 \geqslant v_2/w_2 \geqslant \cdots \geqslant v_n/w_n$，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，从而证明了贪心解是最优解。

10.3 小数背包问题

- 证明：设$(x_1, \ldots, x_n)$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$X = (1,\ldots,1,x_j,0,\ldots,0) \quad x_j \neq 1, 1 \leq j \leq n$。下证$X$就是最优解。设问题的最优解$Y = (y_1,\ldots,y_n)$，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$Z = (z_1,\ldots,z_n)$，并使Z仍为最优解。将y_k增加到x_k，从$(y_{k+1},\ldots,y_n)$中减同样的重量使总量仍是c。即，

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

10.3 小数背包问题

$$\therefore \sum_{i=1}^{n} w_i z_i = \sum_{i=1}^{k-1} w_i z_i + w_k z_k + \sum_{i=k+1}^{n} w_i z_i$$

$$= \sum_{i=1}^{k-1} w_i y_i + w_k z_k + \left(\sum_{i=k+1}^{n} w_i y_i - w_k (z_k - y_k) \right) = \sum_{i=1}^{n} w_i y_i = c$$

$$\therefore \sum_{i=1}^{n} v_i z_i = \sum_{i=1}^{n} v_i y_i + (z_k - y_k) v_k - \sum_{i=k+1}^{n} (y_i - z_i) v_i \quad // \because Z\text{由}Y\text{变得的}$$

$$= \sum_{i=1}^{n} v_i y_i + (z_k - y_k) w_k v_k / w_k - \sum_{i=k+1}^{n} (y_i - z_i) w_i v_i / w_i$$

$$\geq \sum_{i=1}^{n} v_i y_i + \left[(z_k - y_k) w_k - \sum_{i=k+1}^{n} (y_i - z_i) w_i \right] v_k / w_k \quad // \text{利用} v_i / w_i \downarrow$$

$$= \sum_{i=1}^{n} v_i y_i$$

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

10.3 小数背包问题

- 特殊的0-1背包问题：

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

```
0-1-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;
}
```

10.4 最优装载

- **问题的描述:**

轮船载重为c，集装箱重量为$w_i(i=1,2,\cdots,n)$，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:**

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

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

10.4 最优装载

- 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 $n=8$, $[w_1,w_2,\cdots,w_8]=[100,200,50,90,150,50,20,80]$, $c=400$。

  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$[x_1,x_2,\cdots,x_8]=[1,0,1,1,0,1,1,1]$，且$\sum x_i = 6$

- **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]] ≤w[t[i+1]], 1 ≤  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(nlgn)

10.4 最优装载

- **贪心性质证明：**

不失一般性，假设货箱都排好序，即$w_i \leqslant w_{i+1}$ $(1 \leqslant i \leqslant n)$。

令$x=[x_1,\cdots,x_n]$为用贪心算法获得的解，$y=[y_1,\cdots,y_n]$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$\sum y_i$（$1 \leqslant i \leqslant n$）的值不变（即仍为最优解），从而证明了x为最优解。

10.5 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：

$$\min\left\{\sum_{i=1}^{4} x_i\right\}$$

$$s.t.\quad \begin{cases} \sum_{i=1}^{4} d_i x_i = A \\ x_i \text{为非负整数} \quad 1 \le i \le 4 \end{cases}$$

10.5 找钱问题

- 贪心策略

$$x_1 = \lfloor A/d_1 \rfloor \qquad x_3 = \lfloor (A - d_1x_1 - d_2x_2)/d_3 \rfloor$$

$$x_2 = \lfloor (A - d_1x_1)/d_2 \rfloor \qquad x_4 = A - d_1x_1 - d_2x_2 - d_3x_3$$

- 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;
 }
```

10.5 找钱问题

- 最优子结构性质证明：

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'

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

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

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

10.5 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$x_1=\lfloor A/d_1 \rfloor \Rightarrow x_1 \le A/d_1 < x_1+1 \Rightarrow d_1x_1 \le A <$

$d_1(x_1+1)$；因此，$\sum_{i=1}^{4} d_iy_i \ge d_1y_1 \ge d_1(x_1+1) > A$，产生矛盾；

如果$x_1>y_1$，则$10y_2+5y_3+1y_4 \ge 25$（否则，$25y_1+10y_2+5y_3+1y_4$

$<25(y_1+1) \le 25x_1+10x_2+5x_3+1x_4=A$，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≥ 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

10.5 找钱问题

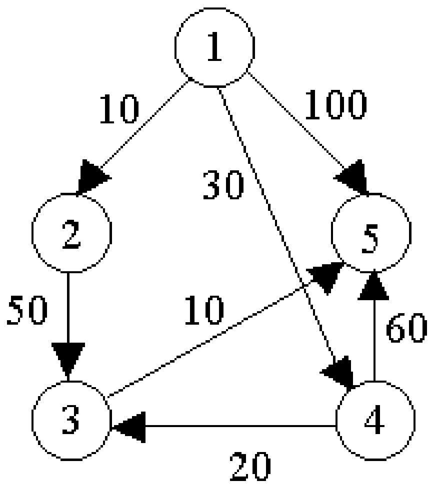
- **思考：**

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

10.6 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。这个问题通常称为单源最短路径问题。



如：计算顶点1（源）到所有其他顶点之间的最短路径。

10.6 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：

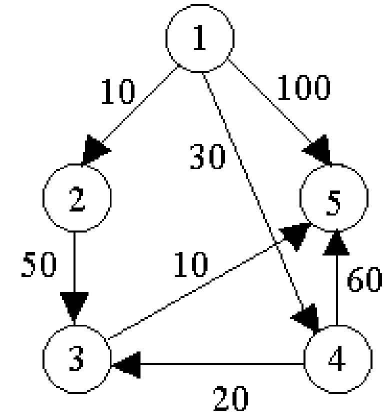
① 初始时，S中仅含有源。设u是G的某一个顶点，把从源到u且中间只经过S中顶点的路称为从源到u的特殊路径，并用数组dist记录当前每个顶点所对应的最短特殊路径长度。

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。

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。

10.6 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

10.6 单源最短路径

□ 算法描述：

```
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？

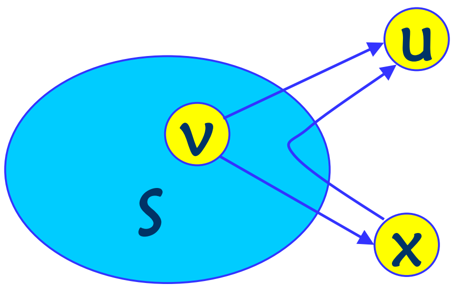
10.6 单源最短路径

- 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$O(n^2)$。

10.6 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dist[u]更短的路，设这条路初次走出S之外到达顶点为x#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。

在这条路径上，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

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

利用边权的非负性，可知d(x,u)>=0，从而推得dist[x]<dist[u]。此为矛盾。这就证明了dist[u]是从源到顶点u的最短路径长度。

10.7 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**。

10.7 最小生成树

- **应用实例：通信线路设计、电子线路设计等**

网络的最小生成树在实际中有广泛应用。**例如**，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。

10.7 最小生成树

最小生成树性质：

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。尽管这2个算法做贪心选择的方式不同，它们都利用了上面的**最小生成树性质**。

10.7 最小生成树—Prim算法

- Prim算法基本思想：

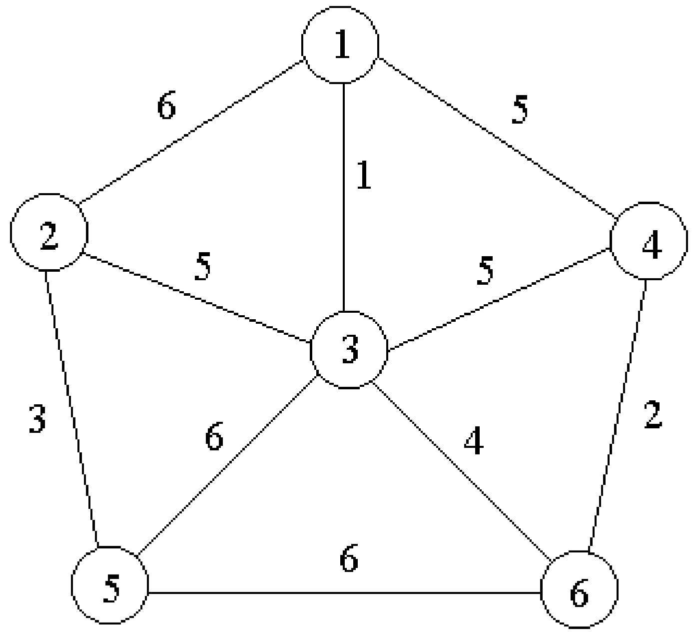
设$G=(V,E)$是连通带权图，$V=\{1,2,\ldots,n\}$，Prim算法首先置$S=\{1\}$，然后，只要S是V的真子集，就作如下的**贪心选择**：选取满足条件$i\in S$，$j\in V-S$，且$c[i][j]$最小的边，将顶点j添加到S中。这个过程一直进行到$S=V$时为止。

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**。

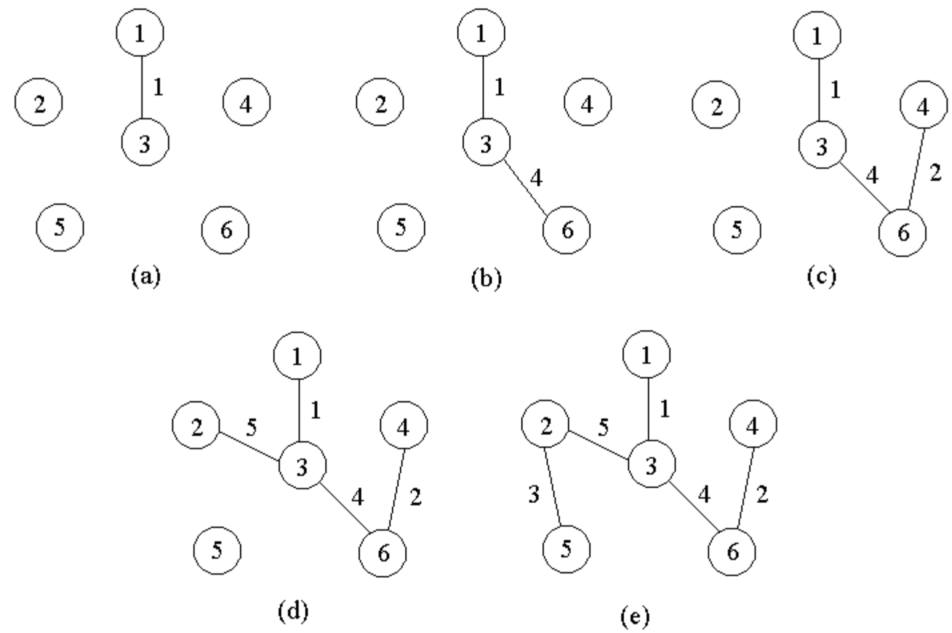
10.7 最小生成树—Prim算法

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**选取边的过程如下页图所示。



10.7 最小生成树—Prim算法



- Prim算法： 参见教材P351
- 时间复杂度： $O(V\lg V + E\lg V) = O(E\lg V)$

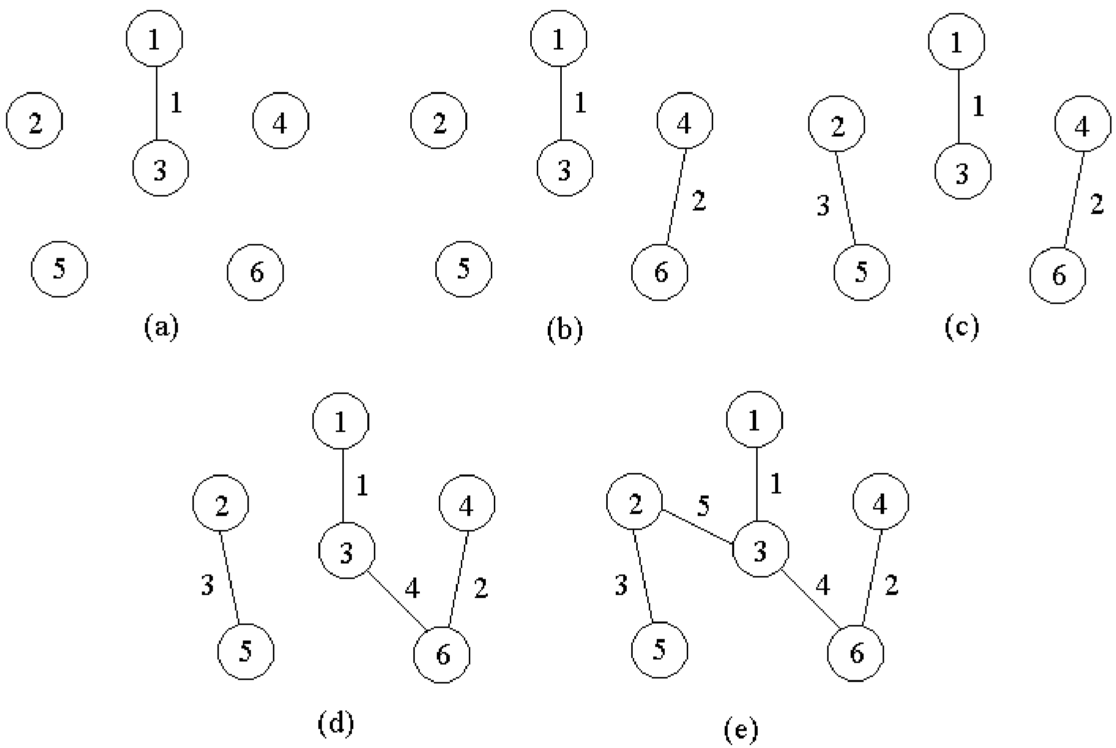
10.7 最小生成树—Kruskal算法

□ 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，依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T1和T2中的顶点时，就用边(v,w)将T1和T2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。

10.7 最小生成树—Kruskal算法

- **例如，**对前面的连通带权图，按Kruskal算法顺序得到的最小生成树上的边如下图所示。



10.7 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removeMin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**UnionFind所支持的基本运算。

- **Kruskal算法：** 参见教材P348

- **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。

谢谢！

Q & A