第一章 算法引论

- 1.5 渐近符号
  - 时间复杂度函数T(n)的简化表示-渐进意义下的阶
    - 算法A的复杂性在规模n很大时才有意义，即考虑n →∞时T(n)的性态。
    - 函数 f (n)的渐进函数 g(n): 当n→∞时
      (f(n) – g(n))/f(n) → 0，称g(n)是f(n) 的渐进复杂性。
    - 数学上称g(n)为f(n)当n →∞时的渐近表达式，一般取f(n)中略去低阶项后留下的主项，比f(n)简单。
    - 比较算法的渐近复杂性，只要确定出各自T(n)的阶即可，不必关心其包含的常数项 、低阶项。
    - 事实上，用的最多的是渐进上界O(g(n))。O忽略阶的系数，所以计算T时可认为所有操作时间单位都是1.

渐近符号

- 常用的渐近函数

函数	名称	函数	名称
1 Logn n nlogn	常数 对数 线性 n倍对数	n^2 n^3 2^n n!	平方 立方 指数 阶乘

渐近符号

- 渐近上界：f(n) = O(g(n))
  - 定义：存在正实数 c 和正整数 N, 使得当n>N时，
    $f(n) \le c \cdot g(n)$
  - 例子： const = O(1); 3n+1 = O(n); $10n^2+4n+3 = O(n^2)$;
    $6 \cdot 2^n + n^5 = O(2^n)$; $n \cdot \log n + n^2 = O(n^2)$; $3n + 2 = O(n^2)$
  - 注：1. 记号f(n)=O(g(n))不能写成g(n)=O(f(n));
  - 2. 最后一个是松散的上界，最好给出最小渐进上界。O(n)是它较好的界。
  - 大欧比率定理：
  - 如果极限 lim(f(n)/g(n))存在，则 f(n) = O(g(n)) 的充要条件是：存在正实数 c , 使得 lim (f(n)/g(n)) ≤ c .
- 小o符号：大O定义中“存在c”改为“任给c”。
  - f(n)=o(g(n)表明f(n)一定低于g(n)的阶，大O可能等于。故o→O。

渐近符号

- 定理：对于任意正实数 χ 、ε 和常数 c ， 下面的不等式成立(常用估计渐近上界的几个不等式)：
  - 1. 存在正整数 N, 使得对于任何n>N, 有 $(c + \log n)^{\chi} < (\log n)^{\chi+\varepsilon}$;
  - 2. 存在正整数 N, 使得对于任何n>N, 有 $(\log n)^{\chi} < n^{\varepsilon}$;
  - 3. 存在正整数 N, 使得对于任何n>N, 有 $(c + n)^{\chi} < n^{\chi+\varepsilon}$;
  - 4. 存在正整数 N, 使得对于任何n>N, 有 $n^{\chi} < (1+ \varepsilon)^{n}$;
  - 5. 组合：对于常数 γ ，当 n 充分大时，有 $n^{\chi}(\log n)^{\gamma} < n^{\chi+\varepsilon}$ 。
- 举例
  - $n^3 + n^2 + \log^{10} n = O(n^3)$;
  - $n^4 + n^{2.5} \log^{20} n = O(n^4)$;
  - $2^n n^4 \log^3 n + 2^n n^5 / \log^3 n = O(2^n n^5)$;
  - $n^5 = O(2^n)$; $n2^n = O(n!)$

渐近符号

- 渐近下界: $f(n) = \Omega(g(n))$
  - 定义：存在正实数 c 和正整数 N, 使得当n>N时，$f(n) \geq c \cdot g(n)$
  - 大欧米茄比率定理：如果极限 lim(f(n)/g(n))存在，则$f(n) = \Omega(g(n))$ 的充要条件是：存在正实数 c , 使得 $\lim(f(n)/g(n)) \geq c$.
  - 注：容易发现一个渐近下界，但人们更关注的是找到最大的渐进下界。
- 渐近同阶： $f(n) = \Theta(g(n))$
  - 定义：指函数g(n)既是f (n)的渐近上界，又是f (n)的渐近下界。此时，如果极限 lim(f(n)/g(n))存在的话，应该是一个正实数。
- 例子
  - $3n+2= \Theta(n)$; $1.5n^2-4n+100 = \Theta(n^2)$;
    $5 \cdot 2^n +3n \log n = \Theta(2^n)$; $n!= \Omega(2^n)$;
  - 一般地，函数主项即是该函数的渐进同阶。

第三章 分治算法

- 3.1 分治策略的基本思想
  - 一个熟悉的例子：折半搜索(二分查找）

```
BiFind(a,n)    //在数组a[1..n]中搜索x，数组满足a[1] ≤ a[2] ≤ … ≤ a[n]。
    //如果找到x，则返回所在位置（数组元素的下标），否则返回 –1
  global a[1..n], n;    integer left,right,middle;
  left:=1; right:=n;
  while left ≤ right do
     middle:=(left+right)/2;
     if x=a[middle] then return(middle); end{if}
     if x>a[middle] then left:=middle+1;
     else right:=middle-1;
    end{if}
  end{while}
  return(–1);          //未找到x
end{BiFind}
```

分治策略的基本思想

- 算法的复杂度分析
  - 通过x与数组a[]中元素的一次比较，a[]的范围至少减少一半，因此关键操作满足：

$$\begin{cases} T(n)=T(\lfloor \frac{n}{2} \rfloor)+1 \\ T(1)=1 \end{cases}$$

  - 不妨设(为什么？)$n=2^k$，有
  - $T(n)= T(n/2)+1 =T(n/4)+2=T(n/2^2)+2 =\ldots =T(1)+k$
  - 其中$2^k=n$，$k=\log n$
  - $T(n)=1+\log n= \Theta(\log n)$
  - 比较：x与a[]一一比较，$T(n)=O(n)$

分治策略的基本思想

- 分治策略的基本思想
  - 将规模为n的问题规约为规模减小的一个或多个子问题，分别求解每个子问题，然后把子问题的解综合，得到原问题的解。
  - 分治算法的一般描述

    Divide-and-conquer(p)

    1. if |p|≤c then S(p)　　//S(p)代表直接求解
    2. divide p into $p_1,p_2,\ldots p_k$
    3. 　for i=1 to k do
    4. 　　yi=divide-and-conque(p_i)　　//递归求解每个子问题
    5. 　Return Merge($y_1,y_2,\ldots,y_k$)　　//把子问题的解进行综合

分治策略的基本思想

- 例2：求n元数组的最小、最大元素

```
MaxMin(i,j,fmax,fmin) //A[1:n]是n元数组，
//参数i，j：1≤i≤j≤n,使用该过程将数组
// A[i..j]中的最大最小元分别赋给fmax和fmin。
global n, A[1..n];
integer i, j;
if i=j then
else   //子数组A[i..j]中的元素多于两个
  mid:=⌊(i+j)/2⌋;
  MaxMin(i, mid, lmax, lmin);
  MaxMin(mid+1, j, rmax, rmin);
  fmax:=max(lmax, rmax);
  fmin:=min(lmin, rmin);
end{if}
end{MaxMin}
```

```
 fmax:=A[i]; fmin:=A[i];
//子数组A[i..j]中只有一个元素
elseif i=j-1 then
//子数组A[i..j]中只有两个元素
     if A[i]<A[j] then
       fmin:=A[i]; fmax:=A[j];
     else fmin:=A[j]; fmax:=A[i];
  end{if}
```

分治策略的基本思想

- T(n)的复杂度分析
  - 递归关系

$$T(n)=\begin{cases}0 & n=1\\ 1 & n=2\\ T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+2 & n>2\end{cases}$$

  - 设$n=2^k$，有
    - T(n)=2T(n/2)+2
    - =2(2T(n/2)+2)+2=4T(n/2)+4+2
    - …
    - $=2^{k-1}T(2)+\sum_{1\le j\le k-1}2^j$
    - $=2^{k-1}+2^k-2$
    - =3n/2-2

对比直接比较法：
fmax=a[1]
fmin=a[1]
for i=2 to n
  fmax,fmin比较a[i]
endfor
T(n)=2(n-1)

第三章 分治算法

- 3.2 排序算法(排序问题时间下界nlogn：第8章)
  - 插入排序算法：T(n)= $\sum_{1\le i\le n-1} i$ =n(n-1)/2= **Θ(n^2)**

```
proc InSort(a, n)
for i from 2 to n do
   x:=a[i];   integer j;
   for j from i-1 by -1 to 1 do
     if x<a[j] then a[j+1]:=a[j];
     else  break;
     end{if}
   end{for}
   a[j+1]:=x;
end{for}
end{InSort}
```

排序算法

- 归并排序算法

```
proc MergeSort(low, high)
 // A[low .. high]是一个全程数组，含有 high-low+1个
 // 待排序的元素。
integer  low, high;
if  low < high  then
      mid:=⌊(low+high)/2⌋ //求当前数组的分割点
      MergeSort(low, mid)  //将第一子数组排序
      MergeSort(mid+1, high)  //将第二子数组排序
      Merge(low, mid, high)  //归并两个已经排序的子数组
end{if}
end{MergeSort}
```

排序算法

- 归并排序算法-合并过程

```
proc Merge(low, mid, high)   //已知A[low .. high], 由两部分已经排好
  //序的子数组构成： A[low .. mid]和A[mid+1 .. high].本程序把它
  //们合并成一个整体排好序的数组，再存于数组A[low .. high].
   integer h, i, j, k, low, mid, high;
   global A[low .. high];
   local B[low .. high];        //借用临时数组B
   h:=low, i:=low, j:=mid+1; // h, j拣取游标, i向B存放元素的游标
   while h≤mid and j≤high do  //当两个集合都没有取尽时
     if A[h]≤A[j] then B[i]:=A[h], h:=h+1;
       else B[i]:=A[j], j:=j+1;
     end{if}
     i:＝i+1;
   end{while}
```

排序算法

- 合并过程proc MergeSort(low, high)续

```
if  h>mid then //当第一子组元素被取尽，而第二组元素未被取尽时
  for k from j to high do
    B[i]:=A[k]; i:=i+1;
  end{for}
else     //当第二子组元素被取尽，而第一组元素未被取尽时
  for k from h to mid do
    B[i]:=A[k]; i:=i+1;
  end{for}
end{if}      //将临时数组B中元素再赋给数组A
for k from low to high do
  A[k]:=B[k];
end{for}
end{Merge}
```

排序算法

- 归并排序算法的时间复杂度
  - 由于合并过程所用时间与n成正比：cn，c是一个正数，则有

$$T(n)=\begin{cases} a & n=1 \\ 2T(n/2)+cn & n>1 \end{cases}$$

  - 令$n=2^k$，则

$$\begin{aligned} T(n) &= 2(2T(n/4)+cn/2)+cn \\ &= 4T(n/4)+2cn \\ &\cdots\cdots \\ &= 2^k T(1)+kcn \\ &= an+cn\log n \end{aligned}$$

  - 对于一般的整数n，可以假定 $2^k \le n \le 2^{k+1}$，于是$T(n) \le T(2^{k+1})$

$T(2^{k+1})=2^{k+1}T(1)+(k+1)c2^{k+1}=2na+2cn(\log n+1)$

$=(2a+2c)n+2cn\log n$ ⟹ $T(n)=O(n\log n)$

排序算法

- 快速排序算法
  - 算法思想：将A[1..n]划分成B[1..p]和B[p+1..n]，使得B[1..p]中的元素均不大于B[p+1..n]中的元素，然后分别对它们排序，最后接起来即可。
  - 划分：选定A中一个元素，将A中的所有元素与之比较，小于等于的元素构成B[1..p]，大于的元素构成B[p+1..n]。
  - 划分A[1..n]比较n-1次

```
proc Partition(m,p) // 被划分的数组是A[m,p-1]，选定做划分元素的是v:=A[m]
 integer m, p, i;      global A[m ..p-1];
  v:=A[m]; i:=m;
  loop
       loop i:=i+1; until A[i]>v; end{loop}
         //自左向右查
       loop p:=p-1; until A[p]≤v; end{loop}
         //自右向左查
       if i<p then   //交换A[i]和A[p]的位置
         Swap(A[i],A[p]);
       else  go to *;
       end{if}
  end{loop}
  *: A[m]:=A[p]; A[p]:= v; // 划分元素在位置p
end{Partition}
```

排序算法

- 例：数组A[1:9]=[65, 50,75,80,85,60,55, 70,45]划分过程

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	*i*	*p*
65	50	75	80	85	60	55	70	45	+∞	3	9
65	50	45	80	85	60	55	70	75	+∞	4	7
65	50	45	55	85	60	80	70	75	+∞	5	6
65	50	45	55	60	85	80	70	75	+∞	5	5
i=5 = p=5											
65	50	45	55	60	85	80	70	75	+∞	1	5
60	50	45	55	|65|	85	80	70	75	+∞		

排序算法

- 快速排序算法主程序

```
proc QuickSort(p,q) //将数组A[1..n]中的元素A[p], A[p+1], … , A[q]
 // 按不降次序排列，并假定A[n+1]是一个确定数，且大于A[1..n]
 //中所有的数。划分后j成为划分元素的位置。
 integer p,q;
 global n, A[1..n];
if p<q then
  j:=q+1;
  Partition(p,j);
  QuickSort(p,j-1);
  QuickSort(j+1,q);
 end{if}
end{QuickSort}
```

快速排序最坏复杂度：划分后一个规模为0，一个n-1,则

$T(n)=T(n-1)+n-1$

$=T(n-2)+n-2+n-1$

$=1+2+\ldots+n-1$

$=(n-1)(n-2)/2$

$=O(n^2)$

排序算法

- 快速排序算法的平均复杂度
  - 设划分元素v是A[m, p-1]中第i小元素的概率均为 1/(p-m)，1≤i≤p-m，因而留下待排序的两个子组为A[m..j-1]和A[j+1..p-1]的概率是1/(p-m)，m≤j≤p-1。由此得递归关系式：
  - $T(n)=n-1+\frac{1}{n}\sum_{1\le k\le n}(T(k-1)+T(n-k))$ ，T(0)=T(1)=0

    n-1是划分A[1..n]所需要的比较次数。
  - nT(n)=n(n-1)+2(T(0)+T(1)+…+T(n-1))，用n-1替换n得

    （n-1)T(n-1)=(n-1)(n-2)+2(T(0)+T(1)+…+T(n-2))，相减得

    nT(n)-(n-1)T(n-1)=2(n-1)+2T(n-1) ⟹

    nT(n)=2(n-1)+(n+1)T(n-1) ⟹

    T(n)/(n+1)=2(n-1)/n(n+1)+T(n-1)/n ≦T(n-1)/n+2/n，递推得

    $T(n)/(n+1)\le T(1)/2+2\sum_{2\le i\le n}\frac{1}{i}=2\log n$ 因为 $\sum_{2\le i\le n}1/i<\int_1^n\frac{dx}{x}=\ln n$

    所以 T(n) = 2(n+1)logn = O(nlogn)

第三章 分治算法

- 3.3 选择问题
  - 问题描述：确定数组A[1..n]的第k小元素。
  - 排序法
    - 使用某种排序算法将A按不降次序排序
    - 从排好序的数组中检出第k个元素
    - 时间复杂度：最坏情况下至少是O(nlogn)。
      - 以比较为基础的排序算法最坏下界为 $T(n)=\Omega(n\log n)$ 。
      - 关于问题的计算复杂度(**不是算法的复杂度**)以及检索、排序和选择问题的时间复杂度下界放到第八章讨论）
      - 事实上选择问题的时间复杂度下界为O(n)。应该有比排序法更好的算法。

选择问题

- 试用划分法：平均复杂度O(n)

```
proc PartSelect(A, n, k) //在数组A[1..n]中找第k小元素 t，并将其
   //存放于位置k，即A[k]=t。
    integer n , k, m, r, j;
    m:=1; r:=n+1; A[n+1]:= +∞;
    loop
       j:= r ;
       Partition(m,j);
       case:
          k=j : return // 返回j,当前数组的元素A[j]是第j小元素
          k<j : r:=j; // j是新的下标上界
       else : m:=j+1; k:=k-j; //j+1是新的下标下界
       end{case}
    end{loop}
end{PartSelect}
```

- 遗憾的是，该算法最坏复杂度为$O(n^2)$:如A按递增有序，找第n小元素时，每次划分仅减少一个元素。
- $T(n)=T(n-1)+n-1$
- $=(n-1)(n-2)/2 =O(n^2)$

矩阵乘法

- Strassen 算法

$$P=(A_{11}+A_{22})(B_{11}+B_{22})$$
$$Q=(A_{21}+A_{22})B_{11}$$
$$R=A_{11}(B_{12}-B_{22})$$
$$S=A_{22}(B_{21}-B_{11})$$
$$T=(A_{11}+A_{12})B_{22}$$
$$U=(A_{21}-A_{11})(B_{11}+B_{12})$$
$$V=(A_{12}-A_{22})(B_{21}+B_{22})$$

$$C_{11}=P+S-T+V$$
$$C_{12}=R+T$$
$$C_{21}=Q+S$$
$$C_{22}=P+R-Q+U$$

$$T(n)=\begin{cases} b & n\le 2 \\ 7T(n/2)+an^2 & n>2 \end{cases}$$

$$T(n)=an^2(1+7/4+(7/4)^2+\cdots+(7/4)^{k-2})+7^{k-1}T(2)$$
$$=an^2\left(\frac{16}{21}\left(\frac{7}{4}\right)^{\log n}-\frac{4}{3}\right)+\frac{b}{7}(7)^{\log n}$$
$$=an^2\left(\frac{16}{21}(n)^{\log\frac{7}{4}}-\frac{4}{3}\right)+\frac{b}{7}(n)^{\log 7}$$
$$=\left(\frac{16a}{21}+\frac{b}{7}\right)n^{\log 7}-\frac{4a}{3}n^2$$
$$=\Theta\left(n^{2.81}\right)$$

目前最好的矩阵乘法的时间复杂度为：$O(n^{2.36})$

第三章 分治算法

- 3.5 最接近点对问题
  - 一维最近点对问题

直线上的n个点采用排序+扫描方法，时间复杂度为O(nlogn)。

  - 分治算法：

$$T(n)=\begin{cases} a, & n<4; \\ 2T(n/2)+cn, & n\geq 4 \end{cases}$$

T(n)= $O(n\log n)$

  - 蛮力算法：两两比较，

T(n)=(n-1)(n-2)/2=O(n^2)

- **proc ClosPair1**(S，d)
  //S是实轴上点的集合，参数d表示S中最
  //近对的距离
- **global** S,d; **integer** n; **float** m,p,q;
- n:=|S|;
- **if** n<2 **then** d:=∞; return(false); **end{if}**
- m:=S中各点坐标的中位数；
- //划分集合S
- S1:={x∈S| x≤m}; S2:={x∈S| x>m};
- ClosPair1(S1,d1);
- ClosPair1(S2,d2);
- p:=max(S1); q:=min(S2);
- d:=min(d1,d2,q-p);
- return(true);
- **end{ClosPair1}**

第三章 分治算法

- 3.6 快速Fourier变换
  - 连续函数a(t)的Fourier变换: $A(f)=\int_{-\infty}^{\infty} a(t)e^{2\pi ift}dt$
  - A(f)的逆变换为：$a(t)=\frac{1}{2\pi}\int_{-\infty}^{\infty} A(f)e^{-2\pi ift}dt$ ，
  - N个离散数据a=(a_0,a_1,…,a_{N-1})的Fourier变换:

$$A_j=\sum_{0\le k\le N-1} a_k e^{2\pi ijk/N},\quad 0\le j<N \qquad a_k=\frac{1}{N}\sum_{0\le j\le N-1} A_j e^{-2\pi ijk/N},\quad 0\le k<N$$

  - 令 $\omega=e^{2\pi i/N}$， 多项式 $a(x)=\sum_{0\le k\le N-1} a_k x^k$，则 $A_j=a(\omega^j),\ j=0,1,\cdots,N-1$
  - 当N=2n时，ω^j 是1在复数域的2n个复数根
  - $\omega^j=e^{\frac{2\pi j}{2n}i}=\cos\frac{\pi j}{n}+i\sin\frac{\pi j}{n}$， 满足
  - $\omega^{j+n}=-\omega^j$, j=0,1,2,..,n-1，因为 $\omega^n=-1$

快速Fourier变换

- 改进Fourier变换：避免$(\omega^j)^i$的重复计算
  - 考虑如下代数变换

    $a_1(x)=a_{n-1}$

    $a_2(x)=a_{n-2}+xa_1(x)=a_{n-2}+a_{n-1}x$

    $a_3(x)=a_{n-3}+xa_2(x)=a_{n-3}+a_{n-2}x+a_{n-1}x^2$

    ….

    $a_n(x)=a_0+xa_{n-1}(x)=a_0+a_1x+a_2x^2+\ldots+a_{n-1}x^{n-1}=a(x)$
  - 对每一个$\omega^0, \omega^1,\ldots, \omega^{2n-1}$，顺序计算$a_1(x),\ldots a_n(x)$即可
  - 复杂度：
    - 由$a_i(x)$计算$a_{i+1}(x)$需一次乘、一次加，计算每根2n次
    - 计算2n个根$4n^2$次。所以$T(n)=O(n^2)$

快速Fourier变换

- 快速Fourier变换：分治算法
  - 定义如下两个N次多项式：分别由a(x)偶、奇系数组成
    - $b(x)=a_1+a_3x+a_5x^2+\ldots+a_{N-1}x^{(N-2)/2}$
    - $c(x)=a_0+a_2x+a_4x^2+\ldots+a_{N-2}x^{(N-2)/2}$，则，
    - $a(x)=b(x^2)x+c(x^2)$，则

$$a(\omega^j)=b(\omega^{2j})\omega^j+c(\omega^{2j}),\quad a(\omega^{j+n})=-b(\omega^{2j})\omega^j+c(\omega^{2j})$$

j=0,1,2…,n-1，　　（$\omega^n=-1$）

  - 时间复杂度

$$T(N)=\begin{cases}a, \text{if } N=1\\ 2T(N/2)+cN, \text{if } N>1\end{cases}$$

    - 直接归纳，或根据后面的主定理情况(2)：T(n)=O(NlogN)
    - 这真是分治策略的一个成功范例！

快速Fourier变换

```
Proc FFT(N, a,w,A)
 # N=2n,w是n次单位根，a是已知的N元数组，代表多项式a(x)的系数，
 # A是计算出来的N元数组，A[j]=a(w^j), j=0,1,…,N-1.
    real b[ ], c[ ];  int j;
    complex B[ ], C[ ], wp[ ];
    if  N=1 then A[0]:=a[0];
    else
        n:=N/2;
        for j from 0 to n-1 do
          b[j]:=a[2*j+1];  c[j]:=a[2*j];
        end{for}
     end{if}
      FFT(n,b,w*w,B);
      FFT(n,c,w*w,C);
      wp[0]:=1;
```

```
for j from 0 to n-1 do
        wp[j+1]:=w*wp[j];
        A[j]:= C[j]+B[j]*wp[j];
        A[j+n]:= C[j]-B[j]*wp[j];
 end{for}
end{FFT}
```

第三章 分治算法

- 3.7 分治算法的分析技术
  - 主定理：设$a \geq 1, b > 1$为常数，$f(n)$为函数，$T(n)$为非负整数，且$T(n)=aT(n/b)+f(n)$，则有：
    - (1)若$f(n)=O(n^{\log_b a-\varepsilon})$，　$\varepsilon>0$，则$T(n)=\Theta(n^{\log_b a})$
    - (2)若$f(n)=\Theta(n^{\log_b a})$，则$T(n)=\Theta(n^{\log_b a}\log n)$
    - (3)若$f(n)=\Omega(n^{\log_b a+\varepsilon})$，　$\varepsilon>0$，且对于某个常数$c<1$和充分大的n有，$af(n/b) \leq cf(n)$，则$T(n)=\Theta(f(n))$.
  - 例:
    - $T(n)=9T(n/3)+n$，$a=9,b=3,n^{\log_3 9}=n^2,f(n)=O(n^{\log_3 9-1})$,符合主定理(1)且$\varepsilon=1$，所以$T(n)=\Theta(n^2)$.
    - $T(n)=T(2n/3)+1$，$a=1,b=3/2,f(n)=1,n^{\log_{3/2} 1}=n^0=1,f(n)=1$,符合主定理(2)，所以$T(n)=\Theta(n^0\log n)=\Theta(\log n)$.
    - ClosPair2,$T(n)=2T(n/2)+cn$,符合(2),$T(n)=\Theta(n^{\log_2 2}\log n)=\Theta(n\log n)$

分治算法的分析技术

- (3)第三种情况，$f(n)=\Omega(n^{\log_b a+\varepsilon})$
  - $T(n)=c_1 n^{\log_b a}+\sum_{j=0}^{k-1} a^j f(\frac{n}{b^j})$

$$\le c_1 n^{\log_b a}+\sum_{j=0}^{\log_b n-1} c^j f(n) \qquad \text{// (af(n/b) ≤cf(n))}$$

$$= c_1 n^{\log_b a}+f(n)\frac{c^{\log_b n}-1}{c-1} = c_1 n^{\log_b a}+\Theta(f(n)) \quad \text{//(c<1)}$$

$$=\Theta(f(n))$$

- 例：T(n)=3T(n/4)+nlogn，a=3,b=4,f(n)=nlogn，由于

$$n\log n=\Omega(n^{\log_4 3+\varepsilon})=\Omega(n^{0.793+\varepsilon}) \quad , \quad \varepsilon\approx 0.2$$

要af(n/b) ≤cf(n)成立，即(3n/4)*logn/4 ≤cnlogn，

只要c ≥3/4，上述不等式即对充分大的n成立，所以符合(3)

$T(n)=\Theta(f(n))=\Theta(n\log n)$

分治算法的分析技术

- 分治算法中常见的递归方程
  - 如果规约后的子问题比原问题呈常数量级的减少，就会出现第一类递推方程：

    (1)$T(n)=\sum_{i=1}^{k} a_i T(n-i)$ +f(n)
    - 该类问题可用迭代求解、递归树分析求解。例：
    - PartSelect最坏情况，T(n)=T(n-1)+n-1，$T(n)=O(n^2)$ (迭代)
  - 如果子问题的规模比原问题呈倍数量级减少，就会出现第二类递推方程：

    (2)T(n)=aT(n/b)+f(n)，a子问题个数，b子问题减少倍数
    - 该类问题大多可用主定理分析。 例：
    - 矩阵乘法Strassen 算法，a=7,b=2,$\log_b a$=2.81,取 ε =0. 81则

      $f(n)=O(n^2)=O(n^{\log_b a-\varepsilon})$， 主定理（1）, $T(n)=\Theta(n^{\log_b a})=\Theta(n^{2.81})$
  - 其他，如Select算法，$T(n)\le T(n/5)+T(3n/4)+cn$,$T(n)=O(n)$ (递归树)

第三章 分治算法

- 3.8 分治算法的改进技术
  - 通过代数变换减少子问题的个数
    - 例：矩阵乘法Strassen 算法
    - 例：设$n=2^k$,X,Y是两个n位二进制数，求XY。
      - 考虑分治算法：将X、Y都分成相等的2段，
      - $X=A2^{n/2}+B$，$Y=C2^{n/2}+D$，则
      - $XY=AC2^n+(AD+BC)2^{n/2}+BD$
      - 4个n/2规模的乘法和n+n/2位移位操作、3个加法复杂度cn，$T(n)=4T(n/2)+cn$，根据主定理$T(n)=O(n\log_2 4)=O(n^2)$
      - 这个分治算法与常规乘法复杂度一样！
      - 改进：利用AD+BC=(A-B)(D-C)+AC+BD可将子问题降为3：
      - $T(n)=3T(n/2)+cn$，得$T(n)=O(\log_2 3)=O(n^{1.59})$

第四章 贪心算法

- 4.1贪心算法的基本思想
  - 例1：找零钱：给孩子找回87分硬币，现有硬币规格50分、10分、5分、2分、1分。
    - 方法：尽量找面值大的硬币。 结果：50+3*10+5+2＝87
  - 贪心策略：*在每一步决策中总是作出在当前看来是最好的选择*。
  - 例2：活动安排问题：有n项活动申请使用同一个礼堂，每项活动有一个起始时间和截止时间，任何两个活动不能同时举行。问如何选择这些活动，使被安排的活动数量最多？
    - 算法策略可以选择
      - 1.按活动的开始时间从小到大选择
      - 2.按活动占用时间从小到大选择
      - 3.按活动的截止时间从小到大选择

贪心算法的基本思想

- 以上策略都是一种贪心决策，但：
- 策略1是不正确的。反例：

  S={1,2,3},s1=0,f1=20;s2=2,f2=5;s3=8,f3=15.

  该贪心策略结果{1};显然{2,3}更好。
- 策略2也不成功。反例：

  S={1,2,3},s1=0,f1=8;s2=7,f2=9;s3=8,f3=15.

  该贪心策略结果{2};而解{1,3}显然更好。

- 只顾眼前的贪心准则，*有时会导致局部最优，而不是全局最优。这种决策的计算量比较小，但需要正确性证明。*
- 许多NP难组合优化问题，目前仍未找到有效的算法，贪心策略常用于设计这些问题的近似算法。

贪心算法的基本思想

```
proc GreedyAction(s, f，n)    //策略3的活动安排贪心算法
   //s[1..n]、f[1..n]分别代表n项活动的起始时间和结束时间,
   //并且满足f[1]≤ f[2]≤…≤ f[n]
  j:=1; solution:={1}; //解向量初始化
  for i from 2 to n do
    if si≥fj then
      // 将i加入解中
       solution:=solution ∪ {i};
       j:=i;
    end{if}
  end{for}
  return(solution);
end{GreedyAction}
```

贪心算法的基本思想

- 活动安排问题的策略3贪心算法正确性证明：归纳法
  - 证：将活动按截止时间递增排列。归纳证明有最优解含算法前k步值。
  - K=1时，算法选择了活动1。设A=$\{i_1,i_2,..,i_j\}$是一个最优解，则A1=$\{1,i_2,..,i_j\}$也是一个最优解，因为活动1比i_1结束时间更早，A1是相容的活动集。即算法的第一步导致最优解。
  - 设对算法的第k步正确，令$i_1=1,i_2,\ldots i_k$是算法选择的活动序列，则存在最优解$A=\{i_1=1,i_2,\ldots i_k\} \cup B$。令$S'$是S中剩下的与$i_1=1,i_2,\ldots i_k$相容的活动，即$S'=\{j|s_j \geq f_{ik},j \in S\}$,那么B是$S'$的一个最优解。否则，若$S'$的最优解是$B'$且$|B'|>|B|$,那么用$B'$替换B得到的解$=\{i_1=1,i_2,\ldots i_k\} \cup B'$比A活动更多，与A是最优解矛盾。
  - 根据归纳假设，算法第一步选择结束时间最早的活动总是导致一个最优解，所以问题S'存在一个最优解$B^*=\{i_{k+1},\ldots\}$，$|B^*|=|B|$。
  - 于是$A'=\{i_1=1,i_2,\ldots i_k\} \cup B^*=\{i_1=1,i_2,\ldots i_k,i_{k+1}\} \cup (B^*-\{i_{k+1}\})$与A的活动一样多，也是最优解，且包含了算法前k+1步选择的活动。算法步数是有限的，k是任意的，得证。

贪心算法的基本思想

- 例3：背包问题
  - 背包容量为M，物品件数 n。重量w_i，价值p_i，变量 x_i，$0 \le x_i \le 1$，
  - 数学模型：$\max \sum p_i x_i$

    $s.t. \sum w_i x_i \le M$
  - 贪心准则：

    1. 价值大的物品优先装包；（28.2）
    2. 重量轻的物品优先装包；（31）
    3. 单位价值大的物品优先装包。(31.5)
  - **例子** n=3, M=20, p=(25, 24, 15), w=(18,15,10)
  - 结论：以“单位价值最大的物品优先装包”为贪心准则的贪心算法获得的效益值最大。(当然需要证明)

贪心算法的基本思想

- 背包问题的贪心算法

```
proc GreedyKnapsack (p, w, M, x, n)//价值数组p[1..n]、重量数组
  float p[1..n], w[1..n], x[1..n], M, rc;   //w[1..n]，它们元素的排列
   integer i, n;                            //顺序满p[i]/w[i]≥p[i+1]/w[i+1]
   x:= 0;  // 将解向量初始化为零             //M是背包容量，x[1..n]是解向量
   rc:= M; // 背包的剩余容量初始化为M
   for i =1 to n do
        if w[i] ≤ rc then
          x[i]:=1; rc:=rc-w[i];
        else break; end{if}
  end{for}
     if i≤n then
        x[i]:=rc/w[i];
     end{if}
end{GreedyKnapsack}
```

第四章 贪心算法

- 4.2 作业调度问题
  - 带期限的单机作业安排问题
    - 已知n项作业 E={1, 2, … ,n}要求使用同台机器完成，而且每项作业需要的时间都是1。第k项作业要求在时刻d_k之前完成, 而且完成这项作业将获得效益p_k， k=1, 2, … , n 。
    - 作业集E的子集称为相容的如果其中的作业可以被安排由一台机器完成。
    - 带限期单机作业安排问题就是要在所给的作业集合中选出总效益值最大的相容子集。

作业调度问题

- **proc GreedyJob**(d, p, n) //带期限的单机作业安排算法
  //d[1..n]和p[1..n]分别代表各项作业的限期和效益值，
  //而且n项作业的排序满足：$p_1 \geq p_2 \geq \ldots \geq p_n$
  local J;
  J:={1}; //初始化解集
  for i **from** 2 **to** n **do**
  if J ∪ {i} 中的作业是相容的 //检验规则：
  then //对任一j ∈ J ∪ {i},作业期限≤d(j)的数量≤ d(j)个
  J:= J ∪ {i}; // 将i加入解中
  end{if}
  end{for}
  end{GreedyJob} //贪心准则：尽量选取效益大的作业安排

作业调度问题

- 定理：GreedyJob对单机作业安排问题获得最优解。
- 证明：
  - 假设贪心算法所选择的作业集J不是最优解，则一定有相容作业集I，其产生更大的效益值。假定I是具有最大效益值的相容作业集中使得|I∩J|最大者，往证 I=J。
  - 反证法：若I=J 不成立，则这两个作业集合之间没有包含关系。这是因为算法GreedyJob的特性和假定I 产生的效益值比J的效益值更大。假设a是J\I中具有最大效益的作业，即**J中比a具有更大效益的作业(如果有的话)都应该在 I(阴影) 中**。
  - 如果作业 $b \in I \setminus J$，则$p_b \le p_a$ 。反证(见下页，下面取自讲义参考)。【若$p_b > p_a$，那么由算法中对J中作业的选取办法（相容性要求），J中至少有 f_b个效益值≥p_b的作业，其期限值≤ f_b。这些作业一定在 I 中(阴影)，因而 I 中至少有f_b+1个作业，其期限值≤ f_b ，这与 I 的相容性矛盾。所以，I\J**中事件的效益值均不超过p_a**。 】

作业调度问题

- 关于$p_b <= p_a$的证明：
- 如果$p_b > p_a$，则运行算法时，检查b时，J里的作业价值一定$>= P_b > p_a$,根据a的选取规则，它们都在I∩J中。b未被选入，说明它与I∩J不相容。但b在I中，I是解，b必与I∩J相容。矛盾。
  - 注意： J中至少有 f_b个效益值$\geq p_b$的作业，其期限值$\leq f_b$。未必成立！因为：
  - 检查规则：作业i可以加入J只要当前解集中期限不大于f_i的作业少于f_i个。似不正确，反例：例4.2.2作业7.
  - 讲义p85有关检查规则应改为：如果对任一$j \in J \cup \{i\}$,作业期限$\leq d(j)$的数量$\leq d(j)$个，作业i可以加入J。

作业调度问题

- 称区间[k-1,k]为时间片k，相容作业集 I 的一个调度表就是指定 I 中各个作业的加工时间片。设S是 I的一个调度表。若存在$b \in I \backslash J$被安排在f_a前,则可用a替换b得I′,见下面的讨论。否则， S 将时刻 f_a 前的时间片均安排给$I \cap J$中的作业，设b是$I \backslash J$中最早被安排的作业,被安排在时间片k上，则$k > f_a$。
- 前k-1个时间片上安排的都是$I \cap J$中的作业，这些作业的集A_1再添上作业 a 得到 J 中k个作业。由作业集的相容性， A_1中至少有一个作业其期限值≥k。将这个作业与作业b交换安排的时间片，得到新的调度表S_1。如果在调度表S_1中作业 b 安排在时间片 k_1，$k_1 > f_a$，则同理，可以将作业 b 再向前移,得到新的调度表S_2。如此做下去，必得到 I 的调度表S′，其在 f_a 前的时间片安排有 I\J 中作业b 。

令 $I' = (I \backslash \{b\}) \cup \{a\}$，则 I′ 也是相容的作业集。而且，由于$p_b \le p_a$，I′ 的效益总值不小于 I 的效益总值，因而 I′ 具有最大的效益总值，但$|I' \cap J| > |I \cap J|$，与 I 的选取相悖。因而，I=J，J 是最优解。 证毕 。

作业调度问题

- 单机作业调度问题的贪心算法
  - 将上述算法找到的作业集按作业期限升序排列，就产生一个作业调度。
  - 算法的复杂度分析
    - J ∪ {i}的相容性判断最坏|J|次，|J|最大i-1。所以
    - $T(n)=1+2+\ldots+n-1=O(n^2)$
  - 例子: 设n=7,

    $(p_1, p_2,\ldots ,p_n)=(35,30,25,20,15,10,5)$,

    $(d_1, d_2,\ldots ,d_n)=(4,2,4,3,4,8,3)$
  - 算法GreedyJob的执行过程 :

作业　1；2，1；2，1，3；2，4，1，3；2，4，1，3，6；

期限　4；2，4；2，4，4；2，3，4，4；2，3，4，4，8；

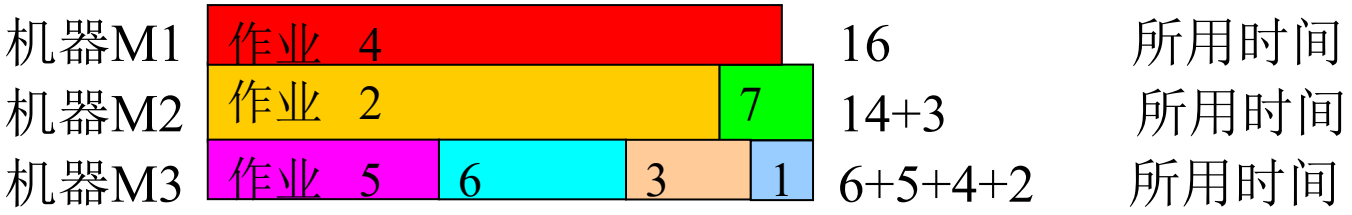
作业调度问题

- **proc GreedyJob**(D,J,n,k)

```
//D(1),…,D(n)是期限值，作业已按p1 ≥ p2 ≥ … ≥ pn排序。J(i)是最优解中的第i个作业。终止时，D(J(i))≤ D(J(i+1))，1≤i≤k
integer D[0..n],J[0..n],i,k,n,r
D(0):=0; J(0):=0;  //初始化
k:=1; J(1):=1;  //计入作业1，k 表示当前选择的作业个数
for i from 2 to n do // 找i的位置，并检查插入的可能性
 r:= k;
 while D(J(r))>D(i) and D(J(r)) > r do  //相容性条件：D(J(r)) ≥r
   r:= r-1;
 end{while};
 if D(J(r)) ≤ D(i) and D(i) > r  then
  for j from k to r+1 by –1 do
    J(j+1):=J(j);  //给作业i腾出位置
  end{for};
 J(r+1):=i;  k:=k+1;
   end{if};
end{for};
end{GreedyJob}
```

作业调度问题

- 多机作业调度问题：NP完全问题，贪心法可得较好解
  - 问题描述：有n项独立的作业{1,2,…, n},由m台相同的机器加工处理。作业 i所需要的处理时间为t_i。约定：任何一项作业可在任何一台机器上处理，但未完工前不准中断处理；任何作业不能拆分更小的子作业分段处理。多机调度问题要求给出一种调度方案，使所给的n个作业在尽可能短的时间内由m台机器处理完。
  - 贪心准则：将加工时间长的作业优先安排给空闲早的机器。
  - **例**：三台机器，七项作业，所需加工时间分别 2,14,4,16,6,5,3
  - 将作业标号按照所需加工时间由长到短排序：4,2,5,6,3,7,1
  - 贪心算法调度结果：所需时间为17.



作业调度问题

- 贪心算法不能对任何实例得到最优解
- 反例：（第10章证明近似比）
  - 有3台机器，8项作业，处理时间依次是3,4,3,6,5,3,8,4。算法分配给3台机器的作业为 {1,4}，{2,6,7}，{3,5,8}，负载分别是3+6=9,4+3+8=15,3+5+4=12，完成时间为15。这不是最优的分配方案，最优方案为：{1,3,4}，{2,5,6}，{7,8}，负载分别为3+3+6=12,4+5+3=12,8+4=12，完成时间为12.
  - 贪心规则：按作业顺序，优先交给空闲机器，2-近似算法
  - 贪心规则：按作业时间从大到小排序，空闲机器优先加工：分配给3台机器的作业为{7,1}，{4,8,6}，{5,2,3}，负载分别为8+3=11，6+4+3=13，5+4+3=12，完成时间为13。这个是3/2近似算法

第四章 贪心算法

- 4.3 最优生成树问题
  - 问题描述
    - 设无向连通带权图G=(V, E,W)，其中w(e) ∈W是边e的权。G的一颗生成树是包含了G的所有顶点的树，树中各边的权之和称为树的权，具有最小权的生成树称为G的最优生成树。
    - 实际问题，如不妨假定图G代表城镇间的交通情况，顶点代表城镇，边代表连接两个城镇的可能的交通线路。将每一条边赋予一个权值，这些权值可代表建造该条线路的成本。最优生成树表示这些城镇间连通的最小费用。

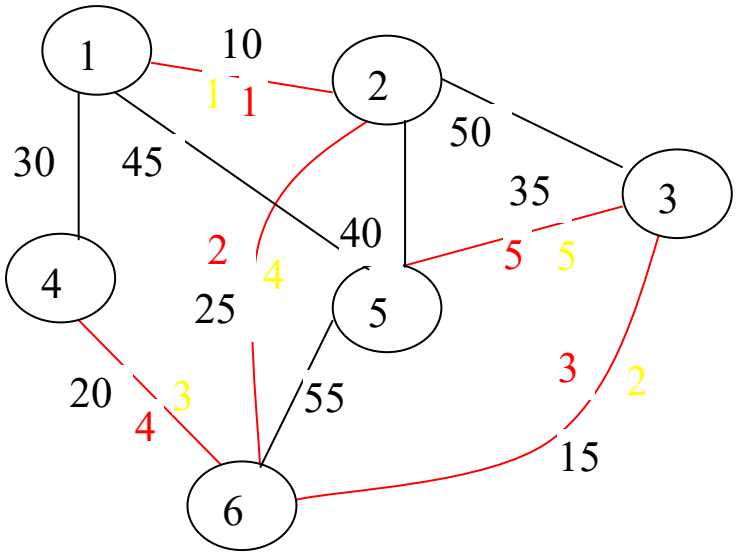
最优生成树问题

- Prim算法

1) 选择图G的一条权值最小的边 e_1，形成一棵两点一边的子树；
2) 假设G的一棵子树T已经确定；
3) 选择G的不在T中的具有最小权值的边e，使得T∪{e}仍是G的一棵子树。

- Kruskal算法

1) 选择图G的一条权值最小的边 e_1；
2) 设已选好一组边 $L=\{e_1,e_2,\ldots,e_k\}$；
3) 选择G的不在L中的具有最小权值的边e_{k+1}，使得$L\cup\{e_{k+1}\}$诱导出的G的子图不含G的圈。



Prim算法：

(1,2), (2,6), (6,3), (6,4), (3,5)

Kruskal算法：

(1,2), (3,6), (4,6), (2,6), (3,5)

最优生成树问题

- **PrimTree**(E,COST,n,T,mincost)

```
// E是边集，COST是带权邻接矩阵。
//最小生成树T作为一个集合存放到
//数组T[1..n-1,1..2]中, mincost:树的权
 1   real COST[1..n,1..n],mincost;
 2   integer  NEAR[1..n], n, i, j, k, s,
       T[1..n-1,1..2];
 3   (k,s):= 权值最小的边;
 4   mincost:＝COST[k,s];
 5   (T[1,1],T[1,2]):=(k,s); //初始子树
 6   for i=1 to n do //将NEAR赋初值
 7     if COST[i,s]<COST[i,k] then
 8       NEAR(i)=s;
 9     else NEAR(i)=k;
10     end{if}
11   end{for}
12   NEAR(k):=0,NEAR(s):=0;
13  for i=2 to n-1 do  //寻找T的其余n-2条边
14    choose an index j such that
      NEAR(j)≠0 and COST[j,NEAR(j)]
      is of minimum value;
15   (T[i,1],T[i,2]):=(j,NEAR(j)); //加入边
16    mincost:=mincost
                +COST[j,NEAR(j)];
17    NEAR(j):=0;
18    for t =1to n do  //更新NEAR
19      if NEAR(t)≠0 and COST[t,NEAR(t)]
          >COST[t,j] then
20        NEAR(t):=j;
21      end{if}
22    end{for}
23  end{for}
24  if mincost ≥ ∞ then
25    print('no spanning tree');
26  end{if}
27 end{PrimTree}
```

最优生成树问题

- PrimTree的复杂度
  - 语句3需要O(|E|)的时间；语句6至语句11的循环体需要时间为O(n)；语句18至22的循环体需要时间O(n); 语句14至17需要时间O(n); 语句13至23的循环体共需要时间$O(n^2)$。
  - 所以,PrimTree算法的时间复杂度为$O(n^2)$.
- 算法正确性证明
  - 定理：对任意正整数k<n，存在最小生成树包含算法前k步选择的边。
  - 根据上述定理，k=n-1时即为算法生成的最小生成树。
  - 关于生成树的引理：设G是n阶连通图，T是G的n阶连通子图，
    - (1)T是G的生成树当且仅当T有n-1条边。
    - (2)如果T是G的生成树，$e \notin T$，那么$T \cup \{e\}$含有一个回路。

最优生成树问题

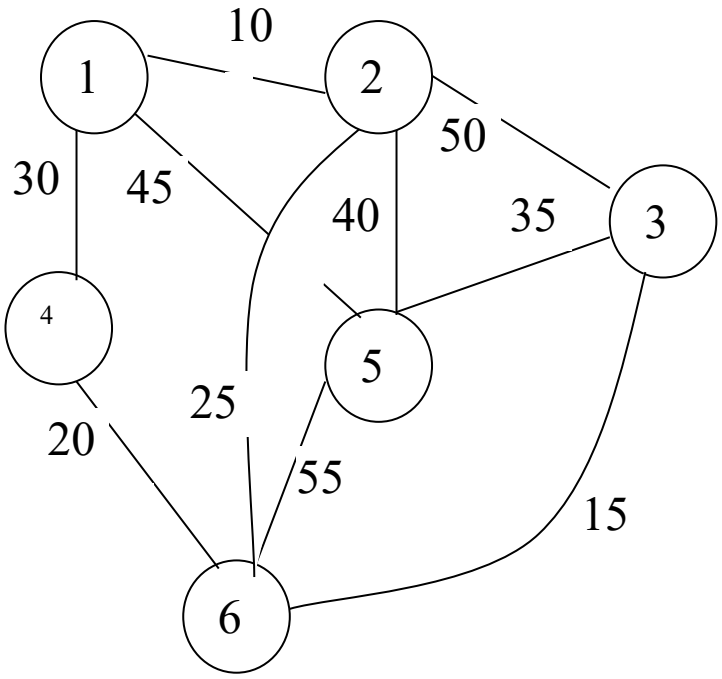
- Prim算法正确性证明：
  - K=1，设(1,i)是关联1的最小权边，T是一颗最小生成树。若T不包含(1,i)，根据引理，T ∪(1,i)含有一个圈。设这个圈中关联1的另一条边是(1,j)，令$T'=(T-\{(1,j)\}) \cup\{(1,i)\}$，则$T'$也是生成树，且$W(T') \leq W(T)$。
  - 假设算法进行了k-1步，生成树的边为$e_1,e_2,\ldots,e_{k-1}$，其k个端点构成集合S。由归纳假设存在G的最小生成树T包含这些边。
  - 设算法k步选择了顶点i_{k+1}，则i_{k+1}到S中顶点的边的权值最小，设它为$e_k=(i_{k+1},i_l)$。若T不包含e_k，将e_k加到T中形成一条回路，这条回路一定有另外一条连接S与V-S中顶点的边e，用e_k替换e得到T*，T*是G的一颗生成树，包含$e_1,e_2,\ldots,e_k$，由于$w(e_k) \leq w(e)$，所以$w(T^*) \leq w(T)$。
  - 根据数学归纳法，定理得证。

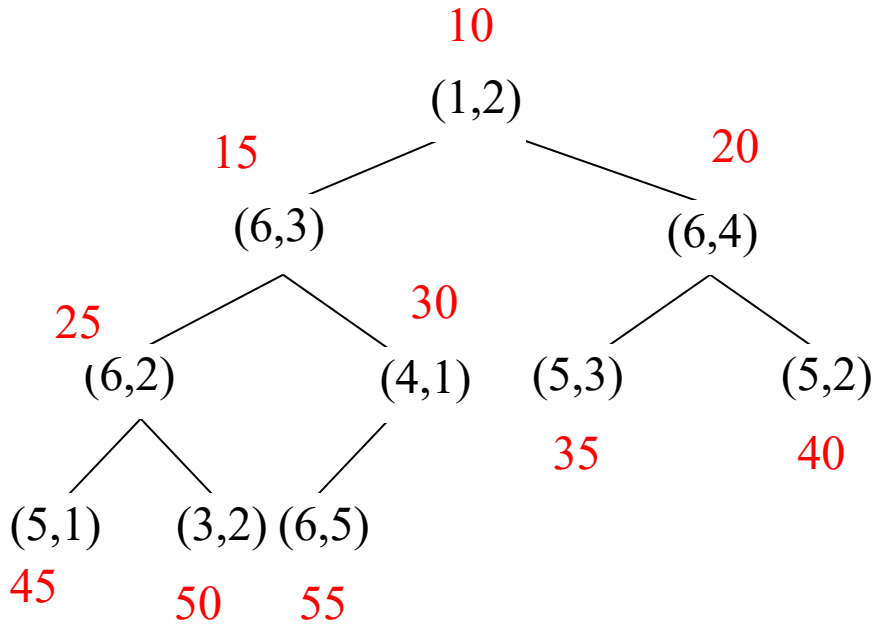
最优生成树问题

```
proc KruskalTree(E,COST,n,T,mincost)//说明同算法PrimTree
 1  real mincost, COST[1..n,1..n];
 2  integer Parent[1..n],T[1..n-1,1..2],n;
 3  以带权的边为元素构造一个min-堆; //对权边排序也可,T(m)=O(mlogm)
 4  Parent:=-1; //每个顶点都在不同的集合中;
 5  i:= 0; mincost:= 0;
 6  while i<n-1 and min-堆非空 do
 7    从堆中删去最小权边（u，v）并重新构造min-堆
 8    j:= Find(u); k:= Find(v);
 9    if j≠k then  //保证不出现圈
10      i=:i+1; T[i,1]:=u; T[i,2]:=v;
11      mincost:=mincost+COST[u,v];
12      Union(j,k); //把两个子树联合起来
13    end{if}
14 end{while}
15 if i≠n-1 then
16   print('no spanning tree');
17 end{if}
18 return;
end{KruskalTree}
```

最优生成树问题

一个赋权图和它的
边的最小堆





最优生成树问题

- KruskalTree正确性证明
  - 设T是算法产生的G的一棵生成树，而T′是G的一棵最优生成树且使得|E(T′)∩E(T)|最大。用E(T)表示树T的边集，w(e)表示边e的权值，而边集E的权值之和用w(E)表示。以下证明E(T)=E(T′).
  - 反证。假设E(T)≠E(T′),因为|E(T)|=|E(T′)|,所以E(T)与E(T′)没有包含关系。设e是E(T)\E(T′)中权值最小的边。将e添加到T′中即得到T′的一个圈, 记为 e，e_1 ,e_2 ,…,e_k。因为T是树，诸e_i中至少有一个不属于E(T),不妨设是e_i,则必然有$w(e)\le w(e_i)$。否则，由$w(e)>w(e_i)$以及E(T)中比e权值小的边都在T′中，e_i同这些边一起不含有圈，因而，按Kruskal算法，e_i将被选到E(T)中，矛盾。
  - 在T′中去掉边e_i，换上边e，得到G的一棵新的生成树T″,这棵树有两个特点：

    a). T″的权值不大于T′的权值，因而与T′有相等的权值；

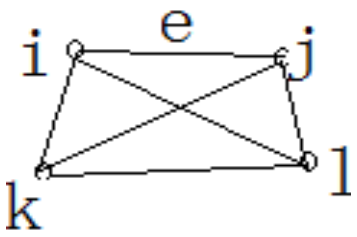
    b). |E(T″)∩E(T)| > |E(T′)∩E(T)|.

    a)说明T″也是一棵最优树，而b)与T′取法相悖。因此

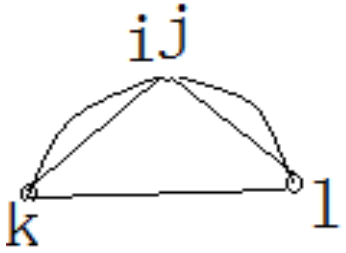
    E(T) =E(T′)，T是最优生成树。

最优生成树问题

- KruskalTree正确性证明2（归纳法，**自学**）*
  - 证：n=2时，只有两个顶点，算法最小边，是最小生成树。
  - 设对n，结论成立。考虑n+1阶连通图G，设其最小边为e=(i,j)，从G中短接i和j，得到n阶图G/，根据归纳假设，使用算法可得到G/的一颗最小生成树T/，令T=T/∪{e}，则T是算法生成的G生成树。下面反正T是最小生成树。



  - 如不然，存在G的一颗最小生成树T*，w(T*)<w(T)。首先说明存在一颗这样的树T*包含e。因为如果e∉ T*，在T*中加入e，根据引理，形成一条回路C，去掉C中任意一条其它的边，所得到的 生成树的权不大于w(T*)，且包含e。



短接边e

  - 在T*中短接i,j就得到G/的生成树T*-{e}，且

    w(T*-{e})=w(T*)-w(e)<w(T)-w(e)=w(T/)，与T/最优矛盾。得证。

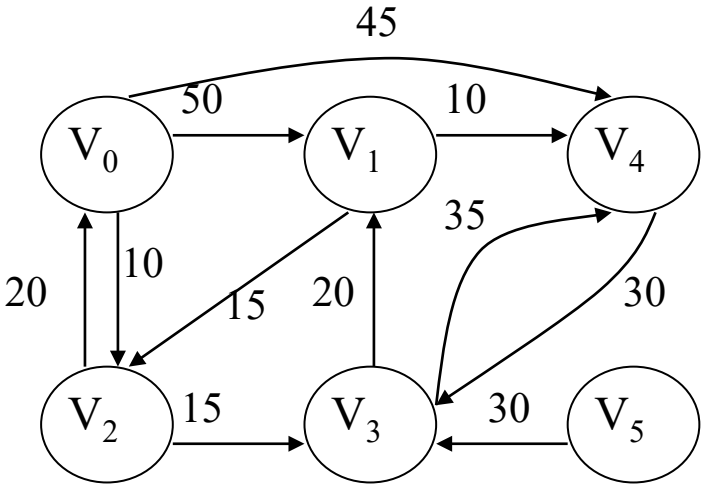
最优生成树问题

- Kruskal算法的复杂度
  - 设图G有n个顶点，m条边。
  - 建立带权边的min-堆O(mlogm)，while循环内每次删除堆顶并调整O(logm)，总计O(mlogm).(讲义附录)
  - Parent赋初值值O(n)、while循环内通过find()、union()操作建立连通分支总的时间为O(nlogn)(严格是n.β(2n,n))。
  - While循环其他操作常数项O(n)
  - 所以 T(n)=O(mlogm+mlogm+nlogn+n)，因为简单连通图的边m满足n(n-1)/2 ≥m ≥n-1，得到 $mlogm=O(mlogn^2)=O(mlogn)$
  - T(n)=O(mlogn)
  - 问：Prim算法、Kruskal算法哪个算法效率高?

第四章 贪心算法

4.4 单点源最短路径问题

- 问题：赋权有向图G=(V,E,w), 指定的顶点v_0，求由v_0 出发到G中其它各个顶点的最短路径。
- 贪心准则：迄今已生成的所有路径长度之和为最小：
- S-当前已构造出最短路径终点集，初始S={v_0}，算法每步将1个顶点加入S，加入的顶点i∈V-S满足贪心规则：v0经过S中的顶点到达i经过的路径最短。
- P=$v_0u_1u_1..u_sv_k$是v_k的最短路径，则p_1= P=$v_0u_1u_1..u_s$是u_s的最短路径。



从v_0到各顶点的最短路径

路径	长度
(1) v_0v_2	10
(2) $v_0v_2v_3$	25
(3) $v_0v_2v_3v_1$	45
(4) v_0v_4	45

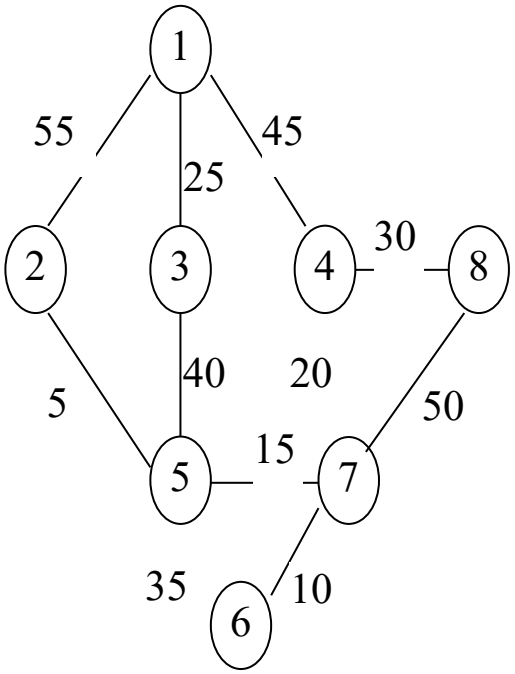
单点源最短路径问题

- Dijkstra算法

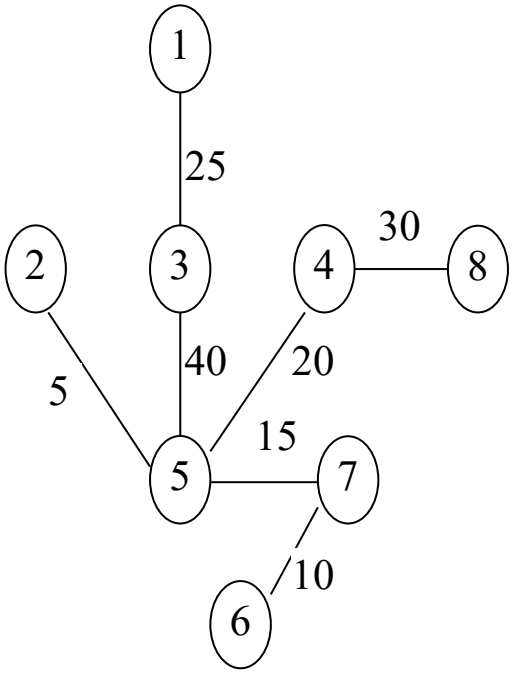
```
Proc DijkstraPaths(v,COST,n, Dist,Parent)
 //G是具有顶点{1,2,…,n}的有向图, v是
 //G中取定的顶点, COST是G的邻接矩阵,
 //Dist表示v到各点的最短路径之长度,
 //Parent表示各顶点在最短路径上的前继。
 //{S[i]=1}表示S集合,{S[i]=0}表示V-S
    bool S[1..n];  float
    COST[1..n,1..n], Dist[1..n];
    integer u,v,n,num,i,w;
    for i =1 to n do  //将集合S初始化为空
       S[i]:=0; Parent[i]:=v;
       Dist[i]:=COST[v,i];
    end{for}
S[v]:=1;  Dist[v]:=0;
Parent[v]:=-1;
//首先将节点v记入 S
for i =1 to n-1 do
 选取顶点w，满足
   Dist(w) = min_{S(u)=0} {Dist(u)}
 S[w]:=1;
 while S[u]=0 do
    //修改v通过S(S新加了w)到达
    // S以外的结点的最小距离值
  if Dist[u]> Dist[w]+COST[w,u]  then
      Dist[u]:=Dist[w]+COST[w,u];
      Parent[u]:=w;
   end{if}
  end{while} ;
 end{for}
end{DijkstraPath}
```

单点源最短路径问题

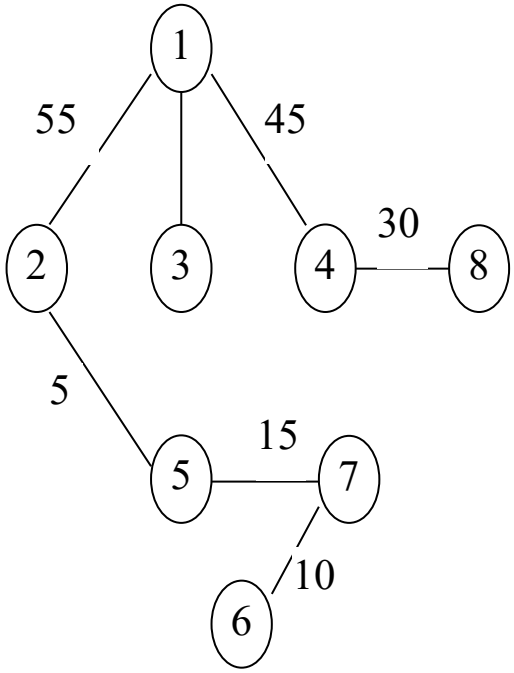
- 例：



赋权连通图G



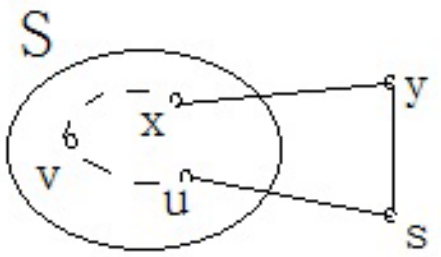
G的最优生成树



G的一棵单点源
最短路径生成树

单点源最短路径问题

- 单点源最短路径问题正确性
  - 证：对k归纳。K=1,S={v},显然dist(v)=short(v)=0.
  - 设n=k时算法得到的路径最短。考虑算法在第k+1步选择了顶点s，其关联边为(u,s)，u在S中。假如存在另一条从v到s的最短路径L，路径中第一次离开S的边为(x,y)，其中x∈S,y ∈V-S。由于在这一步算法选择了s而没有选择y，则有：
  - Dist(s) ≤dist(y)，而dist(y)+d(y,s)=L,(d(y,s)表示y到s的距离)
  - 于是，dist(s)) ≤L，即算法选择的路径是v到s的最短路径。



单点源最短路径问题

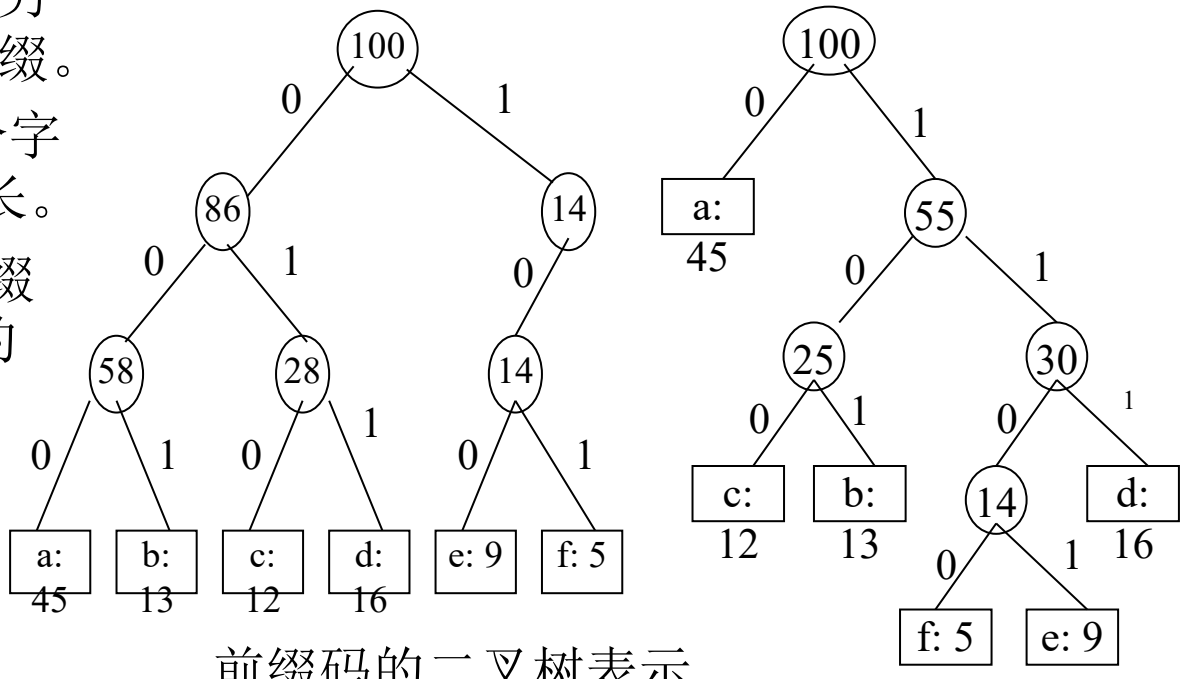
- Dijkstra算法的复杂度
  - For循环中，求元素w需做n-i-1次比较，

    (事实上本算法使用表示S、V-S的数据结构需n-1次判断、n-i-1次比较)
  - While循环更新dist值也需要n-i-1次操作(同上)
  - 所以$T(n)=1+2+\ldots+n-1=n(n-1)/2=O(n^2)$

第四章 贪心算法

- 4.5最优前缀码-Huffman编码
  - **二元前缀码**：表示一个字符的0,1字串不能是表示另一个字符的0,1字串的前缀。
  - B=∑f(x_i)d(x_i)是存储一个字符的平均值。d(x_i)为码长。
  - 平均码长达到最小的前缀编码方案称为字符集C的一个最优前缀编码。
  - 例：字符集及其频率如表.长为100000的文件：
  - 定长码：300000 bit，
    变长码：224000 ,省25%

不同的字符	a	b	c	d	e	f
频率f（千次）	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100



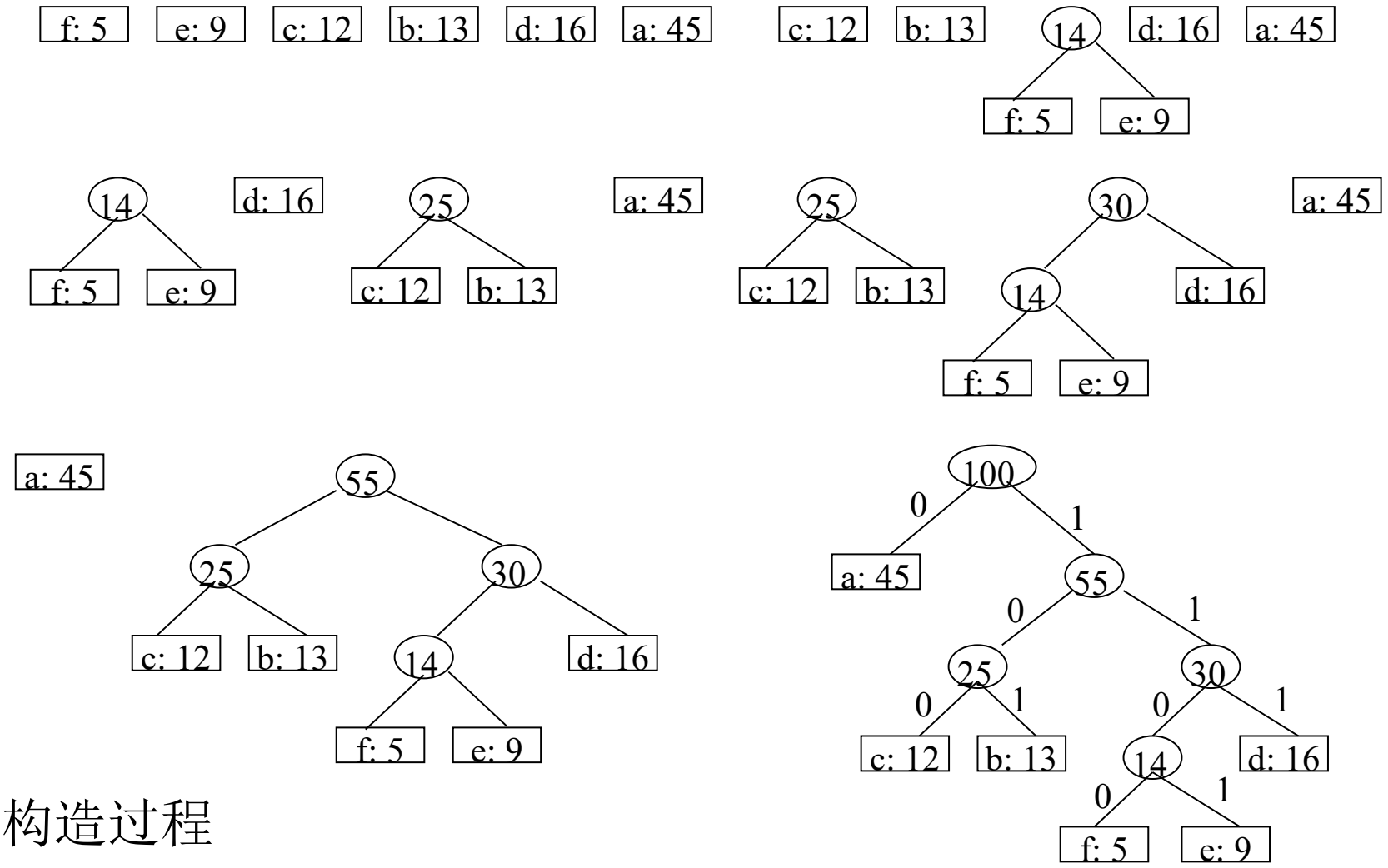
前缀码的二叉树表示

最优前缀码

- 问题描述：给定字符集C={$x_1,x_2,\ldots,x_n$}和每个字符频率$f(x_i)$,i=1,2,..,3,求关于C的一个最优前缀码。
- 一个著名的贪心算法是Huffman编码： Huffman(C)
  - 1.n:=|C|
  - 2.Q:=C //按频率构造min堆
  - 3.for i=1 to n-1 do
  - 4. z:=Allocate-node() //生成树节点z
  - 5. z.left:=Q中最小元x
  - 6. z.right:= Q中最小元y
  - 7. f(z):=f(x)+f(y)
  - 8. insert(Q,z) //将z插入Q并保持递增
  - 9 end for
  - 10. return z //树z从根到叶子的路径即为叶子字符的编码

算法复杂度：
2初始堆O(nlogn)
3行循环n-1次
5、6调整堆2logn
8行循环体内插入操作O(logn)，其它操作常量
所以T(n)=O(nlogn)

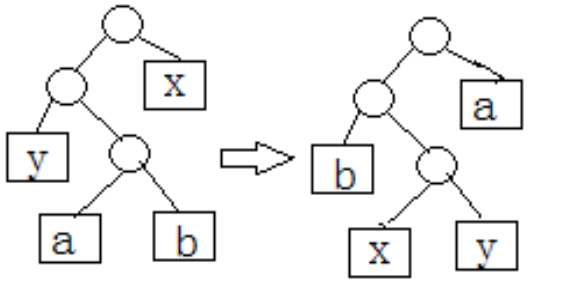
最优前缀码



构造过程

最优前缀码

- Huffman编码的正确性
  - 引理：设C是字符集，f(c)为频率，x,y∈C,f(x),f(y)频率最小，则存在最优二元前缀码使得x,y是最深叶顶点且为兄弟。
  - 证：设T是一颗最优二元前缀码对应的二叉树，且x,y不是最深层的兄弟，那么**存在最深层的2片兄弟树叶a和b(为什么？)**，使得$d_T(x) \le d_T(a), f(x) \le f(a), d_T(y) \le d_T(b), f(y) \le f(b)$.把x与a交换，y与b交换。得到树$T'$，那么两颗树的权值差是：
    - $B(T)-B(T') = \sum f(i)d_T(i) - \sum f(i)d_{T'}(i)$



把最小频率的树叶交换到最底层

$$=[f(x)d_T(x)+f(y)d_T(y)+f(a)d_T(a)+f(b)d_T(b)]$$
$$-[f(x)d_{T'}(x)+f(y)d_{T'}(y)+f(a)d_{T'}(a)+f(b)d_{T'}(b)]$$
$$=[f(x)d_T(x)+f(y)d_T(y)+f(a)d_T(a)+f(b)d_T(b)]$$
$$-[f(x)d_T(a)+f(y)d_T(b)+f(a)d_T(x)+f(b)d_T(y)]$$
$$=[f(x)-f(a)][d_T(x)-d_T(a)]+[f(y)-f(b)][d_T(y)-d_T(b)] \ge 0$$

  - 所以，T'也是最优二元前缀码的二叉树，包含x,y兄弟在最深层。

最优前缀码

- Huffman编码是最优编码
  - 证：n=2显然正确。设命题对n成立。对于|C|=n+1，
  - 设x,y是C中最小频率字符，令x,y的父节点代表一个新的字符z，出现频率为：f(z)=f(x)+f(y)。
  - 根据归纳假设，huffman编码可得到字符集$\{C\setminus\{x,y\})\cup\{z\}$的最优前缀编码树$T'$。将x,y加到$T'$上得到树T，则T就是huffman得到的关于C的编码树。下面证明T是C的最优前缀编码。
  - 如不然，存在权更小的树T*,根据引理，x,y可以是T*的最深层树叶，去掉T*中的x,y，得到树$T^{*'}$，满足
  - $B(T^{*'})=B(T^*)-[f(x)+f(y)]<B(T)-[f(x)+f(y)]=B(T')$
  - 这与T'是$\{C\setminus\{x,y\})\cup\{z\}$的最优前缀码树矛盾。
  - 得证。
  - （也可以使用讲义给出的交换论证证明）

第五章 动态规划算法

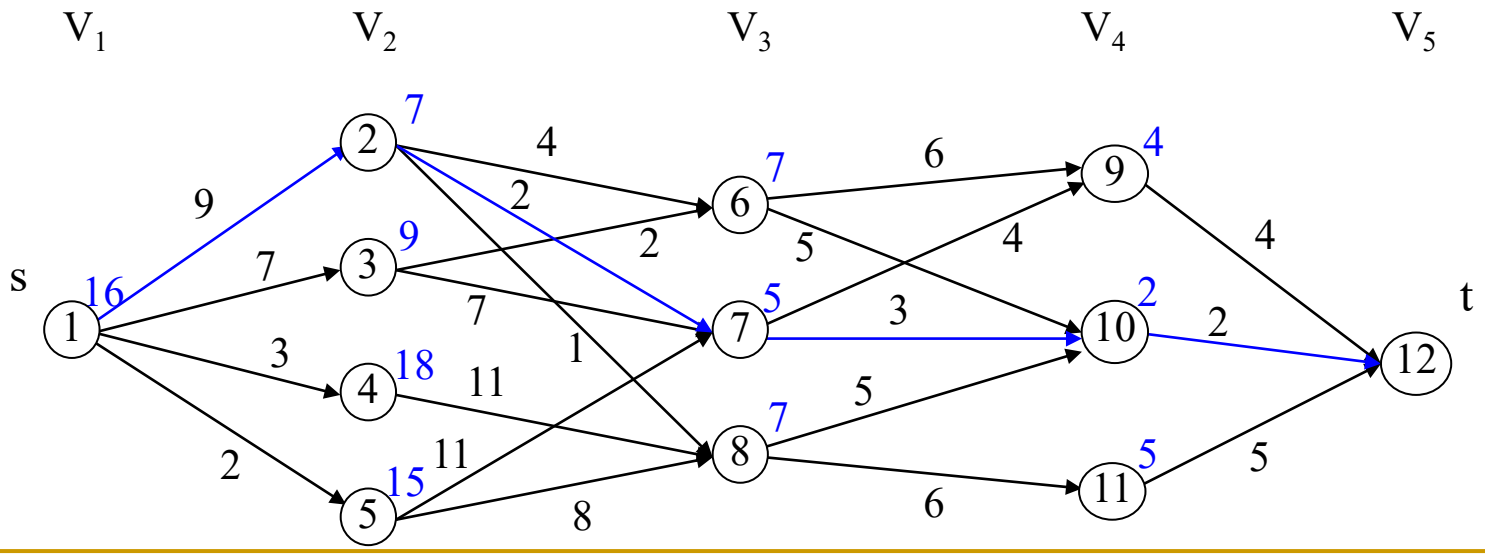
5.1 动态规划设计思想

动态规划方法

- 动态规划方法是处理分段过程优化问题的一类极其有效的方法，已经广泛应用于许多组合优化问题。
- 一般的组合优化问题都有对应用的目标函数和约束函数，如货郎问题，给定城市集合C={$c_1,c_2,\ldots c_n$},距离$d(c_i,c_j) \in Z^+, 1 \le i \le m$。求1,2,...m的一个排列$k_1,k_2,\ldots k_m$,以求得最小值：$\min\{\sum_{i=1}^{m-1} d(c_{ki},c_{ki+1}) + d(c_{km},c_{k1})\}$
- 其中$\sum_{i=1}^{m-1} d(c_{ki},c_{ki+1}) + d(c_{km},c_{k1})$称为目标函数，约束条件是，$k_1,k_2,\ldots k_m$构成1,2,...,m的排列。
- 蛮力算法对于规模较大的问题实际上是不可能运行的，动态规划技术把求解过程变成一个多步判断过程，每一步对应某个子问题，**通过子问题间的依赖关系**，减少重复工作。
- 对许多蛮力算法指数时间问题，动态规划技术都取得了突破进展，将时间降为$O(n^2)$或$O(n^3)$等多项式级别。

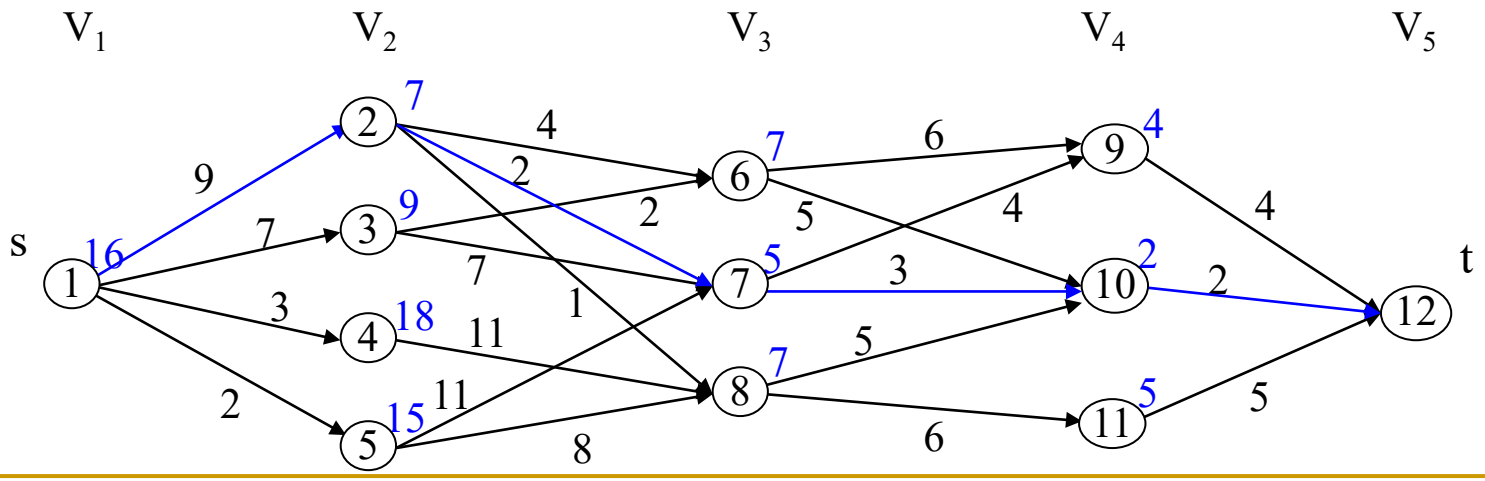
动态规划设计思想

- 例：多段图问题
  - 赋权有向图 G=(V,E)，顶点集V被划分成 k(≥2)个不相交的子集 V_i: 1≤i≤k，其中，V_1和V_k分别只有一个顶点s(称为源)和一个顶点t(称为汇)，所有的边(u,v)的始点和终点都在相邻的两个子集V_i和V_{i+1}中，而且u∈V_i，v∈ V_{i+1}。
  - **多阶段图问题**：求由s到t的最小成本路径（也叫最短路径）。



动态规划设计思想

- 求解过程：从终点向起点回推，把求解过程分为4步，依次求解v4,v3,v2,v1到v5的最短路径。
  - 1.求节点9,10,11到12的最短径，它们是4,2,5，标注节点右上角。
  - 2.求节点6,7,8到12的最短路径，利用第1步结果，如节点6：
    - 节点6径v4到12有两条路径，6-9…12和6-10…12。
    - 9…12和10…12利用第1步结果，得路径长度为10和7。结果经节点10的路径7最小，标记于节点6的右上角。同法依次求节点7,8。
  - 3.依次后推，求得节点1的最短路径1-2-7-10-12,长度16。



动态规划设计思想

- 多段图动态规划算法
  - **最优值递推关系式：** $COST(j)=\min_{(j,l)\in E}\{c(j,l)+\cos t(l)\}$

    COST(j)－节点 j 到 t 的最优路径成本。
  - **MultiGraph**(E, k, n, P) //有n个顶点的k段图G（按段序统一编号），E是边集，c(i, j)是边(i, j)的成本，P[1..k]是最小成本路径。

```
1   float  COST[1..n]; integer D[1..n-1]，P[1..k], r, j, n; COST[n]:=0;
3   for j = n-1 by –1 to 1 do
4     设 r 是这样一个顶点,(j, r)∈E 且使得 c(j, r)+COST[r] 取最小值
5     COST[j]:= c(j, r)+COST[r];
6     D[j]:=r;  //指出j的后继
7   end{for}
8   P[1]:=1; P[k]:=n; //最短路径的起点为s，终点为t
9   for i from 2 to k-1 do
10    P[i]:=D[P[i-1]];  //最短路径上的第i个节点是第i-1节点的后继
11  end{for}   end{MultiGraph}
```

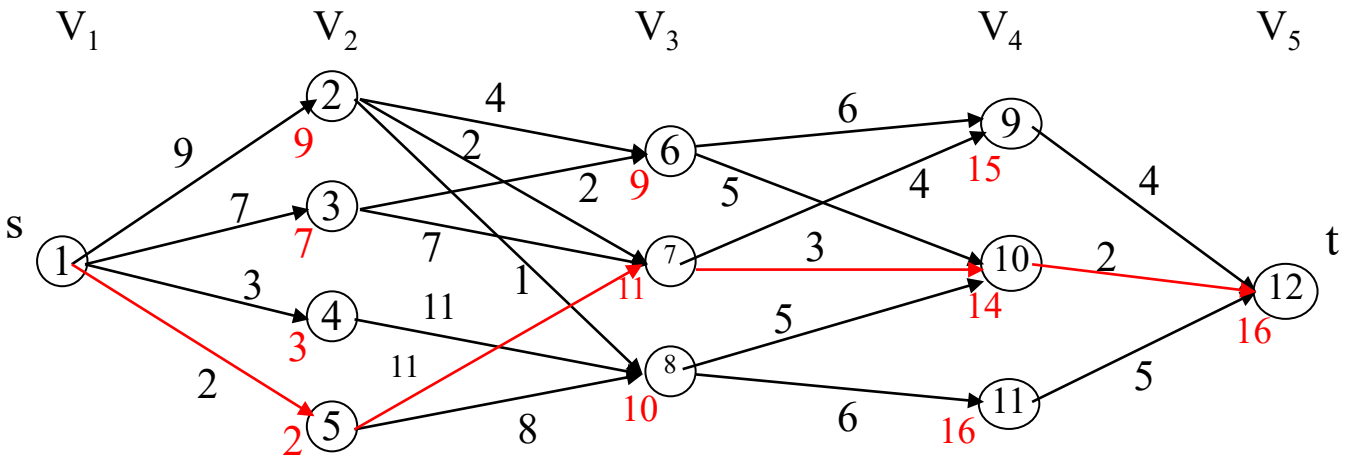
动态规划设计思想

- 上述方法在判断时只考虑前面子问题的最优解可能的延伸结果，把许多不可能成为最优解得路径尽早从搜索中删除，因此能提高效率。(蛮力算法:l层,k出度,$O(k^l)$)
- 算法复杂度：回推做n次加法和m(边数)次比较，T(n)=O(n+m)
- 为什么可以分段解决？最优子结构性质
  - **多段图问题具有最优子结构性质**：s→t： s, v_2, … , v_i,v_{i+1}, … , v_{k-1}, t—最优则 $v_i \rightarrow$ t: v_i,v_{i+1}, … , v_{k-1}, t —是子问题的最优子序列 。
  - 证明：如不然，设 vi, q_{i+1}, … , q_{k-1}, t 是一条由v_i到t的比v_i,v_{i+1}, … , v_{k-1}, t是更短的路径，则s, v_2,…, v_i,q_{i+1}, … , q_{k-1}, t是一条由s到t的比s, v_2, v_3, … , v_{k-1}, t更短的路径，与前面的假设矛盾。
  - 这个性质称多段图问题具有最优子结构性质。

动态规划设计思想

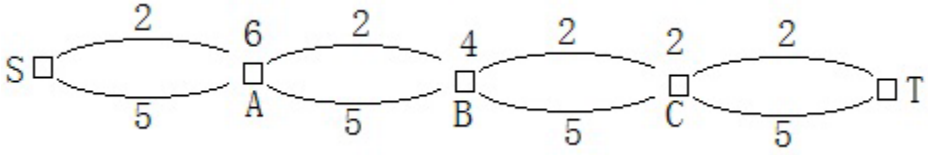
- 对多段图问题，子问题的划分能不能从前向后推？
  - 当然可以，此时的递推关系为

    BCOST(j)=min{(BCOST(l)+c(l,j)}，(l,j)∈E
  - BCOST(j)代表源点s到顶点j的最短路径长度。
  - 运行结果：



动态规划设计思想

- 动态规划技术的必要条件
  - 最优子结构性质：一个最优决策序列的任何子序列本身一定是**相对于子序列的初始和结束状态**的最优决策序列。
  - 一个问题可以使用动态规划求解，必须具有最优子结构性质。所以，动态规划算法也需要证明(大多难度不大)。
  - 并不是所有组合优化问题都具有最优子结构性质。有时子问题的非最优解延伸为整个问题时成了最优解。
  - 反例：求总长模10的最短路径。
    - 采用动态规划算法，每步的最短路径值放节点上方。得到“下上上上上”路径的模10长度为1。
    - 但路径“下下下下”的模10长度为(5+5+5+5)mod10=0更短。

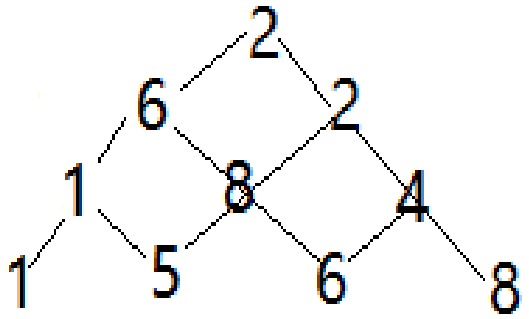


动态规划设计思想

- 不满足最优子结构例
  - 数字三角形问题：从三角顶部沿斜线往下走，找一条数字和最小的路径（到达最低层）。
  - 令D(x)表示从顶到第x层的最小路径值

    D(4)=2+6+1+1=10，但2+6+1=9不是D(3)的最优路径，

    D(3)=2+2+4=8是最优路径。



第五章 动态规划算法

- 5.2 矩阵连乘问题
  - 问题：给定n个数字矩阵A_1，A_2，…，A_n，其中A_i与A_{i+1}是可乘的，设A_i是$p_{i-1}\times p_i$矩阵, i=1,2,…,n。求矩阵连乘$A_1A_2\cdots A_n$的加括号方法，使得所用的乘次数最少。
    - 例：三个矩阵连乘，可以有$(A_1A_2)A_3$和$A_1(A_2A_3)$，乘法次数分别为：$p_0p_1p_2+p_0p_2p_3$和$p_0p_1p_3+p_1p_2p_3$
    - 例子：$p_0=10, p_1=100, p_2=5, p_3=50$, 两种方法的次数分别是：7500和 75000。
  - 如果使用蛮力算法，对所有可能的加括号方法递归搜索，则:

? $$T(n)=\begin{cases}1 & n=1\\ \sum_{k=1}^{n-1}T(k)T(n-k) & n>1\end{cases}$$,T(n)= $$C(n)=\frac{1}{n+1}\binom{2n}{n}=\Omega(4^n/n^{3/2})$$

矩阵连乘问题

- 矩阵连乘问题具有最优子结构性质：但有两个边界
  - 证明：设$(A_1 \ldots A_k)(A_{k+1} \ldots A_n)$ 具有最少乘法次数，则$(A_1 \ldots A_k)$中加括号的方法使A1..Ak乘法次数最少。否则设存在另一种加括号方法$(A_1 \ldots A_k)'$更优，则$(A_1 \ldots A_k)' (A_{k+1} \ldots A_n)$ 比$(A_1 \ldots A_k)(A_{k+1} \ldots A_n)$ 更优，矛盾。同理，$(A_{k+1} \ldots A_n)$ 内的连乘方法也是最优的。
- 用m[i][j]表示A_i到A_j连乘的最小次数，则有递推关系：

$$m[i][j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \{m[i][k] + m[k+1][j] + p_{i-1} p_k p_j\} & i < j \end{cases}$$

矩阵连乘问题

- 矩阵连乘的递归算法

```
int RecurMatrixChain(int i, int j)
 {
   if (i= =j) return 0;
   int u=RecurMatrixChain(i, i)
         +RecurMatrixChain(i+1,j)
         +p[i-1]·p[i]·p[j];
   s[i][j]=i;
   for(int k=i+1; k<j; k++){
    int t=RecurMatrixChain(i,k)
          +RecurMatrixChain(k+1,j)
          +p[i-1]·p[k]·p[j];
    if (t<u) {
      u=t;
      s[i][j]=k;
    }
   } //for 循环
  return u;
}
```

计算复杂度：

$$T(n)\geq\begin{cases}O(1) & n=1\\ 1+\sum_{k=1}^{n-1}(T(k)+T(n-k)+1) & n>1\end{cases}$$

$$T(n)\geq 1+(n-1)+\sum_{k=1}^{n-1}T(k)+\sum_{k=1}^{n-1}T(n-k)$$

$$=n+2\sum_{k=1}^{n-1}T(k)$$

用数学归纳法直接可证明：

$T(n)\geq 2^{n-1}=\Omega(2^n)$

尽管使用动态规划技术，复杂度并无本质改变！

原因：子问题重复计算。

矩阵连乘问题

- 矩阵连乘动态规划算法的迭代实现
  - void MatrixChain(int p, int n, int * *m, int * *s)

```
{ for (int i=1; i<=n; i++) m[i][i]=0;
  for (int r=2; r<=n; r++){      \\ r是跨度
    for (int i=1; i<=n-r+1; i++){
     int j=i+r-1;
     m[i][j]= m[i+1][j]+p[i-1]·p[i]·p[j]; //k=i时
     s[i][j]=i;
     for (int k=i+1; k<j; k++){
        int t= m[i][k]+m[k+1][j]+p[i-1]·p[k]·p[j];  //跨度更小的子问题利用
        if (t< m[i][j]) {                                    //前面结果，不用再计算
          m[i][j]=t; s[i][j]=k; }
     }
   }
} }
```

算法的时间复杂度：
3个for循环的规模均不超n，最内循环体内运算2次加法2次乘法，是常数时间，所以
$T(n)=O(n^3)$

矩阵连乘问题

- 算法过程示例：6个矩阵连乘：P=[30,35,15,5,10,20,25]
- 如：

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20=13000 \\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20=7125 \\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20=11375 \end{cases}$$

$$=7125$$

- 计算m[i][j]、s[i][j]的过程

$m[i][j]$

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

跨度增加方向

$s[i][j]$

	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

矩阵连乘问题

- 回溯构造最优解
  - void Traceback(int i, int j, int * * s)

```
{
  if (i= = j) return;
  Traceback(i, s[i][j], s);
  Traceback(s[i][j]+1, j, s);
   cout << "Multiply A" << i
        << "," << s[i][j];
   cout << "and A" <<(s[i][j] +1)
        << "," << j << endl;
}
```

以s[i][j]为元素的2维数组给出了加括号的全部的信息。因为s[i][j]=k说明，计算连乘积A[i..j]的最佳方式应该在矩阵A_k和A_{k+1}之间断开，即最优加括号方式为

(A[i..k])(A[k+1..j])。

可以从 s[1][n]开始，逐步回溯找出分点的位置，进而得到所有括号。

第五章 动态规划算法

- 5.3 0/1背包问题
  - 问题：容量为C的背包，n件物品，第i件重量和价值分别是w_i和p_i，i=1, 2, …, n。要将这n件物品的某些件完整地装入背包中。给出装包方法，使得装入背包的物品的总价值最大。
  - 数学描述：$\max \sum_{1\le i\le n} p_i x_i$，s.t. $\sum_{1\le i\le n} w_i x_i \le C, \quad x_i \in \{0,1\}, i = 1,2,\cdots,n$
  - 0/1背包问题具有最优子结构性质：若$y_1,y_2,\ldots,y_n$是原问题的最优解，则 $y_2,y_3,\ldots,y_n$是0/1背包问题的下述子问题：

    $\max \sum_{2 \le i \le n} p_i x_i$，s.t. $\sum_{2\le i\le n} w_i x_i \le c - y_1 w_1, \quad x_i \in \{0,1\}, i = 2,\cdots,n$ 的最优解。
    - 证：若不然，存在子问题更优解$z_2,z_3,\ldots,z_n$，

      $\sum_{2\le i\le n} p_i z_i > \sum_{2\le i\le n} p_i y_i$ 且 $\sum_{2\le i\le n} w_i z_i + w_1 y_1 \le C$，则

      $p_1 y_1 + \sum_{2\le i\le n} p_i z_i > \sum_{1\le i\le n} p_i y_i$，说明y1, $z_2,z_3,\ldots,z_n$是原问题的更优解，矛盾。

0/1背包问题

- 目标值递推关系式：
  - 设m[i][j]是容量为j，可选物品为第i,i+1,…,n件时的最优值，则有: (向前推)

$$\begin{cases} m[i][j]=\max\{m[i+1][j], m[i+1][j-w_i]+p_i\} & j \geq w_i \\ m[i][j]=m[i+1][j] & 0 \leq j < w_i \end{cases}$$

  - 初值：

$$m[n][j]=\begin{cases} p_n & if \quad j \geq w_n \\ 0 & if \quad 0 \leq j < w_n \end{cases}$$

  - 或设m[k][X]是容量为X，可选物品为1,2,..,k件时的最优值，则有：(向后推)：m[k][x]=m[k-1][x]，　0≤X<w_k

$$m[k][X]=\max\{m[k-1][X], m[k-1][X-w_k]+p_k\}$$ ，X ≥w_k

初值：m[1][X]=0,0≤X<w_1;m[1][X]=p_1,X≥w_1。

0/1背包问题

// 算法复杂度：O(nc)
//缺点：c很大时复杂度高
//w(i)、c必须是整数

```
Public static void Knapstack(int []p,int []w,int c,int [][]m) {
 int n=p.length-1;
 int jMax=Math.min(w[n]-1,c);
 for (int j=0;j<=jMax;j++) m[n][j]=0;
 for (int j=w[n];j<=c;j++) m[n][j]=p[n];
 for (int i=n-1;i>1;i--){
    jMax=Math.min(w[i]-1,c);
    for (int j=0;j<=jMax;j++) m[i][j]=m[i+1][j];
    for (int j=w[i];j<=c;j++) m[i][j]=Math.max(m[i+1][j],m[i+1][j-w[i]+p[i]);
  }
  m[1][c]=m[2][c]
  if (c>=w[1]) m[1][c]=Math.max(m[1][c],m[2][c-w[1]]+p[1]);
}
```

```
Public static void traceback(
      int [][]m,int []w,int c,int []x)
 { int n=w.lenth-1;
   for (int i=1,i<n,i++)
     if (m[i][c]==m[i+1][c])x[i]=0;
     else { x[i]=1;c-=w[i]; }
    x[n]=(m[n][c]>0?)1:0;
  }
```

第五章 动态规划算法

5.4 流水作业调度问题

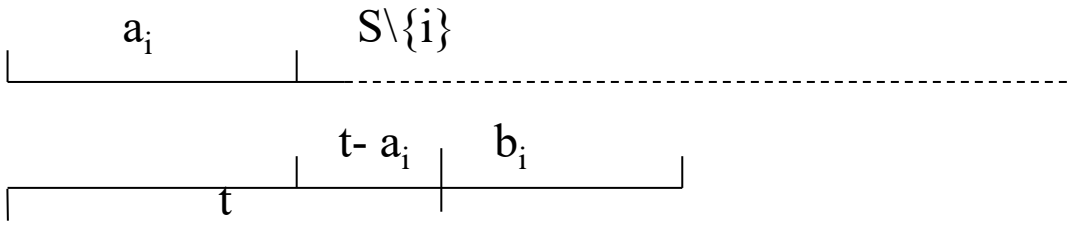
- 问题描述：n个作业{1, 2, . . . , n}，在两台机器M_1和M_2组成的流水线上完成加工。每个作业加工的顺序都是先在M_1上加工，然后在M_2上加工。M_1和M_2加工作业 i 所需的时间分别为a_i 和 b_i,$1 \le i \le n$。
  - 流水作业调度问题要求确定这n个作业的最优加工次序，使得从第一个作业在机器M_1上开始加工，到最后一个作业在机器M_2上加工完成所需的时间最少。
- 流水作业调度具有最优子结构性质
  - 设$S \subseteq N$，当机器M_1开始加工S中的作业时，机器M_2可能正在加工其它的作业，要等待时间t后才可用来加工S中的作业。这种情况下流水线完成S中的作业所需的最短时间记为T(S, t) 。
  - T(N,0)就是问题的解。设一个最优调度π的加工顺序是： π(1), π(2), …, π(n)，则π(2), …, π(n)是作业集N\{π(1)}在机器M_2等待时间为$b_{\pi(1)}$情况下的一个最优调度，且$T(N,0) = a_{\pi(1)} + T(N\backslash\{\pi(1)\},b_{\pi(1)})$。

流水作业调度问题

- 证明：
  - 若π′是作业集N\{π(1)}在机器M_2等待时间为$b_{\pi(1)}$情况下的一个最优调度，则加工次序 π(1), π′ (2), …, π′ (n) 是作业集N的一个调度(机器M_2不需等待时间)，它所用的加工时间即为：
    $a_{\pi(1)} + T(N\backslash\{\pi(1)\},b_{\pi(1)})$，
  - 但π是最优调度，所以$T(N,0) \le a_{\pi(1)} + T(N\backslash\{\pi(1)\},b_{\pi(1)})$。另一方面，$T(N,0)= a_{\pi(1)} + T'$， T′是机器$M_2$等待时间为$b_{\pi(1)}$时，按调度π加工作业集N\{π(1)}所用的时间，于是，
  - $T' \le T(N\backslash\{\pi(1)\},b_{\pi(1)})$，说明π(2), …, π(n) 也是作业集N\{π(1)}在机器M_2等待时间为$b_{\pi(1)}$情况下的一个最优调度。证毕。
- 目标值递推关系：
  - 初始时机器空闲：$T(N,0) = \min_{1\le i\le n}\{a_i + T(N\setminus\{i\},b_i)\}$
  - 一般情况：$T(S,t) = \min_{i\in S}\{a_i + T(S\setminus\{i\},b_i + \max\{t-a_i,0\})\}$

流水作业调度问题

- 其中 $b_i+\max(t-a_i,0)= b_i+\max(t, a_i)-a_i$是安排作业集S\{i}时，机器$M_2$需要等待的时间。



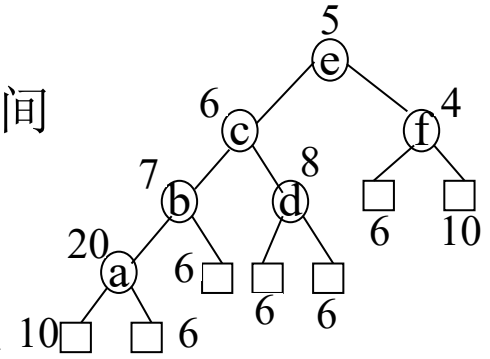
- 按此递推关系设计动态规划算法，时间复杂度
  $T(N)=N(1+T(N-1)=N+N(N-1)+N(N-1)T(N-2)...$
  $>N! > O(2^n)$。

第五章 动态规划算法

5.5最优二叉搜索树

- 问题：求平均路长最小的二叉检索树。例S=(a,b,c,d,e,f)
  - 在计算机中经常采用二叉树的结构来存储排好序的数据，称为二叉(分)检索树。设S={$x_1,x_2,\ldots,x_n$}是已非降排序的数据集，存储S的二叉树T以x_i为根，$x_1,x_2,..,x_{i-1}$都是T的左子树的内结点，$x_{i+1},x_{i+2},\ldots,x_n$则是T的右子树的内结点，而一个叶节点代表一个区间：(x_i,x_{i+1}) ,称T 为S二叉检索树。
  - 存取概率分布：x是数x_i的概率为b_i，位于区间 (x_i,x_{i+1}) 的概率为a_i，约定$x_0=-\infty$，$x_{n+1}=+\infty$。

$$a_i \ge 0,\ 0 \le i \le n, b_j \ge 0, 1 \le j \le n;\ \sum_{i=0}^{n} a_i + \sum_{j=1}^{n} b_j = 1$$

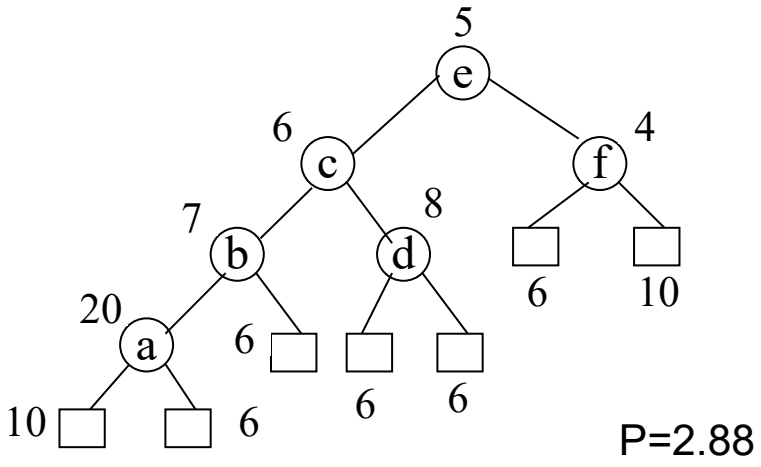


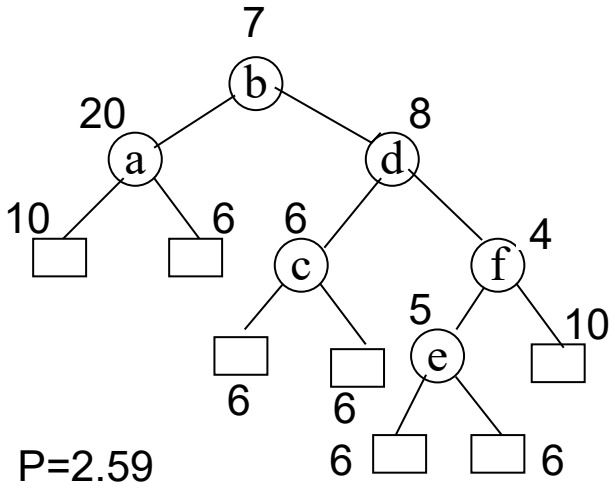
  - 平均路长(搜索次数)：存储x_i的节点的深度为c_i，

代表区间(x_i,x_{i+1})的叶节点的深度为d_i,平均路长为：$p = \sum_{i=1}^{n} b_i(1+c_i) + \sum_{j=0}^{n} a_j d_j$

最优二叉搜索树

- 对上例，
  - S={a,b,c,d,e,f}，a={10%,6%,6%,6%,6%,6%,10%}，b={20%,7%,6%,8%,5%,4%}
  - 下图两颗不同的二叉搜索树，平均路径P是不同的。





最优二叉搜索树

```
void OBSTree( int *a, int *b, int n,
              int  **m, int **s, int **w)
{
 for (int i = 0; i < n; i++) {
  w[i+1][i] = a[i];
  m[i+1][i] = 0;
 }
 for (int r = 0; r < n; r++) {
  for (int i = 1; i <= n-r; i++) {
    int j = i+ r ;
    w[i][j] = w[i][j-1] + a[j] +b[j];
    m[i][j] = m[i+1][j];  //k=i时
    s[i][j] = i;
   for (int k =i + 1; k<= j; k++) {
     int t = m[i][k-1] + m[k+1][j];
     if (t < m[i][j]) {
       m[i][j] = t;
       s[i][j] = k; }
    }
     m[i][j] + = w[i][j];
  }  // end  for i
 }  //end for r
}
```

算法的时间复杂度为：

$$T(n)=\sum_{r=0}^{n-1}\sum_{i=1}^{n-r}O(r+1)$$

$$= \quad O(n^3)$$

s[i][j]保存最优子树的根顶点中元素，s[1][n]=k表明整个二叉搜索树的根节点是x_k，通过s[1][k-1]和s[k+1][n]找到它的两个儿子，依次类推。

第五章 动态规划算法

- 动态规划算法设计的基本步骤
  - **分析最优解的结构**

    选定要解决问题的一个计算模型，其具有最优子结构性质
  - **建立递推关系式**

    关于目标值最优值的递推计算公式，有时可能不是一个简单的解析表达式。仔细划分子问题的边界和初值。
  - **设计求最优值的迭代算法**

    因为要使用已经处理过的子问题的计算结果，所以采用迭代方法，计算过程中需要保留获取最优解的线索，即记录一些信息。
  - **用回溯方法给出最优解**

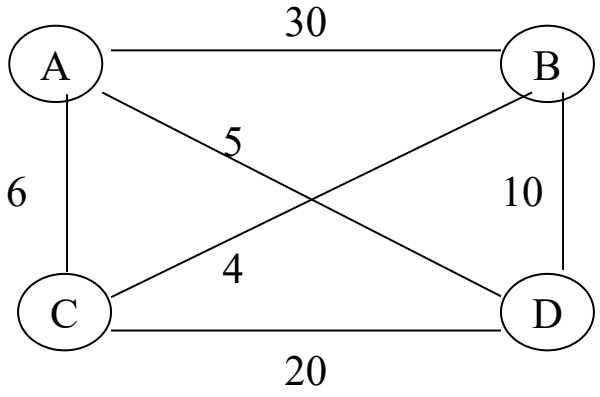
    把求最优值算法的计算过程倒回来，借用那里保留的信息就可追溯到最优解。

第六章 回溯算法

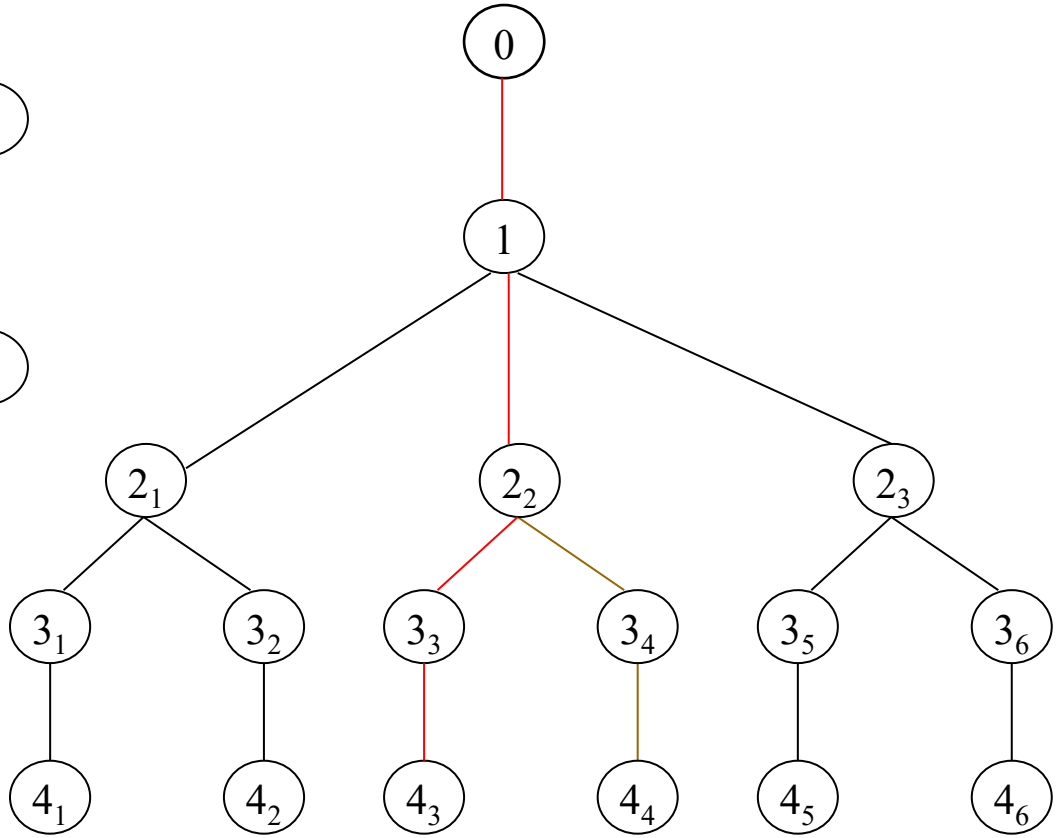
- 6.1回溯算法的基本思想
  - 回溯算法有“通用的解题法”之称。
  - 解空间
    - 多阶段解决问题的方案，所有可能的决策序列构成解空间。
    - k-定解空间：前面k个决策确定后所有可能的决策序列。从求解问题的角度看，每确定一组前k个决策就相当于问题求解到一个状态。未解状态对应0-定子空间。
  - 状态空间树
    - 以未解状态为根节点，确定各步决策的过程可以用一棵树描述出来。由根到所有其他节点的路径描述了问题的状态空间。
    - 解状态S：由根到S的路径确定了解空间的一个元组。
  - 可行解：满足约束条件的解，在状态空间树中称做答案节点。
    - 解向量：$x_1, x_2, \ldots, x_n$；显式约束：只约束变量的取值范围；隐式约束：强制变量之间的关联和制约；目标函数：极大或极小倾向或目的，有些问题不设目标函数。

回溯算法的基本思想

- 旅行商问题状态空间树

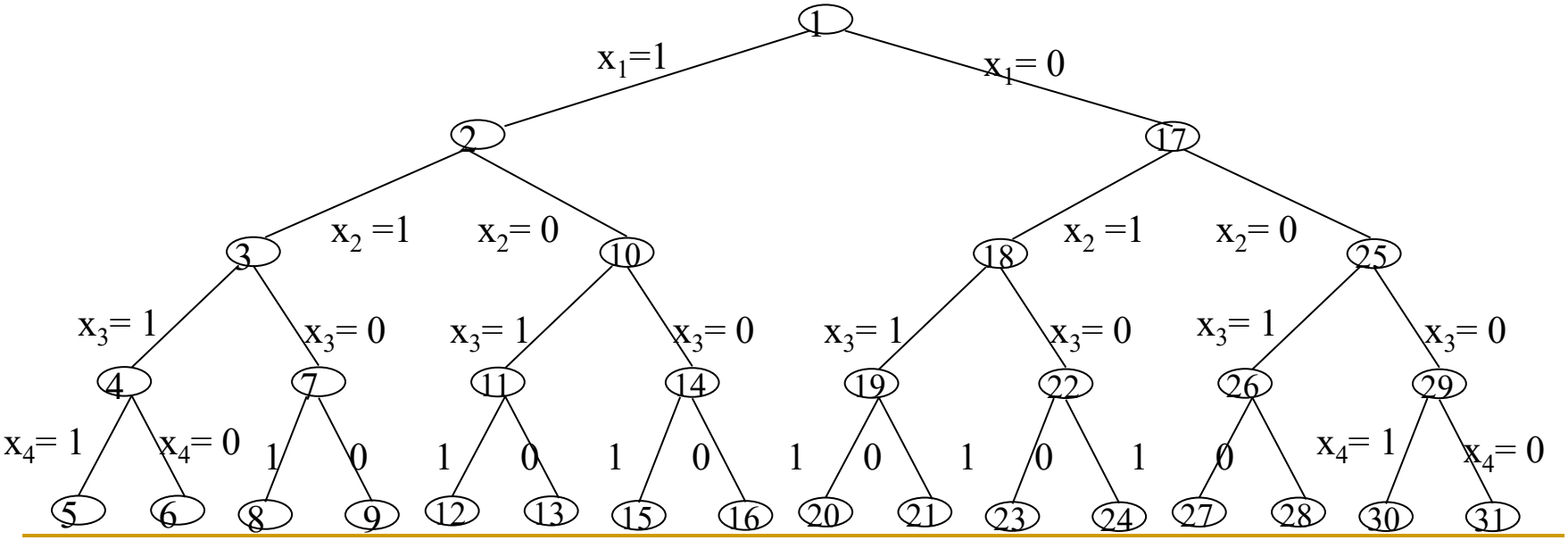


1. A；
2. AB, AC, AD;
3. ABC,ABD,ACB, ACD,ADB,ADC;
4. ABCD,ABDC, ACBD,ACDB, ADBC, ADCB



回溯算法的基本思想

- 定和子集问题状态空间树
  - 问题：已知一个正实数集合A={$w_1,w_2,..,w_n$}和正实数M，求A的所有子集S，使得S中的数之和等于M。
  - 数学模型：数的集合S={$w_1,w_2,\ldots,w_n$}; 定数：M，求解向量：x=($x_1,x_2,\ldots,x_n$)，$x_i \in \{0,1\}$，s.t. $\sum_{1\le i\le n} w_i x_i = M$ 。



回溯算法的基本思想

- 回溯算法求解过程
  - 确定了解空间的组织结构后，回溯法就是从根节点出发，以深度优先的方式搜索整个解空间。所以，
  - 回溯法是枚举的方法： $m=m_1m_2\ldots m_n$ 个可能的解——硬性处理。
  - 但它不是蛮力方法：
    - 剪枝：用约束条件(约束函数)、最优值的界（限界函数）限制搜索空间，剪掉不可能到达解节点的分枝。
  - 搜索过程
    - 针对问题确定解空间，确定易于搜索的解空间结构。
    - 以深度优先方式搜索解空间，边搜索边生成新节点，用剪枝函数避免无效的搜索。区分活节点、扩展节点、死节点。

第六章 回溯算法

6.2 定和子集问题

- 数学模型：数的集合S={$w_1,w_2,\ldots,w_n$}；定数：M，求解向量：x=($x_1,x_2,\ldots,x_n$)，$x_i \in \{0,1\}$，s.t. $\sum_{1\le i\le n} w_i x_i = M$ 。
- 约束条件：
  - 假定前k-1项选择已经确定，并且第k项已选择，使得 $\sum_{1\le i\le k} w_i x_i \le M$
  - 确定是否需要继续向前搜索：当 $\sum_{1\le i\le k} w_i x_i = M$ 时，已达到答案节点，回溯，搜寻其它的解；当 $\sum_{1\le i\le k} w_i x_i < M$ 时，继续向前搜索的条件是B_{k+1}：

    $$\sum_{1\le i\le k} w_i x_i + \sum_{k+1\le i\le n} w_i \ge M \quad \text{且} \quad \sum_{1\le i\le k} w_i x_i + w_{k+1} \le M$$

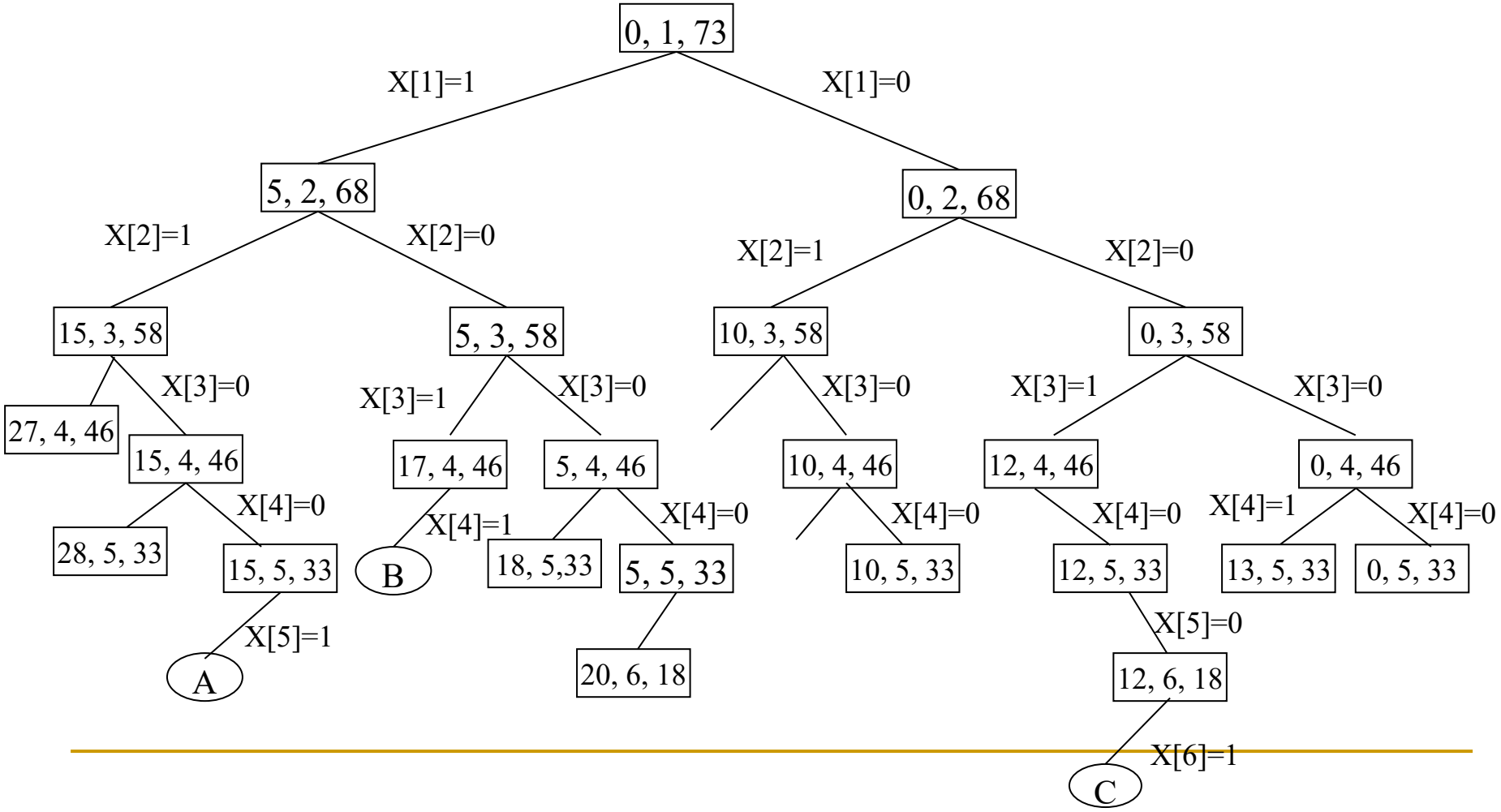
  - 此处，假定$w_1,w_2,\ldots,w_n$按不降顺序排列。

定和子集问题

```
SumSubset(s,k,r) {
 //寻找W[1..n]中元素和为M的
 //所有子集. W[1..n]中元素按不
 //降次序排列，进入此过程时，
 //X[1], . . . , X[k-1]的值已经确
 //定。记 s = ∑1≤i≤k-1 W[i]X[i] ,
 // r = ∑k≤i≤n W[i], 并假定
 //W[1] ≤ M， ∑1≤i≤n W[i]≥M。
 global integer M, n;
 global real W[1..n];
 global bool X[1..n];
 real r, s; integer k, j;
   //由于Bk = true，因此
   //s + W[k] ≤ M
 X[k]:=1; // 生成左儿子。
 if s + W[k]= M then
    print (X[j],j from 1 to k);
 else
   if s + W[k] + W[k+1] ≤ M then
     SumSubset(s+W[k], k+1, r-W[k]);
   end{if}
 end{if}
 if s + r–W[k]≥ M and s + W[k+1] ≤ M then
   X[k]:=0; //生成右儿子
   SumSubset(s,k+1,r-W[k]);
 end{if}
 end{SumSubset}
```

定和子集问题

- 定和子集问题例n=6,M=30; W[1:6]=(5,10,12,13,15,18)



第六章 回溯算法

- 6.3 0/1背包问题
  - 数学模型：物品重量：$w=(w_1,w_2,\ldots,w_n)$；价值：$p=(p_1,p_2,\ldots,p_n)$。求：解向量 $x=(x_1,x_2,\ldots,x_n)$，$x_i\in\{0,1\}$，$i=1,2,\ldots,n$

$$\max\sum_{i=1}^{n} p_i x_i \quad \text{s.t.}\sum_{i=1}^{n} w_i x_i \le M$$

  - 约束条件：在前k-1件物品是否装包的选择确定之后，确定第k件物品是否装进包内。装包条件是：$\sum_{i=1}^{k-1} w_i x_i + w_k \le M$
  - 限界函数：
    - 将物品按单位价值不增顺序排列：$p_1/w_1\ge p_2/w_2\ge\ldots\ge p_n/w_n$。
    - 根据背包问题贪心算法，其最优解是：x=(1,1,..1,t,0,0..,0)的形式,$0\le t\le 1$，可以用它估计前k-1件物品是否装包的选择确定之后的状态下可能达到的目标值的一个上界B_k，只有当目前所知最佳目标值低于这个上界时，即 $fp < B_k$时才有必要继续向前搜索。

0/1背包问题

- 限界函数

```
BoundF(cp,cw,k,M){  //前k-1件物品装包决策已定,考虑可能达到
                    //的最大目标值，返回一个上界。 cp、cw
                    //分别代表背包中当前物品的价值和重量。
 global n, p[1..n],w[1..n];
 integer k,i; real b,c,cp,cw,M;
b:=cp; c:=cw;
for i = k to n do
  c:=c+w[i];
  if c < M then b:=b+p[i];
   else
     return(b+(1-(c-M)/w[i])p[i]);
   end{if}
 end{for}    return(b);
end{BoundF}
```

0/1背包问题

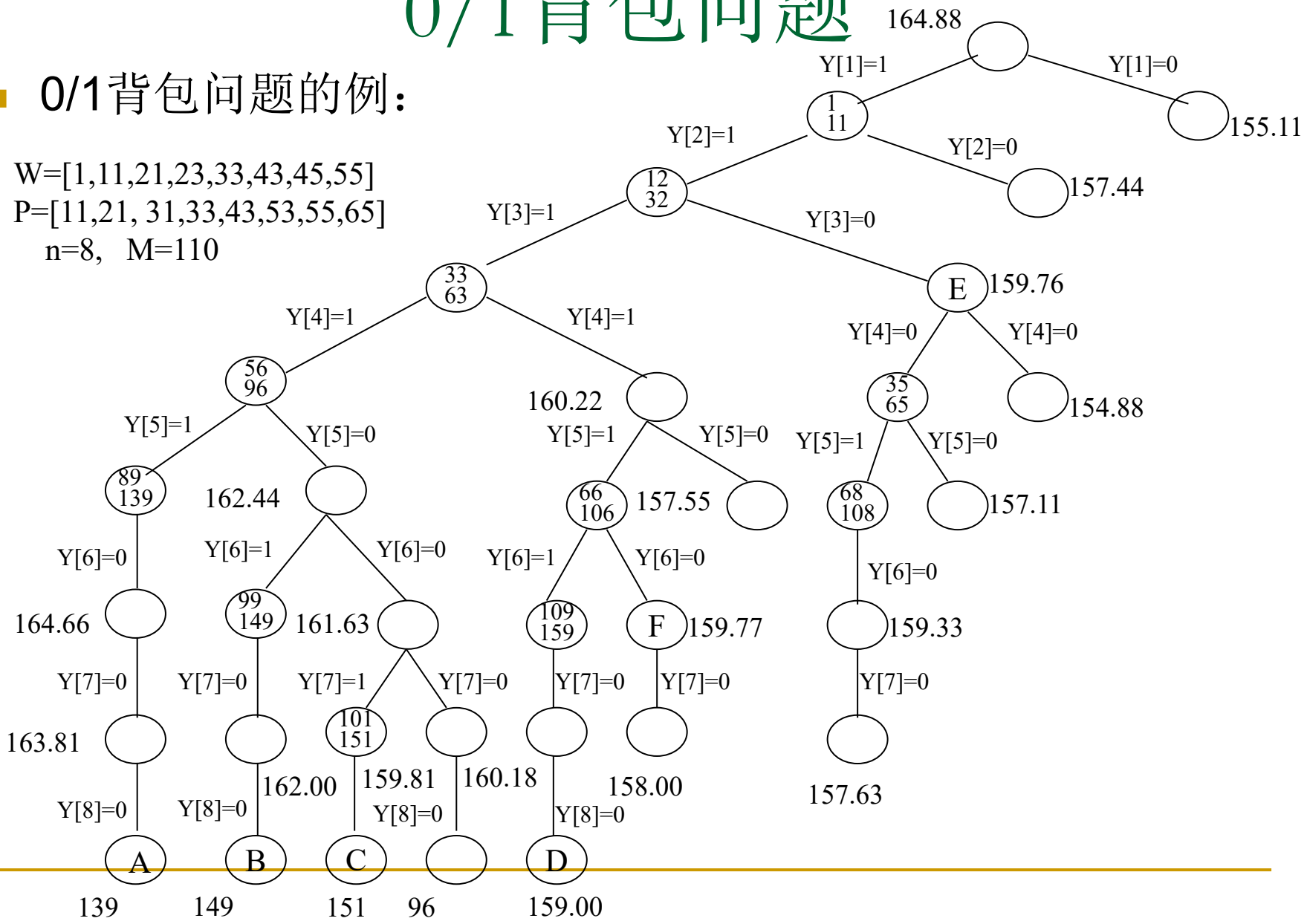
- 0/1背包问题的回溯算法

```
proc BackKnap(M,n,W,P,fp,X){
 //背包容量M.n件物品,重量W[1..n],
 //价值 P[1..n]，按单位价值的不增
 //顺序排列下标。当前所知最佳目
 //标值fp，cw, cp当前重量和价值
  integer n,k,Y[1..n],X[1..n];
  real M,W[1..n],P[1..n],fp,cw,cp;
  cw:=0; cp:=0; k:=1; fp:=-1;
  loop
    while k≤n and cw+W[k] ≤ M do
      cw:=cw+W[k]; cp:=cp+P[k];
      Y[k]:=1; k:=k+1;
    end{while}
    if k>n then
       fp:=cp; k:=n; X:=Y;//修改解
    else Y[k]:=0;
    end{if}
    while BoundF(cp,cw,k+1,M)≤fp do
         //剪枝，停止向前搜索
       while k≠0 and Y[k]≠1 do
          k:=k-1;
       end{while}
     if k:=0 then return; end{if}
      Y[k]:=0; cw:=cw-W[k];
      cp:=cp-P[k];
    end{while}
    k:=k+1;
  end{loop}
end{BackKnap}
```

0/1背包问题

- 0/1背包问题的例：

W=[1,11,21,23,33,43,45,55]
P=[11,21, 31,33,43,53,55,65]
n=8, M=110

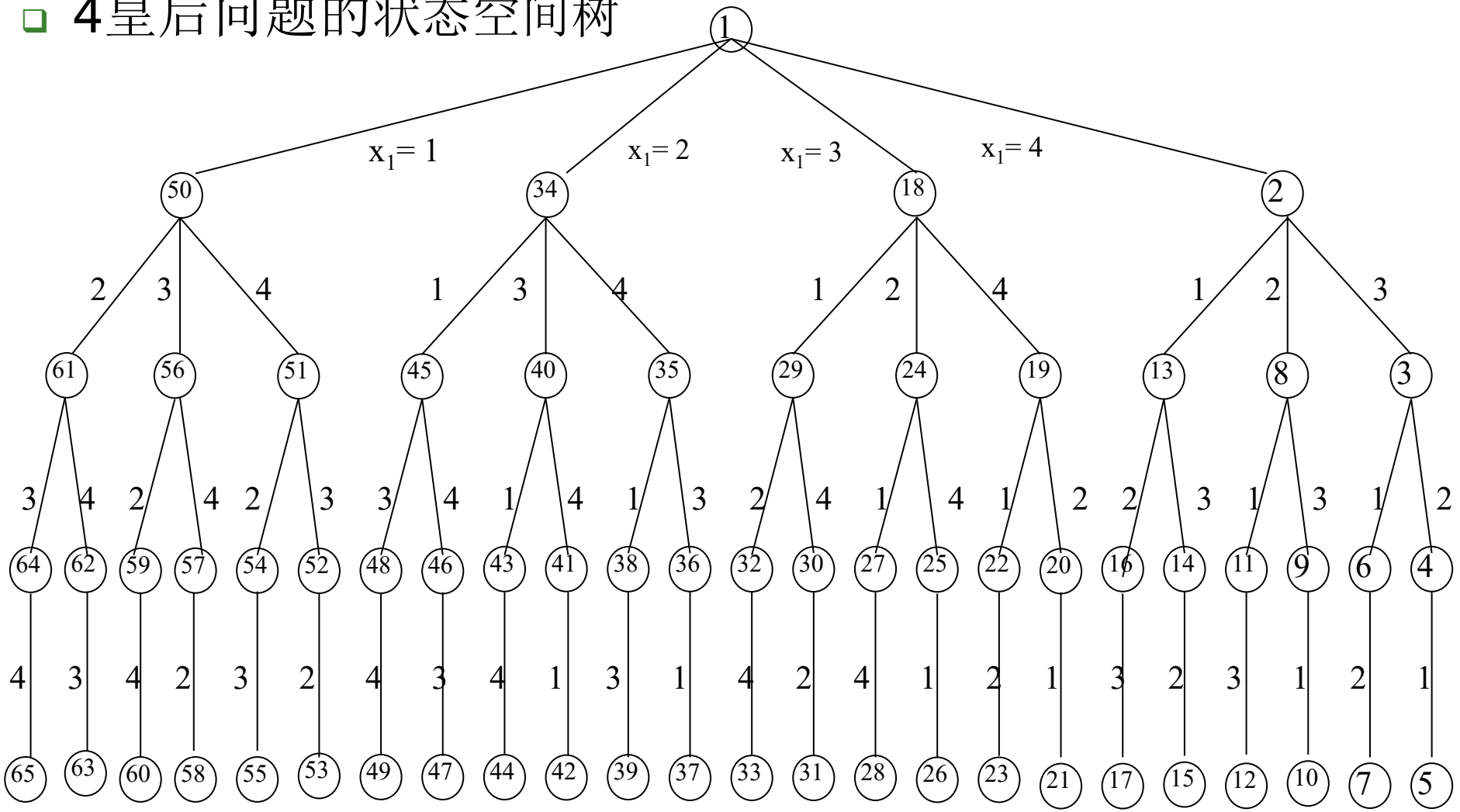


第六章 回溯算法

- 6.4 n皇后问题
  - 问题描述：在n×n棋盘上放置n个皇后，要使得每两个皇后之间都不能互相攻击，即任意两个皇后都不能放在同一行、同一列及同一对角线上。
  - 数学模型：
    - (1) 编号1,2,…,n代表n个皇后，$p=(p_1,p_2,..,p_n)$是编号的一个排列，p_i是棋盘坐标，代表安排第i个皇后的位置。每个皇后有n^2种可能，解空间为n^{2n}个决策序列。
    - (2) 将约束“任意两个皇后均不在同一行”考虑进来，不妨设第i个皇后在第i行，则问题为求解向量$x=(x_1,x_2,\ldots,x_n)$，它是1,2,…n的一个排列，第i个皇后放于x_i列。该模型解空间n!。
    - 约束条件：$x_i \in \{1,2,\ldots,n\}$； $x_i \neq x_j$ 当$i \neq j$时； $i \neq j \Rightarrow |x_i - x_j| \neq |i-j|$。

n皇后问题

- 4皇后问题的状态空间树



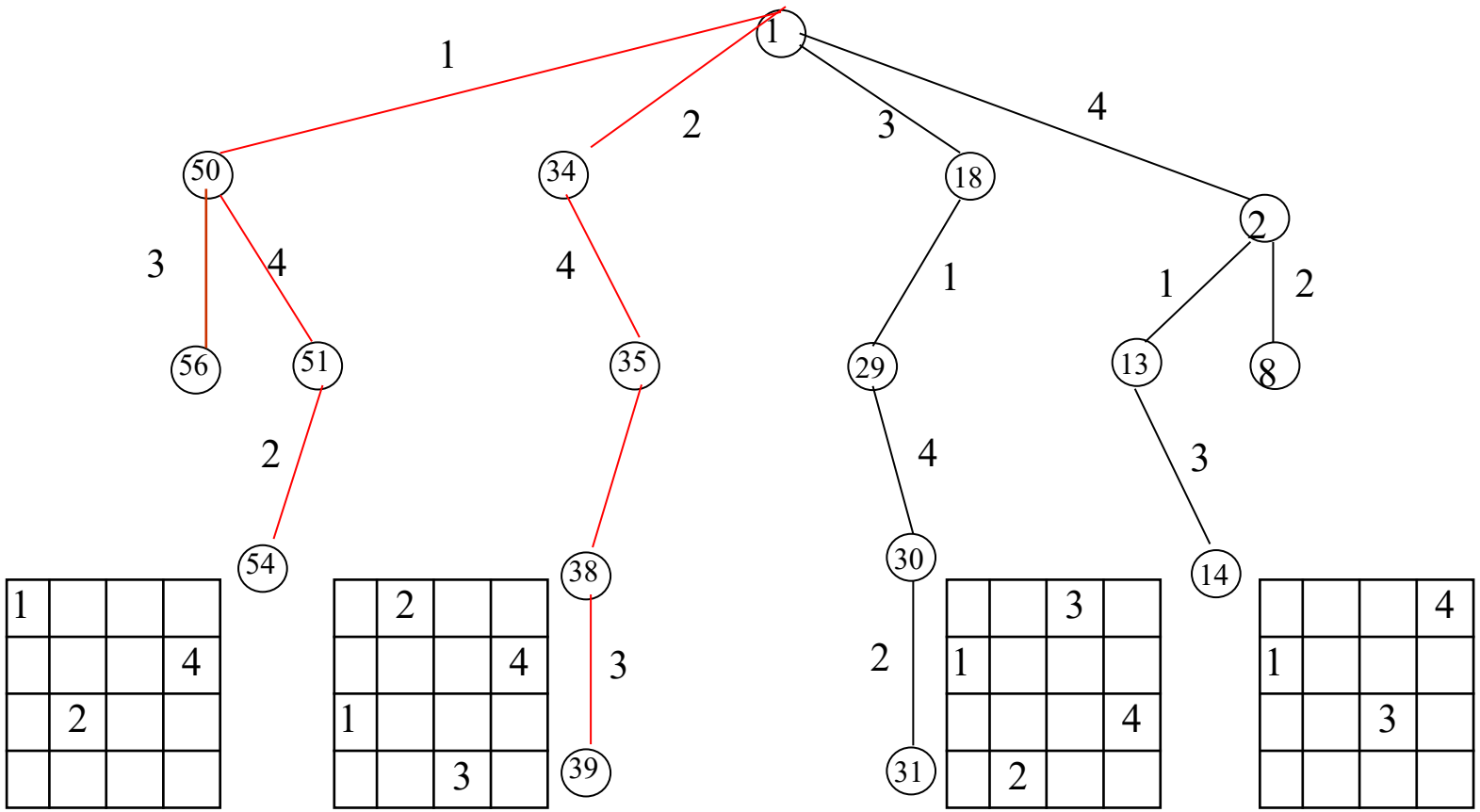
n皇后问题

```
Place(k)  //如果第k个皇后能放
  //在第X[k]列,则返回true,否则
  //返回false. X是一个全程数
  //组,进入此过程时已经确定了
  //前k-1个皇后的位置。
  global X[1..k]; integer i,k;   i:=1;
  while i<k do
    if X[i]=X[k] or  |X[i]-X[k]|=|i-k|
      then    return (false);
    end{if}
    i:=i+1;
  end{while}
  return(true);
end{Place}
```

```
proc nQueens(n) {
 integer k,n,X[1..n];   X[1]:=0; k:=1;
 while k>0 do  //k是当前行,X[k]是当前列
    X[k]:=X[k]+1;//转到下一列
    while X[k]≤n and Place(k)=false do
      X[k]:=X[k]+1;
    end{while}
    if X[k]≤ n then
      if k=n then
        Print(X);
      else k:=k+1; X[k]:=0; //转到下一行
      end{if}
    else k:=k-1;//回溯
    end{if}
 end{while}
end{nQueens}
```

n皇后问题

- 4皇后解空间树的搜索情况



回溯算法

6.5 旅行商问题

- 用1,2,…,n-1,n代表n个城市的顶点; 城市 i, j 之间的路程 w(i, j);

  数组 $(i_1,i_2,\ldots,i_n)$ 代表一个周游 $i_1\ i_2\ \ldots\ i_n\ i_1$。
- 可行解的前 k-1个分量$x_1,\ldots,x_{k-1}$确定后，判定$x_1\ldots x_{k-1}x_k$能否构成一条路径，需要检查 $x_k \neq x_1, \ldots, x_k \neq x_{k-1}$ ——约束条件
- 当前路长：$cl = w(x_1, x_2)+w(x_2, x_3)+ \ldots +w(x_{k-2}, x_{k-1})$,
- 已知最佳游路长：fl。如果 $cl + w(x_{k-1}, x_k) > fl$ (限界函数)

  则 $x_1\ldots x_{k-1}x_k$不是最短周游路径的一部分，因而不必继续向前搜索，即状态空间树中此状态后面的子树被剪掉（剪枝）。
- 更新最佳目标值：当 k = n时，如果

  $$cl + w(x_{k-1}, x_k) +w(x_k, x_1) < fl$$

  则令 $fl = cl + w(x_{k-1}, x_k) +w(x_k, x_1)$ 。

旅行商问题

```
proc BackTSP(n,W) //W是G的邻接矩阵, cl是当前路径的长度，
  //fl是当前所知道的最短周游长度.
  integer  k,n,X[1..n];
  real W[1..n,1..n],cl,fl;
 for i =1 to n do
   X[i]:=0;
 end{for}
 X(1)=1;k:=2; cl:=0; fl:=+∞;
 while k>1 do
   X[k]:=X[k]+1 ;          //给X[k]预
   while X[k]< =n do     //分配一个值
     if  NextValue(k)=true then
       cl:=cl+W(X[k-1],X[k]); break;
     end{if}
     X[k]:= X[k]+1  ;  //重新给
   end{while}          //X[k]分配值
 if X[k]<=n then
   if  fl ≤ cl or (k=n and
                fl < cl+W[X[k],1])  then
     cl:=cl-W(X[k-1],X[k]);
   elseif k=n and fl ≥ cl+W(X[k],1) then
     fl:=cl+W(X[k],1);
     cl:=cl-W(X[k-1],X[k]);
   else k:=k+1;x(k):=0//继续纵深搜索
   end{if}
 else k:=k-1; cl:=cl-W(X[k-1],X[k]);
 end if
end{while}
end{BackTSP}
```

注：此程序与讲义略有不同。

旅行商问题

- **NextValue**(k) //进入此过程时已经
  //设置了k个值,其中X[1]=1, X[k]是
  //当前扩展节点.。若X[k]是前面某
  //个节点X[i](i<k),则返回false,否则
  //返回true. X是一个全程数组。
  global X[1..k]; **integer** i,k;
  i:=1;
  while i<k **do**
    if X[k] = X[i] **then**
      return (false);
    end{if}
    i:=i+1;
  end{while}
  return(true);
  end{NextValue}

回溯算法

- 6.6 图的着色问题
  - 问题：给定无向图G和m种颜色，求出G的所有m-着色。
  - 建模：
    - 图G有n个顶点，邻接矩阵W=(w_{ij})；m种颜色：1, 2, … , m。
    - 解向量X=(x_1,x_2,…,x_n)，X[i]=k表示顶点i被着以颜色k。X[i] = 0表示顶点 i 未被着色。
    - 当顶点1,2,…, j-1已经着好颜色，那么顶点j要着的颜色应该满足约束条件： W[i,j]=1 $\Rightarrow$ X[i]≠X[j], 1≤i<j
  - 搜索策略
    - 选择顶点j的颜色时采用巡回选色的方法；出现选不到合适的颜色时就回溯，修改前一个顶点已着的颜色，没有可换颜色时再向前回溯，如此等等。
    - 每完成最后顶点的着色就打印该着色方案，并回溯。

图的着色问题

```
proc GraphColor(k) // W[1..n, 1..n]是
 //图G的邻接矩阵，m种颜色 1, 2,…, m。
 //下一个要着色的顶点 k。初始时，
 //数组X的每个分量已经赋值0。
  global integer m, n, X[1..n], W[1..n, 1..n];
 loop
  if k=0 then exit; end{if}
  NextColor(k); // 确定X[k]的取值
  if X[k]=0 then   k:=k-1;//说明没有颜色可
  else if k=n then          //以分配给第k个点
        print(X); //已经找到一种着色方法
     else
      k:=k+1;        //下一个顶点
     end{if}
  end{if}  end {loop}
end{GraphColor}
```

```
NextColor ( j ) //X[1],…,X[j-1]已经
 //确定且满足约束条件。本过程给
 //X[j]确定一个整数k，0≤k≤m。
integer  j, k;
loop
  X[j]:=X[j]+1 mod (m+1);
  if X[j]=0 then return; end{if}
  for i =1 to j-1   //验证约束条件
    if W[i,j]=1 and X[i]=X[j] then
      break;  //该颜色不满足约束
    end{if}
  end{for}
  if i=j then return; end{if}
   //找到一种颜色
  end{loop}
end{NextColor}
```

回溯算法

- 6.7回溯算法的效率分析
  - 回溯算法的抽象描述

```
BACKTRACK(n)
 integer k:=1, n; global integer X(1..n);
  while k > 0 do
   if 还剩有没有被检验过的X(k)使得
    X(k)∈ T(X(1), … , X(k-1)) and
    Bk(X(1), … , X(k-1),X(k)) = true   then
     if (X(1), … , X(k-1),X(k)) 是一个解 then
       print(X(1), … , X(k-1),X(k));
     end{if};
     k:=k+1;  //考虑下一个集合
   else k:=k-1; //回溯先前的集合
   end{if};
 end{while}  end{BACKTRACK}
```

//每个解都在X(1..n)中生成，
//一个解一经确 定就立即打
//印。在X(1), … , X(k-1)已经
//被选定的情况下，
//T(X(1), … , X(k-1))给出X(k)
//的所有可能的取值。函数
//$B_k(x_1, x_2, \ldots, x_{k-1}, x_k)$ 给出哪
//些元素X(k)满足约束条件。

回溯算法的效率分析

- 影响算法效率的因素:

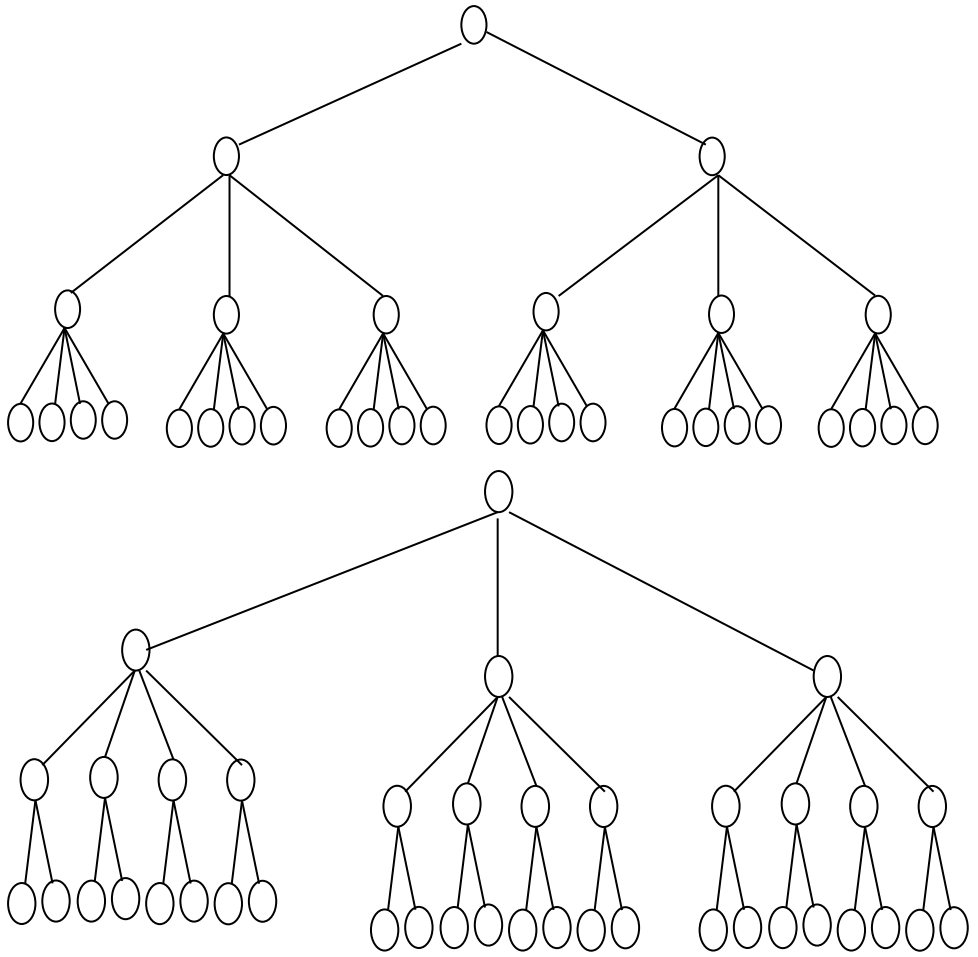
  1) 诸x_k的取值范围;

  2) 产生诸x_k所用的时间

  3) 计算剪枝函数的时间;

  4) 约束函数与限界函数的强度。

- 重排原理
  - 解向量中，可能取值少的分量尽量放在前头。
  - 状态空间树结构与解向量分量的排序有关,在右面第一个状态空间树中搜索比第二个中搜索剪枝力度大。



第二个状态空间树只是交换了解向量分量的位置

回溯算法的效率分析

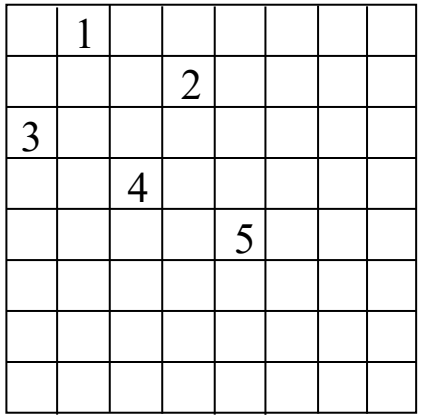
- 蒙特卡罗效率估计
  - 状态空间树上节点的个数：假定x_k有m_k种可能的取值，则解空间树中节点总数为$m=1+m_1+m_1m_2+\ldots+m_1\ldots m_n$
  - 回溯算法在搜索所有可行解过程中，由于采用剪枝函数，并没有生成所有节点。
  - 当$x_1,\ldots,x_{k-1}$的值取定之后，x_k必须在$T(x_1,\ldots,x_{k-1})$中取值，且满足约束条件B_k。
  - 采用“随机生成路径”的办法估计回溯算法对于具体实例所产生的状态空间树中节点的数目，以此来估计算法的效率。

```
Estimate //程序沿着状态空
  //间树中一条随机路径估计
  //回溯法生成的节点总数 m
  m:=1; r:=1; k:=1;
  loop
   T:={X[k]： X[k]∈
     T(X[1], …， X[k-1]) and
     Bk(X[1], …， X[k])=true};
     if T ={} then exit; end{if}
     r:=r*size(T); m:=m+r;
    X[k]:=Choose(T); k:=k+1;
  end{loop}
  return(m);
end{Estimate}
```

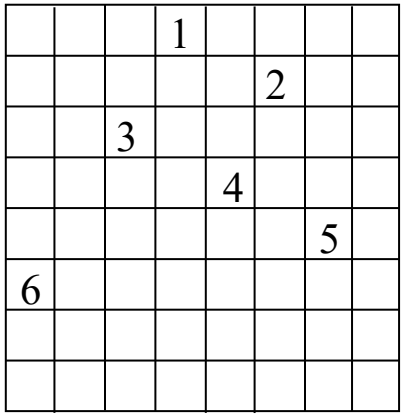
  - Choose(T)是从T中随机地挑选一个元素。

回溯算法的效率分析

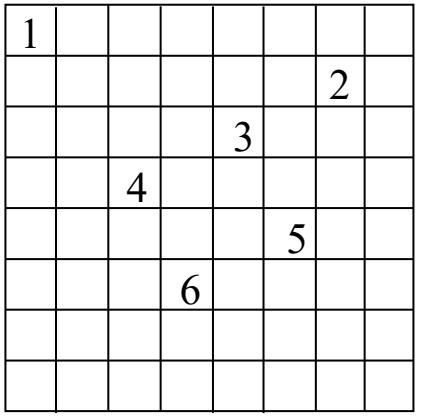
- 回溯法效率估计实例：8皇后问题



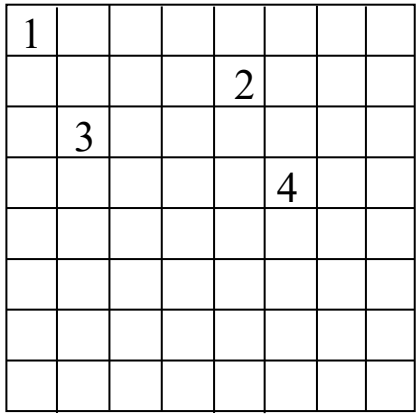
(8,5,4,3,2)=1649



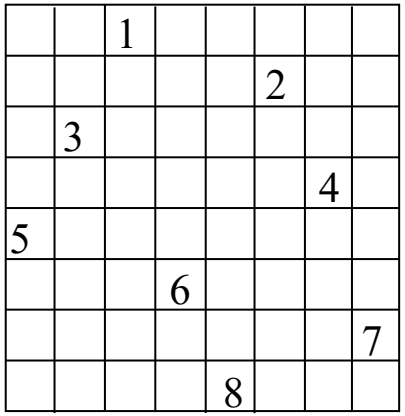
(8,5,3,2,2,1)=1369



(8,6,3,3,2,2)=3225



(8,6,3,2)=489



(8,5,3,2,2,1,1,1)=2329

8皇后状态空间树的节点数的估计值是1812.2

而解空间树的节点总数为

$$1+\sum_{j=0}^{7}\prod_{i=0}^{j}(8-i)=109601$$

$1812.2/109601\approx 1.65\%$

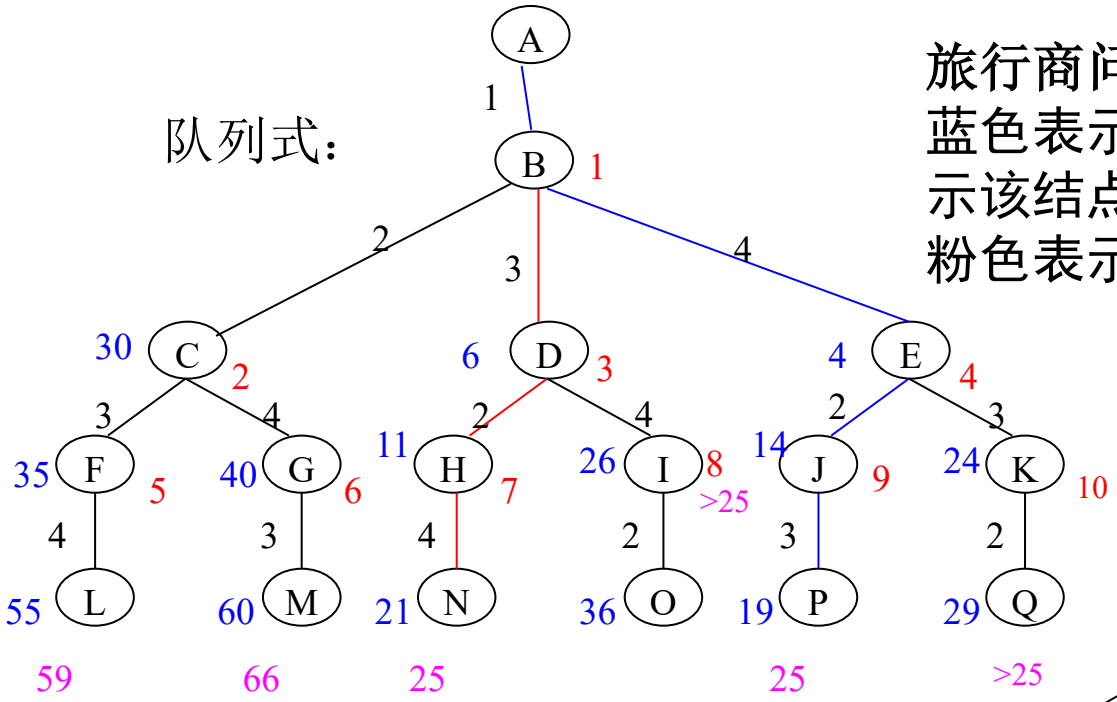
第七章 分枝限界算法

7.1 分枝限界算法基本思想

- 同回溯法，也是在解空间中搜索，生成状态空间树；但采用宽度优先搜索，用表记录活节点。
- 在扩展节点处，首先生成其所有的儿子节点,将那些导致不可行解或导致非最优解的儿子节点舍弃，其余儿子节点加入活节点表中。然后，从活节点表中取出一个节点作为当前扩展节点，重复上述节点扩展过程。
- **队列式分枝限界算法**：将活节点组织成先进先出（FIFO）或后进先出(LIFO)队列，不满足约束条件的节点不放入队列。
- **优先队列式分枝限界算法**：将活节点根据优先级组织成最大堆或最小堆，优先级高的首先取作当前扩展节点(可能导致非宽度优先)。
- 节点的优先级常常根据目标函数确定，最大化问题常引用一个可能获得的最大目标值的一个上界；最小化问题则使用可能获得最小目标值的一个下界。这两个界都是动态确定的。

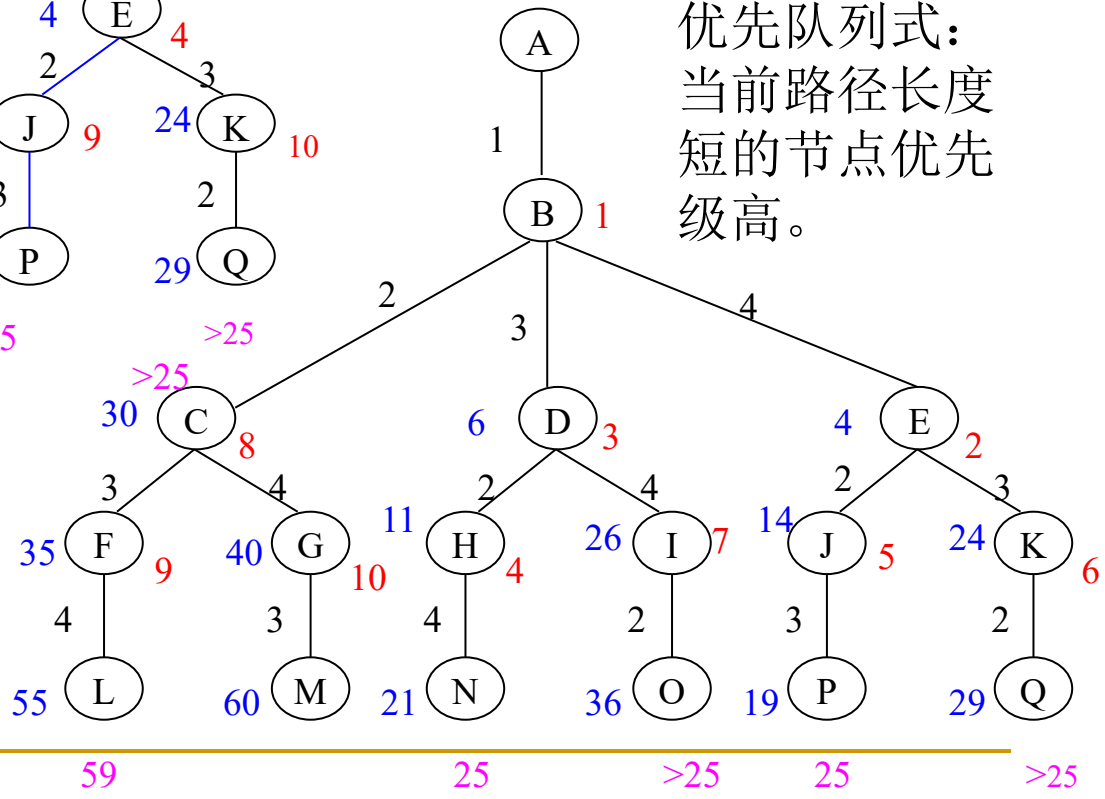
分枝限界算法基本思想

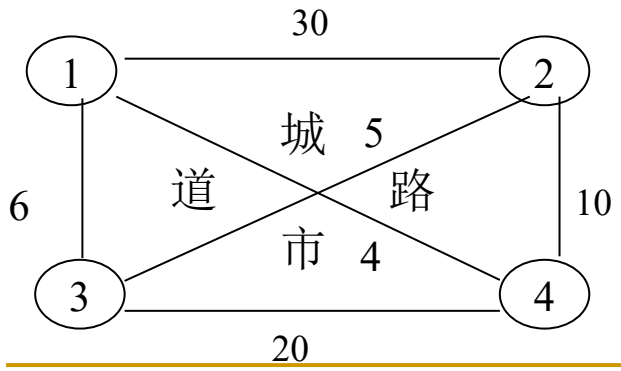
队列式：



旅行商问题的两种分枝限界算法：蓝色表示该节点的费用，红色表示该结点成为扩展节点的序号，粉色表示相应周游的费用。

优先队列式：当前路径长度短的节点优先级高。





分枝限界算法

- 7.2 0/1背包问题的优先队列式分枝限界算法
  - 用优先队列式分枝定界法解0/1背包问题需要确定：
    - 1) 解空间树中节点的结构；
    - 2) 如何生成一个给定节点的儿子节点；
    - 3) 如何组织活节点表；
    - 4) 如何识别答案节点。
  - 每个节点X有六个信息段：
    - Parent：节点X的父亲节点连接指针； Level：节点X在解空间树中的深度； Tag：标记输出最优解的各个分量x_i； CC：记录背包在节点X处(状态下)背包的剩余空间； CV：记录背包在节点X处(状态下)背包内物品的价值； CUB：背包在节点X处可能达到的物品价值上界估值 Pvu。
    - 目标值动态预测prev：到目前为止所知道的最佳目标值。

0/1背包问题

- 活节点表的组织：采用优先队列
  - 信息段CUB中的值做为确定该节点优先级的依据。
  - 如果Pvu(X)≤prev，则杀死节点X，即X不放入节点表。
- 六个辅助子程序
  - LUBound：计算当前被搜索节点的Pvl和Pvu值；
  - NewNode：生成新节点，给各个信息段置入适当的值，并将此节点加入节点表；
  - Init：对可用节点表和活节点表置初值；
  - Largest：在活节点表中取一个具有最大Pvu(存在活节点的CUB域)值节点作为当前扩展节点；
  - Finish：打印出最优解的值和此最优解中的物品标号。

0/1背包问题的分枝界限算法

```
proc LCKNAP(P,W,M,N)//物品序号
 //满足： P[i]/W[i]≥P[i+1]/W[i+1];
 real M, Pvl, Pvu, cap, cv, prev ;
 real P[1..N],W[1..N]; integer ANS,X,N;
 Init; //初始化可用节点及活节点表
 GetNode(E); //生成根节点
 Parent(E):=0; Level(E):=0;
 CC(E):=M; CV(E)=0;
 LUBound(P,W,M,0,N,1,Pvl,Pvu);
 prev:=Pvl- ε; CUB(E):=Pvu; Tag(E):=0;
 loop
   i:=Level(E)+1, cap:=CC(E), cv:=CV(E);
   case:
      i=N+1: //E是解节点
        if cv > prev then
           prev:=cv; ANS:=E; //cv已实现
        end{if}
else: //E是内部节点,有两个儿子
   if cap≥W[i] then //左儿子可行
      NewNode(E,i,1,cap-W[i],
                          cv+P[i],CUB(E));
   end{if}
   LUBound(P,W,cap,cv,N,i+1, Pvl,Pvu);
   if Pvu>prev then //右儿子会活
      NewNode(E,i,0,cap,cv,Pvu);
      prev:=max(prev,Pvl-ε); //Pvl未实现
   end{if}
 end{case}
 if 不再有活节点 then exit; end{if}
 Largest(E);//取下一个扩展节点
 until CUB(E)≤prev
 Finish(cv,ANS,N);
end{LCKNAP}
```

0/1背包问题的分枝界限算法

```
NewNode(par,lev,t,cap,cv ,ub)
  //生成一个新节点J，并
  //把它加到活节点表
  GetNode(J);
  Parent(J):=par;
  Level(J):=lev;
  Tag(J):=t;
  CC(J):=cap;
  CV(J):=cv;
  CUB(J):=ub;
  Add(J);
end{NewNode}
```

```
Finish(CV,ANS,N)//输出
解
 real CV;
 global Tag,Parent;
 print('OBJECTS IN
          KNAPSACK ARE')
 for j from N by –1 to 1 do
  if Tag(ANS)=1 then
    print(j);
  end{if}
  ANS:=Parent(ANS);
 end{for}
end{Finish}
```

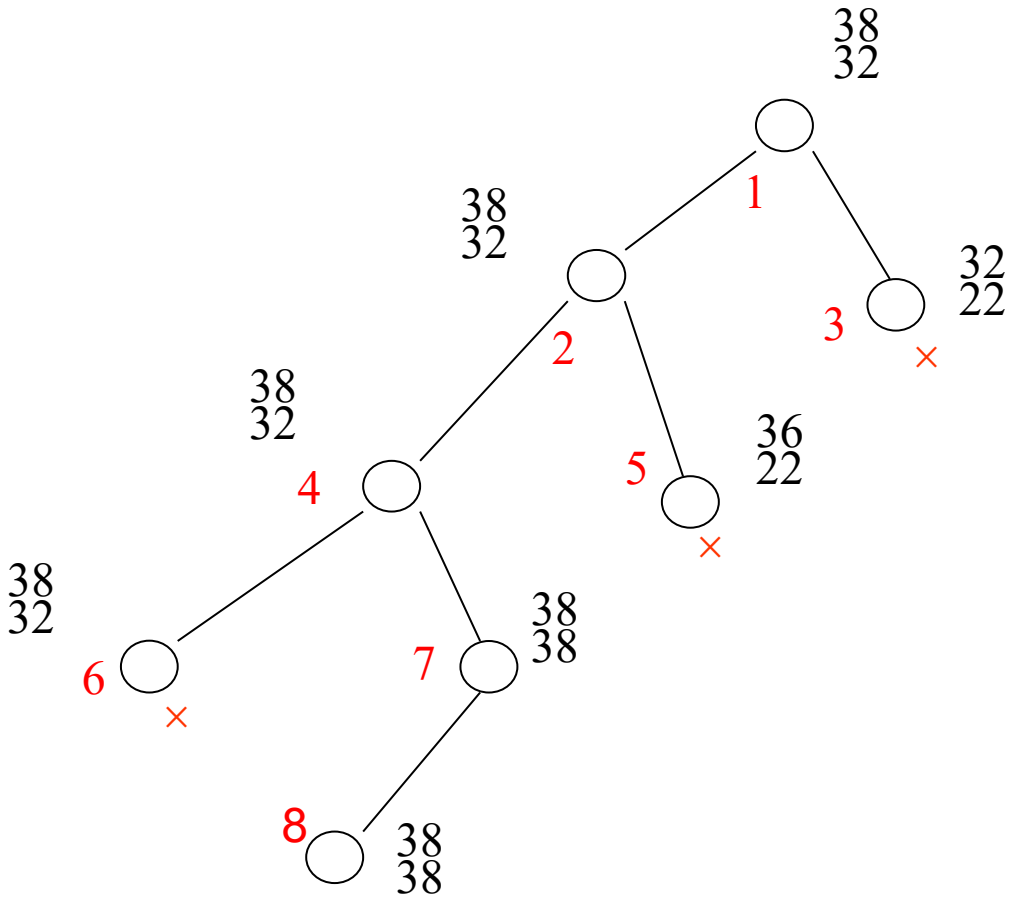
0/1背包问题的分枝界限算法

LUBound(P,W,cap,cv,N,k,Pvl,Pvu) // k为当前节点的级，cap是背包当前的剩余容量，cv是当前背包中物品的总价值，还有物品k,…,N要考虑。

```
 Real rw;    Pvl:=cv; rw:=cap;
 for i from k to N do
   if rw<W[i] then Pvu:=Pvl+rw*P[i]/W[i];
     for j from i+1 to N do  // 第k件到第N件至少有一件物
       if rw≥W[j] then       //品不能装进背包的情形出现
         rw:=rw-W[j]; Pvl:=Pvl+P[j];
       end{if}
     end{for}
     return   //此时Pvl < Pvu
  end{if}
  rw:=rw-W[i]; Pvl:=Pvl+P[i];
 end{for}
 Pvu:=Pvl;  // 从第k件物品到第N件物品都能装进背包的情形出现,
end{LUBound}
```

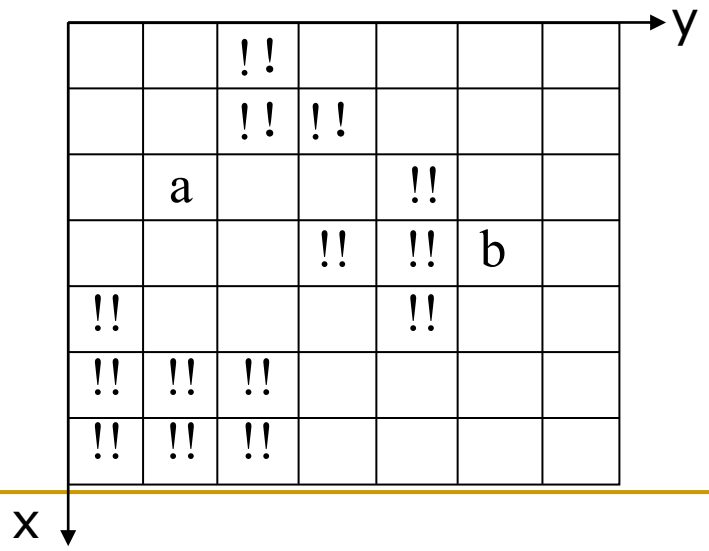
0/1背包问题的分枝界限算法

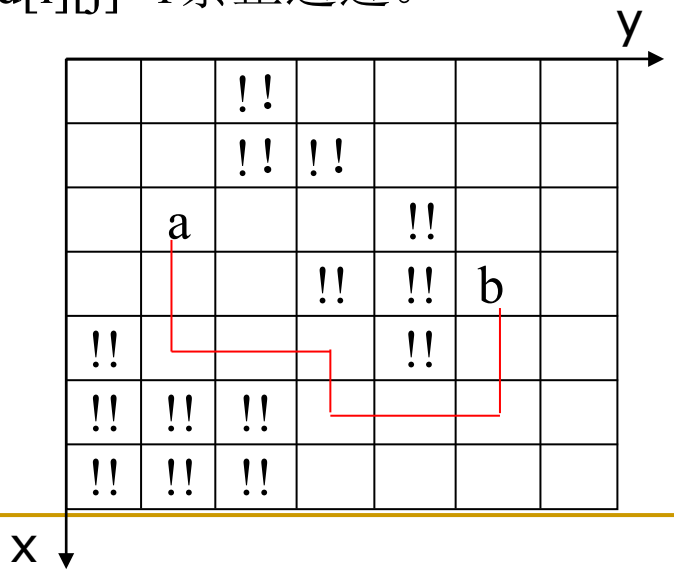
- **例子** n=4,
  P=(10,10,12,18),
  W=(2,4,6,9),
  M=15.
- 算法 LCKNAP
求最优解的检索过程。
- 上：pvu
  下：pvl



分枝限界算法

- 7.3 电路板布线问题
  - 问题：印刷电路板将布线区域分成n×m个方格(阵列)，某些方格有禁入标记(已经布线)。确定连接两个指定方格a和b间的最短折线布置(只能走直线或直角)方案。
  - 模型：方格位置用类Position描述：私有成员 row, col；如a=(3,2), b=(4,6)。平移offset：右(0)、下(1)、左(2)、上(3)，如向右平移一步表示为：offset[0].row=0 and offset[0].col=1。
  - 方格状态： grid[i][j]=0可以通过， grid[i][j]=1禁止通过。





电路板布线问题

- 算法设计：采用队列式分枝限界法
  - 解空间是一颗多叉树。从a开始向相邻逐格延伸，直到某条路径与b相连。每延伸一步到达一个新的方格，形成一个树结点。
  - 以a为第一个扩展结点，与扩展结点相邻且可达的方格为可行结点，按右、下、左、上顺序进入活结点队列。
  - 与a相连的结点标记为1，表示从a到该方格的距离；与距离1相连的结点标记为2，等等。
  - 程序中1用于封锁标记，从2开始标记距离。
  - 依次搜索和扩展活节点表，直到遇到b。扩展时遇到标记封锁的方格时，放弃此方格。

3	2	!!				
2	1	!!	!!			
1	a	1	2	!!		
2	1	2	!!	!!	b	
!!	2	3	4	!!	8	9
!!	!!	!!	5	6	7	8
!!	!!	!!	6	7	8	

电路板布线问题

```
bool FindPath(Position start, Position finish,
              int& PathLen, Position * &path)
{//计算从起点位置start到目标位置
  //finish的最短布线路径.找到最短布
  //线路径则返回true,否则返回false
 if((start.row= =finish.row) &&
                  (start.col= =finish.col)
 {PathLen=0; return true;} //start=finish
   //设置方格阵列“围墙”
  for(int i=0; i<= m+1; i++)
    grid[0][i]=grid[n+1][i]=1;
     //顶部和底部
  for(int i=0; i<= n+1; i++)
    grid[i][0]=grid[i][m+1]=1;
    //左翼和右翼
```

```
Position offset[4]; //初始化相对位移
offset[0].row=0; offset[0].col=1;//右
offset[1].row=1; offset[1].col=0;//下
offset[2].row=0; offset[2].col=-1;//左
offset[3].row=-1; offset[3].col=0;//上
int NumOfNbrs=4;//相邻方格数
Position here, nbr;
here.row=start.row;
here.col=start.col;
grid[start.row][start.col]=2;
//标记可达方格位置
LinkedQueue<Position> Q;
do {    //标记相邻可达方格
  for(int i=0; i<NumOfNbrs; i++){
    nbr.row=here.row + offset[i].row;
    nbr.col=here.col+offset[i].col;
```

电路板布线问题

```
  if(grid[nbr.row][nbr.col]==0){
    //该方格未被标记
    grid[nbr.row][nbr.col]
        =grid[here.row][here.col]+1;
    if((nbr.row= =finish.row) &&
      (nbr.col= =finish.col)) break;
      //完成布线，跳出for
    Q.Add(nbr);}
  }     //若到达目标位置跳出do
  if((nbr.row= =finish.row) &&
   (nbr.col= =finish.col)) break;
  //活结点队列是否非空？
  if(Q.IsEmpty()) return false;//无解
  Q.Delete(here);//取下个扩展结点
}while(true);
//构造最短布线路径
PathLen=grid[finish.row][finish.col]-2;
path=new Position[PathLen];
//从目标位置finish开始向起始位置回溯
here=finish;
for(int j=PathLen-1; j>=0; j--){
   path[j]=here; //找前驱位置
   for(int i=0; i<NumOfNbrs; i++){
    nbr.row=here.row+offset[i].row;
    nbr.col=here.col+offset[i].col;
    if(grid[nbr.row][nbr.col]==j+2)
       break;
    }
    here=nbr;//向前移动
 }
 return true;
}
```

分枝限界算法

- 7.4 优先级的确定与LC-检索
  - 优先队列式分枝限界算法优先级函数的确定
    - 理想的当前扩展节点X

      1) 以X为根的子树中含有问题的答案节点；

      2) 在所有满足条件1)的活节点中，X距离答案节点“最近”。
    - 节点计算代价的度量

      (i) 在生成一个答案节点之前，子树X需要生成的节点数；

      (ii) 以X为根的子树中，离X最近的那个答案节点到X的路径长度。
    - 最低搜索成本函数c(.)

      a) 如果X是答案节点，则c(X)是解空间树中由根节点到X的成本(即所用的代价，如深度、计算复杂度等)；

      b) 如果X不是答案节点，而且以X为根的子树中不含答案节点，则c(X)定义为∞；

      c) 如果X不是答案节点，但是以X为根的子树中含答案节点，则c(X)是具有最小成本的答案节点的成本。

优先级的确定与LC-检索

- 搜索成本估计函数
  - C(X)往往是不易得到的，实际问题中都是采用一个成本估计函数ĉ(.)。它由两部分组成： ĉ(.) = f(X) + g(X)。
  - f(X)—解空间树根节点到X的成本；g(X)—X到答案节点的计算成本。
- 最小成本搜索：LC-检索

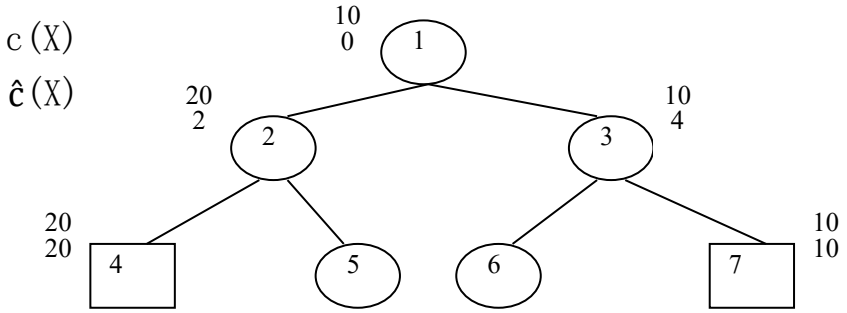
根据成本估计函数选择下一个扩展节点的策略总是选取ĉ(·)值最小的活节点作为下一个扩展节点。

(1)如果g=0，f(x)等于x在解空间树中的深度，则LC-检索即是宽度优先搜索。

(2)如果f=0，而且g满足：Y是X的儿子→g(Y) ≤g(X)，则LC-检索即是深度优先搜索(回溯法)。

优先级的确定与LC-检索

- 成本估计函数应满足的条件
  - 按照成本估价函数ĉ(X)确定的优先级进行搜索，所得到的答案节点未必是最小成本答案节点。



  - **定理** 7.4.1 在有限的解空间树中，如果对每对节点X和Y都有“c(X) < c(Y)” ⇒ “ĉ(X) < ĉ(Y)”，则按照最小成本估计函数搜索能够达到最小成本答案节点。
  - 一般情况下，对于成本估计函数有一个基本要求：
    ĉ(X)≤ c(X)，对任意节点X；ĉ(X)＝c(X)，当X是答案节点时。

优先级的确定与LC-检索

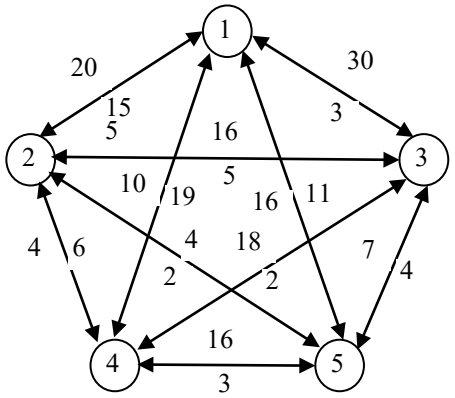
- 最小化问题的LC－分枝限界算法

```
proc LCBB(T, ĉ,u,ε,cost)        //假定解空间
//树T包含一个解节点且 ĉ(X) ≤ c(X) ≤ u(X)。
//c(X)是最小成本函数，ĉ(X)是成本估价函
//数 ,u(X)是限界函数; cost(X)是X所对应的
//解的成本。ε 是一个充分小的正数。
    E:=T; Parent(E):=0;
    if  T 是解节点 then
      U:=min(cost(T),u(T)+ ε); ans:=T;
    else U:=u(T)+ε; ans:=0;
    end{if}
    loop
     for E 的每个儿子X  do
       if ĉ(X)<U && X是一个可行节点
       then
         Add(X); Parent(X):=E;
      case:
       X是解节点 && cost(X)< U:
         U:=min(cost(X),u(X)+ε);
         ans:=X;
       u(X)+ε < U:   U:=u(X)+ε;
     end{case}
    end{if}
  end{for}
  if 不再有活节点 or 下一个扩展
    节点满足 ĉ ≥ U then
    print('least cost=',U);
    while ans≠0 do
      print(ans); ans:=Parent(ans);
    end{while};    return;
  end{if}
  Least(E);//找最小ĉ(E)扩展
end{loop}   end{LCBB}
```

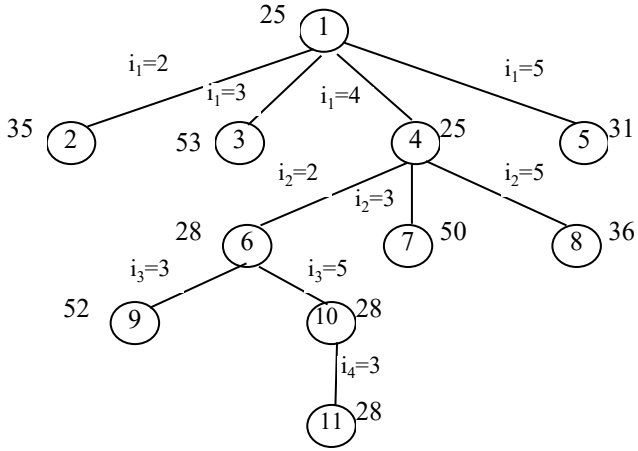
分枝限界算法

- 7.5 旅行商问题的LC-分枝限界算法
  - 例：

具有 5 个城市的旅行商问题



序号由小到大权值标于外侧，由大到小标于内侧



节点外面的数字是ĉ 值

$$A=\begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{pmatrix}$$

$$A(1)=\begin{pmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{pmatrix}$$

旅行商问题的LC-分枝限界算法

- 优先级函数
  - $c(X)=\begin{cases}\text{从根到}X\text{的路径所定义的回路成本，当X是叶节点时}\\ \text{子树}X\text{中一个最小成本叶节点的成本，当}X\text{不是叶节点时}\end{cases}$
  - $\hat{c}(X)$的定义
    - 可以定义$\hat{c}(X)$=为根节点到节点X的路径的成本，显然有$\hat{c}(X)\le C(X)$。但当X是叶节点时，不满足$\hat{c}(X)=C(X)$。
    - 简约矩阵：各行、列都至少有一个元素是零的非负矩阵。成本矩阵A可通过先将每行减掉最小元素、再将每列减掉最小元素得到其简约矩阵。
    - 简约矩阵不改变最小成本回路性质：因为一条环游回路含有每个顶点i的恰好一条出边(i,j)和一条入边(k,i),每行或每列都减去常数t，会使每条回路都减少t。
    - 如果i行减掉r_i，j列减掉c_j，则 $\hat{c}=\sum_{i=1}^{n} r_i+\sum_{j=1}^{n} c_j$ 小于等于最小回路成本。$\hat{C}$可作为根节点的下界估值。

旅行商问题的LC-分枝限界算法

- 其它节点的X的$\hat{c}(X)$：

  设R是S的父节点，(R,S)对应Hamilton回路中包含的边(i,j)。

  - 如果S不是叶节点，S的简约矩阵A_S可以通过修简R的简约矩阵A_R而得：

  (1)将A_R中i行和j列的所有项都改为∞，防止任何其它离开顶点i的边，进入顶点j的边的使用。

  (2)将A_R的(j,1)元素置为∞，防止使用边(j,1)。

  (3)约简经过(1)、(2)两步操作后得到的矩阵。给出节点S的下界估值如下：$\hat{c}(S)=\hat{c}(R)+A_R(i,j)+\tilde{c}$ ，其中，$\tilde{c}$是约简步骤(3)施行时减掉的总数，$A_R(i,j)$是A_R的(i,j)元素。

  - 如果S是叶节点，则直接计算 $\hat{c}(S)=c(S)$ 即可。

- 上界函数U(X)；初始U=+∞，解节点U(X)=c(X)。

问题的计算复杂度

- 以比较为基础的检索问题的时间下界：Ω(logn)
  - 构造决策树：设A是一个以比较为基础的检索算法，对于给定的规模为n的实例，A的决策树的结点为1,2,…,n，标记规则：
    - (1)根据算法A，如果x第一次与L[i]比较，将树根标记为i；
    - (2)假设某结点已被标记为i，分下述情况处理：
    - ① 当x<L[i]时，算法下一步与x比较的是L[j]，那么i的左儿子标记为j；如果算法本次比较后停止，i没有左儿子。
    - ②当x>L[i]时，算法下一步与x比较的是L[j]，那么i的右儿子标记为j；如果算法本次比较后停止，i没有右儿子。
    - ③当x=L[i]时，算法停止，i没有儿子。
  - 例：n=9时，顺序检索和折半检索的决策树(L[]已不降排序)。
  - 给定规模为n的输入L和x，算法A将从这颗具有n个结点树的根开始，沿某条路径前进，直到某个结点停止。最坏路径长度为树高k+1。内结点数为n， $n \le 2^{k+1}-1$ ，所以$k=\Omega(\log n)$。

问题的计算复杂度

- 以比较为基础的排序问题时间下界:$T(n)=\Theta(n\log n)$
  - 决策树的构造：设A是以元素比较为基本运算的排序算法，输入L={$x_1,x_2,\ldots,x_n$}，它的决策树结点如下生成：
    - (1)A的第一次比较元素为x_i,x_j，那么记树根为(i,j)。
    - (2)假设结点k已标识为(i,j)，按照下面规则标记k的儿子：
    - ① 当$x_i<x_j$时，若算法结束，k的左儿子标记为输出；若下一步比较元素x_p,x_q，那么k的左儿子标记为(p,q)；
    - ②当$x_i>x_j$时，若算法结束，k的右儿子标记为输出；若下一步比较元素x_p,x_q，那么k的右儿子标记为(p,q)；
  - 对于任意输入L，A的排序过程所进行的比较操作对应了上述决策树中从根到某片树叶的一条路径上的结点个数。不同的排序算法对应的树可能不同，但树叶数都是n!个，因为 n个不同元素的不同排列共有n！种。

问题的计算复杂度

- 选择问题的时间复杂度

问题	算法	最坏情况	问题下界	最优性
找最大	Findmax	n-1	n-1	最优
找最大最小	FindMaxMin	$\lceil 3n/2 \rceil -2$	$\lceil 3n/2 \rceil -2$	最优
找第二大	锦标赛	$n+\lceil \log n \rceil -2$	$n+\lceil \log n \rceil -2$	最优
找中位数 找第k小	Select Select	O(n) O(n)	3n/2-3/2 n+min(k,n-k+1)-2	阶最优 阶最优

- 证明：参考书4,p146-p152

计算复杂性与NP完全问题

- 8.5 NP完全问题(NP-C)
  - 研究NP类中问题之间的关系，从NP中找出一些具有特定性质的、**与P中问题有显著不同**的问题。形成了NP－完全理论
  - 多项式变换：语言$L_1 \subset \Sigma_1^*$到另一个语言$L_2 \subset \Sigma_2^*$的多项式变换是指映射f: $\Sigma_1^* \rightarrow \Sigma_2^*$，满足下面两个条件：

    1. 存在计算 f 的一个多项式时间DTM程序；
    2. 对于所有的$x \in \Sigma_1^*$有：$x \in L_1$当且仅当$f(x) \in L_2$。

    记号： $L_1 \propto L_2$。 $\Pi_1 \propto \Pi_2$ 如果 $L(\Pi_1, e_1) \propto L(\Pi_2, e_2)$
  - NP－完全问题：称一个语言L(判定问题Π)是NP－完全的，如果$L \in NP$($\Pi \in NP$)，而且，对于**任意**$L' \in NP$($\Pi' \in NP$) 都有

    $L' \propto L$ ($\Pi' \propto \Pi$) ，所以， NP完全问题是NP中最难的。
  - NP-难：对于任意$\Pi' \in NP$，都有 $\Pi' \propto \Pi$，称 Π是NP-难的。NP-难的问题不会比NP中的任何问题容易。

NP完全问题

- 几个性质与定理：

  性质1. 若$L' \propto L$, $L \in P$，则$L' \in P$;

  性质2. 若$L' \propto L$，$L \propto L''$, 则 $L' \propto L''$

  定理1. 若判定问题Π是NP一完全的，$P \neq NP$，则$\Pi \in NP \backslash P$;

  定理2. 若L', $L \in NP$， $L' \propto L$ 则 $L' \in NPC \Rightarrow L \in NPC$。

- 定理2是证明NP完全问题的基础
  - 证明一个判定问题$\Pi \in NP$。
  - 找到一个已知的NP完全问题Π'，并证明$\Pi' \propto \Pi$。
- Cook定理:可满足性问题是NP-完全问题(第一个NP完全问题,1970)
  - 定义：给定布尔变量集$U=\{u_1,u_2,\ldots,u_m\}$，U上的子句c定义为：$c=z_1 \vee z_2 \vee \ldots \vee z_k$，$z_i \in U \cup \bar{U}$，$\bar{U}$ 表示U中变量的非的全体。对子句集合$C=\{c_1,c_2,\ldots,c_n\}$称为可满足的，如果存在U的一个真值分配，使C中每个子句取值为真。判定问题例：给定U及C。
  - 问：是否存在的一个真值分配，使得C是可满足的。

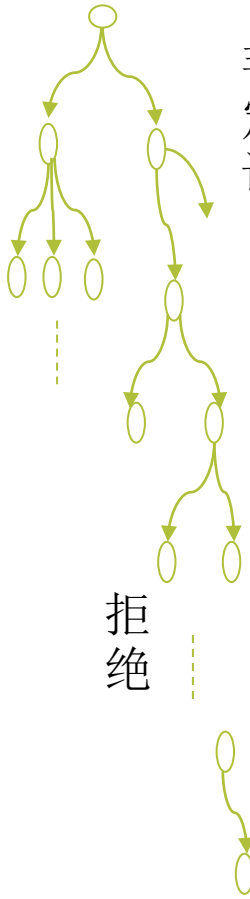
NP完全问题

- 可满足性(SAT)例
  - U={u_1,u_2,u_3}，
  - C_a={c1,c2}， c1=$(\overline{u_1} \vee \overline{u_2} \vee u_3)$ ，c2=$(u_1 \vee \overline{u_2})$
  - C_b={c1,c2,c3}，c1=$(u_1 \vee u_2)$，c2=$(\overline{u_1} \vee u_2 \vee u_3)$ ，c3=$\overline{u_2}$
  - C_c={c1,c2,c3,c4}，c1=$(u_1 \vee \overline{u_2} \vee u_3)$ ，c2=$(\overline{u_1} \vee \overline{u_2} \vee u_3)$

    c3=u_2，c4=$\overline{u_3}$
  - 令t(u_1)=1, t(u_2)=0, t(u_3)=1，则t是C_a、C_b的成真赋值，从而C_a、C_b是可满足的；t不是C_c的成真赋值，事实上C_c是不可满足的。
  - 因为，只有t=*10才可以使c3、c4满足，但110不能满足c2,010不能满足c1。

NP完全问题

- Cook定理的证明 $u(z_1,z_2,...z_n)=(z_1+z_2+...+z_n)\prod_{i\neq j}(\overline{z_i}+\overline{z_j})$
  - 使用NDTM模型。易证SAT∈NP。对任何NP判定问题的一个长度为n的输入W，NDTM可在p(n)步判定。
  - 设共有m个带符号、q个状态，则判定过程为：
    - t1：i_1格、符号为x_{j1}、状态为q_{k1}； t2: i_2格、符号为x_{j2}、状态为q_{k2}; t3，……，$t_{p(n)}$；
    - 定义c(i,j,t)为布尔变量：第t步，i格，符号为x_j则为1，否则0
    - 定义s(t,q_i)为布尔变量：第t步, 状态为q_i为1，否则为0
    - 一个猜想对应一个状态：t、i、x_i、q_k，对应相应布尔量为真
    - 定义谓词u(z_1,z_2,…z_r)=1,当且仅当z_i一个取值为真
    - 定义C=∏u(s(t,0),s(t,1),s(t,2)…s(t,q))，0≤t≤p(n)，
    - C成真什么含义？同理构造A、B、D等代表读哪个格、什么字符的“猜想”值。如D=∏u(c(1,x1,t),c(2,x1,t),…c(m,x_m,t))
  - 任何NTDM判定问题∝ SAT问题:W=ABCD为真。



NP完全问题

- 几个典型的NP－完全问题
  - 2. 图的顶点覆盖问题

    例：给定一个图G(V,E)和一个正整数K≤|V| 。

    问：是否存在G的一个顶点数不超过K的覆盖(G的任一边至少有一个顶点在$V' \subseteq V$中，称V'为G的一个顶点覆盖)?
  - 3. Hamilton回路问题

    例：给定一个图G(V,E)。

    问：G含有一个Hamilton回路吗?
  - 4. 划分问题

    例：已知一个有限集合A，其每个元素a都赋予一个权值$s(a) \in Z^+$

    问：是否存在A的子集A'使得$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)$。

NP完全问题

- 5.三元可满足性问题 3SAT

例：给定布尔变量的一个有限集合U及定义于其上的子句集 $C=\{c_1,c_2,\ldots,c_m\}$, 其中$|c_i|=3$, $i=1,2,\ldots,m$。

问：是否存在U之上的一个真值分配，使得C中所有的子句均被满足？

- 6. 恰好覆盖问题

例：给定有限集$A=\{a_1,a_2,\ldots,a_n\}$和A的子集的集$W=\{S_1,S_2,\ldots,S_m\}$ 。

问：存在子集$U\subseteq W$使得U中的子集都不相交且它们的并集等于A?

称W这样的子集U是A的恰好覆盖。

- 7. 0/1背包判定问题

例：给定一个有限集合X，对每一个$x\in X$，对应一个值 $w(x) \in Z^+$ 和相应的值$p(x) \in Z^+$。另外，还有一个容量约束$M \in Z^+$和一个价值目标 $K \in Z^+$ 。

问：是否存在X的一个子集$X'\subseteq X$,使得$\sum_{x\in X'} w(u)\le M$,而且$\sum_{x\in X'} p(x)\ge K$

计算复杂性与NP完全问题

- 8.6 证明新问题是NPC问题
  - 证明方法：
    - 证明 $\Pi \in NP$
    - 选取一个已知的NP完全问题 Π' ；
    - 构造一个从 Π' 到 Π 的变换f；
    - 证明f为一个多项式变换。
  - 证明的起点和依据：
    - Cook定理： 可满足性问题是NPC-问题。
    - 定理2：若 $L', L \in NP$， $L' \propto L$ 则 $L' \in NPC \Rightarrow L \in NPC$。

证明新问题是NPC问题

- 证明 0/1背包判定问题是NPC问题
  - 例：给定一个有限集合X，对每一个 x∈X，对应一个值 w(x)∈Z^+,和相应的值 p(x) ∈Z^+。另外，还有一个容量约束 M∈Z^+ 和一个价值目标K∈Z^+ 。
  - 问：是否存在X 的一个子集 X/ , 使得$\sum_{x\in X'} w(u) \le M$，而且$\sum_{x\in X'} p(x) \ge K$
- 证明: (限制法)
  - 0/1背包判定问题显然是NP问题。现在我们考虑特殊的 0/1 背包问题：对所有的x∈X 有 w(x)=p(x), 且取$M = K = \frac{1}{2}\sum_{x\in X} w(x)$。
  - 显然，这个0/1背包判定问题回答为“是”当且仅当集合X的划分问题回答为“是”。可见，划分问题恰是0/1 背包问题的特例。
  - 从而由划分问题是NPC问题推得0/1背包判定问题是NPC问题。
  - 证毕。

计算复杂性与NP完全问题

- 8.7 NP难问题
  - 定义：对于任意$\Pi' \in NP$，都有$\Pi' \propto \Pi$，称Π是NP-难的。
  - NP-难的问题不会比NP中的任何问题容易。
  - NP难问题不要求是NP类问题。
  - P、NP、NP完全、NP难之间的关系？
    - $P \subseteq NP$，P=NP?；NP完全是NP中最难的，即NP完全是NP中的NP难问题，$NP\text{-}C \subset NP\text{-}H$；NP难不比任何NP容易。
  - 例：旅行商问题是NP难问题
    - 证：旅行商问题不是NP的。因为对其解的任一猜想，要检验它是否是最优的，需要同所有其它的环游比较，这样的环游会有指数多个，因而不可能在多项式时间内完成。

NP难问题

- 用图的Hamilton回路问题(NPC问题)证明旅行商问题是NP难的：
- 已知无向图G=(V,E), |V|=n,构造其对应的旅行商问题如下：

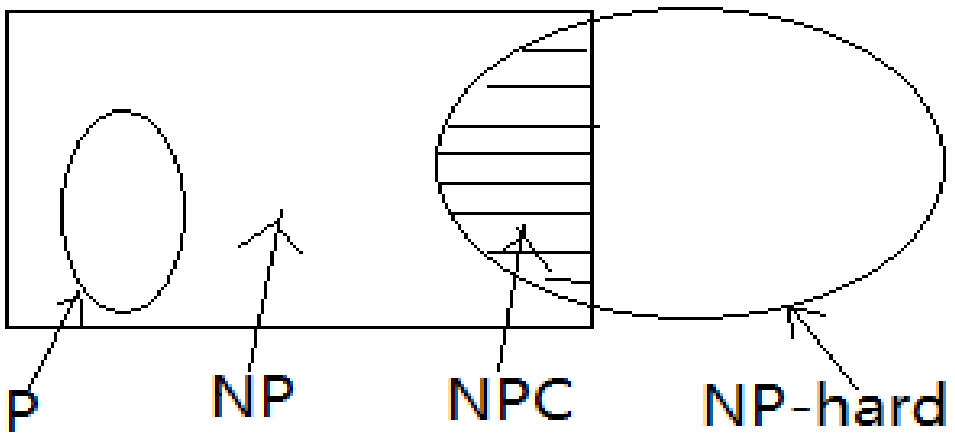
$$d_{ij}=\begin{cases}1, & \text{if } (v_i,v_j)\in E\\ 2, & \text{otherwise}\end{cases}$$

- 这一变换可以在多项式时间内完成，而且，图G有Hamilton回路的充要条件是上述构建的旅行商问题有解，且其解对应的路程长度为n。得Hamilton回路问题$\propto$ 旅行商问题。
- 所以，旅行商问题是NP困难的。得证。

- 0/1背包问题也是NP-难的。（习题八、7）

计算复杂性与NP完全问题

- 8.8 总结
  - 假设P≠NP，未被证实，大家基本接受。则如图所示：
    - P⊂NP，NPC⊂NP-hard，NP与NP-hard的公共部分是NPC。
    - 在NP中，除P和NPC，还有一部分问题的复杂性是未知的。



第九章 概率算法

- 9.1概率算法基本概念
  - 何为概率算法
    - **确定性算法**

      每一计算步骤都是确定的， 有穷的步骤，明确的答案。
    - **概率算法**
      - 当算法在执行过程中面临一个选择时，可随机地选择下一步操作。有穷的步骤，答案可能不确定。
      - 由于随机选择比最优选择省时，概率算法可以在很大程度上降低算法的复杂度，从而用于求解难的问题。
      - 基本特征：对所求问题的同一实例，用同一概率算法求解两次可能得到不同的解、或得到不同的效果(如终止时间)。

概率算法基本概念

- 概率算法的分类
  - **数值概率算法**
    - 常用于数值问题的求解。这类算法所得到的往往是问题的近似解。近似解的精度随着 时间的增加而不断增加。在很多情况下，求解精确解不可能或不必要，此法可得相当满意的解。
  - **舍五德（Sherwood）算法**
    - 虽然在某些步骤引入随机选择，但该算法总能求得问题的一个解，且所求得的解总是正确的。
    - 当一个确定性算法在最坏情况下的计算复杂性与其平均情况下的计算复杂性有较大差别时，可在确定性算法中引入随机性将它改造成一个舍五德算法，消除或减少问题的好坏实例间的差别。
    - 精髓：消除最坏情形与特定实例之间的关联性。

概率算法基本概念

- **拉斯维加斯(Las Vegas)算法**
  - 该算法不会得到不正确的解。一旦用拉斯维加斯找到一个解，这个解一定是正确解。
  - 但**有时该算法找不到解**。找到解的概率随它所用时间的增加而提高。对所求解的任一实例，用同一拉斯维加斯算法求解足够多次，可使求解失败概率任意小。
- **蒙特卡罗（Monte Carlo）算法**
  - 蒙特卡罗算法用于求问题的准确解。有些问题近似解没有意义，如“y/n”的判定问题、求一个整数的因子等。
  - 用蒙特卡罗算法总能求得一个解，但这个**解未必是正确的**。求得正确解的概率随它所用的计算时间增加而提高。
  - 一般情况下，无法有效判定所得解是否肯定正确。

概率算法

- 9.3 Sherwood算法
  - 算法思想
    - 确定性算法A，实例x，计算时间$t_A(x)$，输入规模为n的实例x的全体X_n，平均计算时间 $\bar{t}_A(n)=\sum_{x\in X_n} t_A(x)/|X_n|$
    - 显然不能排除存在$x\in X_n$，$t_A(x)>>\bar{t}_A(n)$ 的可能性。如快速排序算法平均时间O(nlogn)，而当输入已“几乎”排好序时，这个时间界就不再成立。原因：x的分布特性。
    - Sherwood算法通过消除算法时间与输入实例间的联系来使所有实例接近平均性能。
    - 概率算法B， 对每个实例x均有 $t_B(x)=\bar{t}_A(n)+s(n)$，平均计算时间 $\bar{t}_B(n)=\sum_{x\in X_n} t_B(x)/|X_n|$，显然 $\bar{t}_B(x)=\bar{t}_A(x)+s(n)$ ，当
    - s(n)与 $\bar{t}_A(n)$ 相比可忽略时，Sherwood算法可获得很好的平均性能。

Sherwood算法

- 选择算法：确定数组A[1..n]的第k小元素
  - 第三章分治算法：
    - 排序法(O(nlogn)
    - 划分法(最坏$O(n^2)$)
    - 改进的划分法(O(n))：选择合适的划分元素，算法复杂。
  - Sherwood选择算法-线性时间选择算法
    - 随机选择划分元素
    - T(n)=O(n)
- 快速排序算法
  - 平均复杂度O(nlogn)，最坏$O(n^2)$。
  - Sherwood快速排序算法：复杂度O(nlogn)。

Sherwood算法

```
proc PartSelect(A, n, k) { //在数组A[1..n]中找第k小元素 t,并将其
      //存放于位置k，即A[k]=t。
   integer n , k, m, r, j,I; static RandomNumber rnd;
    m:=1; r:=n+1; A[n+1]:= +∞；  ;
    loop
       j:= r ;
       i:=m+rnd.random(j-m-1);  //随机选择划分基准
       MyMath.swap(A,m,i);   //将划分元素A[i]调换到首位置
       Partition(m,j);
       case:   k=j : return // 返回j,当前数组的元素A[j]是第j小元素
                k<j : r:=j; // j是新的下标上界
               else : m:=j+1; //j+1是新的下标下界
       end{case}
      end{loop}
   } end{PartSelect}
```

Sherwood算法

- Sherwood选择算法的复杂度：T(n)=O(n)
  - 证明：划分时划分元素是随机选取的，其为第i小元素的概率为1/n。因为Partition中的比较语句所要求的时间是O(n)，所以，存在常数(其实c<1)，使得算法PartSelect的平均时间复杂度$C_A^{\,k}(n)$可以表示为: $C_A^{\,k}(n) \le cn+\frac{1}{n}(\sum_{i=1}^{k-1} C_A^{\,k-i}(n-i)+\sum_{i=k+1}^{n} C_A^{\,k}(i-1))$ ，令R(n)= $\max_k\{C_A^{\,k}(n)\}$
  - 则：$C_A^{\,k}(n)$ ≤cn+ $\frac{1}{n}(\sum_{i=1}^{k-1} R(n-i)+\sum_{i=k+1}^{n} R(i-1))$ ，以下归纳证明R(n)≤ 4cn。
  - 显然R(1)=0 ，R(2)=max{$C_A^{\,k}(2)$ }≤2c+(R(1)+R(1))/2=2c =nc ≤4cn。设该式对1,2…,n-1成立，则对n、任意k，有：
  - $C_A^{\,k}(n)$ ≤cn+(R(n-1)+R(n-2)+…R(n-k+1)+R(k)+R(k+1)+…+R(n-1))/n
    ≤cn+ (n-1+ n-2+…+ n-k+1 + k + k+1 +…+ n-1)*4c/n
    ≤cn+((2n-k)(k-1)/2+(n+k-1)(n-k)/2)*4c/n

Sherwood算法

- Sherwood快速排序算法：复杂度=平均复杂度O(nlogn)
  - **proc QuickSort**(p,q) //将数组A[1..n]中的元素A[p], A[p+1], …, //A[q] 按不降次序排列，并假定A[n+1]是一个确定数，且大于 //A[1..n]中所有的数。划分后j成为划分元素的位置。

```
integer p,q; static RandomNumber rnd;
global n, A[1..n];
if p<q then
 j:=q+1;
i:=p+rnd.random(j-p-1);  //随机选择划分基准
MyMath.swap(A,p,i);   //将划分元素A[i]调换到首位置
 Partition(p,j);
 QuickSort(p,j-1);
 QuickSort(j+1,q);
end{if}
end{QuickSort}
```

概率算法

- 9.4 Las Vegas算法
  - 算法思想
    - Las Vegas算法能显著改进确定性算法的效率，甚至对迄今找不到有效算法的问题，也能得到满意的算法。
    - 但它的随机性决策可能导致算法**找不到**所需的解。
  - 一般模式
    - Bool success = LV(x, y)，x—输入参数；当算法找到一个解时返回 true ，此时y返回问题的解；否则返回 false，此时可对同一实例再次独立调用相同的算法。
    - Void Obstinate(InputType x, OutputType y)

      { //反复调用Las Vegas算法直到找到问题的一个解

      bool success = false

      while (! success) success = LV(x,y) ; }

Las Vegas算法

- 平均计算时间

设 p(x)是对输入对象x用Las Vegas算法获得问题的一个解的概率，s(x)和e(x)分别是算法对于实例x求解成功和求解失败所需的平均时间，t(x)是Obstinate找到实例x的一个解所需的平均时间，则

$$t(x) = \big(p(x)s(x) + (1 - p(x))e(x)\big) / p(x)$$

$$= s(x) + \frac{1 - p(x)}{p(x)} e(x)$$

- 正确的Las Vegas算法

如果存在正数δ使得p(x) ≥ δ则说该拉斯维加斯算法是正确的。因为经过k次调用算法后，失败的概率降低为 $(1-\delta)^k$，当k充分大时，$(1-\delta)^k$ 趋于0，代码段 while(!LV(x,y)) 出现死循环的概率为0。即是说，只要有足够的时间运行上述代码，得到问题解的概率为1。

Las Vegas算法

- n皇后问题
  - 用回溯法解n皇后问题，实际上是系统地搜索整个解空间。它的每个解，每一个皇后的位置无任何规律。搜素也就难以使用优先队列、价值函数剪枝等来减少搜索点。
  - 这些特征，使用随机算法可以设计出高效算法。
  - 在棋盘上相继的各行中随机地放置皇后，并使新放置的皇后与已经放置的皇后互不攻击，直至 n 个皇后被相容地放好，或者没有下一个皇后可放的位置时为止。
  - 显然这样设计出的算法不一定能找到解。但可通过多次调用找到解。

Las Vegas算法：n皇后问题

```
class Queen {
  friend void nQueen(int);
   private:
   bool Place(int k );  //测试皇后 k 置于第x[k]列的合法性
   bool QueenLV(void);  //随机放置 n 个皇后的 Las Vegas 算法
   int n , * x ;   //皇后个数与解向量
  }
bool Queen : : Place( int k ){
  for ( int j = 1 ; j < k ; j + + )
     if (( abs(k-j) = = abs(x[j] – x[k] )) | | (x[j] = = x[k] ))
          return false ;
  return true ;
}
```

Las Vegas算法：n皇后问题

```
bool Queen::QueensLV(int n){ // 随机放置n个皇后的拉斯维加斯算法
 RandomNumber rnd;
 int k = 1; int count = 1; // k下一个皇后; count:合适位置个数
 while((k <= n) && (count > 0)){
  count = 0;
  for(int i = 1; i<=n; ++i){      // 皇后k可以放置的位置放y，计算个数
   x[k] = i;                      //放count
   if(Place(k))
    y[count++] = i;
  }
 if(count > 0)                                  // 皇后 k有可放位置
  x[k++] = y[rnd.Random(count)];  // 随机选一个可放位置
 }
 return (count > 0); // count > 0表示放置位置成功
}
```

Las Vegas算法：n皇后问题

```
void nQueen( int n ){ // 解 n 皇后问题的 Las Vegas 算法
 Queen  X;     X . n = n;
 int *p = new int [n+1] ;
 for ( int i = 0 ; i <= n ; i + + )
   p[i] = 0;
 X . x = p;
 //反复调用放置 n 皇后的 Las Vegas 算法，直至放置成功
 while ( ! X . QueensLV( ) );
 for ( int i = 0 ; i <= n ; i + + )
   cout < < p[i] < < " " ;
   cout < < endl ;
 delete [ ] p ;
}
```

Las Vegas算法：n皇后问题

- 与回溯法结合
  - 上述算法一旦发现无法再放置下一个皇后，就全部重新开始，可将上述随机策略与回溯法结合：先用随机法放若干个，再回溯继续放置后面的皇后，直到成功或失败。
  - 方法：将queesLV()改造为queesLV(int stopVegas)，程序的while循环改为k<=stopVegas，放置stopVegas皇后退出。将n皇后的回溯算法nqueens(n)改为backtrack(n,t)，k赋初值t，while k>0 …改为while k>=t …, 则与随机算法结合的n皇后算法算法如下：
  - Public static void nQueen(int stop){

```
x=new int [n+1];y=new int [n+1];
for (int i=0,i<=n;i++) x[i]=0;
while(!queesLV(stop);  //随机成功放置前stop个皇后为止
backtrack(n,stop+1);      //回溯搜索后面的皇后位置
}
```

概率算法

- 9.5 Monte Carlo算法
  - Monte Carlo算法一般情况下对问题的所有实例都以高概率给出正确解。但无法判断一个具体解是否正确。重复调用正确率提高快。
  - p正确的 Monte Carlo算法

    对问题的任一实例得到正确解的概率不小于p，1/2<p<1，称该Monte Carlo算法为p正确的。且称p-1/2为该算法的优势。
  - 一致的 Monte Carlo算法

    对于同一个实例，该算法不会给出两个不同的正确答案。
  - 对于一致的、p正确Monte Carlo算法，如果重复地运行这种算法，每一 次运行都独立地进行随机选择，就可以使产生不正确答案的概率变得任意小。

Monte Carlo算法

- **引理** 设正实数 ε,δ 满足$\varepsilon+\delta<1/2$ ，$MC(x)$ 是一个一致的$1/2+\varepsilon$正确的Monte Carlo算法，$C_\varepsilon=-2/\log(1-4\varepsilon^2)$。如果调用算法$MC(x)$至少$\lceil C_\varepsilon \log(1/\delta)\rceil$ 次，并返回各次调用出现频率最高的解，那么可以得到解同一问题的一个一致的$1-\delta$ 正确的 Monte Carlo 算法。
- **结论：**不论算法 $MC(x)$ 的优势 $\varepsilon>0$ 有多小，总可以通过反复调用来放大算法的优势，使得最终得到算法具有可接受的错误概率。
- 偏真的 Monte Carlo算法 (类似定义偏假的算法)
  - 对解判定问题 D的Monte Carlo算法 ，当$MC(x)$返回 true 时解总是正确的，仅当返回 false 时有可能产生错误的解。
  - 多次调用一个偏真Monte Carlo算时，只要一次调用返回true，就可以断定相应的解为true。后面例子：调用4次正确率从55%提高到95%; 调用6次，正确率可提高到99%。

Monte Carlo算法

- 主元素问题
  - n元数组T[1..n] 中的元素x 称为主元素，如果$|\{i \mid T[i]=x\}| > n/2$
  - 判定主元素的 Monte Carlo 算法：

```
RandomNumber rnd ;
template < class Type >
bool Majority ( Type * T, int n ){
    int i = rnd . Random ( n ) + 1;
    Type x = T[i] ;  //  随机选取数组中元素
     int k = 0;
     for ( int j = 1; j <= n ; j + + )
         if ( T[j] = = x ) k + + ;
     return ( k > n/2 ) ;  //    k > n/2 时，T 含有主元素
}
```

当数组 T 中含有主元素时，算法返回true则显然结果正确。非主元素的个数 <n/2 ，算法返回 false 的概率<1/2 ，该算法是一个偏真的 ½ 正确算法。

如果T中不含主元素，算法以>1/2概率返回false。

Monte Carlo算法

- 重复 2 次调用算法 Majority, 得到一个偏真3/4正确的Monte Carlo算法：
  - bool Majority2 (Type * T, int n) {
    if (Majority (T , n)) return true ; //概率p
    else return Majority (T , n) ; //概率(1-p)，返回true概率(1-p)p
    } // 返回true的概率P+(1-p)P=1/2+1/2*1/2=3/4
- 算法 Majority2中，重复调用 Majority(T,n)所得到结果是相互独立的。因此k次重复调用均返回 false 的概率小于 2^{-k} 。
  - template < class Type >
    bool MajorityMC (Type * T , int n , double e) {
    int k = ceil (log(1/e)/log(2));
    for (int i = 1; i <= k; i + +)
    if (Majority (T , n)) return true ;
    return false ; }

重复调用 ln(1/e) 次算法 Majority得到错误概率小于e 的偏真Monte Carlo 算法

第十章 近似算法

- 10.1 近似算法的相关概念
  - 近似算法与NP-难问题
    - 近似算法是求解NP-难问题的一种重要途径，在实践中非常有效且实用。
    - 这首先是因为很多实际应用问题是NP-完全的组合优化问题，不能它们难解就不去解决这些问题。既然不能在多项式时间内找到最优解，只好求其次，去找一个满意的可行解。
    - 实际问题的输入通常也是近似的(如测量所得)，常常也不需要一定获得非常精确的解，往往足够好就可以了。
    - 近似算法的两个问题：设计算法和分析算法的近似比。
      - 近似算法通常比较简单，设计思想常来源于直觉或某种物理现象、社会现象，甚至日常生活常识并依托经典算法思想。
      - 分析算法的近似比的关键是估计最优解的值，这是比较困难的。

近似算法的相关概念

- 近似算法的性能
  - 问题Π，实例I，最优值$F^*(I)$，算法A是一个多项式时间算法产生的可行解的值F(I)。
  - 绝对近似算法：算法A称为问题Π的绝对近似算法，如果对问题Π的每个实例I，有 $|F^*(I)-F(I)|\leq k$，其中，k是常数。
  - 相对近似算法：算法A称为问题Π的相对近似算法，如果对问题Π的每个实例I，有$r_A(I)=\max(F(I)/F^*(I),F^*(I)/F(I)\leq\varepsilon(n)$，
其中，ε(n)是I的规模n的函数(一般是多项式)。称ε(n)为A的近似比，A为ε近似算法。
  - 近似格式：A(ε)称为问题Π的近似格式，如果对每个给定的ε>0和实例I，算法A(ε)产生一个满足$|F^*-F|/F^*\leq\varepsilon$ 的可行解。
  - 多项式时间近似格式：对于任意给定的ε>0，计算时间是问题规模的多项式；
  - 完全多项式时间格式：计算时间是问题规模和1/ε的多项式。

近似算法的相关概念

- 绝对近似算法
  - **0/1背包问题的贪心算法不是绝对近似算法。**

    例子：n=2, P=(2, r), W=(1, r), M = r (r>2)。

    $F^*(I)=r$, $F(I)=2$, $|F^*(I)-F(I)|=r-2$
  - **最多程序存储问题存在1-绝对近似的多项式时间算法。**
  - 最多程序存储问题：两个容量为L的磁盘，n个程序需要的存储空间分别是：$l_1, l_2, \ldots, l_n$,求存入磁盘的程序的最大个数。
  - 近似算法：将程序按照所需存储空间由小到大编号，然后逐一向第一个磁盘存储，第一个磁盘不能存储时，再将剩下的程序逐一向第二个磁盘存储，直至第二个磁盘也不能存储为止。
  - 如果考虑将这n个程序向一个容量为2L的磁盘存储，则上述算法会得到最优解，不妨设为k个。
  - 因而，将n个程序向两个容量各为L的磁盘存储时，存储的最大个数不会超过k。即$F^*(I)\leq k$。
  - 按照上述算法假设第一个磁盘存储了k_1个程序，第二个磁盘存储k_2个程序，则$k_1+k_2+1\geq k$，即$F(I)\geq k-1$，于是$|F^*(I)-F(I)|\leq 1$。

近似算法的相关概念

- 求0/1背包问题的绝对近似解是NP难问题
  - 对于0/1背包问题的实例I：M，$\{p_i,w_i\}$, $1\le i\le n$，构造另一个0/1背包问题的实例I'： M，$\{(k+1)p_i,w_i\}$, $1\le i\le n$，显然，I与I'有相同的可行解集，而且$F^*(I')=(k+1)F^*(I)$。
  - 如果存在求0/1背包问题绝对近似解的多项式时间算法A，使得 $|F^*(I) - F(I)| \le k$，则可以调用算法A求问题实例I'，有

$$|F^*(I') - F(I')| \le k \qquad (1)$$

  - 设对应的解为X。在p_i和k均为整数的情况下，F(I')或等于F*(I')或者至多为$F^*(I') -(k+1)$。因而，由(1)式得到：F(I')=F*(I')，即X是问题实例I'的最优解，也即是问题实例I的最优解。这说明0/1背包问题有多项式时间的确定算法求得最优解。由0/1背包问题是NP难问题可知，绝对近似的0/1背包问题也是NP难问题。

近似算法的相关概念

- ε-近似算法(格式)
  - 多机调度问题：有n项独立的作业集A={1,2,…, n}，由m台相同的机器加工处理。作业 a所需要的处理时间为t(a)。多机调度问题要求给出一种调度方案，使所给的n个作业在尽可能短的时间内由m台机器处理完。
  - 贪心的近似算法GMPS：
    - 贪心规则：按输入的顺序分配作业，把每一项作业分配给当前负载最小的机器。
    - 例：有3台机器，8项作业，处理时间依次是3,4,3,6,5,3,8,4。算法分配给3台机器的作业为 {1,4}，{2,6,7}，{3,5,8}，负载分别是3+6=9,4+3+8=15,3+5+4=12，完成时间为15。这不是最优的分配方案，最优方案为：{1,3,4}，{2,5,6}，{7,8}，负载分别为3+3+6=12,4+5+3=12,8+4=12，完成时间为12.

近似算法的相关概念

- GMPS是2-近似算法
  - 证明：下述事实是显然的：$(1)F^*(I) \geq \frac{1}{m}\sum_{a\in A} t(a)$ $(2)F^*(I) \geq \max_{a\in A} t(a)$
  - 设机器M_j的负载最大，即$GMPS(I)=t(M_j)$，其中$t(M_j)$是M_j的负载。又设b是最后分配给M_j的作业。根据算法的贪心规则，b分配给M_j时它的负载最小，故$t(M_j)-t(b) \leq \frac{1}{m}(\sum_{a\in A} t(a)-t(b))$ ，于是

$$F(I)=t(M_j) \leq \frac{1}{m}(\sum_{a\in A} t(a)-t(b))+t(b)=\frac{1}{m}\sum_{a\in A} t(a)+(1-\frac{1}{m})t(b)$$

$$\leq F^*(I)+(1-\frac{1}{m})F^*(I) \quad =(2-1/m)F^*(I)$$

  - GMPS算法复杂度T(n)=O(mn)，m<=n有意义，多项式算法。
  - 下述实例I是一个紧实例：m台机器，m(m-1)+1项作业，前m(m-1)项作业的处理时间为1，最后一项为m。算法把前m(m-1)项作业平均分配给m台机器，最后一项任意分配给某台机器。显然F(I)=2m-1，最优方案：把前m(m-1)项作业平均分给前m-1个机器，每台m项，最后一项分给第m台机器，则F*(I)=m。得
  - F(I)=(2-1/m)F*(I)

近似算法的相关概念

- 改进的贪心近似算法
  - 递降贪心算法DGMPS：将任务按处理时间从大到小排序，然后按GMPS调度。显然仍是多项式算法T(n)=O(nlogn+mn)。
  - 近似比：F(i) ≤(3/2-1/2m)F*(I），即DGPMS是3/2近似算法。
  - 证明：设作业按处理时间降序排列为$a_1,a_2,\ldots,a_n$，仍考虑负载最大的机器M_j和最后分配给M_j的作业a_i，只有两种可能：

    (1)M_j只有一个作业，此时必有i=1；显然这个方案是最优的，F(I)=F*(I)。

    (2)M_j有两个及以上作业，此时必有i≥m+1，从而F*(I) ≥2t(a_{m+1})。
  - 由于t(a_i) ≤t(a_{m+1}) ≤(1/2)F*(I)，则

    F(I) =t(m_j) ≤$\frac{1}{m}(\sum_{1\le k\le n} t(a_k)-t(a_i))$+t($a_i$)= $\frac{1}{m}\sum_{1\le k\le n} t(a_k)+(1-\frac{1}{m})\ t(a_i)$

    ≤F*(I)+(1-1/m)F*(I)/2=(3/2-1/2m)F*(I)　　证毕。

近似算法的相关概念

- ε-近似旅行商问题是NP难问题
  - 证明：假定存在一个多项式时间算法A求解ε-近似旅行商问题。对于Hamilton问题实例I, 其图为G=(V,E)，构造一个新的图$G_1=(V,E_1)$它是一个完全图，各边赋权如下： 若$(u,v)\in E$, $w(u,v)=1$；否则$w(u,v)=k$。其中，$k=\lceil \varepsilon n \rceil$, n是G的顶点数，得到旅行商问题实例I′。
  - 则图G有Hamilton回路当且仅当图G_1有长为n的环游路线。
  - 因为算法A能够求得旅行商问题的ε-近似解，由$F(I')\le \varepsilon F^*(I')$，得$F(I')\le F^*(I')k/n$。
  - 如果$F(I')>n$，则由图G_1的构造，必有$F(I')\ge n-1+k$，结合上式得$F^*(I')\ge n+(n-1)n/k>n$，说明G_1没有长为n的环游,因而G没有Hamilton回路；
  - 如果$F(I')\le n$,则由G_1的构造,必有$F(I')=n$,G_1有长为n环游,因而G有Hamilton回路。说明算法A能够求解Hamilton问题。因为Hamilton回路问题是NPC问题,故ε-近似旅行商问题是NP难问题。

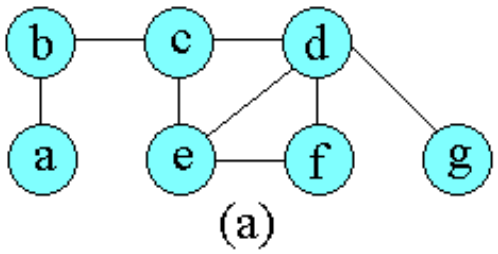
近似算法

- 10.4 顶点覆盖问题的近似算法
  - 问题描述：无向图G=(V,E)的顶点覆盖是它的顶点集V的一个子集V'⊆V，使得若(u,v)是G的一条边，则v∈V'或u∈V'。顶点覆盖V'的大小是它所包含的顶点个数|V'|。求最小覆盖。
  - VertexSet **approxVertexCover**(Graph g)

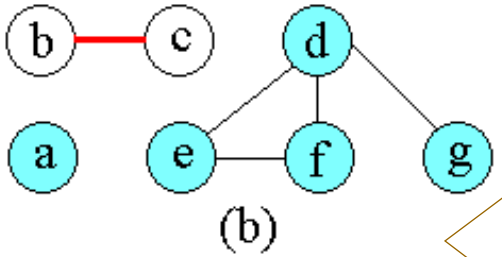
```
{ cset=∅;
  e1=g.e;
  while (e1!= ∅) {
    从e1中任取一条边(u,v);
    cset=cset∪{u,v};
    从e1中删去与u和v相关联的所有边;
  }
 return cset; }
```

Cset用来存储顶点覆盖中的各顶点。初始为空，不断从边集e1中选取一边(u,v)，将边的端点加入cset中，并将e1中已被u和v覆盖的边删去，直至cset已覆盖所有边。即e1为空。

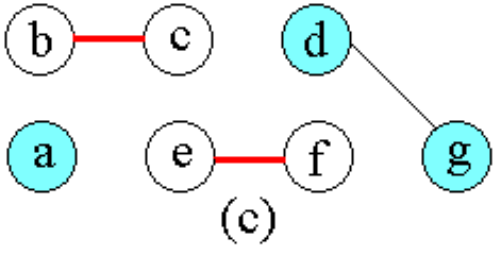
顶点覆盖问题的近似算法



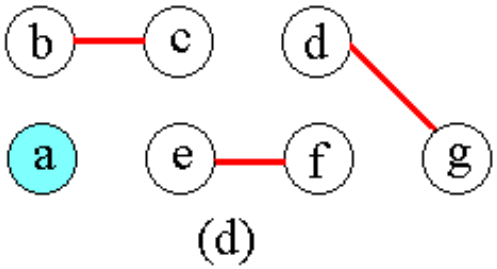
(a)



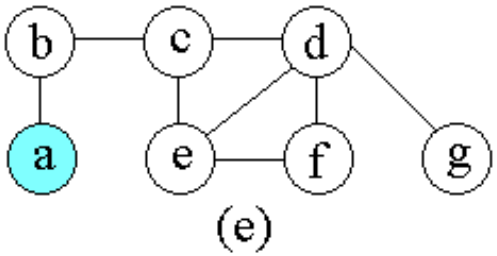
(b)



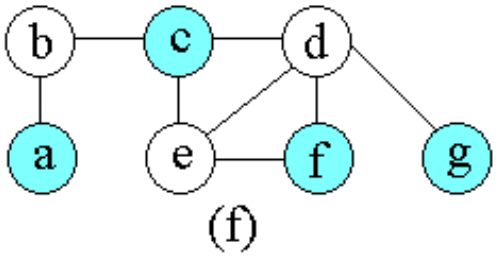
(c)



(d)



(e)



(f)

图(a)～(e)说明了算法的运行过程及结果。(e)表示算法产生的近似最优顶点覆盖cset，它由顶点b,c,d,e,f,g所组成。(f)是图G的一个最小顶点覆盖，它只含有3个顶点：b,d和e。

顶点覆盖问题的近似算法

- 算法**approxVertexCover**的近似比为2
- 证明：
  - 记A为算法选出的边的集合，由A的选取规则可知，A中任何两条边没有公共端点。
  - 算法终止时，|cset|=2|A|。
  - 图的任一顶点覆盖，一定包含A的各条边的至少一个端顶点，设最小顶点覆盖为cset*，则|cset*|>=|A|。
  - 得|cset|<=2|A|<=2|cset*|，ε=|cset|/|cset*|<=2，证毕。

近似算法

- 10.6 0/1背包问题的近似算法
  - 贪心算法G-KK
    - 1.按单位重量价值从大到小排序，设$v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n$;
    - 2.顺序检查每一件物品，只要装的下就将它装入背包，设装入背包的总价值为V;
    - 3.求$v_k=\max\{v_i|i=1,2,..n\}$，若$v_k>V$，则将背包内的物品换成物品k。(算法假设$V_i<=M$，否则可将其排除在外)
  - 例：有4件物品(w,v)为: (3,7),(4,9),(2,2),(5,9), M=6，G-KK给出的解是{(3,7),(2,2)}，总价值9。若把第四件改为(5,10)，则解为{(5,10)}，总价值10。
  - 定理：G-KK是2-近似算法。
    - 证：令OPT(I)是最优解。设t是第一件未装入背包的物品，$OPT(I)<G\text{-}KK(I)+v_t \leq G\text{-}KK(I)+V_{max} \leq 2G\text{-}KK(I)$，证毕。

0/1背包问题的近似算法

- 多项式时间近似方案
  - 把贪心近似G-KK加以改进，可得近似比任意接近1的算法
  - 多项式近似方案PTASε
    - 输入ε>0和实例I。
    - 令m=⌈1/ε⌉。
    - 按单位重量价值排列物品$v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n$。
    - 对每一个t=1,2,..m和t件物品，检查这t件物品的重量之和W，若W ≤M，则接着用G-KK装入剩余物品。
    - 比较得到的所有装法，取最大者作为近似解。
  - 定理：对每一个ε >0和实例I，OPT(I)<(1+ε)PTASε(I)，且PTASε的时间复杂度为$O(n^{1/\varepsilon+2})$。
  - 定理说明，0/1背包问题是多项式时间完全可近似的。

第十一章 启发式算法

- 11.1 启发式(Heuristic Algorithm)算法基本概念
  - 定义
    - 一个基于直观或经验构造的算法，在可接受的花费（计算时间、占用空间）下给出待解决问题实例的一个可行解，该可行解与最优解的偏离程度不一定事先可以预计。
    - 启发式算法是一种技术。这种技术使得在可接受的计算费用内去寻找最好的解，但不一定保证所得解的可行性和最优性，甚至在多数情况下，无法阐述所得到的解同最优解的近似程度。

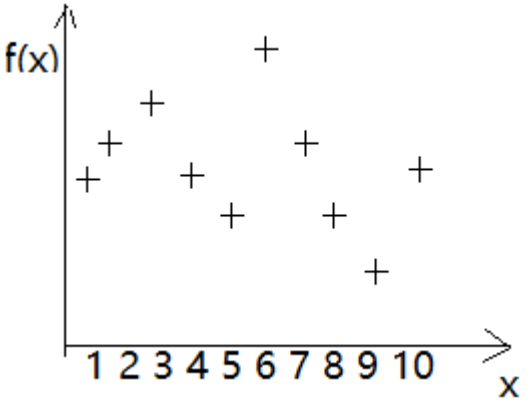
启发式算法基本概念

- 邻域概念
  - 在距离空间中，邻域是以一点为中心的一个圆。光滑函数极值的求解，函数的下降/上升都是在一点的邻域中寻求。
  - 在组合优化问题中，在一点附件搜索另一个下降点是基本方法。但距离概念不适用。
  - 一个组合优化问题可用(D,F,f)表示：D决策变量定义域，F可行解域 $F=\{x|x\in D, g(x)\geq 0\}$，g为约束函数，f为目标函数，求minf(x)，s.t $g(x)\geq 0$，$x\in D$。
  - 邻域的定义：对于组合优化问题(D,F,f)，D上的一个映射：

    N：$S\in D\rightarrow N(S)\in 2^D$，称为一个邻域映射，$2^D$表示D的所有子集组成的集合，N(S)称做S的邻域，$S'\in N(S)$称为S的一个邻居。
  - 例：旅行商问题，解的表示$D=F=\{S=(i_1,i_2,\ldots,i_n)|\ i_1,i_2,\ldots,i_n$是1,2,…n的一个排列}，邻域可定义为2-opt，即S中的两个元素对换，如 S=(1,2,3,4), N(S)={(2,1,3,4),(3,2,1,4),(4,2,3,1),(1,3,2,4),(1,4,3,2),(1,2,4,3)}

启发式算法基本概念

- 例：0/1背包问题，如果每次只允许一个分量变化，则
  $N: x \in D \rightarrow N(x)=\{y \mid \sum_{i=1}^{n} |y_i - x_i| \leq 1\} \in D$，$D=\{(x_1,x_2,..x_n) \in \{0,1\}^n$
- 定义：
  - 若s^*满足$f(s^*) \leq (\geq) f(s)$，对任意$s \in N(s^*) \cap F$，则称s^*为f在F上的局部(local)最小(最大)解。
  - 若$f(s^*) \leq (\geq) f(s)$，对任意$s \in F$，则称s^*为f在F上的全局(global)最小(最大)解。
- 例，一维变量x的定义域为[1,10]中的整数点，目标函数如图，定义$N(x)=\{y \in Z^+ \mid |y-x| \leq 1\}$，则x=9为f的全局最优(小)点，x=5为局部最优，x=4既非局部最大也非局部最小。



启发式算法

- 11.2 禁忌搜索算法
  - 局部搜索算法（建立邻域概念，紧缩搜索范围）的推广，是人工智能在组合优化算法中应用。
  - Glover F. 1986年提出(关于整数规划问题)，1989年的两篇论文奠定了基础。
  - 基本思想：用一个禁忌表记录下已经到达过的局部最优点，在下一次搜索中，禁忌表中的信息不再或有选择地搜索这些点，以此来跳出局部最优点。
  - 最优化问题描述的一般形式：

$$\min f(x)$$

$$s.t.\ g(x) \geq 0$$

$$x \in D$$

禁忌搜索算法

- 局部搜索算法-只按下降转移状态
  - Step1 选定一个初始可行解x^0；记录$x^{cb} = x^0$；令$P = N(x^{cb})$
  - Step2 当$P = \{\}$时，或满足其它停止运算准则时，输出结果，停止计算；否则，从P中选子集S，并从S中选一最优解x^{lb}；
  - 若$f(x^{lb}) < f(x^{cb},)$则$x^{cb} := x^{lb}$，P=$N(x^{lb})$；
    否则$P = P \setminus S$；重复Step2。
- 蒙特卡罗算法-以一定概率接受一个较坏状态
  - Step2/ 当$P = \{\}$时，或满足其它停止运算准则时，输出一个计算结果，停止计算；否则，从P中随机选出解x^{lb}；
  - 若$f(x^{lb}) < f(x^{cb})$，则$x^{cb} := x^{lb}$，否则，根据$f(x^{lb}) - f(x^{cb})$以一定的概率接受x^{lb}($x^{cb} := x^{lb}$)；重复 Step2/。

禁忌搜索算法

- 禁忌搜索算法
  - Step1 选定一个初始可行解x^{cb}；初始化禁忌表$H=\{\}$；
  - Step2 若满足停止规则，停止计算；否则，在x^{cb}的邻域中选出满足禁忌要求的候选集$Can_N(x^{cb})$，并从该候选集中选出一个评价值最佳的解x^{lb}；令$x^{cb}=x^{lb}$。
  - 更新记录H，重复Step2。
- 最优定理
  - 在禁忌搜索算法中，若可行解区域相对邻域候选集$Can_N(x^{cb})$是连通的，且记录H充分大，则一定可以达到全局最优解。(连通的:任给x,y，$x=x_1,x_2,\ldots x_l=y$，$N(x_i)\cap x_{i+1}\neq\phi$)
- 禁忌搜索是局部搜索算法的扩展，主要思想是标记已得到的局部最优解，在进一步的迭代中避开这些局部最优解。

禁忌搜索算法

- 技术问题
  - 解的变化
    - 简单变化：如ABCDE →ACBDE
    - 向量分量变化：如解向量中B、C分量对换，引起解变化：

      ABCDE →ACBDE,ABDCE →ACDBE,ACBED →ABCED

      又如0/1背包问题，解的变化允许一个向量变化则解变化：

      $(x_1,x_2,\ldots x_j,\ldots x_n) \rightarrow (x_1,x_2,..y_j,..x_n)$，$y_j=1-x_j$, j=1,2,..n
    - 目标值变化：如$f(x)=x^2$，目标值从1 →4，则解变化：

      -1 →-2，1 →-2，-1 →2，1 →2
  - 禁忌对象的选取
    - 禁忌对象可以选取上述引起解变化的一种对象。
    - 禁忌范围小搜索范围大，禁忌大搜索小。

禁忌搜索算法

- 禁忌长度*t*：短了陷局部优，大了计算时间长
  - t为常数，如t=10，t=$\sqrt{n}$ ，n为邻居个数。
  - $t \in [t_{min}, t_{max}]$， t_{min}, t_{max}根据规模n或邻域大小T确定。依赖问题、实验和设计者经验。如当函数值下降较大时，可能谷较深，欲跳出局部最优，希望被禁的长度较大。
  - t_{min}, t_{max}动态选取。
- 候选集确定
  - 从邻域中选择若干目标值或评价值最佳的邻居。
  - 随机选取部分邻居。
- 评价函数
  - 基于目标函数
  - 替代评价：当目标值计算复杂时。

禁忌搜索算法

- 特赦规则
  - 特赦：当候选集中对象全部被禁或某一对象解禁则目标值下降明显时，解禁禁忌对象。
  - 特赦规则：
    - 基于评价值-目标值可大幅下降
    - 基于最小错误-候选集全部被禁时
    - 基于影响力-有些对象变化对目标值影响很大(可能变好或变坏)。如0/1背包问题，当包中无法装入新物品时，特赦体积大的分量。
- 记忆频率信息-一定记忆目前最优解
  - 一个最好的目标值出现频率高，提示可能无法得到更好解。
  - 加强禁忌搜索效率，如增加反复出现的对象的禁忌长度。
- 终止规则：确定步、频率控制、目标值变化、目标值偏离度

禁忌搜索算法

- 例1：图的划分判定问题-(也可构造找最小k划分禁忌算法)
  - 无向图G=(V,E)，V={1,2,…n}，若$V_i \subset V, V=\cup_{1..k} v_i$,v_i中任何顶点无边相连，则称$(v_1,v_2,\ldots v_k)$为G的一个k划分。
  - 问题：给定图G和整数k，是否存在k划分？
  - 令$f(v_1,v_2,\ldots,v_k)=|E(v_1)|+|E(v_2)|+\ldots+|E(v_k)|$，$E(v_i)$是$v_i$ 中顶点直接相连的边集。则解$(v_1,v_2,..,v_k)$是G的k划分的充要条件为f()=0。用禁忌搜索求f()最小的解。找到0解输出y，否则N- 蒙特卡罗算法。
  - 解的表示：s=[1..n]，$s[i]\in\{1,2,\ldots,k\}$表示顶点$i\in v_{s[i]}$。
  - 邻域构造：一个解分量的变化构成：$s\rightarrow s^*$:s[i]=s[j]，即顶点i从$v_{s[i]}$挪到$v_{s[j]}$。N(s)共有n(k-1)个邻居。
  - 邻居的选择：随机选择n/2个分量随机变化：n/2个候选解择优。
  - 禁忌对象：$s[j]\rightarrow s[i]$，即禁忌还原到原状态。
  - 特赦规则：若s*是当前最优解，当一个受禁的邻居s满足f(s)<f(s*)-1时特赦受禁变化。(基于评价值特赦)

禁忌搜索算法

- 参考文献：Hertz A, de Werra D. The tabu search metaheuristic: how we use it. Annals of Mathematics and Artificial Intelligence, 1990,1: 111-121.
- 本文设计的禁忌搜索算法求最小k-划分。第一步给定常数k对f()优化计算；第二减小k，重复第一步；从满足f()=0的划分中选择最小k。
- 文献中随机产生的实例参数：G：1000个顶点，边集的密度为50%(大约是$C^2_{1000}/2$条边)。禁忌长度=7，候选解个数为|V|/2=n/2。
- 计算结果：k=87。
- 概率分析的理论结果：k=85。

其它例子：参考书6，图节点着色、车间作业调度问题。

启发式算法

- 11.3 模拟退火算法
  - 模拟退火算法
    - 局部搜索算法的扩展，以一定的概率选择邻域中费用值大的状态(求最小)。理论上是一个全局最优算法。
    - Metropolis N. 1953年提出，Kirkpatrick 1983年成功地应用于组合优化问题。
  - 模型
    - 固体冷却的物理过程中，温度高时，分子活动范围较大，温度逐渐降低时，分子活动范围越来越小。
    - 一种金属在加热到一定温度后，所有分子在状态空间 D中自由运动。随着温度的下降，这些分子渐渐停留在不同的状态。
    - 统计力学研究表明，在温度 T，分子停留在状态 r 满足波兹曼(Boltsmann)分布。

模拟退火算法

- 波兹曼(Boltsmann)分布：

$$\Pr\{\overline{E} = E(r)\} = \frac{1}{Z(T)}\exp\left(-\frac{E(r)}{k_B T}\right)，\quad Z(\mathrm{T}) = \sum_{s\in D}\exp(-\frac{E(s)}{\mathrm{k_B}T})$$

  - 其中，E(r) 为状态的能量，k_B>0为波兹曼常数，$\overline{E}$为分子能量的一个随机变量，Z(T)为概率分布的标准化因子，D状态空间。
  - 上式表明，在同一温度，分子停留在能量小状态的概率比停留在能量大状态的概率大。
  - 当温度很高时，每个状态的概率分布基本相同，接近1/|D|。
- 例1，简化概率分布：$p(x) = \frac{1}{q(t)}\exp\left(-\frac{x}{t}\right)$，$q(t) = \sum_{x=1}^{4}\exp(-\frac{x}{t})$

x=1,2,3,4，考察3个温度点 t=20,5,0.5概率的分布变化:

	x=1	x=2	x=3	x=4
t = 20	0.269	0.256	0.243	0.232
t = 5	0.329	0.269	0.221	0.181
t = 0.5	0.865	0.117	0.016	0.002

模拟退火算法

- 例1显示的退火过程具有的性质：
  - 分子停留在能量小状态的概率比停留在能量大状态的概率要大。
  - 在温度很高时，分子每个状态的概率基本相同；
  - 在温度较低时，分子具有最低状态的概率非常大，接近1。
- 优化问题类比金属状态变化建模

优化问题	金属退火问题
解	状态
最优解	能量最低的状态
费用函数	能量

- 模拟退火算法搜索过程模拟退火过程：温度高时，各种能态均匀分布，温度降低，最低能态概率增大。

模拟退火算法

- 模拟退火算法
  - step1.任选一个初始解x_0；$x_i=x_0$；k:=0；$t_0=t_{max}$(初始温度)；
  - step2. 若在该温度达到内循环停止条件，则转3；
    否则，从邻域$N(x_i)$中随机选择一x_j，计算$\Delta f_{ij}=f(x_j)-f(x_i)$；
    若$\Delta f_{ij}\le 0$，则$x_i=x_j$，否则，若$\exp(-\Delta f_{ij}/t_k)>random(0,1)$时，
    $x_i:=x_j$，重复step2。
  - Step3. $t_{k+1}:=d(t_k)$；k:=k+1；若满足停止条件，终止计算；否则，回到step2。
- 解释
  - Step2 是在同一温度t_k时，在一些状态随机搜索，随机性满足波兹曼分布。
  - Step3将温度下降，再回step2时，接受$\Delta f_{ij}>0$的概率变小。

模拟退火算法

- 模拟退火算法实现的技术问题
  - 按理论要求达到平稳分布来应用模拟退火算法是不可能的，因为：
    - 时齐算法要求在每个温度迭代无穷步以达到平稳分布；
    - 非时齐算法要求温度下降的迭代步数是指数次的。
  - 实际应用中，不去研究问题的结构与收敛性，直接将算法套用于实际问题，在可接受的时间里得到满意解即可。
  - 实现的技术问题，有的与理论有联系，但有一定差距；理论结果是指导性的。技术问题主要是个实践和经验问题。

模拟退火算法

- 解的形式和邻域的构造
  - 问题的编码、邻域的选择方式，两者相互影响。
  - 如背包问题，用$(x_1,x_2,..x_n)$，$x_i \in \{0,1\}$，邻居可构造为 $N(s')=\{s|s是0,1码且|s-s'|\le k, k>1是整数\}$。
  - 问题：无法保证每个邻居都是可行解(容量约束)。解决：罚函数。但罚函数会使搜索空间扩大。
- 温度参数的控制
  - 起点温度的选取：起始温度t_0应保证平稳分布中每一状态的概率相等，即$\exp(-\Delta f_{ij}/t_0)\approx 1$。可取$t_0=K\delta$， K充分大的数，$\delta=\max\{f(j)|j\in D\} - \min\{f(j)|j\in D\}$，取K=10,20,100…等试验值。
  - 温度下降方法：非时齐算法按收敛要求下降(可加速)。时齐算法：(1)$t_{k+1}=\alpha t_k, k\ge 0, 0<\alpha<1$。(2)$t_k=(K-k)t_0/K$,K为温度总下降次数。(3)$t_{k+1}=t_k(1+rt_k/U)^{-1}$, U是$f(i)-f(i_{opt})$的上界，r是充分小的正数。

模拟退火算法

- 每一温度的迭代长度：
  - 固定长度：步数选取按与邻域大小相关规则确定。
  - 接受和拒绝的比率控制法：随温度下降，更多状态被拒绝。为避免过早陷入局优，可令迭代步数增加，以接受(>R)或拒绝(<R)的次数、比率控制迭代终止。
  - 概率控制法：目标以概率1跳出任何局部最优，迭代步数为：$N_i(t) \approx \frac{|N(i)|^2}{(\sum_{\substack{j \in N(i) \\ j \neq i}} \min\{1, \exp(-\frac{f_{\max} - f_{\min}}{t})\})^2}$，随t增长太快。对迭代步数有指导意义。
- 算法的终止规则
  - (1)零度法：给定小ε，$t_k \leq \varepsilon$时终止。(2)循环总数控制法。(3)基于不改进规则的控制法。(4)接受概率控制法：除局部最优解外，其余状态被接受概率 $<x_f$时，停止。(5)邻域控制法：f_0和f_1为局部最优解和次优解，N 邻域大小，当 $\exp(-\frac{f_1 - f_0}{t}) < \frac{1}{N}$ 时停止。

模拟退火算法

- 例：旅行商问题的模拟退火算法
  - 解的编码：path=[i_1,i_2,…,i_n]，为1,2…n的一个排列。

```
struct unit {    //一个解，nCities城市数
double length; //路径长度; double length_table[nCities][nCities];
int path[nCities] } ; //路径
```

  - 邻域结构：2-OPT交换法生成邻居。

```
void getNewSolution(unit &p){
    int i = rand() % nCities;
    int j = rand() % nCities;
    int temp = p.path[i];
    p.path[i] = p.path[j];
    p.path[j] = temp;
    }
```

计算路径长度

```
void CalCulate_length(unit &p) {
   int j = 0;    p.length = 0;
   for (j = 1; j < nCities; j++) {
      p.length += length_table
        [ p.path[j-1] ][ p.path[j] ];  }
   p.length += length_table
    [ p.path[nCities - 1] ][ p.path[0] ];
}
```

模拟退火算法

- 接受新解(状态)规则

```
bool Accept(unit &bestone, unit &temp, double t)  {
    if (bestone.length > temp.length)    return true;
    else  {
        if ((int)(exp((bestone.length-
            temp.length) / t) * 100) > (rand() % 101) )
          return true;
     }
     return false;
   }
```

•部分常量定义：

```
nCities = 99; //城市数量
const double SPEED = 0.98;//退火速度
const int INITIAL_TEMP = 1000;//初始温度
const int L = 100 * nCities;//Markov 链的长度
```

- 其它辅助子过程
  - init_dis(); //初始距离矩阵length_table[nCities][nCities];
  - generate(&temp); //随机产生一个初始解

模拟退火算法

- TSP的模拟退火算法

```
void SA_TSP() {
  srand(time(0)); int i = 0;
  double r = SPEED=0.98;
  double t = INITIAL_TEMP;
  const double t_min = 0.001;
  //choose an initial solution ~
  unit temp; bestone;
  generate(&temp);
  CalCulate_length(temp);
  memcpy(&bestone, &temp,
              sizeof(temp));
```

```
while ( t > t_min )   {
   // L：温度t循环次数，链长
  for (i = 0; i < L; i++) {
    getNewSolution(temp);
    CalCulate_length(temp);
    if(Accept(bestone,temp,t))
      memcpy(&bestone,
           &temp, sizeof(unit));
     else
      memcpy(&temp,
        &bestone, sizeof(unit));
    }
    t *= r; //退火
   }
  return;
}
```

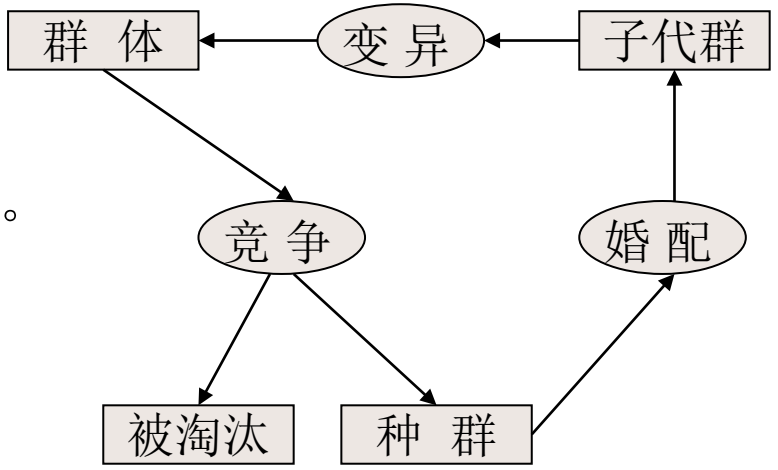
启发式算法

- 11.4 遗传算法(GA)
  - 发展过程
    - 20世纪50~60年代，科学家从生物学中寻求用于计算科学和人工系统的新思想、新方法，发展出适合现实世界复杂系统的计算技术：自然进化系统计算模型和模拟进化算法。主要包括：进化规划(evolutionary programming)、进化策略(evolutionary strategies)和遗传算法(genetic algorithms)。
    - 通过使用类似自然遗传选择和变异的操作算子，不断进化一群候选解，以最终得到问题的最优解或满意解。
    - 1960's, Rechenberg, 进化策略(ES)：面向实参数优化问题求解算法，飞机机翼设计。
    - 1966年，Fogel, Owens, Walsh, 进化规划(EP)，采用有限状态控制器表示候选解，通过随机选择和变异进化状态变换。

遗传算法

- 1976， Holand，遗传算法(GA)自然过程结合到计算机程序之中，Adaptation in Natural and Artificial Systems，采用染色体表示个体，采用自然选择类型算子和遗传类型的交叉算子、变异算子与逆转算子进行群体结构进化。
- 20世纪80年代，GA,ES,EP走向融合。

  基础：自然和社会复杂系统的适应性理论。
- 生物进化循环图：
- 人类进化：
  - 46条、23对同源染色体。
  - 男女结合使对应的23对染色体优胜劣汰产生23对新染色体→下一代。
  - 如性别，由(x(1),x(2))与(x(m),Y)结合产生4种可能2种新性别。



遗传算法

- 生物遗传与遗传算法的对应关系
  - 适者生存　　具有最优目标值的解最可能被留住
  - 个体　　潜在解
  - 染色体　　潜在解的编码（字符串、向量等）
  - 基因　　潜在解每个分量的特征（如分量的值）
  - 适应性　　适应性函数值
  - 群体　　一组潜在解
  - 种群　　根据适应性函数值选择的一组潜在解
  - 交配　　通过交叉选择一组新解的过程
  - 变异　　编码的某个分量发生变化

遗传算法

- 构造遗传算法
  - 选择编码策略，参数集合X和域变成位串空间；
  - 定义适应值函数f(X)；
  - 确定遗传策略：群体规模，选择、交叉 、变异算子，交叉概率，变异概率；
  - 随机初始化群体；
  - 计算群体中个体的适应值；
  - 按照遗传策略选择种群，经交叉、变异形成下一代群体；

遗传算法

- 遗传算法描述
  - Step1 选择问题的一个编码、给出一个有N个染色体的初始群体 pop(1)={$pop_j(1)|j=1,2,..,N$}，t:=1;
  - Step2 对群体pop(1)中的每个染色体$pop_i(t)$计算它的适应性函数 $f_i=fitness(pop_i(t))$;
  - Step3 若停止规则满足，则算法停止；否则，计算概率 $p_i = f_i / \sum_{1 \le j \le N} f_j$ (*)，并以概率分布(*)从pop(t)中随机选出N个染色体(可能有重复)，形成一个种群newpop(t+1)={$pop_j(t)|j=1..N$}。
  - Step4 通过交叉(交叉概率为P_c)得到一个有N个染色体的种群 crosspop(t+1);
  - Step5 以一个较小的概率p，使得染色体的一个基因发生变异，形成种群 mutpop(t+1)；t:=t+1, 一个新的群体诞生pop(t):=mutpop(t);
  - 转到Step2。

遗传算法

- 遗传算法实现的技术问题
  - 编码
    - 常规编码：0/1二进制编码。如前例，0/1背包问题。
      - 问题1：没有考虑能力约束，如0/1背包问题的解$(x_1,x_2,\ldots,x_n)$经变异或与其它解交叉产生的新解可能不满足背包容量约束。解决：使用罚值函数/直接淘汰。
      - 问题2：搜索空间很大。如TSP问题若用解01常规编码，可编为：选择一个边，分量对应的变量为1，否则为0，编码长度为n(n-1)的向量。
      - 编码解的个数$2^{n(n-1)}$，实际可行解(n-1)!，比率$(n-1)!/2^{n(n-1)}$。
    - 非常规编码：同问题密切相关的其它编码
      - 如TSP问题，使用1,2,..n的排列编码。问题：
      - (145623|87) → (14562341)，后代已非可行解。解决：(562378|41) → (56327887)，罚值、交叉新规则。

遗传算法

- 初始群体的规模
  - 根据模板理论结论，理论上的值为m，n较大时规模过大。

$$m = 2^{\delta_s/2}, \delta_s = \frac{(n-1)(1-p_s)}{p_c}$$

  - 实际应用中，取m为编码长度的一个线性倍数：n ≤m ≤2n。
- 适应函数
  - 简单适应函数：fitness=f(x)。可能导致算法收敛到目标值近似的不同染色体，适应函数已难区分这些染色体。
    - 例：求maxf(x)=1+logx,x∈[0.5,1]，$x=0.5+x_1 0.5/2+x_2 0.5/2^2+...$
  - 非线性加速适应函数：M是一个充分大的数，f_{max}当前最优值。

$$fitness(x) = \begin{cases} \dfrac{1}{f_{\max} - f(x)} & , f(x) < f_{\max} \\ M > 0 \quad , & f(x) = f_{\max} \end{cases}$$

x	群体	Fitness	p_x
1/2	0000	0.699	0.221
5/8	0100	0.796	0.252
21/32	0101	0.817	0.259
23/32	0111	0.857	0.271

遗传算法

- 排序适应函数：将染色体按照目标值从小到大排序，直接取分布概率：

$$p(i)=\frac{2i}{m(m+i)}$$

- 线性加速适应函数：fitness(x)=αf(x)+β，

$$s.t.\begin{cases}\alpha\dfrac{\sum_{i=1}^{m}F(x_i)}{m}+\beta=\dfrac{\sum_{i=1}^{m}F(x_i)}{m}\\ \alpha\max\limits_{1\le i\le m}\{F(x_i)\}+\beta=M\dfrac{\sum_{i=1}^{m}F(x_i)}{m}\end{cases}$$

  - 第一个方程表示平均值变换后不变；第二个方程表示当前最优值放大到平均值的M倍。
  - M选取规则：目标值相差较大时，M不要很大，以便遗传的随机性；当群体目标值接近时，逐步扩大 M 。

遗传算法

- 初始群的选取
  - 随机选取(进化代数可能很大)
  - 依据其它算法或经验选择比较好的染色体(有早熟问题)
- 交叉规则
  - 常用方法：双亲双子法。随机选择双亲，将一个随机位后的所有基因交换。
  - 变化交叉：近亲交叉可能造成父子相同，影响搜索范围和收敛速度。本法从头比较两个父染色体，从不同基因位置按常规随机选择交叉位。
  - 多交叉位法：随机选择多个交叉位，以一个交叉位到下一个交叉位基因互换、下一个到下下个不变…交替形成后代。
  - 双亲单子：两个后代随机选取一个或选优。
  - 显性遗传法：对优超基因强制遗传到下一代。（克隆）
  - 单亲遗传：随机两个位置基因交换、交叉位内基因倒排等。

遗传算法

- 非常规码的交叉方法
  - 例：TSP问题整数排列编码方案。
  - 常规交叉法：随机选择交叉位，交叉位后基因按异方基因顺序选取不重复基因：

    213|4567 ⟶ 213|4657

    431|2657 ⟶ 431|2567
  - 不变位法：随机生成同维不变位向量，则对应1的位不变，对应0的位按上述方法从异体选取。
- 种群选取和交叉后群体的确定
  - New-pop(t+1)可以小于pop(t)，如择优选一部分。
  - 不完全替代：用最优的L个子代替换pop(t)中最差的L个个体。
  - 交叉概率不一定为1。

遗传算法

- 终止规则
  - 最大遗传代数；
  - 给定问题的一个下界计算方法，当进化中达到要求的偏差时，算法即终止；
  - 监控进化进程，当算法再进化已无法改进解的性能时即停止算法。如用在线或离线评价函数值是否上升监控进化效果。
- 评价遗传算法的方法
  - 当前最好解方法：可比较其它算法、与问题下界比等。
  - 在线(on-line)比较法：计算 $v^{on-line}(T)=\frac{1}{T}\sum_{t=1}^{T}v(t)$ 观察变化趋势。其中，T当前染色体总数，v(t)为第t个染色体目标值。
  - 离线(off-line)比较法：计算 $v^{off-line}(T)=\frac{1}{T}\sum_{t=1}^{T}v^{*}(t)$ 观察变化趋势，其中，T为至此遗传代数，v*(t)是第t代前所得最优值。