通用解题法

5.1 算法框架

应用回溯法解问题时，首先应明确定义问题的解空间。问题的解空间应至少包含问题的一个(最优)解。

✓问题的解向量：回溯法希望一个问题的解能够表示成一个n元式$(x_1,x_2,\ldots,x_n)$的形式。

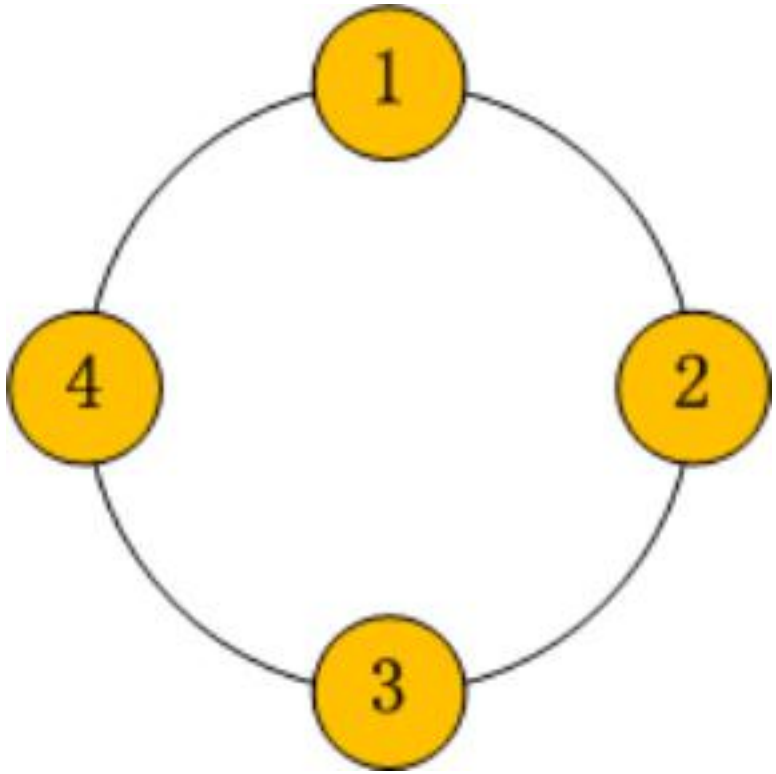
✓显约束：对分量x_i的取值限定。

✓解空间：对于问题的一个实例，解向量满足显式约束

注意：同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）。

素数环问题

1. **问题：** 把整数{1,2,…,20}填写到一个环中,要求每个整数只填写一次,并且相邻的两个整数之和是一个素数。



2. 想法：这个素数环有20个位置，每个位置可以填写的整数为1~20，共20种可能，可以对每个位置从1开始进行试探，约束条件满足：

① 与已经填写到素数环中的整数不重复；

② 与前面相邻的整数之和是一个素数；

③ 最后一个填写到素数环中的整数与第一个填写的整数之和是一个素数。

在填写第k个位置时，满足上述条件，则继续填写k+1位置；如果1-20都无法填写到第k个位置，则取消第k个位置的填写，回溯到第k-1个位置。

算法—— 素数环问题

```
void PrimeCircle(int  n)
{
    int i,k;
    for(i=0;i<n;i++)  a[i]=0;
    a[0]=1;k=1;
    while (k>=1)  {
       a[k]=a[k]+1;
       while  (a[k]<=n)    //查找下一位置的整数
          if (check(k)==1) break;
          else a[k]=a[k]+1;
       if  (a[k]<n&&k==n-1)    //找到可行解
         {for (i=0;i<n;i++)  cout<<a[i]<<“”;
         return ;}
       if  (a[k]<n&&k<n-1)   k=k+1;    //填写下一位置
        else a[k--]=0;    //回溯
    }
}
```

Check(int k)

判断位置k的填写是否满足约束条件：是否重复；相邻数之和是否素数；第一个元素和最后一个元素之和是否素数。

数组a[n]表示素数环

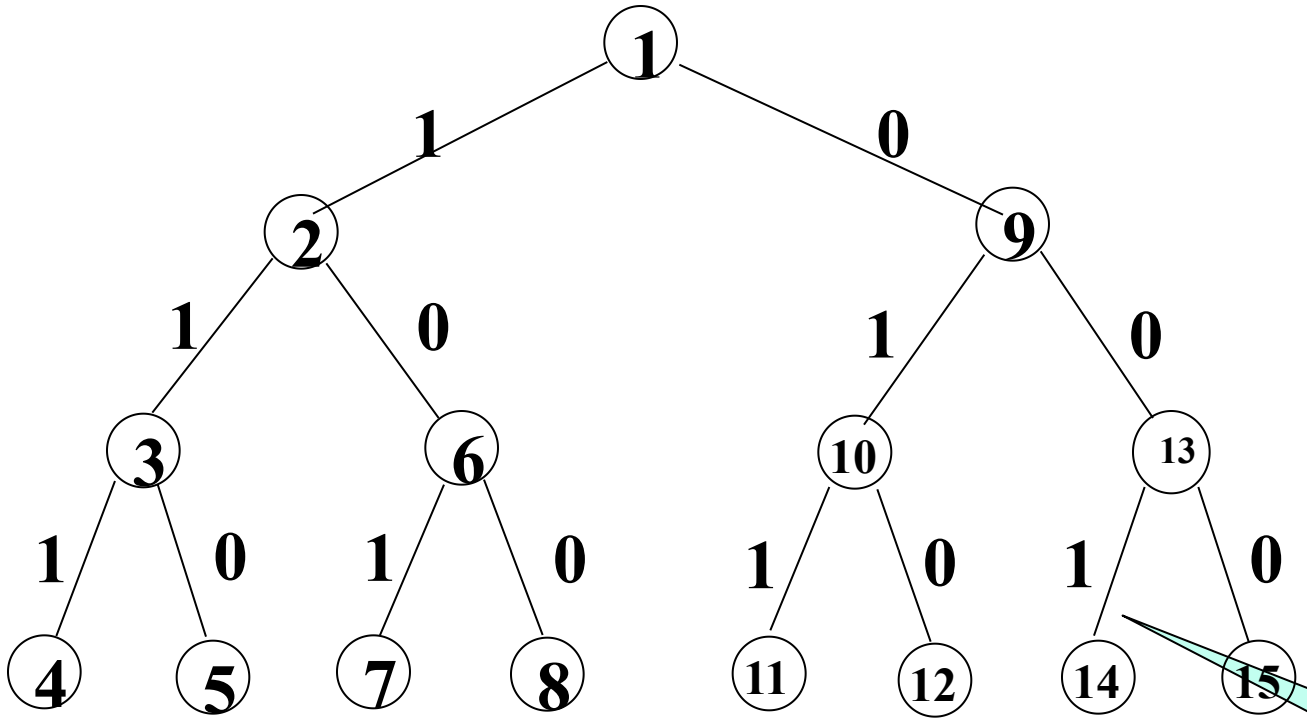
1、问题的解空间

有限离散问题总可以用穷举法求得问题的全部解.

例如　0-1背包问题

设有n个物体和一个背包,物体i的重量为w_i价值为v_i背包的载荷为C, 若将物体$i(1 \le i \le n,)$装入背包,则有价值为v_i .目标是找到一个方案,使得能放入背包的物体总价值最高。

例如 取N=3 , C=30, w={16, 15, 15}, v={45, 25, 25}问题所有可能的解为(解空间): (0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)



对物品1的选择

对物品2的选择

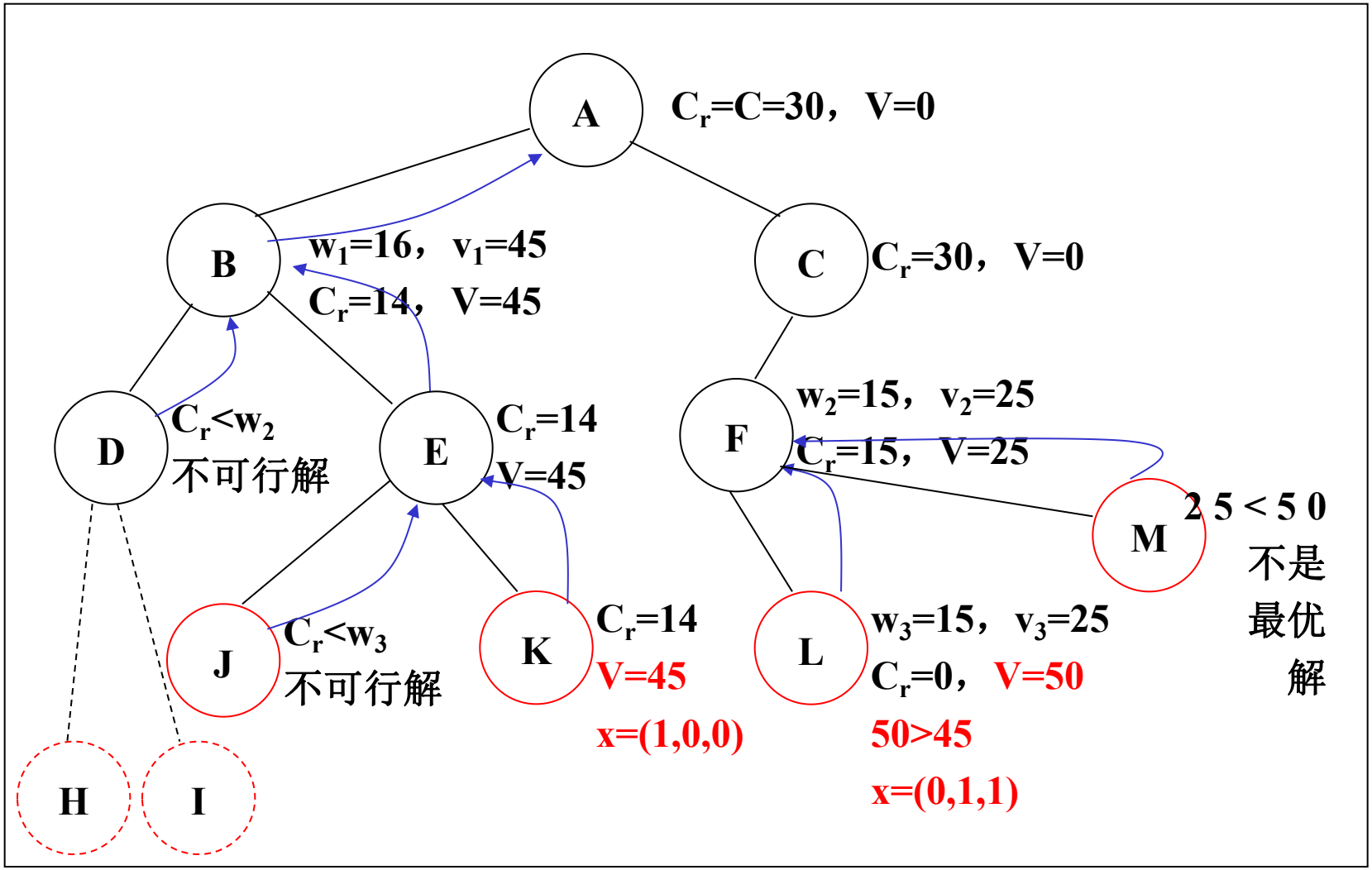
对物品3的选择

时间[illegible]性: $O(2^n)$

可表示为一棵3层的完全正则二叉树

子集树

- n=3, C=30, w={16, 15, 15}, v={45, 25, 25}



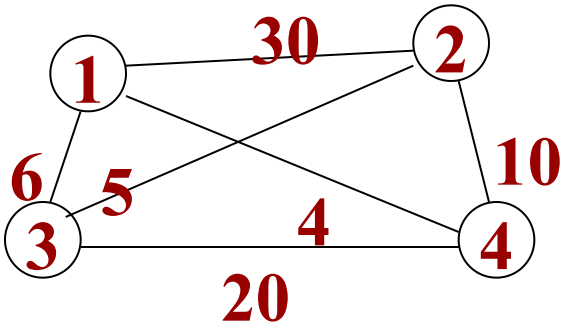
旅行售货员问题

例2 旅行商问题：某售货员要到若干个城市去推销商品。已知各个城市之间的路程（或旅费）。他要选定一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使得总的路程（或总旅费）最小。

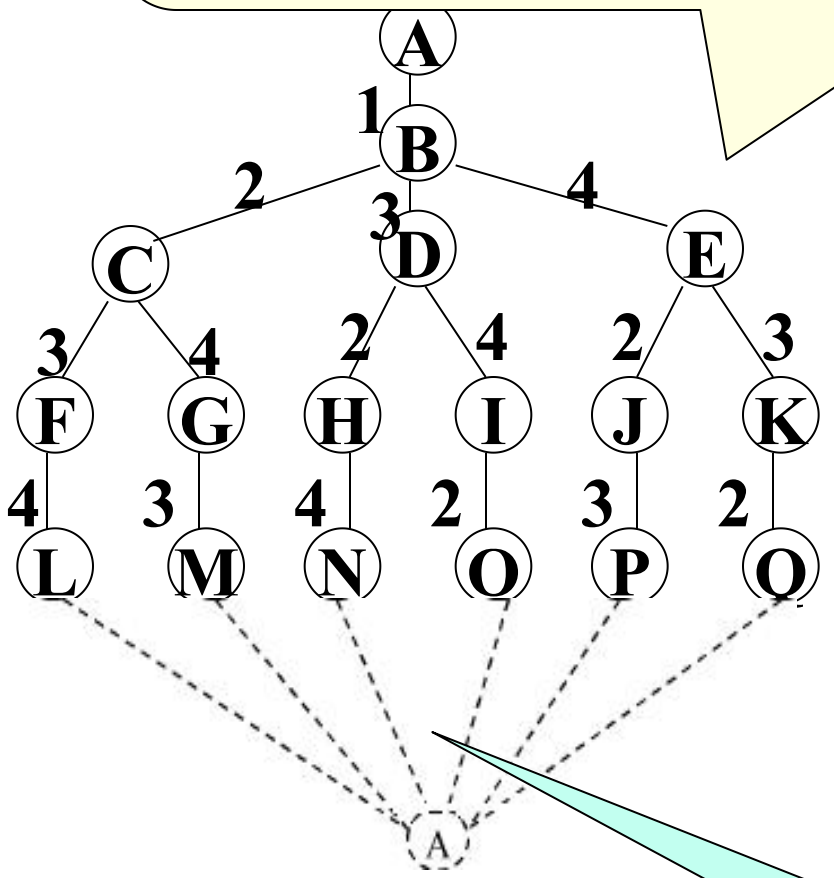
我们用一个带权图G(V, E)来表示，结点代表城市，边表示城市之间的道路。图中各边所带的权即是城市间的路程（或城市间的旅费）

- **该问题是一个NP完全问题， 有(n-1)!条可选路线**

- 旅行售货员问题



解空间组织成一棵树,从树的根结点到任意一叶结点的路径定义了图G的一条周游路线



排列树

- 最优解(1, 3, 2, 4, 1)， 最优值25

2、回溯法的基本思想

① 基本步骤：

(1)针对所给问题，定义问题的解空间；

(2)确定易于搜索的解空间结构；

(3)以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

② 生成问题状态基本方法

- 扩展结点：一个正在产生儿子的结点称为扩展结点
- 活结点：一个自身已生成但其儿子还没有全部生成的节点称做活结点
- 死结点：一个所有儿子已经产生的结点称做死结点
- 回溯法：为了避免生成那些不可能产生最佳解的问题状态，要不断地利用限界函数来处死那些实际上不可能产生所需解的活结点，以减少问题的计算量。具有限界函数的深度优先生成法称为回溯法。

- 深度优先的问题状态生成法：如果对一个扩展结点R，一旦产生了它的一个儿子C，就把C当做新的扩展结点。在完成对子树C（以C为根的子树）的穷尽搜索之后，将R重新变成扩展结点，继续生成R的下一个儿子（如果存在）
- 宽度优先的问题状态生成法：在一个扩展结点变成死结点之前，它一直是扩展结点

③ 常用剪枝函数：

◆用约束函数在扩展结点处剪去不满足约束的子树；

◆用限界函数剪去得不到最优解的子树。

例如:

✓0-1背包问题的回溯法用剪枝函数剪去导致**不可行解**的子树;

✓旅行售货员问题的回溯法中剪去**不含最优解**的子树,如果从根结点到当前扩展结点处的部分周游路线的费用已超过当前找到的最好周游线路费用,则断定该结点的子树不含最优解.

④ 关于复杂性：

- 搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根结点到当前扩展结点的路径。
- 如果解空间树中从根结点到叶结点的最长路径的长度为h(n)，则回溯法所需的计算空间通常为O(h(n))。
- 存储解空间则需要$O(2^{h(n)})$或O(h(n)!)。

3、递归回溯

- 回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

```
backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=f(n,t);i<=g(n,t);i++)
        {
            x[t]=h(i);
            if (constraint(t)&&bound(t)) backtrack(t+1);
        }
}
```

t:递归深度

n:叶结点

记录或输出得到的可行解x

g:终止编号

f:起始编号

表示在扩展节点处的第i个可选值

表示在当前扩展结点处的约束函数和限界函数

4、迭代回溯

- 采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程。

```
void iterativeBacktrack ()
{
    int t=1;
    while (t>0)
    {
        if (f(n,t)<=g(n,t))
            for (int i=f(n,t);i<=g(n,t);i++) {
                x[t]=h(i);
                if (constraint(t)&&bound(t))
                {if (solution(t))  output(x); else t++;}        }
        else t--;        }
}
```

判断在当前扩展节点处是否已得到可行解

5、子集树与排列树

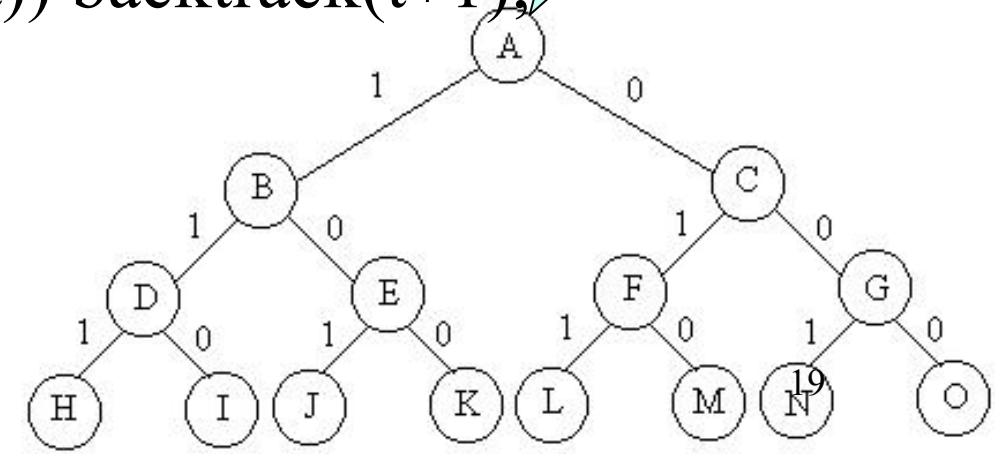
- 假设现在有一列数a[0],a[1], ...a[n-1]
  - ①如果一个问题的解的长度不是固定的，并且解和元素顺序无关，即可以从中选择0个或多个，那么解空间的个数将是为2^n指数级,可以用子集树来表示所有的解
  - ②如果解空间是由n个元素的排列形成，也就是说n个元素的每一个排列都是解空间中的一个元素，那么最后解空间的组织形式是排列树

（1）子集树

遍历子集树需$O(2^n)$计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=0;i<=1;i++) {
        x[t]=i;
        if (constraint(t)&&bound(t)) backtrack(t+1);
      }
}
```

当所给出的问题是从n个元素的集合S中找到满足某种性质的子集时，解空间称为子集树

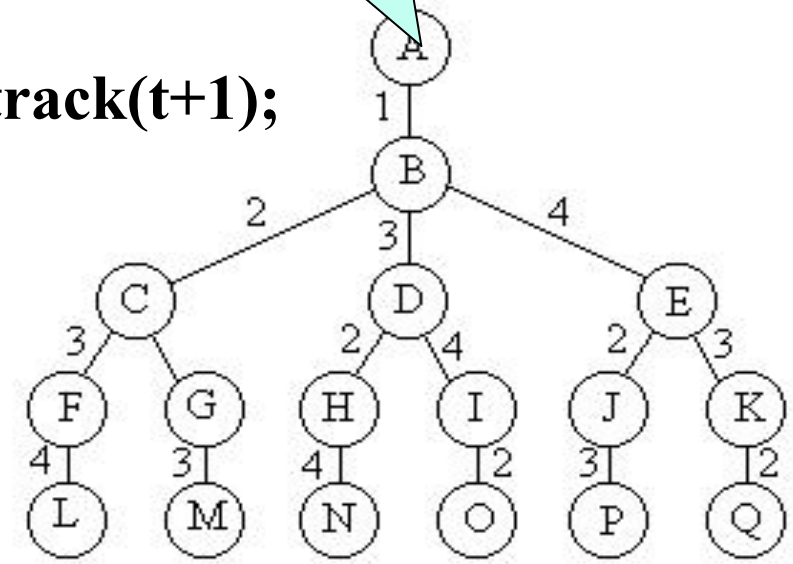


（2）排列树

遍历排列树需O(n!)计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=t;i<=n;i++) {
        swap(x[t],x[i]);
        if (constraint(t)&&bound(t)) backtrack(t+1);
        swap(x[t],x[i]);
      }
}
```

当所给出的问题是确定n个元素满足某种性质的排列时，解空间称为排列树



执行回溯前,先将变量数组初始化为单位排列

5.2 装载问题

1.问题描述: n个集装箱装到2艘载重量分别为c_1, c_2的货轮,其中集装箱i的重量为w_i且 $\sum_{i=1}^{n} w_i \le c_1 + c_2$.

问题要求找到一个合理的装载方案可将这n个货箱装上这2艘轮船。

例如: 当n=3, c1=c2=50,

- 若 w=[10, 40, 50], 有解;
- 若 w=[20, 40, 40], 问题无解.

若装载问题有解，采用如下策略可得一个最优装载方案：

(1)将第一艘轮船尽可能装满；

(2)将剩余的货箱装到第二艘船上。

将第一艘船尽可能装满等价于如下0-1背包问题：

$$\max \sum_{i=1}^{n} w_i x_i, \quad \sum_{i=1}^{n} w_i x_i \le c_1, \qquad x_i \in \{0,1\},\ 1 \le i \le n$$

- 当 $\sum_{i=1}^{n} w_i = c_1 + c_2$ 时，问题等价子集和问题；
- 采用动态规划求解，其时间复杂性为：$O(\min\{C1, 2^n\})$

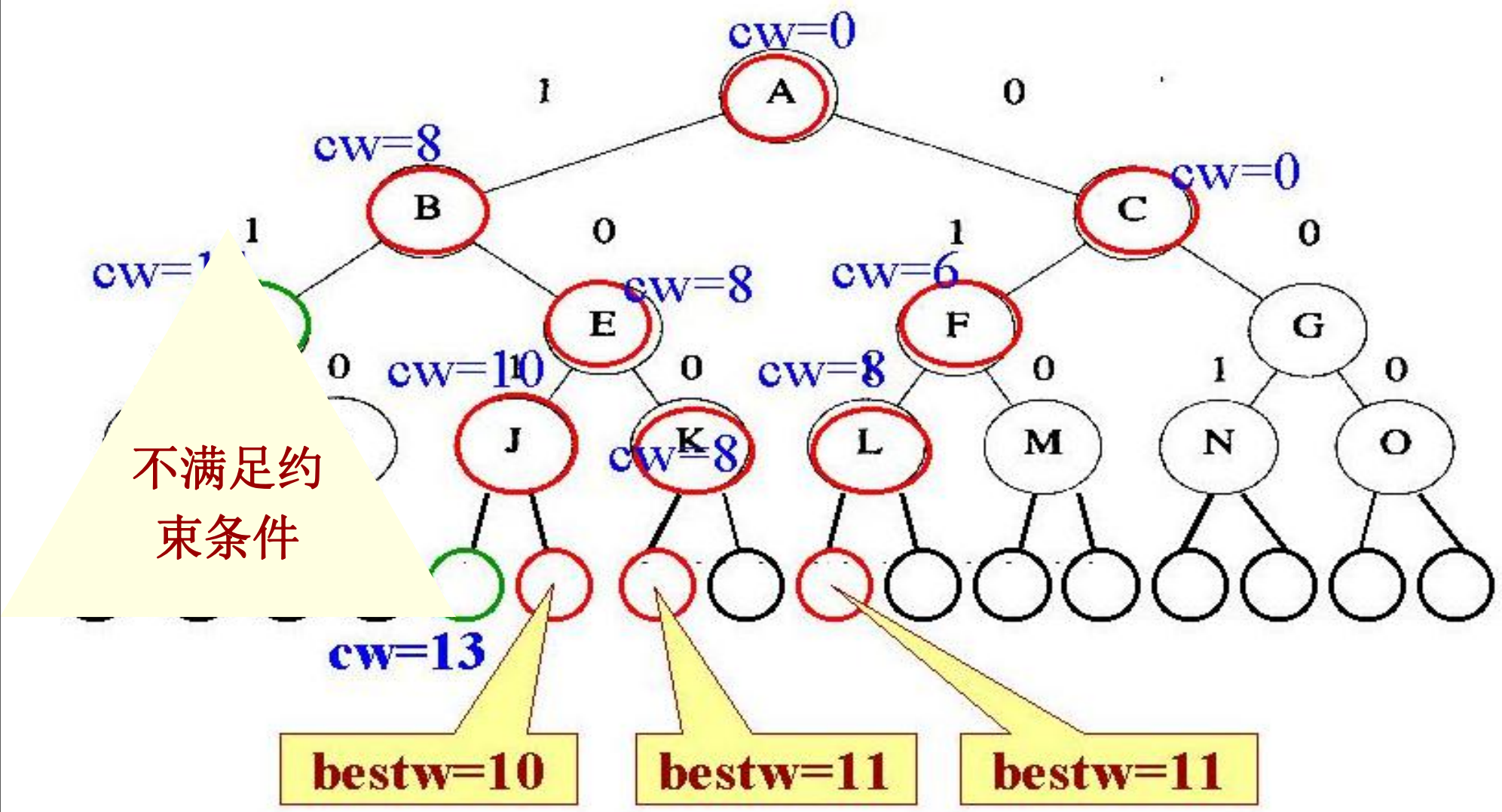
2、回溯算法思路

用子集树表示解空间,则解为n元向量$\{x_1, \dots, x_n\}$, $x_i \in \{0, 1\}$

当前装载量

由于是最优化问题：$cw = \max \sum_{i=1}^{n} w_i x_i$，　当$cw > c_1$,该结点为根结点的子树都不满足约束条件，可利用此条件进一步剪去不含最优解的子树 。

例如 n=4, c1=12, w=[8, 6, 2, 3]. bestw初值=0;



```
template < class Type >
Type Maxloading(type w[], type c, int n)
    loading <Type> X;
   //初始化X
   X. w=w;  //集装箱重量数组
   X. c=c;   //第一艘船载重量
   X. n=n;  //集装箱数
   X. bestw=0; //当前最优载重
   X. cw=0;//当前载重量

   //计算最优载重量
   X.Backtrack(1);
   return X.bestw;   }
```

调用递归函数Backtrack(1)
实现回溯搜索

算法复杂性： $O(2^n)$

◆Backtrack(i)搜索子集树中第i层子树。

◆i≤n,当前扩展节点Z是子集树中的内部结点;该结点有以下两种情况:

➢x[i]=1:进入左儿子, $cw + w_{j+1} \leq c1$,对左子树递归搜索

➢x[i]=0:进入右儿子,总是可行,不检查

◆i>n时,算法搜索至叶结点,其相应装载重量为cw,如果cw>bestw,更新bestw=cw

约束函数

```
template<classType>
void Loading<Type>::Backtrack(int i)
{ / /搜索第i层结点
   if (i>n) {//到达叶结点
        if (cw > bestw)   bestw=cw;
        return;  }

    //搜索子树
   if (cw+w[i]]<=c){     //x[i]=1
        cw += w[i];
        Backtrack (i+1) ;
        cw - = w[i]};
    Backtrack(i+1);     //x[i]=0
}
```

Backtrack(i) 搜索子树集中第i层子树

左子树

回溯过程

右子树

3、加入上界函数

设 bestw: 当前最优载重量,(某个叶节点)

$cw = \sum_{i=1}^{j} w_i x_i$:当前扩展结点的载重量 ;

$r = \sum_{i=j+1}^{n} w_j$: 剩余集装箱的重量;

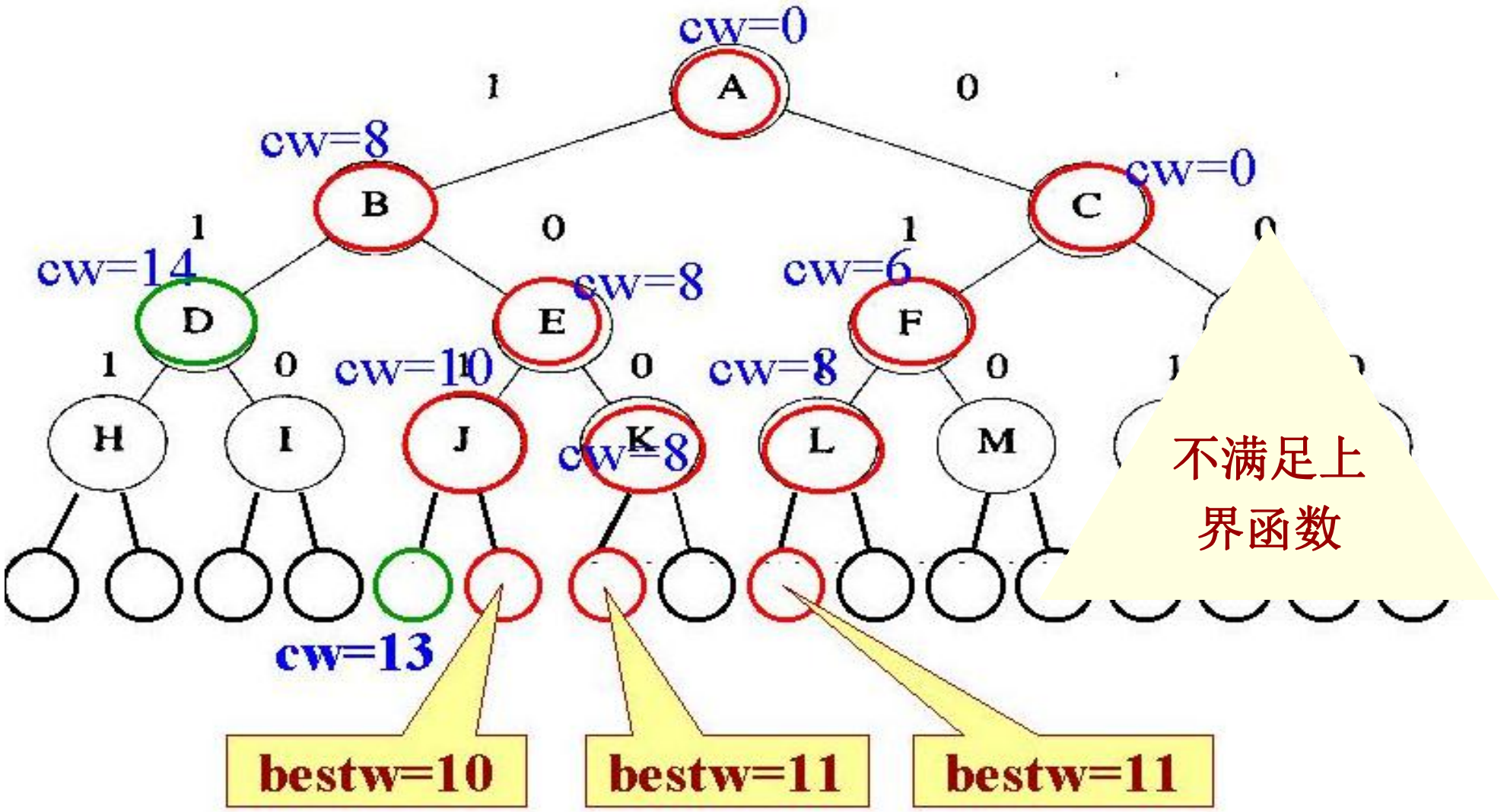
当 cw+r (限界函数)≤bestw 时, 将cw对应的子树剪去。

剪枝条件:

$$\sum_{i=1}^{j} w_i x_i + w_{j+1} > c1$$

$$cw + r \le bestw$$

例如 n=4, c1=12, w=[8, 6, 2, 3]. bestw初值=0;



```
template < class Type >
Type Maxloading(type w[], type c,
            int n,)
 loading <Type> X;
//初始化X
X. w=w;  //集装箱重量数组
X. c=c;   //第一艘船载重量
X. n=n;  //集装箱数
X. bestw=0; //当前最优载重
X. cw=0;//当前载重量
X. r=0;  //剩余集装箱重量
for (int i=1; i<=n; i++)
    X. r +=w[i]
//计算最优载重量
X.Backtrack(1);
return X.bestw;   }
```

```
void Loading<Type>::Backtrack(int i)
{ / /搜索第i层结点
   if (i>n) {//到达叶结点
       bestw=cw;  return;  }
   //搜索子树
   r - = w[i];
   if (cw+w[i]]<=c){     //x[i]=1,搜索左子树
       cw += w[i];   x[i]=1 ;
       Backtrack (i+1) ;
       cw - = w[i]};
    if (cw+r > bestw){   //搜索右子树
      x[i]=0; Backtrack(i+1)};
   r+=w[i] ;
   }
```

引入上界函数,剪去不含最优解的子树,改进算法效率

4、构造最优解

增加两个私有成员：

x：记录从根到当前结点的路径　　int *x;

bestx：记录当前最优解　　int *bestx;

初始化：

X.x=new int [n+1];

X.bestx=bestx;

```
Backtrack(int i){
if (i > n) {// 在叶节点上
        for (int j = 1; j <= n; j++)
            bestx[j] = x[j];//最优解
        bestw = cw;
    return;}
//搜索子树
 if (cw+w[i]]<=c){     //x[i]=1,搜索左子树
        cw += w[i];    x[i]=1 ;
        Backtrack (i+1) ;
        cw - = w[i]};
if (cw+r > bestw){   //搜索右子树
       x[i]=0;
       Backtrack(i+1)};
   r+=w[i]; }
```

5.3 批处理作业调度

[问题描述]给定n个作业的集合J=(1, 2, ... , n)。每一作业i 须先由机器1处理, 再由机器2处理. 设t_{ji}是在机器j上处理作业i的时间, i=1,...,n, j=1, 2. F_{ji}是在 机器j上完成作业i的时间. 所有作业在机器2上完成时间和: $f=\sum F_{2i}$.

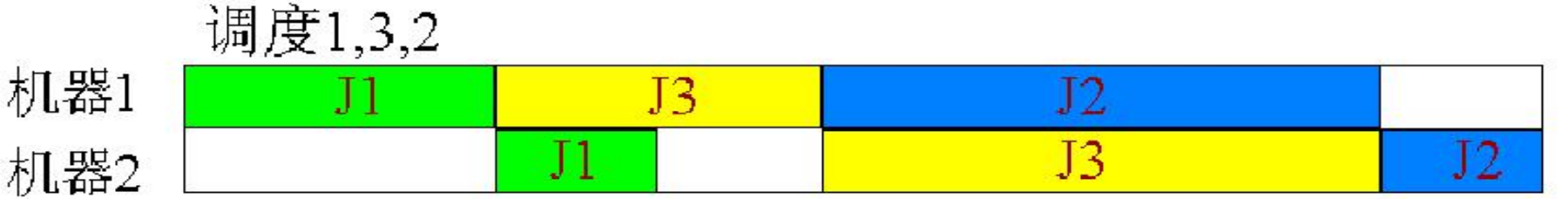
对于给定J, 制定一最佳作业调度方案, 使完成时间和最小.

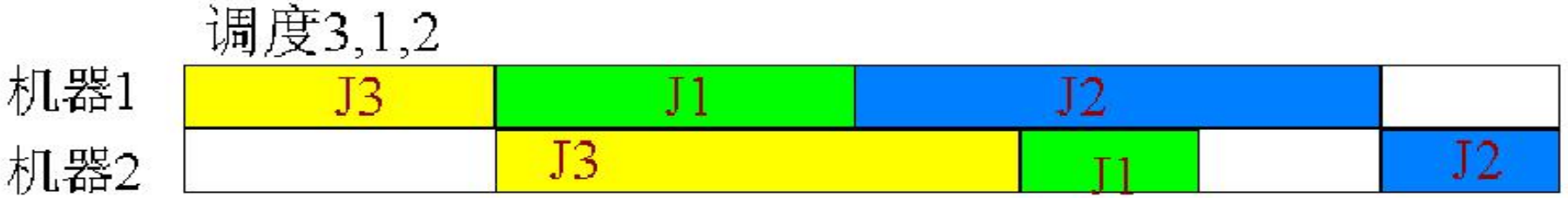
t_{ji}	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3

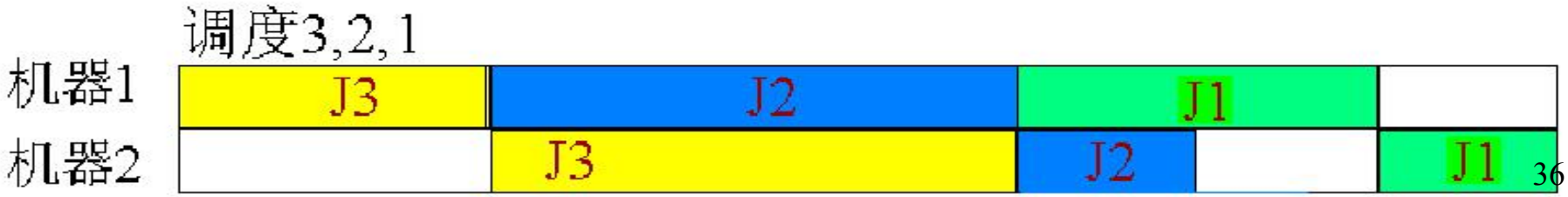
这3个作业的6种可能的调度方案是1,2,3；1,3,2；2,1,3；2,3,1；3,1,2；3,2,1；它们所相应的完成时间和分别是19，18，20，21，19，19。易见，最佳调度方案是1,3,2，其完成时间和为18。

t_{ji}	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3

最佳调度: {1, 3, 2};{ 3,1,2 },
{3, 2, 1 };
完成时间和: 18。







[问题想法]：显然，批处理作业的一个最优调度应使机器1没有空闲时间，且机器2的空闲时间最小。

可以证明，存在一个最优作业调度使得在机器1和机器2上作业以相同次序完成。

- 解空间E=$\{x_1,\ldots,x_n\}$, $x_i \in \{1,..n\}$,
- 约束条件: 当$i \neq j$, $x_i \neq x_j$ (元素不能重复选取)

限界函数: bestf:为当前最小完成时间和

f ：当前作业k的完成时间;

当 f> bestf 时, 将k对应的子树剪去

```
int Flow(int ** M, int n, int bestx[ ] )
{ int ub = 32767;  INT_MAX
   Flowshop X;
   X.x = new int [n+ 1];        //当前调度
   X.f2 = new int[n+ l];        //机器2的完成时间
   X.M = M;                     //各作业所需处理时间
   X.n= n;                      //作业数
   X.bestx = bestx;             //当前最优调度
   X. bestf = ub;               //当前最优调度时间
   X.fl = 0;                    //机器1完成处理时间
   X.f = 0;                     //完成处理时间和
   for(int i = 0;i<= n; i++)
         X.f2[i] = 0,X.x[i]=i;
   X. Backtrack( 1 );
   delete [ ] X. x;
   delete [ ] X. f2;
   return X. bestf ;}
```

算法复杂性： O(n!)

Backtrack中:

当i＞n时，算法搜索至叶结点，得到一个新的作业调度方案，此时算法适合更新当前最优值和相应的当前最佳作业调度。

当i<n时，当前扩展节点位于排列树的第i-1层。此时算法选择下一个要安排的作业，以深度优先的方式递归地对相应子树进行搜索。对于不满足上界约束的节点，则剪去相应的子树。

```
void Flowshop: :Backtrack(int i)
{ if (i>n) {
     for(int j = 1;j <= n; j++)
         bestx[j] = x[j];
     bestf = f;}
  else
     for (int j = i; j < = n; j ++) {
         fl += M[ x[j] ] [1];
         f2[i]= ((f2[i-1]>fl)?f2[i-1]:fl)+M[x[j]][2];
         f+= f2[i];
         if (f <bestf) {
             Swap(x[i], x[j]);
             Backtrack(i + 1);
             Swap(x[ i], x[j] );
           }
         fl - = M[j]][1];
         f- = f2[i] ;}}
```

叶结点

当前扩展结点

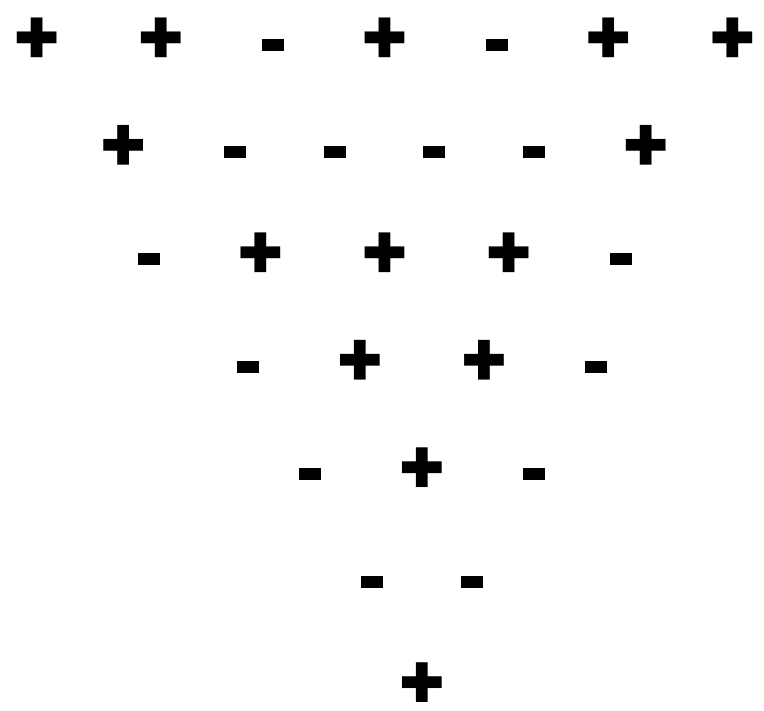
限界函数

5.4 符号三角形问题

- 一、问题描述
- 二、问题分析
- 三、算法描述

1、问题描述

- 右图是由14个“+”和14个“-”组成的符号三角形。2个同号下面都是“+”，2个异号下面都是“-”。
- 在一般情况下，符号三角形的第一行有n个符号。
- 符号三角形问题要求对于给定的n，计算有多少个不同的符号三角形，使其所含的“+”和“-”的个数相同。



2、问题分析

- 定义：用n元组x[1:n]表示符号三角形的第一行。
  - 当x[i]=1,表示符号三角形的第一行的第i个符号为“+”
  - 当x[i]=0,表示符号三角形的第一行的第i个符号为“-”
- 使用回溯法解决符号三角形问题
  - 用一棵完全二叉树表示其解空间.

子树集

三角形中+-号总数为n(n+1)/2

- **可行性约束函数：**

  当前符号三角形所包含的“+”个数与“-”个数均不超过n*(n+1)/4

- **无解的判断：**

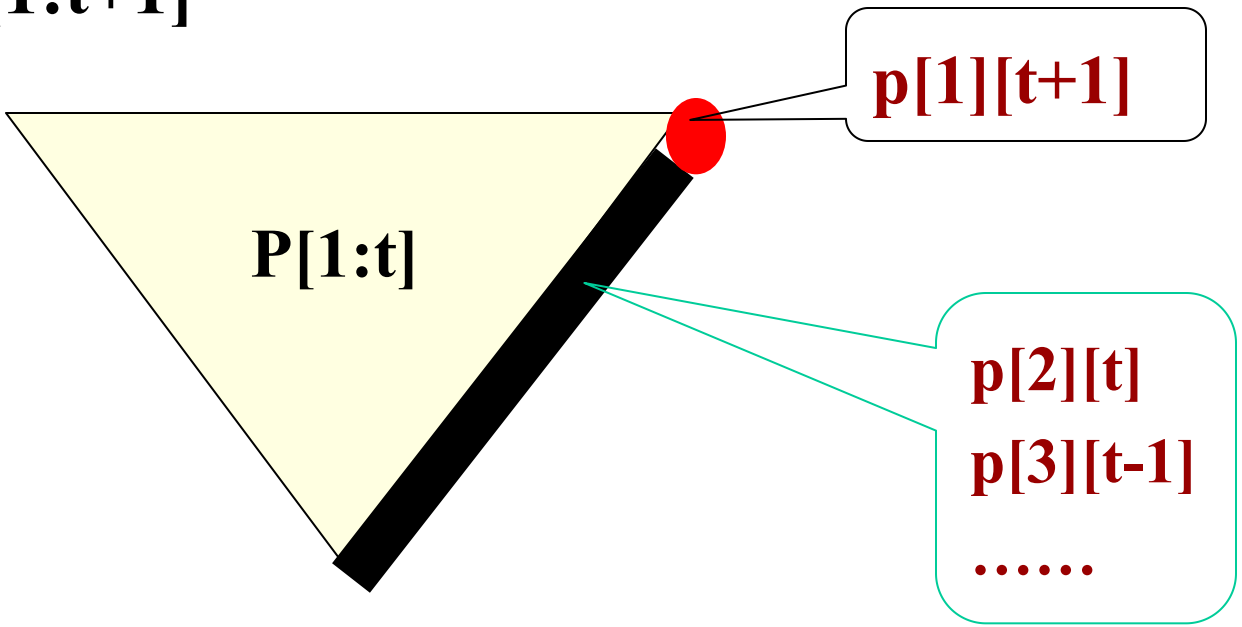
  n*(n+1)/2为奇数

- **复杂度分析：**计算可行性约束需要O(n)时间，在最坏情况下有 $O(2^n)$个结点需要计算可行性约束，故解符号三角形问题的回溯算法所需的计算时间为 $O(n2^n)$。

- 说明：

➢ 符号三角形距阵定义为**p；

➢ 已知深度t的三角形距阵P[1:t]，只需要确定p[1][t+1]的值，就能在已确定的三角形的右边加一条边扩展P[1:t+1]



3、算法描述

```
void Triangle::Backtrack(int t)
{
 if ((count>half)||(t*(t-1)/2-count>half)) return;
   //加或减号个数大于符号总数的一半
 if (t>n)  sum++;     //找到一个符合条件的三角形
 else
    for (int i=0;i<2;i++) {     //i=1 为"+"，i=0为"-"
      p[1][t]=i;
      count+=i;
      for (int j=2;j<=t;j++) {
          p[j][t-j+1]=p[j-1][t-j+1]^p[j-1][t-j+2];
          count+=p[j][t-j+1];       }
    Backtrack(t+1);
    for (int j=2;j<=t;j++)    count-=p[j][t-j+1];
    count-=i;
} }
```

n:第一行符号个数
half:为n(n+1)/4
count:“+”个数
**p:符号三角矩阵
sum:符号三角形数

计算新增的最后一斜列的+和-号个数

5.5 八皇后问题

- 八皇后问题是十九世纪著名的数学家高斯于1850年提出的。
- **问题：**在8×8的棋盘上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上。
- 把八皇后问题扩展到*n*皇后问题，即在*n*×*n*的棋盘上摆放*n*个皇后，使任意两个皇后都不能处于同一行、同一列或同一斜线上。

四皇后问题

- 四皇后问题的解空间树是一个完全4叉树
- 树的根结点表示搜索的初始状态
- 从根结点到第2层结点对应皇后1在棋盘中第1行的可能摆放位置，从第2层结点到第3层结点对应皇后2在棋盘中第2行的可能摆放位置，依此类推。

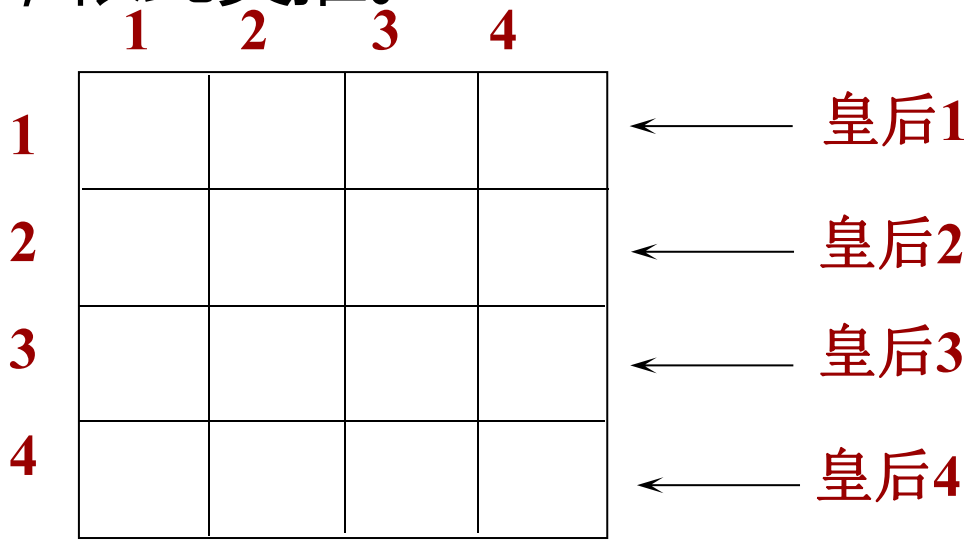
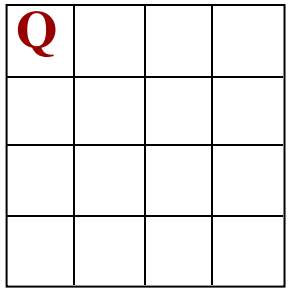
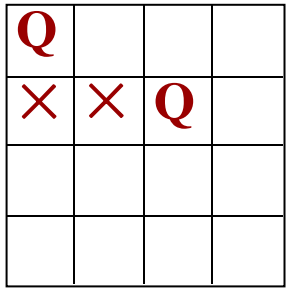


图 四皇后问题

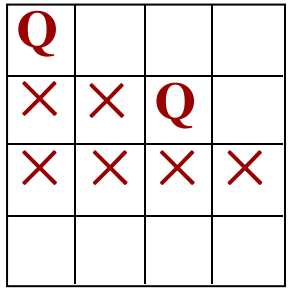
回溯法求解4皇后问题的搜索过程



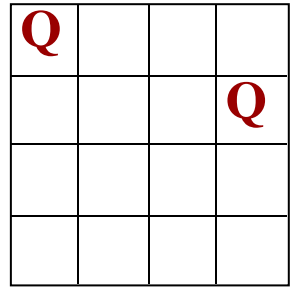
(a)



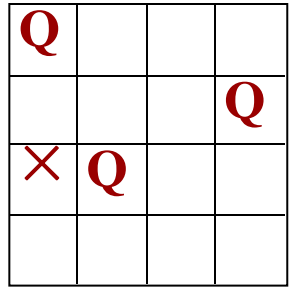
(b)



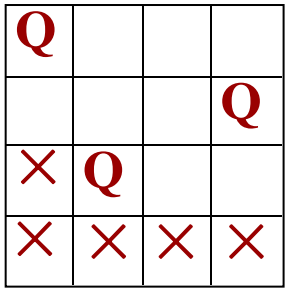
(c)



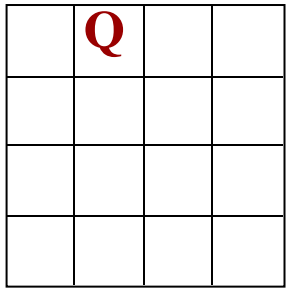
(d)



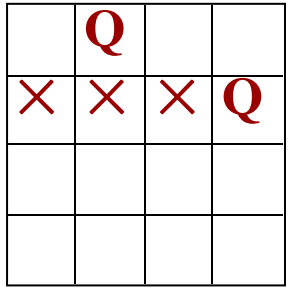
(e)



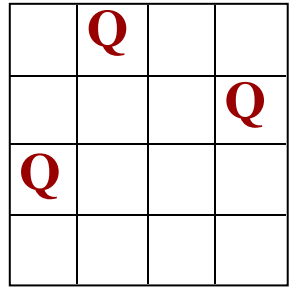
(f)



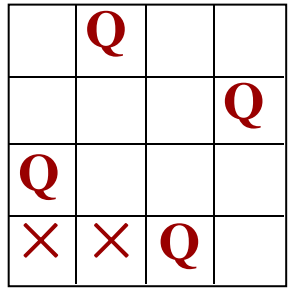
(g)



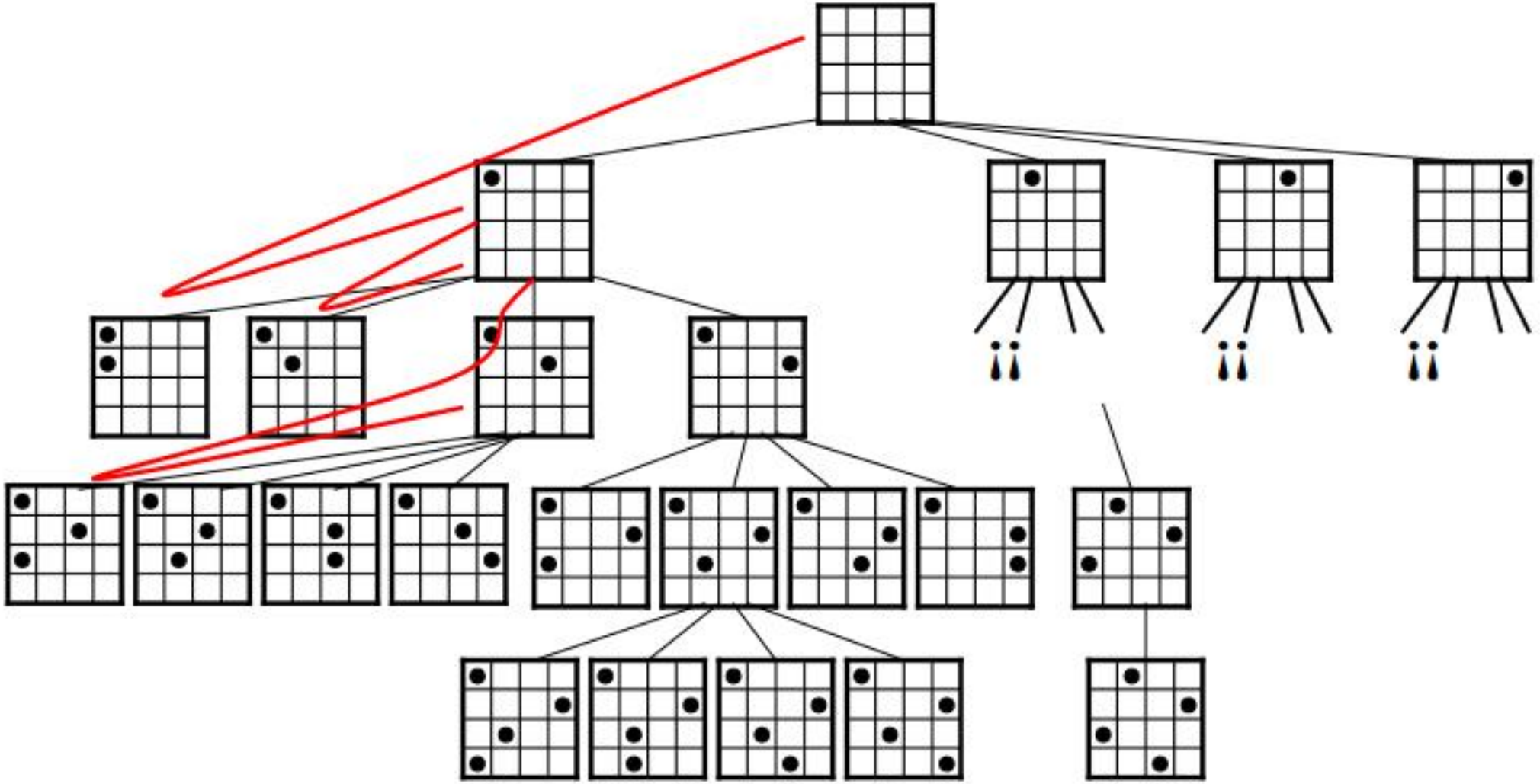
(h)



(i)



(j)



四后问题的状态空间树

➢**想法**:棋盘的每一行上可以而且必须摆放一个皇后，所以，n皇后问题的可能解用一个n元向量$X=(x_1, x_2, \ldots, x_n)$表示，其中，$1\le i\le n$并且$1\le x_i\le n$，即第$i$个皇后放在第$i$行第$x_i$列上。

➢由于两个皇后不能位于同一列上，所以，解向量X必须满足约束条件：

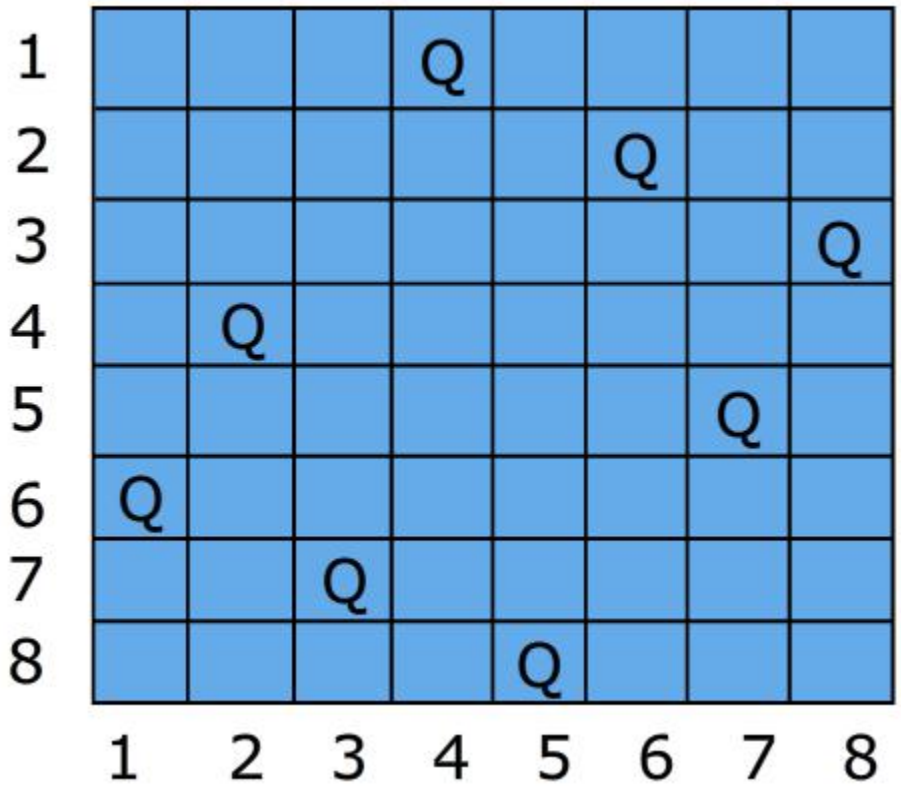
$$x_i\neq x_j$$

皇后i,j不在同一列上

- 若两个皇后摆放的位置分别是(i, x_i)和(j, x_j)，
  - 在棋盘上斜率为-1的斜线上，满足条件$i-j=x_i-x_j$，
  - 在棋盘上斜率为1的斜线上，满足条件$i+j=x_i+x_j$，
- 综合两种情况，由于两个皇后
所以，解向量X必须满足约束条

$$|i-x_i| \neq |j-x_j|$$

皇后i,j不在同一斜



算法：回溯法求解n皇后问题

1. 初始化解向量x[n]={-1};
2. k=1;
3. while (k>=1)
   - 3.1 把皇后摆放在下一列位置，x[k]++;
   - 3.2 从x[k]开始依次考察每一列，如果不发生冲突，则转向步骤3.3；否则x[k]++试探下一列；
   - 3.3 若n个皇后已全部摆放，输出一个解，结束；
   - 3.4 若尚有皇后没摆放，则k++,转向步骤3,摆放下一皇后；
   - 3.5 若x[k]出界，则回溯：x[k]=-1,k--; 转步骤3重新摆放k皇后；
4. 退出循环，无解。

```
void Queue(int n)
  {  k=0;
     while (k>=0)
     {  x[k]++;          //在下一列放置第k个皇后
        while (x[k]<n && Place(k)==1)        //若有冲突,搜索下一列
              x[k]++;
        if (x[k]<n && k= =n-1) {  //得到一个解，输出
            for (i=0; i< n; i++)  cout<<x[i]+1<<“   ”;
            return;          }
        if (x[k]<n && k<n-1)
              k=k+1;       //放置下一个皇后
         else
              x[k--]=-1;    //重置x[k],回溯
      }
      cout<<“无解”<<endl; }
```

```
bool Place(int k)
        //考察皇后k放置在x[k]列是否发生冲突
{
    for (0i<k; i++)
        if (x[k]= =x[i] | | abs(k-i)= =abs(x[k]-x[i]))
            return false;
        return true;
}
```

5.7 最大团问题

[基本概念] 设无向图G=(V, E), U⊂V,

完全子图U :若对任意u, v∈U, 有(u,v) ∈ E.

团(完备子图):完全子图U不包含在G的更大完全子图中。

最大团: G中所含顶点数最多的团。

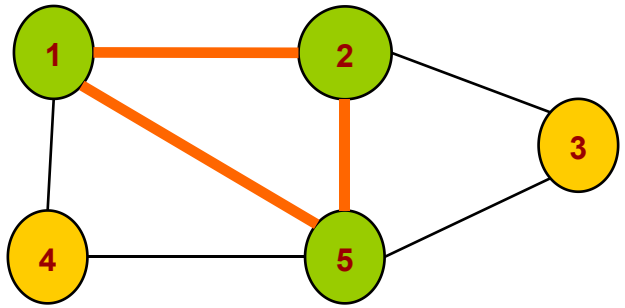
空子图U :如果U⊂V,且对任意u,v∈U,(u,v)∉E。

独立集U: 空子图U不包含在G的更大的空子图中。

最大独立集:G中所含顶点数最多的独立集。

问题描述:在G 中找一个最大团.

例如



最大团:{1, 2, 5}, {1, 4, 5}, :{2, 3, 5}

最大独立集:{2，4}

子集{1，2}是不是团?

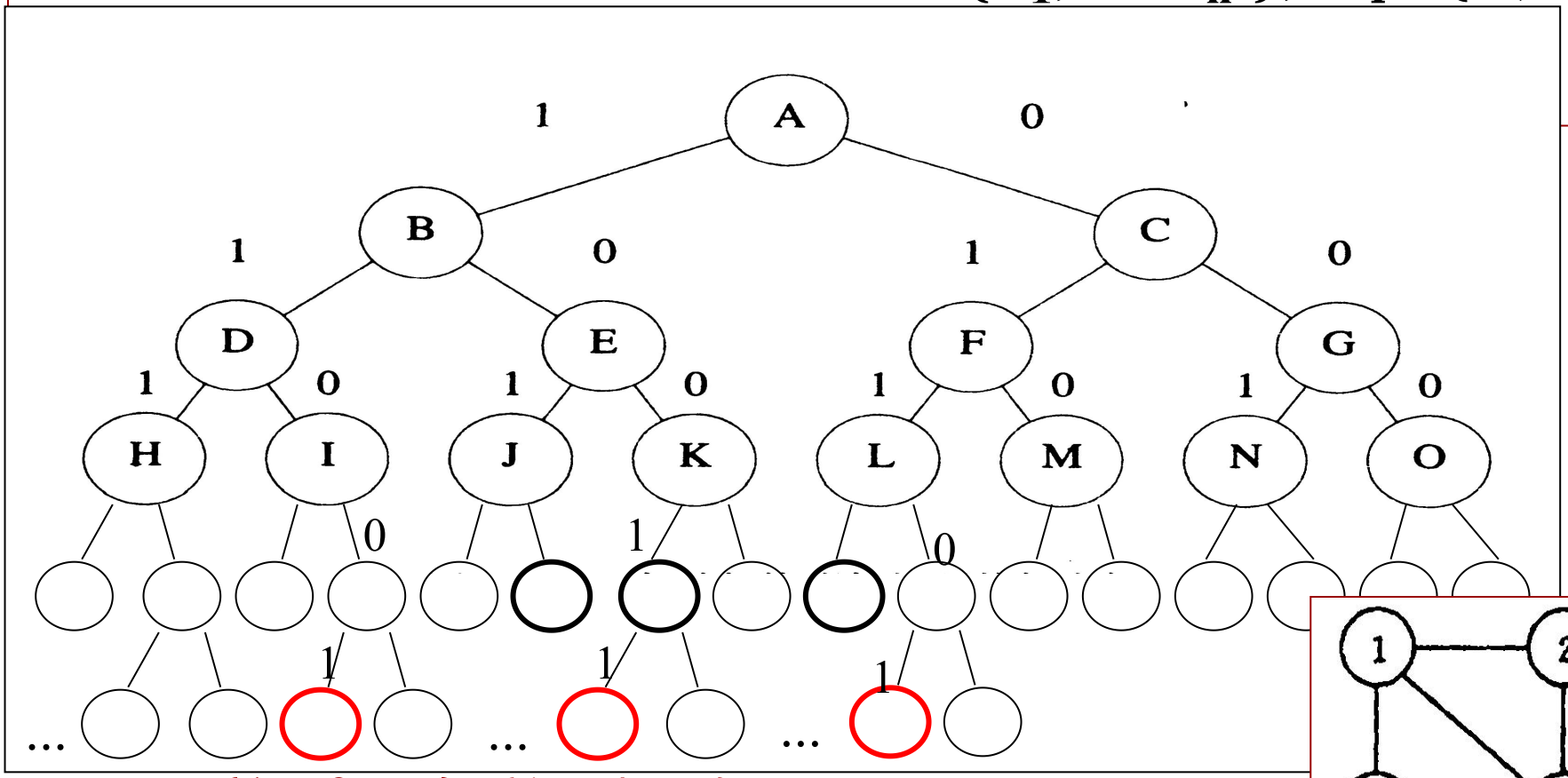
子集{1，2，5}是不是团?

子集{1，2，4，5}是不是团?

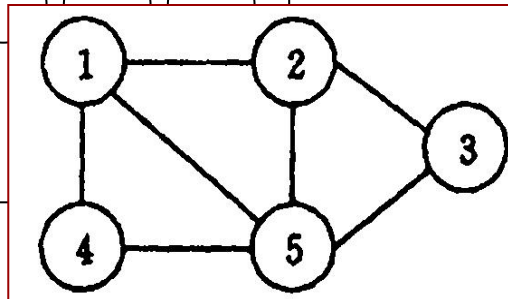
子集{2，4}是不是最大独立集?

算法思路

- 设无向图G=(V, E), |V|=n, 用邻接矩阵a表示图G.
- 问题的解可表示为n元向量$\{x_1, \ldots x_n\}$, $x_i \in \{0,1\}$.



{1，2，5} {1，4，5} {2，3，5}



剪枝原则：

- 约束条件：$\{x_1,x_2,...x_i\}\cup\{x_{i+1}\}$是团.
- 限界函数:设 bestn: 已求出的最大团的尺寸;

cn：当前团的尺寸；

r :剩余结点数目；r=n-i

当 cn+r ≤bestn 时, 将cn对应右子树剪去。

最大团问题的回溯算法

```
int MaxClique(int ** a, int v[i],int n)
  {     Clique Y;
        Y.x = new int [n+l];
        Y.a= a; //图G的邻接矩阵
        Y.n= n; //图G顶点数
        Y.cn = 0; //当前团顶点数
        Y.bestn=0; //当前最大团顶点数
        Y.bestx = v; //当前最优解
        Y. Backtrack( 1 );
        delete [ ] Y. x;
        return Y. bestn;
  }
```

✓设当前扩展节点Z位于解空间树的第i层，

✓在进入左子树之前，必须确认从顶点i到已入选的顶点集中每一个顶点都有边相连。

✓在进入右子树之前，必须确认还有足够多的可选择的顶点使算法有可能在右子树中找到更大的团。

```
void clique::Backtrack(int i)
{    if (i>n) //找到更大团,更新
     {      for (int j=1; j<=n;  j++)      bestx[j] = x[j];
             bestn = cn;        return;        }
  //检查顶点i是否与当前团相连
    int OK = 1;                 //满足加入团的条件
    for(int j =1; j < =i ; j++)
        if (x[j]&&a[i][j]==0)      //i不与j相连，剪左枝
        {   OK = 0;       break; }
    if (OK)    //向左枝递归
    {      x[i] = 1;     //把i加入团
           cn ++;
           Backtrack(i+1);
           x[i]=0;          cn -- ;
    }              //回溯到i，为进入右子树做准备
    if (cn+n- i>bestn)              //剪右枝条件，否则向右枝递归
    {    X[i]=0；Backtrack(i + 1);}
}
```

复杂度分析：回溯算法backtrack所需的计算时间显然为$O(n2^n)$。

5.8 图的m着色问题

问题描述

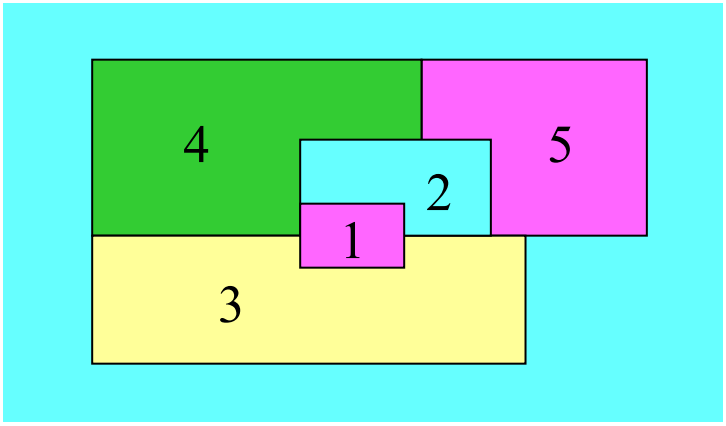
图的m色判定问题: 给定无向连通图G和m种颜色。用这些颜色为图G的各顶点着色. 问是否存在着色方法, 使得G中任2邻接点有不同颜色。

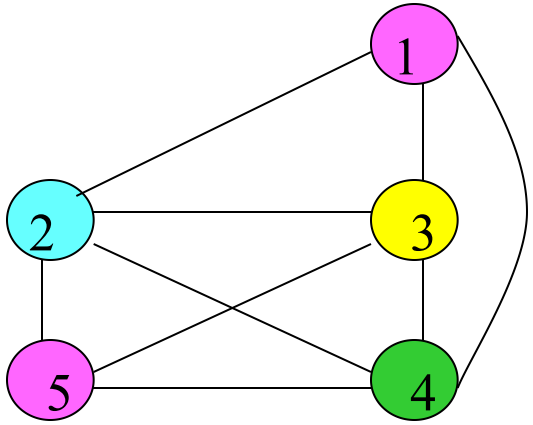
图的m色优化问题:给定无向连通图G,为图G的各顶点着色, 使图中任2邻接点着不同颜色,问最少需要几种颜色。所需的最少颜色的数目m称为该图的**色数**。

可平面图：如果一个图得所有顶点和边都能用某种方式画在平面上，且没有任何两面相交，则称这个图为可平面图。

4色定理：若图G是可平面图,则它的色数不超过4色

4色定理的应用:在一个平面或球面上的任何地图能够只用4种颜色来着色使得相邻的国家在地图上着有不同颜色

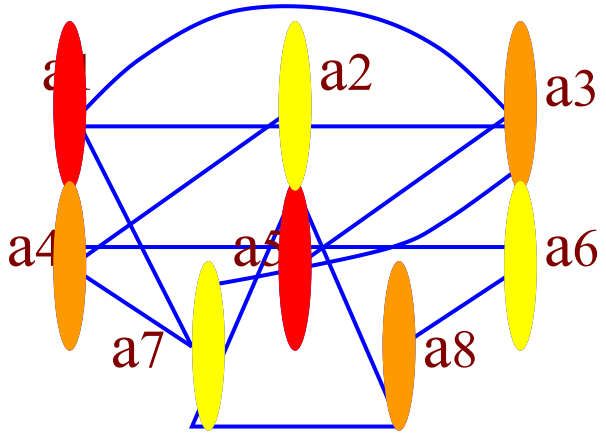




任意图的着色　Welch Powell法

- ✓ a).将G的结点按照度数递减的次序排列.
- ✓ b).用第一种颜色对第一个结点着色,并按照结点排列的次序对与前面着色点不邻接的每一点着以相同颜色.
- ✓ c).用第二种颜色对尚未着色的点重复步骤b).用第三种颜色继续这种作法, 直到所有点着色完为止.

- 排序：a5, a3, a7, a1, a2, a4, a6, a8



- 着第一色：a5, a1,
- 着第二色：a3,a4,a8
- 着第三色：a7, a2, a6

算法思路

◆设图G=(V, E), |V|=n, 颜色数= m, 用邻接矩阵a表示G, 用整数1, 2…m来表示m种不同的颜色。顶点i所着的颜色用x[i]表示。

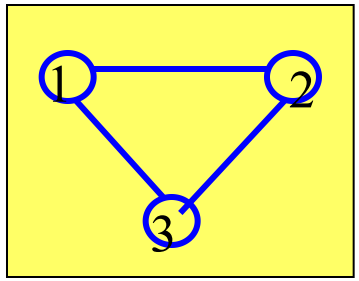
◆问题的解向量可以表示为n元组x=[x[1],...,x[n]].

x[i]∈{1,2,...,m},

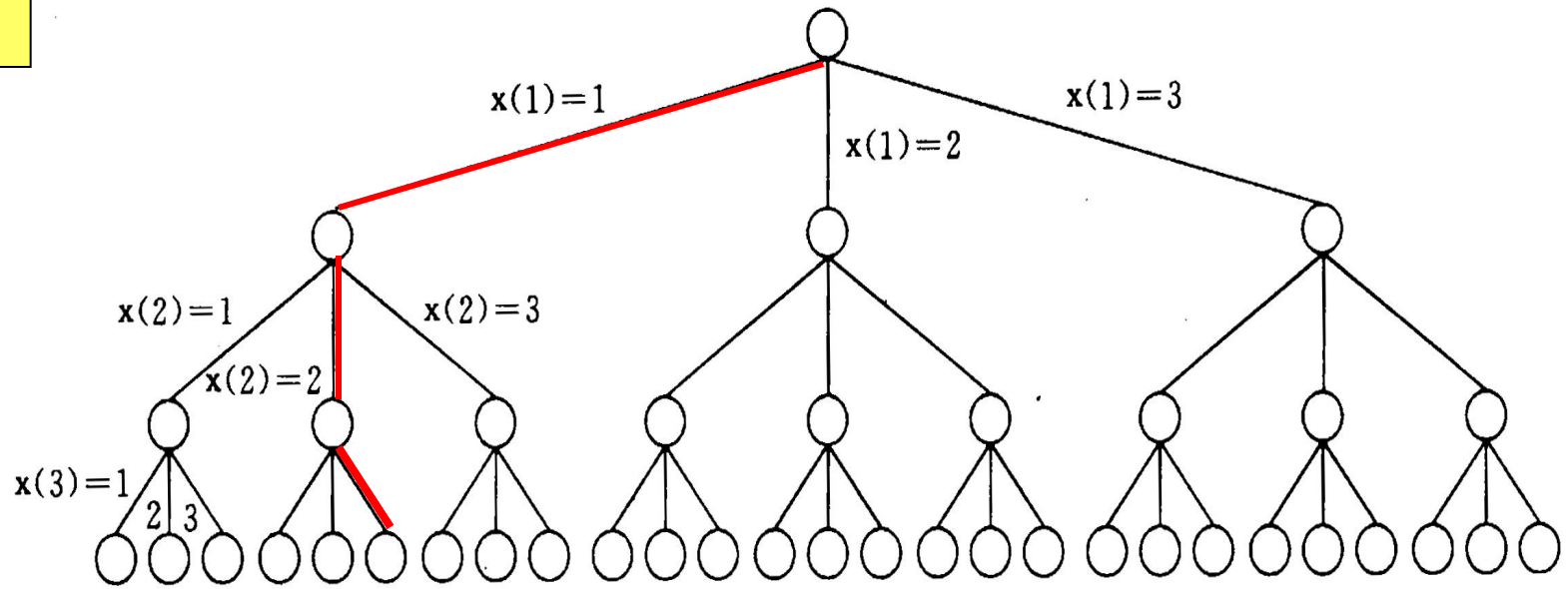
◆解空间树为排序树,是一棵n+1层的完全m叉树.

◆约束条件: x[i] ≠ x[j], 如果a[j][i]=1. (如果两点相邻,不能着同色)

n=3, m=3时的解空间树



$$a=\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$



边集合

```
int mColoring(int n, int m,  int **a  )
 { Color X;
  //初始化X
  X．n=n;  //图的顶点数
  X．m=m  //可用颜色数
  X．a=a;  //图的邻接矩阵
  X．Sum=0; //已找到的着色方案数
  int*p=new int [n+1];
  for (int i=0;  i<=n;  i++)
   p[i]=0;
  X. x=p    //当前解;
  X. Backtrack(1);
  delete [ ]p;
  returnX．sum;  }
```

➢判断颜色是否可用的Ok方法

```
bool Color::Ok(int k)
{//检查颜色可用性，即约束条件
    for(int j=1； j<=n； j++)
        if((a[k][j]==1)&&(x[j]==x[k]) )
         return false;
    return true;
 }
```

Backtrack函数中：

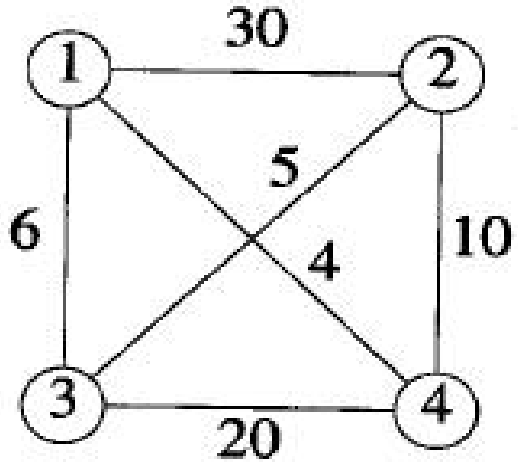
- 当i>n时，算法搜索至叶结点，得到新的m着色方案，当前找到的可m着色的方案数sum加一.
- 当i≤n时，当前扩展结点Z是解空间中的内部结点。
  - 该结点有x[i]=1,2,……,m共m个儿子结点。
  - 对当前扩展结点Z的每一个儿子结点，由函数OK检查其可行性，并以深度优先递归地对可行的子树进行搜索，或减去不可行的子树

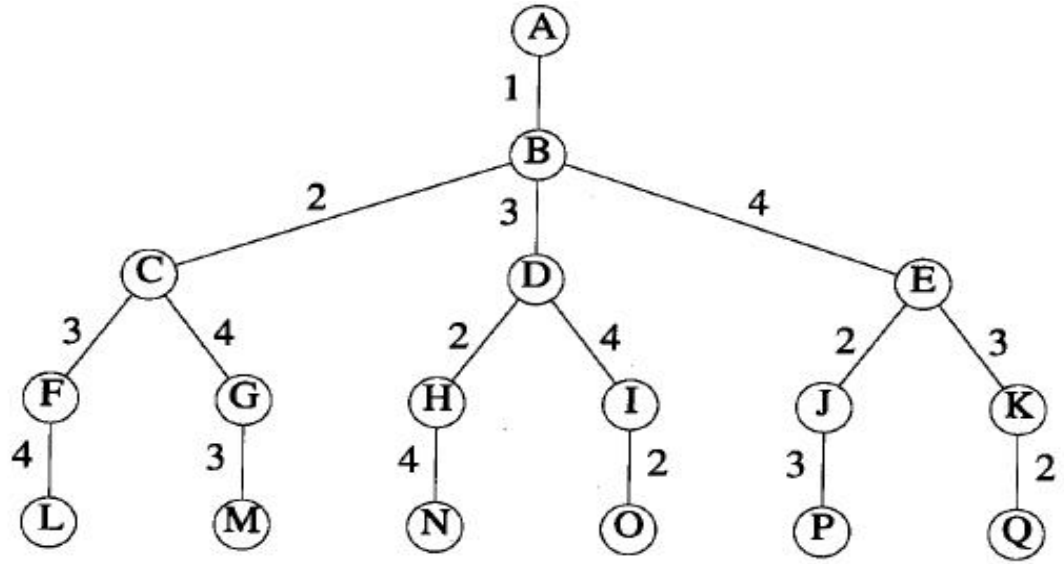
```
voidColor backtrack(int t)
{    if  (t>n)
     {   sum++;
         for(int i=1; i<=n; i++)
               cout << x[i] <<‘’;
         cout << endl ;       }
   else
         for ( int i=1; i<=m; i++)
         {    x[t]=i;
              if (Ok(t))
                Backtrack( t+1);              }
}
```

算法复杂性：$\sum_{i=0}^{n-1} m^i(mn) = nm(m^n - 1)/(m-1) = O(nm^n)$

5.9 旅行售货员问题

- 旅行商问题的解空间是一个排列树。如果以$x=[1, 2, \cdots, n]$开始，那么通过产生从$x2$到xn的所有排列，可生成n顶点旅行商问题的解空间。





```
template<class T>
T AdjacencyWDigraph<T>::TSP(int v[])
{  /* 用回溯算法解决TSP，最优路径存入v [ 1 : n ]*/
   //初始化
    x = new int [n+1];          // x 是排列,定义解空间
    for (int i = 1; i <= n; i++)
          x[i] = i;
    bestc = NoEdge;             //最小耗费
    bestx = v;                  // 使用数组v来存储最优路径
    cc = 0;                     //保存目前所构造的路径的耗费

   Backtrack( 2 ) ;       // 搜索x [ 2 : n ]的各种排列
   delete [] x;
   return bestc;
}
```

```
Void  Backtrack(int i)     // TSP问题的回溯算法
{   if (i == n) // 位于一个叶子的父节点
    {//考虑完成旅行的最后两条边
        if (a[x[n-1]][x[n]] != NoEdge &&a[x[n]][1] != NoEdge &&
           (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc ||bestc == NoEdge))
        {
                for (int j = 1; j <= n; j++)
                bestx[j] = x[j];
                bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];
} }
```

当i=n 时，处在排列树的叶节点的父节点上，需要验证

① x[n-1] 到x[n] 有一条边，

② 从x[n] 到起点x[1] 有一条边。

③ 该旅行是目前发现的最优旅行

```
Else
    {     for (int j = i; j <= n; j++)        //能移动到子树x [ j ]吗?
            if (a[x[i-1]][x[j]] != NoEdge &&(cc + a[x[i-1]][x[i]] < bestc ||
                bestc == NoEdge))            //能搜索该子树
            {     Swap(x[i], x[j]);
                  cc += a[x[i-1]][x[i]];
                  Backtrack( i + 1 ) ;
                  cc -= a[x[i-1]][x[i]];
                  Swap(x[i], x[j]);
            }
}}
```

当i＜n 时，检查当前i-1 层节点的孩子节点，并且仅当以下情况出现时，移动到孩子节点之一：

① 有从x[i-1] 到x[i] 的一条边；

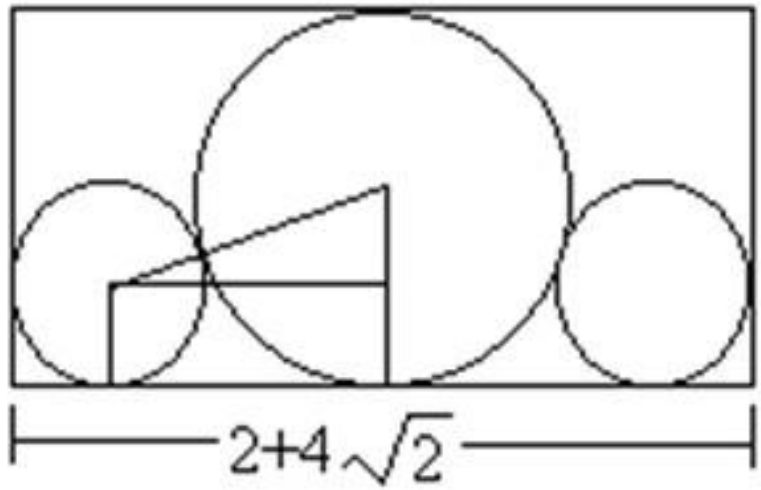
② 路径x[1:i] 的耗费小于当前最优解的耗费。

每次找到一个更好的旅行时，除了更新bestx 的耗费外，Backtrack需耗时O ((*n*- 1) !)。
因为需发生O ((*n*-1)!) 次更新且每一次更新的耗费为(*n*)时间，因此更新所需时间为O (*n** (*n*- 1) !)。

5.10 圆排列问题

*问题描述：*给定n个大小不等的圆$c_1, c_2, \ldots, c_n$，现要将这n个圆排进一个矩形框中，且要求各圆与矩形框的底边相切。圆排列问题要求从n个圆的所有排列中找出有最小长度的圆排列。

例如，当n=3，且所给的3个圆的半径分别为1，1，2时，这3个圆的最小长度的圆排列如图所示。



定义变量:

- min:当前最优值
- *x: 当前圆排列圆心横坐标,第一个圆的圆心横坐标为0
- *r:当前圆排列,初始值为 n个圆的半径

方法:

- Center:计算当前所选择的圆在当前圆排列中圆心的横坐标
- Computer:计算当前圆排列的长度
- CirclePerm:返回找到的最小圆排列长度

```
void Circle::Backtrack(int t)
{     if (t>n)    { Compute(); }  //计算当前圆排列的长度
   else
   {   for (int j = t; j <= n; j++)
       {  Swap(r[t], r[j]);
          float centerx=Center(t);   //计算当前所选圆的圆心坐标
          if (centerx+r[t]+r[1]<min)        //下界约束
          {      x[t]=centerx;
                 Backtrack(t+1);          }
          Swap(r[t], r[j]);
} } }
```

排列树