算法设计与分析

主讲教师：邵蓥侠

联系方式：shaoyx@bupt.edu.cn

个人主页：https://shaoyx.github.io/

第5章 回溯法

学习要点：

理解回溯法的深度优先搜索策略。

掌握用回溯法解题的算法框架。

学习应用范例归纳总结。

（1）递归回溯

（2）迭代回溯

（3）子集树算法框架

（4）排列树算法框架

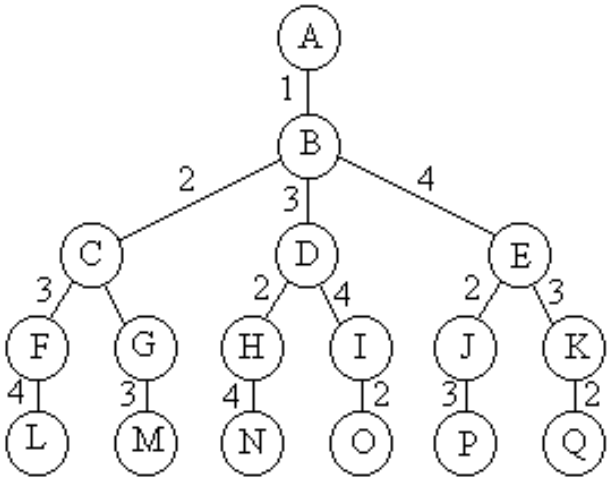
（5）...

1

通用算法、万能算法?

理解
回溯法的深度优先搜索策略

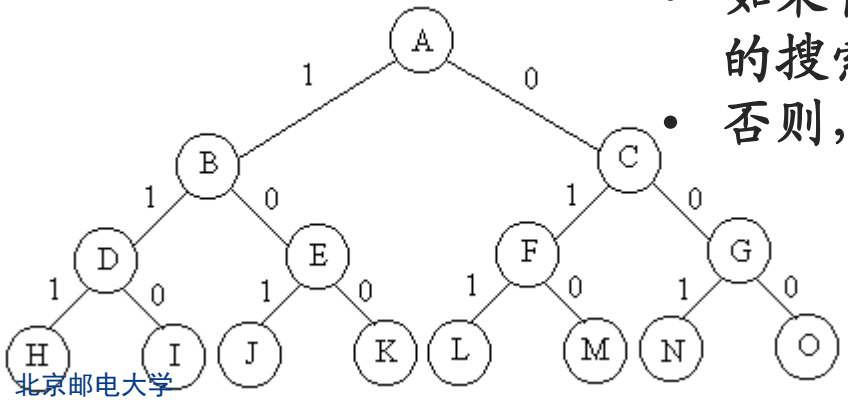
回溯法



回溯法 —“通用解题法”—“万能算法”。
回溯法可以系统的搜索一个问题的所有解或任一解。
回溯法是一个即带有系统性又带有跳跃性的搜索算法。

（1）它在问题的解空间树中，按深度优先策略，从根节点出发搜索解空间树。
（2）算法搜索至解空间树的任一结点时，先判断该节点是否包含问题的解。

- 如果肯定不包含，则跳过对以该结点为根的子树的搜索，逐层向其祖先节点回溯；
- 否则，进入该子树，继续按深度优先策略搜索



生成问题状态的基本方法

- **扩展结点**:一个正在产生子结点称为扩展结点
- **活结点**:一个自身已生成但其子节点还没有全部生成的节点称做活结点
- **死结点**:一个所有子节点已经产生的结点称做死结点
- **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子节点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的下一个子节点（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
- **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(bounding function)**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**

回溯法的算法框架

1. 问题的解空间
2. 回溯法的基本思想
3. 递归回溯
4. 迭代回溯
5. 子集树与排列树

1. 问题的解空间

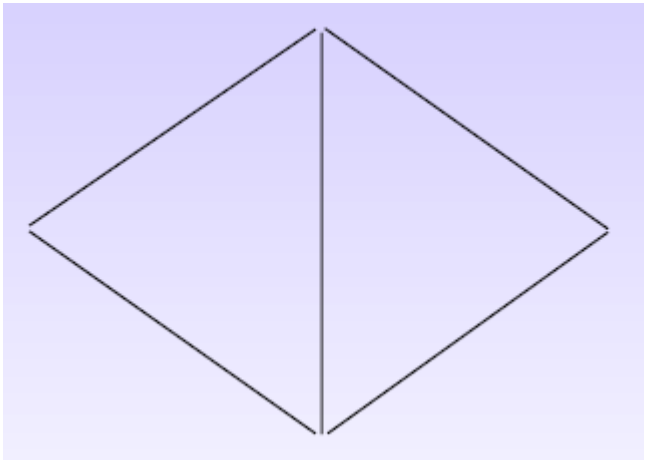
复杂问题常常有很多的可能解，这些可能解构成了问题的解空间。解空间也就是进行穷举的搜索空间，所以，**解空间中应该包括所有的可能解**。确定正确的解空间很重要，如果没有确定正确的解空间就开始搜索，可能会增加很多重复解，或者根本就搜索不到正确的解。

问题的解空间举例

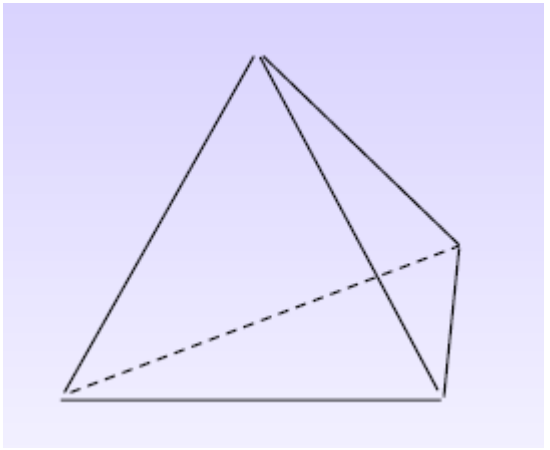
例如：桌子上有6根火柴棒，要求以这6根火柴棒为边搭建4个等边三角形。

错误的解空间将不能搜索到正确答案

二维搜索空间无解



三维搜索空间的解



回溯法的解空间

对于任何一个问题，可能解的表示方式和它相应的解释隐含了解空间及其大小。

例如，对于有n个物品的0/1背包问题，其可能解的表示方式：

（1）可能解由一个不等长向量组成，当物品$i(1 \leqslant i \leqslant n)$装入背包时，解向量中包含分量$i$，否则，解向量中不包含分量$i$，当$n=3$时，其解空间是：

{ ( ), (1), (2), (3), (1, 2), (1, 3), (2, 3), (1, 2, 3) }

（2）可能解由一个等长向量$\{x1, x2, \ldots, xn\}$组成，其中$xi=1(1 \leqslant i \leqslant n)$表示物品$i$装入背包，$xi=0$表示物品$i$没有装入背包，当$n=3$时，其解空间是：

{(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0), (1, 1, 1) }

回溯法的解空间

为了用回溯法求解一个具有n个输入的问题，一般情况下，将其可能的解表示为满足某个约束条件的等长向量

$$X = (x_1, x_2, \ldots, x_n),$$

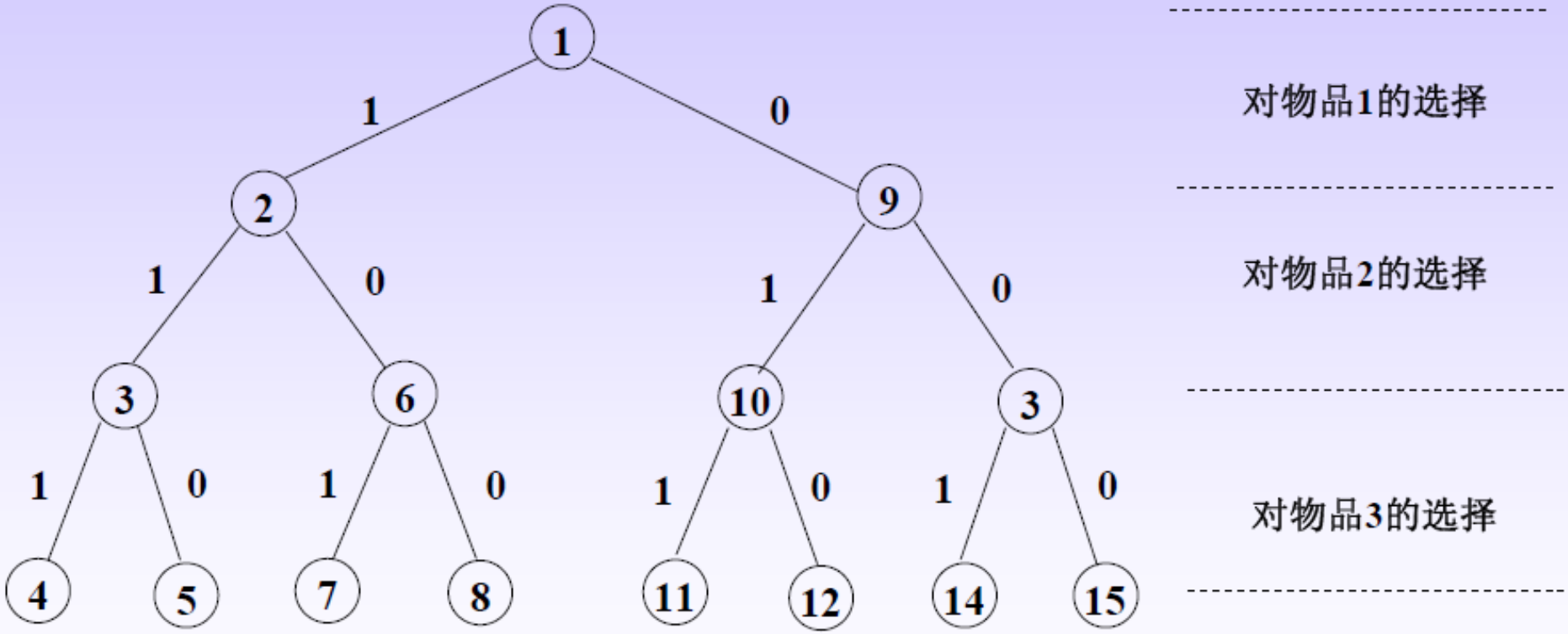
其中分量$x_i(1 <= i <= n)$的取值范围是某个有限集合$S_i = \{a_{i1}, a_{i2}, \ldots, a_{in}\}$，所有可能的解向量构成了问题的解空间。

回溯法的解空间的组织

问题的解空间一般用解空间树（Solution Space Trees,也称状态空间树）的方式组织，树的根结点位于第1层，表示搜索的初始状态，第2层的结点表示对解向量的第一个分量做出选择后到达的状态，第1层到第2层的边上标出对第一个分量选择的结果，依此类推，从树的根结点到叶子结点的路径就构成了解空间的一个可能解。

回溯法的解空间—0/1背包

对于n=3的0/1背包问题，其解空间树如下图所示，树中的8个叶子结点分别代表该问题的8个可能解。

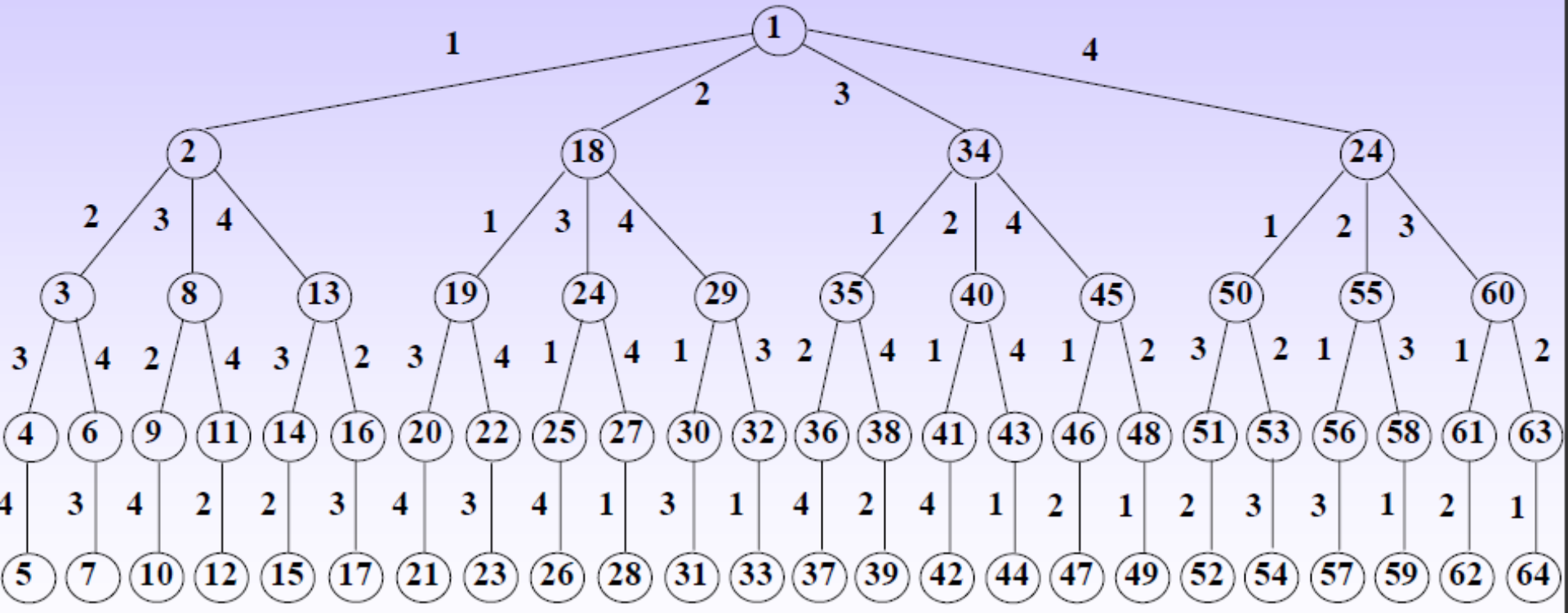


旅行售货员问题

- 某个售货员要到若干城市去推销商品，已知各城市之间的路程（或旅费）。他要选定一条从驻地城市出发，经过每个城市一遍，最后回到驻地的路线，使总的路程（或总旅费）最小。

回溯法的解空间—旅行售货员问题

对于n=4的TSP问题，其解空间树如下图所示，树中的24个叶子结点分别代表该问题的24个可能解，例如结点5代表一个可能解，路径为1→2→3→4→1，长度为各边代价之和。



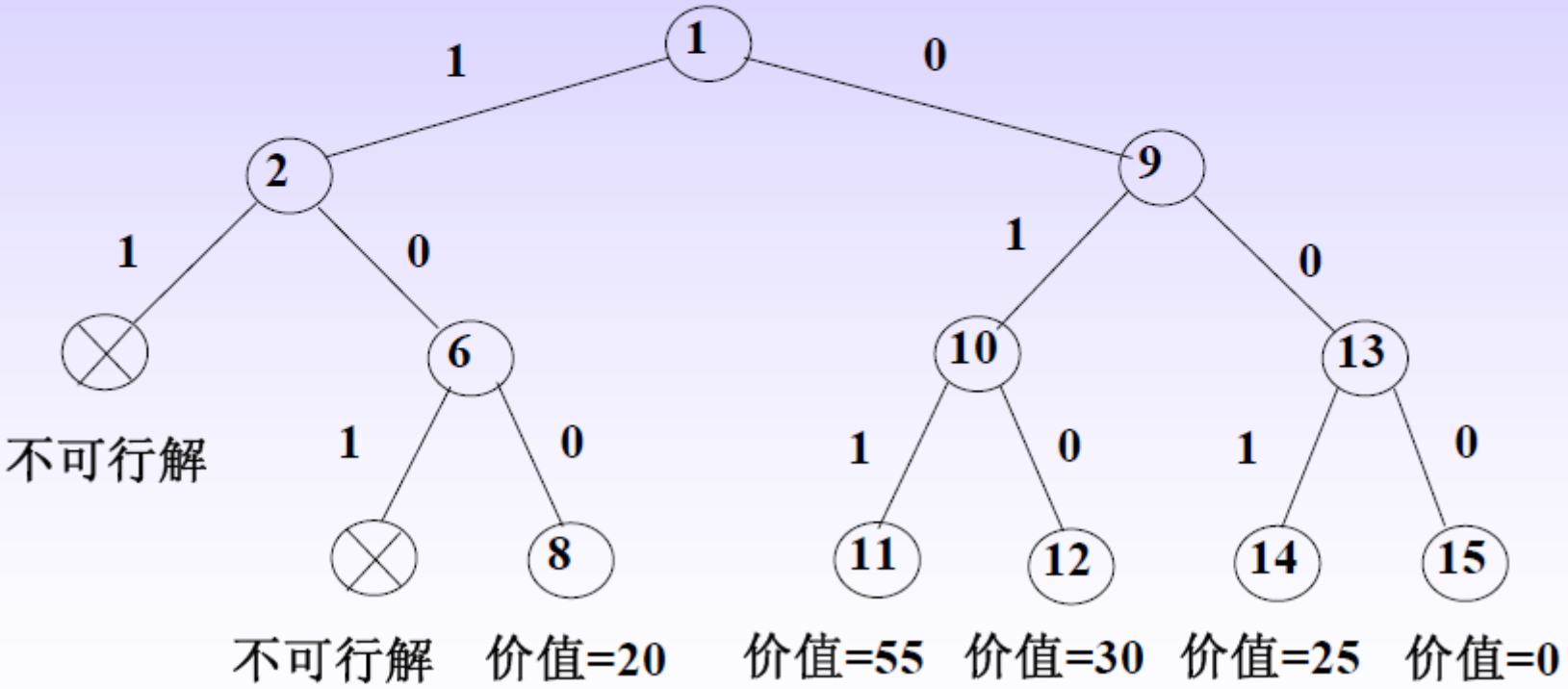
n=4的TSP问题的解空间树

2. 回溯法的基本思想

回溯法从根结点出发，按照深度优先策略遍历解空间树，搜索满足约束条件的解。在搜索至树中任一结点时，先判断该结点对应的部分解是否满足约束条件，或者是否超出目标函数的界，也就是判断该结点是否包含问题的（最优）解，如果肯定不包含，则跳过对以该结点为根的子树的搜索，即所谓剪枝（Pruning）；否则，进入以该结点为根的子树，继续按照深度优先策略搜索。

回溯法的基本思想举例—0/1背包

例如，对于n=3的0/1背包问题，三个物品的重量为{20, 15, 10}，价值为{20, 30, 25}，背包容量为25，从下图所示的解空间树的根结点开始搜索，搜索过程如下：



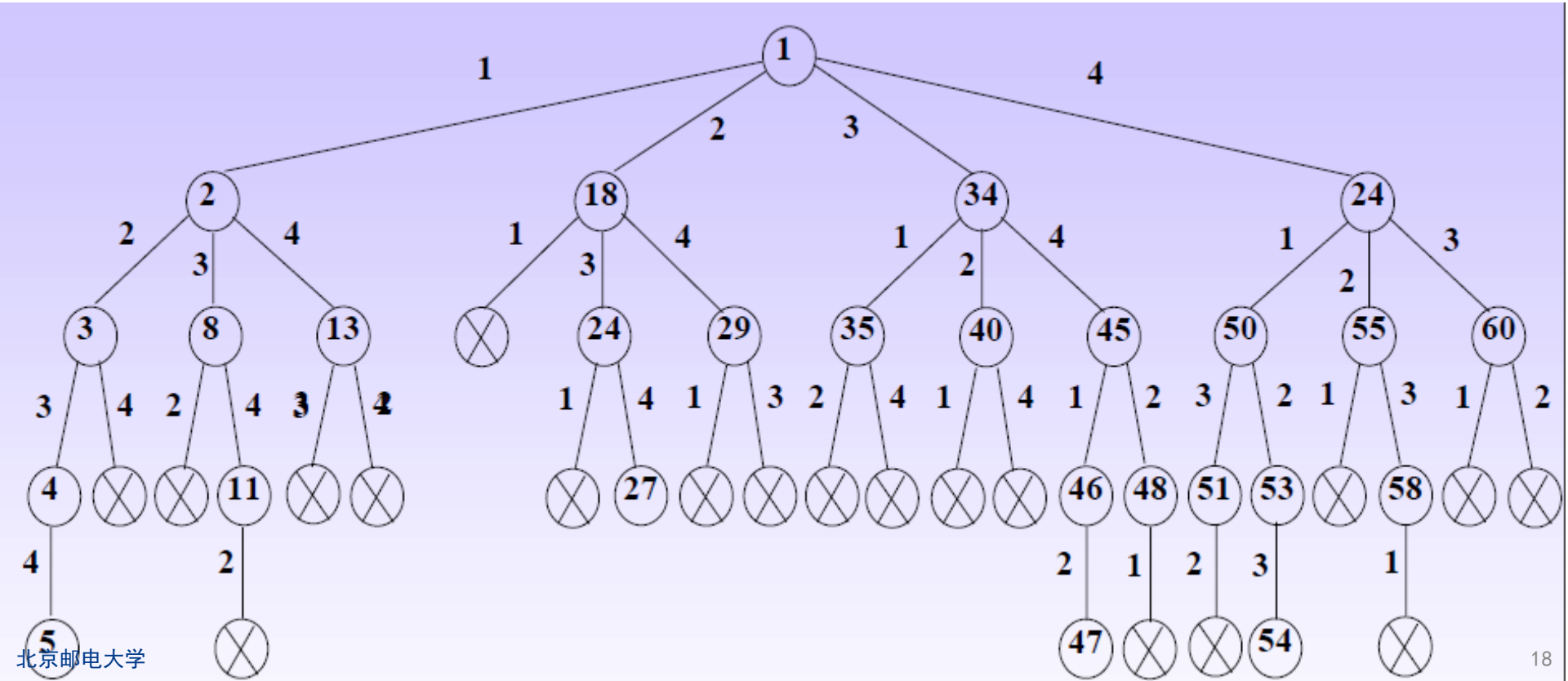
旅行售货员问题

当前找到的最小费用：10

再如，对于n=4的TSP问题，其代价矩阵如下图所示，

$$C=\begin{pmatrix} \infty & 3 & 6 & 7 \\ 12 & \infty & 2 & 8 \\ 8 & 6 & \infty & 2 \\ 3 & 7 & 6 & \infty \end{pmatrix}$$

TSP问题的代价矩阵



回溯法的基本思想

➢ 回溯法的搜索过程涉及的结点（称为搜索空间）只是整个解空间树的一部分，在搜索过程中，通常采用两种策略避免无效搜索：

（1）用约束条件剪去得不到可行解的子树；

（2）用目标函数剪去得不到最优解的子树。

这两类函数统称为剪枝函数（Pruning Function）。

❖ 需要注意的是，问题的解空间树是虚拟的，并不需要在算法运行时构造一棵真正的树结构，只需要存储从根结点到当前结点的路径。

回溯法的解题基本步骤

(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。

- 用回溯法解题的一个显著特征是在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))。而显式地存储整个解空间则需要$O(2^{h(n)})$或O(h(n)!)内存空间。

回溯法的求解过程

由于问题的解向量$X=(x_1, x_2, \ldots, x_n)$中的每个分量$x_i(1\leqslant i\leqslant n)$都属于一个有限集合$S_i=\{a_{i1}, a_{i2}, \ldots, a_{iri}\}$，因此，回溯法可以按某种顺序（例如字典序）依次考察笛卡儿积$S_1\times S_2\times\ldots\times S_n$中的元素。

初始时，令解向量X为空，然后，从根结点出发，选择S_1的第一个元素作为解向量X的第一个分量，即$x_1=a_{11}$，如果$X=(x_1)$是问题的部分解，则继续扩展解向量X，选择S_2的第一个元素作为解向量X的第2个分量，否则，选择S_1的下一个元素作为解向量X的第一个分量，即$x_1=a_{12}$。依此类推，一般情况下，如果$X=(x_1, x_2, \ldots, x_i)$是问题的部分解，则选择$S_{i+1}$的第一个元素作为解向量$X$的第$i+1$个分量时，有下面三种情况：

回溯法的求解过程

（1）如果$X=(x_1, x_2, \ldots, x_{i+1})$是问题的最终解，则输出这个解。如果问题只希望得到一个解，则结束搜索，否则继续搜索其他解；

（2）如果$X=(x_1, x_2, \ldots, x_{i+1})$是问题的部分解，则继续构造解向量的下一个分量；

（3）如果$X=(x_1, x_2, \ldots, x_{i+1})$既不是问题的部分解也不是问题的最终解，则存在下面两种情况：

① 如果$x_{i+1}= a_{i+1,k}$不是集合S_{i+1}的最后一个元素，则令$x_{i+1}= a_{i+1,k+1}$，即选择S_{i+1}的下一个元素作为解向量X的第$i+1$个分量；

② 如果$x_{i+1}= a_{i+1,k}$是集合S_{i+1}的最后一个元素，就回溯到$X=(x_1, x_2, \ldots, x_i)$，选择$S_i$的下一个元素作为解向量$X$的第$i$个分量，假设$x_i= a_{ik}$，如果$a_{ik}$不是集合$S_i$的最后一个元素，则令$x_i= a_{i,k+1}$；否则，就继续回溯到$X=(x_1, x_2, \ldots, x_{i-1})$；

3. 递归回溯

```
void 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);
          }
}
```

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。

4. 迭代回溯

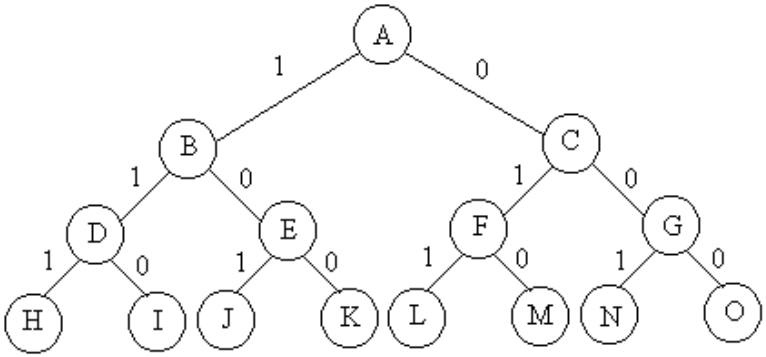
```
void iterativeBacktrack () {
  int t=1;
  while (t>0) {
    if (f(n, t)<=g(n, t))
      for (int i=f(n, t); i<=g(n, t); i++) {
        x[t]=h(i);
        if (constraint(t)&&bound(t)) {
          if (solution(t)) output(x);
          else t++;}
        }
    else t--;
    }
}
```

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。

回溯法：子集树和排列树

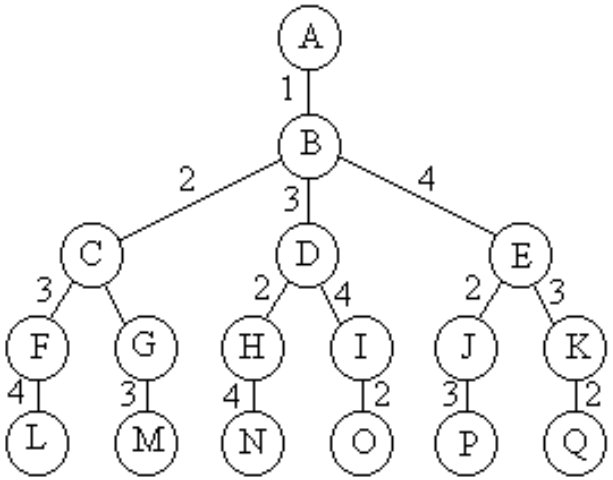
- 子集树：当所给的问题是从n个元素的集合S中找出满足某种性质的子集时，相应的解空间称为子集树。
- 排列树：当所给的问题是确定n个元素满足某种性质的排列时，相应的解空间树成为排列树。

子集树与排列树



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 (legal(t)) backtrack(t+1);
      }
}
```

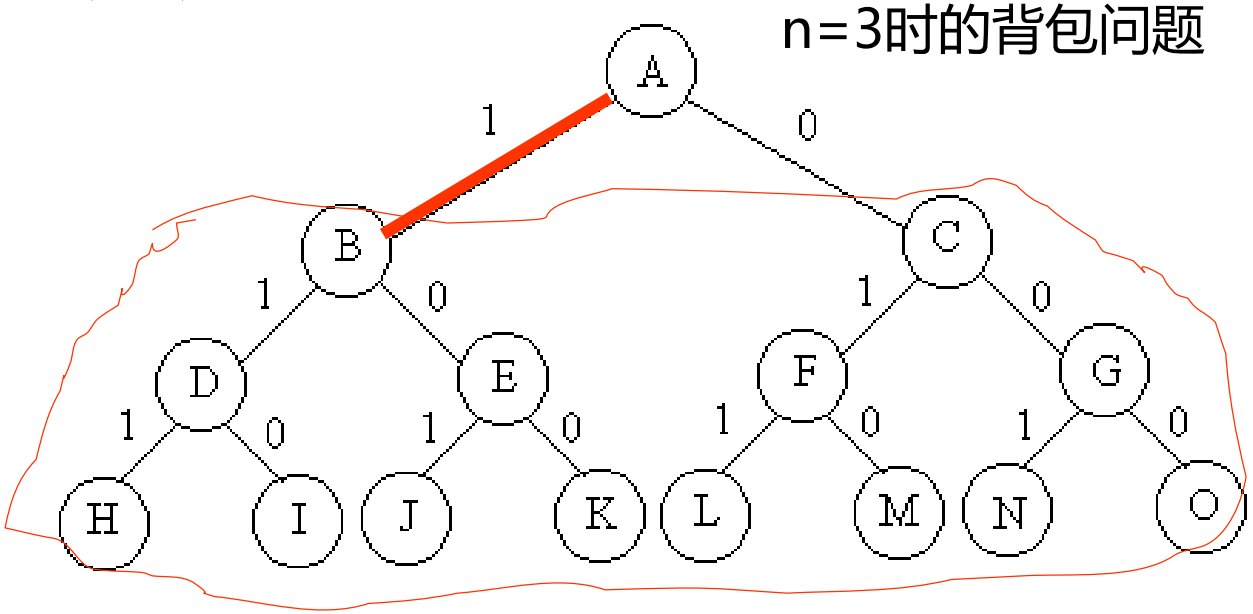


遍历排列树需要O(n!)计算时间

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

```
void backtrack (int t)  //已考虑t-1个物品，部分解<x[1],x[2],..,x[t-1],?, ?, …, ?>;本次考虑选定第t个物品，e.g. t=1,对应于根节点A、第1个物品。
{
 if (t>n) output(x);
 else
    for (int i=0;i<=1;i++) { //第t个物品可以“被选(1)”或“不选(0)”，
                             //对应A的左右子树。
          x[t]=i;           //考虑第t个物品后，部分解变为<x[1], x[2], .., x[t], ?, ?, …, ?>; e.g. <1,?,?>
      if (constraint(t)&&bound(t)) //所选分支的部分解是否合法
          backtrack(t+1);
    }
}
```

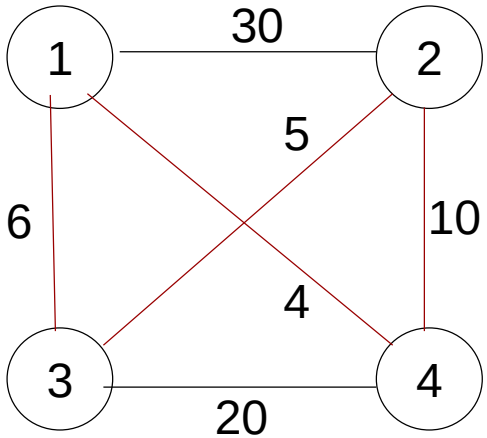
n=3时的背包问题

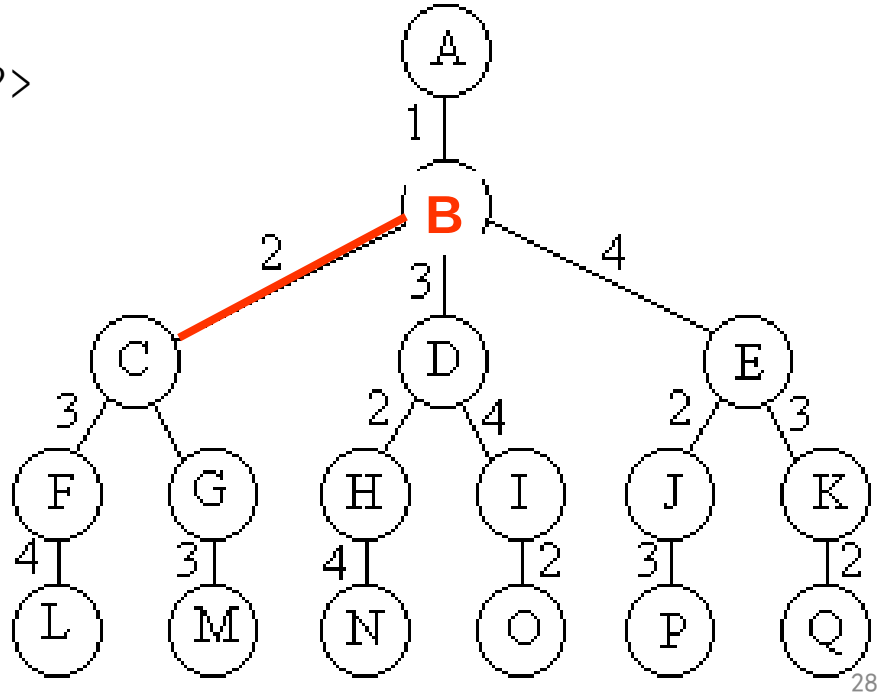


初始时，X=<1,2,3,…,n>, 即x[i]=i

```
void backtrack (int t)
{
 if (t>n) output(x);
 else
    for (int i=t;i<=n;i++) {//依次考虑选择后面的t、t+1、..n步所走城市.
      swap(x[t], x[i]);        //初始时，x[i]=i. 将第t步设置为x[t]=i。
                               //E.g. x[2]=2, 部分解< 1, i, ?,?>, i=2,3,4
      if (constraint(t)&&bound(t)) backtrack(t+1);
        //当前部分解合法，搜索第t+1步;
          e.g. C结点，对应部分解<1,2, ?,?>
      swap(x[t], x[i]);
    }
}
```

//已选择前t-1个城市， 部分解<x[1],x[2],..,x[t-1], ?, ?, …, ?>;本次考虑选定第t个城市，e.g. t=2，对应于节点B、第2个城市，当前部分解<1, ?,?,?>。





枚举全排列

```
template<class type>
void perm(type list[], int k, int m)
{ if (k==m)
     //输出一个排列
  else
    for (int i=k ; i<=m; i++)
     { swap(list[k], list[i]);
       perm(list, k+1, m);
       swap(list[k], list[i]);
        }
}
```

eg: list[]={1, 2, 3, 4}

↑

k

输出结果：

1 2 3 4

1 2 4 3

1 3 2 4

1 3 4 2

……

函数调用： **perm(list, 0, n-1)**。求**n**个元素的排列，应有**n!**种，输出**n!**种排列时间性能**nn!**

```
template<class type>
void perm(type list[], int k, int m)
1 { if (k==m)
      //输出一个排列
  else
4     for (int i=k ; i<=m; i++)
5       { swap(list[k], list[i]);
6         perm(list, k+1, m);
7         swap(list[k], list[i]);}
9  }
```

eg: list[]={1, 2, 3, 4}

perm({1, 2,3,4})

//*perm(list,1,4) k=1,m=4

i=1 k=1; list[1]←→list[1] {1,2,3,4}

1 perm({2,3,4}) k=k+1=2

i=2 list[2]←→list[2]

2 perm({3,4}) k=k+1=3

i=3 k=3 list[3]←→list[3]

3 perm({4}) k=k+1=4

k=m: 输出； perm({4}) 结束；

退栈；返回执行7语句

list[3]←→list[3]

i=i+1=4, k=3 list[3]←→list[4]

4 perm({3}) k=k+1=4

k=m: 输出； perm({4}) 结束；

退栈；返回执行7语句

list[3]←→list[4]

i=i+1=5, i>m for 结束，即perm({3,4})结束

list[2]←→list[2]

i=i+1=3 list[2]←→list[3]

3 perm({2,4}) ……

i=i+1=4 list[2]←→list[4]

4 perm({3,2}) ……

i=i+1=2 list[1]←→list[2] {2,1,3,4}

```
template<class type>
void perm(type list[], int k, int m)
1 { if (k==m)
      //输出一个排列
  else
4     for (int i=k ; i<=m; i++)
5       { swap(list[k], list[i]);
6         perm(list, k+1, m);
7         swap(list[k], list[i]);}
9  }
```

eg: list[]={1, 2, 3, 4}

perm({1, 2,3,4})

//*perm(list,1,4) k=1,m=4

i=i+1=2 list[1]←→list[2] {2,1,3,4}

2 perm({1,3,4}) k=k+1=2

i=k=2 list[2]←→list[2]

1 perm({3,4}) k=k+1=3

.....

list[2]←→list[2]

i=i+1=3 list[2]←→list[3] {2,3,1,4}

3 perm({1,4}) .....

i=i+1=4 list[2]←→list[4] {2,4,1,3}

4 perm({1,3}) .....

.....

i=i+1=3 list[1]←→list[3] {3,2,1,4}

3 perm({2,1,4})

....

i=i+1=4 list[1]←→list[4] {4,2,3,1}

4 perm({2,3,1})

....

2

装载问题、作业调度和0/1背包问题

回溯法的应用例子

装载问题

有一批共n个集装箱要装上2艘载重量分别为c1和c2的轮船，其中集装箱i的重量为wi，且 $\sum_{i=1}^{n} w_i \le c_1 + c_2$

装载问题要求确定：是否有一个合理的装载方案可将这n个集装箱装上这2艘轮船。如果有，找出一种装载方案。

装载例子

n = 3， c1= c2 = 50

w1 = [10, 40, 40]

w2 = [20, 40, 40]

有一批共n个集装箱要装上2艘载重量分别为c1和c2的轮船，其中集装箱i的重量为wi，且 $\sum_{i=1}^{n} w_i \le c_1 + c_2$

容易证明，如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。

(1)首先将第一艘轮船尽可能装满；

(2)将剩余的集装箱装上第二艘轮船。

将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集，使该子集中集装箱重量之和最接近。由此可知，装载问题等价于以下特殊的0-1背包问题。

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

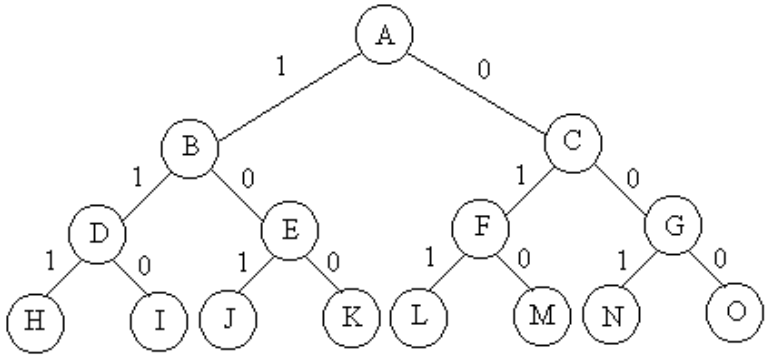
$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \le c_1$$

$$x_i \in \{0,1\}, 1 \le i \le n$$

装载问题 – 回溯法实现

```
class loading
{
        friend Type MaxLoading(Type [], Type int);
        //初始化操作
        private:
                void Backtrack( int i);//对解空间搜索
                int n; //集装箱数
                Type *w, //集装箱重量数组
                      c, //第一艘轮船的载重量
                      cw,//当前载重量
                      bestw;//当前最优载重量
};
```

```
template < class Type >
void Loading< Type > :: Backtrack( int i )
{//搜索第i层结点，i从1开始，则完成了整个空间的搜索
        if (i > n)
        {//到达了叶子节点，获得一个可行解
                if (cw > bestw) bestw = cw;
                return;
        }
        //搜索子树
        if ( cw + w[i] <= c)
        {//x[i] = 1; //二叉树的左孩子
                cw+= w[i];
                Backtrack(i+1);
                cw -= w[i]; //回溯
        }
        Backtrack(i+1); //x[i] =0;//二叉树的右孩子
}
```



```
template < class Type>
Type MaxLoading (Type w[], Type c, int n)
{//返回最优载重量
        Loading<Type> X;
        //初始化x
        X.w = w;
        X.c = c; //第一艘
        X.n = n;
        X.bestw = 0;
        X.cw = 0;
        //计算最优载重量
        X.Backtrack(1);
        return X.bestw;
}
```

约束函数?

```
template < class Type >
void Loading< Type > :: Backtrack( int i )
{//搜索第i层结点，i从1开始，则完成了整个空间的搜索
        if (i > n)
        {//到达了叶子节点，获得一个可行解
                if (cw > bestw) bestw = cwl
                return;
        }
        //搜索子树
        if ( cw + w[i] <= c)
        {//x[i] = 1; //二叉树的左孩子
                cw+= w[i];
                Backtrack(i+1);
                cw -= w[i]; //回溯
        }
        Backtrack(i+1); //x[i] =0;//二叉树的右孩子
}
```

当前载重量cw+剩余集装箱的重量r≤当前最优载重量bestw

```
class loading
{
        friend Type MaxLoading(Type [], Type int);
        //初始化操作
        private:
                void Backtrack( int i);//对解空间搜索
                int n; //集装箱数
                Type *w, //集装箱重量数组
                      c, //第一艘轮船的载重量
                      cw,//当前载重量
                      bestw;//当前最优载重量
                      r;    //剩余集装箱的重量
};
```

上界函数(不选择当前元素):

当前载重量cw+剩余集装箱的重量r≤当前最优载重量bestw

```
void backtrack (int i)
{// 搜索第i层结点
    if (i > n){  // 到达叶结点
      bestw = cw;  //上界函数使搜索到的每个叶结点都是
      return; }          //当前找到的最优解
    r - = w[i];
    if (cw + w[i] <= c) {// x[i] = 1; 搜索左子树
       cw += w[i];
       backtrack(i + 1);
       cw -= w[i];      }
    if (cw + r > bestw ) //上界函数判断, x[i] = 0; 搜索右子树
       backtrack(i + 1);
    r + = w[i];
}
```

上界函数(不选择当前元素):

```
template < class Type>
Type MaxLoading (Type w[], Type c, int n)
{//返回最优载重量
        Loading<Type> X;
        //初始化x
        X.w = w;
        X.c = c; //第一艘
        X.n = n;
        X.bestw = 0;
        X.cw = 0;
        X.r = 0;
        for( int i=1; i<=n; i++)
        X.r += w[i];
        //计算最优载重量
        X.Backtrack(1);
        return X.bestw;
}
```

构造最优解

在类Loading中增加两个私有数据成员x和bestx。

- X记录从根至当前结点的路径,x[i]=0/1;[1 0 1 1 0]
- Bestx记录当前最优解，算法搜索到达一个叶子结点处，就修正besx的值。

```
class loading
{
        friend Type MaxLoading(Type [], Type int);
        //初始化操作
        private:
                void Backtrack( int i);//对解空间搜索
                int n; //集装箱数
                *x,//当前解，数组
                *bestx//当前最优解，数组
                Type *w, //集装箱重量数组
                      c, //第一艘轮船的载重量
                      cw,//当前载重量
                      bestw;//当前最优载重量
                      r;    //剩余集装箱的重量
};
```

```
template < class Type >
void Loading< Type > :: Backtrack( int i )
{
        if (i > n)
        {
                if (cw > bestw)
                {
                        for(j =1; j<=n;j++)
                                bestx[j] =x[j];
                        bestw = cw;
                }
                return;
        }
        //搜索子树
        r -= w[i];
        if ( cw + w[i] <= c)
        {//x[i] = 1; //二叉树的左孩子
                x[i] =1;
                cw+= w[i];
                Backtrack(i+1);
                cw -= w[i]; //回溯
        }
        if( cw+r > bestw)
        {
                x[i] = 0;
                Backtrack(i+1); //x[i] =0;//二叉树的右孩子
        }
        r += w[i];
}
```

复杂度分析

由于bestx可能被更新$O(2^n)$次，改进后算法的计算时间复杂性为$O(n2^n)$。

迭代回溯

数组x记录了解空间树中从根到当前扩展结点的路径，这些信息已包含了回溯法在回溯时所需信息。因此利用数组x所含信息，可将上述回溯法表示成非递归形式。由此可进一步省去O(n)递归栈空间。

```
public static int maxLoading (int [ ] w,
int c, int [ ] bestx)
{ // 迭代回溯法
    // 返回最优载重量及其相应解
    // 初始化根结点
    int i = 1;    // 当前层
    int n = w.length - 1;
    int [ ] x = new int [n + 1];  // x[1:i-1]
为当前路径
    int bestw = 0;      // 当前最优载重量
    int cw = 0;          // 当前载重量
    int r = 0;           // 剩余集装箱重量
    for (int j = 1; j <= n; j++)
      r += w[j];
```

```
// 搜索子树
while (true){
while (i <= n && cw + w[i] <= c)
  {// 进入左子树
    r -= w[i];
    cw += w[i];
    x[i] = 1;
    i++;   }
  if (i > n)
  {// 到达叶结点
    for (int j = 1; j <= n; j++)
      bestx[j] = x[j];
    bestw = cw;  }
```

```
        else {// 进入右子树
            r -= w[i];
            x[i] = 0;
            i++;   }
        while (cw + r <= bestw)
        {// 剪枝回溯
            i--;
            while (i > 0 && x[i] == 0)
            {// 从右子树返回
               r += w[i];
               i--;   }
            if (i == 0) {delete [ ] x; return bestw; }
            // 进入右子树
            x[i] = 0;
            cw -= w[i];
            i++;     }
      }
}
```

通用0/1背包 – 回溯法求解

```
Typep Knap<Typew, Typep>::Bound(int i)
{// 计算上界
    Typew cleft = c - cw;  // 剩余容量
    Typep b = cp;
    // 以物品单位重量价值递减序装入物品
    while (i <= n && w[i] <= cleft) {
        cleft -= w[i];
        b += p[i];
        i++;
        }
    // 装满背包
    if (i <= n) b += p[i]/w[i] * cleft;
    return b;
}
```

- 解空间：子集树
- 可行性约束函数：

$$\sum_{i=1}^{n} w_i x_i \le c$$

- 上界函数：

计算右子树中解的上界的更好方法是将剩余物品依其单位重量价值排序，然后依次装入物品，直至装不下时，再装入该物品的一部分而装满背包。由此得到的价值是右子树中解的上界。

通用0/1背包

```
template <class Typew, class Typep>
void Knap < Typew, Typep> :: Backtrack(int i)
{
        if (i > n) //到达叶子节点
        {
                bestp = cp;
                return;
        }
        if ((cw + w[i] <= c)
        {
                cw += w[i];
                cp += p[i];
                Backtrack(i + 1);
                cw -= w[i];
                cp -= p[i];
        }
        if (Bound ( i + 1) > bestp)
                Backtrack( i + 1 );
}
```

批处理作业调度问题

批处理作业调度问题

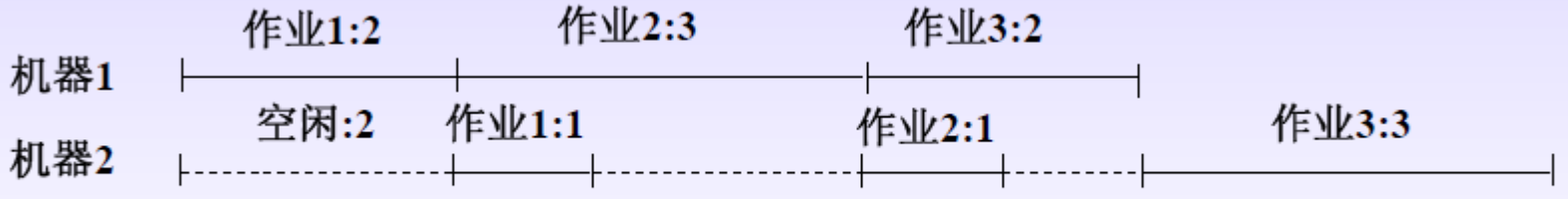
给定n个作业的集合$\{J_1,J_2,\ldots,J_n\}$。每个作业必须先由机器1处理，然后由机器2处理。作业J_i需要机器j的处理时间为t_{ji}。对于一个确定的作业调度，设F_{ji}是作业i在机器j上完成处理的时间。所有作业在机器2上完成处理的时间和称为该作业调度的完成时间和。

批处理作业调度问题要求对于给定的n个作业，制定最佳作业调度方案，使其完成时间和达到最小。

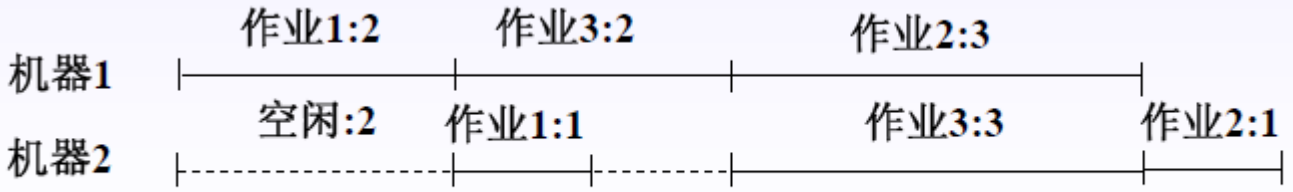
t_{ji}	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3

批处理作业调度问题

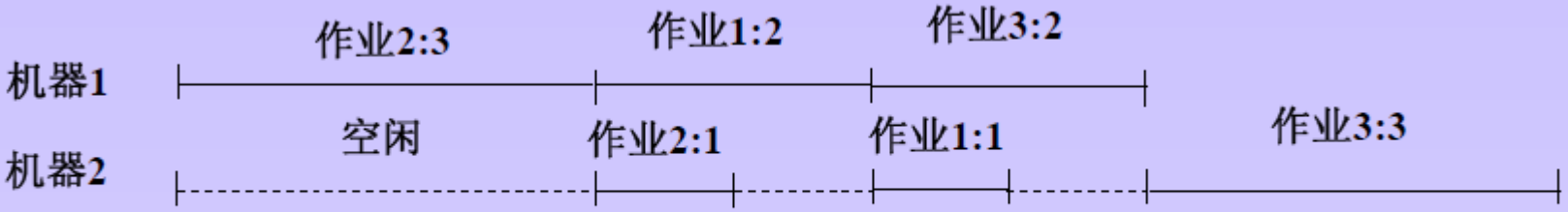
例：三个作业{1, 2, 3}，这三个作业在机器1上所需的处理时间为(2, 3, 2)，在机器2上所需的处理时间为(1, 1, 3)：



(a) 调度方案(1, 2, 3)，完成时间和为19



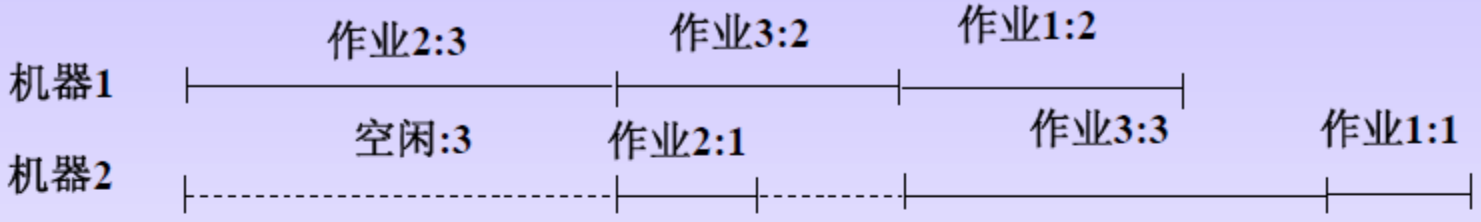
(b) 调度方案(1, 3, 2)，完成时间和为18



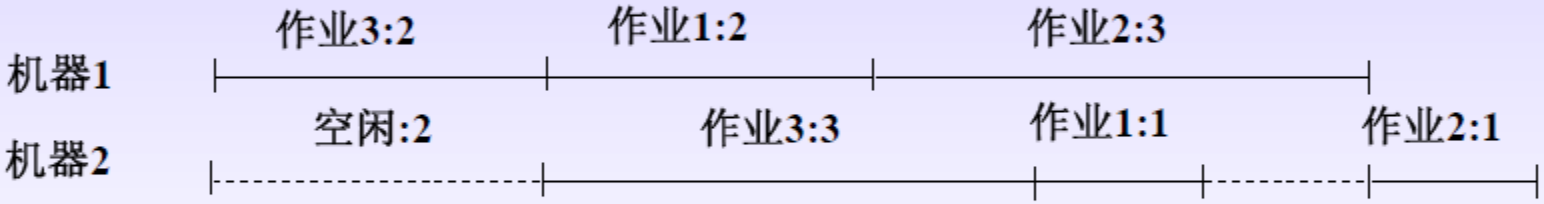
(c) 调度方案(2, 1, 3)，完成时间和为20

批处理作业调度问题

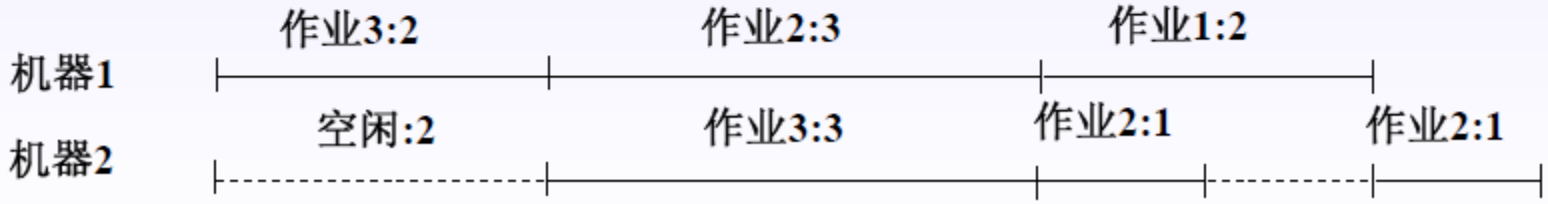
例：三个作业{1, 2, 3}，这三个作业在机器1上所需的处理时间为(2, 3, 2)，在机器2上所需的处理时间为(1, 1, 3)：



(d) 调度方案(2, 3, 1)，完成时间和为21



(e) 调度方案(3, 1, 2)，完成时间和为19



(f) 调度方案(3, 2, 1)，完成时间和为19

批处理作业调度问题

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。

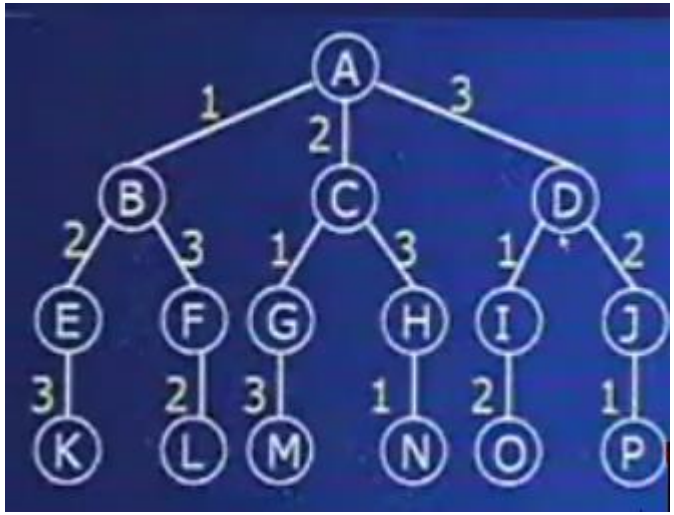
可以证明，
存在一个最优作业调度使得在机器1和机器2上作业以相同次序完成。

则最佳调度方案是(1, 3, 2)，其完成时间和为18。

批处理作业调度问题 -- 解空间?

是一个排列树。

子树随着作业的减少而减少。



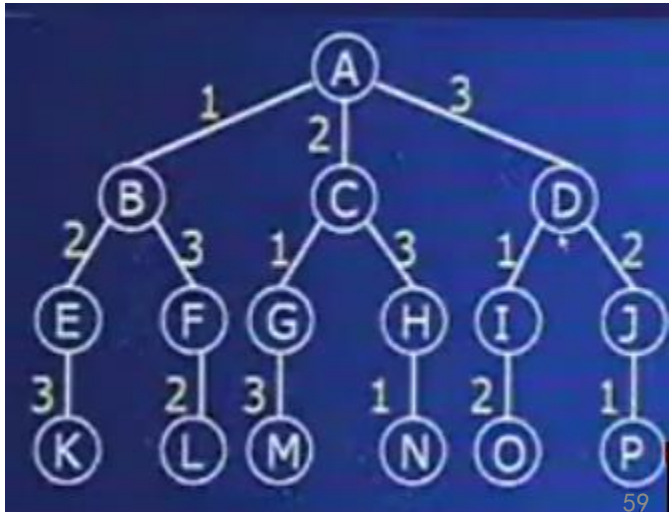
批处理作业调度问题

```
class Flowshop {
        friend Flow(int**, int, int []);
        private:
                        void Backtrack(int i);
                        int  **M,   // 各作业所需的处理时间
                        *x,    // 当前作业调度
                        *bestx,   // 当前最优作业调度
                        *f2,   // 机器2完成处理时间
                        f1,   // 机器1完成处理时间
                        f,    // 完成时间和
                        bestf,   // 当前最优值
                        n;  // 作业数};
```

批处理作业调度问题

```
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++) {
            f1+=M[x[j]][1];
            f2[i]=((f2[i-1]>f1)?f2[i-1]:f1)+M[x[j]][2];
            f+=f2[i];
            if (f < bestf) {
                Swap(x[i], x[j]);
                Backtrack(i+1);
                Swap(x[i], x[j]);
                }
            f1- =M[x[j]][1];
            f- =f2[i];
            }
}
```

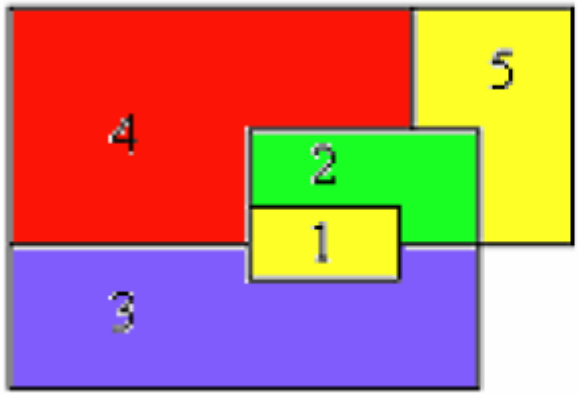
t_{ji}	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3

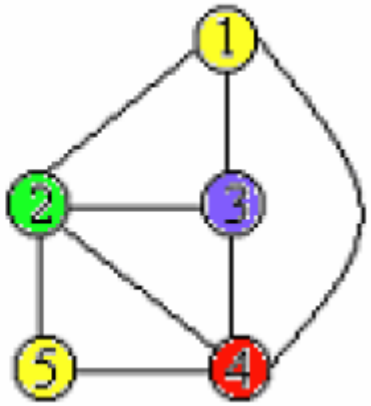


M图着色问题

M图着色问题

给定无向连通图G和m种不同的颜色。用这些颜色为图G的各顶点着色，每个顶点着一种颜色。是否有一种着色法使G中每条边的2个顶点着不同颜色。这个问题是图的m可着色判定问题。若一个图最少需要m种颜色才能使图中每条边连接的2个顶点着不同颜色，则称这个数m为该图的色数。求一个图的色数m的问题称为图的m可着色优化问题。





M图着色问题

由于用m种颜色为无向图$G=(V, E)$着色，其中，V的顶点个数为n，可以用一个n元组$C=(c_1, c_2, \ldots, c_n)$来描述图的一种可能着色，其中，$c_i \in \{1, 2, \ldots, m\}$ $(1 \leqslant i \leqslant n)$表示赋予顶点$i$的颜色。

例如，5元组(1, 2, 2, 3, 1)表示对具有5个顶点的无向图的一种着色，顶点1着颜色1，顶点2着颜色2，顶点3着颜色2，如此等等。

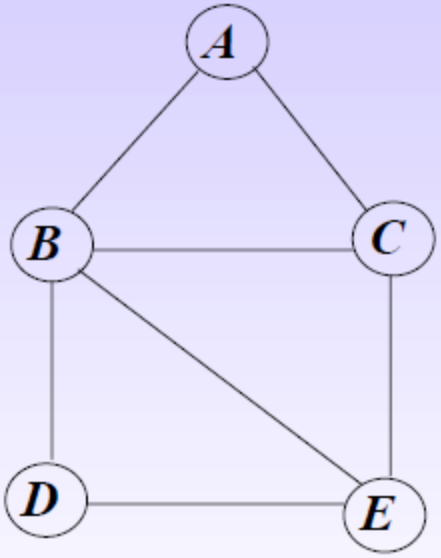
如果在n元组C中，所有相邻顶点都不会着相同颜色，就称此n元组为可行解，否则为无效解。

M图着色问题

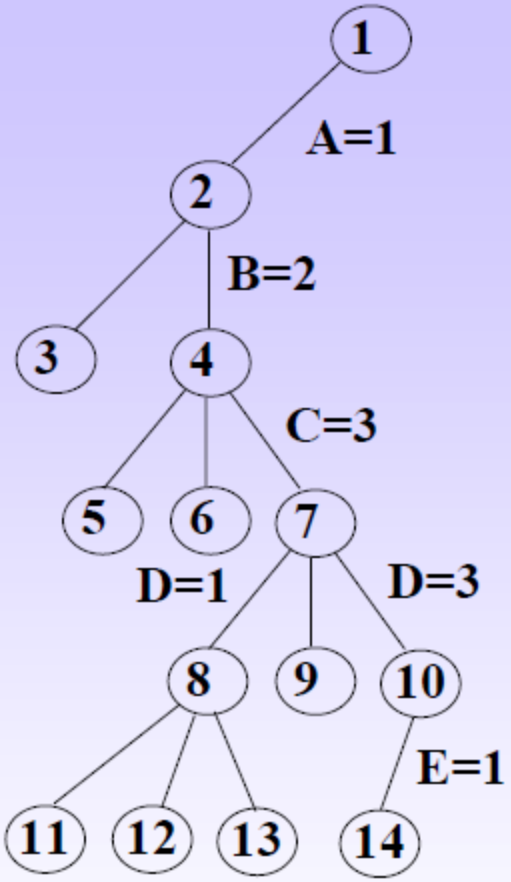
回溯法求解图着色问题，首先把所有顶点的颜色初始化为0，然后依次为每个顶点着色。在图着色问题的解空间树中，如果从根结点到当前结点对应一个部分解，也就是所有的颜色指派都没有冲突，则在当前结点处选择第一棵子树继续搜索，也就是为下一个顶点着颜色1，否则，对当前子树的兄弟子树继续搜索，也就是为当前顶点着下一个颜色，如果所有m种颜色都已尝试过并且都发生冲突，则回溯到当前结点的父结点处，上一个顶点的颜色被改变，依此类推。

M图着色问题

例：判断下图是否是3可着色。



(a) 一个无向图

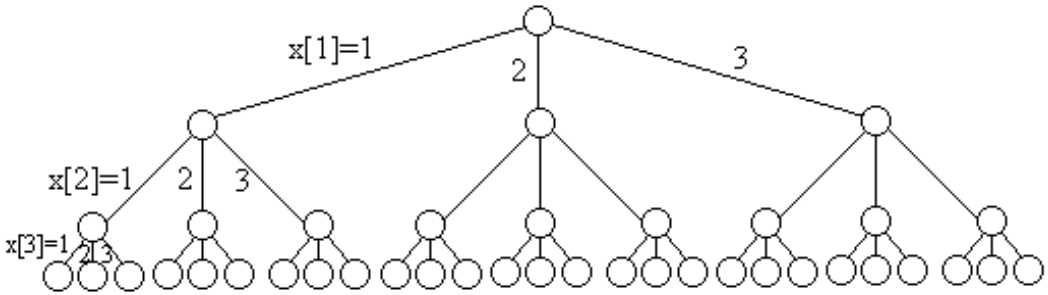


(b) 回溯法搜索空间

图的m着色问题

- 解向量：$(x_1, x_2, \ldots, x_n)$表示顶点i所着颜色x[i]
- 可行性约束函数：顶点i与已着色的相邻顶点颜色不重复。

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



图的m着色问题

复杂度分析

图m可着色问题的解空间树中内结点个数是 $\sum_{i=0}^{n-1} m^i$

对于每一个内结点，在最坏情况下，用ok检查当前扩展结点的每一个儿子所相应的颜色可用性需耗时O(mn)。因此，回溯法总的时间耗费是

$$\sum_{i=0}^{n-1} m^i (mn) = nm(m^n - 1)/(m-1) = O(nm^n)$$