第5章　回溯法

回溯法引言

口 实例

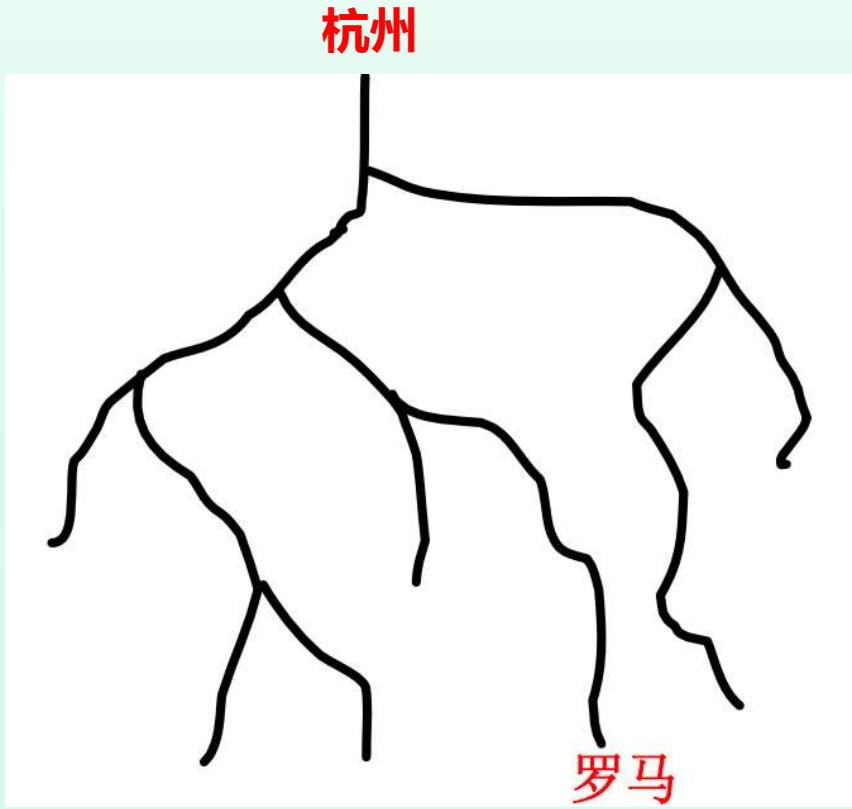
✓ 杭州走到罗马

✓ 若完全不认识路，会怎样走

口 两类问题

✓ 存在性问题（可行解）

✓ 优化问题（最优解）



回溯法引言

- 理论上
  - 寻找问题的解的一种可靠的方法是首先列出所有候选解，然后依次检查每一个，在检查完所有或部分候选解后，即可找到所需要的解。
- 但是
  - 当候选解数量有限并且通过检查所有或部分候选解能够得到所需解时，上述方法是可行的。
  - 若候选解的数量非常大（指数级，大数阶乘），即便采用最快的计算机也只能解决规模很小的问题。

回溯法引言

- 回溯和分枝限界法是比较常用的对候选解进行系统检查两种方法。
- 按照这两种方法对候选解进行系统检查通常会使问题的求解时间大大减少（无论对于最坏情形还是对于一般情形）。
- 可以避免对很大的候选解集合进行检查，同时能够保证算法运行结束时可以找到所需要的解。
- 通常能够用来求解规模很大的问题。

回溯法引言

- 回溯法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。
- 回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

回溯法引言

- 以深度优先的方式系统地搜索问题的解的算法称为回溯法
- 使用场合
  - 对于许多问题，当需要找出它的解的集合或者要求回答什么解是满足某些约束条件的最佳解时，往往要使用回溯法。
  - 这种方法适用于解一些组合数相当大的问题，具有“通用解题法”之称。
- 回溯法的基本做法
  - 是搜索，或是一种组织得井井有条的，能避免不必要搜索的穷举式搜索法。

回溯法引言

具体做法

- 系统性

回溯法在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。

- 跳跃性

算法搜索至解空间树的任意一点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

回溯法引言

回溯法与穷举查找是一样的吗?

- 可以把回溯法和分支限界法看成是穷举法的一个改进。

  该方法至少可以对某些组合难题的较大实例求解。

- 不同点

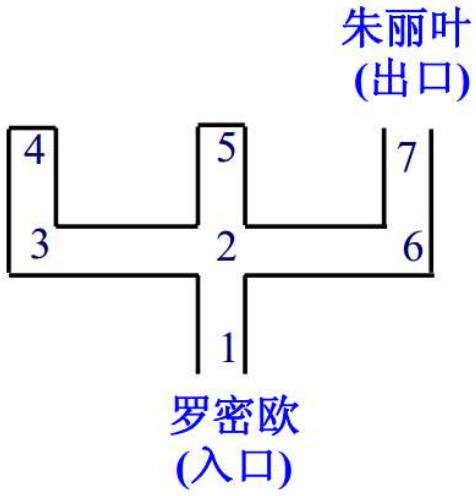
  - 每次只构造侯选解的一个部分
  - 然后评估这个部分构造解：如果加上剩余的分量也不可能求得一个解，就绝对不会生成剩下的分量

回溯法引言

罗密欧与朱丽叶问题：

迷宫是一些互相连通的交叉路口的集合，给定一个入口、一个出口。

当从入口到出口存在通路时，输出选中的一条通路；否则，输出无通路存在。



(a) 交叉路口

✓ **路口动作结果**

1向前进入2
2向左进入3
3向右进入4
4（死路）回溯进入3
3（死路）回溯进入2
2向前进入5
5（死路）回溯进入2
2向右进入6
6向左进入7

(b)搜索过程

回溯法引言

搜索引擎中的网络爬虫：

- 搜索引擎是指根据一定的策略，运用特定的计算机程序从互联网上搜集信息，对信息进行组织和处理后，为用户提供检索服务，将用户检索相关的信息展示给用户的系统
- 搜索引擎的组成：

  下载、索引、查询

回溯法引言

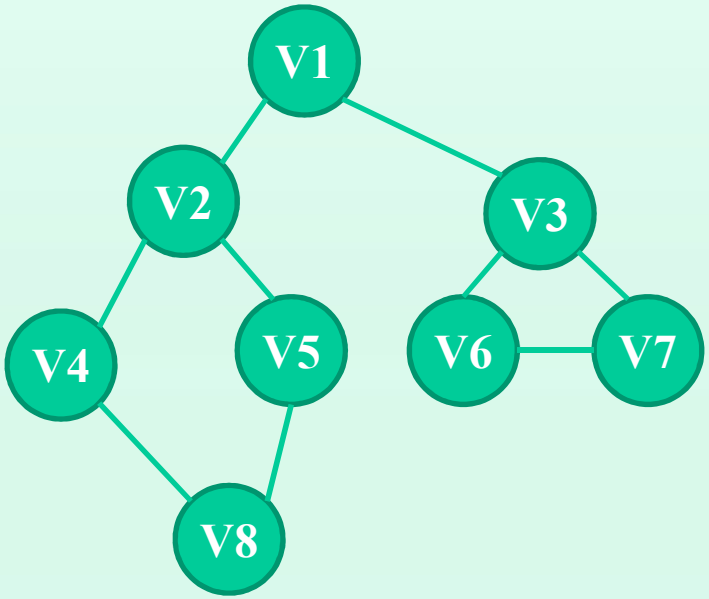
搜索引擎中的网络爬虫：

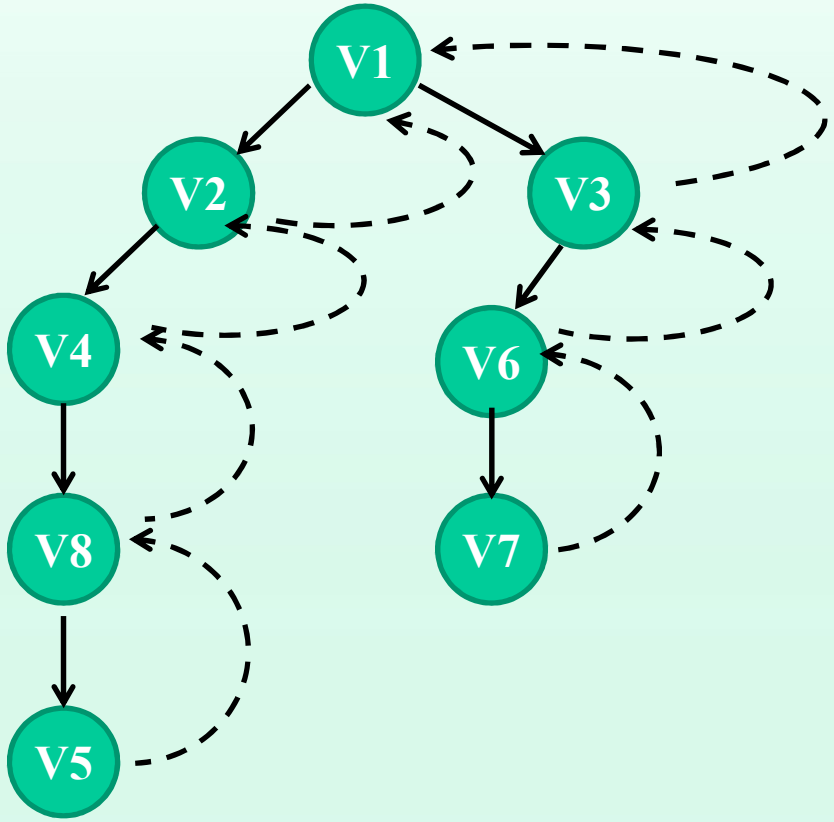
- 网络爬虫：自动下载互联网中的所有网页
- 网络爬虫的原理：

  图的遍历，从图中某一顶点出发遍历图中所有的顶点，且使每个顶点仅被访问一次。

回溯法引言

回溯算法：图的深度优先遍历





5.1 回溯法的算法框架

5.1.1 问题的解空间

- 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。

- 隐约束

为满足问题的解而对不同分量之间施加的约束。

5.1 回溯法的算法框架

解空间（Solution Space）

- 对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

注意：同一问题可有多种表示，有些表示更简单，所需状态空间更小（存储量少，搜索方法简单）。

5.1 回溯法的算法框架

例如

对于有n种可选物品的0-1背包问题

✓ 解空间由2^n个长度为n的0-1向量组成

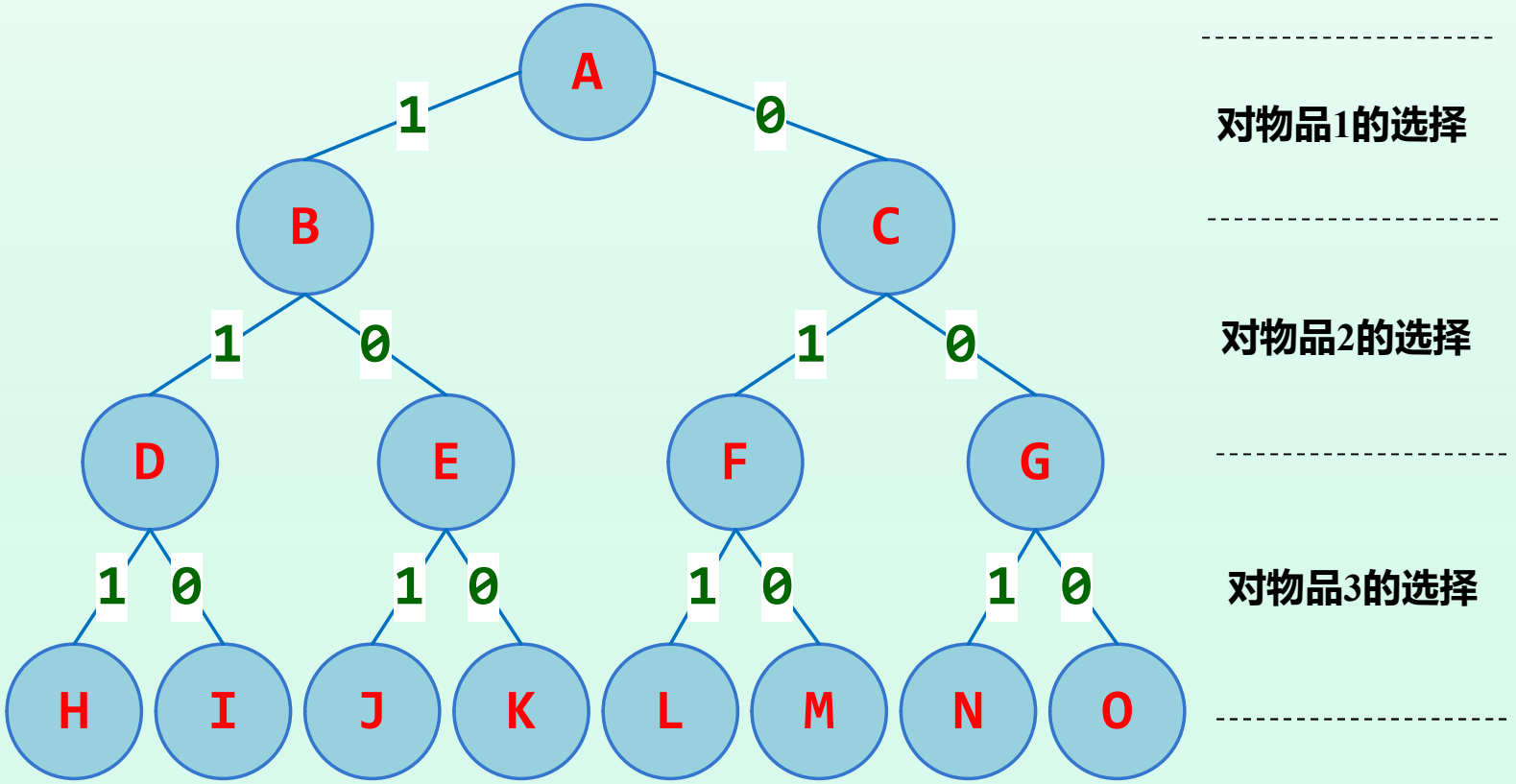
✓ n=3时，解空间为{(0,0,0),(0,0,1),(0,1,0),(0,1,1),
(1,0,0),(1,0,1),(1,1,0),(1,1,1)}

用完全二叉树表示的解空间

✓边上的数字给出了向量x中第i个分量的值x_i

✓根节点到叶节点的路径定义了解问题的一个解

5.1 回溯法的算法框架


A
1
0
B
C
1
0
1
0
D
E
F
G
1
0
1
0
1
0
1
0
H
I
J
K
L
M
N
O
对物品1的选择
对物品2的选择
对物品3的选择

5.1 回溯法的算法框架

5.1.2 回溯法的基本思想

◘回溯法的基本步骤

(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。

◘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。

5.1 回溯法的算法框架

空间复杂性

- 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))。
- 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。

5.1 回溯法的算法框架

生成问题状态的基本方法

- 扩展结点（**E-结点,Expansion Node**）

一个正在产生儿子的结点称为扩展结点

- 活结点（**L-结点,Live Node**）

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（**D-结点,Dead Node**）

一个所有儿子已经产生的结点称做死结点

5.1 回溯法的算法框架

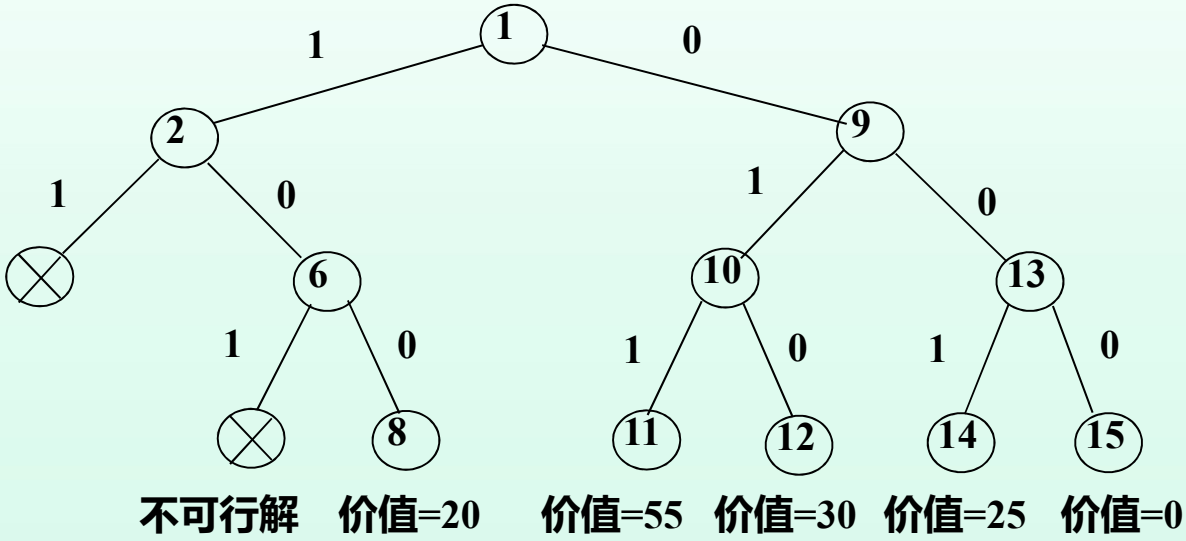
深度优先的问题状态生成法

- 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

- 一个扩展结点变成死结点之前，它一直是扩展结点。

5.1 回溯法的算法框架



当搜索到一个L结点时，就把这个L结点变成为E结点，继续向下搜索这个结点的儿子结点。当搜索到一个D结点，而还未得到问题的最终解时，就向上回溯到它的父亲结点。如果这个父亲结点当前还是E结点，就继续搜索这个父亲结点的另一个儿子结点；如果这个父情结点随着所有儿子结点都已搜索完毕而变成D结点，就沿着这个父结点向上，回溯到它的祖父结点。这个过程继续进行，直到找到满足问题的最终解。

5.1 回溯法的算法框架

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。

5.1 回溯法的算法框架

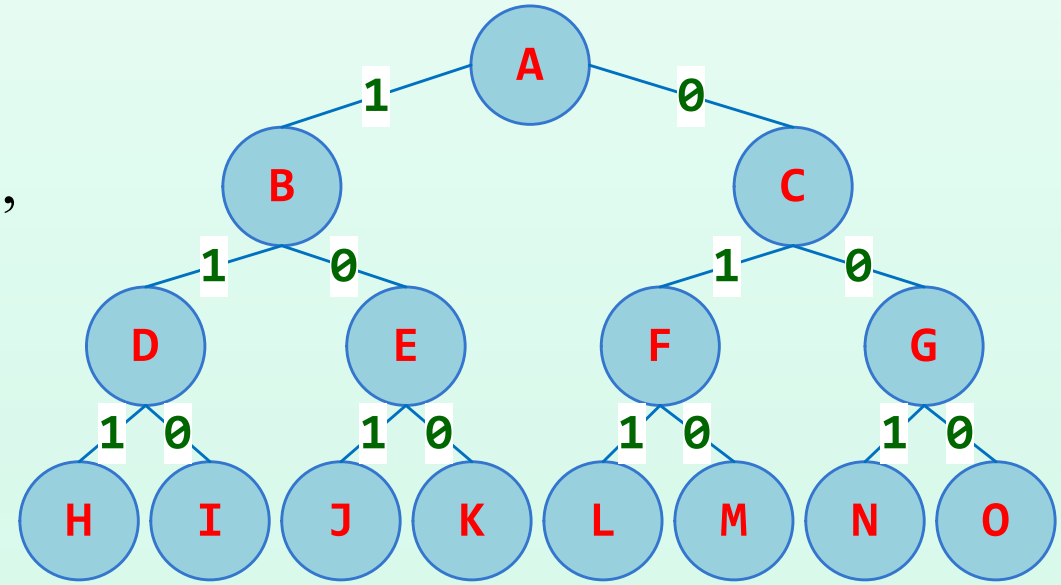
0-1背包问题

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

– **C_r=C=30，V=0**

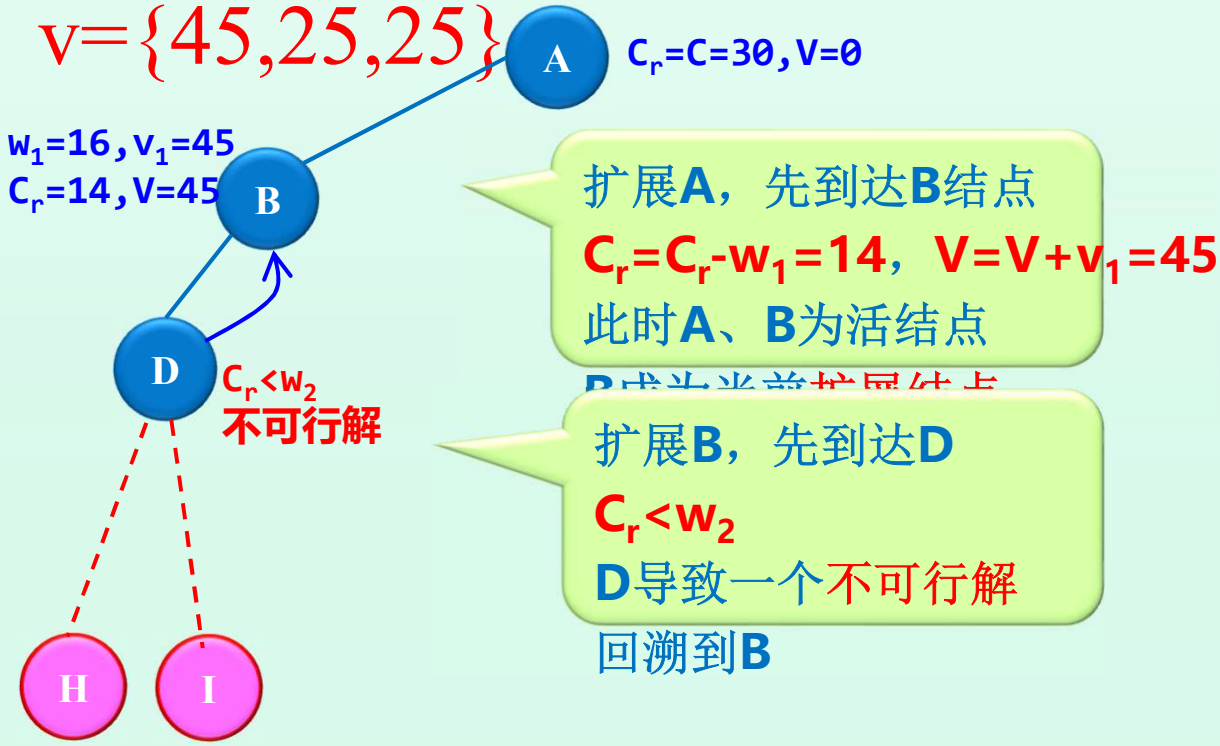
C为容量，C_r为剩余空间，V为价值。

– A为唯一活结点，也是当前扩展结点。



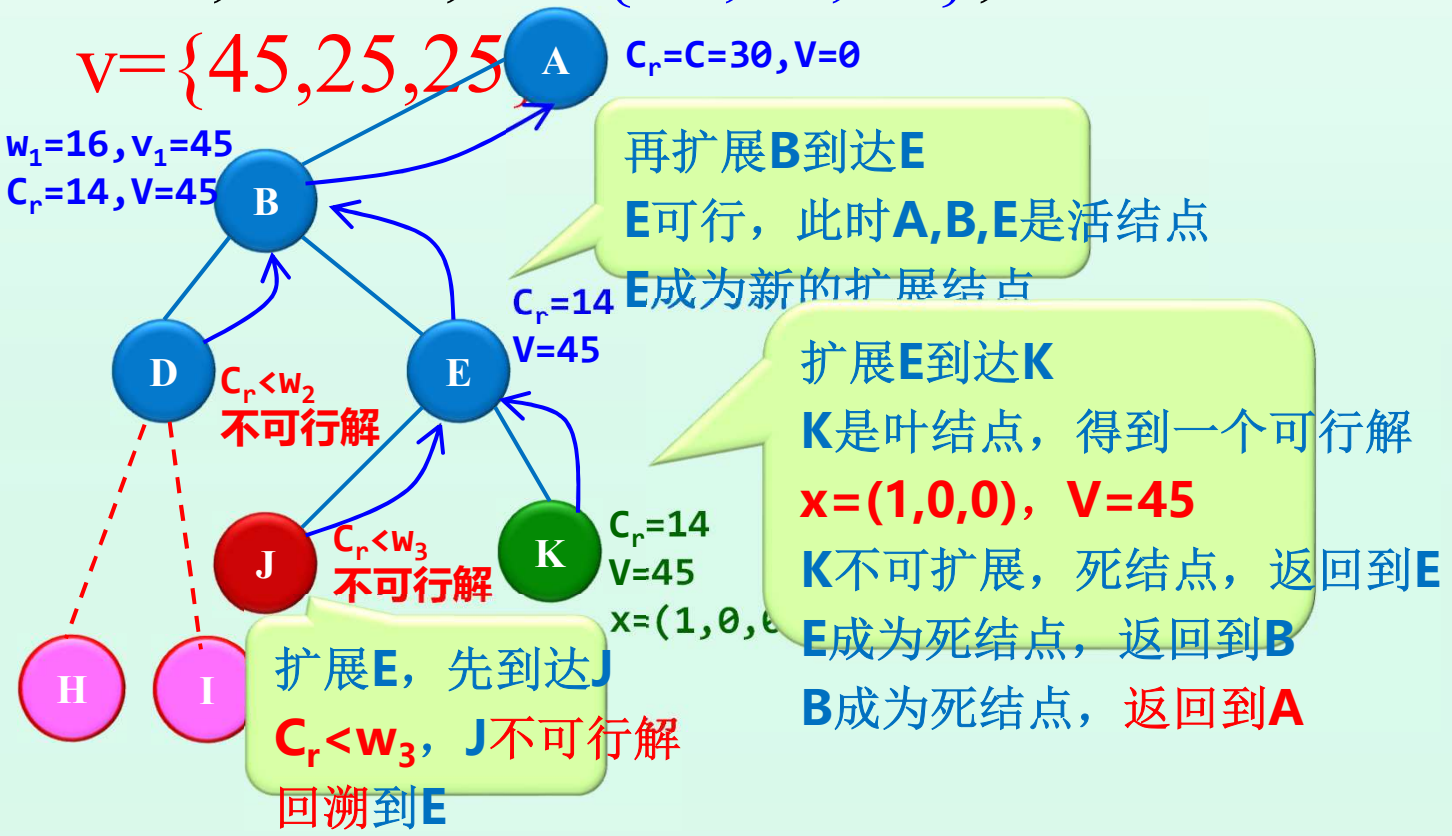
5.1 回溯法的算法框架

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



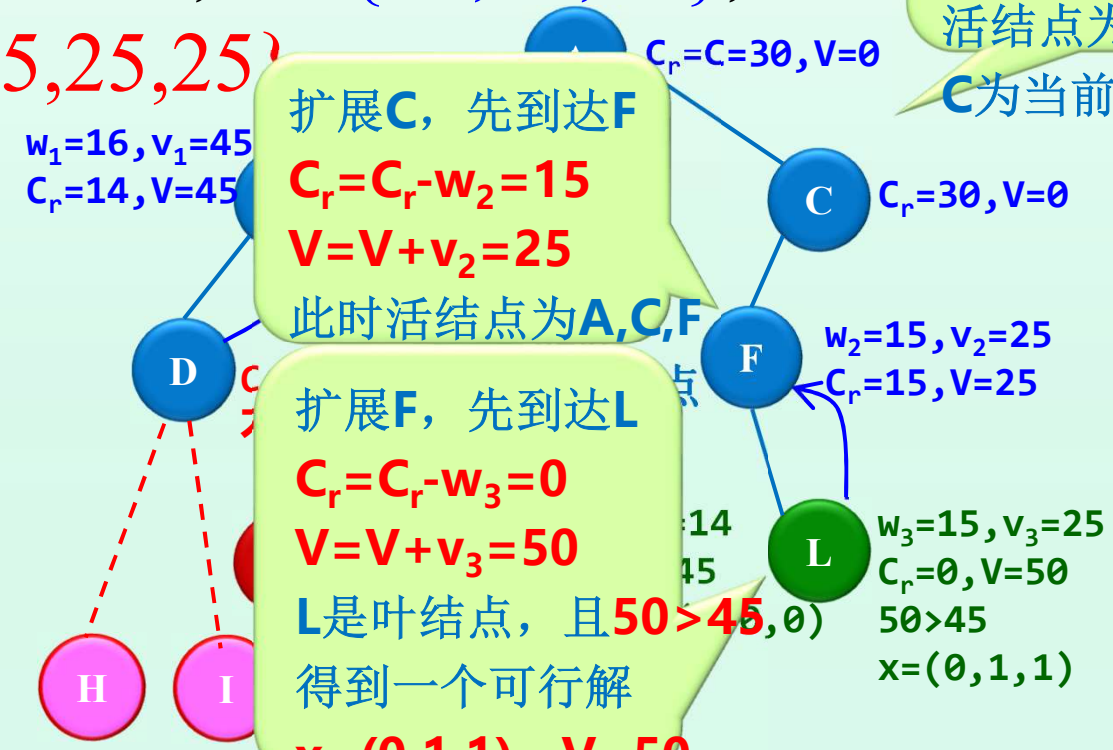
5.1 回溯法的算法框架

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



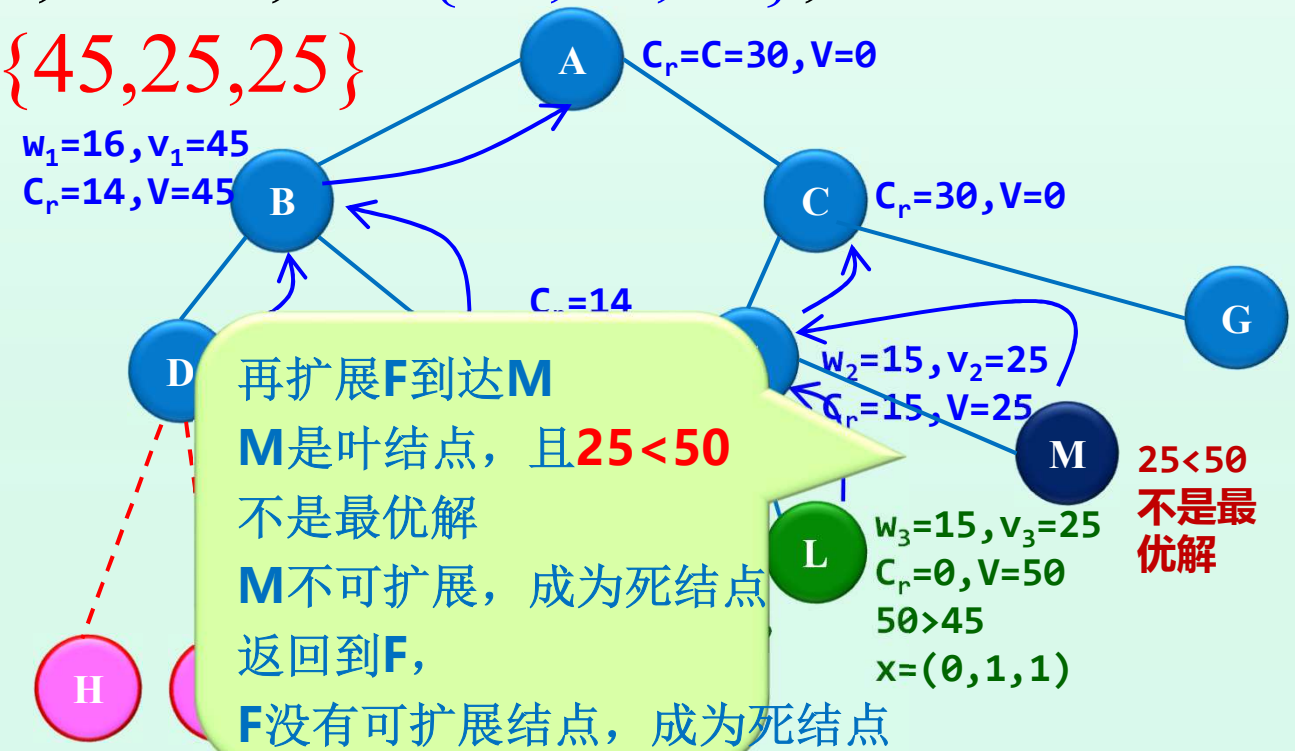
5.1 回溯法的算法框

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



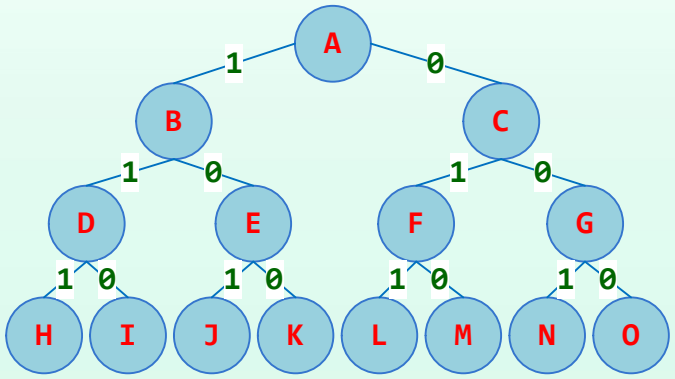
5.1 回溯法的算法框架

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



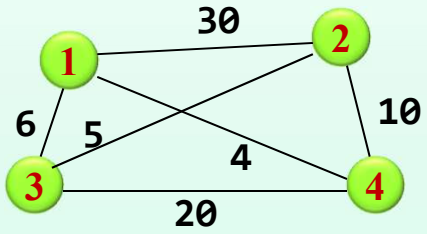
5.1 回溯法的算法框架

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



- 扩展到G以后……
  - C_r=30，V=0，
    活结点为A、C、G，G为当前扩展结点
  - 扩展G，先到达N，N是叶结点，且25<50，不是最优解，又N不可扩展，返回到G
  - 再扩展G到达O，O是叶结点，且0<50，不是最优解，又O不可扩展，返回到G
  - G没有可扩展结点，成为死结点，返回到C
  - C没有可扩展结点，成为死结点，返回到A
  - A没有可扩展结点，成为死结点，算法结束
  - 最优解X=(0,1,1)，最优值50。

5.1 回溯法的算法框架



推销员问题

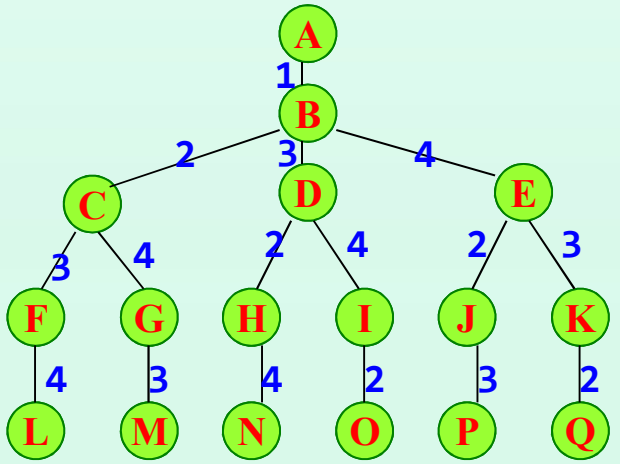
➢ 问题描述

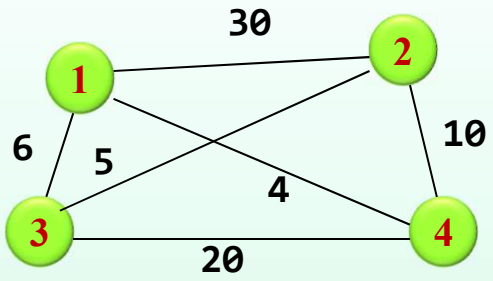
每个城市一遍，最后回到住地的路线，使总的路程最短。

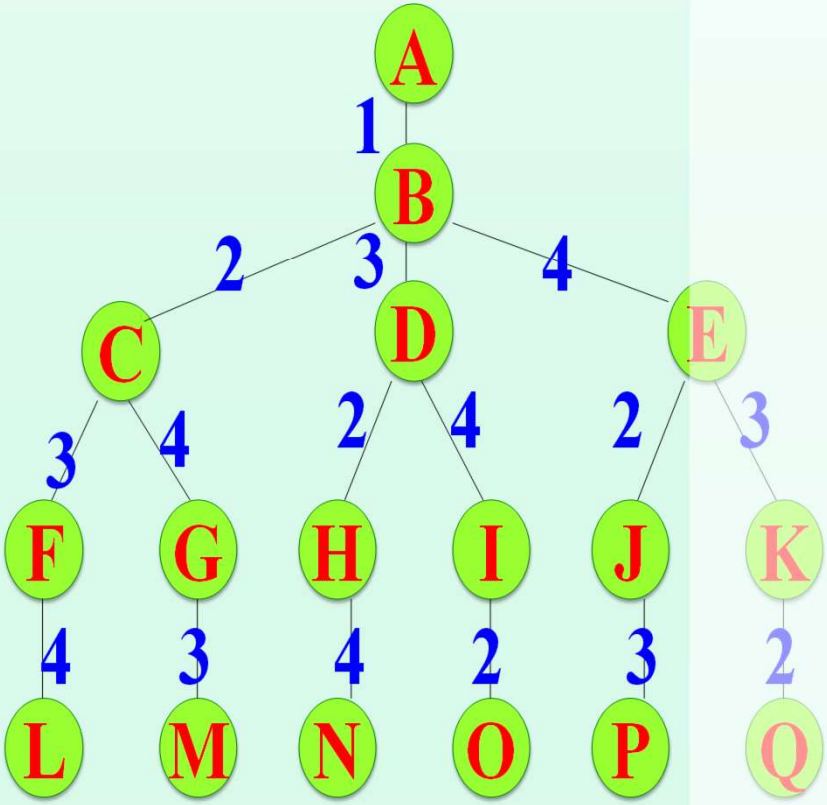
该问题是一个NP完全问题，

有(n-1)!条可选路线

最优解(1,3,2,4,1)，最优值25





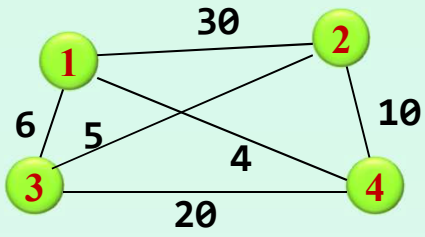


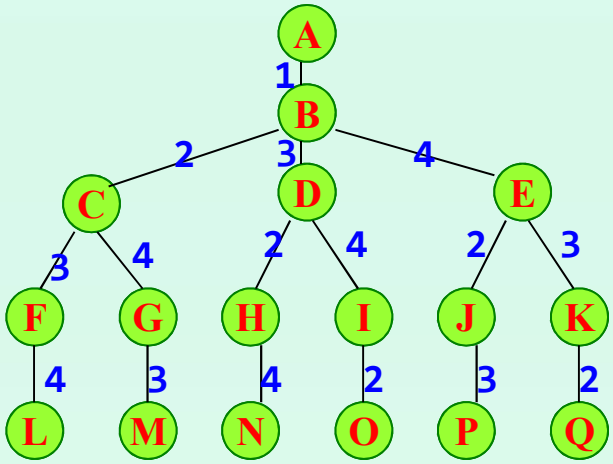
– 回溯算法将用深度优先方式从根节点开始，通过搜索解空间树发现一个最小耗费的旅行。

- 一个可能的搜索为 ABCFL。L点，旅行**1-2-3-4-1**作为当前最好的旅行被记录下来，耗费 59。
- 从L点回溯到活节点F，F没有未被检查的孩子，所以它成为死节点，回溯到 C点。
- C变为E-节点，向前移动到G，然后是M。这样构造出了旅行1-2-4-3-1，它的耗费是66。不比当前的最佳旅行好，抛弃它并回溯到G，然后是C,B。

5.1 回溯法的算法框架

- 从B点，搜索向前移动到D，然后是H,N。这个旅行1-3-2-4-1的耗费是25，比当前的最佳旅行好，把它作为当前的最好旅行。
- 从N点，搜索回溯到H，然后是D。在D点，再次向前移动，到达O点。
- 如此继续下去，可搜索完整个树，得出1-3-2-4-1是最少耗费的旅行，耗费值为25。





5.1 回溯法的算法框架

5.1.3 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，t表示搜索深度。

```
void backtrack (int t){
    if (t>n)            //t>n表示算法已搜索到叶节点
      output(x);    //记录或输出得到的可行解x
    else
        for (int i=f(n,t);i<=g(n,t);i++) {
 //其中f(n,t),g(n,t)分别表示在当前扩展结点处未搜索过的子树的起始编号和终止编号。
            x[t]=h(i);   //h(i)表示在当前扩展节点处x[t]的第i个可选值
            if (constraint(t)&&bound(t))
                backtrack(t+1);
        }
}
```

5.1 回溯法的算法框架

if (Constraint(t)&&Bound(t)) backtrack(t + 1);

if语句含义：Constraint(t)和Bound(t)表示当前扩展节点处的约束函数和限界函数。

- Constraint(t): 返回值为true时，在当前扩展节点处x[1:t]的取值满足问题的约束条件，否则不满足问题的约束条件，可剪去相应的子树
- Bound(t): 返回的值为true时，在当前扩展节点处x[1：t]的取值为目标函数不越界，还需由backtrack(t+1)对其相应的子树做进一步搜索。否则，当前扩展节点处x[1：t]的取值是目标函数越界，可剪去相应的子树
- 递归出口：backtrack(t)执行完毕，返回t-1层继续执行，对还没有测试过的x[t-1]的值继续搜索。当t=1时，若以测试完x[1]的所有可选值，外层调用就全部结束。

5.1 回溯法的算法框架

5.1.4 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);//输出最优解
            else t++;//搜索下一层节点
          }
       }
    else  t--;//回溯到上一节点
  }
}
```

5.1 回溯法的算法框架

5.1.4 迭代回溯

```
if (Constraint(t) &&Bound(t) ) {
        if (Solution(t))      Output(x);
        else t ++;
}
 else t --;
```

分析：

Constraint(t):约束函数，剪枝条件（剪去不可行解）

Bound(t):限界函数，剪枝条件（剪去不可能最优的解）

Solution(t)：判断在当前扩展节点处是否已得到问题的可行解。它返回值为true时，当前扩展节点处x[1：t]是问题的可行解。此时，由Output(x)记录或输出得到的可行解。

它的返回值为false时，在当前扩展结点处x[1:t]只是问题的部分解，还需向纵深方向继续搜索。

搜索边界： f(n,t)和g(n,t)

5.1 回溯法的算法框架

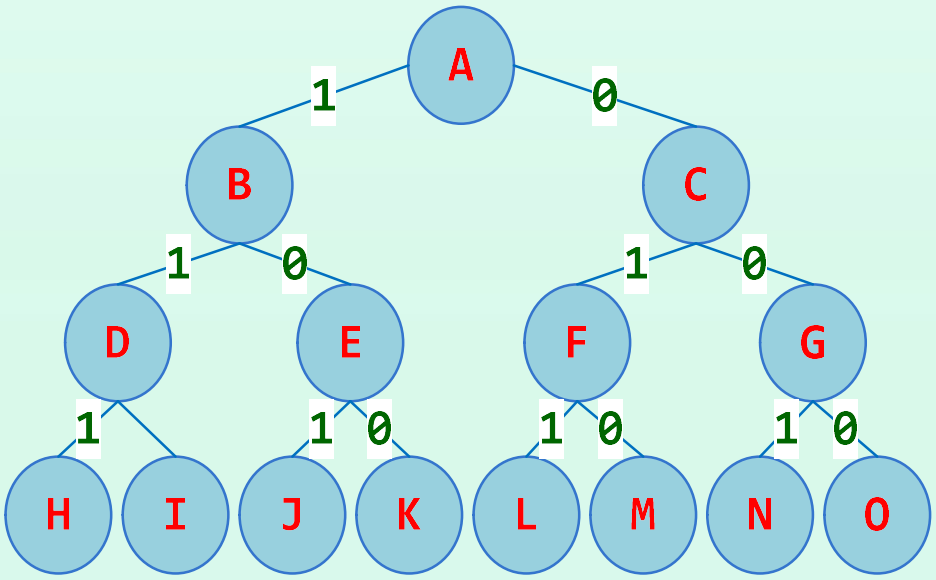
5.1.5 子集树与排列树

- 子集树
  - 从n个元素的集合S中找出S**满足某种性质的子集**，相应的解空间称为子集树。通常有**2^n**个叶结点，结点总数为**$2^{n+1}-1$**。需$\Omega(2^n)$计算时间。
  - 如n个物品的0-1背包问题的解空间是一棵子集树。
- 排列树
  - 当所给问题的确定n个元素**满足某种性质的排列**时，相应的解空间树称为排列树。排列树通常有n!个叶结点。因此遍历排列树需要$\Omega(n!)$计算时间。
  - TSP(旅行售货员问题)的解空间是一棵排列树。

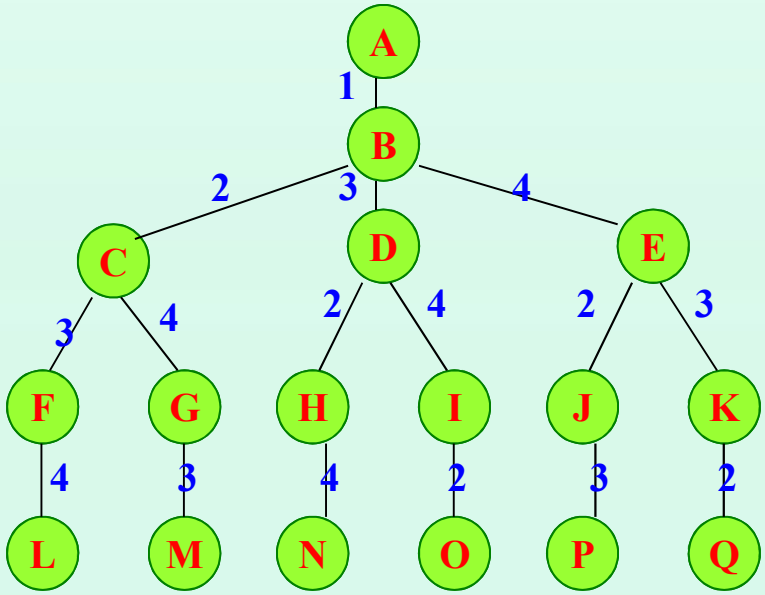
5.1 回溯法的算法框架

5.1.5 子集树与排列树

子集树



排列树



5.2 装载问题

有一批共n个集装箱要装上2艘载重量分别为c_1和c_2的轮船，其中集装箱i的重量为w_i，且

$$\sum_{i=1}^{n} w_i \le c_1 + c_2$$

◆ **装载问题**

- 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。
- 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。

5.2 装载问题

- **转换问题**

- 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第一艘轮船的载重量。
- 由此可知，装载问题等价于以下特殊的0-1背包问题。

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

$$\sum_{i=1}^{n} w_i x_i \le c_1$$

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

用回溯法设计解装载问题的$O(2^n)$计算时间算法。

在某些情况下该算法优于动态规划算法。

5.2 装载问题

- 可行性约束函数：$\sum_{i=1}^{n} w_i x_i \le c_1$

- 在子集树的第j+1层的节点Z处，用cw记当前的装载重量，即$cw=(w_1x_1+w_2x_2+\ldots+w_jx_j)$，当$cw>c_1$时，以节点Z为根的子树中所有节点都不满足约束条件，因而该子树中解均为不可行解，故可将该子树剪去。（该约束函数去除不可行解，得到所有可行解）

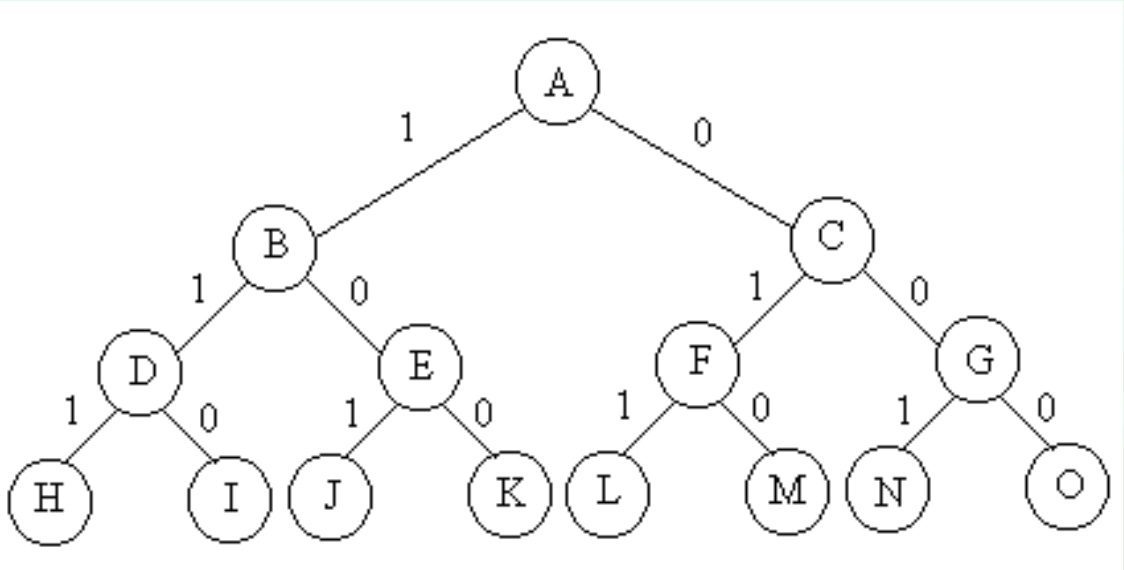
5.2 装载问题

- 上界函数：
  - 设Z是解空间树第i层上的当前扩展结点。cw是当前载重量；bestw是当前最优载重量；r是剩余集装箱的重量，即 $r = \sum_{j=i+1}^{n} w_j$
  - 定义上界函数为cw+r。在以Z为根的子树中任一叶结点所相应的载重量均不超过cw+r。因此，当cw+r<=bestw时，可将z的右子树剪去。

当前载重量cw+剩余集装箱的重量r≤当前最优载重量bestw

5.2 装载问题

- 解空间树



5.2 装载问题

- 剪枝函数
  - **约束条件**剪去”不可行解”的子树
  - **上界条件**剪去不含最优解的子树

保证算法搜索到的每个叶结点都是迄今为止找到的最优解

5.2 装载问题

- 算法设计

- 先考虑装载一艘轮船的情况，依次讨论每个集装箱的装载情况，共分为两种，要么装(1)，要么不装(0)，因此很明显其解空间树可以用子集树来表示。
- 在算法maxLoading中，返回不超过c的最大子集和。
- 在算法maxLoading中，调用递归函数backtrack(1)实现回溯搜索。backtrack(i)搜索子集树中的第i层子树。
- 在算法backtrack中，当i>n时，算法搜索到叶结点，其相应的载重量为cw，如果cw>bestw，则表示当前解优于当前的最优解，此时应该更新bestw。
- 算法backtrack动态地生成问题的解空间树。在每个结点处算法花费O(1)时间。子集树中结点个数为$O(2^n)$，故backtrack所需的时间为$O(2^n)$。另外backtrack还需要额外的O(n)的递归栈空间。

5.3 批处理作业调度

问题描述：

- 给定n个作业的集合$\{J_1,J_2,\ldots,J_n\}$。
- 每一个J_i作业都有两项任务分别在两台机器上完成。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}，其中i=1,2,...n，j=1,2。
- 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 $f=\sum_{i=1}^{n}F_{2i}$ 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。**

5.3 批处理作业调度

问题解析：

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

- 这个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）

- 对应的完成时间和分别是：19,18,20,21,19,19。
- 最佳调度方案是1,3,2，其完成时间和为18。

5.3 批处理作业调度

问题解析：

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

调度方案（1，2，3）的完成时间计算：

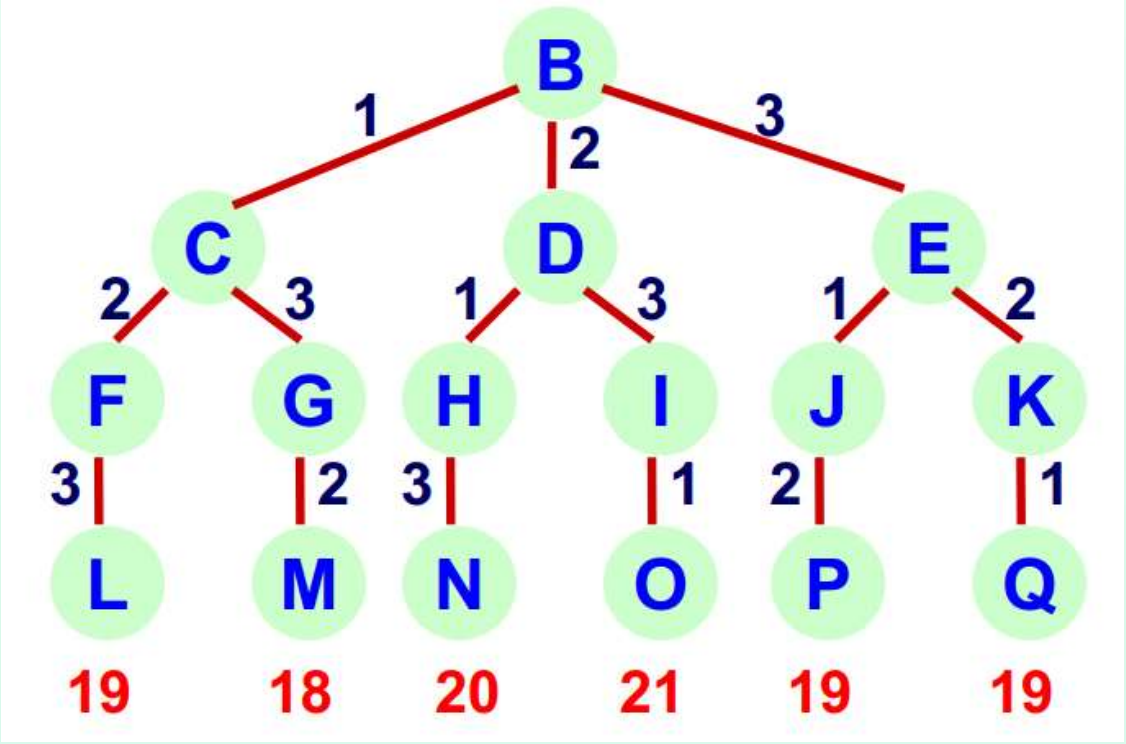
- 作业一在机器1上完成的时间是2，在机器2上完成的时间是3。
- 作业二在机器1上完成的时间是5，在机器2上完成的时间是6。
- 作业三在机器1上完成的时间是7，在机器2上完成的时间是10。

$$f=3+6+10=19$$

5.3 批处理作业调度

算法设计：

批处理作业调度问题要从n个作业的所有排列中找出有最小完成时间和的作业调度，所以批处理作业调度问题的解空间是一颗**排列树**。

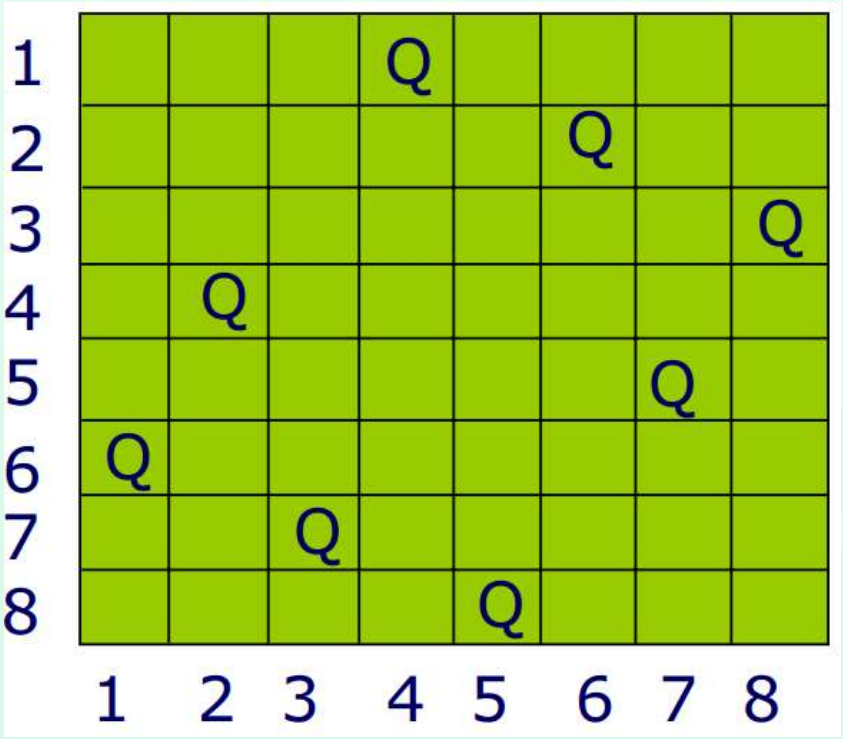


```
private static void backtrack(int i) {
     if (i > n) {
      for (int j = 1; j <= n; j++) bestx[j] = x[j];
      bestf = f;
     }  else
      for (int j = i; j <= n; j++) {
        f1+=m[x[j]][1];
        f2[i]=((f2[i-1]>f1)?f2[i-1]:f1)+m[x[j]][2];
        f+=f2[i];
        if (f < bestf) {
          MyMath.swap(x,i,j);
          backtrack(i+1);
          MyMath.swap(x,i,j);
          }
        f1-=m[x[j]][1];
        f-=f2[i];
        }
 }
```

```
public class FlowShop
     static int  n,      // 作业数
                 f1,     // 机器1完成处理时间
                 f,      // 完成时间和
                 bestf;  // 当前最优值
     static int [][] m;   // 各作业所需的处理时间
     static int [] x;     // 当前作业调度
     static int [] bestx; // 当前最优作业调度
     static int [] f2;    // 机器2完成处理时间
```

5.4 n后问题

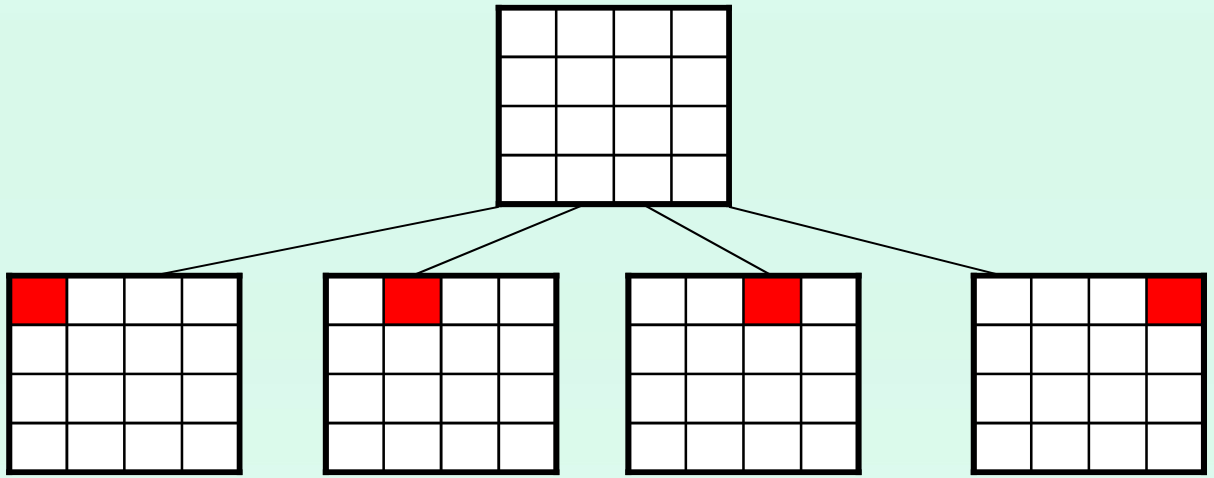
八皇后问题是十九世纪著名的数学家高斯于1850年提出的。问题是：在8×8的棋盘上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上。可以把八皇后问题扩展到**n皇后问题**，即在n×n的棋盘上摆放n个皇后，使任意两个皇后都不能处于**同一行、同一列或同一斜线上**。



5.4 n后问题

- 确定问题状态：问题的状态即棋盘的布局状态。
- 构造状态空间树：状态空间树的根为空棋盘，每个布局的下一步可能布局是该布局结点的子结点。

  –由于可以预知，在每行中有且只有一个皇后，因此可采用逐行布局的方式，即每个布局有*n*个子结点。

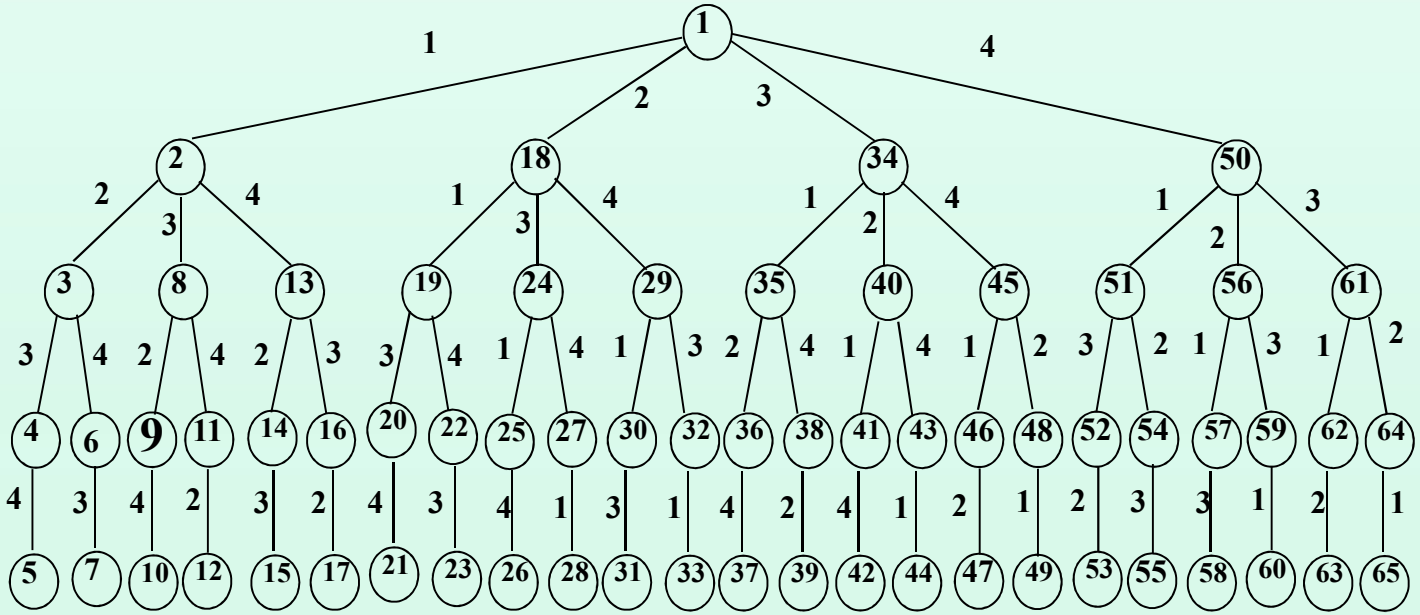


5.4 n后问题

- 设4个皇后为x_i，分别在第i行(i=1，2，3，4)；
- 问题的解状态：可以用$(1,x_1)$，$(2,x_2)$，……，$(4,x_4)$表示4个皇后的位置；
- 由于行号固定，可简单记为：(x_1,x_2,x_3,x_4)；例如：(4, 2, 1,3)
- 问题的解空间：(x_1,x_2,x_3,x_4)，$1\le x_i\le 4(i=1，2，3，4)$，共4！个状态；

5.4 n后问题

4皇后问题解空间的树结构



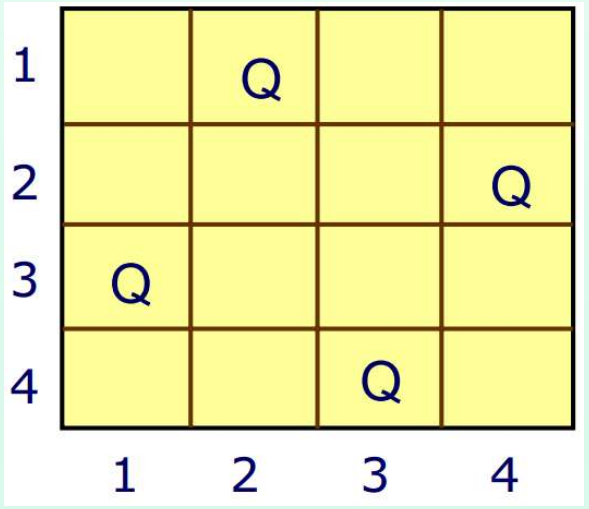
5.4 n后问题

约束条件

- 任意两个皇后不能位于同一行上;
- 任意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：

当$i \neq j$时，$x_i \neq x_j$ （式1）

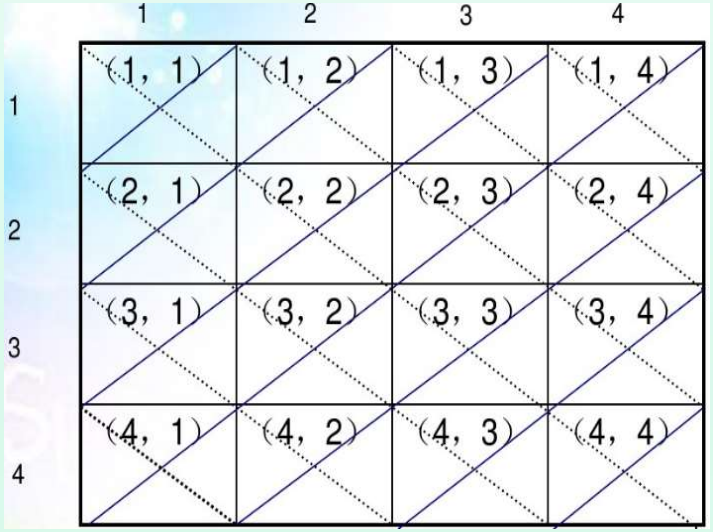


5.4 n后问题

约束条件

- 任意两个皇后不能在同一条斜线上：若两个皇后摆放的位置分别是(i, x_i)和(j, x_j)，若在斜率为-1的斜线上，则有$i - j = x_i - x_j$；若在上斜率为1的斜线上，则有$i + j = x_i + x_j$。综合两种情况，解向量X必须满足约束条件：

$$|i - x_i| \neq |j - x_j| \qquad (式2)$$



原问题即：在解空间中寻找符合约束条件的解状态。

5.4 n后问题

搜索解空间，剪枝

（1）从空棋盘起，逐行放置棋子。

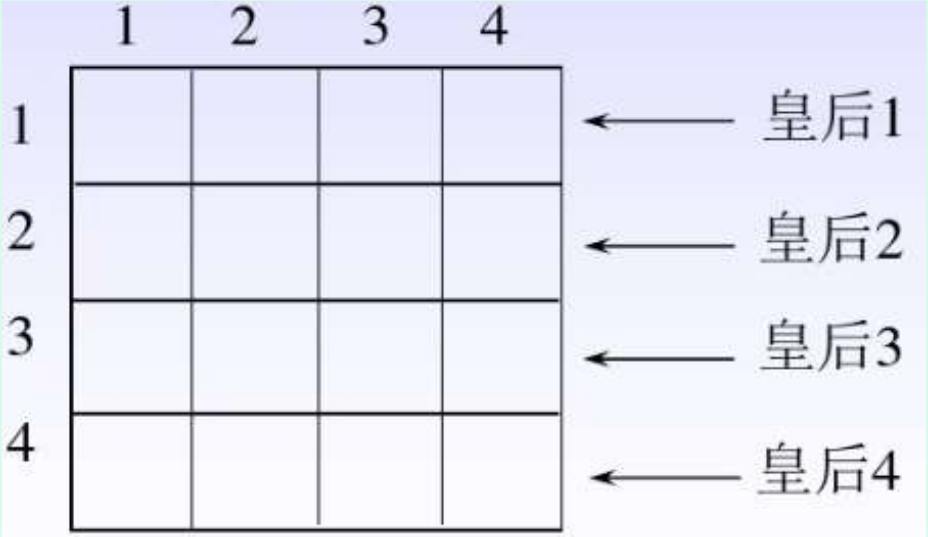
（2）每在一个布局中放下一个棋子，即推演到一个新的布局。

（3）如果当前行上没有可合法放置棋子的位置，则回溯到上一行，重新布放上一行的棋子。

5.4 n后问题

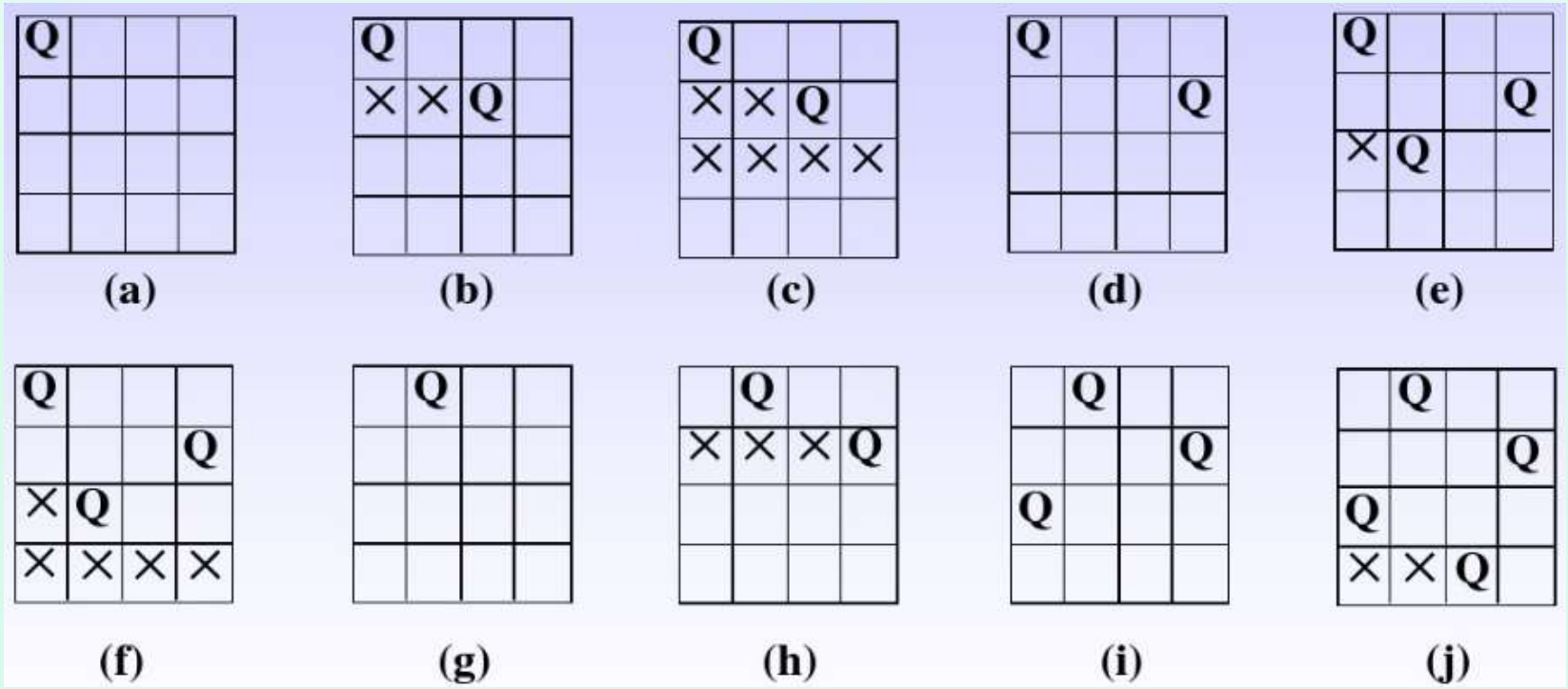
为了简化问题，下面讨论4皇后问题。

4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行可能摆放的位置，从第2层到第3层结点对应皇后2在棋盘中第2行的可能摆放的位置，以此类推。



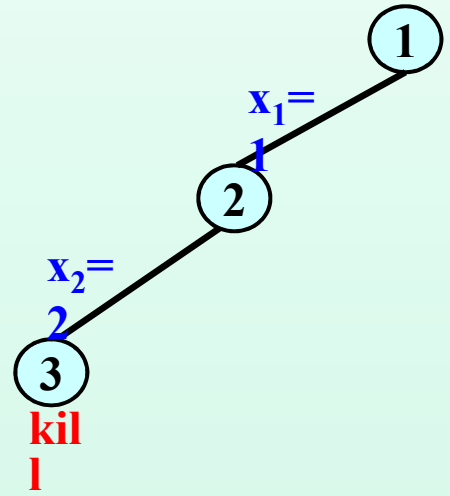
5.4 n后问题

回溯法求解4皇后问题的搜索过程：

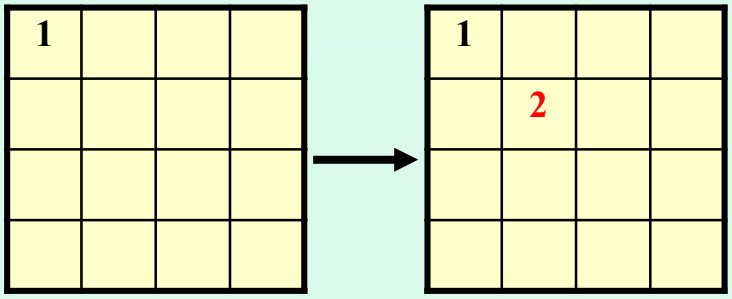


5.4 n后问题

- 开始把根结点1作为唯一的活结点, 根结点就成为E-结点而且路径为( ); 接着生成子结点, 假设按自然数递增的次序来生成子结点, 那么结点2被生成, 这条路径为(1), 即把皇后1放在第1列上。

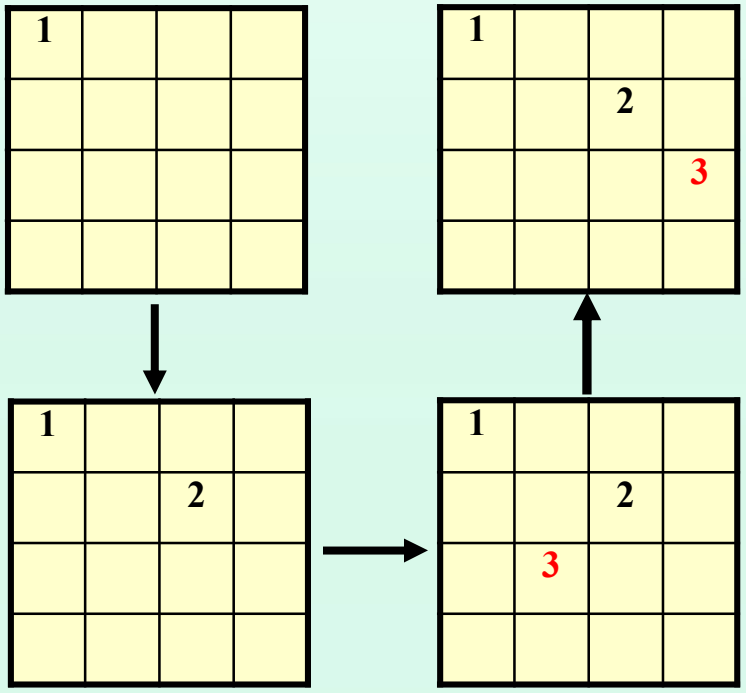


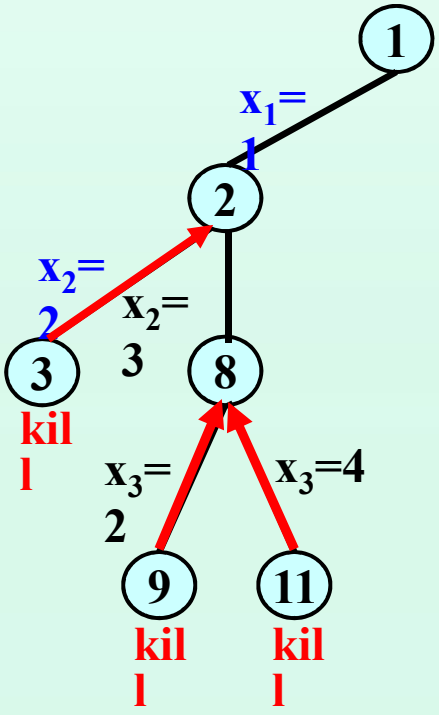
- 结点2变成E-结点, 它再生成结点3, 路径变为(1, 2), 即皇后1在第1列上， 皇后2在第2列上, 所以结点3被杀死, 此时应回溯。



5.4 n后问题

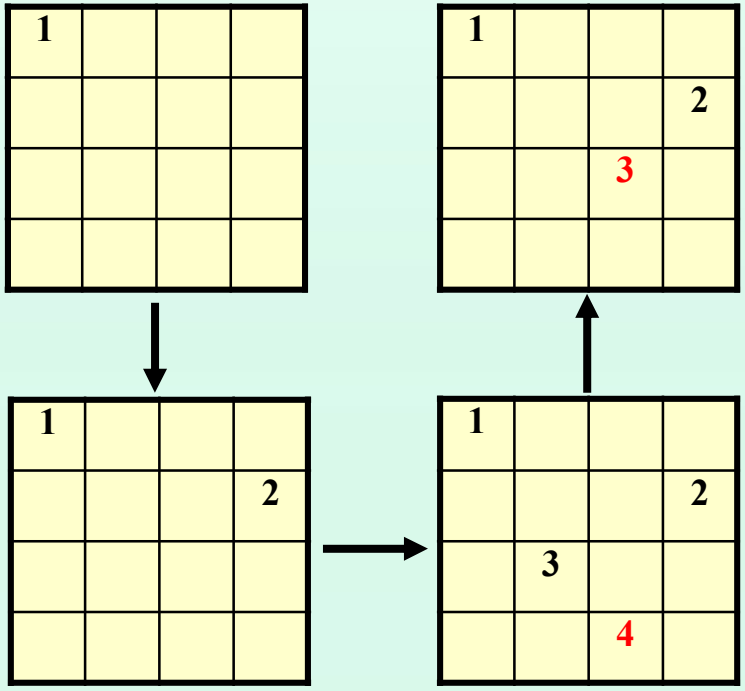
- 回溯到结点2生成结点8, 路径变为(1, 3), 则结点8成为E-结点, 它生成结点9和结点11都会被杀死(即它的儿子表示不可能导致答案的棋盘格局), 所以结点8也被杀死, 应回溯。

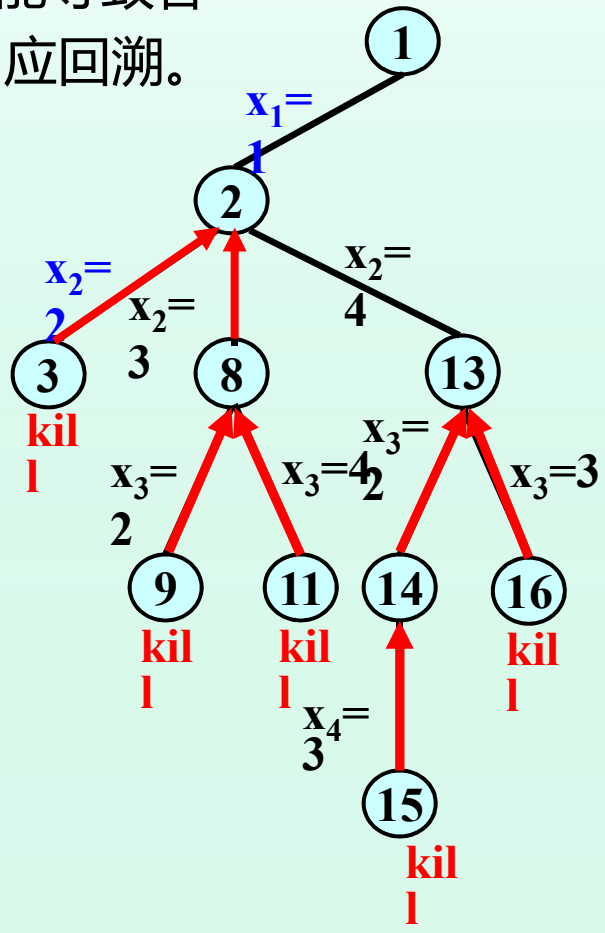




5.4 n后问题

- 回溯到结点2生成结点13, 路径变为(1, 4), 结点13成为E-结点, 由于它的儿子表示的是一些不可能导致答案结点的棋盘格局, 因此结点13也被杀死，应回溯。

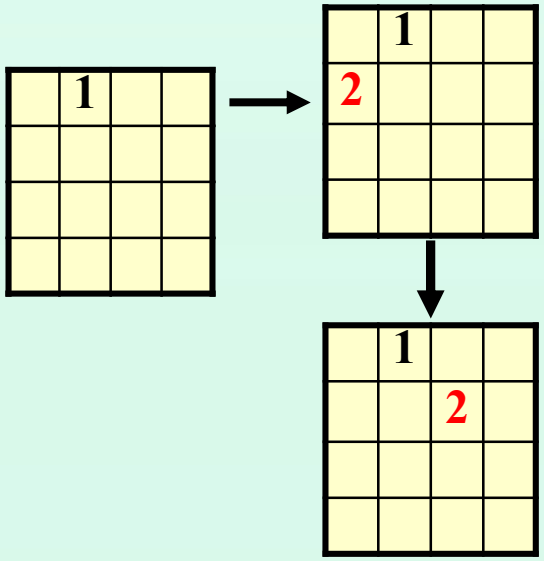


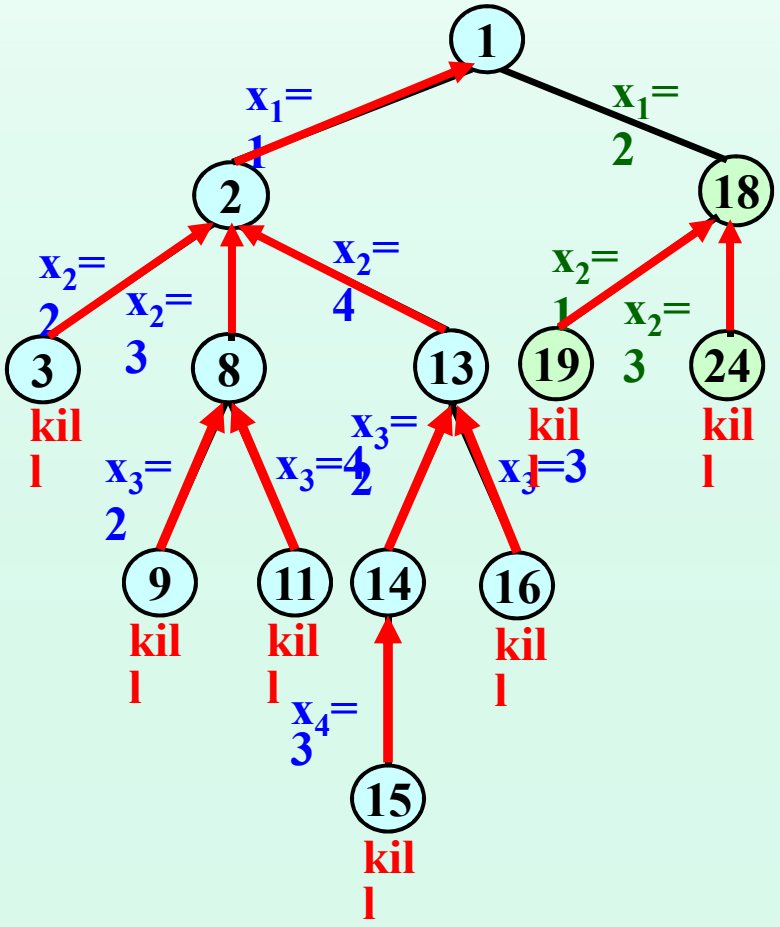


5.4 n后问题

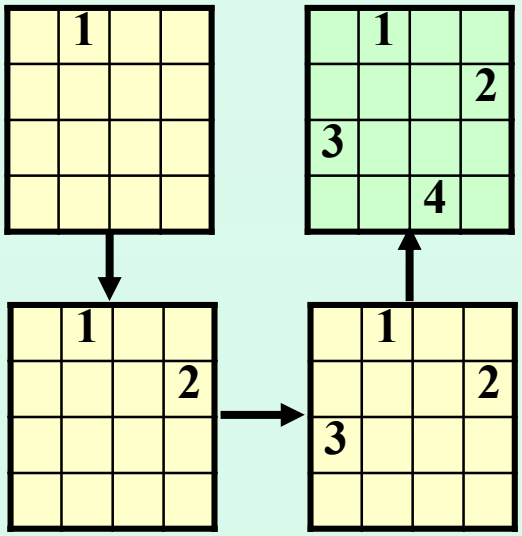
- 结点2的所有儿子表示的都是不可能导致答案的棋盘格局, 因此结点2也被杀死; 再回溯到结点1生成结点18, 路径变为(2)。

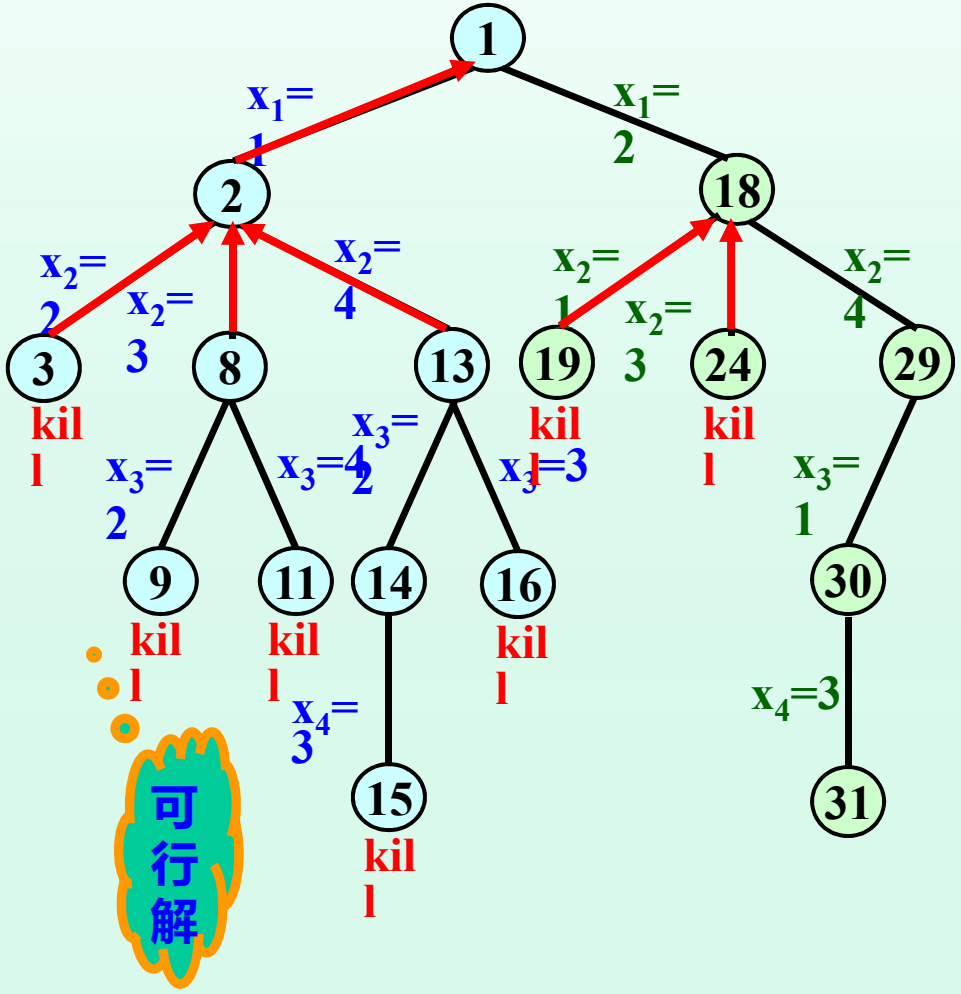
- 结点18的子结点19、结点24被杀死, 应回溯。





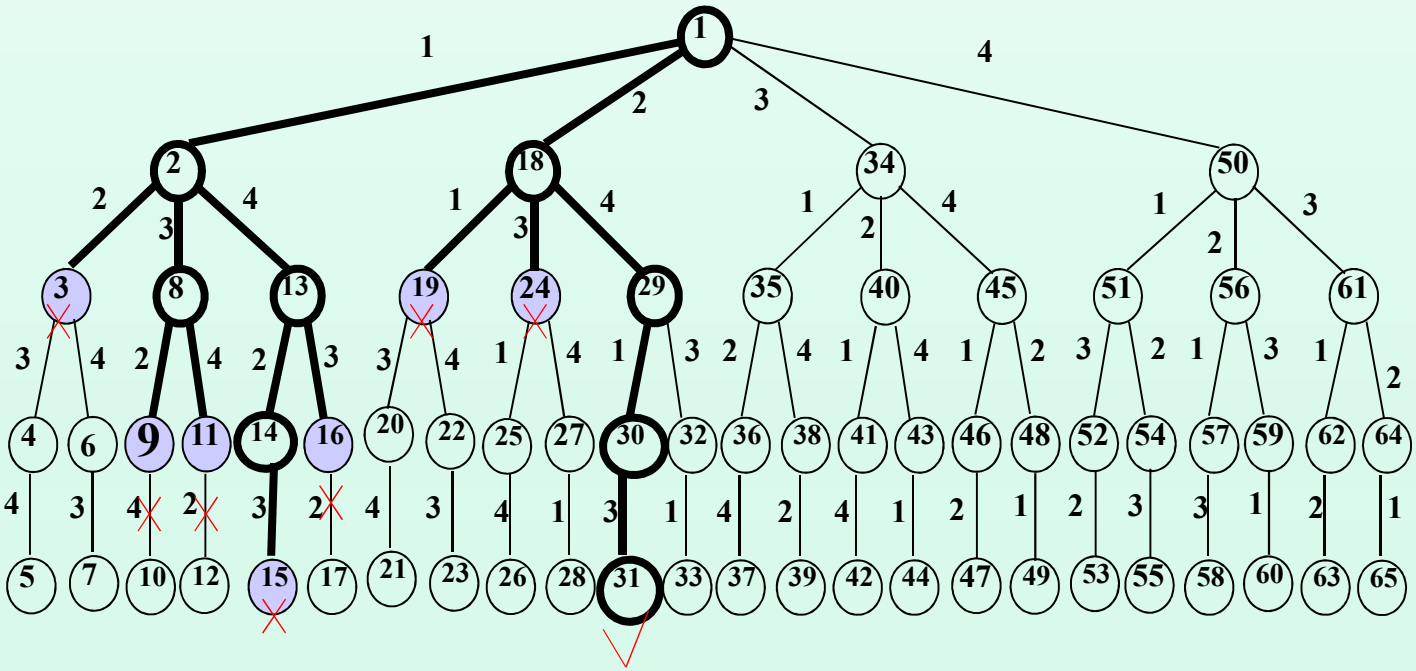
- 结点18生成结点29, 结点29成为E-结点, 路径变为(2,4);
- 结点29生成结点30, 路径变为(2,4,1)
- 结点30生成结点31, 路径变为(2,4,1,3), 找到一个4-王后问题的可行解





5.4 n后问题

n=4的n皇后问题的搜索、剪枝与回溯



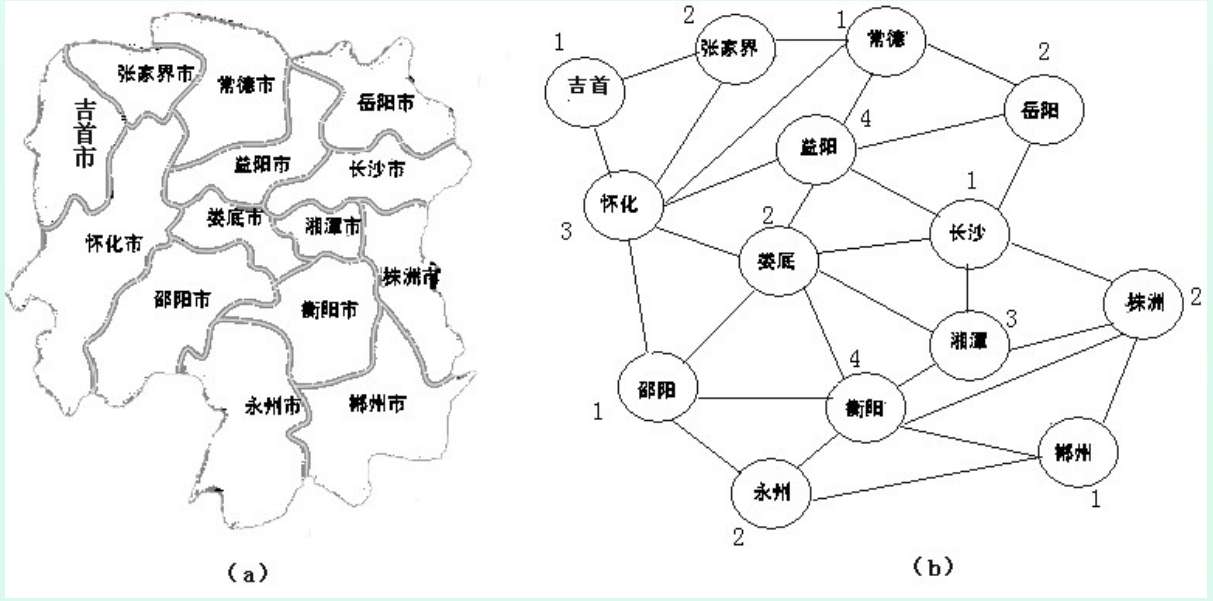
5.5 图的m着色问题

图的着色问题是由地图的着色问题引申而来的：用m种颜色为地图着色，使得地图上的每一个区域着一种颜色，且相邻区域颜色不同。

5.5 图的m着色问题

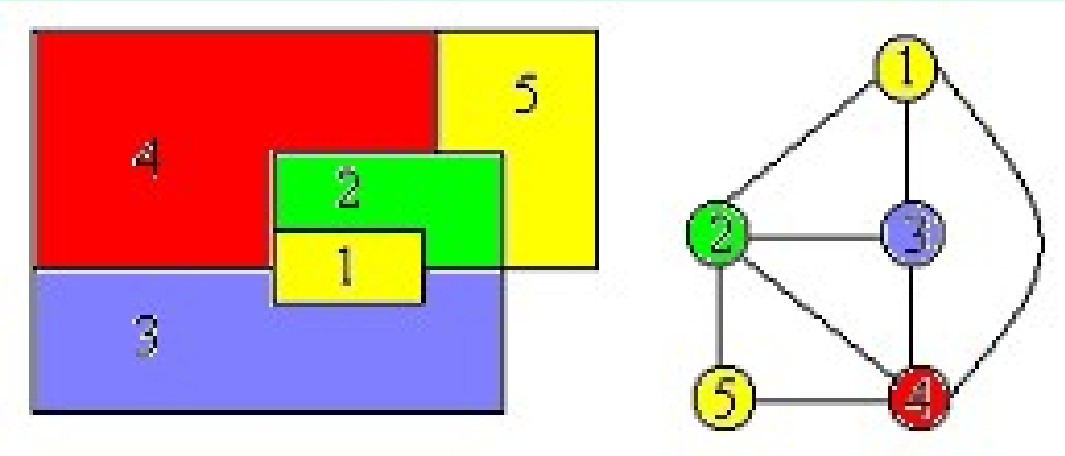
问题处理：如果把每一个区域收缩为一个顶点，把相邻两个区域用一条边相连接，就可以把一个区域图抽象为一个平面图。

例如，图（a）所示的区域图可抽象为（b）所表示的平面图。19世纪50年代，英国学者提出了任何地图都可以4种颜色来着色的4色猜想问题。过了100多年，这个问题才由美国学者在计算机上予以证明，这就是著名的四色定理。例如，在图中，区域用城市名表示，颜色用数字表示，则图中表示了不同区域的不同着色问题 。



5.5 图的m着色问题

- 图着色问题描述为：给定无向连通图G=(V, E)和正整数m，求最小的整数m，使得用m种颜色对G中的顶点着色，使得任意两个相邻顶点着色不同。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。



5.5 图的m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=(c1, c2, …, cn)来描述图的一种可能着色，其中，ci∈{1, 2, …, m}(1≤i≤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。

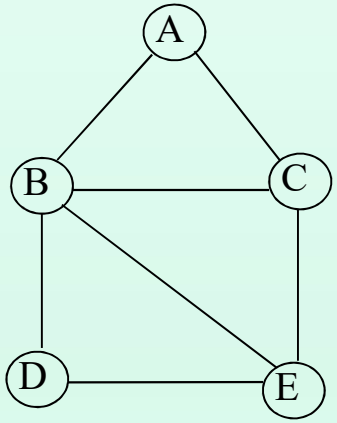
5.5 图的m着色问题

回溯法求解图着色问题：

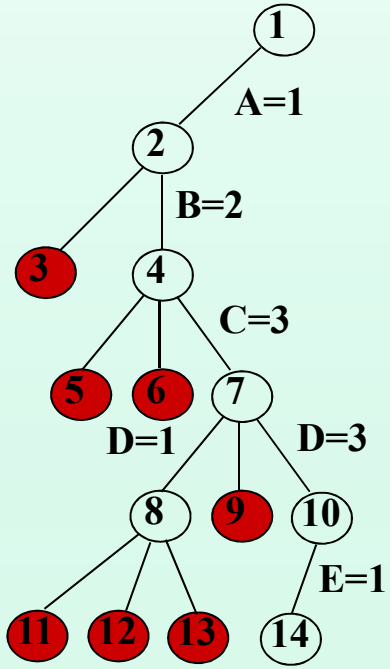
- 首先把所有顶点的颜色初始化为0，然后依次为每个顶点着色。如果其中i个顶点已经着色，并且相邻两个顶点的颜色都不一样，就称当前的着色是有效的局部着色；否则，就称为无效的着色。
- 如果由根节点到当前节点路径上的着色，对应于一个有效着色，并且路径的长度小于n，那么相应的着色是有效的局部着色。这时，就从当前节点出发，继续探索它的儿子节点，并把儿子结点标记为当前结点。在另一方面，如果在相应路径上搜索不到有效的着色，就把当前结点标记为d_结点，并把控制转移去搜索对应于另一种颜色的兄弟结点。
- 如果对所有m个兄弟结点，都搜索不到一种有效的着色，就回溯到它的父亲结点，并把父亲结点标记为d_结点，转移去搜索父亲结点的兄弟结点。这种搜索过程一直进行，直到根结点变为d_结点，或者搜索路径长度等于n，并找到了一个有效的着色为止。

5.5 图的m着色问题

回溯法求解图着色问题示例



(a) 一个无向图



(b) 回溯法搜索空间

5.5 图的m着色问题

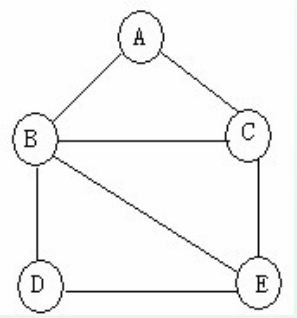
① 把5元组初始化为（0,0,0,0,0），从根结点开始向下搜索，以颜色1为顶点A着色，生成结点2时，产生(1,0,0,0,0)，是个有效着色。

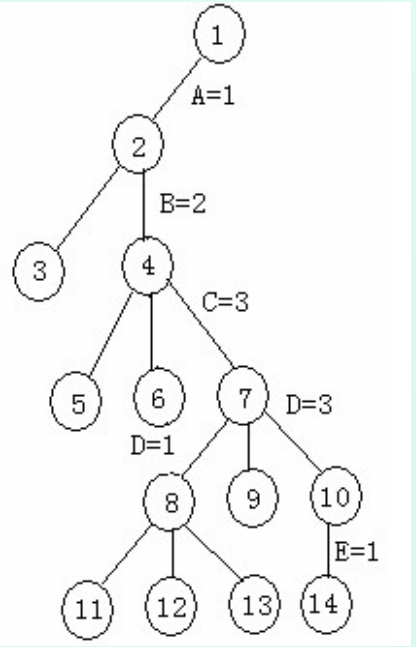
② 以颜色1为顶点B着色生成结点3时，产生(1,1,0,0,0)，是个无效着色，结点3为d_结点。

③ 以颜色2为顶点B着色生成结点4，产生(1,2,0,0,0)，是个有效着色。

④ 分别以颜色1和2为顶点C着色生成结点5和6，产生(1,2,1,0,0)和(1,2,2,0,0)，都是无效着色，因此结点5和6都是d_结点。

⑤ 以颜色3为顶点C着色，产生(1,2,3,0,0)，是个有效着色。重复上述步骤，最后得到有效着色(1,2,3,3,1)。





5.5 图的m着色问题

设数组color[n]表示顶点的着色情况，回溯法求解m着色问题的算法如下：

1．将数组color[n]初始化为0；

2．k=1;

3．while (k>=1)

3.1 依次考察每一种颜色，若顶点k的着色与其他顶点的着色不发生冲突，则转步骤3.2；否则，搜索下一个颜色；

3.2 若顶点已全部着色，则输出数组color[n]，返回；

3.3 否则，

3.3.1 若顶点k是一个合法着色，则k=k+1，转步骤3处理下一个顶点；

3.3.2 否则，重置顶点k的着色情况，k=k-1，转步骤3。

```
GraphColor(int n,int m,int color[],bool c[][5])
{
    int i,k;
   for (i=0; i<n; i++ )          //将解向量color[n]初始化为0
      color[i]=0;
   k=0;
   while (k>=0)
   {   color[k]=color[k]+1;       //使当前颜色数加1
       while ((color[k]<=m) && (!ok(color,k,c,n))) //当前颜色是否有效
                              color[k]=color[k]+1;  //无效，搜索下一个颜色
       if (color[k]<=m)       //求解完毕，输出解
       {
             if (k==n-1)break;  //是最后的顶点，完成搜索
                  else k=k+1;       //否，处理下一个顶点
       }
       else                   //搜索失败，回溯到前一个顶点
       {
              color[k]=0;
               k=k-1;
        }
    }
}
```

5.6 0-1背包问题

在0-1背包问题中，需对容量为c的背包进行装载。从n个物品中选取装入背包的物品，每件物品i的重量为w_i, 价值为p_i。对于可行的背包装载，背包中的物品的总重量不能超过背包的容量，最佳装载是指所装入的物品价值最高

$$\max \sum_{1\le i\le n} p_i x_i$$

约束条件为：

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

在这个表达式中，需求出x_i的值。x_i=1表示物品i装入背包中，x_i=0表示物品i不装入背包。

5.6 0-1背包问题

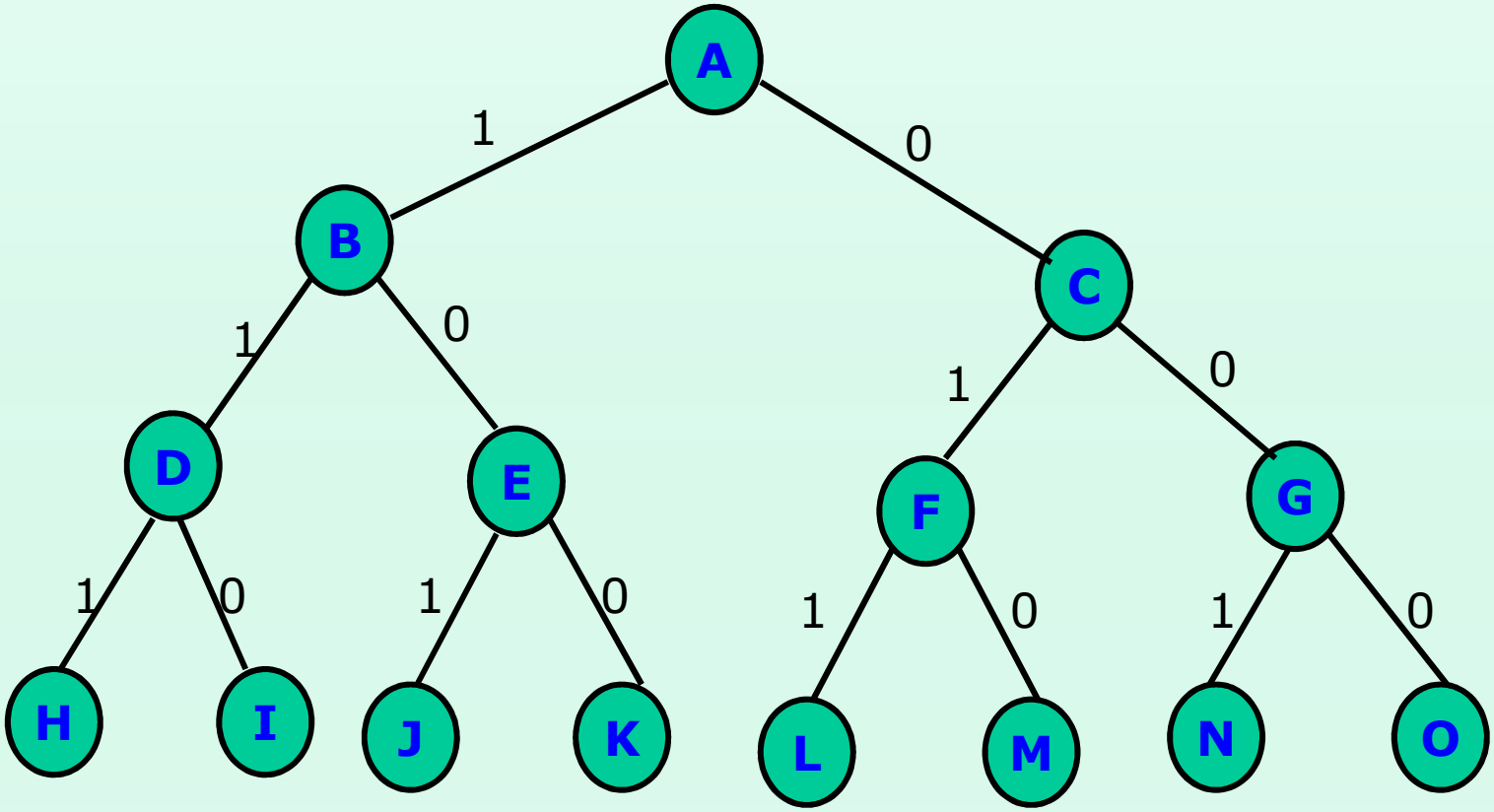
- 0-1背包问题属于找最优解问题，适合于用子集树表示0-1背包问题的解空间。
- 在搜索解空间树时，只要其左儿子节点是一个可行结点，搜索就进入左子树。在右子树中有可能包含最优解是才进入右子树搜索，否则将右子树剪去。
- 设r是当前剩余物品价值总和；cp是当前结点的价值；bestp是当前最优价值。当cp+r≤bestp时，可剪去以当前结点为根的子树。
- 计算右子树中解的上界的更好方法是将剩余物品依其单位重量价值排序，然后依次装入物品，直至装不下时，再装入该物品的一部分而装满背包。由此得到的价值是右子树中解的上界。

5.6 0-1背包问题

对于0-1背包问题回溯法的一个实例，n=4，c=7，p=[9,10,7,4]，w=[3,5,2,1]。这4个物品的单位重量价值分别为[3,2,3.5,4]。以物品为单位价值的递减序装入物品。先装入物品4，然后装入物品3和1。装入这3个物品后，剩余的背包容量为1，只能装入0.2个物品2。由此可得到一个解为x=[1,0.2,1,1]，其相应的价值为22。尽管这不是一个可行解，但可以证明其价值是最大的上界。因此，对于这个实例，最优值不超过22。

5.6 0-1背包问题

对于n=3时的0-1背包问题，其解空间用一棵完全二叉树表示，如下图所示。



5.6 0-1背包问题

0-1背包问题的回溯算法

```
Backtrack(int i)
{ if (i > n)
    { bestp=cp; return;}
  if (cw+w[i]<=c) {//x[i]=1
      cw+=w[i];  cp+=p[i];
      Backtrack(i+1);
      cw-=w[i];  cp-=p[i]; }
if(Bound(i+1)>bestp)
        //x[i]=0
  Backtrack(i+1) ;
}
```

bestp表示当前最优解的装包价值，初值为0。
能走到这儿，意味着得到更优解，因此才改写。

约束条件

cw和cp是全局变量，表示当前路径已装包重量及装包价值，初值均为0 。

限界条件

```
/剩余容量
物品单位重量价值递减序
    装入物品
 cleft)
      i++;
    }
//装满背包
if（i<=n） b+=p[i]/w[i]*cleft;
return b;
}
```

5.6 0-1背包问题

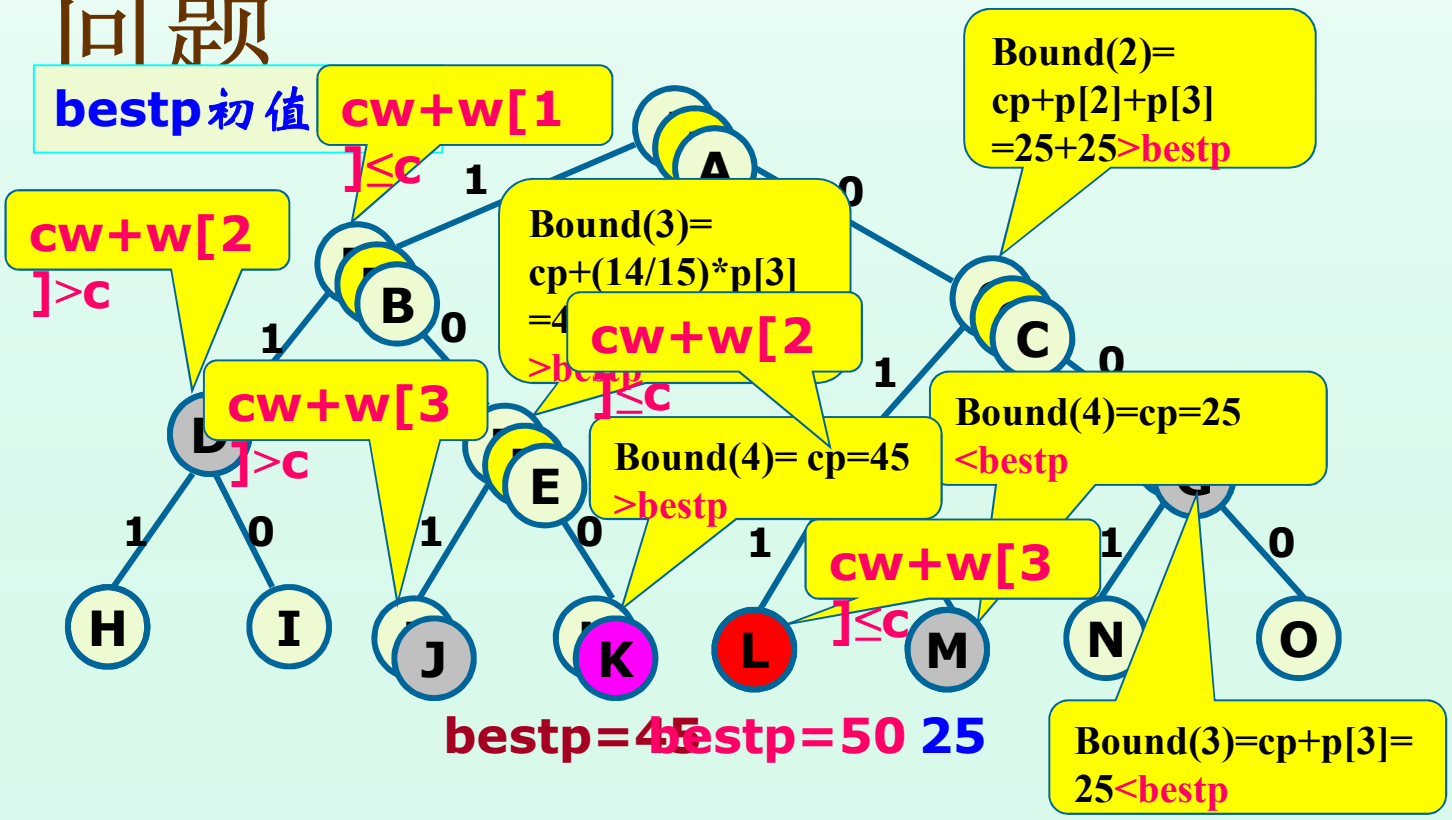
- 算法效率

计算上界函数Bound 需要$O(n)$时间，在最坏情况下有$O(2^n)$个右儿子结点需要计算上界函数，所以解0-1背包问题的回溯算法Backtrack的计算时间为：

$$T(n)=O(n2^n)$$

5.6 0-1背包问题

n=3，W={16，15，15}，p={45，25，25}，C=30。



回溯法效率分析

通过前面具体实例的讨论容易看出，回溯算法的效率在很大程度上依赖于以下因素：

(1)产生x[k]的时间；

(2)满足显约束的x[k]值的个数；

(3)计算约束函数**constraint**的时间；

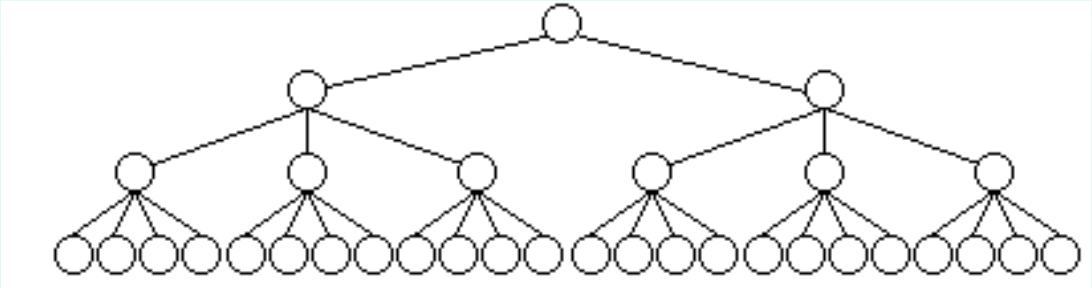
(4)计算上界函数**bound**的时间；

(5)满足约束函数和上界函数约束的所有x[k]的个数。

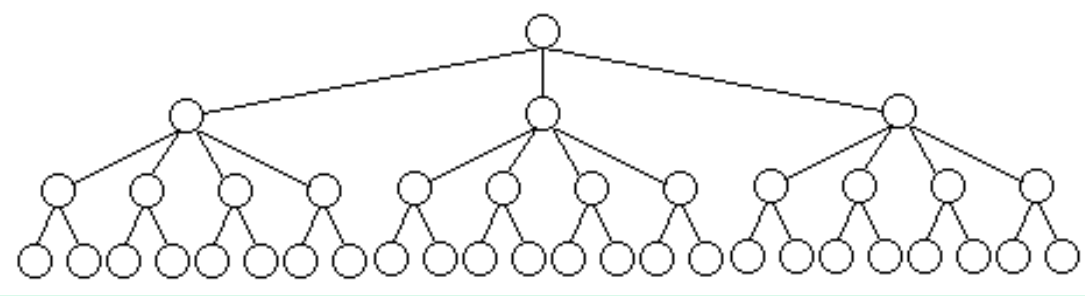
好的约束函数能显著地减少所生成的结点数。但这样的约束函数往往计算量较大。因此，在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷。

重排原理

对于许多问题而言，在搜索试探时选取x[i]的值顺序是任意的。**在其他条件相当的前提下，让可取值最少的x[i]优先**。从图中关于同一问题的2棵不同解空间树，可以体会到这种策略的潜力。



(a)



(b)

图(a)中，从第1层剪去1棵子树，则从所有应当考虑的3元组中一次消去12个3元组。对于图(b)，虽然同样从第1层剪去1棵子树，却只从应当考虑的3元组中消去8个3元组。前者的效果明显比后者好。