第七章　回溯

7.1　回溯法的思想方法

7.2　n 后问题

7.3　图的着色问题

7.4　哈密尔顿回路问题

7.5　背包问题

7.6　回溯法的效率分析

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。

- 于是
  - 回溯和分支限界法是比较常用的对候选解进行系统检查的两种方法。
  - 在系统地检查解空间的过程中，抛弃那些不可能导致合法解的候选解，从而使求解时间大大缩短（无论对于最坏情况还是对于一般情况）。
  - 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。

7.1 回溯法的思想方法

7.1.1 问题的解空间和状态空间树

7.1.2 状态空间树的动态搜索

7.1.3 回溯法的一般性描述

回溯法

- 回溯法是一个既带有系统性又带有跳跃性的搜索算法；
- 它在包含问题的所有解的解空间树中，按照深度优先的策略，从根结点出发搜索解空间树。——系统性

- 算法搜索至解空间树的任一结点时，判断该结点为根的子树是否包含问题的解，如果肯定不包含，则跳过以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，继续深度优先的策略进行搜索。——跳跃性
- 这种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，它适用于解一些组合数较大的问题。

- 回溯法的基本思想：“试探着走”。如果试得不成功退回一步，再换一个方法试。反复进行这种试探性选择与返回纠错过程，直到求出问题的解为止。
- 回溯是穷举法的一个改进，它在所有可行的选择中，系统地搜索问题的解。它假定解可以由向量形式$(x_1,x_2,\ldots,x_n)$来表示，其中x_i的值表示在第i次选择所作的决策值，并以深度优先的方式遍历向量空间，逐步确定x_i的值，直到解被找到。回溯法列举出解空间中所有可能的情形来确保解的正确性。

7.1.1 问题的解空间和状态空间树

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元组$X = (x_1, x_2, \ldots, x_n)$的形式
- 显式约束：
  - 限定每个x_i只从一个给定的集合S_i上取值
- 隐式约束：
  - x_i必须彼此相关，例如0/1背包问题中的背包重量

解空间(Solution Space)

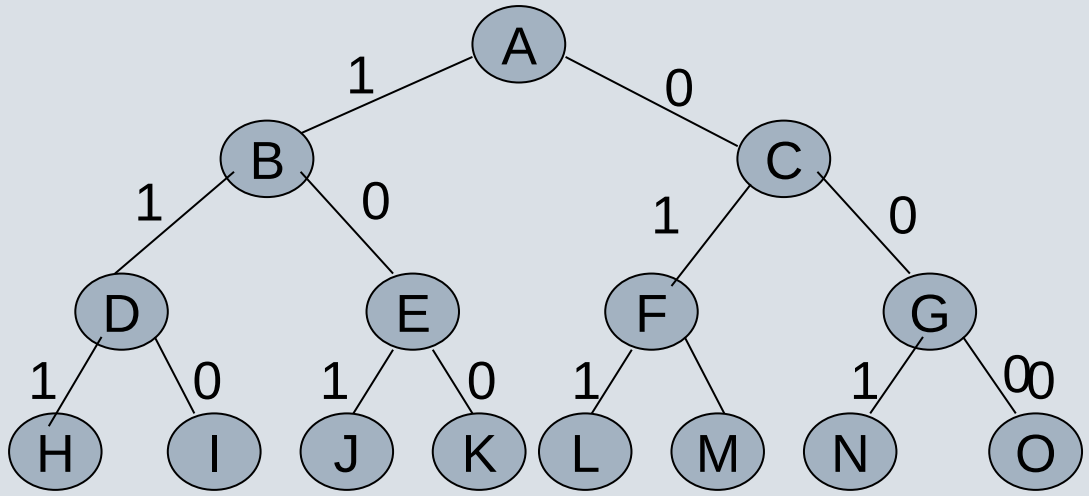
- 问题的解可以用向量$X = (x_1, x_2, \ldots, x_n)$表示，$x_i$的取值范围为有穷集$S_i$；
- 这些解必须使得某一规范函数$P(x_1, x_2, \ldots, x_n)$（也称限界函数）取得极值或满足该规范函数条件；
- 把x_i的所有可能取值组合，称为问题的解空间；
- 每一个组合是问题的一个可能解。

✓ 例：

- **0/1背包问题**：有n种可选物品，S = { 0 , 1 }，其解空间由2^n个长度为n的0/1向量组成；
- n = 3 时，0/1背包问题的解空间是：

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

- 可用完全二叉树表示解空间



✓ 例：货郎担问题，S = { 1, 2, …, n }，当 n = 3 时，S = { 1, 2, 3 }

货郎担问题的解空间是：

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

当输入规模为 n 时，它有n^n种可能的解。

- 考虑到约束方程$x_i \neq x_j$。因此，货郎担问题的解空间压缩为：

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

当输入规模为n时，它有$n!$ 种可能的解

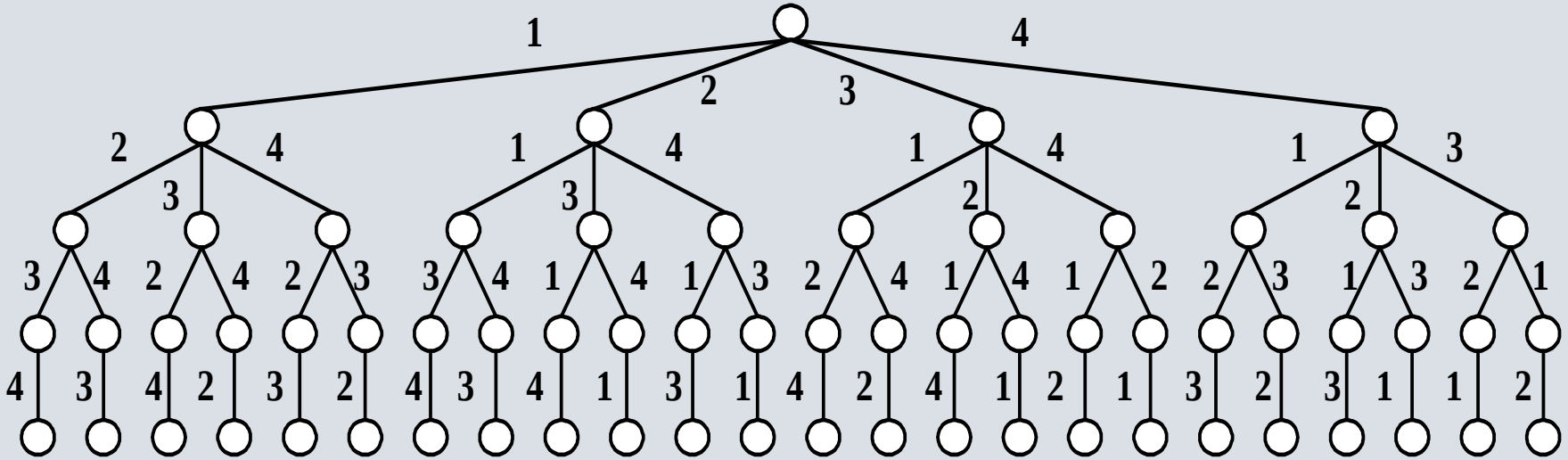
2. 状态空间树

用树的表示形式，把问题的解空间表示出来：

- 在各种情况下变量可能的取值状态；
- 由根结点到叶子结点路径上的标号，构成了问题一个可能的解。

✓ 状态空间树

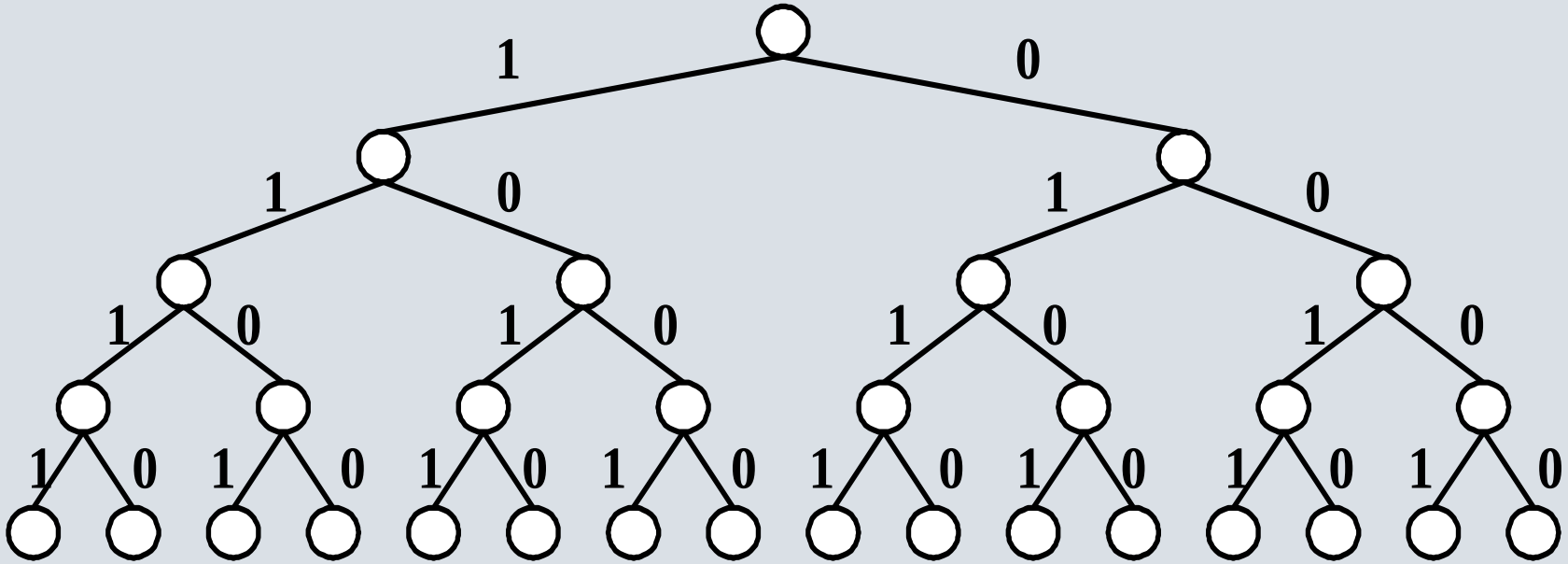
- 例：$n = 4$ 时，货郎担问题的状态空间树



排列树，解空间大小: $n!$

- ✓ 当问题是求n个元素一个排列以使问题最优化时，解空间常可以组织成一棵排列树。

- $n = 4$ 时，0/1背包问题的状态空间树：



子集树，解空间大小: 2^n

- ✓ 当要解决的问题是要求一个使问题最优的n个元素的子集，问题的解空间树可以组织成一棵子集树。

7.1.2 状态空间树的动态搜索

1. 可行解和最优解

可行解：满足约束条件的解，解空间中的一个子集；

最优解：使目标函数取极值（极大或极小）的可行解， 一个或少数几个。

- ✓ 货郎担问题：有n^n种可能解。 $n!$种可行解，只有一个或几个解是最优解。
- ✓ 背包问题：有 2^n种可能解，有些是可行解，只有一个或几个是最优解。
- ✓ 有些问题，只要可行解，不需要最优解，例如皇后问题和图的着色问题。

2. 状态空间树的动态搜索

沿着状态空间树，搜索满足约束条件且使目标函数取极值的最优解。

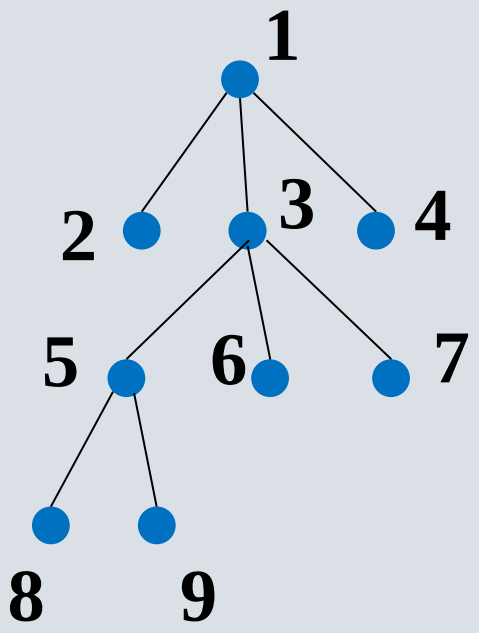
l _结点 （活结点）	所搜索到的结点不是叶结点 满足约束条件和目标函数的界 其儿子结点还未全部搜索完毕
e _结点 （扩展结点）	当前正在搜索其儿子结点的结点 它也是一个 *l*_结点
d _结点 （死结点）	不满足约束条件或目标函数的结点 儿子结点已全部搜索完毕的结点 叶结点

以 *d*_结点作为根的子树，可以在搜索过程中删除

生成问题状态的基本方法

- 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）；
- 广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来去除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。

深度与广度优先搜索



深度优先访问顺序：

1→2 →3 →5 →8 →9
→6 →7 →4

广度优先访问顺序：

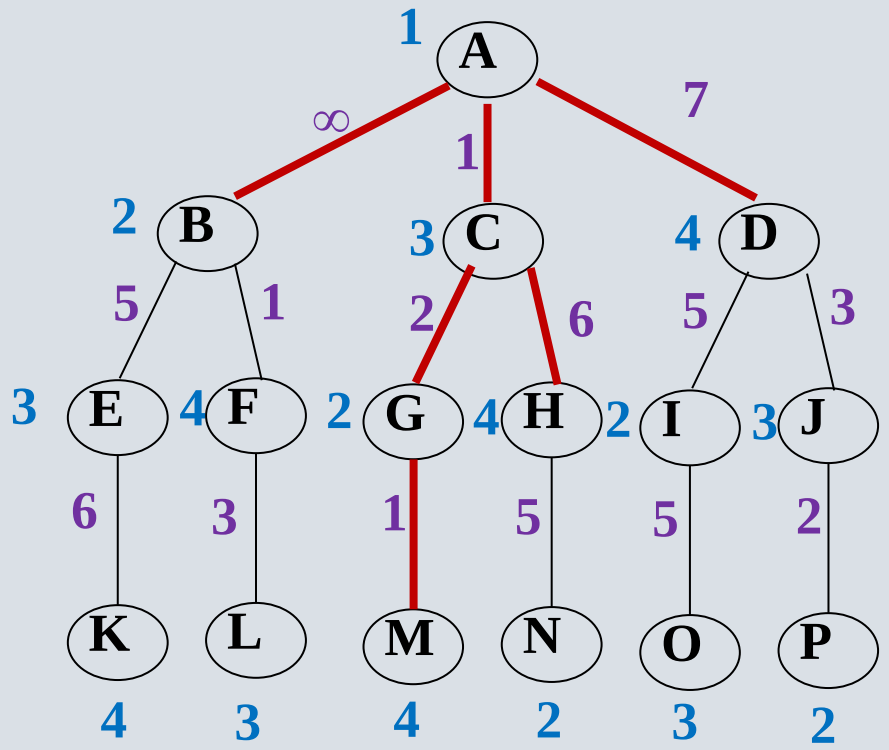
1→2 →3 →4 →5 →6
→7 →8 →9

- 回溯法是一种按深度优先策略，从根结点出发搜索解空间树的穷举式搜索法。
- 为了提高搜索效率，则需要按照某种策略，能避免不必要搜索过程：
  - 放弃当前候选解（放弃搜索该候选解为根的子树），寻找下一个候选解（搜索该候选解兄弟为根的子树）的过程称为回溯；
  - 扩大当前候选解的规模（继续向下搜索该候选解的子孙），以继续试探的过程称为向前试探。

3. 例子

4个顶点的货郎担问题，求从顶点1出发，最后回到顶点1的最短距离。

∞	∞	1	7
8	∞	5	1
7	2	∞	6
2	5	3	∞



状态空间树 V.S. 搜索树

目标函数的下界初始化为∞，从节点A开始搜索。

7.1.3 回溯法的一般性描述

1. 回溯法解题的基本步骤

1）处理一个复杂的问题，常常会有很多可能解，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，解空间中应该包括所有的可能解。

- 令问题的解向量为$X=(x_0,x_1,\dots,x_{n-1})$
- x_i的取值范围为S_i，$S_i=\{a_{i.0},a_{i.1},\dots,a_{i.m_i}\}$
- 问题的解空间由笛卡尔积 $A=S_0\times S_1\times\cdots\times S_{n-1}$构成

2）状态空间树的结构

状态空间树是一棵高度为 n的树，树的所有叶子结点为可行解：

- ✓ 第 0 层有 $|S_0| = m_0$ 个分支；
- ✓ 第1层有m_0个分支结点，构成m_0棵子树，每一棵子树都有 $|S_1| = m_1$个分支结点；
- ✓ 第 2 层，$m_0 \times m_1$ 个分支结点，构成 $m_0 \times m_1$棵子树；
- ✓ 第 n层，有 $m_0 \times m_1 \times \cdots \times m_{n-1}$ 个叶子结点。

3）以深度优先方式搜索解空间，利用剪枝函数来避免搜索进入不可能得到解的子空间：

剪枝函数包括两类：

- ✓ 使用约束函数，剪去不满足约束条件的路径；
- ✓ 使用目标函数，剪去不能得到最优解的路径。

最后生成搜索树，取得问题的解。

2. 回溯法的一般描述

- 几个变量和函数说明：

  $m[i]$：集合S_i 的元素个数，$|S_i| = m[i]$

  $x[i]$：解向量 X的第 i个分量

  $k[i]$：当前算法对集合S_i中的元素的取值位置

  initial(x)：解向量 x初始化为空；

  $a(i, k[i])$：取S_i的第 $k[i]$ 个值

constrain(*x*)：判断解向量是否满足约束条件，若满足则返回真

bound(*x*)：判断解向量是否满足目标函数的界，若满足则返回真

solution(*x*)：判断解向量是否为问题的最终解，如果是则标志置为真

回溯法迭代实现

```
1. void backtrack_item( )
2. {
3.     initial(x);
4.     i = 0;   k[ i ] = 0;   flag = FALSE;
5.     while (i >= 0) {
6.         while (k[ i ] < m[ i ]) {     //遍历当前结点的所有子结点
7.             x[ i ] = a(i,k[ i ]);        //取Si的第k[i]个值赋给分量x[i]
8.             if (constrain(x)&&bound(x)) {        //判断是否满足约束条件和限界条件
9.                 if (solution(x))  {    //判断是否为问题的最终解
10.                  flag = TRUE;    break;  } //是最终解，退出循环
12.                else { i = i + 1;   k[ i ] = 0; }     //没得到最终解，继续向下搜索
15.             }
16.         else k[ i ] = k[ i ] + 1;//取Si的下一个值，即舍弃所有子树，搜索同一个父亲结点下的另一个兄弟子树
17.       }
18.       if (flag) break;  //当前层同一父亲的兄弟子树已全部搜索完毕，if找不到部分解，也找不到最终解。
19.       i = i – 1;  //回溯到上一层
20.   }
21.   if (!flag)
22.   initial(x);
23. }
```

- 解向量初始化为0;
- 从解分量第1个分量处理，搜索第0层.

退出外循环：

- 找到最终解，返回最终解；
- 第0层的子树已全部搜索完，都找不到部分解，则解空间置为空，返回空向量。

- 回溯法求解的三个步骤：

（1）定义一个解空间，它包含问题的解（这个空间必须至少包含问题的一个解）；

（2）用易于搜索的方式组织解空间（典型的组织方法是图或树）；

（3）深度优先搜索解空间，用约束方程和目标函数的界修剪状态空间树，获得问题的解。

7.2 n 后问题

7.2.1 四后问题的求解过程

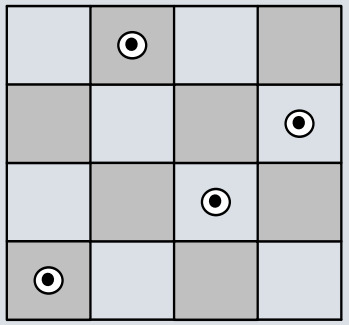
7.2.2 n 后问题算法的实现

7.2.1 四后问题的求解过程

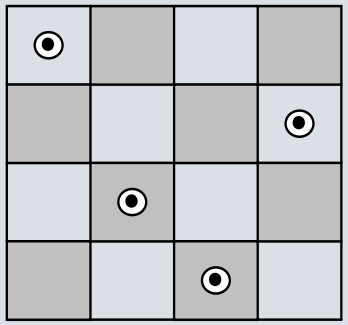
1. 四后问题的解空间

向量$x = (x_1, x_2, x_3, x_4)$表示皇后的布局。分量 x_i 表示第 i行皇后的列位置， x_i的取值范围 $S_i = \{1, 2, 3, 4\}$，有 4^4 个可能解。

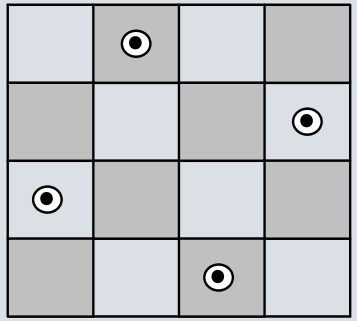
例：解向量为 (2,4,3,1)、(1,4,2,3) 、(2,4,1,3) 所对应的皇后布局如下。 (2,4,1,3) 是问题的一个解。



(a)



(b)



2. 四后问题的约束方程

解向量：(x_1, x_2, x_3, x_4)

解空间：$4^4 \Rightarrow 4!$种可能

显式约束：$x_i = 1, 2, 3, 4$

隐式约束：

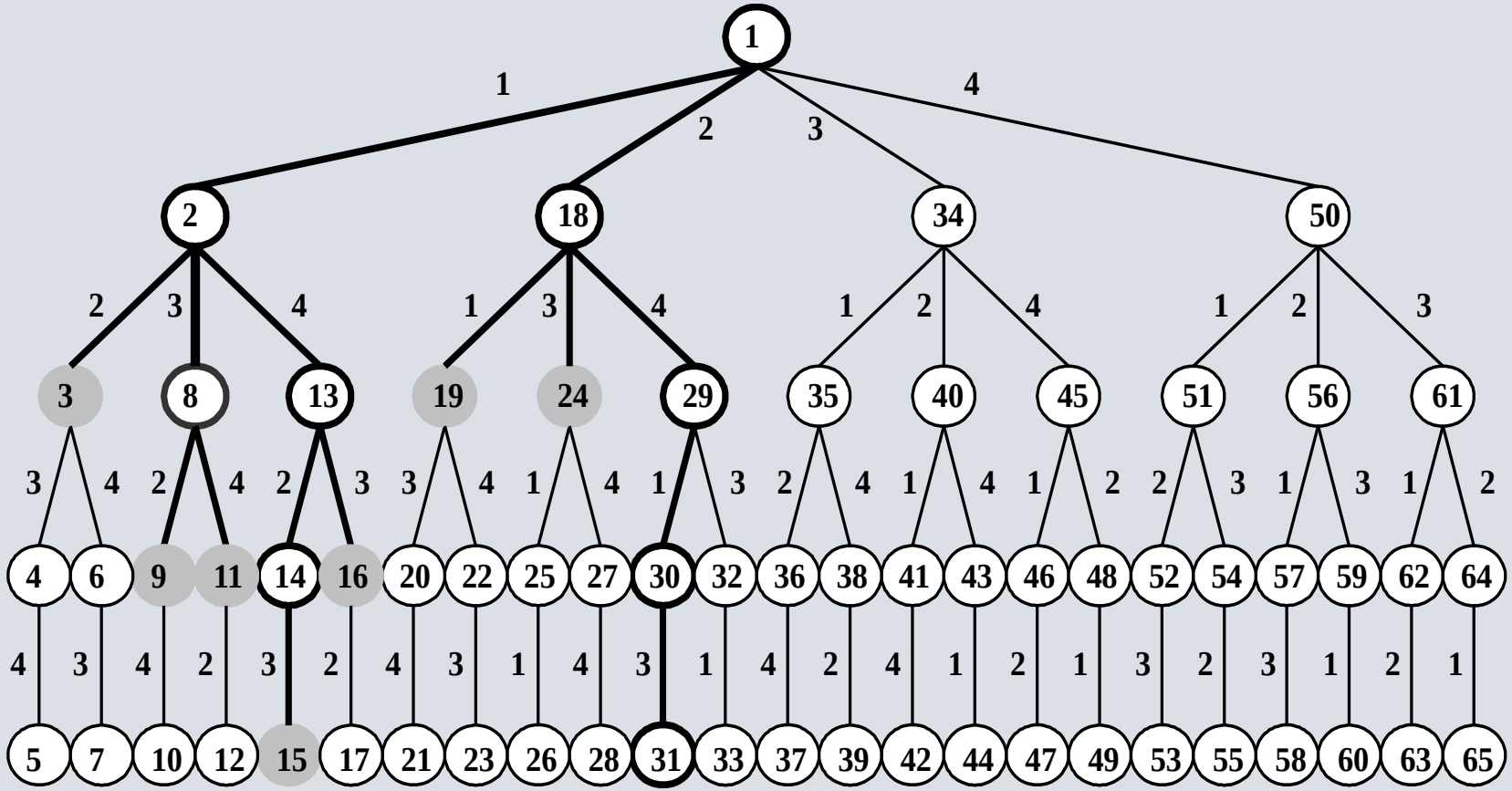
（1）不同列：$x_i \neq x_j \quad 1 \leq i \leq 4, i \neq j$

（2）不处于同一正反对角线：

$|x_i - x_i| \neq |i - j| \quad 1 \leq i \leq 4, i \neq j$

3. 四后问题的状态空间树及其搜索过程

四后问题的状态空间树是一棵完全四叉树，按约束方程简化如下：



4-皇后问题-回溯解

	1	2	3	4
1				
2				
3				
4				

4. n后问题的求解步骤

（1）令皇后的行号 $k = 1$，第 1 行的皇后列号 $x[1] = 0$;

（2）若 $k > 0$，则皇后列号 $x[k]$ 加1，转步骤（3），否则，问题无解，算法结束。

（3）若 $x[k] > n$，或列号满足约束条件，则转步骤（4），否则列号 $x[k]$加 1，继续执行步骤（3）。

（4）若 $x[k] > n$，则执行回溯，即 $x[k]$ 复位为 0，$k = k$ -1，转步骤（2），否则转步骤（5）。

（5）若 $k = n$，算法结束，否则，处理下一行皇后，即 $k = k$ +1，$x[k]$复位为 0，转步骤（2）。

7.2.2 n 后问题算法的实现

1. place 函数

函数place用约束方程来判断第 *k* 行皇后所处位置的正确性

```
1. BOOL place(int x[ ], int k)
2. {
 3.     int i;
 4.     for (i=1; i<k; i++)
 5.         if ((x[ i ] == x[ k ]) || (abs(x[ i ]-x[ k ]) == abs(i-k)))
 6.             return FALSE;
 7.     return TRUE;
8. }
```

2. n 后问题算法的实现

```
1. void n_queens(int n, int x[ ])
2. {    int k = 1;  x[1] = 0;  //从第1个皇后的第0列开始搜索
5.      while (k>0) {     // k：搜索深度
6.         x[k] = x[k] + 1; //在当前列加1的位置开始搜索
7.         while ((x[k] <= n) && (!place(x,k)))
8.             x[k] = x[k] + 1; //不满足条件，继续搜索下一列
9.         if (x[k]<=n) {  //存在满足条件的列
10.            if (k==n)  break; //最后一个皇后，完成搜索
11.            else {  k = k + 1;   x[k] = 0;} //不是，处理下一个皇后
14.        }
15.        else {    x[k] = 0;   k = k – 1;  }//已判断完n列，均不满足条件，
                                                //第k列复位为0，回溯到前一行
18.    }
19. }
```

3. n 后问题算法的分析

- 运行时间，取决于所访问结点个数 c；
- 每访问一个节点，就调用一次place函数计算约束方程；
- place 函数循环体的执行次数，最少一次，最多 $n-1$ 次因此，总次数为 $O(cn)$。
- 结点个数是动态生成的，对不同实例，具有不确定性，一般可由 n 的多项式确定。

7.3 图的着色问题

7.3.1 图着色问题的求解过程

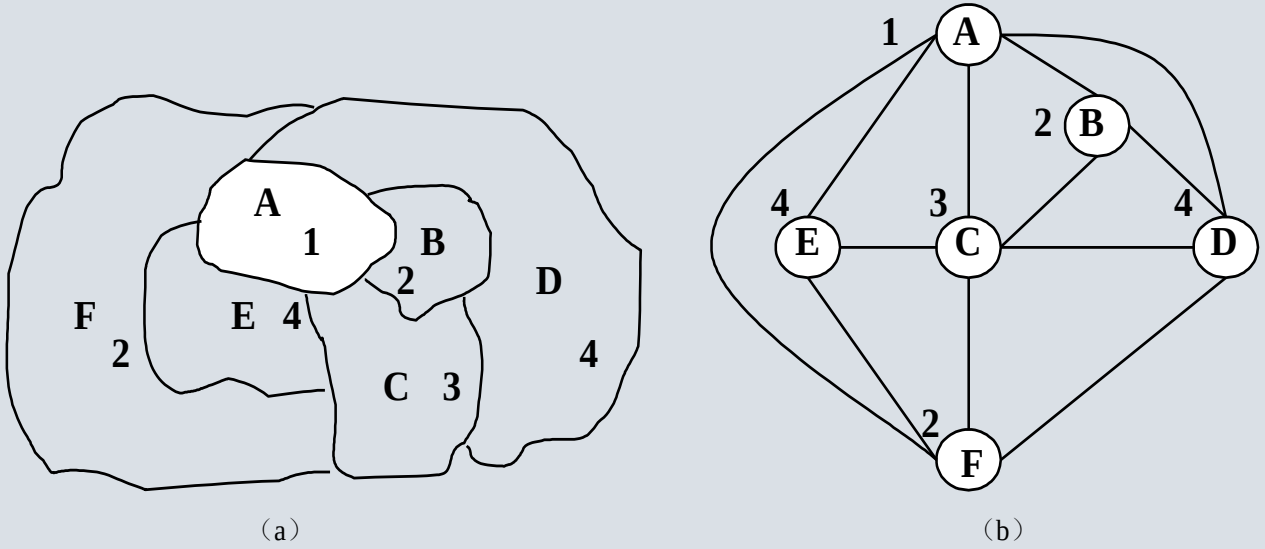
7.3.2 图的m着色问题算法的实现

- 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。
- 整数m为该图的着色数。求一个图的色数m的问题称为图的m~着色最优化问题。

7.3.1 图着色问题的求解过程

1. 图的 m 着色问题

1）区域图抽象为平面图



（a）　　（b）

2）图 m 着色问题的描述

用 m 种颜色为无向图 G = <V, E> 的每个顶点着色，要求每个顶点着一种颜色、并使相邻两个顶点之间具有不同颜色。

设无向图$G=(V, E)$采用如下邻接矩阵表示：

$$c[i][j] = \begin{cases} 1 & if\ (i,j) \in E \\ 0 & others \end{cases}$$

3）图 m 着色问题的解空间

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$(c_1, c_2, \ldots, c_n)$，$c_i \in \{1, 2, \ldots, m\}$，$1 \le i \le n$来描述图的一种可能着色。

用数组x来存放n元组$(c_1, c_2, \ldots, c_n)$，其中，$x_i \in \{1, 2, \ldots, m\}$ $(0 \le i < n)$表示赋予顶点i的颜色，这是显式约束。

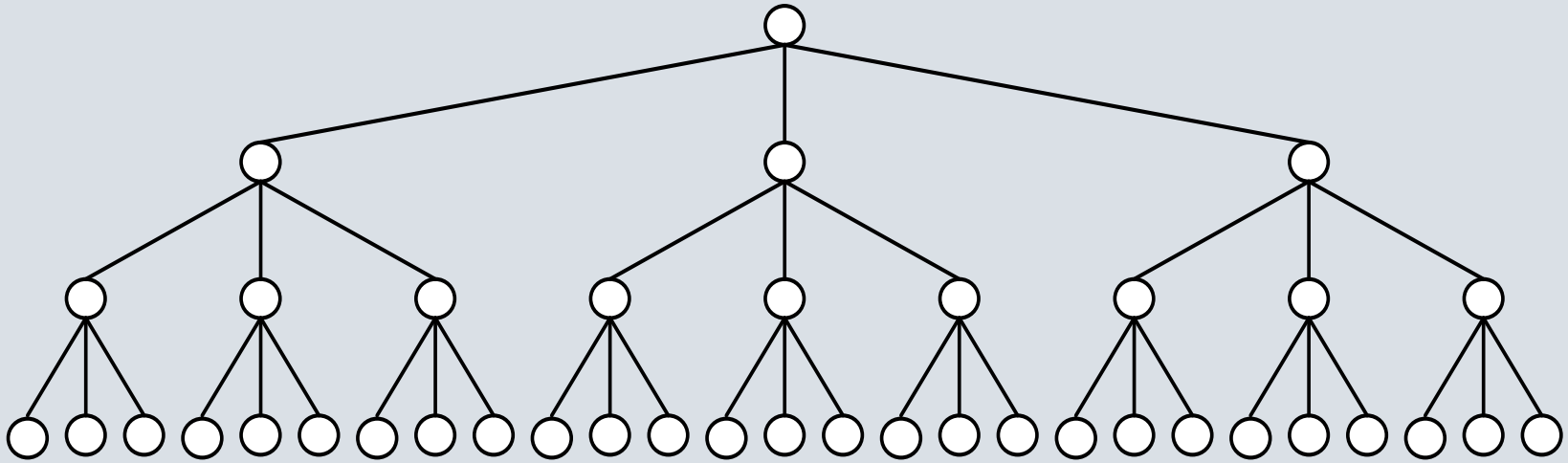
因此解空间大小为m^n。

例：(1, 3, 2, 3, 1) 表示对具有 5 个顶点的图的一种着色。

4）图着色问题的状态空间树

高度为 n 的完全 m 叉树（树的高度指从树的根结点到叶子结点的最长通路的长度）。每一个分支结点，都有 m 个儿子结点。最底层有m^n个叶子结点。

例：3 着色 3 个顶点的状态空间树

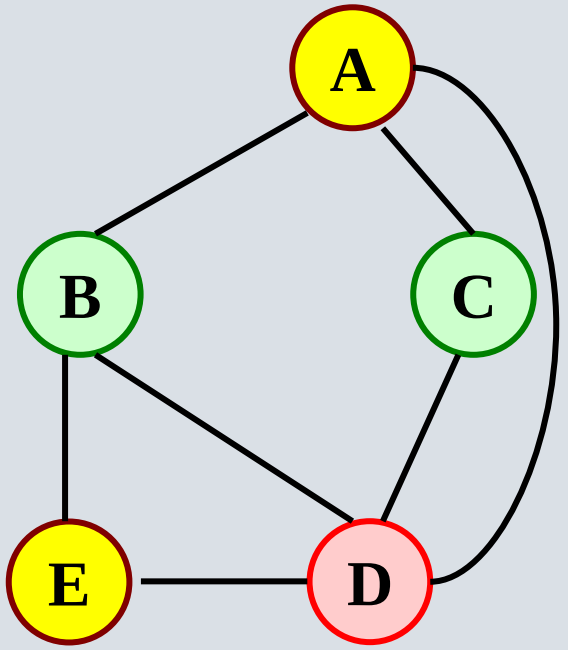


2. 图的 m 着色问题的搜索过程

(1)约束方程

从隐式约束产生：对所有i和j $(0 \le i, j < k, i \ne j)$，若$c[i][j]=1$，则$x_i \ne x_j$。

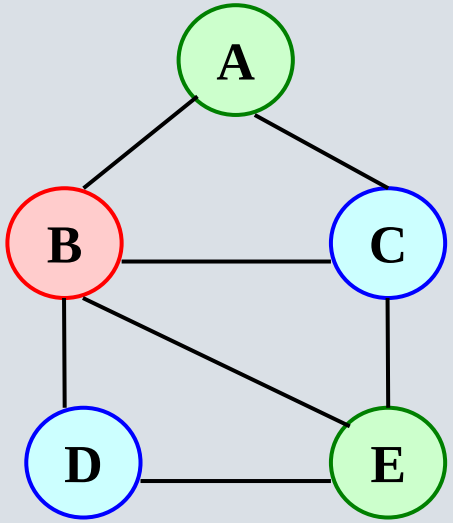
例如， 5元组(1, 2, 2, 3, 1)表示对具有5个顶点的无向图的一种着色，顶点A着颜色1，顶点B着颜色2，顶点C着颜色2，如此等等。



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

(2)搜索过程：

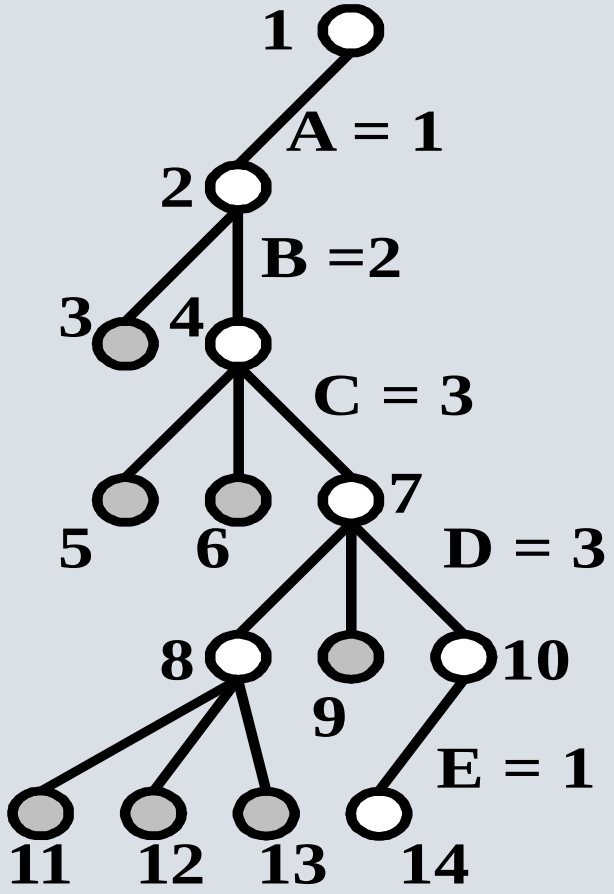
例子：三着色右图



状态空间树结点总数为：

1+3+9+27+81+243=364

搜索过程中所访问的结点数：14



回溯法求解

- 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。

3. 图的 m 着色问题的求解步骤

（1）初始化：对所有的 i，$0 \le i \le n-1$，置顶点 i 的颜 色值 $x[i]=0$，令顶点号 $k=0$；

（2）若 $k \ge 0$，则颜色值 $x[k]$ 加1，转步骤（3），否则，m不可着色，算法结束；

（3）若 $x[k] > m$，或是有效着色，转步骤（4），否则$x[k]$ 加 1，继续执行步骤（3）；

（4）若$x[k] > m$，则 $x[k]$ 复位为 0，$k = k-1$，转步骤（2），否则转步骤（5）；

（5）若 $k = n-1$，m 可着色，算法结束，否则，$k = k+1$，转步骤（2）。

7.3.2 图的 m 着色问题算法的实现

1. 数据结构

```
int     n;              // 顶点个数
int     m;              // 最大颜色数
int     k;              // 顶点号码，或搜索深度
int     x[n];           // 顶点的着色
BOOL    c[n][n];        // 布尔值表示的图的邻接矩阵
```

2. 算法描述

1）函数ok：判断顶点着色的有效的性

有效返回 TRUE，无效返回 FALSE

```
1. BOOL ok(int x[ ], int k, BOOL c[ ][ ], int n)
2. {
3.     int i;
4.     for (i=0; i<k; i++) {
5.         if (c[ k ][ i ] && (x[ k ] == x[ i ])
6.             return FALSE;
7.     return TRUE;
8. }
```

2）m 着色算法

```
1. void m_coloring(int n, int m, int x[ ], BOOL c[ ][ ])
2. {    int i,k;
4.      for (i=0;i<n;i++)      x[i] = 0;
6.      k = 0;
7.      while (k >= 0) {
8.          x[ k ] = x[ k ] + 1;
9.          while ((x[k] <= m) && (!ok(x, k, c, n)))  x[k] = x[k] + 1;
11.         if (x[k] <= m) {     if (k == n-1) break;
13.                              else k = k + 1;      }
15.         else {   x[k] = 0;   k = k – 1;   }
18.     }
19.     if (k==n-1) return TRUE;
20.     else FALSE;
21. }
```

3 算法的分析

状态空间树的结点总数为：

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

每访问一个结点，就调用一次 ok 函数计算约束方程，ok 函数循环体的执行次数与搜索深度有关，最少一次，最多 $n-1$ 次。

在最坏情况下，算法的总花费为$O(nm^n)$

□ 会场分配问题：

有n项活动需要安排，对于活动i, j，如果i, j时间冲突，就说i与j不相容。如何分配这些活动，使得每个会场的活动相容且占用会场数最少？

□ 建模：

活动作为图的顶点，如果i与j不相容，则在i与j之间加一条边，会场标号作为颜色标号。求图的一种着色方案，使得使用的颜色数最少。

7.4 哈密尔顿回路问题

7.4.1 哈密尔顿回路的求解过程

7.4.2 哈密尔顿回路算法的实现

- 哈密尔顿图是一个无向图，由天文学家哈密尔顿提出，由指定的起点前往指定的终点，图中经过所有其他结点且只经过一次。
- 在图论中是指含有哈密尔顿回路的图，闭合的哈密尔顿路径称为哈密尔顿回路，含有图中所有顶点的路径称为哈密尔顿路径。

7.4.1 哈密尔顿回路的求解过程

1. 解空间和状态空间树

- 设图 $G = <V, E>$是一个n结点的连通图，$v_1v_2...v_n$是G的一条通路；图中顶点集为 $V = \{ 0, 1, ..., n - 1 \}$；
- 哈密尔顿通路:G中每个顶点在该通路中出现且仅出现一次；
- 若$v_1=v_n$，且$v_2...v_n$在该通路上出现且仅出现一次，则称该通路为哈密尔顿回路；
- 用向量 $X = (x_0, x_1, ..., x_{n-1})$，$x_i \in \{1, 2, ..., n\}$表示回溯法求得的解，其中$x_i$是找到的环中第$i$个被访问的结点。

- 状态空间树（ n个顶点）：
  - 高度为 n的完全 n叉树
  - 最底层有n^n个叶子结点
  - 根结点到叶子结点的每一条路径，都是一个可能解

2. 搜索过程

1）约束方程：

$c[i][j]$：布尔量表示的图的邻接矩阵，顶点 i 和 j 相邻接则为真，否则为假；

约束方程：

$$c[x_i][x_{i+1}] = TRUE \qquad 0 \leqslant i \leqslant n-1$$

$$c[x_0][x_{n-1}] = TRUE$$

$$x_i \neq x_j \qquad 0 \leqslant i, j \leqslant n-1,\ i \neq j$$

2）搜索过程

- ✓ 回路中所有顶点的编号初始化为-1；
- ✓ 顶点0作为回路中的第一个顶点，搜索与顶点0相邻接的编号最小的顶点；
- ✓ 若在搜索过程中已生成通路$l = x_0x_1 \ldots x_{i-1}$，在继续搜索时，根据约束方程，在V中寻找与$x_{i-1}$相邻接且不属于通路$l$中顶点的编号最小的顶点；
- ✓ 如果搜索成功，则把该顶点作为通路中的顶点，继续搜索下一个顶点；
- ✓ 若搜索失败，l中删去x_{i-1}，从顶点编号加1位置继续搜索；

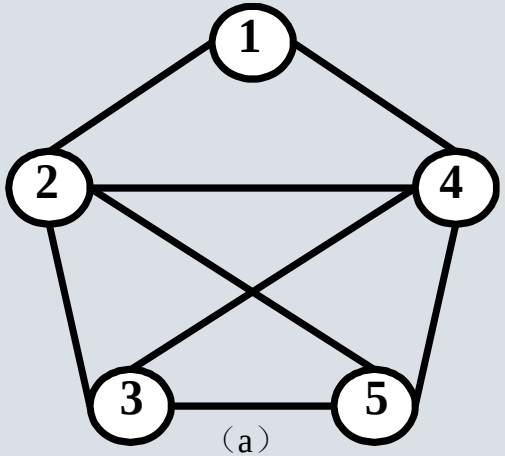
- ✓ 当搜索到通路l中的顶点x_{n-1}时：
  - ➢ 如果x_{n-1}与x_0相邻接，则生成的回路就是一条哈密尔顿回路；
  - ➢ 否则把通路l中的顶点x_{n-1}删去，继续回溯；
- ✓ 如果在回溯过程中，通路l中只剩下一个顶点x_0，则表明图中不存在哈密尔顿回路，即该图不是哈密尔顿图。

3）例子

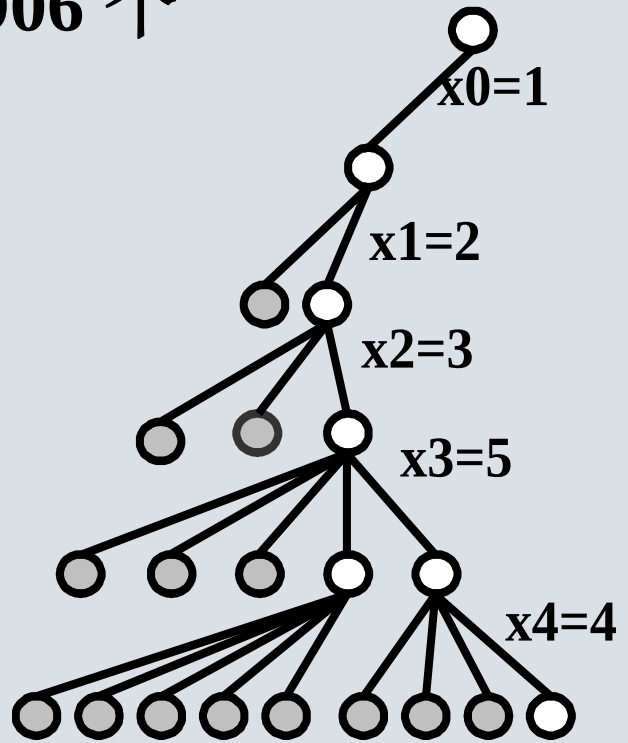
状态空间树的结点总数：

1+5+25+125+625+3125 =3906 个

c = 21 个



（a）



（b）

4）实现步骤

（1）初始化：对所有的 i，$0 \le i \le n-1$，置顶点号 $x[i] = -1$，顶点状态 $s[i]$ = FALSE，令 $k = 1$，$s[0]$ = TRUE，$x[0] = 0$。

（2）若 $k \ge 0$，则顶点号 $x[k]$加1，转步骤（3），否则，不存在哈密尔顿回路，算法结束。

（3）若 $x[k] < n$，转步骤（4），否则转步骤（7）

（4）若顶点 $x[k]$ 不在回路中，且顶点 $x[k]$ 与顶点 $x[k-1]$邻接，转步骤（5），否则 $x[k]$ 加 1，转步骤（3）。

（5）若 $x[k] < n$，且 $k \ne n-1$，则令 $s[x[k]]$为 TRUE，令 $k = k+1$，转步骤（2），否则转步骤（6）。

（6）若 $x[k] < n$，且 $k = n-1$，且顶点 $x[k]$与顶点 $x[0]$邻接，则找到哈密尔顿回路，算法结束，否则转步骤（7）。

（7）$x[k]$ 复位为 -1，$k = k-1$，$s[x[k]]$ 复位为 FALSE，转步骤（2）。

7.4.2 哈密尔顿回路算法的实现

1. 数据结构

```
int     n;              // 顶点个数
int     x[n];           // 哈密尔顿回路上的顶点编号
BOOL c[n][n];     // 布尔值表示的图的邻接矩阵
BOOL s[n];      // 顶点状态,已处于所搜索的通路
                        // 上的顶点为真
```

2. 算法描述

```
1. void hamilton(int n, int x[ ], BOOL c[ ][ ])
2. {
3.     int i, k;
4.     BOOL *s = new  BOOL[ n ];
5.     for (i=0;i<n;i++)            { // 初始化
6.         x[ i ] = -1;   s[ i ] = FALSE;
7.     }
8.     k = 1;   s[ 0 ] = TRUE;   x[ 0 ] = 0;
```

算法描述（续）

```
 9.    while (k >= 0) {
10.        x[ k ] = x[ k ] + 1;
11.        while (x[ k ] < n)                          // 进行搜索
12.            if (!s[ x[ k ] ] && c[ x[ k-1 ] ][ x[ k ] ])   break;
14.            else x[ k ] = x[ k ] + 1;
15.        if ((x[ k ] < n) && (k != n-1))       // 搜索到一个顶点
16.        {    s[ x[ k ] ] = TRUE;   k = k + 1;      }
18.        else if ((x[ k ] < n) && (k == n-1) && (c[x[k]][x[0]]))
19.            break;                  // 搜索到最后顶点，结束搜索
20.        else                  // 搜索失败，回溯
21.        {    x[ k ] = -1;   k = k – 1;   s[ x[ k ] ] = FALSE;   }
23.    }
```

```
24.     delete s;
25.     if (k==n-1) return TRUE;
26.     else return FALSE;
23. }
```

3. 算法分析

状态空间树中的结点总数为：

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

每个结点判断时间为 $O(1)$，算法的总花费为 $O(n^n)$.

7.5 背包问题

7.5.1 回溯法解0/1背包问题的求解过程

7.5.2 回溯法解0/1背包问题算法的实现

7.5.1 回溯法解0/1背包问题的求解过程

1. 问题的解空间和状态空间树

- n个物体 v_i，重量 w_i、价值 p_i，背包的载重量 M
- x_i：物体v_i被装入背包的情况，$x_i = 0, 1$
- 解向量：$X = (x_0, x_1, \dots, x_{n-1})$
- 状态空间树：高度为 n的完全二叉树，结点总数有$2^{n+1} - 1$ 个
  - 第 i层的左儿子子树，表示物体v_i被装入背包；
  - 第 i层的右儿子子树，表示物体v_i未被装入背包；
  - 根结点到叶结点的路径，是问题的可能解；
  - 从可能解中搜索一个最优解。

2. 求解过程

1）约束方程和目标函数：

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

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

2）初始化：

- 目标函数上界置为 0；
- 物体按价值重量比的非增顺序排序

3）搜索过程

- 尽量沿左儿子结点前进，当不能沿左儿子继续前进时，就得到问题的一个局部解，并把搜索转移到右儿子子树；
- 估计由局部解所能得到的最大价值；
  - 估计值高于当前上界：继续由右儿子子树向下搜索，扩大局部解，直到找到可行解；保存可行解，用可行解的值刷新目标函数的上界，向上回溯，寻找其它可行解；
  - 估计值小于当前上界：丢弃当前正在搜索的部分解，向上回溯 。

3. 局部解的最大估值

- 假定，当前局部解是$\{x_0, x_1, \ldots, x_{k-1}\}$，同时，有

  $\sum_{i=0}^{k-1} x_i w_i \leq M$且$\sum_{i=0}^{k-1} x_i w_i + w_k > M$

- 将得到局部解$\{x_0, x_1, \ldots, x_k\}$，其中，$x_k = 0$。

- 由这个局部解继续向下搜索，将有：

$$\sum_{i=0}^{k} x_i w_i + \sum_{i=k+1}^{k+m-1} w_i \leq M$$

$$\sum_{i=0}^{k} x_i w_i + \sum_{i=k+1}^{k+m-1} w_i + w_{k+m} > M$$

$$m = 2, \ldots, n-k-1$$

- 将超过背包的载重量，则可能解的最大值不会超过：

$$\sum_{i=0}^{k} x_i p_i + \sum_{i=k+1}^{k+m-1} x_i p_i + (M - \sum_{i=0}^{k} x_i w_i - \sum_{i=k+1}^{k+m-1} x_i w_i) \times p_{k+m}/w_{k+m}$$

用上述关系估计从局部解$\{x_0, x_1, , \ldots, x_{k-1}\}$继续向下搜索时可能取得的最大价值。

4. 回溯的两种情况

- 当估计值小于已经得到的可行解中的最大值时向上回溯；
- 当前的结点是左儿子分支结点，转而搜索相应的右儿子分支结点；当前的结点是右儿子分支结点，就沿右儿子分支结点向上回溯，直到左儿子分支结点为止，然后，再转而搜索相应的右儿子分支结点。

5. 算法实现步骤

1）变量描述：

w_cur：部分解中装入背包物体的总重量

p_cur：部分解中装入背包物体的总价值

p_est：部分解可能达到的最大估计值

p_total：当前搜索到的所有可行解中的最大价值

x_k：部分解的第 k个分量

y_k：部分解的第 k个分量的拷贝

k：搜索深度。

2）步骤说明

①物体按价值重量比的非增顺序排序；

② p_cur, w_cur和p_total初始化为 0，局部解各分量初始化为 0，搜索树的搜索深度k置为0 ；

③ 估算从当前的局部解可取得的最大价值 p_est；

④ 如果 p_est>p_total，转 5；否则转 8；

⑤ 从v_k开始把物体装入背包，直到没有物体可装或装不下物体 v_i为止，生成局部解$y_k, \ldots y_i$ ，刷新 p_cur；

⑥ 如果 i ≥ n - 1，得到新的可行解，所有y_i拷贝到x_i，

置 p_total = p_cur，p_total是目标函数的新上界；令 $k = n$，转3，以便回溯搜索其它的可能解；否则得到一个部分解，转7；

⑦令 k = i + 1，舍弃物体V_i，转 3，从物体V_{i+1}继续装入；

⑧ 当 i ≥ 0，且$y_i = 0$，执行 i = i – 1，直到 $y_i \neq 0$；即

沿右儿子分支结点方向向上回溯，直到左儿子分支结点

⑨ 如果 i< 0，算法结束；否则，转 10；

⑩ 令 $y_i = 0$，$w_{cur} = w_{cur} - w_i$，$p_{cur} = p_{cur} - p_i$，$k = i + 1$，转 3；从左儿子分支结点转移到相应的右儿子分支结点，继续搜索其它的局部解或可能解。

- 尽量沿着左儿子分支结点向下搜索，直到无法继续向前推进而生成右儿子分支结点为止；
- 在回溯过程中，尽量沿着右儿子分支结点向上回溯，直到遇到左儿子分支结点并转而生成右儿子分支结点；
- 在右儿子分支结点开始搜索时，都对可能取得的最大价值进行估计；
- 在叶子结点开始继续搜索时，通过把搜索深度k置为n，使得不会进行估计值的计算，而直接把估计值置为当前值。从而不会大于当前目标函数的上界，而直接从叶子结点进行回溯。

7.5.2 回溯法解 0/1 背包问题算法的实现

1. 数据结构和变量

```
typedef struct {
    float    w;                    // 物体重量
    float    p;                    // 物体价值
    float    v;                    // 物体的价值重量比
} OBJECT;
OBJECT       ob[n];
float    M;                        // 背包载重量
int      x[n];                     // 可能的解向量
int      y[n];                     // 当前搜索的解向量
float    p_est;                    // 装入背包物体的估值
float    p_total;                  // 装入背包物体的价值上界
float    w_cur;                    // 当前装入背包的物体的总重量
float    p_cur;          // 当前装入背包的物体的总价值
```

2. 算法描述（初始化）

```
1. float knapsack_back(OBJECT ob[ ], float M, int n,
                       int x[ ])
2. {   int i, k;
4.     float w_cur, p_total, p_cur, w_est, p_est;
5.     int *y = new  int[ n ];
6.     for (i=0; i<=n; i++) {              // 计算物体的价值重量比
7.         ob[ i ].v = ob[ i ].p / ob[ i ].w;
8.         y[ i ] = 0;                     // 当前的解向量初始化
9.     }
10.    merge_sort(ob,n);                   // 物体按价值重量比排序
11.    w_cur = p_cur = p_total = 0;                // 初始化
12.    y[n] = 0;    k = 0;
```

算法描述（估算沿当前分支可取得的最大价值）

```
13.    while (k >= 0) {
14.        w_est = w_cur;   p_est = p_cur;
15.        for (i=k; i<n; i++) {          // 计算最大估值
16.            w_est = w_est + ob[ i ].w;
17.            if (w_est < M)
18.                p_est = p_est + ob[ i ].p;
19.            else {
20.                p_est = p_est +
                      ((M – w_est + ob[ i ].w) / ob[ i ].w) * ob[ i ].p;
21.                break;
22.            }
23.        }
```

算法描述（估计值大于上界，局部解的处理）

```
24.        if (p_est>p_total) {            // 估计值大于上界
25.            for (i=k; i<n; i++) {
26.                if (w_cur+ob[ i ].w <= M) {  // 可装入第 i 个物体
27.                    w_cur = w_cur + ob[ i ].w;
28.                    p_cur = p_cur + ob[ i ].p;
29.                    y[ i ] = 1;
30.                }
31.                else {
32.                    y[ i ] = 0;   break;    // 不能装入第 i 个物体
33.                }
34.            }
```

算法描述（估计值大于上界，可行解的处理）

```
35.          if ( i >= n) {          // n个物体已全部装入
36.              if (p_cur > p_total) {
37.                  p_total = p_cur;   k = n;    // 刷新当前上限
38.                  for (i=0; i<n; i++)        // 保存可行的解
39.                      x[ i ] = y[ i ];
40.              }
41.          }
42.          else k = i + 1;       // 继续装入其余物体
43.      }
```

算法描述（估计值小于上界，回溯处理）

```
44.        else {                              // 估计价值小于当前上限
45.            while ((k >= 0) &&( !y[ k ] )      // 沿右分支回溯
46.                k = k – 1;                     // 直到左分支结点
47.            if (k<0)  break;      // 已到达根结点,算法结束
48.            else {
49.                w_cur = w_cur – ob[ k ].w;     // 修改当前值
50.                p_cur = p_cur – ob[ k ].p;
51.                y[ k ] = 0;   k = k + 1;        // 搜索右分支子树
52.            }
53.        }
54.    }
55.    delete y;        return p_total;
57. }
```

3. 算法分析

在最坏情况下，状态空间树有$2^{n+1}-1$个结点，有$O(2^n)$ 个右儿子结点；

每个右儿子结点都需估计继续搜索可能取得的目标函数的最大价值，每次估计时间需花费 $O(n)$ 时间；

其余处理需 O(1) 时间；

算法总花费时间为$O(n2^n)$

7.6 回溯法的效率分析

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：

① 产生结点$x[k]$的时间；

② 满足显示约束的$x[k]$值的个数；

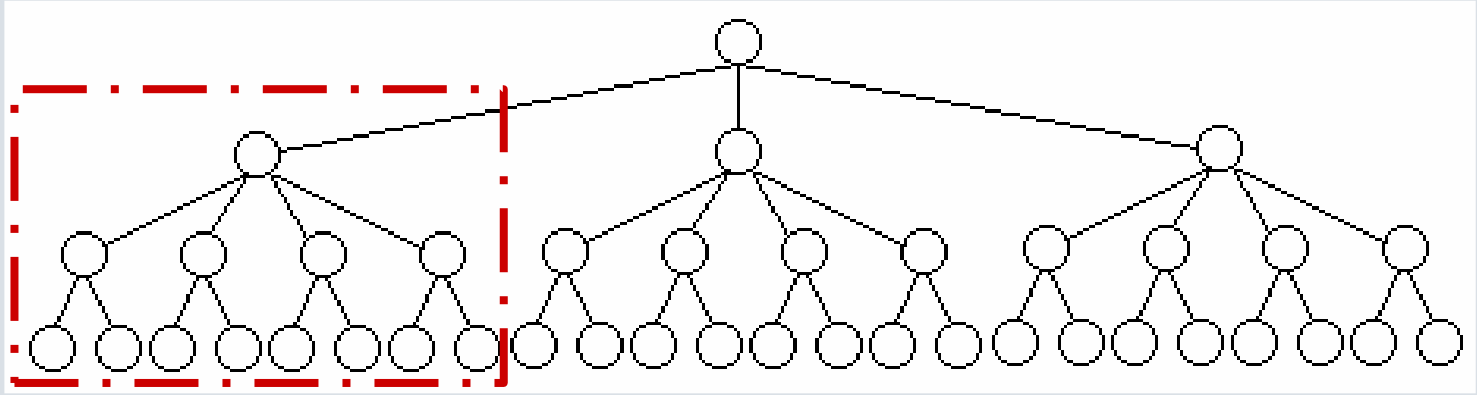
③ 计算约束函数constraint的时间；

④ 计算目标函数边界bound的时间；

⑤ 满足约束函数和目标函数边界约束的所有$x[k]$的个数。

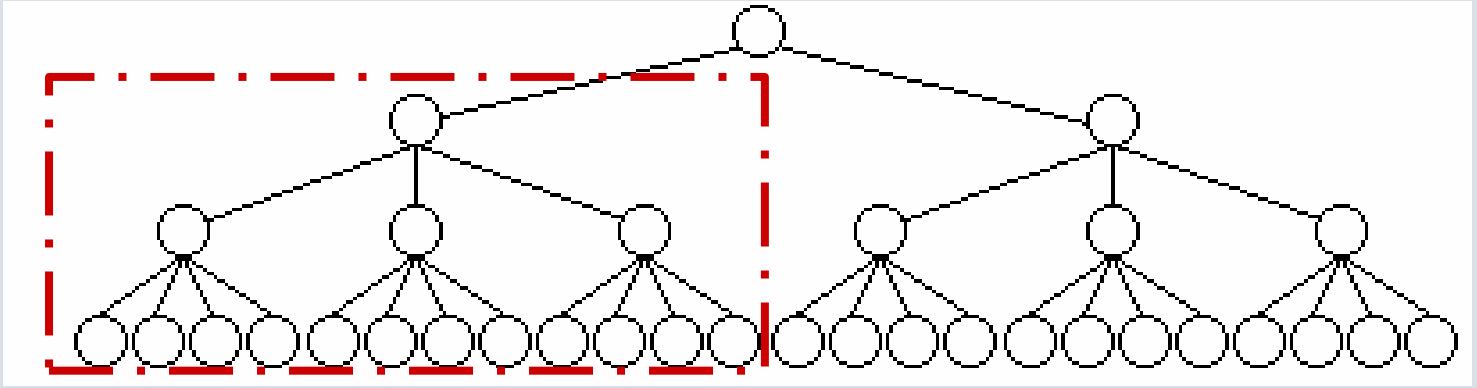
- 好的约束函数能显著地减少所生成的结点数。但这样的约束函数往往计算量较大。因此，在选择约束函数时，通常存在生成结点数与约束函数计算量之间的折中。
- 对于很多问题而言，在搜索试探时选取$x[i]$的值的顺序是任意的。在其他条件相当的前提下，让可取值最少的$x[i]$优先。

- 实例：图中关于同一问题的2棵不同解空间树：



|S1|=3， |S2|=4， |S3|=2

按m_i递增顺序来排列x_i在解向量中的顺序位置：



|S3|=2 ， |S1|=3， |S2|=4

回溯法的效率分析

- **Monte Carlo方法**：在解空间树上产生一条随机路径，沿此路径估算解空间中满足约束条件的结点数m：
  - 设*x*是所产生的随机路径上的一个结点，在解空间树的第*i*层（深度为*i*）；
  - 对*x*的所有儿子节点，用约束函数检测出满足约束条件的结点数*mi*；
  - 随机选取路径上的下一个结点：从*mi*中任意选取一个；
  - 这条路径一直延伸到一个叶结点或一个所有儿子结点都不满足约束条件的结点为止；
  - 通过这些*mi*的值，估出解空间中满足约束条件的结点总数*m*

- 使用Monte Carlo方法的假设条件：
  - 限界函数是固定的，即在算法执行期间当其信息逐渐增加时，限界函数不变；
  - 同一个函数正好用于这棵状态空间树同一级的所有节点。

- 使用Monte Carlo方法估算节点总数的方法：

假设第1层有m_0个满足约束条件的节点，每个节点有m_1个满足约束条件的子节点，则第2层上有m_0m_1个满足约束条件的节点；

同理，假设第2层上的每个节点均有m_2个满足约束条件的子节点，则第3层上有$m_0m_1m_2$个满足约束条件的节点，依次类推，第n层上有$m_0m_1m_2 \ldots m_{n-1}$个满足约束条件的节点，因此，这条随机路径上的节点总数为：

$$m = m_0 + m_0m_1 + m_0m_1m_2 + \cdots + m_0m_1m_2 \ldots m_{n-1}$$

Monte Carlo数率估计算法

```
procedure ESTIMATE
  m←1; r ← 1; k ← 1
  loop
     T_k←{X(k):X(k)∈ T(X(1),…X(k-1)) and
      B(X(1),…X(k))}
     if SIZE(T_k)=0 then exit endif
     r←r*SIZE(T_k)
     m←m+r
     X(k)←CHOOSE(T_k)
     K←K+1
  repeat
  return(m)
end ESTIMATE
```

ESTIMATE是一个确定m值的算法

回溯法小结

- 适用：求解搜索问题和优化问题
- 搜索空间：树，结点对应部分解向量，可行解在树叶上。
- 搜索过程：采用系统的方法隐含遍历搜索树
- 搜索策略：深度优先，宽度优先，函数优先，宽深结合等。
- 结点分支判定条件：满足约束条件——分支扩张解向量；不满足约束条件——回溯到到该点的父节点。
- 结点状态：动态生成
- 存储：当前路径

回溯法的迭代形式的一般框架

```
Void IBacktrack(int n)
{
    int k=0;
    while(k>=0)
    {
        if(还剩下尚未检测的x[k]使得x[k]∈T(x[0],…x[k-1]) &&
    (Bk(x[0],…,x[k]))
        {
          if((x[0],x[1],…,x[k])是一个可行解)   //考虑x[k]的下一个
    可取值
                        输出(x[0],x[1],…,x[k]);
            k++;                  //考虑下一层分量
        }
        else k--;                 //回溯到上一层
    }
}
```