回溯算法

陈长建

计算机科学系

课前问题

- 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}。

- 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。

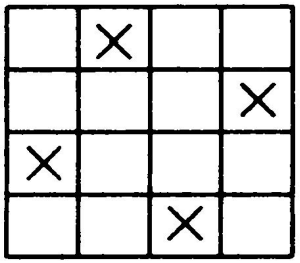
- 伪代码

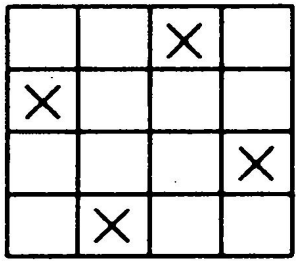
N后问题

- 在n×n格的棋盘上放置彼此不受攻击的n个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

- n后问题等价于在n×n格的棋盘上放置n个皇后，任何2个皇后不放在同一行或同一列或同一斜线上。

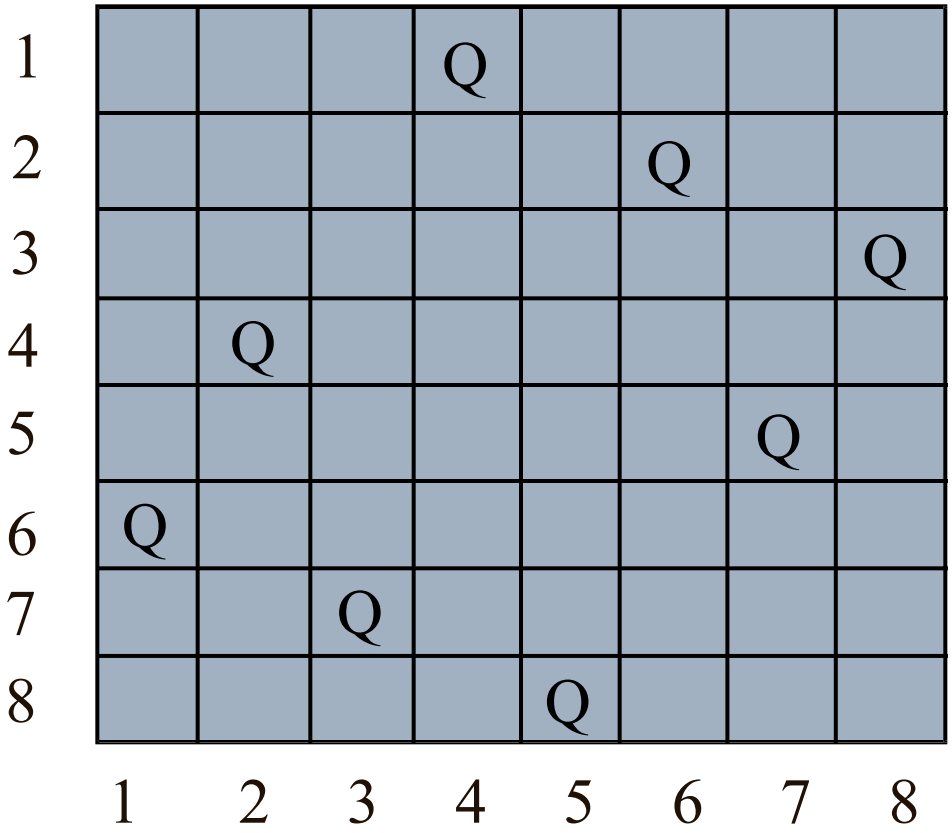
- 如4后问题的2个解





N后问题

8后问题的1个解（共92个解）



N后问题

算法思路:将棋盘从左至右,从上到下编号为1,...,n,皇后编号为1,...,n. 设解为($x1$, ..., xn) , xi为皇后i的列号,且xi的值表示位于第xi行.解空间:E={ ($x1$,..., xn) | $xi \in Si$, i=1,..., n}, Si={1, ..., n}, $1 \le i \le n$. 解空间为排列树.

约束条件D:

1) $x_i \neq x_j$ 皇后i,j不在同一列上

2) $x_i - i \neq x_j - j$

3) $x_i + i \neq x_j + j$

} 皇后i,j不在同一斜线上

N后问题

- 解向量：$(x_1, x_2, \dots, x_n)$
- 显约束：$x_i=1,2, \dots ,n$
- 隐约束：
  - 不同列：$x_i \neq x_j$
  - 不处于同一正、反对角线：$|i-j| \neq |x_i-x_j|$

N后问题

```
bool Queen::Place(int k)
{//判定两个皇后是否在同一斜线或在同一列上
  for (int j=1;j<k;j++)
    if ((abs(k-j)==abs(x[j]-x[k]))||(x[j]==x[k])) return false;
  return true;
}
```

用函数Place(k) 测试待确定的第k皇后是否和前面已确定的k -1个皇后是否在同一条斜角线或同一列上。
第k个皇后只要与任一皇后在同一条斜角线，就返回false，当第k个皇后能放置于X(k)的当前值处时，这个返回值为true。

N后问题

```
void Queen::Backtrack(int t)
{
  if (t>n) sum++;
     else
       for (int i=1;i<=n;i++) {
         x[t]=i;
         if (Place(t)) Backtrack(t+1);
       }
 }
```

N后问题

```
int nQueen(int n)
 {
 QueenX;
 //初始化X
 X. n=n;  //皇后个数
  X. sum=0;
  int*p=new int [n+1];
  for(int i=0;  i<=n;  i++) p[i]= 0;
  X.x=p;
  X.Backtrack(1);
  delete [] p;
  returnX. sum;
}
```

N后问题

- 当皇后数从1到12时各自所对应的可行解的总数如下：

```
n=1       sum=1
n=2       sum=0
n=3       sum=0
n=4       sum=2
n=5       sum=10
n=6       sum=4
n=7       sum=40
n=8       sum=92
n=9       sum=352
n=10      sum=724
n=11      sum=2680
n=12      sum=14200
Press any key to continue
```

8后问题的运行实例（92个可行解）

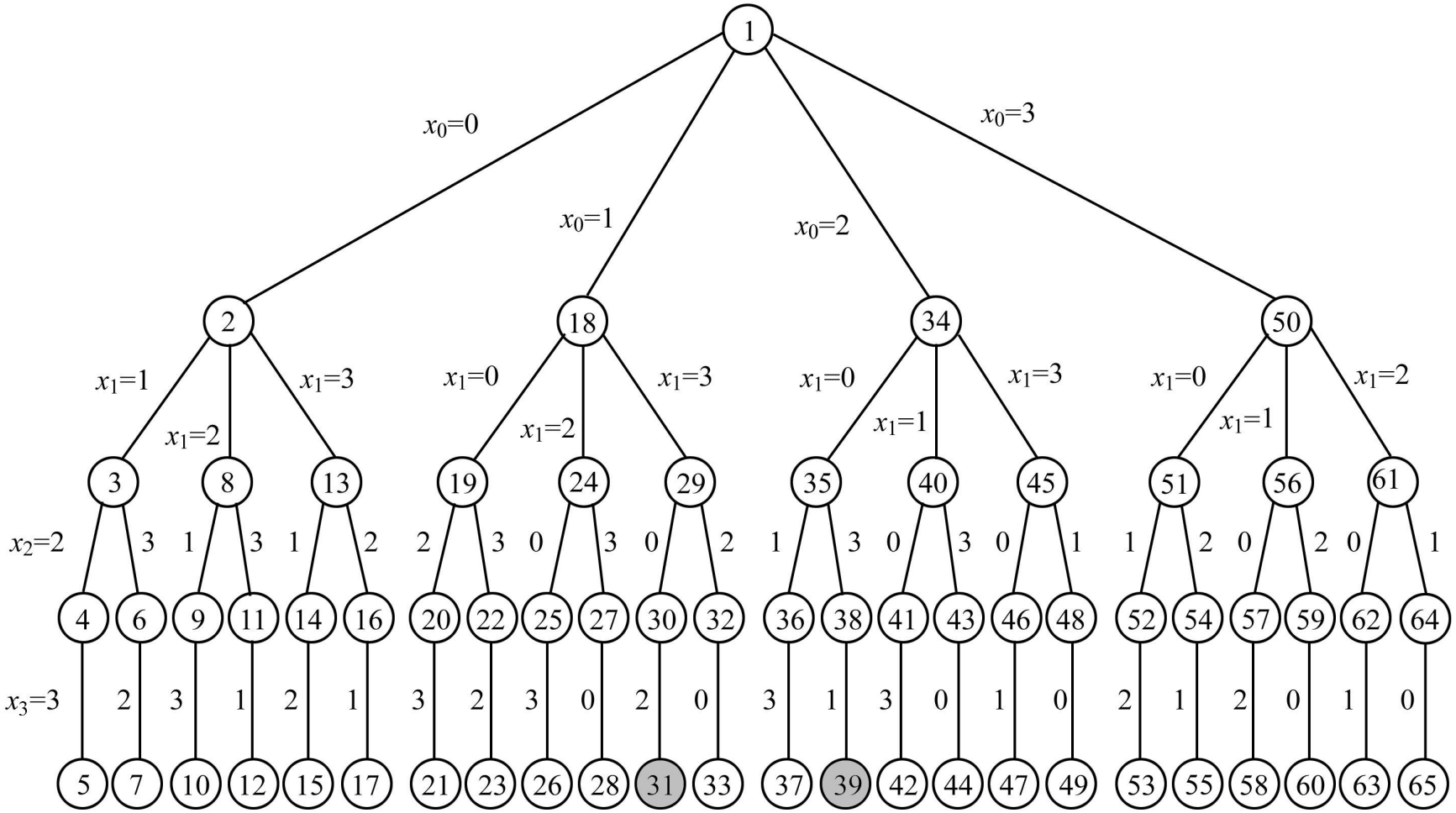
```
      1---------Display all
      2---------Display one by one
      0---------exit
1

Please input the number of queens < Q < (MAX=17):8

 1 5 8 6 3 7 2 4      1 6 8 3 7 4 2 5      1 7 4 6 8 2 5 3      1 7 5 8 2 4 6 3
 2 4 6 8 3 1 7 5      2 5 7 1 3 8 6 4      2 5 7 4 1 8 6 3      2 6 1 7 4 8 3 5
 2 6 8 3 1 4 7 5      2 7 3 6 8 5 1 4      2 7 5 8 1 4 6 3      2 8 6 1 3 5 7 4
 3 1 7 5 8 2 4 6      3 5 2 8 1 7 4 6      3 5 2 8 6 4 7 1      3 5 7 1 4 2 8 6
 3 5 8 4 1 7 2 6      3 6 2 5 8 1 7 4      3 6 2 7 1 4 8 5      3 6 2 7 5 1 8 4
 3 6 4 1 8 5 7 2      3 6 4 2 8 5 7 1      3 6 8 1 4 7 5 2      3 6 8 1 5 7 2 4
 3 6 8 2 4 1 7 5      3 7 2 8 5 1 4 6      3 7 2 8 6 4 1 5      3 8 4 7 1 6 2 5
 4 1 5 8 2 7 3 6      4 1 5 8 6 3 7 2      4 2 5 8 6 1 3 7      4 2 7 3 6 8 1 5
 4 2 7 3 6 8 5 1      4 2 7 5 1 8 6 3      4 2 8 5 7 1 3 6      4 2 8 6 1 3 5 7
 4 6 1 5 2 8 3 7      4 6 8 2 7 1 3 5      4 6 8 3 1 7 5 2      4 7 1 8 5 2 6 3
 4 7 3 8 2 5 1 6      4 7 5 2 6 1 3 8      4 7 5 3 1 6 8 2      4 8 1 3 6 2 7 5
 4 8 1 5 7 2 6 3      4 8 5 3 1 7 2 6      5 1 4 6 8 2 7 3      5 1 8 4 2 7 3 6
 5 1 8 6 3 7 2 4      5 2 4 6 8 3 1 7      5 2 4 7 3 8 6 1      5 2 6 1 7 4 8 3
 5 2 8 1 4 7 3 6      5 3 1 6 8 2 4 7      5 3 1 7 2 8 6 4      5 3 8 4 7 1 6 2
 5 7 1 3 8 6 4 2      5 7 1 4 2 8 6 3      5 7 2 4 8 1 3 6      5 7 2 6 3 1 4 8
 5 7 2 6 3 1 8 4      5 7 4 1 3 8 6 2      5 8 4 1 3 6 2 7      5 8 4 1 7 2 6 3
 6 1 5 2 8 3 7 4      6 2 7 1 3 5 8 4      6 2 7 1 4 8 5 3      6 3 1 7 5 8 2 4
 6 3 1 8 4 2 7 5      6 3 1 8 5 2 4 7      6 3 5 7 1 4 2 8      6 3 5 8 1 4 2 7
 6 3 7 2 4 8 1 5      6 3 7 2 8 5 1 4      6 3 7 4 1 8 2 5      6 4 1 5 8 2 7 3
 6 4 2 8 5 7 1 3      6 4 7 1 3 5 2 8      6 4 7 1 8 2 5 3      6 8 2 4 1 7 5 3
 7 1 3 8 6 4 2 5      7 2 4 1 8 5 3 6      7 2 6 3 1 4 8 5      7 3 1 6 8 5 2 4
 7 3 8 2 5 1 6 4      7 4 2 5 8 1 3 6      7 4 2 8 6 1 3 5      7 5 3 1 6 8 2 4
 8 2 4 1 7 5 3 6      8 2 5 3 1 7 4 6      8 3 1 6 2 5 7 4      8 4 1 3 6 2 7 5

Out the 8 queen's sum is:92
```

4后问题的排列树（共65个结点）



8后问题的效率分析

- 如果用蛮力处理,即在8×8的棋盘上安排出8个位置,这有C_{64}^{8}种安排法,逐一测试,即要检查将近4.4×10^{9}个8元组。
- 用回溯法最多只要作8!次检查,即最多只要查40320个8元组。
- 实际上还少得多，对于8皇后问题，不受限结点是8皇后解空间树的结点总数的2.34%
- 因此,用回溯法处理比用枚举法好得多!

图的m着色问题

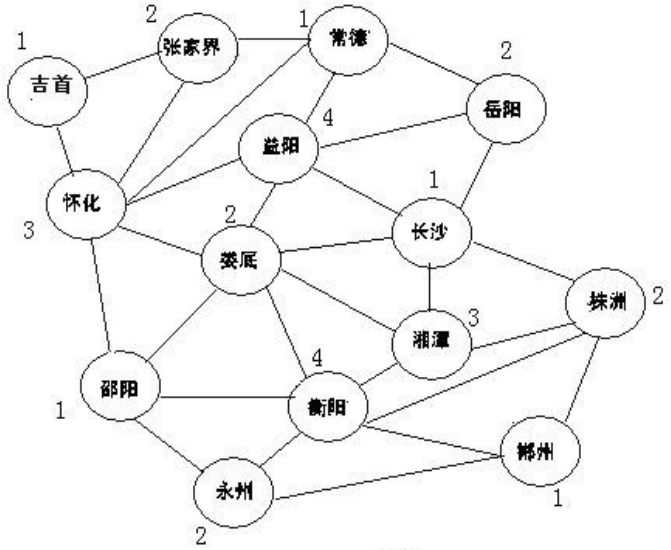
- 图的着色问题是由地图的着色问题引申而来的：用m种颜色为地图着色，使得地图上的每一个区域着一种颜色，且相邻区域颜色不同。

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 。

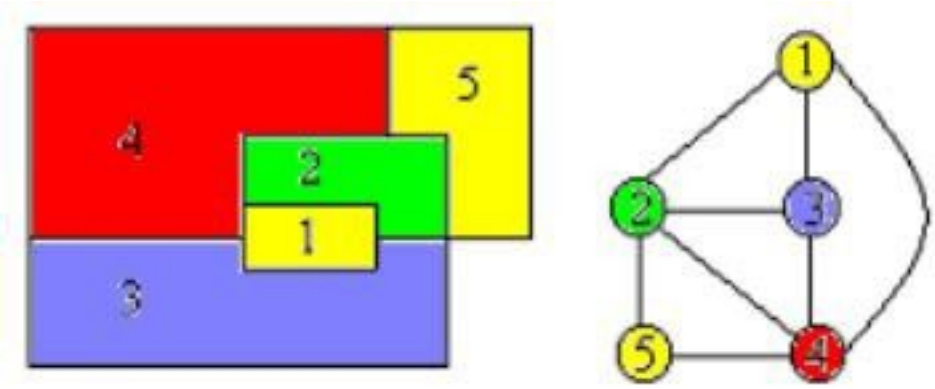
（a）



（b）

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。



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

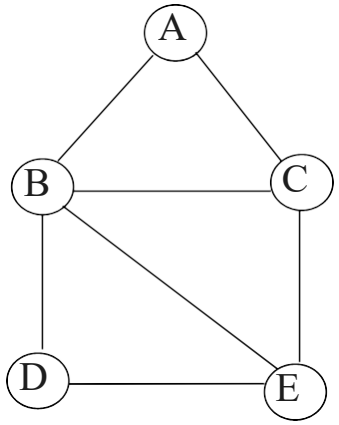
图的m着色问题

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。

图的m着色问题

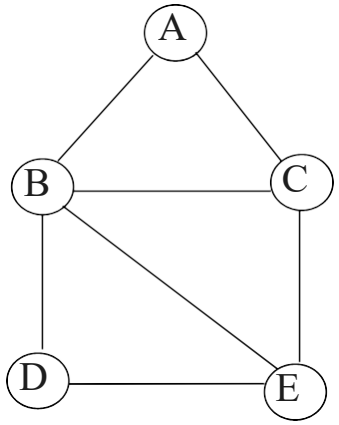
回溯法求解图着色问题示例
(m=3)



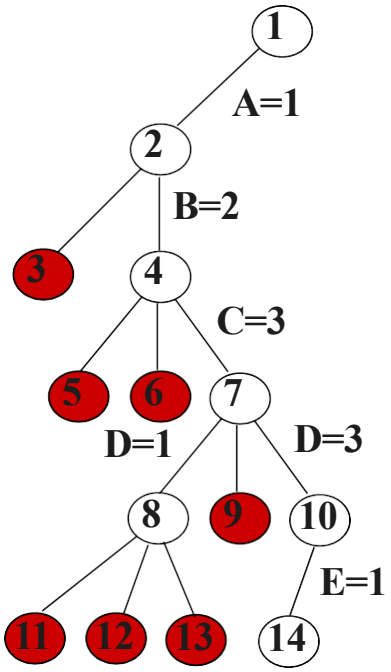
(a) 一个无向图

图的m着色问题

回溯法求解图着色问题示例
(m=3)

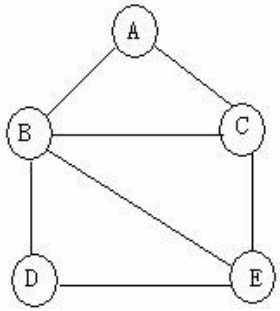


(a) 一个无向图



(b) 回溯法搜索空间

图的m着色问题



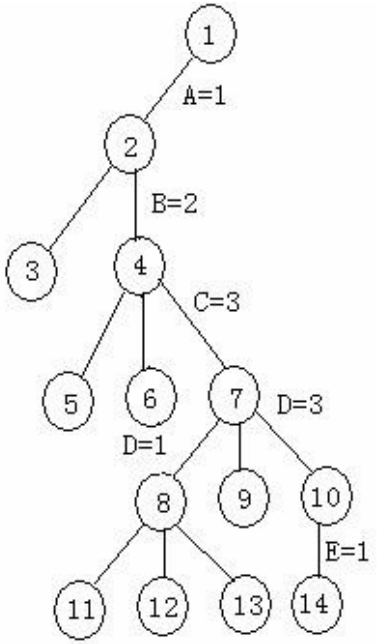
① 把5元组初始化为（0,0,0,0,0），从根结点开始向下搜索，以颜色1为顶点A着色，生成结点2时，产生(1,0,0,0,0)，是个有效着色。

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)。



图的m着色问题

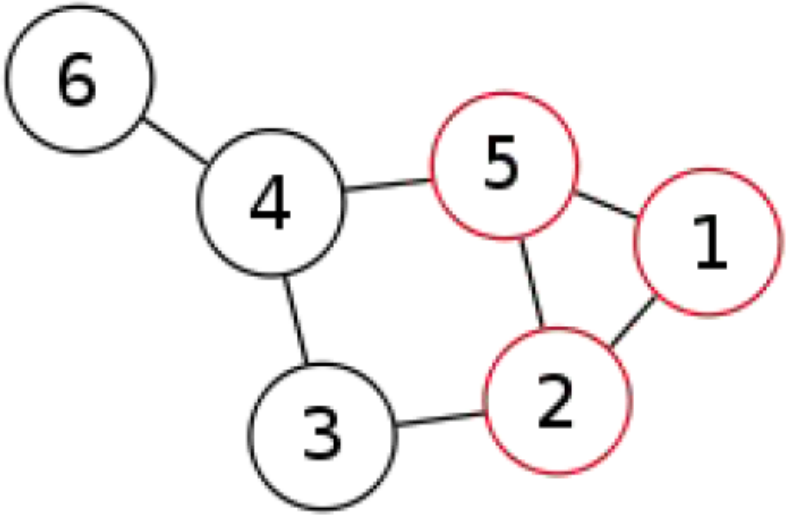
- 设数组color[n]表示顶点的着色情况，回溯法求解m着色问题的算法如下：
  - 1．将数组color[n]初始化为0；
  - 2．k=1；
  - 3. while (k>=1)
    - 3.1 依次考察每一种颜色，若顶点k的着色与其他顶点的着色不发生冲突，则转步骤3.2；否则，搜索下一个颜色；
    - 3.2 若顶点已全部着色，则输出数组color[n]，返回；
    - 3.3 否则
      - 3.3.1 若顶点k是一个合法着色，则k=k+1，转步骤3处理下一个顶点；
      - 3.3.2 否则，重置顶点k的着色情况，k=k-1，转步骤3。

图的m着色问题

```
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;
            }
      }
}
```

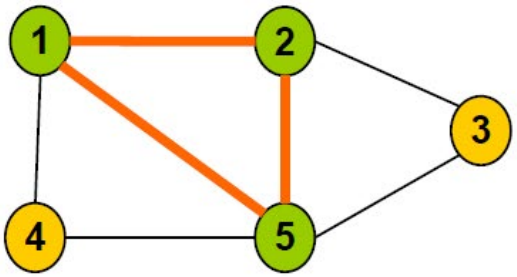
最大团问题

- 团（clique）：社交概念，社会团体，团体中的个体互相认识
- 最大团问题（Maximum Clique Problem, MCP）是图论中一个经典的组合优化问题，也是一类NP完全问题



最大团问题

- 完全子图：给定无向图G=(V，E)。如果U⊆V，且对任意u，v∈U有(u，v) ∈ E，则称U是G的完全子图。
- 团：G的完全子图U是G的团当且仅当U不包含在G的更大的完全子图中
- 最大团：G的最大团是指G中所含顶点数最多的团

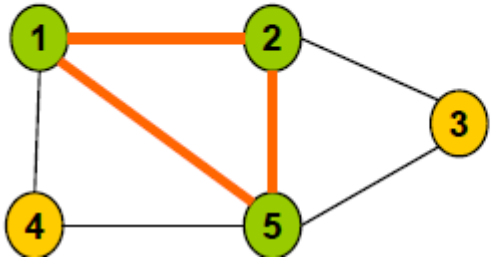


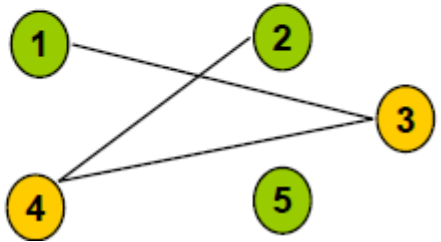
给定无向图G={V，E}，其中V={1, 2, 3, 4, 5}，
E={(1, 2)，(1, 4)，(1, 5)，(2, 3)，(2, 5)，
(3, 5)，(4, 5)}。
根据最大团(MCP)定义，子集{1, 2}是图G的一个大小为2的完全子图，但不是一个团，因为它包含于G的更大的完全子图{1, 2, 5}之中。
{1, 2, 5}是G的一个最大团。
{1, 4, 5}和{2, 3, 5}也是G的最大团。

最大团问题

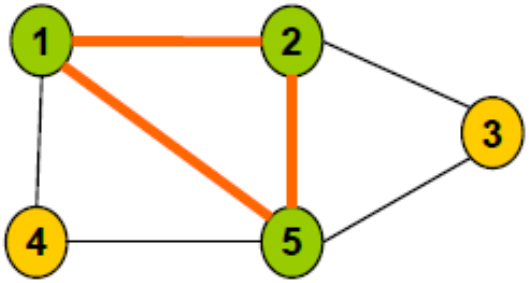
- 空子图与独立集：如果U ⊆ V且对任意u，v∈U有(u，v)∉E，则称U是G的空子图。G的空子图U是G的独立集当且仅当U不包含在G的更大的空子图中。
- 最大独立集：G的最大独立集是G中所含顶点数最多的独立集。
- 补图：对于任一无向图G=(V，E)其补图$\overline{G}$=(V1，E1)定义为：
- V1=V，且(u，v) ∈ E1当且仅当(u，v)∉E。

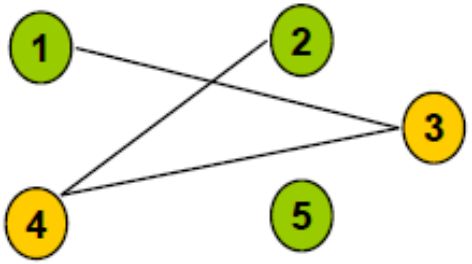
U是G的最大团当且仅当U是$\overline{G}$的最大独立集。





最大团问题





例：右侧图是无向图G的补图G'。
根据最大独立集定义，{2, 4}是G的一个空子图，同时也是G的一个最大独立集。
虽然{1, 2}也是G'的空子图，但它不是G'的独立集，因为它包含在G'的空子图{1, 2, 5}中。
{1, 2, 5}是G'的最大独立集。{1, 4, 5}和{2, 3, 5}也是G'的最大独立集。

最大团问题

- 算法思路：
- • 首先设最大团为一个空团，往其中加入一个顶点，然后依次考虑每个顶点，查看该顶点加入团之后是否仍然构成一个团，如果可以，考虑将该顶点加入团或者舍弃两种情况，如果不行，直接舍弃，然后递归判断下一顶点。
- • 判断当前顶点加入团之后是否仍是一个团：只需要考虑该顶点和团中顶点是否都有连接。
- • 剪枝策略：如果剩余未考虑的顶点数加上团中顶点数不大于当前解的顶点数，可停止继续深度搜索，否则继续深度递归当搜索到一个叶结点时，即可停止搜索，更新最优解和最优值。

最大团问题

- 解空间：子集树
- 可行性约束函数：顶点i到已选入的顶点集中每一个顶点都有边相连。
- 上界函数：有足够多的可选择顶点使得算法有可能在右子树中找到更大的团。

最大团问题

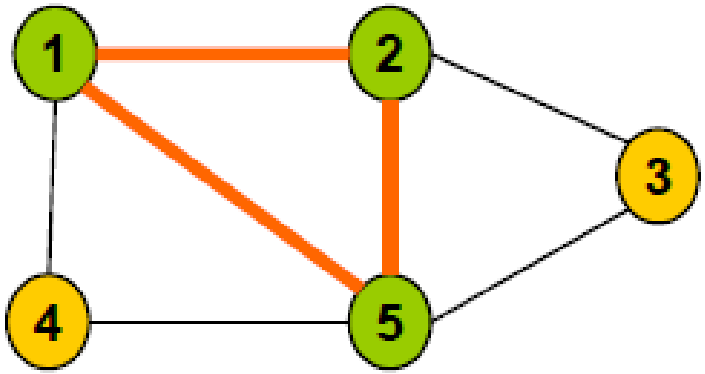
```
void Clique::Backtrack(int i)
{// 计算最大团
if (i > n) {// 到达叶结点
    for (int j = 1; j <= n; j++) bestx[j] = x[j];
    bestn = cn; return;}
// 检查顶点 i 与当前团的连接
int OK = 1;
for (int j = 1; j < i; j++)
    if (x[j] && a[i][j] == 0) {
        // i与j不相连
        OK = 0; break;}
if (OK) {// 进入左子树
    x[i] = 1; cn++;
    Backtrack(i+1);
    x[i] = 0; cn--;}
if (cn + n - i > bestn) {// 进入右子树
    x[i] = 0;
    Backtrack(i+1);}
}
```

复杂度分析

最大团问题的回溯算法backtrack所需的计算时间为$O(n2^n)$。

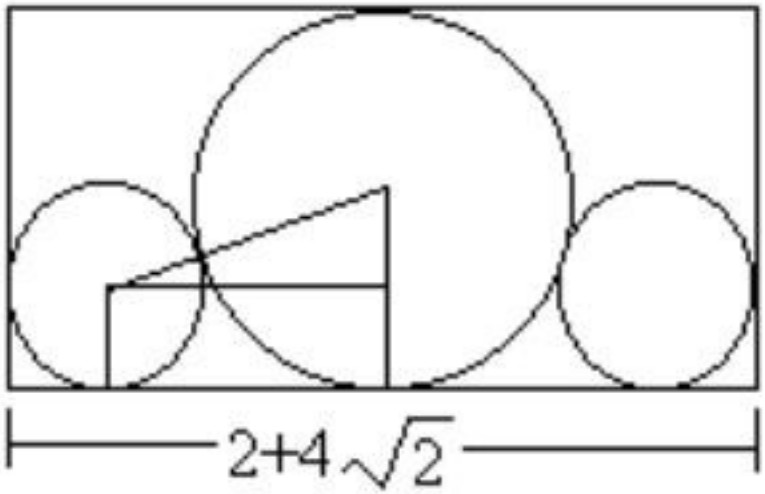
最大团问题

- 请用回溯法对下图求解最大团



圆排列问题

- 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。其最小长度为$2+4\sqrt{2}$



0-1背包问题 – 再思考

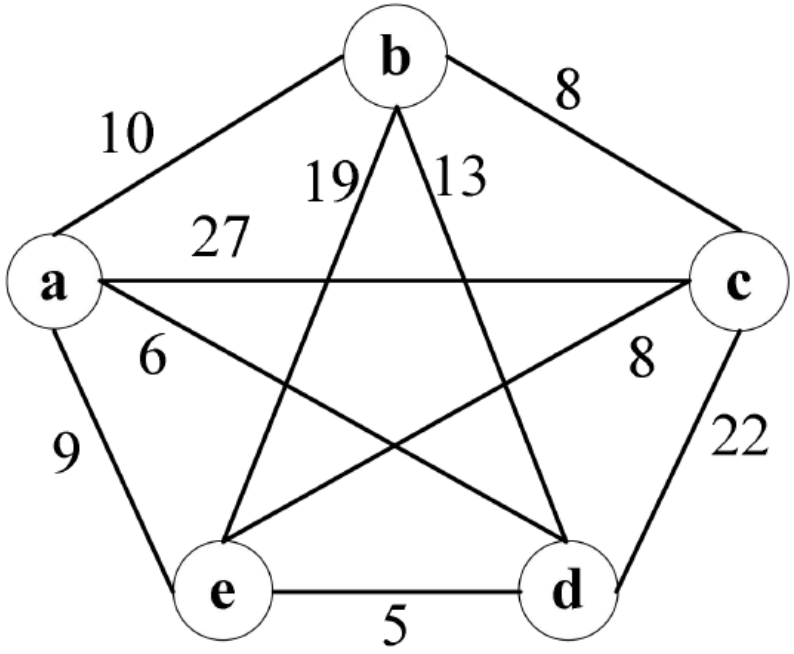
- 0-1背包问题还能否再优化?

0-1背包问题 – 再思考

- 在第四章中,我们用动态规划的方法讨论了背包问题。这里，我们将用回溯的方法求解背包问题。
- 问题的解空间：由各分量 x_i(取值0或1）的2^n个不同的 n元向量组成。与子集和数问题的解空间相同。也用树结构表示（满二叉树）。
- 限界函数：取能产生某些值的上界函数。如果扩展给定活结点和它的任一子孙所导致最好可行解的上界不大于迄今所确定的最好解的值，就可杀死此活结点。
- 0-1背包问题的限界函数：用贪心方法求取上界值。

TSP问题 – 再思考

- 能否再进一步优化?

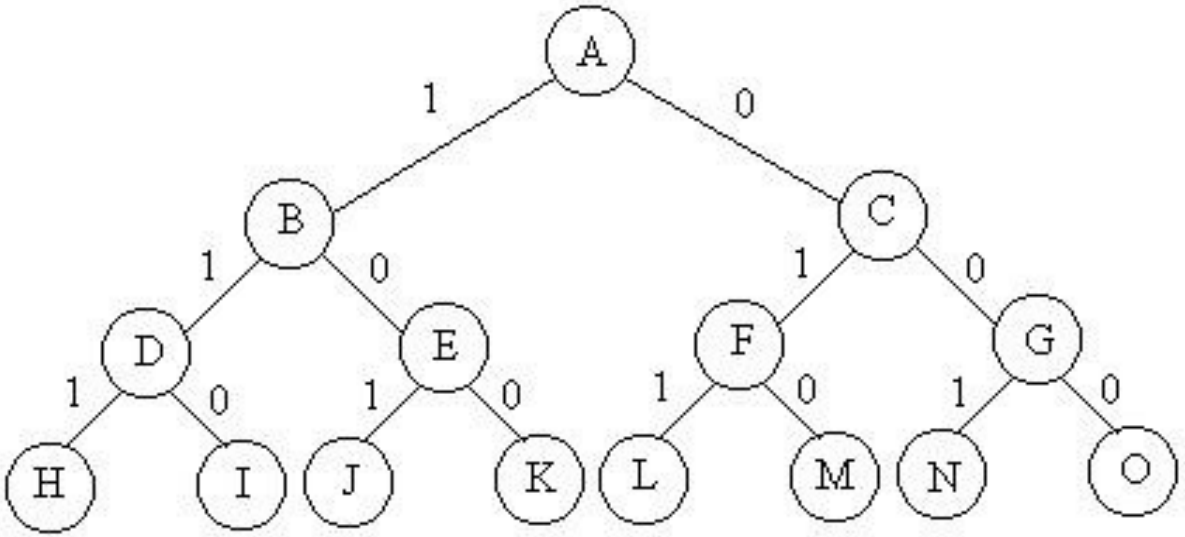


旅行售货员问题

```
template<class Type>
void Traveling<Type>::Backtrack(int i)
{
  if (i == n) { //当前扩展结点是排列树的叶结点的父结点
    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];} }
  else { //当前扩展结点位于排列树的第i-1层
  for (int j = i; j <= n; j++) // 是否可进入x[j]子树?
      if (a[x[i-1]][x[j]] != NoEdge &&
        (cc + a[x[i-1]][x[i]] < bestc || bestc == NoEdge)) { // 搜索子树
        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]);} }
}
```

0-1背包问题

- 解空间：子集树
- 可行性约束函数：$\sum_{i=1}^{n} w_i x_i \leq c_1$
- 上界函数：



0-1背包问题

```
template<class Typew, class Typep> Typep
Knap<Typew, Typep>::Bound(int i)
{ //  (用贪心方法求取上界值)
  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;//右子树中解的上界
}
```

回溯法的效率分析

- 本节要点：
- 一、影响回溯算法效率的因素
- 二、重排原理
- 三、回溯法的效率

影响回溯算法效率的因素

- 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。

影响回溯算法效率的因素

- 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。

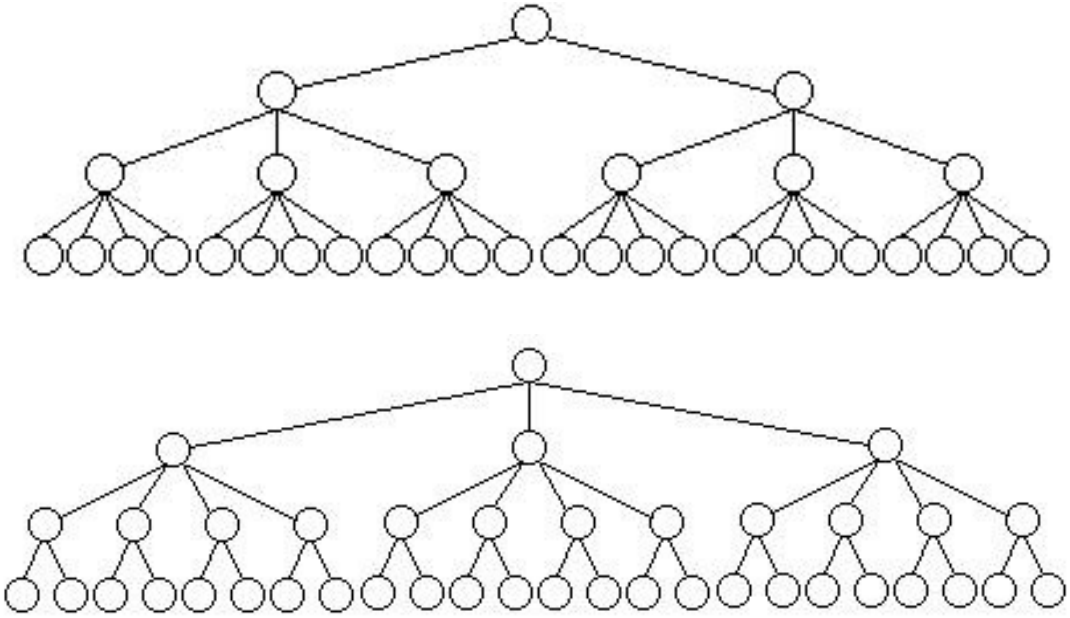
影响回溯算法效率的因素

- 回溯算法的**最坏情况时间复杂度**可达$O(p(n)n!)$（或$O(p(n)2^n)$或$O(p(n)n^n)$），这里$p(n)$是n的多项式，是生成一个结点所需的时间。

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。在其它条件相当的前提下，让可取值最少的x[i]优先。
- 从下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。

重排原理



(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。

回溯法小结

- **回溯法要求**问题P的状态能表达为n元组(x_1，x_2,… x_n)，**要求** $x_i \in s_i$，i =1,2,…n，s_i**为有限集**，**对于给定关于**n元组中的分量的一个约束集D，满足D的全部约束条件的所有n元组为问题P的解。
- 从k=1开始构造k元组，如果k元组满足约束，k=k+1，扩展搜索；如果试探了x[k]的所有值，仍不能满足约束，则 k=k-1，回溯到上一层重新选择x[k-1]的值。这种扩展回溯的搜索方法可以表示为状态空间树上的带约束条件的深度优先的搜索。