電子工業出版社
Publishing House of Electronics Industry

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
- （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

- 通过应用范例学习回溯法的设计策略。
- （1）装载问题；
- （2）批处理作业调度；
- （3）符号三角形问题
- （4）n后问题；
- （5）0-1背包问题；
- （6）最大团问题；
- （7）图的m着色问题
- （8）旅行售货员问题
- （9）圆排列问题
- （10）电路板排列问题
- （11）连续邮资问题

回溯法

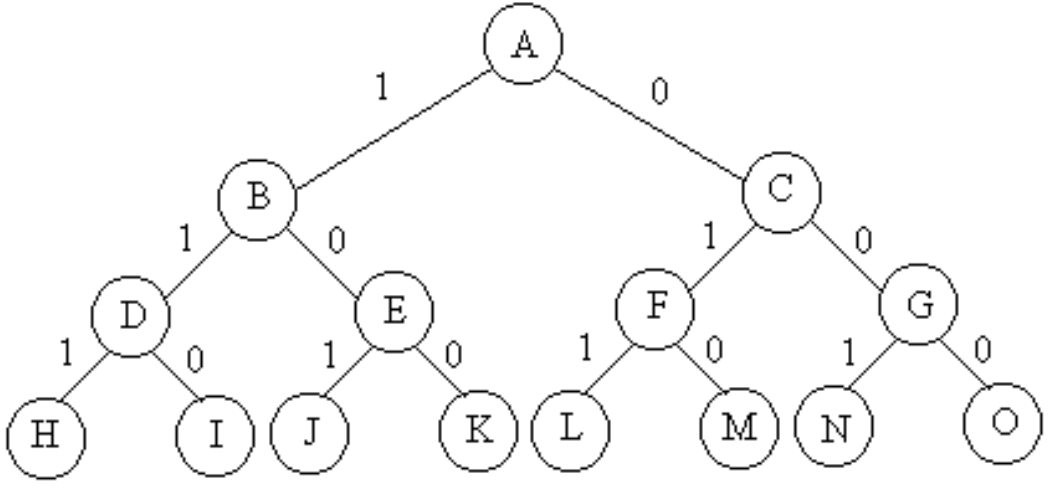
- 有许多问题，当需要找出它的解集或者要求回答什么解是满足某些约束条件的最佳解时，往往要使用回溯法。
- 回溯法的基本做法是搜索，或是一种组织得井井有条的，能避免不必要搜索的穷举式搜索法。这种方法适用于解一些组合数相当大的问题。
- 回溯法在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。算法搜索至解空间树的任意一点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

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$(x_1, x_1, \ldots, x_n)$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。
- 解空间：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。

问题的解空间

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



n=3时的0-1背包问题用完全二叉树表示的解空间

生成问题状态的基本方法

- 扩展结点(E-node):一个正在产生儿子的结点称为扩展结点
- 活结点(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(Dead node):一个所有儿子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）

生成问题状态的基本方法

- 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

回溯法的基本思想

(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。

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。

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。

```
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);
          }
}
```

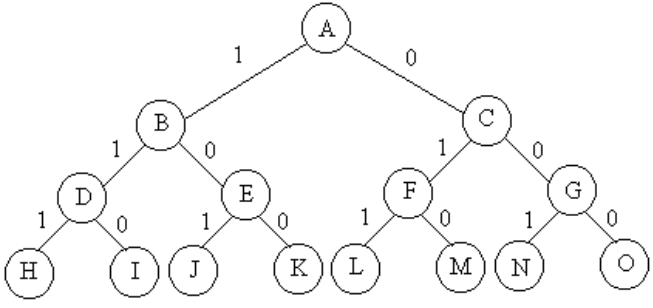
满足约束/限界条件

迭代回溯

采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程。

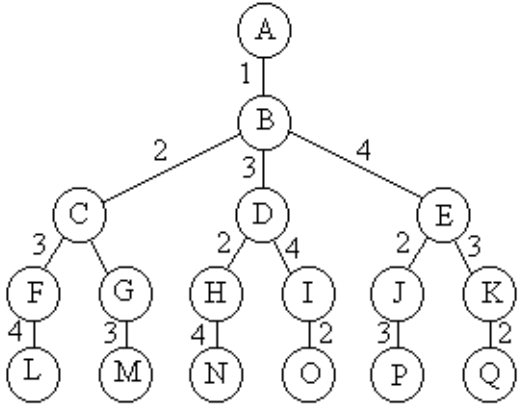
```
void iterativeBacktrack ()
{
  int t=1;
  while (t>0) {
    if (f(n,t)<=g(n,t))
      for (int i=f(n,t);i<=g(n,t);i++) {
        x[t]=h(i);
        if (constraint(t)&&bound(t)) {
          if (solution(t)) output(x);
          else t++;}
        }
    else t--;
    }
}
```

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]);
      }
}
```

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

装载问题要求确定是否有一个合理的装载方案可将这个集装箱装上这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。

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

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。

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

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

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

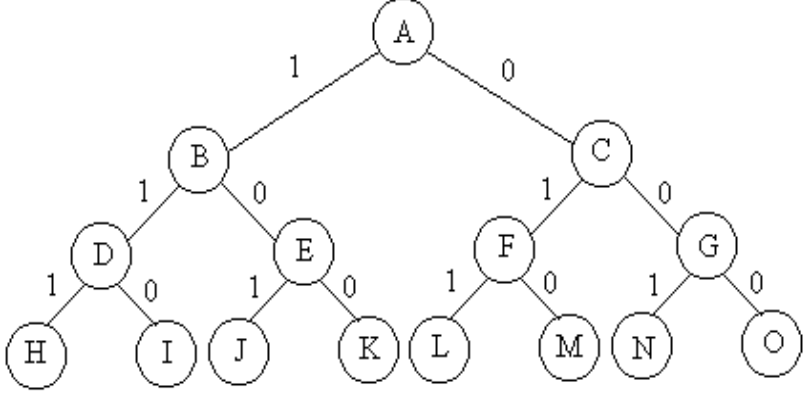
用回溯法设计解装载问题的$O(2^n)$计算时间算法。在某些情况下该算法优于动态规划算法。

装载问题

- 解空间：子集树
- 可行性约束函数(选择当前元素)：$\sum_{i=1}^{n} w_i x_i \le c_1$ 不超过载重量
- 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
    更新最优解bestx,bestw;return;
    r -= w[i];
    if (cw + w[i] <= c) {  // 搜索左子树
      x[i] = 1; cw += w[i];
      backtrack(i + 1);
      cw -= w[i];      }
    if (cw + r > bestw)  {
      x[i] = 0;  // 搜索右子树
      backtrack(i + 1);      }
    r += w[i];
  }
```



批处理作业调度

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

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

批处理作业调度

- 解空间：排列树

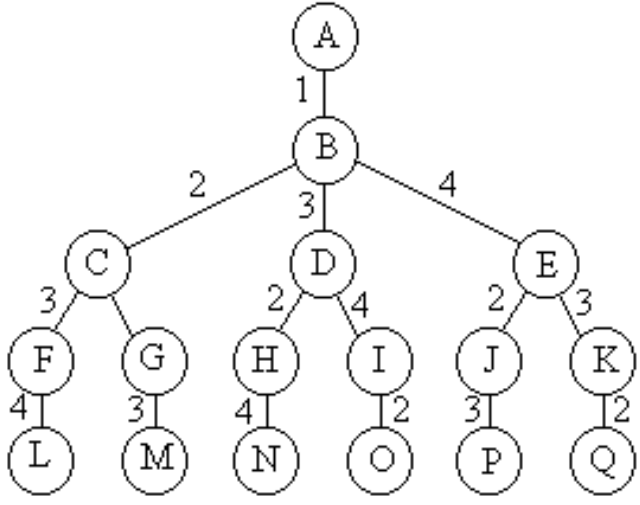
```
class Flowshop {
  friend Flow(int**, int, int []);
  private:
    void Backtrack(int i);
    // 相关的参数
    int  **M,    // 各作业所需的处理时间
           *x,    // 当前作业调度
       *bestx,    // 当前最优作业调度
          *f2,    // 机器2完成处理时间
           f1,    // 机器1完成处理时间
            f,    // 完成时间和
       bestf,  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

批处理作业调度

- 解空间：排列树

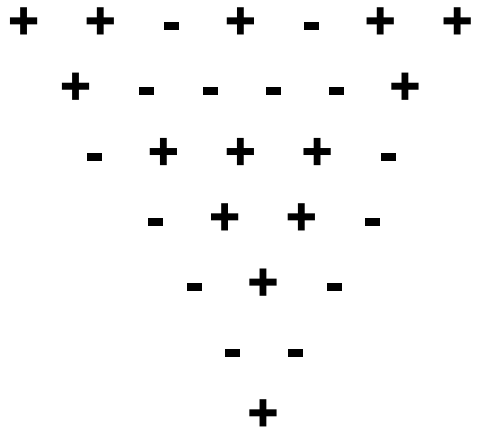
```
void Flowshop::Backtrack(int i)
{
  if (i > n) {
     for (int j = 1; j <= n; j++)
       bestx[j] = x[j];
     bestf = f;
     }
  else
    for (int j = i; j <= n; j++) {
      f1+=M[x[j]][1];
      f2[i]=((f2[i-1]>f1)?f2[i-1]:f1)+M[x[j]][2];
      f+=f2[i];

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



符号三角形问题

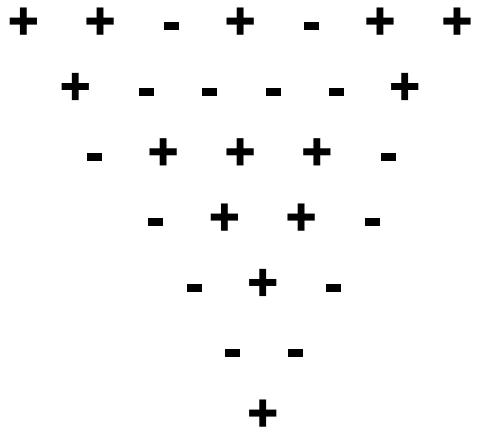
下图是由14个“+”和14个“–”组成的符号三角形。2个同号下面都是“+”，2个异号下面都是“–”。



在一般情况下，符号三角形的第一行有n个符号。符号三角形问题要求对于给定的n，计算有多少个不同的符号三角形，使其所含的“+”和“–”的个数相同。

符号三角形问题

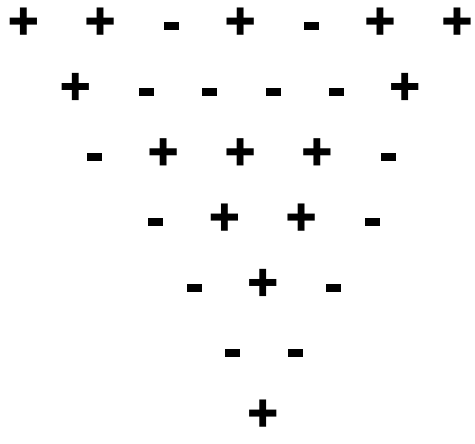
- 解向量：用n元组x[1:n]表示符号三角形的第一行。
- 可行性约束函数：当前符号三角形所包含的“+”个数与“-”个数均不超过n*(n+1)/4
- 无解的判断：n*(n+1)/2为奇数




```
void Triangle::Backtrack(int t)
{
  if ((count>half)||(t*(t-1)/2-count>half)) return;
  if (t>n) sum++;
    else
      for (int i=0;i<2;i++) {
        p[1][t]=i;
        count+=i;
        for (int j=2;j<=t;j++) {
          p[j][t-j+1]=p[j-1][t-j+1]^p[j-1][t-j+2];
          count+=p[j][t-j+1];
        }
      Backtrack(t+1);
      for (int j=2;j<=t;j++)
        count-=p[j][t-j+1];
      count-=i;
  }
}
```

符号三角形问题

- 解向量：用n元组x[1:n]表示符号三角形的第一行。
- 可行性约束函数：当前符号三角形所包含的“+”个数与“-”个数均不超过n*(n+1)/4
- 无解的判断：n*(n+1)/2为奇数

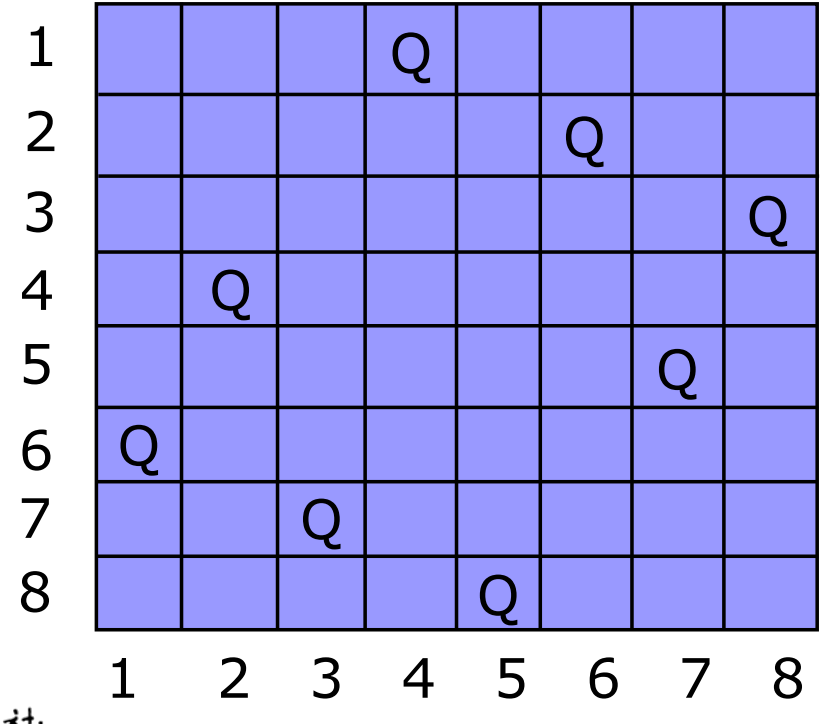


复杂度分析

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

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。



n后问题：找所有解

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

```
bool Queen::Place(int k)
{
  for (int j=1;j<k;j++)
    if ((abs(k-j)==abs(x[j]-x[k]))||(x[j]==x[k])) return false;
  return true;
}

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：找一个解

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

```
int  x[N];  // N个皇后
bool a[N] ,b[2*N-1] , c[2*N-1] ;
x[i]：第i列皇后位置 // 值为0…N-1
a[j]：第j行无皇后时为真
b[k]：第k条 / 对角线上无皇后为真
c[k]：第k条 \ 对角线上无皇后为真
```

位于对角线 / 上：（i,j）,(i+1,j-1),
即i+j一定
位于对角线 \ 上：（i,j）,(i+1,j+1),
即i-j一定

```
void initialize(){
   for(int i=0;i<N;i++){
      x[i]=-1; a[i]=true;     }
   for(int i=0;i<2*N;i++){
      b[i]=true;  c[i]=true;    }
}
```

n后问题：单一解

```
bool  backtrack (int row)  {
    int col=-1;
    bool flag=false;
    do{
        col++; flag=false;
        if (a[col]&&b[row+col]&&c[row-col+(N-1)]){
            a[col]=b[row+col]=c[row-col+N-1]=false;
            x[row]=col;
            if (row<N-1) {
                if (!backtrack(row+1)){
                    a[col]=b[row+col]=c[row-col+N-1]=true;
                } else  flag=true;
            }else flag=true;
        }
    }while((!flag)&&col<N-1);
    return flag;
}
```

Take a step

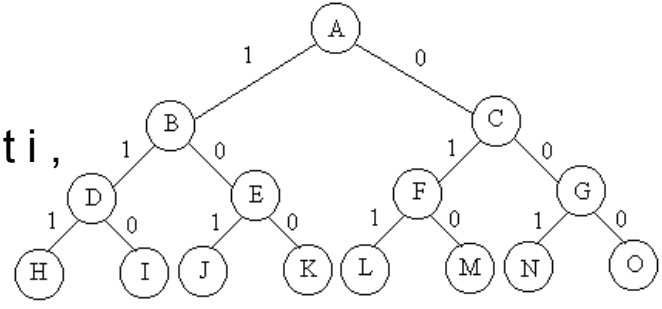
Cancel the step

Call backtrack(0)!

0-1背包问题

- 解空间：子集树
- 递归过程

```
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::backtrack(int i ,
Typep cp, Typew cw)
{ //cw: current weight, cp: current profit
    int j;
    if(i>=n) {// to leaf node
        if(cp>bestp) {
            bestp=cp;
            for(i=0;i<=n;i++) bestx[i]=x[i];
            return;
        }
    }
```

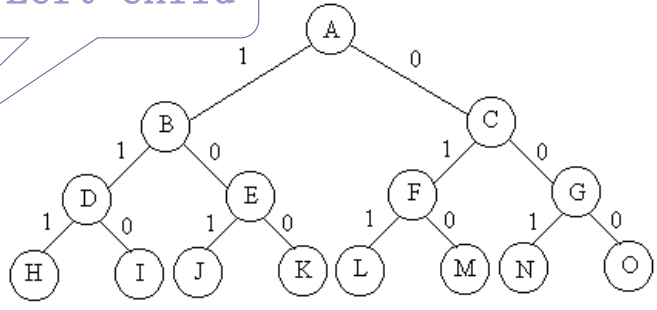


0-1背包问题

- 解空间：子集树
- 递归过程

```
//backtrack(int i , Typep cp, Typew cw)
//
    //  put ith item  into the knapsack
    if(cw+w[i]<=c) {
        cw+=w[i]; cp+=p[i]; x[i]=1;
        backtrack(i+1,cp,cw);
        cw-=w[i]; cp-=p[i]; x[i]=0;
    }
    //  do not put ith item  into the knapsack
    if(bound(i+1,cp,cw)>bestp)
        backtrack(i+1,cp,cw);
}
```

Left child



Cancel the step

Right child

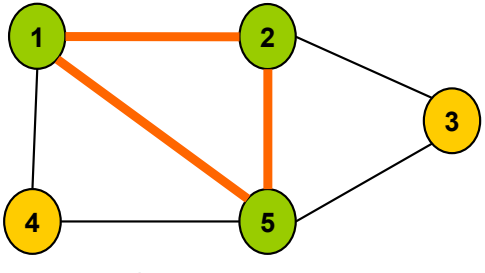
最大团问题

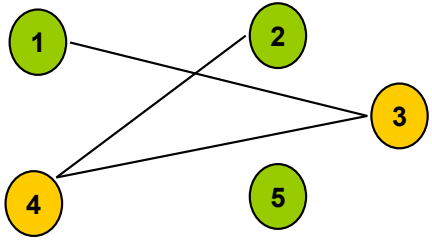
给定无向图G=(V，E)。如果U⊆V，且对任意u，v∈U有(u，v)∈E，则称U是G的**完全子图**。G的完全子图U是G的团当且仅当U不包含在G的更大的完全子图中。G的**最大团**是指G中所含顶点数最多的团。

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。G的**最大独立集**是G中所含顶点数最多的独立集。

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。

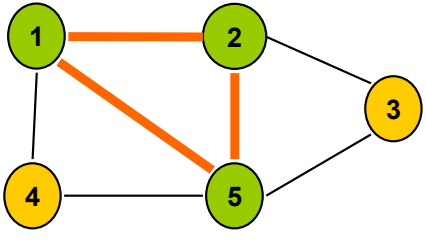
U是G的最大团当且仅当U是$\overline{G}$的最大独立集。





最大团问题

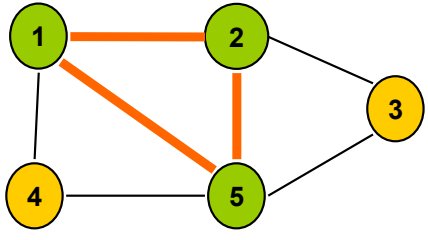
- 解空间：子集树
- 可行性约束函数：顶点i到已选入的顶点集中每一个顶点都有边相连。
- 上界函数：有足够多的可选择顶点使得算法有可能在右子树中找到更大的团。




```
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;}
}
```

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。




```
//void Clique::Backtrack(int i)
    .......
  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

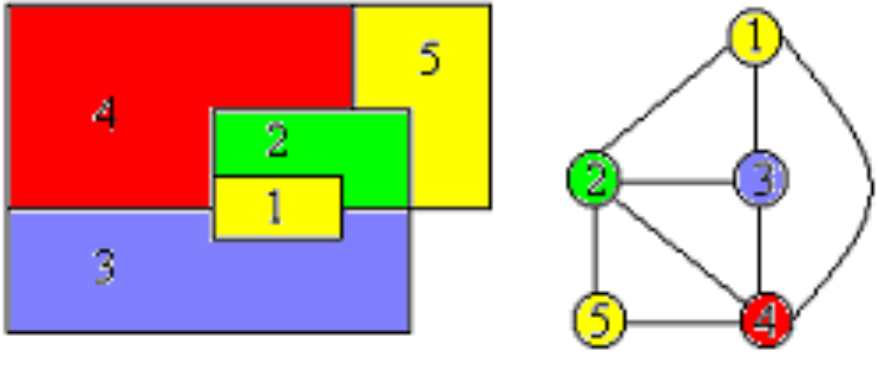
最大团问题的回溯算法**backtrack**所需的计算时间显然为$O(n2^n)$。

进一步改进

- 选择合适的搜索顺序，可以使得上界函数更有效的发挥作用。例如在搜索之前可以将顶点按度从小到大排序。这在某种意义上相当于给回溯法加入了启发性。
- 定义Si={v_i,v_{i+1},…,v_n}，依次求出S_n,S_{n-1},…,S_1的解。从而得到一个更精确的上界函数，若cn+S_i<=max则剪枝。同时注意到：从S_{i+1}到S_i，如果找到一个更大的团，那么v_i必然属于找到的团，此时有S_i=S_{i+1}+1，否则S_i=S_{i+1}。因此只要max的值被更新过，就可以确定已经找到最大值，不必再往下搜索了。

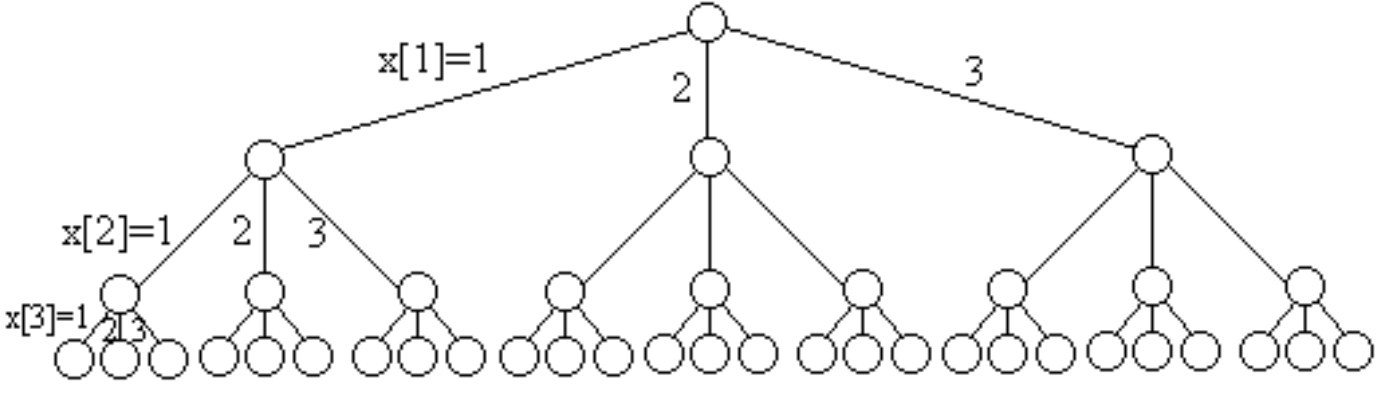
图的m着色问题

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。



图的m着色问题

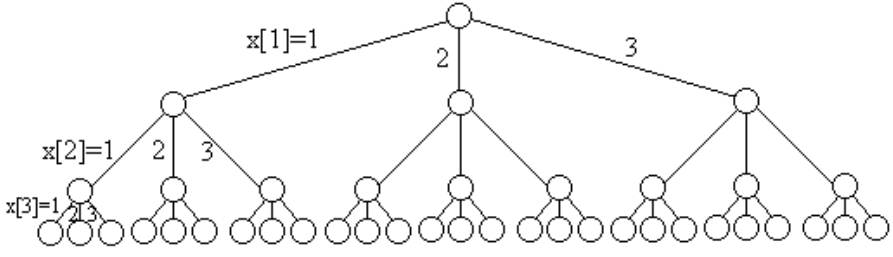
- 解向量：$(x_1, x_2, \ldots, x_n)$表示顶点i所着颜色x[i]
- 可行性约束函数：顶点i与已着色的相邻顶点颜色不重复。



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。

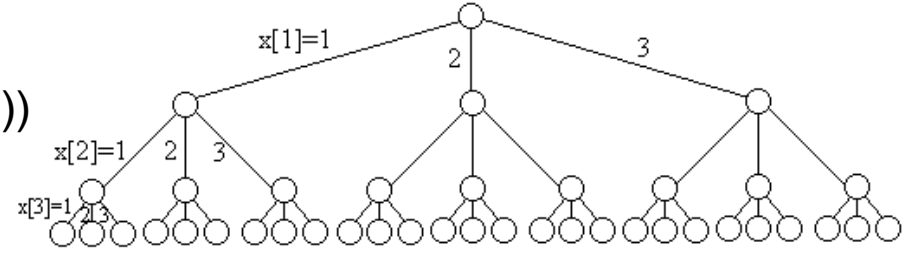
```
void Color::Backtrack(int t)
{
  if (t>n) {
   sum++;
   for (int i=1; i<=n; i++)
     cout << x[i] << ' ';
   cout << endl;
   }
    else
      for (int i=1;i<=m;i++) {
        x[t]=i;
        if (Ok(t)) Backtrack(t+1);
       }
}
```



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。

```
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;
}
```



复杂度分析

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

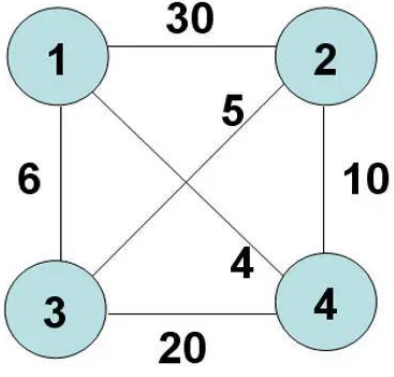
对于每一个内结点，在最坏情况下，用ok检查当前扩展结点的每一个儿子所相应的颜色可用性需耗时O(mn)。因此，回溯法总的时间耗费是

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

旅行售货员问题TSP

- 某售货员要到若干城市去推销商品，已知各城市之间的路程，他要选定一条从驻地出发，经过每个城市一遍，最后回到住地的路线，使总的路程最短。

- 如下面的图，最短路径为：
- 1→4 → 2 →3 →1
- 路程为4+10+5+6=25



旅行售货员问题

- 解空间：排列树

```
template<class Type>
void Traveling<Type>::Backtrack(int i)
{
   if (i == n) {
      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 {
```

旅行售货员问题

- 解空间：排列树

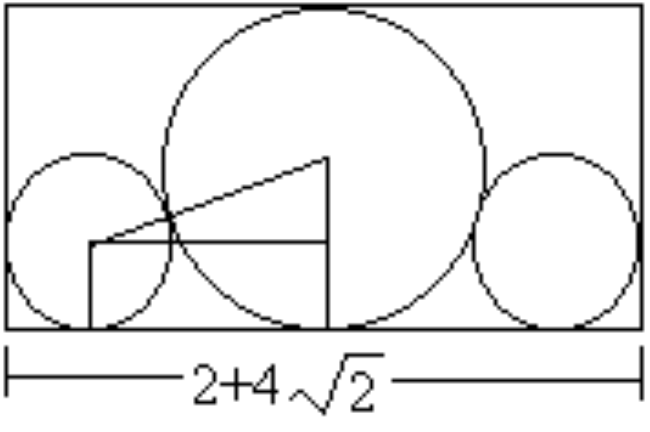
```
//template<class Type>
//void Traveling<Type>::Backtrack(int i)
    for (int j = i; j <= n; j++)
        // 是否可进入x[j]子树?
        if (a[x[i-1]][x[j]] != NoEdge &&
           (cc + a[x[i-1]][x[i]] < bestc || bestc == NoEdge)) {
           // 搜索子树
           Swap(x[i], x[j]);
           cc += a[x[i-1]][x[i]];
           Backtrack(i+1);
           cc -= a[x[i-1]][x[i]];
           Swap(x[i], x[j]);}
    }
}
```

复杂度分析

算法**backtrack**在最坏情况下可能需要更新当前最优解O((n-1)!)次，每次更新bestx需计算时间O(n)，从而整个算法的计算时间复杂性为O(n!)。

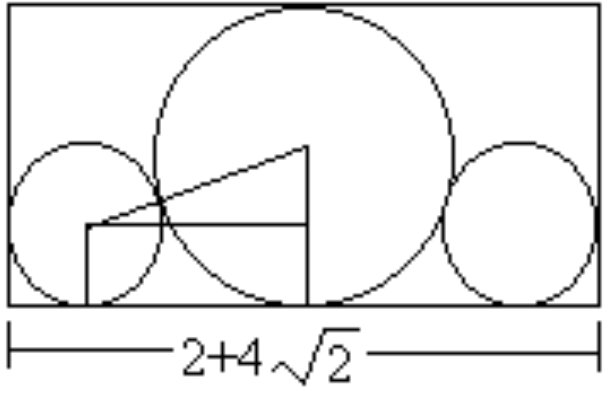
圆排列问题

给定n个大小不等的圆c1, c2, …, cn，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$2+4\sqrt{2}$



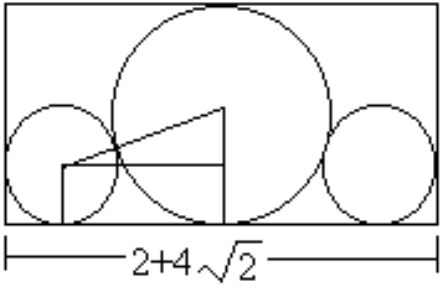
圆排列问题

```
void Circle::Backtrack(int t)
{
    if (t>n) Compute();
    else
      for (int j = t; j <= n; j++)
            Swap(r[t], r[j]);
            float centerx=Center(t);
            if (centerx+r[t]+r[1]<min) {//下界约束
               x[t]=centerx;
               Backtrack(t+1);
            }
            Swap(r[t], r[j]);
            }
}
```



圆排列问题

```
void Circle::Backtrack(int t)
{
    if (t>n) Compute();
    else
      for (int j = t; j <= n; j++) {
            Swap(r[t], r[j]);
            float centerx=Center(t);
            if (centerx+r[t]+r[1]<min) {//下界约束
               x[t]=centerx;
               Backtrack(t+1);
            }
            Swap(r[t], r[j]);
            }
}
```

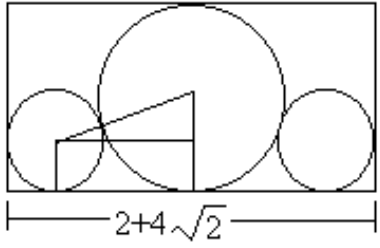



```
float Circle::Center(int t)
{// 计算当前所选择圆的圆心横坐标
    float temp=0;
    for (int j=1;j<t;j++) {
        float valuex=x[j]+2.0*sqrt(r[t]*r[j]);
        if (valuex>temp) temp=valuex;
        }
    return temp;
}
```

```
void Circle::Compute(void)
{// 计算当前圆排列的长度
    float low=0,
          high=0;
    for (int i=1;i<=n;i++) {
        if (x[i]-r[i]<low) low=x[i]-r[i];
        if (x[i]+r[i]>high) high=x[i]+r[i];
        }
    if (high-low<min) min=high-low;
}
```

圆排列问题

```
void Circle::Backtrack(int t)
{
    if (t>n) Compute();
    else
      for (int j = t; j <= n; j++) {
            Swap(r[t], r[j]);
            float centerx=Center(t);
            if (centerx+r[t]+r[1]<min) {//下界约束
              x[t]=centerx;
              Backtrack(t+1);
              }
            Swap(r[t], r[j]);
            }
}
```

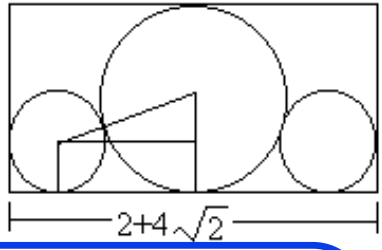


复杂度分析

由于算法**backtrack**在最坏情况下可能需要计算O(n!)次当前圆排列长度，每次计算需O(n)计算时间，从而整个算法的计算时间复杂性为O((n+1)!)

圆排列问题

```
void Circle::Backtrack(int t)
{
    if (t>n) Compute();
    else
      for (int j = t; j <= n; j++) {
          Swap(r[t], r[j]);
```



复杂度分析

由于算法**backtrack**在最坏情况下可能需要计算O(n!)次当前圆排列长度，每次计算需O(n)计算时间，从而整个算法的计算时间复杂性为O((n+1)!)

```
}
```

- 上述算法尚有许多改进的余地。例如，象1,2,…,n-1,n和n,n-1, …,2,1这种互为镜像的排列具有相同的圆排列长度，只计算一个就够了，可减少约一半的计算量。另一方面，如果所给的n个圆中有k个圆有相同的半径，则这k个圆产生的k!个完全相同的圆排列，只计算一个就够了。

连续邮资问题

假设国家发行了n种不同面值的邮票，并且规定每张信封上最多只允许贴m张邮票。连续邮资问题要求对于给定的n和m的值，给出邮票面值的最佳设计，在1张信封上可贴出从邮资1开始，增量为1的最大连续邮资区间。

例如，当n=5和m=4时，面值为(1,3,11,15,32)的5种邮票可以贴出邮资的最大连续邮资区间是1到70。

连续邮资问题

- 解向量：用n元组x[1:n]表示n种不同的邮票面值，并约定它们从小到大排列。x[1]=1是唯一的选择。
- 可行性约束函数：已选定x[1:i-1]，最大连续邮资区间是[1:r]，接下来x[i]的可取值范围是[x[i-1]+1:r+1]。

如何确定r的值？

计算X[1:i]的最大连续邮资区间在本算法中被频繁使用到，因此势必要找到一个高效的方法。考虑到直接递归的求解复杂度太高，我们不妨尝试计算用不超过m张面值为x[1:i]的邮票贴出邮资k所需的最少邮票数y[k]。通过y[k]可以很快推出r的值。事实上，y[k]可以通过递推在O(n)时间内解决：

```
for (int j=0; j<= x[i-2]*(m-1);j++)
        if (y[j]<m)
          for (int k=1;k<=m-y[j];k++)
            if (y[j]+k<y[j+x[i-1]*k]) y[j+x[i-1]*k]=y[j]+k;
      while (y[r]<maxint) r++;
```

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；

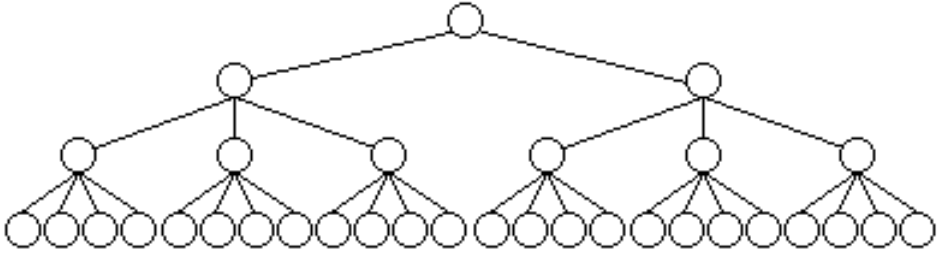
(4)计算上界函数**bound**的时间；

(5)满足约束函数和上界函数约束的所有x[k]的个数。

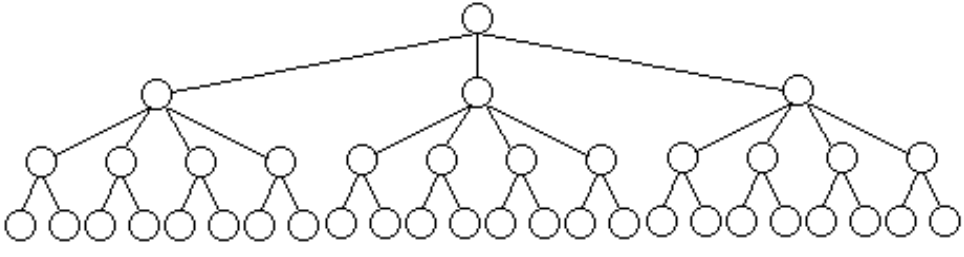
好的约束函数能显著地减少所生成的结点数。但这样的约束函数往往计算量较大。因此，**在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷。**

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。**在其它条件相当的前提下，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。

The End