搜　索　算　法

搜索算法是利用计算机的高性能来有目的的穷举一个问题的部分或所有的可能情况，从而求出问题的解的一种方法。搜索过程实际上是根据初始条件和扩展规则构造一棵解答树并寻找符合目标状态的节点的过程。

所有的搜索算法从其最终的算法实现上来看，都可以划分成两个部分——控制结构和产生系统，而所有的算法的优化和改进主要都是通过修改其控制结构来完成的。现在主要对其控制结构进行讨论，因此对其产生系统作如下约定：

Function ExpendNode(Situation:Tsituation;ExpendWayNo:Integer):TSituation;

表示对给出的节点状态 Sitution 采用第 ExpendWayNo 种扩展规则进行扩展，并且返回扩展后的状态。

（本文所采用的算法描述语言为类 Pascal。）

第一部分　基本搜索算法

一、回溯算法

回溯算法是所有搜索算法中最为基本的一种算法，其采用了一种“走不通就掉头”思想作为其控制结构，其相当于采用了先根遍历的方法来构造解答树，可用于找解或所有解以及最优解。具体的算法描述如下：

```
[非递归算法]
<Type>
   Node(节点类型)=Record
     Situtation:TSituation（当前节点状态）;
     Way-NO:Integer（已使用过的扩展规则的数目）;
   End
<Var>
   List(回溯表):Array[1..Max(最大深度)] of Node;
   pos(当前扩展节点编号):Integer;
<Init>
   List<-0;
   pos<-1;
   List[1].Situation<-初始状态;
<Main Program>
While (pos>0(有路可走)) and ([未达到目标]) do
Begin
  If pos>=Max then (数据溢出,跳出主程序);
  List[pos].Way-NO:=List[pos].Way-No+1;
  If (List[pos].Way-NO<=TotalExpendMethod) then (如果还有没用过的扩展规则)
  Begin
    If (可以使用当前扩展规则) then
    Begin
      (用第 way 条规则扩展当前节点)
      List[pos+1].Situation:=ExpendNode(List[pos].Situation, List[pos].Way-NO);
```

```
      List[pos+1].Way-NO:=0;
      pos:=pos+1;
    End-If;
  End-If
  Else Begin
         pos:=pos-1;
       End-Else
End-While;
```

[递归算法]

```
Procedure BackTrack(Situation:TSituation;deepth:Integer);
Var  I    :Integer;
Begin
  If deepth>Max then (空间达到极限,跳出本过程);
  If Situation=Target then (找到目标);
  For I:=1 to TotalExpendMethod do
  Begin
    BackTrack(ExpendNode(Situation,I),deepth+1);
  End-For;
End;
```

范例：

一个 M*M 的棋盘上某一点上有一个马，要求寻找一条从这一点出发不重复的跳完棋盘上所有的点的路线。

评价：

回溯算法对空间的消耗较少，当其与分枝定界法一起使用时，对于所求解在解答树中层次较深的问题有较好的效果。但应避免在后继节点可能与前继节点相同的问题中使用，以免产生循环。

二、深度搜索与广度搜索

深度搜索与广度搜索的控制结构和产生系统很相似，唯一的区别在于对扩展节点选取上。由于其保留了所有的前继节点，所以在产生后继节点时可以去掉一部分重复的节点，从而提高了搜索效率。这两种算法每次都扩展一个节点的所有子节点，而不同的是，深度搜索下一次扩展的是本次扩展出来的子节点中的一个，而广度搜索扩展的则是本次扩展的节点的兄弟节点。在具体实现上为了提高效率,所以采用了不同的数据结构。

[广度搜索]

```
<Type>
   Node(节点类型)＝Record
     Situtation:TSituation（当前节点状态）;
     Level:Integer(当前节点深度);
     Last :Integer(父节点);
   End
<Var>
   List(节点表):Array[1..Max(最多节点数)] of Node(节点类型);
   open(总节点数):Integer;
   close(待扩展节点编号):Integer;
   New-S:TSituation;(新节点)
```

```
<Init>
   List<-0;
   open<-1;
   close<-0;
   List[1].Situation<- 初始状态;
   List[1].Level:=1;
   List[1].Last:=0;
<Main Program>
   While (close<open(还有未扩展节点)) and
         (open<Max(空间未用完)) and
         (未找到目标节点) do
   Begin
         close:=close+1;
         For I:=1 to TotalExpendMethod do（扩展一层子节点）
         Begin
           New-S:=ExpendNode(List[close].Situation,I);
           If Not (New-S in List) then
           (扩展出的节点从未出现过)
           Begin
             open:=open+1;
             List[open].Situation:=New-S;
             List[open].Level:=List[close].Level+1;
             List[open].Last:=close;
           End-If
         End-For;
   End-While;
[深度搜索]
<Var>
   Open:Array[1..Max] of Node;(待扩展节点表)
   Close:Array[1..Max] of Node;(已扩展节点表)
   openL,closeL:Integer;(表的长度)
   New-S:Tsituation;(新状态)
<Init>
   Open<-0; Close<-0;
   OpenL<-1;CloseL<-0;
   Open[1].Situation<- 初始状态;
   Open[1].Level<-1;
   Open[1].Last<-0;
<Main Program>
   While (openL>0) and (closeL<Max) and (openL<Max) do
   Begin
     closeL:=closeL+1;
     Close[closeL]:=Open[openL];
     openL:=openL-1;
```

```
    For I:=1 to TotalExpendMethod do（扩展一层子节点）
    Begin
      New-S:=ExpendNode(Close[closeL].Situation,I);
      If Not (New-S in List) then
      (扩展出的节点从未出现过)
      Begin
        openL:=openL+1;
        Open[openL].Situation:=New-S;
        Open[openL].Level:=Close[closeL].Level+1;
        Open[openL].Last:=closeL;
      End-If
    End-For;
  End;
```

范例：

迷宫问题，求解最短路径和可通路径。

评价：

广度搜索是求解最优解的一种较好的方法，在后面将会对其进行进一步的优化。而深度搜索多用于只要求解，并且解答树中的重复节点较多并且重复较难判断时使用，但往往可以用**A***或回溯算法代替。

第二部分　搜索算法的优化

一、双向广度搜索

广度搜索虽然可以得到最优解，但是其空间消耗增长太快。但如果从正反两个方向进行广度搜索，理想情况下可以减少二分之一的搜索量，从而提高搜索速度。

范例：

有 N 个黑白棋子排成一派，中间任意两个位置有两个连续的空格。每次空格可以与序列中的某两个棋子交换位置，且两子的次序不变。要求出入长度为 length 的一个初始状态和一个目标状态，求出最少的转化步数。

问题分析：

该题要求求出最少的转化步数，但如果直接使用广度搜索，很容易产生数据溢出。但如果从初始状态和目标状态两个方向同时进行扩展，如果两棵解答树在某个节点第一次发生重合，则该节点所连接的两条路径所拼成的路径就是最优解。

对广度搜索算法的改进：

1.添加一张节点表，作为反向扩展表。

2.在 while 循环体中在正向扩展代码后加入反向扩展代码，其扩展过程不能与正向过程共享一个 for 循环。

3.在正向扩展出一个节点后，需在反向表中查找是否有重合节点。反向扩展时与之相同。

对双向广度搜索算法的改进：

略微修改一下控制结构，每次 while 循环时只扩展正反两个方向中节点数目较少的一个，可以使两边的发展速度保持一定的平衡，从而减少总扩展节点的个数，加快搜索速度。

二、分支定界

分支定界实际上是 A*算法的一种雏形，其对于每个扩展出来的节点给出一个预期值，如果这个预期值不如当前已经搜索出来的结果好的话，则将这个节点(包括其子节点)从解答树中删去，从而达到加快搜索速度的目的。

范例：

在一个商店中购物，设第 I 种商品的价格为 Ci。但商店提供一种折扣，即给出一组商品的组合，如果一次性购买了这一组商品，则可以享受较优惠的价格。现在给出一张购买清单和商店所提供的折扣清单，要求利用这些折扣，使所付款最少。

问题分析：

显然，折扣使用的顺序与最终结果无关，所以可以先将所有的折扣按折扣率从大到小排序，然后采用回溯法的控制结构，对每个折扣从其最大可能使用次数向零递减搜索，设 A 为以打完折扣后优惠的价格，C 为当前未打折扣的商品零售价之和，则其预期值为 A+a*C，其中 a 为下一个折扣的折扣率。如当前已是最后一个折扣，则 a=1。

对回溯算法的改进：

1.添加一个全局变量 BestAnswer，记录当前最优解。

2.在每次生成一个节点时，计算其预期值，并与 BestAnswer 比较。如果不好，则调用回溯过程。

三、A*算法

A*算法中更一般的引入了一个估价函数 f,其定义为 f=g+h。其中 g 为到达当前节点的耗费，而 h 表示对从当前节点到达目标节点的耗费的估计。其必须满足两个条件：

1.h 必须小于等于实际的从当前节点到达目标节点的最小耗费 h*。

2.f 必须保持单调递增。

A*算法的控制结构与广度搜索的十分类似，只是每次扩展的都是当前待扩展节点中 f 值最小的一个，如果扩展出来的节点与已扩展的节点重复，则删去这个节点。如果与待扩展节点重复，如果这个节点的估价函数值较小，则用其代替原待扩展节点，具体算法描述如下：

范例：

一个 3*3 的棋盘中有 1-8 八个数字和一个空格，现给出一个初始态和一个目标态，要求利用这个空格，用最少的步数，使其到达目标态。

问题分析：

预期值定义为 h=|x-dx|+|y-dy|。

估价函数定义为 f=g+h。

```
<Type>
  Node(节点类型)＝Record
    Situtation:TSituation（当前节点状态）;
    g:Integer;(到达当前状态的耗费)
    h:Integer;(预计的耗费)
    f:Real;(估价函数值)
    Last:Integer;(父节点)
  End
```

```
<Var>
    List(节点表):Array[1..Max(最多节点数)] of Node(节点类型);
    open(总节点数):Integer;
    close(待扩展节点编号):Integer;
    New-S:Tsituation;(新节点)
<Init>
    List<-0;
    open<-1;
    close<-0;
    List[1].Situation<- 初始状态;
<Main Program>
    While (close<open(还有未扩展节点)) and
          (open<Max(空间未用完)) and
          (未找到目标节点) do
    Begin
      Begin
        close:=close+1;
        For I:=close+1 to open do (寻找估价函数值最小的节点)
        Begin
          if List[i].f<List[close].f then
          Begin
            交换 List[i]和 List[close];
          End-If;
        End-For;
      End;
      For I:=1 to TotalExpendMethod do（扩展一层子节点）
      Begin
        New-S:=ExpendNode(List[close].Situation,I)
        If Not (New-S in List[1..close]) then
        (扩展出的节点未与已扩展的节点重复)
        Begin
          If Not (New-S in List[close+1..open]) then
          (扩展出的节点未与待扩展的节点重复)
          Begin
            open:=open+1;
            List[open].Situation:=New-S;
            List[open].Last:=close;
            List[open].g:=List[close].g+cost;
            List[open].h:=GetH(List[open].Situation);
            List[open].f:=List[open].h+List[open].g;
          End-If
          Else Begin
                 If List[close].g+cost+GetH(New-S)<List[same].f then
                 (扩展出来的节点的估价函数值小于与其相同的节点)
```

```
                Begin
                  List[same].Situation:= New-S;
                  List[same].Last:=close;
                  List[same].g:=List[close].g+cost;
                  List[same].h:=GetH(List[open].Situation);
                  List[same].f:=List[open].h+List[open].g;
                End-If;
              End-Else;
    End-If
  End-For;
End-While;
```

对 A*算法的改进——分阶段 A*：

当 A*算法出现数据溢出时，从待扩展节点中取出若干个估价函数值较小的节点，然后放弃其余的待扩展节点，从而可以使搜索进一步的进行下去。

第三部分　搜索与动态规划的结合

例 1.

有一个棋子，其 1、6 面 2、4 面 3、5 面相对。现给出一个 M*N 的棋盘，棋子起初处于(1,1)点，摆放状态给定，现在要求用最少的步数从(1,1)点翻滚到(M,N)点，并且 1 面向上。

分析：

这道题目用简单的搜索很容易发生超时，特别当 M、N 较大时。所以可以考虑使用动态规划来解题。对于一个棋子，其总共只有 24 种状态。在(1,1)点时，其向右翻滚至(2,1)点，向上翻滚至(1,2)点。而任意（I，J）点的状态是由（I-1，J）和（I，J-1）点状态推导出来的。所以如果规定棋子只能向上和向右翻滚，则可以用动态规划的方法将到达（M，N）点的所有可能的状态推导出来。显然，从（1，1）到达（N，M）这些状态的路径时最优的。如果这些状态中有 1 面向上的，则已求出解。如果没有，则可以从（M，N）点开始广度搜索，以（M，N）点的状态组作为初始状态，每扩展一步时，检查当前所得的状态组是否有状态与到达格子的状态组中的状态相同，如果有，则由动态规划的最优性和广度搜索的最优性可以保证已求出最优解。

例 2.

给出一个正整数 n，有基本元素 a，要求通过最少次数的乘法，求出 a^n。

分析：

思路一：

这道题从题面上来看非常象一道动态规划题，a^n=a^x1*a^x2。在保证 a^x1 和 a^x2 的最优性之后，a^n 的最优性应该得到保证。但是仔细分析后可以发现，a^x1 与 a^x2 的乘法过程中可能有一部分的重复，所以在计算 a^n 时要减去其重复部分。由于重复部分的计算较繁琐，所以可以将其化为一组展开计算。描述如下：

```
I:=n;(拆分 a^n)
split[n]:=x1;(分解方案)
Used[n]:=True;(在乘法过程中出现的数字)
Repeat(不断分解数字)
```

```
      Used[I-split[I]]:=True;
      Used[split[I]]:=True;
      Dec(I);
      While (I>1) and (not Used[I]) do dec(I);
    Until I=1;
    For I:=n downto 1 do(计算总的乘法次数)
      If Used[I] then count:=count+1;
    Result:=count;(返回乘法次数)
```

思路二：

通过对思路一的输出结果的分析可以发现一个规律：

```
a^19=a^1*a^18
a^18=a^2*a^16
a^16=a^8*a^8
a^8=a^4*a^4
a^4=a^2*a^2
a^2=a*a
```

对于一个 n，先构造一个最接近的 2^k，然后利用在构造过程中产生的 2^I,对 n-2^k 进行二进制分解，可以得出解。对次数的计算的描述如下：

```
count:=0;
Repeat
  If n mod 2 = 0 then count:=count+1
  Else count:=count+2;
  n:=n div 2;
Until n=1;
Result:=count;
```

反思：观察下列数据：

a^15	a^23	a^27
Cost:5	Cost:6	Cost:6
a^2=a^1*a^1	a^2=a^1*a^1	a^2=a^1*a^1
a^3=a^1*a^2	a^3=a^1*a^2	a^3=a^1*a^2
a^6=a^3*a^3	a^5=a^2*a^3	a^6=a^3*a^3
a^12=a^6*a^6	a^10=a^5*a^5	a^12=a^6*a^6
a^15=a^3*a^12	a^20=a^10*a^10	a^24=a^12*a^12
	a^23=a^3*a^20	a^27=a^3*a^24

这些数据都没有采用思路二种的分解方法，而都优于思路二中所给出的解。但是经过实测，思路一二的所有的解的情况相同，而却得不出以上数据中的解。经过对 a^2－a^15 的数据的完全分析，发现对于一个 a^n，存在多个分解方法都可以得出最优解，而在思路一中只保留了一组分解方式。例如对于 a^6 只保留了 a^2*a^4，从而使 a^3*a^3 这条路中断，以至采用思路一的算法时无法得出正确的耗费值，从而丢失了最优解。所以在计算 a^n=a^x1*a^x2 的重复时，要引入一个搜索过程。例如:a^15=a^3*a^12，a^12=a^6*a^6，而 a^6=a^3*a^3。如果采用了 a^6=a^2*a^4，则必须多用一步。

<Type>

```
    Link=^Node;   （使用链表结构纪录所有的可能解）
    Node=Record
      split:Integer;
      next :Link;
    End;
<Var>
    Solution:Array[1..1000] of Link;   （对于 a^n 的所有可能解）
    Cost :Array[1..1000] of Integer;   （解的代价）
    max :Integer;                      （推算的上界）
<Main Program>
Procedure GetSolution;
Var  i,j   :Integer;
    min,c:Integer;
    count:Integer;
    temp,tail:Link;
    plan :Array[1..500] of Integer;
    nUsed:Array[1..1000] of Boolean;
 Procedure GetCost(From,Cost:Integer);   （搜索计算最优解）
 Var   temp:Link;
       a,b :Boolean;
       i :Integer;
 Begin
   If Cost>c then Exit;      （剪枝）
   If From=1 then     （递归终结条件）
   Begin
     If Cost<c then c:=Cost;
     Exit;
   End;
   temp:=Solution[From];
   While temp<>NIL do      （搜索主体）
   Begin
     a:=nUsed[temp^.Split];
     If not a then inc(cost);
     nUsed[temp^.Split]:=True;
     b:=nUsed[From - temp^.Split];
     If not b then inc(cost);
     nUsed[From-temp^.Split]:=True;
     i:=From-1;
     While (i>1) and (not nUsed[i]) do dec(i);
     GetCost(i,Cost);
     If not a then dec(cost);
     If not b then dec(cost);
     nUsed[From-temp^.Split]:=b;
     nUsed[temp^.Split]:=a;
```

```
        temp:=temp^.next;
    End;
  End;
Begin
  For i:=2 to Max do（动态规划计算所有解）
  Begin
    count:=0;
    min:=32767;
    For j:=1 to i div 2 do  （将 I 分解为 I-J 和 J）
    Begin
      c:=32767;
      FillChar(nUsed,Sizeof(nUsed),0);
      nUsed[j]:=True;nUsed[i-j]:=True;
      If j=i-j then GetCost(i-j,1)
      Else GetCost(i-j,2);
      If c<min then
      Begin
        count:=1;
        min:=c;
        plan[count]:=j;
      End
      Else if c=min then
           Begin
             inc(count);
             plan[count]:=j;
           End;
    End;
    new(solution[i]);  （构造解答链表）
    solution[i]^.split:=plan[1];
    solution[i]^.next:=NIL;
    Cost[i]:=min;
    tail:=solution[i];
    For j:=2 to count do
    Begin
      new(temp);
      temp^.split:=plan[j];
      temp^.next:=NIL;
      tail^.next:=temp;
      tail:=temp;
    End;
  End;
End;
```