[搜索]
[动态规划]

# 搜索----------------------------------------RETRUN

## BFS

BFS 的特点是当搜索到正解时，必然保证是与起点距离最近，根据此特点，在搜索一些地图上的最短路等方面有比较突出的效果。

实现：

建立一个队列，将初始状态加入队列；

从初始状态开始扩展，并将扩展得到的点加入队列；

从下一个开始继续扩展，直到找到目标；

地图扩展时经常需要用到转向变量

例：求地图中起点到终点的最短路径（含障碍物）

```
const dx:array[1..4]of -1..1=(-1,0,1,0);
      dy:array[1..4]of -1..1=(0,1,0,-1); //转向常量

var a,b:array[1..1000,1..1000]of integer;
    c:array[1..1000000,1..2]of integer;
    n,i,j,ax,ay,bx,by,t,w,x,y:longint;
    r:ansistring;

begin readln(n);
      for i:=1 to n do
      begin readln(r);
            for j:=1 to n do begin a[i,j]:=ord(r[j])-ord('0');
                                   b[i,j]:=0; //路径长度
                             end;
      end;
      readln(ax,ay,bx,by); //起点及终点坐标
      t:=1;w:=1;
      c[1,1]:=ax;c[1,2]:=ay; //广搜队列
      repeat
      for i:=1 to 4 do
      begin x:=c[t,1]+dx[i];
            y:=c[t,2]+dy[i]; //下一个扩展点
            if (x>0) and (x<=n) and (y>0) and (y<=n) and (a[x,y]=0) and (b[x,y]=0) then
            begin w:=w+1;
```

```
                    c[w,1]:=x;
                    c[w,2]:=y;
                    b[x,y]:=b[c[t,1],c[t,2]]+1; //满足条件则加入队列
                    if (x=bx) and (y=by) then break; //到达终点则退出
                end;
            end;
            t:=t+1;
            until t>w;
            writeln(b[bx,by]);
end.
```

当然，广搜不仅仅是能应用于地图搜索，广义上来说，队列里存放的是状态，地图坐标只是状态的一种

又如广搜还可以这么用：

POJ 1753

棋盘翻转

详见原题

```
const
change:array[1..16]of longint=(51200,58368,29184,12544,35968,20032,10016,4880,2248,
1252,626,305,140,78,39,19);
{用二进制数表示棋盘}

var flag:Array[0..65535] of boolean;
    q1,q2:array[1..65536]of longint;
    now,i,head,tail:longint;
    ch:char;

begin inc(tail);
      for i:=15 downto 0 do
      begin read(ch);
            if ch='b' then inc(q1[tail],1 shl i);
            if i mod 4=0 then readln
      end;{读入棋盘}
      flag[q1[tail]]:=true;
      if (q1[tail]=0) or (q1[tail]=65535) then head:=tail+1;{本来就是满足的则退出}
      while head<tail do
      begin inc(head);
            for i:=1 to 16 do
            begin now:=q1[head] xor change[i];{修改棋盘，xor 为二进制取反操作}
                  if not flag[now] then
                  begin inc(tail);
                        q1[tail]:=now;
                        q2[tail]:=q2[head]+1;
```

```
                        flag[now]:=true;{加入队列，广搜操作}
                        if (now=0) or (now=65535) then{满足则退}
                        begin head:=tail+1;
                              break;
                        end;
                  end;
            end;
      end;
      if head=tail+1 then writeln(q2[tail])
      else writeln('Impossible')
end.
```

这里广搜的队列中存放的就是以一列二进制数表示的棋盘

双向广搜，即从起点和终点共同向外搜索，直到两点重合则找到目标路线。实现时正反交替搜索或者优先搜索扩展结点较少的一边。

# DFS

DFS 自然与 BFS 是不同的，DFS 的搜索方式是一根筋走到底，然后逐层退出，逐层深入。DFS 应用时须注意剪枝问题，否则极易超时。

# A*算法

利用估价函数，改变搜索次序，以达到更快出解的目的。
一下参考 8 数码问题的程序详细解释。

```
例：8 数码难题 ： 2 8 3    1 2 3
                 1 6 4 -> 8   4（用最少的步数）
                 7   5    7 6 5
```

用 A*算法程序如下:

```
type a33=array[1..3,1..3] of 0..8;
    a4=array[1..4] of -1..1;
    node=record//数组链表
        ch:a33;//当前状态
        si,sj:1..3;//0 所在的位置
        f,pnt,dep,next:byte;//估价值、上一个的位置、搜索深度、下一个的位置
        end;

const goal:a33=((1,2,3),(8,0,4),(7,6,5));//目标状态
      start:a33=((2,8,3),(1,6,4),(7,0,5));//初始状态
      di:a4=(0,-1,0,1);
```

```
      dj:a4=(-1,0,1,0);//转向常量

var data:array[0..100] of node;
    temp:node;
    r,k,ni,nj,head,tail,depth:integer;

function check(k:integer):boolean;
begin ni:=temp.si+di[k];
      nj:=temp.sj+dj[k];//0 可能移的位置
      if (ni in [1..3]) and (nj in [1..3])//0 所移的位置符合
      then check:=true
      else check:=false;
end;

function dupe:boolean;//检查重复
var i,j,k:integer;
    buf:boolean;
begin buf:=false;
      i:=0;

      repeat
       inc(i);
      buf:=true;
       for j:=1 to 3 do
        for k:=1 to 3 do
         if data[i].ch[j,k]<>data[tail].ch[j,k]
           then buf:=false;
      until buf or (i>=tail-1);

      dupe:=buf;
 end;

 function goals:boolean;//验证是否达到最终状态
  var i,j:byte;
  begin goals:=true;
        for i:=1 to 3 do
         for j:=1 to 3 do
          if data[tail].ch[i,j]<>goal[i,j]
            then goals:=false;
  end;

 procedure print;//输出
 var buf:array[1..100] of integer;
     i,j,k,n:integer;
```

```
 begin n:=1;
         i:=tail;
         buf[1]:=i;

         repeat
         inc(n);
         buf[n]:=data[i].pnt;
         i:=data[i].pnt;
         until i=0;//得到状态序列

         writeln('steps=',depth+1);
         for i:=1 to 3 do
          begin for k:=n-1 downto 1 do
                   begin for j:=1 to 3 do
                             write(data[buf[k]].ch[i,j]);
                            if (i=2) and (k<>1)
                             then write('->')
                             else write('   ');
                   end;
                   writeln;
          end;//循环输出
          halt;
  end;

function calc(a:a33):byte;//估价函数
var i,j,temp:byte;
begin temp:=0;
        for i:=1 to 3 do
        for j:=1 to 3 do
         if (a[i,j]<>goal[i,j]) and (a[i,j]>0)
         then inc(temp);//记录与目标的差距
       calc:=temp+depth+1;//估价
end;

procedure sort(num:integer);//整理过程
var x,y:word;
begin y:=head;

        repeat
        x:=y;
       y:=data[x].next;
        until (y=0) or (data[y].f>data[num].f);//找到 head 到 num 之间比 num 大的

       data[x].next:=num;
```

```
    data[num].next:=y;//将 y 置于 num 之后
end;

begin head:=0;
      tail:=1;
      data[0].next:=1;
      with data[1] do
      begin ch:=start;
            si:=3;
            sj:=2;//0 所在的位置
            pnt:=0;
            dep:=0;
            next:=0; //初始化
            f:=calc(ch);//取得估价值
      end;

      repeat
      head:=data[head].next;
      temp:=data[head];//当前
      depth:=temp.dep;//深度
      for r:=1 to 4 do
      if check(r) then
      begin inc(tail);
            data[tail]:=temp;//将新状态加入队列
            with data[tail] do
            begin ch[si,sj]:=ch[ni,nj];
                  ch[ni,nj]:=0;
                  si:=ni;
                  sj:=nj;//完成 0 的移位
                  pnt:=head;//上一个的位置
                  dep:=depth+1;//深度
                  f:=calc(ch);//得到估价值
            end;
            if dupe//检查重复
            then dec(tail)
            else if goals//达到最终状态
                 then print//输出
                 else sort(tail);//整理按估价函数修改链表序列
      end;
      until data[head].next=0;

      writeln('no solution');
end.
```

小结下：

A*算法对于 8 数码问题的解法：

基本框架是广搜，以数组模拟链表实现

每次广搜过程中依据估价函数生成估价值

检查状态是否与之前重复，若是则从广搜序列中删除

否则判断是否达到目标，若是则依靠链表循环输出

否则依据估价值调整链表，调整过程由于是从 head 开始，保证对比的状态都是同一层的，若估价比当前生成的 tail 大，则将之排到 tail 后

# 动态规划-------------------------------------RETRUN

## 背包问题

背包问题最基本的就是 01 背包和完全背包，至于其他都是在此基础上发展开来的。这里参考背包九讲整理完成。

01 背包：

给出一个容量为 W 的背包，和 N 件物品，每件物品都有其重量 w[i]，及价值 v[i]，求背包中能达到的最大价值。

解决：作为最基础的背包，存在 f[I,j]=max{f[i-1,j],f[i-1,j-w[i]]+v[i]}，其中 f[I,j]表示前 i 件物品装入一个 j 大的背包能达到的最大价值。这个方程可以说是所有背包问题的最基础方程。然而二维的时空复杂度对于 01 背包来说还是有些大了。于是缩减掉 i，将其变为一维方程，即为 f[i]=max{f[i],f[i-w[i]]+v[i]}，对其的解读为，对于每一件物品可以选择装或不装。

伪代码：for i=1 to N

For j=W to w[i]

F[j]=max{f[j],f[j-w[i]]+v[i]}

完全背包：

给出一个容量为 W 的背包，和 N 种物品，每种物品都有其重量 w[i]，及价值 v[i]，求背包中能达到的最大价值。

解决：同上，先写出一个最为简单易懂的二维方程来。F[I,j]=max{f[i-1,W-k*w[i]+k*v[i]}，当然此处的 k 是有范围限制的。当然，这也可以优化为一维，

如下为一维的伪代码：for i=1 to N

For j=w[i] to W

F[j]=max{f[j],f[j-w[i]]+v[i]}

不难发现，它的一维伪代码跟 01 背包只有 j 循环顺序上不同，因为在 01 背包中，f[I,j]是由状态 f[i-1,W-w[i]]递推而来，转化为一维以后必须保证在考虑选取第 i 件物品时子结果必不包括第 i 件物品。而完全背包则不同，每件物品可以选取无限件，于是顺推子结果中可以并且必须包含第 i 件物品已经选取过的情况。

多重背包：

给出一个容量为 W 的背包，和 N 种物品，每种物品都有其重量 w[i]、价值 v[i]及数量 n[i]，求背包中能达到的最大价值。

解决：多重背包类似于完全背包，然而在数量上又有限制。F[I,j]=max{f[i-1,j-k*w[i]+k*v[i]}二维方程与完全背包相同，只不过 k 值需要作更多的限制。但我们知道二维的时空复杂度并不利于解决一些有设计的题。于是可以转化为 01 背包来做。

我们知道用 1,2,4,8……可以得到所有数（如 3=1+2,5=1+4,6=2+4,7=1+2+4……）由于在完全背包中每种物品的数量都是确定的，于是我们可以采用这样二进制分拆的方式，将原有的每种物品分拆成多件物品，使每一堆物品的数量为 1,2,4,8……2^(k-1),n[i]-2^k，再采用 01 背包的方法解决。如某种物品有 13 件则分拆成 1,2,4,6 件，每件物品的价值与重量分别乘上相应的系数。

例：

```
var n,m,i,j,k,max,t,v1,w1,num1,r:longint;
    v,w:array[1..10000]of longint;
    f:array[0..40000]of longint;

begin readln(n,m);
      t:=0;
      for i:=1 to n do
      begin readln(v1,w1,num1);
            r:=1;
            while r<=num1 do
            begin inc(t);
                  v[t]:=v1*r;
                  w[t]:=w1*r;
                  num1:=num1-r;
                  r:=r*2;
            end;
            if num1>0
            then begin inc(t);
                        v[t]:=num1*v1;
                        w[t]:=num1*w1;
                 end;
      end; //按二进制分组
      fillchar(f,sizeof(f),0);
      for i:=1 to t do//01 背包过程
      for j:=m downto w[i] do
      begin if f[j-w[i]]+v[i]>f[j]
            then f[j]:=f[j-w[i]]+v[i];
            if max<f[j]
            then max:=f[j];
      end;
      writeln(max);
end.
```

混合背包：

给出一个容量为 W 的背包，和 N 种物品，其中有的只能取一次，有的能取无限次，有的有

数量规定，求能达到的最大价值。

解决：考虑到 01 背包与完全背包给出的伪代码只有一处不同，故如果只有两类物品：一类能取一次，一类能取无限次。则只需在对应每一个物品应用方程时，选择不同的循环方向即可。

伪代码：
```
for i=1 to N
    If 第 i 件物品属于 01 背包
        for j=W to w[i]
            f[j]=max{f[j],f[j-w[i]]+v[i]}
    if 第 i 件物品属于完全背包
        for j=w[i] to W
            f[j]=max{f[j],f[j-w[i]]+v[i]}
```

若是加上多重背包，同样如上可以先把多重背包分拆成 01 背包，再继续完成。

伪代码：
```
for i=1 to N
    If 第 i 件物品属于 01 背包（包括分拆完以后的多重背包）
        for j=W to w[i]
            f[j]=max{f[j],f[j-w[i]]+v[i]}
    if 第 i 件物品属于完全背包
        for j=w[i] to W
            f[j]=max{f[j],f[j-w[i]]+v[i]}
```

二维背包：

给出一个能装 W 重 Q 大的背包，给出 N 件物品，每件物品都有其价值 v[i]，重量 w[i]，体积 q[i]，求能达到的最大价值。

解决：其实就是把费用扩展为二维的背包，状态只需在原有基础上再加上一维即可。三维方程：f[I,j,k]=max{f[i-1,j,k],f[i-1,j-w[i],k-q[i]]+v[i]}，同样可以把 i 那一维略去，优化为二维。

伪代码：
```
for i=1 to N
    For j=W to w[i]
    For k=Q to q[i]
    F[j,k]=max{f[j,k],f[j-w[i],k-q[i]]+v[i]
```

以上为二维 01 背包的伪代码，至于二维完全背包及多重背包有了上面的基础，不难完成。

有些一维背包问题中还有这样一种条件：最多可以从这 N 件物品中取 M 件，其余条件不变。其实这也可以看做二维背包，设每一件物品的另一种费用都为 1，费用容量为 M，依照上述，不难完成此类问题的解答。

例：
```
var t1,v1,g1:array[0..400]of longint;
    f:array[0..400,0..400]of longint;
    n,i,j,k,v,g:longint;

begin readln(v,g);
      readln(n);
      fillchar(f,sizeof(f),0);
      for i:=1 to n do readln(t1[i],v1[i],g1[i]);
      for i:=1 to n do
```

```
for j:=v downto v1[i] do
for k:=g downto g1[i] do
if f[j,k]<f[j-v1[i],k-g1[i]]+t1[i] then
f[j,k]:=f[j-v1[i],k-g1[i]]+t1[i];
writeln(f[v,g]);
end.
```

分组背包：
背包中给出的物品分成几组，每组中的物品互相冲突，最多选一件。求能达到的最大价值。

解决：f[k,j]=max{f[k-1,j],f[k-1,j-w[i]]+v[i]}（物品 i 属于 k 组）

伪代码：
```
for 所有的组 k
    For j=W to w[i]
    For 所有的 i 属于组 k
    F[j]=max{f[j],f[j-w[i]]+v[i]}
```

# 树形动规

这类问题看似复杂，其实理解以后实现起来也颇为简单。
主要应对一些具有树形关系的动规问题。
不多说，看题：

例：选课（来自树规七题）

学校实行学分制。每门的必修课都有固定的学分，同时还必须获得相应的选修课程学分。学校开设了 N（N<300）门的选修课程，每个学生可选课程的数量 M 是给定的。学生选修了这 M 门课并考核通过就能获得相应的学分。

在选修课程中，有些课程可以直接选修，有些课程需要一定的基础知识，必须在选了其它的一些课程的基础上才能选修。例如《FrontPage》必须在选修了《Windows 操作基础》之后才能选修。我们称《Windows 操作基础》是《FrontPage》的先修课。每门课的直接先修课最多只有一门。两门课也可能存在相同的先修课。每门课都有一个课号，依次为 1，2，3，…。

若 1 是 2 的先修课，2 是 3、4 的先修课。则如果要选 3，那么 1 和 2 都一定是被选修过。

你的任务是为自己确定一个选课方案，使得你能得到的学分最多，并且必须满足先修课优先的原则。假定课程之间不存在时间上的冲突。

```
var tt:array[1..301]of record
                front,num,data,left,right:integer;
                end;
    x,v:array[1..301]of integer;
    t,f:array[0..301,0..300]of integer;
    n,m,i,j,k,max:integer;

procedure maketree;//建树过程
var head,tail,now:integer;
```

```
begin tt[1].num:=n;
      head:=0;
      tail:=1;
      repeat
      inc(head);
      for i:=1 to t[tt[head].num,0] do
      begin if tt[head].left=0
            then begin inc(tail);
                       tt[tail].front:=head;
                       tt[tail].num:=t[tt[head].num,1];
                       tt[tail].data:=v[tt[tail].num];
                       tt[head].left:=tail;
                 end
            else begin now:=tt[head].left;
                      while tt[now].right<>0 do
                      now:=tt[now].right;
                      inc(tail);
                      tt[tail].front:=now;
                      tt[tail].num:=t[tt[head].num,i];
                      tt[tail].data:=v[tt[tail].num];
                      tt[now].right:=tail;
                 end;
            if tail=n then break;
      end;
      until tail=n;
end;

begin readln(n,m);
      for i:=1 to n do
      begin readln(x[i],v[i]);
            if x[i]<>0
            then begin inc(t[x[i],0]);
                      t[x[i],t[x[i],0]]:=i;
                 end;
      end;//读入数据并以邻接表储存
      for i:=1 to n do
      if x[i]=0
      then begin x[i]:=n+1;
                 inc(t[n+1,0]);
                 t[n+1,t[n+1,0]]:=i;
           end;
      inc(n);
      v[n]:=0;//若存在多门课是不存在先修课的，则添加一个结点作为总根
      fillchar(tt,sizeof(tt),0);
```

```
    maketree;//建树
    fillchar(f,sizeof(f),0);
    for i:=n downto 1 do
    for j:=1 to m do
    begin if f[i,j]<f[tt[i].right,j]
        then f[i,j]:=f[tt[i].right,j];
        for k:=0 to j-1 do
        if j-k-1>=0
        then if f[i,j]<f[tt[i].left,k]+f[tt[i].right,j-k-1]+tt[i].data
            then f[i,j]:=f[tt[i].left,k]+f[tt[i].right,j-k-1]+tt[i].data;
    end;
    writeln(f[2,m]);
end.
```

可以看到整体框架还是较为简单的：
建立树
进行树形动规
只需这样两个步骤。建树时依照左儿子右兄弟的原则进行，为防止因存在多个无先修课的课程存在而生成多棵树，添加一个顶点，将所有无先修课的课程连接在该节点下。
状态转移方程式其实还是不难完成的。