第5章　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结点的过程。

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。

5.1　基本搜索算法

5.1.1　递归与迭代

递归程序设计是编程语言设计中的一种重要的设计方法，它使许多问题简单化，易于求解。递归的特点：函数或过程直接的或间接的调用它自己本身。

所谓迭代，就是在程序中用同一个变量存放每一次推算出的值，每一次循环都执行同一语句，给同一变量赋以新值，即用一个新值代替旧值。

5.1.2　深度优先搜索与广度优先搜索

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结点选取上。由于其保留了所有的前驱结点，所以在产生后继结点时可以去掉一部分重复的结点，从而提高了搜索效率。

这两种算法每次都扩展一个结点的所有子结点，而不同的是，深度搜索下一次扩展的是本次扩展出来的子结点中的一个，广度搜索扩展的则是本次扩展的结点的兄弟结点。在具体实现上为了提高效率，所以采用了不同的数据结构。

1. 深度优先搜索

深度优先搜索(Depth First Search，DFS)属于图算法的一种，其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个结点只能访问一次。

2. 广度优先搜索

广度优先算法(Breadth First Search，BFS)又称宽度优先算法，是最简便的图搜索算法之一。其过程是说在搜索过程中，按层次进行搜索，本层的结点没有处理完毕前，不能对下一层结点进行处理，即深度越小的结点越先进行处理。

5.1.3　回溯

回溯算法是一种系统的搜索问题的解的方法。它的基本思想是：从一条路前行，能进则进，不能进则退回来，换一条路再试。回溯法是一种通用的解题方法。

应用回溯算法的时候，首先要明确定义问题的解空间。解空间中至少应该包含问题的

一个解。确定了解空间后，回溯法从开始结点出发，以深度优先的方法搜索整个解空间。

对于回溯算法一般可以采用递归方式来实现。

递归函数很容易写：

```
void DFS(int deep)                                    //deep 表示当前递归的深度
{
    if 已经超过递归的深度
return;
    for 遍历本结点的所有可扩展结点
    {
        DFS(deep+1);
    }
}
```

当然回溯算法还可以使用迭代等其他方式来描绘。

5.2 搜索算法的一些优化

5.2.1 剪枝函数

对于回溯算法，需要搜索整棵解空间树，剪枝顾名思义，就是通过某种判断，避免一些不必要的遍历过程，剪去解空间树中的一些不必要的枝条，从而缩小整个搜索的规模。通常有两种策略来提高回溯法的搜索效率，一是用约束函数在扩展结点处剪去不满足约束条件的子树，二是用限界函数剪去不能得到最优解的子树。

对于限界函数，一种比较容易的形式的是添加一个全局变量记录当前最优解，而到达结点的时候计算该结点的预期值并与当前最优解比较。如果不好，则回溯。

5.2.2 双向广度搜索

所谓双向广度搜索，顾名思义指的是搜索沿正向（从初始结点向目标结点方向搜索）和逆向（从目标结点向初始结点方向搜索）两个方向同时进行，当两个方向上的搜索生成同一子结点时完成搜索过程。

运用双向广度搜索理想上可以减少二分之一的搜索量，从而提高搜索效率。

双向广度搜索一般有两种方法：一种是两个方向交替扩展，另一种是选择结点个数比较少的方向先扩展。显然第一种方法比第二种方法容易实现，但是由于第二种方法克服了双向搜索中结点生成速度不平衡的状态，效率将比第一种方法高。

5.3 实例分析

例 5-1 宝石游戏

1. 题目描述

宝石游戏非常有趣，它在 13×6 的格子里进行。游戏给出红色、蓝色、黄色、橘黄色、绿

色和紫色的宝石。当任何三个以上宝石具有相同颜色并且在一条直线(横竖斜)时,这些宝石可以消去。

游戏截图如图 5-1 所示。

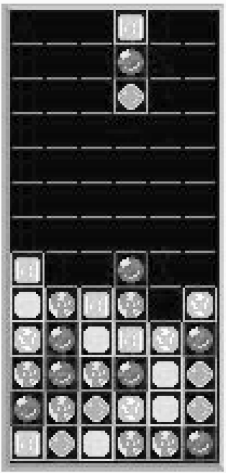

图 5-1

现在给定当前游戏状态和一组新的石头,请计算当所有石头落下时游戏的状态。

输入:

第一行 n 表示有 n 组测试数据。

下面每一个测试数据包含一个 13×6 的字符表,其中 B 表示蓝色,R 表示红色,O 表示橘黄色,Y 表示黄色,G 表示绿色,P 表示紫色,W 表示此处没有宝石。

接下来三行,每行包含一个字符,表示新来的宝石。

最后一个整数 $m(1\leqslant m\leqslant6)$,表示新来的宝石下落位置。

输出:

每一个测试样例,输出当所有宝石落下后游戏状态。

样例输入:

```
1
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
BBWWWW
BBWWWW
OOWWWW
B
B
Y
3
```

样例输出:

```
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
```

```
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
OOYWWW
```

2. 解题思路

看这个题目可以联想到俄罗斯方块，但是跟俄罗斯方块有些不一样，每个方块都有自己的颜色，需要同色，而且（水平、垂直或对角线方向）连着有 3 个或 3 个以上的方块，才能消去。现在的问题是给出一个状态，然后再给 3 个方块，问这 3 个方块掉下来以后，最终的状态。需要注意的是：某个方块满足两个或两个以上方向均可消去时，要将在这些方向上满足条件的方块都消去，然后下落填补空缺。

可以把整个问题分解成几个部分：第一个是判断当前状态有哪些宝石是可以消去的，如果有，那么将那些宝石消去，并在消去的地方用 W 代替，如果没有了，那就是游戏的最终状态，输出结果；第二个是将所有可以下落的宝石下落到无法下落为止，得到一个状态。那么这个问题就是不断地进行这两步操作，直到得到最终结果。所以本题关键是实现这两种操作的两个函数。

对于任意一块宝石，都可以向 8 个方向（实际上只需要 4 个方向）进行扩展判断是否存在可以消去的宝石。

3. 常见错误

在没有完全将当前可以消去的宝石完全消去前，就直接将宝石下落，这样就将使得有些可以消去的宝石没有消去，而不可以消去的宝石消去，导致出错。

4. 参考程序

```
//得到最开始的状态后,按题目指定的位置加入一些宝石
//寻找 3 个或者 3 个以上相同的宝石,将这些相同的宝石删去后得到新状态
//更新后继续寻找,如此循环直到没有 3 个或 3 个以上相等的为止
#include<iostream>
#include<cstring>
#include<cstdlib>
#define H 13                                          //容器的高度
#define W 6                                           //容器的宽度
#define JH 3                                          //新增宝石的高度
#define JW 1                                          //新增宝石的宽度
using namespace std;

char table[H+2][W+2];                                 //容器中宝石的状态
bool isW[H][W];                                       //判断当前格是否为空
char newj[JH][JW];                                    //新宝石的状态
bool flag;
const int step[4][2]={{1,0},{0,1},{1,1},{-1,1}};     //定义 4 个方向的扩展
```

```
void downOp()                                    //刷新一遍得到下一个状态(宝石下落后)
{
    int i,j,x;
    for(j=0;j<W;j++)
    {
        x=H-1;
        for(i=H-1;i>=0;i--)                      //珠宝下落,依次将为空的位置填满
        {
            if(isW[i][j])
                continue;
            table[x--][j]=table[i][j];
        }
        for(i=x;i>=0;i--)                        //最后为空的赋值为'W'
            table[i][j]='W';
    }
}
void search()                                    //消去宝石
{
    flag=true;                                   //默认本次操作没有消去宝石
    int i,j,k,t;
    for(i=0;i<H;i++)
        for(j=0;j<W;j++)
            isW[i][j]=false;

    for(i=0;i<H;i++)                             //开始判断有没有能消去的宝石,有的话消去
    {
        for(j=0;j<W;j++)
        {
            if(table[i][j]!='W')                 //约束判断,如果是 W 则无须操作
            {
                for(k=0;k<4;k++)                 //4 个方向扩展
                {
                    if(i+2*step[k][0]>=0)
                    {                            //约束判断,将不符合条件、越界的舍去
                        if(table[i+step[k][0]][j+step[k][1]]==table[i][j]
                        && table[i+2*step[k][0]][j+2*step[k][1]]==table[i][j])
                        {
                            t=0;
                            while(i+t*step[k][0]>=0
                            && table[i+t*step[k][0]][j+t*step[k][1]]==table[i][j])
                            {                    //找到该方向上的所有能消的宝石
                                isW[i+t*step[k][0]][j+t*step[k][1]]=true;
                                t++;
                            }
                            flag=false;
```

```
                        }
                    }
                }
            }
        }
    }
    if(!flag)                                   //如果有消去宝石则刷新状态
        downOp();
}
int main()
{
    int i,j,T,l,h,t;
    cin>>T;
    while(T--)
    {
        for(i=0;i<H;i++)
        {
            cin>>table[i];
        }
        for(i=0;i<JH;i++)
        {
            cin>>newj[i];
        }
        cin>>l;
        l=l-1;
        for(i=H-1;i>=0;i--)
        {
            if(table[i][l]=='W')
            {
                h=i;
                break;
            }
        }
        for(i=h-JH+1;i<=h;i++)                  //将新增的宝石放入容器
        {
            table[i][l]=newj[i-(h-JH+1)][0];
        }
        flag=false;
        while(1)                                //开始进入循环,判断是否存在能消去的宝石
        {
            search();
if(flag)break;
        }
        for(i=0;i<H;i++)
        {
```

```
            for(j=0;j<W;j++)
            {
                cout<<table[i][j];
            }
            cout<<endl;
        }
    }
    return 0;
}
```

5. 优化

对于本题，如果扩大容器的大小，而每次能消去的宝石数量又是非常的少，那么遍历每个结点将要进行较多次的重复操作，对于一个点上的宝石可能几次操作都没有变化过，但是却每次都要去再遍历一次。对于这种情况，如果每次的扩展可以从上一次的状态中有变化的点开始扩展，那样的话很多没有变化的点就不需要再重复遍历了。

例 5-2　骑士移动

1. 题目描述

许多人都知道“飞马棋”(骑士的移动方式和中国象棋中的子是一样的)，现在你也要模仿这个游戏了。给定正方形棋盘边长 M 和一些骑士不能走的格子，以及骑士起始位置和终点位置，找出从起始位置到达终点位置的最短的路径。如果无法到达，输出“No solution”，我们给每个格子编号，从 1 到 $M\times M$。例如一个 4×4 的棋盘，则棋盘格子号码如下：

```
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
```

输入：

输入包括多组数据，以整数 T 开始，表示共有几组。每组测试样例包括三个部分：

第一部分仅有一个整数 M($M\leqslant100$，棋盘边长)。

第二部分包含骑士初始位置(X_s，Y_s)和终点位置(X_e，Y_e)($1\leqslant X_s,Y_s,X_e,Y_e\leqslant M$)。

第三部分由整数 N 开始，表示共有 N 个骑士不能到达的点，接下来的 $N\times2$ 个数每两个表示一个位置坐标。

输出：

每组测试样例，输出从起始位置到达终点位置的最短的路径，输出的不是坐标而是代表格子的数字，如果找到多组路径，输出最小的数字序列，例如给出 4×4 的棋盘，起始点为(3，2)，终点为(2，3)，你可以找到两条最短路径(10，11，7)和(10，6，7)，你不用输出(10，11，7)，只需输出(10，6，7)。

样例输入：

```
2
4
```

```
3 4
3 2
4
4 4
2 1
4 1
3 3
4
1 1
4 4
3
3 3
3 4
4 3
```

样例输出：

```
12 3 10
No solution
```

2. 解题思路

这个题的意思就是要你模仿象棋中马(可以理解 Knight 为象棋中的马)走路(如果你不知道象棋中马的走法的话,可以问问你的同学)。不过这个题是需要你在方格里走,即马只能停留在格子中。题目是给你一个 $M\times M$ 的矩形,限制一些格子是不能走的(以坐标形式给出),然后告诉你马的起点和终点,让你在余下来的格子中找出一条起点到终点的最短路径。

这是个典型的广度优先搜索题,按广度优先搜索的写法就可以找到需要的最优解的步数,同时在扩展每个结点的时候记录路径就可以得到获得最优解时候的行进路线。

由于广度优先搜索可以直接得到最优解的步数,所以我们可以将最优步数下的所有解都列出来,比较所有解后,再按题目要求输出,虽然这样做固然可以得到问题的答案,但是当问题规模扩大的时候,最优步数下的所有解的数量可能非常多,那么比较所有解的操作将变得非常频繁。因此,如果能找到一种方法使找到的第一组解就是我们所想要的解,那么问题将变得简单得多。

在这个题里,扩展的顺序决定了先得到哪组解,只需要在每次扩展的时候先引入坐标值较小的所得到的第一组解就是所需要的第一组解。

3. 参考程序

```
#include<iostream>
using namespace std;

int visit[301][301];
int board[301][301];
int num[301][301];
int go[][2]={{-2,-1},{-2,1},{-1,-2},{-1,2},{1,-2},{1,2},{2,-1},{2,1}};
                                                    //定义 8 个方向
```

```
typedef struct MyQueue{
    int step;
    int x,y;
    int len;
    int path[1000];                          //用 path 来记录路径
    MyQueue * next;
}Q;
Q * head, * tail, * p, * q, * temp;

int main()
{
    int i,j,step;
    int x,y,xnow,ynow;
    int T;
    int xs,ys,xe,ye;
    int flag;
    int path[1000];
    int m,n;

    cin>>T;
    while(T--)
    {
        //初始化基本信息
        cin>>n;
        memset(board, 0, sizeof(board));
        memset(visit, 0, sizeof(visit));
        for(i=1; i<=n; i++){
            for(j=1; j<=n; j++){
                num[i][j]=(i-1) * n+j;
            }
        }
        cin>>xs>>ys;
        cin>>xe>>ye;
        cin>>m;
        for(i=0; i<m; i++){
            cin>>x>>y;
            board[x][y]=1;
        }

        head=new Q;
        tail=head;
        temp=new Q;
        temp->x=xs;
        temp->y=ys;
```

```
temp->step=0;
temp->path[0]=num[xs][ys];
temp->next=NULL;
tail->next=temp;
tail=temp;
visit[xs][ys]=1;

flag=0;
//开始广度优先搜索
while(head->next!=NULL)
{
    p=head->next;
    x=p->x;
    y=p->y;
    step=p->step;
    for(i=0; i<=step; i++)
    {
        path[i]=p->path[i];
    }
    head->next=p->next;
    if(head->next==NULL)
    {
        tail=head;
    }
    delete p;
    if(x==xe&&y==ye)
    {
        cout<<path[0];
        for(i=1; i<=step; i++)
        {
            cout<<''<<path[i];
        }
        cout<<endl;
        flag=1;
        while(head->next!=NULL)
        {
            temp=head->next;
            delete head;
            head=temp;
        }
        break;
    }
    //向 8 个不同的方向扩展,同时注意扩展的顺序
    for(int i=0;i<8;i++)
    {
```