# 1 需求

## 1.1 项目背景

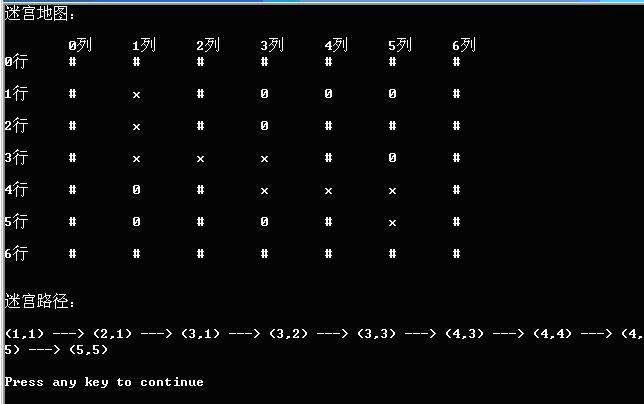
最短路径搜索算法在生活中应用范围较广。“路径”除了表示现实意义上的两点间距以外，还可以表示两个状态间带权的抽象通路，比如状态A到状态B所需的成本。最短路径搜索算法在现实领域的应用有帮助机器人决策最优移动路径，提高其功能执行效率，在抽象领域的应用有页面查找、决策博弈等。而本项目以迷宫游戏为背景，模拟搜索最短路径的过程，并打印最终结果。

为简化模型，本项目所创建的迷宫共有两个门，分别为入口和出口，位置设定为地图的左上角与右下角。玩家作为骑士骑马从入口进入迷宫，避开迷宫内设置的障碍，寻找通路以到达出口。程序执行初玩家在窗口输入所创建迷宫的行列数，程序根据行列数随机生成存在可达路径的迷宫地图，并打印出该迷宫的最短通路。

## 1.2 功能分析

迷宫问题的求解过程可采用回溯法即在一定的约束条件下试探地搜索前进，若前进中受阻，则及时回头纠正错误另择通路继续搜索的方法。从入口出发，按某一方向向前探索，若能走通，即某处可达，则到达新点，否则探索下一个方向；若所有的方向均没有通路，则沿原路返回前一点，换下一个方向再继续试探，直到所有可能的道路都探索到，或找到一条通路，或无路可走又返回入口点。在求解过程中，为了保证在达到某一个点后不能向前继续行走时，能正确返回前一个以便从下一个方向向前试探，则需要在试探过程中保存所能够达到的每个点的下标以及该点前进的方向，当找到出口时试探过程结束。

本项目采用标记数组模拟回溯试探过程，在搜索策略上采用广度优先搜索。当搜索过程完成后，数组中对应的元素数值置为标记值，从而存储该地图的最短可达路径。



# 2 设计

## 2.1 数据结构设计

### 2.1.1 逻辑结构设计

本项目的核心功能是创建地图并搜索起点终点间的最优解。由于元素相对独立，各个元素之间不存在明显的偏序关系或层次关系，也不存在“一对多”与“多对一”的映射，所以可以采用线性结构进行存储。

### 2.1.2 存储结构设计

线性表的存储通常通过顺序存储和链式存储方式实现。而在本项目中对线性表元素只需做读取和修改操作，相对而言顺序存储效率更高，故可建立二维数组模拟地图上的节点，该点的行列值即为其在二维数组中的位置，读取和修改该点数值的时间复杂度均为O(1)。本项目中共有两个二维数组，一个用于存放迷宫初始地图，另一个在初始地图的基础上存放搜索路径结果，程序通过对这两个二维数组操作实现迷宫的初始化与最短路径搜索。

为了节省空间，本项目采用动态二维数组的方式存储int型二维数组，二维数组的大小通过用户输入的迷宫长宽指定，通过指向int\*的指针访问。动态二维数组建立的源代码如下：

//分配初始地图空间

road = **new** **int**\*[width];

**for** (**int** i = 0; i < width; i++)

    road[i] = **new** **int**[length+1];

//分配结果地图空间

result = **new** **int**\* [width];

**for** (**int** i = 0; i < width; i++)

    result[i] = **new** **int**[length+1];

### 2.1.3 road数组

road数组用于存放初始地图，其根据用户输入的width和length参数动态分配存储空间。Road[i][j]则表示地图上第i行第j列的位置状态：

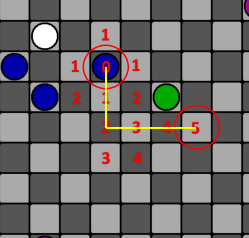
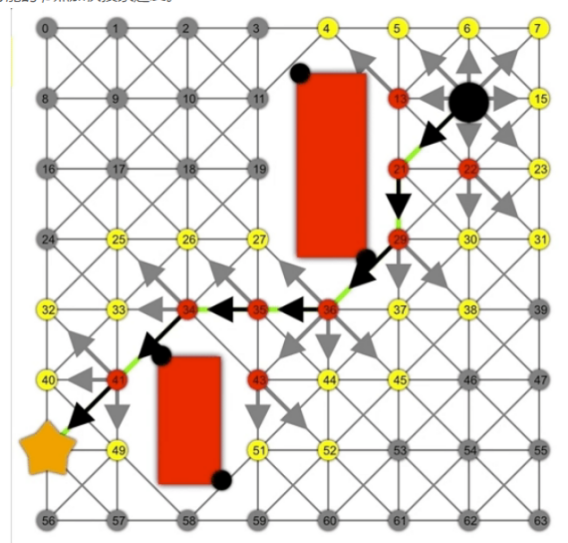
|  |  |  |
| --- | --- | --- |
| road[i][j] | sign[road[i][j]] | 实际含义 |
| 0 | 0 | 空地 |
| 1 | # | 障碍物 |

当road[i][j]值为1时表示该处存在障碍物无法通行，故路径搜索即为寻找一条仅通过0的通路。地图初始化时将road数组的每一个元素随机赋值0或1，若该地图不存在起点到终点的可达通路，则重新给road数组元素赋值。

### 2.1.4 result数组

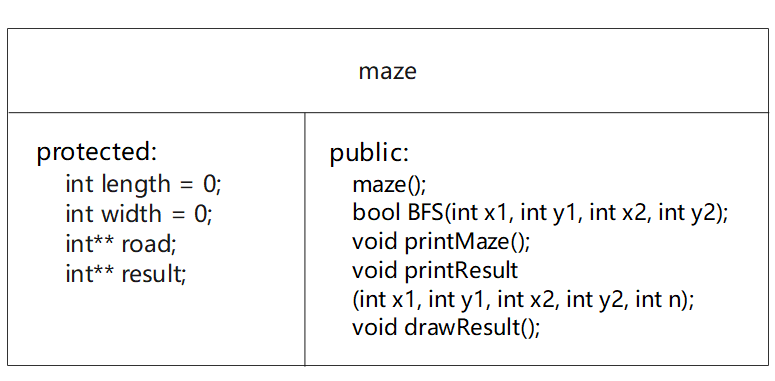
result数组在初始地图的基础上存放搜索路径结果，其根据用户输入的width和length参数动态分配存储空间。result[i][j]表示地图上第i行第j列的路径状态，当result[i][j] > 0时表示其距离终点的位置，故终点处数值为1。当result[i][j] = -1时表示该处为障碍物，需要绕开。最后在打印路径结果时将所有数值大于0的元素赋值为-2，从而在图形化界面打印出最终结果。

|  |  |  |
| --- | --- | --- |
| result[i][j] | result[road[i][j]] | 实际含义 |
| 0 | 0 | 空地 |
| -1 | # | 障碍物 |
| 1 | \* | 终点 |
| 2 | \* | 距终点为2的点 |
| 3 | \* | 距终点为3的点 |
| 4 | \* | 距终点为4的点 |
| ...... | \* | 距终点为...的点 |
| n | \* | 距终点为n的点 |

## 2.2 类设计

本项目共设计一个迷宫类maze，用于初始化迷宫、广度优先搜索迷宫最短路径、打印最终结果等。其protected属性个成员函数有迷宫长度length、迷宫宽度width、初始地图int\*\* road，结果地图int\*\* result。其public属性主要用于存放各项操作。



//迷宫类

**class** maze {

**protected**:

**int** length = 0;    //迷宫长度

**int** width = 0;     //迷宫宽度

**int**\*\* road;        //初始地图

**int**\*\* result;      //结果地图

**public**:

    maze();

**bool** BFS(**int** x1, **int** y1, **int** x2, **int** y2);    //广度优先搜索函数

**void** printMaze();    //打印迷宫地图

**void** printResult(**int** x1, **int** y1, **int** x2, **int** y2, **int** n);    //打印路径文字解

**void** drawResult();    //打印路径图形解

};

# 

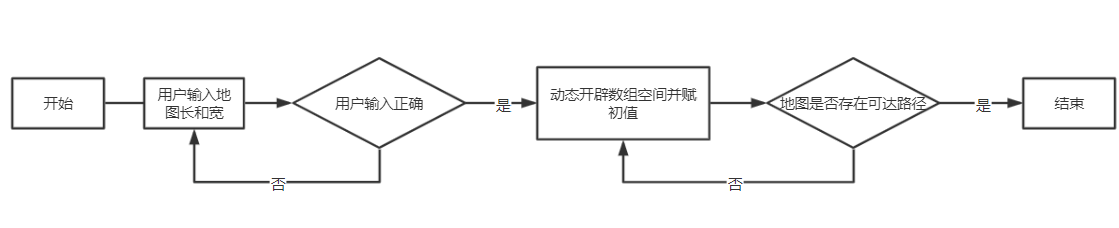
# 3 实现

## 3.1 地图初始化功能的实现

### 3.1.1 地图初始化功能实现思路

地图初始化在程序中体现为二维数组初始化，其先根据用户输入的width、length值动态分配数组空间，再通过random头文件中封装的rand函数将rand数组中的每一个元素随机赋值为0或1，对应result数组中每一个元素赋值为0或-1。若用户输入错误或输入的长宽值过大，则系统报错并提示用户重新输入。生成地图数组后调用BFS函数判断该地图是否存在起点到终点的可达路径，若该路径不存在则重新生成。

### 3.1.2 地图初始化功能实现流程图



### 3.1.3 地图初始化功能实现代码

//构造函数

maze::maze() {

    //输入迷宫长和宽

    cout << "请输入构建迷宫的长和宽：" << endl;

**while** (1) {

        cin >> length >> width;

**if** (cin.fail() || length > 20 || width > 20) {

**if** (length > 20 || width > 20)

                cout << "长和宽应该在0-20范围内，请重新输入:";

**else**

                cout << "输入错误，请重新输入:";

            cin.clear();

**char** t;

**while** ((t = cin.get()) != '\n');

        }

**else**

**break**;

    }

    //分配初始地图空间

    road = **new** **int**\*[width];

**for** (**int** i = 0; i < width; i++)

        road[i] = **new** **int**[length+1];

    //分配结果地图空间

    result = **new** **int**\* [width];

**for** (**int** i = 0; i < width; i++)

        result[i] = **new** **int**[length+1];

**while** (1) {

        //地图初始化

**for** (**int** i = 0; i < width; i++) {

**for** (**int** j = 0; j < length; j++) {

                road[i][j] = rand() % 2;

**if** (road[i][j] == 1)

                    result[i][j] = -1;

**else**

                    result[i][j] = 0;

            }

        }

        result[width - 1][length - 1] = 1;

        //若创建的地图不存在可达路径，则重置

**if** (BFS(width - 1, length - 1, 0, 0))

**break**;

    }

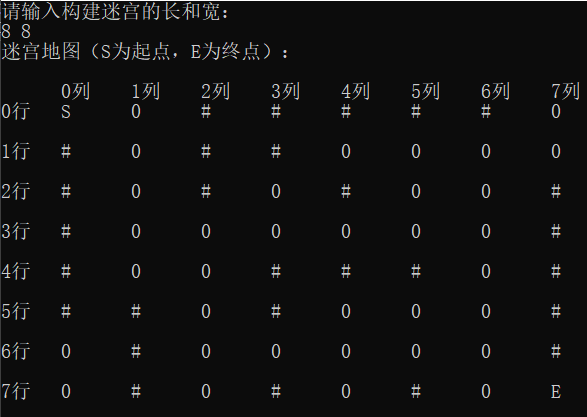
    printMaze();

    printResult(0, 0, width - 1, length - 1, result[0][0]);

    drawResult();

}

### 3.1.3 地图初始化功能实现样例



## 3.2 最短路径搜索功能的实现

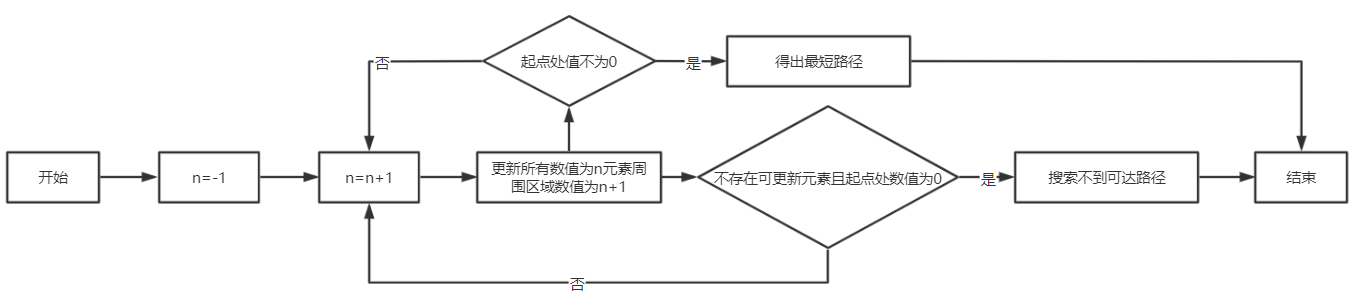
### 3.2.1 最短路径搜索功能实现思路

最短路径搜索可用回溯法，将问题的候选解按某种顺序逐一枚举和检验。当发现当前的候选解不可能是解时，就放弃它并选择下一个候选解。如果当前候选解除了不满足问题的整体约束外，其余约束均已满足，则扩大候选解的规模继续试探。本项目可用广度优先搜索方式实现回溯，一种常见的实现形式是利用队列，每次弹出一个节点，并将该节点的相邻节点压入队列中，而由于该方法记录路径时仍需标记数组，所以本项目整体采用标记数组的方法实现。

初始状态下result数组空地处数值为0，障碍物处为-1，程序从终点出发，循环更新result数组。第一次循环将所有距离终点为1的区域标记为2，第二次循环将标记为2区域四个方位的所有区域标记为3......循环结束的标志有两个，一是不存在任何可以更新的数组元素，此时代表所有可达通路已被遍历完，如若此时起点位置数值仍为0，则表示此时无法到达起点；二是起点处数值不为0，此时说明已经寻找到最短路径，任务完成。

至于为何从终点开始向起点计数，本文会在3.3.1进行说明。

### 3.2.2 最短路径搜索功能实现流程图



### 3.2.3 最短路径搜索功能实现代码

//广度优先搜索函数

**bool** maze::BFS(**int** x1, **int** y1, **int** x2, **int** y2) {

**if** (road[x1][y1] != 0 || road[x2][y2] != 0)

**return** **false**;

**while** (1) {

**int** flag = 1;

**for** (**int** i = 0; i < width; i++) {

**for** (**int** j = 0; j < length; j++) {

**if** (result[i][j] > 0) {

**int** n = result[i][j];

**if** (j - 1 >= 0 && result[i][j - 1] == 0) {

                        result[i][j - 1] = n + 1;

                        flag = 0;

                    }

**if** (j + 1 < length && result[i][j + 1] == 0) {

                        result[i][j + 1] = n + 1;

                        flag = 0;

                    }

**if** (i - 1 >= 0 && result[i - 1][j] == 0) {

                        result[i - 1][j] = n + 1;

                        flag = 0;

                    }

**if** (i + 1 < width && result[i + 1][j] == 0) {

                        result[i + 1][j] = n + 1;

                        flag = 0;

                    }

                }

            }

        }

**if** (flag)

**break**;

    }

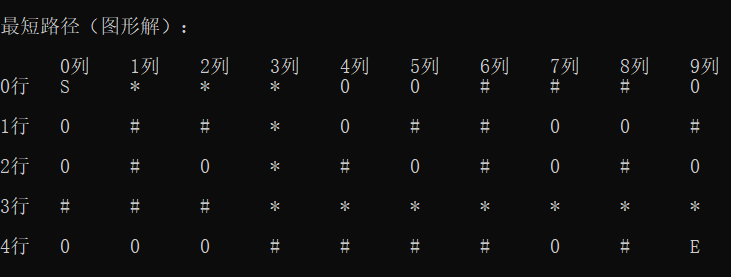
**if** (result[x2][y2] == 0)

**return** **false**;

**return** **true**;

}

### 3.2.4 最短路径搜索功能实现样例



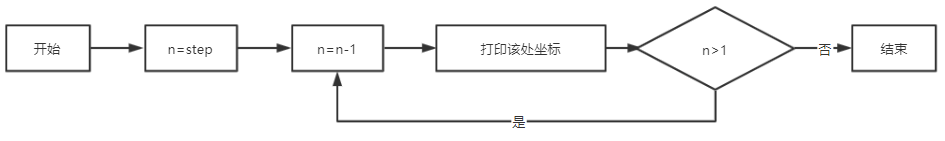
## 3.3 最短路径打印功能的实现

### 3.3.1 最短路径打印功能实现思路

最短路径打印有文字解和图形解两种实现，在最短路径搜索过程中result数组已经形成了一条从起点到终点数值依次减小的通路，而打印文字解只需从起点开始搜索并依次将递减序列出现的位置打印出来即可。由于数值为n处周围必存在数值为n-1的点(n>1)，所以依次打印得出的结果即为最短路径，但需注意的是数值为n处周围未必存在数值为n+1的点，故如若在广度优先搜索最短路径过程中从起点开始计数，则在打印结果的过程中会出现走入死胡同的情况，这也解释了为何3.2.1中我们需要从终点开始循环处理。

图形解的打印相对简单，仅需在所有数值大于1处打印‘\*’表示通路即可。其中将起点标记为S，终点标记为E。

### 3.3.2 最短路径打印功能实现流程图



### 3.3.3 最短路径打印功能实现代码

//打印迷宫地图

**void** maze::printMaze() {

    cout << "迷宫地图（S为起点，E为终点）：" << endl;

    cout << endl;

    cout << "  ";

**for** (**int** i = 0; i < length; i++) {

        cout << setw(5) << i << "列";

    }

    cout << endl;

**for** (**int** i = 0; i < width; i++) {

        cout << i << "行";

**for** (**int** j = 0; j < (4 - locate(i)); j++)

            cout << " ";

**for** (**int** j = 0; j < length; j++) {

**if** (i == 0 && j == 0)

                cout << "S" << "      ";

**else** **if** (i == width - 1 && j == length - 1)

                cout << "E" << "      ";

**else**

                cout << sign[road[i][j]] << "      ";

        }

        cout << endl;

        cout << endl;

    }

}

//打印路径文字解

**void** maze::printResult(**int** x1, **int** y1, **int** x2, **int** y2, **int** n) {

**if** (n == result[0][0])

        cout << "最短路径（文字解）：" << endl << endl;

    cout << "<" << x1 << "," << y1 << ">";

**if** (x1 != x2 || y1 != y2)

        cout << " ---> ";

**else**

        cout << endl;

    result[x1][y1] = -2;

**if** (x1 == x2 && y1 == y2) {

**return**;

    }

**if** (y1 - 1 >= 0 && result[x1][y1 - 1] == n - 1) {

        result[x1][y1 - 1] = n + 1;

        printResult(x1, y1 - 1, x2, y2, n - 1);

    }

**else** **if** (x1 - 1 >= 0 && result[x1 - 1][y1] == n - 1) {

        result[x1 - 1][y1] = n + 1;

        printResult(x1 - 1, y1, x2, y2, n - 1);

    }

**else** **if** (y1 + 1 < length && result[x1][y1 + 1] == n - 1) {

        result[x1][y1 + 1] = n + 1;

        printResult(x1, y1 + 1, x2, y2, n - 1);

    }

**else** **if** (x1 + 1 < width && result[x1 + 1][y1] == n - 1) {

        result[x1 + 1][y1] = n + 1;

        printResult(x1 + 1, y1, x2, y2, n - 1);

    }

}

//打印路径图形解

**void** maze::drawResult() {

    cout << endl;

    cout << "最短路径（图形解）：" << endl << endl;

    cout << "  ";

**for** (**int** i = 0; i < length; i++) {

        cout << setw(5) << i << "列";

    }

    cout << endl;

**for** (**int** i = 0; i < width; i++) {

        cout << i << "行";

**for** (**int** j = 0; j < (4 - locate(i)); j++)

            cout << " ";

**for** (**int** j = 0; j < length; j++) {

**if** (i == 0 && j == 0)

                cout << "S" << "      ";

**else** **if** (i == width - 1 && j == length - 1)

                cout << "E" << "      ";

**else** **if** (result[i][j] == -2)

                cout << "\*" << "      ";

**else**

                cout << sign[road[i][j]] << "      ";

        }

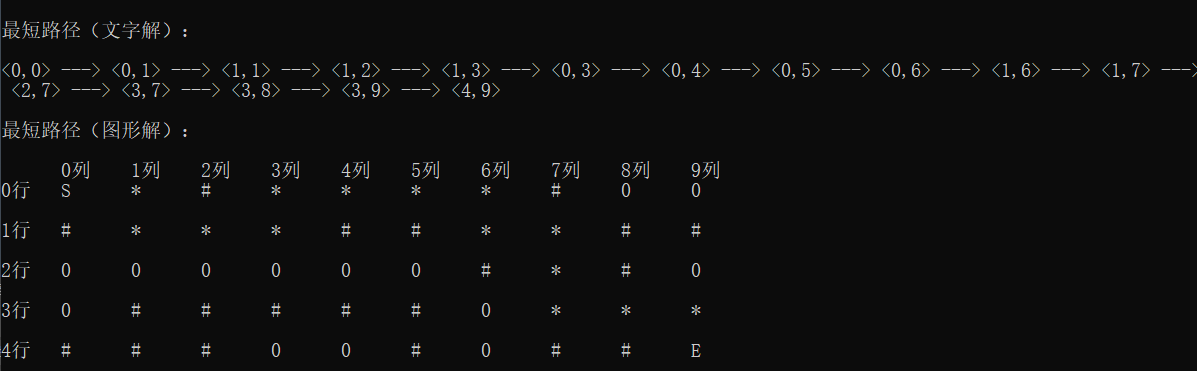
        cout << endl;

        cout << endl;

    }

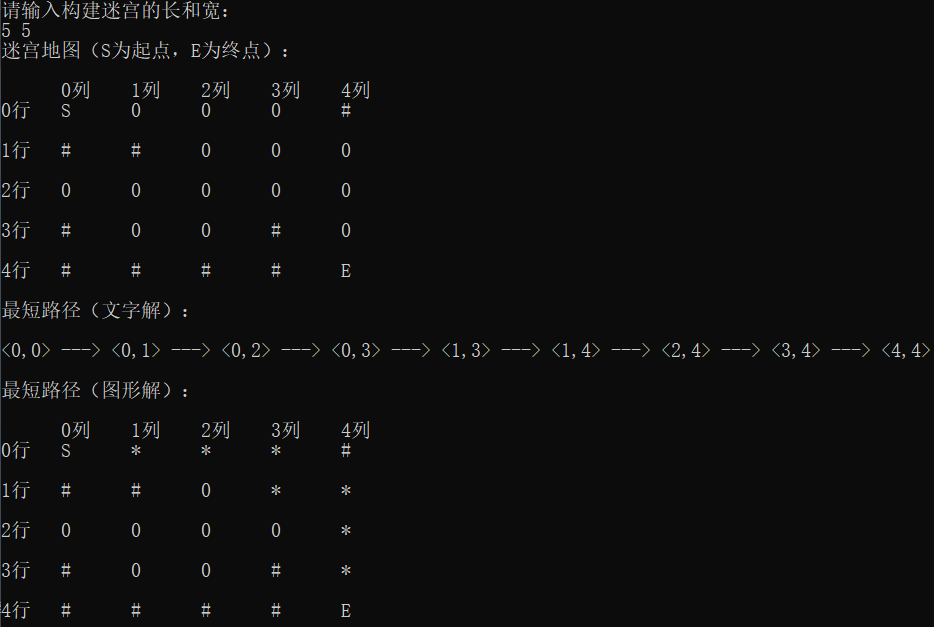
}

### 3.3.3 最短路径打印功能实现样例

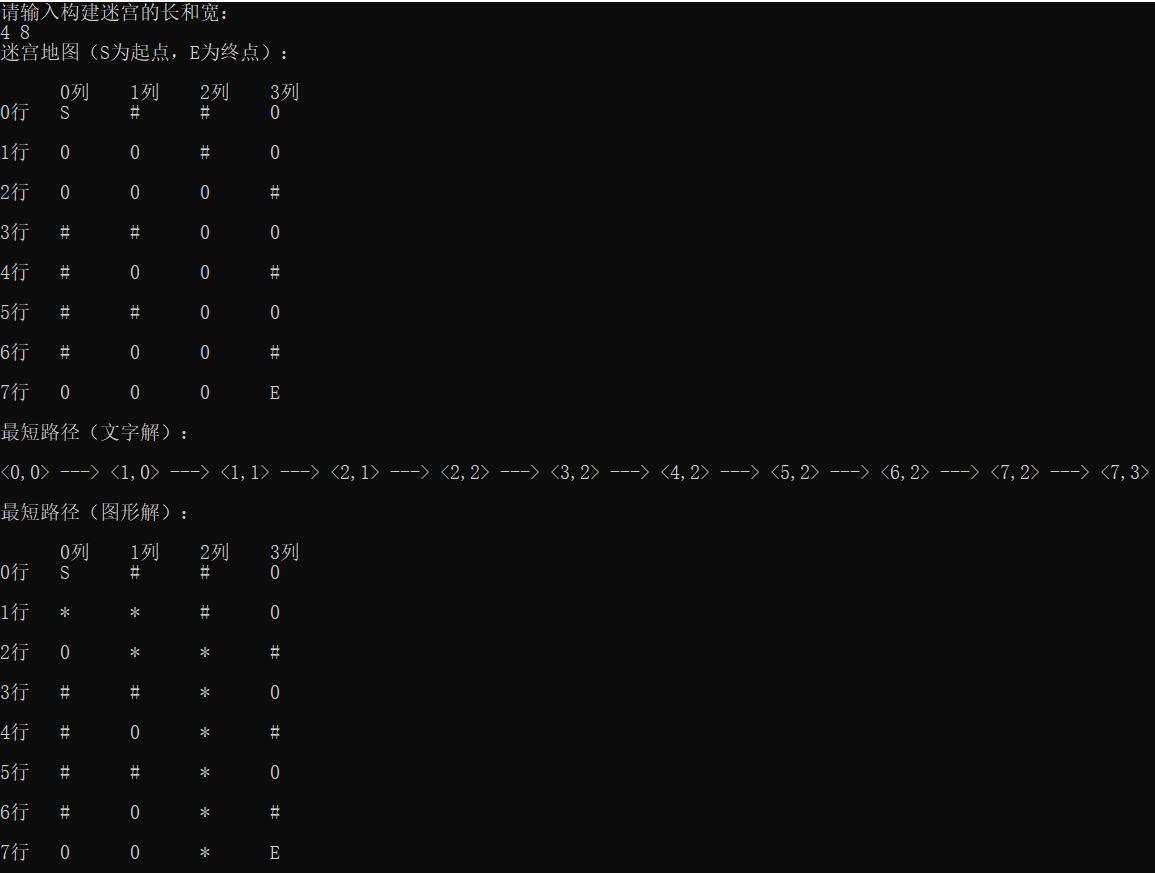


# 4 测试

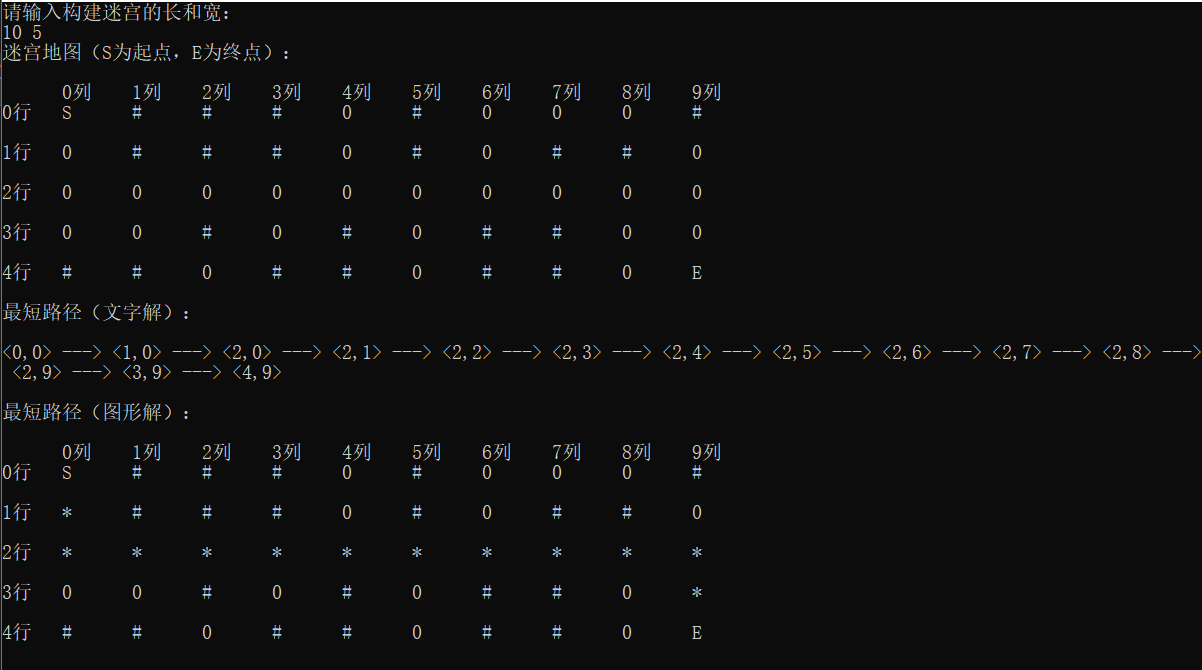
## 4.1 一般测试（5\*5地图）



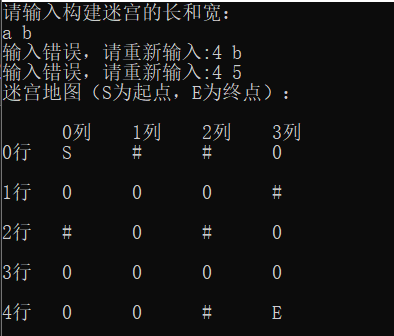
## 4.2 一般测试（8\*4地图）



## 4.3 一般测试（5\*10地图）



## 4.4 错误测试（用户输入长和宽非int类型）



## 4.5 错误测试（用户输入长和宽过大）

