《数据结构与算法》实验报告

<table>
<tr><td>实验名称</td><td colspan="5">迷宫求解问题</td></tr>
<tr><td>姓名</td><td>陈思睿</td><td>学号</td><td>21030021007</td><td>日期</td><td>2024.4.10</td></tr>
<tr><td>实验内容</td><td colspan="5">基于教材和课件讲解内容，利用自己实现的栈结构完成可运行的迷宫求解程序
2. （2 分）实现教材或课件中未给出的“可通”函数、“足迹打印”函数、“下一位置”函数、“打印不能通过的位置”函数等功能函数
3. （2 分）实现 MazeType 数据类型，及可能会用到的数据对象（如，入口、出口、位置属性是墙或通路）、数据关系（如，位置之间的相邻关系）、基本操作（如，返回某个坐标位置是墙还是通路）
4. （2 分）测试有通路和没通路等不同结果的输入迷宫
5. （2 分）尝试进一步完善迷宫求解程序，使得从入口到出口有多条简单路径的迷宫的所有路径都能求解出来，或者从多条可行的路径中给出最短路径。
拓展要求：（附加 3 分）
1.（2 分） 通过实验结果对比“入口-出口相对方向”和“探索方向的优先顺序”一致或不一致时，迷宫求解程序的运行效率。例如，当出口在入口的右下方向时，探索优先顺序是右下左上，或者上左下右时，程序运行“时间/效率/试过的位置数”是不一样的
2.（1 分）分析“可通”函数原理，解释为什么迷宫求解程序得到的路径不会有环</td></tr>
<tr><td>实验目的</td><td colspan="5">利用栈的基本操作来实现迷宫求解问题</td></tr>
</table>

实验步骤

1、基本结构：

创建基本数据对象，数据类型有迷宫结构 Mazetype，坐标结构 PosType，

栈元素结构 SElemType 和顺序栈结构 SqStack。其中后三种结构采取了课本上相同的数据元素、数据类型和命名方式。

```
typedef struct {
    int ord;    //通道序号
    PosType seat; //坐标位置
    int di;     //下一块通道方向
} SElemType;  //元素类型
```

```
typedef struct {
    int i;
    int j;    //通道在迷宫的坐标
} PosType;
```

```
typedef struct {
    SElemType *base;    //栈底指针
    SElemType *top;    //栈顶指针
    int stacksize;  //容量
} SqStack;
```

```
typedef struct {
    int **map;      //迷宫地图（二维数组建立）
    int row, col;    //迷宫的行列
} Mazetype;
```

2、基本操作：

为了实现迷宫求解问题，需要先实现栈的基本操作，另外栈内元素也较为复杂，包含多个值，因此本实验的操作较多，下面根据不同的数据类型来说明各种基本操作：

① 有关栈的基本操作

为了实现迷宫求解问题，需要采用栈。

栈顶基本操作有初始化、判空、删除一个栈，返回栈顶元素、以及最常见的入栈和出栈操作。

其中栈的基本操作和课本采用了类似写法

初始化一个栈

```
SqStack *InitStack(SqStack *S) {
    S->base = (SElemType *)malloc(STACK_INIT_SIZE * sizeof(SElemType));
    if (!S->base)
        exit(0);
    S->top = S->base;
    S->stacksize = STACK_INIT_SIZE;
    return S;
}
```

判断栈是否为空，是返回 1，否则返回 0

```
int IsEmpty(SqStack *S) {
    return (S->top == S->base);
}
```

销毁栈

```
void DestroyStack(SqStack *S) {
    free(S->base);
    S->base = NULL;
    S->top = NULL;
    S->stacksize = 0;
}
```

入栈

```
SqStack *Push(SqStack *S, SElemType e) {
    if (S->top - S->base >= S->stacksize) {
        S->base = (SElemType *)realloc(S->base, (S->stacksize + STACKINCREMENT) * sizeof(SElemType));
        if (!S->base)
            exit(0);
        S->top = S->base + S->stacksize;
        S->stacksize += STACKINCREMENT;
    }

    *(S->top) = e;
    S->top++;
    return S;
}
```

出栈：同时元素放在 e 中返回，如果为空，直接返回，否则删除栈顶元素

```
SqStack *Pop(SqStack *S, SElemType *e) {
    if (IsEmpty(S))
        return S;

    *e = *(--S->top);
    return S;
}
```

求栈顶元素

```
SElemType GetTop(SqStack *S) {
    if (!IsEmpty(S))
        return *(S->top - 1);
    else
        exit(0); // 如果栈空直接返回
}
```

② 迷宫的基本操作

迷宫地图元素的建立：（其中迷宫大小自定义）（采用二维数组方式）

```
void Create_Maze(Mazetype *M, int m, int n) { //创建迷宫地图
    int i, j;
    M->col = n;
    M->row = m;

    M->map = (int **)malloc(M->col * sizeof(int *)); //申请迷宫内
    if (!M->map)
        exit(0);

    for (i = 0; i < M->col; i++) {
        M->map[i] = (int *)malloc(M->row * sizeof(int));
        if (!M->map[i])
            exit(0);
    }
```

其中迷宫地图的创建通过随机数建立，边缘位置默认为墙，其他地方遍历整个数组，采用随机数方式赋值为 1 或 0，其中 1 为通路，0 为墙。

```
srand((unsigned)time(NULL));
for (i = 0; i < M->col; i++) {    //生成地图，默认0为墙，1可以行走
    for (j = 0; j < M->row; j++) {
        if (i == 0 || j == 0 || i == M->col - 1 || j == M->row - 1)
            M->map[i][j] = 0; // 设置边缘，迷宫边缘必须为墙
        else
            M->map[i][j] = (rand() % 3 == 0) ? 0 : 1; //随机生成地图
    }
}
```

返回某个坐标位置是墙还是通路

根据地图创立可知，直接通过坐标返回 map[i][j]的值来判断墙还是通路

```c
int determine_Maze(Mazetype *M, int m, int n){    //返回地图坐标m、n处是通路还是墙
    if(M->map[m][n]==0){    //墙返回1
        return 0;
    }
    return 1;               //通路返回0
}
```

下一位置函数

下一位置函数根据地图默认原则为左上为起点，而右下为终点，因此优先方式为右下左上，如果采用其他位置的话，只需要修改优先原则的顺序即可。

```c
PosType NextPos(PosType pos, int di) {
    switch (di) {
        case 1: pos.j++; break; // 右
        case 2: pos.i++; break; // 下
        case 3: pos.j--; break; // 左
        case 4: pos.i--; break; // 上
    }
    return pos;
}
```

足迹打印函数

足迹打印函数主要和下面的可通函数有关，建立地图时，a[i][j]只有 0 和 1 表示是否通路，可通函数则采用穷举的思想，每次经过时会把当前位置修改，其中 0 本身就是障碍，因此用*表示为障碍，1 为通路，故通过空格打印，每次走到一个地方，如果可通，则要把其改为 2，同时入栈，如果后面是死路，即要退栈时，则需要将 2 改回 1(**这块只能改为 1，因为 0 是障碍，显然不可能入栈**)

```c
void Print_Maze(Mazetype *M) {
    int i, j;

    printf("  ");
    for (i = 0; i < M->row; i++)
        printf("%-2d", i);
    printf("\n");

    for (i = 0; i < M->col; i++) {
        printf("%-2d", i);
        for (j = 0; j < M->row; j++) {
            switch (M->map[i][j]) {
                case 0: printf("* "); break; // 障碍
                case 1: printf("  "); break; // 通路
                case 2: printf("+ "); break; // 足迹
                case 4: printf("S "); break; // 起点
                case 5: printf("E "); break; // 终点
                default: printf("  "); break;
            }
        }
        printf("\n");
    }
}
```

寻找通路函数

调用栈、首先起点肯定需要入栈，然后本次的顺序采用右下左上，探测相邻路径，如果是 1 即未走过的路、则调入栈中，然后更新当前位置，继续向下走、如果当前位置四周没有 1，即周围都是墙或者走过的路径、那么要退栈，同时更新当前位置。

```
void FindOut(int **maze, PosType Start, PosType End) {
    SqStack S;
    SElemType e;
    PosType position = Start;
    int curstep = 1;
    InitStack(&S);
    do {
        if (maze[position.i][position.j] == 1) {  //1表面是未走过的可通路径
            maze[position.i][position.j] = 2;   //    把路径标记为2，然后入栈
            e.di = 1;
            e.seat.i = position.i;
            e.seat.j = position.j;
            e.ord = curstep;
            Push(&S, e);

            if (position.i == End.i && position.j == End.j)//终点
                break;

            position = NextPos(position, 1);
            curstep++;      //继续探索
        } else {
            if (!IsEmpty(&S)) {   //空栈直接退出
                Pop(&S, &e);
                curstep--;
                while (e.di == 4 && !IsEmpty(&S)) {
                    maze[e.seat.i][e.seat.j] = 3; //
                    Pop(&S, &e);
                    curstep--;
                }

                if (e.di < 4) {
                    e.di++;
                    Push(&S, e);
                    curstep++;
                    position = NextPos(e.seat, e.di);
                }
            }
        }
    } while (!IsEmpty(&S));
```

```
if (IsEmpty(&S)) {
    maze[Start.i][Start.j] = 1; //
    printf("\nNo solution.\n");
} else {
    printf("\nPath found:\n");
    while (!IsEmpty(&S)) {
        e = GetTop(&S);
        printf("(%d,%d) -> ", e.seat.i, e.seat.j);  //打印路径
        Pop(&S, &e);
    }
    printf("\n");
}

DestroyStack(&S);
```

如果检测到栈空、说明没有通路、那么直接输出没有通路

打印路径，因为当走到终点时、栈中存着的都是走过的路、因此直接打印栈中元素即可、每打印一次，就退一次栈，直到栈为空，路径打印完毕。

3、实验结果：

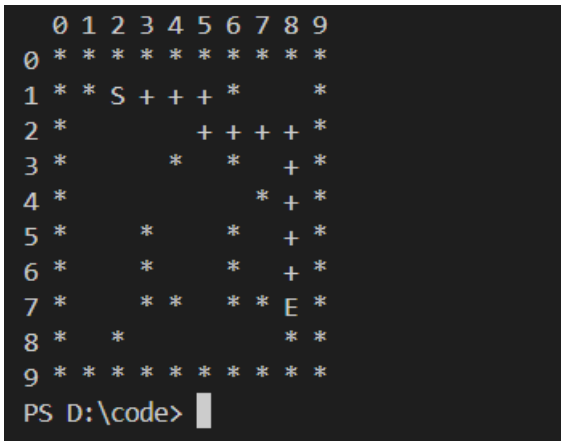
1、设置大小为 10x10 的迷宫，根据地图，设置（1,2）为起点（start），（7,8）为终点（end）

```
Enter Maze Size (LSize RSize): 10 10
  0 1 2 3 4 5 6 7 8 9
0 * * * * * * * * * *
1 * *         *     *
2 *                 *
3 *       *   *     *
4 *             *   *
5 *     *     *     *
6 *     *     *     *
7 *     * *   * *   *
8 *   *           * *
9 * * * * * * * * * *
Enter Start (Si Sj): 1 2
Enter End (Ei Ej): 7 8
```

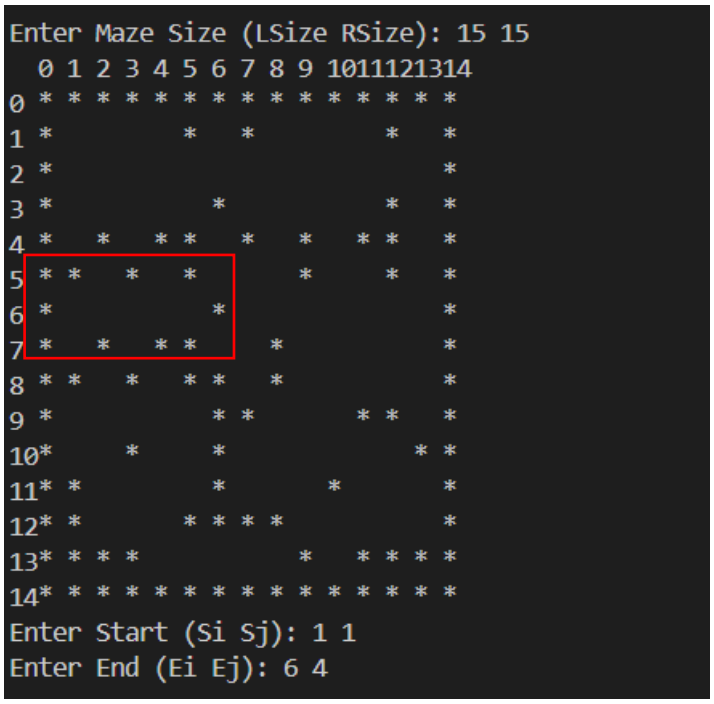
输出结果

足迹输出

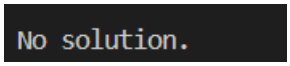
```
Path found:
(7,8) -> (6,8) -> (5,8) -> (4,8) -> (3,8) -> (2,8) -> (2,7) -> (2,6) -> (2,5) -> (1,5) -> (1,4)
-> (1,3) -> (1,2) ->
```

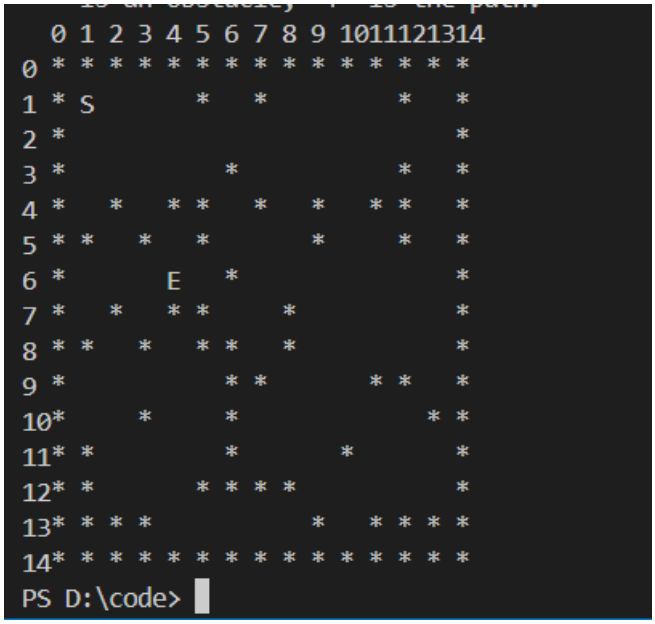


2、当起点不能到达终点时，下方地图显然在（6,4）出不能通过



运行结果



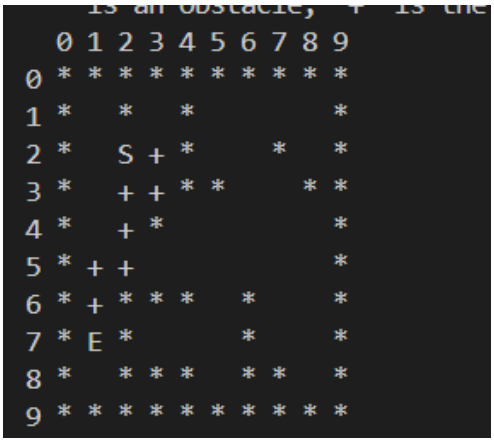


并没有可以通过的路径

4、改进探索算法：

对于该算法、本质上采用穷举的思想，因此得到的不一定是最短路径，如右下图，实际上由于终点在起点的左下，而算法仍然需要往右下走、因此显然不是最短路径。

于是需要修改 findout 函数、来寻求最短路径，



改进思路：寻求最短路径算法时、采用 dfs 来求、需要构造队列、将起点加入队列。

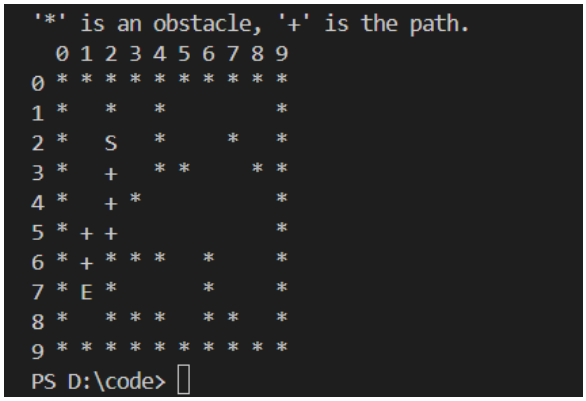
如果队列非空，则将队首位置出队，然后再将相邻元素加入队列。如果直到队列空、都没有找到终点、说明没有通路。

```
void FindShortestPath(int **maze, PosType Start, PosType End, int line, int row) {
    SqQueue Q = InitQueue();
    PosType curpos;
    SElemType e;
    bool found = false;

    EnQueue(&Q, (SElemType){1, Start, 1}); //  BFS 从起点位置开始
    while (!QueueEmpty(Q) && !found) {
        e = DeQueue(&Q);
        curpos = e.seat;

        for (int di = 1; di <= 4; di++) {
            PosType nextpos = NextPos(curpos, di);
            if (nextpos.i >= 0 && nextpos.i < line && nextpos.j >= 0 && nextpos.j < row &&
                maze[nextpos.i][nextpos.j] == 1) {  //当相邻的结点时没有走过的路且是通路时
                maze[nextpos.i][nextpos.j] = 2; //标记走过的路径为2
                EnQueue(&Q, (SElemType){e.ord + 1, nextpos, di});
                if (nextpos.i == End.i && nextpos.j == End.j) {  //当找到终点后、
                    found = true;   //found标记为true
                    break;
                }
            }
        }
    }
```

```
if (!found) {
    printf("\nNo solution found.\n");  //没有找到
} else {
    printf("\nShortest Path from (%d,%d) to (%d,%d):\n", Start.i, Start.j, End.i, End.j);
    curpos = End;
    while (curpos.i != Start.i || curpos.j != Start.j) {
        maze[curpos.i][curpos.j] = 2; // 标记走过的路径
        for (int di = 1; di <= 4; di++) {
            PosType prevpos = NextPos(curpos, di);
            if (prevpos.i >= 0 && prevpos.i < line && prevpos.j >= 0 && prevpos.j < row &&
                maze[prevpos.i][prevpos.j] == maze[curpos.i][curpos.j] - 1) {
                curpos = prevpos;
                break;
            }
        }
    }
    maze[Start.i][Start.j] = 4;
    maze[End.i][End.j] = 5;
    PrintMaze(maze, line, row);
}
```



对比改进算法、显然少走了多余步数、直接是最短路径。

5、拓展

① 通过实验结果对比“入口-出口相对方向”和“探索方向的优先顺序”一致或不一致时，迷宫求解程序的运行效率。例如，当出口在入口的右下方向时，探索优先顺序是右下左上，或者上左下右时，程序运行“时间/效率/试过的位置数”是不一样的

只需要修改 NextPos（）函数中右下左上的位置即可。

```
int count=0;
```

```
count ++;
```

另外在可通函数中定义一个变量 count，用来记录入栈的次数（即试过的次数）

设置计时器，主要统计 findout 的执行时间，另外此次因为统计时间要求，迷宫较大，故不便于展示输出过程，仅仅展示试过的次数和花费时间。

```
LARGE_INTEGER frequency;
LARGE_INTEGER start, end;
double cpu_time_used;

// 获取计时器频率
QueryPerformanceFrequency(&frequency);

// 记录开始时间
QueryPerformanceCounter(&start);
```

```
// 记录结束时间
 QueryPerformanceCounter(&end);

 // 计算执行时间（单位：秒）
 cpu_time_used = (double)(end.QuadPart - start.QuadPart) / frequency.QuadPart;

 printf("程序运行时间为 %.6f 秒\n", cpu_time_used);
```

设置起点在右上，而终点在左下（优先级肯定是先往右、下走最快）

1、 当顺序是右下左上时

```
试过的次数为1278程序运行时间为 0.021575 秒
```

2、 当顺序改为下右上左时

```
试过的次数为407程序运行时间为 0.016803 秒
```

1、2 两种顺序都是先往右下走、不过 1、2 的右和下的优先级不同，2 的情况下试过的次数更小，时间更少。

3、 当顺序改为左上右下时

```
试过的次数为5111程序运行时间为 0.112115 秒
```

4、 当顺序改为上左下右时

```
试过的次数为4755程序运行时间为 0.119966 秒
```

总结：其中 1 与 4、2 与 3 顺序都是相反的，1、2 较快、3、4 最慢。1、2 中都是右下优先、3、4 是左上优先，（其中 1、2 仅仅右和下的顺序不同）

若将最快的时间作为 1 时、其他三种顺序分别消耗的时间从快到慢分别是第二种的 1.284 倍（1）、6.67 倍（3）、7.140 倍（4）.试过的次数其他三种分别是最快的 3.14 倍（1）、11.68 倍（4）、12.55 倍（3）.

显然，优先级不同、其运行的时间相差也会较大。而优先级的顺序和起点、终点的位置有关。

另外、对于上述实验中，如果是右下优先的话，那么对于右和下，到达谁更优先？本次实验采用的是一个正方形地图，因此横纵长度相同、右和下应该和地位相等但实验中依然有 20%的差距，我认为和地图内部有关，地图内部的布局依然会影响到具体而言的右、下、或者左、上到底谁更快一点。

② 分析“可通”函数原理，解释为什么迷宫求解程序得到的路径不会有环

在此，尚需说明一点的是，所谓当前位置可通，指的是未曾走到过的通道块，即要求该方块位置不仅是通道块，而且既不在当前路径上（否则所求路径就不是简单路径），也不是曾经纳入过路径的通道块（否则只能在死胡同内转圈）。

根据课本的可通函数定义以及代码所写，

```
if (maze[position.i][position.j] == 1) {  //1表面是未走过的可通路径
    maze[position.i][position.j] = 2;   //    把路径标记为2，然后入栈
    e.di = 1;
    e.seat.i = position.i;
    e.seat.j = position.j;
    e.ord = curstep;
    Push(&S, e);
```

只有没走过的路，即 map[i][j]为 1 的路径才会入栈，其他的如 0（墙），2（走过的路）都不会再次走过，因此只能是简单路径，而不会重复走，故不会有环。

迷宫求解源代码

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include<windows.h>

#define STACK_INIT_SIZE 20//存储空间初始分配量
#define STACKINCREMENT 10//存储空间配增量

typedef struct {
    int i;
    int j;   //通道在迷宫的坐标
} PosType;

typedef struct {
    int ord;   //通道序号
    PosType seat; //坐标位置
    int di;    //下一块通道方向
} SElemType;  //元素类型

typedef struct {
    SElemType *base;   //栈底指针
    SElemType *top;   //栈顶指针
    int stacksize;  //容量
} SqStack;

typedef struct {
    int **map;     //迷宫地图（二维数组建立）
    int row, col;   //迷宫的行列
} Mazetype;

SqStack *InitStack(SqStack *S) {
    S->base = (SElemType *)malloc(STACK_INIT_SIZE * sizeof(SElemType));
    if (!S->base)
        exit(0);
    S->top = S->base;
    S->stacksize = STACK_INIT_SIZE;
    return S;
}

int IsEmpty(SqStack *S) {
    return (S->top == S->base);
}

void DestroyStack(SqStack *S) {
    free(S->base);
```

```
    S->base = NULL;
    S->top = NULL;
    S->stacksize = 0;
}

SqStack *Push(SqStack *S, SElemType e) {
    if (S->top - S->base >= S->stacksize) {
        S->base = (SElemType *)realloc(S->base, (S->stacksize + STACKINCREMENT) *
sizeof(SElemType));
        if (!S->base)
            exit(0);
        S->top = S->base + S->stacksize;
        S->stacksize += STACKINCREMENT;
    }

    *(S->top) = e;
    S->top++;
    return S;
}

SqStack *Pop(SqStack *S, SElemType *e) {
    if (IsEmpty(S))
        return S;

    *e = *(--S->top);
    return S;
}

SElemType GetTop(SqStack *S) {
    if (!IsEmpty(S))
        return *(S->top - 1);
    else
        exit(0); // 如果栈空直接返回
}
int determine_Maze(Mazetype *M, int m, int n){   //返回地图坐标 m、n 处是通路还是墙
     if(M->map[m][n]==0){   //墙返回 1
        return 0;
     }
     return 1;              //通路返回 0
}
void Create_Maze(Mazetype *M, int m, int n) { //创建迷宫地图
    int i, j;
    M->col = n;
    M->row = m;

    M->map = (int **)malloc(M->col * sizeof(int *)); //申请迷宫内存
    if (!M->map)
        exit(0);
```

```
    for (i = 0; i < M->col; i++) {
        M->map[i] = (int *)malloc(M->row * sizeof(int));
        if (!M->map[i])
            exit(0);
    }

     srand(123); //srand((unsigned)time(NULL));
    for (i = 0; i < M->col; i++) {    //生成地图，默认 0 为墙，1 可以行走
        for (j = 0; j < M->row; j++) {
            if (i == 0 || j == 0 || i == M->col - 1 || j == M->row - 1)
                M->map[i][j] = 0; // 设置边缘，迷宫边缘必须为墙
            else
             M->map[i][j] = (rand() % 3 == 0) ? 0 : 1; //随机生成地图
        }
    }
}

void Print_Maze(Mazetype *M) {
    int i, j;

    printf("  ");
    for (i = 0; i < M->row; i++)
        printf("%-2d", i);
    printf("\n");

    for (i = 0; i < M->col; i++) {
        printf("%-2d", i);
        for (j = 0; j < M->row; j++) {
            /*if((i==3||(i==2))&&j==3){
               printf("  ");
            }
            else{*/switch (M->map[i][j]) {
                case 0: printf("* "); break; // 障碍
                case 1: printf("  "); break; // 通路
                case 2: printf("+ "); break; // 足迹
                case 4: printf("S "); break; // 起点
                case 5: printf("E "); break; // 终点
                default: printf("  "); break;}
            //}
        }
        printf("\n");
    }
}

PosType NextPos(PosType pos, int di) {
    switch (di) {
        case 1: pos.j++; break; // 右
        case 2: pos.i++; break; // 下
        case 3: pos.j--; break; // 左
```

```
        case 4: pos.i--; break; // 上
    }
    return pos;
}

void FindOut(int **maze, PosType Start, PosType End) {

    SqStack S;
    SElemType e;
    PosType position = Start;
    int curstep = 1;
    int count=0;
    InitStack(&S);
    do {
        if (maze[position.i][position.j] == 1) {  //1 表面是未走过的可通路径
            maze[position.i][position.j] = 2;   //   把路径标记为 2，然后入栈
            count ++;
            e.di = 1;
            e.seat.i = position.i;
            e.seat.j = position.j;
            e.ord = curstep;
            Push(&S, e);

            if (position.i == End.i && position.j == End.j)//终点
                break;

            position = NextPos(position, 1);
            curstep++;      //继续探索
        } else {
            if (!IsEmpty(&S)) {   //空栈直接退出
                Pop(&S, &e);
                curstep--;

                while (e.di == 4 && !IsEmpty(&S)) {
                    maze[e.seat.i][e.seat.j] = 3; //
                    Pop(&S, &e);
                    curstep--;
                }

                if (e.di < 4) {
                    e.di++;
                    Push(&S, e);
                    curstep++;
                    position = NextPos(e.seat, e.di);
                }
            }
        }
    } while (!IsEmpty(&S));
```

```
    if (IsEmpty(&S)) {
        maze[Start.i][Start.j] = 1; //
        printf("\nNo solution.\n");
    } else {
        printf("\nPath found:\n");
        while (!IsEmpty(&S)) {
            e = GetTop(&S);
            printf("(%d,%d) -> ", e.seat.i, e.seat.j);  //打印路径
            Pop(&S, &e);
        }
        printf("\n");
        printf("试过的次数为%d",count);
    }

    DestroyStack(&S);
}

int main() {
    int Si, Sj, Ei, Ej;
    int LSize, RSize;
    Mazetype M;
    PosType Start, End;

    printf("Enter Maze Size (LSize RSize): ");
    scanf("%d %d", &LSize, &RSize);

    Create_Maze(&M, LSize, RSize);
    Print_Maze(&M);

    printf("Enter Start (Si Sj): ");
    scanf("%d %d", &Si, &Sj);

    while (Si >= LSize || Sj >= RSize || M.map[Si][Sj] == 0) {
        printf("Invalid Start. Enter Start (Si Sj): ");
        scanf("%d %d", &Si, &Sj);
    }

    printf("Enter End (Ei Ej): ");
    scanf("%d %d", &Ei, &Ej);

    while (Ei >= LSize || Ej >= RSize || M.map[Ei][Ej] == 0) {
        printf("Invalid End. Enter End (Ei Ej): ");
        scanf("%d %d", &Ei, &Ej);
    }
    LARGE_INTEGER frequency;
    LARGE_INTEGER start, end;
    double cpu_time_used;
```

```
    // 获取计时器频率
    QueryPerformanceFrequency(&frequency);

    // 记录开始时间
    QueryPerformanceCounter(&start);



    Start.i = Si;
    Start.j = Sj;
    End.i = Ei;
    End.j = Ej;

    FindOut(M.map, Start, End);

    M.map[Start.i][Start.j] = 4; // Mark start
    M.map[End.i][End.j] = 5; // Mark end

  // 记录结束时间
    QueryPerformanceCounter(&end);

    // 计算执行时间（单位：秒）
    cpu_time_used = (double)(end.QuadPart - start.QuadPart) / frequency.QuadPart;

    printf("程序运行时间为 %.6f 秒\n", cpu_time_used);



    printf("\n'*' is an obstacle, '+' is the path.\n");
    Print_Maze(&M);

    // Free dynamically allocated memory
    for (int i = 0; i < M.col; i++)
        free(M.map[i]);
    free(M.map);

    return 0;
}
```

实验总结

本次实验、通过迷宫求解问题来理解了栈的作用、为了求解迷宫问题、需要先实现栈的各种操作、通过本次实验、使我熟练掌握了栈的各种基本操作和实现。另外本次实验对于迷宫求解算法也有较多思考，对于迷宫的穷举优先顺序不同、导致其尝试次数、运行时间不同、通过对于相同的迷宫，由于顺序优先不同、其运行时间、效率、尝试次数差别巨大，通过分析这些结果来得到结论、也是本次实验的一个巨大收获、本次实验时间较长、各种过程也较为复杂、但很好的锻炼了自己。