项目文档

谈瑞

项目三：迷宫问题

专业班级：软件工程 4 班 学号：1452775 课目：数据结构课程设计
电话：18936361545 电子邮件：tanrui106@163.com 个人网站：http://guitoubing.top

目录

项目简介

项目概要

迷宫只有两个门，一个门叫入口，另一个门叫出口。一个骑士骑马从入口进入迷宫，迷宫设置很多障碍，骑士需要在迷宫中寻找通路以到达出口。

项目功能及要求

迷宫问题的求解过程可以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就结束了。

项目结构

本项目总体概略结构是在使用正交循环双向链表实现迷宫的构造，然而若迷宫的每个节点均存储在链表中会浪费很多资源，于是这里采用了类似稀疏矩阵的构造方法，首先将迷宫的行列头放在矩阵中初步构建出 n*m 的矩阵，而后将迷宫中的通路按照位置放入矩阵中，构建成正交矩阵迷宫。

当然，这不是必要的，实际上对于这样的迷宫，仅使用二维数组、二维链表甚至一维数组、一维链表就可以轻松地解决，但是为了学以致用，同时检验自己链表知识的掌握程度，于是采用了上课所学的正交循环双向链表来构建迷宫。通往成功的路总是坎坷的，这是我第一次接触正交循环双向链表，因此在实现迷宫时出现了这样那样的错误：指针指向了未知空间、释放指针时读取内存错误、行列号混淆等等，但最终都得到了解决，最后正确的实现了迷宫，如果缺少了这些漫途荆棘，就尝不到这来自成功的喜悦了。

项目类的实现

Maze 类

采用正交循环双向链表构建迷宫

	类成员	作用
public 成员	Maze();	Maze 类的默认构造函数，此处读取文件流，同时读取迷宫构建正交链表
	~Maze();	Maze 类的默认析构函数，在程序结束后释放正交链表中所有空间
protected 成员	Node *findNodeInRow(int, Node*);	在一行中找某节点，若找到返回该节点，否则返回 NULL
	void printMaze(ostream &os = cout);	在指定的os输出流中输出Maze，规定其默认值为cout
	void printVec(ostream &os = cout);	打印最后成功找到出口的路线，若找不到则输出另一段话并退出
	void findRoad(bool, bool, bool, bool, Node*);	在四个方向分别寻找出口，找到了终点就返回，否则递归寻找
	void justGo();	定义出口，并开始调用 findNodeInRow
private 成员	Node *head, *present;	定义链表的头结点（并非起点）及现在指向的结点
	int row, line;	存储迷宫的长宽
	fstream in_file;	定义输入文件，文件内包含迷宫的长宽及迷宫形状
	vector<Node*> road_vec;	存放通路路径结点
	bool isEnd;	是否已经找到出口

Node 类

Node 类存储了迷宫的结点

	类成员	作用
public 成员	friend class Maze;	声明 Maze 类为 Node 类的友元函数
	Node();	Node 类默认构造函数
	Node(char);	Node 类重载构造函数，通过参数初始化结点的值
	~Node();	Node 类默认析构函数
private 成员	Node *left, *right, *up, *down;	上下左右四个指针指向其他Node实例
	int x_index, y_index;	定义横纵坐标
	bool is_road;	判断当前Node是否可以作为通路
	char data;	存储数据，除行列头为‘#’外，其他均为‘0’

主要代码分析

Maze.cpp

Maze 类默认构造函数

```
/*
    由于迷宫是从文件中读取，因此Maze类中数据不需要重载构造函数来初始化，用户要想改变迷宫形状只需要更改in_file.txt文件即可。这里我将Maze类所有的操作函数定义为protected，因为对于用户来说只希望更改迷宫文件就能找到路径，而不需要其他的操作。这些操作函数全部在构造函数内被调用或者是间接调用。
*/

Maze::Maze() {
        head = present = NULL;
        row = line = 0;
        in_file.open("in_file.txt");
        in_file >> row >> line;
        //这里的row和line是行号，而不是行数，要注意行号是比行数少1的
        for (int i = 0; i <= row; i++){
                //外侧循环读取列头，内侧循环从相应列头读取行
                Node *_node = new Node();
                in_file >> _node->data;
                _node->y_index = i;
                _node->x_index = 0;
                if (_node->data != '#') _node->is_road = true;
                //如果遇到的不是"#"，则判定为该结点是路，可以添加到Maze正交链表中
                present = _node;

                Node *temp_head1 = head;
                if (i == 0) head = _node;
                else {
                        temp_head1 = _node;
                        Node *temp_head2 = head;
                        for (int k = 1; k < i; k++){
                                temp_head2 = temp_head2->down;
                        }
                        temp_head2->down = temp_head1;
                        temp_head1->up = temp_head2;
                }//若是首列头结点，则直接赋值给head，否则要将其与之前的列头结点连在一起

                if (i == row)       {
                        temp_head1->down = head;
                        head->up = temp_head1;
```

```
        }//首尾相连，完成首列循环

        for (int j = 1; j <= line; j++){
            //从列头下一个开始存储行
            Node *_node = new Node();
            in_file >> _node->data;
            _node->y_index = i;
            _node->x_index = j;
            if (_node->data != '#') _node->is_road = true;
            if (i != 0 && j != 0) {
                if (!_node->is_road){
                    //如果待插入结点不是可走的路，那要判断他是不是行尾，若是，则需要将前一个路结点与行头相连，否则直接跳过
                    if (j == line) {
                        Node *temp_head2 = head;
                        for (int k = 1; k <= i; k++) {
                            temp_head2 = temp_head2->down;
                        }
                        present->right = temp_head2;
                        temp_head2->left = present;
                    }
                    continue;
                }
            }
            _node->left = present;
            present->right = _node;
            present = _node;
            if (j == line) {
                //如果该节点已经插入，也要判断其是否为行尾，同上
                Node *temp_head2 = head;
                for (int k = 1; k <= i; k++) {
                    temp_head2 = temp_head2->down;
                }
                present->right = temp_head2;
                temp_head2->left = present;
            }
        }
    }
    //上述操作仅将各行内结点、各行行头结点连在了一起，以下操作将出首列外的每一列连起来
    Node *temp_head, *temp_line1 = head;
    for (int i = 1; i <= line; i++){
        temp_head = head;
        temp_line1 = temp_line1->right;
        Node *temp_line2 = temp_line1;
        for (int j = 1; j <= row+1; j++){
```

```
                        temp_head = temp_head->down;
                        Node *find_node = findNodeInRow(i, temp_head);
                        if (j == row + 1) {
                                temp_line2->down = temp_line1;
                                temp_line1->up = temp_line2;
                                continue;
                        }
                        if (find_node != NULL) {
                                temp_line2->down = find_node;
                                find_node->up = temp_line2;
                                temp_line2 = find_node;
                        }
                }
        }

        printMaze();
        justGo();
}
```

Maze 类默认析构函数

```
/*
    由于Maze采用的是正交链表，因此析构时需要将每个结点空间全部释放，这是一种递归释放，但是这里把递归解开成非递归循环，逐行释放空间知道链表为空。
*/

Maze::~Maze(){
        if (head == NULL){
                //空迷宫直接返回，无需析构
                return;
        }
        //非空迷宫需从头结点开始遍历每一个元素，使得所有结点指针均被释放，最后留下头结点，删除之
        Node *temp_head = head, *temp_line = head;
        for (int i = 0; i <= row; i++){
                for (int j = 0; j <= line; j++){
                        //这里删除采用和构造时一样的顺序，先找列头，而后从列头往后删除结点，直到所删除结点的right结点是列头，即停止，并删除列头
                        if (temp_line->right == head) {
                                //此处判断是否到达列尾
                                delete temp_line;
                                if (head->data != '#'){
                                        break;
                                }
                                temp_head = head;
```

```
                        head = head->down;
                        delete temp_head;
                        j = line + 1;
                    }
                    else{
                        temp_head = temp_head->right;
                        temp_line = temp_head->right;
                        if (temp_head == temp_line){
                            break;
                        }
                        delete temp_head;
                        temp_head = temp_line;
                        if (temp_line == head) {
                            temp_head = head;
                            head = head->down;
                            delete temp_head;
                            j = line + 1;
                        }
                    }
                }
                temp_head = head;
            }
            head = NULL;
            //最后释放头结点，Maze为空
}
```

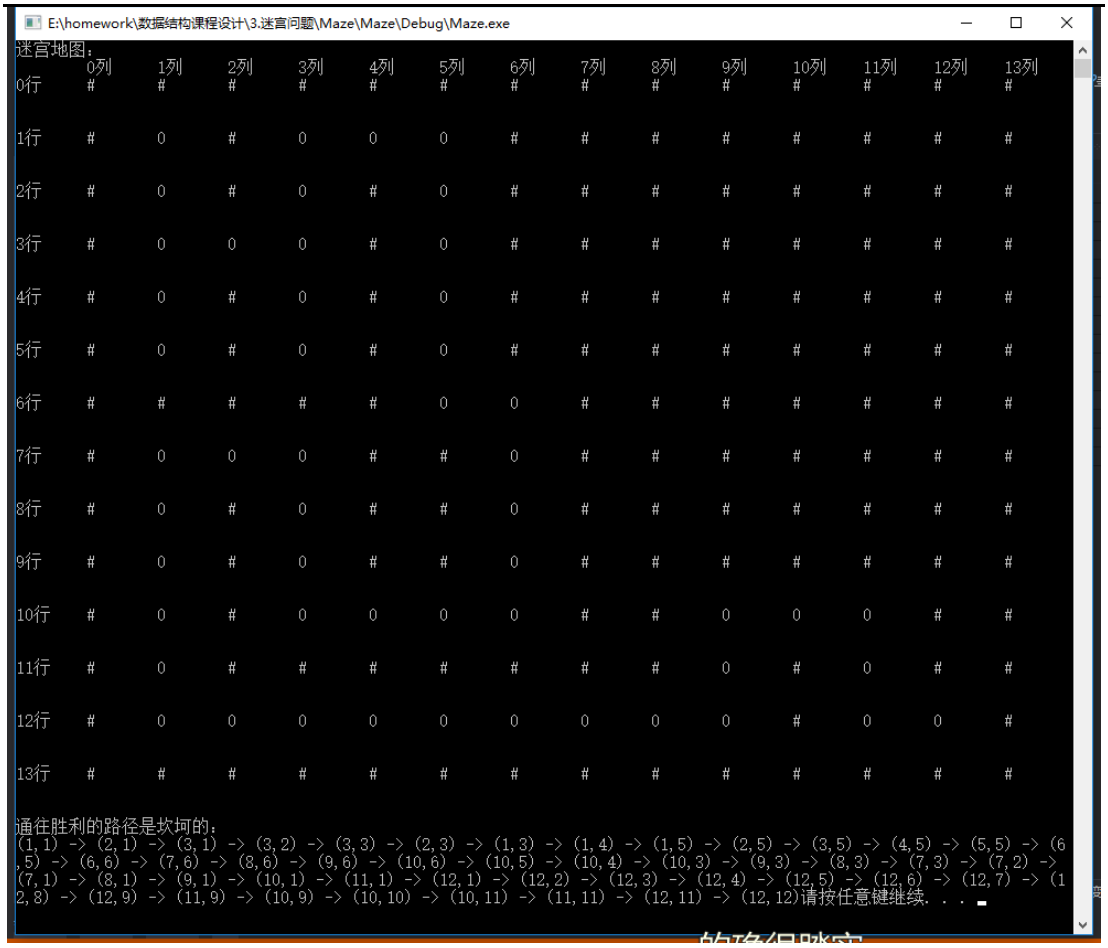
findRoad 函数

```
void Maze::findRoad(bool _right, bool _down, bool _left, bool _up, Node *_present) {
        road_vec.push_back(_present);
        if (_present->x_index == line - 1 && _present->y_index == row - 1) {
            isEnd = true;
            return;
        }//找到了终点就返回

        if (_present->x_index == 1) _left = false;
        if (_present->x_index == line-1) _right = false;
        if (_present->y_index == 1) _up = false;
        if (_present->y_index == row-1) _down = false;
        //靠边界的结点就不再找边界结点了

        if (_right && _present->right->x_index == _present->x_index+1 && !isEnd) {
            findRoad(true, true, false, true, _present->right);
        }
        if (_down && _present->down->y_index == _present->y_index+1 && !isEnd){
```

```
                findRoad(true, true, true, false, _present->down);
            }
            if (_left && _present->left->x_index == _present->x_index - 1 && !isEnd) {
                findRoad(false, true, true, true, _present->left);
            }
            if (_up && _present->up->y_index == _present->y_index - 1 && !isEnd) {
                findRoad(true, false, true, true, _present->up);
            }
            if (!isEnd) {
                road_vec.pop_back();
        }//按照右、下、左、上的顺序搜索下一个路结点，如果没有路了，那同时该结点必然不是终点，则弹出该结点
}
```

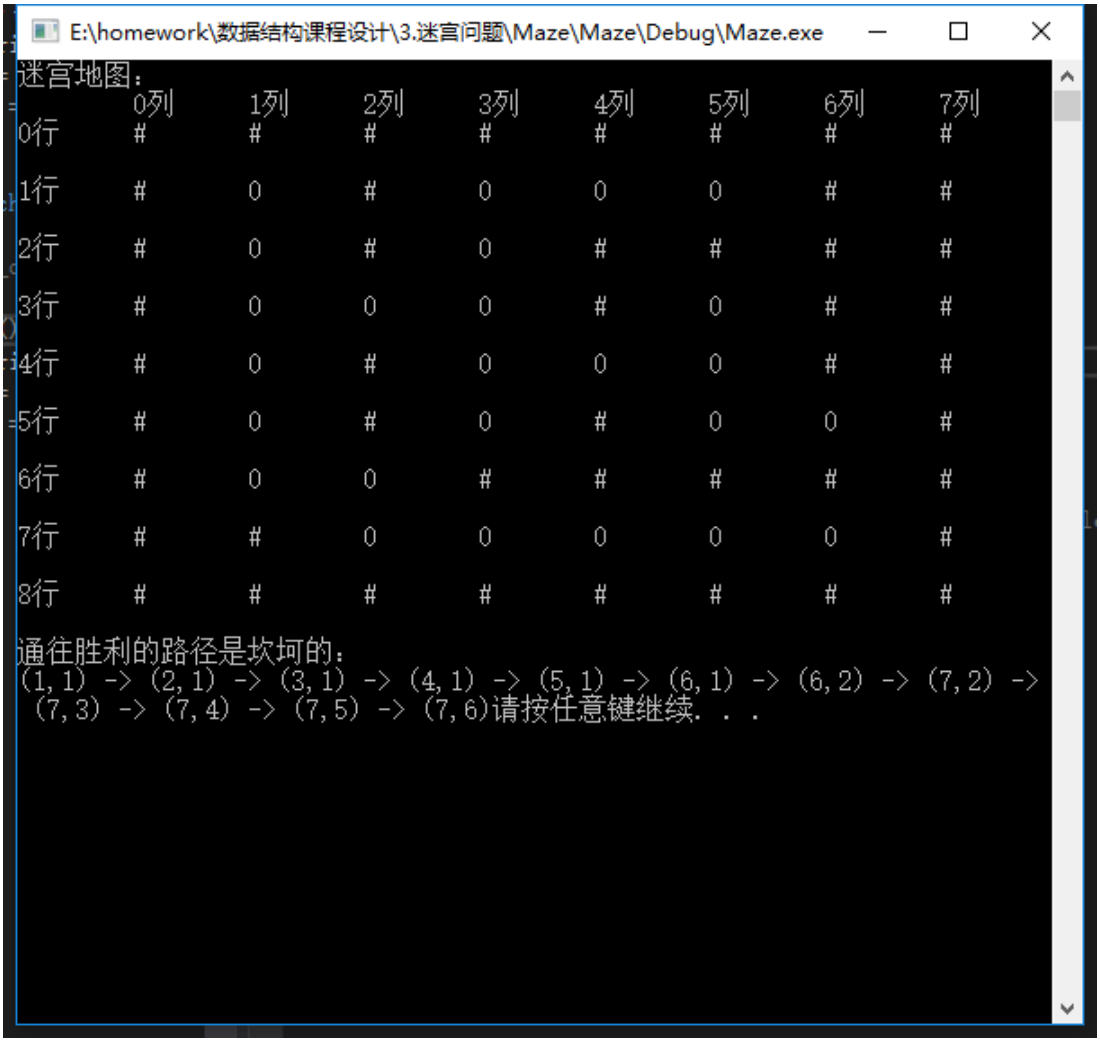
```
8 7
########
#0#000##
#0#0####
#000#0##
#0#000##
#0#0#00#
#00#####
##00000#
########
```

运行测试



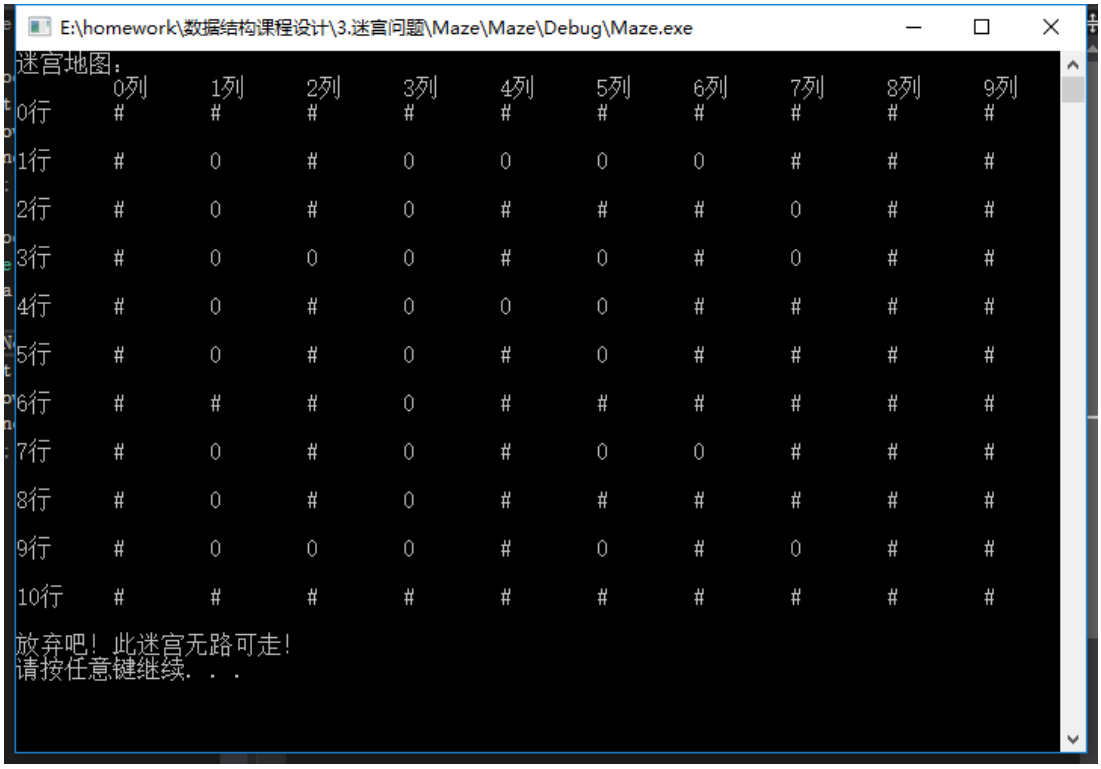
运行测试

```
13 13
##############
#0#000########
#0#0#0########
#000#0########
#0#0#0########
#0#0#0########
#####00#######
#000##0#######
#0#0##0#######
#0#0##0#######
#0#0000##000##
#0#######0#0##
#000000000#00#
##############
```



运行测试

```
10 9
##########
#0#0000###
#0#0###0##
#000#0#0##
#0#000####
#0#0#0####
###0######
#0#0#00###
#0#0######
#000#0#0##
##########
```



不足

代码不够精简

由于是第一次使用正交双向循环链表，对其操作还是不够熟悉，因此有很多冗余代码，但是又不好删，只得留下保证程序基本运行。