数据结构课程设计项目说明文档

——勇闯迷宫游戏

作 者 姓 名： 马威

学 号： 2151294

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

****

目 录

[1 项目分析 1](#_Toc495668153)

[1.1 项目背景 1](#_Toc495668154)

[1.2 项目要求 1](#_Toc495668154)

[1.3 项目需求分析 2](#_Toc495668155)

[2 项目设计 3](#_Toc495668156)

[2.1 数据结构设计 3](#_Toc495668157)

[2.2 类设计 3](#_Toc495668158)

2.2.1 向量类（Vector） 3

2.2.2 结点类（QueueNode） 5

2.2.3 链式队列类（LinkedQueue） 5

2.2.4 结点类（StackNode） 6

2.2.5 链式栈类（LinkedStack） 7

2.3 算法设计 8

2.3.1 算法思路 8

2.3.2 性能评估 9

2.3.3 BFS 10

·流程图表示 10

·代码实现 10

2.3.4 DFS 12

·流程图表示 12

·代码实现 12

2.3.5 A\* 14

·流程图表示 14

·代码实现 14

[3 项目测试 1](#_Toc495668161)6

[3.1 打印迷宫测试 1](#_Toc495668162)6

[3.2 BFS查找测试 1](#_Toc495668166)6

[3.3 DFS查找测试 1](#_Toc495668170)7

[3.4 A\*查找测试 1](#_Toc495668174)8

[3.5 路径不存在的情况 1](#_Toc495668178)9

**1.项目分析**

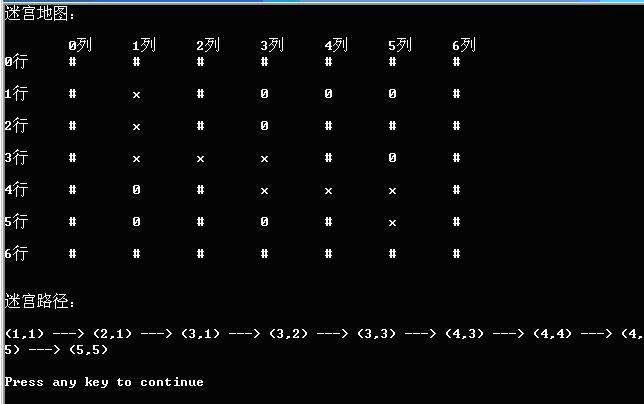
1.1 项目背景

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。

1.2 项目要求

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。

项目示例：



1.3 项目需求分析

对于寻找迷宫路径的程序，需考虑以下需求：

**·正确性**

程序应当能够按照要求对迷宫进行寻路等功能进行实现，没有错误

**·高效性**

迷宫数据较为复杂，求解路径需要用特定方法。这就要求要在尽可能短的时间内处理大量的数据，完成路径的查找。

**·可视化**

程序要能将迷宫和查找到的路径以合适的字符显示出来，用户体验更直观。

**2.项目设计**

2.1 数据结构设计

由项目分析可以得出，本项目需要用合适的数据结构存放地图信息。由于涉及行列信息，同时程序中要对地图中每一个位置进行频繁访问，故用二维数组存储较为合适，可以使用向量类数组进行优化。

寻路算法中，BFS涉及层次遍历，需要实现一个队列；DFS有关试探和回溯，可以实现一个栈进行优化；A\*有从若干个结点中取出权值最小的操作，需要实现一个优先级队列，可以用最小堆进行优化。

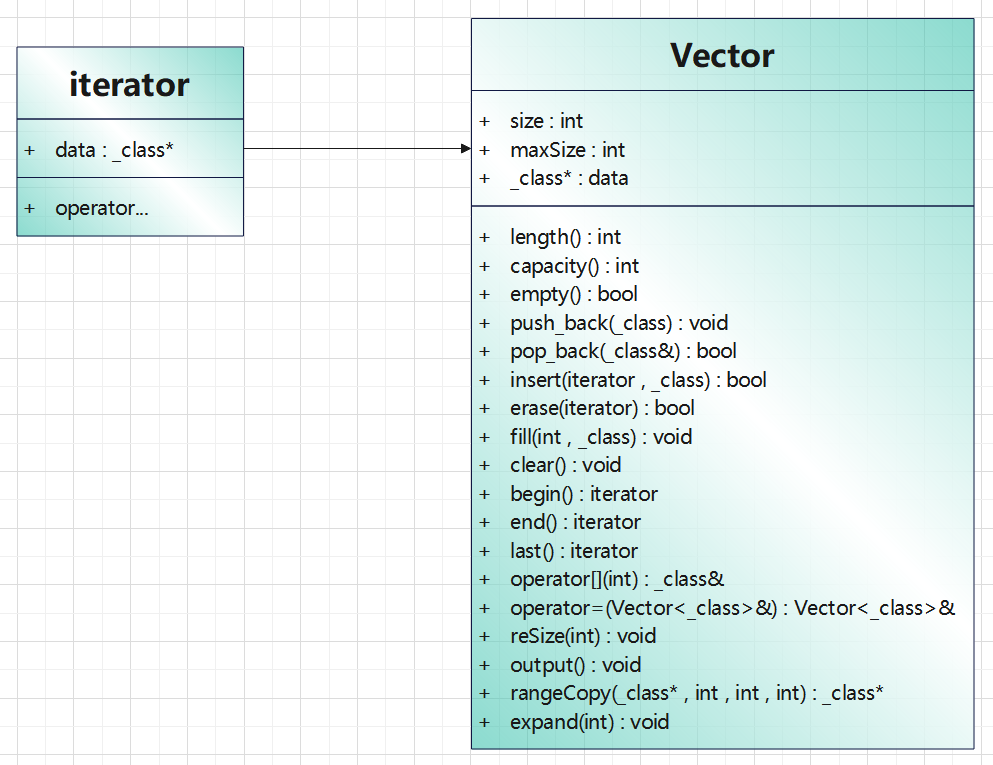
2.2 类设计

**2.2.1 向量类（Vector）**

向量类（Vector）是一种容器，其本质上通过动态数组来实现，使得元素按照线性顺序排列，同时可以通过下标快速访问，也提供了在末尾进行插入、删除元素等操作。

另一种常用的向量类操作是对其的遍历，在此添加了迭代器类（iterator）和一些运算符重载，以便对数组的遍历等操作。由于这个类名在标准库中已有实现，且此处的迭代器类为Vector类特有，故使用了嵌套类的形式，使得iterator只能在Vector类内访问。

该类和其内部的iterator类的UML图如下所示：



其中，主要函数如下：

//返回容器内部数组的长度

int length()const;

//返回容器最大容量

int capacity()const;

//判断容器是否为空

bool empty()const;

//向容器末尾压入一个元素

void push\_back(const \_class item);

//弹出容器末尾元素

bool pop\_back(\_class& x);

//在迭代器位置place上插入元素item，此位置及其后续元素后移一位

bool insert(const Vector<\_class>::iterator place, const \_class item);

//删除迭代器位置place上的元素

bool erase(const Vector<\_class>::iterator place);

//指定容器数组长度，并用元素item进行填充

void fill(const int sz, const \_class item);

//将容器置为空

void clear();

//返回容器的起始位置

inline Vector<\_class>::iterator begin();

inline const Vector<\_class>::iterator begin()const;

//返回容器末尾的后一个位置

inline Vector<\_class>::iterator end();

inline const Vector<\_class>::iterator end()const;

//返回容器的末尾位置

inline Vector<\_class>::iterator last();

inline const Vector<\_class>::iterator last()const;

//重载函数：下标访问

\_class& operator[](const int pos)const;

//重载函数：赋值

Vector<\_class>& operator=(const Vector<\_class>& v);

//调整容器的数组长度

void reSize(const int sz);

//输出容器内元素的值

void output()const;

//从data中的[low,high)区域复制一段长为sz的数据

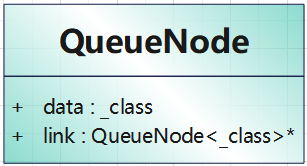
\_class\* rangeCopy(\_class\* data, const int low, const int high, int sz = 0);

//扩大数组空间

void expand(const int \_timesOfExpandingDefaultSize);

**2.2.2 结点类（QueueNode）**

链表结点存储了结点数据、后继节点位置，其UML图如下：

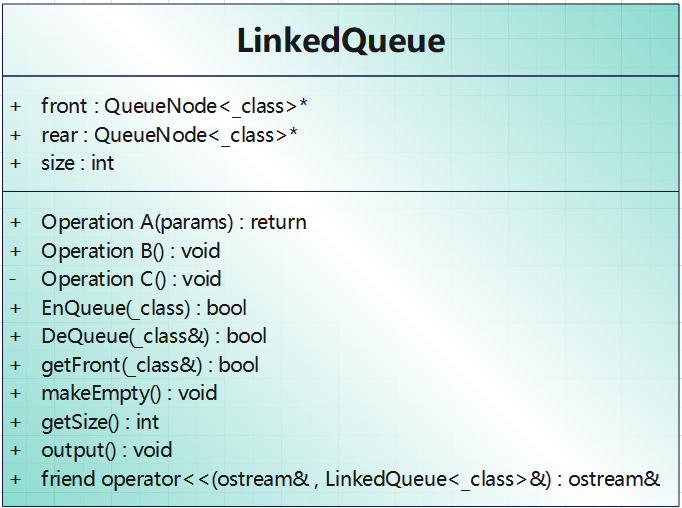


**2.2.3 链式队列类（LinkedQueue）**

链式队列类本质上仍是单链表，保存了链表的基本结构。由于我们在使用队列时仅仅利用其“先进先出”的特性，所以其提供的操作并不像一般链表操作那样齐全，而是仅仅有入队和出队。

由于对队列的插入、删除元素等操作的位置比较固定，为了节省空间，并未采用附加头结点。

其UML图如下：



其中，主要函数如下：

//入队

bool EnQueue(const \_class x);

//出队

bool DeQueue(\_class& x);

//取队列第一个元素

bool getFront(\_class& x)const;

//清空队列

void makeEmpty();

//判断队列是否为空

bool IsEmpty()const;

//取队列中元素个数

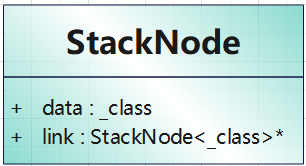
int getSize()const;

//输出队列中所有元素

void output()const;

**2.2.4 结点类（StackNode）**

链表结点存储了结点数据、后继节点位置，其UML图如下：

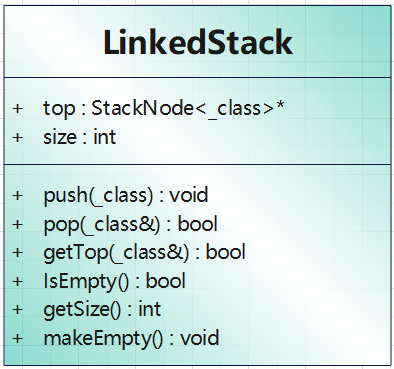


**2.2.5 链式栈类（LinkedStack）**

链式栈类本质上仍是单链表，保存了链表的基本结构。由于我们在使用栈时仅仅利用其“先进后出”的特性，所以其提供的操作并不像一般链表操作那样齐全，而是仅仅有入栈和出栈。

由于对栈的插入、删除元素等操作的位置比较固定，为了节省空间，并未采用附加头结点。

其UML图如下：



其中，主要函数如下：

//进栈

void push(const \_class x);

//出栈

bool pop(\_class& x);

//取栈顶元素

bool getTop(\_class& x)const;

//判栈空否

bool IsEmpty()const;

//返回栈元素个数

int getSize()const;

//清空栈的内容

void makeEmpty();

2.3 算法设计

**2.3.1 算法思路**

查找路径的方法有以下三种：

BFS（广度优先搜索）：

（1）准备过程：从起点开始考虑，显然起点上下左右四个方向上的位置到起点的最短路径长度为1，将这几个位置标为1，而标为1的四个方向上的位置（除了已经标记的位置外）到起点的最短路径长度为2。类似地，标n的位置相邻的未被标记位置需标记n+1。将所有从起点可以达到的位置都标记为止。可以用队列存储下一轮要处理的坐标，完成层次遍历。

（2）判断有无路径：若标记完成后终点仍未被标记，则不存在从起点到终点的路径，反之存在。

（3）得出路径：设终点标记为n，从终点出发，随机选择一个标记为n-1的位置行进，再随机选择一个标记为n-2的位置行进......循环至回到起点，行进轨迹就是路径的反向。

DFS（深度优先搜索）：

（1）准备过程：从起点开始，统计上下左右四个可行进的方向数k（既不是墙也没走过），并根据k的值进行以下操作：k=1直接向那个方向走；k>1则当前来到了路口，记录位置并随机选择一个方向前进；k=0说明无路可走，退回上一个路口。整个过程中有两个栈，一个记录当前的路径，一个记录当前经过所有路口的位置。

（2）判断有无路径：若k=0且存路口位置的栈为空，则说明已经退回了起点且能走过的路全部都走过了，则不存在从起点到终点的路径。反之，若某次行进到达了终点，则寻路直接结束。

（3）得出路径：到达终点后，记录当前路径的栈中就有路径的全部信息，不过是反向。

A\*：

（1）准备过程：首先建立两个表openlist和closelist，分别存放待处理和已处理的顶点。从起点开始，设当前位置为p，其上下左右四个位置为u、d、l、r，若可通行则计算出其权值，标记方向（例如u标↑）后放入openlist中（权值计算方法：设当前位置(x,y)，终点(m,n)，则权值为|x-m|+|y-n|），将p放入closelist中，再从openlist中取出权值最小的顶点赋给p。其中，openlist类似一个优先级队列，可以用最小堆优化。

（2）判断有无路径：若起点加入openlist，说明存在路径。反之，若openlist已空而起点仍然没有进入过openlist，则路径不存在。

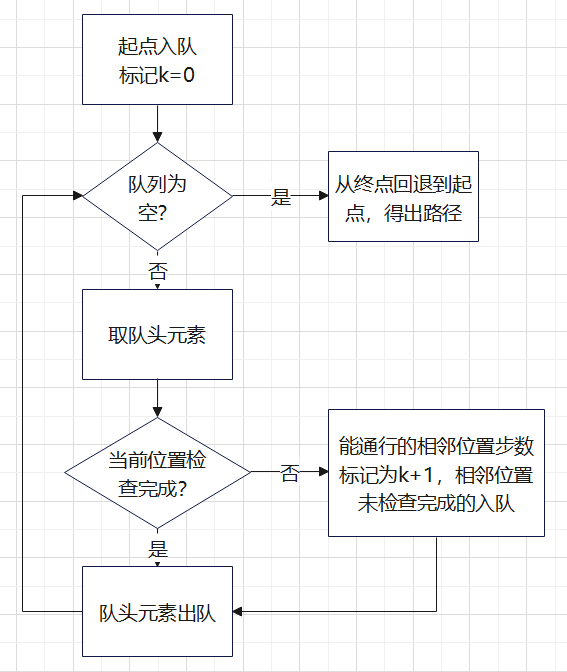
（3）得出路径：从终点开始往回找，可以从当前位置的方向得出该位置的直接前驱，循环直到回到起点为止，就得到了路径。

**2.3.2 性能评估**

设迷宫大小为m行n列，BFS必定遍历迷宫全部位置1次，DFS和A\*最坏情况下也需要遍历整个迷宫，故复杂度都为O(m\*n)。

**2.3.3 BFS**

·流程图表示

****

·代码实现（仅展示总过程）：

bool BFS\_findway(Vector<maps> matrix[], Vector<point>& way\_coord, const point start, const point dst)

{

    countway(matrix, start, dst);  /\*统计所有可移动到的位置到起点的最短步数\*/

    if (matrix[dst.line][dst.row].waynum == -1)  /\*若统计完成后终点步数仍为-1，则路径不存在，返回假\*/

        return false;

    int line = dst.line;

    int row = dst.row;

    for (int i = matrix[dst.line][dst.row].waynum; i > 0; i--) {  /\*每一次循环寻找到路径的其中一步，步数由终点位置元素的移动步数来决定\*/

        /\*检查上下左右四个元素，若该元素存在、移动步数比当前位置少1且不为零，

        则进行路径标记，并将当前位置记录到数组way\_coord[]中，同时将指针移动到下一步\*/

        /\*1、检查正上方元素\*/

        if (line && matrix[line - 1][row].waynum == matrix[line][row].waynum - 1 && matrix[line][row].waynum) {

            way\_coord.push\_back(point(line, row));  //压入当前位置坐标

            matrix[line][row].waymark = true;

            line--;

            continue;

        }

        /\*2、检查正下方元素\*/

        if (line < LINE + 1 && matrix[line + 1][row].waynum == matrix[line][row].waynum - 1 && matrix[line][row].waynum) {

            way\_coord.push\_back(point(line, row));  //压入当前位置坐标

            matrix[line][row].waymark = true;

            line++;

            continue;

        }

        /\*3、检查左侧元素\*/

        if (row && matrix[line][row - 1].waynum == matrix[line][row].waynum - 1 && matrix[line][row].waynum) {

            way\_coord.push\_back(point(line, row));  //压入当前位置坐标

            matrix[line][row].waymark = true;

            row--;

            continue;

        }

        /\*4、检查右侧元素\*/

        if (row < ROW + 1 && matrix[line][row + 1].waynum == matrix[line][row].waynum - 1 && matrix[line][row].waynum) {

            way\_coord.push\_back(point(line, row));  //压入当前位置坐标

            matrix[line][row].waymark = true;

            row++;

            continue;

        }

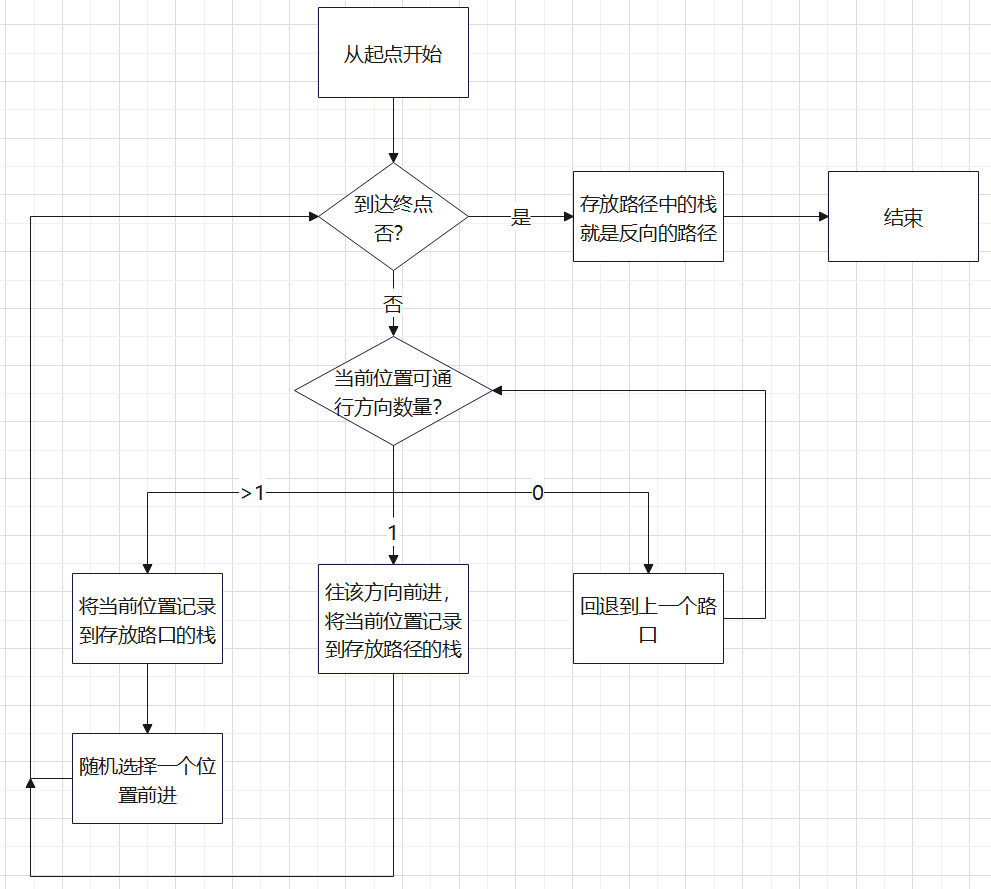
    }

    return true;

}

**2.3.4 DFS**

·流程图表示



·代码实现（仅展示总过程）：

bool DFS\_findway(Vector<maps> matrix[], Vector<point>& way\_coord, const point start, const point dst)

{

    Vector<point> next;  /\*下一步可行位置的数组\*/

    LinkedStack<point> intersections;  /\*存放各路口位置的栈\*/

    LinkedStack<point> marks;  /\*存放走过位置的栈\*/

    matrix[start.line][start.row].waymark = true;  /\*给起点做上标记并将其压入走过位置的栈\*/

    marks.push(start);

    point me(start.line, start.row);  /\*模拟一个正在行进的对象，初始位置是起点\*/

    do {

        const int count = direction\_count(matrix, next, me);  /\*统计可行进方向\*/

        const bool ret = count\_check(count, intersections, marks, me, start);  /\*对统计结果进行处理\*/

        if (ret)  /\*若结果正常，随机向前走一步\*/

            walk(matrix, next, marks, me);

        else  /\*若无路可走，停止寻路，返回假\*/

            return false;

    } while (me.line != dst.line || me.row != dst.row);  /\*走到终点时停止\*/

    while (!marks.IsEmpty()) {  /\*将栈中的元素转移到Vector中，并标记路径位置\*/

        point p;

        marks.pop(p);

        way\_coord.push\_back(p);

        matrix[p.line][p.row].waymark = true;

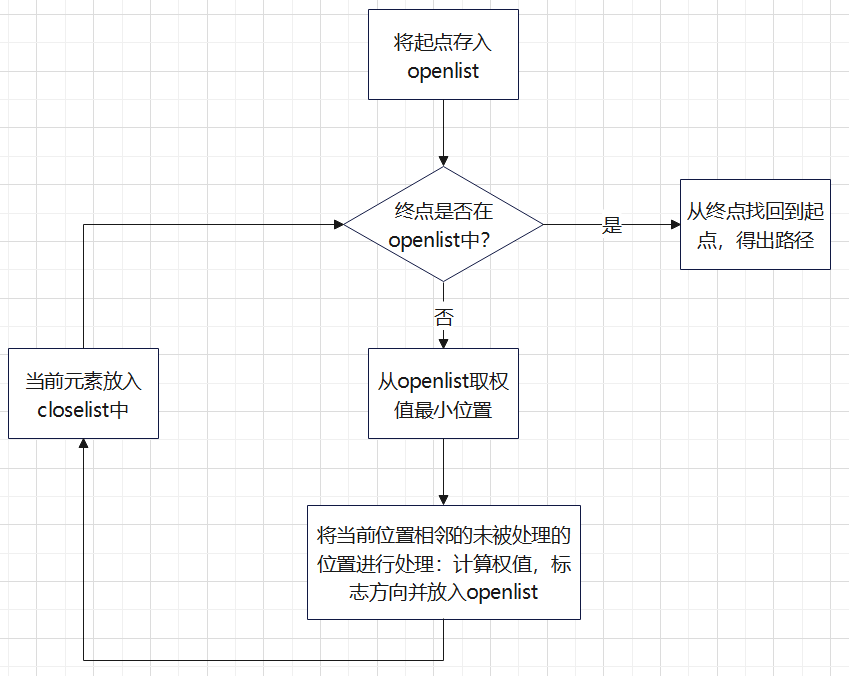
    }

    return true;

}

**2.3.5 A\***

·流程图表示：



·代码实现（仅展示总过程）：

bool Astar\_findway(const Vector<maps> matrix[], Vector<point>& way\_coord, const point start, const point dst)

{

    Vector<astar\_maps> Astar\_map[LINE];  /\*辅助地图，仅记录每个位置的方向和是否在closelist中\*/

    for (int i = 0; i < LINE; i++)

        Astar\_map[i].fill(ROW, astar\_maps());

    MinHeap<Astar\_maps> openlist(4 \* (LINE - 1) \* (ROW - 1));  /\*存放待处理位置的最小堆\*/

    bool if\_end = false;  /\*终点是否存入openlist\*/

    openlist.Insert(Astar\_maps(start.line, start.row));  /\*将起点存入openlist（权值设为0，保证从起点开始搜索）\*/

    while (!openlist.IsEmpty() && !if\_end) {

        Astar\_maps am(-1, -1);

        openlist.RemoveMin(am);  /\*取权值最小元素\*/

        if\_end = Astar\_check(matrix, Astar\_map, point(am.line, am.row), dst, openlist);  /\*将该元素周围位置存入openlist中\*/

        Astar\_map[am.line][am.row].if\_closed = true;  /\*将该元素存入closelist中\*/

        if (if\_end)  /\*终点在openlist中，说明路径已找到，返回真\*/

            break;

        else if (openlist.IsEmpty())  /\*若openlist已空而终点还没存入过openlist中，则路径不存在，返回假\*/

            return false;

    }

    point current(dst.line, dst.row);  /\*模拟对象，初始位置为终点\*/

    /\*根据每个位置上的方向，找出该位置的直接前驱，同时记录下路径位置，直到回到起点为止\*/

    while (current.line != start.line || current.row != start.row) {

        way\_coord.push\_back(current);

        matrix[current.line][current.row].waymark = true;

        switch (Astar\_map[current.line][current.row].direction) {

        case UP:

            current.line++;

            break;

        case DOWN:

            current.line--;

            break;

        case LEFT:

            current.row++;

            break;

        case RIGHT:

            current.row--;

            break;

        }

    }

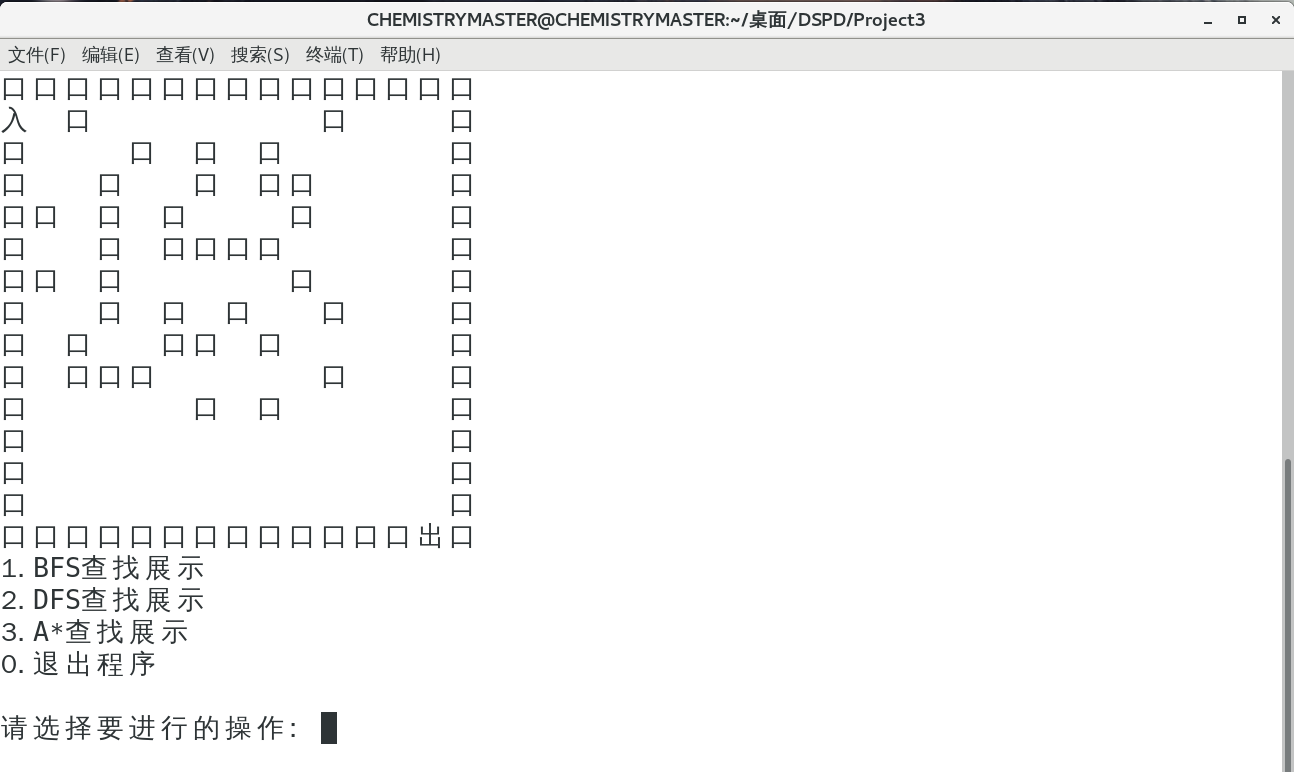
    return true;

}

**3.项目测试**

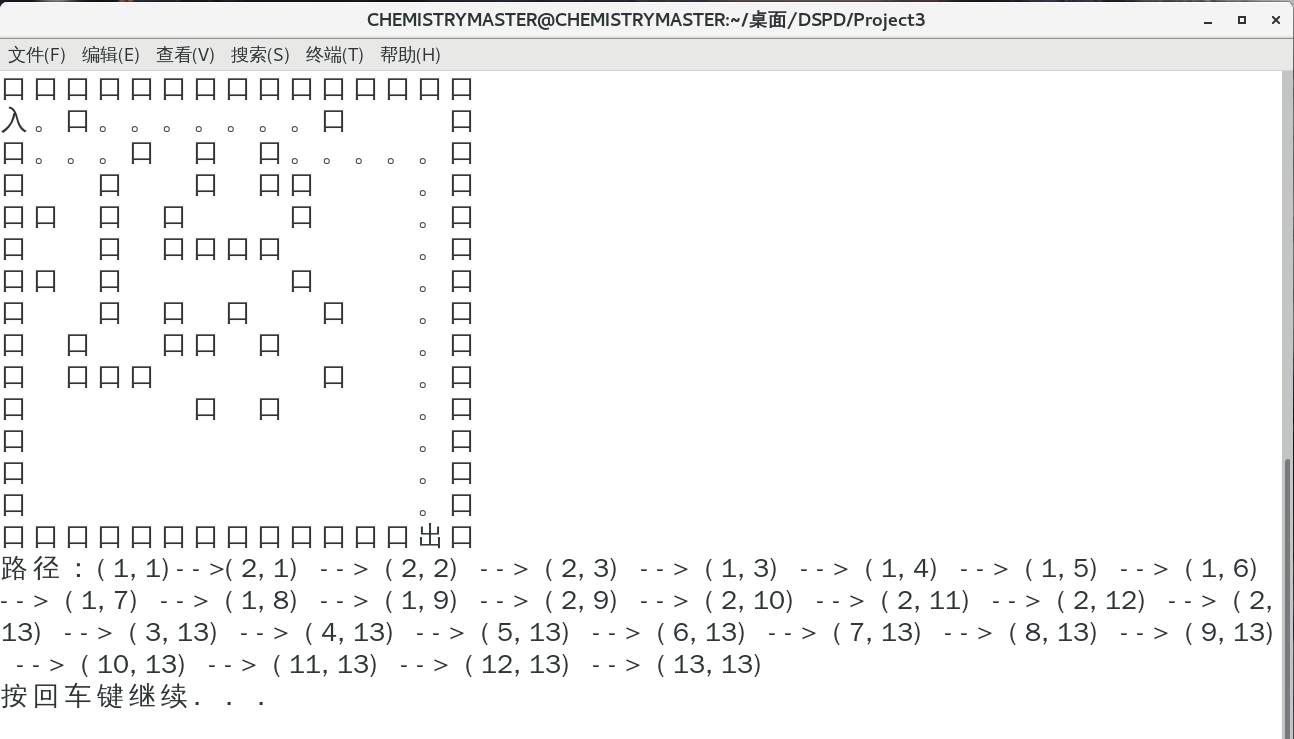
3.1 打印迷宫测试

测试结果：



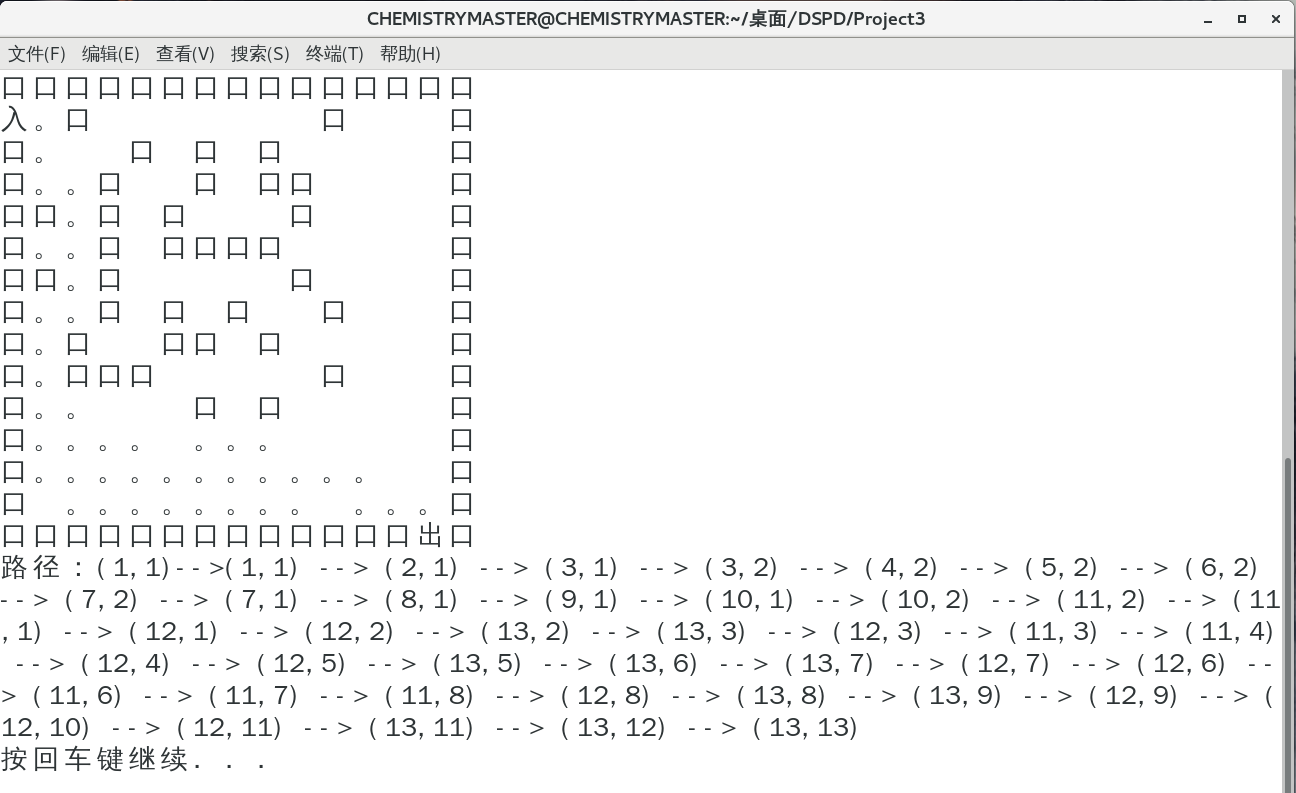
3.2 BFS查找测试

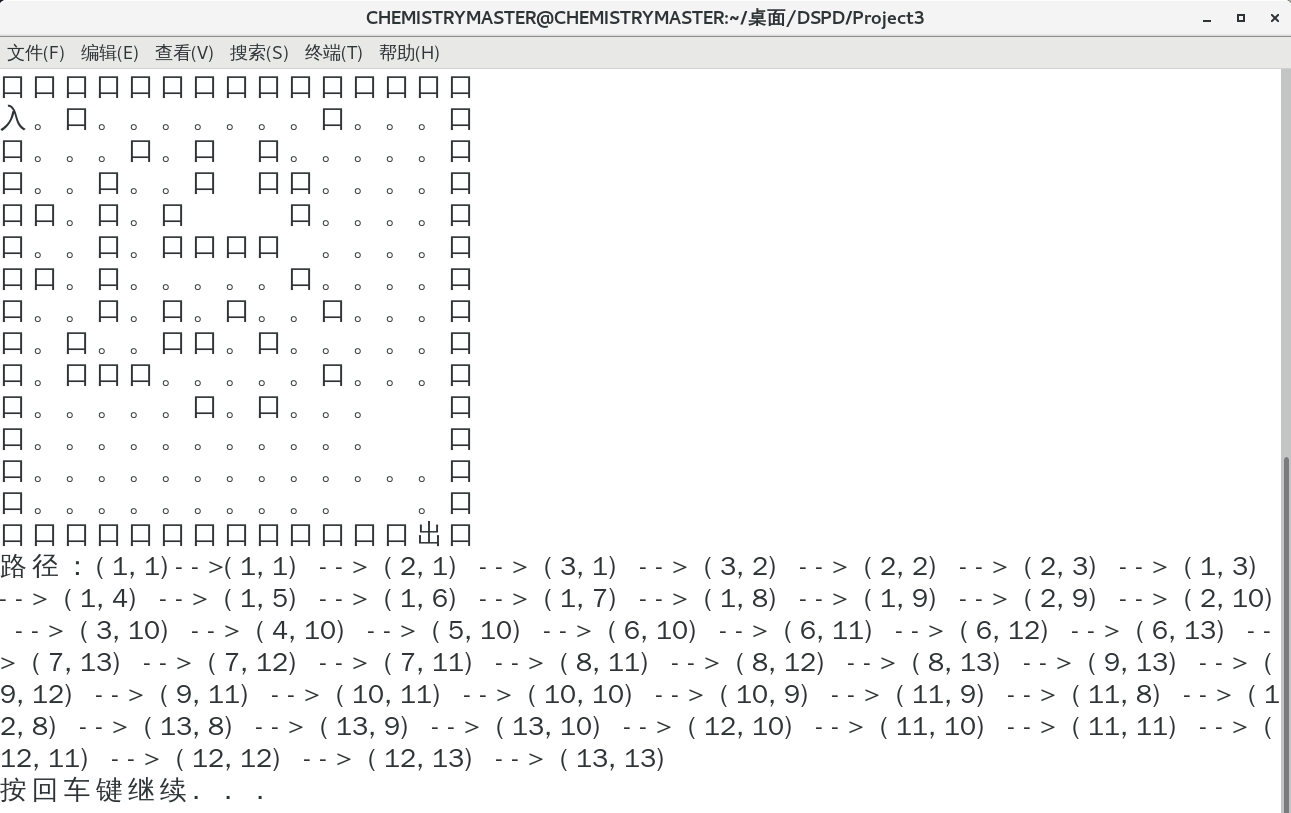
测试结果：

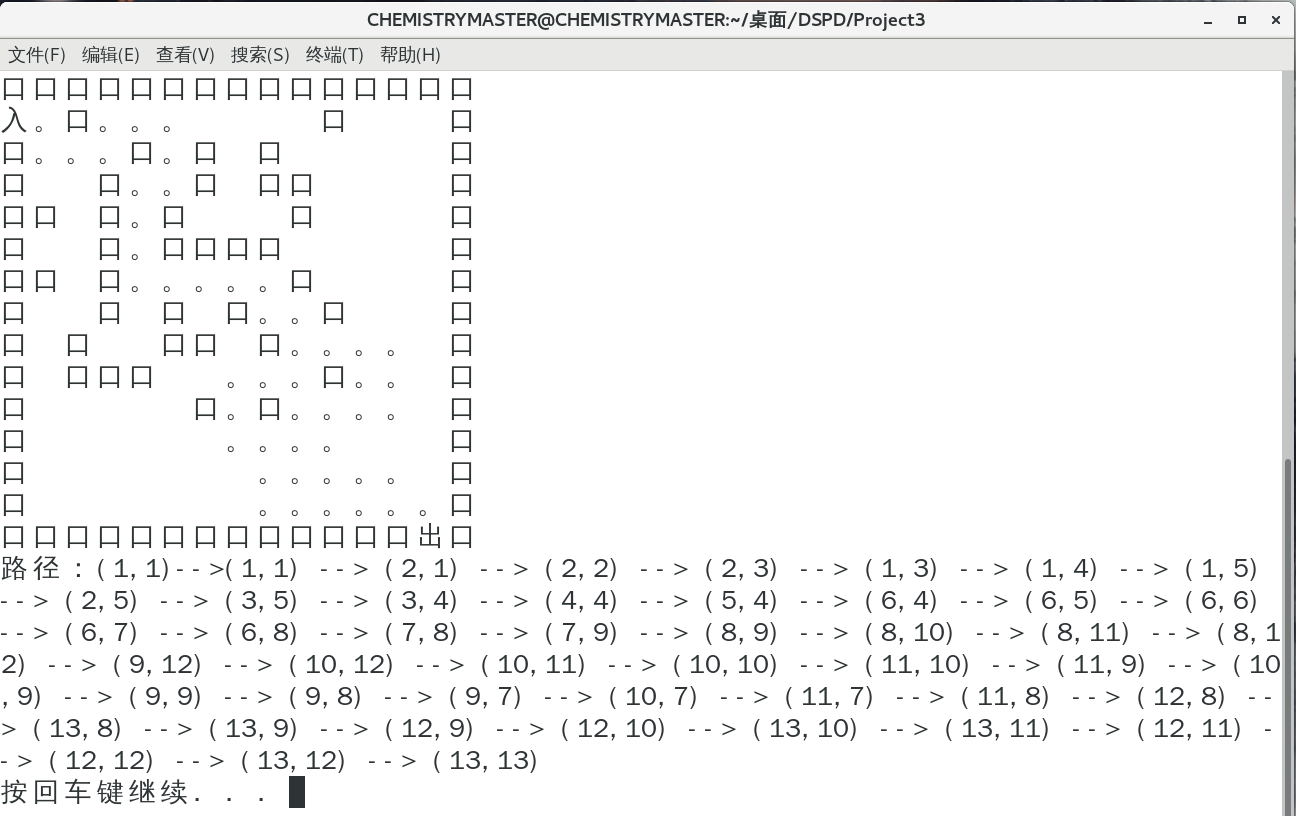


3.3 DFS查找测试

测试结果（多次测试说明DFS找到的路径不唯一，也不一定是最短的）：

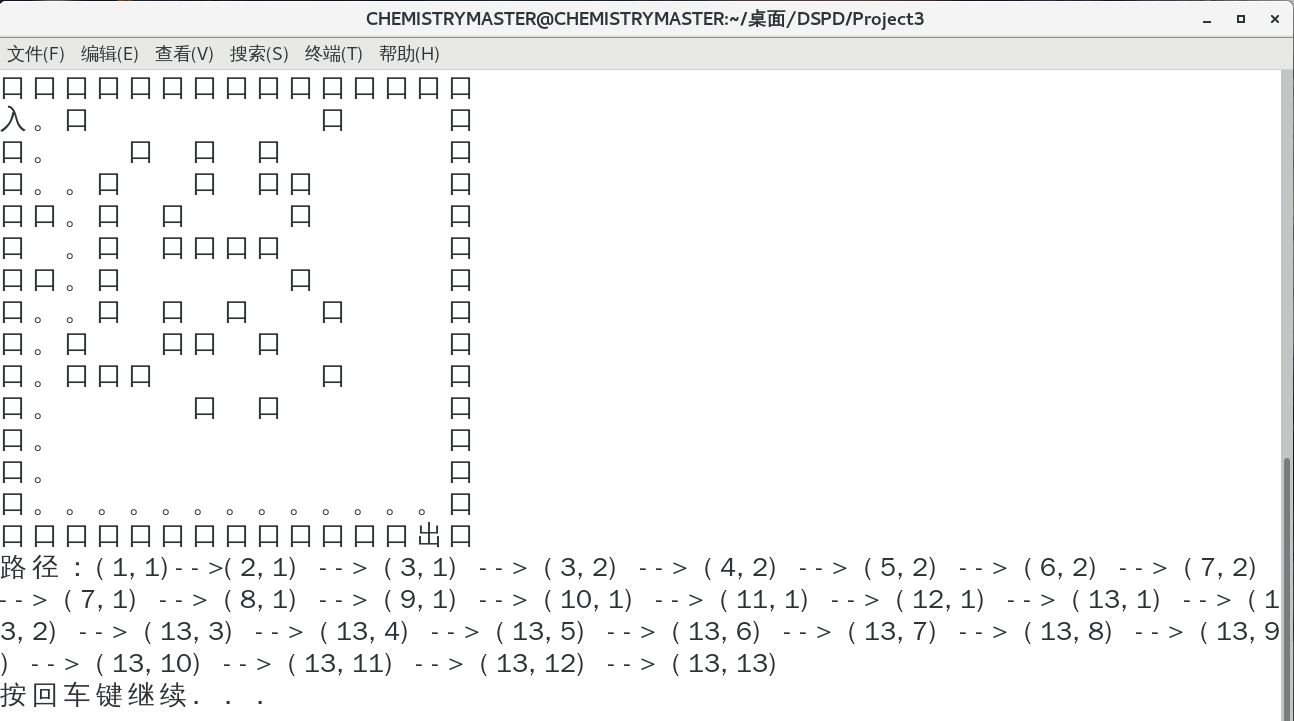






3.4 A\*查找测试

测试结果：



3.5 路径不存在的情况

测试结果（由于地图固定，故修改了源码）：

