项目说明文档

数据结构课程设计

——勇闯迷宫游戏

作 者 姓 名： 杨滕超

学 号： 2151298

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 项目概述 4](#_Toc495668153)

[1.1 项目背景](#_Toc495668154) 4

[1.2 项目功能](#_Toc495668155) 4

[1.3 项目分析](#_Toc495668154) 5

[2 项目设计](#_Toc495668156) 5

[2.1 总体设计](#_Toc495668157) 5

[2.2 类设计](#_Toc495668158) 5

[2.2.1 List类](#_Toc495668163) 5

[2.2.2 Stack类](#_Toc495668163) 6

[2.2.3 Queue类](#_Toc495668163) 7

[2.2.4 Vector类](#_Toc495668163) 7

[2.2.5 Vector游标类](#_Toc495668163) 9

[2.2.6 Heap类 1](#_Toc495668163)0

[2.2.7 迷宫类](#_Toc495668163) 10

[3 项目实现](#_Toc495668161) 10

[3.1 总体功能](#_Toc495668162) 10

[3.2 深度优先搜索递归方法 1](#_Toc495668166)1

[3.2.1 深度优先搜索递归方法流程图 1](#_Toc495668167)1

[3.2.2 深度优先搜索递归方法思路](#_Toc495668167) 11

[3.2.3 深度优先搜索递归方法主要代码](#_Toc495668168) 11

[3.3 深度优先搜索迭代方法 1](#_Toc495668170)3

[3.3.1 深度优先搜索迭代方法流程图 1](#_Toc495668171)3

[3.3.2 深度优先搜索迭代方法思路](#_Toc495668171) 13

[3.3.3 深度优先搜索迭代方法主要代码 1](#_Toc495668171)4

[3.4 广度优先搜索 1](#_Toc495668174)5

[3.4.1 广度优先搜索流程图 1](#_Toc495668175)5

[3.4.2 广度优先搜索思路 1](#_Toc495668175)5

[3.4.3 广度优先搜索主要代码 1](#_Toc495668176)6

[3.5 A\*搜索 1](#_Toc495668174)7

[3.5.1 A\*搜索流程图 1](#_Toc495668175)7

[3.5.2 A\*搜索思路 1](#_Toc495668175)7

[3.5.3 A\*搜索主要代码 1](#_Toc495668176)7

[3.6 寻路算法的比较总结与思考 1](#_Toc495668174)9

[3.7 Prim算法生成迷宫](#_Toc495668174) 20

[3.7.1 Prim算法生成迷宫流程图 2](#_Toc495668175)0

[3.7.2 Prim算法生成迷宫思路 2](#_Toc495668175)0

[3.7.3 Prim算法生成迷宫主要代码 2](#_Toc495668176)0

[3.8 Krusukal算法生成迷宫 2](#_Toc495668174)2

[3.8.1 Krusukal算法生成迷宫流程图 2](#_Toc495668175)2

[3.8.2 Krusukal算法生成迷宫思路 2](#_Toc495668175)2

[3.8.3 Krusukal算法生成迷宫主要代码 23](#_Toc495668176)

[3.9 分割法生成迷宫 2](#_Toc495668174)4

[3.9.1 分割法生成迷宫流程图 2](#_Toc495668175)4

[3.9.2 分割法生成迷宫思路 2](#_Toc495668175)4

[3.9.3 分割法生成迷宫主要代码 2](#_Toc495668176)5

[3.10 回溯递归法生成迷宫 2](#_Toc495668174)7

[3.10.1 回溯递归法生成迷宫流程图 2](#_Toc495668175)7

[3.10.2 回溯递归法生成迷宫思路 2](#_Toc495668175)7

[3.10.3 回溯递归法生成迷宫主要代码 2](#_Toc495668176)7

[3.11 四种生成迷宫算法特点 2](#_Toc495668174)8

[4 项目测试](#_Toc495668161) 30

4.1 [深度优先遍历(递归)](#_Toc495668174) 30

[4.2 深度优先遍历(迭代)](#_Toc495668174) 30

[4.3 广度优先遍历](#_Toc495668174) 31

[4.4 A\_star算法](#_Toc495668174) 31

[4.5 输入错误测试](#_Toc495668174) 32

1 项目概述

* 1. 项目背景

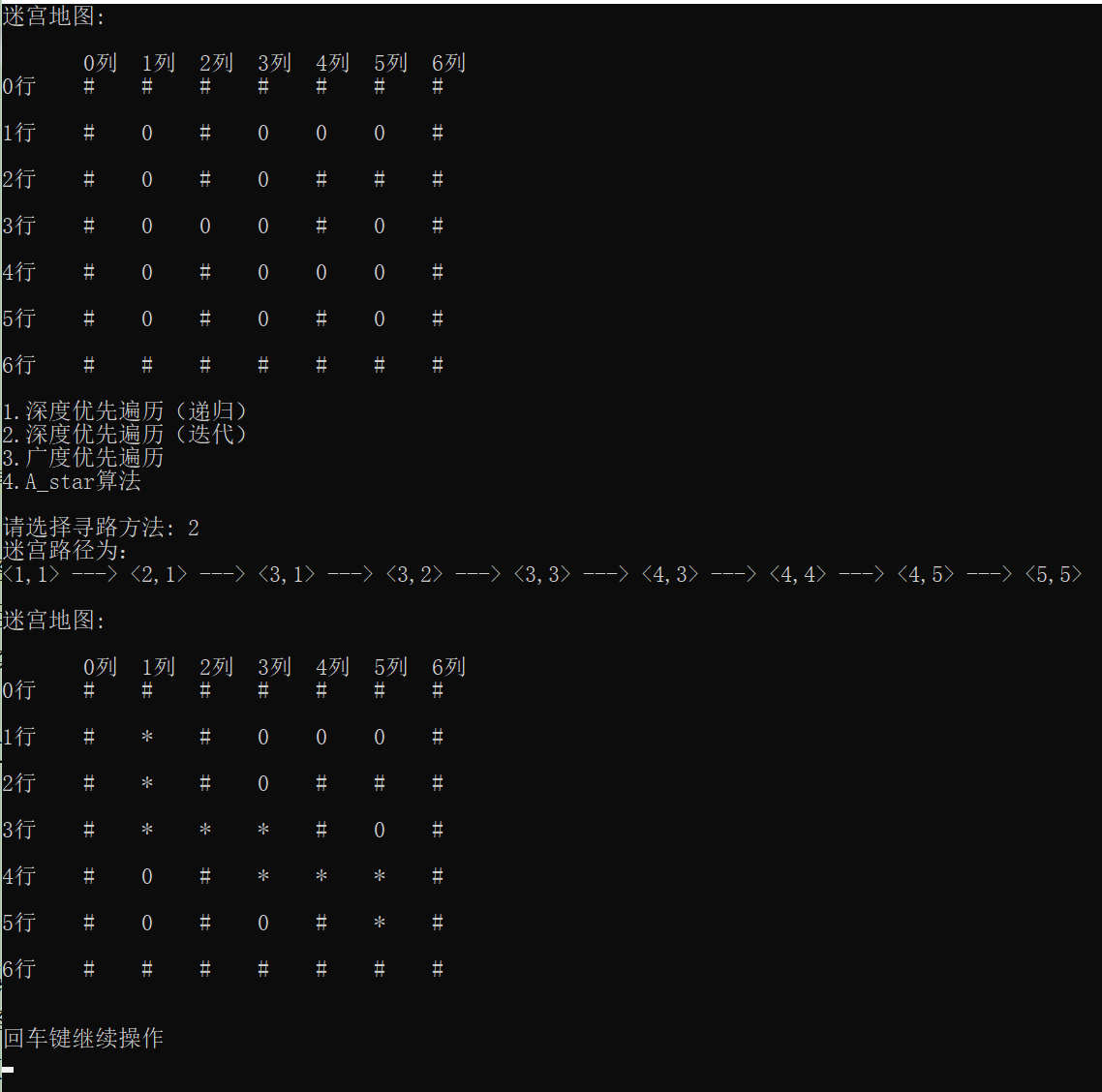
迷宫在游戏中是一中非常常见的结构，但往往承载着非常重要的作用。小到单机游戏，大到网游，几乎所有用于人物位置移动的操作都需要利用到“迷宫”。并且，与之相伴的就是对于路径的寻找，如何从起点移动到终点，如何选取最短路径往往是这个地方的关键。很容易就能想到朴素的暴力的解决方法，即为循环嵌套遍历，但这往往是不可取的，原因在于其时间复杂度之高令人、电脑都难以接受，于是催生出许许多多的寻路方法，它们拥有更好的时间复杂度。

本项目的重点之一即为比较各种迷宫寻路算法的好坏。重点在于深入学习和理解这些算法的思想，并为更深一步的学习打下良好的基础。

* 1. 项目功能

已经实现规定迷宫的大小与其中的障碍等等、迷宫的起点与终点，其中可走路径用0表示，障碍用#表示，标记生成的路径用\*表示，用户自行选择寻路的算法，最后输出寻路的结果，若是存在起点到达终点的路径，则输出路径，并用\*在地图中标记路径；反之输出提示，提示用户没有路径。

示例如下：



1.3 项目分析

1. 多种寻路算法。基于对寻路算法的学习与理解的目标，我们采用多种寻路算法的选择，以此加深对于迷宫寻路算法的工作原理。其中包括深度优先搜索递归方法、深度优先搜索迭代方法、广度优先搜索、A\*算法。其中前三者属于盲目搜索、最后一个输入“有目的”地搜索。后文将对上述算法进行比较和分析。
2. 迷宫生成算法。基于对迷宫寻路的思考，联想到迷宫如何生成，本项目总结了几种生成迷宫的算法。基于几种生成的迷宫的特点，我们将在后文进行比较与分析。
3. 健壮性。对于用户的输入错误与路径寻找不成功，均给出相关提示，而非程序报错，保证对用户的友好性。

2 项目设计

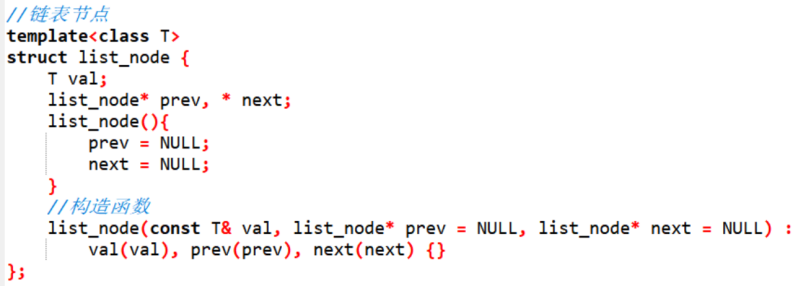
2.1 总体设计

采用二维字符型数组来储存迷宫的信息，按照老师要求，将迷宫初始参数固定。对于深度优先搜索递归方法，我们主要采用函数递归的方法进行寻路；对于深度优先搜索迭代方法，我们主要利用Stack结构将上述递归过程展开；对于广度优先搜索，我们主要采用队列存储将要访问的迷宫的位置的信息；对于A\*算法，我们主要采用优先队列存储将要访问的迷宫的位置的信息。本项目对于不同算法采用了多种不同的数据结构，考验了我们对向量、链表、栈、队列、堆和优先队列的理解与运用。

2.2 类设计

2.2.1 List类

·链表结点类设计：其中包括需要储存的数据，前驱节点和后继节点。如下图所示：



·链表类设计：数据成员包括，头结点，尾结点以及记录节点个数数据。

主要功能(与本题有关)包括：

·//返回链表大小inline int size() const;

·//判断链表是否为空inline bool empty() const;

·//清空链表void clear();

**·//最后添加元素void emplace\_back(const T& x)；**

**·//最后删除元素void pop\_back();**

**·//开头添加元素void emplace\_front(const T& x);**

**·//开头删除元素void pop\_front();**

·//插入void insert(const List<T>::list\_iterator& it, const T& x);

·//删除void remove(const List<T>::list\_iterator& it);

·//返回迭代器的查找typename List<T>::list\_iterator find(const T& x);

·//迭代器begin inline List<T>::list\_iterator begin();

·//迭代器end inline List<T>::list\_iterator end();

2.2.2 Stack类

栈是一种运算受限的线性表，它仅仅允许在栈顶进行元素的插入与删除操作，遵循先进后出的原则。在实现中，分为顺序栈和链式栈，考虑到顺序栈会出现栈满、申请空间未知导致的麻烦等缺点，我们的Stack类采用链式栈的方式构建。其中的有点包括申请空间灵活、不存在栈满的情况。

以链表为基础，我们构造链式栈。由于上述链表List类设置了头尾指针，并且是双链表，因此无论在链表的头或尾，增加或删除一个结点的时间复杂度均为O(1)，因此我们以链表尾作为栈顶。

该类主要成员函数如下：

//判断栈是否为空

inline bool empty() const {

return data.empty();

}

//获得栈中元素的个数

inline int size() const {

return data.Size;

}

//清空栈

void clear() {

data.clear();

}

//向栈顶添加元素

void emplace(const T& x) {

data.emplace\_back(x);

}

//删除栈顶元素

void pop() {

data.pop\_back();

}

//查询栈顶元素

T& top() {

return data.getTail()->val;

}

2.2.3 Queue类

队列和栈一样，同样是一种运算受限的线性表，它允许的操作是在表的前端(front)进行删除，表的后端(rear)进行插入，遵循先进先出的原则。其中的前端称为队头，后端称为队尾。队列的实现同样分为两种方式，顺序队列和链式队列，对于顺序队列，为了实现空间的循环利用，采用循环队列的方式，其中的要点在于队头指针和队尾指针往后移动增加1后对最大空间大小取模。而对于链式队列，其优点在于没有空间的限制，可以灵活地申请空间。

基于上述考虑，我们采用以链表为基础的队列。考虑到链表无论是在头或尾的插入删除结点的时间复杂度均为O(1)，因此我们将链表头作为队列头，链表为作为队列尾。

该类主要成员函数如下：

//判断队列是否为空

inline bool empty() const {

return data.empty();

}

//获得队列中元素个数

inline int size() const {

return data.Size;

}

//清空队列

void clear() {

data.clear();

}

//在队列为插入元素

void emplace(const T& x) {

data.emplace\_back(x);

}

//在队列头删除元素

void pop() {

data.pop\_front();

}

//查询队列头的元素值

T& front() {

return data.getHead()->next->val;

}

2.2.4 Vector类

向量作为存储同一种类型数据的一维数组，相当于是数组的扩展，在其基础上增加方便程序员工作的操作。其中的优点在于：对于查找某个位置的元素的时间复杂度为O(1)，但是它的缺点也十分明显：向量的删除与插入的时间代价是巨大的，时间复杂度基本是O(n)级别；但特殊的是，在最后插入元素时，时间复杂度为O(1)。

我们设计Vector类的同时，同样设计了它的迭代器。

主要成员函数如下：

·//构造函数

Vector();

Vector(int size);

Vector(const Vector<T>& V);

Vector(int size, const T& val);

·//清空Vector

void clear();

·//最后添加元素

void emplace\_back(const T& x);

·//最后删除元素

void pop\_back();

·//返回最后元素

const T back() const ;

T& back() ;

·//返回开头元素

const T front() const ;

T& front() ;

·//插入

void insert(const Vector<T>::vector\_iterator& it, const T& x);

·//删除

void remove(const Vector<T>::vector\_iterator& it);

·//迭代器begin

inline Vector<T>::vector\_iterator begin() ;

·//迭代器end

inline Vector<T>::vector\_iterator end() ;

·//返回迭代器的查找

typename Vector<T>::vector\_iterator find(const T& x);

·//重载[]

T& operator[](const int index) ;

·//重载=

Vector<T>& operator=(const Vector<T>& V);

·//重新设置大小

void resize(int size);

void resize(int size, const T& val);

2.2.5 Vector游标类

其中数据成员包含了指向Vector中数组元素的指针，并实现的重载\*、->、++、--、<、>等操作符。

主要函数如下：

·//重载\*

T& operator\*();

·//重载->

T\* operator->();

·//重载==

inline bool operator==(const vector\_iterator& it) const ;

·//重载!=

inline bool operator!=(const vector\_iterator& it) const;

·//重载不等号

inline bool operator<(const vector\_iterator& it) const;

inline bool operator>(const vector\_iterator& it) const ;

inline bool operator<=(const vector\_iterator& it) const;

inline bool operator>=(const vector\_iterator& it) const ;

·//重载自加

inline Vector<T>::vector\_iterator& operator++();

·//重载自减

inline Vector<T>::vector\_iterator& operator--();

·//后置自加

Vector<T>::vector\_iterator operator++(int);

·//后置自减

Vector<T>::vector\_iterator operator--(int);

·//与数字相加

friend Vector<T>::vector\_iterator operator+(const vector\_iterator& it,int num);

·//与数字相减

friend Vector<T>::vector\_iterator operator-(const vector\_iterator& it,int num);

·//自加

Vector<T>::vector\_iterator operator+=(int num);

·//自减

Vector<T>::vector\_iterator operator-=(int num);

·//两个相减

friend int operator-(const vector\_iterator& it1, const vector\_iterator& it2);

2.2.6 Heap类

堆是一种树形结构，是完全树，因此常常利用数组储存。分为大根堆和小根堆，以小根堆为例，任意一个根的孩子结点的值总是不小于其父节点，同时两个孩子也是一个小根堆。

我们利用向量作为堆的基础，通过迭代方式，实现堆的操作，主要成员函数如下：

·上浮调整void siftFix(Iter beg, Iter end, Compare cmp);

·下沉调整void sinkFix(Iter beg, Iter end, Compare cmp);

2.2.6 迷宫类

数据成员主要包括起点终点参数（int beg\_r , beg\_c , end\_r , end\_c ;）、四个方向（const int dx[4] = { -1, 1, 0, 0 }, dy[4] = { 0, 0, -1, 1 };）、地图（char\*\* map;）、选择的操（int op;）、是否使用了division（bool isDiv;）、bfs与a star用到的记录路径数组（Vector<Vector<Pair<int, int> > > path;）、 dfs用到的记录路径的“栈”（List<Pair<int, int>> stk;）、bfs与a star记录路径（Vector<Pair<int, int>> ans;）、dfs标记已遍历的（Vector<Vector<bool>> vis;）

主要成员函数

·集合寻路算法void solutions();

·设置数组大小，起点终点，生成算法选择void SlcSizePointAlg();

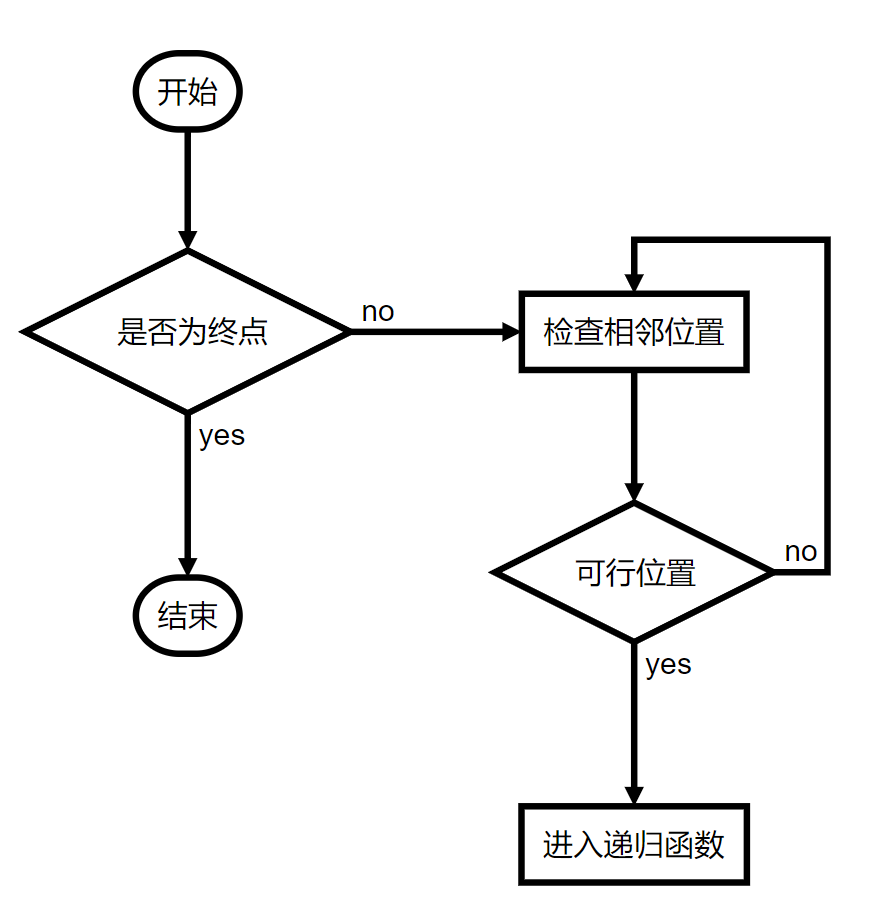
3 项目实现

3.1 总体功能

给予用户选择寻路算法，调用相应寻路函数，若找不到则输出提示，否则以有序对的方式输出路径。

3.2 深度优先搜索递归方法

3.2.1 深度优先搜索递归方法流程图



3.2.2 深度优先搜索递归方法思路

从起点开始，将当前的位置坐标利用Pair合成一个二元组压入栈中，并用二维数组vis[cur\_r][cur\_c]标记已经遍历过，表示经历过这个路径，之后遍历该位置的四个方向，并且保证不越界不是障碍没有遍历过，每遍历一次就进入一次递归函数，如果找到了终点则返回true，函数结束；否则继续循环接下来的方向，四个方向都找不到，栈pop出当前位置坐标，并返回false，表示通过这个点找不到终点，最终栈中的元素即为要求的路径。

3.2.3 深度优先搜索递归方法主要代码

//dfs递归算法

bool maze::\_dfs(int cur\_r, int cur\_c){

stk.emplace\_back(Pair<int, int>{cur\_r, cur\_c});

//终点

if (cur\_r == end\_r && cur\_c == end\_c)

return true;

vis[cur\_r][cur\_c] = true;//不是终点已经遍历过标记

for (int i = 0; i < 4; ++i){//四个方向

int new\_r = cur\_r + dx[i];

int new\_c = cur\_c + dy[i];

//不出界，不走重复的位置，不是障碍

if (new\_r < 0 || new\_c < 0 || new\_r >= R || new\_c >= C || vis[new\_r][new\_c] || map[new\_r][new\_c] == WALL)

continue;

if (\_dfs(new\_r, new\_c))

return true;

}

//都找不到

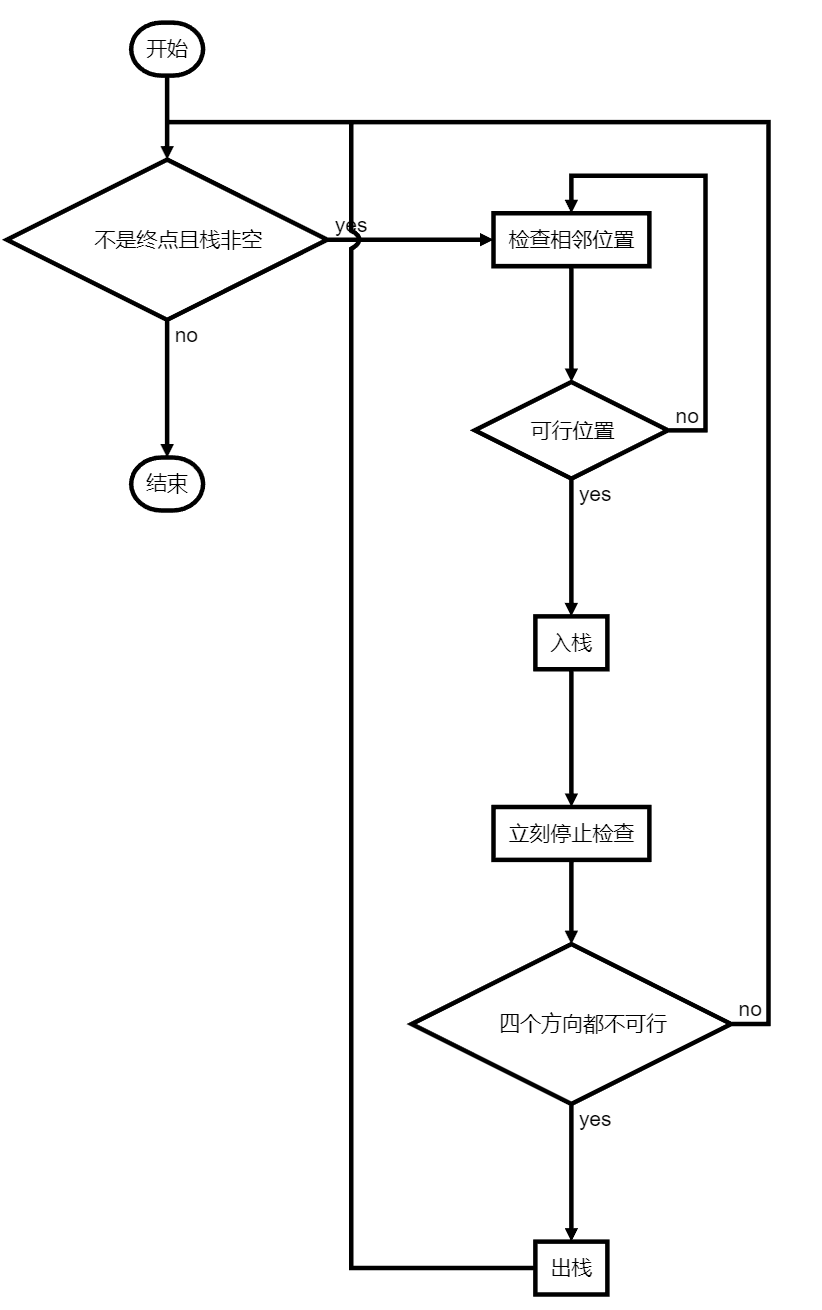
stk.pop\_back();

return false;//找不到终点

}

3.3 深度优先搜索迭代方法

3.2.1 深度优先搜索迭代方法流程图



3.2.2 深度优先搜索迭代方法思路

思路与上述方法基本相同，其区别在于利用了栈模拟了递归的操作，利用cnt记录失败的次数，若四个方向都失败了即说明通过该点找不到终点。弹出栈中。

3.2.3 深度优先搜索迭代方法主要代码

//如果当前不是终点并且栈不空才就循环

while (!(cur.first == end\_r && cur.second == end\_c) && !stk.empty()){

//标记遍历过vis[cur.first][cur.second] = true;

//标记失败次数int cnt = 0;

for (int i = 0; i < 4; ++i){//四个方向

int new\_r = cur.first + dx[i];

int new\_c = cur.second + dy[i];

//不出界，不走重复的位置，不是障碍

if (new\_r < 0 || new\_c < 0 || new\_r >= R || new\_c >= C || vis[new\_r][new\_c] || map[new\_r][new\_c] == WALL) {

++cnt;

continue;

}

//入栈

stk.emplace\_back(Pair<int, int>{new\_r, new\_c});

cur.first = new\_r, cur.second = new\_c;//当前节点移动

break;//只要找到一个位置能走就先走

}

if (cnt == 4){//四个方向都走不了，出栈

stk.pop\_back();//弹出找路失败的位置

cur = stk.getTail()->val;//回溯

}

}

if (stk.empty())//空了都没有路径

return false;

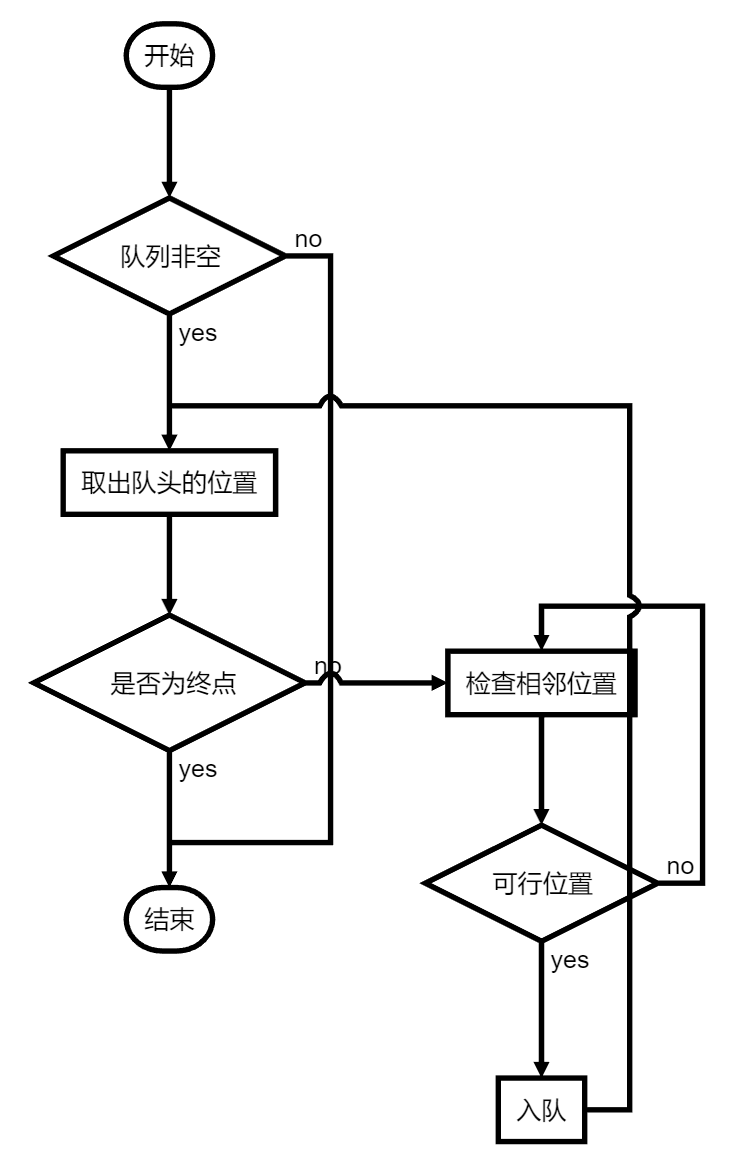
//最后的是否终点

auto last = stk.getTail()->val;

return (last.first == end\_r && last.second == end\_c);

3.4 广度优先搜索

3.4.1 广度优先搜索流程图



3.4.2 广度优先搜索思路

利用队列，从队列的队头获得当前位置，将遍历到的位置的四个方向的位置（保证不越界不是障碍没有遍历过）压入队尾，直到当前位置为终点，返回true，否则遍历完整个地图找不到终点返回false。

3.4.2 广度优先搜索主要代码

while (!q.empty()){

temp\_x = q.front().first;

temp\_y = q.front().second;

q.pop();

//找到终点

if (temp\_x == end\_r && temp\_y == end\_c)

return true;

//四个方向

for (int i = 0; i < 4; ++i){

int new\_x = temp\_x + dx[i];

int new\_y = temp\_y + dy[i];

//不出界，不是阻碍，没有到过

if (new\_x >= 0 && new\_y >= 0 && new\_x < R

&& new\_y < C && vis[new\_x][new\_y] == false

&& map[new\_x][new\_y] != WALL){

//反向记录路线

path[new\_x][new\_y] = Pair<int, int>{temp\_x, temp\_y};

vis[new\_x][new\_y] = true;//标记已经到过

q.emplace(Pair<int, int>{new\_x, new\_y});

}

}

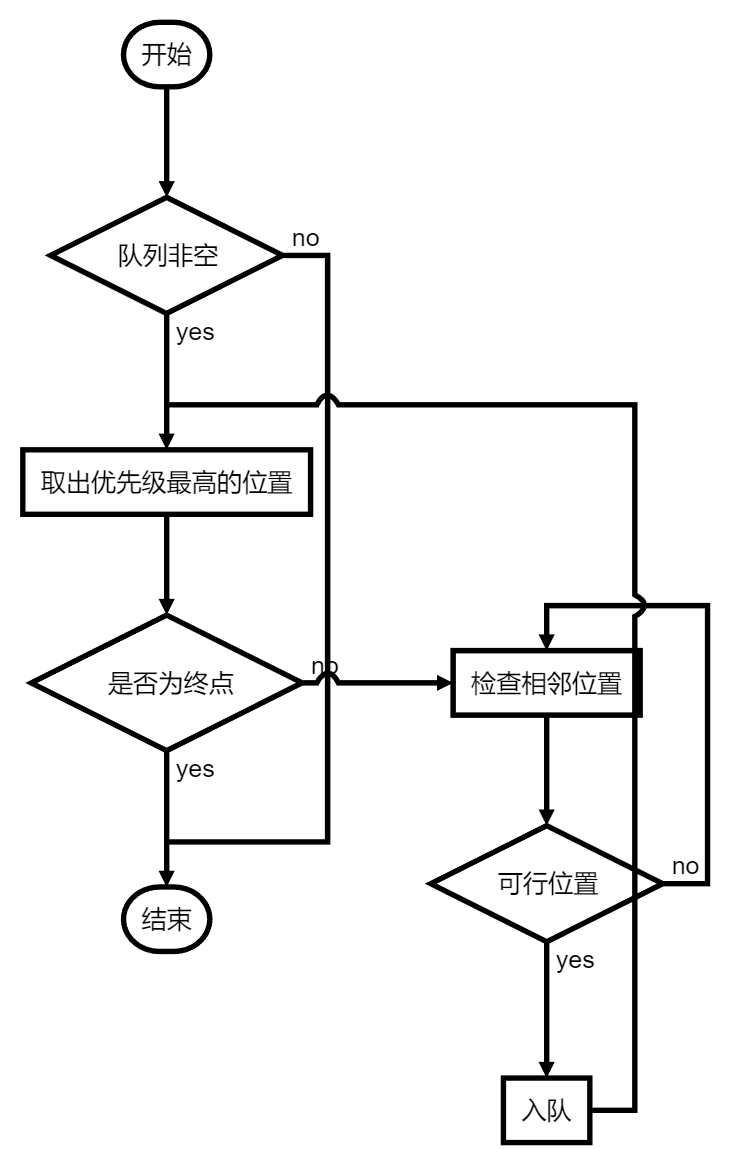
}

//找不到终点

return false;

3.5 A\*搜索

3.5.1 A\*搜索流程图



3.5.2 A\*搜索思路

在广度优先搜索的基础上，A\*利用优先队列，通过判断队列中位置坐标的优先级，取出优先级最高的。此处需要一个判断优先级的函数，这里设置了一个仿函数，仿函数中参数有起点终点坐标，其中比较曼哈顿距离，计算当前的代价以及预估的代价。

关于仿函数，将起点终点的坐标作为参数，每次调用A\*算法之前给它们赋值，因此将四个变量声明为静态成员。这里需要注意的是，静态成员不能于类内赋值，因此必须再类外对应的cpp文件中赋值。

3.5.3 A\*搜索主要代码

//优先队列用到的比较函数

struct CMP {

static int beg\_r, beg\_c, end\_r, end\_c;

bool operator()(Pair<int, int>& a, Pair<int, int>& b){

return abs(a.first - beg\_r) + abs(a.second - beg\_c) + abs(a.first - end\_r) + abs(a.second - end\_c) <

abs(b.first - beg\_r) + abs(b.second - beg\_c) + abs(b.first - end\_r) + abs(b.second - end\_c);}

};

//设置CMP参数

CMP::beg\_c = this->beg\_c;

CMP::beg\_r = this->beg\_r;

CMP::end\_r = this->end\_r;

CMP::end\_c = this->end\_c;

Priority\_queue < Pair<int, int>, Vector<Pair<int, int>>, CMP> q;//初始化一定要对应template的顺序

q.emplace(Pair<int, int>{beg\_r, beg\_c});

int temp\_x, temp\_y;

while (!q.empty())

{

temp\_x = q.top().first;

temp\_y = q.top().second;

q.pop();

//找到终点

if (temp\_x == end\_r && temp\_y == end\_c)

return true;

//四个方向

for (int i = 0; i < 4; ++i){

int new\_x = temp\_x + dx[i];

int new\_y = temp\_y + dy[i];

//不出界，不是阻碍，没有到过

if (new\_x >= 0 && new\_y >= 0 && new\_x < R

&& new\_y < C && vis[new\_x][new\_y] == false

&& map[new\_x][new\_y] != WALL)

{

//反向记录路线

path[new\_x][new\_y] = Pair<int, int>{ temp\_x, temp\_y };

vis[new\_x][new\_y] = true;//标记已经到过

q.emplace(Pair<int, int>{new\_x, new\_y});

}

}

}

//找不到终点

return false;

3.6 寻路算法的比较总结与思考

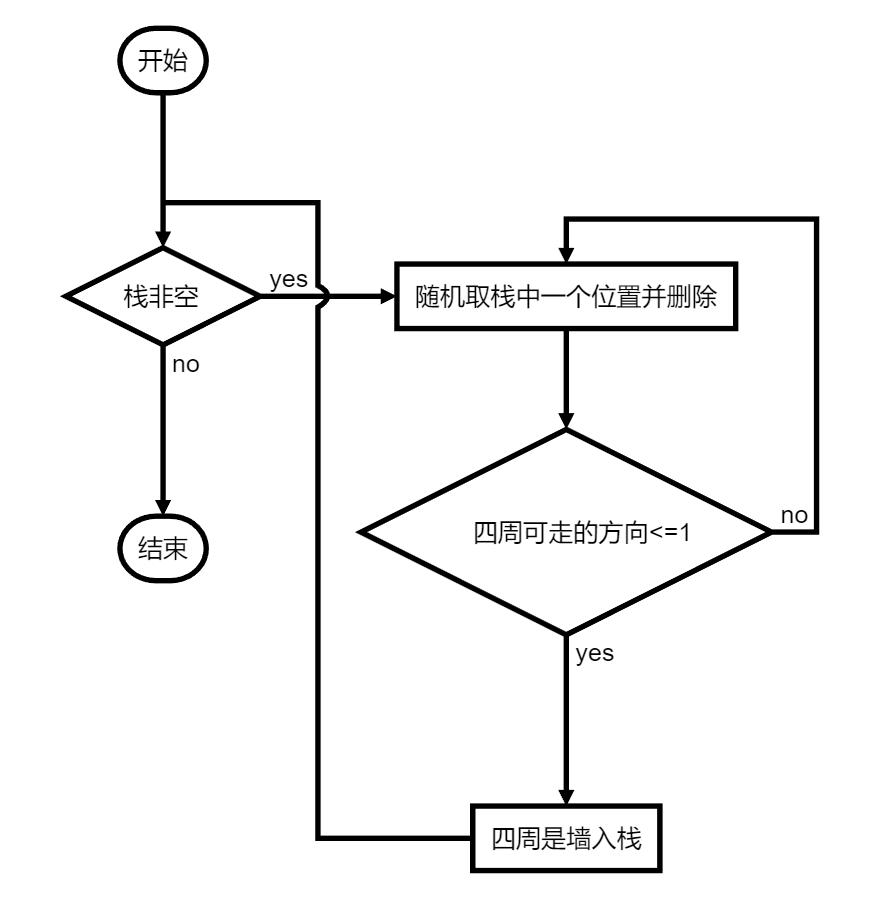
对于两种深度优先搜索算法，迭代方法可以算作对递归方法的改进，在实际项目编写过程中，往往不会用到递归的方法，而更多的是尽最大能力将递归展开成为非递归的算法，由此保证整个项目的性能。其原因在于递归在调用函数的过程中，需要保存临时变量，如果调用次数过多、层级过深，不但占用栈的资源，还可能导致栈资源超过了该线程的最大值，进而溢出，导致一系列问题。而大多数递归都是可以向非递归转化的，因此在本项目中，我们做出了尝试，利用Stack这样的数据结构进行递归的展开，模拟递归的过程，从而优化算法的性能。

对于深度优先搜索算法，利用队列实现了相当于一层一层地从起始点向外无差别扩张搜索。其中的缺点也十分明显，考虑到边界情况，若是起点与终点分别在迷宫地图的两个对角上，时间复杂度会达到O(n)，效率低下；与此同时，每个位置坐标之间是没有权值的，无法定义优先级，从迷宫思考到其他地图场景，若是遇到水域、森林、沙漠、沼泽等不同的地图信息，方向的选择上会出现问题。

基于深度优先搜索算法的缺点，我们引入A\*算法，为每个坐标位置赋予优先级，因此每次取出坐标都是当前情况下最希望走的方向，与前三种算法的区别在于这是一种“有目的”的搜索，基于“贪心”的原则。本项目使用的是哈密顿距离计算每个方向的代价，代价小则优先级高，对于没有障碍或障碍稀疏的地图会有非常好的效果，但是预估代价仅仅是计算从当位置到终点的哈密顿距离，其中的缺陷在于不能考虑到中间是否有障碍阻隔。日后我们将在这里进行更深的研究，寻找更好的代价估算函数完成算法的优化。

3.7 Prim算法生成迷宫

3.7.1 Prim算法生成迷宫流程图



3.7.2 Prim算法生成迷宫思路

利用一个向量存储等待访问的位置坐标，利用循环，从向量中随机选择一个位置作为当前位置，为了保证路径宽度为1，先检查四个方向已经开辟的个数，不多于1个的时候将当前位置设置为可走路径，最后将当前位置的四个方向（保证没有遍历过且是障碍）加入向量中，继续循环。

3.7.3 Prim算法生成迷宫主要代码

while (!stk.empty()){

//随机从栈中取一个路径位置

int index = rand() % stk.size();

Pair<int, int> temp = stk[index];

stk.remove(stk.begin() + index);

//记录四个方向有多少条能走

int road\_count = 0;

for (int i = 0; i < 4; ++i){

Pair<int, int> nextPlace(temp.first + dx[i], temp.second + dy[i]);

if (isRoad(nextPlace))

++road\_count;

}

//没有或者只有一条路，就去开辟新的方向，保证路径宽度为1

if (road\_count <= 1){

map[temp.first][temp.second] = '0';

//周围四个方向入栈

for (int i = 0; i < 4; ++i){

Pair<int, int> nextPlace(temp.first + dx[i], temp.second + dy[i]);

if (isWall(nextPlace))//是墙才入栈

stk.emplace\_back(nextPlace);

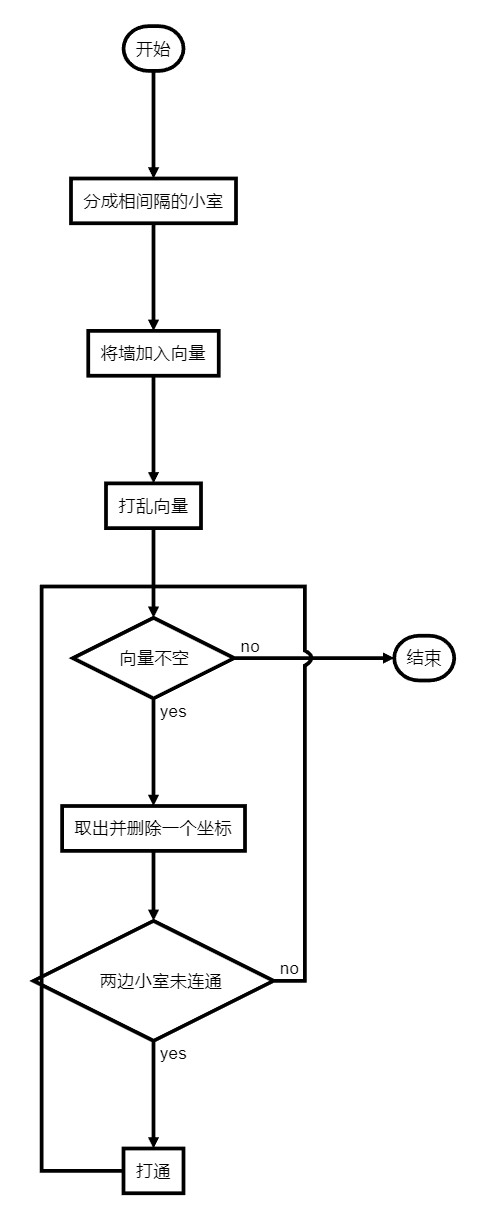
}

}

}

3.8 Kruskal算法生成迷宫

3.8.1 Kruskal算法生成迷宫流程图



3.8.2 Kruskal算法生成迷宫思路

和图论中的Kruskal算法相似，区别在于这里每条边的代价都是1，因此需要随机选取。首先将地图分成一个个小室，然后随机选取边，同时需要通过并查集判断两边的小室是否已经连通，以此防止形成回路，不连通才添加路径，直到边集为空。

3.8.3 Kruskal算法生成迷宫主要代码

//地图初始化，分成小室，并记录墙

for (int i = 0; i < R - RIsEven; i += 2)

for (int j = 0; j < C - CIsEven; j += 2){

map[i][j] = EMPTY;

//上下左右都是墙的墙是不记录的

if (j + 1 < C - CIsEven)//记录空格右边的

walls.emplace\_back(Pair<int, int>(i, j + 1));

if (i + 1 < R - RIsEven)

walls.emplace\_back(Pair<int, int>(i + 1, j));

}

//并查集

const int TOTAL = (R - RIsEven) \* (C - CIsEven);

Vector<int> fa(TOTAL);

for (int i = 0; i < TOTAL; ++i)

fa[i] = i;

//打乱墙序列

Random\_shuffle(walls.begin(), walls.end());

while (!walls.empty()){

Pair<int, int> wall = walls.back();

int x = wall.first, y = wall.second;

//上下只有在奇数行，左右在偶数行

Pair<int, int> c1 = x % 2 ? Pair<int, int>(x - 1, y) : Pair<int, int>(x, y - 1);

Pair<int, int> c2 = x % 2 ? Pair<int, int>(x + 1, y) : Pair<int, int>(x, y + 1);

int index1 = c1.first \* (R - RIsEven) + c1.second;

int index2 = c2.first \* (R - RIsEven) + c2.second;

int index3 = x \* (R - RIsEven) + y;

if (is\_connected(index1, index2, fa) == false){//还不连通可以连

union\_xy(index1, index2, fa);

union\_xy(index1, index3, fa);

map[x][y] = '0';

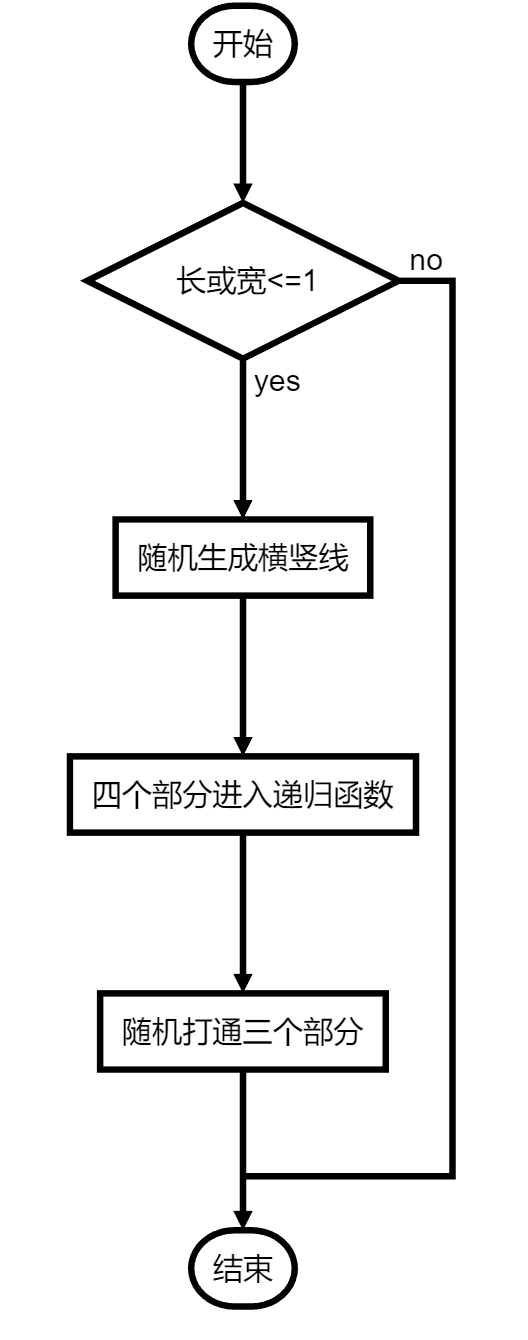
}

walls.pop\_back();

}

3.9 分割法生成迷宫

3.9.1 分割法生成迷宫流程图



3.9.1 分割法生成迷宫思路

随机生成水平、竖直两条线，将地图分成四个部分，通过递归函数调用分别对四个部分生成迷宫，最后将生成好的四个部分打通。打通时候需要注意，是否能真正打通两块区域，即保证需要打通的位置两边应该是路径。

3.9.1 分割法生成迷宫主要代码

//递归结束条件，不能再分割

if (x2 - x1 <= 1 || y2 - y1 <= 1)

return;

//随机取分割线位置

int x = x1 + 1 + rand() % (x2 - x1 - 1);

int y = y1 + 1 + rand() % (y2 - y1 - 1);

//画墙

for (int i = x1; i <= x2; ++i)

map[i][y] = WALL;

for (int i = y1; i <= y2; ++i)

map[x][i] = WALL;

//递归进入更小的分割好的方格

\_division(x1, y1, x - 1, y - 1);//左上

\_division(x + 1, y1, x2, y - 1);//左下

\_division(x1, y + 1, x - 1, y2);//右上

\_division(x + 1, y + 1, x2, y2);//右下

//随机取三面墙开点

bool walls[4] = { false };

walls[rand() % 4] = true;

for (int i = 0; i < 4; ++i){

if (walls[i])

continue;

int wall\_x = x;

int wall\_y = y;

switch (i){

//需要判断随机到的位置是否能打通相邻两块区域

//判断其上下左右少于等于两个墙

case 0:

do {

wall\_x = x1 + rand() % (x - x1);

} while (map[wall\_x - 1][wall\_y] + map[wall\_x + 1][wall\_y]

+ map[wall\_x][wall\_y - 1] + map[wall\_x][wall\_y + 1] == 3 \* WALL

|| map[wall\_x - 1][wall\_y] + map[wall\_x + 1][wall\_y]

+ map[wall\_x][wall\_y - 1] + map[wall\_x][wall\_y + 1] == 4 \* WALL);

break;

case 1:

do {

wall\_y = y + 1 + rand() % (y2 - y);

} while (map[wall\_x - 1][wall\_y] + map[wall\_x + 1][wall\_y]

+ map[wall\_x][wall\_y - 1] + map[wall\_x][wall\_y + 1] == 3 \* WALL

|| map[wall\_x - 1][wall\_y] + map[wall\_x + 1][wall\_y]

+ map[wall\_x][wall\_y - 1] + map[wall\_x][wall\_y + 1] == 4 \* WALL);

break;

case 2:

do {

wall\_x = x + 1 + rand() % (x2 - x);

} while (map[wall\_x - 1][wall\_y] + map[wall\_x + 1][wall\_y]

+ map[wall\_x][wall\_y - 1] + map[wall\_x][wall\_y + 1] == 3 \* WALL

|| map[wall\_x - 1][wall\_y] + map[wall\_x + 1][wall\_y]

+ map[wall\_x][wall\_y - 1] + map[wall\_x][wall\_y + 1] == 4 \* WALL);

break;

case 3:

do {

wall\_y = y1 + rand() % (y - y1);

} while (map[wall\_x - 1][wall\_y] + map[wall\_x + 1][wall\_y]

+ map[wall\_x][wall\_y - 1] + map[wall\_x][wall\_y + 1] == 3 \* WALL

|| map[wall\_x - 1][wall\_y] + map[wall\_x + 1][wall\_y]

+ map[wall\_x][wall\_y - 1] + map[wall\_x][wall\_y + 1] == 4 \* WALL);

break;

default:

break;

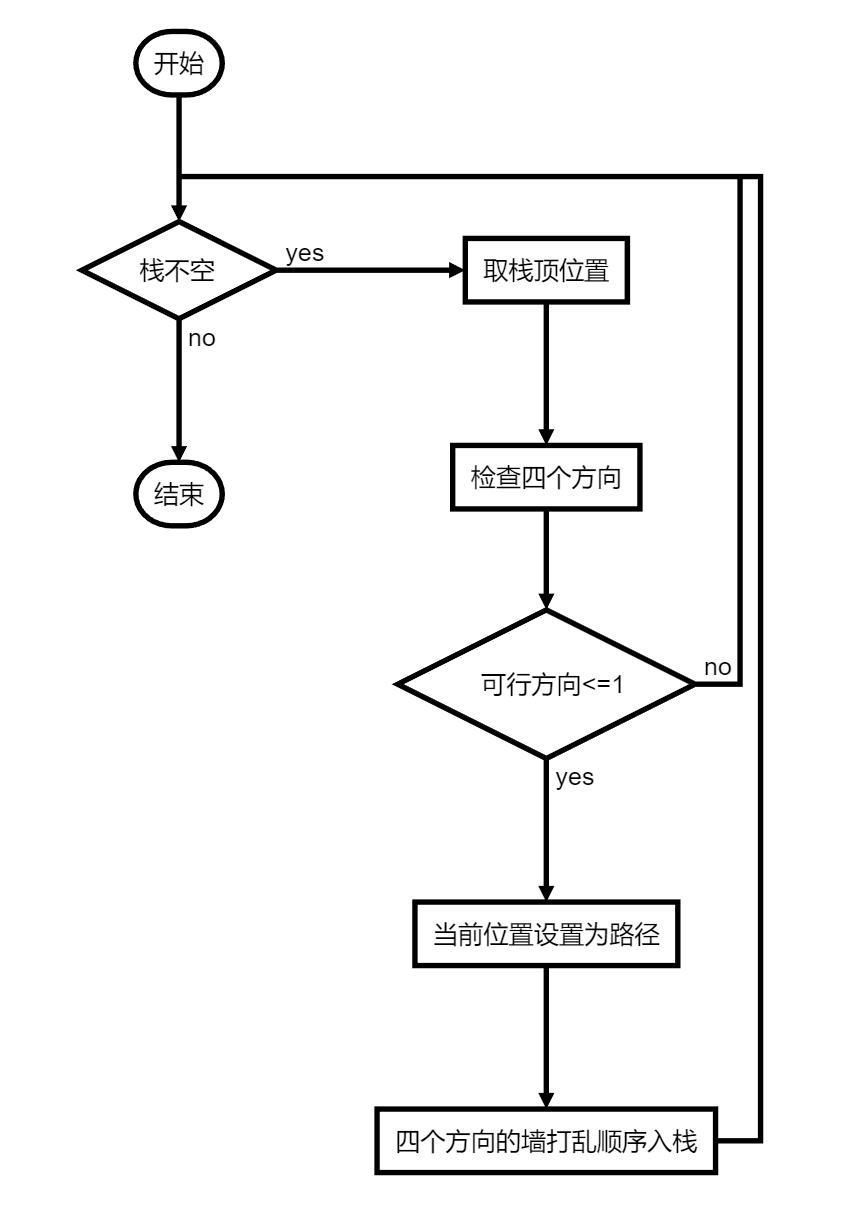
}

map[wall\_x][wall\_y] = EMPTY;

}

3.10 回溯递归法生成迷宫

3.10.1 回溯递归法生成迷宫流程图



3.10.2 回溯递归法生成迷宫思路

定义一个栈存储将要方位的位置坐标，进入循环，取栈顶元素为当前坐标，与Prim生成迷宫算法类似，记录当前位置的四个方向，不多于一个方向就可以将当前位置设置为路径，接着将四个方向（没有遍历过且是障碍）以随机顺序加入栈中。其中的方法是先记录入向量数组，使用Random\_shuffle函数进行打乱，再加入栈中。

3.10.3 回溯递归法生成迷宫主要代码

主要代码如下：

while (!stk.empty()){

Pair<int, int> temp = stk.getTail()->val;

stk.pop\_back();

//记录四个方向有多少条能走

int road\_count = 0;

for (int i = 0; i < 4; ++i){

Pair<int, int> nextPlace(temp.first + dx[i], temp.second + dy[i]);

if (isRoad(nextPlace))

++road\_count;

}

//没有或者只有一条路，就去开辟新的方向，保证路径宽度为1

if (road\_count <= 1){

map[temp.first][temp.second] = '0';

Vector<Pair<int, int>> nextPlaces;

//周围四个方向入栈

for (int i = 0; i < 4; ++i){

Pair<int, int> nextPlace(temp.first + dx[i], temp.second + dy[i]);

if (isWall(nextPlace))//是墙才入栈

nextPlaces.emplace\_back(nextPlace);

}

//入栈前打乱顺序，从栈顶取元素的时候随机

Random\_shuffle(nextPlaces.begin(), nextPlaces.end());

for (auto it = nextPlaces.begin(); it != nextPlaces.end(); ++it)

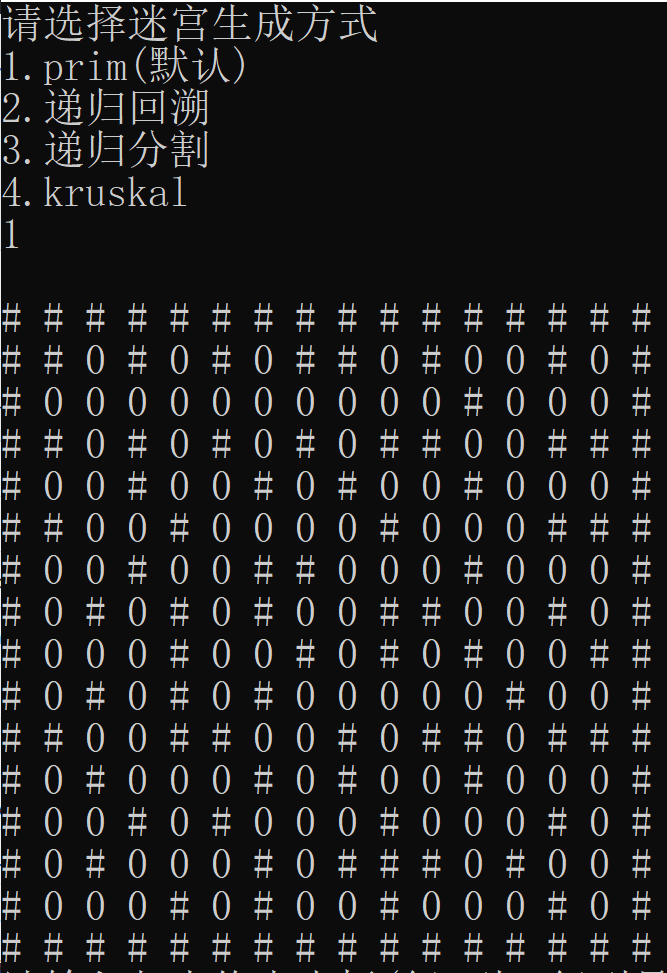
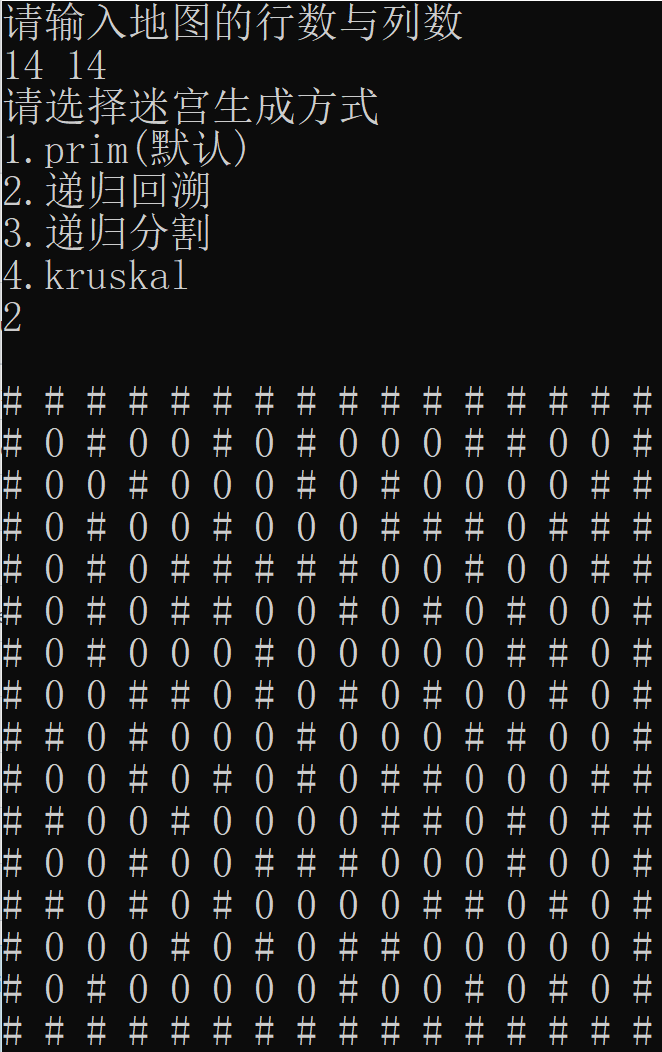
stk.emplace\_back(\*it);

}

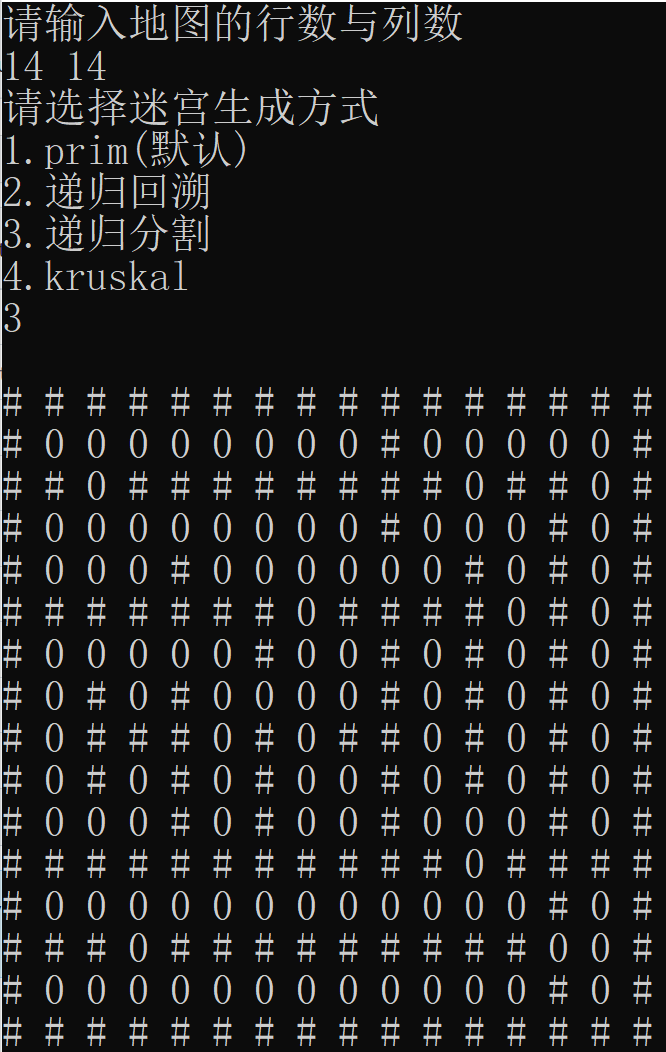
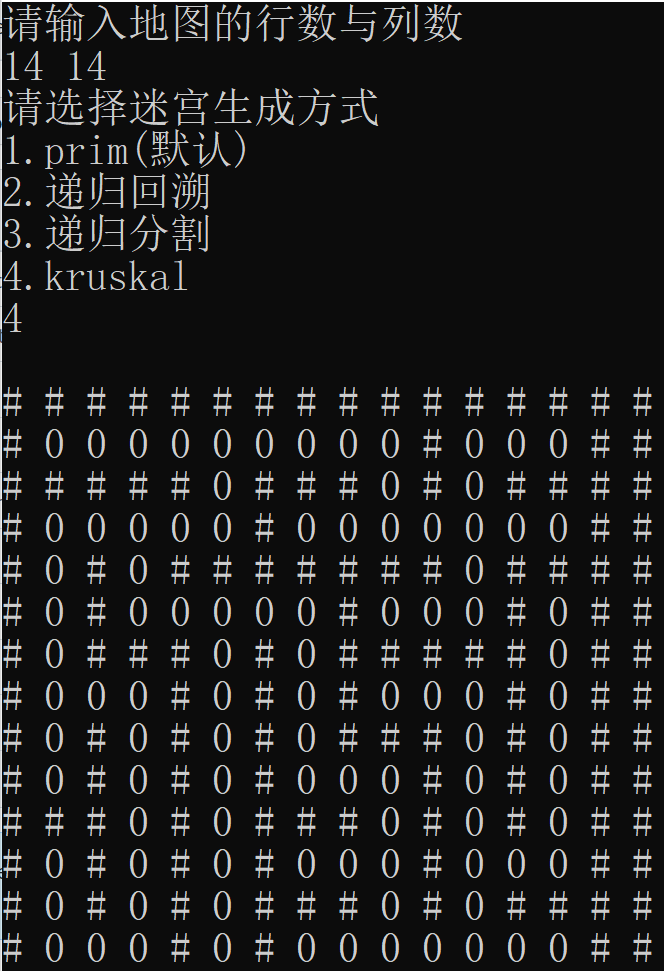
}

3.11 四种生成迷宫算法特点

Prim算法 递归回溯法

递归分割法 Kruskal

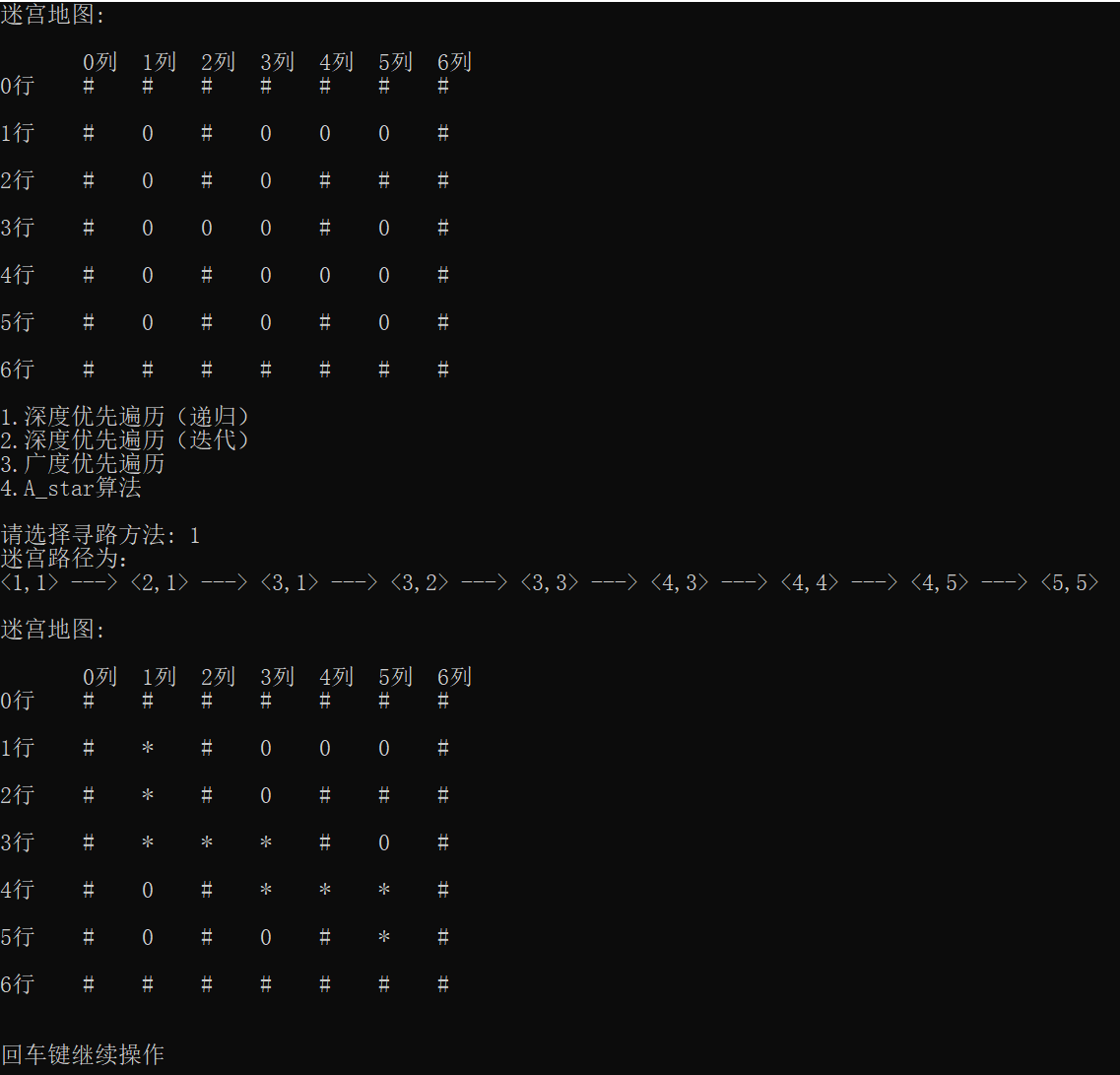
··

通过生成图片可以看出，Prim算法和递归回溯算法生成的迷宫较为复杂也比较贴近预期效果，具有较多的岔路和分支，因而具有更大的实用性，而后两种算法，生成的迷宫较为简单，横平竖直的路径较长，较为规则，较为容易一眼看出路径。

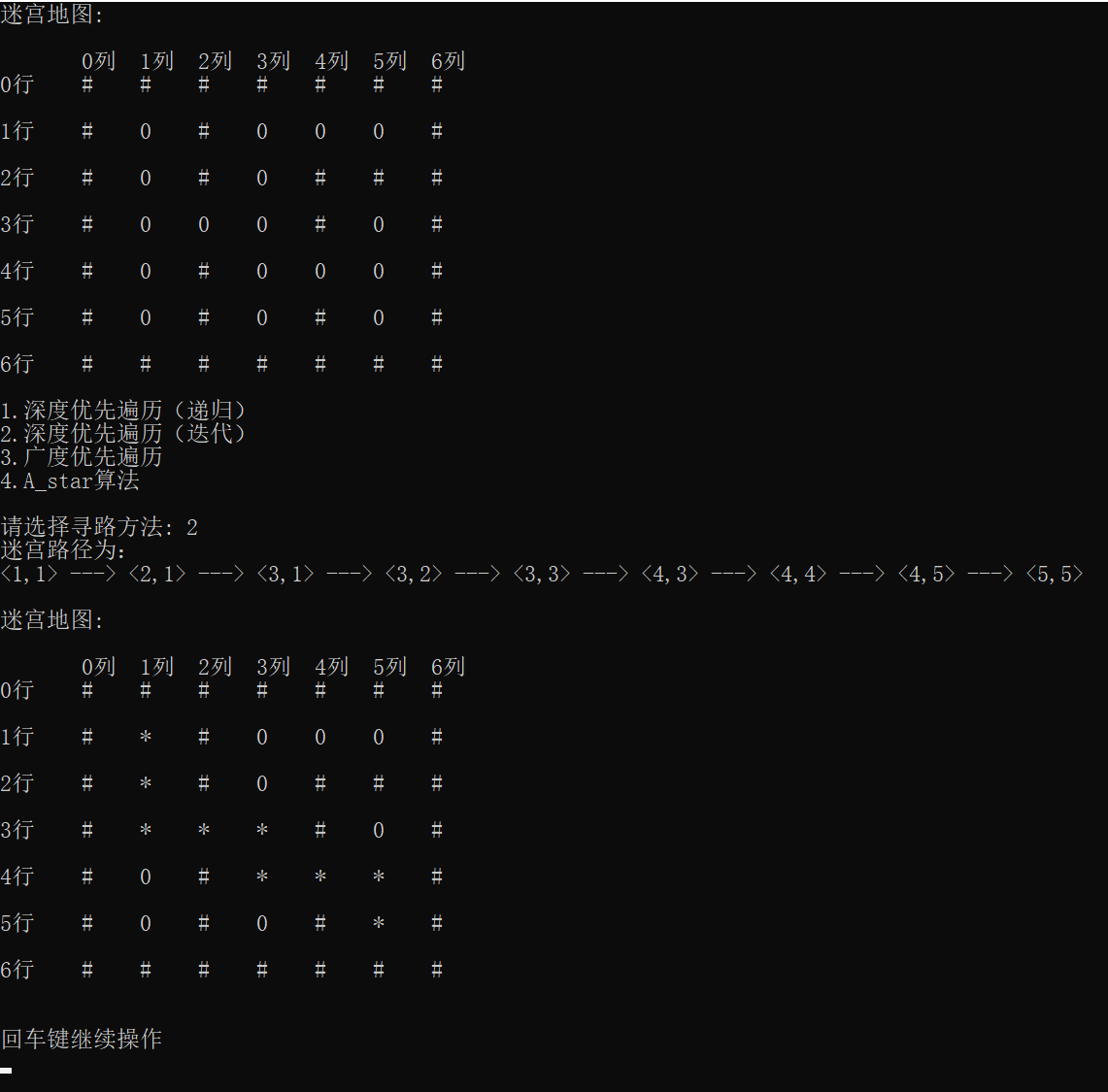
经过比较，基本可以得出上述四种生成迷宫算法的特点，有助于日后对于不同场景的灵活运用。

4 项目测试

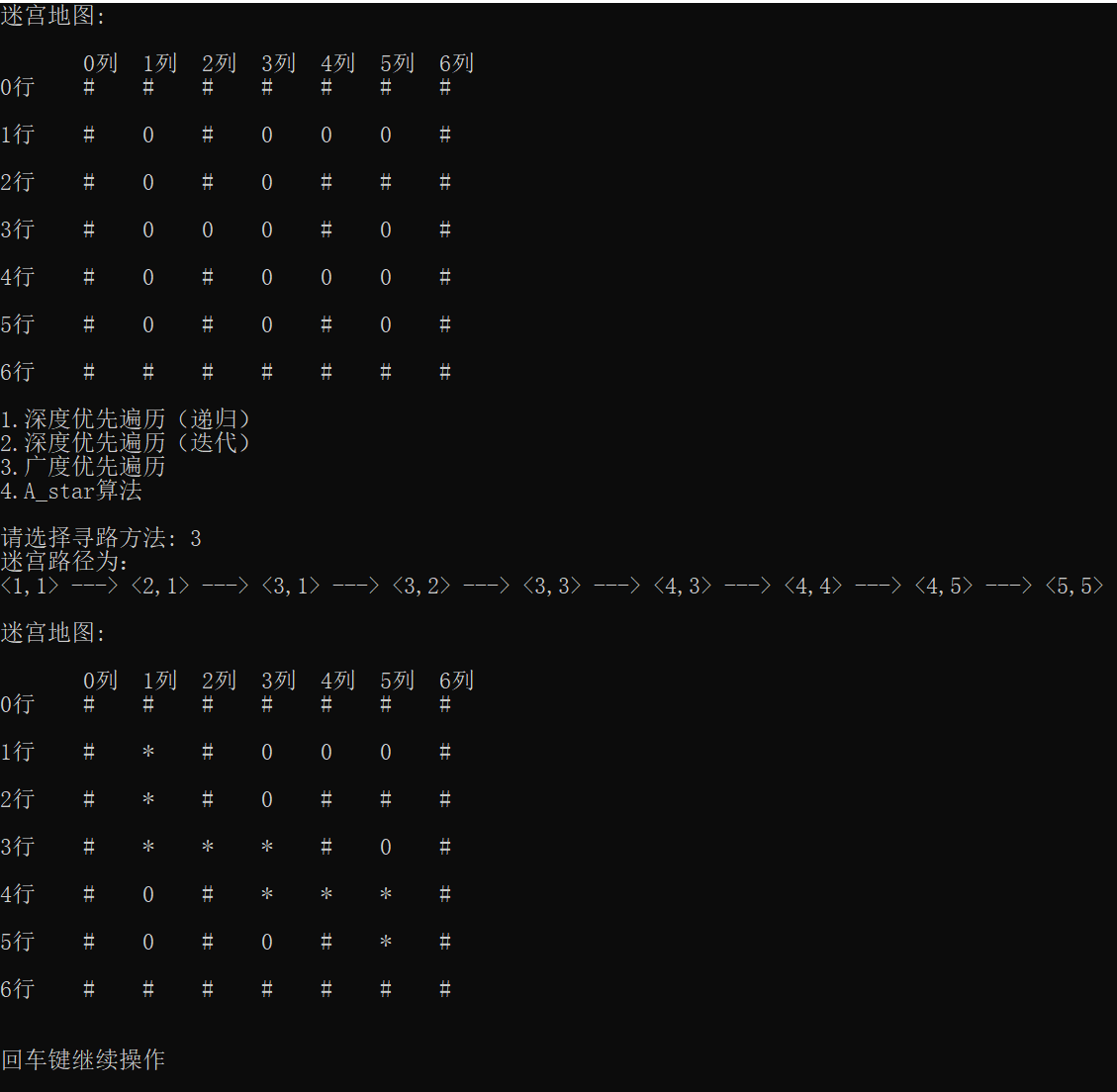
4.1 深度优先遍历(递归)



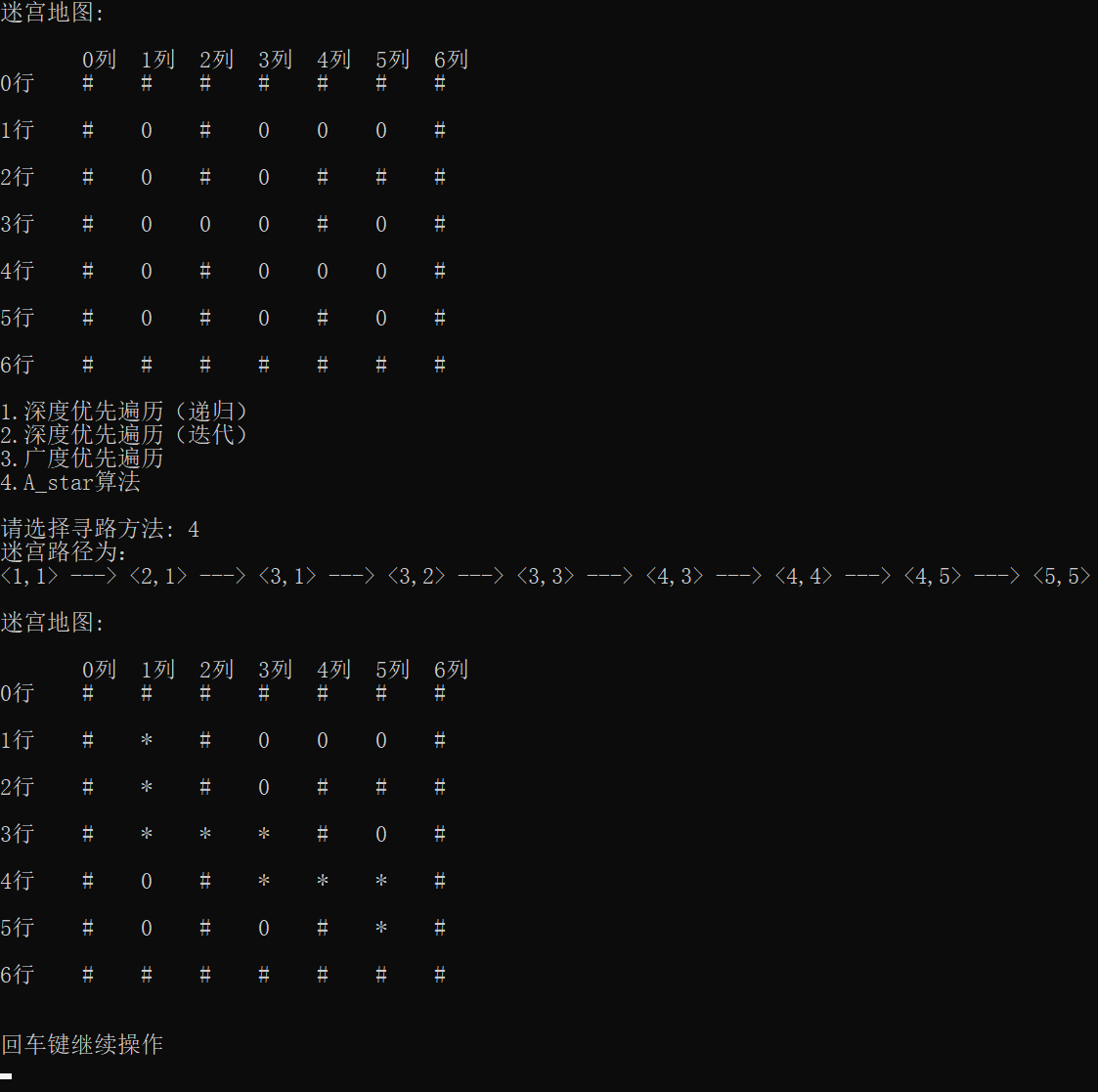
4.2 深度优先遍历(迭代)



4.3 广度优先遍历



4.4 A\_star算法



4.5 输入错误测试

