

# 项目说明文档

## 数据结构课程设计

### ——迷宫游戏

作者姓名 林继申  
学 号 2250758  
指导教师 张 颖  
学院专业 软件学院 软件工程



同濟大學  
TONGJI UNIVERSITY

二〇二三年十二月十三日

# 目录

1 项目分析.....	1
1.1 项目背景分析.....	1
1.2 项目需求分析.....	1
1.3 项目功能分析.....	1
1.3.1 迷宫地图生成功能.....	1
1.3.2 迷宫寻路算法功能.....	2
1.3.2.1 递归回溯搜索算法.....	2
1.3.2.2 深度优先搜索 (DFS) 算法.....	2
1.3.2.3 广度优先搜索 (BFS) 算法.....	2
1.3.2.4 A*搜索算法.....	3
1.3.3 界面设计与可视化功能.....	3
1.3.4 异常处理功能.....	3
2 项目设计.....	3
2.1 数据结构设计.....	3
2.2 结构体与类设计.....	4
2.2.1 <b>MyLinkNode</b> 结构体的设计.....	4
2.2.1.1 概述.....	4
2.2.1.2 结构体定义.....	4
2.2.1.3 数据成员.....	4
2.2.1.4 构造函数.....	4
2.2.2 <b>MyStack</b> 类的设计.....	5
2.2.2.1 概述.....	5
2.2.2.2 类定义.....	5
2.2.2.3 私有数据成员.....	5
2.2.2.4 构造函数.....	5
2.2.2.5 析构函数.....	5
2.2.2.6 公有成员函数.....	6
2.2.3 <b>MyQueue</b> 类的设计.....	6
2.2.3.1 概述.....	6
2.2.3.2 类定义.....	6
2.2.3.3 私有数据成员.....	7
2.2.3.4 构造函数.....	7
2.2.3.5 析构函数.....	7

2.2.3.6 公有成员函数 .....	7
2.2.4 <b>MyPriorityQueue</b> 类的设计 .....	8
2.2.4.1 概述 .....	8
2.2.4.2 类定义 .....	8
2.2.4.3 私有数据成员 .....	8
2.2.4.4 构造函数 .....	8
2.2.4.5 析构函数 .....	9
2.2.4.6 私有成员函数 .....	9
2.2.4.7 公有成员函数 .....	9
2.2.5 <b>Coordinate</b> 结构体的设计.....	9
2.2.5.1 概述 .....	9
2.2.5.2 结构体定义 .....	9
2.2.5.3 数据成员 .....	10
2.2.6 <b>AStarNode</b> 结构体的设计.....	10
2.2.6.1 概述 .....	10
2.2.6.2 结构体定义 .....	10
2.2.6.3 数据成员 .....	10
2.2.6.4 运算符重载 .....	10
2.2.7 <b>Maze</b> 类的设计 .....	11
2.2.7.1 概述 .....	11
2.2.7.2 类定义 .....	11
2.2.7.3 私有数据成员 .....	12
2.2.7.4 构造函数 .....	12
2.2.7.5 析构函数 .....	12
2.2.7.6 私有成员函数 .....	13
2.2.7.7 公有成员函数 .....	13
2.3 项目主体架构设计 .....	14
3 项目功能实现.....	15
3.1 项目主体架构的实现 .....	15
3.1.1 项目主体架构实现思路 .....	15
3.1.2 项目主体架构核心代码 .....	16
3.1.3 项目主体架构示例 .....	18
3.2 迷宫地图生成功能的实现 .....	19
3.2.1 迷宫地图生成功能实现思路 .....	19
3.2.2 迷宫地图生成功能核心代码 .....	20

3.2.3 迷宫地图生成功能示例 .....	21
3.3 迷宫寻路算法功能的实现 .....	22
3.3.1 递归回溯搜索算法 .....	22
3.3.1.1 递归回溯搜索算法实现思路 .....	22
3.3.1.2 递归回溯搜索算法核心代码 .....	22
3.3.1.3 递归回溯搜索算法示例 .....	24
3.3.2 深度优先搜索 (DFS) 算法 .....	24
3.3.2.1 深度优先搜索 (DFS) 算法实现思路 .....	24
3.3.2.2 深度优先搜索 (DFS) 算法核心代码 .....	26
3.3.2.3 深度优先搜索 (DFS) 算法示例 .....	28
3.3.3 广度优先搜索 (BFS) 算法 .....	28
3.3.3.1 广度优先搜索 (BFS) 算法实现思路 .....	28
3.3.3.2 广度优先搜索 (BFS) 算法核心代码 .....	29
3.3.3.3 广度优先搜索 (BFS) 算法示例 .....	31
3.3.4 A*搜索算法 .....	32
3.3.4.1 A*搜索算法实现思路 .....	32
3.3.4.2 A*搜索算法核心代码 .....	33
3.3.4.3 A*搜索算法示例 .....	36
3.4 界面设计与可视化功能的实现 .....	36
3.4.1 界面设计与可视化功能实现思路 .....	36
3.4.2 界面设计与可视化功能核心代码 .....	37
3.4.3 界面设计与可视化功能示例 .....	37
3.5 异常处理功能的实现 .....	37
3.5.1 动态内存申请失败的异常处理 .....	37
3.5.2 MyStack 类栈空的异常处理 .....	38
3.5.3 MyQueue 类队列空的异常处理 .....	38
3.5.4 MyPriorityQueue 类队列满和空的异常处理 .....	38
3.5.5 Maze 类索引越界的异常处理 .....	38
3.5.6 输入非法的异常处理 .....	38
3.5.6.1 迷宫地图大小输入非法的异常处理 .....	38
3.5.6.2 迷宫寻路算法选择输入非法的异常处理 .....	39
4 项目测试 .....	40
4.1 项目主体架构测试 .....	40
4.1.1 输入迷宫地图大小功能测试 .....	40
4.1.2 界面设计与可视化功能测试 .....	41

4.2 迷宫寻路算法功能测试 .....	42
4.2.1 递归回溯搜索算法测试 .....	42
4.2.2 深度优先搜索 (DFS) 算法测试 .....	43
4.2.3 广度优先搜索 (BFS) 算法测试 .....	43
4.2.4 A*搜索算法测试 .....	44
4.3 界面设计与可视化功能测试 .....	44
4.4 退出程序功能测试 .....	45
5 集成开发环境与编译运行环境 .....	45

# 1 项目分析

## 1.1 项目背景分析

迷宫游戏是一个经典的问题，被广泛用于计算机科学和算法学习。它模拟了在复杂环境中找到通路的问题，这可以应用于实际生活中的路径规划、导航系统、游戏开发等领域。不同的搜索算法如递归回溯搜索算法、深度优先搜索 (DFS) 算法、广度优先搜索 (BFS) 算法和 A\* 搜索算法可以用于解决各种路径规划和优化问题。这个项目可以用作寻路算法研究的基础，以评估和比较不同算法在特定情况下的性能。

## 1.2 项目需求分析

基于以上背景分析，本项目需要实现需求如下：

- (1) 迷宫地图生成：迷宫由一个矩形的网格构成，其中包括起点和终点，迷宫中还包括障碍物（墙壁）；
- (2) 迷宫寻路算法：迷宫有不同的寻路算法，如递归回溯搜索算法、深度优先搜索 (DFS) 算法、广度优先搜索 (BFS) 算法和 A\* 搜索算法；
- (3) 界面设计与可视化：迷宫地图与路径的展示需要清晰，可视化程度高；
- (4) 异常处理机制：实现异常处理机制，确保系统稳定性和安全性，避免因用户输入错误导致系统崩溃或信息丢失。

## 1.3 项目功能分析

本项目旨在通过迷宫地图生成、迷宫寻路算法、界面设计与可视化和异常处理等功能，以实现迷宫游戏的核心逻辑。下面对项目的功能进行详细分析。

### 1.3.1 迷宫地图生成功能

本程序基于随机 Prim 生成算法生成迷宫地图。随机 Prim 生成算法生成的迷宫岔路较多，整体上复杂而又较为自然，算法核心思路为：

- (1) 让迷宫全是墙；
- (2) 选一个单元格作为迷宫的通路（一般选择起点），然后把它的邻墙（上下左右）放入列表；
- (3) 当列表里还有墙时，从列表里随机选一个墙，如果这面墙分隔的两个单元格只有一个单元格被访问过，那就从列表里移除这面墙，即把墙打通，让未访

问的单元格成为迷宫的通路，并把这个格子的墙加入列表；如果墙两面的单元格都已经被访问过（都打通了），那就从列表里移除这面墙。

随机 Prim 生成算法生成的迷宫地图没有环形结构，而是树形结构。任何两个空白单元格之间都有路径。

### 1.3.2 迷宫寻路算法功能

本程序使用四种迷宫寻路算法来实现迷宫寻路算法功能，分别为递归回溯搜索算法、深度优先搜索 (DFS) 算法、广度优先搜索 (BFS) 算法和 A\* 搜索算法，以比较不同迷宫寻路算法的特点与性能差异。

这四种算法各有优点和适用场景。递归回溯搜索算法和深度优先搜索 (DFS) 算法适用于简单的迷宫，但可能不够高效。广度优先搜索 (BFS) 算法适用于找到最短路径，但需要更多的内存。A\* 搜索算法结合了启发式估计和搜索，通常在复杂的迷宫中效果最好。

下面是每种迷宫寻路算法的基本实现思路。

#### 1.3.2.1 递归回溯搜索算法

(1) 递归回溯搜索算法是一种基于深度优先搜索的方法，它通过递归地探索迷宫中的每个可能路径；

(2) 从起点出发，尝试向上、下、左、右四个方向移动，每次选择一个方向并继续探索；

(3) 如果遇到障碍物或者超出边界，就回退到上一个位置，然后选择下一个可能的方向；

(4) 重复这个过程，直到找到终点或者确定没有通路可达终点。

#### 1.3.2.2 深度优先搜索 (DFS) 算法

(1) 深度优先搜索也是一种基于栈的搜索算法，它将一条路径完全探索到底，然后再回溯并尝试其他路径；

(2) 从起点开始，选择一个方向前进，一直走到不能再前进为止，然后回退到上一个节点并继续尝试其他方向；

(3) 重复这个过程，直到找到终点或者确定没有通路可达终点。

#### 1.3.2.3 广度优先搜索 (BFS) 算法

(1) 广度优先搜索是一种基于队列的搜索算法，它从起点开始，逐层地向外扩展搜索；

(2) 首先将起点放入队列，然后遍历与起点相邻的节点，并将它们放入队列；

(3) 然后依次取出队列中的节点，继续遍历它们的相邻节点，并将未访问过

的节点放入队列；

(4) 重复这个过程，直到找到终点或者确定没有通路可达终点。

#### 1.3.2.4 A\*搜索算法

(1) A\*搜索算法是一种启发式搜索算法，它结合了深度优先搜索和广度优先搜索的思想，并使用启发式函数来估计每个节点到终点的距离；

(2) 算法维护一个优先队列，根据启发式函数的估计值选择下一个要探索的节点，以便更有可能找到最优解；

(3) A\*算法通过不断更新节点的代价估计值，选择最有希望的路径进行搜索，直到找到终点或者确定没有通路可达终点。

#### 1.3.3 界面设计与可视化功能

迷宫地图与路径的展示需要清晰，可视化程度高。

#### 1.3.4 异常处理功能

实现异常处理机制，处理用户可能输入的非法信息，确保系统的稳定性和安全性。

## 2 项目设计

### 2.1 数据结构设计

基于项目分析，本项目使用动态分配的 `int` 类型二维数组存储迷宫地图，原因如下：

(1) 简单高效：使用动态分配的 `int` 类型的二维数组来表示迷宫地图是一种非常简单高效的方式。整数的大小相对较小，占用的内存较少；

(2) 可读性和维护性：通过宏定义的方式，表示迷宫地图中的不同状态（空白、墙壁、路径）。这种表示方式容易理解，有利于增加项目的可读性和维护性，并进行必要的修改和调试；

迷宫地图中不同状态的宏定义如下：

```
#define MAZE_BLANK 0 //空白
#define MAZE_WALL 1 //墙壁
#define MAZE_PATH 2 //路径
```

(3) 易于扩展和修改：采用整数表示，可以轻松地扩展迷宫的状态。例如可以通过定义更多的宏来表示其他状态，如 `MAZE_TREASURE` 表示宝藏，或 `MAZE_MONSTER` 表示怪物，以扩展迷宫游戏的功能；



可扩展迷宫游戏功能的宏定义如下：

```
#define MAZE_TREASURE 3 //宝藏
#define MAZE_MONSTER 4 //怪物
```

(4) 利于状态检测：使用整数值来表示不同的状态可以方便地检测迷宫中的当前状态，如检查某个位置是否是墙壁、路径或空白，从而执行相应的操作。

## 2.2 结构体与类设计

### 2.2.1 MyLinkNode 结构体的设计

#### 2.2.1.1 概述

**MyLinkNode** 结构体是一个用于构建链表节点的模板结构体。该结构体用于表示链表中的每个节点，其中包括节点存储的数据以及指向下一个节点的指针。经典的链表一般包括两个抽象数据类型（ADT）——链表结点类（**LNode**）与链表类（**LinkList**）。本项目希望链表结点类可以直接访问链表结点，所以使用 **struct** 而不是 **class** 描述链表结点类。

#### 2.2.1.2 结构体定义

```
template <typename Type>
struct MyLinkNode {
    Type data;
    MyLinkNode<Type>* link;
    MyLinkNode(MyLinkNode<Type>* ptr = NULL) { link = ptr; }
    MyLinkNode(const Type& item, MyLinkNode<Type>* ptr = NULL)
{ data = item; link = ptr; }
};
```

#### 2.2.1.3 数据成员

**Type data**: 数据域，存储节点的数据

**MyLinkNode<Type>\* link**: 指针域，指向下一个节点的指针

#### 2.2.1.4 构造函数

**MyLinkNode(MyLinkNode<Type>\* ptr = NULL);**

构造函数，初始化指针域。

**MyLinkNode(const Type& item, MyLinkNode<Type>\* ptr = NULL);**

构造函数，初始化数据域和指针域。

## 2.2.2 MyStack 类的设计

### 2.2.2.1 概述

该通用模板类 **MyStack** 用于创建和操作堆栈数据结构。堆栈是一种基本的数据结构，它遵循先进后出（**Last-In-First-Out, LIFO**）的原则，允许用户推入（**push**）和弹出（**pop**）元素，元素（即链表节点）由 **MyLinkNode** 结构体表示，其中包含数据和指针域。此模板类 **MyStack** 实现了堆栈的基本功能，并提供了一些辅助方法以方便操作。

### 2.2.2.2 类定义

```
template <typename Type>
class MyStack {
private:
    MyLinkNode<Type>* top;
public:
    MyStack() : top(NULL) {}
    ~MyStack() { makeEmpty(); }
    bool isEmpty(void) const { return top == NULL; }
    void makeEmpty(void);
    void push(const Type& item);
    bool pop(Type& item);
    bool getTop(Type& item);
    int getSize(void) const;
};
```

### 2.2.2.3 私有数据成员

**MyLinkNode<Type>\* top**: 指向堆栈顶部元素的指针

### 2.2.2.4 构造函数

```
MyStack() : top(NULL) {}
```

默认构造函数，创建一个空的堆栈。

### 2.2.2.5 析构函数

```
~MyStack() { makeEmpty(); }
```

析构函数，清空堆栈并释放内存。

### 2.2.2.6 公有成员函数

```
bool isEmpty(void) const { return top == NULL; }
```

判断堆栈是否为空，如果堆栈为空，它将返回 `true`，否则返回 `false`。

```
void makeEmpty(void);
```

清空堆栈，释放所有分配的内存。它通过遍历堆栈的节点，释放它们，并将堆栈的 `top` 指针置为 `NULL` 来实现。

```
void push(const Type& item);
```

用于将新元素推入堆栈。它会在堆栈的顶部创建一个新节点，将元素存储在 其中，并将新节点的 `link` 指向旧的堆栈顶部。

```
bool pop(Type& item);
```

从堆栈中弹出顶部元素，将其存储在传入的参数中，并释放相应的节点。如果堆栈为空，它将返回 `false`，否则返回 `true`。

```
bool getTop(Type& item);
```

用于获取堆栈的顶部元素，但不弹出它。如果堆栈为空，它将返回 `false`，否则返回 `true`。

```
int getSize(void) const;
```

计算堆栈中的元素数量，通过遍历堆栈节点并计数节点的数量来实现。

## 2.2.3 MyQueue 类的设计

### 2.2.3.1 概述

该通用模板类 `MyQueue` 用于存储和管理数据元素，遵循先进先出（`First-in-First-Out`, `FIFO`）的原则。该队列是基于链表数据结构实现的，链表节点由 `MyLinkNode` 结构体表示，其中包含数据和指针域。此模板类 `MyQueue` 允许在队列的前端（`front`）添加元素，以及从队列的前端（`front`）移除元素。

### 2.2.3.2 类定义

```
template <typename Type>
class MyQueue {
private:
    MyLinkNode<Type>* front;
    MyLinkNode<Type>* rear;
public:
    MyQueue() : front(NULL), rear(NULL) {}
    ~MyQueue() { makeEmpty(); }
    bool isEmpty(void) const { return front == NULL; }
```

```

    void makeEmpty(void);
    void enqueue(const Type& item);
    bool dequeue(Type& item);
    bool getFront(Type& item) const;
    int getSize(void) const;
};

```

#### 2.2.3.3 私有数据成员

`MyLinkNode<Type>* front`: 指针指向队列的前端，即队列中的第一个元素

`MyLinkNode<Type>* rear`: 指针指向队列的后端，即队列中的最后一个元素

#### 2.2.3.4 构造函数

```
MyQueue() : front(NULL), rear(NULL) {}
```

默认构造函数，创建一个空的队列。

#### 2.2.3.5 析构函数

```
~MyQueue() { makeEmpty(); }
```

析构函数，清空队列并释放内存。

#### 2.2.3.6 公有成员函数

```
bool isEmpty(void) const { return front == NULL; }
```

检查队列是否为空。

```
void makeEmpty(void);
```

清空队列，释放所有元素占用的内存。

```
void enqueue(const Type& item);
```

将元素添加到队列的末尾。

```
bool dequeue(Type& item);
```

移除队列的前端元素并将其值通过引用返回。

```
bool getFront(Type& item) const;
```

获取队列的前端元素的值。

```
int getSize(void) const;
```

获取队列中元素的数量。

## 2.2.4 MyPriorityQueue 类的设计

### 2.2.4.1 概述

该通用模板类 `MyPriorityQueue` 用于存储和管理具有不同优先级的数据元素。这个优先队列是基于线性表数据结构实现的，可以将元素按照它们的优先级进行插入和移除。

### 2.2.4.2 类定义

```
template <typename Type>
class MyPriorityQueue {
private:
    Type* elements;
    int count;
    int maxSize;
    void adjust(void);
public:
    MyPriorityQueue(int _maxSize);
    ~MyPriorityQueue() { delete[] elements; }
    void makeEmpty(void) { count = 0; }
    bool isEmpty(void) const { return count == 0; }
    bool isFull(void) const { return count == maxSize; }
    int getSize(void) const { return count; }
    bool insert(const Type& item);
    bool remove(Type& item);
    bool getFront(Type& item) const;
};
```

### 2.2.4.3 私有数据成员

`Type* elements`: 指向存储元素的数组的指针  
`int count`: 当前队列中包含的元素数量  
`int maxSize`: 队列可以容纳的元素的最大数量

### 2.2.4.4 构造函数

```
MyPriorityQueue(int _maxSize);
```

默认构造函数,用于创建 `MyPriorityQueue` 类的对象(一个空的优先队列),需要指定最大容量 `_maxSize`。

#### 2.2.4.5 析构函数

```
~MyPriorityQueue() { delete[] elements; }
```

析构函数，用于销毁 `MyPriorityQueue` 类的对象并释放内存。

#### 2.2.4.6 私有成员函数

```
void adjust(void);
```

用于根据插入的元素优先级重新调整队列，以维护优先队列的性质。

#### 2.2.4.7 公有成员函数

```
void makeEmpty(void) { count = 0; }
```

清空队列，将队列中元素的数量重置为零。

```
bool isEmpty(void) const { return count == 0; }
```

检查队列是否为空。

```
bool isFull(void) const { return count == maxSize; }
```

检查队列是否已满。

```
int getSize(void) const { return count; }
```

获取队列中元素的数量。

```
bool insert(const Type& item);
```

将元素插入队列，根据元素的优先级进行调整。

```
bool remove(Type& item);
```

移除队列的前端元素，并将其值通过引用返回。

```
bool getFront(Type& item) const;
```

获取队列的前端元素的值。

### 2.2.5 Coordinate 结构体的设计

#### 2.2.5.1 概述

`Coordinate` 结构体是一个用于表示二维坐标的数据结构，用于记录行 (`row`) 和列 (`col`) 的位置信息。

#### 2.2.5.2 结构体定义

```
typedef struct {  
    int row;  
    int col;  
} Coordinate;
```

### 2.2.5.3 数据成员

```
int row: 行数  
int col: 列数
```

## 2.2.6 AStarNode 结构体的设计

### 2.2.6.1 概述

**AStarNode** 结构体是为了支持 A\*搜索算法而设计的数据结构。A\*搜索算法是一种常用的图搜索算法，用于找到从起点到目标点的最短路径。在该算法中 **AStarNode** 结构体用于表示图中的节点，其中每个节点代表搜索过程中的一个状态，包括节点的位置（行和列）、从起始节点到当前节点的累积代价（**accumuCost**），以及预估到目标节点的代价（**totalCost**）。通过 **AStarNode** 结构体，A\*搜索算法能够有效地搜索图并找到最佳路径。

### 2.2.6.2 结构体定义

```
typedef struct {  
    int row;  
    int col;  
    int accumuCost;  
    int totalCost;  
} AStarNode;
```

### 2.2.6.3 数据成员

```
int row: 行数  
int col: 列数  
int accumuCost: 累积代价  
int totalCost: 总代价
```

### 2.2.6.4 运算符重载

**operator<=**函数是一个运算符重载函数，目的是实现<=运算符的重载，以用于比较两个 **AStarNode** 结构体对象的 **totalCost** 数据成员。这个函数通常与 A\*搜索算法一起使用，用于比较两个节点的总代价，以确定哪个节点更适合作为下一个搜索步骤的候选节点。在 A\*搜索算法中，总代价通常是节点的启发式估值（通常是到目标节点的估计代价）与节点的累积代价之和。通过运算符重载，使得代码更加清晰和可读，提高了 A\*搜索算法的可维护性和可理解性。在算法中，通常需要多次比较节点的总代价，因此重载运算符可以使代码更加简洁和易于编写。**operator<=**函数如下：

```

bool operator<=(const AStarNode& left, const AStarNode& right)
{
    return left.totalCost <= right.totalCost;
}

```

## 2.2.7 Maze 类的设计

### 2.2.7.1 概述

**Maze** 类是一个用于生成迷宫并实现多种寻路算法的类。该类的主要目的是提供一个模块化的迷宫生成和寻路解决方案，使用户能够轻松地生成迷宫地图，进行不同类型的路径搜索，以及获取找到的路径。

### 2.2.7.2 类定义

```

class Maze {
private:
    int** mazeMap;
    int rows;
    int cols;
    int startRow;
    int startCol;
    int targetRow;
    int targetCol;
    int currRow;
    int currCol;
    int mazePointCount;
    struct MazePoint { int row; int col; Direction
direction; };
    MazePoint* mazePointList;
    MyStack<Coordinate> path;
    void pushList(const struct MazePoint& mazePoint);
    void popList(int index);
    void findAdjacentWalls(void);
    void generateMaze(void);
    bool isValid(int row, int col);
    bool recursivePathfinding(int row, int col, bool**
visited);
public:

```



```

        Maze(int _rows, int _cols, int _startRow, int _startCol,
int _targetRow, int _targetCol);
        ~Maze();
        void output(void);
        bool recursiveBacktracking(void);
        bool DFS(void);
        bool BFS(void);
        bool AStar(void);
        MyStack<Coordinate>& getPath(void) { return path; }
};

```

### 2.2.7.3 私有数据成员

```

int** mazeMap: 迷宫地图
int rows: 迷宫行数
int cols: 迷宫列数
int startRow: 起始点行数
int startCol: 起始点列数
int targetRow: 目标点行数
int targetCol: 目标点列数
int currRow: 当前行数
int currCol: 当前列数
int mazePointCount: 迷宫中 MazePoint 结构体的数量
struct MazePoint { int row; int col; Direction direction; }:
迷宫中的一个点（成员变量：行数、列数、该点是从哪个方向添加到迷宫中的）
MazePoint* mazePointList: 指向 MazePoint 结构体的动态数组的指针
MyStack<Coordinate> path: 用于存储路径坐标的自定义栈

```

### 2.2.7.4 构造函数

```

Maze(int _rows, int _cols, int _startRow, int _startCol, int
_targetRow, int _targetCol);

```

构造函数,用于初始化 Maze 对象,并为迷宫地图的创建和初始化提供参数。它接受行数、列数、起始点坐标和目标点坐标作为参数,并初始化迷宫地图、起始点、目标点以及其他必要的成员变量。

### 2.2.7.5 析构函数

```

~Maze();

```

析构函数，负责释放动态分配的内存，包括迷宫地图和相关数据结构。当 **Maze** 对象不再需要时，析构函数确保释放所有占用的内存，以避免内存泄漏。

#### 2.2.7.6 私有成员函数

**void pushList(const struct MazePoint& mazePoint);**

将给定的 **MazePoint** 结构添加到 **mazePointList** 中，用于在迷宫生成的过程中，将可能的路径点添加到候选点列表中。

**void popList(int index);**

从 **mazePointList** 中移除指定索引处的元素，以维护候选点列表的完整性。

**void findAdjacentWalls(void);**

查找当前位置周围的墙壁，并将这些墙壁添加到候选点列表中，以便在迷宫生成中进一步处理。

**void generateMaze(void);**

生成迷宫的核心函数，它调用了其他函数来执行迷宫生成的过程。

**bool isValid(int row, int col);**

检查给定的行号和列号是否在迷宫地图的有效范围内。

**bool recursivePathfinding(int row, int col, bool\*\* visited);**

函数实现了迷宫中的递归路径搜索，用于查找从起始点到目标点的路径。

#### 2.2.7.7 公有成员函数

**void output(void);**

将当前生成的迷宫地图以可视化形式输出到控制台，以便用户查看迷宫地图与路径。

**bool recursiveBacktracking(void);**

递归回溯搜索算法，用于在生成的迷宫中查找一条从起始点到目标点的路径。

**bool DFS(void);**

深度优先搜索 (DFS) 算法，用于在生成的迷宫中查找一条从起始点到目标点的路径。

**bool BFS(void);**

广度优先搜索 (BFS) 算法，用于在生成的迷宫中查找一条从起始点到目标点的路径。

**bool AStar(void);**

A\*搜索算法，用于在生成的迷宫中查找一条从起始点到目标点的路径。

**MyStack<Coordinate>& getPath(void) { return path; }**

返回一个指向 **path** 堆栈的引用，允许用户访问通过迷宫寻路算法找到的路径坐标。

## 2.3 项目主体架构设计

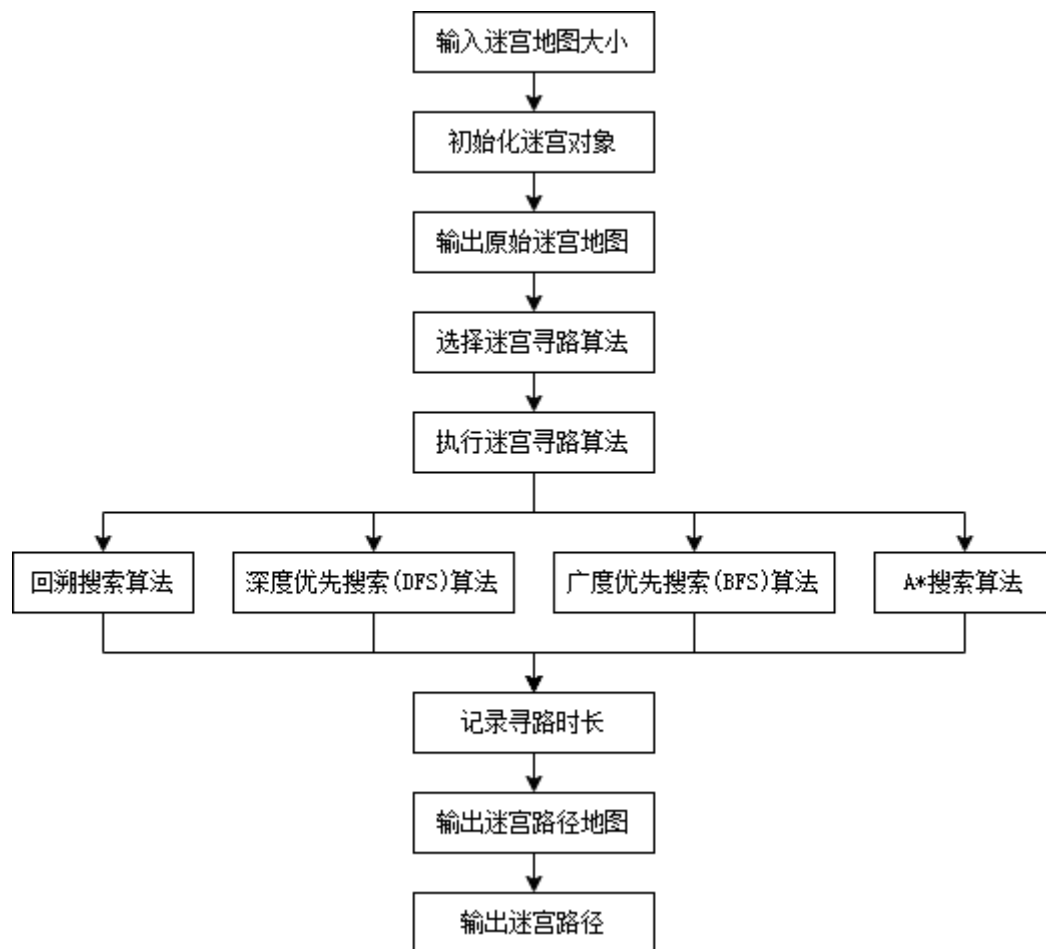


图 2.3.1 项目主体架构设计流程图

项目主体架构设计为：

- (1)输入迷宫地图大小：用户被要求输入迷宫的行数和列数，要求是奇数，并在一定范围内；
- (2)初始化迷宫对象：使用用户提供的行数、列数，以及迷宫的起始位置和终点位置，初始化一个迷宫对象；
- (3)输出原始迷宫地图：输出初始生成的迷宫地图，显示迷宫中的墙壁和通道；
- (4)选择迷宫寻路算法：允许用户选择一个迷宫寻路算法，并根据用户的选择，执行四种迷宫寻路算法之一；
- (5)执行迷宫寻路算法：递归回溯搜索算法、深度优先搜索(DFS)算法、广度优先搜索(BFS)算法、A\*搜索算法；
- (6)记录寻路时长：记录迷宫寻路算法执行所需的时间；
- (7)输出迷宫路径地图：输出包含找到的路径的迷宫地图；
- (8)输出迷宫路径：如果找到了路径，将路径的坐标输出到屏幕上。如果没有找到路径，则显示未找到路径的提示信息。

### 3 项目功能实现

#### 3.1 项目主体架构的实现

##### 3.1.1 项目主体架构实现思路

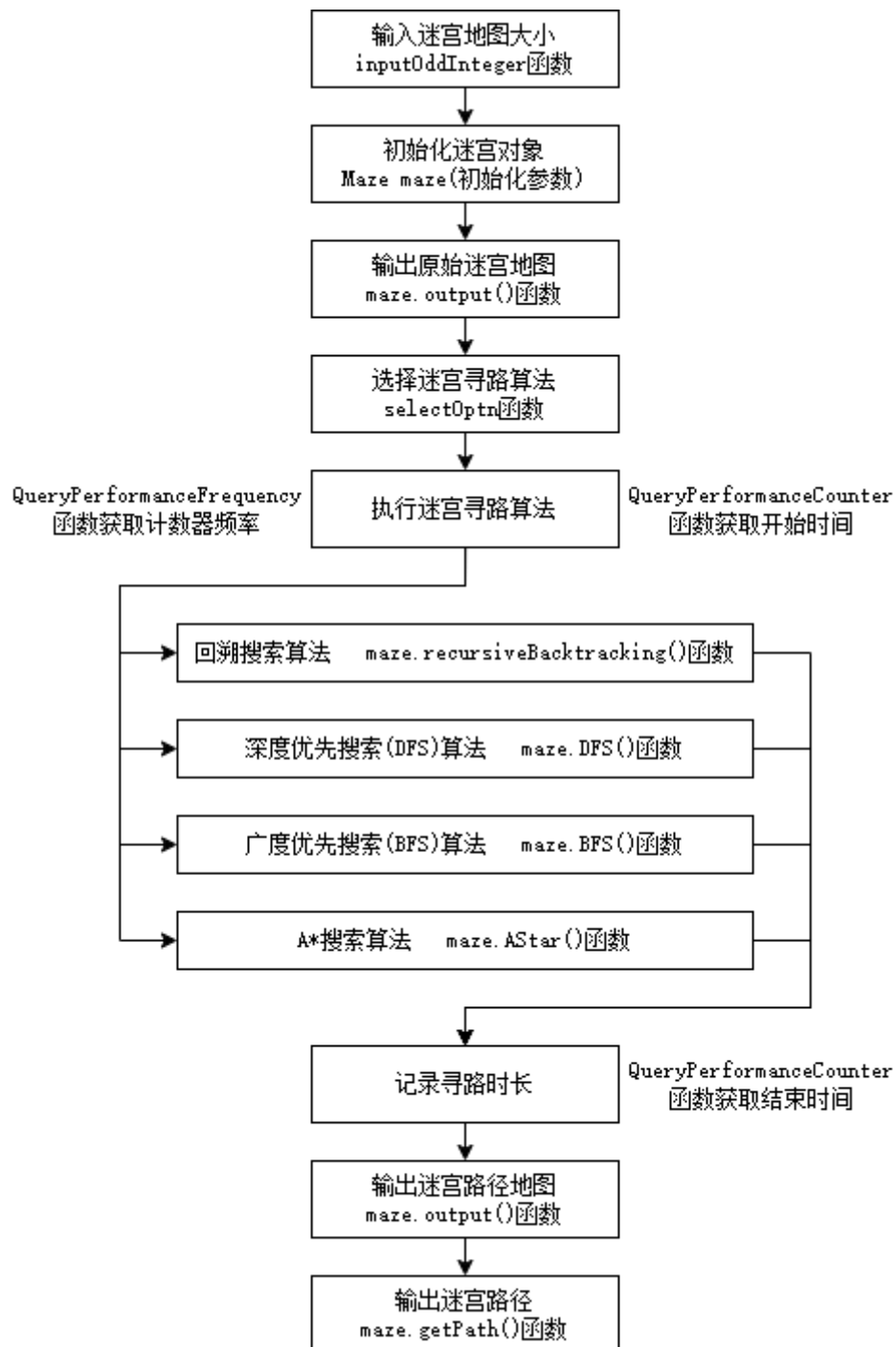


图 3.1.1.1 项目主体架构实现流程图

项目主体架构实现思路为：

(1) 输入迷宫地图大小：分别调用两次 `inputOddInteger` 函数，输入迷宫

地图的行数和列数，要求为奇数，迷宫地图的行数和列数的返回通过常全局变量的方式定义，便于修改和维护；

```
const int mazeSizeLowerLimit = 7;
const int mazeSizeUpperLimit = 99;
```

(2) 初始化迷宫对象：创建一个 `Maze` 对象，传入迷宫的行数、列数，起始坐标（`mazeStartRow` 和 `mazeStartCol`），以及终点坐标（`mazeRows-2` 和 `mazeCols-2`）。起始坐标通过常全局变量的方式定义，便于修改和维护，默认起始坐标为迷宫地图左上角（不算边界），终点坐标为迷宫地图右下角（不算边界）；

```
const int mazeStartRow = 1;
const int mazeStartCol = 1;
```

(3) 调用 `maze.output()` 函数输出原始迷宫地图；

(4) 调用 `selectOptn` 函数选择迷宫寻路算法；

(5) 根据 `selectOptn` 函数返回的选项结果执行迷宫寻路算法：递归回溯搜索算法（调用 `maze.recursiveBacktracking()` 函数）、深度优先搜索 (DFS) 算法（调用 `maze.DFS()` 函数）、广度优先搜索 (BFS) 算法（调用 `maze.BFS()` 函数）、A\* 搜索算法（调用 `maze.AStar()` 函数）；

(6) 记录寻路时长：使用 `QueryPerformanceFrequency` 函数获取计数器频率，使用 Windows 性能计数器 `QueryPerformanceCounter` 函数获取开始时间（`begin`）和结束时间（`end`），转换秒数，来测量算法执行时间；

(7) 调用 `maze.output()` 函数输出迷宫路径地图；

(8) 如果找到了路径，将路径的坐标输出到屏幕上，显示从起点到终点的移动路径；如果未找到路径，输出未找到路径的消息。路径坐标信息存储在 `Maze` 类的私有数据成员自定义栈 `path` 中，可以通过调用 `Maze` 类的成员函数 `maze.getPath()` 获取路径坐标信息。

### 3.1.2 项目主体架构核心代码

```
void mazeGame(void)
{
    /* System entry prompt */
    std::cout << "+-----+" << std::endl;
    std::cout << "|  迷宫游戏  |" << std::endl;
    std::cout << "|  Maze Game |" << std::endl;
    std::cout << "+-----+" << std::endl << std::endl;
    std::cout << ">>> 本程序基于随机 Prim 生成算法生成迷宫地图" <<
std::endl << std::endl;

    /* Input the size of the maze map */
    int      mazeRows      =      inputOddInteger(mazeSizeLowerLimit,
```

```

mazeSizeUpperLimit, "迷宫地图行数");
    std::cout << std::endl;
    int      mazeCols      =      inputOddInteger(mazeSizeLowerLimit,
mazeSizeUpperLimit, "迷宫地图列数");
    std::cout << std::endl;

    /* Initialize the maze */
    Maze  maze(mazeRows,  mazeCols,  mazeStartRow,  mazeStartCol,
mazeRows - 2, mazeCols - 2);

    /* Output the original maze map */
    std::cout << ">>> 迷宫地图" << std::endl << std::endl;
    maze.output();

    /* Select a pathfinding algorithm */
    LARGE_INTEGER tick, begin, end;
    QueryPerformanceFrequency(&tick);
    int optn = selectOptn();
    QueryPerformanceCounter(&begin);
    if(optn == 1)
        maze.recursiveBacktracking();
    else if(optn == 2)
        maze.DFS();
    else if(optn == 3)
        maze.BFS();
    else if (optn == 4)
        maze.AStar();
    QueryPerformanceCounter(&end);

    /* Output the maze path map */
    maze.output();

    /* Output the maze path */
    if (maze.getPath().isEmpty()) {
        std::cout << std::endl << ">>> 未找到迷宫路径" << std::endl <<
std::endl;
        return;
    }
    std::cout << std::endl << ">>> 迷宫路径（寻路时长： " <<
std::setiosflags(std::ios::fixed) << std::setprecision(6) <<
double(end.QuadPart - begin.QuadPart) / tick.QuadPart << "秒" << ") " <<
std::endl << std::endl;
    while (!maze.getPath().isEmpty()) {
        Coordinate coord;

```

```

        maze.getPath().pop(coord);
        std::cout << "(" << coord.row << "," << coord.col << ")";
        if (maze.getPath().getSize() > 0)
            std::cout << " --> ";
    }
    std::cout << std::endl << std::endl;
}

```

### 3.1.3 项目主体架构示例

```

+-----+
|  迷宫游戏  |
|  Maze Game  |
+-----+

>>> 本程序基于随机Prim生成算法生成迷宫地图
请输入迷宫地图行数（奇数）[整数范围：7~99]：15
请输入迷宫地图列数（奇数）[整数范围：7~99]：15
>>> 迷宫地图

  始
  15x15 grid maze visualization with start (始) and end (终) markers.

>>> 迷宫寻路算法：[1]递归回溯搜索算法 [2]深度优先搜索(DFS)算法 [3]广度优先搜索(BFS)算法 [4]A*搜索算法
请选择迷宫寻路算法：[1]

  15x15 grid maze visualization showing the search path (marked with 'x') from start (始) to end (终).

>>> 迷宫路径（寻路时长：0.000027秒）
(1, 1) --> (2, 1) --> (3, 1) --> (3, 2) --> (3, 3) --> (4, 3) --> (5, 3) --> (5, 2) --> (5, 1) --> (6, 1) --> (
7, 1) --> (8, 1) --> (9, 1) --> (9, 2) --> (9, 3) --> (10, 3) --> (11, 3) --> (11, 4) --> (11, 5) --> (12, 5) --
-> (13, 5) --> (13, 6) --> (13, 7) --> (13, 8) --> (13, 9) --> (13, 10) --> (13, 11) --> (13, 12) --> (13, 13)

Press Enter to Quit

```

图 3.1.3.1 项目主体架构示例

## 3.2 迷宫地图生成功能的实现

### 3.2.1 迷宫地图生成功能实现思路

本程序使用随机 Prim 生成算法生成迷宫地图的思路为：

(1) 在构造函数 (`Maze::Maze`) 中，首先初始化迷宫地图 (`mazeMap`) 为墙 (`MAZE_WALL`)，并将起始点 (`startRow` 和 `startCol`) 设置为迷宫的起点，将目标点 (`targetRow` 和 `targetCol`) 设置为迷宫的终点；

(2) 使用随机 Prim 算法来生成迷宫路径的关键部分在 `generateMaze` 函数中。开始时，创建一个空的迷宫点列表 (`mazePointList`)，然后通过 `findAdjacentWalls` 函数查找当前位置周围的墙，将这些墙的坐标和方向添加到 `mazePointList` 中；

(3) 进入循环，从 `mazePointList` 随机选择一个墙，如果这面墙分隔的两个单元格只有一个单元格被访问过，那就从列表里移除这面墙，即把墙打通（将当前位置和墙的位置设置为通路 `MAZE_BLANK`）。如果墙两面的单元格都已经被访问过（都打通了），那就从列表里移除这面墙；

(4) 再次使用 `findAdjacentWalls` 函数查找新的周围墙，将它们添加到 `mazePointList`；

(5) 从 `mazePointList` 中移除已经处理的墙，重复上述步骤，直到 `mazePointList` 为空。

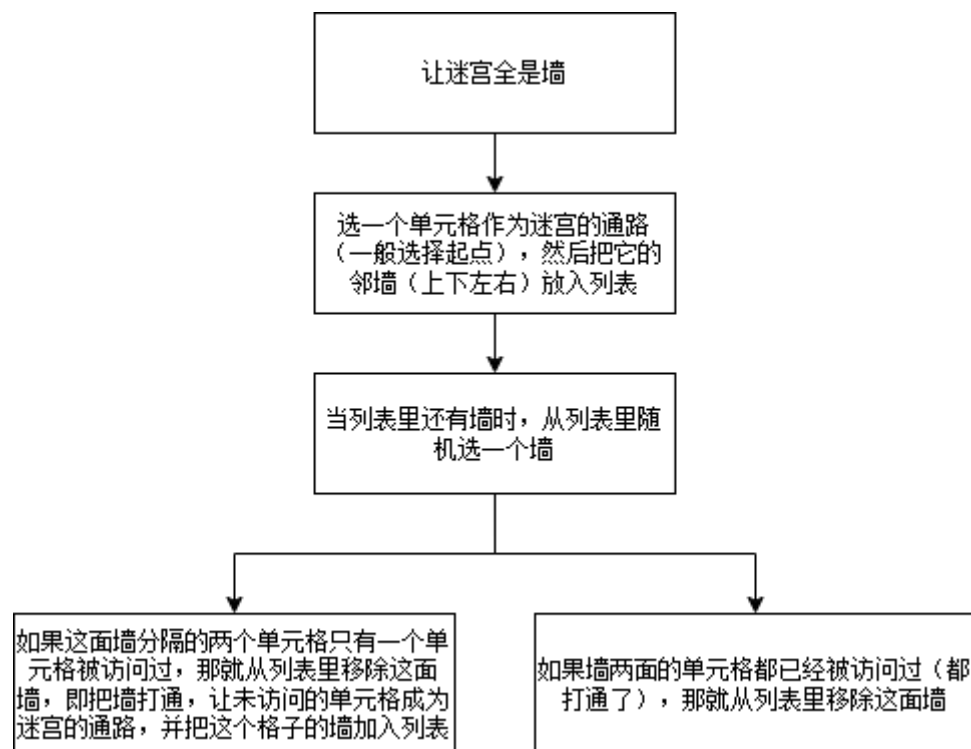


图 3.2.1.1 迷宫地图生成功能实现流程图



### 3.2.2 迷宫地图生成功能核心代码

```
void Maze::pushList(const struct MazePoint& mazePoint)
{
    mazePointList[mazePointCount++] = mazePoint;
}

void Maze::popList(int index)
{
    if (index < 0 || index >= mazePointCount) {
        std::cerr << "Error: Invalid index for erase operation." <<
std::endl;
        exit(INVALID_INDEX_ERROR);
    }
    for (int i = index; i < mazePointCount - 1; i++)
        mazePointList[i] = mazePointList[i + 1];
    mazePointCount--;
}

void Maze::findAdjacentWalls(void)
{
    if (currRow < rows && mazeMap[currRow + 1][currCol] == MAZE_WALL)
        pushList({ currRow + 1, currCol, Down });
    if (currCol < cols && mazeMap[currRow][currCol + 1] == MAZE_WALL)
        pushList({ currRow, currCol + 1, Right });
    if (currRow > 1 && mazeMap[currRow - 1][currCol] == MAZE_WALL)
        pushList({ currRow - 1, currCol, Up });
    if (currCol > 1 && mazeMap[currRow][currCol - 1] == MAZE_WALL)
        pushList({ currRow, currCol - 1, Left });
}

void Maze::generateMaze(void)
{
    findAdjacentWalls();
    while (mazePointCount) {
        int index = rand() % mazePointCount;
        MazePoint currPoint = mazePointList[index];
        currRow = currPoint.row;
        currCol = currPoint.col;
        if (currPoint.direction == Up)
            currRow--;
        else if (currPoint.direction == Down)
            currRow++;
        else if (currPoint.direction == Left)
            currCol--;
    }
}
```

```

        else if (currPoint.direction == Right)
            currCol++;
        if (isValid(currRow, currCol) && mazeMap[currRow][currCol] ==
MAZE_WALL) {
            mazeMap[currPoint.row][currPoint.col]
=
            mazeMap[currRow][currCol] = MAZE_BLANK;
            findAdjacentWalls();
        }
        popList(index);
    }
}

```

### 3.2.3 迷宫地图生成功能示例

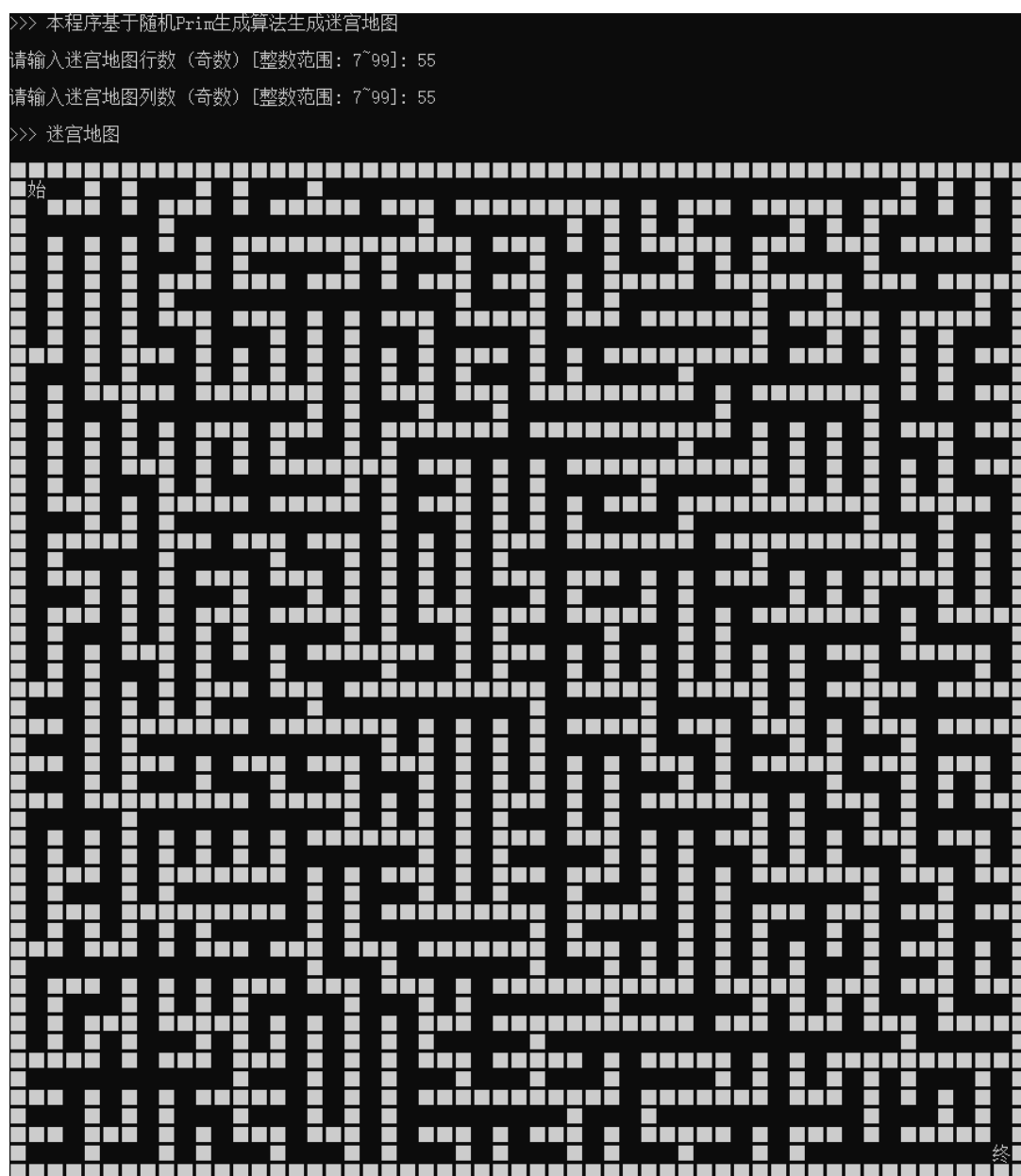


图 3.2.3.1 迷宫地图生成功能示例

## 3.3 迷宫寻路算法功能的实现

### 3.3.1 递归回溯搜索算法

#### 3.3.1.1 递归回溯搜索算法实现思路

执行递归回溯搜索算法的函数为 `Maze` 类的成员函数 `recursiveBacktracking`，其实现思路为：

(1)分配内存和初始化：首先，函数分配一个二维布尔数组 `visited`，用于跟踪迷宫中的每个位置是否已被访问过。检查内存分配是否成功，如果失败则输出错误信息并退出程序；

(2)路径搜索：使用 `recursivePathfinding` 函数来尝试从起点出发递归地搜索路径。在 `visited` 数组中标记当前位置(`startRow, startCol`)为已访问。调用 `recursivePathfinding` 函数，该函数会递归探索四个可能的移动方向：上、下、左、右。如果找到了一条从起点到终点的路径，`recursivePathfinding` 函数会返回 `true`，并在迷宫地图上标记路径上的位置为 `MAZE_PATH`，同时将这些位置压入 `path` 数据结构中，以便之后获取路径；

(3)释放内存：在路径搜索完成后，释放动态分配的 `visited` 数组的内存，以避免内存泄漏；

(4)返回路径搜索结果：返回 `true` 表示找到了从起点到终点的路径，否则返回 `false` 表示没有找到路径。



图 3.3.1.1.1 递归回溯搜索算法实现流程图

#### 3.3.1.2 递归回溯搜索算法核心代码

```
bool Maze::recursivePathfinding(int row, int col, bool** visited)
```

```

{
    if (row == targetRow && col == targetCol)
        return true;
    visited[row][col] = true;
    const int dr[] = { -1, 1, 0, 0 }, dc[] = { 0, 0, -1, 1 };
    for (int i = 0; i < 4; i++) {
        int newRow = row + dr[i], newCol = col + dc[i];
        if (isValid(newRow, newCol) && !visited[newRow][newCol] &&
mazeMap[newRow][newCol] == MAZE_BLANK && recursivePathfinding(newRow,
newCol, visited)) {
            mazeMap[newRow][newCol] = MAZE_PATH;
            path.push({ newRow, newCol });
            return true;
        }
    }
    return false;
}

bool Maze::recursiveBacktracking(void)
{
    bool** visited = new(std::nothrow) bool* [rows];
    if (visited == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    for (int i = 0; i < rows; i++) {
        visited[i] = new(std::nothrow) bool[cols];
        if (visited[i] == NULL) {
            std::cerr << "Error: Memory allocation failed." <<
std::endl;
            exit(MEMORY_ALLOCATION_ERROR);
        }
        for (int j = 0; j < cols; j++)
            visited[i][j] = false;
    }
    bool pathFound = recursivePathfinding(startRow, startCol,
visited);
    if (pathFound) {
        mazeMap[startRow][startCol] = MAZE_PATH;
        path.push({ startRow, startCol });
    }
    for (int i = 0; i < rows; i++)
        delete[] visited[i];
    delete[] visited;
}

```

```

        return pathFound;
    }

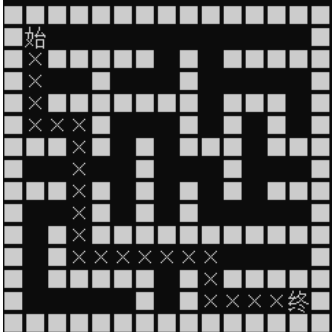
```

### 3.3.1.3 递归回溯搜索算法示例

```

>>> 迷宫寻路算法: [1]递归回溯搜索算法 [2]深度优先搜索(DFS)算法 [3]广度优先搜索(BFS)算法 [4]A*搜索算法
请选择迷宫寻路算法: [1]

```



```

>>> 迷宫路径 (寻路时长: 0.000025秒)
(1, 1) --> (2, 1) --> (3, 1) --> (4, 1) --> (5, 1) --> (5, 2) --> (5, 3) --> (6, 3) --> (7, 3) --> (8, 3) --> (
9, 3) --> (10, 3) --> (11, 3) --> (11, 4) --> (11, 5) --> (11, 6) --> (11, 7) --> (11, 8) --> (11, 9) --> (12,
9) --> (13, 9) --> (13, 10) --> (13, 11) --> (13, 12) --> (13, 13)

```

图 3.3.1.3.1 递归回溯搜索算法示例

## 3.3.2 深度优先搜索 (DFS) 算法

### 3.3.2.1 深度优先搜索 (DFS) 算法实现思路

执行深度优先搜索 (DFS) 算法的函数为 **Maze** 类的成员函数 **DFS**，其实现思路为：

(1) 内存分配和初始化：开始时，分配一个二维布尔数组 **visited**，用于跟踪迷宫中的每个位置是否已被访问。检查内存分配是否成功，如果失败则输出错误信息并退出程序。初始化 **visited** 数组，将所有位置标记为未访问状态；

(2) 初始化 DFS 路径：创建一个 DFS 路径栈 **DFSPath**，并将起始点 (**startRow**, **startCol**) 压入栈中。在迷宫地图中将起始点标记为 **MAZE\_PATH**，表示路径的起点；

(3) DFS 算法：进入一个循环，只要 DFS 路径栈不为空，就继续执行。获取当前栈顶的坐标 (**currRow**, **currCol**)。如果 (**currRow**, **currCol**) 是目标点 (**targetRow**, **targetCol**)，表示找到了从起点到终点的路径，进行以下操作：

- ①从 DFS 路径栈中弹出所有坐标，将它们压入路径记录栈 **path** 中；
- ②释放动态分配的 **visited** 数组的内存；
- ③返回 **true** 表示找到了路径；

(4) 尝试移动：如果 (**currRow**, **currCol**) 不是目标点，则尝试在四个可能的方向上进行移动，即上、下、左、右。对于每个方向，检查是否可以向该方向移动：

①如果可以移动（未越界、为 `MAZE_BLANK` 且未访问），则标记 `(newRow,newCol)` 为已访问，并将其坐标压入 DFS 路径栈，同时在迷宫地图上标记为 `MAZE_PATH`；

②如果成功移动，则设置 `found` 为 `true` 表示找到了可移动的方向。一旦找到一个可移动的方向，立即终止循环；

(5)回溯：如果没有找到可移动的方向（`found` 为 `false`），则需要回溯。这意味着从 DFS 路径栈中弹出当前坐标 `(currRow,currCol)`，将其标记 `MAZE_BLANK`，以便后续尝试其他方向；

(6)路径未找到：如果 DFS 路径栈为空且没有找到路径，表示在整个迷宫中没有从起点到终点的路径。释放动态分配的 `visited` 数组的内存。返回 `false` 表示未找到路径。

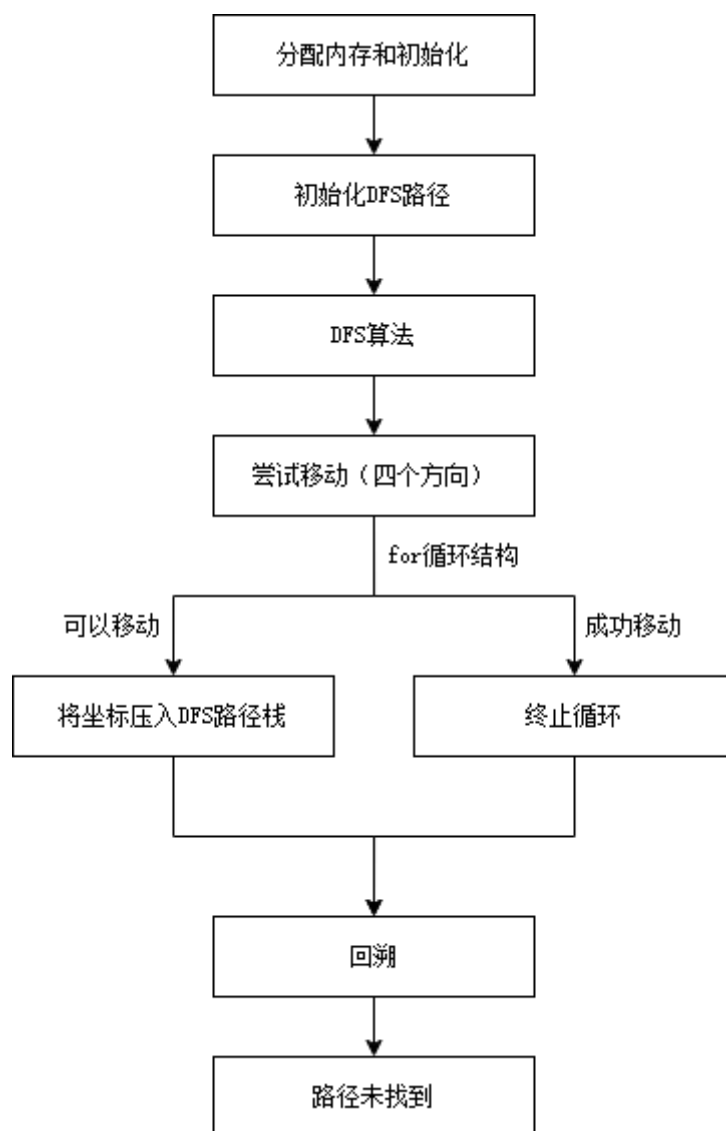


图 3.3.2.1.1 深度优先搜索 (DFS) 算法实现流程图

### 3.3.2.2 深度优先搜索 (DFS) 算法核心代码

```
bool Maze::DFS(void)
{
    /* Create a two dimensional array to mark whether each position
has been visited */
    bool** visited = new(std::nothrow) bool* [rows];
    if (visited == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    for (int i = 0; i < rows; i++) {
        visited[i] = new(std::nothrow) bool[cols];
        if (visited[i] == NULL) {
            std::cerr << "Error: Memory allocation failed." <<
std::endl;
            exit(MEMORY_ALLOCATION_ERROR);
        }
        for (int j = 0; j < cols; j++)
            visited[i][j] = false;
    }

    /* Initialize the stack with the starting point */
    Coordinate start{ startRow ,startCol };
    MyStack<Coordinate> DFSPath;
    DFSPath.push(start);
    mazeMap[startRow][startCol] = MAZE_PATH;

    /* DFS algorithm */
    while (!DFSPath.isEmpty()) {
        /* Get current point */
        Coordinate current;
        DFSPath.getTop(current);
        currRow = current.row;
        currCol = current.col;

        /* Found the target, the path is complete */
        if (currRow == targetRow && currCol == targetCol) {
            while (!DFSPath.isEmpty()) {
                Coordinate tempCoord;
                DFSPath.pop(tempCoord);
                path.push(tempCoord);
            }
            for (int i = 0; i < rows; i++)
                delete[] visited[i];
        }
    }
}
```

```

        delete[] visited;
        return true;
    }

    /* Try moving in each of the four directions */
    bool found = false;
    for (int dir = Up; dir <= Right; dir++) {
        int newRow = currRow, newCol = currCol;
        if (dir == Up)
            newRow--;
        else if (dir == Down)
            newRow++;
        else if (dir == Left)
            newCol--;
        else if (dir == Right)
            newCol++;
        if (isValid(newRow, newCol) && mazeMap[newRow][newCol] ==
MAZE_BLANK && !visited[newRow][newCol]) {
            visited[newRow][newCol] = true;
            Coordinate next;
            next.row = newRow;
            next.col = newCol;
            DFSPath.push(next);
            mazeMap[newRow][newCol] = MAZE_PATH;
            found = true;
            break;
        }
    }

    /* If there are no valid neighbors to move, backtrack */
    if (!found) {
        DFSPath.pop(current);
        mazeMap[current.row][current.col] = MAZE_BLANK;
    }
}

/* No path found */
for (int i = 0; i < rows; i++)
    delete[] visited[i];
delete[] visited;
return false;
}

```



```
>>> 迷宫寻路算法：[1]递归回溯搜索算法 [2]深度优先搜索(DFS)算法 [3]广度优先搜索(BFS)算法 [4]A*搜索算法
请选择迷宫寻路算法：[2]
```

```

  始
  x
 x
 x x x
   x
   x x x x x
     x
     x
     x
     x x x x x
       x
       x
       x x x x x
         x
         x
         x x
           x
           x x 终
```

```
>>> 迷宫路径（寻路时长：0.000041秒）

(1,1) --> (2,1) --> (3,1) --> (3,2) --> (3,3) --> (4,3) --> (5,3) --> (5,4) --> (5,5) --> (5,6) --> (5,7) --> (6,7) --> (7,7) --> (8,7) --> (9,7) --> (9,8) --> (9,9) --> (9,10) --> (9,11) --> (10,11) --> (11,11) --> (12,11) --> (13,11) --> (13,12) --> (13,13)
```

### 3.3.3 广度优先搜索(BFS)算法

执行广度优先搜索 (BFS) 算法的函数为 `Maze` 类的成员函数 `BFS`，其实现思路为：

- (1) 内存分配和初始化
  - ①首先，函数分配一个二维布尔数组 `visited`，用于标记迷宫中的每个位置是否已被访问。同时，分配一个二维坐标数组 `parent`，用于跟踪每个位置的父节点，帮助构建路径；
  - ②检查内存分配是否成功，如果失败则输出错误信息并退出程序；
  - ③初始化 `visited` 数组，将所有位置标记为未访问状态。初始化 `parent` 数组，将每个位置的父节点初始化为无效值；
- (2) 初始化 BFS 队列
  - ①创建一个 BFS 队列 `queue`，并将起始点(`startRow, startCol`)入队；
  - ②在 `visited` 数组中将起始点标记为已访问状态；
- (3) BFS 算法
  - ①进入一个循环，只要 BFS 队列不为空，就继续执行；
  - ②出队队列中的当前坐标(`current.row, current.col`)；
  - ③如果(`current.row, current.col`)是目标点(`targetRow, targetCol`)表示找到了从起点到终点的路径，进行以下操作；
  - ④从当前点开始回溯，找到起点，依次将路径坐标入栈 `path`；
  - ⑤同时，在迷宫地图上标记路径为 `MAZE_PATH`；

⑥释放动态分配的 **visited** 和 **parent** 数组的内存;

⑦返回 **true** 表示找到了路径;

(4)探索周围点

①如果(**current.row**, **current.col**)不是目标点, 对其四个周围点进行探索: 上、下、左、右;

②对于每个周围点, 检查是否可以向其移动。如果可以移动(未越界、为 **MAZE\_BLANK** 且未访问), 则标记(**newRow,newCol**)为已访问, 并将其父节点设置为当前点(**current.row,current.col**);

③将(**newRow,newCol**)入队, 表示要继续搜索这个点;

(5)路径未找到

如果 BFS 队列为空且未找到路径, 表示在整个迷宫中没有从起点到终点的路径。释放动态分配的 **visited** 和 **parent** 数组的内存。返回 **false** 表示未找到路径。

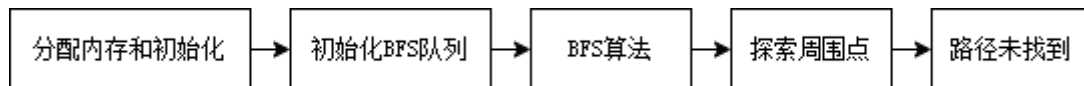


图 3.3.3.1.1 广度优先搜索(BFS)算法实现流程图

### 3.3.3.2 广度优先搜索(BFS)算法核心代码

```
bool Maze::BFS(void)
{
    /* Create a two dimensional array to mark whether each position
has been visited */
    bool** visited = new(std::nothrow) bool* [rows];
    if (visited == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    for (int i = 0; i < rows; i++) {
        visited[i] = new(std::nothrow) bool[cols];
        if (visited[i] == NULL) {
            std::cerr << "Error: Memory allocation failed." <<
std::endl;
            exit(MEMORY_ALLOCATION_ERROR);
        }
        for (int j = 0; j < cols; j++)
            visited[i][j] = false;
    }

    /* Create a two dimensional array to keep track of parent cells
*/
```

```

Coordinate** parent = new(std::nothrow) Coordinate * [rows];
if (parent == NULL) {
    std::cerr << "Error: Memory allocation failed." << std::endl;
    exit(MEMORY_ALLOCATION_ERROR);
}
for (int i = 0; i < rows; i++) {
    parent[i] = new(std::nothrow) Coordinate[cols];
    if (parent[i] == NULL) {
        std::cerr << "Error: Memory allocation failed." <<
std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    for (int j = 0; j < cols; j++)
        parent[i][j] = { -1,-1 };
}

/* Start from the initial cell and mark it as visited */
MyQueue<Coordinate> queue;
Coordinate start = { startRow, startCol };
visited[startRow][startCol] = true;
queue.enqueue(start);

/* BFS algorithm */
while (!queue.isEmpty()) {
    /* Dequeue a cell from the queue */
    Coordinate current;
    queue.dequeue(current);

    /* If we have reached the target, backtrack to find the path
*/
    if (current.row == targetRow && current.col == targetCol) {
        while (!(current.row == startRow && current.col ==
startCol)) {
            path.push(current);
            mazeMap[current.row][current.col] = MAZE_PATH;
            current = parent[current.row][current.col];
        }
        path.push({ startRow, startCol });
        mazeMap[startRow][startCol] = MAZE_PATH;
        for (int i = 0; i < rows; i++) {
            delete[] visited[i];
            delete[] parent[i];
        }
        delete[] visited;

```

```

        delete[] parent;
        return true;
    }

    /* Explore neighbors */
    const int dx[] = { -1, 1, 0, 0 }, dy[] = { 0, 0, -1, 1 };
    for (int i = 0; i < 4; i++) {
        int newRow = current.row + dx[i], newCol = current.col +
dy[i];

        if (isValid(newRow, newCol) && mazeMap[newRow][newCol] ==
MAZE_BLANK && !visited[newRow][newCol]) {
            visited[newRow][newCol] = true;
            parent[newRow][newCol] = current;
            queue.enqueue({ newRow, newCol });
        }
    }
}

/* No path found */
for (int i = 0; i < rows; i++) {
    delete[] visited[i];
    delete[] parent[i];
}
delete[] visited;
delete[] parent;
return false;
}

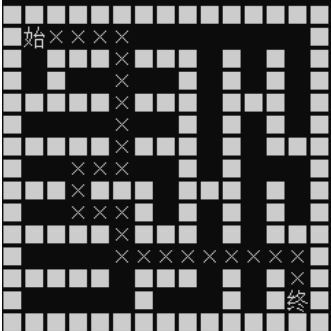
```

### 3.3.3.3 广度优先搜索(BFS)算法示例

```

>>> 迷宫寻路算法: [1]递归回溯搜索算法 [2]深度优先搜索(DFS)算法 [3]广度优先搜索(BFS)算法 [4]A*搜索算法
请选择迷宫寻路算法: [3]

```



```

>>> 迷宫路径 (寻路时长: 0.000054秒)
(1,1) --> (1,2) --> (1,3) --> (1,4) --> (1,5) --> (2,5) --> (3,5) --> (4,5) --> (5,5) --> (6,5) --> (
7,5) --> (7,4) --> (7,3) --> (8,3) --> (9,3) --> (9,4) --> (9,5) --> (10,5) --> (11,5) --> (11,6) -->
(11,7) --> (11,8) --> (11,9) --> (11,10) --> (11,11) --> (11,12) --> (11,13) --> (12,13) --> (13,13)

```

图 3.3.3.3.1 广度优先搜索(BFS)算法示例

### 3.3.4 A\*搜索算法

#### 3.3.4.1 A\*搜索算法实现思路

执行 A\*搜索算法的函数为 **Maze** 类的成员函数 **AStar**，其实现思路为：

(1) 内存分配和初始化

①首先，函数分配一个二维布尔数组 **visited**，用于标记迷宫中的每个位置是否已被访问。同时，分配一个二维坐标数组 **parent**，用于跟踪每个位置的父节点，帮助构建路径；

②检查内存分配是否成功，如果失败则输出错误信息并退出程序；

③初始化 **visited** 数组，将所有位置标记为未访问状态。初始化 **parent** 数组，将每个位置的父节点初始化为无效值；

(2) 定义估算总成本的比较函数

使用一个 **lambda** 函数 **compareF** 用于比较两个 A\*节点的估算总成本，以便构建优先级队列；

(3) 初始化起始节点和目标节点

创建 A\*节点 **startNode** 表示起始位置，以及 **targetNode** 表示目标位置；

(4) 使用优先级队列

创建一个优先级队列 **openSet**，用于存储搜索状态。添加起始节点 **startNode** 到 **openSet**。计算起始节点的估算总成本，该成本通常是从起点到终点的估算距离；

(5) A\*搜索算法

①进入一个循环，只要 **openSet** 不为空，就继续执行；

②从 **openSet** 中移除当前节点 **currentNode**，它是具有最小估算总成本的节点；

③如果 **currentNode** 是目标节点，表示找到了从起点到终点的路径，进行以下操作：

④从目标节点开始回溯，构建路径并将路径坐标入栈 **path**；

⑤同时，在迷宫地图上标记路径为 **MAZE\_PATH**；

⑥释放动态分配的 **visited** 和 **parent** 数组的内存；

⑦返回 **true** 表示找到了路径；

(6) 探索邻居节点

如果 **currentNode** 不是目标节点，探索其四个相邻节点（上、下、左、右）。对于每个邻居节点，检查是否可以向其移动：

①如果可以移动（未越界、为 **MAZE\_BLANK** 且未访问），则创建邻居节点 **neighborNode**，计算其估算总成本，并将其添加到 **openSet** 中；

②如果 **neighborNode** 已经存在于 **openSet**，则比较其估算总成本，选择

较小的值；

③更新 **parent** 数组以跟踪邻居节点的父节点；

(7) 路径未找到

如果 **openSet** 为空且未找到路径，表示在整个迷宫中没有从起点到终点的路径。释放动态分配的 **visited** 和 **parent** 数组的内存。返回 **false** 表示未找到路径。

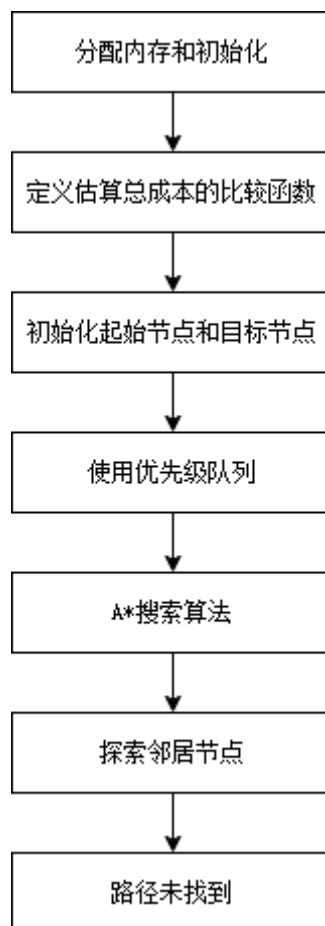


图 3.3.4.1.1 A\*搜索算法实现流程图

### 3.3.4.2 A\*搜索算法核心代码

```
bool Maze::AStar(void)
{
    /* Create a two dimensional array to mark whether each position
    has been visited */
    bool** visited = new(std::nothrow) bool* [rows];
    if (visited == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    for (int i = 0; i < rows; i++) {
```

```

        visited[i] = new(std::nothrow) bool[cols];
        if (visited[i] == NULL) {
            std::cerr << "Error: Memory allocation failed." <<
std::endl;
            exit(MEMORY_ALLOCATION_ERROR);
        }
        for (int j = 0; j < cols; j++)
            visited[i][j] = false;
    }

    /* Create a two dimensional array to keep track of parent cells
    */
    Coordinate** parent = new(std::nothrow) Coordinate * [rows];
    if (parent == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    for (int i = 0; i < rows; i++) {
        parent[i] = new(std::nothrow) Coordinate[cols];
        if (parent[i] == NULL) {
            std::cerr << "Error: Memory allocation failed." <<
std::endl;
            exit(MEMORY_ALLOCATION_ERROR);
        }
        for (int j = 0; j < cols; j++)
            parent[i][j] = { -1,-1 };
    }

    /* Define a lambda function for comparing the estimated total cost
    of two states */
    auto compareF = [](const AStarNode& a, const AStarNode& b) { return
a.totalCost > b.totalCost; };

    /* Initialize the start node and the target node */
    AStarNode startNode = { startRow, startCol, 0, 0 };
    AStarNode targetNode = { targetRow, targetCol, 0, 0 };

    /* Use a priority queue to store the search states */
    MyPriorityQueue<AStarNode> openSet(rows * cols);

    /* Add the start node to the openSet */
    startNode.totalCost = abs(targetNode.row - startNode.row) +
abs(targetNode.col - startNode.col);
    openSet.insert(startNode);

```

```

    /* A-Star algorithm */
    while (!openSet.isEmpty()) {
        /* If the current node is the target node, it means the path
has been found */
        AStarNode currentNode;
        openSet.remove(currentNode);
        if (currentNode.row == targetNode.row && currentNode.col ==
targetNode.col) {
            Coordinate    currentCoord    =    {    currentNode.row,
currentNode.col };
            while (!(currentCoord.row == startRow && currentCoord.col
== startCol)) {
                path.push(currentCoord);
                mazeMap[currentCoord.row][currentCoord.col]          =
MAZE_PATH;
                currentCoord          =
parent[currentCoord.row][currentCoord.col];
            }
            path.push({ startRow, startCol });
            mazeMap[startRow][startCol] = MAZE_PATH;
            for (int i = 0; i < rows; i++) {
                delete[] visited[i];
                delete[] parent[i];
            }
            delete[] visited;
            delete[] parent;
            return true;
        }
        visited[currentNode.row][currentNode.col] = true;

        /* Represent offsets for the four directions */
        const int dr[] = { -1, 1, 0, 0 }, dc[] = { 0, 0, -1, 1 };
        for (int i = 0; i < 4; i++) {
            int    newRow    =    currentNode.row    +    dr[i],    newCol    =
currentNode.col + dc[i];
            if (isValid(newRow, newCol) && mazeMap[newRow][newCol] ==
MAZE_BLANK && !visited[newRow][newCol]) {
                AStarNode    neighborNode    =    {    newRow,    newCol,
currentNode.accumuCost + 1, 0 };
                neighborNode.totalCost    =    neighborNode.accumuCost    +
abs(targetNode.row - newRow) + abs(targetNode.col - newCol);
                if (!openSet.insert(neighborNode)) {
                    openSet.remove(neighborNode);
                }
            }
        }
    }
}

```



```

        if (neighborNode.totalCost < currentNode.totalCost)
            openSet.insert(neighborNode);
    }
    parent[newRow][newCol] = { currentNode.row,
currentNode.col };
    }
}

/* No path found */
for (int i = 0; i < rows; i++) {
    delete[] visited[i];
    delete[] parent[i];
}
delete[] visited;
delete[] parent;
return false;
}

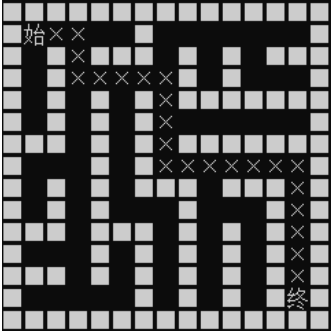
```

### 3.3.4.3 A\*搜索算法示例

```

>>> 迷宫寻路算法: [1]递归回溯搜索算法 [2]深度优先搜索 (DFS)算法 [3]广度优先搜索 (BFS)算法 [4]A*搜索算法
请选择迷宫寻路算法: [4]

```



```

>>> 迷宫路径 (寻路时长: 0.000050秒)
(1, 1) --> (1, 2) --> (1, 3) --> (2, 3) --> (3, 3) --> (3, 4) --> (3, 5) --> (3, 6) --> (3, 7) --> (4, 7) --> (
5, 7) --> (6, 7) --> (7, 7) --> (7, 8) --> (7, 9) --> (7, 10) --> (7, 11) --> (7, 12) --> (7, 13) --> (8, 13) --
-> (9, 13) --> (10, 13) --> (11, 13) --> (12, 13) --> (13, 13)

```

图 3.3.4.3.1 A\*搜索算法示例

## 3.4 界面设计与可视化功能的实现

### 3.4.1 界面设计与可视化功能实现思路

执行界面设计与可视化功能的函数为 **Maze** 类的成员函数 **output**，其实现思路为使用两个嵌套的 **for** 循环，遍历整个迷宫的行和列，输出“起”“终”“ ” “■”“×”分别表示起点、终点、通道、墙壁和路径。

### 3.4.2 界面设计与可视化功能核心代码

```
void Maze::output(void)
{
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (i == startRow && j == startCol)
                std::cout << "始";
            else if (i == targetRow && j == targetCol)
                std::cout << "终";
            else if (mazeMap[i][j] == MAZE_BLANK)
                std::cout << " ";
            else if (mazeMap[i][j] == MAZE_WALL)
                std::cout << "■";
            else if (mazeMap[i][j] == MAZE_PATH)
                std::cout << "x";
        }
        std::cout << std::endl;
    }
}
```

### 3.4.3 界面设计与可视化功能示例

界面设计与可视化功能示例见图 3.1.3.1、图 3.2.3.1、图 3.3.1.3.1、图 3.3.2.3.1、图 3.3.3.3.1、图 3.3.4.3.1。

## 3.5 异常处理功能的实现

### 3.5.1 动态内存申请失败的异常处理

在进行 MyLinkNode 类等的动态内存申请时，程序使用 `new(std::nothrow)` 来尝试分配内存。`new(std::nothrow)` 在分配内存失败时不会引发异常，而是返回一个空指针（NULL 或 `nullptr`），代码检查指针是否为空指针，如果为空指针，意味着内存分配失败，这时程序将执行以下操作：

(1) 向标准错误流 `std::cerr` 输出一条错误消息 "Error: Memory allocation failed."，指出内存分配失败；

(2) 调用 `exit` 函数，返回错误码 `MEMORY_ALLOCATION_ERROR`（通过宏定义方式定义为-1），用于指示内存分配错误，并导致程序退出。

下面是动态内存申请的异常处理的一个代码示例：

```
template <typename Type>
MyList<Type>::MyList()
{
    first = new(std::nothrow) MyLinkNode<Type>;
```

```

        if (first == NULL) {
            std::cerr << "Error: Memory allocation failed." << std::endl;
            exit(MEMORY_ALLOCATION_ERROR);
        }
        last = first;
    }
}

```

### 3.5.2 MyStack 类栈空的异常处理

在调用 MyStack 类的 pop 和 getTop 等成员函数时,如果栈为空(isEmpty() 返回 true), 则函数返回 false, 表示栈不包含任何元素。

### 3.5.3 MyQueue 类队列空的异常处理

在调用 MyQueue 类的 deQueue 和 getFront 等成员函数时, 如果队列为空 (isEmpty()返回 true), 则函数返回 false, 表示队列不包含任何元素。

### 3.5.4 MyPriorityQueue 类队列满和空的异常处理

在调用 MyPriorityQueue 类的 insert 等成员函数时, 如果队列已满 (isFull()返回 true), 则函数返回 false, 表示无法插入新元素。

在调用 MyPriorityQueue 类的 remove 和 getFront 等成员函数时, 如果队列为空 (isEmpty()返回 true), 则函数返回 false, 表示无法执行删除或获取队头元素的操作。

### 3.5.5 Maze 类索引越界的异常处理

在调用 Maze 类的 popList 等成员函数时, 如果索引越界 (index < 0 || index >= mazePointCount), 则向标准错误流 std::cerr 输出一条错误消息 "Error: Invalid index for erase operation.", 调用 exit 函数, 返回错误码 INVALID\_INDEX\_ERROR (通过宏定义方式定义为-2), 用于指示索引越界异常, 并导致程序退出。

### 3.5.6 输入非法的异常处理

#### 3.5.6.1 迷宫地图大小输入非法的异常处理

程序通过调用 inputOddInteger 函数输入迷宫地图大小 (行数和列数)。inputOddInteger 函数用于获取用户输入的奇整数, 同时限制输入必须在指定的范围内, 函数的代码如下:

```

int inputOddInteger(int lowerLimit, int upperLimit, const char* prompt)
{
    while (true) {

```

```

        std::cout << "请输入" << prompt << " (奇数) [整数范围: " <<
lowerLimit << "~" << upperLimit << "]: ";
        double tempInput;
        std::cin >> tempInput;
        if (std::cin.good() && tempInput == static_cast<int>(tempInput)
&& tempInput >= lowerLimit && tempInput <= upperLimit &&
static_cast<int>(tempInput) % 2) {
            std::cin.clear();
            std::cin.ignore(INT_MAX, '\n');
            return static_cast<int>(tempInput);
        }
        else {
            std::cerr << std::endl << ">>> " << prompt << "输入不合法,
请重新输入" << prompt << "! " << std::endl << std::endl;
            std::cin.clear();
            std::cin.ignore(INT_MAX, '\n');
        }
    }
}

```

**inputOddInteger** 函数对输入非法的情况进行了处理，代码具体执行逻辑如下：

- (1) 进入一个无限循环，它会一直运行直到用户提供有效的输入；
- (2) 用户的输入被读取到 **tempInput** 变量中，这里采用 **double** 类型来接收输入以便后续检查；
- (3) 进行输入验证：**std::cin.good()**检查输入流的状态是否正常，确保没有发生数据类型输入错误，**tempInput==static\_cast<int>(tempInput)**检查用户输入是否为整数，通过将其转换为整数再比较，**tempInput>=lowerLimit**和 **tempInput<=upperLimit** 确保输入在指定的范围内，并根据 **static\_cast<int>(tempInput)%2** 输入的整数为奇数；
- (4) 合法输入处理：如果用户提供了合法的输入，函数会清除输入流的错误状态，丢弃输入缓冲区中的任何剩余内容，然后返回转换后的整数值；
- (5) 非法输入处理：如果用户提供的输入不合法，函数会输出错误消息，清除输入流的错误状态，丢弃输入缓冲区中的内容，并继续循环以等待用户提供合法的输入。

### 3.5.6.2 迷宫寻路算法选择输入非法的异常处理

迷宫寻路算法选择输入非法的异常处理通过如下代码实现：

```

int selectOptn(void)
{
    std::cout << std::endl << ">>> 迷宫寻路算法: [1]递归回溯搜索算法 [2]

```

```

深度优先搜索(DFS)算法 [3]广度优先搜索(BFS)算法 [4]A*搜索算法" << std::endl;
std::cout << std::endl << "请选择迷宫寻路算法: ";
char optn;
while (true) {
    optn = _getch();
    if (optn == 0 || optn == -32)
        optn = _getch();
    else if (optn >= '1' && optn <= '4') {
        std::cout << "[" << optn << "]" << std::endl << std::endl;
        return optn - '0';
    }
}
}

```

这段代码会一直等待用户输入，只有当用户输入有效的数字字符（'1'到'4'）时，才会返回该数字的整数值，表示用户选择的操作选项。如果用户输入无效字符，循环会继续等待用户提供有效的输入。这段代码的具体执行逻辑如下：

- (1) 显示提示信息，其中包含选项[1]到[4]；
- (2) 进入一个无限循环，以等待用户输入；
- (3) 用户输入的字符会被\_getch()函数获取。这个函数通常用于在控制台应用程序中获取单个字符而不显示在屏幕上。用户的输入存储在变量 optn 中；
- (4) 代码检查用户输入是否是数字字符（'1'到'4'）或特殊的控制字符。如果用户按下非数字字符或特殊控制字符，它将不执行下面的操作；
- (5) 如果用户输入是数字字符（'1'到'4'），它会在屏幕上显示用户输入的数字字符，并返回对应的整数值；
- (6) 如果用户输入的不是数字字符（'1'到'4'），代码将保持在循环中，继续等待有效的输入，这确保了只有在用户输入正确的选项时才会退出循环。

## 4 项目测试

### 4.1 项目主体架构测试

#### 4.1.1 输入迷宫地图大小功能测试

分别输入超过上下限的整数、偶数、浮点数、字符、字符串，可以验证程序对输入非法的情况进行了处理。

当输入合法时，程序继续运行。

输入测试依次进行两次，第一次进行输入迷宫地图行数功能测试，第二次进行输入迷宫地图列数功能测试。

```

请输入迷宫地图行数（奇数）[整数范围：7~99]：6
>>> 迷宫地图行数输入不合法，请重新输入迷宫地图行数!
请输入迷宫地图行数（奇数）[整数范围：7~99]：100
>>> 迷宫地图行数输入不合法，请重新输入迷宫地图行数!
请输入迷宫地图行数（奇数）[整数范围：7~99]：8
>>> 迷宫地图行数输入不合法，请重新输入迷宫地图行数!
请输入迷宫地图行数（奇数）[整数范围：7~99]：5.5
>>> 迷宫地图行数输入不合法，请重新输入迷宫地图行数!
请输入迷宫地图行数（奇数）[整数范围：7~99]：a
>>> 迷宫地图行数输入不合法，请重新输入迷宫地图行数!
请输入迷宫地图行数（奇数）[整数范围：7~99]：abc
>>> 迷宫地图行数输入不合法，请重新输入迷宫地图行数!
请输入迷宫地图行数（奇数）[整数范围：7~99]：7
请输入迷宫地图列数（奇数）[整数范围：7~99]：6
>>> 迷宫地图列数输入不合法，请重新输入迷宫地图列数!
请输入迷宫地图列数（奇数）[整数范围：7~99]：100
>>> 迷宫地图列数输入不合法，请重新输入迷宫地图列数!
请输入迷宫地图列数（奇数）[整数范围：7~99]：8
>>> 迷宫地图列数输入不合法，请重新输入迷宫地图列数!
请输入迷宫地图列数（奇数）[整数范围：7~99]：5.5
>>> 迷宫地图列数输入不合法，请重新输入迷宫地图列数!
请输入迷宫地图列数（奇数）[整数范围：7~99]：a
>>> 迷宫地图列数输入不合法，请重新输入迷宫地图列数!
请输入迷宫地图列数（奇数）[整数范围：7~99]：abc
>>> 迷宫地图列数输入不合法，请重新输入迷宫地图列数!
请输入迷宫地图列数（奇数）[整数范围：7~99]：7

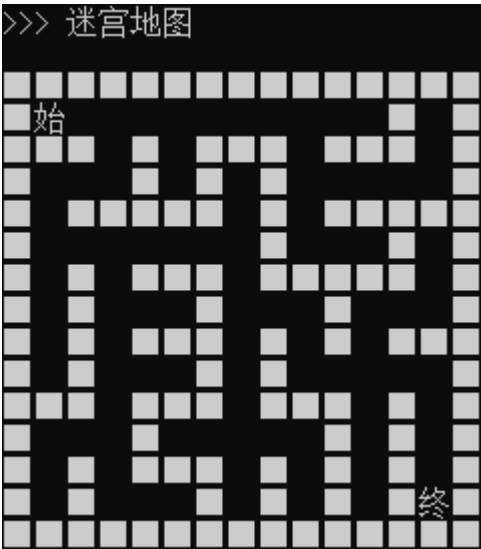
```

图 4.1.1.1 输入迷宫地图大小功能测试

#### 4.1.2 界面设计与可视化功能测试

Maze 类的成员函数 output 使用两个嵌套的 for 循环，遍历整个迷宫的行

和列，输出“起”“终”“ ”“■”“×”分别表示起点、终点、通道、墙壁和路径。



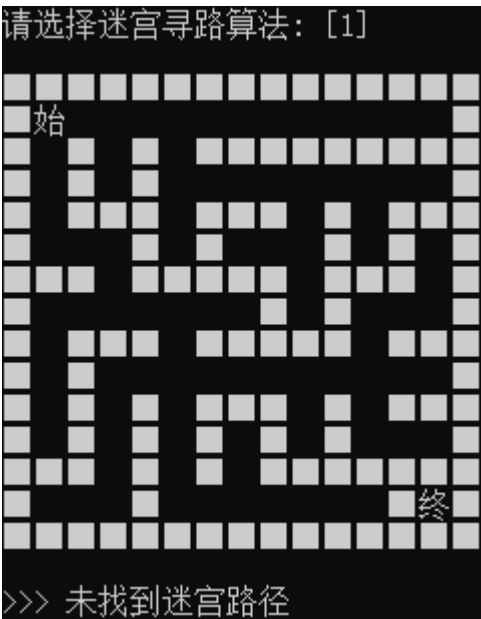
4.1.2.1 界面设计与可视化功能测试（无路径输出）

## 4.2 迷宫寻路算法功能测试

### 4.2.1 递归回溯搜索算法测试

当递归回溯搜索算法找到一条路径时，输出迷宫路径地图并打印迷宫路径（见图 3.3.1.3.1）。

当递归回溯搜索算法未找到路径时，输出提示信息“为找到迷宫路径”。

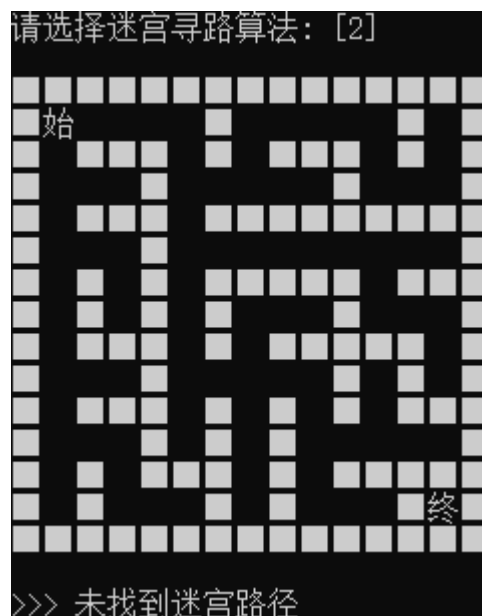


4.2.1.1 递归回溯搜索算法测试

### 4.2.2 深度优先搜索 (DFS) 算法测试

当深度优先搜索 (DFS) 算法找到一条路径时，输出迷宫路径地图并打印迷宫路径（见图 3.3.2.3.1）。

当深度优先搜索 (DFS) 算法未找到路径时，输出提示信息“为找到迷宫路径”。

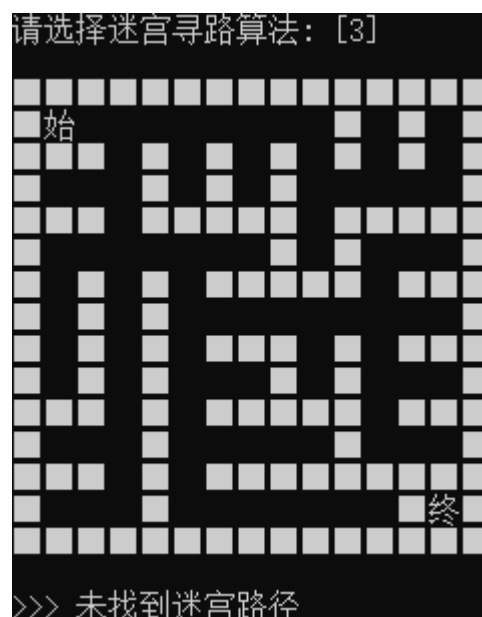


4.2.2.1 深度优先搜索 (DFS) 算法测试

### 4.2.3 广度优先搜索 (BFS) 算法测试

当广度优先搜索 (BFS) 算法找到一条路径时，输出迷宫路径地图并打印迷宫路径（见图 3.3.3.3.1）。

当广度优先搜索 (BFS) 算法未找到路径时，输出提示信息“为找到迷宫路径”。



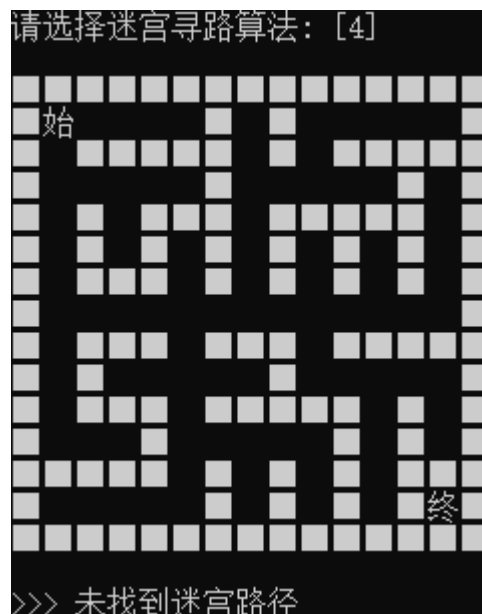
4.2.3.1 广度优先搜索 (BFS) 算法测试



#### 4.2.4 A\*搜索算法测试

当 A\*搜索算法找到一条路径时，输出迷宫路径地图并打印迷宫路径（见图 3.3.4.3.1）。

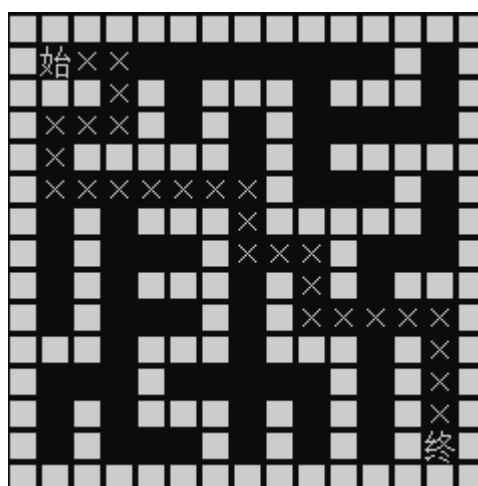
当 A\*搜索算法未找到路径时，输出提示信息“为找到迷宫路径”。



4.2.4.1 A\*搜索算法测试

#### 4.3 界面设计与可视化功能测试

Maze 类的成员函数 output 使用两个嵌套的 for 循环，遍历整个迷宫的行和列，输出“起”“终”“ ”“■”“×”分别表示起点、终点、通道、墙壁和路径。



4.3.1 界面设计与可视化功能测试（有路径输出）

## 4.4 退出程序功能测试

为避免直接运行可执行文件在输入完成后会发生闪退的情况，本程序使用如下代码，创建了一个无限循环，等待用户按下回车键，以便退出程序。避免结果输出结束后程序迅速退出的情况。

```
/* Wait for enter to quit */
std::cout << "Press Enter to Quit" << std::endl;
while (_getch() != '\r')
    continue;
```

## 5 集成开发环境与编译运行环境

Windows 系统：Windows 11 x64

Windows 集成开发环境：Microsoft Visual Studio 2022 (Release 模式)

Windows 编译运行环境：本项目适用于 x86 架构和 x64 架构

Linux 系统：CentOS 7 x64

Linux 编译命令：

```
g++ '/root/桌面/Share_Folder/maze_game.cpp' -std=c++11 -o '/root/桌面/Share_Folder/maze_game' -lncurses
```

Linux 运行命令：

```
'/root/桌面/Share_Folder/maze_game'
```

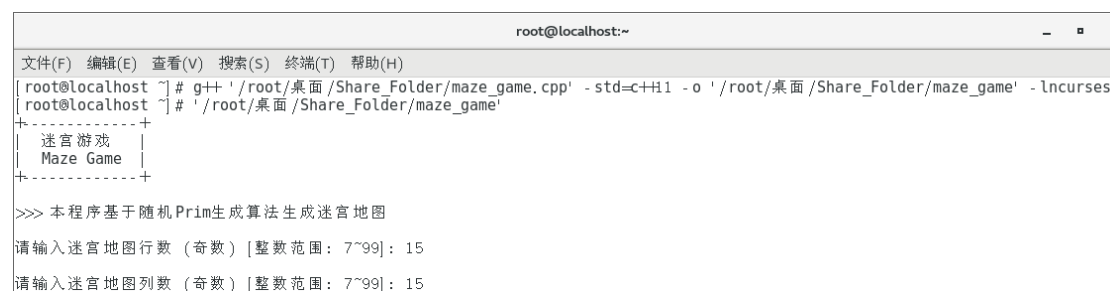


图 5.1 Linux 环境程序运行示例

本项目使用条件编译解决 Windows 系统和 Linux 系统编译环境的差异，示例代码如下。

```
#ifdef _WIN32
#include <conio.h>
#elif __linux__
#include <ncurses.h>
#endif
```