项目说明文档

数据结构课程设计

——勇闯迷宫游戏

作 者 姓 名： 张翔

学 号： 2352985

指 导 教 师： 张颖

学院、 专业： 计算机科学与技术学院 软件工程

同济大学

Tongji University

二〇二四 年 十二 月 七 日

# 1项目分析

## 1.1项目背景分析

迷宫问题源自经典的算法和数据结构研究场景，广泛应用于路径规划、机器人导航和游戏开发等领域。它模拟了在有限空间内解决复杂问题的过程，展示了回溯法在搜索问题中的实际应用。通过设计勇闯迷宫的游戏项目，可以帮助用户理解数据结构的组织方式和回溯算法的基本原理，同时培养解决问题的逻辑思维能力，为后续深入学习算法和系统开发奠定基础。

## 1.2项目需求分析

本项目需要构建一个迷宫环境，包含入口和出口，并在其中随机生成障碍物。骑士需要从入口出发，基于回溯法探索通往出口的路径。系统需能够记录和显示搜索路径，并在找到出口时终止搜索。此外，程序应能有效处理迷宫中的所有点，以避免重复访问，并能够在所有路径探索失败时提示无解的结果，从而满足模拟复杂路径搜索的需求。

## 1.3项目功能分析

(1)迷宫初始化：根据用户输入的迷宫大小动态生成一个包含障碍物的迷宫，并设置入口和出口的位置。

(2)路径搜索：实现回溯算法，从入口出发，按照设定的优先级逐步探索可行路径，记录已访问点并判断当前路径的可行性。

(3)路径显示：在成功找到路径后，显示完整的通路信息，将路径标记为显著状态；若无解，则提示无解信息。

(4)错误处理：在动态内存分配失败或非法输入时，提供友好的错误提示，确保程序稳定性。

# 2项目设计

## 2.1 结构体和类设计

### 2.1.1 MyLinkNode结构体的设计

2.1.1.1概述

MyLinkNode是一个模板结构体，表示链表节点。它包含两个成员: 存储节点的数据和指向下一个节点的指针，它定义了两个构造函数，一个为默认，另一个为有参数输入。

2.1.1.2类定义

template <typename Type>

struct MyLinkNode

{

Type data;

MyLinkNode<Type>\* link;

MyLinkNode(MyLinkNode<Type>\* ptr = nullptr) : link(ptr) {}

MyLinkNode(const Type& item, MyLinkNode<Type>\* ptr = nullptr) : data(item), link(ptr) {}

};

### 2.1.2 MyQueue类的设计

2.1.2.1概述

MyQueue是一个基于链表实现的模板队列类，包含front和rear指针分别指向队列的头部和尾部，count用于记录队列中元素的个数。该类提供了基本的队列操作，包括构造函数和析构函数。通过isEmpty()判断队列是否为空，Size()返回队列中元素的数量。enQueue()用于将元素加入队尾，deQueue()从队头移除并返回元素，getHead()获取队头元素。

2.1.2.2类定义

template <typename Type>

class MyQueue

{

private:

MyLinkNode<Type>\* front;

MyLinkNode<Type>\* rear;

int count;

public:

MyQueue() : front(nullptr), rear(nullptr), count(0) {}

~MyQueue() { makeEmpty(); }

bool isEmpty() const;

void makeEmpty();

int Size()const;

void enQueue(const Type& item);

bool deQueue(Type& item);

bool getHead(Type& item);

};

### 2.1.3 MyStack类设计

2.1.1.1概述

MyStack类实现了一个基于链表的栈数据结构。该类包含私有成员topNode，用于指向栈顶元素的节点，count用于记录栈中元素的数量max\_size限制栈的最大容量。构造函数提供了默认栈大小为100的选项，也允许通过指定大小来创建栈。析构函数会在销毁栈对象时清空栈内容。主要的成员函数包括：isEmpty()检查栈是否为空，makeEmpty()清空栈，Size()获取栈中元素数量，Push()向栈中压入元素，Pop()从栈中弹出元素，getTop()获取栈顶元素的值。

2.1.1.2类定义

template <typename Type>

class MyStack

{

private:

MyLinkNode<Type>\* topNode;

int count;

int max\_size;

public:

MyStack() : topNode(nullptr), count(0), max\_size(100) {}

MyStack(int size) : topNode(nullptr), count(0), max\_size(size) {}

~MyStack() { makeEmpty(); }

bool isEmpty() const;

void makeEmpty();

int Size() const;

bool Push(Type& item);

bool Pop(Type& item);

bool getTop(Type& item);

};

### 2.1.4 Maze类设计

2.1.4.1概述

Maze类是一个用于表示和操作迷宫的结构，支持迷宫的生成与路径搜索。该类包括迷宫的行列数、起点和终点坐标等基本属性，以及当前遍历位置和迷宫格点的状态管理。

Maze使用嵌套的 MazePoint 结构来存储每个迷宫单元的信息，并提供一个动态分配的二维数组 maze 来表示迷宫布局。它支持迷宫的自动生成功能，提供了generateMaze方法用于随机生成迷宫结构，并可以标记邻近的墙体。

路径搜索方面，Maze 实现了深度优先搜索 (DFS) 和广度优先搜索 (BFS) 方法，用于判断从入口到出口是否存在通路并输出路径。此外，类内包含辅助函数如isValid用于判断位置合法性，以及 pushList和popList操作迷宫点列表，便于管理路径和墙体。整体设计适用于迷宫相关的生成与求解问题。

2.1.4.2类定义

class Maze

{

private:

int rows;

int cols;

int startRow;

int startCol;

int targetRow;

int targetCol;

int currRow;

int currCol;

struct MazePoint;

MazePoint\*\* maze;

MazePoint\* mazePointList;

int mazePointListCount;

bool pushList(const MazePoint point);

bool popList(int index);

public:

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

~Maze();

bool isValid(int row, int col) const;

void generateMaze();

void generateMaze(int row, int col);

void findAdjacentWalls();

void ouputMaze();

bool DFS();

bool BFS();

};

## 2.3项目主体架构设计

|  |
| --- |
|  |

2.3.1项目主题架构流程图

# 3项目功能实现

## 3.1项目主体架构实现

### 3.1.1项目主题架构实现思路

这段代码的具体实现思路是：

首先通过main函数启动程序，并设置随机数种子以确保迷宫的随机性。然后进入mazeGame函数，用户输入迷宫的行列数，并通过Maze类创建迷宫对象，使用递归的 generateMaze 函数生成迷宫。接着输出生成的迷宫，并使用深度优先搜索（DFS）算法寻找路径并显示。然后清除DFS路径，使用广度优先搜索（BFS）算法重新计算路径并显示。最后，程序退出，完成游戏。

### 3.1.2项目主体架构核心代码

void mazeGame()

{

std::cout << "=== 勇闯迷宫游戏 ===" << std::endl << std::endl;

int \_rows = inputOdd(MIN\_ROWCOL, MAX\_ROWCOL, "行数");

int \_cols = inputOdd(MIN\_ROWCOL, MAX\_ROWCOL, "列数");

Maze maze(\_rows, \_cols, START\_ROW, START\_COL, \_rows - 2, \_cols - 2);

maze.generateMaze(START\_ROW, START\_COL);

// 输出生成的迷宫

std::cout << std::endl << ">>> 生成的迷宫如下: " << std::endl;

maze.ouputMaze();

// DFS算法生成路径

std::cout << std::endl << ">>> DFS算法生成的路径为: " << std::endl;

maze.DFS();

// 清除上个算法产生的路径

maze.clearPaths();

// BFS算法生成路径

std::cout << std::endl << ">>> BFS算法生成的路径为: " << std::endl;

maze.BFS();

}

int main()

{

// 设置随机数种子

srand((unsigned int)(time(0)));

// 进入迷宫游戏

mazeGame();

// 退出程序

return 0;

}

## 3.2迷宫生成算法实现

### 3.2.1迷宫生成算法实现思路

这段代码通过深度优先的方式生成迷宫,具体实现思路如下:

(1)定义了一个包含四个方向（上、下、左、右）的方向数组，并随机打乱方向顺序。

(2)从当前给定的起始位置（row 和 col）出发，依次尝试四个方向。对于每个方向，根据当前方向计算新的位置。如果新的位置在迷宫范围内且未被访问（即是墙壁），则通过设置当前节点与新节点之间的墙壁为通路，从而打通一条路径，并递归地对新位置进行相同的操作。

(3)递归继续进行，直到所有可能的路径都被探索到。通过这种方式，迷宫的每个通路都被打通，而墙壁仍然保留在其他未访问的区域。

### 3.2.2迷宫生成算法核心代码

void Maze::generateMaze(int row, int col)

{

Direction directions[4] = { Up,Down,Left,Right };

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

int r = rand() % 4;

Direction temp = directions[i];

directions[i] = directions[r];

directions[r] = temp;

}

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

int newRow = row;

int newCol = col;

switch (directions[i]) {

case Up:

newRow -= 2;

break;

case Down:

newRow += 2;

break;

case Left:

newCol -= 2;

break;

case Right:

newCol += 2;

break;

}

if (isValid(newRow, newCol) && maze[newRow][newCol].isWall) {

maze[(row + newRow) / 2][(col + newCol) / 2].isWall = false;

maze[newRow][newCol].isWall = false;

generateMaze(newRow, newCol);

}

}

}

## 3.3迷宫寻路算法实现

### 3.3.1广度优先算法的实现

3.3.1.1广度优先算法实现思路

这段代码的具体实现思路如下:

首先，通过动态分配二维数组初始化访问矩阵（visit）和父节点矩阵（parent），用于记录节点访问状态和路径的父子关系。将起点加入队列并标记为已访问，然后进入主循环：每次从队列中取出一个节点，判断是否到达目标位置；若到达，则沿父节点矩阵回溯重建路径，并标记路径上的节点。若未到达，则检查当前节点的四个方向，筛选出未访问且非墙体的相邻节点，将其加入队列并更新访问状态及父节点矩阵。

循环结束后，若队列为空但未找到目标，返回失败结果。最后释放所有动态分配的内存，确保程序稳定性和资源回收。

3.3.1.2广度优先算法核心代码

bool Maze::BFS()

{

// 初始化访问矩阵和父节点矩阵

bool\*\* visit = new (std::nothrow) bool\* [rows];

Coordinate\*\* parent = new (std::nothrow) Coordinate \* [rows];

if (visit == nullptr || parent == nullptr) {

cerr << "Error: Memory allocation failed." << endl;

exit(-1);

}

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

visit[i] = new (std::nothrow) bool[cols];

parent[i] = new (std::nothrow) Coordinate[cols];

if (visit[i] == nullptr || parent[i] == nullptr) {

cerr << "Error: Memory allocation failed." << endl;

exit(-1);

}

for (int j = 0; j < cols; j++) {

visit[i][j] = false;

parent[i][j] = { -1, -1 }; // 初始化父节点为无效坐标

}

}

// 使用自定义队列初始化和路径记录

MyQueue<Coordinate> BFS\_Queue;

Coordinate start{ startRow, startCol };

BFS\_Queue.enqueue(start);

visit[startRow][startCol] = true;

while (!BFS\_Queue.isEmpty()) {

Coordinate current;

BFS\_Queue.dequeue(current);

// 判断是否到达目标位置

if (current.row == targetRow && current.col == targetCol) {

// 重建路径

Coordinate pathPoint = current;

while (pathPoint.row != startRow || pathPoint.col != startCol) {

maze[pathPoint.row][pathPoint.col].isPath = true;

pathPoint = parent[pathPoint.row][pathPoint.col];

}

maze[startRow][startCol].isPath = true;

// 释放内存

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

delete[] visit[i];

delete[] parent[i];

}

delete[] visit;

delete[] parent;

ouputMaze();

return true;

}

// 遍历当前节点的邻接节点

static const int directions[4][2] = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

for (const auto& dir : directions) {

int newRow = current.row + dir[0];

int newCol = current.col + dir[1];

// 确保新位置在范围内且未访问且不是墙

if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols &&

!visit[newRow][newCol] && !maze[newRow][newCol].isWall) {

Coordinate next{ newRow, newCol };

BFS\_Queue.enqueue(next);

visit[newRow][newCol] = true;

parent[newRow][newCol] = current; // 记录父节点

}

}

}

// 如果队列为空且未找到目标，释放内存

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

delete[] visit[i];

delete[] parent[i];

}

delete[] visit;

delete[] parent;

return false; // 未找到目标

}

### 3.3.2深度优先算法的实现

3.3.2.1深度优先算法实现思路

这段代码的具体实现思路如下:

首先，通过动态分配内存创建一个访问矩阵 visit 来记录每个位置是否已被访问，并将起点入栈。算法在栈不为空时循环进行，每次从栈顶取出一个节点作为当前探索位置，并标记为已访问。如果当前节点为目标节点，则回溯栈中的路径并标记为路径，输出迷宫。如果不是目标节点，算法将探索当前节点的四个邻居（上、下、左、右），对于未访问且不是墙壁的邻居，入栈继续深入。如果所有邻居都不可达，则回溯弹出栈顶节点，尝试新的路径。若栈为空仍未找到目标，则返回失败。最后，无论是否找到路径，都需要释放动态分配的内存。

3.3.2.2深度优先算法核心代码

bool Maze::DFS()

{

bool\*\* visit = new (nothrow)bool\* [rows];

if (visit == nullptr) {

cerr << "Error: Memory allocation failed." << endl;

exit(-1);

}

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

visit[i] = new(nothrow) bool[cols];

if (visit[i] == nullptr) {

cerr << "Error: Memory allocation failed." << endl;

exit(-1);

}

for (int j = 0; j < cols; j++) {

visit[i][j] = false;

}

}

MyStack<Coordinate> DFS\_Path(rows \* cols);

Coordinate start{ startRow ,startCol };

DFS\_Path.Push(start);

while (!DFS\_Path.isEmpty()) {

Coordinate current;

DFS\_Path.getTop(current);

currRow = current.row;

currCol = current.col;

visit[currRow][currCol] = true;

if (currRow == targetRow && currCol == targetCol) {

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

delete[] visit[i];

delete[] visit;

while (!DFS\_Path.isEmpty()) {

Coordinate pathPoint;

DFS\_Path.Pop(pathPoint);

maze[pathPoint.row][pathPoint.col].isPath = true;

}

ouputMaze();

return true;

}

Coordinate temp;

if ((visit[currRow - 1][currCol] || maze[currRow - 1][currCol].isWall) &&

(visit[currRow + 1][currCol] || maze[currRow + 1][currCol].isWall) &&

(visit[currRow][currCol - 1] || maze[currRow][currCol - 1].isWall) &&

(visit[currRow][currCol + 1] || maze[currRow][currCol + 1].isWall) ) {

DFS\_Path.Pop(temp);

}

else {

if (!visit[currRow - 1][currCol] && !maze[currRow - 1][currCol].isWall) {

temp = { currRow - 1 ,currCol };

DFS\_Path.Push(temp);

}

else if (!visit[currRow + 1][currCol] && !maze[currRow + 1][currCol].isWall) {

temp = { currRow + 1 ,currCol };

DFS\_Path.Push(temp);

}

else if (!visit[currRow][currCol - 1] && !maze[currRow][currCol - 1].isWall) {

temp = { currRow ,currCol - 1 };

DFS\_Path.Push(temp);

}

else if (!visit[currRow][currCol + 1] && !maze[currRow][currCol + 1].isWall) {

temp = { currRow ,currCol + 1 };

DFS\_Path.Push(temp);

}

}

}

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

delete[] visit[i];

delete[] visit;

return false;

}

## 3.2异常处理功能的实现

### 3.2.1动态内存申请失败的异常处理

在进行动态内存申请时，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（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."；

(2)调用exit函数，返回错误码-1，用于指示内存分配错误，并导致程序退出。

### 3.2.2输入非法的异常处理

程序通过调用inputOdd函数输入迷宫的宽度和高度。inputOdd函数用于获取用户输入的整数，同时限制输入必须在指定的范围内，函数的代码如下：

int inputOdd(int lowerLimit, int upperLimit, const char\* prompt)

{

cout << ">>> 请输入" << prompt << " [奇数范围: " << lowerLimit << "~" << upperLimit << "]: ";

int input;

while (true) {

cin >> input;

if (cin.good() && input >= lowerLimit && input <= upperLimit && input % 2 == 1) {

cin.clear();

cin.ignore(INT\_MAX, '\n');

return input;

}

else {

cerr << ">>> " << prompt << "输入不合法，请重新输入！" << endl;

cin.clear();

cin.ignore(INT\_MAX, '\n');

}

}

std::cout << std::endl;

}

# 4项目测试

|  |
| --- |
|  |

4.1迷宫生成功能测试示例

|  |
| --- |
|  |

4.2路径生成DFS算法功能测试示例

|  |
| --- |
|  |

4.3路径生成BFS算法功能测试示例

# 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