同济大学

TONGJI UNIVERSITY

数据结构课程设计

项目名称	勇闯迷宫游戏
学　　院	计算机科学与技术学院
专　　业	软件工程
学生姓名	杨瑞晨
学　　号	2351050
指导教师	张颖
日　　期	2024 年 12 月 6 日

目 录

1 项目分析

1.1 项目背景分析

迷宫问题是一个经典的算法问题，涉及到图搜索和路径规划。本项目旨在实现一个迷宫路径搜索系统，该系统能够通过深度优先搜索（DFS）算法找到从迷宫入口到出口的路径。

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。

1.2 项目功能分析

1.2.1 功能要求

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。

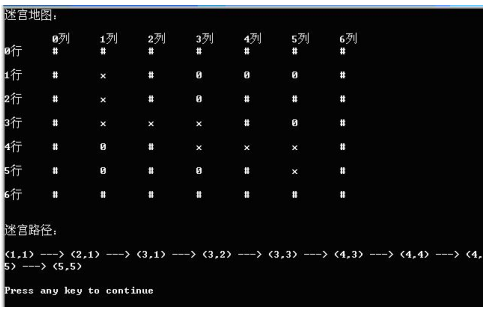
1.2.2 输入要求

输入迷宫地图，地图中包含两个门，一个入口，一个出口，用字符表示，其中“1”表示障碍，“0”表示可通行的道路。

1.2.3 输出要求

输出迷宫地图，“#”表示障碍，“0”表示可通行的道路，“*”表示骑士走过的路径，即迷宫问题的解。同时输出迷宫路径。

1.2.4 项目实例



2 项目设计

2.1 数据结构设计

根据题意，我们可以选择采用深度优先搜索算法来解决迷宫问题，该算法通常使用栈作为路径的存储数据结构。另外，为了有效地表示坐标信息，我们需要自定义一个包含至少 x 和 y 两个成员的类或结构体，用来表示坐标节点。

2.1.1 迷宫路径栈（MazePath）

栈是一种后进先出（LIFO）的数据结构，它在本项目中用于存储操作符和临时节点，以便在构建表达式树时处理操作符的优先级，具有以下优点：

- 栈提供的基本操作如入栈（push）、出栈（pop）、查看栈顶元素（top）等操作都非常简单，易于实现。
- 栈结构天然适合实现路径搜索中的回溯操作。

2.2 类设计

2.2.1 迷宫单元 MazeCell 类

MazeCell 类用于表示迷宫中的单个单元格，它可以是通道或墙，也可以表示迷宫中的路径。

主要成员：

- int x, y：分别表示单元格在迷宫中的行和列坐标。
- bool visited：布尔值，表示单元格是否已被访问过。
- bool wall：布尔值，表示单元格是否为墙。

主要方法：

- 构造函数、拷贝构造函数和析构函数：初始化和销毁 MazeCell 对象。
- getX()、getY()：返回单元格的坐标。
- isVisited()、isWall()：返回单元格的访问状态和墙状态。
- setVisited(bool)、setWall(bool)：设置单元格的访问状态和墙状态。
- operator=：赋值运算符重载，用于复制 MazeCell 对象。

```
// 迷宫单元类
class MazeCell
{
    friend class Maze;
    friend class MazePath;

private:
    int x;        // x 坐标
    int y;        // y 坐标
    bool visited; // 是否访问过
```

```
    bool wall;    // 墙
public:
    MazeCell(int x = 0, int y = 0, bool visited = false, bool wall = false) : x(x), y(y), visited(visited), wall(wall) {} // 构造函数
    MazeCell(const MazeCell &cell) : x(cell.x), y(cell.y), visited(cell.visited), wall(cell.wall) {}           // 拷贝构造函数
    ~MazeCell() {}                                  // 析构函数
    int getX() { return x; }                        // 获取 x 坐标
    int getY() { return y; }                        // 获取 y 坐标
    bool isVisited() { return visited; }            // 是否访问过
    bool isWall() { return wall; }                  // 是否是墙
    void setVisited(bool visited) { this->visited = visited; }      // 设置是否访问过
    void setWall(bool wall) { this->wall = wall; }                  // 设置是否是墙
    MazeCell &operator=(const MazeCell &cell)
    { // 赋值运算符重载
        if (this != &cell)
        {
            x = cell.x;
            y = cell.y;
            visited = cell.visited;
            wall = cell.wall;
        }
        return *this;
    }
};
```

2.2.2 MazePath 类

MazePath 类用于存储搜索过程中的路径，它使用栈结构来实现路径的回溯。

主要成员：

- int top：指示栈顶的位置。
- MazeCell *stack[]：存储路径的数组，每个元素是一个指向 MazeCell 的指针。

主要方法：

- 构造函数和析构函数：初始化和销毁 MazePath 对象。
- push(MazeCell*)：将一个单元格指针压入栈中，表示路径的一部分。
- pop()：从栈中弹出顶部的单元格指针，用于回溯。
- getTop()：获取栈顶的单元格指针，不弹出。
- isEmpty()：检查栈是否为空。
- clear()：清空栈。
- printPath()：打印整个路径。

```
// 迷宫路径栈，用于存放路径
class MazePath
{
    friend class Maze;
    friend class MazeCell;

```

```
private:
    int top;                                // 栈顶
    MazeCell *stack[MAX_SIZE * MAX_SIZE]; // 栈
public:
    MazePath() { top = -1; } // 构造函数
    ~MazePath() {}           // 析构函数
    void push(MazeCell *cell)
    { // 入栈
        if (top < MAX_SIZE * MAX_SIZE - 1) stack[++top] = cell;
    }
    MazeCell *pop()
    { // 出栈
        if (top >= 0) return stack[top--];
        return nullptr;
    }
    MazeCell *getTop()
    { // 获取栈顶元素
        if (top >= 0) return stack[top];
        return nullptr;
    }
    bool isEmpty() { return top == -1; }  // 判断栈是否为空
    void clear() { top = -1; }// 清空栈
    void printPath()
    { // 打印路径
        for (int i = 0; i <= top; i++)
        {
            cout << "(" << stack[i]->getX() << ", " << stack[i]->getY() << ")" << (i == top ? "" : " ---> ");
        }
        cout << endl;
    }
};
```

2.2.3 Maze 类

Maze 类表示整个迷宫，它包含迷宫的布局、起点、终点和搜索路径。

主要成员：

- int size：迷宫的大小。
- MazeCell cells**：二维数组，存储迷宫中所有的单元格。
- MazeCell start, end：分别表示迷宫的起点和终点。
- MazePath path：MazePath 对象，存储搜索到的路径。

主要方法：

- 构造函数和析构函数：初始化和销毁 Maze 对象。
- inputMaze()：从用户输入中获取迷宫布局和起点。
- printMaze()：打印迷宫的当前状态。
- solution()：执行路径搜索并打印结果。
- dfsFindPath(int, int)：递归实现深度优先搜索算法。

```cpp
// 迷宫类
class Maze
{
    friend class MazePath;
    friend class MazeCell;

private:
    int size;         // 迷宫大小
    MazeCell **cells; // 迷宫单元
    MazeCell start;   // 起点
    MazeCell end;     // 终点
    MazePath path;    // 路径
public:
    Maze(int size = 0) : size(size), start(MazeCell()), end(MazeCell()), path(MazePath())
    { // 构造函数
        cells = new MazeCell *[size];
        for (int i = 0; i < size; i++)
        {
            cells[i] = new MazeCell[size];
        }
    }

    ~Maze()
    { // 析构函数
        for (int i = 0; i < size; i++)
        {
            delete[] cells[i];
        }
        delete[] cells;
    }

    void inputMaze(); // 输入迷宫


    void printMaze()
    { // 打印迷宫
        cout << endl
             << " 迷宫如下: 起点为 (" << start.getX() << ", " << start.getY() << "), 终点为 ("
             << end.getX() << ", " << end.getY() << ")" << endl;
        cout << "    ";
        for (int i = 0; i < size; i++)
        {
            cout << "C" << i << (i > 9 ? " " : "  ");
        }
        cout << endl;
        for (int i = 0; i < size; i++)
        {
            cout << "R" << i << (i > 9 ? " " : "  ");
            for (int j = 0; j < size; j++)
            {
                if (cells[i][j].isWall())
                {
                    cout << "#";
                }
                else if (cells[i][j].isVisited())
                {
                    cout << "*";
                }
                else
```

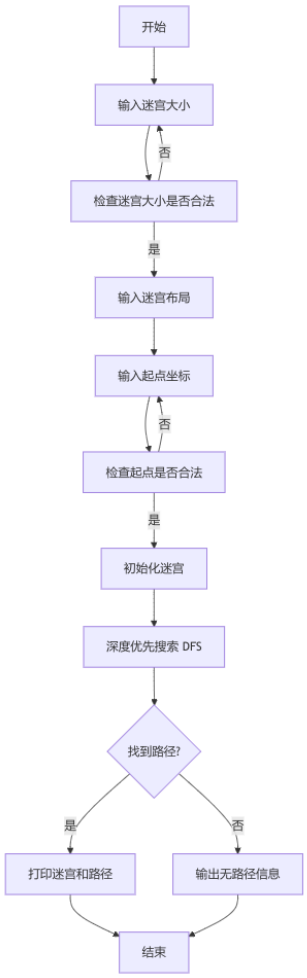
```
                {
                    cout << "0";
                }
                cout << "   ";
            }
            cout << endl
                 << endl;
        }
    }

    void solution()
    {
        if (dfsFindPath(start.getX(), start.getY())) // 查找路径
        {
            printMaze();
            cout << " 路径如下: " << endl;
            path.printPath();
        }
        else
        {
            cout << " 没有找到路径!" << endl;
            return;
        }
    }

    bool dfsFindPath(int x, int y); // DFS 算法查找路径
};
```

3 项目实现

3.1 流程表示



3.2 功能实现

3.2.1 输入迷宫

从用户处接收迷宫布局和起点，并进行合法性检查。

（1）提示用户输入迷宫布局，并读取输入填充 Maze 对象的 cells 数组。（2）检查起点是否合法（必须在迷宫边界上且不是角落，且不能是墙）。（3）检查迷宫出口的唯一性和合法性，确保只有一个出口，并且出口不在角落。

```
    void inputMaze()
    { // 输入迷宫
        cout << " 请输入迷宫 (0: 通道, 1: 墙): " << endl;
        for (int i = 0; i < size; i++)
```

```
        {
            for (int j = 0; j < size; j++)
            {
                int n;
                cin >> n;
                cells[i][j] = MazeCell(i, j);
                if (n == 1) // 墙
                {
                    cells[i][j].setWall(true);
                }
            }
        }
        cout << " 请输入起点: x y (范围: [0, " << size - 1 << "]): ";
        int x, y;
        cin >> x >> y;
        if (!isBorder(x, y, size) || cells[x][y].isWall() || isCorner(x, y, size))
        {
            cerr << " 非法起点! 起点必须在边界上且不能在四个角落, 并且不能是墙。" << endl;
            exit(1);
        }
        start = cells[x][y];
        // cout << start.getX() << " " << start.getY() << endl;

        // 检查出口的唯一性
        int exitCount = 0;
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                if (isBorder(i, j, size) && !cells[i][j].isWall() && !(i == start.getX() && j
                ↪  == start.getY()) && !isCorner(i, j, size))
                {
                    exitCount++;
                    end = cells[i][j];
                }
            }
        }
        if (exitCount != 1)
        {
            cerr << " 迷宫必须有且仅有一个出口, 并且出口不能在四个角落! " << endl;
            exit(1);
        }
    }
```

3.2.2 路径搜索 - dfs 递归回溯

dfsFindPath 方法是迷宫路径搜索系统中的核心算法，使用深度优先搜索（DFS）策略来寻找从起点到终点的路径。该方法递归地探索迷宫中的每个可能的路径，直到找到终点或确认没有路径。

算法步骤：

（1）边界和障碍检查：检查当前位置（由坐标 (x, y) 给出）是否超出迷宫边界，或者是否是墙，或者是否已经被访问过。如果是上述任何一种情况，搜索将停止并回溯。

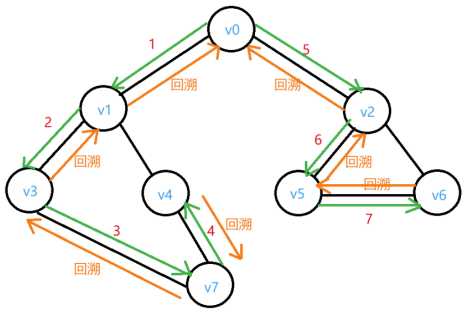
（2）标记访问：如果当前位置是合法的，标记该位置为已访问，并将其添加到路径栈中。

（3）到达终点检查：如果当前位置是迷宫的边界（不是角落或起点）并且是通道（不是墙），则认为找到了终点，返回 true。

（4）递归搜索：递归地搜索当前位置的上、下、左、右四个方向。如果任一方向找到了路径，则整个函数返回 true。

（5）回溯：如果四个方向都没有找到路径，则当前路径不成立，需要回溯。取消当前位置的访问标记，并将其从路径栈中移除。

以下是 DFS 的示意图：



代码实现：

```
bool dfsFindPath(int x, int y)
{ // DFS 算法查找路径
    if (x < 0 || x >= size || y < 0 || y >= size || cells[x][y].isWall() ||
    ↪  cells[x][y].isVisited()) // 越界或者是墙或者已经访问过
    {
        // cout << " 当前位置: (" << x << ", " << y << ") 无法继续前进" << endl;
        return false; // 越界或者是墙或者已经访问过
    }

    // cout << " 当前位置: (" << x << ", " << y << ")" << endl;

    cells[x][y].setVisited(true); // 标记为已访问
    path.push(&cells[x][y]);      // 入栈

    if (isBorder(x, y, size) && (x != start.getX() || y != start.getY())) // 到达终点
    {
        end = MazeCell(x, y);
        return true;
    }

    if (dfsFindPath(x - 1, y) || // 向上
        dfsFindPath(x + 1, y) || // 向下
        dfsFindPath(x, y - 1) || // 向左
        dfsFindPath(x, y + 1))   // 向右
    {
        return true;
    }

    // cout << " 当前位置: (" << x << ", " << y << ") 无法继续前进，回溯" << endl;

```

```
30          cells[x][y].setVisited(false); // 标记为未访问
31          path.pop();                    // 出栈
32          return false;
33      }
34  };
```

3.3 main 函数

（1） 调用 getInputSize() 函数获取用户输入的迷宫大小。（2） 创建 Maze 对象，并传入迷宫大小。（3） 调用 maze.inputMaze() 方法从用户处获取迷宫布局和起点。（4） 调用 maze.solution() 方法执行路径搜索并输出结果。

```
1   int main()
2   {
3       Maze maze(getInputSize());
4       maze.inputMaze();
5       maze.solution();
6
7       return 0;
8   }
```

4 项目测试

4.1 正常测试

4.1.1 普通情况

```
请输入迷宫大小(范围[3, 12]): 7
请输入迷宫(0: 通道, 1: 墙):
1 0 1 1 1 1 1
1 0 1 0 0 0 1
1 0 1 0 1 1 1
1 0 0 0 1 0 1
1 0 1 0 0 0 1
1 0 1 0 1 0 1
1 1 1 1 1 0 1
请输入起点: x y (范围: [0, 6]): 0 1

迷宫如下: 起点为(0, 1), 终点为(6, 5)
    C0  C1  C2  C3  C4  C5  C6
R0  #   *   #   #   #   #   #

R1  #   *   #   0   0   0   #

R2  #   *   #   0   #   #   #

R3  #   *   *   *   #   0   #

R4  #   0   #   *   *   *   #

R5  #   0   #   0   #   *   #

R6  #   #   #   #   #   *   #

路径如下:
(0, 1) ---> (1, 1) ---> (2, 1) ---> (3, 1) ---> (3, 2) ---> (3, 3) ---> (4, 3) ---> (4, 4) ---> (4, 5) ---> (5, 5) ---> (6, 5)
```

```
请输入迷宫大小(范围[3, 12]): 7
请输入迷宫(0: 通道, 1: 墙):
1 1 0 1 1 1 1
1 0 0 0 0 0 1
1 0 1 0 1 1 1
1 0 0 0 1 0 0
1 0 1 0 0 0 1
1 0 1 0 1 0 1
1 1 1 1 1 1 1
请输入起点: x y (范围: [0, 6]): 3 6

迷宫如下: 起点为(3, 6), 终点为(0, 2)
    C0  C1  C2  C3  C4  C5  C6
R0  #   #   *   #   #   #   #

R1  #   0   *   *   0   0   #

R2  #   0   #   *   #   #   #

R3  #   0   0   *   #   *   *

R4  #   0   #   *   *   *   #

R5  #   0   #   0   #   0   #

R6  #   #   #   #   #   #   #

路径如下:
(3, 6) ---> (3, 5) ---> (4, 5) ---> (4, 4) ---> (4, 3) ---> (3, 3) ---> (2, 3) ---> (1, 3) ---> (1, 2) ---> (0, 2)
```

4.1.2 没有出口

```
请输入迷宫大小(范围[3, 12]): 7
请输入迷宫(0: 通道, 1: 墙):
1 1 0 1 1 1 1
1 0 0 0 0 0 1
1 0 1 0 1 1 1
1 0 0 0 1 0 1
1 0 1 0 0 0 1
1 0 1 0 1 0 1
1 1 1 1 1 1 1
请输入起点: x y (范围: [0, 6]): 0 2
迷宫必须有且仅有一个出口，并且出口不能在四个角落!
```

4.1.3 两个出口

```
请输入迷宫大小(范围[3, 12]): 7
请输入迷宫(0: 通道, 1: 墙):
1 1 0 1 1 1 1
1 0 0 0 0 0 1
1 0 1 0 1 1 1
1 0 0 0 1 0 1
1 0 1 0 0 0 1
1 0 1 0 1 0 1
1 0 1 1 1 0 1
请输入起点: x y (范围: [0, 6]): 0 2
迷宫必须有且仅有一个出口，并且出口不能在四个角落!
```

4.1.4 起点在角落

```
请输入迷宫大小(范围[3, 12]): 5
请输入迷宫(0: 通道, 1: 墙):
0 1 1 1 1
0 0 0 1 1
1 1 0 1 1
1 0 0 0 0
1 1 1 1 1
请输入起点: x y (范围: [0, 4]): 0 0
非法起点! 起点必须在边界上且不能在四个角落，并且不能是墙。
```

4.2 健壮性测试

针对非法输入，程序能够给出合理的提示。

```
请输入迷宫大小(范围[3, 12]): 2
输入错误，请重新输入: q
输入错误，请重新输入: 4
请输入迷宫(0: 通道, 1: 墙):
1 0 1 1 1
0 2
3 4 5
q
请输入起点: x y (范围: [0, 3]): 非法起点! 起点必须在边界上
```

5 项目心得与体会

在完成迷宫路径搜索系统的项目开发过程中，我获得了宝贵的实践经验和深刻的认识，这些不仅增强了我的专业知识，也提升了我的实践技能和解决问题的能力。

通过设计和实现 MazeCell、MazePath 和 Maze 类，我深刻体会到了数据结构在解决实际问题中的重要性。这些类不仅帮助我组织和管理迷宫数据，还使得算法实现变得更加直观和高效。在实现深度优先搜索（DFS）算法的过程中，我对递归和回溯有了更深的理解。我学会了如何通过递归遍历迷宫中的每个单元格，并在遇到死胡同时进行有效的回溯。此外，我也探索了如何优化 DFS 算法，以减少不必要的搜索并提高效率。本项目让我学会了如何将复杂的理论问题转化为具体的编程任务，并找到有效的解决方案。我学会了如何规划项目、分解任务、分配资源，并确保项目按时完成。这些经验和技能将为我未来的学习和工作奠定坚实的基础。

总之，这个项目不仅让我掌握了数据结构和算法的具体应用，还提升了我的编程能力、问题解决能力和系统设计能力。这些经验和技能将为我未来的学习和工作奠定坚实的基础。