数据结构课程设计文档

题目：迷宫游戏

姓名：赵卓冰

学号：2252750

专业：软件工程

年级：2023 级

指导教师：张颖

2023 年 12 月 1 日

目录

1. 开发环境与编译运行环境

本项目可以在不同的开发环境和编译运行环境上运行

- Window操作系统：
  - 版本：Windows 10 x64
  - IDE: Visual Studio 2022 (Debug模式)
  - 编译器： MSVC 14.39.33519
- Linux操作系统：
  - 版本：Ubuntu 20.04.6 LTS
  - IDE： VS Code
  - 编译器：gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)

2. 项目要求

- 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。
- 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。

3. 项目简介

本项目是一个迷宫求解的模拟系统，旨在实现一个骑士骑马在迷宫中从入口到出口寻找通路的过程。迷宫中包含各种障碍，骑士需要根据特定的算法来探索可行的路径，最终找到一条通向出口的通路。
迷宫的随机生成采用深度优先搜索算法。
项目实现了多种迷宫求解算法，包括：

- 回溯法（递归）

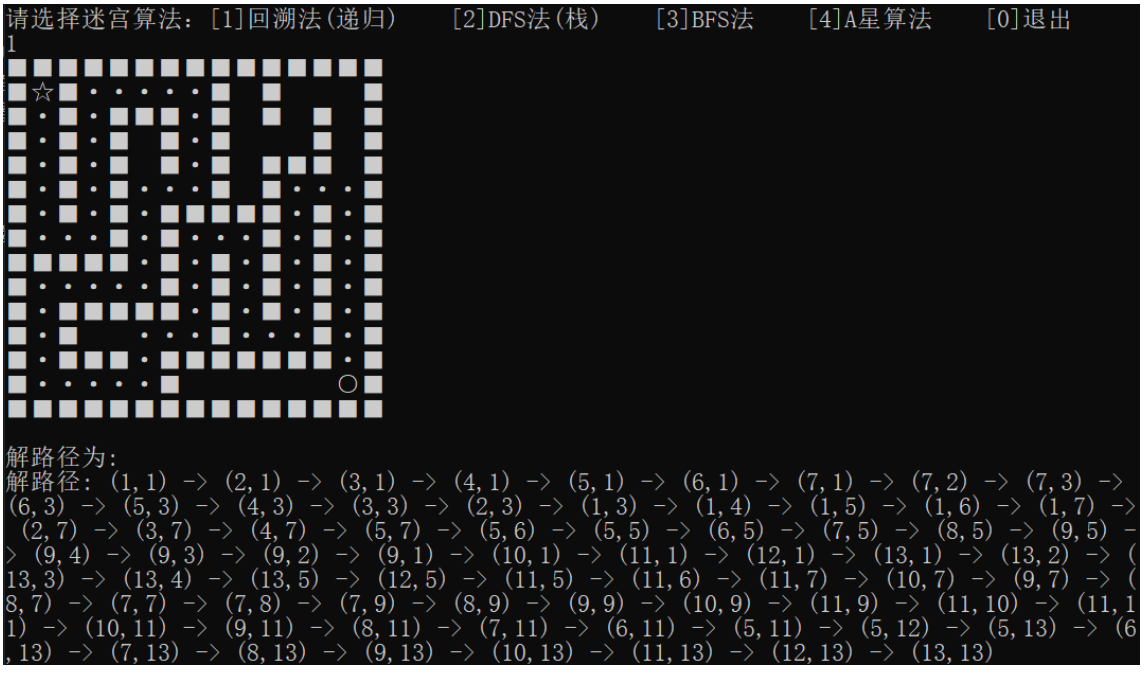
- 基于栈的深度优先搜索（DFS）
- 广度优先搜索（BFS）
- A* 算法（启发式搜索）

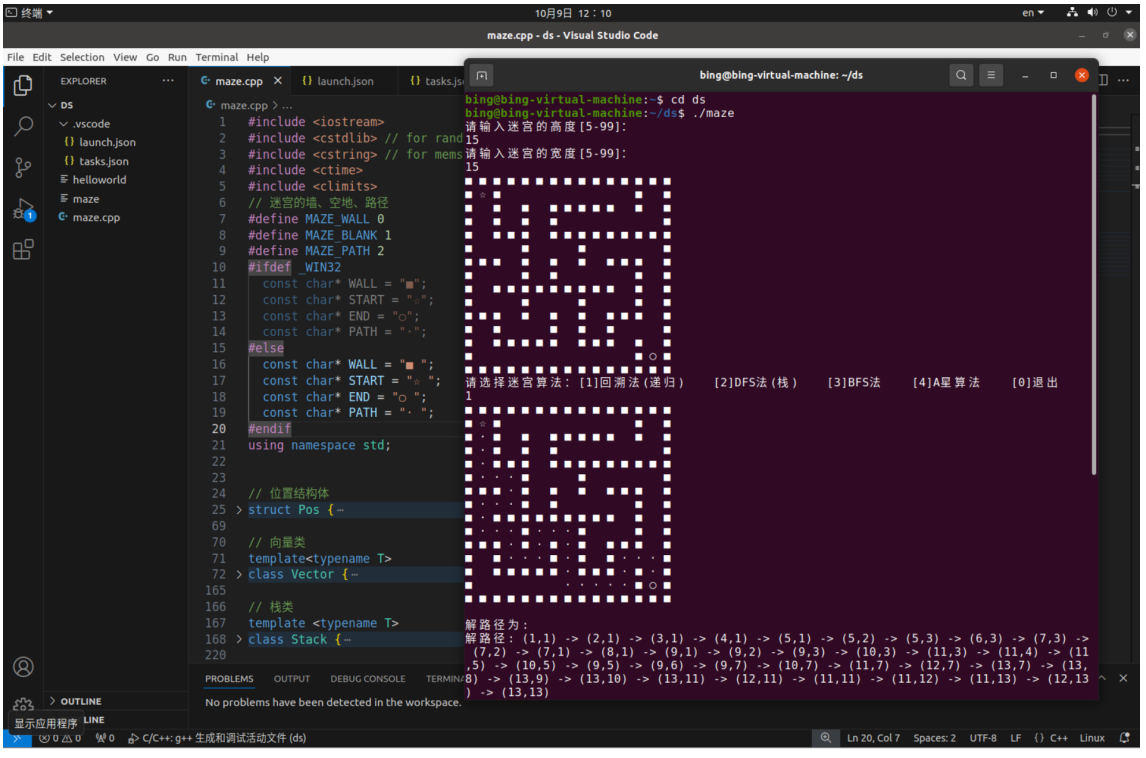
用户可以通过交互式的菜单选择不同的算法进行迷宫求解，并且程序会可视化展示迷宫的生成和解路径。此外，项目还自定义实现了几种基础数据结构（如栈、队列、优先队列等），用于支持迷宫求解过程中的数据管理。

该项目的主要目标是通过迷宫求解问题的解决，学习和掌握经典的算法设计方法及回溯法的应用。

4. UI设计和界面展示

- 首先，用户自定义输入迷宫的高度和宽度，然后程序打印出给定高度和宽度的随机地图， 其中'☆'代表起点，白色块代表墙壁，黑色代表空地，'○'代表终点。
- 

- 接着，用户选择寻找路径的算法，程序展示算法求得的解路径，其中'·'表示解路径。
- 

- 用户可以再次选择不同的算法，也可以输入0退出。
-
```
请选择迷宫算法：[1]回溯法(递归)　　[2]DFS法(栈)　　[3]BFS法　　[4]A星算法　　[0]退出
0
程序成功退出！
```
- Windows界面展示
- 
- Linux界面展示
- 

5. 代码架构

5.1. 数据结构

在本项目中，为了实现迷宫求解算法的各类需求，自定义了几种基础的数据结构，包括动态数组（Vector）、栈（Stack）、队列（Queue）和优先队列（PriorityQueue）。这些数据结构在路径搜索过程中用于存储状态、管理路径和优先级排序。

5.1.1. Pos 结构体

`Pos` 是一个结构体，用于表示迷宫中的位置坐标。

- **成员变量**：
  - `row`：行索引，表示当前坐标的行位置。
  - `col`：列索引，表示当前坐标的列位置。
- **重载运算符**：
  - `+` 和 `-`：用于坐标的加减运算。
  - `/`：对坐标进行除法运算。
  - `==`：判断两个坐标是否相同。
  - `<<`：用于输出坐标，方便调试和展示。

5.1.2. Vector 类

`Vector` 类是一个自定义实现的动态数组，用于存储元素集合，支持动态扩容和访问操作。

- **成员变量**：
  - `data`：指向数组数据的指针。
  - `capacity`：数组当前分配的容量。
  - `size`：当前存储的元素数量。
- **主要方法**：
  - `PushBack()`：在数组末尾添加一个元素。
  - `PopBack()`：删除数组末尾的元素。
  - `Size()`：返回当前数组中的元素数量。
  - `Reverse()`：反转数组中的元素顺序。
  - `Clear()`：清空数组。

5.1.3. Stack 类

`Stack` 类实现了栈的数据结构，用于模拟深度优先搜索的递归行为。

- **成员变量**：
  - `arr`：用于存储栈中元素的数组。
  - `top_index`：栈顶元素的索引。
  - `capacity`：栈的容量。
- **主要方法**：
  - `Push()`：将元素压入栈中。
  - `Pop()`：将栈顶元素弹出。
  - `Top()`：获取栈顶元素。
  - `Empty()`：判断栈是否为空。

5.1.4. Queue 类

`Queue` 类实现了队列的数据结构，用于广度优先搜索的遍历操作。

- **成员变量**：
  - `data`：用于存储队列中元素的数组。
  - `front_index`：队首元素的索引。
  - `back_index`：下一个可插入元素的位置。
  - `count`：队列中元素的数量。
  - `capacity`：队列的容量。
- **主要方法**：
  - `Push()`：将元素添加到队尾。
  - `Pop()`：删除队首元素。
  - `Front()`：获取队首元素。
  - `Back()`：获取队尾元素。
  - `Empty()`：判断队列是否为空。

5.1.5. PriorityQueue 类

`PriorityQueue` 类是一个优先队列的实现，基于最小堆，用于 A* 算法中的优先级排序。

- **成员变量**：

- `data`：存储优先队列中的元素。
- `size`：优先队列中的元素数量。
- `capacity`：优先队列的容量。
- `comp`：比较函数，用于元素的优先级比较。

- **主要方法**：
  - `Push()`：将元素添加到优先队列中。
  - `Pop()`：移除优先队列中的最高优先级元素。
  - `Top()`：获取优先队列中最高优先级的元素。
  - `Empty()`：判断优先队列是否为空。

5.1.6. Node 结构体

`Node` 是 A* 算法中使用的节点结构体，用于表示搜索过程中的一个位置状态。

- **成员变量**：
  - `pos`：当前位置的坐标（`Pos` 类型）。
  - `start_cost`：从起点到当前位置的代价。
  - `end_cost`：从当前位置到终点的估计代价（启发式值）。
  - `sum_cost`：总代价，即 `start_cost + end_cost`。

这些数据结构为迷宫求解提供了基本的功能支持，能够有效地管理算法执行过程中的状态和数据。

5.2. 随机生成地图算法

在本项目中，迷宫的随机生成使用了深度优先搜索（DFS）算法，通过从初始点逐步扩展路径的方式来创建复杂的迷宫结构。随机生成的迷宫会包含大量的墙壁和空地，其中只有少数路径可以通行。这种方法能够保证生成的迷宫既具备挑战性，又有明确的解路径。

5.2.1. 算法步骤

1. **初始化迷宫结构**
   - 创建一个二维数组 `field`，用于表示迷宫的网格结构。
   - 使用 `MAZE_WALL`（值为0）初始化所有位置，将整个迷宫设为墙壁。
   - 设置迷宫的起点和终点，分别位于 `start_row`，`start_col` 和 `end_row`，`end_col`。这些位置的值设为 `MAZE_BLANK`（值为1），表示空地。
2. **定义搜索方向**

- 采用四个可能的方向：上、下、左、右，分别表示为坐标的变化量。

```
上: {-2, 0}
下: {2, 0}
左: {0, -2}
右: {0, 2}
```

  - 这些方向代表了迷宫中的大步移动，跳过一个格子，从而在路径上形成“走廊”的效果。

3. **从起点开始深度优先搜索**
   - 将起点位置推入栈中，并将其设置为空地（`MAZE_BLANK`）。
   - 使用栈来管理搜索过程，以模拟深度优先的路径探索。
4. **随机选择方向并拓展路径**
   - 从栈顶取出当前位置，随机打乱四个方向的顺序以增加迷宫的随机性。
   - 尝试沿随机选择的方向前进。如果新位置是未访问的墙壁且在边界范围内，则将当前位置和新位置之间的墙打通，并将新位置设置为空地。
   - 将当前位置和新位置都压入栈，以便继续探索新的路径。
5. **回溯处理**
   - 如果当前位置的四个方向都没有可行的路径，说明已经到达了一个“死胡同”，则从栈中弹出该位置，返回到上一个位置，继续尝试其他方向。
6. **重复以上步骤，直至栈为空**
   - 当栈为空时，迷宫生成过程结束，此时迷宫中包含了起点到终点的连通路径，同时还有许多死胡同和障碍物。

5.2.2. 代码实现

算法的核心实现可以简要描述如下：

```
void MAZE::MakeRandomMaze() {
  Pos directions[4] = { {-2, 0}, {2, 0}, {0, -2}, {0, 2} }; // 上下左右方向

  field[start_row][start_col] = MAZE_BLANK; // 起点设为空地
  Stack<Pos> stack;
  stack.Push({ start_row, start_col }); // 将起点压入栈

  while (!stack.Empty()) {
    Pos current_pos = stack.Top();
    stack.Pop();
    // 随机打乱方向顺序
```

```
    for (int i = 0; i < 4; ++i) {
      int random_index = rand() % 4;
      swap(directions[i], directions[random_index]);
    }
    // 尝试随机的方向
    for (int i = 0; i < 4; ++i) {
      Pos new_pos = current_pos + directions[i];
      Pos between_pos = current_pos + directions[i] / 2; // 当前路径间的格子

      // 检查新位置是否在边界内且为墙壁
      if (IsInBounds(new_pos) && field[new_pos.row][new_pos.col] == MAZE_WALL) {
        field[between_pos.row][between_pos.col] = MAZE_BLANK; // 打通墙壁
        field[new_pos.row][new_pos.col] = MAZE_BLANK; // 新位置设为空地
        stack.Push(current_pos);
        stack.Push(new_pos); // 将新位置压入栈继续探索
        break;
      }
    }
  }
}
```

5.2.3. 算法特点

- **随机性**：通过随机打乱方向顺序，使得每次生成的迷宫结构都不同。
- **连通性**：使用深度优先搜索的方式保证从起点到终点至少存在一条可行的路径。
- **迷宫的复杂度**：生成的迷宫有较多的死胡同，使得解迷宫的过程更具挑战性。

5.2.4. 优点和缺点

- **优点**：
  - 简单易实现，生成速度快。
  - 生成的迷宫具备复杂的结构，能够提供有趣的解迷挑战。
- **缺点**：
  - 生成的迷宫会有很多死胡同，路径数量相对较少。
  - 迷宫的密度和难度不易控制，生成的路径长度和复杂度具有较大的随机性。

这种随机生成迷宫的算法为项目提供了丰富的测试用例和算法应用场景，是迷宫求解的基础。

5.3. 迷宫求解算法

项目中实现了多种迷宫求解算法，用于找到从入口到出口的通路。这些算法包括回溯法（递归）、基于栈的深度优先搜索（DFS）、广度优先搜索（BFS）和 A* 算法。每种算法的思路和应用场景不同，能够适应不同的迷宫特征和路径搜索需求。

5.3.1. 回溯法（递归）

5.3.1.1. 算法思想

回溯法通过递归地尝试每一个可能的路径，逐步向前探索。如果某个方向可行，则继续前进；如果前进受阻（即遇到死胡同或障碍），则回溯到上一个位置，尝试其他方向。这种试探-回退的过程会在所有可能的路径中进行，直到找到通往出口的通路，或者所有可能的路径都已经探索完毕。

5.3.1.2. 算法步骤

1. **从入口开始递归**，尝试向四个方向移动（上、下、左、右）。
2. **检查新位置**是否在迷宫范围内，且为可通行的空地。
3. 如果找到可通行的路径，**标记当前位置为已访问**，并继续向该方向递归。
4. 如果到达迷宫出口，**返回成功**，表示找到了一条通路。
5. 如果所有方向都不可行，**回溯到上一个位置**，继续尝试其他方向。
6. 当所有递归都结束时，如果没有找到通路，则输出“没有解”。

5.3.1.3. 代码

```
bool MAZE::TraceBack(Pos current_pos, int** visit) {
  if (current_pos.row == end_row && current_pos.col == end_col) {
    solution_path.PushBack(current_pos);
    return true;
  }

  visit[current_pos.row][current_pos.col] = 1; // 标记为已访问
  solution_path.PushBack(current_pos);

  for (int i = 0; i < 4; ++i) {
    Pos new_pos = current_pos + directions[i];
    if (IsInBounds(new_pos) && field[new_pos.row][new_pos.col] == MAZE_BLANK && visit[new_pos.row][new_pos.col] == 0) {
      if (TraceBack(new_pos, visit)) {
        return true;
```

```
      }
    }
  }

  solution_path.PopBack(); // 回溯
  return false;
}
```

5.3.1.4. 优缺点

- **优点**：算法简单，易于实现，能够找到所有可能的路径。
- **缺点**：递归深度过大时可能会导致栈溢出，不适用于非常大的迷宫。

5.3.2. 基于栈的深度优先搜索（DFS）

5.3.2.1. 算法思想

深度优先搜索（DFS）是一种模拟递归行为的路径搜索算法，使用栈来存储当前路径的节点。算法从入口出发，将起始位置压入栈中，然后按一个方向不断深入，直到无法继续为止。这时，从栈中弹出当前位置，回溯到上一个节点，继续探索其他方向。

5.3.2.2. 算法步骤

1. **将起点压入栈**，并标记为已访问。
2. 当栈非空时，取栈顶元素作为当前节点。
3. **检查是否到达终点**，如果到达，输出解路径。
4. 否则，尝试四个方向的移动，若找到一个可行路径，则将该位置压入栈，并标记为已访问。
5. 如果四个方向均不可行，从栈中弹出当前节点，回溯到上一个节点。

5.3.2.3. 代码示例

```
void MAZE::SolveWithStack() {
  Stack<Pos> stack;
  stack.Push({ start_row, start_col });
  visit[start_row][start_col] = 1;

  while (!stack.Empty()) {
    Pos current_pos = stack.Top();
    if (current_pos.row == end_row && current_pos.col ==
end_col) {
      // 找到解路径
      break;
```

```
    }

    bool is_found = false;
    for (int i = 0; i < 4; ++i) {
      Pos new_pos = current_pos + directions[i];
      if (IsInBounds(new_pos) && field[new_pos.row]
[new_pos.col] == MAZE_BLANK && !visit[new_pos.row]
[new_pos.col]) {
        is_found = true;
        visit[new_pos.row][new_pos.col] = 1;
        stack.Push(new_pos);
        break;
      }
    }
    if (!is_found) {
      stack.Pop();
    }
  }
}
```

5.3.2.4. 优缺点

- **优点**：非递归实现，避免了栈溢出问题。

5.3.3. 广度优先搜索（BFS）

5.3.3.1. 算法思想

广度优先搜索（BFS）是一种逐层扩展的路径搜索算法，使用队列来存储当前层的节点。每次从队列中取出一个节点，尝试将其所有未访问的邻居节点加入队列。由于 BFS 是按层次搜索，因此可以保证找到的第一条路径是步数最短的路径。

5.3.3.2. 算法步骤

1. **将起点压入队列**，并标记为已访问。
2. 当队列非空时，取队首元素作为当前节点。
3. **检查是否到达终点**，如果到达，输出解路径。
4. 否则，尝试将当前节点的所有可通行邻居压入队列，并标记为已访问。
5. 继续从队列中取出下一个节点，直到队列为空或找到通路。

5.3.3.3. 代码示例

```
void MAZE::SolveWithQueue() {
  Queue<Pos> queue;
  queue.Push({ start_row, start_col });
  visit[start_row][start_col] = 1;

  while (!queue.Empty()) {
    Pos current_pos = queue.Front();
    if (current_pos.row == end_row && current_pos.col == end_col) {
      // 找到解路径
      break;
    }

    queue.Pop();
    for (int i = 0; i < 4; ++i) {
      Pos new_pos = current_pos + directions[i];
      if (IsInBounds(new_pos) && field[new_pos.row][new_pos.col] == MAZE_BLANK && !visit[new_pos.row][new_pos.col]) {
        queue.Push(new_pos);
        visit[new_pos.row][new_pos.col] = 1;
      }
    }
  }
}
```

5.3.3.4. 优缺点

- **优点**：能够找到步数最短的路径。
- **缺点**：需要较大的内存来存储队列中的节点，适用于较小规模的迷宫。

5.3.4. A* 算法（启发式搜索）

5.3.4.1. 算法思想

A* 算法是一种启发式搜索算法，它通过估计当前节点到终点的距离来优化搜索路径。每个节点都有一个“总代价” `f(n) = g(n) + h(n)`，其中 `g(n)` 是从起点到当前节点的实际代价，`h(n)` 是当前节点到终点的估计代价（启发值）。A* 算法通过优先选择总代价较小的节点进行扩展，从而更快地找到最优解。

5.3.4.2. 算法步骤

1. **将起点加入优先队列**，并设置其总代价。
2. 当优先队列非空时，取出总代价最小的节点作为当前节点。
3. **检查是否到达终点**，如果到达，输出解路径。
4. 否则，尝试将当前节点的所有可通行邻居加入优先队列，并更新其代价。
5. 继续从优先队列中取出总代价最小的节点，直到优先队列为空或找到通路。

5.3.4.3. 代码示例

```
void MAZE::SolveWithAStar() {
  PriorityQueue<Node, NodeCompare> p_queue;
  p_queue.Push(Node(start_pos, 0, Heuristic(start_pos, end_pos)));

  while (!p_queue.Empty()) {
    Node current_node = p_queue.Top();
    if (current_node.pos == end_pos) {
      // 找到解路径
      break;
    }

    p_queue.Pop();
    for (int i = 0; i < 4; ++i) {
      Pos new_pos = current_node.pos + directions[i];
      int new_start_cost = current_node.start_cost + 1;
      int new_end_cost = Heuristic(new_pos, end_pos);
      p_queue.Push(Node(new_pos, new_start_cost, new_end_cost));
    }
  }
}
```

5.3.4.4. 优缺点

- **优点**：启发式搜索，效率高，速度快，适用于大规模迷宫。
- **缺点**：依赖启发函数的准确性，启发函数不佳时可能退化为广度优先搜索。

不同算法为迷宫求解提供了不同路径搜索策略，能够适应不同规模和复杂度的迷宫。

6. 用户交互

在本项目中，用户通过终端进行交互，操作流程清晰直观。程序会提示用户输入迷宫的尺寸，然后在迷宫生成后提供算法选择，用户可以选择不同的求解方法来解迷。

1. **输入迷宫尺寸**
   - 程序会提示用户输入迷宫的高度和宽度，范围为 `[5-99]`。用户需要输入一个有效的数字，超出范围或输入非法字符会被提示重新输入。
   - 示例提示：

```
请输入迷宫的高度[5-99]：
请输入迷宫的宽度[5-99]：
```

2. **选择求解算法**
   - 用户可以选择不同的算法进行迷宫求解，选项包括回溯法（递归）、基于栈的深度优先搜索（DFS）、广度优先搜索（BFS）和 A* 算法。
   - 输入非法选项时，程序会提示重新选择。
   - 示例选项：

```
请选择迷宫算法：[1]回溯法(递归)    [2]DFS法(栈)    [3]BFS法
[4]A星算法    [0]退出
```

3. **显示结果和路径**
   - 每次算法求解后，程序会以字符图的形式可视化迷宫，并显示解路径，标识入口、出口和通路位置。
   - 解路径使用特定字符（如'·'）进行标记，帮助用户直观地了解迷宫的解决过程。
   - 用户可以继续选择其他算法，或者输入 `0` 退出程序。

4. **程序退出**
   - 输入 `0` 时，程序会输出确认消息，表示退出成功。
   - 示例消息：

```
程序成功退出！
```

7. 性能考虑

为优化迷宫求解算法的性能，项目中采取了多种措施，尤其是在处理大规模迷宫时。

1. **算法选择的适配性**
   - 回溯法适合小规模迷宫，大迷宫可能导致递归过深。
   - DFS 和 BFS 算法时间复杂度为 `O(V + E)`，适合中等规模迷宫。
   - A* 算法适用于大规模迷宫，能够快速找到最优路径。
2. **内存管理的优化**
   - 自定义的栈、队列和优先队列数据结构进行了内存管理优化，减少动态分配次数，避免内存碎片化问题。
   - 每次求解前调用 `ClearPath()` 重置迷宫状态，减少不必要的内存分配和释放。
3. **复杂度分析**
   - **回溯法**：时间复杂度较高，当迷宫较大时可能导致性能问题。
   - **DFS/BFS**：对于稀疏图和较小的迷宫性能良好。
   - **A*** 算法：使用启发函数使搜索更加高效，适用于大型复杂迷宫。
4. **并发和异步处理**
   - 可进一步考虑引入多线程优化，以提高算法的运行效率。

8. 未来的改进

为了增强项目的功能和适应性，可以进行以下改进：

1. **引入多种迷宫生成算法**
   - 除了现有的深度优先搜索生成迷宫，还可以加入其他算法，如 Prim 或 Kruskal 算法，以生成不同类型的迷宫。
2. **图形化可视化**
   - 目前的字符可视化可以进一步改进为图形化界面，使用图形库（如 SDL、Qt 或 OpenGL）实现更直观的显示效果。
3. **改进算法**
   - 为 A* 算法引入更高效的数据结构（如斐波那契堆），提升优先队列的操作效率。
   - 使用双向 BFS（从起点和终点同时进行搜索），减少搜索空间。
4. **提高用户交互的多样性**
   - 增加难度设置，让用户可以选择迷宫的复杂程度。
   - 增加实时动画展示路径搜索过程，增强用户体验。
5. **支持更大规模的迷宫**

- 采用分块处理或递归分解的技术，优化算法以处理超大规模的迷宫。

9. 心得体会

这个项目本来要求是比较简单的，但我觉得这是一个很好的练习搜索算法的机会，所以做了很多拓展。每个搜索算法都会有相应的数据结构，在写这些数据结构的模版的过程中，我对各种线性表（向量，队列，栈，堆）有了更深刻的理解。在写不同算法的时候，我体会到了不同算法的差异，以及他们在时间和空间复杂度上的优缺点。我相信这个项目会对以后得数据结构学习和代码学习有很大帮助。