苏 州 大 学 实 验 报 告

院、系	计算机学院	年级专业	19 计算机类	姓名	张昊	学号	1927405160
课程名称	数据结构课程实践					成绩	
指导教师	孔芳	同组实验者	无		实验日期	2021 年 1 月 3 日	

实 验 名 称 迷宫求解

一、问题描述及要求

迷宫求解

一般的迷宫可表示为一个二维平面图形，将迷宫的左上角作为入口，右下角作为出口。迷宫问题求解的目标是寻找一条从入口点到出口点的通路。

例如，可设计一个 8×8 矩阵 maze[8][8]来表示迷宫，如下所示

```
0 1 0 0 0 0 1 1
0 0 0 1 0 0 1 0
1 0 1 0 1 0 1 1
1 0 1 0 1 1 0 1
0 1 1 1 1 1 1 0
1 0 0 1 1 0 0 0
1 0 1 0 0 0 1 1
1 0 1 1 0 1 0 0
```

左上角 maze[0][0]为起点，右下角 maze[7][7]为终点；设“0”为通路，“1”为墙。假设一只老鼠从起点出发，目的为右下角的终点，可向“上、下、左、右、左上、左下、右上、右下”8 个方向行走。

设计一个程序，能自动生成或手动生成一个 8×8 矩阵，针对这个矩阵，程序判断能否从起点经过迷宫走到终点。如果能，请给出一种走出迷宫的路径。

二、概要设计

1. 问题简析

本次实验内容是设计程序解决迷宫求解问题。实验需要实现迷宫在计算机中的表示以及给出计算机求解迷宫问题的步骤。

2. 系统功能列表

程序要实现如下功能：

1. 读取或生成一个 8*8 迷宫；
2. 对读取或生成的迷宫求解通路；
3. 输出迷宫的解。

3. 程序界面设计

程序启动，首先输出提示信息。

```
Welcome to Maze solution. Author: Zhang Hao
You can input the maze or generate it randomly (using python script 'generator.py' in advance).
It is a 0-1 matrix of 8 rows and 8 columns, with each number separated by a space.
"0" is the access path and "1" is the wall.
The program will judge whether it can go from the beginning(left-up) to the end(right-down).
If it can, it will show you a path out of the maze.
"*" means the access path, "#" is the dead end.
```

之后会询问用户手动输入还是随机生成。

```
Do you want to input the maze manually? [Y/n]y
```

若选择手动输入，则程序会要求用户输入一个 8*8 的 0-1 矩阵作为迷宫。

```
Now input the 8*8 Maze:
0 1 1 1 1 0 0 1
1 0 1 1 0 1 0 1
0 1 0 0 1 1 1 0
0 1 1 1 0 1 0 1
1 1 1 0 1 1 0 1
1 1 0 1 1 0 1 1
0 0 0 1 1 0 1 1
1 1 1 1 1 1 0 0
```

若选择随机生成，则程序会询问是否需要立即生成还是读取已有的“maze.init”文件。

```
Do you want to input the maze manually? [Y/n]n
Do you want to generate maze now? [Y/n]y
```

迷宫读取完成后，程序将输出原始迷宫并求解，若存在解则输出迷宫的解，否则输出提示信息。

```
The origin maze is:
0 1 1 1 1 0 0 1
1 0 1 1 0 1 0 1
0 1 0 0 1 1 1 0
0 1 1 1 0 1 0 1
1 1 1 0 1 1 0 1
1 1 0 1 1 0 1 1
0 0 0 1 1 0 1 1
1 1 1 1 1 1 0 0
```

```
The problem solved, solution is:
* 1 1 1 1 * * 1
1 * 1 1 * 1 * 1
0 1 * * 1 1 1 *
0 1 1 1 # 1 * 1
1 1 1 # 1 1 * 1
1 1 # 1 1 * 1 1
# # # 1 1 * 1 1
1 1 1 1 1 1 * *
```

4. 程序结构设计思路

程序以二维数组 maze[height+2][width+2]表示迷宫，其中为了简化操作，在迷宫四周添加一圈障碍 maze[0][j]， maze[height+1][j] (0≤j≤height+1)， maze[i][0] 以及 maze[i][width+1] (0≤i≤width+1)。数组中的元素类型为字符型，0 表示通路，1 表示障碍。限定迷宫的大小为 8*8。

用户可以选择输入迷宫的数据，为 8 行 8 列的 0-1 矩阵，每个数字之间用空格隔开；也可以选择程序自动生成随机的迷宫。迷宫的入口位置设定为左上角，出口位置

设定为右上角。

若给定或生成的迷宫存在通路，则以矩阵形式将迷宫及其通路输出到屏幕上，其中，字符“1”表示障碍，字符“*”表示找到通路路径的位置，字符“#”表示“死胡同”，即程序经过的但不能到达出口的位置，字符“0”表示通路但没有经过的位置。若设定的迷宫不存在通路，则报告相应信息，并输出程序试探的路径，用字符“#”表示。

本程序只求出条成功的通路。然而，只需要对迷宫求解的函数作简单修改，便可求得全部路径。

三、详细设计

1 迷宫数据类型定义

程序以二维数组 maze[height+2][width+2]表示迷宫，在迷宫四周添加一圈障碍 maze[0][j]， maze[height+1][j] (0<j<height+1)， maze[i][0] 以及 maze[i][width+1] (0≤i≤width+1)。数组中的元素类型为字符型，0 表示通路，1 表示障碍，查找通路后字符“*”表示找到通路路径的位置，字符“#”表示 “死胡同”，即程序经过的但不能到达出口的位置。将二维数组和操作包装为迷宫类，共主函数调用。

另在类中添加两个字段 walked 和 found，分别表示是否查找过迷宫的解以及迷宫是否有解；程序复用了 C++标准库中 std::pair 类型表示迷宫中的坐标位置。

迷宫类的数据类型定义如下：

```
1.  class Maze {
2.   public:
3.    Maze(): walked_(false), found_(false) { }
4.    explicit Maze(bool from_file);
5.    void initialize(bool from_file = false);
6.    void show(ostream &output) const;
7.    bool walk();
8.
9.   private:
10.   using Point = std::pair<int, int>;
11.   char maze_[MAZE_HEIGHT + 2][MAZE_WIDTH + 2]{};
12.   bool walked_;
13.   bool found_;
14.   void setup(istream &input);
15.   bool do_walk(const Point &point);
16. };
```

另外定义了一些宏来表示常量：

```
1.  #define MAZE_WIDTH  8
2.  #define MAZE_HEIGHT 8
3.  #define MAZE_PATH   '0'
4.  #define MAZE_WALL   '1'
5.  #define MAZE_DEAD   '#'
6.  #define MAZE_RESULT '*'
```

2 迷宫的读取或生成与输出

使用 Python 脚本来实现迷宫的随机生成，保存在“maze.init”文件中。实现方法：

```
maze = [[str(randint(0, 1)) for _ in range(MAZE_WIDTH)]  for _ in range(MAZE_HEIGHT)]
maze[0][0] = maze[-1][-1] = '0'
```

无论是从标准输入读取，还是从文本文件读取迷宫数据，程序都将其抽象为输入流对象，通过 C++提供的输入流方法依次读取 8*8 个整数，将其存储在字符型的二维数组中。

迷宫的输出只需将二维数组除去四周的障碍后输出即可，即输出 maze [1][1]..maze[height][width]的部分。

3 迷宫求解算法

迷宫求解算法（walk）的基本思想是图的深度优先搜索：假设初始状态是图中所有顶点均未被访问，则从某个顶点 v 出发，首先访问该顶点，然后依次从它的各个未被访问的邻接点出发深度优先搜索遍历图，直至图中所有和 v 有路径相通的顶点都被访问到。显然，深度优先搜索是一个递归的过程。深度优先遍历特点是，选定一个出发点后进行遍历，能前进则前进，若不能前进，回退一步再前进，或再回退一步后继续前进。依此重复，直到所有与选定点相通的所有顶点都被遍历。

具体地，在迷宫中认为 0 是图的顶点，其上、下、左、右、左上、左下、右上、右下 8 个方向上为 0 的点为与其邻接的顶点。选定右侧为前进的起始方向，按顺时针方向依次访问各个为 0 的顶点，进行递归式的前进，直到无路可走，若为终点则返回 true，否则返回 false。

在具体实现上，求解算法设计一个辅助函数（do_walk），在辅助函数中递归调用。函数只有在给定的位置是可走的（即为 0）前提下进行递归操作，如果当前位置为终点则结束递归。首先计算得到 8 个方向的位置，置访问标志后之后按照右、右下、下、左下、左、左上、上、右上的顺序依次递归调用自己，并记录每次调用的返回值，若为 true 则结束递归，返回 true，否则继续搜索，若全为 false 则认为当前位置是“死胡同”的一部分，更换为死胡同的标志。

具体算法可由如下代码表示。

```
bool Maze::do_walk(const Point &point) {
  if (maze_[point.first][point.second] == MAZE_PATH) {
    if (point.first == MAZE_HEIGHT && point.second == MAZE_WIDTH) {
      return true;
    }
    Point next[]{
        {point.first, point.second + 1},      // right
        {point.first + 1, point.second + 1},  // right-down
        {point.first + 1, point.second},      // down
        {point.first + 1, point.second - 1},  // left-down
        {point.first, point.second - 1},      // left
        {point.first - 1, point.second - 1},  // left-up
        {point.first - 1, point.second},      // up
        {point.first - 1, point.second + 1}   // right-up
```

```
    };
    maze_[point.first][point.second] = MAZE_RESULT;
    int cnt = 0;
    for (const auto &item : next) {
      if (do_walk(item)) { return true; }
      else { cnt++; }
    }
    if (cnt >= 8) {
      maze_[point.first][point.second] = MAZE_DEAD;
    }
  }
  return false;
}
```

四、实验结果测试与分析

首先使用两组数据，以手动输入的方式来测试算法的正确性。

第一组数据为题目中给出的数据，运行结果如下：

```
The origin maze is:
0 1 0 0 0 0 1 1
0 0 0 1 0 0 1 0
1 0 1 0 1 0 1 1
1 0 1 0 1 1 0 1
0 1 1 1 1 1 1 0
1 0 0 1 1 0 0 0
1 0 1 0 0 0 1 1
1 0 1 1 0 1 0 0
The problem solved, solution is:
* 1 0 0 0 0 1 1
0 * * 1 * * 1 0
1 0 1 * 1 * 1 1
1 0 1 # 1 1 * 1
0 1 1 1 1 1 1 *
1 0 0 1 1 0 * *
1 0 1 0 0 * 1 1
1 0 1 1 0 1 * *
```

输出符合预期。可以看出在第四行第四列存在一个位置为程序遍历到的但是无路可走的路径，即“死胡同”。

第二组数据以及结果如下：

```
The origin maze is:
```

```
0 1 1 1 1 0 0 1
1 0 1 1 0 1 0 1
0 1 0 0 1 1 1 0
0 1 1 1 0 1 0 1
1 1 1 0 1 1 0 1
1 1 0 1 1 0 1 1
0 0 0 1 1 0 1 1
1 1 1 1 1 1 0 0
The problem solved, solution is:
* 1 1 1 1 * * 1
1 * 1 1 * 1 * 1
0 1 * * 1 1 1 *
0 1 1 1 # 1 * 1
1 1 1 # 1 1 * 1
1 1 # 1 1 * 1 1
# # # 1 1 * 1 1
1 1 1 1 1 1 * *
```

输出符合预期。这一组数据存在一条通向“死胡同”的路径。

之后测试了随机生成迷宫并求解。由于生成的迷宫是完全随机的，故一道测试了迷宫无解时的输出。下面选取两个有代表性的例子作简单说明。

```
Do you want to input the maze manually? [Y/n]n
Do you want to generate maze now? [Y/n]y
The origin maze is:
0 0 1 1 1 1 1 0
0 0 0 1 1 1 0 1
0 0 0 0 0 0 0 1
1 0 1 1 1 0 0 0
1 1 1 0 1 0 1 1
0 1 0 1 1 0 0 0
0 1 1 0 1 1 1 0
0 0 1 1 1 1 0 0
The problem solved, solution is:
* * 1 1 1 1 1 0
0 0 * 1 1 1 0 1
0 0 0 * * * * 1
1 0 1 1 1 0 * *
```

```
1 1 1 0 1 * 1 1
0 1 0 1 1 0 * *
0 1 1 0 1 1 1 *
0 0 1 1 1 1 0 *
```

这个迷宫有解，但是不是最优解，由于深度优先搜索的策略是右、右下、下、……的顺序，导致第 4 行第 8 列多走了看似不必要的一步。

```
Do you want to input the maze manually? [Y/n]n
Do you want to generate maze now? [Y/n]y
The origin maze is:
0 1 0 1 0 1 1 1
0 0 1 1 1 0 1 1
0 0 1 1 0 1 1 1
1 0 1 0 1 1 1 0
1 0 1 1 0 0 0 0
1 0 1 1 1 1 1 1
0 1 0 0 1 1 1 1
1 1 0 1 1 0 0 0
There is no solution to the problem, attempted path is:
# 1 # 1 0 1 1 1
# # 1 1 1 0 1 1
# # 1 1 0 1 1 1
1 # 1 0 1 1 1 0
1 # 1 1 0 0 0 0
1 # 1 1 1 1 1 1
# 1 # # 1 1 1 1
1 1 # 1 1 0 0 0
```

这个迷宫无解，因为存在者三个不连通的分支，算法遍历了左侧的连通分支上的所有可达点都没有到达终点，故全为“死胡同”。

五、小结

通过这次实验，我复习并熟练了图的深度优先遍历的实现方法，对图的邻接矩阵存储方式有了进一步的认识。

六、附录

1. **源代码路径**：C++源代码位于附件中 src 目录下。
2. **文件编码**：UTF-8；**行分隔符**：CRLF。

3. **实验环境**：Windows 操作系统，MSVC 编译器，基于 cmake 构建；在 CLion 集成开发环境中调试运行通过。手动编译运行方法为（在 Linux/Unix shell 中）：

```
$ mkdir build  # pwd: src/
$ cd build
$ cmake ..
$ make
$ cd ..
$ ./build/solution
```