燕山大学

**课 程 设 计 报 告**

**题目：** 多重算法实现的迷宫求解问题

学院（系）**： 理学院**

年级专业**： 19级信息与计算科学**

组 号**： 第14组**

指导教师**：**

**燕山大学课程设计（论文）任务书**

院（系）： 理学院 基层教学单位： 信息与计算科学系

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 学 号 |  | | 学生姓名 |  | 专业（班级） | |  |
| 设计题目 | 多重算法实现的迷宫求解问题 | | | | | | |
| 设  计  技  术  参  数 | 在本项目中，主要使用了以下数据结构和存储方式：  （1）顺序栈 SqStack  （2）链栈 LinkStack  （3）图的邻接表 ALGraph  （4）队列 Queue | | | | | | |
| 设  计  要  求 | 1.问题重述：  “迷宫求解”是指在规定的迷宫中寻找从入口到出口路径的问题。即从入口出发，顺着某一方向向前探索，若能走通，则继续向前走；否则沿原路退回，换一个方向继续探索，直到探索到所有可能连通的路径为止。  2.算法设计：  （1）回溯法  （2）基于深度遍历的算法  （3）基于广度遍历的算法  （4）转换为图的图论算法 | | | | | | |
| 工作量 | 饱满 | | | | | | |
| 工  作  计  划 | 周一：讨论并确定报告选题为——多重算法实现的迷宫求解问题  周二：确定四种问题解决方法并进行分工  周三：编写代码  周四：编写代码并进行论文撰写  周五：编写代码并进行答辩PPT的制作  周六：检查论文和答辩PPT并进行修改，准备答辩内容  周天：准备答辩 | | | | | | |
| 参  考  资  料 | [1]严蔚敏,吴伟民.数据结构(C语言版)[M].北京:清华大学出版社,2007.  [2]小小花梦.《算法笔记》——“迷宫求解”之广度优先搜索(BFS)[DB/OL].  <https://blog.csdn.net/weixin_42100963/article/details/105407923>? | | | | | | |
| 指导教师签字 | |  | | 教学单位主任签字 | |  | |

说明：此表一式四份，学生、指导教师、基层教学单位、系部各一份。

2021年 6月 6 日

**燕山大学课程设计评审意见表**

|  |
| --- |
| **指导教师评语：**  **成绩：**  **指导教师：**    2021年 6 月 6 日 |
| **答辩小组评语：**  **成绩：**  **组内自评成绩：**  2021年 6 月 6 日 |
| **课程设计总成绩：** |
| **答辩小组成员签字：**    2021年 6 月 6 日 |

多重算法实现的迷宫求解问题

一、课设内容

1.1题目背景

迷宫问题是取自心理学的一个古典实验。在该实验中，将一只老鼠放入一个无顶大盒子的门口处，在出口处放置一块奶酪，奶酪吸引老鼠在盒子中寻找出口。对同一只老鼠进行反复实验，最终老鼠学会走通迷宫路线并不走错一步。

1.2题目重述

“迷宫求解”是指在规定的迷宫中寻找从入口到出口路径的问题。即从入口出发，顺着某一方向向前探索，若能走通，则继续向前走；否则沿原路退回，换一个方向继续探索，直到探索到所有可能连通的路径为止。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

1.3题目分析

解决迷宫问题，首先需要利用一种方式，如二维数组将其存储起来。由于计算机解迷宫时，常用的是“穷举求解”的办法，即从入口出发，顺某一方向向前探索，若能走通，则继续往前走；否则沿原路返回，换一个方向继续探索，直至所有可能的通路都探索为止。为了保证在任何位置上都能按原路返回，显然需要用一个后进先出的结构来保存从入口到当前位置的路径。

1.4算法设计

（1）回溯法

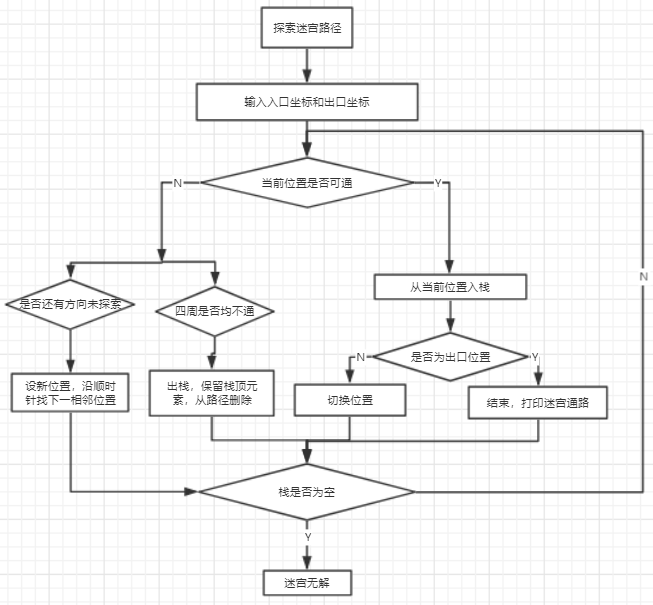
（2）基于深度遍历的算法

（3）基于广度遍历的算法

（4）转换为图的图论算法

1.5程序流程图

以回溯法为例，构造流程图如下：



二、课设步骤

（1）讨论并确定项目主题，拟解决多重算法设计的迷宫求解问题

（2）查找资料，讨论交流，确定项目使用的算法和思路

（3）分工实现不同算法，求解迷宫问题

（4）针对不同迷宫求解原理展开讨论，改进完善、创新提高

（5）项目总结，整理课设资料和成果，撰写报告书，准备结题答辩

三、算法思想

3.1回溯法

回溯法是求解迷宫问题的基础算法之一，其基本思路是从入口出发，顺着某一方向向前探索，若能走通，则继续向前走；否则沿原路退回，换一个方向继续探索，直到探索到所有可能连通的路径为止。为了保证在任何位置上都能沿原路退回，显然需要用一个先进后出的的存储结构来保存迷宫求解中每一步走过的路径。因此我们使用栈结构来存储。下面对算法中几处基本的程序算法做简要说明：

（1）迷宫图的创建和表示

存储迷宫图我们需要两个数组，一个用来构建迷宫，通过用户逐项输入“0”和“1”来表示通路和障碍；另一个数组用来存储探索迷宫中记录的路径（以“\*”表示，下同），障碍点（“#”），死胡同（“@”）。

（2）Stack存储结构

在使用Stack结构存储探索步骤时，我们可以选用顺序栈SqStack或链栈LinkStack存储。需要注意的是，栈中存储的元素是一个含有迷宫位置、下一探索方向和探索步骤的结构体。

（3）回溯法求解迷宫

设定当前位置的初值为入口位置，基本算法思路可表示为：

do{

若当前位置可通，则｛

将当前位置插入栈顶；//纳人路径  
若该位置是出口位置，则结束； //求得路径存放在栈中  
否则切换当前位置的东邻方块为新的当前位置；

｝  
否则，  
若栈不空且栈顶位置尚有其他方向未经探索，  
 则设定新的当前位置为沿顺时针方向旋转找到的栈顶位置的下一相邻块；

若栈不空但栈顶位置的四周均不可通，

则｛

删去栈顶位置；//从路径中删去该通道块  
若栈不空，则重新测试新的栈顶位置，  
直至找到一个可通的相邻块或出栈至栈空；

｝  
｝while(栈不空）；  
 在此，尚需说明一点的是，所谓当前位置可通，指的是未曾走到过的通道块，即要求该方块位置不仅是通道块，而且既不在当前路径上（否则所求路径就不是简单路径），也不是曾经纳入过路径的通道块（否则只能在死胡同内转圈）。

3.2基于深度遍历迷宫求解算法

深度优先搜索遍历法是求解迷宫问题的基础算法之一，其基本思路是构建一个两重循环，内层循环基于当前位置对下一个位置的方向进行判断，若符合要求则将目前位置坐标存储，并将处理的坐标移到下一个位置；外层循环则是当前位置无法继续前进，需要回到上一位置，对其它方向进行尝试。为了保证在每一个当前位置都能返回上一位置，显然需要用一个先进后出的的存储结构来保存迷宫求解中程序试探成功的每一个位置。因此我们使用栈结构来存储。下面对算法中几处基本的程序算法做简要说明：

（1）迷宫图的创建和表示

存储迷宫图我们需要一个数组，一个用来构建迷宫，通过用户逐项输入“0”和“1”来表示通路和障碍，程序在探索过的位置用“-1”表示，最后再在一个拷贝好的初始图上以“9”画出找到的通路。

（2）存储结构

在使用Stack结构存储探索步骤时，我们选用顺序栈SqStack存储。需要注意的是，栈中存储的元素是含有迷宫位置和对应方向的结构体，对应方向用一个方向增量结构体来存储X和Y的增量，表示四个方向。

（3）深度优先搜索遍历法求解迷宫

设定左上角为入口位置，基本算法思路可表示为：

初始化栈，事先压入一个元素，作为初始位置；

while(栈不空）{

栈弹出元素，当前位置返回上一位置；

给该元素一个向右的方向；

While(方向未尝试完){

当前坐标通过方向的坐标增量初始化下一位置；

if(下一位置可通过){

将当前位置和方向压入栈；

将当前位置移动到下一位置；

if(当前位置是右下角终点) return 迷宫有通路；

else 方向初始化向右；  
 }

else 更改方向；  
 }  
 }

return 迷宫没有通路；

在此，说明一下事先压入栈的元素的意义：下面外层循环中第一次回退操作将当前位置定义为初始位置，这个回退操作在遇到错误的路径时用于回退到上一位置，在上一位置寻找新方向。

3.3基于广度遍历迷宫求解算法

3.1.1 解法一：

广度优先搜索遍历也是求解迷宫问题的基础算法之一，类似于树的按层次遍历的过程。与深度优先遍历有所不同的是，广度优先遍历是先遍历层级在前的元素再遍历层级在后的元素。广度遍历求解迷宫问题思路是将目前的所有的可能放入队列之中，然后再去掉队列的头部。如图所示，将(0,0)作为队列的头，然后将该点四个方向所有的点加入队列之中(除去已加入的点、障碍点以及除了迷宫的点)，然后去掉队列的头，将下一个点作为新的队列的头，继续进行新的遍历，直至找到终点。

**0 1 2 3 4**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

下面对算法中几处基本的程序算法做简要说明：

（1）迷宫图的创建和表示

存储迷宫图我们需要一个数组，通过编辑代码，让用户自己构造自己想要的迷宫，用“0”表示通路，用非零实数如“1”表示障碍点。程序对于探索过的位置用“-1”表示，最后输出一个以空格为通路的，以“[]”为障碍物的迷宫。。

（2）广度优先搜索遍历法求解迷宫

设定左上角(0,0)为入口位置，基本算法思路可表示为：

构造迷宫；声明队列的元素类型；定义队列，并初始化队头元素；编辑核心部分，开始BFS。

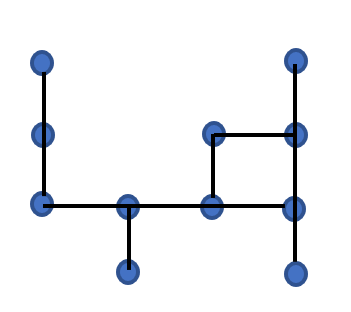
3.3.2解法二：

广度遍历的主要思路是：在每一个节点处都找到与其可以联通的所有路径，进而一层一层得寻找下去，直到找到出口。就类似于我们找朋友帮忙，如果朋友无法解决的话，他就会去找他的朋友寻求帮助一样。而实现这个功能时，我们借助寻路机器人实现，在迷宫求解时，每释放出一个机器人，其会向每个联通方向释放相同的寻路机器人(因而在判断时使用四个if语句而非if…else语句)。  
      在求解问题时，首先将字符迷宫转换为数字迷宫，通过switch语句记录每一步位置的坐标，进而显示出步数。最后判断机器人寻得的坐标是否与给出的迷宫终点坐标一致，若一致，则说明寻路成功。而在判断是否一条路无法走通时，则是通过创建一个函数来记录放出机器人的个数，当某条路径的机器人个数不再增加时，则说明该路已到尽头，无法走通。

3.4基于图论的求解思想

在求解迷宫问题时，我们可以从迷宫本身的几何性质出发，利用图论的一些原理和方法去寻找不同的求解方式。在这里我们定义迷宫中输入元素为0的点为“可达点”，输入元素为1的点为“不可达点”。

对于一个的迷宫图，我们可以将其中的可达点转化为无向图中顶点集中的元素，不可达点忽略不计。我们假设无向图的点集是按顺序整齐排列的，此时对于任意一可达点，可达点只能与上下左右四个方向的可达点是连通的，并且连通各可达点之间的边构成无向图的边集。这样就完成了一个的迷宫图向一个无向图的转化，并且这种转换关系是一一对应的，例如迷宫

可转化为：

迷宫问题是求解两点之间是否存在通路的问题，转换为图论问题后可以变为两顶点之间是否连通问题。若去求解迷宫的最佳路径，即可转换为图论中最短路径问题，可以利用Dijkstra算法或Floyd算法求解，这里就不再赘述了。下面具体说明图的连通性求解。

我们注意到，在实际的迷宫求解问题中，无向图并不都是连通图，但无向图可以分为多个连通分量，这是我们可以将无向图划分为若干连通子图若我们求解的两点正位于其中的一个连通子图上，那可以说明其存在连通的通路。在程序设计中，我们可以简化利用深度遍历和广度遍历判断连通性的思想，直接利用邻接表进行判断。该问题的主要难点也就变为如何将的迷宫图转换为用邻接表表示的无向图。

因为可达点与无向图中的点是一一对应的，因此我们可以直接输入有连接的可达点，完成图的邻接表创建。最后输出图的邻接表，就可以判断任意两点间是否存在通路。

此外，利用最小生成树、粒子群算法、遗传算法、蚁群算法等很多方式均可求解迷宫问题，随着迷宫问题的限制逐渐增加，我们可以考虑利用这些更加复杂的算法进行求解。

四、程序代码与结果显示

4.1回溯法

4.1.1代码展示

在这里只展示顺序栈SqStack中部分重要代码，全部代码详见附录：

void InitMaze(MazeType& maze) {

for (int i = 0; i < row + 2; i++) {

for (int j = 0; j < col + 2; j++) {

if (i == 0 || j == 0 || i == row + 1 || j == col + 1) {

maze.map[i][j] = 0;

maze.arr[i][j] = '#';

}

else {

cout << "第" << i << "行" << "第" << j << "列：";

cin >> maze.map[i][j];

if (maze.map[i][j] == 1) maze.arr[i][j] = '#';

else maze.arr[i][j] = ' ';

}

}

}

}

Status Pass(MazeType maze, PostType curpos) {//判断格子是否走过且能走

if (maze.arr[curpos.x][curpos.y] == ' ') return OK;

return ERROR;

}

Status FootPrint(MazeType& maze, PostType curpos) {//记录走过的格子

if (maze.arr[curpos.x][curpos.y] == ' ') {

maze.arr[curpos.x][curpos.y] = '\*';

return OK;

}

return ERROR;

}

Status Same(PostType curpos, PostType end) {//判断两个格子位置是否一样

if (curpos.x == end.x && curpos.y == end.y)

return OK;

return ERROR;

}

PostType NextPos(PostType curpos, int di) {

switch (di)

{

case 1: {curpos.y = curpos.y + 1; return curpos; break; }//向右移动一格

case 2: {curpos.x = curpos.x + 1; return curpos; break; }//向下移动一格

case 3: {curpos.y = curpos.y - 1; return curpos; break; }//向左移动一格

case 4: {curpos.x = curpos.x - 1; return curpos; break; }//向上移动一格

}

}

Status MarkPrint(MazeType& maze, PostType pos) {

maze.arr[pos.x][pos.y] = '@';

return OK;

}

Status MazePath(MazeType& maze, PostType start, PostType end) {

Stack S; InitStack(S);

PostType curpos = start;//设定“当前位置”为“入口位置”

int curstep = 1;//探索第一步,记录探索步数

bool found = ERROR;//判断是否达到终点

ElemType e;//Stack元素

do {

if (Pass(maze, curpos)) {

//当前位置可以通过，即是未曾走到过的通道留下足迹

FootPrint(maze, curpos);

e.di = 1; e.seat = curpos; e.step = curstep;

Push(S, e);//加入路径

if (Same(curpos, end)) found = OK; //到达终点（出口）

else {

curpos = NextPos(curpos, 1);//下一位置是当前位置的东邻（向右移动一格）

//NextPos(curpos, 1);

curstep++;//探索下一步

}

}

else {//当前位置不能通过

if (!StackEmpty(S)) {

Pop(S, e);//将刚才不能前进的Stack元素出栈，相当于退回一步

while (e.di == 4 && !StackEmpty(S)) {

MarkPrint(maze, e.seat);//对刚才出栈的元素的位置标记

Pop(S, e);//下一个元素出栈，与上一步的e不一样

}

if (e.di < 4) {

e.di++; Push(S, e);//换下一个方向探索

curpos = NextPos(e.seat, e.di);//设定当前位置是该新方向向上

}

}

}

} while (!StackEmpty(S) && !found);

return found;

}

Status PrintMaze(MazeType maze) {

for (int i = 0; i < RANGE + 2; i++) {

for (int j = 0; j < RANGE + 2; j++) {

cout << maze.arr[i][j] << " ";

}

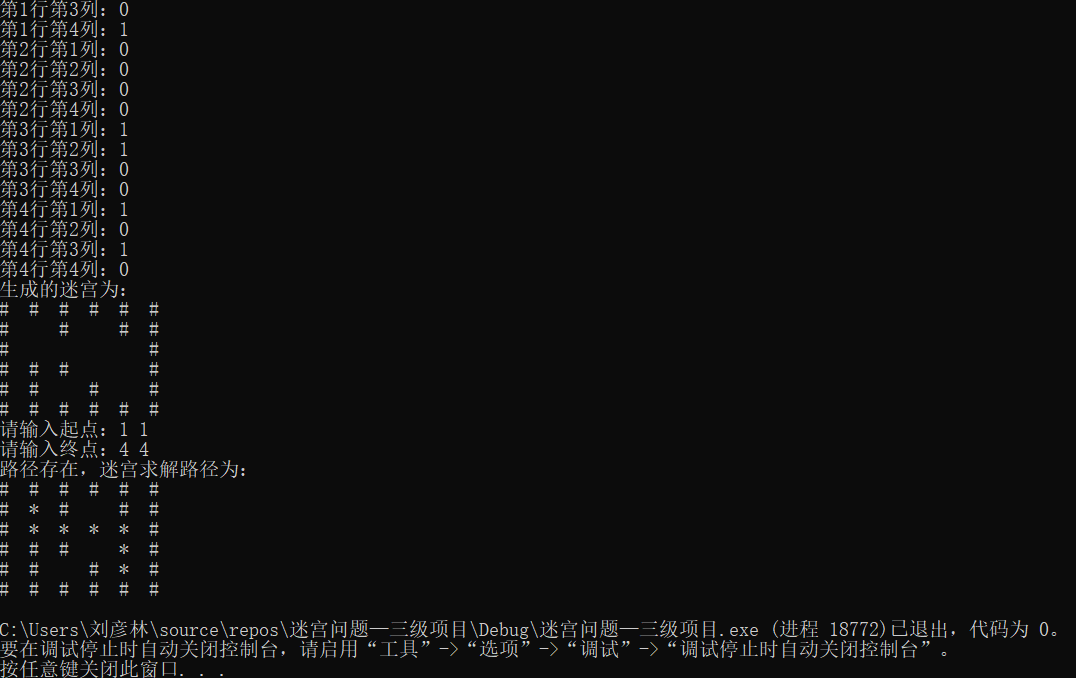
cout << "\n";

}

return OK;

}

4.1.2运行结果



4.2基于深度遍历迷宫求解算法

4.2.1 代码展示

在这里只展示寻找路径的重要代码，全部代码详见附录：

Status findPath(int maze[][N + 2],Direction direct[], SqStack &S)

{

Box temp;

int x, y, di;

int L, C;

maze[1][1] = -1;

temp.x = 1;

temp.y = 1;

temp.di = -1;

Push(S, temp);//将temp压入到堆栈

while (!StackEmpty(S))//当栈不空的时候

{

temp = Pop(S);

x = temp.x; y = temp.y; di = temp.di + 1;

while (di < 4)

{

L = x + direct[di].incX;

C = y + direct[di].incY;

if (maze[L][C] == 0)

{

temp.x = x;

temp.y = y;

temp.di = di;

Push(S, temp);

x = L;

y = C;

maze[L][C] = -1;

if (x == M && y == N)

{

return OK;

}

else

{

di = 0;

}

}

else

{

di++;

}

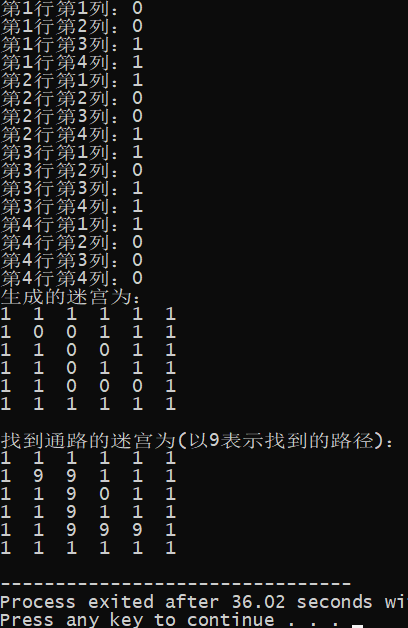
}

}

return ERROR;

}

4.2.2运行结果



4.3基于广度遍历迷宫求解算法

4.3.1代码展示

在这里只展示寻找路径的重要代码，全部代码详见附录：

int main()

{

for (int i = 0; i < 100; i++)

for (int j = 0; j < 100; j++)

flag[i][j] = 1;

cout << "你想要构造迷宫的长度为:" << endl;

cin >> ROW\_NUM;

cout << "你想要构造迷宫的宽度为：" << endl;

cin >> COL\_NUM;

cout << "以0为通路，以非零实数，如1为墙壁，构造你想要的迷宫：" << endl;

int m = 1;

for (int i = 0; i < ROW\_NUM; i++)

{

cout << "请输入第" << m << "行的数字" << endl;

for (int j = 0; j < COL\_NUM; j++)

{

cin >> nMatrix[i][j];

flag[i][j] = 0;

}

if (m < ROW\_NUM)

{

m++;

}

}

cout << "你构造的迷宫为：" << endl;

for (int i = 0; i < ROW\_NUM; i++){

for (int j = 0; j < COL\_NUM; j++){

if (nMatrix[i][j] == 0)

cout << " ";

else cout << "[ ]";

}

cout << endl;

}// 输出二维迷宫图

// 初始化每一个队列元素的值（开始寻找的地方）

que[head].x = que[head].y = 0;

tail++; // 尾部索引 + 1

flag[0][0] = 1; // 标记初始位置已经查找过了

cout << "它走过的路径为：" << endl;

cout << "(0,0)" << endl;

while (head < tail)

{

int i = 0;

for (; i < ROW\_NUM; i++){

// 每次都是以队头为单位向四个方向开始

int nx = que[head].x + direction[i][0];

int ny = que[head].y + direction[i][1];

if (nx < 0 || nx > ROW\_NUM || ny < 0 || ny > COL\_NUM)

continue;

if (nMatrix[nx][ny] == 0 && flag[nx][ny] == 0)

{

flag[nx][ny] = 1;

que[tail].x = nx; // 加入队列

que[tail].y = ny;

cout << "("<<que[tail].x <<","<< que[tail].y<<")" << endl;

tail++;

}

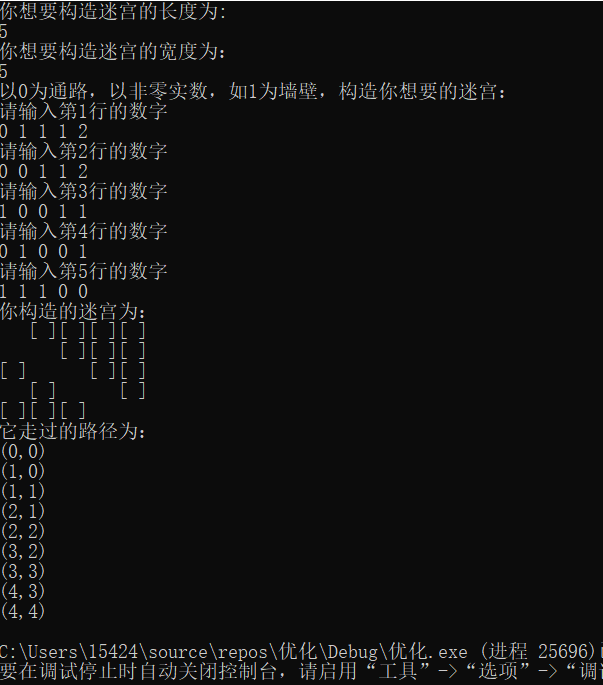
}

head++;

}

return 0;

}

4.3.2运行结果

4.3.3代码展示

在这里只展示寻找路径的重要代码，全部代码详见附录：

void findRoad(int x, int y, int direction)

{

Sleep(WAIT\_TIME);

road\_num++;

/\*\*/

if (flag == 1)//寻路完成提前退出

return;

/\*记录此地距离原点距离\*/

switch (direction)

{

case 1: {

maze\_road[x][y] = maze\_road[x - 1][y] + 1;

break;

}

case 2: {

maze\_road[x][y] = maze\_road[x + 1][y] + 1;

break;

}

case 3: {

maze\_road[x][y] = maze\_road[x][y - 1] + 1;

break;

}

case 4: {

maze\_road[x][y] = maze\_road[x][y + 1] + 1;

break;

}

default:

break;

}

/\*寻路成功，标记并退出\*/

if (x == aim\_x && y == aim\_y)

{

flag = 1;

return;

}

/\*向四个方向，有路则释放新的机器人（创建新线程）\*/

if (maze\_road[x - 1][y] == 0)

{

thread find\_road\_up(findRoad, x - 1, y, 2);

find\_road\_up.detach();

}

if (maze\_road[x + 1][y] == 0)

{

thread find\_road\_down(findRoad, x + 1, y, 1);

find\_road\_down.detach();

}

if (maze\_road[x][y - 1] == 0)

{

thread find\_road\_left(findRoad, x, y - 1, 4);

find\_road\_left.detach();

}

if (maze\_road[x][y + 1] == 0)

{

thread find\_road\_right(findRoad, x, y + 1, 3);

find\_road\_right.detach();

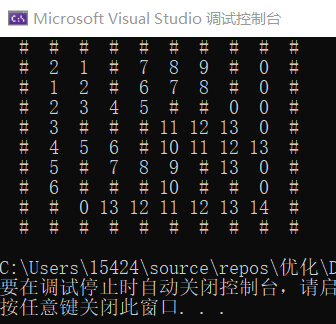
}

/\*分裂成新的机器人后，本机器人销毁\*/

Sleep(WAIT\_TIME \* 2);

return;

}

4.3.4运行结果

五、成员分工

六、收获体会

刘：在学习数据结构的过程中，我认为最重要的应该弄清楚不同数据类型的存储结构，其次是实现不同功能的算法原理，再次是实际编程设计。数据结构是一门原理与实践并重的学课，在清楚数据的存储结构和方式以后，应多加以练习和应用，尤其是在自主解决实际问题的能力上一定要加强。我们可以明显发现，在参加各类建模比赛中，面对一个相对陌生的问题，即使是使用自己最熟虚的语言，在操作起来还是很生疏，所以在今后的学习中一定要注意实践应用。

李：在解决实际问题的过程中，我认为最大的难点是如何将解决方案的具体步骤转化为程序语言实现，最重要的是熟练运用已学的数据结构对问题求解中的数据进行存储使用。数据结构比较抽象，只有联系到具体问题上才方便理解和记忆。

张：在解决迷宫求解的问题过程中，我回顾到了C++的知识，同时对广度遍历有了更深层次的认识，同时认识到将理论付诸于实践当中也有一定的困难。而且，遇到问题要善于结合他人的帮助，比如在迷宫求解问题中，无法将“迷宫”扩大到5\*5以上。论文的撰写过程加深了我对广度遍历的整体掌握程度，对其他方法也有了更深层次的认知。

张：在数据结构学习中，弄清楚各种不同数据类型的特点和原理十分重要，只有这样才能根据实际的需求来选取合适的数据类型，即便问题十分抽象。其次是应当注重实践，因为在具体编程时会遇到各种各样的问题，有时还可以发现自己理解不到位的知识点。

七、代码附录

7.1回溯法—顺序栈

#include<stdlib.h>

#include<stdio.h>

#include<iostream>

using namespace std;

#define RANGE 4

#define row 4

#define col 4

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVER\_FLOW -2

typedef int Status;

typedef int DirectiveType;

//位置坐标

typedef struct {

int x, y;//表示迷宫中的位置信息x行y列

}PostType;

//迷宫类型

typedef struct {

int map[row + 2][col + 2];//用户输入矩阵（0，1）表示迷宫的初始生成

char arr[RANGE + 2][RANGE + 2];//程序的输入矩阵，以字符“@# ”表示探索状态

}MazeType;

//栈类型

typedef struct {

int step;//当前位置在路径上的“序号”

PostType seat;//当前位置坐标

DirectiveType di;//往下一坐标位置的方向

}ElemType;//栈元素类型

typedef struct {

ElemType\* base;

ElemType\* top;

int stacksize;

}Stack;//利用顺序栈实现

Status InitStack(Stack& S) {

S.base = new ElemType[100];

if (!S.base) exit(OVER\_FLOW);

S.top = S.base;

S.stacksize = 100;

return OK;

}

Status StackEmpty(Stack S) {

if (S.top == S.base) return OK;

else return ERROR;

}

Status Push(Stack& S, ElemType e) {

if (S.top - S.base >= S.stacksize) {

ElemType\* newbase = (ElemType\*)realloc(S.base, (S.stacksize + 10) \* sizeof(ElemType));

if (!newbase) exit(OVER\_FLOW);

S.base = newbase;

S.top = S.base + S.stacksize;

S.stacksize += 10;

}

\*S.top++ = e;

return OK;

}

Status Pop(Stack& S, ElemType& e) {

if (S.top == S.base) return ERROR;

e = \*--S.top;

return OK;

}

void InitMaze(MazeType& maze) {

for (int i = 0; i < row + 2; i++) {

for (int j = 0; j < col + 2; j++) {

if (i == 0 || j == 0 || i == row + 1 || j == col + 1) {

maze.map[i][j] = 0;

maze.arr[i][j] = '#';

}

else {

cout << "第" << i << "行" << "第" << j << "列：";

cin >> maze.map[i][j];

if (maze.map[i][j] == 1) maze.arr[i][j] = '#';

else maze.arr[i][j] = ' ';

}

}

}

}

Status Pass(MazeType maze, PostType curpos) {//判断格子是否走过且能走

if (maze.arr[curpos.x][curpos.y] == ' ') return OK;

return ERROR;

}

Status FootPrint(MazeType& maze, PostType curpos) {//记录走过的格子

if (maze.arr[curpos.x][curpos.y] == ' ') {

maze.arr[curpos.x][curpos.y] = '\*';

return OK;

}

return ERROR;

}

Status Same(PostType curpos, PostType end) {//判断两个格子位置是否一样

if (curpos.x == end.x && curpos.y == end.y)

return OK;

return ERROR;

}

PostType NextPos(PostType curpos, int di) {

switch (di)

{

case 1: {curpos.y = curpos.y + 1; return curpos; break; }//向右移动一格

case 2: {curpos.x = curpos.x + 1; return curpos; break; }//向下移动一格

case 3: {curpos.y = curpos.y - 1; return curpos; break; }//向左移动一格

case 4: {curpos.x = curpos.x - 1; return curpos; break; }//向上移动一格

}

}

Status MarkPrint(MazeType& maze, PostType pos) {

maze.arr[pos.x][pos.y] = '@';

return OK;

}

Status MazePath(MazeType& maze, PostType start, PostType end) {

Stack S; InitStack(S);

PostType curpos = start;//设定“当前位置”为“入口位置”

int curstep = 1;//探索第一步,记录探索步数

bool found = ERROR;//判断是否达到终点

ElemType e;//Stack元素

do {

if (Pass(maze, curpos)) {

//当前位置可以通过，即是未曾走到过的通道留下足迹

FootPrint(maze, curpos);

e.di = 1; e.seat = curpos; e.step = curstep;

Push(S, e);//加入路径

if (Same(curpos, end)) found = OK; //到达终点（出口）

else {

curpos = NextPos(curpos, 1);//下一位置是当前位置的东邻（向右移动一格）

//NextPos(curpos, 1);

curstep++;//探索下一步

}

}

else {//当前位置不能通过

if (!StackEmpty(S)) {

Pop(S, e);//将刚才不能前进的Stack元素出栈，相当于退回一步

while (e.di == 4 && !StackEmpty(S)) {

MarkPrint(maze, e.seat);//对刚才出栈的元素的位置标记

Pop(S, e);//下一个元素出栈，与上一步的e不一样

}

if (e.di < 4) {

e.di++; Push(S, e);//换下一个方向探索

curpos = NextPos(e.seat, e.di);//设定当前位置是该新方向向上

}

}

}

} while (!StackEmpty(S) && !found);

return found;

}

Status PrintMaze(MazeType maze) {

for (int i = 0; i < RANGE + 2; i++) {

for (int j = 0; j < RANGE + 2; j++) {

cout << maze.arr[i][j] << " ";

}

cout << "\n";

}

return OK;

}

int main() {

MazeType maze;

InitMaze(maze);

cout << "生成的迷宫为：" << endl;

PrintMaze(maze);

PostType start, end;

cout << "请输入起点："; cin >> start.x >> start.y;

cout << "请输入终点："; cin >> end.x >> end.y;

if (MazePath(maze, start, end)) {

cout << "路径存在，迷宫求解路径为：" << endl;

PrintMaze(maze);

}

else {

cout << "路径不存在，迷宫求解路径为：" << endl;

PrintMaze(maze);

}

return 0;

}

7.2回溯法—链栈

只有结构体定义与顺序栈不同，其他程序均相同

typedef int Status;

typedef int DirectiveType;

//位置坐标

typedef struct {

int x, y;//表示迷宫中的位置信息x行y列

}PostType;

//迷宫类型

typedef struct {

int map[row+2][col+2];//用户输入矩阵（0，1）表示迷宫的初始生成

char arr[RANGE+2][RANGE+2];//程序的输入矩阵，以字符“@# ”表示探索状态

}MazeType;

//栈类型

typedef struct {

int step;//当前位置在路径上的“序号”

PostType seat;//当前位置坐标

DirectiveType di;//往下一坐标位置的方向

}ElemType;//栈元素类型

typedef struct StackNode{

ElemType data;

struct StackNode \* next;

}StackNode, \* LinkStack;//节点类型，指针类型

7.3深度优先搜索遍历法

//显示中文

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <windows.h>//用于函数SetConsoleOutputCP(65001);更改cmd编码为utf8

#include<stdlib.h>

#include<stdio.h>

#include<iostream>

#define M 4

#define N 4

#define MaxSize M \* N

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVER\_FLOW -2

using namespace std;

typedef int Status;

typedef struct {

int map[M + 2][N + 2];//用户输入矩阵（0，1）表示迷宫的初始生成

}MazeType;

//初始化迷宫

void InitMaze(MazeType&maze) {

for (int i = 0; i < M + 2; i++) {

for (int j = 0; j < N + 2; j++) {

if (i == 0 || j == 0 || i == M + 1 || j == N + 1) {

maze.map[i][j] = 1;

}

else {

cout << "第" << i << "行" << "第" << j << "列：";

cin >> maze.map[i][j];

}

}

}

}

//x与y的增量

typedef struct {

int incX,incY;

}Direction;

//位置

typedef struct{

int x,y; //当前坐标

int di; //当前方向

}Box;

//栈

typedef struct

{

Box data[MaxSize];

int top;

}SqStack;

//初始化栈

void InitStack(SqStack &S)

{

S=\*(SqStack \*)malloc(sizeof(SqStack));

S.top=0;

}

//四个方向增量初始化

void Init(Direction direct[])

{

//右

direct[0].incX = 0;

direct[0].incY = 1;

//下

direct[1].incX = 1;

direct[1].incY = 0;

//左

direct[2].incX = 0;

direct[2].incY = -1;

//上

direct[3].incX = -1;

direct[3].incY = 0;

}

//判栈空

Status StackEmpty(SqStack &S)

{

if(S.top==0)

{

return OK;//栈空返回真

}

else

{

return ERROR;

}

}

//入栈

void Push(SqStack &S,Box e)

{

if(S.top==MaxSize)

{

printf("入栈失败！\n");

}

S.data[S.top]=e;

S.top++;

}

//出栈

Box Pop(SqStack &S)

{

Box e;

S.top--;

e = S.data[S.top];

return e;

}

//销毁

void Destroy(SqStack &S)

{

free(&S);

cout<<"栈已销毁！"<<endl;

}

//寻找路径

Status findPath(int maze[][N + 2],Direction direct[], SqStack &S)

{

Box temp;

int x, y, di;

int L, C;

maze[1][1] = -1;

temp.x = 1;

temp.y = 1;

temp.di = -1;

Push(S, temp);//将temp压入到堆栈

while (!StackEmpty(S))//当栈不空的时候

{

temp = Pop(S);

x = temp.x; y = temp.y; di = temp.di + 1;

while (di < 4)

{

L = x + direct[di].incX;

C = y + direct[di].incY;

if (maze[L][C] == 0)

{

temp.x = x;

temp.y = y;

temp.di = di;

Push(S, temp);

x = L;

y = C;

maze[L][C] = -1;

if (x == M && y == N)

{

return OK;

}

else

{

di = 0;

}

}

else

{

di++;

}

}

}

return ERROR;

}

//打印创建的迷宫

Status PrintMaze(MazeType maze) {

for (int i = 0; i < M + 2; i++) {

for (int j = 0; j < N + 2; j++) {

cout << maze.map[i][j] << " ";

}

cout << "\n";

}

return OK;

}

//画路径

Status printPath(MazeType maze,Box path[], int count)

{

//描绘路径

for(int i = count-1; i >= 0; i--)

{

int x = path[i].x;

int y = path[i].y;

maze.map[x][y] = 9;

}

maze.map[M][N] = 9;

PrintMaze(maze);

return OK;

}

int main()

{

//显示中文

SetConsoleOutputCP(65001);

cout << "迷宫只能从左上角开始走到右下角" << endl;

MazeType MAZE;

InitMaze(MAZE);

MazeType\* result = new MazeType(MAZE);

cout << "生成的迷宫为：" << endl;

PrintMaze(MAZE);

cout<<endl;

cout << "找到通路的迷宫为(以9表示找到的路径)：" << endl;

Direction direct[4];//初始化方向结构体

Init(direct);//初始化方向结构体

SqStack S;

InitStack(S);

StackEmpty(S);

Box path[MaxSize];

if(findPath(MAZE.map, direct, S))

{

int count = 0;

for(int i = 0; !StackEmpty(S); i++)

{

path[i] = Pop(S);

count++;

}

printPath(\*result,path,count);

Destroy(S);

}

else

{

cout<<"迷宫没有通路"<<endl;

}

system("pause");

return 0;

}

7.4广度优先搜索遍历法

7.4.1解法一：

#include <iostream>

using namespace std;

#define nRow 100

#define nCol 100

struct note // 队列中的元素类型

{

int x;

int y;

};

// 地图

int ROW\_NUM, COL\_NUM;

int nMatrix[nRow][nCol] = { 0 };

int flag[100][100] = { 0 }; // 标记是否走过的路

int head = 0; // 队列的头部索引

int tail = 0; // 队列的尾部索引

// 方向

int direction[4][2] = {

0, 1,

1, 0,

0, -1,

-1, 0

};

struct note que[10000]; // 队列

int main()

{

for (int i = 0; i < 100; i++)

for (int j = 0; j < 100; j++)

flag[i][j] = 1;

cout << "你想要构造迷宫的长度为:" << endl;

cin >> ROW\_NUM;

cout << "你想要构造迷宫的宽度为：" << endl;

cin >> COL\_NUM;

cout << "以0为通路，以非零实数，如1为墙壁，构造你想要的迷宫：" << endl;

int m = 1;

for (int i = 0; i < ROW\_NUM; i++)

{

cout << "请输入第" << m << "行的数字" << endl;

for (int j = 0; j < COL\_NUM; j++)

{

cin >> nMatrix[i][j];

flag[i][j] = 0;

}

if (m < ROW\_NUM)

{

m++;

}

}

cout << "你构造的迷宫为：" << endl;

for (int i = 0; i < ROW\_NUM; i++)

{

for (int j = 0; j < COL\_NUM; j++)

{

if (nMatrix[i][j] == 0)

cout << " ";

else cout << "[ ]";

}

cout << endl;

}// 输出二维迷宫图

// 初始化每一个队列元素的值（开始寻找的地方）

que[head].x = que[head].y = 0;

tail++; // 尾部索引 + 1

flag[0][0] = 1; // 标记初始位置已经查找过了

cout << "它走过的路径为：" << endl;

cout << "(0,0)" << endl;

while (head < tail)

{

int i = 0;

for (; i < ROW\_NUM; i++)

{

// 每次都是以队头为单位向四个方向开始

int nx = que[head].x + direction[i][0];

int ny = que[head].y + direction[i][1];

if (nx < 0 || nx > ROW\_NUM || ny < 0 || ny > COL\_NUM)

continue;

if (nMatrix[nx][ny] == 0 && flag[nx][ny] == 0)

{

flag[nx][ny] = 1;

que[tail].x = nx; // 加入队列

que[tail].y = ny;

cout << "("<<que[tail].x <<","<< que[tail].y<<")" << endl;

tail++;

}

}

head++;

}

return 0;

}

7.4.2解法二：

#include <thread>//多线程头文件  
#include<Windows.h>//用于窗口等待Sleep  
#include <iostream>  
using namespace std;  
#define WAIT\_TIME 1000//加载时间，越小越快  
string\* maze = NULL;//输入的迷宫  
int maze\_height = 0;//迷宫高度  
int flag = 0;//结束标志  
int aim\_x = 0, aim\_y = 0;//终点坐标  
int\*\* maze\_road;//迷宫数组  
int road\_num = 0, road\_num\_flag = -1;//创建寻路线程次数，寻路线程次数标记  
void printMap();  
void findRoad(int x, int y, int direction);  
void continuePrintMap();  
int main()  
{  
/\*输入数据\*/  
cout << "迷宫高为：";  
cin >> maze\_height;  
cout << "请输入迷宫（墙壁为#）：" << endl;  
maze = new string[maze\_height];  
for (int i = 0; i < maze\_height; i++)  
{  
cin >> maze[i];  
}  
cout << "请输入迷宫终点(x,y)：" << endl;  
cin >> aim\_x >> aim\_y;  
/\*构造迷宫数组\*/  
maze\_road = new int\* [maze\_height];  
if (maze\_road)  
memset(maze\_road, 0, sizeof(int\*) \* maze\_height);  
for (int i = 0; i < maze\_height; i++)  
{  
maze\_road[i] = new int[maze[i].size() + 1];  
for (unsigned int j = 0; j < maze[i].size() + 1; j++)  
{  
if (maze[i][j] != &apos;#&apos;)  
maze\_road[i][j] = 0;  
else  
maze\_road[i][j] = -1;  
}  
}  
/\*打开多线程\*/  
system("cls");//清屏  
thread print\_map(continuePrintMap);  
print\_map.detach();  
thread find\_road(findRoad, 1, 1, 0);  
find\_road.detach();  
/\*后续\*/  
while (1)  
{  
/\*没找到路径不继续\*/  
if (flag == 1)  
break;  
}  
system("cls");  
printMap();//最终迷宫图  
/\*收尾删除\*/  
for (int i = 0; i < maze\_height; i++)  
{  
delete[] maze\_road[i];  
}  
if (maze\_road)  
delete[] maze\_road;  
maze\_road = NULL;  
delete[] maze;  
maze = NULL;  
return 0;  
}  
/\*多线程持续打印迷宫\*/  
void continuePrintMap()  
{  
while (1)  
{  
/\*不再找了也停止（机器人全灭）\*/  
if (road\_num\_flag == road\_num)  
{  
flag = 1;  
return;  
}  
/\*找到位置就停止\*/  
if (flag == 1)  
return;  
printMap();  
}  
  
}  
/\*打印迷宫\*/  
void printMap()  
{  
/\*光标移动到（0，0），不用cls因为会闪\*/  
road\_num\_flag = road\_num;  
printf\_s("\33[0;0H");  
for (int i = 0; i < maze\_height; i++)  
{  
for (unsigned int j = 0; j < maze[i].size(); j++)  
{  
if (maze\_road[i][j] == -1)  
printf\_s("%3c", maze[i][j]);//打印墙  
else  
printf\_s("%3d", maze\_road[i][j]);//打印路  
//if  
//(maze\_road[i][j] == -2);  
//printf\_s("Y");  
}  
cout << endl;  
}  
Sleep(WAIT\_TIME);  
}  
/\*  
多线程迷宫寻路机器人  
x 当前x坐标  
y 当前y坐标  
direction:  
0 没有前一个位置  
1 前一个位置在↑  
2 前一个位置在↓  
3 前一个位置在←  
4 前一个位置在→  
\*/  
void findRoad(int x, int y, int direction)  
{  
Sleep(WAIT\_TIME);  
road\_num++;  
/\*\*/  
if (flag == 1)//寻路完成提前退出  
return;  
/\*记录此地距离原点距离\*/  
switch (direction)  
{  
case 1: {  
maze\_road[x][y] = maze\_road[x - 1][y] + 1;  
break;  
}  
case 2: {  
maze\_road[x][y] = maze\_road[x + 1][y] + 1;  
break;  
}  
case 3: {  
maze\_road[x][y] = maze\_road[x][y - 1] + 1;  
break;  
}  
case 4: {  
maze\_road[x][y] = maze\_road[x][y + 1] + 1;  
break;  
}  
default:  
break;  
}  
/\*寻路成功，标记并退出\*/  
if (x == aim\_x && y == aim\_y)  
{  
flag = 1;  
return;  
}  
/\*向四个方向，有路则释放新的机器人（创建新线程）\*/  
if (maze\_road[x - 1][y] == 0)  
{  
thread find\_road\_up(findRoad, x - 1, y, 2);  
find\_road\_up.detach();  
}  
if (maze\_road[x + 1][y] == 0)  
{  
thread find\_road\_down(findRoad, x + 1, y, 1);  
find\_road\_down.detach();  
}  
if (maze\_road[x][y - 1] == 0)  
{  
thread find\_road\_left(findRoad, x, y - 1, 4);  
find\_road\_left.detach();  
}  
if (maze\_road[x][y + 1] == 0)  
{  
thread find\_road\_right(findRoad, x, y + 1, 3);  
find\_road\_right.detach();  
}  
/\*分裂成新的机器人后，本机器人销毁\*/  
Sleep(WAIT\_TIME \* 2);  
return;  
}.