学号：19S103256

姓名：文荟俨

实验二：搜索算法

1. 实验目的

(1)掌握搜索算法的基本设计思想与方法
(2)掌握 A*算法的设计思想与方法
(3)熟练使用高级编程语言实现搜索算法
(4)利用实现测试验证搜索算法的正确性

2. 实验问题

寻路问题，输入一个方格表示的地图，要求用 A*算法找到并输出从起点(再放各种标示字母 S)到终点(在方格中标示字母 T)的代价最小的路径，有如下条件及要求:

- 每一步都落在方格中,而不是横竖线的交叉点
- 灰色格子表示障碍,无法通行
- 在每个格子处,若无障碍,下一步可以达到八个相邻的格子,并且可以到达无障碍的相邻格子。其中，向上、下、左、右四个方向移动的代价为 1，向四个斜角方向移动的代价为$\sqrt{2}$
- 在一些特殊格子上行走要花费额外的地形代价，比如黄色格子代表沙漠，经过它的代价为 4；蓝色格子代表溪流，经过它的代价为 2；白色格子为普通地形，经过它的代价为 0
- 经过一条路径的总代价为移动代价+地形代价。其中移动代价是路径上所做的所有移动的代价的总和；地形代价为路径上除起点外所有格子的地形代价的总和。

3. 实验步骤

(1) 单向 A*算法
算法 **A* single**(D, S, T)
输入 :平面上 *n* 个点的二维数组 D，起始点 S，终点 T
输出 :从 S 到 T 的路径

1: openlist.push(S)
2: While(true)
3: 寻找 openlist 中 F 值最低的格，称为当前格 P
4: closelist.push(P)
5: **If** 它不可通过||已经在 closelist 中 **Then** ;
6: **If** 它不在 openlsit 中
7: **Then** openlist.push(P)，把 P 作为该格的父节点，计算 FGH 值
8: **If** 它已在 openlist 中

9: **Then If** 它的 F 值更低

10: **Then** 把 P 作为该格的父节点，重新计算 FGH 值

11: **If** 目标格已经在 openlist 中

12: **Then break**

13: 从 T 开始，沿着每一格的父节点回溯，直到回到 S，输出路径

为了便于计算和回溯，我们定义了清单 1 所示的结构体。

清单 1 my_point 结构体定义

```
struct my_Point
{
	int x, y;
	float F, G, H; //F=G+H
	my_Point *parent; //parent 坐标
	my_Point(int _x, int _y) :x(_x), y(_y), F(0), G(0), H(0),
parent(NULL)
	{
	}
};
```

算法的主要时间消耗在第 2-10 行，该部分的核心代码如清单 2 所示。

清单 2 单向 A*核心代码

```
openList.push_back(new my_Point(startPoint.x, startPoint.y));
while (!openList.empty())
{
	auto curPoint = getLeastFpoint(openList);
	cout << "(" << curPoint->x << "," << curPoint->y << ") ";
	cout << "F:" << curPoint->F << "," << "G:" << curPoint->G << ",H:"
<< curPoint->H << endl;;
	openList.remove(curPoint);
	closeList.push_back(curPoint);
	auto surroundPoints = getSurroundPoints(curPoint, isIgnoreCorner,
closeList);
	for (auto &target : surroundPoints)
	{

		if (!isInList(openList, target))
		{
			target->parent = curPoint;
			target->G = calcG(curPoint, target, maze);
			target->H = calcH(target, isIgnoreCorner, maze);
			target->F = calcF(target);
			openList.push_back(target);
		}
		else
		{
			float tempF = calcF(curPoint);
			if (tempF < target->F)
			{
				target->parent = curPoint;
```

```
                        target->F = calcF(target);
                    }
                }
                my_Point *resPoint = isInList(openList, &endPoint);
                if (resPoint)
                    return resPoint;
        }
}
return NULL;
```

(2) 双向 A*算法

双向 A*的思路是在单向 A*的基础上，从 S 和 T 分别开始寻找。不同之处在于需要开两个 openlist 和 closelist，然后判定的条件不再是到达终点 T，二是两者的路径相遇时停止。

算法 **A* bidirection**(D, S, T)

输入 :平面上 *n* 个点的二维数组 D，起始点 S，终点 T

输出 :从 S 到 T 的路径

1: openlist1.push(S)
2: openlist2.push(T)
3: While(true)
4: 寻找 openlist1 中 F 值最低的格，称为当前格 P1
5: 寻找 openlist2 中 F 值最低的格，称为当前格 P2
6: **If** P1 不可通过||已经在 closelist1 中 **Then** ;
7: **If** 它已在 openlist1 中
8: **Then** **If** 它的 F 值更低
9: **Then** 把 P1 作为该格的父节点，重新计算 FGH 值
10: **If** P2 不可通过||已经在 closelist2 中 **Then** ;
11: **If** 它已在 openlist2 中
12: **Then** **If** 它的 F 值更低
13: **Then** 把 P2 作为该格的父节点，重新计算 FGH 值
14: If openlist1 和 openlist2 中存在相同点
15: **Then break**
16: 从 S 和 T 分别开始，沿着每一格的父节点回溯，直到回到相遇点，输出路径

该部分的核心代码如清单 3 所示。

清单 3 双向 A*核心代码

```
void Astar::findPath_bidirection(my_Point &startPoint, my_Point
&endPoint, bool isIgnoreCorner, std::vector<std::vector<int>> maze,
std::list<my_Point *> &path1, std::list<my_Point *> &path2)
{
      openList1.push_back(new my_Point(startPoint.x, startPoint.y));
      openList2.push_back(new my_Point(endPoint.x, endPoint.y));
      while ((!openList1.empty())&&(!openList2.empty()))
      {
            auto curPoint1 = getLeastFpoint(openList1);
            auto curPoint2 = getLeastFpoint(openList2);
```

```
            cout << "(" << curPoint1->x << "," << curPoint1->y << ") ";
            cout << "F:" << curPoint1->F << "," << "G:" << curPoint1->G
<< ",H:" << curPoint1->H << endl;
            cout << "2(" << curPoint2->x << "," << curPoint2->y << ") ";
            cout << "F:" << curPoint2->F << "," << "G:" << curPoint2->G
<< ",H:" << curPoint2->H << endl;
            openList1.remove(curPoint1);
            closeList1.push_back(curPoint1);
            openList2.remove(curPoint2);
            closeList2.push_back(curPoint2);

            auto surroundPoints1 = getSurroundPoints(curPoint1,
isIgnoreCorner, closeList1);
            auto surroundPoints2 = getSurroundPoints(curPoint2,
isIgnoreCorner, closeList2);
            for (auto &target : surroundPoints1)        {
                  if (!isInList(openList1, target))
                  {
                        target->parent = curPoint1;
                        target->G = calcG(curPoint1, target, maze);
                       target->H = calcH(target, isIgnoreCorner, maze);
                        target->F = calcF(target);
                        openList1.push_back(target);
                  }
                  else
                  {
                        float tempF = calcF(curPoint1);
                        if (tempF < target->F)
                        {
                              target->parent = curPoint1;
                              target->F = calcF(target);
                        }
                  }

                  for (auto &target2 : surroundPoints2)
                  {
                        if (!isInList(openList2, target2))
                        {
                              target2->parent = curPoint2;
                              target2->G = calcG(curPoint2, target2,
maze);
                              target2->H = calcH(target2,
isIgnoreCorner, maze);
                              target2->F = calcF(target2);
                              openList2.push_back(target2);
                        }
                        else
                        {
                              float tempF = calcF(curPoint2);
                              if (tempF < target2->F)
```

```
                                {
                                        target2->parent = curPoint2;
                                        target2->F = calcF(target2);
                                }
                        }
                        for (auto &temp1 : openList1) {
                                for (auto &temp2 : openList2) {
                                        if (temp1->x == temp2->x&&temp1->y
== temp2->y) {
                                                my_Point *result1 =
isInList_not_const(openList1, temp1);
                                                my_Point *result2 =
isInList_not_const(openList2, temp1);
                                                while (result1)
                                                {

        path1.push_front(result1);
                                                        result1 = result1-
>parent;
                                                }
                                                while (result2)
                                                {

        path2.push_front(result2);
                                                        result2 = result2-
>parent;
                                                }
                                                return;
                                        }
                                }
                        }
                }
            }
        }
        return ;
}
```

4. 实验结果与分析

(1)实验参数配置

本实验将 exe 程序和 config 配置文件抽离开来，能够实现较好的交互，对配置文件的说明如清单 4 所示。我们可以在[0]的功能模块下选择功能，是否采用双向 A*算法。其他具体的参数也可以在功能区下面配置。

清单 4 config.ini

```
[method]
; 0: A star single
method = 0

[0]
row = 20
```

```
col = 40
data_path = L:/why_workspace/cpl/txt/node2.txt
whether_bidirection = 2
```

(2)数据集文件格式

本实验数据集采用 txt 存储，采用了如清单 5 所示的格式，前面 4 个 int 分别存储起点和终点的坐标值，接着 2 个 int 存储地图的行和列的数目，最后是存储地图的具体值，0 表示普通点，1 表示障碍，2 表示溪流，4 表示沙漠。

清单 5 数据集文件格式说明(以指导书中案例 1 的地图为例)

```
8 3
9 14
14 17
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

(3)案例 1

- 单向 A*

实验指导书中案例 1 的初始情况如图 1 所示。

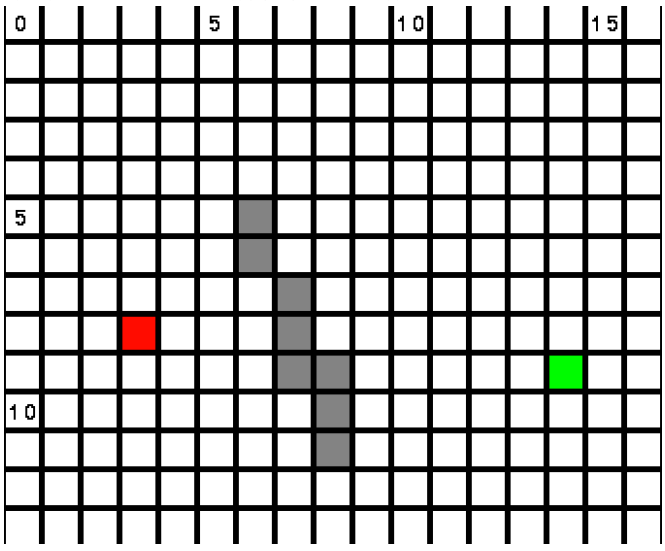

图 1 指导书中案例 1 初始情况(红色为起始点，绿色为终点，灰色为障碍)

经过单向 A*得到的结果如图 2 所示。

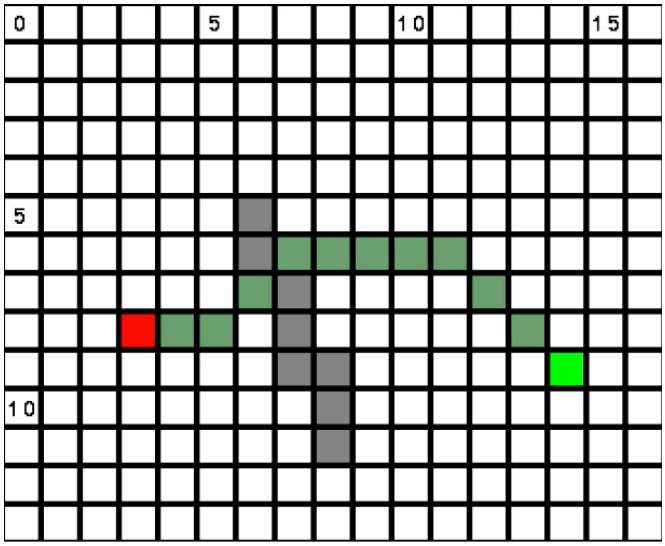

图 2 指导书中案例 1 的单向 A*结果(墨绿色为路径)

为了进一步验证 FGH 值的准确性，将其打印如图 3 所示。经过验证，结果计算正确。

```
(8,3) F:0,G:0,H:0
(7,3) F:2,G:1,H:1
(8,2) F:2,G:1,H:1
(8,4) F:2,G:1,H:1
(9,3) F:2,G:1,H:1
(7,2) F:2.414,G:1.414,H:1
(7,4) F:2.414,G:1.414,H:1
(9,2) F:2.414,G:1.414,H:1
(9,4) F:2.414,G:1.414,H:1
(6,3) F:3,G:2,H:1
(8,1) F:3,G:2,H:1
(8,5) F:3,G:2,H:1
(10,3) F:3,G:2,H:1
(6,2) F:3.414,G:2.414,H:1
(6,4) F:3.414,G:2.414,H:1
(7,1) F:3.414,G:2.414,H:1
(9,1) F:3.414,G:2.414,H:1
(7,5) F:3.414,G:2.414,H:1
(9,5) F:3.414,G:2.414,H:1
(10,2) F:3.414,G:2.414,H:1
(10,4) F:3.414,G:2.414,H:1
(6,1) F:3.828,G:2.828,H:1
(6,5) F:3.828,G:2.828,H:1
(10,1) F:3.828,G:2.828,H:1
```

图 3 指导书中案例 1 的单向 A* FGH 值中间计算过程

- 双向 A*

  经过双向 A*得到的结果如图 4 所示。

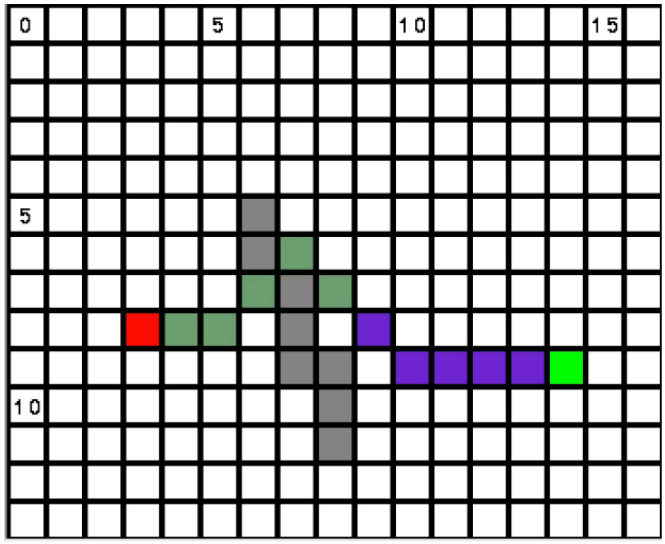

图 4 指导书中案例 1 的双向 A*结果(墨绿色为起始点出发路径，紫色为终点出发路径)

(4) 案例 2

实验指导书中案例 2 的初始情况如图 5 所示。

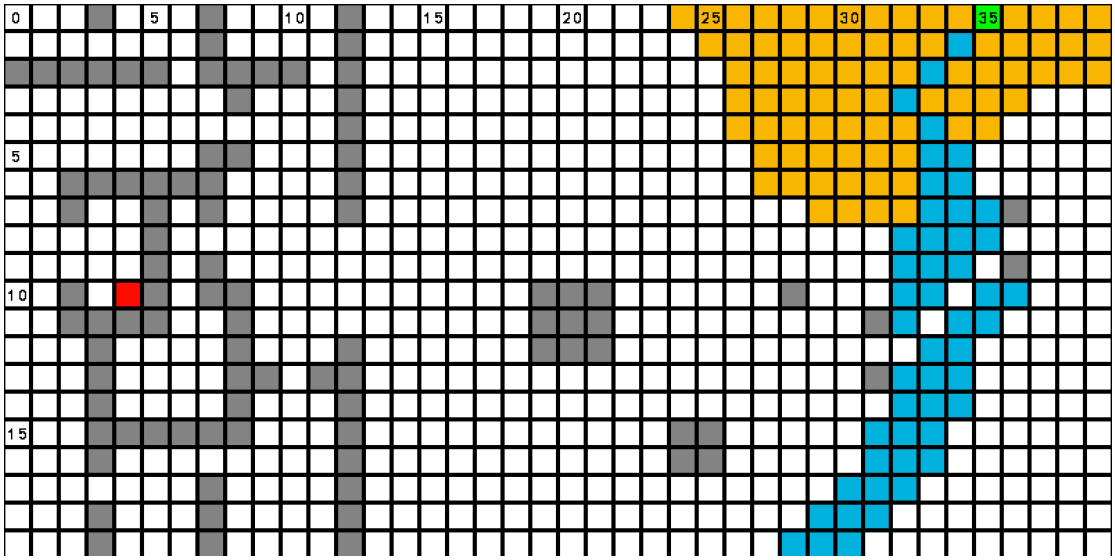

图 5 指导书中案例 2 的初始情况(红色为起始点，绿色为终点，灰色为障碍，蓝色为溪流，橙色为沙漠)

经过单向 A*得到的结果如图 6 所示。

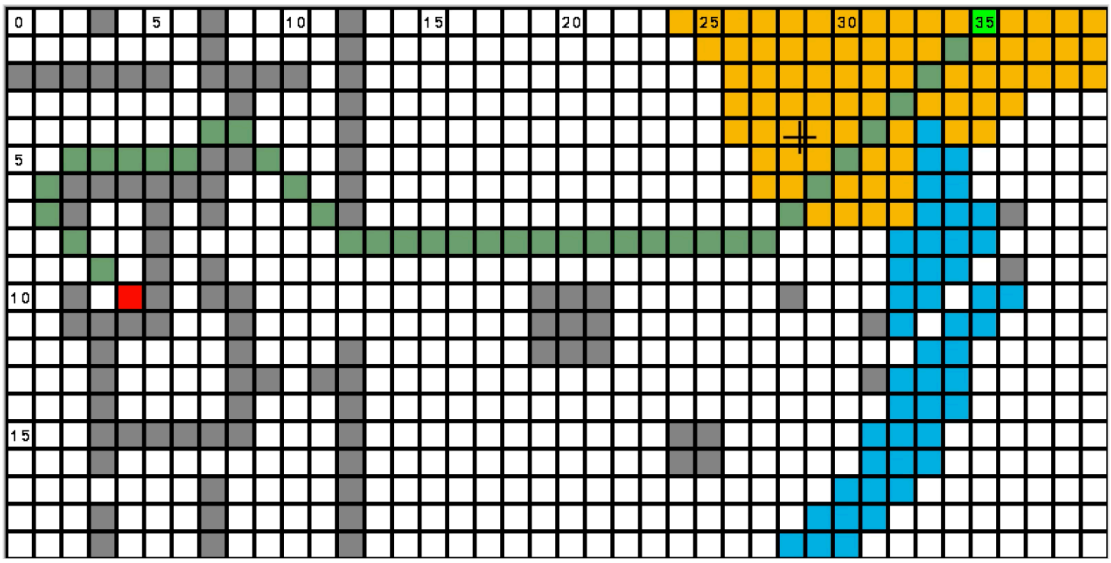

图 6 指导书中案例 2 的单向 A*结果(墨绿色为路径)

经过双向 A*得到的结果如图 7 所示。

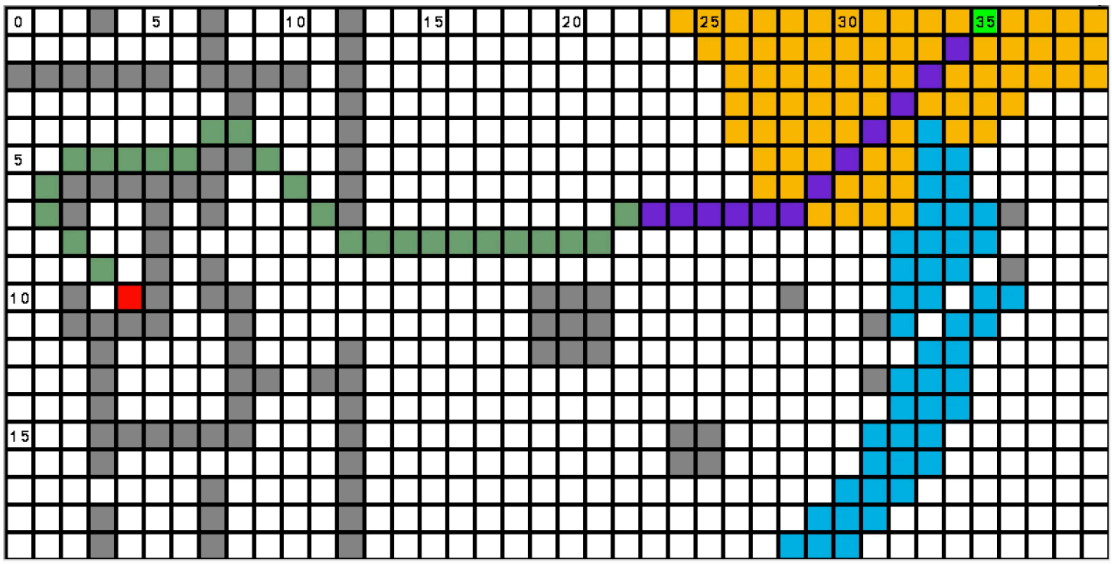

图 7 指导书中案例 2 的双向 A*结果(墨绿色为起始点出发路径，紫色为终点出发路径)

(5)其他案例

为了验证算法准确性，我们更改了算法的起点和终点。结果分别如图 8 和图 9 所示。

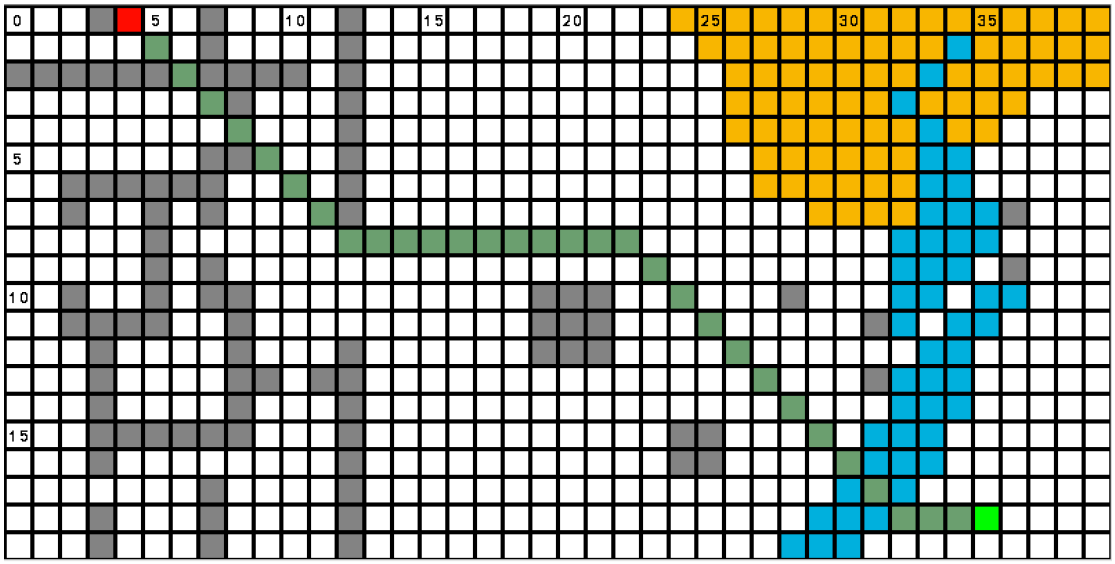

图 8 指导书中案例 2 的搜索结果(单向 A*)

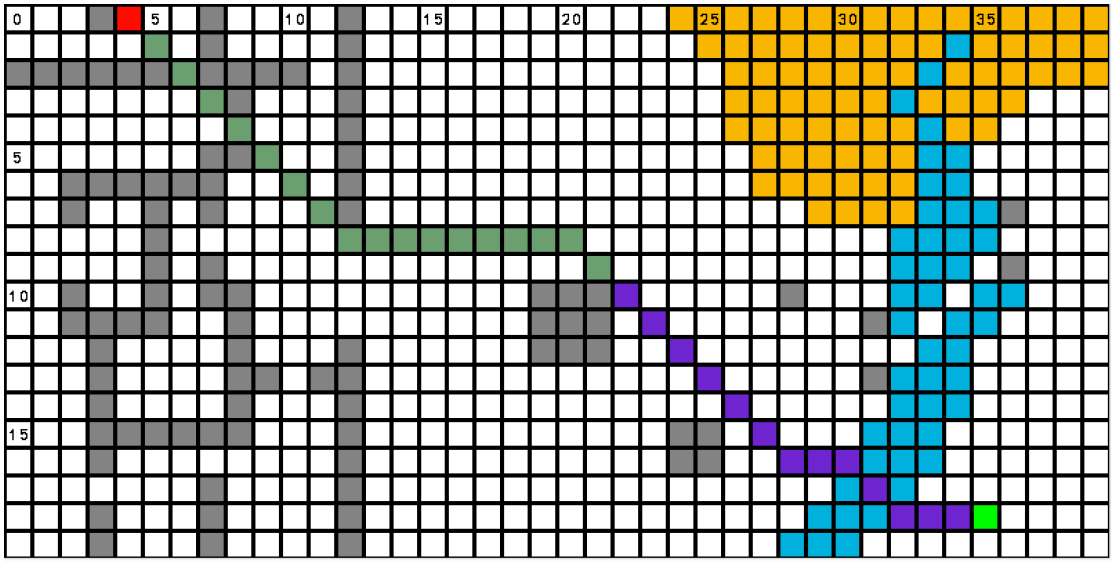

图 9 指导书中案例 2 的搜索结果(双向 A*)

5. 实验心得

这次实验写的比上一次顺利，但仍然有不少坑。比如计算 G 值和 H 值时，很容易把地图的代价重复计算；再就是在得到 openlist 后，返回到调用函数外，需要采用深拷贝，否则当对象调用完成后，原有的 openlist 会被 free 掉，这样就传不出来最后的路径；另外 Opencv 画图也是很坑的一点，比起 python 的一些库，它更加底层，因此画图部分的代码也比较繁杂，写起来不是那么轻松，最后是一个点一条线，挨个画出来的…吐血。

总的来说，实验二收获很多，是第一次从头到尾完整的写这种类似迷宫搜索的算法，最后也有图像可视化，挺有成就感的。