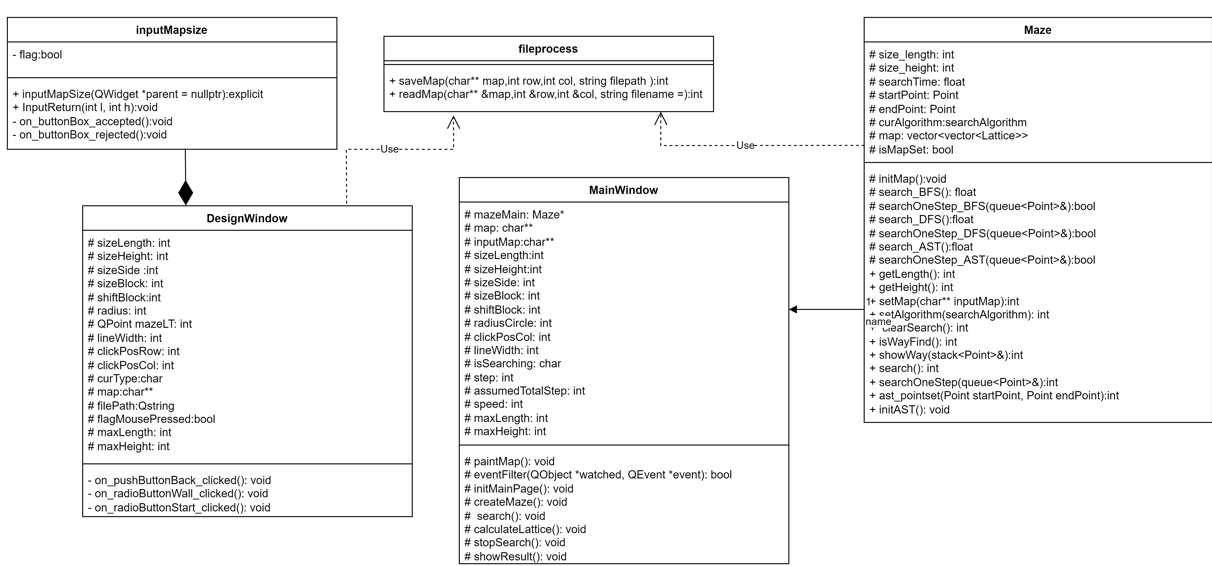
**一、设计与实现**

**1、设计思想**

1. **迷宫类的封装逻辑**
   1. 定义结构体Lattice（格子）作为迷宫的最小数据单元，迷宫的存储结构为Lattice二维数组。Lattice以char type表示当前格子为“通路”、“墙体”等；设置Point coordinate与其二维数组中的索引对应；bool isSearched用于记录当前格子是否被搜索过；struct Lattice\* previous指向当前格子的父格子（父格子的相邻格子为其子格子），方便获得搜索到的路径。
   2. 统一对外接口。所有算法都经过setAlgorithm()和search()函数与外界沟通。
   3. 支持同一张地图多次搜索。设置clearSearch()函数清空和搜索相关的状态而不改变地图本身的结构，以便更换算法多次搜索。
   4. 算法加速。用isSearched作为将某格子压栈/入队的判断依据，墙体isSearched设置为true，免去type判断，type只用于图形化展示。在类内，将地图最外围额外添加一圈围墙，免去边界判断。
   5. 设置两个搜索函数，search()是全速搜索，用于衡量算法用时；searchOneStep(queue<Point>&)是单步搜索，将每一步搜索新搜索的格子坐标返回给调用者，以便动态展示搜索过程。
   6. 针对函数调用的先后关系，类内建立了报错机制。可以有效避免后续修改时可能出现的逻辑性漏洞。
2. **迷宫搜索算法**
   1. 深搜广搜分别用栈和队列实现，二者代码高度复用，区别仅在于是用栈还是队列存储格子。
   2. A\*算法在广搜的基础上，引入以当前点和目的地间距离为基础的评价函数，对队列中的待搜索点进行排序，减少不必要的搜索，达到加速搜索的目的。
3. **界面设计思想**
   1. 动态展示算法搜索过程。
   2. 软件内可自带迷宫编辑器。
   3. 迷宫文件校验。只能导入txt文件，导入的txt会被核查是否为迷宫地图；迷宫设计器和搜索界面均会检查导入的迷宫是否符合要求，例如只能由一个入口/出口，迷宫外墙需要闭合等。
   4. 用户友好的设计理念。例如迷宫设计界面支持按住鼠标滑动放置迷宫围墙，针对不同迷宫大小动态展示的速度会自适应变化。

**2、类结构**



**3、主要数据结构**

1. **迷宫地图**

定义结构体Lattice（格子）作为迷宫的最小数据单元，迷宫的存储结构为Lattice二维数组。

1. **搜索算法**
   1. 深搜：栈（stack）
   2. 广搜：队列（queue）
   3. A\*算法：可排序队列（用vector实现）

**4、算法设计与核心代码**

**1）是否搜索到出口**

地图读入时会记录出口的坐标。int isWayFind()函数会判断出口格子的isSearched状态。如果为true，说明出口已经被搜索到了，即找到了迷宫的解；如果为false，则还未找到解；如果已经空栈/空队列，即搜索完了地图中所有可搜索的点，isSearched仍然为false，则判定迷宫没有解。

int Maze::isWayFind()

{

    if (isMapSet == false) return -1;

    if (map[endPoint.row][endPoint.col].previous == NULL) return false;

    else return true;

}

**2）返回迷宫的解**

当迷宫搜索到解时，int showWay(stack<Point>&)函数会从出口格子开始，用previous指针向前遍历父格子，直到previous == NULL，即遍历到了入口格子（入口的previous指针为NULL）。将这些格子的Point coordinate（坐标）记录下来返回给调用者。

int Maze::showWay(stack<Point>& output)

{

    if (isMapSet == false) return -1;//判断迷宫地图是否已经导入

    Lattice\* curLattice = &map[endPoint.row][endPoint.col];

    if (curLattice->previous == NULL) return false;

    while (curLattice != NULL) {

        Point toOutside;

        toOutside.col = curLattice->coordinate.col - 1;

        toOutside.row = curLattice->coordinate.row - 1;

        output.push(toOutside);

        curLattice = curLattice->previous;

    }

    return true;

}

**3）搜索算法**

1. 统一接口函数

int Maze::search()

{

    if (isMapSet == false) return -1;

    if (curAlgorithm == BFS) {

        return search\_BFS();

    }

    else if (curAlgorithm == DFS) {

        return search\_DFS();

    }

    //add your algorithm

    else if (curAlgorithm == AST) {

        return search\_AST();

    }

    return -1;

}

1. 深搜

由于深搜使用栈结构，遵循后进先出，因此深搜算法会优先搜索当前格子的子格子，将其他同代的格子的搜索后置，从效果上能达到优先沿一个方向深入的效果。深搜不能搜到最优解。

float Maze::search\_DFS()

{

    auto start = steady\_clock::now();

    stackDFS.push(startPoint);

    map[startPoint.row][startPoint.col].isSearched = true;

    while (stackDFS.empty() == false) {

        //赋值，出栈

        Point cur = stackDFS.top();

        stackDFS.pop();

        //四联通，这里是用空间换时间

        Point toSearch;

        toSearch.col = cur.col - 1;

        toSearch.row = cur.row;

        if (map[toSearch.row][toSearch.col].isSearched == false) {

            stackDFS.push(Point(toSearch.row, toSearch.col));

            map[toSearch.row][toSearch.col].isSearched = true;

            map[toSearch.row][toSearch.col].previous = &map[cur.row][cur.col];

        }

        toSearch.col = cur.col + 1;

        if (map[toSearch.row][toSearch.col].isSearched == false) {

            stackDFS.push(Point(toSearch.row, toSearch.col));

            map[toSearch.row][toSearch.col].isSearched = true;

            map[toSearch.row][toSearch.col].previous = &map[cur.row][cur.col];

        }

        toSearch.col = cur.col;

        toSearch.row = cur.row - 1;

        if (map[toSearch.row][toSearch.col].isSearched == false) {

            stackDFS.push(Point(toSearch.row, toSearch.col));

            map[toSearch.row][toSearch.col].isSearched = true;

            map[toSearch.row][toSearch.col].previous = &map[cur.row][cur.col];

        }

        toSearch.row = cur.row + 1;

        if (map[toSearch.row][toSearch.col].isSearched == false) {

            stackDFS.push(Point(toSearch.row, toSearch.col));

            map[toSearch.row][toSearch.col].isSearched = true;

            map[toSearch.row][toSearch.col].previous = &map[cur.row][cur.col];

        }

        //是否搜到了出口

        if (isWayFind() == true) break;

    }

    auto end = steady\_clock::now();

    auto last = duration\_cast<microseconds>(end - start);

    return last.count();

}

1. 广搜

由于广搜使用队列结构，遵循先入先出，可以实现由某一点逐层向外扩张搜索的效果。广搜是外扩式的遍历，而且先被搜索的点到起始点的距离一定小于等于后被搜索的点到起始点的距离，因此广搜能搜到最短路径。

float Maze::search\_BFS()

{

    auto start = steady\_clock::now();

    queueBFS.push(startPoint);

    map[startPoint.row][startPoint.col].isSearched = true;

    while (queueBFS.empty() == false) {

        //赋值，出队

        Point cur = queueBFS.front();

        queueBFS.pop();

        //四联通，这里是用空间换时间

        Point toSearch;

        toSearch.col = cur.col - 1;

        toSearch.row = cur.row;

        if (map[toSearch.row][toSearch.col].isSearched == false) {

            queueBFS.push(Point(toSearch.row, toSearch.col));

            map[toSearch.row][toSearch.col].isSearched = true;

            map[toSearch.row][toSearch.col].previous = &map[cur.row][cur.col];

        }

        toSearch.col = cur.col + 1;

        if (map[toSearch.row][toSearch.col].isSearched == false) {

            queueBFS.push(Point(toSearch.row, toSearch.col));

            map[toSearch.row][toSearch.col].isSearched = true;

            map[toSearch.row][toSearch.col].previous = &map[cur.row][cur.col];

        }

        toSearch.col = cur.col;

        toSearch.row = cur.row - 1;

        if (map[toSearch.row][toSearch.col].isSearched == false) {

            queueBFS.push(Point(toSearch.row, toSearch.col));

            map[toSearch.row][toSearch.col].isSearched = true;

            map[toSearch.row][toSearch.col].previous = &map[cur.row][cur.col];

        }

        toSearch.row = cur.row + 1;

        if (map[toSearch.row][toSearch.col].isSearched == false) {

            queueBFS.push(Point(toSearch.row, toSearch.col));

            map[toSearch.row][toSearch.col].isSearched = true;

            map[toSearch.row][toSearch.col].previous = &map[cur.row][cur.col];

        }

        //是否搜到了出口

        if (isWayFind() == true) break;

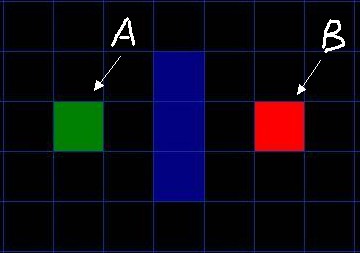
    }

    auto end = steady\_clock::now();

    auto last = duration\_cast<microseconds>(end - start);

    return last.count();

}

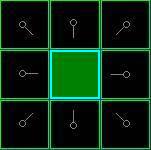
1. A\*算法

如图所示简易地图, 其中绿色方块的是起点 (用 A 表示), 中间蓝色的是障碍物, 红色的方块 (用 B 表示) 是目的地. 为了可以用一个二维数组来表示地图, 我们将地图划分成一个个的小方块。

寻路步骤：

1. 从起点A开始, 把它作为待处理的方格存入一个"开启列表", 开启列表就是一个等待检查方格的列表。

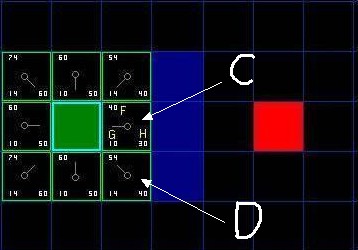
2. 寻找起点A周围可以到达的方格, 将它们放入"开启列表", 并设置它们的"父方格"为A。

3. 从"开启列表"中删除起点 A, 并将起点 A 加入"关闭列表", "关闭列表"中存放的都是不需要再次检查的方格。

从 "开启列表" 中找出相对最靠谱的方块, 什么是最靠谱? 它们通过公式 F=G+H 来计算。

G 表示从起点 A 移动到网格上指定方格的移动耗费 (可沿斜方向移动).

H 表示从指定的方格移动到终点 B 的预计耗费 (H 有很多计算方法, 这里我们设定曼哈顿距离.)

我们假设横向移动一个格子的耗费为10, 为了便于计算, 沿斜方向移动一个格子耗费是14。为了更直观的展示如何运算 FGH, 图中方块的左上角数字表示 F, 左下角表示 G, 右下角表示 H.

4. 从 "开启列表" 中选择 F 值最低的方格 C，把它从 "开启列表" 中删除, 并放到 "关闭列表" 中。

5. 检查它所有相邻并且可以到达 (障碍物和 "关闭列表" 的方格都不考虑) 的方格. 如果这些方格还不在 "开启列表" 里的话, 将它们加入 "开启列表", 计算这些方格的 G, H 和 F 值各是多少, 并设置它们的 "父方格" 为 C。

6. 如果某个相邻方格 D 已经在 "开启列表" 里了, 检查如果用新的路径 (就是经过C 的路径) 到达它的话, G值是否会更低一些, 如果新的G值更低, 那就把它的 "父方格" 改为目前选中的方格 C, 然后重新计算它的 F 值和 G 值 (H 值不需要重新计算, 因为对于每个方块, H 值是不变的). 如果新的 G 值比较高, 就说明经过 C 再到达 D 不是一个明智的选择, 因为它需要更远的路, 这时我们什么也不做。

**简而言之：**A\*算法在广搜的基础上，引入以当前点和目的地间距离为基础的评价函数，对队列中的待搜索点进行排序，减少不必要的搜索，达到加速搜索的目的。

float Maze::search\_AST()

{

    double startTime = clock();

    while (!openlist.empty())

    {

        sort(openlist.begin(), openlist.end(), NodeCompare{});

        Node\* current = \*openlist.begin();

        openlist.erase(openlist.begin());

        closelist.push\_back(current);

        map[current->point.y][current->point.x].isSearched = true;

        //map[current->point.x][current->point.y].type = '.';//此处直接更改了格子的类型

        if (isWayFind() == true)

        {

            openlist.clear();

            closelist.clear();

            break;

        }

        int x = current->point.x;

        int y = current->point.y;

        vector<astPoint> neighbors =

            { // 4个邻近节点的坐标

                { x - 1,y },

                { x,y - 1 }, { x,y + 1 },

                { x + 1,y }

            };

        for (auto n : neighbors)

        {

            if ((n.x >= 1 && n.x <= size\_length) && (n.y >= 1 && n.y <= size\_height) && map[n.y][n.x].type != '#')

            {

                bool incloselist = false;

                for (auto c : closelist)

                {

                    if (c->point.x == n.x && c->point.y == n.y)

                    {

                        incloselist = true;

                        break;

                    }

                }

                if (incloselist)

                {

                    continue;

                }

                bool inopenlist = false;

                for (auto o : openlist)

                {

                    if (o->point.x == n.x && o->point.y == n.y)

                    {

                        inopenlist = true;

                        double g = current->g + n.distance(current->point);

                        double h = n.distance(goal);

                        double f = g + h;

                        if (f < (o->f))

                        {

                            o->f = f;

                            o->parent = current;

                            map[n.y][n.x].previous = &map[current->point.y][current->point.x];

                        }

                        break;

                    }

                }

                if (!inopenlist)

                {

                    double g = current->g + n.distance(current->point);

                    double h = n.distance(goal);

                    double f = g + h;

                    openlist.push\_back(new Node(n, g, h, current));

                    map[n.y][n.x].previous = &map[current->point.y][current->point.x];

                }

            }

        }

    }

    stack<Point>output;

    showWay(output);

    while(!output.empty())

    {

        cout << output.top().col << ',' << output.top().row <<endl;

        output.pop();

    }

    double endTime = clock();

    return (endTime - startTime);

}

bool Maze::searchOneStep\_AST(queue<Point>& newSearchedPoint)

{

    sort(openlist.begin(), openlist.end(), NodeCompare{});

    Node\* current = \*openlist.begin();

    openlist.erase(openlist.begin());

    closelist.push\_back(current);

    map[current->point.y][current->point.x].isSearched = true;

    //map[current->point.x][current->point.y].type='.';//此处直接更改了格子的类型

    Point a(current->point.y-1,current->point.x-1);

    if(start.x!=current->point.x||start.y!=current->point.y)

        newSearchedPoint.push(a);

    if (isWayFind() == true)

    {

        openlist.clear();

        closelist.clear();

        return true;

    }

    int x = current->point.x;

    int y = current->point.y;

    vector<astPoint> neighbors =

        { // 4个邻近节点的坐标

            { x - 1,y },

            { x,y - 1 }, { x,y + 1 },

            { x + 1,y }

        };

    for (auto n : neighbors)

    {

        if ((n.x >= 1 && n.x <= size\_length) && (n.y >= 1 && n.y <= size\_height) && map[n.y][n.x].type != '#')

        {

            bool incloselist = false;

            for (auto c : closelist)

            {

                if (c->point.x == n.x && c->point.y == n.y)

                {

                    incloselist = true;

                    break;

                }

            }

            if (incloselist)

            {

                continue;

            }

            bool inopenlist = false;

            for (auto o : openlist)

            {

                if (o->point.x == n.x && o->point.y == n.y)

                {

                    inopenlist = true;

                    double g = current->g + n.distance(current->point);

                    double h = n.distance(goal);

                    double f = g + h;

                    if (f < (o->f))

                    {

                        o->f = f;

                        o->parent = current;

                        map[n.y][n.x].previous = &map[current->point.y][current->point.x];

                    }

                    break;

                }

            }

            if (!inopenlist)

            {

                double g = current->g + n.distance(current->point);

                double h = n.distance(goal);

                double f = g + h;

                openlist.push\_back(new Node(n, g, h, current));

                map[n.y][n.x].previous = &map[current->point.y][current->point.x];

            }

        }

    }

    return false;

}

**4）单步搜索**

单步搜索函数是将全速搜索的循环移除，循环过程由外部调用者自行编写，达到动态展示搜索过程的需要

**5）文件读写**

实现从计算机任意路径读取并加载迷宫txt文件以及将绘制好的迷宫保存为txt文件的功能

地图存取说明：

1.savemap：接受主程序的二维数组指针，以及行列数，将迷宫信息保存到指定的txt文件

2.readmap：打开指定路径的txt迷宫文件，返回行列数 以及指向迷宫的二维数组指针

要点：

1.在迷宫文件里设置校验码，写入文件时自动写入，读取文件时判断校验码来确定是否打开迷宫文件

2.文件读取中设置多处错误判断，一但读取文件错误就返回读取文件失败，保证程序正常运行

int saveMap(char\*\* map,int row,int col, string filepath)//txt文件格式：第一行存入校验码，第二行行和列，后面存入迷宫

{

    ofstream fout;                                      //将设计的迷宫存入txt

    //string filepath = ".//dataset//" + filepath;

    fout.open(filepath);

    if (!fout.is\_open()) {

        std::cerr << "Unable to open file"+filepath;

        return -1;

    }

    fout << "C++migongxianren" << endl;

    fout << row << " " << col << endl;

    for (int i = 0; i<row; i++)

    {

        for (int j = 0; j<col; j++)

        {

            fout << map[i][j];

        }

        fout << endl;

    }

    fout.close();

    return 1;

}

int readMap(char\*\*& map, int &row,int &col,std::string filepath)//将迷宫取出，存到\*\*map指向的二维数组。

{

    std::ifstream fin;

    //std::string filepath = ".//dataset//" + filepath;

    fin.open(filepath);

    if (!fin.is\_open()) {

        std::cerr << "Unable to open file" << filepath << std::endl;

        return -1;

    }

    string mima;//校验码，看是否打开迷宫文件

    getline(fin, mima);

    if (mima != "C++migongxianren")

    {

        cout << "打开错误文件" << endl;

        return -1;

    }

    fin >> row;

    fin >> col;

    string buffer;

    getline(fin,buffer);

    map = new char\* [row];

    for (int i = 0; i < row; ++i) {

        map[i] = new char[col+1];

    }

    for (int i = 0; i < row; i++)

    {

        if (!std::getline(fin, buffer)) {

            // 处理文件结束或读取错误

            std::cerr << "Failed to read line " << i + 1 << std::endl;

            // 释放已分配的内存并返回错误

            for (int j = 0; j < i; j++) {

                delete[] map[j];

            }

            delete[] map;

            return -1;

        }

        for (int j = 0; j < col; j++)

        {

            if (buffer.size() != col) {

                cerr << "Line " << i + 1 << " does not have the expected length." << endl;

                return -1;

            }

            map[i][j] = buffer[j]; // 直接从buffer中复制字符

        }

        map[i][col] = '\0';// 添加字符串终止符

    }

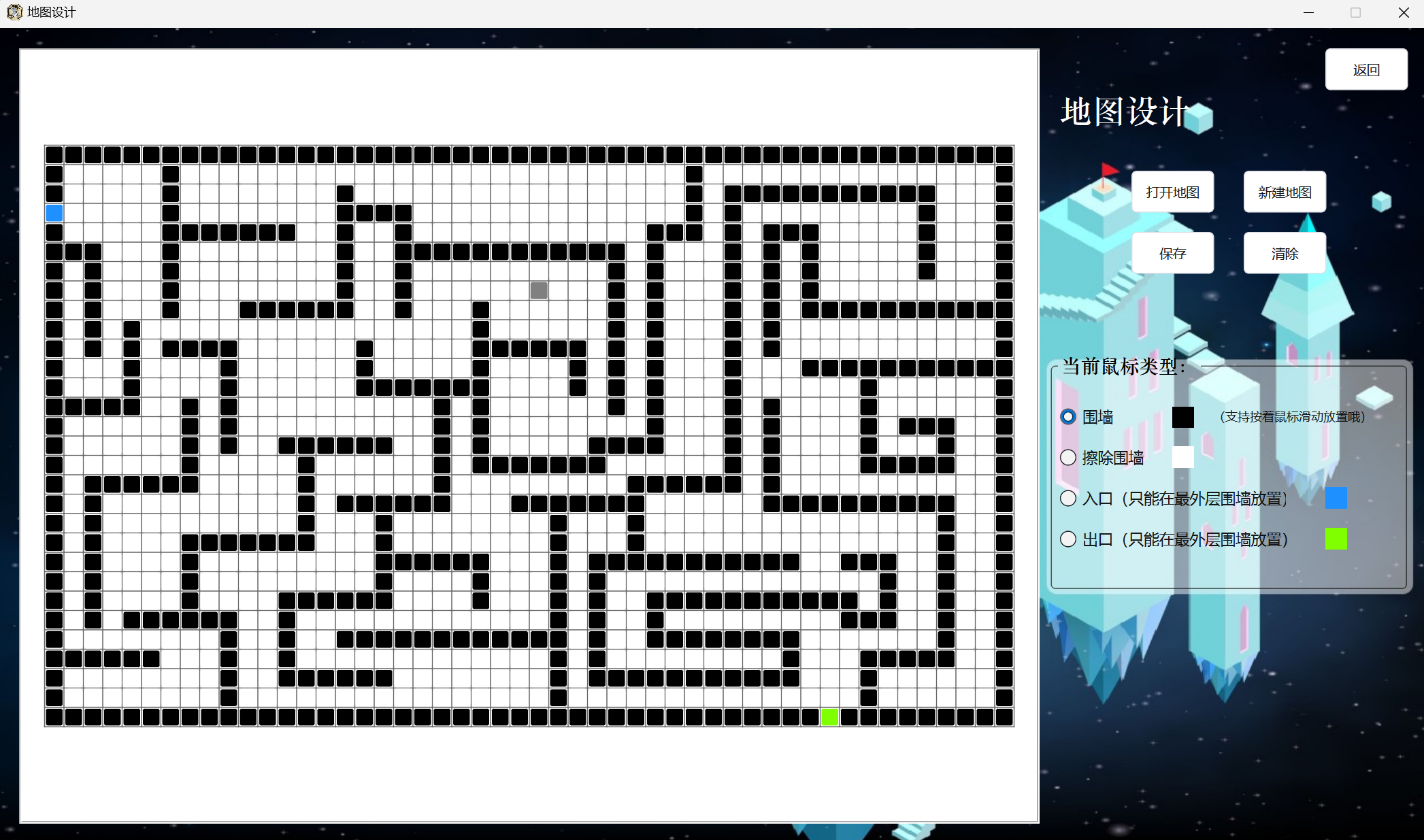
    fin.close();

    return 1;

}

**5、对题目要求的实现情况**

1. 创建迷宫
   1. 可以选择创建的迷宫的方式，包括文件读取和用户输入；
   2. 展示生成的迷宫，入口、出口、通路和障碍清晰可辨。



1. 迷宫求解
   1. 展示探索过程，确定是否有解；



搜索中，还有进度条提示。

* 1. 根示探索结果，对于有解情况给出路径；



广度优先得到的最短路径。

1. 如果有多个路径，能够求解最短路径

2）b）展示的广度优先结果即为最短路径。

1. 尝试提高寻路效率。
   1. 引入了A\*算法加速搜索。具体测试结果见“三、2”。
   2. 对深搜广搜做了优化
      1. 用isSearched作为将某格子压栈/入队的判断依据，墙体isSearched设置为true，免去type判断，type只用于图形化展示；
      2. 在类内，将地图最外围额外添加一圈围墙，免去边界判断；
      3. 将循环中重复计算的变量外置。
2. 图形界面

已经展示。

**二、测试与结论**

**1、测试环境与数据**

本项目使用C++ Qt 6.7.0和Qt Creator 13.0.0 Community编写，用Qt 6.7.0 (MinGW 11.2.0 64-bit)构建。本软件以exe + 动态链接库的形式发行。在win11系统，分辨率大于的不同设备上均正常运行。

**2、测试用例**

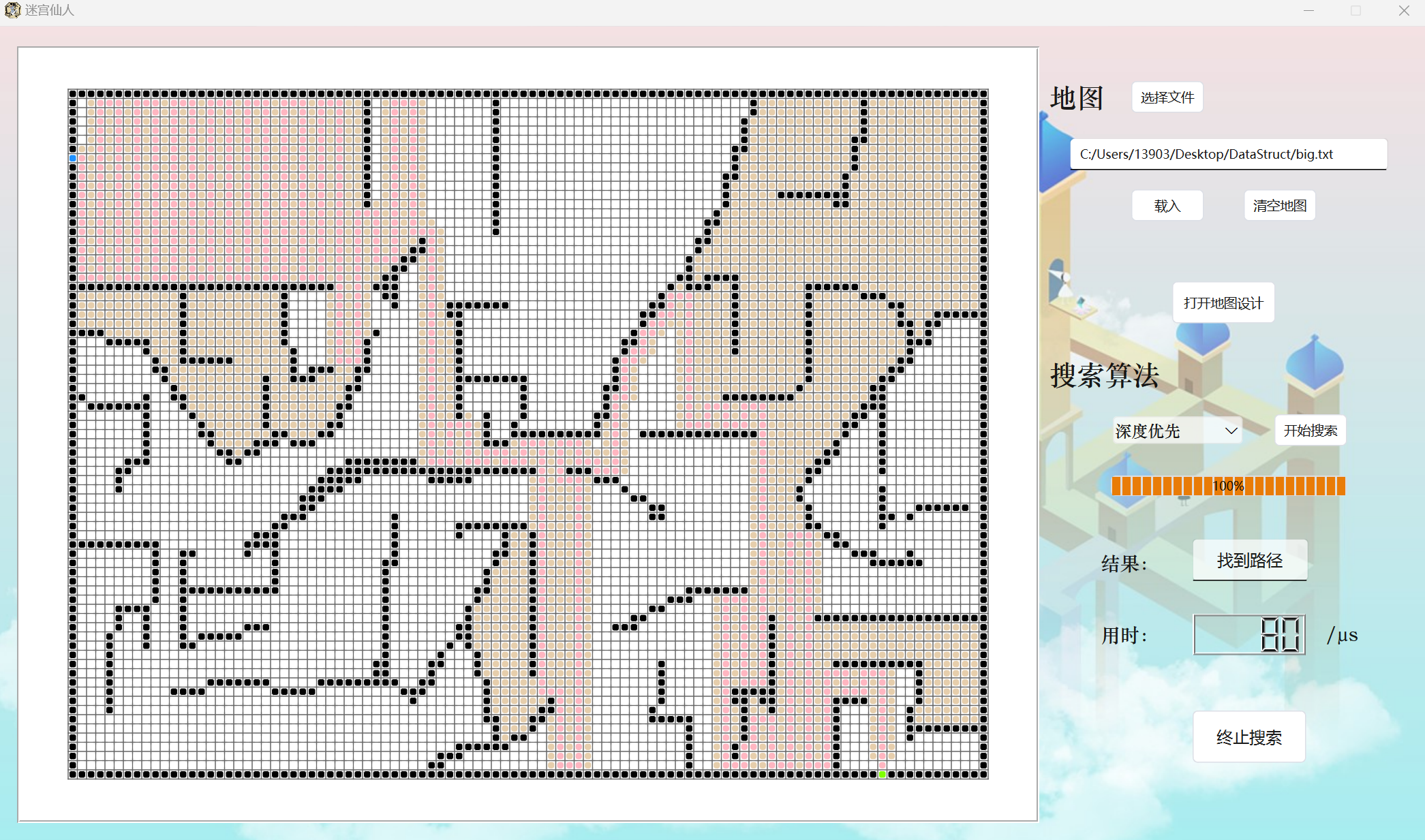
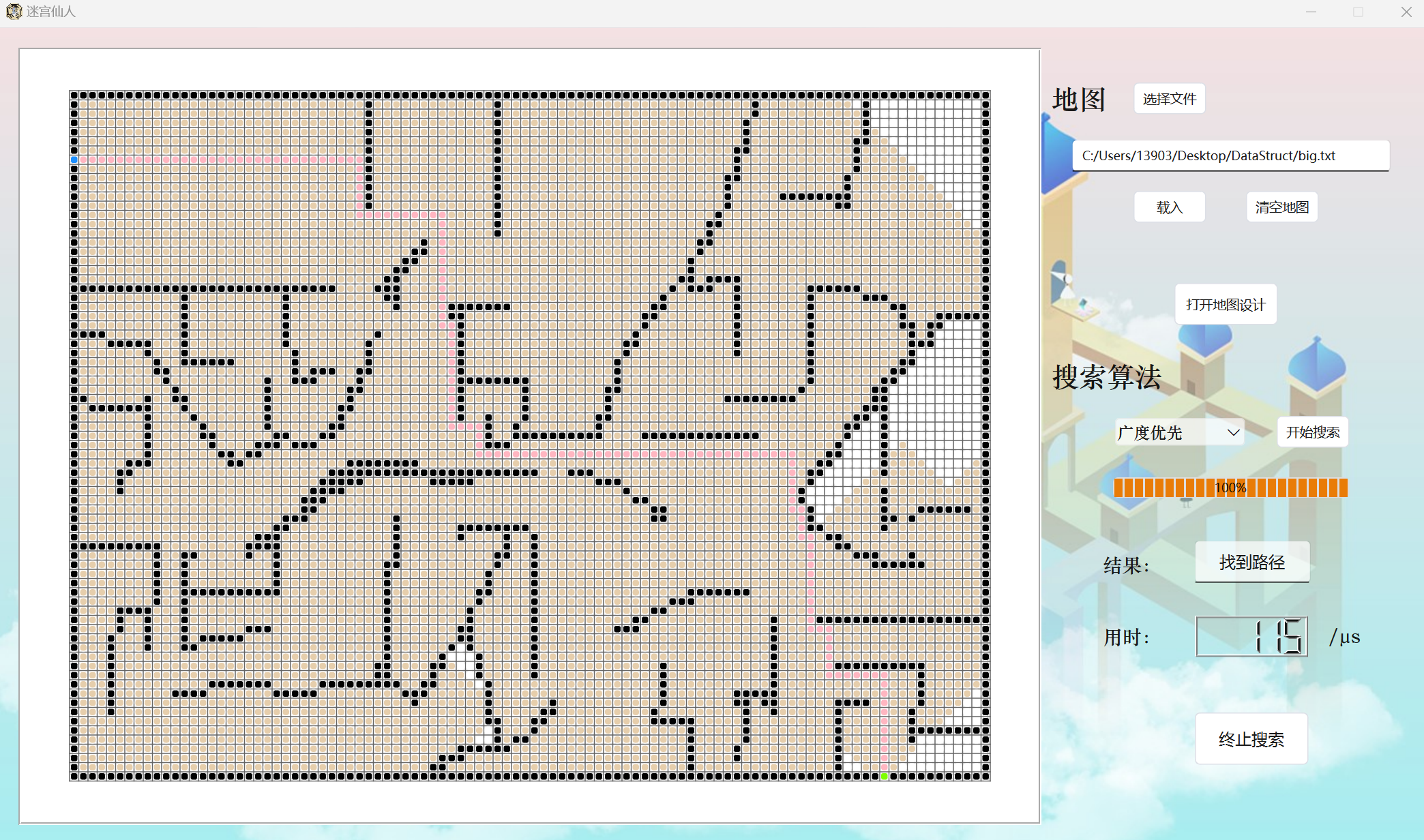
1. 寻找到解

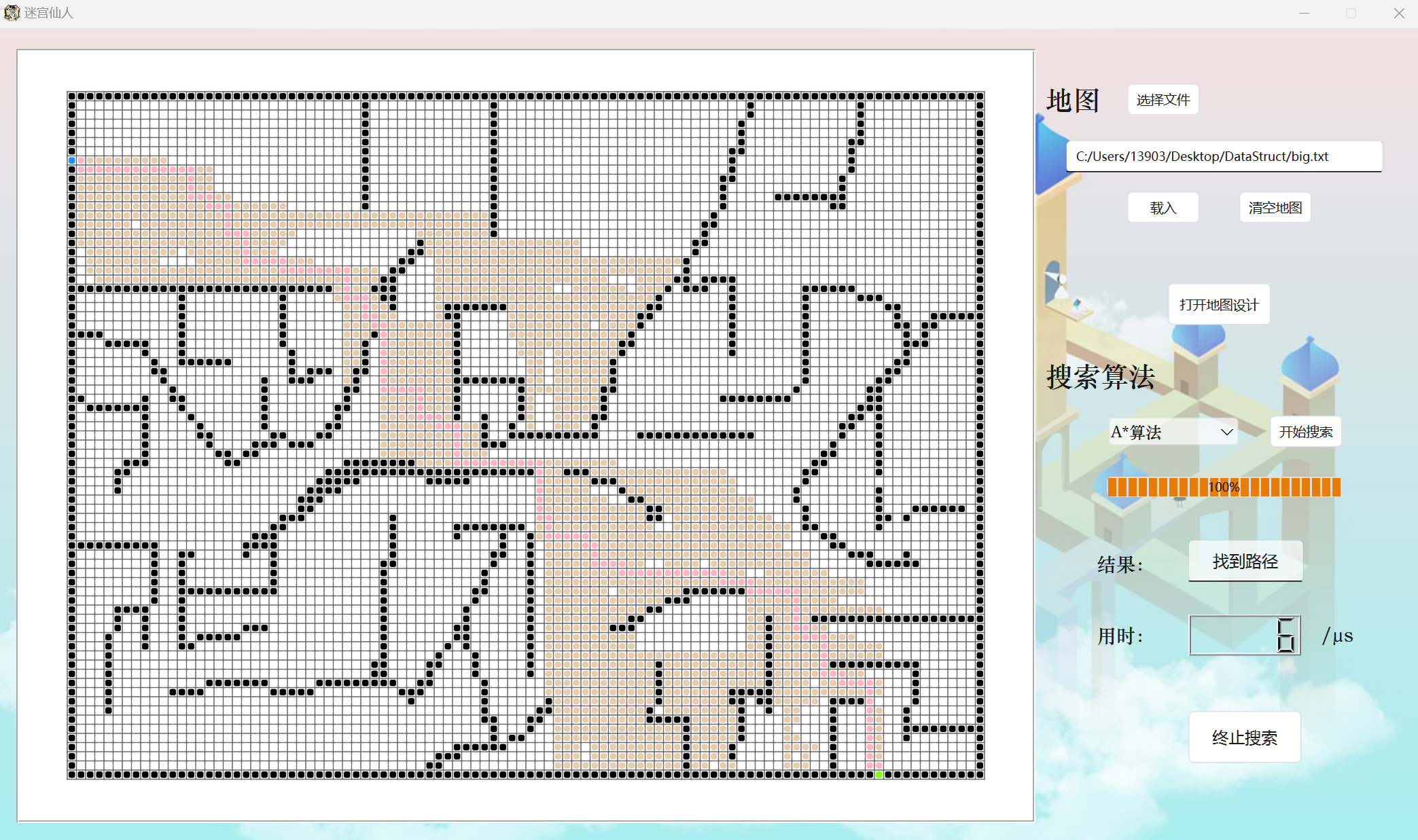


1. 未寻找到解



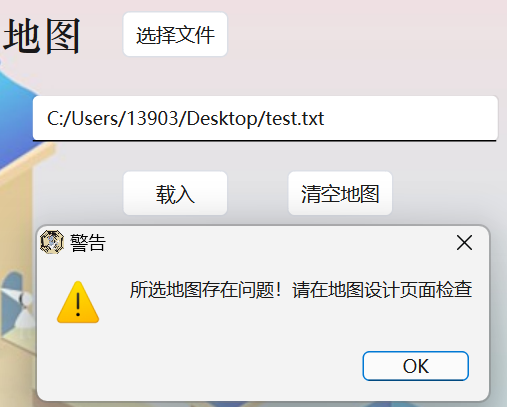
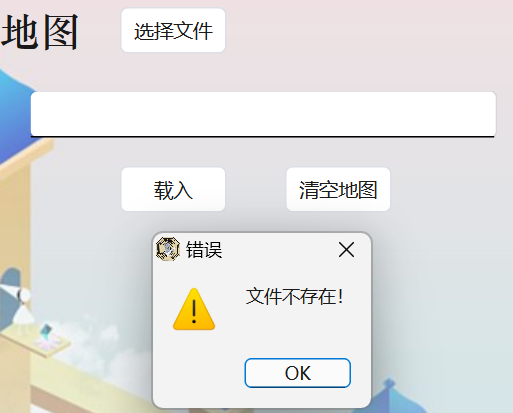
1. 算法速度对比

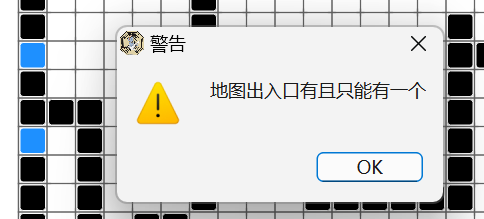
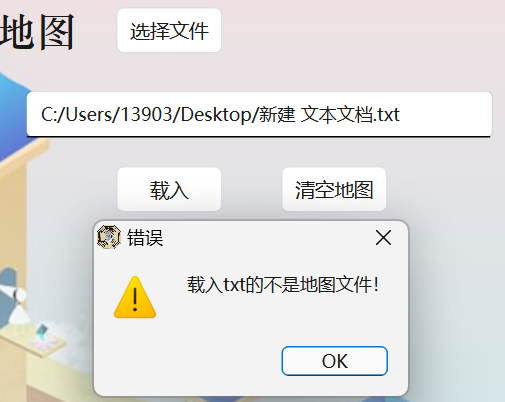




广搜：116μs 深搜：80μs A\*：6μs

1. 各类报错提示





**3、测试结论**

软件各部分运行正常，高质量完成了题目要求。

**三、总结与思考**

**1、题目难点要点**

1. 搜索算法的实现
   1. 需要用到栈和队列
   2. 需要对算法进行优化
   3. 需要引入更高效的搜索算法
2. 搜索过程的动态展示
3. 迷宫地图的创建：除了读取文件，还需要让用户在软件内可以交互式输入

**2、本组工作特点**

独创性：集成的图形化迷宫设计器，优秀的前端设计，多种算法搜索过程动态展示。

拓展性：拥有良好设计的API，开发者均遵循OOP而非面向过程。

题目中最有价值的内容：扩展算法（如A\*）的动手实现，OOP设计理念和项目开发能力。

**3、本组改进方向**

1. 迷宫设计器加入随机迷宫生成。
2. 迷宫设计器提示当前迷宫有解与否。
3. 加入更多高效搜索算法
4. 重构A\*算法的代码，提高对类内已有代码的复用性