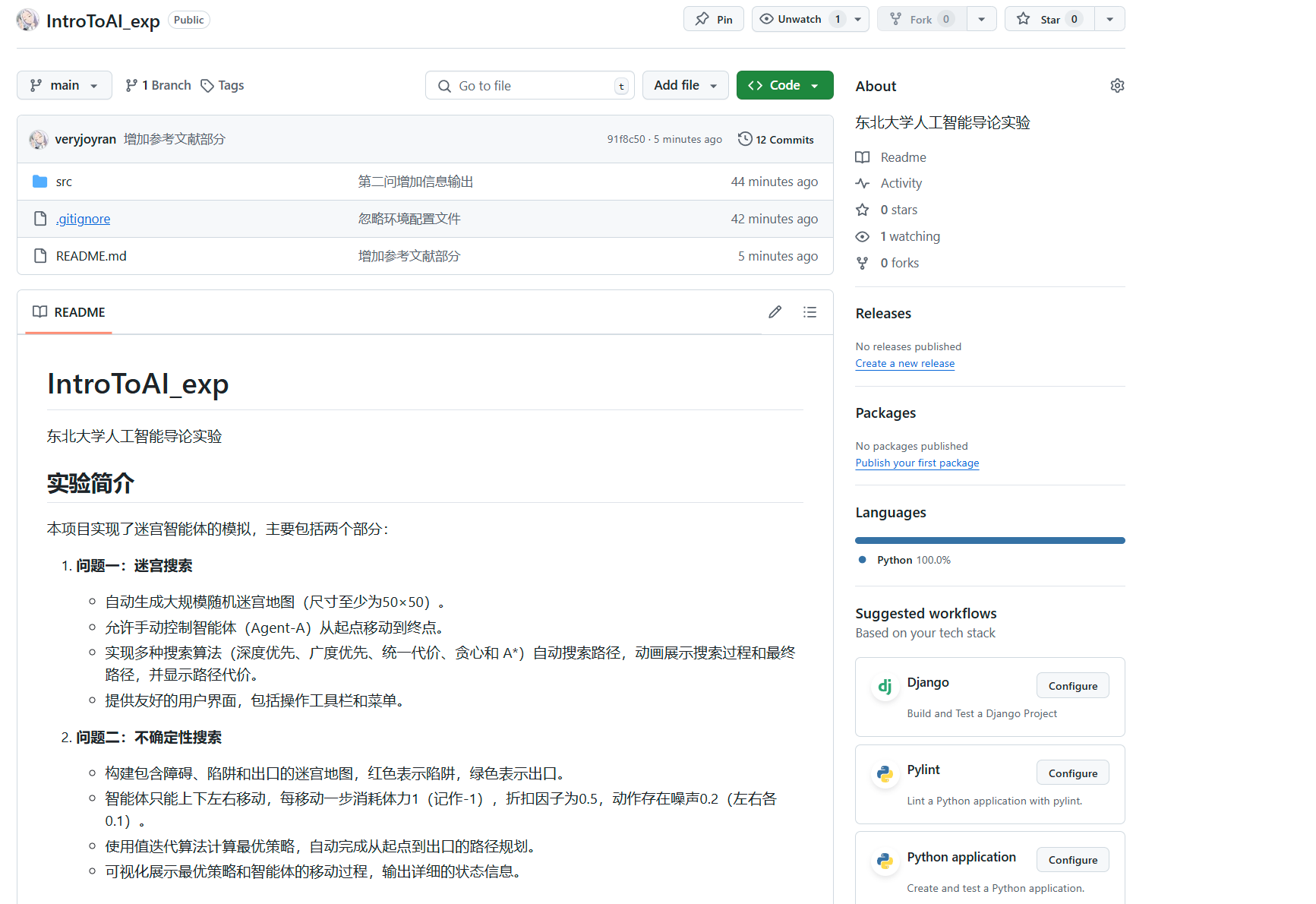
实验一：迷宫搜索

人工智能2202班-刘庆然-20226494

所有代码以及README均已上传至github: <https://github.com/veryjoyran/IntroToAI_exp>



目录

[一、 问题描述 3](#_Toc181925799)

[二、问题描述与解决方案 4](#_Toc181925800)

[三、实验结果 5](#_Toc181925801)

[预定义地图部分： 6](#_Toc181925802)

[1.手动搜索(wasd控制上下左右)： 6](#_Toc181925803)

[2.DFS 8](#_Toc181925804)

[3.BFS 9](#_Toc181925805)

[4.统一代价搜索 10](#_Toc181925806)

[5.贪心搜索 11](#_Toc181925807)

[6.A\*搜索 12](#_Toc181925808)

[随机生成地图部分 13](#_Toc181925809)

[1.输入地图尺寸、起终点坐标 14](#_Toc181925810)

[2.刷新地图 17](#_Toc181925811)

[3.DFS 21](#_Toc181925812)

[4.BFS 26](#_Toc181925813)

[5.统一代价搜索 36](#_Toc181925814)

[6.贪心搜索 42](#_Toc181925815)

[7.A\*搜索 46](#_Toc181925816)

[四、源码解读 53](#_Toc181925817)

[Search.py: 53](#_Toc181925818)

[SearchAlgorithm: 53](#_Toc181925819)

[DFS: 55](#_Toc181925820)

[BFS： 56](#_Toc181925821)

[UniformCostSearch： 57](#_Toc181925822)

[GreedySearch： 58](#_Toc181925823)

[AStarSearch： 59](#_Toc181925824)

[源代码： 60](#_Toc181925825)

[My\_Map.py: 65](#_Toc181925826)

[My\_Map: 65](#_Toc181925827)

[变量 65](#_Toc181925828)

[函数 65](#_Toc181925829)

[整体解读 67](#_Toc181925830)

[源代码： 68](#_Toc181925831)

[problem1.py: 78](#_Toc181925832)

[解读： 78](#_Toc181925833)

[代码结构与核心功能 78](#_Toc181925834)

[源代码： 79](#_Toc181925835)

# 问题描述

编写程序实现一个迷宫智能体。 具体要求如下：

1.自动构建地图。通过程序自动生成地图文件（文件名.map，纯文本文件 utf-

8 编码），格式如图 1 所示， 1 表示墙， 0 表示通路， @表示起点， $表示终

点。 根据该地图文件创建迷宫地图，如图 2 所示（风格自拟）。 要求地图尺寸

在 50\*50 以上， 道路和墙的宽度都为一个单位（方格）。

2. 手动搜索。 通过移动（上下左右箭头控制，有墙壁阻挡不可移动） 操控智能

体（Agent-A） 从入口移动到出口，智能体可以自己设置一个图标。

3. 自动搜索：通过采用深度优先、广度优先、一致代价、贪心和 A\*不同搜索策

略，确定搜索路径并显示出来。 控制智能体移动速度，动画展示最后的路径，

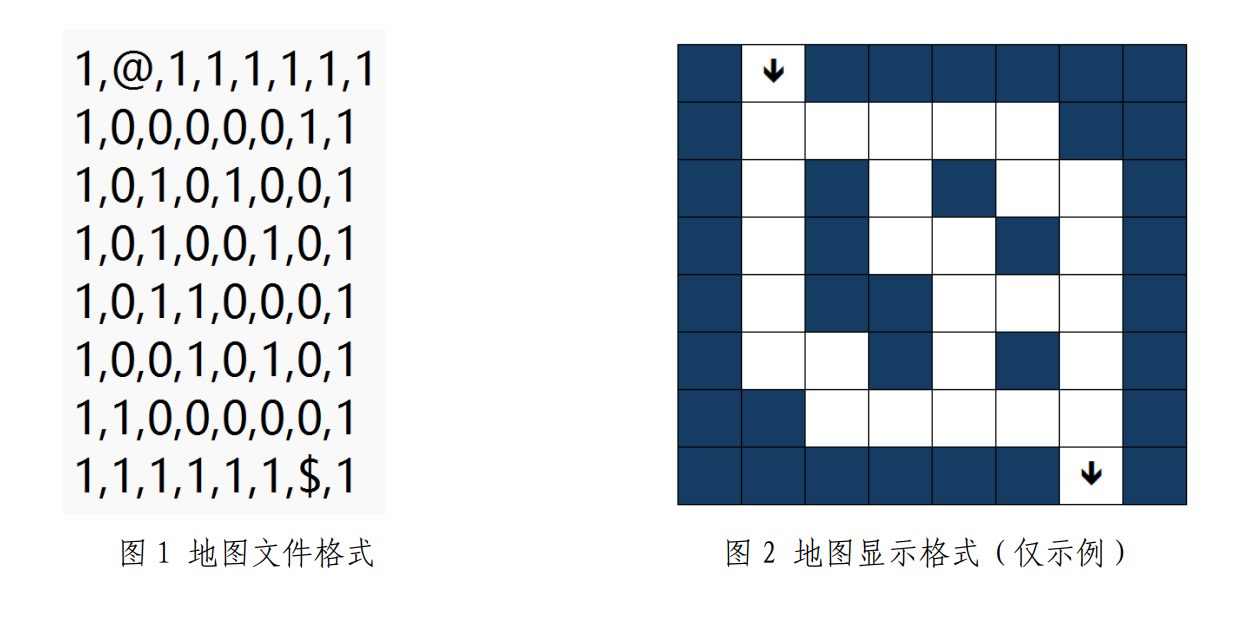
并显示路程代价（一个方格消耗 1）。

4. 对抗搜索：地图中随机位置（除了出入口）设置一个智能体（Agent-B），

Agent-A 和 Agent-B 每秒均移动一次。 Agent-B 随机游走， 若 Agent-B 吃

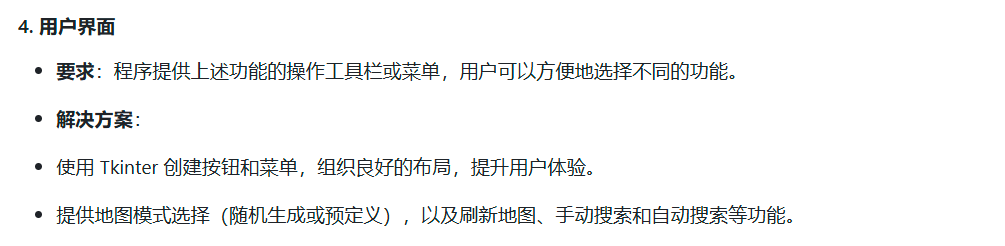
掉 Agent-A 游戏结束； Agent-A 采用 MinMax 搜索策略，动态展示行动路径到出口（选作）。

5. 程序给出上述功能的操作工具栏或菜单。



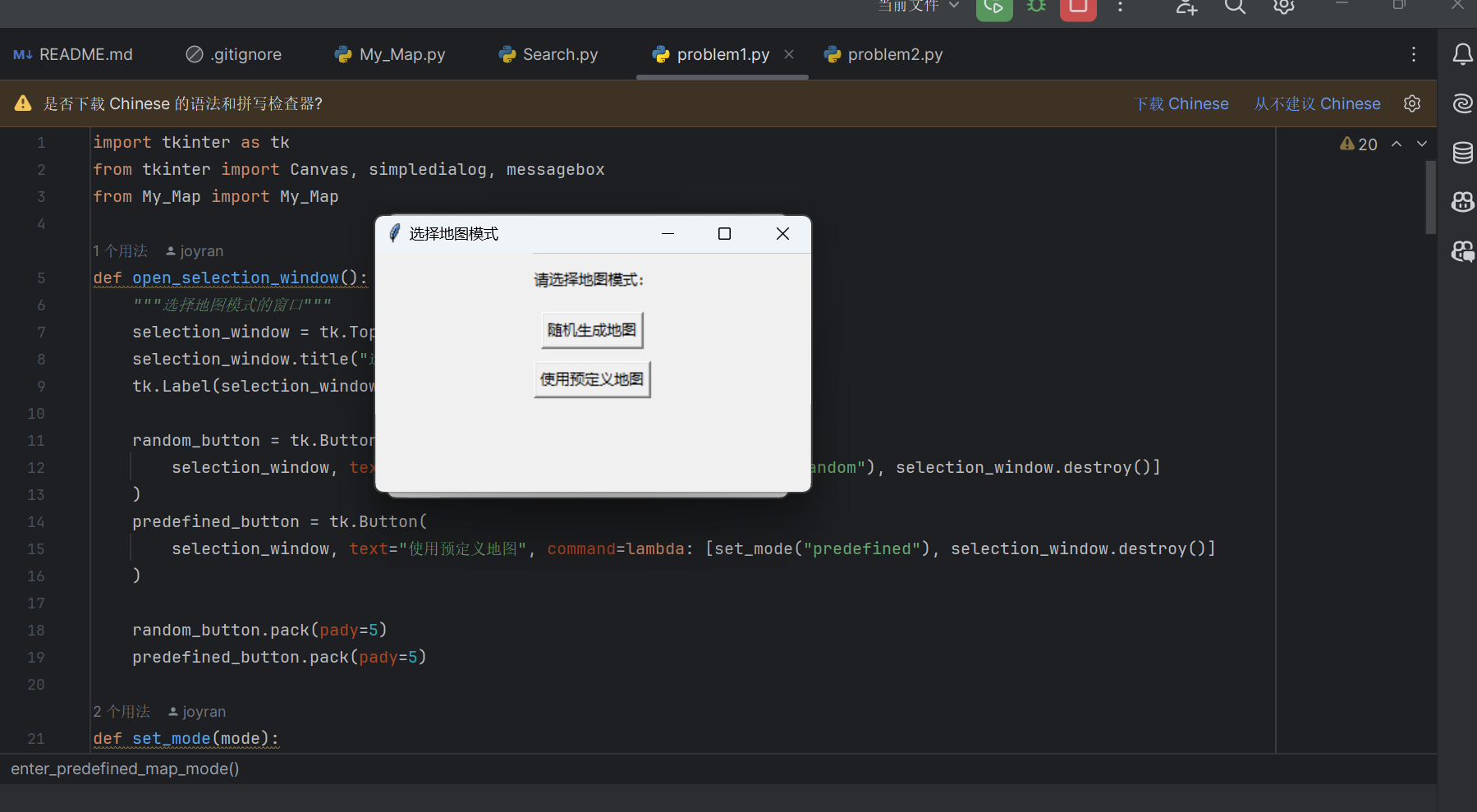
# 二、问题描述与解决方案





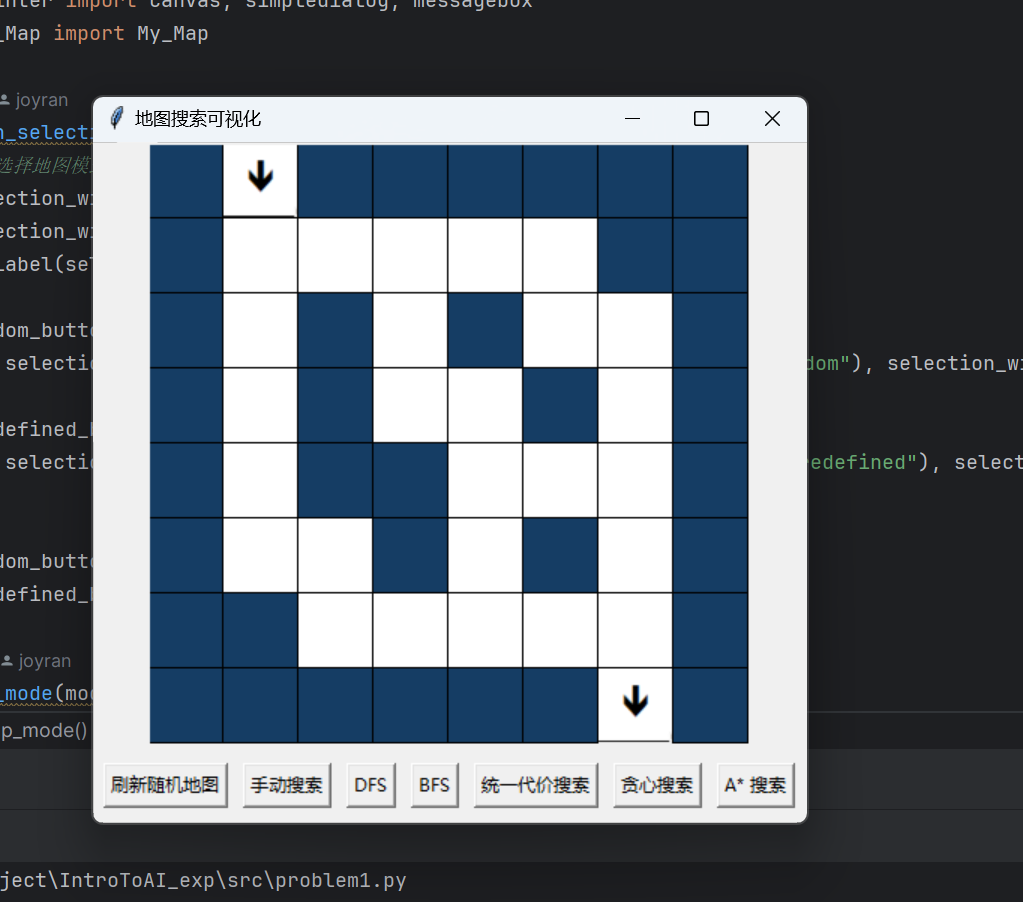
# 三、实验结果

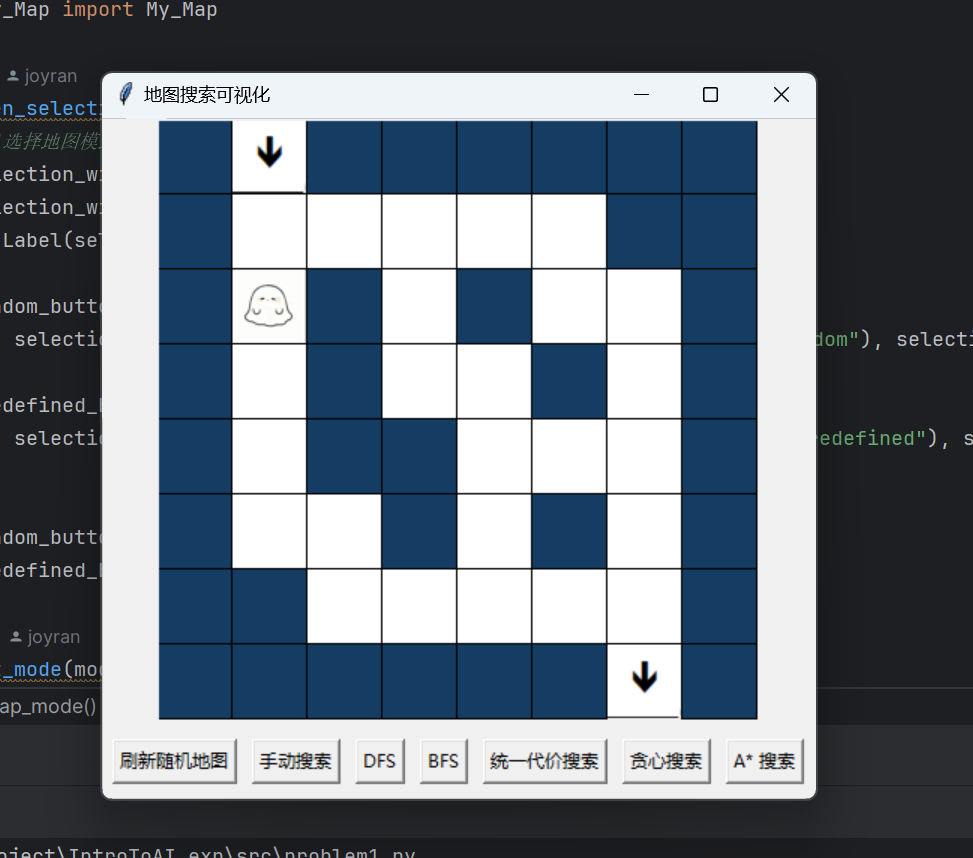
首先是模式选择：

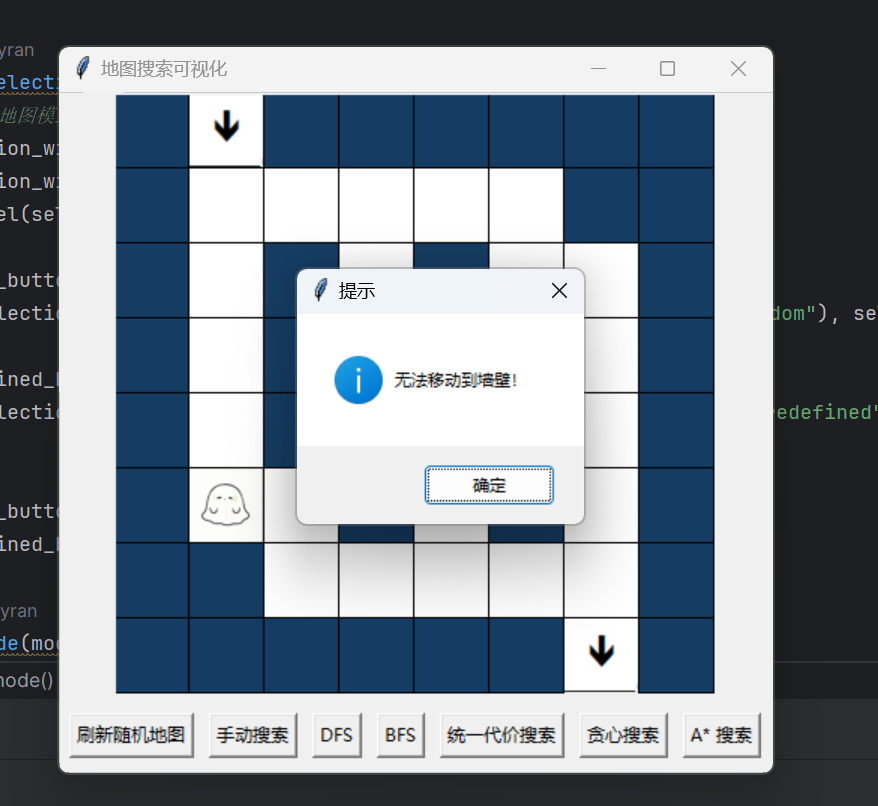


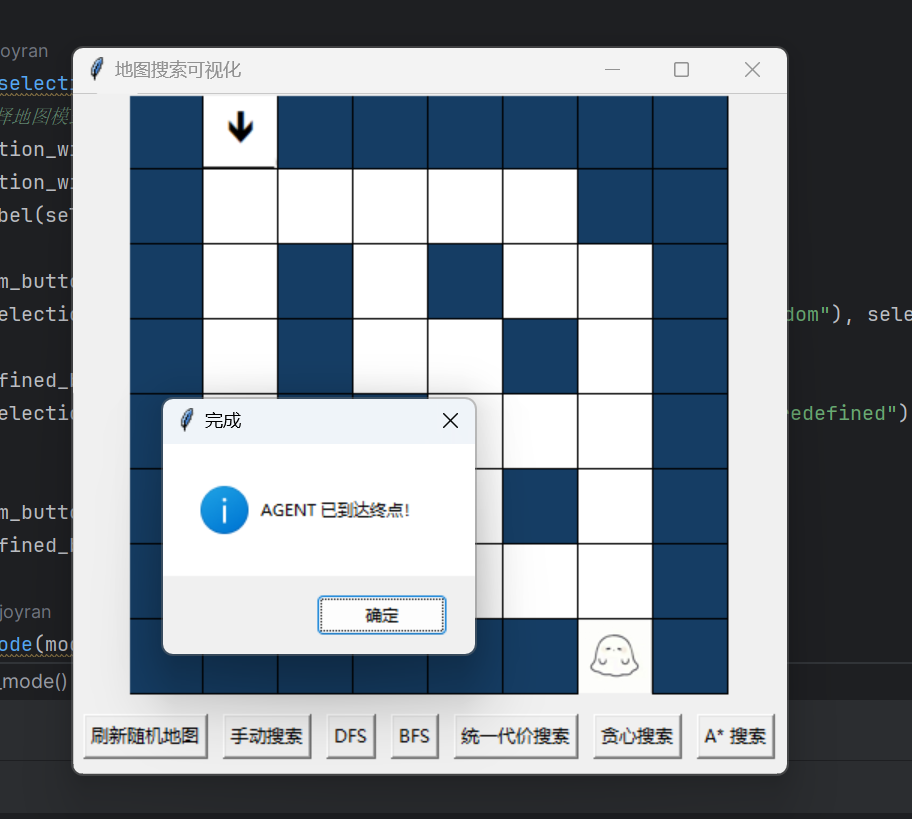
## 预定义地图部分：

### 1.手动搜索(wasd控制上下左右)：

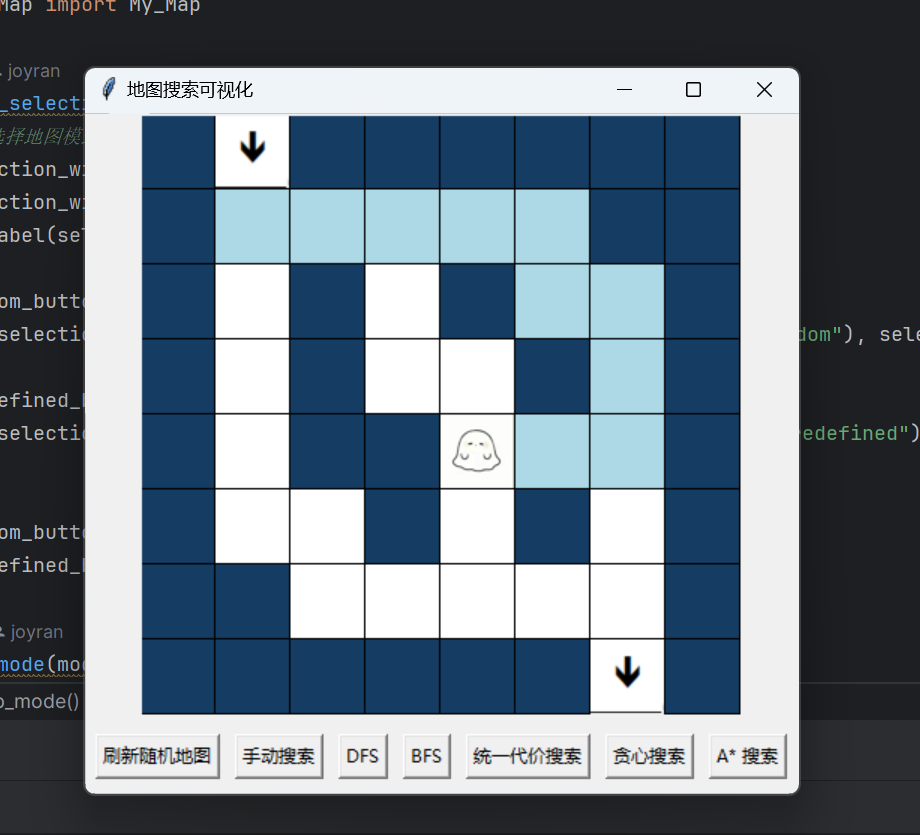


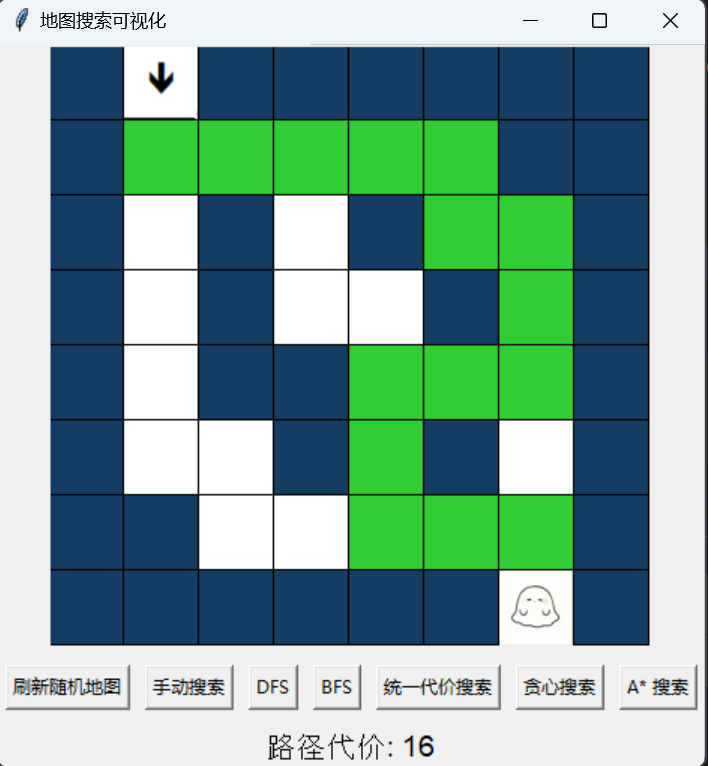




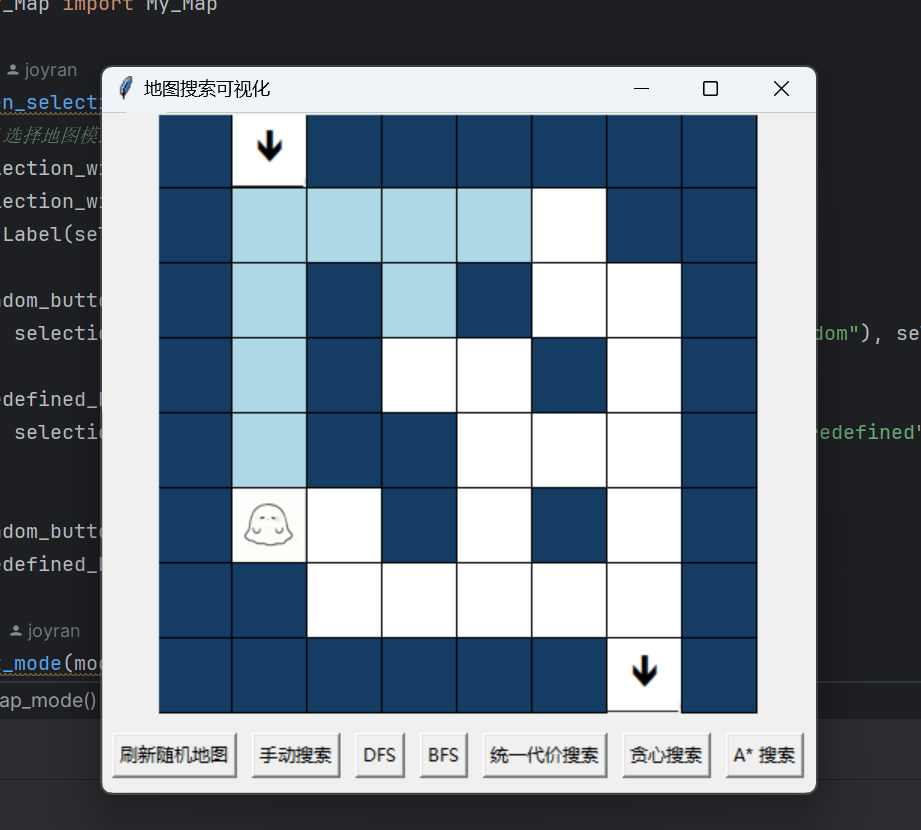


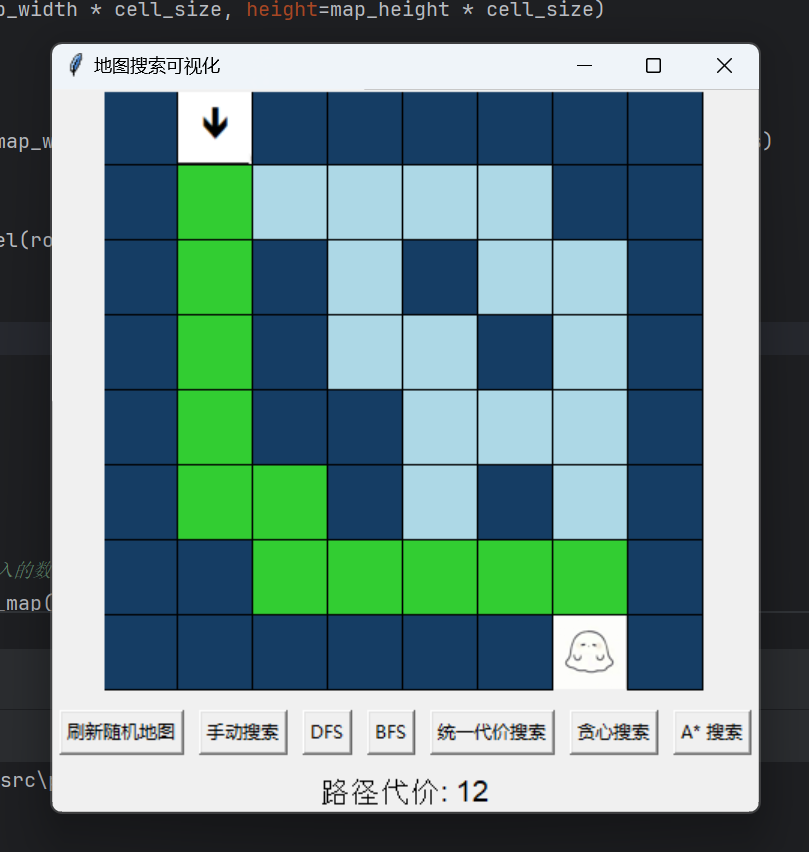
### 2.DFS



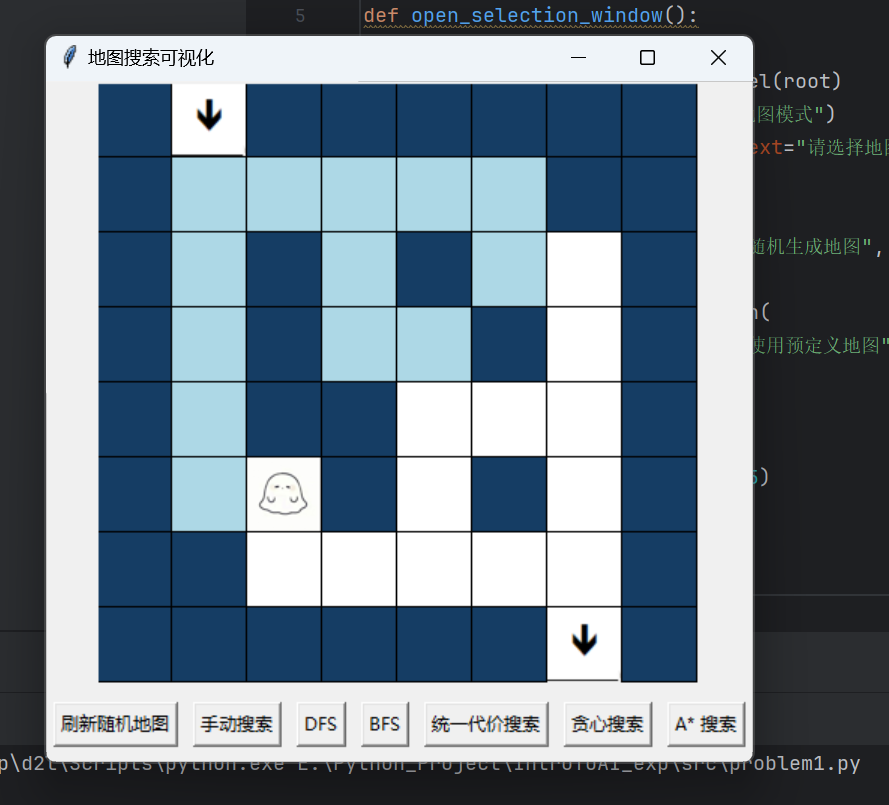


### 3.BFS



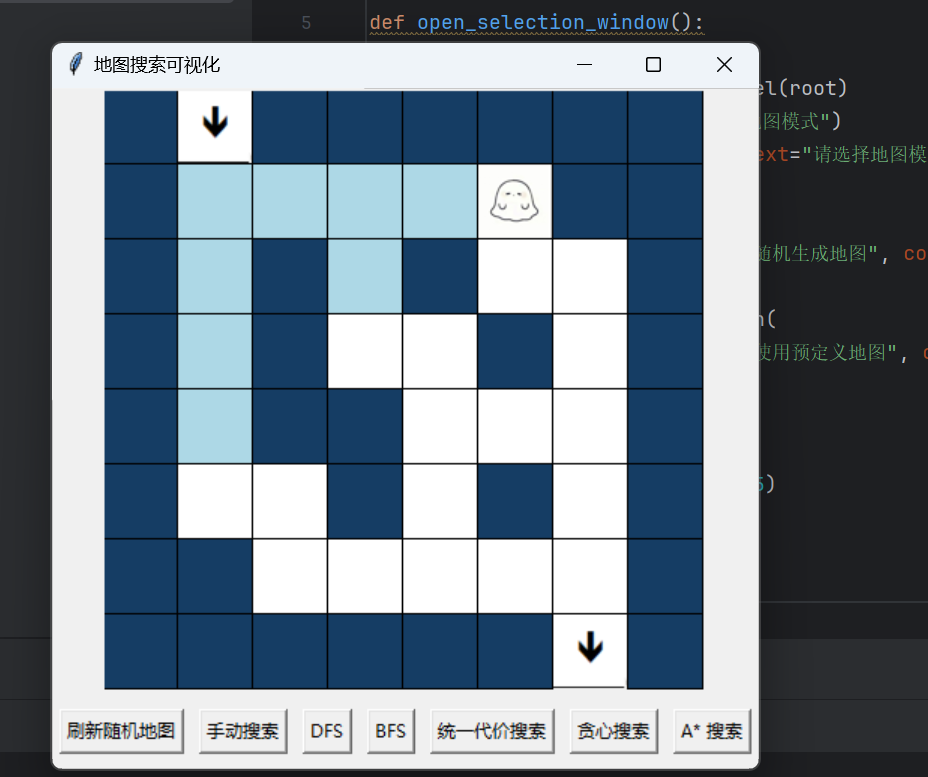


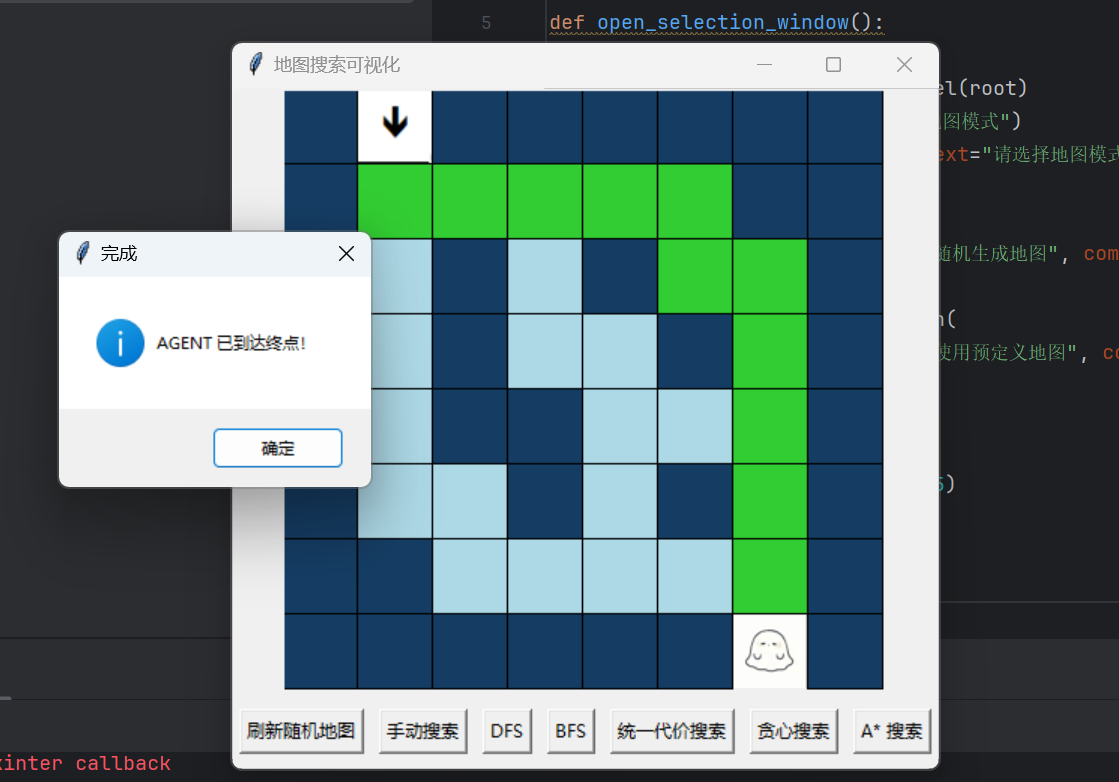
### 4.统一代价搜索



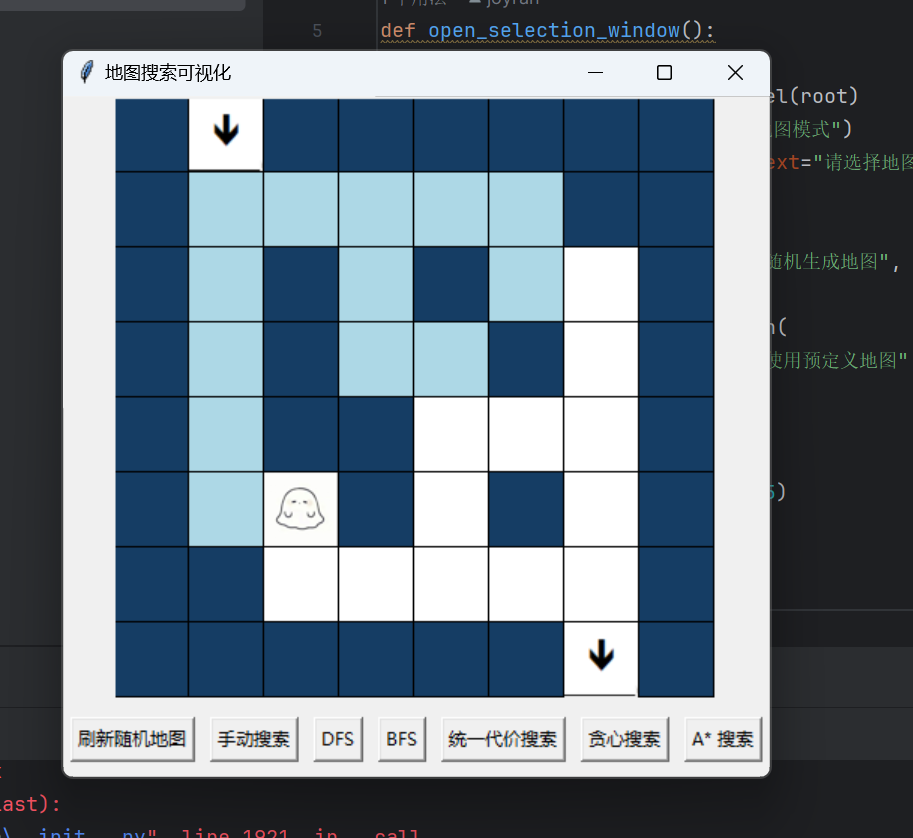


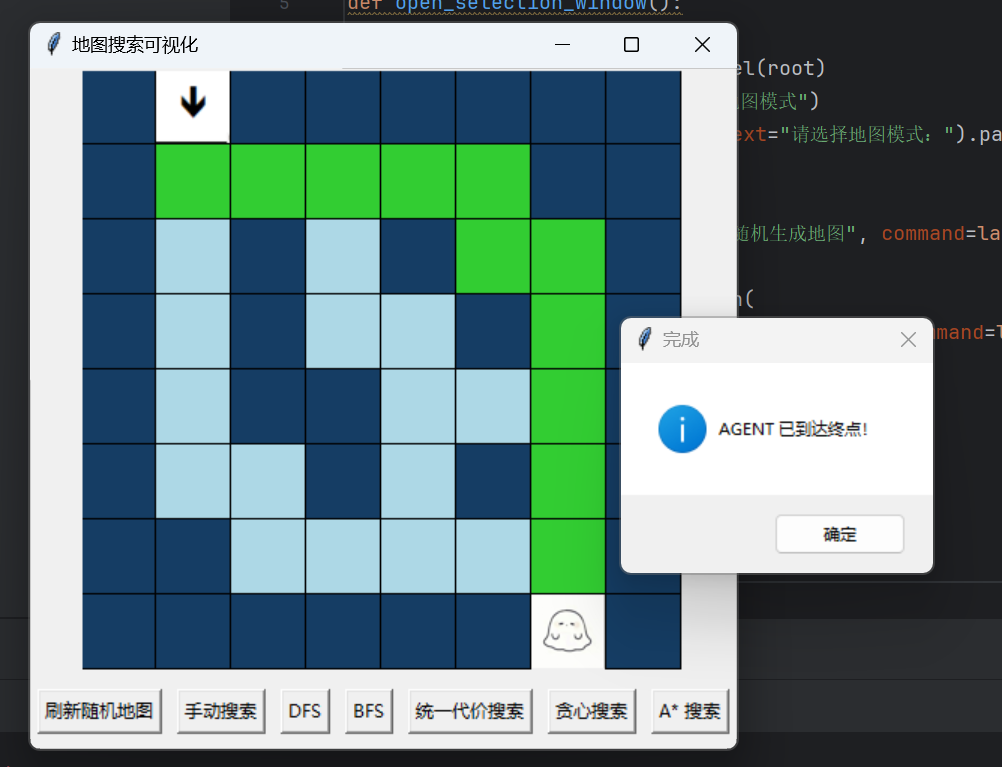
### 5.贪心搜索





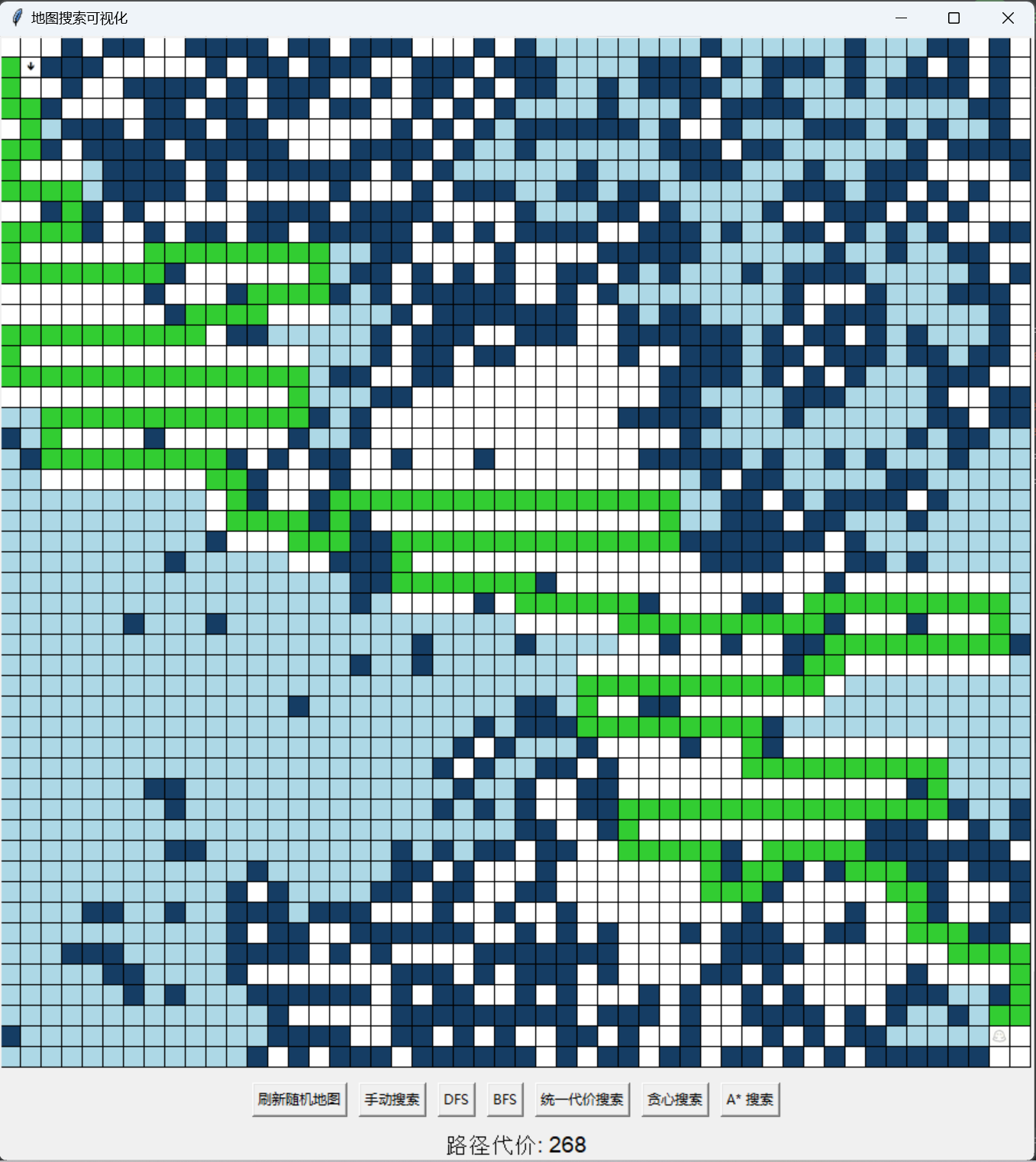
### 6.A\*搜索



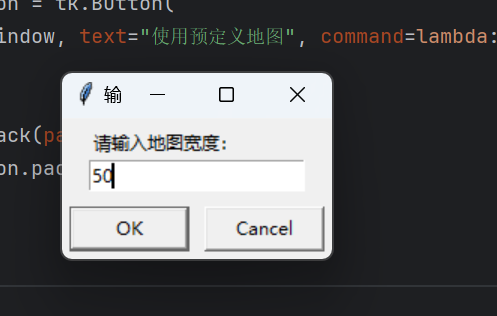


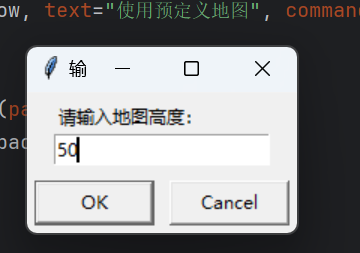
## 随机生成地图部分

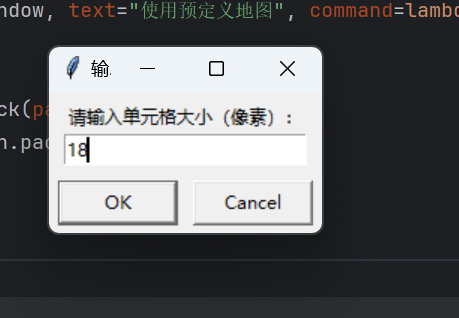
首先附一张带有路径代价的照片（由于带有代价计算过慢，后续展示各个算法的时候便没有带有代价了，在次只是为了说明含有路径代价显示功能）：

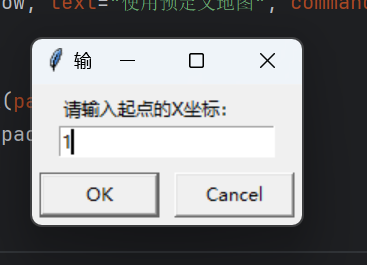


### 1.输入地图尺寸、起终点坐标

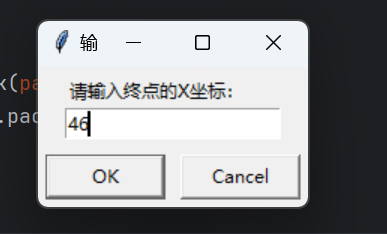


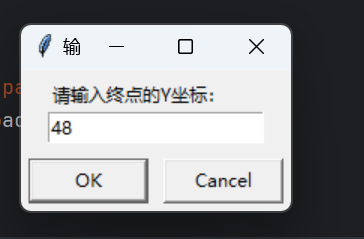


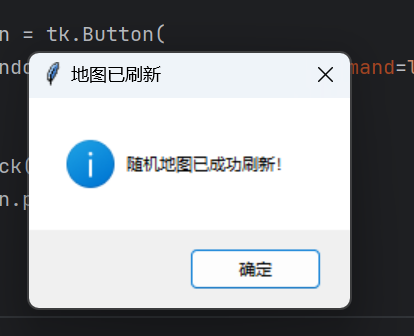




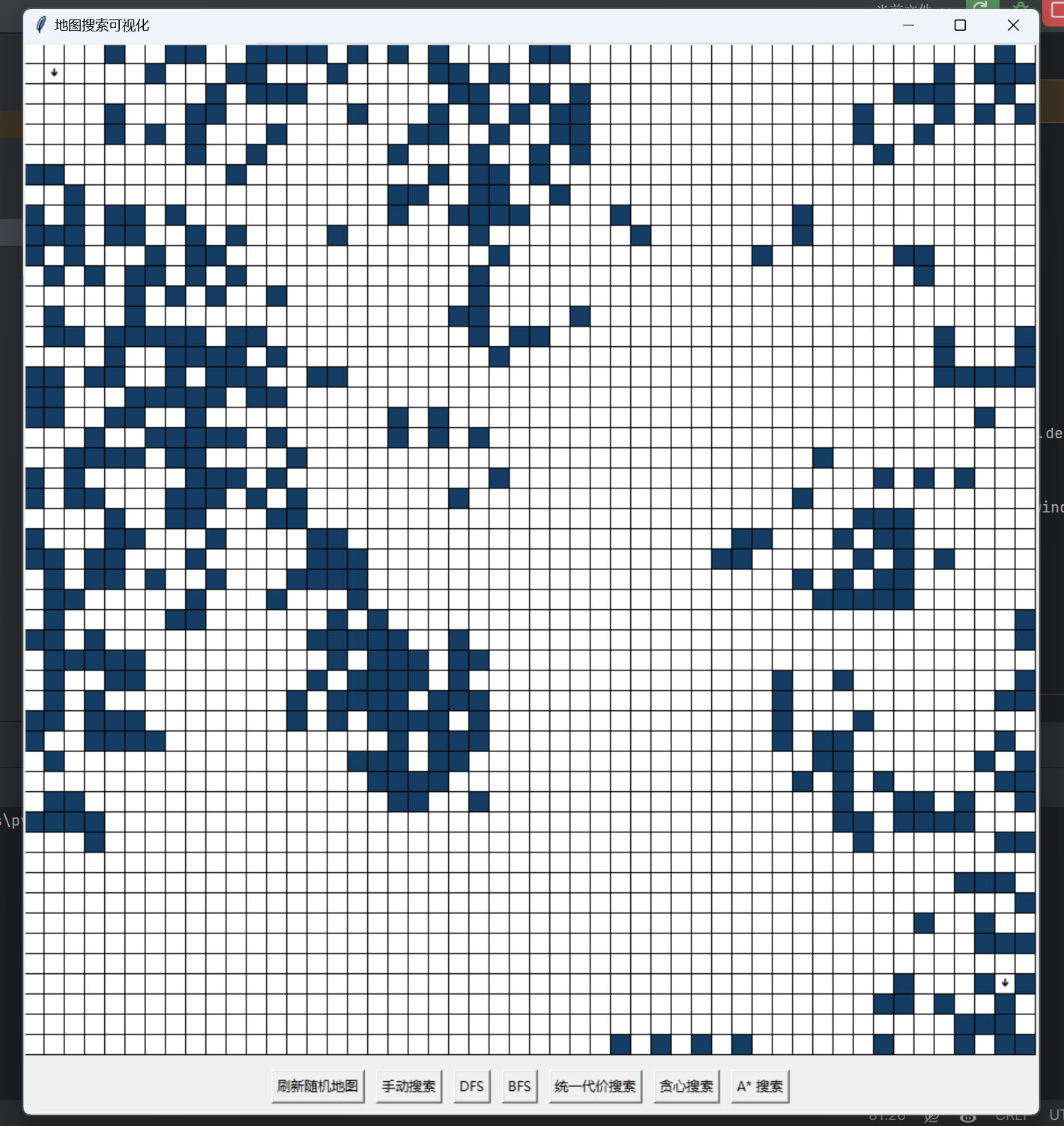




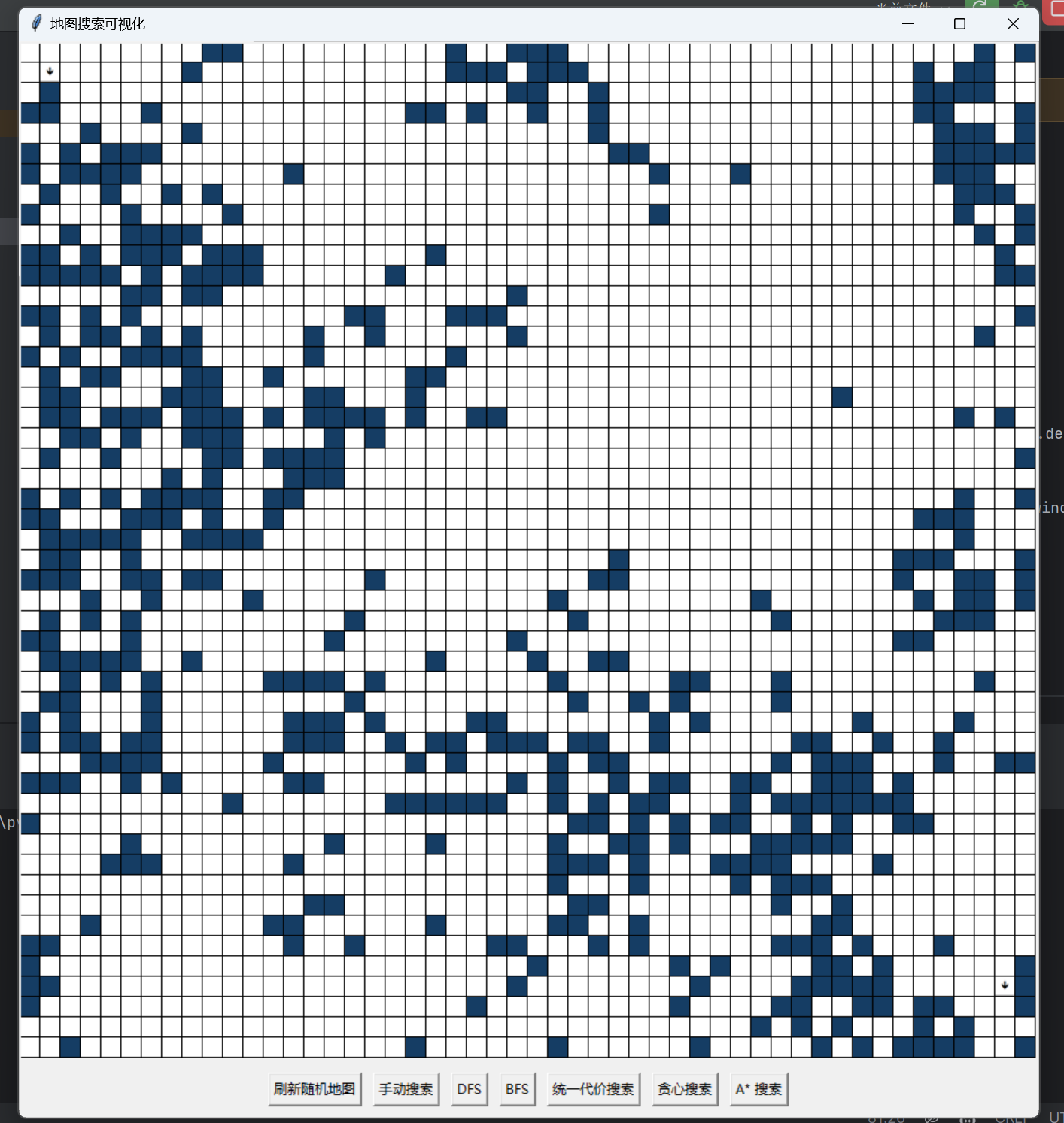




### 2.刷新地图









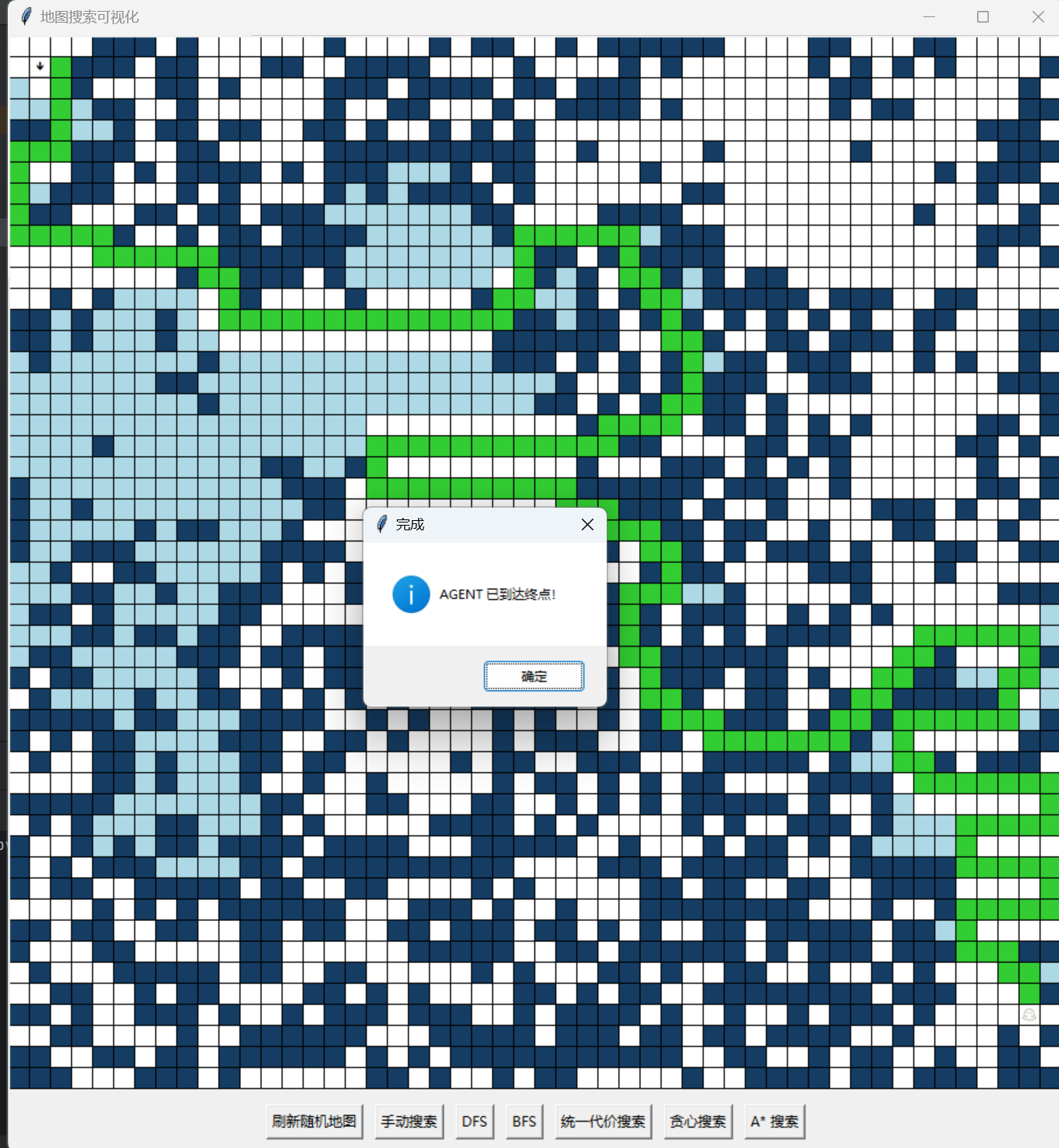
### 3.DFS











### 4.BFS





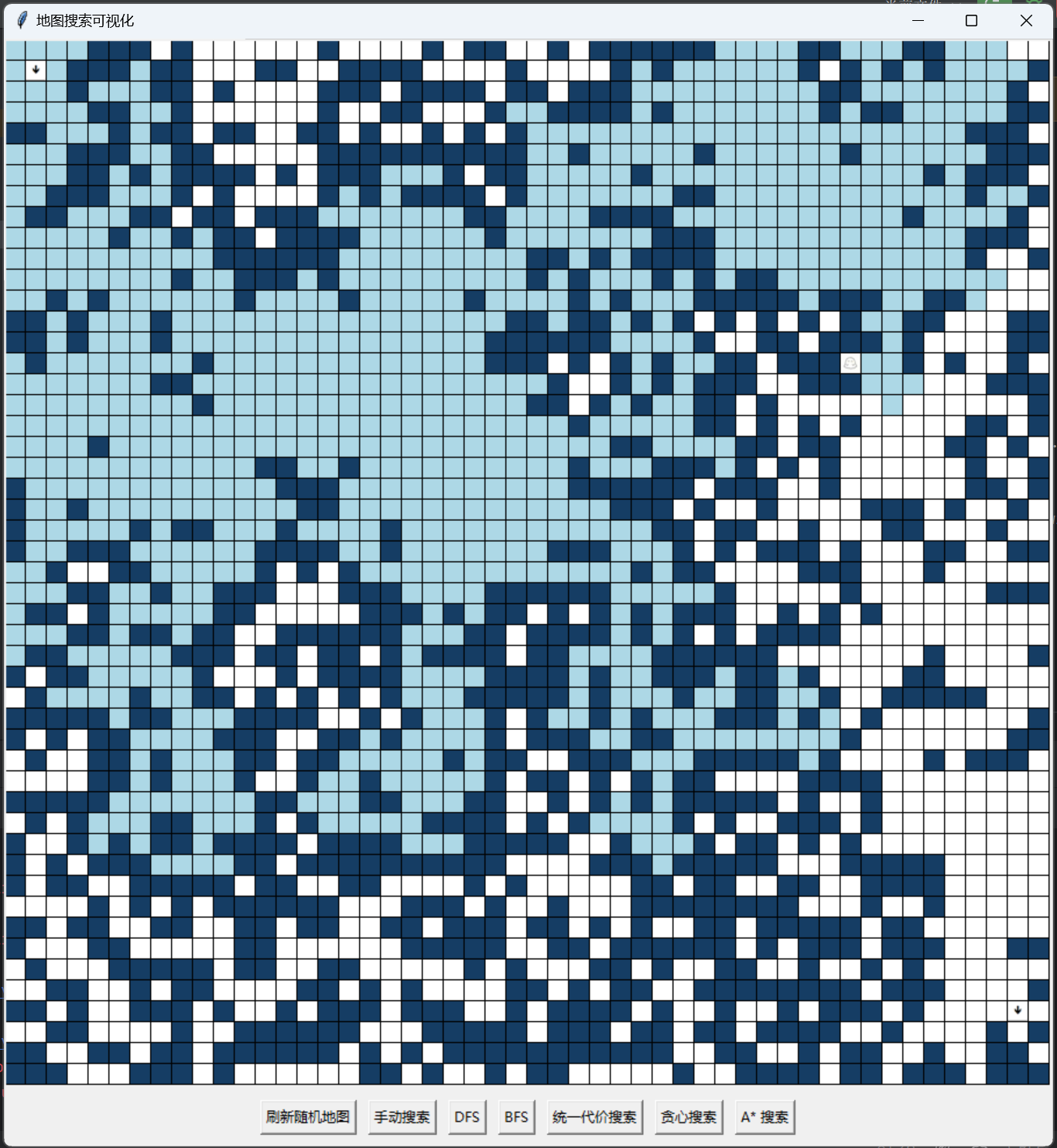


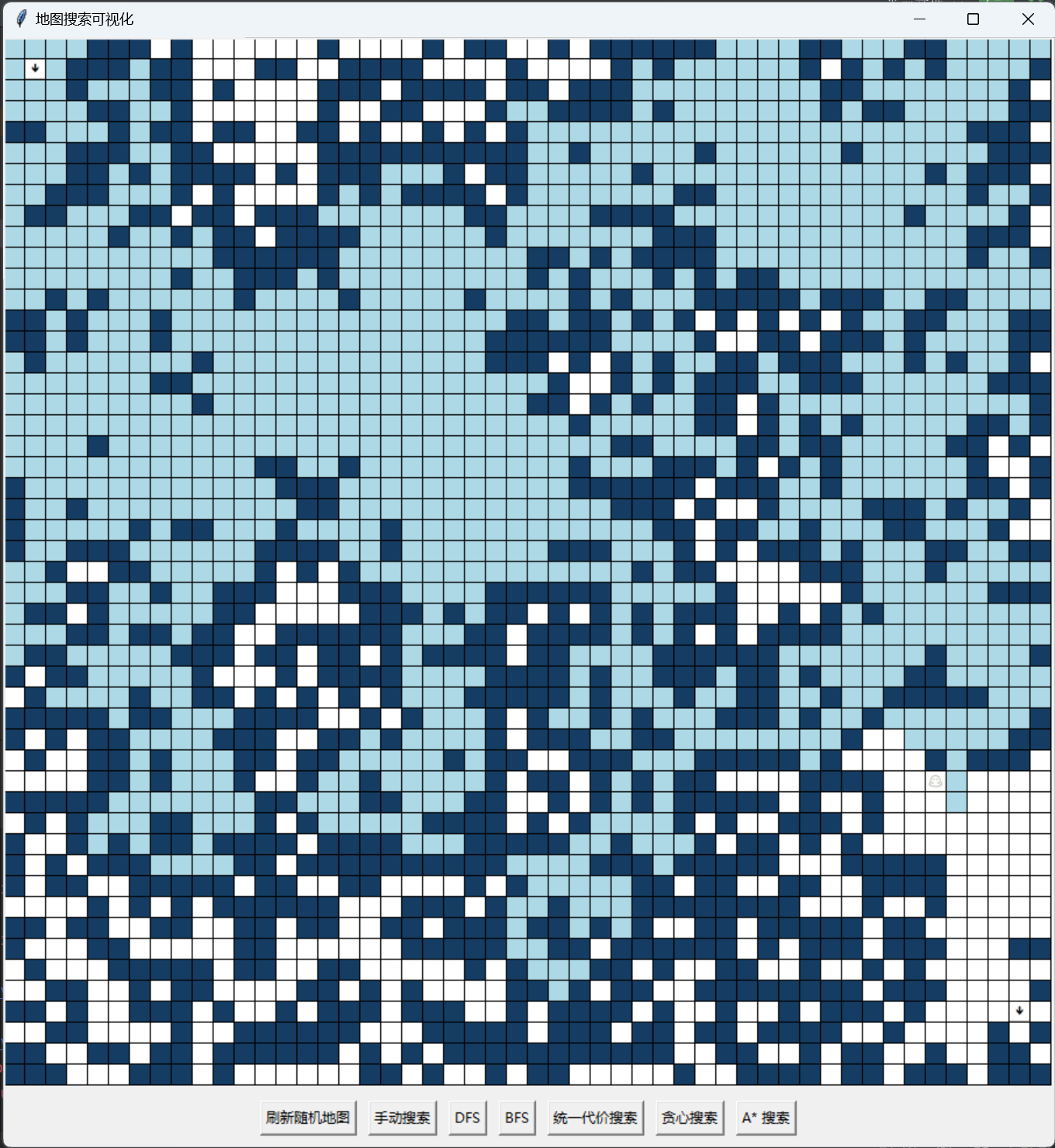


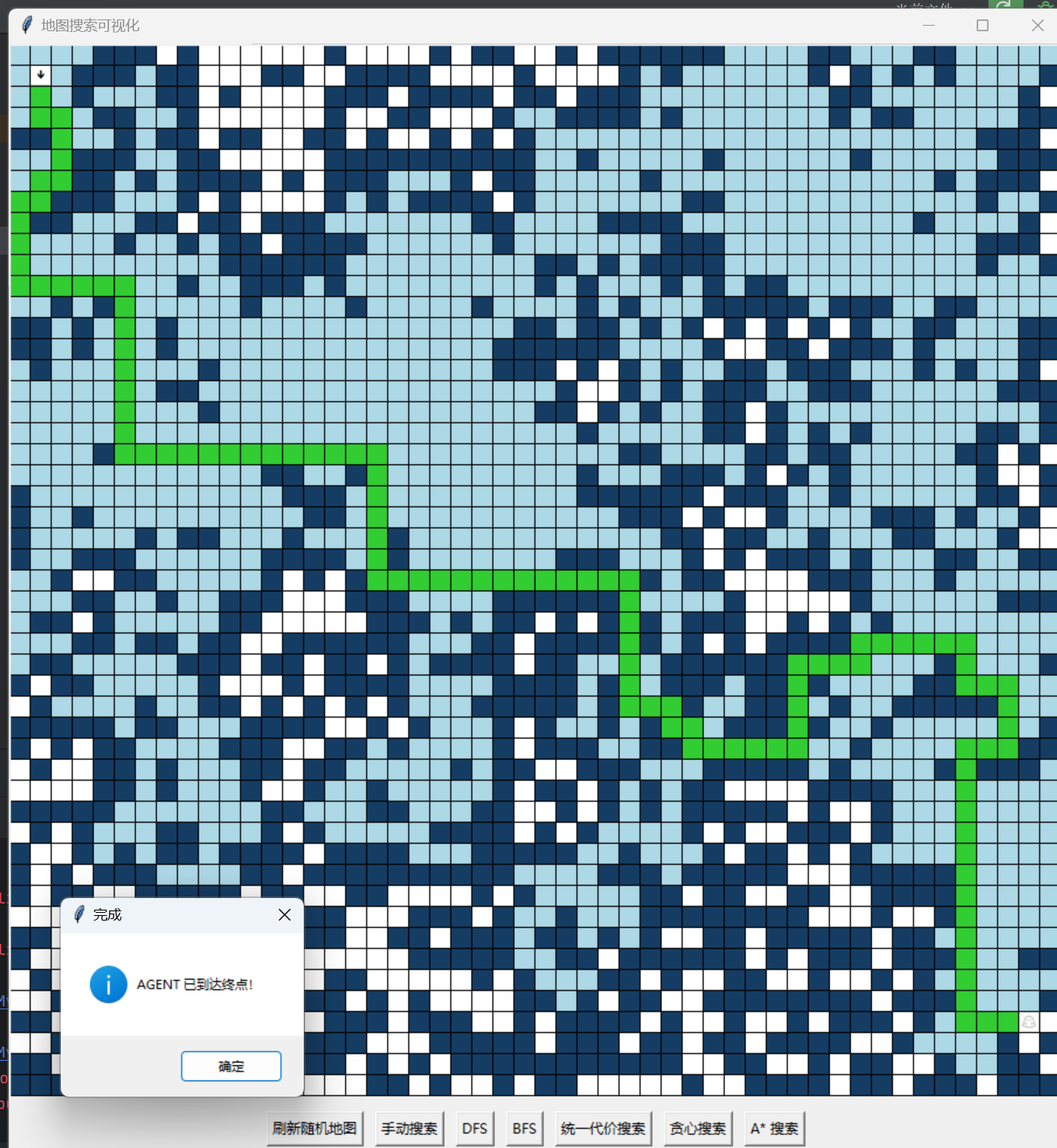












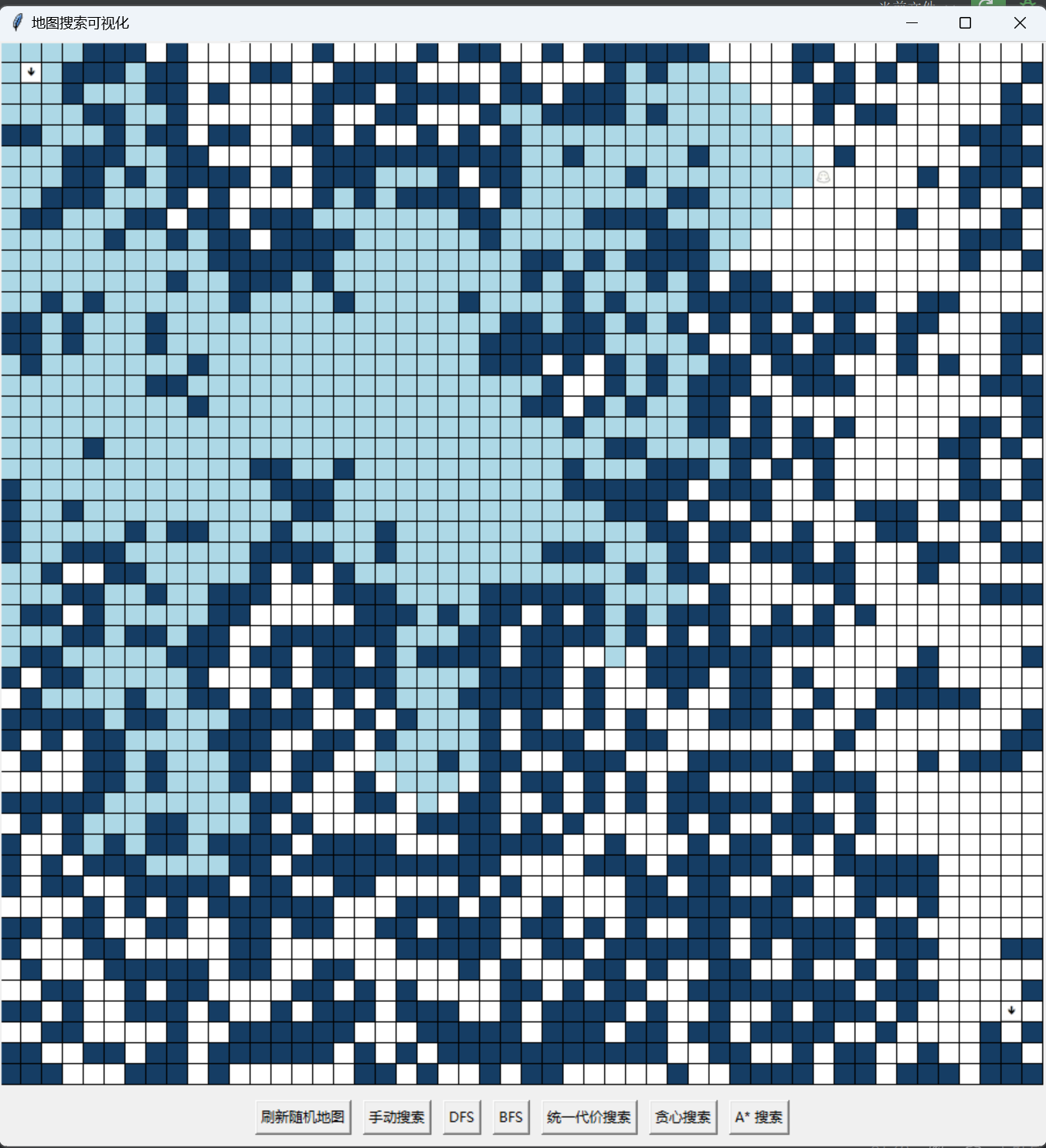
4

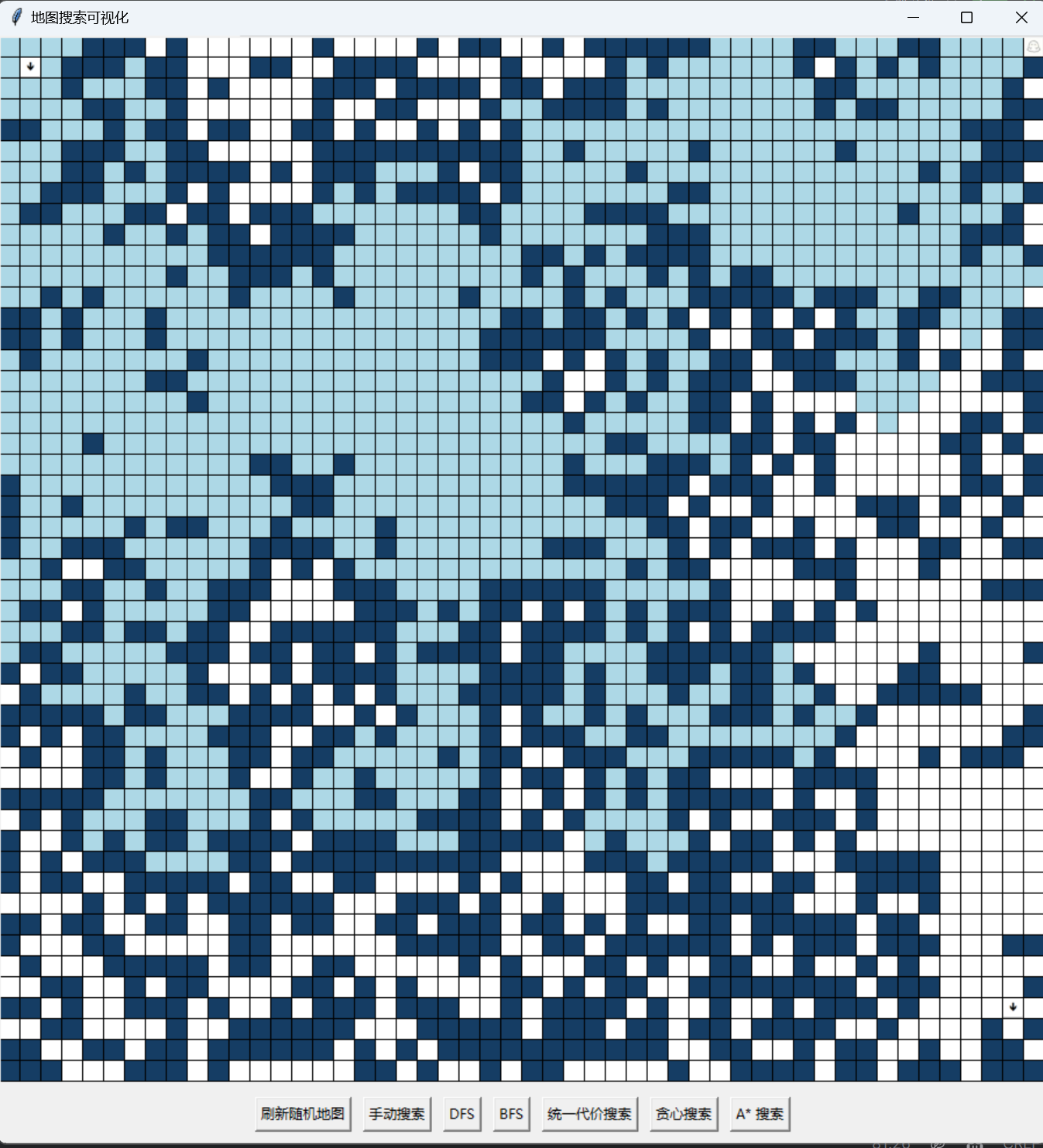
### 5.统一代价搜索

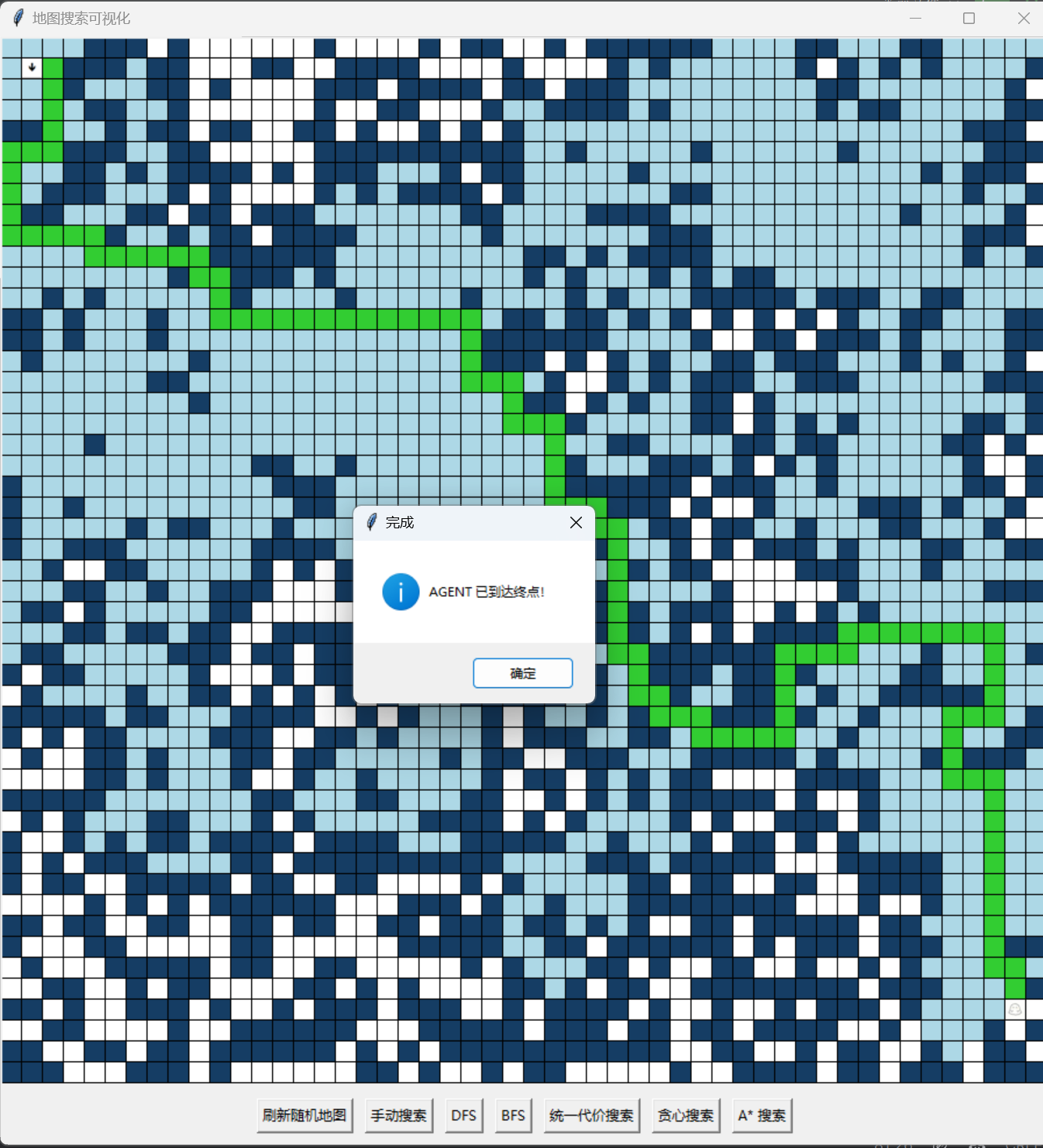






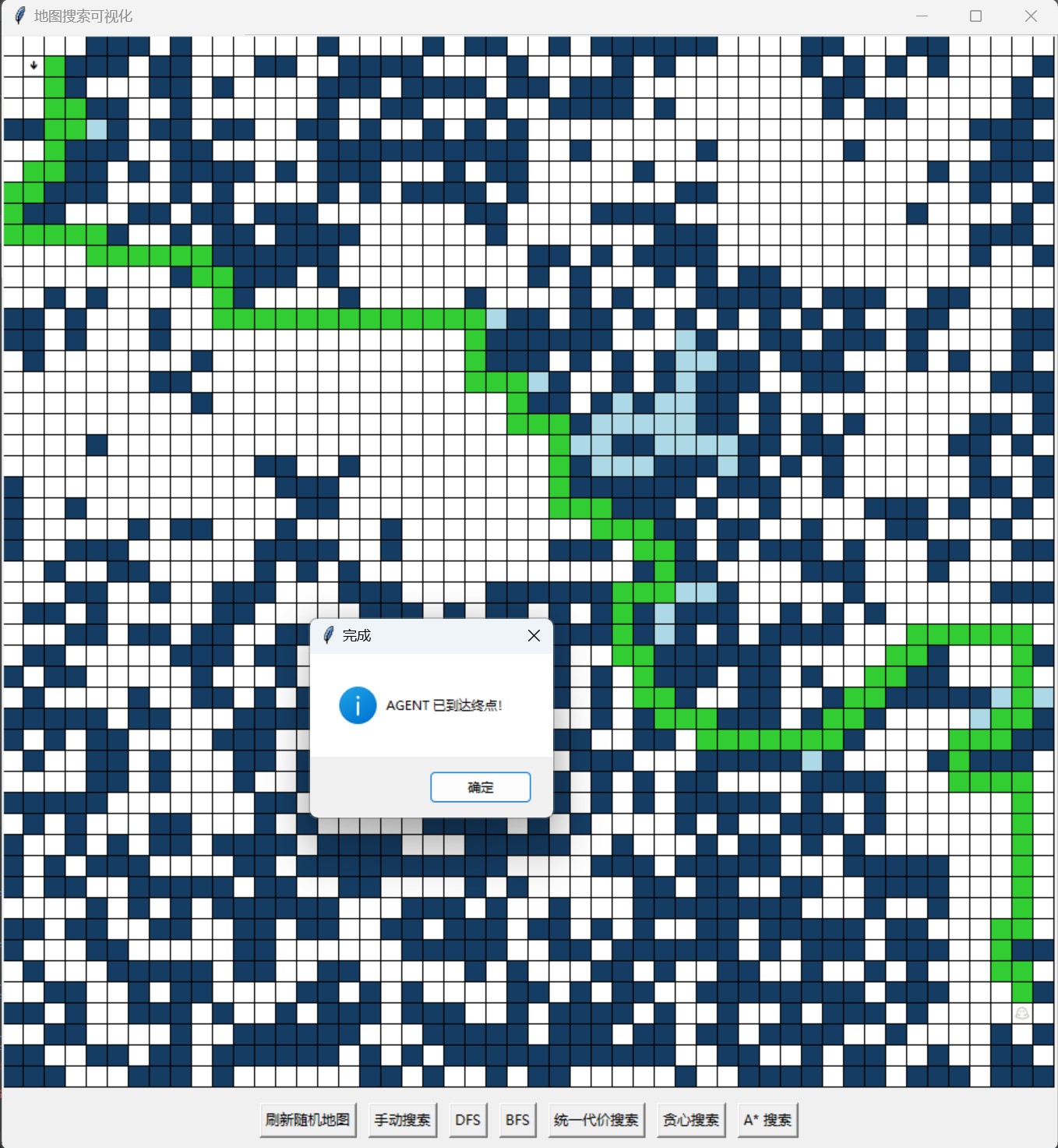




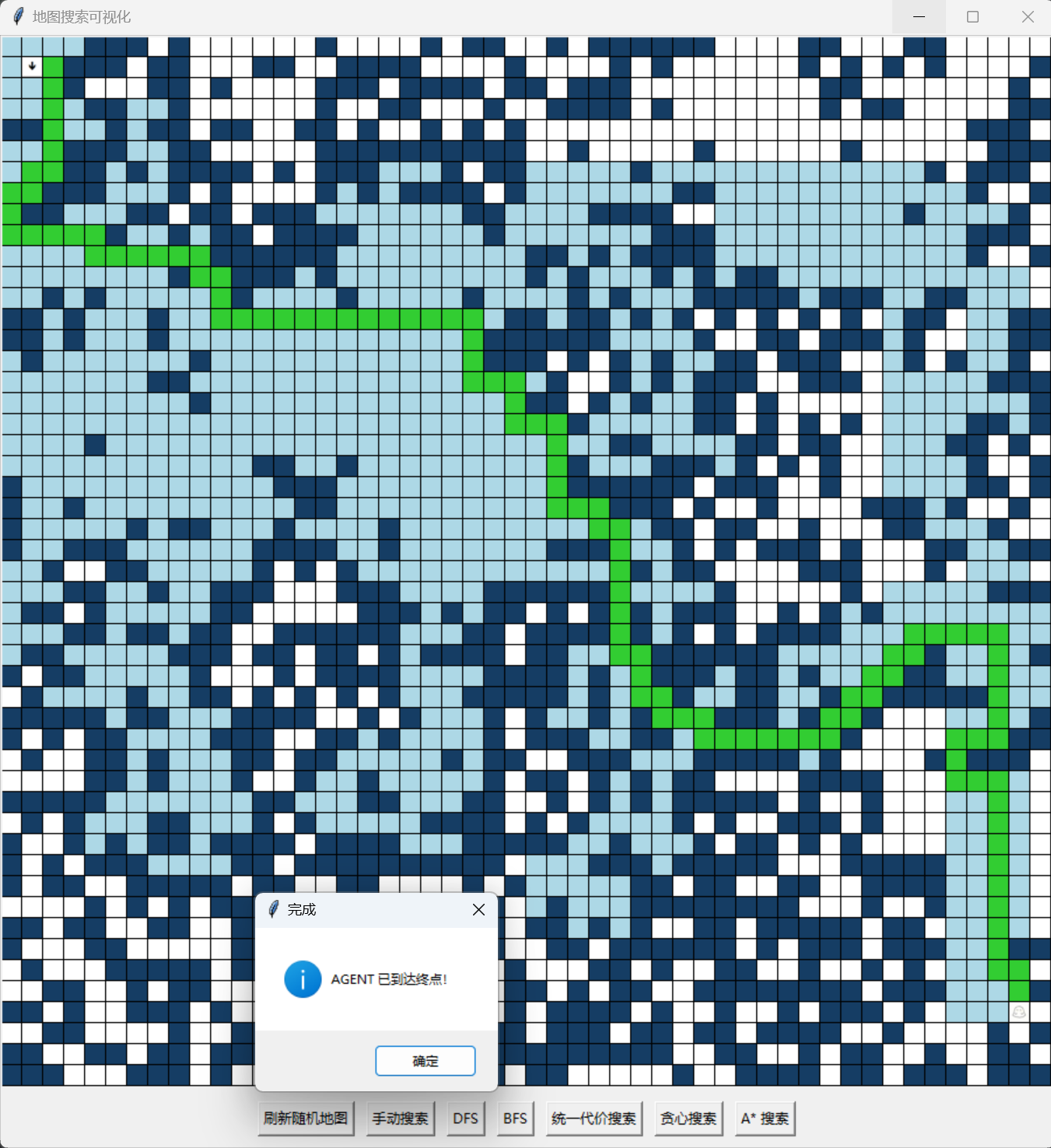
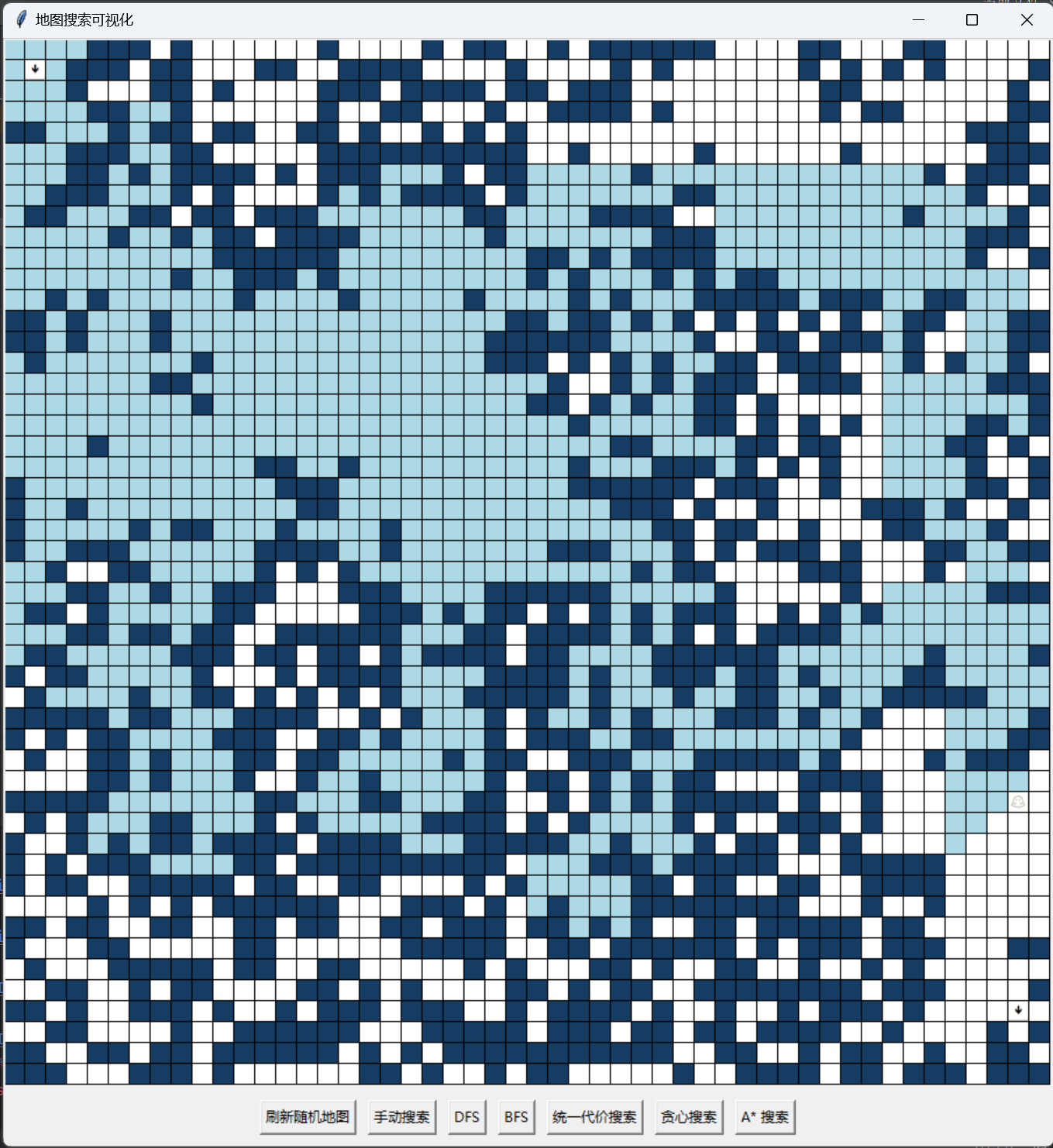
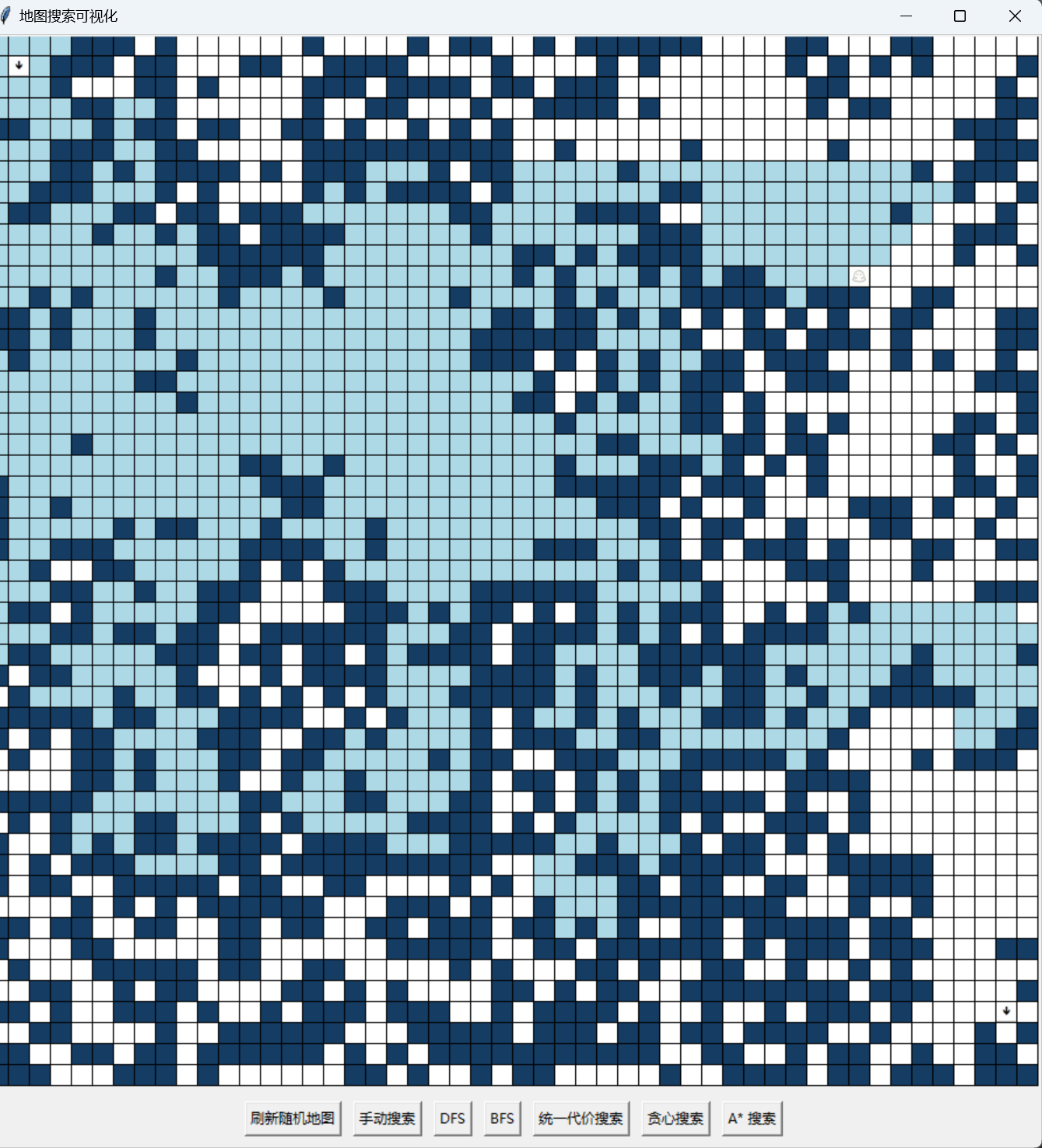
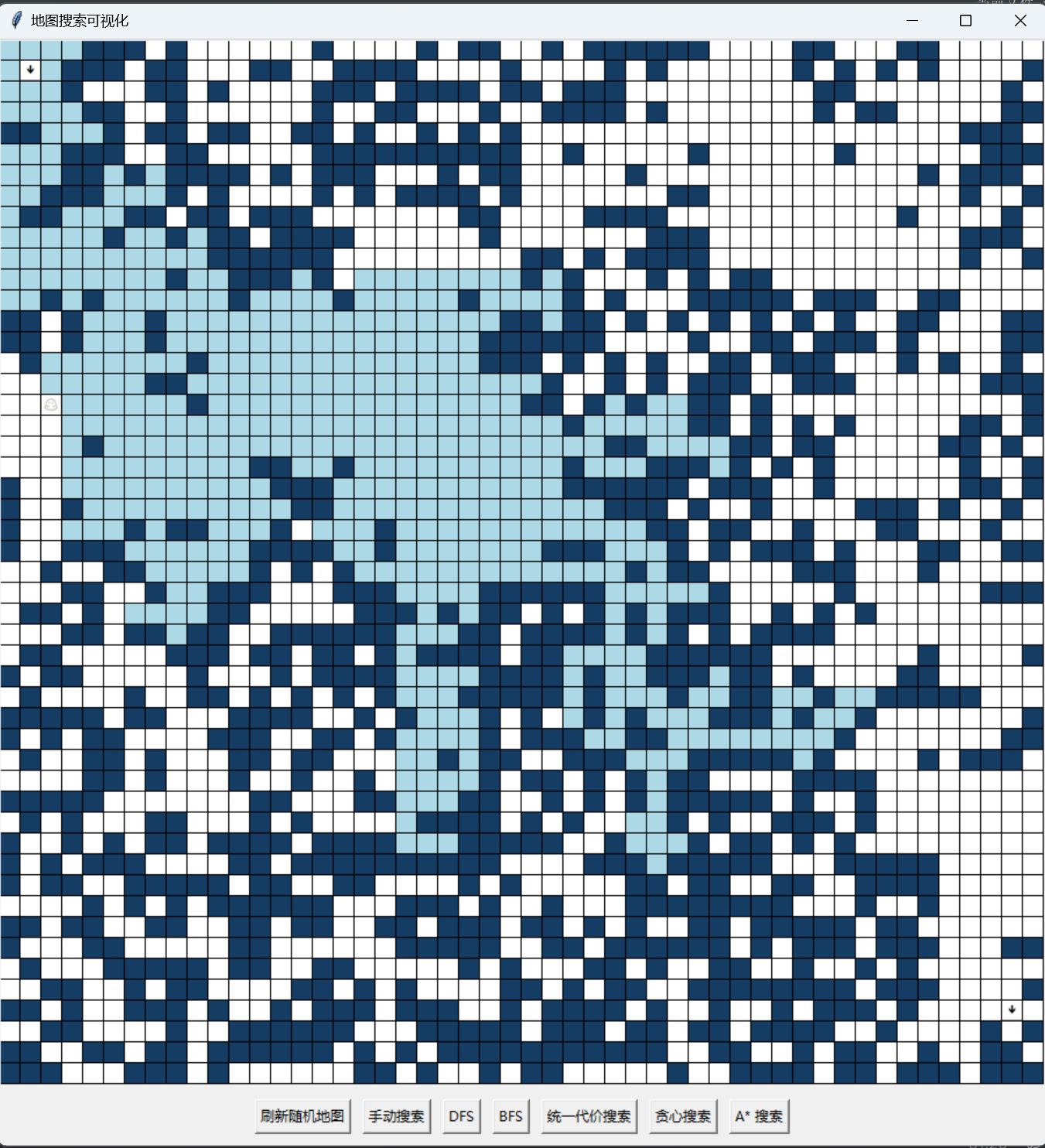
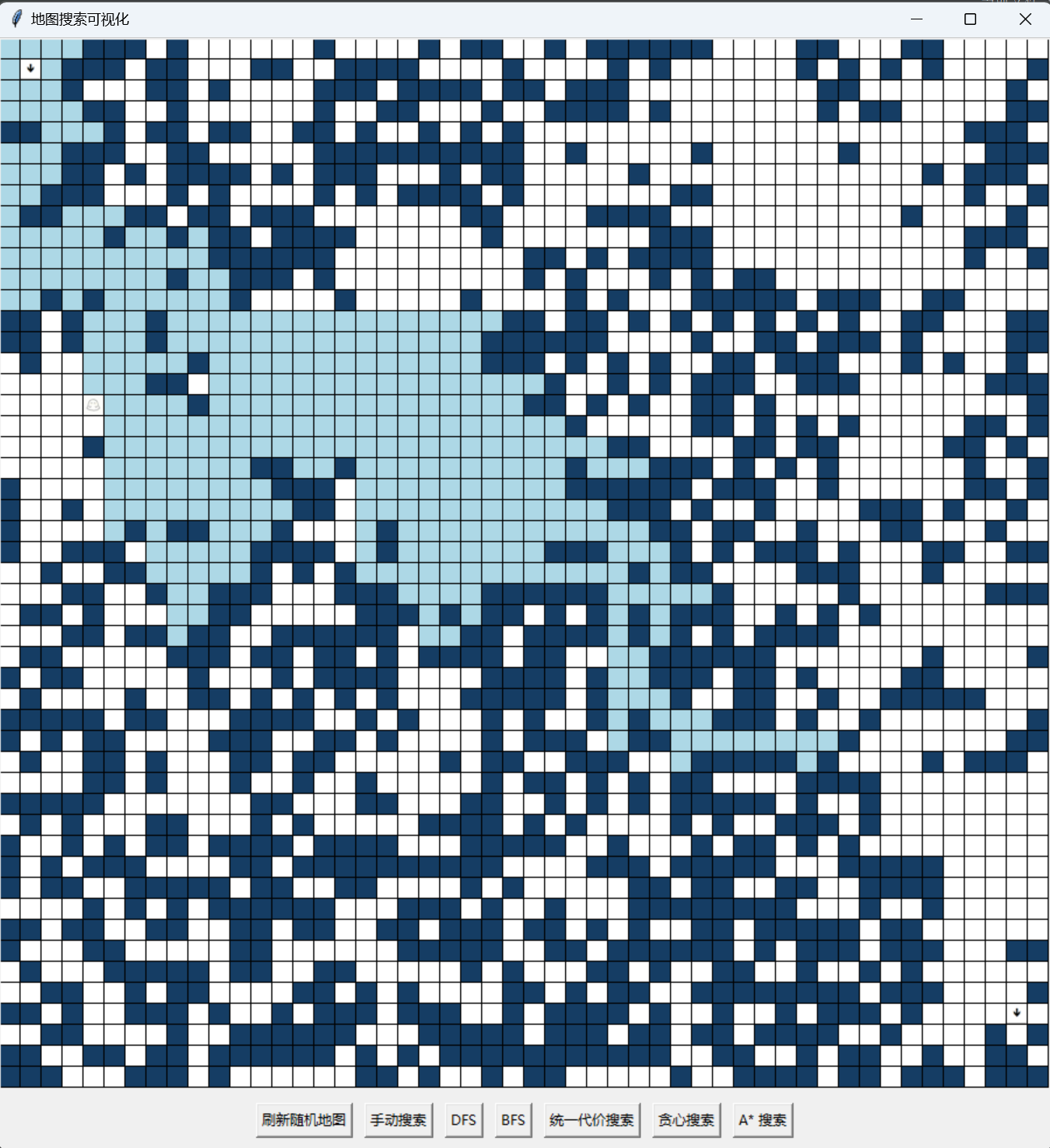


### 6.贪心搜索





### 7.A\*搜索



# 四、源码解读

## Search.py:

### SearchAlgorithm:

#### 变量

* map\_data：迷宫地图的二维数组，包含迷宫的各个元素（如障碍、起点、终点）。
* start：起点坐标，表示搜索的起始位置。
* goal：终点坐标，表示搜索的目标位置。
* height：地图的高度，计算得出，即 map\_data 的行数。
* width：地图的宽度，计算得出，即 map\_data 的列数。

#### 函数

* \_\_init\_\_(self, map\_data, start, goal)：构造函数，初始化地图数据、起点、终点，并计算地图的宽度和高度。
* get\_neighbors(self, position)：获取给定位置的可行邻居，检查上下左右四个方向的相邻节点是否在地图范围内且不是障碍，返回可行邻居的列表。
* reconstruct\_path(self, parent, current)：从当前节点 current 回溯到起点，根据 parent 字典重建路径。
* heuristic(self, position)：启发式函数，默认返回 0，用于启发式搜索（如 A\*）中被子类重写。
* search(self)：抽象方法，定义完整搜索过程的接口，需在子类中实现。
* step\_search(self)：抽象方法，用于逐步执行搜索过程，便于可视化显示，需在子类中实现。

#### 整体解读

SearchAlgorithm 是一个抽象基类，用于定义搜索算法的通用接口，提供路径搜索的基本功能。其主要作用是通过 get\_neighbors 和 reconstruct\_path 等方法，为各个子类的搜索逻辑提供基础支持。每种具体搜索算法会继承该类并实现各自的 search 和 step\_search 方法。

### DFS:

#### 变量

* stack：栈结构，用于存储待探索的节点。
* visited：集合，存储已访问的节点，避免重复访问。
* parent：字典，用于记录每个节点的前驱节点，以便路径重建。

#### 函数

* step\_search(self)：逐步执行 DFS 搜索，每次从栈中取出一个节点，将其未访问的邻居入栈，直到找到目标节点或栈为空。

#### 整体解读

DFS 类继承自 SearchAlgorithm，使用深度优先搜索算法查找迷宫中的路径。其 step\_search 方法基于栈结构实现，每次访问一个节点的所有邻居，按深度优先的顺序推进，适合于无权图的路径搜索。

### BFS：

#### 变量

* queue：队列结构，用于存储待探索的节点。
* visited：集合，存储已访问的节点。
* parent：字典，用于记录每个节点的前驱节点，以便路径重建。

#### 函数

* step\_search(self)：逐步执行 BFS 搜索，每次从队列中取出一个节点，将其未访问的邻居入队，直到找到目标节点或队列为空。

#### 整体解读

BFS 类继承自 SearchAlgorithm，使用广度优先搜索算法查找迷宫中的路径。BFS 基于队列结构，按层次逐层探索节点，确保找到的路径为最短路径，适合无权图的最短路径搜索。

### UniformCostSearch：

#### 变量

* heap：优先队列，按代价排序存储待探索的节点。
* visited：集合，存储已访问的节点。
* parent：字典，用于记录每个节点的前驱节点，以便路径重建。
* cost\_so\_far：字典，记录从起点到各节点的当前最小代价。

#### 函数

* step\_search(self)：逐步执行统一代价搜索，每次从优先队列中取出代价最小的节点，更新其邻居的代价，若找到目标节点则返回路径。

#### 整体解读

UniformCostSearch 类继承自 SearchAlgorithm，使用统一代价搜索查找迷宫路径，适合加权图。通过优先队列实现贪婪选择当前代价最小的节点，从而找到代价最低的路径。

### GreedySearch：

#### 变量

* heap：优先队列，按启发式代价排序存储待探索的节点。
* visited：集合，存储已访问的节点。
* parent：字典，用于记录每个节点的前驱节点，以便路径重建。

#### 函数

* heuristic(self, position)：使用曼哈顿距离作为启发式函数，估算当前位置到目标的距离。
* step\_search(self)：逐步执行贪心搜索，每次从优先队列中取出启发式代价最小的节点，继续扩展，直到找到目标节点。

#### 整体解读

GreedySearch 类继承自 SearchAlgorithm，实现了基于启发式的贪心搜索。通过估算到目标的距离来指导搜索，优先选择离目标最近的节点，可能会导致次优解。

### AStarSearch：

#### 变量

* heap：优先队列，按实际代价加启发式代价排序存储待探索的节点。
* visited：集合，存储已访问的节点。
* parent：字典，用于记录每个节点的前驱节点，以便路径重建。
* cost\_so\_far：字典，记录从起点到各节点的当前最小代价。

#### 函数

* heuristic(self, position)：使用曼哈顿距离作为启发式函数，估算当前位置到目标的距离。
* step\_search(self)：逐步执行 A\* 搜索，每次从优先队列中取出优先级最高的节点（综合考虑实际代价和启发式代价），继续扩展，直到找到目标节点。

#### 整体解读

AStarSearch 类继承自 SearchAlgorithm，实现了 A\* 搜索算法，结合路径的实际代价和启发式代价，从而在保证最优性的同时提升搜索效率，适用于加权图的最优路径搜索问题。

### 源代码：

# Search.py  
  
from collections import deque  
import heapq  
  
# 定义地图符号  
WALL = '1'  
ROAD = '0'  
START = '@'  
GOAL = '$'  
  
class SearchAlgorithm:  
 *"""基类，定义搜索算法的接口"""* def \_\_init\_\_(self, map\_data, start, goal):  
 self.map\_data = map\_data  
 self.start = start  
 self.goal = goal  
 self.height = len(map\_data)  
 self.width = len(map\_data[0]) if self.height > 0 else 0  
  
 def get\_neighbors(self, position):  
 *"""获取当前位置的可行邻居"""* i, j = position  
 neighbors = []  
 directions = [(-1,0), (1,0), (0,-1), (0,1)] # 上下左右  
  
 for di, dj in directions:  
 ni, nj = i + di, j + dj  
 if 0 <= ni < self.height and 0 <= nj < self.width:  
 if self.map\_data[ni][nj] in [ROAD, GOAL, START]:  
 neighbors.append((ni, nj))  
 return neighbors  
  
 def reconstruct\_path(self, parent, current):  
 *"""重建路径从起点到终点"""* path = []  
 while current != self.start:  
 path.append(current)  
 current = parent.get(current)  
 if current is None:  
 return [] # 无法到达  
 path.append(self.start)  
 path.reverse()  
 return path  
  
 def heuristic(self, position):  
 *"""启发式函数，默认不使用"""* return 0  
  
 def search(self):  
 *"""执行搜索，需在子类中实现"""* raise NotImplementedError("必须在子类中实现此方法")  
  
 def step\_search(self):  
 *"""执行逐步搜索，需在子类中实现"""* raise NotImplementedError("必须在子类中实现此方法")  
class DFS(SearchAlgorithm):  
 *"""深度优先搜索算法"""* def step\_search(self):  
 stack = []  
 visited = set()  
 parent = {}  
  
 stack.append(self.start)  
 visited.add(self.start)  
  
 while stack:  
 current = stack.pop()  
 yield current # 返回当前节点以进行可视化  
  
 if current == self.goal:  
 path = self.reconstruct\_path(parent, current)  
 yield path # 返回最终路径  
 return  
 for neighbor in self.get\_neighbors(current):  
 if neighbor not in visited:  
 stack.append(neighbor)  
 visited.add(neighbor)  
 parent[neighbor] = current  
  
 yield [] # 没有找到路径  
  
  
  
class BFS(SearchAlgorithm):  
 *"""广度优先搜索算法"""* def step\_search(self):  
 queue = deque()  
 visited = set()  
 parent = {}  
  
 queue.append(self.start)  
 visited.add(self.start)  
  
 while queue:  
 current = queue.popleft()  
 yield current # 返回当前节点以进行可视化  
  
 if current == self.goal:  
 path = self.reconstruct\_path(parent, current)  
 yield path # 返回最终路径  
 return  
 for neighbor in self.get\_neighbors(current):  
 if neighbor not in visited:  
 queue.append(neighbor)  
 visited.add(neighbor)  
 parent[neighbor] = current  
  
 yield [] # 没有找到路径  
  
  
  
class UniformCostSearch(SearchAlgorithm):  
 *"""统一代价搜索算法"""* def step\_search(self):  
 heap = []  
 heapq.heappush(heap, (0, self.start))  
 visited = set()  
 parent = {}  
 cost\_so\_far = {self.start: 0}  
  
 while heap:  
 current\_cost, current = heapq.heappop(heap)  
 yield current # 返回当前节点以进行可视化  
  
 if current == self.goal:  
 path = self.reconstruct\_path(parent, current)  
 yield path # 返回最终路径  
 return  
 if current in visited:  
 continue  
 visited.add(current)  
 for neighbor in self.get\_neighbors(current):  
 new\_cost = current\_cost + 1 # 假设每步代价为1  
 if neighbor not in cost\_so\_far or new\_cost < cost\_so\_far[neighbor]:  
 cost\_so\_far[neighbor] = new\_cost  
 heapq.heappush(heap, (new\_cost, neighbor))  
 parent[neighbor] = current  
  
 yield [] # 没有找到路径  
  
  
  
class GreedySearch(SearchAlgorithm):  
 *"""贪心搜索算法"""* def heuristic(self, position):  
 *"""使用曼哈顿距离作为启发式函数"""* return abs(position[0] - self.goal[0]) + abs(position[1] - self.goal[1])  
  
 def step\_search(self):  
 heap = []  
 heapq.heappush(heap, (self.heuristic(self.start), self.start))  
 visited = set()  
 parent = {}  
  
 while heap:  
 \_, current = heapq.heappop(heap)  
 yield current # 返回当前节点以进行可视化  
  
 if current == self.goal:  
 path = self.reconstruct\_path(parent, current)  
 yield path # 返回最终路径  
 return  
 if current in visited:  
 continue  
 visited.add(current)  
 for neighbor in self.get\_neighbors(current):  
 if neighbor not in visited:  
 heapq.heappush(heap, (self.heuristic(neighbor), neighbor))  
 parent[neighbor] = current  
  
 yield [] # 没有找到路径  
  
  
  
class AStarSearch(SearchAlgorithm):  
 *"""A\* 搜索算法"""* def heuristic(self, position):  
 *"""使用曼哈顿距离作为启发式函数"""* return abs(position[0] - self.goal[0]) + abs(position[1] - self.goal[1])  
  
 def step\_search(self):  
 heap = []  
 heapq.heappush(heap, (self.heuristic(self.start), 0, self.start))  
 visited = set()  
 parent = {}  
 cost\_so\_far = {self.start: 0}  
  
 while heap:  
 \_, current\_cost, current = heapq.heappop(heap)  
 yield current # 返回当前节点以进行可视化  
  
 if current == self.goal:  
 path = self.reconstruct\_path(parent, current)  
 yield path # 返回最终路径  
 return  
 if current in visited:  
 continue  
 visited.add(current)  
 for neighbor in self.get\_neighbors(current):  
 new\_cost = current\_cost + 1 # 假设每步代价为1  
 if neighbor not in cost\_so\_far or new\_cost < cost\_so\_far[neighbor]:  
 cost\_so\_far[neighbor] = new\_cost  
 priority = new\_cost + self.heuristic(neighbor)  
 heapq.heappush(heap, (priority, new\_cost, neighbor))  
 parent[neighbor] = current  
  
 yield [] # 没有找到路径

## My\_Map.py:

### My\_Map:

### 变量

* **width**：地图的宽度，以单元格数量表示。
* **height**：地图的高度，以单元格数量表示。
* **cell\_size**：每个单元格的大小（像素）。
* **map\_data**：存储地图数据的二维数组。地图元素包括起点、终点、道路、障碍等。
* **canvas**：用于显示地图和路径的 Tkinter 画布组件。
* **start\_goal\_bg**：起点和终点的图像或颜色背景。
* **agent\_bg**：智能体（AGENT）的图像或颜色背景。
* **agent\_pos**：当前智能体所在的位置坐标。
* **start\_pos**：起点坐标。
* **goal\_pos**：终点坐标。
* **path**：当前搜索路径。
* **is\_animating**：布尔值，表示动画状态，用于控制动画过程中的用户交互。
* **cost\_label**：用于显示路径代价的标签组件。

### 函数

1. **\_\_init\_\_(self, width, height, cell\_size, canvas)**  
   初始化地图的宽度、高度和单元格大小，设置默认值并初始化画布组件和图像背景。
2. **generate\_random\_map(self, width=None, height=None, cell\_size=None, start\_position=None, goal\_position=None)**  
   生成一个随机地图，确保起点和终点之间有一条可通行的路径。先生成起点到终点的随机通路，再随机设置其他单元格。
3. **create\_random\_path(self, start, goal)**  
   生成一条从 start 到 goal 的通路，用于保证地图中存在起点到终点的可行路径。
4. **get\_neighbors\_for\_path(self, i, j, goal)**  
   获取生成路径时的合法邻居，返回上下左右方向的邻居节点列表，优先朝向目标节点。
5. **set\_map(self, map\_data)**  
   从指定的 map\_data 数据设置地图，初始化起点和终点位置。
6. **render\_map(self)**  
   在 canvas 画布上渲染地图。根据 map\_data 数据绘制不同颜色或图像的单元格，显示路径、障碍、起点、终点、智能体等。
7. **find\_start\_position(self)**  
   返回起点的位置，用于初始化智能体位置。
8. **set\_agent\_position(self, position)**  
   设置智能体的当前位置，并更新地图显示，避免在动画播放过程中手动移动。
9. **move\_agent(self, direction)**  
   根据方向键 w、a、s、d 控制智能体移动。如果移动至终点，则显示成功提示。
10. **start\_manual\_search(self)**  
    初始化智能体位置，开始手动搜索。找到起点并设置为当前智能体位置。
11. **bind\_keys(self, root)**  
    将方向键与 move\_agent 函数绑定，用于手动控制智能体移动。
12. **display\_path(self, path, final=False)**  
    显示路径，将路径绘制到地图上。final 参数控制是否用绿色显示最终路径。
13. **create\_cost\_label(self, root)**  
    在界面底部创建一个标签显示路径代价。
14. **display\_cost(self, cost)**  
    在路径代价标签上显示代价值。
15. **animate\_agent\_movement(self, path, delay=300)**  
    用动画方式显示智能体沿路径的移动。delay 参数控制每步之间的延迟。
16. **animate\_step(self)**  
    动画移动中的单步操作，按路径顺序逐步移动智能体。
17. **run\_search\_and\_animate(self, algorithm\_class)**  
    运行指定搜索算法并动画显示结果。初始化搜索算法后逐步显示路径搜索过程。
18. **animate\_search\_step(self)**  
    逐步动画展示搜索过程，将当前节点显示在地图上。搜索结束后显示完整路径及路径代价。
19. **run\_all\_searches(self, algorithm\_name)**  
    根据算法名称运行对应的搜索算法，并在地图上展示结果。支持的算法包括 DFS、BFS、统一代价搜索、贪心搜索和 A\* 搜索。

### 整体解读

My\_Map 类将地图生成、路径搜索和结果显示集成在一个交互式界面中。通过 generate\_random\_map 和 create\_random\_path 实现地图生成，通过 run\_search\_and\_animate 调用不同的搜索算法，结合 Tkinter 画布实时显示搜索过程，使用户可以观察智能体在迷宫中的路径搜索。

### 源代码：

import tkinter as tk  
from tkinter import Canvas, messagebox  
from PIL import Image, ImageTk # 导入Pillow库  
import random  
import time  
from Search import DFS, BFS, UniformCostSearch, GreedySearch, AStarSearch  
  
# 定义地图符号  
WALL = '1'  
ROAD = '0'  
START = '@'  
GOAL = '$'  
AGENT = 'A'  
PATH = '.' # 用于显示路径  
  
# 定义RGB颜色并转换为十六进制颜色  
def rgb\_to\_hex(rgb):  
 return "#%02x%02x%02x" % rgb  
  
# 定义颜色  
wall\_rgb = (21, 61, 100)  
wall\_bg = rgb\_to\_hex(wall\_rgb)  
  
road\_rgb = (255, 255, 255)  
road\_bg = rgb\_to\_hex(road\_rgb)  
  
path\_rgb = (173, 216, 230) # 浅蓝色用于路径  
path\_bg = rgb\_to\_hex(path\_rgb)  
  
final\_path\_rgb = (50, 205, 50) # 绿色用于最终路径  
final\_path\_bg = rgb\_to\_hex(final\_path\_rgb)  
  
class My\_Map:  
 def \_\_init\_\_(self, width, height, cell\_size, canvas):  
 *"""初始化地图的宽度、高度和每个单元格的大小"""* self.width = width  
 self.height = height  
 self.cell\_size = cell\_size  
 self.map\_data = [] # 用于存储地图的二维列表  
 self.canvas = canvas # 画布引用  
  
 # 调整图片大小并加载  
 try:  
 start\_goal\_image = Image.open("E:\Python\_Project\IntroToAI\_exp\src\image\Start\_Goal.png")  
 agent\_image = Image.open("E:\Python\_Project\IntroToAI\_exp\src\image\Agent.png")  
 except FileNotFoundError:  
 # 使用简单的颜色块代替图片  
 start\_goal\_image = Image.new("RGB", (self.cell\_size - 1, self.cell\_size - 1), (255, 0, 0))  
 agent\_image = Image.new("RGB", (self.cell\_size - 1, self.cell\_size - 1), (0, 255, 0))  
  
 # 调整为与单元格相匹配的大小  
 self.start\_goal\_bg = ImageTk.PhotoImage(start\_goal\_image.resize((self.cell\_size-1, self.cell\_size-1)))  
 self.agent\_bg = ImageTk.PhotoImage(agent\_image.resize((self.cell\_size-1, self.cell\_size-1)))  
  
 # AGENT 的初始位置  
 self.agent\_pos = None  
  
 # 记录 START 和 GOAL 的位置  
 self.start\_pos = None  
 self.goal\_pos = None  
  
 # 路径列表  
 self.path = []  
  
 # 动画控制  
 self.is\_animating = False  
  
 def generate\_random\_map(self, width=None, height=None, cell\_size=None, start\_position=None, goal\_position=None):  
 *"""生成随机地图，确保起点和终点之间有通路，先生成路径，再随机设置其他方块"""* if width:  
 self.width = width  
 if height:  
 self.height = height  
 if cell\_size:  
 self.cell\_size = cell\_size  
  
 # 初始化空地图  
 self.map\_data = [[WALL for \_ in range(self.width)] for \_ in range(self.height)]  
  
 # 设置起点和终点  
 if start\_position:  
 start\_x, start\_y = start\_position  
 self.start\_pos = (start\_x, start\_y)  
 else:  
 self.start\_pos = (1, 1)  
  
 if goal\_position:  
 goal\_x, goal\_y = goal\_position  
 if 1 <= goal\_x < self.height - 1 and 1 <= goal\_y < self.width - 1:  
 self.goal\_pos = (goal\_x, goal\_y)  
 else:  
 raise ValueError("终点位置超出地图范围")  
 else:  
 self.goal\_pos = (self.height - 2, self.width - 2)  
  
 # 确保起点和终点设置正确  
 self.map\_data[self.start\_pos[0]][self.start\_pos[1]] = START  
 self.map\_data[self.goal\_pos[0]][self.goal\_pos[1]] = GOAL  
  
 # 随机生成一条从起点到终点的通路  
 self.create\_random\_path(self.start\_pos, self.goal\_pos)  
  
 # 随机设置剩余的格子  
 for i in range(self.height):  
 for j in range(self.width):  
 if self.map\_data[i][j] == ROAD:  
 continue # 保留生成路径时设置的道路  
 elif self.map\_data[i][j] not in [START, GOAL]:  
 self.map\_data[i][j] = random.choice([WALL, ROAD])  
  
 def create\_random\_path(self, start, goal):  
 *"""在起点和终点之间生成一条随机路径"""* current = start  
 path = [current]  
  
 while current != goal:  
 i, j = current  
 neighbors = self.get\_neighbors\_for\_path(i, j, goal)  
  
 if not neighbors:  
 raise ValueError("无法生成路径，请检查地图大小或起点终点位置")  
  
 next\_step = random.choice(neighbors)  
 path.append(next\_step)  
 current = next\_step  
  
 # 将路径上的格子设置为道路  
 for i, j in path:  
 if self.map\_data[i][j] not in [START, GOAL]:  
 self.map\_data[i][j] = ROAD  
  
 def get\_neighbors\_for\_path(self, i, j, goal):  
 *"""获取用于生成路径的合法邻居（上下左右，并朝向终点优先）"""* directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # 上下左右  
 neighbors = []  
  
 for di, dj in directions:  
 ni, nj = i + di, j + dj  
 if 0 <= ni < self.height and 0 <= nj < self.width:  
 if self.map\_data[ni][nj] == WALL or (ni, nj) == goal:  
 neighbors.append((ni, nj))  
  
 return neighbors  
  
 def set\_map(self, map\_data):  
 *"""根据指定的字符串地图数据来设置地图"""* self.map\_data = [row.split(',') for row in map\_data]  
  
 # 记录 START 和 GOAL 的位置  
 for i, row in enumerate(self.map\_data):  
 for j, cell in enumerate(row):  
 if cell == START:  
 self.start\_pos = (i, j)  
 elif cell == GOAL:  
 self.goal\_pos = (i, j)  
  
 def render\_map(self):  
 *"""在 Tkinter 画布中渲染地图"""* self.canvas.delete("all") # 清除之前的绘制  
  
 # 根据 map\_data 渲染地图  
 for i, row in enumerate(self.map\_data):  
 for j, cell in enumerate(row):  
 x1 = j \* self.cell\_size  
 y1 = i \* self.cell\_size  
 x2 = x1 + self.cell\_size  
 y2 = y1 + self.cell\_size  
  
 if cell == WALL:  
 color = wall\_bg  
 self.canvas.create\_rectangle(x1, y1, x2, y2, fill=color, outline="black")  
 elif cell == ROAD:  
 color = road\_bg  
 self.canvas.create\_rectangle(x1, y1, x2, y2, fill=color, outline="black")  
 elif cell == PATH:  
 color = path\_bg  
 self.canvas.create\_rectangle(x1, y1, x2, y2, fill=color, outline="black")  
 elif cell == START or cell == GOAL:  
 self.canvas.create\_image(x1 + self.cell\_size // 2, y1 + self.cell\_size // 2,  
 image=self.start\_goal\_bg)  
 elif cell == AGENT:  
 self.canvas.create\_image(x1 + self.cell\_size // 2, y1 + self.cell\_size // 2, image=self.agent\_bg)  
 elif cell == 'F':  
 # 渲染最终路径  
 color = final\_path\_bg  
 self.canvas.create\_rectangle(x1, y1, x2, y2, fill=color, outline="black")  
  
 def find\_start\_position(self):  
 *"""找到起点的位置"""* return self.start\_pos  
  
 def set\_agent\_position(self, position):  
 *"""设置 AGENT 的位置"""* if self.is\_animating:  
 return # 在动画过程中禁止手动移动  
  
 if self.agent\_pos:  
 # 将之前的位置恢复为原始符号（ROAD, START, GOAL）  
 prev\_i, prev\_j = self.agent\_pos  
 if (prev\_i, prev\_j) == self.start\_pos:  
 self.map\_data[prev\_i][prev\_j] = START  
 elif (prev\_i, prev\_j) == self.goal\_pos:  
 self.map\_data[prev\_i][prev\_j] = GOAL  
 else:  
 self.map\_data[prev\_i][prev\_j] = ROAD  
  
 i, j = position  
  
 # 将新的位置设置为 AGENT  
 if self.map\_data[i][j] == START or self.map\_data[i][j] == GOAL:  
 # 如果新的位置是 START 或 GOAL，依然显示 AGENT  
 self.map\_data[i][j] = AGENT  
 else:  
 self.map\_data[i][j] = AGENT  
  
 self.agent\_pos = position  
 self.render\_map()  
  
 def move\_agent(self, direction):  
 *"""根据方向移动 AGENT"""* if not self.agent\_pos:  
 messagebox.showwarning("警告", "请先点击“手动搜索”按钮开始移动。")  
 return  
  
 if self.is\_animating:  
 return # 在动画过程中禁止手动移动  
  
 i, j = self.agent\_pos  
 if direction == 'w': # 上  
 new\_i, new\_j = i - 1, j  
 elif direction == 'a': # 左  
 new\_i, new\_j = i, j - 1  
 elif direction == 's': # 下  
 new\_i, new\_j = i + 1, j  
 elif direction == 'd': # 右  
 new\_i, new\_j = i, j + 1  
 else:  
 return  
  
 # 检查边界  
 if new\_i < 0 or new\_i >= self.height or new\_j < 0 or new\_j >= self.width:  
 messagebox.showinfo("提示", "无法移动到地图外。")  
 return  
  
 target\_cell = self.map\_data[new\_i][new\_j]  
 if target\_cell == WALL:  
 messagebox.showinfo("提示", "无法移动到墙壁！")  
 elif target\_cell in [ROAD, GOAL, START]:  
 self.set\_agent\_position((new\_i, new\_j))  
 if (new\_i, new\_j) == self.goal\_pos:  
 messagebox.showinfo("完成", "AGENT 已到达终点！")  
 else:  
 # 其他情况，如目标是 AGENT（不太可能）  
 pass  
  
 def start\_manual\_search(self):  
 *"""初始化 AGENT 并绑定键盘事件"""* if self.is\_animating:  
 return # 在动画过程中禁止手动移动  
  
 start\_pos = self.find\_start\_position()  
 if not start\_pos:  
 messagebox.showerror("错误", "找不到起点！")  
 return  
 self.set\_agent\_position(start\_pos)  
  
 def bind\_keys(self, root):  
 *"""绑定键盘事件"""* root.bind('<w>', lambda event: self.move\_agent('w'))  
 root.bind('<a>', lambda event: self.move\_agent('a'))  
 root.bind('<s>', lambda event: self.move\_agent('s'))  
 root.bind('<d>', lambda event: self.move\_agent('d'))  
  
 def display\_path(self, path, final=False):  
 *"""在地图上显示路径，final参数用于区分是否为最终路径"""* for pos in path:  
 if pos != self.start\_pos and pos != self.goal\_pos:  
 i, j = pos  
 if final:  
 # 最终路径用绿色显示  
 self.map\_data[i][j] = 'F' # 标记为最终路径  
 else:  
 # 探索路径用浅蓝色显示  
 self.map\_data[i][j] = PATH  
 self.render\_map()  
  
 def create\_cost\_label(self, root):  
 *"""在窗口上创建一个用于显示路径代价的Label"""* self.cost\_label = tk.Label(root, text="路径代价: 未知", font=("Helvetica", 14), fg="black")  
 self.cost\_label.pack(side=tk.BOTTOM)  
  
 def display\_cost(self, cost):  
 *"""在Label上显示路径代价"""* self.cost\_label.config(text=f"路径代价: {cost}")  
  
 def animate\_agent\_movement(self, path, delay=300):  
 *"""动画展示 AGENT 沿路径移动"""* if not path:  
 messagebox.showinfo("提示", "没有找到从起点到终点的路径。")  
 return  
  
 self.is\_animating = True  
 self.path = path  
 self.current\_step = 0  
 self.animation\_delay = delay  
 self.total\_steps = len(self.path)  
 self.animate\_step()  
  
 def animate\_step(self):  
 *"""执行动画的每一步"""* if self.current\_step >= self.total\_steps:  
 self.is\_animating = False  
 messagebox.showinfo("完成", "AGENT 已到达终点！")  
 return  
  
 position = self.path[self.current\_step]  
 self.set\_agent\_position(position)  
 self.current\_step += 1  
  
 # 调用下一步动画  
 self.canvas.after(self.animation\_delay, self.animate\_step())  
  
 def run\_search\_and\_animate(self, algorithm\_class):  
 *"""运行指定的搜索算法并动画展示路径"""* if self.is\_animating:  
 return # 防止重复点击  
  
 # 清除之前的路径和代理位置  
 for i, row in enumerate(self.map\_data):  
 for j, cell in enumerate(row):  
 if cell in [PATH, AGENT, 'F']: # 清除之前标记的路径和最终路径  
 self.map\_data[i][j] = ROAD  
 elif cell == 'F': # 清除最终路径标记为道路  
 self.map\_data[i][j] = ROAD  
 elif cell == AGENT: # 清除AGENT位置  
 self.map\_data[i][j] = ROAD  
  
 self.render\_map() # 重绘地图，确保之前的路径已清除  
  
 # 初始化搜索算法  
 search\_algo = algorithm\_class(self.map\_data, self.start\_pos, self.goal\_pos)  
 self.search\_generator = search\_algo.step\_search()  
 self.is\_animating = True  
 self.animate\_search\_step()  
  
 def animate\_search\_step(self):  
 *"""逐步动画展示搜索过程"""* try:  
 result = next(self.search\_generator)  
 if isinstance(result, list):  
 # 搜索完成，显示最终路径  
 self.is\_animating = False  
 if result:  
 self.display\_path(result, final=True) # 用绿色显示最终路径  
 self.animate\_agent\_movement(result)  
  
 # 计算路径代价并显示  
 path\_cost = len(result) - 1 # 路径代价为路径长度（步数），减去起点  
 self.display\_cost(path\_cost)  
  
 else:  
 messagebox.showinfo("提示", "没有找到从起点到终点的路径。")  
 else:  
 # 更新地图，显示当前探索的节点  
 i, j = result  
  
 # 处理之前的AGENT位置  
 if self.agent\_pos: # 将之前的AGENT位置标记为PATH或START/GOAL  
 prev\_i, prev\_j = self.agent\_pos  
 if (prev\_i, prev\_j) == self.start\_pos:  
 self.map\_data[prev\_i][prev\_j] = START  
 elif (prev\_i, prev\_j) == self.goal\_pos:  
 self.map\_data[prev\_i][prev\_j] = GOAL  
 else:  
 self.map\_data[prev\_i][prev\_j] = PATH  
  
 # 将当前节点标记为AGENT  
 self.map\_data[i][j] = AGENT  
 self.agent\_pos = (i, j) # 更新AGENT位置  
 self.render\_map()  
  
 # 设置一个延迟，然后继续下一步  
 self.canvas.after(10, self.animate\_search\_step)  
 except StopIteration:  
 self.is\_animating = False  
 messagebox.showinfo("提示", "搜索结束，没有找到路径。")  
  
 def run\_all\_searches(self, algorithm\_name):  
 *"""根据算法名称运行对应的搜索算法"""* algorithm\_map = {  
 "DFS": DFS,  
 "BFS": BFS,  
 "Uniform Cost Search": UniformCostSearch,  
 "Greedy Search": GreedySearch,  
 "A\* Search": AStarSearch  
 }  
  
 algorithm\_class = algorithm\_map.get(algorithm\_name)  
 if not algorithm\_class:  
 messagebox.showerror("错误", f"未知的搜索算法：{algorithm\_name}")  
 return  
  
 self.run\_search\_and\_animate(algorithm\_class)

## problem1.py:

### 解读：

### 代码结构与核心功能

1. **open\_selection\_window()**  
   打开模式选择窗口，用户可选择“随机生成地图”或“使用预定义地图”。
2. **set\_mode(mode)**  
   根据用户选择的模式初始化地图：
   * “random” 模式调用 enter\_random\_map\_mode()，生成随机地图。
   * “predefined” 模式调用 enter\_predefined\_map\_mode()，加载预定义的静态地图。
3. **enter\_random\_map\_mode()**  
   进入随机地图模式，提示用户输入地图宽度、高度、单元格大小，以及起点和终点的坐标。生成地图后，调用 generate\_random\_map() 来创建随机地图。
4. **generate\_random\_map()**  
   生成随机地图并在画布上渲染。地图刷新后，会弹出一个提示框告知用户刷新完成。
5. **enter\_predefined\_map\_mode()**  
   进入预定义地图模式，加载一个静态地图，并自动调整画布大小以适配地图。生成地图后在画布上渲染。
6. **display\_map()**  
   显示地图并在按钮框架中提供不同的搜索算法按钮，允许用户选择不同的路径搜索算法。

### 源代码：

import tkinter as tk  
from tkinter import Canvas, simpledialog, messagebox  
from My\_Map import My\_Map  
  
def open\_selection\_window():  
 *"""选择地图模式的窗口"""* selection\_window = tk.Toplevel(root)  
 selection\_window.title("选择地图模式")  
 tk.Label(selection\_window, text="请选择地图模式：").pack(pady=10)  
  
 random\_button = tk.Button(  
 selection\_window, text="随机生成地图", command=lambda: [set\_mode("random"), selection\_window.destroy()]  
 )  
 predefined\_button = tk.Button(  
 selection\_window, text="使用预定义地图", command=lambda: [set\_mode("predefined"), selection\_window.destroy()]  
 )  
  
 random\_button.pack(pady=5)  
 predefined\_button.pack(pady=5)  
  
def set\_mode(mode):  
 *"""根据用户选择的模式初始化地图"""* global map\_mode  
 map\_mode = mode  
 if map\_mode == "random":  
 enter\_random\_map\_mode()  
 else:  
 enter\_predefined\_map\_mode()  
  
def enter\_random\_map\_mode():  
 *"""进入随机地图模式，用户输入参数生成地图"""* global map\_width, map\_height, cell\_size, start\_x, start\_y, goal\_x, goal\_y  
  
 # 用户输入地图参数  
 map\_width = simpledialog.askinteger("输入", "请输入地图宽度：", minvalue=5, maxvalue=50)  
 map\_height = simpledialog.askinteger("输入", "请输入地图高度：", minvalue=5, maxvalue=50)  
 cell\_size = simpledialog.askinteger("输入", "请输入单元格大小（像素）：", minvalue=10, maxvalue=100)  
 start\_x = simpledialog.askinteger("输入", "请输入起点的X坐标：", minvalue=1, maxvalue=map\_width - 2)  
 start\_y = simpledialog.askinteger("输入", "请输入起点的Y坐标：", minvalue=1, maxvalue=map\_height - 2)  
 goal\_x = simpledialog.askinteger("输入", "请输入终点的X坐标：", minvalue=1, maxvalue=map\_width - 2)  
 goal\_y = simpledialog.askinteger("输入", "请输入终点的Y坐标：", minvalue=1, maxvalue=map\_height - 2)  
  
 # 调整画布大小  
 canvas.config(width=map\_width \* cell\_size, height=map\_height \* cell\_size)  
  
 # 初始化 My\_Map 对象  
 global my\_map  
 my\_map = My\_Map(width=map\_width, height=map\_height, cell\_size=cell\_size, canvas=canvas)  
  
 # 创建路径代价标签  
 my\_map.create\_cost\_label(root)  
  
 # 生成随机地图  
 generate\_random\_map()  
  
 # 显示地图和控制按钮  
 display\_map()  
  
def generate\_random\_map():  
 *"""生成随机地图，保留之前输入的数据"""* my\_map.generate\_random\_map(  
 width=map\_width,  
 height=map\_height,  
 cell\_size=cell\_size,  
 start\_position=(start\_x, start\_y),  
 goal\_position=(goal\_x, goal\_y)  
 )  
 my\_map.render\_map() # 实时更新地图  
 messagebox.showinfo("地图已刷新", "随机地图已成功刷新！") # 刷新后显示提示  
  
def enter\_predefined\_map\_mode():  
 *"""进入预定义地图模式，直接显示固定地图"""* MAP = [  
 "1,@,1,1,1,1,1,1",  
 "1,0,0,0,0,0,1,1",  
 "1,0,1,0,1,0,0,1",  
 "1,0,1,0,0,1,0,1",  
 "1,0,1,1,0,0,0,1",  
 "1,0,0,1,0,1,0,1",  
 "1,1,0,0,0,0,0,1",  
 "1,1,1,1,1,1,$,1",  
 ]  
 map\_width = len(MAP[0].split(','))  
 map\_height = len(MAP)  
 cell\_size = 50  
  
 # 调整画布大小  
 canvas.config(width=map\_width \* cell\_size, height=map\_height \* cell\_size)  
  
 # 初始化 My\_Map 对象  
 global my\_map  
 my\_map = My\_Map(width=map\_width, height=map\_height, cell\_size=cell\_size, canvas=canvas)  
  
 # 创建路径代价标签  
 my\_map.create\_cost\_label(root)  
  
 # 设置预定义地图  
 my\_map.set\_map(MAP)  
  
 # 显示地图和控制按钮  
 display\_map()  
  
def display\_map():  
 *"""显示地图和相关搜索按钮"""* my\_map.render\_map()  
 canvas.pack() # 现在显示画布  
 button\_frame.pack(pady=10) # 现在显示按钮框架  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # 创建 Tkinter 主窗口  
 root = tk.Tk()  
 root.title("地图搜索可视化")  
  
 # 创建画布，初始不显示  
 canvas = Canvas(root)  
  
 # 打开地图模式选择窗口  
 open\_selection\_window()  
  
 # 创建按钮框架，但初始不显示  
 button\_frame = tk.Frame(root)  
  
 # 创建刷新按钮（仅在随机地图模式下显示）  
 refresh\_button = tk.Button(  
 button\_frame,  
 text="刷新随机地图",  
 command=generate\_random\_map  
 )  
 refresh\_button.grid(row=0, column=0, padx=5)  
  
 # 创建搜索按钮  
 manual\_search\_button = tk.Button(  
 button\_frame,  
 text="手动搜索",  
 command=lambda: [my\_map.start\_manual\_search(), my\_map.bind\_keys(root)]  
 )  
 manual\_search\_button.grid(row=0, column=1, padx=5)  
  
 dfs\_button = tk.Button(  
 button\_frame,  
 text="DFS",  
 command=lambda: my\_map.run\_all\_searches("DFS")  
 )  
 dfs\_button.grid(row=0, column=2, padx=5)  
  
 bfs\_button = tk.Button(  
 button\_frame,  
 text="BFS",  
 command=lambda: my\_map.run\_all\_searches("BFS")  
 )  
 bfs\_button.grid(row=0, column=3, padx=5)  
  
 ucs\_button = tk.Button(  
 button\_frame,  
 text="统一代价搜索",  
 command=lambda: my\_map.run\_all\_searches("Uniform Cost Search")  
 )  
 ucs\_button.grid(row=0, column=4, padx=5)  
  
 greedy\_button = tk.Button(  
 button\_frame,  
 text="贪心搜索",  
 command=lambda: my\_map.run\_all\_searches("Greedy Search")  
 )  
 greedy\_button.grid(row=0, column=5, padx=5)  
  
 astar\_button = tk.Button(  
 button\_frame,  
 text="A\* 搜索",  
 command=lambda: my\_map.run\_all\_searches("A\* Search")  
 )  
 astar\_button.grid(row=0, column=6, padx=5)  
  
 root.mainloop()