同济大学计算机系

人工智能原理与技术综合实验报告

学　　号 2352595

姓　　名 张嘉麟

专　　业 计算机科学与技术

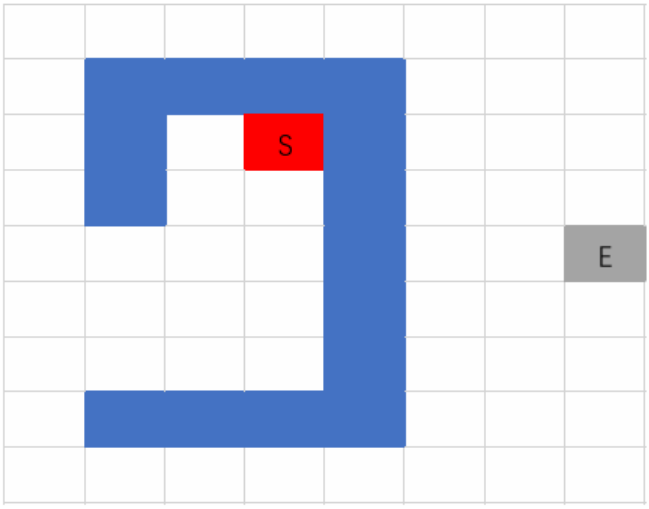
授课老师 苗夺谦，张红云

目录

一、问题概述

1.1 问题的工程化描述

在本实验中，我们给定如下的机器人避障寻径场景图：



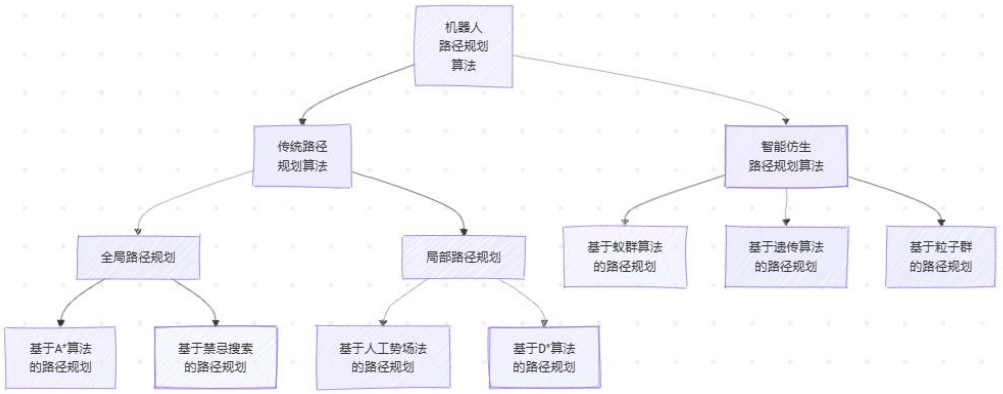
其中 S 为机器人寻径的起点，E 为寻径的终点，蓝色区域为无法通过的障碍。要求机器人从 S 点出发，绕开障碍，到达 E 点。本实验需要参与者自行设计核心算法规则完成机器人从起点到终点的最短路径搜索。

本项目的任务是让机器人在带有障碍物的网格环境中找到最短的路径。网格由多个单元格组成，其中一些单元格为障碍物，机器人需要避免碰撞这些障碍物，找到从起点（S）到终点（E）的最短路径。

1.2 解决问题的思路和想法

使用传统方法如 A*算法、Dijkstra 算法完成最短路径规划。使用蚁群算法和粒子群算法实现智能仿生的路径规划算法。使用强化学习训练的方法对于有动态障碍物的场景进行路径规划。

A*算法、Dijkstra 算法对于较小的静态地图有很不错的运行表现，但当地图变得很大或很复杂的时候时间复杂度、空间复杂度会大大增大。蚁群算法和粒子群算法学习了群体整体化的寻路机制，有更好的整体性，在一定程度上可以避免局部最优，但同时却不保证可以找到最优路径。强化学习可以适应变化的地图，对于动态地图有很好的实验表现，同时鲁棒性也较好。



二、算法设计

2.1 传统最短路径规划算法

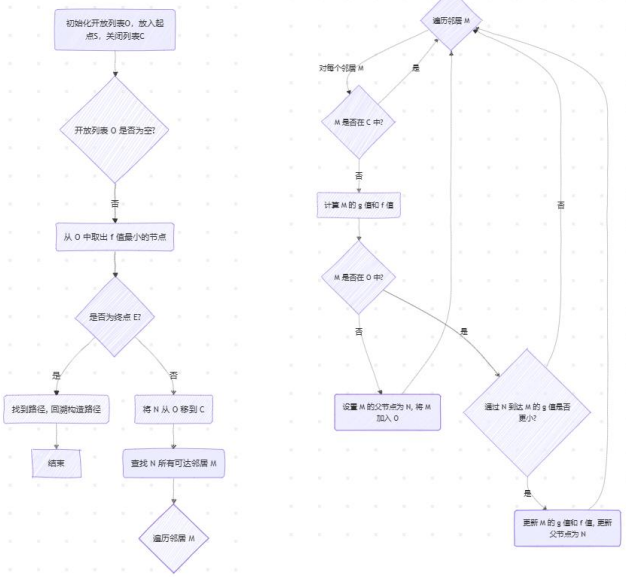
2.1.1 BFS 函数实现广度优先搜索

BFS 是一种基础的图遍历算法。通过逐层扩展搜索范围来探索所有可达节点，确保在找到目标时所经过的层数即为最少的步数。

为了实现这种逐层搜索，需要借助队列。初始时将起点放入队列，当队列不为空时，取出队首节点，访问其所有未被访问过的邻居节点，将这些邻居标记为已访问。通过队列先进先出的特性，保证了节点按距离起点的远近顺序被访问，从而在第一次遇到目标节点时，所回溯的路径即为步数最短的路径。

2.1.2 A*算法

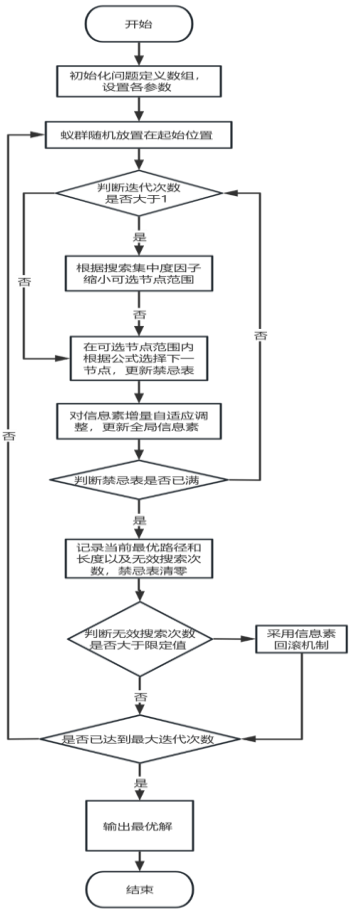
A*算法是一种启发式搜索算法。实际已行走成本和预估到目标成本来评估节点的优先级，能够高效地引导搜索方向，避免不必要的探索，效率方面在各寻路算法中一般是最快的。



设计核心在于其评估函数 $f(n)=g(n)+h(n)$。其中 g 是实际累计成本，h 是预估成本。每次迭代 A*从列表中取出 $f(n)$值最小的节点进行扩展，计算其邻居节点的 g 值和 f 值。利用离散数学的知识更新各节点的权。

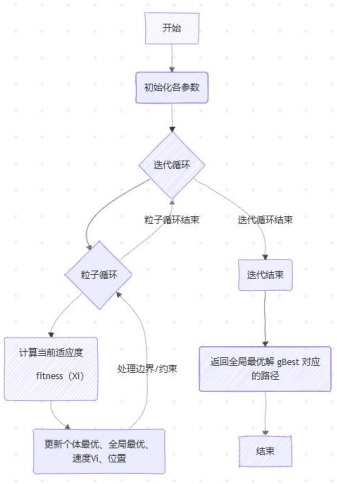
2.2 基于生物智能的寻路算法

2.2.1 蚁群算法



ACO 是一种模拟蚂蚁找食物的启发式优化算法。通过模拟蚂蚁释放和感知信息素的机制，让“蚂蚁群”逐渐发现最优路径，在复杂场景中鲁棒性不错。

ACO 的核心思路是利用信息素作为间接通信媒介来指导搜索。算法模拟一群蚂蚁从起点出发寻找终点。每只蚂蚁在选择下一步路径时，会受到信息素浓度（信息素越浓被选择的概率越大）和启发式信息（距离目标更近的节点更优先）影响。蚂蚁走后，会在路径上留下信息素。同时，信息素会随时间推移而缓慢蒸发。这样好路径被加强，信息素蒸发避免过早收敛，使得逐渐找到最短路径。



2.2.2 粒子群算法

PSO 是一种模拟鸟群的智能优化算法。可以将一个潜在的路径表示为一个“粒子”，通过模拟粒子群在解空间中的飞行和信息共享，迭代地改进路径质量，最终找到路径。简单的理解就是，每一个粒子都是一只鸟，通过每个个体的找路，集体就会趋向于最短路径。

PSO 算法的核心思想是群体协作和经验共享。在每次迭代中，粒子会根据三个因素更新其速度和位置：自身的惯性（保持当前运动状态）、自身历史最佳位置（pBest）、以及整个群体发现的最佳位置（gBest）。通过这种结合个体经验和群体智慧的机制，粒子群能够有效地探索解空间，并逐渐向最优解区域收敛。

2.3 强化学习（RL）在动态路障中的寻路算法

强化学习本项目中主要用的是 Q-learning 模型。核心是奖励和惩罚机制，在特定地图里面不断训练的过程中能够让寻路智能体学会有效避障并到达终点的路径，它不需要预先知道障碍物的确切运动模式，而是通过学习适应环境的变化。

三、算法实现

3.1 传统最短路径规划算法

3.1.1 BFS 函数实现广度优先搜索

广度优先搜索是逐层搜索的，可以确保找到最短路径。

```
def bfs(self):
        start_time = time.time()
        directions = [(-1,0), (1,0), (0,-1), (0,1)]
        queue = deque()
        visited = [[False]*self.cols for _ in range(self.rows)]
        parent = [[None]*self.cols for _ in range(self.rows)]
        queue.append(self.start)
        visited[self.start[0]][self.start[1]] = True
        while queue:
            current = queue.popleft()
            if current == self.end:
                break

            for dx, dy in directions:
                x = current[0] + dx
                y = current[1] + dy
                if 0 <= x < self.rows and 0 <= y < self.cols \
                    and not visited[x][y] and self.matrix[x][y] != 1:
                    visited[x][y] = True
                    parent[x][y] = current
                    queue.append((x, y))
        execution_time = time.time() - start_time
        if current == self.end:
            return self.reconstruct_path(parent), execution_time
        return None, execution_time
```

3.1.2 A*算法

A*算法是一种启发式的搜索算法，结合了实际代价和估计代价。其中估计代价中的启发函数使用的是曼哈顿距离。

```
g(n)：从起点到当前节点的实际代价
h(n)：从当前节点到终点的估计代价（启发函数）
f(n) = g(n) + h(n)：总估计代价
```

```
def astar(self):
        start_time = time.time()
        open_set = []
        closed_set = set()

        start_node = self.Node(self.start)
        end_node = self.Node(self.end)
        heapq.heappush(open_set, (0, start_node))

        while open_set:
            current = heapq.heappop(open_set)[1]
            if current.pos == self.end:
                execution_time = time.time() - start_time
                return self._reconstruct_astar_path(current),
execution_time

            closed_set.add(current.pos)

            for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:
                x = current.pos[0] + dx
                y = current.pos[1] + dy
                if 0 <= x < self.rows and 0 <= y < self.cols and
self.matrix[x][y] != 1:
                    neighbor = self.Node((x, y), current)
                    if neighbor.pos in closed_set:
                        continue

                    neighbor.g = current.g + 1
                    neighbor.h = self._heuristic(neighbor.pos, self.end)

                    if not any(n[1].pos == neighbor.pos for n in open_set):
                        heapq.heappush(open_set, (neighbor.g + neighbor.h,
neighbor))
        return None, time.time() - start_time
```

3.1.3 随机生成地图函数

分为两个函数，一个是小规模的随机生成，直接在面板生成并提供可视化。另一个是大规模的 map 生成，调用了 pygame 库实现可视化。

```
def generate_random_map(self):
    # 随机生成小规模地图(5-15 x 5-15)
    # 1. 生成 5 个候选地图
    # 2. 控制障碍物密度(20%)
```

```
    # 3. 确保起点到终点有可行路径
    # 4. 提供可视化预览

def generate_large_map(self):
     # 生成大规模地图(16-100 x 16-100)
     # 1. 使用分块生成策略
     # 2. pygame 实时可视化
     # 3. 支持重新生成(R 键)和确认(空格键)
```

3.1.4 可视化相关函数

主要分为两个部分：终端路径可视化打印和 pygame 的路径可视化显示。其中调用 pygame 时分成了两个函数，分别是构建背景（即打印之前生成的地图）和路径显示。

```
    def visualize(self, path, algorithm):
        # 终端中的路径可视化
        # 使用 ASCII 字符显示：
        # S-起点，E-终点，*-路径，0/1-空地/障碍
    def draw_map(self):
        # pygame 中绘制地图的基础函数
        # 绘制：空地(白)、障碍(黑)、起点(绿)、终点(红)
    def visualize_path_pygame(self, path):
        # pygame 中的路径可视化
        # 1. 绘制基础地图
        # 2. 在路径上绘制蓝色方块(不包括起终点)
        # 3. 支持 ESC 键退出
```

3.2 其他寻路算法

3.2.1 蚁群算法

3.2.1.1 基本原理

蚁群算法(Ant Colony Optimization, ACO)是一种模拟蚂蚁觅食行为的启发式搜索算法。主要参考了蚂蚁寻路的方式，包括以下四点：蚂蚁在移动时会留下信息素；后续蚂蚁倾向于选择信息素浓度高的路径；信息素会随时间逐渐蒸发；路径越短，信息素累积越多。

3.2.1.2 蚁群算法伪代码

```
Initialize:
    pheromone[i][j] = 0.1 ∀ edges
    best_path = None; best_length = ∞
ACO():
    for iter = 1 to max_iters:
```

```
        paths, lengths = [], []
        // 每只蚂蚁构建路径
        for ant = 1 to n_ants:
            path, length = ConstructSolution()
            if length < best_length: best_path, best_length = path, length
            paths.append(path); lengths.append(length)
        UpdatePheromone(paths, lengths)
    return best_path
ConstructSolution():
    path = [start]; current = start
    while current ≠ end:
        next = SelectNext(current, path) ?: return None, ∞
        path.append(next); current = next
    return path, length(path)

SelectNext(current, path):
    possible_steps = GetValidMoves(current)
    if empty(possible_steps): return None
    // 计算转移概率并选择下一步
    probs = [(pheromone[current][next]**α * (1/distance(next, end))**β) ∀ next in possible_steps]
    return RouletteWheelSelection(possible_steps, probs)

UpdatePheromone(paths, lengths):
    // 蒸发信息素
    pheromone *= (1 - ρ)
    // 更新信息素
    for path, len in zip(paths, lengths):
        Δτ = Q / len
        for edge in path: pheromone[edge] += Δτ
```

3.2.1.3 技术路径

关键参数如下（不同版本代码有少量修改）：

```
self.aco_params = {
    'n_ants': 20,            # 蚂蚁数量
    'n_iterations': 50,     # 迭代次数
    'evaporation': 0.1,     # 信息素蒸发率
    'alpha': 1.0,           # 信息素重要程度
    'beta': 2.0,            # 启发式信息重要程度
    'Q': 1.0,                 # 信息素增加强度
}
```

其中启发式信息矩阵使用当前位置到终点的曼哈顿距离倒数。

3.2.1.4 算法效率与适用性分析

ACO 时间复杂度为 O(N * M * T)（其中 N 是蚂蚁数量、M 是迭代次数、T 是构建单个解的时间）。ACO 空间复杂度:为 O(V^2)，即取决于信息素矩阵大小。

与 A*算法和 D*算法相比，ACO 优点和缺点都很明显。ACO 可以很好地适应复杂环境，而且可并行实现，且动态适应性强。但是同时 ACO 参数较多且调整复杂，收敛时间较长且不保证可以找到最优解。

3.2.2 粒子群算法

3.2.2.1 粒子群优化（Particle Swarm Optimization, PSO）原理

粒子群优化是一种群体智能优化算法，通过模拟鸟群或鱼群觅食行为中的协作和信息共享来寻找最优解，在该算法中每个潜在解被称为一个“粒子”，其位置代表具体的解如路径，速度决定下次移动的方向和距离，适应度函数评估解的质量以路径为例即衡量路径长度、平滑度等指标，每个粒子会记住自己的个体最优 pBest 位置和整个群体的全局最优 gBest 位置，在每次迭代中粒子根据惯性保持当前趋势并结合学习因子 c1 向 pBest 学习和学习因子 c2 向 gBest 学习更新速度和位置，从而在不断迭代过程中使粒子群倾向于向最优区域聚集最终找到近似最优解。

3.2.2.2 粒子群优化伪代码

```
// 初始化粒子群
For 每个粒子 i = 1 到 N:
    初始化 粒子 i 的位置 x[i] (例如，随机生成一条路径)
    初始化 粒子 i 的速度 v[i] (例如，设为 0 或随机小值)
    计算 粒子 i 的适应度 fitness[i] = Evaluate(x[i])
    设置 个体最优位置 pBest[i] = x[i]
    设置 个体最优适应度 pBest_fitness[i] = fitness[i]

// 初始化全局最优
设置 全局最优位置 gBest = 粒子群中适应度最优的粒子的 pBest
设置 全局最优适应度 gBest_fitness = 粒子群中适应度最优的 pBest_fitness

// 主循环
For t = 1 到 MaxIterations:
    For 每个粒子 i = 1 到 N:
        // 更新粒子速度
        For 每个维度 d = 1 到 D:
            生成 随机数 r1, r2 在 [0, 1] 之间
            v[i][d] = w * v[i][d] +
                      c1 * r1 * (pBest[i][d] - x[i][d]) +
                      c2 * r2 * (gBest[d] - x[i][d])

        // 更新粒子位置
```

```
        x[i] = x[i] + v[i]
        // (可选) 处理边界，确保粒子位置在有效范围内
        x[i] = ClampToBounds(x[i])

        // 计算新位置的适应度
        current_fitness = Evaluate(x[i])

        // 更新个体最优
        If current_fitness < pBest_fitness[i]:
            pBest[i] = x[i]
            pBest_fitness[i] = current_fitness
        // 更新全局最优
        If current_fitness < gBest_fitness:
            gBest = x[i]
            gBest_fitness = current_fitness
```

PSO 参数设置：

```
self.pso_params = {
    'n_particles': 50,        # 粒子数量
    'n_iterations': 150,     # 迭代次数
    'w': 0.729,                # 惯性权重，控制粒子保持当前运动趋势的程度
    'c1': 1.49445,             # 个体学习因子，控制粒子向个体历史最优位置移动的程度
    'c2': 1.49445,             # 社会学习因子 控制粒子向群体历史最优位置移动的程度
    'max_velocity': 1.0       # 最大速度限制
}
```

3.3 强化学习

3.3.1 强化学习原理

本项目采用深度 Q 网络（DQN）算法解决动态环境下的路径规划问题，结合深度学习与 Q-Learning，通过神经网络近似最优动作价值函数 Q*(s, a)以处理高维状态空间问题，其中智能体即路径规划器基于 DynamicRLPathPlanner 类决策，环境由 DynamicEnvironment 类定义为包含移动障碍物的动态网格世界，状态是智能体观察到的信息如自身与目标位置及最近障碍物信息，动作包括上下左右四个离散动作，奖励根据到达目标、碰撞障碍物等情况给出反馈指导学习，使用 DQNNetwork 类实现的 Q 网络估计 Q(s, a)值，经验回放机制存储经历以打破数据相关性提高样本利用率，目标网络用于稳定训练过程中的 Q 值计算，学习过程中通过策略网络和目标网络分别计算当前和下一状态的 Q 值并更新策略网络参数，探索与利用阶段采用 ε -贪婪策略平衡两者， ε 值随训练逐渐从 0.9 衰减至 0.01。

3.3.2 算法伪代码

```
// 训练循环
```

```
For episode = 1 到 M:
    获取 环境初始状态 s_1 (包含起点、终点、初始障碍物位置)
    For t = 1 到 T:
        // 1. 环境动态更新
        更新 环境中障碍物的位置

        // 2. 选择动作 (ε-贪婪)
        生成 随机数 rand 在 [0, 1] 之间
        If rand < ε:
            随机选择 动作 a_t 从所有可能动作中
        Else:
            a_t = argmax_a Q(s_t, a; θ) // 选择 Q 值最大的动作

        // 3. 执行动作并观察反馈
        在环境中执行动作 a_t
        观察 获得奖励 r_t
        观察 得到新的状态 s_{t+1}
        检查 当前回合是否结束 done_t (到达目标、碰撞、出界)
        // 4. 存储经验
        将转换 (s_t, a_t, r_t, s_{t+1}, done_t) 存入经验回放缓冲区 D
        // 5. 更新状态
        s_t = s_{t+1}
        // 6. 训练网络 (从经验回放中采样)
        If 缓冲区 D 中的样本数量 > BatchSize:
            从 D 中随机采样一个 MiniBatch 的转换 (s_j, a_j, r_j, s_{j+1}, done_j)
            // 计算目标 Q 值
            For 每个样本 j:
                If done_j:
                    y_j = r_j // 目标 Q 值
                Else:
                    y_j = r_j + γ * max_{a'} Q_target(s_{j+1}, a'; θ') // 目标 Q 值

            // 计算损失
            计算 损失 L = (1/BatchSize) * Σ [ (y_j - Q(s_j, a_j; θ))^2 ] (例如使用
Smooth L1 Loss)
            // 执行梯度下降更新策略网络参数 θ
            θ = θ - α * ∇_θ L
        // 7. 定期更新目标网络
        If t % C == 0:
            θ' = θ // 更新目标网络参数
        // 8. 检查回合是否结束
        If done_t:
            break // 结束当前回合
    // 更新探索率 ε
```

```
        ε = max(ε_min, ε * ε_decay)
// 训练结束
输出  训练好的策略网络  Q(s, a;  θ )
```

3.3.3 奖励机制

最终版代码奖励机制如下：（前序版本有不同程度微调，最终确认最后版本）

1. 使用对数尺度来平滑距离变化带来的奖励
2. 对障碍物的惩罚相对较大（-10）
3. 到达目标的奖励适中（+10）
4. 有小的移动成本（-0.1）来鼓励寻找更短的路径
5. 对超出边界有较大惩罚（-50）

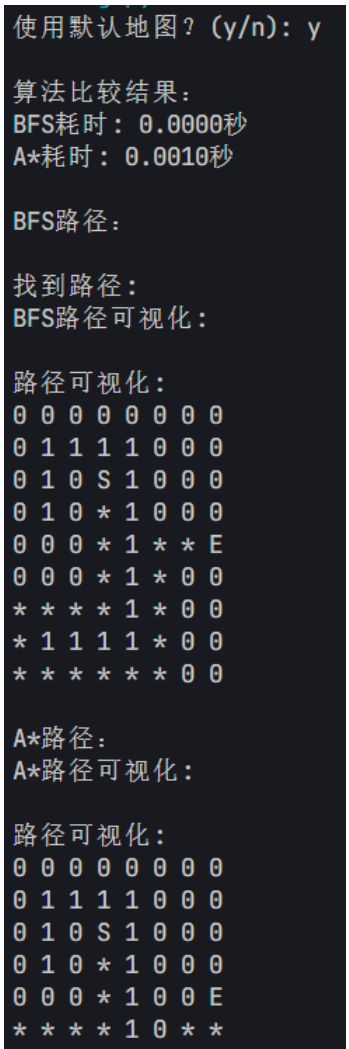
四、实验结果

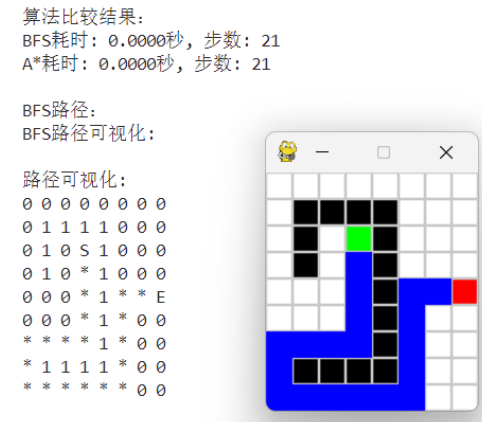
注：最终整合版运行结果请参考附录，这里展示的是每个板块单独的测试

4.1 传统寻路算法运行结果

4.1.1 默认地图的最短路径规划

后台已提前写入了默认地图的数据，可以直接调用，会直接显示运行结果。





```
(mcm2025) PS D:\dpan\Uni\学习study\同济课内课程\必修重要课程\AI design\路径规划> & C:/Users/16943/anaconda3/envs/mcm2025/python.exe "d:/dpan/Uni/学习study/同济课内课程/必修重要课程/AI design/路径规划/pathfinding.py"
pygame 2.6.1 (SDL 2.28.4, Python 3.9.18)
Hello from the pygame community. https://www.pygame.org/contribute.html

请选择地图初始化方式:
1. 使用默认地图
2. 随机生成地图
3. 自行绘制地图
4. 生成大规模地图(使用pygame可视化)
请输入选项(1/2/3/4): 1

正在计算路径...

算法比较结果:
BFS耗时: 0.0000秒, 步数: 21
A*耗时: 0.0000秒, 步数: 21
```

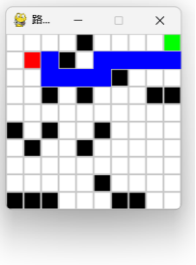
4.1.2 随机生成地图的最短路径规划

选择随机生成地图之后，会要求填写地图行数和列数（要求在 5-15 之间，如果大于 15，可以适用生成大规模地图来实现）。

之后会随机生成 5 个地图并打印在面板供用户选择，其中会显示障碍物比例、终点起点路径长度（曼哈顿距离）。

```
请选择地图初始化方式:
1. 使用默认地图
2. 随机生成地图
3. 自行绘制地图
4. 生成大规模地图(使用pygame可视化)
请输入选项(1/2/3/4): 2
请输入地图行数(5-15): 10
请输入地图列数(5-15): 10

正在生成10x10规格的候选地图...

生成完成，请选择要使用的地图:
1) 规格: 10x10 障碍物: 18(18.0%) 路径长度: 8
   00 01 02 03 04 05 06 07 08 09
00 ■  □  ■  □  ■  □  □  □  ■  □
01 □  □  □  ■  □  □  □  □  □  □
02 ■  □  □  □  ■  □  □  □  □  □
03 □  □  □  □  ■  □  □  □  ■  □
04 □  □  □  □  □  □  □  □  □  □
05 □  ■  □  ⚑  □  □  □  ■  □  ■
06 □  ■  □  □  □  □  □  □  □  □
07 □  □  □  ■  □  □  □  □  □  □
08 □  □  ■  □  □  ■  □  □  □  □
09 ▩  □  ■  ■  □  □  □  □  □  □

2) 规格: 10x10 障碍物: 18(18.0%) 路径长度: 3
   00 01 02 03 04 05 06 07 08 09
00 □  □  ■  ■  ■  ▩  □  ⚑  ■  □
01 □  □  □  □  □  □  □  ■  ■  □
02 □  □  □  □  □  □  ■  ■  □  ■
03 □  □  □  □  ■  □  □  □  □  □
04 □  □  □  □  □  □  □  □  □  □
05 □  □  ■  □  □  □  □  □  □  □
06 □  □  □  □  ■  □  ■  □  □  □
07 □  □  □  ■  ■  □  □  □  □  □
08 □  □  □  □  □  ■  □  ■  ■  □
09 □  □  □  □  □  □  □  □  □  □

3) 规格: 10x10 障碍物: 18(18.0%) 路径长度: 10
```

```
5) 规格: 10x10 障碍物: 18(18.0%) 路径长度: 12
   00 01 02 03 04 05 06 07 08 09
00 □  □  □  □  ■  □  □  □  ⚑
01 □  ▩  □  ■  □  □  □  □  □
02 □  □  □  □  □  □  ■  □  □
03 □  □  ■  □  ■  □  □  ■  ■
04 □  □  □  □  □  □  □  □  □
05 ■  □  ■  □  □  ■  □  □  □
06 □  ■  □  □  ■  □  □  □  □
07 □  □  □  □  □  □  □  □  □
08 □  □  □  □  □  ■  □  □  □
09 ■  ■  ■  □  □  □  ■  ■  □

请输入选择(1-5): 5

已加载选择的地图!

正在计算路径...

算法比较结果:
BFS耗时: 0.0000秒, 步数: 12
A*耗时: 0.0000秒, 步数: 12

BFS路径:
BFS路径可视化:
0-可通行 1-障碍 S-起点 E-终点
路径可视化:
0 0 0 0 1 0 0 0 0 S
0 E * 1 0 * * * * *
0 0 * * * * 1 0 0 0
0 0 1 0 1 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0
1 0 1 0 0 1 0 0 0 0
0 1 0 0 1 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
1 1 1 0 0 0 1 1 0 0

路径已显示，按ESC或关闭窗口继续...
```



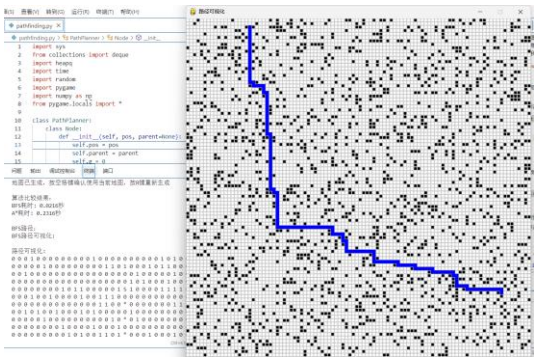
4.1.3 自主绘制地图

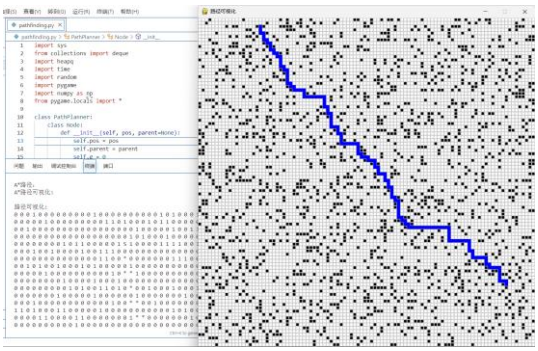
同时本程序支持自主绘制地图，根据面板提示输入自主地图即可。如右图所示。

```
请选择地图初始化方式:
1. 使用默认地图
2. 随机生成地图
3. 自行绘制地图
4. 生成大规模地图(使用pygame可视化)
请输入选项(1/2/3/4): 3
输入行数: 5
输入列数: 5
输入矩阵(每行用空格分隔，0-可通行 1-障碍 2-起点 3-终点):
0 0 0 0 2
0 1 1 1 1
0 0 0 0 0
1 1 1 1 0
3 0 0 0 0

正在计算路径...

算法比较结果:
BFS耗时: 0.0000秒, 步数: 17
A*耗时: 0.0000秒, 步数: 17

BFS路径:
BFS路径可视化:
0-可通行 1-障碍 S-起点 E-终点
路径可视化:
* * * * S
* 1 1 1 1
* * * * *
1 1 1 1 *
E * * * *
```

4.1.4 大规模地图的随机生成

为实现大地图的显示和路径规划，调用 Pygame 库绘制。与 4.2 操作一致即可选择 16-100 行/列的大规模地图。如下图所示。



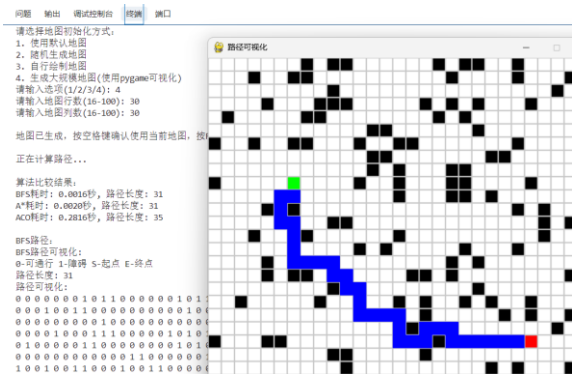


4.2 基于生物只能的寻路算法运行结果

4.2.1 蚁群算法

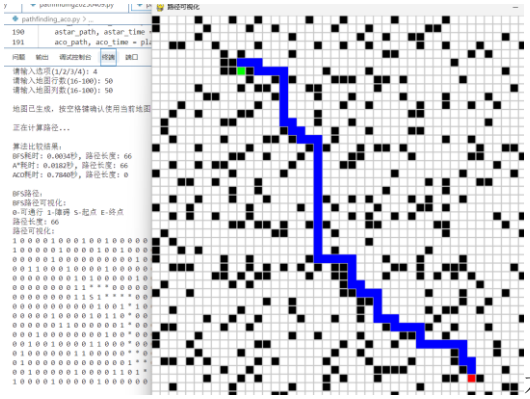
蚁群算法中每个“蚂蚁”具有随机性，导致算法的局限性非常明显。

一是可能陷入局部最优解而找不到全局最优，如下图所示：（BFS 和 A*算法的最短路径为 31 曼哈顿距离，而蚁群算法为 35 曼哈顿距离）

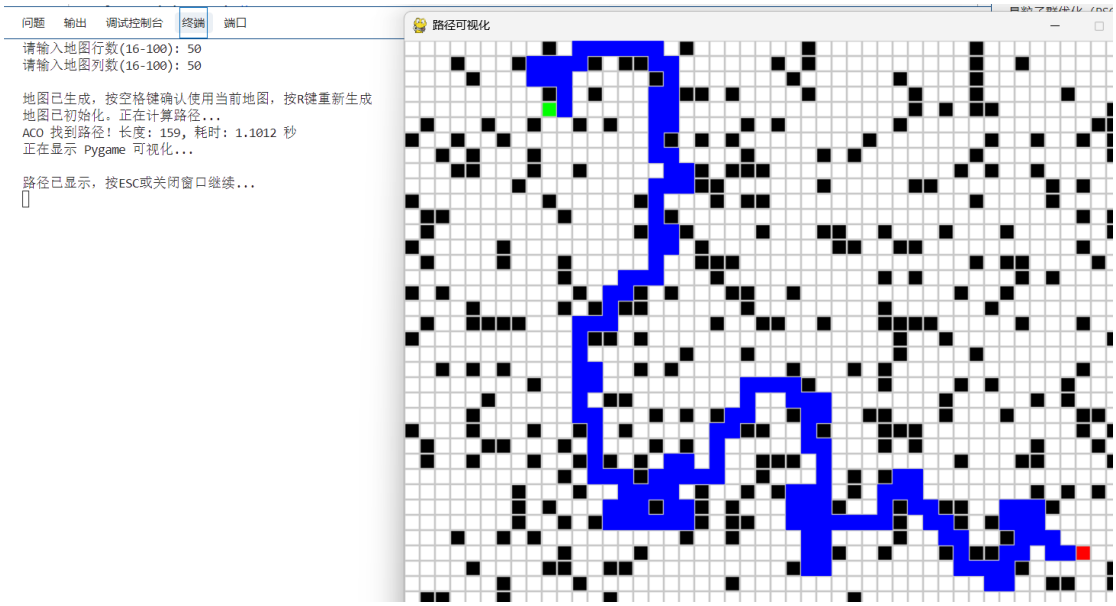


右图为蚁群算法的解

二是可能无法找到路径，如下图所示：


右图显示的是 BFS 算法的路径

另一个地图的展示如下，可以很明显地看出和我们日常看到的蚁群确实很像。事实上，在自然界，如果搬食物的时间足够长，由于蚁群寻路具有个体随机性，最终蚁群依然可以找到最短路径。



4.2.2 粒子群算法

第一次运行出现失败，经研究发现为参数设置不当。

```
请选择地图初始化方式：
1. 使用默认地图
2. 随机生成地图
3. 自行绘制地图
4. 生成大规模地图(使用pygame可视化)
请输入选项(1/2/3/4): 1

正在计算路径...

算法比较结果：
BFS耗时: 0.0000秒, 路径长度: 21
A*耗时: 0.0000秒, 路径长度: 21
PSO耗时: 1.0590秒, 路径长度: 0
```

在阅读 CSDN 上相关教程后，根据现有研究成果的参数进行修改。

修改后 PSO 参数设置：

```
self.pso_params = {
```

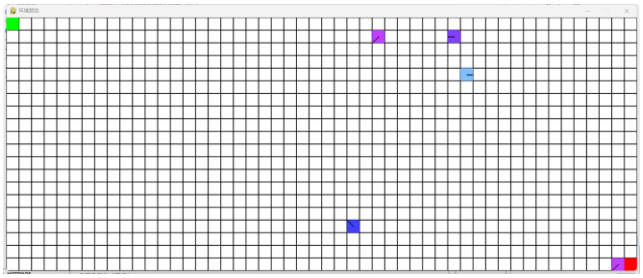
```
    'n_particles': 50,        # 粒子数量
    'n_iterations': 150,      # 迭代次数
    'w': 1→0.729,                 # 惯性权重，控制粒子保持当前运动趋势的程度
    'c1': 0.5→1.49445,            # 个体学习因子，控制粒子向个体历史最优位置移动的程度
    'c2':0.5→1.49445,             # 社会学习因子 控制粒子向群体历史最优位置移动的程度
    'max_velocity': 1.0       # 最大速度限制
}
```

在改进之后，随机生成了多个地图尝试，但发现经常不是最短路径。这是由于其随机性，很容易陷入局部最优的策略中。由于受算法的局限性影响，此问题难以在不改变算法本质的基础上完全修改，现有研究也只是对此做了不少优化。由于时间限制，本人并没有再对算法在这方面做优化。

```
算法比较结果:
BFS耗时: 0.0000秒, 路径长度: 7
A*耗时: 0.0000秒, 路径长度: 7
PSO耗时: 0.0150秒, 路径长度: 9
```

4.3 强化学习在动态路障中的寻路算法

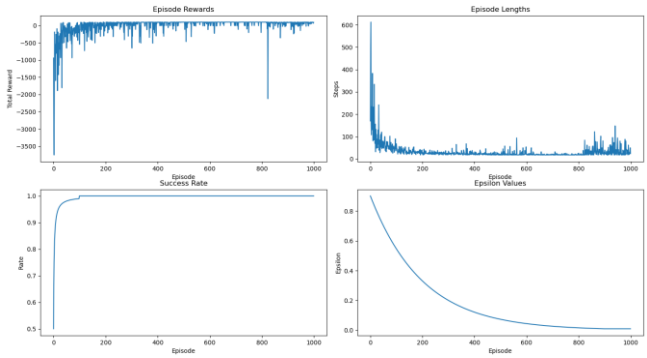
下图展示的是制作的动态障碍物地图，其中绿色为起点，红色为终点，其余五个为动态障碍物，障碍物中间的线表示上一时刻此障碍物的移动方向。由于这是第一遍尝试，所以对于起点和终点没有加随机生成函数，而是固定在左上角和右下角，后期有加上随机生成。



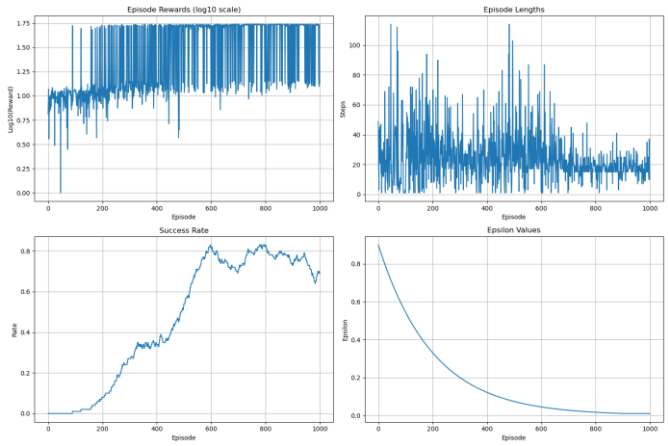
在运行中，每一轮的运行情况都会在终端打印，方便监控运行情况。

```
Episode 840, Steps: 16, Total Reward: 121.71, Epsilon: 0.01
Episode 850, Steps: 21, Total Reward: 119.04, Epsilon: 0.01
Episode 860, Steps: 27, Total Reward: 118.80, Epsilon: 0.01
Episode 870, Steps: 22, Total Reward: 121.90, Epsilon: 0.01
Episode 880, Steps: 13, Total Reward: 121.80, Epsilon: 0.01
Episode 890, Steps: 22, Total Reward: 121.90, Epsilon: 0.01
Episode 900, Steps: 14, Total Reward: 117.38, Epsilon: 0.01
Episode 910, Steps: 42, Total Reward: 110.44, Epsilon: 0.01
Episode 920, Steps: 31, Total Reward: 113.37, Epsilon: 0.01
Episode 930, Steps: 17, Total Reward: 122.01, Epsilon: 0.01
Episode 940, Steps: 15, Total Reward: 121.60, Epsilon: 0.01
Episode 950, Steps: 22, Total Reward: 118.60, Epsilon: 0.01
Episode 960, Steps: 28, Total Reward: 106.66, Epsilon: 0.01
Episode 970, Steps: 16, Total Reward: 121.32, Epsilon: 0.01
Episode 980, Steps: 16, Total Reward: 113.73, Epsilon: 0.01
Episode 990, Steps: 13, Total Reward: 122.80, Epsilon: 0.01
Episode 1000, Steps: 39, Total Reward: 105.84, Epsilon: 0.01

寻找路径...
找到路径!
```

最终生成对应的各个参数的变化表。其中 Rewards 由于前期有很多负值很大的样本，事实上最终 Rewards 稳定在 100 上下而非 0，现在看来其实可以 log 化可能更方便查看（但我懒了 qaq）



改了 log 显示：



运行动图可以在压缩文件包里可以查看，也可以运行相关代码后自动生成。同时我也放了四个 gif 样例在 GitHub 仓库中，为防止抄袭，我会在 DDL 之后一天内将 PDF 更新到此仓库内。参考 GitHub 仓库静态展示网页：https://zhangjialin53.github.io/AI-design/

五、总结与分析

这次作业让我对有关寻路的几个算法都有了基础的了解。之前在其他课程中已经学习和研究过 A*算法和 BFS 广度优先算法。此次作业中系统查看了蚁群算法和粒子群算法的核心思想，搜索资料的过程中翻到了 A*算法、蚁群算法在知识图谱嵌入 KGE 方面的类比运用，因为科研课题研究与此有关，顺便一起做了了解。

强化学习是首次接触，之前有了解过通过奖励与惩罚的机制训练，此次作业为首次亲身实践，还蛮好玩的。

在运行的过程中看到了传统方法确实在高维复杂场景运用的局限性，比如设置到 100*100 的地图，时间复杂度就大大增加，优点也很明显，保证算力的情况下可以保证找到对应最短路径。生物仿生智能的蚁群算法和粒子群算法在复杂场景的运用更好一点，但具有一定的随机性，同时也不保证可以收敛，即不保证找到最短路径。

强化学习的办法可以应用在有动态路障的场景，但训练过程中只能让它知道如何拿高分，这个奖励和惩罚机制的数值还是需要多次测试的，同时也不保证最终测试的时候一定能避开障碍物。

六、参考文献

A 星(A*、A Star)路径规划算法详解_a 星算法路径规划-CSDN 博客

A* 算法解决最短路径问题_最短路径问题中障碍问题-CSDN 博客

基于蚁群算法的路径规划_蚁群算法路径规划详细步骤-CSDN 博客

粒子群算法（PSO）求解路径规划_粒子群算法路径规划-CSDN 博客

利用强化学习 Q-Learning 实现最短路径算法-CSDN 博客

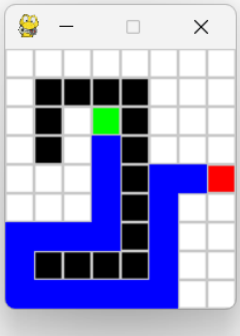
七、附录

最终运行结果：运行主菜单文件 `main_menu.py`

```
pygame 2.6.1 (SDL 2.28.4, Python 3.9.18)
Hello from the pygame community. https://www.pygame.org/contribute.html

===== 路径规划算法菜单 =====
1. 广度优先搜索 (BFS)
2. A* 搜索
3. 粒子群优化 (PSO)
4. 蚁群优化 (ACO)
5. 强化学习 (DQN)
0. 退出
============================
请输入选项编号:
```

运行 BFS：（使用默认地图）

```
请输入选项编号：1

--- 运行广度优先搜索 (BFS) ---

请选择地图初始化方式：
1. 使用默认地图
2. 随机生成地图
3. 自行绘制地图
4. 生成大规模地图(使用pygame可视化)
请输入选项(1/2/3/4)：1
地图已初始化。正在计算路径...
BFS 找到路径！长度：21，耗时：0.0000 秒
正在显示 Pygame 可视化...

路径已显示，按ESC或关闭窗口继续...
```



运行 Astar：（选择随机生成地图）

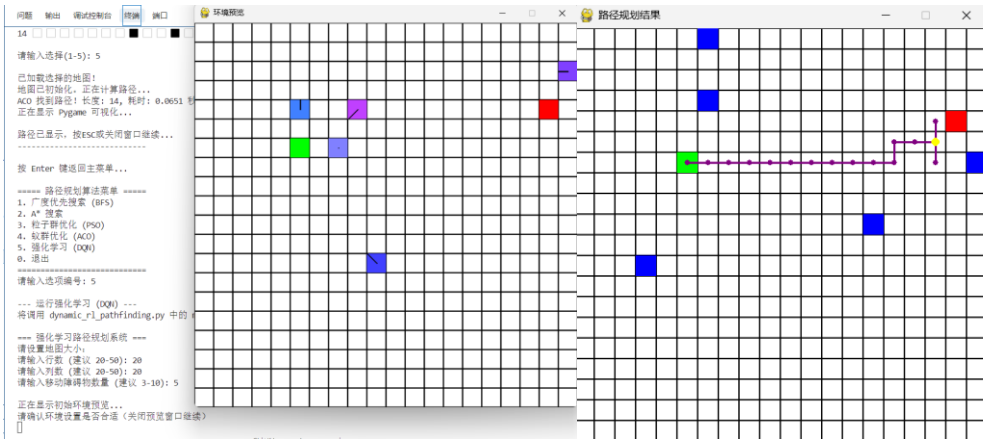
```
--- 运行 A* 搜索 ---

请选择地图初始化方式：
1. 使用默认地图
2. 随机生成地图
3. 自行绘制地图
4. 生成大规模地图(使用pygame可视化)
请输入选项(1/2/3/4)：2
请输入地图行数(5-15)：10
请输入地图列数(5-15)：10

正在生成10x10规格的候选地图...

生成完成，请选择要使用的地图：
1) 规格：10x10 障碍物：18(18.0%) 路径长度：7
   00 01 02 03 04 05 06 07 08 09
00 □ □ □ □ □ □ ■ □ □ ■
01 ■ ⚑ □ ■ □ □ □ □ □ □
02 □ ■ □ □ □ ■ ■ □ ■ □
03 □ ■ ■ □ □ □ □ □ □ □
04 □ □ □ □ □ □ □ □ □ □
05 □ □ □ ▩ □ □ □ □ □ ■
06 ■ □ ■ □ □ □ □ □ □ □
07 □ ■ □ □ □ □ □ ■ □ □
08 □ □ □ □ □ □ □ ■ □ □
09 □ □ □ □ ■ □ □ □ ■ □

2) 规格：10x10 障碍物：18(18.0%) 路径长度：11
   00 01 02 03 04 05 06 07 08 09
00 □ □ □ ■ □ □ □ □ ■ □
01 □ □ □ □ □ □ □ ■ □ □
02 ■ □ ■ □ □ ▩ □ □ □ □
03 □ □ □ □ ■ □ □ □ □ □
04 □ □ ■ ■ □ □ □ □ □ □
05 □ □ ■ □ □ □ ■ □ □ ■
```

```
5) 规格：10x10 障碍物：18(18.0%) 路径长度：6
   00 01 02 03 04 05 06 07 08 09
00 □ ■ □ □ ■ □ □ ■ □ ■
01 □ ■ □ □ □ □ □ ■ □ □
02 □ ■ □ ■ ▩ ■ □ □ □ □
03 □ □ □ □ □ □ □ □ □ □
04 □ □ □ □ ■ □ ■ □ □ □
05 □ □ □ □ □ □ □ □ □ □
06 □ □ □ ⚑ □ ■ □ □ □ □
07 □ □ □ □ □ □ ■ □ □ ■
08 □ □ □ □ □ □ □ □ □ □
09 ■ □ □ □ □ ■ □ ■ ■ □

请输入选择(1-5)：2

已加载选择的地图！
地图已初始化。正在计算路径...
A* 找到路径！长度：11，耗时：0.0000 秒
正在显示 Pygame 可视化...

路径已显示，按ESC或关闭窗口继续...
```

运行粒子群算法：（选择随机生成大规模地图 50*50）

运行蚁群算法：（选择随机生成地图）

运行强化学习：


问题 输出 调试控制台 终端 端口
14
请输入选择(1-5): 5
已加载选择的地图!
地图已初始化，正在计算路径...
ACO 找到路径! 长度: 14, 耗时: 0.0651 秒
正在显示 Pygame 可视化...
路径已显示，按ESC或关闭窗口继续...
按 Enter 键返回主菜单...
===== 路径规划算法菜单 =====
1. 广度优先搜索 (BFS)
2. A* 搜索
3. 粒子群优化 (PSO)
4. 蚁群优化 (ACO)
5. 强化学习 (DQN)
0. 退出
请输入选项编号: 5
--- 运行强化学习 (DQN) ---
将调用 dynamic_rl_pathfinding.py 中的
=== 强化学习路径规划系统 ===
请设置地图大小:
请输入行数 (建议 20-50): 20
请输入列数 (建议 20-50): 20
请输入移动障碍物数量 (建议 3-10): 5
正在显示初始环境预览...
请确认环境设置是否合适（关闭预览窗口继续）
环境预览
路径规划结果