机器人避障

寻径问题

——AI 课程设计作业 2

实验报告

计算机科学与技术学院

2352018 刘彦

2025 年 4 月 22 日

AI 课程设计作业 2——搜索模块：机器人避障寻径问题

2352018 刘彦

1. 问题概述

⑴问题的直观描述

①实验内容

机器人避障寻径是机器人领域的一项重要技术，该技术要求实现机器人在复杂环境中的自主导航。在本实验中，我们给定如下的机器人避障寻径场景图：

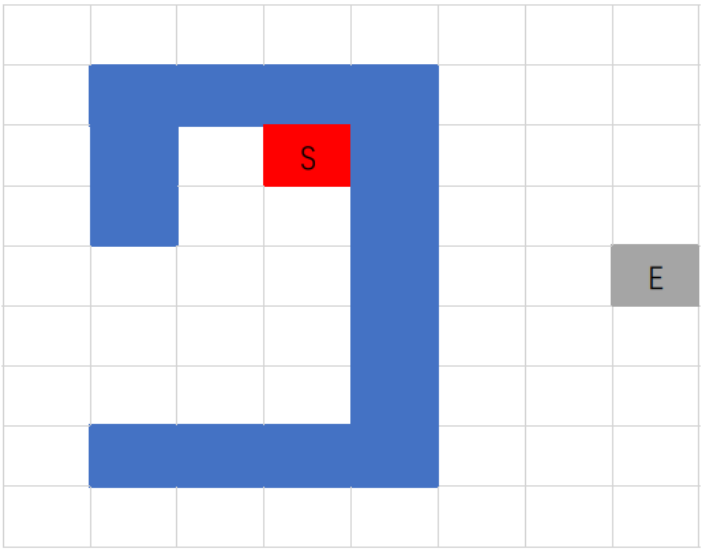


图 1 机器人避障寻径场景图

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。

并且在此基础上自主设计更高难度场景图，并结合强化学习、深度强化学习等算法解决问题。分析所设计算法的局限性，及改进点，并给出最短路径查找过程的动态化展示。

②实验要求

掌握机器人路径规划与避障的基本原理与方法，通过在一个给定的地图中，设计并实现机器人从起点 S 绕开障碍物到达终点 E 的最短路径搜索算法，学生需选择合适的图搜索算法（如 A*、Dijkstra 等）或自定义规则完成路径规划任务。实验要求输出路径坐标及可视化结果。

⑵解决问题的思路分析

在机器人导航领域中，避障寻径是一个经典的路径规划问题。该问题可以抽象为在一个

二维网格地图中，已知起点和终点，以及障碍物的分布情况，需要为机器人找到一条从起点到终点的最优路径。这里的"最优"通常指路径最短或代价最小，同时需要确保路径能够完全避开所有障碍物。在实现上，我使用字典存储距离信息和前驱节点，通过优先队列管理待访问节点，结合算法计算出最短路径。

主要解决思路有两个——Dijkstra 算法和 A*算法，并且 A*算法还可以结合机器学习（Random Forest）和深度学习，以达到更好的性能。

Dijkstra 算法的核心思路是：首先基于广度优先搜索的最短路径算法，使用优先队列维护待访问节点。该算法不使用启发式信息，完全基于实际距离进行搜索，适用于无方向性搜索场景。该算法实现简单，可以保证结果准确，保证找到最短路径，但搜索范围较大，效率相对较低，只能适合简单场景使用。

A*算法相较于 Dijkstra 算法增加了启发式函数，使用曼哈顿距离作为启发式估计的代价。结合实际代价和预估代价，利用评估每个节点优先级的核心公式评估节点优先级。

$$f(n) = g(n) + h(n)$$

其中，$f(n)$表示从起点出发，通过当前节点 n 到终点的总估计代价。它是我们排序和选择下一个节点时的依据。$g(n)$表示从起点到当前节点 n 的实际代价，也就是已经走过的路径长度。$h(n)$表示从当前节点 n 到终点的预估代价，即启发式函数。常用的估算方法是曼哈顿距离（在没有斜向移动时）或欧几里得距离（允许斜向移动时），在本任务中，我使用曼哈顿距离进行启发式估计。A*算法搜索效率高于 Dijkstra 算法，也保证找到最短路径，适合有明确目标的搜索。

A*算法结合机器学习（Random Forest）的核心思路是将机器学习与传统路径规划算法相结合，通过训练一个随机森林模型来预测从任意节点到目标点的距离，从而作为 A*中的启发式函数$h(n)$，最终提升路径搜索的效率与智能性。模型输入不仅包括当前节点的位置（如行列坐标），还加入了环境特征（如周围的障碍情况、距离边界的远近等），更全面地刻画节点状态。相比深度学习模型，训练效率更高，调参简单，预测结果也较为稳定，可以作为基础 A*算法的补充。

除了可以通过训练一个随机森林模型作为 A*中的启发函数，还可以使用神经网络学习启发式函数，可以做到更丰富的特征提取，实现批量训练和优化策略。拥有特征学习能力强，可以处理复杂场景的特点，但相对而言计算开销较大。

这四种方法展示了从传统算法到现代机器学习的演进过程，每种方法都有其特定的应用场景和优势，可以满足本次实验对搜索模块的学习要求。

2.算法设计

⑴算法原理

①Dijkstra 算法

Dijkstra 算法是一种经典的单源最短路径算法，用于寻找图中一个起点到所有其他点的最短路径。其核心思想是“贪心”：在每一步中选择当前距离起点最近的未访问节点进行扩展。

为了实现在一个二维网格地图（有障碍物）中从起点找到终点的最短路径，我先对有障碍物的二维网格地图进行处理，每个格子初始化为“无穷大距离”，起点距离设为 0，并且使用最小堆（优先队列）来维护当前待扩展的节点（按最短距离排序）。

在算法主循环中，从优先队列中取出当前最短路径节点并将其标记为已访问。遍历其上下左右 4 个邻居节点，若该邻居未被访问、且不是障碍物，则计算从起点经当前最短路径节点到达邻居的新距离，若这个新距离小于邻居节点的已有距离，则更新，将邻居节点重新加入优先队列。若成功到达终点，反向追踪前驱节点，重建最短路径。

②A*算法

A*算法是一种启发式搜索算法，广泛应用于路径规划问题。目标是在地图上找到一条从起点到终点的“代价最低”的路径。

A*算法在搜索时综合考虑两个因素：

- 已走过的路径代价（$g(n)$）：从起点走到当前节点 n 的实际代价。
- 预计剩余代价（$h(n)$）：从当前节点 n 到目标的启发式估计代价（即预测）。

两者相加构成总评估函数：

$$f(n) = g(n) + h(n)$$

算法优先探索$f(n)$最小的节点。

在预计剩余代价时，我使用曼哈顿距离作为启发函数，这种估计在只能上下左右移动的网格中非常有效：

$$h(n) = |x_{current} - x_{end}| + |y_{current} - y_{end}|$$

与 Dijkstra 算法不同的是，A*算法在实现时，先将起点加入优先队列，该队列根据$f(n)$排序，然后进入算法主循环中，先取出$f(n)$最小的节点，遍历其上下左右 4 个邻居节点，若该邻居未被访问、且不是障碍物，则更新其$g(n)$，计算$f(n)$后加入优先队列。若成功到达终点，反向追踪前驱节点，重建最短路径。

③结合机器学习（Random Forest）的 A*算法

传统的 A*算法的$h(n)$通常使用几何距离（如曼哈顿、欧几里得距离），结合机器学习（Random Forest）后使用随机森林模型预测启发值。

我的目标是预测 “从某个节点出发，到终点的实际路径距离”，考虑到影响因素包括：当前节点的位置和当前节点周围的环境（有没有障碍、绕路多不多），因此我设计了以下两个特征：

● 当前节点的(x,y)坐标

用于学习“当前位置”与终点位置之间的整体空间关系；相当于一个弱版的几何启发信息。

● 该节点周围 3x3 邻域内的障碍物密度

作用是评估当前位置是否“拥堵”或接近死路；以及评估绕远路的可能性，越多障碍，越可能绕远路。

用模型替代几何启发值：预测出来的值比欧几里得距离更贴合真实环境（考虑障碍分布），从而提高搜索效率。

④结合深度学习的 A*算法

在结合机器学习（Random Forest）后使用随机森林模型预测启发值之后，我又尝试了使用神经网络代替传统启发式函数。本质上，A*中使用的启发式函数$h(n)$被泛化为一个数据驱动的函数$h_\theta(n)$，θ为网络权重。

输入特征主要包括用于标准化处理的“当前节点横坐标除以地图宽度”、“当前节点纵坐标除以地图高度”；用作参考启发式的“当前点到终点的曼哈顿距离”；反映局部环境复杂度的“当前点周围 5×5 区域中障碍的占比”。这些信息有助于捕捉空间关系，引导网络学习避障策略。

对于神经网络结构方面，我设计的是一个前馈全连接神经网络（Feedforward Neural Network），专门用于回归任务（预测启发式路径代价），分为输入层、隐藏层和输出层。输入维度是 4，表示你提取的 4 个特征，信息依次传递到一个或多个隐藏层，通过加权求和和激活函数进行非线性变换，最后传递至输出层给出预测结果，具体结构在后面展示。数据标签是从当前点到终点的真实路径代价（最短距离）。

在训练过程中，我使用正则化方法（如 L2、Dropout）防止模型过度拟合训练数据，并添加早停机制，在验证集性能不再提升时终止训练，避免过度学习。同时，我还加入了预测值合理性约束，如果神经网络预测的启发式距离比曼哈顿距离大出太多（比如超过了 1.5 倍），那就放弃神经网络的预测值，改用传统的曼哈顿距离作为回退方案。这有效避免了非可采纳启发式函数造成搜索不最优或失败，保证算法最坏情况下等价于传统 A*算法，属于保底策略。

⑵算法功能

这些算法旨在解决智能路径规划问题，在包含障碍物的网格环境中，为机器人找到一条从起点到终点的最优路径。并通过机器学习或深度学习优化传统 A*算法的启发式函数，减少搜索空间。

最终找出一条解决方案并输出，可视化展示解决方案。在可视化过程中，会用黄色“×”来描述寻路过程，红色“×”来描述寻路过程中找到的正确路径，在寻路过程结束后，会用

实线展示寻找到的最优路径。同时输出找到的最短路径坐标、长度、总探索步数和算法总耗时。

⑶对 python 机器学习和深度学习相关库的介绍

①scikit-learn(sklearn)

这是 python 机器学习库的标准工具，提供简单高效的数据挖掘和数据分析工具，构建在 NumPy、SciPy 和 matplotlib 之上，它功能完整且有统一的 API 设置，常见功能如下：

```
from sklearn import (
    preprocessing,      # 数据预处理
    model_selection,  # 模型选择
    metrics,          # 评估指标
    ensemble         # 集成学习
)
```

下面特别介绍本次实验使用的 RandomForestRegressor 函数，这是一个随机森林回归器，基于决策树的组合预测。它是一种集成学习方法（Ensemble Learning），通过组合多个决策树（每棵树预测一个结果），然后对这些结果取平均，来提升模型的性能和稳定性。

常用参数示例：

```
from sklearn.ensemble import RandomForestRegressor
# 常用参数示例
rf_model = RandomForestRegressor(
    n_estimators=100,     # 树的数量
    max_depth=None,       # 树的最大深度
    min_samples_split=2, # 分裂所需最小样本数
    random_state=42       # 随机种子
)
```

这个函数对噪声和异常值不敏感，可自动评估特征重要程度，使用示例如下：

```
from sklearn.ensemble import RandomForestRegressor

# 1. 创建模型

rf_model = RandomForestRegressor(n_estimators=100)

# 2. 训练模型

rf_model.fit(X_train, y_train)

# 3. 预测

predictions = rf_model.predict(X_test)

# 4. 特征重要性分析

importances = rf_model.feature_importances_
```

②PyTorch

PyTorch 是一个基于 Python 的 深度学习框架，它提供了灵活、易用的张量操作接口和强大的神经网络构建工具，可以非常轻松地定义自己的深度神经网络，下面给出一个神经网络模块的调用示例：

```
import torch

import torch.nn as nn

class Net(nn.Module):

    def __init__(self):

        super(Net, self).__init__()

        self.fc = nn.Linear(10, 5)
```

(4)算法思路

①Dijkstra 算法

在 dijkstra_with_steps 函数中，算法通过优先队列（pq）实现 Dijkstra 路径搜索。首先，初始化所有节点的最短距离为无穷大，起点的距离为 0，起点加入队列（pq = [(0, start)]）。在主循环中，每次弹出队列中距离最小的节点（current_dist, current = heapq.heappop(pq)），然后检查其邻居节点。如果邻居可通行且未访问过，就更新其最短距离，并将其加入队列。当到达终点时，通过 previous 字典回溯路径（while current is not None: path.append(current)），最终返回最短路径和探索过程。

②A*算法

在 astar_with_steps 函数中，A*算法用于搜索最短路径。首先，初始化每个节点的距离为无穷大，起点的距离为 0，并将其加入优先队列 pq。队列中的元素是三元组（f_score, g_score, node），其中 f_score 是当前节点的总估计成本（g_score + h_score），g_score 是起点到当前节点的实际距离，h_score 是当前节点到终点的曼哈顿启发式估计距离。每次从队列中弹出具有最低 f_score 的节点进行扩展，更新其邻居节点的距离，并将其加入队列。最终，当终点被找到时，通过回溯 previous 字典重建路径。算法返回最短路径和探索过程。

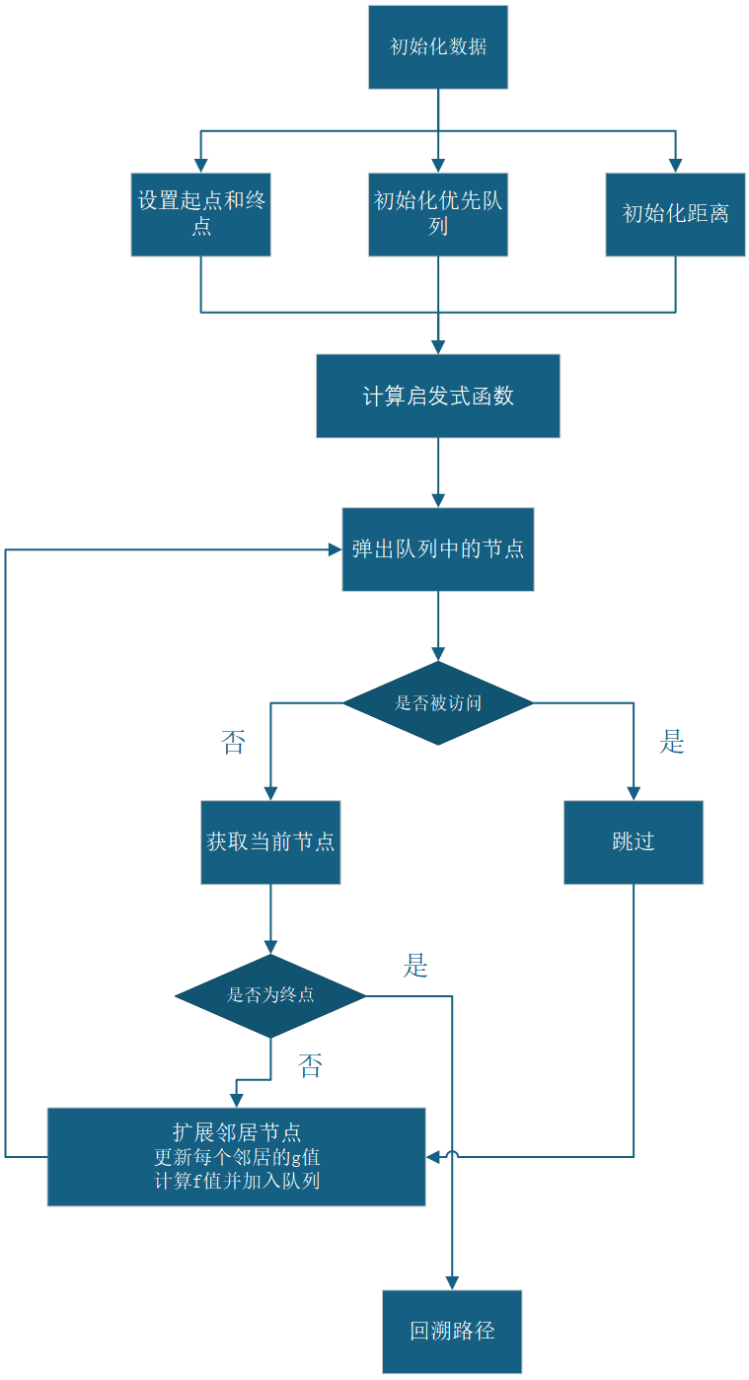


图 2 A*搜索算法结构图

在结合机器学习（Random Forest）的 A*算法的实现中，是在计算启发式函数阶段使用随机森林模型预测启发值，同样的，在结合深度学习的 A*算法的实现中，是在计算启发式函数阶段使用神经网络预测启发值，在这里，$h(n)$被泛化为一个数据驱动的函数$h_\theta(n)$，其他流程类似。深度学习中的神经网络结构如下：

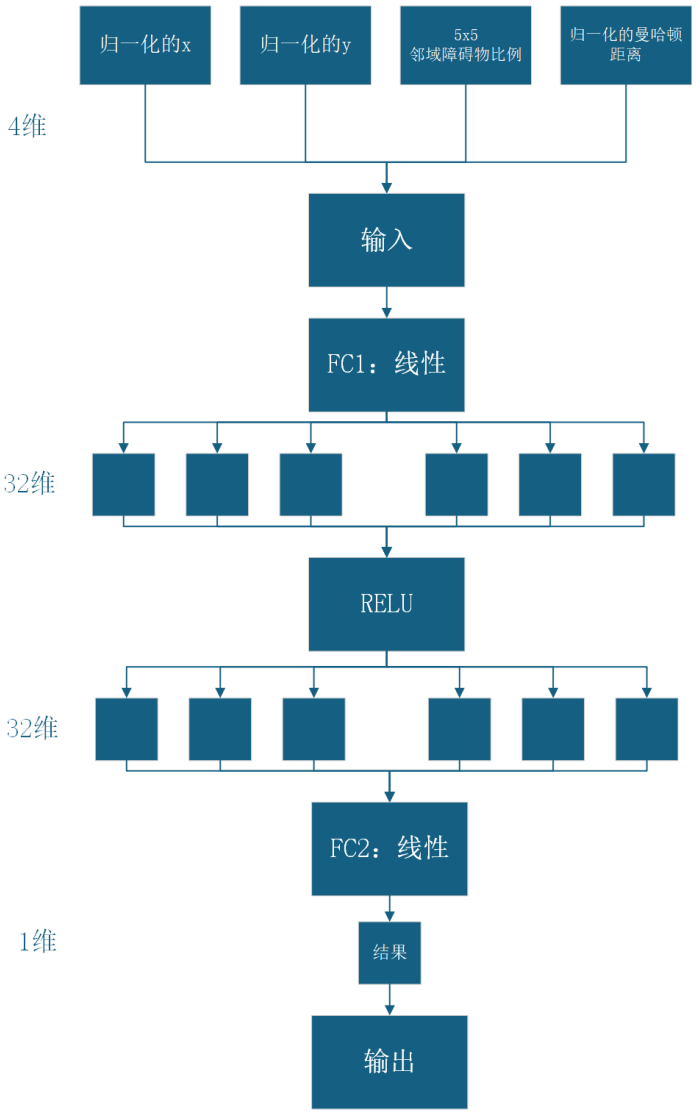


图 3 神经网络结构图

3. 算法实现

(1)Dijkstra 算法实现函数

```
def dijkstra_with_steps(grid, start, end):
    start_time = time.perf_counter()  # 使用更高精度的计时器
    step_times = []  # 记录每一步时间和截至该步的总时间（格式：
(step_duration, total_so_far)）
    rows, cols = grid.shape
    distances = {(i, j): float('inf') for i in range(rows) for j in
range(cols)}
    distances[start] = 0
```

```
    previous = {(i, j): None for i in range(rows) for j in range(cols)}
    pq = [(0, start)]
    visited = []
    explored = []  # 记录探索顺序
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    while pq:  # 算法主循环
        step_start = time.perf_counter()  # 记录每步开始时间
        current_dist, current = heapq.heappop(pq)
        if current in set(visited):
            continue
        visited.append(current)
        explored.append(current)
        step_duration = time.perf_counter() - step_start  # 本步耗时
        total_so_far = time.perf_counter() - start_time #截至当前的总时间
        step_times.append((step_duration, total_so_far))  # 记录每步时间
和总时间
        if current == end:
            break
        # 遍历上下左右四个方向，更新邻居节点的距离和前驱节点
        for dx, dy in directions:
            next_x, next_y = current[0] + dx, current[1] + dy
            neighbor = (next_x, next_y)
            if (0 <= next_x < rows and 0 <= next_y < cols and
                    grid[next_x, next_y] == 0 and neighbor not in
set(visited)):
                new_dist = current_dist + 1
                if new_dist < distances[neighbor]:
                    distances[neighbor] = new_dist
                    previous[neighbor] = current
                    # 优先队列中取出距离最短的节点
                    heapq.heappush(pq, (new_dist, neighbor))
    # 重建路径
    path = []
    current = end
    while current is not None:
        path.append(current)
        current = previous[current]
    path.reverse()
    total_time = time.perf_counter() - start_time  # 计算总时间
    return path if distances[end] != float('inf') else None, explored,
step_times, total_time
```

(2)A*算法实现函数

```
def astar_with_steps(grid, start, end):
    start_time = time.perf_counter()  # 使用更高精度的计时器
```

```
    step_times = []  # 记录每一步时间和截至该步的总时间（格式：
(step_duration, total_so_far)）
    rows, cols = grid.shape
    distances = {(i, j): float('inf') for i in range(rows) for j in
range(cols)}
    distances[start] = 0
    previous = {(i, j): None for i in range(rows) for j in range(cols)}
    # 优先队列存储 (f_score, g_score, node)，f_score = g_score + h_score
    pq = [(0, 0, start)]  # (f_score, g_score, node)
    visited = []
    explored = []  # 记录探索顺序
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    # 曼哈顿距离启发式函数
    def heuristic(node):
        return abs(node[0] - end[0]) + abs(node[1] - end[1])
    while pq:
        step_start = time.perf_counter()  # 记录每步开始时间
        _, current_g, current = heapq.heappop(pq)
        if current in set(visited):
            continue
       //计时部分，与 Dijkstra 算法实现函数该部分相同
        if current == end:
            break
        for dx, dy in directions:
            next_x, next_y = current[0] + dx, current[1] + dy
            neighbor = (next_x, next_y)
            if (0 <= next_x < rows and 0 <= next_y < cols and
                    grid[next_x, next_y] == 0 and neighbor not in
set(visited)):
                new_g = current_g + 1  # 每步成本为 1
                if new_g < distances[neighbor]:
                    distances[neighbor] = new_g
                    previous[neighbor] = current
                    h_score = heuristic(neighbor)
                    f_score = new_g + h_score
                    heapq.heappush(pq, (f_score, new_g, neighbor))
    # 重建路径部分，与 Dijkstra 算法实现函数该部分相同
```

⑶结合机器学习（Random Forest）的 A*算法启发式函数替代部分

①训练数据生成，为每个可通行节点计算到终点的真实距离

```
def generate_training_data(grid, end):
    rows, cols = grid.shape
    X = []
    y = []
```

```
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    def get_neighbors_feature(node):
        # 计算 3x3 邻域的障碍物比例
        x, y = node
        count = 0
        total = 0
        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < rows and 0 <= ny < cols:
                    count += grid[nx, ny]
                    total += 1
        return count / total if total > 0 else 0
    def astar_distance(start, end):
        distances = {start: 0}
        pq = [(0, start)]
        visited = set()
        while pq:
            dist, current = heapq.heappop(pq)
            if current in visited:
                continue
            visited.add(current)
            if current == end:
                return dist
            for dx, dy in directions:
                nx, ny = current[0] + dx, current[1] + dy
                if (0 <= nx < rows and 0 <= ny < cols and
                        grid[nx, ny] == 0 and (nx, ny) not in visited):
                    new_dist = dist + 1
                    if new_dist < distances.get((nx, ny), float('inf')):
                        distances[(nx, ny)] = new_dist
                        heapq.heappush(pq, (new_dist, (nx, ny)))
        return float('inf')
    for i in range(rows):
        for j in range(cols):
            if grid[i, j] == 0:
                node = (i, j)
                distance = astar_distance(node, end)
                if distance != float('inf'):
                    # 特征：坐标 (x, y) 和邻域障碍物比例
                    features = [i, j, get_neighbors_feature(node)]
                    X.append(features)
                    y.append(distance)
```

```
    return np.array(X), np.array(y)
# 训练随机森林模型
X_train, y_train = generate_training_data(grid, end)
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
```

②使用随机森林模型预测启发式距离

```
    def heuristic(node):
        x, y = node
        # 计算邻域障碍物比例
        count = 0
        total = 0
        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < rows and 0 <= ny < cols:
                    count += grid[nx, ny]
                    total += 1
        neighbor_feature = count / total if total > 0 else 0
        # 输入特征到模型
        features = np.array([[x, y, neighbor_feature]])
        h_score = rf_model.predict(features)[0]
        # 确保启发式值非负
        return max(0, h_score)
```

⑷结合深度学习的 A*算法启发式函数替代部分

这里的训练数据生成与上面机器学习的方法类似，这里就不再赘述。

①定义神经网络模型

```
class HeuristicNet(nn.Module):
    def __init__(self):
        super(HeuristicNet, self).__init__()
        self.fc1 = nn.Linear(4, 32)  # 输入：4 个特征（x, y, 邻域比例, 曼哈顿距离）
        self.fc2 = nn.Linear(32, 1)  # 输出：预测距离
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.3)
    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

②神经网络训练

```
def train_model(X, y):
```

```
    model = HeuristicNet()  # 创建神经网络模型实例
    criterion = nn.MSELoss()  # 使用均方误差作为损失函数
    optimizer = optim.Adam(model.parameters(), lr=0.001)  # 使用 Adam 优化
器，学习率为 0.001
    # 学习率调度器：当损失不再下降时降低学习率
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(
        optimizer,
        mode='min',  # 监控最小化指标
        factor=0.5,  # 学习率减半
        patience=10  # 等待 10 轮无改善后再调整
    )
    X_tensor = torch.FloatTensor(X)
    y_tensor = torch.FloatTensor(y).reshape(-1, 1)
    # 训练参数设置
    batch_size = 16  # 小批量训练的大小
    n_samples = X.shape[0]  # 样本总数
    indices = np.arange(n_samples)  # 用于随机打乱数据的索引
    model.train() # 设置训练模式
    epochs = 200  # 最大训练轮数
    best_loss = float('inf')  # 记录最佳损失值
    patience = 20  # 早停耐心值
    patience_counter = 0  # 早停计数器
    losses = []  # 记录每轮的损失值
    pbar = tqdm(range(epochs), desc='训练进度')
    for epoch in pbar:
        np.random.shuffle(indices)   # 随机打乱训练数据
        epoch_loss = 0
        for i in range(0, n_samples, batch_size):  # 小批量训练
            batch_indices = indices[i:i + batch_size]
            X_batch = X_tensor[batch_indices]
            y_batch = y_tensor[batch_indices]
            # 前向传播和反向传播
            optimizer.zero_grad()  # 清空梯度
            outputs = model(X_batch)  # 前向传播
            loss = criterion(outputs, y_batch)  # 计算损失
            # 添加 L1 正则化以防止过拟合
            l1_lambda = 0.001  # L1 正则化系数
            l1_norm = sum(p.abs().sum() for p in model.parameters())
            loss = loss + l1_lambda * l1_norm
            loss.backward()  # 反向传播
            optimizer.step()  # 更新参数
            epoch_loss += loss.item() * len(batch_indices) # 累加批次损失
        epoch_loss /= n_samples  # 计算平均损失
        losses.append(epoch_loss)
```

```
        scheduler.step(epoch_loss)   # 更新学习率
        # 早停机制，这里省略具体步骤
        pbar.set_postfix({'loss': f'{epoch_loss:.4f}'})
    return model
# 预计算邻域特征以加速推理
def precompute_neighbor_features(grid):
    rows, cols = grid.shape
    neighbor_features = np.zeros((rows, cols))
    for i in range(rows):
        for j in range(cols):
            count = 0
            total = 0
            for dx in [-2, -1, 0, 1, 2]:
                for dy in [-2, -1, 0, 1, 2]:
                    nx, ny = i + dx, j + dy
                    if 0 <= nx < rows and 0 <= ny < cols:
                        count += grid[nx, ny]
                        total += 1
            neighbor_features[i, j] = count / total if total > 0 else 0
    return neighbor_features
# 生成并训练模型
X_train, y_train = generate_training_data(grid, end)
model = train_model(X_train, y_train)
model.eval()
neighbor_features = precompute_neighbor_features(grid)
```

③使用神经网络模型预测启发式距离

```
    def heuristic(node):
        x, y = node
        # 使用预计算的邻域特征
        neighbor_feature = neighbor_features[x, y]
        manhattan = (abs(x - end[0]) + abs(y - end[1])) / (rows + cols)
        features = np.array([[x / rows, y / cols, neighbor_feature,
manhattan]])
        features_tensor = torch.FloatTensor(features)
        with torch.no_grad():
            h_score = model(features_tensor).item()
        # 确保启发式值非负且可接受
        h_score = max(0, h_score)
        manhattan_dist = manhattan_distance(node)
        if h_score > manhattan_dist * 1.5:  # 限制高估
            h_score = manhattan_dist
        return h_score
```

⑸动态可视化函数

该函数使用 matplotlib.animation.FuncAnimation 动态展示路径搜索过程，包括：

探索节点的动态渲染：

- 已探索节点以黄色标记；
- 最短路径上的节点以红色标记；
- 最新节点更亮，旧节点逐渐变暗，体现探索过程。

信息面板：显示当前步耗时和截至当前总耗时。

```
def animate_pathfinding(grid, start, end, path, explored, step_times,
total_time):
    # 创建两个图形窗口
    fig_vis, ax_vis = plt.subplots(figsize=(8, 8))  # 路径可视化窗口
    fig_info, ax_info = plt.subplots(figsize=(4, 3))  # 信息显示窗口
    # 设置路径可视化窗口
    ax_vis.set_title("Dijkstra 路径搜索可视化", fontsize=12)
    ax_vis.imshow(grid, cmap=plt.colormaps['Greys'],
interpolation='nearest')
    # 设置网格线
    ax_vis.grid(True, which='both', color='black', linestyle='-',
linewidth=0.5)
    ax_vis.set_xticks(np.arange(-0.5, grid.shape[1], 1))
    ax_vis.set_yticks(np.arange(-0.5, grid.shape[0], 1))
    ax_vis.set_xticklabels([])
    ax_vis.set_yticklabels([])
    # 初始化起点、终点和路径
    start_point, = ax_vis.plot(start[1], start[0], 'go', label='起点',
markersize=10)
    end_point, = ax_vis.plot(end[1], end[0], 'ro', label='终点',
markersize=10)
    path_line, = ax_vis.plot([], [], 'b-', label='最短路径',
linewidth=2)
    # 创建一个空的绘图对象列表，用于动态更新探索节点
    explored_plots = []
    # 将路径转换为集合以加速查找
    path_set = set(path)
    ax_vis.legend(loc='upper right')
    # 设置信息显示窗口
    ax_info.set_axis_off()  # 隐藏坐标轴
    info_text = ax_info.text(0.05, 0.95, '',
                             transform=ax_info.transAxes,
                             bbox=dict(facecolor='white',
                                       alpha=0.8,
                                       edgecolor='gray',
```

```
                                    boxstyle='round,pad=0.5'),
                            fontsize=10,
                            verticalalignment='top')
    # 动画更新函数
    def update(frame):
        nonlocal explored_plots
        # 清除旧的探索节点绘图对象
        for plot in explored_plots:
            plot.remove()
        explored_plots = []
        if frame < len(explored):
            # 绘制所有已探索的节点，根据是否在路径上设置颜色
            for i, node in enumerate(explored[:frame + 1]):
                alpha = max(0.2, 1.0 - (frame - i) * 0.05)  # 最新节点
alpha=1.0，较旧节点逐渐变淡
                # 如果节点在最短路径上，标为红色；否则标为黄色
                color = 'rx' if node in path_set else 'yx'
                plot, = ax_vis.plot(node[1], node[0], color,
markersize=8, alpha=alpha)
                explored_plots.append(plot)
            step_duration, total_so_far = step_times[frame]  # 获取每步时
间和截至当前的总时间
            current_text = (
                "实时搜索信息\n"
                "————————\n"
                f"当前步数: {frame+1}\n"
                f"已探索节点: {len(explored[:frame+1])}\n"
                f"当前步耗时: {step_duration:.7f}秒\n"
                f"截至当前总耗时: {total_so_far:.6f}秒\n"
            )
        elif frame == len(explored) and path:
            # 绘制最短路径
            path_x, path_y = zip(*path)
            path_line.set_data(path_y, path_x)
            # 保持所有探索节点可见，路径上的节点为红色
            for i, node in enumerate(explored):
                alpha = max(0.2, 1.0 - (len(explored) - 1 - i) * 0.05)
                color = 'rx' if node in path_set else 'yx'
                plot, = ax_vis.plot(node[1], node[0], color,
markersize=8, alpha=alpha)
                explored_plots.append(plot)

            _, final_total = step_times[-1] if step_times else (0,
total_time)
```

```
            current_text = (
                "搜索完成！\n"
                "————————\n"
                f"总步数：{len(explored)}\n"
                f"路径长度：{len(path)}\n"
                f"算法总耗时：{total_time:.6f}秒"
            )
        info_text.set_text(current_text)
        fig_info.canvas.draw()
        return [start_point, end_point, path_line] + explored_plots
    # 调整布局
    fig_vis.tight_layout()
    fig_info.tight_layout()
    # 创建动画
    frames = len(explored) + 1
    ani = FuncAnimation(fig_vis, update, frames=frames, interval=200,
                        blit=True, repeat=False)
    plt.show()
```

⑹主函数实现

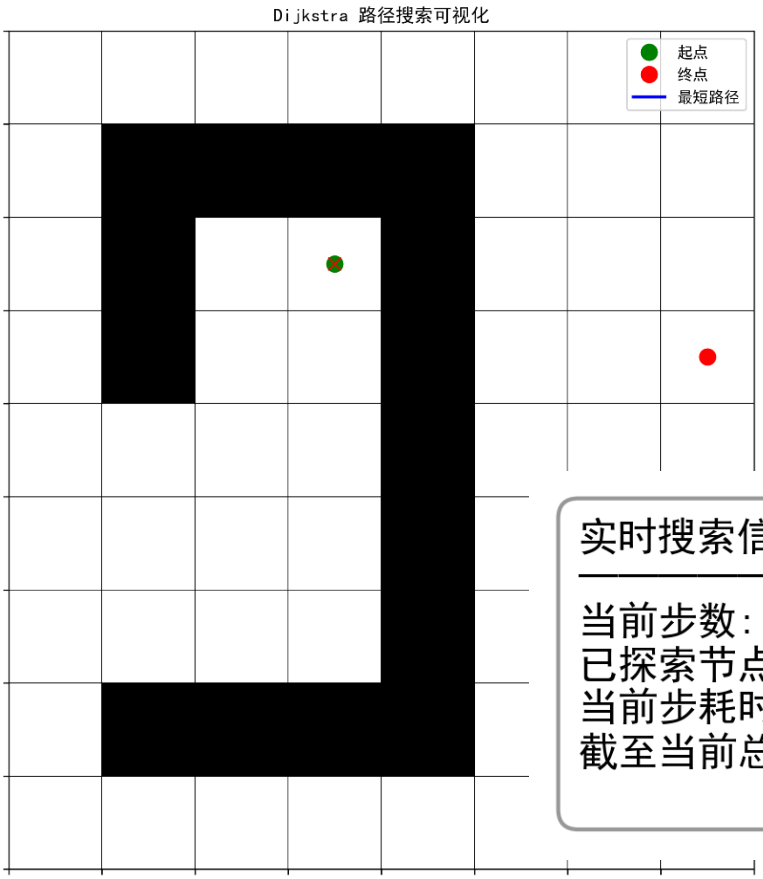
首先调用 astar_with_steps 函数，返回最短路径、探索过的节点、每步耗时以及总耗时。如果成功找到路径，就打印路径内容、路径长度、探索步数和总耗时；否则提示无法找到路径。最后调用 animate_pathfinding 函数对整个路径搜索过程进行动态可视化，帮助直观理解寻路算法的搜索策略和效率。

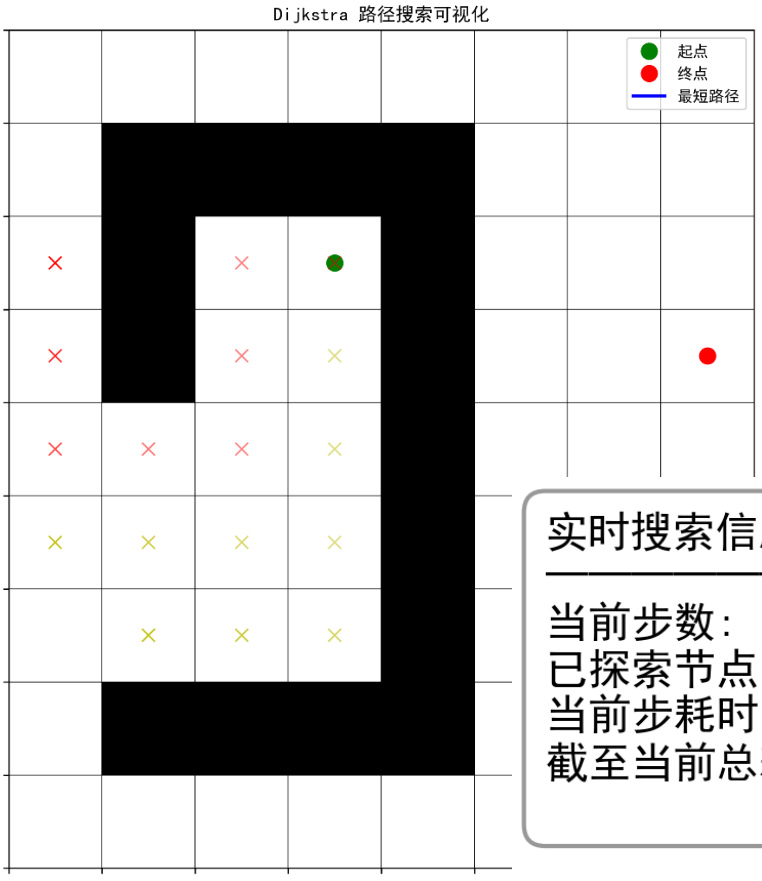
```
path, explored, step_times, total_time = astar_with_steps(grid, start,
end)
if path:
    print(f"找到的最短路径：{path}")
    print(f"路径长度：{len(path)}")
    print(f"总探索步数：{len(explored)}")
    _, final_total = step_times[-1] if step_times else (0, total_time)
    print(f"算法总耗时：{total_time:.6f}秒")
    animate_pathfinding(grid, start, end, path, explored, step_times,
total_time)
else:
    print("无法找到路径！")
```

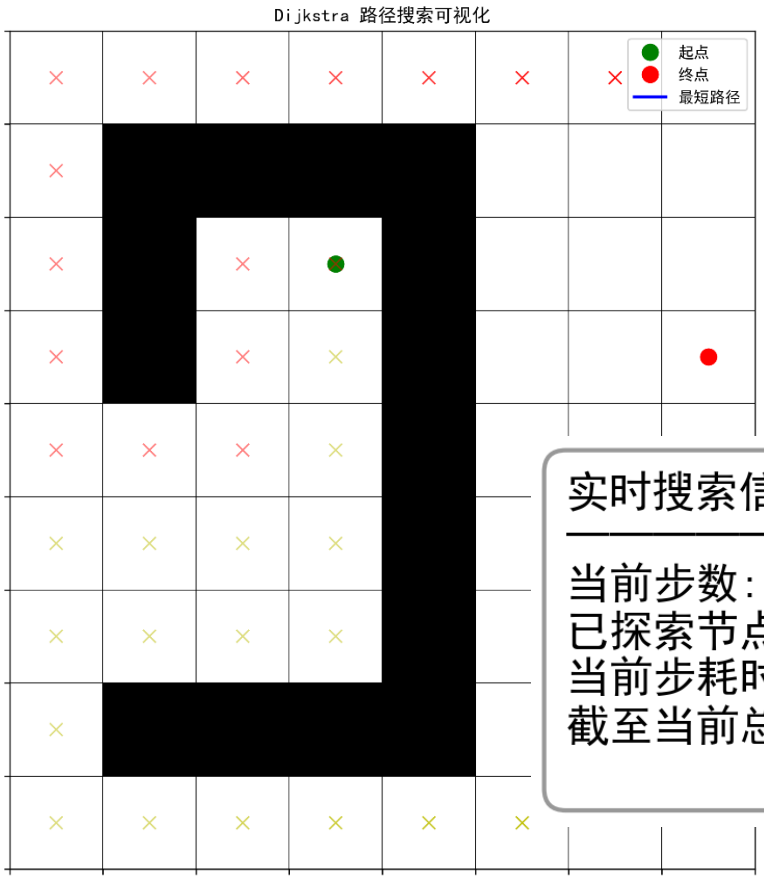
4. 实验结果

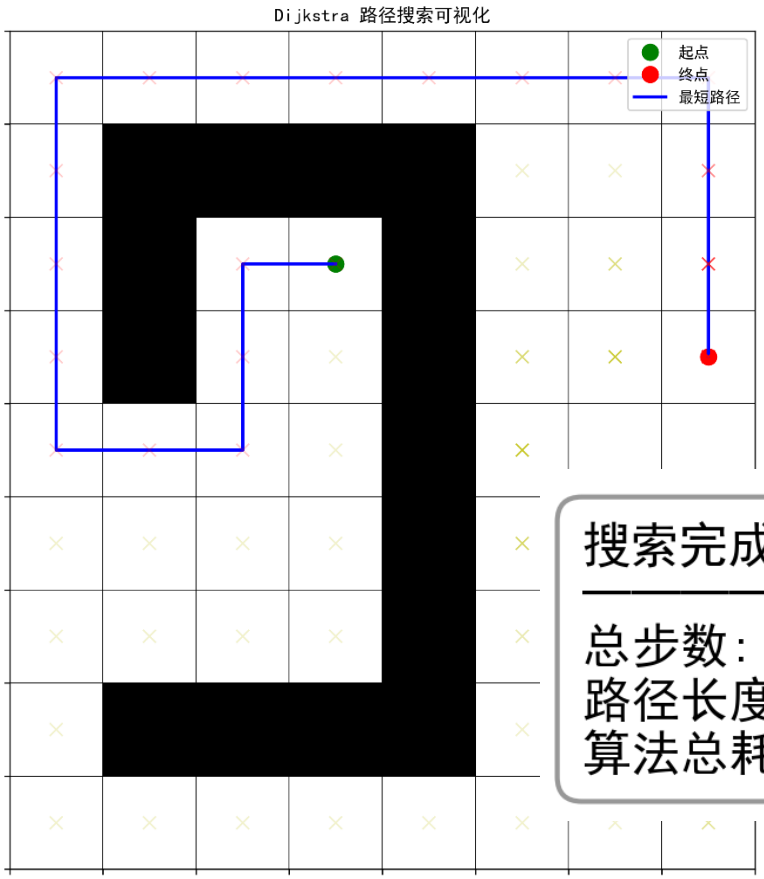
⑴Dijkstra 算法

对于第一个算法的结果展示，我会放多张图片展示动态过程，后面的算法就会只放关键图片。


Dijkstra 路径搜索可视化
起点
终点
最短路径
实时搜索信息
当前步数：1
已探索节点：1
当前步耗时：0.0000027秒
截至当前总耗时：0.000081秒

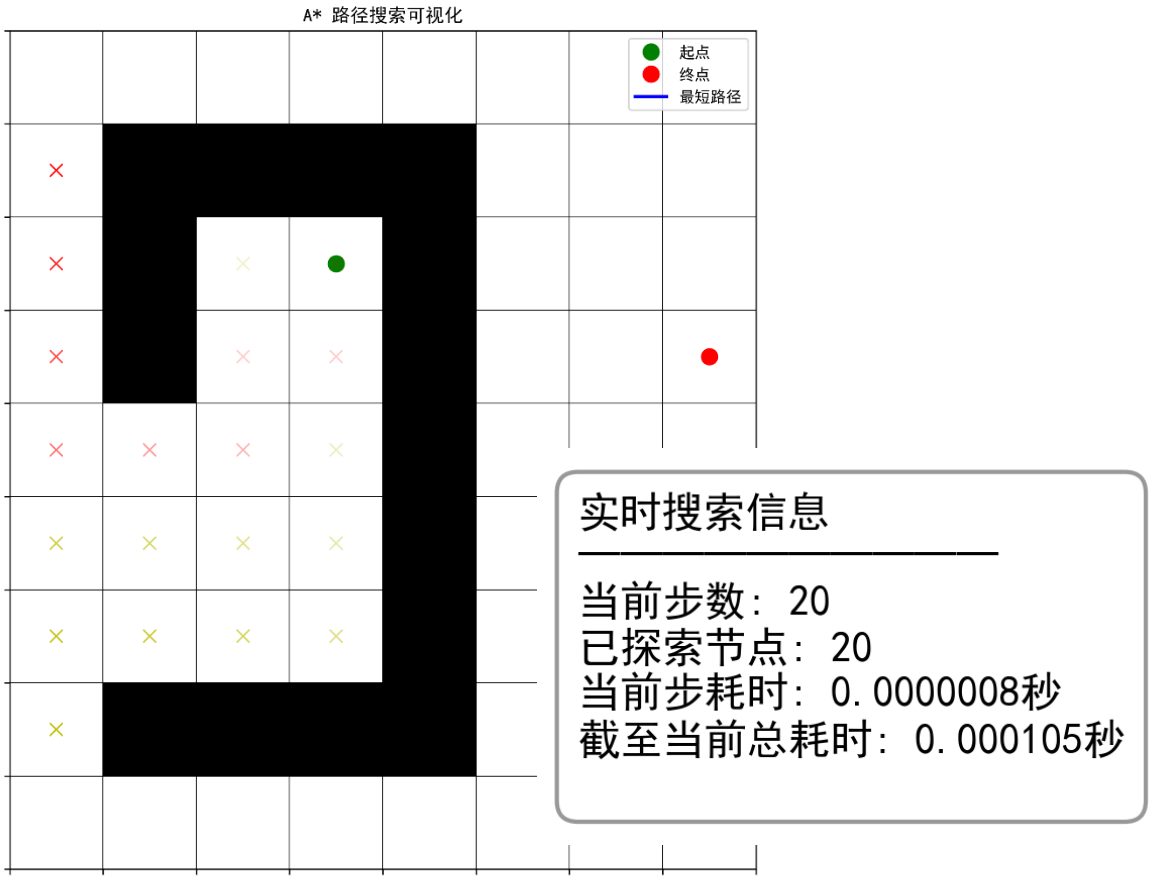

Dijkstra 路径搜索可视化
起点
终点
最短路径
实时搜索信息
当前步数：17
已探索节点：17
当前步耗时：0.0000007秒
截至当前总耗时：0.000141秒

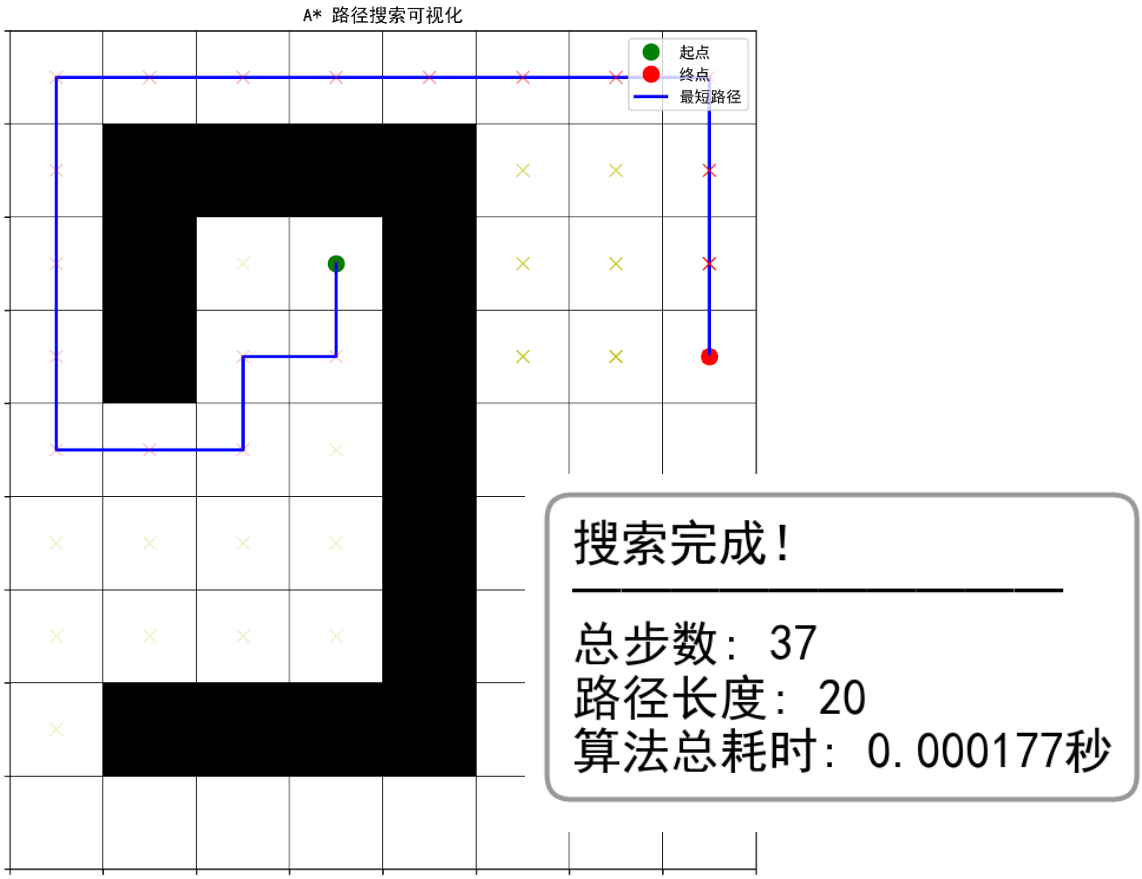

Dijkstra 路径搜索可视化
起点
终点
最短路径
实时搜索信息
当前步数：33
已探索节点：33
当前步耗时：0.0000010秒
截至当前总耗时：0.000215秒


Dijkstra 路径搜索可视化
起点
终点
最短路径
搜索完成！
总步数：54
路径长度：20
算法总耗时：0.000323秒

```
找到的最短路径: [(2, 3), (2, 2), (3, 2), (4, 2), (4, 1), (4, 0), (3, 0), (2, 0),
 (1, 0), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (1, 7),
 (2, 7), (3, 7)]
路径长度: 20
总探索步数: 54
算法总耗时: 0.000323秒
```

(2)A*算法



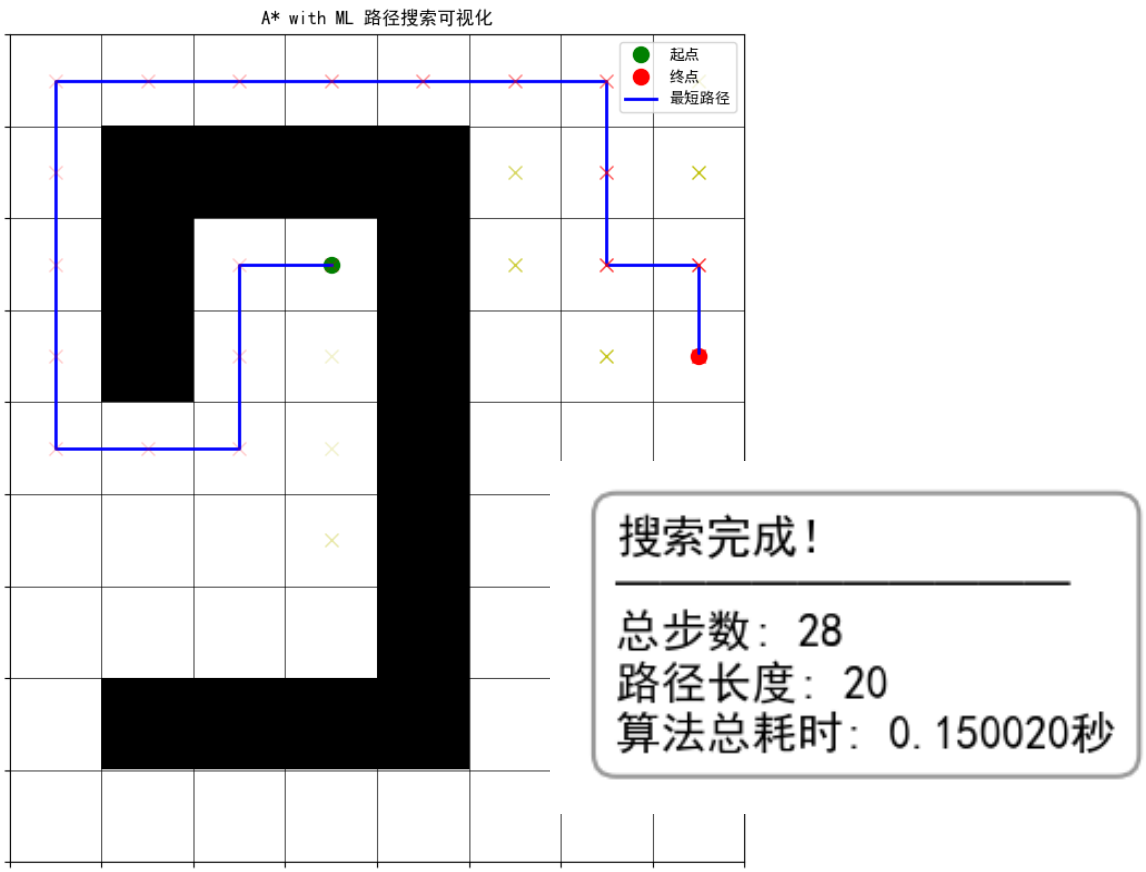



```
找到的最短路径: [(2, 3), (3, 3), (3, 2), (4, 2), (4, 1), (4, 0), (3, 0), (2, 0),
 (1, 0), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (1, 7),
 (2, 7), (3, 7)]
路径长度: 20
总探索步数: 37
算法总耗时: 0.000177秒
```

比较 Dijkstra 算法和 A*算法，发现 A*算法无论是总探索步数还是算法总耗时，都优于 Dijkstra 算法

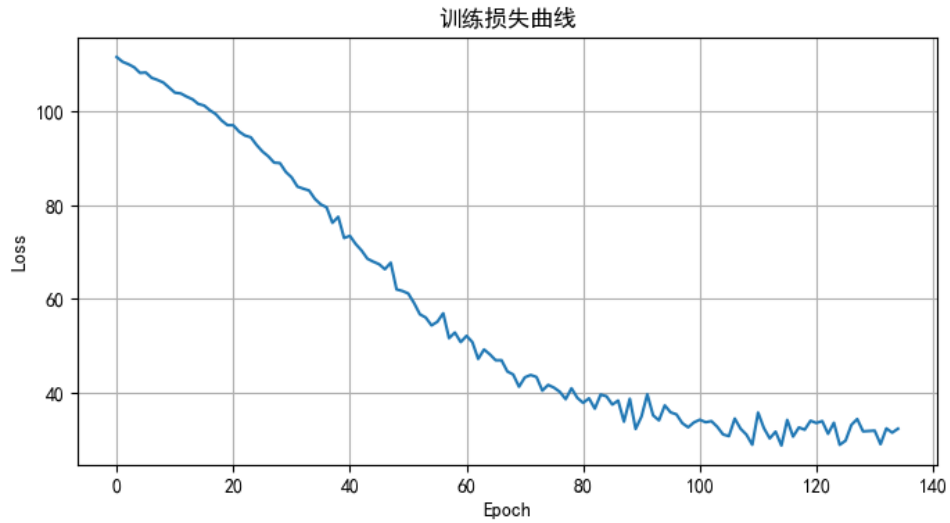
⑶结合机器学习（Random Forest）的 A*算法

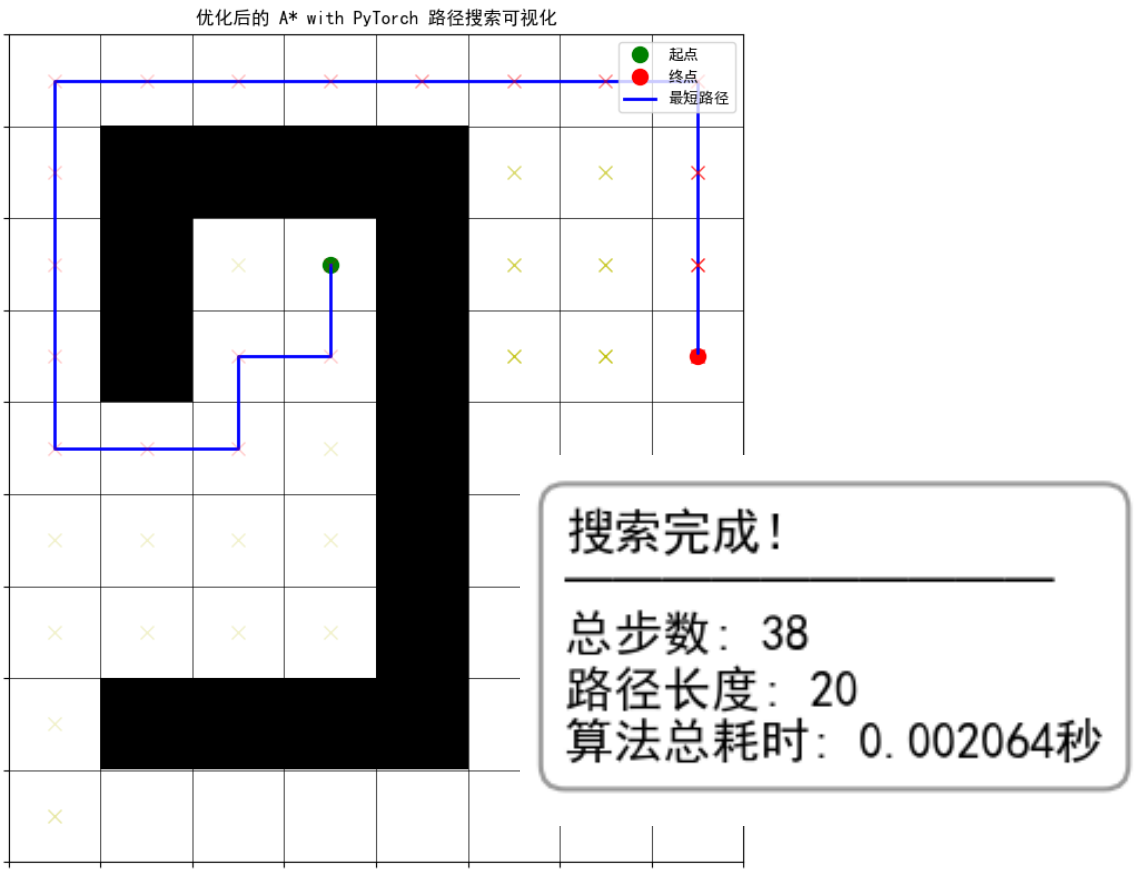
```
找到的最短路径: [(2, 3), (2, 2), (3, 2), (4, 2), (4, 1), (4, 0), (3, 0), (2, 0),
 (1, 0), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (1, 6), (2, 6),
 (2, 7), (3, 7)]
路径长度: 20
总探索步数: 28
算法总耗时: 0.150020秒
```



(4)结合深度学习的 A*算法

```
训练进度：  52%|          | 104/200 [00:00<00:00, 356.73it/s, loss=31.4717]
早停于第 135 轮
训练进度：  67%|          | 134/200 [00:00<00:00, 348.63it/s, loss=31.4717]
找到的最短路径: [(2, 3), (3, 3), (3, 2), (4, 2), (4, 1), (4, 0), (3, 0), (2, 0),
 (1, 0), (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (1, 7),
 (2, 7), (3, 7)]
路径长度: 20
总探索步数: 38
算法总耗时: 0.002064秒
```





可以看出，相较于传统 A*算法，在路径较为简单时，因为模型需要训练，使用机器学习或深度学习花费时间会更长，但总探索步数有所降低，这两种方法的优势在更复杂的地图上显示更明显，在后文会加以补充。

5. 总结与分析

(1)对算法的理解和评价

Dijkstra 是 A*的无启发式版本，探索所有可能路径找出最短路径。A*是一种带有启发式函数的最佳优先搜索算法，兼具 Dijkstra 的代价最小性和贪心搜索的方向性。引入随机森林的 A*启发式搜索利用随机森林回归训练模型预测任意节点到终点的“启发式估计距离”，预测距离相较于手工设定更细致，可降低不必要的扩展。引入深度学习的 A*启发式搜索利用神经网络学习估计某个节点到终点的代价，以替代人工设置的曼哈顿距离，使搜索更智能化、与实际地图结构更适配。

①Dijkstra 算法

a. 优点

- 代码简洁，问题描述直观，实现相对简单，逻辑清晰。
- 保证找到最短路径。

b. 缺点

- 时间复杂度较高，$O\big((V+E)\log V\big)$。
- 对于大型网格或复杂图，效率低于启发式算法（如 A*）。

②A*算法

a. 优点

- 启发式函数减少了无关节点的探索。
- 保证找到最短路径。

b. 缺点

- 实现稍复杂，需设计合适的启发式函数。

③使用机器学习或深度学习计算启发式函数

a. 优点

- 适应性强，可通过增加数据和调整结构提升性能。
- 能捕捉复杂网格中的非线性模式。

b. 缺点

- 不保证找到最短路径。
- 训练时间长，需大量标注数据。

⑵调试中遇到的问题

①使用随机森林模型计算启发式函数增加对曼哈顿距离的考虑使模型性能变差且出错

随机森林可以通过学习坐标之间的差值自动推导出类似曼哈顿距离的信息，一开始使用随机森林模型没有添加曼哈顿距离作为特征，在尝试改进模型，添加曼哈顿距离作为特征后，随机森林模型的性能变差。A*算法探索的节点数增加，且路径不正确。推测原因是显式添加曼哈顿距离可能引入冗余，导致模型过拟合或权重分配不合理。如果模型预测的$h(n)$过高（高于真实距离），A*会失去最优性，可能找到非最短路径或探索过多节点。

```
# 特征：坐标 (x, y) 和邻域障碍物比例，以及新加的曼哈顿距离
                    manhattan = abs(i - end[0]) + abs(j - end[1])
                    features = [i, j, get_neighbors_feature(node),
manhattan]
                    X.append(features)
```

```
找到的最短路径: [(2, 3), (2, 2), (3, 2), (4, 2), (5, 2), (6, 2), (6, 1), (6, 0),
 (7, 0), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (7, 5), (6, 5), (5, 5),
 (4, 5), (3, 5), (3, 6), (3, 7)]
路径长度: 22
总探索步数: 46
算法总耗时: 0.211751秒
```

在删去曼哈顿距离这个特征后，模型又恢复最优性。

②神经网络模型过拟合

一开始在实现深度学习模型时，训练后模型出现过拟合，泛化能力差，使总探索步数和算法用时较大。为了解决这个问题，我首先应用 dropout、L1 正则化等防过拟和，然后采用监控训练损失与验证损失，设置早停机制，不但减少训练时间，还可以防止过拟合。

在模型定义中使用 dropout：

```
    def __init__(self):
        super(HeuristicNet, self).__init__()
        self.fc1 = nn.Linear(4, 32)  # 输入：4 个特征（x, y, 邻域比例, 曼哈顿距离）
        self.fc2 = nn.Linear(32, 1)  # 输出：预测距离
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.3)
```

在模型训练中使用 F1 正则化：

```
            # 添加 L1 正则化以防止过拟合
            l1_lambda = 0.001  # L1 正则化系数
            l1_norm = sum(p.abs().sum() for p in model.parameters())
            loss = loss + l1_lambda * l1_norm
```

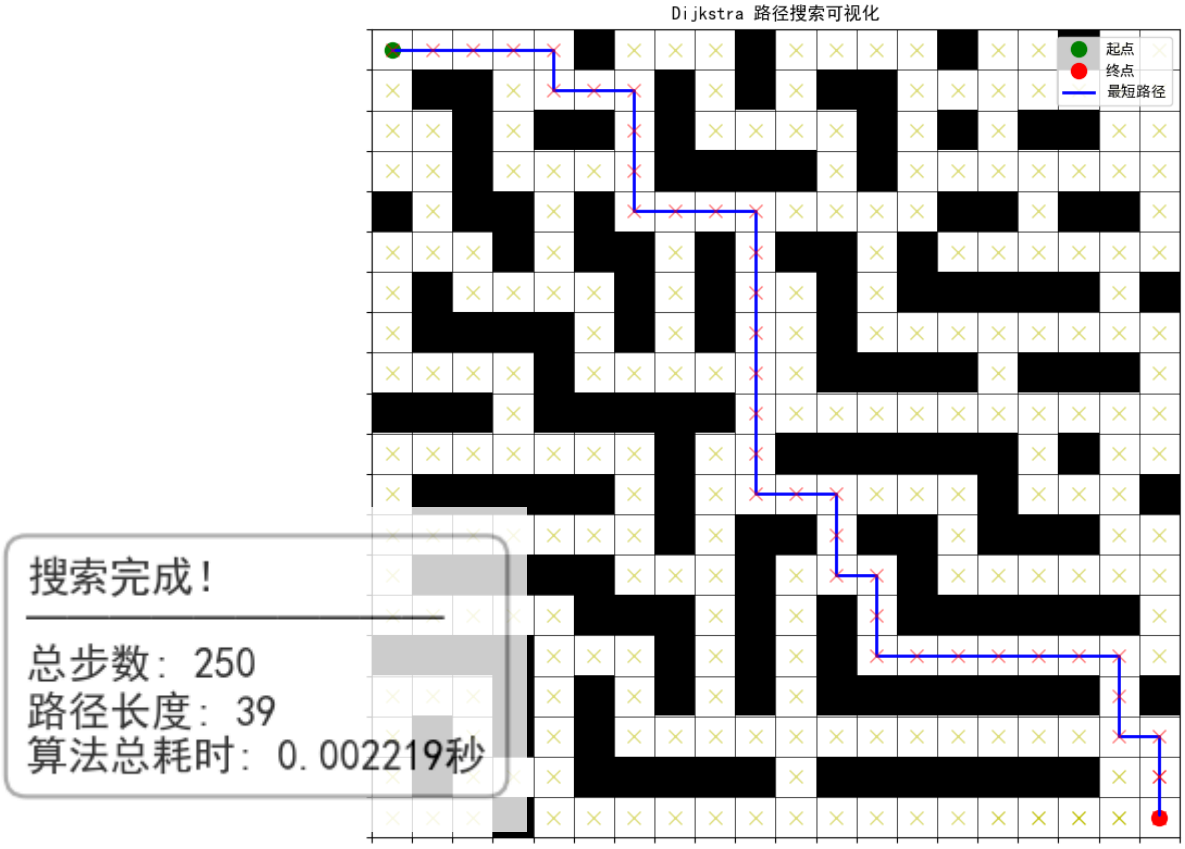
在模型训练中使用早停机制：

```
        # 早停检查
        if epoch_loss < best_loss:
            best_loss = epoch_loss
            patience_counter = 0
        else:
            patience_counter += 1
        if patience_counter >= patience:
            print(f"早停于第 {epoch + 1} 轮")
            break
```

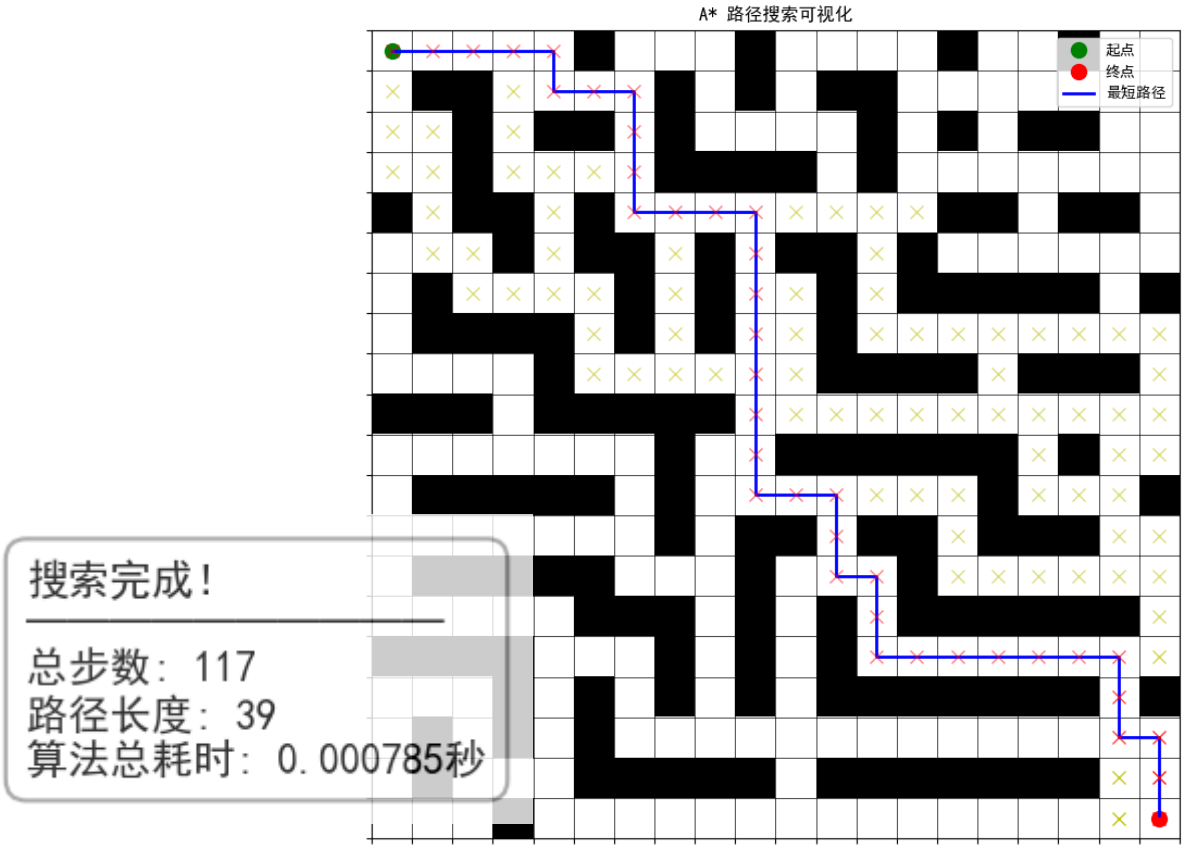
⑶对其他更复杂场景图的尝试

在完成对题设要求的场景图的实验后，为了验证模型性能，我有设计了一个 20×20 的场景图，并运用上述 4 个算法对该地图进行寻路实验，结果如下。

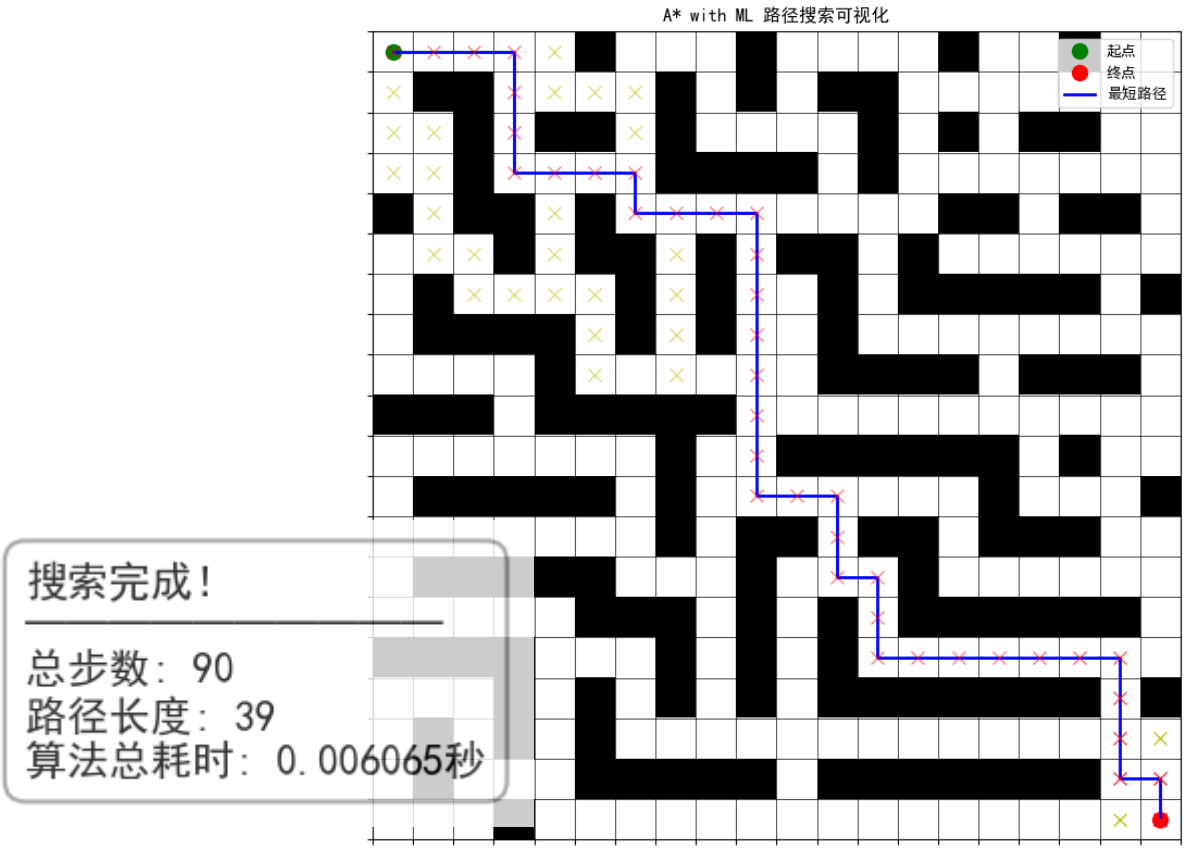
①Dijkstra 算法



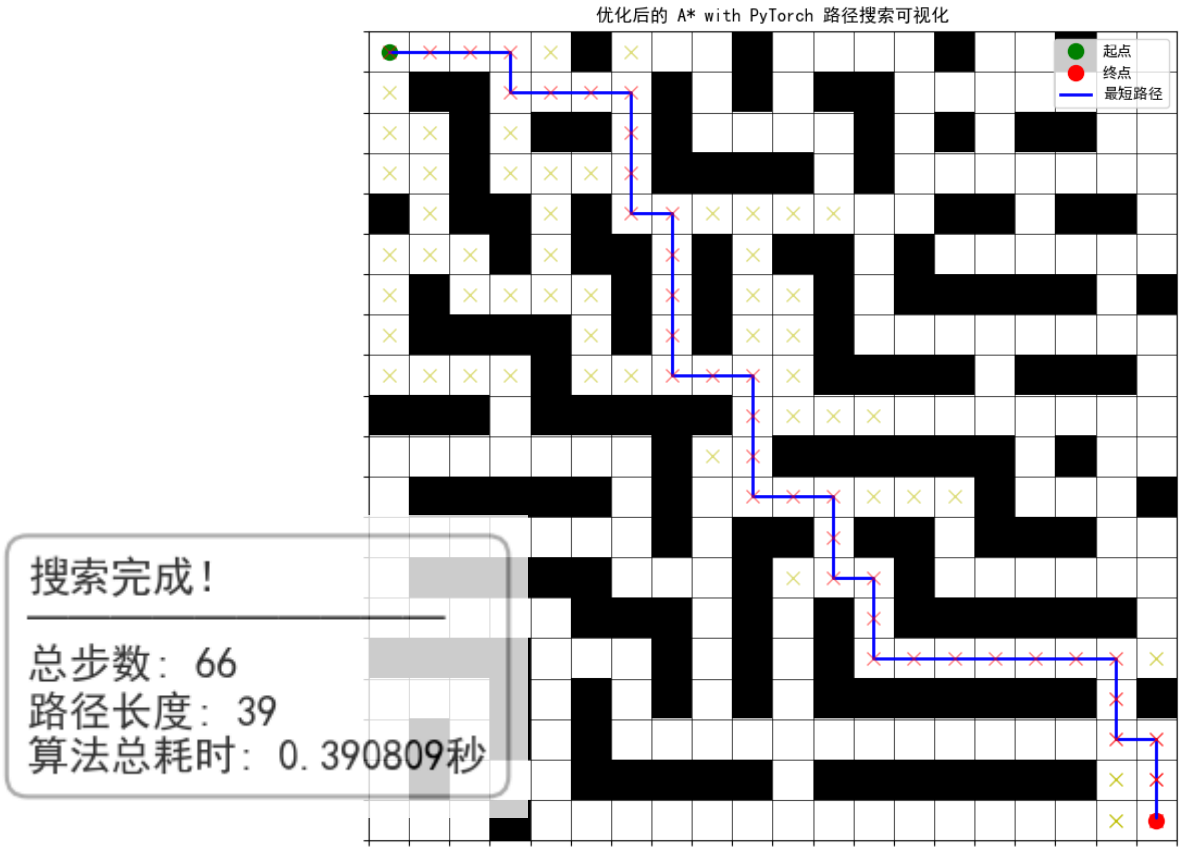
②A*算法



③结合机器学习（Random Forest）的 A*算法



④结合深度学习的 A*算法



可以看出，在面对较复杂的场景图时，相较于 Dijkstra 算法，A*算法在效率上存在较大优势。而且较复杂的场景图更加突出了机器学习和深度学习的优点，使用机器学习或深度学习的总探索步数降低幅度大，尤其是到寻路后期，寻路路径几乎与所求最优路径重合。可以推测，在面临更大且更复杂的场景图时，使用机器学习或深度学习的优势会更加明显。

⑷实验收获

这次实验让我深入理解了路径规划算法，分别有 Dijkstra 算法（最基础的搜索算法，总能找到最短路径但搜索范围大）、A*算法（在 Dijkstra 基础上增加启发式函数，提高搜索效率）。并通过多种方式计算启发式函数，使用曼哈顿距离或使用神经网络/随机森林预测。同时我还学习了 Sklearn 等算法库和 Pytorch 等深度学习框架，提高了算法能力。

我的算法设计思维也得到了提升，例如，传统算法与机器学习结合，模块化设计提高代码复用，算法性能与实现效率平衡等。同时，我也增强了可视化技巧：动态展示搜索过程，实时更新搜索信息，将路径与探索区域区分显示，依靠渐变效果展示搜索顺序，这些方法使我的最短路径图搜索过程更加清晰。

这次实验让我深入理解了传统算法与机器学习的结合方式，算法优化的多个层面，可视化在算法理解中的作用，以及工程实践中的各种考虑因素，这些经验对今后的算法设计和工程实践都有重要参考价值。