**Q1:使用深度优先搜索（DFS）寻找固定食物点：**

首先测试 SearchAgent 是否正常：

结果如下：可成功走出迷宫；

```
(ML) duqiu@duqiudeMBP task_5 % python pacman.py -l tinyMaze -p SearchAgent -a fn
=tinyMazeSearch
[SearchAgent] using function tinyMazeSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 0
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```

在运行测试后，会发现探索顺序符合预期，但是 Pacman 不一定会走过所有探索的方格

（1）为何**探索顺序**会符合预期？

因为 DFS 的核心是“优先深入，再回溯”（利用栈的后进先出特性）：总是先把当前结点的“后继结点”压入栈，然后优先探索最新压入栈的节点（即更深层的分支）

反映到游戏界面上，越早探索的方格颜色越亮，而 DFS 的探索顺序会呈现出：沿着某条分支深入，直到死胡同，再回溯到上一个节点继续深入其余分支，这种“深入-回溯-再深入”节奏与 DFS 的逻辑是一致的，所以探索顺序符合预期。

（2）为何 Pacman 不一定会走过所有探索的方格：

因为 DFS 探索阶段是算法在找路径时候，会对地图节点的遍历，而 Pacman 的“移动路径”是算法找到的“从起点到终点的可行路径”。

就比如 Pacman 不会去走算法探索过的死胡同里的方格。

运行后的结果截图：

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
```

（3）DFS 在 mediumMaze 的解路径长度为 130，是否最优？

DFS 的路径通常不是最优的，因为 DFS 的核心逻辑是“**优先深入，再回溯**”，它追求的是**快速探索深层节点**，并非寻找最短路径。

DFS 可能会沿着一条很深但绕远的分支一直探索，直到碰到死胡同才回溯，最终找到的路径是“能到达终点，但步数不一定最少”的路径

（4）DFS 出错与否？

我倾向于 DFS 本身没有“错误”，但它的设计目标与“找到最短路径”不匹配：

DFS 是深度优先，更适合需要快速搜索深层结构，不在乎路径长短的场景

**Q2:使用广度优先搜索（BFS）：**

编写代码，运行测试后得到如下结果：

```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

同时运行八数码问题得到：

```
After 8 moves: left
-------------
| 1 |   | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
After 9 moves: left
-------------
|   | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
```

最终的结论是 BFS 可以找到代价最小解。

**Q3:统一代价搜索（UCS）实现：**

根据任务要求，我们需要在 search.py 中实现 uniformCostSearch 函数，处理带不同代价的路径搜索。UCS 的核心是每次扩展总代价最小的节点，适用于边权不同的场景。

（1）UCS 核心逻辑：

使用优先队列（最小堆），用 util.PriorityQueue 存储待探索结点，按总代价为优先级排序；

维护 cost_so_far 字典，记录每个状态的最小总代价，避免重复处理高代价路径；

每个结点存储的是总代价，当前状态，到达该状态的动作序列；

（2）关键细节解析：

使用优先队列：util.PriorityQueue 的 push 方法第二个参数是优先级，确保队列按照代价升序排列，每个队列元素是（总代价，当前状态，路径），便于回溯路径与计算代价

通过 cost_so_far 字典，仅当新路径的总代价严格小雨已记录代价时，才更新并入队列。
运行结果展示：

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
Path found with total cost of 68719479864 in 0.0 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:        418.0
Win Rate:      1/1 (1.00)
Record:        Win
```

（3）结果分析：
指数函数的增长/衰减是爆炸式的，若向东走的代价是$2^n$，那么向西走的代价是$\left(\frac{1}{2}\right)^n$
那么在路径中多走几个方向，代价会急剧放大或者缩小。
这两个智能体（StayEast/StayWest）的测试场景，是为了验证 UCS 在极端代价分布下的能力，即使代价是指数变换的，UCS 仍能通过“优先扩展总代价最小的节点”，找到该场景下总代价最优的路径。
而运行结果中出现：总代价从 1 到数十亿的波动，正是指数代价的“偏向性”；
总之这种代价差异极大是预期内的，因为场景使用了指数代价函数来测试 UCS 处理极端代价的能力。

**Q4:A*搜索：**
要解决 A*搜索算法实现，需要结合已经实现的统一代价搜索（UCS）思路，并引入启发式函数来优化搜索效率。A*算法的核心是综合考虑从起点到当前结点的实际代价（g 值）和从当前结点到目标结点的估计代价（h 值），来选择下一个扩展的结点，即优先扩展 f 值最小的结点（f = g + h）
实现步骤：
（1）初始化优先队列：存储结点信息（从起点到该状态的实际 g 值，当前状态，到达该状态的路径），队列优先级由 f 值（g+启发函数 h 估计值）
（2）记录最小代价：使用字典记录每个状态的最小 g 值，避免重复扩展高代价路径
（3）扩展结点：从队列中弹出 f 值最小的结点，若为目标状态则返回路径；否则扩展其所有后续结点，计算新的 g 值和 h 值，若发现更优路径则更新并入路径。
代码执行结果：

```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
(ML) duqiu@duqiudeMBP task_5 %
```

在 openMaze 上，不同搜索策略的表现：

**（1）深度优先搜索（DFS）：**

不保证最优，DFS 优先“深入探索”，不考虑路径代价，容易绕弯路，最终路径总代价通常不是最小的

**（2）广度优先搜索（BFS）：**

若 openMaze 是无权图，BFS 能保证找到步数最少（总代价最少）的路径，但是扩展结点数会多于 A*，因为 BFS **无目标方向的引导**，会均匀扩展所有等距离的结点。

**（3）统一代价搜索（UCS）：**

能保证找到总代价最小的路径，但是扩展结点数目会多于 A*，因为 UCS 只依据实际代价搜索，缺乏目标方向的导向。

**（4）A*搜索（带曼哈顿启发式）：**

能保证找到总代价最小的路径，因此 A*不会错过最优解。

扩展结点数最少，因为曼哈顿启发式会引导搜索向目标方向移动。

**Q5：利用 PPO+神经网络**

（1）实验目的：

实现 PPO 算法的训练流程（数据收集，策略更新，模型保存）；

完成游戏状态预处理与神经网络推理集成；

验证智能体是否能学会“主动吃食物，尝试躲避幽灵”的核心逻辑；

排查并解决训练/推理过程中的工程问题；

（2）核心文件与功能：

ppo_train.py：PPO 算法实现，包括策略网络，价值网络，轨迹收集，GAE 优势计算，模型训练与保存；

searchAgents.py:集成 PPO 推理逻辑，补全搜索问题；

common.py：游戏状态预处理；

pacman.py：游戏入口，调用智能体运行并输出结果

（3）实验步骤：

先补全 searchAgents.py：

补全 CornersProblem；

实现 foodHeuristic，基于当前到最近食物的距离+食物 MST 长度，设计可采纳启发式；

新增 PPOPacmanAgent 类，加载 PPO 模型，实现推理时的动作选择；

将 commom.py 文件里的 preprocess_state 方法进行了完善；

对 PPO 进行训练：

网络结构分为策略网络+价值网络：

```python
class PolicyNetwork(nn.Module):
    """策略网络：输入状态，输出动作概率分布（4个动作：上下左右）"""
    def __init__(self, input_dim: int, action_dim: int = 4):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 64)  # 隐藏层1
        self.fc2 = nn.Linear(64, 64)         # 隐藏层2
        self.fc3 = nn.Linear(64, action_dim) # 输出层（4个动作）

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """前向传播：返回动作概率（softmax归一化）"""
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.softmax(self.fc3(x), dim=-1)  # 概率分布
        return x
```

```python
class ValueNetwork(nn.Module):
    """价值网络：输入状态，输出状态价值（标量）"""
    def __init__(self, input_dim: int):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 1)  # 输出标量价值

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """前向传播：返回状态价值"""
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)  # 线性输出（无激活，价值可正可负）
        return x
```

对奖励函数进行了调整，原本惩罚过重，导致模型不能有较好的训练效果，

```python
def _calculate_reward(self, prev_state, curr_state):
    reward = curr_state.getScore() - prev_state.getScore()  # 基础分数
    reward -= 0.1  # 每步惩罚（从-1.0下调）
    reward += 500 if curr_state.isWin() else -500 if curr_state.isLose() else 0  # 输赢奖励
    return reward
```

模型参数：

总轮次 300，学习率 3e-4，批次大小 64，GAE 系数 0.95；

（4）实验结果：

```
✅ PPO模型加载成功！路径： ppo_pacman_model_ep40.pth
Pacman died! Score: -439
Average Score: -439.0
Scores:        -439.0
Win Rate:      0/1 (0.00)
Record:        Loss
(ML) duqiu@duqiudeMacBook-Pro task_5 %
```

视频在另一个单独部分；

（5）实验不足与优化方向：

未实现通关：可增加训练轮次至 500-1000 一轮，也可优化网络结构；

策略不够灵活：可采用衰减学习率；

推理速度较慢，可适当简化状态特征；