Part 1. 搜索

介绍

在这个项目中，你的吃豆人智能体将在他的迷宫世界中寻找路径，包括到达一个特定的位置，以及有效地收集食物。您需要构建通用搜索算法，并将其应用于吃豆人场景。

注意，该系列文件用于北大教学，请不要上传至互联网。

环境配置

本项目依赖python 3.10版本，可以使用以下两种方式进行安装

通过Anaconda

可以前往anaconda官网下载并安装anaconda，随后在终端中创建环境并测试：

```
conda create -n py310 python=3.10
conda activate py310
python pacman.py
```

如果弹出游戏窗口，则表明环境安装成功。

通过Python官网

进入python官网，根据你的系统选择相应的安装包。

你需要编辑的文件

文件	作用
search.py	包含所有的搜索算法
searchAgents.py	包含所有基于搜索的智能体

你或许需要查看的文件

文件	作用
pacman.py	运行吃豆人游戏的主文件。这个文件描述了一个Pacman GameState类型
game.py	这是吃豆人游戏运行的逻辑。该文件描述了几个支持类型，如AgentState、Agent、Direction和Grid
util.py	用于实现搜索算法的有用数据结构

你无需关心的文件

文件	作用
graphicsDisplay.py	Graphics for Pacman
graphicsUtils.py	Support for Pacman graphics
textDisplay.py	ASCII graphics for Pacman
ghostAgents.py	Agents to control ghosts
keyboardAgents.py	Keyboard interfaces to control Pacman
layout.py	Code for reading layout files and storing their contents
autograder.py	Project autograder
testParser.py	Parses autograder test and solution files
testClasses.py	General autograding test classes
test_cases/	Directory containing the test cases for each question
searchTestClasses.py	Project 1 specific autograding test classes

代码提交与评分

你只需要填写`search.py`和`searchAgents.py`的部分内容。
最终，**你只需要打包提交这两个文件**。千万注意，不要修改文件内其他部分，否则可能无法正常评分。

我们使用`autograder.py`对你的提交进行评分，但包含的测试用例与本地给出的样例文件不同。你可以在本地运行评分器对你的代码进行评测，但仅限于帮助调试代码，该分数不等于最终分数。

学术诚信

我们会将你的代码与课堂上其他提交的代码进行逻辑查重。如果你拷贝了别人的代码，并做一些微小的修改，我们会很容易发现，请不要尝试。我们相信你们会独立完成作业。

熟悉吃豆人

吃豆人生活在一个明亮的蓝色世界里，有蜿蜒的走廊和美味的圆形食物。在这个世界高效地游走将是吃豆人掌握自己领域的第一步。

`searchAgents.py`中包含了一个最简单的智能体称为GoWestAgent，它在吃豆人世界中无脑地向西走，偶尔可以赢。运行指令如下：

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

注意：

`--layout`参数指定了一个游戏地形，`testMaze`是一个笔直的形状。

`--pacman`参数指定了运行何种智能体，`GoWestAgent`即最简单的智能体。

如果地形需要智能体转向，那么`GoWestAgent`就会卡住，例如：

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

如果智能体卡住，你可以使用组合键`Ctrl + C`退出程序。

此外，`pacman.py`支持很多可选参数，你可以列出它所支持的参数，通过运行：

```
python pacman.py -h
```

解决问题

熟悉完吃豆人世界之后，你将需要解决4个问题，每个问题具有不同的分值。
在`searchAgents.py`文件中，你可以找到`SearchAgent`这个类，该类负责在吃豆人世界中规划一条路径，并一步一步地执行，它的框架已经被完整地实现了，无需你进行修改。但是规划过程中所需要的**搜索算法**没有被实现，这正是你要做的事情。请你在`search.py`中实现以下4种搜索算法。

在此之前，你可以运行以下命令测试`SearchAgent`是否能够正常运行：

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

所有搜索函数都需要返回将代理从起点引导到目标的action列表。这些action都必须是合法的（有效的方向，不能穿墙而过）。确保使用`util.py`中提供的Stack、Queue和PriorityQueue数据结构！这些数据结构实现具有与自动评分器兼容所需的特定属性。

例题：广度优先算法

目标：使用广度优先算法（BFS）找到一个固定的食物点。
填充`search.py`中有关广度优先搜索的部分：

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    # 构建一个队列
    Frontier = util.Queue()
    # 创建已访问节点集合
    Visited = []
    # 将(初始节点,空动作序列)入队
    Frontier.push( (problem.getStartState(), []) )
    # 将初始节点标记为已访问节点
    Visited.append( problem.getStartState() )
    # 判断队列非空
    while Frontier.isEmpty() == 0:
        # 从队列中弹出一个状态和动作序列
```

```
        state, actions = Frontier.pop()
        # 判断是否为目标状态，若是则返回到达该状态的累计动作序列
        if problem.isGoalState(state):
            return actions
        # 遍历所有后继状态
        for next in problem.getSuccessors(state):
            # 新的后继状态
            n_state = next[0]
            # 新的action
            n_direction = next[1]
            # 若该状态没有访问过
            if n_state not in Visited:
                # 计算到该状态的动作序列，入队
                Frontier.push( (n_state, actions + [n_direction]) )
                Visited.append( n_state )
```

注意，编写图搜索算法时避免扩展任何已经访问过的状态。测试代码如下：

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

使用评分器验证：

```
python autograder.py --student-code search.py,searchAgents.py -q q0
```

函数解释

函数名	参数	返回值	函数介绍
problem.getStartState()	None	int	返回开始位置
problem.isGoalState()	state	bool	判断位置是否为终点
problem.getSuccessors()	state	List[(dst, action, cost)]	返回状态state的后继状态（边）列表，每条边包含下一个状态dst，动作action，边权cost
problem.getCostOfActions()	List[action]	int	返回执行一段行为序列的所需的代价，序列必须使智能体合法移动

在这里state是一个元组 `(int, int)`

题目1：深度优先算法（3分）

目标：使用深度优先算法（DFS）找到一个固定的食物点。

填充`search.py`中有关深度优先搜索的部分：

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print( "Start:", problem.getStartState() )
    print( "Is the start a goal?", problem.isGoalState(problem.getStartState()) )
    print( "Start's successors:", problem.getSuccessors(problem.getStartState()) )
    """
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```

要使算法完整，请编写DFS的图搜索版本，这样可以避免扩展任何已经访问过的状态。

一旦你编写完，运行以下命令，验证你的代码：

```
python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

使用评分器验证：

```
python autograder.py --student-code search.py,searchAgents.py -q q1
```

提示：如果你使用Stack作为你的数据结构，你的DFS算法为mediumMaze找到的解决方案的长度应该是130（前提是你把后继节点按get提供的顺序推到fringe；如果你把它们按相反的顺序推，你可能会得到246）。

题目2：一致代价搜索（3分）

虽然BFS会找到通往目标的最少行动路径，但我们可能想要找到其他意义上的“最佳”路径。考虑一下mediumDottedMaze和mediumScaryMaze这两种游戏地形。

通过改变成本函数，我们可以鼓励吃豆人寻找不同的路径。例如，我们可以对幽灵出没地区的危险步骤收取更高的代价，或对食物丰富地区的步骤收取更低的代价，理性的吃豆人代理应该相应地调整其行为。

请在`search.py`中的`uniformCostSearch`函数中实现一致代价图搜索算法。我们鼓励你在`util.py`查找一些可能在你的实现中有用的数据结构。现在，你应该在以下三个游戏地形中观察到成功的行为，其中下面的智能体都是采用UCS算法的智能体，仅在它们使用的成本函数上有所不同（智能体和成本函数已经为你写好了）：

填充`search.py`中有关一致代价搜索的部分：

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```

完成算法后，使用如下指令进行测试：

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

使用评分器验证：

```
python autograder.py --student-code search.py,searchAgents.py -q q2
```

题目3：A*算法（3分）

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```

A*接收一个启发式函数作为参数。这里启发式函数的函数接收两个参数:搜索问题中的状态（主要参数）和问题本身（用于访问一些信息）。`search.py`中的`nullHeuristic`是一个平凡的启发式函数例子。

你可以使用Manhattan distance heuristic（在`searchAgents.py`中已经实现为Manhattan heuristic）在最初的问题上测试你的A*实现，即通过迷宫找到到达固定位置的路径。

完成算法后，使用如下指令进行测试：

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic
```

使用评分器验证：

```
python autograder.py --student-code search.py,searchAgents.py -q q3
```

题目4：启发函数设计（3分）

请确保在完成这道题之前你已经完成题目3的编写。

请为CornersProblem实现一个非平凡一致性启发函数，

即完成如下代码部分（你可在`searchAgents.py`文件中找到）

```python
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible (as well as consistent).
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
    node = state[0]
    Visited_Corners = state[1]
    h_sum = 0

    "*** YOUR CODE HERE ***"
    return h_sum # Default to trivial solution
```

完成算法后，你可以运行如下代码进行验证：

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

使用评分器验证：

```
python autograder.py --student-code search.py,searchAgents.py -q q4
```

注意`AStarCornersAgent`是`-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic`的缩写。

Admissibility vs. Consistency:

启发函数以状态为参数，返回其到目标状态的所需代价的估计值。更有效的启发函数将返回与真实值接近的值。为了使启发函数的值是admissible的，启发函数值必须是到最近目标的实际最短路径代价的下界（并且是非负的）。为了保持consistency，它必须满足，如果一个行动的成本是c，那么采取该行动只能导致启发函数值最多减少c。

Admissibility并不足以保证图搜索的正确性——你需要更强的consistency条件。不过，admissible启发函数通常也是consistent。因此，通常最简单的方法是通过头脑风暴来设计admissible启发函数。一旦你有了一个工作良好的admissible启发函数，你也可以检查它是否满足consistency。通过验证对于展开的每个节点，其后续节点的f值是否满足等于或更大。此外，如果UCS和A*返回不同长度的路径，则启发函数是不一致的。

非平凡启发函数:

平凡的启发函数是那些处处返回0的函数（如UCS算法）和表示真实完成成本的启发式。前者不会节省任何时间，而后者会被评分器判定为超时。你需要一个减少总计算时间的启发函数，不过对于这个题目，自动评分器除了判断是否超过时间限制外，只检查节点数。

评分

你的启发函数必须是非平凡的非负的一致的启发函数才能得到所有分数。
确保你的启发函数给每个目标状态评估的值都返回0，并且永远不会返回负值。