Part 2.

目录

介绍

在这个项目中，你将为经典版本的吃豆人设计Agent。在此过程中，你将实现minimax搜索，alpha-beta以及蒙特卡洛树搜索。

代码库与前一个项目相比变化不大，但请从全新的安装开始，而不是将项目1的文件混在一起。

与项目1一样，本项目包括一个自动评分器，供你在机器上对答案进行评分。

用以下指令可以对所有题目进行评测。

```
python autograder.py
```

你也可以通过`-q`参数指定对题目q2进行评测：

```
python autograder.py -q q2
```

你也可以通过`-t`参数指定使用的测试用例：

```
python autograder.py -t test_cases/q2/0-small-tree
```

你可以通过使用`--graphics`标志强制显示图形界面，或者通过使用`--no-graphics`标志强制不显示图形界面。

这个项目的代码包含以下文件:

你需要修改的文件:

`multiAgents.py` 你将在这里实现三种算法。

你需要阅读的文件:

`pacman.py` 运行Pacman游戏的主要文件。这个文件还描述了一个Pacman `GameState`类型。

`game.py` 吃豆人世界运作的逻辑。这个文件描述了几个支持类型，如AgentState、Agent、Direction和Grid。

`util.py` 实现算法时的可能会用到的数据结构。

你可以忽略的文件:

`graphicsDisplay.py` Pacman图像显示界面

`graphicsUtils.py` 支持图像

`textDisplay.py` Pacman的ASCII编码

`ghostAgents.py` Ghosts程序

`keyboardAgents.py` 键盘接口

`layout.py` 导入地图的程序

`autograder.py` 自动评分器

`testParser.py` 解析测试用例

`testClasses.py` 通用

`test_cases/` 测试用例文件夹

`multiagentTestClasses.py` Project 2 specific autograding test classes

需要修改并提交的文件: 你需要在作业中填写`multiAgents.py`的部分内容（标识为## YOUR CODE HERE##的区域），并且最终**只需要**提交这个文件，并附上你的代码和注释。请不要改变这个文件夹中的其他文件，或提交这个文件以外的任何文件。

评估: 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有所不同。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。

Multi-Agent Pacman

首先，玩一个经典的吃豆子游戏:

```
python pacman.py
```

现在，查看`multiAgents.py`中提供的`ReflexAgent`: 他为我们提供了一些必要的方法以查看环境当前的状态。

```
python pacman.py -p ReflexAgent
```

```
python pacman.py -p ReflexAgent -l testClassic
```

检查`ReflexAgent`的代码，确保你明白它在做什么。

Question 1 (4 points): Minimax

现在，请你完成`multiAgents.py`中的`MinimaxAgent`类中的minimax搜索算法，以实现一个可以与幽灵👻进行对抗的智能体（你的程序需要支持一个吃豆人与多个幽灵智能体的博弈；但如果你实在不会，我们在Q1中使用的最终测试样例只设置了1个幽灵，因此一个仅支持双智能体博弈的MiniMax搜索也能通过最终测试）。你所实现的搜索树应当支持任意深度的搜索。

注意，`MinimaxAgent`类继承了`MultiAgentSearchAgent`类，因此在该类中你可以访问其父类中定义的`self.depth`和`self.evaluationFunction`属性，其定义如下:

- self.depth：minimax搜索树的最大深度。一个搜索层（即同一个depth）被认为是由一个吃豆人的动作和幽灵们的反应共同确定的，所以self.depth=2搜索将涉及吃豆人和幽灵们分别移动两次。
- self.evaluationFunction：一个评估函数，用于评估某一state的好坏。在本题中，默认的评估函数`self.evaluationFunction=scoreEvaluationFunction`，仅可以对搜索树的叶子节点进行评分。

请确保你编写的minimax代码在适当的地方引用这两个变量。注意，这两个变量是由命令行选项来指定的，无需你来设置。

Grading: 我们将检查你的代码以确定它是否探索了正确的游戏状态数量。这是检测学生正确实现minimax算法的唯一可靠方法。因此，autograder会对你的代码调用`GameState.generateSuccessor`的次数非常敏感。如果你调用的次数多或少，autograder都会报错。为了测试和调试你的代码，请运行

```
python autograder.py -q q1
```

这将显示你的算法在一些小规模测试用例上的表现。想要在没有图形界面的情况下运行它，请使用:

```
python autograder.py -q q1 --no-graphics
```

提示

- 正确实现minimax有可能导致吃豆人在某些测试中输掉比赛。不必担心，因为这是期望中的结果，它不会影响最终评分。
- 初始传入给maxmizer的depth参数为self.depth。因此，若涉及到递归调用，请在合适的位置将depth参数依次递减。
- 部分测试中存在两个鬼，但是对于Q1最终判分时我们的测试用例中只存在一个鬼。如果你能直接写出一个更general的算法，那么对Q2也是有帮助的。
- 这一部分中吃豆人测试的评估函数已经写好了（`self.evaluationFunction`）。请不要改变这个函数，但是要认识到，现在我们是在评估*states*，而不是行动。
- 在 "minimaxClassic"布局中，初始状态的最小值分别是9、8、7、-492，深度为1、2、3和4。请注意，尽管深度4的minimax预测很难，但你的minimax智能体往往会赢（对我们来说是665/1000局）。

  ```
  python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
  ```
- **吃豆人始终是agent 0**。
- minimax中的所有状态都应该是`GameStates`类型的，要么被传递给`getAction`，要么由`gamestate.generateSuccessor`方法生成。
- 在`openClassic`和`mediumClassic`（默认）等较大的棋盘上，你会发现吃豆人擅长生存，但不擅长取胜。他经常会白费力气，毫无进展。它甚至会在一个点旁边乱蹦乱跳，却不吃那个点，因为它不知道吃了那个点之后会去哪里。如果你看到这种行为，不必担心。
- 当吃豆人认为他的死亡是不可避免的，他会试图尽快结束游戏，避免持续的生存惩罚。有时候，这对基于随机策略的幽灵来说是错误的，但minimax算法总是假设最坏的情况:

  ```
  python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
  ```

  单纯为了确保你明白为什么吃豆人在这种情况下会冲到最近的Ghost。

Question 2 (4 points): Alpha-Beta Pruning

接下来，请你在`multiAgents.py`的`AlphaBetaAgent`类中实现一个新的智能体，其可以使用alpha-beta修剪来更有效地探索Minimax树。

完成后，你可以运行以下命令初步验证你的算法。你应该会发现这个算法比minimax搜索更快（也许深度3 alpha-beta将与深度2 minimax运行得一样快）。理想情况下，`smallClassic`上的深度3应该在每次移动中耗时几秒钟或更少。

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

另外，`AlphaBetaAgent`的minimax值应该与`MinimaxAgent`的minimax值相同。同样，在minimaxClassic布局中，初始状态对应于深度分别为1,2,3和4的minimmax值分别应为9,8,7和-492。

Grading: 因为我们需要检查你的代码以确定它是否探索了正确数量的状态，所以执行alpha-beta修剪时，务必不要改变子子节点的访问顺序。换句话说，后续状态应该总是按照`GameState.getLegalActions`返回的顺序进行处理。同样，不要调用`GameState.generatessuccer`超过必要次数。

为了测试和调试你的代码，运行

```
python autograder.py -q q2
```

这将显示你的算法在一些小规模测试用例上的表现，以及一个吃豆人游戏。要在没有图形的情况下运行它，请使用

```
python autograder.py -q q2 --no-graphics
```

同样，正确实现的alpha-beta剪枝算法仍旧有可能导致吃豆人在某些情况下输掉比赛，不必担心。

Question 3 (4 points): MCTS (Monte Carlo Tree Search)

Minimax和alpha-beta都很好，但它们都假设你是在与一个做出最优决定的对手比赛。任何赢过井字棋的人都可以告诉你，情况并不总是如此。在这个问题中，你将实现`MCTSAgent`。

具体而言，请你在`MCTSAgent`中创建一个新的基于蒙特卡洛树搜索智能体，并完成`selection`, `expansion`等所有标注有“YOUR CODE HERE”的函数。

注意:

- 我们在MCTSAgent中给出了大量的辅助代码（UCT，heuristic等），这些你都不需要修改，只要在了解其功能后直接调用即可。另外，我们在`getAction`函数中预留了`Node`类，它是你在实现蒙特卡洛树中需要使用的节点数据结构。
- 我们的MCTS的框架与一般的MCTS框架略有不同。例如我们的simulation函数不模拟幽灵的行动（这会降低simulation的复杂度），直接调用selection函数进行子节点的扩展和模拟（这通常比使用一个随机policy有更高的收敛效率）。在不同的应用中，我们往往会根据问题的特点对MCTS进行一些细节上的自定义。
- 请认真阅读相关代码注释，了解函数逻辑以及相关变量的类型，如`Node`，`GameState`等。

在完成相应的函数后，请使用下面的命令测试MCTSAgent：

```
python autograder.py -q q3
```

要查看MCTSAgent在Pacman中的表现，请运行:

```
python pacman.py -p MCTSAgent -l testClassic
```

同样，MCTS的正确实现可能导致Pacman在一些测试场景下输掉比赛。这不是一个问题：因为它是正确的行为，因此仍将通过我们最终的测试。

最后

请注意，我们在一些测试样例中设置了maxTime参数，以避免程序陷入无意义的循环。在最终的测试样例中，我们将其设置为10min，请检查你的程序用时。

我们的程序最终运行于服务器集群，通常比笔记本环境拥有更好的运行效率，因此不必担心。

Submission

你只需要提交`multiAgents.py`以及包含学号信息的yaml文件（`info.yaml`），以方便我们进行自动评分。