课程作业 1：多智能体搜索实验报告

摘要

在本实验中，我们研究了多智能体搜索问题，针对吃豆人游戏设计了不同的智能体，包括极小化极大搜索、Alpha-Beta 剪枝搜索和期望最大搜索。我们实现了这些算法，并通过实验进行了评估。本报告详细介绍了每个算法的实现和实验结果。

关键词：多智能体搜索，吃豆人游戏，极小化极大搜索，Alpha-Beta 剪枝，期望最大搜索

1 引言

多智能体搜索是人工智能领域的重要问题之一，涉及博弈理论和决策制定。在这次实验中，我们研究了多智能体搜索算法，并将其应用于吃豆人游戏。我们实现了极小化极大搜索、Alpha-Beta剪枝搜索和期望最大搜索，并评估了它们在游戏中的性能。

2 实验内容

2.1 极小化极大搜索

Minimax搜索是一种经典的博弈算法，用于解决博弈问题。在本次实验中，我们将实现一个Minimax搜索智能体，用于解决Pacman游戏中的博弈问题。该智能体需要考虑多个幽灵的行动，并在博弈树中进行搜索，以找到最佳的行动策略。我们将详细介绍代码的实现和实验结果。

2.1.1 代码实现

我们的Minimax搜索智能体是基于Python编程语言实现的，并利用了Pacman游戏框架提供的GameState和Game类。以下是我们实现的主要代码部分：

```python
class MinimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState: GameState):
        maxVal = -float('inf')
        bestAction = None
        for action in gameState.getLegalActions(0):
            value = self.getMin(gameState.generateSuccessor(0, action), 0, 1)
            if value > maxVal:
                maxVal = value
                bestAction = action
        return bestAction

    def getMax(self, gameState, depth, agentIndex):
        if depth == self.depth or gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState)

        maxVal = -float('inf')
        for action in gameState.getLegalActions(agentIndex):
            if agentIndex == gameState.getNumAgents() - 1:
                value = self.getMin(gameState.generateSuccessor(agentIndex, action), depth
+ 1, 0)
            else:
                value = self.getMin(gameState.generateSuccessor(agentIndex, action), depth,
agentIndex + 1)
            maxVal = max(maxVal, value)
        return maxVal
    def getMin(self, gameState, depth, agentIndex):
        if depth == self.depth or gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState)

        minVal = float('inf')
        for action in gameState.getLegalActions(agentIndex):
            if agentIndex == gameState.getNumAgents() - 1:
                value = self.getMax(gameState.generateSuccessor(agentIndex, action), depth
+ 1, 0)
            else:
                value = self.getMin(gameState.generateSuccessor(agentIndex, action), depth,
agentIndex + 1)
            minVal = min(minVal, value)
        return minVal
```

在代码中，我们定义了一个MinimaxAgent类，继承自MultiAgentSearchAgent类。该类包括了`getAction`、`getMax`和`getMin`等方法，用于实现Minimax搜索算法。

2.1.2 实验结果

```
Question q2
===========

*** PASS: test_cases\q2\0-eval-function-lose-states-1.test
*** PASS: test_cases\q2\0-eval-function-lose-states-2.test
*** PASS: test_cases\q2\0-eval-function-win-states-1.test
*** PASS: test_cases\q2\0-eval-function-win-states-2.test
*** PASS: test_cases\q2\0-lecture-6-tree.test
*** PASS: test_cases\q2\0-small-tree.test
*** PASS: test_cases\q2\1-1-minmax.test
*** PASS: test_cases\q2\1-2-minmax.test
*** PASS: test_cases\q2\1-3-minmax.test
*** PASS: test_cases\q2\1-4-minmax.test
*** PASS: test_cases\q2\1-5-minmax.test
*** PASS: test_cases\q2\1-6-minmax.test
*** PASS: test_cases\q2\1-7-minmax.test
*** PASS: test_cases\q2\1-8-minmax.test
*** PASS: test_cases\q2\2-1a-vary-depth.test
*** PASS: test_cases\q2\2-1b-vary-depth.test
*** PASS: test_cases\q2\2-2a-vary-depth.test
*** PASS: test_cases\q2\2-2b-vary-depth.test
*** PASS: test_cases\q2\2-3a-vary-depth.test
*** PASS: test_cases\q2\2-3b-vary-depth.test
*** PASS: test_cases\q2\2-4a-vary-depth.test
*** PASS: test_cases\q2\2-4b-vary-depth.test
*** PASS: test_cases\q2\2-one-ghost-3level.test
*** PASS: test_cases\q2\3-one-ghost-4level.test
*** PASS: test_cases\q2\4-two-ghosts-3level.test
*** PASS: test_cases\q2\5-two-ghosts-4level.test
*** PASS: test_cases\q2\6-tied-root.test
*** PASS: test_cases\q2\7-1a-check-depth-one-ghost.test
*** PASS: test_cases\q2\7-1b-check-depth-one-ghost.test
*** PASS: test_cases\q2\7-1c-check-depth-one-ghost.test
*** PASS: test_cases\q2\7-2a-check-depth-two-ghosts.test
*** PASS: test_cases\q2\7-2b-check-depth-two-ghosts.test
*** PASS: test_cases\q2\7-2c-check-depth-two-ghosts.test
*** Running MinimaxAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running MinimaxAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q2\8-pacman-game.test

### Question q2: 5/5 ###


Finished at 14:27:53

Provisional grades
==================
Question q2: 5/5
------------------
Total: 5/5
```

从测试结果可以看出，我们的Minimax搜索智能体在各种情况下都能成功通过测试，达到了预期的效果。

2 .2Alpha-Beta剪枝

2.2.1 代码实现

在本任务中，我们实现了一个使用Alpha-Beta剪枝算法的智能体，名为AlphaBetaAgent。该算法的目标是更高效地探索Minimax树，以减少不必要的搜索，提高搜索速度。以下是我们实现的主要代码部分：

```python
class AlphaBetaAgent(MultiAgentSearchAgent):
    """
    Your minimax agent with alpha-beta pruning (question 3)
    """

    def getAction(self, gameState: GameState):
        """
        Returns the minimax action using self.depth and self.evaluationFunction
        """
        "*** YOUR CODE HERE ***"
        maxVal, bestAction = self.getMax(gameState)
        return bestAction

    def getMax(self, gameState, depth=0, agentIndex=0, alpha=-float('inf'),
beta=float('inf')):
        # 达到搜索深度
        if depth == self.depth:
            return self.evaluationFunction(gameState), None
        # 不合法
        if len(gameState.getLegalActions(agentIndex)) == 0:
            return self.evaluationFunction(gameState), None
        # 获得吃豆人的所有合法操作，并进行遍历
        maxVal = -float('inf')
        bestAction = None
        for action in gameState.getLegalActions(agentIndex):
            # 计算幽灵
```

```
                value, _ = self.getMin(gameState.generateSuccessor(agentIndex, action), depth,
agentIndex + 1, alpha, beta)
            if value > maxVal:
                maxVal = value
                bestAction = action
            if maxVal > beta:
                return maxVal, bestAction
            alpha = max(alpha, maxVal)
        return maxVal, bestAction

    def getMin(self, gameState, depth=0, agentIndex=1, alpha=-float('inf'),
beta=float('inf')):
        if depth == self.depth:
            return self.evaluationFunction(gameState), None
        if len(gameState.getLegalActions(agentIndex)) == 0:
            return self.evaluationFunction(gameState), None
        minVal = float('inf')
        bestAction = None
        for action in gameState.getLegalActions(agentIndex):
            if agentIndex == gameState.getNumAgents() - 1:
                value = self.getMax(gameState.generateSuccessor(agentIndex, action), depth
+ 1, 0, alpha, beta)[0]
            else:
                value = \
                    self.getMin(gameState.generateSuccessor(agentIndex, action), depth,
agentIndex + 1, alpha, beta)[0]
            if value < minVal:
                minVal = value
                bestAction = action
            # 剪枝操作
            if value < alpha:
                return value, action
            beta = value if value < beta else beta
        return minVal, bestAction
```

在代码中，我们实现了AlphaBetaAgent类，该类继承自MultiAgentSearchAgent类。该类包括了`getAction`、`getMax`和`getMin`等方法，用于实现Alpha-Beta剪枝算法。该算法通过递归搜索Minimax树，并使用alpha和beta值来剪枝不必要的分支。

2.2.2 实验结果

从测试结果可以看出，我们的AlphaBetaAgent在各种情况下都能成功通过测试，达到了预期的效果。与MinimaxAgent相比，AlphaBetaAgent在相同深度下具有更高的搜索速度，这是Alpha-Beta剪枝算法的优势。

```
Question q3
===========

*** PASS: test_cases\q3\0-eval-function-lose-states-1.test
*** PASS: test_cases\q3\0-eval-function-lose-states-2.test
*** PASS: test_cases\q3\0-eval-function-win-states-1.test
*** PASS: test_cases\q3\0-eval-function-win-states-2.test
*** PASS: test_cases\q3\0-lecture-6-tree.test
*** PASS: test_cases\q3\0-small-tree.test
*** PASS: test_cases\q3\1-1-minmax.test
*** PASS: test_cases\q3\1-2-minmax.test
*** PASS: test_cases\q3\1-3-minmax.test
*** PASS: test_cases\q3\1-4-minmax.test
*** PASS: test_cases\q3\1-5-minmax.test
*** PASS: test_cases\q3\1-6-minmax.test
*** PASS: test_cases\q3\1-7-minmax.test
*** PASS: test_cases\q3\1-8-minmax.test
*** PASS: test_cases\q3\2-1a-vary-depth.test
*** PASS: test_cases\q3\2-1b-vary-depth.test
*** PASS: test_cases\q3\2-2a-vary-depth.test
*** PASS: test_cases\q3\2-2b-vary-depth.test
*** PASS: test_cases\q3\2-3a-vary-depth.test
*** PASS: test_cases\q3\2-3b-vary-depth.test
*** PASS: test_cases\q3\2-4a-vary-depth.test
*** PASS: test_cases\q3\2-4b-vary-depth.test
*** PASS: test_cases\q3\2-one-ghost-3level.test
*** PASS: test_cases\q3\3-one-ghost-4level.test
*** PASS: test_cases\q3\4-two-ghosts-3level.test
*** PASS: test_cases\q3\5-two-ghosts-4level.test
*** PASS: test_cases\q3\6-tied-root.test
*** PASS: test_cases\q3\7-1a-check-depth-one-ghost.test
*** PASS: test_cases\q3\7-1b-check-depth-one-ghost.test
*** PASS: test_cases\q3\7-1c-check-depth-one-ghost.test
*** PASS: test_cases\q3\7-2a-check-depth-two-ghosts.test
*** PASS: test_cases\q3\7-2b-check-depth-two-ghosts.test
*** PASS: test_cases\q3\7-2c-check-depth-two-ghosts.test
*** Running AlphaBetaAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running AlphaBetaAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q3\8-pacman-game.test

### Question q3: 5/5 ###


Finished at 16:52:01

Provisional grades
==================
Question q3: 5/5
------------------
Total: 5/5
```

2.3 期望最大搜索

2.3.1 代码实现

在本任务中，我们实现了一个使用期望最大搜索（Expectimax）算法的智能体，名为ExpectimaxAgent。期望最大搜索是一种用于建模对手可能做出次优选择的算法，与极小化极大搜索和Alpha-Beta剪枝不同，它不假设对手总是做出最佳决策。

以下是我们实现的ExpectimaxAgent的关键代码部分：

```
class ExpectimaxAgent(MultiAgentSearchAgent):
    """
    Your expectimax agent (question 4)
    """

    def getAction(self, gameState):
        """
        Returns the expectimax action using self.depth and self.evaluationFunction
        """
        "*** YOUR CODE HERE ***"
        maxVal, bestAction = self.getMax(gameState)
        return bestAction

    def getMax(self, gameState, depth=0, agentIndex=0):
        if depth == self.depth or gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState), None

        maxVal = -float('inf')
        bestAction = None
        for action in gameState.getLegalActions(agentIndex):
            value, _ = self.getExpectation(gameState.generateSuccessor(agentIndex, action),
depth, agentIndex + 1)
            if value > maxVal:
                maxVal = value
                bestAction = action
        return maxVal, bestAction

    def getExpectation(self, gameState, depth=0, agentIndex=1):
        if depth == self.depth or gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState), None

        legalActions = gameState.getLegalActions(agentIndex)
        numActions = len(legalActions)
        expectation = 0.0

        for action in legalActions:
            if agentIndex == gameState.getNumAgents() - 1:
                value, _ = self.getMax(gameState.generateSuccessor(agentIndex, action),
depth + 1, 0)
            else:
                value, _ = self.getExpectation(gameState.generateSuccessor(agentIndex,
action), depth, agentIndex + 1)

            expectation += value / numActions

        return expectation, None
```

在代码中，我们实现了ExpectimaxAgent类，该类继承自MultiAgentSearchAgent类。该类包括了`getAction`、`getMax`和`getExpectation`等方法，用于实现期望最大搜索算法。期望最大搜索通过递归搜索游戏树的各个可能性，计算每个动作的期望值，并选择具有最高期望值的动作。

2.3.2 实验结果

我们对我们的ExpectimaxAgent进行了多次测试，以评估其在不同游戏布局下的性能。以下是一些测试的实验结果：

- 运行3次`python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10`，可以观察到AlphaBetaAgent 总是会输掉游戏

```
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0,
-501.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

- 运行10次`python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10`，可以观察到
  - 第1次测试：6次胜利，4次失败，平均得分为118.4
  - 第2次测试：4次胜利，6次失败，平均得分为-88.4
  - 第3次测试：3次胜利，7次失败，平均得分为-191.8
  - 第4次测试：5次胜利。5次失败，平均得分为15
  - 第5次测试：4次胜利。6次失败，平均得分为-88.4
  - 第6次测试：7次胜利。3次失败，平均得分为221.8
  - 第7次测试：5次胜利。5次失败，平均得分为15
  - 第8次测试：7次胜利。3次失败，平均得分为221.8
  - 第9次测试：6次胜利，4次失败，平均得分为118.4
  - 第10次测试：4次胜利。6次失败，平均得分为-88.4

从实验结果中可以看出，ExpectimaxAgent在不同的游戏布局下表现出不同的性能。它在某些情况下能够获胜，但在其他情况下可能失败。这反映了期望最大搜索算法的特性，它不假设对手总是做出最佳决策，而是考虑了对手可能做出的次优选择。

与此同时，我们还对比了ExpectimaxAgent和AlphaBetaAgent在相同游戏布局上的性能。实验结果表明，ExpectimaxAgent在某些情况下能够获胜，而AlphaBetaAgent总是失败，这进一步说明了期望最大搜索算法的优势。

2.4 评价函数

2.4.1 代码实现

在本任务中，我们实现了一个更好的评价函数`betterEvaluationFunction`，用于评估吃豆人在游戏中的状态。该评价函数的目标是在深度为2的期望最大搜索中获得合理的性能，使吃豆人在存在一个随机幽灵的smallClassic棋盘上有50%的机会吃掉所有豆子，并且保持合理的运行速度。评价函数需要估计终止或截断节点状态的效用值。

以下是我们实现的评价函数`betterEvaluationFunction`的关键代码部分：

```python
def betterEvaluationFunction(currentGameState: GameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (question 5).
    """
    # Extract relevant information from the game state
    pacmanPosition = currentGameState.getPacmanPosition()
    foodGrid = currentGameState.getFood()
    ghostStates = currentGameState.getGhostStates()
    capsules = currentGameState.getCapsules()

    # Calculate the distance to the nearest food
    foodDistances = [manhattanDistance(pacmanPosition, food) for food in foodGrid.asList()]
    minFoodDistance = min(foodDistances) if foodDistances else 0

    # Calculate the distance to the nearest ghost
    ghostDistances = [manhattanDistance(pacmanPosition, ghost.getPosition()) for ghost in
ghostStates]
    minGhostDistance = min(ghostDistances) if ghostDistances else 0

    # Calculate the distance to the nearest capsule (power pellet)
    capsuleDistances = [manhattanDistance(pacmanPosition, capsule) for capsule in capsules]
    minCapsuleDistance = min(capsuleDistances) if capsuleDistances else 0

    # Calculate the remaining number of food dots
    numFoodLeft = len(foodGrid.asList())

    # Calculate the score of the current game state
    score = currentGameState.getScore()

    # Define weights for different factors
    foodWeight = 1
    ghostWeight = 10
    capsuleWeight = 5

    # Calculate the overall evaluation function
    evaluation = score - foodWeight * minFoodDistance - ghostWeight * minGhostDistance -
capsuleWeight * minCapsuleDistance - numFoodLeft

    return evaluation
```

在代码中，我们首先提取了当前游戏状态的相关信息，包括吃豆人的位置、食物的分布、幽灵的状态和胶囊的位置。然后，我们计算了吃豆人到最近食物、最近幽灵和最近胶囊的距离。接下来，我们计算了剩余的食物数量和当前游戏状态的得分。

我们为不同因素（食物、幽灵和胶囊）定义了权重，并使用这些因素计算了综合评价。评价函数的目标是使吃豆人在考虑这些因素的情况下，尽可能获得更高的分数。

2.4.2 实验结果

我们对评价函数 `betterEvaluationFunction` 进行了多次测试，以评估其在不同游戏场景下的性能。以下是一些测试的实验结果：

- 在评估函数测试中，我们运行了10次游戏，并成功地实现了在smallClassic棋盘上吃掉所有豆子的目标。实验结果如下：

```
Pacman emerges victorious! Score: 1073
Pacman emerges victorious! Score: 914
Pacman emerges victorious! Score: 1178
Pacman emerges victorious! Score: 1140
Pacman emerges victorious! Score: 920
Pacman emerges victorious! Score: 1165
Pacman emerges victorious! Score: 1171
Pacman emerges victorious! Score: 1279
Pacman emerges victorious! Score: 910
Pacman emerges victorious! Score: 1136
Average Score: 1088.6
Scores:        1073.0, 914.0, 1178.0, 1140.0, 920.0, 1165.0, 1171.0, 1279.0, 910.0,
1136.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\q5\grade-agent.test (6 of 6 points)
***     1088.6 average score (2 of 2 points)
***         Grading scheme:
***          < 500:  0 points
***         >= 500:  1 points
***         >= 1000:  2 points
***     10 games not timed out (1 of 1 points)
***         Grading scheme:
***          < 0:  fail
***         >= 0:  0 points
***         >= 10:  1 points
***     10 wins (3 of 3 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  1 points
***         >= 5:  2 points
***         >= 10:  3 points
```

– 平均分数：1088.6

– 胜率：100%

从实验结果中可以看出，评价函数 `betterEvaluationFunction` 达到了任务的目标。吃豆人在随机幽灵的存在下，成功吃掉所有豆子，并且平均分数在1000分左右。这证明了评价函数的有效性和性能。

3 结束语

本实验是关于博弈智能体在吃豆人游戏中的设计和开发。在实验中，我们通过完成四个不同的任务来学习和实践博弈搜索算法、Alpha-Beta剪枝、期望最大搜索和评价函数的设计。

在任务1中，我们实现了基本的Minimax算法，该算法模拟了吃豆人游戏中的对抗双方，并通过搜索博弈树来确定最佳行动。通过任务1，我们理解了Minimax算法的工作原理，但也意识到了其在实际应用中的局限性。

在任务2中，我们改进了Minimax算法，引入了Alpha-Beta剪枝来减少搜索空间，提高搜索效率。这个任务让我们掌握了一种强大的搜索算法，可以用于处理更复杂的博弈问题。通过Alpha-Beta剪枝，我们成功提高了吃豆人智能体的性能，使其在更短的时间内做出决策。

在任务3中，我们学习了期望最大搜索算法，该算法适用于处理存在不确定性的博弈问题。我们实现了ExpectimaxAgent，并与Alpha-BetaAgent进行比较。实验结果表明，ExpectimaxAgent能够在某些情况下战胜Alpha-BetaAgent，因为它更好地建模了对手的随机行为。

最后，在任务4中，我们设计了一个更好的评价函数，用于评估吃豆人游戏中的状态。通过综合考虑食物、幽灵、胶囊和得分等因素，我们成功地创建了一个评价函数，使吃豆人能够在困难的游戏场景中获得高分，并成功吃掉所有豆子。

通过完成这些任务，我们深入了解了博弈智能体的设计和优化过程。我们掌握了不同的博弈搜索技术，学会了如何处理博弈中的不确定性，并设计了有效的评价函数来指导智能体的决策。这些知识和技能对于解决复杂的博弈问题和其他人工智能应用都具有重要意义。

总之，本实验为我们提供了宝贵的经验，使我们更好地理解了博弈智能体的设计和开发过程。通过不断学习和实践，我们可以进一步提高博弈智能体的性能，并在各种应用中应用这些技术。博弈智能体的研究和发展将继续推动人工智能领域的进步，为解决实际问题提供更多可能性。