实验报告

（2023~2024 学年第二学期）

课程名称 人工智能课程设计

实验名称 课程设计作业 Project2

姓名 范潇 学号 2254298

1. 项目概述

1.1. 主要内容与目标

本次项目的主要内容为多智能体环境下的智能体实现。具体目标是实现算法使得吃豆人能够在迷宫中收集食物，并避免被幽灵击败。这里的吃豆人游戏被化简为回合制游戏，即各个智能体轮流执行动作，而非同时执行动作。同时，需要考虑有三个及以上的智能体的情况。

在该项目中，首先需要对于基于反射的智能体进行优化。该智能体的特点在于，它是根据当前状态，来评估各个合法的动作。而该项目涉及到的另一个智能体则是直接对于依据未来的状态来选择下一动作，要求在此基础上实现 Minimax，Alpha-Beta 算法。这两种算法的使用前提是对手绝对理性，总是依据最优解执行操作。但实际情况可能并非如此，为此，项目要求我们实现 Expectimax。同时，项目还要求实现 Evaluation Function 来直接评估未来状态，以便减少计算量。

1.2. 已有代码

下面对已有代码中与本项目的具体实现相关的代码进行分析。

1.2.1. util.py

本项目中主要需要使用该文件中提供的 manhattanDistance 函数，用于计算两坐标之间的曼哈顿距离。

1.2.2. multiAgents.py

该文件用于存储各种智能体的实现。

该文件给出了一个样例：ReflexAgent(Agent)。在该类中，有 evaluationFunction 和 getAction 两个方法。前者用于评估各个合法的下一个状态，这里直接返回得分；后者随机选取使得下一个状态得分最大的动作。这里 evaluationFunction 的特殊指出在于接收的是状态-动作二元组。在 evaluationFunction 的实现中，可以了解到，我们可以从 GameState 类中获取到吃豆人当前位置，剩余食物位置等有用的信息。

该文件还给出了 MultiAgentSearchAgent 类，有 index，evaluationFunction，depth 三个属性。其中 index 恒为 0，因为我们约定吃豆人为 0 号智能体；evaluationFunction 是用于对给定状态进行评估的函数；depth 是当前状态所处的深度。它有三个子类：MinimaxAgent,AlphaBetaAgent,ExpectimaxAgent，需要我们去实现。

1.2.3. GameState.py

该文件中的内容为吃豆人的游戏主体，包含 GameState 等类。GameState 类中包含了获取吃豆人合法动作、智能体的下一个状态、吃豆人位置、幽灵位置、智能体数量、食物数量、食物位置、是否胜利等有用的状态信息。

1.2.4. Game.py

该文件中的内容为吃豆人游戏中的智能体相关实现。其中包含 Configuration、AgenState 等类。通过 Configuration 类，我们能够获取到智能体的当前位置。

2. Question1

2.1. 问题概述

本问题要求改进已有的基于反射的智能体。这里需要考虑有多个幽灵的场景，默认情况下幽灵的行动是随机的，最后评分的依据首先是能够确保吃豆人不被幽灵击败并收集到所有的食物，然后是分数尽可能地高。

2.2. 算法设计

已有代码中实现的智能体选择下一个行为的唯一依据是下一个状态的得分。这不仅导致吃豆人收集所有食物的效率大大降低——当四周没有食物时，很有可能会来回兜圈子，更重要的是，这几乎没有考虑幽灵对于吃豆人的影响——吃豆人可能会移动到幽灵的相邻位置，这导致吃豆人暴露在被幽灵吃掉的风险之中，而这并不能体现在下一个状态的得分之中，因为这需要涉及到再下一个状态。

为此，应该充分利用幽灵的位置信息和剩余食物的位置信息，使得确保吃豆人不会被幽灵击败的前提下尽可能高效地收集所有食物。为了保证吃豆人的安全，我们只需要确保下一个状态和幽灵的曼哈顿距离不小于等于 1 即可。

在确保没有被幽灵击败的前提下，我优先让吃豆人选择直接收集到食物的动作，如果没有，则以到最近的食物的曼哈顿距离的相反数为评估函数，即采用贪心的策略，让吃豆人缩短其到最近的食物的距离。

2.3. 算法实现

```
def evaluationFunction(self, currentGameState: GameState, action):
    successorGameState = currentGameState.generatePacmanSuccessor(action)
    newPos = successorGameState.getPacmanPosition()
    newFood = successorGameState.getFood()
    newGhostStates = successorGameState.getGhostStates()
    newScaredTimes = [ghostState.scaredTimer for ghostState in
        newGhostStates]

    if action == 'Stop':  # 不允许停止
        return -float('inf')
    newGhostPos = [ghostState.configuration.pos for ghostState in
        newGhostStates]
    # 只有当不进入无敌状态时才要考虑和幽灵之间的距离
    if sum(newScaredTimes) == 0 and min([manhattanDistance(ghostPos, newPos)
        for ghostPos in newGhostPos]) <= 1:
        return -float('inf')
    foods = newFood.asList()
```

```
15      # 没有食物，即该步使得游戏胜利
16      if not foods:
17          return float('inf')
18      score = -min([manhattanDistance(food, newPos) for food in foods]) -
            100000 * len(foods)
19      return score
```

这里要求吃豆人永远不停止不动，但实际上，在特别极端的情况下，例如吃豆人位于角落，幽灵处于它的紧邻的对角位置时，静止不动以观察幽灵的下一步才是正确的。实现时，我用惩罚项-100000*len(foods) 来体现出收集食物的优先性。值得注意的是，如果某个动作使得吃豆人收集到了一个食物，那么在进入该函数时，该食物已经从食物列表中移除出来了，如果仅仅依据到食物列表中各个食物的曼哈顿距离的最小值，是无法判断出这一步究竟是否能够收集到食物的。

2.4. 实验结果

我成功获得了本问题的所有分数。

```
Average Score: 1239.9
Scores:        1238.0, 1244.0, 1239.0, 1235.0, 1233.0, 1241.0, 1246.0, 1242.0, 1239.0, 1242.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\q1\grade-agent.test (4 of 4 points)
***     1239.9 average score (2 of 2 points)
***         Grading scheme:
***          < 500:  0 points
***         >= 500:  1 points
***         >= 1000:  2 points
***     10 games not timed out (0 of 0 points)
***         Grading scheme:
***          < 10:  fail
***         >= 10:  0 points
***     10 wins (2 of 2 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  0 points
***         >= 5:  1 points
***         >= 10:  2 points

### Question q1: 4/4 ###
```

图 2.1: Question1 实验结果

本问题的测试方式如图 2.2所示。该地图中没有墙壁，且只有一个随机移动的幽灵。通过该测试可以说明我所实现的算法能够指导吃豆人高效地收集所有的食物，同时避免被幽灵击败。同时，我还额外对于有两个幽灵、有墙壁的中等迷宫场景进行了测试，如图 2.3，在连续两次测试中都成功收集所有的食物，结果如图 2.4所示。

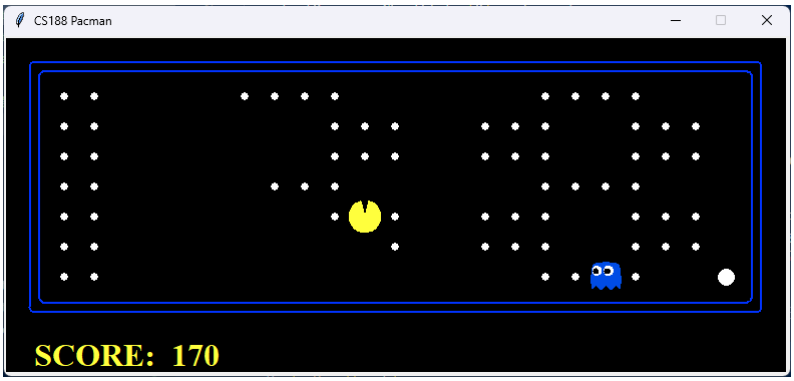

图 2.2: Question1 测试样例

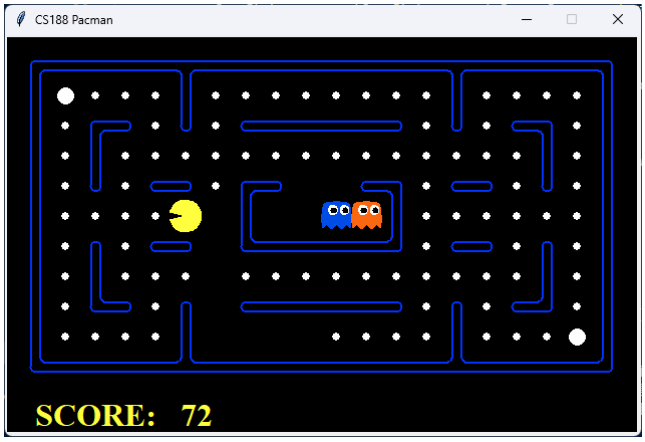

图 2.3: 中等迷宫测试场景

```
(base) PS C:\Users\15800\Desktop\cs188\project\multiagent> python pacman.py --frameTime 0 -p ReflexAgent -k 2
Pacman emerges victorious! Score: 1728
Average Score: 1728.0
Scores:        1728.0
Win Rate:      1/1 (1.00)
Record:        Win
(base) PS C:\Users\15800\Desktop\cs188\project\multiagent> python pacman.py --frameTime 0 -p ReflexAgent -k 2
Pacman emerges victorious! Score: 1201
Average Score: 1201.0
Scores:        1201.0
Win Rate:      1/1 (1.00)
Record:        Win
```

图 2.4: 中等迷宫测试结果

3. Question2

3.1. 问题概述

本问题要求实现 Minimax 算法，其中正方只有一个智能体，即吃豆人，但是反方，即幽灵，可能有多个。

3.2. 算法设计

由于可能有多个反方，所以在博弈树中，对应着正方的每一层下方都紧接着 n 层反方的结点，其中 n 为反方的个数。同时，题目明确要求当层数到达给定值后，便使用评估函数，其中，“一层”对应着一轮游戏，即吃豆人和所有的幽灵都轮流各走了一步。

Algorithm 3.2.1: Minimax(gameState)

```
dummy,action = maximize (gameState,1)// 这里无需返回的分数
return action
```

Procedure maximize(gameState,depth)

```
Output: bestScore,bestAction
// 给定状态和对应深度，返回从当前状态出发所能达到的最佳分数和对应的动作
actions = gameState.getLegalActions()
ghostNum = gameState.getNumAgent - 1
if depth > depthLimit or actions == None then
    return evaluationFunction(gameState),None// 无法继续行动，直接返回当前状态的分数
end if
bestScore = −∞
bestAction = None
foreach action in actions do
    next = gameState.generateSuccessor(gameState,action)// 获取下一个状态
    score,dummy = minimize (next,1,ghostNum,depth)// 这里并不需要返回的动作
    if score > bestScore then
        bestAction = action
        bestScore = score
    end if
end foreach
return bestScore,bestAction
```

Minimax 算法的核心是递归，并且使用的前提是博弈双方是理性的。maximize，minimize 函数分别用于给出正反方的下一步以及对应的最优解的分数。当无法继续行动时，当前状态是唯一可行的状态，自

Procedure minimize(gameState,ghostNo,ghostNum,depth)

Output: bestScore,bestAction

// 给定状态和对应深度，返回从当前状态出发所能达到的最佳分数和对应的动作

actions = gameState.getLegalActions()
if *depth* > *depthLimit or actions* == *None* **then**
  return *evaluationFunction(gameState),None*// 无法继续行动，直接返回当前状态的分数
end if
bestScore = ∞
bestAction = None
foreach *action* **in** *actions* **do**
  next = gameState.generateSuccessor(gameState,action)// 获取下一个状态
  if *ghostNo* == *ghostNum* **then**
    score,dummy = `maximize` (next,depth+1)// 当前智能体是最后一个幽灵，下一层为吃豆人
  else
    score,dummy = **minimize** (next,ghostNo+1,ghostNum,depth)// 这里并不需要返回的动作
  end if
  if *score* < *bestScore* **then** // 对于幽灵而言，目标是让吃豆人的分数尽可能的低
    bestAction = action
    bestScore = score
  end if
end foreach
return *bestScore,bestAction*

然是最优解。如果可以正方行动，那么在行动前，由于反方是理性的，正方需要枚举所有可能的行为，通过调用 minimize 推算反方的反应和最终的结果，并最终选取出最佳的下一步。反方亦然。特别之处在于有多位反方，由于算法中的 score 统一是指正方的分数，所以无论是调用 minimize 的反方，还是最后一位调用 maximize 的反方，其选取下一步的依据都是让 score 最小。

如果不在指定的深度调用评估函数，那么 Minimax 算法可能会产生无穷深的博弈树，这是不可接受的。

3.3. 算法实现

```python
class MinimaxAgent(MultiAgentSearchAgent):
    def maximize(self, gameState: GameState, depth: int):
        actions = gameState.getLegalActions(0)
        if depth > self.depth or not actions:
            return self.evaluationFunction(gameState), None
        bestScore = -float('inf')
        bestAction = None
```

```
        for action in actions:
            next = gameState.generateSuccessor(0, action)
            score, _ = self.minimize(next, 1, gameState.getNumAgents() - 1,
                depth)
            if score > bestScore:
                bestAction = action
                bestScore = score
        return bestScore, bestAction

    def minimize(self, gameState: GameState, ghostNo: int, ghostNum: int,
        depth: int):
        actions = gameState.getLegalActions(ghostNo)
        if not actions:
            return self.evaluationFunction(gameState), None
        bestScore = float('inf')
        bestAction = None
        for action in actions:
            next = gameState.generateSuccessor(ghostNo, action)
            if ghostNo == ghostNum:
                score, _ = self.maximize(next, depth + 1)
            else:
                score, _ = self.minimize(next, ghostNo + 1, ghostNum, depth)
            if score < bestScore:
                bestScore = score
                bestAction = action
        return bestScore, bestAction

    def getAction(self, gameState: GameState):
        return self.maximize(gameState, 1)[1]
```

3.4. 实验结果

我成功获得了本问题的所有分数，图 3.1为实验结果。测试用例分为以下几类：

1. eval-function-lose-states,eval-function-win-states：测试内容为一颗高度为 2 的博弈树，目的在于测试算法是否判断当前状态为胜负已定，并调用评估函数
2. lecture-6-tree,small-tree,minmax：测试内容为 Minimax 的基本功能
3. vary-depth：在 minmax 测试的基础上，对深度的限制进行调整，以测试算法是否正确处理该情况
4. x-ghost-xlevel：测试了多位反方和多层深度限制的情况
5. tied-root：测试了算法是否正确处理得分相同的情况
6. check-depth-x-ghost：重点测试了算法是否正确处理深度限制

7. pacman-game：在小型迷宫中使用 Minimax 算法进行实战

```
*** 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 ###
```

图 3.1: Question2 实验结果

4. Question3

4.1. 问题概述

本问题要求实现 Alpha-Beta 剪枝算法，其中正方只有一个智能体，即吃豆人，但是反方，即幽灵，可能有多个。

4.2. 算法设计

题目中明确要求不能对子节点进行重排序，同时不修剪掉和得分和已有 Alpha，Beta 值相同的子树。

Algorithm 4.2.1: Alpha-Beta(gameState)

```
dummy,action = maximize (gameState,1,−∞,∞)// 这里无需返回的分数
return action
```

Procedure maximize(gameState,depth,alpha,beta)

```
Output: bestScore,bestAction
// 给定状态和对应深度，返回从当前状态出发所能达到的最佳分数和对应的动作
actions = gameState.getLegalActions()
ghostNum = gameState.getNumAgent - 1
if depth > depthLimit or actions == None then
    return evaluationFunction(gameState),None// 无法继续行动，直接返回当前状态的分数
end if
bestScore = −∞
bestAction = None
foreach action in actions do
    next = gameState.generateSuccessor(gameState,action)// 获取下一个状态
    score,dummy = minimize (next,1,ghostNum,depth,alpha,beta)// 这里并不需要返回的动作
    if score > bestScore then
        bestAction = action
        bestScore = score
    end if
    if score>beta then return bestScore,bestAction // 被剪枝
    if bestScore>alpha then alpha = bestScore // 更新 alpha 值
end foreach
return bestScore,bestAction
```

Alpha-Beta 剪枝算法只是在 Minimax 的算法基础上略作改动，但是却能够在性能上得到很大提升。其核心思想在于维护两个值：Alpha 和 Beta，分别记录从博弈树根节点到当前结点的路径上的正方和反

Procedure minimize(gameState,ghostNo,ghostNum,depth,alpha,beta)

Output: bestScore,bestAction

// 给定状态和对应深度，返回从当前状态出发所能达到的最佳分数和对应的动作

actions = gameState.getLegalActions()
if *depth* > *depthLimit or actions* == *None* **then**
    return *evaluationFunction(gameState),None*// 无法继续行动，直接返回当前状态的分数
end if
bestScore = ∞
bestAction = None
foreach *action* **in** *actions* **do**
    next = gameState.generateSuccessor(gameState,action)// 获取下一个状态
    if *ghostNo* == *ghostNum* **then**
        score,dummy = `maximize` (next,depth+1,alpha,beta)// 当前智能体是最后一个幽灵，下一层为吃豆人
    else
        score,dummy = **minimize** (next,ghostNo,ghostNum,depth,alpha,beta)// 这里并不需要返回的动作
    end if
    if *score* < *bestScore* **then** // 对于幽灵而言，目标是让吃豆人的分数尽可能的低
        bestAction = action
        bestScore = score
    end if
    if *score*<*alpha* **then return** *bestScore,bestAction* // 被剪枝
    if *bestScore*<*beta* **then** beta = bestScore // 更新 `beta` 值
end foreach
return *bestScore,bestAction*

方的最佳选择。在 Minimax 算法中，虽然算法最终给出的只是从根节点出发的下一步，但是它的背后是博弈树上的一整条从根节点到某一叶子结点的路径，因此，如果我们可以证明某一结点不会处于该路径上，那么以这一结点为根节点的子树便可以被剪枝掉了。

对于正方的结点，如果它的子节点上的值大于 Beta 值，那么它自身也肯定大于 Beta。由于对于反方而言，Beta 值对应着另外一条不经过该结点的更优的解，所以最终反方选取的最优解肯定不会包含该节点，从而我们可以将该结点及其子树剪枝掉。对于反方的结点亦然。

4.3. 算法实现

```
1 class AlphaBetaAgent(MultiAgentSearchAgent):
2     def maximize(self, gameState: GameState, depth: int, alpha: float, beta:
          float):
3         actions = gameState.getLegalActions(0)
4         if depth > self.depth or not actions:
```

```
            return self.evaluationFunction(gameState), None
        bestScore = -float('inf')
        bestAction = None
        for action in actions:
            next = gameState.generateSuccessor(0, action)
            score, _ = self.minimize(next, 1, gameState.getNumAgents() - 1,
                depth, alpha, beta)
            if score > bestScore:
                bestAction = action
                bestScore = score
            if score > beta:
                return bestScore, bestAction
            if bestScore > alpha:
                alpha = bestScore
        return bestScore, bestAction

    def minimize(self, gameState: GameState, ghostNo: int, ghostNum: int,
        depth: int, alpha: float, beta: float):
        actions = gameState.getLegalActions(ghostNo)
        if not actions:
            return self.evaluationFunction(gameState), None
        bestScore = float('inf')
        bestAction = None
        for action in actions:
            next = gameState.generateSuccessor(ghostNo, action)
            if ghostNo == ghostNum:
                score, _ = self.maximize(next, depth + 1, alpha, beta)
            else:
                score, _ = self.minimize(next, ghostNo + 1, ghostNum, depth,
                    alpha, beta)
            if score < bestScore:
                bestScore = score
                bestAction = action
            if score < alpha:
                return bestScore, bestAction
            if bestScore < beta:
                beta = bestScore
        return bestScore, bestAction

    def getAction(self, gameState: GameState):
        return self.maximize(gameState, 1, -float('inf'), float('inf'))[1]
```

4.4. 实验结果

我成功获得了本问题的所有分数，图 4.1为实验结果。测试用例与问题 2 中所使用的一致。

```
*** 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
```

图 4.1: Question3 实验结果

5. Question4

5.1. 问题概述

本问题要求实现基于期望的智能体，它能够较好地处理对手为并不保证按照最优解执行的智能体的情况。

5.2. 算法设计

本问题中假设幽灵是随机移动的。对于这种情况，Minimax 算法不再适用，因为对手不再保证执行最优解。对此，应该调整正方，即吃豆人，选择下一步的策略。在 Minimax 算法中，选择的依据是“对手总是选择最优解”这一前提，而此时，对手的操作是完全随机的，即任何一个合法操作都等概率被执行。将这些操作的效果综合起来，便是分数的期望，因此，正方选择操作的依据是使得分数的期望最大化。

Algorithm 5.2.1: Minimax(gameState)

```
1 dummy,action = maximize (gameState,1)// 这里无需返回的分数
2 return action
```

Procedure maximize(gameState,depth)

```
   Output: bestScore,bestAction
   // 给定状态和对应深度，返回从当前状态出发所能达到的最佳分数和对应的动作
1  actions = gameState.getLegalActions()
2  ghostNum = gameState.getNumAgent - 1
3  if depth > depthLimit or actions == None then
4  |  return evaluationFunction(gameState),None// 无法继续行动，直接返回当前状态的分数
5  end if
6  bestScore = −∞
7  bestAction = None
8  foreach action in actions do
9  |  next = gameState.generateSuccessor(gameState,action)// 获取下一个状态
10 |  score = randomMove (next,1,ghostNum,depth)
11 |  if score > bestScore then
12 |  |  bestAction = action
13 |  |  bestScore = score
14 |  end if
15 end foreach
16 return bestScore,bestAction
```

Procedure randomMove(gameState,ghostNo,ghostNum,depth)

Output: bestScore,bestAction

// 给定状态和对应深度，返回从当前状态出发所能达到的最佳分数和对应的动作

actions = gameState.getLegalActions()
if *depth* > *depthLimit or actions* == *None* **then**
  return *evaluationFunction(gameState),None*// 无法继续行动，直接返回当前状态的分数
end if
totScore = 0
foreach *action* **in** *actions* **do**
  next = gameState.generateSuccessor(gameState,action)// 获取下一个状态
  if *ghostNo* == *ghostNum* **then**
    score,dummy = `maximize` (next,depth+1)// 当前智能体是最后一个幽灵，下一层为吃豆人
  else
    score = **randomMove** (next,ghostNo+1,ghostNum,depth)
  end if
  totScore += score
end foreach
return *totScore/* **len** *(actions)*// 等概率时期望便是平均值

5.3. 算法实现

```python
class ExpectimaxAgent(MultiAgentSearchAgent):
    def maximize(self, gameState: GameState, depth: int):
        actions = gameState.getLegalActions(0)
        if gameState.isWin() or depth > self.depth or not actions:
            return self.evaluationFunction(gameState), None
        bestScore = -float('inf')
        bestAction = None
        for action in actions:
            next = gameState.generateSuccessor(0, action)
            score = self.randomMove(next, 1, gameState.getNumAgents() - 1,
                depth)
            if score > bestScore:
                bestAction = action
                bestScore = score
        return bestScore, bestAction

    def randomMove(self, gameState: GameState, ghostNo: int, ghostNum: int,
        depth: int):
        actions = gameState.getLegalActions(ghostNo)
        if gameState.isLose() or not actions:
```

```
            return self.evaluationFunction(gameState)
        totScore = 0
        for action in actions:
            next = gameState.generateSuccessor(ghostNo, action)
            if ghostNo == ghostNum:
                score, _ = self.maximize(next, depth + 1)
            else:
                score = self.randomMove(next, ghostNo + 1, ghostNum, depth)
            totScore += score
        return totScore / len(actions)

    def getAction(self, gameState: GameState):
        return self.maximize(gameState, 1)[1]
```

5.4. 实验结果

我成功获得了本问题的所有分数，图 5.1为实验结果。所用到的测试用例中，expectimax 测试是特有的，测试的是该算法的基本功能。

```
*** PASS: test_cases\q4\0-eval-function-lose-states-1.test
*** PASS: test_cases\q4\0-eval-function-lose-states-2.test
*** PASS: test_cases\q4\0-eval-function-win-states-1.test
*** PASS: test_cases\q4\0-eval-function-win-states-2.test
*** PASS: test_cases\q4\0-expectimax1.test
*** PASS: test_cases\q4\1-expectimax2.test
*** PASS: test_cases\q4\2-one-ghost-3level.test
*** PASS: test_cases\q4\3-one-ghost-4level.test
*** PASS: test_cases\q4\4-two-ghosts-3level.test
*** PASS: test_cases\q4\5-two-ghosts-4level.test
*** PASS: test_cases\q4\6-1a-check-depth-one-ghost.test
*** PASS: test_cases\q4\6-1b-check-depth-one-ghost.test
*** PASS: test_cases\q4\6-1c-check-depth-one-ghost.test
*** PASS: test_cases\q4\6-2a-check-depth-two-ghosts.test
*** PASS: test_cases\q4\6-2b-check-depth-two-ghosts.test
*** PASS: test_cases\q4\6-2c-check-depth-two-ghosts.test
*** Running ExpectimaxAgent 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 ExpectimaxAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q4\7-pacman-game.test

### Question q4: 5/5 ###
```

图 5.1: Question4 实验结果

6. Question5

6.1. 问题概述

本问题要处理的场景中可能有多个幽灵，但是它们的行动是完全随机的，要求为该场景中的基于期望的智能体设计一个评估函数。最终目标是确保吃豆人能够获胜的前提下，使得分数最大化。

6.2. 算法设计

本题与第一题的区别在于，这里的评估函数只以给定的状态为参数，而在第一题中，既有状态，也有动作，相当于给出了动作先后的两个状态。这一区别导致的最大的影响在于我们很难使用某个值的变化量作为评估函数的组成部分。例如，我们无法直接量化时间的流逝，我们只能通过当前的游戏得分来间接反映时间因素——游戏得分会随着时间逐渐减少。同时，这还会带来一些不方便之处，例如，如果把吃豆人到最近的能量点的距离的倒数作为激励项，来激励吃豆人获取能量点的话，反而会适得其反，因为当吃豆人来到某一能量点相邻的位置时，此时，如果它吃掉了能量点，那么在这一状态下，到最近的能量点的距离反而增加了，因为最近的能量点更新为了之前的第二近的能量点。这可能会导致吃豆人尽管会被能量点吸引，但始终不去收集它。

因此，评估函数中的激励项应该满足“递增”的特性，惩罚项要满足“递减”的特性。最终，经过实验，我将评估函数设计为三部分：1）游戏得分；2）到最近的食物的距离的倒数；3）能量点的相反数。其中，第 3 项会使得当吃豆人路过能量点时顺带去收集它；第 2 项中，虽然采用了“最近距离的倒数”的形式，但是并不会导致吃豆人不收集食物的情况，这是因为食物和能量点不同，收集食物时游戏得分便会增加 10 分，“最近距离的倒数”最多只会减少 1 分，因此对于吃豆人而言，收集相邻的食物总是有益的，而对于能量点，由于收集它时并无法立即得到加分，“最近距离的倒数”反而会减少，如果吃豆人不够“有远见”，即无法搜索博弈树中足够多的层数的话，它会认为该行为是没有收益的；第一项游戏得分十分重要，因为这一项便能驱使吃豆人在普通状态下远离幽灵，同时主动追逐附近的处于虚弱状态的幽灵。

6.3. 算法实现

```
1 def betterEvaluationFunction(currentGameState: GameState):
2     newPos = currentGameState.getPacmanPosition()
3     newFood = currentGameState.getFood()
4     foods = newFood.asList()
5     capsules = currentGameState.getCapsules()
6     closeToFood = 0 if not foods else 1 / (min([manhattanDistance(newPos,
          food) for food in foods]))
7     return currentGameState.getScore() + closeToFood - len(capsules)
```

6.4. 实验结果

我成功获得了本问题的所有分数。

```
Average Score: 1006.6
Scores:        1083.0, 1145.0, 889.0, 444.0, 951.0, 991.0, 1132.0, 1252.0, 1277.0, 902.0
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)
***     1006.6 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

### Question q5: 6/6 ###
```

图 6.1: Question6 实验结果

本问题的测试方式如图 6.2所示。该测试样例从三个角度评价评估函数的质量：1）得分高低；2）在规定时间内结束游戏；3）获胜率。如果想要获得所有分数，则吃豆人必须多次吃掉处于虚弱状态下的幽灵。

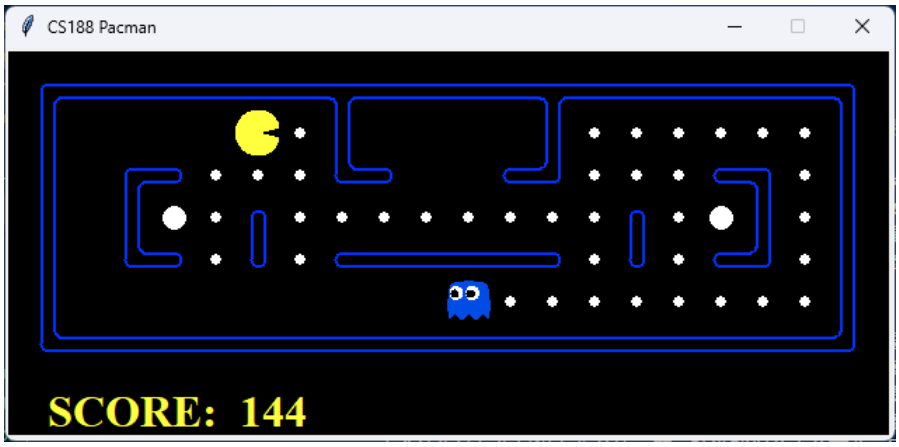

图 6.2: Question6 测试样例

7. 总结与分析

通过本次实验，我深刻地体会到了基于 Minimax、Alpha-Beta 算法和 Expectimax 算法的区别。前两者适用的场景需要满足很强的假设，即每个智能体都执行最优解。但是现实情况下，往往不是这样，如果此时仍然适用这两个算法，则会导致智能体“过于谨慎”，从而表现一般。例如对于图 7.1的场景，如果吃豆人认为幽灵会执行最优解，那么它将必死无疑，此时，为了尽可能地增加分数，最优解是“自杀”，因为这样能减少由于时间带来的扣分。但是，如果这两个幽灵的行为是随机的，又或者甚至会刻意避开吃豆人，那么吃豆人实际上是可以获胜的。Expectimax 算法可以很好的处理这种情况，幽灵的行为特征可以由计算期望时的概率分配来刻画，如果行为是完全随机的，那么各行为的概率相同，如果总是取最优解，那么最优行为的概率总是 1.

图 7.1: 迷你地图

在实现算法时，还需要考虑的是所需的时间和计算量。为了降低开销，常常适用评估函数来进行估计，而非按照算法完成所有计算。虽然评估函数本质上是一个估计，但是只要设计地足够好，便能够在降低开销的同时维持足够好的表现。在设计估计函数时，往往需要多次尝试。在本次实验中，我曾尝试用奖励项来激励吃豆人主动收集能量点并追击幽灵、用惩罚项来让吃豆人避免进入死胡同等较不安全的位置，但事实证明，这些都可以用游戏得分来代替

在实验的过程中，遇到的问题集中在第 5 题。一开始吃豆人在某些情况下会停在最后一个食物旁边，但是一直不移动，直到幽灵靠近后才会开始移动。通过输出日志的方式进行调试后，我发现问题在于当时我所设计的评估函数中，如果判断为获胜，则直接返回 float('inf')，而在本题中，博弈树的深度限制在两层，意味着吃豆人可以立刻移动至最后一处食物，从而游戏获胜，但也可以暂停移动一回合，然后再移动至最后一处食物，而这两种情况在博弈树的第一层中都体现为 float('inf')，如果解决平手的规则选择了暂停移动作为最佳行为，那么在幽灵靠近之前，吃豆人会一直保持静止。最后，我不再单独判断游戏失败或获胜，因为这些都体现在了游戏分数之中，只需要在评估函数中包含游戏分数这一项即可。