同济大学

TONGJI UNIVERSITY

《人工智能原理课程设计》

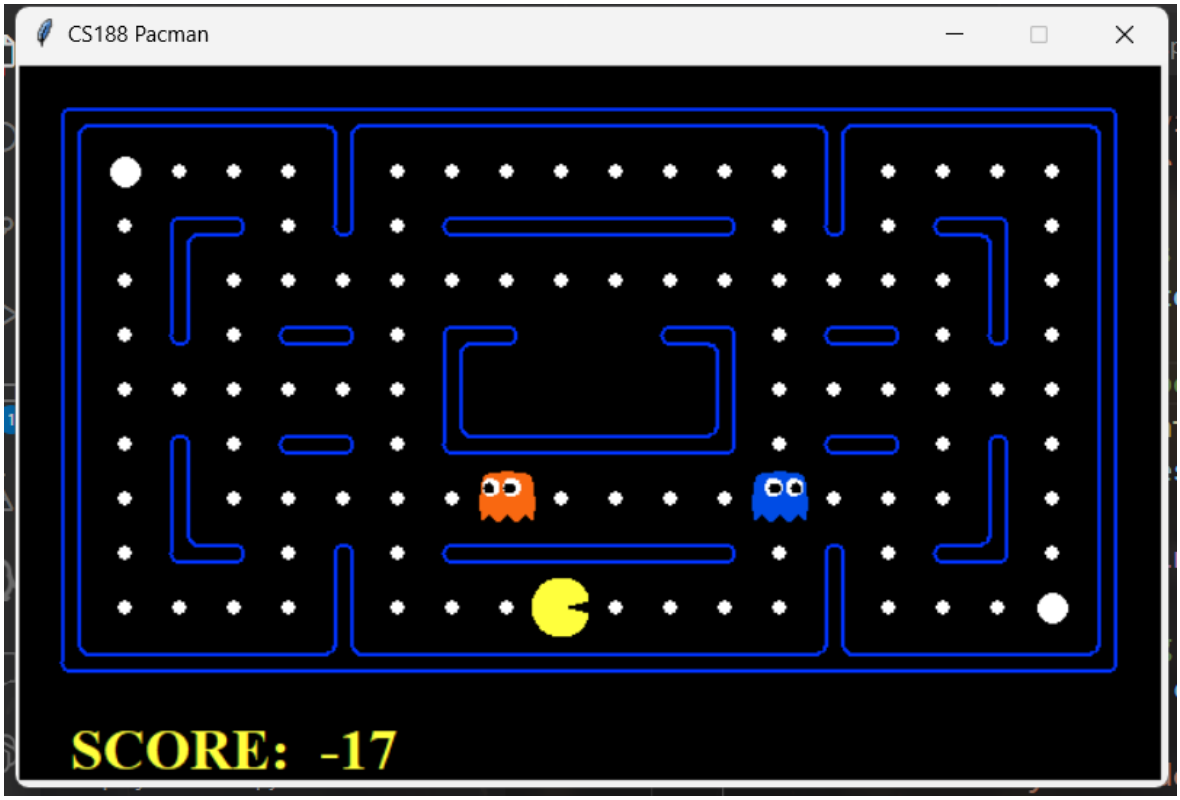
实验报告

实验名称	Project2：Multi-Agent Search
学号姓名	2154057 汪清濯
学院（系）	电子与信息工程学院
专　　业	计算机科学与技术
任课教师	张红云/苗夺谦
日　　期	2023 年 4 月 15 日

Project2：Multi-Agent Search

一、项目简介

在本项目中，我将设计经典的 Pacman 智能体，包括幽灵在内。我需要完成极大极小搜索，alpha-beta 剪枝，最佳期望搜索，以及评估函数的设计等。代码库与项目一没有太大的差别，与项目一相同，项目仍为我们提供了许多有用的参数，合理的调用它们可以很好的测试我们的代码。



装

订

线

二、实验内容

Q1.Reflex Agent——反映智能体

Q1.1 问题概述

Q1.1.1 直观描述

改进 multiAgents.py 中的 ReflexAgent 以获得更好的行动。已提供的 ReflexAgent 代码提供了一些有用的方法示例，这些方法可以查询 GameState 以获取信息。一个优秀的 ReflexAgent 必须同时考虑食物位置和鬼的位置才能表现良好。

Q1.1.2 已有代码的阅读和理解

需要阅读 multiAgents.py 中的 ReflexAgent 类

```
1. class ReflexAgent(Agent)
```

该类继承自 Agent 基类。这个类的目的是为一个具有固定状态评估函数的反射智能体提供一种方法来选择动作。类内有两个成员函数，我们需要完成 **evaluationFunction** 函数。

1. getAction 函数

```
1. def getAction(self, gameState: GameState)
```

参数：gameState：游戏状态

主要功能：函数首先通过 gameState.getLegalActions() 获得所有合法的移动方向，然后使用 evaluationFunction() 对每个动作进行评估，返回一个最好的动作。如果有多个最好的动作，则从中随机选择一个。函数最终返回最好的动作。

Q1.1.3 解决问题的思路与方法

要实现一个良好的评估函数需要更全面的考虑参数以及它们所占的权重，我考虑了食物位置、鬼位置、鬼是否被恐惧等参数，并根据重要程度的不同为它们赋予了不同的权值。

Q1.2 算法设计

Q1.2.1 算法功能

良好完成评估函数，返回对一个行动较准确的评估值。

Q1.2.2 设计思路

先分别找出食物、鬼相对于吃豆人位置的最近曼哈顿距离，离食物越近评估值越高，离鬼越远评估值越高。我在测试中发现，当吃豆人吃掉大能量球后，鬼会进入被恐惧的状态，在被恐惧状态下的鬼速度减慢，并且可以被吃豆人吃掉，分值是普通食物的二十倍，所以被恐惧状态下的鬼非常重要，一定要吃掉它。我设定普通食物的权值为 10，正常鬼权值为-10，被恐惧的鬼权值为 200。

Q1.3 算法实现

下面贴出代码，细节已在注释中标明

```
1.  def evaluationFunction(self, currentGameState: GameState, action):
2.
3.          # 生成 Pacman 的后继状态
4.          successorGameState = currentGameState.generatePacmanSuccessor(action)
5.          # 获取 Pacman 的位置
6.          newPos = successorGameState.getPacmanPosition()
7.          # 获取新的食物网格
8.          newFood = successorGameState.getFood()
9.          # 获取新的鬼状态列表
10.         newGhostStates = successorGameState.getGhostStates()
11.         # 获取每个鬼魂因为 Pacman 吃掉豆子而恐惧的剩余时间
12.         newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates
    ]
13.
14.         "*** YOUR CODE HERE ***"
15.
16.         # 如果后继状态已经赢或输，直接返回得分
```

```
17.         if successorGameState.isWin() or successorGameState.isLose():
18.             return successorGameState.getScore()
19.
20.         # 获取新的食物列表
21.         food_list = newFood.asList()
22.         # 获取离 Pacman 最近的食物距离
23.         nearest_food = min([manhattanDistance(food, newPos) for food in food_list
    ])
24.         # 获取离 Pacman 最近的鬼距离
25.         nearest_ghost = min([manhattanDistance(ghoststate.getPosition(), newPos)
    for ghoststate in newGhostStates])
26.
27.         # 如果鬼被恐惧（吃豆人可以吃它，且得分是食物的十倍）权值为 100，食物权值为 10
28.         if newScaredTimes[0]:
29.             answer = 200/nearest_ghost + 10/nearest_food + successorGameState.get
    Score()
30.         # 否则鬼权值为-10，食物权值为 10
31.         else:
32.             answer = -10/nearest_ghost + 10/nearest_food + successorGameState.get
    Score()
33.
34.         return answer
```

Q1.4 实验结果

所有测试用例均已通过

```
Question q1
===========

Pacman emerges victorious! Score: 1429
Pacman emerges victorious! Score: 1253
Pacman emerges victorious! Score: 1247
Pacman emerges victorious! Score: 1247
Pacman emerges victorious! Score: 1252
Pacman emerges victorious! Score: 1198
Pacman emerges victorious! Score: 1426
Pacman emerges victorious! Score: 1237
Pacman emerges victorious! Score: 1252
Pacman emerges victorious! Score: 1249
Average Score: 1279.0
Scores:        1429.0, 1253.0, 1247.0, 1247.0, 1252.0, 1198.0, 1426.0, 1237.0, 1252.0, 1249.0
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)
***     1279.0 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 ###
```

由上图可知该测试的评判标准是平均得分和胜率。平均得分、胜率越高，给出的总成绩越高。吃掉的食物越多，花费的时间越少则得分越高。吃掉所有食物则游戏胜利。

分别观察有一只鬼与两只鬼时智能体的表现

python pacman.py -p ReflexAgent -k 1 -q -n 10

```
Pacman emerges victorious! Score: 1032
Pacman emerges victorious! Score: 1490
Pacman emerges victorious! Score: 1630
Pacman emerges victorious! Score: 1008
Pacman died! Score: 536
Pacman emerges victorious! Score: 1474
Pacman emerges victorious! Score: 1387
Pacman emerges victorious! Score: 1322
Pacman emerges victorious! Score: 1713
Pacman emerges victorious! Score: 1644
Average Score: 1323.6
Scores:        1032.0, 1490.0, 1630.0, 1008.0, 536.0, 1474.0, 1387.0, 1322.0, 1713.0, 1644.0
Win Rate:      9/10 (0.90)
Record:        Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win
```

只有一只鬼时，智能体表现良好，胜率接近 100%

python pacman.py -p ReflexAgent -k 2 -q -n 10

```
Pacman died! Score: 501
Pacman died! Score: 58
Pacman emerges victorious! Score: 1398
Pacman died! Score: 258
Pacman emerges victorious! Score: 1675
Pacman emerges victorious! Score: 1869
Pacman died! Score: 706
Pacman died! Score: 564
Pacman died! Score: 63
Pacman emerges victorious! Score: 1668
Average Score: 876.0
Scores:        501.0, 58.0, 1398.0, 258.0, 1675.0, 1869.0, 706.0, 564.0, 63.0, 1668.0
Win Rate:      4/10 (0.40)
Record:        Loss, Loss, Win, Loss, Win, Win, Loss, Loss, Loss, Win
```

有两只鬼时，胜率明显下降，不到 50%

再测试更智能的定向鬼

python pacman.py -p ReflexAgent -k 1 -q -n 10 -g DirectionalGhost

```
Pacman emerges victorious! Score: 1509
Pacman emerges victorious! Score: 1549
Pacman died! Score: 503
Pacman died! Score: 38
Pacman emerges victorious! Score: 1622
Pacman emerges victorious! Score: 1695
Pacman died! Score: 47
Pacman emerges victorious! Score: 1713
Pacman died! Score: 569
Pacman died! Score: 487
Average Score: 973.2
Scores:        1509.0, 1549.0, 503.0, 38.0, 1622.0, 1695.0, 47.0, 1713.0, 569.0, 487.0
Win Rate:      5/10 (0.50)
Record:        Win, Win, Loss, Loss, Win, Win, Loss, Win, Loss, Loss
```

python pacman.py -p ReflexAgent -k 2 -q -n 10 -g DirectionalGhost

```
Pacman died! Score: -28
Pacman died! Score: -147
Pacman died! Score: 147
Pacman died! Score: -401
Pacman died! Score: -28
Pacman died! Score: -43
Pacman died! Score: 297
Pacman died! Score: -229
Pacman died! Score: 155
Pacman died! Score: 41
Average Score: -23.6
Scores:        -28.0, -147.0, 147.0, -401.0, -28.0, -43.0, 297.0, -229.0, 155.0, 41.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

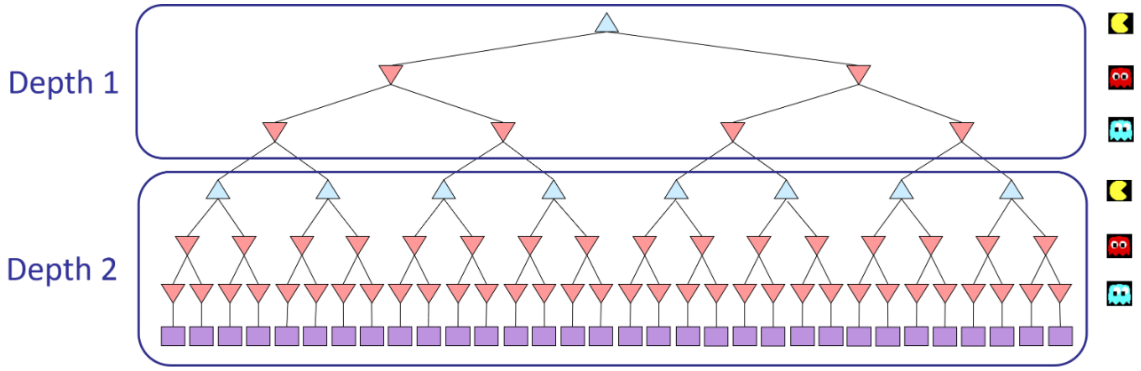
一只和两只鬼的胜率都明显下降，所以我们需要后续搜索更深的搜索算法来改进智能体

Q2. Minimax——极小极大搜索

Q2.1 问题概述

Q2.1.1 直观描述

我需要在 multiAgents.py 提供的 MinimaxAgent 类中编写一个对抗搜索智能体。Minimax 智能体应该能够与任意数量的鬼一起工作。特别的，对于每个最大层，我的极小极大搜索树中应该有多个最小层（每个鬼一个）。我应该能够将博弈树扩展到任意深度。 使用提供的 self.evaluationFunction（默认为 scoreEvaluationFunction）对 minimax 树的叶子进行评分。单个搜索层被认为是吃豆人的一次移动和所有幽灵的反应，因此深度 2 搜索将涉及吃豆人和每个幽灵移动两次



Q2.1.2 已有代码的阅读和理解

需要阅读 pacman.py 中的 GameState 类

GameState 类

```
1. class GameState
```

这个类表示游戏状态，包括食物、智能体配置和分数更改等。它具有以下成员函数

1. getLegalActions 函数

```
1. def getLegalActions(self, agentIndex=0)
```

参数：agentIndex：智能体编号
主要功能：返回指定智能体的合法操作。

2．generateSuccessor 函数

```
1. def generateSuccessor(self, agentIndex, action)
```

参数：agentIndex：智能体编号
action：行动

主要功能：返回指定代理执行操作后的继承状态。

3．getLegalPacmanActions 函数

```
1. def getLegalPacmanActions(self)
```

参数：无

主要功能：返回 Pacman 的合法操作

4．generatePacmanSuccessor 函数

```
1. def generatePacmanSuccessor(self, action)
```

参数：action：行动

主要功能：返回 Pacman 执行操作后的继承状态。

5．getPacmanState 函数

```
1. def getPacmanState(self)
```

参数：无

主要功能：返回 Pacman 的智能体状态对象

6．getPacmanPosition 函数

```
1. def getPacmanPosition(self)
```

参数：无

主要功能：返回 Pacman 位置

7．getGhostStates 函数

```
1. def getGhostStates(self)
```

参数：无

主要功能：返回所有鬼的智能体状态对象的列表

8．getGhostState 函数

```
1. def getGhostState(self, agentIndex)
```

参数：agentIndex：智能体编号

主要功能：返回指定鬼智能体的状态对象

9．getGhostPosition 函数

```
1. def getGhostPosition(self, agentIndex)
```

参数：agentIndex：智能体编号
主要功能：返回指定鬼的位置

10. getGhostPositions 函数

```
1. def getGhostPositions(self)
```

参数：无
主要功能：返回所有鬼的位置列表

11. getNumAgents 函数

```
1. def getNumAgents(self)
```

参数：无
主要功能：返回智能体数量

12. getScore 函数

```
1. def getScore(self)
```

参数：无
主要功能：返回分数

13. getCapsules 函数

```
1. def getCapsules(self)
```

参数：无
主要功能：返回剩余能力球的位置列表

14. getNumFood 函数

```
1. def getNumFood(self)
```

参数：无
主要功能：返回剩余食物的数量

15. getFood 函数

```
1. def getFood(self)
```

参数：无

主要功能：返回 bool 值的食物指示变量的网格

16．getWalls 函数

```
1. def getWalls(self)
```

参数：无

主要功能：返回 bool 值的墙指示变量的网格

17．hasFood 函数

```
1. def hasFood(self, x, y)
```

参数：x，y：横纵坐标

主要功能：返回指定位置是否有食物

18．hasWall 函数

```
1. def hasWall(self, x, y)
```

参数：x，y：横纵坐标

主要功能：返回指定位置是否是强

19．isLose 函数

```
1. def isLose(self)
```

参数：无

主要功能：返回游戏是否已经输了

20．isWin 函数

```
1. def isWin(self)
```

参数：无

主要功能：返回游戏是否已经赢了

Q2.1.3 解决问题的思路与方法

既然要对吃豆人和鬼分别做不同的搜索，不如分别设计 Max 和 Min 两个函数分别代表它们。再分别对不同的深度，智能体编号做特殊的判断。

Q2.2 算法设计

Q2.2.1 算法功能

通过极小极大搜索返回当前的最优行动。

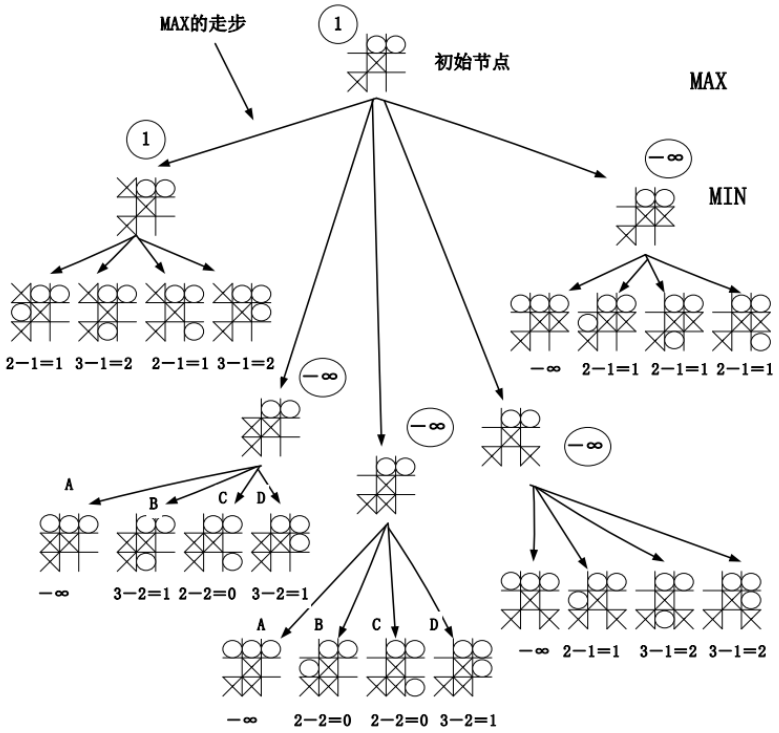
Q2.2.2 设计思路

设计两个函数 MaxStep 和 MinStep 互相递归调用，Max 层对每个后继调用 MinStep 函数，每步搜索完后更新评估分数，取最大值。若是第一层返回最优行动，反之返回评估分数。Min 层先判断当前搜索的是否是最后一只鬼（可能有多只鬼，需要在一个深度中全部搜索），若是，则对每个后继调用 MaxStep 函数。反之继续对每个后继调用 MinStep 函数。每步搜索完后更新评估分数，取最小值。返回评估分数。

时间复杂度：O(b^d)，空间复杂度：O(bd)。b：平均分支数。d：搜索深度。

Q2.2.3 算法流程图

以井字棋为例



Q2.3 算法实现

下面贴出代码，细节已在注释中标明

```
1.  class MinimaxAgent(MultiAgentSearchAgent):
2.      def getAction(self, gameState: GameState):
3.          "*** YOUR CODE HERE ***"
4.          # 返回第一层确定的行动
5.          return self.MaxStep(gameState, 0, 1)
6.          util.raiseNotDefined()
7.
8.      def MaxStep(self, state: GameState, agentIndex, depth):
9.          actions = state.getLegalActions(0)
10.         evascore = -float("inf")  # 评估分数
11.
12.         # 当前状态赢或输或已经搜索到要求的最大深度，直接返回得分
13.         if state.isWin() or state.isLose() or depth == self.depth + 1:
14.             return self.evaluationFunction(state)
15.
16.         # bestaction = ""
```

```
        for action in actions:
            successor = state.generateSuccessor(agentIndex, action)
            # 调用 MinStep
            minstep = self.MinStep(successor, agentIndex + 1, depth)
            # 若大于当前评估分数，则更新评估分数和最佳行动
            if minstep > evascore:
                bestaction = action
                evascore = minstep
        # 深度为 1 返回行动
        if depth == 1:
            return bestaction
        # 否则返回分数
        else:
            return evascore

    def MinStep(self, state: GameState, agentIndex, depth):
        agentnums = state.getNumAgents()
        actions = state.getLegalActions(agentIndex)
        evascore = float("inf")  # 评估分数

        # 当前状态赢或输或已经搜索到要求的最大深度，直接返回得分
        if state.isWin() or state.isLose():
            return self.evaluationFunction(state)

        for action in actions:
            successor = state.generateSuccessor(agentIndex, action)
            # 若鬼智能体还没搜索完，继续搜索鬼
            if agentIndex < agentnums - 1:
                minstep = self.MinStep(successor, agentIndex + 1, depth)
                # 更新评估分数
                if minstep < evascore:
                    evascore = minstep
            # 否则进入下一层，搜索吃豆人的分数
            else:
                maxstep = self.MaxStep(successor, 0, depth + 1)
                # 更新评估分数
                if maxstep < evascore:
                    evascore = maxstep

        return evascore
```

Q2.4 实验结果

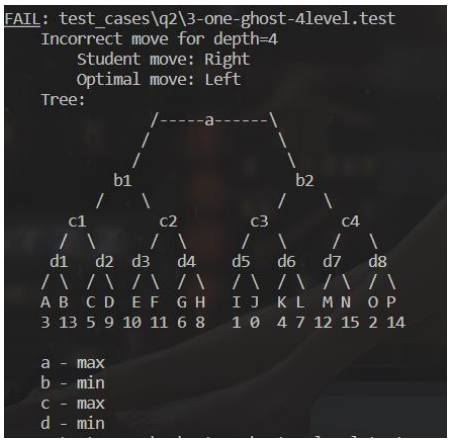
所有测试用例均已通过

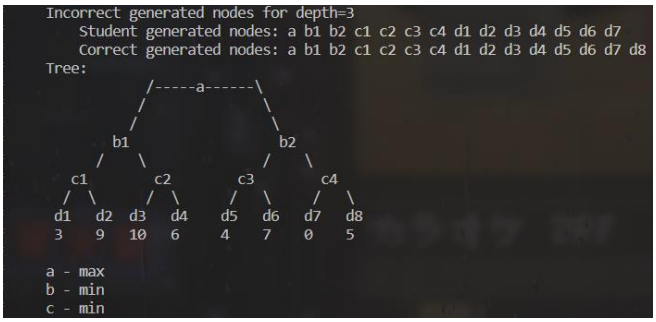
```
*** PASS: test_cases\q2\0-eval-function-lose-states-1.test
*** PASS: test_cases\q2\0-eval-function-lose-states-2.test
*** PASS: test_cases\q2\0-eval-function-win-states-1.test
*** PASS: test_cases\q2\0-eval-function-win-states-2.test
*** PASS: test_cases\q2\0-lecture-6-tree.test
*** PASS: test_cases\q2\0-small-tree.test
*** 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 ###
```

查看测试用例

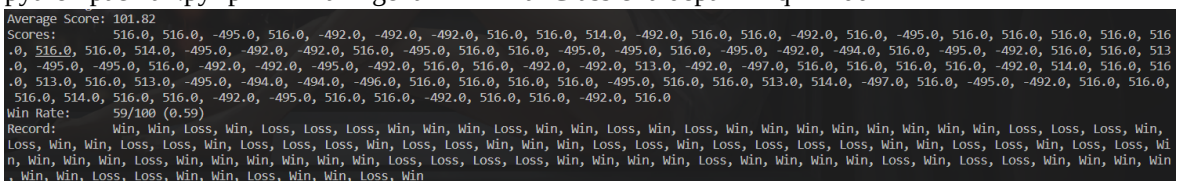




判断依据是行动以及扩展结点是否正确，因为极小极大搜索的每一步都是确定的，所以每一步的最优行动和扩展的结点也必然确定。

测试 100 次 minimaxClassic 下 minmax 智能体的表现

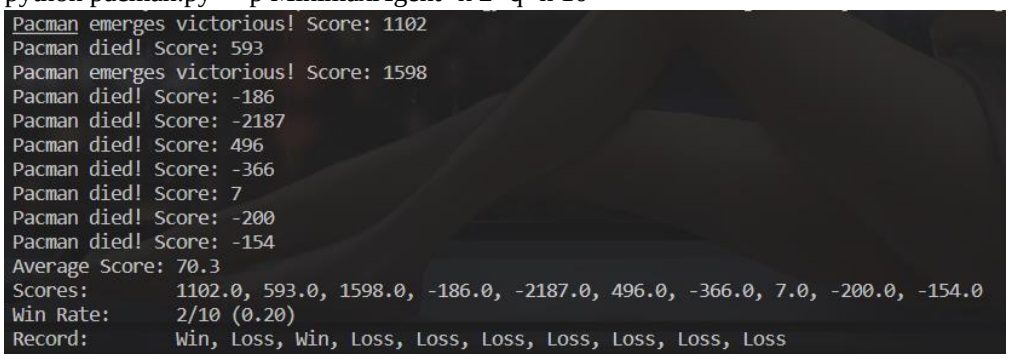
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4 -q -n 100



胜率在 60%左右

与 Q1 对比

python pacman.py -p MinimaxAgent -k 2 -q -n 10



在有两只随机移动的鬼的情况下，胜率仅有 20%，且搜索速度非常慢，这两点分别会在 Q5 和 Q3 中解决。

在 Pacman 认为自己的死亡不可避免时，因为持续生存的惩罚它会尽快尝试结束游戏。这对随机移动的鬼来说有时是错误的做法，但极小极大搜索总是假设最坏的情况，这点将在 Q4 中解决。

Q3. Alpha-Beta Pruning——阿尔法-贝塔剪枝

Q3.1 问题概述

Q3.1.1 直观描述

在 AlphaBetaAgent 中创建一个使用 alpha-beta 剪枝更有效地探索 minimax 树的新智能体。我的代码应该更通用，能适应一个 Max 层下有多个 Min 层的情况。我应该会看到加速（也许深度 3 alpha-beta 的运行速度与深度 2 minimax 一样快）。 理想情况下，smallClassic 上的深度 3 每次移动只需几秒或更快。AlphaBetaAgent 的 minimax 值应该与 MinimaxAgent 的 minimax 值相同，尽管它选择的动作可能会因为不同的打破平局行为而有所不同。

Q3.1.2 已有代码的阅读和理解

无

Q3.1.3 解决问题的思路与方法

在 minmax 的基础上加上剪枝，为函数增加代表 max 层和 min 层最值的参数，每次递归返回时更新，通过这两个参数判断是否需要剪枝。

Q3.2 算法设计

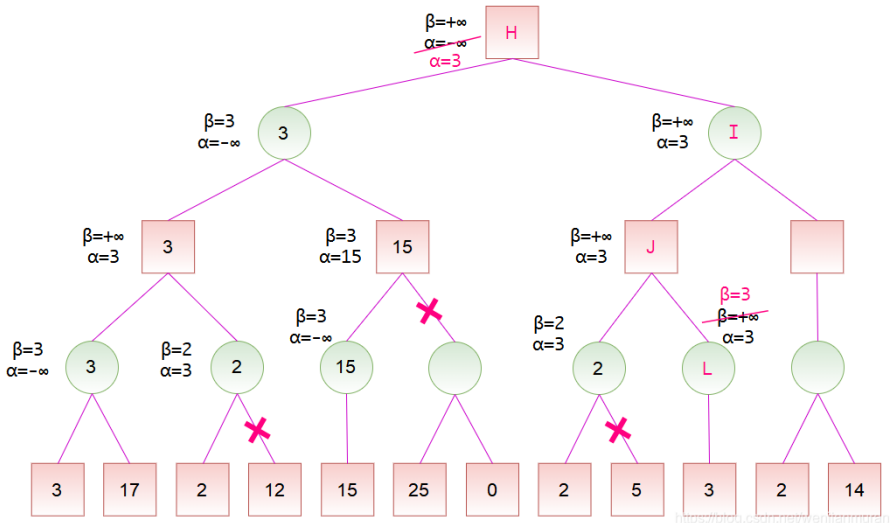
Q3.2.1 算法功能

在 minmax 的基础上完成 alpha-beta 剪枝。

Q3.2.2 设计思路

在 minmax 的基础上加上 maxscore 和 minscore 两个参数。若当前为 max 层，若下一层返回的分数大于评估值，更新评估值。若评估值大于 minscore，进行 beta 剪枝，函数直接返回。反之，若评估值大于 maxscore，更新 maxscore 为评估值。若当前为 min 层，若下一层返回的分数小于评估值，更新评估值。若评估值小于 maxscore，进行 alpha 剪枝，函数直接返回。反之，若评估值小于 minscore，更新 minscore 为评估值。

Q3.2.3 算法流程图



Q3.3 算法实现

下面贴出代码，细节已在注释中标明

```
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 ***"
        return self.MaxStep(gameState, 0, 1, -float("inf"), float("inf"))
        util.raiseNotDefined()

    def MaxStep(self, state: GameState, agentIndex, depth, maxscore, minscore):
        actions = state.getLegalActions(0)
        evascore = -float("inf")  # 评估分数

        # 当前状态赢或输或已经搜索到要求的最大深度，直接返回得分
        if state.isWin() or state.isLose() or depth == self.depth + 1:
            return self.evaluationFunction(state)

        # bestaction = ""
        for action in actions:
            successor = state.generateSuccessor(agentIndex, action)
            # 搜索 min 层
            minstep = self.MinStep(successor, agentIndex + 1, depth, maxscore, mi
nscore)
            # 若 min 层返回的值大于评估值，更新
            if minstep > evascore:
                bestaction = action
                evascore = minstep
                # 若评估值大于 minscore，beta 剪枝
                if evascore > minscore:
                    return evascore
                # 若评估值大于 maxscore，更新 maxscore
                if evascore > maxscore:
                    maxscore = evascore

        # 深度为 1 返回行动
        if depth == 1:
            return bestaction
        # 否则返回分数
        else:
            return evascore

    def MinStep(self, state: GameState, agentIndex, depth, maxscore, minscore):
        agentnums = state.getNumAgents()
        actions = state.getLegalActions(agentIndex)
        evascore = float("inf")  # 评估分数

        # 当前状态赢或输或已经搜索到要求的最大深度，直接返回得分
        if state.isWin() or state.isLose() or depth == self.depth + 1:
            return self.evaluationFunction(state)

        for action in actions:
            successor = state.generateSuccessor(agentIndex, action)
            # 若鬼还未搜索完，继续搜索鬼
            if agentIndex < agentnums - 1:
                # 搜索 min 层
                minstep = self.MinStep(
                    successor, agentIndex + 1, depth, maxscore, minscore
                )
                if minstep < evascore:
```

```
                    evascore = minstep
                    # 若评估值小于 maxscore，alpha 剪枝
                    if evascore < maxscore:
                        return evascore
                    # 若评估值小于 minscore，更新 minscore
                    if evascore < minscore:
                        minscore = evascore
            # 否则进入下一层，搜索 max 层
            else:
                maxstep = self.MaxStep(successor, 0, depth + 1, maxscore, minscor
    e)
                if maxstep < evascore:
                    evascore = maxstep
                    # 若评估值小于 maxscore，alpha 剪枝
                    if evascore < maxscore:
                        return evascore
                    # 若评估值小于 minscore，更新 minscore
                    if evascore < minscore:
                        minscore = evascore

        return evascore
```

装

订

线

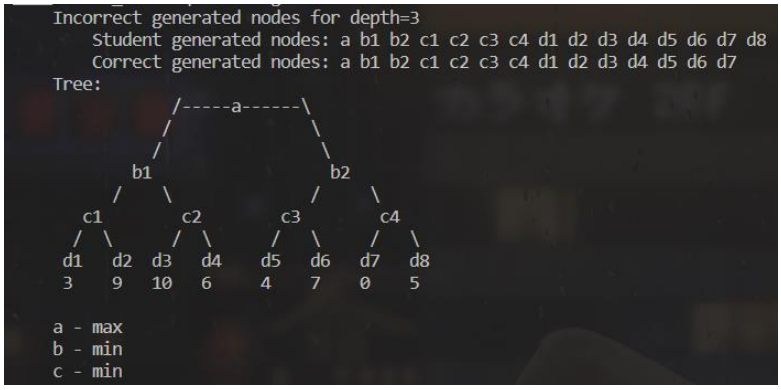
Q3.4 实验结果

所有测试用例均已通过

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

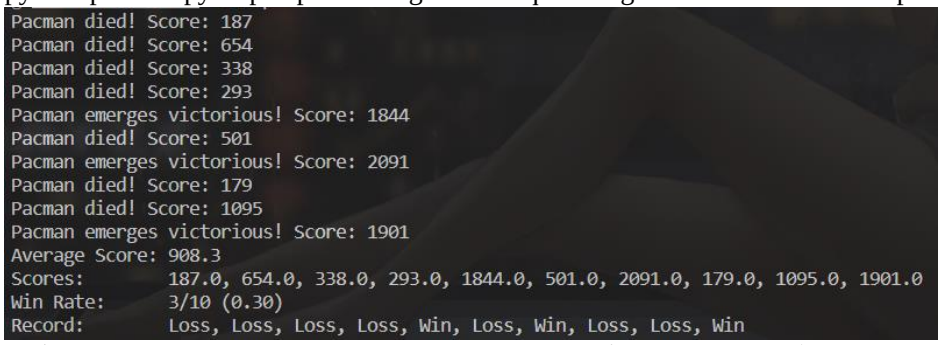
查看测试用例



与 Q2 的地图相同，判断依据也是行动以及扩展的结点是否正确，但此处需要按照 alpha-beta 的要求扩展。Alpha-beta 剪枝的每一步都是确定的，所以每一步的最优行动和扩展结点也是确定的。AlphaBetaAgent 在迷宫中的表现与 minmax 相同，但搜索速度要块很多。

因为搜索速度变快，我们尝试在加深深度的情况下测试 AlphaBetaAgent 面对更智能的定向鬼时的表现

python pacman.py -p AlphaBetaAgent -k 2 -q -n 10 -g DirectionalGhost -a depth=3



胜率达到 30%，优于 Q1（0%），所以 minmax 搜索也许更适用于能做出最优解的对手

Q4. Expectimax——期望最大

Q4.1 问题概述

Q4.1.1 直观描述

Minimax 和 alpha-beta 都很好，但它们都假设我正在与做出最佳决策的对手对战。 任何赢过井字棋的人都会告诉我，情况并非总是如此。 在本题中，我需要实现 ExpectimaxAgent，它可用于对可能做出次优选择的智能体的概率行为进行建模。随机移动的鬼的行动当然不总是极小极大智能体判断的行动，因此用极小极大搜索对它们建模可能不合适。 ExpectimaxAgent 将不再对所有鬼动作取最小值，而是根据我的智能体模型返回对鬼的行为的期望。

Q4.1.2 已有代码的阅读和理解

无

Q4.1.3 解决问题的思路与方法

依然设计两个函数代表吃豆人和鬼，但不再叫它们 max 和 min，吃豆人函数依然返回最优行动或评估分数，但鬼函数返回后续所有得分的期望。

Q4.2 算法设计

Q4.2.1 算法功能

实现一个根据鬼行动期望做出移动判断的智能体。

Q4.2.2 设计思路

设计一个 Pacmanstep 函数代表吃豆人，功能几乎与 minmax 的 maxstep 相同，搜索所有后继的得分，找到最佳行动和最高分数。再设计一个 Ghoststep 函数代表吃豆人，搜索所有后继的得分并返回它们的平均值（期望）。

Q4.3 算法实现

下面贴出代码，细节已在注释中标明

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

    def getAction(self, gameState: GameState):oves.
        """
        "*** YOUR CODE HERE ***"
        return self.Pacmanstep(gameState,1,0)
        util.raiseNotDefined()

    def Pacmanstep(self, state:GameState, depth, agentIndex = 0):
        actions = state.getLegalActions(agentIndex)

        # 当前状态赢或输或已经搜索到要求的最大深度，直接返回得分
        if depth == self.depth + 1 or state.isWin() or state.isLose():
            return self.evaluationFunction(state)

        evascore = -float('inf')
        for action in actions:
            successor = state.generateSuccessor(agentIndex,action)
            # 搜索鬼的下一步
            ghoststep = self.Ghoststep(successor,depth,1)
            # 更新评估分数，最佳行动
            if ghoststep > evascore:
                bestaction = action
                evascore = ghoststep

        if depth == 1:
            return bestaction
        else:
            return evascore


    def Ghoststep(self, state:GameState, depth, agentIndex = 1):
        actions = state.getLegalActions(agentIndex)
        agentnums = state.getNumAgents()

        # 当前状态赢或输或已经搜索到要求的最大深度，直接返回得分
        if depth == self.depth + 1 or state.isWin() or state.isLose():
            return self.evaluationFunction(state)

        evascore = 0
        for action in actions:
            successor = state.generateSuccessor(agentIndex,action)
```

```
                # 若鬼还未搜索完，继续搜索鬼
                if agentIndex < agentnums - 1:
                    evascore += self.Ghoststep(successor,depth,agentIndex+1)
                # 否则进入下一层，搜索吃豆人
                else:
                    evascore += self.Pacmanstep(successor,depth + 1, 0)

            # 返回得分期望
            return evascore / len(actions)
```

Q4.4 实验结果

所有测试用例均已通过

```
*** 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 14 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q4\7-pacman-game.test

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

查看测试用例

```
    Incorrect move for depth=4
        Student move: Left
        Optimal move: Right
    Tree:
                /-----a------\
               /             \
              /               \
            b1                b2
           /   \             /   \
         c1     c2         c3     c4
        /  \    /  \       /  \   /  \
       d1  d2  d3  d4     d5  d6 d7  d8
      /\  /\  /\  /\     /\  /\ /\  /\
     A B  C D  E F  G H   I J  K L M N  O P
     3 13 5 9 10 11 6 8   1 0  4 7 12 15 2 14

    a - max
    b - exp
    c - max
    d - exp
```

和 Q2、Q3 基本相同，判断依据是是否选择了正确的行动

测试 100 次 ExpectimaxAgent 在 minimaxClassic 下的表现，搜索深度为 3

python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3 -q -n 100

```
Average Score: 218.04
Scores:        -498.0, 499.0, 513.0, -495.0, 511.0, 511.0, 492.0, 513.0, -495.0, 511.0, 508.0, 513.0, 515.0, -512.0, 514.0, -494.0, 507.0, 515.0, 503.0, 512.0, -498.
0, 512.0, 510.0, 511.0, 510.0, 510.0, -497.0, -494.0, 511.0, -494.0, 507.0, 512.0, 513.0, 513.0, -494.0, 508.0, -497.0, 510.0, -494.0, -511.0, 506.0, 510.0, -494.0,
507.0, -516.0, 513.0, 513.0, 511.0, 511.0, -504.0, -495.0, -506.0, 515.0, 514.0, 515.0, -503.0, 511.0, -497.0, 511.0, 511.0, 515.0, -495.0, 508.0, 508.0, 511.0, 512.
0, 515.0, 511.0, 513.0, -495.0, 507.0, 511.0, 511.0, 512.0, 513.0, 500.0, 511.0, -494.0, 510.0, 512.0, -497.0, -501.0, 515.0, 508.0, 511.0, 511.0, -502.0, 515.0, 512
.0, 515.0, 511.0, 512.0, 511.0, -497.0, 513.0, -495.0, 515.0, 511.0, 511.0, -495.0
Win Rate:      71/100 (0.71)
Record:        Loss, Win, Win, Loss, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Loss, Win, Loss, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Loss, Loss, Wi
n, Loss, Win, Win, Win, Win, Loss, Win, Loss, Win, Loss, Loss, Win, Win, Loss, Win, Loss, Win, Win, Win, Win, Loss, Loss, Loss, Win, Win, Win, Loss, Win, Loss, Win,
Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Loss, Loss, Win, Win, Win, Win, Loss, Win, Win, Win, Win,
 Win, Win, Loss, Win, Loss, Win, Win, Win, Loss
```

胜率为 71%，均分为 218.04，都要高于深度为 4 的 MinmaxAgent（胜率 59%，均分 101.82）

对比 ExpectimaxAgent 和 AlphaBetaAgent 在陷阱地图下的表现

python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10

```
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
```

python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10

```
Pacman died! Score: -502
Pacman died! Score: -502
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Average Score: -88.4
Scores:        -502.0, -502.0, -502.0, 532.0, -502.0, 532.0, -502.0, 532.0, -502.0, 532.0
Win Rate:      4/10 (0.40)
Record:        Loss, Loss, Loss, Win, Loss, Win, Loss, Win, Loss, Win
```

AlphaBetaAgent 总是死亡而 ExpectimaxAgent 有 40%的几率成功

测试 ExpectimaxAgent 在面对可定向移动的更智能的鬼时的表现

python pacman.py -p ExpectimaxAgent -k 2 -q -n 10 -g DirectionalGhost -a depth=3

```
Pacman died! Score: 344
Pacman died! Score: 268
Pacman died! Score: 380
Pacman died! Score: 229
Pacman died! Score: 789
Pacman died! Score: 875
Pacman emerges victorious! Score: 1863
Pacman died! Score: 470
Pacman emerges victorious! Score: 1875
Pacman died! Score: 933
Average Score: 802.6
Scores:        344.0, 268.0, 380.0, 229.0, 789.0, 875.0, 1863.0, 470.0, 1875.0, 933.0
Win Rate:      2/10 (0.20)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Win, Loss, Win, Loss
```

略差于 AlphaBetaAgent 在 Q3 中的测试

综上，我们可以认为 ExpectimaxAgent 更适合随机移动的鬼，而 AlphaBetaAgent（MinmaxAgent）更适合能做出最优解的鬼。

Q5. Evaluation Function——评估函数

Q5.1 问题概述

Q5.1.1 直观描述

在提供的函数 betterEvaluationFunction 中为 Pacman 编写一个更好的评估函数。使用深度为2的搜索，我的评估函数应该在超过一半的时间内清除带有一个随机幽灵的 smallClassic 布局，并且仍然以合理的速度运行。

Q5.1.2 已有代码的阅读和理解

无

Q5.1.3 解决问题的思路与方法

几乎与 Q1 相同，我考虑了食物位 置、鬼位置、鬼是否被恐惧等参数，并根据重要程度的不同为它们赋予了不同的权值。

Q5.2 算法设计

Q5.2.1 算法功能

良好完成评估函数，返回对一个行动较准确的评估值。

Q5.2.2 设计思路

几乎与 Q1 相同。先分别找出食物、鬼相对于吃豆人位置的最近曼哈顿距离，离食物越近评估值越高，离鬼越远评估值越高。当吃豆人吃掉大能量球后，鬼会进入被恐惧的状态，在被恐惧状态下的鬼速度减慢，并且可以被吃豆人吃掉，分值是普通食物的二十倍，所以被恐惧状态下的鬼非常重要，一定要吃掉它。我设定普通食物的权值为 10，正常鬼权值为-10，被恐惧的鬼权值为 200。

Q5.3 算法实现

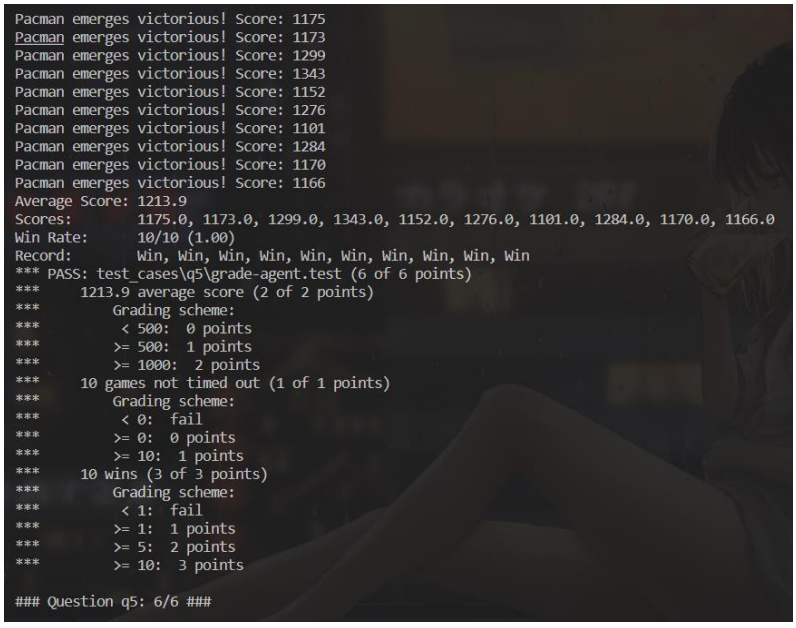
下面贴出代码，细节已在注释中标明

```
1.  def betterEvaluationFunction(currentGameState: GameState):
2.      "*** YOUR CODE HERE ***"
3.      # 当前吃豆人的位置
4.      curPacmanpos = currentGameState.getPacmanPosition()
5.      # 当前食物
6.      curFood = currentGameState.getFood()
7.      # 当前鬼位置
8.      curGhostspos = currentGameState.getGhostPositions()
9.      # 当前鬼被恐惧时间
10.     curScaredTimes = [ghostState.scaredTimer for ghostState in currentGameState.g
    etGhostStates()]
11.
12.     if currentGameState.isWin() or currentGameState.isLose():
13.         return currentGameState.getScore()
14.
15.     food_list = curFood.asList()
16.     # 最近的食物
17.     nearestfood = min(manhattanDistance(curPacmanpos,foodpos) for foodpos in food
    _list)
18.     # 最近的鬼
19.     nearestghost = min(manhattanDistance(curPacmanpos,curGhostpos) for curGhostpo
    s in curGhostspos)
20.
21.     score = 0
22.     # 如果鬼被恐惧（吃豆人可以吃它，且得分是食物的二十倍）权值为 200，食物权值为 10
23.     if curScaredTimes[0]:
24.         score = 200 / nearestghost + 10 / nearestfood + currentGameState.getScore
    ()
25.     # 否则鬼权值为-10
26.     else:
27.         score = - 10 / nearestghost + 10 / nearestfood + currentGameState.getScor
    e()
```

```
28.
29.     return score
```

Q5.4 实验结果

所有测试用例均已通过



评判标准是平均得分和胜率。平均得分、胜率越高，给出的总成绩越高。吃掉的食物越多，花费的时间越少则得分越高。吃掉所有食物则游戏胜利。

三、总结与分析

本项目包括四个主要的算法和技术，分别是迭代深化极小化搜索算法（Iterative Deepening Minimax Search）、Alpha-Beta 剪枝算法、期望最大算法（Expectimax Algorithm）和评估函数的设计。这些算法和技术在实现多智能体协作和竞争问题方面发挥着重要作用。以下是对这些算法和技术的详细总结与分析：

1. 极小极大搜索算法

极小极大搜索算法是一种基于深度优先搜索的博弈树搜索算法，其核心思想是通过极小化搜索和极大化搜索交替进行，以找到最优解。迭代深化极小化搜索算法具有以下优点：

- 可以在有限时间内找到最优解，即使搜索空间非常大。
- 可以在搜索过程中逐步增加搜索深度，从而更好地适应不同的搜索空间。

但是，极小极大搜索算法也存在以下缺点：

- 搜索时间可能会很长，尤其是在搜索空间非常大时。
- 可能会出现搜索树重复的情况，从而导致搜索效率降低。
- 对于某些搜索空间，可能无法找到最优解。

2. Alpha-Beta 剪枝算法

Alpha-Beta 剪枝算法是一种基于极小极大搜索算法的博弈树剪枝算法，其核心思想是通过剪枝操作来减少搜索空间，从而找到最优解。Alpha-Beta 剪枝算法具有以下优点：

- 可以在有限时间内找到最优解，即使搜索空间非常大。
- 可以减少搜索空间，从而提高搜索效率。
- 可以在搜索过程中及时停止搜索，以节省时间和资源。

但是，Alpha-Beta 剪枝算法也存在以下缺点：

- 对于某些搜索空间，可能无法找到最优解。
- 可能会出现搜索树重复的情况，从而导致搜索效率降低。

3. 期望最大算法

期望最大算法是一种基于概率的博弈树搜索算法，其核心思想是通过计算期望值来找到最优解。期望最大算法具有以下优点：

- 可以处理随机性较高的问题，例如扑克牌游戏。
- 可以在有限时间内找到最优解，即使搜索空间非常大。

但是，期望最大算法也存在以下缺点：

- 复杂度较高，需要在搜索树上进行大量的递归计算。特别是在对深度较大的搜索树进行搜索时，计算量会呈指数级增长，导致算法效率较低。
- 对搜索树的假设比较强，要求搜索树是完全平衡的；算法的结果可能受到随机因素的影响，导致结果不稳定；评估函数的设计需要考虑各种因素，包括随机性、不确定性等，增加了评估函数的设计和调试的难度。

在随机性较强的问题中期望最大算法也许更合适，而在对手更有可能做出最优解的问题中，极小极大搜索也许是更好的选择。

在设计评估函数时，我们需要考虑游戏状态的特征，以便对当前状态进行评估和比较。在本项目中，我们采用了基于线性加权和的评估函数，其中权重系数是通过手动调整和实验得出的。在实际应用中，评估函数的设计需要结合具体问题进行优化和调整。

综上，本项目涉及到多个算法和概念。这些算法和概念在博弈论和人工智能领域有着广泛的应用，如在围棋、象棋、扑克等游戏中的决策、机器人路径规划等领域。通过参与该项目，我深刻理解了这些算法的原理和实现细节，加深了对博弈论和人工智能领域的理解和认识。