**同济大学计算机系**

**人工智能课程设计综合实验报告2**

****

**实验名称 Multi-Agent Search**

**学 号**

**姓 名**

**专 业 计算机科学与技术**

**授课老师 武妍老师**

**完成日期 2024年5月14日**

1. 项目概述与分析

**1、项目的总体概述**

吃豆人生活在一个闪亮的蓝色世界里，那里有蜿蜒的走廊和美味的圆形食物，我们需要帮助吃豆人有效率地到达某个地方或者吃掉食物并且避开危险。

本项目主要是为经典版本的吃豆人设计代理，包括幽灵。在此过程中，您将实现 minimax 和 expectimax 搜索，并尝试评估函数设计。

**2、项目已有代码分析**

****

**multiAgents.py:**这个文件里包含了实现多智能体搜索的各种类和函数，以协助实现这些类的一些方法；

class MultiAgentSearchAgent(Agent):使用指定的搜索算法解决一个指定的多智能体搜索问题。

class ReflexAgent(MultiAgentSearchAgent):包含了一个基本的反射型智能体，通过考虑食物位置和幽灵位置来改进其性能。

class MinimaxAgent(MultiAgentSearchAgent):实现了使用Minimax算法的多智能体搜索代理，能够处理任意数量的幽灵，并扩展搜索树到任意深度。

class AlphaBetaAgent(MinimaxAgent):实现了使用Alpha-Beta剪枝的多智能体搜索代理，以更高效地探索搜索树。

class ExpectimaxAgent(MultiAgentSearchAgent):实现了使用Expectimax算法的多智能体搜索代理，用于建模可能采取次优选择的智能体的概率行为。

**util.py:**包含了实现各种搜索算法可能用到的几种数据结构

class Stack:包含栈压入弹出判断为空的函数

class Queue:包含队列压入弹出判断为空的函数

class PriorityQueue:包含优先队列的压入弹出更新判断为空的函数

项目已有代码包括核心的multiAgents算法部分以及游戏的主文件pacman部分，同时还包括一些便于我们编写算法的工具。

**3、子问题概述及简要分析**

**Q1：Reflex Agent**

某个行为的好坏取决于该行为导致的下一个状态，这应当和豆子的剩余数量，距离豆子的距离，吃豆人和鬼的距离等因素决定，豆子剩余数量越少，距离豆子距离越近，距离鬼的距离越远，评分就应当越高。

Q1需要设计一个评估函数，对一个给定状态可能的行动进行评分，越好的行动应当赋予它越高的分数值。

**Q2：Minimax**

从当前状态开始，递归地扩展最大最小搜索树，使用评估函数评估叶子结点，再根据是 min 结点或者 max 结点从底层向顶层确定其他结点的值，最后返回当前状态的最值以及取得该最值的对应行动。

Q2需要设计一个函数，能够从游戏当前的状态开始扩展指定层数的最大最小搜索树，用给定的 evaluationFunction 函数评估叶子结点的值，并根据是 max 层还是 min 层确定叶子结点之外的结点的值，最后返回根节点的值并返回使得根节点取得最值的对应行动。

**Q3：Alpha-Beta Pruning**

在问题 2 的基础上除叶子结点的每个结点都对应 alpha 和 beta 值，如果 min 结点的无法更新父辈结点的 alpha 值或者 max 结点无法更新父辈结点的 beta 值，则无需再访问该节点的其他子节点(剪掉)

Q3需要在Q2 的基础上应用α-β剪枝算法，将不用扩展的结点剪除，提高搜索的效率

**Q4：Expectimax**

由于鬼是随机选择可能的行动，因此 min 结点的评估值不该是其最小的子节点的值，由于是均匀随机选择，所以我选择将其所有子节点值的平均值作为该 min 结点的评估值。

Minimax 和 alpha-beta 算法是假设对手始终会选择最优的对策，但是实际情况往往不是这样的，因此上述两种算法有时候会导致游戏失败和分数更低，Q4需要设计一个搜索算法能够更好的应对这样一种情况：对手不是总选择最优的行动，而是在可能的行动中等可能随机选择。

**Q5：Evaluation Function**

该问题和问题 1 由相似之处，使评估函数更加优秀，我添加了当前的分数，周围墙的数量，随机数，并将最短的豆子的距离取代为：最近的豆子距离 min，距离该豆子最远的豆子两个豆子之间的距离 max，这两个值的和 min+max，用以上量来对某个状态进行评估。

Q5需要设计一个评估函数对某一个状态进行评估而不是对某一个状态的的行为进行评估，并使其满足给定的要求。

1. 算法设计与实现

**Q1：Reflex Agent**

为了使吃豆人能够躲避鬼，应当将鬼距离吃豆人的距离考虑在内

为了使吃豆人能够吃到豆子，应当将最近的豆子的距离和剩余豆子的数量考虑在内

当鬼的距离不足以构成威胁时，主要目标是吃掉豆子，当鬼的距离已经构成威胁时，应当以躲避鬼为主要目标

在躲避鬼时，可能有多种方案，应当从中选择利于吃到豆子的

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]

        "\*\*\* YOUR CODE HERE \*\*\*"

        ghost\_distances = [manhattanDistance(newPos, ghostState.getPosition()) for ghostState in newGhostStates]

        min\_ghost\_distance = min(ghost\_distances) if ghost\_distances else float('inf')

        food\_distances = [manhattanDistance(newPos, food) for food in newFood.asList()]

        min\_food\_distance = min(food\_distances) if food\_distances else float('inf')

        if min\_ghost\_distance <= 2:

            evaluation\_score = -min\_ghost\_distance \* 100

            if not newFood.asList():

                evaluation\_score += 10

            else:

                evaluation\_score += 1.0 / min\_food\_distance

            evaluation\_score -= 10000

        else:

            if not newFood.asList():

                evaluation\_score = 10000

            else:

                evaluation\_score = -len(newFood.asList()) \* 100 + 100.0 / min\_food\_distance

        return evaluation\_score

用最近的豆子的距离模拟豆子的距离

当鬼和吃豆人的距离小于等于 2 时，主要通过鬼的距离进行评估，次要通过豆子的数量和位置评估；当鬼和吃豆人的距离大于 2 时，不考虑鬼的影响，只通过豆子的信息进行评估

保证大于 2 的评估值总是大于小于等于 2 的评估值

**Q2：Minimax**

由于子节点的值也需要调用该函数，所以是一个递归的结构。

根据当前树的层数和要求扩展的深度来判断是 min 结点还是 max 结点

def getAction(self, gameState: GameState):

        "\*\*\* YOUR CODE HERE \*\*\*"

        numAgents = gameState.getNumAgents()

        value, action = self.minimaxSelect(gameState, numAgents, 0)

        return action

    def minimaxSelect(self, gameState: GameState, numAgents, layer):

        if layer == numAgents \* self.depth or gameState.isWin() or gameState.isLose():

            return [self.evaluationFunction(gameState), 'Stop']  # 递归终止条件

        player\_index = layer % numAgents

        if player\_index == 0:  # 当前玩家是Max玩家

            best\_value = float('-inf')

            best\_action = 'Stop'

            for action in gameState.getLegalActions(player\_index):

                successor\_state = gameState.generateSuccessor(player\_index, action)

                value, \_ = self.minimaxSelect(successor\_state, numAgents, layer + 1)

                if value > best\_value:

                    best\_value = value

                    best\_action = action

            return [best\_value, best\_action]

        else:  # 当前玩家是Min玩家

            best\_value = float('inf')

            best\_action = 'Stop'

            for action in gameState.getLegalActions(player\_index):

                successor\_state = gameState.generateSuccessor(player\_index, action)

                value, \_ = self.minimaxSelect(successor\_state, numAgents, layer + 1)

                if value < best\_value:

                    best\_value = value

                    best\_action = action

            return [best\_value, best\_action]

扩展的树的深度等于要求扩展的深度与智能体数量的乘积，所以递归结束的条件就是树的深度等于要求深度与智能体数量的乘积

通过树的层数来模智能体的数量来判断是哪个智能体对应的状态，依次判断是min 结点还是 max 结点

**Q3：Alpha-Beta Pruning**

在算法 2 的基础上将父辈 max 结点对应的 alpha 值列表和父辈 min 结点对应的beta 值列表作为参数传入函数，如果当前 min 结点已经不能再改变父辈的 alpha 值或者当前max 结点已经不能再改变父辈的 beta 值，则不再扩展该结点的剩余结点，实现剪枝

def getAction(self, gameState: GameState):

        "\*\*\* YOUR CODE HERE \*\*\*"

        numAgents = gameState.getNumAgents()

        value = self.alphaBetaSelect(gameState, 0, numAgents, [-float('inf')], [float('inf')])

        return value[1]

    def alphaBetaSelect(self, gameState: GameState, layer, numAgents, alphaList, betaList):

        if gameState.isLose() or gameState.isWin() or layer == numAgents \* self.depth:

            return [self.evaluationFunction(gameState), 'Stop']

        alpha = -float('inf')

        alphaAction = 'Stop'

        beta = float('inf')

        betaAction = 'Stop'

        legalActions = gameState.getLegalActions(layer % numAgents)

        for action in legalActions:

            if layer % numAgents == 0:

                alphaList.append(alpha)

            else:

                betaList.append(beta)

            successorState = gameState.generateSuccessor(layer % numAgents, action)

            temp = self.alphaBetaSelect(successorState, layer + 1, numAgents, alphaList, betaList)

            if layer % numAgents == 0:

                if temp[0] > alpha:

                    alpha = temp[0]

                    alphaAction = action

            else:

                if temp[0] < beta:

                    beta = temp[0]

                    betaAction = action

            if layer % numAgents == 0:

                alphaList.pop()

            else:

                betaList.pop()

            flag = 0

            if layer % numAgents != 0:

                for x in alphaList:

                    if beta < x:

                        flag = 1

                        break

            else:

                for x in betaList:

                    if alpha > x:

                        flag = 1

                        break

            if flag == 1:

                break

        if layer % numAgents == 0:

            return [alpha, alphaAction]

        else:

            return [beta, betaAction]

将父辈结点的 alpha\beta 值存放在列表中作为参数传入函数，当子节点无法更新父辈结点的值时，就不再扩展该结点的子节点

更新结点的时机是子节点返回值时，然后更新列表里的值，用于剪枝

**Q4：Expectimax**

由于鬼的行动是随机的，因此鬼并不一定向扩展出的最不利于吃豆人的状态行动，而是所有行动都可能采取，并且等可能，所以用 min 结点的子节点的平均值作为 min 结点的值。

def getAction(self, gameState: GameState):

        "\*\*\* YOUR CODE HERE \*\*\*"

        numAgents = gameState.getNumAgents()

        value = self.expectimaxSelect(gameState, numAgents, 0)

        return value[1]

    def expectimaxSelect(self, gameState: GameState, numAgents, layer):

        actionChoice = '停止'

        if gameState.isLose():

            return [self.evaluationFunction(gameState), '停止']

        if gameState.isWin():

            return [self.evaluationFunction(gameState), '停止']

        if layer == numAgents \* self.depth:

            return [self.evaluationFunction(gameState), actionChoice]

        else:

            maxVal = float('-inf')

            maxAction = '停止'

            avgVal = 0

            legalActions = gameState.getLegalActions(layer % numAgents)

            actionCount = len(legalActions)

            for action in legalActions:

                legalState = gameState.generateSuccessor(layer % numAgents, action)

                value = self.expectimaxSelect(legalState, numAgents, layer + 1)

                if maxVal < value[0]:

                    maxVal = value[0]

                    maxAction = action

                avgVal += value[0]

            avgVal /= actionCount  # 如果是最小结点，则该结点的值为子结点值的平均值

            if layer % numAgents == 0:

                return [maxVal, maxAction]

            else:

                return [avgVal, actionChoice]

因为鬼是均匀随机的选择可能的行动，因此在评估原来 minmax 树中 min 结点的值时不能够再选取最小值，而是应该取所有子节点的平均值（等可能）

**Q5：Evaluation Function**

为了能够更好的评估一个游戏状态，需要将以下内容考虑在内：

1、鬼和吃豆人的距离，因为只要一个鬼碰到吃豆人就代表游戏失败，因此用和吃豆人距离最近的鬼的距离来代表和鬼的距离，该距离越近，游戏状态的评分应当越低

2、豆的数量和与吃豆人的距离，因为吃豆人的目标是将所有豆子都吃完，所以豆子的数量越少，吃豆人距离豆子的距离越近，状态评分应当越高。为了相对于问题 1 更好的评估和豆子的距离，我将最近的豆子距离 min，距离该豆子最远的豆子两个豆子之间的距离 max，这两个值的和 min+max，来代表和吃豆人和豆子之间的距离

3、墙的位置，如果一个状态是吃豆人的三面有墙，则它不得不返回之前的位置，如果这个位置没有豆子，那么这就是无效行动，我将这种状态的评分给的很低以避免吃豆人陷入反复的来回运动

def betterEvaluationFunction(currentGameState: GameState):

    "\*\*\* YOUR CODE HERE \*\*\*"

    if currentGameState.isWin():

        return 10000 + random.random()#使吃掉最后一个后吃豆人知道去哪个地方，防止停在最后一个豆子前

    if currentGameState.isLose():

        return -10000

    walls=currentGameState.getWalls()

    pacmanPosition = currentGameState.getPacmanPosition()

    ghostStates = currentGameState.getGhostStates()

    scaredTimes = [ghostState.scaredTimer for ghostState in ghostStates]

    ghostLocations = currentGameState.getGhostPositions()

    food = currentGameState.getFood()#获取各项信息

    countWalls = 0

    if walls[pacmanPosition[0] + 1][pacmanPosition[1]] == True:

        countWalls=countWalls + 1

    if walls[pacmanPosition[0] - 1][pacmanPosition[1]] == True:

        countWalls = countWalls + 1

    if walls[pacmanPosition[0]][pacmanPosition[1] + 1] == True:

        countWalls = countWalls + 1

    if walls[pacmanPosition[0]][pacmanPosition[1] - 1] == True:

        countWalls = countWalls + 1

    if countWalls == 3:

        return -10000#如果是死胡同，返回一个很低的值

    dGhost = 10000

    i = 0

    for x in ghostLocations:

        if scaredTimes[i]<2:

            tem = abs(pacmanPosition[0] - x[0]) + abs(pacmanPosition[1] - x[1])

            if tem < dGhost:

                dGhost = tem #计算最近的鬼的距离

        i = i + 1

    min = 10000

    foodLocation = food.asList()

    min\_x = 0

    min\_y = 0

    for i in foodLocation:

        x = abs(pacmanPosition[0] - i[0])

        y = abs(pacmanPosition[1] - i[1])

        temp = x + y

        if min > temp:

            min = temp

            min\_x = i[0]

            min\_y = i[1]# 最近的豆子

    max = 0

    for i in foodLocation:

        x = abs(min\_x - i[0])

        y = abs(min\_y - i[1])

        temp = x + y

        if max < temp:

            max = temp # 离最近豆子最远的豆子

    d = min + max #豆子距离的估计值

    num = food.count()

    if dGhost <= 2:

        Evaluation = dGhost \* 100

        if num == 0:

            Evaluation = Evaluation + 10

        else:

            Evaluation = Evaluation - num \* 1 + 1.0 / d

        Evaluation = Evaluation - 10000

    else:

        if num == 0:

            Evaluation = 10000

        else:

            Evaluation = -num \* 100 + 100.0 / d #同问题1的赋值

    return Evaluation+random.random()/10000+currentGameState.getScore() #防止吃掉某个豆子后不知道去那里

在判断胜利的返回值加上一个很小的随机数，使之在不影响其他的情况下每次都不相同，避免吃豆人在吃掉最后一个豆子后不知道去哪里从而停在最后一个豆子前

判断吃豆人四个方向的墙的数量，如果是 3，则说明该处是一个胡同，为了使吃豆人不进入胡同，避免无意义的徘徊，我将吃豆人处于胡同处的状态赋值为一个较低的值

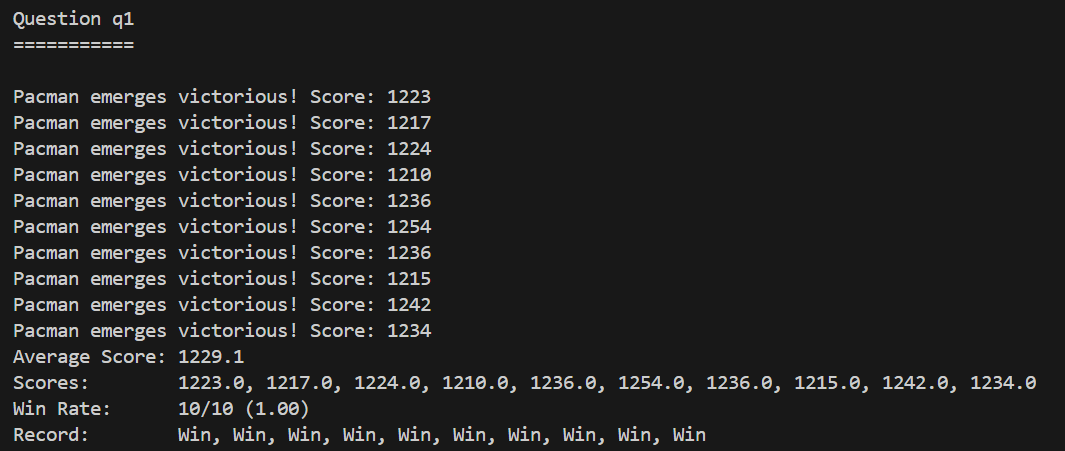
由于可能有多个鬼，但是最近的一个鬼决定吃豆人的状态是否危险，所以取最近的鬼的距离来代表鬼的距离

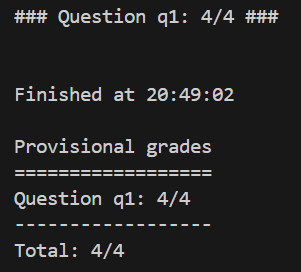
用最近的豆子距离 min，距离该豆子最远的豆子两个豆子之间的距离 max，这两个值的和 min+max 来估计吃豆人吃完所有豆子的路径耗散

评估值要加上当前的得分，防止吃豆人一直在某个地方徘徊

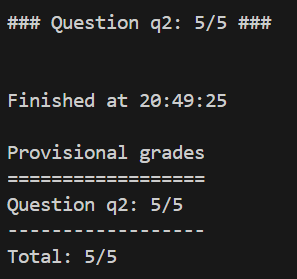
1. 结果展示

**Q1：Reflex Agent**

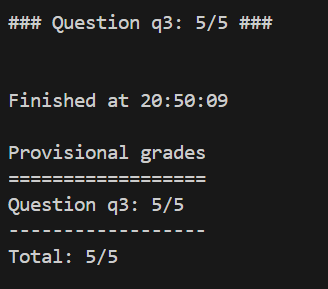
****

****

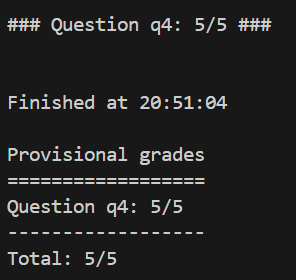
**Q2：Minimax**

****

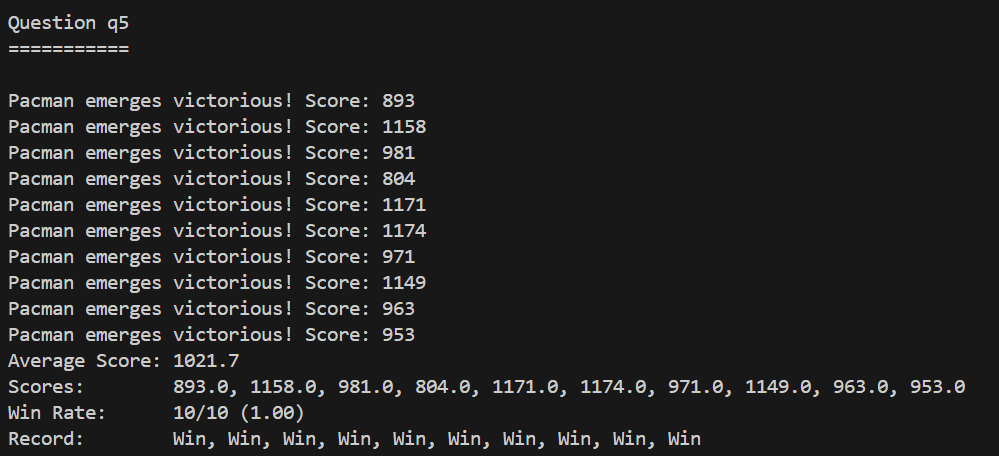
**Q3：Alpha-Beta Pruning**

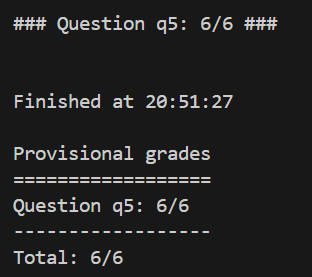
****

**Q4：Expectimax**

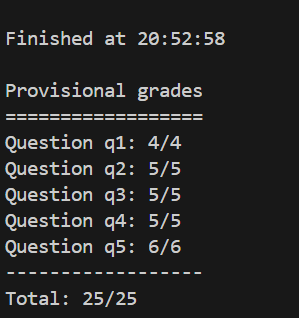


**Q5：Evaluation Function**





**Total Result**



1. 总结与分析

**1、算法分析**

在 minimax 算法中，智能体将自己视为最大化自己的利益，而将对手视为最小化自己的利益。该算法在深度优先搜索的过程中逐步确定每个决策节点的值，从而确定最佳行动方案。

然而，minimax 算法的计算复杂度很高，因此 alpha-beta 剪枝算法可以用来优化搜索过程。该算法通过评估子树中的最大和最小值来减少搜索时间。在实践中，alpha-beta 算法比 minimax 算法更有效，并且能够找到最佳决策方案。

但是 minimax 算法和 alpha-beta 算法是基于对手也会做出最优选择的前提下的，当对手并不总是做出最优选择时，可能并不适用，因此可以根据对手行动的特点来具体的对状态进行评估，二不是总是评估为子节点的最小值

关于评估函数的设计，既可以评估某个状态下智能体的行为，也可以直接评估某个状态。

**2、项目总结**

**2.1 遇到的困难及解决方法：**

在完成此项目过程中，在代码方面遇到了一些困难，虽然经过第一次项目的实践，让我对python的基础语法有了一定程度的掌握，但是仍然还谈不上熟悉，因此在本次项目的过程中，我积极借助网络上的相关资料，丰富自己的python知识，才得以顺利地解决问题。但是这也导致我在项目刚开始时看已有代码的过程中用掉了大量时间，不过也得到了相应的成效，花费大量时间把项目要求仔细阅读的代码逻辑理顺后，再补全算法函数就比较简单了。

**2.2 项目总结与个人收获：**

通过这两次的项目，我对于python这门语言的掌握程度得到了很大的提升，对于python的运用也越来越得心应手，虽然在项目实现过程中还是遇到了不少困难，但是经过仔细思考和反复斟酌，最终能很好地克服了。经过这两个项目的循序渐进，我不但增长了我对人工智能搜索问题的理解，而且夯实了我的python程序设计基础，这是一次非常有意义的实践。