同济大学

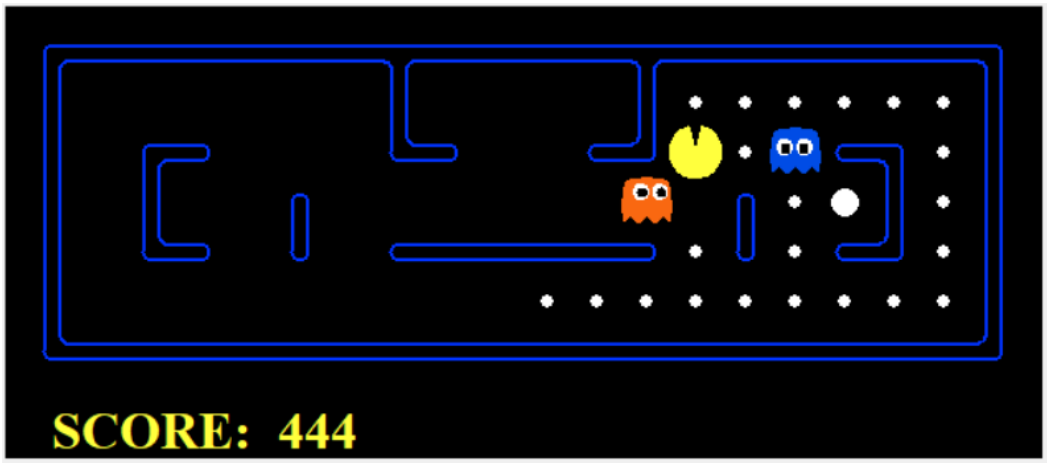
实验报告

实验名称：Pacman吃豆人 对抗搜索

学号：2153538

姓名：刘博洋

完成日期：2023.4.23



1. 项目分析与概述

1.1.项目总体描述

吃豆人生活在一个蓝色世界里,由弯曲的迷宫构成，迷宫内有分散在各处的食物，同时还有追杀吃豆人的幽灵，有效地收集食物并避开危险，是吃豆人生存的第一要务。

本项目主要是设计程序控制吃豆人针对有幽灵的情况进行针对性的行动，收集食物并且避开怪物，并获得更高的分数。本项目有不同的地图资源，同时也有不同的智能Agent，我们将为不同的Agent设计不同对抗搜索算法来解决这些问题。

1.2.项目已有代码结构及类分析

需要编辑的文件	
multiAgents.py	所有对抗搜索的算法代码
需要阅读的文件	
pacman.py	运行 pacman 项目的主要代码，同时包括一个重要的 GameState 类
game.py	吃豆人世界的底层逻辑
util.py	有用的数据结构和计算距离的函数

项目已有代码包括核心的对抗搜索算法部分的multiAgents部分，同时包括一些便于我们编写算法的工具，除了以上提及的部分之外，同时比较重要的部分还有Gamestate类，类中包含了指向不同问题的接口，以及比较重要的generatesuccessor方法和方法用来寻找后继搜索节点和判断是否到达目标，在编写算法前需要对这部分进行仔细阅读。

1.3.子问题概述及简要分析

1.3.1. Q1：反应智能体（Reflex Agent）

Reflex Agent只会对当前世界的状态作出反应，不会考虑它的操作的后果，而只是根据世界的当前状态而选择一个操作。本问题只需要找到后继状态并对当前状态作出反应即可。

1.3.2. Q2：极大极小算法（MiniMAX）

把吃豆人看作Max玩家，每一步选择效用值最大的行动，而默认ghost为Min玩家，每次作出ghost视角的最佳选择。这需要在搜索中不断递归深入并更新Max和Min的值。

1.3.3. Q3： α－β剪枝（Alpha-Beta Pruning）

虽然MiniMax算法可以通过minimax搜索树来不断递归查找得到步骤，但是往往搜索到的代价比较大，遍历了一些没有必要搜索的节点。这时候用到Alpha-Beta剪枝算法，简单来说当你已经知道后继节点的效用值不如已经搜索到的优，那么就不必去遍历后继节点了。其中α，它是一个结点所能选取的评价值的下界，β，它是一个结点所能选取的评价值的上界，在搜索每一个

节点的过程中动态更新α和β，达到剪枝条件的时候便进行剪枝

1.3.4. Q4：期望最优（ExpectiMax）

由于在问题2中我们假设ghost每次都执行最优行动，但是在实际情况中这是难以实现的，可能ghost没有我们预料的那样智能，而只是随机的进行搜索，在这种情况下，我们仍然要设计算法吃掉食物并避开ghost，其实只需要把之前的ghost每次min函数效用值改为ghost采用所有合法行动的期望即可

1.3.5 Q5：更好的评估函数（Better evaluation function f）

要求为pacman设计一个更好的评估函数，以便其能够在随机的多ghost环境和不同的地图下都能存活下来并得到更多分数。

2. 算法设计与实现

2.1.1. Q1：反应智能体（Reflex Agent）

Reflex Agent在每一个节点通过评估函数来选择一个动作，而不会考虑后继结果，所以本题只需要为Agent设计一个比较合适的评估函数即可。

```
class ReflexAgent(Agent):
    """
    A reflex agent chooses an action at each choice point by examining
    its alternatives via a state evaluation function.
    """
    def getAction(self, gameState: GameState):
        """
        getAction chooses among the best options according to the evaluation
function.
        """
        # Collect legal moves and successor states
        legalMoves = gameState.getLegalActions()
        # Choose one of the best actions
        scores = [self.evaluationFunction(gameState, action) for action in
legalMoves]
        bestScore = max(scores)
        bestIndices = [index for index in range(len(scores)) if scores[index]
== bestScore]
        chosenIndex = random.choice(bestIndices) # Pick randomly among the
best
        "Add more of your code here if you want to"
        return legalMoves[chosenIndex]

    def evaluationFunction(self, currentGameState: GameState, action):
        """
        Design a better evaluation function here.
```

```
        The code below extracts some useful information from the state, like
the
        remaining food (newFood) and Pacman position after moving (newPos).
        newScaredTimes holds the number of moves that each ghost will remain
        scared because of Pacman having eaten a power pellet.
        """
        # Useful information you can extract from a GameState (pacman.py)
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition() # 当前吃豆人位置
        newFood = successorGameState.getFood() # 当前食物位置
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in
newGhostStates]

        "*** YOUR CODE HERE ***"
        GhostPos=successorGameState.getGhostPositions() # 获取幽灵位置
        ghostdist=min([(abs(each[0] - newPos[0]) + abs(each[1] - newPos[1]))
for each in GhostPos])  # 计算吃豆人行动后到幽灵的距离
        if ghostdist<=4 and ghostdist!=0:
            ghostscore=-15/ghostdist # 计算幽灵惩罚函数
        else:
            ghostscore=0
        foodaround=[]
        Width=newFood.width
        Height=newFood.height
        for i in range(Width):
            for j in range(Height):
                if newFood[i][j]==1:
                    foodaround.append((i,j))    # 暴力枚举每一个存在食物的地方
        if ghostdist>=2 and len(foodaround)>0:
            nearestfooddis=min([manhattanDistance(newPos,food) for food in
foodaround ]) # 计算距离吃豆人最近的食物距离
            foodscore=10/nearestfooddis # 计算食物评估函数
        else:
            foodscore=0
        currentscore=successorGameState.getScore()
        newscore=currentscore+foodscore+ghostscore
        return newscore

def scoreEvaluationFunction(currentGameState: GameState):
    """
    This default evaluation function just returns the score of the state.
    The score is the same one displayed in the Pacman GUI.

    This evaluation function is meant for use with adversarial search agents
    (not reflex agents).
    """
    return currentGameState.getScore()
```

可以看到本问题主要在于评估函数的实现，因为要实现吃掉食物并避开ghost，所以要设置评估函数对附近最近的食物存在奖励机制，而采用foodscore=10/nearestfooddis是因为采用除法后，当最近食物距离增加时，得到的奖励反馈会减少，这也符合实际的要求，而且在计算食物距离的时候，我还加上了一个判断条件ghostdist>=2，这是因为如果不限制ghost到pacman的距离，那么会导致不论pacman离ghost多远，行动都会受到不小的限制，而这种显示造成pacman效率十分低下，甚至会出现

原地往返踱步的情况。同时值得注意的是对于ghost距离的效用值，ghostscore=-15/ghostdist，这表示在距离食物和ghost相同时，pacman会优先选择逃避ghost而暂时先不考虑食物，这是处于pacman的首要任务是存活，其次才是获取食物。

2.1.2. Q2：极大极小算法（MiniMAX）

把吃豆人看作Max玩家，每一步选择效用值最大的行动，而默认ghost为Min玩家，每次作出ghost视角的最佳选择。这里有可能有多个ghost，所以博弈树可能是一层Max，多层Min的样式。极大极小算法从当前状态计算极大极小决策，使用简单递归算法计算每个后继的极大极小值。

```
    class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (question 2)
    """
    def getAction(self, gameState):
        """
        Returns the minimax action from the current gameState using
self.depth
        and self.evaluationFunction.

        Here are some method calls that might be useful when implementing
minimax.

        gameState.getLegalActions(agentIndex):
        Returns a list of legal actions for an agent
        agentIndex=0 means Pacman, ghosts are >= 1

        gameState.generateSuccessor(agentIndex, action):
        Returns the successor game state after an agent takes an action

        gameState.getNumAgents():
        Returns the total number of agents in the game

        gameState.isWin():
        Returns whether or not the game state is a winning state

        gameState.isLose():
        Returns whether or not the game state is a losing state
        """
        "*** YOUR CODE HERE ***"
        # ghost 可能不止一个
        GhostIndex = range(1, gameState.getNumAgents())

        # 目标状态：游戏结束或者搜索到一定深度
        def targetstate(state, Depth):
            return state.isWin() or state.isLose() or Depth == self.depth

        # ghost 是 min 玩家
        def min_value(state, Depth, ghost):  # minimizer

            if targetstate(state, Depth):
                return self.evaluationFunction(state)

            MIN = float("inf")  # MIN 初始值为无穷大
            for action in state.getLegalActions(ghost):
```

```
                if ghost == GhostIndex[-1]:  # 递归的查找最小值，如果最后一个 ghost 已
经作出行动了，下一次便是轮到 pacman
                    MIN = min(MIN, max_value(state.generateSuccessor(ghost,
action), Depth + 1))
                else:  # 否则就遍历所有 ghost，每一个 ghost 作出行动后再由 pacman 行动
                    MIN = min(MIN, min_value(state.generateSuccessor(ghost,
action), Depth, ghost + 1))
            return MIN
        # pacman 是 max 玩家，每次选择效用值最大的行动
        def max_value(state, Depth):

            if targetstate(state, Depth):
                return self.evaluationFunction(state)

            MAX = -float("inf")  # MAX 初始值为无穷小
            for action in state.getLegalActions(0):
                # 递归查找下一个状态中效用值最大的
                MAX = max(MAX, min_value(state.generateSuccessor(0, action),
Depth, 1))
            return MAX

        ans = [(action, min_value(gameState.generateSuccessor(0, action), 0,
1)) for action in
               gameState.getLegalActions(0)]
        ans.sort(key=lambda k: k[1])
          return ans[-1][0]
```

首先明确博弈的中止状态，调用state类中的isWin和isLose方法来进行判断，然后用min_value和max_value两个函数来递归计算后继节点的极大极小值，首先设MAX为无穷小，MIN为无穷大，每次比较其值与后继节点取大，然后最后因为可以理解为ghost先动，所以调用min_value并递归搜索到指定深度，把得到的最佳行动回传。

2.1.3. Q3：α－β剪枝（Alpha-Beta Pruning）

α－β剪枝剪去那些一定对结果没有影响的节点和枝干，其剪枝的规则如下：设置两个值α，β，分别为目前位置路径上发现的MAX的最佳选择（极大值）和目前位置当前路径上发现的MIN的最佳选择（极小值）。有以下几点性质：

1. Max层的α = max(α， 它的所有子结点的评价值)，Max层的β = 它的父结点的β

2. Min层的β = min(β， 它的所有子结点的评价值)，Min层的 α = 它的父结点的α

3. 当某个结点的 α >= β，停止搜索该节点的其他子结点

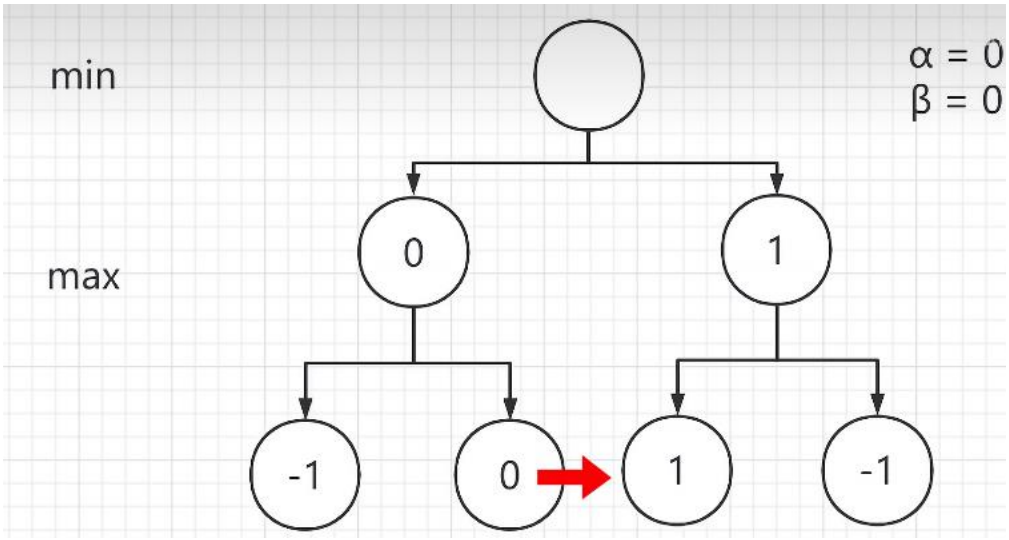
Max层中，若某个结点的最优解已经大于它的父结点的最差解，则不必继续搜索，剪枝；Min层中，若某个结点的最差解已经小于它的父结点的最优解，则不必继续搜索，剪枝。

```
    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 ***"
        # ghost 可能不止一个
        GhostIndex = range(1, gameState.getNumAgents())

        # 目标状态：游戏结束或者搜索到一定深度
        def targetstate(state, Depth):
            return state.isWin() or state.isLose() or Depth == self.depth

        # ghost 是 min 玩家
        def min_value(state, Depth, ghost,A,B):

            if targetstate(state, Depth):
                return self.evaluationFunction(state)

            MIN = float("inf")  # MIN 初始值为无穷大
            for action in state.getLegalActions(ghost):
                if ghost == GhostIndex[-1]:  # 递归的查找最小值，如果最后一个 ghost 已
经作出行动了，下一次便是轮到 pacman
                    MIN = min(MIN, max_value(state.generateSuccessor(ghost,
action), Depth + 1,A,B))
                else:  # 否则就遍历所有 ghost，每一个 ghost 作出行动后再由 pacman 行动
                    MIN = min(MIN, min_value(state.generateSuccessor(ghost,
action), Depth, ghost + 1,A,B))
                if MIN < A:  # 剪枝
                    return MIN
                B = min(B, MIN)
            return MIN

        # pacman 是 max 玩家，每次选择效用值最大的行动
        def max_value(state, Depth,A,B):

            if targetstate(state, Depth):
                return self.evaluationFunction(state)

            MAX = -float("inf")  # MAX 初始值为无穷小
            for action in state.getLegalActions(0):
                # 递归查找下一个状态中效用值最大的
                MAX = max(MAX, min_value(state.generateSuccessor(0, action),
Depth, 1,A,B))
                if MAX > B:
                  return MAX
                A = max(A, MAX)
            return MAX


        def alphabeta(state):

            MAX = -float("inf")
            act = None
            A = -float("inf")   # A 为目前已经发现的 MAX 的极大值，如果搜索过程中发现
            B = float("inf")
```

```
            for action in state.getLegalActions(0):  # 求最大值
                value = min_value(gameState.generateSuccessor(0, action), 0, 1,
A, B)

                if MAX< value:
                    MAX = value
                    act = action

                if MAX > B:  # 剪枝
                    return MAX
                A = max(A, value)
            return act

        return alphabeta(gameState)
```

其实很好解释这一过程，如下图



此时遍历到右子树的叶子节点1，此时是Max层，要找效用值较大的，但是在遍历左边子树后，更新了α=β=0，所以根节点min在选择时一定会选0和右子树中max的最小值，但是遍历到当前节点时发现max最小也为1，大于β，所以最后一个叶子节点就不用遍历了，最后一定是取左边的0，用代码写出来就是

```
MAX = -float("inf")  # MAX 初始值为无穷小
            for action in state.getLegalActions(0):
                # 递归查找下一个状态中效用值最大的
                MAX = max(MAX, min_value(state.generateSuccessor(0, action),
Depth, 1,A,B))
                if MAX > B:
                  return MAX
                A = max(A, MAX)
            return MAX
```

这里如果发现Max大于beta就直接返回了，否则还要更新alpha。对于min_value函数也是同理，这样就实现了剪枝操作，大大减少了搜索代价。

2.1.4. Q4：期望最优（ExpectiMax）

Ghost每次都进行最优操作是可遇不可求的，我们必须也要能在对手不作出最佳操作时进行最合理的操作，扩大自己的优势，这里可以把ghost的行动看成随机的，

```
    class ExpectimaxAgent(MultiAgentSearchAgent):
    """
      Your expectimax agent (question 4)
    """
    def getAction(self, gameState):
        """
        Returns the expectimax action using self.depth and
self.evaluationFunction

        All ghosts should be modeled as choosing uniformly at random from
their
        legal moves.
        """
        "*** YOUR CODE HERE ***"
        GhostIndex = [i for i in range(1, gameState.getNumAgents())]

        # 目标状态：游戏结束或者搜索到一定深度
        def target(state, d):
            return state.isWin() or state.isLose() or d == self.depth

        # ghost 不一定每次作出最好决定，要计算期望值
        def exp_value(state, d, ghost):

            if target(state, d):
                return self.evaluationFunction(state)

            value = 0
            prob = 1 / len(state.getLegalActions(ghost))# 每种情况的概率

            for action in state.getLegalActions(ghost):
                if ghost == GhostIndex[-1]:# 递归的查找期望值，如果最后一个 ghost 已经
作出行动了，下一次便是轮到 pacman
                    value += prob * max_value(state.generateSuccessor(ghost,
action), d + 1)
                else:# 否则就遍历所有 ghost，每一个 ghost 作出行动后再由 pacman 行动
                    value += prob * exp_value(state.generateSuccessor(ghost,
action), d, ghost + 1)

            return v

        def max_value(state, d):

            if target(state, d):
                return self.evaluationFunction(state)

            MAX = -float("inf")
            for action in state.getLegalActions(0):
                MAX = max(MAX, exp_value(state.generateSuccessor(0, action), d,
1))
            return MAX
```

```
        ans = [(action, exp_value(gameState.generateSuccessor(0, action), 0,
1)) for action in
               gameState.getLegalActions(0)]
        ans.sort(key=lambda k: k[1])

        return ans[-1][0]
```

设置ghost每个行动的概率都相等，然后每一次搜索后都把所得效用值加入value,然后pacman再根据ghost生成的效用值进行行动。

2.1.5 Q5：更好的评估函数（Better evaluation function f）

要求为pacman设计一个更好的评估函数，以便其能够在随机的多ghost环境和不同的地图下都能存活下来并得到更多分数。

首先可以看到之前的评估函数只是看到了ghost和food的距离，但是要是更加细致的考虑，还可以考虑对吃到胶囊作为一个奖励（胶囊的分数很高）并且吃掉胶囊会让ghost进入受惊模式，这是ghost变成pacman的食物，吃掉可以加分，同时scaredtimer这一指标，表示幽灵剩下能被吃掉的时间，这些都是可以考虑加入评估函数的因素。

```
    def betterEvaluationFunction(currentGameState:GameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (question 5).
    DESCRIPTION: <write something here so we know what you did>
    """
    Pos = currentGameState.getPacmanPosition()  # current position
    Food = currentGameState.getFood()  # current food
    GhostStates = currentGameState.getGhostStates()  # ghost state
    ScaredTimes = [ghostState.scaredTimer for ghostState in GhostStates]
    capsulepos=currentGameState.getCapsules()
    # 食物带来的积极影响
    if len(Food.asList()) > 0:
        nearestFood = (min([manhattanDistance(Pos, food) for food in
Food.asList()]))
        foodScore = 10 / nearestFood
    else:
        foodScore = 0
    if len(capsulepos) > 0:
        nearestcapsule = (min([manhattanDistance(Pos, capsule) for capsule in
capsulepos]))
        capsuleScore = 0.5 / nearestcapsule
    else:
        capsuleScore = 0
    # 幽灵带来的负面影响
    nearestGhost = min([manhattanDistance(Pos, ghostState.configuration.pos)
for ghostState in GhostStates])
    if nearestGhost<4 and nearestGhost!=0:
        dangerScore = -11 / nearestGhost
    else:
        dangerScore=0
```

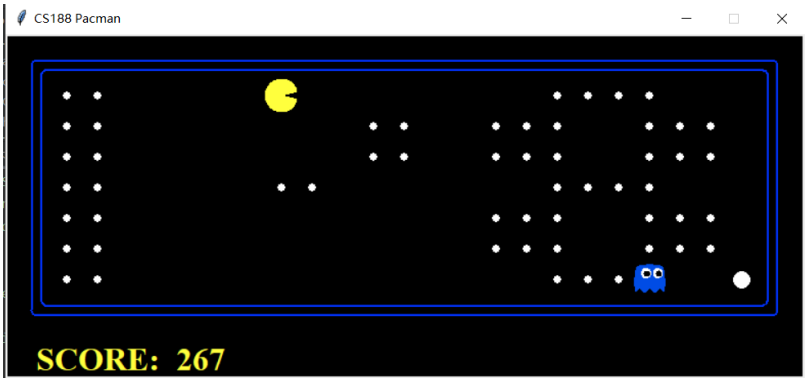
```
    # 幽灵剩下能被吃掉的时间
    totalScaredTimes = sum(ScaredTimes)
    # return sum of all value
    return currentGameState.getScore() + foodScore + dangerScore +
totalScaredTimes+capsuleScore
```

但是针对这么多因素，要考虑它们的比例，首先在ghost距离小于4时再考虑ghost的影响，然后这个影响要大于食物和胶囊影响的总和。同时胶囊的影响不能过大，否则pacman就会因为优先去吃两个胶囊而绕路而降低总分，所以我经过多次试验将食物的分数设置为`foodScore = 10 / nearestFood`, `capsuleScore = 0.5 / nearestcapsule`, `dangerScore = -11 / nearestGhost`,最后再加上幽灵剩下能被吃的时间，构成最后的评估函数。

3. 实验结果

Q1：

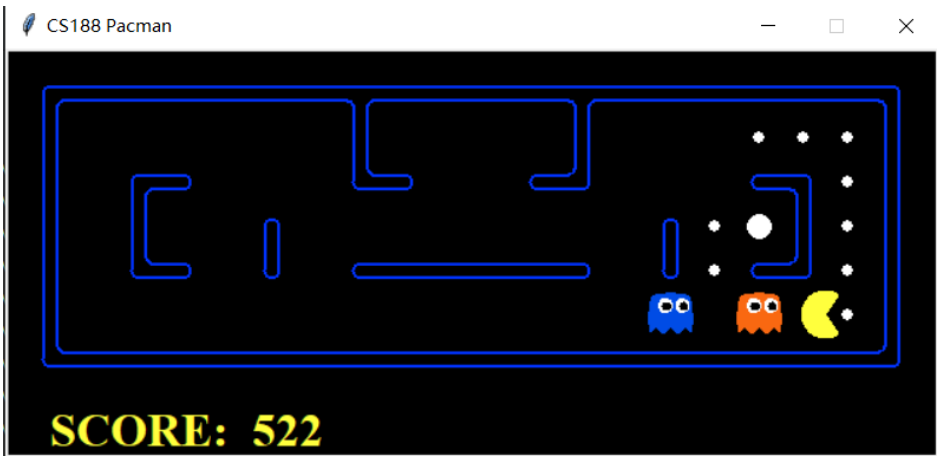
```
Question q1
===========

Pacman emerges victorious! Score: 1230
Pacman emerges victorious! Score: 1241
Pacman emerges victorious! Score: 1245
Pacman emerges victorious! Score: 1221
Pacman emerges victorious! Score: 1223
Pacman emerges victorious! Score: 1232
Pacman emerges victorious! Score: 1235
Pacman emerges victorious! Score: 1224
Pacman emerges victorious! Score: 1243
Pacman emerges victorious! Score: 1243
Average Score: 1233.7
Scores:        1230.0, 1241.0, 1245.0, 1221.0, 1223.0, 1232.0, 1235.0, 1224.0, 1243.0, 1243.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```



可以看到pacman吃完了所有食物并没有被ghost抓住，不过路线有些愚蠢，绕路较多。

Q2：

```
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
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q2\8-pacman-game.test

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



多数情况下pacman会被ghost追上，但是这并不妨碍pacman在这个狭小的地图里且面对两个ghost已经作出了比较好的反应和行为。

Q3：

```
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-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 15 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q3\8-pacman-game.test

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

相对于Q2运行速度和搜索成本上都有了明显的提升。

Q4：

```
D:\study_files\programming\pythoncode\AI_project\multiagent>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
```

这是运用AlphaBetaAgent进行博弈是在trappedClassic地图中运行的结果，pacman最终都被吃掉，因为过高估计了ghost每一次都以最佳行动进行，而实际上ghost行动可以认为是随机的。

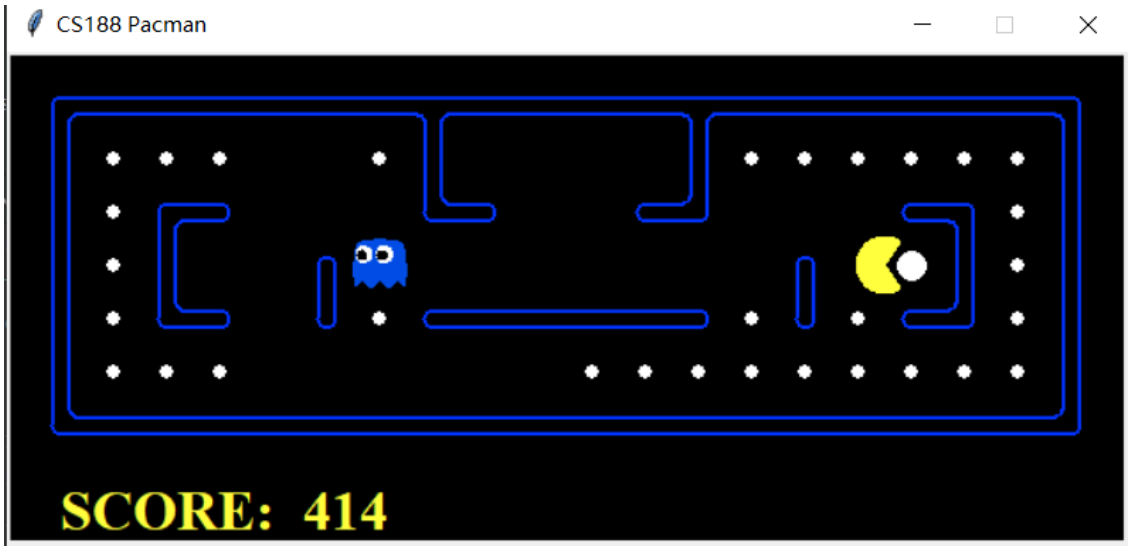
```
D:\study_files\programming\pythoncode\AI_project\multiagent>python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
Pacman emerges victorious! Score: 531
Pacman emerges victorious! Score: 531
Pacman emerges victorious! Score: 531
Pacman emerges victorious! Score: 531
Pacman emerges victorious! Score: 531
Pacman emerges victorious! Score: 531
Pacman emerges victorious! Score: 531
Pacman emerges victorious! Score: 531
Pacman died! Score: -502
Pacman died! Score: -502
Average Score: 324.4
Scores:        531.0, 531.0, 531.0, 531.0, 531.0, 531.0, 531.0, 531.0, -502.0, -502.0
Win Rate:      8/10 (0.80)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Loss, Loss
```

但是用改良后的ExpectimaxAgent就可以实现大部分情况下获胜。

```
Question q4
===========

*** 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\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 15 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q4\7-pacman-game.test

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


Finished at 21:45:44

Provisional grades
==================
Question q4: 5/5
```

最后测试结果

Q5：




```
Question q5
===========

Pacman emerges victorious! Score: 1372
Pacman emerges victorious! Score: 1351
Pacman emerges victorious! Score: 1373
Pacman emerges victorious! Score: 1367
Pacman emerges victorious! Score: 1218
Record:       Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\q5\grade-agent.test (6 of 6 points)
***     1271.8 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 ###


Finished at 21:47:29

Provisional grades
==================
Question q5: 6/6
------------------
Total: 6/6
```

改进后的评估函数使得pacman会主动去获取胶囊，以此获得吃掉ghost的机会，并且对周围的食物和ghost规划行动也变得更加合理。

4. 总结与分析

4.1 算法分析

在多 Agent 环境中（竞争环境），每个 Agent 的目标之间是有冲突的，所以就引出了对抗搜索（通常称为博弈）。由于在博弈问题中，博弈（树）实在太大，一般的搜索效率是很低的。在博弈搜索的算法中，剪枝允许我们在搜索树中忽略那些不影响最后决定的部分，启发式的评估函数允许在不进行完全搜索的情况下，估计某状态的真实效用值。

博弈问题的的 6 个元素：

1）S0：初始状态（游戏开始时的情况）

2）Player(s)：定义此时该谁动

3）Actions(s):此状态下的合法移动集合

4）Result(s,a):转移模型，定义行动的结果

5）Terminal-test(s)：终止测试，游戏结束返回真，否则假。

6）Utility(s,p)：效用函数，定义游戏者 p 在终止状态 s 下的数值。

由此引出 MiniMax 算法，在该算法中，每个节点都有一个极小极大值，即为 MiniMax(n)。Max 玩家每一步想要最大的效用值，而 Min 玩家始终想要最小的效用值。采用了简单的递归算法计算每个后继的极小极大值，递归算法自上而下一直前进到树的叶节点，然后随着递归回溯通过搜索树把极小极大值回传。

MiniMax 算法对博弈树执行完整的深度优先搜索，如果树的最大深度是 m，在每个节点合法的行棋有 b 个，时间复杂度：$O(b^m)$，空间复杂度：$O(b^m)$（一次性生成所有后继），O(m)（每次只生成一个后继）。然而，对于真实的游戏，这样的时间开销完全不实用，但是此算法仍然可以作为对博弈进行数学分析和设计实验算法的基础。

为了节省搜索成本，我们引入 α－β 剪枝操作，这一操作是每次搜索中加入一步判断和更新操作，把必定不会影响结果的极大极小值节点和枝干不再搜索。

```
function ALPHA-BETA-SEARCH(state) returns an action
  v←MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
---------------------------------------------------
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v←−∞
  for each a in ACTIONS(state) do
    v←MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α←MAX(α, v)
  return v
---------------------------------------------------
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v←+∞
  for each a in ACTIONS(state) do
    v←MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if v ≤ α then return v
    β←MIN(β, v)
  return v
```

图 5.7　α-β搜索算法

4.2 项目总结

遇到的困难及解决：

在完成此项目过程中，在代码方面遇到了一些困难，主要还是因为python基础知识比较薄弱导致的，如在一个元组中按关键字进行排序，这部分内容我尝试了多次也参考了一些网上资料才写出来比较正确的排序操作。

还有一个比较难的地方是在α-β剪枝操作的函数设计中，必须要弄清楚α，β究竟代表什么，又是如何在每一步更新的，在min_value函数中判断当min层如果min的后继已经小于已经有的α，那么剪枝，在max_value函数中判断当max层如果max的后继大于已经有的beta，剪枝。

随后就是最后一个问题设计评估函数时，也是经过多次对效用值比例的修改，才控制pacman不至于太过于重视胶囊而忽略危险和顺路的食物。

项目总结与收获：

通过这次项目，通过循序渐进的4个问题，我不但增长了我对人工智能对抗搜索问题的理解，而且夯实了我的python程序设计基础，是一次非常有意义的实践。