Homework 3: Multi-Agent Search

Part I. Implementation (5%):

- **Part 1**

```python
    def getAction(self, gameState):
        # Begin your code (Part 1)
        '''
        I implemented minimax recursively instead of using a stack.
        The choice of max or min is determined by the agentIndex sent as a parameter.

        When the now_node(gameState) is in the parameter, the function is computing the max or min value of the descendent nodes.
        It goes through each nodes iteratively and go deeper if the descendent node isn't the end node.
        It then return the value of descendent node to let the now_node decide which one to choose.

        Notice: since I add depth whenever an agent make a choice, the condition indicating it reach the depth specified
        is depth == self.depth * number_of_agents.
        '''
        bestAction = self.compute_util_minimax(gameState, 0)
        return bestAction[1]

    def compute_util_minimax(self, gameState, agentIndex,depth=0):
        if agentIndex==0:                                           # give initial value according to agent
            maxOrMin = (float('-inf'), 'STOP')
        else:
            maxOrMin = (float('inf'), 'STOP')
        depth += 1                                                  # add depth by 1
        legalActions = gameState.getLegalActions(agentIndex)
        for action in legalActions:
            nextState = gameState.getNextState(agentIndex, action)  # get all possible next states
                                                                    # return the value if it is end node
            if nextState.isLose() or nextState.isWin() or depth==self.depth*gameState.getNumAgents():
                temp = (self.evaluationFunction(nextState),)
            else:                                                   # compute downwards with the same way
                temp = self.compute_util_minimax(nextState, (agentIndex + 1) % nextState.getNumAgents(), depth)

            if  agentIndex == 0:                                    # if it is pacman, choose the max
                if temp[0] > maxOrMin[0]:
                    maxOrMin = (temp[0], action)
            else:                                                   # else, choose min
                if temp[0] < maxOrMin[0]:
                    maxOrMin = (temp[0], action)
        return maxOrMin
        # End your code (Part 1)
```

- **Part 2**

```python
    def getAction(self, gameState):
        """
        Returns the minimax action using self.depth and self.evaluationFunction
        """
        # Begin your code (Part 2)
        '''
        I only made some modification on the recursive function.
        1. adding 2 parameter alpha and beta
        2. add some code according to the alpha-beta pruning psuedocode (between ###s)
        '''
        bestAction = self.compute_util_minimax(gameState, 0, -float('inf'), float('inf'))
        return bestAction[1]

    def compute_util_minimax(self, gameState, agentIndex, alpha, beta ,depth=0):
        if agentIndex==0:
            maxOrMin = (float('-inf'), 'STOP')
        else:
            maxOrMin = (float('inf'), 'STOP')
        depth += 1
        legalActions = gameState.getLegalActions(agentIndex)
        for action in legalActions:
            nextState = gameState.getNextState(agentIndex, action)
            if nextState.isLose() or nextState.isWin() or depth==self.depth*gameState.getNumAgents():
                temp = (self.evaluationFunction(nextState),)
            else:
                temp = self.compute_util_minimax(nextState, (agentIndex + 1) % nextState.getNumAgents(), alpha, beta, depth)
            if  agentIndex == 0:
                if temp[0] > maxOrMin[0]:
                    maxOrMin = (temp[0], action)
                if temp[0] > beta:                  ###
                    return temp
                alpha = max(alpha, temp[0])         ###
            else:
                if temp[0] < maxOrMin[0]:
                    maxOrMin = (temp[0], action)
                if temp[0] < alpha:                 ###
                    return temp
                beta = min(beta, temp[0])           ###
        return maxOrMin
        # End your code (Part 2)
```

- **Part 3**

```python
    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.
        """
        # Begin your code (Part 3)
        '''
        The code is quite similar to minimax.
        The difference is that when it comes to the turn of the ghost, compute the value as the sum of values of all the
        possible paths divided by the number of possible paths.
        The difference is marked with ##

        '''
        bestAction = self.compute_util_expectimax(gameState, 0)
        #print(bestAction)
        return bestAction[1]

    def compute_util_expectimax(self, gameState, agentIndex, depth=0):
        if agentIndex==0:
            maxOrMin = (float('-inf'), 'LEFT')
        else:                                                                   ##
            number = 0
        depth += 1
        legalActions = gameState.getLegalActions(agentIndex)
        for action in legalActions:
            nextState = gameState.getNextState(agentIndex, action)
            if nextState.isLose() or nextState.isWin() or depth==self.depth*gameState.getNumAgents():
                temp = (self.evaluationFunction(nextState),)
            else:
                temp = self.compute_util_expectimax(nextState, (agentIndex + 1) % nextState.getNumAgents(), depth)
            if  agentIndex == 0:
                if temp[0] > maxOrMin[0]:
                    maxOrMin = (temp[0], action)
            else:
                number += temp[0]                                               ##
        if agentIndex!= 0:                                                      ##
            return (number/len(legalActions),action)                            ##
        return maxOrMin
        # End your code (Part 3)
```

- **Part 4**

```python
def BFS(wholeMap, position):
    Q = util.Queue()
    done = [position]
    ans = []
    Q.push((position,0))                                    # push initial
    direction = [(1,0),(-1,0),(0,1),(0,-1)]
    while not Q.isEmpty():
        temp = Q.pop()                                      # pop from queue
        ans.append(temp)                                    # add to answer
        for i in direction:
            newPos = (temp[0][0]+i[0], temp[0][1]+i[1])
            if(wholeMap[newPos[0]][newPos[1]] == False and newPos not in done):
                Q.push((newPos, temp[1]+1))                 # add neccesary node to queue and done
                done.append((temp[0][0],temp[0][1]))
    return ans
```

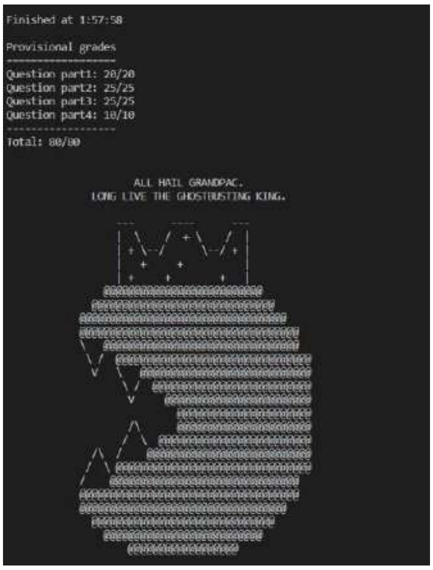
BFS is implemented with Queue defined in util.py.

It returns a tuple of tuples ((positionX, positionY), search value), and it is used in the evaluation function.

```python
def betterEvaluationFunction(currentGameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (Part 4).
    """
    # Begin your code (Part 4)
    '''
    There are three factors I used to evaluate the condition.
    1. whether the position is near the food.
        I used BFS to mark all the non-wall positions and use the value gained
        by BFS to evaluate the position.
        I tried several functions to deal with the list of values, and the one
        I am using is the best of all the function I tried.
    2. currentGameState.getScore().
        The score is higher -> the state is better
    3. capsules not eaten.
        The game will end even if the capsule isn't eaten, since you will have
        more point if you eat all the capsules, I add number of capsules * -100
        to the return value (better if less capsule).
    Additionally, I add the evaluation of ghost hunting to achieve better score.
    When the ghost is scared, the return value will become the negative of the
    distance of ghost and pacman.
    '''
    if currentGameState.isLose():                       # huge number if win or lose
        return -9999999
    elif currentGameState.isWin():
        return 9999999 + currentGameState.getScore()

    nowPos = currentGameState.getPacmanPosition()
    food = currentGameState.getFood()
    wholeMap = currentGameState.getWalls()
    mapWithDist = BFS(wholeMap,nowPos)                          # do BFS
    foodValues = []

    capsulePositions = currentGameState.getCapsules()
    for j in mapWithDist:                                       # get all the distance of foods
        if food[j[0][0]][j[0][1]]:
            foodValues.append(j[1])
    temp = 0

    foodValueScore   = 2 / (min(foodValues) / float(max(foodValues) - min(foodValues) + 1) + 1) + temp
    capsuleScore = 0

    ghostState = currentGameState.getGhostStates()[0]
    hunt = False                                                # know if the ghost should be hunted
    if ghostState.scaredTimer > 0 and ghostState.scaredTimer < 28:
        hunt = True
    if not hunt:
        capsuleScore = len(capsulePositions)*-100               # less capsule -> better
    if hunt:
        ghostPositions = currentGameState.getGhostPositions()
        return - manhattanDistance(nowPos,ghostPositions[0])    # hunt the ghost (evaluation for hunting)
    return  foodValueScore*2  + currentGameState.getScore()*5 + capsuleScore
    # End your code (Part 4)
```

Part II. Results & Analysis (5%):

- **Result**



- **Analysis**

1. Design factors on evaluation funciton

   At first, I considered several types of factors, e.g., the distance of ghost, the distance of capsules. When I use all the factors together, it is hard for me to decide the ratio of these factors, and the pacman seems to be dizzy, not knowing where to go. This might result from the interactions of each factors (because it changes rapidly and might make the pacman walk in a way resembling to walking randomly).

2. Deletion of ghost distance factor

   There is only one ghost in all the test cases, so it is easy for pacman to avoid bumping into the ghost in all the cases (can not be surrounded because there is only one ghost). Hence, I remove the factor of the ghost from the factor to reach better performance (the factor was evaluating how far you are with all the ghosts).

   However, if there are more pacman in a game, the evaluation might do poorly because it can not foresee whether the position might be surrounded by or be close to ghosts.

3. Adding ghost hunting

   One thing I observed in the game is that the score will be higher if you eat all the capsules and eat all the ghosts per capsule. I designed the evaluation function for the hunting process in a way that all the evaluation will only based on the distance of pacman and ghost so that the choice will focus on minimizing the distance between pacman and ghost.

   Before I add the hunting trategy, the average score is about 1170, and the variance of each game is quite high, the scores range from 1370 to 980. The figure below is the result after I add the hunting stratigy. It is obvious that the new evaluation function can have a more stable and higher score than the one without hunting strategy.

```
Question part4
==============

Pacman emerges victorious! Score: 1373
Pacman emerges victorious! Score: 1370
Pacman emerges victorious! Score: 1369
Pacman emerges victorious! Score: 1362
Pacman emerges victorious! Score: 1371
Pacman emerges victorious! Score: 1364
Pacman emerges victorious! Score: 1364
Pacman emerges victorious! Score: 1373
Pacman emerges victorious! Score: 1350
Pacman emerges victorious! Score: 1342
Average Score: 1363.8
Scores:        1373.0, 1370.0, 1369.0, 1362.0, 1371.0, 1364.0, 1364.0, 1373.0, 1350.0, 1342.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```