Homework 3: Multi-Agent Search

Part I. Implementation (20%):

Part 1: Minimax Search

```python
        # Begin your code (Part 1)
        def minimax(state, depth, agentIndex):
            if depth == 0 or state.isWin() or state.isLose():
                return self.evaluationFunction(state)
            elif agentIndex == 0:  # Pacman's turn (maximizing)
                maxEval = float("-inf")
                for action in state.getLegalActions(agentIndex):
                    successor = state.getNextState(agentIndex, action)
                    maxEval = max(maxEval, minimax(successor, depth, 1))
                return maxEval
            else: # Ghost's turn (minimizing)
                minEval = float("inf")
                for action in state.getLegalActions(agentIndex):
                    successor = state.getNextState(agentIndex, action)
                    if agentIndex == state.getNumAgents() - 1:
                        minEval = min(minEval, minimax(successor, depth - 1, 0))
                    else:
                        minEval = min(minEval, minimax(successor, depth, agentIndex + 1))
                return minEval

        legalActions = gameState.getLegalActions(0)
        bestAction = None
        bestValue = float("-inf")
        for action in legalActions:
            successor = gameState.getNextState(0, action)
            value  = minimax(successor, self.depth, 1) # Start with depth 1 for ghosts
            if value > bestValue:
                bestValue = value
                bestAction = action
        return bestAction
        # End your code (Part 1)
```

→ First of all, I use the **getLegalActions** function to get the possible actions pacman will do. Next, utilize for loop to determine which the best action is. I use the agentIndex and action to find the successor. Subsequently, utilize **minimax** function I define to give each situation a value. Finally, I will choose the bestAction which got the highest score from **minimax** function.
In the **minimax** function, there are three situations. First, if it is Pacman's turn, I choose the maximum evaluation value by considering each possible outcome. This is done by recursively calling the **minimax** function, switching to the ghost's turn. Second, if it is ghost's turn, I choose the minimum evaluation value by considering each possible outcome.This is done by recursively calling the **minimax** function. Note that if it is the last ghost's turn, I will switch the turn to pacman again and decrease

the depth by 1. Otherwise, just switch the turn to next ghost. Finally, if the depth is equal to 0 or I already can determine whether the game is win or lose, I will just return the evaluation value through calling **evaluationFunction** .

Part 2: Alpha-Beta Pruning

```
        # Begin your code (Part 2)
        def alphaBeta(state, depth, alpha, beta, agentIndex):
            if depth == 0 or state.isWin() or state.isLose():
                return self.evaluationFunction(state)
            elif agentIndex==0: # Pacman's turn (maximizing)
                return max_value(state, depth, alpha, beta, agentIndex)
            else:  # Ghost's turn (minimizing)
                return min_value(state, depth, alpha, beta, agentIndex)
        def max_value(state, depth, alpha, beta, agentIndex):
            maxEval = float("-inf")
            for action in state.getLegalActions(agentIndex):
                successor = state.getNextState(agentIndex, action)
                eval = alphaBeta(successor, depth, alpha, beta, 1)
                maxEval = max(maxEval, eval)
                if beta < maxEval:
                    return maxEval
                alpha = max(alpha, maxEval)
            return maxEval

        def min_value(state, depth, alpha, beta, agentIndex):
            minEval = float("inf")
            for action in state.getLegalActions(agentIndex):
                successor = state.getNextState(agentIndex, action)
                if agentIndex == state.getNumAgents()-1:
                    eval = alphaBeta(successor, depth-1, alpha, beta, 0)
                    minEval = min(minEval, eval)
                else:
                    eval = alphaBeta(successor, depth, alpha, beta, agentIndex+1)
                    minEval = min(minEval, eval)
                if minEval < alpha:
                    return minEval
                beta = min(beta, minEval)
            return minEval

        legalActions = gameState.getLegalActions(0)
        bestAction = None
        bestValue = float("-inf")
        alpha = float("-inf")
        beta = float("inf")
        for action in legalActions:
            successor = gameState.getNextState(0, action)
            value  = alphaBeta(successor, self.depth, alpha, beta, 1) # Start with depth 1 for ghosts
            if value > bestValue:
                bestValue = value
                bestAction = action
            alpha = max(alpha, bestValue)
        return bestAction
        # End your code (Part 2)
```

→ This method is the modification of **minimax** algorithm. First of all, I use the **getLegalActions** function to get the possible actions pacman will do. Next, utilize for loop to determine which the best action is. I use the agentIndex and action to find the successor. Subsequently, utilize the **alphaBeta** function I define to give each situation a value. Finally, I will choose the bestAction which got the highest score from **alphaBeta** function.

In **alphaBeta** function, there are two more parameters, which are alpha and beta. I can use them to perform pruning, which can save lots of time. There are three

situations in **alphaBeta** function. First, if it is Pacman's turn, I choose the maximum evaluation value by considering each possible outcome. This is done by recursively calling the **max_value** function. In **max_value** function, it will continuously update alpha and try to find the best value by recursively calling the **alphaBeta** function, switching to the ghost's turn. If the max evaluation value is larger than beta, the function will just return max evaluation value. Second, if it is ghost's turn, I choose the minimum evaluation value by considering each possible outcome.This is done by recursively calling the **min_value** function. In **min_value** function, it will continuously update beta and try to find the smallest value by recursively calling the **alphaBeta** function, switching to the ghost's turn. Note that if it is the last ghost's turn, I will switch the turn to pacman again and decrease the depth by 1. Otherwise, just switch the turn to next ghost. If the minimum evaluation value is smaller than alpha, the function will just return minimum evaluation value. Finally, if the depth is equal to 0 or I can determine whether the game is win or lose, I will just return the evaluation value through calling **evaluationFunction** .

Part 3: Expectimax Search

```
240         # Begin your code (Part 3)
241         def expectimax(state, depth, agentIndex):
242             if depth == 0 or state.isWin() or state.isLose():
243                 return self.evaluationFunction(state)
244             elif agentIndex == 0:  # Pacman's turn (maximizing)
245                 maxEval = float("-inf")
246                 for action in state.getLegalActions(agentIndex):
247                     successor = state.getNextState(agentIndex, action)
248                     maxEval = max(maxEval, expectimax(successor, depth, 1))
249                 return maxEval
250             else: # Ghost's turn (minimizing)
251                 expEval = 0
252                 actions = state.getLegalActions(agentIndex)
253                 probability = 1.0 / len(actions)
254                 for action in actions:
255                     successor = state.getNextState(agentIndex, action)
256                     if agentIndex == state.getNumAgents() - 1:
257                         expEval += probability * expectimax(successor, depth-1, 0)
258                     else:
259                         expEval += probability * expectimax(successor, depth, agentIndex+1)
260                 return expEval
261 
262         legalActions = gameState.getLegalActions(0)
263         bestAction = None
264         bestValue = float("-inf")
265         for action in legalActions:
266             successor = gameState.getNextState(0, action)
267             value  = expectimax(successor, self.depth, 1) # Start with depth 1 for ghosts
268             if value > bestValue:
269                 bestValue = value
270                 bestAction = action
271         return bestAction
272         # End your code (Part 3)
```

→ First of all, I use the **getLegalActions** function to get the possible actions pacman will do. Next, utilize for loop to determine which the best action is. I use the agentIndex and action to find the successor. Subsequently, utilize **expectimax** function I define to give each situation a value. Finally, I will choose the bestAction

which got the highest score from **expectimax** function.
In the **expectimax** function, there are three situations. First, if it is Pacman's turn, I choose the maximum evaluation value by considering each possible outcome. This is done by recursively calling the **expectimax** function, switching to the ghost's turn. Second, if it is ghost's turn, I use the expected evaluation value by considering each possible outcome.This is done by recursively calling the **expectimax** function. Note that if it is the last ghost's turn, I will switch the turn to pacman again and decrease the depth by 1. Otherwise, just switch the turn to next ghost. Finally, if the depth is equal to 0 or I already can determine whether the game is win or lose, I will just return the evaluation value through calling **evaluationFunction**.

Part 4: Evaluation Function

```
    # Begin your code (Part 4)
    # initialize the basic information
    pacman_position = currentGameState.getPacmanPosition()
    ghost_positions = currentGameState.getGhostPositions()
    score = currentGameState.getScore()
    capsule_list = currentGameState.getCapsules()
    capsule_count = len(capsule_list)
    food_list = currentGameState.getFood().asList()
    food_count = len(food_list)
    nearest_food = 1
    nearest_ghost = 1
    nearest_capsule = 1

    # find the distance from pacman to each food
    food_distance = [manhattanDistance(pacman_position, food_position) for food_position in food_list]
    if food_count > 0:
        nearest_food = min(food_distance)

    # find the distance from pacman to each capsule
    capsule_distance = [manhattanDistance(pacman_position, capsule_position) for capsule_position in capsule_list]
    if capsule_count > 0:
        nearest_capsule = min(capsule_distance)
    #print(nearest_capsule)
    #print(type(nearest_capsule))

    # find the distance from pacman to each ghost
    ghost_distance = [manhattanDistance(pacman_position, ghost_position) for ghost_position in ghost_positions]
    if len(ghost_distance) > 0:
        nearest_ghost = min(ghost_distance)


    # If pacman is too close to the ghost,
    # then let it escape first by setting
    # the distance to the nearest food
    if nearest_ghost <= 1 or nearest_ghost > 13:
        nearest_food = 10000000
    if nearest_ghost==0:
        nearest_ghost = 0.0001

    features = [1./nearest_food, score, food_count, capsule_count, 1./nearest_capsule, 1./nearest_ghost]

    weights = [50, 200, -100, -150, 60, 5]

    # Linear combination of features
    return sum([feature * weight for feature, weight in zip(features, weights)])
    # End your code (Part 4)
```

→First of all, utilize the different function to get the information I need. Second, start to calculate some features I will use later, including the shortest distance between food and pacman, the score, the number of remaining food, the shortest distance between capsule and pacman, the number of remaining capsule and the shortest distance between ghost and pacman. Finally, multiply the weights (get them via trial and error) and features, and sum them together to get a final result.

Since the objective is for Pacman to reach the food and capsules, the evaluation function assigns higher values to shorter distances. This is achieved by taking the reciprocal of the distances. Besides, the reason why I take the distance between pacman and ghost is that avoiding pacman stops at the same position time too long. The ghost is a good factor to drive pacman to walk around. However, I have already considered the situation where ghost is too close or far from pacman. Consequently, I only set 5, which is a small number, to multiply the reciprocal of the ghost distance. Moreover, owing to the same reason that make Pacman reach the food and capsules, I set the weight, which will multiply the numbers of them, to be negative. In this way, when pacman eat one of them, the score will increase. Additionally, with an eye to getting more score, I try to make pacman eat all the capsules. If I want to achieve this goal, it is neccesary for pacman to eat capsule before eating all the food. Hence, I set higher absolute value for capsule-related parameter.

Part II. Results & Analysis (10%):

part1 result in autograder:

```
Question part1
==============

*** PASS: test_cases\part1\0-eval-function-lose-states-1.test
*** PASS: test_cases\part1\0-eval-function-lose-states-2.test
*** PASS: test_cases\part1\0-eval-function-win-states-1.test
*** PASS: test_cases\part1\0-eval-function-win-states-2.test
*** PASS: test_cases\part1\0-lecture-6-tree.test
*** PASS: test_cases\part1\0-small-tree.test
*** PASS: test_cases\part1\1-1-minmax.test
*** PASS: test_cases\part1\1-2-minmax.test
*** PASS: test_cases\part1\1-3-minmax.test
*** PASS: test_cases\part1\1-4-minmax.test
*** PASS: test_cases\part1\1-5-minmax.test
*** PASS: test_cases\part1\1-6-minmax.test
*** PASS: test_cases\part1\1-7-minmax.test
*** PASS: test_cases\part1\1-8-minmax.test
*** PASS: test_cases\part1\2-1a-vary-depth.test
*** PASS: test_cases\part1\2-1b-vary-depth.test
*** PASS: test_cases\part1\2-2a-vary-depth.test
*** PASS: test_cases\part1\2-2b-vary-depth.test
*** PASS: test_cases\part1\2-3a-vary-depth.test
*** PASS: test_cases\part1\2-3b-vary-depth.test
*** PASS: test_cases\part1\2-4a-vary-depth.test
*** PASS: test_cases\part1\2-4b-vary-depth.test
*** PASS: test_cases\part1\2-one-ghost-3level.test
*** PASS: test_cases\part1\3-one-ghost-4level.test
*** PASS: test_cases\part1\4-two-ghosts-3level.test
*** PASS: test_cases\part1\5-two-ghosts-4level.test
*** PASS: test_cases\part1\6-tied-root.test
*** PASS: test_cases\part1\7-1a-check-depth-one-ghost.test
*** PASS: test_cases\part1\7-1b-check-depth-one-ghost.test
*** PASS: test_cases\part1\7-1c-check-depth-one-ghost.test
*** PASS: test_cases\part1\7-2a-check-depth-two-ghosts.test
*** PASS: test_cases\part1\7-2b-check-depth-two-ghosts.test
*** PASS: test_cases\part1\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 1 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\part1\8-pacman-game.test

### Question part1: 15/15 ###
```

part2 result in autograder:

```
Question part2
==============

*** PASS: test_cases\part2\0-eval-function-lose-states-1.test
*** PASS: test_cases\part2\0-eval-function-lose-states-2.test
*** PASS: test_cases\part2\0-eval-function-win-states-1.test
*** PASS: test_cases\part2\0-eval-function-win-states-2.test
*** PASS: test_cases\part2\0-lecture-6-tree.test
*** PASS: test_cases\part2\0-small-tree.test
*** PASS: test_cases\part2\1-1-minmax.test
*** PASS: test_cases\part2\1-2-minmax.test
*** PASS: test_cases\part2\1-3-minmax.test
*** PASS: test_cases\part2\1-4-minmax.test
*** PASS: test_cases\part2\1-5-minmax.test
*** PASS: test_cases\part2\1-6-minmax.test
*** PASS: test_cases\part2\1-7-minmax.test
*** PASS: test_cases\part2\1-8-minmax.test
*** PASS: test_cases\part2\2-1a-vary-depth.test
*** PASS: test_cases\part2\2-1b-vary-depth.test
*** PASS: test_cases\part2\2-2a-vary-depth.test
*** PASS: test_cases\part2\2-2b-vary-depth.test
*** PASS: test_cases\part2\2-3a-vary-depth.test
*** PASS: test_cases\part2\2-3b-vary-depth.test
*** PASS: test_cases\part2\2-4a-vary-depth.test
*** PASS: test_cases\part2\2-4b-vary-depth.test
*** PASS: test_cases\part2\2-one-ghost-3level.test
*** PASS: test_cases\part2\3-one-ghost-4level.test
*** PASS: test_cases\part2\4-two-ghosts-3level.test
*** PASS: test_cases\part2\5-two-ghosts-4level.test
*** PASS: test_cases\part2\6-tied-root.test
*** PASS: test_cases\part2\7-1a-check-depth-one-ghost.test
*** PASS: test_cases\part2\7-1b-check-depth-one-ghost.test
*** PASS: test_cases\part2\7-1c-check-depth-one-ghost.test
*** PASS: test_cases\part2\7-2a-check-depth-two-ghosts.test
*** PASS: test_cases\part2\7-2b-check-depth-two-ghosts.test
*** PASS: test_cases\part2\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 1 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\part2\8-pacman-game.test

### Question part2: 20/20 ###
```

part3 result in autograder:

```
Question part3
==============

*** PASS: test_cases\part3\0-eval-function-lose-states-1.test
*** PASS: test_cases\part3\0-eval-function-lose-states-2.test
*** PASS: test_cases\part3\0-eval-function-win-states-1.test
*** PASS: test_cases\part3\0-eval-function-win-states-2.test
*** PASS: test_cases\part3\0-expectimax1.test
*** PASS: test_cases\part3\1-expectimax2.test
*** PASS: test_cases\part3\2-one-ghost-3level.test
*** PASS: test_cases\part3\3-one-ghost-4level.test
*** PASS: test_cases\part3\4-two-ghosts-3level.test
*** PASS: test_cases\part3\5-two-ghosts-4level.test
*** PASS: test_cases\part3\6-1a-check-depth-one-ghost.test
*** PASS: test_cases\part3\6-1b-check-depth-one-ghost.test
*** PASS: test_cases\part3\6-1c-check-depth-one-ghost.test
*** PASS: test_cases\part3\6-2a-check-depth-two-ghosts.test
*** PASS: test_cases\part3\6-2b-check-depth-two-ghosts.test
*** PASS: test_cases\part3\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 1 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\part3\7-pacman-game.test

### Question part3: 20/20 ###
```

part4 result in autograder:

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

Pacman emerges victorious! Score: 1160
Pacman emerges victorious! Score: 1171
Pacman emerges victorious! Score: 1368
Pacman emerges victorious! Score: 1161
Pacman emerges victorious! Score: 1121
Pacman emerges victorious! Score: 1176
Pacman emerges victorious! Score: 1361
Pacman emerges victorious! Score: 1356
Pacman emerges victorious! Score: 1334
Pacman emerges victorious! Score: 1351
Average Score: 1255.9
Scores:        1160.0, 1171.0, 1368.0, 1161.0, 1121.0, 1176.0, 1361.0, 1356.0, 1334.0, 1351.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\part4\grade-agent.test (8 of 8 points)
*** EXTRA CREDIT: 2 points
***     1255.9 average score (4 of 4 points)
***         Grading scheme:
***          < 600:  0 points
***         >= 600:  2 points
***         >= 1200:  4 points
***     10 games not timed out (2 of 2 points)
***         Grading scheme:
***          < 0:  fail
***         >= 0:  0 points
***         >= 5:  1 points
***         >= 10:  2 points
***     10 wins (4 of 4 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  1 points
***         >= 4:  2 points
***         >= 7:  3 points
***         >= 10:  4 points

### Question part4: 10/10 ###
```

provisional grades:

```
Provisional grades
==================
Question part1: 15/15
Question part2: 20/20
Question part3: 20/20
Question part4: 10/10
------------------
Total: 65/65
```

observation in my evaluation function:

First version of my evaluation function:

At the beginning, I just use 4 features, which are the distance between pacman and food, the score, the remaining numbers of food, and the remaining numbers of capsule.

```
features = [1./nearest_food, score, food_count, capsule_count]

weights = [10, 200, -100, 10]
```

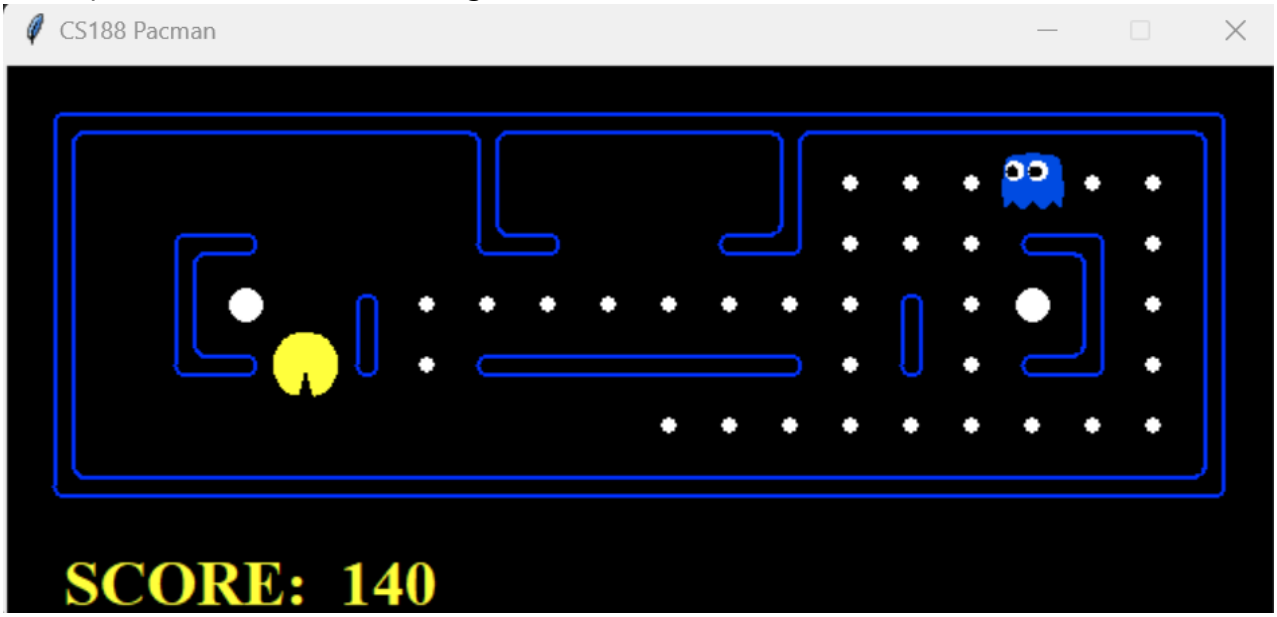
However, the result is not as ideal as I think.

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

Pacman emerges victorious! Score: 884
Pacman emerges victorious! Score: 880
Pacman emerges victorious! Score: 922
Pacman emerges victorious! Score: 843
Pacman emerges victorious! Score: 817
Pacman emerges victorious! Score: 917
Pacman emerges victorious! Score: 920
Pacman emerges victorious! Score: 927
Pacman emerges victorious! Score: 813
Pacman emerges victorious! Score: 802
Average Score: 872.5
Scores:        884.0, 880.0, 922.0, 843.0, 817.0, 917.0, 920.0, 927.0, 813.0, 802.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\part4\grade-agent.test (8 of 8 points)
***     872.5 average score (2 of 4 points)
***         Grading scheme:
***          < 600:  0 points
***         >= 600:  2 points
***         >= 1200:  4 points
***     10 games not timed out (2 of 2 points)
***         Grading scheme:
***          < 0:  fail
***         >= 0:  0 points
***         >= 5:  1 points
***         >= 10:  2 points
***     10 wins (4 of 4 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  1 points
***         >= 4:  2 points
***         >= 7:  3 points
***         >= 10:  4 points

### Question part4: 8/10 ###
```

Therefore, I tried to observe the movement of pacman and discovered that in some similar situation in the following picture, the pacman would prefer staying at the same position instead of moving around.



The reason is that neither direction pacman choose will change the numbers of food and capsule or make the distance to food shorter. As a result, pacman preferred to stay until the ghost is too close to pacman.

Second version of my evaluation function:

If I want to avoid pacman staying at the same position, I need to add some features, which can motivate pacman moving around. This feature should always changed. Thus, I think the most suitable feature is the distance between ghost and pacman. This feature is utilized to avoid pacman staying at the same position, so I do not set a large weight to it.

```
features = [1./nearest_food, score, food_count, capsule_count, 1./nearest_ghost]

weights = [10, 200, -100, -10, 8]
```

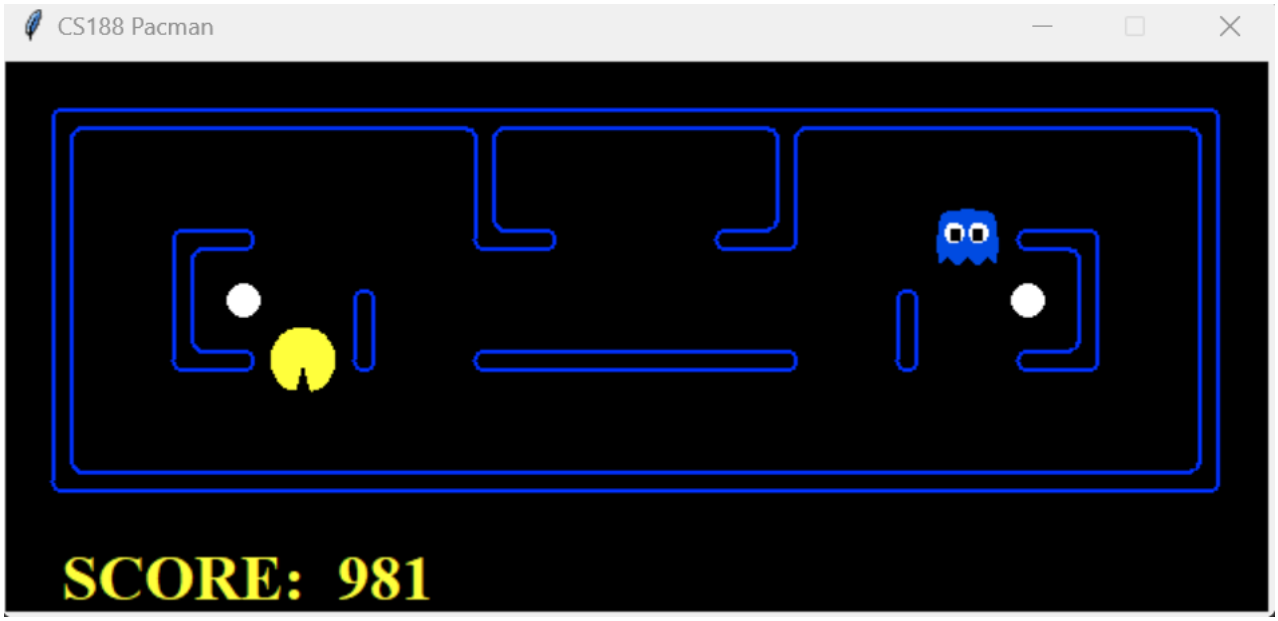
The result is better than the score of the initial version and average score is very close to target 1200.

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

Pacman emerges victorious! Score: 1370
Pacman emerges victorious! Score: 971
Pacman emerges victorious! Score: 1315
Pacman emerges victorious! Score: 1174
Pacman emerges victorious! Score: 1361
Pacman emerges victorious! Score: 1179
Pacman emerges victorious! Score: 1166
Pacman emerges victorious! Score: 1175
Pacman emerges victorious! Score: 981
Pacman emerges victorious! Score: 1306
Average Score: 1199.8
Scores:        1370.0, 971.0, 1315.0, 1174.0, 1361.0, 1179.0, 1166.0, 1175.0, 981.0, 1306.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\part4\grade-agent.test (8 of 8 points)
***     1199.8 average score (2 of 4 points)
***         Grading scheme:
***          < 600:  0 points
***         >= 600:  2 points
***         >= 1200:  4 points
***     10 games not timed out (2 of 2 points)
***         Grading scheme:
***          < 0:  fail
***         >= 0:  0 points
***         >= 5:  1 points
***         >= 10:  2 points
***     10 wins (4 of 4 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  1 points
***         >= 4:  2 points
***         >= 7:  3 points
***         >= 10:  4 points

### Question part4: 8/10 ###
```

Therefore, I tried to observe the movement of pacman again and discovered that in some similar situation in the following picture, the score is lower than others.



The reason is that pacman has already eaten all the food before he eats any capsules. Therefore, pacman cannot eat the ghost to get the additional score. As a result, the score will be relatively low.

Final version of my evaluation function:

My goal is to make pacman eat two capsules before eating all the food. Therefore, I add one more feature, which is the distance between and pacman and the capsule. Moreover, I have increased the weight associated with capsules to prioritize pacman's consideration of consuming them. I have also adjusted other weight in order to get higher score.

```
features = [1./nearest_food, score, food_count, capsule_count, 1./nearest_capsule, 1./nearest_ghost]

weights = [50, 200, -100, -150, 60, 5]
```

Finally, I get the average score which exceeds 1200 and pass the test.

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

Pacman emerges victorious! Score: 1160
Pacman emerges victorious! Score: 1171
Pacman emerges victorious! Score: 1368
Pacman emerges victorious! Score: 1161
Pacman emerges victorious! Score: 1121
Pacman emerges victorious! Score: 1176
Pacman emerges victorious! Score: 1361
Pacman emerges victorious! Score: 1356
Pacman emerges victorious! Score: 1334
Pacman emerges victorious! Score: 1351
Average Score: 1255.9
Scores:        1160.0, 1171.0, 1368.0, 1161.0, 1121.0, 1176.0, 1361.0, 1356.0, 1334.0, 1351.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\part4\grade-agent.test (8 of 8 points)
*** EXTRA CREDIT: 2 points
***     1255.9 average score (4 of 4 points)
***         Grading scheme:
***          < 600:  0 points
***         >= 600:  2 points
***         >= 1200:  4 points
***     10 games not timed out (2 of 2 points)
***         Grading scheme:
***          < 0:  fail
***         >= 0:  0 points
***         >= 5:  1 points
***         >= 10:  2 points
***     10 wins (4 of 4 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  1 points
***         >= 4:  2 points
***         >= 7:  3 points
***         >= 10:  4 points

### Question part4: 10/10 ###
```