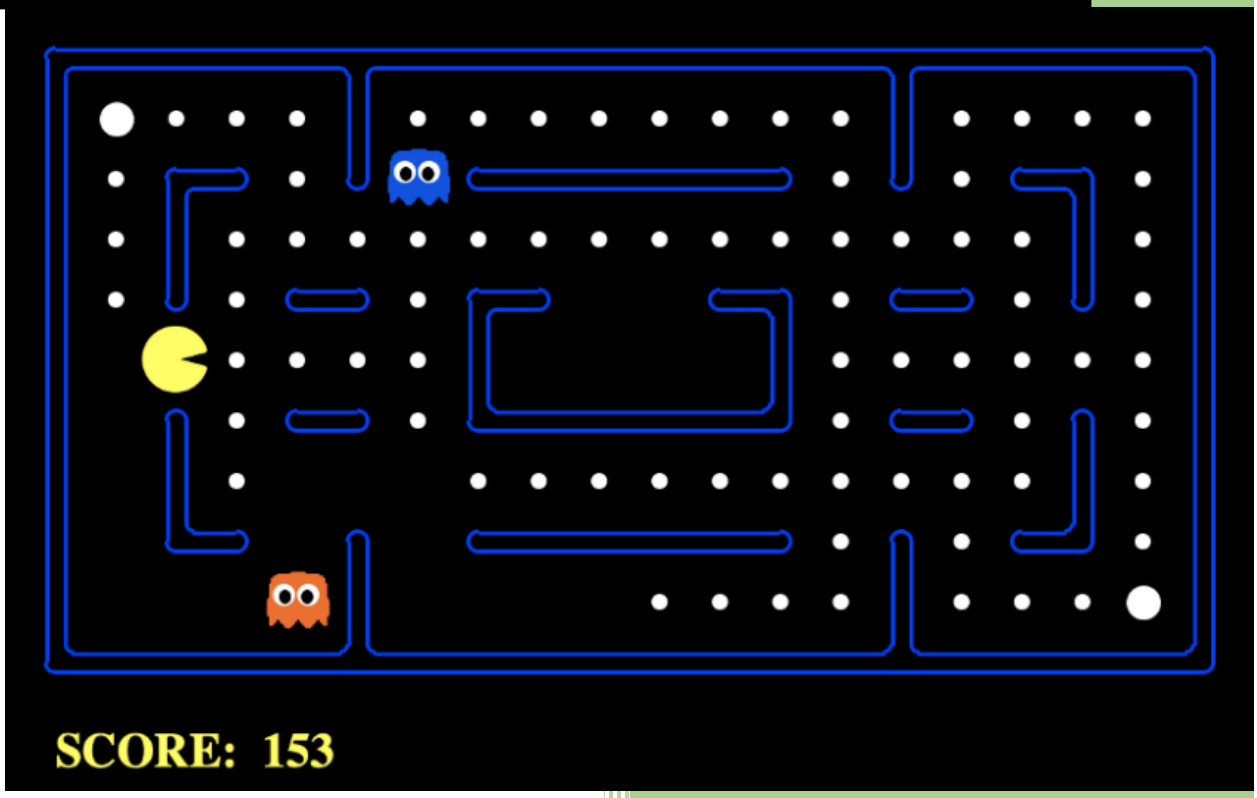
2020

AI Course – Pacman Project Report



Sara Elbesomy 201601849

Fall 2020

First Question: Reflex agent

In this question, I developed an evaluation function that makes the agents play respectably. In the evaluation function, I used two factors to control the actions of the pacman; the Manhattan distance to the food and the Manhattan distance to the ghost.

```
newFood = newFood.asList()
distance_to_food = float("inf")
for food in newFood:
    d_food = manhattanDistance(newPos, food)
    distance_to_food = min(distance_to_food,d_food)
```

Figure 1

As shown in figure 1, I used the Manhattan distance between the pacman and the food to motivate the pacman to move towards the nearest food.

```
ghost_pos = successorGameState.getGhostPositions()
for ghost in ghost_pos :
    d_ghost = manhattanDistance(newPos, ghost)
    if d_ghost < 5:
        return -float('inf')
```

Figure 2

In figure 2, I used the Manhattan distance between the pacman and the ghost to motivate the pacman to move in the opposite direction of the ghost's movement when it stops. When the distance is less than 5, that means there are less than 5 steps between the pacman and the ghost so, the pacman should run in the opposite direction immediately.

```
return successorGameState.getScore() + 1.0 / distance_to_food + 1.0/d_ghost
```

Figure 3

In the return of the evaluation function as shown in figure 3, I return the score of the game itself, (it is calculated in the game according to how many food the pacman eats and how much time it does not move). I add to the previous score the reciprocal of the minimum Manhattan distance between the pacman and the food. In addition, I add the reciprocal of the Manhattan distance between the pacman and the ghost. This combination of metrics represents my final evaluation metric.

The evaluation function meet all the question requirement and pass with the full mark (4 points out of 4 points) in the auto grader test as shown in figure 4.

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

Pacman emerges victorious! Score: 1238
Pacman emerges victorious! Score: 1195
Pacman emerges victorious! Score: 1090
Pacman emerges victorious! Score: 1173
Pacman emerges victorious! Score: 1212
Pacman emerges victorious! Score: 1220
Pacman emerges victorious! Score: 1230
Pacman emerges victorious! Score: 1225
Pacman emerges victorious! Score: 1194
Pacman emerges victorious! Score: 1179
Average Score: 1195.6
Scores:        1238.0, 1195.0, 1090.0, 1173.0, 1212.0, 1220.0, 1230.0, 1225.0, 1194.0, 1179.0
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)
```

Figure 4

Second Question: Minimax

In this question, I developed an adversarial search agent that works according to minimax search algorithm. The main idea is my agent calculates all the future states with the actions leading to them, and then it choose the best action that leads to its desired state. The desired state in the case of pacman is achieving the maximum for itself while the desired state for the ghosts is achieving the minimum state for the pacman.

Note: As mentioned in the project description, the pacman index is always 0, and the ghosts take the remaining numbers.

```
value = float("-inf")
bestAction = []
agent_index = 0
actions = gameState.getLegalActions(agent_index)
successors = [(action, gameState.generateSuccessor(agent_index, action)) for action in actions]
num_agents = gameState.getNumAgents()
for successor in successors:
    result = minimax(1, range(num_agents), successor[1], self.depth, self.evaluationFunction)
    if result > value:
        value = result
        bestAction = successor[0]
return bestAction
```

Figure (5)

Figure (5) shows the main segment of my code. I defined a value equals to negative infinity in the beginning to ensure that this is the minimum possible value as I compares it later with other states' values ,and I need to ensure that the maximum value will be always chosen.

The actions array contains all the legal actions that could be done by the agent, in this case it is the pacman. The successors array contains a list of two elements for each action; the action

itself and the game state it generates. The num_agents array includes all the agents in the game.

Then I checked every future state (successor) in the array and use it to call the function minimax that yields a value represents how good the state for the pacman is. If this value returned by the minimax function is greater than the value of the previous state, then I add the action that leads to this state to the array bestActions. This array contains the sequence of actions that are supposed to achieve the maximum benefit for the pacman.

```
def minimax(agent, all_agents, state, depth, evalFunc):

    if depth <= 0 or state.isWin() == True or state.isLose() == True:
        return evalFunc(state)

    if agent == 0:
        stat = float("-inf")
    else:
        stat = float("inf")

    actions = state.getLegalActions(agent)
    successors = [state.generateSuccessor(agent, action) for action in actions]
    for i in range(len(successors)):
        successor = successors[i]
        if agent == 0:
            stat = max(stat, minimax(all_agents[agent + 1], all_agents, successor, depth, evalFunc))
        elif agent == all_agents[-1]:
            stat = min(stat, minimax(all_agents[0], all_agents, successor, depth - 1, evalFunc))
        else:
            stat = min(stat, minimax(all_agents[agent + 1], all_agents, successor, depth, evalFunc))

    return stat
```

Figure (6)

Figure (6) shows in details the recursive function “minimax”. This function takes the competitor agent, array of the whole agents, the game state, the depth and the evaluation function as inputs.

Note: The evaluation function used in this part is a default one written by the project’s author.

At the beginning of the function, I checked if the game already reaches a terminate state by wining, losing or the depth is zero (the options tree is fully checked and no one and there is no other options to checked). When we reach any one of the three options, the game is terminated and the state passes to the evaluation function to return the final value.

Then I checked if the agent is the pacman or any ghost to initialize the variable “stat” (it is different than “state”) with the suitable value. Negative infinity it the agent is the pacman because pacman always want the maximum in the comparison, and infinity if the agent is the ghost because it always wants the minimum value in the comparison.

I defined an array “actions” to include all the legal actions for the current agent, also an array “successors” to include all the game states that are generated by the actions.

After that, I loop over all the successors (game states) to evaluate each one of them according to which agent is playing now. If the agent index is 0, then it is the pacman so, I will compare the current stat (which is initially negative infinity) with the output stat produced by recalling of "minimax" function (that gets the following agent in the array of agents), and choose the maximum. If the agent index is the last one in the agents' array, I compared the value of the variable "stat" (which is initially infinity) with the value returned by recalling of "minimax" function (that gets the following agent in the array of agents), and choose the minimum. The previous case is also applied on any other ghost in the array but I divide them into two cases (the case of any ghost and the case of the last ghost) because determining the next ghost is with different approaches. The program will recursively calling the "minimax" function until reaching a termination state (one of the three I talked about them previously).

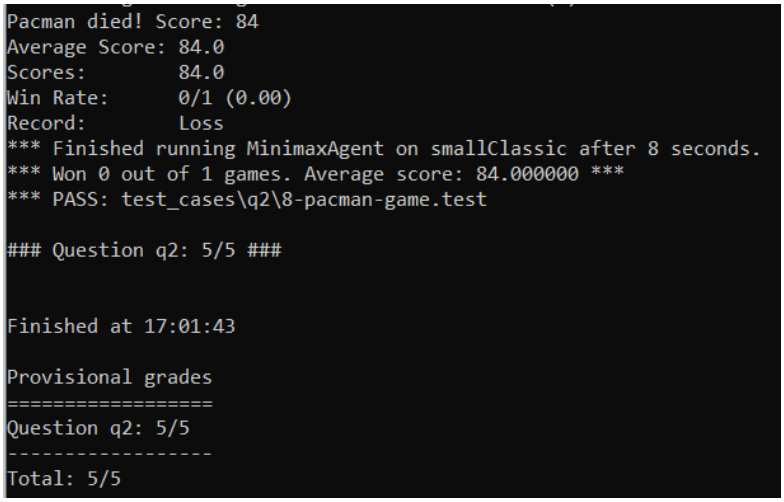


Figure (7)

Figure (7) shows a sample of the result when running the auto grader. You can notice that the pacman loses the game but I still get the full mark. It is normal that pacman here loses some games because it sometimes choose to die when it is obvious that it will lose at the end to decrease the cost of living. The pacman agent here evaluates the state not the action so, it sometimes just try not to be eaten. The criteria of evaluation in this question is following the correct minimax behavior not to always win as the first question.

Third Question: Alpha-Beta Pruning

In this question, I developed a pacman agent that behaves according to alpha-beta pruning algorithm. It is very similar to the minimax algorithm but it is faster because it avoids exploring the nodes of the tree that are not effective in deciding the best action so, the main difference is that this algorithm speeds up the game.

```
v = float("-inf")
alpha = float("-inf")
beta = float("inf")
bestAction = []
agent = 0
actions = gameState.getLegalActions(agent)
successors = [(action, gameState.generateSuccessor(agent, action)) for action in actions]
for successor in successors:
    temp = minimaxPrune(1, range(gameState.getNumAgents()), successor[1], self.depth, self.evaluationFunction,
                        alpha, beta)

    if temp > v:
        v = temp
        bestAction = successor[0]

    if v > beta:
        return bestAction

    alpha = max(alpha, v)

return bestAction
```

Figure (8)

Figure (8) shows the “getAction” function that returns the list of best action to be taken by the agent. It is similar to the logic of minimax except some additions related to alpha and beta. Alpha is initialized with negative infinity because it represents the maximum best option in the tree to the root. Beta is initialized with infinity because it represents the minimum best option in the tree to the root. Another variable “v” is initialized with negative infinity. The remaining arrays and variables includes the same elements as minimax code.

The next step is looping over the successors list, and then calculates a value “temp” for each successor by calling “minimaxPrune” function. Then, comparing the “temp” with “v”. If “temp” is the greatest, it replaces ”v” and then the best action will be the action that leads to the current successor (which results the great v). If “v” is also bigger than beta, then terminate now and return the list “bestAction”. If not, just update the current alpha with the maximum between the value of “v” and the value of the current alpha. The “getAction” function returns the list “bestAction” after the end of the loop, or when “v” is bigger than beta.

```
def minimaxPrune(agent, agentList, state, depth, evalFunc, alpha, beta):
    if depth <= 0 or state.isWin() == True or state.isLose() == True:
        return evalFunc(state)

    if agent == 0:
        v = float("-inf")
    else:
        v = float("inf")

    actions = state.getLegalActions(agent)
    for action in actions:
        successor = state.generateSuccessor(agent, action)

        if agent == 0:
            v = max(v, minimaxPrune(agentList[agent + 1], agentList, successor, depth, evalFunc, alpha, beta))
            alpha = max(alpha, v)
            if v > beta:

                return v

        elif agent == agentList[-1]:
            v = min(v, minimaxPrune(agentList[0], agentList, successor, depth - 1, evalFunc, alpha, beta))
            beta = min(beta, v)
            if v < alpha:

                return v

        else:
            v = min(v, minimaxPrune(agentList[agent + 1], agentList, successor, depth, evalFunc, alpha, beta))
            beta = min(beta, v)
            if v < alpha:

                return v

    return v
```

Figure (9)

Figure (9) shows the recursive function “minimaxPrune”. It is similar to the “minimax” function, which was also recursive. It takes the same inputs in addition to the values of alpha and beta. It also loops over all the legal actions and from them it gets the possible successors. It has the same three cases with the same logic of “minimax” except for the following additions:

- In the case of the agent is the pacman, it computes the value “v”, then it compares “v” with the current alpha and updates alpha with the maximum. It also checks if “v” is greater than beta. If it is true, it terminates and return the value “v” because the pacman always the maximum alpha.
- In the second and third cases, it does the same. It computes the value “v”, then it compares “v” with the current beta and updates beta with the minimum. It also checks if “v” is less than alpha. If it is true, it terminates and returns the value “v” because the pacman always the maximum alpha.

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

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


Finished at 17:13:38

Provisional grades
==================
Question q3: 5/5
------------------
Total: 5/5
```

Figure (10)

Figure (10) shows the result of the auto grader in evaluating the alpha-beta pruning. It is obvious that this is the same result of minimax algorithm but it finished quickly; only in 4 seconds while the minimax took 8 seconds. This is exactly the expected result because the two algorithms is the same except that the alpha-beta pruning is faster.

Fourth Question: Expectimax

In this question, I developed an agent that behaves according to expectimax algorithm. The main difference between this approach and the minimax is that minimax assumes that the opponent is always acting optimally while the expectimax does not assume this information so; it is the best choice while dealing when the random ghosts in this game. In minimax, there are max node which represented by the pacman and mini nodes which represent by the ghosts , while in expectimax, there are max nodes which represented by the pacman and chance nodes that represented by the ghosts. Therefore, here the ghosts do not choose the minimum but choose the expectation of the utilities. Since the ghosts are random, all the probabilities are equal.

```
value = float("-inf")
bestAction = []
agent = 0
actions = gameState.getLegalActions(agent)
successors = [(action, gameState.generateSuccessor(agent, action)) for action in actions]
for successor in successors:
    result = expectimax(1, range(gameState.getNumAgents()), successor[1], self.depth, self.evaluationFunction)
    if result > value:
        value = result
        bestAction = successor[0]
return bestAction
```

Figure (11)

Figure (11) shows the “getAction” function, it is the exactly the same as the “getAction” function in minimax because there is no change in the logic of how we get the best action, the

difference here in the approach of traversing the tree. Instead of calling “minimax” function, we here call “expectimax” function. It takes the same inputs of “minimax”.

```
def expectimax(agent, agentList, state, depth, evalFunc):

    if depth <= 0 or state.isWin() == True or state.isLose() == True:
        return evalFunc(state)

    if agent == 0:
        v = float("-inf")
    else:
        v = 0

    actions = state.getLegalActions(agent)
    successors = [state.generateSuccessor(agent, action) for action in actions]
    for successor in successors:
        if agent == 0:
            v = max(v, expectimax(agentList[agent + 1], agentList, successor, depth, evalFunc))
        elif agent == agentList[-1]:
            v = v + expectimax(agentList[0], agentList, successor, depth - 1, evalFunc)
        else:
            v = v + expectimax(agentList[agent + 1], agentList, successor, depth, evalFunc)

    if agent == 0:
        return v
    else:
        return v / float(len(successors))
```

Figure (12)

Figure (12) shows the recursive function “expectimax”. The termination cases are the same as “minimax” as shown. Then, I checked if the agent is the pacman to set a variable “v” with value “-inf” or a ghost to set “v” with the value “0”. The arrays “actions” and “successors” have the same meaning as usual. The three cases also are the same as any previous algorithm except that when the agent is the ghost, we do not get the minimum anymore but we calculate the average of the successors because we assume that they all have equal probability. So, if the agent is the pacman, return the maximum “v”, if it is a ghost, return “v/number of successors”.

```
*** 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 5 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 15:52:53

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

Figure (13)

Figure (13) shows the result of the auto grader on question 4. It is normal that the pacman loses the game. It wins the game other times. I also noticed that the pacman here is a bit more risky, sometimes it takes actions that are risky but it ends with big utility and other times with less utility so, it win sometimes and lose other ones.

Fifth Question: Evaluation Function

In this question, I improved the evaluation function that had been used in the first question. I used the same factors used in the first question, the distance between the pacman and the nearest food and the distance between the pacman and the ghost, in addition to other factors such the remaining food and the remaining capsules. In addition, I added another factor to make a penalty on the loss state or a reward on the win state.

```
newPos = currentGameState.getPacmanPosition()
newFood = currentGameState.getFood().asList()

distance_to_food = float('inf')
for food in newFood:
    d_food = manhattanDistance(newPos, food)
    distance_to_food = min(distance_to_food, d_food)

d_ghost = 0
for ghost in currentGameState.getGhostPositions():
    d_ghost = manhattanDistance(newPos, ghost)
    if (d_ghost < 2):
        return -float('inf')
```

Figure (14)

This part in figure (14) is almost the same as the evaluation function in question (1) except that here, I made the pacman to move in the opposite direction of the ghost's position if the distance between them is two steps only (in the first question, the distance was 5 steps).

```
remaining_food = currentGameState.getNumFood()
remaining_capsules = len(currentGameState.getCapsules())

foodLeft_factor = 1000000
capsLeft_factor = 10000
foodDist_factor = 1000

additionalFactors = 0
if currentGameState.isLose():
    additionalFactors -= 50000
elif currentGameState.isWin():
    additionalFactors += 50000
```

Figure (15)

In this part, I added the list “remaining_food” that contains the number of food points that are still in the game, and the array “remaining_capsules” that includes the number of the capsules that still exist.

In addition, I added weights for some factors that controls how much I found that factors positively affect the pacman’s behavior.

- “foodLeft_factor” is the weight that I multiply the reciprocal of remaining food number with it.
- “capsLeft_factor” is the weight that I multiply the reciprocal of remaining capsules number with it.
- “foodDist_factor” ” is the weight that I multiply the reciprocal of the distance to nearset food with it.

The values of the weights are got by many trials. I tried a range of numbers until I reached those numbers, which yielded the expected behavior.

Finally, I added “additionalFactors” that are considered a reward on the win state or a penalty on the lose state.

```
return 1.0 / (remaining_food + 1) * foodLeft_factor + 1.0/ d_ghost + \
       1.0 / (distance_to_food + 1) * foodDist_factor + \
       1.0 / (remaining_capsules + 1) * capsLeft_factor + additionalFactors
```

Figure (16)

Figure (16) shows the return of the function. It is clear that it is a summation of all the previous factors.

Note: I always add (1) in the denominator of the reciprocal to avoid dividing be zero.

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

Pacman emerges victorious! Score: 960
Pacman emerges victorious! Score: 1172
Pacman emerges victorious! Score: 1371
Pacman emerges victorious! Score: 971
Pacman emerges victorious! Score: 1157
Pacman emerges victorious! Score: 1380
Pacman emerges victorious! Score: 1182
Pacman emerges victorious! Score: 1174
Pacman emerges victorious! Score: 1171
Pacman emerges victorious! Score: 974
Average Score: 1151.2
Scores:        960.0, 1172.0, 1371.0, 971.0, 1157.0, 1380.0, 1182.0, 1174.0, 1171.0, 974.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\q5\grade-agent.test (6 of 6 points)
```

Figure (17)

Figure (17) shows the result of the auto grader on question (5). My evaluation function passed the challenging requirements and got the full mark.