Spring 2022 Introduction to Artificial Intelligence

Report of Homework #3

Student name: 劉子齊 Jonathan
Student ID: 0716304

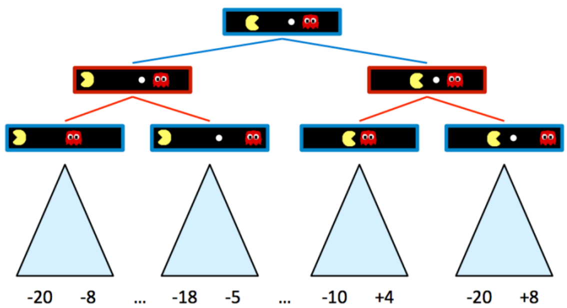
Part 1 : Adversarial search

Part 1-1: Minimax Search

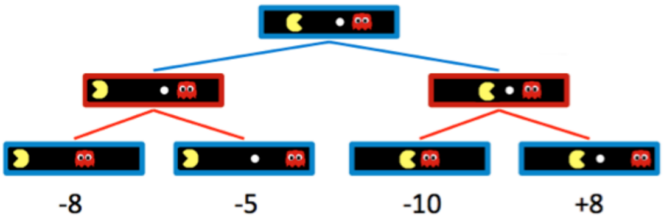
When the computer wants to make the next move in Pacman, of course it will play the one with the most points, which is eating all the food in this case, but this can easily fall into the opponent's trap, which is being eaten by the ghosts.

Therefore, a reasonable idea is to divide all levels into two categories of enemy and us, which will be Pacman and the Ghost in ours case. The more points the level under our side has, the better, and the level under the other side loses as few points as possible.

Also, we can't assume that the opponent is a fool, so at every level, we have to think that "the opponent may make the move that will cost us the most points", and we must choose the strategy of "minimizing maximum points loss" as much as possible. As a result, this strategy makes the Minimax Search.



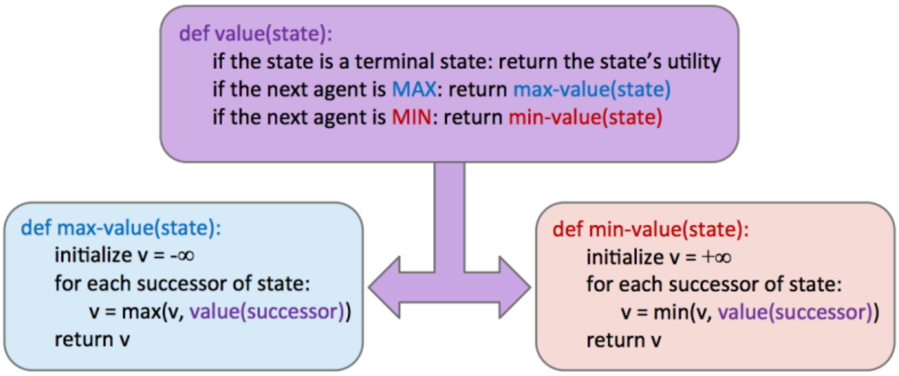
In the figure above, blue nodes correspond to nodes that Pacman controls and can decide what action to take, which will pick the choice with the max value, while red nodes correspond to ghost-controlled nodes, which will pick the minimum. Now let's move onto the second depth of the tree in the figure above.



As we can observe from the figure above, the blue Pacman nodes chose the option with the max value, which Pacman believes to be the best choice. The minimax algorithm only maximizes over the children of nodes controlled by Pacman, while minimizing over the children of nodes controlled by ghosts. Hence, the two ghost nodes above have values of min(—8,—5) = —8 and min(—10,+8) = —10 respectively. Correspondingly, the root node controlled by Pacman has a value of max(—8,—10) = —8. As a result, the Pacman will get −8 as the score of this game.

However, if I was the Pacman in this game, I will mot be satisfied, since I knew that I could have get +8 as my final score, if I chose the right way. Hence, we will need to have the Pacman put some bet to move to way which is not so straightforward and will take some risks correspondingly.

When implementing the Minimax algorithm, I found it kind of similar to DFS (Deep First Search), which they both start with the leftmost terminal node and all the way to the right of the game tree. Basically, I followed the pseudocode found on a website of UC Berkeley in the figure below.



```python
class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (par1-1)
    """
    def getAction(self, gameState):
        """...
        "*** YOUR CODE HERE ***"
        # Begin your code

        pacman_legal_actions = gameState.getLegalActions(0)

        # print()
        # print(pacman_legal_actions)

        max_value = float('-inf')
        decision  = None

        for action in pacman_legal_actions:

            action_score = self.Min_Value(gameState.getNextState(0, action), 1, 0)
            # print("action: ", action, "action_score: ", action_score)

            if ((action_score) > max_value ):
                max_value = action_score
                decision = action

        # print("final_call", decision)
        return decision

        util.raiseNotDefined()
        # End your code
```

At the very first of my program in the figure above, I call the getLegalActions() function to get all the possible actions the Pacman can move for the next step. Besides, I declared the initial max_value, which determines the final action, as a negative infinite float number, and the decision as a NULL.

After declaring, I construct a loop which depends on the elements in the list named “pacman_legal_actions”. For each action in the list, I first obtain the “action_score” by the Minimax algorithm, then I compare all of them throughout the whole loop and get the final decision, which is the action with the highest “action_score”.

```python
    def Max_Value (self, gameState, depth):

        # end of the game
        if(depth == self.depth):
            return self.evaluationFunction(gameState)
        # dead
        elif(len(gameState.getLegalActions(0)) == 0):
            return self.evaluationFunction(gameState)

        # print("Max_Value Self Depth: ", self.depth)

        return max([self.Min_Value(gameState.getNextState(0, action), 1, depth)
                    for action in gameState.getLegalActions(0)])
```

For the Minimax algorithm, I separated it into two parts, one is for the minimum value, and the other for the maximum. For the Max_Value() part, I first confirm if the current depth equals to the “self.depth”, which is the end of the game tree. If we’ve reach the end of the game tree, then I call the evaluationFunction() to get the final result and return it. Also, I’ll check whether there are no legal actions left, which might mean the Pacman just ran into a ghost and died. In this case, I’ll also return the final result obtained by the evaluationFunction().

If none of the two scenarios mentioned above happened, I’ll call the Min_Value function, which will be mentioned later, and iterate it through all the legal actions obtained by the getLegalActions(0), which returns a list contains all the legal actions of Pacman since “0” stands for Pacman here. As I keep getting value from the Min_Value function, I’ll compare them all and keep the maximum of all of them. Finally, I’ll return the maximum value.

```python
    def Min_Value (self, gameState, agentIndex, depth):

        # dead
        if(len(gameState.getLegalActions(agentIndex)) == 0):
            return self.evaluationFunction(gameState)

        # when the game continues
        if(agentIndex < gameState.getNumAgents() - 1):
            return min([self.Min_Value(gameState.getNextState(agentIndex, action), agentIndex + 1, depth)
                        for action in gameState.getLegalActions(agentIndex)])
        # when the last ghost remaining
        else:
            return min([self.Max_Value(gameState.getNextState(agentIndex, action), depth + 1)
                        for action in gameState.getLegalActions(agentIndex)])
```

For the Min_Value() part, I’ll check whether there are no legal actions left, which might means the Pacman just ran into a ghost and died. In this case, I’ll also return the final result of the current gameState obtained by the evaluationFunction().

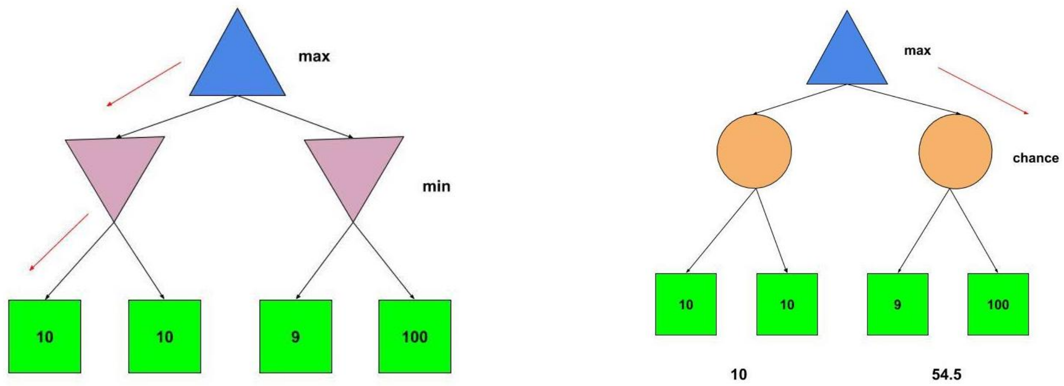
Then I’ll check if there’s still agents remaining. If there are still agents remaining, I’ll call the Min_Value function, and iterate it through all the legal actions obtained by the getLegalActions(agentIndex), which returns a list contains all the legal actions of the corresponding agent. As I keep getting value from the Min_Value function, I’ll sum them up and return the final value.

If there’s no agents remaining, I’ll call the Max_Value function, and iterate it through all the depth and all the legal actions obtained by the getLegalActions(agentIndex), which returns a list contains all the legal actions of the corresponding agent. As I keep getting value from the Min_Value function, I'll compare them all and keep the minimum of all of them. Finally, I'll return the minimum value.

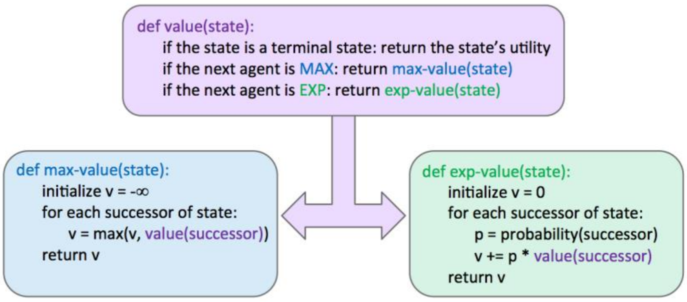
Part 1-2: Expectimax Search

I'll take Expectimax as a variation of Minimax Algorithm, which replaces minimizer nodes into chance nodes in the original Minimax game tree. As we know that the minimizer plays optimally, it makes sense to go to the left. But what if there is a possibility of the minimizer making a mistake. Therefore, going right might sound more appealing or may result in a better solution.

Hence, in the Expectimax Search, I replace minimizer nodes by chance nodes, which takes the average of the agent numbers, which can be seen in the figure below. On the left, we have the original Minimax Search, and on the left we have the tree of Expectimax Search.



For the implementation, I followed the pseudocode found on a website of UC Berkeley in the figure below. Which we can find both of them are basically the same, except for changing the Min_Value() function of the Minimax Search into Exp_Value() function which contains the chance node, of the Expectimax Search.




```python
class ExpectimaxAgent(MultiAgentSearchAgent):
    """
      Your expectimax agent (part1-2)
    """

    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 ***"
        # Begin your code

        pacman_legal_actions = gameState.getLegalActions(0)

        # print()
        # print(pacman_legal_actions)

        max_value = float('-inf')
        decision  = None

        for action in pacman_legal_actions:

            action_score = self.Exp_Value(gameState.getNextState(0, action), 1, 0)
            # print("action: ", action, "action_score: ", action_score)

            if ((action_score) > max_value ):
                max_value = action_score
                decision = action

        # print("final_call", decision)
        return decision

        util.raiseNotDefined()
        # End your code
```

```python
    def Max_Value (self, gameState, depth):

        # end of the game
        if(depth == self.depth):
            return self.evaluationFunction(gameState)
        # dead
        elif(len(gameState.getLegalActions(0)) == 0):
            return self.evaluationFunction(gameState)

        # print("Max_Value Self Depth: ", self.depth)

        return max([self.Exp_Value(gameState.getNextState(0, action), 1, depth)
                    for action in gameState.getLegalActions(0)])
```

By the figure, we can find the get_action() function and the Max_Value() function are almost the same. I also checked the same condition and return the same value in both functions as I did in the Minimax Search. The only thing changed was in the Max_Value() function, which I changed the original Min_Value() function into Exp_Value() function.

```
    def Exp_Value (self, gameState, agentIndex, depth):

        num_actions = len(gameState.getLegalActions(agentIndex))

        # dead
        if(len(gameState.getLegalActions(agentIndex)) == 0):
            return self.evaluationFunction(gameState)

        # when the game continues
        if(agentIndex < gameState.getNumAgents() - 1):
            return sum([self.Exp_Value(gameState.getNextState(agentIndex, action), agentIndex + 1, depth)
                            for action in gameState.getLegalActions(agentIndex)]) / float(num_actions)\
        # when the last ghost remaining
        else:
            return sum([self.Max_Value(gameState.getNextState(agentIndex, action), depth + 1)
                            for action in gameState.getLegalActions(agentIndex)]) / float(num_actions)
```

In the Exp_Value() function, basically, it’s also kind of similar to the Min_Vlaue() function of the Minimax Search. I checked whether there are no legal actions left, which might mean the Pacman just ran into a ghost and died. In this case, I’ll also return the final result of the current gameState obtained by the evaluationFunction().

Then I’ll check if there’s still agents remaining. If there are still agents remaining, I’ll call the Exp_Value() function, and iterate it through all the legal actions obtained by the getLegalActions(agentIndex), which returns a list contains all the legal actions of the corresponding agent. As I keep getting value from the Exp_Value() function, I’ll sum them up. After getting the final sum value, I divided the sum with the total num of the actions and return the final value.

If there’s no agents remaining, I’ll call the Max_Value() function, and iterate it through all the depth and all the legal actions obtained by the getLegalActions(agentIndex), which returns a list contains all the legal actions of the corresponding agent. As I keep getting value from the Max_Value() function, I’ll sum them up. After getting the final sum value, I divided the sum with the total num of the actions. Finally, I’ll return the final value.

Part 1-3: Evaluation Function (Bonus)

```python
def betterEvaluationFunction(currentGameState):
    """...
    "*** YOUR CODE HERE ***"
    # Begin your code

    pacman_position = currentGameState.getPacmanPosition()
    ghost_positions = currentGameState.getGhostPositions()

    #number of remaining capsule
    capsule_num = len(currentGameState.getCapsules())

    # print()
    # print(currentGameState.getCapsules())

    game_score = currentGameState.getScore()

    # ------ food area ------
    # get the position of all food
    food_place = currentGameState.getFood()
    food_place = food_place.asList()

    #number of remaining food
    food_num = len(food_place)

    # print()
    # print(food_place)

    closest_food = 1
    food_distances = [manhattanDistance(pacman_position, food_position) for food_position in food_place]
    # print("food_distances: ", food_distances)

    if food_num > 0:
        closest_food = min(food_distances)

    #distance of ghost to us
    for ghost_position in ghost_positions:

        ghost_distance = manhattanDistance(pacman_position, ghost_position)

        # If ghost is too close to us, then its time to stop eating and escape lol
        if ghost_distance < 5:
            closest_food = 878787

    #calculate all the weighted features and sum them up
    features = [(1.0 / closest_food), game_score, food_num, capsule_num]
    weights = [10, 200, -100, -10]
    return sum([feature * weight for feature, weight in zip(features, weights)])

    util.raiseNotDefined()
    # End your code
```

In part 1-3, I redefined the way the program evaluates the better option of the following move. First, I declared the two variables, "pacman_position" and "ghost_positions", which obtains the current position of the Pacman and all the ghosts in the current match. Then I get the current "food_num" and the "capsule_num" to know how many food or capsules are lest in the map. Also, I declared "food_place" which is a list that store the position of all the food in the form of a two-dimensional coordinate.

After all the variables are set, I set the value of "closest_food" to 1, which "closest_food" indicated the distance of the closest food to the Pacman. Then I get a list of all the food distances by iterating through all the food positions in the "food_place" list and calculate the distance through Manhattan distance. With the list of distances to all the food, if there' s still food lest in the map, I' ll set the value of "closest_food" to the minimum value in the list of food distances.

Next, I started to deal with the ghosts. For every ghost position in the list named "ghost_positions", I obtain the distance from the Pacman to the ghost through Manhattan distance. When the distance is smaller than 5, my Pacman will leave the food alone, and start to escape to keep itself alive.

For last, I defined a list containing the reciprocal of "closest_food", "game_score", "food_num", and "capsule_num". Also, the other list with all the corresponding weight that I would like to give every variable in the previous list, which are 10, 200, −100, and −10. Then I iterate through them simultaneously through the zip() function and multiply them. Finally, I sum the result of multiplication up, and return it as the final result.

Part 2: Q-learning

Part 2-1: Value Iteration

In this part batch version of Value Iteration is implemented. The values of states in previous iteration is used for updating values of states in next iteration which might not always be the case with online update.

The getStates() function returns all the states in the environment. The function getPossibleActions(state) returns the possible actions from state. The function getQValue(state, action) returns the QValue. I iterated through all the iterations, states and actions one by one, then get the corresponding value through the three functions mentioned above.

After calculating every state value through the functions mentioned above, I updated the value for each state by using the argmax() function. Finally, I copy back the new value through the copy() function and return it.

```
    def runValueIteration(self):
        # Write value iteration code here
        "*** YOUR CODE HERE ***"
        # Begin your code

        for state in self.mdp.getStates():
            self.values[state] = float(0)

        for iteration in range(self.iterations):
            next_values = self.values.copy()

            for state in self.mdp.getStates():
                state_values = util.Counter()

                for action in self.mdp.getPossibleActions(state):
                    state_values[action] = self.getQValue(state, action)

                next_values[state] = state_values[state_values.argMax()]

            self.values = next_values.copy()

        return self.values

        util.raiseNotDefined()
        # End your code
```

In the computeQValueFromValues() function in the figure below, I first obtain all the transition probabilities through the getTransitionsStatesAndProbs(state, action) function. Also, I set the initial value of the QValue to 0.0, which is a float type number.

For every transition probability in all the transition probabilities obtained, I extract the Tstate and the probability in each transition. With the Tstate value and the probability, I updated the Qvalue through summing up every possible value in all the transitions. By following the function below, I obtain all the QValues in this function.

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

Note: I only calculate all the Qvalues in this function, I'll determine the max Qvalue and return it in the next function.

```python
    def computeQValueFromValues(self, state, action):
        """
          Compute the Q-value of action in state from the
          value function stored in self.values.
        """
        "*** YOUR CODE HERE ***"
        # Begin your code

        transition_probs = self.mdp.getTransitionStatesAndProbs(state, action)
        QValue = float(0)

        for transition in transition_probs:
            Tstate, prob = transition
            QValue += prob * (self.mdp.getReward(state, action, Tstate) + self.discount * self.getValue(Tstate))

        return QValue

        util.raiseNotDefined()
        # End your code
```

In the computeActionFromValue() function, I first check whether the game is about to terminal, if not, I will start to calculate.

First, I initiate the QValue by the Counter() function in the utisl.py, which is a dictionary with a default value of zero. Also, I obtained all the actions through the getPossibleActions().

With all the actions, I get the QValue for each action in a list, then return the maximum QValue with the argMax() function, just exactly as the following function I mentioned earlier.

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

```python
    def computeActionFromValues(self, state):
        """...
        "*** YOUR CODE HERE ***"
        # Begin your code

        #check for terminal

        if(self.mdp.isTerminal(state)):
            return None
        else:
            QValues = util.Counter()
            actions = self.mdp.getPossibleActions(state)

            for action in actions:
                QValues[action] = self.computeQValueFromValues(state, action)

            return QValues.argMax()

        util.raiseNotDefined()
```

Part 2-2: Q-learning

In the very first _init_() function, the only thing I did is I initiated the Qvalues to a dictionary with a default value of zero by the Counter() function in the util.py.

```python
    def __init__(self, **args):
        "You can initialize Q-values here..."
        ReinforcementAgent.__init__(self, **args)

        "*** YOUR CODE HERE ***"
        # Begin your code

        self.QValues = util.Counter()
        print("Alpha:", self.alpha)
        print("Discount: ", self.discount)
        print("Exploration: ", self.epsilon)

        # End your code
```

In the getQvalue() function, I return the Qvalues, which should return the Q Value if the state actually exists, and will return 0.0, as I initiated in the _init_() function if the state didn't exist.

```python
    def getQValue(self, state, action):
        """
          Returns Q(state,action)
          Should return 0.0 if we have never seen a state
          or the Q node value otherwise
        """
        "*** YOUR CODE HERE ***"
        # Begin your code

        return self.QValues[(state, action)]

        util.raiseNotDefined()
        # End your code
```

In the computeValueFromQvalues() function, I first obtain all the Q values for all the legal actions through the getQValue() function. If I didn't get any value, which means there's no legal actions, I'll return 0.0 for the result. Else, I'll return the maximum element from all the values.

```python
    def computeValueFromQValues(self, state):
        """...
        "*** YOUR CODE HERE ***"
        # Begin your code

        values = [self.getQValue(state, action) for action in self.getLegalActions(state)]

        if(values):
            return max(values)
        else:
            return 0.0

        util.raiseNotDefined()
        # End your code
```

For the computeActionFromQvalues() function, for each action from all the legal actions, I calculate its Q value through the getQValue() function. If the QValue is greater than the max_QValue or there's no max action, then I'll update the max_QValue and the max_action through the Qvalue and the action correspondingly. After iterating through all the legal actions, I will return the final biggest max_action as the final result.

```python
def computeActionFromQValues(self, state):
    """...
    "*** YOUR CODE HERE ***"
    # Begin your code

    max_action = None
    max_Qvalue = 0

    for action in self.getLegalActions(state):
        Qvalue = self.getQValue(state, action)

        if(Qvalue > max_Qvalue or max_action == None):
            max_Qvalue = Qvalue
            max_action = action

    return max_action

    util.raiseNotDefined()
    # End your code
```

For the update() function, I divided it into two parts. For the first part, I multiplied the Q value obtained by the getQValue() function with (1 – alpha). For the second part, I first check if there still exist the next action, if not, the sample will be set to the value of the reward. Else, the sample value will be the reward plus the maximum value obtained from the getQValue function through all the legal next actions. Then multiplies the sample value with alpha, which results the second part. Finally, I set the Q value of the current state and action into the addition of the first part and the second part.

```python
def update(self, state, action, nextState, reward):
    """...
    "*** YOUR CODE HERE ***"
    # Begin your code

    first_part = (1 - self.alpha) * self.getQValue(state, action)

    if len(self.getLegalActions(nextState)) == 0:
        sample = reward
    else:
        sample = reward + (self.discount * max([self.getQValue(nextState, next_action)
                                   for next_action in self.getLegalActions(nextState)]))

    second_part = self.alpha * sample

    self.QValues[(state, action)] = first_part + second_part

    # util.raiseNotDefined()
    # End your code
```

Part 2-3: epsilon-greedy action selection

For the Epsilon-Greedy Action Selection, I modify the getAction() function directly. The main idea here is that exploration of state space has to be done first and after getting good policy, exploitation of that policy has to done. With epsilon probability choose a random action from a state otherwise choose the present optimal policy.

Hence, from my code, we can find that as the recommendation of the comments, I chose to use the flipCoin() function to decide whether to get the action from the choice() function to pick randomly from the list of legal actions, or from the getPolicy() function directly. After that, I'll return the action which I get from my final randomly decision.

```python
    def getAction(self, state):
        """...
        # Pick Action
        legalActions = self.getLegalActions(state)
        action = None
        "*** YOUR CODE HERE ***"
        # Begin your code

        if(util.flipCoin(self.epsilon)):
            action = random.choice(legalActions)
        else:
            action = self.getPolicy(state)

        return action

        util.raiseNotDefined()
        # End your code
```

Part 2-4: Approximate Q-learning

For approximate Q learning, I implemented through the function provided by the CS188 course of the UC Berkeley.

$$\text{difference} = \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s, a)$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\text{difference}]$$

$$w_i \leftarrow w_i + \alpha\,[\text{difference}]\, f_i(s,a)$$

First of all, I implemented the getQValue() function. I obtained all the features by the getFeatures() function. For each feature obtained, I multiplied its weight, which I obtained it through the getWeight() function that can extract the weight of a particular function from the getWeights() function, with the feature. Then I sum them up as the "QValue" and return the final QValue.

Then I implement the update() function, which I basically followed the formula mentioned above. I initiate the difference as "the reward" plus "the multiplication of the discount and the value of next state obtained by the getValue() function" subtract "the Q Value of the current state and action". For each feature obtained by the getFeatures function, I set the weights for each feature as the original weight of the feature plus the multiplication of alpha, difference, and the feature value.

```python
class ApproximateQAgent(PacmanQAgent):
    """...
    def __init__(self, extractor='IdentityExtractor', **args):
        self.featExtractor = util.lookup(extractor, globals())()
        PacmanQAgent.__init__(self, **args)
        self.weights = util.Counter()

    def getWeights(self):
        return self.weights

    def getWeight(self,feature):
        return self.weights[feature]

    def getQValue(self, state, action):
        """...
        "*** YOUR CODE HERE ***"
        # Begin your code
        # get weights and feature

        features = self.featExtractor.getFeatures(state, action)
        QValue = 0.0

        for feature in features :
            QValue += self.getWeight(feature) * features[feature]

        return QValue

        util.raiseNotDefined()
        # End your code

    def update(self, state, action, nextState, reward):
        """...
        "*** YOUR CODE HERE ***"
        # Begin your code

        features = self.featExtractor.getFeatures(state, action)
        difference = reward + (self.discount*self.getValue(nextState)) - self.getQValue(state, action)

        for feature in features :
            self.weights[feature] = self.getWeight(feature) + (self.alpha * difference * features[feature])

        # util.raiseNotDefined()
        # End your code


    def final(self, state):
        "Called at the end of each game."
        # call the super-class final method
        PacmanQAgent.final(self, state)
```

Part 3: DQN

```
C:\Users\jonat\Desktop\Intro. to A.I\2022Spring_Artificial_Intellegence\hw3\DQN>
C:\Users\jonat\Desktop\Intro. to A.I\2022Spring_Artificial_Intellegence\hw3\DQN>python pacman.py -p PacmanDQN -n 10000 -x 10000 -l smallClassic
```

```
C:\Users\jonat\Desktop\Intro. to A.I\2022Spring_Artificial_Intellegence\hw3\DQN>python pacman.py -p PacmanDQN -n 200 -x 100 -l smallClassic
Started Pacman DQN algorithm
Model has been trained
```

```
Average Score: 1256.13
Scores:        1552.0, 93.0, 1709.0, 1308.0, 1752.0, 425.0, 1381.0, 1557.0, 1567.0, 1762.0, 540.0, 1157.0, 1566.0, 1677.0, 1744.0, 1559.0, 536.0
, 1160.0, 361.0, 1162.0, 756.0, 1138.0, 51.0, 350.0, 1361.0, 1523.0, 909.0, 1774.0, 1561.0, 1686.0, 1556.0, 1560.0, 222.0, -160.0, 1728.0, 1527.
0, 220.0, 1360.0, 1722.0, 1742.0, 1370.0, 1557.0, 1773.0, 1555.0, 1753.0, 1736.0, 1547.0, 1772.0, 1562.0, 1704.0, 1554.0, 552.0, 1543.0, 1711.0,
 50.0, 1550.0, 245.0, 1168.0, 700.0, -158.0, 1175.0, 1308.0, 1569.0, 1732.0, 1720.0, 1750.0, 1165.0, 1775.0, 1346.0, 1566.0, 1567.0, 1337.0, 131
7.0, 1748.0, 559.0, 1544.0, 1729.0, 50.0, 1741.0, 565.0, 1380.0, 1358.0, 656.0, 1338.0, 114.0, 1758.0, 1757.0, 1770.0, 1363.0, 1355.0, 1549.0, 1
380.0, 1167.0, 1151.0, 1555.0, 1508.0, 1749.0, 759.0, 1575.0, 710.0
Win Rate:      77/100 (0.77)
Record:        Win, Loss, Win, Win, Win, Loss, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Loss, Win, Loss, Win, Loss, Win, Loss, Loss, W
in, Win, Win, Win, Win, Win, Win, Win, Loss, Loss, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, W
in, Win, Loss, Win, Loss, Win, Loss, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Loss, Win, Loss
, Win, Win, Loss, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Loss
```

For part 3, I executed the code provided by the TA, and made a comparison with the methods implemented above, which are Minimax Agent, Expectimax Agent, Approximate Q-learning Agent, and DQN Agent. Following are the results of the four agents, which I executed by the following command lines. I made all my agents to play Pacman 100 times and made a comparison between their performances. For Approximate Q Learning agent and DQG agent, I trained both of them with 10000 episodes.

Minimax:

```
python pacman.py -n 100 -p MinimaxAgent -l smallClassic -q -a depth=3
```

Expectimax:

```
python pacman.py -n 100 -p ExpectimaxAgent -l smallClassic -q -a depth=3
```

Approximate Q Learning:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -q -x 10000 -n 10100 -l smallClassic -q
```

DQN:

```
python pacman.py -p PacmanDQN -a extractor=SimpleExtractor -q -x 10000 -n 10100 -l smallClassic -q
```

Agent Name	Win Rate
Minimax Agent	40%
Expectimax Agent	49%
Approximate Q Learning Agent	83%
DQN Agent	89%

According to the result in the chart above, Minimax and Expectimax agents didn't perform very well, but their effectiveness has been proved in smaller mazes. I believe the reason why adversarial search agents can maintain their performances in smaller mazes is in smaller mazes, all the search ranges, even the very deep one, will still be in a tree with controllable depth. Hence, when it come to bigger mazes, Q learning agents and DQN agents will take their advantages, which makes them takes it like a fish to water.