Introduction to Artificial Intelligence

HW3-Report

0716325-曾正豪

Part.1: Minimax Search

In this part, I follow the instruction in the slide of Lec6 p.29.

```
        # Begin your code (Part 1)
        act, val = self.value(gameState, 0, 0)
        return act
```

```
137     def value(self, gameState, player, current_depth):
138         '''
139         return (action, value)
140         '''
141         if current_depth >= self.depth and player == 0:
142             return 0, self.evaluationFunction(gameState)
143         if gameState.isWin() or gameState.isLose():
144             return 0, self.evaluationFunction(gameState)
145 
146         if player == 0:
147             current_depth += 1
148             return self.max_value(gameState, player, current_depth)
149         else:
150             return self.min_value(gameState, player, current_depth)
151 
152         return 0
```

The first part is function value(). Since it need the action for the getAction function, and value for max_value function and min_value function. So I modify it to return the action and value at the same time. It's parameter is gameState for the state of current game, player for which player currently deciding action, current_depth indicate the depth of the search currently. In the getAction function, I call it with player = 0 (pacman) and depth = 0.

In line 141 to 144, it will check if it reach the terminal state, and will return the evaluation function value. In the first "if", it check whether it reach the maximum depth. In the second "if" it check if the game is over.

In line 146 to 148 if player turns to pacman, indicate that the depth should plus 1, and return the maximum value of child nodes. Otherwise, it's ghost's turn, it should return the minimum value of child nodes in line 149 to 150.

```python
    def max_value(self, gameState, player, current_depth):
        '''
        return (action, value)
        '''
        v = -99999999
        ret_act = None
        action_list = gameState.getLegalActions(player)
        for action in action_list:
            next_state = gameState.getNextState(player, action)
            next_player = (player + 1) % gameState.getNumAgents()
            act, val = self.value(next_state, next_player, current_depth)
            if val > v:
                v = val
                ret_act = action

        return ret_act, v
```

In the max_value function, its parameters' meaning is exactly the same as in the value function.

Firstly, I initialize the value to $-\infty$(I think -99999999 is big enough), and I get all the possible action of the player. For all the action in the action list, I firstly get the next state and next player respectively. And send it into the value function to evaluate the value of this child node. Finally, if the new value is bigger than v, I will update the new value and action to return.

```python
    def min_value(self, gameState, player, current_depth):
        '''
        return (action, value)
        '''
        v = 99999999
        ret_act = None
        action_list = gameState.getLegalActions(player)
        for action in action_list:
            next_state = gameState.getNextState(player, action)
            next_player = (player + 1) % gameState.getNumAgents()
            act, val = self.value(next_state, next_player, current_depth)
            if val < v:
                v = val
                ret_act = action

        return ret_act, v

        # End your code (Part 1)
```

In the min_value function, its parameters' meaning is exactly the same as in the value function.

Firstly, I initialize the value to ∞(I think 99999999 is big enough), and I get all the possible action of the player. For all the action in the action list, I firstly get the next state and next player respectively. And send it into the value function to evaluate the value of this child node. Finally, if the new value is smaller than v, I will update the new value and action to return.

Part.2: Alpha-Beta Pruning

In this part, I modify the code from part.1 by adding parameters alpha and beta.

```
        # Begin your code (Part 2)
        act, val = self.value(gameState, 0, 0, -99999999, 99999999
        return act
```

```
206     def value(self, gameState, player, current_depth, alpha, beta):
207         '''
208         return (action, value)
209         '''
210         if current_depth >= self.depth and player == 0:
211             return 0, self.evaluationFunction(gameState)
212         if gameState.isWin() or gameState.isLose():
213             return 0, self.evaluationFunction(gameState)
214 
215         if player == 0:
216             current_depth += 1
217             return self.max_value(gameState, player, current_depth, alpha, beta)
218         else:
219             return self.min_value(gameState, player, current_depth, alpha, beta)
220 
221         return 0
```

The first part is function value(). I add 2 parameters, alpha and beta. Alpha is for the alpha in Alpha-Beta Pruning, beta is for the beta in Alpha-Beta Pruning. In the getAction function, I call it with player = 0 (pacman) and depth = 0, alpha = -99999999 for -∞, beta = 99999999 for ∞.

In line 210 to 213, it will check if it reach the terminal state, and will return the evaluation function value. In the first "if", it check whether it reach the maximum depth. In the second "if" it check if the game is over.

In line 215 to 217 if player turns to pacman, indicate that the depth should plus 1, and return the maximum value of child nodes. Otherwise, it's ghost's turn, it should return the minimum value of child nodes in line 218 to 219.

```
    def max_value(self, gameState, player, current_depth, alpha, beta):
        '''
        return (action, value)
        '''
        v = -99999999
        ret_act = None
        action_list = gameState.getLegalActions(player)
        for action in action_list:
            next_state = gameState.getNextState(player, action)
            next_player = (player + 1) % gameState.getNumAgents()
            act, val = self.value(next_state, next_player, current_depth, alpha, beta)
            if val > v:
                v = val
                ret_act = action
            if v > beta:
                return ret_act, v
            alpha = max(alpha, v)

        return ret_act, v
```

In the max_value function, its parameters' meaning is exactly the same as in the value function.

Firstly, I initialize the value to -∞(I think -99999999 is big enough), and I get all the possible action of the player. For all the action in the action list, I firstly get the next state and next player respectively. And send it into the value function to evaluate the value of this child node. then, if the new value is bigger than v, I will update the new value and action to return. Next is to check if v is bigger than beta, if yes, means it need to prune the remaining child nodes, and return immediately. Otherwise after that, it update the alpha by max(alpha, v)

```python
    def min_value(self, gameState, player, current_depth, alpha, beta):
        '''
        return (action, value)
        '''
        v = 99999999
        ret_act = None
        action_list = gameState.getLegalActions(player)
        for action in action_list:
            next_state = gameState.getNextState(player, action)
            next_player = (player + 1) % gameState.getNumAgents()
            act, val = self.value(next_state, next_player, current_depth, alpha, beta)
            if val < v:
                v = val
                ret_act = action
            if v < alpha:
                return ret_act, v
            beta = min(beta, v)

        return ret_act, v

        # End your code (Part 2)
```

In the min_value function, its parameters' meaning is exactly the same as in the value function.

Firstly, I initialize the value to -∞(I think -99999999 is big enough), and I get all the possible action of the player. For all the action in the action list, I firstly get the next state and next player respectively. And send it into the value function to evaluate the value of this child node. then, if the new value is smaller than v, I will update the new value and action to return. Next is to check if v is smaller than alpha, if yes, means it need to prune the remaining child nodes, and return immediately. Otherwise after that, it update the beta by min(beta, v)

Part.3: Expectimax Search

In this part, I modify the code from part.1 by replacing the min_value function by expect_value function.

```
277         # Begin your code (Part 3)
278         act, val = self.value(gameState, 0, 0)
279         return act
```

```
281     def value(self, gameState, player, current_depth):
282         '''
283         return (action, value)
284         '''
285         if current_depth >= self.depth and player == 0:
286             return 0, self.evaluationFunction(gameState)
287         if gameState.isWin() or gameState.isLose():
288             return 0, self.evaluationFunction(gameState)
289 
290         if player == 0:
291             current_depth += 1
292             return self.max_value(gameState, player, current_depth)
293         else:
294             return self.expect_value(gameState, player, current_depth)
295 
296         return 0
```

The first part is function value(). It is almost the same as in part 1, besides the 294 line I replace the min_value function by expect_value function

```
298     def max_value(self, gameState, player, current_depth):
299         '''
300         return (action, value)
301         '''
302         v = -99999999
303         ret_act = None
304         action_list = gameState.getLegalActions(player)
305         for action in action_list:
306             next_state = gameState.getNextState(player, action)
307             next_player = (player + 1) % gameState.getNumAgents()
308             act, val = self.value(next_state, next_player, current_depth)
309             if val > v:
310                 v = val
311                 ret_act = action
312 
313         return ret_act, v
```

The max_value function is the same as in part.1

```python
    def expect_value(self, gameState, player, current_depth):
        '''
        return (action, value)
        '''
        ret_act = None
        action_list = gameState.getLegalActions(player)
        total = 0
        for action in action_list:
            next_state = gameState.getNextState(player, action)
            next_player = (player + 1) % gameState.getNumAgents()
            act, val = self.value(next_state, next_player, current_depth)
            total += val

        return ret_act, total / len(action_list)

        # End your code (Part 3)
```

In the expect_value function, its parameters' meaning is exactly the same as in the value function.

Firstly, I initialize the *total* to 0, and I get all the possible action of the player. For all the action in the action list, I firstly get the next state and next player respectively. And send it into the value function to evaluate the value of this child node. Lastly, I add the child node's value to *total*. In the end of this function, I return the *total* divided by the length of possible actions since it is uniformly at random.

Part.4: Evaluation Function (Bonus)

My result:

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

Pacman emerges victorious! Score: 1154
Pacman emerges victorious! Score: 1147
Pacman emerges victorious! Score: 1357
Pacman emerges victorious! Score: 1158
Pacman emerges victorious! Score: 1338
Pacman emerges victorious! Score: 1340
Pacman emerges victorious! Score: 1161
Pacman emerges victorious! Score: 1171
Pacman emerges victorious! Score: 1252
Pacman emerges victorious! Score: 1367
Average Score: 1244.5
Scores:        1154.0, 1147.0, 1357.0, 1158.0, 1338.0, 1340.0, 1161.0, 1171.0, 1252.0, 1367.0
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
***     1244.5 average score (4 of 4 points)
***         Grading scheme:
***          < 500:  0 points
***         >= 500:  2 points
***         >= 1000:  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 ###
```

```
340     # Begin your code (Part 4)
341     pacman = currentGameState.getPacmanPosition()
342
343     ret = 2*currentGameState.getScore()
344     if currentGameState.isLose():
345         ret += -9999999
346     if currentGameState.isWin():
347         ret += 9999999
```

Firstly, I get the position of pacman, and get the score of that state. Then, I will check if this state is win state or loss state. If yes, I will set the return value to a very high or very low number.

```
348
349     for capsule_loc in currentGameState.getCapsules():
350         ret -= 10 * distance(pacman, capsule_loc)
351     ret += 1000 - 200 * len(currentGameState.getCapsules())
```

Secondly, in the for loop, I will let the pacman to get closer to the capsule. The 351 line will make the pacman to eat as many capsules as possible.

```
352
353     for ghost in currentGameState.getGhostStates():
354         if ghost.scaredTimer > 5:
355             ret -= 20*distance(pacman, ghost.getPosition())
356         else:
357             ret += 5*distance(pacman, ghost.getPosition())
```

In this for loop, it will check the state of ghosts, if a ghost is scared, the pacman will try to get closer to the ghost to eat it, otherwise, it will run away the ghost. Here, I set different weight since I want to let the pacman have different tendency to the ghost. In this case, the pacman will have more willing to eat the ghost than run away the ghost. In my test, I found that the pacman will allow a short distance to the ghost not scared rather run very far away. And it will crazily to get closer to the ghost get scared.

```
358
359     ret += 1000 - 20*currentGameState.getNumFood()
360
361     currentFood = currentGameState.getFood()
362     for i, d in enumerate(currentGameState.getFood()):
363         for j, e in enumerate(d):
364             if currentFood[i][j] == True:
365                 ret -= distance(pacman, (i,j))
366     return ret
```

In line 359, I will let the pacman to eat as many as food as possible. In the for loop, the pacman will try to go to the region that has more food.

Finally, return the return value.

```python

def distance(l1, l2):
    return sqrt((l1[0] - l2[0]) ** 2 + (l1[1] - l2[1]) ** 2)

    # End your code (Part 4)
```

Here, I write a distance function to calculate the Euclidean distance between 2 points.

Final result:

```
Provisional grades
==================
Question part1: 25/25
Question part2: 30/30
Question part3: 30/30
Question part4: 10/10
------------------
Total: 95/95
```