AU 342 Principles of Artificial Intelligence

By: Cheng Xinwen(518021910031)
Shen Zhen(518021910149)

HW#: 1

October 17, 2020

I. INTRODUCTION

A. Purpose

This homework consists of designing and implementing a program that plays Chinese Checker. It will exemplify the minimax algorithm, and alpha-beta pruning, and the use of heuristic(evaluation/static) functions to prune the adversarial search.

Chinese checkers is a perfect information game for 2 players. A Chinese Checkers board is shown in Figure 1. The goal of the game is to get 10 pegs or marbles from one's starting position to one's ending position as quickly as possible. Starting and ending positions are always directly across from each other on the board, and players are placed as symmetrically as possible around the board. In a two-player game, the players would start at the top and bottom of the board. The goal of the game is moving all marbles from starting point to the star point on the opposite side of the board. Specially, each player has two color marbles. Player 1 at the top of the board has seven red marbles and three green marbles. Player 2 at the bottom of the board has seven blue marbles and three yellow marbles. If all of the red marbles move to the blue marbles' positions and the green marbles move to the yellow marbles' positions, player 1 would will. Player 2 wins by the same rules as player.

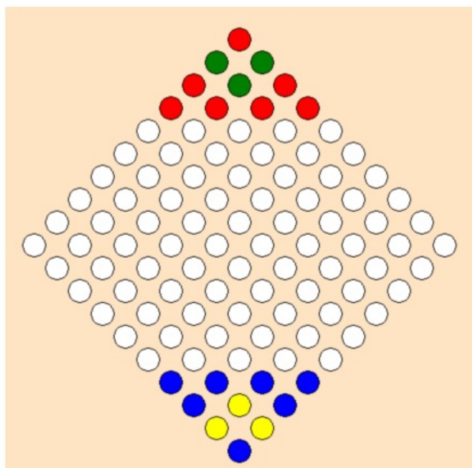


FIG. 1: A Two Player Chinese Checker

The main purpose of this project is to move the marbles of our side as fast as possible to occupy the opponents' original positions with marbles of certain colors in their predetermined right places. The marbles are supposed to move following such rules as below:

- Marbles are moved by stepping to an adjacent position on the board or by jumping over adjacent marbles. One can jump over any player's marbles, or chain together several jumps, but marbles are not removed from the board after a jump. We demonstrate this with a set of consecutive moves in Figure 2.
- Special rules are that every move should be assigned to any marble in one second.
- To prevent illicit competition, if any marble of the player is still in its own triangle, the player is judged to be defeated immediately.
- In case that both sides cannot battle it out after 100 times of iteration,the game ends in a tie.

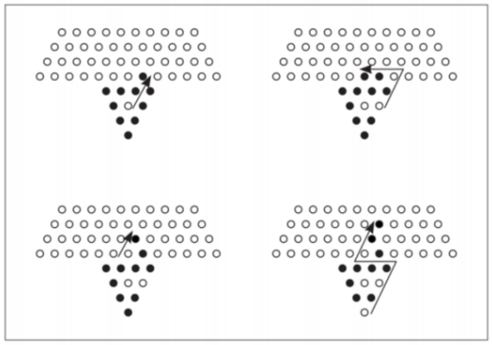


FIG. 2: Move possibilities for Chinese Checker

In this project,we generally apply the minimax algorithm, and alpha-beta pruning, and the use of heuristic functions to prune the adversarial search to our agent. We name our agent RockMinimaxAgent(agent).

B. Environment

There is a minimal amount of equipment to be used in this lab. The few requirements are listed below:

- Windows 10
- Python 3.7
- Pycharm CE

C. Procedure

1. Estimate_Func(self, state) is used to evaluate the values of the present board state,in order to calculate the heuristic value.
2. ALPHA_BETA_SEARCH(self, state) is composed of the minimax algorithm, and alpha-beta pruning, and the use of heuristic functions to prune the adversarial search.
3. getAction(self, state) is used to update next action.

II. CLASS FUNCTIONS

This section will consist of elaborate explanations of the bulit-in functions in the agent class.

A. Estimation on Concerning Values Dynamically

This part of the laboratory was done for estimate the effects which the possible next actions brought to the overall composition of the board.

In the Estimate_Fun,we assign a weight to common marbles and special marbles respectively.Our early goal is to move all the three special marbles as soon as possible and make them stay put until the game ends.So the special marbles should have larger weights because their coming into right positions is the key factor that determines the winner.So in the heuristic function, we first filter those actions which have counteractive on special marbles and will move common marbles into special marbles' destinations.Then we give a large weight to the special ones to get them faster into their positions.

After all of the special marbles arriving at the destination,we increase the weight of the common marbles and decrease special marbles' to zero.In this part,we first tried to make a rule which stipulates common marbles to move as deep into their domain as possible,and on the edge of the domain,they move towards middle unless middle part is occupied.But we found this single rule is not optimal and sometimes the last two common pegs moving towards one side and without direction of its destination it stuck in one side.We later take the unoccupied destination into account and thus the last several actions are directed by unoccupied destination 's column,and finally this rule achieves a better result.

B. Minimax Alpha-Beta Pruning Algorithm

In the minimax algorithm,before we determine each action,we should choose the maximum action from the legal actions.Then for each action,we further consider the possible actions the opponent will take next step.The opponent will probably choose the action which will minimize our score, so we utilize this to make an assumption about what our opponent will take and prevent our next action benefit our opponent.

In the minimax algorithm, we initially write three function:ALPHA_BETA_SEARCH,MAX_VALUE and MIN_VALUE to execute minimax algorithm recursively.But later we also tried use two for loop to execute our algorithm without recursive call.We compared two implementations' time,out of our expectation, their time only has slight difference.

The following is our implementation of MiniMax Algorithm:
Recursive Version:

```
def ALPHA_BETA_SEARCH(self, state):
    v = self.MAX_VALUE(state, -9999, 9999, self.depth)
    return self.dic[v]


def MAX_VALUE(self, state, alpha, beta, depth):
    if depth == 1:
        return self.Estimate_Func(state)
    v = -99999
    depth -= 1
    legal_actions = self.game.actions(state)
    random.shuffle(legal_actions)
    for action in legal_actions:
        self.action_list.append(action)
        v = max(v, self.MIN_VALUE(self.game.succ(state, action), alpha, beta, depth))
        if depth == 2:
            if v in self.dic.keys():
                v+=self.action_list[0][0][0] - self.action_list[0][1][0]
            self.dic[v] = action
        self.action_list.pop()
        if v >= beta:
            return v
```

```
            alpha = max(alpha, v)
        return v

    def MIN_VALUE(self, state, alpha, beta, depth):
        if depth == 1:
            return self.Estimate_Func(state)
        v = 99999
        depth -= 1
        legal_actions = self.game.actions(state)
        random.shuffle(legal_actions)
        for action in legal_actions:
            self.action_list.append(action)
            v = min(v, self.MAX_VALUE(self.game.succ(state, action), alpha, beta, depth))
            self.action_list.pop()
            if depth == 2:
                self.dic[v] = action
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v
```

Non-recursive Version:

```
i = -1
        index = 0
        last_index = 0
        alpha = -99999999999
        player = state[0]
        for action in legal_actions:
            i += 1
            if player == 1:
                if action[1][0] > action[0][0]:
                    legal_actions.remove(action)
                    i -= 1
                    continue
            else:
                if action[1][0] < action[0][0]:
                    legal_actions.remove(action)
                    i -= 1
                    continue
            beta = 999999999

            player = state[0]
            state[1].board_status[action[1]] = state[1].board_status[action[0]]
            state[1].board_status[action[0]] = 0
            state1 = (3 - player, state[1])
            legal_actions1 = self.game.actions(state1)
            state[1].board_status[action[0]] = state[1].board_status[action[1]]
            state[1].board_status[action[1]] = 0
            random.shuffle(legal_actions1)
            if state[1].board_status[action[0]] == player + 2 and (action[1] == (5, 1) or action[1] == (5, 2)):
                legal_actions.remove(action)
                i -= 1
                continue
            if action[0] in self.list2:
                if state[1].board_status[action[0]] == player + 2:
                    legal_actions.remove(action)
                    i -= 1
                    continue
            if action[1] in self.list2:
                if state[1].board_status[action[0]] == player:
                    legal_actions.remove(action)
                    i -= 1
                    continue
            if action[0] in self.list2 and state[1].board_status[action[0]] == player:
                index = i
                break
            if action[0] not in self.list2 and action[1] in self.list2 and state[1].board_status[
                action[0]] == player + 2:
                index = i
                break
            for action1 in legal_actions1:
                player2 = state1[0]
                if player2 == 1:
                    if action1[1][0] > action1[0][0]:
                        continue
                else:
                    if action1[1][0] < action1[0][0]:
                        continue
                heuristic = Estimate_Func(state, action, player, action1, player2)
                beta = min(heuristic, beta)
                if heuristic <= alpha:
                    break
            if alpha < beta:
                last_index = index
                index = i
                alpha = beta
        final_action = legal_actions[i]
        print(legal_actions[i])
        if self.last_action is not None and final_action[1] == self.last_action[0]:
            final_action = legal_actions[last_index]
        self.last_action = final_action
        print(final_action)
        self.action = final_action
```

C. Get the Optimal Next Step

In the simple getAction function,we update next step chosen from the results selected by the minimax alpha-beta-pruning algorithm.In the first a few steps,we apply greedy search algorithm to get the marbles move faster and disturb the rival's choice.After that,we just update the action by rule.Additionally,we set a limit to the time every iteration of action takes.If it fails to reach the globally optimal solution in the limited time,a locally optimal solution will take place of it.

```python
def Estimate_Func(self, state):
    value = float(0.0)
    weight2 = 1
    player = state[0]
    board = state[1]
    player2 = player
    if player == 2:
        player2 += 1

    state[1].board_status[self.action_list[1][0]] = state[1].board_status[self.action_list[1][1]]
    state[1].board_status[self.action_list[1][1]] = 0
    state[1].board_status[self.action_list[0][0]] = state[1].board_status[self.action_list[0][1]]
    state[1].board_status[self.action_list[0][1]] = 0

    pos = state[1].getPlayerPiecePositions(player)
    pos1 = set((row, col) for (row, col) in pos if state[1].board_status[(row, col)] == player)
    pos2 = set((row, col) for (row, col) in pos if state[1].board_status[(row, col)] == player + 2)
    pos1 = list(pos1)
    pos2 = list(pos2)
    unoccupied_common_des = set(self.list[player2]).difference(pos1)
    unoccupied_special_des = set(self.list[player2 + 1]).difference(pos2)

    if state[1].board_status[self.action_list[0][0]] == player + 2:
        weight = 10
        if len(unoccupied_special_des) == 1:
            weight = 11
        if len(unoccupied_special_des) == 0:
            weight = 0
        if self.action_list[0][1] in self.list[player2 + 1]:
            if self.action_list[0][0] in self.list[player2 + 1]:
                value -= 1000000000
                weight = 0
            else:
                value += 1000000000
        if self.action_list[0][0] in self.list[player2 + 1]:
            value -= 1000000000
            weight = 0
    else:
        weight = 1
        if self.action_list[0][0] in self.list[player2 + 1]:
            if self.action_list[0][1] in self.list[player2 + 1]:
                value -= 1000000
            else:
                value += 1000000
        if len(unoccupied_special_des) == 0:
            self.count = 0

    if player == 1:
        if (self.action_list[0][1][0] - 2) > self.action_list[0][0][0]:
            value -= 5000 * weight
        if self.action_list[0][0][0] < self.action_list[0][1][0]:
            weight *= 5
        if self.action_list[0][1] in unoccupied_common_des:
            weight *= 5

        value += 1000 * (self.action_list[0][0][0] - self.action_list[0][1][0]) * weight - 200 * (
                self.action_list[1][0][0] - self.action_list[1][1][0])
        value += weight2 * 150 * (self.action_list[0][0][0] - 4)

    else:
        if (self.action_list[0][0][0] - 2) > self.action_list[0][1][0]:
            value -= 5000 * weight
        if self.action_list[0][1][0] < self.action_list[0][0][0]:
            weight *= 5
        if self.action_list[0][1] in unoccupied_common_des:
            weight *= 5
        value += 1000 * (self.action_list[0][1][0] - self.action_list[0][0][0]) * weight - 200 * (
                self.action_list[1][1][0] - self.action_list[1][0][0])
        value += weight2 * 150 * (16 - self.action_list[0][0][0])

    return value
```

III. DISCUSSION & CONCLUSION

The goal of this lab was to exemplify the minimax alpha-beta pruning algorithm and inspire us to develop our own AI Chinese Checker game based on the algorithm. By personally comprehend and utilize it, I was illuminated how to use such an adversarial search algorithm to carry out a two-player zero-sum game.

After the two-week laboratory of this project,we finally successfully built a basic conceptual framework of the minimax alpha-beta pruning algorithm.In all fairness,it was not a cinch for us to actually implement the thoughts into our code with a coarse and shallow understanding of the theory.Chances are that we came up with a new possible solution while we were stuck by how to accomplish our settled goal by means of codes,which led to us falling into a dilemma.Also,it was really a difficulty to debug the program.

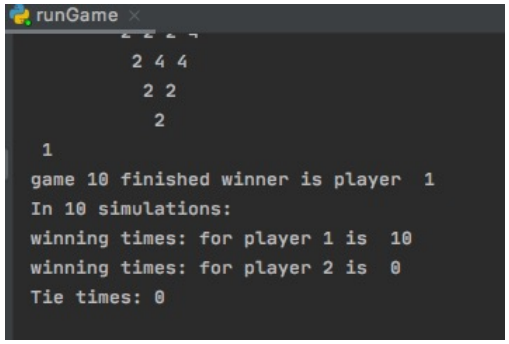
The basic obstacle we were faced with is the time limit.We were demanded to update the action as soon as possible.It sets a challenge for our algorithm to be simplified in logic,and the search tree should be restricted from unfolding wildly.The alpha-beta pruning algorithm is in application for such a problem.Besides,we added an equative sentence to examine present time of processing.If it times out,a locally optimal solution instead of a globally optimal one is selected to be implemented.

In addition,we totally implemented different search methods into our experiment.Naturally,we used the minimax alpha-beta algorithm with heuristic functions to prune the adversarial search.In the first stance that the marbles moved forward with an average speed,which squandered a large number of iteration times.To deal with the situation,we chose to adopt a so-called 'updated greedy' algorithm that owns a better performance compared to the given simple greedy algorithm.As a result,we had a more powerful control over the marbles forcing them to move in a higher speed and larger vertical leap.Furthermore,it had an additional function in adversarial game to disorganize the rival's rhythm.

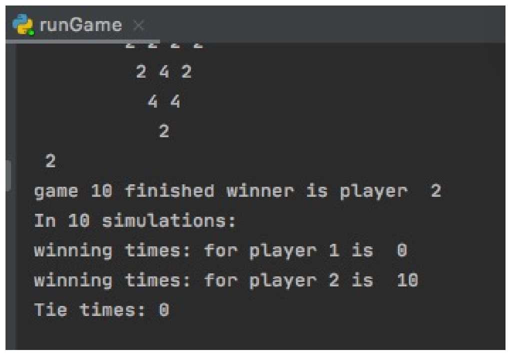
After all,we completed the task basically during the two-week laboratory,with a win rate of seventy to eighty percentage.Our best performance are illustrated in Figure 3.

However,many problems remains unsolved within limited time.First of all,we did not handle the 'stuck' situation,which we are supposed to avoid.In our code,we are striving to prevent all marbles from be stuck or blocked.It is flawed that we did not provide an independent function to manage the problem,thus put our marbles in risk.Moreover,to be frank,the heuristic function and the parameters to be designed is just passable. It is regrettable that we did not have enough competence to find the optimal values of the parameters,in which case we are easier to get into a scrape when moving forward,the marbles maybe retreating or jumping repeatedly at times.

All in all, this laboratory gave me an insight on how adversarial search algorithm like minimax alpha-beta pruning to be used in games.Through the hands-on practice,we learned to combine theory with practice.And last but not the least,it opens a new world to us of AI algorithms,which we have not heard before.We really appreciate it and derive a good deal of benefit from it.



(a) Agent as player1



(b) Agent as player2

FIG. 3: The result of RockMinimaxAgent versus SimpleGreedyAgent

...

IV. ACKNOWLEDGEMENT

We highly appreciate our professor Yue Gao,who gave us an insight into these brand new AI algorithms and provided a perfect platform for us to implement our own algorithm.Besides,we really appreciate the teaching assistants,who gave a hand in the times we were stuck.We are grateful to our classmates,who always gave us inspirations to do with the challenges faced with us.

V. APPENDIX

The following is our code of agent.py:

```python

import random, re, datetime
import math
import copy


class Agent(object):
    def __init__(self, game):
        self.game = game
        self.dic = {}
        self.list =[[],[(1, 1), (3, 1), (3, 3), (4, 1), (4, 2), (4, 3), (4, 4)],[(2, 1), (2, 2), (3, 2)],[(19, 1),
            (17, 1), (17, 3), (16, 1), (16, 2), (16, 3), (16, 4)],[(18, 1), (18, 2), (17, 2)]]
        self.last_action = None
        self.depth = 3
        self.action_list = []
        self.count=0

    def getAction(self, state):
        raise Exception("Not implemented yet")


class RandomAgent(Agent):
    def getAction(self, state):
        legal_actions = self.game.actions(state)
        self.action = random.choice(legal_actions)


class SimpleGreedyAgent(Agent):
    # a one-step-lookahead greedy agent that returns action with max vertical advance
    def getAction(self, state):
        legal_actions = self.game.actions(state)
        player = self.game.player(state)
        # self.action = random.choice(legal_actions)
        for action in legal_actions:
            if action[1][0] == 2 or (action[1][0] == 3 and action[1][1] == 2):
                if self.game.succ(state, action)[1].board_status[(action[1][0], action[1][1])] == player + 2:
                    self.action = action
        if player == 1:
            max_vertical_advance_one_step = max([action[0][0] - action[1][0] for action in legal_actions])

            max_actions = [action for action in legal_actions if
                           action[0][0] - action[1][0] == max_vertical_advance_one_step]
        else:
            max_vertical_advance_one_step = max([action[1][0] - action[0][0] for action in legal_actions])
            max_actions = [action for action in legal_actions if
                           action[1][0] - action[0][0] == max_vertical_advance_one_step]
        self.action = random.choice(max_actions)


class RockMinimaxAgent(Agent):
    def getAction(self, state):
        player = state[0]
        board = state[1]
        self.start = datetime.datetime.now()
        global count,delta
        legal_actions = self.game.actions(state)
        player_status = board.getPlayerPiecePositions(player)
        self.count += 1
        if self.count >= 100:
            self.count = 0
        if self.game.isEnd(state,100):
            self.count = 0

        if self.count <= 10:
            legal_actions = self.game.actions(state)
            player = self.game.player(state)
            board = state[1]
            flag = 0

            if player == 1:
                max_vertical_advance_one_step = -100
                max_actions = {}
                for action in legal_actions:
                    if board.board_status[action[0]] == 3:
                        if action[0] == (2, 1) or action[0] == (2, 2) or action[0] == (3, 2) or action[1] == (1, 1):
                            continue
                        else:
                            if board.board_status[action[0]] == 3 and (
                                    action[1] == (2, 1) or action[1] == (2, 2) or action[1] == (3, 2)):
                                flag = 1
                                self.action = action
                                break
                            else:
                                if (action[0][0] < 13):
                                    priority = 1
                                else:
                                    priority = 0
                                v = action[0][0] - action[1][0] + priority
                                if v >= max_vertical_advance_one_step:
                                    max_vertical_advance_one_step = v
                                    max_actions.setdefault(v, []).append(action)
                    else:
                        if action[1] == (2, 1) or action[1] == (2, 2) or action[1] == (3, 2):
```

```
                            continue
                        v = action[0][0] - action[1][0]
                        if v >= max_vertical_advance_one_step:
                            max_vertical_advance_one_step = v
                            max_actions.setdefault(v, []).append(action)
            else:
                max_vertical_advance_one_step = -100
                max_actions = {}
                for action in legal_actions:
                    if board.board_status[action[0]] == 4:  # special pegs
                        if action[0] == (18, 1) or action[0] == (18, 2) or action[0] == (17, 2) or action[1] == (19, 1):
                            continue
                        else:
                            if board.board_status[action[0]] == 4 and (
                                    action[1] == (18, 1) or action[1] == (18, 2) or action[1] == (17, 2)):
                                flag = 1
                                self.action = action
                                break
                            else:
                                if (action[0][0] > 7):
                                    priority = 1
                                else:
                                    priority = 0
                                v = action[1][0] - action[0][0] + priority
                                if v >= max_vertical_advance_one_step:
                                    max_vertical_advance_one_step = v
                                    max_actions.setdefault(v, []).append(action)
                    else:
                        if action[1] == (18, 1) or action[1] == (18, 2) or action[1] == (17, 2):
                            continue
                        v = action[1][0] - action[0][0]
                        if v >= max_vertical_advance_one_step:
                            max_vertical_advance_one_step = v
                            max_actions.setdefault(v, []).append(action)
            if flag == 0:
                self.action = random.choice(max_actions[max_vertical_advance_one_step])

        else:
            action = self.ALPHA_BETA_SEARCH(state)
            now = datetime.datetime.now()
            time = str(now - self.start)
            delta = float(time.split(':')[-1])  #
            print(delta)
            if self.last_action is not None and self.last_action[0] == action[1]:
                cnt = 1
                sorted_d = sorted(self.dic.keys(), reverse=True)
                while (self.last_action[0] == action[1]):
                    action = self.dic[sorted_d[cnt]]
                    cnt += 1
            self.last_action = action
            self.action = action


    def ALPHA_BETA_SEARCH(self, state):
        v = self.MAX_VALUE(state, -9999, 9999, self.depth)
        return self.dic[v]


    def MAX_VALUE(self, state, alpha, beta, depth):
        if depth == 1:
            return self.Estimate_Func(state)
        v = -99999
        depth -= 1
        legal_actions = self.game.actions(state)
        random.shuffle(legal_actions)
        for action in legal_actions:
            self.action_list.append(action)
            v = max(v, self.MIN_VALUE(self.game.succ(state, action), alpha, beta, depth))
            if depth == 2:
                if v in self.dic.keys():
                    v+=self.action_list[0][0][0]-self.action_list[0][1][0]
                self.dic[v] = action
            self.action_list.pop()
            if v >= beta:
                return v
            alpha = max(alpha, v)
        return v

    def MIN_VALUE(self, state, alpha, beta, depth):
        if depth == 1:
            return self.Estimate_Func(state)
        v = 99999
        depth -= 1
        legal_actions = self.game.actions(state)
        random.shuffle(legal_actions)
        for action in legal_actions:
            self.action_list.append(action)
            v = min(v, self.MAX_VALUE(self.game.succ(state, action), alpha, beta, depth))
            self.action_list.pop()
            if depth == 2:
                self.dic[v] = action
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v

    def Estimate_Func(self, state):
        value = float(0.0)
        weight2 = 1
        player = state[0]
        board = state[1]
        player2=player
```

```
        if player==2:
            player2+=1

        state[1].board_status[self.action_list[1][0]] = state[1].board_status[self.action_list[1][1]]
        state[1].board_status[self.action_list[1][1]] = 0
        state[1].board_status[self.action_list[0][0]] = state[1].board_status[self.action_list[0][1]]
        state[1].board_status[self.action_list[0][1]] = 0

        pos = state[1].getPlayerPiecePositions(player)
        pos1 = set((row, col) for (row, col) in pos if state[1].board_status[(row, col)] == player)
        pos2= set((row, col) for (row, col) in pos if state[1].board_status[(row, col)] == player+2)#special pos
        pos1=list(pos1)
        pos2=list(pos2)
        unoccupied_common_des = set(self.list[player2]).difference(pos1)
        unoccupied_special_des = set(self.list[player2+1]).difference(pos2)


        if state[1].board_status[self.action_list[0][0]] == player + 2:
            weight=10
            if len(unoccupied_special_des) == 1:
                weight=11
            if len(unoccupied_special_des) == 0:
                weight=0
            if self.action_list[0][1] in self.list[player2+1]:#
                if self.action_list[0][0] in self.list[player2+1]:
                    value -= 1000000000
                    weight=0
                else:
                    value+=1000000000
            if self.action_list[0][0] in self.list[player2+1]:
                value-=1000000000
                weight=0
        else:
            weight=1
            if self.action_list[0][0] in self.list[player2+1]:
                if self.action_list[0][1] in self.list[player2+1]:#
                    value -= 1000000
                else:
                    value+=1000000
            if len(unoccupied_special_des)==0:
                self.count=0


        if player==1:
            if (self.action_list[0][1][0] - 2) > self.action_list[0][0][0]:
                value -= 5000 * weight
            if self.action_list[0][0][0] < self.action_list[0][1][0]:
                weight *= 5
            if self.action_list[0][1] in unoccupied_common_des:
                weight *= 5

            value += 1000 * (self.action_list[0][0][0] - self.action_list[0][1][0]) * weight + 200 * (
                    self.action_list[1][0][0] - self.action_list[1][1][0])
            value += weight2 * 150 * (self.action_list[0][0][0] - 4)

        else:
            if (self.action_list[0][0][0] - 2) > self.action_list[0][1][0]:
                value -= 5000 * weight
            if self.action_list[0][1][0] < self.action_list[0][0][0]:
                weight *= 5
            if self.action_list[0][1] in unoccupied_common_des:
                weight *= 5
            value += 1000 * (self.action_list[0][1][0] - self.action_list[0][0][0]) * weight + 200 * (
                    self.action_list[1][1][0] - self.action_list[1][0][0])
            value += weight2 * 150 * (16-self.action_list[0][0][0] )

        return value



### END CODE HERE ###
```