AU 332 Artificial Intelligence: Principles and Techniques

By: Wenjie Wu (518021910025) & Fan Ding(518021910010)

HW#: 1

September 28, 2020

I. INTRODUCTION

A. Purpose

The goal of homework 1 is to design an agent to play Chinese Checker, which will exemplify the **Minimax Algorithm**, **alpha-beta pruning** and the use of **heuristic functions** to prune the adversarial search.

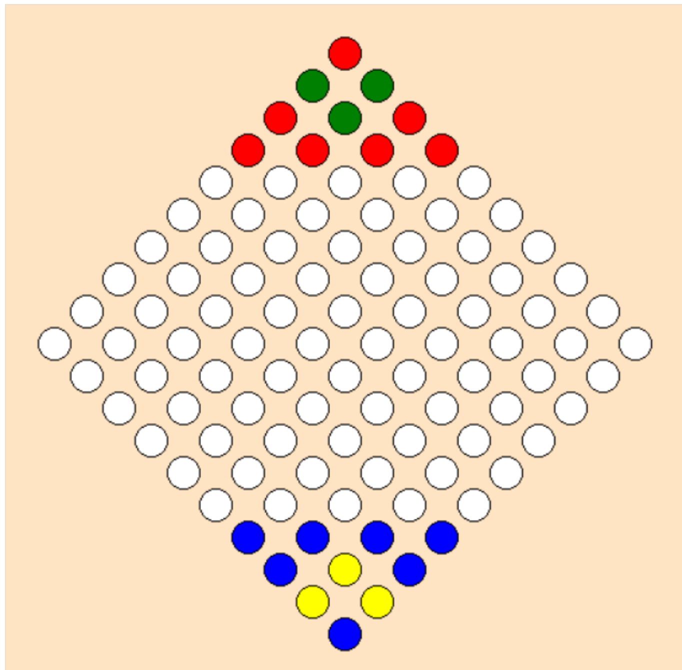

FIG. 1: our checker board

Unlike the traditional Chinese Checker game, each player has two types of marbles with different colors in our game. The Checker board is shown in Figure 1. The purpose of our agent is not only to move all the marbles into opponents' home, namely the triangle area at the top(bottom), but also to ensure that each type of marbles is in the right position respectively.

In order to beat our opponent, our agent must achieve the goal above faster than its opponent. Also, there are some extra constraints listed below:

- One second is limited for each action.
- In order to prevent the situation that one of the players may not leave its triangle and prevent the opponent to occupy its space, after 100 turns, if any of the marbles of one player are still in its own triangle, the player loses its game immediately.

B. Equipment

1. Ubuntu 18.04
2. Python 3.8.3

C. Procedure

1. Get familiar with the utility functions and variables in the code.
2. Implement the **Minimax Algorithm** and **alpha-beta pruning**.
3. Work out some potential evaluating methods.
4. Design the evaluating function and test it.
5. Refine the evaluating function and **Minimax Algorithm**.
6. Complete our FlappyAgent and write the report.

II. IMPLEMENTATION

A. Evaluating function

In order to make full use of the **Minimax Algorithm**, we must design a robust evaluating function. Apparently, the vertical distance from home(V_v) is the most important factor because it directly determines the game result; hence we should attach a big coefficient to it. Notably, we also have to put each type of marbles into right position, thus designing a reward function(V_r) listed in Equation (1).

$$V_r = \begin{cases} V_r + 5 & \text{for marble in right position} \\ V_r - 5 & \text{for marble in wrong position} \end{cases} \tag{1}$$

Since stepping into a wrong position will reduce the reward, the marbles can ultimately find the right one. Besides, we also consider other factors like horizontal distance(V_h) when playing Chinese Checker. We define it intuitively as the sum of horizontal distance from the middle column because the smaller horizontal distance are, the more chance marbles can step over each other. However, when we are in the middle, we also provide chance for our opponent. Therefore, the weight of horizontal distance should not be too large. The final factor we take into accont is the row variance(V_{var}), which is defined in Equation (2).

$$V_{var} = \sum_i |r_i - \overline{r}| \tag{2}$$

where r_i means row coordinate of marble i and $\overline{r}$ means the average row coordinate.

This factor can ensure that all marbles move forward in a whole in case that one marble has to move step by step in the end. Consequently, we get the value of our state shown in Equation (3). Do the same for the opponent except for the reward function and then subtract them to derive our evaluating function.

$$V = w_v V_v + w_h V_h + w_{var} V_{var} + V_r \tag{3}$$

B. Minimax Algorithm and alpha-beta pruning

Minimax alpha-beta pruning evaluates the local subtree rather than the entire subtrees of a node to decide its value. It uses two dynamically computed bounds alpha and beta to bound the values that nodes can take. Alpha is the minimum value that the max player is guaranteed (regardless of what the min player does) through another path throughout the game tree. This value is used to implement pruning at the minimizing levels. When the min player has discovered that the score of a min node would necessarily be less than alpha, it need not evaluate any more choices from that node because the max player already has a better move (the one which has value alpha).

Beta is the maximum value that the min player is guaranteed and is used to implement pruning at the maximizing levels. When the max player has discovered that the score of a max node would necessarily be greater than beta, it can stop evaluating any more choices from that node because the min player would not allow it to take this path since the min player already has a path that guarantees a value of beta. The pseudocode of this algorithm is shown in Figure 2.

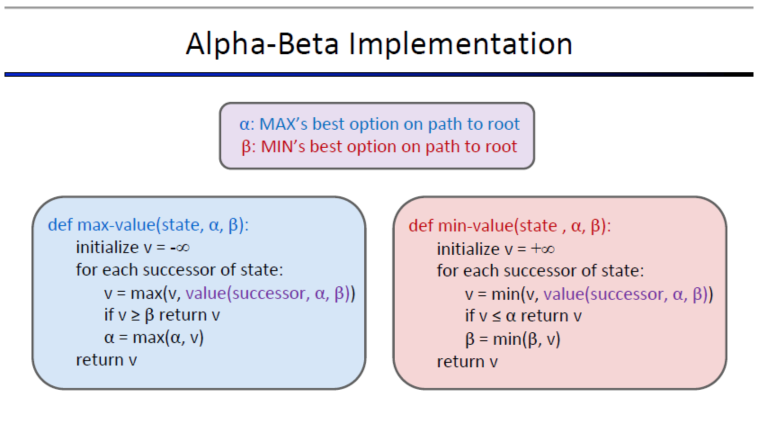

FIG. 2: pseudocode

C. Get action

Initially, we decide to employ **Monte Carlo Tree Search**, which proves useful in similar games like Go. Nevertheless, due to the computation time limit, we cannot use it directly. Instead, we try to apply one of its ideas to our strategy that it only searches nodes that have potential to win. Therefore, we first sort all the valid actions by their step length and only choose the toppest 20 actions to compute, which boosts our algorithm. Besides, we get rid of those actions leading to a big step backward.

To make full use of the one second, we can gradually increase the searching depth of **Minimax Algorithm** from 2. As long as we get a better action with **Minimax Algorithm**, we renew the chosen action immediately. Sometimes, we may get stuck in some situations. To prevent that, we store the last action we made and disregard the action leading to our last state.

III. EXPERIMENT

As shown in Figure 3, our FlappyAgent beats SimpleGreedyAgent by 10:0.

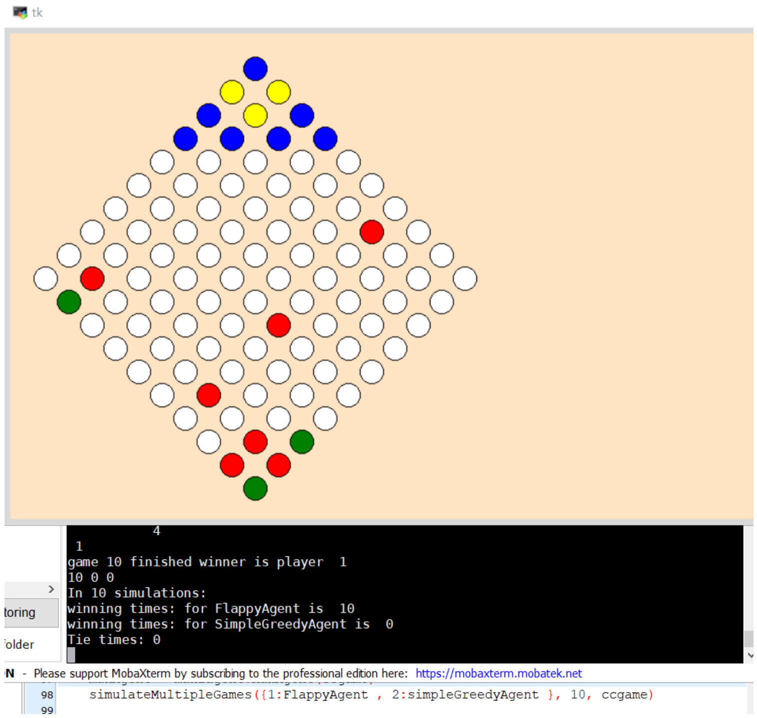

FIG. 3: game with SimpleGreedyAgent

IV. DISCUSSION & CONCLUSION

Though we can beat the SimpleGreedyAgent by 10:0 at ease, there is still much improvement we can do. For example, we think we can divide the game into two stages. In the first stage, we can step over both our marbles and opponents' and the same for the opponent. Therefore, the **Minimax Algorithm** can be useful. However, in the second stage, marbles of two players leave apart from each other, so it is unnecessary for us to consider opponents' marbles. Since we only need to focus on ourselves, the computation cost is diminished a lot. Actually, we try to implement this idea in our code(see line 253 and below in V). Unfortunately, we failed to achieve a stable performance for lack of time.

Finally, thanks to Prof Gao, TA and those open-source programs on Github. We hope to compete with other students and acquire some inspirations from them.

V. APPENDIX

agent.py:

```python
import random, re, datetime, copy
from queue import PriorityQueue

class Agent(object):
    def __init__(self, game):
        self.game = game

    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)  # actions:[pos,new_pos]

        self.action = random.choice(legal_actions)

        player = self.game.player(state)  # state:[player,board]
        if player == 1:   # choose action with the biggest step
            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 FlappyAgent(Agent):
    def __init__(self, game):
        super().__init__(game)
        self.lastaction = None  # my last action
        self.exp_depth = 2   # current exploration depth, start from 2
        self.end = 0   # not used
        self.breadth = 20      # the maximal breadth one layer extends
    def getAction(self, state):
        legal_actions = self.game.actions(state)
        self.action = random.choice(legal_actions)

        player = self.game.player(state)

        depth = 2   # initial exploration depth
        self.exp_depth = depth

        value = -1e5   # namely negative infinity

        pq = PriorityQueue()  # sort legal actions
        for action in legal_actions:
            if player==1:
                if action[0][0]>3:      # constrain the most rows one piece can step backward in order to prune
                    if action[0][0] - action[1][0] < -1:
                        continue
                else:
                    if action[0][0] - action[1][0] < -2:
                        continue
            else:
                if action[0][0] <17:
                    if action[0][0] - action[1][0] > 1:
                        continue
                else:
                    if action[0][0] - action[1][0] > 2:
                        continue
            if self.lastaction != None and action[0]==self.lastaction[1] and action[1] == self.lastaction[0]: # prevent being stuck
                continue
            pq.put((-(2*player-3)*(action[1][0]-action[0][0]),action)) # sort by step length
        next_pq=PriorityQueue()
        while True:
            cnt = 0
            while self.breadth > cnt and not pq.empty():
                action = pq.get()[1]
                if self.getValue(self.game.succ(state, action),player) == 1e4:   # I win
                    self.action = action
                    break
                cnt += 1
                next_value=self.minimax(value,1e5,self.game.succ(state,action),depth,player)  # player is always me
                next_pq.put((-next_value,action))
                if next_value > value:
                    value = next_value
                    self.action=action

            self.lastaction = self.action
            depth+=1  # if there is time left, increase the exploration depth
            self.exp_depth=depth
            del pq
            pq = PriorityQueue()  # renew the priority queue
            while not next_pq.empty():
                pq.put(next_pq.get())
```

```python


    def getValue(self, state, player):
        board = state[1]
        my_pos = board.getPlayerPiecePositions(player)
        enemy_pos = board.getPlayerPiecePositions(3-player)
        my_dist_from_home = 0  # the average of vertical distance of all my pieces from home
        enemy_dist_from_home = 0
        my_col_dist = 0  # the sum of horizontal distance of some pieces from the middle column
        enemy_col_dist = 0
        var = 0 # the variance of my rows
        var2=0
        value=0  # reward for arrived pieces
        value2=0

        my_arr_num=0  # the number of arrived pieces
        #enemy_arr_num=0
        if player==1:
            special_loc1=[(2,1),(2,2),(3,2)]    # terminal for three special pieces
            # special_loc2=[(18,1),(18,2),(17,2)]
            for pos in my_pos:
                my_dist_from_home += (20 - pos[0])/10.0  # average distance

                if pos[0]<=4:
                    if pos in special_loc1:    # get reward if arriving at right terminal, lose reward if arriving at wrong terminal
                        if board.board_status[pos] == 3:
                            value+=5
                            my_arr_num+=1
                        else:
                            value-=5
                    else:
                        if board.board_status[pos] == 1:
                            value+=5
                            my_arr_num+=1
                        else:
                            value-=5
                            if pos[0]==1:
                                value-=50
                else:
                    if pos[0]&1:
                        if pos[1]==(board.getColNum(pos[0])+1)/2:  # the exact middle column may lead to "stuck" situation
                            my_col_dist+=1
                        else:
                            my_col_dist += abs(pos[1]-(board.getColNum(pos[0])+1)/2)-1
                    else:
                        my_col_dist+=abs(pos[1]-(board.getColNum(pos[0])+1)/2)

            for pos in enemy_pos:    # the same for enemy except for the reward
                enemy_dist_from_home+=pos[0]/10.0
                if pos[0]<16:
                    if pos[0]&1:
                        if pos[1]==(board.getColNum(pos[0])+1)/2:
                            enemy_col_dist+=1
                        else:
                            enemy_col_dist += abs(pos[1]-(board.getColNum(pos[0])+1)/2)-1
                    else:
                        enemy_col_dist+=abs(pos[1]-(board.getColNum(pos[0])+1)/2)
            for pos in my_pos:
                var += abs(20-pos[0] - my_dist_from_home)   # wrong
            for pos in enemy_pos:
                var2 += abs(pos[0] - enemy_dist_from_home)
        else:
            #special_loc2=[(2,1),(2,2),(3,2)]
            special_loc1=[(18,1),(18,2),(17,2)]
            for pos in my_pos:
                my_dist_from_home += pos[0]  /10.0

                if pos[0]>=16:
                    # value+=10
                    if pos in special_loc1:
                        if board.board_status[pos] == 4:
                            value+=5
                            my_arr_num+=1
                        else:
                            value-=5
                    else:
                        if board.board_status[pos] == 2:
                            value+=5
                            my_arr_num+=1
                        else:
                            value-=5
                            if pos[0]==19:
                                value-=50
                else:
                    if pos[0]&1:
                        if pos[1]==(board.getColNum(pos[0])+1)/2:
                            my_col_dist+=1
                        else:
                            my_col_dist += abs(pos[1]-(board.getColNum(pos[0])+1)/2)-1
                    else:
                        my_col_dist+=abs(pos[1]-(board.getColNum(pos[0])+1)/2)
            for pos in enemy_pos:
                enemy_dist_from_home+=20-pos[0]
                if pos[0]>4:
                    if pos[0]&1:
                        if pos[1]==(board.getColNum(pos[0])+1)/2:
                            enemy_col_dist+=1
                        else:
                            enemy_col_dist += abs(pos[1]-(board.getColNum(pos[0])+1)/2)-1
                    else:
                        enemy_col_dist+=abs(pos[1]-(board.getColNum(pos[0])+1)/2)
```

```python
            for pos in my_pos:
                var += abs(pos[0] - my_dist_from_home)
            for pos in enemy_pos:
                var2 += abs(20-pos[0] - enemy_dist_from_home)
        if my_arr_num==10:  # I win
            return 1e4
        #if enemy_arr_num==10:  # I lose
            #return -1e4
        score = 24*(my_dist_from_home-enemy_dist_from_home) - 0.2 * (var-var2)-0.6*(my_col_dist-enemy_col_dist)+value #-value2
        return score


    def minimax(self, alpha, beta, state, depth, player):
        if depth==0:
            return self.getValue(state, player)
        depth-= 1
        if state[0]!=player:  # it's enemy's turn
            if depth != self.exp_depth-1:
                if self.getValue(state,player)==1e4:  # I win
                    return 1e4

            actions = self.game.actions(state)
            pq = PriorityQueue()
            for action in actions:  # enemy's actions
                pq.put(((2 * player-3) * (action[1][0] - action[0][0]), action))
            cnt =0
            value=1e5
            while self.breadth>cnt and not pq.empty():
                action=pq.get()[1]
                cnt+=1
                value=min(value,self.minimax(alpha,beta,self.game.succ(state,action),depth,player))  # min node
                if value<=alpha:   # prune
                    return value
                beta=min(beta,value)
            return value
        else: # it's my turn
            #if self.getValue(state,player) ==-1e4:
                #return -1e4
            actions=self.game.actions(state)
            value=-1e5
            pq=PriorityQueue()
            for action in actions:  # my actions
                pq.put((-(2 * player-3) * (action[1][0] - action[0][0]), action))
            cnt=0
            while self.breadth>cnt and not pq.empty():
                action=pq.get()[1]
                cnt+=1
                value=max(value,self.minimax(alpha,beta,self.game.succ(state,action),depth,player))  # max node
                if value>=beta:  # prune
                    return value
                alpha=max(alpha,value)
            return value

    ''' For lack of time, the functions below are not used but they may have certain potential to improve our agent. '''
    def GetFirstLastPiece(self, all_pos, player):
        first = all_pos[0][0]
        last = all_pos[0][0]
        if player==1:
            for pos in all_pos:
                first = min(first,pos[0])
                last = max(last, pos[0])
        else:
            for pos in all_pos:
                first = max(first,pos[0])
                last = min(last,pos[0])
        return first,last

    def near_end(self, my_last, enemy_last, player):
        return enemy_last > my_last if player == 1 else my_last > enemy_last

    def midtime(self, my_first, enemy_first, player):
        return my_first - enemy_first <= 2 if player == 1 else enemy_first - my_first <= 2

    def maximax(self, state, depth, player):
        if depth==0:
            return self.getValue2(state, player)
        if self.getValue2(state,player)==1e4:
            return 1e4 # *depth
        depth-= 1

        actions=self.game.actions(state)
        value=-1e5
        pq=PriorityQueue()
        for action in actions:  # my actions
            pq.put((-(2 * player-3) * (action[1][0] - action[0][0]), action))
        cnt=0
        while self.breadth>cnt and not pq.empty():
            action=pq.get()[1]
            cnt+=1
            value=max(value,self.maximax(self.pseudo_succ(state,action),depth,player))
        return value


    def getValue2(self, state, player):  # suppose depth is even and this is always my turn
        board = state[1]
        my_pos = board.getPlayerPiecePositions(player)
        my_dist_from_home = 0  # the average of vertical distance of all my pieces from home
        my_col_dist = 0  # the sum of horizontal distance of some pieces from the middle column
        var = 0 # the variance of my rows
        value=0  # reward for arrived pieces
        my_arr_num=0  # the number of arrived pieces
```

```
        if player==1:
            special_loc1=[(2,1),(2,2),(3,2)]    # terminal for three special pieces
            for pos in my_pos:
                my_dist_from_home += (20 - pos[0])/10.0  # average distance

                if pos[0]<=4:
                    if pos in special_loc1:    # get reward if arriving at right terminal, lose reward if arriving at wrong terminal
                        if board.board_status[pos] == 3:
                            value+=5
                            my_arr_num+=1
                        else:
                            value-=10
                    else:
                        if board.board_status[pos] == 1:
                            value+=5
                            my_arr_num+=1
                        else:
                            value-=5
                            if pos[0]==1:
                                value-=5
                else:
                    if pos[0]&1:
                        if pos[1]==(board.getColNum(pos[0])+1)/2:  # the exact middle column may lead to "stuck" situation
                            my_col_dist+=1
                        else:
                            my_col_dist += abs(pos[1]-(board.getColNum(pos[0])+1)/2)-1
                    else:
                        my_col_dist+=abs(pos[1]-(board.getColNum(pos[0])+1)/2)

            for pos in my_pos:
                var += abs(20-pos[0] - my_dist_from_home)

        else:
            special_loc1=[(18,1),(18,2),(17,2)]
            for pos in my_pos:
                my_dist_from_home += pos[0]  /10.0

                if pos[0]>=16:
                    if pos in special_loc1:
                        if board.board_status[pos] == 4:
                            value+=5
                            my_arr_num+=1
                        else:
                            value-=10
                    else:
                        if board.board_status[pos] == 2:
                            value+=5
                            my_arr_num+=1
                        else:
                            value-=5
                            if pos[0]==19:
                                value-=5
                else:
                    if pos[0]&1:
                        if pos[1]==(board.getColNum(pos[0])+1)/2:
                            my_col_dist+=1
                        else:
                            my_col_dist += abs(pos[1]-(board.getColNum(pos[0])+1)/2)-1
                    else:
                        my_col_dist+=abs(pos[1]-(board.getColNum(pos[0])+1)/2)

            for pos in my_pos:
                var += abs(pos[0] - my_dist_from_home)
        if my_arr_num==10:  # I win
            return 1e4

        score = 24*my_dist_from_home - 0.2 * var-0.6*my_col_dist+value
        return score

    def pseudo_succ(self, state, action): # always my turn, regardless of enemy's action
        board = copy.deepcopy(state[1])
        board.board_status[action[1]] = board.board_status[action[0]]
        board.board_status[action[0]] = 0
        return (state[0], board)
```