AU 332 Artificial Intelligence: Principles and Techniques

By: Meng Zhou, Yangqian Wu

Instructor: Yue Gao

October 8, 2019

I. INTRODUCTION

In this homework, we are going to implement an intelligent agent in Chinese checkers game. Chinese checkers is a perfect information game for 2 players. The goal of the game is to get 10 pieces from one's starting position to one's ending position as quickly as possible. To win the game, the agent needs to try to make its pieces jump further and prevent the opppsite agent's pieces from jumpping through the board. The alogorithms our agent uses are minimax search, alpha-beta pruning and some other methods.

II. METHODOLOGY

1. Game start

At the very beginning of the game, the pieces of both sides are independent of each other. Therefore, a minimax strategy is not necessary. But actually, a greedy agent which we have tried at the beginning doesn't make much difference of the overall performance.

2. Minimax Alpha-beta Pruning

Minimax search is the main policy of our agent. During the game, every step our agent takes is modeled as a max layer and every step the opponent takes is modeled as a min layer. This algorithm assumes that our agent will choose the action that will lead to a highest score while our opponent will try to minimize our score. And the agent will take action following this policy.

The linear growth of the search depth will result in exponential growth of the search space. Considering the time limit of 1 second, we choose *a search depth of 2*.

Although we only choose a shallow search depth, we still find that the decision of our agent will take a relatively long time. So we need alpha-beta pruning to make our search space smaller.

Alpha-beta pruning is an effective way to prune unnecessary leaves that can be inferred not to be the result. After applying this, the average decision time of our agent is lowered from 0.9s to 0.6s.

In addition, according to the textbook, we learn that the order of minimax search is essential for better effect. In order to prune more leaves, we can put those which are the most possible to be the optimal. And we designed two ways to achieve this.

1. sort the list of legal actions in a way that put those actions that make the biggest vertical advances in the front of the list.

2. calculate the evaluation score of the state after taking this action and put these action into a Priority Queue whose priority is the evaluation score.

As we imagine the second scheme is more computationally expensive but receives a slightly better result by pruning more leaves of the search tree.

3. Evaluation Function

It is self-evident that the evaluation function is cruicial for the performance of our agent. We have tried a lot of different standards to evaluate, and we finally adopted:

1. the squared sum of the distance of all the pieces to the destination corner ((1,1) or (19,1))

2. the squared sum of the distance of all the pieces to the midline

3. the vertical advance that the opponent pieces make

- The distance from all pieces to destination corner(L2_eval)

  This evaluation function calculates the distance from player's pieces to its destination corner(for player1 it's (1,1), for player2 it's (19,1)), and then add them up. We will give this evaluation a negative weight because the score it recieves will be higher if this distance is smaller.

  In the experiment, we have compared this evaluation to the distance from all pieces to the 10 completed positions. But finally found that this evaluation outperformed the other one. The reaon for this may be by using this evaluation, we encourage those trailing pieces to move first so that all of our pieces will move more cohesively and therefore make more hopping.

```
def L2_eval(self, my_pos, player):
    L2_dis = 0  #a new approach to count L2_dis
    if player == 2:
        for pos in my_pos:
            col_num = self.game.board.getColNum(pos[0])
            mid_col = (col_num + 1) / 2
            L2_dis += (pos[0]-19)**2 + (pos[1]-mid_col)**2
    if player ==1:
        for pos in my_pos:
            col_num = self.game.board.getColNum(pos[0])
            mid_col = (col_num + 1) / 2
            L2_dis += (pos[0]-1)**2 + (pos[1]-mid_col)**2
    return L2_dis
```

(a) The code for L2_eval function

- The squared sum of the distance of all the pieces to the midline(hor_eval)

  This evaluation calculates the distance between pieces and the midline of the game board. We use 0 to minus all the distance we calculate. Finally, we get a negative number wihch represents the "deviation" of all pieces.

  In the game, we want every piece to move as close to the midline as possible, for we will take less steps in moving the pieces near the midline.

```
def hor_eval(self, my_pos, player):
    partial_eval = 0
    for chess in my_pos:
        col_num = self.game.board.getColNum(chess[0])
        mid_col = (col_num + 1) / 2
        partial_eval -= abs(chess[1] - mid_col) ** 2
    return partial_eval
```

(b) The code for hor_eval function

- The vertical advance that the opponent pieces make(opp_eval)

  In the sum-zero game, we want to maximize our gain while minimize opponent's gain.

  So, in the game, we calculate the vertical distance from opponent's pieces' positions to the destination. This evaluation will be higher if opponent's pieces are further from their destination.

- Some other heuristics During testing, we have also tried some other possible heuristcs, which is not finally adopted for various reasons. The sum of maximum vertical advance for all pieces of player i takes too much computation and will lead to timeout. The horizontal or veritical variance of pieces of each player turn out to be not that useful.

```
def opp_eval(self, opp_pos, player):
    partial_eval = 0
    if player == 2:
        for chess in opp_pos:
            partial_eval += chess[0]
    else:
        for chess in opp_pos:
            partial_eval += 20 - chess[0]
    return partial_eval
```

(c) The code for opp_eval function

Consequently, the evaluation value can be computed as follow:

$$eval = a * hor\_eval - b * L2\_eval + c * opp\_eval \tag{1}$$

```
def evaluation(self, state):
    evaluate = 0
    player = state[0]
    board = state[1]
    my_pos = board.getPlayerPiecePositions(state[0])
    opp_pos = board.getPlayerPiecePositions(3 - state[0])

    p1 = 20  #weights for each eval
    p2 = 1
    p3 = -2
    p4 = 10
    if self.count_completeness(state) > 5:
        if player == 2:
            complete_state = [(19,1), (18,1), (18,2), (17,1), (17,2), (17,3), (16,1), (16,2), (16,3), (16,4)]
        else:
            complete_state = [(1,1), (2,1), (2,2), (3,1), (3,2), (3,3), (4,1), (4,2), (4,3), (4,4)]
        complete_dis = 0
        dif_set1 = list(set(complete_state) - set(my_pos))
        dif_set2 = list(set(my_pos) - set(complete_state))
        for pos in dif_set2:
            complete_dis += min([ (pos[0] - targ_pos[0])**2 + ((pos[1] - (self.game.board.getColNum(pos[0])+1)/2)+\
            ((self.game.board.getColNum(targ_pos[0])+1)/2-targ_pos[1]))**2  for targ_pos in dif_set1])
        evaluate = p2 * self.hor_eval(my_pos, player) + p3 * complete_dis + p4 * self.opp_eval(opp_pos, player)
    else:
        evaluate = p2 * self.hor_eval(my_pos, player) + p3 * self.L2_eval(my_pos, player) + p4 * self.opp_eval(opp_pos, player)
    return evaluate
```

(d) The code for whole eval function

The weight of each parameters can be tuned to fit the game best.

When we are testing our agent, we accidentally found that the heuristic evaluation function should also vary during the different period of the game. Theorically, there are 3 periods we should take into consideration. "start","mid","end" indicate different situation of the game and in each period, the agent should have different evaluation function, and then it can perform corresponding actions.

After a long time of testing, we eventually change the evaluation in the end of game. We use completed_dis instead of L2_eval when calculating the distance from pieces' position to completed position, which performs better in the competition and specifies the detail in completed positions.

```
def count_completeness(self, state):
    my_pos = state[1].getPlayerPiecePositions(state[0])
    if state[0] == 2:
        complete_state = [(19,1), (18,1), (18,2), (17,1), (17,2), (17,3), (16,1), (16,2), (16,3), (16,4)]
    else:
        complete_state = [(1,1), (2,1), (2,2), (3,1), (3,2), (3,3), (4,1), (4,2), (4,3), (4,4)]

    count = len(complete_state) - len(list(set(complete_state) - set(my_pos)))

    return count
```

(e) The code for counting current completeness

4. Some points in our strategy

Our main strategy is Alpha-beta pruning and the combination of three evaluations in adversarial search. Besides, we divide the game into different periods based on counting the completeness of current state. In different period, we use different evaluation function to make the desicion. We also want to use different strategy like greedy strategy in depth of 2 in the start and end of the game, but when we practice it, we find it have made no improvement. Moreover, we tried to use reinforcement learning in our parameters tuning, but find it hard to achieve for the following reasons:

1. It's hard to judge whether one set of parameter is better than the other. The fact that agent A beats agent B doesn't prove that agent A will beat more unknown agent than agent B.

2. We always change our model and evaluation function when coming up with a new possibly better idea, so it makes it hard to train the weight.

Finally we choose our parameters by intuition(e.g.L2 distance to the destination is evidently more important than horizontal compactness) and experiment with different weights to see how it works.

Besides, we have made some main points in our code, we will talk about it below:

Acoording to the rule of game. We have to return the action within 1 second, which is too hard for us to accomplish. In MINIMAX strategy with the depth of 2, it's ususl to spend more than 1 second on searching the best action. Therefore, in the process of the searching, we update the best action every time when we reach a temporary optimal action. That makes sure we can get a better action when we return in 1 second.

In order to lessen the time each search use, we introduce the priority queue to make a order of each legal action. In this way, the Alpha-beta pruning will prune more, and thus save more time.

In deciding the order of each legal action, we have tried a lot of different reference. Consider the vertical moving distance, or combining the horizontal distance, or even straightly the evaluation if we take this action. We eventually take the last one.

We also write a function to determine the greedy action which achieves the biggest vertical move distance in 2 moves. But when we apply it to our agent, we find it's not as effective as we expected. So we finally discard this strategy.

III. IMPROVEMENT AND THOUGHTS

Although we have accoplished our agent, we still have some ideas remained wihch might make some improvement.

1. We could use Genetic Algorithm in the process of weight tuning.

2. We still wonder whether Greedy Algorithm used in the end of the game could improve agent's performance. But that's actually really challenging to judge whether two agents are independent or not. We believe if we have more time and are able to search more deeply, we can keep a consistent strategy in the whole game and get a reasonably well result. Due to the limitation of time, we have to change our evaluation function at the end of the game.

During the time of completing our agent, we update our agent more than 5 versions. To examine the latter's performance, we let it compete with the former one and simplegreedyagent. Finally, we get the the agent we submit. In the end, thanks for the help we received during the testing time. We totally understand a lot from this procedure. Minimax search and alpha-beta pruning are really ingenious algorithms!