AU 332 Artificial Intelligence: Principles and Techniques

By: Zhengbao He(517030910157) & Jingwei Zhao(5517030910047)

HW#: 2

October 8, 2019

1 Introduction

1.1 Purpose

In this assignment, we are going to apply minimax search with alpha-beta pruning on Chinese Checkers, in the aim of building an intelligent Chinese Checkers play agent and winning the tournament.

Through this assignment, we will design and implement a Minimax algorithm and try to optimize it using alpha-beta pruning. Besides, we will devise a decent evaluation function as searching heuristics because we can not search the whole adversarial tree with limited computing resources.

Finishing this assignment, we are more conversant with the concept of adversarial search, as well as basic algorithms such as MiniMax and Alpha-Beta Pruning, and how to implement them.

1.2 Equipment

- A laptop with Windows 10 or Linux
- Anaconda 3
- VS Code
- TexLive

1.3 Procedure

1. Implement Minimax and Alpha-Beta Pruning algorithms in python file *agent.py*.
2. Devise an evaluation function, which returns a heristic value indicating which step to take at a cetain searching depth. The evaluation function is necessary because we can not search the whole adversarial tree due to limited computing resources.
3. Test the agent, refine the algorithms, and enhance the agent's performance.

2 Code and algorithm

This section will introduce the code and algorithm we design and some attempts we made.

2.1 The basic Minimax algorithm with alpha-beta pruning

The code below implements a simple MiniMax algorithm with alpha-beta pruning and the default depth is 2, which means that we only consider two movement of our pieces. This is because when the pieces of both players are entangled, considering too many moves makes the algorithm very complicated. What we are thinking about is how to reach the end point faster than the opponent, and the opponent is also. Because both players will give priority to their own movements, too deep search will introduce more uncertainty and consume more computing resources. So when the two players' pieces are not separated, we just use the MiniMax algorithm whose depth is 2.

```
def MiniMax_pruned_version(self, state, depth, al, be, Depth=2):
  if depth != Depth:
    alpha = al
    beta = be
    if depth % 2 == 0: # max layer
      evaluation = -10000
    else:
      evaluation = 10000
    selected_action = None
    legal_actions = self.game.actions(state)
    player = state[0]
    for action in self.stimulation_max(state, legal_actions):
      if action[1][0]*(-1)**player < action[0][0]*(-1)**player:
        continue
      board = state[1]
      board.board_status[action[0]] = 0
      board.board_status[action[1]] = player
      value, next_action = self.MiniMax_pruned_version((3 - player, board), depth + 1, al=alpha, be=beta)
      board.board_status[action[0]] = player
      board.board_status[action[1]] = 0
      if depth % 2 == 0: # max layer
        if value > evaluation:
          evaluation = value
          selected_action = action
          alpha = value
      else:  # min layer
        if value < evaluation:
          evaluation = value
          selected_action = action
          beta = value
      if alpha >= beta:
        #print('\t'*depth + "Pruned in layer",depth)
        return evaluation, selected_action
    return evaluation, selected_action
  else:
    evaluation_value = self.evaluation(state, p=2, i=10, d=5)
    return evaluation_value, None
```

2.2 SelfMax algorithm

When two players' pieces are separated, we apply another algorithm which we refer to as *SelfMax* algorithm. It just considers the movement of the agent itself, without considering that of the opponent. So in this case, since pieces of two players have already been saparated with each side, movement of each side will not affect the other. Thus considering opponent's movement only consume computing resources. Bacause we don't need to consider the opponent's movement, the two factors limiting depth in MiniMax above disappear, we can deepen the search depth and get better performence. The detailed codes of *SelfMax* are listed below(the variable *Depth* is 2 means that we consider 3 movements of our agent).

```python
def selfMax(self, state, depth, Depth=2):
  if depth != Depth:
    evaluation = -10000
    selected_action = None
    legal_actions = self.game.actions(state)
    player = state[0]
    for action in self.stimulation_max(state, legal_actions):
      if action[1][0] * (-1) ** player < action[0][0] * (-1) ** player:
        continue
      board = state[1]
      board.board_status[action[0]] = 0
      board.board_status[action[1]] = player
      ver_positions = [position[0] for position in board.getPlayerPiecePositions(player)]
      if sum(ver_positions) == (2 - player) * 30 + (player - 1) * 170:
        board.board_status[action[0]] = player
        board.board_status[action[1]] = 0
        return 10000, action
      value, next_action = self.selfMax((player, board), depth + 1)
      board.board_status[action[0]] = player
      board.board_status[action[1]] = 0
      if value > evaluation:
        evaluation = value
        selected_action = action
    return evaluation, selected_action
  else:
    evaluation = self.evaluation(state, p=1, i=5, d=0)
    return evaluation, None
```

2.3 Evaluation Function

When the search depth is limited because of limited computing resources, it is very important to evaluate the current game when the outcome is not divided. In the evaluation function below, we use four factors to evaluate the move decision:

- Total vertical distancement of our pieces
- The distance of the piece from the center line of the board
- The lagger piece–the piece at the end of the board
- Total vertical distancement of opponent's pieces

We use three parameters p, i, and d to weight the four different parts, where the total vertical distancement of our pieces and that of the opponent's pieces share the same parameter (p).

```python
def evaluation(self, state, p, i, d):
  player = state[0]
  board = state[1]

  if player == 1:
    ver_positions = [position[0] for position in board.getPlayerPiecePositions(player)]
    op_p = [position[0] for position in board.getPlayerPiecePositions(3 - player)]
    hor_positions = [abs(position[1] - board.getColNum(position[0]) / 2) / board.getColNum(position[0])
         for
             position in board.getPlayerPiecePositions(player) if position[0] % 2 == 0]

    ver_displacement = sum(ver_positions)
    lagger = 2 * board.size - 1 - max(ver_positions)
    hor_displacement = sum(hor_positions)
```

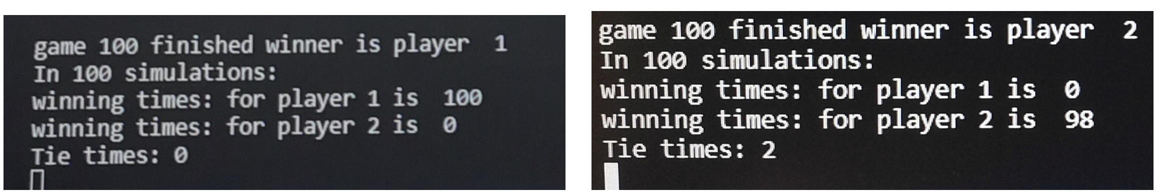
```
    else:
      ver_positions = [2 * board.size - 1 - position[0] for position in board.getPlayerPiecePositions(player)]
      op_p = [2 * board.size - 1 - position[0] for position in board.getPlayerPiecePositions(3 - player)]
      hor_positions = [abs(position[1] - board.getColNum(position[0]) / 2) / board.getColNum(position[0]) for
              position in board.getPlayerPiecePositions(player) if position[0] % 2 == 0]
      ver_displacement = sum(ver_positions)
      lagger = 2 * board.size - 1 - max(ver_positions)
      hor_displacement = sum(hor_positions)

    return - p * ver_displacement + i * lagger - d * hor_displacement - p * sum(op_p)
```

It's obvious that we should use the total vertical distance to evaluate the game because a smaller vertical distance means more possiblility to win. And it's also better to stay close to the center line of the board. When a piece is on the edge of the board, it has less chances to interact with other pieces, which means less possiblity to make a long hop. We also consider the "lagger" piece. If one piece is left behind by other pieces, it's more likely to move step by step. The last factor we consider is the total vertical distance of opponent's pieces. The reason is that our opponent gives priority to his own movement. If we don't consider our opponent, the "MIN" in MiniMax is meaningless.

3 Test and Performance

In order to test the validity of our play agent, we make a series of testing games in which our agent competes with a greedy agent. The greedy agent tries to maximize its utility at each move. During the competition, we also modify the parameters of the evaluation funciton so as to let our agent more powerful. We finaly set the parameters p, i, and d to be 2, 10, and 5 respectively before pieces of two players are saparated with each other, and to be 1, 5, 0 after.

The detailed competing results are show in figure, in which we can see our agent has absolute advantage over the greedy agent, since it wins all of the 100 sequential games which it takes the first step. When our agent takes the second step, it appears to be less advantageous than the previous situation, but it still shows much more superiority to the greedy agent, with 98 wins and 2 ties in a sequence of 100 games.



(a) Our agent takes the first step

(b) Our agent takes the second step

Figure 1: Results of simulation games with a greedy agent

4 Discussion & Conclusion

In this assignment, we implemented an inteligent agent pleyer to play Chinese Checkers. Concretely, we assume both players are intelligent enough, and utilize depth limited MiniMax algorithm to select a next step. In order to enhance time efficiency, we further implemented alpha-beta pruning so that the MiniMax algorithm can go down deeper through the adversarial tree.

On innovation of our design is that, we realise that it is not reasonable to continue with MiniMax algorithm after the pieces of both players saparate with each side. At this moment, both sides will no longer interact with each other, thus we do not have to consider the opponent's movement. As a result, our agent only goes down the tree to maxmize its own utility. In practice, our disign effectively booms the performance of our agent.