**AIND PROJECT 2 Heuristic Analysis**

In this project, heuristic scoring function plays a very import role in determine which player is going to be a winner. But if the ‘SearchTimeout’ occurred, the player always lose the game. So, a ‘Good’ heuristics must be

1. ‘**simple**’ enough so that not much time is consumed to calculate the score of the board state, and
2. ‘**accurate**’ enough so that it decides which state is advantageous for player, eventually leads to win.

In order to make the best heuristic, I think it is simplicity we need to consider preferentially. Because we use iterative deepening technique and the end of game state is obvious, one player can’t move anywhere.

Next, we can think some information we can use in the Board class.

1. move\_count
2. Number of the legal moves of each player.
3. Relative, and absolute location in the board, board state
4. Can my movement be limited by the other player?

By considering these, I tried to find a heuristic that could win AB\_improved significantly, which use difference of own\_moves and opp\_moves as heuristic score.

* **Heuristics**

1. **Custom\_1**

**if** game.is\_loser(player):  
 **return** float(**"-inf"**)  
**if** game.is\_winner(player):  
 **return** float(**"inf"**)  
own\_moves = len(game.get\_legal\_moves(player))  
opp\_moves = len(game.get\_legal\_moves(game.get\_opponent(player)))  
**return** float(1.5 \* own\_moves - 2\*opp\_moves)

This heuristic have different coefficients to own\_moves and opp\_moves compared to AB\_improved to see what is more important feature.

1. **Custom\_2**

**return** 2\*own\_moves / (opp\_moves+1)

Just division of players moves. Not really different from AB\_Improved and Custom\_1 in terms of it gives high score when own\_moves is greater than opp\_moves.

1. **Custom\_3**

w, h = game.width / 2., game.height / 2.  
y, x = game.get\_player\_location(player)  
y2, x2 = game.get\_player\_location(game.get\_opponent(player))  
center\_dist = abs(w - y) + abs(h - x)  
own\_moves = set(game.get\_legal\_moves(player))  
opp\_moves = set(game.get\_legal\_moves(game.get\_opponent(player)))  
**if** game.move\_count > 30:  
 hide = bool(own\_moves & opp\_moves)  
 **return** float(len(own\_moves) - len(opp\_moves) + center\_dist / 3 + hide)  
**return** float(len(own\_moves) - len(opp\_moves) + center\_dist / 3)

Now I’m trying to use more information on the board like distance from the center, and move\_count, whether player 2 can limit my moves or not. Also, I changed the strategy after 30 moves.

1. **Custom\_4**

w, h = game.width / 2., game.height / 2.  
y, x = game.get\_player\_location(player)  
center\_dist = abs(w - y) + abs(h - x)  
own\_moves = set(game.get\_legal\_moves(player))  
opp\_moves = set(game.get\_legal\_moves(game.get\_opponent(player)))  
hide = bool(own\_moves & opp\_moves)  
**return** float(len(own\_moves) - 2\*len(opp\_moves) + center\_dist/3 - hide)

-hide, instead of + hide.

1. **Custom\_5**

w, h = game.width / 2., game.height / 2.  
y, x = game.get\_player\_location(player)  
y2, x2 = game.get\_player\_location(game.get\_opponent(player))  
center\_dist = abs(w - y) + abs(h - x)  
player\_dist = abs(y2 - y) + abs(x2 - x)  
own\_moves = set(game.get\_legal\_moves(player))  
opp\_moves = set(game.get\_legal\_moves(game.get\_opponent(player)))  
hide = bool(own\_moves & opp\_moves)  
  
**return** float(len(own\_moves) - len(opp\_moves) + center\_dist/3 + 1/player\_dist + hide)

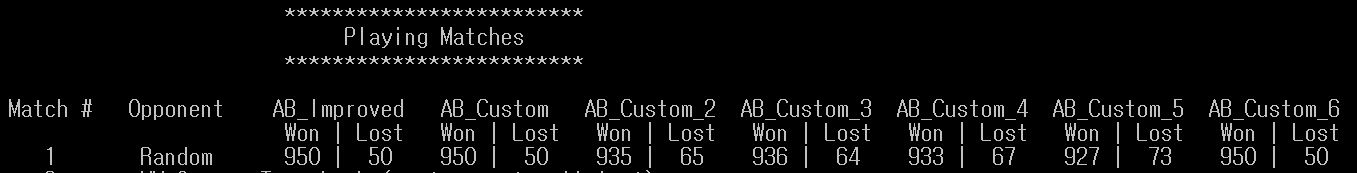
I also looked at the effect of distance between the two players.

1. **Custrom\_6**

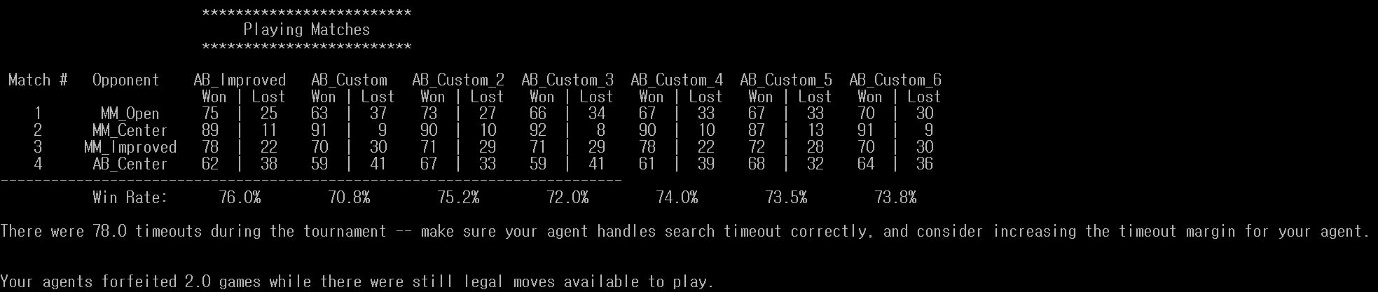
w, h = game.width / 2., game.height / 2.  
y, x = game.get\_player\_location(player)  
y2, x2 = game.get\_player\_location(game.get\_opponent(player))  
center\_dist = abs(w - y) + abs(h - x)  
player\_dist = abs(y2 - y) + abs(x2 - x)  
own\_moves = len(game.get\_legal\_moves(player))  
opp\_moves = len(game.get\_legal\_moves(game.get\_opponent(player)))  
count\_weight = game.move\_count // 10 + 1  
**return** float(own\_moves - opp\_moves - center\_dist/count\_weight + 1/(player\_dist+count\_weight))

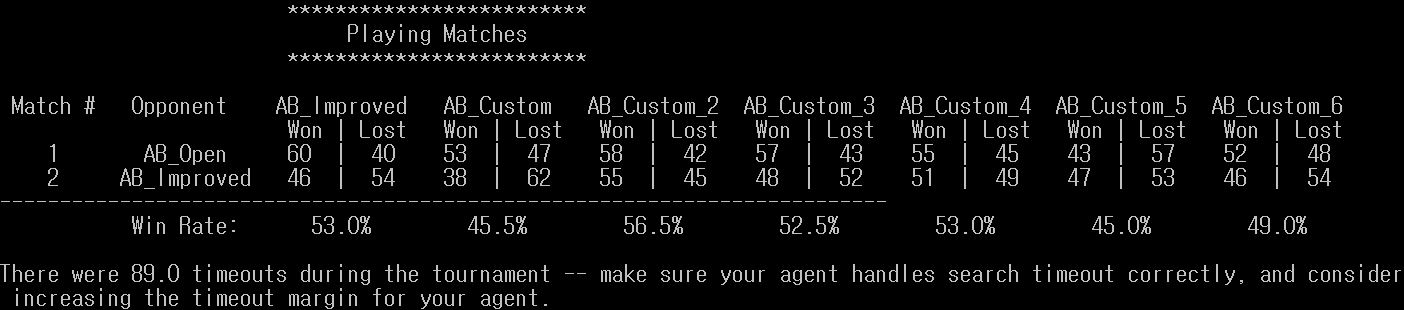
I adjusted the weights according to move\_count.

* **Match results** Intel M-5Y10c CPU @ 0.80GHz 1.00 GHz



1000 games with Random Player. Every heuristic shows approximately 95% of win rate. But why can’t I always win? There is so called ‘**A move of God**’, which seems like in poor position in our heuristics but ultimately lead to win.

 This is 100 isolation games with MM player and AB\_Center. The winning rate is still good enough, about 70% + alpha.



However, I failed to find the heuristics that overwhelming AB\_Open and AB\_Improved. Custom heuristics show 40% to 60% of win rate in 100 matches so these heuristics can be considered as equally powerful.

I think this result can be interpreted for two reasons.

1. Other features except number of players valid moves have no effect on the game.
2. It is helpful but simple heuristics can go down deeper in game-tree in limited time. I think this inference is supported by the difference in win rate against to the Min Max Open heuristic and alphabeta Open heuristic.

* **Sugestion**

As introduced in the video lecture, I also tried to implement the ‘mirror’ strategy that player1 take the center at the beginning and if player2 choose the first location that player1 can move next point symmetrically based on the center. e.g. (p1(3,3) ‘center’ -> p2(2,5) then -> p1(4,1) …)). To do this, I modified some code in Board class, but I realize that if I change the codes except game\_agent.py, then udacity submit fails. Anyway, if mirror strategy applied, against to player2 selecting first location randomly, since there are 8 possible mirror locations out of total 48 remaining board cells, the strategy would increase the win rate a little (1/2 \* 1/6) compared to original heuristic function. So, I suggest applying mirror strategy to one of heuristics. It is very simple and It guarantees victory in possible situation. Here is one part of the code that I had tried.

1. **In the heuristic function.**

*# Take the center.***if** game.move\_count == 1: *# player==player1.* **if** y == center\_y **and** x == center\_x:  
 **return** float(**"inf"**)  
*# If mirror movement is possible. At the Beginning.*

**if** game.move\_count == 3:  
 opp\_y, opp\_x = game.get\_player\_location(opp)  
 **if** y == 2 \* center\_x - opp\_y **and** x == 2 \* center\_y - opp\_x:  
 **return** float(**"inf"**)  
  
**if** game.mirror:  
 opp\_y, opp\_x = game.get\_player\_location(opp)  
 **if** y == 2 \* center\_x - opp\_y **and** x == 2 \* center\_y - opp\_x:  
 **return** float(**"inf"**)

1. **Board.play method in Isolation.py**

**while True**:  
  
 legal\_player\_moves = self.get\_legal\_moves()  
 game\_copy = self.copy()  
  
 move\_start = time\_millis()  
 time\_left = **lambda** : time\_limit - (time\_millis() - move\_start)  
 curr\_move = self.\_active\_player.get\_move(game\_copy, time\_left)  
  
 *# For Mirror Strategy* **if** self.move\_count == 3:  
 center\_y, center\_x = self.width // 2, self.height // 2  
 y, x = self.get\_player\_location(self.\_active\_player)  
 opp\_y, opp\_x = self.get\_player\_location(self.\_inactive\_player)  
 **if** y == 2 \* center\_y - opp\_y **and** x == 2 \* center\_x - opp\_x:  
 self.mirror = **True**