Targeted Algorithm Analysis

The heuristic function of the game is *board\_score.* The time taken by that algorithm varies depending upon the size of the board, number of pawns, bishop, knight, rook, and king. Majority of the time taken by the heuristic function by calling np.where function to go through the whole board, and calculating the score based on number of pieces.

In term of the Big-O complexity, the function np.where seems to take the most time, but it is written in C, so I assume np.where goes through whole array once. And np.where is used check where is the piece on the board, so the time complexity of np.where for the *board* array is,

**Heigh of the board \* Width of the board**

The heuristic function calls the np.where 11 times (5 times of white, 5 times for black, and 1 time to check if both the kings are alive) in most of the cases. So the time complexity of np.where in that case would be,

**11 \* (Heigh of the board \* Width of the board)**

Also, the program goes through each piece that is on the board to calculate the score, and so the Big-O calculation also varies based on number of pieces on the board. So, the total time complexity in terms of Big-O for the heuristic function is:

**Big-O(11\* (Heigh of the board \* Width of the board) + Number of pieces left on the board)**

Removing the constant terms:

**Big-O((Heigh of the board \* Width of the board) + Number of pieces left on the board)**

And the lower bound of the heuristic function (big-Theta) is when either of the kings is captured. In that case, the np.where function is called twice, so the time complexity, in that case, would be,

**Big-Theta(2 \* (Heigh of the board \* Width of the board))**

Removing the constant terms:

**Big-Theta(Heigh of the board \* Width of the board)**

|  |  |
| --- | --- |
| **Big-O** | **Big-Theta** |
| (Heigh of the board \* Width of the board) + Number of pieces left on the board | Heigh of the board \* Width of the board |