***squareWeights():***

**//Invariants-Based Representations**

In this game a position can be reflected symmetrically along horizontal, vertical or any of the two main diagonal axes as well as rotated by the , K= 1, 2,3 degrees without affecting the board evaluation[1]. Taking into account these invariance properties can simplify the form of the evaluation function (by lowering the number of weights) and also help keeping it consistent (by applying the same weights/assessments to isomorphic game position).

In the game of Othello, board representation method can capitalize on the existence of 4 symmetry lines, which from 8-way symmetry. This allows treating particular squares as indistinguishable, since from the algorithm point of view they bear the same meaning. For example squares b3, b6, c2, c7, f2, f7, g3, and g6 from such an “equivalence class” in the board squares space[1]. Consequently, coefficients assigned to these fields in the WPC evaluation function should be pairwise equal and on the algorithmic level all of them may be treated as one parameter. Similarly, any game pattern covering one of this fields can be automatically transformed into other parts of the board by using symmetry lines[1].

Division of the Othello board into 10 abstract classes, denoted from A to J, according to board symmetries. In Othello-related literature elements of class B are usually referred to as C-squares and the E-class ones as X-squares.

std::vector<int> weights = {

200, -100, 100, 50, 50, 100, -100, 200,

-100, -200, -50, -50, -50, -50, -200, -100,

100, -50, 100, 0, 0, 100, -50, 100,

50, -50, 0, 0, 0, 0, -50, 50,

50, -50, 0, 0, 0, 0, -50, 50,

100, -50, 100, 0, 0, 100, -50, 100,

-100, -200, -50, -50, -50, -50, -200, -100,

200, -100, 100, 50, 50, 100, -100, 200,

};

***corners():***

If a player cannot move any of their pieces (cannot flank any of the opponent’s pieces), their turn passes to the other player. If neither of the players has a move available, then it'’ game over, and the winner is the player with the highest number of pieces on the board; likewise for the case where all 64 pieces are on the board. This is clearly a deterministic, perfect information, zero-sum game. Therefore, one can develop an AI under a Minimax search. Heuristics applied to this game seek to improve the performance of the search (Minimax); some of these heuristics are as follows:

***diskDifference():***

This function indicates relative disc difference between the two players. A basic feature to analyze and build a heuristic from in Othello is piece difference; i.e., the difference between black and white pieces. Ultimately, the value obtained is the percentage of black (B) or white (W) pieces on the board, except when W = B. The calculation goes as follows:

• (B > W): 100 \* B / (W + B)

• (B < W): 100 \* W / (W + B)

• (B = W): 0

• Corner Occupancy:

Corners are key positions in an Othello game; the player controlling corners controls a big part of the game. Corner occupancy measures how many corners are owned by each player. To compute the corner occupancy, we count the number of black pieces in corners, B, and the number of white pieces in corners, W. We then let the corner occupancy score be:

• 25B − 25W

• Corner Closeness:

Squares contiguous to corners can be deadly if the corner is empty; they can create an opportunity for the opponent to capture the corner. Therefore, corner closeness measures those “deadly” pieces adjacent to empty corners. To compute the corner closeness score, we count the number of black pieces adjacent to corners and the number of white pieces adjacent to corners. The final score would be:

• -12.5B + 12.5W

***Mobility():***

This function shows number of possible moves. One of the worst scenarios in Othello occurs when a player is out of moves and misses their turn; thus, this heuristic measures how many moves a player has. As with the Piece Difference heuristic, it’s calculated as a percentage, as follows[2]:

• (B > W): 100 \* B / (W + B)

• (B < W): 100 \* W / (W + B)

• (B = W, W = 0, B = 0): 0

***IsLegalMove():***

returns true if a move to cell (i, j) is valid according to board specifications