Othello Writeup

Toh Yong Cheng 1000039
Swayam Narain 1000183

Note:

The player that we are using in this game is *ycsmartestplayer.YcSmartestPlayer* or *ycsmartestplayer_no_memo.YcSmartestPlayer* (with no memoization).

1. **Representation of the Othello Game State:**
   The Game State is represented via a 8x8 array, where each cell in the grid is a character of either 'B' to represent Black, 'W' to represent White and 'G' to represent empty. We also use a character to describe which player's turn it currently is, 'W' for white and 'B' for black.

   We also convert it to a bitboard representation for the array to store as key for memoization in dictionaries. This is done so by storing a tuple of two 64 bit integers that are formed by using bitmasks on the gamestate array.

2. **Search Algorithms used:**
   Alpha-beta Pruning using the evaluation heuristic. Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree.

   Moreover, we have a variable number of plies but most of the search happens using 6 plies. However, one can comfortably play an 7 ply game using this AI or even 8ply if they switch off a part of the caching module. However, when the game reaches close to the end game states, it increases the number of plys to up-to 14 plys.

3. **Methods for maximizing your search space over a fixed amount of time:**
   a. Alpha-Beta Pruning used to minimize the branches that are visited
      i. With Alpha beta pruning, the branches which are redundant to search are ignored by comparing the value returned by one of the branches to the running alpha and beta values. If the value is above beta or below alpha (depending on whether it is the max player or the min player), it is pruned from the search.
   b. Memoization of valid moves and board configuration + move
      i. This has been exhaustively described in section 5, but to summarize, the

c. Search Depth that increases when it reaches endgame.

4. **Board evaluation heuristic:**
    a. **Game-phase dependent weights for squares**: The game is divided into 3 phases: opening, middle and end-game., and each phase has its own set of weights. Each square has a certain weight/score for either black or white discs. These is added up to give a score for the max player for his colored discs, and the min player for his colored discs.

    b. **Coin Parity**: The number of discs for the max-player minus the number of discs for the min-player

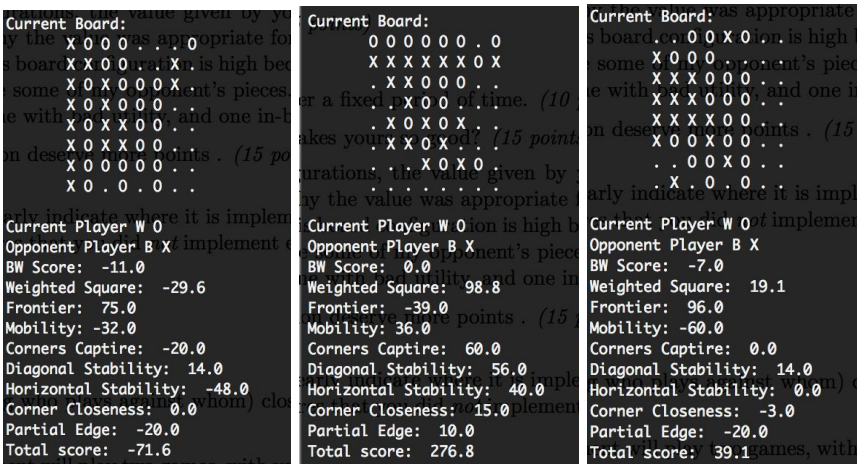
    c. **Mobility**: The mobility for each state is determined by the difference between the number of available moves for the max player and the min player.

    d. **Potential Mobility**: The potential mobility for each state is determined by the number of frontier squares which are occupied by the player's color and are adjacent to an empty square. This means that there could be a potential move at that square.

    e. **Edge Stability**: Edge stability is calculated for stable discs that lie on the edge of the grids. Discs are only stable when can be guaranteed that they cannot be flipped. In this case, discs are only stable when they are adjacent to a stable edge disc, or a corner of the same colour. It is separated into 2 different types: horizontal/vertical or diagonal ede stabilities

    f. **Corner Capture**: Corner Capture gives a score to the player for having their colored discs on a corner of the grid.

    g. **Corner Closeness**: Corner capture is a score to give negative scores to adjacent squares to an empty corner square. This is because placing discs at these adjacent squares run the risk of allowing the opponent to place a disc in the corner.



Figures showing the bad, good and in-between utilities and configurations.
Left: Opponent controls 2 corners, even though I control 1. Notice that the corner I have currently does not give me much advantage
Center: I control 2 corners and am starting to achieve horizontal stability at the top.
Right: No corners are achieved yet, and there is not a lot of mobility for myself

5. Anything cool?

Memoization & Memory Management Module (4M)

Since we are limited to 80MB of memory, we wrote an intelligent, concurrent memory management tool. Coupling the memory management module with memoization of game states, we achieved between 30-40% reduction in the time taken for the search. The following observations and steps were taken to realize this module:

1. For every call to AlphaBeta, memoize the best move to play at a given board and depth

The first observation important to memoization is that given a board and the depth at which alphabeta was called, it will always return the same result. This case happens in the scenario where the same four moves are considered in different orders. While these moves are tested for in different phases of the alphabeta search, the outcome of the three moves given the same board state is the same. For instance, assume that the gameboard is at state S, and four moves are considered M1, M2, M3, M4, playing M1 M2 M3 M4 in order from state S results in the same state as playing M3 M2 M1 M4. Since both the states are same, the cache will be hit and therefore time to search is reduced.

The player at a board need not be memoized because that information is already captured by the depth variable. If the depth is odd, we have a max player whereas if it is even, we have a min player. Since the depth is either even or odd, the same depth will always be found by only and only one player (either min or max).

2. For every call to AlphaBeta, memoize the list of moves at a given board and player

This is simpler. From our benchmark, we found the number of calls to sent to generating the list of moves at a board to be a bottleneck to performance. Therefore we simply created another memo that stored the board and the current player as key and the list of valid moves as its value.

3. Key and Value optimizations for storage

Since we are limited to only 80mb, it makes sense to compress the data that is memoized as much as possible while being quick to uncompress this data again when the memo is hit. To do so, we treated the board as a tuple of two 64 bit integers and the move list as a single integer (generated via bitmasks)

4. Concurrent Memory Management to make sure Memos don't exceed size limit.

In an ideal world we would like to memo as many states as possible, but since we have a hard limit to the number of states that can be memoized (80MB) we implemented a smart memory manager that runs every three seconds to check the amount of memory left. If it was close to a threshold, it releases every board that is cached that have fewer tokens than the initial board that is being tested.If cleaning redundant boards is not enough, the entire cache is cleaned and we start over.

The effectiveness of this module comes from the fact that it removes the list of redundant cached states in O(n) time. This is done by simply counting the number of bits that are high in each bitboard, and marking those that have fewer bits than the reference board for deletion. This works because of the observation that the number of tokens on the board strictly grows in time.

Finally, we respect the rules of the game and stop the memory manager as soon as the best move is determined after running alphabeta search. It is restarted when it is time to search for another move.