Next Up Previous
Next: Time Comparisons **Up:** Parallel Implementation and Optimization **Previous:** Minimax and Alpha-Beta

Othello implementation

To implement and parallelize the minimax and alpha-beta algorithm, I chose the game Othello. (For Othello playing instructions, see Appendix A). This game has simple rules that can be easily translated to computer instructions, while retaining a challenging strategy.

Because the implementation is a game, not only are the number of nodes searched a factor in optimization, but also the actual clock time required by the computer to make a move. The number of nodes is important, and easily reducing this value will necessarily reduce the amount of time required to make a move. However, streamlining the code in other ways to provide further optimizations will allow a search to a given depth to be completed more quickly.

Another factor to consider is the quality of the game played. There are many ways to evaluate a given board which require very little time, but such evaluations often reduce the quality of the game. A tradeoff must be made between playing fast and playing well. Since the only factor that the computer player uses to choose a move is the backed-up score generated after searching a given number of moves, it can make a better move if it searches further in the future of the game and has more information on which to base a decision. The faster score evaluation is at any level, the deeper a search will be possible.

Score storing and evaluation

The current game represents scores by a single integer, with higher scores more advantageous for the black player and lower scores more advantageous for the white player. The score is comprised of values for the total number of pieces, player mobility, and piece stability. With an integer score, comparisons consist of a single line. Storage and management require little memory.

The current static evaluation function requires the board to store individual piece counts along with placement information. This is rather quick as such values need only to be referenced, not computed, with each score evaluation. Also, there is a special case for static evaluation, actually a separate function, which is used to determine the score at the end of the game. At this point the mobility, or the number of moves available, is automatically zero for both players and doesn't need to be computed. Likewise, since no moves can be made, all pieces for each side are stable, or unable to be flipped, so this value can simply be copied from the total piece information. This function is much smaller than the overall evaluation function and greatly reduces evaluation time needed at the endgame.

Othello Types and Functions

board type
: This is a 4-part record. The board itself is stored as an 8x8 array of pieces which can be black, white, or empty. Also stored are: the current player, or which player is next to move; how many pieces belong to Black; and the total number of pieces on the board, since the number of White pieces can be derived from these numbers.

score type
: This is an integer comprised of 4 parts. The values for stability, mobility, and total pieces are each represented by 8 bits and computed by subtracting the white value from the black value and adding 64 to

eliminate negative numbers. These numbers are packed into one integer using bit shifts. The last value, the most significant 2 bits, determines if the game has a win or is unfinished.

generate_moves
: This function steps through each square on the board. If the square is empty, it determines if it is a valid move for the current player by looking for a string of opposite colored pieces adjacent to it that end in a piece belonging to the player. If the move is valid, it adds it to a list of valid moves. It returns this list.

apply_move
: This function updates the board by placing a piece at the chosen move and flipping any necessary pieces. It increments the total piece count and adjusts the Black count with every piece flipped. It also switches control of the game to the other player.

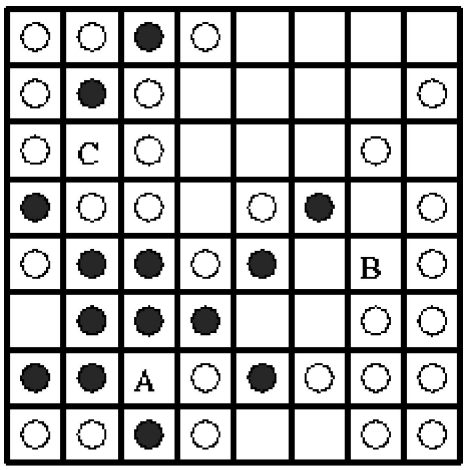
null_move
: Player forfeits turn because no moves exist. Current player is switched.

static_evaluation
: This function takes the board and the mobility value for the previous player as input. It computes pieces for each side from the information stored with the board. It computes mobility by generating moves for the current player and comparing that to the moves available to the prior player. The method for computing stability is discussed below. This function is only called if the game outcome is undecided, so this value is predetermined. The resultant values are packed using bit shifts and the score is returned.

piece_evaluation
: This function is only called when an end to the game is detected. Thus, the only values needed can be derived from the piece values. Each square on the board need not be examined.

Stability

In Othello, a piece is stable when no other piece can be played that will cause it to be flipped. This occurs when a piece is in rows that are completely filled in all four flipping directions. It also occurs when a piece is next to a row of stable squares of its own color in each of the four directions. As shown in Figure 3, piece A is stable because it is in filled rows in all four directions, piece B is stable because it is adjacent to stable squares in all four direction, and piece C is stable because of a combination of these.



Squares A, B, and C are filled with White pieces.

Figure 3: Examination of Piece Stability

The current stablility function recomputes piece stability with each static evaluation. This function determines which rows are filled and marks any qualifying squares as stable. It then examines the corners and does the same. Any squares that are found stable are put on a list. A search of this list looks for squares adjacent to these stable squares which might also be stable due to their proximity the first square.

This stablility function finds all stable squares in one iteration, partially compensating for the time needed to completely reevaluate the board at each move. However, since by definition, the number of stable squares for each player is a monotonically increasing figure, I feel this can be improved upon by storing the stability information in addition to the list of squares to search from. When a stable square is completely surrounded by other stable squares, it can be removed from the search list, reducing the amount of searching necessary.

Lastly, I have observered that there is no possibility that any square in the game can be stable until a piece is played in either a corner or one of the edge squares directly adjacent to it on either side. The corner is an obvious square, but the second square in along each edge is necessary to provide that a square may be in completely filled rows in all 4 directions as Figure 4 shows. If stability checking is turned off until this point in the game, there is an additional possibility for early speedups.

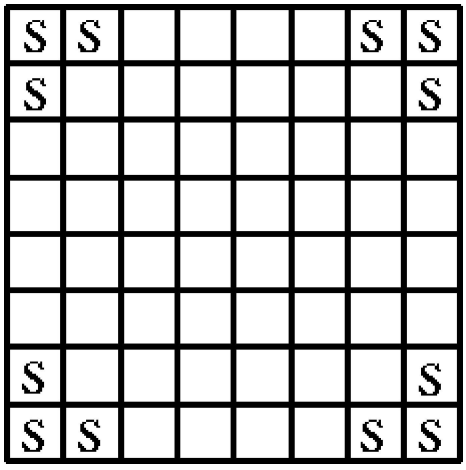


Figure 4: Squares necessary for Stability

Next Up Previous

Next: Time Comparisons **Up:** Parallel Implementation and Optimization **Previous:** Minimax and Alpha-Beta

---

abierman@cs.caltech.edu