**A Program to Play Othello**

Rachel Mertz Matthew Varas

Affiliation: St. Cloud State University Affiliation: St. Cloud State University

mera0902@stcloudstate.edu vama1304@stcloudstate.edu

**ABSTRACT**

Othello is a two-player board game which is well-suited to the application of artificial intelligence. A program was written to implement the Othello game and play it interactively against an opponent. Game states are configurations of the game board represented by two-dimensional 64-cell arrays; these states are arranged in a game tree which is traversed using the Minimax decision rule and a basic static evaluation function to determine the program’s best move from the current state. The depth to which the tree is examined is entered by the user at the start of the game. Program performance is tested using various depths; the results indicate room for improvement, given that novice opponents won against the program eight games out of ten. Improvement could be achieved by adding one or more heuristics to the static evaluation function, such as piece stability and corners captured.

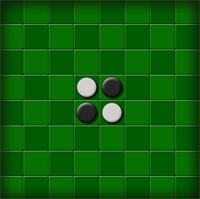
1. **INTRODUCTION**

The study of game theory is an important application area in artificial intelligence. In the following paper, we describe a specific application dealing with the two-player strategy game Othello.

We have written a program which plays Othello interactively against an opponent. To do this, states in the game are represented by various board configurations. These boards are arranged in a game tree, with the advantages or disadvantages of each board judged by a simple static evaluation function (SEF). The program uses the Minimax decision rule to examine the game tree and thus select the best move to make from the current state. The tree is examined only to a depth entered by the user at the beginning of the game; the program’s performance is tested multiple times using different depths.

1. **THE GAME**

Othello is a game of strategy in which two players take turns laying pieces on a game board with eight rows and eight columns. Spaces on the game board are notated using letters A-H for columns and numbers 1-8 for rows. One player lays only black pieces, and the other lays only white pieces. The board is initially set up with two black and two white pieces in a checkerboard pattern in the center (see **Figure 1** following this paragraph).

****

**Figure 1:** Initial board configuration

A valid move is made by laying, for example, a black piece that creates a straight line – vertical, horizontal, or diagonal – with black pieces at both ends and only white pieces between. (If no lines are created by laying the piece, the move is invalid.) When the black piece is placed, all white pieces in the line are turned to black; this action applies to all lines created by the black piece placed. The same rules apply when a white piece is placed.At the end of the game, whichever player has the most pieces of their color on the board wins.

1. **DESIGN CHOICES AND FEATURES OF THE PROGRAM**

The Othello program is written in the C++ language, with the game board represented by a class object whose primary member is a two-dimensional array with 64 cells. This array stores ‘B’ in the board spaces filled by black pieces, and ‘W’ in the spaces filled by white pieces.

As mentioned previously, the states of the game are different configurations of the Othello board, which are created by sequences of moves during the game. These states make up the game tree. It is important to note that the game tree is not stored in any kind of tree in memory; rather, the recursive nature of the Minimax search, which the program uses to move through game states, mimics the action of traversing the tree. To move down a level in the tree, the program makes a move; to move up a level, the program “unmakes” that move, reverting to the most recent state of the game board.

Continuing with this idea, the program uses the Minimax rule to search through the states represented in the game tree. To begin the search, the program moves down through the tree to a level governed by the depth parameter entered by the user. At each leaf, the game state is assigned a score by the state evaluation function. Using the Minimax rule to *minimize* black SEF scores and *maximize* white SEF scores, the highest possible SEF score for white travels up the game tree. The program then makes the move that corresponds to this score.

The static evaluation function uses a single heuristic – mobility – to calculate a score for the game state to which it is applied. Essentially, mobility denotes the number of moves it is possible for a player to make given the current state of the game board. The SEF implements this measure by scanning the board and counting all the valid moves that can be made by one of the colors from the current game state. The number of valid moves is the output of the SEF.

1. **USING THE PROGRAM**

The game starts by taking input from the user: an integer for the lookahead depth, and either ‘B’ or ‘W’ to choose the color that makes the first move. By default, the program always makes moves for the white player; that is, the artificial intelligence will automatically select and make white moves only. Black, then, is the program’s opponent, and the user must enter moves for black. Black moves can be input using letter-number combinations, for example H8, which corresponds to column H, row 8. (Letters entered must be uppercase.) When the black player has no valid moves, the user must enter a pass using S1. The game board is displayed after each move.

The game ends when there are no valid moves remaining. This happens in two cases: either when two passes in a row have occurred, or when all spaces on the game board are filled. When the game ends, the number of pieces on the board for each player are counted, and the program indicates the winner (or a tie).

1. **THE PROGRAM’S PERFORMANCE**

Performance testing consisted of playing ten games against the Othello program. In each game, the black. Player (the program’s opponent) was allowed to make the first move. **Table 1** below presents relevant results of the testing.

**Table 1:** Game results

|  |  |  |
| --- | --- | --- |
| **Depth** | **Winner** | **Score** |
| 2 | Black | 47-17 |
| 3 | Black | 36-28 |
| 4 | Black | 50-14 |
| 5 | White | 52-12 |
| 5 | White | 37-27 |
| 5 | Black | 38-26 |
| 5 | Black | 45-19 |
| 5 | Black | 36-28 |
| 5 | Black | 38-26 |
| 6 | Black | 38-26 |

We can see that the program won only two games out of ten. Looking only at this data set, we conclude that, while using only mobility is a valid strategy for the SEF, the mobility measure alone is a simplistic approach that does not lead to good performance. Testing using greater depths would also confirm the intuitive theory that the greater the depth, the better the program plays.

1. **CONCLUSIONS**

We have written a program in C++ which plays Othello interactively against an opponent. The user can enter a lookahead depth (the level to examine the game tree to when choosing the next move), and the program uses the Minimax rule to search the game tree to this depth for the most advantageous move to make. The static evaluation function employs the mobility heuristic to assign an SEF score to a game state. A small set of tests of the program revealed sub-par performance: eight games lost out of ten.

In considering improvements, better performance could likely be achieved by adding other heuristics to the SEF, such as the stability of particular pieces and the importance of capturing corner positions on the board. In theory, the greater the depth, the better the program’s performance, so more testing could also be done using greater lookahead depths to establish a relationship between depth and performance.