

Playing Othello with Artificial Intelligence

Michael J. Korman

December 11, 2003

Abstract

In this paper, we describe the Othello project completed by Michael Korman and David Walluck in the Fall semester of 2003. We will examine the general paradigm for artificial intelligence game playing, followed by a look at a powerful optimization technique known as $\alpha\beta$ -pruning. Next, we will move on to a description of MLia, the AI created for our project. We finish with a discussion of OthelloGUI and OthelloD, the graphical client and the server.

1 Two-Player Games

Othello is known by game theorists as a *two-player game of perfect information*. Games in this category are played by two players, each player being completely informed at all times of the state of the game. Neither player has information that is concealed from the other player. Games of perfect information have several characteristics that make them worthy of study. One such quality is that there always exist one or more optimal solutions of play, due to the lack of uncertainty at each move. Nonetheless, such optimal solutions are often expensive to compute, and for games such as Othello may be intractable. Those who wish to construct artificial intelligence to play these games are therefore forced to rely on heuristics that approximate optimal game play.

1.1 Minimax

Minimax is a technique that yields discovery of the optimal solution to a two-player game of perfect information, given enough time and space. It has the appealing property that it can be applied only partially, giving a solution that is as good as the amount of time at one's disposal.

The minimax technique relies on the conceptually simple idea of *game trees*. A game tree is an expansion of the total state space of a game. Each node of the tree contains the complete description of one state of the game. In the case of Othello, this information would consist of a snapshot of the board, showing the location of each piece. The root of the tree consists of the starting configuration. Its children are the boards that result from all possible moves of the first player. Its grandchildren are thus the boards that result from all possible moves of

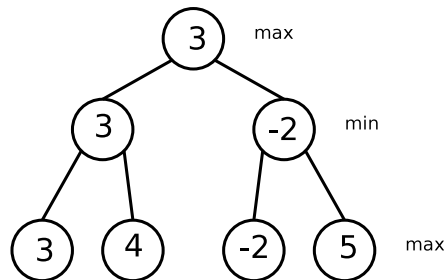


Figure 1: An example of the minimax procedure.

the second player. This continues down the tree, until the entire state space has been expanded. The leaves contain snapshots of the board at winning or drawing positions. Every path from the root to a leaf signifies an entire game.

The key to minimax is the *evaluator*. The evaluator takes a board and returns a score within a specified range. Formally, the evaluator is a function $e : \mathcal{B} \rightarrow [m, M]$, where \mathcal{B} is the set of all possible boards, m (respectively M) is the minimum (respectively maximum) allowable score for a board. One player is designated as the *maximizing* player, and the other the *minimizing* player. If a board has a score of M , this signifies that the maximizing player has won. If it has a score of m , it means that the minimizing player has won. If the score is 0, the board configuration is a draw. Therefore, it is in the maximizing player's best interest to seek boards that have a positive score, and likewise it is in the minimizing player's best interest to seek boards that have a negative score. For an example of techniques used in evaluation functions, see §2.1.

To explain the minimax method, suppose we would like to use it to play a game as the first player. Since we are the first player, we are the maximizing player. Our ultimate goal is to find which of the children of the root yields the highest score. To compute this, we consider each of the children individually, and apply the minimax procedure to them one-by-one. However, since we are now on the second level, we must play the part of the minimizing player, whose turn it now is. Therefore, we try to find the node that yields the lowest score. We continue this way down the tree, alternating between maximizing and minimizing, until we reach the bottom. At the bottom, we apply the evaluator to each of the leaves. The scores are then propagated up through the tree until we get back to the root. Now, the score of each child is known, and we can choose the *best* move: the greatest if we are the maximizing player, or the least if we are the minimizing. See Figure 1 for an illustration of how the score of a root node is derived from its children.

There is one problem, however. Due to the combinatorial explosion of the game tree, there is no way we can hope to reach the bottom of the tree from the top. Instead, we must define a fixed depth that we wish to search to. As we expand down the tree, we approach our fixed depth. The nodes at the depth

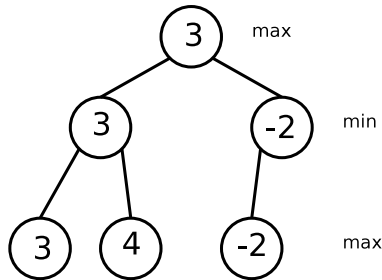


Figure 2: The minimax tree with $\alpha\beta$ -pruning.

form our *search horizon*, and will be treated as the leaves. We then apply the evaluator to these nodes, and hope that they serve as an approximation of the actual leaves. The evaluator must now return a value between m and M that indicates how close each player is to winning.

1.2 $\alpha\beta$ -pruning

It turns out that there is a clever optimization technique that allows minimax to run with significant speedup. The general strategy of *pruning* enables us to avoid searching certain branches of a tree when we know the optimal solution is not to be found in those branches. The specific pruning technique applicable to minimax search is known as $\alpha\beta$ -pruning.

To perform $\alpha\beta$ -pruning, we start with two parameters, α and β , α initially equal to $-\infty$ and β initially equal to ∞ . If we have reached the search horizon, evaluate the current node and return its score. If the current level is a minimizing level, we perform the $\alpha\beta$ algorithm on each of the children, until $\alpha \geq \beta$. To do this, we evaluate perform $\alpha\beta$ on the child, and determine if the evaluation is less than β . If it is, reset β to be the evaluation. Finally, return β . If the current level is a maximizing level, we perform the same sequence of steps, replacing β with α and $>$ with $<$.

Figure 2 shows the tree from Figure 1 with $\alpha\beta$ -pruning applied. Note that the 5 does not need to be explored, since the minimizer is guaranteed to choose a node ≤ -2 , and the maximizer has already found something greater than that.

1.3 Iterative Deepening

We note that as the tree expands, each successive expansion potentially leads to a larger number of nodes than were present in the previous level. Because of this, the minimax algorithm will take longer time to compute each level as it expands the tree. This means that searching to a specified depth will take a different amount of time depending on what level we are starting from. The problem arises when we limit ourselves to a fixed time to perform our search,

as is common in tournament play. Instructing the AI to search nine levels deep may only take a few seconds early in the game, when there is a low branching factor, but may take several minutes later in the game, when the number of possible choices has increased. We would like to tell the AI to search for a specified time, rather than a specified depth.

The solution to this is called *iterative deepening*. The core idea is to run several searches sequentially, at successively higher depths. Suppose we wish to perform a minimax search that lasts T seconds. We begin by searching to level one, and then check our elapsed time t_1 . If the time is less than T , we search to level two, and check time t_2 . We continue in this manner until we search to level n , where $t_n \geq T$. At this point, we return the value obtained by searching to level $n - 1$.

It may seem that since each level involves re-computation of all previous levels, we are wasting considerable time. Surprisingly, this is not the case. To demonstrate, suppose that the time it takes to perform the evaluation is the dominant factor in the time analysis. Letting b equal the average branching factor of the tree, and d the depth, we know that the number of leaves requiring the application of the evaluator is b^d . The sum of the run times of all levels is then

$$b^0 + b^1 + \dots + b^{d-1} = \frac{b^d - 1}{b - 1}.$$

If we compute the ratio of the number of nodes in the bottom level to the number of nodes up to the bottom level, we get

$$\frac{b^d(b - 1)}{b^d - 1} \approx b - 1. \tag{1}$$

This tells us that the number of evaluations at a given level is *much greater* than the sum of the evaluations at all previous levels. Iterative deepening is clearly worthwhile.

2 MLia

The AI we developed for this project is named MLia. The name is a play on two things: Emilia, a character from Shakespeare's *Othello*, and ML, the language MLia was written in.

2.1 Evaluator

MLia uses an sophisticated evaluation function that measures several characteristics to compute its final score. These characteristics are described below.

Piece difference Perhaps the most straightforward characteristic is piece difference, which measures how many pieces of each color are on the board. To calculate the score, we count the number of black pieces and white

pieces on the board. If the number of black pieces B is greater than the number of white pieces W , we let the score p be defined as

$$p = 100 \frac{B}{B + W}.$$

If the number of white pieces is greater than the number of black pieces, the score is

$$p = -100 \frac{W}{B + W}.$$

If $B = W$, then we let $p = 0$.

Corner occupancy Corners are the most valuable squares on the board, and whoever controls them, controls 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 c be

$$c = 25B - 25W.$$

Corner closeness The squares adjacent to the corners can be deadly if the corner is empty, as this can create an opportunity for the opponent to capture the corner. Hence, corner closeness measures the pieces adjacent to empty corners. To compute the corner closeness score, we count the number of black pieces adjacent to corners, B , and the number of white pieces adjacent to corners, W . The corner closeness score ℓ is then

$$\ell = -12.5B + 12.5W.$$

Mobility This characteristic measures how many moves each player has. One of the worst things that can happen in Othello is to be out of moves, forced to pass the turn. Mobility is computed by finding the number of all possible black moves, B , and the number of all possible white moves, W . If $B > W$, then the mobility is

$$m = 100 \frac{B}{B + W}.$$

If $B < W$, then the mobility is

$$m = -100 \frac{W}{B + W}.$$

If $B = W$, or if B or W equals 0, then $m = 0$.

Frontier discs Frontier discs are discs that are adjacent to empty squares. Most of these discs are volatile as they have a greater chance of being flipped by an opponent's move in the empty square. Therefore, we would like to minimize the number of frontier discs we have. This quantity is

computed by counting the number of frontier discs of black, B , and the number of frontier discs of white, W . If $B > W$, then the frontier discs score is

$$f = -100 \frac{B}{B+W}.$$

If $W > B$, then the frontier discs score is

$$f = 100 \frac{W}{B+W}.$$

If $B = W$, then $f = 0$.

Disc squares This characteristic attempts to assign a value to every square on the board. Noting that the board is symmetric about both its horizontal and vertical axes, we only need to assign values to one quadrant, and then mirror them in the other three. The upper left quadrant is represented as a matrix:

$$\mathbf{V} = \begin{pmatrix} 20 & -3 & 11 & 8 \\ -3 & -7 & -4 & 1 \\ 11 & -4 & 2 & 2 \\ 8 & 1 & 2 & -3 \end{pmatrix}$$

These values were determined experimentally. To compute the disc squares score, we calculate a linear combination of the form

$$d = \sum_{i=1}^4 \sum_{j=1}^4 \sigma(i, j) \mathbf{V}_{ij},$$

where $\sigma(i, j)$ is the *color coefficient*

$$\sigma(i, j) = \begin{cases} -1 & \mathbf{V}_{ij} \text{ is white} \\ 0 & \mathbf{V}_{ij} \text{ is empty} \\ 1 & \mathbf{V}_{ij} \text{ is black} \end{cases} . \quad (2)$$

Note that each of these characteristics is normalized appropriately to scale it into the range $[-100, 100]$. This provides uniformity among our characteristics, and ensures that we can multiply each of them by whatever weights we'd like, without concern that there is already a weighting "built-in." Once all of these characteristics are computed, the final evaluation score S is calculated as

$$S = \sum_{k=0}^n w_k c_k, \quad (3)$$

where n is the number of characteristics, c_i is the i th characteristic, and w_i is the *weight* of the i th characteristic. The weights were determined experimentally. We ultimately decided on a weight vector of $(10, 801.724, 382.026, 78.922, 74.396, 10)$.

2.2 Special Techniques

Several other techniques are used by MLia to improve its searching.

Move ordering This technique provides a minor improvement upon standard $\alpha\beta$ -pruning. It tries to hurry the discovery of better moves by “stacking the deck” so that they are searched first. MLia implements move ordering by first computing a shallow search of two levels, and using that as a heuristic for determining which moves are most likely to be better in the long run. It forms a list out of these evaluated moves, and sorts it. Then, it expands nodes from the head of this list.

Dynamic programming Throughout the course of a search, the same nodes appear over and over again, and need to be evaluated many times. Dynamic programming is a technique whereby we avoid recomputing anything that has already been computed. MLia implements dynamic programming by storing all seen nodes along with their evaluations in a hash table. When a node is evaluated, it is first searched for in the hash table. If it is found, then value in the table is used, rather than recomputing the evaluation. If it is not found, the evaluation is computed, added to the hash table for next time, and returned. Experimental data showed that as many as 25% of all requested evaluations were actually retrieved from the hash table throughout the course of a game.

Killer moves There are certain situations when we wish to override the minimax method and perform a tactical move that almost *always* works. MLia is programmed to recognize particular board configurations as begging for killer moves. There is much room for experimentation in this area. One move that was implemented was the capturing of corners. If MLia sees that it can take a corner immediately, it will do so. This may not always be the best thing, but it often works, and can partially make up for a weak evaluator. In Othello, where once a corner is taken, it’s won permanently, the risk is lower. In contrast, hasty capturing of the Queen in Chess can be extremely dangerous.

Iterative deepening Refer to §1.3 for a general discussion of this technique. MLia employs it by utilizing a thread-based design. When MLia needs to compute a move, it delegates the computation to a special thread which successively computes values of deeper and deeper levels, setting a global variable after each one. Meanwhile, the original thread is blocking for the specified amount of time. When it wakes, it reads the value from the global variable, and returns it as the final answer.

End-game solver The minimax method is designed to return an approximation to the optimal solution. Because our evaluation function employs heuristic methods to perform its evaluation, it may not necessarily be completely correct when a true leaf of the game tree is evaluated. To sidestep this problem, we use an end-game solver. When MLia detects that

a node to be evaluated is a leaf, it does not use minimax to evaluate it. Instead, it applies a simple piece count, which will indicate whether the leaf is a win or a draw.

2.3 Stuff that Didn't Work

There were several features we hoped to implement but did not have time for. One of these was an opening book. We started developing this feature, but as it did not work right, it remains commented out in `MLia.ml`. The opening book theoretically operates by pre-generating a huge hash table of opening positions and their evaluation scores, evaluated at a higher depth than is permitted in an actual game.

Another feature we would have liked to have implemented was a genetic algorithm to tune the weights of the evaluator characteristics. The current weights were achieved through experimentation. A genetic algorithm would have made slight modifications to the weights, and then played games with the modified version against the original version and determining which one wins. It would have caused the weights to evolve in this manner, eventually arriving at an optimized assignment of values.

2.4 Implementation

`MLia` was written in the OCaml[8] dialect of ML. OCaml is a fast, type-inferring functional programming language developed at INRIA's Cristal project for programming language research. It provides all of the benefits of functional programming with all of the benefits of imperative and object-oriented programming. OCaml was chosen for its ease of use and speed, the latter of which rivals that of C.

To achieve high modularity in our OCaml program, we based our work off of the two-player game module framework illustrated in [10]. The goal of this framework is to separate the game representation, game evaluation, and game search from each other, so that each could be easily replaced. It would be quite easy to substitute another game besides Othello into this project, and the `MLia` source code would undergo very few changes.

OCaml's exceptions are used to implement $\alpha\beta$ -pruning. The game tree is stored as a list. To perform the search, each element of the list is extracted, and the $\alpha\beta$ algorithm is performed on it. To make a branch cut, an exception is raised, with the new value being passed along with it.

`MLia` has networking code built in, using the amazing OCaml `Unix` library, which provides a nearly complete cross-platform implementation of the Unix system calls. Upon startup, a thread is created to monitor the network connection for incoming messages. Initially, we used the fork system call to do this, as is typical in the Unix world. However, we learned that fork is not available on Windows, whereas threads were ubiquitous. Hence, we switched to threads.

When `MLia` receives a message from the server indicating that it is its turn, it creates another thread to compute the move. This thread is the main thread

that is described in the section on iterative deepening.

3 OthelloD

OthelloD is the Othello server developed for this project. It was written in the Java programming language, and tested with the gcj[2] compiler, a free Java compiler from the GNU project. It allows multiple users to connect and play games. Additionally, it supports a global chat room, in-game chatting, and private messaging. The protocol is text-based and human readable.

A separate component was written to communicate with Professor Russell's server, which we named ACRbridge. This component connects to OthelloD as a normal client, and allows other players on our server to play against players on Professor Russell's server. ACRBridge was also written in Java.

4 OthelloGUI

OthelloGUI is a graphical client for OthelloD. It was written in the C++ programming language. It relies on the gtkmm[6] GUI toolkit, and the GNet[3] network library. Each of these toolkits provides an excellent cross-platform API that theoretically works equally well under Unix and Windows. In practice, however, we found that the Windows implementations of both libraries failed to work properly. Nonetheless, the OthelloGUI code should be perfectly portable, assuming the bugs in its dependencies are fixed. The source code is fully documented, so we will give a broad overview of the structure of the program.

OthelloGUI is designed to work strictly with OthelloD. However, the network code is separated into its own class, and could probably be replaced with a minimal amount of trouble. Unfortunately, OthelloGUI does not yet support all the features that OthelloD provides. It has support for global chat, but lacks support for private messaging, in-game chat, and multiple games. As the focus of this project was developing an AI, we had to make concessions about how much we could implement in the GUI, due to time constraints.

4.1 User Interface

The user interface consists of a single tabbed window. The first tab is the main chat room, the second the list of games, and the third the game window.

The main chat room tab allows chatting among all users of the server. It presents a user list, a chat window, and a text entry box. Users can send a chat message to the server by typing it in the text entry box and pressing the ENTER key.

The game list tab shows a list of all currently active games. Each item in the list has three fields: the game number, the first player, and the second player. As players enter and leave games, this list is updated. By clicking on a game, and selecting the appropriate menu item, a user can opt to join an existing game as either a player or a spectator. Additionally, a new game can be created.

Finally, the game tab displays an tile-based image of the game board. To the right of the board is a user list, and underneath is a chat window. As of this writing, these widgets are not active, but it should require small modification to the source code to implement them, since the server already has all this functionality.

4.2 Program Design

The GUI was written completely by hand, without the use of any interface creation utilities. This was too time-consuming. If we were to implement this again, we would take advantage of such utilities as Glade (<http://glade.gnome.org>).

This C++ project consists of numerous classes. Each of the board, chat widget, game list, and main window has its own class. The main window class, `MainWindow` is the driver for the entire program. It creates a `Connection` object, which acts as a wrapper around GNet functions. Refer to the `OthelloGUI` source code documentation for detailed information on how the project works.

5 Software Engineering

Several tools were of great use in the development of this project. One of these was the Subversion[7] version control system (SVN), a replacement for the venerable CVS. It supports many of the same features as CVS, but also has advanced features such as versioning of directories, renames, and file meta-data. Additionally, transactions are truly atomic. SVN repositories were used for all four components of the project.

The GNU Automake[5] and Autoconf[4] tools were also indispensable. These tools allows easy configuration and compiling of software. We managed to set up Automake and Autoconf for our C++, Java, and OCaml software without much trouble.

The Doxygen[1] source code documentation system was a huge help in the organization of our source code. It operates by scanning source files for special comments and generating a website and a \LaTeX file based on these comments. The generated documentation contains descriptions of each function, with its return values and parameters, and is cross-referenced. It is almost completely compatible with Javadoc. We documented both `OthelloD` and `OthelloGUI` with Doxygen.

One of our primary concerns in the development of this project was portability. Our goal was to have this software run without complaint on all major systems. The GUI was written in Standard C++, the AI in OCaml, which is a portable high-level language that encapsulates traditionally platform dependent services, such as the Unix and thread libraries, and the server was written in Java, and tested with GCJ. Additionally, the C++ program used the GNet network library, which provides a wrapper around the platform specific sockets library. All of these compilers and libraries should be portable.

Unfortunately, our goal was not realized, due to bugs in the Windows implementations of OCaml and GNet. Although GCJ compiles Java code to native machine language, it's performance left much to be desired when compared with the proprietary Sun Java compiler, especially in terms of the garbage collector.

6 Conclusion

We found this project to be entertaining and educational. The scope of the project left much opportunity for exploration of various areas of computer programming, such as graphical user interfaces, networking, multi-threading, and artificial intelligence. The theoretical techniques of minimax and $\alpha\beta$ -pruning will be a useful addition to our repertoire of algorithms.

References

- [1] Doxygen. <http://www.stack.nl/~dimitri/doxygen/>.
- [2] GCJ: The GNU Compiler for Java. <http://gcc.gnu.org/java/>.
- [3] GNet. <http://www.gnetlibrary.org/>.
- [4] GNU Autoconf. <http://www.gnu.org/software/autoconf/>.
- [5] GNU Automake. <http://www.gnu.org/software/automake>.
- [6] gtkmm - the C++ interface to GTK+. <http://www.gtkmm.org/>.
- [7] Subversion. <http://subversion.tigris.org/>.
- [8] The OCaml Language. <http://www.ocaml.org/>.
- [9] Gunnar Andersson. Writing an Othello Program. <http://www.nada.kth.se/~gunnar/howto.html/>, 2003.
- [10] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications with Objective Caml*. O'Reilly.
- [11] Thomas Dean, James Allen, and Yiannis Aloimonos. *Artificial Intelligence: Theory and Practice*. Addison-Wesley, 1995.
- [12] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, 1992.