Bjorn Bjornsson and Alexander Leong

present

“In the Mood for Love”:

A Search for the Most Powerful Othello Player

CS 4701

“Thou weigh'st thy words before thou givest them breath.”

—*William Shakespeare,* Othello

Section 1: Introduction, Initial Goals and Objectives

Othello is a fairly simple game to play but can involve highly complex strategies, making it a particularly interesting problem in the context of artificial intelligence. The game is played on an 8-by-8 board. One player plays black tiles, and the other white tiles. The ultimate objective of the game is to fill the board with as many tiles of your color as possible. However, a valid move must "surround" an opponent's tile(s), either vertically, horizontally, or diagonally. When a player surround their opponent's tile(s), they are flipped to the player's color. This presents a significant problem to a naïve evaluation function, as the proportion of black to white tiles on the board during the middle of the game has very little to say about which player will actually win. In the last few moves, players can easily gain or lose significant leads.

Our initial plans were as follows: we planned to implement an evaluation function that would primarily rely on the number of tiles each player had, with perhaps some consideration for the position of tiles. Further, we planned to use local beam search to decide which move to take next. However, it became abundantly clear early on that such a strategy would not be able to rival even inexperienced human players. In particular, a naïve evaluation function such as the one just described would severely limit the algorithm's performance. Instead of such a simple approach, we decided to implement a number of progressively more complex algorithms. We figured the best way to evaluate different algorithms would be for them to play against each other.

We began with a random player, then moved on to an evolutionary greedy player (using a more sophisticated evaluation function). After that, we developed a minimax algorithm, and later added alpha-beta pruning for increased efficiency. Furthermore, we implemented a Monte Carlo tree search algorithm. Finally, we created a neural network that uses Q-learning.

Section 2: The Model Module

We developed a game model module to capture the essence of an Othello game state and develop a fully functional interactive program. The model was written in object-oriented Python; two classes encapsulate the game environment: Square and Board. The square class has one field which holds one of three values (_WHITE / _BLACK / _EMPTY) to express its occupant tile. The board class holds an 8x8 array of square objects as well as the tile counts of each player, each player's possible moves, and the winner of the game if there is one at the given state. Additional components of the model include methods on the board class to update board values, parsing functions for human input in the program REPL, and functions to search the board for possible moves.

Section 3: The Random Player

The first step in developing an AI was to write a module that plays a random move from its set of possible moves. This player, henceforth referred to as Rand, was planned to be the first opponent against which our greedy AI could play to develop a sensible evaluation function. Taking advantage of Rand's infinitesimal computation requirements, we were able to simulate 100,000 games Rand vs Rand to get some sense of Othello's latent properties. These simulations showed that white wins 50.53%, black wins 45.30%, and a tie occurs in 4.17% of games. In addition, the average branching factor is 8.34 for white and 8.32 for black.

These numbers indicate an underlying advantage for white. White's higher winning frequency can be explained by parity: in the general case of random play it is unlikely that either player will be forced into a situation where they cannot play — unless the board is filled. Thus, with an even number of squares to occupy, the player who goes second (always white) will tend to place the final tile and swing the board score to their favor. However, parity does not explain

white's slight advantage in mobility, its tendency to have more moves to choose from at any given state.

Section 4: Evaluation

The basis of each of our first AI players was an evaluation function considering six features of a game state. These features were chosen based on well known Othello strategies found in online guides. The features are, in no particular order:

1. Corner Adjacency — squares adjacent to the corner are disadvantageous to occupy
2. Points — having more tiles than one's opponent is advantageous
3. Corner Occupancy — corner squares are advantageous to occupy
4. Mobility — having more move options than one's opponent is advantageous
5. Frontier — having tiles which are adjacent to empty squares is disadvantageous
6. Stability — having tiles which occupy stable squares is advantageous[1]

	a	b	c	d	e	f	g	h	
1	20	-3	11	8	8	11	-3	20	1
2	-3	-7	-4	1	1	-4	-7	-3	2
3	11	-4	2	2	2	2	-4	11	3
4	8	1	2	-3	-3	2	1	8	4
5	8	1	2	-3	-3	2	1	8	5
6	11	-4	2	2	2	2	-4	11	6
7	-3	-7	-4	1	1	-4	-7	-3	7
8	20	-3	11	8	8	11	-3	20	8
	a	b	c	d	e	f	g	h	

FIGURE 1. MAP OF STABILITY VALUES

[1] A tile in a square S_{ij} can be flanked by a number N_{ij} of squares should an opponent's tile be placed there. Squares associated with low N values are considered more stable, where corner squares have N = 0 and are the most stable. Actual stability values are functions of N and other values.

The evaluation function returns the sum of each of these features multiplied by a weight. Each of the AI agents which use this evaluation function has its own optimal set of weights. We opted to use an evolutionary algorithm to find these particular sets of weights. Taking a list of n sets of weights and a list of m AI agents as inputs, the algorithm simulates all ${}_{n*m}C_2$ combinations of games in parallel over four processes and determines the best weight set of those inputted for each type of agent. It maintains a score for each weight set associated with an agent type, crossing in addition to randomly mutating the best sets to be passed into recursive calls. This algorithm runs indefinitely, printing to console the best current weights for each agent as they are found. We ran this for ~12 hours for each AI agent that uses the eval function. The results are as follows:

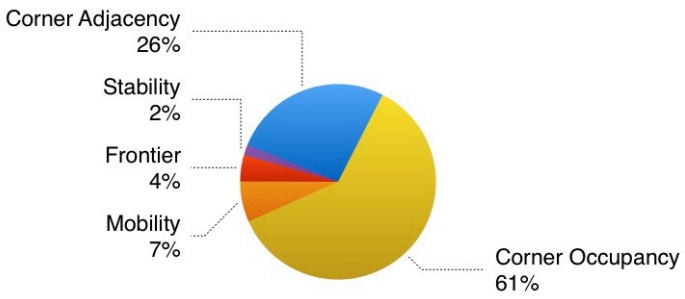

FIGURE 2A. OPTIMAL WEIGHT PROPORTIONS FOR GREEDY

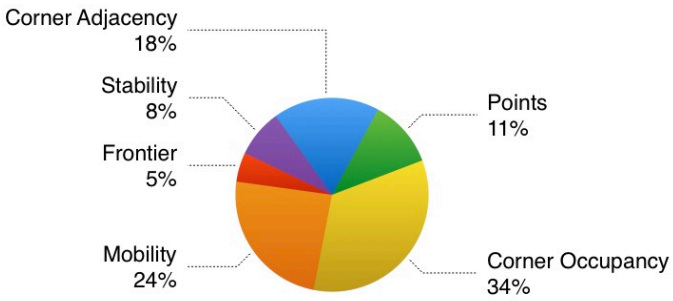

FIGURE 2B. OPTIMAL WEIGHT PROPORTIONS FOR MINIMAX

Figures 2A and 2B show how Greedy and Minimax operate optimally with very different weight sets. It makes intuitive sense that Greedy plays best with a simple strategy: it focuses primarily on getting to the corners and avoiding the squares which may give its opponent the corners. It even neglects to consider its score on the board. Minimax on the other hand takes a more balanced view of its considerations: corner occupancy is still most highly valued, but mobility plays a much larger role than for Greedy. This is likely due to Minimax finding combinations of moves which force its opponent to skip a turn, an extremely powerful tactic in Othello. The frontier and stability features are very low-valued for both agents, but neither of the features were eliminated in the evolution process which suggests that they are indeed important tipping factors under some circumstances.

Section 5: The Greedy Module

The Greedy AI module is simple. Choosing a move is a matter of running the evaluation function on all possible moves and choosing the move which results in the highest scoring state. Greedy performs well against Rand; in 1,000 simulations Greedy wins 984 with an average branching factor of 9.43 and Rand wins 16 with an average branching factor of 4.01. This is ok, but performing worse than a random player 1.6% of the time leaves something to be desired. Thus we moved forward to more advanced AI techniques.

Section 6: Minimax

Developing the Minimax AI module was not very challenging — it was simply a matter of augmenting the Greedy module to follow the algorithm provided in lecture. The first implementation was a simple depth bounded minimax algorithm. Given the fairly high branching factor of Othello and a computationally expensive evaluation function, Minimax required a prohibitively long time to compute individual moves at a bounded depth of 5. This made testing

difficult and inefficient, so Minimax was run at a maximum depth of 3 levels. Despite the shallowness of the search, Minimax outperformed Greedy completely, winning all 100 of 100 games with a branching factor of 8.43 while Greedy had a branching factor of 3.76. When run against Rand, Minimax won 100 of 100 games, with a branching factor of 9.58 to Rand's branching factor of 3.67.

To optimize the time complexity of Minimax, we implemented Alpha-Beta pruning. This made it possible to perform tests at a bounded depth of 5 levels. At a depth of 5 levels, Minimax again wins all 100 of 100 games against Greedy, with a slightly improved branching factor of 8.74. To further evaluate our Minimax AI, we played it against 5 online Othello AIs[2] at their hardest settings. Not a scientific finding as these AIs' playing algorithms are unknown, but our minimax at depth 5 beat each one.

Section 7: The Monte Carlo Tree Search

In addition to developing a minimax search function with alpha-beta pruning, we decided to develop a Monte Carlo tree search algorithm. The Monte Carlo algorithm is particularly interesting due to the incorporation of randomness and the lack of a real evaluation function—its choosing of moves simply by trial and error made it an interesting competitor against our minimax algorithm. On a higher level, a Monte Carlo algorithm consists of four steps: selection, expansion, simulation, and backpropagation. At the selection stage, the algorithm starts at a root *R* and selects child nodes all the way to a leaf node *L*, a process which is explained in more detail below. During expansion, the algorithm selects an arbitrary child node *C* from *L* (assuming that *L* was not an endgame state). At simulation, the algorithm performs a random playout from *C*, all

[2] http://www.othelloonline.org, http://hewgill.com/othello/, http://www.web-games-online.com/reversi/, http://www.archimedes-lab.org/game_othello/othello.html, http://othellogame.net

the way to an endgame state. Finally, during backpropagation, the algorithm uses the win/loss information of the random playout to update the information of all the nodes on the path from *C* to *R*. This four-step process is run repeatedly until a timer (set by the user) runs out. At this point, the child node of *R* that has the best win/play ratio is chosen as the move to play.

An interesting choice the algorithm must make is that between exploration and exploitation—how should we choose between a state with a promising win/play ratio and one that hasn't been played much? The following formula, known as UCT (Upper Confidence Bound 1 applied to trees), aids with the decision process:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n\_i}}$$

Here, w_i denotes the number of wins after the *i*-th move; n_i denotes the number of simulations after the *i*-th move; *c* is a constant known as the "exploration parameter" (which is theoretically $\sqrt{2}$, but in practice is chosen empirically); and *t* is the total number of plays for the parent node in question. The first quotient in the expression represents exploitation—this quotient is high for nodes with good win/play ratios. The second half of the expression represents exploration—it is high for nodes with few total plays in comparison to other child nodes. After calculating the UCT value for all child nodes of node *N,* we choose the node which yields the maximum value.

Although the Monte Carlo algorithm is relatively straightforward, implementing it in Othello presented several unique challenges. Namely, we had to make use of our earlier random player implementation for the simulation part of the algorithm. Further, we had to determine a way to account for the fact that Othello is a two-player game. We decided that each Monte Carlo tree would only record wins from the perspective of the root node's player—to do otherwise would make for an almost certainly more confusing implementation.

Prior literature on the algorithm has proven that Monte Carlo converges to minimax, albeit very slowly[3]. Intuitively, this makes sense—with infinite simulations, Monte Carlo should be able to tell us exactly the value of every state. However, this carries the implicit assumption that we have a perfect minimax algorithm. In practice, this is not the case—our minimax algorithm, for example, does not have a perfect evaluation function, and given our finite computing power, can only play at depth 4 if we want to play games at a reasonable pace. Therefore, our imperfect minimax player makes for an interesting opponent against our Monte Carlo player.

Much to our surprise, our Monte Carlo player (with its simulation time set to 10 seconds per move) beat the minimax player (at depth 4) every single time. Upon further review, however, we can rationalize this; firstly, as mentioned previously, the minimax player has an imperfect evaluation function. Secondly, and perhaps much more importantly, the minimax player is never able to see past depth 4, whereas the Monte Carlo player simulates many games to completion each time it makes a move. Such differences in strategy could explain the vast differences in performance between the two.

Section 8: The Neural Network

The interesting difference between minimax and Monte Carlo is that the latter learns as it plays. However, a problem with the Monte Carlo algorithm is that it forgets everything after each move, let alone a whole game. Hence, we found the idea of developing an algorithm that would improve after each game to be a particularly appealing concept.

[3] http://ewh.ieee.org/cmte/cis/mtsc/ieeecis/tutorial2007/Bruno_Bouzy_2007.pdf

A neural network satisfies these demands rather well. On a higher level, a neural net takes in a vector input, and through matrix multiplication with a "hidden layer," outputs a vector or a value. This output is then evaluated using a loss function.

Developing a sophisticated neural net for Othello presents several challenges. The first challenge we had to consider was simply the dimensions of the neural net. Based on past literature, we decided to go with a 1-by-64 vector input, which represented a game state (with -1 as a white tile, 1 as a black tile, and 0 as an empty tile). Naturally, we also had to consider the number of hidden layers and their dimensions. Based on accepted conventions in various literature[4], we decided on a hidden layer of size 42 (typically, hidden layers are 2/3rds the size of the input layer). The greatest challenge in determining the size of the neural net was the size of the output layer. Eventually, we settled on a simple evaluator. The output was simply a single digit between -1 and 1. This served as an evaluation of the input board state. A number closer to -1 would represent a board state more favorable for white, whereas a number closer to 1 would represent a board state more favorable for black.

Looking at this from a higher level, the purpose of such an evaluation function becomes more apparent. Given a state and a color to move, we begin with some preprocessing where we determine the possible moves for that color. Then, those states enter the neural net, and each one is evaluated. The state that returns the most favorable evaluation for the color in question is chosen as the move to play.

[4] http://www.ai.rug.nl/~mwiering/GROUP/ARTICLES/paper-othello.pdf, http://i.cs.hku.hk/fyp/2016/fyp16038/assets/docs/Interim%20Report.pdf

We eventually settled on using Q-learning. Q-learning is a particularly fitting strategy for zero-sum, delayed-reward games such as Othello. Q-learning relies on the calculation of Q-values, which take in state-action pairs and return their values.

$$Q(s_t, a_t) = E[r_t] + \gamma \sum_{s_{t+1}} T(s_t, a_t, s_{t+1}) \max_a Q(s_{t+1}, a)$$

To clarify, s_{t+1} is the state the player is in after the opponent has moved; E is the expectancy operator; s_t denotes the state at time t; and a denotes the action taken. We then make use of the following temporal difference algorithm:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - \hat{Q}(s_t, a_t))$$

Here, α represents the learning rate. This equation reduces the difference between the current Q-value and the backed-up estimate. Finally, we can use these values to select the best possible move with the following policy:

$$\pi(s) = \arg\max_a Q(s, a)$$

Finally, we made use of an *ε-greedy exploration*—this means there is a ε probability that the agent behaves randomly instead of using the policy. To make for a smooth implementation, we decided to use Keras and TensorFlow—developing a neural net of this level of complexity without a library would be an overwhelmingly complicated task. Following the successes of past literature[5], we set our discount factor to 1.0 and our ε to 0.1 (which linearly decreases to 0 as training progresses). We trained the neural net against both the minimax algorithm (at depth 4) and the Monte Carlo algorithm (at max time set to 10 seconds). As expected, the neural net began by performing very poorly against both algorithms, but eventually reached a success rate

[5] http://www.ai.rug.nl/~mwiering/GROUP/ARTICLES/paper-othello.pdf

of 0.788 against minimax and 0.685 against Monte Carlo. Although we would have liked for these numbers to have been higher, they can perhaps best be explained by the simplicity of our neural net. By using only one hidden layer, we may have had an overly simplistic model on our hands. Furthermore, given more time, it would have been useful to train the model against more types of players. This could give us higher success rates than the ones we experienced. It is also possible that our ε value was too low, and that in order to develop better strategies, we should have incorporated more randomness. Lastly, an ideal approach would be to train the model for a longer period of time. Given our time frame, we were unable to do as many simulations as we would have liked to.

Section 9: Conclusions

At first, the goal of this project was simply to develop a fairly sophisticated Othello AI. Over time, however, the project evolved into an exploration of an array of different, competing strategies to find the best possible approach to playing Othello. Given the constraints we placed on our minimax algorithm (i.e., limiting it to depth 4) and on our Monte Carlo tree search (i.e., limiting our maximum simulation time to 10 seconds) the neural net emerged as the winner. Given the neural net's complexity and its ability to learn from thousands of games or more, these results perhaps do not come as a surprise. However, research on Othello has shown greater success using CNN's and Deep Q-learning; given more time, such methods would have been useful and likely would have yielded a greater success rate against minimax and Monte Carlo.