AN INTELLIGENT OTHELLO PLAYER COMBINING MACHINE LEARNING AND GAME SPECIFIC HEURISTICS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Systems Science

in

The Interdepartmental Program in Systems Science

by
Kevin Anthony Cherry
B.S., Louisiana State University, 2008
May 2011

Table of Contents

Abstract

Artificial intelligence applications in board games have been around as early as the 1950's, and computer programs have been developed for games including Checkers, Chess, and Go with varying results. Although general game-tree search algorithms have been designed to work on games meeting certain requirements (e.g. zero-sum, two-player, perfect or imperfect information, etc.), the best results, however, come from combining these with specific knowledge of game strategies.

In this MS thesis, we present an intelligent Othello game player that combines game-specific heuristics with machine learning techniques in move selection. Five game specific heuristics, namely corner detection, killer move detection, blocking, blacklisting, and pattern recognition have been proposed. Some of these heuristics can be generalized to fit other games by removing the Othello specific components and replacing them with specific knowledge of the target game. For machine learning techniques, the normal Minimax algorithm along with a custom variation is used as a base. Genetic algorithms and neural networks are applied to learn the static evaluation function. The five game specific techniques (or a subset of) are to be executed first and if no move is found, Minimax game tree search is performed. All techniques and several subsets of them have been tested against three deterministic agents, one non-deterministic agent, and three human players of varying skill levels. The results show that the combined Othello player performs better in general. We present the study results on the basis of four main metrics: performance (percentage of games won), speed, predictability of opponent, and usage situation.

Chapter 1 - Introduction

1.1 Introduction

Artificial intelligence is a topic that can be found in multiple fields of study. It can be found in spoken language recognition [1], autonomous vehicle systems [2, 3], and even in the armed forces for training and other non-combative roles [4]. This thesis will explore its affects in the two-player, perfect information, zero-sum game called “Othello”.

1.2 Othello

Inspired by the Shakespearean play of the same name, Othello was first created around 1883 and was first introduced in American culture around 1975 after the rules were changed to what we know of them today [5]. The game’s slogan, “A minute to learn... a lifetime to master!” [6] explains why it can be problematic to attack from an artificial intelligence perspective since although the rules are simple, there are many strategies to consider. This thesis will present several techniques for accomplishing such a task and explain the relative merits of each by examining their aptitude when pitted against other artificial intelligence agents and human players.

The game is played on an 8x8 grid and the player with the most pieces in the end wins. A valid move is any piece placed on the grid that will cause one or more opponent pieces to be surrounded either vertically, horizontally, or diagonally by the player’s pieces already on the board. After the move, all opponent pieces surrounded because of the newly placed piece are converted to the player’s pieces. When the game starts, two white and two black pieces are placed in the center of the board (figure 1.1 part A).

The black player always goes first. His valid moves are shown in part B below and are the result of his already placed pieces at board locations (3, 3) and (4, 4).

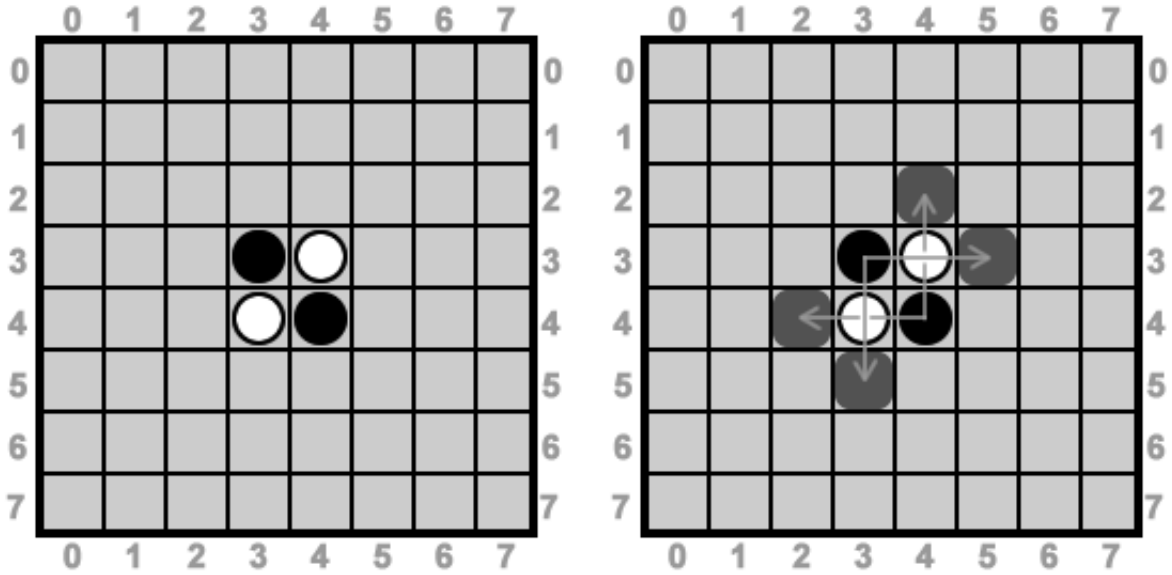


Figure 1.1

A) Initial board state. B) Valid moves for black player. If the black player places one of his pieces at location (2, 4), his opponent's piece at (3, 4) will be surrounded vertically from this newly placed piece and black's piece at (4, 4), therefore this is a valid move.

If this player were to place his piece at location (5, 3), his opponent's piece at (4, 3) will be surrounded vertically. This white piece will then be flipped over to a black piece and it will become the white player's turn (shown in Figure 1.2 part A). The white player will then have to choose from his set of valid moves shown in part B below. This is repeated until either the entire grid has been filled or either player has all his pieces flipped and therefore has no pieces left on the board. As mentioned the player with the most pieces at the end of the game wins. If during the game, a player does not have any valid moves on his turn, his turn is skipped. His opponent is then allowed to continue playing until the player has a valid move.

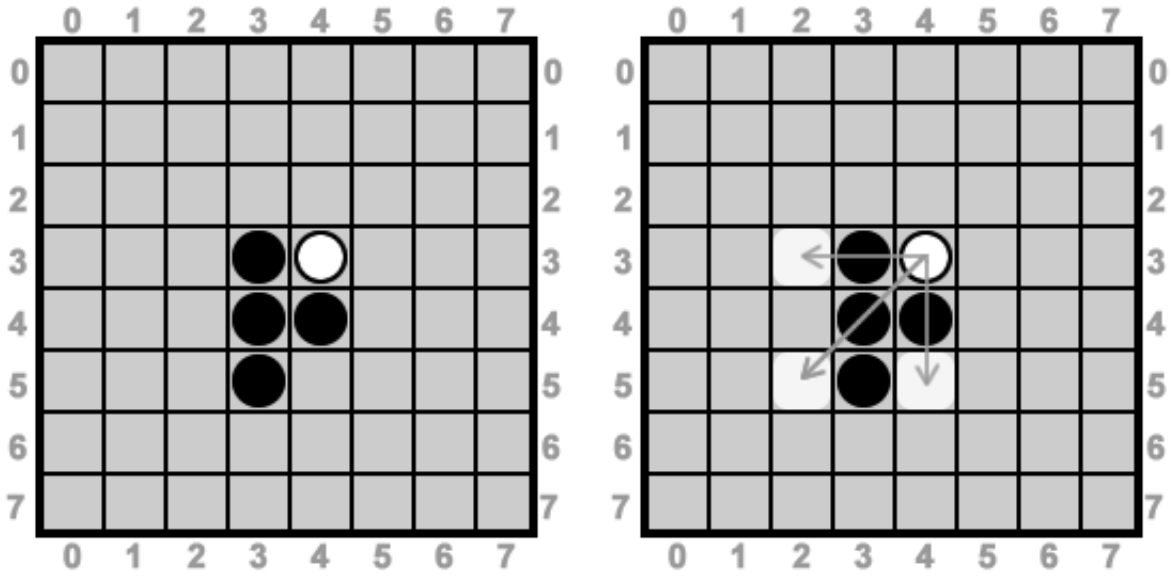


Figure 1.2

A) Board state after black moves to (5, 3). B) Valid moves for white player.

Othello is also known by the name of Reversi, however there are slight differences to the rules. The main one being that in Reversi, the board starts out empty, and players alternate turns to fill the center four locations [7].

Chapter 2 – Common Methods

2.1 Introduction

Before more detail can be stated about the approach and design, brief explanations on the algorithms used will be given.

2.2 Minimax

Minimax is an exhaustive search approach to finding an ideal move among the valid choices [8]. For every one of a player's valid moves, the opponent's valid move list is evaluated. The player's valid move list in response to every one of its opponent's moves is then evaluated, and so on constructing a tree of board states. The player creating this structure is known as the "max" player and the opponent is "min". To build the initial tree, the current board state is considered the root. Each of the player's initial valid moves become a child of that root, then each of the opponent's moves in response to the player's moves become children of that node, and so on. The tree construction stops when a certain depth has been reached (which is open for the implementer to decide). This basic structure is shown in figure 2.1. Each of the leaf nodes represents a possible board state that is the result of its parent's move, which is one of the results of its own parent's move, and so on. These leaf nodes get their value from a static evaluation function. This function takes in features of a board state and assigns a real value indicating how good that state is for each player. If this value is low (typically in the negative range), the state is more ideal for the min player; a high value (typically in the positive range) is more ideal for the max player; and a value close to zero (or in the middle of the function's range) represents a more neural board state. After assigning

values to each leaf node by running their represented board states through this evaluation function, these values must be propagated upwards.

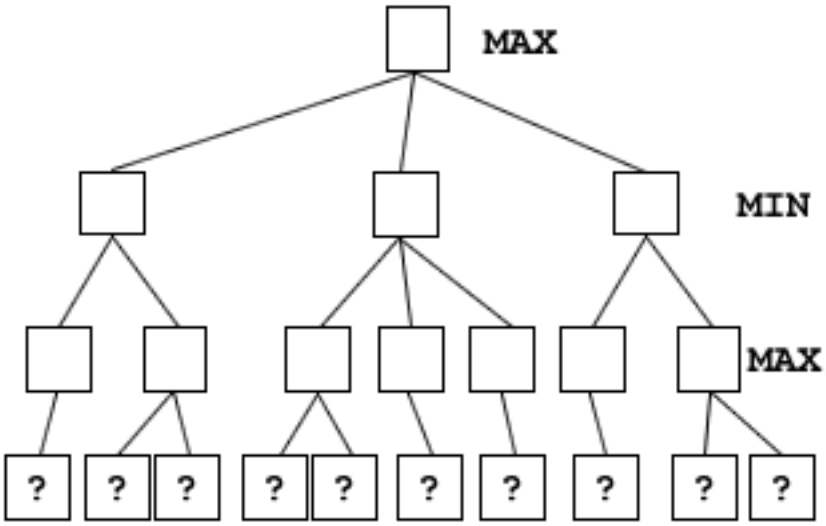


Figure 2.1

Minimax tree structure for a depth of 3

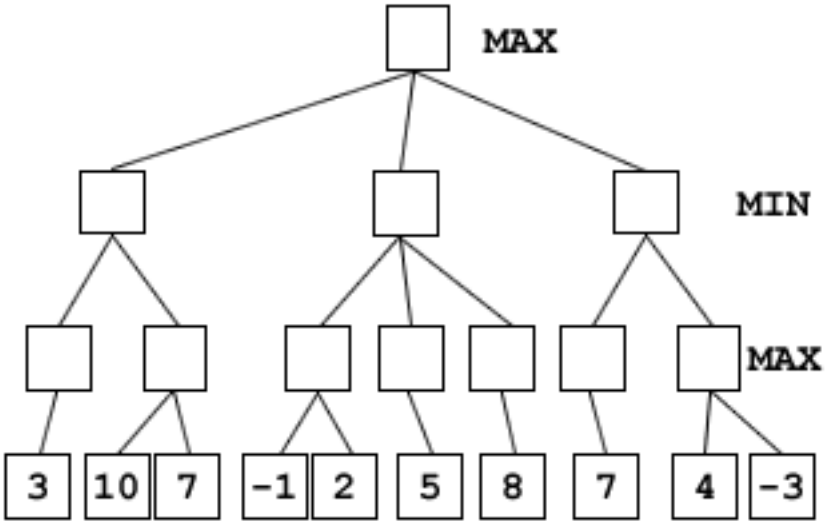


Figure 2.2

Static evaluation function has been used on the leaf nodes to calculate a value for the board state they represent

In figure 2.2 the level right above the leaf nodes is a max level, meaning that each of these nodes chose the maximum value from their children (shown in figure 2.3). The next level represents min's move and so the minimal value of each child node is chosen (figure 2.4). This happens because at each max node, the children represent possible moves for the max player and their values indicate how good the resulting board state will be for this player. Therefore the max player would want to take the path that maximizes this value. Min's children represent possible moves for the min player to take and therefore the minimal value is chosen as it is more ideal for this player. The final level is always max since we are evaluating a move for the max player. After this root node gets a value, the path becomes clear (figure 2.5). This path represents a board configuration after a certain number of moves that is the result of the max and min player playing optimally.

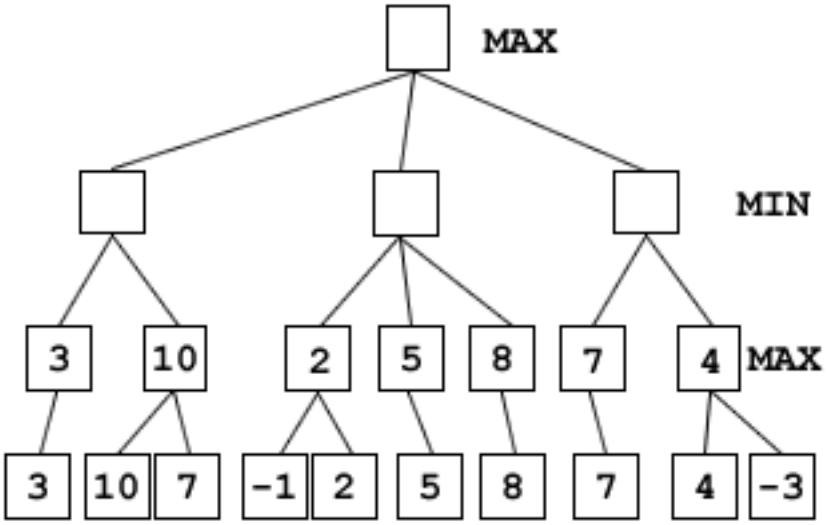


Figure 2.3

Values have been propagated up from the leaf nodes. Since it is at a max level, each parent takes the maximum node value of each of their children.

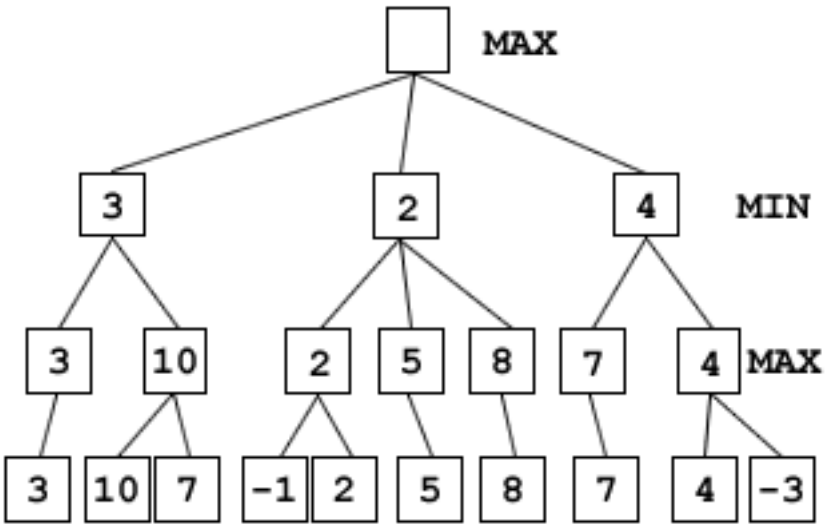


Figure 2.4

Values have been propagated up to the min level. Min parent nodes take the minimal value of each of their child nodes.

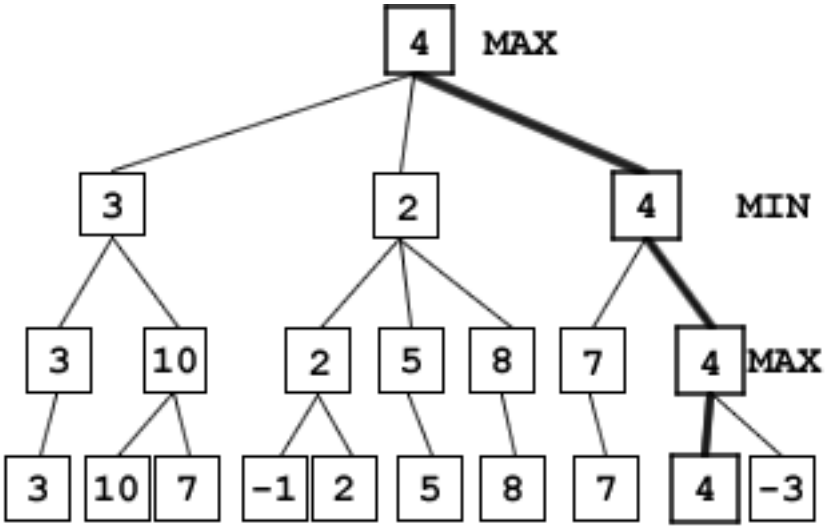


Figure 2.5

The highlighted lines show the path from the root that leads to a board state 3 moves away. If the max player picks the best move during his turn (the move/child node with the maximum value) and the min player does the same (picking the move/child node with the minimum value), then this represents the resulting board state's evaluated value.

2.2.1 Minimax Optimizations

Normally with the Minimax algorithm, the greater the depth before the static evaluation function is applied, the more accurate the result. This comes at the expense of time, however, as the number of nodes grow exponentially resulting in an estimated total number of b^d, where d is the maximum depth and b is the "branching factor" or average number of children for each node [9]. It is this reason that so many different optimizations were created and can be combined to allow one to search deeper without performing unnecessary computations.

Alpha-beta is among the most common of these optimizations [10]. The idea is simple - don't expand (i.e. create children) nodes that can't possibly change the final decision.

2.3 Genetic Algorithms

Genetic algorithms are based on the notion of survival of the fittest [11]. If we take several solutions to our problem, evaluate their accuracy or fitness using some measurement, use the best of them to create a new set of solutions, and repeat this until some stopping criteria, the hope is that our latest set of solutions will be much better than the ones we started with.

To be more explicit, the set of solutions is called a population and the individual solutions are called chromosomes. Methods such as crossover and selection are used to create a new population from the previous one. Crossover takes two "parent" chromosomes and combines them to form two new "child" chromosomes. Selection simply selects the top x percent of the fittest chromosomes and puts them into the new population. Mutation can be applied after, in the hopes of creating a better chromosome, by introducing something into the population that wasn't there before. If

we randomly select a small amount of chromosomes in the new population and make some small random change to each one, we introduce something into the population that might not have otherwise formed. We can repeat this procedure, creating more and more populations until either the average fitness (judged by some evaluation function) over all chromosomes or the fittest chromosome in a population reaches a specified threshold. When this happens the fittest chromosome in this population holds the best solution.

2.4 Neural Networks

Modeled after human brain activity, neural networks consist of multiple discriminatory functions contained in what are known as “perceptrons” [12]. Each perceptron takes input from multiple sources and produces a single output. Each input it receives is multiplied by its own weight value and the sum of these inputs form the final input that is given to an “activation function.” The result of this function is the output of the perceptron. If these perceptrons are chained together, making the output of one become part of the input of another, they form a neural network. Since perceptrons can receive multiple input values, many perceptrons can feed their output into the next perceptron, meaning it is possible to form “layers” of perceptrons that all calculate their value simultaneously so they can give their output to each of the perceptrons in the next layer. This hierarchical design allows for a final, arbitrarily complex decision boundary/boundaries to form, giving neural networks their incredible flexibility and power. The number of layers and perceptrons at each layer as well as the activation function within the perceptrons are free parameters and are up to the implementer to decide. Once a topology and function are chosen, the weights at each edge connecting perceptrons of different layers must be learned to increase the accuracy of the final

outcome. This is normally done using sets of target values corresponding to sets of input values, i.e. a training set. Common algorithms for training these networks include backpropagation and feedforward. If no known target values exist, “unsupervised” learning must occur in which the network attempts to adapt to its environment as much as possible and sometimes seeks to maximize a certain reward. This is significantly more challenging than supervised learning, however many real-world problems require unsupervised learning as quantifiable ideal outcomes are difficult to predict/calculate.

2.5 Pattern Detection

In order to create the best agent for a particular game, one must find game-specific information to exploit. General methods will work well for most cases, but can only go so far. Past this point specific knowledge of good plays in the game as well as subtle tricks and techniques known to experienced players must be mimicked by the agent. Since there can be literally billions of possible board states in a game (there are 3^{64} states in Othello), trying to recognize and take action on specific states is futile. A better approach is to recognize board patterns that could manifest themselves in several different board states and to have an ideal move ready when this pattern is matched. Then one can create a collection of patterns and ideal moves for each.

2.6 Related Works

Now that the basic algorithms have been examined, we will look at some applications that show successful implementations.

MOUSE (MOnte Carlo learning Using heuriStic Error reduction) is an agent built for Othello [13]. The paper explains that the main problem with using reinforcement learning for function approximation is in its inability for good generalization. To solve this

MOUSE uses reinforcement learning along with past experience. It's main decision making progress uses a series of 46 board patterns, each with its own weight value, formed from reflections and rotations of eleven unique cases. When handed a valid move, all patterns are checked and a value is produced from the sum of the weights of those that match. This sum represents an estimate of the "disc differential", or the difference between the two player's pieces on the board at the end of the game. Supervised learning was used with training examples coming from games played by at least one good player. After training and after several adjustments were made, MOUSE became good enough to compete in a GGS Othello tournament, which holds the world's best human and artificial players.

Another example of successful artificial intelligence implementations for well-known board games comes from Gerry Tesauro [14]. Using temporal-difference learning [15], Tesauro developed TD-Gammon, an expert player for the game of Backgammon. Since this game has a branching factor of 400, searching to even a modest depth becomes impractical. Therefore instead of relying on a Minimax approach, TD-Gammon uses Neural Networks only on the current list of valid moves. This is performed in an interesting fashion as no patterns or special features are actually extracted from the board to be sent to the network, but instead the entire board is encoded in 198 input nodes. The first 192 come from the fact that there are 24 valid locations on the board, and the number of pieces the white or black player has at any one location is encoded in four input features. Therefore 24 locations with each location having four input features for white and four for black, gives an initial 192 features. Two more (one each for white and black players) were used to represent the number of pieces found on the "bar", two for those removed from the board, and two for a bit

masked representation of whose turn it was (e.g. 01 for white and 10 for black). All feature values were scaled to a range of approximately zero to one. Online gradient descent backpropagation was used for training and after about 300,000 games played against itself, the system developed knowledge rivaling that of the best Backgammon programs at the time. Note that the previous best agents relied upon deep knowledge of the game, including one created by Tesauro himself. Without this knowledge the TD approach was still able to produce similar results.

Chapter 3 – Getting Started and Using Game-Specific Heuristics

3.1 Study Common Methods – In General and Game Specific

The approach was to study many common techniques in board game artificial intelligence and see how each was used. It was also important to see some creative game-specific solutions for inspiration on developing custom techniques. After doing so as practice an agent was created for a very simple game. Simple heuristics were used with a mix of offensive and defensive approaches.

3.2 Simple Agent for Game: TIC-TAC-TOE

The game of Tic-Tac-Toe was simple enough to serve as practice creating an agent. The concept of bit boards, that is, bit strings that represent different aspects of the current game board, was explored [16]. Four agents were created each with their own method for finding an ideal move. The simple agent scanned the board left to right, top to bottom to find the first available spot to take. The influence map agent uses an influence map, discussed later, to pick its move. The two other agents both evaluate each valid spot by examining its influence map value, how many “two-in-a-rows” it will create for both itself and its opponent, and if that spot will cause it to win or will block its opponent from winning. Each of these aspects is given a weight and after adding them all up, the spot with the highest value gets chosen. For the defensive agent, the weights are chosen to give more emphasis on preventing the opponent from winning. The offensive agent has higher weights for moves leading it to victory. Both these agents were able to successfully prevent anyone from winning a single game allowing, at best,

a tie. The same could not be said for the influence map and simple agents, however they were merely tests leading up to the other two agents.

3.3 Combining Common Methods

The next step was to take common methods for two-player, zero-sum games and combine them into one agent. The methods chosen to combine were genetic algorithms, neural networks, and Minimax which will be discussed later. Custom methods were also added. These could be game-specific or game-independent.

3.4 Choosing a Game

A game has to be chosen that is well known to the implementer and due to personal experience, Othello was selected. Throughout many times playing the game, several strategies including what board locations where better than others, which moves were good among the valid choices, which moves could end up tricking the opponent into making a bad decision, and which moves one should never take under certain circumstances were developed.

Knowing the chosen game well, one has an easier time coming up with exploits of specific game features to add to one's agent than would be for other games.

3.5 Exploitation of Game Characteristics

The following are explanations of each game specific technique created as well as the motivation behind them. Although these are mostly only valid for the game of Othello, some may apply to other games if modifications are made. For instance, in the case of pattern detection, any board configuration, regardless of the game, where good moves are well known can be represented as a pattern.

3.5.1 Pattern Detection

In Othello a good player knows that the corners of the board are the best locations and will try his best to capture them. Therefore several patterns were created to enable the agent's next move to be a corner.

3.5.1.1 Theory

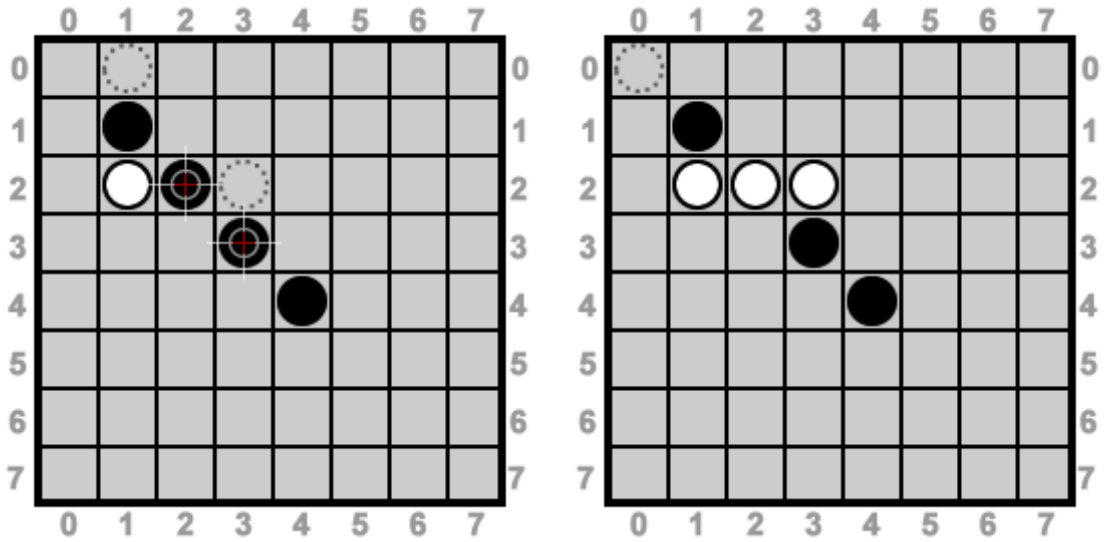


Figure 3.1

The agent is the white player and its opponent is the black player. A) The agent's valid moves are shown as dotted circles. This board configuration is one of the ones that match one of the agent's patterns. The two opponent pieces that have white and red crosshairs on them are possible targets, meaning one of these must be flipped to satisfy the pattern. The pattern does not recognize the opponent piece at (1, 1) since overtaking that spot would give the opponent the corner. Piece (4, 4) is also not a target since the opponent could flip that piece right back over on its next turn due to it having a piece at (3, 3). Since one of the agent's valid moves takes the target opponent at (2, 2), that move is chosen. B) The result of overtaking a target piece. This gives the agent the corner as one of its valid moves for its next turn. Notice that the spot at (2, 2) cannot be taken back by the opponent's next move, as it is protected by the opponent's own pieces.

The theory behind this is that corners are the best locations in the game. If the list of valid moves does not include a corner, we want the agent to be able to set itself up for a corner at a later time. Therefore a collection of patterns was created that would not only attempt to make its next valid move list contain a corner, but would try to guarantee it. This was accomplished by flipping over an opponent's piece that could not be flipped back over during the opponent's next move. That piece would create a diagonal capture line for the agent to one of the corners (figure 3.1 part A, above). Since the piece the agent targeted could not be flipped back over by its opponent, unless the opponent took the corner itself, it is certain that the agent's next valid move list would include that corner (figure 3.1 part B, above).

3.5.1.2 Implementation

The problem with implementing patterns is that they are an abstract concept. They have to be flexible enough to represent multiple (perhaps hundreds) of different concrete board states. If a pattern only represents a single board state, the chances of that state appearing in a given game are extremely small and as such that pattern would be useless in practice. The original concept of a pattern was an xml file that used bit masking to represent a location's status. 1 was untaken, 2 was taken by the agent, and 4 was taken by the opponent. If a board location's status was of no concern, it was given a 7, which is the result of 1 + 2 + 4, or in other words, either untaken, taken by agent, or taken by opponent (the only three possible choices). 0 meant that its true value was inherited by a template. Templates were simply xml files used to store reoccurring values in different patterns so those values could easily be referred to in the specific pattern used.

The xml file would contain a list of rows, cols, and bit masked states at each. For a pattern to match, each location, described by its row and col, would have to have a matching state to that of the actual board. So if a location had a state of 3, the location on the current board state would have to be either untaken or taken by the agent for the pattern to match that location. If all location states match for the current board state, that pattern is considered matched. To save computation time and pattern file complexity and length, any location not explicitly stated was considered a “don’t care” or having a state of 7 and was not checked when the pattern was being evaluated.

This initial approach worked well for matching patterns, however there had to be some way of representing an ideal move if the pattern was matched. So to state the best move(s) for each pattern, a separate collection was used that stated the row and column of each move to try in the order they are specified (since some moves might be invalid). This didn’t work too well, though, as there could be several good locations given similar board configurations and only differing in a couple locations. In fact most of the time a good move was not a specific location, but was a specific opponent piece that needed to be flipped. Since pieces can be flipped from at most 8 directions and up to 7 spots away, this means there could be several ideal moves that all target that specific piece.

The original approach to handling this was to create a string of conditions that must be met in order for a move to be selected. If a pattern was matched, the list of best moves each with its own conditional was checked. If any conditional statement evaluated to true, that move would be chosen. Due to the number of possible situations a board could be in, these conditionals grew very complex. An example of one conditional could be: 542 & (242 | (241 & 643)) & !(434 ^ 454) & !(354 ^ 534). Each

three tuple consisted of row, col, and state. So 542 meant that the location at row 5, col 4 had to have a state of 2 for this to be true. &, |, and ^ were the bitwise and, or, and xor operations, respectively. ! was negation, and parentheses were used for grouping. This was very complex to figure out by hand for each location that could be used to attack a certain opponent piece as well as computationally expensive to parse. The conditionals were being used to ensure that a location was a valid move for the agent, something that probably should be decided in the game and not explicitly specified by the pattern.

A more abstract way of taking over a location was created. Instead of specifying the exact location of an ideal move, the pattern would specify which location it wanted to overtake. If this location was empty, the agent would simply place a piece there. If the location was taken by its opponent, it would look through its list of valid moves and choose one that would flip over that piece, thereby taking the location over. If no valid move could accomplish this, the pattern would be considered unmatched and the next pattern was evaluated. This allowed for the game itself to decide how to overtake a location using the list of valid moves. Now a pattern need only declare the target location and let the game (with its knowledge that the pattern doesn't possess) decide how. Several target locations could also be specified in order of precedence with the first location that could be overtaken picked. This allowed for a single pattern to state multiple ideal moves in a simple manner and allow the game to decide which, if any, it could take. This made patterns much more dynamic and easier to write as well as having a greater chance that they would be used in a given game.

3.5.2 Corner Detection

Since we have patterns that will set the agent up to be able to take a corner location on its next turn, we need to make sure the agent would do just that. Corner

detection is therefore used to force the agent to take a corner anytime it is in the valid move list.

The main reason for this is so the agent doesn't pass up an opportunity to take a corner and have its opponent block that opportunity for its next turn, or have its opponent take the corner instead.

3.5.3 Killer Move Detection

Since it is possible to win the game early, checking for moves that will eliminate all your opponent's pieces is important. This exploit performs an initial check to see if any of the agent's valid moves accomplish this.

The reason for this technique is simple; if the agent can win the game immediately after moving to a certain spot on the board, it should move there every time regardless of the benefits of the other move choices.

3.5.4 Blocking

In Othello if a player does not have any valid moves on his turn, his turn is forfeited and control is returned to his opponent. For the blocking exploit, the agent checks to see if any of its valid moves will cause its opponent to forfeit his turn, thereby allowing the agent to go again. If such moves exist, the agent will arbitrarily take one of them. This action can be repeated as long as there are moves which prohibit its opponent from taking a turn.

3.5.5 Blacklisting

If any of the agents valid moves set up its opponent to make a great move, that move should not be taken. The concept of blacklisting takes those moves and forbad

them from being picked. This technique should not be used too aggressively, however, as sometimes other methods, such as Minimax, will seem to give the opponent the edge, but is actually allowing a great move for the agent possibly several turns later. Therefore this should only be used if the agent's opponent will be able to make a game changing or other type of ideal move in response to the agent's choice. For the experiments discussed later which use this technique, it attempts to ban the agent's opponent from taking a corner. The agent scans through all of its valid moves and any move that allow its opponent to take a corner on his next turn would be blacklisted. If all valid moves ended up being blacklisted, then this was unavoidable and the blacklist was cleared. At this point the agent would just try to pick the best move knowing its opponent will have a chance to take a corner no matter what it did.

3.6 Order of Exploits

If all exploits are active, killer move detection is used first followed by corner detection, blocking, pattern detection, and finally blacklisting.

Chapter 4 – Using Machine Learning Techniques

4.1 Using Machine Learning Techniques

If all game-specific exploits fail to find an ideal move, we fall back onto Minimax. Minimax is guaranteed to choose a move, even if it is not always optimal, and works for any zero-sum game.

4.2 Minimax and the Expected Min Technique

Before Minimax could be implemented, there is a drawback that needs to be addressed, that is its assumption of what move the min player will choose. This is the motivation behind “expected min”. Normal Minimax will choose the child node with the least value for the min parent. This will result in the agent choosing a move under the assumption that its opponent will always choose the best move for him among the valid choices. There are two main problems with this. First, there is no guarantee the min player will always choose this path and second, the “ideal” min move is chosen by a subjective static evaluation function and may not represent the actual best move for the min player or at least what the min player thinks is the best move. So basically the min player must play exactly like the max player for the max player to have an accurate estimate of the min player’s behavior. The expected min technique was therefore created to help account for the uncertainty of the min player and help to lessen the stringent assumptions being made. Instead of choosing the smallest value for the min player, all values are taken into account and are given weights according to how likely the min player is to choose them. The algorithm is as follows:

1. Take all child node values
2. Subtract each value by the maximum of those values plus 1 (e.g. if we have 1, 2, and 3 then produce (1 – 4), (2 – 4), and (3 – 4) to get -3, -2, and -1). The reason for this is due to both the desire to end up with higher weights on lower numbers, and also to allow values of zero to have some contribution to the weight distribution.
3. Sum these new values up and divide each value by that sum (e.g. for the -3, -2, and -1 values from above, we have (-3 / -6), (-2 / -6), (-1 / -6) to get 0.5, 0.333, 0.1667)
4. Multiply the original values by these weights (e.g. our original value 1, 2, and 3 become (1 * 0.5), (2 * 0.333), and (3 * 0.1667) to get 0.5, 0.667, 0.5)
5. Sum these values up to get the min parent's value (e.g. 0.5 + 0.667 + 0.5 = 1.667)

This is in contrast to a value of one that normal Minimax would assign. This new number attempts to give a more accurate value for that parent node since it merely applies more weight to lower values instead of automatically choosing the lowest. Experimental results will be shown later that state how well this performs and conclusions on which situations this technique is best applied will be made.

In the game of Othello, having the most pieces in the beginning or middle game states alone is not a good indication of how well one is doing [17]. This is due to the fact that any one move may flip over several pieces, possibility from several different capture lines, and can change the score dramatically. Therefore heuristics must be developed to decide if a player is truly winning during any game state. This is the purpose of the static evaluation function.

The original equation involved a simple weight vector and four input features: an influence map (discussed later) sum of all board positions held by the agent and opponent and the total number of unique pieces that could be flipped by the agent's and opponent's next set of moves if their move was to immediately follow. These input features were used to decide not only how many pieces a player has, but also a heuristic on how important their locations are and how easily they can be flipped in the next move. This gives some indication on how quickly the game can change in the next few moves and aims to prevent the false security the currently winning player often feels.

The equation started out taking this form:

$$Eval(I_A, I_O, F_A, F_O) = w_0 I_A + w_1 I_O + w_2 F_A + w_3 F_O$$

Where w is the weight vector (discussed later), I_A is the influence map sum for the agent, I_O is the influence map sum for the opponent, F_A is the sum of the unique agent's pieces flipped by the opponent's next valid move set, F_O is the sum of the unique opponent's pieces flipped by the agent's next valid move set. Before the weights could be learned, good influence map values had to be established.

4.2.1 Learning Influence Map for Evaluation Function

An influence map is a matrix that holds a value for each location on the board [18]. This value, from 0 to 10 in this case, indicates how valuable that location is to gaining the upper hand during a game. A corner spot, for example, would be given a value of 10 due to it being the best location on the board. The sides of the board would also be given a high value. The location right next to the corner, however, is probably the worst spot on the board if one does not have the corresponding corner. Taking a

spot right next to a corner drastically increases the chances of your opponent having that corner as one of his valid moves. All these values are generated from knowledge of the game and, as such, can be very subjective.

Multiple intuitive values were tried and tested for performance, but none proved to be very successful. We therefore turn to genetic algorithms to find a more ideal set of values.

4.2.1.1 Using Genetic Algorithms

The original generation contains a population of chromosomes each with their own influence maps created randomly. Since each location can have an integer value from 0 to 10 and since there are 64 board locations, the search space becomes 11^{64}. To reduce the size of this space, we have to observe a few game specifics aspects. First, the corners should have the highest value. There are four corners, therefore that brings the search space down to 11^{60}. Next, the four spots in the center of the board are occupied right when the game starts. This means those locations will never appear in either player's valid move list and so do not need a value. This brings it down to 11^{56}. Finally, the influence map matrix should be symmetric. Each quadrant of the board contains the same values and is just a mirror of one another. This is due to the fact that each quadrant is just as important as the next and only changes due to the locations owned by the players. In fact each of them forms a symmetric matrix of its own. This makes the entire matrix not only symmetric about its main diagonal, but symmetric about its cross diagonal too. This puts our space at a relatively small size of 11^{8} which is 214,358,881. This drastically speeds up the time taken by the genetic algorithm as it only needs to learn 8 values.

	0	1	2	3	4	5	6	7	
0	10	A	C	F	F	C	A	10	**0**
1	A	B	D	G	G	D	B	A	**1**
2	C	D	E	H	H	E	D	C	**2**
3	F	G	H	X	X	H	G	F	**3**
4	F	G	H	X	X	H	G	F	**4**
5	C	D	E	H	H	E	D	C	**5**
6	A	B	D	G	G	D	B	A	**6**
7	10	A	C	F	F	C	A	10	**7**
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	

Figure 4.1

Influence map.
The numbers on the outside in bold are indices, the corners are given the max amount, and the center locations are “don’t care” values as they are taken when the game starts. The quadrants have bold borders around them. A, B, C, D, E, F, G, and H are the values that the genetic algorithm learns. With the highlighting on both diagonals, one can clearly see the extreme symmetry of the matrix.

4.2.1.2 Fitness Function

For the fitness function we have:

$$Fitness(W, W_r, L, L_r, S_w, S_l, N_c) = (W * W_r - L * L_r) + (w_{sw} * S_w - w_{sl} * S_l) + (w_{nc} * N_c)$$

$Where$:

$$W = \begin{cases} 1, & if\ agent\ won \\ 0, & otherwise \end{cases}$$

$$L = \begin{cases} 1, & if\ agent\ lost \\ 0, & otherwise \end{cases}$$

$$W_r = \begin{cases} \min\left({}^{num\ of\ agent\ pieces}/_{num\ of\ opp\ pieces}, 5 \right), & if\ opponent\ has\ pieces\ on\ board \\ 5, & otherwise \end{cases}$$

$$L_r = \begin{cases} \min\left({}^{num\ of\ opp\ pieces}/_{num\ of\ agent\ pieces},\ 5 \right), & if\ agent\ has\ pieces\ on\ board \\ 5, & otherwise \end{cases}$$

$$S_w = \begin{cases} 0, & if\ opponent\ has\ pieces\ on\ board \\ 1, & otherwise \end{cases}$$

$$S_l = \begin{cases} 0, & if\ agent\ has\ pieces\ on\ board \\ 1, & otherwise \end{cases}$$

$$N_c = number\ of\ corners\ held\ by\ agent$$

The parameters w_{sw}, w_{sl}, and w_{nc} are the weights for S_w, S_l, and N_c, respectively and were set at 5, 5, 2, respectively. These values represent heuristic estimates. In situations where multiple games are played, the fitness becomes the average over all games. The range of this function is from -10 to +18 with a winning agent receiving no less than 1.06 and a losing agent receiving no more than 6.94. This overlap between the two scores is due to the addition of the corners taken by the agent as a losing agent could potentially have taken all four corners and a winning agent could have taken none, although these are rare circumstances. Basically if the agent lost, we still want to reward it if it took some corners. If the agent lost but took all four corners, this could be the result of a few bad moves and not the result of an overall bad strategy and therefore the agent should still receive some reward. Ties were considered a loss for both players.

4.2.1.3 Genetic Algorithm Parameters

The population size was set at 21 (a small value so each generation would run quickly and many of them could be produced) with a crossover rate of 0.75 and a mutation rate of 0.05. At a rate of one minus the crossover rate, the chromosomes were selected to move onto the next generation, while the rest underwent the crossover

operation. To be more specific, 5 chromosomes were moved to the next generation while 16 participated in crossover. Wanting most of the chromosomes to undergo crossover, this seemed to be a good balance. A crossover rate of 0.60 was also tested, but 0.75 was found to be a better value. The selection was random with each chromosome's normalized fitness (fitness divided by the total population's fitness) weighing its chances of being picked, so the more fit chromosomes had a better chance. This procedure is called fitness proportionate selection [19]. Single-point crossover was then used were a random value from 1 to 7 (one less than the number of values representing each chromosome's knowledge as stated previously) was chosen as the swapping index. Mutation was then run and caused a single chromosome chosen randomly without weighting to have a random value in its knowledge to be changed to a random number from 0 to 10 (the range of any valid value). Mutation shouldn't be a big factor in the learning process and is why a rate was chosen purposely to allow only one chromosome to be affected.

The simulation ran for 1,000 generations as shown in figure 4.2. The fitness was found by using the above formula and putting an influence agent with the chromosome's knowledge against a greedy agent (explained later). Since they are both are deterministic, each game only needed to be ran once to get an accurate fitness value (since all games would produce the same results) although there were two games per test; one where the influence agent was the white player, and one where it was the black player. The fitness from the two games was averaged. The stopping point was set at a maximum fitness of 18, the highest attainable value, meaning that a chromosome would have to overtake all corners and flip over all of its opponent's pieces. Since it was only tested against a single agent, this was not an impossible task. After reaching this

goal, the knowledge of that chromosome (the fittest) was taken and made into a new target agent. The genetic algorithm was then restarted with a new initial population and all chromosomes would play against this new target agent. The criteria remained the same. This repeated for approximately six times as after that, the fittest chromosome's knowledge was taken and locked in as the final influence map values to be used by any influence map agent and by the main agent. Exactly how many generations each restart took was not recorded; however the true value was between 500 to 1,000. Fortunately each generation only took around 5 seconds.

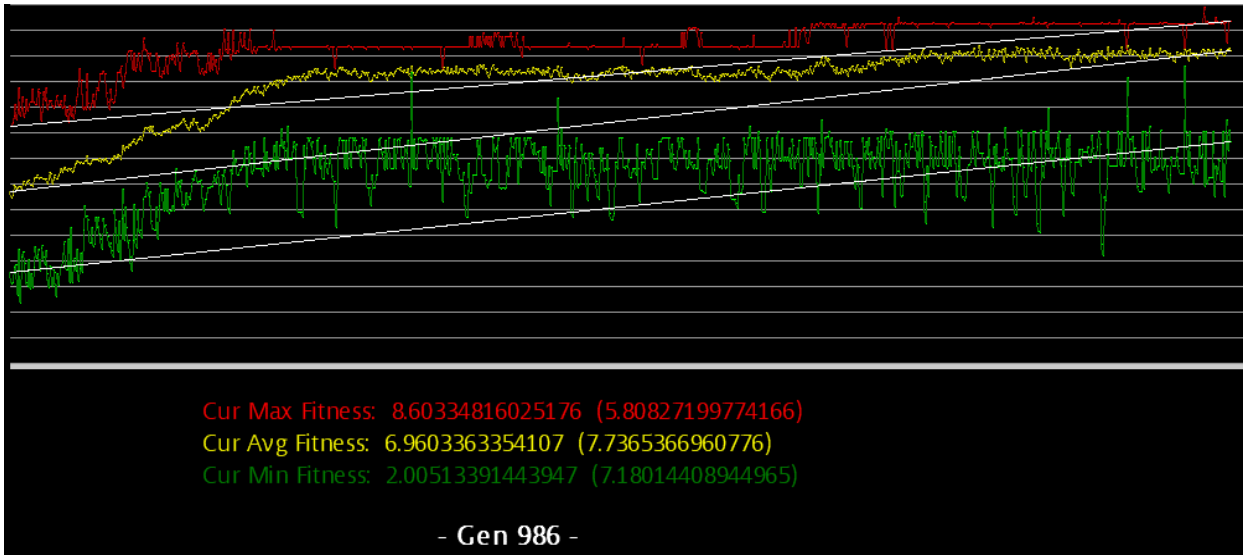


Figure 4.2

Graph of genetic algorithm learning an ideal set of influence map values. "Gen 986" means it is on its 986th generation and the max fitness of that generation is around 8.6, as shown in red. The white line shows the different from the initial generation to the current one. The fitness is found from playing against a greedy agent.

4.2.2 Learning Weights for Evaluation Function

After learning the influence map, we need to learn a good set of weights to for an accurate evaluation of the board. Seeing as how the equation was basically a single perceptron with an activation function of f(x) = y, this could easily be expanded to a

more flexible neural network. Since learning can take a long time and since this is a more powerful approach, we go with the neural network without attempting to learn the weights of the original linear formula.

Four input nodes were used corresponding to the four input features of the formula along with one five-node hidden layer and one output node. Sigmoid was chosen as it is the most common activation function [20] and values below 0.1 were set to zero while values greater than 0.9 were set to one. This range adjustment is due to the fact that total saturation values can only theoretically occur at –infinity and +infinity and so adjustments must be made to treat values closer than a given threshold to the asymptotic boundary as that boundary's value. Since there weren't any training examples, heuristics relating an ideal number of weights to the size of one's training set were not applicable [21]. So without such guidance, the approach was to use a small number of hidden layers and nodes to reduce training time.

4.2.2.1 Why Use Genetic Algorithms?

Since no target values existed to train the neural network with, unsupervised learning algorithms had to be taken into account [22]. The code for training using genetic algorithms was already implemented and was used to successfully learn the influence map values, so it was tried on the neural network weights. It performed well so a different approach didn't seem to be needed. Therefore the unsupervised methods studied were not implemented.

4.2.2.2 Parameters

Since this was to be an influential part of the agent's decision making process and since the search space is larger, we increase the population size over what was

used for the influence map. A population size of 100 was decided upon as crossover stayed at 0.75 since that seemed to do well. Mutation started out low, around 0.01, but later on (as will be discussed) it was slightly raised. Since the chromosomes' knowledge is similar to that of the influence map only with floating point numbers and more of them, the same selection, crossover and mutation operations were used. We initialize all weights to random values from -1 to 1.

4.2.2.2.1 Setup

The final weights must work well against all types of testing agents. It must also work well regardless of whether the agent goes first or second when the game begins. Therefore each chromosome played against each of the three different training agents as the white player and again as the black player for a total of 6 games. The fitness scores over these games were averaged and that became the chromosome's fitness value.

4.2.2.2.2 Addition of Input Features

Initially only a single laptop was used to run the genetic algorithm and was used as often as practical. It ran a total of about 1,500 generations receiving a max fitness of any generation at 10.58 (absolute maximum being at 18) using the same fitness function from learning the influence map. This wasn't too bad but it could have been a lot better. Therefore to improve the accuracy, four more input features were added to the neural network running the static evaluation function: number of corners held by agent and opponent and number of sides held by agent and opponent. Although this meant retraining the network, this time other machines were used. Using a single laptop took too long to be practical to run all those generations again, so instead the program

was executed on four different desktop machines, each more powerful than the original laptop. They ran for an entire weekend and produced the results shown below:

Total number of generations	14,608	11,769	11,808	12,597
Maximum fitness of any generation	8.95	14.21	14.22	11.24

Figure 4.3

GA Learning Results.
These are the results of running the genetic algorithm designed to find the weights of the neural network on four different machines. They were started at slightly different times and hence have different numbers of total generations run (the 14,608 machine started hours before the others). The average running time was around 80 hours. Each machine generated its own set of initial weights.

The maximum fitness attained this time was 14.22; much better than 10.58 found before. It is uncertain whether this is the result of adding more input features or running the algorithm for significantly more generations with four different starting points, but either way this is a more respectable value than before. With this new value, training was stopped.

4.2.2.3 Quicker Training

It is interesting to note that since each chromosome plays six games, each population contains 100 chromosomes, and the total number of generations combined from the four machines was 50,782, the resulting total of games played was 30,469,200 (with each game taking around 38ms)! This was accomplished by using some intuition to speed up Minimax. Since we are only interested in Minimax's ability to accurately estimate the value of any board state, taking it to a depth of one should suffice during training. The only reasons to go deeper than that is to get closer to an end game state,

in which case the estimation might be more accurate, and to help avoid unpredictable and undesirable board states in the next few moves [23]. However, giving the Minimax an end game state would only lessen the need for very accurate weights and undesirable board states should only arise if a less than ideal move is chosen, something the weights should help the agent avoid anyway. Therefore it was only necessary to evaluate each of the current moves and try to make that evaluation as accurate as possible. This decision caused the training to go a lot faster and more generations could be created and examined in a shorter period of time allowing for more instances in the search space to be covered.

4.2.2.4 Plateau Effect

A problem was encountered while training where the maximum fitness of the population would stagnate. Knowing that this most likely represents a local maximum in the search space, the training was stopped, mutation rate was increased to around 0.02, and then training was allowed continue where it left off. Genetic algorithms main strongpoint is its ability to overcome local minima/maxima by using this mutation rate and so by increasing it, this problem could be mitigated.

4.2.2.5 Max Depth

After training a representative depth had to be chosen for testing. Since the agent should be tested under many different settings, a range of depths were chosen instead. The lower bound was one and the upper bound was decided by the average amount of time taken per move. Without any optimizations a depth exceeding four seemed to take too long, however after adding alpha beta (discussed later), that was able to increase to six. Therefore each test done against another computer agent was

repeated for all depths between one and six inclusively. For tests performed with human players, only two depths could be chosen to make the number of tests reasonable for a human to perform. Those depths were one and six, chosen to test the two extremes of the Minimax algorithm.

4.2.2.6 Optimizations

As mentioned above the maximum depth that was practical without optimizations was four. Due to the exhaustive nature of the algorithm, in most situations, basic optimizations must be added to achieve a satisfactory depth. Therefore alpha beta was added.

4.2.2.7.1 Alpha Beta

Alpha beta is among the most common optimization to add to Minimax [10]. Its concept and implementation are simple and its impact can be heavy. Say one has multiple child nodes, one of which has a value of four and the others have not been evaluated yet. Their parent is a min node with a sibling node having a value of five. Since both parents have a parent max node, that node will choose the higher of the two parent nodes and since one of them already have a value of five, the other parent node must have a value greater than that for it to be chosen. Since that parent is a min node and has a child with a value of four, the highest it can be is four, since any child node with a lesser value will be chosen and any with a higher value will be discarded. Therefore since this parent node will never have a value exceeding five, it will never be chosen regardless of the value held by its other child nodes, and therefore those child nodes do not need to be evaluated. These nodes whose evaluations do not affect the selection of the ideal path from the root (or “principle variation” [24]) are considered

pruned and are ignored. It is because of this pruning that alpha beta can search the same depth in less time than regular Minimax.

This brings up one of the shortcomings of the expected min approach talked about earlier. For that approach to be successful, alpha beta cannot be used as each child value is taken into account to produce the final min parent value. Therefore other, non-pruning techniques are the only optimizations allowed. Even though depths past four without alpha beta take a long time, it was allowed to run at depth five for tests involving expected min. At depth six out of memory exceptions occur and so alpha beta had to be used.

4.2.2.7.2 Other Techniques

Due to alpha beta's popularity, there were several other optimizations that were created for Minimax. Only alpha beta was implemented as it is the most common, however other optimizations such as Negascout [25], created in 1983 by Dr. Alexander Reinefeld, have been proven to search as many nodes or less than alpha beta [26]. Negascout uses Negamax [27], a variation on Minimax where each of min's child nodes have their value negated. Picking the minimum from several values is the same as picking the maximum of the negation of those values, therefore each node becomes a max node. Having all max nodes means not having to check whether the min or max of each child value must be taken and can therefore speed up the amount of time spent per node. Negascout is similar to alpha beta however it not only uses this slight improvement, it also can search less nodes under certain circumstances. Since the order of the child nodes for any parent will not affect the final outcome, it can be manipulated. If child nodes can be arranged from best to worst, the principal variation would be the first depth-first path down the tree. This ordering can be approximated by

various heuristics such as results from previous searches and pre-evaluating nodes [25]. In this case alpha beta would prune most of the nodes during search. Negascout, however, uses initial values that form a narrower “window” (narrower than the range from alpha to beta) in an attempt to prune the most nodes. In fact it examines the case where alpha is equal to beta, forming what is known as a “scout window.” With this window size, exploration is much quicker. To see if the current ordering is truly ideal and the first nodes are in the principal variation, Negascout uses this scout window on the other nodes. If it turns out that the ordering is not ideal, Negascout starts over performing normal alpha beta. Therefore if the nodes are sorted ideally, Negascout will finish in less time than alpha beta, however if not, Negascout will actually be slower as the initial test would have been in vain.

Chapter 5 - Experiments

5.1 Introduction

There were several experiments conducted to gauge the overall performance of the agent as well as to determine under what conditions did the agent excel and fall short. These experiments consisted of several test games played against four agents (three deterministic, one non-deterministic) as well as human players of different, self-imposed levels of competence.

5.2 Test Agents

Four test agents were designed that all use simple approaches and are used as a base to grade performance on. Some of them were used during training.

5.2.1 Deterministic

The first three agents are deterministic. All three of these agents were involved in at least one of the learning phases of the main agent (i.e. influence map and neural network weights).

5.2.1.1 Greedy Agent

The first agent takes a greedy approach to picking a move. It goes through its list of valid moves and chooses the one that flips over the most opponent pieces. In the game of Othello, the number of pieces one has in the beginning and middle of the game is often a poor indication of how well one is actually performing. Therefore, an agent that makes its decisions based solely on the number of pieces it can flip will not do well. This agent was therefore the easiest of the three to beat.

The other two agents, which will be discussed shortly, rely on influence maps to help make their decisions. When creating the influence maps for these agents and for the main agent (using genetic algorithms as discussed earlier), this greedy agent was used for testing.

This agent was also used as a test agent to train the weights of the static evaluation function's neural network.

5.2.1.2 Influence Map Agent

The influence map agent uses nothing but the learned influence map values, stated previously, to make its decisions and disregards the current board state, i.e. the arrangement and placement of any of its or its opponent's pieces. It basically just gets the corresponding influence map value for each of its valid moves and chooses the move with the maximum value.

This agent was used in the continual learning process for the values of the final influence map and also in learning the weights for the neural network.

5.2.1.3 Greedy Influence Agent

The next agent combines both these approaches. It retrieves the influence map value of each of its valid moves then multiplies this by the number of opponent pieces that move would flip. This turns out to be a pretty decent approach for its simplicity and is the strongest test agent. It was used to train the weights of the neural network.

5.2.2 Non-Deterministic

The final agent was non-deterministic and was not used to train the main agent or any other agent.

5.2.2.1 Random Agent

The random agent is the simplest agent as it just picks a move at random from the list of valid moves. Since the other three agents are all deterministic, there needed to be at least one agent that wasn't, therefore this agent was created.

5.2.3 Human

Lastly, actual human players tested their skills against the agent. Since humans are the most unpredictable and non-deterministic, this served as a true test of the agent's capabilities. There were three players at self-imposed skill levels of one, five, and seven out of ten (with ten being an expert player and one being a newcomer) who played a total of two, five, and two games respectively. The agent and their scores were averaged for each run under the same agent configuration and the winner was the one with the higher score. Each human player played as the white and black player against the agent with configurations of Minimax only at depths one and six, with and without expected min; and with all exploits active at Minimax depths one and six, with and without expected min.

5.3 Results

The following shows the results of the tests ran over the deterministic, non-deterministic, and human test agents.

5.3.1 Deterministic

Since the agents are deterministic, each test performed against these agents was only run once. The results are summarized in figure 5.1 below. The full results can be found in the appendix.

Without EM								
	M	MCK	MCKBoBa	MCKBoP	ALL	Total	Avg	Per
1	6	6	5	5	5	27.0	5.40	90%
2	4	4.5	4	4	3	19.5	3.90	65%
3	4	4	4.5	4	4.5	21.0	4.20	70%
4	3	4	6	5	6	24.0	4.80	80%
5	4	4	6	5	6	25.0	5.00	83%
6	5	5	3	6	4	23.0	4.60	77%
Total	26.0	27.5	28.5	29.0	28.5			
Avg	4.33	4.58	4.75	4.83	4.75			
Per	72%	76%	79%	81%	79%			

With EM								
	M	MCK	MCKBoBa	MCKBoP	ALL	Total	Avg	Per
1	6	6	5	5	5	27.0	5.40	90%
2	1.5	2	4	1	4	12.5	2.50	42%
3	3	3	5	3	5	19.0	3.80	63%
4	3	5	6	4	5	23.0	4.60	77%
5	6	5	6	6	6	29.0	5.80	97%
6	4	5	4.5	5	4.5	23.0	4.60	77%
Total	23.5	26.0	30.5	24.0	29.5			
Avg	3.92	4.33	5.08	4.00	4.92			
Per	65%	72%	85%	67%	82%			

Figure 5.1

Summary of tests ran against influence, greedy, and greedy influence agents. Each cell represents how many games out of six were won (1 game for each of the three agents as the white player, and 1 game each as the black player. Ties count as half a point). ‘M’ stands for tests using only Minimax; ‘MCK’ is Minimax, corner, and killer move detection; ‘MCKBB’ is Minimax, corner, killer, blocking, and blacklisting; ‘MCKBoP’ is Minimax, corner, killer, blocking and pattern recognition; and ‘ALL’ is for all techniques. The number on the left of each chart represents the maximum Minimax depth searched to and the chart heading shows whether or not the “expected min” technique was used. ‘Total’ was the total number of games won (the sum of the appropriate cells), ‘Avg’ is the average, and ‘Per’ is the percent won. This was applied to the different combinations of techniques (shown on the bottom of each chart) and to the different Minimax depths (shown on the right of each chart). Finally, the cells shaded in green represents situations were all six games were won by the agent, cells in red mean half or less of the games were won, and cells in gray represent the best Minimax depth or best combination of techniques for that chart.

When using the expected min approach, alpha beta pruning was turned off since, as mentioned earlier, all child nodes must be considered to propagate a value to its min parent. However, as stated previously, it had to be used at a Minimax depth of six. Therefore while depths one to five are accurate, depth six is more of an approximation.

Overall it was quite successful. Looking at the summary we see different results for the two approaches, with one using expected min and the other using the normal Minimax approach. The two best depths are one and five and the two best technique combinations have Minimax (which is required), corner detection, killer move detection, and blocking in common. Adding blacklisting causes the best results when using expected min and patterns are the best addition for the case with normal Minimax. Overall the expected min approach performed worse, however it had the highest maximum values (i.e. 97% compared to 90% for the different depths and 85% over 81% for combinations of techniques). The worse technique combination on both was using Minimax alone and the worse depth was two by a decent margin.

These results may be misleading, however, as these same agents were used in training, just not with the different parameter value combinations. It is this reason that the results of each depth and technique combination are summarized. It is not enough to just pick a parameter combination that results in the agent winning six out of six games, since this may not be the case against every type of test agent. Instead we use the parameter combination and depth that has the best overall performance. This is shown in gray in figure 5.1. The highest values were obtained by using expected min at a depth of five with the techniques of Minimax, corner detection, killer moves, blocking, and blacklisting. Without using expected min, the ideal approach would be using a depth of one with Minimax, corner detection, killer move detection, blocking and pattern

recognition. To see what kind of results the agent could obtain if it was tested over unseen agents, we use the random agent.

5.3.2 Non-Deterministic

The random agent tests were run much the same as with the deterministic agents with the exception that ten games instead of one were ran with each parameter combination. The average of these ten games became the final value (shown in the appendix). These tests were widely successful as every parameter set resulted in an overall win over the ten games. This proves the agent can adapt to an opponent making random choices and that some strategy must be used if the opponent is to have a chance at winning. Human players are truly the most unpredictable of all and therefore the next tests involve playing against several of them with different skill levels.

5.3.3 Human

The agent proved to be quite a formidable opponent against even experienced human players as shown in figure 5.2 below. At depth one with only Minimax, it won half of the games and as depths increased and all techniques were added, the performance improved. This leads to the conclusion that the agent works well in unpredictable circumstances both with and without the expected min approach.

Without EM					
	M	**ALL**	**Total**	**Avg**	**Per**
	3	5	8.0	4.00	67%
6	6	6	12.0	6.00	100%
Total	9.0	11.0			
Avg	4.50	5.50			
Per	75%	92%			

With EM					
	M	ALL	Total	Avg	Per
1	3	5	8.0	4.00	67%
6	3	6	9.0	4.50	75%
Total	6.0	11.0			
Avg	3.00	5.50			
Per	50%	92%			

Figure 5.2

Summary of tests ran against human testers. The format is the same as that of figure 5.1.

Like the tests ran against the three initial agents, at a depth of six when expected min was used, alpha beta pruning was turned on. This would mean that the results at that depth with expected min would be an estimate, however this shouldn't necessarily be considered a bad thing. Tests were ran against the deterministic agents using expected min with alpha beta active at all depths. The results are shown in figure 5.3 below.

EM with Alpha Beta Pruning								
	M	MCK	MCKBoBa	MCKBoP	ALL	Total	Avg	Per
1	6	6	5	5	5	27.0	5.40	90%
2	1	1	2	3	1	8.0	1.60	27%
3	3	0.5	5	4	5	17.5	3.50	58%
4	4	3	6	3	6	22.0	4.40	73%
5	4	5	3	4	3	19.0	3.80	63%
6	4	5	4.5	5	4.5	23.0	4.60	77%
Total	22.0	20.5	25.5	24.0	24.5			
Avg	3.67	3.42	4.25	4.00	4.08			
Per	61%	57%	71%	67%	68%			

Figure 5.3

Same as expected min chart in figure 5.1 only with alpha beta pruning turned on. These results are equivalent or better than those in figure 5.1 with alpha beta off. The format is the same as that in figure 5.1.

Comparing this to figure 5.1, we see that the performance was either the same or worse than with alpha beta turned off. Therefore, the results of tests ran using human players at depth six with expected min using alpha beta could only be improved by not using alpha beta. Since with all techniques on, the agent had an overall higher score on six out of six games than its opponent, this distinction should matter.

5.4 Conclusion

The agent seems to have performed very well in all three situations, i.e. against artificial deterministic agents, an artificial non-deterministic agent, and human agents. There were, however, some anomalies in the results that must be explained.

5.4.1 Reason for Results

The reason for the high performance over the three deterministic agents at a Minimax depth of one, could be, as stated earlier, due to the agent being trained at a depth of one using those same agents. This is further proven by the fact that in human trials, a depth of one scored the lowest. Had those same agents not have been used and/or had the training not occurred at depth one, it should have a performance lower than that of depth two. This would complement the expected result of having the performance increase with depth as is the case from depths two to five. If we look at the expected min case, neglecting a depth of one due to the reasons above, we see that from depths two to five that not only does the overall performance increase, the number of games won increases or remains the same for every combination of techniques. We notice, however, that at depth six, the performance degrades considerably. This can be explained by the use of alpha beta pruning at that depth as this can negatively affect the

decisions of the expected min algorithm. To explain the drop in performance when using normal Minimax, we have to examine each set of techniques. Notice that for three of the five columns/sets of techniques, the performance actually increases. The decrease only comes when blacklisting is used. Therefore it seems as though blacklisting might be removing valid choices that would have otherwise been chosen by the more accurate decisions of the depth six Minimax. This sort of scenario complements the warning mentioned when describing the blacklisting technique; that making blacklisting (or many other game specific heuristics) too aggressive could override a better decision made by Minimax. Unfortunately, there isn't really a good way to tell whether Minimax or the other techniques result in the better move, therefore all techniques are to be used with caution and plenty of forethought into the various possible scenarios.

Knowing the results of different experiments and reasons for them is not enough, though. One must think critically about choosing the right technique combination that not only satisfies the desired performance threshold, but also is ideal for the chosen environment. Therefore the next section examines the different metrics needed to make that decision and offers up some suggestions on ideal parameter and techniques combinations.

5.4.2 Picking the Best Combination

Figure 5.4 below shows graphically the performance of using over not using expected min. At depth one only max is checked and therefore both expected and normal min Minimax have the same accuracy. For depths two, three and four, normal min performs better. Depth five shows expected min giving better results and depth six shows equal accuracy for both.

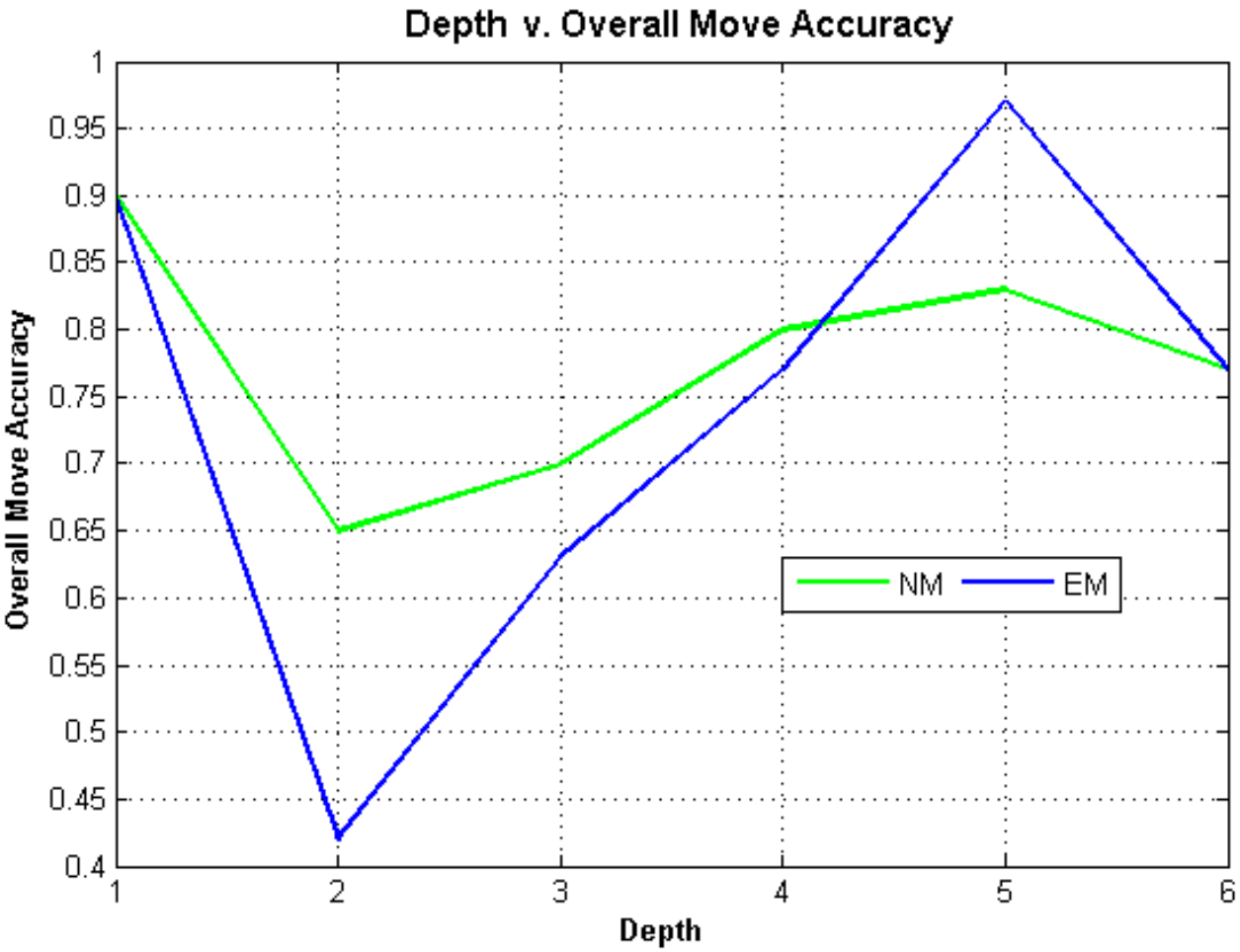


Figure 5.4

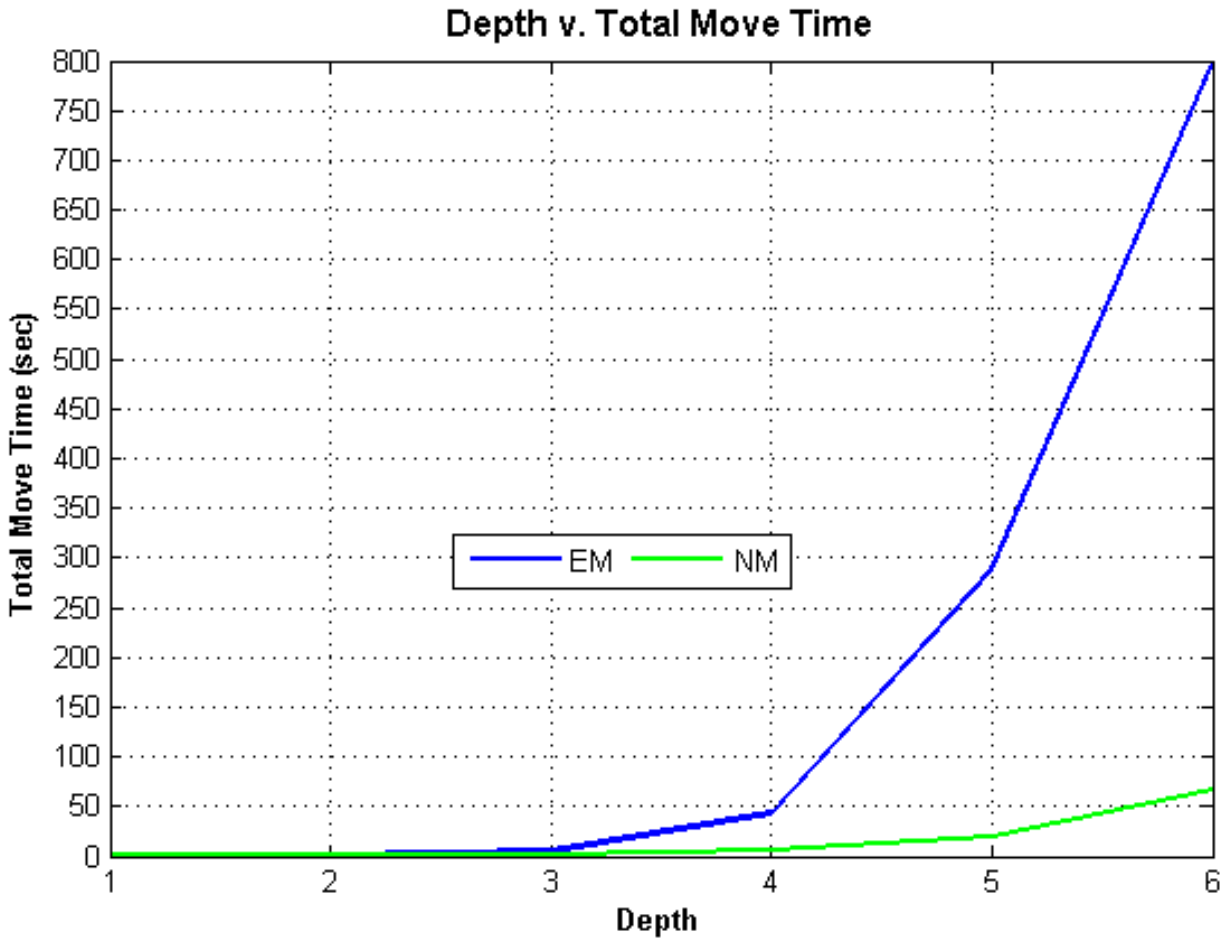
This shows how the accuracy of the move choice varies with the Minimax maximum depth searched. “NM” is for normal Minimax and is shown in green and “EM” is for expected min Minimax and is shown in blue. The data for this table is taken from figure 5.1.

The main problem with the expected min approach is that fact that using Alpha Beta with it can reduce its performance (as shown at depth 6 in figure 5.4). Most optimizations for Minimax involve pruning and none of these can be used, therefore the number of nodes searched and the time it takes to make a decision increases dramatically. It is important, then, to know what application the agent is going to be used in before choosing an ideal set of parameters.

If the agent is going to be placed in an Othello game meant for human users (e.g. as an online or store bought game), then speed will need to be taken into account. In this case move accuracy is negotiable since one would want to give users some sort of difficulty choice. One might therefore choose different parameter combinations keeping speed in mind. If giving the users choices such as easy, medium and hard difficulty, choosing expected min, Minimax only, and searching to a depth of 2 for easy; normal min, Minimax with corner and killer move detection, and a depth of 3 would do well for medium; and normal min, all techniques, and depth of six for hard (due to its performance in human testing).

On the other hand, if move accuracy is more important than move time, such as the case where the agent will be put against other computer agents, then a good combination could be: expected min, Minimax, corner detection, killer move detection, blocking, and blacklisting to a depth of five as it scored the highest over the trials against the deterministic agents and matched performance with other parameter combinations in the random agent trials. Due to an average move time of over nine seconds, this wouldn't be ideal for human players to play against.

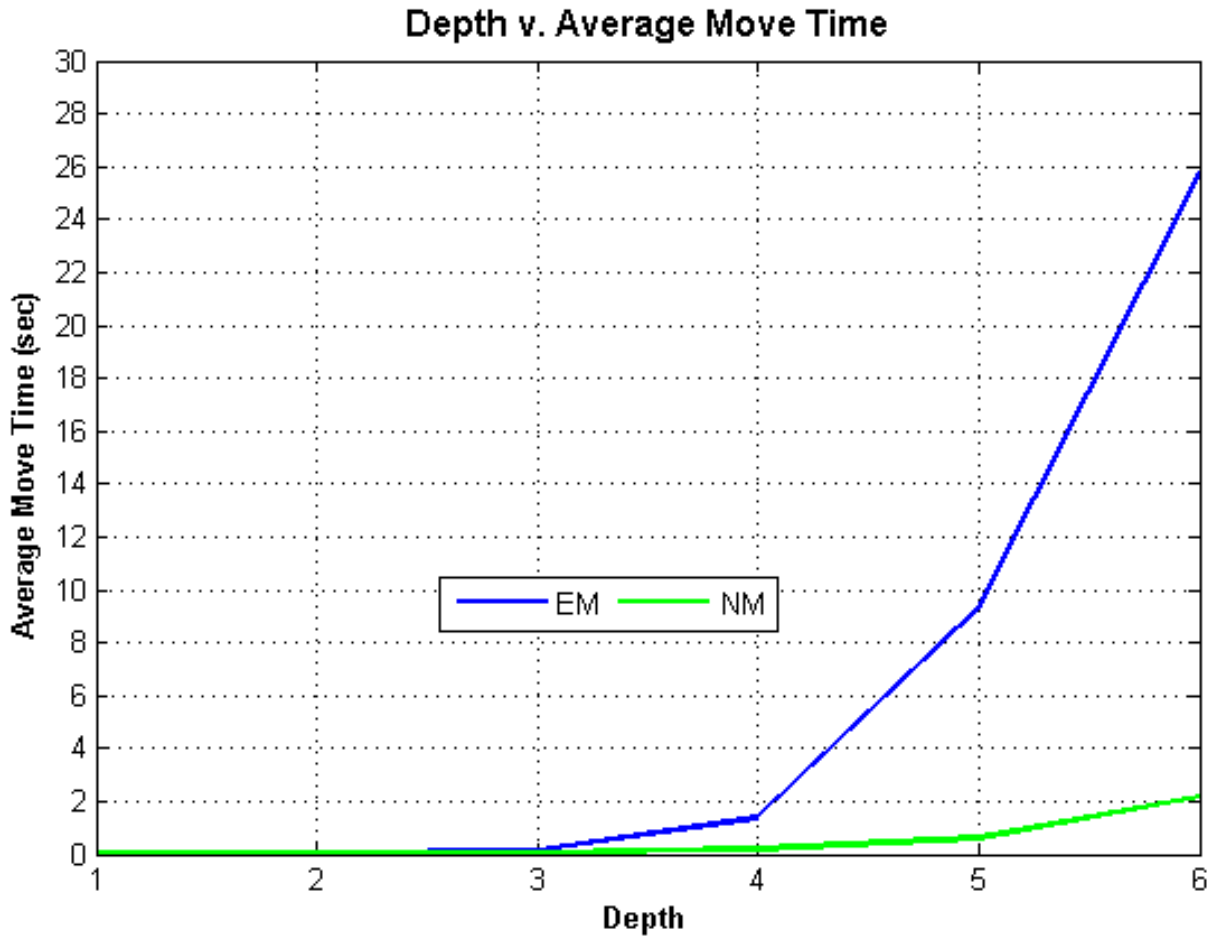
The following figures show total and average move times and total and average nodes searched for different Minimax depths used with and without the expected min algorithm. As mentioned it is important to figure out whether memory, time, or accuracy is the most important factor in one's own implementation. Figure 5.4 gives a good indication of accuracy, while the figures below give detailed information about memory consumption and execution time. Striking a balance between all three factors should be the goal, however if one factor is more important than the rest, this data can help one understand the tradeoffs one is making.



Total Move Times (sec)						
	1	**2**	**3**	**4**	**5**	**6**
NM	0.081	0.254	2.032	6.325	20.098	67.553
EM	0.080	0.608	4.555	43.278	288.101	798.370

Figure 5.5

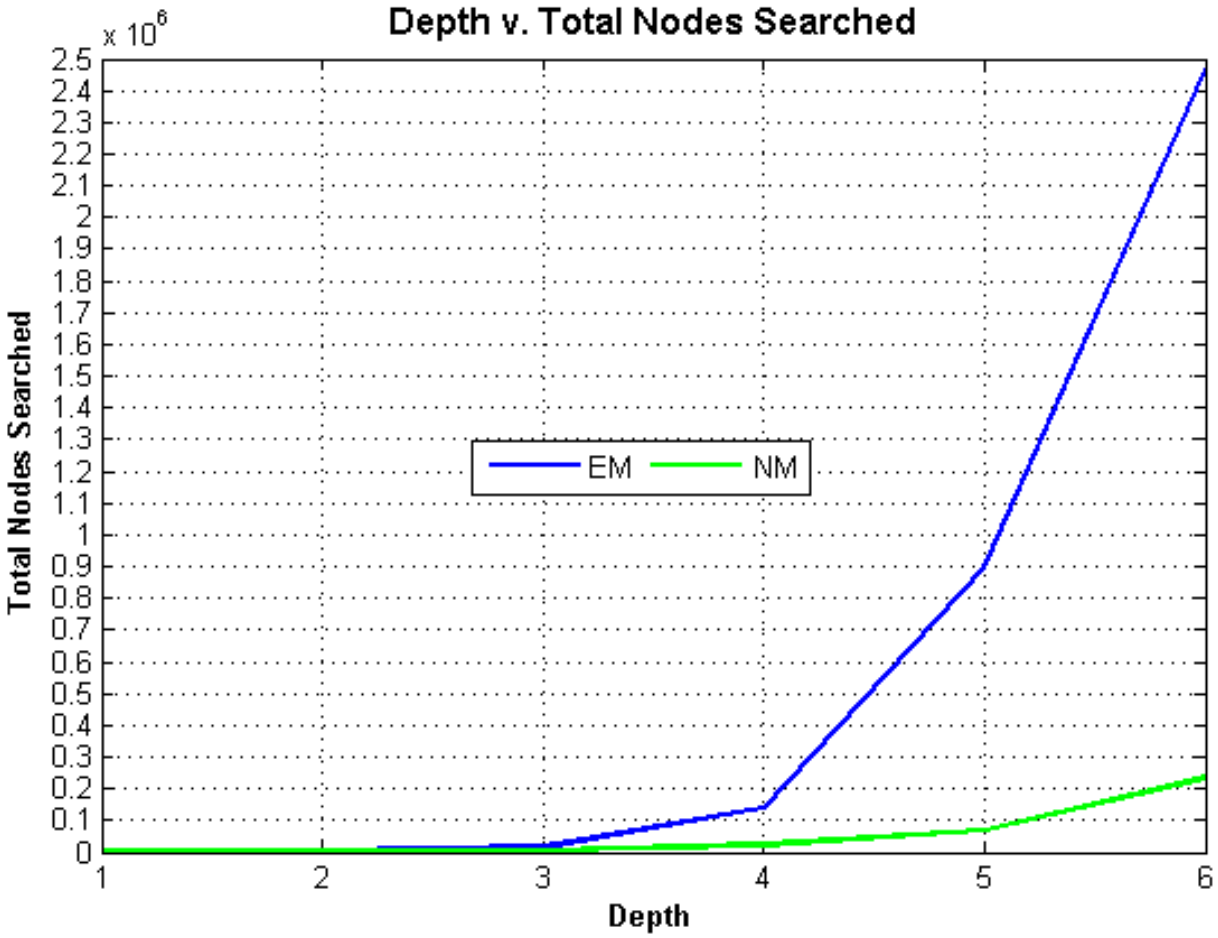
This graph shows how the total move time varies with the depth for normal and expected min Minimax. The total move time refers to the cumulative amount of time spent during each of the agent moves in a single game (not counting its opponent's moves). The data for the graph is shown in the table below it. Note that since expected min could not be run at depth six without alpha beta, the value for that cell was approximated using the data from depths one to five and a third order polynomial (decided from testing different methods using Excel).



Average Move Times (sec)						
	1	**2**	**3**	**4**	**5**	**6**
NM	0.002	0.008	0.063	0.197	0.609	2.179
EM	0.002	0.019	0.151	1.396	9.293	25.749

Figure 5.6

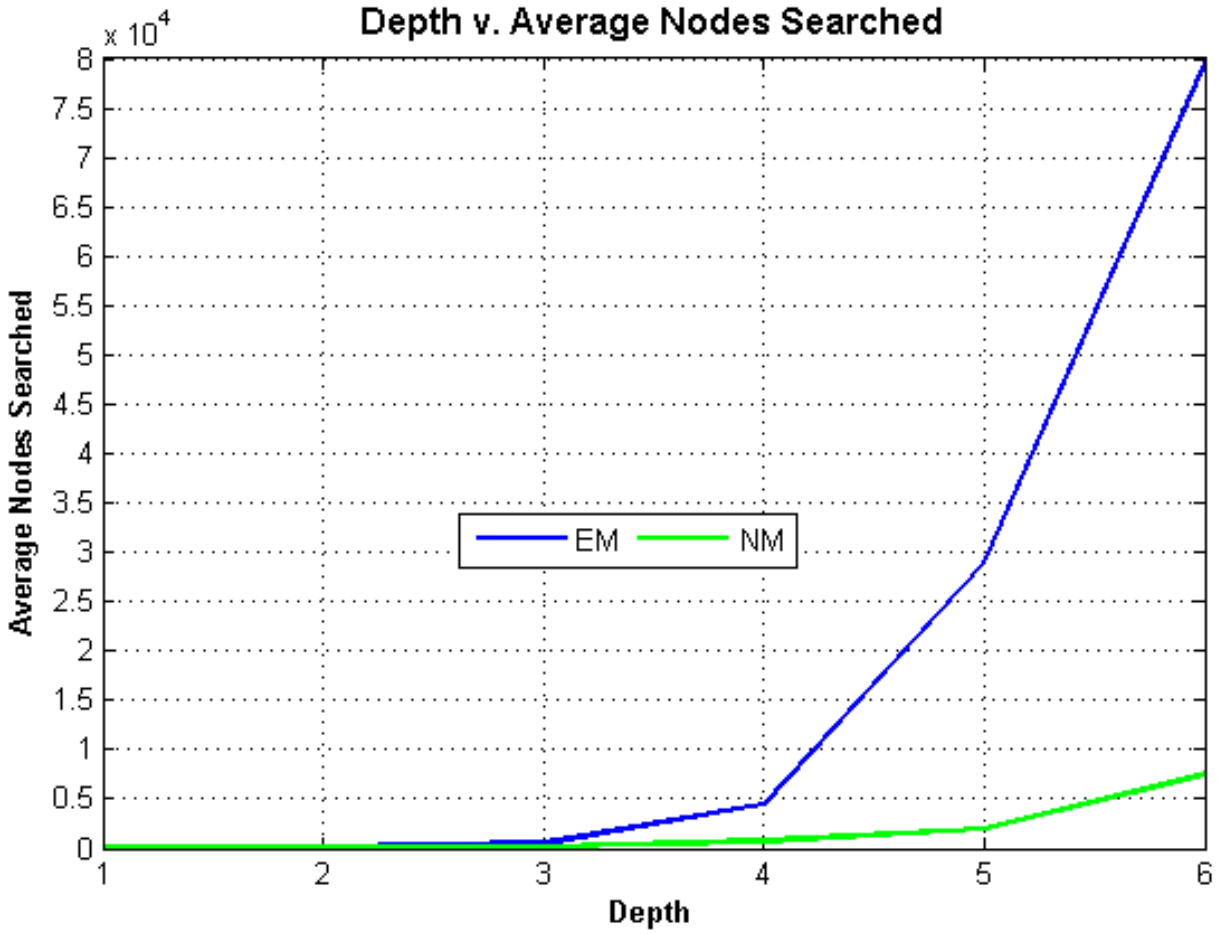
This graph shows how the average move time varies with the depth for normal and expected min Minimax. The average move time refers to the cumulative amount of time spent during each of the agent moves in a single game (not counting its opponent's moves) divided by its total number of moves during that game. The data for the graph is shown in the table below it. Note that, like in figure 5.5, a third order polynomial was used to find data for a depth of six for expected min.



Total Nodes Searched						
	1	**2**	**3**	**4**	**5**	**6**
NM	215	828	6,995	22,851	67,223	233,976
EM	215	2,049	14,916	139,552	894,735	2,464,762

Figure 5.7

This graph shows how the total number of nodes searched varies with the depth for normal and expected min Minimax. The total number of nodes searched refers to the cumulative amount of Minimax tree nodes built during each of the agent moves in a single game (not counting its opponent's moves). The data for the graph is shown in the table below it. The data for depth six using expected min was approximated using the same technique as in the figures above.



Average Nodes Searched						
	1	**2**	**3**	**4**	**5**	**6**
NM	7	28	219	714	2,037	7,548
EM	7	66	497	4,502	28,862	79,496

Figure 5.8

This graph shows how the total number of nodes searched varies with the depth for normal and expected min Minimax. The total number of nodes searched refers to the cumulative amount of Minimax tree nodes built during each of the agent moves in a single game (not counting its opponent's moves) divided by its total number of moves during that game. The data for the graph is shown in the table below it. The data for depth six using expected min was approximated using the same technique as in the figures above.

Chapter 6 – Conclusion and Future Work

6.1 Conclusion

The research and approach to tackling the problem of artificial intelligence in the game Othello has been shown. Five different types of game specific heuristics, namely killer move detection, corner detection, blocking, pattern detection, and blacklisting were developed along with a variation on normal Minimax and a creative mix of genetic algorithms and neural networks to form Minimax's static evaluation function. Tests were run on several combinations of these game specific techniques, with and without using the Minimax variation and at several Minimax search depths. These tests have covered already seen cases as well as unseen and very unpredictable cases. These results have been summarized and explained and suggestions were given on how best to utilize the techniques on those bases. Overall the agent performed quite well on all tests presented to it.

6.2 Future Work

There are always ways to improve upon techniques being used. The following are ideas for further work in this area that could be used to improve upon existing performance.

6.2.1 Cross Validation with Training Agents

As mentioned some of the same agents used for testing were used in training and although the parameters were varied, this did influence the testing results.

Introducing a new, non-deterministic agent as well as testing against human agents helped to establish more accurate results, however other methods could have also been used. A variation on the “leave-one-out” cross-validation algorithm could help wherein two of the three agents would be used for training with the third used for testing. For example, greedy and greedy influence agents could be used in genetic algorithm training while the influence map agent will be used for testing. This would be repeated for all three agents using a different one for testing each time.

6.2.2 More In-Depth Static Evaluation Function

Initially only a few board features were taken as input into the neural network for the static evaluation function. When more were added and learning restarted, the final results were better. This creates the assumption that adding more input features could improve the performance, although only up to a point and only with certain features. Some features to consider are: using a normalized value for the number of unique pieces flipped by all of the opponent’s next moves as it may be misleading without it; taking the average or maximum number of pieces flipped by the opponent’s next move(s) to gauge how good his next move(s) are; number of pieces in straight lines of 5 or more as this indicates a strong collection of pieces that are hard to overtake; and using that same logic, pieces forming a 3 by 3 square are also strong collections. For the line to be strong, at least one of the vertices needs to lie on a corner or side of the board. This will guarantee that an opponent’s piece placed at the end of that line cannot be used to capture the entire line. However, if both vertices are found on the side of the board, an opponent can still overtake the entire line. This only happens if both vertices are on the same side as vertices on different sides form a strong line. Therefore multiple cases must be taken into account and only strong lines should be counted.

Keep in mind that each added feature means retraining the neural network from scratch and will take longer to reach a level greater than the one it was previously at. Also note that the added feature may not even help the overall performance.

6.2.3 More Minimax Optimizations

Since speed wasn't a big concern, no optimizations past alpha beta were implemented. However if more were used, searching deeper may have been possible and could have been beneficial. Some of the more common approaches are Negascout (discussed earlier) and aspiration search, to allow for deeper searching on specific nodes where certain criteria is meet. This quote from Singer explains his approach on a variation of aspiration search, "Othello has a nice property: with each move made, the number of empty squares left on the board decreases by one. Therefore, once the game reaches the point where there are only a handful of empty squares left on the board, the remaining game-tree can be completely calculated" [28]. Basically if there are only a few moves left before the game ends, expand the tree all the way on that node to get a truer value of the board state.

6.2.4 Reinforcement Learning with Neural Networks

Normally, backpropagation or feedforward methods are used to train neural networks, however a genetic algorithm was used instead. The maximum fitness attained was pretty high using this method, yet proper unsupervised neural network learning algorithms may still yield better results.

6.2.5 Move History

A history table is basically a lookup table for known good moves that were found by the agent during training. These moves could be repeated if ever the same board

features occur in later games. This can complement algorithms such as Minimax by remembering node values of earlier trees and using them instead of recalculating. This can also be used to search deeper without as many calculations. The agent might benefit from storing such moves and/or node values.

6.2.6 More Patterns

The patterns implemented were solely used for capturing corners on the agent's next move. With the flexibility of the pattern implementation, more patterns could be created to improve performance. This has to be done with caution, however, as sometimes allowing the Minimax algorithm to choose a move is a better idea. Therefore to truly see if adding more patterns are beneficial, one would need to add them one at a time and run tests after each addition to see how the agent's performance is affected. If there is no effect, then either more tests must be run or the pattern is too specific. If there is a negative effect on some tests and positive on others, it must be determined why and under what situations is that pattern productive. It may even be the case that increasing the Minimax depth by one results in better gameplay than adding more patterns.

References

[1] Waltz, David. “Artificial Intelligence.” *University of Washington Computer Science & Engineering*. 1996. NEC Research Institute and the Computing Research Association. 28 February 2011. <http://www.cs.washington.edu/homes/lazowska/cra/ai.html>.

[2] Touretzky, David S. *Advances in neural information processing systems 1*. CA, USA: Morgan Kaufmann Publishers Inc, 1989. 305-313. Print.

[3] “Autonomous Vehicle Systems.” *Autonomous Vehicle Systems*. 2004. Autonomous Vehicle Systems LLC. 28 February 2011. <http://www.autonvs.com/technology.html>.

[4] BryanSpear. “Military Use of Artificial Intelligence | eHow.com.” *eHow | How To Do Just About Everything! | How To Videos & Articles*. 15 May 2010. eHow, Inc. 28 February 2011. <http://www.ehow.com/about_6516343_military-use-artificial-intelligence.html>.

[5] Amrut Software. “Othello Origin – The History of Othello Game.” *Latest information and latest videos on popular board games - Boardgaminginfo.com*. BoardGaminginfo. 26 February 2011. <http://boardgaminginfo.com/othello.php>.

[6] Mattel, Inc. *Othello*. California: J.A.R. Games Co, 2002. Print.

[7] Rognlie Richard. “Reversi.” *Gamerz.NET Enterprises*. 26 February 2011. <http://www.gamerz.net/pbmserv/reversi.html#rules>.

[8] “AI Horizon: Minimax Game Tree Programming, Part 1.” *AI Horizon: Computer Science and Artificial Intelligence Programming Resources*. AI Horizon. 27 February 2011. <http://www.aihorizon.com/essays/basiccs/trees/minimax.htm>.

[9] Rajiv Bakulesh Shah, “minimax,” in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed, U.S. National Institute of Standards and Technology. 10 January 2007. 27 February 2011. <http://xlinux.nist.gov/dads//HTML/minimax.html>.

[10] Shapiro, Stuart C. *Encyclopedia of Artificial Intelligence*. USA: Wiley-Interscience, 1987. 4-7. Print.

[11] Mitchell, Tom M. *Machine Learning*. Singapore: Mcgraw Hill, 1997. 249-270. Print.

[12] Graupe, Daniel. *Principles of Artificial Neural Networks*. Singapore: World Scientific, 1997. 1-3. Print.

[13] Tournavitis, Konstantinos. *MOUSE(μ): A Self-teaching Algorithm that Achieved Master-Strength at Othello*. Berlin: Springer, 2003. Pdf.

[14] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning*. USA: A Bradford Book, 1998. 261-267. Print.

[15] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning*. USA: A Bradford Book, 1998. 133-176. Print.

[16] "Bitboad – Wikipedia, the free encyclopedia." *Wikipedia, the free encyclopedia*. 2 November 2010. Wikipedia. 27 February 2011. <http://en.wikipedia.org/wiki/Bitboard>.

[17] Rose, Brian. *Othello: A Minute to Learn... A Lifetime to Master*. Anjar Co., 2005. Pdf.

[18] Matthews, James. "generation5 – Simple Board Game AI." *generation5 – At the forefront of Artificial Intelligence*. 27 December 2004. Generation5. 27 February 2011. <http://www.generation5.org/content/2000/boardai.asp>.

[19] Mitchell, Tom M. *Machine Learning*. Singapore: Mcgraw Hill, 1997. 255. Print.

[20] Shepherd, Adrian J. *Second-Order Methods for Neural Networks*. Great Britain: Springer, 1997. 3. Print.

[21] Duda, Richard O., Hart, Peter E., and David G. Stork. *Pattern Classification*. USA: Wiley-Interscience, 2001. 310-311. Print.

[22] Bishop, Christopher M. *Pattern Recognition and Machine Learning*. Spring, 2006. 3. Print.

[23] Smed, Jouni, and Harri Hakonen. *Algorithms and Networking for Computer Games*. Finland: John Wiley & Sons, Ltd, 2006. 80-82. Print.

[24] "Variation (game tree) – Wikipedia, the free encyclopedia." *Wikipedia, the free encyclopedia*. 4 September 2010. Wikipedia. 27 February 2011. <http://en.wikipedia.org/wiki/Variation_(game_tree)>.

[25] "Negascout – Wikipedia, the free encyclopedia." *Wikipedia, the free encyclopedia*. 12 September 2010. Wikipedia. 27 February 2011. <http://en.wikipedia.org/wiki/Negascout>.

[26] Reinefeld, Alexander. *Die Entwicklung der Spielprogrammierung: Von John von Neumann bis zu den hochparallelen Schachmaschinen*. Zuse Institut Berlin: Humboldt-Universität zu Berlin, 2005. 47-51. Pdf.

[27] "Negamax – Wikipedia, the free encyclopedia." *Wikipedia, the free encyclopedia*. 11 February 2011. Wikipedia. 27 February 2011. <http://en.wikipedia.org/wiki/Negamax>.

[28] Singer, Joshua A. *Co-evolving a Neural-Net Evaluation Function for Othello by Combining Genetic Algorithms and Reinforcement Learning*. Stanford University: Springer, 2001. Pdf.

Appendix: Experiment Results

The next several tables detail the exact values for the trials against the three test agents and the random agent. Each test was done against each agent, at each depth from one to six, with and without the “expected min” approach, and as the white and black player. For the first charts, the agent is indicated in the first column as either “inf” for the influence map agent, “gre” for greedy, and “ginf” for greedy influence. The numbers from one to six at the top are the Minimax depths. The fitness and score values are for the “A” agent, and “O” opponent. The corners value is a ratio where the top number represents the number of corners taken by the agent at the end of the game, and the bottom represents those taken by the opponent. Those cells highlighted in green represent a win for the agent and red indicates a loss. For the first charts, the last two columns are the total and average values for each row. The last charts have total and average indicated in the first column.

The tests over the influence, greedy and greedy influence agents were run using these parameters:

- **A.** Minimax only
- **B.** Minimax, corner detection, and killer move detection
- **C.** Minimax, corner detection, killer move detection, blocking and blacklisting
- **D.** Minimax, corner detection, killer move detection, blocking and patterns
- **E.** Minimax, corner detection, killer move detection, blocking, blacklisting, and patterns (i.e. all techniques)

The tests over the random agent were run using these parameters:

F. Minimax only

G. Minimax, corner detection, killer move detection, blocking, blacklisting, and patterns (i.e. all techniques)

This data is provided only for completeness as summaries and useful trends are provided in previous figures and in the main text.

A. Minimax only

White Player										
		Depth	**1**	**2**	**3**	**4**	**5**	**6**	**Totals**	**Avg**
Inf	**w/ em**	**Fitness: A**	1.78	0.63	6.76	13.00	6.56	7.27	36.00	6.00
		Fitness: O	6.22	7.37	1.24	-5.00	1.44	0.73	12.00	2.00
		Score: A	41	27	47	54	46	49	264.00	44.00
		Score: O	23	37	17	10	18	15	120.00	20.00
		Corners	0/4	1/3	2/2	4/0	2/2	2/2	11/13	1.83/2.17
	w/o em	**Fitness: A**	1.78	4.20	1.78	5.56	-2.37	1.29	12.24	2.04
		Fitness: O	6.22	3.80	6.22	2.44	10.37	6.71	35.76	5.96
		Score: A	41	44	41	39	19	36	220.00	36.67
		Score: O	23	20	23	25	45	28	164.00	27.33
		Corners	0/4	1/3	0/4	2/2	0/4	0/4	3/21	0.5/3.5
Gre	**w/ em**	**Fitness: A**	13.00	3.00	7.91	0.79	12.33	13.00	50.04	8.34
		Fitness: O	-5.00	3.00	0.09	7.21	-4.33	-5.00	-4.04	-0.67
		Score: A	60	32	42	29	52	58	273.00	45.50
		Score: O	4	32	22	35	12	6	111.00	18.50
		Corners	4/0	2/2	3/1	1/3	4/0	4/0	18/6	3/1
	w/o em	**Fitness: A**	13.00	7.56	10.56	7.27	10.82	6.37	55.57	9.26
		Fitness: O	-5.00	0.44	-2.56	0.73	-2.82	1.63	-7.57	-1.26
		Score: A	60	39	46	49	53	45	292.00	48.67
		Score: O	4	25	18	15	11	19	92.00	15.33
		Corners	4/0	3/1	4/0	2/2	3/1	2/2	18/6	3/1
GInf	**w/ em**	**Fitness: A**	8.33	-3.00	-3.27	3.91	4.20	6.05	16.22	2.70
		Fitness: O	-0.33	11.00	11.27	4.09	3.80	1.95	31.78	5.30
		Score: A	52	16	15	42	44	43	212.00	35.33
		Score: O	12	48	49	22	20	21	172.00	28.67
		Corners	2/2	0/4	0/4	1/3	1/3	2/2	6/18	1/3
	w/o em	**Fitness: A**	8.33	0.87	-1.37	-2.20	6.20	-1.67	10.16	1.69
		Fitness: O	-0.33	7.13	9.37	10.20	1.80	9.67	37.84	6.31
		Score: A	52	30	27	20	44	24	197.00	32.83
		Score: O	12	34	37	44	20	40	187.00	31.17
		Corners	2/2	1/3	0/4	0/4	2/2	0/4	5/19	0.83/3.17

A. Minimax only

Black Player										
		Depth	1	2	3	4	5	6	Totals	Avg
Inf	w/ em	Fitness: A	18.00	7.29	-1.06	-2.37	1.56	-2.37	21.04	3.51
		Fitness: O	-10.00	0.71	9.06	10.37	6.44	10.37	26.96	4.49
		Score: A	64	36	31	19	39	19	208.00	34.67
		Score: O	0	28	33	45	25	45	176.00	29.33
		Corners	4/0	3/1	0/4	0/4	0/4	0/4	7/17	1.17/2.83
	w/o em	Fitness: A	18.00	-1.56	0.09	4.05	-1.37	1.91	21.12	3.52
		Fitness: O	-10.00	9.56	7.91	3.95	9.37	6.09	26.88	4.48
		Score: A	64	25	22	43	27	42	223.00	37.17
		Score: O	0	39	42	21	37	22	161.00	26.83
		Corners	4/0	0/4	1/3	1/3	0/4	0/4	6/18	1/3
Gre	w/ em	Fitness: A	14.00	-10.00	3.06	2.22	5.21	7.13	21.62	3.60
		Fitness: O	-10.00	10.00	4.94	5.78	2.79	0.87	14.38	2.40
		Score: A	38	0	33	23	35	34	163.00	27.17
		Score: O	0	13	31	41	29	30	144.00	24.00
		Corners	2/0	0/0	1/3	2/2	2/2	3/1	10/8	1.67/1.33
	w/o em	Fitness: A	14.00	13.00	13.00	0.63	7.37	4.76	52.76	8.79
		Fitness: O	-10.00	-5.00	-5.00	7.37	0.63	3.24	-8.76	-1.46
		Score: A	38	57	55	27	37	47	261.00	43.50
		Score: O	0	7	9	37	27	17	97.00	16.17
		Corners	2/0	4/0	4/0	1/3	3/1	1/3	15/7	2.5/1.17
GInf	w/ em	Fitness: A	16.00	-10.00	-3.00	8.56	9.27	-10.00	10.82	1.80
		Fitness: O	-10.00	10.00	11.00	-0.56	-1.27	12.00	21.18	3.53
		Score: A	53	0	16	46	49	0	164.00	27.33
		Score: O	0	13	48	18	15	13	107.00	17.83
		Corners	3/0	0/0	0/4	3/1	3/1	0/1	9/7	1.5/1.17
	w/o em	Fitness: A	16.00	5.91	5.29	0.87	9.92	9.00	46.98	7.83
		Fitness: O	-10.00	2.09	2.71	7.13	-1.92	-1.00	-0.98	-0.16
		Score: A	53	42	36	30	51	48	260.00	43.33
		Score: O	0	22	28	34	13	16	113.00	18.83
		Corners	3/0	2/2	2/2	1/3	3/1	3/1	14/9	2.33/1.5

B. Minimax, corner detection, and killer move detection

White Player										
		Depth	1	2	3	4	5	6	Totals	Avg
Inf	w/ em	Fitness: A	1.78	0.63	6.37	11.27	6.56	6.56	33.16	5.53
		Fitness: O	6.22	7.37	1.63	-3.27	1.44	1.44	14.84	2.47
		Score: A	41	27	45	49	46	46	254.00	42.33
		Score: O	23	37	19	15	18	18	130.00	21.67
		Corners	0/4	1/3	2/2	4/0	2/2	2/2	11/13	1.83/2.17
	w/o em	Fitness: A	1.78	5.91	1.78	5.78	-2.37	1.29	14.17	2.36
		Fitness: O	6.22	2.09	6.22	2.22	10.37	6.71	33.83	5.64
		Score: A	41	42	41	41	19	36	220.00	36.67
		Score: O	23	22	23	23	45	28	164.00	27.33
		Corners	0/4	2/2	0/4	2/2	0/4	0/4	4/20	0.67/3.33
Gre	w/ em	Fitness: A	12.82	5.78	16.00	0.79	4.71	13.00	53.11	8.85
		Fitness: O	-4.82	2.22	-10.00	7.21	3.29	-5.00	-7.11	-1.18
		Score: A	53	41	61	29	28	56	268.00	44.67
		Score: O	11	23	0	35	36	8	113.00	18.83
		Corners	4/0	2/2	3/0	1/3	3/1	4/0	17/6	2.83/1
	w/o em	Fitness: A	12.82	7.78	7.27	7.91	8.76	6.37	50.91	8.48
		Fitness: O	-4.82	0.22	0.73	0.09	-0.76	1.63	-2.91	-0.48
		Score: A	53	41	49	42	47	45	277.00	46.17
		Score: O	11	23	15	22	17	19	107.00	17.83
		Corners	4/0	3/1	2/2	3/1	3/1	2/2	17/7	2.83/1.17
GInf	w/ em	Fitness: A	8.33	-3.00	-3.27	9.57	7.00	5.67	24.30	4.05
		Fitness: O	-0.33	11.00	11.27	-1.57	1.00	2.33	23.70	3.95
		Score: A	52	16	15	50	48	40	221.00	36.83
		Score: O	12	48	49	14	16	24	163.00	27.17
		Corners	2/2	0/4	0/4	3/1	2/2	2/2	9/15	1.5/2.5
	w/o em	Fitness: A	8.33	3.06	-1.37	-2.20	8.05	-1.67	14.21	2.37
		Fitness: O	-0.33	4.94	9.37	10.20	-0.05	9.67	33.79	5.63
		Score: A	52	33	27	20	43	24	199.00	33.17
		Score: O	12	31	37	44	21	40	185.00	30.83
		Corners	2/2	1/3	0/4	0/4	3/1	0/4	6/18	1/3

B. Minimax, corner detection, and killer move detection

Black Player										
		Depth	1	2	3	4	5	6	Totals	Avg
Inf	w/ em	Fitness: A	18.00	5.21	-1.06	5.29	1.56	6.20	35.19	5.86
		Fitness: O	-10.00	2.79	9.06	2.71	6.44	1.80	12.81	2.14
		Score: A	64	35	31	36	39	44	249.00	41.50
		Score: O	0	29	33	28	25	20	135.00	22.50
		Corners	4/0	2/2	0/4	2/2	0/4	2/2	10/14	1.67/2.33
	w/o em	Fitness: A	18.00	-1.56	0.09	4.05	11.00	1.91	33.49	5.58
		Fitness: O	-10.00	9.56	7.91	3.95	-3.00	6.09	14.51	2.42
		Score: A	64	25	22	43	55	42	251.00	41.83
		Score: O	0	39	42	21	9	22	133.00	22.17
		Corners	4/0	0/4	1/3	1/3	3/1	0/4	9/15	1.5/2.5
Gre	w/ em	Fitness: A	14.00	-10.00	8.56	10.56	10.37	7.13	40.61	6.77
		Fitness: O	-10.00	10.00	-0.56	-2.56	-2.37	0.87	-4.61	-0.77
		Score: A	38	0	46	46	45	34	209.00	34.83
		Score: O	0	13	18	18	19	30	98.00	16.33
		Corners	2/0	0/0	3/1	4/0	4/0	3/1	16/2	2.67/0.33
	w/o em	Fitness: A	14.00	9.85	12.82	8.37	4.22	13.00	62.25	10.38
		Fitness: O	-10.00	-1.85	-4.82	-0.37	3.78	-5.00	-18.25	-3.04
		Score: A	38	50	53	45	23	61	270.00	45.00
		Score: O	0	13	11	19	41	3	87.00	14.50
		Corners	2/0	3/1	4/0	3/1	3/1	4/0	19/3	3.17/0.5
GInf	w/ em	Fitness: A	16.00	-10.00	-3.00	8.20	9.27	-10.00	10.47	1.74
		Fitness: O	-10.00	10.00	11.00	-0.20	-1.27	12.00	21.53	3.59
		Score: A	53	0	16	44	49	0	162.00	27.00
		Score: O	0	13	48	20	15	13	109.00	18.17
		Corners	3/0	0/0	0/4	3/1	3/1	0/1	9/7	1.5/1.17
	w/o em	Fitness: A	16.00	3.00	11.00	0.09	9.57	11.57	51.23	8.54
		Fitness: O	-10.00	3.00	-3.00	7.91	-1.57	-3.57	-7.23	-1.21
		Score: A	53	32	59	22	50	50	266.00	44.33
		Score: O	0	32	5	42	14	14	107.00	17.83
		Corners	3/0	2/2	3/1	1/3	3/1	4/0	16/7	2.67/1.17

C. Minimax, corner detection, killer move detection, blocking and blacklisting

White Player										
		Depth	1	2	3	4	5	6	Totals	Avg
Inf	w/ em	Fitness: A	-1.29	9.91	6.05	10.73	16.00	6.20	47.60	7.93
		Fitness: O	9.29	-1.91	1.95	-4.73	-10.00	1.80	-3.60	-0.60
		Score: A	28	42	43	52	48	44	257.00	42.83
		Score: O	36	22	21	11	0	20	110.00	18.33
		Corners	0/4	4/0	2/2	3/0	3/0	2/2	14/8	2.33/1.33
	w/o em	Fitness: A	-1.29	2.44	11.27	7.91	5.37	-1.78	23.92	3.99
		Fitness: O	9.29	5.56	-3.27	0.09	2.63	9.78	24.08	4.01
		Score: A	28	25	49	42	37	23	204.00	34.00
		Score: O	36	39	15	22	27	41	180.00	30.00
		Corners	0/4	2/2	4/0	3/1	2/2	0/4	11/13	1.83/2.17
Gre	w/ em	Fitness: A	12.82	13.00	11.00	8.20	13.00	8.20	66.22	11.04
		Fitness: O	-4.82	-5.00	-5.00	-0.20	-5.00	-0.20	-20.22	-3.37
		Score: A	53	60	62	44	57	44	320.00	53.33
		Score: O	11	4	1	20	7	20	63.00	10.50
		Corners	4/0	4/0	3/0	3/1	4/0	3/1	21/2	3.5/0.33
	w/o em	Fitness: A	12.82	11.00	13.00	16.00	10.76	10.37	73.95	12.33
		Fitness: O	-4.82	-5.00	-5.00	-10.00	-2.76	-2.37	-29.95	-4.99
		Score: A	53	55	54	59	47	45	313.00	52.17
		Score: O	11	8	10	0	17	19	65.00	10.83
		Corners	4/0	3/0	4/0	3/0	4/0	4/0	22/0	3.67/0
GInf	w/ em	Fitness: A	7.91	5.67	13.00	10.25	11.00	7.56	55.39	9.23
		Fitness: O	0.09	2.33	-5.00	-2.25	-3.00	0.44	-7.39	-1.23
		Score: A	42	40	60	51	58	39	290.00	48.33
		Score: O	22	24	4	12	6	25	93.00	15.50
		Corners	3/1	2/2	4/0	3/1	3/1	3/1	18/6	3/1
	w/o em	Fitness: A	7.91	9.78	-1.67	9.92	11.00	-1.57	35.38	5.90
		Fitness: O	0.09	-1.78	9.67	-1.92	-5.00	9.57	10.62	1.77
		Score: A	42	41	24	51	55	14	227.00	37.83
		Score: O	22	23	40	13	7	50	155.00	25.83
		Corners	3/1	4/0	0/4	3/1	3/0	1/3	14/9	2.33/1.5

C. Minimax, corner detection, killer move detection, blocking and blacklisting

Black Player										
		Depth	1	2	3	4	5	6	Totals	Avg
Inf	w/ em	Fitness: A	14.00	12.33	11.00	10.76	8.05	3.00	59.15	9.86
		Fitness: O	-10.00	-4.33	-5.00	-2.76	-0.05	3.00	-19.15	-3.19
		Score: A	46	52	59	47	43	32	279.00	46.50
		Score: O	0	12	2	17	21	32	84.00	14.00
		Corners	2/0	4/0	3/0	4/0	3/1	2/2	18/3	3/0.5
	w/o em	Fitness: A	14.00	-3.27	-1.00	9.00	10.56	-3.00	26.29	4.38
		Fitness: O	-10.00	11.27	7.00	-1.00	-2.56	11.00	15.71	2.62
		Score: A	46	15	32	48	46	16	203.00	33.83
		Score: O	0	49	32	16	18	48	163.00	27.17
		Corners	2/0	0/4	0/4	3/1	4/0	0/4	9/13	1.5/2.17
Gre	w/ em	Fitness: A	9.00	-10.00	9.21	14.00	10.37	10.37	42.94	7.16
		Fitness: O	-1.00	10.00	-1.21	-10.00	-2.37	-2.37	-6.94	-1.16
		Score: A	54	0	35	49	45	45	228.00	38.00
		Score: O	10	13	29	0	19	19	90.00	15.00
		Corners	2/2	0/0	4/0	2/0	4/0	4/0	16/2	2.67/0.33
	w/o em	Fitness: A	9.00	16.00	10.76	14.00	11.00	13.00	73.76	12.29
		Fitness: O	-1.00	-10.00	-2.76	-10.00	-5.00	-5.00	-33.76	-5.63
		Score: A	54	59	47	38	57	61	316.00	52.67
		Score: O	10	0	17	0	4	2	33.00	5.50
		Corners	2/2	3/0	4/0	2/0	3/0	4/0	18/2	3/0.33
GInf	w/ em	Fitness: A	11.00	-10.00	2.94	9.92	6.05	-10.00	9.91	1.65
		Fitness: O	-3.00	10.00	5.06	-1.92	1.95	10.00	22.09	3.68
		Score: A	61	0	31	51	43	0	186.00	31.00
		Score: O	3	13	33	13	21	13	96.00	16.00
		Corners	3/1	0/0	2/2	3/1	2/2	0/0	10/6	1.67/1
	w/o em	Fitness: A	11.00	10.20	16.00	5.46	11.00	8.37	62.03	10.34
		Fitness: O	-3.00	-2.20	-10.00	2.54	-5.00	-0.37	-18.03	-3.00
		Score: A	61	44	52	38	55	45	295.00	49.17
		Score: O	3	20	0	26	8	19	76.00	12.67
		Corners	3/1	4/0	3/0	2/2	3/0	3/1	18/4	3/0.67

D. Minimax, corner detection, killer move detection, blocking and patterns

White Player										
		Depth	1	2	3	4	5	6	Totals	Avg
Inf	w/ em	Fitness: A	1.78	0.63	13.00	9.92	9.00	11.00	45.34	7.56
		Fitness: O	6.22	7.37	-5.00	-1.92	-1.00	-3.00	2.66	0.44
		Score: A	41	27	62	51	54	60	295.00	49.17
		Score: O	23	37	1	13	10	4	88.00	14.67
		Corners	0/4	1/3	4/0	3/1	2/2	3/1	13/11	2.17/1.83
	w/o em	Fitness: A	1.78	5.91	1.78	5.78	-2.37	1.29	14.17	2.36
		Fitness: O	6.22	2.09	6.22	2.22	10.37	6.71	33.83	5.64
		Score: A	41	42	41	41	19	36	220.00	36.67
		Score: O	23	22	23	23	45	28	164.00	27.33
		Corners	0/4	2/2	0/4	2/2	0/4	0/4	4/20	0.67/3.33
Gre	w/ em	Fitness: A	11.00	13.00	5.46	0.79	16.00	13.00	59.25	9.88
		Fitness: O	-3.00	-5.00	2.54	7.21	-10.00	-5.00	-13.25	-2.21
		Score: A	60	62	38	29	48	55	292.00	48.67
		Score: O	4	2	26	35	0	9	76.00	12.67
		Corners	3/1	4/0	2/2	1/3	3/0	4/0	17/6	2.83/1
	w/o em	Fitness: A	11.00	13.00	12.33	11.00	9.00	11.00	67.33	11.22
		Fitness: O	-3.00	-5.00	-4.33	-3.00	-3.00	-3.00	-21.33	-3.56
		Score: A	60	60	52	55	54	55	336.00	56.00
		Score: O	4	4	12	9	6	9	44.00	7.33
		Corners	3/1	4/0	4/0	3/1	2/1	3/1	19/4	3.17/0.67
GInf	w/ em	Fitness: A	-5.00	-3.00	-3.27	13.00	11.00	5.56	18.29	3.05
		Fitness: O	11.00	11.00	11.27	-5.00	-5.00	2.44	25.71	4.28
		Score: A	3	16	15	56	55	39	184.00	30.67
		Score: O	51	48	49	8	8	25	189.00	31.50
		Corners	0/3	0/4	0/4	4/0	3/0	2/2	9/13	1.5/2.17
	w/o em	Fitness: A	-5.00	-3.92	-1.37	-2.20	6.05	10.20	3.75	0.63
		Fitness: O	11.00	11.92	9.37	10.20	1.95	-2.20	42.25	7.04
		Score: A	3	13	27	20	43	44	150.00	25.00
		Score: O	51	51	37	44	21	20	224.00	37.33
		Corners	0/3	0/4	0/4	0/4	2/2	4/0	6/17	1/2.83

D. Minimax, corner detection, killer move detection, blocking and patterns

Black Player										
		Depth	1	2	3	4	5	6	Totals	Avg
Inf	w/ em	Fitness: A	9.00	-1.46	-1.06	5.29	2.05	6.76	20.57	3.43
		Fitness: O	-5.00	9.46	9.06	2.71	5.95	1.24	23.43	3.90
		Score: A	54	26	31	36	43	47	237.00	39.50
		Score: O	1	38	33	28	21	17	138.00	23.00
		Corners	2/0	0/4	0/4	2/2	0/4	2/2	6/16	1/2.67
	w/o em	Fitness: A	9.00	-1.56	0.09	4.05	11.00	1.37	23.95	3.99
		Fitness: O	-5.00	9.56	7.91	3.95	-3.00	6.63	20.05	3.34
		Score: A	54	25	22	43	55	37	236.00	39.33
		Score: O	1	39	42	21	9	27	139.00	23.17
		Corners	2/0	0/4	1/3	1/3	3/1	0/4	7/15	1.17/2.5
Gre	w/ em	Fitness: A	9.92	-10.00	8.20	12.82	9.57	10.37	40.88	6.81
		Fitness: O	-1.92	10.00	-0.20	-4.82	-1.57	-2.37	-0.88	-0.15
		Score: A	51	0	44	53	50	45	243.00	40.50
		Score: O	13	13	20	11	14	19	90.00	15.00
		Corners	3/1	0/0	3/1	4/0	3/1	4/0	17/3	2.83/0.5
	w/o em	Fitness: A	9.92	13.00	11.00	9.27	16.00	10.20	69.39	11.56
		Fitness: O	-1.92	-5.00	-5.00	-1.27	-10.00	-2.20	-25.39	-4.23
		Score: A	51	56	47	49	54	44	301.00	50.17
		Score: O	13	8	3	15	0	20	59.00	9.83
		Corners	3/1	4/0	3/0	3/1	3/0	4/0	20/2	3.33/0.33
GInf	w/ em	Fitness: A	11.00	-10.00	-3.00	2.94	9.57	-10.00	0.51	0.08
		Fitness: O	-3.00	10.00	11.00	5.06	-1.57	12.00	33.49	5.58
		Score: A	58	0	16	31	50	0	155.00	25.83
		Score: O	6	13	48	33	14	13	127.00	21.17
		Corners	3/1	0/0	0/4	2/2	3/1	0/1	8/9	1.33/1.5
	w/o em	Fitness: A	11.00	7.27	5.67	3.21	11.00	6.56	44.70	7.45
		Fitness: O	-3.00	0.73	2.33	4.79	-3.00	1.44	3.30	0.55
		Score: A	58	49	40	35	56	46	284.00	47.33
		Score: O	6	15	24	29	8	18	100.00	16.67
		Corners	3/1	2/2	2/2	1/3	3/1	2/2	13/11	2.17/1.83

E. Minimax, corner detection, killer move detection, blocking, blacklisting and patterns (i.e. all techniques)

White Player										
		Depth	1	2	3	4	5	6	Totals	Avg
Inf	w/ em	Fitness: A	-1.29	10.05	13.00	10.73	16.00	6.20	54.69	9.11
		Fitness: O	9.29	-2.05	-5.00	-4.73	-10.00	1.80	-10.69	-1.78
		Score: A	28	43	62	52	48	44	277.00	46.17
		Score: O	36	21	1	11	0	20	89.00	14.83
		Corners	0/4	4/0	4/0	3/0	3/0	2/2	16/6	2.67/1
	w/o em	Fitness: A	-1.29	2.44	11.27	7.91	7.06	-1.78	25.61	4.27
		Fitness: O	9.29	5.56	-3.27	0.09	0.94	9.78	22.39	3.73
		Score: A	28	25	49	42	33	23	200.00	33.33
		Score: O	36	39	15	22	31	41	184.00	30.67
		Corners	0/4	2/2	4/0	3/1	3/1	0/4	12/12	2/2
Gre	w/ em	Fitness: A	11.00	13.00	5.78	11.92	16.00	10.56	68.26	11.38
		Fitness: O	-3.00	-5.00	2.22	-3.92	-10.00	-2.56	-22.26	-3.71
		Score: A	55	62	41	51	48	46	303.00	50.50
		Score: O	9	2	23	13	0	18	65.00	10.83
		Corners	3/1	4/0	2/2	4/0	3/0	4/0	20/3	3.33/0.5
	w/o em	Fitness: A	11.00	11.27	12.33	16.00	10.76	12.82	74.18	12.36
		Fitness: O	-3.00	-3.27	-4.33	-10.00	-2.76	-4.82	-28.18	-4.70
		Score: A	55	49	52	59	47	53	315.00	52.50
		Score: O	9	15	12	0	17	11	64.00	10.67
		Corners	3/1	4/0	4/0	3/0	4/0	4/0	22/1	3.67/0.17
GInf	w/ em	Fitness: A	11.00	11.00	9.78	13.00	11.00	9.00	64.78	10.80
		Fitness: O	-5.00	-3.00	-1.78	-5.00	-5.00	-1.00	-20.78	-3.46
		Score: A	61	56	41	56	55	48	317.00	52.83
		Score: O	1	8	23	8	8	16	64.00	10.67
		Corners	3/0	3/1	4/0	4/0	3/0	3/1	20/2	3.33/0.33
	w/o em	Fitness: A	11.00	0.44	-1.67	12.33	6.05	10.20	38.35	6.39
		Fitness: O	-5.00	7.56	9.67	-4.33	1.95	-2.20	7.65	1.27
		Score: A	61	25	24	52	43	44	249.00	41.50
		Score: O	1	39	40	12	21	20	133.00	22.17
		Corners	3/0	1/3	0/4	4/0	2/2	4/0	14/9	2.33/1.5

E. Minimax, corner detection, killer move detection, blocking, blacklisting and patterns (i.e. all techniques)

Black Player										
		Depth	1	2	3	4	5	6	Totals	Avg
Inf	w/ em	Fitness: A	9.00	5.46	8.56	1.00	11.27	3.00	38.28	6.38
		Fitness: O	-5.00	2.54	-0.56	7.00	-3.27	3.00	3.72	0.62
		Score: A	54	38	46	16	49	32	235.00	39.17
		Score: O	1	26	18	48	15	32	140.00	23.33
		Corners	2/0	2/2	3/1	2/2	4/0	2/2	15/7	2.5/1.17
	w/o em	Fitness: A	9.00	-3.27	-1.00	8.20	10.05	-3.00	19.98	3.33
		Fitness: O	-5.00	11.27	7.00	-0.20	-2.05	11.00	22.02	3.67
		Score: A	54	15	32	44	43	16	204.00	34.00
		Score: O	1	49	32	20	21	48	171.00	28.50
		Corners	2/0	0/4	0/4	3/1	4/0	0/4	9/13	1.5/2.17
Gre	w/ em	Fitness: A	9.00	-10.00	8.20	12.82	13.00	10.37	43.39	7.23
		Fitness: O	-1.00	10.00	-0.20	-4.82	-5.00	-2.37	-3.39	-0.56
		Score: A	54	0	44	53	57	45	253.00	42.17
		Score: O	10	13	20	11	7	19	80.00	13.33
		Corners	2/2	0/0	3/1	4/0	4/0	4/0	17/3	2.83/0.5
	w/o em	Fitness: A	9.00	13.00	9.00	13.00	13.00	10.20	67.20	11.20
		Fitness: O	-1.00	-5.00	-5.00	-5.00	-5.00	-2.20	-23.20	-3.87
		Score: A	54	56	52	55	56	44	317.00	52.83
		Score: O	10	8	10	9	8	20	65.00	10.83
		Corners	2/2	4/0	2/0	4/0	4/0	4/0	20/2	3.33/0.33
GInf	w/ em	Fitness: A	11.00	-10.00	1.80	5.78	9.27	-10.00	7.85	1.31
		Fitness: O	-3.00	10.00	6.20	2.22	-1.27	10.00	24.15	4.03
		Score: A	61	0	20	41	49	0	171.00	28.50
		Score: O	3	13	44	23	15	13	111.00	18.50
		Corners	3/1	0/0	2/2	2/2	3/1	0/0	10/6	1.67/1
	w/o em	Fitness: A	11.00	7.27	11.00	5.46	11.00	13.00	58.73	9.79
		Fitness: O	-3.00	0.73	-5.00	2.54	-5.00	-5.00	-14.73	-2.45
		Score: A	61	49	57	38	55	56	316.00	52.67
		Score: O	3	15	6	26	8	8	66.00	11.00
		Corners	3/1	2/2	3/0	2/2	3/0	4/0	17/5	2.83/0.83

F. Minimax only

White Player								
		Depth	**1**	**2**	**3**	**4**	**5**	**6**
Totals	**w/ em**	**Fitness: A**	34.99	55.85	61.82	65.89	97.02	60.91
		Fitness: O	45.01	18.15	18.18	6.11	-17.02	11.09
		Score: A	330	396	430	429	501	394
		Score: O	310	242	209	165	138	198
		Corners	16/24	22/17	20/20	24/12	30/10	25/11
	w/o em	**Fitness: A**	57.59	64.01	72.46	71.02	90.89	55.60
		Fitness: O	18.41	15.99	7.54	6.98	-10.89	24.40
		Score: A	375	412	445	430	500	383
		Score: O	250	228	195	210	140	257
		Corners	20/18	22/18	24/16	24/16	28/12	22/18
Averages	**w/ em**	**Fitness: A**	3.50	5.59	6.18	6.59	9.70	6.09
		Fitness: O	4.50	1.81	1.82	0.61	-1.70	1.11
		Score: A	33	40	43	43	50	39
		Score: O	31	24	21	17	14	20
		Corners	1.6/2.4	2.2/1.7	2/2	2.4/1.2	3/1	2.5/1.1
	w/o em	**Fitness: A**	5.76	6.40	7.25	7.10	9.09	5.56
		Fitness: O	1.84	1.60	0.75	0.70	-1.09	2.44
		Score: A	38	41	45	43	50	38
		Score: O	25	23	20	21	14	26
		Corners	2/1.8	2.2/1.8	2.4/1.6	2.4/1.6	2.8/1.2	2.2/1.8

F. Minimax only

Black Player								
		Depth	**1**	**2**	**3**	**4**	**5**	**6**
Totals	**w/ em**	**Fitness: A**	54.59	45.87	84.30	104.96	84.35	90.77
		Fitness: O	23.41	32.13	-4.30	-24.96	-4.35	-10.77
		Score: A	384	359	474	495	472	470
		Score: O	256	281	166	145	168	170
		Corners	22/18	20/20	26/14	35/5	28/12	31/9
	w/o em	**Fitness: A**	65.30	78.81	88.27	85.95	79.00	76.64
		Fitness: O	14.70	-0.81	-10.27	-5.95	-1.00	3.36
		Score: A	406	460	499	474	458	440
		Score: O	232	179	138	166	181	200
		Corners	24/16	25/14	26/13	28/12	25/14	26/14
Averages	**w/ em**	**Fitness: A**	5.46	4.59	8.43	10.50	8.44	9.08
		Fitness: O	2.34	3.21	-0.43	-2.50	-0.44	-1.08
		Score: A	38	36	47	50	47	47
		Score: O	26	28	17	15	17	17
		Corners	2.2/1.8	2/2	2.6/1.4	3.5/0.5	2.8/1.2	3.1/0.9
	w/o em	**Fitness: A**	6.53	7.88	8.83	8.59	7.90	7.66
		Fitness: O	1.47	-0.08	-1.03	-0.59	-0.10	0.34
		Score: A	41	46	50	47	46	44
		Score: O	23	18	14	17	18	20
		Corners	2.4/1.6	2.5/1.4	2.6/1.3	2.8/1.2	2.5/1.4	2.6/1.4

G. Minimax, corner detection, killer move detection, blocking, blacklisting and patterns (i.e. all techniques)

White Player								
		Depth	**1**	**2**	**3**	**4**	**5**	**6**
Totals	**w/ em**	**Fitness: A**	90.34	83.68	99.31	126.36	113.64	98.02
		Fitness: O	-14.34	-5.68	-23.31	-48.36	-35.64	-22.02
		Score: A	430	420	463	540	514	470
		Score: O	189	219	174	91	119	157
		Corners	31/7	32/7	35/3	39/0	37/2	31/7
	w/o em	**Fitness: A**	109.52	104.04	122.11	109.53	99.34	103.62
		Fitness: O	-37.52	-24.04	-44.11	-35.53	-39.34	-27.62
		Score: A	467	477	540	491	485	497
		Score: O	142	163	96	120	79	137
		Corners	32/4	36/4	37/2	31/6	29/1	33/5
Averages	**w/ em**	**Fitness: A**	9.03	8.37	9.93	12.64	11.36	9.80
		Fitness: O	-1.43	-0.57	-2.33	-4.84	-3.56	-2.20
		Score: A	43	42	46	54	51	47
		Score: O	19	22	17	9	12	16
		Corners	3.1/0.7	3.2/0.7	3.5/0.3	3.9/0	3.7/0.2	3.1/0.7
	w/o em	**Fitness: A**	10.95	10.40	12.21	10.95	9.93	10.36
		Fitness: O	-3.75	-2.40	-4.41	-3.55	-3.93	-2.76
		Score: A	47	48	54	49	49	50
		Score: O	14	16	10	12	8	14
		Corners	3.2/0.4	3.6/0.4	3.7/0.2	3.1/0.6	2.9/0.1	3.3/0.5

G. Minimax, corner detection, killer move detection, blocking, blacklisting and patterns (i.e. all techniques)

Black Player								
		Depth	1	2	3	4	5	6
Totals	w/ em	Fitness: A	80.33	116.14	119.57	109.98	130.87	108.64
		Fitness: O	-2.33	-40.14	-47.57	-33.98	-56.87	-28.64
		Score: A	423	498	484	530	548	492
		Score: O	216	117	125	104	74	148
		Corners	30/9	35/3	34/2	34/4	36/1	37/3
	w/o em	Fitness: A	109.05	116.45	115.83	114.69	112.88	116.61
		Fitness: O	-37.05	-38.45	-45.83	-38.69	-38.88	-42.61
		Score: A	490	526	529	521	502	516
		Score: O	128	113	83	112	111	111
		Corners	34/2	38/1	32/3	35/3	32/5	36/1
Averages	w/ em	Fitness: A	8.03	11.61	11.96	11.00	13.09	10.86
		Fitness: O	-0.23	-4.01	-4.76	-3.40	-5.69	-2.86
		Score: A	42	50	48	53	55	49
		Score: O	22	12	13	10	7	15
		Corners	3/0.9	3.5/0.3	3.4/0.2	3.4/0.4	3.6/0.1	3.7/0.3
	w/o em	Fitness: A	10.90	11.64	11.58	11.47	11.29	11.66
		Fitness: O	-3.70	-3.84	-4.58	-3.87	-3.89	-4.26
		Score: A	49	53	53	52	50	52
		Score: O	13	11	8	11	11	11
		Corners	3.4/0.2	3.8/0.1	3.2/0.3	3.5/0.3	3.2/0.5	3.6/0.1

Vita

Kevin Cherry was born in June, 1983, in Baton Rouge, Louisiana. He earned his bachelor of science degree in computer science at Louisiana State University in May 2008. In January 2009 he came back to Louisiana State University to pursue graduate studies in systems science. He is currently a candidate for the degree of Master of Science in Systems Science, which will be awarded in May 2011.