Developing an Artificial Intelligence Bot for Othello

Arvind Vijayakumar
Bridgewater-Raritan Regional High School, arvindvjkumar@gmail.com

***Abstract* - In this paper, I describe the Othello AI project completed during the 2014 summer Leap program at Carnegie Mellon University. I will first look at the basic components of two-player game AI. I will then look at basic properties of Othello AI design in the context of the project. Then I will examine the development of the Othello bot from the earliest to the latest versions. The performance of the bot relevant to humans and other bots will be analyzed and discussed. Finally I will look at future possible improvements to *MyPlayer*, the Othello Artificial Intelligence bot.**

Index Terms - STEM, Artificial Intelligence, Game theory, Computer science

INTRODUCTION

Artificial Intelligence (AI) is defined by the Association for the Advancement of Artificial Intelligence as "the scientific understanding of the mechanisms underlying thought and intelligent behavior and their embodiment in machines." As we have further computerized our society, Artificial Intelligence has become more relevant in computing, and its technology is now being applied across a variety a fields from finance, to medicine, to game theory [8]. In particular, this paper will consider AI design for two-player games through the board game Othello.

I. Two-Player Games

Artificial Intelligence has a long history in two-player games. Chess became one of the first popular games to feature computer players. Although initially restricted by computing power, programs began to achieve more and more success against humans in the 80s and the 90s. The 1997 victory of the IBM Deep Blue Chess AI over then world champion, Gary Kasparov, is considered one of the great hallmarks in the field. Software-based AIs have progressed considerably in efficiency since then, and are relatively commonplace on both home and mobile devices. Most common two-player games such as Chess, Checkers, and Othello(Reversi) now have fully functional AI programs [9].

II. Othello

Othello is a two-player game that is classified as a deterministic zero-sum game of perfect information. Perfect information means that each player is capable of knowing all of its opponent's possible moves. This means that there exists an optimal move at every turn for each player. Deterministic means that each player can independently determine this optimal move. Zero-sum means that the gain of one player is exactly equal to the loss of the other. Therefore, there can always be only one winner and one loser (or a tie) [4]. The goal of the game is to have a majority of discs of your color (black or white) on the board. The game is turn-based; each move consists of making a capture by placing one piece of your color on the board. A capture is made by outflanking the opponent, and flipping the pieces in between. If no valid move may be made, then the player passes. The game ends when both players are unable to make a valid move. The game is usually played on an 8 by 8 board, but there are 6 by 6 and 4 by 4 variants [4]. This paper will focus on the 8 by 8 version.

BASICS OF GAME AI

I. Game Tree

Much of the decision-making involved in AI is predicated on the concept of a game or move tree. A game tree is simply a hierarchical structure that details every possible outcome at the end of a turn. In smaller Othello boards, the entire game tree has been solved, which is to say, all possible outcomes at every turn have been calculated. However the 8 by 8 game is still too computationally challenging to be fully solved [1].

II. Search Algorithms

For an AI program to use the game tree, it is necessary to have a search algorithm. A search algorithm searches through every possible move up to a certain level or depth (called ply) from the current level, and returns the best possible move. The best possible move is determined using the scoring heuristic, a topic that will be discussed later on.

III. Minimax Algorithm

The Minimax algorithm is a popular search algorithm often used in two-player board game AIs. Minimax can determine the optimal solution to any game given enough time and memory. It operates on the principle of two players: one minimizing and one maximizing player. The maximizing player is always trying to choose the move that will give him the maximum score (to therefore win the game) while the minimizing player is trying to choose the move that minimizes the maximizing player's score. A Minimax player recursively searches through the game tree, reapplying itself to each additional node it encounters. It keeps searching until it reaches all the "leaves," the nodes that indicate endgame states. At this point it uses an evaluator to evaluate each leaf and assign it a score of win, lose, or draw. The key aspect to this recursive function is that at each consecutive depth or level, it switches from a minimizing to a maximizing player, or vice versa. This is where the key

978-1-4799-1829-4/15/$31.00 ©2015 IEEE

assumption of the Minimax algorithm comes in. The key assumption is that the opponent is equally as intelligent as the other player. This assumption is what allows Minimax to use this recursive switching to essentially predict the opposing player's behavior in order to choose it's own best move. When all scores are calculated, the highest scores for each maximizing player, and the lowest scores for each minimizing player are selected. Unfortunately, in practical use, Minimax algorithms cannot fully follow this form, as it is computationally impossible to fully solve an 8 by 8 Othello game. Therefore the leaves at the last level or depth of the search are considered as roughly accurate approximations of the final result. As a result, it is generally better to have bots search to greater depths when performing a Minimax search [4].

IV. Alpha-Beta Pruning

Unfortunately, programs quickly encounter a combinatorial explosion of nodes as they calculate deeper and deeper levels of the game tree. To make Minimax feasible for two-player games at these higher depths, different forms of optimization are employed. Alpha-Beta optimization is a fairly simple form of optimization that can reduce the necessary time to complete a search by about 30%. Alpha-Beta pruning works on the principle that one player will not make a move if it allows the other player to make an even better move. Therefore, if moves are encountered that result in a worse score than an already guaranteed best score, that node and its pathway is canceled out. In essence, it eliminates calculations on moves and move pathways that will not be influential on the result of the game (given the assumption of equal intelligence in each player). The name originates from the parameters of alpha and beta, which are used to score the best scores for the maximizing and minimizing players respectively [1].

MyPlayer Evaluation

My bot, which was called MyPlayer, employed an evaluator, along with a specialized endgame.

I. Evaluator

As referenced earlier, the Minimax algorithm functions on the basis of an evaluator. As a result, the performance of a Minimax-based program is largely reliant on the quality of its heuristic. The type of heuristic that I used is called a static-board evaluator. This type of heuristic evaluates each board statically without context of depth or chronological position in the game. My earliest heuristics were self-developed (and will be outlined in Development) and involved simple techniques such as a piece count, weighted squares, and a recursion weight. However, as I looked to achieving higher performance I utilized external, more complex heuristics in the bot. The final heuristic used in my bot had already been implemented independently by two different research groups [4]-[6]. As a result, I will not delve as deeply into the specifics of the heuristic's features, and rather into just the conceptual value of each feature. Each of the features are given a weight in the final heuristic sum.

II. Piece Difference

Piece difference is the most basic aspect of the scoring heuristic. The game is essentially won or lost by which side has a greater number of pieces, and therefore it must factor into the heuristic. It is numerically calculated as the percent of total pieces that are of the maximizing color (black) or the minimizing color (white). It is returned as a positive number for black, and a negative for white in accordance with this trend. It equals 0 when there are equal numbers of pieces of each color [4]-[6].

III. Corner Occupancy

Corners are the most powerful positions in the game as once taken, they cannot be flipped. Furthermore, they control the horizontal, vertical, and diagonal lines, which can be lucrative if taken advantage of. It is almost impossible for a player to lose if he/she controls all four corners. Therefore, corner occupancy is given the highest weight in the sum of the heuristic [4]-[6].

IV. Corner Closeness

The positions next to empty corner spaces are considered the worst positions in the game. Taking a position next to an empty corner potentially gives your opponent the ability to take the corner immediately, if not later on in the game. Therefore spaces next to a corner are given a strong negative weight [4]-[6].

V. Mobility

Mobility is an important factor in Othello strategy because if mobility is restricted at a given point, the opponent can force the other player into taking unfavorable moves or passing the turn. Mobility is measured by number of available moves in the given board. Mobility has a positive weight in the final sum of the heuristic [4]-[6].

VI. Stability

Stability is perhaps the most or second-most important factor of Othello strategy because pieces won are only valuable if they can be retained throughout the game. Pieces on the edge of the board are typically very stable because they eliminate at least 3 adjacent positions from which they could be flanked. Exactly how to measure stability is sometimes debated, though the heuristic in question measures it by counting frontier squares. Frontier squares are spots that are adjacent to empty squares, and can therefore be flanked. Boards with a relatively higher number of frontier squares receive lower rankings than board containing a relatively lower number of frontier squares. Stability is given a high positive weight [4]-[6].

VII. Disk Scores

The final element of the researched heuristic involved experimentally determining optimal scores for each position

978-1-4799-1829-4/15/$31.00 ©2015 IEEE

on the board. These scores could be both negative and positive. This feature of the heuristic was calculated as the sum of the scores of all the present positions on the board in question. It was weighted positively [4]-[6].

VIII. Final Sum

The final heuristic is taken as the sum of each of these features multiplied by their individual weights. These weights were determined experimentally by both research groups. This essentially involved running the bot against itself with a variety of different weights as a simulation to determine those that were most effective. The final weighting places the greatest emphasis on corner occupancy (positively), and next-to-corner occupancy (negatively) [4]-[6].

IX. Killer Moves/Endgame

In addition to employing the researched heuristic, my program utilizes my own basic killer move and endgame functionality. The killer move element simply forces the bot to take a corner if it is immediately available. While this strategy is already accounted for in the heuristic, experimental testing with a dedicated killer move method produced better results. In addition, I created an endgame aspect which makes the bot resort to a simple piece count as it reaches final game state leafs in the game tree. Using an endgame yielded improvements when compared to earlier versions.

DEVELOPMENT

The time frame of the project was a little over two weeks, and involved a great deal of personal effort outside of the regular amount of time allotted by the Leap program. The bot was developed in Java, under a framework provided by my Teaching Assistant (TA). That framework is used annually in a CS class at Caltech for their final project of creating an Othello bot.

I. Early Versions

Early in development, the largest amount of time was used for research. I had no prior experience in creating board-game AIs and therefore I had to do extensive research into the subject. It took me a couple of days to fully understand and get familiar with the provided framework. I finished the first version of the bot after 3-4 days. It was a simple program that looked at the available moves, and picked one at random. The next version involved further work in understanding the framework, but was also very simple, as it used a piece count static-board evaluator at only the current level. I was easily able to defeat both of these bots in test games. After these early versions, I began to study Othello strategy more extensively, and produced programs that took into account position weighting. For example, where a regular piece was given a weight of 1, a corner was given a weight of 40. These programs were able to defeat the basic sample players provided to me by my TA, but not the more complex programs. At this stage, I personally was still able to consistently defeat my program. As a result, I began looking into implementing the Minimax algorithm.

II. Minimax Implementation

Implementing the Minimax algorithm was possibly the most difficult aspect of the Othello program. Research for developing a complete understanding constituted the majority of the time. Eventually, I was able to distill its ideas into logic, and turn it into an early version. This started a lengthy bug-fixing period, though after a couple days of frustration, it finally began to work. I was able to search up to 3-ply or 4-ply using my Minimax implementation. Using the Minimax algorithm along with my basic disk square method proved relatively more successful as *MyPlayer* was then able to defeat slightly more advanced sample programs. It even defeated me on the first try, though never again after that, as I picked up its strategy fairly quickly. The ineffectiveness of these static evaluators soon led me to consider employing a recursion weight, which would therefore provide context to the scoring during the Minimax game tree search. This method also helped progress the bot, though the bot was still unable to consistently defeat me.

III. External Heuristic

At this point in the development, I turned towards external heuristics. After some research, I was able to find a strategy employed by multiple previous research projects. I have described the strategy at length in the previous section. With this heuristic working correctly my bot was finally able to defeat me. There was a certain element of euphoria and accomplishment when that happened. However, I continued to work on the bot.

IV. Alpha-Beta Pruning and Other Slight Improvements

I implemented a basic form of optimization in Alpha-Beta pruning. In fact implementing Alpha-Beta was a fairly simple modification on top of the previous Minimax implementation. I used a slightly modified form of the Othello framework that saved the board as a type long variable rather than a bit set. With these modifications in place, I was able to practically use 6-ply, though 7-ply never became realistically feasible. I was able to implement an endgame at an experimentally determined point in the game, along with a basic killer move function that targeted corners. With all these improvements, and optimizations in effect, I tested my final bot against both humans and other bots. In its final form, my bot has defeated all human challengers, and has only lost to one other bot known as *DeepKwok* that is the number one competition bot used by TAs in the Caltech course on Othello. *DeepKwok* has been refined over the years. The best bot that my bot was able to defeat was a program called *OogeePlayer* which last year ranked 9th out of 55 competing bots in the end-of-course Caltech Othello bot tournament. Aside from *DeepKwok* my bot did not face any higher ranked bots other than *OogeePlayer* from that year's particular tournament. Another bot from the tournament, *yellowmamba*, bears striking similarities to my

978-1-4799-1829-4/15/$31.00 ©2015 IEEE

own bot. As described by one of its creators, *yellowmamba* used a heuristic that accounted for piece difference, corners, squares adjacent to corners, mobility, frontier discs, and a static evaluation of the squares on the board. It employed a recursive min-max algorithm with alpha-beta pruning and searched to depths from 6-9 ply. *yellowmamba* secured first place in the tournament (among student created bots, so *DeepKwok* which did better, does not qualify). Given the similarity in design between *MyPlayer* and *yellowmamba*, I think my bot has a solid design foundation for an Othello AI program [10].

FUTURE IMPROVEMENTS

Although I am happy with the performance of my bot insofar, I am already working on implementing improvements.

I. Iterative Deepening

Iterative Deepening is a simple modification that decides the depth of the search based off a given time argument. Using this feature therefore enables the program to reach the deepest feasible depth at each turn. Implementing this feature is currently in progress [4].

II. Transposition Table

Arguably the most difficult feature to implement, a transposition table is a data structure that stores each calculated position and its evaluated score using a Zobrist hash. By doing this, the program can eliminate the time used to recalculate redundant positions (as there are often multiple ways to reach the same position) and can save approximately 25% to 50% of the search time in the midgame [7]. Implementing this feature is currently in progress.

III. Opening Book

An opening book is a pre-calculated set of optimal opening moves given certain scenarios that help set the game on a favorable path. These move sequences are usually above the grasp of the general-purpose heuristic used throughout the game [5]. An opening book would be very useful, but would add considerably to the size of the program due to the addition of a large move database.

IV. More Advanced Algorithms

There exist more advanced search algorithms such as NegaScout and MTD(f) that make it possible to search to much greater depth. NegaScout is an optimization to the alpha-beta optimization of Minimax. MTD(f) is an advanced search algorithm that uses null-window searches and transposition tables to converge on the optimal Minimax value [3]. These algorithms are long-term goals to implement after the completion of the aforementioned three.

CONCLUSION

The Othello AI project was a successful project that produced a high-level Othello bot. Developing the bot helped improve my skills in logic, strategy, and programming. Perhaps most uniquely, the experience helped me gain appreciation for algorithms. It taught me that thinking and understanding is the real challenge, and that programming is the tool by which understanding is applied.

My experience has also had a very influential bearing on my college and potential career choice. It strongly influenced my choice to select computer science as my major as I applied to college. Creating an Othello program gave me a glimpse of the mathematical underbelly of artificial intelligence. I am now very interested in pursuing further studies in Artificial Intelligence once in college, particularly in algorithms.

I understand that there are many better-developed Othello programs that are far superior to mine. An example is Michael Buro's Logistello [2] that employs statistical analysis in addition to static board evaluators, and is better than all existing logical programs. However, I still take pride from my accomplishment, and can truly say that the Othello AI project has been one of my proudest achievements in high school.

REFERENCES

[1] Andersson, Gunnar. "Writing an Othello Program." Gunnar Andersson's Homepage. April 2, 2007. Accessed August 15, 2014. http://www.radagast.se/othello/

[2] Buro, Michael. An evaluation function for Othello based on statistics. Technical Report 31, NEC Research Institute, 1997.

[3] Chen, Jack. "Applications of Artificial Intelligence and Machine Learning in Othello." Computer Systems Lab at Thomas Jefferson High School of Science & Technology. June 16, 2010. Accessed November 28, 2014. http://www.tjhsst.edu/~rlatimer/techlab10/Per5/FourthQuarter/ChenPaperQ4-10.pdf.

[4] Korman, Michael. "Playing Othello with Artificial Intelligence." 2003. Accessed August 6, 2014. mkorman.org.

[5] MacGuire, Steve. "Strategy Guide for Reversi & Reversed Reversi." Samsoft. April 1, 2011. Accessed September 5, 2014. http://www.samsoft.org.uk/reversi/strategy.htm.

[6] Sannidhanam, Vaishnavi, and Muthukaruppan Annamalai. "An Analysis of Heuristics in Othello." 2004. Accessed August 7, 2014. cs.washington.edu.

[7] Zobrist, A.L. A new hashing method with application for game playing. Technical Report 88, Univ. of Wisconsin, 1970.

[8] Mackworth, Alan K. "AAAI Is Now the Association for the Advancement of Artificial Intelligence." The Association for the Advancement of Artificial Intelligence! March 1, 2007. Accessed August 7, 2014. http://www.aaai.org/Organization/name-change.php.

[9] Smith, Chris, Brian McGuire, Ting Huang, and Gary Yang. "The History of Artificial Intelligence." *History of Computing*, no. December 2006 (2006): 10. Accessed September 14, 2014. cs.washington.edu.

[10] Qu, David. "Othello." David Qu. August 16, 2014. Accessed September 14, 2014. http://users.cms.caltech.edu/~dqu/othello.html.

978-1-4799-1829-4/15/$31.00 ©2015 IEEE

AUTHOR INFORMATION

Arvind Vijayakumar, Senior at Bridgewater-Raritan High School. Arvind Vijayakumar is the Webmaster of the local FBLA (Future Business Leaders of America) chapter.

978-1-4799-1829-4/15/$31.00 ©2015 IEEE