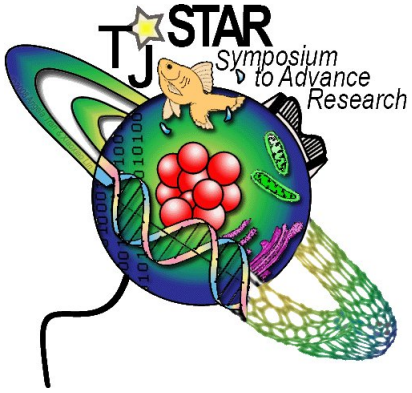

The Implementation of Artificial Intelligence and Temporal Difference Learning Algorithms in a Computerized Chess Program

James Patrick "Enter the Dragon" Mannion
Thomas Jefferson High School for Science and Technology
Alexandria, Virginia

February 19, 2009

Abstract

Computers have developed to the point where searching through a large set of data to find an optimum value can be done in a matter of seconds. However, there are still many domains (primarily in the realm of game theory) that are too complex to search through with brute force in a reasonable amount of time, and so heuristic searches have been developed to reduce the run time of such searches. That being said, some domains require very complex heuristics in order to be effective. The purpose of this study was to see if a computer could improve (or learn) its heuristic as it runs more searches. The domain

used was the game of chess, which has a very high complexity. The heuristic, or evaluation function, of a chess program needs to be able to accurately quantify the strength of a players position for any instance of the board. Creating such an evaluation function would be very difficult because there are so many factors that go into determining the strength of a position: the relative value of pieces, the importance of controlling the center, the ability to attack the enemys stronger pieces something that chess masters spend entire lives trying to figure out. This study looked to see if it was possible for a computer program to learn an effective evaluation function by playing many games, analyzing the results, and modifying its evaluation function accordingly. The process by which the program improved its evaluation function is called Temporal Difference learning.

1 Introduction

Heuristic searches (such as the A*) in general can be applied to practically any domain that somebody would want to search through. The actual heuristic functions used, however, are extremely domain-specific. Some problems require very simple heuristics (or estimations that allow a program to make good choices that will lead to the best outcome without actually knowing the entire search tree), such as a problem whose domain involves points on a flat surface that are connected by paths, in which the shortest path from point A to point B is desired. Such a problems heuristic would simply be the distance of a hypothetical straight path from the current point on the path to point B. By adding this estimation to the total distance traveled along a path thus far and comparing it to the total distances and estimated remaining distances of other possible paths, a search program can find the optimum solution to this problem quite easily. This is an example of a domain with a very simple heuristic to calculate. However, many problems that researchers are interested in nowadays have much more complex domains, and therefore much more complex heuristics. The age-old game of chess is one such problem.

In 1997, IBM created a chess program called Deep Blue that beat the world champion chess player at the time. Deep Blue used a brute-force approach to chess. It would look at every possible move, and then every possible move that could result after each of those original moves, and each third possible move, and so on and so on to about 6-12 moves into the future but in some cases up to as many as 40 moves. And it would calculate all

of this before it even made the first move. Each time it would search all the possibilities and make the move that would give the highest chance of checkmate down the road. Chess champion Garry Kasparov was defeated by Deep Blue. Most chess masters can look at their deepest about 10-12 moves into the future for a few possible paths (certainly not all of them), but no human has the brain power to play out every possible path out to 6 or more moves in their head before making each decision. However, IBM needed a state of the art supercomputer to run this program at a reasonable speed. Running it on anything less than a supercomputer would take days or even weeks for the game to finish. On average, there are about 30 moves that a player can make at any given time. This means that each turn, Deep Blues brute-force algorithm would have to search through at least 30^6 moves, or about 7.29 x 10^8 moves. It is quite clear that using such a method on a machine other than a supercomputer would simply be impractical.

It is possible to write a computer chess program that is still effective without using the brute-force method. By looking only, say, two or three moves into the future rather than 10 moves or more, a program can still make educated decisions about good moves to make if it has some way of estimating how strong a projected position would be. This is where a heuristic, or evaluation function as it is more commonly called in the context of board games like chess, comes in handy. By looking a few moves into the future, applying the evaluation function to each possible board, and choosing a move based on which projected board has the highest strength of position, a computer can still be an effective chess player, while at the same time dramatically cutting down the number of required calculations.

It sounds simple enough, but once you actually try to create such an evaluation function, it becomes much, much harder. How do you effectively evaluate a position? Do you just look at the number of your pieces versus the number of their pieces? Do you look at whether or not your pieces control the center? Do you look at the possibility of sacrificing one of your own pieces in order to capture a more important piece? An evaluation function can be very complicated to formulate, especially in a game such as chess where there are so many strategic factors to take into account. Chess masters spend entire lifetimes figuring out the best way to evaluate which moves they should make, so at first creating such a function could seem very daunting, especially for someone who has not devoted years to learning the game of chess.

But if chess masters learn how to decide which moves to make by playing a lot of games and learning from their mistakes, why couldnt a computer

do the same thing? In fact, computers, although they lack human intuition, can play full games of chess in a matter of seconds, so it is conceivable that they could learn how to effectively evaluate moves much faster than a human could. Therein lies the purpose of this study: to see whether or not a computer can be programmed to not only play chess effectively and efficiently, but also learn from its mistakes, like humans do, and get better from game to game. The chess program I have developed uses Temporal Difference learning to analyze the games it plays and modify its evaluation function accordingly. This means that theoretically, the program could start out by making random moves, but then modify its evaluation function so that it progresses from simply choosing a random move to actually making an educated choice.

2 Background

In 1950, Claude Shannon wrote a groundbreaking paper called Programming a Computer for Playing Chess. It dealt with various issues, such as how one might go about writing an artificial intelligence program that could play chess well, and what the evaluation of such a program might look like. It discussed the problems involved with brute-force chess algorithms and suggested an AI algorithm that a computer chess program could use. It suggested using a 2-ply algorithm (an algorithm that looks two turns into the future before applying the evaluation function), however if the program finds that a move could lead to a check or checkmate, then it would investigate that move and subsequent moves out to as many as 10 turns. At the end of the paper, Shannon speculated about the possibility of computer programs that can learn, but noted that such developments were probably man years down the road.

About 50 years later in 1999, D.F. Beal and M.C. Smith published a paper called Temporal difference learning for heuristic search and game playing. At this point, programs that could play chess had been well-developed, but research in machine learning was really just beginning. The paper investigated the use of Temporal Difference learning as a way of making a computer chess program modify and improve its evaluation function each time it plays. The researchers made their chess program learn for 20,000 games, and then ran their program against a program that had not learned its evaluation function. The learned program performed decisively better over 2000 games than

the unlearned program, showing that it is indeed possible for computers to improve their evaluation functions.

Similar studies have been done in other areas. In 2007, Shiu-li Huang and Fu-ren Lin used Temporal Difference learning to teach two agents how to effectively negotiate prices with each other so that agreements that maximize payoff and settlement rates for both parties. TD learning was shown to be very effective for this purpose. One of the reasons TD learning was chosen from the many different types of learning for this project was its usefulness in different domains. If it proves to be effective in chess, in addition to bargaining, then it could very well be effective for any problem in the very general and vast field of artificial intelligence.

Researchers David E. Moriarty and Riso Miikkulainen from the University of Texas at Austin used a different type of machine learning called evolutionary neural networks to create programs that could play Othello. They found that even though their programs were given no information about the game, they eventually learned very complex Othello strategies that have only been mastered by high-level human Othello masters. It is very interesting that a computer would naturally learn strategies in a matter of days that took humans years and years to conceive. In this project I will analyze the fruits of TD learning to see if well-known chess strategies naturally emerge over the course of testing.

3 Development

Python was used to code this chess program. The first stage of the program simply involved a console-based chess game where two humans could input their moves into the command line, and the board would be re-printed with the new move, as shown in Figures 1-3.

This program underwent numerous debugging sessions to make sure that it would correctly handle illegal moves, checks, checkmates, castling, the obscure en passant pawn capture, and other game play mechanics. For the sake of simplicity, stalemates were called after 200 moves (100 turns) without checkmate.

Stage two of the project involved writing simple AI players. The first AI player written was one that simply made a random legal move. The next AI player written was simply a three-ply minimax search with alpha-beta pruning. Its heuristic function was such that it would chose moves based

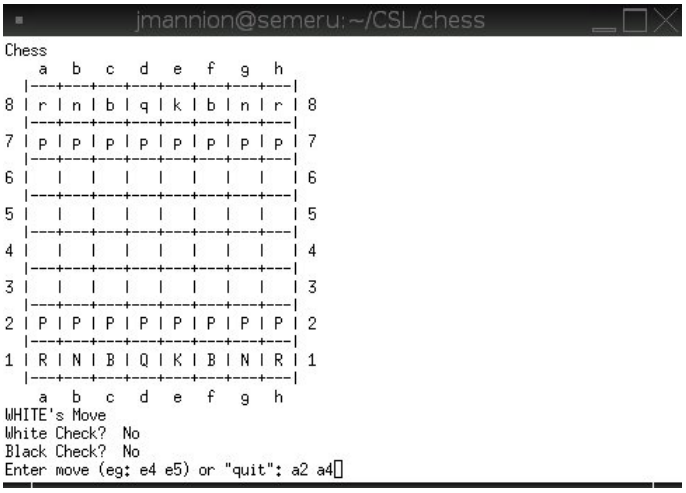

Figure 1: The starting board, with white about to move a pawn.

on a simple piece differential, with each piece being weighted the same as every other piece, regardless of position on the board. These two programs were created for the purpose of comparing the progress of a player with TD learning against a constant player over time.

Stage three introduced temporal difference learning. TD learning compares predictions of future board states to actual values seen later on in the game and adjusts the weights of the heuristic function in order to make predictions made in later games closer to observed values. As more and more games are played, the weights of the heuristic function will improve and move toward equilibrium values. The more in-game values that are observed, the more accurate and more effective these equilibrium values will become.

The heuristic function is set up as follows:

$$h = c_1(p_1) + c_2(p_2) + c_3(p_3) + c_4(p_4) + c_5(p_5) \tag{1}$$

In this equation, c (each initially set to 1) represents the weight given to each type of piece (1=pawn, 2=knight, 3=bishop, 4=rook, 5=queen) and p represents the sum of the values of the spaces being occupied by each type of piece. The following is used to calculate p:

$$p_i = \sum w_i \sum b_i \tag{2}$$

Here, w_i is the value associated with the square occupied by the piece of type i, and $\sum w_i$ is the sum of these values for every white piece of type i. The

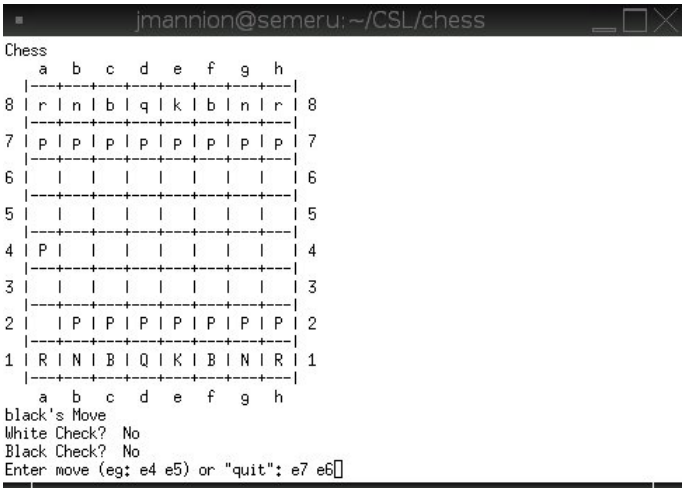

Figure 2: White pawn moved, black about to move a pawn.

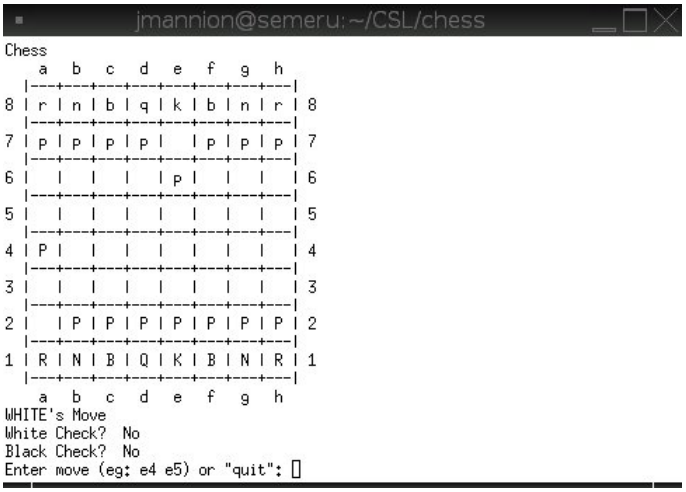

Figure 3: Black pawn moved.

same sum for the black pieces is subtracted from the sum of white pieces. The values for w and b are found by looking at the corresponding piece-square table which is stored in a text file. For example, the initial piece-square table for pawns would look like this:

1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1

This means that initially, pawns are valued the same regardless of where they are on the board, which would result in p_1 simply being the number of pawns white has minus the number of pawns black has.

As the program learns, the coefficients c_1 through c_5 will be adjusted, as well as the values in the piece-square tables. This means that board states will be evaluated based on the relative numbers of pieces each player controls, their respective locations on the board, and the value of each type of piece with respect to the other types of pieces.

4 Testing and Analysis

The program with TD learning was run against a static program. All weight values in the heuristic were initialized as 1. The win-loss differential of the learner was tracked over 2,000 games to see if the learner would start winning more and more as it underwent the learning process.

The altered piece-square tables were analyzed after the learning session was complete to see if the program developed generally accepted strategies such as maintaining control of the center. Higher numbers toward the center of the boards would indicate such a strategy.

5 Conclusion

References

[1] Shannon, Claude E. "Programming a Computer for Playing Chess". 1950.

[2] Beal, D.F. and Smith, M.C. "Temporal Difference Learning for Heuristic Search and Game Playing". 1999.

[3] Moriarty, David E. and Miikkulainen, Risto. "Discovering Complex Othello Strategies Through Evolutionary Neural Networks".

[4] Huang, Shiu-li and Lin, Fu-ren. "Using Temporal-Difference Learning for Multi-Agent Bargaining". 2007.

[5] Russell, Stuart and Norvig, Peter. *Artificial Intelligence: A Modern Approach*. Second Edition. 2003.