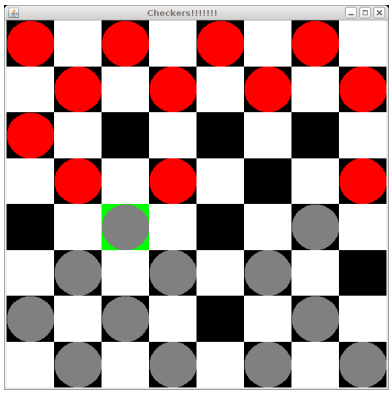
The Implementation of Machine Learning in Checkers



Abstract
Most games have a set algorithm that does not change. This means that these programs cannot adapt to a situation or learn from mistakes that it makes. However if a machine could learn, then it could adapt to new situations and would have a nearly boundless skill level. Machine Learning programs can be beaten once, but against an opponent that did not change, it eventually be able to beat it. The project that I am writing will learn how to play the game of checkers as it plays, by modifying itself after ever game played. It will review its play and if it played well it will play that way more often, if it did not it will avoid that way of playing.

Background

Most programs that play games today use a search through a tree of moves and boards. Each different possible move and the board that would result from that move is evaluated and when all of the boards have been searched an evaluated; the program chooses the move that results in the best outcome quickest. This is called the minimax algorithm. Each player wants to act in their own interest, to either maximize or minimize an evaluation function called a heuristic. The heuristic determines which boards are good and which ones are bad, without being able to perfectly predict the results of the game. The minimax algorithm looks at all of the possible boards that can be reached from one state and then looks at all the boards that can be reached from another state, and so on and so on. At some depth the algorithm stops and then applies the evaluation function to determine the fitness of a board. At each level, the player whose turn it is will make the move that results in the best outcome for them. This algorithm is limited by two things, which are the depth of the decision tree that it searches, and the quality of the heuristic function. However this can become very costly in terms of time very quickly. There is not enough time to search through the entire decision tree for all but the simplest games. The play of the AI increases in skill as you can more perfectly predict the future states of the board.

One method of creating a learning program is to have the program learn by rote. An experience base stored all of the boards encountered and the moves that had been tried off of these boards. While this is similar to creating all of the possible board combinations and simply searching them exhaustively, this does not require as much memory and is used in conjunction with a static evaluation function. A different method of creating a learning program is described by Olsen (2), generalization learning. The program modifies the heuristic function each time it plays based on the outcome of the play. The program starts with a heuristic function that has all of the features weighted equally. After the program plays a game, it will change the values of the weights of the heuristic function based on the outcome of the game. This type of learning when implemented in a program, generally played very well during the middle game play, but less well in beginning and end game play, while the rote learning process played very well in end game play.

Development

I am using a combination of the learning by rote algorithms and the generalization algorithms. I will create a file and store all of board combinations that have been seen and the moves taken from that particular board. I will then also use a changing heuristic to evaluate each board. The heuristic will also learn based on the outcome of the game. Since the capability of these methods changes based on where in the game you are, I will attach more weight onto the predictions of the learning by rote method in the beginning and end game and more weight to the generalization algorithm during the middle game phase.

For the learning by generalization algorithm I will use a temporal difference learner. Temporal difference learning adjusts the evaluation function during play based on the difference between the evaluation function at one point and the evaluation function at a different time. This will bring the evaluation function into a state of equilibrium toward the ideal evaluation function. Every time another move is made, the heuristic is changed based on the previous heuristic. The equation for this is $U(s) \leftarrow U(s) + \alpha(R(s) + \gamma U(s') - U(s))$. s is the current state of a system, U is the evaluation function of state s, α is the learning coefficient and decreases as the number of times state s has been encountered increases and R is the reward that you will get for state s.

The learning by rote system that I am using also uses the minimax algorithm. But instead of learning a better and better heuristic function, it stores a set of boards that it has seen that are most probable to be played. The program then uses this to increase the depth of the minimax search because the time it takes to access a stored board is much less than the time it takes to find all of the moves that can be made. This increases the skill of the AI by increasing the number of turns the AI can look into the future.