Applications of Artificial Intelligence and Machine Learning in Othello

Jack Chen
TJHSST Computer Systems Lab 2009-2010

Abstract

This project explores Artificial Intelligence techniques in the board game Othello. Several Othello-playing programs were implemented and compared. The performance of minimax search algorithms, including alpha-beta, NegaScout and MTD(f), and of other search improvements such as transposition tables, was analyzed. In addition, the use of machine learning to enable AI players to improve play automatically through training was investigated.

Introduction and Background

Othello (also known as Reversi) is a two-player board game and abstract strategy game, like chess and checkers. I chose to work with Othello because it is sufficiently complex to allow significant exploration of advanced AI techniques, but has a simple set of rules compared to more complex games like chess. It has a moderate branching factor, larger than checkers and smaller than chess, for example, which makes advanced search techniques important without requiring a great deal of computational power for strong play. Although my AI programs are implemented to play Othello, most of the algorithms, data structures, and techniques I have investigated are designed for abstract strategy games in general instead of Othello specifically, and many machine learning algorithms are widely applicable to problems other than games.

The basic goal of an AI player is to consider the possible moves from the current game state, evaluate the position resulting from each move, and choose the one that appears best. One major component of an AI player is the static evaluation function, which heuristically estimates the value of a position without exploring moves. This value indicates which player has the advantage and how large that advantage is. A second major component is the search algorithm, which more accurately evaluates a state by looking ahead at potential moves.

Search Algorithms

The minimax search algorithm is the basic algorithm to search the game tree. Minimax recursively evaluates a position by taking the best of the values for each child position. Alpha-beta pruning is an extremely important improvement to minimax that greatly reduces the number of nodes in the game tree that must be searched. As shown in the figure below, alpha-beta is far faster than minimax, especially at high search depth.

NegaScout and MTD(f) are enhancements of alpha-beta that use null-window searches, which result in many more cutoffs than wide window alpha-beta searches. Null-window searches provide only a bound on the minimax value, so repeated re-searches may be necessary to find the true value. The chart below compares the performance of NegaScout, MTD(f), and alpha-beta.

Training

Initially, my static evaluation function's feature weights were set manually based on human strategy, and hand-tuned somewhat with a manual hill-climbing process. However, this process is slow and ineffective. A much better way to set feature weights is to use machine learning to automatically train the static evaluation function by optimizing the feature weights.

I divided the game into 59 stages, each stage representing positions with a certain number of total pieces from 5 to 63, and trained a separate set of weights for each stage with a gradient descent method.

The trained weights for some of the more important features over each game stage are shown below. The corners and adjacent squares are extremely important, especially early in the game, while moves become increasingly important near the end of the game. Frontier squares are consistently weighted slightly negatively.