Semester Project - Learning Game Board Evaluation

Allison Regier, Shawn O'Neil

October 6, 2006

1 Introduction

As we see in the field of minimax game playing, coming up with evaluation functions for particular game states can be a daunting problem. For instance, in a simple game of pawns (a game where two players have a number of pawns and try to reach the other side of the game board), the value of a board is clear near the end of the game where depth limited minimax search is able to search to the end of the game. However, this is not the case for early game states. This indicates that for some games, even very simple ones, strategy can be very subtle.

We propose developing a reusable framework for an evaluation function which learns over time, by playing many games, which board states are more valuable and which are less valuable. With any luck, the same framework would be able to learn a number of games (though, we obviously wouldn't train a tic-tac-toe player and then try and get it to play othello.) In any case, we should get a good feel for different reinforcement learning techniques, and how they may be applied (or can't be applied) to tricky problems such as ours, when reinforcement must be delayed.

2 Related Work

Michael Gherrity did a similar project for his PhD thesis (which can be found at http://www.gherrity.org/) called SAL, which can learn any two-player board game using a back-propagating neural network. Also, Tesauro's TD-Gammon system successfully learned backgammon using temporal difference learning.[1] We may look to his paper as a resource, especially the learning methodology section.

One interesting approach similar to ours, undertaken by a class at Columbia University, applied the paradigms of genetic algorithms for evolving othello playing strategies.[2] This paper covers not only the methods implemented, but also follows the students in the class on their course.

Additionally, there is a Reinforcement Learning Tutorial on the web that may be useful.[3] Topics covered include both Q-Learning and game playing.

3 Proposed Solution

We plan to use some form of reinforcement learning in order to make our agent a better game-player by training against itself. At a high level, each agent will perform a minimax search in order to choose its next move. As a black-box, the evaluation function will have the same functionality as in our hand-coded solution: given a description of a board state, it will return an expected utility value of that board state, regardless of previous or future board states. However, inside of the black box, the mechanism will be different. Instead of us hand-coding an evaluation function, the evaluation function will be trained using reinforcement. In order to make our learning evaluation function extensible, we will require that it be possible to represent board states in an easily indexable form; thus we will only test simple board games of perfect information.

One goal of the project is to train the evaluation function based on the features of board states rather than on individual boards. We feel that this will be more interesting, as many games will have large state spaces, and thus the probability that any particular board state is seen multiple times is small. By allowing the evaluation function the ability to generalize over board features, it should be able to make useful decisions about board states it has not seen before, because it will be able to make connections between certain features of similar board states, even if they are not identical. One task in the project will be to come up with a general set of features to describe board states. There will likely be very many such features. Note that one will not necessarily be able to reconstruct the entire board state only based on its feature description, although this may be possible.

A particular wrinkle this introduces is that we are limited to games which have a certain generalizable set of features; thus we will limit ourselves to games which take place on a square board, where each position can be occupied by either player's piece (which are unique, eliminating chess) or no piece. Tic-tac-toe, othello, and the pawns game are all examples of this.

Our evaluation function (which will be implemented as a Java class) will learn in the following way: Upon instantiation, the evaluation function believes that each possible value of each feature has only a default utility. In other words, every possible board state would have equal utility. When asked to evaluate a board state that contains feature values it has never seen before, it will return an initial constant. As it sees additional values of board features, it will add them to an internal database. Whenever asked to find the utility of a board state, the evaluation function will look up all of the features contained in the state description and somehow combine the utilities of the individual feature values to come up with an overall utility of that board state. As a particular game is played, the function will keep a list of all the board states seen in that game. When a game ends, if the agent tells the function that it lost, it will update the expected utility values for the features of the last several board states it observed during the game by decrementing their utility values. Alternatively, in a win situation, the utilities will be increased with respect to who is playing. Note that during self-play, one of the agents will be playing for "Max" and one of the agents will be playing for "Min". So, if Max won, we want to increase the utility values of the feature values we just observed. This will simultaneously punish the loser, Min, for whom smaller utility values are actually better.

Overall, we will implement this using a Java minimax game playing system. We intend to play two evaluation functions against one another, update them as necessary, and repeat. We will test the trained function against a hand-coded evaluation function every X number of games, to see if the evaluation function is improving with training, and if so, how fast it is improving.

This methodology includes a number of variables we can adjust, including the depth of the minimax search, and the number of last seen board states' features to update in the evaluation database. We will need to experiment with different methods of updating the utility values. For example, one idea is to use some kind of distribution to make the update change the utility values of more recent board states more than the board states toward the beginning of the game.

4 Implementation Details

Our proposed solution will consist of a number of Java classes. An agent class will control the minimax playing algorithm. A main class will control the training games, as well as periodically evaluating our players' performance against a hand coded evaluation function (which will perhaps make random poor choices occasionally, to give our players a chance so that we can measure relatively untrained functions.) Also, this agent class will give reward and punishment signals to their evaluation functions at the end of each game.

The game rules (ie, initial board state, terminal states, and possible move generation function) as well as the state of an individual board will reside in a board class particular to each game type.

The evaluation function will be implemented as a Java class as well. The main class will have two instances of this at any time: one for the white player and one for the black player. It is important to note that the white player's evaluation function will have to also evaluate positions in which it is black's turn, and vice versa. Let's consider the situation at the end of a particular 10 move game in which White won, and we want to reward its evaluation function by updating just one feature (for simplicities sake) for the last few boards. (How many boards back to update and in which way to update them are questions fundamental to our problem, which we'll have to address.) White has a list of features it has seen throughout the game: $F_1 3$ ... $F_8 = 4$ $F_9 = 4$ $F_{10} = 8$. White might have corresponding utilities over these last three values for the feature of $3, 3, 5$. (For instance, if the feature is "number of white pieces on the board" then white considers having 4 white pieces on the board worth 3, and 8 white pieces worth 5.)

Notice that if on the last board it was White's turn, then on the second to last it was Black's, and so on. Because White is playing Max, in order to reward the evaluation function we want to increase the utilities over the features, which, because Black is playing Min, is the same as punishing the utilities from Black's perspective. Thus, after we copy the changed evaluation function over so Black can play with it, it is also a stronger evaluation for Black. This also leads to the observation that rewarding a Black win means decreasing utilities over the features - because Black is playing min.

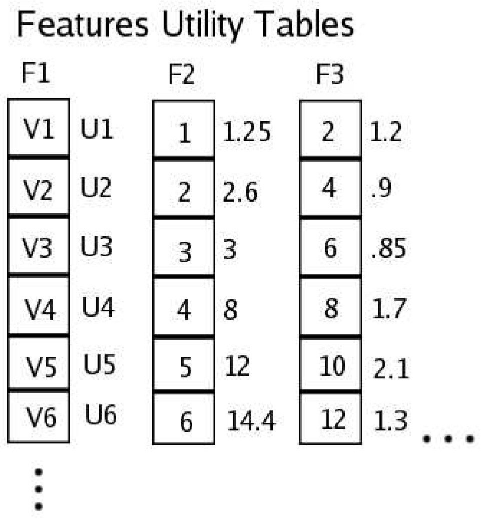

Figure 1: Illustration of dynamic tables used for feature utilities.

The evaluation class will have a number of dynamically updated tables corresponding to utilities for each feature. 1 Initially, however, we will likely implement our evaluation class to consider utilities for whole board states, rather than the features. Adding code for features (the interesting part of the whole exercise) should be a relatively small (but perhaps tricky) job, code wise. In fact, considering utilities for complete board state descriptions is merely a special case of evaluating feature-based board state descriptions. In this case, the complete board state is the only feature.

5 Milestones/Timeline

Initially, we will need to develop the depth limited minimax search framework and adapt it to our purposes of training evaluation functions. This work has already been started, as part of the chapter 6 project.

Secondly, we will need to develop one or more evaluation classes which implement the learning function. We will need to experiment at this point and find a good way of updating the utility values for all (or part) of the observed board states in a game that the agent has won or lost. As stated before, we plan to initially evaluate board states using only one feature- the complete board state description. Then, we will need to come up with a generalized set of independent features to describe a board state. This learner should be much more flexible than the one that treats the complete board state description as a single feature.

Also, at some point we will develop at least two other games which satisfy the game type requirements. As we already have one game class developed, this will leave us with three games in total to run our evaluation on. All games should be trained and tested using the same set of features to describe the board state.

The final stage will consist primarily of training, testing, and collecting data about different configurations, searching for the most optimal way to actually train our evaluation function.

6 Evaluation Tools/Techniques

We would like to evaluate the progress of our evaluation functions as they are trained at periodic intervals in some quantifiable way; perhaps by having the function play against a hand coded function a number of times and recording the percentage of wins accumulated. Doing so will give us an idea of the number of training games which must be run to get to a particular skill level.

References

[1] Tesauro, G. (1995). "Temporal difference learning and TD-Gammon" Communications of the Association for Computing Machinery, 38(3), 357-390. Also at http://www.research.ibm.com/massive/tdl.html

[2] E. Eskin, E. Siegel, “Genetic Programming applied to Othello: introducing Students to Machine Learning Research” The proceedings of the thirtieth SIGCSE technical symposium on Computer Science Education, 1999. Pages 242-246.

[3] M. Harmon, S. Harmon “Reinforcement Learning: A Tutorial” http://www.nbu.bg/cogs/events/2000/Readings/Petrov/rltutorial.pdf