Bowron Abernethy Chess Engine

Christopher W. Bowron Robert D. Abernethy IV

December 7, 2000

Abstract

In this paper we will discuss the game of chess and how the AI community has attempted to solve the problem of creating an agent that is capable of beating human grand masters. We will also discuss our chess agent, BACE, and describe the specific algorithms that we implemented. We will focus our attention on two main aspects of BACE - searching and learning. The searching algorithm consists of a modified version of the alpha-beta search called NegaScout. Our learning algorithm is a variation on the TD(λ) technique. A discussion of our experiments and results will follow our description of BACE. Finally, we will state our conclusions and contemplate future work.

1 Chess

The game of chess is a good domain for practicing artificial intelligence methods. This is mainly due to the environment. Chess consists of a set of states, from each state a set of legal actions are possible. The game can end in one of three outcomes - win, lose, or draw. The chess environment can be characterized as follows:

- Accessible - all aspects of the game are perceivable by the agent
- Deterministic - the actions available are completely dependent on the current state
- Non-episodic - actions taken in the game will affect later actions
- Semi-static - the environment does not change while an agent is thinking, however, the amount of time the agent has does.
- Discrete - there are a finite number of possible actions at any given state

A simple strategy for building a chess agent would be to begin at the initial state, examine all the possible moves, and search each move and every possible counter-move until a path is found to a win state. This is indeed a simple strategy; however, such a brute force method is not feasible due to the large branching factor at each state. Thus, AI strategies must be used. These strategies view the game of chess as one large tree and they attempt to search through the tree as efficiently as possible.

2 History

The history of chess in the AI community dates back about fifty years when Claude Shannon distinguished between two different types of chess agents. The first type of agent, Type-A, referred to agents that implemented a brute force method. Type-B, on the other hand, described those agents that took into account some chess logic and only examined a subset of moves at any given state. Since the 1950's, great strides have been occured in the AI community. In May 1997, a brute force agent named Deep Blue defeated the human world champion, Gary Kasporov.

Although there are many chess agents around, we wish to discuss one in particular due to the influence it had in the development of BACE's learning algorithm. In Baxter, Tridgell, and Weaver's paper "Learning to Play Chess using Temporal Differences", they discuss a chess agent called KnightCap. What makes KnightCap interesting and different than most chess agents is its ability to learn its evaluation function. It does this by using a modified version of Sutton's TD(λ) algorithm.

The TD(λ) algorithm had been used previously (and with great success) by Tesauro's backgammon agent, TD-Gammon. However, simply implementing the algorithm in a chess agent isn't practical because chess requires a deeper search through the tree. In TD-Gammon the search is very shallow - one to three ply. To be competitive in chess it is necessary to search as deep as at least five or six ply. Therefore, Baxter, Tridgell, and Weaver needed to make some modifications to the general algorithm. The major modification was to have the TD algorithm execute on the leaf states of its searches, rather than the actual positions that occurred during the game.

3 BACE

3.1 Overview

BACE is a fairly simple implementation of chess. It is designed to work in one of three modes: independently, with the X interface XBoard, or run remotely on a server using RoboFics. The core chess engine was written by Christopher Bowron. The learning algorithms were implemented by Christopher Bowron and Robert Abernethy. Table 2 in the appendix contains locations of the files.

While being a relatively simple implementation, it has a number of powerful features, including transposition tables, NegaScout search, move ordering, and a temporal difference learning algorithm.

3.1.1 Board Representation

The basic board representation in BACE is an array of 64 integers. Each array item is a mapping between a chess piece and an integer between 0 and 13. The high order bits of the integer represent the piece. 0 represents an open square, 1 represents pawn, and so on, with 6 representing a king. The lowest order

bit represents the piece colors. 0 represents white, 1 represents black. This representation was chosen to allow for quick testing of a squares occupancy as well as determine if the piece are friend or foe.

Along with the basic square information, some additional information is updated dynamically. One such feature is the flags integer which consists of 12 bits enumerating castling rights, en passantes, double pawn pushes, castling information, and promotional information.

An array representing the number of pawns on each file is also kept dynamically, as well as the material each side has, a positional evaluation of each sides pieces, an integer representing the current board hash value, the number of pieces on the board, an array containing the number of each individual piece on the board, and also the squares on which the kings reside. This information is kept to facilitate evaluations, move generations and move legality checking.

3.2 Search Algorithms

Implemented in BACE are many different variants of the negamax search algorithm. By default, BACE uses the NegaScout algorithm invented by Prof. Dr. Alexander Reinefeld which is a modified alpha-beta pruning technique which greatly reduces the number of nodes searched. The main idea is to search with a null window in most of the nodes and only with a wider window where it is necessary. If the search requires the full window the position must be re-searched. Thus, move ordering and storing encountered positions are very important when using NegaScout.

The basic NegaScout algorithm is:

```
int NegaScout ( position p; int alpha, beta );
{
   int a, b, t, i;

   determine successors p_1,...,p_w of p;

   if ( w == 0 )
      return ( Evaluate(p) );                    /* leaf node */

   a = alpha;
   b = beta;
   for ( i = 1; i <= w; i++ )
   {
      t = -NegaScout ( p_i, -b, -a );
      if (t > a) && (t < beta) && (i > 1) && (d < maxdepth-1)
         a = -NegaScout ( p_i, -beta, -t );      /* re-search */

      a = max( a, t );
      if ( a == beta )
         return ( a );                             /* cut-off */
```

```
      b = a + 1;                        /* set new null window */
   }
   return ( a );
}
```

Also implemented is a generic alpha-beta pruning search, as well as a full negamax search.

After doing a full search to the specified depth, BACE uses a quiescent search to find positions that are relatively stable to facilitate better evaluations.

BACE uses an iterative deepening search algorithm on top of the other search. The search depth begins at the previously stored depth based on the transposition table entries. The search is made at the current depth, then increased and searched again, and so on, until the search limit is reached. The search can be limited by time, depth, or both.

An aspiration window can also be set up but was not used in the course of these experiments. Using the aspiration search, The alpha and beta values are initialized to the evaluation of the current position minus the value of two pawns for alpha, and plus two pawns for beta. If our search results in a number between those bounds, it is valid. If it lies outside that range, we must search again with the full window. After each iteration, the alpha and beta values are updated based on the evaluation at the previous depth.

3.3 Move Ordering

Move Ordering is very important when using alpha-beta pruning techniques. Therefore, BACE uses a number of different criteria for sorting moves.

(To scale numeric values, assume that a pawn has a value of 1,000.)

The first criteria used in ordering moves is the difference in material value. A bonus is given for captures of 1,000 plus 300 times the ratio of capturer to capturee. Thus, the bonus for a pawn capturing a queen is very high, and a queen capturing a pawn is much lower. Non-capture moves are initialized to zero.

Bonuses are awarded to the base result in the following way:

- If the move has been stored as a refutation of the last move, a 2,000 point bonus is awarded.
- If the position is stored in the transposition table, a 1,500 point bonus is awarded to the stored move for that position.
- If the move has been stored as the primary 'killer move' for the current ply, a 1,000 point bonus is awarded.
- If the move has been stored as the secondary 'killer move' for the current ply, a 500 point bonus is awarded.
- Finally, an array called history is kept that is maintained as a count of the number of times a move resulted in the changing of the alpha-beta

bounds. The value of that array for this move is added to the value of this move.

Once each move has been evaluated according to these criteria, it is sorted using a QuickSort algorithm.

3.4 Book

BACE uses the book from Marcel's Simple Chess Program, written by Marcel van Kervinck. The book consists of around 800 openings, of which BACE will select the lines consistant with the current game and randomly select one line for its next move. Once there are no longer available openings, BACE will begin using its searching algorithms for move selection.

3.5 Transposition Tables

By default, BACE uses a transposition table of 2^20 entries. Each entry consists of an integer hash value, an integer depth, a flag, a score and move. The flag is used to signify whether the value stored is an upper bound, a lower bound, or the exact result of evaluation this position searching to the specified depth. The transposition table greatly reduces the number of positions that need to be searched, because once an encountered position is reached, the score from the table can be used rather than continuing the search. Positions are located in the table by indexing into the transposition table array based on the lower order bits of the current board hash value. If the value stored in the hash value in the table is equal to the current board hash value, it is assumed that the stored position is the same as the current position and the table can be used.

3.6 Evaluation Features

The evaluation function uses a number of different criteria. The score is first computed for white. Points are added for each white feature on the board, and subtracted for each black feature. If the evaluation is used to determine how well black is doing, the result is negated.

Each piece has a weight associated with it. Originally, a pawn was 1000, a knight 3250, a bishop 3500, a rook 5000, and queen 9000. The king was not given a value because the loss of a king is the loss of a game.

Each piece has an associated array of 64 integers. Each entry in the array is a bonus for that piece to be on that square. This array promotes the advancement of pawns, and central control for knights, bishops, rooks, and queens.

Checkmate and stalemate are not recognized by the evaluation function, rather they are evaluated inside the search function when a side to move is found to have no valid course of action. However, if there is not enough material for either side to mate, the draw is recognized by the evaluator. If one side does not have potential mating material and the other does, the losing side is penalized by a weight called "NOMATERIAL".

Table 1 contains the a summary of the additional evaluation features.

Table 1: Evaluation Features

KCASTLEBONUS	Bonus for having castled on the king's side.
QCASTLEBONUS	Bonus for having castled on the queen's side.
NOCASTLEQUEEN	Penalty for not having castled, and losing the right to castle queen side.
NOCASTLEKING	Penalty for not having castled, and losing the right to castle king side.
QUEEN_TROPISM	Bonus for having queen close to opponent's king. This bonus is multiplied by the minimum of the difference of the queen's rank and the opponents king's rank and the difference of their files.
ROOK_TROPISM	Bonus for having rook close to opponent's king. This bonus is multiplied by the minimum of the difference of the queen's rank and the opponents king's rank and the difference of their files.
DOUBLEDROOKS	Bonus for having two rooks on the same rank or file.
OPENFILE	Bonus for having a rook on a file that contains no friendly pawns or opponent's pawns.
SEMIOPEN	Bonus for having a rook on a file that contains no friendly pawns.
TWOBISHOPS	Bonus for having two or more bishops.
KNIGHT_TROPISM	Bonus for having knight close to opponent's king. The bonus is calculated as this number multiplied by the addition of the rank difference and the file difference.
ISOLATED	Penalty for pawns that have no friendly pawns on the adjacent squares.
DOUBLED	Penalty for having multiple pawns on a file. Scored once for each pawn on the file.
BACKEDUP	Penalty for a pawn that is behind adjacent friendly pawns.
NOMATERIAL	Penalty for not having enough material to mate. This is defined as
SEVENTH_RANK_ROOK	Bonus for having rook on its seventh rank. That is white rooks on the seventh rank and black rooks on the second rank.

3.7 Learning Algorithm

The learning algorithm is a modification of TD(λ) known as TDLeaf. It was introduced by Baxter, et. al. The main difference between TD(λ) and TDLeaf is that the leaf nodes that were actually evaluated are those on which the temporal differences algorithm is used.

To implement TDLeaf, we modified our quiescent search to save the board of positions that evaluate between the alpha and beta bounds. After the search is completed BACE checks to see if the result of the search is the same as that stored. If not it checks to see if the value stored is the same as the previous value stored. If it is, BACE will use the stored board from the previous evaluation. If neither of these boards are correct, BACE checks the principle variation returned to see if it leads the the correct board evaluation. If so, it is stored, if not the position is ignored when calculating the temporal differences. These steps were necessary because the use of the transposition tables means that sometimes searches are not actually carried out fully to the leaf nodes. We used this approach to decrease the amount of necessary overhead for board storage and copying.

At the end of the game, each weight was updated according to the TD(λ) algorithm (see 1). For our experiments we used a λ value of 0.70 and a learning rate, α of 0.70.

$$w := w + \alpha \sum_{t=0}^{n} (E_{t+1} - E_t) \sum_{k=0}^{t} \lambda^{t-k} \nabla_w E_t \tag{1}$$

To decrease the fluctuations encountered in evaluations, we used the tanh function to smooth our values. The temporal differences used where based on the values of $\tanh(\beta E_t)$ where β was a value chosen so that an evaluation of up one pawn was equal to .25.

The actual code used to implement the TD learning was as follows:

```
for (w = PAWN_VALUE + 1; w < LAST_WEIGHT; w ++)
{
    for (p = 1; p <= (n-1);p++)
    {
        int j;
        double S2 = 0.0;

        for (j = 1; j<= p; j++)
        {
            double grad =
                    (1.0 - tanhvector[j] * tanhvector[j]) *
                    EVAL_SCALE * gradients[w][j];

            S2 += pow(TD_LAMBDA, p-j) * grad;
        }
        delta[w] += TD_ALPHA * d[p] * (S2 / EVAL_SCALE);
```

```
    }

    weights[w] += delta[w];
}
```

The $1 - tanh^2(\beta E_t)$ was used instead of directly calculating $sech^2(\beta E_t)$ because it was faster to use the values we had already computed.

4 Experiments

During the development and debugging stage, BACE was tested using self-play to ensure that the values arrived at through the temporal differences was reasonable. Once we were fairly certain things were running smoothly, we reset the weights to their original states. No experiments were done to compare how a self-taught BACE compared with with an online taught BACE, although this may be an interesting experiment to run.

We placed BACE on the Free Internet Chess Server (freechess.org) with its initial weights. BACE was set to accept only rated 5 minute games. Initially, its rating was around 1300. Before implementing the learning features its rating was around 1400-1500.

5 Results

With our learning algorithm, BACE's rating rose from around 1300 to a maximum of 1660, over a period of 24 hours, playing approximately 200 games. For comparison, Master level rankings begin at 2,000. Figure 1 contains a graph of BACE's performance.

6 Conclusions

We feel that BACE performed reasonably well, despite the fact that it's rating dropped after implementing our learning algorithm. Having a static evaluation function with predetermined weights, BACE was able to achieve a rating of approximately 1500. After implementing our learning algorithm, its rating dropped down to about 1300. We expected this initial drop due to the fact that BACE would not be able to search as far as previously because of the increased overhead that was necessary to store the extra board positions needed by our learning algorithm. This overhead took up valuable time and limited the depth of our searching algorithm. We predicted an increase in our rating as the number of games played increased and our agent was able to learn a more accurate evaluation function. As expected BACE's rating increased but was limited by the small set of evaluation features.

7 Future Work

Although BACE enjoyed modest success, we believe that some minor changes would be beneficial. One possible adjustment would be to add more evaluation features. BACE uses very few features in its evaluation compared to many other engines. Another possible improvement would be to move the currently hard-coded positional arrays into the weights that are learned based on temporal differences. This could have a significant effect on BACE's positional play. The way in which features are updated by temporal difference make it easy to add more features.

Another possible approach likely to increase level of play would be to split the evaluation weights into more than one set of values. We believe that separating weights into two, or possibly three stages, beginning, middle and endgame may be useful.

Once the weights begin to converge, it may be a useful feature to learn which openings BACE is successful with and which it is not. This could be done by recording information on which openings have led to wins, and selecting those with a higher probability in the opening moves.

A Related Work

1. Tridgell, A., Baxter J., & Weaver L. Learning To Play Chess Using Temporal Differences,*Machine Learning, 40*,243-263.

2. Sutton, Richard. Learning to Predict by the Methods of Temporal Differences. *Machine Learning, 3*,39-44. Richard Sutton.

3. Samuel, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development, 3*, 210-229.

B File Locations

BACE	*http://www.cse.msu.edu/ bowronch/FILES/bce.tar.gz*
XBoard	*http://research.compaq.com/SRC/personal/mann/xboard.html*
RoboFics	*http://www.freechess.org/ hawk/robofics.html*
mscp	ftp://ftp.freechess.org/pub/chess/Unix/mscp-1.0.tar.gz

Table 2: file locations

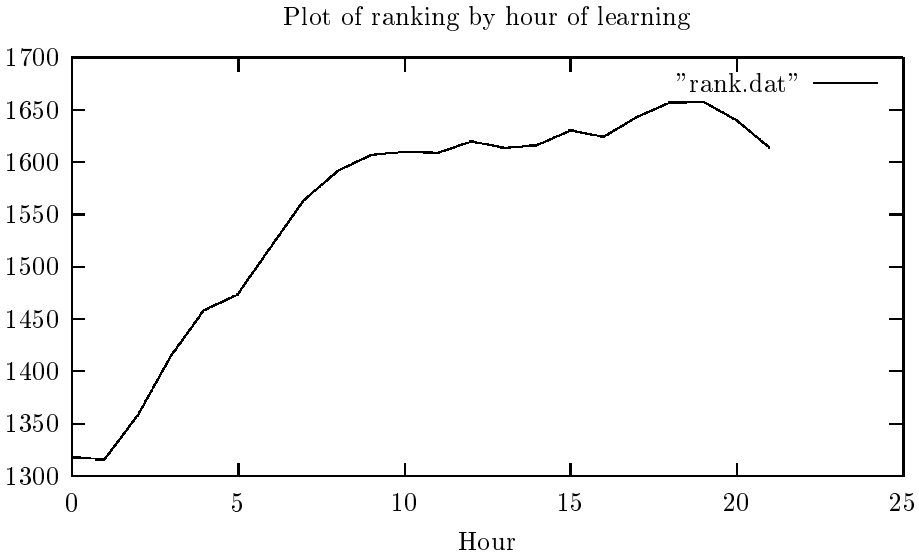

Figure 1: graph depicting rating fluctuation