Balrog - An AI Chess Engine

CS221 Fall 2013: Homework [p-final]
SUNet IDs: [arijitb, aparnak]
Team Members: [Arijit Banerjee, Aparna Krishnan]

1 Introduction

Game playing is one of the most important areas in AI. Chess is a very popular two-player strategy board game. In the context of AI, chess falls under the category of adversarial games where the opponent tries to reduce your chance of winning as much as possible. Being a classic example of a two-player zero sum game (sum of the utilities of agent and opponent is zero), the game of chess provides a natural platform for testing new advances in the field of Artificial Intelligence.

2 Task Definition

For our project, we have built a Universal Chess Interface compliant chess engine. The UCI protocol is an open communication protocol that enables a chess engine to communicate with the graphical user interface controlling it. Hence, implementing the UCI protocol abstracts out the work related to graphics and makes a chess engine testable using existing GUIs like Arena and PyChess.

3 Related Work

The state of the art computer chess programs right now can convincingly beat their top human counterparts and have been doing so for several years (beginning with the famous victory of IBMs supercomputer Deep Blue against Garry Kasparov in 1997). The strongest chess engines right now (including Stockfish, Houdini, Rybka and a few others) have an ELO rating of around 3200 [1], while the current chess World Champion, Magnus Carlsen, has an ELO rating of 2872 (a gap of 400 rating points means that the stronger player is ten times as likely to win as his opponent). So in a sense, the game of chess can already be considered solved. However, judging by the increase in ELO ratings[2] of the top chess engines over the past few years, there is still plenty of room at the top.

An important aspect of a chess engine is leaf node evaluation. As search depth cannot be increased beyond a point, a good evaluation function is needed to decide the value of a given board position. Sacha Droste and Johannes Frnkranz[1] in their paper 'Learning the piece values for three chess variants' follow a reinforcement learning approach to determine

[1] http://www.computerchess.org.uk/ccrl/4040/rating_list_all.html
[2] http://en.wikipedia.org/wiki/Elo_rating_system

piece values and piece square values. We follow a similar approach to calculate these values along with weights for new hybrid features.

4 Approach

Chess is a two-player zero sum game. To formalize, there is a fixed start state, various actions that can be taken at each state, a deterministic successor for each action taken in a state and several end states. The end state has a utility value associated with it based on whether it is a win, loss or draw. This game structure can be represented as a tree with the nodes representing the states and the edges representing actions. In this sort of two player game, each alternating level of nodes corresponds to a player turn. This is the overall structure of the game.

The task of building a chess engine can be broken down into the following subtasks:

- implementing UCI protocol
- having an efficient board representation
- move generation
- a search algorithm over the game tree
- leaf node evaluation

We describe the methods and algorithms we have used to implement the above tasks below.

4.1 UCI Protocol and GUIs used

We have implemented a subset of the UCI protocol[3] for our chess engine. Our engine can now interpret commands from the GUI telling it to set up the board at a particular position, find the best move at a particular position, and output its current board representation. This allows us to test our engine on already existing chess GUIs. Pychess and Arena are free and the easiest to obtain and use on Linux and Windows respectively (we used these for our project). However, both of them have a few limitations - Pychess lacks fine grained time control options and an opening book, while Arena crashes on move 48 every now and then. Better GUIs such as Fritz exist, though they are not free.

[3] http://wbec-ridderkerk.nl/html/UCIProtocol.html

4.2 Board Representation

Choosing a good board representation is important for fast move generation. While advanced representations such as bitboards[4] exist which use roughly one bit per square of the board, we chose to represent our chess board as an array of pieces. The board state comprises the current board structure which gives information about the pieces present on the board as well as their positions. In addition to this, the board state has the information corresponding to player turn, castling rights and en passant squares.

Although this is a bit wasteful in terms of memory, this allowed us to develop faster and concentrate more on the AI aspects of our project rather than on optimizing bit operations.

As a consequence, the amount of nodes our engine could search through in a second was slightly reduced. While our engine can look at about 20000 nodes a second (we talk about search speed a bit later in this report), the fastest engines right now can look at about a million nodes every second.

After implementing a board representation and the UCI protocol, we could interpret an **FEN string** (a standard notation for specifying chess positions) from the GUI and setup the board at a given position.

For instance, the command
'position fen rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1'
would tell our engine to setup its board at the initial position of a chess game.

4.3 Move Generation

The board state has all the information required to generate the moves. There are two steps involved in move generation:

- generating all the moves possible for all the pieces on the board for the player
- choosing only the legal moves from the list of moves generated (a move is legal iff the king is not in check after the move)

Move generation is a bit tricky as it requires checking for pawn promotions, possible en passant captures and determining the validity of castling at a given position.

To test the validity and performance of move generators, there exist several Perft benchmarks online, which list the total number of legal states reachable from a given position at different depths. For instance, there are 4,865,609 positions reachable from the initial position of the chessboard at depth 5, 347 of which are checkmates. We did Perft testing on the positions mentioned in the site[5] and verified that our move generator was indeed working correctly.

[4]https://chessprogramming.wikispaces.com/Bitboards
[5]http://chessprogramming.wikispaces.com/Perft+Results

4.4 Search

The move generation step outputs all the valid moves corresponding to a given board state. The best move has to be chosen from the set of valid moves. Since chess is a finite two player zero sum game, it can theoretically be solved using a Minimax Search.

4.4.1 Minimax Search

Chess can be formally represented as shown.

Players = White, Black
States = Pieces of white and black on the board along with the positions and player turn
Actions(state) = All legal moves possible for the state based on chess rules
IsEnd(state) = Is the state checkmate or a draw (computers never resign)
Utility(state) = +1 if agent wins, -1 if opponent wins, 0 if draw

Since the search depth can be very high, we cut the minimax search off at a certain depth and perform a static evaluation of the state at that depth and approximate it as the utility. We use a slightly modified version of Minimax, called Negamax algorithm.

$$V_{opt}(s,d) = \begin{cases} Utility(s) & \text{if } IsEnd(s) \\ Eval(s) & \text{if } d = 0 \\ \max\limits_{a \in Actions(s)} V_{opt}(Succ(s,a),d) & \text{if } Player(s) = agent \\ \max\limits_{a \in Actions(s)} -V_{opt}(Succ(s,a),d-1) & \text{if } Player(s) = opponent \end{cases}$$

Here $V_{opt}(s)$ is the utility when both agents play optimally.

If no legal moves are possible at a position, the search ends at either a checkmate or a stalemate.

This is a complete search algorithm, that is, all nodes upto depth d are looked at. After implementing this algorithm, our engine took a few seconds to search to depth 3 and about a minute to reach a depth of 4. This is quite slow, and we needed to optimize our algorithm so that it could search deeper.

4.4.2 Alpha-Beta pruning

In order to make Minimax search faster, we implement pruning and restrict the number of nodes looked at. The core idea of alpha-beta pruning is based on the branch and bound principle. As we are searching (branching), we keep lower and upper bounds on each value we're trying to compute.

If when choosing between two options whose intervals dont overlap non-trivially, then we choose the interval containing larger values.

4.4.3 Move ordering

The speedup provided by alpha beta pruning is related to the move ordering chosen. Good moves at a position are more likely to generate beta cutoffs. In our implementation, we considered pawn promotions and captures as more likely to be good moves and searched along these moves first.

There are more complicated move ordering heuristics that we did not implement, such as using heuristics to decide which moves are more interesting among non-capture moves as well as ordering the captures themselves (for instance, it makes sense to capture the most valuable piece of our opponent with our least valuable piece).

After alpha-beta pruning and move ordering, our engine was able to reach a depth of 5 within a few seconds. It could now play at an amateur level. However, it still made occasional blunders related to not being able to look deep enough.

4.4.4 Quiescence Search

The main problem with a fixed depth search is the so called 'horizon effect'. Assume that our algorithm can search upto a depth of 5. Consider a position where, with best play, a rook is lost after 5 moves. Suppose by sacrificing a bishop, the computer can push the loss of the rook two more moves. This is a worse move than giving up the rook, since it leads to the loss of both pieces. However, because of the horizon effect, the computer is not able to see this and plays a worse move. This is the problem that quiescence search tries to solve.

We implemented a quiescence search for our engine by extending our search at leaf nodes until the position became quiet. We define quiet in our implementation as the absence of capture moves. Hence, when we reach depth 0 in our search, instead of returning the leaf evaluation, we call a quiescence search which continues the search until no captures are possible.

Alpha beta pruning can be performed here as well, with an alpha value equal to the initial static evaluation of the board. Note that this assumes that after playing a move our position will atleast be better than our current position. This could fail for the so called **'zugzwang'** positions of chess, where the side to play is at a disadvantage. Luckily, these positions are not very common in chess (they are a bit more common in endgames), and our algorithm worked well in practice.

After implementing a quiescence search, the strength of our engine suddenly increased significantly. At this point, our engine stopped making tactical blunders and was calculating upto 10-15 moves deep in some variations (although the highest depth at which it looked at every possible sequence of moves was still 5). With a simple piece value evaluation, our engine was now playing at a moderate level of strength and could easily beat beginners.

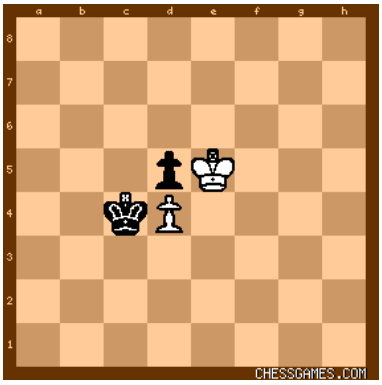

Figure 1: The 'trebuchet' position: both sides are in zugzwang, i.e: the side to play loses

Below is the algorithm we used for the quiescence search.

```
def Quiesce(alpha, beta):
    eval = static_evaluation()
    if (eval >= beta):
        return beta

    alpha = max(alpha, eval)
    for every move M which is a capture:
        make move M
        score = -Quiesce(-beta, -alpha)
        undo move M

        if (score >= beta):
            return beta

        alpha = max(alpha, score)

    return alpha
```

4.4.5 Iterative Deepening

So far, we have not talked about time controls. It is often a good strategy for the computer to search till depth d and update its best move before searching to a depth of 'd+1'. This is because in the event that it runs out of time or is forced to play a move (by the user controlling the GUI), it still has a good move to play. This is the method of iterative deepening, and we implemented it for our engine.

In practise, a strategy to speed up the search is to give the moves calculated in the main line for a depth 'd' search higher priority while calculating the move ordering for a depth

'd+1' search. This leads to faster beta cutoffs while pruning. Although we calculated the main line at each depth, we did not implement the rest of this strategy.

4.5 Evaluation of board state

In chess, as the utility of the game is only at the end state (win, loss or draw), we should search till the end state is reached. This is impossible with the computing resources that are available. Hence, we fix a depth, search till that depth and evaluate the board state at that depth to get an approximation of the utility. Having a good evaluation function is very important to make the correct moves.

We use a straightforward evaluation function of the form

$$P(x; w) = \sum_{f} w_f . f(x) \tag{1}$$

where,

$f(x)$ = feature extracted from the board state x

w_f = weight associated with the corresponding feature f

4.5.1 Feature Extraction

$f(x)$ are elementary feature values that are computed for a given board position x, and w_f are the weights corresponding to the feature. These weights are automatically tuned in the learning phase. We refer to features as f omitting the board state x for brevity.

Three types of features are considered:

- **Material**: For each piece type pawn, knight, bishop, rook and queen, the material feature for that piece is the difference in the number of pieces of that type for White and Black. For a given board state, there are 5 features of type Material.
- **Piece-square tables**: For each piece type and for each square on the board, the piece-square feature is an indicator feature of whether the piece is on that square. For a given board state, there are 6*64 = 384 features of this type.
- **Combined piece values**: This feature is to show the combined values of pieces - eg. is a rook better than a bishop and a knight, are 2 rooks better than a queen and a pawn. It is calculated by finding the difference in number of pieces of the described types between White and Black.

4.5.2 Learning feature weights

Once the features for a particular board state are extracted by the method described in 4.5.1, the weights corresponding to the features are tuned using TD Learning. To restrict

Pawn	Knight	Bishop	Rook	Queen
0.983327	3.499627	3.034167	4.060071	8.077552

Table 1: Piece values

the value of the evaluation function between 0 and 1, we use a log linear evaluation function.

$$V(x;w) = \frac{1}{1+e^{-P(x;w)}} \tag{2}$$

The chess engine plays games against itself and we use the game logs to learn the weights.

$w_f \leftarrow 0; \forall f$
For every (s, a, r, s) [state, action, reward, next state]
$w \leftarrow w - \eta[V(s;w) - (r + V(s;w))] \bigtriangledown_w V(s;w)$

This is the general equation for weight update in TD Learning. We performed a few experiments to get the best weights for the features

- $r = 0$ when s is not end state and $r =$ game result if s is end state
- $r =$ game result for the last few plies and $r = 0$ for the remaining steps.
- $r =$ game result for all the steps

Initially, we performed learning using a dataset comprising 1575 games[6] of the current world chess champion Magnus Carlsen. However, this method yielded poor results. On analysis of the games, we noticed that games at the grandmaster level of chess do not consist of very unequal positions and hence are not suitable for TD learning (we dont know that a rook is more valuable than a pawn since grandmasters often resign long before one side has a large difference in material). Hence, we made our chess engine play games against itself and used these games to compute the weights. This approach let to more accurate piece values and piece square tables.

5 Experiments and Results

1. After TD-learning, we learned piece values, piece square tables and values of features involving pairs of pieces. The values of pieces we learned are shown in Table 1. These values are similar to the established values in modern chess theory (1, 3, 3, 5, 9 respectively).

   The piece-square table values we learned for a pawn are shown in Table 2.

[6]http://www.pgnmentor.com/files.html#players

0	0	0	0	0	0	0	0
4.57882	4.81033	3.75259	4.98593	4.85371	3.72536	4.55165	4.48902
1.13791	3.43121	1.78167	2.82176	2.49351	3.81822	2.21291	1.97927
3.68113	3.0492	4.33818	1.83413	0.667213	0.799824	4.95316	4.33214
1.69611	2.83684	3.18282	1.71422	2.46621	2.44414	5.11124	2.5963
0.53815	0.167995	3.1618	2.65402	0.146033	0.219336	2.33031	0.0910465
0	1.99697	2.98606	2.30225	0.0147418	0.00524247	0.000342101	5.79535e-06
0	0	0	0	0	0	0	0

Table 2: Piece square table for pawn

These values also make sense, since pawns have a higher value in higher ranks as they get closer to promotion. In practise, for our engine, we slightly increased the values of pawns at the center (control of the center is an important chess maxim). We also made the values symmetric.

Finally, our hybrid feature weights for (Q + N, Q + B) were (12.4, 13.3) which suggests that a queen and bishop is more valuable than a queen and a knight. This is a surprising result as chess theory suggests the opposite.

2. With quiescence search, our learned piece values and piece square tables (massaged a bit to make pawns stay near the king, knights move away from the edges and to make kings prefer castling), we had a strong chess playing engine.

   We tested our engine, Balrog, against other chess engines such as Pychess and GNUchess. It was able to defeat the default level of Pychess most of the time, while it lost to GNUchess (a very strong engine which plays at the grandmaster level). However, estimating the ELO of our engine turned out to be difficult since we could not find the ELO rating of the Pychess engine online.

   Before the poster session, we tested Balrog against a few human opponents with known ratings. It was able to beat players with an ELO about 1500 at blitz time controls but lost to a player with an ELO of 1700. Many of its losses were due to a faulty endgame technique, where it was unable to convert a winning advantage of a few pawns.

   With this data, we estimate the ELO rating of our engine to be around 1600 at Blitz time controls (where both players have 5 minutes to think). This is approximately equal to the skill of a mid level chess player. At the poster session, Balrog defeated everyone it played against except for an opponent with an ELO rating of 1900.

   Our engine searches to a depth of 5 in a few seconds (while reaching a depth of more than 10 along some variations often due to quiescence search) and examines about 20000 positions per second. It performs iterative deepening and can give the main line (the best moves assuming self play) upto the depth it has searched so far.

   Our engine's evaluation of a chess position on the Arena GUI is shown in Figure 2.

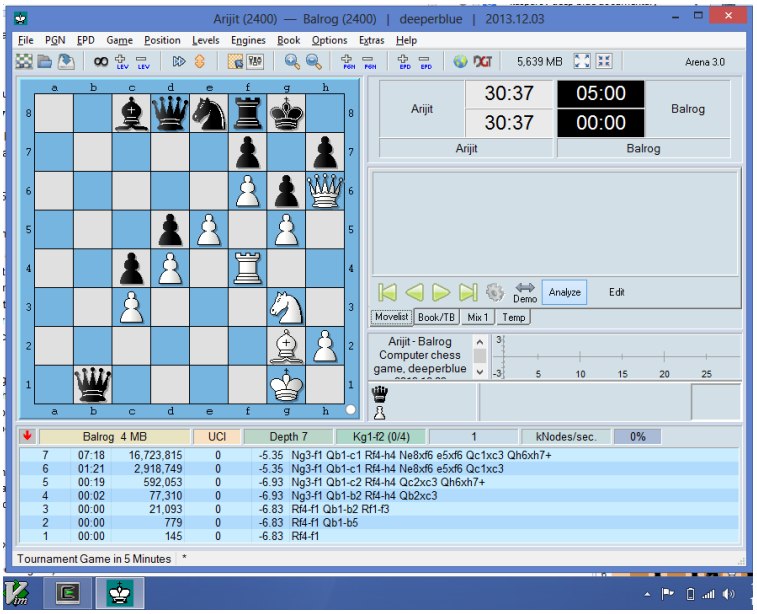

Figure 2: Anand-Carlsen 2013 Game 9, Evaluation by Balrog

6 Conclusion and Future work

Our chess engine performs well in Blitz time controls and is estimated to have an ELO rating of **1600**. There are many improvements which can be implemented. An efficient board representation using bitmasks can increase the number of nodes searched per second drastically and hence bring down the time required for search. Further, search could be made faster by using better move ordering by ordering the captures. Use of transposition tables (hashing the best moves for game states) can also make search faster.

The current evaluation function we use does not include subtle positional considerations and hence using a better evaluation function comprising of features such as open files, passed pawns, the bishop pair and piece mobility will give better results. Rather than a static evaluation function, a position specific evaluation function can also be implemented (to take into account cases like endgame positions where the king is suddenly a strong piece and should move to the center rather than stay hidden in the corner).

Finally, we have open sourced our engine (named **Balrog**) on GitHub at chess-bot in the hope that it will prove useful for people interested in computer chess.

References

[1] Sacha Droste and Johannes Frnkranz; Learning the piece values for three chess variants; International Computer Games Association Journal; 2008