Design & Implementation of a Chess Engine

Dennis Ideler, Viktor Stremler
{di07ty,vs05uc}@brocku.ca
Computer Science Department, Brock University

January 11, 2010

Abstract

We offer an overview of designing and implementing a computer chess engine. Although our actual end product is a fully interactive & visually pleasing three dimensional chess game, in this paper we focus on the artificial intelligence aspect of computer chess. After introducing a generic chess engine architecture, we examing each engine component in turn. We cover board representation, valid move generation, performance measure, and adversarial search. The most common design and implementation techniques for each of these components are presented. We draw for this presentation from the literature, from our own experimental chess engine testbed, and from current existing chess engines. Emphasis is on introducing the fundamental concepts, and the results of several performance analyses we conducted for comparisons.

Keywords: chess engine, knowledge representation, minimax, alpha-beta pruning, ply, performance measure

1 Introduction

Chess is a board game between two players, which emerged in Europe during the second half of the 15th century. The board is called a chessboard, and it is a square-checkered board with 64 squares arranged in an eight-by-eight grid. Each player initially controls sixteen pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The object of the game is to checkmate the opponent's king, which is when the king is directly under attack and cannot elimate the threat in any way on the next move.

Computer chess is software, or a combination of software and hardware, that is capable of playing chess autonomously without human guidance. Computer chess typically has the intention of solo entertainment, although it does have many other uses, such as: an algorithm benchmark, research pertaining to different areas, aids in chess analysis, and computer chess competitions. Ever since the early days of computers, computer scientists, particularly those with interests in artificial intelligence (AI), have been fascinated with computer chess. A major goal in AI is to create intelligent software, and if a chess program can outsmart humans in this strategic,

intellectual game, then AI will have reached a milestone on the way to the intelligent computer. Another reason why games such as chess are interesting is because they are (usually) too hard to solve. For instance, chess has a branching factor of about 35, and games often go up to 50 moves per player, thus the game tree will have about 35^{100} nodes. In the past, it was unclear whether any chess program would ever be able to defeat top human players. However, with processing power increasing and memory costs low, it soon became clear in the late 1990s that computer chess programs were on par or even surpassing top human players.

A chess agent is a rational agent. It acts so as to achieve the best possible outcome (and if there were uncertainty, then the best expected outcome). However, due to time and cost constraints, this is not possible. A chess agent has to *satisfice*, that is, it has to make decisions that are good enough, rather than expensive optimal solutions. To understand the type of agent this is, we first need to determine its environment types. Chess is *fully observable*; all information is known to both agents (i.e. both players), in other words, there is no hidden or noisy information. Chess is *strategic*; the environment is deterministic (i.e. the next state of the environment is determined by the current state and the action executed by the agent) except for the actions of other agents. Chess is *sequential*; the current decision could affect all future decisions. For instance, short-term actions can have long-term consequences, which means the agent needs to think ahead. Chess is *static*; the environment does not change during decision making[1]. Chess is *discrete*; it has a finite number of distinct states and also has a discrete set of percepts and actions. Lastly, chess is *multiagent*; it is a two-agent environment. In chess, an agent's behaviour is best described as maximizing a performance measure whose value depends on the other agent's behaviour. Under the rules of chess, maximizing one's performance measure minimizes the other's performance measure. Therefore chess is a *competitive* multiagent environment.

Now that the environment types are known, we can determine the agent type. A chess agent is a *utility-based agent*. The goal in chess can be reached in different ways, thus the agent not only looks at succeeding, but also how it gets to the goal. It keeps track of the world state, as well as the goal it is trying to achieve and chooses an action that will eventually lead to the achievement of its goal. However, not just any action is chosen, but the best possible action it has come across so far is chosen. Thus the agent uses a model of the world along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility.

Every chess program has a chess engine. This generic chess engine used in computer chess consists of multiple components. Choice of programming language is not very important as a chess engine can be written in any language. Knowledge representation is a crucial step for any problem. In a chess engine, this is done in the board representation. It is one of the most impor-

[1] If chess is played with a clock, the environment is semidynamic. The environment itself does not change with the passage of time but the agent's performance does.

tant components in terms of program structure, and it can also have a big impact on performance. The strength of the AI depends on the *ply* and on the *performance measure*. In game tree searches, the amount of ply determines how many moves the program looks ahead (i.e. how deep the search goes in the game tree). Thus the higher the ply, the better the expected move will be. The performance measure evaluates the behaviour of the agent in an environment. It does this with an *evaluation function* or *heuristic*. In order to have working AI, a chess engine first needs to be able to generate valid moves. Chess contains many rules, and all of these have to be included in the program.

2 Board Representation

Representing a chess board in memory so that it is both efficient and easy to work with is challenging. There exist many different chess board representations, each with pros and cons, and that is why there is no such thing as the perfect chess board representation.

2.1 Traditional Method

One of the earliest (and arguably one of the simplest) representations is the 64 element array (can be one dimension of size 64 or two dimensions of size 8). Each element represents a square on the board and information about that current square, such as the piece on that square and its colour. This can be done in multiple ways as well. For instance, one can identify white pieces by using a unique positive number for every type (e.g. 1 for pawn, 2 for knight, etc.) and black pieces by using a unique negative number for every type (e.g. -1 for pawn, -2 for knight, etc.). One can also have an array of objects and each object has attributes such as the piece type and colour. The board will have a representation similar to the one shown in Figure 1. The rows

Figure 1: 8x8 board representation

```
8: 56,57,58,59,60,61,62,63
7: 48,49,50,51,52,53,54,55,
6: 40,41,42,43,44,45,46,47,
5: 32,33,34,35,36,37,38,39,
4: 24,25,26,27,28,29,30,31,
3: 16,17,18,19,20,21,22,23,
2:  8, 9,10,11,12,13,14,15,
1:  0, 1, 2, 3, 4, 5, 6, 7,
  A   B  C  D  E  F  G  H
```

are named *ranks* and the columns are *files*. In the actual representation in memory, ranks and files have a range of 0–7. To calculate the rank and file of a square given its index, we use the formulae

$$\text{rank} = \text{index} \div 8 \tag{1}$$

$$\text{file} = \text{index} \bmod 8 \tag{2}$$

and vice versa

$$\text{index} = \text{rank} * 8 + \text{file} \tag{3}$$

The concept is easy, as every square is represented and they contain information regarding the world state. However, move evaluation can become tedious with this method as bounds and pieces must be checked many times per turn. There are two way how to perform boundary checking with this method. They differ in the sense that one uses the file and rank to move pieces, while the other uses the index. The

method using the index can move pieces by doing simple calculations, for instance if a rook were to move one square to the right, simply add +1 to the index. You can very easily see where this can go wrong, and that is when pieces are near edges. If the rook were on square 15, then adding 1 would mean its new location is square 16 — an illegal move. This method would have to check if the piece is near an edge first. The method that uses file and rank for moving pieces is a bit simpler; the program checks if the new move is within bounds by looking at the new file and rank. The 8x8 representation being a very simple concept was the reason why we chose this method for our chess engine.

2.2 Enclosed Board Method

The enclosed board representation is designed to make move computation easier. It combines the normal move generator and boundary checking mechanism. This system uses a 12x12 matrix as seen in Figure 2. The extra two layers on each side contain sentinel values that mark a square as illegal. The extra outer layer is to ensure that all knight moves lie within the matrix. Here the boundary checking is done by looking at the target value of a move; a move is valid if the target value is not equal to -1. This creates an enclosed board where pieces cannot go out of bounds. Formulae for calculating the file and rank are similar to formula 2 and 1 but with 8 replaced by 12 since it is now a 12x12 representation.

Figure 2: 12x12 board representation

```
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,A8,B8,C8,D8,E8,F8,G8,H8,-1,-1,
-1,-1,A7,B7,C7,D7,E7,F7,G7,H7,-1,-1,
-1,-1,A6,B6,C6,D6,E6,F6,G6,H6,-1,-1,
-1,-1,A5,B5,C5,D5,E5,F5,G5,H5,-1,-1,
-1,-1,A4,B4,C4,D4,E4,F4,G4,H4,-1,-1,
-1,-1,A3,B3,C3,D3,E3,F3,G3,H3,-1,-1,
-1,-1,A2,B2,C2,D2,E2,F2,G2,H2,-1,-1,
-1,-1,A1,B1,C1,D1,E1,F1,G1,H1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
```

2.3 Bitboard Method

Bitboard representation is a very innovative method. A 64-bit integer can store a square state at each bit because there are also 64 squares in a chess board. A 64-bit integer can be processed very quickly on a 64-bit CPU, thus if it is possible to represent a chess board on a single 64-bit integer, processing could be done very quickly and fairly easily.

Because a bit can only have a value of 1 or 0 (true or false), bitboards can only have values of true and false. Thus a separate bitboard would be required to represent each chess piece type. For example: there would be a bitboard for all empty squares, another for all white pawns, another for all black pawns, etc. Figure 3 shows an example of a bitboard for all white pawns. Bitboards are designed for speed and therefore most of the top chess engines today use this method. Rybka, currently the top-rated chess engine, uses a bitboard design when run in 64-bit mode and results in an extra 60% processing efficiency[1].

Figure 3: A white pawn is on position B7

```
00000000
01000000
00000000
00000000
00000000
00000000
00000000
00000000
```

3 Move Generation

There are two types of move generators: (1) legal move generator, (2) pseudo legal move generator. A legal move generator only generates legal moves, whereas a psuedo legal move generator generates all moves and checks if a move is legal in a "make move" function; if it is not a legal move, it will undo the move and try another. Some may argue that a pseudo legal move generator is faster because checking if a move is legal is time consuming. We beg to differ. In our chess engine, we use a legal move generator, but we only generate valid moves for a piece once that specific piece is selected by the user. This is much quicker than generating all moves for every piece on the board and more efficient. Our move generator simply checks squares in which the selected unit can move to from its current location by making sure it is within bounds, and if the current square being looked at is either empty or an enemy unit, it is a legal move (with some minor variations for different units of course). Additionally, a chess engine will also need to cover a variety of special cases, such as en passant, castling, promotion, and restricting certain moves while under check.

4 Performance Measure

The performance measure defines the criteration of success. In chess, it is to capture the king, this is otherwise known as checkmate. This can be done in many ways and how it is done usually depends on the strategy of the player.

4.1 Evaluation Function

The evaluation function *f(n)* evaluates the current state of the board. This provides some guidance in decision making by estimating the goodness of a position in game searches. Evaluation functions are typically designed to be fast and therefore accuracy is not a concern. It should be noted that the function only looks at the current state presented. A popular strategy for evaluation functions is to have a weighted sum of various factors that are thought to influence the value of a move. This is called a *weighted linear function*. For instance, an weighted linear evaluation function for a chess engine may look like

$$\begin{aligned} f(n) \quad = \quad & \sum c_1 * \text{material} + c_2 * \text{mobility} \\ & + c_3 * \text{board control} + c_4 * \text{king safety} \\ & + c_5 * \text{development} + c_6 * \text{pawn formations} \\ & + \cdots \end{aligned}$$

Most simple chess programs only take material into account, that is, they assign numerical values to each piece and sum up the score for all pieces on the board. This evaluation function is called *material balance* and is the function that we used for our chess engine. The summed scores from each side can then be compared (e.g. through division or subtraction) to see which side

is doing better and which move will produce the highest score. In order for this evaluation function to work (well) with chess, the pieces are assigned values in order of importance. What exactly these values are is not important, but it is important that pieces of higher importance get higher values. For instance, for our material balance we used the values in Figure 1.

Type	Value
Pawn	1
Bishop	3
Knight	3
Rook	5
Queen	8
King	95

Table 1: chess piece values

As a player must capture the opponent's king (while defending his own) to win the game, the king must have the highest possible value on the board as it is the most important piece on the board. To determine what a good value would be for the king, we have to calculate the maximum board fitness without a king piece on the board. This maximum board fitness is achieved when a player has lost no pieces and all their pawns are promoted to the highest possible piece (i.e. the queen).

$$\begin{aligned} fitness_{max(noking)} &= (0 \text{ pawns} \times 1) + \\ &\quad (2 \text{ bishops} \times 3) + \\ &\quad (2 \text{ knights} \times 3) + \\ &\quad (2 \text{ rooks} \times 5) + \\ &\quad (9 \text{ queens} \times 8) = 94 \end{aligned}$$

Thus the king can safely have the value 95.

The initial board state will have the value

$$\begin{aligned} initialState &= \sum 95 + 8 + (2 \times 5) + (2 \times 3) \\ &\quad + (2 \times 3) + (8 \times 1) = 189 \end{aligned}$$

The board score is then a simple subtraction of the player's score with the opponent's score (see Equation 4).

$$score = \sum playerValues - \sum enemyValues \tag{4}$$

If the result is positive, the player is winning; if it is negative, the opponent is winning; if it is zero, neither is winning.

Chess programs tend to be very materialistic compared to human players. The speed of the evaluation compensates for this. It is also very difficult to refine an evaluation function enough to gain as much performance as you would from an extra ply of search, thus it is better overall to have a weak evaluator in exchange for as much processing power as possible for the game search.

5 Game Search

Although the most appropriate method for creating an AI chess agent is by searching a game tree, there are also other methods such as using artificial neural networks (ANN) or a combination of ANN and genetic algorithms (GA)[2].

5.1 Adversarial Search

Competitive environments (i.e. agent's goals are in conflict) give rise to *adversarial search* problems—also known as games. Chess is a

[2]Using a GA by itself would not be a good idea as it would be very hard, take a very long time, and still be very weak.

game of *perfect information*. This means that it is a deterministic, fully observable environment in which there are two agents whose turns alternate and in which the utility values at the end of the game are usually equal and opposite (e.g. if one player wins (1), the other necessarily loses (-1), unless it is a draw (0)).

5.1.1 Minimax

Minimax is a decision rule in decision or game theory (and several other uses) for minimizing the maximum possible loss. It (recursively) assumes optimal play by both sides where one side is represented by min and the other by max. For instance, one player (e.g. the human) is trying to minimize the value of the position, while the other player (e.g. the computer) is trying to maximize it. Minimax is probably the most used strategy in (competitive) games of perfect information. The depth of the search in the game tree is proportional to the amount of ply. Obviously the higher the ply, the higher the amount of moves ahead can be seen, but as this tree grows very quickly (remember, chess has a branching factor of about 35), it would be too time consuming to go too deep.

5.1.2 $\alpha - \beta$ Pruning

For complex games such as checkers, chess, and go (just to name a few), it is obviously infeasible to search all the way to the terminal nodes[3]; the search is stopped after a certain depth of recursion is reached. Because the single most important factor is the ply (i.e. the number of look-aheads the program is capable of) it would be very beneficial if the search can get any additional ply by reducing the search space. This is where $\alpha - \beta$ pruning comes in. It effectively cuts the tree in half while still computing the correct minimax decision. When applied to a standard minimax tree, it returns the same move minimax would but prunes decisions that cannot affect the final decision. Deciding on which nodes to prune is done by looking at tentative min and max values. Max is represented by α and min by β.

$$\text{If tentative min} < \text{tentative max, } \alpha\text{-prune.} \quad (5)$$

$$\text{If tentative max} > \text{tentative min, } \beta\text{-prune.} \quad (6)$$

[3]It is estimated that if such a search were conducted for chess, at least 10^{100} positions would be examined for the first move. Even with pruning, it still could not be reduced to a practical level.

6 Conclusion

Many additional improvements can be made to our chess engine, for instance the following could be added:

- transposition table for saving positions in order to prevent recomputing them
- a stronger heuristic/evalution function
- using ordering heuristics to search parts of the tree that are likely to cut off early
- forward pruning
- introduce a narrow search window (i.e. aspiration search)
- quiescence search for a more sophisticated cutoff
- opening book (of moves)
- $\cdots$

Currently, our AI is capable of searching 5-ply in approximately 5 seconds. We compared it to other chess engines to see where we stand. Judging from a very quick comparison ($\approx$ 15 seconds of play time) with Andrew Eller's chess engine, our AI seemed to be on relatively the same level. Compared with Robert Flack's chess engine, ours seemed relatively weak (as expected). However, compared with Neil Skrypuch's chess engine (which is hosted on the Computer Graphics course homepage), our AI seemed to be much more advanced.

References

[1] Rybka computer chess engine. http://en.wikipedia.org/wiki/Rybka