Towards a comprehensive understanding of DeepBlue

Hanyuan Zhang[*]
Xi'an Jiaotong-Liverpool University
Suzhou, China
Hanyuan.Zhang21@student.xjtlu.edu.cn

Honglin Zhang[*]
Macao University of Science and Technology
Macao SAR, China
1230031644@student.must.edu.mo

Jiahui Zhang[*]
South China Normal University
Guangzhou, China
20223801002@m.scnu.edu.cn

Jialin Zhang[*]
Tongji University
Shanghai, China
2352595@tongji.edu.cn

Kefeng Duan[*]
Sun Yat-sen University
Zhuhai, China
duankf@mail2.sysu.edu.cn

Ruohan Yan[*]
University of Nottingham in Ningbo China
Ningbo, China
biyry12@nottingham.edu.cn

Xinyan Du[*]
Sichuan University
Chengdu, China
2022141210115@stu.scu.edu.cn

Yiwei Jiang[*]
Tsinghua University
Beijing, China
jiangyw2222@163.com

Abstract

Deep Blue, a milestone in the history of artificial intelligence, showcases profound advancements in computational power and strategic thinking achieved through innovative algorithms and unique problem-solving techniques. This review explores the architectural and algorithmic innovations behind Deep Blue, examining its impact on both AI research and the game of chess. By analyzing Deep Blue and its related work, we offer a comprehensive insight into the evolution of artificial intelligence for chess. To serve a vivid instance of modern chess engine techniques, we create our chess engine Seraphina and its source code is available here: https://github.com/HenryZNNUE/Seraphina.

Keywords

DeepBlue, Artificial Intelligence, Review

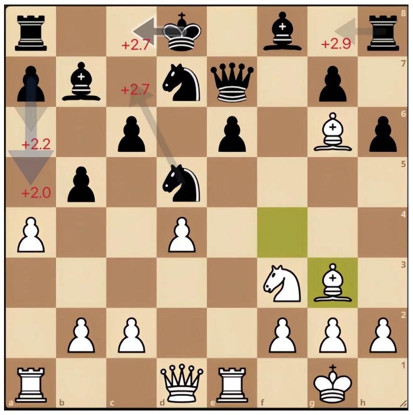

Figure 1: The classic game between Deep Blue and Garry

1 Introduction

In 1997, IBM's Deep Blue [4] AI competed against the reigning World Chess Champion, Garry Kasparov, in a highly publicized match. Figure 1 illustrates a crucial moment of this game.

At this point in the game, white, also known as DeepBlue, is clearly in a favorable position. Deep Blue managed to capitalize on this advantage and secured a victory by converting it into an endgame win. The triumph of Deep Blue was a landmark achievement in both AI research and technological advancement. Unlike human players who rely on pattern recognition and intuitive reasoning, Deep Blue's approach is marked by its systematic search algorithms and exhaustive evaluation of possible moves.

To comprehensively understand techniques utilized in DeepBlue, this paper offers a retrospective analysis of Deep Blue and its related work, intending to provide an in-depth summary of AI's progress in chess.

The following paper will be divided into two sections: discussions of techniques used by Deep Blue in Section 2 and other related works to tackle chess games in Section 3.

2 Deep Blue's Approaches

DeepBlue considers several essential features to evaluate the situation of chess, search for further moves and generate optimal steps. we discuss these in this section.

2.1 Board Representation (Bitboard)

In a chess analysis engine, a Bitboard[10] is used to represent the number of each type of piece (kings, queens, rooks, knights, bishops, and pawns) for both white and black sides. For each type of piece, a 64-bit integer (typically of the unsigned long long type) is used to denote the positions of a specific type of piece on the board; since the chessboard has 64 squares, a 64-bit integer exactly corresponds to it. Each bit represents one square on the board, with a 1 indicating that the square contains the specified type of piece and a 0 indicating that it does not. For example, the Bitboard representation for white

[*]Equal contribution. Sort by alphabetical order.

pawns would store a '1' in the bit corresponding to the square where a white pawn resides, within the 8-byte (64-bit) space.

2.2 Move Generation

The goal of move generation is to efficiently find all legal moves for a given board state. This involves generating all legal moves as quickly as possible to minimize the search time. The fundamental steps of move generation include the identification of the current player, the traversal of all pieces associated with that player to generate potential moves, the validation of the legality of each move, and the exclusion of any moves that are not permitted. Techniques such as magic Bitboards and sliding piece masks are commonly applied to accelerate this procedure. Finally, all legal moves are returned to the search algorithm for further processing.

2.3 Evaluation

Piece	Main Value
Pawn	100
Knight	320
Bishop	330
Rook	500
Queen	900
King	N/A (20,000)

Table 1: Piece Values Used in Deep Blue

Piece Values Used in Deep Blue are listed in Table 1 (Typically, we assign a very large number such as 20,000 to the King because using infinity can cause issues in calculations. Infinity cannot be compared or combined with other values, so a large number is chosen to simulate the King's irreplaceable value without breaking the arithmetic operations.)

A typical chess engine evaluates the board using various features that contribute to the overall position score. These features, listed in Table 2, are combined using a weighted sum, where each feature is assigned a coefficient based on its importance, and the resulting score reflects the engine's assessment of the position.

Feature	Description
Material Balance	The difference in material value between sides
Pawn Structure	The connectivity and advancement of pawns
Piece Activity	The mobility and control of space by pieces
King Safety	The safety and development of the king
Passed Pawns	Pawns that have no opposing pawns in front
Center Control	Control over the central squares
Threats	Immediate threats to capture valuable pieces
Development	The number of developed pieces

Table 2: Features Used in Chess Evaluation

2.4 Search

To select the best solution out of billions of results, Deep Blue uses Minimax, a typical algorithm based on decision tree [3]. This algorithm can be used to guide classic complete information zero-sum games such as Tic-tac-toe, and Chess.

In a zero-sum game, players select between options that either maximize their advantage or minimize their opponent's advantage after N moves. The decision-making process of both sides is regarded as a decision tree. If a certain layer of the decision tree is the decision basis state of the own side (that is, the next action of the own side), the own side will choose the path that maximizes the benefits of the own side, and this layer is called the MAX layer. If such layer is the decision basis state of the opponent, the opponent will choose the path that minimizes our benefit, and this layer is the MIN layer. Thus, a minimax decision tree will contain max nodes (nodes in the MAX layer), min nodes (nodes in the MIN layer), and terminating nodes (game terminating state nodes or state nodes at N moves). The expected return corresponding to each node becomes the minimax value of that node.

Algorithm 1 Minimax Algorithm

```
function MINIMAX(node, depth, maximizingPlayer)
    if depth = 0 or node is NULL then
        return static evaluation of node
    end if
    if maximizingPlayer then
        $maxEva \leftarrow -\infty$
        for each child of node do
            $eva \leftarrow \text{Minimax}(child, depth - 1, \text{False})$
            $maxEva \leftarrow \max(maxEva, eva)$
        end for
        return $maxEva$
    else
        $minEva \leftarrow +\infty$
        for each child of node do
            $eva \leftarrow \text{Minimax}(child, depth - 1, \text{True})$
            $minEva \leftarrow \min(minEva, eva)$
        end for
        return $minEva$
    end if
end function
```

For terminating nodes, the minimax value is a direct estimate of the situation. For the max node, the value of the child node with the largest minimax value is selected as the value of the max node because the action selected by the max node will be given by its party. For the min node, it will select the smallest min node. The minimax algorithm process can be described as follows:

1. Construct a decision tree;
2. The evaluation function is applied to leaf nodes;
3. Calculate the minimax value of each node from the bottom up;
4. Select the branch with the largest minimax value from the root node as the action policy.

3 Methods used in Modern Chess Engines

In this section, we discuss methods used in modern chess engines.

3.1 Magic Bitboard

Despite its huge table size, register usage and code size are important issues as well - and here Magic Bitboards [7] are unbeatable. There are enough variations of space-time tradeoff and implementation details of that theme for all who like to play the optimization game. C-source code with various precompiler options is available from Pradu Kannan's site. MINIMIZE-MAGIC is about Plain versus Fancy , while PERFECT-MAGIC-HASH enables an additional indirection via 16-bit indices. As always, with space-time tradeoffs - it depends on the individual cache- and memory footprint inside a particular chess program and the hardware architecture, which solution is preferable.

3.2 Move Picking and Ordering (Histories)

The following standard techniques are commonly employed in the process of finding a good first move: First, the PV-Move [1] is selected from the principal variation of the previous iteration. Secondly, if available, the Hash Move is chosen, which is a stored move from the Transposition Table. Lastly, Internal Iterative Deepening is utilized when no hash move is available, a technique that is likely applied only at PV-Nodes.

3.3 Transposition Table

In many games, there are multiple ways to reach a given location. Such strategies are referred to as "transposition" [11]. In chess, the maximum possible transposition after n moves is $(n!)^2$. Although many of them are illegal movement sequences, the program may still analyze the same location on multiple occasions.

The transposition table can be employed to address this issue, as it is a cache of previously seen locations and related evaluations. When encountering a new location, the program will check the table to see if the location has been analyzed, which can be quickly completed within a constant amortization time. If so, the table contains the value previously assigned to this location and the value is used directly. If not, calculate the value and input the new location into the hash table.

The number of locations searched by a computer usually far exceeds the memory limit of its operating system. Therefore, not all locations can be stored. When the table is full, less used positions will be removed to make space for new positions, which makes the transposition table a type of cache.

3.4 Search

Based on Deep Blue, Minimax search algorithm get consistently improved which has already been extended to Negamax Alpha-Beta Pruning [6] and Quiescence Search [2] with multiple and various aspects of reductions and pruning techniques.

Instead of processing all available moves under current position in Minimax which causes gigantic search tree, poor speed and lower ELO rating, pruning nodes that obtain out-of-range values via applying a lower bound, Alpha, and an upper bound, Beta resolves the issue to some extent.

In addition, to avoid the horizon effect in Alpha-Beta Pruning, Quiescence Search only evaluates positions that have no winning tactical moves could possibly avoid making blunders such as winning a pawn currently but lose a queen next. Techniques such as null move pruning, late move reduction, etc are employed to further shrink search tree and eliminate null and non-essential moves.

3.5 Evaluation

Using deep neural networks [5, 8] for chess evaluation seems to be resource-intensive. To balance accuracy and speed, NNUE is introduced.

NNUE[9] , or Efficiently Updatable Neural Network, leverages the nature of chess where a move typically results in only minor changes to the board position. Therefore, NNUE uses a two-dimensional array called "Accumulator" with indexed inputs to avoid tremendous scale of updates. A quantized shallow network using SIMD instructions can optimize the speed, overcome the mimic loss of accuracy and result in better evaluation. Related source code can be viewed in the Github link in Abstract.

Simultaneously, NNUE networks commonly use ClippedReLU instead of ReLU, adding a scalar to the ReLU function. Therefore, the formula of ClippedReLU should be: $f(x) = \min(\max(0, input), scalar)$. Influenced by the aggressive quantization of NNUE, picking quantization number as scalar is suitable. Also, weight clipping in hidden layers constrains the magnitude of weights that enhances evaluation.

Training NNUE requires backward propagation with gradient operation and adam, huge datasets that contain positions with portioned Syzygy Tablebases Win Draw Loss rate and engine's evaluation score controlled by lambdas.

Finally, incorporating material balance called SimpleEval[8] by combining tuned material values and counts with NNUE's score provides a more precise assessment because NNUE lacks specific material headers in Lc0's network. SimpleEval Code can be viewed here.[1]

4 Conclusion

In conclusion, Deep Blue stands as a significant milestone in the history of chess programming. Employing techniques such as Bitboards, Move Generation, the Minimax algorithm, and nuanced evaluation functions, Deep Blue charts new territories in chess. As the field of chess programming matures, its methods have undergone significant evolution. Advanced search algorithms have collectively elevated the search efficacy and evaluative precision of computer chess engines.

As AI technologies continue to advance, we anticipate the development of increasingly sophisticated and powerful systems. These technologies extend beyond chess, offering deep insights into broader AI and computational problem-solving fields.

References

[1] Thomas S Anantharaman. Extension heuristics. *ICGA Journal*, 14(2):47–65, 1991.
[2] Don F Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, 1990.
[3] M Campbell, A Joseph Hoane, and Feng-hsiung Hsu. Search control methods in deep blue. In *AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information*, pages 19–23, 1999.
[4] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.
[5] Omid E David, Nathan S Netanyahu, and Lior Wolf. Deepchess: End-to-end deep neural network for automatic learning in chess. In *Artificial Neural Networks and Machine Learning–ICANN 2016: 25th International Conference on Artificial*

[1]https://github.com/official-stockfish/Stockfish/blob/master/src/evaluate.cpp

Neural Networks, Barcelona, Spain, September 6-9, 2016, Proceedings, Part II 25, pages 88–96. Springer, 2016.

[6] Evta Indra, Ningot Putra Sijabat, Muhammad Alvin Riady, Josep Sutoyo Muda Lumbantobing, et al. Analisa efektivitas algoritma minimax, alpha beta pruning, dan negamax dalam penerapannya pada permainan papan (board game). *Jurnal Ilmu Komputer dan Sistem Informasi (JIKOMSI)*, 3(2):49–59, 2020.

[7] Pradu Kannan. Buzz - a winboard chess playing program, 2010. Source of Pradu Kannan's magic Move Generator, MINIMIZE-MAGIC.

[8] Dominik Klein. Neural networks for chess. *arXiv preprint arXiv:2209.01506*, 2022.

[9] Yuȷ Nasu. Efficiently updatable neural-network-based evaluation functions for computer shogi. *The 28th World Computer Shogi Championship Appeal Document*, 185, 2018.

[10] Pablo San Segundo, Ramon Galan, Fernando Matia, Diego Rodriguez-Losada, and Agustin Jimenez. Efficient search using bitboard models. In *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, pages 132–138. IEEE, 2006.

[11] David J Slate. A chess program that uses its transposition table to learn from experience. *ICGA Journal*, 10(2):59–71, 1987.