Algorithmic Clairvoyance in Computer Chess

Abir Lakehal – z5453120
Major Project – COMP4121

1 Introduction

Computerised opponents serve us well. They are a friend to the loner who has nobody to play against, a coach for the amateur who plays even in areas without internet connection and a worthy opponent for a world champion. The latter, though impressive, is even more impressive in the context of chess, a game which requires strategic thinking for skilful play [1]. So, when grandmaster Garry Kasparov was defeated by the supercomputer Deep Blue in 1997, the latency of mechanised thinking evolved into something more tangible, and the bar for computerised opponents was raised higher.

The story behind the feat has already been told by the man behind it [2], and this paper seeks not to retell a story. Instead, we aim to define boundaries between brute-force approaches and crude limitations of thinking. In chess, any position can be perfectly evaluated if one were able to look to the ends of the game [1]. Humans of course do not possess such clairvoyance, and so a player must use their arsenal of familiar situations and common manoeuvres to guide their play along the path that will most likely lead to a win. We call a player who does this well a *master*.

Machines could possess such clairvoyance if computational power were not a concern – for all positions, follow the play along each possible move until a win, loss or draw is reached and then backtrack on the assumption that each player will do what is best for themselves [3]. What we see in chess engines today is a refinement of Claude Shannon's idea - a game tree is searched to some depth using an alpha-beta optimisation principle. Time control is addressed with iterative deepening, and a sense of strategy is incorporated using heuristics. Is this engine then, worthy of the title master? Or, are we biased in our merit? The approach is not exactly one of brute-force, since not all 10^{120} variations are calculated [1]. But unlike humans, the machine will not perform better tomorrow because it played today [3]. Is it worthy of the title only once it can learn?

In Section I, we analyse the algorithms behind chess engines, and how they are both similar and different to our concept of thinking and in Section II, we explore the optimisations and heuristics used to refine such algorithms to incorporate learning into the system. As part of the project, we implement a simple chess engine with these refinements.

2 The minimax idea

There is a large psychological influence in the ways that humans play chess. A human may be crafty, and fulfil an optimistic, ulterior motive under the guise of sacrificing a 'better' move for no apparent reason. An embryonic chess machine has no concept of such thinking and so at its essence is selfishness – one player is a 'Maximiser' and the other is a 'Minimiser'. The Maximiser tries to maximise their score and assumes that the Minimiser will employ an optimal defensive strategy to reduce this score; both players do what is best for themselves.

Now, suppose we have a large tree where nodes represent positions and branches represent moves. Then the root of such a tree is the player's current position and its children are the positions that can be reached by playing a certain move. The nodes at which the game ends are terminal states.

Since a move consists of one player's choice and another player's reaction, a traversal of a branch is a half-move, and we call this a *ply*. If no legal moves exist from a position p, then we associate a value to the leaf node called the *static evaluation score* to approximate its usefulness for the player whose turn it is to play from p. The Maximiser favours positive such values, while the Minimiser favours negative such values. The goal then becomes to choose the optimal move out of all those legal from the current position.

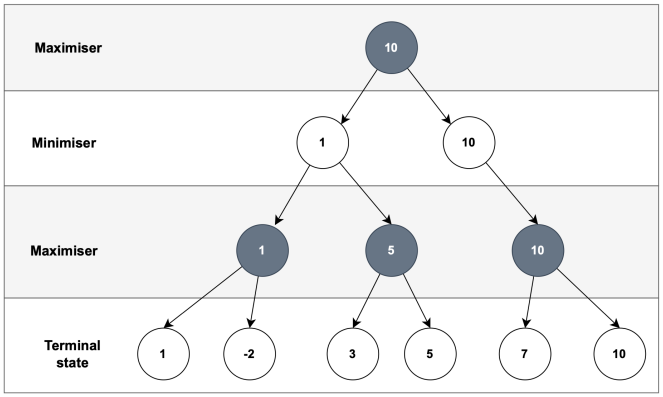

Figure 1: Minimax evaluation

The value of any move can only be perfectly evaluated if the move is pursued to the end of the game, so the machine must backtrack from the terminal states. As depicted in Figure 1, if it is the Maximiser's turn to play from p, the static evaluation score at the corresponding

node is the maximum of all the scores immediately below. Conversely, if it is the Minimiser's turn, the minimum is taken. We keep evaluating these scores until the root node is reached; the final value signifies the highest value achievable from the current position.

This brute-force approach however, is not computationally feasible. If we search the entire game tree, the number of nodes analysed is exponential – the number of children of each node i.e, the *branching factor*, raised to the power of the number of plies. It turns out that there are more moves to pursue than the number of atoms in the known universe. Could we perhaps analyse less nodes and still yield the same static evaluation score at the current position?

3 Alpha-Beta pruning

It would take immortality for a match between the embryonic chess machine and a human to reach the endgame. And since it is much easier to devise an efficient algorithm than to formulate the elixir of life, we refine the machine to avoid pursuing moves that are certainly bad and which cannot alter the outcome.

The Alpha-Beta principle dictates that a game position need not be evaluated if it is worse than a previously examined move. This is deduced by keeping track of the score that each player is guaranteed to attain; α is the lower bound for the Maximiser's score and β is the upper bound for the Minimiser's score. Initially, we set α and β to their worst case values: $-\infty$ and $+\infty$ respectively. At each level during the search, the value of the move that gives the current player the highest evaluation is brought up to the level above as α or β.

Suppose that it is the Maximiser's turn to make a move from a position p and that we have evaluated the static evaluation score of one move from the d legal moves available, say d_i. Then the value of p must be at least the score of d_i. To see why, assume that we have evaluated all d moves. The Maximiser will choose d_i if all the other moves produce a score less than d_i, or a different move d_j if $d_j > d_i$.

If $d_i > \beta$ then a better alternative has already been found. So, the Maximiser's parent, the Minimiser, will always choose the move that yielded β. This means that the remaining moves no longer need to be explored; their subtrees can be pruned away. The same applies when it is the Minimiser's turn to play. If a move is evaluated to be less than α, then the Minimiser's parent, the Maximiser, will always choose the move that yielded α. In general, pruning occurs if at any point $\alpha \geq \beta$ or $\beta < \alpha$.

Consider the scenario in Figure 2. The left most sub-tree is evaluated first, returns an α value of 5 to the Minimiser

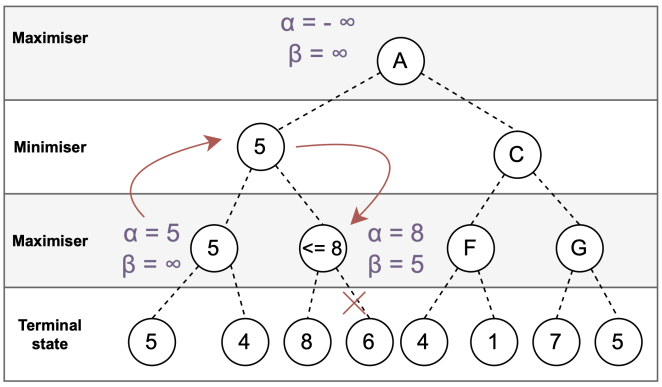

Figure 2: Alpha-Beta pruning

above it, and this value is passed down to the next left sub-tree. At this sub-tree, the Maximiser first sees an 8, and can safely deduce that the score of at its node must be at least 8. But the Minimiser above already knows of a move of value 5, which it will certainly choose since $5 < 8$. Therefore, the leaf node with value 6 need not be checked.

Earlier, it was mentioned that the number of nodes to be explored by the machine is more than the atoms in the known universe. Whether this remains true with the Alpha-Beta procedure is contingent upon the order in which nodes are visited. If the nodes are visited in the worst order, the observation holds because no pruning occurs. But when the tree is perfectly ordered wherein the leftmost child is the optimal move, the machine evaluates only the *critical* positions of the game tree [4].

Although Knuth and Moore provide a rigorous mathematical analysis of perfectly ordered trees in terms of critical positions in their paper, it is worthy to recapitulate the proof to appreciate how the Alpha-Beta procedure defines a lower bound on the number of positions that need to be examined.

We begin by defining the local position of a node as its position on level l. The global position of a node is a sequence $a_1 \ldots a_l$ of the local positions of all the nodes taken to reach it. For example, the sequence 2112 denotes the position reached when the second move is pursued from the root node, and then from that resulting position the first move is pursued, and then the first and then the second. A node is a *critical position* if the first move is pursued at all even levels or at all odd levels of the tree i.e, $a_i = 1$ for all even i or for all odd i. So, if the branching factor of the tree is d, then there are $d^{\lceil l/2 \rceil}$ positions where $a_i = 1$ for all even i, and $d^{\lfloor l/2 \rfloor}$ positions where $a_i = 1$ for all odd i on each level. Every critical position can be categorised into one of the following types:

- Type 1: if at every level the first move is pursued i.e $a_i = 1$ for all i in the sequence.

- Type 2: if $l - j$ is even, where j is the level of the first non-optimal move pursued in the sequence.
- Type 3: if $l - j$ is odd for the same j as type 2.

Clearly, the first child of a type 1 node is also of type 1. The rest are of type 2, because $l = j$ so $l - j = 0$ which is even. By a similar calculation, one can find that all children of a type 2 node are of type 3 and vice-versa. The idea in the paper then is that by the Alpha-Beta procedure, all children of type 1 and type 3 nodes are examined, and all children except for the first of a type 2 node are pruned [4]. This means that $O(d^{\frac{l}{2}})$ nodes are examined in the entire tree, which is half the number of static evaluations otherwise made by the embryonic machine.

The irony in the best case is that it is purely theoretical; to perfectly order a tree, all nodes need to be evaluated, yet the alpha-beta procedure's raison d'être is to avoid doing such. Nevertheless, the benefits of the Alpha-Beta procedure can still be reaped, albeit not fully, by pursuing moves randomly instead. Knuth and Moore show that if nodes are allocated random values and are searched on the relative order of such values, then for positive constants C_3 and C_4 the number of static evaluations s is bounded by

$$C_3 \frac{d}{\log d} \leq s \leq C_4 \frac{d}{\log d}$$

as $d \to \infty$.

But since the branching factor is significantly smaller in practice, s usually tends to $O(d^{\frac{3l}{4}})$ instead [5].

Whether avoiding moves that are certainly bad is a simulation of thinking is arguable – efficiency per se is not strategic, but making decisions which lead to efficiency is. Of equal importance is the fact that the machine is still pursuing moves to the end of the game to determine their positional consequences, which gives the impression of brute force rather than logical analysis [1]. Competitive chess, after all, is played under time control, and the current machine would no doubt collapse under such constraint.

4 Iterative deepening

Rather than spending time to perfectly evaluate positions, it may be a better idea to allocate computational effort such that a *good enough* move is always available before the time is up. However, adopting this "done is better than perfect" approach with the current machine's depth-first search framework is difficult because:

1. the tree needs to be searched to sufficient depth to be confident that a move will maximise the chances of winning; and
2. multiple moves need to be pursued to establish a performance standard for what the 'best' move is.

Essentially, the search needs to be both deep and wide. A simple solution is to modify the program to search to a fixed depth. Nodes at this depth become pseudo terminal states, so their static evaluation score is an *estimate* of how good the position is for a win. Then, instead of one search, multiple searches are conducted with the search depth incremented each time – first the tree is searched to depth one using the Alpha-Beta procedure and then to depth two and so on. This way, the program always has a 'best' move available at each level of iterative deepening to return in case the time is exhausted before the given depth is reached.

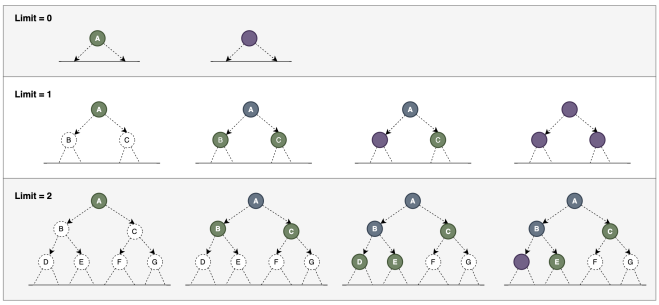

Figure 3: Iterative deepening

While this approach embeds the Alpha-beta procedure with the completeness of a breadth-first search as seen in Figure 3, there are two major concerns. The first is that the same nodes are visited with each new iteration, which appears wasteful. The second is that searching up to a fixed depth is risky. How can the machine foresee disasters that may be lurking further down the tree if the search does not go as far?

One may argue that because the number of evaluations increases exponentially with depth, the time spent re-evaluating positions at shallow depths is still small compared to a search with no cut-offs and is therefore not a major concern. Alternatively, it may be thought that the accurate move ordering facilitated by iterative deepening compensates for the issue entirely. Since the program determines the best move at each level, searching this move first at each new level would lead to very efficient pruning. However, these best moves must still be stored, and repeated searches are still due to occur by transposition.

In the next section, we will see that the former is quite easily addressed with transposition tables. The latter, however, poses an interesting problem from which early chess programs suffered from, called the *horizon effect*. Searching to a fixed depth establishes a horizon, and a move is *good enough* if no consequences are apparent before it. As such, the programs would exhibit behaviour wherein low value pieces are captured because they are

locally optimal, but which led to the loss of higher value pieces later on.

At this point, it may be tempting to deduce that mechanised thinking is simply unattainable for a complex game like chess, for each refinement to the program seems to introduce a cascade of new problems. Iterative deepening in its current form renders the machine blatantly unintelligent; it makes no exceptions to extend the search for promising moves and to withdraw for hopeless ones. Still, there is merit in the idea because uncertainty induces strategy, and strategy is an imitation of intelligence. Incorporating heuristics which dictate when such exceptions should be made can actually mitigate the horizon effect. In turn, the machine develops quasi-intuition.

5 Transposition tables

With iterative deepening, the same board position is analysed multiple times. Without iterative deepening, the same board position is still analysed multiple times because chess is a game in which transpositions are possible and likely. As an example, consider Figure 4 which demonstrates how the sequence of moves defining the English Opening can transpose into the Queen's Gambit.

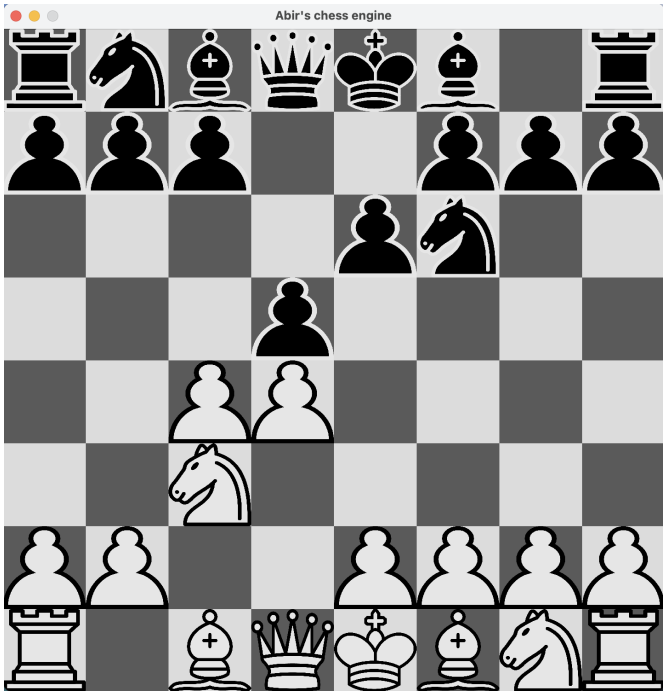

Figure 4: Transposition on my chess engine. Moves d4, d5, c4 define the Queen's Gambit and c4 defines the English opening. The above board position can be produced using either opening.

The most obvious solution is to use memoisation in the form of a transposition table. A transposition table stores information (score, depth, bounds and optimal successor) of positions which have already been searched so that if a position is re-encountered, the information can be immediately retrieved from the table, eliminating the need to search its sub-tree again.

A transposition table is implemented via a hash function h which maps board positions to entries in a hash table. Before we describe the construction of h using Zobrist hashing, we must introduce the graph-history interaction problem.

In chess, the legality of moves is path dependent. Therefore, if we have two identical board positions p_i and p_j, it can be that a move is legal from p_i but illegal from p_j. This manifests through castling rights and en passant captures. Castling describes the cross over of a king and a rook in one move. This is only permitted if both pieces have not moved beforehand, the squares between the king and rook are empty, and the king does not move out of, through or into check.

In an en passant capture, the capturing pawn does not overtake the square of the captured pawn. Such a move is only valid when the capturing pawn has advanced exactly three ranks, and immediately after the captured pawn has moved two squares at once. This is demonstrated in Figure 5.

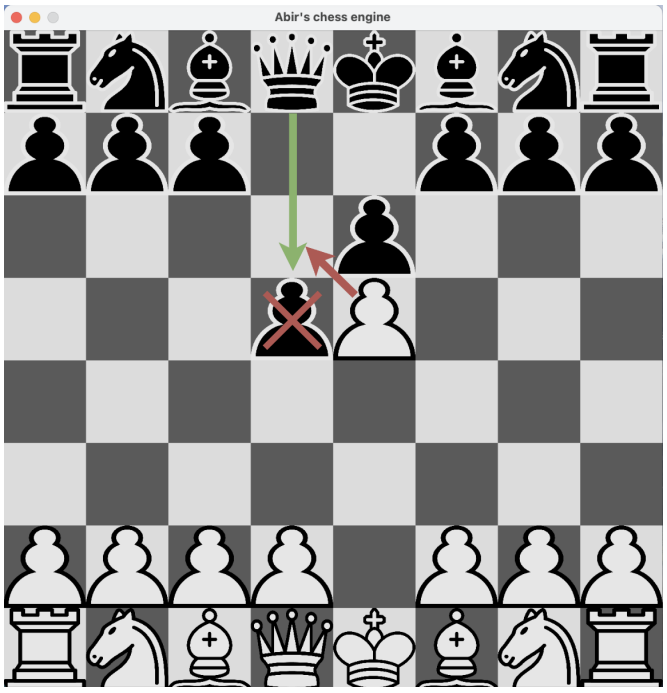

Figure 5: En passant

Any board position can be described by whose turn it is to play (white or black), the placement of the pieces on the board, castling rights, and whether pawns can capture en passant. In Zobrist hashing, each of these features f_i (including every combination of a $(piece, placement)$ pair) is assigned a pseudo random N-bit number $H[f_i]$ (usually 64-bits). Then, h is defined by the XOR $\oplus$ of all

the random numbers relevant to the board position

$$h(p_i) = H[f_1] \oplus ... \oplus H[f_n].$$

For example, the hash for the board's initial state is:

`[Hash for White Rook on a1]` $\oplus$ `[Hash for White Knight on b1]` $\oplus$ `[Hash for White Bishop on c1]` $\oplus$ `[ Hash for remaining white pieces]` $\oplus$ `[Hash for Black Rook on a8]` $\oplus$ `[Hash for Black Knight on b8]` $\oplus$ `[Hash for Black Bishop on c8]` $\oplus$ `[Hash for remaining black pieces]` $\oplus$ `[Hash for king and queen side castling]` $\oplus$ `[Hash for possible en passant captures]` $\oplus$ `[Hash for white to move]`

This particular hashing method is useful for two reasons. The first is *involution*. Since the XOR operation is its own inverse, the hash value of a position need not be recomputed entirely after every move; it can be quickly obtained by XORing the hash value of the board before the move with the features that are removed and then with the features that are added. For example, if a white pawn is queened:

`[Hash for original position]` $\oplus$ `[Hash for White pawn on f7] (removing pawn from f7)` $\oplus$ `[Hash for White queen on f8] (promotion)` $\oplus$ `[Hash for black to move]`

The second is *pairwise independence*, or strong universality. Let us denote the set of board positions as S. A family of hash functions $\mathcal{H}$ is k-wise independent if for each $h \in \mathcal{H}, h : U \to N$, then for any distinct key (board position) $p_1, p_2, ..., p_m \in S$ and any hash code (possible N-bit hash) $a_1, a_2, a_m \in N$,

$$\Pr_{h \in H}[h(p_1) = a_1 \wedge \cdots \wedge h(p_k) = a_k] = |N|^{-k}.$$

Since Zobrist hashing is 3-wise independent [6], and therefore 2-wise independent, $h(p_i)$ does not influence the outcome of $h(p_j)$ in any way. With an N-bit hash, the size of the hash space $|N| = 2^N$. Thus, for any two distinct board positions p_i and p_j, the probability that $h(p_i) = h(p_j)$ is $\frac{1}{2^N}$.

Let $C_{p_i}p_j = 1$ if and only if $h(p_i) = h(p_j)$ and 0 otherwise. Then the expected number of collisions in the hash table satisfies

$$E[\#C_{p_i}p_j] = \sum_{p_i \neq p_j} E[C_{p_i}p_j] = \binom{m}{2} \times \frac{1}{2^{-N}}.$$

Collisions, though minimised, are not eliminated entirely, however Hyatt and Cozzie conclude that with a 64-bit signature, the number of collisions is not fatal to the accuracy of the search due to the substantial size of the game tree [7].

6 Heuristics

It is often said that past performance is not indicative of future results, but a machine should view "la vie en rose" – if a move was once good, then it is likely to be good again. By maintaining a list of recent 'killer' moves such as checks, promotions, captures (especially those of highly valuable pieces) and then pursuing them first during the search, alpha-beta cutoffs are likely to be induced.

A stronger version of the killer heuristic is the singular extension heuristic. In addition to maintaining multiple good moves, maintain the best move that has been made so far and if it is a legal move, consider it first. This mimics the experience that a human player might leverage.

Lastly, we revisit the problem of the horizon effect i.e, the situation where a search stops just before a capture, commonly referred to as a *non-quiescent* position. The machine should prolong the search until a *quiescent* position is reached (one in which no captures are imminent).

7 Conclusion

We followed closely the metamorphosis of a chess machine, from a primitive brute-force algorithm into one that is optimised and sprinkled with heuristics. Most importantly, we emerge with an understanding of how a machine can simulate thinking, even for one of the most complex of games. Of course, more can be said, particularly apropos replacement strategies in a transposition table (since entries must be overwritten due to hardware constraints) and the evaluation function (we have assumed that positions are allocated a static evaluation score without delving into the evaluation function itself), but this diverges from the paper's primary intent to assess whether the machine can act as if it were intelligent.

References

[1] Shannon, C.E. (1988), Programming a Computer for Playing Chess. In: Levy, D. (eds) Computer Chess Compendium. Springer, New York, NY.

[2] Hsu, F. (2022), Behind Deep Blue: Building the Computer That Defeated the World Chess Champion. Princeton University Press.

[3] Newell, A. (1988), The Chess Machine: An Example of Dealing with a Complex Task by Adaption. In: Levy, D. (eds) Computer Chess Compendium. Springer, New York, NY.

[4] D. E. Knuth and R. W. Moore (1975), An analysis of alpha-beta pruning, Artificial Intelligence, vol. 6, no. 4, pp. 293–326

[5] S. Russell and P. Norvig (2010), Artificial Intelligence: A Modern Approach, 3rd ed. New Jersey: Pearson

[6] M. Patrascu and M. Thorup (2010), The Power of Simple Tabulation Hashing, Cornell University

[7] Hyatt, Robert M. and Cozzie, A. (2005), 'The Effect of Hash Signature Collisions in a Chess Program', pp. 131 – 139.

[8] G.Robinson (1995), "14.3 Computer Chess," Netlib.org

[9] A. N. Walker (1997), "G13GAM – Game Theory – computer chess notes," Archive.org

[10] B. Berwick (2011), "Game trees, minimax, & alpha-beta search - Recitation 3"

[11] M. Weinberg (2018), Class Lecture, Topic: "Lecture 1: Course Intro and Hashing" COS 521: Advanced Algorithm Design, Department of Computer Science, Princeton University, United States.