School of Informatics

Informatics Research Review
Tree Algorithms for Computer Chess

[redacted]

January 31, 2021

Abstract

The game of chess is a hugely popular two-player perfect-information game, thematically featuring tactics and strategy. Academia and the industry have consequently developed strong chess engines over decades which feature many hundreds of techniques and methods, yet a comprehensive literature survey of chess programming techniques does not exist. Combating this, we present a systematic literature survey in tree algorithms for computer chess programs. We discuss traditional Minimax-based approaches which have seen steady development since 1970, as well as novel state-of-the-art approaches employing Monte-Carlo Tree Search. We conclude that a dominant approach is not yet apparent, and suggest further research in MCTS–Minimax hybrids and other mixed-technique approaches.

Date: Friday 31st January, 2021

Supervisor: [redacted]

1 Introduction

Chess has been an academic and industry interest for at least several decades and is recently becoming an increasingly popular. Many millions of players play it regularly and top-players use computer chess programs to gain insight and competitive advantages. However, few academic literature surveys exist, and none are recent. This paper presents the state of the art in chess program design in a systematic survey covering both traditional and newer Monte-Carlo Tree Search (MCTS) approaches.

Chess is a massively popular two-player zero-sum perfect-information game demanding strategy, tactics, and board evaluation. World chess federation *Fédération Internationale des Échecs* (FIDE) estimates there are more than 605 million regular chess players in the world,[1] of which 360,000 are tournament players,[2] and 1700 are expert-level *grandmasters*.[3] Moreover, online chess is becoming increasingly popular. Driven by the Coronaviris pandemic and popular Netflix series *The Queen's Gambit*, online platform *Chess.com* has seen the number of live chess players increase 160 percent,[4] and online chess video streaming hit records of almost one million unique viewers.[5]

Historicaly, chess is often celebrated as the grandfather of modern artificial intelligence. In early 1948, computer pioneers such as Alan Turing proposed the first chess programs, and computer chess since has remained an interest for scientists and engineers [1]. In 1966, Velski *et al.* used a M-20 computer to perform the first match between a human and a computer [2], which was won by the human. Subsequent decades have seen a variety of chess programs emerge and a strong effort towards improving the playing strength of chess programs, and in 1997 IBM's computer DEEP BLUE defeated reigning world champion Garry Kasparov [3]. Since then, no human chess player has seriously challenged chess engines.

Chess engines are now used widely by players from all levels to analyse positions and games, including grandmaster-level players to aid in tournament preparation. Strong players with access to strong chess engines enjoy an advantage and consequently a desire to measure and compare the relative strength of chess engines has emerged. Modern chess programs now far exceed superhuman level and their strength is measured in computer chess championships such as *Top Chess Engine Championship* (TCEC) and *World Speed Chess Championship* (WSCC) [4], [5] in which hundreds of chess programs compete. They have elected the open-source *Stockfish*[6] as the reigning champion, and it employed in most popular online chess platforms.[7]

Despite regular progress, however, there are few literature surveys on computer chess methodology. Game tree pruning has been reviewed, but not recently [6]. Other reviews are too specific, detailing only deep-learning-based chess techniques [7], or do do not pertain to chess specifically [8]. Moreover, seminal work by Silver et al., has reshaped assumptions about how to develop state-of-the-art chess engines [9], [10], yet a comparative review of the literature has not emerged. Novel techniques in traditional engines also have continued to emerge at steady pace, which also have not seen literature survey review.

Addressing this knowledge gap, this article presents a *systematic* survey on tree-based computer

[1]https://web.archive.org/web/20160904032818/http://www.fide.com/component/content/article/1-fide-news/6376-agon-releases-new-chess-player-statistics-from-yougov.html

[2]http://ratings.fide.com/download.phtml

[3]https://www.fide.com/news/782

[4]https://www.chess.com/blog/erik/incredible-second-wave-of-interest-in-chess

[5]https://www.chess.com/news/view/blockchamps-sets-chess-viewership-record

[6]https://stockfishchess.org/

[7]Chess.com, Lichess.org, and Chess24.com all use variants of the Stockfish engine.

chess techniques and methods. The contribution comprises four parts:

1. Section 2 introduces the high-level design of computer chess engines, as well as common enhancements that strong engines all implement such as transposition tables, iterative deepening, opening books, and endgame tablebases.
2. Sections 3 and 4 present the implementation of traditional Minimax chess engines, their common variations, and how domain-level heuristics selectively extend candidate move exploration (search extension), reduce it (search reduction), or eliminate it from search (pruning).
3. Section 5 Discusses four common techniques to evaluate leaf node board evaluations, which are used in both Minimax-based and MCTS-based approaches.
4. Section 6 introduces the new and competitive approach based on Monte-Carlo Tree search. We discuss a competitor to the state of the art, *AlphaZero*, and several variations and improvements upon the initial work, such as MCTS–Minimax hybrids.

Method

This survey incorporates literature which has been obtained from a systematic search process. We query databases *Google Scholar*, *dblp.org*, and *ACM Digital Library* for permutations of keywords *chess*, *minimax*, *tree search*, *Monte-Carlo*, *time management*, *alpha beta pruning*, *game tree*, and *search tree*, while limiting the search to one hundred articles per query. We subsequently filter results if they are irrelevant to the survey topic, of low academic quality, of a publication date before 1990 and with low citation count, or unobtainable in digital format.

This process resulted in 132 unique articles, which were further filtered based on whether they contribute key concepts such that this article can present more manageable bibliography. Additionally, individual articles are only discussed briefly if they do not contribute novel techniques. Because the history of chess research is long, some articles predating 1990 are included where they establish fundamental concepts. We also include academic articles cited in *Chess Programming Wiki*,[8] which contains technical descriptions of topical techniques and implementations. Finally, informational references are provided as footnotes throughout this article.

2 Background

We now cover essential theory, developed over decades, which forms the basis for traditional chess engines. Chess is a sequential two-player perfect-information game [6]. Each game is described by a list of moves from the starting position, and each intermediary game state corresponds to a position on the chess board. We note that the game state is not only predicated by the position on the board, but also other factors, such as the castling permissions, the *en-passant* rule, the three-move repetition rule, and the fifty move repetition rule, all of which depend on the exact sequence of moves [11].

Formally, a finished game G of length n is a sequence of actions (in chess terms, a *ply*, two of which from a *move*): $G = (a_1, ..., a_n)$ in which each player chooses an action $a_t \in A_t$ from the set of valid moves A_t in turn. We denote that state of the game $G_t = (a_1, ..., a_t)$ as the actions up until move t. A_t is dependent on the game state at that point, where G_t is a prefix of G.

[8]https://www.chessprogramming.org

That is, $A_t = f(G_{t-1})$ for some legal action generator $f(\cdot)$. The result of a game, R is a win (+1 point), a lose (0 points), or a draw (1/2 a point), depending on whether the last move of a game resulted in checkmate, or a draw is claimed based on the rules of chess or otherwise agreed by the players. In practice, chess engines emulate agents interacting with the game state. Their goal during play is to maximise the result. That means, at time t, play the action a_t for which the evaluation $S_{a_t} = P(R = 1|a_t, G_{t-1}) + \frac{1}{2}P(R = 1/2|a_t, G_{t-1})$ is maximal.

2.1 Traditional Chess Program Design

Traditional chess engines comprise of four components: a representation of the board, a function to generate moves based on the game state, a tree search implementation, and a function to evaluate a given game state [6]. Many different board representations exist, which can be broadly categorised in piece-centric (e.g., piece lists and bitboards), square-centric (e.g., mailbox and piece arrays) and hybrid solutions.[9] Board representations feature trade-offs in speed and memory requirements, but we will not cover them in detail for the purpose of this review. The interested reader can find further treatise in [12] and [13].

The move generation component and tree search component work in tandem to generate and explore candidate moves, which are subsequently ranked by the evaluation function [6], [13]. The total number of possible moves grows exponentially with each consecutive move, and it is impossible for the evaluation function to determine the true strength of each move. Hence, a more precise evaluation is achieved by exploring the game tree systematically and evaluating the resulting positions at the leaf nodes. The evaluations are combined in the MINIMAX algorithm, or a variant thereof, further presented in Section 3, which yields the improved evaluation for the game state. Typically, strong chess engines increase or reduce the search depth based on game heuristics, known as *search selectivity*, which we further present in Section 4. Finally, the evaluation function approximates the winning chances of the outcome at the leaf nodes. We discuss the evaluation function further in Section 5.

2.2 Common Enhancements

In addition to the aforementioned key components of chess engines, several other techniques are employed to increase playing strength. Given a game state G_t and who candidate actions a_1 and a_2. In certain scenarios, we find that $G_t \cup (a_1) \cup (a_2)$ may be equal to $G_t \cup (a_2) \cup (a_1)$ in terms of board position and order-dependent rules (e.g. 50-move rule). These actions can lead to the same effective game state *by transposition.* In chess, transposition often occurs and strong engines make use of *transposition tables*, effectively caches, which store computed recursive evaluations of certain positions during search. If a subsequent search discovers an equal position, it does not need to be recomputed [14]. This, in addition to other performance enhancing techniques such as parallelism [15], can significantly increase playing strength.

As most chess games are timed events, proper time management is often a crucial component of chess engines, and time management strategy should be adaptable to different chess time controls. Many strategies exist [16], [17]. A common and flexible technique is *iterative deepening*, in which the search depth starts low and is increased once each search iteration completes. This approach has two benefits: firstly, a candidate good move is always available (i.e., the node with highest evaluation at the last completed search), enabling the parent time management strategy to abort search at any time. Secondly, heuristic *search selectivity* techniques (See Section 4)

[9]https://www.chessprogramming.org/Board_Representation

can reorder candidate moves in between search iterations and prioritise high-potential candidate moves. Other time management super-strategies exist, and they often account for many factors, including game phase, time remaining on the clock, phase of the game, and the complexity of the position. A more comprehensive analysis is presented in [18].

Often, engines also use precomputed *opening books* and *endgame tablebases* (EGTs) during the start and end phases of the game. Opening books store strong responses to common opening actions. They are typically constructed by expert players, although some automatic opening book creation techniques have proved to be strong contenders [19]. Conversely, endgame tablebases store exact best-moves for positions in the final stages of the game [20]. They are typically created through an automatic process using retrograde analysis (i.e., by generating reverse moves from all possible winning, losing, and drawn terminal positions). The advantage of this approach is that, if the generation process is exhaustive, they are known to be correct. The current state-of-the art is *Syzygy Bases*, which store the correct sequence of moves in all possible positions with up to six pieces (149.2GiB) or seven pieces (16.7TiB).[10]

3 Minimax-based Search Algorithms

Almost all chess programs search using a variation or optimised version of the *Minimax* algorithm. In principle, all moves are considered from the current node (i.e., the current game state), iteratively towards the leaf nodes specified by the search depth [6], [21]. For a two-player game such as chess, the minimax algorithm finds the move with the highest evaluation at the root node (the 'max' step). To do so, it determines the evaluation which is worst for the player, and thus best for the opponent one action ahead (the 'min' step), and so forth. At every level, the algorithm assumes that the player and the opponent both play the best theoretical moves, and consequently the algorithm minimises and maximises the evaluations of the actions at every depth.

Some chess programs instead implement *Negamax* [6], which is equivalent to Minimax by emulation. The algorithm does not have two interlinked steps, but only a maximisation step, and reformulates the minimisation step in terms of maximisation, at each node in the tree. This yields an equivalent evaluation due to the equality $\min(S_{a_1}, S_{a_2}) = -\max(-S_{a_1}, -S_{a_2})$ for scores S_α after a move α. The primary benefit is the simplification of the implementation as each evaluation yields a recursive call to the same procedure.

Note that the number of nodes grows exponentially with the depth of the search tree. As a consequence, strong chess engines benefit from minimising exploration relative to evaluation confidence. In other words, if the best action is a_b, and for two actions a_1 and a_2, and if the credence $Cr(a_1 = a_b) > Cr(a_2 = a_b)$, then the program can gain playing strength by exploring a_1 more deeply than a_2. This gives rise to tree pruning techniques, which aim to minimise the number of positions the algorithm explores and evaluate.

3.1 Alpha-Beta ($\alpha - \beta$) Search

Alpha-Beta pruning, first popularised by Knuth and Moore [22], is a technique for search tree reduction by tracking a valuation bound $[\alpha, \beta]$ on the valuation of the subtree during search [6]. By employing a depth-first search approach, Knuth and Moore showed that certain subtrees can be eliminated from search. In particular, consider a search depth of two an action at timestep

[10]https://www.chessprogramming.org/Syzygy_Bases

t, $a_{t,1}$ which is fully explored and evaluates to a score of $S_{t,1} = 1/2$. If another action $a_{t,2}$ with score leads to move for the opponent $a_{t+1,1}$ with score $S_{t+1,1} < S_{t,1}$, then we can ignore all other responses of the opponent to $a_{t,2}$ because $a_{t,1}$ is a preferable move to $a_{t,2}$. Effectively, this is an application of the minimax assumption that the opponent will always play the best possible move. For search depths higher than three, both an upper bound and a lower bound should be maintained because both players can affect the evaluation at a given depth.

There are several variations of alpha-beta search. Firstly, SSS*, first introduced by Stockman [23] in 1979 has provably equal best-case performance when contrasted to alpha-beta search, yet shows potential speedup by evaluating promising nodes first and exploring the entire search tree in parallel, akin to breadth-first search techniques. The speedup of SSS* can attain depends on the properties the search tree. However, subsequent literature conjectured that due to the parallel nature of the algorithm, it has worse memory bounds, and needs significant tuning to be effective, to consequence of impracticality [24]. Reformulations of SSS* in terms of a depth-first search algorithm derived from alpha-beta yielded AB-SSS* and SSS-2 which largely solved memory issues [25].

As shown in early work by Marsland [26], more nodes can be pruned in narrower search windows $[\alpha, \beta]$. Fail-Soft alpha-beta pruning [27] proposed by Fishburn in 1983 which, through experiments, shows further search evaluation reduction by up to 16 percent. This approach places stricter soft bounds on the subtree evaluation. If the evaluation yields a value outside the soft bounds, the subtree must be searched again with the stricter bounds. In combination with transposition tables, which often store bounds, the resulting implementation searches fewer nodes when transpositions occur.

3.2 Aspiration Windows and Null-Move Search

Further improvement can be obtained using *aspiration windows* [28], [29] which reduce the search space by estimating the bounds of alpha and beta in alpha-beta search based on the evaluation of the current node. If a game state G_t has a score S_t. The aspiration window is $[\alpha, \beta] = [S_t - \epsilon, S_t + \epsilon]$ where typically $\epsilon = 0.5$ equalling to half a pawn. This approach leads to the *gradual widening* technique,[11] in which the bounds are increased when the evaluation exceeds the aspiration window asymmetrically. For example, if the valuation fails high, the upper-bound is increased gradually until passing bounds are found.

Principal Variation Search (PVS) is based on null-move search [26]. If the regular alpha-beta search process finds a sequence of actions without cut-off from the root node to the leaf node, it is selected as the principal variation, and it is likely that deviations early in the sequence will not lead to a better valuation. Therefore, PVS will subsequently test exploration of moves at lower relative depth to the root node using a null-move search which tests whether the new move can be pruned due to the impossibility of a better valuation than the principal move. Such a search is cheaper than full exploration of the subtree.

In the extreme case that the aspiration window $\lim_{\epsilon \to 0} \beta - \alpha = \epsilon$, tree pruning can be applied maximally, called *null-move search*, leading to highly efficient search algorithms [24], [27], [30]. The Negascout [31] algorithm is based on null-move search and has provably better performance, in particular in combination with transposition tables, although alpha-beta search remains more popular in practice. Applying the same principle to the reformulated version of SSS* in minimax terms (i.e., SSS-2), resulted in the *Memory-enhanced Test Driver* algorithm [25], often abbreviated MTD(f), which outperforms almost all versions of alpha-beta search, as well

[11] Some chess programs, such as Crafty, employ this technique. https://www.chessprogramming.org/Crafty

as SSS* [24].

In 2011, Rutko presented *Fuzzified Game Tree Search*, more commonly known as *Best Node Search* (BNS), which is the current state of the art in game tree search, and was found to be the most efficient minimax algorithm, outperforming SSS*, NegaScout, and MTD(f), including their variations with ten percent speedup [32]. BNS guesses the minimax value, and iteratively calls alpha-beta search with null-move aspiration windows to determine if the minimax value in the subtrees are better or worse than than the current guess. Although BNS has theoretical performance gains, it has not been tested on chess, and it is unclear whether it leads to practical strength gains.

4 Search Selectivity

In most turn-based games, including chess, the quality of the actions A_t at timestep t varies greatly, and chess engines can thus benefit from exploring more promising variations more extensively than variations which are clearly worse. Moreover, the complexity of evaluation for game states also varies. In chess terms, simple board positions are 'easier' to evaluate than thuse with many tactical possibilities. This gives rise to *selective search* [33], [34], which are techniques to prioritise, extend, or reduce subtrees based on heuristic methods. We discuss these methods, and several specific techniques, in turn.

4.1 Search Extensions

Search algorithms suffer from the *horizon effect* at the leaf nodes [6], [35], [36]. Often, it is possible to delay an inevitable negative consequence such as checkmate by several actions, leading to significant differences between approximated and true evaluation. Formally, consider a shallow evaluation S_d at depth d to be significantly different from the evaluation $S_{d+\delta}$ for some positive integer δ. If the search depth s is fixed and bounded by $d < s < d + \delta$, then the search will yield an evaluation with a significant error from the true evaluation. The evaluation effect may be positive or negative, and if discovered early on, may permit for more aggressive alpha-beta pruning (see Section 3.1).

To combat this effect, several *search extensions* exist which aim to resolve tactical positions at the leaf nodes. Many search extensions exist, and all are based on domain-level heuristics pertaining to chess. Among others, these extensions include evaluating whether checks (re)captures, passed pawn moves, single-reply moves, or similar heuristics influence the game evaluation. In principle, the goal of search extensions is to evaluate 'quiet' positions as much as possible, also known as *quiescence search* [6], [34], as they give a more accurate representation of the evaluation score at the leaf node than those which create tactics.

4.2 Search Reductions and Pruning

In certain cases, it may be beneficial to instead reduce or remove subtree evaluation based on domain-specific heuristics. We distinguish between two techniques: *pruning*, in which subtrees are removed entirely from search, and *reduction*, in which the the recursive depth is reduced. While some pruning techniques guarantee the equality of the evaluation after pruning (*strong pruning*), heuristic approaches do not have this guarantee (*weak pruning*), yet still result in an increase in playing strength [6]. Heuristic pruning techniques present some risks: inherently, there may exist specific sequences of actions resulting in an evaluation different from the evaluation at the leaf node. In this way, the horizon effect is created willingly, and strong chess engines typically balance heuristic pruning and search extensions to maximise play strength [37].

Several pruning techniques, both strong and weak, have been proposed in literature. The most common is alpha-beta pruning (see Section 3.1), but we will mention several others here briefly. *Futility Pruning* [38] discards candidate actions which cannot increase the lower-bound of the alpha-beta search at nodes which are distance one from the search horizon. It has been extended to *Deep Futility Pruning*,[12] *Late Move Pruning*,[13] *Extended Futility Pruning* [39] and *AEL pruning* [40], the last of which has been tested experimentally to attain at least 20 percent fewer nodes explored while maintaining play strength. *Multi-cut pruning*, and variations *Enhanced Multi-Cut Pruning* and *Enhanced Forward Pruning* cut-off multiple lines of play if they are not likely to influence the root node evaluation [41].

Levy et al. proposed the Search Extension (SEX) algorithm [33], which combines several search extension and reductions, considering the latter negative extensions. In essence, their approach aims to, at each node in the search tree, sort subtree actions based on their tactical merits. By exploring tactical moves, such as checks and captures first, more aggressive strong pruning can be applied in subsequent candidate top-level moves. Their work also makes use of *fractional reductions* in which multiple heuristic rules each contribute fractional depths in the recursive search call. Fractional depths may constitute increased or reduced whole-ply depth in subiterations of the algorithm. Their algorithm was improved throughout 1981–1988 with a focus on distinguishing which moves lead to interesting variations. The SEX algorithm had wide theoretical influence but is no longer commonly employed.

5 Evaluation Functions

After search yields a leaf node, it is evaluation function which estimates the winning chances at the given game state. The result is often a real number which, for the benefit of the chess user, is scaled according to traditional assumptions about piece values (i.e., a valuation of +1 means that the white player has an advantage of one pawn). Evaluation function implementation differs greatly between implementation and is often at the crux of chess program strength. Many different approaches to current board evaluations exist, but they can be broadly categorised into (i) manually tuned functions, (ii) automatically tuned functions of crafted features, (iii) neural-network-based approaches, and (iv) evolutionary or genetic approaches. We discuss each in turn.

The first chess program evaluation functions developed manually tuned [2], but reference implementations for new program designers often also advocate for such simple and straightforward approaches [13]. They consist of applying broad rules which are widely held to be valid. (e.g., a knight is thrice as valuable as a pawn, centre control and king safety is important, etc.), and assigning to each rule a manually tuned weight. Weighted features can be combined in linear or nonlinear combinations,[14] and because chess theory is extensive, many rules can be applied on a trial-and-error basis. This approach can yield strong chess engines given sufficient effort.

Strong chess engines such as employ a combination of handcrafted features in combination with autotuning. Many engines employ this approach but the most notable is Stockfish. In partic-

[12]https://home.hccnet.nl/h.g.muller/deepfut.html

[13]This reduction is combined with Late Move Reductions in the strongest open-source chess engine Stockfish. See https://www.chessprogramming.org/Futility_Pruning

[14]https://www.chessprogramming.org/Evaluation

ular, Stockfish incorporates dosens of carefully selected features into its nonlinear evaluation function, including those pertaining to material, attacks, pawn structure, game phase, pins, king safety, space, mobility, centre control, passed pawns, material imbalance, and winnable pieces.[15]. These features are subsequently automatically tuned using the *Confident Local Optimization for Noisy Black-Box Parameter Tuning* framework (CLOP).[16] This is a stochastic gradient descent approach which iteratively finds more optimal solutions of the weights, which are determined by self-play.

Academic literature has proposed several neural network based approaches. Primarily, these approaches offer the benefit that no manual selection of features is necessary. As deep learning methods have been reviewed recently by Mehta et al. [7], we here only mention the most prominent. *Giraffe*, proposed by Lai attains master-level play while searching an order of magnitude slower than weaker chess engines. *DeepChess* [43], [44] attains grandmaster-level play, inputting two chess positions and outputting a prediction on which is better. Sabatelli et al. evaluates many different network architectures and performed supervised learning taking human grandmaster games as input vectors and Stockfish evaluation results as the corresponding labels. Finally Wan and Kaneko investigate a novel type of regularisation that can improve upon typical L_2-regularisation for game trees [46]. Many other network-based approaches exist, but are beyond the scope of this review [47]–[55].

Evolutionary approaches have also been tried, as the complexity and intricacy of valuations lends itself to an incremental approach. Typically, these evaluation functions are either genetic individuals trained in populations or neural networks of which the weights are trained in similar fashion. *Pocket Fritz 2.0* [56] shows master-level strength using unsupervised learning through self-play. In a hybrid approach, *Tempo* [57] generates neural networks in a modular fashion which are invariant to the network size and static hyperparameters. David et al. first presented experimental proof that grandmaster-level play can be attained using genetic algorithms augmented by coevolution, although the algorithm still relies on crafted input features. There are other similar attempts, some attaining strong sub-master-level play [59], while others do not contest traditional approaches at all [60].

6 Monte-Carlo Tree Algorithms

Traditionally, alpha-beta pruning chess engines have shown strongest play, although other techniques have been attempted. Recently, however, Monte-Carlo Tree search (MCTS) algorithms have seen a rise in popularity since DeepMind introduced *AlphaZero* in 2017 based on their prior work on *AlphaGo* in 2016 [10]. We present their novel approach in Section 6.1 and discuss subsequent literature in Section 6.2.

6.1 AlphaZero

AlphaZero [9], [61] is a state-of-the art chess engine which outperforms Stockfish, heralded by the Top Chest Engine Championship [4] as the strongest contender. Their approach is dissimilar from regular tree based search in that it iteratively plays games during search, selecting continuously a action move based on a stochastic Monte-Carlo process $\pi_\mathbf{t}$ until the game concludes in a win, draw, or loss. The stochastic process selects moves which are most unexplored, have high evaluated score, and high average win rate in previous explorations.

[15]https://hxim.github.io/Stockfish-Evaluation-Guide/
[16]https://www.remi-coulom.fr/CLOP/

Their evaluation function $f_\Theta = (\mathbf{p}, v)$ is a neural network which not only yields the estimated result of the position scaled to $[-1, 1]$ (i.e., $v = 2R - 1$ in earlier notation), but also a policy vector $\mathbf{p}$ which, for each candidate action $p_a \in \mathbf{p}$ represents the normalised evaluation of each candidate action. The neural network is trained on game data with an error function which comprises the squared difference between the valuation v and the result of the game at each move, and the cross-entropy loss of the policy vector relative to $\pi_\mathbf{t}$.

Furthermore, their approach is completely based on self-play. At each training step, the program plays itself and updates the neural network weights by the average gradient for each weight update of both players. The update employs the Bayes' theorem for updating the weights. The algorithm does not receive prior information about the game rules, except that only legal moves are considered by the MCTS process, and that noise is added to the Bayesian optimisation update priors to stimulate exploration.

The work by Silver et al. is notable because it not only contests widely held beliefs that alpha-beta algorithms or variants thereof are inherently stronger than differing approaches, but also because their work is general, as they apply the framework to Go and Shogi. The work also contests the notion that programs which visit more nodes during search are inherently stronger. Furthermore, the authors demonstrate how AlphaGo selects natural moves in complicated positions while traditional chess programs often struggle in these situations [35].

6.2 Monte-Carlo Tree Search Extensions

Since the publication of AlphaZero, more strong competitors have emerged [62], including strong chess engines such as *Komodo MCTS*[17] *Rybka*,[18], adaptations of Stockfish,[19][20] and an open-source engine based on AlphaZero directly, *LeelaChess*.[21] LeelaChess has subsequently placed second in the Top Engine Chess ChampionShip [4], although the developers note that many hyperparameters are approximations AlphaZero's implementation as many technical details are not released in the original publications. Wan and Kaneko show how to augment AlphaZero's training procedure for reduced computation requirements for equal performance.

But MCTS techniques shown to be broadly applicable to other similar two-player games. Go and Shogi have since seen seen novel research in their fields [63], as well as in Chinese chess [64]. But but also in real-time games, MCTS has made an appearance. Further research by DeepMind showed that MCTS-based approaches can achieve state-of-the-art performance in playing Atari video games [62].

Although previous literature reviews have seen dosens of MCTS extensions, many have not yet been applied to chess, which may prompt interesting avenues for further research [8], [65]. For example, MCTS extensions have shown to greatly improve evaluation performance, including MCTS–minimax hybrid approaches proposed by Baier and Winands [66] who argue that MCTS approaches can make tactical mistakes by falling into traps because only the most promising variations are sufficiently explored. They propose three methods to integrate traditional Minimax approaches into the MCTS process, thus ensuring that crucial variations are not missed, and thus outperform classical MCTS. Their approach untested on chess, however. Huang propose rollout techniques to unify MCTS and minimax architectures [37].

[17]https://komodochess.com/Komodo13.htm
[18]http://www.chessbase.com/newsdetail.asp?newsid= 5075
[19]https://github.com/OhJayGee/SugaR
[20]https://github.com/amchess/ShashChess
[21]https://github.com/leela-zero/leela-zero

Another MCTS extension, *Rapid Action Value Estimation* builds on the assumption that for some games, the valuation delta of an action a_1 is invariant to depth within the search tree [67]. In other words, the order of the actions does not matter. This can be the case when the pieces moved by two actions do not interact with each other. By extracting this *temporal invariance* from the evaluation procedure, the search speed may see significant speedup, although the approach has been tested on Go, it has also not seen experimentation on chess [68], [69].

Parallelism, is also more straightforward to apply in MCTS programs in contrast to alpha-beta search [8], [67]. Alpha-beta algorithms operate on a single search tree for which the breadth is unknown at the start of search, and thus reallocation of subtrees to processors is common. MCTS simplifies parallelism as each simulation of a game can be perform independently, and in parallel. Synchronisation of game results is not immediately necessary for simulation of further games to occur, and an *eventually consistent* data sharing approach can be applied until the move must be played.

7 Conclusion and Further Research

Chess is more popular than ever, and although chess continues to pique the interest of academics in literature and engineers in design alike, it has seen few comparative literature studies. This article presents both traditional minimax-based chess algorithms and new developments in MCTS-based approaches. Although the MCTS contender AlphaZero has beat previous iterations of Stockfish, this is no longer the case for newer versions. Still, the differences in playing strength are small. Both approaches have benefits: Minimax-algorithms are deterministic and provide strong guarantees that certain variations are not missed, while MCTS algorithms excel in finding long-term advantages beyond the depth that exhaustive Minimax search can provide. It is clear that the dominant strategy is not yet determined, and may only become apparent with further experimental research and application.

Many extensions and variants of Monte-Carlo algorithms exist in literature, including several MCTS–minimax hybrids, but have not been applied to chess. Conversely, the state-of-the-art Best-Node Search has not yet seen application to chess. These could be interesting avenues for further research. The creative reader may consider other combinations of techniques which remain unexplored. For example, MCTS could be combined with a genetic-programming-based evaluation function. Search reductions, which are common in alpha-beta pruning algorithms, could also speed up evaluation in MCTS by determining game results when they become apparent. MCTS algorithms still evaluate fewer positions than their counterparts. As collaborative efforts between academics and the industry further investigate these methods further, they may well find further advantages yet unexplored.

References

[1] A. Bozhinovski and F. Jankuloski, "Chess as Played by Artificial Intelligence," en, 2020.

[2] V. D. Herik and H. Jaap, "Computer chess: From idea to DeepMind," en, *ICGA Journal*, vol. 40, no. 3, pp. 160–176, 2018.

[3] M. Campbell, A. J. Hoane, and F.-h. Hsu, "Deep Blue," en, *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002.

[4] G. Haworth and N. Hernandez, "TCEC13: The 13 th top chess engine championship," en, *ICGA Journal*, vol. 41, no. 2, pp. 92–99, 2019.

[5] J. Krabbenbos, J. van den Herik, and G. Haworth, “WSCC 2019: The World Speed Chess Championship,” en, *ICGA Journal*, vol. 41, no. 4, pp. 237–240, 2019.

[6] T. A. Marsland, “A Review of Game-Tree Pruning,” en, *ICGA Journal*, vol. 9, no. 1, pp. 3–19, 1986.

[7] F. Mehta, H. Raipure, S. Shirsat, et al., “A Survey of Deep Learning on Chess,” en, vol. 10, no. 4, p. 5, 2020.

[8] C. B. Browne, E. Powley, D. Whitehouse, et al., “A Survey of Monte Carlo Tree Search Methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[9] D. Silver, T. Hubert, J. Schrittwieser, et al., “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm,” *arXiv:1712.01815 [cs]*, 2017.

[10] D. Silver, A. Huang, C. J. Maddison, et al., “Mastering the game of Go with deep neural networks and tree search,” en, *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[11] W. C. Federation, “FIDE laws of chess,” 2008.

[12] V. Vučković, “THE COMPACT CHESSBOARD REPRESENTATION,” en, *ICGA Journal*, vol. 31, no. 3, pp. 157–164, 2008.

[13] V. Manohararajah, “Rajah: The Design of a Chess Program,” en, *ICGA Journal*, vol. 20, no. 2, pp. 87–91, 1997.

[14] R. M. Hyatt and T. Mann, “A LOCKLESS TRANSPOSITION-TABLE IMPLEMENTATION FOR PARALLEL SEARCH,” en, *ICGA Journal*, vol. 25, no. 1, pp. 36–39, 2002.

[15] L. Li, H. Liu, H. Wang, et al., “A Parallel Algorithm for Game Tree Search Using GPGPU,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2114–2127, 2015.

[16] R. Šolak and V. Vučković, “Time Management During a Chess Game,” en, *ICGA Journal*, vol. 32, no. 4, pp. 206–220, 2009.

[17] H. Baier and M. H. M. Winands, “Time Management for Monte-Carlo Tree Search in Go,” en, in *Advances in Computer Games*, H. J. van den Herik and A. Plaat, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2012, pp. 39–51.

[18] S. Markovitch and Y. Sella, “Learning of Resource Allocation Strategies for Game Playing,” en, *Computational Intelligence*, vol. 12, no. 1, pp. 88–105, 1996.

[19] T. R. Lincke, “Strategies for the Automatic Construction of Opening Books,” en, in *Computers and Games*, T. Marsland and I. Frank, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2001, pp. 74–86.

[20] V. B. Zakharov, M. G. Mal’kovskii, and A. I. Mostyaev, “On Solving the Problem of 7-Piece Chess Endgames,” en, *Program Comput Soft*, vol. 45, no. 3, pp. 96–98, 2019.

[21] M. Luštrek, M. Gams, and I. Bratko, “Why Minimax Works: An Alternative Explanation,” en, p. 6,

[22] D. E. Knuth and R. W. Moore, “An Analysis of Alpha-Beta Priming’,” en, *Artificial Intelligence*, p. 34, 1975.

[23] G. C. Stockman, “A minimax algorithm better than alpha-beta?” en, *Artificial Intelligence*, vol. 12, no. 2, pp. 179–196, 1979.

[24] A. Plaat, J. Schaeffer, W. Pijls, et al., “Best-First and Depth-First Minimax Search in Practice,” *arXiv:1505.01603 [cs]*, 2015.

[25] A. de Bruin and W. Pijls, “Trends in game tree search,” en, in *SOFSEM’96: Theory and Practice of Informatics*, K. G. Jeffery, J. Král, and M. Bartošek, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1996, pp. 255–274.

[26] T. A. Marsland and M. Campbell, “Parallel Search of Strongly Ordered Game Trees,” *ACM Comput. Surv.*, vol. 14, no. 4, pp. 533–551, 1982.

[27] J. P. Fishburn, “Another optimization of alpha-beta search,” *SIGART Bull.*, no. 84, pp. 37–38, 1983.

[28] H. Kaindl, R. Shams, and H. Horacek, “Minimax Search Algorithms With and Without Aspiration Windows,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 13, no. 12, pp. 1225–1235, 1991.

[29] R. Shams, H. Kaindl, and H. Horacek, “Using Aspiration Windows for Minimax Algorithms,” in *IJCAI*, 1991.

[30] M. S. Campbell and T. A. Marsland, “A comparison of minimax tree search algorithms,” en, *Artificial Intelligence*, vol. 20, no. 4, pp. 347–367, 1983.

[31] J. Pearl, “The solution for the branching factor of the alpha-beta pruning algorithm and its optimality,” en, *Commun. ACM*, vol. 25, no. 8, pp. 559–564, 1982.

[32] D. Rutko, “Fuzzified Tree Search in Real Domain Games,” en, in *Advances in Artificial Intelligence*, I. Batyrshin and G. Sidorov, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2011, pp. 149–161.

[33] D. Levy, D. Broughton, and M. Taylor, “The Sex Algorithm in Computer Chess,” en, *ICGA Journal*, vol. 12, no. 1, pp. 10–21, 1989.

[34] D. F. Beal, “A generalised quiescence search algorithm,” en, *Artificial Intelligence*, vol. 43, no. 1, pp. 85–98, 1990.

[35] H. J. Berliner, “Some necessary conditions for a master chess program,” in *Proceedings of the 3rd international joint conference on Artificial intelligence*, ser. IJCAI’73, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 77–85.

[36] H. Kaindl, “Tree Searching Algorithms,” en, in *Computers, Chess, and Cognition*, T. A. Marsland and J. Schaeffer, Eds., New York, NY: Springer, 1990, pp. 133–158.

[37] B. Huang, “Pruning Game Tree by Rollouts,” en, p. 13,

[38] K. Hoki and M. Muramatsu, “Efficiency of three forward-pruning techniques in shogi: Futility pruning, null-move pruning, and Late Move Reduction (LMR),” en, *Entertainment Computing*, Games and AI, vol. 3, no. 3, pp. 51–57, 2012.

[39] O. David-Tabibi and N. S. Netanyahu, “Extended Null-Move Reductions,” en, in *Computers and Games*, H. J. van den Herik, X. Xu, Z. Ma, et al., Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2008, pp. 205–216.

[40] E. A. Heinz, “AEL Pruning,” en, in *Scalable Search in Computer Chess: Algorithmic Enhancements and Experiments at High Search Dephts*, ser. Computational Intelligence, E. A. Heinz, Ed., Wiesbaden: Vieweg+Teubner Verlag, 2000, pp. 53–61.

[41] Y. Björnsson and T. A. Marsland, “Multi-cut -pruning in game-tree search,” en, *Theoretical Computer Science*, CG’98, vol. 252, no. 1, pp. 177–196, 2001.

[42] M. Lai, “Giraffe: Using Deep Reinforcement Learning to Play Chess,” *arXiv:1509.01549 [cs]*, 2015.

[43] E. David, N. S. Netanyahu, and L. Wolf, “DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess,” *arXiv:1711.09667 [cs, stat]*, vol. 9887, pp. 88–96, 2016.

[44] ——, “DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess,” *arXiv:1711.09667 [cs, stat]*, 2017.

[45] M. Sabatelli, F. Bidoia, V. Codreanu, et al., “Learning to Evaluate Chess Positions with Deep Neural Networks and Limited Lookahead:” en, in *Proceedings of the 7th International Conference on Pattern Recognition Applications and Methods*, Funchal, Madeira, Portugal: SCITEPRESS - Science and Technology Publications, 2018, pp. 276–283.

[46] S. Wan and T. Kaneko, “Building Evaluation Functions for Chess and Shogi with Uniformity Regularization Networks,” in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 2018, pp. 1–8.

[47] D. Real and A. Blair, “Learning a multi-player chess game with TreeStrap,” in *2016 IEEE Congress on Evolutionary Computation (CEC)*, 2016, pp. 617–623.

[48] S. Thrun, “Learning to play the game of chess,” in *Proceedings of the 7th International Conference on Neural Information Processing Systems*, ser. NIPS’94, Cambridge, MA, USA: MIT Press, 1994, pp. 1069–1076.

[49] Y. Efroni, G. Dalal, B. Scherrer, et al., "Beyond the One Step Greedy Approach in Reinforcement Learning," *arXiv:1802.03654 [cs, stat]*, 2018.

[50] B. Oshri and N. Khandwala, "Predicting Moves in Chess using Convolutional Neural Networks," en, p. 8,

[51] C. Ren and Y. Wu, "A New AI Open Problem: WUGU Chess," in *2019 Eleventh International Conference on Advanced Computational Intelligence (ICACI)*, 2019, pp. 274–277.

[52] H. Dehghani and S. M. Babamir, "A GA based method for search-space reduction of chess game-tree," en, *Appl Intell*, vol. 47, no. 3, pp. 752–768, 2017.

[53] J. Baxter, A. Tridgell, and L. Weaver, "KnightCap: A chess program that learns by combining TD(lambda) with game-tree search," *arXiv:cs/9901002*, 1999.

[54] J. Veness, D. Silver, A. Blair, et al., "Bootstrapping from Game Tree Search," en, *Advances in Neural Information Processing Systems*, vol. 22, pp. 1937–1945, 2009.

[55] D. F. Beal and M. C. Smith, "Temporal difference learning for heuristic search and game playing," en, *Information Sciences*, vol. 122, no. 1, pp. 3–21, 2000.

[56] D. B. Fogel, T. J. Hays, S. L. Hahn, et al., "A self-learning evolutionary chess program," *Proceedings of the IEEE*, vol. 92, no. 12, pp. 1947–1954, 2004.

[57] M. Autonès, A. Beck, P. Camacho, et al., "Evaluation of Chess Position by Modular Neural Network Generated by Genetic Algorithm," en, in *Genetic Programming*, M. Keijzer, U.-M. O'Reilly, S. Lucas, et al., Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2004, pp. 1–10.

[58] E. David, H. J. v. d. Herik, M. Koppel, et al., "Genetic Algorithms for Evolving Computer Chess Programs," *IEEE Trans. Evol. Computat.*, vol. 18, no. 5, pp. 779–789, 2014.

[59] E. Vázquez-Fernández, C. A. C. Coello, and F. D. S. Troncoso, "An evolutionary algorithm with a history mechanism for tuning a chess evaluation function," en, *Applied Soft Computing*, vol. 13, no. 7, pp. 3234–3247, 2013.

[60] N. Ramos, S. Salgado, and A. C. Rosa, "Chess Player by Co-Evolutionary Algorithm," *arXiv:1605.06710 [cs]*, 2016.

[61] D. Silver, T. Hubert, J. Schrittwieser, et al., "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," en, *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[62] J. Schrittwieser, I. Antonoglou, T. Hubert, et al., "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model," *arXiv:1911.08265 [cs, stat]*, 2020.

[63] C. F. Sironi, J. Liu, and M. H. M. Winands, "Self-Adaptive Monte Carlo Tree Search in General Game Playing," *IEEE Transactions on Games*, vol. 12, no. 2, pp. 132–144, 2020.

[64] H. Chang, C. Hsueh, and T. Hsu, "Convergence and correctness analysis of Monte-Carlo tree search algorithms: A case study of 2 by 4 Chinese dark chess," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, 2015, pp. 260–266.

[65] Y. Efroni, G. Dalal, B. Scherrer, et al., "How to Combine Tree-Search Methods in Reinforcement Learning," en, *AAAI*, vol. 33, no. 01, pp. 3494–3501, 2019.

[66] H. Baier and M. H. M. Winands, "Monte-Carlo Tree Search and minimax hybrids," in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 2013, pp. 1–8.

[67] S. Gelly, L. Kocsis, M. Schoenauer, et al., "The grand challenge of computer Go: Monte Carlo tree search and extensions," *Commun. ACM*, vol. 55, no. 3, pp. 106–113, 2012.

[68] C. F. Sironi and M. H. M. Winands, "Comparison of rapid action value estimation variants for general game playing," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1–8.

[69] S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go," en, *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.