Transfer Learning in Discrete Zero-Sum Games

COSC470 Research Project

Nathan Cleaver

48941512

njc165@uclive.ac.nz

Supervised by

Dr. Kourosh Neshatian

kourosh.neshatian@canterbury.ac.nz

Computer Science and Software Engineering

University of Canterbury

Christchurch, New Zealand

1 October 2020

Abstract

The construction of a general artificial intelligence is a long-standing goal of computer science. Transfer learning, the ability to transfer knowledge from one context to another, is an essential component of a general intelligence. This work investigates the effects of transfer learning in discrete, zero-sum games. We use distributed parallel Monte Carlo Tree Search to construct game-playing models for chess and a chess variant, Chessplus. Transfer learning is used to allow the chess model to play Chessplus, and results measured. To this end, we also introduce a novel pruning algorithm for Monte Carlo Tree Search. We demonstrate that positive transfer learning in Monte Carlo Tree Search is possible, and in our work, highly successful. The model with transfer outperforms the model without, winning 47.4% of games and drawing 16.9%. Statistical analysis supports the validity of our results. Further analyses offer valuable insights into the specific benefits of transfer between these two domains.

Contents

List of Symbols and Notation

In Game Theory:

S	The set of states reachable in a game.
S_T	The set of terminal states reachable in a game.
A	The set of actions playable in a game.
$f: S \times A \rightarrow S$	The state transition function, which returns the state resulting from an action.
$g: S \times A \rightarrow S$	The reverse transition function, $g(s_i, a_{a-1}) = s_{i-1}$
$R: S_T \rightarrow \mathbb{R}^k$	The reward function, which returns a real number for each of k players.
$E: S \rightarrow \mathbb{R}^k$	The evaluation function, which approximates R for non-terminal states.

In Monte Carlo Tree Search:

ρ	The Upper Confidence Bound for Trees (UCT) score of a given node, used to determine which nodes to explore during Monte Carlo Tree Search.

In Neural Networks:

ϕ	The activation function, which transforms a linear combination of inputs into a single output.
α	The learning rate, which controls the rate of change during back-propagation. This value is usually chosen empirically.
$u(t)$	In Recurrent Neural Networks, the current internal state of a node at time t.

In Transfer Learning:

J	The loss function, which measures the performance of a given model.
A	The transfer algorithm, which takes a model, source data and target data, and returns a new model.
S	A set of data from the source domain.
T	A set of data from the target domain.

1. Introduction

Board games have a long history and a well-established reputation as a measure of intelligence amongst peers. In the modern era, the intricate strategies and hidden consequences in games such as chess, shogi and go have attracted scientists looking to progress the state of Artificial Intelligence (AI). Such events as the defeat of chess grandmaster Garry Kasparov by the computer program Deep Blue and AlphaGo's more recent conquest over world champion go player Lee Sedol are strong indicators that we may be in the final stages of solving these games [1].

New games such as Arimaa have been created with the specific intention of being difficult for computers to play, by introducing additional complexities or increasing the search space [2]. Further, variations of existing games, such as chess, present new challenges for AI researchers: A shortcoming of the most common AI methods is that they generalize poorly to other domains. That is, an AI that has learned to defeat grandmaster-level chess players would struggle to win a game of Arimaa against an amateur-level player. This is a poor reflection of how humans really learn – the two games have a lot in common, and one could reasonably expect a grandmaster chess player to be a strong Arimaa player after only a brief introduction to the game.

Transfer Learning is the branch of AI which explores how to transfer knowledge from one domain to another, related domain. This is particularly useful in instances where the target domain contains insufficient data for common neural network-based learning methods such as supervised or unsupervised learning. Such instances would include, among other examples, new games or game variants.

This research looks at the transfer of learning from the domain of chess to the domains of chess-inspired games known as chess variants. While our research focuses on one variant, Chessplus, we expect results to generalize strongly to other variants. As a by-product of the research direction, we also discuss the practical limitations of an existing search algorithm, Monte Carlo tree search, and introduce a novel approach for overcoming these limitations.

1.1 Motivation and Aims

One of the foremost long-term goals of modern AI research is to create general artificial intelligence. To this end, many advances in the last two decades have contributed new approaches, architectures and heuristics for learning within a single domain or solving a specific problem. However, there is comparatively little current literature on the abstraction of learning from multiple domains such that existing knowledge may be applied to new problems. Abstraction is a defining trait of the human brain, and any general AI must necessarily replicate it.

We identify four goals that this research aims to achieve:

- To explore the many representations of knowledge in the context of machine learning, and to identify from these one most suitable for transfer learning.
- To construct two knowledge models using this representation, one in each of two closely related domains.
- To transfer the first of these models to the domain of the second, such that knowledge may be reused.
- To compare the relative performances of the two models, to determine if knowledge transfer had a positive or negative effect on learning, and the magnitude of this effect.

These goals will be addressed in the Chapters regarding our contributions (Chapter 3 onwards).

1.2 Document Structure

This work begins with a broad overview of existing literature on machine learning, game-playing artificial intelligence, deep learning, transfer learning and Monte Carlo Tree Search, which are relevant to our research. We also include a brief introduction to chess and chess variants, which our research focuses on. Our own contributions are provided in Chapters 3, 4, 5 and 6. Chapter 3 describes the design and implementation of the game representation used in our research. Chapters 4 and 5 look at the application of transfer learning in the context of Monte Carlo Tree Search. Our experimental results and subsequent analyses are given in Chapter 6. We conclude with Chapter 7, which summarizes the work to date, discusses its limitations, and offers a path forward for future work.

2. Background & Literature Review

This research focuses heavily on chess variants, which will not be familiar to all readers. Therefore, this section begins with a brief summary of what variants are and looks at three examples. In each case, we provide an overview of the variant's differences from classic chess and discuss how these changes make the variant more or less suitable for study. Classic chess itself is a common game which should be familiar to most readers, and therefore is not outlined.

Further, this research builds on a broad foundation of prior knowledge from game theory, machine learning, deep learning, mathematics and statistics. We provide an overview of the background knowledge required to understand the research that follows, with emphasis on those topics most heavily relied on.

2.1 Chess Variants

Chess has been the inspiration for a multitude of other board games: Fischer random chess, Kriegspiel, Suicide or anti-chess, Alice chess, Infinite chess, Chessplus, *n*-dimensional chess, Arimaa, and many more. Loosely, we define a chess variant as any turn-based, zero-sum, chess-like game played on one or more boards, with a set of pieces with distinct movement patterns and some conditions for victory. Chess variants may differ in their starting position, piece set, victory conditions, information availability, determinism, board size, shape or dimensions, overarching rules, or any combination of these.

This section describes three chess variants, including Chessplus, the variant chosen for transfer learning in our research. We discuss the similarities and differences these variants have with classic chess and explain why these properties make each variant a poor or strong candidate for transfer learning.

2.1.1 Fischer Random Chess

Fischer Random Chess is a chess variant in which the starting arrangement of non-pawn pieces is randomized, following some rules: the position is mirrored for each player, bishops are on opposite-coloured squares and rooks are on opposite sides of the king, preserving castling. These restrictions result in 960 possible starting positions, giving rise to the variant's alternate name, Chess960. An example starting position is shown in Figure 1.

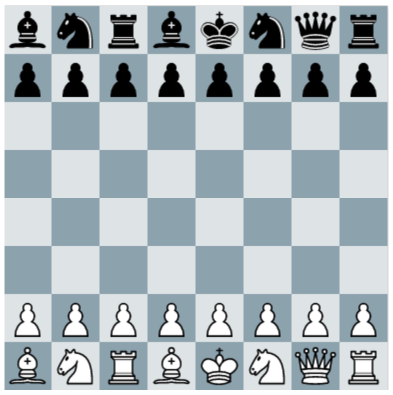


Figure 1: One of 960 possible starting positions for Fischer Random Chess.

The rules of Fischer Random Chess are identical to classic chess, but the different starting positions mean human players are unable to reuse common classic chess openings. Despite differences in the early game, Fischer Random Chess is extremely similar to classic chess – their domains overlap almost completely. We can therefore anticipate a high degree of positive transfer from classic chess to Fischer Random Chess. However, due to their strong similarities, there is little value in demonstrating this; we could not reasonably expect any positive transfer to hold for other, less-similar variants.

2.1.2 Infinite Chess

Infinite Chess has the same set up as classic chess but is played on an unbounded two-dimensional square-grid board. Three new rules are introduced to balance the game:

1. Knights are replaced with a new piece called the 'nightrider', which moves like a knight but for any number of hops in the same direction. This brings the piece up to strength with other major pieces, namely the bishop and queen, on an infinite board.
2. Promotion occurs when a pawn moves to a rank after which there are no enemy pieces. This allows for promotion on the infinite board and adds a level of complexity as players must be careful to avoid promoting enemy pieces.
3. For each piece there must exist an 8x8 square which includes at least one enemy piece. This prevents the board from becoming infinitely sparse and introduces a new method of capture: *Capture by stranding*. After every move, any enemy piece that is stranded (is not near enough to a piece of the opposite colour) is captured and removed from play.

Despite being played on an infinite board, the search space of Infinite Chess is bounded by the fifty-move draw rule of classic chess: A game is automatically drawn if there has been no pawn move or no piece has been captured in the last fifty moves. Due to the complications of implementing Infinite Chess, the introduction of a new piece and new method of capture, it was decided that this chess variant was not suitable for transfer learning research.

2.1.3 Chessplus

Chessplus is a relatively new chess variant invented by Christian Simpson. With the same pieces, starting position and board as classic chess, it is very easy for a knowledgeable chess player to learn. This is an early sign that transfer learning is possible in this domain. The variant introduces just one rule: two pieces of the same colour can be combined by moving them onto the same square (Figure 2). The resulting piece can move as either constituent piece or can be split back into two pieces again by moving one half-piece while leaving the other.

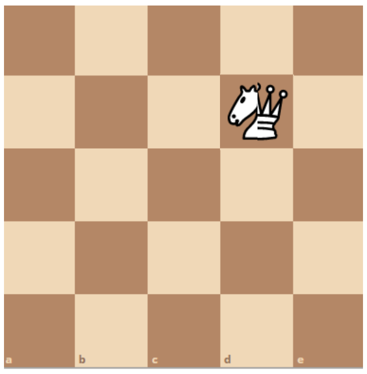


Figure 2: A Chessplus piece; the knight and queen are merged and may move together.

Chessplus is highly similar to classic chess and as such we anticipate significant overlap of strategies in the two games. Further, the number of legal games in Chessplus can be assumed to be several orders of magnitude higher than that of chess due to the additional moves introduced by merging pieces. This provides an opportunity to emphasis the time-saving value of transfer learning. Finally, as Chessplus is new and uncommon relative to other chess variants, there is little to no public data on the game. Consequently, traditional machine learning methods such as supervised and unsupervised learning become less viable and we further highlight the benefits of transfer learning.

For the reasons stated, we select Chessplus as the variant to focus on in our research. However, we expect results to generalize to other variants with similar properties, that is, highly overlapping strategies and domains.

2.2 Literature Review

This work succeeds decades of progress in such areas as game theory, machine learning, deep learning, mathematics and statistics. To fully understand the underlying concepts that support this research, we provide a review of the existing literature in those most closely related areas.

2.2.1 Game Theory

Game Theory is the area of research pertaining to the use of mathematical models in multi-agent decision-making processes [3]. Given a set of states S, a set of actions A and a transition function $f: S \times A \to S$, a complete game of $t+1$ moves is captured by a sequence of actions $(a_0, a_1, \dots, a_t)$ such that $f(s_0, a_0) = s_1$, $f(s_1, a_1) = s_2$, …, $f(s_{t-1}, a_{t-1}) = s_t$, for some initial state s_0 and some terminal state s_t [4].

A reward function $R: S_T \subseteq S \to \mathbb{R}^k$ allocates a reward to each of the k players for each terminal state $s_t \in S_T$. In many games, player k takes reward $r_k = -1, 0$ or $+1$ for a loss, draw, or win, respectively. The aim of each player is to maximise their own reward.

Further, games may be organised into classes by defining several properties:

- *Discrete:* Describes whether the game is turn-based, in which players $1, 2, \dots, k, 1, 2, \dots$ sequentially select an action to play, or continuous, where all players may select and apply actions simultaneously (in real-time).
- *Zero-sum:* A game is zero-sum if and only if, for each realised state s, $\sum_{i=0}^{k} R_i(s) = 0$. That is, the sum of the reward for all players is zero at every point in the game. This can indicate whether a game is cooperative, competitive, or somewhere in between.
- *Information:* Describes the visibility of the current state to each player. A game has *perfect information* if and only if all players can fully observe the current state, otherwise it has *imperfect information.*
- *Determinism:* Describes how factors of chance, such as rolling a die or drawing a card from a shuffled deck, impact a game. A game is *deterministic* if there is no chance factor, otherwise it is *non-deterministic.*

A special class of games, *combinatorial games,* are those games which are discrete, zero-sum, deterministic, have perfect information, and are played by two players. Chess and Chessplus are combinatorial games.

2.2.2 Game Representation

The data structure chosen to represent information in any given scenario is integral to the type and efficiency of algorithms applicable to that problem. This is no different in the domains of games. There are many data structures which may sufficiently capture games as they are defined in the previous section. The simplest data structure may be no more than an initial state s_0 and a sequence of actions $(a_0, a_1, \dots, a_t)$. However, traversing such a structure becomes difficult: without a 'reverse transition' function $g: S \times A \to S\,; g(s_i, a_{i-1}) = s_{i-1}$, one may only traverse forward through the action sequence. This greatly restricts the usefulness of any algorithm. Therefore, games are typically represented using tree structures, with nodes representing some $s \in S$ and edges representing some $a \in A$; the root node represents the initial state s_0, likewise leaf nodes are for each terminal state $s_t \in S_T$. A single game instance is given by a traversal of the game tree from the root to a leaf.

Many board games have a significant number of actions available to a player on each of their turns. Chess has on average 35 moves available to each player on each turn while 19x19 go has 250 [5, 6]. The correspondingly high branching factors of the representative trees therefore make it infeasible to generate an entire game tree in advance, thus a full search is not possible. Instead, most approaches iteratively expand the game tree from the initial node, moving only one or a few layers deeper into the tree to avoid expanding sections that will not be explored.

While the time complexity of game tree generation stems from the breadth and depth of a game's search tree, additional space complexity is introduced in the generation of new nodes at each depth and how each state is represented. During move generation, one may create new objects for each state reachable from the current state; this introduces unnecessary space complexity and also increases computation time as memory write operations must be performed frequently. More efficient implementations will introduce a stack to keep track of moves made from the initial state up to the current state, and push or pop actions from this stack, as necessary. However, in some situations, such as in Monte Carlo Tree Search (Section 2.2.4), multiple states must be stored simultaneously, increasing memory requirements.

2.2.3 Minimax Tree Search

With trees being a popular and effective way to represent games, tree traversal and search algorithms play an important role in the selection of moves. In smaller games such as tic-tac-toe or Connect Four, the game tree is small enough to be generated in its entirety and simple search methods such as breadth-first search or depth-first search can be applied in tandem with an evaluation function to find the best move on each turn. In more complex games where it is infeasible to check every possible state in an evaluation function, a more efficient approach is required.

Minimax is used in discrete zero-sum games to select the move at each turn which minimises the player's maximum loss. In combinatorial games, one player acts as the *maximising* player, the other the *minimising* player. At each turn, player one will select the move which leads to the state with the maximum value while player two will select the move which leads to the state with the minimum value. Figure 3 shows a game tree with minimax values assigned to each node. Each node's value is determined as follows: if it is player one's turn to move, the value of the current node is the maximum of the values of its children, while if it is player two's turn, the value is the minimum. If a state is

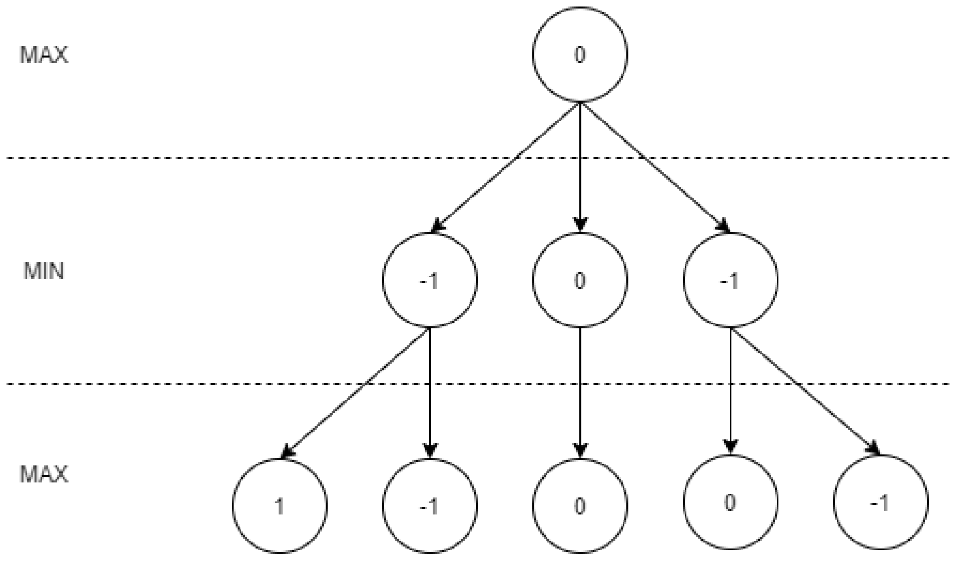


Figure 3: A simple minimax tree for two players.

terminal and thus has no children, it is given a relative value. This is usually $-1, 0$ or $+1$ for a minimising win, draw or maximising win, respectively.

Notably, the value of a state in plain minimax must take the value of a reachable terminal state. Therefore, basic minimax requires that the entire game tree be generated from the current state. As stated earlier, this is not feasible in any game of significant size. Furthermore, the algorithm searches the entire game tree from any given node in order to assign that node a value. In some cases, this may lead to redundant computation as certain paths cannot possibly be more beneficial to the moving player than previously seen paths. These points led to the introduction of *maximum search depth* and *alpha-beta pruning*.

To prevent extensive exploration of a game tree, minimax may be augmented with an evaluation function $E: S \to \mathbb{R}^k$ and maximum search depth d. E takes a game state and returns a tuple of k real values, which are the estimated rewards for each of the k players. Alternatively, in zero-sum games with $k = 2$, E may return a single real value to be minimised or maximised by the respective players, allowing for a simpler implementation known as *negamax* [7]. Minimax will now search up to d nodes ahead of the current node before applying the evaluation function, rather than continuing the search to a leaf node. Crucially, an evaluation function which poorly estimates the value of a node is likely to negatively impact the quality of results. A good evaluation function is one which closely approximates the true reward function R assuming optimal play by all players from the current node to a terminal node.

Minimax with maximum search depth may still perform unnecessary searches of branches that cannot possibly lead to beneficial outcomes. *Alpha-beta pruning* is an optimal algorithm which can potentially speed up the minimax algorithm, by pruning away paths that cannot possibly lead to an outcome more preferable than the current best outcome [8, 9]. Pruned paths do not need to be explored.

For any node in a combinatorial game tree, a value α tracks the minimum score guaranteed to the maximising player and a value β tracks the maximum score guaranteed to the minimising player. These values are initiated to $-\infty$ and ∞ respectively and are updated as paths from the current node are explored. If at any stage $\alpha > \beta$, the maximising player may confidently stop considering options down the current path, as these states will never be realized in optimal play. Figure 4 shows a case where alpha-beta pruning is applied:

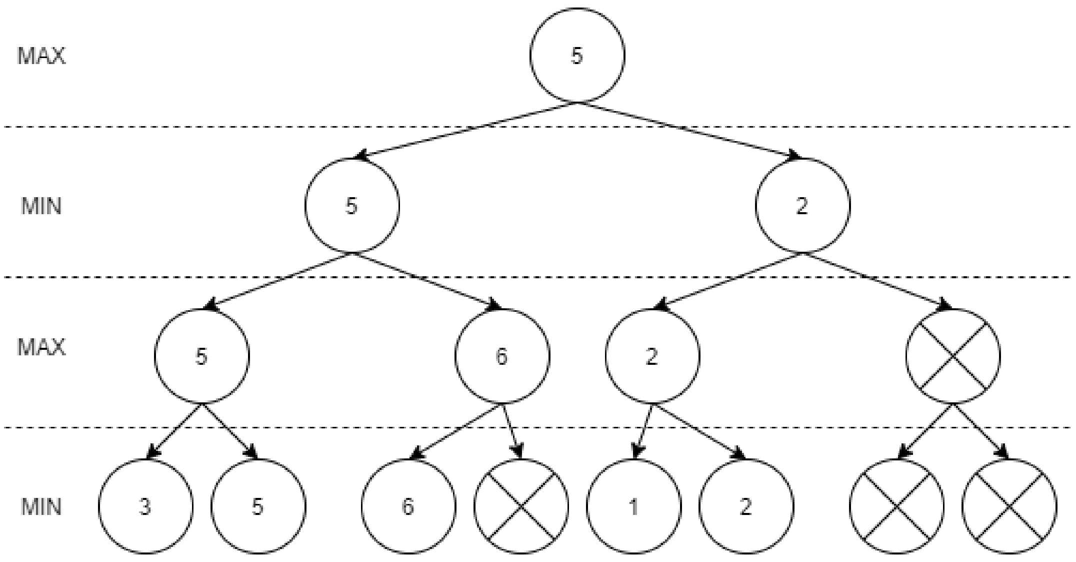


Figure 4: Alpha-Beta pruning applied to a simple tree. After exploring the left child of the right branch, it becomes redundant to continue the search.

Alpha-beta pruning is most effective when triggered earlier in the game tree, where the maximum number of paths are pruned. Therefore, alpha-beta pruning can be optimised by ordering the game tree such that moves more likely to be favourable are considered before moves more likely to be harmful. For example, in chess, capture move are generally more beneficial to the moving player that non-capture moves. A game tree with captured moves ordered before non-capture moves would thus be more likely to benefit from alpha-beta pruning than an equivalent game tree without this ordering.

2.2.4 Monte Carlo Tree Search

For games such as chess and shogi, the minimax algorithm with alpha-beta pruning and other extensions has been sufficient for the efficient selection of strong moves, as demonstrated by the defeat of then-world chess champion Garry Kasparov in 1997 by Deep Blue. However, this approach deteriorates in the much more complex game of go, which has a significantly higher average branching factor of 250 and in which a move cannot be said to be definitively good or bad for the moving player until much later in the game [5]. This makes a strong evaluation function, necessary for effective depth-limited minimax, difficult to define and led to the development of Monte Carlo Tree Search (MCTS) [1, 10]. MCTS is a combination of Monte Carlo simulation, outlined shortly, with typical tree search. As the number of iterations tends to infinity, MCTS converges to the minimax solution [11-14]. We can therefore conclude that MCTS is at least as good as minimax. In lieu of evaluation functions, Monte Carlo simulation uses *policies*, one for each player, which probabilistically map game states to moves. In a two-player game, the value of a game state s to a player can be estimated by applying each player's policy in move order until a terminal game state s_t is reached. This process can be repeated any number of times and the rewards given by the terminal states averaged to acquire an approximation of the value of the initial state s.

Starting from the initial game state s_0, MCTS iteratively grows the game tree through a sequence of four phases: *selection, rollout, back-propagation* and *expansion*. Each node in the game tree contains a game state as in minimax, but also contains a value which reflects the estimated probability of that node resulting in a win for the moving player, according to rollouts played in previous iterations of the algorithm.

During the *selection* phase, the algorithm starts at the current node s_i and descends to a leaf node s_l, at each ply selecting the action which leads to the child node s_c which maximises a value ρ:

$$\rho = \frac{W_c}{r_c} + \lambda \sqrt{\frac{\ln R}{r_c}} \tag{1}$$

…where W_c is the number of winning rollouts through child s_c, r_c is the total number of rollouts through child s_c, λ is an empirically-derived scalar factor, and R is the number of previous iterations in which the parent of s_c was selected. The first term is known as the *exploitation bonus* and is higher for children apparently more likely to lead to winning positions, which the second term is the *exploration bonus* and is higher for less-visited children. This term is necessary to avoid zero-probability nodes and to encourage exploration of low-valued nodes which may in fact lead to strong positions. λ may be adjusted to put more or less emphasis on the exploration of less-promising nodes. In Equation 1, the exploration bonus is the *Upper Confidence Bound Applied to Trees* (UCT), which is very commonly used in practice [12].

During the rollout phase, a fixed number H of rollouts are performed. Each rollout involves running a Monte Carlo simulation, as described above, until a terminal state is reached. During the *back-propagation* phase, the reward returned by each rollout is propagated backwards through the tree, updating the win and visit counts W and r of each node along the selection path. Finally, during the *expansion* phase, the current node s_i is expanded with child nodes for each of the states first generated in each rollout. These children are initiated with a value according to the result of their rollout. In practice, it is common to implement these phases in the order: selection, expansion, rollout, back-propagation. Figure 5 demonstrates each phase on a game tree with $H = 1$.

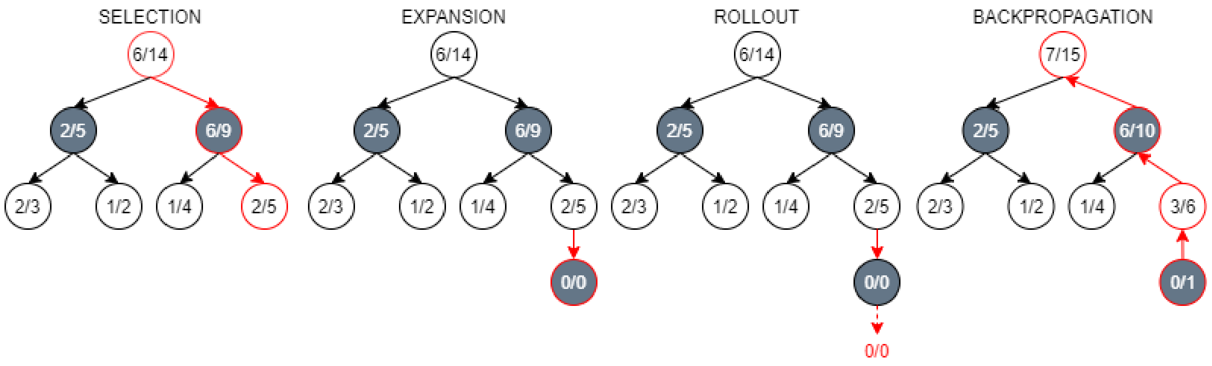


Figure 5: An example of Monte Carlo Tree Search. During expansion, a new child node is created from the selected node. A roll-out is played which results in a loss for the black player. The result is then back-propagated through the tree. In practice, these phases can be implemented in different orders.

Importantly, if during selection multiple children have the same maximum ρ, one is selected at random. This ensures non-deterministic selection outcomes. MCTS will balance the exploitation of nodes with high win rates and the exploration of nodes with few rollouts, building a highly asymmetric game tree with deeper branches exploring the most promising moves.

Due to its nature, MCTS improves its performance over time. This means performance is worst at the beginning of a game when few rollouts have been completed and moves are essentially random. It can take many iterations before the performance of MCTS reaches an acceptable level, and many more before it approaches optimal. In many combinatorial games, permutations of the same action sequence will result in identical positions. For example, in a chess game, moving a particular pawn and then a particular knight yields the same board position as first moving the knight and then the pawn. One could therefore argue that the order of these move is not relevant, only the moves themselves. This idea is presented in the *All Moves As First* (AMAF) enhancement to MCTS. AMAF is an extension to the rollout and back-propagation phases of core MCTS in which if any action playable from the node being rolled out is played at any stage of the rollout, the win and visit statistics of the corresponding child are updates according to the rollout reward [15].

AMAF operates as if any action made during the rollout phase was instead made during a previous iteration of the selection phase, and has been shown to improve the performance of MCTS for many games, but is less successful for others [4]. *Rapid Action Value Estimation* (RAVE) is a variation on AMAF in which each node s additionally stores rollout results from all rollouts in which the action that led to s is played [16, 17].

With modern hardware providing access to excessive memory resources for most applications, it is surprising to find that for larger games, Monte Carlo Tree Search may ultimately be bounded by memory rather than time. In particular, this occurs when the entire tree is kept intact in memory, where it is otherwise common to prune away upper parts of the tree as moves are made and a game progresses. Multiple modifications to the original algorithm have been proposed to address this memory management issue:

- *Stopping:* The simplest solution is to stop building the tree once the memory limit is reached and to retrieve results from the current tree. Any remaining available time is neglected.

- *Stunting:* A more efficient solution than simple stopping is to stop creating new nodes, but to continue updating those that are already in the tree. These additional iterations help to reinforce the win rates of each node.
- *Ensemble:* When the memory limit is reached, discard the entire tree while keeping the result. Restart the algorithm from a new tree, repeating some number of times. Finally, combine the results from each tree to decide on the final action [18]. We will see in section 2.2.5 that this is conceptually very similar to *root parallelization*; the key difference being that Ensemble MCTS is executed in sequence while root parallelization is performed concurrently.
- *Tree flattening:* Similar to Ensemble MCTS, tree flattening discards the current tree when the memory limit is reached. However, the root and first ply (children of the root) are retained, preserving some information to be reused in the next tree.
- *Node recycling:* A queue is introduced which tracks the least-recently accessed nodes in the tree. When a node is accessed by the selection policy it is removed from its current position and returned to the end of the queue. If the memory limit is reached, the front node in the queue is 'recycled,' removing it from the tree and allowing more promising branches to be expanded. This method discards the least relevant information, retaining as much as possible to inform continued searches [19].
- *Garbage collection:* When the memory limit is reached, discard all nodes that have not been visited at least k times, and continue the search as normal [20].

Of these, stopping and stunting are the simplest to implement, but provide the least benefit. Ensemble MCTS has been shown to outperform each in almost all instances, while tree flattening provides a more efficient option. Garbage collection is also simple to implement but may be accused of discarding information unnecessarily. Node recycling is the most efficient of these options with respect to reusing information but requires significant reworking of the base algorithm. However, it has been demonstrated to outperform other methods in highly memory-limited domains.

2.2.5 Parallel Monte Carlo Tree Search

Monte Carlo Tree Search is slow to converge for games of non-trivial size. There has been much research into reducing the time to convergence through distribution to a machine cluster and through a variety of parallelization methods. In particular, three primary approaches to parallelization have been developed: *leaf parallelization, root parallelization* and *tree parallelization [21, 22].*

In leaf parallelization, outlined in Figure 6, each of the H rollouts played from the selected node are run in parallel across independent threads. The rewards from all rollouts are collected together and back-propagated only once. Leaf parallelization is simple to implement, but often results in most threads idling for extended periods as some random rollouts are completed before others. Further, early rollouts may indicate that a particular node is not promising – but this information cannot be communicated to active threads, which continue regardless and waste valuable time.

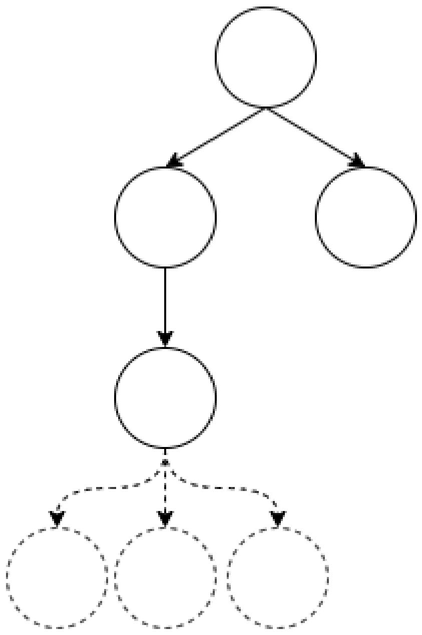


Figure 6: Leaf parallelization with H=3. Each rollout from the selected node is performed on its own thread.

Root parallelization (Figure 7), also named single-run parallelization in early research, is conceptually simple. With N threads, construct on each thread one Monte Carlo tree, with no communication between threads. The best move at each turn may be decided by one of two methods: *average selection,* in which a single tree is created through the aggregation of visit and win statistics for duplicate nodes and the best move decided by this merged tree, or *majority voting*, where each tree votes for the best move from its own perspective, and the move with the most votes selected as the best move [23]. Majority voting has been shown to improve performance in several machine learning and game-playing applications [24, 25]. One downside of root parallelization is the lack of communication between trees, as a promising action discovered by one tree cannot be exploited by any other tree. A possible solution

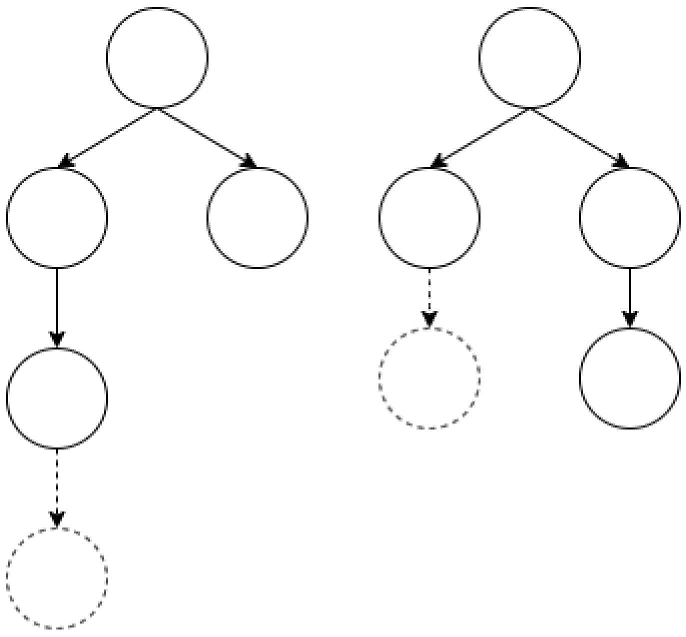


Figure 7: Root parallelization across two threads. Each thread constructs a tree completely independent of all other threads.

to this problem is to establish a master-slave relationship between threads, where each slave thread periodically reports updates to the master thread, which in turn communicates valuable information to all the slaves.

Tree parallelization (Figure 8) is the most complicated of the parallelization methods discussed. The approach uses multiple threads working on a single Monte Carlo tree for rapid construction, allowing each thread to take advantage of information discovered by other threads. One *global mutex* or multiple *local mutexes* are used to prevent simultaneous writes to the tree. Use of a global mutex majorly restricts the potential of tree parallelization as tree traversal is a significant time factor in most games. Therefore, local mutexes are preferred. A potential issue with tree parallelization is that multiple threads may select the same node during the selection phase of Monte Carlo Tree Search, or nodes that are nearby to one another. In large trees, it may be more beneficial to explore different neighbourhoods, rather than the same neighbourhood multiple times.

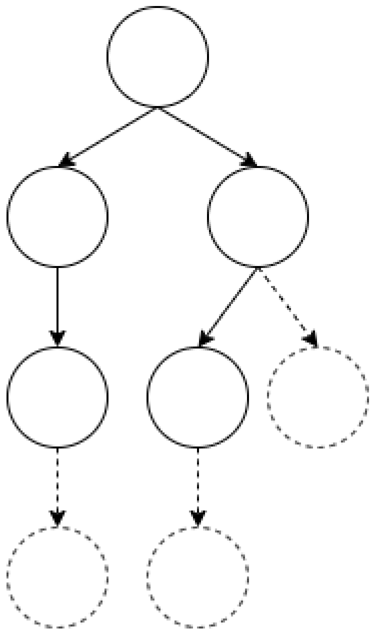


Figure 8: Tree parallelization with three threads operating on one tree.

Research has shown that leaf parallelization is the least useful parallelization method, likely due to the problem of idling threads [22]. However, subsequent research has resolved this issue by introducing a client-server parallelization model, which dispatches simulation jobs to a server cluster [26]. There are conflicting results on the performance of root parallelization against tree parallelization [22, 23]. One advantage of the former over the latter is that it is much simpler to implement.

Further to the aim of reducing computation times, the typical UCT formula for tree traversal (Equation 1), may be transformed into a depth-first algorithm to significantly reduce the proportion of time spent in traversing the tree [27]. This method exploits the fact that branches with high UCT scores are likely to be visited multiple times before a different branch becomes more favoured.

2.2.6 Deep Learning

Deep learning is a popular approach to solving otherwise difficult problems, similar to MCTS methods in that performance improves over time as prior experience influences future predictions. Deep learning techniques have great potential, having been used in combination with MCTS to defeat world champion go player Lee Sedol [1].

In complex board games, the performance of game-playing algorithms is highly sensitive to the evaluation function (when minimax is used) or policy (when MCTS is used). As the consequence or benefit of a move may not be apparent until much later in a game, it is difficult to approximate the optimal evaluation function or policy. Deep learning methods can be used to iteratively improve the accuracy of these heuristics [28, 29].

The core of deep learning is the *Artificial Neural Network* (ANN), modelled loosely on the human brain [30]. Though not directly related to our research, a basic understanding of neural networks will prove beneficial in understanding concepts soon discussed, and so here we introduce two types of ANN, the basic *Feed-Forward Neural Network* (FNN) and the more advanced *Recurrent Neural Network* (RNN).

An FNN is a directed acyclic graph with layers of nodes, called neurons, each of which possesses a vector of weights $\overline{w}$ [31]. The network consists of an *input layer* and an *output layer* with typically one or more *hidden layers* between. A fixed-length input vector $\bar{x}$, for example a representation of a game state, is fed to the neurons in the input layer. Each neuron calculates its own output y as a function of the weighted sum of its inputs plus some bias b:

$$y = \phi(b + \sum_i w_i x_i) \tag{2}$$

…where ϕ is an *activation function*. This output is fed as input to neurons in the next layer of the network. In this way, information propagates through the network until it reaches the final output layer. Each neuron in the output layer represents an outcome, for example legal moves from a given position, and the output of these final neurons is a probability, for example the probability that a move results in a winning game. The output with the highest probability is selected as the strongest action.

Of course, it is possible that the network will reach incorrect or sub-optimal conclusions. To combat this, output results are back-propagated through the network to update the weights and biases of each neuron:

$$\begin{gathered} w_i \leftarrow w_i + \alpha x_i (y - \hat{y}) \\ b \leftarrow b + \alpha (y - \hat{y}) \end{gathered} \tag{3}$$

…where α is the *learning rate*, y is the expected output and $\hat{y}$ is the prediction. With this update, neurons iteratively update their weights and biases to converge to the correct output, assuming the output can indeed be learned. This form of learning, known as *supervised learning*, is one of several learning methods discussed in the following section.

While being the most basic form of neural network, FNNs have seen much use and success in turn-based board games. Research has demonstrated their ability to play backgammon with a win rate of 40% against human experts [32]. Of particular note here is the stochastic nature of backgammon due to dice rolls. Despite this, the network was able to generalize the game and provide a reasonable challenge for expert players.

A significant limitation of basic FNNs is that the input vector $\bar{x}$ is fixed-length; every input into the FNN must have the same shape and size. Often, we would like to input an arbitrary-length vector, for example, a list of all prior moves of a chess game. A further downside of FNNs is that they do a poor job of modelling sequences because each output depends only on the current inputs. Recurrent Neural Networks (RNNs) are neural networks which exhibit sequential memory. That is, they remember prior inputs, allowing them to consider these when determining the next output.

Effectively, RNNs are FNNs with the addition of a looped connection from each node back into itself. With each timestep t, a node's current internal state $u(t)$ is fed back to the node as input, as shown in Figure 9. We can also see that RNNs can be unfolded in time into FNNs. Therefore, we can consider RNNs as FNNs with a large number of layers [29].

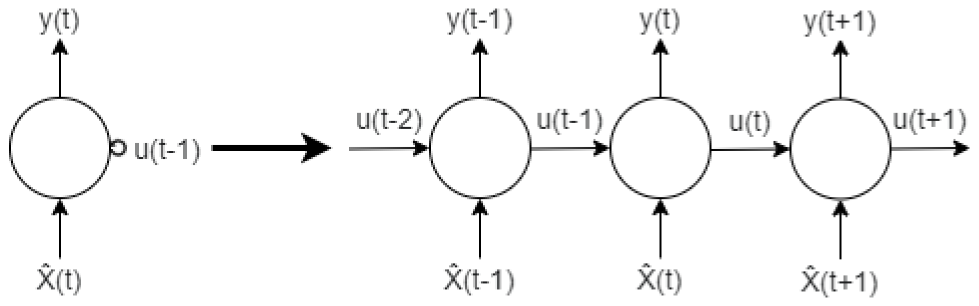


Figure 9: An RNN may be unfolded in time to an FNN.

One major flaw in RNNs is the vanishing gradient problem [33]. During back-propagation to update node weights, every node that was involved in calculating the output has its weight updated by a function of the error (Equation 3). In RNNs, this includes the weight of the recurrent connection from a node to itself. As the output of any node in an RNN is dependent on its previous outputs, back-propagation must occur through all previous time steps. This involves repeated multiplication of the propagated error by the weight of the recurrent connection. When the weight is small, the error received by each node at each timestep shrinks, such that the weights of the earliest connections in the network undergo negligible change. When the weight is large, a similar problem, called the *exploding gradient*, occurs. The consequence of the vanishing gradient problem is that nodes earlier in the network do not properly learn and their weights will remain relatively unchanged from their initial values. As later nodes depend on the outputs of these earlier nodes, the network as a whole will fail to learn correctly. Similarly, the exploding gradient problem results in networks whose weights grow too rapidly, such that they will never realise their optimal values.

The vanishing gradient problem can be seen as a network having a short-term memory, giving great value to the latest inputs and little value to earlier inputs. In games such as go, where moves early in the game have great implications on the final outcome, this is not desirable. To combat this, more advanced architectures such as *Long Short-Term Memory* and *Gated Recurrent Unit* can be considered [34, 35].

As the weights of connections in a neural network are initialized to random values, any network must undergo a *training phase* for its weights to approximate optimal values. This is followed by a *validation phase,* where the network is exposed to yet-unseen data. The purpose of the validation phase is to ensure that there is no overfitting, which can occur when the network becomes highly sensitive to random variations or noise in the training data. A sign of overfitting is when the network performs very well on the training data, but poorly on the validation data.

There are three popular methods for training a network: supervised learning, unsupervised learning and reinforcement learning [36]. In supervised learning, the network is given a set of example inputs are corresponding outputs. The network's objective is to find a function which maps given inputs to expected outputs, and which can be used to map new inputs to correct outputs. If outputs are discrete, this is a classification problem. Otherwise, this is known as regression.

In unsupervised learning, training example inputs have no corresponding known output. While there are many instances where unsupervised learning is applied, board game networks do not typically employ this method, and as such it will not be discussed further here.

Finally, reinforcement learning is similar to supervised learning, but expected outputs are replaced with a *reward function* [37]. The function maps actions to expected rewards, which are higher for more accurate predictions of the correct output. By maximising the total expected reward, a network can improve its estimations. Reinforcement learning is especially useful in scenarios where 'good' data is scarce, or when it is infeasible to label such data. As an example, while there is ample publicly-available data of expert go games, not only is it infeasible for a human to manually assign expected outputs, it is also unlikely that even an expert would correctly assign the truly optimal output for each game state. In this scenario, reinforcement learning allows us to train a network without labelling the data. Reinforcement learning is a highly favoured learning method for board games. Notably, they were applied to the superhuman go computer, AlphaGo Zero [38]. While reinforcement learning methods can be successful, they can also lead to failure. In particular, the effectiveness of reinforcement learning algorithms is highly dependent on the reward function used by the network [39].

2.2.7 Transfer Learning

In machine learning, *transfer learning* is the application of a pre-trained model from one *source* domain to another closely related *target* domain. This is usually followed by some finetuning in the target domain if some data is available [40]. In almost all instances where transfer learning is used, it is in the context of neural networks, particularly when there is insufficient data available in the target domain – without enough relevant data, a good network cannot be trained. One of the most common scenarios is in image recognition, where publicly-available networks, trained on very large datasets, are used to identify objects in images that they have not necessarily been trained to recognize. In another scenario, there is plentiful public data of classic chess games which could be used to train a neural network to play chess, but comparatively little data exists for chess variants, most notably Chessplus. Taking note that the aim of each variant is usually similar to classic chess – check the enemy king; capture enemy pieces, etc. – a network may be taught to play a chess variant relatively well by training it on classic chess data. Transfer learning has also been used in domains where there is enough data to properly train a network; this is done to save computation time in training a network from scratch.

Transfer learning in neural networks is achieved by replacing the final output layer of a pre-trained network from the source domain with a new output layer in which the weights are initialized to random values. The resulting network is fine-tuned by repeating the training process in the target domain. Figure 10 outlines the general idea.

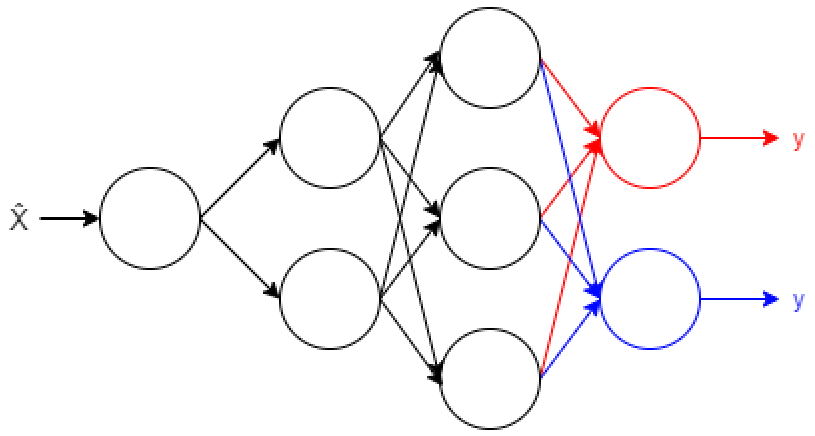


Figure 10: Transfer learning applied to a network with two hidden layers. The final output layer of the pre-trained network (red) is removed and replaced with a new output layer (blue), with random initial weights.

Fine-tuning is the process of adjusting the transferred model to better suit the target domain. Typically, the earlier layers of a neural network capture more generic features, while the later layers are more specific. Therefore, it is of greater importance to retrain the later layers of the transferred network; with limited data, it may not be effective or efficient to retrain every layer. Early layers may be *frozen*, that is, their weights will not be updated during back-propagation, while the later layers are fine-tuned. However, with sufficient relevant data, it can become feasible to fine-tune even the earliest layers of a transferred network.

There have been many applications of transfer learning to playing a variety of games. Asawa, Elamri & Pan have used transfer learning in the creation of a network that plays the OpenAI game Snake after learning how to play PuckWorld, while Mittel, Munukutla & Yadav have demonstrated the use of transfer learning to transfer game knowledge between the Atari games Pong and Breakout [41, 42]. Within the domain of board games, transfer learning may be used to transfer knowledge from smaller game boards to larger ones. For example, the board game *go* may be played on a variety of board sizes. Training time is shortened as a consequence of the reduced search space of smaller boards, while general strategies may be transferred to larger boards.

Transfer learning is most effective when the target domain is very closely related to the source domain. A good analogy for this is musical instruments: the viola, violin, and cello have much in common and one may reasonably expect a musician who can already play one of these instruments to learn more easily to play another, compared to someone who cannot play any. Likewise, chess and Chessplus have strong similarity in their rules and strategies. Such cases where transferring learning is beneficial is known as *positive transfer.* However, there are also situations where transfer learning is overall detrimental to the learning process: when making predictions, the resulting model is comparatively worse than an equivalent model that did not begin with the source data. This is *negative transfer*.

More formally, we may measure the quality of a transferred model by comparing its expected loss with the expected loss of the same model trained without the source data [43]:

$$J\big(A(S,T)\big) - J(A(\emptyset,T)) \tag{4}$$

where J is the loss function, A the transfer algorithm used, which takes a source domain and a target domain and returns a model, and S and T the source data and target data, respectively. This is the *negative transfer gap*, which also gives rise to the *negative transfer condition*:

$$J\big(A(S,T)\big) > J(A(\emptyset,T)) \tag{5}$$

which allows us to say that negative transfer has occurred if the expected loss of the transferred model is greater than the expected loss of the same model without transfer. Positive transfer has occurred if the inverse is true.

Critically, although in most situations transfer learning is applied to neural networks, the definition does not preclude transfer in other models. Any scenario in which a model benefits from reusing knowledge learned by another model may fairly be called transfer learning. We will later see that Monte Carlo Search Trees can also benefit from transfer learning.

3. Representing Games

Before research into the effects of transfer learning can begin in earnest, we must first decide on representations suitable for chess and Chessplus. Indeed, we will see that one representation is sufficient for both games. This chapter discusses the design and reasoning behind our chosen representation, and looks at the implementation and validation of move generators. We seek here to address the first of our research aims: to explore various representations and identify from these one most suitable for transfer learning.

3.1 Design & Implementation

We begin our research by designing and implementing a single representation capable of capturing both classic chess and Chessplus games. We first look at classic chess and then expand this to Chessplus. There are several properties which together can unambiguously define the current state of a chess game:

- *Board:* The current board, including information about which pieces occupy which squares. For an 8x8 board, this is represented as a 64-element array, with integers uniquely defining the twelve different piece type/colour combinations and an additional integer for the empty square.
- *Castling rights:* Information regarding which players have the right to castle and in which direction ('kingside' and/or 'queenside'). This is a four-element tuple of Boolean flags.
- *Halfmove clock:* In chess, a 'fullmove' consists of one move by white and a response move by black. A 'halfmove,' then, is a single move by either player. A less-known rule of chess states: "The game is drawn… if the last 50 moves by each player have been completed without the movement of any pawn and without any capture" (FIDE). The halfmove clock counts the number of halfmoves since the last pawn move or capture, and the game is automatically drawn if this count reaches 100.
- *En passant square:* A pawn that has moved two squares from its home rank may be captured on the immediately following move by an enemy pawn as though it had moved forward only one square. This is known as 'capture en passant,', or 'capture in passing.' The en passant square identifies the board index that the capturing pawn should move to, or is a null reference when no en passant capture is available.
- *Player to move*: A simple binary flag indicating the player to make the next move.

A chess state may therefore be reconstructed from a data structure containing these five pieces of information. In practice, we use a Python class with these properties as instance variables. For generating moves and playing games, we also introduce some functions:

$$next: S \rightarrow \wp(A) \tag{6}$$
$$move: S \times A \rightarrow S \tag{7}$$
$$terminal: S \rightarrow \{0, 1\} \tag{8}$$
$$reward: S_T \rightarrow \{-1, 0, 1\} \tag{9}$$

…where $next$ takes a state s and returns a subset of A. Specifically, $next$ returns exactly those moves $\{a_1, a_2, \dots, a_n\} \subseteq A$ which are playable from the given state. $move$ is the state transition function described in game theory which takes a state s_i and an action a, and returns the new state s_{i+1} which results from applying action a to state s_i. $terminal$ takes a state and returns 0 or 1, indicating whether or not that state is terminal. Finally, $reward$ takes a terminal state and returns an integer: -1 if black has won, 0 for a draw or stalemate, and 1 if white has won.

Three considerations must be made when extending this representation to include Chessplus. Firstly, as one representation is used for both game types, we introduce a Boolean flag *chessplus* to the state representation, which is True if and only if the state belongs to a Chessplus game. Secondly, as each board square may contain up to two merged pieces and there are 25 possible combined piece plus a king for each player (a total of 62 piece types including singular pieces), it becomes impractical to continue representing pieces as integers. Thirdly, $next$ must be updated to also generate legal Chessplus moves.

To address the second of these issues, we implement a Python class 'Piece'. The game board is now an array of Pieces, or null references for empty squares. To distinguish between Chessplus pieces, which may be merged or singular, and classic chess pieces, which are always singular, we will henceforth refer to Chessplus Pieces as 'Pieces' and singular pieces as 'half-pieces'. A Piece therefore contains either one or two half-pieces. Each Piece has a colour, left value, and right value, representing the player colour, first half-piece and second half-piece, respectively. An uncombined Piece will have either the left value or right value as a null reference.

Finally, we expand the $next$ function to also generate those moves that are legal in Chessplus but not in chess. This includes merging two half-pieces, or separating them, or moving a Piece containing two half-pieces. Importantly, these moves are only included in the result if the *chessplus* flag of the given state is True. This allows us to use the same $next$ function to generate moves for both classic chess and Chessplus.

3.2 Validation

It is of course essential that the chess and Chessplus move generators we have implemented are without error. It is very easy to miss hidden edge cases, in particular regarding promotion moves, castling, checks and en passant. We therefore perform extensive testing and validation of our representation and move generators.

$perft$ ('Performance Test') is a function commonly used to debug chess engines. It is defined as:

$$perft_s(d) = \sum_{s' \in S'} perft_{s'}(d-1) \qquad (10)$$

$$perft_s(0) = 1$$

…where S' is the set of states reachable from some given state s: $S' = \{move(s,a): a \in next(s)\}$. Put simply, the function counts the number of nodes in the generated game tree which are reachable in exactly d moves from some starting node s. Results are compared to other chess engines, which are known to be correct, to identify any errors.

We use $perft$ on a number of starting states crafted specifically to find different common errors in move generators, and at different depths. Referencing results from the roce chess engine[1], we found several edge case errors in our move generator.

Importantly, while $perft$ can assert the presence or absence of bugs in a move generator, it cannot identify what those bugs are. $divide$ is another debugging function which helps with this:

$$divide_s(d) = \{\big(a, perft_{move(s,a)}(d-1)\big): a \in next(s)\} \qquad (11)$$

[1]http://www.rocechess.ch/rocee.html

That is, $divide$ returns a set of (action, value) pairs, showing the $perft$ values for each action playable from the given starting state. This allows us to identify which branch of the game tree contains the error, and following this branch will eventually reveal the bug. As an example, Figure 11 shows that our move generator was incorrectly allowing pawns in file A to capture enemy pawns in file H en passant, by 'wrapping over' to the other side of the board, and similarly for pawns in file H capturing pawns in file A. We also found that knights were able to reach the opposite file in the same fashion.

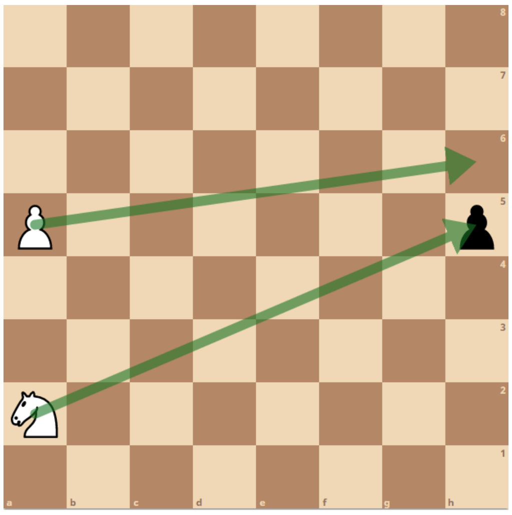


Figure 11: An example set-up demonstrating two bugs discovered during testing.

Unfortunately, there are no public Chessplus engines which may be used as a reference for $perft$ testing in the Chessplus domain. Indeed, if there were, it would not be necessary for us to implement our own! This does mean that we cannot properly test our Chessplus move generator. However, with confidence that classic chess move generation is operating correctly, there are only four important cases to consider: Merging pieces, separating pieces, promoting merged pieces, and capturing merged pieces en passant. By hand-crafting our own test cases to cover these scenarios, we identified the final error in our engine: if a merged piece containing a pawn reached the opposite end of the board via a non-pawn move, it would not be promoted. The error was amended, leaving us with what we believe to be a functioning Chessplus move generator.

3.3 Gardner Chess

In addition to asserting the correctness of move generation, $perft$ is commonly used to measure the relative computational efficiency of different chess engines by comparing the times taken to generate results. During $perft$ testing of our own engine, it became apparent that its performance was significantly lower than the reference engine and others, most likely as a result of the power of C programs over our own Python implementation. This had major implications on the feasibility of constructing adequate models for chess and Chessplus in a timely manner. At this stage, we decided to constrain our research to a 5x5 board. In particular, we settled on Gardner chess, the starting position for which involves removing all pieces to the right of each king and moving the remaining pieces onto the smaller board. The domain of chess on this board is orders of magnitude smaller than that of chess on the 8x8 board: the larger has $\leq 5 \times 10^{52}$ unique legal positions, while the smaller has $\approx 9 \times 10^{18}$ [5, 44]. The rules of Gardner chess are identical to those of classic chess, the only difference being the smaller board. Similarly, we play Gardner Chessplus as we would regular Chessplus.

An important consideration here is that the aim of our research does not change. To begin, we sought to investigate the effect of transfer learning in discrete zero sum games. Gardner chess and Gardner Chessplus are discrete zero sum games. However, one may fairly question the applicability of any results garnered from experiments on such a small, specific pair of games. The nature of our method of transfer is such that we may expect similar results on larger boards (i.e. the original 8x8 boards). Further, the fact that the domain of one game is a subset of the other means that we can again expect similar results for other game pairs with this property.

4. Monte Carlo Tree Search

Despite the significantly smaller search spaces of chess and Chessplus on a 5x5 board compared to an 8x8 board, it remains impractical to generate the full game tree of either: with an efficient engine generating one million moves per second, it would take over four thousand years to generate the full game tree for Gardner chess. Gardner Chessplus can only take longer. Therefore, we must carefully explore only those parts of the game tree which would be considered 'good' games, and avoid wasting time and other resources on branches which would not be played in practice. Monte Carlo Tree Search is an asymmetric tree search algorithm which focuses heavily on 'promising' branches of the game tree according to the results of past simulations. This is perfect for our needs. We therefore look to implement a version of Monte Carlo Tree Search which is compatible with our chess and Chessplus engines.

4.1 Limitations of Basic Implementation

We implemented Monte Carlo Tree Search in Python as it is described in section 2.2.4. During the selection phase, we select at each level the child node with the highest ρ-value (Equation 1) until we reach a node which is not fully expanded (it has children that are not yet in the tree) or is terminal. In the former case, we move to the expansion phase. In the latter, we move straight to simulation. When calculating the ρ-values of nodes, we use $\lambda = \sqrt{2}$, giving only minor consideration to less frequently explored branches.

During expansion, we simply iterate over all children of the selected node and add to the tree a random child that is missing. For simulating games, we use a fully random rollout policy for both players. While this is the least effective policy one could use, there are several factors that prevent us from using another: firstly, we could have used a heuristic-based policy to estimate the value of each child. For example, we could have assigned a value to each piece type and for each move in a rollout selected the child that minimises the sum of the values of enemy pieces, or maximises the value of the current player's. There are many such heuristics used in chess. However, we cannot definitively say that these heuristics intended for classic chess would also be suitable for Chessplus. A poorly chosen heuristic could lead to negative transfer. Secondly, we could have followed current literature and used neural networks to find a rollout policy. This presents the same issues; we would need to find different policies for each game. We would then not know if any transfer learning that occurred was as a result of our method or simply due to the different policies.

Finally, during back-propagation we update the values of all nodes from the simulated node to the root. At each step, we increment the visit count of every node and the win count of nodes belonging to the player that won the rollout. If the rollout was a draw, we increment the win count by 0.2. This is lower than the typical 0.5 used for draws as we would like to emphases the search for wins, not draws.

Initial attempts to construct models for chess and Chessplus using this basic implementation of Monte Carlo Tree Search were unsuccessful. While the algorithm worked as expected, we found that the tree grew at an alarmingly slow rate and that memory limits were reached before the tree could begin to find strong moves. We take two paths to solving these issues: firstly, we move from a single machine to a cluster of machines. This helps to reduce the time required to find promising moves by dividing the workload. Secondly, we introduce a novel algorithm for pruning Monte Carlo trees. This manages memory usage in a manner that allows trees to continue building promising branches by disposing of less promising nodes.

4.2 Distributed Monte Carlo Tree Search with Pruning and Merging

To address time issues encountered when using a single machine, we build Monte Carlo trees on a cluster of thirty. We use root parallelization with average selection to find the best move at each ply.

To maximise the depth of our Monte Carlo trees, we introduce a novel pruning algorithm, which we call *dropping*. When the memory limit is reached, the tree selects the best child of the current *pseudoroot* and prunes away all other branches. The pseudoroot is initially the same node as the root; after every prune, the selected best child becomes the new pseudoroot. Iterations of the tree-building algorithm are always started from the pseudoroot. In this manner, we prevent the re-expansion of nodes that have been pruned in the past. Figure 12 demonstrates the principle of the pruning algorithm.

Dropping offers a significant advantage over other pruning algorithms. When the memory limit is reached, we maximise the memory freed by pruning away all but the most promising branch. This allows the most possible further exploration of this branch compared to other methods. As a cost, however, dropping discards more information about the game tree than any other pruning algorithm, short of restarting the search. In situations where having a deep game tree is more important than having a broad game tree, dropping is beneficial. Otherwise, pruning algorithms which preserve more branches will likely lead to better results. In our research, we seek to compare the relatively playing strengths of a chess model transferred to the Chessplus domain and a Chessplus model *over full games*. Therefore, tree depth is more important in our context, and dropping is an effective pruning algorithm. However, a game tree which is insufficiently broad will have very few distinct games played to completion (terminal leaf nodes). We therefore preserve information in pruned branches by serializing those branches and saving them to permanent disk storage. We later use a simple merging algorithm to reconstruct complete Monte Carlo trees from their various pruned branches, by reconnecting those branches to the root of the surviving tree.

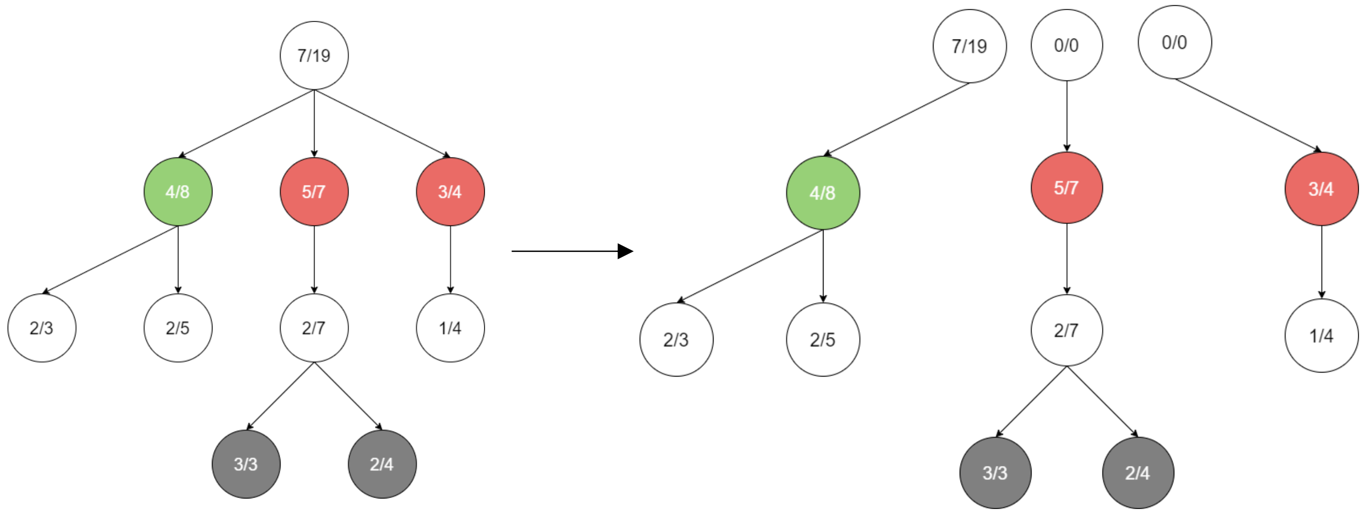


Figure 12: An example of dropping. On the left is the complete tree prior to pruning. The best child is shown in green, others are red. On the right is the result of pruning. The search continues from the best (green) child, while other branches (red) are removed.

5. Model Creation and Applied Transfer Learning

Until this point, effort has been focused on the implementations necessary for applied transfer learning. Chapter 5 focuses on the experimental process used to investigate the effect of transfer learning in Monte Carlo Tree Search, and addresses the second and third research aims highlighted in section 1.1: the construction of two knowledge models, one each in the domains of classic chess and Chessplus, and transfer between these models.

5.1 Constructing Monte Carlo Trees

With a functional game representation, move generators and Monte Carlo Tree Search, we may finally construct models for chess and Chessplus. We begin by building a Monte Carlo tree for chess.

We used a cluster of thirty machines to construct the Monte Carlo trees. Each machine was equipped with 32 GB of RAM and an Intel Core i7-8700 CPU @ 3.20 GHz, and ran the Linux Mint 19.3 Operating System. Using root parallelization, each machine independently constructed a Monte Carlo tree. To avoid differences in final tree sizes due to the random rollout policy used, we did not use a time bound. Instead, each machine was allowed to complete 10,000 iterations of the Monte Carlo Tree Search algorithm. When a machine reached its memory limit, we used dropping to prune away all but the most promising branch. Pruned branches were serialized and written to disk for later merging with the final tree. One may question how merging of these pruned branches is possible; if they were pruned due to a memory limit, merging them back together would surely exceed that limit. In practice, the machines used to build the Monte Carlo trees were not exclusive to our use. Other processes run by other users contributed to memory consumption. Where possible, tree building was done overnight to minimise this issue, but the problem persisted regardless. Further, interactions between the Python memory manager and the Operating System may mean that the Python processes constructing the trees reach a memory limit while there is still technically memory available on the system.

After each machine finished 10,000 algorithm iterations, we began the average selection process by combining results. All trees from all machines, including pruned branches, were merged into one aggregate tree. To achieve this, we used a dictionary which mapped a chess state to a single node. For each node in each tree to be merged, if that node's state is already in the dictionary, we merge the win count, visit count, parent information and child information of the node with that of the existing node. Otherwise, we simply add an entry in the dictionary mapping the new node's state to the new node. Importantly, as one chess state may be reached in many ways, it could be the case that nodes are merged which do not come from the same part of the game tree. See that this is not a problem: these nodes represent the same state, and from the perspective of *continuing* games from such a state, the way in which that state was reached (the move history) is irrelevant. The final merged Monte Carlo tree for chess had 195,884 nodes, with an average branching factor of 3.77. The maximum depth was 89. This process was repeated to construct a Monte Carlo tree for Chessplus. The resulting merged tree had 241,097 nodes with an average branching factor of 4.53.

5.2 Transfer Learning in Monte Carlo Trees

As mentioned previously, transfer learning is usually done in the context of neural networks. It is interesting, then, to see that transfer learning is also possible in Monte Carlo trees. In particular, transfer between trees is possible when their search spaces overlap (that is, when they have nodes or states common to both domains). In our research, every chess game is also a valid Chessplus game.

Transfer learning is thus possible by using knowledge from the Monte Carlo tree built for chess as a baseline for playing Chessplus.

Section 2.2.7 discusses how transfer learning usually involves some form of fine-tuning to improve performance in the target domain. In Monte Carlo trees, we consider fine-tuning to be the continuation of Monte Carlo Tree Search in the target domain, starting from a Monte Carlo tree constructed in the source domain. The utilization of source knowledge in the target domain is restricted by the relationship between the two. In situations where domains are disjoint, source knowledge cannot be used, and in fact transfer learning is fundamentally not possible. In situations where domains partially overlap, some pruning of the source tree may be required. In particular, one must prune away nodes and branches which capture states that are invalid in the target domain. The final possibility, and the one realized in our research, is the situation in which one domain completely overlaps the other – Chessplus completely overlaps chess. In this case, no initial pruning is required as every state in the source domain is valid in the target domain. Figure 13 visualises these ideas.

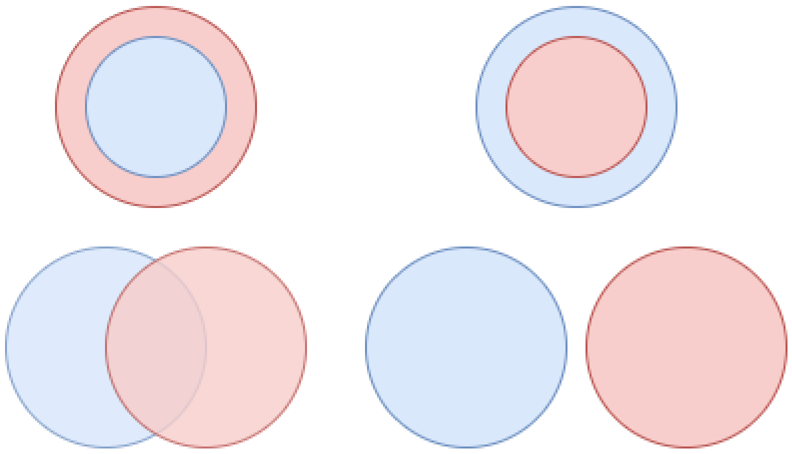


Figure 13: Possible configurations of the source (blue) and target (red) domains. (top left) The source domain is a subdomain of the target domain. This is the case in our research. In this configuration, no pruning is required after transfer. (top right) The source domain is a superdomain of the target domain. Significant pruning will be required after transfer as many states in the source domain are no longer valid. (bottom left) The domains partially overlap. Some pruning is required. (bottom right). The domains are disjoint. Transfer learning is not possible.

We transfer the Monte Carlo tree built for chess, the *source tree*, to the domain of Chessplus. Due to our representation, given in section 3.1, this is as simple as performing a traversal of the tree, at each node switching the *chessplus* flag to 'True'.

To fine-tune the transferred tree to the Chessplus domain, we continue Monte Carlo Tree Search for a further 10,000 iterations. As we are now in the Chessplus domain, the tree begins to explore those branches of the tree which were previously not available. Due to the nature of Monte Carlo Tree Search, these branches are explored with much preference over existing branches, at least until it becomes apparent that they are not promising (or that they are, in which case they continue to be exploited).

6. Comparisons and Results

We performed an experiment to measure the degree and quality of transfer. The transferred tree, which was allowed 10,000 iterations in the source domain (chess) plus 10,000 further iterations in the target domain (Chessplus) played 1000 games of Chessplus against the Monte Carlo tree which was constructed through 10,000 iterations in the Chessplus domain only. For convenience, we henceforth refer to this second tree as the *Chessplus tree*.

In each game, the starting player was decided randomly. The transferred tree and Chessplus tree took turns to select a move to be played; each tree could use only the information it had learned itself. Commonly, games would play out in such a way that one or both trees reached the end of the active branch before the game was completed. We continue such games until a terminal state is reached, switching to an 'online' tree search and giving an allowance of 200 Monte Carlo Tree Search iterations to each tree on each of their turns. From this point onwards, we expect each player to play with equal strength, though the current state of the game at the point we switch to online play should favour the stronger player, and this is expected to carry through to the end of the game. If one player reaches the end of the active branch in their tree while the other can continue, only the former switches to online play. The latter continues to select moves from their tree until either the game is terminal or they too reach the end of the active branch.

6.1 Statistical Validation

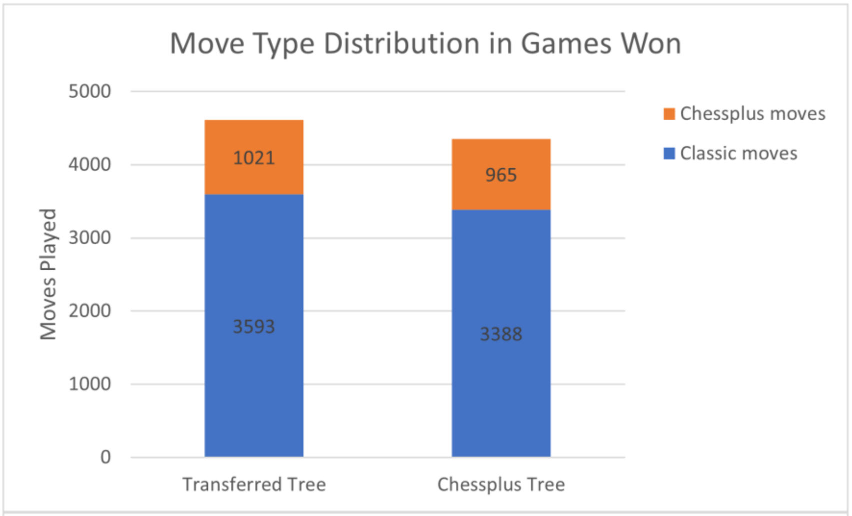
Of the 1000 games played, 474 (47.4%) were won by the transferred tree, 357 (35.7%) by the Chessplus tree, and the remaining 169 (16.9%) drawn. Intuitively, this indicates some degree of positive transfer as the transferred tree, with additional knowledge from the classic chess domain, has won 117 more games than the Chessplus tree, a significant number. We make the claim that the experiment resulted in positive transfer: $J\big(A(S,T)\big) < J(A(\emptyset,T))$ (see Equations 4, 5). To support this claim, we conduct a statistical analysis of the results. A binomial test was performed to determine the probability of the observed results given the null hypothesis being that transfer had no effect, i.e. each player is of equal strength. Discounting draws, the null hypothesis assumes a win probability for each player: $p(\text{transferred tree wins}) = p(\text{chessplus tree wins}) = 0.5$. With 831 non-drawn games, we would expect $831 \times 0.5 = 415.5$ wins for the transferred tree. The observation of 474 wins is higher than this expected value. Assuming the null hypothesis is true, the probability of observing results at least as extreme as this is:

$$p(\text{at least 474 wins for the transferred tree}) = \sum_{i=474}^{831} \binom{831}{i} 0.5^{i}(1-0.5)^{831-i} = 0.0000279$$

This is much smaller than the standard significance level of 5%, which allows us to confidently reject the null hypothesis. This reinforces the claim that positive transfer has occurred.

6.2 Further Analysis

We now move to a more qualitative and empirical inspection of the results. We focus on interesting aspects of gameplay, including the number of Chessplus-exclusive moves made by each player in games they won and lost, how many moves in each game were sampled from the offline trees versus online generation, and other interesting outcomes. Figure 14 shows the first of these:



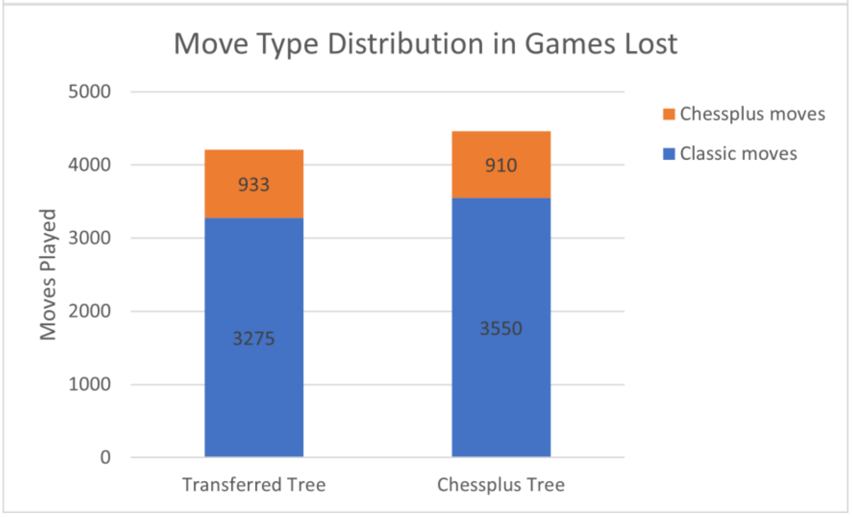


Figure 14: A chart showing the distribution of classic and Chessplus moves in games won (top) and lost (bottom) for each tree.

Across all games won by each respective tree, 4614 moves were played by the transferred tree, while 4353 were played by the Chessplus tree. Of these, 1021 (22.13%) and 965 (22.17%) respectively were so-called *Chessplus-exclusive* moves: merging pieces, separating pieces or moving merged pieces. In contrast, across games lost by each tree, 4208 moves were played by the transferred tree and 4460 by the Chessplus tree. Of these, 933 (22.17%) and 910 (20.4%) were Chessplus-exclusive. The proportion of Chessplus-exclusive moves made by each tree in both games won and games lost is highly similar; we cannot make any inference about the relative strength of these moves compared to classic chess moves.

During gameplay, we allowed each player to switch to online Monte Carlo Tree Search in the event that they could no longer select moves from their respective pre-generated trees. Figure 15 shows the proportion of moves that were pre-generated and those that were generated online, for ten games:

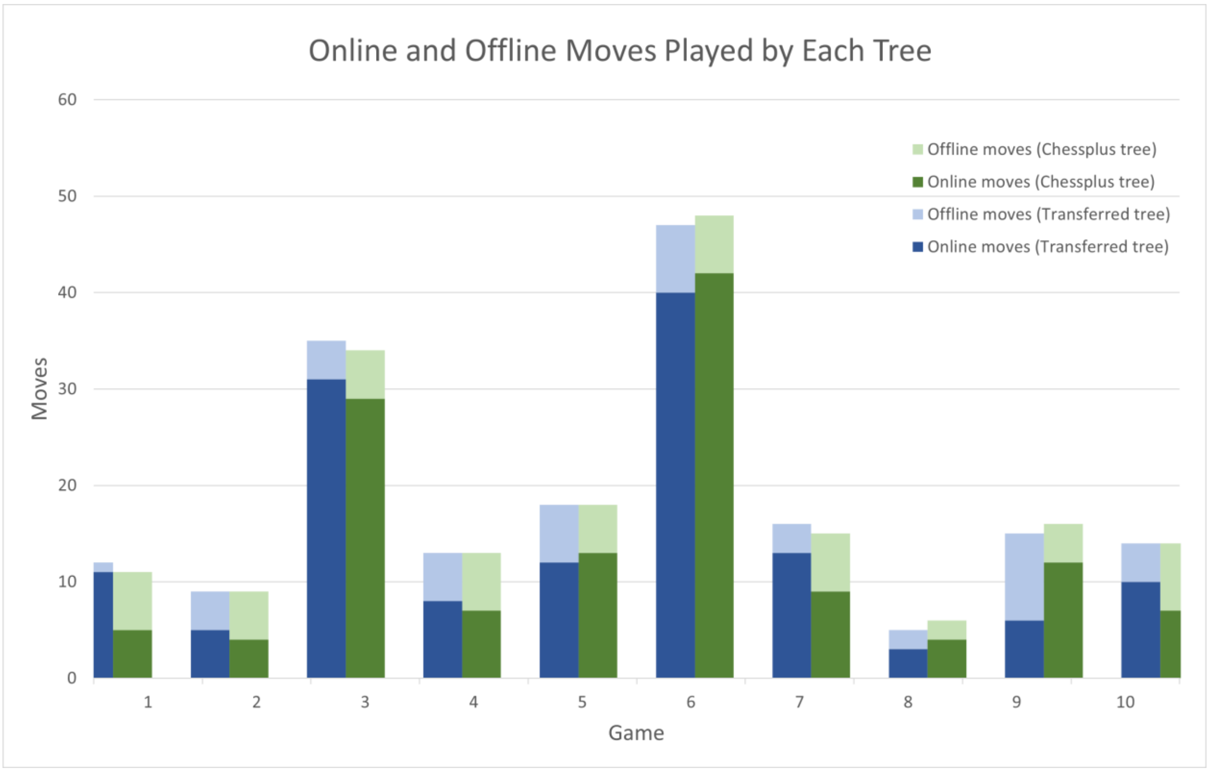


Figure 15: The proportion of moves that were sampled online and offline in ten simulated games.

Of great interest here is that for both players, the majority of games appear to consist mostly of moves generated online – not moves that were explored in the original trees. This is to be expected, as it is unlikely both trees explored the same neighbourhoods during offline Monte Carlo Tree Search. Combined with probabilistic sampling of moves from each tree, it is highly unlikely that a deep branch, common to both trees, would actualize. However, as online play, which constitutes the majority of moves, is equal in that each player is given 200 iterations per move, the observation that the transferred tree won more games would imply that those earlier moves selected from the offline tree had a significant effect on the final outcome of each game.

We also look at some statistics regarding the length of games played: the mean game length was 40 halfmoves; one could not say for certain what the expected value would be, though we point out that the average length of a classic chess game on an 8x8 board is approximately 80 halfmoves[1]. The observed result, for Chessplus played on a 5x5 board, therefore seems reasonable. The median game length was 30 halfmoves. This differs significantly from the mean, which indicates the presence of some outliers. Indeed, the shortest game consisted of only four halfmoves, while the longest was 152.

For interested readers, we provide in Appendix A a sample game played between the two trees.

[1]Based on data retrieved from https://www.chessgames.com/chessstats.html on 12/10/2020

7. Conclusion

This work has been built on a broad foundation of mathematics, statistics, game theory and machine learning. We present valuable contributions to the machine learning community with our novel pruning algorithm for Monte Carlo trees, *dropping*, and have explored new territories in applying transfer learning to Monte Carlo Tree Search. We have shown that the proposed architecture allows for positive transfer across Monte Carlo trees.

7.1 Contributions and Observations

Perhaps the most pivotal component of this work was to design and implement a game representation capable of capturing both classic chess and Chessplus, and to create a move generator for both. Without this work, continuing to investigate transfer learning in Monte Carlo Tree Search would not have been possible. Our implementation, having been written in Python, is highly accessible to the scientific community and may be expanded on in future works.

Further, we wrote a Python implementation of Monte Carlo Tree Search compatible with our representation and move generators. This section of work, though apparently small on paper, constituted a large proportion of the total workload and was essential to the completion of our research. In particular, distribution over a machine cluster and pruning for memory management proved to be significant obstacles. To overcome the former, we used the *ray* Python library; for the latter, we created a novel pruning algorithm, *dropping*, which attempts to maximise the memory returned after every prune.

We constructed two trees using Monte Carlo Tree Search. The first of these was built in the classic chess domain, the second in the Chessplus domain. Each was given 10,000 algorithm iterations to navigate the search space. These trees were constructed through the aggregation of thirty root-parallelized trees built on a cluster of machines. This method of average selection allowed us to rapidly construct our two trees. In addition, the use of multiple machines avoids the problem of local minima, as one machine may find an apparently promising move before a more promising move, and expend energy exploring the former which could be better used exploring the latter.

Finally, we transferred the first tree, trained to play classic chess, to the Chessplus domain. Fine-tuning involved performing a further 10,000 iterations of Monte Carlo Tree Search on this transferred tree.

To test for transfer learning, we simulated 1000 games played between the transferred tree and the tree trained originally in the Chessplus domain (the Chessplus tree). Results indicated that positive transfer had occurred: 47.4% of games were won by the transferred tree, while only 35.7% were won by the Chessplus tree. A binomial test confirmed that these results are highly improbable ($p = 0.0000279$) under the assumption that the trees play with equal strength, allowing us to accept the alternative: the transferred tree is the stronger player. However, the proportion of moves played by each tree which were Chessplus-exclusive, in both games won and games lost, was roughly equal. This meant that we could not draw any conclusions about the relative strength of Chessplus-exclusive moves over classic chess moves. Indeed, this observation indicates that positive transfer occurred perhaps only due to the higher number of overall Monte Carlo Tree Search iterations completed by the transferred tree. We also looked at the number of moves selected from these trees versus those generated through online play. We found that in most games, very few moves were selected from the original trees. This, alongside the higher win rate for the transferred tree, indicates that those early moves had a significant effect on the outcome of each game.

7.2 Limitations and Future Work

We have made every effort to ensure this research is as valuable as it can be. However, given time and resource constraints, some decisions were made which will have affected the quality of our outcomes.

Firstly, Monte Carlo Tree Search is, by its nature, an anytime algorithm. The longer such as algorithm is permitted to run, the better the quality of the solutions it produces. Better results may be obtained simply by allowing more time for each tree to be constructed. In our research, each iteration of the algorithm may be seen as a discrete timestep; running the algorithm for ‘longer’ is equivalent to increasing the number of iterations completed by each tree.

Secondly, and significantly, our decision to implement our solutions in Python had a significant impact on the performance of our move generators and of Monte Carlo Tree Search. This is the primary reason why we chose to pursue a strict iteration limit when constructing trees, rather than a true time limit. More efficient implementations in a low-level language such as C would allow for orders of magnitude more computations in an equal amount of time.

Thirdly, major limitations were encountered due to constraints on computation resources. In particular, due to the memory limitations of a single machine, we were required to expend significant effort on the distribution and parallelization of Monte Carlo Tree Search. Our chosen method, root parallelization, is substantially simpler to implement than other methods. However, each parallel tree is unable to exploit results found by other trees, leading to inefficient use of search time. Further, the cluster used had only thirty machines; better results may easily be obtained by increasing this number.

Fourthly, we used a novel pruning algorithm, dropping, to assist in memory management during tree construction. This technique seeks to maximise the liberation of memory resources. While this is suitable for our purposes, in practice, this disposes of a great amount of information which could otherwise have been exploited by the search algorithm. Future works may seek to improve upon the dropping algorithm, or investigate the results obtained by using different pruning algorithms.

Bibliography

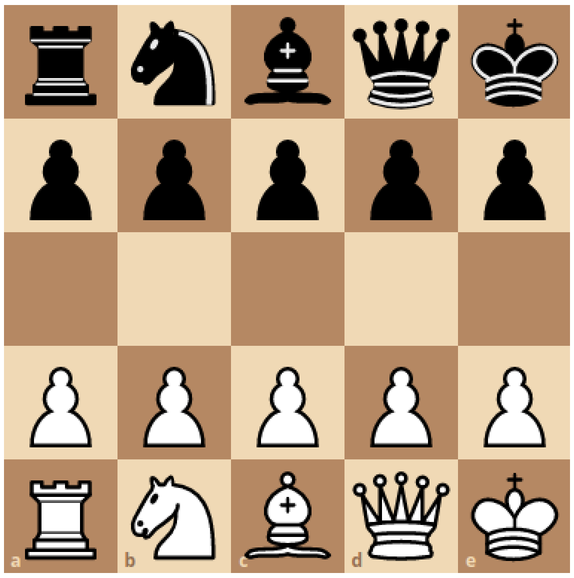
1. Silver, D., et al., *Mastering the game of Go with deep neural networks and tree search.* 2016. **529**(7587): p. 484-489.
2. Syed, O. and A. Syed, *Arimaa-a new game designed to be difficult for computers.* 2003. **26**(2): p. 138-139.
3. Myerson, R.B., *Game theory*. 2013: Harvard university press.
4. Browne, C.B., et al., *A survey of monte carlo tree search methods.* 2012. **4**(1): p. 1-43.
5. Allis, L.V., *Searching for solutions in games and artificial intelligence*. 1994: Ponsen & Looijen Wageningen.
6. Shannon, C., *Programming a computer for playing chess (pp. 2-13)*. 1988, Springer New York.
7. Abdelbar, A., *Alpha-Beta Pruning and Althöfer's Pathology-Free Negamax Algorithm.* 2012. **5**(4): p. 521-528.
8. Pearl, J., *Asymptotic properties of minimax trees and game-searching procedures.* 1980. **14**(2): p. 113-138.
9. Pearl, J., *The solution for the branching factor of the alpha-beta pruning algorithm and its optimality.* 1982. **25**(8): p. 559-564.
10. Gelly, S., et al., *The grand challenge of computer Go: Monte Carlo tree search and extensions.* 2012. **55**(3): p. 106-113.
11. Bouzy, B. and C. Tutorial. *Old-fashioned computer go vs monte-carlo go*. in *IEEE Symposium on Computational Intelligence in Games (CIG)*. 2007.
12. Kocsis, L. and C. Szepesvári. *Bandit based monte-carlo planning*. in *European conference on machine learning*. 2006. Springer.
13. Kocsis, L., C. Szepesvári, and J.J.U.T. Willemson, Estonia, Tech. Rep, *Improved monte-carlo search.* 2006. **1**.
14. Shah, D., Q. Xie, and Z. Xu. *Non-asymptotic analysis of monte carlo tree search*. in *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*. 2020.
15. Gelly, S. and D. Silver. *Combining online and offline knowledge in UCT*. in *Proceedings of the 24th international conference on Machine learning*. 2007.
16. Gelly, S. and D. Silver, *Monte-Carlo tree search and rapid action value estimation in computer Go.* 2011. **175**(11): p. 1856-1875.
17. Sun, Y., Z. Zhang, and X. Wang. *The research of UCT and rapid action value estimation in NoGo game*. in *2016 Chinese Control and Decision Conference (CCDC)*. 2016. IEEE.
18. Lewis, P.P.A., *Ensemble monte-carlo planning: An empirical study.* 2010.
19. Powley, E., P.I. Cowling, and D. Whitehouse, *Memory bounded Monte Carlo tree search.* 2017.
20. Enzenberger, M., et al., *Fuego—an open-source framework for board games and Go engine based on Monte Carlo tree search.* 2010. **2**(4): p. 259-270.
21. Cazenave, T. and N. Jouandeau. *On the parallelization of UCT*. 2007.
22. Chaslot, G.M.-B., M.H. Winands, and H.J. van Den Herik. *Parallel monte-carlo tree search*. in *International Conference on Computers and Games*. 2008. Springer.
23. Soejima, Y., et al., *Evaluating root parallelization in Go.* 2010. **2**(4): p. 278-287.
24. Obata, T., et al. *Consultation algorithm for Computer Shogi: Move decisions by majority*. in *International Conference on Computers and Games*. 2010. Springer.
25. Snir, M., et al., *MPI--the Complete Reference: the MPI core*. Vol. 1. 1998: MIT press.
26. Kato, H. and I. Takeuchi. *Parallel monte-carlo tree search with simulation servers*. in *2010 International Conference on Technologies and Applications of Artificial Intelligence*. 2010. IEEE.
27. Yoshizoe, K., et al. *Scalable Distributed Monte-Carlo Tree Search*. in *SoCS*. 2011.

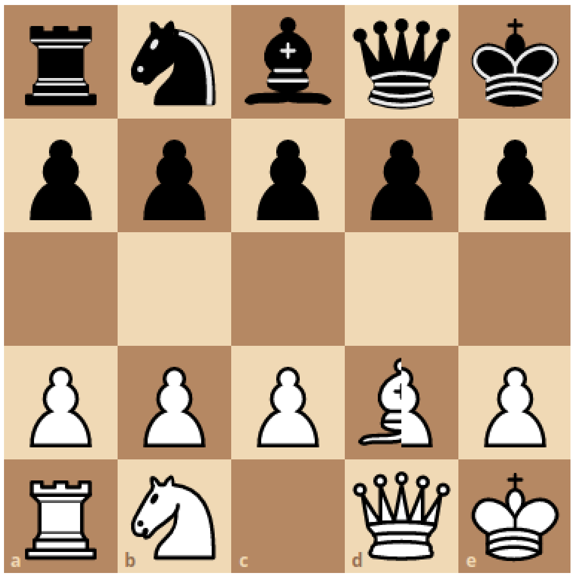
28. Farsal, W., S. Anter, and M. Ramdani. *Deep learning: An overview*. in *Proceedings of the 12th International Conference on Intelligent Systems: Theories and Applications*. 2018.
29. LeCun, Y., Y. Bengio, and G.J.n. Hinton, *Deep learning.* 2015. **521**(7553): p. 436-444.
30. Schmidhuber, J., *Deep learning in neural networks: An overview.* 2015. **61**: p. 85-117.
31. Minsky, M. and S.A. Papert, *Perceptrons: An introduction to computational geometry*. 2017: MIT press.
32. Tesauro, G. and T.J. Sejnowski. *A'neural'network that learns to play backgammon*. in *Neural information processing systems*. 1988.
33. Bengio, Y., P. Simard, and P.J.I.t.o.n.n. Frasconi, *Learning long-term dependencies with gradient descent is difficult.* 1994. **5**(2): p. 157-166.
34. Cho, K., et al., *Learning phrase representations using RNN encoder-decoder for statistical machine translation.* 2014.
35. Hochreiter, S. and J. Schmidhuber, *Long short-term memory.* 1997. **9**(8): p. 1735-1780.
36. Russell, S. and P. Norvig, *Artificial intelligence: a modern approach.* 2002.
37. Mou, L. and Z. Jin, *Tree-Based Convolutional Neural Networks: Principles and Applications*. 2018: Springer.
38. Silver, D., et al., *Mastering the game of go without human knowledge.* 2017. **550**(7676): p. 354-359.
39. Konen, W. and T. Bartz-Beielstein. *Reinforcement learning for games: failures and successes*. in *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*. 2009.
40. Pan, S.J. and Q. Yang, *A survey on transfer learning.* 2009. **22**(10): p. 1345-1359.
41. Asawa, C.E.C. and D. Pan, *Using transfer learning between games to improve deep reinforcement learning performance and stability, 2017.*
42. Mittel, A. and P. Sowmya Munukutla. *Visual Transfer between Atari Games using Competitive Reinforcement Learning*. in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2019.
43. Wang, Z., et al. *Characterizing and avoiding negative transfer*. in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019.
44. Mhalla, M. and F. Prost, *Gardner’s minichess variant is solved.* 2013. **36**(4): p. 215-221.

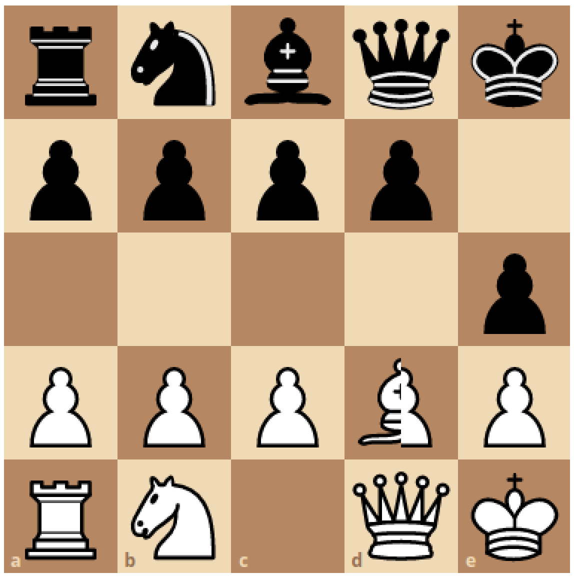
Appendices

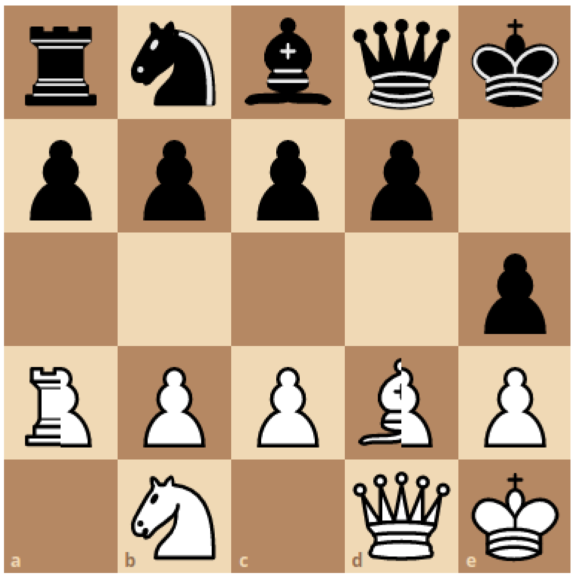
Appendix A

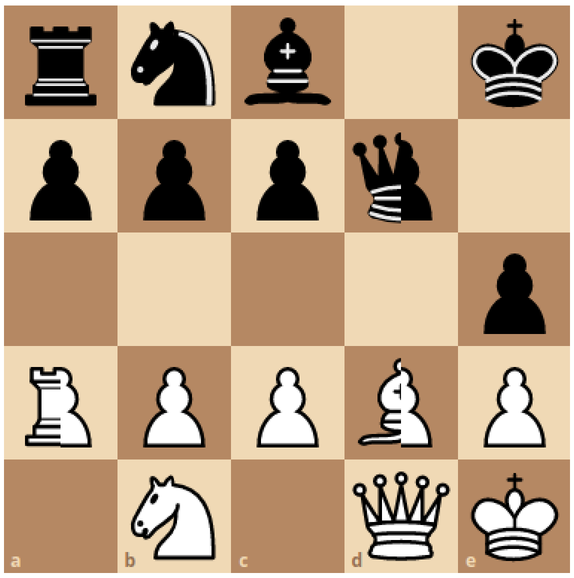
We provide a sample game played between the transferred tree and Chessplus tree. Move order is from left to right, top to bottom. White is played by the transferred tree, and black by the Chessplus tree.



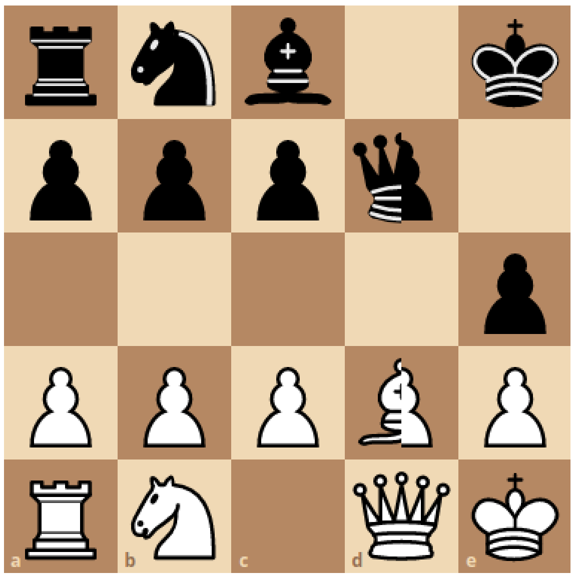




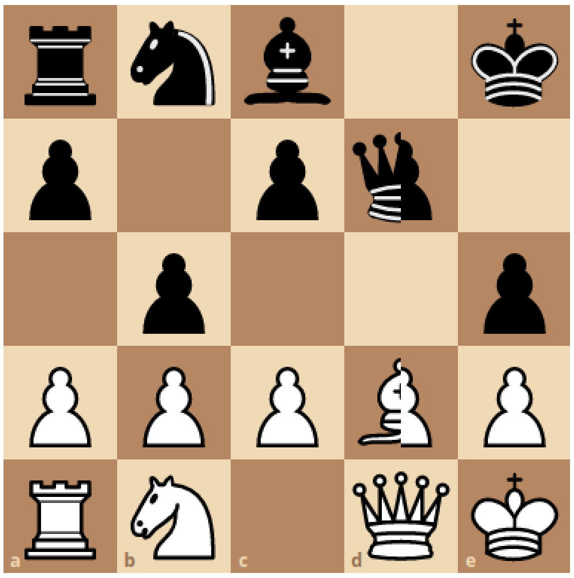




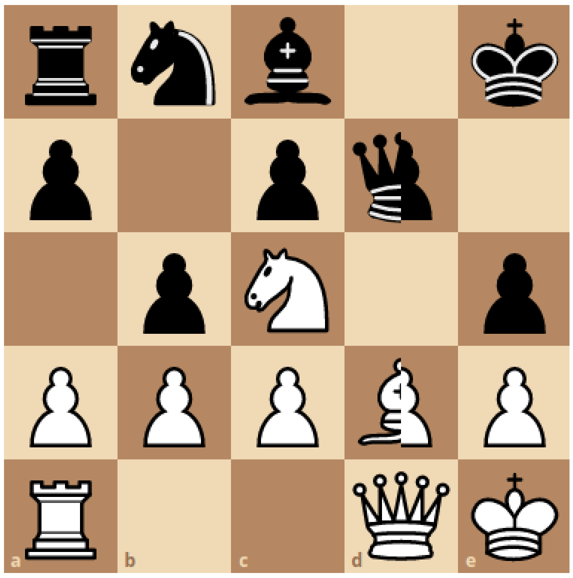
a
b
c
d
e



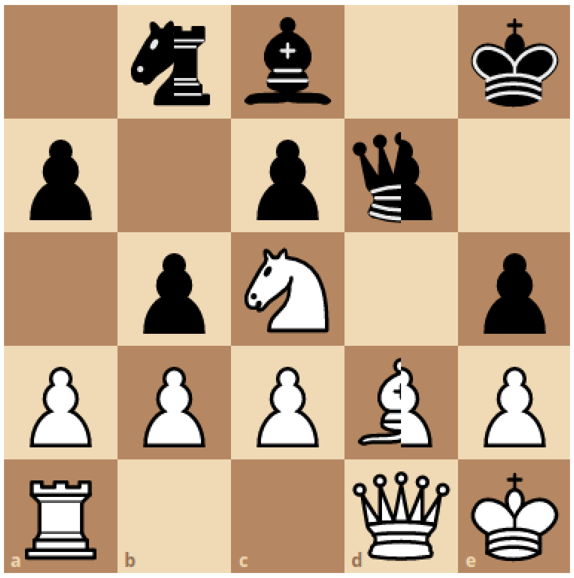
a
b
c
d
e



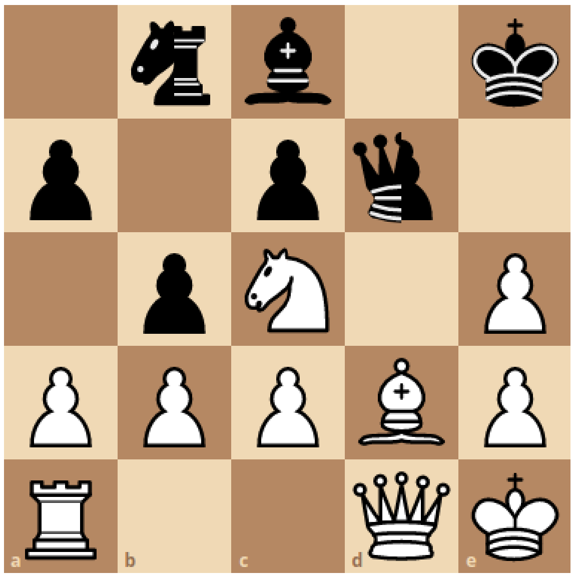
a
b
c
d
e



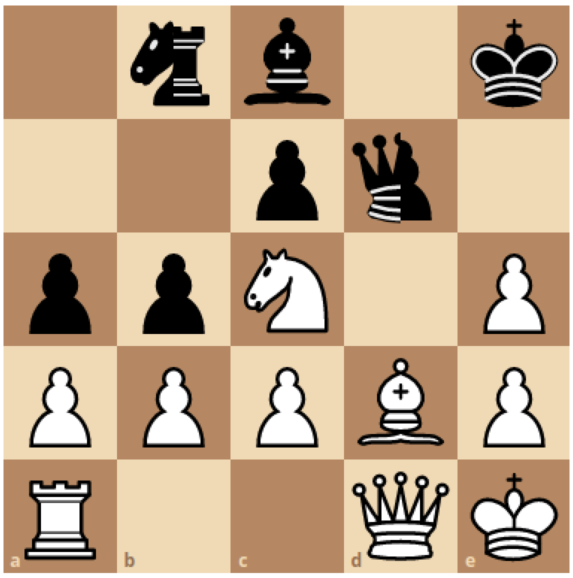
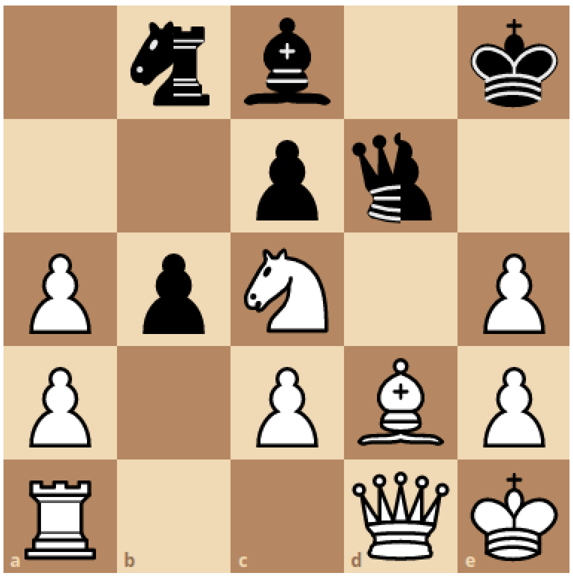
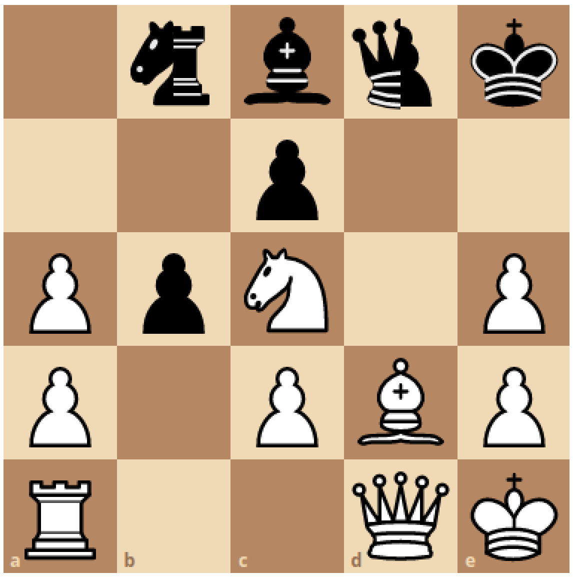
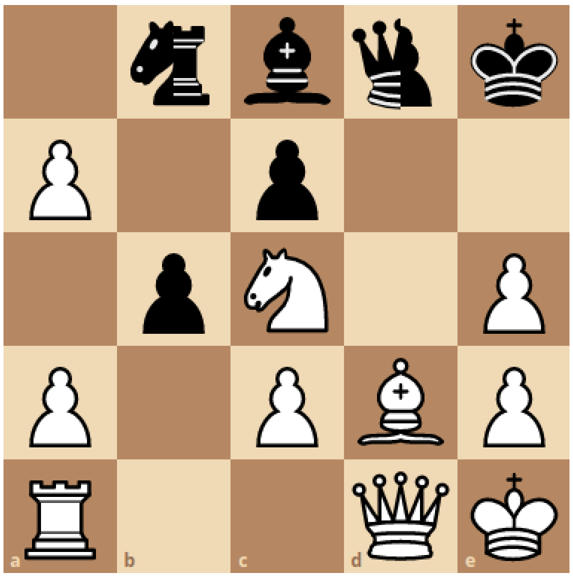
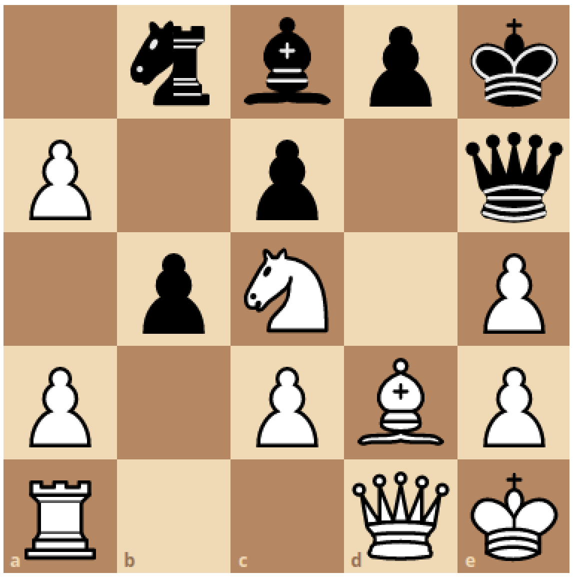
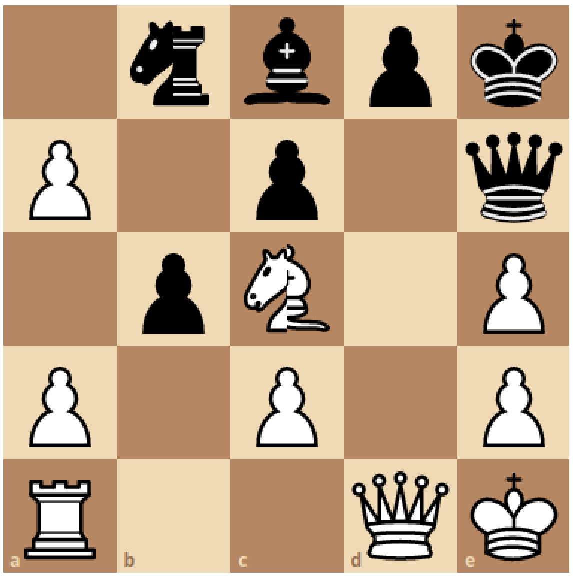
a
b
c
d
e

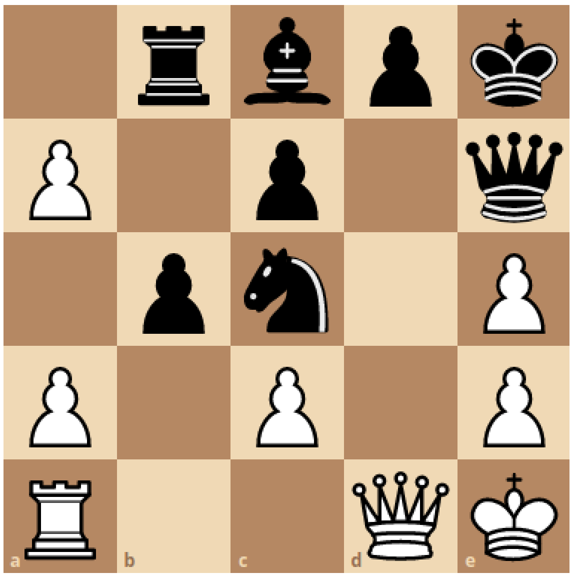


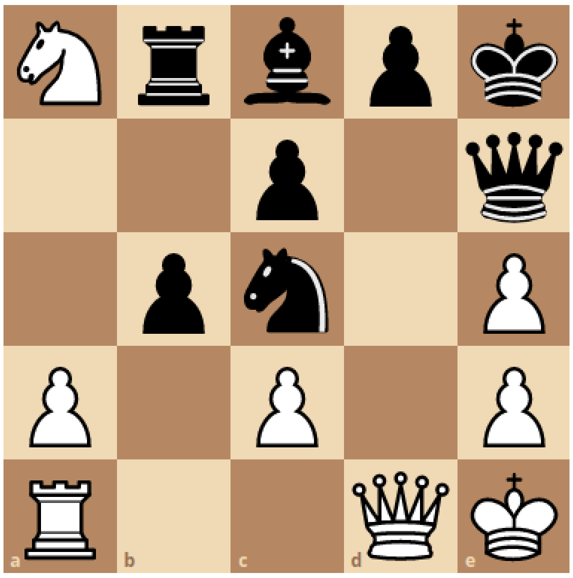
a
b
c
d
e

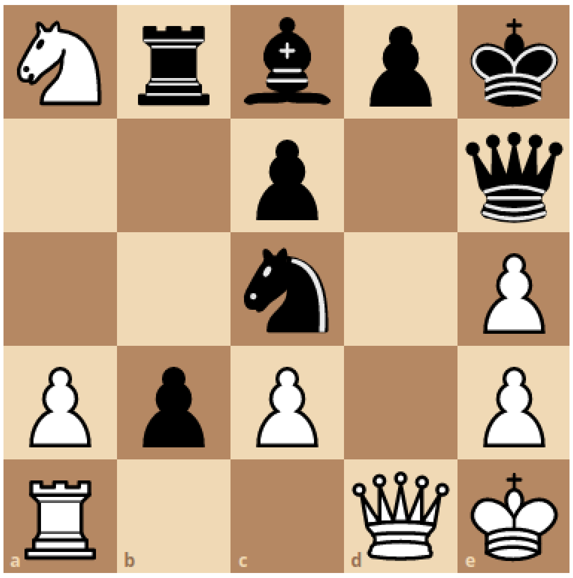


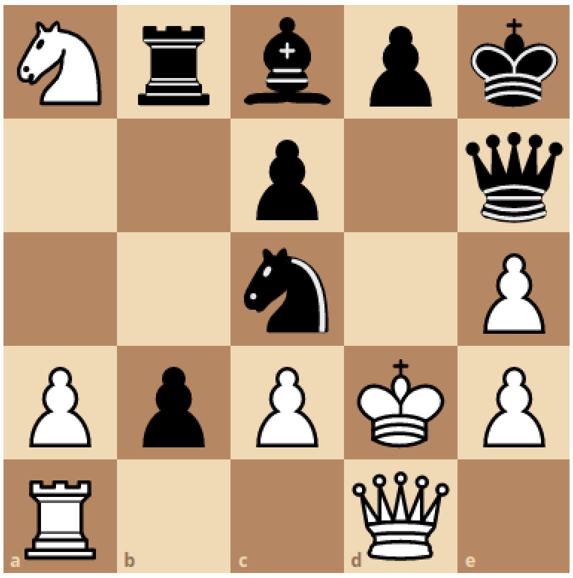
a
b
c
d
e

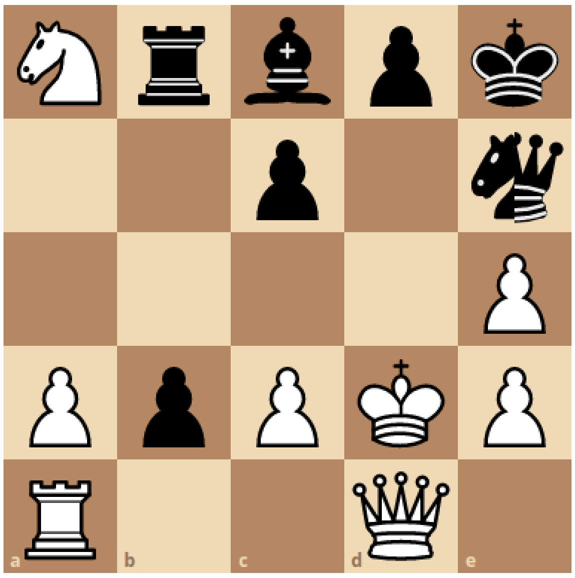







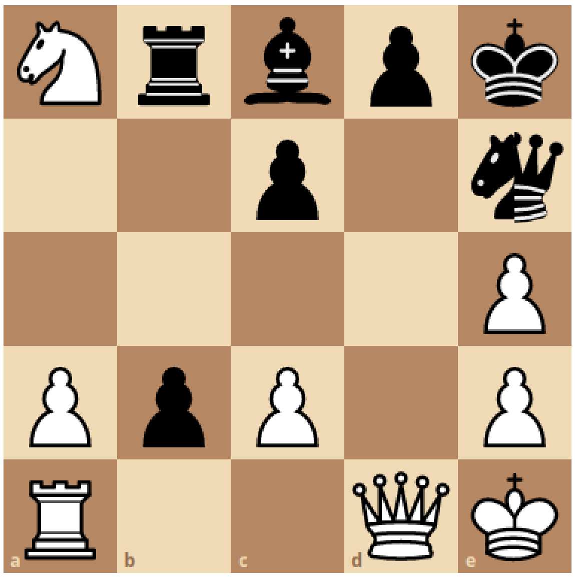

a
b
c
d
e

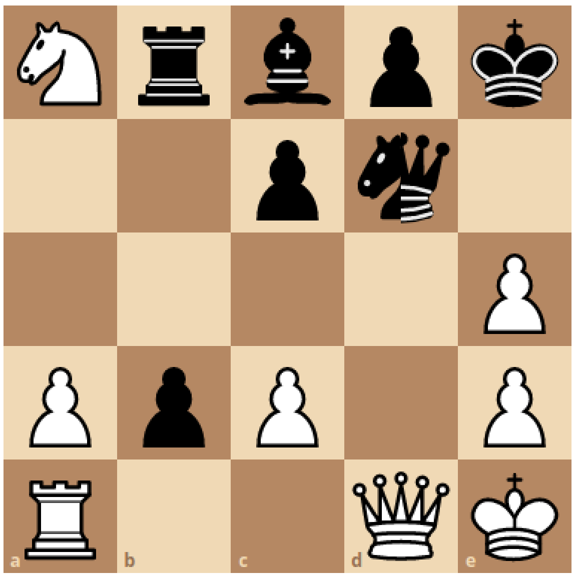
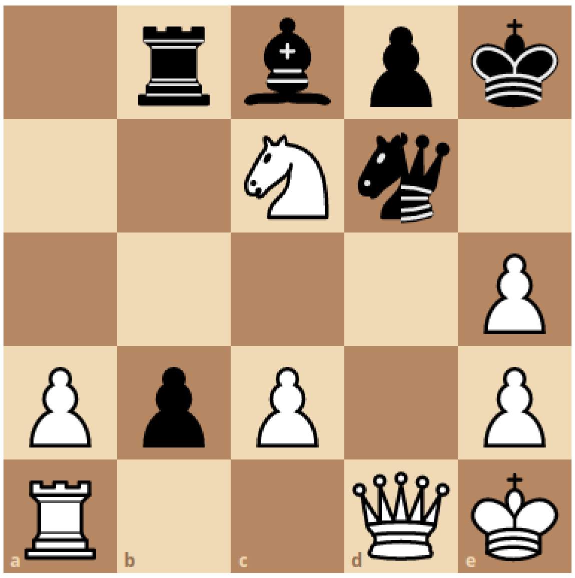
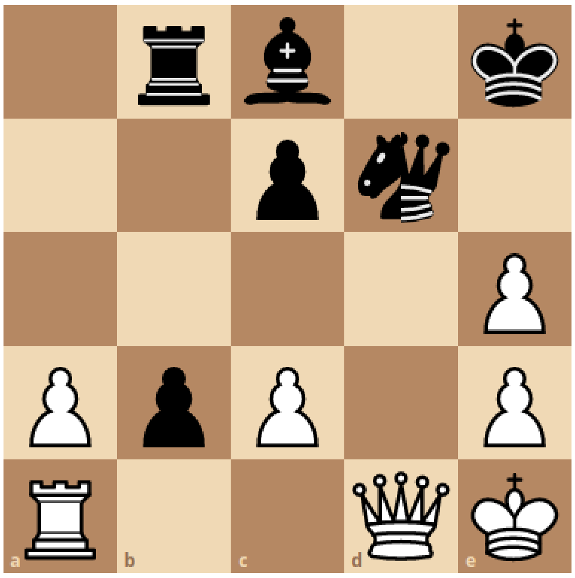
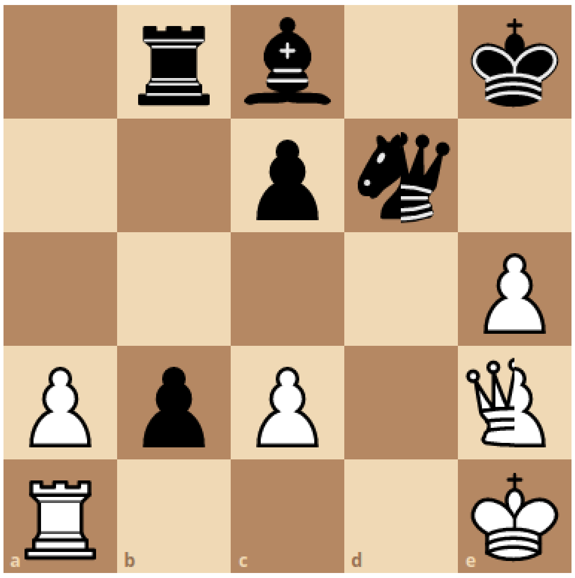
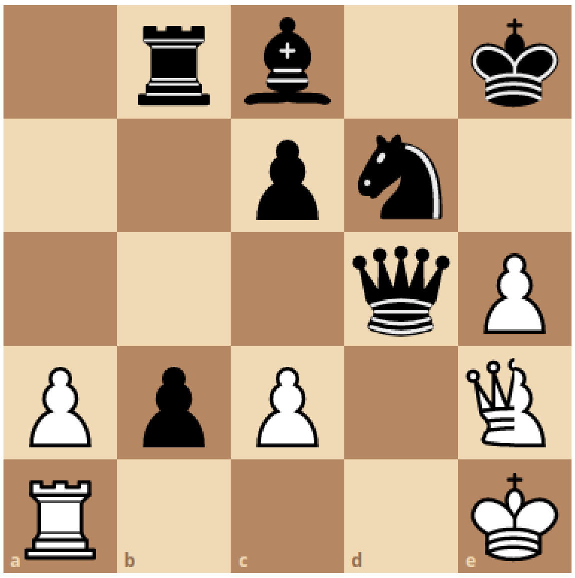
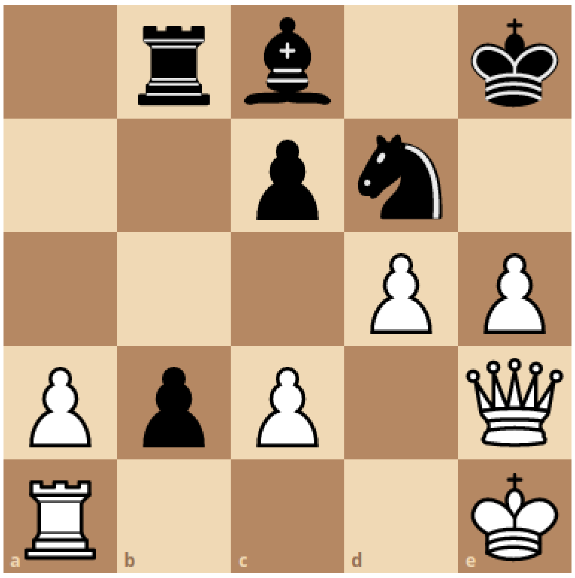

a
b
c
d
e

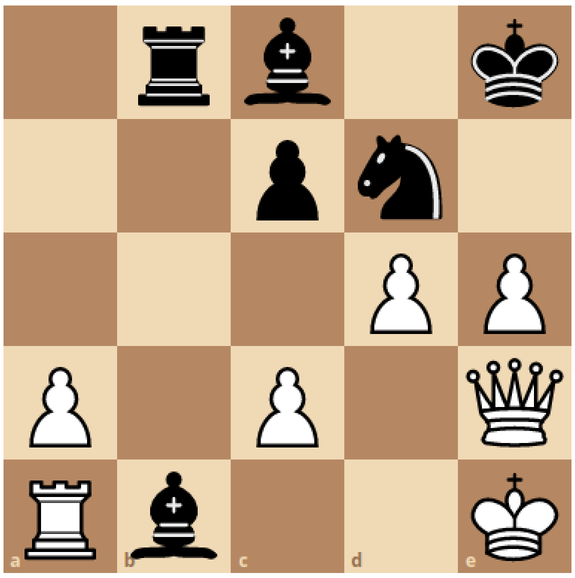
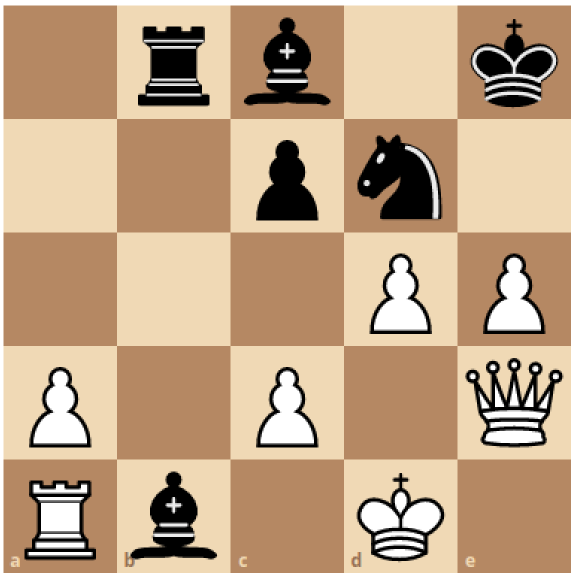
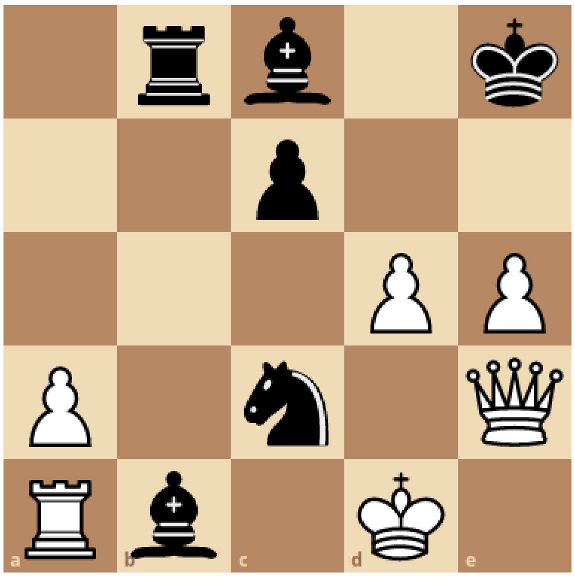
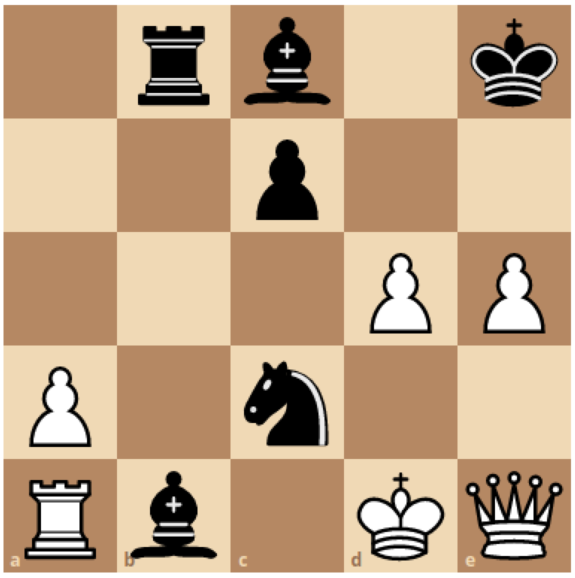
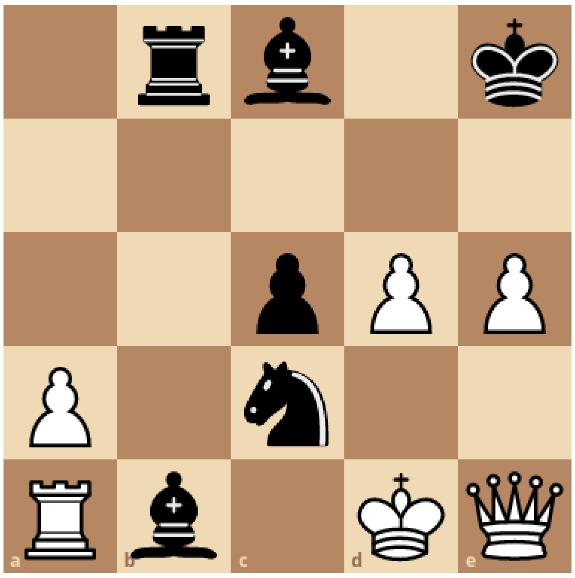
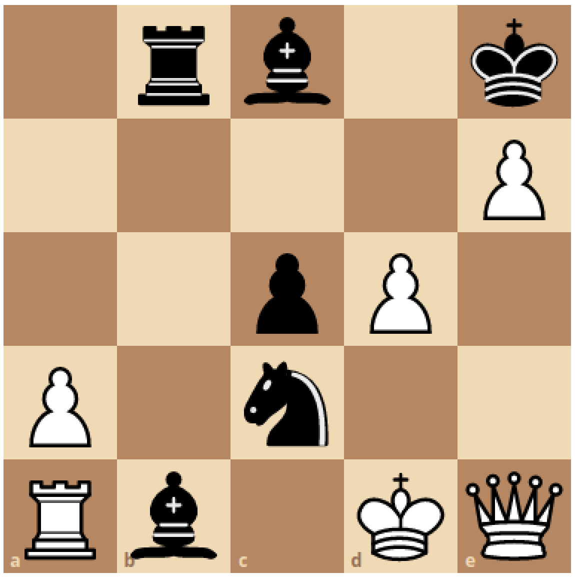

a
b
c
d
e

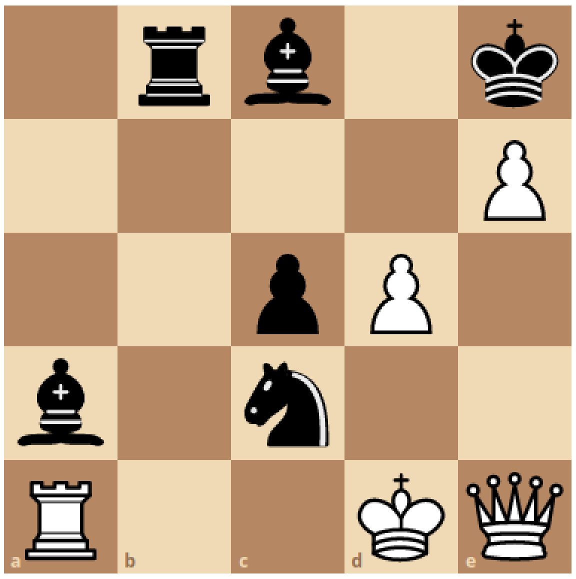

a
b
c
d
e

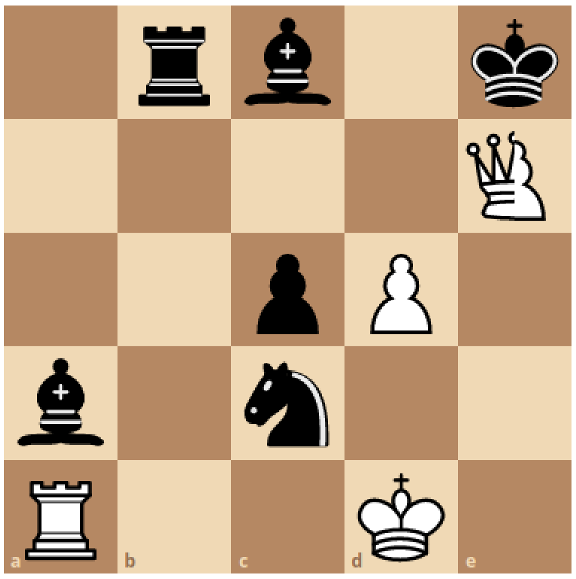

a
b
c
d
e


a
b
c
d
e


a
b
c
d
e

a
b
c
d
e

a
b
c
d
e

a
b
c
d
e

a
b
c
d
e

a
b
c
d
e


a
b
c
d
e

a
b
c
d
e

a
b
c
d
e

a
b
c
d
e

a
b
c
d
e

a
b
c
d
e





The transferred tree (white) wins by checkmate. Interestingly, this was achieved through a Chessplus move. One can observe seemingly weak moves by both players, and often moves that undo previous moves. These are a result of the low number of Monte Carlo Tree Search iterations completed, both offline and online.