CS310: Final Report

Improving Decision-Making in Chess by Implementing a Trap-Adaptive Monte Carlo Tree Search Algorithm

April 29, 2019

Author	Marek Topolewski
University ID	1633084
Year of Study	3
Supervisor	Paolo Turrini

Abstract

Although agents based on Monte Carlo Tree Search (MCTS) dominate in many games of perfect information, a number of challenges posed by the algorithm remain unaddressed. One of such issues is the expansion of trap states which has a negative impact on its performance. A domain that is highly susceptible to this disadvantage is Chess. Multiple extensions to the base implementation of MCTS have been proposed, including enhancements increasing its trap awareness, such as explicit markers to indicate trap states. Furthermore, solutions that abstain from exploiting domain-specific knowledge are favoured due to them preserving the algorithm's generality. The existing trap adaptive enhancements have been proven to improve the overall performance of the agent, however, their analysis indicates unresolved issues, most notably trap persistence verification. Further development of the algorithm and addressing this challenge may allow further advancements in the MCTS emerging as an optimal algorithm capable of General Game Playing.

Keyword list

Artificial intelligence, game theory, Chess, Monte Carlo, positional similarity, search trap, general game playing.

Contents

Chapter 1

Introduction

1.1 Motivation

The improvement of Monte Carlo (MC) methods has led to great advancements in the field of artificial intelligence. Computers managed to defeat human world champions in games of perfect information for which it was previously deemed impossible. One of such examples is the ancient game of Go, which has been mastered by people since the VI century BC. It had seemed to be infeasible for a computer to reach human level of play due to its high branching factor (over 7 times greater than Chess [1]) resulting in $10^{10^{48}}$ possible board configurations. Deep Mind's AlphaGo overcame this challenge as a result of employing the Monte Carlo Tree Search (MCTS) and defeated 1-4 one of the best Go players of all time - Lee Sedol [30].

This achievement proved how powerful and efficient statistical approaches are and sparked interest in the area of Monte Carlo methods. Since its first formal definition MCTS has been tested in numerous domains, not only in Go but also Poker, Checkers, Connect 5 and many others. Moreover, a great number of enhancements improving its performance have been developed and tested. The versatility and performance of the algorithm makes MCTS one of the leading candidates for creating an agent that is able to infer intelligent decisions in any game, therefore, suitable for General Game Playing (GGP) [27]. Although AlphaZero, the subsequent iteration of AlphaGo also utilising MCTS, can attain a master level of play regardless of the domain, the algorithm incorporates *a priori* knowledge from hours self-play [31]. Hence, a GGP algorithm able to operate without domain-specific or past knowledge is yet to be developed. Such a breakthrough could lead to further advancements such as the creation of an AI capable of making optimal decision also outside of game environment.

Unfortunately, unsolved challenges in MCTS still exist and result in poor performance in some domains, most notably in Chess. Although the game has a lower branching factor than Go, it produces trap states more frequently [47]. If these search traps are not avoided, they may lead to a severe downgrade in overall gameplay. The aim of this paper is to research the nature of trap states and improve the existing trap detection mechanisms for MCTS.

1.2 Importance

Chess was named one of the unsuitable games to be solved by MCTS due to frequently occurring trap states [47]. The project builds upon the Monte Carlo Chess engine developed by Oleg Arenz (2010) [4], explores trap awareness issues encountered in the thesis by Benjamin Chun How Aw (2017) [6] and aims to improve the performance of the trap detection mechanisms. To achieve its goal the project considers relevant extensions to the MCTS that has been proposed in the recent years as well as artificial intelligence techniques outside of the MC methods.

Despite its superior performance over other state-of-the-art algorithms in environments such as Go, MCTS is vulnerable to traps in other board games of perfect information. Successful solutions adopt heuristic approaches, such as hard-coded trap states or neural networks trained to detect them. In this project, however, the goal is to provide an aheuristic extension to the MCTS algorithm itself and potentially pave the way for it to become a candidate for a General Game Playing method. This way, the algorithm could be deployed in any environment, with no prior knowledge except for the rules (constraints) and yet be capable of expert gameplay.

The ultimate goal of the project is to provide a more adaptive trap awareness technique for MCTS and evaluate the outcomes in a game well-known to generate the trap problem - Chess. Developing an algorithmic solution without incorporating environmental knowledge is likely to revive the interest in the Monte Carlo methods in the game theory. Despite its great success thus far, before MCTS can be named a prospective algorithm to be capable of GGP, its critical disadvantages must be addressed. This project intends to produce an aheuristic answer to one of such issues - search traps.

1.3 Objectives

The primary goal of the project is to provide further insight into the **trap adaptiveness of Monte Carlo Tree Search** and improve previously proposed solutions to trap detection and avoidance mechanisms. Paramount focus will be directed towards the **positional similarity theory** which utilisation can enhance the performance of the trap awareness algorithm.

1.4 Project Outline

A deep understanding of algorithm's functionality in its base implementation is necessary for the analysis of potential extensions. Hence, the project will outline the MCTS algorithm along with the necessary game theoretic knowledge including Markov Decision Process, Nash Equilibrium, game trees and adversarial search methods. The modified implementation will be then examined in contrast with the original to evaluate its performance. The following elements constitute the overall plan of this project report:

1. Chapters 1 and 2: **Understanding of MCTS**.
   Discussion of the related literature and notions necessary to understand the operation of MCTS with its extensions and the nature of trap states.

2. Chapter 3: **Model development**.
   Introduction of the positional similarity theory and modifications to the MCTS.
3. Chapter 4: **Implementation**.
   Presentation of the implemented enhancements to the trap adaptive Chess engine.
4. Chapter 5: **Evaluation**.
   Theoretical and empirical evaluation of the developed components and the algorithm with integrated modifications.
5. Chapter 6: **Conclusion**.
   Conceptualisation of the obtained results and project's summary with reference to its motivation. Present the future development of the MCTS and the search traps problem arising from the introduced concepts.

1.5 Choice of Domain

Artificial Intelligence algorithms are oftentimes developed in the environment of games as they provide a well-defined set of rules governing its behaviour, hence, simplifying the modelling. Despite Chess being the chosen environment, the MCTS has been employed in numerous domains in the past and its application scope exceeds games. Not only does that include perfect information (also deterministic) multi-player games of (Go, Checkers, Connect 5) but furthermore imperfect information (also non-deterministic) games (Poker, Backgammon, Wumpus World) and single-player games (mazes, Sudoku) [11]. Such large variety of applications is possible as a consequence of MCTS being aheuristic, meaning it lacks the use of domain-specific knowledge, and hence, making it a potential solution to the GGP problem.

In a number of these domains MCTS has achieved great results and many improvements have been developed to enhance its gameplay, not all of which uphold the generality of the original algorithm. The domain of Chess has been selected for this project for the following reasons:

- **Results of MCTS**. MCTS is not yet capable of satisfactory gameplay in Chess where minimax algorithms still outperform it. Developing modifications to its algorithm without using domain-specific knowledge may greatly improve its generality.
- **Existence of trap states**. It has been proven that Chess, in contrary to for example Go, frequently generates such search pitfalls [47]. Numerous traps that are only several moves deep (1-7) can be generated which makes them easily reproducible, and therefore, provides excellent means for testing.
- **Overall complexity**. Even though the board size is only 8x8 (compared to 19x19 in Go), the game of Chess consists of six figure kinds with distinct movement patterns, as well as other special rules (promotion or castling) [44]. Moreover, the mean number of moves in a match (also depth) is 80 while at every state the player has the average of 35 possible actions to choose from (also depth) [1]. This makes the domain sufficiently complex for the project to be challenging and demonstrate the strength of MCTS.
- **Popularity and infrastructure**. Chess is one of the most known board games of all time. General public interest in the game is a key factor, it means that the paper is accessible to a larger audience and extensible infrastructure has been already developed. This most notably includes variety of Chess engines, tournament programs and evaluation functions.

Chapter 2

Background

2.1 Game Theory

Game theory problems are an extension to the decision theory problems which allow multiple agents within the same environment and define rules of the game that allow these agents to interact. A formal description of a game is a set of the following elements [11]:

S	set of states forming the state space,
$s_0 \in S$	the initial state,
$S_T \subseteq S$	the set of terminal (final) states,
$n \in \mathbb{N}$	number of players,
A	set of actions,
$f : S \times A \to S$	state transition function,
$R : S \to \mathbb{R}^k$	utility function,
$\rho : S \to (0, 1, ..., n)$	sequence representing players about to act in each state.

A game begins in the initial state s_0 and after a sequence of transitions over time $t = 1, 2, ..., m$ (where m is the game length) the game ends in one the final states in S_T.

A subclass of games in which all agents share the full knowledge of the environment is known as the *games of perfect information*. The information includes possible actions the agent and the opponents can take as well as their actions' corresponding outcomes.

Extensive form games, are a subset of games of perfect information which can be represented as a *game tree* (see Figure 2.1). The tree begins at the root s_0 and each child node represents an outcome of the state transition function f from its parent node. A depth level of the tree corresponds to turn n in which the player defined by the ρ function is about to act.

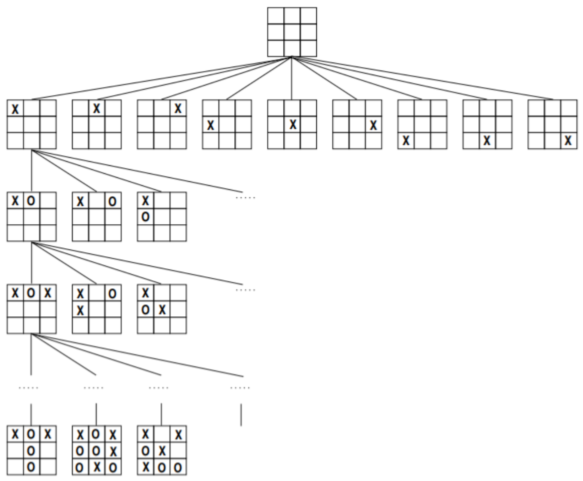


Figure 2.1: Partially expanded Tic-Tac-Toe game tree [52], where the top node is the root (an empty board) and the bottom leaves are the terminal states (a draw or either player wins).

Furthermore, in an extensive form game each player n has a policy π_n according to which they select their subsequent move. A policy π, often referred to as a *strategy*, is defined as a function that maps an action a from a given state s to a probability value of selecting that move $p(s,a)$. It determines the actions of the player at each stage of the game, for which the player receives an expected utility $R(s')$ at a terminal state $s' \in S_T$.

Next important notion that will be introduced is the *Nash equilibrium*, which in the given context of game theory is defined as a set of strategies (policies) for each player in the environment, such that none of the players can benefit by adjusting their strategy.

$$\forall n \in N, \pi_n : R(\pi_n, \pi_{-n}) \geqslant R(\pi_n^*, \pi_{-n}) \tag{2.1}$$

where π_{-n} is the strategy of every player but n and π_n^* is an alternative strategy of n to π_n.

Every agent partaking in a game generally assumes a mirror match (opponents are the same as itself), e.g. opponents are always optimal for an agent based on minimax (see section 2.4.1), hence, resulting in a Nash equilibrium. From its definition it can be further derived, that a agent will not be basing its model on choices that the opponents makes if its actions are inconsistent with the ones deemed as favourable by the agent itself, instead it should treat them as a mistake.

The final term included in this section is the *backward induction*. Assuming that all players take relational subsequent actions, this process allows the player to deduce the most profitable action at a given state by deducing it from the terminal history preferable to that players (further discussed in 2.4.1). Backward induction is possible due to the existence of *equilibrium paths* in extensive form game trees [40]. *Nash's Existence Theorem* states that in the extensive form games of perfect information (e.g. Chess), at each node along the tree, a Nash equilibrium is bound to exist, therefore, forming equilibrium paths in the game tree [41].

2.2 Chess

Chess is one of the most popular board games in the world. Its recognition is heavily linked to the mild learning curve allowing beginners to easily grasp the rules and yet pose a tough challenge to the professional players. Although many games have been studied to develop artificial intelligence techniques, the game of Chess is one of the most well-suited environments.

Chess can be played in a two-player (most common) or multi-player variant. Regardless, the key properties of the game are:

- **Deterministic**. Given the current state of the environment and a legal action that the agent is about to select from that state, the agent undoubtedly knows what the subsequent state is. There is no uncertainty generated by the environment or the agents.
- **Perfect Information**. The entire environment is visible to each player (see section 2.1). However, in Chess the agents are not aware of the opponents' strategies or the payoffs associated with each action outcome, hence, it is not a game of complete information.

- **Zero-sum**. The values of rewards gained (or lost) by each player sum to zero. In the case of two-player games (such as Chess) this property implies that both players are in strict competition and that the utility gain of one player is equivalent to the utility loss of the other.

The game of Chess is also an instance of the *Markov Decision Process* (MDP), which is a subset of decision problems. For all MDP problems the *Markov property* must hold, which says that the reward from transition outcomes (i.e. subsequent states) from any given state $s \in S$ depend solely on the that state and not the prior history [36]:

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_t, s_{t-1}, ..., s_1) \tag{2.2}$$

Problems that are MDP allow for the sequential evaluation of the utility $R(s)$ a a given state s by a sequence of decisions.

Due to Chess being both an extensive form game and an MDP, a game of Chess can be represented as a game tree. The root of the game tree s_0 is the initial configuration of the board (see Figure 2.2 generated using *Arena* Chess engine GUI [9]). Reaching a leaf node (terminal state) in the game tree is equivalent to satisfying an end condition, which may vary depending on the rules of the tournament. Most common conditions are:

- **Checkmate**. A game state in which player's king is in *check*, i.e. the king is in a direct danger of capture and no legal action exists to avoid the threat (see Figure 2.3). Checkmating a player results in a win for their opponent.

- **Resignation**. Either of the players may wish to resign from continuing to play. This results in a win for their opponent.

- **Draw**. The game can drawn in several cases, these include: (1) both players agree to draw the game, (2) threefold repetition - same position has been reached at least three times, (3) stalemate - player about to make a move is not in check but has no legal actions to take.

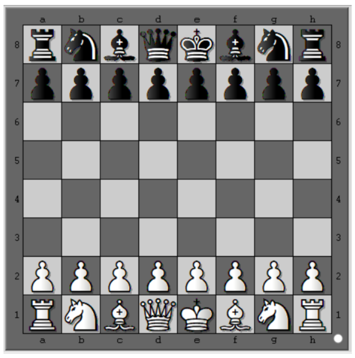


Figure 2.2: Initial configuration of a standard Chess board.

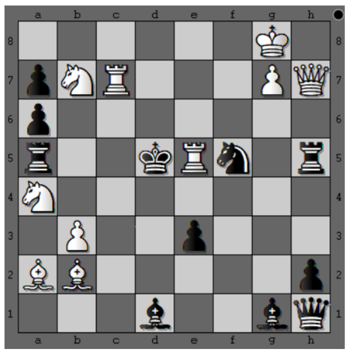


Figure 2.3: A terminal configuration of a Chess board. The black king is in a check.

2.3 Heuristic Evaluation

Heuristic evaluation functions are employed to assign a score, also known as desirability value, to each possible state. The exact solution of defining such mapping may vary, but it generally involves employing domain-specific knowledge and measuring hard-coded parameters. The Chess engine examined in this project is based on Stockfish 8, which estimates the desirability of transitioning to the given board configuration based on the following parameters [48]:

- **Pawn structure**. Score based on the position of players Pawns. High values for Pawns positioned in the centre of the board and low values for Pawns that are doubled, backward or blocked.
- **Piece placement**. Encourage effective positioning of player's pieces, e.g. Knights in board's centre, Queens and Rooks on each others' diagonal for mutual protection from capture moves.
- **Threat**. Encourage placement of pieces in the protection of other pieces.
- **King safety**. Prefer moves that protect the King. Such actions include positioning the King in a corner, building a protection shield from Pawns, maintaining a Pawn shield.
- **Mobility**. The heuristic function favours moves that allow more freedom in terms of possibility of moves in subsequent turns of the player.

Although the Monte Carlo Tree Search does not employ a hard-coded heuristic evaluation, neither in the original algorithm nor in the implementation of this project, the method may be utilised in development. One of such application is the decision evaluation, in which such a function allows to measure the strength of agent's move selections by calculating the gain in heuristic values in a sequence of states.

2.4 Adversarial Search

Adversarial search problems are used to describe games of perfect information in a competitive multi-agent environment, where players' goals are in conflict. The opponents can introduce contingencies into the the process of problem-solving [52]. A potential answer to the issue may involve the use of an agent which behaviour is modelled by the Monte Carlo Tree Search. To provide further context for the project, the state-of-the-art algorithms providing optimal solutions to such problems will be covered.

2.4.1 Minimax Search

The *Minimax Tree Search* is an algorithm that yields *optimal strategy* given that its opponents also employ an optimal policy. An optimal strategy is a strategy such that there does not exist another policy which generates an higher reward. In this project, a special version of the method is considered, which restricts its application to two-player zero-sum games of perfect information, however, extensions exist to facilitate multiplayer and non-zero-sum perfect information domains [52]. The agent is the MAX player and attempts to maximise the utility gain at each MAX node of the game tree, while the opponents are MIN players who minimises the agent's reward at each MIN vertex. The following recursive formula can be defined to assign a *minimax value* of each node n of the game tree:

$$\text{minimax}(n) = \begin{cases} R(n) & \text{if } n \in S_T \\ \max_{s \in \text{children}(n)} \text{minimax}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in \text{children}(n)} \text{minimax}(s) & \text{if } n \text{ is a MIN node} \end{cases} \tag{2.3}$$

The algorithm is executed whenever the MAX player is about to move, hence, the MAX node is the root of the game tree. Subsequently, the MIN and MAX player vertices are interleaved at each tree level until a leaf node is reached. Recursively applying the above formula (depth-first order) results in the terminal values being *backed up* through the tree in a process known as *backpropagation*. At the root node the algorithm will choose the vertex with the highest minimax value, i.e. the backpropagated reward. This final selection is known as the *Minimax decision* (see Figure 2.4).

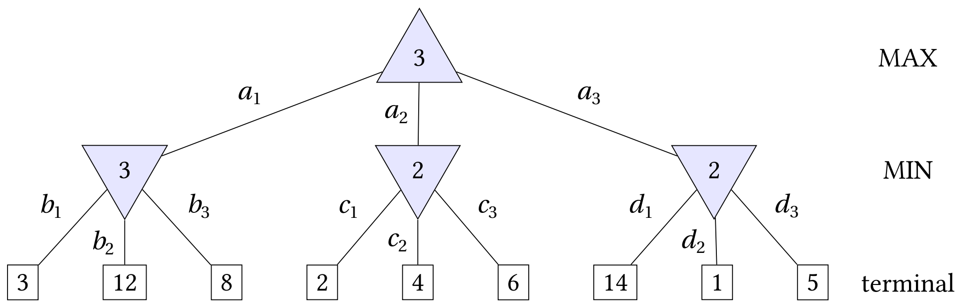


Figure 2.4: An example of a 2-ply (two turns, one of each player) Minimax tree, where $a, b, c \in A$ are legal actions from corresponding states. The leaf nodes represent terminal states to which a reward can be assigned through the utility function. Each MIN node (here: tree depth 1), stores the minimal value of its children. At the root, the MAX player will select action a_1 as the Minimax decision, as it yields highest reward.

The two considerable disadvantages of Minimax Tree Search are: (1) the optimal opponent assumption which may result in sub-optimal play if the condition does not hold, (2) the necessity to explore the entire game tree which can be computationally infeasible in games with high branching factor.

2.4.2 Alpha-Beta Search

The *Alpha-Beta Tree Search* addresses a crucial issue of the Minimax algorithm, i.e. complete expansion of the game tree resulting in exponential time complexity which expressed in the **Big O** notation [23] is $O(d^{\,b})$, where d is the maximal depth of the game tree and b is the breadth at each state. In the best case, Alpha-Beta enables to reduce the computational cost to

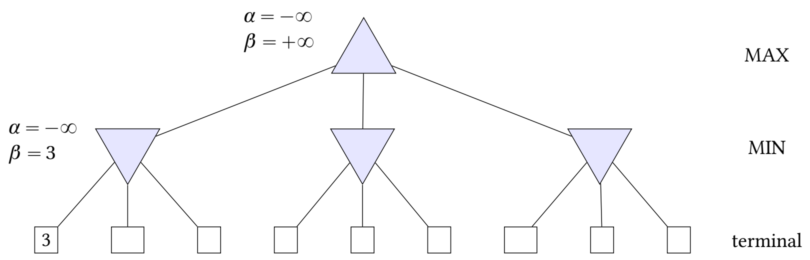
$O(d^{b/2})$ [52] by excluding from consideration subtrees that do not influence the selection of the best node in higher levels of the tree.

The above approach is employed in a process known as *alpha-beta pruning*. To implement it, each non-terminal node is assigned the two following variables [52]:

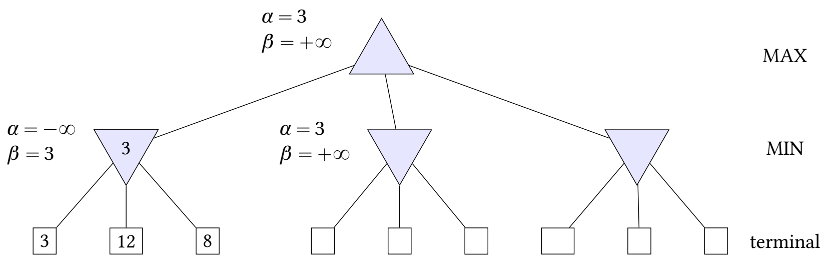
- α - the highest reward the MAX player is bound to achieve in the explored subtree.
- β - the lowest reward the MIN player is bound to achieve in the explored subtree.

The values are initialised to $\alpha = -\infty$ and $\beta = +\infty$, child nodes inherit the α-β values of their parent nodes. Once a improvement to a variable is found, the corresponding variable in the parent node is updated. If a child of a certain node is chosen such that $\alpha \geqslant \beta$, the selected child is pruned and is no longer considered (see Figure 2.5). Otherwise, the algorithm follows the standard Minimax procedure.

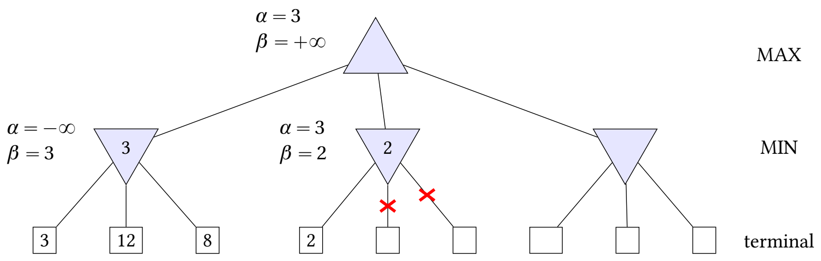
Despite the use of alpha-beta pruning resolving computational expense issues of the naive Minimax, the algorithm continued to be insufficiently fast to be employed in a real-time Chess engine. However, combined with a powerful heuristic used to evaluate nodes in the early and middle stage of the game, Deep Blue was capable of defeating a human Chess grandmaster Garry Kasparov in 1997 [12]. The achievement sparked great interest in the field as it marked the first victory over a world champion by a computer AI under tournament conditions.



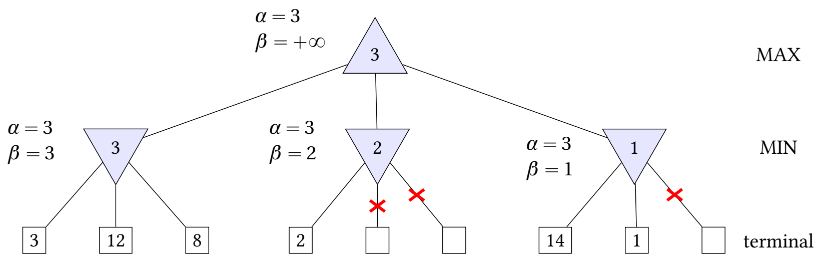
(a) The Alpha-Beta algorithm is based on the Minimax, hence, it traverses the tree in the depth-first order. Firstly, node 3 is visited and corresponding α and β values are updated in the parent node.



(b) The remaining children of the left-most MIN node do not improve the MIN value β, hence, the minimax value is updated to 3. The parent is a MAX node, therefore, the MAX variable α is updated to 3. The search of the next MIN child is initialised with α and β values of the parent.



(c) The first leaf of the second MIN node is 2 so update $\beta = 2$. It can be observed, that regardless of remaining leaves, the MIN node will have a value of at most 2, hence, it will never be chosen over the previous MIN node equal to 3. This is further reflected in the α-β values: $\alpha \geqslant \beta$, hence, remaining terminal nodes of this MIN node can be pruned. The minimax value is 2, but it provides no improvement to the parent node so its α-β values are not updated.



(d) Similarly to 2.5c, search of the last MIN node is initialised with parent α-β values. Discover subsequent terminal children until 1 is explored and $\alpha \geqslant \beta$, so the last branch can be pruned. The node has minimax value of 1, but it does not improve current α of the parent MAX node. The final minimax decision is again the left-most node.

Figure 2.5: An example of a Alpha-Beta Tree Search run on a 2-ply deep game tree.

2.5 Monte Carlo Tree Search

The most notable challenge posed by the games of perfect information is the extensive search of terminal states. Domains such as Chess and Go have high enough branching that pondering all outcomes is impossible. This cost can be reduced by adopting pruning techniques, for example alpha-beta pruning (see section 2.4.2) which preserves optimality while disregarding unaffecting search branches, however, to be applied to a real-time game depend upon an evaluation function. To maintain the generality of the algorithm, or in case of a lack of an effective heuristic, statistical approaches may be considered, such as the Monte Carlo Tree Search.

The Monte Carlo (MC) methods originate from the problem of solving intractable integrals in statistical physics by approximation. Although the approach has been widely applied in reinforcement learning [57], using it in the context of adversarial search is comparably new. The MC method has been extensively be adopted to approximate the desirability a move as described by Gelly and Silver (2011) [26]. Each iteration of the algorithm, known as a playout or rollout, chooses a node from the root (representing current state of the game) and simulating play until a leaf node is reached (see Figure 2.6). Outcomes of these simulations can be aggregated into the expected reward $Q(s,a)$ that selecting the action a from the state s yields [11]:

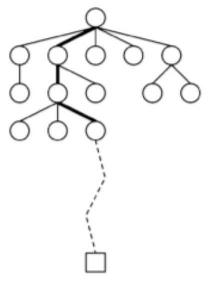


Figure 2.6: Simplified Monte Carlo rollout [7].

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a) z_i$$

$$\mathbb{I}_i(s,a) = \begin{cases} 0 & \text{if } a \text{ was selected from } s \text{ on the } i\text{th playout} \\ 1 & \text{otherwise} \end{cases}$$

where $N(s,a)$ is the number of times a has been expanded from s, $N(s)$ is the number of times s has been selected in a playout path, and z_i is the result of the ith rollout from s. Once the last iteration is finished, an action with the highest Q-value is selected as the next best move by the player.

The most notable advantages arising from the use of MC simulations for approximation are:

- Lack of the need to incorporate any domain-specific knowledge other than that required to determine legal actions from game states (i.e. game rules).
- Iterative search tree expansion allows for early termination of the algorithm, which makes it more suitable for real-time systems and complex domains where exploration of the entire game tree is intractable.
- Due to random nature of the simulations, the systematic error is reduces, which may prevent the opposing agents from exploiting repetitive behaviour and suboptimal evaluation functions.
- Relaxation of the optimal opponent assumption. The search favours states with higher expected rewards and more available wining strategies. Hence, it enables the agent to account for imperfect opponents and be more flexible in terms of its strategy.

The Monte Carlo Tree Search (MCTS) applies the Monte Carlo method to game tree search and aims to approximate the true game-theoretic value of the tree. In its basic implementation, it operates by utilising random sampling and iteratively building a game tree on a best-first basis. Four main stages of a single iteration of the algorithm (see Figure 2.7) can be outlined as follows:

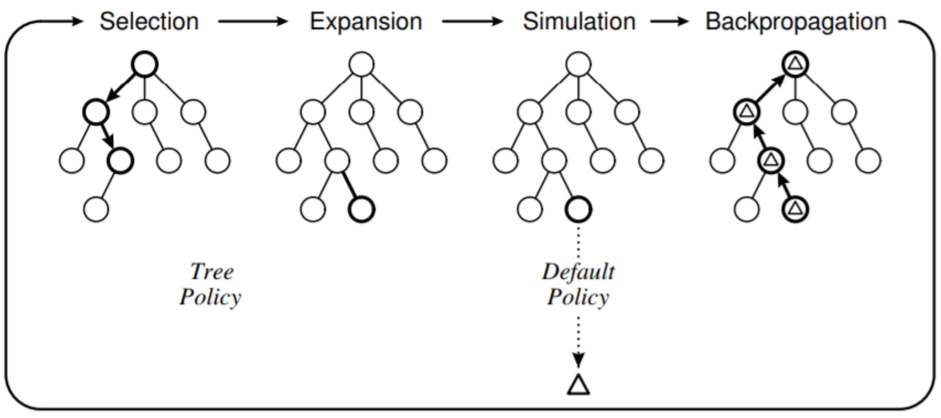


Figure 2.7: A single iteration of the MCTS algorithm in its base implementation [11].

1. **Selection**. Recursive application of a *child selection policy* chooses subsequent states to descend through the search tree. The process halts once the most urgent expandable node is reached, i.e. a node that is non-terminal and has unvisited (unexpanded) child nodes is selected.

2. **Expansion**. The selected node is extended by taking a legal action from that state to generate a new child node to grow the search tree. Alternatively, multiple children may be added by expanding several actions.

3. **Simulation**. Starting at the newly expanded node(s), transition to subsequent states is invoked according to a *default policy* until a terminal node is reached and an outcome is produced. In Chess the possible rewards are: 1 for a win, 0.5 for a draw, and a 0 for a loss. However, the payoffs need not sum to zero. Moreover, if multiple agents interact in the environment,rewards can be represented as vectors.

4. **Backpropagation**. The reward(s) is backed up (see section 2.4.1) through the nodes chosen in the selection phase to update their statistics. The backpropagated outcomes are used to inform the future selection (1) and expansion (2) decisions.

The search terminates once an interrupt condition is reached, for example exceeding the computation budget or maximal number of iterations, and a legal action from the root node is selected according to the chosen policy. As presented by Schadd et al. [53], one or a combination of the four criteria for selecting the immediate action can be utilised as such a policy:

- **Max child**. Select the child with the highest reward.
- **Robust child**. Select the child with highest number of visits.
- **Max-Robust child**. Combines policies (1) and (2) by selecting the root child with both the highest visit count and the highest reward. If none meets the condition, resume search until the max child exceeds the minimal visit count.
- **Secure child**. Select the child which maximises a lower confidence bound.

Algorithm 1 The Monte Carlo Tree Search algorithm with UCT selection policy

```
procedure UCT-SEARCH($s_0$)
    create root node $v_0$ with state $s_0$
    while interrupt condition not reached do
        $v_l \leftarrow$ TREEPOLICY($v_0$)
        $\Delta \leftarrow$ DEFAULTPOLICY($v_l$. STATE)
        BACKUP($v_0, \Delta$)
    return BESTCHILD($v_0, 0$) . ACTION

procedure TREEPOLICY($v$)
    while $v$ is nonterminal do
        if $v$ is expandable then
            return EXPAND($v$)
        else
            $v \leftarrow$ BESTCHILD($v, Cp$)
    return $v$

procedure EXPAND($v$)
    $a \leftarrow$ unexpanded legal action from $v$. STATE
    add $a$ new child $v'$ to $v$ with
        $v'$. STATE $= f(v$. STATE, $a)$
        $v'$. ACTION $= a$
    return $v'$

procedure BESTCHILD($v, c$)
    return $argmin\ \bar{X}_j + C\sqrt{\frac{2\ln(n)}{n_j}}$

procedure DEFAULTPOLICY($s$)
    while $s$ is non-terminal do
        choose $a \in A(s)$ randomly
        $s \leftarrow f(s, a)$
    return reward for state $s$

procedure BACKUP($v, \Delta$)
    while $v$ is not null do
        $N(v) \leftarrow N(v) + 1$
        $Q(v) \leftarrow Q(v) + \Delta(v, p)$
        $v \leftarrow$ parent of $v$
```

Utilising naive uniform random sampling, also known as *flat Monte Carlo*, may result in time wasted on exploring subtrees that produce little improvement in large search spaces. Therefore, the challenge is to determine an efficient selection policy which ensures that the Monte Carlo simulations has an optimal performance. Such a strategy must balance the exploration of suboptimal subtrees with the selection of nodes that are currently believed to be optimal. This dilemma is known as the *exploration-exploitation trade-off* and is often modelled using the *Multi-Armed Bandit* (MAB) problem.

The MAB problem is a popular class of sequential decision problems which aims to devise a strategy that maximises the cumulative reward of playing k arms of a multi-armed bandit slot machine. More specifically, the challenge is to determine a sequence of arms that will yield the highest overall payoff without any prior knowledge of the reward probability distribution among the arms. Once interrupted, the selection policy decides whether exploit a currently promising arm or explore another based on the previous outcomes (history).

To gain intuition in tackling the MAB problem, the notion of a *regret* is introduced:

$$R_N = \mu^* n - \mu_j \sum_{j=1}^{K} \mathbb{E}\left[T_j(n)\right] \tag{2.4}$$

where μ^* is the best possible expected reward and $\mathbb{E}\left[T_j(n)\right]$ denotes the expected number of plays for arm j in the first n trials [11]. Therefore, the regret associated with playing an arm N is just the expected loss from not playing the best one.

Furthermore, it has been proven by Lai and Robbins (1985) [33] that the selection policy must yield a regret growth rate at least as fast as $O(\ln(n))$ for a large class of reward distributions. Subsequently, a simple and efficient *Upper Confidence Bound* (UCB) strategy, known as *UCB1* [5], has an expected logarithmic growth of regret uniformly distributed over n. The policy chooses to play an arm j such that:

$$\text{UCB1} = \overline{X_j} + \sqrt{\frac{2\ln(n)}{n_j}} \tag{2.5}$$

where $\overline{X_j}$ is the average reward from playing j, n_j is the number of times j was played and n is the number of all plays so far. The two terms of the addition provide exploitation-exploration balance: element $\overline{X_j}$ promotes selection of high-payoff arms while element $\sqrt{\frac{2\ln(n)}{n_j}}$ encourages visiting less explored choices. This results in an important *restart* property of the policy, i.e. given sufficient resources even the low-payoff arms are bound to be chosen eventually.

The policy can be employed as a child selection in the Monte Carlo Tree Search, known as the *Upper Confidence Bound on Trees* (UCT) policy [20], where each child node is assigned an upper confidence bound of being the subsequent optimal game state (see Algorithm 1).

$$\mathrm{UCT} = \overline{X_j} + C\sqrt{\frac{2\ln(n)}{n_j}} \tag{2.6}$$

where to the UCB1 formula an additional constant C, also referred to as the *exploration term*, is added to adjust the importance of exploration by the policy. The UCT score is the highest for a node such that 1) it yields a high average of estimated values in the simulations that have been conducted through that node, and 2) has been infrequently visited. Such tree selection policy results in favouring nodes that the algorithm believes to be promising or important, hence, building an *asymmetric* search tree (see Figure 2.8).

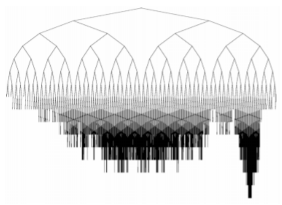


Figure 2.8: Example of an asymmetric search tree generated by a variation of MCTS (BAST) [21]. The search is more sparse in areas that were deemed less promising and, hence, unimportant.

A property of the MCTS with UCT (also referred to as just the *UCT*) that is of paramount importance is the *convergence to Minimax*. Since the Minimax algorithm was demonstrated to be optimal (see section 2.4.1), above characteristic implies that the method itself is bound to provide the optimal strategy. This MCTS property was shown by Kocsis and Szepesvari (2006) [32] by proving that given sufficient resources (memory and time) the probability of choosing a suboptimal action at the root of the search tree converges to zero.

2.6 Search Traps in Monte Carlo Tree Search

Although the UCT outperforms state-of-the-art algorithms such as the Alpha-Beta search in environments with high branching factor and in other game-playing scenarios, the method's performance deteriorates severely in domains with numerous *search traps* [47]. The issue arises from the nature of UCT, i.e. selection of the most promising nodes, which results in building an asymmetric search tree (see section 2.5) generated. Despite improving the overall performance, the insufficient exploration of suboptimal paths may lead to choosing subtrees that inevitably lead to a disadvantage or a loss (see Figure 2.9).

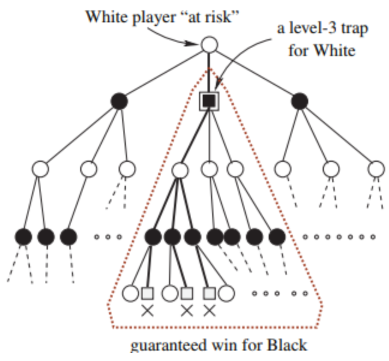


Figure 2.9: An example of a level-3 trap. Player 'White' is at risk, since executing action leading to the middle subtree, player 'Black' has a 3-move winning strategy. Grey square boxes represent terminal states where player 'Black' wins [47].

To deepen the understanding of the trap problem in the context of the Monte Carlo Tree Search, the following terminology is introduced [47]:

- **K-move Winning Strategy**. A sequence of k moves that results in the given player's victory and there does not exist a counter-play for the opponent to prevent it.
- **Trap Risk**. A player it at risk if there exits an action from a given state that results in the opponent having a k-move winning strategy.
- **Level-k Search Trap**. Occurs when a player at risk executes an action that leads to a state resulting in the opponent having a k-winning strategy.

It is important to distinguish between the problem of *hard traps*, which are discussed in this project, and *soft traps*, which are not in its scope. The first regard a subset of search traps that lead to a loss for the player, while the latter result in a non-terminal node and considerable disadvantage in terms of a game-theoretic value. In general, the detection and avoidance of soft traps by the UCT is more difficult than that of hard traps.

Another characteristic of a trap is its depth. A *shallow trap* (also referred to as an *early loss*) is a search trap up to level 7, whereas, a *deep trap* is a trap of level greater than 7 plies. Due to the difficulty in finding deep traps and the low probability of an opponent abusing it, the project concentrates on more dangerous shallow traps. An example of such a shallow trap in Chess known as the *Légal Trap* is presented in Figure 2.10.

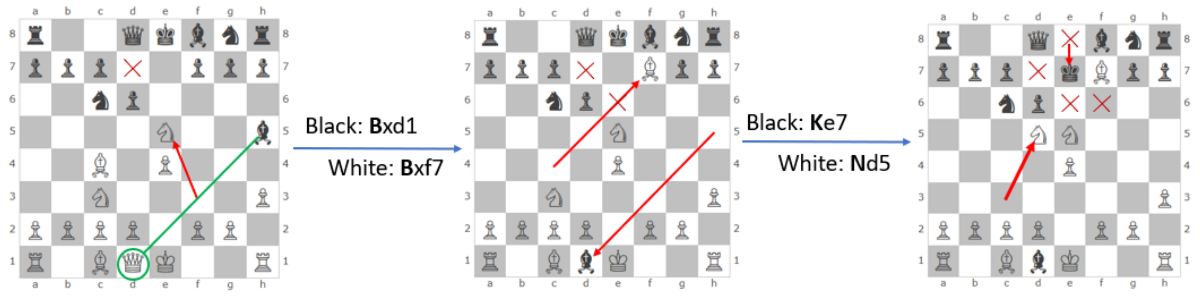


Figure 2.10: An example of a trap state in Chess. The red crosses indicate squares which the Black king cannot occupy. The left-most board appears to grant the Black player an advantage by capturing the White's queen. However, the position can actually be proven to be a trap state because the White agent is able to execute a 3-move winning strategy: Bxf7, Ke7 (Black must take this action) and Nd5.

On of the most notable examples of a game that involves a large number of shallow traps in its search space is Chess. As presented in research conducted by Ramanujan et al. (2010), the presence of numerous shallow traps can in observed in other games, especially those with a winning condition which denotes a relatively narrow subset of the entire state description [47]. The paper further investigates the property in Chess by generating a set of semi-random games as well as real grandmaster games, and examining them for the existence of traps at any level. The observations (see Figure 2.11) conclude that in Chess not only are shallow traps frequent, but also present at all phases of the game.

In the context of MCTS, the term *shallow trap* can be restricted to search traps of depth between 3 and 7, due to the possibility of imposing a lower bound of 3, as the UCT is able to detect traps

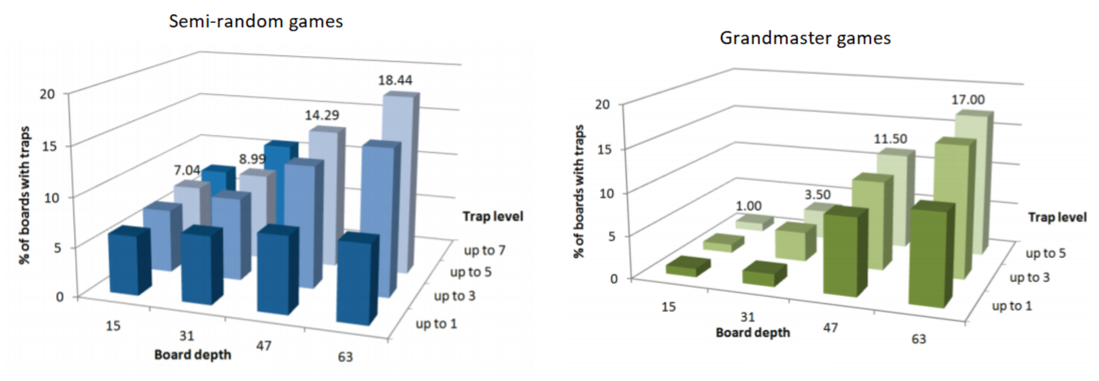


Figure 2.11: Plots that describe the percentage of boards (Y axis) containing shallow traps (Z axis) after a specific number of plies (X axis). Left subplot illustrates the results obtained for games that were generated using a composite of random move selection and GNU heuristic, and the right for actual Chess grandmaster games. It can be observed that the percentage of shallow traps increases from approximately 10% at 31-ply deep to 15% at 47-ply deep for semi-random games.

of depth 1 while it fails to recognise those deeper than 3 plies. This disadvantageous property prevails even if additional resource is supplied to the algorithm and conclude in the UCT assigning high value estimates to trap states (see Figure 2.12). Additionally, it was discovered that the UCT spends 70% of the time visiting states deeper than 5 plies in the presence of a level-5 trap, and 95% of the time exploring 7 plies for level-7 traps [47]. In other words, the UCT tree growth takes place beyond trap levels, which implies near complete inability of their detection by the algorithm. Hence, the research suggests low performance of the MCTS without an efficient trap detection and avoidance mechanism.

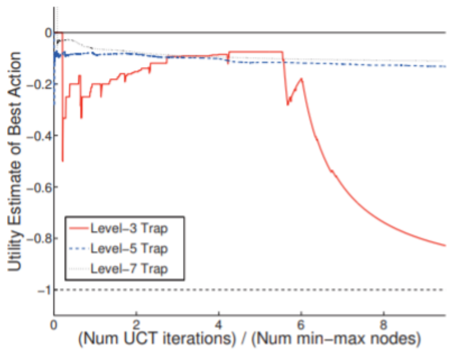


Figure 2.12: Estimates of the game theoretic value plotted proportionally to the number of UCT iteration. Considered states are: (1) a level-3 trap (red solid line), (2) a level-5 trap (blue dashed line), and (3) a level-7 trap (black dotted line) [47]. Only after substantial sampling the level-3 search trap is detected (value converges to -1), whereas the remainder of values maintain high estimates.

2.7 Trap-Adaptive Monte Carlo Tree Search

A solution to the problem of level-3 hard search traps was proposed by Benjamin Chun How Aw in a paper 'Trap-Adaptive Monte Carlo Tree Search Algorithms' [6]. In his work, the author suggests a series of extensions to the original MCTS-UCT algorithm inspired by the Minimax-MCTS hybrids [8]. The modifications aim to successfully detect, verify and adapt to search traps. The modifications and performance of the *Trap Aware Monte Carlo Tree Search* (TA-MCTS) presented in this research are employed as the baseline for the project.

2.7.1 Trap Avoidance

The enhanced algorithm uses two extensions to the original description, each of which is designated to prevent the selection of subtrees containing search traps in different stages of the search tree exploration. Both of the methods build upon the idea of *Breadth First Search* (BFS), which is a tree traversal algorithm that iteratively explores all nodes at each depth level of the structure, hence, in a breadth-first order.

Firstly, the *Top-level Breadth First Search* (TL-BFS) is applied to the first two levels of the tree. Consecutively, the built-in ability of avoiding traps of depth 1 and 3 may be further expanded to account for deeper search traps. The expense of simulating all nodes expanded by TL-BFS can be reduced by utilising Alpha-Beta pruning (see section 2.4.2). The TL-BFS of depth 2 has been empirically deemed the most efficient configuration [6], therefore, allowing for a use of a simpler variation of pruning with alpha (MAX variable) as the sole variable determining the exclusion of a branch (see Figure 2.13).

The second technique adopted in search trap avoidance is the *Bottom-level Breadth First Search* (BL-BFS), which employs an analogous approach as the TL-BFS to the last two levels of the simulation. The search tree expansion is applied whenever a *proven win* or a *proven draw* is encountered as the outcome of the simulation.

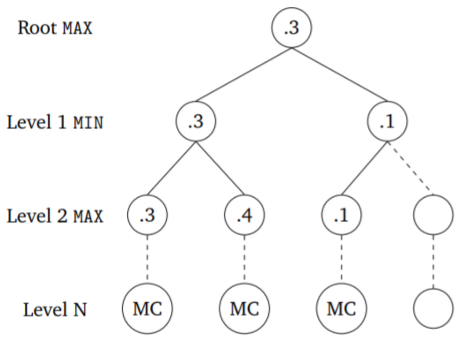


Figure 2.13: An example of TL-BFS (depth 2) with alpha pruning [6]. The right-most level-2 node of the search tree is pruned (no MCTS simulation performed for its children). Regardless of its values, this subtree may only further decrease the estimate of its parent (right-most level-1 node) equal to 0.1, which does not influence the minimax decision ($\alpha = 0.3$ and $\beta = 0.1$).

The concept of a states guaranteeing a specific game result was first described in context of the MCTS-Solver by Martin J. Osborne (2004) [63]. The paper proposes an MCTS variant capable of proving the game-theoretical values by backpropagating *proven wins* and *proven losses* [11]:

- **Proven loss**. A node which is either a terminal loss node or all its children nodes are terminal loss nodes.
- **Proven win**. A node which is either a terminal win node or at least one of its children nodes is a proven win.

Similarly to the TL-BFS, computational cost reduction is facilitated by Alpha-Beta pruning (see Figure 2.14). Another MCTS extension known as the *Decisive Moves* [58] guides the search during the rollout phase towards game states resulting in a guaranteed win. The depth of a *decisive move*, a proven win or a proven loss, that strikes the cost-efficiency balance was determined to be 2, therefore, the overall extent of the BL-BFS is likewise 2-ply deep [6].

2.7.2 Trap Identification

In an attempt to achieve the adaptiveness to encountered search traps, the TA-MCTS must further facilitate trap identification. By solely employing the trap prevention, the MCTS has no capability of reusing the gathered information and, hence, is unable to adapt to them. In his

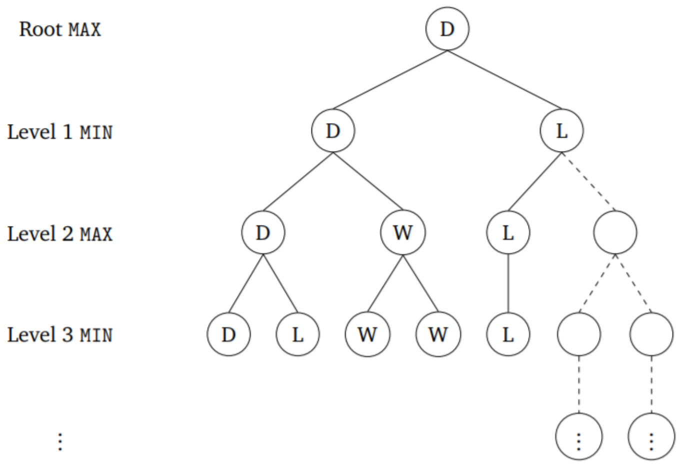


Figure 2.14: An example of BL-BFS (depth 2) with alpha-beta pruning [6]. Letters L, D, W stand for loss, draw and win correspondingly. Here, a level-2 node is a proven win, hence its grandparent becomes the root of the BL-BFS tree. The right-most level-2 node of the search tree is pruned, i.e. no further expansion of its children is carried out. Regardless of their values, the minimax decision at the root node will always prefer the level-1 left child over the level-1 right child.

work, Benjamin Chun How Aw focuses on 3-level traps that, from the definition of a k-level search trap, are states from which exists an action allowing the opponent to execute a 3-move winning strategy (see section 2.6). Given a list of legal actions, the investigation of a search tree for the above property is relatively inefficient to conduct using a BFS of depth 3. The detected traps are stored in dedicated data structure for a given retention period.

Should the trap not be detected or in the absence of an identification procedure, the BL-BFS or TL-BFS of depth 2 will assign low estimated values if the search enters a level-3 traps.

2.7.3 Trap Verification

Once the information on the identified level-3 traps is recorded, it may be used in consecutive turns during the selection phase of MCTS. A trap verification procedure is implemented, which determines whether any move on the current list of legal actions is present in the structure storing detected level-3 traps. Actions in the intersection of the two sets is analysed in terms of the level-3 trap still persisting. If true, a very low value is assigned at backpropagation stage effectively removing the move from consideration, otherwise, the action is removed from the trap history structure. The subroutine is repeated until a one of the termination conditions is reached:

- **Set of legal actions is empty**. Occurs when all legal moves are exhausted, hence, every extendible action was deemed to be a level-3 trap state. Since a loss is guaranteed regardless of the action selected, a random move is chosen.
- **Set of identified traps is empty**. All traps were successfully excluded from the search and the MCTS algorithm continues according to the default policy.

The time complexity of the MCTS (with UCT) is reduced thanks to propagating information on potential traps to grandchild nodes [6], however, the memory complexity is increased as additional space must be allocated for the trap history structure. Moreover, due to the constant verification of traps an additional computational expense is introduced. This project aims to resolve the outlined problem and improve the time complexity of the TA-MCTS with the use of positional similarity as a measure of the probability of a given trap persisting between two board states.

Chapter 3

Theory

3.1 Scope

The primary goal of the project is to develop a positional similarity measure that allows to reduce the cost of trap verification in the Trap-Adaptive Monte Carlo Tree Search (TA-MCTS). This approach has been suggested by B. C. H. Aw [6] but has not been sufficiently explored, resulting in a fairly naive trap verification procedure. Positional similarity, as an area of game theory, has not been a subject of research and this project aims define its underlying theory, examine possible measure and evaluate them in terms of efficiency in TA-MCTS.

A suitable method must be applicable as a probability measure of a trap persisting between two states of a game. Moreover, it must preserve the characteristics of the original Monte Carlo Tree Search algorithm that attribute to its popularity [11]:

- **Convergence to Minimax**. The MCTS with UCT has been proven to converge to the (optimal) Minimax decision at the tree root given sufficient resources.
- **Aheuristic**. The MCTS is applicable to any domain, provided that it can be represented as a tree, as the algorithm does not employ any environmental knowledge.
- **Anytime**. The algorithm is capable of returning a decision at any point of execution. The MCTS may also continue to sample the search space by completing subsequent iterations until termination condition is reached and leading to an improved outcome.

- **Asymmetric**. The tree selection policy favours regions of the search that MCTS deems to be more promising resulting in asymmetric search tree growth.

Developing an accurate and effective method addressing the issue will improve the overall performance of Trap-Adaptive MCTS and hopefully enable it to be considered a potential algorithm solving the GGP problem.

The aim of the project is to provide further enhancements the MCTS algorithm to improve its performance in domains with frequent search traps. The modified versions of MCTS incorporating different positional similarity measures will be then examined in contrast with the original implementation and themselves to evaluate their performance. The outcome of this analysis will lay the ground for project's conclusion. This chapter begins by describing the perimeters of the positional similarity theory, subsequently it proceeds to the formal definitions and theoretical characteristics of proposed probability measures.

3.2 Problem of Trap Verification Cost

To maintain the advantage over other algorithms that require entire game tree generation, such as the Minimax tree search, the Monte Carlo Tree Search simulates sequences of moves which leads to a significant decrease in the computational cost and therefore performance increase. The improvement is further achieved by guiding the search towards more desirable states using the UCT policy and adapting to the discovered trap states. However, this edge can be lost if Trap-Adaptive Monte Carlo Tree Search further requires state space expansion in order to verify or search for trap states - e.g. exploration using breadth-first search (BFS).

To mitigate this issue, both the resource usage and frequency of trap updates ought to be minimised. Dangerous states are expected to remain traps in the span of a move or several moves, hence, implementing a data structure for storing trap information (including the ply path to the state) can reduce the verification frequency and increase overall performance. Not only does this data decrease the time to validate the trap, but also enables prioritise most probable trap states resulting in improved resource allocation.

Storing this information allows the verification algorithm to be executed on a probabilistic basis, i.e. only when the game tree has changed significantly and there is a reason to believe that the previous trap states no longer persist.

3.3 Positional Similarity

Two chessboard states, obtained by taking a sequence of transformations from first board to the second board, can be compared using a positional similarity measure that examines the game trees generated by both configurations. In this project the focus is shifted towards the methods that enable to maintain the generality of General Game Playing agent. Hence, instead of directly comparing the Chess game boards, the game trees at the two board states ought to evaluated in terms of similarity instead.

A similarity measure can have many application in the context of Monte Carlo tree search, most notably, it can be utilised as the probability of the identified trap to persists between the first state and the latter. The above correlation is an implication of an assumption that *similar* game states yield comparable game trees, hence, probability of identical move sequences (e.g. search traps) to be present is higher in strategically equivalent states.

The remainder of this chapter focuses on attempting to define possible measures for establishing the *similarity*, i.e. degree of correlation between two strategical states. The similarity evaluations were attempted to have been adopted in TA-MCTS to backpropagate persistent traps. The additional application procedure introduced in this project is presented in the flowchart below (see Figure 3.1), however, the theory does not restrict the scope and may be adopted in a variety of game theoretical solutions.

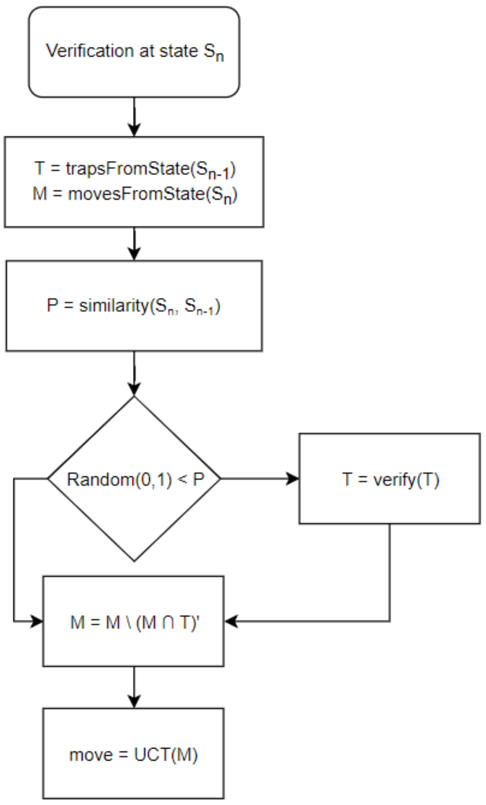

Verification at state S_n
T = trapsFromState(S_{n-1})
M = movesFromState(S_n)
P = similarity(S_n, S_{n-1})
Random(0,1) < P
T = verify(T)
M = M \ (M ∩ T)'
move = UCT(M)

Figure 3.1: The procedure describing the use of similarity measures in the scope of the project.

Apart from the maintenance of generality, a suitable similarity measure $S(A, B)$ between two boards A and B should have the following properties:

- **Symmetric**. The same value is returned by calculating $S(A, B)$ as $S(B, A)$
- **Normalised**. S(A,B) returns a real value k such that: $k \in [0, 1]$

The value reflects the likelihood of generating the same properties, hence, ought to be 0 for game trees that are completely different and 1 for boards that are equivalent. The above features will enable the similarity measures to be applied directly as a probability measure of a trap persisting between two states of the game.

3.4 Positional Similarity Measures

This section presents the theoretical framework for a number of positional similarity measures implemented in this project. Each method is introduced by defining a mathematical formula for calculating its value as well as an example of two game positions compared by employing the considered measure.

3.4.1 Constant Factor

A similarity measure can be approximated by specifying a constant real number in a range between 0 and 1. This approach is naive and uninformative of the actual similarity, but its main advantage is the simplicity and, more importantly, minimal computational expense. The constant factor similarity measure between game tree A and B can be defined as:

$$S(A,B) = k \quad \text{where} \quad k \in [0,1] \land k \in \mathbb{R} \tag{3.1}$$

This method was used in the previous implementation of a trap-aware MCC by B. C. H. Aw (2017) [6]. The measure may be further generalised to adapt to the frequency of trap persistence, which can could vary in depending on the domain, by employing an *On-line Parameter Tuning* approach. Although beyond the scope of this project, such methods were described and successfully adopted in tuning constants incorporated within the MCTS by C. Sironi and M. Winand (2017) [55].

3.4.2 Game-Influential Pieces

A game-influential piece can be defined as an object in the domain which change in presence results in significant alterations to the game tree, e.g. restricts the actions that a player can take. While it is easy to find such pieces in Chess or Checkers (every figure can be such a piece), the task is much harder in complex environments such as Wumpus World and even impossible for some non-deterministic games. The possible similarity for a set of game-influential pieces X can be calculate using the following formula:

$$S(A,B) = 1 - \frac{\sum_{x \in X} \mid g(A,x) - g(B,x) \mid}{\mid X \mid} \tag{3.2}$$

where $g(C,x)$ is a function determining the number of pieces of class x present in C. The class can also be a singleton, i.e. comprise of only a unique piece (e.g. Black's right bishop), hence, the outcome of $g(C,x)$ is binary for each such class.

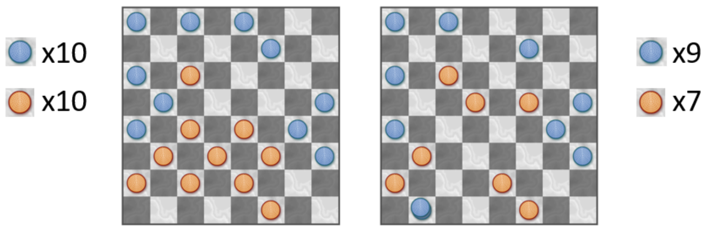


Figure 3.2: An example of the *Game-Influential Pieces* positional measure for two Checker board states. Assuming a class (colour) comparison instead of unique pieces, the final similarity value is: $1 - \frac{1+3}{10+10} = \frac{16}{20} = 0.8$.

The measure is easy to compute given that game-influential pieces can be defined in the game. On the other hand, the method requires additional information in conjunction with positions' game trees, hence, indicating the use of domain knowledge. Therefore, the method should not be considered as a solution to the project, however, it can be employed as a benchmark. Moreover, it is clear that in many domains the environment may undergo drastic changes despite few or no captures. This may imply poor performance especially in the early stages of a game like Chess where the majority of actions are non-capturing.

3.4.3 Depth and Breadth of the Game Tree

To determine the similarity of positions one can utilise the game trees that each of the states yield. Comparison of such representations can conducted by considering the trees' dimensions.

One shape of a game tree is its *depth* representing the number of plies that players can make from the given state to the leaf nodes. The depth is variant as it depends on the path to a final state, therefore, an aggregate d such as the maximum or the mean has to be used.

Another dimension is the *breadth* of a game tree, which value corresponds to the number states reachable from a given node. Its value can be either the breadth of the given node or, similarly to the depth, an aggregate b (not necessarily the same function as for the depth) over the breadth of the child nodes.

Either one of the dimensions can be selected or their combination may be used to determine the similarity of the two trees. The weighted composite is a generalisation to the case of a single dimension, i.e. weight of the depth similarity $w_d = 0$ or the breadth similarity $w_d = 0$, the formula for the method is the following:

$$S(A,B) = \frac{w_d\left(1 - \frac{|d(A)-d(B)|}{d(A)+d(B)}\right) + w_b\left(1 - \frac{|b(A)-b(B)|}{b(A)+b(B)}\right)}{w_d + w_b} \tag{3.3}$$

Unfortunately, MCTS rarely generates an entire game tree, so the depth must be approximated which may result in inaccuracy. This furthermore implies the method must be utilised post-simulation, hence, it is inapplicable in TA-MCTS. The issue can be mitigated by choosing the depth as the path length from the starting position instead. On the other hand, the cost of computing the breadth introduced additional overhead.

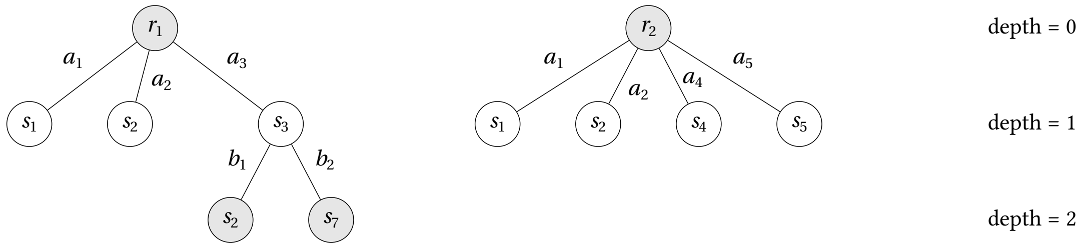


Figure 3.3: An example of *Depth and Breadth* similarity measure for two complete game trees with $w_d = 0.5$ and $w_b = 1$. The left tree has dimensions $\{d(A) = 2,\ b(A) = 3\}$ and the right tree $\{d(B) = 1,\ b(A) = 4\}$, therefore, their similarity is $S(A,B) \approx \frac{1.19}{1.5} \approx 0.79$.

3.4.4 Legal Actions

This approach applies the set theory to determine the similarity between two game tree, i.e. states of the board. Given two search tree roots A and B, denote a set of all the immediate (only expandable from the root itself) actions from these nodes correspondingly as $M(A)$ and $M(B)$. From the set theory it is known that:

$$M(A \cup B) = M(A) + M(B) \setminus M(A \cap B) \tag{3.4}$$

where $M(A \cup B)$ is the set of legal moves from node A or B (set union), and $M(A \cap B)$ is the set of actions that are expandable from A as well as from B (set intersection). The similarity defined through the sets of legal actions is denoted as:

$$S(A,B) = \frac{|M(A \cap B)|}{|M(A \cup B)|} = \frac{|M(A \cap B)|}{|M(A)| + |M(B)| \setminus |M(A \cap B)|} \tag{3.5}$$

This approach is more informative than the methods introduced in earlier subsections. Furthermore, it can be easily computed provided an appropriate infrastructure for the environment exists, e.g. Stockfish implements a function returning immediate actions from a given state.

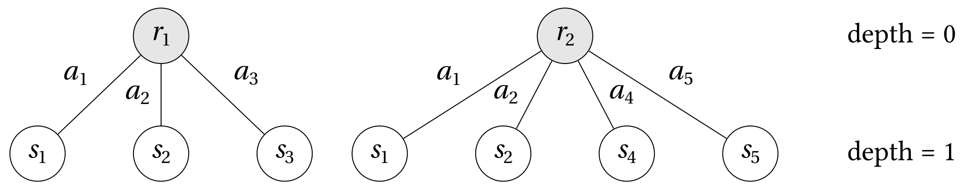


Figure 3.4: Illustration of the Legal Actions similarity measure functionality. The game trees with root nodes $\{r_1, r_2\}$ represent two distinct game states from which actions $\{a_1, ..., a_5\}$ result in states $\{s1, ..., s5\}$. The intersection of actions from the two roots is $\{a_1, a_2\}$, hence, the similarity given by the Legal Actions measure can be calculated as follows: $S(r_1, r_2) = \frac{|\{a_1,a_2\}|}{|\{a_1,a_2,a_3,a_4,a_5\}|} = \frac{2}{5} = 0.4$

3.4.5 Recursive Legal Actions

The *Recursive Legal Actions* method builds upon the *Legal Actions* measure introduced in 3.4.4 by further examining states that do not belong to the intersection of the legal actions of the two game states. The objective of such augmented analysis is determining the resemblance of the two non-identical actions, hence, requiring a similarity measure of their child nodes to be adopted (see Figure 3.5).

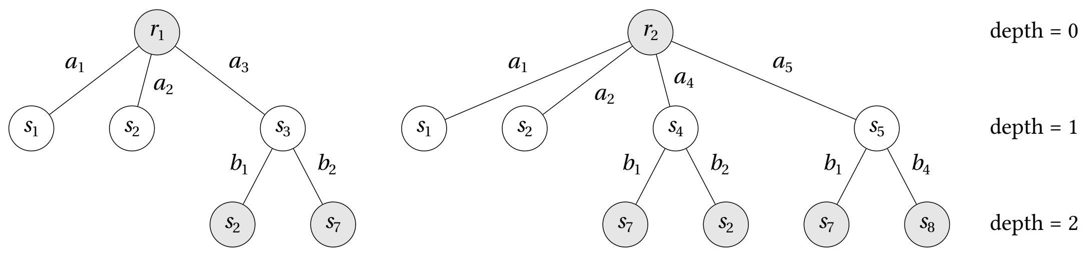


Figure 3.5: The procedure of *Recursive Legal Actions* described on two game trees (as described in 3.4). Nodes $\{s_3, s_5, s_6\}$ are further expanded to determine their similarity. Here *Legal Actions* is employed for children evaluation with depth $d = 1$ and similarity threshold $t = 0.75$. State s_3 is mapped to s_4 because $S(s_3, s_4) \geqslant t$ but not to s_4 as $S(s_3, s_5) < t$. The overall similarity is $S(r_1, r_2) = \frac{|\{a_1,a_2,a_3\}|}{|\{a_1,a_2,a_3,a_5\}|} = \frac{3}{4} = 0.75$

Investigated nodes are considered to be equivalent if they exceed a given *similarity threshold* and are mapped in a one-to-one fashion, i.e. a single node can only map to one vertex from another parent. The employed function compares two tree nodes (results of the actions) and may be implemented by any of the applicable measures discussed.

Furthermore, the measure can be utilised in recursive fashion, hence, ultimately leading to actions which transition to terminal states. Alternatively, the deepening could be interrupted after reaching a certain depth or exceeding a given *dissimilarity threshold.*
A possible variation could have the following formula:

$$S(A, B) = \frac{|\,M(A \cap B)\,| + |\,C\,|}{|\,M(A)\,| + |\,M(B)\,| \;-\; |\,M(A \cap B)\,|} \tag{3.6}$$

where C is the set of inequivalent moves satisfying the similarity condition determined by the Rec() function:

$$C = \textsc{Rec}\;(A \setminus B,\;\; B \setminus A) \tag{3.7}$$

3.4.6 Expandable States

Another measure introduced in the project is the *Expandable States* and the *Recursive Expandable States* measures. The formulas are identical to those presented in sections 3.4.4 and 3.4.5 correspondingly, however, instead of using the moves as the basis for comparison the subsequent state representations are utilised (see Figure 3.6).

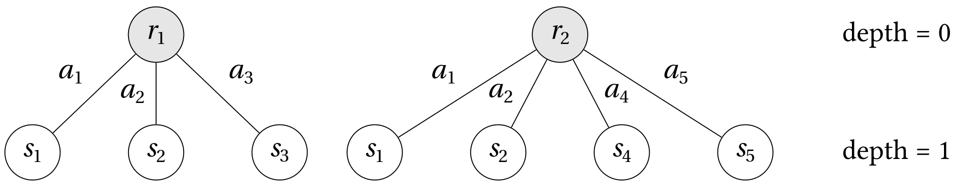


Figure 3.6: Illustration of the Expandable States similarity measure functionality. The game trees with root nodes $\{r_1, r_2\}$ represent two distinct game states from which actions $\{a_1, ..., a_5\}$ result in states $\{s1, ..., s5\}$. The similarity given by the measure is: $S(r_1, r_2) = \frac{|\{s_1, s_2\}|}{|\{s_1, s_2, s_3, s_4, s_5\}|} = \frac{2}{5} = 0.4$

An issue introduced by this measure is the state and environment encoding. Similarly to the *Game-Influential Pieces* method, it could be argued that a degree of domain-specific is required to represent the entire environment. However, assuming that the agent is supplied the set of rules as well as the actions its able to take at every moment, at least a partial image can be derived. Although Stockfish conveniently implements a function to produce a hashkey each board configuration, a non-unique mapping can be employed.

3.4.7 Statistical Desirability Evaluation

The MCTS operates by approximating values of states using sampling. Therefore, each expanded node in the generated game tree has an estimated desirability associated with it. The states for which expansion was performed in both board configurations can be compared assuming that the difference of these approximations is inversely proportional to their similarity:

$$S(A, B) = \eta \sum_{x \in M(A \cap B)} |\, h(A, x) - h(B, x) \,| \tag{3.8}$$

where $h(A, x)$ is the estimated by MCTS value of reaching state x in board A and η is the normalising factor.

Approach uses precomputed information supplied by the MCTS algorithm, hence, it is computationally inexpensive. Moreover, this method has the potential to be highly adequate in the setting of trap-awareness. An increase in the estimated value of a state that previously detected a trap might be an indicate of the trap no longer persisting; that is, the state space in the proximity of the trap has been sufficiently sampled and a positive outcome is more probable.

However, a non-trivial problem poses the normalising factor η, which ensures that the expression $S(A, B)$ evaluates to a value in the range $[0, 1]$, hence, suitable to use as a probability. To account for this, an approximation using a sigmoid function may be employed (see section 5.7).

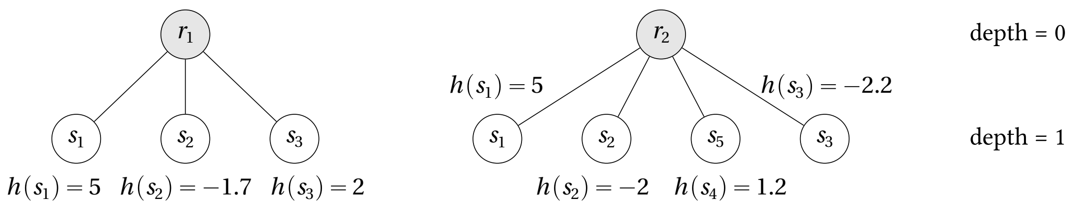


Figure 3.7: The equivalent states identified by their representation are $\{s_1, s_2, s_3\}$ and amount to a total difference of 0.5. Prior to using it as a probability, the value ought to be normalised.

Chapter 4

Implementation

This chapter begins by discussing the base implementation of the Stockfish Chess engine and will proceed to the Monte Carlo Chess (MCC) engine and its trap adaptive (TA-MCTS) variant. Emphasis will be placed on their relevant routines, applicable data structures and benefits accrued from expanding upon them. Subsequently, the design challenges and the implementation of proposed positional similarity measures will be examined. Finally, pseudocode of the most significant procedures will be presented.

4.1 Base Implementation

Stockfish is one of the strongest Chess agents constantly on the top of the rankings (based on data from Feb 2019 [13]). The engine is open-source and licensed under the GNU GPLv3 allowing for anyone to use and modify the software. Initially developed by T. Romstad, M. Costalba and J. Kiiski in 2008, the engine is currently maintained by the community [51].

Stockfish is implemented in C++ granting high degree of control and efficiency [24]. The engine owes its edge over the competitors to the following key features:

- **Bitboards**. A Bitboard is an efficient data structure representing a single chessboard as a set of 64 bits, i.e. one bit per every board square. Despite the need to maintain a Bitboard for each piece and both players to encapsulate the game state, the method benefits in the speed up of chessboard functions through CPU supported bitwise operations [61].

- **High search depth**. The engine owes its extensive search depth in part to employing an advanced Alpha-Beta algorithm with *aggressive pruning* [48] and *late move reductions* (LMR) [17]. As a consequence, the algorithm asymmetrically expands the game tree and avoids unpromising subtrees, which shifts focus to the most advantageous regions.
- **Static Databases**. An *opening book* containing a predetermined set of opening legal position sequences with their game theoretic values [34]. Instead of conducting an exhaustive search, a random move is selected from such a database, which provides speed up, improves variety and quality of play [18]. An analogous approach can be adopted for perfect endgame play by utilising *endgame tablebases* [37].

Unfortunately, most of the above solutions are inapplicable in the construction of a Monte Carlo Tree Search based Chess engine, as they introduce a high degree of game-specific information. Nonetheless, these methods can be adopted for analysis of the MCTS variant's performance. Moreover, the utilisation of structures such as Bitboards and other related objects or functionalities that preserve the generality of a GGP agent is highly convenient.

As previously mentioned, Stockfish provides useful data structures that are employed in the project. This includes the MoveStack class encapsulating a move and its associated score. Actions are wrapped in this structure upon automatic generation of different types of moves supported by Stockfish, e.g. all legal moves or only pawn moves. Another important object is the Position which represents the current board state as well as numerous derivative properties. The implementations of the similarity measures are heavily based on this structure, hence, it is discussed in greater detail in section 4.4.

The practicality of the use of a base implementation is further increased if extensive literature and documentation is available. Although an abundance of third-party and community research has been published since its release in 2008, the original Stockfish support website previously hosted on http://support.stockfishchess.org/ is no longer online. As minor as the issue appears, it may hinder the ability to precisely determine behaviour of complex methods and to troubleshoot implementation difficulties.

The original source code of the latest version is available at https://github.com/official-stockfish/Stockfish/ and all previous releases at https://www.dropbox.com/sh/75gzfgu7qo94pvh/NlXX-QLGu6.

4.2 Monte Carlo Chess Engine

The Monte Carlo Chess (MCC) engine is variant of the Stockfish engine which utilises the Monte Carlo Tree Search. It has been proposed and implemented by Oleg Arenz in 2010 [4]. In his thesis, the author suggests altering the call to the best move search subroutine in the uci.cpp. The default behaviour encoded in Stockfish, which performs an advanced Alpha-Beta search defined in search.cpp, is substituted by an *UCT-MCTS* search in uctsearch.cpp.

Upon the examination of the algorithm's pseudocode (see Alogrithm 2), a clear relation to the MCTS stages outlined in subsection 2.5 can be observed. The four phases are interleaved in steps 7-10 to expand the game tree according to the UCT policy until a termination condition is triggered by the UCTPoll() method on line 12.

The remainder of the components utilised by the UCT() function are defined as follows:

- **Limits** - an object describing the search parameters and limitations specified by the user or the a Chess IDE.
- **MonteCarloTreeNode**(Move, Position) - a MonteCarloTreeNode constructor called with a Move object representing the action leading to the node and its parent Position object. Initialisation involves setting default parameters, such as: parent node pointer, number of MCTS visits and the aggregate value of the simulation results.

- MonteCarloTreeNode::**select**(Position) - a method to select a node for expansion according to the UCT policy.
- MonteCarloTreeNode::**expand**(Position) - a function expanding the node selected by select() by generating a list of all legal moves from the given position and appending it to the MonteCarloTreeNode object as its children.
- MonteCarloTreeNode::**simulate**(Position) - a procedure performing an MCTS rollout by playing a sequence of random legal moves from the given position until a terminal node is reached.
- MonteCarloTreeNode::**update**(Double) - a method which backpropagates the given Double value (here: the outcome of the simulation from line 9, i.e. win, draw or loss) to the root node.

Once terminated, the method ought to return the optimal result. To achieve this, an approach commonly utilised in Q-learning [59] can be employed to find the *best* child. Such a method determines the final outcome to be an action which maximizes the expected total reward (Q-value) over the undertaken iterations:

$$\pi_{\text{select}}(s) = \underset{a}{\text{argmax}}\ Q(s,a) \tag{4.1}$$

However, instead of this max child implementation, a *robust* child (see section 2.5) selection policy finds the most frequently visited child node of the root state:

$$\pi_{\text{select}}(s) = \underset{a}{\text{argmax}}\ N(s,a) \tag{4.2}$$

Such a mechanism was employed in place of the highest Q-value due to the UCT favouring more promising nodes, hence, statistically expected to be visited by MCTS more regularly [14].

The outlined functionality is embedded in the MonteCarloTreeNode object as a bestChild() method. The function returns another MonteCarloTreeNode element which encodes the state resulting from the move chosen according the selection strategy shown in Equation 4.2.

Algorithm 2 The main routine of MCC performing UCT-MCTS search from a given position.

```
procedure UCT(POSITION rootPos, LIMITS searchLimits)
    StopRequest ← false
    startTime ← CURRENTTIME()
    thinkingTime ← searchLimits.TIME / timeRate
    root ← new MONTECARLOTREENODE (MOVE_NONE, NULL)

    while StopRequest = false do
        selected ← root.SELECT(rootPos)
        expanded ← selected.EXPAND(rootPos)
        result ← expanded.SIMULATE(rootPos)
        expanded.UPDATE(result)

        if (iterations++) %1000 = 0 then
            StopRequest ← UCTPOLL(startTime, thinkingTime, searchLimits)
    endWhile

    return root.BESTMOVE()

procedure UCTPOLL(startTime, thinkingTime, LIMITS searchLimits)
    if INPUTFROMUSER() == STOP_SIGNAL then
        return true
    else if CURRENTTIME() - startTime ⩾ thinkingTime then
        return true
    else
        return false
```

Algorithm 3 A procedure finding the best action from the current tree node after a series of MCTS iterations has terminated.

```
procedure BESTCHILD( )
    maxVisits ← -1
    for child in this.CHILDREN() do
        if child.VISITS < maxVisits then
            maxVisits ← child.VISITS
            bestChild ← child
    endFor
    return bestChild
```

4.3 Trap-Adaptive Monte Carlo Chess Engine

This section discusses the implementation of the *Trap Aware Monte Carlo Tree Search* Chess engine (TA-MCC) as proposed by Benjamin Chun How Aw. The engine is heavily based on the MCC engine (see section 4.2) as well as the base implementation of MCC - Stockfish (see section 4.1). Following subsections expand upon the theory introduced in section 2.7 by presenting algorithms utilised to improve trap-adaptiveness of the MCC.

4.3.1 Breadth First Search MCTS

The first extension of the TA-MCC involves improved trap avoidance mechanisms facilitated by the Top-level Breadth First Search (TL-BFS) and Bottom-level Breadth First Search (BL-BFS). The approaches are variations of the Minimax and MCTS hybrids [8], however, are utilised less aggressively by restricting the depth and application areas to the top and bottom of the tree.

To implement TL-BFS, the selection procedure is altered to choose each child node at least once. Algorithm 4 describes the modified method with lines 6 - 7 guaranteeing the selection of a node with unexplored children before picking the best one according to the UCT formula. This ensures that the expansion can be postponed one ply deeper in the game tree, however, iterative application of this exhaustive selection will further increase the depth of TL-BFS. An Alpha-Beta search is conducted on the expanded TL-BFS (see section 2.7.1) with MCTS simulation as the evaluation function for each leaf node of the BFS tree.

The BL-BFS implementation requires adjustments to two phases of the MCTS. Firstly, the UPDATE() function which is modified to not only update the MCTS rollout results but also proven wins and proven losses. To implement the theory outlined in 2.7.1, additional properties of the MONTECARLOTREENODE object must be assigned to facilitate the propagation of these values alike the MCTS-Solver [63]. The enhanced UDPATE() method is presented in Algorithm 7.

Algorithm 4 Modified selection phase used in TA-MCC based on the method implemented in the MCC.

```
procedure SELECT(POSITION rootPos, LIMITS searchLimits)
    currentPos ← GETCURRENTPOSITION(rootPos)
    sideToMove ← currentPos.SIDETOMOVE()
    bestVal ← −1

    while cur.CHILDRENSIZE > 0 do
        if cur.CHILDRENSIZE < cur.MAXMOVES then
            return cur
        cur ← this.BESTCHILD()
        sideToMove ← not sideToMove
    endWhile
    return cur
```

The second improvement required to facilitate the BL-BFS involves an adjustment to the simulation phase of the MCTS and aims to guide the search towards nodes producing a guaranteed win. The modified SIMULATION() function ensures this property by overriding the random selection and choosing a proven outcomes instead. A function determining whether such an action exists from a given postion is represented as pseudocode in Algorithm 5.

4.3.2 Trap Adaptivness

The latter enhancement to the MCC engine that was implemented allows for trap identification. To achieve this, an exhaustive search of depth 3 is conducted to find any grandchild node that contains a decisive move, i.e. the root node is a level-3 search trap (see Algorithm 6). This procedure is known as **trap generation** and results in a list (here, a vector) of moves leading that enter a 3-move winning strategy for the opponent.

Algorithm 5 A procedure used in the simulation phase of TA-MCC for determining if a proven win or loss is extendible from a list of legal moves of a given position.

```
procedure checkDecisiveMove ( Position pos, vector<MoveStack> mlist )
    for ms in mlist do
        childPos ← pos.do_move(ms.move)
        if childPos.is_mate() then
            return true
    endFor
    return false
```

Algorithm 6 Procedure generate<Trap> which generates a list of search traps from the current position and trapCheck that determines if a given move leads to a 3-move winning strategy.

```
procedure trapGen ( Position pos, vector<MoveStack> mlist )
    tlist ← mlist
    for ms in tlist do
        childPos ← pos.do_move(ms.move)
        if not childPos.is_trap() then
            tlist.remove(ms)
    endFor
    return tlist

procedure is_trap()
    mlist ← generate<MV_LEGAL>(this)
    for ms in mlist do
        childPos ← pos.do_move(ms.move)
        if childPos.is_mate() then
            return true
    endFor
    return false
```

Algorithm 7 Modified MCC backpropagation phase implementation used in TA-MCC.

```
 1: procedure updateDecisive ( Double score,  Boolean playerToMove )
 2:     if winFor(playerToMove) then
 3:         this.totalValue ← score
 4:     if this.parent == NULL then
 5:         return
 6:     if winFor(not playerToMove) then
 7:         this.parent.updateDecisive(score, not playerToMove)

 8:     parentKnownLoss ← true
 9:     farthestLoss ← 0
10:     if winFor(BLACK) then
11:         farthestLoss ← -MAX_VALUE
12:     if winFor(WHITE) then
13:         farthestLoss ← MAX_VALUE

14:     for child in this.children do
15:         if not provenWin(child) then
16:             parentKnownLoss ← false
17:             goto step 21
18:         else if child.totalValue < farthestLoss then
19:             farthestLoss ← totalValue
20:     endFor

21:     if parentKnownLoss then
22:         if playerToMove == WHITE then
23:             this.parent.updateDecisive(farthestLoss+1, false)
24:         else
25:             this.parent.updateDecisive(farthestLoss-1, true)
26:     else if winFor(WHITE) then
27:         this.parent.updateNormal(0)
28:     else if winFor(BLACK) then
29:         this.parent.updateNormal(1)

30: procedure updateNormal(Double score)
31:     if not (winFor(WHITE) or winFor(BLACK)) then
32:         this.totalValue = score

33:     this.visits += 1
34:     if this.parent != NULL then
35:         this.parent.updateNormal(score)
```

4.4 Positional Similarity

According to the earlier defined theoretical framework, the identified traps are subsequently maintained through to the next turn and verified only on a probabilistic basis. Pseudocode in Algorithm 8 shows the modified UCT() function with lines 1 - 2 storing the stateful data and probabilistic updates at 12 and 17. This way, a statistical time complexity reduction of TA-MCTS is achieved and the negative impact of the constant trap generation is mitigated. Once the trap list is obtained from either the previous turn or a new verification, actions that are contained within it can be pruned from consideration during the selection phase.

Algorithm 8 The updated main routine of MCC performing UCT-MCTS.

```
 1: POSITION prevPosBlanc, prevPosNoir
 2: VECTOR<MOVESTACK> tlistBlanc, tlistNoir

 3: procedure UCT(POSITION rootPos, LIMITS searchLimits)
 4:     StopRequest ← false
 5:     startTime ← CURRENTTIME()
 6:     thinkingTime ← searchLimits.TIME / timeRate
 7:     root ← new MONTECARLOTREENODE (MOVE_NONE, NULL)

 8:     simMethod ← *selected similarity measure*
 9:     if rootPos.SIDE_TO_MOVE() == WHITE then
10:         sim ← SIMILARITY<simMethod>(rootPos, prevPosBlanc)
11:         prevPosBlanc ← rootPos

12:         if tlistBlanc.EMPTY() or random(0,1) > sim then
13:             tlistBlanc ← GENERATE<TRAP>(pos)
14:     else
15:         sim ← SIMILARITY<simMethod>(rootPos, prevPosNoir)
16:         prevPosNoir ← rootPos

17:         if tlistNoir.EMPTY() or random(0,1) > sim then
18:             tlistNoir ← GENERATE<TRAP>(pos)

19:     while StopRequest = false do
20:         ...
```

In order to fully grasp the feature space that enables for the comparison of position, the POSITION object implemented by Stockfish must be first examined. The class is defined in a header file position.h and the key attributes are:

- POSITION(STRING fen, BOOLEAN isChess960, INTEGER threadID) - a constructor that allows to create a position object from a string encoded FEN game state (see section 5.2).
- BITBOARD empty_squares(), pieces(PIECETYPE pt, COLOR c), ... - a number of methods returning a BITBOARD data structure describing squares matching the selected query.
- INTEGER piece_count(COLOR c, PIECETYPE pt) - method which allows to find the number of pieces of a specified colour and type (king, pawn, etc.).
- BOOLEAN is_mate(), is_trap(), ... - returns TRUE if the query is met, FALSE otherwise.

Numerous methods and properties constitute the POSITION object, however, the functionality presented above is sufficient for implementing the similarity measures considered in this thesis.

To calculate the similarity an abstract method is defined as follows:

$$\textsc{similarity} < \textsc{SimMethod} > (\textsc{Position}\ \text{prev}, \textsc{Position}\ \text{cur}) \tag{4.3}$$

where SIMMETHOD is a enumumaration **enum** that specifies which of the implemented similarity measures to employ to evaluate the positional similarity:

- CONSTANT - Constant Factor measure (see subsection 4.4.1),
- INFL_PIECES - Game-Influential Pieces measure (see subsection 4.4.2),
- DEPTH_BREADTH - Depth and Breadth measure (see subsection 4.4.3),
- LEGAL_ACTIONS - Legal Actions measure (see subsection 4.4.4),
- REC_LEGAL_ACTIONS - Recursive Legal Actions measure (see subsection 4.4.5),
- EXPANDABLE_STATES - Expandable States measure (see subsection 4.4.6),
- REC_EXPANDABLE_STATES - Recursive Expandable States measure (see subsec. 4.4.7).

The function compares two Position objects which must store the game state prior to move selection by the same player. Therefore, the previous position representation of the agent ought to be maintained between subsequent MCTS iterations. The updated UCT() method is displayed in Algorithm 8. Once obtained, the similarity measure is stored in a local variable and can be employed in an arbitrary manner.

4.4.1 Constant Factor

As introduced in section 3.4.1, this positional similarity measure simply returns a predefined constant value DEFAULT_SIM. In this project, this value is static and equal to 0.5. The pseudocode for the method's implementation is presented in Algorithm 9.

Algorithm 9 A method that implements the Constant Factor positional comparison measure.

```
1: procedure SIMILARITY<CONSTANT_FACTOR> (Position pos1, Position pos2)
2:     return DEFAULT_SIM
```

4.4.2 Game-Influential Pieces

The variant of the Game-Influential Pieces selected in this project compares types of pieces as opposed to unique pieces, i.e. any two pieces of the same type (pawn, king, etc.) are indistinguishable. Moreover, it was deemed beneficial to account not only for the figures of a single agent but rather compare the pieces of both players (see Algorithm 10).

Algorithm 10 A method that implements the Constant Factor positional comparison measure.

```
1: procedure SIMILARITY<INFL_PIECES> (Position pos1, Position pos2)
2:     union, intersect ← 0
3:     for pt in PieceType do
4:         pos1Blanc ← pos1.PIECE_COUNT(WHITE, pt)
5:         pos2Blanc ← pos2.PIECE_COUNT(WHITE, pt)
6:         pos1Noir ← pos1.PIECE_COUNT(BLACK, pt)
7:         pos2Noir ← pos2.PIECE_COUNT(BLACK, pt)
8:         union += MAX(pos1Blanc, pos2Blanc) + MAX(pos1Noir, pos2Noir)
9:         intersec += MIN(pos1Blanc, pos2Blanc) + MIN(pos1Noir, pos2Noir)
10:    endFor
11:    return intersec / union
```

4.4.3 Depth and Breadth

For this measure, a number of configurations can be adopted, namely, different aggregation functions for each dimension can be chosen. For the purpose of this project, the depth component is equivalent to the number of plies that were played until each game state. The breadth factor consists of the count of actions that can be taken in the first ply from the given positions.

Every shape component has an associated weight which values can be either constant or adaptive [55]. To maintain the simplicity of the project and computational speed, the former was applied for both dimensions:

- **Depth** $w_d = 0.7$
- **Breadth** $w_b = 0.6$

Algorithm 11 A method that implements the Constant Factor positional comparison measure.

```
procedure SIMILARITY<DEPTH_BREADTH> (POSITION pos1, POSITION pos2)
    pos1_d ← pos1.STARTPOS_PLY_COUNTER()
    pos2_d ← pos2.STARTPOS_PLY_COUNTER()
    if pos1_d == 0 and pos2_d == 0 then
        d ← 1
    else
        d ← 1 - $\frac{|\text{pos1\_d - pos2\_d}|}{\text{pos1\_d + pos2\_d}}$

    pos1_b ← GENERATE<MV_LEGAL>(pos1).SIZE()
    pos2_b ← GENERATE<MV_LEGAL>(pos2).SIZE()
    if pos1_b == 0 and pos2_b == 0 then
        b ← 1
    else
        b ← 1 - $\frac{|\text{pos1\_b - pos2\_b}|}{\text{pos1\_b + pos2\_b}}$

    return $\frac{w_d \times d + w_b \times b}{w_d + w_b}$
```

4.4.4 Legal Actions

The theory of Legal Actions measure was first proposed in the original Trap-Adaptive MCTS paper [6], however, the method was incorrectly implemented. The error involved iterating over a non-instantiated array of pointers, hence, randomly assigned memory addresses. Another issue was the comparison between two MoveStack objects that was implemented through pointer equivalence. Such a check may result in false negatives, as the outcome is true if and only if the pointers reference the same memory address, which clearly is not necessarily the case for all equivalent moves. Upon further analysis, not only were the bugs fixed, but the inefficiency of iterating the legal moves from both positions was also reduced.

The enhanced procedure that computes the similarity measure according to the proposed Legal Actions method is presented in Algorithm 12. The time complexity improvement was achieved by utilising the vector class for storing the actions. Such solution mitigates the need of pointer iteration to find the size of the union between to MoveStack sets by enabling a constant time check for the number of elements in the structure. Moreover, the pointer comparison was replaced with a less constrictive one which uses the Move property of the MoveStack object.

It is worth noting that upon finding a matching pair of MoveStack objects, the current element of the inner loop is removed from its list which avoids further unnecessary comparisons with an already matched action. After the deletion, the algorithm returns to the outer to search for a pair with the consecutive item from the first position. As a result, the union of the sets cannot be calculate from equation 3.4 prior to calculating the intersection as the size of the second set is decreased. Therefore, the cardinality of the union set is initially set to the sum of the individual sets' cardinalities (line 6) and subsequently the intersection size is subtracted to obtain the final length of the list (line 16).

Algorithm 12 A method that implements the Legal Actions positional comparison measure.

```
procedure SIMILARITY<LEGAL_ACTIONS> (POSITION pos1, POSITION pos2)
    pos1moves ← GENERATE<MV_LEGAL>(pos1)
    pos2moves ← GENERATE<MV_LEGAL>(pos2)

    if pos1moves.SIZE == 0 and pos2moves.SIZE == 0 then
        return 1

    union ← pos1moves.SIZE() + pos2moves.SIZE()
    intersec ← 0
    for ms1 in pos1moves do
        for ms2 in pos2moves do
            if ms1.MOVE == ms2.MOVE then
                intersec ← intersec + 1
                pos2moves.REMOVE(ms2)
                break
        endFor
    endFor

    union ← union - intersec
    return intersec / union
```

4.4.5 Recursive Legal Actions

For the measure to produce fewer false negatives and achieve a better inclusivity of positions, the Recursive Legal Actions measure is considered. The method builds upon the Legal Actions by applying the similarity evaluation to the states that are a result of mismatching moves.

Algorithm 13 A method that implements the Recursive Legal Actions similarity measure.

```
 1: procedure SIMILARITY<REC_LEGAL_ACTIONS> (POSITION pos1, POSITION pos2)
 2:     ...
 3:     for ms1 in pos1moves do
 4:         if pos1moves.SIZE() == 0 then
 5:             break
 6:         for ms2 in pos2moves do
 7:             if pos2moves.SIZE() == 0 then
 8:                 break
 9:             if ms1.MOVE == ms2.MOVE then
10:                 intersec ← intersec + 1
11:                 pos1moves.REMOVE(ms1)
12:                 pos2moves.REMOVE(ms2)
13:                 break
14:         endFor
15:     endFor

16:     mismatch ← INTERSECREC<LEGAL_MOVES>(pos1, pos2, pos1moves, pos2moves)
17:     union ← union - intersec
18:     return (intersec + mismatch) / union
```

Achieving this is possible by further expanding actions of such states and measuring their pairwise similarity which is implemented by the INTERSECREC<SIMMETHOD>() function presented in Algorithm 14. Here, the method applies a positional similarity measure of choice (line 16) with a depth limit equal to a single ply.

In this implementation, a match is defined as a positions pair that is either the same position or has a similarity above the similarity threshold SIM_THRESHOLD (line 11). Moreover, in this instance, an increment of similarity value was chosen (line 12) instead of a binary classification - 0 for a mismatch or 1 for a match. Accumulated value of these matches is then returned to the parent procedure as a floating-point number.

Algorithm 14 A method that calculates the number of similar positions that are a result of mismatching action passed from the layer above.

```
 1: procedure INTERSECREC<SIMMETHOD> (POSITION pos1, POSITION pos2, VECTOR<MOVESTACK> mismatch1, VECTOR<MOVESTACK> mismatch2)
 2:     intersec ← 0
 3:     for ms1 in mismatch1 do
 4:         for ms2 in mismatch2 do
 5:             pos1child ← pos1.DO_MOVE(ms1.MOVE)
 6:             pos2child ← pos2.DO_MOVE(ms2.MOVE)
 7:             if pos1child.GET_KEY() == pos2child.GET_KEY() then
 8:                 intersec ← intersec + 1
 9:                 break
10:             sim ← SIMILARITY<SIMMETHOD>(pos1child, pos2child)
11:             if sim > SIM_THRESHOLD then
12:                 intersec ← intersec + sim
13:                 break
14:         endFor
15:     endFor
16:     return intersec
```

4.4.6 Expandable States

Alike the Legal Actions measure, the Expandable States method also obtains the positional similarity by calculating the ratio of the set intersection to the set union (see Algorithm 12). However, instead of performing the evaluation for sets of actions, the sets of states are utilised. To find a states expandable from the current position, the transition function is employed, i.e. such a state is an outcome of playing an action from the current state (line 4, 6).

To compare two representation of states a hashing function POSITION::GET_KEY() is utilised. The method returns a key that uniquely identifies any board configurations in terms of piece location. Alternatively, the first segment of the FEN encoding for a given position could be employed in place of the above function.

The implementation of this method is displayed in Algorithm 15.

Algorithm 15 A method that implements the Expandable States positional comparison measure. The procedure is based on the Legal Actions measure presented in Algorithm 12.

```
procedure SIMILARITY<EXPANDABLE_STATES> (POSITION pos1, POSITION pos2)
    ...
    for ms1 in pos1moves do
        pos1child ← pos1.DO_MOVE(ms1.MOVE)
        for ms2 in pos2moves do
            pos2child ← pos2.DO_MOVE(ms2.MOVE)
            if pos1child.GET_KEY() == pos2child.GET_KEY() then
                intersec ← intersec + 1
                ms2.REMOVE(ms2)
                break
        endFor
    endFor

    union ← union - intersec
    return intersec / union
```

4.4.7 Recursive Expandable States

The Recursive Expandable States positional similarity measure is a combination of the Recursive Legal Actions (see subsection 4.4.5) and Expandable States (see subsection 4.4.6) methods. The major difference being line 18 where instead of deepening the evaluation using the Legal Actions similarity, the algorithm opts for the Expandable States measure. The implementation of this measure is presented in Algorithm 16.

4.4.8 Statistical Evaluation

As introduced in subsection 3.4.7, the Statistical Evaluation measure computes the difference of accumulated rollout scores for actions present in both given positions. One of the challenges posed by this measure is the need for precomputed simulation outcomes which renders the method inapplicable in the scope of the trap verification problem.

Nonetheless, the measure can be adopted in an auxiliary manner, hence, the psuedocode is presented in this project (see Algorithm 17).

Algorithm 16 A method that implements the Recursive Expandable States similarity measure.

```
procedure SIMILARITY<REC_EXPANDABLE_STATES> (POSITION pos1, POSITION pos2)
    ...
    for ms1 in pos1moves do
        if pos1moves.SIZE() == 0 then
            break
        pos1child ← pos1.DO_MOVE(ms1.MOVE)
        for ms2 in pos2moves do
            if pos2moves.SIZE() == 0 then
                break
            pos2child ← pos2.DO_MOVE(ms2.MOVE)
            if pos1child.GET_KEY() == pos2child.GET_KEY() then
                intersec ← intersec + 1
                pos1moves.REMOVE(ms1)
                pos2moves.REMOVE(ms2)
                break
        endFor
    endFor

    mismatch ← INTERSECREC<EXP_STATES>(pos1, pos2, pos1moves, pos2moves)
    union ← union - intersec
    return (intersec + mismatch) / union
```

Algorithm 17 A method that implements the Statistical Evaluation similarity measure.

```
procedure SIMILARITY<STAT_EVAL> (POSITION pos1, POSITION pos2)
    pos1moves ← GENERATE<MV_LEGAL>(pos1)
    pos2moves ← GENERATE<MV_LEGAL>(pos2)
    if pos1moves.SIZE == 0 and pos2moves.SIZE == 0 then
        return 1

    acc ← 0
    for ms1 in pos1moves do
        for ms2 in pos2moves do
            if ms1.MOVE == ms2.MOVE then
                acc ← acc + | ms1.SCORE - ms2.SCORE |
                pos2moves.REMOVE(ms2)
                break
        endFor
    endFor

    return SIGMOID(acc)
```

Chapter 5

Testing

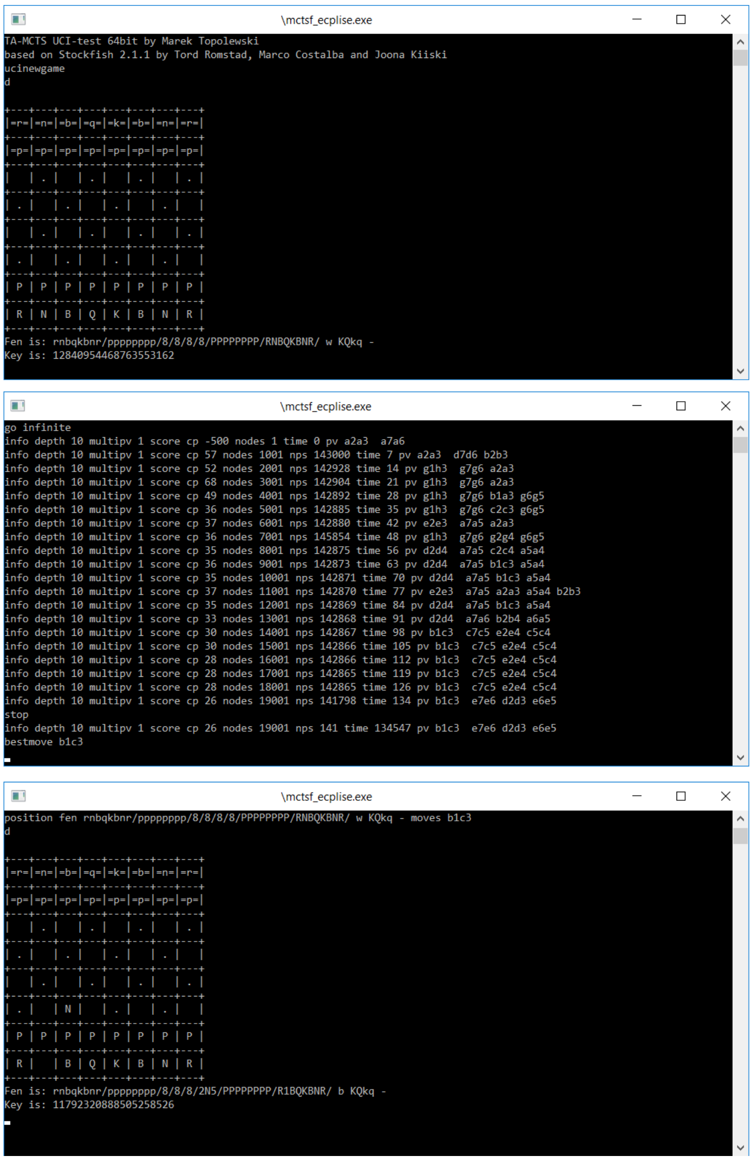
Upon designing and implementing the positional similarity measures, those methods ought to be tested for correctness and performance. This chapter begins by presenting the notions necessary for understanding the testing procedure - the UCI protocol and the FEN notation. The developed measures are then verified in respect to requirements of the outlined in section 3.1. Subsequently, the methods are evaluated in terms of efficiency, accuracy as well as their theoretical and practical computational performance. Lastly, the integration with the MCC engine is considered.

5.1 Universal Chess Interface

The Universal Chess Interface (UCI) was introduced in 2000 by Rudolf Huber and Stefan Meyer-Kahlen [29]. The protocol describes a two-way communication exchange between the Chess engine and an arbitrary program but its key use case is for Chess GUIs. The abstract specification allows for the UCI to be independent of the underlying operating system. The message passing is done via text commands to the standard input and returned to the standard output also as text. An exhaustive list of these commands with accompanying detailed explanations is shared by WBEC Ridderkerk [60]. A subset of instructions supported by the MCC engine and essential for the testing of this project are denoted and described below:

- **GUI/user to engine commands:**
  - position startpos [moves <string>] - resets the game state to the initial board configuration. If the moves command follows, then the sequence of space-separated actions is played.
  - position fen <string> [moves <string>] - transitions to the game state represented by the given FEN string. Optionally, the list of space-separated moves is parsed.
  - d - prints the information on the current position which includes: (1) an ASCII-encoded visual representation of the board, (2) FEN notation and (2) hash key.
  - go infinite - begin searching for the optimal action from the current position. In the infinite mode the execution can be interrupted only through the stop command.
- **Engine to GUI/user commands:**
  - info <string> - prints a set of parameters providing information on the current state of the search. The attributes include: (1) depth - current search depth in plies, (2) score cp - the maximal payoff determined by the engine, (3) nodes - number of states considered thus far and (4) pv - the optimal path that was found [60].
  - bestmove <string> [ponder <string>] - returns the best action according to the engine. Optionally, if searching in ponder-mode (continue execution during opponent's turn) then a move to ponder on is appended as well [3].

The application of the UCI protocol for the Stockfish engine is illustrated in Figure 5.1.



```
TA-MCTS UCI-test 64bit by Marek Topolewski
based on Stockfish 2.1.1 by Tord Romstad, Marco Costalba and Joona Kiiski
ucinewgame
d

+---+---+---+---+---+---+---+---+
|=r=|=n=|=b=|=q=|=k=|=b=|=n=|=r=|
+---+---+---+---+---+---+---+---+
|=p=|=p=|=p=|=p=|=p=|=p=|=p=|=p=|
+---+---+---+---+---+---+---+---+
|   | . |   | . |   | . |   | . |
+---+---+---+---+---+---+---+---+
| . |   | . |   | . |   | . |   |
+---+---+---+---+---+---+---+---+
|   | . |   | . |   | . |   | . |
+---+---+---+---+---+---+---+---+
| . |   | . |   | . |   | . |   |
+---+---+---+---+---+---+---+---+
| P | P | P | P | P | P | P | P |
+---+---+---+---+---+---+---+---+
| R | N | B | Q | K | B | N | R |
+---+---+---+---+---+---+---+---+
Fen is: rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR/ w KQkq -
Key is: 12840954468763553162
```

```
go infinite
info depth 10 multipv 1 score cp -500 nodes 1 time 0 pv a2a3  a7a6
info depth 10 multipv 1 score cp 57 nodes 1001 nps 143000 time 7 pv a2a3  d7d6 b2b3
info depth 10 multipv 1 score cp 52 nodes 2001 nps 142928 time 14 pv g1h3  g7g6 a2a3
info depth 10 multipv 1 score cp 68 nodes 3001 nps 142904 time 21 pv g1h3  g7g6 a2a3
info depth 10 multipv 1 score cp 49 nodes 4001 nps 142892 time 28 pv g1h3  g7g6 b1a3 g6g5
info depth 10 multipv 1 score cp 36 nodes 5001 nps 142885 time 35 pv g1h3  g7g6 c2c3 g6g5
info depth 10 multipv 1 score cp 37 nodes 6001 nps 142880 time 42 pv e2e3  a7a5 a2a3
info depth 10 multipv 1 score cp 36 nodes 7001 nps 145854 time 48 pv g1h3  g7g6 g2g4 g6g5
info depth 10 multipv 1 score cp 35 nodes 8001 nps 142875 time 56 pv d2d4  a7a5 c2c4 a5a4
info depth 10 multipv 1 score cp 36 nodes 9001 nps 142873 time 63 pv d2d4  a7a5 b1c3 a5a4
info depth 10 multipv 1 score cp 35 nodes 10001 nps 142871 time 70 pv d2d4  a7a5 b1c3 a5a4
info depth 10 multipv 1 score cp 37 nodes 11001 nps 142870 time 77 pv e2e3  a7a5 a2a3 a5a4 b2b3
info depth 10 multipv 1 score cp 35 nodes 12001 nps 142869 time 84 pv d2d4  a7a5 b1c3 a5a4
info depth 10 multipv 1 score cp 33 nodes 13001 nps 142868 time 91 pv d2d4  a7a6 b2b4 a6a5
info depth 10 multipv 1 score cp 30 nodes 14001 nps 142867 time 98 pv b1c3  c7c5 e2e4 c5c4
info depth 10 multipv 1 score cp 30 nodes 15001 nps 142866 time 105 pv b1c3  c7c5 e2e4 c5c4
info depth 10 multipv 1 score cp 28 nodes 16001 nps 142866 time 112 pv b1c3  c7c5 e2e4 c5c4
info depth 10 multipv 1 score cp 28 nodes 17001 nps 142865 time 119 pv b1c3  c7c5 e2e4 c5c4
info depth 10 multipv 1 score cp 28 nodes 18001 nps 142865 time 126 pv b1c3  c7c5 e2e4 c5c4
info depth 10 multipv 1 score cp 26 nodes 19001 nps 141798 time 134 pv b1c3  e7e6 d2d3 e6e5
stop
info depth 10 multipv 1 score cp 26 nodes 19001 nps 141 time 134547 pv b1c3  e7e6 d2d3 e6e5
bestmove b1c3
```

```
position fen rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR/ w KQkq - moves b1c3
d

+---+---+---+---+---+---+---+---+
|=r=|=n=|=b=|=q=|=k=|=b=|=n=|=r=|
+---+---+---+---+---+---+---+---+
|=p=|=p=|=p=|=p=|=p=|=p=|=p=|=p=|
+---+---+---+---+---+---+---+---+
|   | . |   | . |   | . |   | . |
+---+---+---+---+---+---+---+---+
| . |   | . |   | . |   | . |   |
+---+---+---+---+---+---+---+---+
|   | . |   | . |   | . |   | . |
+---+---+---+---+---+---+---+---+
| . |   | N |   | . |   | . |   |
+---+---+---+---+---+---+---+---+
| P | P | P | P | P | P | P | P |
+---+---+---+---+---+---+---+---+
| R |   | B | Q | K | B | N | R |
+---+---+---+---+---+---+---+---+
Fen is: rnbqkbnr/pppppppp/8/8/8/2N5/PPPPPPPP/R1BQKBNR/ b KQkq -
Key is: 11792320888505258526
```

Figure 5.1: A sequence of UCI commands supplied to the Stockfish engine with the resulting output.

5.2 FEN Notation

The Forsyth–Edwards Notation (FEN) has been introduced for computer Chess by Steven Edwards as a part of the Portable Game Notation (PGN) [25]. FEN is based on a system recording board configurations published by David Forsyth in Glasgow Weekly Herald in 1883.

A single FEN structure represents a given Chess game state by encoding the board configuration in one line of ASCII characters. Such a positional record consists of six space-separated fields resulting in the following syntax [16]:

$$\begin{aligned} < \text{FEN} > = {} & \text{<Piece placement> [SPACE] <Side to move> [SPACE]} \\ & \text{<Castling ability> [SPACE] <En passant target square> [SPACE]} \\ & \text{<Halfmove clock> [SPACE] <Fullmove counter>} \end{aligned} \tag{5.1}$$

For instance, the starting position in Chess has the following FEN encoding:

$$\text{rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq} - 0\ 1$$

where each field is defined as follows:

- **<Piece placement>** The board is divided into eight '/'-separated ranks that are represented in a big-endian [19] (decreasing) order from rank '8' to rank '1'. Subsequently, each rank segment is separated into eight cells encoded in a little-endian order from 'A' to 'H'. assigned a string containing pieces identifiers or integer numbers $x \in [1,8]$ representing the number adjacent empty board squares. A cell occupied by a figure is represented by an associated ASCII character of that piece:

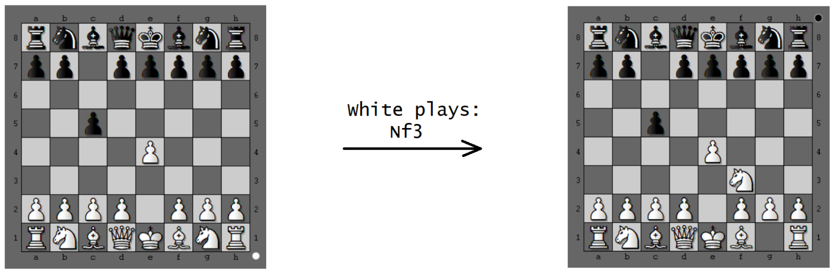
– p Black pawn	P White pawn
– n Black knight	N White knight
– B Black bishop	B White bishop
– r Black rook	R White rook
– q Black queen	Q White queen
– k Black king	K White king

- **<Side to move>** A colour of the agent that is to perform a move at the given position represented as a lower-case ASCII character: 'w' or 'b'.
- **<Castling ability>** A string of case-sensitive ASCII characters that indicate which figures (identifiers defined as above) can castle [44]. Possible values are a non-empty subset of { 'K', 'k', 'Q', 'q' } or '−' if neither of the agents is able to castle.
- **<En passant target square>** Regardless of the possibility of en passant capture [15], should a double-step of a pawn occur in the previous move, the resulting "target" cell is recorded. If such a move was not played then '−' is utilised.
- **<Halfmove clock>** An integer value employed to determine whether the 50-move draw rule [44] is applicable. The number is incremented each turn unless a pawn move or a capture occurred in which case the counter is reset to zero.
- **<Fullmove counter>** An integer encoding the number of *full moves*, i.e.one ply per each player. The initial value is 1 and, assuming White begins the game, it is incremented after Black moves.

It is important to note that despite the large number of properties that can be represented using the FEN notation, the history of the game, i.e. previous actions selected by each player, cannot be encoded. Even though other notations such as the EPD or PGN supporting historical information, preceding games states are dispensable in the scope of this project.

The use of the Forsyth–Edwards notation is particularly beneficial when combined with Stockfish as a number of implemented functions exist that take advantage of this representation. As introduced in section 4.4, it is possible to construct a Position object from a supplied FEN string. This function is incredibly convenient for testing as it enables the conduct evaluation without having create the objects and assigning its numerous properties manually.

Generation of a FEN from a Position object is implemented within the Stockfish engine as well. Unfortunately, the console environment requires the user to begin from either the initial board configuration or another FEN state and enter to the desired position by sequentially executing UCI commands (see section 5.1). Alternatively, FEN game states can be supplied not to the Chess engines directly, but to a Chess GUI such as the Arena instead. A software which allows for creation as well as modification of board configurations and performs FEN validation is offered by *lichess.org* [35]. Examples of FEN encoded positions and the corresponding board configurations are presented in Figure 5.2.



```
rnbqkbnr/pp1ppppp/8/2p5/4P3/8/PPPP1PPP/RNBQKBNR
w KQkq c6 0 2
```

```
rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R
b KQkq - 1 2
```

Figure 5.2: An example of a Chess position transition with corresponding FEN representations and board configurations generated using Arena [9] Chess GUI.

5.3 Unit Testing

To validate the accurate implementation of the similarity measures, **unit testing** is conducted on the developed methods. The steps necessary to complete this testing phase are:

1. Preserve the aheuristic property of MCTS.
2. Symmetric.
3. Normalised.
4. Conforms to the theory.

Adherence to the first step is predominantly achieved in the theory and implementation stage of the development. Therefore, it is sufficient to perform manual source code analysis to verify that no game-specific knowledge is utilised. An example of a method not adhering to the generality guideline can be adopting the endgame tablebase (see section 4.1) to evaluate states and consecutively using the results in the Statistical Evaluation measure. As recognised in section 3.4.2, the Game-Influential Pieces measure may introduce a degree of domain-specific information. However, one could argue that if the environment rules must be supplied to the MCTS algorithm, then a naïve set of pieces ought to be derivable as a consequence. Since none of the other measures were found to have adopted any heuristic knowledge, the ahuristicity phase of unit testing was deemed successful.

To conform to the symmetry property of each similarity measure, a simple test suite can be devised to explicitly confirm this invariance. The pseudocode for a procedure that implements this functionality is shown in Algorithm 18. The function takes a set of Position pair serving as a suit, however, regardless of the exhaustiveness of the testtest su the completeness of the method cannot be guaranteed. Hence, the symmetry property must furthermore be ensured already at the theoretical level.

Algorithm 18 A procedure which determines whether a similarity measure conforms the symmetry property for a set of Position pairs.

```
1: procedure SYMMETRYCHECK<SimMethod> (VECTOR< Pair<Position, Position>> testSuite )
2:     for pair in testSuite do
3:         res1 ← SIMILARITY<SimMethod>(pair[0], pair[1])
4:         res2 ← SIMILARITY<SimMethod>(pair[1], pair[0])
5:         if res1 ≠ res2 then
6:             return false
7:     endFor

8:     return true
```

Analogous to the symmetry test, the normalisation check may be facilitated by employing a combination of theoretical proofs as well as an explicit empirical function. The method takes a list of Position pairs, for each of which the similarity measure ought to return a value between 0 and 1. Implementation psuedocode of such a procedure is presented in Algorithm 19.

Algorithm 19 A procedure which ensures that a similarity measure returns a normalised values for a set of Position pairs.

```
1: procedure normalisedCheck<SimMethod> (vector< Pair<Position, Position>> testSuite )
2:     for pair in testSuite do
3:         if $0 \leqslant$ similarity<SimMethod>(pair[0], pair[1]) $\leqslant 1$ then
4:             return false
5:     endFor
6:     return true
```

Finally, to ensure conformance to the theoretical results, positional similarity measure for a number of board configurations must be manually calculated. To find those values, the formulas introduced in sections 3.4.1 to 3.4.7 are applied. Once obtained, they are compared to the computed similarity scores and solely exactly equivalent values are accepted. The psuedocode implementing the above functionality is presented in Algorithm 20.

Algorithm 20 A procedure which ensures that a similarity measure returns expected values for a set of Position pairs and corresponding target outcomes.

```
1: procedure theoryCheck<SimMethod> (vector< Pair<Position, Position>> testArg,
   vector<Float> testTarget)
2:     for idx in [0, testArg.length] do
3:         sim $\leftarrow$ similarity<SimMethod>(testArg[idx][0], testArg[idx][1])
4:         if sim $\neq$testTarget[idx] then
5:             return false
6:     endFor
7:     return true
```

Verification of a measure's correctness is considered successful if and only if all of the above unit tests return a positive value. Once completed, the methods can then be empirically evaluated in terms of performance, i.e. validate that the theoretical time complexity is proportional to the execution time. A function providing such measurements can be easily implemented with the aid of the "chrono" library [22] in C++ (see Algorithm 21).

Algorithm 21 A procedure measuring the mean execution time of a given similarity measure.

```
1: procedure TIMECHECK<SIMMETHOD> (VECTOR< PAIR<POSITION, POSITION>> testSuite)
2:     sumTime ← 0
3:     for pair in testSuite do
4:         startTime ← CURRENTTIME()
5:         sim ← SIMILARITY<SIMMETHOD>(pair[0], pair[1])
6:         sumTime ← sumTime + CURRENTTIME() - startTime
7:     endFor
8:     return sumTime / testSuite.LENGTH
```

5.4 Positional Testing

Upon correct verification of the developed positional similarity measures, the efficiency of these methods ought to be analysed. To facilitate this, two *modes* of **positional testing**, also referred to as **strategical testing**, were implemented:

- **FEN** testing
- **Child nodes** testing

In both scenarios, the expected outcome is a table containing similarity values for different positions and measures utilised. However, the two modes can be distinguished based on the board configurations considered in the given suite.

The **FEN testing** mode takes as an input a CSV [62] file containing a pair of comma-separated FEN positions in each line. The strings are subsequently converted into the POSITION objects and positional similarity is computed for every applicable measure. The results for each position pair are encoded onto a single comma-separated string and appended to an output CSV file. Moreover, to indicates whether both the first and the second position are a trap states, an additional attribute **TRAP** is created.

The use case of the FEN testing mode is to manually measure the values returned by the developed positional similarities with correlation to the trap persistence property. A particular sequence of moves can be considered by supplying to the method consecutive FEN positions.

On the other hand, the **child nodes** mode takes only a single position represented in the FEN notation. Analogous to the previous method, the Position instance is generated from the encoded string, however, instead of matching it against an auxiliary game state, the grandchild nodes of that vertex are selected. This approach attempts to simulate the practical usage of the similarity measures within the Monte Carlo Chess engine, namely, to determine the probability of trap persistence between agent's turns. The grandchild states are obtained by first generating a list of all successors of the root which is equivalent to inspecting all states the opponent may play from. Lastly, every legal move the rival can select is considered by expanding all children of the previously created vertices. Although the second mode also contains the TRAP property for every root-grandchild position pair, its value depends entirely on the trap existence in the latter (grandchild) position.

To conclude, the output CSV file in either case has the following schema:

FEN / MOVE	**FEN / MOVE**	$\mathbf{sim}_1$	$\dots$	$\mathbf{sim}_n$	**TRAP**
string	string	float	$\dots$	float	boolean

The above format allows for the data to be imported to another environment were the information can be easily parsed and studied. In this project Python is utilised for this purpose, as it is a language commonly employed for data processing tasks with abundance of advantageous tools [24], particularly the pandas [42] and numpy [39] packages. The outcome analysis focuses on data generated in the child nodes mode testing due to the resemblance to the intended setting of the similarity measures.

Firstly, a root position should be selected to be trap state which simulates the trap identification phase. Subsequently, the trap verification procedure is replicated by analysing the grandchild nodes for trap persistence. To obtain the output information, all pairs of root-grandchild nodes ought to be considered and corresponding similarity values must be calculated. To examine the performance of each measure in regards to the trap persistence problem, the observations need to be further divided into two subsets:

- **trap persists** - set of similarity values where a trap is present in both game states.
- **trap gone** - observations representing the pairs in which no trap is found in the second position.

Once the CSV file is loaded into a DataFrame [43], a process of the reduction of dataset's complexity known as **feature extraction** [2] is performed. The payoff of its successful application is a set of informative and representative features of the underlying data. This feature vector substitutes the raw observation format which is difficult to analyse and introduces unnecessary redundancy. For the purpose of this project, the devised feature extraction procedure allows for the examination of the similarity measures' efficiencies. The pandas library implements a describe() function enabling fast derivation of a number of descriptive statistics. The following features are extracted from the returned structure for every similarity measure:

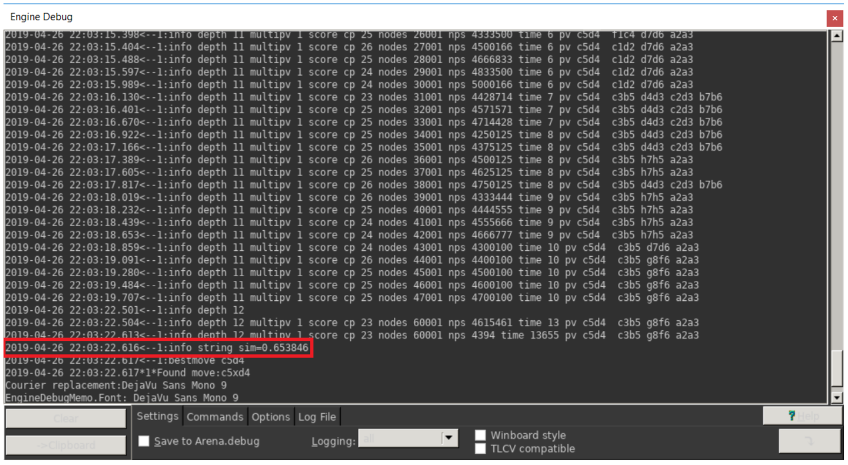
- **AVG** - the mean of the observations.
- **STD** - the standard deviation of the values.
- **MAX** - the maximum value across the dataset.
- **MIN** - the minimum similarity score.
- **SKW** - the skewness of the distribution [54] [64].

5.5 Integration Testing

Integration testing ensures that the developed measures are combined properly with the Monte Carlo Chess engine its embedded within. Having corrected the calculation of the positional similarity (see section 4.4), the components are interconnected by executing the updated uct() function.

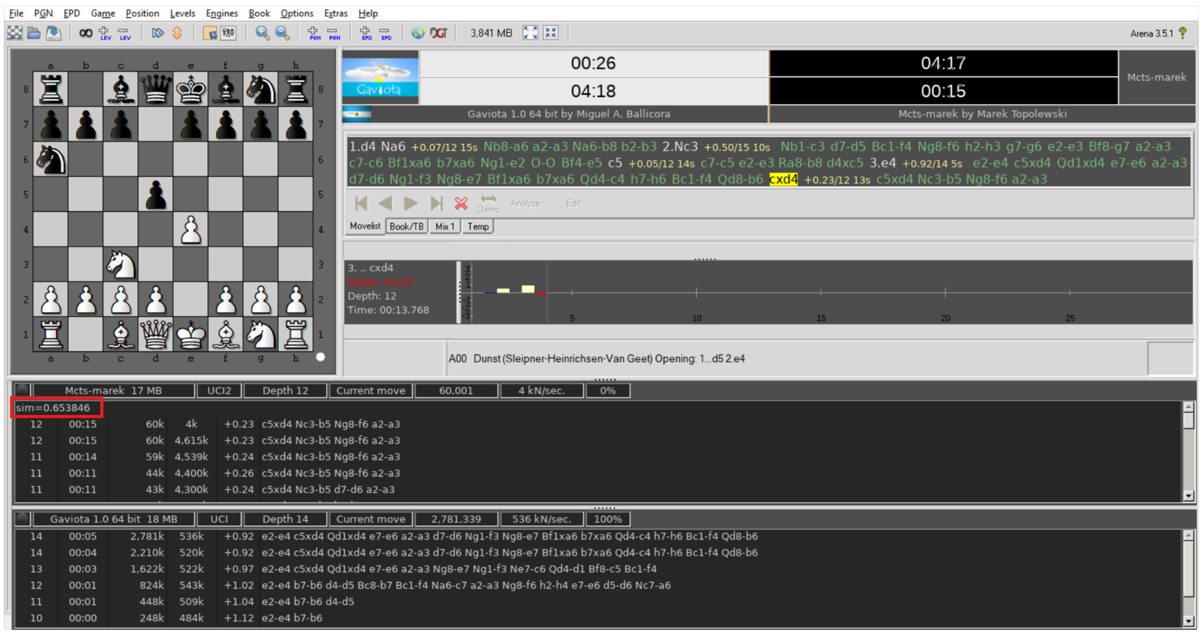
To validate correct integration, the initial step involves compiling and building the project which provides an assertion from the *linker* [46] that the appended source code was integrated successfully. Despite ensuring syntactic correctness, the built itself provides no semantic proof, therefore, the *makefile* of the Stockfish is utilised. The makefile is a script comprised of a sequence of instructions aiming to automate the generation of the target executable [49]. Not only does it perform compilation and linking, but is also able to optimise the result based on the supplied dependencies between the specified files. The semantic integration is achieved by firstly modifying its directives to interconnect the newly implemented similarity component. Subsequently, a performance test of the engine is executed by invoking the BENCHMARK() function supplied by Stockfish. Should any anomalies occur at runtime, either the procedure will fail or the resulting benchmark will indicate depleted performance.

Another technique employed in this project is to log the obtained similarity scores at each call to the UCT() search method. This functionality must be compatible with the UCI protocol to avoid erroneous parsing of the instructions by the GUI, hence, the info command ought to be utilised (see Figure 5.3). The logged messages will either raise an error at the execution time should an unexpected value be returned (e.g. a null value) or they must be manually investigated to assert their correctness. The investigation may be expedited through the use of the Arena GUI which supports automatic gameplay between engines and conveniently prints the info commands (see Figure 5.4).



```
2019-04-26 22:03:15.398<--1:info depth 11 multipv 1 score cp 25 nodes 26001 nps 4333500 time 6 pv c5d4  f1c4 d7d6 a2a3
2019-04-26 22:03:15.404<--1:info depth 11 multipv 1 score cp 26 nodes 27001 nps 4500166 time 6 pv c5d4  c1d2 d7d6 a2a3
2019-04-26 22:03:15.488<--1:info depth 11 multipv 1 score cp 25 nodes 28001 nps 4666833 time 6 pv c5d4  c1d2 d7d6 a2a3
2019-04-26 22:03:15.597<--1:info depth 11 multipv 1 score cp 24 nodes 29001 nps 4833500 time 6 pv c5d4  c1d2 d7d6 a2a3
2019-04-26 22:03:15.989<--1:info depth 11 multipv 1 score cp 24 nodes 30001 nps 5000166 time 6 pv c5d4  c1d2 d7d6 a2a3
2019-04-26 22:03:16.130<--1:info depth 11 multipv 1 score cp 23 nodes 31001 nps 4428714 time 7 pv c5d4  c3b5 d4d3 c2d3 b7b6
2019-04-26 22:03:16.401<--1:info depth 11 multipv 1 score cp 25 nodes 32001 nps 4571571 time 7 pv c5d4  c3b5 d4d3 c2d3 b7b6
2019-04-26 22:03:16.670<--1:info depth 11 multipv 1 score cp 25 nodes 33001 nps 4714428 time 7 pv c5d4  c3b5 d4d3 c2d3 b7b6
2019-04-26 22:03:16.922<--1:info depth 11 multipv 1 score cp 25 nodes 34001 nps 4250125 time 8 pv c5d4  c3b5 d4d3 c2d3 b7b6
2019-04-26 22:03:17.166<--1:info depth 11 multipv 1 score cp 25 nodes 35001 nps 4375125 time 8 pv c5d4  c3b5 d4d3 c2d3 b7b6
2019-04-26 22:03:17.389<--1:info depth 11 multipv 1 score cp 26 nodes 36001 nps 4500125 time 8 pv c5d4  c3b5 h7h5 a2a3
2019-04-26 22:03:17.605<--1:info depth 11 multipv 1 score cp 25 nodes 37001 nps 4625125 time 8 pv c5d4  c3b5 h7h5 a2a3
2019-04-26 22:03:17.817<--1:info depth 11 multipv 1 score cp 26 nodes 38001 nps 4750125 time 8 pv c5d4  c3b5 d4d3 c2d3 b7b6
2019-04-26 22:03:18.019<--1:info depth 11 multipv 1 score cp 26 nodes 39001 nps 4333444 time 9 pv c5d4  c3b5 h7h5 a2a3
2019-04-26 22:03:18.232<--1:info depth 11 multipv 1 score cp 25 nodes 40001 nps 4444555 time 9 pv c5d4  c3b5 h7h5 a2a3
2019-04-26 22:03:18.439<--1:info depth 11 multipv 1 score cp 24 nodes 41001 nps 4555666 time 9 pv c5d4  c3b5 h7h5 a2a3
2019-04-26 22:03:18.653<--1:info depth 11 multipv 1 score cp 24 nodes 42001 nps 4666777 time 9 pv c5d4  c3b5 h7h5 a2a3
2019-04-26 22:03:18.859<--1:info depth 11 multipv 1 score cp 24 nodes 43001 nps 4300100 time 10 pv c5d4  c3b5 d7d6 a2a3
2019-04-26 22:03:19.091<--1:info depth 11 multipv 1 score cp 26 nodes 44001 nps 4400100 time 10 pv c5d4  c3b5 g8f6 a2a3
2019-04-26 22:03:19.280<--1:info depth 11 multipv 1 score cp 25 nodes 45001 nps 4500100 time 10 pv c5d4  c3b5 g8f6 a2a3
2019-04-26 22:03:19.484<--1:info depth 11 multipv 1 score cp 25 nodes 46001 nps 4600100 time 10 pv c5d4  c3b5 g8f6 a2a3
2019-04-26 22:03:19.707<--1:info depth 11 multipv 1 score cp 25 nodes 47001 nps 4700100 time 10 pv c5d4  c3b5 g8f6 a2a3
2019-04-26 22:03:22.501<--1:info depth 12
2019-04-26 22:03:22.504<--1:info depth 12 multipv 1 score cp 23 nodes 60001 nps 4615461 time 13 pv c5d4  c3b5 g8f6 a2a3
2019-04-26 22:03:22.613<--1:info depth 12 multipv 1 score cp 23 nodes 60001 nps 4394 time 13655 pv c5d4  c3b5 g8f6 a2a3
2019-04-26 22:03:22.616<--1:info string sim=0.653846
2019-04-26 22:03:22.617<--1:bestmove c5d4
2019-04-26 22:03:22.617*1*Found move:c5xd4
Courier replacement:DejaVu Sans Mono 9
EngineDebugMemo.Font: DejaVu Sans Mono 9
```

Figure 5.3: The engine log providing an overview of the entire UCI communication between the Arena GUI and the modified TA-MCC engine.



```
00:26    04:17
04:18    00:15
Gaviota 1.0 64 bit by Miguel A. Ballicora    Mcts-marek by Marek Topolewski

1.d4 Na6 +0.07/12 15s Nb8-a6 a2-a3 Na6-b8 b2-b3 2.Nc3 +0.50/15 10s Nb1-c3 d7-d5 Bc1-f4 Ng8-f6 h2-h3 g7-g6 e2-e3 Bf8-g7 a2-a3
c7-c6 Bf1xa6 b7xa6 Ng1-e2 O-O Bf4-e5 c5 +0.05/12 14s c7-c5 e2-e3 Ra8-b8 d4xc5 3.e4 +0.92/14 5s e2-e4 c5xd4 Qd1xd4 e7-e6 a2-a3
d7-d6 Ng1-f3 Ng8-e7 Bf1xa6 b7xa6 Qd4-c4 h7-h6 Bc1-f4 Qd8-b6 cxd4 +0.23/12 13s c5xd4 Nc3-b5 Ng8-f6 a2-a3

3. .. cxd4
Value: +0.23
Depth: 12
Time: 00:13.768

A00  Dunst (Sleipner-Heinrichsen-Van Geet) Opening: 1...d5 2.e4

Mcts-marek  17 MB | UCI2 | Depth 12 | Current move | 60,001 | 4 kN/sec. | 0%
sim=0.653846
 12   00:15      60k     4k   +0.23  c5xd4 Nc3-b5 Ng8-f6 a2-a3
 12   00:15      60k  4,615k  +0.23  c5xd4 Nc3-b5 Ng8-f6 a2-a3
 11   00:14      59k  4,539k  +0.24  c5xd4 Nc3-b5 Ng8-f6 a2-a3
 11   00:11      44k  4,400k  +0.26  c5xd4 Nc3-b5 Ng8-f6 a2-a3
 11   00:11      43k  4,300k  +0.24  c5xd4 Nc3-b5 d7-d6 a2-a3

Gaviota 1.0 64 bit  18 MB | UCI | Depth 14 | Current move | 2,781,339 | 536 kN/sec. | 100%
 14   00:05   2,781k   536k   +0.92  e2-e4 c5xd4 Qd1xd4 e7-e6 a2-a3 d7-d6 Ng1-f3 Ng8-e7 Bf1xa6 b7xa6 Qd4-c4 h7-h6 Bc1-f4 Qd8-b6
 14   00:04   2,210k   520k   +0.92  e2-e4 c5xd4 Qd1xd4 e7-e6 a2-a3 d7-d6 Ng1-f3 Ng8-e7 Bf1xa6 b7xa6 Qd4-c4 h7-h6 Bc1-f4 Qd8-b6
 13   00:03   1,622k   522k   +0.97  e2-e4 c5xd4 Qd1xd4 e7-e6 a2-a3 Ng8-e7 Ng1-f3 Ne7-c6 Qd4-d1 Bf8-c5 Bc1-f4
 12   00:01     824k   543k   +1.02  e2-e4 b7-b6 d4-d5 Bc8-b7 Bc1-f4 Na6-c7 a2-a3 Ng8-f6 h2-h4 e7-e6 d5-d6 Nc7-a6
 11   00:01     448k   509k   +1.04  e2-e4 b7-b6 d4-d5
 10   00:00     248k   484k   +1.12  e2-e4 b7-b6
```

Figure 5.4: Snapshot of the Arena with a search information window correctly parsing similarity value.

5.6 Experimental Results

This section presents the results of empirical testing methods presented in sections 5.3 to 5.5. The designed positional similarity measures are evaluated by analysing the numerical outcomes and investigating the supporting illustrations derived from the data.

The first stage discussed is the unit testing. The aim of this phase is to ensure that the developed measures are compliant with the prerequisites and evaluate their practical performance. The test suite necessary for the execution of the unit tests contains 100 records in total and can be divided into two equal partitions: (1) consecutive positions and (2) random state pairs. Moreover, each tuple contains the expected similarity value.

As presented in Table 5.1, all methods conform to the four properties outlines in section 5.3, however, variations can be observed for in the last two columns regarding the performance. Although the theoretical time complexity is proportional to the practical efficiency, the statement is false in case of the *Legal Actions* and the *Depth and Breadth* measures. Such behaviour is the result of unpredictably low efficiency in retrieving the depth which introduces a significant constant overhead. Another anomaly occurs for the *Expandable States* method which despite equivalent time complexity and similar implementation to *Legal Actions* has approximately 180 times longer execution time. In this case, the bottleneck is a result of slow position generation required to obtain the expandable state. Another property for this measure can be observed, namely, it is extremely rare for a given position to reoccur (especially outside of the endgame), hence, the similarity returned is 0 for an overwhelming majority of positions. The two characteristics combined are the root cause of the inability to calculate the *Recursive Expandable States* due to all child nodes being considered as mismatches, hence, attaining the worst case complexity for nearly every pair. Interestingly enough, the *Game-Influential Pieces* measure achieved the second best average execution time and demonstrated the evident benefit of the Bitboard structures. However, not all environments can be represented in this format or the required space complexity may outweigh the performance gain.

Similarity Measure	Aheuristic	Symmetric	Normalised	Accurate	Time Compl.	Execution Time [μs]
Constant Factor	Yes	Yes	Yes	Yes	$O(1)$	10.0
Depth and Breadth	Yes	Yes	Yes	Yes	$O(N)$	186.5
Game-Influential Pieces	Yes	Yes	Yes	Yes	$O(M)$	29.6
Legal Actions	Yes	Yes	Yes	Yes	$O(N^2)$	80.2
Rec. Legal Actions	Yes	Yes	Yes	Yes	$O(N^4)$	12,927.8
Expandable States	Yes	Yes	Yes	Yes	$O(N^2)$	14,504.0
Rec. Expandable States	Yes	Yes	Yes	Yes	$O(N^4)$	N/A

Table 5.1: Results of the unit testing on a test suite comprised of one hundred input records. Time complexities and execution times are colour coded, red for the least, through yellow, and green for the most efficient values. The "N/A" execution time of the *Recursive Expandable States* measure indicates the *"Out of memory"* exception being caught.

The next testing stage is the positional evaluation and child node mode specifically. The feature extraction process implemented in Python is performed on a game state representing the Légal Trap presented in section 2.6. The results for trap persisting states are presented in Table 5.2 whereas the outcomes for the grandchild nodes without trap presence are shown in Table 5.3.

Similarity Measure	**AVG**	**STD**	**MAX**	**MIN**	**SKW**
Constant Factor	0.500	0.000	0.500	0.500	0.000
Depth and Breadth	0.978	0.031	1.000	0.553	-9.763
Game-Influential Pieces	0.995	0.017	1.000	0.933	-3.362
Legal Actions	0.740	0.126	0.919	0.056	-0.977
Recursive Legal Actions	0.844	0.095	0.950	0.056	-2.521
Expandable States	0.000	0.000	0.000	0.000	0.000
Recursive Expandable States	0.000	0.000	0.000	0.000	0.000

Table 5.2: Feature extraction results for the trap persisting pairs of states.

Similarity Measure	**AVG**	**STD**	**MAX**	**MIN**	**SKW**
Constant Factor	0.500	0.000	0.500	0.500	0.000
Depth and Breadth	0.955	0.088	1.000	0.527	-4.028
Game-Influential Pieces	0.992	0.022	1.000	0.933	-2.352
Legal Actions	0.625	0.172	0.944	0.000	-1.748
Recursive Legal Actions	0.766	0.174	0.955	0.027	-2.981
Expandable States	0.000	0.000	0.000	0.000	0.000
Recursive Expandable States	0.000	0.000	0.000	0.000	0.000

Table 5.3: Feature extraction results for the not trap persisting pairs of states.

The features itself are insufficient to objectively determine which positional similarity measure are ”better”, hence, prior to their analysis a definition of an efficient measure must be established. Let an extracted property be AGG for the entire observation set, whereas the same attribute for the *trap persisting* and *trap gone* subsets are AGG_1 and AGG_0 respectively. An ideal similarity measure would have the mean values satisfying $\text{AVG}_0 < 0.5 \wedge \text{AVG}_1 > 0.5$ which implies that methods that on average classify the positions correctly are favoured. Another property is the non-overlapping ranges of observations, namely: $\text{MIN}_1 > \text{MAX}_0$. Finally, the values should be moderately uniformly distributed as indicated by the skewness of the dataset nearing 0. The distribution can be further illustrated by histograms [45] displayed below for each of the similarity methods separately.

The Constant Factor similarity measure is remarkably fast to compute, however, it does not match any of the feature requirements listed above. In fact, the value vector as well as the distribution (see Figure 5.5) is equivalent for both trap persistence cases indicating no distinguishment between them and, therefore, implying low efficiency.

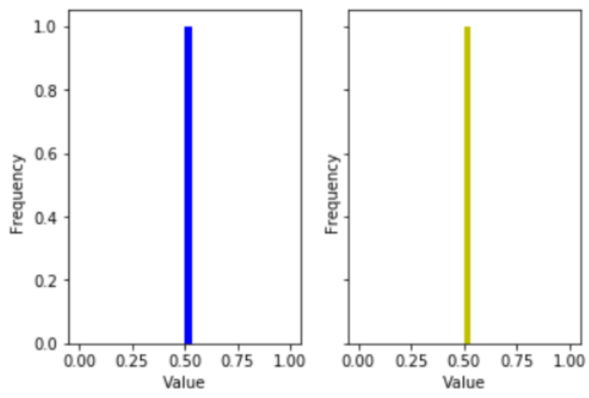


Figure 5.5: Two histograms illustrating the distribution of the similarity values as returned by the Constant Factor measure for the trap persisting (left) and not trap persisting (right) grandchild nodes.

The Depth and Breadth similarity measure resembles in the results the Constant Factor method in that it also is heavily centred around its average as proved by the low deviation from the mean AVG_0 and AVG_1. Despite AVG_1 being higher than AVG_0, the difference is negligible. The histograms presented in Figure 5.6 reveal that the measure does classify some not trap persisting states correctly which proves the method superior to the Constant Factor, however, a definite majority of observations remain misidentified. After further analysis, the positions with a low similarity score proved to be the states in which the breadth of actions is reduced by the agent having to act upon being checked by the opponent. Although this discovery demonstrates that the depth has little influence over the target accuracy and that the breath is indeed correlated with the persistence trap, the improved performance is hindered by the overall increase in the observed similarity values.

The successive measure is the Game-Influential Pieces method. Having analysed the extracted features, a conclusion that the considered properties are relatively equivalent implies no partitioning between the persisting and not persisting states. The histograms show that the observations are skewed towards 1 less in the latter case, however, the divergence is unsubstantial. In summary, the Game-Influential Pieces measure has the worst performance out of thus far evaluated methods, the cause being a high dependence on the domain characteristics. For instance, while in Checkers most of the environment transitions result in a capture, the opposite is true for the game of Chess, hence, insignificant variance from similarity of value 1.

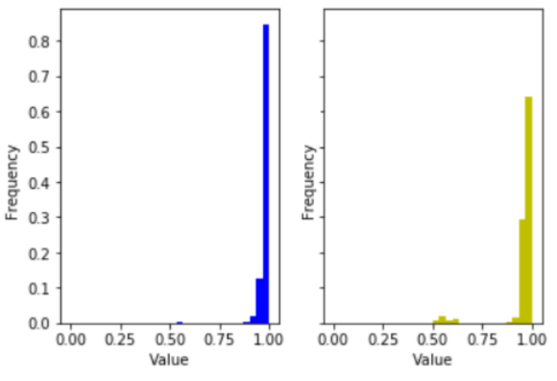


Figure 5.6: Distributions of Depth and Breadth similarities for trap persisting and trap gone states.

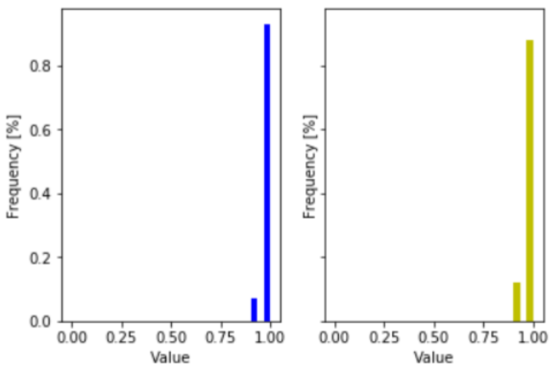


Figure 5.7: Histograms representing the distribution of the similarity scores returned by the Game-Influential Pieces measure for the trap persisting (left) and not trap persisting (right) grandchild nodes.

Although the preceding measures were largely determined to be inefficient, the Legal Actions measure exhibits promising characteristics. Examining the histograms of the similarity values reveals that the scores are more uniformly distributed and the observations for the trap persistent positions are shifted towards 1 further than in the alternative case. Moreover, the both means are lower and maintain the $\text{AVG}_1 > \text{AVG}_0$ property. On the other hand, the extrama do not meet the characteristics of an efficient similarity measure. Furthermore, the skew remains negative indicating a shift in observations towards higher values.

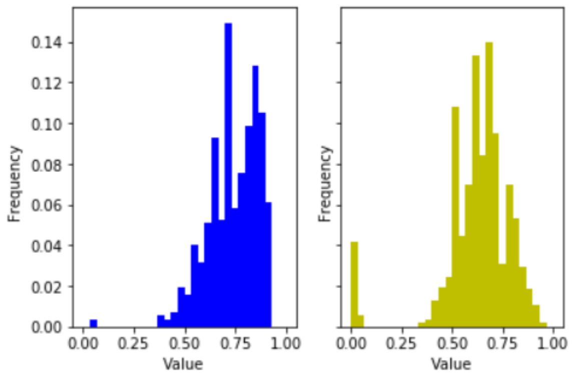


Figure 5.8: Two histograms illustrating the distribution of the similarity values as returned by the Legal Actions measure for the trap persisting (left) and not trap persisting (right) grandchild states.

The goal of the *Recursive Legal Actions* measure is to improve the classification through extending the comparison procedure to the children of mismatching actions. As illustrated by the graphs (see Figure 3.5) as well as the feature data, an overall increase in the similarity values can be detected. The observation confirms the theoretical expectation, i.e. the recursive variation will always return at least as high of a result as the standard *Legal Actions* method. In spite of the potential to increase the accuracy of distinguishing trap persisting states, the measure is outperformed by its basic implementation as indicated by the decrease in the mean difference from $\Delta\text{AVG} \approx 0.115$ to $\Delta\text{AVG}_{\text{REC}} \approx 0.078$.

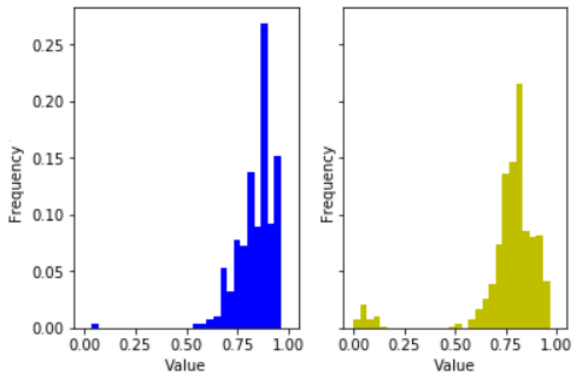


Figure 5.9: Distribution plots of the similarity values returned by the Recursive Legal Actions measure for the trap persisting (left) and not trap persisting (right) grandchild nodes.

The final two methods that are evaluated in the positional testing are the *Expandable States* and *Recursive Expandable States* measures. Both approaches were determined to be exceptionally inefficient due to the properties of the Chess domain, namely, the infrequency of encountering the same position in a legal sequence of moves. The measures were tested for other positions, however, the observations were identical regardless of the root state. Due to this invariance, the similarity values are equivalent to zero for all grandchild nodes (see Figure 5.10).

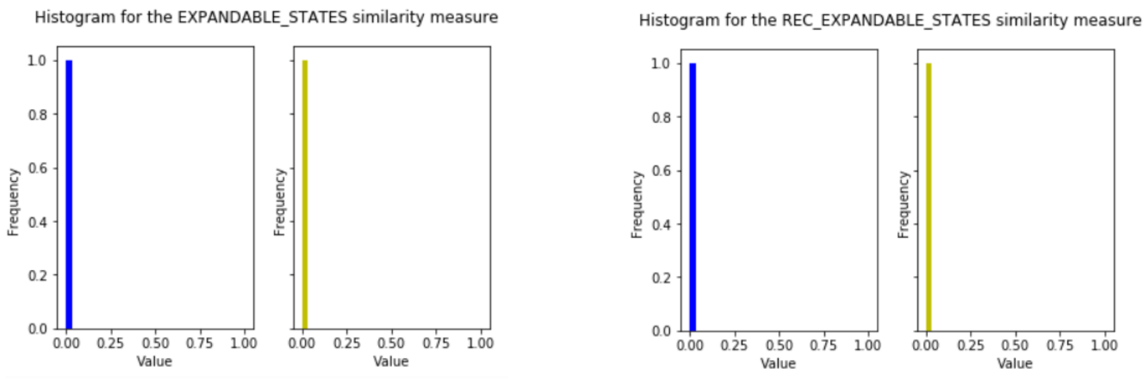
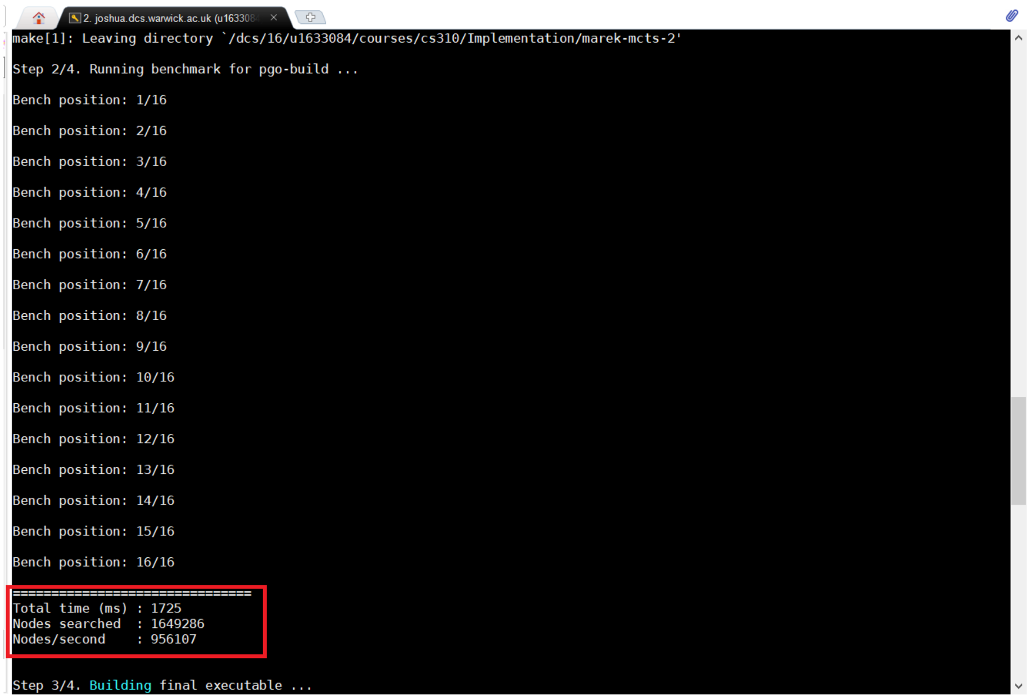


Figure 5.10: Distribution plots of the similarity values returned by the Expandable States and Recursive Expandable States measure. Observations are divided into trap persisting (blue) and not trap persisting (yellow) position pairs.

Lastly, the results of integration testing are discussed. This phase focuses on analysing the output of the make function which behaviour is defined by the adjusted makefile. As illustrated by Figure 5.11, the building and linking process is completed successfully which allows for the benchmark to be obtained. The console logs show no errors or warning with the exception of two unused Position variables in the Constant Factor measure signature necessary for the generic definition of the similarity method (see signature 4.3).

Moreover, the developed engine outperforms the MCC variation proposed in the TA-MCTS thesis and is nearly as fast as the base MCC implementation (see Table 5.4). The program is able to evaluate 8,242 nodes per second (nps) more than the previous TA-MCC variant and is only 12,921 nps slower than the MCC implementation which does not account for search traps.



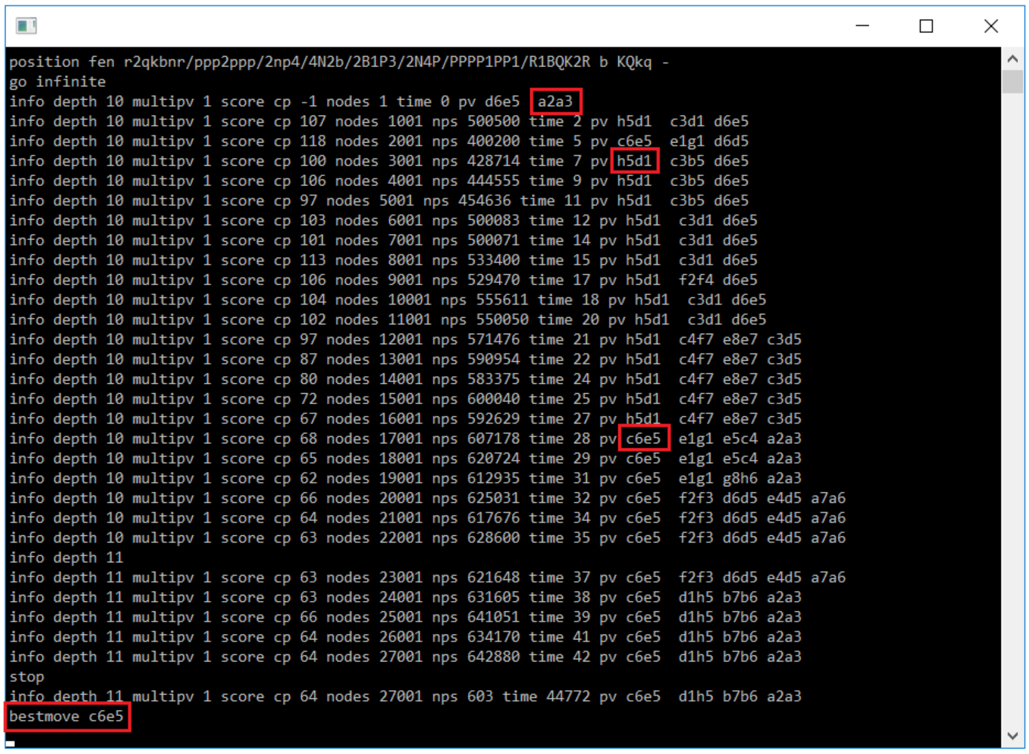
```
make[1]: Leaving directory `/dcs/16/u1633084/courses/cs310/Implementation/marek-mcts-2'

Step 2/4. Running benchmark for pgo-build ...

Bench position: 1/16

Bench position: 2/16

Bench position: 3/16

Bench position: 4/16

Bench position: 5/16

Bench position: 6/16

Bench position: 7/16

Bench position: 8/16

Bench position: 9/16

Bench position: 10/16

Bench position: 11/16

Bench position: 12/16

Bench position: 13/16

Bench position: 14/16

Bench position: 15/16

Bench position: 16/16

===============================
Total time (ms) : 1725
Nodes searched  : 1649286
Nodes/second    : 956107

Step 3/4. Building final executable ...
```

Figure 5.11: The engine log providing an overview of the entire UCI communication between the Arena GUI and the modified TA-MCC engine.

Engine	Total time [ms]	Nodes searched	Speed [nps]
MCC	1,702	1,649,286	969,028
TA-MCC	1,740	1,649,286	947,865
Enhanced TA-MCC	1,725	1,649,286	956,107

Table 5.4: Benchmarks of three variations of the MCC engine: (1) the original Monte Carlo Chess engine, (2) the Trap-Adaptive version and (3) the implementation of this project.

To ensure that the engine is capable of detecting and avoid search traps, the starting position of the Légal Trap is manually loaded. The UCT() function is initialised and the search data is examined for the best move indicated by the program at each step. The logs presented in Figure 5.12 indicates that although the trap move "h5d1" was deemed favourable in the beginning, the agent mitigates the risk by choosing the "c6e5" action and just after the search depth of 11 plies assigns the following scores: "h5d1" = 463.245; "c6e5" = 790.001, which demonstrates correct identification of being at a danger of a search trap.



```
position fen r2qkbnr/ppp2ppp/2np4/4N2b/2B1P3/2N4P/PPPP1PP1/R1BQK2R b KQkq -
go infinite
info depth 10 multipv 1 score cp -1 nodes 1 time 0 pv d6e5  a2a3
info depth 10 multipv 1 score cp 107 nodes 1001 nps 500500 time 2 pv h5d1  c3d1 d6e5
info depth 10 multipv 1 score cp 118 nodes 2001 nps 400200 time 5 pv c6e5  e1g1 d6d5
info depth 10 multipv 1 score cp 100 nodes 3001 nps 428714 time 7 pv h5d1  c3b5 d6e5
info depth 10 multipv 1 score cp 106 nodes 4001 nps 444555 time 9 pv h5d1  c3b5 d6e5
info depth 10 multipv 1 score cp 97 nodes 5001 nps 454636 time 11 pv h5d1  c3b5 d6e5
info depth 10 multipv 1 score cp 103 nodes 6001 nps 500083 time 12 pv h5d1  c3d1 d6e5
info depth 10 multipv 1 score cp 101 nodes 7001 nps 500071 time 14 pv h5d1  c3d1 d6e5
info depth 10 multipv 1 score cp 113 nodes 8001 nps 533400 time 15 pv h5d1  c3d1 d6e5
info depth 10 multipv 1 score cp 106 nodes 9001 nps 529470 time 17 pv h5d1  f2f4 d6e5
info depth 10 multipv 1 score cp 104 nodes 10001 nps 555611 time 18 pv h5d1  c3d1 d6e5
info depth 10 multipv 1 score cp 102 nodes 11001 nps 550050 time 20 pv h5d1  c3d1 d6e5
info depth 10 multipv 1 score cp 97 nodes 12001 nps 571476 time 21 pv h5d1  c4f7 e8e7 c3d5
info depth 10 multipv 1 score cp 87 nodes 13001 nps 590954 time 22 pv h5d1  c4f7 e8e7 c3d5
info depth 10 multipv 1 score cp 80 nodes 14001 nps 583375 time 24 pv h5d1  c4f7 e8e7 c3d5
info depth 10 multipv 1 score cp 72 nodes 15001 nps 600040 time 25 pv h5d1  c4f7 e8e7 c3d5
info depth 10 multipv 1 score cp 67 nodes 16001 nps 592629 time 27 pv h5d1  c4f7 e8e7 c3d5
info depth 10 multipv 1 score cp 68 nodes 17001 nps 607178 time 28 pv c6e5  e1g1 e5c4 a2a3
info depth 10 multipv 1 score cp 65 nodes 18001 nps 620724 time 29 pv c6e5  e1g1 e5c4 a2a3
info depth 10 multipv 1 score cp 62 nodes 19001 nps 612935 time 31 pv c6e5  e1g1 g8h6 a2a3
info depth 10 multipv 1 score cp 66 nodes 20001 nps 625031 time 32 pv c6e5  f2f3 d6d5 e4d5 a7a6
info depth 10 multipv 1 score cp 64 nodes 21001 nps 617676 time 34 pv c6e5  f2f3 d6d5 e4d5 a7a6
info depth 10 multipv 1 score cp 63 nodes 22001 nps 628600 time 35 pv c6e5  f2f3 d6d5 e4d5 a7a6
info depth 11
info depth 11 multipv 1 score cp 63 nodes 23001 nps 621648 time 37 pv c6e5  f2f3 d6d5 e4d5 a7a6
info depth 11 multipv 1 score cp 63 nodes 24001 nps 631605 time 38 pv c6e5  d1h5 b7b6 a2a3
info depth 11 multipv 1 score cp 66 nodes 25001 nps 641051 time 39 pv c6e5  d1h5 b7b6 a2a3
info depth 11 multipv 1 score cp 64 nodes 26001 nps 634170 time 41 pv c6e5  d1h5 b7b6 a2a3
info depth 11 multipv 1 score cp 64 nodes 27001 nps 642880 time 42 pv c6e5  d1h5 b7b6 a2a3
stop
info depth 11 multipv 1 score cp 64 nodes 27001 nps 603 time 44772 pv c6e5  d1h5 b7b6 a2a3
bestmove c6e5
```

Figure 5.12: An engine console log proving the engine adapts to identified search trap that can be entered by selecting the "h5d1" move.

5.7 Observation Scaling

The experimental results indicate that all of the investigated similarity measures tend to overestimate. This observation is demonstrated by the negative skewness and the high mean for the entire dataset as well as the trap persisting and not trap persisting subsets. A frequently employed solution to this problem in Machine Learning is **standardisation**. This procedure is a part of the feature scaling process which aim to equalise the weights of the observations by scaling their magnitudes. To perform a standardisation the complete dataset ought to be available. This requirement implicates that the procedure is an *offline* method and cannot be applied to an *online* algorithm such as the one considered in this project. However, the potential advantages of standardisation are discussed for the purpose of the analysis of the similarity measures themselves.

The calculation of the standardised values is performed by substituting the actual observations with their 'Z scores' [56]:

$$x' = \frac{x - \text{mean}(\boldsymbol{x})}{\text{std}(\boldsymbol{x})} \tag{5.2}$$

The outcome is a set of observations with a zero-mean distribution and a unit variance. An important point to consider is the fact that the standardisation cannot be performed for the Constant Factor or the Expandable States measures as their observation values are all equivalent, hence, are excluded in the subsequent investigation.

The feature scaling is conducted on the Legal Actions measure to instantiate the process. As presented in Figure 5.13, the histogram preserves the original shape but becomes shifted to account for adjusted mean and standard deviation. Another conclusion that can be drawn from the illustration is the no longer normalised range. Even though the issue could be resolved through **normalisation** [28] which is the other phase of feature scaling, the advantages of standardisation will be lost should it be applied (see Figure 5.13). The distributions appear to be identical, however, the overall mean decreased from 0.708 to 0.669.

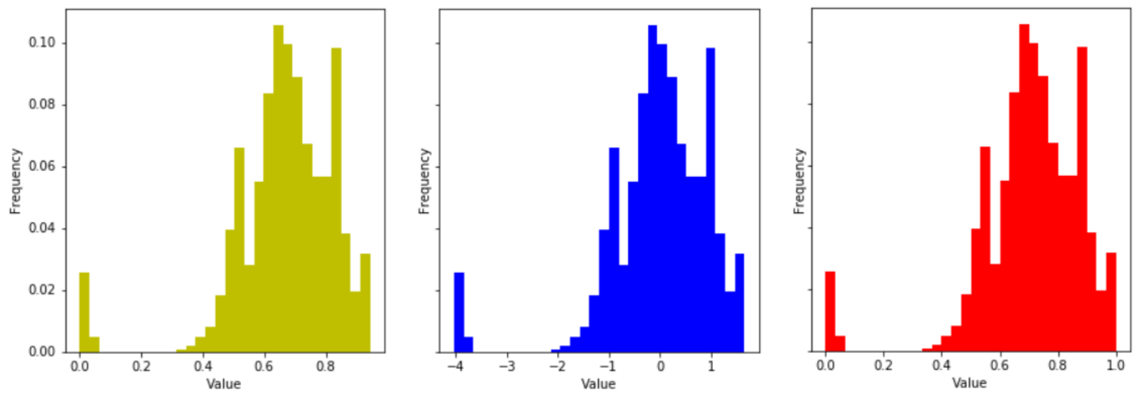


Figure 5.13: Histograms illustrating the data transformations from the raw observations (left) to the standardised values (middle) and from the standardised to the normalised scores (right).

An alternative approach is to utilise a *squashing* function which approximates the normalisation process and removes the offline requirement. Mathematical functions which exhibit the above property are known as the *sigmoid* functions. The specific formula employed in this analysis is the **logistic** function often applied in Reinforcement Learning [38] and described by the following equation:

$$x' = \frac{1}{1 + e^{-x}} \tag{5.3}$$

The process yields a more uniform distribution which is illustrated by the sequence of histogram transformations presented in Figure 5.14.

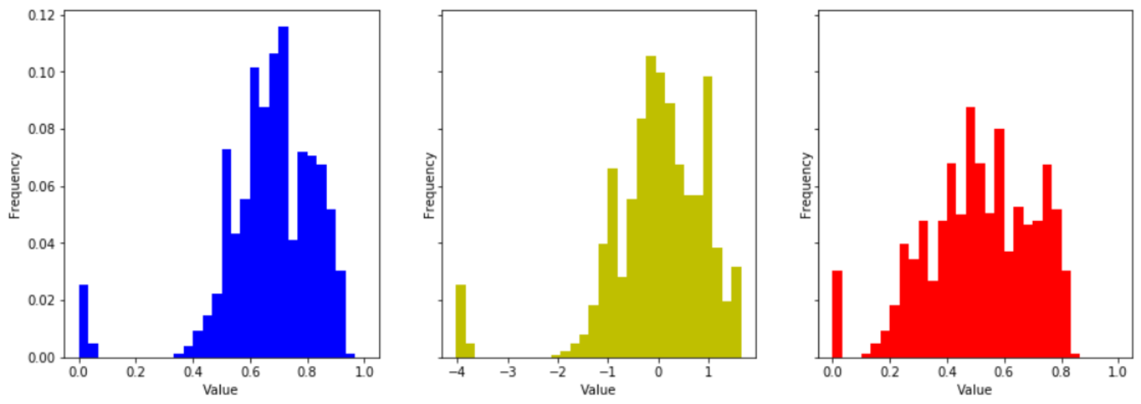


Figure 5.14: Histograms illustrating the data transformations from the raw observations (left) to the standardised values (middle) and from the standardised to the sigmoid outputs (right).

The feature extraction results obtained after the data scaling are presented in Table 5.5 for trap persisting states and in Table 5.6 for not trap persisting grandchild nodes. A significant improvement can be observed as in the case of the *Legal Actions* and *Recursive Legal Actions* the mean property of an efficient similarity measure is satisfied: $0.0 < \text{AGG}_0 < 0.5 < \text{AGG}_1 < 1.0$. Not only is the overall classification more accurate but the observations are also less skewed.

Similarity Measure	**AVG**	**STD**	**MAX**	**MIN**	**SKW**
Depth and Breadth	0.551	0.066	0.621	0.003	-2.951
Game-Influential Pieces	0.545	0.137	0.583	0.048	-3.362
Legal Actions	0.598	0.165	0.819	0.024	-0.689
Recursive Legal Actions	0.577	0.131	0.732	0.008	-1.010

Table 5.5: Feature extraction results for the scaled observation in the trap persisting pairs subset.

Similarity Measure	**AVG**	**STD**	**MAX**	**MIN**	**SKW**
Depth and Breadth	0.510	0.128	0.621	0.002	-2.832
Game-Influential Pieces	0.519	0.173	0.583	0.048	-2.352
Legal Actions	0.458	0.180	0.841	0.017	-0.361
Recursive Legal Actions	0.484	0.159	0.738	0.007	-1.116

Table 5.6: Feature extraction results for the scaled observation in the not trap persisting pairs subset.

It is important to note that despite a performance increase that feature scaling on the two subsets separately yield it cannot be utilised as it uses the knowledge of trap persistence that the similarity measures are attempting to infer. Therefore, the only valid approach is to consider all observations equally at this stage.

An issue that ought to be addressed is the offline manner of the standardisation. Although the normalisation can be performed dynamically by applying a sigmoid function, the other component of feature scaling requires the mean to be calculated (standard deviation can be derived from using the mean). A possible method involves dynamically tuning the average after seeing each sample. The procedure can be implemented using a naïve formula or a more complex weigthed moving average (WMA). The advantage of a WMA is that it favours more recent values which can be beneficial in a domain with high variance in similarity outputs in different stages of the game, e.g. main game and endgame in Chess. An example of a WMA that is commonly used in CPU scheduling is the **exponential moving average** (EMA). The EMA after observing the n^{th} sample is given the subsequent equation:

$$\text{WMA}_n = \begin{cases} \boldsymbol{x}_1 & \text{if } n = 1 \\ \alpha \boldsymbol{x}_n + (1-\alpha)\,\text{WMA}_{n-1} & \text{otherwise} \end{cases} \tag{5.4}$$

where α is the *constant smoothing factor* which is a prior placed on the rate of weight decrease and denotes a real value between 0 and 1. Other implementations of online feature scaling exist, most notably many variations were proposed by Danushka Bollegala in the paper 'Dynamic Feature Scaling for Online Learning of Binary Classifiers' [10].

The observations presented were reproducible for other hard trap states including those presented at https://www.chess.com/article/view/the-10-best-chess-traps.

An alternative to feature scaling outside of the scope of this project is **dynamic thresholding** which is inspired by the k-means clustering method frequently utilised in simple unsupervised learning models [50]. The threshold is an adaptive real value in the domain of the positional similarity function. The value is dynamically tuned as new observations are verified to have been a trap persisting in the position pair or the search trap has disappeared. The tuning is a result of incrementally improving the mean (centroid) of the samples in the given class and assigning the threshold to be the midway point between them. The described process demonstrates how the problem can be reduced to a k-means binary classification ($k = 2$) with the decision boundary behaving as the dynamic threshold (see Figure 5.15).

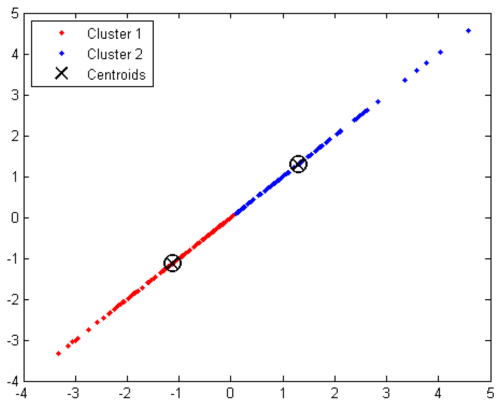


Figure 5.15: A binary classification problem on a 1D feature vector solved by a k-means model. Source: https://i.stack.imgur.com/

Chapter 6

Conclusion and Future Work

The last chapter of the report begins by recapping the introduced concepts in reference to the motivational and background material. Afterwards, a highlight of the accomplishments of the project and a relevant critique of the results is presented. Lastly, a discussion of the identified limitations and the groundwork for further research laid by the project is conducted.

6.1 Conclusions

This thesis explored the opportunities and challenges of the Monte Carlo Tree Search algorithm leveraging the developed positional similarity theory to aid the algorithm's advancement. The project initially began by exploring the key advantages of the MCTS algorithm namely random sampling and UCT. The thesis demonstrated how these beneficial attributes may become a severe limitation due to search traps in domains such as Chess. The project begun aiming to reduce the computational resource necessary for continuous trap verification and to validate the proposed solutions by implementing a Chess agent. Apart from researching the potential benefits of positional similarity, a series of other critical issues in the Trap-Adaptive Monte Carlo Chess engine were addressed along the way.

The recent advancements of programs employing the MCTS in the game of GO (AlphaGo) as well as domain-independent agents (AlphaZero) have demonstrated the potential of the algorithm. The overreaching aim of the project was to progress the MCTS in terms of a general game playing and subsequently evolve outside of the game environments into the real world. Therefore, the introduced concepts did not incorporate domain-specific or *a priori* knowledge and attempt to provide aheuristic enhancements to the algorithm instead. In addition, a number of other characteristics including the anytime performance exhibited by MCTS were preserved to ensure the GGP goal remained attainable.

Not only did the modifications include a number of semantic and syntactic error fixes but also introduce the use of positional similarity in AI. The true potential of the project and the established theoretical framework was realised upon scaling the similarity scores which had led to a significant advancement in the ahueristic prediction of trap persistence. The approach demonstrated the strength of the *Leagal Actions* measure while giving a rise at least three other viable method: *Recursive Legal Actions*, *Expandable States* and *Statistical Evaluation.* Although some of the developed solutions underperformed in empirical evaluation, a theoretical basis has been formed and a range of possible variations was proposed, paving the way to the discovery of other measures.

The findings and critical corrections led to a successful reduction in the time overhead introduced by repetitive trap verification. Combined with the explicit trap identification and search trap avoidance incorporated in the TA-MCTS implementation, a powerful variant of a Chess engine employing the MCTS was designed. The improvements proposed in the project proved modestly but substantially advantageous theoretically as well as in the benchmark testing indicating that the engine can evaluate over eight thousand nodes more per second than the previous TA-MCC variation.

Perhaps of greater importance than the MCC performance boost are the developed positional similarity and the improved trap-adaptive theoretical frameworks. Despite barely scratching the surface of the trap states problem in adversarial search spaces, the thesis aims to spark an interest in this fascinating but infrequently researched area and revive the enthusiasm in game theory initiated by AlphaGo after its historical victory.

6.2 Limitation and Future Research

Although the Monte Carlo methods provide a number of advantages, the approach may introduce some drawbacks of which most notable is the algorithm execution analysis. The MCTS performs numerous psuedorandom and statistical routines and as a result complicates the debugging process leaving the empirical evaluations as the only alternative.

Another limitation is related to the choice of domain. Although Chess is a challenging environment and enables to examine the trap states dilemma, its complexity introduces difficulties. The high branching factor, the depth of an average game and a complicated set of rules produce a number of analysis and development problems as a large number of combinations ought to be considered at each step.

The thesis addresses the issue of trap verification computation cost, however, a number of other challenges remain unaccounted for in the model. An important notion not covered is a *soft trap* which can be fairly easily distinguished when a sequence of moves is encountered at the selection or stimulation phase that leads to a node of comparably low score instead of a check mate condition. Moreover, the project focuses on the more impactful shallow traps and disregards the deep traps that a skilful opponent may utilise as well. Finally, the engine should also be able to favour states in which the opponent is at risk, hence, employ the trap identification algorithm in its own advantage. The problem posed by the idea is similar to the exploration-exploitation trade-off where a balance between trap avoidance and preferring dangerous states for the adversary must be struck.

The scope of search traps is a prominent the direction future research in the area of trap adaptivness may progress. An abundance of studies and heuristic approaches has been developed in the game of Chess to gradually gain an advantage over the opponent. Another promising area are the dynamic methods such as online parameter tuning [55] and online feature scaling [10].

To summarise, considering the recent developments in the field of Neural Networks, such as the AlphaZero [31], the ahueristic algorithms will no longer provide a viable alternative. It is possible that the training phase will become sufficiently fast to outweigh the overhead of the slower aheuristic algorithms. As demonstrated by the MCTS-Minimax hybrids, it is feasible to successfully integrate seemingly contradicting ideas and perhaps finally attain an anytime GGP capable agent.

Appendices

Source code: uctsearch.cpp

```
#include <cmath>
#include <cstring>
#include <fstream>
#include <iostream>
#include <sstream>
#include <ostream>
#include <algorithm>

#include "ucioption.h"
#include "misc.h"
#include "search.h"
#include "montecarlotreenode.h"
#include "position.h"
#include "movegen.h"
#include "move.h"

#include "rkiss.h"
#include "similarity.h"

static RKISS rk;

using namespace std;

namespace {
    bool StopOnPonderhit, StopRequest, QuitRequest;
    bool whiteToMove;
    SearchLimits Limits;
    int depth;
    int UCIMultiPV;
    unsigned int iterations;

    // Trap Adaptiveness
    Position *prevPosBlanc, *prevPosNoir;
    MoveStack tlistBlanc[MAX_MOVES], tlistNoir[MAX_MOVES];
    MoveStack *lastTrapBlanc, *lastTrapNoir;

    // Time Management
    int searchStartTime;
```

```
        int thinkingTime;
        const int timeRate = 20;
        int current_search_time() {
            return get_system_time() - searchStartTime;
        }

        void uct_poll(MonteCarloTreeNode * root) {
            if (log(iterations) > depth)
                cout << "info depth " << ++depth << endl;

            root->printMultiPv(depth, iterations, current_search_time() ...
                / 1000, whiteToMove, UCIMultiPV);

            //  Poll for input
            if (input_available()) {
                string command;
                if (!getline(cin, command) || command == "quit") {
                    Limits.ponder = false;
                    QuitRequest = StopRequest = true;
                    return;
                }
                else if (command == "stop") {
                    Limits.ponder = false;
                    StopRequest = true;
                }
                else if (command == "ponderhit") {
                    Limits.ponder = false;
                    if (StopOnPonderhit) StopRequest = true;
                }
            }

            // Stop if we have no thinking time left
            if (!Limits.infinite && current_search_time() >= ...
                thinkingTime) {
                StopRequest = true;
                Limits.ponder = false;
            }
        }
    }

    bool uct(Position& pos, const SearchLimits& limits){
          // Read UCI options
        UCIMultiPV = Options["MultiPV"].value<int>();

        // Initialize variables for the new search
        Limits = limits;
        whiteToMove = (pos.side_to_move()==WHITE);
        StopOnPonderhit = StopRequest = QuitRequest = false;
        depth = 10;
        iterations = 0;
        thinkingTime = Limits.time / timeRate;
        searchStartTime = get_system_time();
        MonteCarloTreeNode * root = new MonteCarloTreeNode(MOVE_NONE, ...
            NULL, VALUE_ZERO);
        MonteCarloTreeNode * selected0;
        MonteCarloTreeNode * selected1;
        MonteCarloTreeNode * expanded0;
        MonteCarloTreeNode * expanded1;
        double result, sim;

        if (whiteToMove) {
            sim = similarity<LEGAL_MOVES>(&pos, prevPosBlanc);
            prevPosBlanc = new Position(pos, pos.thread());

            // generate trap list if no previous or on a probabilistic basis
            if (!lastTrapBlanc || rk.rand<unsigned int>() % 100 > 100 * ...
                sim) {
                 lastTrapBlanc = generate<TRAP>(&pos, tlistBlanc);
            }
        }
        else {
            sim = similarity<LEGAL_MOVES>(&pos, prevPosNoir);
            prevPosNoir = new Position(pos, pos.thread());

            // generate trap list if no previous or on a probabilistic basis
            if (!lastTrapNoir || rk.rand<unsigned int>() % 100 > 100 * ...
                sim) {
                 lastTrapNoir = generate<TRAP>(&pos, tlistNoir);
            }
        }

        while(!StopRequest) {
            selected0 = root->UCT_select(&pos);
            expanded0 = selected0->UCT_expand(&pos);
            selected1 = expanded0->UCT_select(&pos);
            expanded1 = selected1->UCT_expand(&pos);
            result = expanded1->simulate(sim, &pos);
            expanded1->update(result, &pos);
            if (iterations++ % 1000 == 0)
                 uct_poll(root);
        }

        root->printMultiPv(depth, iterations, current_search_time(), ...
            whiteToMove, UCIMultiPV);
        cout << "bestmove " << move_to_uci(root->bestChild()->lastMove, ...
            false) << endl;

        delete root;
        return !QuitRequest;
    }

    bool trapCheck(Move m) {
        if (whiteToMove) {
            for (MoveStack *i = tlistBlanc; i != lastTrapBlanc; i++) {
                if (m == i->move) return true;
            }
        }
        else {
            for (MoveStack *i = tlistNoir; i != lastTrapNoir; i++) {
                if (m == i->move) return true;
            }
        }
        return false;
    }
```

Source code: montecarlotreenode.cpp

```
#include "montecarlotreenode.h"

#include <cmath>
#include <iostream>
#include <sstream>
#include <vector>
#include <algorithm>

#include "movegen.h"
#include "position.h"
#include "rkiss.h"
#include "evaluate.h"
#include "uctsearch.h"

static RKISS rk;

// Class Constructor, initializes node variables
MonteCarloTreeNode::MonteCarloTreeNode(Move move, MonteCarloTreeNode ...
    * parentNode, Value score) {
     maxMoves = MAX_MOVES;
     visits = 0;
     totalValue = 0;
     parent = parentNode;
     lastMove = move;
     heuristicScore = score;
     simcounter = 1;
}

MonteCarloTreeNode * MonteCarloTreeNode::UCT_select(Position * ...
    rootPosition) {
     MonteCarloTreeNode * cur = this;
     MonteCarloTreeNode * chosen = this;
     double bestVal;
     Position * Pos = getTreeNodePosition(this, rootPosition);
     bool whiteToMove;
     if (Pos->side_to_move() == WHITE)
         whiteToMove = true;
     else whiteToMove = false;
```

```
        while (cur->children.size() > 0) {
            bestVal = -INT_MAX;
            std::vector<MonteCarloTreeNode*>::iterator child;

            // select this node, if not every legal move was expanded yet
            if (cur->children.size() < cur->maxMoves) {
                delete Pos;
                return cur;
            }

            // every legal move was expanded, step into the next node ...
                according to UCT
            double uctVal;

            for (child = cur->children.begin(); child < ...
                cur->children.end(); child++){
                double winningrate = ((*child)->totalValue / ...
                    (*child)->visits);
                if (Pos->side_to_move() == BLACK)
                    winningrate = 1 - winningrate;
                uctVal = winningrate + sqrt(2* log( (double) ...
                    cur->visits)/ (*child)->visits) + 0.001 * ...
                    (((int)(*child)->heuristicScore) / (*child)->visits);
                if (uctVal > bestVal) {
                    chosen = &(**child);
                    bestVal = uctVal;
                }
            }

            Pos->do_setup_move(chosen->lastMove);
            whiteToMove = !whiteToMove;
            cur = chosen;
        }
        delete Pos;
        return cur;
    }

    MonteCarloTreeNode * MonteCarloTreeNode::UCT_expand(Position * ...
        rootPosition) {
        MoveStack mlist[MAX_MOVES];
        MoveStack * lastLegal;

        // Generate all legal moves
        Position * pos = getTreeNodePosition(this, rootPosition);

        if (pos->is_really_draw() || pos->is_mate()) {
            delete pos;
            return this;
        }

        MoveStack* last = generate<MV_LEGAL>(*pos, mlist);
        if (maxMoves == (size_t) MAX_MOVES)
            maxMoves = last - mlist;

        if (maxMoves <= (children.size())) {
            delete pos;
            return this;
        }

        lastLegal = mlist+children.size();
        Value margin;
        Value score = VALUE_ZERO;
        pos->do_setup_move(lastLegal->move);
        score -= evaluate(*pos, margin);
        delete pos;
        return addChild(lastLegal->move, score);
    }

    bool stmHasDecisiveMove(Position * pos, MoveStack * mlist, MoveStack ...
       * last) {
        StateInfo st;
        CheckInfo ci(*pos);

        // On the huge benefit of decisive moves
        for (MoveStack* cur = mlist; cur != last; cur++) {
            if (pos->move_gives_check(cur->move)) {
                pos->do_move(cur->move, st, ci, true);
                if (pos->is_mate()) {
                    pos->undo_move(cur->move);
                    return true;
                }
                pos->undo_move(cur->move);
            }
        }
        return false;
    }

    double MonteCarloTreeNode::simulate(double sim, Position * ...
       rootPosition) {
        simcounter = exp(simcounter - 1 + 0.001);
        MoveStack mlist[MAX_MOVES];
        MoveStack* last;
        int numMoves, index;
        Position * pos = getTreeNodePosition(this, rootPosition);

        if (pos->is_draw()) {
            delete pos;
            return simcounter*0.5;
        }

        if (pos->is_mate()) {
            if (pos->side_to_move() == WHITE) {
                delete pos;
                return BLACK_MATES_IN_ONE;
            }
            else {
                delete pos;
                return WHITE_MATES_IN_ONE;
            }
        }

        last = generate<MV_LEGAL>(*pos, mlist);
        numMoves = last - mlist;

        // PRNG sequence should be non deterministic
        for (int i = abs(get_system_time() % 50); i > 0; i--)
            rk.rand<unsigned>();

        while (!pos->is_draw() && !pos->is_mate()) {

            // Generate all legal moves
            last = generate<MV_LEGAL>(*pos, mlist);
            numMoves = last - mlist;

            // stalemate positions are not recognized by pos->is_draw()
            if (numMoves == 0) {
                delete pos;
                return 0.5;
            }

            // Check for decisive moves
            if (stmHasDecisiveMove(pos, mlist, last)) {
                if (pos->side_to_move() == WHITE) {
                    delete pos;
                    return 1;
                }
                else {
                    delete pos;
                    return 0;
                }
            }

            StateInfo st;
            if (rk.rand<unsigned int>() % 10 < 6) {
                index = pickMoveBySee(mlist, last, pos);
                pos->do_move(mlist[index].move, st);
            }
            else {
                index = rk.rand<unsigned int>() % numMoves;
                pos->do_move(mlist[index].move, st);
            }

            // Check for trap (if this was in the old trap list,
            // backpropagate it with probability p = similarity)
            if (trapCheck(mlist[index].move)) {
                if (rk.rand<unsigned int>() % 100 < 100 * sim) {
                    if (pos->side_to_move() == WHITE) {
                        delete pos;
                        return simcounter*1;
                    } else {
                        delete pos;
                        return 0;
                    }
                }
            }


            // play chosen move
            pos->undo_move(mlist[index].move);
            pos->do_setup_move(mlist[index].move);
        }

        if (pos->is_mate()) {
            if (pos->side_to_move() == WHITE) {
                delete pos;
                return 0;
            }
            delete pos;
            return simcounter*1;
        }
        delete pos;
        return simcounter*0.5;
    }

    void MonteCarloTreeNode::normalUpdate(double value) {
        if (!whiteWins(totalValue) && !blackWins(totalValue))
            totalValue += value;
        visits++;

        if (parent) parent->normalUpdate(value);
    }

    void MonteCarloTreeNode::updateKnownWin(double value, bool ...
        whiteToMove) {
        visits++;

        if (! ( (whiteToMove && whiteWins(value) && totalValue > value)
                || (!whiteToMove && blackWins(value) && totalValue < ...
                   value)))
            totalValue = value;

        if (!parent)
            return;

        // If the side to move is proven losing, the parent node is ...
           proven winning
        if ((!whiteToMove && whiteWins(value))) {
            parent->updateKnownWin(value, !whiteToMove);
            return;
        }

        if ((whiteToMove && blackWins(value))) {
            parent->updateKnownWin(value, !whiteToMove);
            return;
        }

        //The parent's node is a proven losing, if all its children are ...
            proven winning
        std::vector<MonteCarloTreeNode*>::iterator child;
        bool parentKnownLoss = true;
        int farthestLoss = 0;
        if (blackWins(value))
            farthestLoss = BLACK_MATES_IN_ONE;
        else if (whiteWins(value))
            farthestLoss = WHITE_MATES_IN_ONE;

        if (parent->children.size() == parent->maxMoves) {
            for (child = parent->children.begin(); child < ...
                parent->children.end(); child++) {
                if ((whiteWins(value) && !whiteWins((*child)->totalValue))
                        || (blackWins(value) && ...
                            !blackWins((*child)->totalValue))) {
                    parentKnownLoss = false;
                    break;
                }
                else if (abs((*child)->totalValue) < abs(farthestLoss)) {
                    farthestLoss = (*child)->totalValue;
                }
            }
        } else {
            parentKnownLoss = false;
        }

        if (parentKnownLoss) {
            if (!whiteToMove)
                parent->updateKnownWin(farthestLoss + 1, false);
            else
                parent->updateKnownWin(farthestLoss - 1, true);
        }
        else {
            if (blackWins(value))
                parent->normalUpdate(0);
            else if (whiteWins(value))
                parent->normalUpdate(1);
        }
    }

    void MonteCarloTreeNode::update(double value, Position * root) {
        if (value < 0 || value > 1) {
            bool whiteToMove;
            if (movesFromRoot().size() % 2 == 0)
                whiteToMove = (root->side_to_move() == WHITE) ? true : ...
                    false;
            else
                whiteToMove = (root->side_to_move() == WHITE) ? false : ...
                    true;
            updateKnownWin(value, whiteToMove);
        } else {
            this->normalUpdate(value);
        }
    }

    MonteCarloTreeNode * MonteCarloTreeNode::bestChild() {
        int maxVisits = 0;
        MonteCarloTreeNode * curBestChild = children[0];
        for (unsigned int i = 0; i < children.size(); i++) {
            if (children[i]->visits > maxVisits) {
                maxVisits = children[i]->visits;
                curBestChild = children[i];
            }
        }
        return curBestChild;
    }
```

Source code: similarity.cpp

```
#include <vector>
#include <iostream>
#include <fstream>

#include "similarity.h"
#include "position.h"
#include "movegen.h"
#include "move.h"

template<SimMethod>
double intersectionRec(Position* curPos, Position* prevPos, ...
    std::vector<MoveStack> v1, std::vector<MoveStack> v2);
double intersection(std::vector<MoveStack> v1, ...
    std::vector<MoveStack> v2);
std::vector<MoveStack> getMoves(Position* pos);

/* * * SIMILARITY MEASURES * * */
template<>
double similarity<CONSTANT>(Position* curPos, Position* prevPos) {
     return CONST_SIM;
}

template<>
double similarity<DEPTH_BREADTH>(Position* curPos, Position* ...
    prevPos) {
     double curDepth  = curPos->startpos_ply_counter(), prevDepth = ...
        prevPos->startpos_ply_counter();
     double curBreadth  = getMoves(curPos).size(), prevBreadth = ...
        getMoves(prevPos).size();

     double depthComponent;
     if (curDepth + prevDepth == 0)
         depthComponent = 1;
     else
         depthComponent =  1 - abs(curDepth - prevDepth) / (curDepth ...
            + prevDepth);

     double breadthComponent;
```

```
        if (curBreadth + prevBreadth == 0)
            breadthComponent = 1;
        else
            breadthComponent = 1 - abs(curBreadth - prevBreadth) / ...
                (curBreadth + prevBreadth);

        return (DEPTH_WEIGHT*depthComponent + ...
            BREADTH_WEIGHT*breadthComponent) / (DEPTH_WEIGHT + ...
            BREADTH_WEIGHT);
    }

    template<>
    double similarity<INFL_PIECES>(Position* curPos, Position* prevPos) {
        if (!prevPos) return DEFAULT_SIM;
        double sum = 0, inter = 0;
        int curPieceBlanc, prevPieceBlanc, curPieceNoir, prevPieceNoir;
        for (PieceType pt = PIECE_TYPE_NONE; pt <= KING; pt++) {
            curPieceBlanc = curPos->piece_count(WHITE, pt);
            prevPieceBlanc = prevPos->piece_count(WHITE, pt);

            curPieceNoir = curPos->piece_count(BLACK, pt);
            prevPieceNoir = prevPos->piece_count(BLACK, pt);

            sum += std::max(curPieceBlanc, prevPieceBlanc) + ...
                std::max(curPieceNoir, prevPieceNoir);
            inter += std::min(curPieceBlanc, prevPieceBlanc) + ...
                std::min(curPieceNoir, prevPieceNoir);
        }
        return inter / sum;
    }

    template<>
    double similarity<LEGAL_MOVES>(Position* curPos, Position* prevPos) {
        if (!prevPos) return DEFAULT_SIM;
        std::vector<MoveStack> curMoves = getMoves(curPos);
        std::vector<MoveStack> prevMoves = getMoves(prevPos);

        if (curMoves.size()==0 && prevMoves.size()==0) return 1.0;

        double uni = curMoves.size() + prevMoves.size();
        double inter = intersection(curMoves, prevMoves);
        uni = uni - inter;
        return inter / uni;
    }

    template<>
    double similarity<REC_LEGAL_MOVES>(Position* curPos, Position* ...
        prevPos) {
        if (!prevPos) return DEFAULT_SIM;
        std::vector<MoveStack> curMoves = getMoves(curPos);
        std::vector<MoveStack> prevMoves = getMoves(prevPos);
        double inter = 0,
               uni   = curMoves.size() + prevMoves.size();

        for (long long i = curMoves.size()-1; i >= 0; i--) {
            if (curMoves.empty()) break;
            for (long long j = prevMoves.size()-1; j >= 0; j--) {
                if (prevMoves.empty()) break;
                if (curMoves[i].move == prevMoves[j].move) {
                    curMoves.erase(curMoves.begin()+i);
                    prevMoves.erase(prevMoves.begin()+j);
                    inter++;
                    break;
                }
            }
        }

        double simMismatches = intersectionRec<LEGAL_MOVES>(curPos, ...
            prevPos, curMoves, prevMoves);
        uni = uni - inter;
        return (inter + simMismatches) / uni;
    }

    template<>
    double similarity<EXPANDABLE_STATES>(Position* curPos, Position* ...
        prevPos) {
        if (!prevPos) return DEFAULT_SIM;
        std::vector<MoveStack> curMoves = getMoves(curPos);
        std::vector<MoveStack> prevMoves = getMoves(prevPos);

        Position *p1, *p2;
        StateInfo st1, st2;
        double inter = 0, uni = curMoves.size() + prevMoves.size();

        for (unsigned int i = 0; i < curMoves.size(); i++) {
            for (unsigned int j = 0; j < prevMoves.size(); j++) {
                p1 = new Position(*curPos, curPos->thread());
                p2 = new Position(*prevPos, prevPos->thread());
                p1->do_move(curMoves[i].move, st1);
                p2->do_move(prevMoves[j].move, st2);

                if (p1->get_key() == p2->get_key()) {
                    prevMoves.erase(prevMoves.begin()+j);
                    inter++;
                    break;
                }
            }
        }

        uni = uni - inter;
        return inter / uni;
    }
```

```
template<>
double similarity<REC_EXPANDABLE_STATES>(Position* curPos, Position* ...
    prevPos) {
     if (!prevPos) return DEFAULT_SIM;
     std::vector<MoveStack> curMoves = getMoves(curPos);
     std::vector<MoveStack> prevMoves = getMoves(prevPos);

     double inter = 0, uni = curMoves.size() + prevMoves.size();

     Position *p1, *p2;
     StateInfo st1, st2;
     for (long long i = curMoves.size()-1; i >= 0; i--) {
          if (curMoves.empty() || prevMoves.empty()) break;
          for (long long j = prevMoves.size()-1; j >= 0; j--) {
               p1 = new Position(*curPos, curPos->thread());
               p2 = new Position(*prevPos, prevPos->thread());
               p1->do_move(curMoves[i].move, st1);
               p2->do_move(prevMoves[j].move, st2);

               if (p1->get_key() == p2->get_key()) {
                    curMoves.erase(curMoves.begin()+i);
                    prevMoves.erase(prevMoves.begin()+j);
                    inter++;
                    break;
               }
          }
     }

     double simMismatches = ...
         intersectionRec<EXPANDABLE_STATES>(curPos, prevPos, curMoves, ...
         prevMoves);
     uni = uni - inter;
     return (inter + simMismatches) / uni;
}

/* * * HELPER METHODS * * */
std::vector<MoveStack> getMoves(Position* pos) {
     MoveStack moveList[MAX_MOVES];
     std::vector<MoveStack> vec;

     MoveStack *last = generate<MV_LEGAL>(*pos, moveList);
     for (MoveStack *i = moveList; i != last; i++)
          vec.push_back(*i);

     return vec;
}

double intersection(std::vector<MoveStack> v1, ...
    std::vector<MoveStack> v2) {
     double inter = 0;
     for (unsigned int i = 0; i < v1.size(); i++) {
          for (unsigned int j = 0; j < v2.size(); j++) {
                if (v1[i].move == v2[j].move) {
                    v2.erase(v2.begin()+j);
                    inter++;
                    break;
                }
            }
        }
        return inter;
    }

    template<SimMethod simMethod>
    double intersectionRec(Position* curPos, Position* prevPos, ...
        std::vector<MoveStack> v1, std::vector<MoveStack> v2) {
        Position *p1, *p2;
        StateInfo st1, st2;
        double inter = 0;

        for (unsigned int i = 0; i < v1.size(); i++) {
            for (unsigned int j = 0; j < v2.size(); j++) {
                p1 = new Position(*curPos, curPos->thread());
                p2 = new Position(*prevPos, prevPos->thread());
                p1->do_move(v1[i].move, st1);
                p2->do_move(v2[j].move, st2);

                if (p1->get_key() == p2->get_key()) {
                    inter += REC_INCREMENT;
                    break;
                }

                double sim = similarity<simMethod>(p1, p2);
                if (sim > ACC_THRESHOLD) {
                    inter += REC_INCREMENT; // alternatively: sim * ...
                        REC_INCREMENT
                    break;
                }
            }
        }
        return inter;
    }
```

Source code: similarity_test.cpp

```
#include <vector>
#include <c++/cmath>
#include <chrono>

#include <iostream>
#include <fstream>
#include <ctime>
#include <regex>
#include <direct.h>

#include "similarity_test.h"
#include "similarity.h"
#include "movegen.h"
#include "move.h"

void fenTest();
void childTest();
void unitTest();
void manTest();

std::string getTimeStamp();

double simFromKey(int key, Position *pos1, Position *pos2);
double simFromKey(int key, std::string fen1, std::string fen2);

std::string trapPersistence(Position *pos1, Position *pos2);
std::string trapPersistence(std::string fen1, std::string fen2);
bool isTrap(Position *pos);


/* Test wrapper function. Loop over possible test cmds until 'exit' ...
   found. */
void similarityTest() {

    std::string cmd="";
    while (true) {
        getline(std::cin, cmd);
        if (cmd == EXIT_CMD) break;
```

```
            else if (cmd==FEN_CMD) fenTest();
            else if (cmd==MANUAL_CMD) manTest();
            else if (cmd==UNIT_CMD) unitTest();
            else if (cmd==CHILD_CMD) childTest();
            else std::cout << "[ERROR] Invalid command." << std::endl;
        }
    }

    /* * * UNIT TESTING * * */
    void unitTest() {
        std::ifstream testFile;
        std::ofstream resultFile;
        std::string resultName = "..\\test\\result_unit_" + ...
            getTimeStamp() + ".csv";
        resultFile.open(resultName);
        if (!resultFile) {
            std::cout << "[ERROR] Unable to create the result file." ;
            std::cout << std::endl;
            return ;
        }

        resultFile << "Similarity ...
            Measure,Symmetric,Normalised,Accurate,Time\n";

        std::string simNames[] = {
                "CONSTANT",
                "DEPTH_BREADTH",
                "INFL_PIECES",
                "LEGAL_MOVES",
                "REC_LEGAL_MOVES",
                "EXPANDABLE_STATES",
                "REC_EXPANDABLE_STATES"
        };

        std::string inStr = "", outStr, fen1, fen2, expSim;
        bool isSym, isNorm, isAcc;
        double sim, sim2;
        long long avgTime;
        int count;

        for (int simMethod = CONSTANT; simMethod < ...
            REC_EXPANDABLE_STATES; simMethod++) {
            testFile.open("..\\test\\test_suite.in");
            if (!testFile) {
                std::cout << "[ERROR] Unable to open the test file." ;
                std::cout << std::endl;
                return ;
            }

            isSym = true; isNorm = true; isAcc = true; avgTime = 0.0; ...
                count = 0;
            while( getline(testFile, inStr) ) {
                std::stringstream ss(inStr);
                getline(ss, fen1, ',');
                getline(ss, fen2, ',');
                getline(ss, expSim);

                // Time check
                auto sTime = std::chrono::high_resolution_clock::now();
                sim = simFromKey(simMethod, fen1, fen2);
                auto eTime = std::chrono::high_resolution_clock::now();
                avgTime += ...
                    std::chrono::duration_cast<std::chrono::nanoseconds>(eTime-sTime).cou

                // Symmetry check
                if (isSym) {
                    sim2 = simFromKey(simMethod, fen2, fen1);
                    if (sim != sim2) isSym = false;
                }

                // Normalised check
                if (isNorm && (sim < 0 || sim > 1)) isNorm = false;

                // Accuracy check
                if (isAcc && sim != atof(expSim.c_str())) isAcc = false;

                count++;
            }
            resultFile << simNames[simMethod] << "," << isSym << "," << ...
                isNorm << "," << isAcc << ","
                << (avgTime/count) << "\n";
            testFile.close();
        }
        resultFile.close();

        std::cout << "[INFO] Results of automatic tests exported to: " ...
            << resultName << std::endl;
    }

    /* * * MANUAL TESTING * * */
    void manTest() {
        std::string keyStr;
        int key = 0;
        double sim = DEFAULT_SIM;
        while(true) {
            std::cout << "[INFO] Select similairty measure key (or '9' ...
                to quit):\n";
            std::cout <<
                          "    Name                          |key\n" <<
                          "    ------------------------------    \n" <<
                          "    CONSTANT                      | 0 \n" <<
                          "    DEPTH_BREADTH                 | 1 \n" <<
                          "    INFL_PIECES                   | 2 \n" <<
                          "    LEGAL_MOVES                   | 3 \n" <<
                          "    REC_LEGAL_MOVES          |  4 \n" <<
                          "    EXPANDABLE_STATES        |  5 \n" <<
                          "    REC_EXPANDABLE_STATES |  6 \n";
            getline(std::cin, keyStr);
            key = atoi(keyStr.c_str());
            if (key == 9) {
                break;
            }
            if (keyStr.size()>1 || key < CONSTANT || key > ...
                REC_EXPANDABLE_STATES) {
                std::cout << "[ERROR] Invalid similairty key" << std::endl;
            }
            else {
                std::string fen1, fen2;
                std::cout << "[INFO] Enter FEN one: ";
                getline(std::cin, fen1);
                std::cout << "[INFO] Enter FEN two: ";
                getline(std::cin, fen2);

                sim = simFromKey(key, fen1, fen2);
                break;
            }
        }

        if (sim == INVALID_FEN) std::cout << "[ERROR] Invalid FEN ...
            position(s).\n";
        else std::cout << "[INFO] Similairty is: " << sim << "\n\n";
    }

    /* * * POSITIONAL: FEN TESTING * * */
    void fenTest() {
        std::ifstream testFile;
        std::ofstream resultFile;
        std::string resultName = "..\\test\\result_auto_" + ...
            getTimeStamp() + ".csv";

        testFile.open("..\\test\\auto_test_set.in");
        resultFile.open(resultName);

        if (!testFile || !resultFile) {
            std::cout << "[ERROR] Unable to open the required file(s)." ;
            std::cout << std::endl;
            return ;
        }

        resultFile << "CONSTANT,DEPTH_BREADTH,INFL_PIECES,"
                   << "LEGAL_MOVES,REC_LEGAL_MOVES,EXPANDABLE_STATES,"
                   << "REC_EXPANDABLE_STATES,"
                   << "trap_presence\n";

        std::string inStr = "", outStr, prevFen, curFen;
        int fenCount;
```

```
        double sim;
        while(true) {
            getline(testFile, inStr);
            fenCount = atoi(inStr.c_str());
            if (fenCount < 2) break;            // insufficient number ...
               of positions to calculate the similairty

            prevFen = "";
            for (int i=0; i < fenCount; i++) {
                getline(testFile, curFen);
                if (prevFen != "") {
                    outStr = "";
                    for (int simMethod = CONSTANT; simMethod <= ...
                       REC_EXPANDABLE_STATES; simMethod++) {
                        sim = simFromKey(simMethod, curFen, prevFen);
                        outStr += std::to_string(sim) + ",";
                    }
                    resultFile << outStr << trapPersistence(curFen, ...
                       prevFen) << std::endl;
                }
                prevFen = curFen;
            }
            resultFile << std::endl;
        }
        testFile.close();
        resultFile.close();

        std::cout << "[INFO] Results of automatic tests exported to: " ...
           << resultName << std::endl;
    }

    /* * * POSITIONAL: CHILD NODES TESTING * * */
    void childTest() {
        std::ifstream testFile;
        testFile.open("..\\test\\child_test_set.in");
        if (!testFile ) {
            std::cout << "[ERROR] Unable to open the input file(s)." ;
            std::cout << std::endl;
            return ;
        }

        std::string resultDir = "..\\test\\result_child_" + getTimeStamp();
        const char *path = resultDir.c_str();
        int check = mkdir(path);
        if (check) {
            std::cout << "[ERROR] Unable to create the result ...
               directory." ;
            std::cout << std::endl;
            return ;
        }

        std::string rootFen;
```

```
std::ofstream resultFile;
double sim;
int fenCounter = 1;

while(getline(testFile, rootFen)) {
    std::string resultName = resultDir + "\\fen_" + ...
        std::to_string(fenCounter) + ".csv";
    resultFile.open(resultName);

    resultFile << "move1,move2,"
               << "CONSTANT,DEPTH_BREADTH,INFL_PIECES,LEGAL_MOVES,"
               << ...
                "REC_LEGAL_MOVES,EXPANDABLE_STATES,REC_EXPANDABLE_STATES,"
               << "trap\n";

    Position *rootPos  = new Position(rootFen, false, 0);
    if (rootPos->get_key()==0) continue;

    // print first row - the root position itself
    resultFile << "root,root,1,1,1,1,1,1,1," << (isTrap(rootPos) ...
        ? 1 : 0) << std::endl;

    StateInfo st1, st2;
    std::vector<MoveStack> moves = getMoves(rootPos);
    if (moves.empty()) continue;

    // iterate over all grand-children to analyse trap existence ...
        in correspondence to similairty measures
    for (MoveStack ms : moves) {

        Position *childPos  = new Position(*rootPos, ...
            rootPos->thread());        // calculate child position
        childPos->do_move(ms.move, st1);
        std::vector<MoveStack> childMoves = getMoves(childPos);

        for (MoveStack cms : childMoves) {

            Position *grandPos  = new Position(*childPos, ...
                childPos->thread());  // calculate grandchild ...
                position
            grandPos->do_move(cms.move, st2);
            resultFile << move_to_uci(ms.move, false) << "," << ...
                move_to_uci(cms.move, false) << ",";

            for (int i = CONSTANT; i <= REC_EXPANDABLE_STATES; ...
                i++) {
                 sim = simFromKey(i, grandPos, rootPos);
                 resultFile << std::to_string(sim) + ",";
            }
            resultFile << (isTrap(grandPos) ? 1 : 0) << std::endl;
        }
    }
```

```
            fenCounter++;
            resultFile.close();
        }
        testFile.close();
        std::cout << "[INFO] Results of children tests exported to " << ...
            resultDir << std::endl;
    }
```

Bibliography

[1] L. V. Allis, “Searching for Solutions in Games and Artificial Intelligence,” *PhD thesis, University of Limburg, Maastricht, The Netherlands*, p. 3, 1944.

[2] E. Alpaydin, “Introduction to Machine Learning,” *The MIT Press, London*, pp. 109–110, 2017.

[3] I. Althöfer, C. Donninger, U. Lorenz, and V. Rottmann, “On Timing, Permanent Brain and Human Intervention,” *Advances in Computer Chess 7, Computer Science Department, Universiteit Maastricht*, 1994.

[4] O. Arenz, “Monte-Carlo Chess. BSc thesis,” *Technische Universitat Darmstadt, Darmstadt, Germany*, 2012.

[5] P. Aue, N. Cesa-Bianchi, and P. Fischer, “Finite-time Analysis of the Multiarmed Bandit Problem,” *Machine Learning*, vol. 47, p. 235–256, 2002.

[6] B. C. H. Aw, “Trap-adaptive Monte Carlo Tree Search Algorithms,” *Imperial College London, London, England*, 2017.

[7] H. Baier and P. D. Drake, “The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go,” *IEEE Trans. Comp. Intell. AI Games*, vol. 2, p. 303, 2010.

[8] H. Baier and M. H. M. Winands, “MCTS-Minimax Hybrids,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, pp. 167–179, 2015.

[9] M. Blume. Arena Chess GUI: Main Page. [Online]. Available: http://www.playwitharena.com/

[10] D. Bollegala, “Dynamic Feature Scaling for Online Learning of Binary Classifiers,” *University of Liverpool*, pp. 3–6, 2014.

[11] C. Browne, E. Poweley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A Survey of Monte Carlo Tree Search Methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 29–39, March 2012.

[12] M. Campbell, A. Hoane, and F. Hsu, “Deep Blue,” *Artificial Intelligence 134*, pp. 58–65, 2002.

[13] CCRL: Computer Chess Rating Lists. (2019) ”CCRL 40/40 Rating List”. [Online]. Available: http://www.computerchess.org.uk/ccrl/4040/rating_list_all.html

[14] G. Chaslot, M. Winands, J. Uiterwijk, and H. van den Herik, “Progressive Strategies for Monte-Carlo Tree Search,” *New Mathematics and Natural Computation*, vol. 4, 2008.

[15] Chess Programming Wiki. (2018) En passant. [Online]. Available: https://www.chessprogramming.org/En_passant

[16] ——. (2018) Forsyth-Edwards Notation. [Online]. Available: https://www.chessprogramming.org/Forsyth-Edwards_Notation#cite_note-1

[17] ——. (2018) Late Move Reductions. [Online]. Available: https://www.chessprogramming.org/Late_Move_Reductions

[18] ——. (2018) Opening Book. [Online]. Available: https://www.chessprogramming.org/Opening_Book

[19] D. Cohen, “On Holy Wars and a Plea for Peace,” *Computer, IEEE Computer Society*, vol. 14, pp. 48 – 54, 1981.

[20] N. Companez and A. Aleti, “Can Monte-Carlo Tree Search learn to sacrifice?” *Journal of Heuristics*, pp. 2–58, 2016.

[21] P.-A. Coquelin and R. Munos, “Bandit Algorithms for Tree Search,” *UAI, Vancouver, Canada*, p. 67–74, 2007.

[22] CPlusPlus Reference. (2019) Date and Time Utilities. [Online]. Available: https://en.cppreference.com/w/cpp/chrono/

[23] S. Dasgupta, C. Papadimitriou, and U. Vazirani, “Algorithms,” *McGraw-Hill Higher Education, New York*, pp. 15–16, 2008.

[24] V. Dwarampudi, S. S. Dhillon, J. Shah, and N. Kanigicharla, “Comparative study of the Pros and Cons of Programming languages Java, Scala, C++, Haskell, VB .NET, AspectJ, Perl, Ruby, PHP & Scheme,” *Team 11 COMP6411-S10 Term Report*, 2010.

[25] S. J. Edwards. (1994) Portable Game Notation Specification and Implementation Guide. [Online]. Available: http://www.thechessdrum.net/PGN_Reference.txt

[26] S. Gelly and D. Silver, “Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go,” *Artificial Intelligence*, vol. 175, p. 1856–1875, 2011.

[27] M. Genesereth, N. Love, and B. Pell, “General Game Playing,” *Overview of the AAAI Competition*, pp. 63–64, 2005.

[28] T. Hastie, R. Tibshirani, and J. Friedman, "The Elements of Statistical Learning: Data Mining, Inference, and Prediction," *Springer*, pp. 398–400, 2009.

[29] R. Huber and S. Meyer-Kahlen, "UCI (=universal chess interface)," *Computer Chess Club*, 2000.

[30] IBM: DeepMind. (2018) The story of AlphaGo so far. [Online]. Available: https://deepmind.com/research/alphago

[31] ——. (2019) AlphaZero: Shedding new light on the grand games of Chess, Shogi and Go. [Online]. Available: https://deepmind.com/blog/alphazero-shedding-new-light-grand-games-chess-shogi-and-go/

[32] L. Kocsis and C. Szepesvari, "Bandit based Monte-Carlo Planning," *European Conference on Machine Learning*, pp. 282–293, 2006.

[33] T. L. Lai and H. Robbins, "Asymptotically Efficient Adaptive Allocation Rules," *Advances in Applied Mathematics*, vol. 6, pp. 4–22, 1985.

[34] M. Levene and J. Bar-Ilan, "Comparing Typical Opening Move Choices Made by Humans and Chess Engines," *Birkbeck College, University of London*, pp. 1–4,9–10, 2006.

[35] Lichess. (2019) Board editor. [Online]. Available: https://lichess.org/editor

[36] H. Mannen, "Learning to Play Chess Using Reinforcement Learning with Database Games," *Master's Thesis, Utrecht University*, 2003.

[37] S. Meyer-Kahlen. (2018) Shredder User Manual. [Online]. Available: http://download.shredderchess.com/mac/sc3/manual/UserManual.pdf

[38] T. Mitchell, "Machine Learning," *McGraw-Hill*, pp. 96–98, 1997.

[39] NumPy. (2019) About NumPy. [Online]. Available: https://www.numpy.org

[40] M. J. Osborne, "An Introduction to Game Theory," *Oxford University Press*, 2004.

[41] A. Ozdaglar, "Continuous and Discontinuous Games," *MIT 6.254: Game Theory with Engineering Applications*, 2010.

[42] Pandas PyData. (2018) Python Data Analysis Library. [Online]. Available: https://pandas.pydata.org

[43] ——. (2019) Pandas API Reference: DataFrame. [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

[44] B. Pandolfini, "Pandolfini's Chess Complete: The Most Comprehensive Guide to the Game," *History to Strategy*, pp. 15–21,41–44, 1992.

[45] K. Pearson, "Contributions to the Mathematical Theory of Evolution. II. Skew Variation in Homogeneous Material," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, p. 343–414, 1895.

[46] L. Presser and J. White, "Linkers and Loaders," *ACM Computing Surveys*, vol. 4, p. 149–151, 1972.

[47] R. Ramanujan, A. Sabharwal, and B. Selman, "On Adversarial Search Spaces and Sampling-Based Planning," *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, pp. 242–245, 2010.

[48] C. Ray. (2013) How stockfish works: An evaluation of the databases behind the top open-source chess engine. [Online]. Available: http://rin.io/chess-engine/

[49] Richard M. Stallman and Roland McGrath and Paul D. Smith. (2016) GNU Make: A Program for Directing Recompilation. [Online]. Available: https://www.gnu.org/software/make/manual/make.pdf

[50] S. Rogers and M. Girolami, "A First Course in Machine Learning," *CRC Press*, pp. 208–2014, 2011.

[51] T. Romstad, M. Costalba, and J. Kiiski. (2019) Stockfish Chess: About. [Online]. Available: https://stockfishchess.org/about/

[52] S. J. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach (3rd ed)," *Pearson*, 2010.

[53] F. C. Schadd, "Monte-Carlo Search Techniques in the Modern Board Game Thurn and Taxis," *M.S. thesis, Maastricht University, Netherlands*, 2009.

[54] SciPy.org. (2019) Statistical Functions: Scipy Stats Skew. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.skew.html

[55] C. F. Sironi and M. H. M. Winands, "On-Line Parameter Tuning for Monte-Carlo Tree Search in General Game Playing," *Computer Games*, pp. 75–95, 2018.

[56] Sudharsan Asaithambi. (2017) Why, How and When to Scale your Features. [Online]. Available: https://medium.com/greyatom/why-how-and-when-to-scale-your-features-4b30ab09db5e

[57] R. S. Sutton and A. Barto, "Reinforcement Learning: An Introduction," *MIT Press*, p. 13, 1998.

[58] F. Teytaud and O. Teytaud, "On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms," *Copenhagen, Denmark*, 2010.

[59] C. Watkins, "Learning from Delayed Rewards," *Ph.D. Thesis, Cambridge University*, 1989.

[60] WBEC Ridderkerk. (2004) Description of the universal chess interface (UCI). [Online]. Available: http://wbec-ridderkerk.nl/html/UCIProtocol.html

[61] Wikipedia. (2019) Bitboard. [Online]. Available: https://en.wikipedia.org/wiki/Bitboard

[62] Wikipedia. (2019) Comma-separated values. [Online]. Available: https://en.wikipedia.org/wiki/Comma-separated_values

[63] M. Winands, Y. Bjornsson, and J. T. Saito, “Monte-Carlo Tree Search Solver,” *Computers and Games. Lecture Notes in Computer Science, vol 5131.*, 2008.

[64] D. Zwillinger and S. Kokoska, “CRC Standard Probability and Statistics Tables and Formulae,” *Chapman & Hall: New York*, pp. 4–5, 18, 2000.