Monte Carlo Algorithms for Time-Constrained General Game Playing

Victor Scherer Putrich,[1] **Felipe Meneguzzi,** [2 1]

[1] Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
[2] University of Aberdeen
scherer.victor98@gmail.com, felipe.meneguzzi@abdn.ac.uk

Abstract

General Game Playing (GGP) is a complex field for Artificial Intelligence (AI) agents because it demands the ability to play varied games without prior knowledge. This paper introduces two algorithms to enhance move suggestions in time-limited GGP. Our first strategy is a modification of Sequential Halving Applied to Trees (SHOT), a non-exploiting algorithm. The second strategy is a hybrid version of Upper Confidence Tree (UCT) that combines Sequential Halving and $UCB_{\sqrt{}}$ to focus more on acquiring information at the root node. To test how agents perform, we use three different evaluation scenarios. First, we observe how resources are allocated among the selection policies. Next, we compare the performance of these strategies over five different board games with a set number of playouts, and in a competitive GGP environment where each game is played in one minute. These tests allow us to analyze the outcomes and implications of our proposed strategies.

1 Introduction

General Game Playing (GGP) is a research area focused on developing intelligent agents capable of playing a wide variety of games without prior knowledge (Genesereth, Love, and Pell 2005). GGP agents must cope with the rules of games and play them efficiently without human intervention. Developing general agents and the artificial intelligence (AI) techniques that support them is essential for creating real-world agents that can handle unpredictable and novel situations.

The Upper Confidence for Trees (UCT) (Kocsis, Szepesvári, and Willemson 2006) algorithm has been effectively utilized in GGP environments. UCT is based on building a search tree using Monte Carlo Tree Search (MCTS). MCTS employs Monte Carlo simulations to iteratively build a game tree, which progressively converge on the best action as it gathers more statistical information about the domain.

UCT guarantees converging to the best sequence of actions (Kocsis, Szepesvári, and Willemson 2006), while it also optimally minimizes the cumulative regret of not following this optimal path. However, this assurance of optimality is not without its costs. Depending on the complexity of decisions inherent in the given game scenario, UCT might require an impractically long time to produce high-quality recommendations.

A significant challenge in GGP is designing algorithms that can efficiently find solutions in a timely manner, particularly in competitive contexts, where the time required to find a solution is critical to the agent's performance. However, little discussion has been made about building GGP agents in environments with scarce time resources.

In this paper, we tackle the problem of General Game Playing with scarce time resources. Specifically, we focus on the following question: Is it UCT the best option for GGP environments with scarce time resources?

We are presenting two algorithms designed to outperform UCT under time constraints. We adapt the Sequential Halving Applied to Trees (SHOT) algorithm (Cazenave 2014) for a GGP scenario as our first approach. Our second approach we present the $\text{UCT}_{\sqrt{\text{SH}}}$ algorithm, a hybrid method based on Simple Regret plus Cumulative Regret (SR+CR) scheme (Tolpin and Shimony 2012) and the Hybrid MCTS (H-MCTS) (Pepels et al. 2014) algorithm. Those hybrid approaches aim to be more exploratory than UCT to avoid overspending time on the immediate best-rewarded movements while searching through game's action-space.

To evaluate the capabilities of our agent, we conducted three distinct experiments. The first experiment involves a simplified Multi-Armed Bandits (MAB) problem, designed to compare the way each polices spends its resources in scenarios characterized by high and low variance distribution of rewards. The second experiment evaluates the performance of the agents across five distinct games, employing a fixed number of playouts for each move. In the third and final experiment, we assessed the performance of the agents under a GGP scenario, involving a wide array of games under stringent time constraints. For this purpose, we utilized the Ludii GGP environment (Piette et al. 2020). We adopted the Kilothon competition scheme, which is one of the tracks of the GGP competition held on the Ludii platform [1].

The main contributions of our work are as follows: Firstly, through our MAB, we underscore the importance of more explorative policies for the context of game playing, whenever a promising option has already been found. Under the second experiment, we shown that $\text{UCT}_{\sqrt{\text{SH}}}$ method

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]https://github.com/Ludeme/LudiiAICompetition

achieved better performance under all tested games using different number of playouts, and discuss about the weak performance of SHOT under those games. Finally, Kilothon indicates UCT$_{\sqrt{SH}}$ outperforms UCT when operating under a GGP scenario with strict time constraints.

2 Game Playing

This section explores the interaction between Artificial Intelligence and games, specifically focusing on GGP. Game trees are discussed, serving as a critical structure for reasoning in game domains, along with the minimax algorithm, an essential strategic decision-making process in adversarial games. The focus then shifts to GGP, which aims to cultivate AI agents adept at playing a range of games without specific prior knowledge of the game. The discussion concludes with an exploration of the Ludii system, a platform designed for game research and development.

2.1 Games and Game Theory

AI employs games as a practical tool to train and evaluate algorithms. A game, in this context, refers to an activity where two or more agents, whether human or AI, interact in a structured and competitive manner to achieve their own victory. These games can come in various forms, including board games, card games, puzzles, video games, and so on.

Game Theory (Fudenberg and Tirole 1991) is a vital tool that enables AI to formalize conflict and cooperation between agents. It helps identify strategies, decision-making processes, as well as classify the domain that AI is expected to deal with.

In this work, we focus specifically on **adversarial** agents. In these scenarios, we can't predict the opponent's decisions, so each player must anticipate the potential opponent's reactions to their moves and assess the outcomes' benefits or drawbacks. We categorize these games as **zero-sum**, where every player's move results in an advantage that directly converts into a disadvantage for the opponent. We consider represent these gains and losses as utility values, where the player's gains equate to the opponent's losses.

Games can have **perfect** or **imperfect information**. An AI agent that plays a game with perfect information has knowledge of all the components used in the game, while in games with imperfect information, the agent's visibility within the environment is restricted. To give one example, most card games are games with imperfect information, where one player cannot see which cards the opponent has. Additionally, games can be **deterministic** or **stochastic**. Deterministic games ensure that whenever an action is taken into a specific state, the environment is modified always in the same way. On the other hand, stochastic games do not offer such a guarantee. To given one example, using dices makes some aspects of the game uncertain, where in many cases, the number displayed by the dice dictates how the action will modify the environment.

2.2 Game Trees and Game Tree Complexity

A game tree serves as a representation on the possible outcomes of a game. A complete game tree includes all potential game progressions, from the initial position to every possible conclusion. Each path from the root to a leaf is a full match that could happen within the game. The tree grows through a searching algorithm in its state-space. In certain cases, a complete game tree might be infinite if the game is unbounded, or if the game rules permit infinite repetitions of looped movements [Chapter 5.1](Russell and Norvig 2020).

Each node contains a game state, while each edge denotes a valid move that transitions from one state to its successor. The tree's leaves represent the final outcomes, such as a win, loss, or draw. The root node corresponds to the current configuration of the ongoing match; in a complete game tree, it represents the game's initial state.

The game tree complexity is the number of leaf nodes necessary to solve a game-tree, starting at the initial position of the game (Allis et al. 1994). For many domains, computing the entire game-tree is unfeasible, so we estimate it through an approximation of game depth and branching factor. The tree "depth", denotes the maximum number of turns or steps in a game, and the "branching factor", represents the average number of decisions available at each node in the game. We approximate the game-tree complexity with the branching factor raised to the power of the depth.

The evaluation of game tree complexity is important as it signifies the difficulty an AI algorithm will face in evaluating every possible outcome to decide on which move is the best. Higher complexity means more potential game states to examine, thus requiring more computational resources. Hence, AI algorithms often rely on techniques such as heuristic evaluation functions, pruning, and approximations to build and traverse partial game trees, assessing the utility of potential moves to make intelligent decisions.

2.3 Minimax

A fundamental concept in adversarial games is the Minimax algorithm (Van Der Werf 2004). Minimax operates by recursively evaluating potential game outcomes searching for the optimal move, which in turn maximizes the *minimax value*. This minimax value reflects the utility gained from following the best path from a given node to a leaf node. The underlying premise of the algorithm is that every participant aims to maximize their own utility values. In the context of zero-sum games, a player's gain is the opponent's loss; hence, the opponent maximizes its value at the expense of the player. The minimax tree construction alternates between player and opponent moves at each level, maximizing values at the player's level and minimizing them at the opponent's.

The minimax algorithm provides an effective solution to identify optimal strategies in two-player, zero-sum games. However, in games that don't match this model, such stochatis games or games with imperfect information, the minimax algorithm is not appropriate. For these cases, Minimax needs to be modified. For instance, the expectminimax algorithm models uncertain scenarios by including chance nodes, computing the expected value of a position instead of its minimax value [Chapter 5.5] (Russell and Norvig 2020).

Minimax algorithm used in its pure form is very computational cost for several game domains. For many occasions, it needs to use enhancements to be a viable strategy (Van Der Werf 2004). In such cases, one may need to use pruning methods, like alpha-beta pruning (Knuth and Moore 1975), to lessen the search space and speed up the algorithm. Usually, exploring the complete game tree isn't practical, so the algorithm should limit the searching to a certain depth. In case of reaching a non-terminal node limited by depth, the agent relies on handcrafted heuristic function to evaluate and compute a score for the node. However, this approach loses optimality because incomplete trees don't compute the actual game outcome in their leaves. Instead, they estimate the game state's quality using an evaluation function.

2.4 General Game Playing

General Game Playing (GGP) aims to develop intelligent computer agents adaptable to a wide variety of games, thereby eliminating the need for game-specific knowledge and minimizing human intervention in tackling different domain issues. The singular requirement for a GGP agent is the presence of a well-defined game domain. However, the agent's understanding of the game's rules, objectives, and strategies is constrained to the information available in the game description itself.

The GGP concept is highly related to GGP competitions, in order to promote the area and encourage researches to improve their techniques. The official GGP competition, proposed by Stanford Logic Group in 2005 (Genesereth, Love, and Pell 2005), focuses on finite combinatorial games with perfect information, including board games, puzzles, and any game that can be represented in a formal logic-based language, called Game Description Language (GDL). Swiechowski et. al underscores the crucial role of International GGP Competitions in incentivize GGP research (Świechowski et al. 2015).

MCTS has been the main strategy for building GGP agents, catalyzing significant progress in the field. Recently, Minimax surged as a viable competitor, combining it with Deep Reinforcement Learning (DRL) and zero-learning. The winner of the last edition of the Ludii GGP competition (2022), used DRL with a variation of Minimax called Unbounded Best-First Minimax (UBFM) (Cohen-Solal 2020). Other interesting approaches are under research using searching algorithms and reinforcement learning (Soemers et al. 2019, 2021a; Scheiermann and Konen 2022). However, our work will focus solely on improving the searching method based on Monte Carlo simulation.

2.5 Ludii

Ludii is a system designed to promote research in the field of gaming, specifically in the area of general game research (Piette et al. 2020). Ludii is an enviornment for playing a diverse range of games, test AI search algorithms, and design new games. Some of the key features of Ludii can be found at Ludii's documentation[2].

One feature of Ludii is to create and replicate historical games using its GDL. Ludii's GDL is simple and offers a wide variety of options for building games. Games are built through a combination of "ludemes", which is a term used to specify tokens that represent conceptual elements of a game. It is a data unit that allows game designers to describe how their game works in a meaningful way (Parlett 2016).

The concept of having a common nomenclature among games has many advantages, such as the ability to propagate elements from one game to another, making it possible to create variants of existing games and new ones, automatic generation of game rule sets, comparison of games, and demonstration of new strategies learned from AI algorithms in a more comprehensible way.

Some GGP research directions using Ludii are:

- Meta Agents: Defining hyper heuristics that choose the agent that better fits to the game types.
- Transfer Learning: Consists in transferring knowledge from one game to another where their domains share similar characteristics, such as heuristics, value functions or policies (Soemers et al. 2021b).
- Game Generation: New games can be generated through a combination of concepts seen in other games. Ludii has a vast database of games and ludemes making it a tool for creating new games (Browne 2008).
- Game Reconstruction: There are many historical games for which the rules are unknown or incomplete. A process similar to that of game generation is used to reconstruct rules of incomplete games (Browne 2020).
- Heuristic Prediction: Based on a game description, is it possible to learn which heuristics are important to use along with its respective weights (Stephenson et al. 2021).

By providing a friendly environment for GGP research, Ludii promotes innovation and collaboration among researchers, game designers, and AI enthusiasts.

3 Monte-Carlo Methods

Monte Carlo techniques employ random sampling to address problems that are otherwise intractable. The key idea behind Monte Carlo methods is to simulate a problem many times, each time using a different set of random inputs. The empirical average of the results obtained from these simulations provides an estimate of the true value. As the number of simulations increases, this estimate converges to the most likely outcome. In game-playing algorithms, Monte Carlo methods can be used to evaluate a game-tree node, computing the expected outcome of taking actions on it. This is achieved by randomly generating and assessing a sufficiently large set of game completions, often referred to as *playouts*.

The first algorithm for game playing based on Monte Carlo simulation was introduced by Abramson (Abramson 1990). Flat Monte Carlo algorithm samples uniformly the possible movements applicable to the current state of the game. The problem with the algorithm is its inability to model adversarial scenarios due to no computing a game-tree. Using Monte Carlo removes the need of defining functions for evaluating states, making them useful in situations where defining such heuristics are difficult or computational costly.

[2]https://ludii.games/downloads/LudiiUserGuide.pdf

3.1 Regret on Bandits Problem

The Multi-Armed Bandit (MAB) problem involves making decisions under uncertainty when the rewards and probabilities of success for each action are unknown. Imagine a casino slot machine with K distinct arms, each with its own reward distribution. The gambler's objective is to plan a strategy that maximizes their overall profit. The challenge lies in determining the number of times to pull each arm to maximize returns while learning rewards and probabilities distributions. The bandits problem presents a trade-off: the gambler must balance the pursuit of immediate profits by selecting the currently best-performing arm (exploitation), against the exploration of lesser-known arms to potentially uncover higher rewards with more trials (exploration).

In game-playing, the MAB problem can be translated into the searching challenges faced by agents when they should decide how Monte Carlo simulations gathers information about the possible movements. Just as in the MAB problem, game-playing agents using Monte Carlo to find an optimal balance between exploring new moves and exploiting promising moves.

Measuring performance in MAB problem is made through the concept of regret, which is defined as the difference in the reward obtained from the arm pulled and the optimal arm. We use two important measures of regret adapted from the definitions at Pepels (Pepels et al. 2014) and Bubeck (Bubeck, Munos, and Stoltz 2011). Specifically, we use cumulative and simple regret from Definitions 1 and 2.

Definition 1 *Cumulative regret is the accumulated regret over a set of arm pulls. Let $\mu^\star$ be the best expected reward, μ_j be the reward obtained from arm j, and $\mathbb{E}[T_j(n)]$ be the expected number of plays for arm j in the first n trials. Then, the cumulative regret R_n can be defined as:*

$$R_n = \sum_{j=1}^{k} \mathbb{E}[T_j(n)](\mu^\star - \mu_j) \tag{1}$$

An alternative experimental setup involves finding the optimal arm by allowing the gambler to discover the rewards and probabilities through a simulated version of the problem, where taking actions has no repercussions on the real environment. In order to evaluate situations where only the last arm pull is under consideration, we define Simple Regret.

Definition 2 *Simple regret is the expected difference between the best expected reward $\mu^\star$ and the reward of the arm pulled μ:*

$$r_n = \mu^\star - \mu \tag{2}$$

In the context of Multi-Armed Bandit problems, upper and lower bounds are used to provide theoretical guarantees on the performance of a strategy or algorithm, particularly with respect to regret. These bounds help quantify the efficiency of an algorithm in terms of worst-case and best-case scenarios:

- **Upper bounds:** An upper bound on cumulative or simple regret represents the maximum amount of regret which an algorithm is expected to experience under the worst-case scenario. A lower upper bound is preferred as it suggests that the algorithm's worst-case performance is better controlled, signifying that the algorithm, even under challenging conditions, can limit its deviation from optimal decision-making to a smaller extent.
- **Lower bounds:** A lower bound on the cumulative or simple regret represents the minimum amount of regret that any algorithm can achieve for a given problem. A higher lower bound suggests a greater level of inherent difficulty in the problem, as it signifies that even the most optimally performing algorithm will inevitably experience a certain degree of regret.

In the study by Bubeck et al. (Bubeck, Munos, and Stoltz 2011), the authors showed a trade-off between minimizing cumulative regret and simple regret. Specifically, they found that A smaller upper bound on $\mathbb{E}R_n$ leads to a higher lower bound on simple regret $\mathbb{E}r_n$, meaning that when an algorithm performs well in terms of cumulative regret (better upper bounds), it is likely to have a higher minimum simple regret (worse lower bounds). Conversely, a smaller upper bound on simple regret would lead to a higher lower bound on cumulative regret.

This trade-off indicates that no single policy can provide an optimal guarantee on both simple and cumulative regret at the same time. Depending on the context of the problem, one may prioritize either minimizing cumulative regret or minimizing simple regret.

3.2 Upper Confidence Bound

Upper Confidence Bound (UCB) (Auer, Cesa-Bianchi, and Fischer 2002) is a exploration policy in MAB problems and MCTS. A widely adopted variant, UCB$_1$, is favored for its simplicity and its ability to consistently deliver robust performance outcomes.

The key mechanism behind UCB$_1$ is its computation of potential rewards for each action. This is accomplished by establishing a confidence interval for the value of the arm, a range within which the value can be estimated to lie with high confidence (Russell and Norvig 2020, Chapter 17.3.3).

The UCB policy is notable for its optimal minimization of cumulative regret over the course of searching. Importantly, however, this minimization of cumulative regret is subject to an inherent limit. Specifically, according to Lai and Robbins (Lai, Robbins et al. 1985), the cumulative regret cannot grow slower than a logarithmic rate $\log(n)$, where n represents the number of trials. Furthermore, suboptimal moves in the UCB policy come with a theoretical guarantee of being selected at most $O(\log(n))$ times.

UCB$_1$ estimates the expected reward for state-action pairs as follows:

$$\mathrm{UCB}_1(s,a) = Q_{s,a} + 2c\sqrt{\frac{2\ln N_s}{n_{s,a}}}\ . \tag{3}$$

Here, $Q_{s,a}$ is the expected reward received each time action a is selected in state s. N_s is the number of visits the state s received, while $n_{s,a}$ is the number of times action a has been selected in state s. The square-root term measures uncertainty in the estimate of a's value, and parameter c regulates the exploration term's weight. Each problem domain should fine-tune the c parameter, but a commonly used value for c is $c = 1/\sqrt{2}$ (Kocsis, Szepesvári, and Willemson 2006).

Whenever action a is taken at state s, our estimate of the achievable reward, denoted as $Q_{s,a}$, tends to decrease. This is because the more we experience a specific action in a given state, the more confident we become about the estimated reward associated with that action. In other words, we progressively reduce the uncertainty about our expectation of $Q_{s,a}$.

On the other hand, when state s is visited but action a is not selected, our estimate of the potential reward from action a increases. The idea is that there might be higher rewards associated with lesser-explored actions. However, this increase in potential reward diminishes over time due to the natural logarithm's influence on the count of state visitations, N_s.

Finally, in the context of UCB_1, the confidence interval around each action's reward estimate promotes a balance between exploration and exploitation. The UCB1 algorithm typically leans towards exploring actions with high uncertainty, gradually shifting to exploitation as it gains confidence about the best actions.

UCB_1 offers a desirable property: the discovery process can be interrupted at any time, providing an estimate of each option's quality based on collected samples. This anytime property allows for more flexibility in managing computational resources.

3.3 Sequential Halving

Non-exploitative selection policies[3] have been advanced with the aim of rapidly decreasing simple regret while maintaining low regret bounds. Considering that UCB_1 has an optimal rate of cumulative regret convergence, and given the conflicting limits on simple regret bounds pointed by Bubeck et. al (Bubeck, Munos, and Stoltz 2011), policies with higher rates of exploration than UCB_1 tend to have better bounds on simple regret.

In many decision problems in games, there are often only a few promising moves to identify. As a result, when employing a uniform selection policy, the majority time is spent on sampling suboptimal arms. In order to minimize the frequency of selecting inferior arms, a more efficient policy is essential.

Sequential Halving is a flat, non exploiting, algorithm which provides better bounds on simple regret than UCB, and other non exploiting policies (Karnin, Koren, and Somekh 2013). The algorithm uniformly distributes a predetermined budget among all actions and progressively eliminates the bottom half in terms of lower performant set of options.

[3]Policies without an exploitation phase, they allocate resources uniformly among a steadily reducing set of options.

Algorithm 1: Pseucode for the Sequential Halving algorithm

```
1:  function SEQUENTIALHALVING($s$, $\mathcal{B}$)
2:      Input: state $s$, budget $\mathcal{B}$
3:      Output: Recommended action
4:      $v_{root} \leftarrow \langle s, \text{ACTIONS}(s) \rangle$
5:      $C \leftarrow ||\text{C}(v_{root})||$
6:      $k \leftarrow C$
7:      while $k > 1$ do
8:          $b \leftarrow \lfloor \frac{\mathcal{B}}{k \times \lceil \log_2 C \rceil} \rfloor$
9:          for $v' \in \text{HEAD}(\text{C}(v_{root}, k))$ do
10:             $Q(v') \leftarrow Q(v') + \text{PLAYOUT}(v', b)$
11:             $N(v') \leftarrow N(v') + b$
12:         $\text{SORT}(\text{C}(v_{root}), k)$
13:         $k = \lceil k/2 \rceil$
14:     return $\text{C}(v_{root})[0]$
```

Algorithm 1 illustrates an implementation of Sequential Halving, in which we use a tree-like structure for consistency with subsequent algorithms. A node v is a 5-tuple composed by: expected reward Q, number of visits N, set of applicable actions (given by calling ACTIONS), and a list children of v (when calling $\text{C}(v)$). We employ an index, represented as k, to restrict the range of accessible children starting at position zero up to k. This enables the pruning of nodes, efficiently limiting their consideration in subsequent computations, without necessitating their complete removal from memory.

The algorithm iterates over child nodes using $\text{HEAD}(\text{C}(v_{root}), k)$ to allocate a portion of the budget $\mathcal{B}$ among the available children. It sorts the nodes based on the data collected from playouts and then halves the value of k for the upcoming round. The Sequential Halving formula, mentioned in Line 8, determines the number of times the set of children can be halved until only two remain, represented as $\log_2(\text{C}(v))$, and divides the budget $\mathcal{B}$ by this number. To allocate the budget evenly across the remaining children in the current round, the algorithm divides $\mathcal{B}$ by k.

3.4 Monte-Carlo Tree Search

Monte Carlo simulations have great potential in various game scenarios, however, flat solutions can pose challenges in making accurate recommendations. In this context, building a tree is more effective for designing AI in turn-based games, as it better models the true dynamics between player and opponents responses.

Unlike the exhaustive search requirements of Minimax, Monte Carlo Tree Search (MCTS) employs Monte Carlo simulations to iteratively build a game tree. MCTS is designed to progressively converge to the best action as it gathers more statistical information about the domain. This method form the basis of effective approaches for games with complex strategies, such as Go, Poker, Chess, Hex, Othello, Settlers of Catan, and general game-playing environments (Świechowski et al. 2022). MCTS is based on two principles: (1) with sufficient time, the sampled average reward from random simulations converges to the true state value, and (2) previous samples can guide future searches.

Algorithm 2: Pseudocode for Monte-Carlo Tree Search algorithm

```
1:  function MCTS($s$, $\mathcal{R}$)
2:      Input: State $s$, Resource $\mathcal{R}$
3:      Output: Recommended action
4:      start $r$
5:      $v_{root} \leftarrow \langle s, \text{ACTIONS}(s) \rangle$
6:      while $r$ not achieved $\mathcal{R}$ do
7:          $v_k \leftarrow \text{SELECTION}(v_{root}, \pi)$
8:          $v_{k+1} \leftarrow \text{EXPANSION}(v_k)$
9:          $rw \leftarrow \text{SIMULATION}(v_{k+1}, \pi_\Delta)$
10:         $\text{BACKPROPAGATION}(v_{k+1}, rw)$
11:         increase $r$
12:     return $\text{RECOMMEND}(\text{C}(v_{root}))$
```

Algorithm 2 outlines the MCTS process, starting at the root node, denoted as v_{root}. A node v consists of a state s, the set of untried actions in s, the parent node (null for the root node), the children from v, and Q and N values for expected reward and visit count, respectively.

The searching process involves following four steps:

- *Selection*: Starting from the root of the tree, the selection phase navigates through the tree using a *tree policy* (π) that directs the search towards promising nodes. The SELECTION function explores the tree until it identifies a node with untested actions.
- *Expansion*: The EXPANSION function selects an untested action at random, removes it from the set of untested actions, and creates a new child node, appending it to the existing list of children. This child node is initialized with new state values, a set of applicable actions, and the parent node defined as $v_{k+1}\langle s_{k+1}, \text{ACTIONS}(s_{k+1}), v_k \rangle$.
- *Simulation*: The algorithm runs a playout to reach a result represented by reward r, following the *default policy* (π_Δ). In a classic Monte Carlo simulation, the game concludes by taking random actions. Various improvements are employed during this phase to enhance simulations or estimate the return without the need to complete the simulation (Świechowski et al. 2022).
- *Backpropagation*: The algorithm updates each traversed node from v_{k+1} up to the root. It updates the expected reward Q by adding the weighted reward rw, and it increases the visit count N by 1.

The searching continues until it uses up a specified resource limit $\mathcal{R}$; which may be defined as number of iterations, memory usage or searching time. After completing the searching phase, the program select an action to be taken in the game. The function $\text{RECOMMEND}(\text{C}(v_{root}))$ selects the best root's child as a movement to be played in the real environment according to one specific criteria:

1. Max Child: chooses the child with the highest Q value.
2. Robust Child: chooses the child with the highest N value, which indicates the number of times it has been tested.
3. Max-Robust Child: combines the previous two criteria, choosing the child with both the highest Q and N values. If no child can be selected, the search continues until a suitable child is found. This ensures that the algorithm always selects a valid action to play in the game.

When used as a selection policy in MCTS, UCB$_1$ turns into a recursive version of the MAB problem, where each node minimizes its internal cumulative regret for selecting its children. When used in MCTS the algorithm is called Upper Confidence Bounds applied for Trees (UCT) algorithm (Kocsis, Szepesvári, and Willemson 2006). MCTS and UCT exhibit an anytime property, allowing them to recommend useful actions even when the search execution is interrupted.

4 Alternatives to UCT

Simple regret minimization is strictly related to choosing a child node from the root at the recommendation phase, and the cumulative regret is related to the search process through the tree. UCB$_1$ has optimal bounds on cumulative regret, but it is penalized in terms of simple regret. At the root node, sampling in MCTS/UCT typically focuses on finding the best move with high confidence. Once UCB$_1$ identifies such a move, it continues to spend time on it, possibly with low information gain (Tolpin and Shimony 2012).

In time-sensitive situations, not considering other options and continuing with the current best choice may be a potential flaw that could be improved. By exploring more, the agent may quickly switch towards other promising alternatives, potentially reaching higher reward regions of the search tree. The subsequent sections explore three strategies developed in response to this observed characteristic of UCT.

4.1 Sequential Halving Applied On Trees

Sequential Halving Applied to Trees (SHOT) is an algorithm that combines the action elimination strategy of Sequential Halving with a tree-based search approach (Cazenave 2014). Instead of backing up individual simulations, SHOT backpropagates groups of simulations. This approach allows the algorithm to grow its tree almost twice as fast in optimized environments without re-searching the tree after each simulation (Pepels 2014). Another key advantage of SHOT over traditional UCT algorithms is its reduced memory requirements; Unlike UCT, SHOT does not store leaf nodes, which significantly reduces the amount of memory required for the algorithm to operate. SHOT's pseudocode is provided in *Algorithm* 1, only the most important steps are considered for understanding SHOT.

The algorithm starts from the root and expands its nodes using the EXPAND function. The expansion apply one simulation for each unvisited child, and discount new node discoveries from budget $\mathcal{B}$. SHOT grows in depth until reaches a terminal state, or there only one budget left. Visiting a node differs from using a budget, because reaching a terminal node does not require simulations, then it is not considered as a budget usage. When nodes have one budget left, it is recorded a visit and budget spending.

A notable distinction between Sequential Halving and SHOT is the recursive evaluation. While the Sequential

Algorithm 3: Pseudocode of SHOT algorithm

```
1: function SHOT($v$, $\mathcal{B}$)
2:     Input: node $v$, budget $\mathcal{B}$
3:     Output: Searching a game-tree using SHOT
4:     if $v$ is terminal then
5:         UPDATE values from $v$
6:         return
7:     if $\mathcal{B} = 1$ then
8:         SIMULATE a random game starting from $v$
9:         UPDATE values from $v$ using SIMULATE
10:        return
11:    EXPAND unseen children from $v$
12:    repeat
13:        for child $v'$ in $v$ do
14:            define $b$ using SHOT budget formula
15:            SHOT$(v', b)$
16:            UPDATE values from $v$ with $v'$ values
17:        SORT children from $v$
18:        ELIMINATE half children from $v$
19:    until one child left
20:    return RECOMMEND$(\mathrm{C}(v_{root}))$
```

Halving algorithm runs once, SHOT executes multiple *cycles* of Sequential Halving at the same node. When an internal node completes a cycle, the values obtained are back-propagated to its parent. In case the node is revisited, the previously collected information will be considered in the follow cycle. After each halve of Sequential Halving, the algorithm SORT and ELIMINATE half the remain children. Nodes not eliminated receive a larger portion of $\mathcal{B}$ in the next iteration. The search continues until the root node has only two children remaining. At this point, the root node selects the child with the highest expected reward.

The budget allocation in SHOT diverges from the original Sequential Halving by implementing an incremental strategy that distributes uniformly its budget to nodes considering previous cycles. All the unspent budget, coming from achieving terminal nodes and from non integer divisions, are carried over next halves and cycles, so it can be spend in the follow iterations. The management required for ensure an equal distribution wont be covered, but is important to keep in mind that SHOT is quite more complex than Sequential Halving and MCTS/UCT in its distribution because of those conditions.

Two concerns regarding SHOT were previously raised by (Pepels 2014):

1. **SHOT cannot be interrupted:** A limitation of the SHOT method is its inability to be halted prematurely, as it necessitates prior knowledge of the available budget. Consequently, the pure exploration policy can only offer a satisfactory low simple regret on recommendations after all simulations are concluded. It is impossible to stop the process or request a reasonable recommendation before reaching the limit.
2. **SHOT computes more like an average expected value of a node, rather than its minimax value:** UCT consistently selects the best node and has a formal guarantee that suboptimal nodes are selected at most $O(\ln n)$ times, ensuring that node values converge to the best-reply over time. However, with Sequential Halving formula, the guarantee relies on the available budget and branching factor, both are considered for Sequential Halving and SHOT distribution. This method samples the two best nodes equally, regardless of their reward difference. In many games, only a few good moves exist for a position, and evaluating a node requires determining the value of its move's best-reply. Sequential Halving's use throughout the tree causes values to back-propagate differently than in UCT, resulting in internal node values being more like overall averages of their children.

In the experiments conducted by Cazenave (Cazenave 2014), SHOT was compared to UCT in NoGo, achieveving win-rates between 75% and 100% on both 9x9 and 19x19 boards. The experiments by Pepels (Pepels 2014) involved SHOT competing in several matches against UCT for various games, including Amazons, AtariGO, Ataxx, Breakthrough, NoGo, and Pentalath, under a fixed budget allocation of playouts. SHOT performs best in games with the highest branching factors, such as Amazons, AtariGo, and, to a lesser extent, NoGo. The NoGo results from Cazenave's differ from those presented in Pepel's work due to the difference in experimental setups, with the former employing a time constraint for both algorithms and the latter using a fixed allocation of playouts.

These results reinforce the evidence that SHOT is best applied in games with high branching factors. In games with narrow winning-lines, such as Breakthrough and Pentalath, SHOT's performance declines significantly against UCT. However, given SHOT's speed improvement over UCT, it is possible that the technique performs better in a time-based experiment. This suggests that, the choice between SHOT and UCT may depend on the desired balance between speed and performance.

4.2 UCB$_{\surd}$ and SR+CR

Bubeck et. al (Bubeck, Munos, and Stoltz 2011) shows that UCB$_1$ exhibits a slow decrease in terms of simple regret, with the best-case scenario being a polynomial rate decrease. This can be problematic in game evaluation, as simple regret is closely related to move recommendations. Karning et. al (Karnin, Koren, and Somekh 2013) suggest that the so-called non-exploiting policies reduce simple regret more rapidly given enough time.

UCB$_1$ allocates new samples based on sample means and often chooses the current top-performing option, leading to a slow reduction in simple regret and an infrequent exploration of other potentially superior options. Tolpin and Shimony. (Tolpin and Shimony 2012) modify UCB$_1$'s policy into UCB$_{\surd}$. This policy adjusts the UCB$_1$ formula using a quicker-growing sublinear function, leading to a faster increase in the upper bound on the reward allocated to nodes. The new policy changes the $\ln N_s$ term in UCB$_1$ equation to $\sqrt{N_s}$, aiming to narrowing the gap between selections of non-optimal nodes.

Tolpin and Shimony point out that nodes closer to the root and those deeper in the tree have different levels of importance. The former is more crucial for move recommendations. As a result, the search strategy near the root should prioritize reducing simple regret more quickly, while nodes deeper in the tree should perceive to match the value of taking the optimal path, aligning more with prioritizing cumulative regret minimization.

The SR+CR (Simple Regret plus Cumulative Regret) scheme proposed by Tolpin and Shimony integrates two different policies to strike a balance between minimizing simple regret and cumulative regret. They introduced two specific algorithms, both of which combine the UCT policy with more exploratory strategies.

The first one, $\text{UCB}_{\surd} + \text{UCT}$, operates by applying the $\text{UCB}_{\surd}$ policy at the root node and the UCT policy to all child nodes. This is indicative of an approach that emphasizes different levels of exploration at different stages of the search. Their second algorithm, $\frac{1}{2}$-greedy $+$ UCT, introduces an even more exploratory policy. The $\frac{1}{2}$-greedy policy behaves such that it randomly selects a move 50% of the time, without considering the immediate reward of the action. These approaches reflect innovative combinations of policies, each aiming to optimize the trade-off between simple and cumulative regret minimization in different ways.

4.3 Hybrid Monte Carlo Tree Search

Hybrid Monte Carlo Tree Search (H-MCTS) is an algorithm that combines pure exploration strategies with MCTS to improve the performance of Monte Carlo agents (Pepels et al. 2014). This approach aims to minimize both simple and cumulative regret in the search tree, much like the SR+CR scheme discussed in Section 4.2. It does so by incorporating the Sequential Halving Applied on Trees (SHOT) algorithm (Cazenave 2014) with MCTS in a hybrid algorithm. In contrast to the SR+CR scheme, H-MCTS applies the pure exploration policy not only at the root node but also deep down the tree according to a specific condition. This allows the algorithm to minimizing simple regret internally for identifying quicker the best replies to parent moves throughout the rest of the tree.

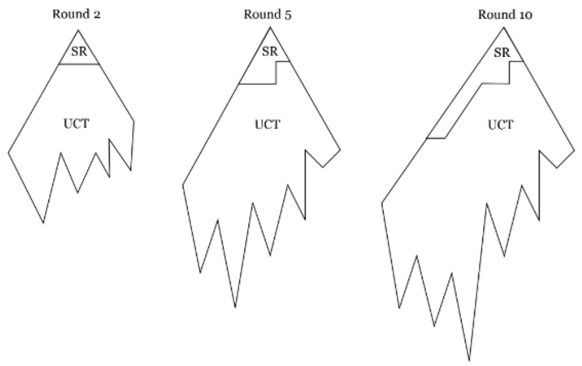

Figure 1: Progression of Sequential Halving over UCT in H-MCTS.

Figure 1 depicts how the progression of Simple Regret (SR) area under UCT. The search tree initially starts as a SHOT tree at the root, with UCT trees rooted at its leaves. The proposed method switches from Sequential Halving cycle to UCT when the computational budget spent in the node achieves a certain threshold. Since the computational budget per node is initially small, the simple regret tree remains shallow. As Sequential Halving eliminates nodes from selection, the budget spent increases, causing the SHOT tree to grow deeper.

Like Sequential Halving and SHOT, H-MCTS has the drawback of being unable to return a recommendation at any given time, as it must know its exact computational budget in advance. However, H-MCTS takes advantage of the speed gains provided by SHOT, caused by multiple playouts to be executed and back-propagated simultaneously in UCT. H-MCTS outperforms UCT for various exploration coefficients (Pepels et al. 2014) and is highly effective in games with large branching factors, as it prunes low-promising nodes and directs the search towards the most promising areas. However, in games requiring tactical strategies with narrow branching, exploiting strategies might be more suitable.

5 Seeking for Overcome UCT Under Time-Restricted Scenario

In this section, we focus on implementing a refined SHOT algorithm and adapting it for time-restricted scenarios. We also propose a variation of the SR+CR algorithm to enhance recommendations. Finally, we tackle the challenge of effective time management for these algorithms in unknown GGP scenarios. All the enhancements and modifications we discuss are available in our GitHub repository[4].

5.1 SHOT Modifications

SHOT algorithm offers an effective alternative to UCT for handling typical combinatorial games. However, to fit it into previously unknown domains, we introduce modifications aimed at simplifying the budget allocation policy.

In our version of SHOT, we opt for a straightforward budget allocation based on the budget distribution of Sequential Halving. This approach leads to two notable situations: firstly, nodes with higher rewards receive slightly more visits, leading to an uneven budget distribution. Secondly, unspent budgets from earlier cycles aren't reallocated to subsequent ones.

Algorithm 4 showcases our implementation of SHOT. SHOT requires an input specifying the number of playouts it should employ in the search. This requirement can pose a challenge for a GGP agent since the agent won't know the number of playouts to perform. The WARMUP function expands the first n nodes using depth-first search. After discovering each node, it executes a simulation and stores the time taken for completion. The thinking time $\mathcal{T}$ granted to the agent make a move is divided by the average time required for completing the n simulations at the warmup phase.

[4]For implementation details, please visit our GitHub repository: https://github.com/schererl/GraduateThesis

Algorithm 4: Pseudocode for SHOT agent

```
1: function SHOTRECOMMENDATION(s, $\mathcal{T}$)
2:     Input: State $s$, Thinking Time $\mathcal{T}$
3:     Output: Recommended action
4:     $v_{root} \leftarrow \langle s, \text{ACTIONS}(s) \rangle$
5:     $\tau_{warmup} \leftarrow \text{WARMUP}(v_{root}, n)$
6:     $\mathcal{B} \leftarrow \mathcal{T}/\tau_{warmup}$
7:     $\text{MSHOT}(v_{root}, \mathcal{B})$
8:     $\text{CLEARCYCLE}(v_{root})$
9:     $b \leftarrow \mathcal{B} - \text{N}(root)$
10:    distribute b among the top $s$ children
11:    return $\text{RECOMMEND}(\text{C}(v_{root}))$
12: function MSHOT($v$, $\mathcal{B}$)
13:    Input: Node $v$, Budget $\mathcal{B}$
14:    Output: Game-tree
15:    if $\text{TERMINAL}(v) \lor \mathcal{B} = 1$ then
16:        $\text{UPDATE}(v, \text{SIMULATE}(v) \times \mathcal{B}, \mathcal{B})$
17:        return
18:    if $||\text{UNVISITEDACTIONS}(v)|| > 0$ then
19:        $\text{EXPAND}(v, \mathcal{B})$
20:        if spend all $\mathcal{B}$ then
21:            return
22:    $k \leftarrow ||\text{C}(v)||$
23:    repeat
24:        $b \leftarrow \frac{\mathcal{B}}{\lfloor k \times \lceil \log_2 ||\text{C}(v)|| \rceil \rfloor}$
25:        $b' \leftarrow \text{MAX}(1, \lfloor b/k \rfloor)$
26:        for $v' \in \text{HEAD}(\text{C}(v), \text{MIN}(b', k))$ do
27:            $\text{MSHOT}(v', b')$
28:            $\text{UPDATE}(v, \text{QCYCLE}(v'), \text{NCYCLE}(v'))$
29:            $\text{CLEARCYCLE}(v')$
30:        $\text{SORT}(\text{C}(v), l)$
31:        $k \leftarrow \lceil k/2 \rceil$
32:    until $k > 1$
```

The original algorithm employs transposition tables (specifically, hash tables) for node storage, and utilizes variables with differing scopes using local variables passed through reference as a way of backpropagate values. Our approach implements a tree structure that store all variables, thereby simplifying the management of necessary information. We have implemented several enhancements to the original SHOT algorithm:

1. The original algorithm relies on a single function that returns the best move after each cycle completion. However, this leads to multiple unnecessary returns and makes the recommendation intertwined with searching. To address this, we split the recommendation (SHOTRECOMMENDATION) from the search process (MSHOT).
2. We consider only the number of visits N, rather than distinguishing with budget spending. This approach simplifies the variables required in the algorithm and combines the two terminal cases (when a node is terminal and when it has only one budget left). In case of reaching a terminal node, the PLAYOUT returns the value of the current node without making any additional moves.
3. To keep track of the information obtained in the current cycle, we use temporary variables, called "cycle" variables. We store into cycle variables the number of wins and visits got from the current Sequential Halving execution. After completing Sequential Halving, CLEARCYCLE function adds the cycle variables to Q and N, and reinitialize cycle variables to zero. Before closing a cycle, the parent node call UPDATE to add the values obtained from its child to its own cycle variables.
4. In case of a terminal node being reached, we consider $\mathcal{B}$ as the number of times the terminal node is visited and compute the accumulated reward according to it (line 16). We found that explicitly addressing this avoids potential inconsistencies on node's evaluation [5].
5. We allocate a fixed budget to each node for each cycle, divided equally among its children (line 24).
6. The budget given to each child b have a minimum value of 1 (line 25). In case the budget is lower than the number of children, distribute its budget to b-ith first children calling HEAD (line 26).

Our method of budget allocation deviates from the original policy, primarily because we opt not to carry over unspent budget. Instead of transferring unused budget into subsequent iterations, we use at *SHOT Recommendation* function. Upon concluding the primary search, we proportionally distribute the remaining budget to an arbitrary subset of children from root.

Though this approach may yield outcomes that differ from the original version, it should display reducing discrepancies when given a sufficient budget. This is because both algorithms move towards a uniform distribution, rather than adhering to a policy that relies on exploitation for selection.

While we understand the rationale behind the use of an incremental budget, we opt for a simpler, more reproducible strategy for the sake of simplicity. Therefore, we apply the modified SHOT as an initial validation of the algorithm's effectiveness as a GGP agent.

5.2 UCT$_{\sqrt{\text{SH}}}$

Although H-MCTS is promising at reducing simple regret, it requires a predefined budget for the SHOT portion, which is not possible to estimate for previous unknown environments. Furthermore, by neglecting the exploitation of nodes, the agent becomes prone to excessive resource allocation in unpromising regions. SR+CR uses UCT at CR area, so it doesn't benefit from performance gains using pure exploration policies across the entire tree, unlike SHOT.

We develop a different SR+CR method, using Sequential Halving and UCB$_{\sqrt{\cdot}}$, as shown in Algorithm 5. We take

[5]Consider a scenario involving a parent node *N* which distributes an equal budget of 100 to node *A* and node *B*. *A* is terminal and results in a direct win, returning a value of 1. Meanwhile, *B* is non-terminal sibling, and wins half of its 100 simulations. In this case, *N* would receive significantly more values from *B* than *A*, even though *A* had a certian winning. This disparity occurs because the number of backpropagated values from *B* is greater. Consequently, *B* would have a more substantial impact on the decision of node *N*.

inspiration on SR+CR scheme (Section 4.2) and H-MCTS (Section 4.3), which aim to enhance recommendations based on simple regret minimization near the root.

Algorithm 5: Pseudocode of UCT$_{\sqrt{\text{SH}}}$ algorithm

```
1: function UCT_{√SH}($s$, $\mathcal{R}$)
2:     Input: State $s$, Resource $\mathcal{R}$
3:     Output: Recommended action
4:     start $r$
5:     $v_{root} \leftarrow \langle s, \text{ACTIONS}(s)\rangle$
6:     $C \leftarrow |\text{C}(v_{root})|$
7:     $h \leftarrow 1$; $k \leftarrow n$
8:     while $r \leq \mathcal{R}$ do
9:         if $k > k_{min}$ and $r > (\mathcal{R}\frac{h}{\log_2 C})$ then
10:            $\text{SORT}(v_{root}, k)$
11:            $h \leftarrow h + 1$
12:            $k \leftarrow \text{MAX}(k_{min}, k/2)$
13:        $v_s \leftarrow \arg\max_{v \in \text{HEAD}(\text{C}(v_{root}),k)} \pi_{UCB_{\sqrt{}}}(v)$
14:        $v_k \leftarrow \text{SELECTION}(v_s, \pi_{UCB_1})$
15:        $v_{k+1} \leftarrow \text{EXPANSION}(v_k)$
16:        $rw \leftarrow \text{PLAYOUT}(v_{k+1}, \pi_{\Delta})$
17:        $\text{BACKPROPAGATION}(v_{k+1}, rw)$
18:        update $r$
19:    $\text{RECOMMEND}(\text{C}(v_{root}))$
```

UCT$_{\sqrt{\text{SH}}}$ prioritize simple regret minimization at root node by combining UCB$_{\sqrt{}}$ with Sequential Halving eliminations, and the cumulative regret component uses UCT. In UCT$_{\sqrt{\text{SH}}}$, the aim of Sequential Halving is not to converge to the single best move, but rather to limit the number of children to search, which allows UCB$_{\sqrt{}}$ to explore the most promising areas.

We establish a lower boundary on the number of children, k_{min}, for elimination to take place. When an elimination happens, the algorithm organizes the root's child nodes in descending order based on their expected reward. The halve counter, h, increases and k halves. During the root's child selection process, we use k to limit the selection to the first k-th children, as shown in Line 13.

We employ an iterative methodology to ascertain when to halve the number of children. This involves dividing h by the maximum potential number of halving operations, $\log_2 C$, which gives us a ratio representing the fraction of halving stages already completed. Multiplying this ratio with the total resource $\mathcal{R}$, we can estimate the resource allocation required for the next halving operation. When the actual resource consumption, r, surpasses this expected allocation, we increment h to reach the subsequent halving stage. This method ensures the resource $\mathcal{R}$ is consistently allocated across all $\log_2 C$ halving stages, proportionally to the ratio of halving stages completed.

A key distinction from traditional MCTS lies in the separate treatment of root selection. The root selection, depicted at line 13, chooses the child that maximizes $\pi_{UCB_{\sqrt{}}}$, iterating over the first k-th children.

5.3 Clock Bonus Time

GGP agents face the challenge of playing games without prior knowledge. Certain GGP situations provide agents with a fixed time budget to play an entire game, necessitating a decision on how much time to allocate to each move. In contests like the Kilothon, if an agent exhausts its total time budget, subsequent moves are randomly selected as a form of penalty.

We propose a methodology for estimating the time to allocate for each move in a GGP environment. Our model runs a specific number of simulations during the search process to gather data about the game. A minimum "thinking" time is designated, and for games where the agent can afford more time, we grant a "thinking time" bonus. The formula for our Clock Bonus Time (cbt) is:

$$\textit{cbt} = \max(\tau_{min}, \min(\tau_{max}, G/\overline{m})) - \tau_{min} \ . \qquad (4)$$

In Eq. 4, G represents the total game time, while τ_{min} and τ_{max} stand for the minimum and maximum permitted thinking times per move, respectively. The bonus time is calculated as G divided by the estimated number of moves the player have left, $\overline{m}$, which we determine using Monte Carlo simulations. The max and min functions guarantee that the agent commits to at least the minimum thinking time and prevents it from overestimating its available time. The final time calculation subtracts τ_{min}, as it represents a bonus addition to the minimum thinking time.

To incorporate *cbt* into MCTS, one approach involves calling *cbt* once half of τ_{min} has elapsed, or when $r \geq \mathcal{R}/2$.

6 Prize Box Selection Experiment

The Prize Box Selection Experiment [6] is a simplified MAB where there are K boxes containing a deterministic amount of money. The money for each box is pre-selected from a Gaussian distribution $N(\mu, \sigma)$. We test different policies for a given number of trials and boxes, recording how often the policy selects each box during the experiment.

6.1 30-Prize Box Experiment

The 30-box experiment evaluates UCB$_1$, Sequential Halving, UCB$_{\sqrt{}}$, and UCT$_{\sqrt{\text{SH}}}$ selection policies with 10000 trials, under low and high variance scenarios. In the low variance case, $\mu = 0.3$ and $\sigma = 0.05$, limited to [-0.5, 0.5]. For the high variance case, $\mu = 0.3$ and $\sigma = 0.5$, limited to [-1,1].

Figure 2 depicts the low variance scenario, showing only the 20 boxes with the highest rewards (i.e., boxes 1-10 have the lowest reward and are omitted).

In this low variance situation, UCB$_{\sqrt{}}$ exhibits the greatest spread of trials among all boxes, followed by UCB$_1$. In such an environment, the use of elimination strategies drives the policies adopting them towards a smaller subset of boxes, yielding higher rewards.

Figure 3 depicts the high variance test. In this context, both UCB$_1$ and UCT$_{\sqrt{\text{SH}}}$ show a marked preference for the

[6]https://tinyurl.com/selectionboxexperiment

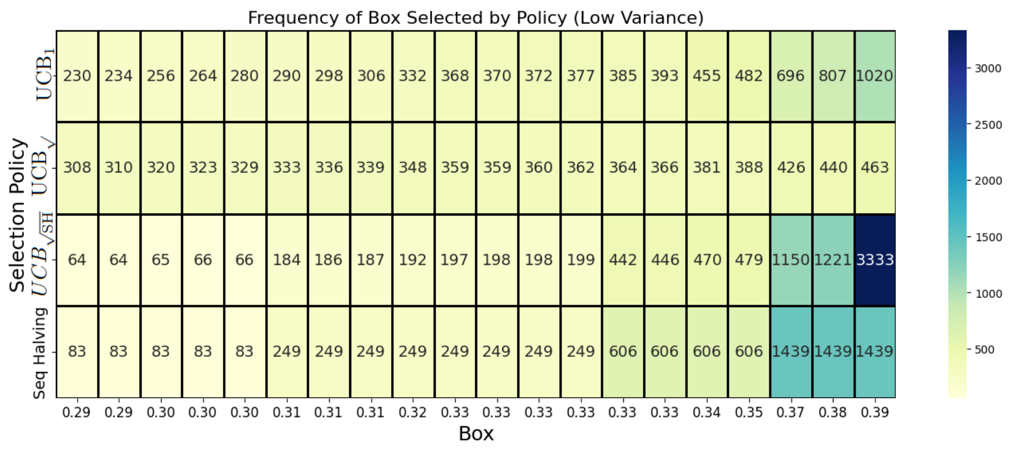

Figure 2: Box selection frequency under low variance reward distribution.

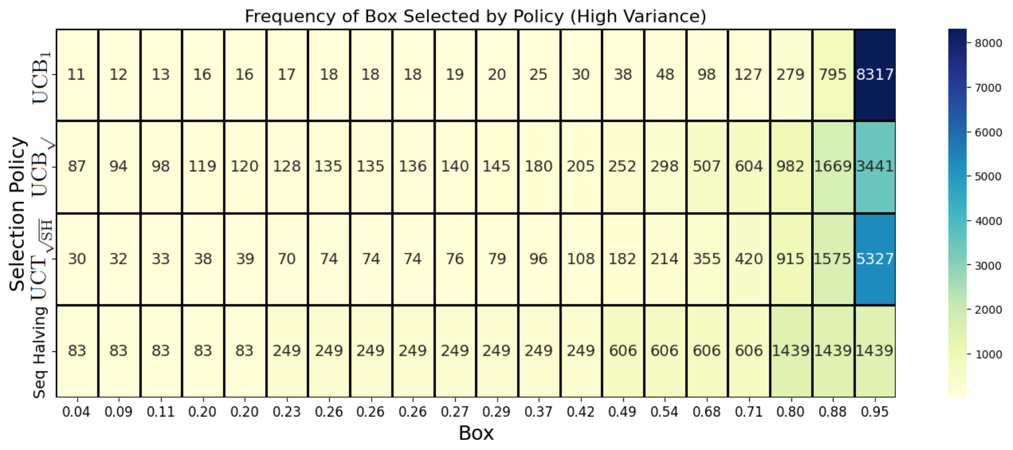

Figure 3: Box selection frequency under high variance reward distribution.

arm with the highest reward. On the other hand, $\mathrm{UCB}_{\sqrt{}}$ and Sequential Halving demonstrate more similar frequencies of selection, indicating their preference for exploration over the more exploitative UCB_1 policy. Notably, Sequential Halving, due to its non-exploitative nature, preserves a constant budget allocation regardless of reward distribution.

Despite $\mathrm{UCT}_{\sqrt{\mathrm{SH}}}$ displaying a notable preference for the box yielding the highest reward, it ensures a more even distribution of trials across other boxes compared to UCB_1. This tactic prevents over-investing trials in the best box in high-variance scenarios, which can be seen as a beneficial characteristic for the minimization of Simple Regret. In the pursuit of maximizing rewards, exploitation is generally advantageous. Nevertheless, according to SR+CR argument, exploration in the root node is often a more desirable trait, after achieving a certain level of confidence in identifying a movement.

6.2 2-Prize Box Experiment

2-Prize Box Experiment illustrates the evaluation problem of Sequential Halving approach in situations with high variance. We define a smaller experiment using two boxes and 2000 trials. These boxes contain rewards of -1 and 1.

In the context of a policy whose aim is to compute the expected reward given a set of choices, this scenario parallels the search process in a Monte Carlo tree. Each node in this tree has an empirical average of the rewards received during search of its child nodes. Assuming that the best strategy for internal nodes is to shift towards evaluating the best moves of players and anticipates responses of opponents. This reasoning guides actions under a scenario where all players strive to maximize their own expected rewards, where exploratory policies diverges from this assessment, because they don't exploit moves.

Table 1: 2-Box Experiment comparing UCB_1 and Sequential Halving.

Policies	Box1	Box2
UCB_1	2	1998
Seq Halving	1000	1000

Table 1 shows the results comparing Sequential Halving and UCB. Box1 have a reward of -1, and Box2 +1. Sequential Halving evaluates this situation in a unrealistic way computing the expected reward to 0, where UCB_1 clearly moves the expected reward towards 1. Sequential Halving perceives the situation as a draw and cannot consider preference of selecting Box1, that should be preferred as much as possible.

This scenario can be translated in two situations that are critic for Sequential Halving as a tree search algorithm (i.e SHOT): the lack of identification of certain winning or inevitable loss; and for high variance scenarios, may not quickly eliminate the obvious boxes that should not be taken. Consequently, it will overspend time on them and even backpropagate unrealistic evaluations.

7 Arena Experiment

To assess the performance of $\text{UCT}_{\sqrt{\text{SH}}}$ and SHOT, we chose five board games for the agents to compete against each other. In these cases, SHOT didn't require the warmup phase, as we had already specified the budget it was expected to use. The agents competed in these games with a fixed number of playouts as their resource. We selected the same five games that were used in the study conducted by Pepels (Pepels 2014).

The selected board games are as follows:

- In **Amazons**, played on a 10x10 chessboard, each player commands four Amazon pieces that move in the same way as chess queens. Each turn consists of a movement action and the shooting of an arrow to block a square on the board. The game is won by the last player capable of making a move.
- **AtariGo**, or first-capture Go, is a Go variant where the first player to capture any stones is declared the winner. These experiments were performed on a 9x9 board.
- **Breakthrough** is played on an 8x8 board where each player starts with 16 pawns. The goal is to move one of these pawns to the opponent's side.
- **NoGo** has the opposite rules of Go. Captures are disallowed and the first player who can't make a move due to this rule loses the game. These experiments were conducted on a 9x9 board.
- **Pentalath** is a connection game on a hexagonal board. The objective is to place 5 pieces in a row. A player's set of pieces can be captured by surrounding them completely.

We conducted three experimental setups of $\text{UCT}_{\sqrt{\text{SH}}}$ against UCT, each setup, agents played 100 matches. Table 2 showcases the results of $\text{UCT}_{\sqrt{\text{SH}}}$ vs UCT. In this specific setup, we carried out tests for 1000, 5000, and 10000 playouts for each move recommendation. Across all games tested, $\text{UCT}_{\sqrt{\text{SH}}}$ consistently outperformed UCT with a higher win rate. Generally, we did not observe a substantial performance differences of $\text{UCT}_{\sqrt{\text{SH}}}$ over UCT when increasing the number of iterations used, excepting for AtariGo which we observed a consistent win rate decreasing, in total 15% drop from 1000 playouts to 10000.

Table 2: Results for $\text{UCT}_{\sqrt{\text{SH}}}$ over UCT.

Game	1000 pl	5000 pl	10000 pl
Pentalath	$64\% \pm 9\%$	$59\% \pm 9\%$	$63\% \pm 9\%$
AtariGo	$70\% \pm 9\%$	$64\% \pm 9\%$	$55\% \pm 9\%$
NoGo	$68\% \pm 9\%$	$71\% \pm 8\%$	$68\% \pm 9\%$
Breakthrough	$59\% \pm 9\%$	$52\% \pm 9\%$	$51\% \pm 9\%$
Amazons	$56\% \pm 9\%$	$50\% \pm 9\%$	$58\% \pm 9\%$

Table 3 presents the results between SHOT and UCT for 1000 and 5000 iterations. On the whole, with 1000 iterations, SHOT appears to outpace UCT in nearly every game except for Amazons, demonstrating win rates exceeding 50% in all games. However, when we raise the number of playouts to 5000, SHOT's performance significantly declines across all games. This pattern could suggest a potential scalability issue in SHOT algorithm and indicate that 1000 iterations may not be enough for UCT to find superior strategies.

Table 3: Results for SHOT over UCT.

Game	1000 playouts	5000 playouts
Pentalath	$74\% \pm 8\%$	$20\% \pm 7\%$
AtariGo	$77\% \pm 8\%$	$41\% \pm 9\%$
NoGo	$63\% \pm 9\%$	$18\% \pm 7\%$
Breakthrough	$58\% \pm 9\%$	$46\% \pm 9\%$
Amazons	$47\% \pm 9\%$	$21\% \pm 8\%$

In Table 4, which illustrates the results of SHOT vs $\text{UCT}_{\sqrt{\text{SH}}}$ for the same iteration counts and game numbers, a similar downward trend is observed when increasing from 1000 to 5000 iterations. In all games except Breakthrough, the win rate for SHOT declines, with the most drastic decrease observed in the NoGo game. Breakthrough is the only game in which SHOT maintains similar performance with an increase in iterations as its win rate improves.

Table 4: Results for SHOT over $\text{UCT}_{\sqrt{\text{SH}}}$.

Game	1000 playouts	5000 playouts
Pentalath	$50\% \pm 9\%$	$25\% \pm 8\%$
AtariGo	$65\% \pm 8\%$	$32\% \pm 9\%$
NoGo	$51\% \pm 9\%$	$5\% \pm 4\%$
Breakthrough	$42\% \pm 9\%$	$52\% \pm 9\%$
Amazons	$38\% \pm 9\%$	$32\% \pm 9\%$

Our assessment of SHOT against UCT showed inferior results when compared with Pepels work, particularly in the game of Amazons. In Pepels's research, SHOT exhibited a 60% win rate in Amazons and 53% in AtariGo with 10000 playouts. Our results varied greatly from these when we escalated the number of playouts from 1000 to 5000. Our data does not suggest that this trend would change if we double the number of playouts to 10000. The probability of SHOT surpassing UCT as the number of playouts increases appears low. In the game NoGo, SHOT had the weakest performance, with UCT$_{\sqrt{\text{SH}}}$ achieving a win rate of 95% at 5000 playouts, while UCT recorded an 82% win rate. With the same number of playouts, UCT$_{\sqrt{\text{SH}}}$ outperformed UCT with a win rate of 71%. In comparison, UCT$_{\sqrt{\text{SH}}}$ approach demonstrates superior performance over both UCT and SHOT.

8 Kilothon Competition

In July 2022, Ludii hosted a competition to promote GGP research and demonstrate the capabilities of the Ludii platform as a competition environment. The competition focused on games implemented using Ludii's GDL from Ludii 1.3.2, with specific properties that defines the scope of games included:

- **Turn-based:** Games in which players take turns to make their moves, ensuring a clear order of play.
- **Adversarial:** Games that involve direct competition between players, where the success of one player is often at the expense of another.
- **Sequential:** Games where players make a series of moves, one after another, and the outcome of the game is determined by the sequence of actions taken.
- **Fully observable:** Games where all players have complete information about the game state, and no hidden information is present.

The Kilothon track, one of the three tracks in the competition, involves competing agents in every game available in the Ludii framework.

Table 5 display the official results of Kilothon competition:

PARTICIPANT	AGENT NAME	PAYOFF
CyprienMD	UBFM Contender	0.231
perrierludo	MCTimeS	0.031
Victor Putrich	SHOT-br	-0.034
Jingyang Zeng	Zkealinvo	-0.456

Table 5: Kilothon Results.

Kilothon participants play 1094 games against an implementation of UCT algorithm native from Ludii. Each agent has a strict one-minute time limit to play each game in its entirety. When the one-minute time limit is reached, the agent must resort to random moves until the game concludes. The UCT agent from Kilothon operates with a fixed thinking time of 0.5 seconds per move. The c value from UCB$_1$ is set to $\sqrt{2}$, and simulations are limited to 500 moves for each player, meaning that when the simulation is interrupted, the playout is considered a draw. To enhance its performance, the Kilothon agent incorporates two modifications to the pure UCT algorithm: Tree Reuse enables the agent to store the search tree from previous plays and reusing it in the future, and Open Loop (Perez Liebana et al. 2015) for dealing with stochastic games, by re-applying actions into states whenever a non-deterministic node is found.

8.1 Kilothon Experimental Settings

For our Kilothon experiments, we implemented our UCT without tree reuse, neither open loop, to compare with UCT$_{\sqrt{\text{SH}}}$. For both we utilize the c parameter of $1/\sqrt{2}$ (see Eq. 3), and a budget of 500 simulations. We tested UCB and UCT$_{\sqrt{\text{SH}}}$ using cbt method.

We conduct 10 Kilothon trials for each agent, computing the overall payoff and the average performance to evaluate their effectiveness in Kilothon.

We pay special attention to the "Board" games category, which contains 1006 of 1116 Kilothon games at Ludii's version 1.3.4. Ludii board games are classified according to (Brice 1954; Parlett 1999) into the following classes: **hunt**, where a player controls more pieces and aims to immobilize the opponent; **race**, where the first to complete a course, with moves controlled by dice or other random elements, wins; **sow or mancala**, where players sow seeds to specific positions and capture opponent seeds; **space**, where players place and/or move pieces to achieve a specific pattern, with possibility of blocks and captures; and **war**, where the goal is to control territory, immobilize or capture all opponent's pieces. The performance of our agents in these game categories gives us a informative view on specific agent abilities.

8.2 Kilothon Results

This section presents the results of our agents in Kilothon. We evaluate our baseline UCT and UCT$_{\sqrt{\text{SH}}}$ with and without cbt.

Table 6 presents the average payoff of our tested agents, along with the maximum payoff achieved by each of them.

AGENT	PAYOFF $\pm$	MAX
UCT$^{cbt}_{\sqrt{SH}}$	0.1512 ± 0.0176	0.1984
UCT^{cbt}	0.0813 ± 0.0334	0.1489
UCT$_{\sqrt{\text{SH}}}$	0.0672 ± 0.0196	0.1019
UCT	-0.0063 ± 0.0168	0.0157
SHOT	-0.0198 ± 0.0121	-0.0053

Table 6: Average payoff in Kilothon.

The results from Table 6 highlights the performance of UCT$_{\sqrt{\text{SH}}}$method over UCT, which achieves better scores than UCT, including when adding the cbt method in both. UCT$^{cbt}_{\sqrt{SH}}$ had the highest score, that could achieve second place in the official competition. In the official competition, the first place achieved 0.231 and the second 0.031.

SHOT had never been tested under a true GGP environment, nor under such time restriction. SHOT achieved the lowest score, which we will not investigate further, but we hypothesize it is a combination of three factors: The $warmup$ model can produce inaccurate estimations for some domains where it is more challenging to make good estimations through Monte Carlo simulations; As the 2-box experiment indicates, states with high variance on the outcomes could lead Sequential Halving to poor estimations; SHOT needs more time to start increasing the quality of its evaluations.

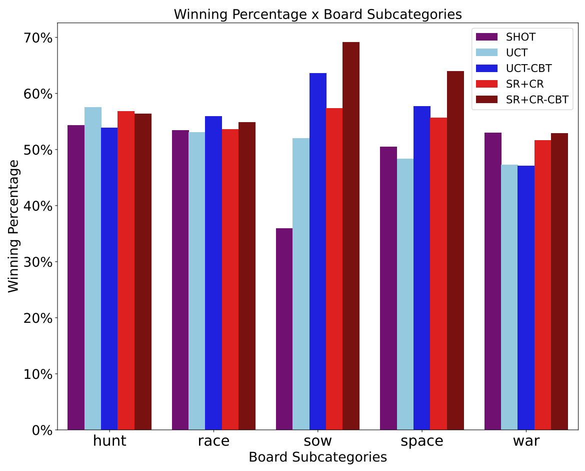

Figure 4: Winning percentage in board subcategories for each agent after 10 Kilothon runs.

Figure 4 presents the winning percentage for each agent, differentiated by board game subcategories. While board games are not the sole category in Kilothon, they constitute a significant portion of the game selection permitted in this competition. The legend of the figure indicates that "UCT-CBT" represents a UCT agent utilizing cbt, while SR+CR agents refer to UCT$_{\sqrt{\text{SH}}}$. The winning percentage is computed as the ratio of games won to the total games played (excluding draws), represented as win/(win+loss).

UCT$_{\sqrt{\text{SH}}}$ exhibits higher winning percentages over the baseline UCT in Sow (+6%), Space (+7%), and War (+4%). In matches against the Kilothon agent, UCT$_{\sqrt{\text{SH}}}$ managed to secure at least 50% of wins across all categories, with respective percentages of (56%, 53%, 57%, 55%, 51%).

Sow and Space games display the highest variability between agents and the top scores attained by UCT$^{cbt}_{\sqrt{SH}}$ (69%, 63%). Both Space and Sow games show significant gains through the *cbt* method for UCT^{cbt} and UCT$^{cbt}_{\sqrt{SH}}$ achieving winning percentages higher than 60%.

SHOT surpassed the baseline UCT in Space (+2%), War (+5%), and achieved over a 50% winning percentage against the Kilothon agent in Hunt (54%), Race (53%), and War (53%). Nonetheless, SHOT had a considerably lower winning percentage of 30% in Sow.

Our evaluations reveal that UCT$_{\sqrt{\text{SH}}}$ strategy, especially with the *cbt* method, outperforms baseline UCT across Kilothon's game categories. The UCT$^{cbt}_{\sqrt{SH}}$ agent got the highest score, showcasing its improvement over the baseline. These results suggest that prioritizing information acquisition in the root node using a hybrid approach improve performance in a multitude of board games. These outcomes highlight the effectiveness of UCT$_{\sqrt{\text{SH}}}$ and *cbt* in enhancing agent performance in time-constrained GGP scenarios.

9 Conclusion

This project developed different approaches to address two key limitations in UCT: (i) the UCT exploitation factor that ensures asymptotic optimality but limits efficient information acquisition under rigid time constraints; and (ii) the adoption of a fixed time-budget per move which might be over or underestimated for lengthy and short games respectively.

To address (i), we explored two distinct methodologies: We propose modifications into SHOT algorithm. Previous research on SHOT suggests that it can surpass UCT in terms of search speed, thereby indicating its potential for enhanced performance in scenarios with limited time. Subsequently, we introduce UCT$_{\sqrt{\text{SH}}}$, a novel MCTS method, that trades asymptotic optimality for timely decision-making. UCT$_{\sqrt{\text{SH}}}$ employs the simulation budget in the root node with more emphasis on exploration, while simultaneously implementing continuous reductions in the range of possible actions. In response to (ii), we propose the Clock Bonus Time strategy to improve time distribution per move, given a fixed time budget for the entire game.

The Prize Box Experiment (Section 6) highlights the disparities in selection strategies, which depend on reward distributions in high and low variance scenarios. While UCB$_1$ struggled to focus on a select subset of moves, the utilization of action elimination at Sequential Halving and UCT$_{\sqrt{\text{SH}}}$ showed a distinct preference for concentrating their efforts on a smaller, more promising subset of the available possibilities.

Conversely, in the high variance scenario, UCB$_1$ allocated a significantly larger number of its trials to the optimal box, while alternative strategies displayed more tendency for exploration. It's important to note that UCT$_{\sqrt{\text{SH}}}$ displayed what we consider to be a more desired behavior compared to UCB$_1$. It demonstrated a clear preference for the most promising option, but still maintained a substantial level of exploration among other alternatives.

The arena experiment (Section 7) provides a snapshot of how the agents perform against each other within a limited GGP environment, featuring five selected board games. From the results displayed in Table 2, we generally observed UCT$_{\sqrt{\text{SH}}}$ securing at least more than than 50% winning against UCT. For instance, in Atarigo, UCT$_{\sqrt{\text{SH}}}$ secured a win rate of 70% with 1000 playouts, but this rate fell to 55% when the playouts increased to 10000. Conversely, Nogo recorded the highest win rate against UCT over other games, reaching 68%.

SR+CR scheme aligns with our initial hypotheses for time-limited games. UCT$_{\sqrt{\text{SH}}}$ outperforms UCT with 1000 playouts, suggesting its suitability for situations requiring fewer simulation runs. However, while SHOT results (Table 3 and 4) also performs well with 1000 playouts, its performance significantly declines when the playouts increase to 5000.

In the Kilothon competition (Section 8), according to result presented at Table 6, our method outperforms the baseline UCT, where *cbt* more than doubles the score for both agents, with UCT$^{cbt}_{\sqrt{SH}}$ achieving second place in the official Kilothon competition. This is noteworthy, given our agent relies solely on Monte Carlo simulations and uses no other enhancements or parallelism, like the ones described at (Świechowski et al. 2022).

SHOT exhibits a weaker performance in both of our performance experiments. Moreover, the 2-box experiment (Section 6.2) provided empirical evidence of SHOT's deviation from Minimax values, which challenges its credibility as a rational model for adversarial agents.

Our implementation of SHOT deviates from the original, as outlined in Section 5.1. We define the warmup phase, enabling SHOT to work in a time constrained GGP environment. In addition, we adopted a simplified approach to budget distribution in an effort to reduce the complexity involved in modeling the algorithm. This divergence could potentially account for the subpar results, thus underscoring the need for further investigation using the original algorithm to validate these observations.

Looking ahead, we plan to extend our research on the $\text{UCT}_{\sqrt{\text{SH}}}$ method and apply it to additional domains. Currently, $\text{UCT}_{\sqrt{\text{SH}}}$ is not equipped to handle stochastic games, as our algorithm constructs a tree without considering nondeterministic nodes. It would be valuable to refine our agent to tackle such games, as well as puzzles, which represent another domain where MCTS algorithms can be better prepared for (Rosin 2011). Furthermore, we aim to assess the scalability of $\text{UCT}_{\sqrt{\text{SH}}}$ beyond simply increasing time and number of playouts. We are interested in exploring whether the application of heuristics and optimization of playouts could further enhance performance. We expect to uncover more ways to improve and broaden the applicability of the $\text{UCT}_{\sqrt{\text{SH}}}$ method, thereby contributing to the ongoing evolution of efficient general game playing strategies.

References

Abramson, B. 1990. Expected-outcome: a general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2): 182–193.

Allis, L. V.; et al. 1994. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2): 235–256.

Brice, W. C. 1954. A History of Board-Games other than Chess. By H. J. R. Murray. Oxford: Clarendon Press, 1952. Pp. viii 287, 86 text figs. 42s. *The Journal of Hellenic Studies*, 74: 219–219.

Browne, C. 2020. AI for ancient games: report on the Digital Ludeme Project. *KI-Künstliche Intelligenz*, 34(1): 89–93.

Browne, C. B. 2008. *Automatic generation and evaluation of recombination games*. Ph.D. thesis, Queensland University of Technology.

Bubeck, S.; Munos, R.; and Stoltz, G. 2011. Pure exploration in finitely-armed and continuous-armed bandits. *Theoretical Computer Science*, 412(19): 1832–1852.

Cazenave, T. 2014. Sequential halving applied to trees. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(1): 102–105.

Cohen-Solal, Q. 2020. Learning to play two-player perfect-information games without knowledge. *arXiv preprint arXiv:2008.01188*.

Fudenberg, D.; and Tirole, J. 1991. *Game Theory*. Cambridge, MA: MIT Press. Translated into Chinesse by Renin University Press, Bejing: China.

Genesereth, M.; Love, N.; and Pell, B. 2005. General game playing: Overview of the AAAI competition. *AI magazine*, 26(2): 62–62.

Karnin, Z.; Koren, T.; and Somekh, O. 2013. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, 1238–1246. PMLR.

Knuth, D. E.; and Moore, R. W. 1975. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4): 293–326.

Kocsis, L.; Szepesvári, C.; and Willemson, J. 2006. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1: 1–22.

Lai, T. L.; Robbins, H.; et al. 1985. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1): 4–22.

Parlett, D. 1999. The Oxford history of board games.

Parlett, D. 2016. What'sa ludeme? *Game & Puzzle Design*, 2(2): 81–84.

Pepels, T. 2014. Novel Selection Methods for Monte-Carlo Tree Search. *Master's thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands*.

Pepels, T.; Cazenave, T.; Winands, M. H.; and Lanctot, M. 2014. Minimizing simple and cumulative regret in monte-carlo tree search. In *Workshop on Computer Games*, 1–15. Springer.

Perez Liebana, D.; Dieskau, J.; Hunermund, M.; Mostaghim, S.; and Lucas, S. 2015. Open loop search for general video game playing. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 337–344.

Piette, É.; Soemers, D. J. N. J.; Stephenson, M.; Sironi, C. F.; Winands, M. H. M.; and Browne, C. 2020. Ludii – The Ludemic General Game System. In Giacomo, G. D.; Catala, A.; Dilkina, B.; Milano, M.; Barro, S.; Bugarín, A.; and Lang, J., eds., *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, 411–418. IOS Press.

Rosin, C. D. 2011. Nested rollout policy adaptation for Monte Carlo tree search. In *Ijcai*, volume 2011, 649–654.

Russell, S.; and Norvig, P. 2020. Adversarial Search. In *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ: Pearson, 4rd edition.

Scheiermann, J.; and Konen, W. 2022. AlphaZero-Inspired General Board Game Learning and Playing. *arXiv preprint arXiv:2204.13307*.

Soemers, D. J.; Mella, V.; Browne, C.; and Teytaud, O. 2021a. Deep learning for general game playing with ludii and polygames. *ICGA Journal*, 43(3): 146–161.

Soemers, D. J.; Mella, V.; Piette, E.; Stephenson, M.; Browne, C.; and Teytaud, O. 2021b. Transfer of fully convolutional policy-value networks between games and game variants. *arXiv preprint arXiv:2102.12375*.

Soemers, D. J.; Piette, E.; Stephenson, M.; and Browne, C. 2019. Learning policies from self-play with policy gradients and MCTS value estimates. In *2019 IEEE Conference on Games (CoG)*, 1–8. IEEE.

Stephenson, M.; Soemers, D. J.; Piette, É.; and Browne, C. 2021. General game heuristic prediction based on ludeme descriptions. In *2021 IEEE Conference on Games (CoG)*, 1–4. IEEE.

Świechowski, M.; Godlewski, K.; Sawicki, B.; and Mańdziuk, J. 2022. Monte Carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, 1–66.

Świechowski, M.; Park, H.; Mańdziuk, J.; and Kim, K.-J. 2015. Recent advances in general game playing. *The Scientific World Journal*, 2015.

Tolpin, D.; and Shimony, S. 2012. MCTS based on simple regret. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, 570–576.

Van Der Werf, E. 2004. *AI techniques for the game of Go*. Citeseer.