A Monte Carlo Algorithm for Time-Constrained General Game Playing

Victor Scherer Putrich[1],
Anderson Rocha Tavares[2], and
Felipe Meneguzzi[3,1]

[1] Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil
`Victor.Putrich@edu.pucrs.br`
[2] Federal University of Rio Grande do Sul, Porto Alegre, Brazil
`artavares@inf.ufrgs.br`
[3] University of Aberdeen, Aberdeen, Scotland
`felipe.meneguzzi@abdn.ac.uk`

Abstract. General Game Playing (GGP) is a challenging domain for AI agents, as it requires them to play diverse games without prior knowledge. In this paper, we develop a strategy to improve move suggestions in time-constrained GGP settings. This strategy consists of a hybrid version of UCT that combines Sequential Halving and $\text{UCB}_{\surd}$, favoring information acquisition in the root node, rather than overspend time on the most rewarding actions. Empirical evaluation using a GGP competition scheme from the Ludii framework shows that our strategy improves the average payoff over the entire competition set of games. Moreover, our agent makes better use of extended time budgets, when available.

Keywords: General Game Playing · Monte Carlo Methods · Sequential Halving.

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 of any specific game being played [5]. GGP agents receive the rules of potentially unknown games and must play them effectively. This prevents the creation of game-specific heuristics.Developing supporting artificial intelligence (AI) techniques for such agents is a step towards real-world agents that handle unpredicted situations.

The Upper Confidence for Trees (UCT) [7] 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 asymptotic optimal convergence. 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.

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

In response, we develop $\text{UCT}_{\sqrt{\text{SH}}}$ [4] (presented at Section 4). Our algorithm is a hybrid approach, based on the Simple Regret plus Cumulative Regret (SR+CR) scheme [15], and Hybrid Monte Carlo Tree Search (H-MCTS) [10], which are presented at Section 3. Both approaches aim to be more exploratory in the root node than UCT, in order to avoid overspending time exploiting, i.e. repeatedly probing the highest-reward movements. Our hypothesis is that more exploration may improve information acquisition, increasing the chance of finding better actions.

To evaluate our agent capabilities, we conduct two distinct experiments (Section 5). First we present the Prize Box Selection experiment, which is a simplified Multi Armed Bandit (MAB) problem to compare how selection polices allocate their resources under scenarios with high and low reward variance. The second experiment aims to measure the agents' performance relative to UCT under time constraints. For this purpose, we use the Ludii GGP environment [12]. Specifically, we use the Kilothon tournament, one of the tracks of Ludii's GGP competition [5]. Such international competitions have a crucial role in motivating GGP research [14].

The main contributions of this paper are as follows: First, we introduce $\text{UCT}_{\sqrt{\text{SH}}}$ algorithm, a new SR+CR method. Second, we propose the Clock Bonus Time (*cbt*) approach, which enhances the estimation of thinking time in a GGP environment. Through the Prize Box Selection experiment, we highlight $\text{UCT}_{\sqrt{\text{SH}}}$ allocation criteria, compared to UCB_1 and other selection policies (examined in our work at Section 2 and 3) in its resilience over dealing with decision-making problems with high and low variance. Lastly, we demonstrate the improved performance of $\text{UCT}_{\sqrt{\text{SH}}}$ over UCT, suggesting its effectiveness as a selection policy.

2 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

[4] https://github.com/schererl/GraduateThesis
[5] https://github.com/Ludeme/LudiiAICompetition

Carlo methods can be used to evaluate a game-tree node by computing the expected outcome of its actions, by sampling a sufficient large number of random completions of a game.

2.1 Regret on Bandit Problem

Multi-Armed Bandit (MAB) problems [1] constitute a category of decision-making scenarios in which the outcomes of chosen actions are unknown. Imagine a casino slot machine with k distinct arms, each with its own reward and probability of winning. 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 bandit 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).

One way to measure performance in the Multi-Armed Bandit problem is 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 of Pepels [10] and Bubeck [3]. 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 trials spent into arm j after 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 study conducted by Bubeck et al. [3], the authors showed that there is a trade-off between minimizing cumulative regret and simple regret. Specifically, they found that a smaller upper bound on cumulative regret (R_n) leads to a higher lower bound on simple regret (r_n), meaning that when an algorithm performs well in terms of cumulative regret (worst-case scenario), it is likely to have a higher minimum simple regret (best-case scenario). 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.

2.2 Upper Confidence Bound

Upper Confidence Bound (UCB) [1] is an exploration policy in MAB problems and MCTS. The policy optimizes cumulative regret over time at a logarithmic rate over the number of trials performed. A widely adopted variant, UCB_1, is favored for its simplicity and its ability to consistently deliver robust performance outcomes.

UCB_1 computes values to actions considering the potential rewards they can achieve. This is accomplished by establishing a confidence interval for the value of the action, a range within which the value can be estimated to lie with high confidence.

The UCB_1 equation, as adapted from Auer et al. [1], is presented below:

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

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 taking action a given state s.

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.

2.3 Sequential Halving

Sequential Halving [6] is a flat, non-exploiting approach[6] for the MAB problem. The algorithm uniformly distributes a predetermined budget among all actions and progressively eliminates the bottom half in terms of performance. While effective at reducing expected simple regret compared to UCB_1, it reduces expected cumulative regret at slower rates than UCB_1.

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 data structure storing the state s, cumulative reward Q, number of visits N, and a list of children from v (returned by CHILDREN if not a leaf). We maintain k for limiting the number of available actions, where HEAD function iterates over

[6] In contrast with exploiting policies, that allocate most resources to the most promising choice, non-exploiting policies allocate resources uniformly among choices, iteratively discarding the poorly-performing ones.

Algorithm 1 The Sequential Halving algorithm (adapted from [6])

```
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:     $k \leftarrow |\text{CHILDREN}(v_{root})|$
6:     while $k > 1$ do
7:         $b \leftarrow \lceil \frac{\mathcal{B}}{k \times \log_2 |\text{CHILDREN}(v_{root})|} \rceil$
8:         for $v' \in \text{HEAD}(\text{CHILDREN}(v_{root}, k))$ do
9:             $Q(v') \leftarrow Q(v') + \text{SIMULATE}(v', b)$
10:            $N(v') \leftarrow N(v') + b$
11:        $\text{SORT}(\text{CHILDREN}(v_{root}), k)$
12:        $k = \lceil k/2 \rceil$
13:    return $\text{CHILDREN}(v_{root})[0]$
```

the first k children from v_{root}. The Sequential Halving formula in Line 7 divides the budget $\mathcal{B}$ by the number of times children can be halved until only one child remains, given by $\log_2 \text{CHILDREN}(v_{root})$. To distribute the budget over the remaining children in the current iteration, $\mathcal{B}$ is also divided by k.

Algorithms like Sequential Halving aim to minimize simple regret by allocating the same budget to all options prior to each halving. Sequential Halving act less "greedily" in perceiving the move with highest immediate reward than UCB_1. This results in a lower bound on simple regret only after completing all simulations. Such algorithms cannot be terminated at any time and have weaker guarantees on the number of suboptimal selections made [6].

2.4 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) employs Monte Carlo simulations to iteratively build a game tree. MCTS is designed to progressively converge on 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 [13]. 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 outlines the MCTS process, starting instanciating a root node, denoted as v_{root}. A node v consists of a state s, the list of applicable actions in s, the parent node, a list of children, and the Q and N values for cumulative reward and visits count, respectively.

The search process involves the following four steps:

- *Selection*: Beginning at the tree root, the selection phase traverses the tree using a *tree policy* (π) that guides the search towards promising nodes. The SELECTION function searches through the tree until it finds a node with untried actions.

Algorithm 2 Pseudocode for MCTS algorithm (adapted from [13])

```
1: function MCTS($s$, $\mathcal{R}$)
2:     Input: State $s$, Resource $\mathcal{R}$
3:     Output: Recommended action
4:     $v_{root} \leftarrow \langle s, \text{ACTIONS}(s) \rangle$
5:     while $r \leq \mathcal{R}$ do
6:         $v_k \leftarrow \text{SELECTION}(v_{root}, \pi)$
7:         $v_{k+1} \leftarrow \text{EXPANSION}(v_k)$
8:         $rw \leftarrow \text{SIMULATION}(v_{k+1}, \pi_{\Delta})$
9:         $\text{BACKPROPAGATION}(v_{k+1}, rw)$
10:        update $r$
11:    return $\text{RECOMMEND}(v_{root})$
```

– *Expansion*: A node is expanded by applying a random untried action to its state, resulting in a new child node. This new node is initialized with the new state, a list of applicable actions, an empty child list, and its parent reference.
– *Simulation*: A playout evaluates the potential reward r of the new node. This is done by following a *default policy* (π_{Δ}), which usually applies random actions until it reaches a terminal state.
– *Backpropagation*: Each node from v_{k+1} up to the root are updated: their Q is updated by rw and N increases by 1.

The search process continues until the algorithm uses up a specified resource $\mathcal{R}$, which can be time or a number of iterations. RECOMMEND function selects a move according to one of three criteria: Max Child, with the highest Q value; Robust Child, with the highest N value; or Max-Robust Child, combining both Q and N values.

The most popular tree policy for the selection phase is the Upper Confidence Bound (UCB$_1$, Section 2.2), which considers each node as an individual MAB problem. When used in MCTS, the algorithm is called Upper Confidence Bound Applied to Trees (UCT). MCTS and UCT exhibit an anytime property, allowing them to recommend useful actions even if the search execution is interrupted.

3 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 searching process. UCB$_1$ has optimal bounds on cumulative regret recommendation, 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 resulting in low information gain [15].

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.

3.1 $\mathbf{UCB}_{\surd}$ and SR+CR

Bubeck et. al [3] 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 when fast recommendations are required. Karning et. al [6] suggest that the more exploratory policies have better bounds on simple regret minimization.

UCB_1 allocates budget based on sample means and often chooses the current top-performing option, leading to a slow reduction in simple regret. Tolpin and Shimony [15] modify UCB_1's policy into $\mathrm{UCB}_{\surd}$. This policy adjusts the UCB_1 formula using a quicker-growing sublinear function, leading to a faster increase in the uncertainty bonus on less visited nodes. The new policy changes the $\ln N_s$ term in UCB_1 to $\sqrt{N_s}$, aiming to narrowing the gap between the selections of non-optimal nodes.

Tolpin and Shimony also point out that nodes closer to the root and those deeper in the tree have different goals. 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 aim to match the value of taking the optimal path, aligning more with minimizing cumulative regret.

The Simple Regret plus Cumulative Regret (SR+CR) 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, $\mathrm{UCB}_{\surd}$+UCT, operates by applying the $\mathrm{UCB}_{\surd}$ policy at the root node and UCB_1 to all child nodes. 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 action's rewards.

3.2 Hybrid Monte Carlo Tree Search

Hybrid Monte Carlo Tree Search (H-MCTS) [10] is a SR+CR algorithm that combines the Sequential Halving Applied on Trees (SHOT) algorithm [4] with UCT. SHOT is a recursive Sequential Halving for building game-trees using a non-exploiting policy. The simple regret minimization at H-MCTS applies SHOT not only at the root node, but also deep down the tree.

The proposed method switches from UCT to SHOT when the computational budget spent in the node achieves a certain threshold, after considering changing the policy to be safe (when a subset of good moves are already identified and evaluated). The algorithm transition between focusing on cumulative regret minimization to simple regret minimization after start growing SHOT at UCT regions that had a sufficient number of visits. Since the computational budget per node is initially small, the simple regret tree remains shallow, as SHOT eliminate nodes from selection, the budget spent increases, causing the SHOT tree to grow deeper.

H-MCTS outperforms UCT for various exploration coefficients [10] 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.

4 Improving UCT in Time-Restricted Scenarios

In this Section, we modify the basic UCT algorithm to improve performance under tight temporal bounds.

4.1 UCT$_{\sqrt{\mathbf{SH}}}$

Although H-MCTS is promising at balacing simple and cumulative regret, it requires a predefined budget for the SHOT portion, which is not possible to estimate for previous unknown domains. Furthermore, by neglecting the exploitation of nodes, the agent becomes prone to excessive resource allocation in unpromising regions.

Algorithm 3 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:     $n \leftarrow |\text{CHILDREN}(v_{root})|$
7:     $h \leftarrow 1$; $k \leftarrow n$
8:     while $r \leq \mathcal{R}$ do
9:         if $k > k_{min}$ and $r > (\mathcal{R}\frac{h}{\log_2 n})$ then
10:            SORT($v_{root}, k$)
11:            $h \leftarrow h + 1$; $k \leftarrow \text{MAX}(k_{min}, k/2)$
12:        $ch \leftarrow \text{CHILDREN}(v_{root})$
13:        $v_s \leftarrow \underset{v \in \text{HEAD}(ch,k)}{\arg\max}\ \pi_{UCB_{\surd}}(v)$
14:        $v_k \leftarrow \text{SELECTION}(v_s, \pi_{UCB_1})$
15:        $v_{k+1} \leftarrow \text{EXPANSION}(v_k)$
16:        $rw \leftarrow \text{SIMULATION}(v_{k+1}, \pi_{\Delta})$
17:        BACKPROPAGATION($v_{k+1}, rw$)
18:        update $r$
19:    RECOMMEND(CHILDREN($v_{root}$))
```

To enhance the performance of UCT under a GGP environment with rigid time constraints, we propose a different SR+CR method, using Sequential Halving and $\text{UCB}_{\surd}$, as shown in Algorithm 3. We take inspiration on the SR+CR scheme and H-MCTS (Section 3), which aim to enhance recommendations based on simple regret minimization near the root.

$\text{UCT}_{\sqrt{\text{SH}}}$ prioritize simple regret minimization at root node by combining $\text{UCB}_{\surd}$ with Sequential Halving eliminations, and the cumulative regret component uses UCT. In $\text{UCT}_{\sqrt{\text{SH}}}$, 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 $\text{UCB}_{\surd}$ 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 halving operation is represented diving k by two. 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. We compute a ratio representing the fraction of halving stages already completed. For that, we divide the halve counter h by the maximum number of halving operations $\log_2 n$. We compute the resource allocation required for the next halving operation multiplying this ratio with the total resource $\mathcal{R}$. After the used resource r surpasses this value, we increment h by one. This method ensures that the same portion of $\mathcal{R}$ is equally divided across all halving stages.

A key distinction from traditional MCTS lies in the separate treatment of root selection. The root selection, depicted at line 13, iterates over the first k-th children, by calling the $\textsc{head}(ch, k)$. The selected child is the one which maximizes $\pi_{UCB_{\surd}}$. Notice that rather than eliminating moves based on the number of visits a node has, we determine when to apply eliminations based on $\mathcal{R}$, which can be time or number of playouts.

4.2 Clock Bonus Time

GGP agents face the challenge of playing games without prior knowledge. In some GGP scenarios, agents have a time budget to play the entire game and must decide how much time to allocate for each move. In Kilothon competition, when the total time budget exhausts, the agent is punished by having its subsequent moves randomly selected. A common strategy is to use a predefined fixed time budget, which can lead to inefficient time management. Agents may lose long games by exhausting their time, or could benefit from using more time to better decisions in shorter-duration games. We propose a method for estimating the time to spend on each move in a GGP environment, using a certain number of simulations during the search to gather information about the game itself. Our model employs a minimum "thinking" time, and for games where the agent can spend more time, it provides a thinking time bonus. The Clock Bonus Time (cbt) formula is as follows:

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

In Eq. 4, G is the total time to play a game, τ_{min} and τ_{max} are the minimum and maximum allowed thinking times per move, respectively. The bonus is given by G divided by the estimated number of moves left to finish the game $\overline{m}$, which we compute using playouts. The max and min functions ensure that the agent performs at least the minimum thinking time and avoids overestimating the time it has. We then discount the new time by τ_{min} because it is a bonus increased to the minimum thinking time. One way to integrate *cbt* with MCTS consists of calling *cbt* after half of τ_{min} has passed, which is when $r \geq \mathcal{R}/2$.

5 Experiments

To evaluate UCT$_{\sqrt{\text{SH}}}$, we conduct two experiments. First, we examine the agent's decision-making in different scenarios of reward variance. This design simulates game situations where making a suboptimal choice significantly affects the outcome (high variance), as well as those where suboptimal choices have a milder effect and are less harmful (low variance). However, in these latter scenarios, a series of misjudgments due to lack of confidence in the most rewarding decision can potentially lead to an overall loss. The experiment examines our agent's performance within a practical context. For this, we leverage a game competition called *Kilothon*, hosted within the *Ludii* environment. This competition serves as a benchmark on UCT$_{\sqrt{\text{SH}}}$ performance in a GGP environment.

5.1 Prize Box Selection Experiment

The Prize Box Selection Experiment 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.

We compare UCB$_1$, Sequential Halving, UCB$_{\surd}$, and UCB$_{\sqrt{\text{sh}}}$ (root policy of UCT$_{\sqrt{\text{SH}}}$), at a scenario with high and low variance in reward's distributions. Both scenarios with 10000 trials for distribute across 30 prize boxes. In the low variance case, we set $\mu = 0.3$ and $\sigma = 0.05$, limited to [-0.5, 0.5]. For the high variance case, we set $\mu = 0.3$ and $\sigma = 0.5$, limited to [-1,1].

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

In this scenario, UCB$_{\surd}$ presents the most dispersed trials among all boxes, with UCB$_1$ following. The use of eliminations in this specific scenario guides the policies that adopt them towards more focused selections, UCB$_{\sqrt{\text{sh}}}$ and Sequential Halving concentrated a higher quantity of resources on a smaller subset of boxes than UCB$_1$ and UCB$_{\surd}$.

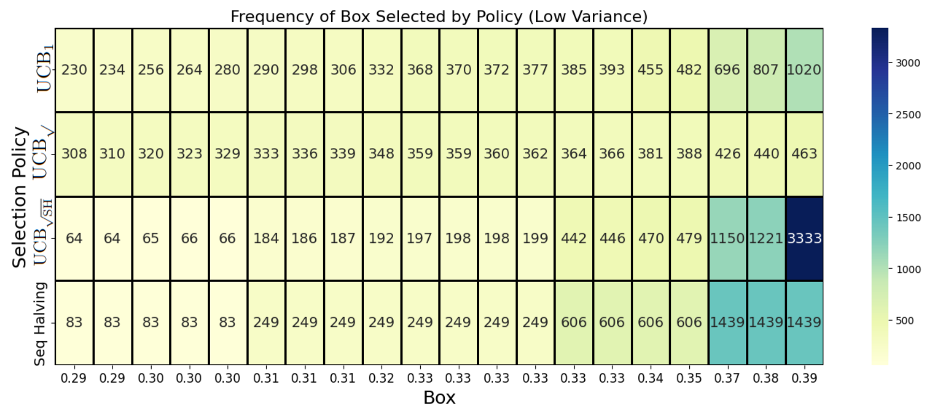

Fig. 1. Box selection frequency under low variance reward distribution.

In the high variance test at Figure 2, both UCB_1 and $\text{UCB}_{\sqrt{\text{sh}}}$ exhibit a stronger preference for the highest rewarded box. While Sequential Halving does not change its selection distribution no matter the reward distribution is, due to its non-exploiting nature. $\text{UCB}_{\sqrt{}}$ and Sequential Halving share more similar frequencies of selection between them, indicating their preference for exploration over UCB_1. Using $\text{UCB}_{\sqrt{\text{sh}}}$ avoids overspending trials on the best box in high-variance scenarios, which can be a desirable characteristic from the perspective of simple regret minimization, although it has a clear preference for the best rewarding box.

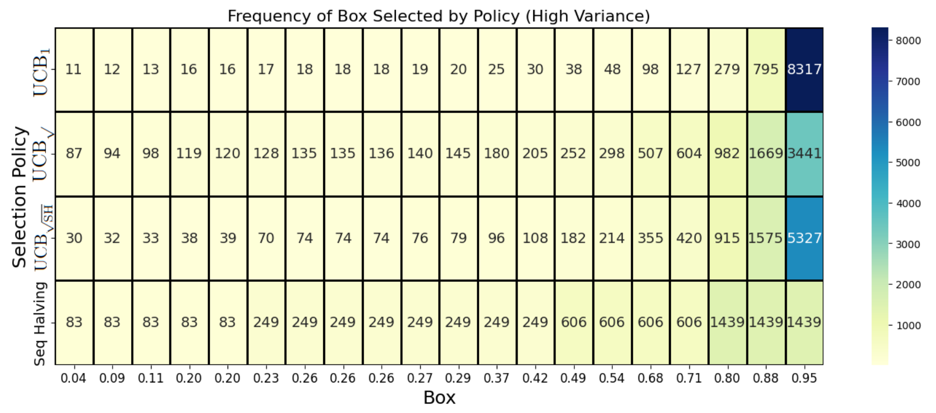

Fig. 2. Box selection frequency under high variance reward distribution.

Our analysis emphasizes that while exploiting (i.e allocating budget to the most promising option) is valuable for reward maximization, exploration is crucial in game playing as it leads to the rapid discovery of beneficial moves. In this context, $\text{UCB}_{\sqrt{\text{sh}}}$ displays a particularly desirable quality of enhanced exploration in our experiments.

Essentially, the primary objective is to identify and execute the optimal move in the game. The frequency of selecting the best move during the search is not of primary concern. Furthermore, $\text{UCB}_{\sqrt{\text{sh}}}$ appears to be less sensitive to varying rewards distributions. This resilience stems from its ability to not overlook exploitation in high variance scenarios, and to focus resources on low variance situations through the application of Sequential Halving eliminations.

5.2 GGP Competition Experiment

General Game Playing (GGP) is a research area focused on developing intelligent agents capable of playing a wide variety of games without prior knowledge of any specific game being played [5]. Ludii is a system for general game research, which has contributed significantly to the field. Games are implemented using Ludii's Game Description Language (GDL). Ludii's GDL is robust and straightforward, it allows researchers and game designers to create new games and even reproduce historical ones [12].

Ludii hosted a GGP competition, where games chosen for the competition were turn-based, adversarial, sequential, and fully observable, including deterministic and stochastic games. Kilothon was one of the competition tracks, where participants play 1094 games against an implementation of UCT algorithm, native from Ludii (which we will refer to as *Kilothon agent*). 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 Kilothon agent uses a fixed thinking time of 0.5 seconds per move, and 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 [11] for dealing with stochastic games.

Results As a baseline, we implemented our UCT version, without tree reuse neither open loop, to compete against the Kilothon agent, and compare results with and $\text{UCT}_{\sqrt{\text{SH}}}$. For both, we use 0.5s of thinking time, and we differentiate agents when using the *cbt* method. We conduct 10 Kilothon trials for each agent, computing the average payoff of our agents to evaluate their effectiveness in Kilothon.

Table 1 presents the average payoff ± standard deviation of our tested agents across all games, along with the maximum payoff achieved by each of them. Our results highlights the performance of $\text{UCT}_{\sqrt{\text{SH}}}$ method over UCT, which achieves better scores than UCT including after adding the *cbt* method. $\text{UCT}^{cbt}_{\sqrt{\text{SH}}}$ had the highest score, that could achieve second place in the official competition, where the first place achieved 0.231, and the second 0.031.

Table 1. Average payoff ± standard deviation and maximum payoff of each agent in Kilothon.

AGENT	PAYOFF ± s.d.	MAX
$\text{UCT}^{cbt}_{\sqrt{\text{SH}}}$	0.1512 ± 0.0176	0.1984
UCT^{cbt}	0.0813 ± 0.0334	0.1489
$\text{UCT}_{\sqrt{\text{SH}}}$	0.0672 ± 0.0196	0.1019
UCT	-0.0063 ± 0.0168	0.0157

The *board* games category, contains the vast majority of games in Kilothon contest. Ludii board games are classified according the following classes classifications [2, 8]: **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.

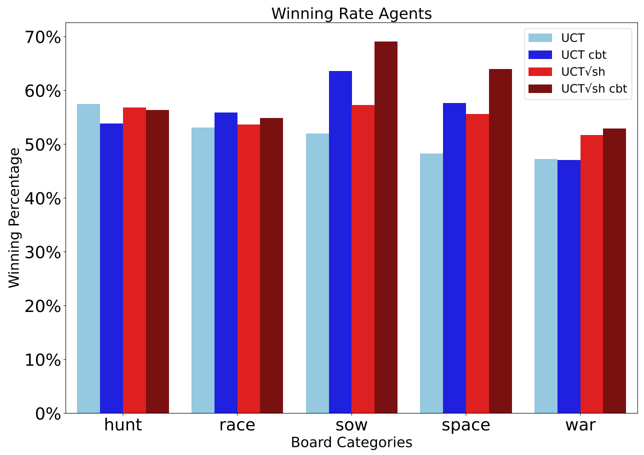

Fig. 3. Winning percentage in board categories for each agent, after running 10 Kilothon tournaments each.

Figure 3 showcases the winning rate of our agents under the five board game categories. The win ratio is computed as win/(win+loss), not including draws.

$\text{UCT}_{\sqrt{\text{SH}}}$ outperforms UCT in Sow (+6%), Space (+7%), and War (+4%) games. Against the Kilothon agent, $\text{UCT}_{\sqrt{\text{SH}}}$ secures the respective win ratios in hunt, race, sow, space and war, respectively: 56%, 53%, 57%, 55%, 51% . Sow and

Space games show high variability among agents, with the highest scores achieved by $\text{UCT}^{cbt}_{\sqrt{\text{SH}}}$ of 69% and 63%. Both these games display significant performance boosts via the *cbt* method for UCT^{cbt} and $\text{UCT}^{cbt}_{\sqrt{\text{SH}}}$, both surpassing a 60% win rate. Our evaluations reveal that the $\text{UCT}_{\sqrt{\text{SH}}}$ strategy, especially with the *cbt* method, outperforms baseline UCT. The $\text{UCT}^{cbt}_{\sqrt{\text{SH}}}$ agent had the highest score, showcasing its improvement over the baseline.

Sampling Five GGP Board Games While Kilothon competition encompassed an extensive variety of games, we examine a subset that fell within the board game category. To this end, we selected games that were also part of a study conducted by Pepels [9].

We compare $\text{UCT}_{\sqrt{\text{SH}}}$ vs UCT, where both agents had 0.5, 1, and 2 seconds of thinking time for each move. Each experiment running over 1000 matches. Table 2 showcases the results.

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

Game	0.5s/move	1s/move	2s/move
Pentalath	$51.7\% \pm 3\%$	$64.9\% \pm 2\%$	$66.8\% \pm 2\%$
AtariGo	$57.0\% \pm 3\%$	$63.6\% \pm 2\%$	$71.9\% \pm 2\%$
NoGo	$61.0\% \pm 3\%$	$66.9\% \pm 2\%$	$79.6\% \pm 2\%$
Breakthrough	$48.8\% \pm 3\%$	$56.2\% \pm 3\%$	$66.9\% \pm 2\%$
Amazons	$46.9\% \pm 3\%$	$60.0\% \pm 3\%$	$70.1\% \pm 2\%$
Overall	$53.0\% \pm 3\%$	$62.3\% \pm 2\%$	$71.0\% \pm 2\%$

Our study reveals the advantages of $\text{UCT}_{\sqrt{\text{SH}}}$ over UCB_1 across various game domains. Although a thinking time of 0.5s appears to be insufficient for $\text{UCT}_{\sqrt{\text{SH}}}$ to gain an edge over UCB_1, which increases significantly after 1 and 2 seconds. In Pentalath, the win rate began at 51.7% and rose to 66.8% as the thinking time increased, while NoGo achieved the highest final win rate of 79.6%. Both Breakthrough and Amazons exhibited a significant increase in win rates, escalating from less than 50% to 66.9% and 70.1%, respectively.

6 Conclusion

This work addresses two drawbacks in UCT, the base method of most general game playing (GGP) agents: (i) UCT exploitation factor guarantees asymptotic optimality but prevents information acquisition under strict time constraints; (ii) the use of fixed time-budget for search per move may be an overestimate and an underestimate for long and short games, respectively.

For (i), we introduce $\text{UCT}_{\sqrt{\text{SH}}}$, a new MCTS method, which foregoes the asymptotic optimality in exchange for a timely response. $\text{UCT}_{\sqrt{\text{SH}}}$ uses the simulation budget more exploratively than traditional UCT, since during the search time the goal is to find the best possible move and return it to the game. For (ii), we present the Clock Bonus Time (*cbt*) strategy to better allocate the search time per move, given a fixed time budget to play the entire game.

We use two experiments to empirically evaluate $\text{UCT}_{\sqrt{\text{SH}}}$ against UCT. The Prize Box Experiment indicates that $\text{UCT}_{\sqrt{\text{SH}}}$ is less sensible to changes in the distribution of rewards as UCB_1 and $\text{UCB}_{\surd}$ do. These latter two have a more spread-out choice allocation when rewards have little variation. However, when there's a lot of variation in the rewards, $\text{UCT}_{\sqrt{\text{SH}}}$ tends to favor the best option, although not as consistently as UCB_1, which almost always selects the top choice.

Although it may appear that constantly selecting the known best option would maximize rewards, as proposed in Tolpin and Shimony's study [15], they argue differently. They suggest that policies that promote more exploration at the root level can actually lead to faster identification of better moves.

In the Kilothon competition, our method exceeded the performance of the baseline UCT. The implementation of the *cbt* strategy more than doubled the scores for both agents, with $\text{UCT}^{cbt}_{\sqrt{\text{SH}}}$ securing the second place in the official Kilothon competition. This achievement is remarkable, considering our agent relies solely on Monte Carlo simulations and does not utilize any other enhancements or parallelism.

Results displayed at Table 2 demonstrate a significant advantage of $\text{UCT}_{\sqrt{\text{SH}}}$ over UCT, this advantage becomes more pronounced when the thinking time is increased to at least 1 second. These findings suggest that using 0.5 seconds of thinking time, as Kilothon agent does, the time constraint imposed might be too unrealistic for agents to play in many game domains.

References

1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine learning **47**(2), 235–256 (2002)
2. Brice, W.C.: 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 (1954). https://doi.org/10.2307/627627
3. Bubeck, S., Munos, R., Stoltz, G.: Pure exploration in finitely-armed and continuous-armed bandits. Theoretical Computer Science **412**(19), 1832–1852 (2011)
4. Cazenave, T.: Sequential halving applied to trees. IEEE Transactions on Computational Intelligence and AI in Games **7**(1), 102–105 (2014)
5. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the aaai competition. AI magazine **26**(2), 62–62 (2005)
6. Karnin, Z., Koren, T., Somekh, O.: Almost optimal exploration in multi-armed bandits. In: International Conference on Machine Learning. pp. 1238–1246. PMLR (2013)
7. Kocsis, L., Szepesvári, C., Willemson, J.: Improved monte-carlo search. Univ. Tartu, Estonia, Tech. Rep **1**, 1–22 (2006)
8. Parlett, D.: The Oxford history of board games. Oxford University Press, Oxford (1999)

9. Pepels, T.: Novel selection methods for monte-carlo tree search. Master's thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands (2014)
10. Pepels, T., Cazenave, T., Winands, M.H., Lanctot, M.: Minimizing simple and cumulative regret in monte-carlo tree search. In: Workshop on Computer Games. pp. 1–15. Springer (2014)
11. Perez Liebana, D., Dieskau, J., Hunermund, M., Mostaghim, S., Lucas, S.: Open loop search for general video game playing. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. pp. 337–344 (2015)
12. Piette, É., Soemers, D.J.N.J., Stephenson, M., Sironi, C.F., Winands, M.H.M., Browne, C.: Ludii – the ludemic general game system. In: Giacomo, G.D., Catala, A., Dilkina, B., Milano, M., Barro, S., Bugarín, A., Lang, J. (eds.) Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020). vol. 325, pp. 411–418. IOS Press (2020)
13. Świechowski, M., Godlewski, K., Sawicki, B., Mańdziuk, J.: Monte carlo tree search: A review of recent modifications and applications. Artificial Intelligence Review pp. 1–66 (2022)
14. Świechowski, M., Park, H., Mańdziuk, J., Kim, K.J.: Recent advances in general game playing. The Scientific World Journal **2015** (2015)
15. Tolpin, D., Shimony, S.: Mcts based on simple regret. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 26.1, pp. 570–576 (2012)