Minimizing Simple and Cumulative Regret in Monte-Carlo Tree Search

Tom Pepels[1], Tristan Cazenave[2], Mark H.M. Winands[1], and Marc Lanctot[1]

[1] Department of Knowledge Engineering, Maastricht University
{tom.pepels,m.winands,marc.lanctot}@maastrichtuniversity.nl
[2] LAMSADE - Université Paris-Dauphine
cazenave@lamsade.dauphine.fr

Abstract. Regret minimization is important in both the Multi-Armed Bandit problem and Monte-Carlo Tree Search (MCTS). Recently, simple regret, *i.e.,* the regret of not recommending the best action, has been proposed as an alternative to cumulative regret in MCTS, *i.e.,* regret accumulated over time. Each type of regret is appropriate in different contexts. Although the majority of MCTS research applies the UCT selection policy for minimizing cumulative regret in the tree, this paper introduces a new MCTS variant, Hybrid MCTS (H-MCTS), which minimizes both types of regret in different parts of the tree. H-MCTS uses SHOT, a recursive version of Sequential Halving, to minimize simple regret near the root, and UCT when descending further down the tree. We discuss the motivation for this new search technique, and show the performance of H-MCTS in six distinct two-player games: Amazons, AtariGo, Ataxx, Breakthrough, NoGo, and Pentalath.

1 Introduction

The Multi-Armed Bandit (MAB) problem is a decision making problem [3] where an agent is faced with several options. On each time step, an agent selects one of the options and observes a reward drawn from some distribution. This process is then repeated for a number of time steps. Generally the problem is described as choosing between the most rewarding arm of a multi-armed slot machine found in casinos. The agent can explore by pulling an arm and observing the resulting reward. The reward can be drawn from either a fixed or changing probability distribution. Each pull and the returned reward constitutes a sample. Algorithms used in MAB research have been developed to minimize *cumulative regret.* Cumulative regret is the expected regret of not having sampled the single best option in hindsight. This type of regret is accumulated during execution of the algorithm, each time a non-optimal arm is sampled the cumulative regret increases. UCB1 [3] is a selection policy for the MAB problem, which minimizes cumulative regret, converging to the empirically best arm. Once the best arm is found by exploring the available options, UCB1 exploits it by repeatedly sampling it, minimizing overall cumulative regret. This policy was adapted to be used in Monte-Carlo Tree Search (MCTS) in the form of UCT [11].

Recently, *simple regret* has been proposed as a new criterion for assessing the performance of both MAB [2,6] and MCTS [7,9,18] algorithms. Simple regret is defined as the expected error between an algorithm's recommendation, and the optimal decision. It is a naturally fitting quantity to optimize in the MCTS setting, because all simulations executed by MCTS are for the mere purpose of learning good moves. Moreover, the final move chosen after all simulations are performed, *i.e.,* the *recommendation*, is the one that has real consequence. Nonetheless, since the introduction of Monte-Carlo Tree Search (MCTS) [11] and its subsequent adoption by games researchers UCT [11], or some variant thereof, has become the "default" selection policy (cf. [5]).

In this paper we present a new, MCTS technique, named Hybrid MCTS (H-MCTS) that utilizes both UCT and Sequential Halving [10]. As such, the new technique uses both simple and cumulative regret minimizing policies to their best effect. We test H-MCTS in six distinct two-player games: Amazons, AtariGo, Ataxx, Breakthrough, NoGo, and Pentalath.

The paper is structured as follows, first MCTS and UCT are introduced in Section 2. Section 3 explains the difference between cumulative and simple regret, and how this applies to MCTS. Next, in Section 4 a recently introduced, simple regret minimizing technique for the MAB problem, Sequential Halving [10], is discussed. Sequential Halving is used recursively in SHOT [7], which is described in detail in Section 5. Together, SHOT and UCT form the basis for the new, hybrid MCTS technique discussed in Section 6. This is followed by the experiments, in Section 7 and finally by the conclusion and an outline of future research, in Section 8.

2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a best-first search method based on random sampling by Monte-Carlo simulations of the state space of a domain [8,11]. In game play, this means that decisions are made based on the results of randomly simulated play-outs. MCTS has been successfully applied to various turn-based games such as Go [16], Lines of Action [20], and Hex [1]. Moreover, MCTS has been used for agents playing real-time games such as the Physical Traveling Salesman [14], real-time strategy games [4], and Ms Pac-Man [13], but also in real-life domains such as optimization, scheduling, and security [5].

In MCTS, a tree is built incrementally over time, which maintains statistics at each node corresponding to the rewards collected at those nodes and number of times they have been visited. The root of this tree corresponds to the current position. The basic version of MCTS consists of four steps, which are performed iteratively until a computational threshold is reached, *i.e.,* a set number of simulations, an upper limit on memory usage, or a time constraint.

Each MCTS simulation consist of two main steps, 1) the *selection* step, where moves are selected and played inside the tree according to the selection policy until a leaf is *expanded*, and 2) the *play-out*, in which moves are played according to a simulation policy, outside the tree. At the end of each play-out a terminal state is reached and the result is *back-propagated* along the selected path in the tree from the expanded leaf to the root.

2.1 UCT

During the selection step, a policy is required to explore the tree to decide on promising options. For this reason, the widely used Upper Confidence Bound applied to Trees (UCT) [11] was derived from the UCB1 [3] policy. In UCT, each node is treated as a bandit problem whose arms are the moves that lead to different child nodes. UCT balances the exploitation of rewarding nodes whilst allowing exploration of lesser visited nodes. Consider a node p with children $I(p)$, then the policy determining which child i to select is defined as:

$$i^* = argmax_{i \in I(p)} \left\{ v_i + C \sqrt{\frac{\ln n_p}{n_i}} \right\}, \tag{1}$$

where v_i is the score of the child i based on the average result of simulations that visited it, n_p and n_i are the visit counts of the current node and its child, respectively. C is the exploration constant to tune. UCT is applied when the visit count of p is above a threshold T, otherwise a child is selected at random.

Note that UCB1 and consequently UCT incorporate both exploitation and exploration. After a number of trials, a node that is identified as the empirical best is selected more often. In tree search, this has three consequences:

1. Whenever a promising move is found, less time is spent on suboptimal ones. Since UCT is generally time-bounded, it is important to spend as much time as possible exploiting the best moves. Due to the *MinMax* principle, which states that an agent aims to maximize its minimum gain, on each ply we expect a player to perform the best reply to its opponent's move.
2. The valuation of any node in the tree is dependent on the values back-propagated. Given that UCT spends less time on suboptimal moves, any values back-propagated are based on increasingly improved simulations, because they are performed deeper in the tree. In fact, given infinite time, UCT converges to almost exclusively selecting nodes with the highest estimates.
3. The current value of the node can be falsified by searching deeper. In UCT, each simulation increases the depth of the search, and as such may reveal moves as becoming worse over time due to an unpredicted turn of events. If an expected good move is not reselected often, such "traps" [15] are not revealed. More generally, when sampling a game-tree rewards are not necessarily drawn from a fixed distribution.

3 Regret

In this section we discuss regret in both the MAB, and MCTS context. The differences between cumulative and simple regret are explained in Subsection 3.1. Next, we discuss regret in the context of MCTS in Subsection 3.2.

3.1 Cumulative and Simple Regret

Suppose a trial is set-up such that a forecaster (a player, or agent) has K actions, which can be repeatedly sampled over $n \in \{1, 2, \cdots, T\}$ trials. Each arm has a mean reward μ_i, and there exists a maximum mean reward μ^*. Suppose further that the forecaster employs a selection policy $I(n)$ that outputs some a to be sampled at time n, and a recommendation policy $J(n)$ that recommends the best arm at time T.

Cumulative regret is defined as the regret of having not sampled the best single action in hindsight,

$$R_n = \sum_{t=1}^{n} \mu^* - \mu_{I(t)}. \tag{2}$$

In other words, the regret is accumulated over time, for each sample the forecaster takes.

Now suppose that we change the experimental set-up, such that the actions chosen on trials $1, 2, \ldots, T-1$ are taken under some realistic "simulated environment" that represents the true on-line decision problem but without committing to the actions. The only *real* decision is made after having played all T simulations. In contrast, *simple regret* [6] quantifies only the regret for the recommendation policy J at time T,

$$r_n = \mu^* - \mu_{J(n)}, \tag{3}$$

i.e., the regret of not having recommended the best action.

Given these definitions, a performance metric for a selection technique can be described as the expected cumulative $\mathbb{E}R_n$ or simple regret $\mathbb{E}r_n$ over different experiments. In their analysis of the links between simple and cumulative regret in MABs, Bubeck *et al.* [6] found that upper bounds on $\mathbb{E}R_n$ lead to lower bounds on $\mathbb{E}r_n$, and that the smaller the upper bound on $\mathbb{E}R_n$, the higher the lower bound on $\mathbb{E}r_n$, regardless of the recommendation policy, *i.e.,* the smaller the cumulative regret, the larger the simple regret. As such, no policy can give an optimal guarantee on both simple and cumulative regret at the same time. In the case of an MAB the strategy used depends on the context of the problem.

3.2 Regret in MCTS

Based on the analysis in Subsection 2.1, the minimization of cumulative regret is naturally suitable to tree search, and the UCB1 selection policy can be used nearly unaltered in this setting as UCT. However, there exist two contexts for the MAB problem, also to be considered in MCTS. These are:

1. Each trial results in a direct reward for the agent. As such we want to minimize the number of suboptimal arms pulled in order to achieve a rewards as high as possible. This relates, for example, to slot machines in a casino. Every choice made at each point in the algorithm has a direct effect on the agent's reward. In this case, the reward of the agent is related to the inverse of its **cumulative regret**.

2. The agent can perform a number of trials, without consequence, in a simulated environment. The agent is allowed T trials in this fashion, after which it must make a recommendation. Based on its recommendation, the agent is rewarded. In this case, the performance of the agent is measured by the **simple regret** of its recommendation. A low simple regret implies that the recommendation is close to the actual best option.

In most MCTS literature, UCT is used as selection policy (cf. [5]), suggesting that only the first context applies. However, the second context is a more natural fit when MCTS is used to play games, because the behavior of the agent in the domain is based solely on its recommendations. Nevertheless, simple regret minimization cannot replace UCT in this case without consideration. Unlike in an MAB, sampling does have an immediate impact on performance in MCTS because reward distributions are non-stationary. Spending more time on suboptimal moves when descending the tree decreases the amount of time available to explore nodes expected to have high rewards. Moreover, since all values are back-propagated, we risk underestimating ancestors based on sampling descendants that are known to be bad. This trade-off was also shown in [18] where the authors use a measure based on the Value of Information (VOI) to determine whether to exploit an expected good move, or continue exploring others. This trade-off is also described as a "separation of exploratory concerns" in BRUE [9].

4 Regret Minimization

Non-exploiting selection policies have been proposed to decrease simple regret at high rates. Given that UCB1 [3] has an optimal rate of cumulative regret convergence, and the conflicting limits on the bounds on the regret types shown in [6], policies that have a higher rate of exploration than UCB1 are expected to have better bounds on simple regret. Sequential Halving (SH) [10] is a novel, pure exploration technique developed for minimizing simple regret in the MAB problem. In this section, both SH and its recursive definition SHOT [7] are discussed.

4.1 Sequential Halving

In many problems there are only one or two good decisions to be identified, this means that when using a pure exploration technique, a potentially large portion of the allocated budget is spent sampling suboptimal arms. Therefore, an efficient policy is required to ensure that inferior arms are not selected as often as arms with a high reward. Successive Rejects [2] was the first algorithm to show a high rate of decrease in simple regret. It works by dividing the total computational budget into distinct rounds. After each round, the single worst arm is removed from selection, and the algorithm is continued on the reduced subset of arms. Sequential Halving (SH) [10], was later introduced as an alternative to Successive Rejects, offering better performance in large-scale MAB problems.

Algorithm 1: Sequential Halving [10].

```
Input: total budget $T$, $K$ arms
Output: recommendation $J_T$
$S_0 \leftarrow \{1, \dots, K\}$, $B \leftarrow \lceil \log_2 K \rceil - 1$
for $k$=0 to $B$ do
    sample each arm $i \in S_k$, $n_k = \left\lfloor \frac{T}{|S_k| \lceil \log_2 |S| \rceil} \right\rfloor$ times
    update the average reward of each arm based on the rewards
    $S_{k+1} \leftarrow$ the $\lceil |S_k|/2 \rceil$ arms from $S_k$ with the best average
return the single element of $S_B$
```

SH divides search time into distinct rounds, and during each round arms are sampled uniformly. After each such round, the empirically worst half of the remaining arms are removed until a single arm remains. The rounds are equally distributed such that each round is allocated approximately the same number of trials (budget), but with smaller subset of available arms to sample. Sequential Halving is detailed in Algorithm 1.

In the next section a recently introduced MCTS technique called SHOT, is discussed which uses SH recursively. This technique is the basis for H-MCTS discussed in Section 6.

5 Sequential Halving Applied to Trees

Sequential Halving applied to Trees (SHOT) [7] is a search technique that utilizes Sequential Halving at every node of the search tree. A difference with regular SHOT and Sequential Halving is that SHOT comes back to already visited nodes with an increased budget. When the search returns to an already visited node, instead of distributing the new budget as if it was a new node, SHOT takes into account the budget already spent at the node and how it was spent. In order to apply Sequential Halving, SHOT considers the overall budget as the already spent budget plus the new budget to spend. It then calculates for each move the budget per move using Sequential Halving with this overall budget. The other difference with simple Sequential Halving is that each move already has an associated number of play-outs coming from the previous visits to the node. In order to take into account this already spent budget, SHOT only gives to each move the difference between the new budget for the move and the budget already spent for the move during previous visits. If the new budget is less or equal to the already spent budget the move is not given any budget for the current round.

SHOT has four beneficial properties: 1) it uses less memory than standard UCT, whereas standard UCT creates a new node for each play-out SHOT only creates a new entry in the transposition table when a node has more than one play-out. In practice, for 19×19 NoGo for example, it means that SHOT uses fifty times less memory than standard UCT. 2) SHOT uses less time descending the tree than UCT. Instead of descending the tree for each play-out, SHOT descends in a child for a possibly large number of play-outs. In practice this means that for the same number of play-outs SHOT was shown to be approximately twice as fast as UCT in the game NoGo [7]. 3) SHOT allocates a possibly large number of play-outs to the possible moves. This makes it easy to parallelize the search without loss of information and without changing the behavior of the algorithm. 4) SHOT is parameter free, contrary to UCT, which requires tuning its C constant. On the negative side, in order to run SHOT the total number of play-outs has to be known in advance. This is less convenient than UCT, which is an any-time algorithm.

6 A Hybrid MCTS

Recall that in the MAB context, in which simple regret minimization is appropriate, only the final recommendation made by an algorithm has effect on the agent's reward. In game play, this holds for the nodes of the search tree at the first ply. Only after running all the allocated simulations a recommendation is made, which affects the state of the game being played. Nodes deeper in the tree have an implicit effect on this decision. Because the shape of an MCTS tree is directly related to the potential reward of internal nodes, promising nodes are selected more often to grow the tree in their direction. This both enforces the confidence of the reward of promising nodes, but also ensures that their reward can be falsified based on results deeper in the tree.

Treating a game tree as a recursive MAB thus reveals different objectives for the distinct plies of the tree. At the root, simple regret should be as low as possible, since the recommendation of the algorithm is based on the first ply of the tree. On deeper plies, we want to both sample efficiently, avoiding time wasted on bad options, and back-propagate correct values from leafs to their ancestors. Where the former can be achieved by using selection policies such as Successive Rejects or Sequential Halving, the latter, as discussed in Section 2 is inherently performed by UCT. Intuitively, this leads to the belief that we should only minimize simple regret at the root, and use UCT throughout of the tree, as suggested by [18]. However, considering that at any node, based on the MinMax principle, we want to find the *best reply* to the action of the parent. It may also be beneficial to ensure a lower simple regret on that particular move because this could intrinsically lead to an improved evaluation of the parent.

Using a selection policy based on both SHOT and UCT, Hybrid MCTS (H-MCTS) combines simple and cumulative regret minimization in a tunable algorithm. The rationale is based on the results in [6], which show that given a low sampling budget, UCB empirically realizes lower simple regret. The proposed technique switches from Sequential Halving to UCT whenever the computational budget is below the budget limit B. Consequently, the search tree is composed of a *simple regret tree* at the root, and *UCT trees* rooted at the leafs of the simple regret tree. As shown in Figure 1, initially the simple regret tree is shallow

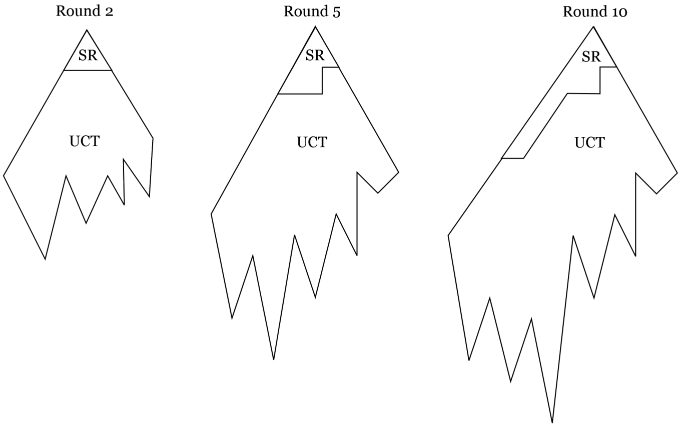

Fig. 1. Example progression of H-MCTS. In the top part of the tree (SR), simple regret is minimized using SHOT. Deeper in the tree, UCT minimizes cumulative regret. The round-numbers represent the Sequential Halving round at the root.

because the computational budget per node is small. Later, when the budget per node increases due to nodes being removed from selection as per Sequential Halving, the simple regret tree grows deeper. Note that since the root's children are sorted in descending order, the left part of the simple regret and UCT tree is always the deepest, since it its root is selected the most.

H-MCTS is outlined in Algorithm 2. Similar to UCT and SHOT, on line 4 terminal conditions are handled, followed by the main feature of the algorithm on line 7 where the initial simulation budget b for each child of the current node is computed. Based on b, a decision is made whether to progress into the UCT tree if $b < B$ or, if $b \geq B$ to continue with SHOT. Note that the $b < B$ check is overridden at the root, since only one cycle is initiated there. Assuming the allocated budget is sufficiently large, at the root simple regret minimization is preferred over cumulative regret minimization. From line 16 the algorithm is similar to the Sequential Halving portion of SHOT. As in SHOT, because multiple play-outs are back-propagated in a single descent from root to leaf, the algorithm returns a tuple t_p, which contains: 1) the number of visits v, and 2) the number of wins per player w_1 and w_2. On line 23, the budget used b_u is incremented by v from the results returned by the recursion. Moreover, the current node's statistics are updated, alongside the cumulative tuple t_p, which are returned to the node's parent. UCT also maintains a tuple of statistics such that it can return the same t_p to the simple regret tree. For the UCT tree, any implementation can be used, as long as it is adapted to return t_p and update the *budgetSpent* value alongside the usual node's visit count because any UCT node in the tree can be "converted" to a simple regret node at any time, when $b > B$ on line 7.

Whenever a UCT node is included in the simple regret tree, all its values are maintained. As such, Sequential Halving has an initial estimate of the values of

Algorithm 2: Hybrid Monte-Carlo Tree Search (H-MCTS).

```
Input: node $p$, allocated budget $budget$
Output: $t_p$: number of play-outs, $p1$ and $p2$ wins
H-MCTS(node $p$, $budget$):
    if $isLeaf(p)$ then $S \leftarrow$ EXPAND($p$)
    $t_p \leftarrow \langle 0, 0, 0 \rangle$
    if $isTerminal(p)$ then
        UPDATE $t_p$, with $budget$ wins for the appropriate player, and
        $budget$ visits
        return $t_p$
    $b \leftarrow \max\left(1, \left\lfloor \frac{p.budgetSpent + budget}{s \times \lceil log_2|S| \rceil} \right\rfloor\right)$
    if $not\ isRoot(p)$ and $b < B$ then
        for $i{=}0$ to $budget$ do
            $\langle v, w_1, w_2 \rangle_i \leftarrow$ UCT($p$)
            UPDATE $p, t_p$ with $\langle v, w_1, w_2 \rangle_i$
        return $t_p$
    $b_u, k \leftarrow 0$
    $S_0 \leftarrow S$
    $s \leftarrow |S|$
    repeat
        for $i{=}1$ to $s$ do
            $n_i \leftarrow$ node $n$ at rank $i$ of $S_k$
            if $b > n_i.visits$ then
                $b_i \leftarrow b - n_i.visits$
                if $i = 0$ and $s = 2$ then
                $b_i \leftarrow \max(b_i, budget - b_u - (b - n_1.visits))$
                $b_i \leftarrow \min(b_i, budget - b_u)$
                $\langle v, w_1, w_2 \rangle_i \leftarrow$ H-MCTS($n_i$, $b_i$)
                UPDATE $p, b_u$, and $t_p$ with $\langle v, w_1, w_2 \rangle_i$
            break if $b_u \geq budget$
        $k \leftarrow k + 1$
        $S_k \leftarrow S_{k-1}$, with the first $s$ elements sorted in descending order
        $s \leftarrow \lceil s/2 \rceil$
        $b \leftarrow b + \max\left(1, \left\lfloor \frac{p.budgetSpent + budget}{s \times \lceil log_2|S| \rceil} \right\rfloor\right)$
    until $b_u \geq budget$ or $s < 2$
    UPDATE $p.budgetSpent$ with $b_u$
return $t_p$
```

the nodes. Based on the budgeting method of SHOT [7], budget is reallocated such that it adheres to Sequential Halving's allocation.

In the scheme presented, a limit on the available budget determines whether to continue in the simple regret tree. However, other methods such as a fixed depth limit for the simple regret tree, or a time-partitioned method, can be viable. Based on the simple regret theory in MABs, pure exploration methods only provide empirically better simple regret than UCB, given a sufficiently large budget. To minimize simple regret given a small budget, UCB with a properly tuned constant should be preferred [6]. Directly applying this result to MCTS means that whenever the available budget is low, UCT with a properly tuned constant should be preferred as selection policy. Therefore, whenever a Sequential Halving round can be initiated with a budget per child higher than B, we continue in the simple regret tree. Otherwise the budget is assigned to UCT, which runs b simulations, and returns the result of their play-outs. Play-outs are only ever initiated in the UCT tree, because UCT immediately takes advantage of the values stored at nodes, whereas Sequential Halving selects all children b times in the first round regardless of their prospects.

As with MCTS, H-MCTS can be separated in four discrete steps:

1. **Budgeting**: A budget is determined for each child. Based on the budget, we enter the UCT tree, or remain in the simple regret tree. If we enter the UCT tree, the four basic MCTS steps apply.
2. **Selection**: In the simple regret tree, nodes are sampled based on Sequential Halving. Nodes in the simple regret tree are assigned a budget, to be spent in their rooted UCT tree, in which play-outs are initiated.
3. **Removal**: Based on the results obtained, children are removed from selection. A new Sequential Halving round starts with half of the best children from the previous round. If the budget is spent, the currently accumulated results are back-propagated.
4. **Back-propagation**: Since H-MCTS is performed depth-first, the final result is only available after all budget is spent. This results in simultaneous back-propagation of numerous results in the simple regret tree.

In this case Sequential Halving is presented as the simple regret algorithm. However, it is certainly possible to replace it with any other algorithm such as Successive Rejects, or any other form of sequential reduction.

H-MCTS shares its disadvantage of not being able to return a recommendation at any-time with SHOT. It must know its exact computational budget beforehand. However, it does make use of the fact that UCT is any-time. Suppose a node was selected and expanded by H-MCTS, then at each time in the simple regret tree, nodes have an appropriate value based on the results back-propagated by UCT. Thus, when SHOT finishes a round by sorting the nodes by their accumulated values on line 27, UCT's any-time property ensures nodes have a representative value.

To a lesser extent, H-MCTS also shares the speed benefit of SHOT. However, because a part of the search is spent in the UCT tree, H-MCTS still spends more time in the tree than SHOT overall. Given a lower budget limit B, H-MCTS can be tuned to run faster by decreasing time spend in the UCT tree.

In the form presented in Algorithm 2, H-MCTS cannot solve proven wins or losses in the simple regret tree. Although we can employ the MCTS-Solver proposed by Winands *et al.* [19] in the UCT tree, this solver is to be adapted to SHOT to be able to solve nodes in the simple regret tree. Such a mechanism has been developed and is presented in [12].

7 Experiments and Results

In this section we show the results of the experiments performed on six two-player games. H-MCTS and the games were implemented in two different engines. Amazons, Breakthrough, NoGo and Pentalath are implemented in a Java based engine. Ataxx and AtariGo are implemented in a *C++* based engine.

- *Amazons* is played on an 8×8 board. Players each have four Amazons that move as queens in chess. Moves consist of two parts: movement, and blocking a square on the board. The last player to move wins the game.
- *AtariGo*, or first-capture Go, is a variant of Go where the first player to capture any stones wins. Experiments are performed on a 9×9 board.
- *Ataxx* is a game similar to Reversi. Played on a square board, players start with two stones each placed in an opposite corner. Captures are performed by moving a stone alongside an opponent's on the board. In the variant used in this paper, jumps are not allowed. The game ends when all squares are filled, or when a player has no remaining stones. The player with the most stones wins. Experiments are performed on a 7×7 board.
- *Breakthrough* is played on an 8×8 board. Players start with 16 pawns. The goal is to move one of them to the opponent's side.
- *NoGo* is a combinatorial game based on Go. Captures are forbidden and the first player unable to play due to this rule, loses. Experiments are performed on a 9×9 board.
- *Pentalath* is a connection game played on a hexagonal board. The goal is to place 5 pieces in a row. Pieces can be captured by fully surrounding an opponent's group.

A uniform random selection policy is used during the play-outs, unless otherwise stated. The C constant, used by UCT (Equation 1) was tuned in each game and was not re-optimized for H-MCTS, both UCT and H-MCTS use the same C constant in the experiments. The budget limit B which determines the switching point between the simple regret and UCT tree, was optimized for each game independently using a range between 10 and 110, with an interval of 20.

7.1 Results

For each table, the results are shown with respect to the first algorithm mentioned in the captions, along with a 95% confidence interval. For each experiment, the players' seats were swapped such that 50% of the games are played as the first player, and 50% as the second, to ensure no first-player or second-player bias. Because H-MCTS cannot be terminated any-time we present only results for a fixed number of simulations. In each experiment, both players are allocated a budget of both 10,000 and 25,000 play-outs.

Table 1 shows results of the matches played by H-MCTS against a standard UCT player. H-MCTS performs best in Amazons, AtariGo, Ataxx, and Pentalath. For Amazons this is in part due to the high branching factor of approximately $1,200$ moves at the start of the match. Since UCT cannot explore and exploit all options in time, Sequential Halving ensures that only a limited subset of the large action-space is under consideration. For NoGo and Breakthrough we see no significant improvement over UCT. This may be due to the fact that these games are more tactical and have narrow winning-lines, and a more exploiting algorithm applies better by identifying good moves and exploiting them fast.

To determine the effect of UCT in H-MCTS, the results of matches played against SHOT are shown in Table 2. H-MCTS shows significant improvement in 10 of the 12 cases. No use is made of the speed benefits of either technique in these experiments. These results give evidence for the claim that H-MCTS makes use of UCT's any-time property to provide better reward estimates in the simple regret tree. Values back-propagated and averaged by using UCT may be more effective than those back-propagated by SHOT. As a benchmark, SHOT played 1,000 matches against UCT per game in Table 3. The results for NoGo differ from those presented in [7], because our experiment is performed using a fixed budget of play-outs for both players, whereas in [7], results are based on time-based experiments. SHOT performs best against H-MCTS and UCT in the games with the highest branching factors, Amazons and AtariGo. This reinforces the evidence that Sequential Halving is best applied in games with high branching factors. In the games with narrow winning-lines such as Breakthrough and Pentalath, SHOT's performance declines against UCT. However, given SHOT's speed improvement over UCT, it is possible that the technique performs better in a time-based experiment.

Game	B	**10,000** **play-outs**	**25,000** **play-outs**
Amazons 8×8	50	**65.2** ± 3.0	**62.0** ± 3.0
AtariGo 9×9	30	**60.6** ± 3.1	**60.6** ± 3.1
Ataxx 7×7	30	52.4 ± 3.1	47.2 ± 3.0
Breakthrough 8×8	70	**53.2** ± 3.1	50.4 ± 3.1
NoGo 9×9	30	52.4 ± 3.1	48.8 ± 3.1
Pentalath	30	46.7 ± 3.1	**54.7** ± 3.1

Table 1. H-MCTS vs. UCT with random play-outs, 1,000 games

Game	B	**10,000 play-outs**	**25,000 play-outs**
Amazons 8×8	50	51.2 ± 3.1	**55.4** ± 3.1
AtariGo 9×9	30	50.0 ± 3.1	**57.5** ± 3.1
Ataxx 7×7	30	**54.5** ± 3.1	**56.0** ± 3.1
Breakthrough 8×8	70	**68.4** ± 2.9	**84.0** ± 2.3
NoGo 9×9	30	**56.3** ± 3.1	**55.5** ± 3.1
Pentalath	30	**62.1** ± 3.0	**78.3** ± 2.6

Table 2. H-MCTS vs. SHOT with random play-outs, 1,000 games

Game	**10,000 play-outs**	**25,000 play-outs**
Amazons 8×8	**60.2** ± 3.0	**55.2** ± 3.1
AtariGo 9×9	**53.8** ± 3.1	**55.7** ± 3.1
Ataxx 7×7	46.7 ± 3.1	40.8 ± 3.1
Breakthrough 8×8	31.2 ± 3.1	16.4 ± 2.3
NoGo 9×9	44.7 ± 3.1	41.4 ± 3.1
Pentalath	33.7 ± 3.0	22.8 ± 2.6

Table 3. SHOT vs. UCT with random play-outs, 1,000 games

In Table 4, an informed play-out policy is used to select moves for Breakthrough. A capture move is four times more likely to be selected than a non-capture one, and a defensive capture (near the winning line) is five times more likely to be selected and (anti-)decisive [17] moves are always played when available. UCT with this play-out policy enabled wins approximately 78% of the games played against UCT with random play-outs. H-MCTS benefits more from the informed play-outs than UCT in Breakthrough, winning up to 56.6% of the games against UCT.

The second part of Table 4 shows results for matches played between the H-MCTS Solver presented in [12] and the MCTS-Solver. Breakthrough employs the heuristic play-out policy, for which we see a significant boost in performance proportional to the allocated budget. Overall, the results show some improvement over Table 1 in Pentalath and NoGo with 25,000 play-outs, although the difference is not sufficient to conclude that the Solver performs better in H-MCTS than in UCT in these games with the same C constant.

8 Conclusion and Future Research

In this paper an MCTS technique is presented based on the results of research in regret theory. The conclusions of the research performed in [6] were interpreted into the form of a Hybrid MCTS technique (H-MCTS). Based on minimizing simple regret near the root, where the overall budget is high, and cumulative

Game	B	**10,000 play-outs**	**25,000 play-outs**
		Heuristic play-outs (no solver)	
Breakthrough 8×8	70	50.4 ± 3.1	**56.6** ± 3.1
		H-MCTS Solver	
Amazons 8×8	50	**65.2** ± 3.0	**64.0** ± 3.0
Breakthrough 8×8	70	**56.7** ± 3.1	**61.3** ± 3.0
NoGo 9×9	30	50.5 ± 3.1	50.6 ± 3.1
Pentalath	70	**53.6** ± 3.1	**55.6** ± 3.1

Table 4. H-MCTS vs. UCT with heuristic play-outs, with/without solver, 1,000 games

regret deeper in the tree [18]. Depending on the available budget during search H-MCTS' simple regret tree can expand deeper to provide better bounds on simple regret on the best replies of its rooted subtrees. The simple regret tree is traversed using SHOT [7]. H-MCTS requires beforehand knowledge of the available play-out budget and therefore cannot be terminated at any time to provide a recommendation. In tournament-play, when search time is strictly limited, an approximation of the number of simulations per second can be used to determine the available play-out budget.

H-MCTS performed better against SHOT given the same allocation of play-outs in 10 out of 12 experiments. Moreover, results show that in different two-player games, H-MCTS performs either better, or on par with UCT. In Amazons, AtariGo, and Pentalath H-MCTS outperforms UCT by up to 65.2%. In Breakthrough using an informed play-out policy H-MCTS outperformed UCT by up to 61.3% using the solver technique.

Although the hybrid technique is founded on theoretical work in both the MAB context and MCTS, we have not shown that it provides better bounds on simple regret compared to other techniques. This is work for future research. In order to show that H-MCTS exhibits lower simple regret in practice, it should be validated in smaller, proven games for which the game-theoretic value of each action is known. Moreover, investigation is required regarding the effects of the budget limit B in relation to the total allocated number of play-outs, and the interrelation between H-MCTS' B, and UCT's C constant. In the experiments presented in this paper, both were fixed per game, rather than per experiment. Finally, the speed benefits of H-MCTS, combined with parallelization is open to investigation. H-MCTS can be parallelized efficiently by dividing budgets in the simple regret tree over multiple threads [7].

Acknowledgments

This work is partially funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project Go4Nature, grant number 612.000.938.

References

1. Arneson, B., Hayward, R., Henderson, P.: Monte-Carlo tree search in Hex. IEEE Trans. Comput. Intell. AI in Games 2(4), 251–258 (2010)
2. Audibert, J., Bubeck, S., Munos, R.: Best arm identification in multi-armed bandits. In: Proc. 23rd Conf. on Learn. Theory. pp. 41–53 (2010)
3. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning 47(2-3), 235–256 (2002)
4. Balla, R.K., Fern, A.: UCT for tactical assault planning in real-time strategy games. In: Boutilier, C. (ed.) Proc. of the 21st Int. Joint Conf. on Artif. Intel (IJCAI). pp. 40–45 (2009)
5. Browne, C., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte-Carlo tree search methods. IEEE Trans. on Comput. Intell. AI in Games 4(1), 1–43 (2012)
6. Bubeck, S., Munos, R., Stoltz, G.: Pure exploration in finitely-armed and continuous-armed bandits. Theoretical Comput. Sci. 412(19), 1832–1852 (2010)
7. Cazenave, T.: Sequential halving applied to trees. IEEE Trans. on Comput. Intell. AI in Games (2014), in Press
8. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M. (eds.) Proc. 5th Int. Conf. Comput. and Games. LNCS, vol. 4630, pp. 72–83 (2007)
9. Feldman, Z., Domshlak, C.: Simple regret optimization in online planning for markov decision processes. CoRR abs/1206.3382 (2012)
10. Karnin, Z., Koren, T., Somekh, O.: Almost optimal exploration in multi-armed bandits. In: Proc. of the Int. Conf. on Mach. Learn. pp. 1238–1246 (2013)
11. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) Euro. Conf. Mach. Learn., LNAI, vol. 4212, pp. 282–293. Springer-Verlag (2006)
12. Pepels, T.: Novel Selection Methods for Monte-Carlo Tree Search. Master's thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands (2014)
13. Pepels, T., Winands, M.H.M., Lanctot, M.: Real-time Monte-Carlo tree search in Ms Pac-Man. IEEE Trans. Comp. Intell. AI Games (2014), accepted
14. Powley, E.J., Whitehouse, D., Cowling, P.I.: Monte Carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem. In: IEEE Conf. Comput. Intell. Games. pp. 234–241. IEEE (2012)
15. Ramanujan, R., Sabharwal, A., Selman, B.: Understanding Sampling Style Adversarial Search Methods. In: Proceedings of the Conference on Uncertainty in Artificial Intelligence. pp. 474–483 (2010)
16. Rimmel, A., Teytaud, O., Lee, C., Yen, S., Wang, M., Tsai, S.: Current frontiers in computer Go. IEEE Trans. Comput. Intell. AI in Games 2(4), 229–238 (2010)
17. Teytaud, F., Teytaud, O.: On the huge benefit of decisive moves in Monte-Carlo Tree Search algorithms. In: IEEE Conference on Computational Intelligence and Games. pp. 359–364. IEEE (2010)
18. Tolpin, D., Shimony, S.: MCTS based on simple regret. In: Proc. Assoc. Adv. Artif. Intell. pp. 570–576 (2012)
19. Winands, M.H.M., Björnsson, Y., Saito, J.T.: Monte-Carlo Tree Search Solver. In: Proc. Comput. and Games. LNCS, vol. 5131, pp. 25–36 (2008)
20. Winands, M.H.M., Björnsson, Y., Saito, J.T.: Monte Carlo Tree Search in Lines of Action. IEEE Trans. Comp. Intell. AI Games 2(4), 239–250 (2010)