NOVEL SELECTION METHODS FOR MONTE-CARLO TREE SEARCH

Tom Pepels

Master Thesis 14-14

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Faculty of Humanities and Sciences
of Maastricht University

Thesis committee:

Dr. Mark H.M. Winands
Dr. Marc Lanctot

Maastricht University
Department of Knowledge Engineering
Maastricht, The Netherlands
2 July 2014

Preface

In this thesis I present the result of my investigation into regret minimization for Monte-Carlo Tree Search. The thesis presents the motivation, background, and formal definition of a novel search technique based on minimizing both simple and cumulative regret in a game tree: Hybrid MCTS (H-MCTS). The technique minimizes the two types of regret in a single search-tree. This ensures that recommendations made by the algorithm have a low simple regret, and at the same time internal nodes are sampled efficiently. It was developed for, and tested in six two-player games: Amazons, AtariGo, Ataxx, Breakthrough, NoGo, and Pentalath. The research was performed at the Department of Knowledge Engineering, Maastricht University, The Netherlands.

Special thanks goes to both Dr. Mark Winands and Dr. Marc Lanctot for providing the inspiration and guidance required to develop this novel algorithm. Their combined experience was crucial to obtain the results presented in this work. Thanks goes to Prof. Dr. Tristan Cazenave for his time and assistance with the implementation of SHOT, and for the experiments he performed in his award-winning engine. Thanks also to Dr. Steve Kroon for his insightful input and assistance in proof-reading the work. Moreover, I would like to thank my wife Priscilla for her support, and for her patience and understanding. Without both her emotional and financial assistance you would not be reading this thesis.

Tom Pepels
Maastricht, June 2014

Summary

Monte-Carlo Tree Search (MCTS) is a best-first search technique, which bases decisions on sampling the state-space of a domain. In different domains, MCTS has proven to be an effective approach when complex decision-making based on future rewards and outcomes is required. The technique was initially inspired by algorithms used to solve multi-armed bandit (MAB) problems. Such a problem can be described as a single-ply MCTS search, in which an agent is given a choice of options (arms), each with their own probability distribution. Sampling an arm returns a random result from its underlying distribution, and the goal of the agent is to maximize its reward and/or provide a recommendation of which arm has the most rewarding distribution.

Based on the context of the MAB problem, the agent's goal is to either minimize simple regret, *i.e.,* the regret of not recommending the best action, or cumulative regret, *i.e.,* regret accumulated over time. Applying this theory to MCTS however, may require more consideration. In a recursive MAB (such as MCTS), where the distribution of each arm is based on an underlying growing search-tree, minimizing a single type of regret throughout the tree implies that at each ply of the tree this specific type of regret minimization is optimal. However, when MCTS is applied to games, the behaviour of the agent in the domain is based solely on its recommendations, *i.e.,* the moves it makes in the match, implying that a recommendation made by the algorithm should have an low as possible simple regret.

The majority of MCTS research uses the UCT selection policy, which minimizes cumulative regret. Other techniques have been proposed to minimize simple regret in MCTS. Based on recent discoveries in simple regret theory in MABs, and two recently introduced MCTS variants, a new MCTS variant is introduced: Hybrid MCTS (H-MCTS). H-MCTS uses different selection policies to specifically minimize both types of regret in different parts of the tree. H-MCTS is inspired by the notion that at the root simple regret is a more natural quantity to minimize. Since all recommendations made by MCTS are based on the values of the root's children, we want the lowest possible simple regret for these nodes. However, deeper down the tree cumulative regret has several beneficial properties which ensure the root's children are properly evaluated. To construct a combination of selection policies, H-MCTS uses SHOT, a recursive version of Sequential Halving, to minimize simple regret near the root, and UCT when the overall budget is lower.

A solver is introduced for SHOT and H-MCTS, allowing wins and losses to be proven in the tree. The solver is designed such that it does not disrupt the budget allocations of Sequential Halving, budget that is unspent when a proven node is encountered is spent during later rounds. Finally, an enhancement is introduced for SHOT, which excludes known bad moves from selection. This enhancement ensures that the majority of the allocated budget is spent on promising moves.

The performance of H-MCTS and the proposed enhancements is assessed in six distinct two-player games: Amazons, AtariGo, Ataxx, Breakthrough, NoGo, and Pentalath. H-MCTS improves performance significantly over UCT in Amazons, AtariGo, Ataxx, and Pentalath when random play-outs are used. Moreover, in Breakthrough H-MCTS outperforms UCT when using an informed play-out policy.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Chapter contents: An overview of Monte-Carlo Tree Search and the main topic of this thesis, simple regret minimization applied to Monte-Carlo Tree Search. Moreover, the problem statement and research questions are drafted and a general outline of the structure of the thesis is given.

1.1 Artificial Intelligence and Games

Decision-making and problem solving have been core topics in Artificial Intelligence (AI) since its birth over half a century ago. In many domains an agent is required to find a specific sequence of actions to achieve a certain goal. A search algorithm can be used to explore the state space to find rewarding states in the future and determine the best action given the current state. Given that most real domains are too complicated to result in a limited, specific set of rules for the agent to follow, an abstract domain is more appropriate when investigating search algorithms. For a single agent, puzzles, graph problems and simplified real-world models are often investigated. In this case, the agent's goals are non-adversarial, and it takes to maximize its utility over time to reach a set goal. When more than one agent is involved, games provide adversarial challenges with simple rules that result in large and complex state spaces. For most interesting games, an exhaustive search is not feasible, but heuristics and approximation techniques have been developed.

Even before the first computer capable of playing games at a reasonable level was developed, Alan Turing was thinking about computer chess (Turing, 1953). Over the decades, faster computers allowed for deeper investigation into game-playing algorithms such as $\alpha\beta$ (Knuth and Moore, 1975), Principal-Variation Search (Marsland, 1983), Proof-number search (Allis, Van der Meulen, and Van den Herik, 1994). One of the reasons game AI research has sparked interest over the years is that its techniques can be directly measured against human players. In 1997, DEEP BLUE (Campbell, Hoane Jr, and Hsu, 2002) defeated then-reigning world chess champion Garry Kasparov in a six-game match, the first time a computer beat the human champion. After plentiful research had been performed in computer chess, Go was the next target for game AI research. In contrary to chess, for Go it is not straightforward to find a decent evaluation function. Moreover, in Go, over the course of the game stones can be played anywhere on the board leading to a high branching factor.

With the introduction of Monte-Carlo Tree Search (Kocsis and Szepesvári, 2006; Coulom, 2007), and UCT (Kocsis and Szepesvári, 2006), researchers could reach expert-level play in Go (Gelly and Silver, 2008; Rimmel *et al.*, 2010) on small boards. Since it requires no static heuristic evaluation, using simulations to determine the rewards of states in the tree, and a selection policy to explore the tree, MCTS performed better than any algorithm had before (Lee *et al.*, 2009). The success of MCTS and UCT in Go sparked researchers' interests in developing a better understanding of the algorithm and applying it to different domains ranging from games, planning problems and real-time domains (cf. Browne *et al.*, 2012).

1.2 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 for a specified domain (Coulom, 2007; Kocsis and Szepesvári, 2006). In gameplay, 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 (Gelly and Silver, 2008; Rimmel *et al.*, 2010), Lines of Action (Winands, Björnsson, and Saito, 2010), and Hex (Arneson, Hayward, and Henderson, 2010). Moreover, MCTS has been used for agents playing real-time games such as the Physical Traveling Salesman (Powley, Whitehouse, and Cowling, 2012), and Ms Pac-Man (Pepels, Winands, and Lanctot, 2014), and real-time strategy games (Balla and Fern, 2009), but also in real-life domains such as optimization, scheduling, and security (Browne *et al.*, 2012).

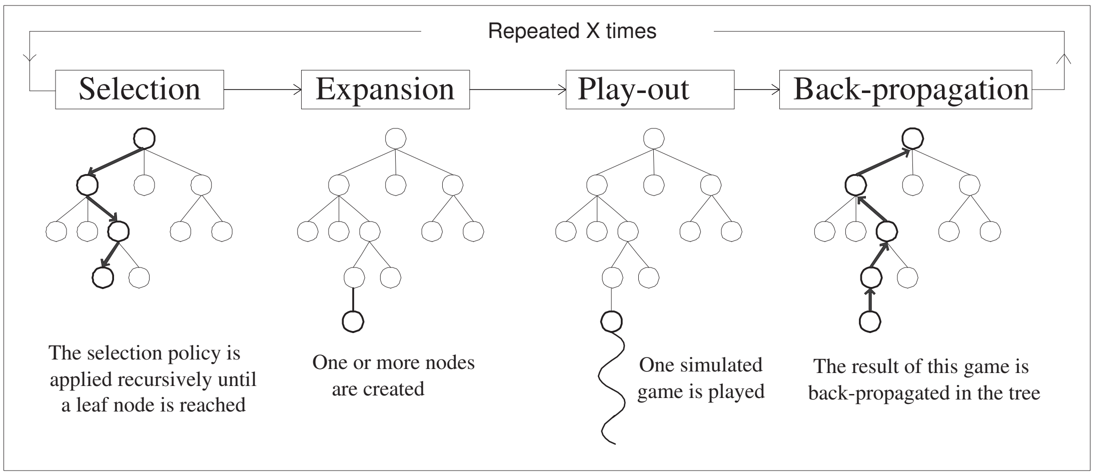
Repeated X times
Selection
Expansion
Play-out
Back-propagation
The selection policy is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is back-propagated in the tree

Figure 1.1: Strategic steps of Monte-Carlo Tree Search (Chaslot *et al.*, 2008).

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. When using a bandit-based method to select moves, such as in UCT, MCTS resembles a recursive multi-armed bandit. 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 iterations, an upper limit on memory usage, or a time constraint. The basic version of MCTS consists of four steps (Chaslot *et al.*, 2008):

- **Selection**. Starting at the root node, children are selected recursively according to a selection policy. When a leaf node is reached that does not represent a terminal state it is selected for expansion.
- **Expansion**. One or more children are added to the selected leaf node.
- **Play-out**. A simulated play-out is performed, starting from the state of (one of) the added node(s). Moves are performed randomly or according to a heuristic strategy until a terminal state is reached.
- **Back-propagation**. The result of the simulated play-out is propagated immediately from the added node back up to the root node. Statistics are updated along the tree for each node selected during the selection phase and visit counts are increased.

The combination of moves selected during the selection step, and the play-out form a single simulation. During the selection step, moves are executed according to the nodes selected in the tree, and during play-out moves are performed randomly, or according to some play-out policy.

Because results are immediately back-propagated, MCTS can be terminated any time to determine the decision to be made. Moreover, no static heuristic evaluation is required when simulations reach an end state. However, in most cases it is beneficial to add domain knowledge for choosing moves made during the play-out.

The main benefit of MCTS is that it requires no explicit heuristic state evaluation. Rather, states are evaluated by repeatedly sampling them and measuring an average reward. It is often beneficial to add some domain knowledge to the play-outs such that the simulated games better approximate good play. Moreover, many enhancements to MCTS have been proposed to improve the general performance of the algorithm. Some of the most notable enhancements are: MCTS-Solver (Winands, Björnsson, and Saito, 2008), which recognizes solved wins and losses in the tree and back-propagates their values, and RAVE (Gelly and Silver, 2007), used to speed up node valuation in the tree. Moreover, several on-line learning techniques for play-out policies have been proposed, such as low-level $\alpha\beta$ searches (Winands and Björnsson, 2011), the Last-Good-Reply policy (Baier and Drake, 2010), Move-average Sampling Technique (MAST) (Finnsson and Björnsson, 2008), and N-grams (Tak, Winands, and Björnsson, 2012). Combining these enhancements often offers greatly improved play by MCTS.

1.3 Regret Minimization

Algorithms used in multi-armed bandit research have been developed to minimize *cumulative regret.* Cumulative regret is the expected regret of not having sampled the optimal decision. This type of regret is accumulated during execution of the algorithm, each time a non-optimal arm is sampled the cumulative regret increases. UCB1 (Auer, Cesa-Bianchi, and Fischer, 2002) is a selection policy for the MAB problem, which minimizes cumulative regret at a fast rate, converging to the empirically best arm fast. Once a candidate best arm is found by exploring the available options, UCB1 exploits it by repeated sampling. This policy was adapted to be used in MCTS in the form of UCT (Kocsis and Szepesvári, 2006).

Recently, *simple regret* has been proposed as a new criterion for assessing the performance of both MAB (Audibert, Bubeck, and Munos, 2010; Bubeck, Munos, and Stoltz, 2010) and MCTS (Tolpin and Shimony, 2012; Feldman and Domshlak, 2012) algorithms. Simple regret is defined as the expected error between an algorithm's recommendation and the optimal decision. Simple regret is a naturally fitting quantity to optimize in the MCTS setting, since all simulations executed by MCTS are for the mere purpose of learning good moves. However, the final move chosen after all simulations are performed, *i.e.,* the *recommendation*, is the one that has real consequence. Therefore, the choice of this move should have as low as possible simple regret. Moreover, once MCTS finds a good move with high certainty, the utility of re-selecting that move diminishes over time. When a promising node, or group of nodes is visited too often, at a certain point insufficient simulation time remains to determine whether there exists a viable alternative. Rather it may be favourable to explore other options sooner, even if a single move is identified as the best. At the same time, MCTS should not 'waste' its time on moves that are expected to be bad.

This is the driving idea behind pure exploration algorithms such as Successive Rejects (Audibert *et al.*, 2010) and Sequential Halving (Karnin, Koren, and Somekh, 2013). Contrary to UCB, these algorithms have no specific exploitation phase, they divide their time uniformly between a continuously reduced set of options. The final recommendation is the single arm that remains after all trials are finished. These pure exploration techniques form the basis of the simple regret argument for MCTS presented in this thesis.

1.4 Problem Statement and Research Questions

When MCTS is applied to games it is only the final recommendation, *i.e.,* the actual move played, that has an impact. To this end, we want the probability of selecting a suboptimal move, *i.e.,* the expected simple regret, to be as low as possible. Based on this assumption, simple regret minimization can be a better way to determine a move's utility. As such, a method that minimizes both types of regret in their appropriate settings can improve the overall performance of MCTS. Simple regret optimization (pure exploration) has practical problems, especially at low search times because it spends more time on suboptimal options, whereas techniques based on cumulative regret minimization perform particularly well under such limited circumstances. Currently, several algorithms have been proposed to minimize simple regret. However, some do not improve performance in games, while others only provide benefits in specific circumstances.

The problem statement of the thesis is:

How can a Monte-Carlo Tree Search variant be constructed to minimize both simple and cumulative regret effectively?

The following four research questions arise from this problem statement:

1. *How can a search technique minimize both simple, and cumulative regret in a game tree?*
2. *When should selection switch from simple to cumulative regret minimization in the tree?*

To answer these questions, an investigation into state-of-the-art multi-armed bandit algorithms is performed. Two MCTS variants that minimize simple regret, SHOT (Cazenave, 2014) and the SR+CR scheme (Tolpin and Shimony, 2012) are discussed. These algorithms provide the inspiration and technique for a new Hybrid MCTS (H-MCTS). Given the new Hybrid technique, the initial assumption regarding simple regret minimization in games ought to be verified:

3. *Do pure exploration selection policies in H-MCTS improve performance in two-player games?*

This question is answered by performing experiments in six distinct two-player games: Amazons, AtariGo, Ataxx, Breakthrough, NoGo, and Pentalath. The performance of H-MCTS' is compared to both UCT and SHOT in these games.
The last research question relates to an existing MCTS enhancement called MCTS-Solver (Winands *et al.*, 2008), which back-propagates proven wins and losses in the game tree:

4. *How can the MCTS-Solver be adapted to work with H-MCTS and SHOT?*

To address the last research question, the MCTS-Solver is adapted and implemented in SHOT and H-MCTS.

1.5 Thesis Outline

The thesis opens with two introductory chapters introducing and discussing the background of the theory used. **Chapter 2**, which discusses the Multi-armed Bandit problem and its application to MCTS, and **Chapter 3**, in which current simple regret minimizing MCTS algorithms are discussed. The former details the exact difference between simple and cumulative regret, and how these types of regret may be minimized using different selection policies, and the latter shows how these selection policies are applied to MCTS.

Next, **Chapter 4** goes into detail on H-MCTS. Based on the analysis in the previous chapters, the new algorithm is defined and described. Moreover, the problems and shortcomings of the algorithm are discussed, and possible solutions provided. In **Chapter 5** the proposed algorithm is tested on six two-player games: Amazons, AtariGo, Ataxx, Breakthrough, NoGo, and Pentalath. **Chapter 6** concludes the thesis, and offers directions for future research and improvements.

Chapter 2

Regret and Multi-Armed Bandits

Chapter contents: The foundation of regret minimization for Monte-Carlo Tree Search (MCTS) is given. Given that UCT is defined as a recursive multi-armed bandit (MAB) algorithm, regret minimization is discussed in this context first. Two algorithms designed to minimize simple regret in MABs are outlined. Moreover, the link between simple and cumulative regret is detailed and recent findings are discussed.

2.1 Introduction

The multi-armed bandit (MAB) problem is defined as a stochastic decision-making problem (Robbins, 1952). An agent is faced with several options, each with their own reward distribution. Based on sampling the search space an agent is to select the option with the best reward distribution. 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 is drawn from a fixed probability distribution. Each pull and the returned reward constitutes a sample. When combined with MCTS, the MAB problem becomes both recursive and non-stationary which is explained in Chapter 3.

In the classic MAB setting, the goal is to maximize the cumulative sum of rewards, *e.g.,* winning the most money in the slot machine example. Since the agent does not know the distribution of the arms beforehand, he has to *explore* the possible choices, and when a rewarding arm is found, *exploit* this option to gain a high total reward. Generally, after a certain limit, *e.g.,* a time-span or number of trials, the agent must return a recommendation to determine the best arm.

The performance of the agent can be evaluated by observing the difference in the rewards obtained over time and the theoretical reward obtained by pulling only the true best arm, called *cumulative regret*. Or, in the case of *simple regret*, observing the difference between the recommended arm and the true best arm, after the forecaster makes its final recommendation.

In this chapter the different facets of the MAB problem are discussed and related to MCTS. In Section 2.2 a formal definition of the two types of regret is given. Next, Section 2.3 discusses the well-known UCB1 selection policy and its relation to MCTS. This is followed by a review of two recently introduced selection policies for MAB aimed at minimizing simple regret, in Section 2.4.

2.2 Regret and The Multi-Armed Bandit Problem

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 an 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 selects the best arm at time T.

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

$$R_n = \sum_{t=1}^{n} \mu^* - \mu_{I(t)}. \tag{2.1}$$

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 at step T after having played $T-1$ simulations. In contrast, *simple regret* (Bubeck *et al.*, 2010) quantifies only the regret for the recommendation policy J at time T,

$$r_n = \mu^* - \mu_{J(n)}, \tag{2.2}$$

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, Bubeck *et al.* (2010) 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.

2.3 Upper Confidence Bounds

Auer *et al.* (2002) proposed a finite time strategy for the MAB problem. Upper Confidence Bounds (UCB) optimizes cumulative regret over time at an optimal logarithmic rate, without knowledge of the reward distributions. The selection policy consists of two terms, 1) the current average reward, and 2) the size of the one-sided confidence interval for the average reward. For each arm, UCB1 gives an upper bounding value, below which, the true expected reward of the arm falls with high probability. The UCB1 selection policy is outlined in Algorithm 1.

Algorithm 1: Upper Confidence Bounds (UCB1) (Auer *et al.*, 2002).

```
Input: total budget $T$, $K$ arms
Output: recommendation $J_T$
1 play each arm once, for each arm $i$, update $\bar{x}_i$ with the obtained reward, and $n_i \leftarrow 1$
2 for $t=1$ to $T$ do
3     Sample arm $i$ that maximizes $\bar{x}_i + \sqrt{\frac{2 \ln t}{n_i}}$
      $\bar{x}_i$ is the current average reward of arm $i$, $n_i$ is the total number of samples for arm $i$
4 end for
5 return the arm with the highest average reward
```

The rate of growth of cumulative regret of UCB1 is $O(\ln(n))$, where n is the number of samples, which was shown to be the optimal rate by (Lai and Robbins, 1985). Bubeck *et al.* (2010) show that a recommendation based on such a selection policy suffers a simple regret that decreases at best at a polynomial rate.

Later in this chapter, in Section 2.5 UCB1 is discussed in the context of MCTS. In MCTS, a version of UCB1 adapted to tree search named Upper Confidence Bounds for Trees (UCT) is widely used as the preferred selection policy.

2.4 Pure Exploration in Multi-Armed Bandits

Non-exploiting selection policies have been proposed to decrease simple regret at a high rate, and with low bounds. Given that UCB1 has an optimal rate of cumulative regret convergence, and the conflicting limits on the bounds on the regret types shown by Bubeck *et al.* (2010), policies that have a higher rate of exploration than UCB1 have better bounds on simple regret.

Consider a uniform selection policy that samples each arm of an MAB T/K times. Assuming that there are h best arms, $(K-h)T/K$ trials are spent sampling inferior arms, and hT/K on the best one(s).

Such a selection policy has simple regret $\mathbb{E}r_n = \mathbb{E}R_n/n$ (Bubeck *et al.*, 2010). In games, there are often only a few promising moves to be identified, and therefore when using uniform selection, most time is spent sampling suboptimal arms. Therefore, a more efficient policy is required to ensure that inferior arms are not selected as often as arms with a high utility over time. Two algorithms, discussed in this section, have been proposed to solve this problem.

Both algorithms have shown to outperform the other in different problem settings (Karnin *et al.*, 2013). Sequential Halving gives better results when there are multiple groups of suboptimal arms, or when the arm's rewards form an arithmetic or geometric series. Successive Rejects performs best when there is a single group of suboptimal arms, or when selecting over a small subset of arms. Both methods have proven theoretical guarantees with regard to the simple regret bounds, and a near-optimal, exponential rate of decrease on simple regret. Based on this it holds merit to consider these algorithms as candidate substitutes for UCT in MCTS. However, based on the analysis made by (Bubeck *et al.*, 2010) discussed in Section 2.2, any method that effectively minimizes simple regret has inferior bounds on its cumulative regret. Therefore, there may be some trade-off between using pure exploration methods and UCT.

2.4.1 Successive Rejects

Algorithm 2: Successive Rejects (Audibert *et al.*, 2010).

```
Input: total budget $T$, $K$ arms
Output: recommendation $J_T$
1  $S_1 \leftarrow \{1, \dots, K\}$, $n_0 \leftarrow 0$
2  $\overline{log}(K) \leftarrow \frac{1}{2} + \sum_{i=2}^{K} \frac{1}{i}$
3  foreach $k \in \{1, \dots, K-1\}$ do
4      $n_k = \left\lceil \frac{1}{\overline{log}(K)} \frac{T-K}{K+1-k} \right\rceil$
5  end foreach
6  for $k$=1 to $K-1$ do
7      Sample each arm $i \in S_k$ $n_k - n_{k-1}$ times
8      update the average reward of each arm based on the samples
9      $S_{k+1} \leftarrow$ the $|S_k| - 1$ empirically best arms from $S_k$
10 end for
11 return the single element of $S_K$
```

Successive Rejects (Audibert *et al.*, 2010) works by successively removing the single worst arm from the selection. The algorithm computes a budget n_k for each round. During a round, each arm is selected uniformly, and after each round, the empirically worst arm is removed from the selection and the next round starts. The current average values of the arms are retained between rounds. The lengths of the rounds are computed in such a manner that a specific lower bound on simple regret is guaranteed.

Successive Rejects is detailed in Algorithm 2. Because $n_0 = 0$, the algorithm starts with a relatively long initial round, followed by the shortest one, due to the subtraction on line 7. However, the length of the rounds increases when more arms are removed. An example run is depicted in Figure 2.1, where a total budget $T = 200$ is allocated to 5 arms, the number of samples per arm for each round is shown. For the first round the trials per arm are $\lceil \frac{60}{107} \frac{200-5}{5+1-1} \rceil - 0 = 22$ subsequently, for the second round $\lceil \frac{60}{107} \frac{200-5}{5+1-2} \rceil - 22 = 6$ and so on. The recommended arm is sampled a total of 57 times.

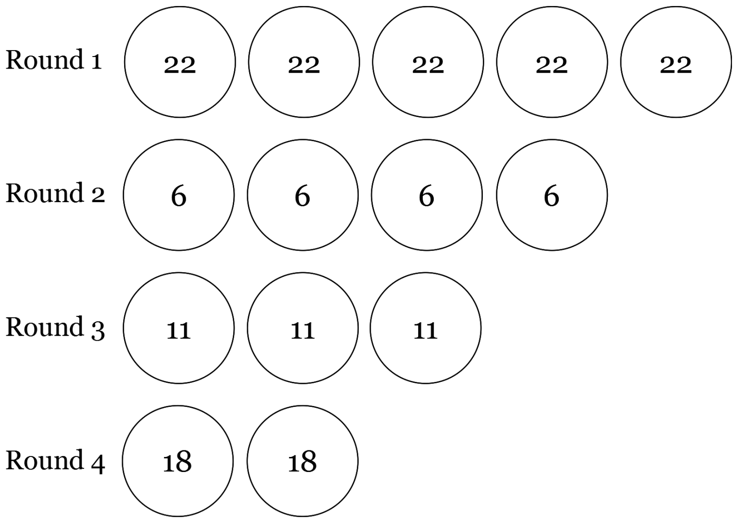

Figure 2.1: Successive Rejects example on 5 arms and a total budget of 200.

2.4.2 Sequential Halving

Sequential Halving (Karnin *et al.*, 2013) takes an approach similar to Successive Rejects. As with Successive Rejects, search time is divided into rounds, and during each round arms are sampled uniformly. However, instead of removing a single arm from selection after each round, half the arms are removed until a single one remains. The rounds in Sequential Halving are equally distributed such that constitute the same number of trials, but with a decreased subset of arms to select. Sequential Halving is detailed in Algorithm 3.

Algorithm 3: Sequential Halving (Karnin *et al.*, 2013).

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

An example of the budget allocation by Sequential Halving is depicted in Figure 2.2, where a total budget $T = 200$ is allocated to 5 arms, the allocated budget per arm for each round is shown. At time T, after $\lceil \log_2 5 \rceil = 3$ rounds, the single remaining arm is recommended. This arm is sampled 68 times, 11 times more than the recommended arm in the same example given for Successive Rejects.

Sequential Halving is a candidate for a recursive definition to be used in MCTS. The algorithm performs well in the cases described in the original article, and samples the best node more frequently than Successive Rejects. Moreover, Sequential Halving currently provides the best bounds on simple regret (Karnin *et al.*, 2013). An MCTS variant that implements Sequential Halving recursively named SHOT (Cazenave, 2014), is discussed in Chapter 3. Unlike Sequential Halving, Successive Rejects has not been used as a selection policy for tree search in current literature, this is open to future research.

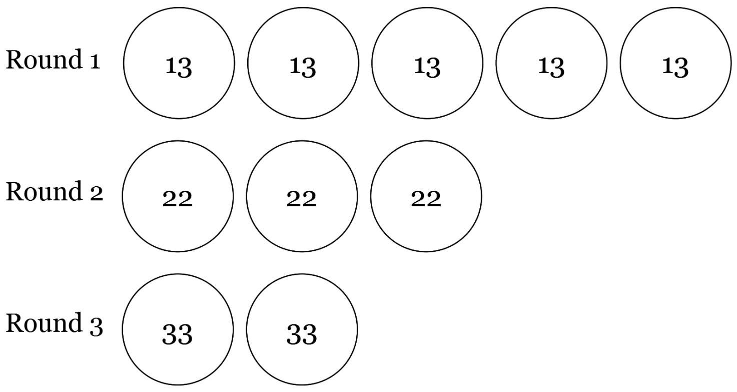

Figure 2.2: Sequential Halving example on 5 arms and a total budget of 200.

2.5 Relation to MCTS

2.5.1 Upper Confidence Bounds for Trees

During the MCTS 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) (Kocsis and Szepesvári, 2006) was derived from the UCB1 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 with the exploration of lesser visited ones. Consider a node p with children $I(p)$, then the policy determining which child i to select is defined as:

$$i^* = \operatorname*{arg\,max}_{i \in I(p)} \left\{ x_i + C\sqrt{\frac{\ln n_p}{n_i}} \right\}, \tag{2.3}$$

where x_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 an exploration constant to tune. UCT is generally applied when the visit count of a child node is above a threshold (Coulom, 2007). When a node's visit count is below this threshold, a child is selected at random.

Note that UCB1, and consequently UCT, incorporates 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. Because, by the *MinMax* principle, which states that an agent aims to maximize its minimum gain, on each ply we expect a player to play the best-reply to its parent.

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 average values.

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" (Ramanujan, Sabharwal, and Selman, 2010) are not revealed. More generally, when sampling a game-tree rewards are not necessarily drawn from a fixed distribution.

2.5.2 Regret in MCTS

Based on the analysis in the previous subsection, 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, as is shown in Section 2.2 there exist two contexts for the multi-armed bandit 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. Browne *et al.*, 2012), suggesting that the first context applies. However, the second context is a more natural fit when MCTS is used to play games, because the behaviour 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 can be 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 utility. Moreover, since all values are back-propagated, we risk under-evaluating ancestors based on sampling nodes that are known to be bad. This trade-off was also shown in (Tolpin and Shimony, 2012), 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 Best Recommendation with Uniform Exploration (BRUE) (Feldman and Domshlak, 2012).

The general performance of MCTS may be improved by applying a strategy that is focused on minimizing simple regret fast near the root, rather than cumulative regret. Because the recommendation made by MCTS is based on the values of the root's children, assuming that simple regret is a better measure of the quality of a given decision at time T, and that we can apply both the aforementioned contexts to search in a specific manner, the algorithms discussed in this chapter may improve the performance of MCTS when they are combined with UCT. In this manner, the benefits of lower bounds on simple regret for the recommendation at the root will originate from results back-propagated from selections made by UCT deeper in the tree, using both methods in their appropriate context.

In the next chapter, MCTS variants designed to (partially) minimize simple regret are discussed. Subsequently, in Chapter 4, a new search technique is proposed that uses both simple and cumulative regret minimizing policies at appropriate segments of the MCTS tree. Using this technique, we can determine whether whether the assumption holds that preferring simple over cumulative regret minimization near the root is both practical, and improves overall performance.

Chapter 3

Simple Regret in MCTS

Chapter contents: A discussion on MCTS variants using simple regret minimizing selection policies. This chapter serves as introduction and inspiration to the next in which the new search technique, which combines different selection policies, is detailed.

3.1 Introduction

Since the introduction of MCTS (Kocsis and Szepesvári, 2006) and its subsequent adoption by games researchers (cf. Browne *et al.*, 2012) UCT, or some variant thereof, has become the "default" selection policy. Since the introduction of simple regret (Bubeck *et al.*, 2010), more MCTS research has been performed using simple regret as the minimizing metric.

Feldman and Domshlak (2012) have proposed an algorithm named BRUE, developed to guarantee an exponential rate of reduction of simple regret. Rather than building a connected tree such as MCTS, BRUE generates non-connected nodes based on a switching function. The authors mention that the algorithm is a possible replacement for UCT. In Feldman and Domshlak (2013), BRUE is extended such that it may be used in more practical domains. Importantly, the proposed BRUE$_\mathcal{I}$ can be used without specifying a fixed horizon, rather it builds a tree connected to the root incrementally. Therefore, based on experimentation with BRUE and BRUE$_\mathcal{I}$ in the games discussed in this thesis, no improvement was shown over UCT. Based on these findings, the algorithm is not discussed in this chapter.

Rather than optimizing either simple or cumulative regret throughout the MCTS tree, Tolpin and Shimony (2012) propose a technique in which simple regret is minimized using either a modified version of UCB or a $\frac{1}{2}$-greedy policy only at the root, and UCT throughout the rest of the tree. Similar to BRUE, the authors developed this algorithm for solving *Markov decision processes*. SHOT (Cazenave, 2014) is an MCTS algorithm based on sequential-halving (Karnin *et al.*, 2013). This algorithm was developed to provide a speed-improvement over UCT, allowing more simulations per second to be performed and thereby increasing performance in game-play. SHOT spends less time in the tree back-propagating values from leaf to root because it assigns a potentially large budget of play-outs to each node. This means that accumulated values are back-propagated in the tree, instead of a single win or loss after each play-out.

Two MCTS variants are discussed in this chapter, as their approaches lead to key insights for the development of a hybrid MCTS. First, in Section 3.2, the so-called Simple Regret + Cumulative Regret (SR+CR) scheme proposed by (Tolpin and Shimony, 2012) is discussed. Next, in Section 3.3, the recently introduced SHOT (Cazenave, 2014) is detailed. Both algorithms form the foundation of Hybrid MCTS (H-MCTS), which is introduced in the next chapter.

3.2 MCTS Based on Simple Regret

Tolpin and Shimony (2012) give the same arguments presented in Chapter 2: that when MCTS is used in a the context of search in an MDP, *" it is usually only the final 'arm pull' (the actual move selection) that collects a reward, rather than all 'arm pulls' "* (Tolpin and Shimony, 2012). Moreover, the exploitation of high valued nodes is not preferable in all circumstances. Rather, more time should be spent exploring the alternatives. The hypothesis of the article is that since only the children of the root represent an action to be taken in the domain, a selection policy that minimizes simple regret at the root, and UCT throughout the rest of the tree, should have better performance than using only UCT.

Two selection policies with lower bounds than UCT on simple regret were constructed by the authors for the purpose of selecting nodes at the root.

1. $\frac{1}{2}$-greedy, a policy that selects a move at random half of the time, and the current empirical best the other half. This policy offers an exponentially decreasing simple regret.

2. $UCB_{\sqrt{\cdot}}$, which is similar to UCT, however the logarithm in the numerator in the upper confidence bound is replaced by a square root.

$$i^* = argmax_{i \in I(p)} \left\{ v_i + C\sqrt{\frac{\sqrt{n_p}}{n_i}} \right\} \tag{3.1}$$

   This selection policy has a super-polynomially decreasing simple regret (Tolpin and Shimony, 2012).

In both cases, the recommendation made by the algorithm is based on the node with the highest value at time T.

Next, a two-stage sampling scheme named SR+CR uses one of the above policies at the root, and UCT in the rest of the tree. In Algorithm 4, the scheme proposed by the authors is given in the context of game-play. Although originally, the algorithm was designed for maximizing rewards on each ply, it is presented in th negamax context such that it applies to games.

The selection policies were empirically validated to have lower simple regret than UCB in MABs. Moreover, the SR+CR scheme was shown to have lower simple regret in the MDP sailing domain. Particularly, the scheme was more advantageous when the number of nodes at the root are higher. Although the SR+CR scheme and two-stage MCTS were not developed for game-play, it holds merit to attempt to replicate the favourable results presented in games. Since, as the authors argue, only the

Algorithm 4: Two-stage Monte-Carlo Tree Search (Tolpin and Shimony, 2012).

```
Input: node $p$, current search depth
SRCR-MCTS(node $p$, $depth = 1$):
    if $isLeaf(p)$ then Expand($p$)
    if $depth = 1$ then
        select child $i$ using $\frac{1}{2}$-greedy or $UCB_{\sqrt{\cdot}}$
    else
        select child $i$ using UCT
    endif
    if $isLeaf(i)$ then
        $r \leftarrow$ PLAYOUT($i$)
        UPDATE node $i$ with $r$
    else
        $r \leftarrow$ -SRCR-MCTS($i$, $depth + 1$)
    endif
    UPDATE node $p$ with $r$
return $r$
```

options given at the root collect any true reward from the domain, and this is the case in both MDP domains and games. The algorithm presented was implemented for Amazons, Breakthrough, NoGo and Pentalath. However, using either of the two selection methods proposed declined performance when competing against UCT in preliminary experiments.

3.3 Sequential Halving Applied to Trees

A recent addition to simple regret techniques in MCTS is Sequential Halving applied to Trees (SHOT) (Cazenave, 2014). The algorithm is presented as a "faster" version of MCTS. Instead of using UCT and backing up values after each simulation from leaf to root, SHOT uses Sequential Halving throughout the tree. This essentially turns MCTS into an iterative deepening, depth first search. The main benefit the author points out is that SHOT spends much less time in the tree, updating and back-propagating values. Consequently, an optimized engine with fast play-outs can perform more simulated games per turn using SHOT than it could with UCT. In practice it means that for the same number of play-outs SHOT is approximately twice as fast as UCT. SHOT allocates a possibly large number of play-outs to the possible moves. This makes it quite easy to parallelize without loss of information and without changing the behaviour of the algorithm.

In the MAB context, Sequential Halving is run once, but in MCTS, nodes can be revisited, and therefore Sequential Halving is "restarted" upon revisiting a node. Define a *cycle* as one full iteration of Sequential Halving which starts on line 20, and a *round* as a sub-iteration over a given subset of S which starts on line 22, in Algorithm 5. Because any number of cycles may be started on a given node, care must be taken that budget is divided such that after any cycle the visit counts of all nodes are as if only one cycle was run. Therefore, when dividing the budget on line 21 the current budget spent is added to the allocated budget, essentially overestimating the total budget available. Next, on line 24 only the budget that exceeds the current child's visits is allocated. This is illustrated in Figure 3.1, where a node that was previously visited 64 times starts a new cycle with a budget of 128. The new budget per arm is: $\left\lfloor \frac{64+128}{4\times\lceil log_2 4\rceil} \right\rfloor = 24$. However, since the second and fourth child have previously been assigned a total budget of 24, only the first and third node are assigned $24 - 8 = 16$ budget.

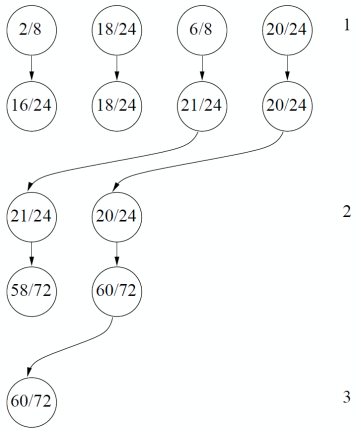

Figure 3.1: Division of 128 play-outs. The current node was previously visited 64 times (Cazenave, 2014).

SHOT was demonstrated to outperform UCT in NoGo, both on 9×9, and 19×19 boards, with win-rates between 75% and 100% (Cazenave, 2014). However, based on the results in the article and Chapter 5, these results were due to SHOT being significantly faster than UCT, when both algorithms are given the same budget of play-outs, different results are achieved.

Although in the original article SHOT is presented with the use of a transposition table instead of a tree, in Algorithm 5 a version of SHOT using a tree of game states is presented.

Algorithm 5: Sequential Halving applied to Trees (SHOT) (Cazenave, 2014).

```
Input: node $p$, allocated budget $budget$
Output: tuple containing the number of visits, $p1$ and $p2$ wins, budget used
SHOT(node $p$, $budget$):
    if $isLeaf(p)$ then $S \leftarrow$ EXPAND($p$)
    $t_p \leftarrow \langle 0, 0, 0, 0 \rangle$
    if $isTerminal(p)$ then
        UPDATE $t_p$, with $budget$ wins for the appropriate player and $budget$ visits
        return $t_p$
    endif
    if $budget = 1$ then
        $r \leftarrow$ PLAYOUT($p$)
        UPDATE $t_p$, with 1 budget used, 1 win for the appropriate player, and 1 visits
        return $t_p$
    endif
    if $|S| = 1$ then
        $n_0 \leftarrow$ the single element in $S$
        $t_p \leftarrow$ SHOT($n_0$, $budget$)
        UPDATE $p$ with $t_p$
        return $t_p$
    endif
    $s \leftarrow |S|$, $S_0 \leftarrow S$, $b_u, b, k \leftarrow 0$
    while $s > 1$ and $b_u < budget$ do
        $b \leftarrow b + \max\left(1, \left\lfloor \frac{p.budgetSpent + budget}{s \times \lceil log_2|S| \rceil} \right\rfloor\right)$
        for $i=1$ to $s$ do
            $n_i \leftarrow$ node $n$ at rank $i$ of $S_k$
            $b_i \leftarrow b - n_i.visits$
            if $b_i > 0$ then
                if $p$ is root and $i = 0$ and $s = 2$ then
                    // Spend any left-over budget on the empirically best node
                    $b_i \leftarrow \max(b_i, budget - b_u - (b - n_1.visits))$
                endif
                $b_i \leftarrow \min(b_i, budget - b_u)$
                $\langle v, w_1, w_2, b_{u,n_i} \rangle_i \leftarrow$ SHOT($n_i$, $b_i$)
                UPDATE $p, b_u$, and $t_p$ with $\langle v, w_1, w_2, b_{u,n_i} \rangle_i$
            endif
            break if $b_u \geqslant budget$
        end for
        $k \leftarrow k + 1$
        $S_k \leftarrow S_{k-1}$ sorted in descending order
        $s \leftarrow \lceil s/2 \rceil$
    end while
    UPDATE $p.budgetSpent$ with $b_u$
return $t_p$
```

SHOT recursively performs Sequential Halving until either a terminal state is encountered (line 4), or the available budget for the current node is 1 (line 8). Because multiple play-outs can be performed from a single recursive call, a tuple t_p is maintained containing: 1) the budget used b_u, 2) the number of wins per player w_1 and w_2, and 3) the number of visits v. v differs from b_u, because visiting a terminal node spends no budget, but performing a play-out does, and in both cases the visit count of the node should be increased. On line 30, p is updated with the results returned from the recursion, b_u is updated with the total budget used by play-outs, and t_p is updated with all returned values. On line 38 the total budget spent at the current node is incremented. This value is used to determine the budget of the next round of SHOT.

A disadvantage of SHOT is that it cannot be terminated any time, and requires a-priori knowledge of the available budget. The pure exploration policies discussed in Chapter 2 guarantee a low simple regret on the recommendation only after all available T simulations are performed. Therefore, they cannot be terminated, or asked for a reasonable recommendation before this limit is reached.

This leads to another concern when using any pure exploration policy recursively. As discussed in Chapter 2, UCT has the advantage of almost exclusively selecting the best node over time, giving the formal guarantee that suboptimal nodes are selected at most $O(\ln n)$ times. This ensures that over time, values accumulated at nodes approximate the value of the best-reply to the node's move. With a policy such as Sequential Halving this guarantee is related to the available budget and the branching factor. In fact, the two empirically best nodes are sampled equally, however different their reward may be. Moreover, for a node with a set of children S, the worst $\frac{|S|}{2}$ of these are sampled at least at total of $\lfloor \frac{T}{\lceil log_2|S| \rceil} \rfloor$ times. In many games there are only a handful of good moves given a position, and to evaluate a node the value of its move's best-reply must be found. By using Sequential Halving throughout the tree, values back-propagated do not converge in the same manner as UCT, as such, values of internal nodes are constituted of more overall averages over their children.

Chapter 4

A Hybrid MCTS

Chapter contents: The main contribution of this thesis, a hybrid version of MCTS named H-MCTS, which combines simple and cumulative regret minimizing selection policies into a single search technique.

4.1 Introduction

As a selection policy for MCTS, UCT minimizes cumulative regret over time throughout the tree. At internal nodes, minimizing cumulative regret ensures a node with a high expected reward is visited more often. UCT converges asymptotically to a greedy selection policy, where at each iteration only the expected best node is selected. In effect, the average rewards at parent nodes converge to the maximum over their children, *i.e.,* UCT converges to maximum back-propagation on a stable distribution (Kocsis and Szepesvári, 2006). Secondly, UCT ensures that search time is divided efficiently, limiting the number of suboptimal nodes selected by $O(\ln(n))$ where n is the number of trials (Auer *et al.*, 2002; Kocsis and Szepesvári, 2006). However, such a selection policy offers a simple regret that decreases at best polynomially fast (Bubeck *et al.*, 2010). These two properties are not only practical when long deliberation times are feasible, rather, it is possible to terminate UCT at any time to obtain a reasonable approximation of the utility of a decision. During search this means that at each ply and at any time, MCTS can be asked for a best response to a parent's move.

In contrast, pure exploration algorithms, designed to minimize simple regret in multi-armed bandits such as Successive Rejects (Audibert *et al.*, 2010) and Sequential Halving (Karnin *et al.*, 2013), discussed in Chapter 2, explore all options without specifically exploiting the best one. This results in a lower bound on simple regret only after all simulations have been performed. As such, to be used effectively, these algorithms can not be terminated at *any time*, and have a lower formal guarantee on the number of suboptimal selections made. Moreover, Bubeck *et al.* (2010) have shown that at moderate deliberation times, a modified UCB selection policy, which includes an exploration constant not unlike UCT, resulted in a better bound on simple regret than algorithms designed to minimize it. Relating these findings to MCTS, and assuming that given sufficient time, minimizing simple regret gives a lower simple regret than UCT, we may choose to use a simple regret minimizing algorithm whenever sufficient trials can be performed, and switch to UCT when the computational budget is smaller.

Inspired by the analysis and discussions in Chapters 2 and 3, a new MCTS variant is proposed named Hybrid MCTS (H-MCTS). H-MCTS initializes with an exploratory depth-limited search, applying a specific simple regret minimizing policy. After reaching a node where computational budget per node is smaller than a set limit, the algorithm switches to the UCT selection policy.

Section 4.2 gives an introduction and analysis of H-MCTS. Next, Section 4.3 details the implementation of MCTS-Solver (Winands *et al.*, 2008), designed to solve positions within the tree, for H-MCTS. Finally, in Section 4.4, an enhancement for SHOT and H-MCTS is introduced that reduces the number of unpromising nodes selected based on whether their upper confidence bounds overlap with their current empirically best sibling's lower confidence bound.

4.2 Hybrid MCTS

Recall that in the MAB context, in which simple regret minimization is appropriate, only the final recommendation made by an algorithm has an effect on the agent's reward. In game play this holds for the nodes of the search tree at the first ply. Only after performing all simulations is a recommendation 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 reinforces the confidence of the reward of promising nodes, but also ensures that their reward can be falsified based on results deeper in the tree.

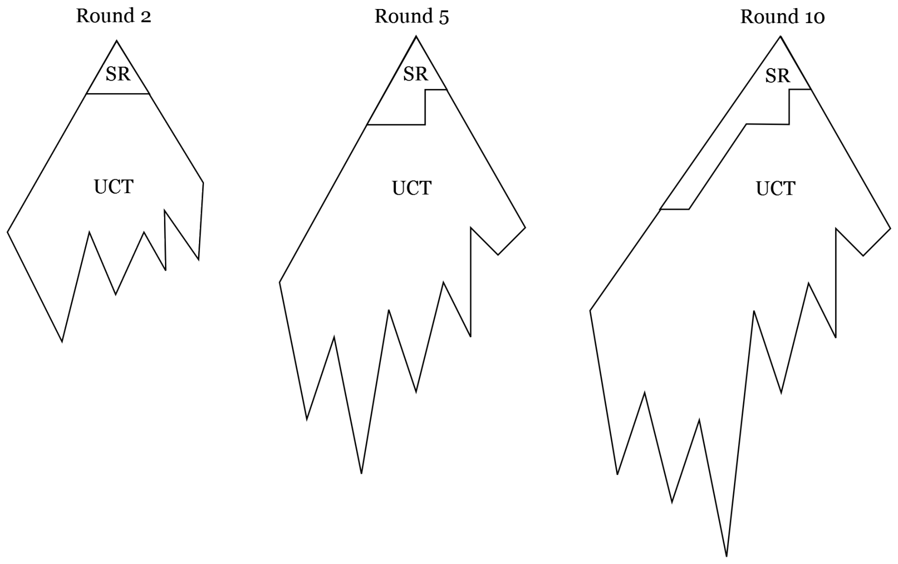

Figure 4.1: Example progression of H-MCTS. In the top part of the tree (SR), simple regret is minimized, in the lower part UCT minimizes cumulative regret. The rounds represent the Sequential Halving round at the root.

Treating a game tree as a recursive multi-armed bandit 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. Further down, 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 2.5 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 the rest of the tree, as suggested by Tolpin and Shimony (2012). 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 low simple regret on that particular move because this could intrinsically lead to an improved evaluation of the parent. Using the SHOT technique, we can apply Sequential Halving recursively, essentially combining recursive simple and cumulative minimization in a single search tree, as depicted in Figure 4.1.

Using a selection policy based on both SHOT and UCT, H-MCTS combines simple and cumulative regret minimization in a tunable algorithm. Bubeck *et al.* (2010) have shown that given a low sampling budget, UCB empirically realizes lower simple regret. Therefore, 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 4.1, initially the simple regret tree is shallow 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 because the root's children are sorted in descending order, the left part of the simple regret and UCT trees are always the deepest, since those UCT trees are selected the most, see Figure 4.2.

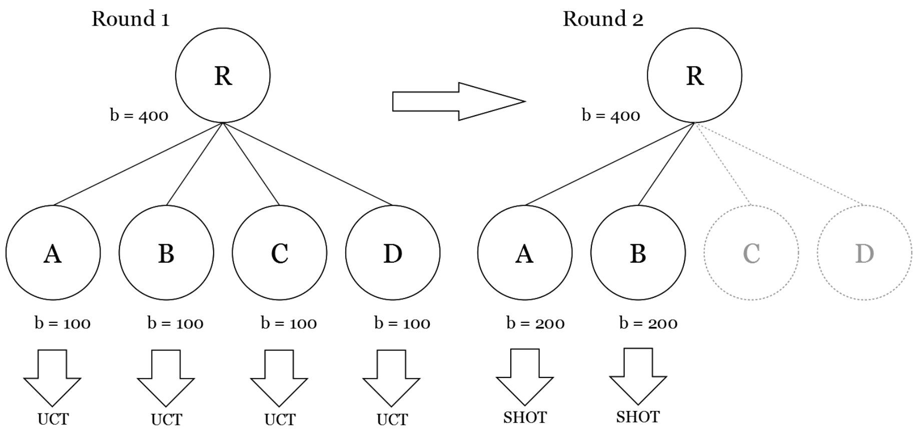

Figure 4.2: Example rounds of H-MCTS with a budget limit $B = 150$. During the first round SHOT is used only at the root R. In round 2, the budget per node is higher, and SHOT is used recursively at nodes A and B.

H-MCTS is outlined in Algorithm 6. Similar to UCT and SHOT, on line 4 terminal conditions are handled, followed by the main feature of the algorithm on line 8 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 $b \geqslant B$ 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 large, at the root simple regret minimization is preferred over cumulative regret minimization. From line 19 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 26, 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 8.

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.

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 were 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 32, UCT's any-time property ensures nodes have a representative value.

When all T simulations have been performed, H-MCTS recommends the single remaining node $n_0 \in S$ at the root. Note that this is not necessarily the one with the highest value, it is possible that the best

Algorithm 6: 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 $t_p$
    endif
    $b \leftarrow \max\left(1, \left\lfloor \frac{p.budgetSpent+budget}{s \times \lceil log_2|S| \rceil} \right\rfloor\right)$
    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 $\langle v, w_1, w_2 \rangle_i$
        end for
        return $t_p$
    endif
    $b_u, k \leftarrow 0$
    $S_0 \leftarrow S$
    $s \leftarrow |S|$
    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 $\langle v, w_1, w_2 \rangle_i$
            endif
            break if $b_u \geqslant budget$
        end for
        $k \leftarrow k + 1$
        $S_k \leftarrow S_{k-1}$, 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 $b_u \geqslant budget$ or $s < 2$
    UPDATE $p.budgetSpent$ with $b_u$
return $t_p$
```

node's value has decreased after several rounds. However, we cannot assume that the values of the previously removed nodes do not also decrease given more samples. Therefore, because most samples were spent on n_0, we are the most confident in its value. Other nodes with less visits may have higher values, but the confidence in these values is much lower. A possible solution to this problem presents itself in the manner that S_k is constructed, on line 32. An alternative would be to setup each round's subset of nodes as follows: $S_k \leftarrow S$ sorted in descending order. This way, nodes that were previously removed from selection can 'return', and replace nodes whose value decreased.

To a lesser extent, H-MCTS also shares the speed benefit of SHOT. However, because a large part of the search tree is composed of the UCT tree, based on the budget limit B H-MCTS still spends more time in the tree than SHOT overall. However, given a lower budget limit B, H-MCTS can be made to run faster by increasing the ratio of the simple regret tree related to the UCT tree.

4.2.1 Budgeting

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. However, based on the simple regret theory in MABs, pure exploration methods only provide empirically better simple regret than UCB, given a sufficiently large budget. Given a small budget, UCB with a properly tuned constant should be preferred (Bubeck *et al.*, 2010). Directly applying this result to MCTS means that whenever the available budget is low, UCT with a properly tuned constant should be preferred to reduce simple regret.

The budget limit B is compared to SHOT's budget allocation:

$$b = \left\lfloor \frac{p.budgetSpent + budget}{s \times \lceil log_2|S| \rceil} \right\rfloor, \tag{4.1}$$

which includes the budget previously spent at the node.

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.

4.2.2 Selection

Selection is performed by uniformly distributing the assigned budget according to the method used in SHOT. However, in H-MCTS any left-over budget is spent in the final round of Sequential Halving, as opposed to spending all residual budget at the root's best child. This ensures that internal nodes' best-replies are selected more often, improving the confidence in their valuation, and of their parent's values.

Note that although Sequential Halving is presented as the simple regret algorithm in H-MCTS, it is certainly possible to replace it with a different selection policy, such as Successive Rejects, or another form of sequential reduction of nodes.

4.2.3 Removal

Since it is possible to visit a node more than once with a new budget, children are merely "excluded", instead of removed, from selection. The current list of selection candidates S_k consists of all children sorted from 0 to s. Whenever a new budget is assigned to a node, all the previously excluded children are returned so S_0, and a new Sequential Halving cycle can begin.

4.2.4 Back-propagation

Typically in MCTS, results of all play-outs are back-propagated and accumulated at every ancestor of the expanded leaf, *i.e., average back-propagation.* To determine a node's value, its descendants should be selected such that the (expected) best ones are sampled most often. In the classic MiniMax case, a node's value is determined by the value of its best-reply. In MCTS, this idea is not explicit: over time a node's value converges to the value of its best-reply. However, when a node is insufficiently sampled, the general approach assigns it the current average value of its children.

Whereas UCT selects only promising nodes when it can, Sequential Halving samples all nodes in the current subset, each round. This means that the statistics collected at the parent of a given node do not reflect the best-reply to its move. Moreover, unlike Sequential Halving, UCT can be asked for a recommendation at any time, this means that a node explored by UCT has a representative value at any time. With Sequential Halving, intermediate averages of a node explored by the selection policy may be under-evaluated. Recall that the best and second best children are sampled equally often, in case there exists only a single best-reply to a parent's move, this means that its value never reflects that of its best-reply. This is depicted in an example in Figure 4.3, where the parent is under-evaluated as it is an expected win for the player to move, whereas the parent's value represents a predicted loss. When this node is reselected with new budget to allocate, this situation does not improve, because Sequential Halving samples all nodes, not just the best one.

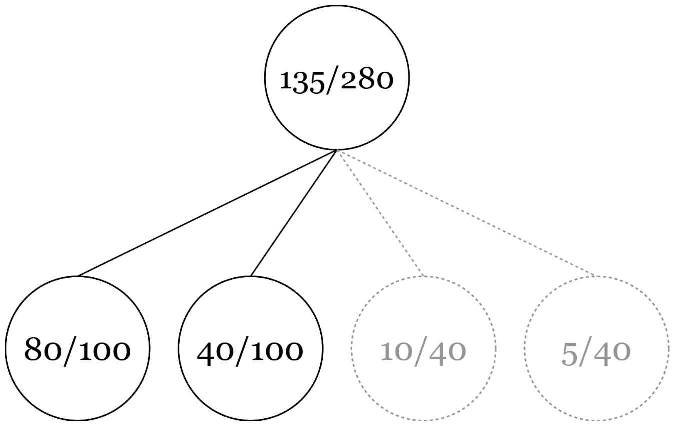

Figure 4.3: Average back-propagation and Sequential Halving example. The parent's value 0.48, whereas the value of the best-reply is 0.8.

As an alternative to average back-propagation in the simple regret tree, *maximum back-propagation* sets the value of each node to the maximum over their descendants. However, because nodes explored by Sequential Halving can only provide a recommendation at time T, the best node's intermediate values may be incorrect. Moreover, considering that the number of visits per ply declines proportional to the branching factor, deeper nodes that are sampled infrequently may only appear to have a good value due to insufficient sampling. Back-propagating such a node's value overrules many samples performed in the rest of the tree, disregarding many of the samples taken in potentially large sub-trees with higher confidence in their value.

The problem of back-propagation remains open. According to the theory, full exploration selection policies cannot give a recommendation at any time. However, in a tree, nodes' values should have a representative value at any time for UCT and Sequential Halving to be able to select promising nodes or sort nodes by value, respectively. In Section 4.4, an enhancement to Sequential Halving is discussed, which potentially reduces the number of suboptimal nodes sampled. As such, we can ensure that Sequential Halving converges to a best-reply over time.

4.3 Hybrid MCTS Solver

In the form presented in Algorithm 6, 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.* (2008) in the UCT tree, the technique requires adaptation to Sequential Halving to be able to solve nodes in the simple regret tree. The main issue here is that when a solved child is encountered, the current round is interrupted to determine whether the node itself is solved as well. This leads to three different possible cases, in which this interruption can be encountered, and properly handled. Note that it is assumed that values are according to negamax, such that at each node, children's rewards are stored with respect to the player to move.

1. A solved win is encountered, in this case, we immediately want to remember this node and back-propagate to the parent. Any residual budget remains unspent in the current round.
2. A solved loss is encountered, but not all siblings lead to a loss. According to Winands *et al.* (2008) it is appropriate in this case to count the visit as a loss. However, since there still exist nodes that do not lead to a loss, we have to ensure the solved node is not reselected. If it is reselected, a potentially high budget may be spent on the node, which results in an underestimation of the parent.
3. A solved loss is encountered, and all siblings lead to a loss. In this case the parent is a win and we should back-propagate immediately. Any residual budget remains unspent in the current round.

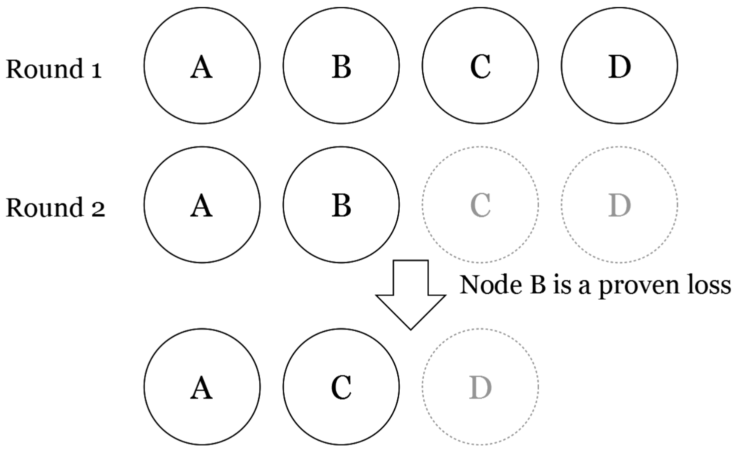

Figure 4.4: H-MCTS Solver removing a proven loss from selection. In the second round, node B is determined to be a proven loss, node C is returned to selection to fill the opened position.

When running Sequential Halving, the first and last cases can be handled similar to the MCTS-Solver. The second case potentially leaves the assigned budget partially unspent. Because some children have been identified as a loss, while other unsolved options still remain, the proven losses should be removed from selection. Moreover, any budget left unspent due to this should be "brought over" to the next round, which starts with a reduced set of nodes. When a proven loss is discarded from selection, all children following this node move one position left, possibly returning an unsolved node to the current selection, as depicted in Figure 4.4. If there is no unsolved node to return, the set of remaining arms is reduced after the round if it is smaller than $\lceil s/2 \rceil$, which means that more than half the nodes currently in S_k were solved.

Algorithm 7: Hybrid Monte-Carlo Tree Search Solver (H-MCTS Solver).

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

H-MCTS Solver is outlined in Algorithm 7. The general procedure when a solved node is encountered is straightforward in the case of a solved win, and when all children are solved losses. The *budgetSpent* is updated and the function immediately returns. The tuple t_p now also contains a 'solved' field, which holds the index of the player for which the node is solved. In this case the UPDATE function updates the node accordingly by setting the *isSolved* flag. When an isolated proven loss is encountered, *i.e.,* not all siblings are losses, then we should not sample that node again, since it would under-evaluate the parent. This possibly leaves a residual budget b_r declared on line 14, which is maintained per round. When an isolated proven loss is found, the residual budget is updated on line 31 such that it can be spent in the next round with a reduced set of nodes, on line 43. Moreover, since the isolated losses are removed from S on line 35, it is possible that $s > |S|$, which is restored on line 39 after halving s. This ensures that previously removed nodes that were not proven losses can be returned to selection in case more than half of the nodes were solved in this round. On line 37 the set of children from the previous round S_{k-1} is sorted, as such $|S_k| = |S|$ for each round. Therefore, when a solved loss is removed from selection, another child can take its place. Note also that on line 2, a second set I contains all children. Because it is possible to remove all nodes from S in case of a proven loss, I is never altered and can be used to give a recommendation at the root.

4.4 Upper and Lower Bounds for H-MCTS

A disadvantage of Sequential Halving applied to game trees is that nodes that can never improve on the current best reply are sampled relatively frequently. Although this is theoretically sound in providing lower guarantees on simple regret, practically it means that nodes are often sampled when it is clearly futile to do so.

For example, consider a set of 5 nodes S_1, in which there exists one node with a true mean μ^* of 0.7, all other nodes have means of -0.5. Only a few samples are required to find out which node is the best reply. However, as shown in the example in Figure 2.2, the second-best node are sampled as often as the best one, and in effect the parent's value is underestimated.

To counter this problem we can compare a child's upper confidence bound to its best sibling's lower confidence bound. In this optimistic scheme it is assumed that a node can still improve its value such that it becomes better than the current best reply, if their respective upper and lower confidence bounds overlap. Recall from Chapter 2, that the upper confidence bound represents the highest value a node can achieve with overwhelming probability. This approach can be inverted to acquire the lowest possible value a node is likely to achieve,

$$v_i - C \times \sqrt{\frac{\ln n_p}{n_i}}. \tag{4.2}$$

Because a sorted list of children is maintained in the simple regret tree, the best child's bound can be compared to it siblings' bounds at any time. Therefore, when selecting a node to be investigated, only if the current node's upper confidence bound overlaps with the currently estimated best-reply's lower confidence bound, should we assume that it is still possible for this node to improve upon the current best node's score. As such we can skip any node with a non-overlapping bound with the best node. This potentially saves a substantial budget which would otherwise have been spent sampling suboptimal nodes. An example of this enhancement is depicted in Figure 4.5. In this example, the current best node's lower bound overlaps only with the upper bounds of nodes B and C.

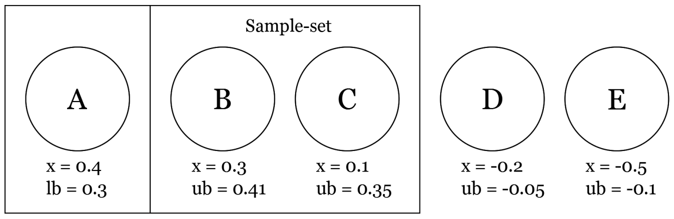

Figure 4.5: Example upper/lower bound Sequential Halving. Node A is the current empirical best, with a lower confidence bound of 0.1. Only nodes B and C have upper confidence bounds that overlap with this lower bound. D and E are not sampled this round.

Algorithm 8: Hybrid Monte-Carlo Tree Search Solver with Overlapping Bounds.

```
Input: node $p$, allocated budget $budget$
Output: $t_p$: number of play-outs, $p1$ and $p2$ wins, solved player
H-MCTS-SOLVER(node $p$, $budget$):
    if $isLeaf(p)$ then $I, S \leftarrow$ EXPAND($p$)
    $t_p \leftarrow \langle 0,0,0,0 \rangle$
    $b \leftarrow \max\left(1, \left\lfloor \frac{p.budgetSpent+budget}{s \times \lceil log_2|S| \rceil} \right\rfloor\right)$
    if $p$ is not root and $b < B$ then
        for $i=0$ to $budget$ do
            $\langle v, w_1, w_2, s \rangle_i \leftarrow$ UCT-SOLVER($p$)
            UPDATE $p, t_p$ with $\langle v, w_1, w_2, s \rangle_i$
        end for
        return $t_p$
    endif
    $b_u, k \leftarrow 0$, $S_0 \leftarrow S$, $s \leftarrow |S_0|$
    repeat
        $b_r \leftarrow 0$
        for $i=1$ to $\min(s, |S_k|)$ do
            $n_i \leftarrow$ node $n$ at rank $i$ of $S_k$
            if $n_i$ is not solved then
                if $b > n_i.visits$ then
                    $b_i \leftarrow b - n_i.visits$
                    if $i = 0$ and $s = 2$ then $b_i \leftarrow \max(b_i, budget - b_u - (b - n_1.visits))$
                    $b_i \leftarrow \min(b_i, budget - b_u)$
                    if $i > 0$ and $n_i.value + C \times \sqrt{\frac{\ln p.visits}{n_i.visits}} < l_b$ then
                        $b_r \leftarrow b_r + b_i$
                        continue
                    endif
                    $\langle v, w_1, w_2, s \rangle_i \leftarrow$ H-MCTS-SOLVER($n_i$, $b_i$)
                    UPDATE $p, b_u$, and $t_p$ with $\langle v, w_1, w_2, s \rangle_i$
                    if $i = 0$ then $l_b \leftarrow n_i.value - C \times \sqrt{\frac{\ln p.visits}{n_i.visits}}$
                endif
            endif
            if $n_i$ is a win or ($n_i$ is a loss and $\forall n \in S : (n$ is a loss$)$) then
                set the solved player in $t_p$
                UPDATE $p.budgetSpent$ with $b_u$
                return $t_p$
            else if ($n_i$ is a loss and $\exists n \in S : (n$ is not a loss$)$) then
                $b_r \leftarrow b_i - v$
            endif
            break if $b_u \geqslant budget$
        end for
        remove all solved nodes from $S$ and $S_k$
        $k \leftarrow k + 1$
        $S_k \leftarrow S_{k-1}$, with the first $\max(2, s)$ elements sorted in descending order
        $s \leftarrow \lceil s/2 \rceil$
        $s \leftarrow \min(s, |S|)$
        if $s = 1$ then $b \leftarrow b + (budget - b_u)$
        else
            $b \leftarrow b + \max\left(1, \left\lfloor \frac{p.budgetSpent+budget}{s \times \lceil log_2|S| \rceil} \right\rfloor\right)$
            $b \leftarrow b + \lceil b_r/s \rceil$
        endif
    until $b_u \geqslant budget$ or $s < 2$
    UPDATE $p.budgetSpent$ with $b_u$
return $t_p$
```

This enhancement also ensures that over time Sequential Halving samples greedily. Given infinite samples, the lower and upper confidence bounds of two nodes should only ever overlap if they have the same value. Since, in this case only the nodes with the highest averages are sampled, Sequential Halving with the Upper/Lower Bounds enhancement tends to sample only the best node(s) over time.

The Overlapping Bounds enhancement for H-MCTS is detailed in Algorithm 8, in which it is combined with the solver discussed in the previous subsection. On line 28 the lower confidence bound of the current best node is determined. Note that this is performed in each round, right after the best node of the round is sampled. This ensures that we have the best estimate for the current lower bound. All other children's upper confidence bounds are then compared to the current best lower bound on line 22. If the bounds do not overlap, the node is skipped, line 24. Similar to the solver, the unspent budget is reserved in b_r, it is reallocated on line 48 to be spent in the next round.

Chapter 5

Experiments and Results

Chapter contents: H-MCTS is thoroughly evaluated in six distinct two-player games: Amazons, AtariGo, Ataxx, Breakthrough, NoGo, and Pentalath.

5.1 Experimental Setup

In this chapter the results of the experiments performed on six two-player games are presented. 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.[1] Whereas the former is a full implementation of SHOT, H-MCTS, H-MCTS Solver, and the Upper/Lower Bounds enhancement, the latter consists of an implementation of SHOT and H-MCTS.

A brief description of each game is given below:

- *Amazons* is played on a 10×10 chessboard. Players each have four Amazons that move as queens in chess. Moves consist of two parts, movement, and shooting an arrow to block 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. The 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 thesis, 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. The 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. The 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 set of pieces.

All games use a uniform random selection policy during the play-outs, unless otherwise stated. All algorithms use transposition tables to store the values of game-states, these tables are cleared between moves, such that no information is retained. The C constant, used by UCT (Equation 2.3) was optimized for each game and was not re-optimized for H-MCTS, unless mentioned otherwise, both UCT and H-MCTS use the same C constant. Table 5.1 lists the C constant used by UCT and H-MCTS in the experiments in this section.

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.

[1]Experiments in Ataxx and AtariGo were performed in collaboration with Prof. Dr. Tristan Cazenave.

Game	C Constant
Amazons	0.4
AtariGo 9×9	0.125
Ataxx 7×7	0.5
Breakthrough 8×8	0.8
NoGo 9×9	0.6
Pentalath	0.8

Table 5.1: C Constants used by H-MCTS and UCT.

5.2 Results

For the tables and graphs in this section, 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.

This section is structured as follows: first, in Subsection 5.2.1 the performance of SHOT is compared to UCT. Followed by the results of games played by H-MCST against UCT in Subsection 5.2.2. The results presented in these subsections form the baseline for the experiments in the following ones. First, to determine the influence of UCT in H-MCTS, Subsection 5.2.3 details the results of games played by H-MCTS against SHOT. Followed by a detailed investigation into the performance of the H-MCTS Solver in Subsection 5.2.4. Finally, the performance of the Upper/Lower Bounds enhancement for H-MCTS is investigated in Subsection 5.2.5. Finally, Subsection 5.2.6 explores the influence of the C constant in H-MCTS.

5.2.1 SHOT and UCT

SHOT is the algorithm used in H-MCTS' simple regret tree, thus SHOT can be considered an instance of H-MCTS when $B = 0$. In this subsection SHOT is compared to UCT to determine a baseline for the H-MCTS experiments.

Game	**10,000 play-outs**	**25,000 play-outs**
Amazons	**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 5.2: SHOT vs. UCT, random play-outs
Win percentages with respect to SHOT. 1,000 games

SHOT played 1,000 matches against UCT per game, for each budget setting, results are shown in Table 5.2. SHOT performs best in the games with the highest branching factors, Amazons, AtariGo and to a lesser extent in NoGo. The NoGo results differ from those presented in (Cazenave, 2014) because this experiment is run using a fixed allocation of play-outs for both algorithms, unlike the experiments performed by Cazenave where SHOT receives a fixed budget, and UCT is allocated SHOT measured running time. The results reinforce 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 significantly against UCT. However, given SHOT's speed improvement over UCT, it is possible that the technique performs better in a time-based experiment.

5.2.2 H-MCTS and UCT

In this subsection, experiments comparing H-MCTS to UCT are detailed. Considering that H-MCTS uses UCT when a small budget is available, the benefit of minimizing simple regret near the root is determined by performing experiments where H-MCTS plays against UCT.

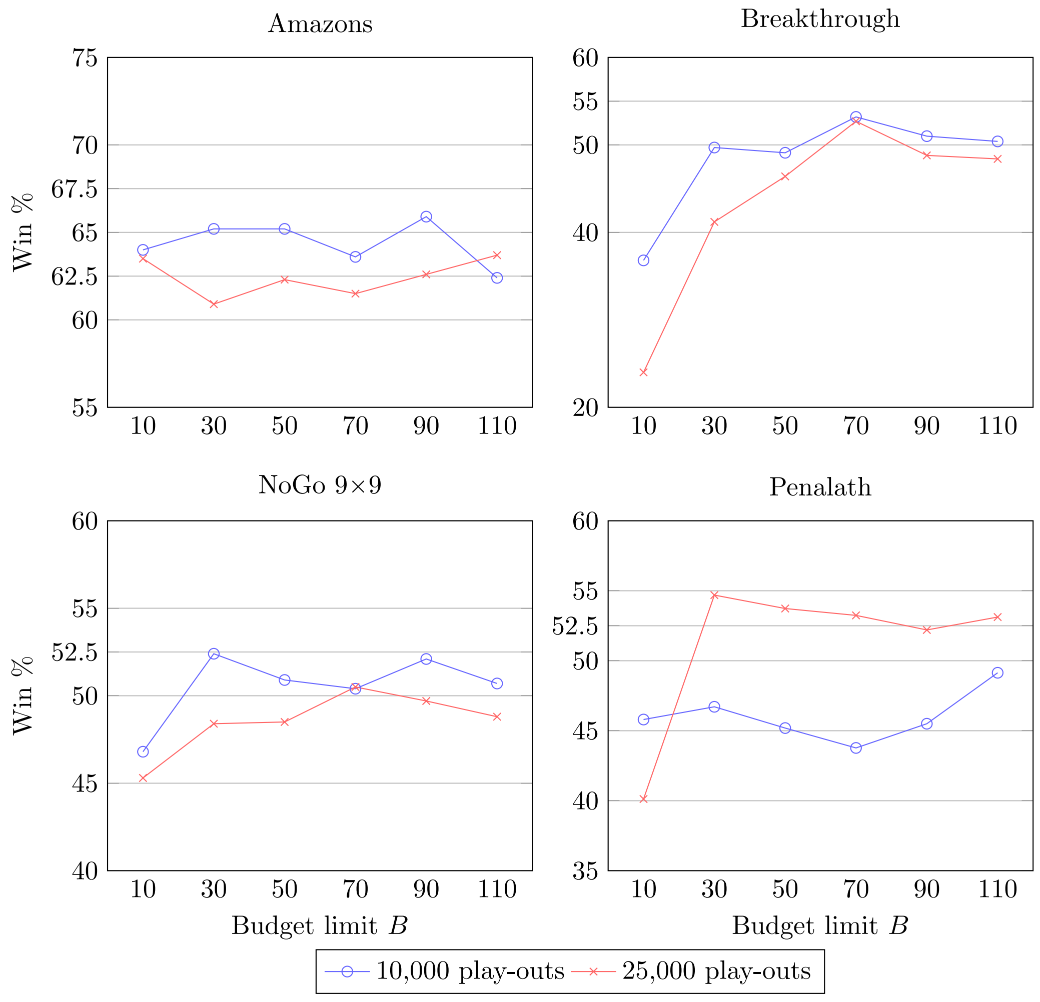

Figure 5.1: H-MCTS vs. UCT, random play-outs.
Win percentages with respect to H-MCTS. 1,000 games per data point.

Figure 5.1 gives an overview of the results for different budget limits B in H-MCTS. For each game 1,000 matches were played with a budget of 10,000 and 25,000 play-outs against UCT.

For Amazons, different settings of B do not significantly influence the performance of H-MCTS, H-MCTS performs consistently better than UCT in this domain. This is mainly due to the large branching factor of approximately $1,200$ moves at the start of the match. In this case, UCT spreads its budget out, allocating it almost randomly because it is unable to procure sufficient visits per move to decrease their confidence bounds. Unlike UCT though, H-MCTS initially performs several rounds with a large set of options, but reduces the number of nodes under consideration quickly, allowing it to focus on a subset of moves found to be promising.

Mainly in Breakthrough and Pentalath does the value of B have an impact on performance. In Pentalath the graphs show evidence that an increase in performance may be obtained given a larger budget of play-outs. For NoGo, the trend is similar to Amazons', the value of B has no large influence on performance. The graphs show that the best budget limit per game does not change significantly when the algorithm is given a higher deliberation time.

The best results for the values of B from Figure 5.1 were used to run an additional 1,000 matches per

Game	B	**10,000 play-outs**	**25,000 play-outs**
Amazons	90	**64.3** ± 2.1	**64.3** ± 2.1
Breakthrough 8×8	70	51.8 ± 2.2	47.7 ± 2.2
NoGo 9×9	30	49.9 ± 2.2	48.3 ± 2.2
Pentalath	30	46.6 ± 2.2	51.7 ± 2.2

Table 5.3: H-MCTS vs. UCT, random play-outs.
Win percentages with respect to H-MCTS. 2,000 games

Game	B	**10,000 play-outs**	**25,000 play-outs**
AtariGo 9×9	30	**57.9** ± 3.1	**69.1** ± 2.9
Ataxx 7×7	30	52.5 ± 3.1	**65.2** ± 3.0

Table 5.4: H-MCTS vs. UCT, random play-outs.
Win percentages with respect to H-MCTS. 1,000 games

game, of which the results are presented in Table 5.3. The best B for either 10,000 or 25,000 play-outs is selected based on the highest win-rate achieved. Results for 1,000 games in AtariGo and Ataxx are shown in Table 5.4. A significant performance increase for H-MCTS in both games is revealed in these games.

H-MCTS performs best in Amazons, Ataxx, and AtariGo. In Pentalath, given a higher total budget there is evidence that performance increases. To validate this conjecture in Pentalath, an additional 1,000 games were played with a budget of 50,000 play-outs. This resulted in a win-rate for H-MCTS of **55.0**%±3.1, a significant increase in performance over both UCT, and the 25,000 play-out case.

In Breakthrough, performance drops significantly given a higher overall budget. This may be due to the fact that the game has narrow winning-lines, and a more exploiting algorithm works better by identifying promising moves and exploiting them rapidly. Moreover, in this experiment both techniques use a random play-out policy. This means that because the results returned by these play-outs are less trustworthy, it is likely that Sequential Halving excludes a good move from selection too soon. This has a potentially large impact on performance, because moves that have been removed from selection cannot be re-selected at the root, and will therefore never be recommended.

Note that it is possible that H-MCTS requires separate tuning of its UCT C constant in games. This optimization was performed for Ataxx and AtariGo, which resulted in winning rates for H-MCTS of **61.7**%±3.0, and **66.6**%±3.0 in AtariGo with a C constant of 0.25, and **57.6**%±3.1, and **58.1**%±3.1 in Ataxx with a C constant of 0.125, for 10,000 and 25,000 play-outs, respectively.

5.2.3 H-MCTS and SHOT

To determine the effect of UCT in H-MCTS, the results of matches played against SHOT are shown in Table 5.5. 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 proper utility estimates in the simple regret tree. Values back-propagated by UCT may be more effective than those back-propagated by using Sequential Halving throughout the tree. Because H-MCTS also performs better than SHOT against UCT in these games, H-MCTS likely performs better in general than SHOT given a fixed play-out budget.

Game	B	**10,000 play-outs**	**25,000 play-outs**
Amazons	50	51.2 ± 3.1	**55.4** ± 3.1
AtariGo 9×9	30	**53.2** ± 3.1	**61.2** ± 3.0
Ataxx 7×7	30	**55.2** ± 3.1	**65.3** ± 3.0
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 5.5: H-MCTS vs. SHOT, random play-outs
Win percentages with respect to H-MCTS. 1,000 games

5.2.4 H-MCTS Solver

In this subsection, the influence of the solver enhancement in H-MCTS and SHOT is investigated. Experiments are performed to show the performance of the solver per technique individually, and for H-MCTS against UCT.

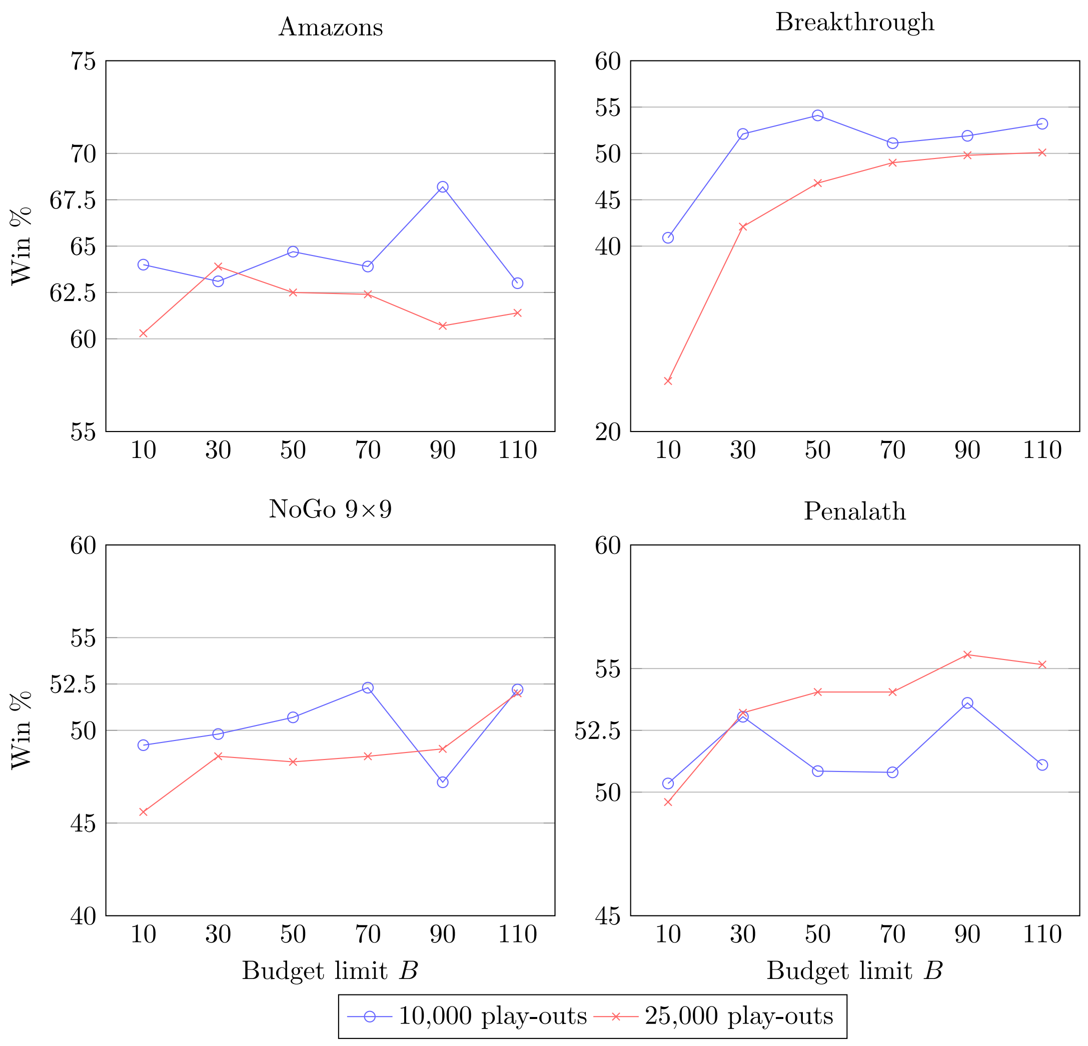

Figure 5.2: H-MCTS Solver vs. MCTS-Solver, random play-outs.
Win percentages with respect to H-MCTS. 1,000 games per data point.

In Figure 5.2 results are shown for different B values for H-MCTS Solver. All games were played against the MCTS-Solver. As in Subsection 5.2.2, the largest increases in performance are shown in Amazons and Pentalath. Breakthrough and NoGo both show positive results when the total budget is small, the benefit of H-MCTS decreases when the allocated budget increases. This is also visible in the best values for B, which are high for both games, implying that UCT is the preferred selection policy.

Game		**10,000 play-outs**	**25,000 play-outs**
		MCTS (UCT) Solver	
Amazons		50.2 ± 3.1	50.6 ± 3.1
Breakthrough 8×8		**53.7** ± 3.1	49.0 ± 3.1
NoGo 9×9		49.9 ± 2.2	49.9 ± 3.1
Pentalath		**53.7** ± 3.1	48.6 ± 3.1
	B	**H-MCTS Solver**	
Amazons	90	50.1 ± 3.1	50.6 ± 3.1
Breakthrough 8×8	50	52.8 ± 3.1	48.4 ± 3.1
NoGo 9×9	110	50.3 ± 3.1	48.3 ± 3.1
Pentalath	90	**55.3** ± 3.1	52.4 ± 3.1
		SHOT Solver	
Amazons		49.6 ± 3.1	49.4 ± 3.1
Breakthrough 8×8		**53.8** ± 3.1	**53.4** ± 3.1
NoGo 9×9		50.1 ± 3.1	51.3 ± 3.1
Pentalath		**60.4** ± 3.0	**70.5** ± 2.8

Table 5.6: UCT, H-MCTS, and SHOT with and without solver, random play-outs.
Win percentages for the players with the solver enabled, 1,000 games.

The implementation of the solver was validated empirically, by comparing the performance of UCT, H-MCTS, and SHOT against themselves with the solver enhancement enabled. The results of this experiment are shown in Table 5.6. For H-MCTS without the solver enhancement, the B values used in Subsection 5.2.1 were applied, the B values presented in the table are those obtained from the results shown in Figure 5.2, this ensues both versions of the algorithm use a reasonable value for B.

The use of a solver has the most influence in Pentalath and Breakthrough. In Amazons and NoGo, games are mostly decided at the start, or in the middle of the game, meaning that the added benefit of discovering solved nodes near the end adds little to the overall performance. Although the solver did not result in a significant performance increase in most cases, it is possible that longer search-times and the use of heuristic play-outs results in an increase in performance overall.

Notably, the SHOT solver achieves a significantly high performance increase in both experimental cases in Pentalath. In Pentalath, all unoccupied positions can be played at any time, meaning that at each ply, excluding rarely occurring suicide moves, $o_p - 1$ positions can be played where o_p is the number of unoccupied positions at the parent. This also means that if any position results in a direct win for a player, it does so via multiple paths in the tree, which consequently results in almost all children of the root achieving a high value. Because of this, the algorithm does not recommend moves that will let it win fast, but possibly postpones its victory allowing its opponent to strengthen his position. In this case, the solver makes sure that when a solved node is found, it is excluded from normal search and immediately back-propagated, *i.e.,* when a solved position is found it is immediately played, instead of continuously searching for similarly good options.

Table 5.7 shows the results for games played with the solver enabled in both H-MCTS and UCT. Unlike Table 5.3 there is a significant performance increase in four cases when the solver is enabled. It is likely that the increased exploration performed by SHOT is responsible for finding solved nodes faster, or in parts of the tree that UCT is less likely to visit. Similar to the results for H-MCTS against UCT in Table 5.3, the results give evidence that a higher number of play-outs give increasingly better results in Pentalath. To validate this conjecture an additional experiment was run with a budget of 50,000 play-

Game	B	**10,000 play-outs**	**25,000 play-outs**
Amazons	90	**64.9** ± 2.1	**62.5** ± 2.1
Breakthrough 8×8	50	**54.5** ± 2.2	48.4 ± 2.2
NoGo 9×9	110	49.6 ± 2.2	46.8 ± 2.2
Pentalath	90	50.8 ± 2.2	**53.5** ± 2.2

Table 5.7: H-MCTS Solver vs. MCTS-Solver, random play-outs.
Win percentages with respect to H-MCTS, 2,000 games.

Game	B	**10,000 play-outs**	**25,000 play-outs**
		Heuristic play-outs (no solver)	
Breakthrough 8×8	70	50.9 ± 3.1	**56.6** ± 3.1
		Heuristic play-outs & solver	
Breakthrough 8×8	70	**56.7** ± 3.1	**61.3** ± 3.0

Table 5.8: H-MCTS vs. UCT, heuristic play-outs, with/without solver.
Win percentages with respect to H-MCTS, 1,000 games

outs, which resulted in a win-rate for H-MCTS of **55.7**%±3.1, another significant increase in performance over UCT.

A heuristic play-out policy was developed for Breakthrough, it selects moves during play-out over a non-uniform distribution based on the properties of the moves. 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. Moreover, (anti-)decisive (Teytaud and Teytaud, 2010) 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.

For the results in Table 5.8, the informed play-out policy is used to select moves for Breakthrough. H-MCTS benefits more from the informed play-outs than UCT in Breakthrough, both when the solver is disabled, and even more so when it is enabled. This may be due to Breakthrough's nature of leading the search into traps (Ramanujan *et al.*, 2010). Although these traps may remain undetected when random play-outs are used, they become more apparent with an informed policy. In this case SHOT has the advantage of spreading budget more evenly over different options, whereas UCT may have insufficient budget remaining to search for, and evaluate alternatives.

Time-based Experiments

Because H-MCTS cannot be terminated at any time, the experiments performed in this section are based on a fixed budget of simulations. However, Cazenave showed that SHOT was generally twice as fast as UCT in his NoGo engine (Cazenave, 2014). The set-up for this experiment is such that for each move one algorithm's execution is timed using a fixed budget of play-outs, and its opponent is allocated a search-time based on the measured time spent. For example, in an experiment where SHOT plays against UCT, SHOT would be allocated a fixed play-out budget of 25,000, the time it takes SHOT to perform these play-outs t^m is measured at each move m, subsequently, a search-time t^m is allocated to UCT. This method assigns both players the same search time and allows us to determine whether one technique is faster than the other, and how this affects performance.

The results for the time-based runs with H-MCTS Solver against MCTS-Solver are shown in Table 5.9. Note that in all games, a different B constant was found to be optimal than the fixed-budget experiments. Because B is responsible for the size of the simple regret tree, it also plays a role in the speed of the simulations. In all cases there is no significant change in performance over the results presented in Table 5.7. In Pentalath the increased speed of H-MCTS appears to increase performance up to 60.7%

Game	B	**10,000 play-outs**	**25,000 play-outs**
Amazons	10	**63.4** ± 3.0	**62.1** ± 3.0
Breakthrough 8×8	70	**54.6** ± 3.1	50.6 ± 3.1
NoGo 9×9	30	49.5 ± 3.1	47.4 ± 3.1
Pentalath	50	51.5 ± 3.1	**60.7** ± 3.0

Table 5.9: H-MCTS Solver vs. MCTS-Solver, time-based, random play-outs.
Win percentages with respect to H-MCTS, 1,000 games.

over UCT. An additional 1,000 games were played for Pentalath with a budget of 50,000 play-outs, which resulted in a win-rate for H-MCTS of **59.0**%±3.1.

Like SHOT, H-MCTS spends less time in the tree than UCT, and therefore the faster the play-outs, the higher the increase in play-outs per seconds for H-MCTS. In the framework used, Pentalath allows the highest number of play-outs per second. Measured over 10 games, H-MCTS performed an average of 26,165±919 play-outs per second, whereas UCT ran 24,385±846 play-outs per second in the same matches, a 6.8% increase in speed for H-MCTS.

5.2.5 H-MCTS Upper/Lower Bounds

In Table 5.10 the results of the Upper/Lower Bounds enhancement are presented. Games were played by the H-MCTS without enhancements against UCT. The C constant used in the enhancement is set to the same constant used by UCT. The results show no significant improvements over those presented in Table 5.3, although the enhancement does not decrease performance significantly in all but one case, it adds no benefit over the default version of H-MCTS. Evidently, the added exploration inherently performed by Sequential Halving is responsible for the performance gains achieved in previous experiments.

Game	B	**10,000 play-outs**	**25,000 play-outs**
Amazons	90	**67.4** ± 2.9	**60.1** ± 3.0
Breakthrough 8×8	70	49.3 ± 3.1	51.6 ± 3.1
NoGo 9×9	90	52.0 ± 3.1	49.6 ± 3.1
Pentalath	50	44.6 ± 3.1	52.5 ± 3.1

Table 5.10: H-MCTS with Upper/Lower Bounds vs. UCT, random play-outs.
Win percentages with respect to H-MCTS, 1,000 games.

5.2.6 H-MCTS and the UCB Constant

For the experiments in this section so far, the values for the C constant for UCB, which is used by UCT 2.3 to determine the rate of exploration, was tuned empirically by experimentation for UCT only. So far, the same value was used in the experiments for both UCT and H-MCTS. However, because H-MCTS broadly explores near the root it is probable that a varied values for the C constant in H-MCTS affect performance differently from UCT.

In Figure 5.3, results are depicted for H-MCTS using different values for its C constant. In each graph the constant previously tuned for UCT, C* is marked. Each game is played against MCTS-Solver using C* as the value for C.

The graphs show that for these games, the performance related to the C values used by H-MCTS and UCT do not vary significantly. In Pentalath, H-MCTS' best measured value for C differs from UCT's. It is likely that in Pentalath, H-MCTS benefits from an exploiting configuration of UCT, relying more on the values of nodes than their confidence bounds. This is possibly due to the extra exploration provided by Sequential Halving near the root, allowing fast exploitation on deeper plies.

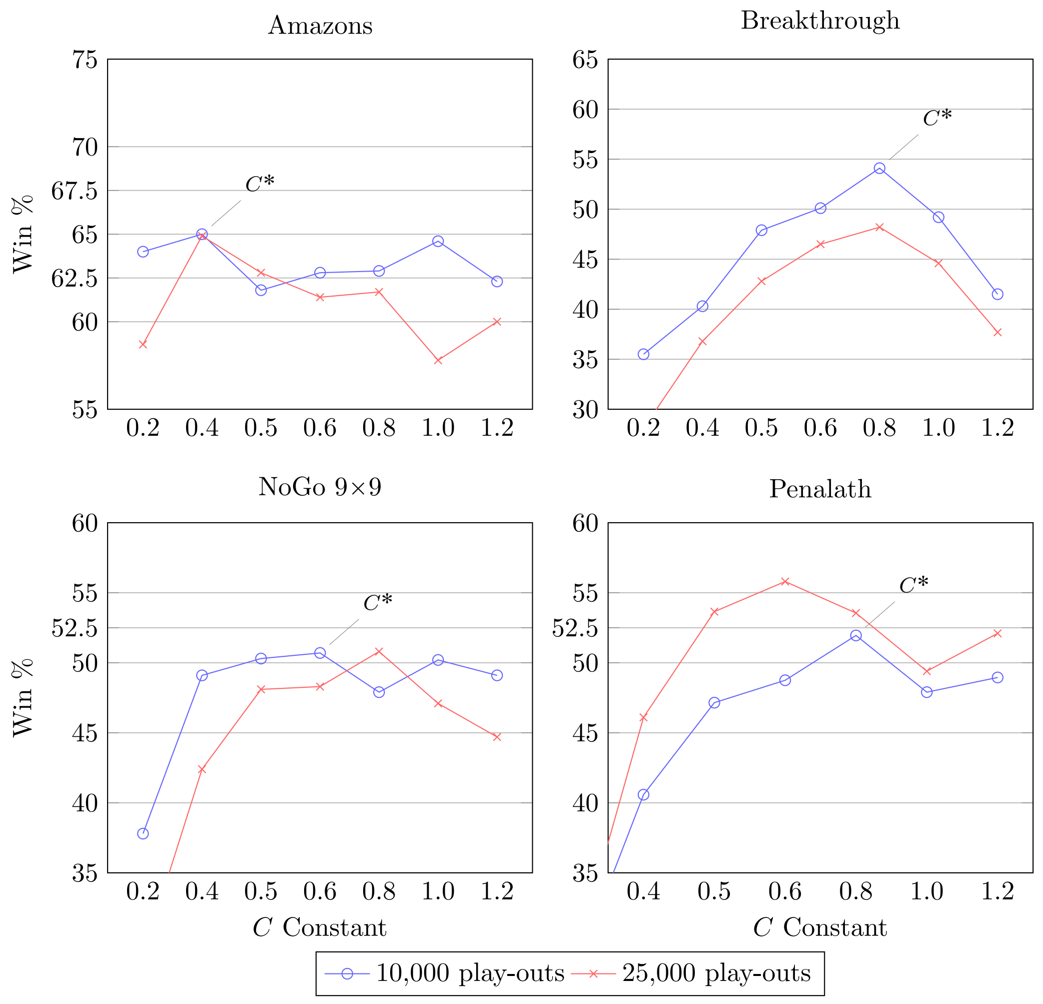

Figure 5.3: H-MCTS Solver vs. MCTS-Solver, varying values for H-MCTS' C constant. Win percentages with respect to H-MCTS. 1,000 games per data point.

Using the tuned $C = 0.6$ constant found for Pentalath, three additional experiments were performed to determine the influence of the tuned parameter. The results of these experiments are presented in Table 5.11, compared to the results presented in Table 5.7, in the 25,000 case H-MCTS' performance apparently increased by up to 2.2%, which is not a significant improvement over the previous results.

Game	B	**10,000 play-outs**	**25,000 play-outs**	**50,000 play-outs**
Pentalath	90	48.6 ± 2.2	**55.7** ± 2.2	**55.6** ± 2.2

Table 5.11: H-MCTS Solver vs. MCTS-Solver, tuned C for H-MCTS: $C = 0.6$, and UCT: $C = 0.8$. Win percentages with respect to H-MCTS, 2,000 games.

Because H-MCTS is less dependent on the C constant for its exploration than UCT, it is likely that when a non-optimal value is used for C, it has a smaller effect on H-MCTS' performance than it has for UCT's. To determine whether this holds, an experiment was performed in which both H-MCTS and UCT use the same, but varying C constants. This experiment aims to show whether H-MCTS is more stable with respect to the C constant than UCT.

Figure 5.4 shows the results of this experiment. In each graph, the constant previously optimized $C*$ is marked. In every domain tested, when both H-MCTS and UCT use a non-optimal C constant,

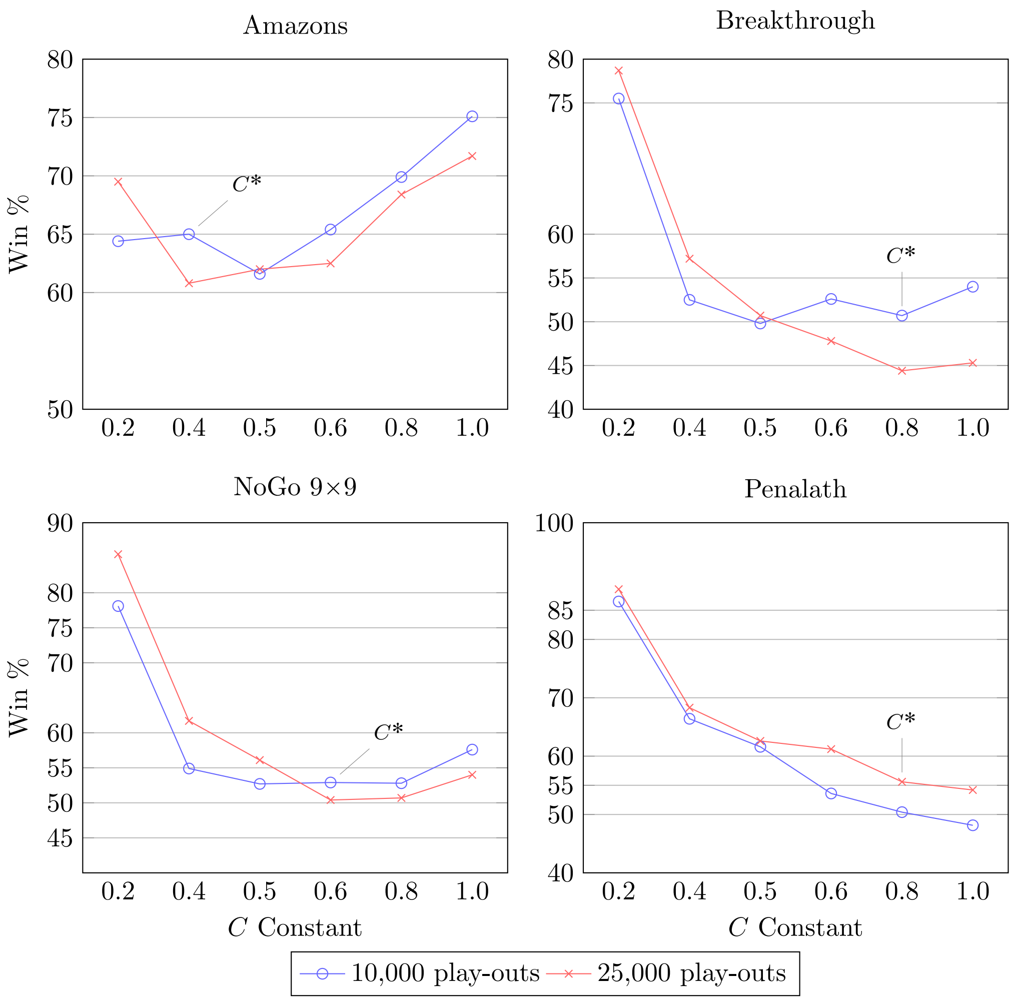

Figure 5.4: H-MCTS Solver vs. MCTS-Solver using the same UCT C constant. Win percentages with respect to H-MCTS. 1,000 games per data point.

H-MCTS significantly outperforms UCT in distinct cases. Using a low constant such as 0.2 or 0.4, which means UCT will exploit promising moves faster, H-MCTS significantly outperforms UCT in all cases. When a higher than optimal value is used this increase in performance is only visible in Amazons and NoGo. In Pentalath and Breakthrough a value for C which is too high is in some cases better handled by UCT.

This result is important in domains such as General Game Playing (GGP), because in GGP agents have no prior knowledge of the games being played. As such, the UCT constant cannot be optimally tuned for each game in GGP competitions. Using H-MCTS with an approximate value for B possibly circumvents this problem, because it is less sensitive to suboptimal values for C.

Chapter 6

Conclusion

Chapter contents: Reflecting on the research questions and problem statement, conclusions are drawn. Moreover, possible directions for future research are provided.

6.1 Research Questions

In this section the findings presented in this thesis are related to the research questions proposed in Chapter 1.

Research question 1: How can a search technique minimize both simple, and cumulative regret in a game tree?

In Chapter 2 the main findings of the research performed be Bubeck *et al.* (2010) were laid out in order to form the basis for minimizing both types of regret in a single tree. Concluding that cumulative and simple regret share conflicting bounds, and that no single selection policy can efficiently minimize both types of regret. Therefore, a combination of selection policies is required when minimizing both types of regret. Moreover, Chapter 2 discusses two recently introduced methods whit low bounds on simple regret in multi-armed bandits, Successive Rejects and Sequential Halving. Both are potential candidates to use as a selection policy in MCTS. In Chapter 3, research into two state-of-the art MCTS techniques based on simple regret was discussed. The scheme presented by Tolpin and Shimony (2012) gives the inspiration for minimizing simple regret only near the root. Moreover, a more recent algorithm SHOT (Cazenave, 2014) was shown to improve performance in games. SHOT uses Sequential Halving as its selection policy, and is therefore a suitable candidate to minimize simple regret in MCTS. The two methods discussed in Chapter 3 were combined in Chapter 4, in which a new technique, Hybrid MCTS (H-MCTS) was proposed, which combines SHOT and UCT in a single tree.

H-MCTS uses SHOT near the root, and UCT when overall budget is low, and therefore benefits from both the low simple regret offered by Sequential Halving, and the fast convergence and any time property of UCT. 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.

Research question 2: When should selection switch from simple to cumulative regret minimization in the tree?

Bubeck *et al.* (2010) showed that only when given a sufficiently large budget, should simple regret minimization be preferred over UCT. 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. Based on the available budget, H-MCTS' simple regret tree can expand deeper to provide better bounds on simple regret on the best-replies of rooted subtrees.

In Chapter 5 results for different values of B are shown for Amazons, Breakthrough, NoGo and Pentalath. In most cases the tuned value remains stable when the number of allocated play-outs increases.

Research question 3: Do pure exploration selection policies in H-MCTS improve performance in two-player games?

H-MCTS uses Sequential Halving as its simple regret minimizing selection policy. To answer this question, experiments were performed, of which the results are presented in Chapter 5. H-MCTS' performance is assessed in six distinct two-player games. Although performance did not increase in all experimental cases, H-MCTS was able to boost performance considerably in several experiments. In Amazons and Pentalath H-MCTS gave a significant boost to performance. When using an informed play-out in Breakthrough, H-MCTS outperformed UCT given a sufficient budget. Moreover, with a fixed-budget H-MCTS outperformed SHOT in 10 out of 12 cases, it consistently outperformed SHOT given a budget of 25,000 play-outs.

- In *Amazons*, likely due to its high branching factor, H-MCTS increases performance by up to 64.9%. In all cases tested did H-MCTS improve performance over UCT. With a budget of 25,000 play-outs, H-MCTS performed significantly better against SHOT with a budget of 25,000 play-outs.
- In *AtariGo*, H-MCTS outperformed UCT with 69.1% given a budget of 25,000 play-outs. Moreover, H-MCTS won up to 61.3% of the games against SHOT.
- In *Ataxx*, H-MCTS performed better than UCT and SHOT in all experiments. H-MCTS outperformed UCT up to 65.2%.
- In *Breakthrough*, H-MCTS outperforms UCT consistently up to 61.3% when an informed play-out policy is used.
- In *NoGo*, the performance of H-MCTS is on-par with UCT. In none of the experiments did H-MCTS improve performance in this domain over UCT. However, against SHOT, H-MCTS performs better in all cases.
- In *Pentalath* performance increased up to 55.7% for H-MCTS with the solver enabled. In this domain, the tuned B constant was stable for budgets of both 10,000, 25,000, and 50,000 play-outs. Moreover, performance increased with the total budget assigned. In a time-based experiment H-MCTS outperformed UCT significantly, by 60.7%.

Finally, because H-MCTS is less sensitive to the value of UCT's C constant used by UCT, the technique consistently outperforms UCT whenever both algorithms use a suboptimal value for C.

Research question 4: How can the MCTS-Solver be adapted to work with SHOT and H-MCTS?

The H-MCTS Solver was adapted to SHOT to be used in H-MCTS, where is is combined with the MCTS-Solver (Winands *et al.*, 2008) in the UCT tree, the new technique is named H-MCTS Solver. H-MCTS Solver ensures that the subset of children selected by Sequential Halving remains valid, and redistributes all unspent budget whenever a solved node is encountered. Moreover, it handles the interconnection between the simple regret and UCT trees when UCT returns the result of a solved position.

The implementation of the solver was validated by comparing UCT, H-MCTS and SHOT against their counterparts with the respective solver enabled. It was shown to be a highly beneficial enhancement in SHOT in the game Pentalath, and to a lesser extent in Breakthrough. However, in most experiments with H-MCTS Solver, it performed on par when playing against itself with a disabled solver. This may be due to the domains chosen for experimentation, as the same experiment with the MCTS-Solver showed similar results. It is possible that in less tactical domains with fewer traps the solver has a higher impact.

H-MCTS Solver was shown to outperform MCTS-Solver in different cases. In Breakthrough, when using a heuristic play-out policy, H-MCTS Solver won up to 61.3% against MCTS-Solver. Moreover, in Pentalath the beneficial performance of H-MCTS Solver against MCTS-Solver increased from 50.8% to 55.7% given budgets of 10,000 up to 50,000 play-outs, respectively. Both cases show that there is a benefit of using the solver in H-MCTS when competing with UCT.

6.2 Problem Statement

Problem statement: How can a Monte-Carlo Tree Search variant be constructed to minimize both simple, and cumulative regret effectively?

In this thesis an MCTS technique is presented based on the results of research in regret theory. The main findings of research performed by Bubeck *et al.* (2010) were adapted 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 regret deeper in the tree (Tolpin and Shimony, 2012). Based on available budget H-MCTS' simple regret tree can expand deeper to provide better bounds on simple regret on the best-replies of subtrees of the simple regret tree. The simple regret tree is traversed in a similar manner as SHOT (Cazenave, 2014), and when a certain budget threshold B is reached, selection switches to UCT. A number of play-outs are assigned to to the selected UCT tree, and the total result of these are back-propagated to the simple regret tree.

Results show that in different two-player games, H-MCTS performs either better than, or on par with UCT. In Amazons, Ataxx, and AtariGo, H-MCTS outperforms UCT by up to 69.1%. Moreover, H-MCTS performed better than SHOT given the same allocation of play-outs in 10 out of 12 experiments performed. However, in 9×9 NoGo no significant performance increase was shown against UCT. H-MCTS was shown to provide stable results in Pentalath, given an increasing budget of play-outs H-MCTS' performance significantly increased over UCT. In Pentalath, H-MCTS Solver outperformed MCTS-Solver up to 55.7% given a play-out budget of 50,000. Moreover, results in Breakthrough give evidence that when a heuristic play-out policy is used, H-MCTS outperforms MCTS, both with the solver enabled and disabled. In the experiment presented, H-MCTS Solver won up to 61.3% against MCTS-Solver in Breakthrough with a heuristic play-out policy for both players.

H-MCTS was shown to be faster than UCT in terms of the number of simulations per second. This resulted in an increased performance in several cases, but most notably in Pentalath, where H-MCTS won up to 60.7% of the games when given the same deliberation time as UCT. In Pentalath, H-MCTS was approximately 7% faster than UCT. In Breakthrough, with heuristic play-outs, H-MCTS was approximately 4% faster than UCT. However, no significant performance improvements were achieved due to this increase in speed in Breakthrough.

The Upper/Lower Bounds enhancement was shown not to benefit performance over standard H-MCTS. This gives evidence that the increased exploration near the root performed by H-MCTS is the beneficial factor to the increased performance shown in the other experiments. When unpromising moves are removed from selection too early, the benefit of using the Sequential Halving selection policy declines.

Finally, experiments were performed to determine the robustness of H-MCTS with respect to the C constant used by UCT. The first experiment determined that in Pentalath $C = 0.6$ performed better than the previously optimized value for UCT, in all other games, the best used value for the constant did not change. The second experiment showed that H-MCTS outperforms UCT in most cases when both techniques use a suboptimal C constant. This result is important in the General Game Playing (GGP) domain. Generally, the C constant is tuned by experimentation per game. However, in GGP the agent does not know beforehand which game it is playing, and therefore the value used for C is likely suboptimal. H-MCTS outperforms UCT when using a suboptimal value for the C constant, and therefore may improve results in GGP.

6.3 Future Research

Although the hybrid technique is founded on theoretical work in both multi-armed bandits, and MCTS, it was not proven that it provides better bounds on simple regret when compared to UCT. This is work for future research. In order to show that H-MCTS exhibits lower simple regret in practice, the technique should be validated in smaller, proven games for which the game-theoretic value of each action is known.

H-MCTS outperformed UCT in several cases, which leads to the following new research questions, open for future research:

1. In Amazons, likely due to its high branching factor, H-MCTS increases performance by up to 64.9%. More research into domains with such large branching factors may show similar benefits.

2. In Breakthrough, the performance benefit of H-MCTS increased drastically when informed play-outs were enabled. Although this was only shown in Breakthrough in this thesis, it is possible that a heuristically improved engine may benefit from H-MCTS.

3. The optimal C constant to be used in H-MCTS was shown to differ from UCT in several cases. The cause of this difference, and how both the C and B constants interact remains open to future research.

4. In this thesis H-MCTS is presented as a combination of SHOT and UCT. However, it is possible that other selection policies in the simple regret tree offer better performance in specific domains. How the performance of H-MCTS is influenced in different domains by using, for instance Successive Rejects, in the simple regret tree remains an open research question.

5. H-MCTS outperforms UCT when using a suboptimal value for the C constant, and therefore may improve results in GGP. Further research in using H-MCTS for GGP may result in improved results in this domain.

The speed benefits of H-MCTS, combined with parallelization of the algorithm is open to investigation. H-MCTS can be parallelized efficiently by dividing budgets in the simple regret tree over multiple threads (Cazenave, 2014).

An open research direction relates to back-propagation in MCTS in general. Using UCT, the results of play-outs are back-propagated and averaged throughout the tree. This averaging is possible because, over time UCT tends to select the best node exclusively. However, when using a pure exploration policy, this is not the case, and back-propagated results do not tend toward a best-reply. A different back-propagation strategy may achieve better results when using for instance Sequential Halving as a selection policy.

Finally, in this thesis H-MCTS' switching point between selection policies is determined by a tuned constant B. However, alternative methods may be used to tune the switching point. For instance a fixed depth could be used or a time-constraint. Further investigation is required to find methods to determine when to change selection from cumulative to simple regret minimization.

References

Allis, L.V., Meulen, M. van der, and Herik, H.J. van den (1994). Proof-number search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91–124. [2]

Arneson, B., Hayward, R.B., and Henderson, P. (2010). Monte-Carlo Tree Search in Hex. *IEEE Trans. Comput. Intell. AI in Games*, Vol. 2, No. 4, pp. 251–258. [3]

Audibert, J., Bubeck, S., and Munos, R. (2010). Best arm identification in multi-armed bandits. *Proc. 23rd Conf. on Learn. Theory*, pp. 41–53. [4, 8, 17]

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, Vol. 47, Nos. 2–3, pp. 235–256. [4, 7, 17]

Baier, H. and Drake, P.D. (2010). The power of forgetting: Improving the last-good-reply policy in Monte Carlo Go. *IEEE Trans. on Comput. Intell. AI in Games*, Vol. 2, No. 4, pp. 303–309. [4]

Balla, R.K. and Fern, A. (2009). UCT for tactical assault planning in real-time strategy games. *Proc. of the 21st Int. Joint Conf. on Artif. Intel (IJCAI)* (ed. C. Boutilier), pp. 40–45. [3]

Browne, C., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A Survey of Monte-Carlo Tree Search Methods. *IEEE Trans. on Comput. Intell. AI in Games*, Vol. 4, No. 1, pp. 1–43. [2, 3, 11, 12]

Bubeck, S., Munos, R., and Stoltz, G. (2010). Pure exploration in finitely-armed and continuous-armed bandits. *Theoretical Computer Science*, Vol. 412, No. 19, pp. 1832–1852. [4, 7, 8, 12, 17, 18, 21, 38, 40]

Campbell, M., Hoane Jr, A.J., and Hsu, F. (2002). Deep Blue. *Artificial Intelligence*, Vol. 134, No. 1, pp. 57–83. [2]

Cazenave, T. (2014). Sequential Halving applied to Trees. *IEEE Trans. on Comput. Intell. AI in Games*, Vol. In Press. [5, 9, 12, 14, 15, 29, 34, 38, 40, 41]

Chaslot, G.M.J-B., Winands, M.H.M., Herik, H.J. vanden, Uiterwijk, J.W.H.M., and Bouzy, B. (2008). Progressive strategies for Monte-Carlo tree search. *New Math. Nat. Comput.*, Vol. 4, No. 3, pp. 343–357. [3]

Coulom, R. (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Proc. 5th Int. Conf. Comput. and Games* (eds. H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science (LNCS)*, pp. 72–83, Springer-Verlag. [2, 3, 10]

Feldman, Z. and Domshlak, C. (2012). Simple Regret Optimization in Online Planning for Markov Decision Processes. *CoRR*, Vol. abs/1206.3382. [4, 11, 12]

Feldman, Z. and Domshlak, C. (2013). Monte-Carlo Planning: Theoretically Fast Convergence Meets Practical Efficiency. *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pp. 212–221, AUAI Press, Corvallis, Oregon. [12]

Finnsson, H. and Björnsson, Y. (2008). Simulation-Based Approach to General Game Playing. *Proc. Assoc. Adv. Artif. Intell.*, Vol. 8, pp. 259–264. [4]

Gelly, S. and Silver, D. (2007). Combining Online and Offline Knowledge in UCT. *Proceedings of the Annual International Conference on Machine Learning*, pp. 273–280, ACM. [4]

Gelly, S. and Silver, D. (2008). Achieving Master Level Play in 9 x 9 Computer Go. *AAAI*, Vol. 8, pp. 1537–1540. [2, 3]

Karnin, Z., Koren, T., and Somekh, O. (2013). Almost optimal exploration in multi-armed bandits. *Proc. of the Int. Conf. on Mach. Learn.*, pp. 1238–1246. [4, 8, 9, 12, 17]

Knuth, D.E. and Moore, R.W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [2]

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *Euro. Conf. Mach. Learn.* (eds. J. Fürnkranz, T. Scheffer, and M. Spiliopoulou), Vol. 4212 of *Lecture Notes in Artificial Intelligence*, pp. 282–293. Springer-Verlag. [2, 3, 4, 10, 12, 17]

Lai, T-L. and Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, Vol. 6, No. 1, pp. 4–22. [7]

Lee, C., Wang, M., Chaslot, G., Hoock, J., Rimmel, A., Teytaud, F., Tsai, S., Hsu, S., and Hong, T. (2009). The computational intelligence of MoGo revealed in Taiwan's computer Go tournaments. *Computational Intelligence and AI in Games*, Vol. 1, No. 1, pp. 73–89. [2]

Marsland, T.A. (1983). Relative Efficiency of Alpha-Beta Implementations. *IJCAI*, pp. 763–766. [2]

Pepels, T., Winands, M.H.M., and Lanctot, M. (2014). Real-Time Monte-Carlo Tree Search in Ms Pac-Man. *IEEE Trans. Comp. Intell. AI Games*, Vol. Preprint. [3]

Powley, E.J., Whitehouse, D., and Cowling, P.I. (2012). Monte Carlo Tree Search with macro-actions and heuristic route planning for the Physical Travelling Salesman Problem. *IEEE Conf. Comput. Intell. Games*, pp. 234–241, IEEE. [3]

Ramanujan, R., Sabharwal, A., and Selman, B. (2010). Understanding Sampling Style Adversarial Search Methods. *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pp. 474–483. [10, 34]

Rimmel, A., Teytaud, O., Lee, C., Yen, S., Wang, M., and Tsai, S. (2010). Current frontiers in computer Go. *IEEE Trans. Comput. Intell. AI in Games*, Vol. 2, No. 4, pp. 229–238. [2, 3]

Robbins, H. (1952). Some Aspects of the Sequential Design of Experiments. *Bulletin of the American Mathematical Society*, pp. 527–535. [6]

Tak, M.J.W., Winands, M.H.M., and Björnsson, Y. (2012). N-Grams and the Last-Good-Reply Policy Applied in General Game Playing. *IEEE Trans. Comp. Intell. AI Games*, Vol. 4, No. 2, pp. 73–83. [4]

Teytaud, F. and Teytaud, O. (2010). On the huge benefit of decisive moves in Monte-Carlo Tree Search algorithms. *IEEE Conference on Computational Intelligence and Games*, pp. 359–364, IEEE. [34]

Tolpin, D. and Shimony, S.E. (2012). MCTS Based on Simple Regret. *Proc. Assoc. Adv. Artif. Intell.*, pp. 570–576. [4, 5, 11, 12, 13, 18, 38, 40]

Turing, A. (1953). Chess. *Digital Computers Applied to Games*, pp. 288–295. Reprinted in 1988 in Computer Chess Compendium, pages 14-17. [2]

Winands, M.H.M. and Björnsson, Y. (2011). $\alpha\beta$-based Play-outs in Monte-Carlo Tree Search. *IEEE Conf. Comput. Intell. Games*, pp. 110–117. [4]

Winands, M.H.M., Björnsson, Y., and Saito, J-T. (2008). Monte-Carlo Tree Search Solver. *Proc. Comput. and Games*, Vol. 5131 of *LNCS*, pp. 25–36. [4, 5, 17, 23, 39]

Winands, M.H.M., Björnsson, Y., and Saito, J-T. (2010). Monte Carlo Tree Search in Lines of Action. *IEEE Trans. Comp. Intell. AI Games*, Vol. 2, No. 4, pp. 239–250. [3]