University of Alberta

Investigating UCT and RAVE: Steps Towards a More Robust Method

by

David Tom

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Spring 2010
Edmonton, Alberta

Examining Committee

Martin Müller, Computing Science

Michael Buro, Computing Science

External, Robert Hayes, Chemical and Materials Engineering

Abstract

The *Monte-Carlo Tree Search* (MCTS) algorithm *Upper Confidence bounds applied to Trees* (UCT) has become extremely popular in computer games research. Because of the importance of this family of algorithms, a deeper understanding of when and how their different enhancements work is desirable. To avoid hard-to-analyze intricacies of tournament-level programs in complex games, this work focuses on a simple abstract game: *Sum of Switches* (SOS).

In the SOS environment we measure the performance of UCT and two of popular enhancements: *Score Bonus* and the *Rapid Action Value Estimation* (RAVE) heuristic. RAVE is often a strong estimator, but there are some situations where it misleads a search. To mimic such situations, two different error models for RAVE are explored: *random error* and *systematic bias*. We introduce a new, more robust version of RAVE called RAVE-max to better cope with errors.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Martin Müller for making this project possible and providing invaluable knowledge and advice throughout the course of its realization. I know that I have gained a lot if I have learned just a fraction of what he has taught me. His advice has been both helpful in the theoretical aspects and technical aspects of this project.

The SmartGame library which my program FUEGO-SOS was built upon, was written by Martin Müller and Markus Enzenberger. They deserve my thanks for creating an excellent piece of software and being there to help me understand it when I encountered problems. I would also like to thank my examiners Dr. Michael Buro and Dr. Robert Hayes for taking the time to evaluate my work.

Lastly, I would like to thank my family and friends. They have been wonderfully patient and supportive. Perhaps they still do not understand my masters career, but they helped me understand that the pursuit of knowledge is something for oneself.

Table of Contents

List of Figures

List of Acronyms

AMAF All-Moves-As-First

AI Artificial Intelligence

MC Monte-Carlo

MCTS Monte-Carlo Tree Search

RAVE Rapid Action Value Estimation

SOS Sum of Switches

UCT Upper Confidence bounds applied to Trees

Chapter 1

Introduction

1.1 Monte Carlo Tree Search Methods

Monte-Carlo Tree Search (MCTS), especially in form of the UCT algorithm [25], has become an immensely popular approach for game-playing programs. MCTS has been especially successful in environments for which a good evaluation function is hard to build, such as Go [22] and General Game-Playing [16]. MCTS-based programs are also on par with the best traditional programs in Hex [3], Amazons [26], and Lines of Action [44]. Recently MCTS has been successfully applied to single-agent search as well [12, 32].

Part of the success of MCTS is due to enhancements developed to improve its effectiveness in games. Methods inspired by Schaeffer's history heuristic [33] include All-Moves-As-First (AMAF) [6] and RAVE [19]. Whereas the value of a move is normally computed from simulations where the move is the first one played, these heuristics use all simulations where the move is played at any point in the game. This produces a low variance estimate that is fast to learn [19]. Methods such as progressive pruning [5] focusing simulations on strong branches by ignoring branches that have lower simulated means.

While the game-independent algorithms above can be used with minor variations across different games, typical tournament-level programs add a large number of game-specific enhancements, such as opening books [7] and specialized playout policies [14]. Further examples are patterns [22, 11] and tactical subgoal solvers in Go [30], and virtual connections in Hex [2].

As an example of a tournament-level Go program, consider *Fuego* [13]. The computer Go program Fuego uses a game independent UCT engine as a base with various game specific enhancements including RAVE, prior knowledge, specialized playout policies, and the capability of multi-threaded search.

1.2 Motivation

While practical applications abound, up to this point there has been relatively little detailed analysis of the core MCTS algorithm and its enhancements. Gaining a deeper understanding of its behaviour

and performance is difficult in the context of complex game playing programs. Rigorous testing, evaluation and interpretation of the results is necessary but difficult to do in such environments. A simpler, well-controlled environment seems necessary.

Since MCTS is a relatively new approach, there are a large number of open research questions, both in theory and in practice. For example,

- How does the performance of MCTS vary with the complexity and type of game that is played?
- What are the conditions on a game for which a specific enhancement works? How much does it improve MCTS in the best case? When and how do these enhancements fail?
- How should a general framework for MCTS be designed, and how can it then be adapted to a specific game?

The final item is addressed in practice by the Fuego system [13], an open-source library for games which includes the MCTS engine used for the experiments in this thesis. The experiments help to address some of the other research questions regarding MCTS.

One way to study questions about MCTS in more precision than is possible for real games is to use highly simplified, abstract games for which a complete mathematical analysis is available. Such games allow a deeper study of the core algorithms while avoiding layers of game-specific complexity in the analysis.

In this thesis, a simple artificial game called SOS is used for an experimental study of MCTS algorithms. In particular, SOS is selected as a close to ideal scenario for the RAVE heuristic. Through experiments in our software FUEGO-SOS, we analyze the behaviour of UCT and RAVE in this well-controlled environment. We manipulate the complexity of the SOS game, as well as the strength and accuracy of UCT and RAVE. This analysis results in the development of RAVE-max as a solution for dealing with games involving poor RAVE accuracy. Parts of this work have previously been printed in [40, 41]. The rest of this thesis is organized as follows: Chapter 2 presents concepts and methods used in games Artificial Intelligence (AI) research and the implementation of current methods, such as UCT, in Fuego. Chapter 3 introduces the research questions addressed in this thesis. Chapter 4 compares other models used to study MCTS and introduces the SOS game model. Chapter 5 describes our experiments on SOS involving UCT and RAVE. Chapter 6 concludes with a discussion of our results and ideas for future work.

Chapter 2

Overview of the Research Area

This chapter discusses classical and contemporary techniques used in the computer study and play of two-player sequential games. The Fuego framework mentioned in Section 1.2 is an example of game-playing software containing many of the techniques described. Some implementation details of Fuego are also discussed in this chapter as the framework is used to develop the software used in the experiments of Chapter 5.

2.1 Game Trees

Classical board games have often been used as the focus of games AI research. Games that have been studied include Go [4, 28], Chess [8], Checkers [34, 35], Amazons [29], Hex [3], Havannah [39], and Lines of Action [43]. The structure used to model the state space of such games is a type of *directed graph* called a *game tree* [23, 15]. A directed graph, $G = (V, E)$, is an ordered pair formed by a set of *vertices* V and a set of *directed edges* $E \subseteq V \times V$. We also define the form $e_{a,b} = (v_a, v_b)$ for use as shorthand. A *path* from vertex v_1 to v_n is a sequence of vertices $s = (v_1, v_2, \ldots, v_n)$ such that $e_{i,i+1} \in E \; for \; 1 \leq i < n$. In a game tree, vertices are commonly referred to as *nodes* and edges as *branches*, *moves*, or *actions*. A tree T is a graph in which for all v_i there exists exactly one path from v_0 to v_i, where v_0 is called the *root node* of the tree. If $\exists e_{1,2} \in E$, then v_1 is called the *parent* of v_2 and v_2 is the *child* of v_1. If $\exists e_{1,2}, e_{1,3} \in E$, then v_2 and v_3 are called *siblings*. If $\exists p = (v_0, \ldots, v_n) \; in \; T$, then $v_0, ..., v_{n-1}$ are said to be *ancestors* of v_n. Nodes of a game tree represent states of the game and edges represent actions or moves that cause transitions between those states. The *root node* v_0 represents the initial state of the game and has only outgoing edges, whereas *leaf nodes* have no outgoing edges and represent the terminal states of the game; all other nodes represent intermediate states and are referred to as *interior nodes*. Leaf nodes are associated with a *payoff*, $\pi(v_t) \in \Re$, which we define as the result Player 1 receives when the leaf node v_t is reached. We will only be discussing 2-player zero-sum games; the player playing at the root node is labeled p_1 and the other player is labeled p_2. The payoff for p_2 is the negative of the payoff for p_1.

Researchers strive to *solve* games using game trees. To solve a game means to know, at the root

node, the game result and a strategy required to attain that result. This definition of solving a game is actually the definition of a weak solution [1], but for the purposes of this thesis it suffices. Solving a game is trivial with a complete game tree, as it enumerates all possible strategies [24]. However, a complete game tree has approximately $\Theta(b^d)$ leaf nodes; where d is the depth of the game tree and b is the number of legal moves at each state. This number is very large for most games; *e.g.*, estimated to be about 10^{31} for 8x8 Checkers [1]. As it is unfeasible to store or even calculate trees of such size, a partial game tree must be used instead. This is acceptable as only a subtree of the complete game tree is necessary to solve a game [24]. The smallest tree necessary to solve a game is called a *minimal proof tree*. A proof tree P is a subtree of game tree T for a property x that satisfies the following properties:

1. $v_0 \in P$ and $v_0 = root(P)$.
2. if $v_i \in P$ and $v_i \in leaves(P)$, property x holds in v_i.
3. otherwise, if $v_i \in P$ and p_1 is to play at v_i, then $\exists v_j \in children(v_i), v_j \in P$.
4. otherwise, if $v_i \in P$ and p_2 is to play at v_i, then $\forall v_j \in children(v_i), v_j \in P$.

Property x is defined as a condition for one of the players; *i.e.* if property x is "p_1 has won", then T is a complete winning strategy for p_1. The problem is determining how to intelligently build and utilize a proof tree while mitigating time and space complexity issues. Although the ultimate goal in games research is to solve a game, a more accessible goal is often to produce the strongest player possible. The *minimax* [31, 36] algorithm is used for evaluating a game tree and determining the best move to play.

2.2 Minimax

The minimax algorithm involves labeling p_1 as the MAX player and p_2 as the MIN player. A node is a MAX node if it is MAX's turn to play and a MIN node if it is MIN's turn. Beginning at the leaf nodes, we back up the payoff values toward the root node. Nodes are assigned values $\pi(v_i)$ based on the player whose turn it is. The value $\pi(v_i)$ for each node is known as its minimax value and represents the game result obtained if both players play optimally from that point on. $\pi(v_n) = \max(\pi(v_i), \ldots, \pi(v_j))$, where $v_i, \ldots, v_j \in children(v_n)$ if v_n is a MAX node at v_n and $\pi(v_1) = \min(\pi(v_i), \ldots, \pi(v_j))$, where $v_i, \ldots, v_j \in children(v_n)$ if v_n is a MIN node. Through this process the payoff assigned to the root node, $\pi(v_0)$, is the solution of the game and is associated with the move which leads to the node with the same payoff.

2.2.1 Minimax Search

The basic method of minimax search involves a depth first expansion of the tree, followed by minimax backups of the payoffs, and a best-first search to determine a winning strategy. The search

is guided by the result of an *evaluation function* applied at each node encountered. An evaluation function assigns a value $\hat{\pi}(v_i)$ to each node in a game tree. In a minimax search the evaluation function assigns minimax values to nodes. For interior nodes $\hat{\pi}(v_i)$ is the minimax value obtained from backups. For terminal nodes $\hat{\pi}(v_i)$ is the payoff value associated with the game state. In practice tree expansion is only performed to some depth d, as the storage and computation of a complete game tree is intractable. If d is less than the depth of the complete game tree, then this expansion does not contain all the leaf nodes of T. The payoff values must be estimated for some leaf nodes in the partial game tree that is generated. The accuracy of the estimates and the search result depend on the *heuristic function* used at the leaf nodes of the partial tree. The heuristic function estimates the minimax value of node v_i based on the game state represented by the node. Typically a deeper expansion results in a more accurate estimate. However, the uniform expansion used in basic minimax search involves expanding and storing many unnecessary nodes, causing the algorithm to quickly reach computational and storage limits.

2.2.2 $\alpha\beta$ Search

To improve upon these constraints, minimax is often used in conjunction with alpha-beta ($\alpha\beta$) pruning [31]. In $\alpha\beta$ pruning two bounds, α and β, are maintained during the search. During the depth-first traversal of the partial game tree, α holds the highest value available at a MAX node along the path to the current state and β holds the lowest value available at a MIN node along the path. α is initialized to $-\infty$ and β is initialized to $+\infty$. If at node v_i $\alpha > \beta$, then v_i need not be explored and can be pruned. In the best case, $\alpha\beta$ pruning allows us to evaluate a significantly smaller number of nodes, the square root of the number of leaf nodes in the tree. This reduction allows for a search that is twice as deep as without pruning, producing more accurate $\hat{\pi}(v_i)$ estimates and better play.

$\alpha\beta$ search has been successful in domains such as Checkers, Chess, and Go-moku, and has produced stronger-than-human computer players [4]. However, in domains such as Go and General Game Playing, alpha-beta search has been less successful. This result has been attributed to the difficulty of producing a strong heuristic evaluation function in these domains [4, 16]. Recent developments incorporating *Monte Carlo Tree Search Methods* in computer Go and General Game Playing have shown very significant improvement over traditional alpha-beta approaches [6, 16].

2.3 Monte Carlo Tree Search Methods

Monte-Carlo (MC) Methods have long been used in the physical sciences to simulate physical systems [27]. However, MC Methods were not widely employed in zero-sum games research until Brügmann's work on "Monte Carlo Go" [6]. Multiple MC-based tree search methods simultaneously gained popularity in the 2000s following Brügmann's work [9, 10, 22]. These methods became collectively classified as MCTS algorithms. The basic MCTS algorithm involves generating a game tree using random simulations and collecting statistics in the tree to find the best solution available

[9]. The game tree is grown incrementally in a best first manner guided by the simulations. The tree is initialized to have only the root node, v_0. MCTS simulations consist of two phases: during the *in-tree phase*, a best first search is conducted to find the leaf node v_n of the partial tree with the best value according to an evaluation function such as the UCT formula [25]; during the *play-out phase*, random simulations are performed with the starting state v_n. If v_n is visited for the second time during simulations, v_n is expanded, *i.e.* all children of v_n are added to the tree. Statistics are kept for each node such as the average outcome $\bar{X}(v_n)$ of all simulations resulting from that node. The evaluation function is typically composed of $\bar{X}(v_n)$ and an exploration component in order to guide the search towards both promising and nodes with high uncertainty about their evaluation. Since the in-tree phase directs simulations in a best-first manner, if a large number of simulations are conducted, stronger moves are simulated far more frequently than other moves. The exploration component of the evaluation function allows for weaker moves to occasionally be simulated over stronger ones. As long as the exploration component is never zero, given an infinite number of simulations, the $\hat{\pi}(v_i)$ values of all nodes would converge to their true minimax values, $\pi(v_i)$ [10].

The problem of devoting search to exploring unverified options or exploiting current maxima is called the *exploration-exploitation dilemma*. Unlucky simulations may lead the search down a sub-optimal branch that may not reach a refutation until many simulations later. However, the simulations may instead be following the optimal branch and the exploration of alternative options may simply be a waste of time and resources. To effectively use time and resources, an algorithm needs to balance between exploring and exploiting the search space in an efficient manner. The algorithm Upper Confidence bounds applied to Trees (UCT) is a popular solution to the exploration-exploitation dilemma [25]. At each node v_i UCT chooses a child that maximizes the formula:

$$\tilde{\pi}(v_j) = \bar{X}(v_j) + c\sqrt{\frac{\log T_i(v_i)}{T_{i,j}(v_i)}} \tag{2.1}$$

$T_i(v_i)$ represents the number of times the v_i was visited. $\bar{X}(v_j)$ denotes the average simulated payoff of v_j. $T_{i,j}(v_i)$ is the number of times move $e_{i,j}$ has been played at v_i. The exploration constant c is set by the user; increasing c leads to more exploration. In the basic MCTS algorithms such as UCT, the estimated minimax value of node v_j is its simulated mean, *i.e.*, $\hat{\pi}(v_j) = \bar{X}(v_j)$. By estimating minimax values through random simulations, MC methods have succeeded in domains previously limited by heuristic functions that were difficult to formulate. In particular, since the UCT algorithm is simple to implement and elegantly solves the exploration-exploitation dilemma, it is frequently employed in current game-playing programs. UCT has been successfully used in Go [21], General Game Playing [16], Amazons [26], Lines of Action [44], Havannah [39], and Hex [3].

2.4 RAVE

To produce low variance estimates for $\hat{\pi}(v_j)$, v_j must be simulated many times. However, in a large state space where there are many other candidate nodes, this may be very time consuming.

Gelly and Silver proposed the use of an algorithm called Rapid Action Value Estimation (RAVE) to reduce the time required to produce a low variance estimate for $\hat{\pi}(v_j)$ [20]. In RAVE, $\hat{\pi}(v_j)$ is a linear combination of $\bar{X}(v_j)$ and $\bar{Y}(v_j)$, where $\bar{Y}(v_j)$ is an average generated by the RAVE updates. Before we continue, let us define a term that will be used in the description of RAVE updates. We define a *move class* $M \subseteq E$ as a set of edges, such that each edge in M represents the same move of a game piece x from position a to position b. Whereas traditional MC methods update the value of a node, $\hat{\pi}(v_j)$, for all simulation results that arise from choosing move $e_{i,j}$, RAVE updates the value of $\bar{Y}(v_j)$ for all moves in the same move class that occur later in the game than v_i. That is, given $\{e_{a,b}, e_{c,d}, e_{i,j}\} \subseteq M$, if v_i is an ancestor of v_a, a simulation involving move $e_{a,b}$ will cause an update to $\bar{Y}(v_j)$.This situation is illustrated in Figure 2.4. Low variance estimates can be produced with only a few simulations using RAVE. But because RAVE updates ignore temporal information, they may not always be accurate despite the large number of simulations they represent. With enough simulations, $\bar{X}(v_j)$ becomes more accurate than $\bar{Y}(v_j)$ as it represents only the game state v_j. By varying the weighting of the $\bar{X}(v_j)$ and $\bar{Y}(v_j)$ components appropriately during search, MCTS programs can produce strong value estimates both while simulation counts are high or relatively low. RAVE has been successfully used in the games of Go [19], Havannah [39], and Hex [3].

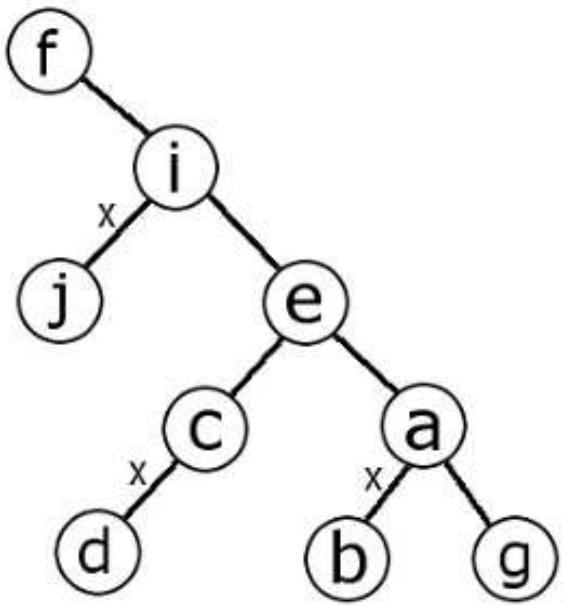

Figure 2.1: An example used to illustrate node relations that would be affected by the RAVE backup procedure. Branches $e_{a,b}, e_{c,d}, e_{i,j}$ are of the same move class, x. Simulations passing through v_d or v_b will cause an update to v_j. A simulation through v_d will not affect v_b and vice versa.

2.5 Fuego

Modern programs for games such as Hex, Go, Lines of Action, and General Game Playing now contain a MCTS engine with a number of enhancements. The Fuego Go program developed at the U of A is one such program [13]. It includes an UCT-based MCTS engine with RAVE capabilities and domain knowledge capacities. Although the Fuego framework is primarily developed for the computer Go program of the same name, its basis is the game-independent SmartGame library: a set of classes and functions to handle gameplay, file storage, and game tree search as well as other utility functions. The SmartGame library includes a generic MCTS engine with support for UCT, RAVE, and using prior knowledge. As the Fuego framework was used to develop the software used for the experiments presented in Chapter 5, relevant features of the Fuego framework are explained in this section.

Similar to other MCTS frameworks, in Fuego, the game tree is grown incrementally. A value called the *expand threshold* determines how many visits a node must receive before it is expanded. This threshold is set to one by default, meaning that all nodes are expanded on their second visit in the game tree. Unexpanded nodes are assigned a *First Play Urgency* value. The default value of 10000 is used in the experiments, which gives high priority to unexpanded nodes [25]. The UCT engine uses a modified UCT formula, with user-defined parameters controlling the search behaviour.

$$\tilde{\pi}(v_j) = \frac{T_{i,j}(v_i)}{T_{i,j}(v_i) + W_{i,j}} \bar{X(v_j)} + \frac{W_{i,j}}{T_{i,j}(v_i) + W_{i,j}} \bar{Y(v_j)} + c\sqrt{\frac{\log T_i(v_i)}{T_{i,j}(v_i) + 1}} \tag{2.2}$$

$T_i(v_i)$ represents the number of times the parent node v_i was visited. $\bar{X(v_j)}$ denotes the average reward at v_j and $\bar{Y(v_j)}$ the RAVE value of move class j at v_i. The parameter c is defined by the user to determine the influence of the UCB bound value; this parameter is usually optimized by hand, but has the default value of 0.7. $T_{i,j}(v_i)$ is the MoveCount, the number of times move j has been played at v_i. Adding 1 to $T_{i,j}(v_i)$ in the bias term avoids a division by 0 in case move class j has a RAVE value but $T_{i,j}(v_i) = 0$. The RAVE weight $W_{i,j}$ will be explained later.

Along with UCT, the SmartGame library includes an implementation of RAVE. A parametric function is used to control the influence of the RAVE value. When RAVE is active, the estimated value for a move is determined by a linear combination of the mean value and RAVE value of the move. The weighting function used here is simplified from the one originally proposed in [19], but has been found to work as well as the original formula in Fuego. The unnormalized weighting of the RAVE estimator is determined by the formula:

$$W_{i,j} = \frac{S_{i,j}(v_i) w_f w_i}{w_f + w_i S_{i,j}(v_i)} \tag{2.3}$$

The *RaveCount*, $S_{i,j}(v_i)$, represents the number of rave updates of move class j at node v_i. w_i and w_f stand for *RaveWeightInitial* and *RaveWeightFinal* respectively; these parameters determine the influence of RAVE relative to the mean value. They are manually set by the user. w_i describes

the initial slope of the weighting function and w_f describes its asymptotic bound. As the number of simulations increases, the weight of the RAVE value diminishes relative to weight of the mean value. This formula is designed to lower the mean squared error of the weighted sum; it is optimal when the weight of each estimator is proportional to the inverse of its mean squared error [37]. The default value of RaveWeightInitial is 1.0 and a suitable RaveWeightFinal is found experimentally. In our experiments, RaveWeightInitial is kept at 1.0 as we do not make any assumptions about the accuracy of early RAVE and UCT estimates.

A tournament program also contains a time control that maintains and limits the time spent in search, as the program has a finite amount of time to perform a play. We use time control only for large game experiments where the required search time becomes impractical otherwise.

Chapter 3

Research Questions and Contributions

Compared to methods such as $\alpha\beta$ search and minimax search, MCTS is a relatively new approach in games AI research. However, it has been more successful than traditional search methods in difficult domains such as Go [21] and General Game Playing [16]. MCTS has also exhibited comparable performance in other domains traditionally dominated by $\alpha\beta$ such as Hex [3], Havannah [39] and Lines of Action [44]. Despite recent success in these domains, MCTS is still in its infancy with relatively undeveloped theory. Research directed at MCTS algorithms themselves would greatly improve the effectiveness of their application to current and new domains. One way to approach this ideal is by solving the research questions posed in Section 1.2.

How does the performance of MCTS vary with the complexity and type of game that is played? That is, given game A and game B, and similar implementations of MCTS, how does the performance of MCTS differ? Furthermore, how does the performance differ within a game when you change game parameters such as board size or number of pieces? These questions can be answered by observing how MCTS interacts with the different game trees T_A and T_B. Properties such as the number of leaf nodes, the depth of the tree, and the branching factor should be related to the performance of MCTS on a given game tree. Experiments on P-game trees provide some performance benchmarks for the MCTS algorithm UCT [25]. However, other game trees may be used for specific properties such as consistent move values. If we have benchmark values for MCTS, we can know what to expect before adapting it to a new game. Prior knowledge of the affinity of a game towards MCTS allows a researcher to determine whether implementing MCTS is a profitable endeavour for his project. Furthermore, if certain properties of the game tree improve MCTS performance, we can look for ways to manipulate the game tree to better suit the algorithm's needs.

What are the conditions on a game for which a specific enhancement works? How much does it improve MCTS in the best case? Where may these enhancements fail? By knowing the conditions for which an enhancement works, benefits towards that enhancement may be reaped similar to those mentioned in the previous paragraph. In addition, there may be points in a game during

which the enhancement is more or less effective. By understanding the conditions for which an enhancement works, we can strengthen or weaken the enhancement to suit the game state. However, enhancements may be produced for research or for improving the performance of a game-playing program in competitions. There may be abundant information about an enhancement or very little. Furthermore, the enhancement may be very general or very specific, being only suitable for a particular game. Because a large number of game-playing programs in games AI research tend to be competition programs, they tend to contain several enhancements. It is difficult to discern the effect of a particular enhancement with so many conflicting variables. If we know the effectiveness of an enhancement in its best case, we can determine how well-suited it is for a particular environment. If a certain property improves the enhancement, it may be exploited to maximize the benefit. Move ordering in $\alpha\beta$ search is an example of this kind of improvement. Similarly, by knowing where enhancements are detrimental in a game, we can avoid weak play caused by misusing the enhancement.

How should a general framework for MCTS be designed, and how can it then be adapted to a specific game? How should software be produced to use and test MCTS? The Fuego framework attempts to answer these questions by providing a game-independent MCTS engine that can be adapted to specific games. Having a general framework to build upon allows users to achieve a level of consistency when comparing performance across games. Separating the game specific from the game-independent structures also improves the process of changing either.

By using the Fuego framework, we can essentially adapt MCTS to any adversarial game for testing. However, most classical games are not yet solved. This means that it is difficult to measure the correctness of moves that are played in most games. In addition, we would like to be able to control the complexity of the game on which MCTS is operating on. Overly large game depth and complexity are not necessary for determining basic algorithm properties. Thus, we propose the use of a simple abstract game for the purpose of studying MCTS. By design, this abstract game should be well suited to an enhancement(s), so we may test some of the enhancements that are popular in practice currently. The abstract game we propose to use is called Sum of Switches (SOS). In the SOS game, the value of a move is independent of when and by which player it is chosen. This represents a best-case scenario for history-based heuristics such as RAVE that generalize move performance across different points in the game.

Using the SOS game we attempt to answer some of these research questions. By approaching these research questions, this thesis has produced the following contributions:

- We demonstrate the use of SOS as a test bed for UCT and RAVE. By implementing SOS in Fuego, we show that such a general framework may be adapted to a game. Furthermore, we use the various facilities of Fuego to gather data on the performance of the algorithms. By varying the size of SOS, we establish performance baselines for UCT and RAVE based on game complexity. We also show the performance of the Score Bonus enhancement on SOS.

- We show how some of the strengths and weaknesses of RAVE can affect search in SOS. We show the use of SOS as a best case environment for RAVE by demonstrating the effectiveness of using RAVE as the sole estimator. Results we present using false RAVE updates further provide insight into the properties of UCT and RAVE algorithms. Using the indiscriminate false updates we show that UCT and RAVE are fairly resilient against noise. By using selective false RAVE updates, bad cases for RAVE can be mimicked. We show the detrimental effect that such scenarios can have on the search performance and provide a partial solution in RAVE-max, a modification of the RAVE algorithm that handles cases where the RAVE value is underestimating the true value of a node.

Chapter 4

Artificial Games

4.1 Artificial Game Models

The original UCT paper [25] contains an experiment showing the performance of UCT on the artificial *P-game tree model* [38]. Each edge, or move in this game, is associated with a random number from a specified range. The value of edges corresponding to opponent moves is negated. The value of a leaf node is the sum of the edge values along the path from the root. A positive value indicates a win for the player at the root and a negative value indicates a win for the opponent. Zero results in a draw. In such a model many properties of the tree can be easily manipulated, such as the branching factor and tree depth. However, the model does not naturally provide any benefits to UCT or RAVE, making it unsuitable to our goals of comparing best-case performance. Furthermore, the full game tree must be generated, stored, and traversed in order to obtain an optimal strategy to compare to. Larger problems are harder to model using this approach as storage requirements become harder to meet.

A special case of the P-game tree model is the *Prefix Value Game Tree Model* [18]. In a Prefix Value tree, each move in a state is associated with the amount it deviates from the optimal move in that state. The value of a node is the alternating sum of the moves along the path from the root to that node plus the value of the root node from the perspective of the turn player. By construction this means that the negamax value for each node can be produced incrementally. This property allows for large game trees to be easily constructed containing accurate node values. Determining an optimal strategy in this paradigm is trivial. The optimal strategy is determined by construction as the path containing all moves with zero value. Although this model provides a simple way to generate synthetic game trees, it does not provide any natural benefit to UCT or RAVE. Synthetic game trees may be too abstract for the purposes of researching algorithm enhancements dependent on structural properties of classical games. In order to mimic those same properties in a synthetic game tree, various changes would need to be made, but may simply complicate the model.

One of our initial intentions upon approaching this project was to observe the behaviour of UCT and RAVE in a best-case environment. This was the primary reason behind choosing an abstract

environment other than synthetic game trees as a means to test the properties of UCT and RAVE. The abstract game SOS appears to complement the RAVE algorithm by design. Furthermore, SOS also has an easily extractable optimal strategy. The scalability of the game provides a means of manipulating the difficulty level and comparing to alternative models.

4.2 Sum of Switches

Sum of Switches (SOS) is a number picking game played by two players. The game has one parameter n. In SOS(n) players alternate turns picking one of n possible moves. Each move can only be picked once. The moves have values $\{0, \ldots, n-1\}$, but the values are hidden from the players. The only feedback for the players is whether they win or lose the overall game. After n moves, the game is over. Let s_1 be the sum of all p_1's picks, $s_1 = p_{1,1} + \ldots + p_{1,\lceil n/2 \rceil}$, and s_2 the sum of p_2's picks, $s_2 = p_{2,1} + \ldots + p_{2,\lfloor n/2 \rfloor}$. Scoring is similar to the game of Go. The *komi* k is set to the perfect play outcome, $k = (n-1) - (n-2) + \ldots \pm 0 = \sum_{i=1}^{n} (n-i)(-1)^{i+1} = \lfloor n/2 \rfloor$. The first player wins *iff* $s_1 - s_2 \geq k$.

The optimal strategy for both players would be to simply choose the largest remaining number at each step. However, since both the move values and the final scoring system are unknown to the players, good moves must be discovered through exploration, by repeated play of the same game.

SOS can be viewed as a generalized multi-armed bandit game [25]. In classical multi-arm bandit problems, each game consists of picking a single arm i out of n possible arms, which leads to an immediate reward r_i, a realization of a random variable X_i. The player uses exploration to find the arm with best expected reward, and exploits that arm by playing it. In SOS, one episode consists of playing *all* arms once. The reward r_i for choosing arm i is constant, but is not directly shown to the player. Only the success of all choices relative to the opponents choices is revealed at the end of the episode.

Chapter 5

Experimental Results

The experiments investigate the properties of MCTS with UCT and RAVE. We show results for the basic behaviour of UCT in SOS followed by the basic behaviour of RAVE in SOS. Next, we justify our choices in the Move Selection Policy and the Exploration Constant. We also show the results of using the Score Bonus enhancement from Fuego on SOS. Then we present the results of experiments where RAVE is intentionally misled. By using false RAVE updates, we mimic the problems in special-case game situations that tend to be difficult for RAVE. These situations involve underestimations and overestimations on move values.

5.1 Setup

5.1.1 Graphs

This section describes the general properties of the experiments in this chapter. Each experiment compares the performance of the UCT+RAVE SOS game-playing program, implemented as described in Chapter 4, across different settings. The program is referred to as FUEGO-SOS for the remainder of this chapter. Performance with respect to time is measured by changing the number of MC simulations the program is allowed to execute before returning a search result; this simulation maximum is varied between powers of 2 from 2^1 to 2^{16}. Three types of graphs are used to plot the results of these experiments, which will be referred to as: general performance graphs, specific performance graphs, and performance differential graphs. General performance graphs plot the performance of the FUEGO-SOS player on a per-game basis. Each data point in the resulting graph represents the number of correct first plays performed by FUEGO-SOS across 1000 trials. The correct first play is known since we know the ordering of all moves, as discussed in Section 4. Figure 5.1 serves as a sample general performance graph showing the performance of FUEGO-SOS on SOS(10). The 99% confidence intervals are represented on Figure 5.1 using error bars. This interval is largest at 2^7 where it is 40.79 units wide on each side. Error bars are omitted from other experiments in this section to avoid clutter. As SOS(10) is the most common testing environment in this chapter, error is expected to be similar to this example. Specific performance graphs are used to in-

vestigate the properties of individual moves. To obtain data values, 1000 search trials are performed each using the search limit of 2^{16} simulations. In each trial, data is recorded after $2^1, 2^2, \ldots, 2^{16}$ simulations. Graphs show the averages over all 1000 runs. We show how variables of interest such as the individual move counts and estimated move values change with the number of simulations using these results. Moves are labeled with their true value from 0 to $n-1$ in SOS(n). Performance differential graphs present the performance difference between the general performance data of two different settings. These graphs show the relative performance of one setting using the other as the baseline measure. Performance differential graphs merely serve as visual aids. The corresponding values from two general performance graphs are simply subtracted to obtain these graphs.

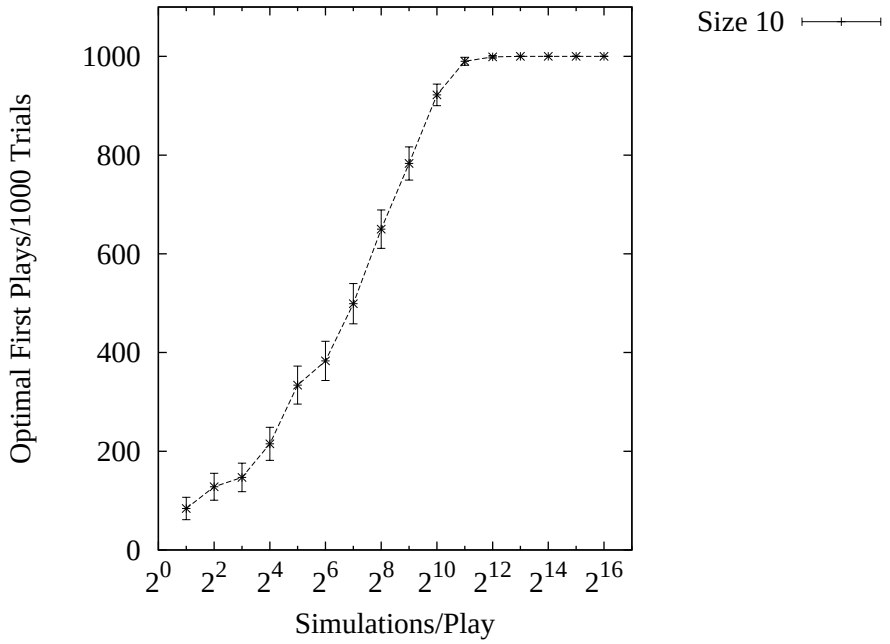

Figure 5.1: UCT on SOS(10), with 99% confidence intervals.

5.1.2 Settings

The basic UCT search engine is kept in its default form as described in Section 2.5, except in the experiments of Section 5.6.2. Most UCT search settings are kept at default Fuego values [13]. The Fuego version used in these experiments was Fuego release 0.3. Values that were changed are explained here. Default move selection in Fuego chooses the most simulated move, as this has been shown to provide more stable performance in Go. We use the highest estimated move value as our selection policy as per the original UCT [25]. FUEGO-SOS also performed better using the highest valued move than the most simulated move, as will be shown in Section 5.4. WeightRaveUpdates is another feature that is disabled in these experiments, but on by default in Fuego. WeightRaveUpdates involves scaling the RAVE updates for moves based on how late in the game they were played. This

modification to RAVE is a newer feature in the Fuego system and it is unclear how it affects the algorithm. Although it may be interesting to investigate how the WeightRaveUpdates performs in SOS in future work, preliminary results showed no significant change in performance. Fuego uses a default time limit of 10s in the search. This time limit is disabled in experiments where $n \leq 20$ to avoid bias in our results. Our experiments were performed on 2 GHz i686 computers with 1GB of memory running Linux 2.6.25.14-108.fc9.i686 Fedora release 9 (Sulphur).

5.2 Basic Behaviour

The complexity of a SOS game is solely determined by its size. SOS(n) produces a game tree of size $n!$ assuming transpositions and tree pruning are not used. For example, the complete SOS(10) game tree contains 3628800 leaf nodes. The larger the game, the more difficult it is for a game-playing program to solve. Similarly, the playing strength of a plain, unenhanced UCT game-playing program is mainly determined by a single parameter s: the number of simulations it is allowed to perform before playing a move. To establish a performance baseline for UCT in SOS, experiments varying n and s were performed. A uniform random playout policy is used during simulations in all experiments found in this chapter.

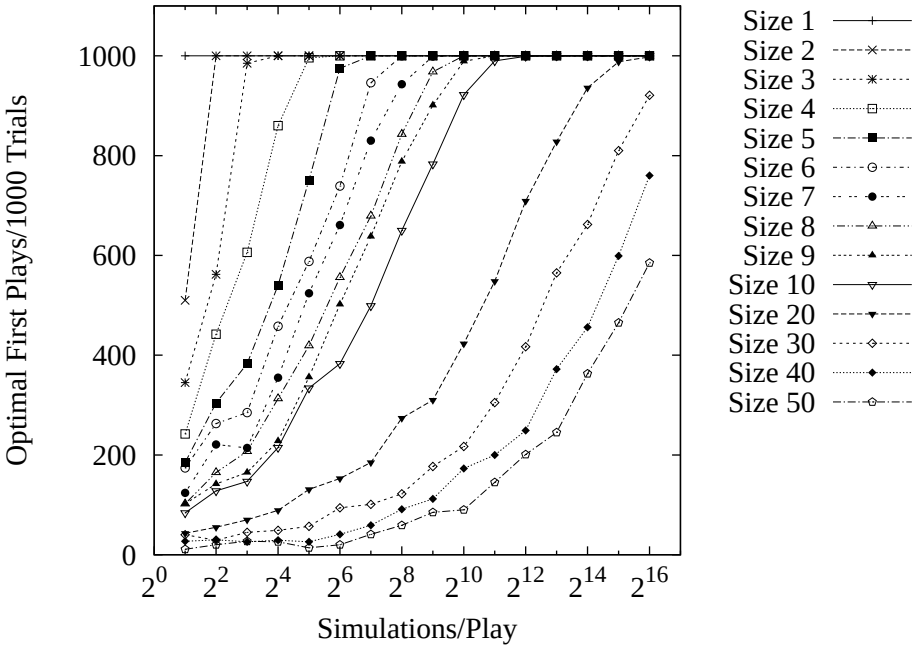

Figure 5.2: Performance of UCT without additional enhancements in SOS(n).

As n increases, convergence to optimal play becomes progressively slower. For $n \leq 10$ the program quickly converges to optimal play in under 2^{12} simulations. However, for larger games in the range $20 < n \leq 50$, the number of simulations required for convergence appears to exceed the limits of the values tested. Overall, convergence rates seem similar to those for P-games in [25] for

games with a comparable number of leaf nodes. Convergence is quickly achieved in small games, but if the game is too simple, it becomes difficult to measure changes introduced to the system. Additionally, practicality must be considered, as results should be obtained in a reasonable amount of time. Estimated simulation speeds for SOS(10), SOS(20), SOS(30), SOS(40), and SOS(50) were respectively 68000, 28000, 16000, 10000, and 7000 games per second. For further experiments, SOS(10) was chosen as a compromise between game difficulty and runtime until convergence.

5.3 RAVE

Section 2.5 describes how RAVE is used in Fuego's UCT engine and introduces the *RaveWeightFinal* variable. By manipulating *RaveWeightFinal*, we manipulate the overall and long-term influence of the RAVE value. Figure 5.3 shows experiments where RAVE is active. Compared to the performance of FUEGO-SOS with RAVE disabled, even low values of *RaveWeightFinal* such as 2 give noticeable improvements. Increasing the value of *RaveWeightFinal* improves performance, but shows diminishing returns in the range tested: settings of 1024 producing similar results to 131072. As a matter of convenience, 32768 was arbitrarily chosen as the *RaveWeightFinal* value used for most experiments involving RAVE in this chapter. The value was sufficiently large to cause strong RAVE influence and, as shown in Figure 5.3 produces very similar performance to other larger values.

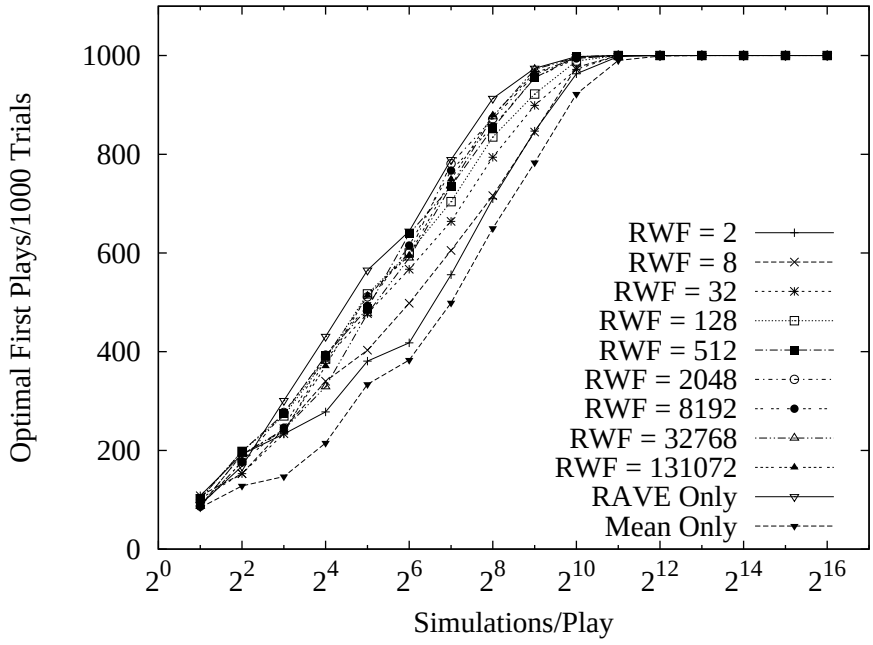

Figure 5.3: UCT+RAVE in SOS(10). Varying *RaveWeightFinal*, abbreviated as RWF in the legend.

5.3.1 Exploiting RAVE

As stated in Section 4, this game is designed as a kind of best case for RAVE: the relative value between moves is consistent at all stages of the game. In fact, in SOS it is possible and beneficial to base the UCT search exclusively on the RAVE value and ignore the mean value. Figure 5.3 includes this RAVE-only data as well. The performance of the RAVE-only setting in SOS outperforms all *RaveWeightFinal* settings tested. This method may not work in other games where the value of a move depends on the timing of when it is played. In such games RAVE may cause overestimations or underestimations in move values. The experiments in Section 5.6 approximate the difficulties that may arise if UCT+RAVE overestimations or underestimations in RAVE values occur; preliminary tests showed that these kinds of problems were simply exaggerated if RAVE were used exclusively.

5.4 Move Selection Policy and Exploration Constant

In this section we discuss the Move Selection policy and Exploration Constant and the reasoning behind their settings. As stated in Section 5.1.2, the Move Selection policy used in FUEGO-SOS chooses the move with the highest estimated value. If strictly UCT is used, the search returns the move at the root with the highest mean value, but if RAVE is enabled, the search returns the move with the highest weighted sum of mean and RAVE values. The alternative policy of choosing the most simulated move avoids the risk of "lucky" simulations skewing the search in favour of moves with less reliable estimated values. However, results in Figure 5.4 show that the two strategies perform approximately the same on SOS(10), but the highest estimated value policy appears to be more successful while the simulation counts are low.

The exploration constant c in UCT is used to control the frequency of exploration in nodes with high uncertainty. This constant has typically been hand-tuned to optimize UCT performance in different games. However, there is debate about the effectiveness of the exploration component of UCT, as some programs appear to perform best when $c = 0$. Figure 5.5 shows the results of varying c on SOS(10). Differing values of c do not seem to produce a change in the performance of FUEGO-SOS on SOS(10), so the default value of 0.7 is maintained throughout our experiments.

5.5 Score Bonus

Score Bonus is an enhancement that differentiates between strong and weak wins and losses. If game results are simply recorded as a 0 or 1, the program does not receive any feedback on how close it was to winning or losing. With score bonus, a strong win that probably contained many high-scoring moves gets a slightly better evaluation than a close win.

In SOS with score bonus, losses are evaluated in a range from 0 to γ and wins from $1 - \gamma$ to 1, for a parameter $\gamma < 1$. A minimal win is awarded $1 - \gamma$, and a maximal win a score of 1. All other game outcomes are scaled linearly in this interval. The values assigned for losses are analogous.

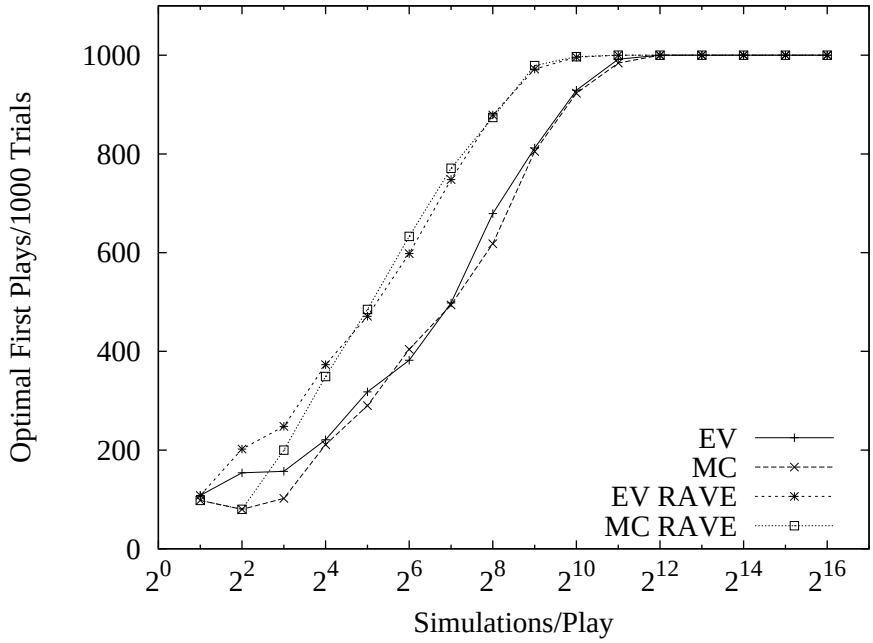

Figure 5.4: Performance values upon changing Move Selection Policy. Estimated Value is abbreviated as EV and Move Count is abbreviated here as MC. Do not confuse with Monte-Carlo from text.

Score Bonus is used in the Fuego Go program. Unpublished large-scale experiments by Markus Enzenberger showed that small positive values of γ improve the playing strength slightly but significantly for 9×9. Best results were achieved for $\gamma = 0.02$. Figure 5.6 shows results for how this enhancement performed in SOS.

We tested $\gamma = 0.1$, $\gamma = 0.05$, $\gamma = 0.02$, and $\gamma = 0.01$ in SOS to search for a suitable setting, as γ needs to be hand-tuned. Despite success in Go, score bonus fails to improve gameplay in SOS. None of the tested settings showed noticeable improvement.

5.6 False Updates

5.6.1 Indiscriminate False Updates

While RAVE has improved general performance in Go and other games, it is not always reliable. RAVE can fail in some Go positions where basic UCT succeeds. Since RAVE updates the value of *all* moves in a winning sequence and ignores temporal information, in less favourable environments moves may develop skewed RAVE values. False RAVE values can lead the search astray. In situations where specific moves are only helpful at a given time, RAVE can weaken game-play instead of improving it. Suppose that in a game, a certain last move, $Move_a$, will always lead to a win, but is useless at all other times. The high RAVE value that $Move_a$ is likely to earn early in simulations causes the game-playing program to waste time exploring paths starting with $Move_a$

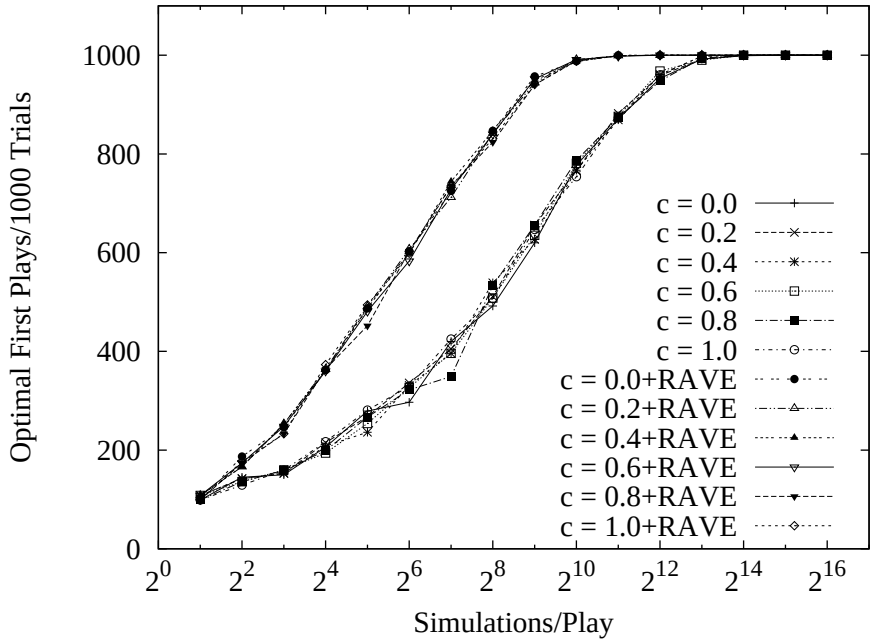

Figure 5.5: Results of varying the constant c that regulates the exploration component.

at higher points in the tree. Although these simulations continually lead to losses, the high RAVE value held by $Move_a$ keeps the search focused on it for a significant amount of time. Such a situation produces very poor value estimates when the simulation limit is reached and thus, poor play results. Conversely, the more typical case is that a move, $Move_b$, is only good if it is chosen as the first move. In this situation, $Move_b$ will gain a low RAVE value from simulations. With a low RAVE value, $Move_b$ is less likely to be sampled as the first move, meaning that many other moves will be simulated before $Move_b$ is simulated at the root node. The search is unlikely to discover that $Move_b$ is a strong move at the root and again, a poor play results.

Experiments involving random false updates can simulate the effect of misleading RAVE values. In the experiment shown in Figure 5.7, we randomly perform false RAVE updates on all moves. With a probability of μ, the RAVE update for all moves in the current simulation uses the inverse evaluation $InverseEval = 1 - Eval$ instead of $Eval$.

Even with the influence of the mean value as a steadying force, the performance of RAVE with false updates deteriorates as the value of μ increases. The decay is gradual until μ is about 0.5, where performance drops significantly. RAVE still outperforms plain UCT when μ is between 0 and 0.3. Up to a μ value of 0.5, the error introduced by the false update can be interpreted as noise that slows down convergence. Even with $\mu = 0.6$ the performance still improves with the number of simulations. However, μ values of 0.7 or higher seem to prevent convergence altogether in this simulation range. These results suggest RAVE is a robust heuristic that is resilient against a reasonable level of error.

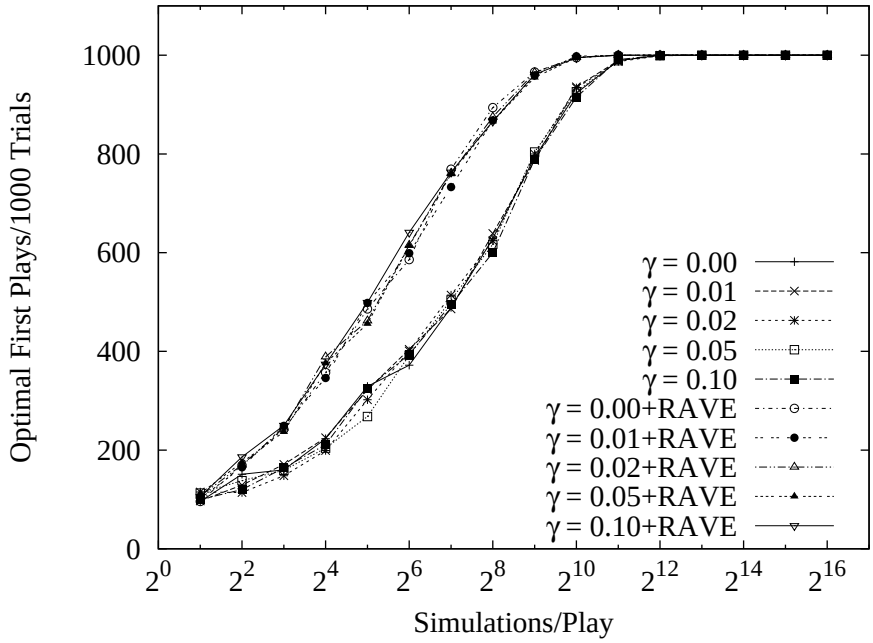

Figure 5.6: Score Bonus results involving various γ on SOS(10).

In Section 5.3.1, we showed that using RAVE without the mean value component in the UCT formula works exceptionally well in SOS. We investigate the effect of false updates on this previously favourable arrangement. The results in Figure 5.8 show a similar trend while the error rate, μ, is low. The $\mu = 0$ data corresponds to the RAVE-only line in Figure 5.3, where RAVE-only without artificial error is significantly better than plain UCT and UCT+RAVE. However, at $\mu = 0.3$ RAVE-only is already slightly worse than plain UCT. At $\mu = 0.4$, RAVE-only is performing far worse than plain UCT and UCT+RAVE with false updates from Figure 5.7. At $\mu = 0.5$ the algorithm behaviour becomes equivalent to a uniform random player, and beyond $\mu = 0.5$, plays become even worse. This result can be expected in an arrangement where RAVE is overly emphasized, or in this case the sole predictor, and RAVE is grossly skewed.

5.6.2 Selective False Updates

The false update model discussed in Section 5.6.1 is suitable for representing games where moves have different values when played in different order or other situations where move values at different times in a game have low correlation. However, that model does not actually represent the situations discussed where a specific move is of special importance at a certain point in time. In the following versions of false updates, we selectively distort the updates related to specific moves and specific outcomes. By consistently manipulating the RAVE values in a specific manner, we mimic the hypothetical result of the scenarios described at the start of Section 5.6. This model is closer to what is seen in Go, and presents a different problem for the search than the model in Section 5.6.1.

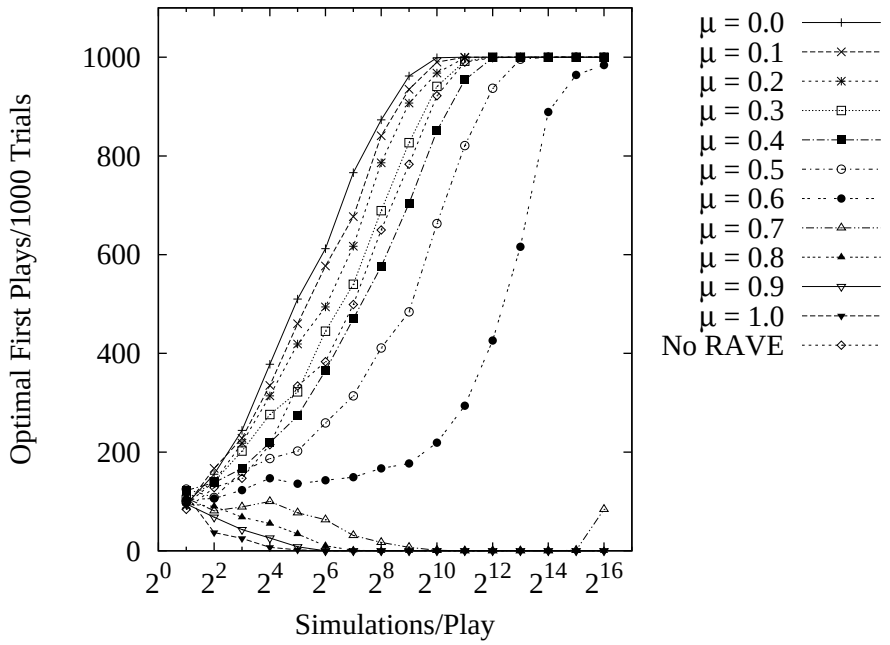

Figure 5.7: Effect of Indiscriminate False Updates on UCT+RAVE in SOS(10). False RAVE updates are performed randomly on all moves.

Whereas the previous model injects noise into the RAVE values, causing the search to proceed more randomly, this model manipulates the RAVE values in a more directed manner, focusing the search towards or away from specific moves.

Lowering Strong Values

One scenario described at the start of Section 5.6 is the case of RAVE presenting a false negative. In this scenario, RAVE statistics make a strong move look weak. We simulate this effect in these experiments by manipulating the value of the strongest move, move 9 in SOS(10). If the RAVE update for the optimal move would be the value of a win, with probability μ, we update it as a loss instead. In this context, the optimal move refers only to the optimal *first* move at the start of the game. Results are shown in Figure 5.9.

At all values of μ greater than 0, FUEGO-SOS performs significantly worse than in the indiscriminate false update case. To analyze the behaviour of RAVE in this experiment, let us focus on the setting where $\mu = 0.2$. At this setting, FUEGO-SOS performs significantly worse than where $\mu = 0$, but still achieves optimal play. After 2^{13} simulations, the UCT+RAVE algorithm seems to overcome the effects of the false update. The performance recovery is significant and drastic. Figures 5.10 and 5.11 can be used to explain this effect. The estimated value for the optimal move, move 9, is not the highest option prior to the 2^{13} point. However, the point at which this trend changes coincides with the point at which the move count of the optimal move begins to rise past the previous leader, move 8. At this point, move 9 was found to be at least as good as move 8 and the

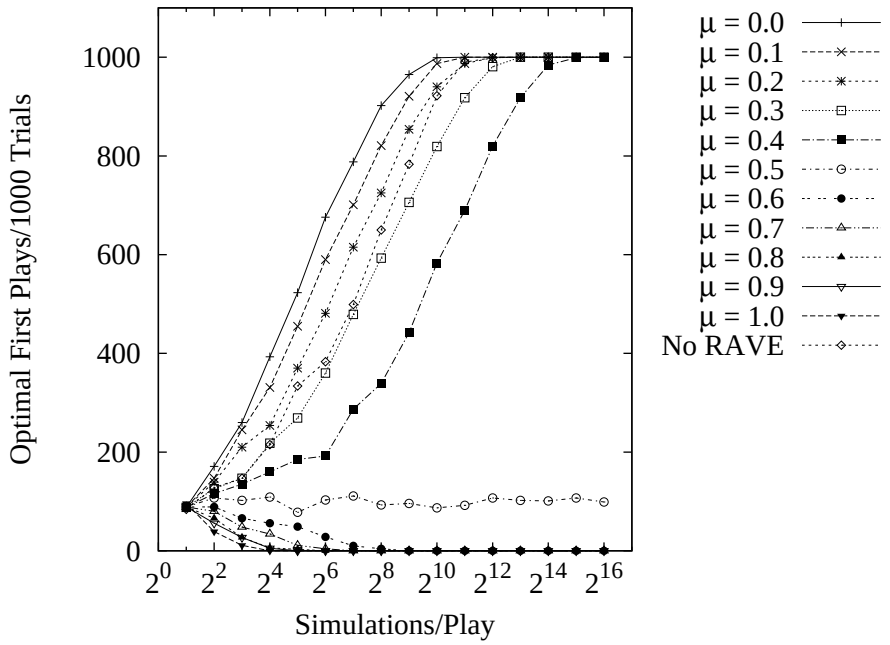

Figure 5.8: Effect of Indiscriminate False Updates for RAVE-only on SOS(10). False RAVE updates are performed randomly with probability μ.

move 9 branch of the game tree was more thoroughly explored as games were won more frequently than previously. As the move count increases, the UCT value gains dominance, and the effect of the false RAVE value is diminished. This is a result of the weighting function introduced in Section 2.5. This idealised scenario shows the effect in its clearest form, but is unrealistic since in practice the RAVE updates of all moves in a simulation will be affected in the same way.

To more accurately mimic a real-game scenario, we investigate the effect of forcefully lowering the RAVE value of the optimal move by generalizing the false RAVE update to all moves that were played in the same simulated game. Performance results after applying this change are presented in Figure 5.12. FUEGO-SOS performs better in this scenario than in the previous, more similar to the indiscriminate false updates case. However, the sudden rebound effect seen in Figure 5.9 is seen here as well. As soon as the estimated move value of the optimal move becomes the highest amongst all alternatives, the algorithm appears to sample the optimal move almost exclusively, resulting in consistent optimal play. Figures 5.13 and 5.14 show how the estimated values and move simulation counts evolve. Again increased sampling correlates with the change in estimated value. This is expected to result from the change in weights as the weight of the RAVE value diminishes. The example $\mu = 0.6$ was chosen as it was the highest μ value that converged to optimal play within the tested number of simulations.

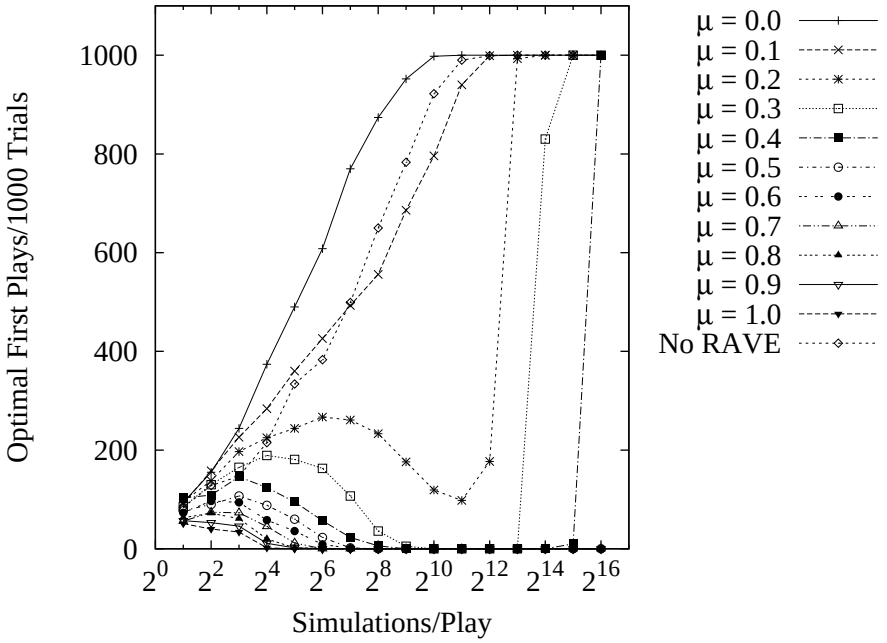

Figure 5.9: Effect of False Updates where winning RAVE updates of the best move on SOS(10) are inverted with probability μ.

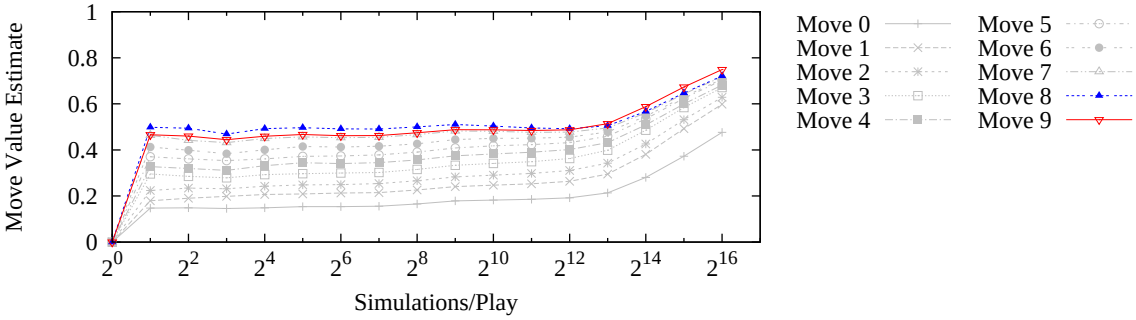

Figure 5.10: Value estimates(UCT+RAVE) from the experiment in Figure 5.9 for $\mu = 0.2$.

Raising Weak Values

The other scenario mentioned at the start of Section 5.6 involved RAVE resulting in false positives in its search. In this situation RAVE overestimates the value of a move, $Move_a$, wrongly believing that $Move_a$ is a strong candidate for best move. To simulate this problem, we manipulate RAVE in experiments similar to Section 5.6.2. However, simulation losses are given the value of a win during the RAVE update with probability μ, since we are bolstering the value of the target move instead. Two different moves were investigated in this series of experiments: the worst move and the second-best move, move 0 and move 8 respectively in SOS(10). The initial hypothesis proposed that bias towards the second-best move should mislead the search more significantly than bias towards the worst move, as distinguishing between the best and second best moves should be more difficult for

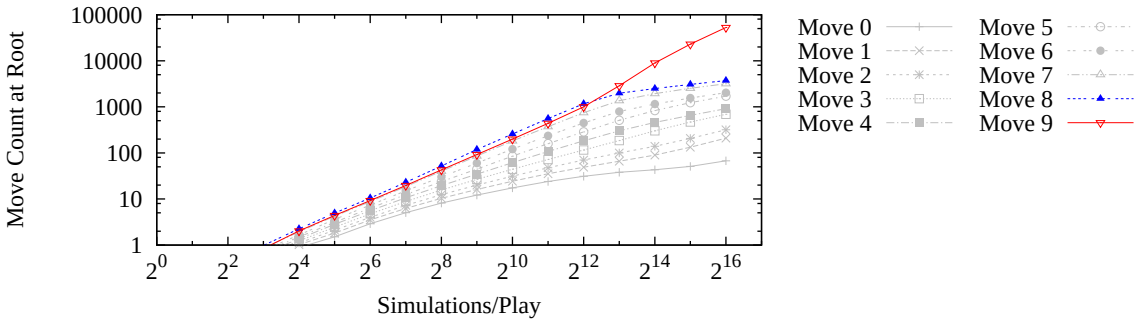

Figure 5.11: Move Counts at the root node from the experiment in Figure 5.9 for $\mu = 0.2$.

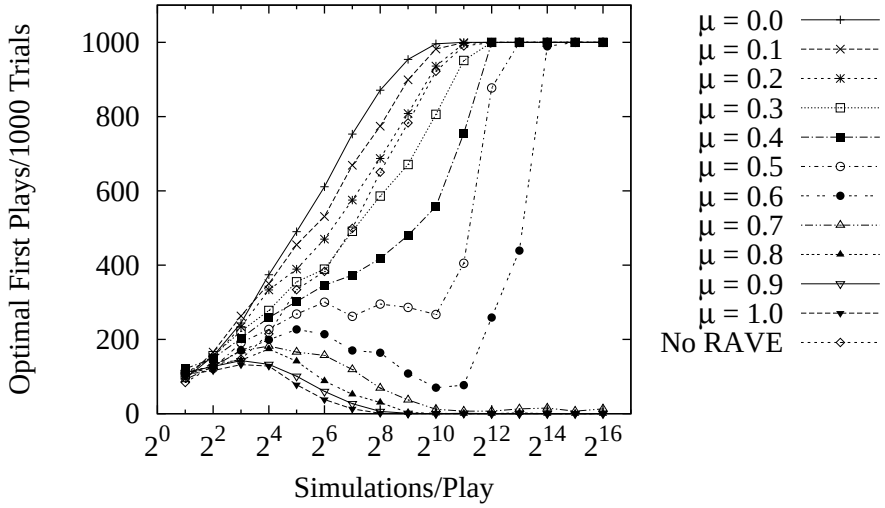

Figure 5.12: Effect of False Updates where winning RAVE updates of the best move on SOS(10) are inverted with probability μ. Inverted updates are also applied to all other moves played by the player that chose the best move.

UCT. Results from these experiments are shown in Figures 5.15 to 5.20.

When the value of the worst move was inflated, FUEGO-SOS performed better than in the indiscriminate false update case. Although the false updates encourage simulations related to the worst move in this scenario, simulations related to the optimal move are not discouraged. Thus, once the worst move is refuted, the algorithm quickly switches to the next best choice: the optimal move. The value estimates and move counts of the $\mu = 1.0$ setting in Figures 5.16 and 5.17 support this hypothesis. The move value of move 9 remains consistently above all other moves save move 0. When the value of move 0 drops below a certain point, 0.37 in this example, FUEGO-SOS begins to simulate move 9 much more frequently.

When the value of the second-best move was inflated, the behaviour of FUEGO-SOS became similar to inverting the updates of the best move in Figure 5.9. However, convergence rates were better. We again investigate by closer examination of the move counts and value estimates of the $\mu = 1.0$ setting in Figures 5.19 and 5.20. As with inflating the value of the worst move, the value

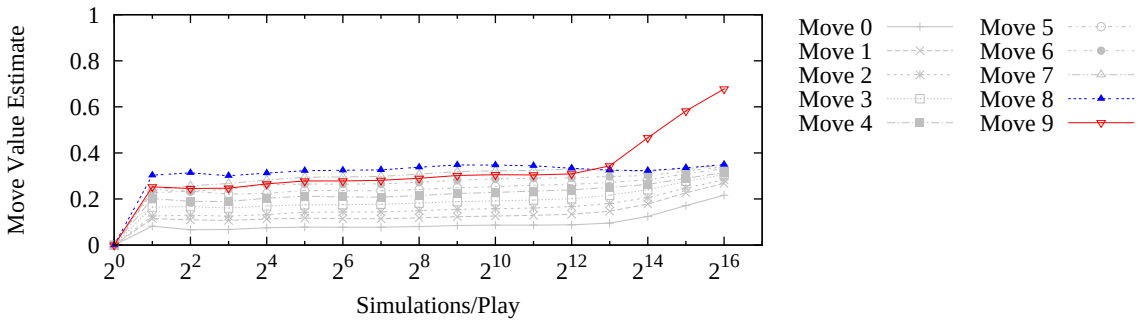

Figure 5.13: Value estimates(UCT+RAVE) from the experiment in Figure 5.12 for $\mu = 0.6$.

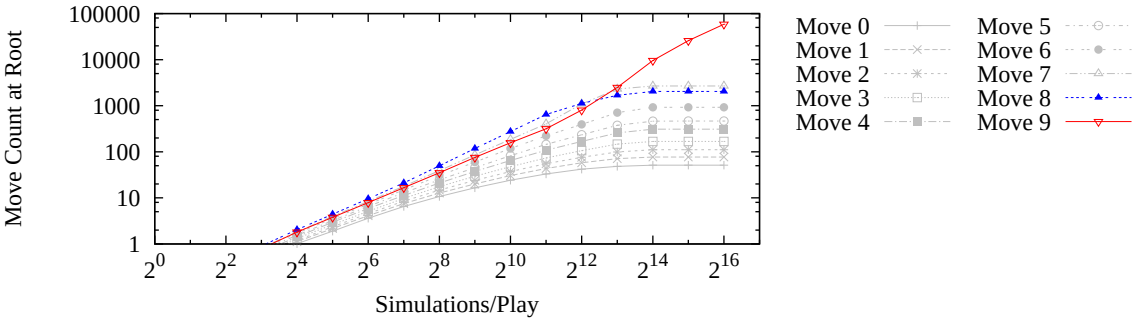

Figure 5.14: Move Counts at the root node from the experiment in Figure 5.12 for $\mu = 0.6$.

of the optimal move remains consistently high, which allows FUEGO-SOS to quickly converge to optimality once the false move has been refuted.

We also investigate the effect of generalizing the false RAVE update to other moves played with the target move. One would expect the effect of generalizing the false update for weak moves to be less disastrous than the results from Section 5.6.2, as the optimal move is not negatively manipulated. Figures 5.21 and 5.24 show the results of generalizing the false update value to other moves. Figures 5.22 and 5.23 show the moves values and move counts for Figure 5.21. The same μ values were used for our experiments for shared false updates. When this shared update was applied to the inflated worst move scenario, the worst move dominated the best move in move value until 2^9 simulations were reached where the roles reversed. For simulations from 2^4 on, the values of the best and worst move remained within 0.01 of each other. This is not the case when the second-best move becomes the target of the shared false update. A sizeable gap between the inflated second-best move and the best move causes convergence to be delayed until the 2^{16} datapoint. As one would expect, it appears that manipulating the move value of the second-best move is far more detrimental to MCTS. The move values shown in Figure 5.25 show a considerable gap between the values of the inflated move 8 and the optimal move, move 9. The generalized false update causes FUEGO-SOS to simulate move 7 before further considering move 9, as seen in the move counts plotted in Figure 5.26. The large time investment spent in simulating the inflated decoy moves greatly delays convergence.

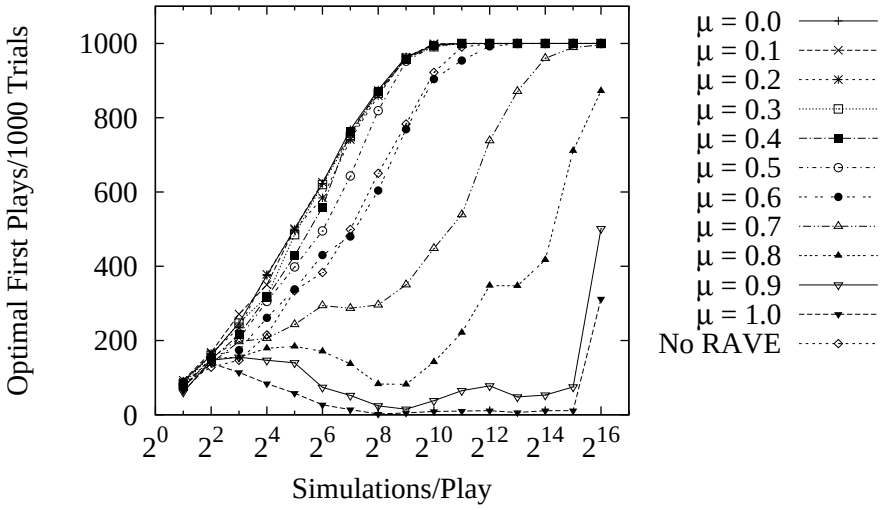

Figure 5.15: Effect of False Updates where the losing RAVE updates of the worst move on SOS(10) are inverted with probability μ.

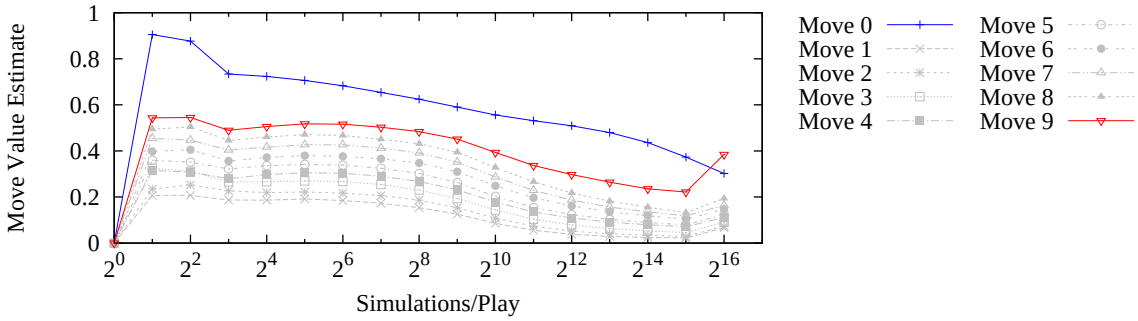

Figure 5.16: Value estimates(UCT+RAVE) from the experiment in Figure 5.15 for $\mu = 1.0$.

RAVE-max: Towards a more robust RAVE

In the false update experiments, the measured mean $\bar{X}_j$ of the optimal move is often high even though its RAVE value $\bar{Y}_j$ is low. The following experiments test *RAVE-max*, the simple modification of replacing $\bar{Y}_j$ by $\max(\bar{Y}_j, \bar{X}_j)$ in Equation 2.2 from Section 2.5.

Figure 5.28 shows the performance difference between $\max(\bar{Y}_j, \bar{X}_j)$ and $\bar{Y}_j$ for indiscriminate false updates. Initial performance is worse for $\mu \leq 0.3$, but significantly better for almost all other cases. The weaker result at higher simulation counts for one case, $\mu = 0.6$ is surprising. The performance drop for small μ and low number of simulations is natural since RAVE data is relatively high quality and the sample size for the mean is so small.

Using RAVE-max on the selective false update problems from Section 5.6.2, we see performance gains in both the individual move experiments and the shared false update experiments. Figures 5.29 and 5.30 show the performance differences. The initial performance loss is seen at $\mu = 0.0$ in the individual case and for $\mu \leq 0.2$ in the shared case in this scenario. However, in general, performance

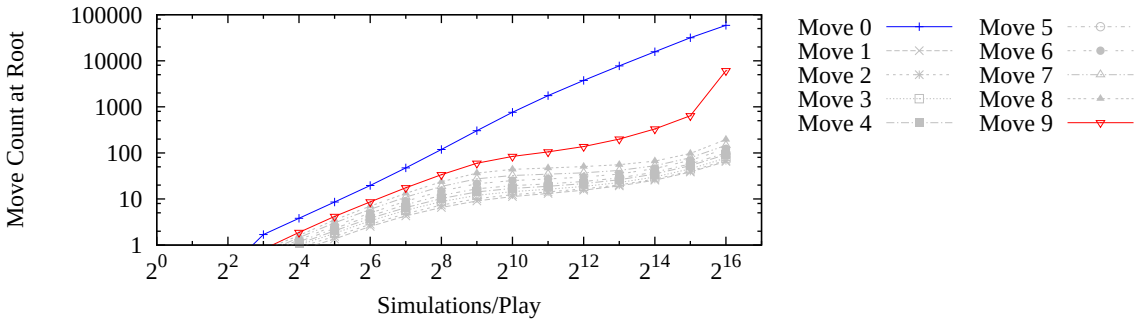

Figure 5.17: Move Counts at the root node from the experiment in Figure 5.15 for $\mu = 1.0$.

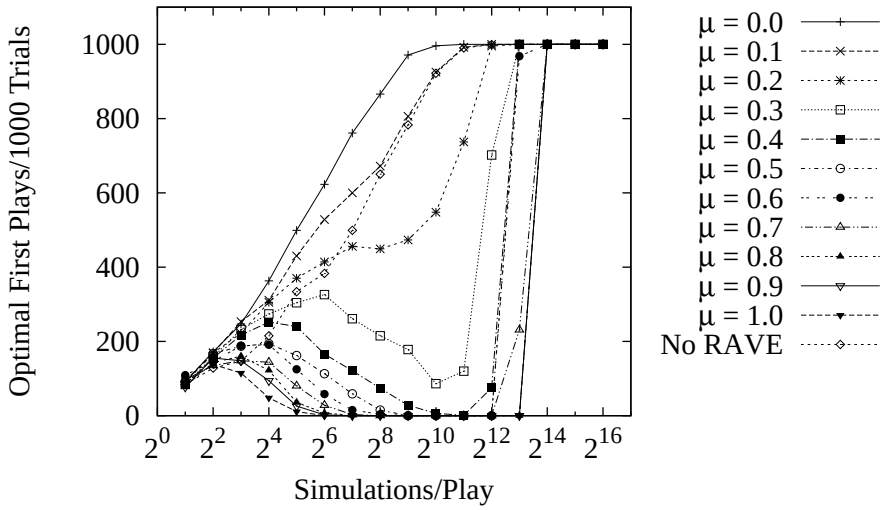

Figure 5.18: Effect of False Updates where the losing RAVE updates of the second-best move on SOS(10) are inverted with probability μ.

improves significantly in these tests. This result is to be expected as taking the max should negate any undermining caused by RAVE.

For the boosting RAVE value problems from Section 5.6.2, RAVE-max showed overall positive results only for individually boosting the worst move, with slightly negative results for small numbers of simulations. RAVE-max was also weaker when $\mu \leq 0.6$, but positive results for all $\mu > 0.6$. This result is surprising since this scenario is the opposite of the reducing RAVE problems. Using RAVE-max on the other boosting settings resulted in performance similar to the RAVE, but with performance losses during low simulations counts. The performance differential graphs for these settings are presented in Figures 5.31, 5.32, 5.33 and 5.34. RAVE-max improves performance in situations where RAVE values of moves are underestimated, but does not help against overestimates. Even so, RAVE-max performs at least as well as RAVE on overestimation situations. A more robust solution needs to be found to handle both classes of RAVE problems.

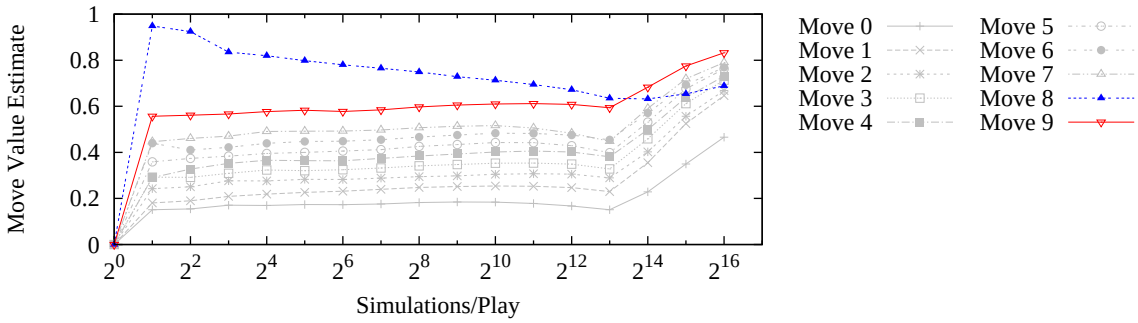

Figure 5.19: Value estimates(UCT+RAVE) from the experiment in Figure 5.18 for $\mu = 1.0$.

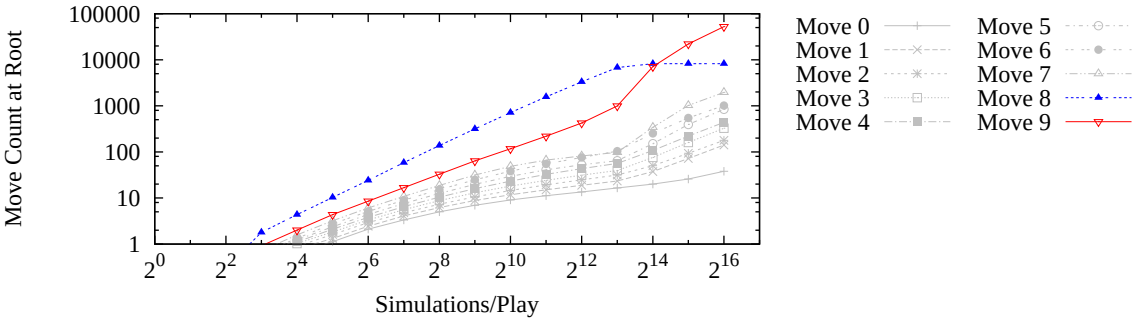

Figure 5.20: Move Counts at the root node from the experiment in Figure 5.18 for $\mu = 1.0$.

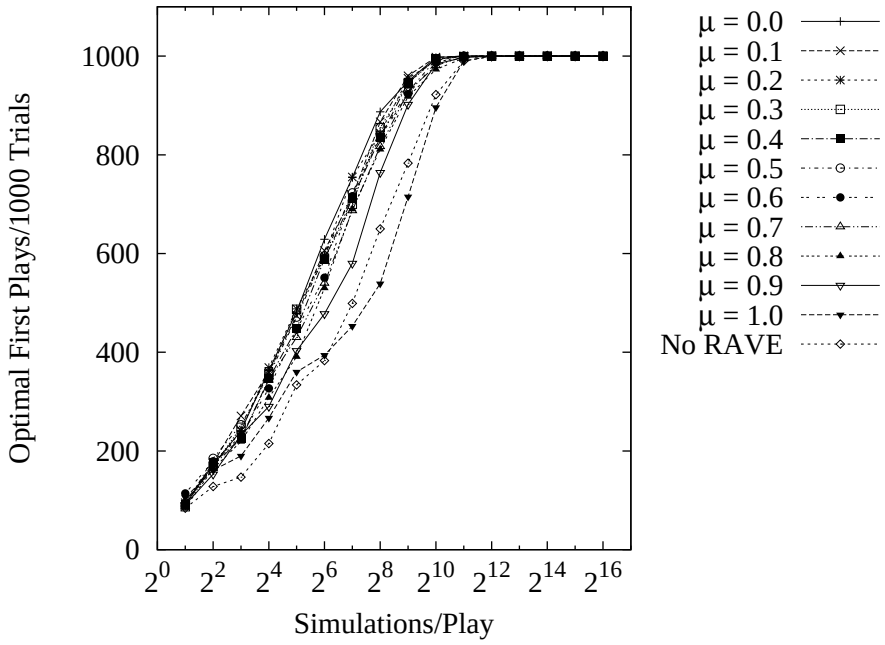

Figure 5.21: Effect of False Updates where the losing RAVE updates of the worst move on SOS(10) are inverted with probability μ. Inverted updates are also applied to all other moves played by the player that chose the worst move.

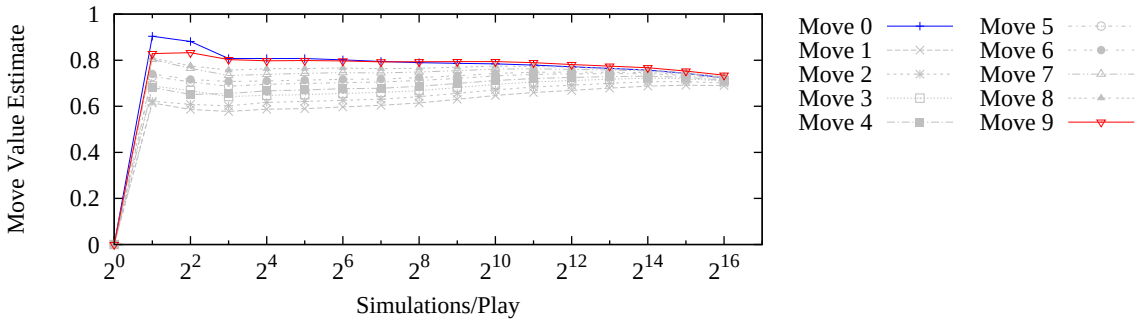

Figure 5.22: Value estimates(UCT+RAVE) from the experiment in Figure 5.21 for $\mu = 1.0$.

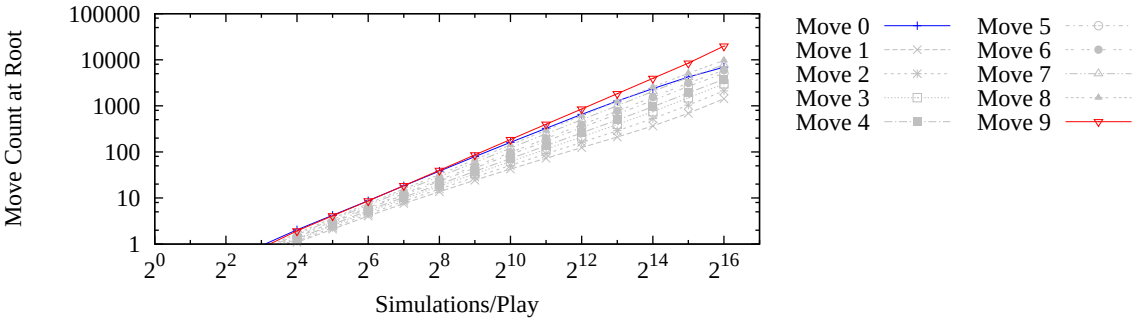

Figure 5.23: Move Counts at the root node from the experiment in Figure 5.21 for $\mu = 1.0$.

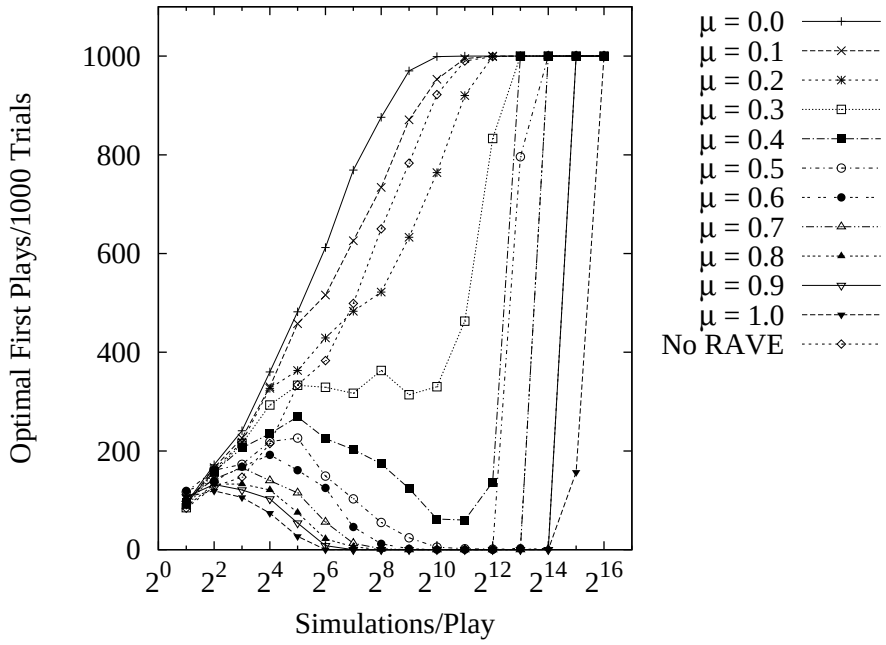

Figure 5.24: Effect of False Updates where the losing RAVE updates of the second-best move on SOS(10) are inverted with probability μ. Inverted updates are also applied to all other moves played by the player that chose the second-best move.

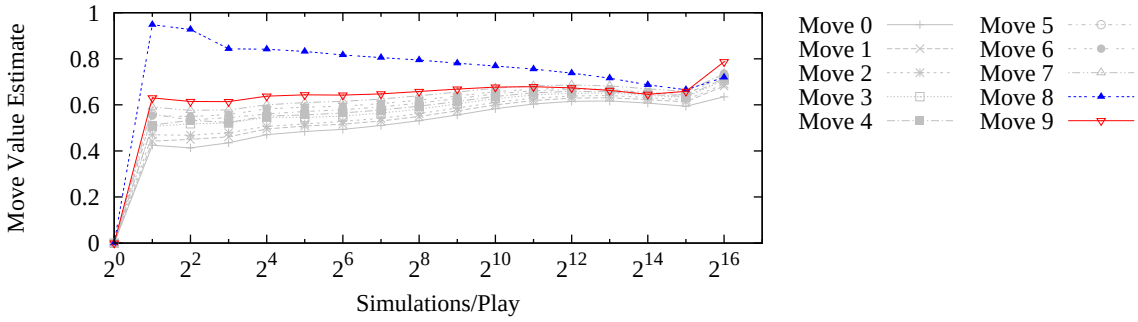

Figure 5.25: Value estimates(UCT+RAVE) from the experiment in Figure 5.24 for $\mu = 1.0$.

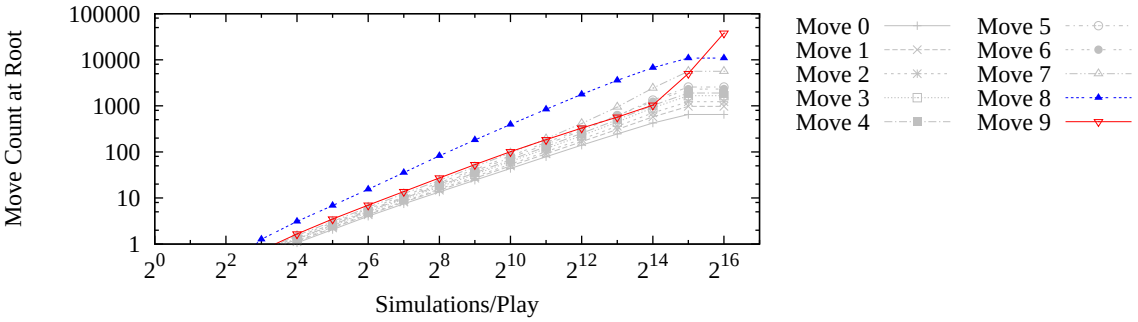

Figure 5.26: Move Counts at the root node from the experiment in Figure 5.24 for $\mu = 1.0$.

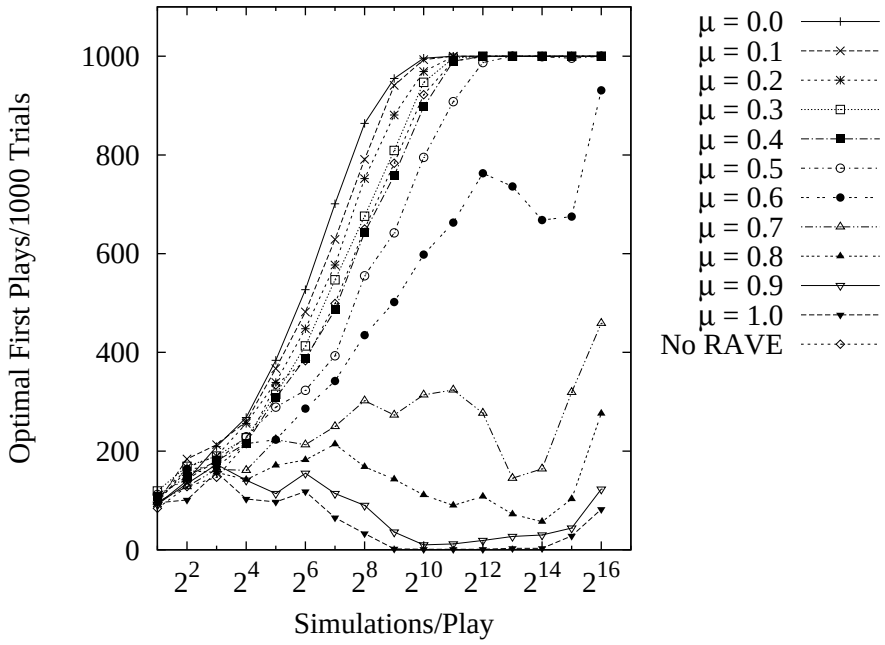

Figure 5.27: Results for experiments using Indiscriminate False Updates with RAVE-max.

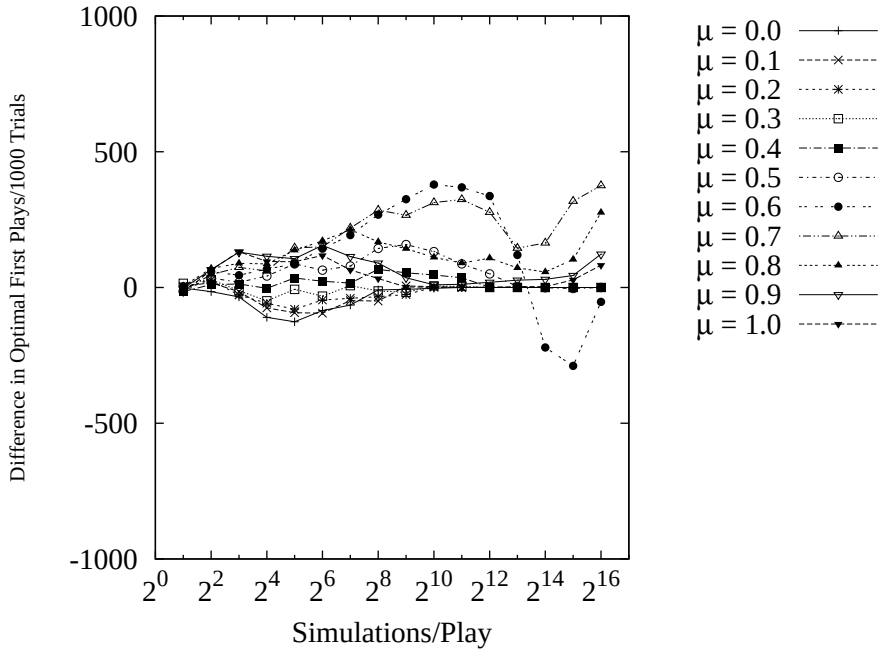

Figure 5.28: Performance differences for Indiscriminate False Updates. Compares RAVE-max to RAVE.

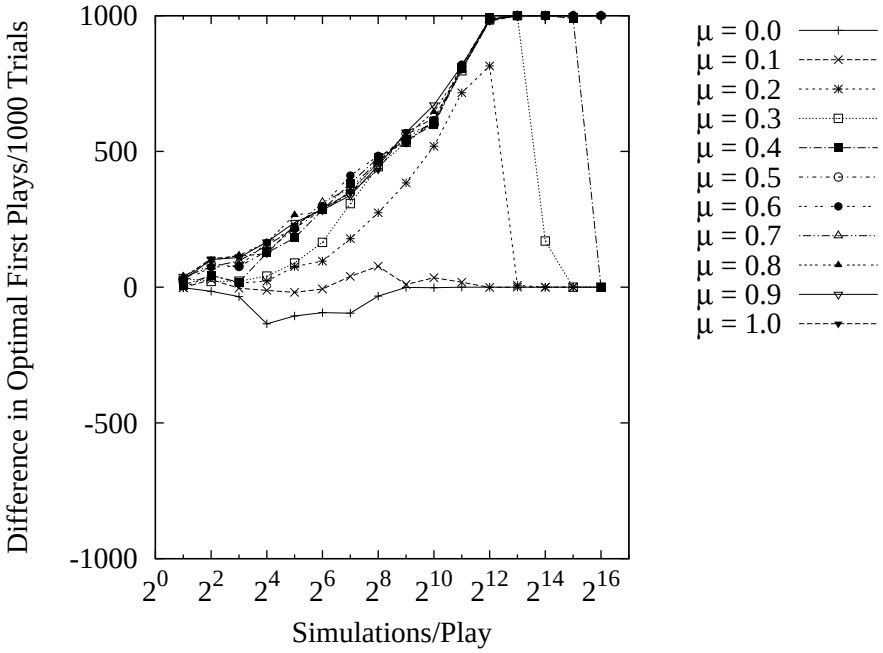

Figure 5.29: Performance differences for False Updates lowering the RAVE value of best move. Compares RAVE-max to RAVE.

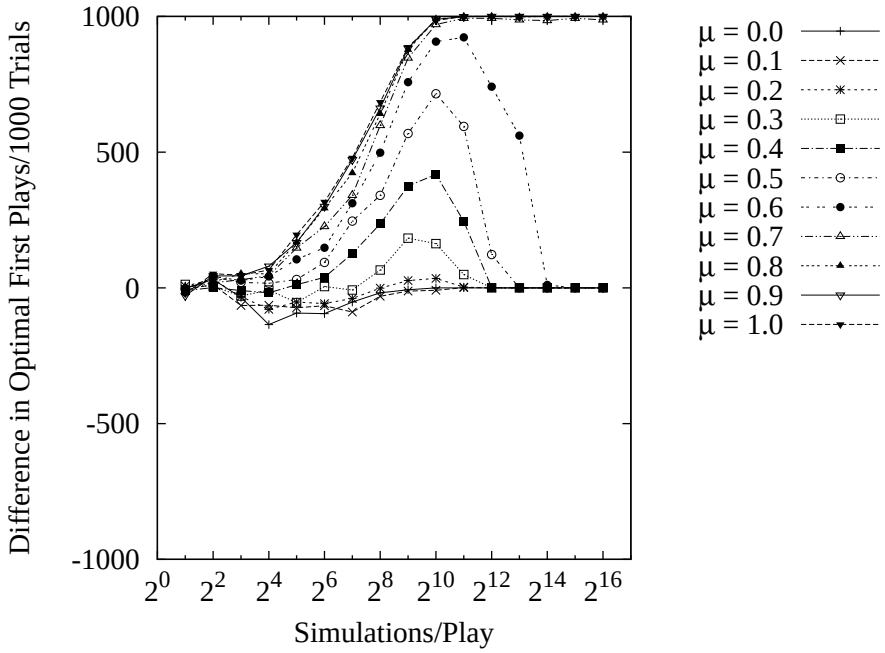

Figure 5.30: Performance differences for False Updates lowering the RAVE value of best move and other moves played with it. Compares RAVE-max to RAVE.

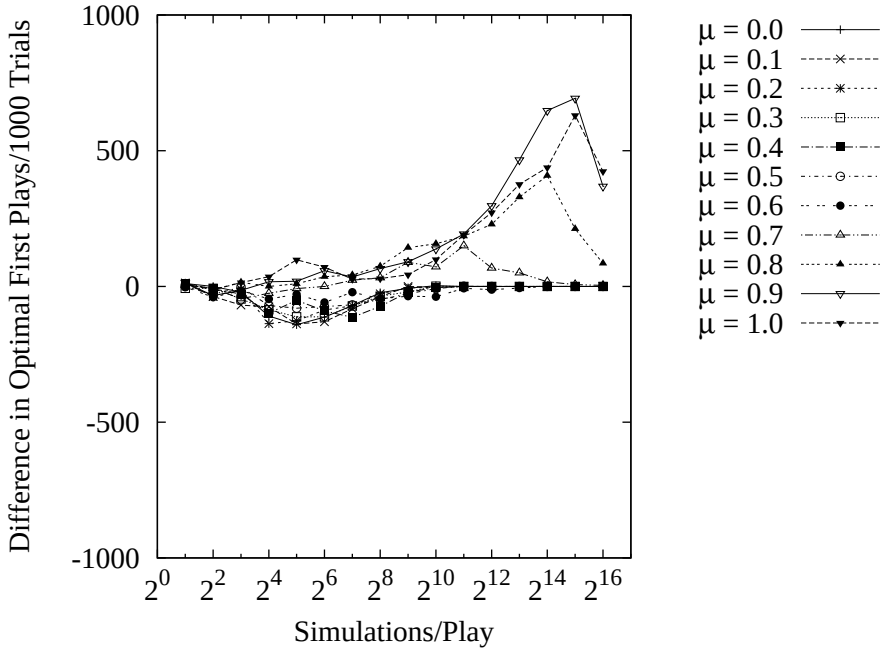

Figure 5.31: Performance differences for False Updates raising the RAVE value of worst move. Compares RAVE-max to RAVE.

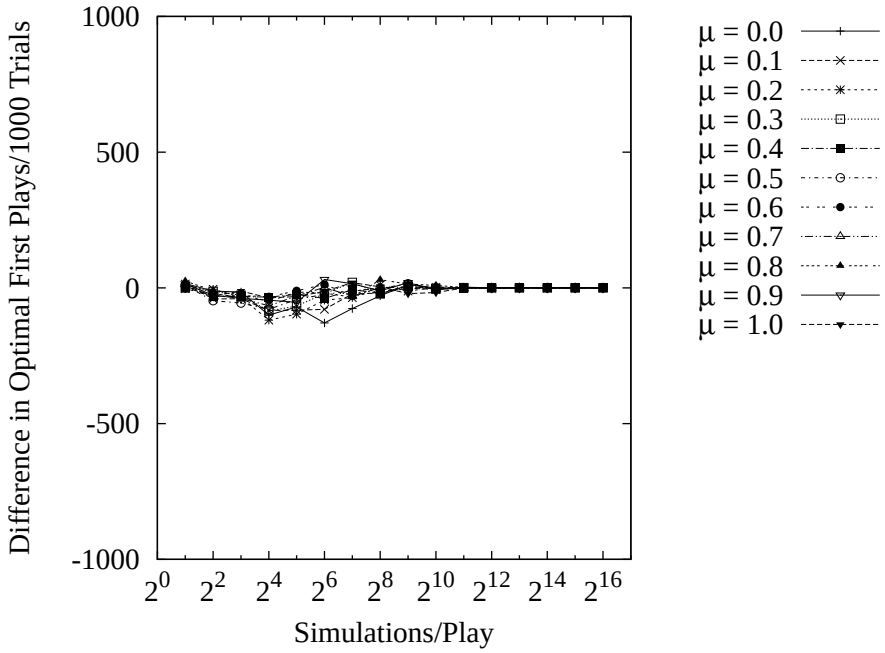

Figure 5.32: Performance differences for False Updates raising the RAVE value of worst move and other moves played with it. Compares RAVE-max to RAVE.

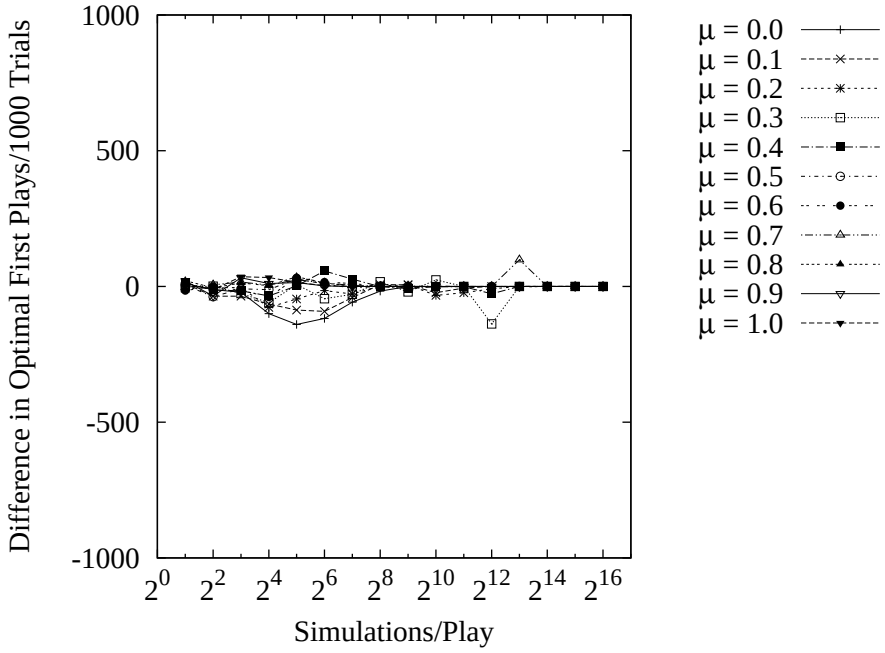

Figure 5.33: Performance differences for False Updates raising the RAVE value of second-best move. Compares RAVE-max to RAVE.

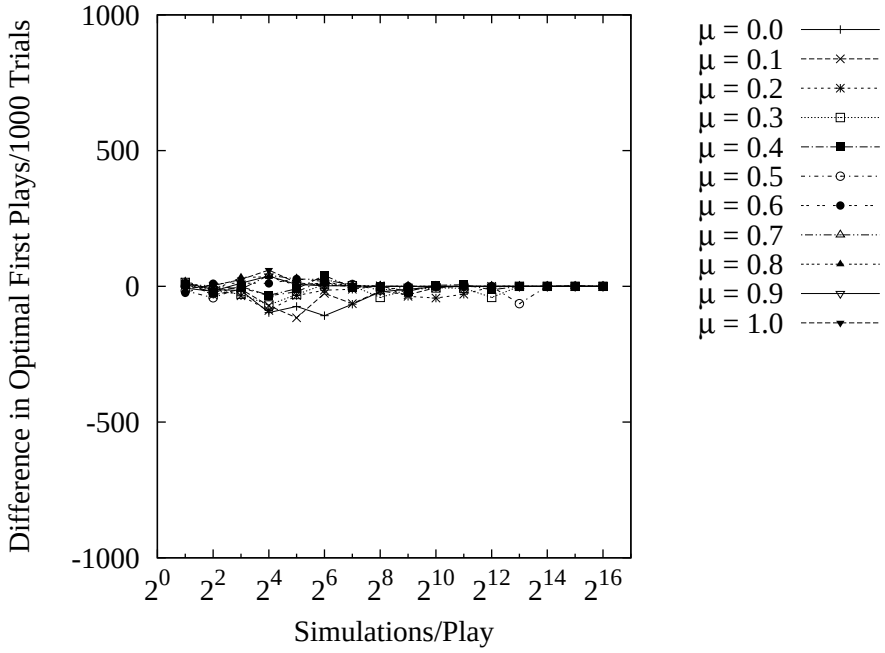

Figure 5.34: Performance differences for False Updates raising the RAVE value of second-best move and other moves played with it. Compares RAVE-max to RAVE.

Chapter 6

Conclusions and Future Work

6.1 Thesis Conclusions

The Sum of Switches game provides a simple, well-controlled environment where behaviour is easily measured. In this environment, our experiments studied UCT and two common enhancements, RAVE and Score Bonus. Score Bonus did not produce favourable results in SOS, but had a positive effect in Go. This discrepancy needs further study.

The RAVE experiments show significantly better performance than plain UCT, even with distorted RAVE updates. In games where the values of moves do not change over the course of a game, RAVE provides a much stronger estimate than the mean value. Furthermore, the indiscriminate false update experiments suggest that the RAVE heuristic is robust against unbiased noise and performs well even with a fair level of error.

When false updates were performed in a more realistic manner in the selective false update experiments, RAVE continued to demonstrate resilience. By watching the development of move values and move counts throughout the search, we inferred some of the mechanisms behind the recovery process of rebounding from a poor RAVE heuristic. From this, we produced RAVE-max that attempts to hasten the rebound process. Although RAVE-max appears successful in overcoming underestimated RAVE values, it does not improve situations where RAVE is overestimated.

6.2 Future Work

Future work related to this thesis can be pursued in the following areas: SOS, Score Bonus, RAVE-max, and RAVE error models. One direction of research is studying the effect of improved, non-uniform random playout policies in SOS. The Score Bonus enhancement is another topic that can be further researched. Interesting topics that deserve investigation include why Score Bonus does not improve performance on SOS and a general understanding of the types of environments where Score Bonus does improve performance. One possible reason why Score Bonus was ineffective on SOS was the accuracy of the estimated values. Score Bonus may be more useful in larger games where value estimations are less accurate. Future work related to RAVE-max should look at handling the

overestimation cases in RAVE errors. Of course, it may be more important to measure the actual effectiveness of RAVE-max in popular classical games. Although our error models are designed with the intent to model real situations, testing in real games is necessary to verify our results. Furthermore, more accurate error models may be found to better model real situations.

Bibliography

[1] L.V. Allis. *Searching for solutions in games and artificial intelligence*. PhD thesis, University of Limburg, 1994.

[2] V. V. Anshelevich. A hierarchical approach to computer hex. *Artificial Intelligence*, 134(1-2):101 – 120, 2002.

[3] B. Arneson, R. Hayward, and P. Henderson. Wolve 2008 wins Hex Tournament. *ICGA Journal*, 32(1):49–53, March 2009.

[4] B. Bouzy and T. Cazenave. Computer Go: an AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[5] B. Bouzy and B. Helmstetter. Monte-Carlo Go developments. In J. van den Herik, H. Iida, and E. Heinz, editors, *Advances in Computer Games. Many Games, Many Challenges. Proceedings of the ICGA / IFIP SG16 10th Advances in Computer Games Conference*, pages 159 – 174. Kluwer Academic Publishers, 2004.

[6] B. Brügmann. Monte Carlo Go, March 1993. Unpublished manuscript, `http://www.cgl.ucsf.edu/go/Programs/Gobble.html`.

[7] M. Buro. Toward opening book learning. *ICCA Journal*, 22(2):98–102, 1999.

[8] M. Campbell, A.J. Hoane, and F. Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[9] G. Chaslot, M. Winands, J. Uiterwijk, J. van den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008.

[10] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. *Lecture Notes in Computer Science*, 4630:72–83, 2007.

[11] R. Coulom. Whole-history rating: A bayesian rating system for players of time-varying strength. In van den Herik et al. [42], pages 113–124.

[12] F. de Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. Bandit-based optimization on graphs with application to library performance tuning. In A. P. Danyluk, L. Bottou, and M. L. Littman, editors, *ICML*, volume 382 of *ACM International Conference Proceeding Series*, page 92. ACM, 2009.

[13] M. Enzenberger and M. Müller. Fuego, 2008. `http://fuego.sf.net/` Retrieved December 22, 2008.

[14] M. Enzenberger and M. Müller. Fuego – an open-source framework for board games and Go engine based on Monte-Carlo tree search. Technical Report TR 09-08, Dept. of Computing Science. University of Alberta, Edmonton, Alberta, Canada, 2009. `http://www.cs.ualberta.ca/research/techreports/2009/TR09-08.php`.

[15] T. S. Ferguson. Game Theory, 2004. Online text, `http://www.math.ucla.edu/~tom/Game_Theory/Contents.html`.

[16] H. Finnsson and Y. Björnsson. Simulation-based approach to General Game Playing. In Fox and Gomes [17], pages 259–264.

[17] D. Fox and C. Gomes, editors. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*. AAAI Press, 2008.

[18] T. Furtak and M. Buro. Minimum Proof Graphs and Fastest-Cut First Search Heuristics. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 492–498, Pasadena USA, 2009.

[19] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In Z. Ghahramani, editor, *ICML*, volume 227 of *ACM International Conference Proceeding Series*, pages 273–280. ACM, 2007.

[20] S. Gelly and D. Silver. Achieving Master Level Play in 9 x 9 Computer Go. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008*, pages 1537–1540, 2008.

[21] S. Gelly and Y. Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, Canada, 12 2006.

[22] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo Go, 2006. Technical Report RR-6062, INRIA, France.

[23] H. Gintis. *Game Theory Evolving: A Problem-Centered Introduction to Modeling Strategic Interaction (Second Edition)*. Princeton University Press, 2nd edition, February 2009.

[24] R. P. Jones and D. J. Thuente. The role of simulation in developing game playing strategies. In *ANSS '90: Proceedings of the 23rd annual symposium on Simulation*, pages 89–97, Piscataway, NJ, USA, 1990. IEEE Press.

[25] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *Proceedings of 17th European Conference on Machine Learning, ECML 2006*, pages 282–293, 2006.

[26] R. J. Lorentz. Amazons discover Monte-Carlo. In van den Herik et al. [42], pages 13–24.

[27] N. Metropolis. The beginning of the Monte Carlo method. *Los Alamos Science*, 15:125–130, 1987.

[28] M. Müller. Computer Go. *Artificial Intelligence*, 134(1-2):145–179, 2002.

[29] M. Müller and T. Tegos. Experiments in computer Amazons. In R.J. Nowakowski, editor, *More Games of No Chance*, pages 243–257. Cambridge University Press, 2002.

[30] X. Niu and M. Müller. An open boundary safety-of-territory solver for the game of Go. In J. van den Herik, P. Ciancarini, and H. Donkers, editors, *Computer and Games. 5th International Conference*, volume 4630 of *Lecture Notes in Computer Science*, pages 37–49, 2006.

[31] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[32] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. Chaslot, and J. W. H. M. Uiterwijk. Single-Player Monte-Carlo Tree Search. In van den Herik et al. [42], pages 1–12.

[33] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(11):1203–1212, 1989.

[34] J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518, 2007.

[35] J. Schaeffer, J.C. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992.

[36] A. Scheucher and H. Kaindl. The reason for the benefits of minimax search. In *Proceedings of the 11th international joint conference on Artificial intelligence-Volume 1*, pages 322–327, 1989.

[37] D. Silver. *Reinforcement Learning and Simulation-Based Search*. PhD thesis, University of Alberta, 2009.

[38] S. J. J. Smith and D. S. Nau. An analysis of forward pruning. In *AAAI'94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)*, pages 1386–1391, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[39] F. Teytaud and O. Teytaud. Creating an Upper-Confidence-Tree program for Havannah. In *Advances in Computer Games 12*, Pamplona Espagne, 2009. To appear in LNCS.

[40] D. Tom and M. Müller. A Study of UCT and its Enhancements. In *Advances in Computer Games 12*, 2009. To appear in LNCS.

[41] D. Tom and M. Müller. Towards a Robust RAVE Heuristic. 2010. Submitted to the International Conference on Computers and Games 2010.

[42] H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors. *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings*, volume 5131 of *Lecture Notes in Computer Science*. Springer, 2008.

[43] M.H.M. Winands. Analysis and implementation of Lines of Action. *Master's thesis, Department of Computer Science, Universiteit Maastricht*, 2000.

[44] M.H.M. Winands and Y. Björnsson. Evaluation Function Based Monte-Carlo LOA. In *Advances in Computer Games 12*, 2009. To appear in LNCS.