A Study of UCT and its Enhancements in an Artificial Game

David Tom and Martin Müller

Department of Computing Science, University of Alberta, Edmonton, Canada, T6G 2E8
{dtom, mmueller}@cs.ualberta.ca

Abstract. Monte-Carlo tree search, especially the UCT algorithm and its enhancements, have become extremely popular. Because of the importance of this family of algorithms, a deeper understanding of when and how the different enhancements work is desirable. To avoid the hard to analyze intricacies of tournament-level programs in complex games, this work focuses on a simple abstract game, which is designed to be ideal for history-based heuristics such as RAVE. Experiments show the influence of game complexity and of enhancements on the performance of Monte-Carlo Tree Search.

1 Introduction

Monte Carlo Tree Search (MCTS), especially in form of the UCT algorithm [8], 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 [7] and General Game-Playing [5]. MCTS-based programs are also on par with the best traditional programs in Hex and Amazons [9].

Part of the success of MCTS is due to its enhancements. Methods inspired by Schaeffer's history heuristic [10] include *All-Moves-As-First* (AMAF) [2] and Rapid Action Value Estimation (RAVE) [6]. Whereas the value of a move is typically based on 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 [6]. Methods such as progressive pruning [1] focus MCTS on stronger-looking candidate branches.

While the game-independent algorithms above can be used with minor variations across different games, typical tournament-level programs contain a large number of game-specific enhancements as well, such as opening books and specialized playout policies. Further examples are patterns [7, 3] and tactical subgoal solvers in Go, and virtual connections in Hex.

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

1.1 Research Questions

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

- How does the performance of an algorithm vary with the complexity and type of game that is played?
- What are the conditions on a game under which a specific enhancement works? How much does it improve MCTS in the best case?
- How should a general framework for Monte-Carlo Tree Search be designed, and how can it then be adapted to a specific game?

Some of these questions are addressed in practice by the Fuego system [4], an open-source library for games which includes the MCTS engine used for the experiments in this paper. 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. Ideally, such games should allow deeper study of the core algorithms while avoiding layers of game-specific complexity in the analysis.

In this paper, a simple artificial game, called Sum of Switches (SOS), is used for an experimental study of MCTS algorithms, in particular, as a close to ideal scenario for the RAVE heuristic. Section 2 introduces and motivates the SOS game model, and discusses related work on analysis of MCTS. Section 3 briefly summarizes relevant parts of the Fuego framework used in the experiments. Section 4 describes our experiments. Sections 5 and 6 conclude with a discussion of our results and ideas for future work.

2 The Sum of Switches Game

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 first player's picks, $s_1 = p_{1,1} + \ldots + p_{1,n/2}$, and s_2 the sum of second player's picks, $s_2 = p_{2,1} + \ldots + p_{2,n/2}$. Scoring is similar to the game of Go. The *komi* k is set to the perfect play outcome, $k = (n-1) - (n-2) + \ldots = \lfloor n/2 \rfloor$. 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. 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 X_i, a random variable. 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 X_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.

2.1 Related Work and Motivation for SOS

The original UCT paper [8] contains an experiment showing the performance of UCT on the artificial P-game tree model [11]. Each edge representing a move is associated with a random number from a specified range. The value of a leaf node is the sum of the edge values along the path from the root. The value of edges corresponding to opponent moves is negated.

In the SOS model, the value of a move is independent of when and by which player it is chosen. This should represent a best-case scenario for history-based heuristics such as RAVE.

The RAVE heuristic is a frequently used enhancement for MCTS. In contrast to basic Monte-Carlo tree search, it collects statistics over *all* moves played in a simulation. In a game such as SOS, that extra information should be of high quality since moves have the same value independent of when they are played.

In the original work on RAVE [6], Gelly and Silver analyze its performance in the context of computer Go. The weights for the RAVE heuristic were chosen empirically to work well in Go. Empirically, RAVE is shown to have very strong overall performance in Go. However, it causes occasional blunders by introducing a strong bias against the correct move. For example, if a move is very good right now, but very bad if played at any time later in a simulation, RAVE updates would be misleading. Such misleading biases do not exist in the case of SOS.

3 The Fuego Framework and its MCTS implementation

The experiments with SOS use the *Fuego* framework [4], which includes the computer Go program with the same name. One component of the Fuego framework is the game-independent SmartGame library: a set of tools to handle game play, 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. The UCT and RAVE engines are used in the SOS experiments with no modifications. No experiments on utilizing prior knowledge are presented in this paper.

The UCT engine uses the basic UCT formula, with user-defined parameters controlling the UCT behaviour. 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 for the purposes of SOS, we chose to keep it at the default value of 0.7.

When RAVE is active, the value of the estimate 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 a little different from the one originally proposed in [6], 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:

$$W_j = \frac{\beta_j w_f w_i}{w_f + w_i \beta_j}$$

the RaveCount β_j represents the number of rave updates of move j. w_i and w_f stand for RaveWeightInitial and RaveWeightFinal; 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 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. In practice, the values of RaveWeightInitial is usually kept at the default value of 1.0, and a suitable RaveWeightFinal is found experimentally. RaveWeightInitial is kept at 1.0 as we do not make any assumptions about the accuracy of early RAVE and UCT estimates. The weight W_j is used in the UCT formula in the following manner [4]:

$$MoveValue(j) = \frac{T_j(\alpha)}{T_j(\alpha) + W_j}\bar{X}_j + \frac{W_j}{T_j(\alpha) + W_j}\bar{Y}_j + c\sqrt{\frac{\log \alpha}{T_j(\alpha) + 1}}$$

α represents the number of times the parent node was visited. $\bar{X}_j$ denotes the average reward and $\bar{Y}_j$ the RAVE value of move j. The c term is a constant bias term set by the user and $T_j(\alpha)$ is the MoveCount, the number of times move j has been played at the parent node. Adding 1 to $T_j(\alpha)$ in the bias term avoids a division by 0 in case move j has a RAVE value but $T_j(\alpha) = 0$. In MCTS, the game tree is grown incrementally. In the SmartGame library unexpanded nodes are assigned a *FirstPlayUrgency* value. Large values cause the program to prioritize exploration whereas small values encourage exploitation. The default value of 10000 is used in the experiments, which gives high priority to unexpanded nodes [8].

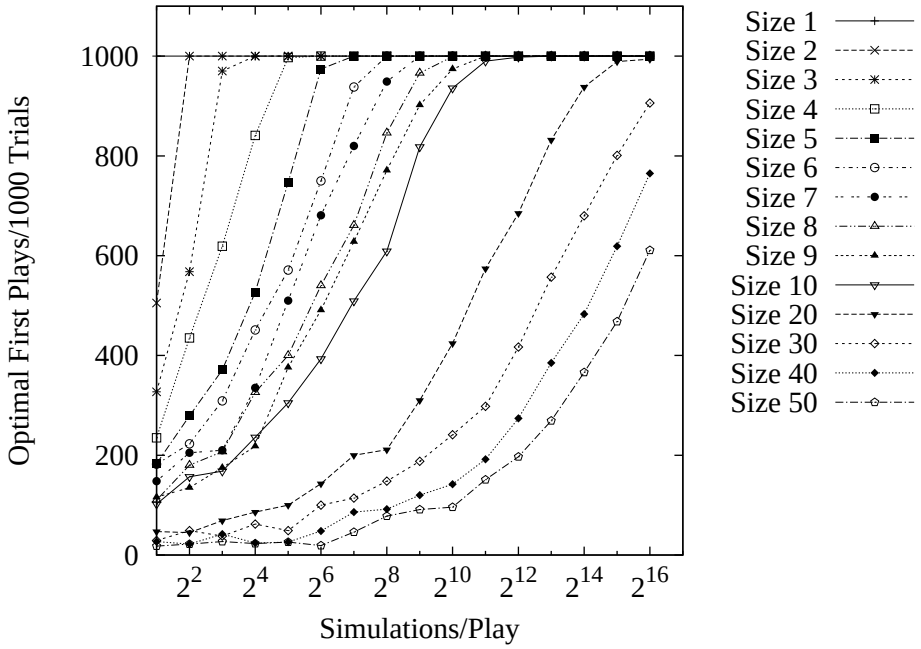
4 Experiments

The experiments investigate the properties of MCTS with UCT and RAVE. Results are shown for varying the size of the game and the number of simulations used in the search, the influence of RAVE, training on optimal play vs. "good" play, and the effect of misleading RAVE updates. This paper reports our findings thus far, and will hopefully lead to further experiments with MCTS enhancements and improved algorithms. The 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). The Fuego version used in these experiments was Fuego release 0.2.

4.1 Game Size and Simulation Limits

The complexity of the SOS game is determined solely by its size. SOS(n) produces a game tree of size $n!$ since transpositions and tree pruning are not present with in this model. For example, the complete SOS(10) game tree contains 3628800 leaf nodes. The larger the game is, the more difficult it is for a game-playing program to solve. The performance of the 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 baseline for the performance of UCT in SOS, experiments varying n and s were performed.

Fig. 1. Plain UCT without enhancements in SOS(n).



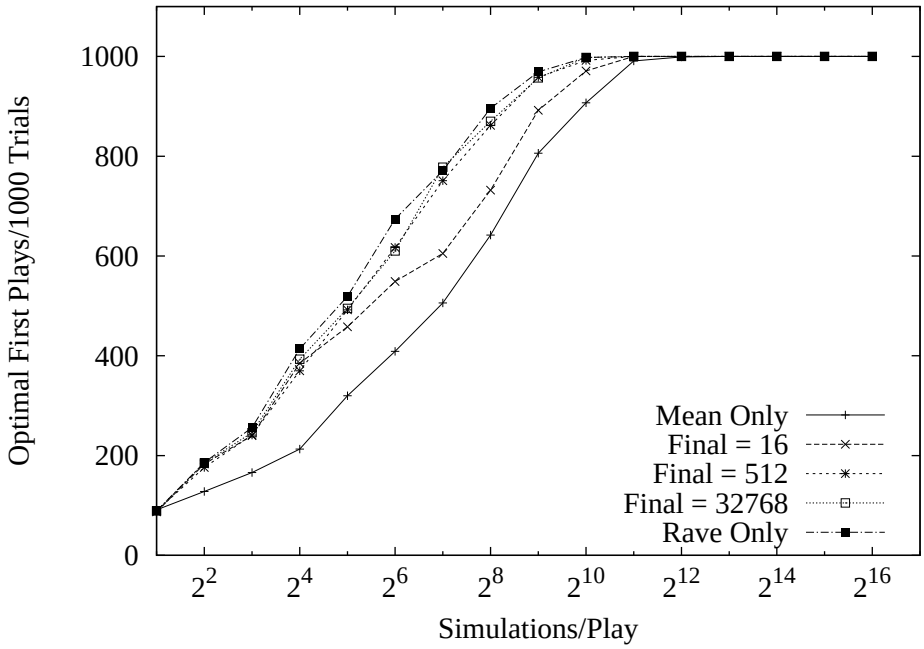
Each data point in Figure 1 represents how often the optimal first move was chosen in 1000 trials. For $n < 10$ the program quickly converges to optimal play. In the range $10 \leq n \leq 50$, convergence becomes progressively slower. The convergence rates seem similar to those in [8] for games with a comparable number of leaf nodes. For further experiments, SOS(10) was chosen as a compromise between game difficulty and runtime until convergence.

4.2 RAVE

Figure 2 shows experiments with RAVE. Even low values of RaveWeightFinal such as 16 give noticeable improvements. Large values of RaveWeightFinal show diminishing returns, with 512 producing similar results to 32768 or higher values.

As stated previously, 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 2 includes this RAVE-only data as well. Of course, this method would not work in other games where the value of a move depends on the timing when it is played.

Fig. 2. UCT+RAVE, varying RaveWeightFinal in SOS(10).



4.3 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 high win that probably contained many high-scoring moves gets a slightly better evaluation than a close win.

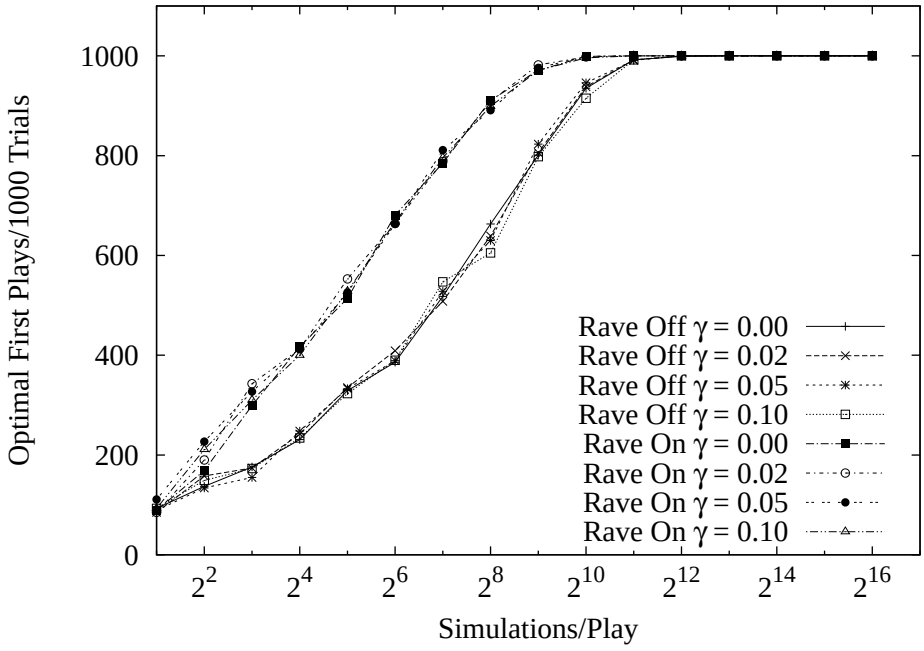
In SOS with score bonus, losses are evaluated in a range from 0 to γ and wins from $1-\gamma$ to 1, for a parameter γ. A minimal win is awarded $1-\gamma$, and as maximal possible win a score of 1. All other game outcomes are scaled linearly in this interval. The values assigned for losses are analogous.

Results for $\gamma = 0.1$, $\gamma = 0.05$, and $\gamma = 0.02$ are shown in Figure 3. Score bonus fails to improve gameplay in SOS. However, it 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 $\gamma = 0.02$.

4.4 False Updates

While RAVE works very well in SOS and Go, it is not reliable in all games. Since RAVE updates the value of all moves in a winning sequence and ignores temporal

Fig. 3. Graph of Score Bonus results on SOS(10). The RAVE experiments were performed with the RAVE-only settings.

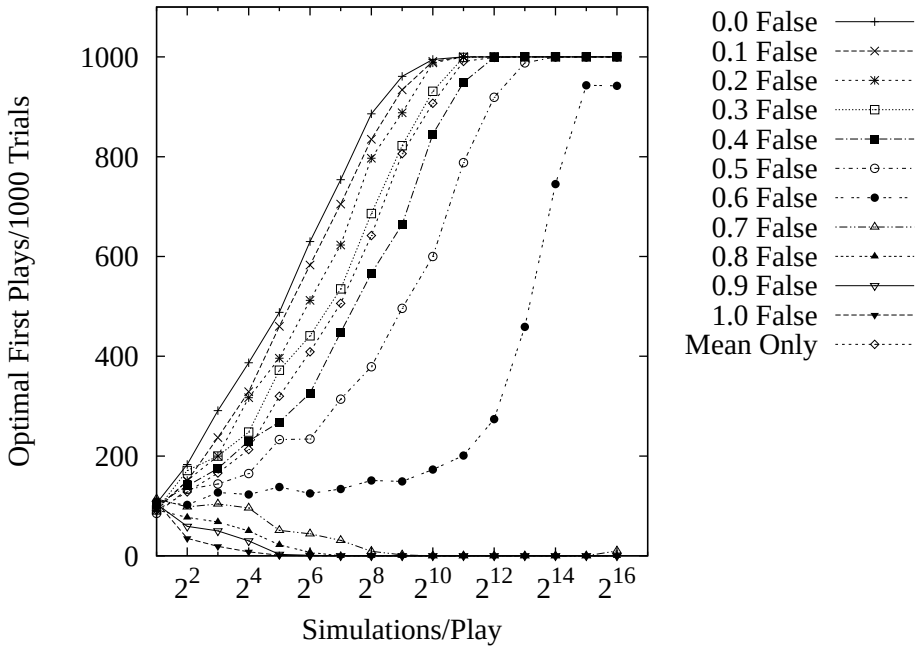


information, it can lead the search astray. In situations where specific moves are only helpful at a given time, RAVE can weaken game-play instead of improving it. Suppose that in a game, a certain last move will always lead to a win, but is useless at all other times. The high RAVE value that this move is likely to earn early in simulations is likely to cause the game-playing program to waste a lot of time exploring paths related to this winning move at higher points in the tree. It is potentially possible for such a situation to result in very poor value estimates when the simulation limit is reached and thus, a poor play to result.

Experiments involving random false updates can simulate the effect of misleading RAVE values. With a probability of μ, the Rave update for all moves in the current simulation uses the inverse evaluation $InverseEval = 1 - Eval$. $RaveWeightFinal$ was set to a high value in this set of experiments so as to pronounce the effect of the experiment; additionally, this setup also reflects scenarios where little is known about the game, but RAVE is expected to be a strong estimator. The results of these experiments are summarized in Figure 4.

Even with the influence of the mean value as a steadying force, the performance of a program with RAVE influence 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 the false update rate is between 0 and 0.3. Up to an error rate of 0.5, the error can be interpreted as noise that slows down convergence; error rates

Fig. 4. Effect of False Updates on RAVE with $RaveWeightFinal = 32768$. Experiments performed in SOS(10).



above 0.5 have an antagonistic effect upon the RAVE heuristic. Even with $\mu = 0.6$ the performance still improves with the number of simulations. These results suggest that with unbiased noise as provided by false updates above, RAVE is a robust heuristic that is resilient against a reasonable level of error.

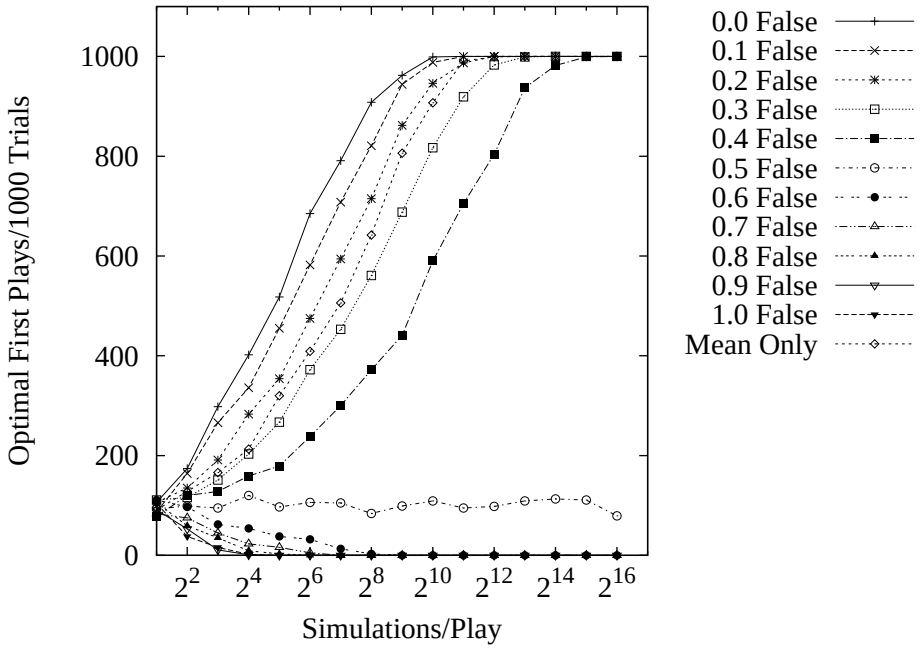
It would be interesting to study a biased version of false updates, that selectively distorts the updates related to specific moves. This may be a model that is closer to what is seen in Go, and present more problems for the search.

Since RAVE-only works well in SOS, it is interesting to see the effect of false updates here. The results in Figure 5 show a similar trend while the error rate is low. The $\mu = 0$ data corresponds to Figure 2, where RAVE-only is better than UCT+RAVE. However, at $\mu = 0.2$ RAVE-only is already slightly worse, and at $\mu = 0.4$, RAVE-only is far worse than the UCT+RAVE version shown in Figure 4. At $\mu = 0.5$ the algorithm behaviour becomes random.

5 Analysis

The 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.

Fig. 5. Effect of False Updates in RAVE-only on SOS(10).



The RAVE experiments show significantly better performance than plain UCT, even with distorted RAVE updates. The experiments suggest that the RAVE heuristic is robust against unbiased noise and performs well even with a fair level of error. However, the RAVE experiments also suggest that performance can be significantly improved if we understand a little about the environment we are applying RAVE in. In games where the value of moves do not change, RAVE provides a much stronger estimate than the mean value. The false update experiments also suggest that if RAVE updates are strongly misleading, RAVE can be very detrimental, and thus, it needs to be weakened or eliminated from the estimate to improve program performance.

6 Conclusion and Future Work

The Sum of Switches game provides a simple, well-controlled environment where behaviour is easily measured. In this framework, a series of experiments with UCT and RAVE were performed. Although current trends promote parallelization as a means to increase simulations completed and program performance, the fact remains that game trees are often exponentially growing in size, meaning that simulations have to be increased by large quantities in order to produce small gains in performance. However, the RAVE experiments also suggest that by enhancing our algorithm and fine-tuning the parameters, significantly stronger play can be achieved without requiring more samples. Future work includes further investigation of Rave in hostile environments as well as

exploration on how to moderate the influence of Rave to adapt to the environment it is in. The goal is to automatically adapt a complex UCT-based algorithm to a particular game situation.

Acknowledgements

This research was supported by the DARPA GALE project, contract No. HR0011-08-C-0110, and by NSERC, the Natural Sciences and Engineering Research Council of Canada.

References

1. B. Bouzy and B. Helmstetter. Monte-carlo go developments. In *ACG. Volume 263 of IFIP., Kluwer (2003) 159174 5 Typically the*, pages 159–174. Kluwer Academic, 2003.
2. B. Brügmann. Monte Carlo Go, March 1993. Unpublished manuscript, `http://www.cgl.ucsf.edu/go/Programs/Gobble.html`.
3. R. Coulom. Whole-history rating: A bayesian rating system for players of time-varying strength. In van den Herik et al. [12], pages 113–124.
4. M. Enzenberger and M. Müller. Fuego, 2008. `http://fuego.sf.net/` Retrieved December 22, 2008.
5. H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In D. Fox and C. P. Gomes, editors, *AAAI*, pages 259–264. AAAI Press, 2008.
6. 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.
7. S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo Go, 2006. Technical Report RR-6062.
8. 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.
9. R. J. Lorentz. Amazons discover monte-carlo. In van den Herik et al. [12], pages 13–24.
10. J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(11):1203–1212, 1989.
11. Stephen J. J. Smith and Dana 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.
12. H. Jaap van den Herik, X. Xu, Z. Ma, and M. 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.