Investigating the Limits of Monte Carlo Tree Search Methods in Computer Go

Shih-Chieh Huang*[1] and Martin Müller†[2]

[1]DeepMind Technologies, London, United Kingdom
[2]Department of Computing Science, University of Alberta, Edmonton, Canada T6G 2E8

Abstract

Monte Carlo Tree Search methods have led to huge progress in Computer Go. Still, program performance is uneven - most current Go programs are much stronger in some aspects of the game, such as local fighting and positional evaluation, than in others. Well known weaknesses of many programs include the handling of several simultaneous fights, including the *two safe groups* problem, and dealing with coexistence in seki.

Starting with a review of MCTS techniques, several conjectures regarding the behavior of MCTS-based Go programs in specific types of Go situations are made. Then, an extensive empirical study of eleven leading Go programs investigates their performance in two specifically designed test sets containing *two safe group* and seki situations.

The results give a good indication of the state of the art in computer Go as of 2012/2013. They show that while a few of the very top programs can apparently solve most of these evaluation problems in their playouts already, these problems are difficult to solve by global search.

*shihchie@ualberta.ca
†mmueller@ualberta.ca

1 Introduction

In computer Go, it has been convincingly shown in practice [3] that the combination of simulations and tree search in Monte Carlo Tree Search is much stronger than either simulation by itself, or other types of tree search which do not use simulations. It is also well known from practice that there remain several types of positions where most current MCTS-based programs do not work well. Examples are capturing races or semeai [9], positions with multiple cumulative evaluation errors [7], ko fights, and close endgames.

We present a number of case studies of specific situations in Go where many MCTS-based programs are known to have trouble. In each case, we develop a collection of carefully chosen test cases which illustrate the problem. We study the behavior of a number of state of the art Go programs on those test positions, and measure the success of their MCTS searches and their scaling behavior when increasing the number of simulations. Measures include the ability to select a correct move in critical situations and the estimated winning probability (the *UCT value*) returned from searches.

2 Some Hypotheses and Research Questions about MCTS for Computer Go

The main components influencing the strength of current Go programs are their simulation policy, their tree search and their other in-tree knowledge such as patterns. If we are trying to evaluate Go programs indirectly, without analyzing their source code, we can use test positions, which can be classified as follows:

1. Positions which are *well-suited* for MCTS - programs can solve them with either a good search, or with a good policy, or both.

2. Positions which are *search-bound*: a stronger search can solve them, but simply using a stronger policy is not enough. Candidates might be hard tactical problems with unusual solution moves, such as filling your own eye.

3. Positions which are *simulation-bound*: better simulation policies can play these positions well, but search alone will fail. Possible examples are seki and semeai.

4. Positions which are *hard* for current programs compared to humans. Neither better policies nor (global) search helps. The main candidates are positions with multiple simultaneous fights [7], whose combined depth is too deep to be resolved by global search, and which contain too many "surprising" moves to be "solved" in simulations.

Some related research questions are:

1. Is it possible to design tests that separate playout strength and search strength of current Go programs?
2. Is it possible to identify types of positions where all current Go programs fail?
3. Is it possible to estimate the overall strength of programs from such analysis of specific types of positions?

The current paper begins this investigation by analyzing two cases in detail, and showing test results for many of today's leading Go programs.

3 Test Scenarios

This section discusses and develops two test scenarios in detail: *Two safe groups* (TSG) and seki. All test data and experimental results are available at `https://sourceforge.net/apps/trac/fuego/wiki/MctsGoLimitsTestData`.

3.1 Two Safe Groups

A main motivation for developing this test scenario is the observed poor performance of the Fuego program when playing Black on a 9×9 board, especially with the large komi of 7.5 points [7]. A frequent 9×9 opening sees Black starting on the center point, then building a wall through the middle, while White establishes a group on each side. If both groups live on a reasonable scale, then it is hard for Black to get enough territory. Problems in Fuego's play as Black are often caused by over-optimism about being able to kill at least one of the white groups.

In this context, a *safe* group is defined to be a group that is safe with reasonable play, but may easily die during playouts. In contrast, a *solid* group already has very definite eye shape, and will rarely die even in simulations. It has no weaknesses that could lead to accidental death during simulation. Figure 1 shows an example from a game between Fuego and a professional player. Both white groups are safe, since it is easy to make two eyes for them with competent play. An amateur low-level Dan player should have no trouble winning with White. However, these groups are not solid at the level of Fuego's simulations - it is easy for one or both of these groups to die in the course of a simulation following a not-too-informed policy. Accordingly, Fuego's winning rate was close to 50%, and remained so for the next thirty moves of this game, while for the professional human player the game was decided by move 12 at the latest.

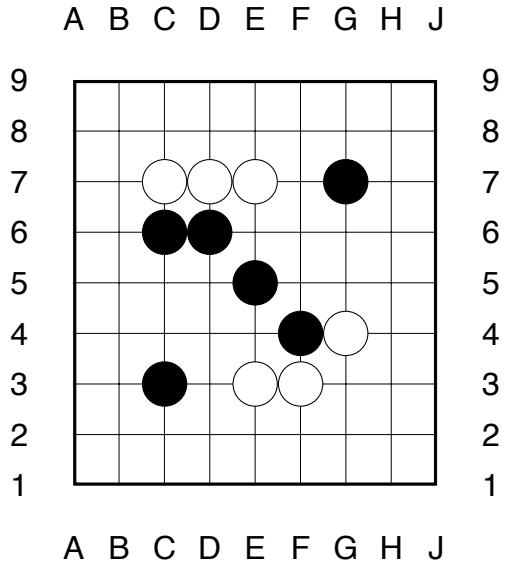

Figure 1: Two safe white groups. Example from game Ping-Chiang Chou 4 Dan (W) - FUEGO-GB PROTOTYPE (B), Barcelona 2010. White wins by resignation.

3.1.1 Test sets for One and Two Safe Groups

The tests are developed in groups of three positions which are closely related. All positions are lost for Black. In the first case (TSG), White has two groups which are *safe* but not *solid* according to the definition above. The second and third position in each group (TSG-1 and TSG-2) is very similar to the first one, but one of the two *safe* groups has been made *solid* by adding a few extra stones to simplify the eye space. Figures 2 and 3 show an example.

In creating the test cases and test scenarios, the authors attempted to control the difficulty, such that the game tree built by Fuego's MCTS is deep enough to (mostly) resolve the life and death for a single *safe* group, but not enough for two or more groups. In informal tests with Fuego, the simulations made enough mistakes that groups got killed a reasonable fraction of the time.

Our working hypothesis is that these test cases are well suited to show the difference between simulation policies and search in current MCTS Go programs.

The purpose of the two-safe-groups (TSG) test set is to check each program's evaluation of a given position. The set consists of 15 problems, all with Black to play. In each problem, White is winning, with two safe groups. Therefore, the closer the black

program's evaluation is to 0 (loss), the better.

Many test cases in TSG are hard for many current Go programs, since they involve two simultaneous and mostly independent fights. Even if the evaluation of each single fight is say 60% correct, the probability of White winning a simulation is only $0.60 \times 0.60 = 0.36$. The conjecture is that under the tested conditions, programs cannot resolve both fights within the game tree, and must rely on simulations to resolve many groups.

In contrast, the two corresponding simpler test cases contain only a single fight: in the TSG-1 set, the top White group is 100% alive for any reasonable random playout that does not fill real eyes, while in TSG-2, the bottom white group is 100% alive.

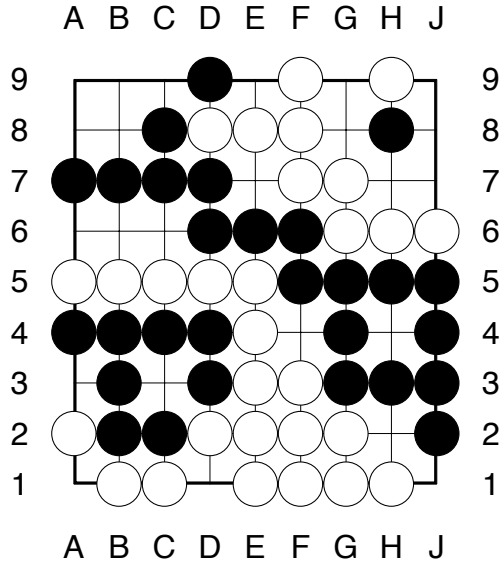

Figure 2: TSG case 7.

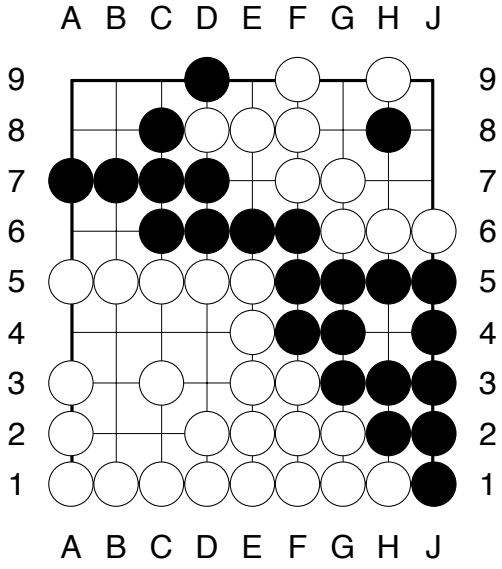

Figure 3: TSG-1 case 7, with *solid* white group on bottom.

3.2 Seki

Seki positions are designed to illustrate the "blindness" of global search when simulations are misleading. The goal was to create positions that are decided not by search, but by playouts. If crucial moves are missing, or are systematically misplayed in simulations, then the correct solution does not "trickle up" the tree until it is much too late.

The Seki Test Set consists of 33 cases, all with White to play. Figure 4 shows an example. At least one local seki situation is involved in each case. In each case, there is at

least one correct answer that leads to White's win. Correct answers might include a pass move.

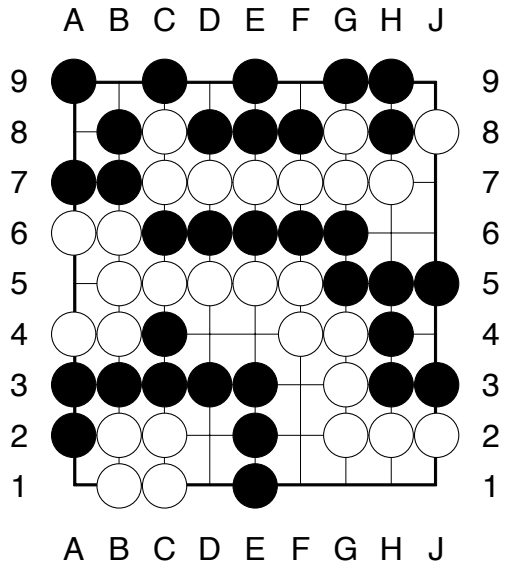

Figure 4: Seki case 13.

4 Experiments

Experiments were run with the TSG, TSG-1, TSG-2 and Seki Test Sets described above. Each test case is a 9x9 Go position. The aim is to measure a program's performance, including scaling, in these cases. Scaling is measured by increasing the number of playouts from 1k to 128k, doubling in each step.

For each test set, one or more regression test files in GTP format were created. The seki regression test checks a program's best move in a given test position. A program's answer is counted as correct only if its move is identical to one of the provided correct answers.

For the Two-Safe-Groups regression tests, in order to create problems with a simple, binary answer, first a threshold T in the range [0, 0.5] was selected. A program was requested to output -1 if its winning rate after search was at most T, and +1 otherwise. Since Black is losing in all test instances, the correct answer is -1 in all cases. Three tests with threshold T set to 0.3, 0.4 and 0.5 respectively were run.

4.1 The Programs

Table 1 lists the participating programs in alphabetical order, with the versions and processor specifications provided by their authors. These programs strongly represent the state-of-the-art of MCTS in view of their splendid records in recent computer Go competitions. CRAZY STONE won the 6th UEC Cup in 2013, followed by ZEN, AYA, PACHI and FUEGO. ZEN won all three Go events - 9x9, 13x13 and 19x19 - at the most recent Computer Olympiad in Tilburg, and STEENVRETER came in 2nd place on both 13x13 and 19x19. PACHI won the May 2012 KGS bot tournament. FUEGO won the 4th UEC Cup in 2010. THE MANY FACES OF GO won the 13x13 Go event at the 2010 Computer Olympiad and the 2008 Computer Olympiad. GOMORRA has many solid results in recent competitions. GNU GO is included along with THE MANY FACES OF GO as an example of a program with a large knowledge-intensive, "classical" component.

Table 1: The programs participating in the experiments.

Name	Version	Processor Specification
AYA	7.36e	Core2Duo 1.83GHz
CRAZY STONE	0013-07	Six-Core AMD Opteron(TM) 8439 SE
FUEGO	tilburg	Intel(R) Xeon(R) E5420 @ 2.50GHz
GNU GO	3.9.1	Intel(R) Xeon(R) E5420 @ 2.50GHz
GOMORRA	r1610	Intel(R) Core(TM) i7-860 @ 2.80GHz
HAPPYGO	1.0	Intel(R) Xeon(R) E3-1225 @ 3.10GHz
THE MANY FACES OF GO	0013-07	Intel(R) Core(TM) i7-3770 @ 3.40GHz
PACHI	9.99	Intel(R) Xeon(R) E5420 @ 2.50GHz
STEENVRETER	r123	Intel(R) Core(TM)2 Duo T9300 @ 2.50GHz
STONEGRID	0.3.4 155	Intel(R) Core(TM) i7-2600 @ 3.40GHz
ZEN	9.6	MacPro

4.2 Results, Discussion and Future Work

Figure 5 shows the results of the seki regression test. The Programs with heavy Go-knowledge implementations such as ZEN, STEENVRETER, THE MANY FACES OF GO and CRAZY STONE solved 33, 33, 32 and 30 out of 33 cases at peak respectively. FUEGO didn't have any seki knowledge in the playout and failed in about half of the cases. Overall, running with more playouts just slightly helps with solving these cases. Only AYA and PACHI showed good scaling: AYA improved from 27 to 32 solved cases when scaling from 1k to 128k playouts, and PACHI improved from 21 to 26 solved. GNU GO solved 29 cases at level 10, which is not shown in the figure. The test strongly demonstrated that applying seki knowledge to the playouts is crucial to MCTS programs to play in seki

situations correctly.

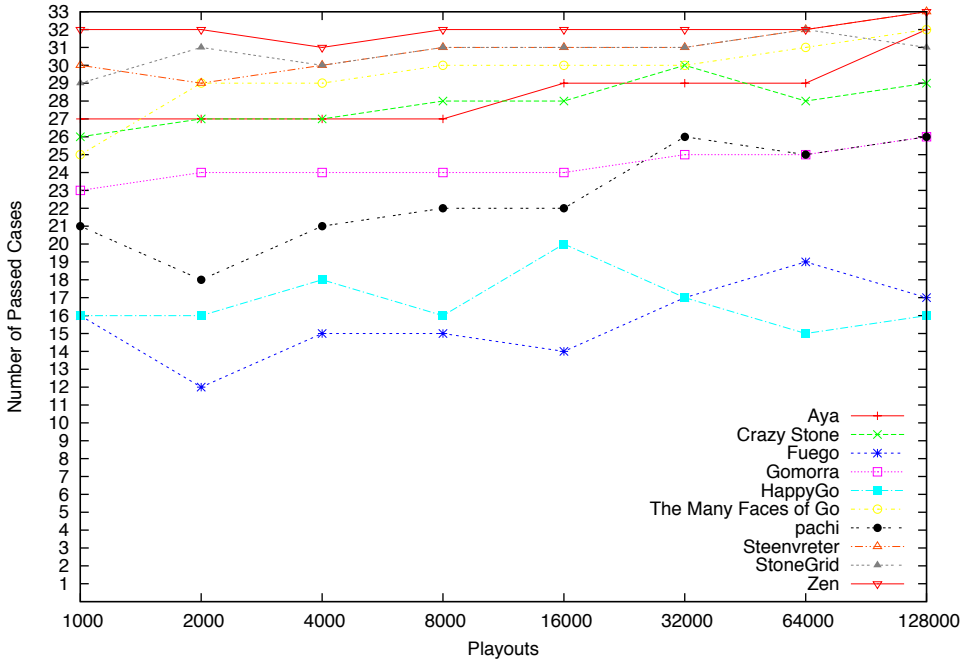

Figure 5: The results of the seki regression test.

Figures 6 to 12 show the individual results of the two-safe-groups regression tests (TSG, TSG-1, TSG-2). Note that only 7 out of 11 programs supported querying their UCT value, and these results are shown.

TSG is a difficult test for these MCTS programs because most of them failed in more than half of the cases, even for $T = 0.5$. FUEGO didn't solve any case for $T = 0.3$, even with 128k playouts. GOMORRA solved only one case, with 8k playouts. This indicates that many MCTS Go-playing programs cannot evaluate life-and-death and semeai situations correctly without specific knowledge. Programs solved more cases for $T = 0.4$ and $T = 0.5$, indicating that their evaluations of these positions are mostly between 0.3 and 0.5, far from the optimal value 0. The TSG problem, the difference in performance between TSG on one side and TSG-1 and TSG-2 on the other side, is most pronounced in FUEGO, GOMORRA, PACHI, THE MANY FACES OF GO and STEENVRETER. It is less pronounced in AYA, which surprisingly does relatively poorly even with one solid group. TSG-1 and TSG-2 can often be resolved by search, since there is only a single fight. STEENVRETER solved most of the cases of TSG-1 and TSG-2 and did well for TSG with a $T = 0.5$ cutoff.

ZEN did by far the best overall, and even solved most cases of TSG. While the techniques used in ZEN have not been published, it's authors have publicly described ZEN as using knowledge-heavy, slow but very well-informed playouts.

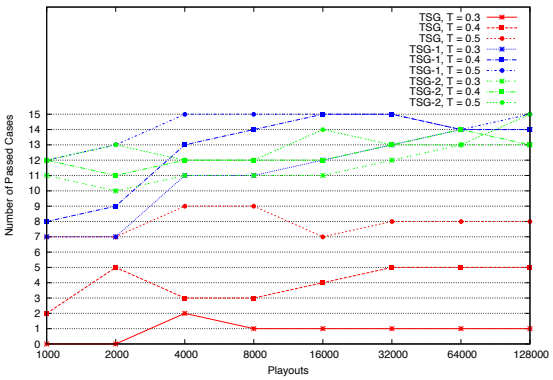

Figure 6: The results of the two-safe-groups regression test of AYA.

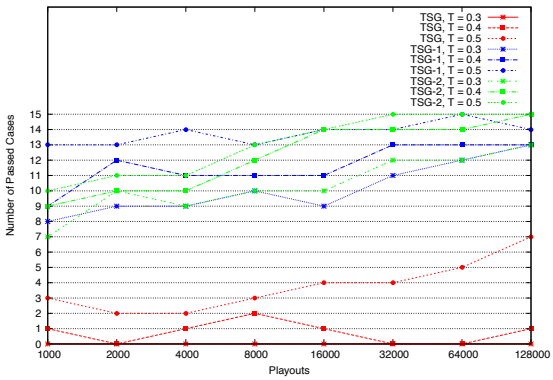

Figure 7: The results of the two-safe-groups regression test of FUEGO.

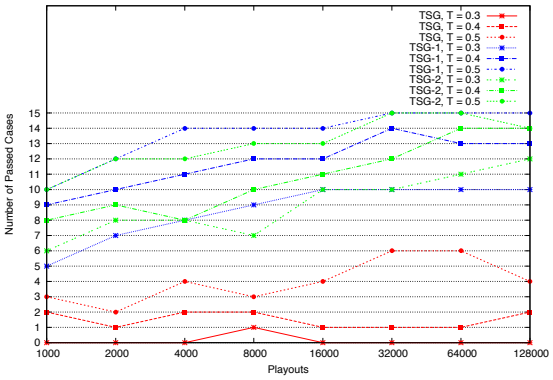

Figure 8: The results of the two-safe-groups regression test of GOMORRA.

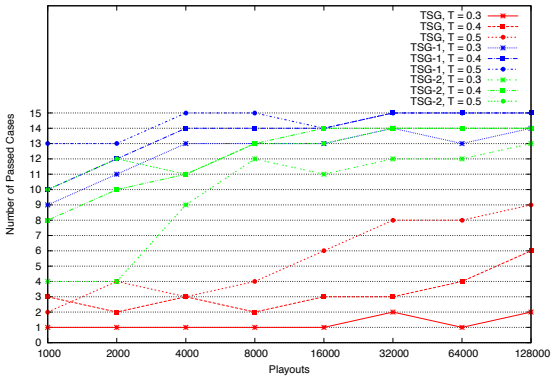

Figure 9: The results of the two-safe-groups regression test of THE MANY FACES OF GO.

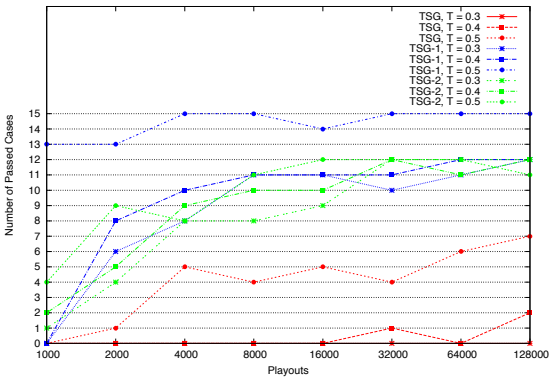

Figure 10: The results of the two-safe-groups regression test of PACHI.

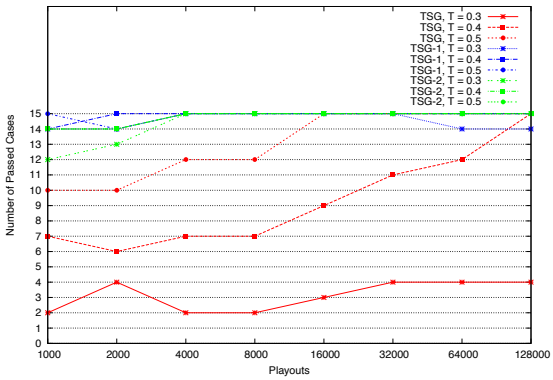

Figure 11: The results of the two-safe-groups regression test of STEENVRETER.

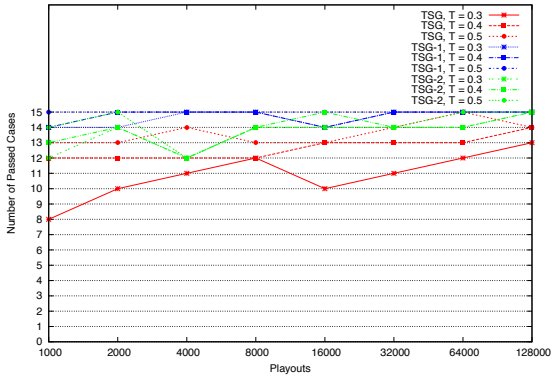

Figure 12: The results of the two-safe-groups regression test of ZEN.

We believe that this work provides a valuable first step towards analyzing the state of the art in Computer Go. One lesson is that programs which are quite similar in strength can exhibit very different behavior in terms of their simulation policies and search scaling. We hope that this approach can be helpful for current and future Go programmers to analyze the behavior of their programs. Future work includes: developing more test sets, such as semeai, more than two groups, connection problems, and endgames; scaling to more simulations; and developing similar tests for other games such as Hex.

5 Related work

There is a huge and quickly growing literature about applications of MCTS to games and many other domains. See [1] for a recent survey. Much of that work focuses on algorithm variations, parameter tuning etc. There is much less work on identifying limitations.

One well-known result is the tower-of-exponentials worst case convergence time for UCT [2, 6]. Work by Ramanujan and Selman shows that UCT can be over-selective and miss narrow tactical traps in chess [8]. In practice, use of MCTS in games such as chess and shogi has been confined to small niche applications so far.

Negative experimental results on the correlation between the strength of the policy as a standalone player vs its strength when used for guiding simulations in MCTS were shown in [4]. Followup work on simulation balancing includes [5, 10].

Acknowledgements

This project would not have been possible without the support of all the Go program's authors. Many of them supported us by implementing extra GTP commands in their programs, and by helping to debug the test set through testing early versions with their programs.

Financial support was provided by NSERC, including a Discovery Accelerator Supplement grant for Müller which partially supported Huang's stay.

References

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

[2] P.-A. Coquelin and R. Munos. Bandit algorithms for tree search. In R. Parr and L. van der Gaag, editors, *UAI*, pages 67–74. AUAI Press, 2007.

[3] S. Gelly. *A Contribution to Reinforcement Learning; Application to Computer-Go*. PhD thesis, Université Paris-Sud, 2007.

[4] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[5] S. Huang, R. Coulom, and S. Lin. Monte-Carlo simulation balancing in practice. In J. van den Herik, H. Iida, and A. Plaat, editors, *Computers and Games*, volume 6515 of *Lecture Notes in Computer Science*, pages 81–92. Springer, 2010.

[6] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[7] M. Müller. Fuego-GB Prototype at the human machine competition in Barcelona 2010: a tournament report and analysis. Technical Report TR 10-08, Dept. of Computing Science. University of Alberta, Edmonton, Alberta, Canada, 2010. `https://www.cs.ualberta.ca/research/theses-publications/technical-reports/2010/TR10-08`.

[8] R. Ramanujan and B. Selman. Trade-offs in sampling-based adversarial planning. In F. Bacchus, C. Domshlak, S. Edelkamp, and M. Helmert, editors, *ICAPS*, pages 202–209. AAAI, 2011.

[9] A. Rimmel, O. Teytaud, C.-S. Lee, S.-J. Yen, M.-H. Wang, and S.-R. Tsai. Current frontiers in computer Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):229–238, 2010. Special issue on Monte-Carlo Techniques and Computer Go.

[10] D. Silver and G. Tesauro. Monte-Carlo simulation balancing. In A. Danyluk, L. Bottou, and M. Littman, editors, *ICML*, volume 382 of *ACM International Conference Proceeding Series*, pages 945–952. ACM, 2009.