Revisiting Monte-Carlo Tree Search on a Normal Form Game: NoGo

C.-W. Chou, Olivier Teytaud, Shi-Jim Yen

Revisiting Monte-Carlo Tree Search on a Normal Form Game: NoGo

C.-W. Chou[2],O. Teytaud[1], S.-J. Yen[2]

[1] TAO (Inria), LRI, UMR Cnrs 8623 Univ. Paris-Sud
[2] CSIE, in National Dong-Hwa University, Houlian, Taiwan.

Abstract. We revisit Monte-Carlo Tree Search on a recent game, termed NoGo. Our goal is to check if known results in Computer-Go and various other games are general enough for being applied directly on a new game. We also test if the known limitations of Monte-Carlo Tree Search also hold in this case and which improvements of Monte-Carlo Tree Search are necessary for good performance and which have a minor effect. We also tested a generic Monte-Carlo simulator, designed for "no more moves" games.

1 Introduction

Monte-Carlo Tree Search has emerged in computer-Go[8]; in this old very difficult game, it quickly outperformed classical alpha-beta techniques. It was then applied in many games[1, 19], with great success in particular when no prior knowledge can be used (as in e.g. general game playing[18]). It was also applied in planning[14], difficult optimization [9], active learning[17], general game playing[18], incidentally reaching infinite (continuous) domains.

The NoGo game is similar to Go, in the sense that each player puts a stone on the board alternatively, and stones do not move; but the goal is different: the first player who either suicides or kills a group has lost the game (it can be rewritten conveniently as a normal form game, i.e. a game in which the first player with no more legal moves looses the game). It has been invented by the organizers of the Birs workshop on Combinatorial Game Theory 2011 (`http://www.birs.ca/events/2011/5-day-workshops/11w5073/`) for being a completely new game; in spite of the syntaxic similarity with Go (notion of group, killing, black and white stones put alternately on the board), it is not (at all) tactically related to Go. We use NoGo as a benchmark as it is really different from the classical testbeds, and it is non-trivial as shown by human and computer tournaments in Birs.

NoGo is immediately PSPACE because it is solvable in polynomial time by an alternating Turing machine[6] (the horizon of the game is at most the number of cells as each location is played at most once per game). The NoGo game is difficult to analyze as it does not look like any known game and we don't see how to simulate any game in NoGo positions. On the other hand, we could not find any proof of PSPACE-hardness (i.e. NoGo is PSPACE, but PSPACE-completeness is not ensured).

We tested whether NoGo is also almost solved in 7x7 by playing games with 1 second, 2s, 4s, 8s, 16s and 32s per move; we got 30% ± 8%, 40% ± 9%, 57% ± 9%, 50% ± 9%, 40% ± 15% respectively for white[1], which suggests that the game is deeper than expected and far from being an immediate win for any player, in spite of the fact that it is PSPACE (whereas Go which japanese rules is EXP-complete[16]).

2 A brief overview of Monte-Carlo Tree Search (MCTS)

The reader is refered to [8, 21, 13] for a complete introduction to Monte-Carlo Tree Search; we here only briefly recall the algorithm in order to clarify notations. The most well known variant of MCTS is probably Upper Confidence Trees (UCT); it is based on the Upper Confidence Bound algorithm [12, 2] and presented below[11]. Basically, this algorithm is based on:

- A main loop; each iteration in the loop simulates a game, from the current state to a game over.
- A memory (basically a subtree of the game tree, each node being equipped with statistics such as the number of simulations having crossed this node in the past and such as the sum of the rewards associated to these simulations).
- A bandit algorithm, which chooses which action is chosen in a given state and a given simulation. The bandit algorithm is based on statistics on previous simulations; it boils down to a naive random choice when no statistics are available in the state[2]. The difference between Monte-Carlo algorithms[4] and Monte-Carlo Tree Search algorithms[8] is precisely the existence of this bandit part, which makes simulations different from a default Monte-Carlo when there are statistics in memory (these statistics come from earlier simulations).

$L(s)$ denotes the set of legal moves in state s. $o_l(s)$ is the action chosen by the bandit in state s the l^{th} time the bandit algorithm was applied in state s. Alg. 1 presents the UCT algorithm in short. We refer to [13] for a complete description of the state of the art in Monte-Carlo Tree Search algorithms for Go.

3 Improvements over the initial MCTS algorithm

We recall and test some classical improvements or known facts over the initial Monte-Carlo Tree Search algorithms: Rapid Action-Value Estimates (section 3.1), slow node creation (section 3.2), anti-decisive moves (section 3.3) and upper confidence bounds formula (section 3.4). We also tested an expensive heuristic in the Monte-Carlo part (section 3.5). We tested the scalability of the algorithm (section 3.6). Unless otherwise stated, experiments are performed in 7x7 versions of the games.

[1] With the best version we had, after all the experiments reported in this paper.

[2] In Computer-Go, better choices exist than a uniform random player; we'll discuss this later.

3.1 Rapid action value estimates

Rapid action value estimates (RAVE) are probably the most well known improvement of MCTS[10]. The principle is to replace, in the score function $score_t(s,o)$, the average reward, among past simulations applying move o in state s, by a compromise r'' between (i) the average reward r, among past simulations applying move o in state s; (ii) and the average reward r', among past simulations applying move o *after* state s. r'' is usually computed by a formula like $\alpha r + (1-\alpha)r'$, where $\alpha = nb_t(s,o)/(K + nb_t(s,o)$ for some empirically tuned constant K. We then get a formula as follows:

$$score_t(s,o) = \alpha r + (1-\alpha)r' + C\sqrt{\frac{\log(t+1)}{nb_t(s,o)+1}}.$$

We refer to [5, 10] for the detailed implementation when a given location can be played by two players in the same game (at different time steps); this can happen in Go (thanks to captures which make some locations free), but this is not the case for NoGo. It works quite well in many games, and we see below that it also works for NoGo (we test the efficiency of the algorithm for various numbers of simulations per move):

Number of sims per move	Success rate vs the version without RAVE ± 2 std deviations
50	56 % ± 5%
500	64 % ± 4%
1000	64 % ± 2%
2000	70 % ± 4%
4000	78 % ± 3%
5000	71 % ± 11%
8000	79 % ± 3%
16000	82 % ± 3%
32000	83 % ± 3%
50000	89 % ± 14%
64000	83 % ± 3%
128000	82 % ± 3%
256000	84 % ± 2%
1024000	83 % ± 3%

3.2 Rate of node creation

MCTS algorithms can require a huge memory; we here test the classical improvement consisting in creating a node in memory if and only if (i) its father node has already been created in memory and (ii) it has been simulated at least z times.

We test the number of games won with various values of z, against the classical $z = 5$ from Computer-Go. Results are as follows (over 300 games, with $z = 1$, 2, 4, 8, 16 sims before creation):

Nb of sims per move	$z=1$	$z=2$	$z=4$	$z=8$	$z=16$
NoGo game					
100	183	180	153	161	174
200	193	189	158	133	158
400	186	171	164	129	120
1600	171	156	147	140	133
Go game					
100	177	174	142	156	133
200	196	173	168	135	139
400	180	167	162	140	116
1600	165	175	167	126	120

The optimum is here clearly for 1 simulation before creation, but these results are for a fixed number of simulations per move. A main advantage of this modification is that it makes simulations faster by reducing the numbers of creations (which take a lot of CPU, in spite of a strongly optimized memory management).

So we now reproduce the experiments with 1s per move (instead of a fixed number of simulations per move), against the value $z = 5$ (NoGo game and game of Go), and get results as follows:

Experimental condition	Success rate against $z = 5$ $\pm$ 2 std deviations
1 s/move	
Go, $z = 1$	37.7 % $\pm$ 3%
Go, $z = 2$	48.3 % $\pm$ 3%
NoGo, $z = 1$	36.0 % $\pm$ 3%
NoGo, $z = 2$	42.7 % $\pm$ 3%
4 s/move	
NoGo, $z = 1$	23.4% $\pm$4%

We see that with $z = 1$ we get poor results; and with $z = 2$ we are still worse than $z = 5$, in particular when the time per move increases (more time makes memory requirements important). This is with constant time per move, and for a small time per move; we now check the memory usage (which is an issue when thinking time increases, as memory management heuristics waste a lot of time) when playing just one move; we plot below the memory usage (as extracted by valgrind[15]) as a function of z (we removed 4,900,000 bytes, which is the constant memory usage by the program independently of simulations):

z (nb of simulations before node creation)	Used memory
NoGo, 1000 sims/move	
1	845,060 bytes in 72,177 blocks
2	518,496 bytes in 70,625 blocks
4	315,240 bytes in 69,639 blocks
10	172,244 bytes in 68,922 blocks
NoGo, 10000 sims/move	
1	8,204,836 bytes in 109,233 blocks
2	5,144,512 bytes in 94,818 blocks
4	3,054,572 bytes in 84,590 blocks
10	1,450,064 bytes in 76,292 blocks
NoGo, 100000 sims/move	
1	61,931,036 bytes in 361,275 blocks
2	53,613,076 bytes in 327,108 blocks
4	30,998,976 bytes in 212,358 blocks
10	15,499,728 bytes in 129,705 blocks

We therefore see that we have both (i) clear memory improvement (ii) better results even with moderate time settings (even 100 000 simulations per move is not very high when using a strong machine).

3.3 Efficiency of the playouts and decisive moves

We first checked the efficiency of the "playout" part, i.e. the bandit algorithm when no statistics are available. For doing this we tested the simple replacement of the playout part (i.e. the part of the simulation after we leave the part of the graph which is kept in memory), by a coin toss (a winner is randomly chosen with probability $\frac{1}{2}$): with 2000 simulations per move, we get a 78% success rate for the version with random playouts versus the version with coin-toss (with standard deviation 3.25%); the playout principle is validated.

The MCTS revolution in Go started when a clever playout part was designed[21]. We here test the standard Go playouts for NoGo; the performance is as follows against a naive Monte-Carlo algorithm:

Nb of simulations per move	Success rate of the version with Go playouts versus the naive playouts ± 2 standard deviations
200	43%±2%
2000	38%±5%
20000	37%±9%

We recall that numbers above are written ± two standard deviations in order to get a confidence interval; we can see that all numbers are below 50%, i.e. the playouts from Go decrease the performance in NoGo; a naive Monte-Carlo is better than a Monte-Carlo from a significantly different game.

A generic improvement of random playouts was proposed in [20]: it consists in playing only moves which do not provide an opportunity of immediate win for the opponent. In the case of Havannah (in [20]) such moves make clearly sense and are quickly computed; for NoGo, we just remove all moves which lead to an immediate loss. The efficiency is very clear:

Nb of simulations per move	Success rate against previous version
100	60%±3
300	61%±3
1000	70%±3
3000	74%±4
10000	79%±5
30000	87%±2
100000	86%±4

This shows the known great efficiency of adapting the playout part; very simple modifications have a great impact, which scales very well as we see that it still works with 100 000 simulations per move.

3.4 Upper Confidence Trees

It is somehow controversial to decide if the UCT parameter C should be 0 or not. We believe that when a new implementation is made, and before tuning, then the parameter C should be > 0, as a first step; but later on, when the implementation

is carefully optimized (and for deterministic games), then C should be set to 0. As our implementation on NoGo is new we have a nice opportunity for testing this:

Constant C	winning rate against $C = 0$ $\pm$ 2× std
NoGo, 200 sims/move	
0	50 %
0.025	52.9±.015
0.1	58.6±.015
0.4	71.4±.014
1.6	**76.0±.014**
6.4	60.5±.028
NoGo, 2000 sims/move	
0	50 %
0	52.0% ±1.5
0.025	60.1% ±1.5
0.1	70.3% ±1.4
0.4	**71.3% ±1.4**
1.6	67.9% ± 1.4
6.4	62.0% ±1.5

Constant C	winning rate against $C = 0$ $\pm$ 2× std
Go, 200 sims/move	
0	**50 %**
0.025	49.2% ±.015
0.1	34.2% ±.015
0.4	11.9% ± .010
Go, 2000 sims/move	
0	**50 %**
0	49.5% ±1.5
0.025	49.3% ±1.5
0.1	29.0%±1.9
0.4	4.3% ± 1.0

We clearly see that in Go the UCT-guided exploration can not help the algorithm (which is a Monte-Carlo Tree Search with RAVE values[10] and patterns [8, 7]). On the other hand, it is helpful in NoGo, in spite of the presence of RAVE values. However, a second case in which $C > 0$ is usefull is, from our experience, cases in which there is a random part in the game, either due to stochasticity in the game or due to stochasticity in the Nash strategies. Exploring this is left as further work.

3.5 A generic Monte-Carlo algorithm for normal form games

NoGo is a normal form game, i.e. a game in which the first player with no legal move looses the game. It is known that MCTS can be greatly improved by modifying the Monte-Carlo part (the generator of random moves, for previously unseen situations). We propose the following heuristic, which can be applied for all normal form games:

- for each location, compute, if you play in it:
  - the number a of removed legal moves for your opponent;
  - the number b of removed legal moves for you.

  and compute, if your opponent plays in it:
  - the number a' of removed legal moves for your opponent;
  - the number b' of removed legal moves for you.
- choose randomly (uniformly) a move with maximum score $a - b - a' + b'$.

We present here the performance of the simplified $a - b$ formula (see Fig. 1), which seemingly performs well:

Number of simulations per move	success rate against the naive case
10	51.9 % ± 7.9 %
20	54.7% ± 6.5 %
40	56.6% ± 5.7%
80	67.7% ± 5.4 %
160	77.7% ± 4.9%
320	83.3% ± 5.3%
640	84.5% ± 5.1%

The improvement is just huge. This is for a fixed number of simulations per move, as we did not implement it for being fast, but the implementation, if carefully made, should have a minor computational cost.

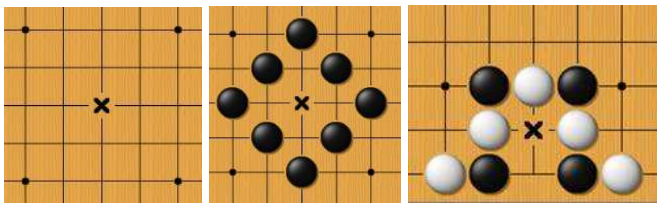

Fig. 1. Left: standard case, $a - b = 0$. Middle, best case for black: $a - b = 4$. Right, worst case for black: $a - b = -4$.

3.6 Scalability analysis

We first tested the efficiency of 2N simulations per move against N simulations per move. We get Table 1 for 7x7 NoGo (divisions show the exact numbers of wins/losses):

Without RAVE	
N=50	138/(138+65)=.679
N=500	251/(251+160)=.610
N=5000	91/(91+63)=.590
N=50000	24/(24+22)=.521
With RAVE	
N=50	1163/(1163+1279)=.476
N=500	198/(198+135)=.594
N=5000	176/(176+119)=.596
N=50000	60/(60+59)=.504
With RAVE and anti-decisive moves	
N=50	155/(155+144)=.518
N=500	219/(219+81)=.730
N=5000	186/(186+114)=.620
N=50000	117/(117+82)=.587

2N sims without RAVE against N sims with RAVE	
N=50	1002/(1002+912)=.523
N=500	294/(294+258)=.532
N=5000	26/(26+69)=.273
2N sims with RAVE against N sims without RAVE	
N=50	216/(216+127)=.629
N=500	113/(113+47)=.706
N=5000	39/(39+5)=.886
N=50000	13/(13+3)=.812
2N sims +RAVE+UCT against N sims+RAVE+UCT	
N=50	117/(117+181)=.393
N=500	234/(234+66)=.780
N=5000	175/(175+125)=.583
N=50000	36/(36+30)=.546

Table 1. MCTS results on 7x7 NoGo.

The UCT parametrization is the one discussed in section 3.4. We tested many variants in order to assess clearly (i) a decreasing scalability when N increases (but, with the best version including anti-decisive moves, the scaling is still good at 50 000 simulations per move) (ii) a clearly good efficiency of RAVE values (which is much better than multiplying the number of simulations per 3). This contradicts the idea, often mentioned in the early times of Monte-Carlo Tree Search, that the success rate of "2N vs N" is constant; but it is consistent with more recent works on this question[3].

Also, we point out that RAVE is not efficient for very small numbers of simulations; maybe this could be corrected by a specific tuning, but 50 simulations per move is not the interesting framework.

4 Conclusion

NoGo is surprisingly simple (in terms of rules) and deep. The classical MCTS tricks (evaluation by playouts, rapid-action value estimates, anti-decisive moves, slow node creation) were efficient in this new setting as well as in Go and in other tested games. We point out that the "slow node creation", not often cited, is in fact almost necessary for avoiding memory troubles, on fast implementations or computers with small memory. We have also seen that the upper confidence term could have a non-zero constant in the new game of NoGo, whereas it is useless in highly optimized programs for Go. An interesting point is that, as in other games, we get a plateau in the scalability; importantly, the plateau is roughly at the same number of simulations per move with and without Rave, but the strength at the plateau is much better with RAVE.

A somehow disappointing point is that tweaking the Monte-Carlo part is more efficient than any other modification; this is also consistent with the game of Go[21]. However, please note that the "tweaking" here is somehow general as it involves a general principle, i.e. avoiding immediate loss (for the anti-decisive moves) and maximizing the improvement in terms of legal moves (for the heuristic value for normal form game).

References

1. B. Arneson, R. Hayward, and P. Henderson. Mohex wins hex tournament. *ICGA journal*, pages 114–116, 2009.
2. P. Auer. Using confidence bounds for exploitation-exploration trade-offs. *The Journal of Machine Learning Research*, 3:397–422, 2003.
3. A. Bourki, G. Chaslot, M. Coulm, V. Danjean, H. Doghmen, J.-B. Hoock, T. Hérault, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssière, and Z. Yu. Scalability and parallelization of monte-carlo tree search. In *Proceedings of Advances in Computer Games 13*, 2010.
4. B. Bouzy and T. Cazenave. Computer go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.
5. B. Bruegmann. Monte-carlo Go (unpublished draft http://www.althofer.de/bruegmann-montecarlogo.pdf). 1993.
6. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.
7. G. Chaslot, M. Winands, J. Uiterwijk, H. van den Herik, and B. Bouzy. Progressive Strategies for Monte-Carlo Tree Search. In P. Wang et al., editors, *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661. World Scientific Publishing Co. Pte. Ltd., 2007.
8. R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, pages 72–83, 2006.

9. 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.
10. 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, New York, NY, USA, 2007. ACM Press.
11. L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.
12. T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22, 1985.
13. C.-S. Lee, M.-H. Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, 2009.
14. H. Nakhost and M. Müller. Monte-carlo exploration for deterministic planning. In *IJCAI*, pages 1766–1771, 2009.
15. N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.
16. J. M. Robson. The complexity of go. In *IFIP Congress*, pages 413–417, 1983.
17. P. Rolet, M. Sebag, and O. Teytaud. Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the ECML conference*, 2009.
18. S. Sharma, Z. Kobti, and S. Goodwin. Knowledge generation for improving simulations in uct for general game playing. In *AI '08: Proceedings of the 21st Australasian Joint Conference on Artificial Intelligence*, pages 49–55, Berlin, Heidelberg, 2008. Springer-Verlag.
19. F. Teytaud and O. Teytaud. Creating an Upper-Confidence-Tree program for Havannah. In *ACG 12*, Pamplona Espagne, 2009.
20. F. Teytaud and O. Teytaud. On the huge benefit of decisive moves in monte-carlo tree search. In *Proceedings of the IEEE conference on Computational Intelligence in Games*, 2010.
21. Y. Wang and S. Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, pages 175–182, 2007.

Algorithm 1 The UCT algorithm in short.

Bandit applied in state s with parameter $C > 0$
Input: a state s.
Output: an action.
Let $nbVisits(s) \leftarrow nbVisits(s) + 1$
  and let $t = nbVisits(s)$
Choose an option $o_{(t)}(s) \in L(s)\}$ maximizing $score_t(s, o)$ as follows:
  $totalReward_t(s, o) = \sum_{1 \leq l \leq t-1, o_l(s)=o} r_l(s)$
  $nb_t(s, o) = \sum_{1 \leq l \leq t-1, o_l(s)=o} 1$
  $score_t(s, o) = \frac{totalReward_t(s,o)}{nb_t(s,o)+1} + C\sqrt{\log(t+1)/(nb_t(s,o)+1)}$
    ($+\infty$ if $nb_t(o) = 0$)
Test it: get a state s'.

UCT algorithm.
Input: a state S, a time budget.
Output: an action a.
Initialize: $\forall s, nbSims(s) = 0$
while Time not elapsed **do**
  // starting a simulation.
  $s = S$.
  while s is not a terminal state **do**
    Apply the bandit algorithm in state s for choosing an option o.
    Let s' be the state reached from s when choosing action o.
    $s = s'$
  end while
  // the simulation is over; it started at S and reached a final state.
  Get a reward $r = Reward(s)$ // s is a final state, it has a reward.
  For all states s in the simulation above, let $r_{nbVisits(s)}(s) = r$.
end while
Return the action which was simulated most often from S.